Technical Report TR-884 April 1980
AFOSR-77-3181B

UNDERSTANDING AND DOCUMENTING PROGRAMS*

Victor R. Basili and Harlan D. Mills

Department of Computer Science
University of Maryland
College Park, MD 20742

*Research supported in part by the Air Force Office of Scientific Research Grant AFOSR-77-3181B to the University of Maryland.

ABSTRACT

This paper reports on an experiment in trying to understand an unfamiliar program of some complexity and to record the authors' understanding of it. The goal was to simulate a practicing programmer in a program maintenance environment using the techniques of program design adapted to program understanding and documentation; that is, given a program, a specification and correctness proof were developed for the program. The approach points out the value of correctness proof ideas in guiding the discovery process. Toward this end, a variety of techniques were used: direct cognition for smaller parts, discovering and verifying loop invariants for larger program parts, and functions determined by additional analysis for larger program parts. An indeterminate bounded variable was introduced into the program documentation to summarize the effect of several program variables and simplify the proof of correctness.

Acknowledgements

The authors are grateful to Douglas Dunlop for his insightful review of this report and to Claire Bacigaluppi for patiently typing numerous drafts.

UNDERSTANDING AND DOCUMENTING PROGRAMS

I. INTRODUCTION

Understanding programs - We report here on an experiment in trying to understand an unfamiliar program of some complexity and to record our understanding of it. We are as much concerned with recording our understanding as with understanding. Every day programmers are figuring out what existing programs do more or less accurately. But most of this effort is lost, and repeated over and over, because of the difficulty of capturing this understanding on paper. We want to demonstrate that the very techniques of good program design can be adapted to problems of recording hard won understandings about existing programs.

In program design, we advocate the joint development of design and correctness proof, as shown by Dijkstra in (Dahl, Dijkstra, and Hoare) and (Dijkstra) and by (Linger, Mills, and Witt), rather than a posteriori proof development. Nevertheless, we believe that the idea of program correctness provides a comprehensive a posteriori strategy for developing and recording an understanding of an existing program. In fact, we advocate another kind of joint development, this time, of specification and correctness proof. In this way, we have a consistent approach dealing always with three objects; namely, (1) a specification, (2) a program, and (3) a correctness proof. In writing a program, we are given (1) and develop (2) and (3) jointly; in reading a program, we are given (2) and develop (1) and (3) jointly. In either case, we end up with the same harmonious arrangement of (1) and (2) connected by (3) which contains our understanding of the program.

In the experiment at hand, our final understanding exceeded our most optimistic initial expectations, even though we have seen these ideas succeed before. One new insight from this experiment was how little we really had to know about the program to develop a complete understanding and proof of what it does (in contrast to how it does it). Without the correctness proof ideas to guide us, we simply would not have discovered how little we had to know. In fact, we know a great deal more than we have recorded here about how the program works, which we chalk up to the usual dead ends of a difficult discovery process. But the point is, without the focus of a correctness proof, we would still be trying to understand and record a much larger set of logical facts about the program than is necessary to understand precisely what it does.

In retrospect, we used a variety of discovery techniques. For simpler parts of the program, we used direct cognition. In small complex looping parts, we discovered and verified loop invariants. In the large, we organized the effect of major program parts as functions to be determined by additional analysis. We also discovered a new way to express the effect of a complex program part by introducing a bounded indeterminate variable which radically simplified the proof of correctness of the program part.

The experiment - We were interested in a short but complex program using real arithmetic, and felt that more attention might be paid to the structure and correctness of programs that deal with real arithmetic. The program was chosen by Professor James Vandergraft of the University of Maryland as a difficult program to understand. It was a FORTRAN program called ZEROIN which claimed to find a zero of a function given by a FORTRAN subroutine.

Our goal was to simulate a practicing programmer in a program maintenance environment. We were given the program and told its general function. The problem then was to understand it, verify its correctness, and possibly modify it, to make it more efficient or extend its applicability. We were not given any more about the program than the program itself. The program given to us is shown in Figure 1. Professor Vandergraft played the role of a user of the program and posed four questions regarding the program:

1. I have a lot of equations, some of which might be linear. Should I test for linearity and then solve the equation directly, or just call ZEROIN? That is, how much work does ZEROIN do to find a root of a linear function?
2. What will happen if I call ZEROIN with FA and FB both positive? How should the code be changed to test for this condition?
3. It is claimed that the inverse quadratic interpolation saves only .5 function evaluations on the average. To get a shorter program, I would like to remove the inverse quadratic interpolation part of the code. Can this be done easily? How?
4. Will ZEROIN find a triple root?

It should be noted that the authors are not currently working in the area of numerical analysis, though it is not an unknown area to them.

```
*****  ZEROIN.PROGRAM  *****

      REAL FUNCTION ZEROIN(AX, BX, F, TOL,IP)
      REAL AX, BX, F, TOL
C
C
      REAL A, B, C, D, E, EPS, FA, FB, FC, TOL1, XM, P, Q, R, S
C
C  COMPUTE EPS, THE RELATIVE MACHINE PRECISION
C
      EPS = 1.0
   10 EPS = EPS/2.0
      TOL1 = 1.0 + EPS
      IF (TOL1 .GT. 1.0 ) GO TO 10
C
C  INITIALIZATION
C
       IF (IP .EQ. 1) WRITE(6,11)
 11    FORMAT(' THE INTERVALS DETERMINED BY ZEROIN ARE')
      A = AX
      B = BX
      FA = F(A)
      FB = F(B)
C
C  BEGIN STEP
C
   20 C = A
      FC = FA
      D = B - A
      E = D
 30    IF (IP .EQ. 1) WRITE(6,31) B,C
 31    FORMAT (2E15.8)
      IF ( ABS(FC) .GE. ABS(FB) ) GO TO 40
      A = B
      B = C
      C = A
      FA = FB
      FB = FC
      FC = FA
C
C  CONVERGENCE TEST
C
   40 TOL1 = 2.0*EPS*ABS(B) + 0.5*TOL
      XM = .5*(C - B)
      IF (ABS(XM) .LE. TOL1 ) GO TO 90
      IF (FB .EQ. 0.0 ) GO TO 90
C
C  IS BISECTION NECESSARY
C
      IF (ABS(E) .LT. TOL1) GO TO 70
      IF(ABS(FA) .LE. ABS(FB) ) GO TO 70
C
C  IS QUADRATIC INTERPOLATION POSSIBLE
C
      IF (A .NE.  C) GO TO 50
C
C  LINEAR INTERPOLATION
C
      S = FB/FA
      P = 2.0*XM*S
      Q = 1.0 - S
      GO TO 60
C
C  INVERSE QUADRATIC INTERPOLATION
C
   50 Q = FA/FC
      R = FB/FC
      S = FB/FA
      P = S*(2.0*XM*Q*(Q - R) - (B - A)*(R - 1.0))
      Q = (Q - 1.0)*(R - 1.0)*(S - 1.0)
C
C  ADJUST SIGNS
C
   60 IF (P .GT. 0.0 ) Q = -Q
      P = ABS(P)
```

<u>Figure 1</u>. (Page 1)

```
C
C   IS INTERPOLATION ACCEPTABLE
C
      IF ((2.0*P) .GE. (3.0*XM*Q - ABS(TOL1*Q))) GO TO 70
      IF (P .GE. ABS(0.5*E*Q)) GO TO 70
      E = D
      D = P/Q
      GO TO 80
C
C   BISECTION
C
   70 D = XM
      E = D
C
C   COMPLETE STEP
C
   80 A = B
      FA = FB
      IF (ABS(D) .GT. TOL1) B = B + D
      IF (ABS(D) .LE. TOL1) B = B + SIGN(TOL1,XM)
      FB = F(B)
      IF ((FB*(FC/ABS(FC))) .GT. 0.0 ) GO TO 20
      GO TO 30
C
C   DONE
C
   90 ZEROIN = B
      RETURN
      END
```

```
*****  ZEROIN.INFO  *****

      ZEROIN IS A FUNCTION SUBPROGRAM WHICH FINDS
      A ZERO OF THE FUNCTION F(X) IN THE INTERVAL AX,BX .
      THE CALLING STATEMENT SHOULD HAVE THE FORM

         X* = ZEROIN(AX,BX,F,TOL,IP)
  WHERE THE PARAMETERS ARE DEFINED AS FOLLOWS.
   INPUT

   AX      LEFT ENDPOINT OF INITIAL INTERVAL
   BX      RIGHT ENDPOINT OF INITIAL INTERVAL
   F       FUNCTION SUBPROGRAM WHICH EVALUATES F(X) FOR ANY X IN
           THE INTERVAL AX,BX
   TOL     DESIRED LENGTH OF THE INTERVAL OF UNCERTAINTY OF THE
           FINAL RESULT ( .GE. 0.0)
  IP       AN INTEGER PRINT FLAG.  WHEN SET TO 0, NO PRINTING
           WILL BE DONE BY ZEROIN.  IF SET TO 1, THEN
           ALL OF THE INTERVALS COMPUTED BY ZEROIN WILL
           BE PRINTED OUT .

  OUTPUT

   ZEROIN ABCISSA APPROXIMATING A ZERO OF F IN THE INTERVAL AX,BX

      IT IS ASSUMED THAT F(AX) AND F(BX) HAVE OPPOSITE SIGNS
  WITHOUT A CHECK. ZEROIN RETURNS A ZERO X IN THE GIVEN INTERVAL
  AX,BX TO WITHIN A TOLERANCE 4*MACHEPS*ABS(X) + TOL, WHERE MACHEPS
  IS THE RELATIVE MACHINE PRECISION.
      THIS FUNCTION SUBPROGRAM IS A SLIGHTLY MODIFIED TRANSLATION OF
  THE ALGOL 60 PROCEDURE ZERO GIVEN IN RICHARD BRENT, ALGORITHMS FOR
  MINIMIZATION WITHOUT DERIVATIVES, PRENTICE - HALL,INC. (1973).
  THIS VERSION IS COPIED FROM "COMPUTER METHODS FOR MATHEMATICAL
  COMPUTATIONS" BY FORSYTHE, MALCOLM, AND MOLER.  THE ONLY CHANGE
  IS THE INCLUSION OF THE PRINT FLAG IP.
```

Figure 1. (Page 2)

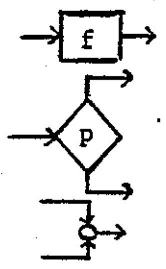
II. TECHNIQUES FOR UNDERSTANDING PROGRAMS

Flowcharts - Any flowchartable program can be analyzed in a way we describe next for better understandability and documentation. For a fuller discussion, see (Linger, Mills and Witt). We consider flowcharts as directed graphs with nodes and lines. The lines denote flow of control and the nodes denote tests and operations on data. Without loss of generality, we consider flowcharts with just three types of nodes, namely:

function node:

predicate node:

collecting nodes:



where f is any function mapping the data known to the program to new data, e.g., a simple FORTRAN assignment statement, and p is any predicate on the data known to the program, e.g., a simple FORTRAN test. An entry line of a flowchart program is a line adjacent to only one node, at its head; an exit line is adjacent to only one node, its tail.

Functions and data assignments - Any function mapping the data known to a program to new data can be defined in a convenient way by generalized forms of data assignment statements. For example, an assignment, denoted

x := e, (e.g., x := x + y)

where x is a variable known to the program and e is an expression in variables known to the program, means that the value of e is assigned to x. Such an assignment also means that no variable except x is to be altered. The concurrent assignment, denoted

x1, x2, ..., xn := e1, e2, ..., en

means that expressions e1, e2, ..., en are evaluated independently, and their values assigned simultaneously to x1, x2, ..., xn, respectively. As before, the absence of a variable on the left side means that it is unchanged by the assignment.

The conditional assignment, denoted

$$(p1 \rightarrow A1 \mid p2 \rightarrow A2 \mid \ldots \mid pn \rightarrow An)$$

where p1, p2, ..., pn are predicates and A1, A2, ..., An are assignments (simple, concurrent or conditional) means that particular assignment Ai associated with the first pi, if any, which evaluates true; otherwise, if no pi evaluates true, then the conditional assignment is undefined.

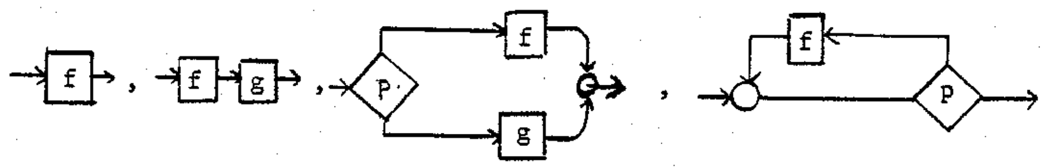
An expression in an assignment may contain a function value, e.g.,

x := max (x, abs(y))

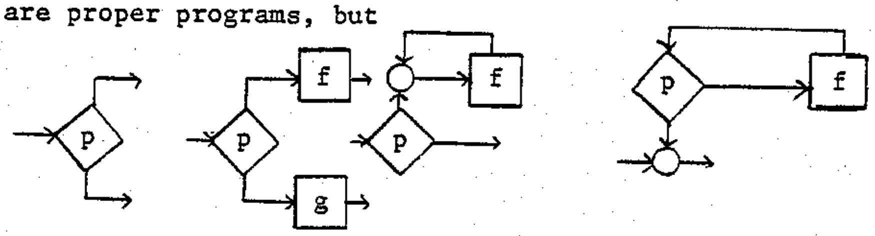
where max and abs are functions. But the function defined by the assignment statement is different, of course, from max or abs.

We note that many programming languages permit the possibility of so-called side effects, which alter data not mentioned in assignment statements or in tests. Side effects are specifically prohibited in our definition of assignments and tests.

Proper programs - We define a proper program to be a program whose flow-chart has exactly one entry line, one exit line, and, further, for every node a path from the entry through that node to the exit. For example,



are proper programs, but

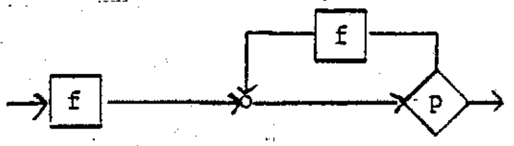


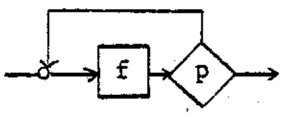
are not proper programs.

Program functions - We define a program function of a proper program P, denoted [P], to be the function computed by all possible executions of P which start at its entry and terminate at its exit. That is, a program function [P] is a set of ordered pairs, the first member being a state of the data on entry to P, the second being the resulting state of the data on exit. Note that the state of data includes input, output files which may be read from or written to intermittently during execution. Also note that if a program does not terminate by reaching its exit line from some initial data at its entry, say by looping indefinitely or by aborting, no such pair will be determined and no trace of this abnormal execution will be found in its program function.

Proper programs are convenient units of documentation. Their program functions abstract their entire effect on the data known to the program. Within a program, any subprogram which is proper can be also abstracted by its program function, that is, the effect of the subprogram can be described by a single function node whose function is the program function of the subprogram.

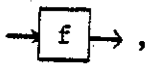
We say two programs are function equivalent if their program functions are identical. For example, the programs

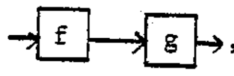


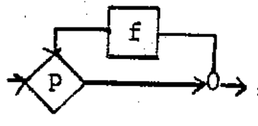
,


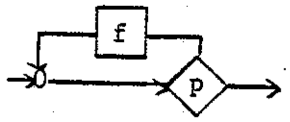
have different flowcharts but are function equivalent.

Prime programs - We define a prime program to be a proper program which contains no subprogram which is proper except for itself and function nodes. For example,

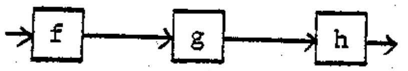


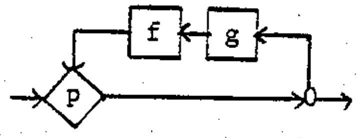
,


,


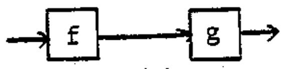
,


are primes, while

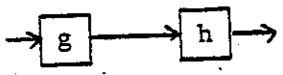


,


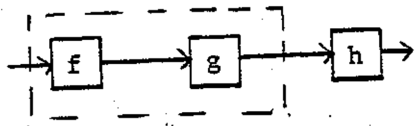
are not prime (composite programs), the first (of the composites) having subprograms

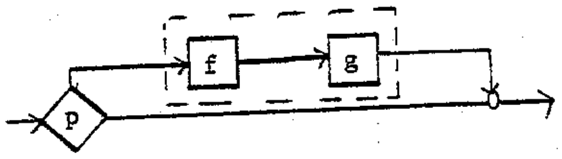


and

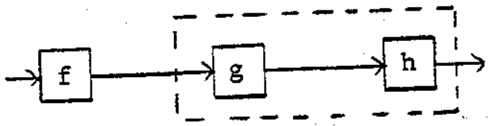


Any composite program can be decomposed into a hierarchy of primes, a prime at one level serving as a function node at the next higher level. For example, the composite programs above can be decomposed as shown next.



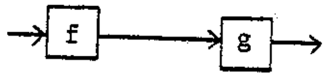


In each case, a prime is identified to serve as a function node in another prime at the next level. Note also that the first composite can also be decomposed as

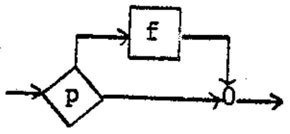


so that the prime decomposition of proper programs is not necessarily unique.

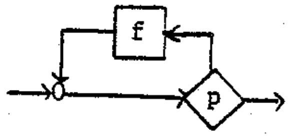
Prime programs in text form - There is a striking resemblance between prime programs and prime numbers, with function nodes playing the node of unity, and subprograms the role of divisibility. Just as for numbers, we can enumerate the control graphs of prime programs and give a text description of small primes as follows:



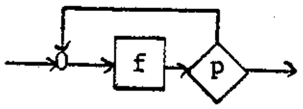
f; g



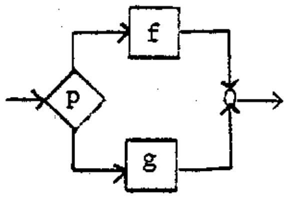
if p then f fi



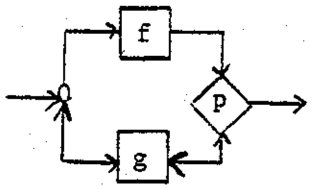
while p do f od



do f until p od



if p then f else g fi



do1 f while p do2 y od

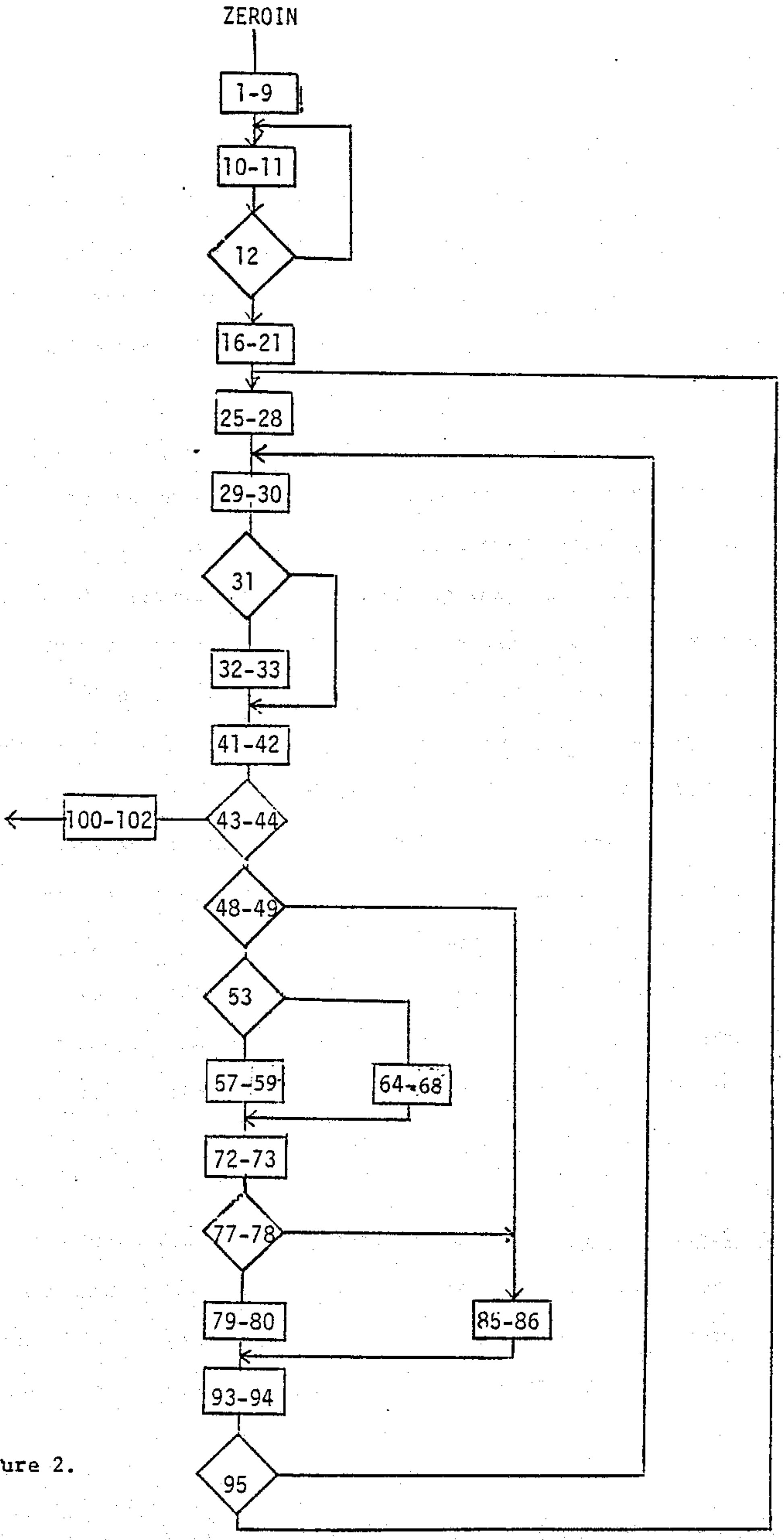
Larger primes will go unnamed here, although the case statement of Pascal is a sample of a useful larger prime. All of the primes above except the last (dowhiledo) are common to many programming languages. Prime programs in text form can be displayed with standard indentation to make the subprogram structure and control logic easily read, which we will illustrate for ZEROIN.

III. UNDERSTANDING ZEROIN

The prime program decomposition of ZEROIN - Our first step in understanding ZEROIN was to develop a prime program decomposition of its flowchart. After a little experimentation, the flowchart for ZEROIN was diagrammed as shown in Figure 2. The numbers in the nodes of the flowchart represent contiguous segments of the FORTRAN program of Figure 1, so all lowest level sequence primes are already identified and abstracted.

The flowchart program of Figure 2 was then reduced, a step at a time, by identifying primes therein and replacing each such prime by a newly numbered function node, e.g., R.2.3 names prime 3 in reduction 2 of the process. This reduction is shown in Figure 3, leading to a hierarchy of 6 levels. Of all primes shown in Figure 3, we note only two which contain more than one predicate, namely, R.3.1 and R.5.1, and each of these is easily modified into a composite made up of primes with no more than one predicate. These modifications are shown in Figure 4. We continue the reduction of these new composite programs to their prime decompositions in Figure 5. In each of these two cases, a small segment of programs is duplicated to provide a new composite which clearly executes identically to the prime. Such a modification which permits a decomposition into one predicate primes is always possible, provided an extra counter is used. In this case, it was fortunate that no such counter was required. It was also fortunate that the segments duplicated were small; otherwise, a program call in two places to the duplicated segment might be a better strategy.

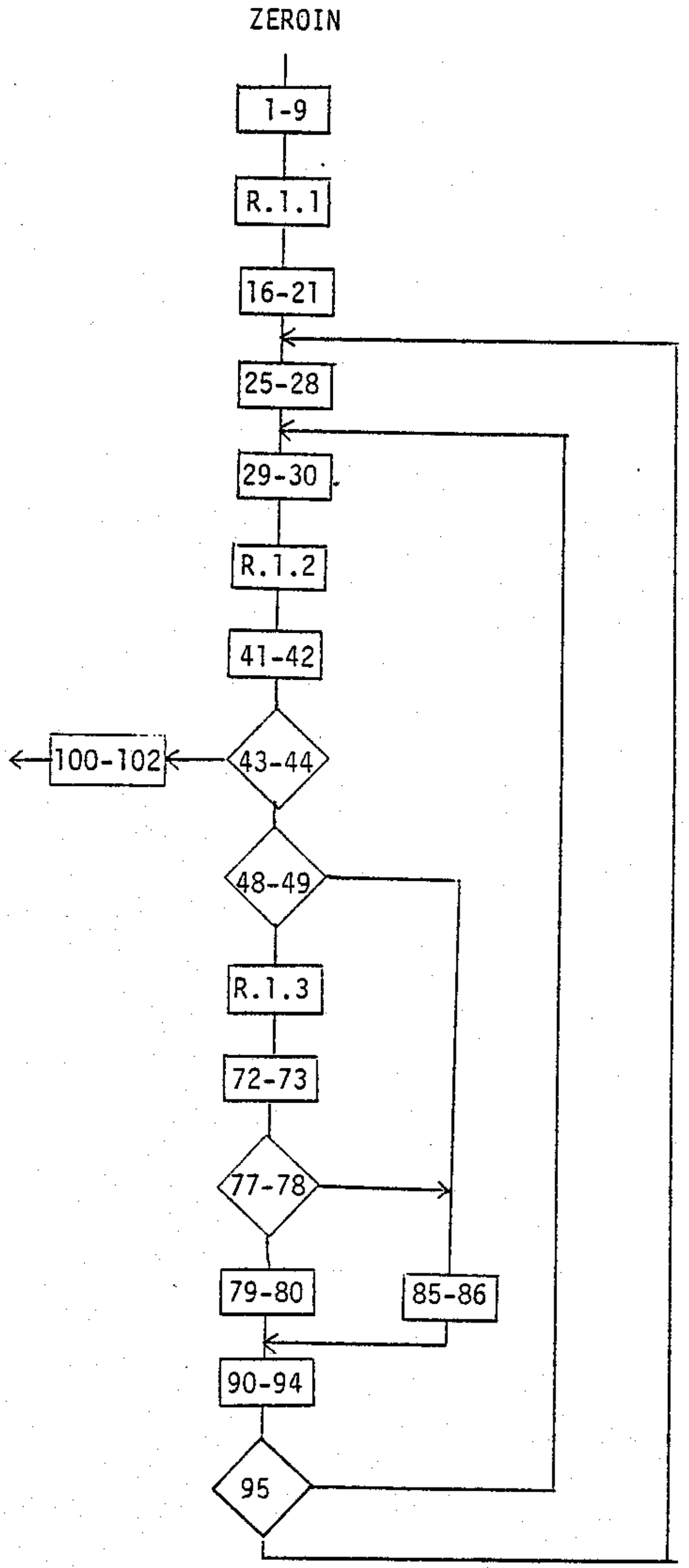
A structured design of ZEROIN - Since a prime program decomposition of a program equivalent to ZEROIN has been found with no primes of more than one predicate, we can reconstruct this program in text form in the following way: The final reduced program of ZEROIN is given in Reduction 6 of Figure 3, namely,



ZEROIN
1-9
10-11
12
16-21
25-28
29-30
31
32-33
41-42
100-102
43-44
48-49
53
57-59
64-68
72-73
77-78
79-80
85-86
93-94
95

Figure 2.

Reduction 1



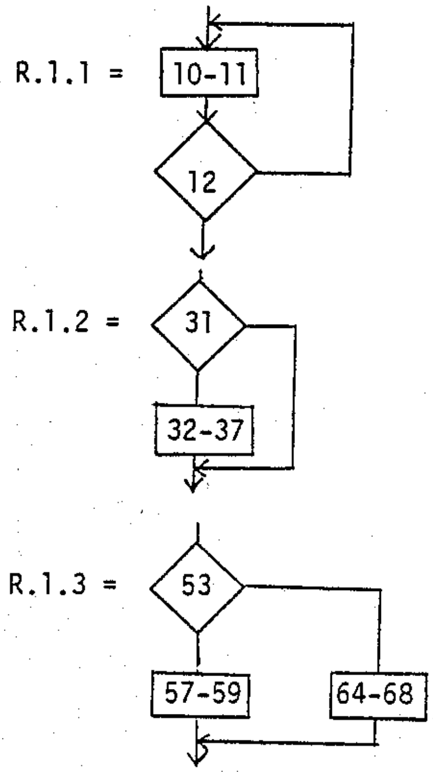


Figure 3 (1 of 4 pages)

Reduction 2

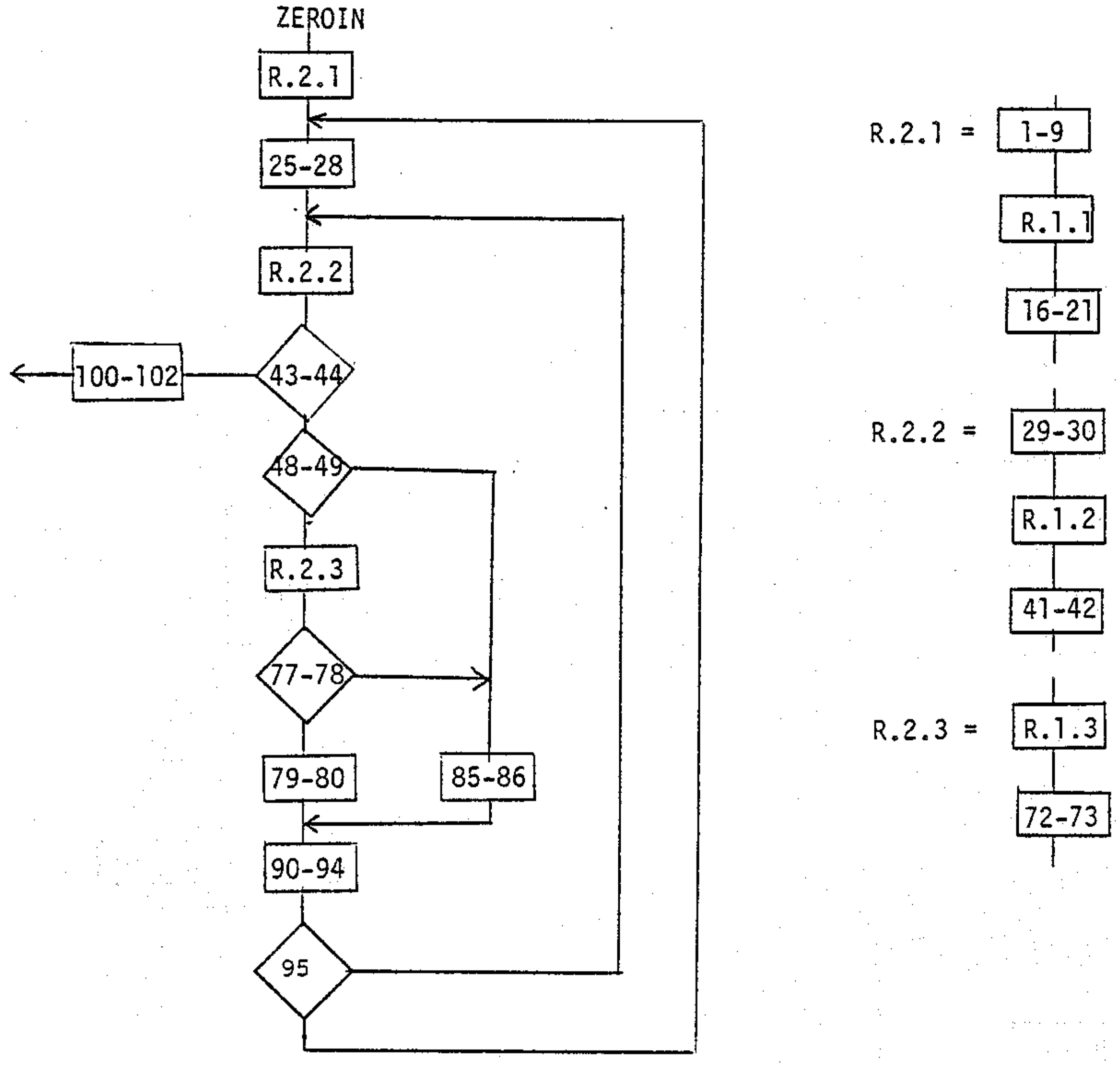
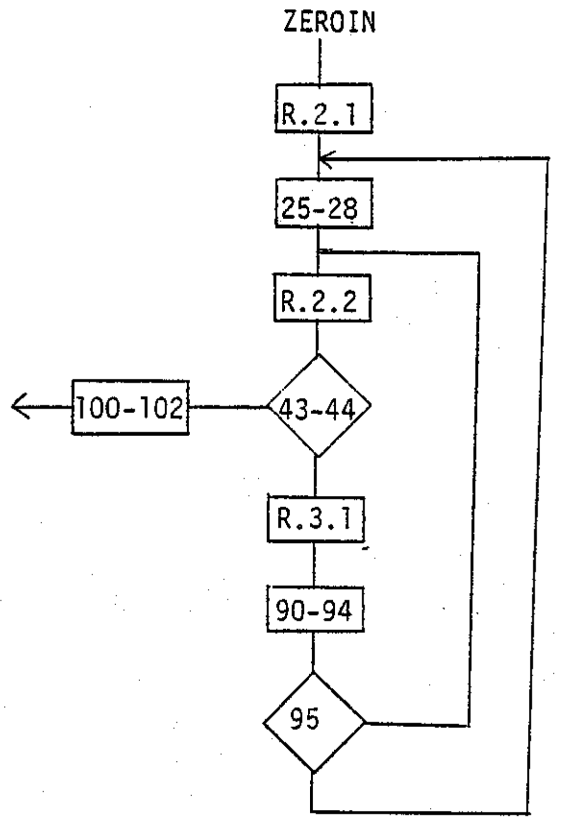
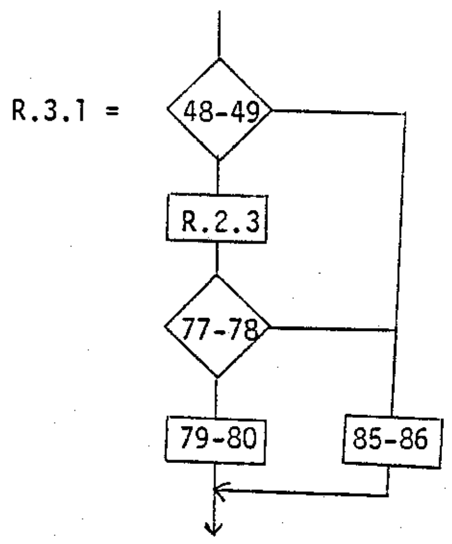


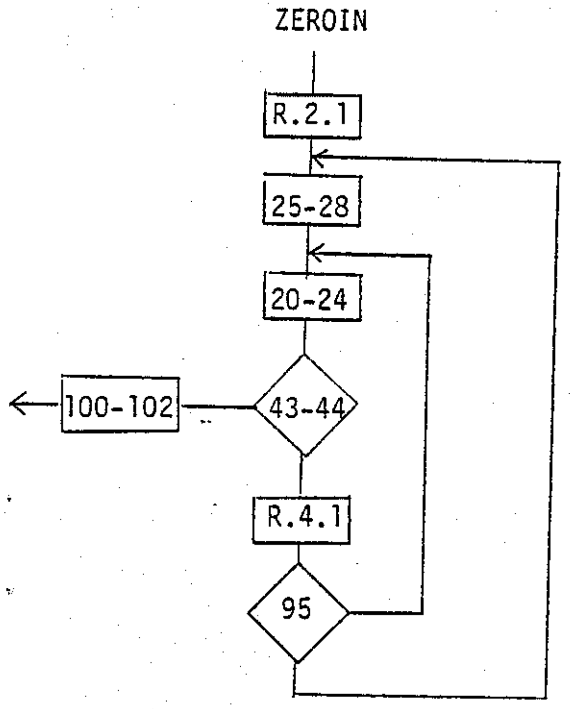
Figure 3 (2 of 4 pages)

Reduction 3





Reduction 4



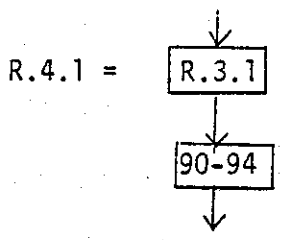
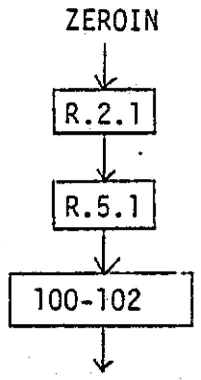
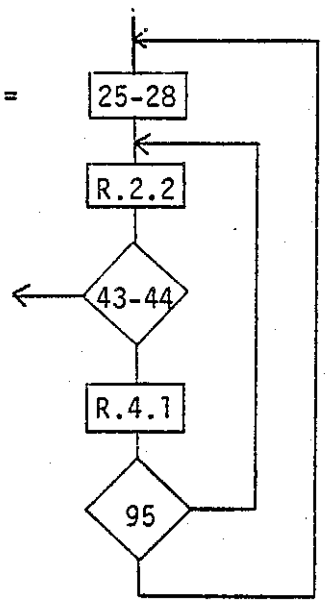


Figure 3 (3 of 4 pages)

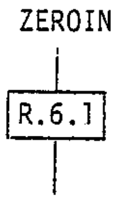
Reduction 5



R.5.1 =



Reduction 6



R.6.1 =

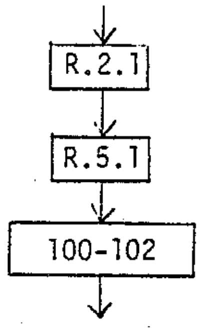
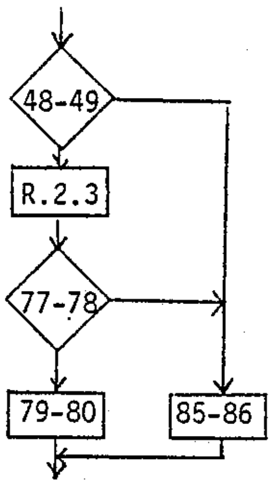


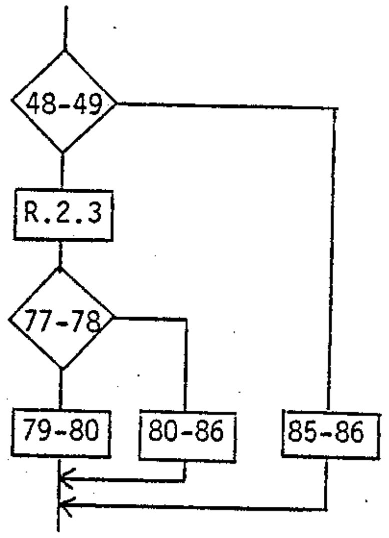
Figure 3 (4 of 4 pages)

R.3.1 =

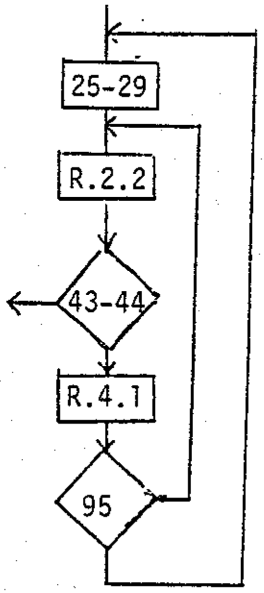


can be modified to

M.3.1 =



R.5.1 =



can be modified to

M.5.1 =

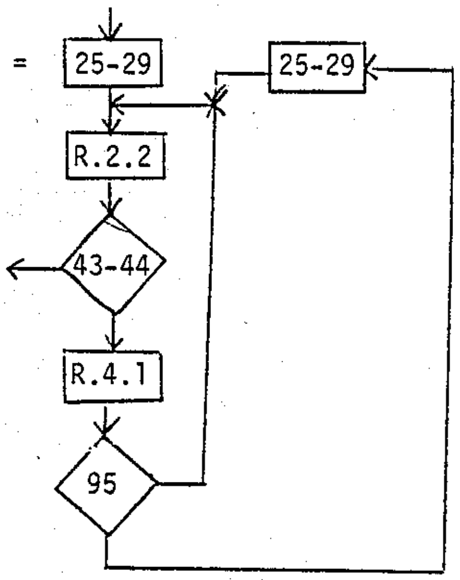
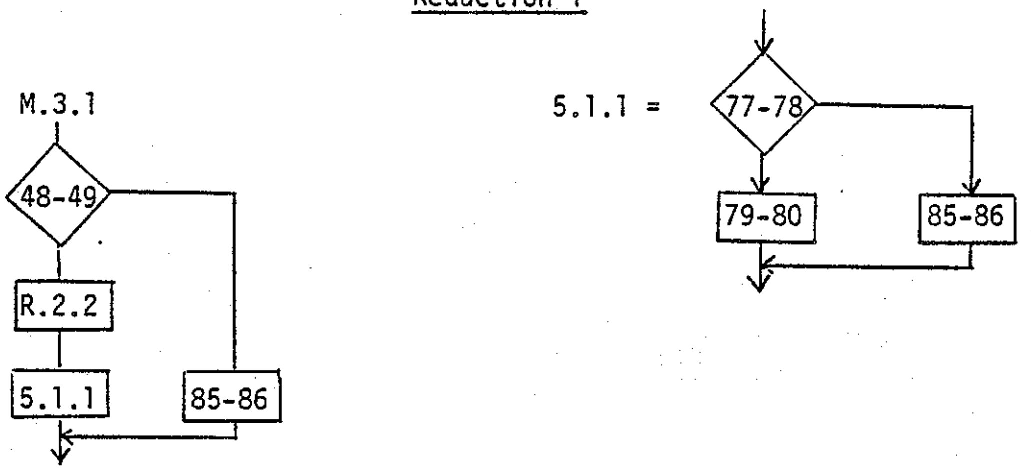
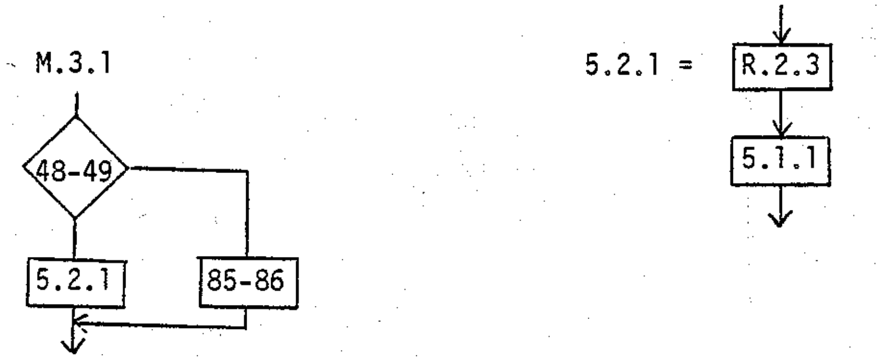


Figure 4

Reduction 1



Reduction 2



Reduction 3

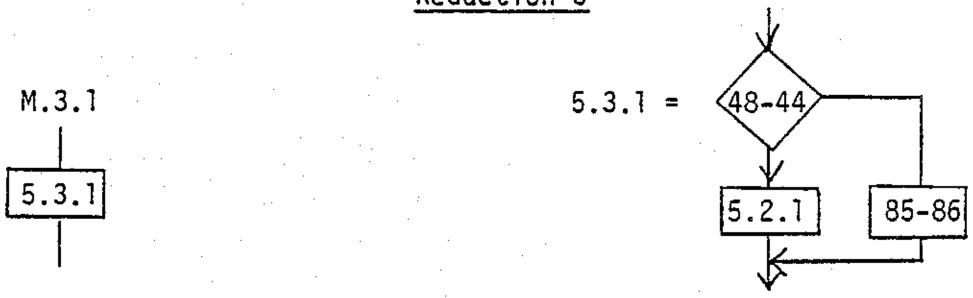
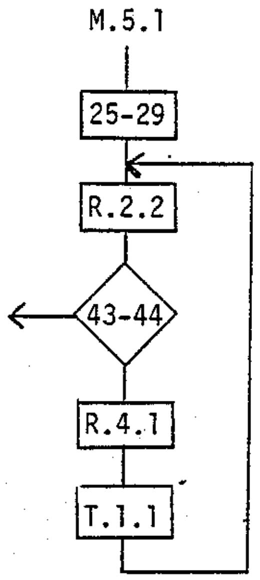
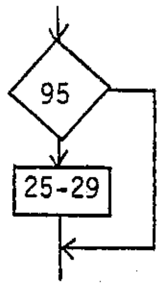


Figure 5 (1 of 2 pages)

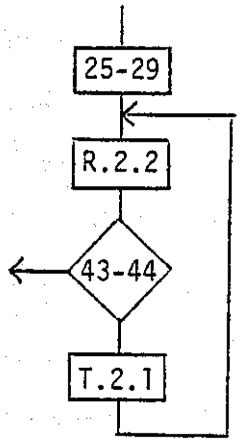
Reduction 1



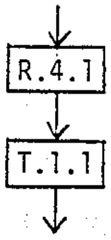
T.1.1 =



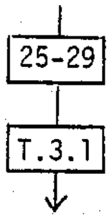
Reduction 2



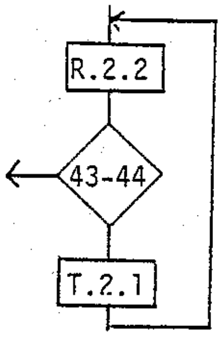
T.2.1 =



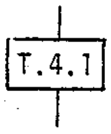
Reduction 3



T.3.1 =



Reduction 4



T.4.1 =

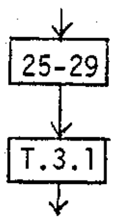
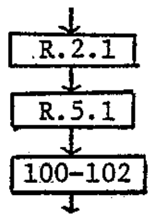


Figure 5 (2 of 2 pages)

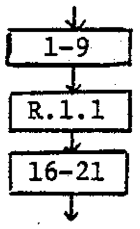
that R.6.1 is a sequence, repeated here,

R.6.1 =



Now R.2.1 can be looked up, in turn, as:

R.2.1 =



etc., until all intermediate reductions have been eliminated. Recall that R.5.1 (and R.3.1) was further reduced in Figure 5. When these intermediate reductions have all been eliminated, we obtain a structured program in PDL (Process Design Language) for ZEROIN shown in Figure 6. Note there are three columns of statement numberings. The first column holds the PDL statement number; the second holds the FORTRAN line numbering of Figure 1; the third holds the FORTRAN statement numbering of Figure 1. The FORTRAN comments have been kept intact in the structured program and appear within square brackets [,]. From here on, statement numbers refer to the PDL statements of Figure 6.

The duplication of code introduced in Figure 4 can be seen in PDL 72, 73, and PDL 96-99. It should be noted, however, that in PDL 87-91 the second IF STATEMENT in FORTRAN 93 can be eliminated by use of the if-then-else. This permits an execution time improvement to the code. A second improvement can be seen in PDL 62-66. The use of the absolute value function can be eliminated and the if-then-else can be used to transform the else negative p into a positive p only in the case where p is negative.

```
FORTRAN
Line       Stmt
Refer-      #
ence       ref.
                 ZEROIN. PROGRAM

 1   1-2         func zeroin (real ax, bx, f, tol, integer ip)
 2    5            real a, b, c, d, e, eps, fa, fb, fc,
 3                 tol 1, xm, p, q, r, s
 4    7           [COMPUTE EPS, THE RELATIVE MACHINE PRECISION]
 5    9            eps := 1.0
 6                 do
 7   10     10       eps := eps/2.0
 8   11              tol 1 := 1.0 + eps
 9                 until
10   12              tol 1 ≤ 1
11                 od
12   14           [INITIALIZATION]
13   16            if ip = 1 then write ('THE INTERVALS DETERMINED BY ZEROIN ARE') fi
14   18            a := ax
15   19            b := bx
16   20            fa := f(a)
17   21            fb := f(b)
18   23           [BEGIN STEP]
19   25     20     c := a
20   26            fc := fa
21   27            d := b-a
22   28            e := d
23                 do1
24   29     30       if ip = 1 then write (b, c) fi
25                   if
26   31                 abs (fc) < abs (fb)
27                   then
28   32                 a := b
29   33                 b := c
30   34                 c := a
31   35                 fa := fb
32   36                 fb := fc
33   37                 fc := fa
34                   fi
35   39          [CONVERGENCE TEST]
36   41     40       tol 1 := 2.0 * eps * abs (b) + 0.5 * tol
37   42              xm := .5 * (c-b)
38  {43            while
39  {44              abs (xm) > tol 1 and fb ≠ 0
40                 do2
41                   [IS BISECTION NECESSARY]
42                   if
43                      abs (e) < tol 1 or abs (fa) ≤ abs (fb)
44   83              then [BISECTION]
45   85     70          d := xm
46   86                 e := d
47   46              else [IS QUADRATIC INTERPOLATION POSSIBLE]
48                      if
49   48                   a ≠ c
50   62                 then [INVERSE QUADRATIC INTERPOLATION]
```

Figure 6. (1 of 2 pages)

```
Line     Stmt |
Refer-    #   |
ence     ref. |

 51 | 64 | 50 |          q := fa/fc
 52 | 65 |    |          r := fb/fc
 53 | 66 |    |          s := fb/fa
 54 | 67 |    |          p := s * (2.0 * xm * q * (q-r) - (b-a) * (r-1.0))
 55 | 68 |    |          q := (q-1.0) * (r-1.0) * (s-1.0)
 56 | 55 |    |        else [LINEAR INTERPOLATION]
 57 | 57 |    |          s := fb/fa
 58 | 58 |    |          p := 2.0 * xm * s
 59 | 59 |    |          q := 1.0 - s
 60 |    |    |        fi
 61 | 70 |    |        [ADJUST SIGNS]
 62 |    |    |        if                                /* note can be          */
 63 | 72 | 60 |          p > o                           /* if p > o then q := -q */
 64 |    |    |        then                              /*          else p := -p */
 65 | 72 |    |          q := -q                         /* in PDL               */
 66 |    |    |        fi                                /*                      */
 67 | 73 |    |        p := abs(p)
 68 | 75 |    |        [IS INTERPOLATION ACCEPTABLE]
 69 |    |    |        if
 70 | 77 |    |         (2.0 * p) ≥ (3.0 * xm * q - abs (tol 1 * q))
 71 | 83 |    |        then [BISECTION]
 72 | 85 | 70 |           d := xm                        /* note 85-86 repeated */
 73 | 86 |    |           e := d                         /*        in PDL       */
 74 |    |    |        else
 75 | 79 |    |           e :=d
 76 | 80 |    |           d := p/q
 77 |    |    |        fi
 78 |    |    |     fi
 79 |    |    |     [COMPLETE STEP]
 80 | 90 | 80 |     a := b
 81 | 91 |    |     fa := fb
 82 |    |    |     if
 83 | 92 |    |       abs(d) > tol 1                     /* note test done twice */
 84 |    |    |     then                                 /*      in FORTRAN      */
 85 | 92 |    |       b := b + d
 86 |    |    |     fi                                   /*   here and           */
 87 |    |    |     if                                   /*   in line 88         */
 88 | 93 |    |       abs(d) ≤ tol 1
 89 |    |    |     then
 90 | 93 |    |       b := b + sign (tol 1, xm)
 91 |    |    |     fi

 92 | 94 |    |     fb := f(b)
 93 |    |    |     if
 94 |    |    |       fb * (fc/abs (fc))  > 0.0
 95 |    |    |     then [BEGIN STEP]
 96 | 25 | 20 |       c := a                             /* note 25-28 */
 97 | 26 |    |       fc := fa                           /*  repeated  */
 98 | 27 |    |       d := b - a                         /*  in PDL    */
 99 | 28 |    |       e := d
100 |    |    |     fi
101 |    |    |  od
102 | 98 |    |  [DONE]
103 |100 |    |  zeroin := b
104 |101 |    |  return
105 |102 |    | cnuf
```

Figure 6. (2 of 2 pages)

By construction, the PDL program of Figure 6 is function equivalent to the FORTRAN program of Figure 1. But the PDL program will be simpler to study and understand.

Data references in ZEROIN - Our next step in understanding ZEROIN was to develop a data reference table for all data identifiers. While straightforward and mechanical, there is still much learning value in carrying out this step, in becoming familiar with the program in the new structured form. The results are given in Figure 7. This familiarization led to the following observations about the data references in ZEROIN (in no particular order of significance, but as part of a chronological, intuitive, discovery process):

1. ax, bx, f, ip, tol are never set, as might be expected, since they are all input parameters (but this check would determine initialized data if it existed, and also checks for the presence of side effects by the program on its parameters if passed by reference).
2. Zeroin is never used, but is returned as the purported zero found for f (since Zeroin is set to b just before the return of the program, it appears that b may be a candidate for this zero during execution).
3. eps is set by the dountil loop 6-11 at the start of program execution, then used as a constant at statement 36 from then on.
4. tol 1 is used for two different unrelated purposes, namely, as a temporary in the dountil look 6-11 which sets eps, then reset at statement 36 as part of a convergence consideration.
5. the function f is called but three times, at 16, 17 to initilize fa, fb, and at 92 to reset fb to f(b) (more evidence that b is the candidate zero to be returned).
6. the identifiers a, c are set to and from b, and the triple a, b, c seems to be a candidate for bracketing the zero which b (and zeroin) purports to approach.

	Set	Used
a	14,28,80	16,19,21,30,49,54,96,98
ax		14
b	15,29,85,90	17,21,24,28,36,37,54,80,85,90,92,98,103
bx		15
c	19,30,96	29,37,49
d	21,45,72,76,98	22,46,73,75,83,85,88,99
e	22,46,73,75,99	43
eps	5,7	7,8,36
f		16,17,92
fa	16,31,81	20,33,43,51,53,57,97
fb	17,32,92	26,31,39,43,52,53,57,81,94
fc	20,33,97	26,32,51,52,94
ip		13,24
p	54,58,67	63,67,70,76
q	51,55,59,65	54,55,65,70,76
r	52	54,55
s	53,57	54,55,58,59
tol		36
tol 1	8,36	10,39,43,70,83,88
xm	37	39,45,54,58,70,72,90
zeroin	101	

Figure 7.

7. the identifiers fa, fb, fc are evidently standins for f(a), f(b), f(c), and serve to limit the calls on function f to a minimum.
8. the identifiers p, q, r, s are initialized and used only in the section of the program that the comments indicate is concerned with interpolation.
9. focusing on b, aside from initialization at statement 15, and as part of a general exchange among a, b, c at statement 28-29, b is updated only in the ifthenelse 83-90, incremented by either d or tol 1.
10. d is set to xm or p/q (as a result of a more complex bisection and interpolation process); xm is set only at statement 37 to the half interval of (b, c) and appears to give a bisection value for b.

A function decomposition of ZEROIN - The prime program decomposition and the familiarity developed by the data reference tabulation and observations suggest the identification of various intermediate prime or composite programs in playing important roles in summing up a functional structure for ZEROIN. Each such intermediate prime or composite program computes values of a function. The inputs (function arguments) of this function are defined by the initial values of all identifiers which are inputs (function arguments) for statements which make up the intermediate program. The outputs (function values) of this function are defined by the final values of all identifiers which are outputs (function values) for statements which make up the intermediate program. Of course, further analysis may disclose that such a function is independent of some inputs, if, in fact, such an identifier is always initialized in the intermediate program before its use.

On the basis of this prime decomposition and data analysis, we reformulated ZEROIN of Figure 6 as zeroinl, a sequence of four intermediate programs, as shown in Figure 8, with function statements using the form f. n-m where n, m are the boundary statements of the intermediate programs of ZEROIN from Figure 6. The identifier *outfile in the output lists refers to the fact that data is being transferred to an outfile by an intermediate program. The phrase (x,z,v) projection of some function x,y,z,u,v,w := p,q,r,s,t,u means the new function x,z,v := p,r,t.

In the program descriptions which follow, all arithmetic operations are assumed to represent machine arithmetic. However, we will occasionally apply normal arithmetic axioms in order to simplify expressions. We next look at the intermediate programs.

<u>f.5-11</u> - The intermediate program which computes the values of f.5-11 is a sequence, namely, an initialized dountil, i.e.

```
 5  eps := 1.0
 6  do
 7      eps := eps/2.0
 8      tol 1 := 1.0 + eps
 9  until
10      tol 1 ≤ 1
11  od
```

After some thinking, we determined that at PDL 6, an invariant of the form

$$I6 = (\exists\, k \geq o\ (eps = 2^{-k}))\ \wedge\ 1 + eps > 1$$

must hold, since entry to PDL 6 must come from PDL 5 or PDL 10 (and in the latter case tol 1 > 1, having just been set to 1.0 + eps, so 1.0 + eps > 1). Furthermore, at PDL 9 the invariant

$$I9 = (\exists\, k \geq 1\ (eps = 2^{-k}))\ \wedge\ tol\ 1 = 1 + eps$$

must hold, by observing the effect of PDL 7, 8 on the invariant I6 at PDL 6. Therefore, at exit (if ever) from the segment PDL 5-11, we must have the condition I9 Λ PDL 10, namely

$$(\exists\, k \geq 1\ (eps = 2^{-k}))\ \wedge\ 1 + 2\ eps > 1\ \wedge\ tol\ 1 = 1 + eps \leq 1$$

```
func zeroin 1 (real ax, bx, f, tol, integer ip)
   real a, b, c, d, e, eps, fa, fb, fc, p, q, r, s, tol 1, xm
   integer ip
   [compute eps, the relative machine precision]
   eps, tol 1 := f. 5-11
   [initialize data]
   a, b, c, d, e, fa, fb, fc, *outfile := f. 13-22 (ip, ax, bx, f)
   [estimate b as a zero of f]
   a, b, c, d, e, fa, fb, fc, p, q, r, s, tol 1,xm, *outfile :=
      f. 23-101 (a, b, c, d, e, f, fa, fb, fc, ip, p, q, r, s, tol 1, xm)
   [set zeroin for return, zeroin := b]
   zeroin := f. 103-103(b)
   return
cnuf
```

Figure 8

Thus we have

Lemma 5-11 The program function of f.5-11 is the constant function.

$\{(\emptyset, (\text{eps}, \text{tol } 1)) \mid (\exists\, k \geq 1\ (\text{eps} = 2^{-k}))\ \wedge\ 1 + 2\ \text{eps} > 1\ \wedge\ \text{tol } 1 = 1 + \text{eps} \leq 1\}$

Since tol 1 is reassigned (in PDL 36) before it is used again, f.5-11 can be thought of as computing only eps.

f.13-22 - The intermediate program which computes the value of f.13-22 is a sequence which can be written directly as a multiple assignment. It is convenient to retain the single output statement PDL 13, and write

f.13-22 = f.13-13; f.14-22

yielding

Lemma 13-22 The (a,b,c,d,e,*outfile) projection of f.13-22 is function equivalent to the sequence

f.13-13; f.14-22

where f.13-13 = *if* ip = i *then* write ('THE INTERVALS DETERMINED BY ZEROIN ARE')

f.14-22 = a,b,c,d,e := ax,bx,ax,bx-ax,bx-ax

f.23-101 - The intermediate program which computes the value of f.23-101 is a bit more complicated than the previous program segments and will be broken down into several subsegments. We begin by noticing that several of the input and output parameters may be eliminated from the list. Specifically, as noted earlier, p, q, r, and s are local variables to f.23-101 since they are always recalculated before they are used in f.23-101 and they are not used outside of f.23-101. The same is true for xm and tol 1. fa, fb, and fc can be eliminated since they are only used to hold the values of f(a), f(b) and f(c).

After considerable analysis and a number of false starts leading into a great deal of detail, we discovered an amazing simplification, first as a conjecture, then as a more precise hypothesis, and finally as a verified result. This simplification concerned the main body of the iteration of zeroin, namely

PDL 41-92, and obviated the need to know or check what kind of interpolation strategy was used, step by step. This discovery was that the new estimate of b always lay strictly within the interval bracketed by the previous b and c. That is, PDL 41-92, among other effects, has the (b) projection

$$b := b + \alpha(c-b), \text{ \underline{for some} } \alpha,\ 0 < \alpha < 1$$

so that the new b was a fraction α of the distance from the previous b to c. With a little more thought, it became clear that the precise values of d, e could be ignored, their effects being captured in the proper (but precisely unknown) value of α. Furthermore, this new indeterminate (but bounded) variable α could be used to summarize the effect of d, e in the larger program part PDL 23-101, because d, e are never referred to subsequently. Thus, we may rewrite f.23-101 at this level as

a, b, c *outfile := f.23-101 (a, b, c, f, ip)

and we define it as an initialized while loop.

Lemma 23-101 The (a, b, c, *outfile) projection of f.23-101 is function equivalent to

```
(ip = 1 → write (b, c) | true → I);                         [Lemma 24]
( | f(c) | < | f(b) | → a, b, c := b, c, b | true → I)      [Lemma 25-34]
while
   f(b) ≠ 0 ∧ | (c-b)/2 | > 2 eps | b | + tol/2
do
   a, b, c := b, b + α(c-b), c where 0 < α < 1;             [Lemma 41-92]
   (f(b) * f(c) > 0 → a, b, c := a, b, a | true → I);       [Lemma 93-100]
   (ip = 1 → write (b, c) | true → I);                      [Lemma 24]
   ( | f(c) | < | f(b) | → a, b, c := b, c, b | true → I)   [Lemma 25-34]
od
```

where I is the identity mapping.

The structure of f.23-101 corresponds directly to the structure of PDL 23-101 except for a duplication of segment PDL 23-34 in order to convert the dowhiledo into a whiledo. The proof of the correctness of the assignments of f.23-101 is given in separate lemmas as noted in the comments attached to the functions in Lemma 23-101. The while test is obtained by direct substitution of values for tol 1 and xin defined in PDL 36-37 into the test in PDL 39 using eps as defined in Lemma 5-11.

Lemma 24 PDL 24 is equivalent to $(ip = 1 \rightarrow \text{write } (b, c) \mid \text{true} \rightarrow I)$

pf: By direct inspection

Lemma 25-34 The (a, b, c) projection of the program function of PDL 25-34 is function equivalent to

$$(\,|f(c)| < |f(b)| \rightarrow a, b, c := b, c\ b \mid \text{true} \rightarrow I)$$

pf: By direct inspection of PDL 25-34

Lemma 41-92 The (a, b, c) projection of the program function of PDL 41-92 is function equivalent to

$$a, b, c := b, b + \alpha(c-b), c \text{ where } 0 < \alpha < 1$$

The proof will be done by examining the set of relationships that must hold among the variables in PDL 41-92 and analyzing the values of p and q only. That is, it is not necessary to have any knowledge of which interpolation was performed to be able to show that the new b can be defined by

$$b := b + \alpha(c-b) \quad , \quad 0 < \alpha < 1$$

We will ignore the test on PDL 48 since it will be immaterial to the lemma whether linear or quadratic interpolation is performed. We will examine only the key tests and assignments and do the proof in two basic cases--interpolation and bisection--to show that the (d) projection of the program function of PDL 41-78 is

$$d = (c-b)\ (\alpha) \text{ where } 0 < \alpha < 1$$

<u>Case 1 Interpolation</u>

If interpolation is done, an examination of Figure 6 shows that the following set of relations holds at PDL 78:

- I1. tol 1 = 2 * eps * abs (b) + .5 * tol (PDL 36)
- I2. xm = (c-b)/2 (PDL 37)
- I3. abs (xm) > tol 1 (PDL 39)
- I4. $p \geqslant 0$ (PDL 67)
- I5. 2. * p < 3 * xm * q - abs(tol 1 * q) (PDL 70)
- I6. d = p / q (PDL 76)
- I7. abs(d) > tol 1 (PDL 83)

Now let's examine the set of cases on p and q

<u>$p > 0 \wedge q < 0$</u>

We have $d = p/q < 0$ (by hypotheses),

$\frac{p}{q} > \frac{3}{2}$ xm + $\frac{\text{tol 1}}{2}$ (by I5), and tol 1 > 0 by(I1)

Since abs(xm) > tol 1 (by I3) and $\frac{3}{2}$ xm + $\frac{\text{tol 1}}{2} < 0$ (since $p/q < 0$)

we have xm < 0 implying $0 > d > \frac{p}{q} > \frac{3}{2}\,\text{xm} > \frac{3}{4}\,(c-b) > (c-b)$.

Thus $0 > d > (c-b)$ yielding $d = \alpha(c-b)$ where $0 < \alpha < 1$

<u>$p > 0 \wedge q > 0$</u>

We have $d = \frac{p}{q} > 0$ (by hypotheses),

$\frac{p}{q} < \frac{3}{2}\,\text{xm} - \frac{\text{tol 1}}{2} < \frac{3}{2}\,\text{xm} = \frac{3}{4}\,(c-b) < (c-b)$ (by I5, I1, I2)

implying $0 < d < (c-b)$. Thus $d = \alpha(c-b)$ where $0 < \alpha < 1$

<u>$p > 0 \wedge q = 0$</u>

q = 0 implies 0 > 2 * p (by I5) and we know p > 0 (by hypotheses), implying a contradiction

<u>$p = 0 \wedge q =$ anything</u>

abs(p/q) > tol 1 (by I6, I7) and tol 1 $\geqslant$ 0 (by I1) implies p cannot be 0

<u>$p < 0 \wedge q =$ anything</u>

p $\geqslant$ 0 (by I4) implies a contradiction

<u>Case 2 Bisection</u>

If bisection is done, an examination of Figure 6 shows that the following set of relations holds at PDL 78

B1. xm = (c-b)/2 (PDL 37)

B2. abs(xm) > tol 1 (PDL 39)

B3. d = xm (PDL 45 or PDL 72)

Here d = xm (by B3) implies $\alpha = \frac{1}{2}$ (by B1) and thus d = (c-b)(α) where

$0 < \alpha < 1$

PDL 82-91 implies if $|d| \leqslant$ tol 1 (i.e., if d is too small) then increment b by tol 1 with the sign adjusted appropriately

i.e. set $\alpha = \left\{\begin{array}{ll} d & \text{abs}(d) > \text{tol } 1 \\ \text{sign (tol 1, xm)} & \text{otherwise} \end{array}\right\}$

But tol 1 < abs(xm) (by I3 and B2) = abs((c-b)/2) and the sign (tol 1) is set to the sign (xm) implying

tol 1 = α(c-b) where $0 < \alpha < 1$

Thus, in PDL 82-91 b is incremented by d or tol 1, both of which are of the form α(c-b) where $0 < \alpha < 1$. Thus we have

$$b := b + \alpha(c-b) \quad , \quad 0 < \alpha < 1$$

and since in PDL 80-81 we have a, fa := b, fb we get the statement of the Lemma.

Once again, the reader is reminded that the proof of Lemma 41-92 was done by examining cases on p and q only. No knowledge of the actual interpolations was necessary. Only tests and key assignments were examined. Also, the program function was abstracted to only the key variables a, b, c and α represented the effect of all other significant variables.

<u>Lemma 93-100</u> The (a,b,c) projection of PDL 93-100 is function equivalent to

$$(f(b) * f(c) > 0 \rightarrow a, b, c := a, b, a \mid \underline{true} \rightarrow I)$$

<u>pf</u>: By direct inspection, PDL 93-100 is an if then statement with if test equivalent to the condition shown above and assignments which include the assignments above.

The last function in zeroin 1 (from Figure 8) is the single statement PDL 103 which can be easily seen as

<u>Lemma 103</u> f.103 is function equivalent to zeroin := b

Now that each of the pieces of zeroin 1 have been defined, the program function of zeroin will be given. First, let us rewrite zeroinl, all in one place, using the appropriate functions (Figure 9).

```
func zeroinl  (real ax, bx, f, tol, integer ip)
  real a, b, c, d, e, eps, fa, fb, fc, α
  file *outfile
  [compute eps, the relative machine precision]
     eps := {x | (∃ k ≥ 1 (x = 2^-k)) ∧ 1 + 2 x > 1 ∧ 1 + eps ≤ 1} ;
  [initialize data]
     (ip = 1 → *outfile := 'THE INTERVALS DETERMINED BY ZEROIN
                                ARE'  | true → I) ;
     a,b,c,d,e := ax,bx,ax,bx-ax,bx-ax
  [estimate b as a zero of f]
     (ip = 1 → *outfile (b, c) | true → I) ;
    (abs(f(c)) < abs(f(b)) a, b, c := b, c, b | true → I)
    while
       f(b)  ≠ 0 ∧ | (c-b)/2 | > 2 eps | b | + tol/2
    do
       a, b, c := b, b + α (c-b), c  where  0 < α < 1;*
       (f(b) * f(c) > 0 → a, b, c := a, b, a | true  → I) ;
       (ip = 1 → *outfile(b, c) | true → I) ;
       (abs(f(c)) < abs(f(b)) → a, b, c := b, c, b | true → I)
    od
    [set zeroin for return, zeroin := b]
    zeroin := b
  return
cnuf
```

Figure 9

* α is an indeterminate based on the current values of a, b, c, d, e, f, fa, fb, fc, tol and eps

Theorem 1-105

func zeroin has program function [zeroin] =

$(ax = bx \rightarrow root := bx \mid$

$f(bx) = 0 \rightarrow root := bx \mid$

$f(ax) = 0 \rightarrow root := ax \mid$

$f(ax) * f(bx) < 0 \rightarrow root := approx\ (f, ax, bx, tol) \mid$

$\underline{true} \rightarrow (\forall\ k = 1,2,\ldots,f(b_k) * f(c_k) > 0 \rightarrow root := unpredictable \mid$

$\exists\ k > 0\ (f(b_k) * f(c_k) \leq 0 \wedge \forall\ j = 1,2,\ldots k-1,\ f(b_j) * f(c_j) > 0) \rightarrow$

$root := approx\ (f, b_k, c_k, tol)$

where

approx (f, ax, bx, tol) is some value in the interval (ax, bx) within $4 * eps * |x| + tol$ of some zero x of the function f

and

the sequence (b1, c1), (b2, c2), ... is defined so that each succeeding interval is a sub-interval of the preceding interval; and in the case where abs(d)≤tol 1 never occurs {b1, c1} = {ax, bx}, $\{b_{k+1}, c_{k+1}\}$ defines the half interval of $\{b_k, c_k\}$ including b_k, and b_{k+1} is chosen to minimize $abs(f(b_{k+1}))$.

Proof: The proof will be carried out in cases, corresponding to the conditions in the rule given in the Theorem, The first three cases follow directly by inspection of zeroinl, as special cases for input values, which bypass the while loop. I.e., if ax = bx, then the values of a, b, c and root can be traced in zeroin 1 as follows:

	a	b	c	root
zeroin 1.8	bx	bx	bx	
.11	bx	bx	bx	
[condition 13 fails since c-b = 0]				
.21	bx	bx	bx	bx

Cases 2 and 3 proceed in a similar fashion.

Case 4, f(ax) * f(bx) <0, will be handled by an analysis of the whiledo loop and its results will apply to the last subcase of the last case as well. The first subcase of the last case arises when no zero of f is even bracketed and zeroinl runs a predictable course, as will be shown.

Case 4: It will be shown that the entry condition f(ax) * f(bx) <0 leads to the following condition at the whiletest of zeroinl:

$$I = (a = c \neq b \lor a < b < c \lor c < b < a) \land f(b) * f(c) \leq 0 \land abs(f(b) \leq abs(f(c))$$

The proof is by induction. First I holds on entry to the whiledo loop because by direct calculation

after zeroinl.8 $a = c \land f(b) * f(c) < 0 \land c \neq b$

after zeroinl.11 $a = c \land f(b) * f(c) < 0 \land abs(f(b)) \leq abs(f(c)) \land c \neq b$

Next, suppose the invariant I holds at any iteration of the whiledo at the whiletest, and the whiletest evaluates true, it can be shown that I is preserved by the three-part sequence of the do part. In fact, it will appear that the first part, in seeking a better estimate of a zero of f may destroy this invariant, and the last two parts do no more than to restore the invariant. It will be shown in Lemma 15-18 that

after zeroinl.15 $(a < b < c \lor c < b < a) \land f(a) * f(c) < 0$

after zeroinl.16 $(a=c\neq b \lor a < b < c \lor c < b < a) \land f(b) * f(c) \leq 0$

after zeroinl.18 $(a=c\neq b \lor a < b < c \lor c < b < a) \land f(b) * f(c) \leq 0 \land abs(f(b)) \leq abs(f(c))$

which is I, again. Thus, I is indeed an invariant at the whiletest.

Consider the question of termination of the whiledo. In Lemma 15-18T it will be shown using c_o and b_o as entry values to the do part, that for some α, $0<\alpha<1$, after zeroinl.18 $abs(c-b) < abs(c_o - b_o) \max(\alpha, 1-\alpha)$.

Therefore, the whiledo must finally terminate because the condition

$$f(b) \neq 0 \ \Lambda \ abs((c-b)/2) > 2 * eps * abs(b) + tol/2$$

must finally fail, because by the finiteness of machine precision abs(c-b) will go to zero if not terminated sooner.

When the whiledo terminates, the invariant I must still hold. In particular $f(b) * f(c) \leqslant 0$, which combined with the negation of the whiletest gives

$$IT = f(b) * f(c) \leqslant 0 \ \Lambda (f(b)) = 0 \ V \ abs((c-b)/2) \leqslant 2 * eps * abs(b) + tol/2$$

IT states that

1) a zero of f is bracketed by the interval (b, c)

2) either the zero is at b or the zero is at most $|c-b|$ from b,

   i.e., the zero is within $4 * eps * |b| + tol$ of b.

This is the definition of approx (f, b, c, tol).

Now, beginning with the interval (ax, bx), every estimate of b created at zeroinl.15 remains within the interval (b,c) current at the time*. Since c and b are initialized as ax and bx at zeroinl.8, the final estimate of b is given by approx (f, ax, bx, tol). The assignment zeroin := b at zeroinl.21 provides the value required by case 4.

<u>Case 5: part 1</u>. We first show that in this case the condition a = c will hold at zeroinl.15 if $f(b) * f(c) > 0$. By the hypothesis of case 5, part 1, f((b+c)/2) is of the same sign as f(b) and f(c). Therefore, the first case of zeroinl.16 will hold and the assignment c := a will be executed implying a = c when we arrive at zeroinl.15 from within the loop. Also, if we reach zeroinl.15 from outside the loop (zeroinl.8-11) we also get a = c.

We now apply Lemma 15L, which states that under the above condition the (a, b, c) projection of zeroinl.15 is

*this is because $f(b) * f(c) \leqslant 0$ is part of I

$$(f(b) * f(c) > 0 \rightarrow a,\ b,\ c := b, \left\{\begin{array}{ll} b + (c-b)/2, & \text{if abs}(c-b)/2 > \text{tol 1} \\ b + \text{tol 1}, & \text{otherwise} \end{array}\right\}, c$$

$$\underline{\text{true}} \rightarrow a,\ b,\ c := b,\ b + \alpha(c-b),\ c)$$

which is a refinement of zeroinl.15.

Note that zeroinl.18 may exchange b,c depending on abs(f(b)) and abs(f(c)). Thus, the (b,c) projection of the function computed by zeroinl.15-18 in this case is

$$b,\ c := \left\{\begin{array}{l} b + (c-b)/2 \\ b + \text{tol 1} \end{array}\right\},\ b \text{ or } b,\quad c := b, \left\{\begin{array}{l} b + (c-b)/2 \\ b + \text{tol 1} \end{array}\right\}$$

i.e., the new interval (b, c) is the half interval of the initial (b_0, c_0) which includes b_0 (for increments greater than tol 1), and the new b is chosen to minimize the value abs(f(b)). The result of iterating this dopart is unpredictable unless more is known about the values of f. For example, if the values of f in (ax, bx) are of one sign and monotone increasing or decreasing, then the iteration will go to the end point ax or bx for which abs(f) is minimum. In general, the iteration will tend toward a minimum for abs(f), but due to the bisecting behavior, no guarantees are possible.

Case 5: part 2. This covers the happy accident of some intermediate pair b,c bracketing an odd number of zeroes of f by happening into values b_k, c_k, such that $f(b_k) * f(c_k) \leq 0$. The tendency to move towards a minimum for abs(f(b)) may increase the chances for such a happening, but provide no guarantee. Once such a pair b_k, c_k is found, case 4 applies and some zero will be approximated.

This completes the proof of the theorem except for the proofs of the three lemmas used in the proofs which follow directly.

<u>Lemma 15-18</u> The invariant I defined as

$$I \equiv (a = c \neq b \vee a < b < c \vee c < b < a) \wedge f(b) * f(c) \leqslant 0 \wedge abs(f(b)) \leqslant abs(f(c))$$

is preserved by the execution of the loop body ZEROIN1.15-18.

<u>Proof</u>: We use the following abbreviations:

$P \equiv abs(f(b)) \neq 0 \wedge abs((c-b)/2) > 2 * eps * abs(b) + tol/2$

$I_o \equiv ((c < b) \vee (c > b)) \wedge f(b) * f(c) < 0$

$I_1 \equiv (a < b < c \vee c < b < a) \wedge f(a) * f(c) < 0$

$I_2 \equiv (a = c \neq b \vee a < b < c \vee c < b < a) \wedge f(b) * f(c) \leqslant 0.$

Note that P is the loop predicate. The validity of the Lemma is an immediate consequence of the following conditions:

C1 : $I \wedge P \Longrightarrow I_o$

C2 : I_o {ZEROIN1.15} I_1

C3 : I_1 {ZEROIN1.16} I_2

C4 : I_2 {ZEROIN1.18} I

Condition C1 is straightforward. C2 can be seen by considering c < b and c > b as different input cases. Condition C3 follows from

$I_1 \wedge f(b) * f(c) > 0$ {c := a} I_2 (note that setting c = a changes the sign of f(c))

$I_1 \wedge f(b) * f(c) \leqslant 0 \Longrightarrow I_2$

Similarly, C4 can be inferred from

$I_2 \wedge abs(f(c)) < abs(f(b))$ {a, b, c := b, c, b} I

$I_2 \wedge abs(f(c)) \geqslant abs(f(b)) \Longrightarrow I.$

<u>Lemma 15-18T</u> Given b_o, c_o on entry to zeroinl.15-18 then for some α, $0<\alpha<1$

after zeroinl.15 $abs(c-b) = (1-\alpha)\ abs(c_o-b_o)$

after zeroinl.16 $abs(c-b) \le abs(c_o-b_o)\ \max(\alpha, 1-\alpha)$

after zeroinl.18 $abs(c-b) \le abs(c_o-b_o)\ \max(\alpha, 1-\alpha)$

<u>proof</u>: after zeroinl.15

$$abs(c-b) = abs(c_o-b_o-\alpha(c_o-b_o)) = abs(c_o-b_o)(1-\alpha) \quad 0<\alpha<1$$

$$abs(b-a) = abs(b_o+\alpha(c_o-b_o) - b_o) = abs\ \alpha(c_o-b_o) \quad 0<\alpha<1$$

after zeroinl.16

$$abs(c-b) \le \max\ abs(c_o-b_o)\ (1-\alpha),\ abs(c_o-b_o)\alpha)$$

$$\le abs(c_o-b_o)\ \max\ (\alpha, 1-\alpha)$$

after zeroinl.18

$abs(c-b) \le abs(c_o-b_o)\ \max\ (\alpha,1-\alpha)$ since b and c are unchanged or exchanged.

It should be noted that in the above discussion, zeroinl.17 was ignored because its effect on the calculation of the root and termination of the loop is irrelevant.

We have one last lemma to prove.

<u>Lemma 15L</u> Given a = c and f(a) * f(b) > 0 then zeroinl.15 calculates the new b using the bisection method, i.e.,

$$b := b + \left\{\begin{array}{ll} (b-c)/2 & \text{if } abs(c-b) > tol\ 1 \\ tol\ 1 & \text{otherwise} \end{array}\right\}$$

<u>proof</u>:

From PDL 43, either abs(f(b)) < abs(f(a)) or bisection is done (PDL 45) with d = xm = (c-b)/2. Then PDL 82-91 implies

$$b := \left\{\begin{array}{ll} b + d = b + (c-b)/2 & \text{if } abs(c-b)/2 > tol\ 1 \\ b + tol\ 1 & \text{otherwise} \end{array}\right\}$$

Since by hypothesis a = c, PDL 49 implies inverse quadratic interpolation is not done and linear interpolation (PDL 56) is attempted. Thus

s = fb/fa and 0 < s < 1 since fb * fa > 0 and abs(fb) < abs(fa)

p = (c-b) * s, using xm + (c-b)/2

q = 1-s, implying q > o in PDL 59

The proof will be done by cases on the relationship between b and c.

<u>c > b</u>

c > b implies p > 0 in PDL 58. Since p > 0 before PDL 62, PDL 65 sets q to -q, so q < 0. Then the test at PDL 70 is true since

2 * p = a. * s is positive,

3.0 * xm * q = $\frac{3}{2}$ (c-b) * q is negative, and

abs(tol 1 * q) is positive

implying PDL 70 evaluates to true

and bisection is performed in PDL 72-73.

<u>c < b</u>

c < b implies p < 0 in PDL 58. Since p < 0 before PDL 62, PDL 65 leaves q alone and PDL 67 sets p > 0 implying p = (b-c) * x. Then the test at PDL 70 is true since

2 * p = 2 * (b-c) * s is positive,

3.0 * xm * q = $\frac{3.}{2.}$ (c-b) * q is negative, and

abs(tol 1 *q) is positive

implying PDL 70 evaluates to true

and bisection is performed in PDL 72-73.

IV. CONCLUSION

Answering the questions - We can now answer the questions originally posed by Professor Vandergraft.

Question 1:

If the equation is linear, the program will do a linear interpolation and find the root on one pass through the loop, except in the case where the size of the interval (a, b) is smaller than tol 1. Then it will do a bisection (from the test at PDL 43). Note the other potential condition where it may pass to PDL 44 for bisection is if abs(fa) - abs(fb) (from PDL 19, 26, and 43). However, in this case bisection is an exact solution. The case that the size of the interval is smaller than tol 1 is unlikely, but can happen.

Question 2:

The theorem states that if f(a) and f(b) are both of the same sign, we will get an answer that is some point between a and b even though there is no root in the interval (a, b) (case 5a of the Theorem). If there are an even number of roots in the interval (a, b) then it is possible the program will happen upon one of the roots and return that root as an answer (case 5b of the Theorem). To check for this condition, we should put a test right at entry to the program between PDL 3 and PDL 4 of the form.

```
if
   f(a) * f(b) > 0
then
   write ('F(A) and F(B) ARE BOTH OF THE SAME SIGN, RETURN B')
else
   PDL 4-102
fi
```

Question 3:

It would be easy to remove the inverse quadratic interpolation part of the code. We can do this simply by removing several PDL statements, i.e., PDL 47-55. However, this would not leave us with the best solution since much of the code surrounding the inverse quadratic interpolation could be better written. For example:

(1) there would be no need to keep a, b, and c

(2) the test in PDL 70 could be removed if we checked in the loop that f(a) * f(b) was always greater than zero, since bisection and linear interpolation would never take us out of the interval.

Cleaning up the algorithm would probably require a substantial transformation.

Question 4:

Zeroin will find a triple root. It will not inform the user that it is a triple root, but will return it as a root because once it has a root surrounded by two points such that f(a) and f(b) are of opposite signs, it will find that root (case 4 of the Theorem).

Program history - Since most programs seen by practicing programmers do not have a history in the literature, we did not research the history of ZEROIN until we had completed our experiment. The complexity of the program is partially due to the fact that it was modified over a period of time by different authors, each modification making it more efficient, effective or robust. The code is based on the secant method (Ortega and Reinboldt). The idea of combining it with bisection had been suggested by several people. The first careful analysis seems to have been by T. J. Dekker (Dekker).
R. P. Brent (Brent) added to Dekker's algorithm the inverse quadratic interpolation option, and changed some of the convergence tests. The Brent book

contains an ALGOL 60 program. The FORTRAN program of Figure 1 is found in (Forsythe, Malcolm & Moler) and is a direct translation of Brent's algorithm, with the addition of a few lines that compute the machine-rounding error. We understand that ZEROIN is a significant and actively used program for calculating the roots of a function in a specific interval to a given tolerance.

Understanding and documenting - As it turns out, we were able to answer the questions posed and discover the program function of ZEROIN. The techniques used included function specification, the discovery of loop invariants, case analysis, and the use of a bounded indeterminate auxiliary variable. The discovery process used by the authors was not as direct as it appears in the paper. There were several side trips which included proving the correctness of the inverse quadratic interpolation (an interesting result but not relevant to the final abstraction or the questions posed).

There are some implications that the algorithm of the program was overdesigned to be correct and that the tests may be more limiting than necessary. This made the program easier to prove correct, however.

We believe this experience shows that the areas of program specification and program correctness have advanced enough to make them useful in understanding and documenting existing programs, an extremely important application today. In our case, we are convinced that without the focus of searching for a correctness proof relating the specification to the program, we would have learned a great deal, but would have been unable to record very much of what we learned for others.

Hamming pointed out that mathematicians and scientists stand on each other's shoulders, but programmers stand on each other's toes. We believe that will continue to be true until programmers deal with programs as mathematical objects, as unlikely as they may seem to be in real life, as we have tried to do here.

References

[Brent] R. P. Brent. ALGORITHMS FOR MINIMIZATION WITHOUT DERIVATIVES, Prentice-Hall, 1973.

[Dahl, Dijkstra, and Hoare] O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare. STRUCTURED PROGRAMMING, Academic Press, 1972.

[Dekker] T. J. Dekker. FINDING A ZERO BY MEANS OF SUCCESSIVE LINEAR INTERPOLATION, published in "Constructive Aspects of the Fundamental Theorem of Algebra," B. Dejou and P. Henrici, editors, Interscience, 1969.

[Dijkstra] E. W. Dijkstra. A DISCIPLINE OF PROGRAMMING, Prentice-Hall, 1976.

[Forsythe, Malcolm, and Moler] G. Forsythe, M. Malcolm, and M. Moler. COMPUTER METHODS FOR MATHEMATICAL COMPUTATIONS, Prentice-Hall, 1977.

[Linger, Mills, and Witt] R. C. Linger, H. D. Mills, and B. I. Witt. STRUCTURED PROGRAMMING THEORY AND PRACTICE, Addison-Wesley, 1979.

[Ortega and Rheinboldt] James Ortega and W. C. Rheinboldt. ITERATIVE SOLUTION OF NONLINEAR EQUATIONS IN SEVERAL VARIABLES, Academic Press, 1970.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE *(and Subtitle)* UNDERSTANDING AND DOCUMENTING PROGRAMS		5. TYPE OF REPORT & PERIOD COVERED Technical Report
		6. PERFORMING ORG. REPORT NUMBER TR-884
7. AUTHOR*(s)* Victor R. Basili and Harlan D. Mills		8. CONTRACT OR GRANT NUMBER*(s)* AFOSR-77-3181B
9. PERFORMING ORGANIZATION NAME AND ADDRESS Department of Computer Science University of Maryland College Park, Maryland 20742		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Math. & Info. Sciences, AFOSR Bolling AFB Washington, D. C. 20332		12. REPORT DATE May 1980
		13. NUMBER OF PAGES 29 plus 15 pp. of Figures
14. MONITORING AGENCY NAME & ADDRESS*(if different from Controlling Office)*		15. SECURITY CLASS. *(of this report)* UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

18. SUPPLEMENTARY NOTES

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

program maintenance, correctness, documentation, specification

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

This paper reports on an experiment in trying to understand an unfamiliar program of some complexity and to record the authors' understanding of it. The goal was to stimulate a practicing programmer in a program maintenance environment using the techniques of program design adapted to program understanding and documentation; that is, given a program, a specification and correctness proof were developed for the program. The approach points out the value of correctness proof ideas in guiding the discovery process. Toward this end, a variety of techniques were used: direct cognition for smaller parts, discovering and

DD FORM 1 JAN 73 1473

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)* verifying loop invariants for larger program parts, and functions determined by additional analysis for larger program parts. An indeterminate bounded variable was introduced into the program documentation to summarize the effect of several program variables and simplify the proof of correctness.