An Example of Program Understanding

Spencer Rugaber
December, 1997

Introduction

What does it mean to understand a program? What sorts of questions can be answered about a program? What background knowledge is required to answer them? What tools can help the process? To answer questions like these, we will look at an example of program understanding in action. Imagine the following scenario: You are assigned responsibility for maintaining a program you have never seen before. It is written in the FORTRAN language and is concerned with finding the roots of function. We will assume that you know the FORTRAN language but are not an expert at it. That is, you still have to occasionally look at the reference manual to answer questions about the language. We will also assume that you have a computer science background, either through formal education or by on-the-job experience, so that you know how to design and compose programs similar to the one you are about to maintain. Finally, we will assume that you have a passing acquaintance with numerical analysis, possibly from a course you took. Hence, you are familiar with the idea of finding a root of a function, but you would have to look up an actual algorithm in order to write a root-finding program yourself.

As you are responsible for long-term maintenance of the program, you want to understand it better. So you decide to systematically read it. This is distinct from the situation where you have a specific task to accomplish, such as finding a bug, adding a new feature, or updating the program to conform to a change in the language or operating environment. In these cases, instead of systematic reading, you might direct your efforts to accomplishing the specific task. Here, we will assume that you are going to make a single, sequential pass through the program text, with perhaps a few side trips to answer small questions as they arise.

The following program text is taken directly from its source [1]. We have added three digits of line numbers in the left margin for expository purposes; they are not part of the program itself. We will annotate each line in the program as we come to it. The idea is to express some idea of what that line is there for. Sometimes we will raise questions, sometimes we will generate hypotheses that need to be confirmed, and sometimes we will speculate about the application domain. The intent is to get a feel for the kinds of knowledge required to understand a program on a line-by-line basis.

Understanding ZEROIN

Before we even read the first line, we can tell some things about this program. First of all, it is written in FORTRAN. Even if we had not been told this, the use of upper case and the rigorously aligned columns would have told us. And the variant of FORTRAN is old, probably FORTRAN-66. We can tell this from the absence of more modern control structures, such as **if-then-else**. Another global observation is that the programmer has used comments to break the program text into blocks or paragraphs. Presumably, each such collection of statements corresponds to a concept we would like to understand about the program. In what follows, lines from the program are displayed using Courier font and bold highlighting.

```
  1       REAL FUNCTION ZEROIN (AX,BX,F,TOL,IP)
```

The first line is a function declaration informing us of the name, return type, and formal parameters of the function. The term "**ZEROIN**" sounds mathematical and conforms to our preconception of "zeroing-in on" the root of a function. There are five arguments. **TOL** might be short for "tolerance," that is, a measure of how close we need to get to the solution. **AX** and **BX** have similar names. We will be on the lookout for code in which they are treated similarly. **F** is sometimes used in FORTRAN as a functional parameter, that is, a name which can be used to invoke a function passed in as an argument. The fifth argument is **IP** about which we get no hints from the declaration.

Note that in making the observations in the preceding paragraph, we called upon our knowledge of FORTRAN, of root-finding, and of how programs are written using FORTRAN.

```
2       REAL AX,BX,F,TOL
```

All of the arguments except **IP** are declared to be **REAL**. **IP** is left as an **INTEGER** using FORTRAN's default declaration rules.

```
3 C
4 C
```

The author is using empty comments to improve the readability of the program.

```
5       REAL A, B, C, D, E, EPS, FA, FB, FC, TOL1, XM, P, Q, R, S
```

Here are the declarations of more variables. We can make the following guesses about their intent from their names and the similarities among them. **EPS** might be short for *epsilon*, a term often used in mathematics to denote a very small real number. **A, B, C, D, E** are named similarly. We should look for similarities in the pattern of their use. Likewise, **FA, FB, FC** are named similarly. It may also be the case that whatever similarity exists between **A**, **B**, and **C**, there exists a corresponding similarity among **FA**, **FB**, and **FC**. **TOL1** might be another tolerance. **P, Q, R, S** are named; similarly. *Are there are other variables used in the program that are not explicitly declared?*

```
6 C
7 C COMPUTER EPS, THE RELATIVE MACHINE PRECISION
8 C
```

Here is the start of the first program block. The programmer has used three consecutive comments, the first and the last empty, to indicate the start and purpose of the block. We will call this a paragraph comment and use the abbreviation **PC(9-12)** to indicate that the corresponding paragraph extends from lines **9** through **12**. The comment on the middle line uses the term "**RELATIVE MACHINE PRECISION**." *Is this a term that we should already know, or is it something that the programmer has made up for the purposes of this program only?*

```
9       EPS = 1.0
```

Here **EPS** is initialized.

```
10    10 EPS = EPS/2.0
```

The next statement is labeled. Whenever we come across a labeled statement, several questions come immediately to mind. *Is it used for flow of control or is it a **FORMAT** statement? If the former, what statements refer to (can branch to) this label? Under what circumstances can those branches be taken? That is, what can we assume is true after such a branch has been taken?* We will denote such statements with an **L** to indicate that these questions ultimately must be answered in order to understand the execution of the labeled statement.

The actual statement being executed is a floating division by **2.0**. We can interpret this to mean that we want to cut the value of **EPS** in half.

```
11      TOL1 = 1.0 + EPS
```

*Are we trying to compute a value for **TOL1** or for **EPS**? Which of these will be referred to below and which is just a temporary variable?*

```
12       IF (TOL1 .GT. 1.0) GO TO 10
```

Here is a branch back to **10**. In order to understand the branch condition, we can insert the computation for **TOL1** into the condition on line **12**. We can think of this as a form of *symbolic evaluation*. In fact, we could make a permanent substitution of **1.0 + EPS** for **TOL1**. But to do this, we need to check whether the value of **TOL1** is required later in the program, or just the value of **EPS**.

These three lines actually implement a **do-while** loop such as is found in the C language. This version of FORTRAN evidently does not support this construct. If we do the symbolic evaluation and the syntactic change, we get the following source statements.

```
 9  EPS = 1.0
10  DO
11      EPS = EPS / 2.0
12  WHILE (1.0 + EPS .GT. 1.0)
```

We would like to further simplify line **12** to be **WHILE (EPS .GT. 0.0)** using the laws of arithmetic. But this may not be allowed due to the way that normalized floating point arithmetic works.

{Opportunity for side bar on floating point normalization}

To summarize, after the loop has terminated, **EPS** is the largest floating point number which, when added to **1.0**, still yields **1.0**. *Is this what the author means by "**RELATIVE MACHINE PRECISION**"?*

Are we even sure that the loop terminates? Might there not be a "smallest floating point number" which when divided by two and the results rounded upward, yields itself. Only a thorough understanding of floating point arithmetic and the hardware of the machine on which the program is run can answer the question.

The code in **9-12** makes no reference to the input parameters. This raises the question, *should the value of **EPS** be computed external to **ZEROIN** and passed in as a parameter to avoid repeated recomputation? Or should it be computed at compile time and available in some form of environmental constant such as is provided in the ANSI C language?*

```
13 C
14 C INITIALIZATION
15 C
```

PC (16-21). This flavor of comment suggests that the program is structured functionally as opposed to being built around some sort of abstract data type.

```
16       IF (IP .EQ. 1) WRITE (6,11)
```

IP is an input parameter. *Is its value ever set in **ZEROIN**? Or can it be treated as a read-only parameter?* What *other values can **IP** hold? Is it being used as a flag? Why does the program perform output inside of a function?* Knowledge of how FORTRAN works tells us that **6** is the standard output. *Could this statement be doing debugging printout?*

```
17    11 FORMAT('THE INTERVALS DETERMINED BY ZEROIN ARE')
```

L. The statement indicates that the output will be a constant string. It also introduces another term from the application domain: "**INTERVAL**".

```
 18        A = AX
```

Here the input parameter **AX** is being referred to and saved away in a variable named **A**. *Are there any other references to **AX**?* Is the value of **AX** ever changed?

```
 19        B = BX
```

Here something similar to what happened to **A** in line **18** is being done for **B**. Note that our observation about the similarity of **AX** and **BX** is reinforced by the last two lines.

```
 20        FA = F(A)
```

Our guess about the purpose of **F** is confirmed. It is a function parameter, with the actual function to invoke being determined where and when **ZEROIN** is called. The name **FA** suggests that **FA** is a surrogate for the call to **F(A)**, perhaps introduced to save what might be a costly reexecution. In fact, we can hypothesize an invariant condition: **FA** shadows the value of **F(A)** and whenever **A** changes, **F(A)** should be called and assigned to **FA**. To check this requires scanning the entire program to see if there is ever any violation.

```
 21        FB = F(B)
```

The same thing is done for **FB.**

```
 22 C
 23 C BEGIN STEP
 24 C
```

PC (25-37). *Does "**BEGIN**" mean "**FIRST**"?*

```
 25    20 C = A
```

L. Remember whatever was in **A.** For the first time through, anyway, **A** is the same as **AX**.

```
 26        FC = FA
```

*Is **FC** a shadow variable for **F(C)** like **FA** is for **A**?*

```
 27        D = B - A
```

*Could **D** stand for "distance"?* After all, if we are beginning to think of **A** and **B** as denoting the boundaries of an interval, then their difference corresponds to the (signed) distance between them.

```
 28        E = D
```

Why are there two copies of the same value? In a well-written program, there must be some point where the value of **E** and **D** differ. We want to understand what the special case is.

```
 29    30 IF (IP .EQ. 1) WRITE (6,31) B, C
```

L. Here is another conditional predicated on the input parameter **IP**. It requests the printing of the values **B** and **C**: an interval determined by **ZEROIN**. So we should adjust our earlier assumption (line **27**) that the interval is defined by **A** and **B**. It looks like we should use **B** and **C** instead. And we should think of **A** shadowing **C** instead of vice verse. This latter observation suggests a minor programming anomaly. Whereas **AX** and **BX** were supplied as

the original interval definition, which suggests that **A** and **B** hold the interval endpoints, the programmer is actually using **B** and **C** for this purpose.

```
30    31 FORMAT (2E15.8)
```

L. The **E** format is being used to print two real numbers. Note that if B and C are large, the print out might not provide any blank space between their values.

```
31        IF (ABS(FC) .GE. ABS(FB)) GO TO 40
```

A quick scan downward confirms that the next few lines of code will be skipped if the magnitude of **F(C) .GT. ABS(F(B))**. This implies that in the next few statements, we can assume that **|F(C)| .LT. |F(B)|.**

```
32        A = B
33        B = C
34        C = A
```

These three lines are an example of a programming *idiom*, a stylized fragment of code that collectively accomplishes some purpose beyond that provided by the individual lines. In this case, the purpose is to swap the values of **B** and **C** while retaining the fact that **A** shadows **C**.

```
35        FA = FB
36        FB = FC
37        FC = FA
```

Here is the idiom again, this time for **FA**, **FB**, and **FC**. Note that the three lines also maintain the fact that **FA** retains the value of **F(A), FB** for **F(B),** and **FC** for **F(C).**

```
38 C
39 C CONVERGENCE TEST
40 C
```

PC (41-44). "**CONVERGENCE**" means coming closer (*to a solution?*). We should expect a convergence test, a loop, and the computation of value near **0.0**.

```
41    40 TOL1 = 2.0*EPS*ABS(B) + 0.5*TOL
```

L. **40** is the label referred to in line **31**. If looks like that line was really implementing a compound **if** statement. Moreover, its purpose was to establish the fact that **FC** is greater than **FB**. So not only do we know that **B** and **C** define the interval within which the root is to be found, but henceforth, we can assume that the value at point **C** is greater in magnitude than the value at point **B**.

{Opportunity for a diagram.}

The **TOL1** variable name is being recycled here. That is, it held a value, but the value is no longer of any use. *How can we confirm this?* We have to make sure that no path from the previous definition can reach any reference to **TOL1** without going through the current statement.

TOL is an input parameter. It has not been altered by any statement on a direct path from the start of the program to this line. But what is the meaning of **TOL1?** If we assume that **TOL** is some form of tolerance, **EPS** is the machine precision, and **B** is one of the endpoints of the current interval, then **TOL1** defines some form of neighborhood or region around **B**.

```
42        XM = .5*(C-B)
```

If **C** and **B** define the current interval, then **C-B** is the interval length, and **XM** is the distance to the midpoint. *Could the name **XM** stand for the X value of midpoint of interval, actually the distance to the midpoint?*

```
43        IF (ABS(XM) .LE. TOL1) GO TO 90
```

Here is a candidate for the termination test of the convergence loop. *Could line **90** be the function exit?* The two values being compared correspond to (one half of) the length of the interval (**XM**) and the size of the region (**TOL1**). When the interval gets small enough **(.LE. TOL1)** then it doesn't make sense to continue shrinking it.

```
44        IF (FB .EQ. 0.0) GO TO 90
```

If our hypothesis is correct about **FB** being always the value of **F(B)**, then another reason for quitting is that **FB** is **0.0**; that is, that we have found a root. *But why do this check in a separate statement*? *Why not just have one compound statement with an .**AND.** ?*

```
45 C
46 C IS BISECTION NECESSARY
47 C
```

PC (48-49). "**BISECTION**" is a method of root finding. Actually, it is a way of finding an intermediary point so the interval can be shrunk. Bisection would naturally make use of the midpoint (**XM**).

```
48        IF (ABS(E) .LT. TOL1) GO TO 70
```

Recalling that based on the top-down reading, **E** is nothing more than the size of the interval, then perhaps we are deciding on the suitability of bisection based on the size of the interval. More than ever, we need to know how **E** is intended to differ from **D**. *How does this comparison differ from that on line **43**?* Plugging in the values that we know about, we get **E == D == B - A**, which yields **ABS(E) == 2.*ABS(XM)**. But if **ABS(XM) .GT. TOL1** (by line **43**) then **2.0*ABS(XM)** will also be **.GT. TOL1**. Hence, this statement should never executes (at least on the first path through the code). So we need to be on the lookout for where either **E** or **TOL1** is set somewhere later in the program.

```
49        IF (ABS(FA) .LE. ABS(FB)) GO TO 70
```

These two lines (**48-49**) have the same property that lines **43-44** had: two consecutive tests, both branching to the same place. Hence, this is another candidate for the use of **.AND.**.
Moreover, **ABS(FA)** can't be less than **ABS(FB)** (based on line **31**) unless things have changed on another path to this line.

```
50 C
51 C IS QUADRATIC INTERPOLATION POSSIBLE
52 C
```

PC (53). *"**QUADRATIC INTERPOLATION**"—could this be another way of shrinking the interval?*

```
53        IF (A .NE. C) GO TO 50
```

This cannot be the case if the only path is the one from the top. *So the question arises as to what conditions would lead **A** and **C** to have different values?*

```
54 C
```

```
 55 C LINEAR INTERPOLATION
 56 C
```

PC (57-60). Another domain term is "**LINEAR INTERPOLATION**". The term suggests that the program should contain an intermediate point proportionate to the relative heights of **F(A)** and **F(B).** *Under what conditions can control reach this spot? How do* ***LINEAR INTERPOLATION****,* ***BISECTION*** *and* ***QUADRATIC INTERPOLATION*** *relate to each other in the code?*

```
 57       S = FB/FA
```

Compute the proportion and save it in **S**.

```
 58       P = 2.0*XM*S
```

XM was half the interval length. Hence, **2.0*XM** is the interval length. **2.0*XM*S** is that part of the interval which we will shrink.

{Sidebar on linear interpolation.}

```
 59       Q = 1.0 - S
```

1.0 - S is the remaining portion of the interval.

```
 60       GO TO 60
```

What do we do with these values that we have computed?

```
 61 C
 62 C INVERSE QUADRATIC INTERPOLATION
 63 C
```

PC (64-68). The earlier comment (line **51**) for the statement that lead to the branch here, didn't say anything about "**INVERSE**".

```
 64    50 Q = FA/FC
```

L. Compute the ratio of the functional values at the boundaries. *Is this* ***Q*** *analogous to the* ***Q*** *on line* ***59****?*

```
 65       R = FB/FC
```

Compute another ratio into **R.**

```
 66       S = FB/FA
```

Identical statement to line **57**. It looks like an interval proportion similar to what was used above.

```
 67       P = S*(2.0*XM*Q*(Q-R) - (B-A) * (R-1.0))
```

Complicated formula for **P**, although there is some similarity to line **58.**

```
 68       Q = (Q-1.0)*(R-1.0)*(S-1.0)
```

And another complicated formula for **Q**, using the ratios computed on lines **64-66**. Note that Q is defined in terms of itself. Actually, a different variable name could be used for this (or the previous) **Q**. *What sort of similarity is there between **57-59** and **64-68**?* Are the conditions under which they execute simply differentiated? *Do they produce output values that have similar signatures?* (A signature includes the number and types of inputs and results.) If they do, then we can think of the similar sections as being special cases of some more general computation.

```
 69 C
 70 C ADJUST SIGNS
 71 C
```

PC (72-73). *Why do we need to adjust the signs?* If we work under the assumption that there are several special cases of some more general computation, then we may have to make sure that all of the special cases have the same form. For example, that the endpoints of the interval have their signs ordered in a specific way.

```
 72    60 IF (P .GT. 0.0) Q = -Q
```

L. There are two ways to get here, either falling through or branching. *What are the circumstances for each? Why do we need to flip the value of **Q**?*

```
 73       P = ABS(P)
```

Make sure **P** is positive. We would like to know the "meaning" of each program variable in terms of its role in the application domain. An approximation to this is the ability to express it in terms of the input parameters.

```
 74 C
 75 C IS INTERPOLATION ACCEPTABLE
 76 C
```

PC (77-81). *What is an "**ACCEPTABLE INTERPOLATION**"?*

```
 77       IF ((2.0*P) .GE. (3.0*XM*Q - ABS(TOL1*Q))) GO TO 70
```

A complicated formula telling us whether or not to skip the rest of the paragraph.

```
 78       IF (P .GE. ABS(0.5*E*Q)) GO TO 70
```

Another possible reason for skipping. As there are no side effects, *why not just combine these two conditions with an **.AND.**?* One possible reason is to guarantee order of evaluation; that is, to make sure that the first condition gets checked before the second. If the language definition allows, some compilers might take the liberty to reorder the evaluations of these two conditionals. This is a language semantics question. Another possible reason is that each of the two conditions have its own relevance to the interpretation of the underlying algorithm. That is, it makes the program more readable to see the two conditions on separate lines.

```
 79       E = D
```

Remember **D** in **E**. We still have yet to see how **D** could ever be different from **E** (remember the question raised by line **48**).

```
 80       D = P/Q
```

And here is where **D** can be set to something (presumably) different from **E**.

```
 81       GO TO 80
```

Skip around the bisection. There are no references to **P, Q, R,** or **S** below. *What are their regions of influence (the regions in the code in which their values have significance)?*

```
 82 C
 83 C BISECTION
 84 C
```

PC (85-86). What we deferred from above (lines **77** and **78**).

```
 85    70 D = XM
```

L. With bisection, we want to use the midpoint as one of the interval boundaries.

```
 86       E = D
```

Set **E** from **D**. Sounds like what we just did in **79-80**. *What relationship have we just established?*

```
 87 C
 88 C COMPLETE STEP
 89 C
```

PC (90-96). "**COMPLETE**" sounds similar to **BEGIN.**

```
 90    80 A = B
```

L. We still don't fully understand the role of **A**. But it looks we are getting ready to actually do the shrinking of the interval.

```
 91       FA = FB
```

Preserves the **FA == F(A)** invariant.

```
 92       IF (ABS(D) .GT. TOL1) B = B + D
```

Conditionally update **B**. So **D** is the amount to change **B**. I suppose we have to worry about signs on **B** and **D** to make sure the new interval boundaries are correct.

```
 93       IF (ABS(D) .LE. TOL1) B = B + SIGN(TOL1,XM)
```

But this is the negation of the condition tested in **92**. *Does that make this line amount to the **else** branch of an **if-then-else** statement? What is the **SIGN** function*? This is a FORTRAN language question. *What is the significance of its two arguments?*

```
 94       FB = F(B)
```

This satisfies the obligation that **FB == F(B)**.

```
 95       IF ((FB*(FC/ABS (FC))) .GT. 0.0) GO TO 20
```

Here is another programming idiom. The complicated looking condition. **FC/ABS(FC)** must be == +/-1. That is, **FC/ABS(FC)** maps to +1 if **FC .GT. 0** and to -1 otherwise. Hence, **FB*(FC/ABS(FC))** must == +/-**FB**.

But we are only checking whether the whole value is **.GT. 0.0**. So the condition succeeds if either **FB .GT. 0.0** and **FC .GT. 0.0** or if **FB .LT. 0.0** and **FC .LT. 0.0**. That is, we are checking if they have the same signs. This is an expensive (and obscure) way of doing this involving a multiply, a divide, a function call, and a comparison.

20 is the **BEGIN** step where **C, D,** and **E** got set up.

```
 96       GO TO 30
```

I guess we don't need to do the **BEGIN** step if the signs are different.

```
 97 C
 98 C DONE
 99 C

PC (100-102).

100    90 ZEROIN = B
```

L. FORTRAN returns a value by using the name of the function as if it were a program variable. The function is returning the value is **B**, the interval boundary whose **F** value is smallest.

```
101       RETURN
```

This is the only place to exit the function.

```
102       END
```

Discussion

We have learned much during our one-pass, top-down reading of the program. We have also built some hypotheses that need to be confirmed through further analysis. And there are some questions that require further knowledge of the application domain to answer. Our new-found knowledge concerns not only the purpose or intent of the program (to solve some application-domain problem) but also is about programming and, more specifically, programming in FORTRAN.

Concerning the application domain, we have learned that the program is intended for finding the root (domain value whose corresponding functional value is **0.0**) of a function (**F**) that is passed in as a parameter. The root is located in an interval (the segment of the real line between two points) initially defined by two other parameters (**AX** and **BX**). The value returned has one of two attributes: either the corresponding functional value is **0.0** or the root is found in a surrounding interval that is too small to be effectively reduced any further. We have also learned that the size of the surrounding interval is controlled by a fourth parameter (**TOL**). And there is a fifth parameter, **IP**, which is used only to control whether certain values are written to file descriptor **6**.

We have conjectured that the algorithm proceeds by successively shrinking the interval while maintaining certain relationships (invariants) among the program variables. Various methods for shrinking the interval appear to be in use including **BISECTION**, **LINEAR INTERPOLATION**, and **INVERSE QUADRATIC INTERPOLATION**. We also conjecture that the shrinking process is controlled by the sign of the functional values of the end points of the interval.

We have also learned some FORTRAN programming idioms. Lines **9-12** computed a value called **THE RELATIVE MACHINE PRECISION** by repeatedly halving a value and adding it to **1.0** until the value got so small that adding it had no effect. We also saw a way (on line **95**) where we could check whether two values have

the same sign. We surmised that both of these tricks were inefficient, and we should look for ways to improve them.

We saw several places where this version of FORTRAN's limited control structures were used to simulate more structured constructs that are available in more recent versions of the language. Among these were several **if-then-else** constructs and a **do-while** loop.

There are still some mysteries concerning the program. We saw several places (lines **67-68** and **77-78**) where complicated expressions are computed whose purposes we can begin to guess. But more knowledge of the application domain and its various solution techniques (algorithms) is required before we can definitively annotate these computations.

Furthermore, there are also several places in the program where we need to know about certain global program properties. For example, we want to know all the places where a variable is used, all the paths that could lead to a certain statement and the conditions under which those paths can be taken, and whether two sections of code are similar in the sense that they both were responsible for setting certain variables based on certain input parameters. More systematic analysis requiring multiple passes is required to answer these questions.

Alternative Approaches

A single-pass, top-down approach is not the only way to read a program. One alternative is to examine the program bottom-up, building up a description of overall program functionality from the computations performed on the specific lines. This can even be done formally, with a mathematical description being given to the computations. Basili and Mills [2] have done this for the **ZEROIN** program, and we summarize their approach in the next subsection.

Another possibility is to combine a top-down and a bottom-up approach. The bottom-up approach looks at the mechanics of each statement from the point of view of design decisions made by the programmer while a simultaneous, top-down process is building up a description of the problem the program is solving. Rugaber, Ornburn, and LeBlanc have described this approach for **ZEROIN** [3], and we summarize it in the second subsection to follow.

Basili and Mills

Basili and Mills are interested in applying ideas from structured programming and program correctness proofs to the understanding and annotation of computer programs. In particular, they construct a *prime program decomposition* of the program, define *program functions* for each prime program, build a data reference table describing the uses of each program variable, and then synthesize a program correctness proof demonstrating exactly how the program accomplishes its goals.

A *proper program* is a contiguous program segment with a single entry point, a single exit point, and the property that each statement in the segment is on a path from the entry to the exit (that is, that each statement is actually used during execution). A prime program is a proper program that does not contain any more basic proper program except for the individual program statements. Roughly speaking, a prime program corresponds to a structured control construct. In fact, for the two prime programs in **ZEROIN** that contained more than one predicate, the authors syntactically converted them into structured control constructs.

A useful property of prime programs is that they can be composed. That is, larger program units can be built out of smaller ones while still retaining the single entry, single exit, useful-statement properties. Thus, we can compose the meanings of the prime units in the same ways that we can compose them syntactically.

Program functions are predicates that describe how a statement or larger program segment produces its output values in terms of its inputs. They are abstractions that summarize what a series of statements accomplishes without going into the details of how it is accomplished.

A data reference table is like a cross-reference listing that breaks out references into those that set or define a variable from those that merely access or use it.

Using their approach, the authors are able to gain an understanding of **ZEROIN**. The approach helps them organize their analysis and provides a completeness criterion. That is, they know that they have to keep working until they complete their proof. Hence, it gives them a way of knowing when their understanding is deep enough. It should be noted, however, that like other uses program correctness proofs, the authors' approach to program understanding requires a degree of mathematical sophistication and a certain inventiveness for constructing loop invariant conditions and for recognizing an *indeterminate bounded variable* that they used to summarize the program's progress.

Rugaber, Ornburn and LeBlanc

Rugaber, Ornburn, and LeBlanc view programs as the results of *design decisions* made by a programmer during the course of development. Understanding an existing program therefore requires recognizing and annotating its decisions based on the evidence provided by the source code.

The paper characterizes design decisions based on the kinds of abstractions provided by programming languages, by database modeling techniques, and by transformational programming theory. The decisions belong to one of several categories. A *composition* decision groups lower level constructs and gives them a single name. For example, variables can be grouped into a record structure, or program statements can be grouped into a subprogram. In **ZEROIN**, the programmer decomposed the algorithm into a series of paragraph each described in a comment.

A separate but related decision is whether the internals of the resulting aggregate are visible to its clients. If not, the aggregate is said to be *encapsulated*. In the case of program statements, encapsulation can be enforced by the use of modules, abstract data types, or information hiding. An alternative to encapsulation is *interleaving* where the internals of two or more constructs are intentionally intermixed, usually for reasons of improved efficiency. Perhaps the most difficult aspect of understanding **ZEROIN** is appreciating how it interleaves three different methods for shrinking the interval during each iteration of the main loop.

The three methods for shrinking the loop also illustrate a *specialization* design decision. That is, the three ways of shrinking the interval all have the effect of preparing the program variables for the next iteration. Specialization is even easier to accomplish in object-oriented programming languages where inheritance can be used to quickly define functionality in a subclass that specializes that in its superclass.

A fourth category of design decision is called *representation*. Representation is typically seen when one type of data structure, such as an array, is used to implement another, such as a stack. In **ZEROIN**, an interval is represented by a pair of floating point numbers. Additionally, the programmers who wrote **ZEROIN** made disciplined use of its **GOTO** statement to represent more structured control structures such as the **if-then-else statement**.

Programmers often have to decide whether to compute a value or look it up. In ZEROIN, for example, the variables **FA**, **FB**, and **FC** are used to hold the values of **F(A)**, **F(B)**, and **F(C)**, respectively. These values could have been computed when they were needed, but the programmer decided that a call to **F** might be costly, and looking up the value in a variable could accomplish the same purpose even if it made the code marginally more difficult to understand.

The final category of design decision described by Rugaber, Ornburn, and LeBlanc has to do with functions and relations. **ZEROIN** is a FORTRAN function that takes as input another function, a tolerance, and an interval and produces a root. It is possible to imagine a related function that takes the function, tolerance, and root, and produces the interval. In fact, the Prolog programming language supports this duality by, in many cases, allowing

the same code to be used for computing in both directions. That is, the Prolog program allows the designer to express the relation between the values (interval, tolerance, root, and function) without requiring him or her to specify which values are inputs and which are outputs. While FORTRAN does not permit this degree of generality, specifications are often couched in terms of relations that must be explicitly implemented as function.

After describing the categories of design decisions, Rugaber, Ornburn, and LeBlanc describe many examples of them in the code for **`ZEROIN`**. They suggest that the annotation of decisions such as these made during program development can significantly ease the burden of subsequently maintaining the program.

Conclusions

Understanding even a relatively small program is a complex process. It requires both knowledge and analysis. Knowledge is required of the programming language syntax, semantics, and run-time libraries, of the computing machine and its operating environment, of the application domain including how problems in the domain are typically solved, and of programming in general, how programs are structured, variables used, and correctness guaranteed.

A single pass over a program is insufficient to understand it, even when an expert is doing the reading. Multiple passes can help answer global questions concerning variable use and execution paths. There are a variety of program analysis tools that can help with this process. The following chapters describe them and demonstrate what they can tell us about **`ZEROIN`**.

References

[1] G. Forsythe, M. Malcolm, and M. Moler.
Computer Methods for Mathematical Computations.
Englewood Cliffs, New Jersey: Prentice-Hall, 1977.

[2] Victor R. Basili and Harlan D. Mills.
"Understanding and Documenting Programs."
IEEE Transactions on Software Engineering, SE-8(3):270-283, May, 1982.

[3] Spencer Rugaber, Stephen B. Ornburn, and Richard J. LeBlanc, Jr.
"Recognizing Design Decisions in Programs."
IEEE Software, 7(1):46-54, January, 1990.