Copyright © 1973, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

INTRODUCTION TO PROGRAMMING SCIENCE, PART II:

PROOFS OF ASSERTIONS ABOUT PROGRAMS

by

W. D. Maurer

Memorandum No. ERL-394

21 August 1973

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

PART II

CONTENTS

The research reported here was partially supported by National Science Foundation Grant GJ-31612.

CHAPTER SIX

PARTIAL CORRECTNESS

6-1 The Assertion Method

What does it mean for a program to be correct? To illustrate what correctness is, and how we prove it for programs written in algorithmic languages, let us consider a program for Euclid's algorithm.

Suppose we want to find the greatest common divisor of 16 and 10. The divisors of 16 are 1, 2, 4, 8, and 16; the divisors of 10 are 1, 2, 5, and 10. The common divisors are 1 and 2, and the greatest of these, of course, is 2. Euclid's algorithm, in its original form, is a method of obtaining the answer (2 in this case) by successive subtraction. At each stage, we subtract the smaller of the two numbers we have from the larger, and then cross out the larger number, leaving us again with two numbers. Thus, if we start with the two numbers

16 10

the first step gives us

~~16~~ 10 6

leaving us with 10 and 6, so that the second step gives us

~~16~~ ~~10~~ 6 4

and the next two steps give

~~16~~ ~~10~~ ~~6~~ 4 2

~~16~~ ~~10~~ ~~6~~ ~~4~~ 2 2

The process stops when we have two numbers that are the same.

Why does this work? Let us look at the second stage, where our numbers are 10 and 6. The greatest common divisor of 10 and 6 is the same as the greatest common divisor of 16 and 10. Indeed, this illustrates a general fact: If $I > 0$, $J > 0$, and $I-J > 0$, then $GCD(I, J) = GCD(I-J, J)$. Thus, at each stage, we have two numbers whose greatest common divisor remains the same. At the end, we have two numbers that are equal; say they are both equal to I, then $GCD(I, I) = I$.

First let us prove the above mathematical properties of GCD. Let x be a common divisor of I and J, so that $I = ax$, $J = bx$. Then $I-J = (a-b)x$, so that x is a divisor of I-J. Conversely, if x is a common divisor of I-J and J, then $I-J = cx$ and $J = bx$, so that $I = (c+b)x$, and x is a divisor of I. This shows that the set of all common divisors of I and J is the same set as the set of all common divisors of I-J and J (so surely the greatest element of one set is the greatest element of the other). Our other fact, $GCD(I, I) = I$, is clear since $I = 1 \cdot I$ and no integer larger than I may divide I.

Now let us write a program to implement this algorithm:

```
      I = M
      J = N
1     IF (I-J) 2, 4, 3
2     J = J - I
      GO TO 1
```

```
3       I = I-J
        GO TO 1
4       CONTINUE
```

At statement number 1, we test which of our two current quantities, I and J, is the larger. If they are the same, we stop (by transferring to 4). If I is larger, we subtract J from it, and then "cross it out" by giving it this new value. If J is larger, we do the reverse.

Our first step in proving the correctness of this program is to say precisely what it does. It is not enough to say that it calculates the GCD of M and N; we also have to say where the result will be found. In this case, the result will be the final value of I (and also of J, but we can ignore this fact if we want to). We say that the final assertion of this program is I = GCD(M, N). In general, an assertion is a condition, or relation involving the variables of the program; this assertion is called final because it is true when the program is finished.

Next we must ask whether this program always works, or whether there are any initial conditions that must be true in order for the program to work. In this case, there are indeed certain initial conditions. We are finding the greatest common divisor of M and N, and therefore M and N must be positive integers. (In fact, if either M or N is zero, for example, the program goes into an endless loop, and thus cannot be correct.) We say that our initial assertion is (M > 0 and N > 0). We are now able to make a precise mathematical statement of the correctness of this program:

If the program is started at the beginning, and if, at that time, M > 0 and N > 0, then it will terminate (that is, it will not go into an endless loop), and, when it terminates, we will have

I = GCD(M, N).

In general, the statement of correctness of any program will be of this form; it will involve a specific starting point (which is not always at the beginning of the program), an initial assertion, and a final assertion.

The next step in the proof is to construct intermediate assertions. An intermediate assertion is an assertion which must be true every time the program reaches a certain statement in our program, called a control point. In this case, we choose one control point, namely statement number 1. Every time the program reaches this statement, we will have

$$(GCD(I, J) = GCD(M, N) \text{ and } I > 0 \text{ and } J > 0)$$

(just before the statement is executed). That is, the two intermediate quantities, I and J, will be positive, and their greatest common divisor will be the same as the greatest common divisor of our original two quantities, M and N.

Learning how to find the proper intermediate assertions in a program is very much like learning programming itself. It is necessary to determine informally why the program works, and then to express this in the form of conditions which must hold every time we pass some point in the program. In this case, we determined that the GCD of our two intermediate quantities would remain constant. Our mathematical statement GCD(I, J) = GCD(I-J, J), which will be used in the proof, depends on the hypotheses I > 0 and J > 0, and, anticipating this, we have included these as part of our intermediate assertion (since it is, at any rate, clear that they are always true whenever the program passes the given point). Learning where to put intermediate assertions, on the other hand, is a quite mechanical process, which we shall take up in sections

7-5 and 8-3.

We rewrite the program with our three assertions -- the initial, intermediate, and final assertions -- given as comments, and with a statement number on every statement:

```
C  M > 0, N > 0
5      I = M
6      J = N
C  GCD(I, J) = GCD(M, N), I > 0, J > 0
1      IF (I-J) 2, 4, 3
2      J = J - I
7      GO TO 1
3      I = I - J
8      GO TO 1
C  I = GCD(M, N)
4      CONTINUE
```

Note that we may think of the initial and the final assertions, as well as the intermediate assertion, attached to control points, namely the beginning and the end of the program.

Next we determine the control paths. A control path is a sequence of statements which the program executes as it goes from one control point to the next. The initial assertion and the final assertion of a control path are those assertions which have been associated with the beginning and the end of the path, respectively. The verification condition of a control path may be stated as follows:

If the path is started with its initial assertion valid, and if the path is actually followed, then, when it is finished, its final assertion will be valid.

Let us see what this means for our first control path:

```
C   M > 0, N > 0
5       I = M
6       J = N
C   GCD(I, J) = GCD(M, N), I > 0, J > 0
```

Is the verification condition valid? In other words: If M > 0 and N > 0, and if we then set I equal to M and J equal to N, will GCD(I, J) be equal to GCD(M, N), and will I and J be greater than zero? Clearly yes. In fact

GCD(I, J) = GCD(M, N)	follows from	I = M and J = N;
I > 0	follows from	I = M and M > 0;
J > 0	follows from	J = N and N > 0.

Our second control path is

```
C   GCD(I, J) = GCD(M, N), I > 0, J > 0
1       IF (I-J) 2, 4, 3
2       J = J - I
7       GO TO 1
C   GCD(I, J) = GCD(M, N), I > 0, J > 0
```

(Recall that our intermediate assertion is supposed to hold just <u>before</u> statement number 1 is executed. Thus statement number 1 occurs at the beginning of this path, but not at the end.) If the intermediate assertion of our program is true at the beginning of this path, is it true at the end?

Clearly I > 0 will be true at the end, because I > 0 was true at the beginning, and we never changed the value of I. Also, J > 0 will be true at the end, because we set J equal to the positive

quantity J-I. (We know that J-I is positive, because we do not take this path unless I-J is negative -- we would transfer to statement 3 or 4, rather than 2.) To verify GCD(I, J) = GCD(M, N), let us set up a table of variables and their values:

Variable	I	J	M	N
Initial value	$\underline{i}$	$\underline{j}$	$\underline{m}$	$\underline{n}$
Final value	$\underline{i}$	$\underline{j}-\underline{i}$	$\underline{m}$	$\underline{n}$

In particular, if $\underline{i}$ and $\underline{j}$ are the initial values of I and J, then $\underline{j}-\underline{i}$ is the final value of J. We must show that if GCD(I, J) = GCD(M, N) was true at the beginning, then it will be true at the end. That is, we must show that

$$\text{GCD}(\underline{i}, \underline{j}) = \text{GCD}(\underline{m}, \underline{n}) \quad \text{implies} \quad \text{GCD}(\underline{i}, \underline{j}-\underline{i}) = \text{GCD}(\underline{m}, \underline{n})$$

This will clearly be true if we can show that

$$\text{GCD}(\underline{i}, \underline{j}) = \text{GCD}(\underline{i}, \underline{j}-\underline{i})$$

and this follows, as we have seen, provided that $\underline{i} > 0$, $\underline{j} > 0$, and $\underline{j}-\underline{i} > 0$. The first two of these follow immediately from the intermediate assertion; while the fact that $\underline{j}-\underline{i}$ is positive has already been shown and used in connection with the previous path.

What about the third path? We may write this as

```
C  GCD(I, J) = GCD(M, N), I > 0, J > 0
1      IF (I-J) 2, 4, 3
C  I = GCD(M, N)
```

This is the path involving the transfer to statement number 4. If this path is taken, then I and J must have been equal when the path started. But in that case, since GCD(I, I) = I, we must have

had I = GCD(M, N) at the beginning of the path. This means that we will still have I = GCD(M, N) at the end, since none of the variables have their values changed.

We pass over the proof of the verification condition of the fourth path, which is

```
C  GCD(I, J) = GCD(M, N), I > 0, J > 0
1      IF (I-J) 2, 4, 3
3      I = I - J
8      GO TO 1
C  GCD(I, J) = GCD(M, N), I > 0, J > 0
```

since this is exactly the same as the second path if we interchange the rôles of I and J, and also the rôles of M and N.

What can we now show by induction? Suppose that the program is started at the beginning, and suppose that its initial assertion is valid. The first time it gets to a control point, the assertion at that point will be true, since it can only get to a control point by following a control path, and we have shown for every control path that its verification condition holds. In fact, no matter how the program twists and turns, every time it passes a control point, the associated assertion will be true. This is, in particular, true of the final assertion, if we ever get to the end of the program -- which we may not. In other words, we have now proved that either the program is correct or it goes into an endless loop. Thus there is one final stage to our proof -- to prove that the program terminates -- and that will be done in section 8-1.

A program is partially correct if it is either correct or endless. Thus our GCD program has now been proved partially correct.

6-2 Justification of the Method

The proof given above is typical of the way in which we normally prove the partial correctness of a program. There remain, however, certain questions about its validity, or the validity of any other such proof. Some of these questions are concerned with the fact that assignment statements, such as I = M, mean different things in different situations; others are concerned with the variety of finite-precision results which are obtained on various computers. This second type of objection will be met in sections 9-4 and 9-5, after we have introduced the notion of a restricted command; for the moment, let us confine ourselves to the first type of objection.

Consider the statement I = M. What does it mean? We have assumed that it means "set the new value of I to be the current value of M." In FORTRAN, this is all that it could mean; but, even in FORTRAN, there are difficulties. For example, what if our program includes a declaration of the form REAL I, J, M, N ? Nowhere in the proof have we required M and N to be integers. Yet if they are not integers, the result produced will be meaningless. Another peculiar situation arises if I and M are formal parameters to a subroutine, and the corresponding actual parameters are the same. If we call F(K, K), where our GCD program is embedded in a subroutine defined by writing SUBROUTINE F(I, M), then I = M sets K equal to itself. This would nullify our proof of partial correctness.

More obvious problems arise if the GCD program is written in some other language than FORTRAN. It should be clear that our

assertion method is not specific to any particular language, and it is easy to see how the same sort of arguments given in our proof might be applied to a PL/I program, an ALGOL program, etc. But in these languages, the assignment I = M, or its analogue, might have completely different meanings. In PL/I, I = M might be an array operation, setting I(k) equal to M(k) for every value of k as determined by the declarations of I and M. In ALGOL, where we would write I := M, it is quite possible for M to be the name of a function having no parameters, and this function might, for example, have the side effect of setting N to zero, which would cause the algorithm to loop endlessly under all conditions whatsoever.

It is therefore clear that the validity of a proof of partial correctness of any program depends in an essential way upon the meaning of the statements in that program. The "meaning" of a statement, however, is, more precisely, its semantics. As we have seen in Part I, if we define the semantics of a language, then every executable statement in that language, and every assertion which is expressible in it, has certain semantic attributes which are state vector functions. In order to resolve difficulties of this type, therefore, we must determine the relations between these semantic attributes and the verification conditions which we have defined.

Let us consider the first control path in our GCD program:

```
C  M > 0, N > 0
5      I = M
6      J = N
C  GCD(I, J) = GCD(M, N), I > 0, J > 0
```

We have seen in Chapter 2 that an expression such as (M > 0 and N > 0) has a value which is a function of the state vector, and, since this is a Boolean expression, the value is either true or false (of any state vector). We have also seen, in Chapter 3, that a statement such as I = M has an effect, which takes the old state vector into the new state vector. Similarly, J = N has an effect, while (GCD(I, J) = GCD(M, N) and I > 0 and J > 0) has a value. If these effects and values are properly defined, we may express the verification condition for this path in terms of them.

Let P be the expression (M > 0 and N > 0), and let v_P be its value. Let Q_1 be the statement I = M, and let e_1 be its effect; let Q_2 be the statement J = N, and let e_2 be its effect. Finally, let R be the expression (GCD(I, J) = GCD(M, N) and I > 0 and J > 0), and let v_R be its value. The verification condition may be informally stated as "If P is true, and Q_1 and Q_2 are then executed, then R will be true." If P is true, then $v_P(S) =$ true, where S is the current state vector. The next state vector, after I = M is executed, is $e_1(S)$; after J = N is executed, the state vector is $e_2(e_1(S))$. At this point R must be true, that is, we must have $v_R(e_2(e_1(S))) =$ true. Thus the verification condition may be formally stated as

$$(v_P(S) = \text{true}) \text{ implies } (v_R(e_2(e_1(S))) = \text{true})$$

for any state vector S.

What are v_P, e_1, e_2, and v_R? We have made certain unstated assumptions, in the proof of our GCD program, about the meanings of its various components. Let us now state these assumptions by making plausible formal definitions of the above functions:

$$v_P(S) = \begin{cases} \underline{\text{true}} & \text{if } S(M) > 0 \text{ and } S(N) > 0 \\ \underline{\text{false}} & \text{otherwise} \end{cases}$$

$e_1(S) = S'$, where $S'(I) = S(M)$

$S'(z) = S(z)$ for $z \neq I$

$e_2(S) = S'$, where $S'(J) = S(N)$

$S'(z) = S(z)$ for $z \neq J$

$$v_R(S) = \begin{cases} \underline{\text{true}} & \text{if } GCD(S(I), S(J)) = GCD(S(M), \\ & \quad S(N)) \text{ and } S(I) > 0 \text{ and } S(J) > 0 \\ \underline{\text{false}} & \text{otherwise} \end{cases}$$

The validity of the verification condition is now easily proved. Suppose that $v_P(S) = \underline{\text{true}}$; that is, $S(M) > 0$ and $S(N) > 0$. We must show $v_R(e_2(e_1(S))) = \underline{\text{true}}$. This reduces to

$$GCD(S''(I), S''(J)) = GCD(S''(M), S''(N)) \ \underline{\text{and}} \ S''(I) > 0 \ \underline{\text{and}} \ S''(J) > 0$$

where $S'' = e_2(e_1(S))$. Writing $S' = e_1(S)$, so that $S'' = e_2(S')$, we have $S''(J) = S'(N) = S(N)$ and $S''(I) = S'(I) = S(M)$. Therefore the above equation reduces to

$$GCD(S(M), S(N)) = GCD(S(M), S(N)) \ \underline{\text{and}} \ S(M) > 0 \ \underline{\text{and}} \ S(N) > 0$$

which reduces immediately to the original hypothesis.

We have shown that if v_P, e_1, e_2, and v_R are defined as above, then the proof of validity of the given verification condition is mathematically rigorous. The definitions of these functions, however, may be obtained in an equally rigorous manner from the semantic rules of the language in which this program is written. If there are side effects or array declarations which give rise to inherited attributes that cause the function definitions to be different from those given above, the verification condition may pos-

sibly be invalid. In that case, we are often able to exhibit a particular state vector S for which the condition is invalid.

Let us do one more example. Here is the second control path of our GCD program:

```
C  GCD(I, J) = GCD(M, N), I > 0, J > 0
1     IF (I-J) 2, 4, 3
2     J = J - I
7     GO TO 1
C  GCD(I, J) = GCD(M, N), I > 0, J > 0
```

We shall use the definition of the value v_R of the condition R as in the preceding path. If Q is the statement J = J - I, then its effect e_Q will be defined as

$$e_Q(S) = S', \text{ where } \quad S'(J) = S(J) - S(I)$$
$$S'(z) = S(z) \text{ for } z \neq J$$

The effect of the GO TO statement is the identity function, and thus may be ignored.

Our informal statement of the verification condition here is: If R is true, and the given path is followed, then R will again be true at the end of the path. Now when is the given path followed? Only if I-J < 0 at the beginning of the path. The condition I-J < 0 also has a value which is a function of the state vector. If we denote this condition by C and its value by v_C, then we may take

$$v_C(S) = \begin{cases} \underline{\text{true}} & \text{if } S(I) - S(J) < 0 \\ \underline{\text{false}} & \text{otherwise} \end{cases}$$

What is our formal statement of the verification condition? Let S be the current state vector at the start of the path. Our hypothe-

sis is that R is initially true, that is, $v_R(S) = \underline{\text{true}}$. Another hypothesis is that the given path is actually followed; and, for this to happen, we must have $v_C(S) = \underline{\text{true}}$ also. At the end of the path, the current state vector will be $e_Q(S)$, and R must be true at this time. Thus the verification condition is

$$(v_R(S) = \underline{\text{true}} \text{ and } v_C(S) = \underline{\text{true}}) \underline{\text{implies}} (v_R(e_Q(S)) = \underline{\text{true}})$$

Again, this is easily proved. Suppose that $v_R(S) = \underline{\text{true}}$ and $v_C(S) = \underline{\text{true}}$. Then GCD(S(I), S(J)) = GCD(S(M), S(N)), S(I) > 0, and S(J) > 0, and also S(I) - S(J) ≤ 0. We must show that

$$\text{GCD}(S'(I), S'(J)) = \text{GCD}(S'(M), S'(N)) \text{ and } S'(I) > 0 \text{ and } S'(J) > 0$$

where $S' = e_Q(S)$. By the definition of e_Q, we have

$$S'(I) = S(I)$$
$$S'(J) = S(J) - S(I)$$
$$S'(M) = S(M)$$
$$S'(N) = S(N)$$

so that the above equation reduces to

$$\text{GCD}(S(I), S(J) - S(I)) = \text{GCD}(S(M), S(N)) \text{ and } S(I) > 0$$
$$\text{and } S(J) - S(I) > 0$$

The hypothesis directly implies S(I) > 0 and S(J) - S(I) > 0. The hypothesis, together with the condition

$$(a > 0, b > 0, b-a > 0) \text{ implies } \text{GCD}(a, b) = \text{GCD}(a, b-a)$$

with a = S(I) and b = S(J), imply GCD(S(I), S(J) - S(I)) = GCD(S(I), S(J)) = GCD(S(M), S(N)). But this last condition may be proved for all integers a and b, using the definition of the

GCD function, just as in the preceding section. This completes the proof of validity of this verification condition.

In the remainder of Part II, the programs with which we will be dealing are assumed to contain assignments, transfer statements, conditionals, and assertions whose semantics is obtained in roughly the same manner as we have done in these examples. In particular, side effects will be assumed absent, and each identifier will be presumed to refer to a single variable. Treatment of more general cases will be taken up in Part III.

A different kind of question about the assertion method concerns its universality, rather than its validity. Is it really true that the correctness of an arbitrary program may be expressed in terms of initial and final assertions? There is, in particular, one type of program for which the expression of its correctness in this way involves the introduction of extra variables. Consider, for example, the usual method of interchanging the values of two algebraic-language variables X and Y:

```
TEMP = X
X = Y
Y = TEMP
```

Suppose we treat this as a three-statement program. What is its final assertion? When the program is finished, X is equal to the initial value of Y, and Y is equal to the initial value of X. We cannot express this, however, by writing (X = Y and Y = X).

Let us denote the initial value of X by X0 and the initial value of Y by Y0. Then we may write the final assertion X = Y0 and Y = X0. The fact that X0 is the initial value of X may be expressed by writing X = X0 as an initial assertion -- and similarly

for YO. Hence our program, expressed as a single control path with initial and final assertion, is

```
C  X = XO, Y = YO
       TEMP = X
       X = Y
       Y = TEMP
C  X = YO, Y = XO
```

It is assumed here that XO and YO are identifiers which do not appear as variable names in the program containing the above three statements; clearly such identifiers may always be chosen.

In general, suppose that we have a program which calculates the new values of certain variables x_1, x_2, ..., as functions of their old values. We may always give initial and final assertions for such a program by introducing new variables y_1, y_2, ..., and writing $x_1 = y_1$, $x_2 = y_2$, ... as initial assertions and $x_1 = f_1(y_1, y_2, \ldots)$, $x_2 = f_2(y_1, y_2, \ldots)$, ... as final assertions. Such programs are very common; we could obtain one from our GCD program, in fact, by simply omitting the first two statements. The resulting program would set I equal to the GCD of the initial values of I and J; we could write I = GCD(IO, JO) as the final assertion and (I = IO and J = JO and I > 0 and J > 0) as the initial assertion. Even the one-statement "program"

```
I = I + 1
```

should be treated in this way; the initial assertion is I = IO and the final assertion is I = IO + 1. A more practical example is the sorting of an array in place, to be treated in section 10-3.

6-3 Control Path Determination

Given a program with control points and assertions, how do we determine all of its control paths? In a small program, it is easy to check informally that we have them all; for a larger program, we need an algorithmic procedure. We shall now give such a procedure, and at the same time, while discussing it, we shall be able to gain further insight into the problem of where control points should be placed in a program (see also section 7-5).

Before defining our algorithm formally, we shall examine how it operates on the following sample program:

```
C  K < 20
       J = K - 10
1      IF (J .GE. 0) GO TO 2
       KX = 5
       GO TO 99
2      K = 0
3      IF (N) 98, 6, 4
98     KX = 6
       GO TO 99
4      M = N
C  K = (N*(N+1)/2) - (M*(M+1)/2)
5      K = K + N
       M = M - 1
       IF (M .NE. 0) GO TO 5
6      KX = 1
C  KX ≠ 1 or K = N*(N+1)/2
99     CONTINUE
```

Our algorithm starts by finding the first control point in the program, which in this case is at the beginning of the program. It will initially find all control paths which start at this first control point. It keeps, at all times, a current path containing zero or more statements, and a stack which keeps track of conditional statements. The current path is initialized to contain no statements, and the stack is initialized to be empty.

The first statement we encounter is J = K - 10. This is an assignment statement, so we simply add it to the current path. Now comes IF (J .GE. 0) GO TO 2. Clearly there are two classes of paths -- those which go to 2 at this point and those which do not. We will first treat those which go to 2. If a path goes to 2, then clearly $J \geq 0$ at this point in that path. Therefore we add the condition $J \geq 0$ to our current path, which now reads

$$J = K - 10$$
$$(J \geq 0)$$

Note that we have abbreviated our notation for paths. We no longer include statement numbers; also, instead of including an entire IF statement in a path, we include only the condition which must be true if that path is taken.

It is also necessary to record on the stack what we have done. In this case, we put on the stack a pointer to the condition $(J \geq 0)$ in the current path, together with a pointer to statement number 1 in the program.

Now we proceed from statement number 2. The first statement we encounter is K = 0; as before, this is simply added to our current path. Now we come to IF (N) 98, 6, 4 . As a general rule, when

any statement has several alternative transfer points, we start by considering the first of these. If transfer is made to statement number 98, then we must have $N < 0$ at this point, and this is the condition we put in the current path. At the same time, we put a pointer to this condition on the stack, together with a pointer to statement number 3 in the program. Our current path and our stack are now as follows:

CURRENT PATH	STACK
$J = K - 10$	Pointer to $(J \geq 0)$ in current path
$(J \geq 0)$	Pointer to statement number 1 in program
$K = 0$	Pointer to $(N < 0)$ in path
$(N < 0)$	Pointer to statement number 3 in program

The next statement is KX = 6, which is added to the current path. Now we come to the statement GO TO 99. In our abbreviated notation for control paths, there is no reason to include GO TO statements; so all we do at this point is to continue processing from statement number 99. When we arrive at that statement, however, we discover that there is an assertion there. Thus the current path is a control path, as follows (with its initial and final assertion):

C $K < 20$

$J = K - 10$

$(J \geq 0)$

$K = 0$

$(N < 0)$

$KX = 6$

C $KX \neq 1$ <u>or</u> $K = N*(N+1)/2$

Having found a control path, our next step is to look at the top of the stack. In this case we have a pointer to (N < 0) and a pointer to statement number 3. We change the condition (N < 0) in the current path to the next alternative condition, which is in this case (N = 0). The pointer to (N < 0) which we had in the stack is therefore now a pointer to (N = 0). Since we also have on the stack a pointer to statement number 3 in our program, we can see where to go if N = 0 -- in this case, to statement number 6. The current path and the stack are now:

CURRENT PATH	STACK
J = K - 10	Pointer to (J ≥ 0) in current path
(J ≥ 0)	Pointer to statement number 1 in program
K = 0	Pointer to (N = 0) in current path
(N = 0)	Pointer to statement number 3 in program

The next statement is KX = 1, which is added to the current path, and now, just as before, we are at statement number 99, which has an associated assertion. Thus the second control path has been found, as follows:

```
C  K < 20
       J = K - 10
       (J ≥ 0)
       K = 0
       (N = 0)
       KX = 1
C  KX ≠ 1 or K = N*(N+1)/2
```

Now, just as before, we look at the top of the stack. The next alternative transfer point is clearly (N > 0), and we change

$(N = 0)$ to $(N > 0)$ in the current path. This is the last alternative transfer point for statement number 3, and because of this we <u>remove</u> the pointer to $(N = 0)$, and the pointer to statement number 3, from the stack entirely. Thus our current path and stack are now:

CURRENT PATH	STACK
$J = K - 10$	Pointer to $(J \geq 0)$ in current path
$(J \geq 0)$	Pointer to statement number 1 in program
$K = 0$	
$(N > 0)$	

The last alternative transfer point of statement number 3 is statement number 4. We proceed there, put $M = N$ on the end of the current path, and now, since statement number 5 is associated with an assertion, we have found the third control path:

```
C  K < 20
       J = K - 10
       (J ≥ 0)
       K = 0
       (N > 0)
       M = N
C  K = (N*(N+1)/2) - (M*(M+1)/2)
```

At this point, let us consider what might have happened if there were no assertion at statement number 5. In that case we would have added $K = K + N$ and $M = M - 1$ to our current path, as well as the condition $M \neq 0$. This would have brought us back to statement 5 again. If we are not careful, our path-construction routine will go into an endless loop! It should be clear, however,

that this is an error case, since it would give us an infinite number of control paths. In fact, for every positive integer n, there would be a control path that went n times around the loop before emerging at statement number 6. The error is found when our current path comes back to a statement that is already on it (unless that statement has an associated assertion).

We proceed with our routine by looking again at the top of the stack. This time we determine that the next alternative condition is (J < 0). Since this is the last of the (two) alternative conditions for statement number 1, we remove two pointers from the stack, as before -- leaving the stack empty. The condition (J < 0) is taken to be the end of the new current path, which is

```
      J = K - 10
      (J < 0)
```

Proceeding as before, we find our fourth path:

```
C   K < 20
        J = K - 10
        (J < 0)
        KX = 5
C   KX ≠ 1 or K = N*(N+1)/2
```

At this point, we try to look at the top of the stack; but the stack is empty. This means that we have finished all paths which start at the first control point. We thus must look through the program until we find the second control point, at which point we proceed, exactly as before, to find all control paths which start there. In this case there are two of them; the first one that we find is

```
C   K = (N*(N+1)/2) - (M*(M+1)/2)
        K = K + N
        M = M - 1
        (M ≠ 0)
C   K = (N*(N+1)/2) - (M*(M+1)/2)
```

and the second is

```
C   K = (N*(N+1)/2) - (M*(M+1)/2)
        K = K + N
        M = M - 1
        (M ≠ 0)
        KX = 1
C   KX ≠ 1 or K = N*(N+1)/2
```

The algorithm now proceeds to the next control point as a starting point. This one, however, corresponds to a terminal point of the program, and thus there are no paths starting there. Since this is the last control point in the program, the algorithm stops.

Let us now specify our algorithm more or less rigorously. In what follows, CS denotes the current statement, and CCP denotes the current control point (that is, the starting point for the current path).

1. Initialize CCP to be the first control point in the program. Initialize the stack to be empty. Initialize the current control path to contain no statements. Mark every statement as not being contained in the current control path.

2. If the current starting point is associated with a terminal node, go to step 6. Otherwise, initialize CS to be the statement associated with CCP.

3. If CS is GO TO n, set CS to be statement number n and skip to step 4. Otherwise, mark the current statement as being in the current path. If CS is an assignment (or function call, input-output statement, etc.), add it to the current path, set CS to be the next statement in the program, and skip to step 4. If CS is not an IF statement, transfer to an error routine. Otherwise, put the first alternative condition of the IF statement in the current path and put a pointer to this condition in the path and to the IF statement in the program on top of the stack.

4. If the current statement has an associated assertion, skip to step 5. Otherwise, if the current statement is already on the current path (that is, if it is marked), transfer to an error location; otherwise, return to step 3.

5. The current path is a control path and may be output or otherwise analyzed. If the stack is currently empty, unmark all statements on the current control path (that is, mark them as not being on that path) and skip to step 6. Otherwise, unmark only those statements on the current control path which follow the condition S to which the top of the stack currently points. By analyzing the IF statement in the program, to which the top of the stack points, find the next alternative transfer point and set CS equal to it; also find the next alternative condition and change S to be that condition. If it is the last alternative, remove the two pointers currently on top of the stack. Now return to step 4.

6. Move CCP forward to the next control point, if this is possible, and return to step 2. If there are no more control points, stop; the algorithm is complete.

6-4 Verification Conditions

Each control path in a program is specified by giving an initial assertion, one or more commands, and a final assertion. We shall now construct algorithmic methods of obtaining the verification condition of a control path from its assertions and commands.

The simplest case we have to consider is that in which there is one command. This command may be either an assignment or a conditional transfer. If it is a conditional transfer, the verification condition is very simple. Since no variables may change their values, the initial assertion and the condition, taken together, must imply the final assertion. A good example of this is the third control path in the GCD program of section 6-1. Here the initial assertion is GCD(I, J) = GCD(M, N) and I > 0 and J > 0, and the condition is I-J = 0. We saw in section 6-1 that these imply the final assertion, I = GCD(M, N), since GCD(I, I) = I.

If the command is an assignment, we may follow the procedure which we used to verify GCD(I, J) = GCD(M, N) in the second control path of the GCD routine. We set up a table of initial and final values of all variables. Since there is only one assignment, only one variable will have its value changed. If the assignment is v=e, where v is a variable and e is an expression, the final assertion is thus altered in a very simple way -- by changing all occurrences of the variable v to the expression e. Consider the following control path:

C GCD(I, J) = GCD(M, N)

I = I - J

C GCD(I, J) = GCD(M, N)

Our table is constructed as follows:

Variable	I	J	M	N
Initial value	i	j	m	n
Final value	$i-j$	j	m	n

The initial assertion becomes GCD(i, j) = GCD(m, n) and the final assertion becomes GCD($i-j$, j) = GCD(m, n). This form of the final assertion could also have been obtained from its original form by substituting I-J for I (and keeping all symbols upper-case).

Let us now see what happens when there can be more than one assignment in the path, as in the first control path of our GCD program:

```
C  M > 0, N > 0
       I = M
       J = N
C  GCD(I, J) = GCD(M, N), I > 0, J > 0
```

There are now two ways of proceeding, known as <u>forward substitution</u> and <u>back substitution</u>.

In the forward substitution method, we use a table as above. We introduce another row in the table which corresponds to the values which the variables assume after the assignment I = M. Our table is as follows:

Variable	I	J	M	N
Initial value	i	j	m	n
Value after I = M	m	j	m	n
Final value	m	n	m	n

The final assertion is now altered as before; its new form is GCD(m, n) = GCD(m, n) <u>and</u> $m > 0$ <u>and</u> $n > 0$, which reduces to simply

m > 0 and n > 0. This is the same as the initial assertion, and thus the verification condition is valid.

In the back substitution method, we proceed from the end of the control path to the beginning. At each stage we modify our assertion by replacing the variable on the left of the equal sign in the assignment by the expression on the right. In this case we start with the final assertion

$$GCD(I, J) = GCD(M, N),\ I > 0,\ J > 0$$

Passing through J = N, we replace each J by an N, obtaining

$$GCD(I, N) = GCD(M, N),\ I > 0,\ N > 0$$

Now passing through I = M, we replace each I by an M, obtaining

$$GCD(M, N) = GCD(M, N),\ M > 0,\ N > 0$$

which reduces to our initial assertion, M > 0 and N > 0. Thus the two methods essentially produce the same result.

Now let us introduce both conditional transfer and assignment commands into a control path, as in the second control path of our GCD program:

```
C   GCD(I, J) = GCD(M, N), I > 0, J > 0
1       IF (I-J) 2, 4, 3
2       J = J - I
        GO TO 1
C   GCD(I, J) = GCD(M, N), I > 0, J > 0
```

The verification condition of such a path may be obtained by a suitable modification of either the forward substitution or the back substitution method.

Using the forward substitution method, whenever we come to a conditional statement, we may assume as a hypothesis that the given condition is true of the current values of the variables. In our example, the condition is $\underline{i}-\underline{j} < 0$. Since the conditional statement in our example is the first statement in the control path, the current values of all variables at that point are their initial values (although, in general, this need not be the case). The condition, with each variable replaced by its current value, now becomes part of the hypothesis. Thus the hypothesis in our example is now

$$GCD(\underline{i}, \underline{j}) = GCD(\underline{m}, \underline{n}),\ \underline{i} > 0,\ \underline{j} > 0,\ \underline{i}-\underline{j} < 0$$

and the conclusion is

$$GCD(\underline{i}, \underline{j}-\underline{i}) = GCD(\underline{m}, \underline{n})$$

This is obtained from the final assertion as before, by substituting for each variable its final symbolic value.

Using the back substitution method, we work with a "current conclusion" and a "current hypothesis." The current conclusion is initialized to the final assertion; the current hypothesis is initialized to be null, or, what is the same, to be always true. Each time we pass a conditional statement, moving backwards through the path, it is added to the current hypothesis. Each time we pass an assignment statement, both the current hypothesis and the current conclusion are modified by replacing the variable on the left side of the assignment symbol by the expression on the right. When we reach the beginning of the path, the initial assertion and the current hypothesis, taken together, must imply the current conclusion.

In our example, we would start with

$$GCD(I, J) = GCD(M, N),\ I > 0,\ J > 0$$

Proceeding through GO TO 1, nothing happens, because a GO TO statement does not change the values of any variables. Proceeding through J = J - I, we replace each J by J-I, obtaining

$$GCD(I, J-I) = GCD(M, N),\ I > 0,\ J-I > 0$$

as before. Passing through I-J < 0, we add this condition to our hypothesis. Since the hypothesis is currently empty, I-J < 0 becomes the hypothesis. We are now at the beginning of the path, and the current hypothesis, together with the initial condition, must imply the current conclusion -- that is,

$$(GCD(I, J) = GCD(M, N),\ I > 0,\ J > 0,\ J-I > 0) \Rightarrow$$
$$(GCD(I, J-I) = GCD(M, N),\ I > 0,\ J-I > 0)$$

The full back substitution process has not been illustrated by this example, since, in it, the only conditional transfer occurs at the very beginning. The following artificially constructed control path will now be used as a further demonstration of both forward and back substitution:

```
C  I ≥ 2
1      K = I
2      IF (K .EQ. M) GO TO 7
3      J = K*K
4      IF (J .GT. N) GO TO 8
5      L = J - K
C  1 ≤ L, L ≤ N-1
```

If forward substitution is used, our table is as follows:

Variable	I	J	K	L	M	N
Initial value	$\underline{i}$	$\underline{j}$	$\underline{k}$	$\underline{l}$	$\underline{m}$	$\underline{n}$
Value after statement 1	$\underline{i}$	$\underline{j}$	$\underline{i}$	$\underline{l}$	$\underline{m}$	$\underline{n}$
Value after statement 2	$\underline{i}$	$\underline{j}$	$\underline{i}$	$\underline{l}$	$\underline{m}$	$\underline{n}$
Value after statement 3	$\underline{i}$	$\underline{i}*\underline{i}$	$\underline{i}$	$\underline{l}$	$\underline{m}$	$\underline{n}$
Value after statement 4	$\underline{i}$	$\underline{i}*\underline{i}$	$\underline{i}$	$\underline{l}$	$\underline{m}$	$\underline{n}$
Final value	$\underline{i}$	$\underline{i}*\underline{i}$	$\underline{i}$	$\underline{i}*\underline{i}-\underline{i}$	$\underline{m}$	$\underline{n}$

The condition $K \neq M$, after statement 1, becomes $\underline{i} \neq \underline{m}$; the condition $J \leq N$, after statement 3, becomes $\underline{i}*\underline{i} \leq \underline{n}$. The initial assertion, of course, becomes $\underline{i} \geq 2$; while the final assertion, evaluated in terms of final values, becomes $1 \leq \underline{i}*\underline{i}-\underline{i}$ and $\underline{i}*\underline{i}-\underline{i} \leq \underline{n}-1$. So our verification condition is:

$$(\underline{i} \neq \underline{m} \text{ and } \underline{i}*\underline{i} \leq \underline{n} \text{ and } \underline{i} \geq 2) \rightarrow (1 \leq \underline{i}*\underline{i}-\underline{i} \text{ and } \underline{i}*\underline{i}-\underline{i} \leq \underline{n}-1)$$

and this is clearly true.

If back substitution is used in this example, we proceed as follows:

Statement	Current hypothesis	Current conclusion
5	None	$1 \leq L$, $L \leq N-1$
4	None	$1 \leq J-K$, $J-K \leq N-1$
3	$J \leq N$	$1 \leq J-K$, $J-K \leq N-1$
2	$K*K \leq N$	$1 \leq K*K-K$, $K*K-K \leq N-1$
1	$K*K \leq N$, $K \neq M$	$1 \leq K*K-K$, $K*K-K \leq N-1$
Start	$I*I \leq N$, $I \neq M$	$1 \leq I*I-I$, $I*I-I \leq N-1$

Adding our initial hypothesis, we obtain the verification condition

$$(I \geq 2 \text{ and } I*I \leq N \text{ and } I \neq M) \Rightarrow (1 \leq I*I-I \text{ and } I*I-I \leq N-1)$$

just as before.

Let us now specify our forward and back substitution methods a bit more formally. (See also section 6-5, where these definitions are modified and extended.) We shall denote the initial assertion by A_i, the final assertion by A_f, and the other things in a control path (each of which is either an assignment or a condition) by C_1, ..., C_n, so that the total number of these is n.

The forward substitution method, as we have seen, keeps a symbolic value for every variable. It makes use of a function SUBSTITUTE(S), where S is any condition or other expression; the value of SUBSTITUTE(S) is the string obtained by substituting, for each variable occurring in S, its current symbolic value. The method proceeds as follows:

1. Initialize I to zero. Initialize H (the hypothesis) to be the initial assertion A_i. Initialize the symbolic value of each variable in the program to be the same as its name.

2. Increase I by 1. If C_I is an assignment V = E, set the new symbolic value of V to be SUBSTITUTE(E). If C_I is a condition K, set H to be (H and SUBSTITUTE(K)).

3. If I = n, stop; the verification condition of the path is (H implies SUBSTITUTE(A_f)). Otherwise, return to step 2.

The back substitution method makes use of a function SUB(S, V, E), whose value is the result of substituting E for V throughout the string S. It proceeds as follows:

1. Initialize I to equal n. Initialize H (the hypothesis) to true. Initialize C (the conclusion) to be the final assertion A_f.

2. If C_I is an assignment V ← E, set C equal to SUB(C, V, E) and set H equal to SUB(H, V, E). If C_I is a condition K, set H equal to (H and K).

3. Decrease I by one. If I = 0, stop; the verification condition is ((A$_i$ and H) implies C). Otherwise, return to step 2.

Note that, in both the forward and the back substitution methods, substitution may involve the insertion of extra parentheses. If the current value of J is I+2, then SUBSTITUTE(J*J) is (I+2)*(I+2), not I+2*I+2 (which would be 3I+2). Similarly, SUB(I*J, J, M+N) is I*(M+N), not I*M+N.

6-5 Subscripted Variables in Assertions

A number of modifications and extensions to our methods must be made if we wish to prove the correctness of programs involving subscripted variables. As an example, consider the following simple program to move an array from one place to another:

```
      K = 0
1     K = K + 1
      B(K) = A(K)
      IF (K .NE. N) GO TO 1
```

The final assertion of this program is that A(I) = B(I) for each value of I, $1 \leq I \leq N$. We shall write this as

$$(A(I) = B(I),\ I\ .IN.\ (1..N))$$

Here (1..N) denotes the set of integers 1, 2, ..., n, where n is the current value of N. Note the difference between this and

$$((1 \leq I \text{ and } I \leq N) \text{ implies } A(I) = B(I))$$

which would mean "If the current value of I is in the set (1..N), then A(I) = B(I) for this particular (current) value of I." It would not be an assertion about all values of I in the given set. Logically, our given final assertion is equivalent to a use of the quantifier $\forall$ ("for all"):

$$\forall I((1 \leq I \text{ and } I \leq N) \text{ implies } A(I) = B(I))$$

with the bound variable I. We shall, however, continue to use the

simpler notation given above.

What will our initial assertion be? Clearly, N must be greater than zero, since our loop is of the standard FORTRAN type that is always executed at least once. It is also clear, however, that, in order for this program to work properly, N must not be greater than the given dimension of either of the arrays A and B. Suppose that these arrays are declared by the statement

```
DIMENSION A(200), B(100)
```

Then our program will not execute properly if $N > 100$, so that $N \leq 100$ must be part of our initial assertion.

Why is this true? If $N > 100$, then at some time during the running of the program we will execute B(K) = A(K) with $K > 100$. This will not work because of the definition of the array B. In general, B(K) = A(K) cannot be executed properly unless K is in its proper range, that is, in this case, unless $1 \leq K$ and $K \leq 100$. We say that B(K) = A(K) is a <u>restricted command</u> with the <u>restriction</u> ($1 \leq K$ <u>and</u> $K \leq 100$). If this command occurs in a control path, we shall write the restriction, contained in <u>brackets</u>, just before the command, as follows:

```
C  ASSERTION 1
       [1 ≤ K, K ≤ 100]
       B(K) = A(K)
C  ASSERTION 2
```

The verification condition here is "If assertion 1 is true, then we must have $1 \leq K$ and $K \leq 100$, and if in addition B(K) = A(K) is executed, then assertion 2 must be true." Note that this is quite

different from including the condition ($1 \leq K$ and $K \leq 100$) in the control path. If we wrote

```
C  ASSERTION 1
       (1 ≤ K, K ≤ 100)
       B(K) = A(K)
C  ASSERTION 2
```

with the condition (rather than the restriction) $1 \leq K$ and $K \leq 100$, the verification condition would be "If assertion 1 is true, and if $1 \leq K$ and $K \leq 100$, then if $B(K) = A(K)$ is executed we must have assertion 2 true."

The presence of restrictions normally affects our choice of intermediate assertions. In our example program, it is clear that, using the notation we have introduced, we may write

$$(A(I) = B(I),\ I\ .IN.\ (1..K))$$

as an assertion at statement number 1. However, we must include other assertions here in order to assure that the restriction holds in the proper place. The further assertions we need are

$$0 \leq K,\ K < N,\ N \leq 100$$

Note in particular the assertion $K < N$ rather than $K \leq N$. The last time through the loop, K will be equal to N-1 at statement number 1; it will be increased to N; and then, since it is equal to N, the loop will terminate.

Let us rewrite our example program with all the assertions which we have specified:

```
C  N > 0, N ≤ 100
       K = 0
C  (A(I) = B(I), I .IN. (1..K)), 0 ≤ K, K < N, N ≤ 100
1      K = K + 1
       B(K) = A(K)
       IF (K .NE. N) GO TO 1
C  (A(I) = B(I), I .IN. (1..N))
```

The first control path in this program is

```
C  N > 0, N ≤ 100
       K = 0
C  (A(I) = B(I), I .IN. (1..K)), 0 ≤ K, K < N, N ≤ 100
```

Here (A(I) = B(I), I .IN. (1..K)) is true "vacuously," since K = 0 and therefore the set (1..K) is the null set. We have $0 \leq K$ since K = 0; we have K < N since K = 0 and N > 0; and we have $N \leq 100$ at the end of the path, as at the beginning, because the value of N remains unchanged. Therefore the verification condition is valid.

The second control path is

```
C  (A(I) = B(I), I .IN. (1..K)), 0 ≤ K, K < N, N ≤ 100
       K = K + 1
       [1 ≤ K, K ≤ 100]
       B(K) = A(K)
       (K ≠ N)
C  (A(I) = B(I), I .IN. (1..K)), 0 ≤ K, K < N, N ≤ 100
```

Let us apply the forward substitution method to this path. The variables in the path are K, A(K), and B(K); however, by the time

we get to B(K) = A(K), the value of K has increased by 1. This fact, together with the presence of the restriction $1 \leq K$ and $K \leq 100$, necessitates a change in our forward substitution method.

First of all, we shall no longer start with an initialized table of symbolic values of variables. Instead, whenever we substitute the current symbolic values of variables into an expression, we check to see whether there are any variables in that expression which are not yet in our table. If so, these variables are placed in the table, and the value of each of them is initialized to be the same as its name. Thus we would start by looking at K + 1; this contains the variable K, which is put in the table, so that the table looks like this:

Variable	K
Current value	K

Since no substitution takes place, K + 1 becomes the new value of K, so that the table is now:

Variable	K
Current value	K + 1

We now come to the restriction $1 \leq K$ and $K \leq 100$. We start by substituting the current value of K, just as we would with a condition; the result is $1 \leq K+1$ and $K+1 \leq 100$. However, this must now be part of the conclusion, rather than part of the hypothesis as would have been the case with a condition.

Now comes the statement B(K) = A(K). Substituting the current symbolic value of K for K, we obtain B(K+1) = A(K+1). In addition, we have the new variables B(K+1) and A(K+1). These are

appended to the table, so that it now reads

Variable	K	A(K+1)	B(K+1)
Current value	K+1	A(K+1)	B(K+1)

Next we perform the assignment to B(K+1), obtaining

Variable	K	A(K+1)	B(K+1)
Current value	K+1	A(K+1)	A(K+1)

The next element of the path is the condition K ≠ N. This involves a new variable, N, and this is added to the table, just as before. The result is

Variable	K	A(K+1)	B(K+1)	N
Current value	K+1	A(K+1)	A(K+1)	N

and the result of substituting these values into our condition is K+1 ≠ N.

We are now at the end of our path. We have two hypotheses: the original hypothesis, (A(I) = B(I), I = 1, K) and 0 ≤ K and K < N and N ≤ 100, and the condition K+1 ≠ N. We also have two conclusions. The first of these is the restriction, converted to the form 1 ≤ K+1 and K+1 ≤ 100. The second is the result of substituting the symbolic values above into the final assertion, namely

(A(I) = (if I = K+1 then A(K+1) else B(I)), I .IN. (1..K+1)) and 0 ≤ K+1 and K+1 < N and N ≤ 100

Now, if I = K+1, the first condition becomes A(K+1) = A(K+1), which is identically true. Hence we only need to consider I in the set (1..K+1) such that I ≠ K+1 -- that is, we only need to consider I in the set (1..K). Our verification condition there-

fore reads

$$((A(I) = B(I),\ I\ .IN.\ (1..K)),\ 0 \leq K,\ K < N,$$
$$N \leq 100,\ K+1 \neq N)\ \underline{implies}\ ((A(I) = B(I),$$
$$I\ .IN.\ (1..K)),\ 0 \leq K+1,\ K+1 < N,\ N \leq 100)$$

and this is straightforward.

The use of subscripted variables in tables, as we have done here, is subject to one condition. Suppose we have two entries in our table which involve the same array -- say A(I) and A(J). The question then arises: Is I equal to J? If so, we do not want two separate entries in the table. Usually, if the forward substitution method is used, we will be able to tell whether I = J. However, if not, the entire process must be split up into two cases -- I = J and I ≠ J. If there are more than two such entries, the process must be split up even further.

Let us now try back substitution. Our initial conclusion is

$$(A(I) = B(I),\ I\ .IN.\ (1..K)),\ 0 \leq K,\ K < N,\ N \leq 100$$

After considering the last element of the path, namely the condition K ≠ N, this condition becomes our current hypothesis, and our conclusion remains unchanged. Now we come to the statement B(K) = A(K). As in the forward substitution method, we must ask whether K is equal to I. If so, we want to substitute A(K) for B(I) (= B(K)); if not, no substitution should be done. In this, method, however, we do not need to split up the entire process; instead, we replace B(I) by the conditional expression

$$\underline{if}\ I = K\ \underline{then}\ A(K)\ \underline{else}\ B(I)$$

Thus the result of our substitution is

$$(A(I) = \underline{\text{if}}\ I = K\ \underline{\text{then}}\ A(K)\ \underline{\text{else}}\ B(I),\ I\ .IN.\ (1..K)),$$
$$0 \leq K,\ K < N,\ N \leq 100$$

We may now notice that if $I = K$ we get $A(K) = A(K)$, which is obviously always true and may be omitted. Therefore we may assume $I \neq K$, and the range $(1..K)$ may be shortened to $(1..K-1)$. Thus we have

$$(A(I) = B(I),\ I\ .IN.\ (1..K-1)),\ 0 \leq K,\ K < N,\ N \leq 100$$

as our current conclusion.

Next we pass the restriction [$1 \leq K$ and $K \leq 100$]. This restriction is a conclusion of our verification condition, and must be added to our current conclusion. Thus the current conclusion becomes

$$(A(I) = B(I),\ I\ .IN.\ (1..K-1)),\ 0 \leq K,$$
$$K < N,\ N \leq 100,\ 1 \leq K,\ K \leq 100$$

We now see that $0 \leq K$ is superfluous (since if $1 \leq K$ then clearly $0 \leq K$) and may be omitted. Likewise $K \leq 100$, which we have just added, is implied by $K < N$ and $N \leq 100$, so that it is also superfluous. We are left with

$$(A(I) = B(I),\ I\ .IN.\ (1..K-1)),\ K < N,\ N \leq 100,\ 1 \leq K$$

as our current conclusion.

Finally we come to $K = K + 1$. We must replace K in both the current hypothesis and the current conclusion by $K + 1$. The current hypothesis, $K \neq N$, becomes $K+1 \neq N$, and the current con-

clusion becomes

$$(A(I) = B(I),\ I\ .IN.\ (1..K)),\ K+1 < N,\ N \leq 100,\ 1 \leq K+1$$

The verification condition is thus

$$((A(I) = B(I),\ I\ .IN.\ (1..K)),\ 0 \leq K,\ K < N,\ N \leq 100,\ K+1 \neq N)$$
$$\underline{\text{implies}}\ ((A(I) = B(I),\ I\ .IN.\ (1..K)),\ K+1 < N,\ N \leq 100,\ 1 \leq K+1)$$

and this may be checked in a straightforward manner, just as before.

This may seem like a long proof of the correctness of a very simple program. It is very easy, however, to construct a simpler "proof" that appears not to depend upon the initial assertion $N \leq 100$. Such a "proof," of course, cannot be valid.

NOTES

The assertion method of section 6-1 is the most widely used method of proving the correctness of programs. It was first described in [Floyd 67], and independently, though much less rigorously, in [Naur 66]; it is introduced on pages 14-20 of [Knuth 68]; it has been computerized by J. King, a student of Floyd [King 71], and independently by D. Good [Good 70] and many others (for a survey of the state of the art as of August 1972, see [London 72]). The method itself seems to have been thought of independently by a number of people. Floyd credits both Perlis and Gorn with having known it, and Gorn with having written an unpublished paper about it; Knuth traces the germ of the idea back to von Neumann (see [Goldstine and von Neumann 47]); whereas London (see [London 72]) traces it to Turing [Turing 50].

Floyd's original method involved attaching assertions to the arrows in a flowchart. Our method, in which each assertion is attached to the command which it follows, and describes the relation among the variables which holds just <u>before</u> that command is executed, is essentially the one used in computerized versions of the method, and is given here for that reason. Floyd also originally specified an assertion between <u>every</u> pair of adjacent commands in a program (and the example in [Knuth 68], another version of Euclid's algorithm which is more complex and more efficient than the one given here, is of this form); but Floyd also introduces what is essentially the method given here of placing intermediate assertions only where needed, and this method is used by King. It is to be noted that King's "correctness" is what we have called partial correctness, a term which made its first appearance in [Manna 69].

Proofs of correctness of programs written in a given language have been related in various ways to the semantics of that language. In [Floyd 67], it is proposed that the verification conditions of a command be themselves the semantic definition of that command. This idea is expanded in [Hoare 69], [Hoare 71], and [Hoare and Wirth 72]. Relations between our notion of semantics and that of Floyd are given in [Lauer 71] and in [Maurer 72]; for a related treatment, see [Cooper 69].

Our treatment of control path determination and of verification conditions is essentially that of [King 69] (also see [Good 70]). The proper treatment of subscripted variables in assertions is of considerable importance, since it is quite easy to produce "proofs" that are invalid if subscripted variables are treated improperly.

EXERCISES

1. Suppose that we wished to avoid using the FORTRAN arithmetic IF statement in the GCD program of section 6-1. This might be done by replacing the single statement

```
1     IF (I-J) 2, 4, 3
```

by the two statements

```
1     IF (I .EQ. J) GO TO 4
9     IF (I .GT. J) GO TO 3
```

Describe completely the changes that would have to be made in the proof of partial correctness given in section 6-1, if this change is made in the program.

2. Suppose that the two statements

```
10    IF (I .LT. 0) I = -I
11    IF (J .LT. 0) J = -J
```

are inserted in the GCD program of Chapter 6-1, immediately following statement number 6.

(a) What does the modified program do? Give a final assertion for it.

(b) Are there any initial conditions, as there are for the unmodified program? Give an initial assertion, if one is necessary.

(c) What changes would have to be made to the intermediate assertion if this modification is made?

(d) What changes would have to be made in the proof of partial correctness?

3. The following control path appears in a program to calculate exponentials by successive multiplication:

C $N \geq 0$
5 $X = 1$
6 $I = N$
C $N \geq 0$, $I \geq 0$, $X = A^{N-I}$

(a) Make plausible definitions of v_P and v_R (as in section 6-2), where P is the initial assertion of this path and R is its final assertion.

(b) Make plausible definitions of the effects e_1 and e_2 of the two assignments in this path.

(c) In terms of the above definitions, prove rigorously that the verification condition of the above path is valid.

4. Suppose that our GCD program were modified as indicated at the end of section 6-2. Describe completely the changes which would be necessary in the proof of partial correctness of that program.

5. How might the method of section 6-3, for determining control paths, be modified if such paths are permitted to contain computed GO TO statements in FORTRAN (which are of the form

GO TO $(n_1, \ldots, n_k)$ v

and which transfer to statement number n_i, $1 \leq i \leq k$, if the integer variable v is equal to i)? Describe in detail.

6. List the control paths of the GCD program of section 6-1 in the order in which they would be found by the control path

determination procedure of section 6-3, and, with each path, give the contents of the stack at the time that path is found.

7. Construct a table, as in section 6-4, illustrating the forward substitution method as applied to the four control paths of the GCD program of section 6-1.

8. Construct a table, as in section 6-4, illustrating the back substitution method as applied to the four control paths of the GCD program of section 6-1.

9. Prove, using forward substitution as in section 6-5, that the verification condition of the third control path of the example program given in that section is valid.

10. Prove, using back substitution as in section 6-5, that the verification condition of the third control path of the example program given in that section is valid.

CHAPTER SEVEN

FINDING ASSERTIONS

7-1 Determining Assertions Automatically

In Chapter 8, we shall learn how to complete the proof that a program is correct. In the present chapter, we take up the question of how to determine the intermediate assertions which must be supplied in order to prove partial correctness. These intermediate assertions, in general, become more complex as we treat larger and larger programs.

As a first step, we shall ask ourselves whether there are any programs for which the intermediate assertions may be determined automatically. We shall see that any program which contains no loops is of this form. In fact, for a program which contains no loops, all we have to do is to give the initial assertion; the final assertion, along with the proof of correctness, may then be determined by a relatively simple algorithm. (In this case, of course, correctness and partial correctness are the same, since a program having no loops must always terminate.)

Let us start by analyzing a program containing no transfer statements whatsoever. The following program raises the variable A to the 23rd power:

B = A × A

C = A × B

B = B × C

B = B × B

B = B × B

A = B × C

We introduce three new variables, A0, B0, and C0, to stand for the initial values of A, B, and C. Thus our initial assertion is A = A0, B = B0, and C = C0. Using these initial values, we may apply the forward substitution algorithm of section 6-4 to the above program, considered as a path:

Variable	A	B	C
Initial value	$A0$	$B0$	$C0$
Value after first statement	$A0$	$A0^2$	$C0$
Value after second statement	$A0$	$A0^2$	$A0^3$
Value after third statement	$A0$	$A0^5$	$A0^3$
Value after fourth statement	$A0$	$A0^{10}$	$A0^3$
Value after fifth statement	$A0$	$A0^{20}$	$A0^3$
Final value	$A0^{23}$	$A0^{20}$	$A0^3$

The final assertion is thus $A = A0^{23}$, $B = A0^{20}$, and $C = A0^3$.

Since the initial values B0 and C0 are never used, we may remove them from the initial assertion of this program, leaving A = A0. The final assertion, similarly, may be shortened if we agree that the final values of B and C are temporary results of intermediate calculations and are thus of no further interest. Thus the program, with initial and final assertions, becomes

```
C  A = A0
       B = A ✕ A
       C = A ✕ B
       B = B ✕ C
       B = B ✕ B
       B = B ✕ B
       A = B ✕ C
C  A = A0**23
```

Now suppose we change the final statement of this program to

```
       B = B ✕ C
```

The final assertion now becomes A = A0, B = $A0^{23}$, and C = $A0^{3}$. Since A is still equal to its initial value A0, we may substitute A for A0 and obtain B = A^{23} and C = A^{3}. The variable A0 is no longer needed, and we may remove it from the initial assertion, leaving nothing remaining -- or, what is the same thing, leaving an initial assertion which is always true. This means that the proof of correctness of this program may be expressed as:

```
C  .TRUE.
       B = A ✕ A
       C = A ✕ B
       B = B ✕ C
       B = B ✕ B
       B = B ✕ B
       B = B ✕ C
C  B = A**23, C = A**3
```

If we, as before, remove from the final assertion here the equa-

tion involving C (viewing C as a temporary variable) we obtain the intuitively obvious statement of what the program does -- it sets B equal to A^{23} -- as the final assertion.

Let us now consider programs which contain transfer and conditional transfer statements, but which do not contain loops. A path from the beginning of such a program to the end cannot contain repeated statements, because otherwise there would be a loop in the program; hence there can be only a finite number of such paths, which we shall denote by $\pi_1, \ldots, \pi_n$. Along each π_i, we carry out the analysis above, and arrive at a final assertion A_i. There will, in general, also be a condition C_i which must be satisfied for the path π_i to be taken. Then

$$(A_1 \text{ and } C_1) \text{ or } (A_2 \text{ and } C_2) \text{ or } \ldots \text{ or } (A_n \text{ and } C_n)$$

is the final assertion of the program.

We illustrate this by considering a program to find the absolute value of a number. The program is:

```
      Y = X
      IF (Y .GE. 0) GO TO 1
      Y = -X
1     CONTINUE
```

The first control path of this program is

```
      Y = X
      (Y ≥ 0)
```

Introducing new variables X0 and Y0, as before, to stand for the initial values of X and Y, one obtains by forward substitution:

Variable	X	Y
Initial value	X0	Y0
Value after first statement	X0	X0
(Y ≥ 0) becomes (X0 ≥ 0)		
Final value	X0	X0

Thus A_1, for this program, is X = X0 and Y = X0, while C_1 is X0 ≥ 0. The second control path of the program is

Y = X

(Y < 0)

Y = -X

Forward substitution in this case gives us:

Variable	X	Y
Initial value	X0	Y0
Value after first statement	X0	X0
(Y < 0) becomes (X0 < 0)		
Final value	X0	-X0

Thus A_2 is X = X0 and Y = -X0, C_2 is X0 < 0, and the final assertion is

(X = X0 and Y = X0 and X0 ≥ 0) or (X = X0 and Y = -X0 and X0 < 0)

Since X has remained unchanged, we may substitute X for X0 in this assertion, and eliminate X = X0, obtaining

(Y = X and X ≥ 0) or (Y = -X and X < 0)

-- that is, Y = |X|.

What happens in this method for a program containing a

loop? In this case there will be an <u>infinite</u> number of paths from the beginning of the program to the end. We may still carry out the analysis above, obtaining an infinite sequence of assertions A_i and conditions C_i. However, we cannot, in general, reduce these to a single final assertion, although there are special cases in which we can.

Consider the following program to compute A^N:

```
      X = 1
      I = 0
1     IF (I .EQ. N) GO TO 2
      X = X * A
      I = I + 1
      GO TO 1
2     CONTINUE
```

The first four paths from the beginning to the end of this program are:

X = 1	X = 1	X = 1	X = 1
I = 0	I = 0	I = 0	I = 0
(I = N)	(I ≠ N)	(I ≠ N)	(I ≠ N)
	X = X * A	X = X * A	X = X * A
	I = I + 1	I = I + 1	I = I + 1
	(I = N)	(I ≠ N)	(I ≠ N)
		X = X * A	X = X * A
		I = I + 1	I = I + 1
		(I = N)	(I ≠ N)
			X = X * A
			I = I + 1
			(I ≠ N)

Forward substitution in the first path yields:

Variable	X	I	N	A
Initial value	XO	IO	NO	AO
Value after first statement	1	IO	NO	AO
Value after second statement	1	0	NO	AO

(I = N) becomes (0 = NO)

The final assertion is X = 1 together with facts about I, N, and A, which we shall agree to ignore. If we substitute N for NO, much as we did in the previous example, then the condition for this path to be followed becomes N = 0.

Forward substitution in the second path yields the final assertion X = AO, or X = A after a similar substitution is made. The condition here is N = 1. Forward substitution in the third path yields X = A * A, under the condition N = 2; in the fourth path we get X = A * A * A, under the condition N = 3. Thus the final assertion of the entire program is

$$(N = 0 \text{ and } X = 1) \text{ or } (N = 1 \text{ and } X = A) \text{ or } (N = 2 \text{ and } X = A * A) \text{ or } (N = 3 \text{ and } X = A * A * A) \text{ or } \ldots.$$

It requires very little ingenuity to see that the above infinite disjunction may be written more simply as

$$(X = A^N)$$

This program, however, was carefully constructed in such a way that the final step -- sometimes called the inductive inference -- is obvious. In most programs, it will be much less so.

7-2 Assertions in Loops

The simplest loop programs are those having only a single loop. The exponentiation program of the preceding section is a typical example:

```
      X = 1
      I = 0
1     IF (I .EQ. N) GO TO 2
      X = X * A
      I = I + 1
      GO TO 1
2     CONTINUE
```

How does this program work? The variable I counts up from 0 to N, and, at the same time, the partial product, X, assumes values from 1 (that is, A^0) to A^N. At the kth stage, we have $I = k$ and $X = A^k$; and we can write this more simply as $X = A^I$. The assertion $X = A^I$, then, will be true whenever this program passes statement number 1 (and just before that statement is executed). At the end of the program, of course, we will have the assertion $X = A^N$.

What is our initial assertion? This is a bit tricky. It is clear that if we start with $N < 0$, our program will run on indefinitely. Therefore we must start with $N \geq 0$. However, in this program there is something else to watch out for: if we start with N = 0 and A = 0, we will have X = 1 at the end, which makes no sense because 0^0 is undefined. (Is that obvious enough? $0^x = 0$, for $x \neq 0$; but $x^0 = 1$, for $x \neq 0$.) For the moment, we shall assume $A \neq 0$, regardless of the value of N (for the general case, see Exercise 4 at the end of this section). Thus our initial assertion is $N \geq 0$ and $A \neq 0$. Let us rewrite the program with these assertions:

HEARST EUCLID

NORTH CAMPUS COPY CENTER
2483 Hearst Ave.
849-0700

CUSTOMER'S RECEIPT

DATE CLERK AMOUNT RECEIPT NO.

CASH ORDER

DATE CLERK AMOUNT RECEIPT NO.

COPY CENTERS of BERKELEY

NORTH CAMPUS COPY CENTER
2483 Hearst Ave.
849-0700

ITEM (PAGE NUMBERS, TITLES, ETC.	No. ORIGINAL PAGES	COPIES PER PAGE	TOTAL COPIES	COST PER COPY	TOTAL PRICE
B/K a/hr		1	60	5	300
OFFSET MASTERS @ 30¢ EACH					
COLLATE @ ½¢ / SHEET					

Ⓒ COLLATE Ⓓ 2PP/COPY Ⓑ B-B Ⓛ LEGAL Ⓟ 3HP

Sub-total	300
Sales tax	20
TOTAL DUE	320

Circle one: • While I Wait • Today at _____ • Overnight _____

3:30 P.M.

Name J Maurer

Address _____

Date 4-25-75

```
C  N ≥ 0 and A ≠ 0
       X = 1
       I = 0
C  X = A^I
1      IF (I .EQ. N) GO TO 2
       X = X * A
       I = I + 1
       GO TO 1
C  X = A^N
2      CONTINUE
```

There are three control paths in this program, as follows:

```
C  N ≥ 0 and A ≠ 0
       X = 1
       I = 0
C  X = A^I
```

```
C  X = A^I
     (I = N)
C  X = A^N
```

```
C  X = A^I
     (I ≠ N)
       X = X * A
       I = I + 1
C  X = A^I
```

In the first control path, we have $X = A^I$ at the end because $X = 1$, $I = 0$, and $1 = A^0$ for $A \neq 0$. (The assertion $N \geq 0$ is not actually needed here. This may seem peculiar, but remember that, if $N < 0$, the program does not terminate. So far, we are proving only <u>partial</u> correctness -- either the program is correct or it is endless. This condition holds even without the initial assertion $N \geq 0$.) The second control path is obvious -- if $I = N$, then $X = A^I$ becomes $X = A^N$. In the third path, we start with $X = A^I$, then multiply X by A (yielding $A^I * A$, or A^{I+1}), and then increase I by 1, so that again $X = A^I$ in terms of the new value of I. Thus the above program is partially correct.

What if we replace .EQ. (equal to) by .GE. (greater than or equal to) in statement number 1? In this case the program still works, but not with the given assertions. For example, the second control path now becomes

C $X = A^I$

$(I \geq N)$

C $X = A^N$

and now we cannot deduce $X = A^N$ from $X = A^I$ and $I \geq N$.

It is clear, in this program, that our assertion $X = A^I$ still does, in fact, hold whenever the program passes statement number 1. It should also be clear that, if the above control path is taken, X is set equal to A^N, because I, in this program, never becomes larger than N. Let us express this fact by writing $I \leq N$ as another intermediate assertion. The second control path then becomes

C $X = A^I$, $I \leq N$

$(I \geq N)$

C $X = A^N$

Under these conditions, in order for this path to be taken, we must have both $I \leq N$ and $I \geq N$. This can only happen if $I = N$, so that $X = A^I$ implies $X = A^N$, just as before.

However, if we are going to put $I \leq N$ as an assertion at statement number 1, the first and third control paths of the program will also be affected. We may write these as

C $N \geq 0$ and $A \neq 0$

$X = 1$

$I = 0$

C $X = A^I$, $I \leq N$

C $X = A^I$, $I \leq N$

$(I < N)$

$X = X * A$

$I = I + 1$

C $X = A^I$, $I \leq N$

In the first control path, we have $I \leq N$ at the end because $I = 0$ and $N \geq 0$. (This time we do need the condition $N \geq 0$. Note that if $N < 0$, this time, the program terminates and gives the wrong answer, namely $X = 1$.) In the third control path, we have the condition $I < N$, which becomes $I-1 < N$ in terms of the new value of I, after we set $I = I + 1$. Since I and N are assumed to be integers, $I-1 < N$ is equivalent to $I \leq N$. The other assertions at the ends of these two paths follow just as they did before, completing the proof of partial correctness of the modified program.

Assertions like $I \leq N$ are useful in a number of other situations. As we saw in section 6-5, they are needed in programs containing references to subscripted variables, where it is necessary to prove that the subscripts are in range. We shall see in Chapter 8 that they are also useful in proving that a loop terminates. Even in our original, unmodified version of the $X = A^I$ program, we need to include the intermediate assertion $I \leq N$ for the purposes of proving termination.

It should be emphasized that an informal analysis of the control paths of a program, as above, should be carried out before we attempt to derive the verification conditions of these paths by forward or back substitution. This is because, as we saw in the preceding example, it may be necessary for us to include more assertions at certain points in a program than we originally thought were necessary. Once the informal analysis seems to work, however, it is still necessary to perform the forward or back substitution in order to have a rigorous proof on hand. Let us do this for the third path in the modified version of the program; the others follow in a similar manner. Forward substitution yields the following table:

Variable	X	I	N	A
Initial value	XO	IO	NO	AO
(I < N) becomes (IO < NO)				
Value after X = X * A	XO*AO	IO	NO	AO
Final value	XO*AO	IO+1	NO	AO

The final assertion thus becomes $XO*AO = AO^{IO+1}$ and $IO+1 \leq NO$, and this must be implied by $XO = AO^{IO}$, $IO \leq NO$, and $IO < NO$. Back substitution yields the following table:

Statement	Current hypothesis	Current conclusion
3	None	$X = A^{I}$, $I \leq N$
2	None	$X = A^{I+1}$, $I+1 \leq N$
1	None	$X*A = A^{I+1}$, $I+1 \leq N$
Start	$I < N$	$X*A = A^{I+1}$, $I+1 \leq N$

and the verification condition is $(X = A^{I}, I \leq N, I < N)$ <u>implies</u> $(X*A = A^{I+1}, I+1 \leq N)$. In either case, we see immediately that the path is valid, provided that I and N are integers.

Many other loop programs having a single loop may be analyzed as above. In particular, the assertion to be placed at the single intermediate control point will normally involve the partial result of the calculation, together with certain restrictions on the behavior of the loop index. We shall now analyze a number of programs of this kind.

The following program sets S equal to the sum of the elements of the array A:

```
      I = 1
      S = 0
1     S = S + A(I)
      I = I + 1
      IF (I .LE. N) GO TO 1
```

The assertion at statement number 1, besides $1 \leq I \leq N$, says that S is equal to the sum of all the array elements A(k) for $1 \leq k \leq I-1$. (For this and the other examples of this section, we shall not complete the analysis, which follows in a straightforward manner as before; we are interested only in determining the assertion to be placed within the loop.)

The following program searches the array A for an element equal to X, and transfers to statement number 2 if such an element is found:

```
      I = 1
1     IF (X .EQ. A(I)) GO TO 2
      I = I + 1
      IF (I .LE. N) GO TO 1
```

The assertion at statement number 1 is that X is unequal to A(k) for all k, $1 \leq k \leq I-1$. As before, we must also assert here that $1 \leq I \leq N$.

Note that in each of the preceding programs the assertion in the loop is "offset" -- that is, it involves the range $1 \leq k \leq I-1$, rather than $1 \leq k \leq I$. This is determined by the form of the loop; in certain variations of the above programs, the assertion would not be offset. For example, the second program above could be rewritten as

```
      I = 0
1     IF (I .EQ. N) GO TO 3
      I = I + 1
      IF (X .NE. A(I)) GO TO 1
```

Here statement number 2 is assumed to follow the program immediately. The assertion here (at statement number 1) is that $X \neq A(k)$ for $1 \leq k \leq I$, and also that $0 \leq I \leq N$. This version, of course, unlike the preceding version, works properly if $N = 0$ -- that is, no tests are made. However, consider the following version:

```
      I = 1
1     IF (I .GT. N) GO TO 3
      IF (X .EQ. A(I)) GO TO 2
      I = I + 1
      GO TO 1
```

Now the program works properly if $N = 0$, but the assertion is still offset; that is, at statement number 1, we have $X \neq A(k)$ for $1 \leq k \leq N-1$. The condition on I in this version is $1 \leq I \leq N+1$.

7-3 The Language Of Assertions

Any Boolean (or LOGICAL, in FORTRAN) expression may be thought of as an assertion, so long as it has no side effects. In some programs, such assertions are sufficient; our exponential program of the previous section is a good example. In other programs, however, we must formulate assertions (on arrays, for example) which cannot, in general, be reduced to this form. Therefore we must concern ourselves with other ways of expressing assertions.

Consider the first assertion of section 6-5 -- the assertion that A(I) = B(I) for each I, $1 \leq I \leq N$. We have expressed this as

```
(A(I) = B(I), I .IN. (1..N))
```

First, let us remark that this same assertion could also have been expressed by the use of recursion. If we define

```
F(0) = .TRUE.
F(J) = F(J-1) .AND. (A(J) = B(J))
```

then F(N) is equivalent to the above assertion. Recursive definitions such as the above may often be used directly in the proof of correctness of programs; for example, in the proof of the program of section 6-5, we effectively used the fact that

```
(A(I) = B(I), I .IN. (1..K)) and A(K+1) = B(K+1)
    implies (A(I) = B(I), I .IN. (1..K+1))
```

Using the assertion F(J), this becomes

```
F(K) and A(K+1) = B(K+1) implies F(K+1)
```

and this is an immediate consequence of the definition of F(J).

Our preference for assertions stated in terms of sets of integers, rather than by recursion formulas, arises from the fact that there are many programs to which the recursive definitions do not apply directly. For example, consider the following program:

```
      K = N
1     B(K) = A(K)
      K = K - 1
      IF (K .NE. 0) GO TO 1
```

The final assertion of this program is the same as before, and may be represented as F(N), using the terminology above. However, the definition of F(N) can no longer be used in the proof, as an attempt to prove the correctness of this program will show. We could define a new assertion G(N) by the formulas

```
G(0) = .TRUE.
G(J) = G(J-1) .AND. (A(N-J) = B(N-J))
```

and use this definition in the proof of the new program; however, we must now show that G(N) and F(N) are the same. These and similar difficulties with the recursive approach strongly motivate the construction of a language of assertions which deals directly with sets of subscripts, and with finite sets in general. As we shall see, the assertions needed in a wide class of programs may be formulated in such a language.

Our primitive sets, such as (1..N) in the example at the beginning of this section, shall be constructed as lists of ele-

ments separated by commas. The elements are of the following forms:

(1) I (the integer I alone);

(2) I..J (the integers I, I+1, ..., J);

(3) I..J..K (the integers I, I+J, I+2J, ..., K).

Thus, for example,

(1, 6, 10..15, 17, 18..3..27)

stands for the set of integers 1, 6, 10, 11, 12, 13, 14, 15, 17, 18, 21, 24, and 27.

In the form I..J, if J < I, then I..J is the null set of integers, also denoted by () . In the form I..J..K, we assume that K is always of the form I+nJ, for some integer n; if K < I, then I..J..K = () . The letters I, J, and K, in all of the above, may be replaced by any integer expression. Of course, as the current values of our expressions vary, so the current values of our sets will vary as well.

Sets may be given names, using a set assignment statement of the form

S = X

where S is the name of a set and X is some specification of that set. Here X may be one of the primitive set descriptions above; it may also involve the operations .U. (union), .I. (intersection), or .D. (difference). If S1 and S2 are any set specifications, then so are S1 .U. S2, S1 .I. S2, and S1 .D. S2.

As an example, the assignments

S1 = (1..N)

S2 = (N+1..N2)

S3 = S1 .U. S2

S4 = (1..N2)

S5 = S4 .D. S1

S6 = S1 .I. S2

for N < N2, set S3 equal to (1..N2), S5 equal to S2, and S6 equal to the null set.

Set assignments may also be parametrized. If we write

$$S(i_1, \ldots, i_n) = X$$

where X involves the integer variables $i_1, \ldots, i_n$, then $S(e_1, \ldots, e_n)$, if used later, denotes X with the integer expressions $e_1, \ldots, e_n$ substituted for $i_1, \ldots, i_n$, respectively.

The most basic assertion involving sets is of the form

(F(I), I .IN. S)

meaning that F(I) is true for each I in the set S. Here F(I) is any assertion involving I. All of the assertions involving sets which we have used up to now have been of this form. The form may be generalized to

$$(F(I_1, \ldots, I_n), I_1 \text{ .IN. } S_1, \ldots, I_n \text{ .IN. } S_n)$$

meaning that $F(I_1, \ldots, I_n)$ is true for all possible combinations of $I_1, \ldots, I_n$ taken from the respective sets $S_1, \ldots, S_n$. Here $F(I_1, \ldots, I_n)$ is any assertion involving $I_1, \ldots, I_n$.

Assertions, like sets, may be given names. The assignment

A = (F(I), I .IN. S)

gives the name A to the assertion (F(I), I .IN. S), so that this name may be used in combination with other assertions using the

standard Boolean functions .AND., .OR., .NOT., .IMP. (implies), and .EQV. (equivalence). Such assignments, like set assignments, may be parametrized. Using a single parameter, we may write, for example,

A(J) = (F(I, J), I .IN. S)

after which A(K) would stand for the assertion (F(I, K), I .IN. S). Extension to multiple parameters is immediate.

Two other methods of obtaining sets are of interest. The first is by means of a property of elements of an existing set. The specification

(I .IN. S .ST. P(I))

stands for the set of all integers I in the set S such that P(I) is true. (Here .ST. stands for "such that.") The second is as the range of a function; the specification

SET(F(I), I .IN. S)

refers to the set of all F(I), for I in the set S, or the range of the function F whose domain is S. This is to be carefully distinguished from simply writing (F(I), I .IN. S), which is an assertion, not a set (the assertion that F(I) is true for all I in S).

The preceding two definition methods may be combined. We may write

SET(F(I), I .IN. S .ST. P(I))

This is the set of all F(I) for all I in the set S for which P(I) is true. The SET function may also be generalized to several vari-

ables; thus

$$\text{SET}(F(I_1, \ldots, I_n),\ I_1\ .\text{IN.}\ S_1,\ \ldots,\ I_n\ .\text{IN.}\ S_n)$$

signifies the set of all $F(I_1, \ldots, I_n)$ for all combinations of I_j in the respective sets S_j, while

$$\text{SET}(F(I_1, \ldots, I_n),\ I_1\ .\text{IN.}\ S_1,\ \ldots,\ I_n\ .\text{IN.}\ S_n\ .\text{ST.}\ P(I_1, \ldots, I_n))$$

signifies the set of all $F(I_1, \ldots, I_n)$ for only those combinations of I_j in S_j for which $P(I_1, \ldots, I_n)$ is true.

Certain functions have sets as their arguments. In general, if any function F is commutative and associative -- that is, if

$$F(I, J) = F(J, I)$$
$$F(I, F(J, K)) = F(F(I, J), K)$$

for all I, J, and K -- then F may be applied to finite sets. The sum function

SUM(S)

where S is a set, is a good example. The sum of all the elements of a finite set is independent of the order in which these elements are combined, and this fact follows directly from the fact that the plus-function -- that is, F(I, J) = I + J -- is commutative and associative. We shall denote this sum, taken over the set S, by SUM(S); thus, for example, SUM(SET(I*I, I .IN. (1..N))) denotes the sum of the squares of the numbers from 1 to N.

In a similar way, we shall take

PROD(S)

MAX(S)

MIN(S)

as standing respectively for the product, the maximum, and the minimum of the elements in the set S; as before, this may be done because each of the functions

F(I, J) = I * J

F(I, J) = MAX(I, J) (= if I > J then I else J)

F(I, J) = MIN(I, J) (= if I < J then I else J)

is commutative and associative.

We shall also allow forms like

SUM(F(I), I .IN. S)

SUM(F(I, J), I .IN. S1, J .IN. S2)

and so on. Thus the sum of the squares of the integers from 1 to N would normally be written SUM(I*I, I .IN. (1..N)). This, in general, is more than an abbreviation; SUM(F(I), I .IN. S) is not, in general, the same as SUM(SET(F(I), I .IN. S)), because of the possibility that two or more of the F(I) may be the same. For example, suppose that F(I) is always either 0 or 1, for all I in S. Then SET(F(I), I .IN. S), unless F is identically either 0 or 1, is the set (0, 1) -- the set containing only the elements 0 and 1 -- and SUM(SET(F(I), I .IN. S)) = 0 + 1 = 1. On the other hand, SUM(F(I), I .IN. S) is meant to be the sum of all F(I), for all I in S -- that is, in this case, the total number of I in S for which F(I) = 1.

The "such that" construction (.ST.) may be applied to assertions as well as to sets. Thus

(P(I), I .IN. S .ST. Q(I))

means that P(I) is true for all I in the set S such that Q(I) is also true. We include this form as a convenience only, since it is clearly equivalent to

(P(I) .OR. .NOT. Q(I), I .IN. S)

Likewise the extended form,

$$(P(I_1, \ldots, I_n),\ I_1 \text{ .IN. } S_1, \ldots, I_n \text{ .IN. } S_n \text{ .ST. } Q(I_1, \ldots, I_n))$$

is included as a convenience, equivalent to

$$(P(I_1, \ldots, I_n) \text{ .OR. .NOT. } Q(I_1, \ldots, I_n),\ I_1 \text{ .IN. } S_1, \ldots, I_n \text{ .IN. } S_n)$$

Finally, we introduce three assertions which effectively involve the use of the logical quantifier $\exists$ ("there exists"):

EI(I, X)	There exists an integer, which we shall call I, and I is a member of the set X
EF(F, X)	There exists a function, which we shall call F, and the set X is the domain of F
EP(P, X)	There exists a permutation of the set X, which we shall call P

The names I, F, and P may be used in assertions which follow the respective uses of EI, EF, and EP that contain them. For example,

(EI(I4, X), .NOT. P(I4))

means that there exists an integer I4 in the set X for which the property P(I4) is not true. It is equivalent to

.NOT. (P(I), I .IN. X)

-- that is, the assertion that it is false to say that P(I) holds for each element I of X. The GCD function discussed in section 6-1 may be defined using MAX, MIN, and EI, as follows:

GCD(I, J) = MAX(K, K .IN. (1..MIN(I, J)) .ST.
(EI(K1, (1..I)), I = K*K1, EI(K2, (1..J)), J = K*K2))

7-4 Debugging By Proving Correctness

We shall now consider a very simple example of the payoff that is to be expected from proving correctness of programs. Namely, we shall show how our methods may be used to find bugs in programs that not only look correct, but actually are correct for most (but not all possible) input situations.

Let us write a program to test whether the positive integer I is prime. We assume that we have access to the standard remainder function MOD(I, J) whose value is the remainder when I is divided by J. Since I will be non-prime if MOD(I, J) = 0 for suitably chosen values of J, we start our programming task by writing

```
IF (MOD(I, J) .EQ. 0) GO TO 2
```

where at statement number 2 we will note the fact that I is not prime. Let us do this by setting the value of a function, called PRIME(I), to .FALSE.; thus our program reads

```
      LOGICAL FUNCTION PRIME(I)
      . . .
      IF (MOD(I, J) .EQ. 0) GO TO 2
      . . .
2     PRIME = .FALSE.
      RETURN
      . . .
```

Now what do we want to do if MOD(I, J) is unequal to 0? We want to increase J by 1 and loop back. We must make a test at this point, and we may remember that if I is the product of any two integers then at least one of them must be less than or equal to the square root of I.

Since it is easier to take squares than square roots, we write

```
      LOGICAL FUNCTION PRIME(I)
      . . .
1     IF (MOD(I, J) .EQ. 0) GO TO 2
      J = J + 1
      IF (J*J .LE. I) GO TO 1
      . . .
2     PRIME = .FALSE.
      RETURN
      . . .
```

There are now several things remaining to be cleared up. If the test on J fails, then I is in fact prime, and we must set PRIME to .TRUE. . We must remember to initialize J to 2, rather than 1, since any integer is the product of itself and 1. Finally, we must put an END statement on the program. The result is:

```
      LOGICAL FUNCTION PRIME(I)
      J = 2
1     IF (MOD(I, J) .EQ. 0) GO TO 2
      J = J + 1
      IF (J*J .LE. I) GO TO 1
      PRIME = .TRUE.
      RETURN
2     PRIME = .FALSE.
      RETURN
      END
```

We now have a program which seems to work; we may compile it, and it will tell us, for example, that PRIME(17) is .TRUE. while

PRIME(28) is .FALSE. . Let us, however, seek to verify its correctness. The final assertion, which we wish to place at both the RETURN statements, should clearly be that PRIME = P(I), where P(I) is true if I is prime and false otherwise. In the terminology of the preceding section, we may write by definition

$$P(I) = (I \text{ .NE. } K*L, K \text{ .IN. } (2..I-1), L \text{ .IN. } (2..I-1))$$

-- that is, I is prime if I is unequal to K*L for any K and L in the range $2 \leq K \leq I-1$, $2 \leq L \leq I-1$. Let us modify this definition slightly to take into account the trick we are using in this program having to do with the square root of I. First of all,

$$P(I) = (I \text{ .NE. } K*L, K \text{ .IN. } (2..I-1),$$
$$L \text{ .IN. } (2..I-1) \text{ .ST. } K \text{ .LE. } L)$$

That is, we can restrict ourselves to the case $K \leq L$ in choosing K and L from the above ranges (this is clear, since K*L = L*K). We can then make another transformation of P(I), as follows:

$$P(I) = (I \text{ .NE. } K*L, K \text{ .IN. } (2..I-1), L \text{ .IN. } (2..I-1)$$
$$\text{.ST. } (K \text{ .LE. } L, K*K \text{ .LE. } I))$$

As we have already mentioned, the restriction to those K for which $K*K \leq I$ is justified by the fact that the smaller of the two factors of the positive integer I is less than or equal to the square root of I. Formally, if $K*K > I$, then $K*L \geq K*K > I$, so that $I \neq K*L$.

The initial assertion of our program is I .GT. 0 (since the program does not check for zero or negative I). What is the intermediate assertion at statement number 1? At this point we have checked all possible factors of I which are less than J;

that is,

```
(I .NE. K*L, K .IN. (2..J-1), L .IN. (2..I-1))
```

Let us check to see whether these assertions are enough. The control path which leads around the loop may be written as

```
C  (I .NE. K*L, K .IN. (2..J-1), L .IN. (2..I-1))
      (MOD(I, J) ≠ 0)
      J = J + 1
      (J*J ≤ I)
C  (I .NE. K*L, K .IN. (2..J-1), L .IN. (2..I-1))
```

The only difference between the assertion at the beginning of this path and at the end is that we have increased J by one, and thus the set (2..J-1) has been expanded by one element, namely the initial value of J. Thus we must verify that

```
(I .NE. J*L, L .IN. (2..I-1))
```

if this path is taken, and this follows directly from MOD(I, J) ≠ 0. Furthermore, if J = 2, then (2..J-1) is the null set, and therefore the assertion (I .NE. K*L, K .IN. (2..J-1), L .IN. (2..I-1)) is always true in this case. This shows that the initial control path is valid.

There are two further control paths, corresponding to the two RETURN statements. The first is

```
C  (I .NE. K*L, K .IN. (2..J-1), L .IN. (2..I-1))
      (MOD(I, J) ≠ 0)
      J = J + 1
      (J*J > I)
      PRIME = .TRUE.
C  PRIME = P(I)
```

We have seen, from the analysis of the preceding control path, that (I .NE. K*L, K .IN. (2..J-1), L .IN. (2..I-1)) holds at the end of this path; also, we have J*J > I. We must show that this implies the truth of P(I). As shown above, P(I) is equivalent to

(I .NE. K*L, K .IN. (2..I-1), L .IN. (2..I-1)
.ST. (K .LE. L, K*K .LE. I))

We can assume J < I (otherwise we are done), in which case

(2..I-1) = (2..J-1) .U. (J..I-1)

But for K in (J..I-1) we have K*K > I (since K ≥ J and thus K*K ≥ J*J > I; clearly J can be assumed positive) and so this range may be disregarded, since we only need to consider those K for which K*K ≤ I.

There remains the control path for the other RETURN statement, which reads

```
C  (I .NE. K*L, K .IN. (2..J-1), L .IN. (2..I-1))
      (MOD(I, J) = 0)
      PRIME = .FALSE.
C  PRIME = P(I)
```

Since MOD(I, J) = 0, we have I = J*K for some K. From this, we must show the falsity of P(I). It suffices to show that J ≠ 1 and J ≠ I. This, however, cannot be done with the given initial assertion for this path, which tells us nothing about the behavior of J.

Let us look again at the program. The initial value of J is 2, and it increases thereafter; so it is reasonable to assume

that J can never be 1. Furthermore, the test condition (J*J ≤ I) at the end of the loop assures us that, at this position at least, J can be no more than half of I (since J is itself at least 2). Let us proceed, then, by placing the assertions J ≥ 2 and J*J ≤ I (which will imply J < I) at statement number 1, and seeing if this helps us. We must, of course, now go back and re-verify all the control paths.

For the control path containing PRIME = .FALSE., the two new conditions are, as we have seen, exactly what we need. For the control path containing PRIME = .TRUE., there is no problem, since we have merely added some new hypotheses without any new conclusions. For the control path around the loop, we must verify that J ≥ 2 and J*J ≤ I at the end of the path. The second of these conditions follows directly from the condition J*J ≤ I in the path; also, J ≥ 2 at the beginning of the loop, and J is increased in the loop, so clearly J ≥ 2 at the end.

Unfortunately, with the new conditions, the initial control path is <u>not</u> valid. Since J = 2, we certainly have J ≥ 2; but J*J ≤ I will not hold unless I is at least 4. What do we do now? We seem to have verified the partial correctness of the program with the initial assertion I ≥ 4; the first three special cases, I = 1, 2, and 3, can be "run through" to complete the proof. For I = 1, for example, we have MOD(1, 2) = 1, so we increase J by 1, and then J*J is certainly greater than I. For I = 2 -- Good heavens! We have uncovered a bug!

In fact, for I = 2 (and for <u>no</u> other positive integer value of I), the program given above is <u>not</u> partially correct. It will return PRIME(2) = .FALSE. -- that is, 2 is not a prime --

when in fact it is. This is the kind of bug that gives programming managers nightmares: a program is written, checked out, the programmer goes on to something else, maybe moves to another city, the program is included as a subroutine in a larger program, which in turn is made into a subroutine of one still larger, and then strange erroneous behavior begins to appear. Often it takes weeks to trace the bug back to the original program in which it appears; sometimes it is not found at all, and the larger programs are completely rewritten. In this case it is obvious that, if we were testing numbers one at a time, it would never occur to us to test such an obvious prime as 2; but if this routine is part of a larger one, the logic of the larger program might very easily require it to test the primeness of 2.

Fixing the bug, of course, is very easy; we rewrite the program as follows:

```
      LOGICAL FUNCTION PRIME(I)
C  I > 0
      IF (4 .GT. I) GO TO 3
      J = 2
C  (I .NE. K*L, K .IN. (2..J-1), L .IN. (2..I-1)),
C  J .GE. 2, J*J .LE. I
1     IF (MOD(I, J) .EQ. 0) GO TO 2
      J = J + 1
      IF (J*J .LE. I) GO TO 1
3     PRIME = .TRUE.
C  PRIME = P(I)
      RETURN
```

```
2       PRIME = .FALSE.
C   PRIME = P(I)
        RETURN
        END
```

Here P(I) has the same definition as before. The initial control path (which, in the end, was the only one that gave us trouble) has now been replaced by two paths. The first is

```
C   I > 0
        (4 > I)
        PRIME = .TRUE.
C   PRIME = P(I)
```

This path is valid, because it cannot be taken unless I = 1, 2, or 3, in which case I is prime and P(I) holds. (Note that 1 is prime according to our definition; some authors do not include 1 among the primes.) The other control path is

```
C   I > 0
        (4 ≤ I)
        J = 2
C   (I .NE. K*L, K .IN. (2..J-1), L .IN. (2..I-1)),
C   J .GE. 2, J*J .LE. I
```

This is the same as our original first control path, with the added condition (4 ≤ I). We have already verified all the final assertions here except for J*J .LE. I, and, as we have noted, this follows from the added condition. This completes the proof of partial correctness of the revised program.

7-5 A Larger Example

The following program contains 13 statements (including the final CONTINUE) and three loops, two of which are nested. It is included as an example of finding assertions and proving partial correctness of a program which is larger than the one-loop programs considered up to now. We may notice that, as our programs get longer, the treatment of each separate assertion gets longer as well. This is to be expected, since it reflects the increasing complexity of larger programs; a 100-statement program usually takes much more than ten times as long as a 10-statement program to write and debug.

Our program finds all prime numbers from 1 to N, using the sieve of Eratosthenes. The basic idea is that in order to find all the prime numbers we first find all the numbers that are not prime; then the ones left over will be prime. A number is not prime if it is a multiple of something, and these multiples occur in regular progressions: 4, 6, 8, 10, 12, etc.; 9, 12, 15, 18, etc.; 25, 30, 35, 40, etc., and so on. Once we have found all these in a given range, if we have some way of remembering which ones we did not find, these will be the primes.

The program uses a logical (i. e., Boolean) array of length 1000, so that it will work for $N \leq 1000$. At the end of the program, the elements of this array, called A, will be set to .TRUE. or .FALSE. depending upon whether their indices are prime or not. That is, the final assertion will be

$$(A(I) = P(I),\ I\ .IN.\ (1..N))$$

where P(I) is defined as in the preceding section. The initial

assertion is $N \leq 1000$ (so that the subscripts will always stay within range) and $N > 0$; the program with its initial and final assertions is:

```
C  N > 0, N ≤ 1000
       I = 1
1      A(I) = .TRUE.
       I = I + 1
       IF (I .LE. N) GO TO 1
       I = 2
2      J = I*I
       IF (J .GT. N) GO TO 4
3      A(J) = .FALSE.
       J = J + I
       IF (J .LE. N) GO TO 3
       I = I + 1
       GO TO 2
C  (A(X) = P(X), X .IN. (1..N))
4      CONTINUE
```

We have used the variable X, rather than I, in the final assertion in order to avoid confusion with the variable I in the program.

How many intermediate assertions do we need, and where should we put them? In section 6-3, we saw that our procedure for finding all control paths in a program can find an infinite number of such paths if there is a closed loop in the program which does not contain any control point. We therefore choose our control points in such a way that this does not happen; that is, every closed loop in the program must contain at least one control point. This important condition is known as the closed

loop condition; it is proved in section 8-3 that this condition is necessary and sufficient for the control paths of any "reasonable" program to be finite in number.

How can we make sure that the closed loop condition is satisfied? There are various ways; we shall choose one simple way, namely, to make each statement to which a backward transfer is made into a control point. Clearly every closed loop in any program must contain at least one backward transfer. In our sieve program, there are backward transfers to statements 1, 2, and 3, and these become our intermediate control points.

At the beginning of our program, we initialize all the variables A(1) through A(N) to .TRUE. . The loop which performs this initialization is much like the loops in section 7-2, and the intermediate assertion at statement number 1 is chosen in the same way as was done there. It is the assertion that A(X) has been set to .TRUE. for all indices X which are less than the current value of I, or, in our notation of section 7-3,

$$(A(X) = .TRUE., X .IN. (1..I-1))$$

To determine the other two intermediate assertions, we have to look a bit more closely at how the program works. As soon as we find a non-prime J, we set A(J) to .FALSE. (at statement number 3). The first few non-primes we find are 4, 6, 8, 10, etc.; each of these is found by adding 2 to the previous one (at the statement J = J + I; note that I = 2 the first time through the loop). We stop this process when we have found all multiples of 2 that are less than or equal to N. Now we have to find all such multiples of 3. We can start with 9 = 3*3, since 3*2 is a multiple of 2 and has already been found. Similarly, when finding the mul-

tiples of 7 (say), we do not have to worry about 7*2 through 7*6, because these are all multiples of numbers less than 7. This accounts for the initializing statement J = I*I. If there are no multiples of I less than N, other than those which we have found already, we are done.

At statement number 2, then, we have found all multiples of all numbers less than I that we want to find. Let us form these numbers into a set S1(I); then A(X) = .FALSE. if X is in S1(I), and A(X) = .TRUE. otherwise. This can be written as

$$(A(X) = .FALSE., X .IN. S1(I)), (A(X) = .TRUE., X .IN. ((1..N) .D. S1(I)))$$

We recall that .D. means "difference"; (1..N) .D. S1(I) is the set of all integers in (1..N) that are <u>not</u> in S1(I). The set S1(I) may then be defined as follows:

$$S1(I) = SET(Y*Z, Y .IN. (2..I-1), Z .IN. (2..N-1) .ST. (Y .IE. Z, Y*Z .LE. N))$$

-- that is, the set of all Y*Z for $2 \leq Y \leq I-1$ and $2 \leq Z \leq N-1$ which are such that $Y \leq Z$ and $Y*Z \leq N$. We note that the condition Y .IE. Z could have been omitted, and the same set would have been obtained.

At statement number 3, we have found all elements of the set S1(I), and a few more besides -- namely, the multiples of I that are less than J. Let us make all these into a new set S2(I, J). The assertion here is the same as before, but with S2(I, J) substituted for S1(I) -- that is, it is

$$(A(X) = .FALSE., X .IN. S2(I, J)), (A(X) = .TRUE., X .IN. ((1..N) .D. S2(I, J)))$$

where S2(I, J) is defined by

$$S2(I, J) = S1(I) \text{ .U. SET}(I*Z, Z \text{ .IN. } (2..N-1) \text{ .ST. } (I \text{ .IE. } Z, I*Z \text{ .LT. } J))$$

-- that is, S1(I) together with all those multiples of I that are less than J (recall .U. stands for "union"; also, as before, I .IE. Z could have been omitted).

The behavior of the integers I, J, and N must also be part of the assertions. At statement number 1, we have $1 \leq I$ and $I \leq N$. At statement number 2, we have $I \geq 2$; we are just about to test I*I for being not greater than N, and we therefore know this to be true of the <u>previous</u> value of I -- that is, $(I-1)*(I-1) \leq N$. At statement number 3, we have tested the current value of I, and we know also that J is between I*I and N; that is, $I \geq 2$, $I*I \leq J$, and $J \leq N$. We also know that J is a multiple of I; this may be expressed by asserting that MOD(J, I) = 0, where the MOD function is as in the preceding section. Finally, $N > 0$ and $N \leq 1000$ hold throughout the program if they hold at the beginning; these facts are needed in connection with the restricted commands (see section 6-5) which make reference to the subscripted variable A. Of these, only $N \leq 1000$ needs to be stated explicitly, since $N > 0$ is implied by each of the other intermediate assertions. The program with all assertions is thus:

```
C  N > 0, N ≤ 1000
       I = 1
C  1 ≤ I, I ≤ N, N ≤ 1000, (A(X) = .TRUE., X .IN. (I..I-1))
1      A(I) = .TRUE.
       I = I + 1
       IF (I .IE. N) GO TO 1
```

```
      I = 2
C  I ≥ 2, (I-1)*(I-1) ≤ N, N ≤ 1000, (A(X) = .FALSE., X
C  .IN. S1(I)), (A(X) = .TRUE., X .IN. ((1..N) .D. S1(I)))
2     J = I*I
      IF (J .GT. N) GO TO 4
C  I ≥ 2, I*I ≤ J, J ≤ N, MOD(I, J) = 0, N ≤ 1000,
C  (A(X) = .FALSE., X .IN. S2(I, J)),
C  (A(X) = .TRUE., X .IN. ((1..N) .D. S2(I, J)))
3     A(J) = .FALSE.
      J = J + I
      IF (J .LE. N) GO TO 3
      I = I + 1
      GO TO 2
C  (A(X) = P(X), X .IN. (1..N))
4     CONTINUE
```

where S1(I), S2(I, J), and P(X) are defined as before.

There are seven control paths in this program, as follows:

Path 1

```
C  N > 0, N ≤ 1000
      I = 1
C  1 ≤ I, I ≤ N, N ≤ 1000, (A(X) = .TRUE., X .IN. (1..I-1))
```

Path 2

```
C  1 ≤ I, I ≤ N, N ≤ 1000, (A(X) = .TRUE., X .IN. (1..I-1))
      [1 ≤ I, I ≤ 1000]
      A(I) = .TRUE.
      I = I + 1
      (I ≤ N)
C  1 ≤ I, I ≤ N, N ≤ 1000, (A(X) = .TRUE., X .IN. (1..I-1))
```

Path 3 C $1 \leq I$, $I \leq N$, $N \leq 1000$, (A(X) = .TRUE., X .IN. (1..I-1))

[1 ≤ I, I ≤ 1000]

A(I) = .TRUE.

I = I + 1

(I > N)

I = 2

C $I \geq 2$, $(I-1)*(I-1) \leq N$, $N \leq 1000$, (A(X) = .FALSE., X

C .IN. S1(I)), (A(X) = .TRUE., X .IN. ((1..N) .D. S1(I)))

Path 4 C $I \geq 2$, $(I-1)*(I-1) \leq N$, $N \leq 1000$, (A(X) = .FALSE., X

C .IN. S1(I)), (A(X) = .TRUE., X .IN. ((1..N) .D. S1(I)))

J = I*I

(J > N)

C (A(X) = P(X), X .IN. (1..N))

Path 5 C $I \geq 2$, $(I-1)*(I-1) \leq N$, $N \leq 1000$, (A(X) = .FALSE., X

C .IN. S1(I)), (A(X) = .TRUE., X .IN. ((1..N) .D. S1(I)))

J = I*I

(J ≤ N)

C $I \geq 2$, $I*I \leq J$, $J \leq N$, MOD(I, J) = 0, $N \leq 1000$,

C (A(X) = .FALSE., X .IN. S2(I, J)),

C (A(X) = .TRUE., X .IN. ((1..N) .D. S2(I, J)))

Path 6 C $I \geq 2$, $I*I \leq J$, $J \leq N$, MOD(I, J) = 0, $N \leq 1000$,

C (A(X) = .FALSE., X .IN. S2(I, J)),

C (A(X) = .TRUE., X .IN. ((1..N) .D. S2(I, J)))

[$1 \leq J$, $J \leq 1000$]

A(J) = .FALSE.

J = J + I

($J \leq N$)

C $I \geq 2$, $I*I \leq J$, $J \leq N$, MOD(I, J) = 0, $N \leq 1000$,

6 (A(X) = .FALSE., X .IN. S2(I, J)),

C (A(X) = .TRUE., X .IN. ((1..N) .D. S2(I, J)))

Path 7 C $I \geq 2$, $I*I \leq J$, $J \leq N$, MOD(I, J) = 0, $N \leq 1000$,

C (A(X) = .FALSE., X .IN. S2(I, J)),

C (A(X) = .TRUE., X .IN. ((1..N) .D. S2(I, J)))

[$1 \leq J$, $J \leq 1000$]

A(J) = .FALSE.

J = J + I

($J > N$)

I = I + 1

C $I \geq 2$, $(I-1)*(I-1) \leq N$, $N \leq 1000$, (A(X) = .FALSE., X

C .IN. S1(I)), (A(X) = .TRUE., X .IN. ((1..N) .D. S1(I)))

The verification conditions of these paths may be obtained by the forward or back substitution method as extended in section 6-5. The details of this are left as exercises.

NOTES

Experimentation with the assertion method has been vigorously pursued since its popularization in [Knuth 68]. An account of much of this experimentation, together with a number of comments about the language of assertions, about forward and back substitution, and about computer-aided methods of verification is given in [London 71]. A further summary, including a list of programs that have been proved correct and a list of computer-aided methods in progress of being constructed, is given in [London 72].

There is more to the "23rd-power algorithm" of section 7-1 than there appears to be. This algorithm has been cited by Knuth as an example of the fact that the standard algorithm for raising A to a fixed power (squaring and multiplying by A in some order dependent on the power) does not always take the smallest number of steps.

EXERCISES

1. (a) Determine a final assertion for the following program:

```
      IF (I .EQ. 0) GO TO 2
      IF (I .GT. 0) GO TO 1
      J = -1
      GO TO 3
1     J = 1
      GO TO 3
2     J = 0
3     CONTINUE
```

Express the final assertion using *if*, *then*, and *else*. (This program calculates one of the standard functions of ALGOL.)

(b) Verify each of the paths from the beginning of this program to the end by back substitution.

2. (a) Determine a final assertion for the following program, in terms of the initial value I0 of I:

```
      I = I + J
      IF (J .LT. 0) GO TO 2
      IF (I .LE. N) GO TO 1
      GO TO 3
2     IF (I .GE. N) GO TO 1
3     CONTINUE
```

(This program represents one method of ending a loop in which it is not known whether the step size J is positive or negative. It is assumed that statement number 1 is the beginning of the loop.)

(b) Verify each of the paths from the beginning of this program to the end by back substitution.

3. In each of the following programs, give assertions at the end and at statement number 1.

```
(a)       I = 1
    1     A(I) = 0
          I = I + 1
          IF (I .LE. N) GO TO 1
```

```
(b)       I = 0
          J = 0
    1     I = I + 1
          J = J + I
          IF (I .LE. N) GO TO 1
```

```
(c)       I = 1
          K = 0
    1     IF (A(I) .NE. B(I)) GO TO 2
          I = I + 1
          IF (I .LE. 100) GO TO 1
          GO TO 3
    2     K = 1
    3     CONTINUE
```

4. Suppose that we wish to prove the correctness of the program at the beginning of section 7-2, under the more general assumption that initially $N \geq 0$ and ($N \neq 0$ or $A \neq 0$).

(a) With the intermediate assertion $X = A^I$, show that the verification condition of the first control path is not necessarily valid.

(b) Suppose we define EXP(U, V) to be U^V, except that EXP(0, 0), instead of being undefined, is equal to 1. Place the assertion X = EXP(A, I), rather than X = A^I, at statement number 1. Notice that, in the proof in section 7-2, we needed the general fact $U^V \cdot U^W = U^{V+W}$. Show that the corresponding equation for EXP, namely EXP(U, V)•EXP(U, W) = EXP(U, V+W), does <u>not</u> hold.

(c) With the assertion X = EXP(A, I) at statement number 1, show that the program is nevertheless partially correct. (Hint: Show that EXP(U, V)•U = EXP(U, V+1) always holds if U and V are integers and $V \geq 0$.)

(d) Place the assertion X = EXP(A, N) at the end, and the assertion $N \geq 0$ (omitting all reference to the value of A) at the beginning. Is the program now partially correct? Prove or disprove.

5. After the following set assignments are made, specify which pairs of sets (chosen from S1, S2, and so on up to S10) are the same. Assume that N1 < N2 < N3.

S1 = (N1, N2, N3)

S2 = (N1..N2, N3)

S3 = (N1, N2..N3)

S4 = S1 .U. S2

S5 = S1 .I. S3

S6 = S2 .U. S3

S7 = (N1-1, N3+1)

S8 = (N1-1..N3+1)

S9 = S8 .D. S7

S10 = ()

6. Simplify each of the following set expressions:

(a) SET(I+1, I .IN. (1..10))

(b) SET(6*I, I .IN. (1..10))

(c) SET(K*K, K .IN. (1, 10, 100))

(d) (I1..K..I2) .U. (I2..K..I3) (assume I1 < I2 < I3)

7. A number which is not prime must be divisible, either by 2, or by some odd prime. This suggests a method of speeding up the program of section 7-4: after we test whether I is divisible by 2, we initialize J to 3 and then increase it by 2 each time, rather than by 1. Write such a program and verify informally that it works.

8. Prove the partial correctness of the program of problem 7.

9. Complete the proof of partial correctness of the program of section 7-5.

10. In discovering all non-primes in a given range (in the program of section 7-5), it is sufficient to consider all multiples of primes. Because of the order in which multiples are found, it turns out that any non-prime value of I (the quantity whose multiples we are discovering) will already have been found as a multiple of something else by the time we get around to consider multiples of it. Hence the two statements

```
      I = I + 1
      GO TO 2
```

in the algorithm may be replaced by

```
5     I = I + 1
      IF (A(I)) GO TO 2
      GO TO 5
```

(recall that, in FORTRAN, "IF (A(I))" -- where A(I) is a LOGICAL quantity -- means "If A(I) = .TRUE."). Prove the partial correctness of the modified program. (It is sufficient to consider only those control paths and assertions which are different from those of the unmodified program.)

CHAPTER EIGHT

TERMINATION

8-1 Loop Expressions

We shall now finish what we started in Chapter 6, and show how to prove that a program terminates. If we have already proved the partial correctness of the program, it will follow that the program is correct. We shall always assume that partial correctness has already been proved; in fact, for most programs, some of the facts we needed in order to prove partial correctness will also be used to prove termination.

Every time a program is run, it either eventually stops, or it does not. In the first case, we say that the program *terminates* (or "terminates in a finite number of steps"). If the program does not terminate, it *runs on indefinitely* (or "goes into an endless loop"). If a program has no loops, it always terminates; and so our termination proof methods will concern themselves with the behavior of loops. Specifically, we will be concerned with the *loop indices*, and with other expressions that act like loop indices.

As an example of how to prove that a program terminates, let us consider the following program to calculate A^N, modified so as to have a decreasing rather than an increasing index I:

```
      X = 1
      I = N
1     IF (I .EQ. 0) GO TO 2
      X = X * A
      I = I - 1
      GO TO 1
2     CONTINUE
```

Let us first prove that this program is partially correct. The index I counts from N down to zero, while the partial product is built up from from A^0 to A^N. Hence the intermediate assertion at statement number 1 is not $X = A^I$, but $X = A^{N-I}$. The final assertion is $X = A^N$, as before; and the initial assertion is $N \geq 0$, since it is clear that the program is not correct (and does not even terminate) if $N < 0$. In addition, as before, we shall require $A \neq 0$.

The three control paths are:

```
C  N ≥ 0, A ≠ 0
      X = 1
      I = N
C  X = A**(N-I)
```

```
C  X = A**(N-I)
      (I = 0)
C  X = A**N
```

```
C  X = A**(N-I)
      (I ≠ 0)
      X = X * A
      I = I - 1
C  X = A**(N-I)
```

At the end of the first path, we have $X = A^{N-I}$ because $I = N$ and so $X = A^0 = 1$ (assuming $A \neq 0$). In the second path, if $I = 0$, then clearly $X = A^{N-I}$ and $X = A^N$ are equivalent. Back substitution in the third path gives $X*A = A^{N-(I-1)}$, which is equivalent to the condition $X = A^{N-I}$ at the beginning of the path. So the program is partially correct.

We now observe that, in order to prove that this pro-

gram is correct, we need to prove only that it terminates when started with its initial assertion valid. It is not necessary to prove that it always terminates. In fact, this program does not always terminate; it runs on indefinitely if it is started with $N < 0$. This, in fact, is characteristic of all our termination proofs; they are, in a sense, proofs of "partial" termination.

Intuitively speaking, why does this program terminate? Consider the behavior of the loop index I. Every time we go around the loop, it decreases. How do we know that it does not keep on decreasing indefinitely? Because of the test at statement number 1. But the existence of such a test does not, in itself, guarantee that a loop like this will terminate. We could, for example, have written $I = I - 2$ instead of $I = I - 1$; in that case, if N is odd (3, for example), the program runs on indefinitely.

Suppose, however, that we can prove that $I \geq 0$. In this case it should be clear that the loop must terminate -- an integer variable which is always positive cannot continue to decrease indefinitely. How can we prove that $I \geq 0$? In exactly the same way that we proved partial correctness. We place the assertion $I \geq 0$ at statement number 1, along with all our other assertions at that point; then we recalculate the verification conditions. The three control paths are now:

C $N \geq 0, A \neq 0$
$X = 1$
$I = N$
C $X = A^{N-I}, I \geq 0$

C $X = A^{N-I}, I \geq 0$
$(I = 0)$
C $X = A^{N}$

C $X = A^{N-I}, I \geq 0$
$(I \neq 0)$
$X = X * A$
$I = I - 1$
C $X = A^{N-I}, I \geq 0$

At the end of the first path, we have $I \geq 0$ because $I = N$ and $N \geq 0$. (This program is <u>partially</u> correct for $N < 0$, since it does not terminate, and thus only in the proof of termination do we use the initial assertion $N \geq 0$.) At the <u>beginning</u> of the third path, we have $I \geq 0$ but $I \neq 0$, and therefore $I > 0$ (that is, $I \geq 1$). Thus at the end of this path, after I has been decreased by 1, we have $I \geq 0$. The rest of the assertions at the ends of paths have already been proved; the second path is valid because no new conclusions have been added. Thus the program is still partially correct, and, in addition, we have shown that $I \geq 0$ at statement number 1.

We want to say that the program terminates because I is an integer variable which decreases every time we go around the loop, and is bounded from below at statement number 1. For more general loops, however, we will need to find an <u>expression</u>, rather than a single variable, which has these properties. Such an expression will be called a <u>loop expression</u>.

For a simple example of a loop expression which is not a simple variable, consider the exponentiation program in unmodified form (as given at the start of section 7-2):

```
      X = 1
      I = 0
1     IF (I .EQ. N) GO TO 2
      X = X * A
      I = I + 1
      GO TO 1
2     CONTINUE
```

Here the variable I <u>increases</u> every time we go around the loop,

and is bounded from above. We could, of course, construct dual theories of increasing and decreasing loop expressions, but there is a much simpler method available. If I increases and is bounded from above, then -I decreases and is bounded from below.

Actually, -I is not quite the expression we want here. The reason is that -I is bounded from below by -N (we have $-I \geq -N$, since $I \leq N$) and -N is not a constant. This is an important point, because a loop might not terminate if its expression is bounded by a variable. Consider the following loop:

```
      J = 1
      K = 10
1     J = J + 1
      K = K + 1
      GO TO 1
```

Here K decreases every time we go around the loop, and K > J at all times, but the loop does not terminate. Again, we could make a special case of a variable, such as N in our exponential program, which is not changed by any instruction within the loop. There is, however, no need to make such a special case. If we have a bound which is a variable, we simply add it to our loop expression. Thus in this case the loop expression is N-I; it is bounded from below by zero, and decreases every time we go around the loop.

Another example of a loop expression is furnished by our GCD program from section 6-1. Here there are two integer variables, I and J, which are continually getting smaller. At the same time,

they remain positive throughout the running of the program. This time, we do not have to introduce anything extra into the proof of partial correctness; the assertions $I > 0$ and $J > 0$ at statement number 1 in this program had to be introduced anyway, in order to derive the necessary facts about the behavior of GCDs.

However, the variable I does not always decrease as we pass from statement number 1 around a loop and back to statement number 1 again. In particular, if we perform statement number 2, then I does not decrease. Likewise, J does not always decrease as we go around the loop. The quantity that always decreases is the sum of I and J; and this is our loop expression.

In formal terms:

A loop expression for a given loop containing a given control point is

(1) an integer-valued expression

(2) that decreases every time we go around the loop

(from the given control point, assuming that the assertion given there is valid, around the loop and back to itself)

(3) and that is bounded from below, by a constant, at the given control point (as implied by the assertion given there).

Any program having only one loop always terminates, when started with its initial assertion valid, if that loop has a loop expression. Any program having several loops always terminates, when started with its initial assertion valid, if there is an expression which is a loop expression for all of them. The proof of these facts is quite easy. Suppose, on the contrary, that such a program did not terminate; then there must be one loop which is traversed an infinite number of times in

some particular case. Let v_1, v_2, v_3, ..., be the successive values of the given expression in this case (evaluated each time the program passes the given control point). By definition of a loop expression, we have $v_1 > v_2 > v_3 > \ldots$, which is impossible, because, again by definition of a loop expression, all the v_i are integers and each v_i (since it is evaluated at the control point) is not less than some constant lower bound.

As an application of the above facts, let us prove the correctness of some of the other examples given in the two preceding chapters:

(1) The array-moving program of section 6-5 has loop expression N-K. This program resembles the exponentiation program of section 7-2 in this regard. Here K is the loop index, but it increases, rather than decreasing, and so we must consider -K; and to this we must add N, the bound, since this is variable.

(2) Each of the four programs at the end of section 7-2 has loop expression N-I. The loop expression is not affected by the offset, as this term is used in that section. In fact, a loop expression never needs a constant term; it is easy to see that if *e* is any loop expression and *k* any positive or negative integer, then *e*+*k* is also a loop expression.

(3) The function PRIME(I), given in its correct form near the end of section 7-4, has loop expression I-J*J. This one is a bit harder to determine. It is clear that J increases and therefore -J decreases as we go around the loop. It is also true that J never becomes larger than I (and, indeed, we could also have used I-J as our loop expression). However, the fact that J*J never becomes larger than I is already known, by the assertion

$J*J \leq I$ at statement number 1. Thus, by using I-J*J, we obtain a simpler proof of termination.

The sieve program of section 7-5 has several loops, and the proof of its termination will be deferred to section 8-5.

8-2 Finding Loop Expressions

Loops terminate, in practice, for quite a variety of reasons. Each of these corresponds to a type of loop expression. In this section, we shall take up a number of these, and show how the loop expressions are found in each case.

We may remark, first of all, that all three of our conditions on a loop expression (as given at the end of the preceding section) are necessary. Consider condition (1). If we have a real expression that satisfies conditions (2) and (3), our loop might not terminate, as in the following program:

```
      A = 1.0
5     A = A/2.0
      GO TO 5
```

Here A decreases every time we go around the loop, and remains strictly positive at all times. (Of course, in an actual computer, the finite representation of A would ultimately stop decreasing, but that is irrelevant to the mathematical argument.)

What do we do if we have a program in which the loop index -- or whatever corresponds to the loop index -- is a real number? Unless this real number is converging to some value, as in the preceding example, it will have a minimum increment or a minimum decrement. That is, each time around the loop, the expression -- let us call it e -- is increased, or decreased, by no less than some real number x each time around the loop. By taking negatives if necessary, we can assume that e is decreased by at least x, in which case e/x is decreased by at least 1. The integer part of e/x -- that is, the greatest integer in e/x -- is then also

decreased by at least 1. The following program illustrates this:

```
      FUNCTION SIMPS(F, A, B, H)
      EXTERNAL F
      X = A
      S = F(X) + 4.0*F(X+H)
1     X = X + 2.0*H
      IF (X .GE. B) GO TO 2
      S = S + 2.0*F(X) + 4.0*F(X+H)
      GO TO 1
2     S = (S + F(X))/(3.0*H)
      IF (X .EQ. B) GO TO 3
      S = S - (F(B) + 4.0*F((B+X)/2.0) + F(X))/(3.0*(X-B)/2.0)
3     SIMPS = S
      RETURN
      END
```

(The purpose of this program is to use Simpson's rule, with step size H, to calculate the integral from A to B of the function F.) Let us assume that A < B and H > 0 are included in the initial assertion. The variable X increases by 2H each time around the loop, so that -X/H decreases by 2 each time; the greatest integer in -X/H is thus a loop expression for the loop in this program.

Let us now turn to condition (2). The loop expression must decrease every time we go around the loop. The operative words here are "<u>every</u> time." For example, consider the following loop:

```
      I = 10
      J = 0
1     IF (I .LE. 1) GO TO 2
      I = I - 1
```

```
2       J = J + 1
        GO TO 1
```

The integer variable I remains positive at all times. There is an instruction in the loop, I = I - 1, which decreases the value of I; but this instruction is not executed <u>every</u> time through the loop. In fact, of course, the loop does not terminate.

Forward substitution can tell us when an expression decreases. At the end of the loop, we evaluate our expression, using the table we have built up, and subtract this new symbolic value from the initial one. If the result is always positive, the expression decreases. The fact that the result is positive is sometimes quite obvious, as, for example, when a variable which is a loop expression is decreased by 1. However, sometimes it is necessary to use the assertions at the control point in the loop in order to show that the given expression decreases. In our GCD program, the loop expression I + J is decreased either by I or by J each time; thus we need the assertions I > 0 and J > 0 at the control point, in the course of showing that I + J is a loop expression.

If we find an expression which increases, then, as we have already seen, its negative will decrease. What if we have an expression which alternately increases and decreases? First, we must determine intuitively what general direction the expression is headed in (whether upward or downward); then we may analyze the specific method by which the index is changed in order to obtain an expression which always decreases. The initial assertion of the following program is N(1) = 2, N(2) = -1, N(3) =2:

```
      I = 1
      J = 1
1     CALL SUB(I)
      I = I + N(J)
      J = J + 1
      IF (J .EQ. 4) J = 1
      IF (I .LE. 100) GO TO 1
```

This program calls SUB(I) where I = 1, 3, 2, 4, 6, 5, 7, 9, 8, and so on in that order. Without worrying about what SUB does (so long as it does not change the value of I), let us consider whether this program terminates. The variable I alternatively increases and decreases, but its general direction is upward. We can obtain an increasing expression by subtracting 1 from I if it is divisible by 3 and adding 1 to I if MOD(I, 3) = 2. A little experimentation will convince us that 2*I + N(J) is also increasing; it successively takes the values 4, 5, 6, 10, 11, 12, 16, 17, 18, and so on. We can then get a loop expression by taking the negative of either of these.

A related, but simpler, problem occurs when a variable always increases and decreases every time through a loop, as in the following loop:

```
      I = 1
1     I = I - 1
      CALL F(I)
      I = I + 2
      CALL G(I)
      IF (I .LE. 100) GO TO 1
```

Assuming, as before, that the value of I cannot be changed as a

result of the subroutine calls, -I is a decreasing expression (and in fact a loop expression) in this loop. This example points out the generality of considering the behavior of an expression as an entire loop is traversed, rather than requiring, for example, that the value of a loop expression cannot be increased by any individual statement in the loop.

Now consider condition (3) -- the last of the three conditions on a loop expression. We have seen that, in order to insure termination, it is not enough to have a test in the loop which exits if the given expression is zero (or any other final value). We have likewise seen that it is not enough for our expression to be bounded by a variable quantity, because, as the expression decreased, this variable quantity might be decreasing along with it. Sometimes the bound will change but the program will still terminate, as in the following loop:

```
      I = 1
      J = N
1     CALL GET(X)
      IF (F(X) .NE. 0) GO TO 2
      A(I) = X
      I = I + 1
      GO TO 3
2     A(J) = X
      J = J - 1
3     IF (I .NE. J) GO TO 1
```

The loop expression here is J-I. In this case the bound is becoming smaller as the loop index is becoming larger. It is even possible, however, for both the index and the bound to become

larger at the same time, as in the following (admittedly contrived) example:

```
      I = 1
      J = N
1     CALL F(I)
      I = I + 2
      J = J + 1
      IF (I .NE. J) GO TO 1
```

Again the loop expression is J-I; it is easy to see that the value of this expression decreases by 1 every time around the loop. These two examples serve further to illustrate the advantage of our method of loop expressions over any method which allows the bound to be variable and then requires that it not be changed within the loop, or that it be changed only in one direction.

One other necessary condition on loop expressions is that they be bounded from below at the same point at which the decrease is measured. It is not enough for an expression to be decreasing as we go around a loop, when measured from one point, and to be bounded at a different point in that same loop. To see this, consider the following loop:

```
      I = 1
      J = 1
1     I = 0
3     J = J + 1
      I = -J
2     GO TO 1
```

At statement number 3, I is certainly bounded. If we go around the loop from statement 2 back to itself, I decreases; yet the

loop never terminates. Of course, I is not decreasing if measured from statement 3 around the loop and back to statement 3.

Many programs terminate for special reasons. Among these are the following:

(1) A program which calculates the elements of a convergent sequence, and which stops whenever two adjacent elements of the sequence are within a certain tolerance of each other, can be shown to terminate by making use of Cauchy's criterion for convergence. The following program calculates the square root of the positive real number X, to within a strictly positive tolerance TOL, by Newton's method:

```
      Y = 2.0
1     Z = Y
      Y = 0.5*(Y + (X/Y))
      IF (ABS(Z-Y) .GT. TOL) GO TO 1
```

The sequence calculated here is defined by $y_1 = 2$, $y_{i+1} = (y_i + X/y_i)/2$. We know that this sequence converges, and thus, by Cauchy's criterion, there exists, for every positive ϵ (and, in particular, for ϵ = TOL), a positive integer N such that $|y_a - y_b| < \epsilon$ for all a, b > N. The loop expression for this loop is then N-I, where N is defined as above and I is such that y_I (in the sequence defined above) is the current value of Y.

The loop expression in this case is <u>nonconstructive</u>. It involves variables which do not appear in the program, and which, although they are clearly functions of the current state of the computation (in this case, of the current values of X and TOL), have not been given explicit functional definitions. In section - , where the validity of the loop expression method is proved,

it will be seen that a loop expression can be any integer function of the current state. (It is also true that, unless TOL is sufficiently large, this algorithm may, in fact, not terminate on some actual computers because of the finite precision of the quantities involved. This subject will be discussed further in Chapter 16. The partial correctness of the above program is also of interest, since it involves an unusual kind of assertion; this is also discussed in Chapter 16.)

(2) A sorting program, which puts an array in order, may be shown to terminate by considering the total number of pairs of elements in the array (not necessarily adjacent elements) which are out of order. This number is bounded from below by zero, and it may be shown to decrease whenever two out-of-order elements are interchanged or whenever a partial merge process is performed. This subject is considered further in section 10-3.

(3) A program which reads a new card every time around a loop, and which contains a test that exits from the loop when there are no more cards left in the hopper, can be shown to terminate by considering, as a loop expression, the number of cards which remain to be read. This number is bounded below by zero and decreases each time a card is read. For further discussion of this subject, see Chapter 18.

(4) A program which operates upon the elements of a list, moving from one to the next and stopping when the pointer field in the current list item indicates that there are no more items, can be shown to terminate by considering, as a loop expression, the number of items remaining in the list. Such programs require initial assertions that insure that they are not, in fact, working on circular lists. This subject is taken up in Chapter 17.

8-3 Graphical Properties of Programs

The statement of the termination theorems we will use depends on the "closed loop condition," one of several graphical properties of programs which are necessary to an understanding of partial correctness and termination. We shall now define these properties and prove certain important facts about them.

First let us consider programs which "obviously" always terminate. Among these are the straight-line programs, or programs which have no transfer statements. Whenever we execute an n-statement straight-line program, we do the first statement first, then the second statement, and so on through the last statement, so that the program always terminates. There is, however, a more general class of programs which always terminate: those in which all transfers are in the forward direction.

DEFINITION 8-1. A program containing n executable statements $C_1, \ldots, C_n$ is a **branch-forward program** if, whenever statement C_i transfers to C_j, we have $i < j$.

It is obvious that an n-statement branch-forward program always terminates in a finite number ($\leq n$) of steps. We shall now show that there is a sense in which branch-forward programs are the only programs which "obviously" always terminate.

DEFINITION 8-2. A **cycle** (or "closed loop") in a program is a path from some statement in that program to itself. An **acyclic** program is one which does not contain any cycles.

It is clear that, if a program does not terminate, it will eventually execute a statement which it has already executed. Thus a cycle will exist, and hence the program cannot be acyclic. This shows that acyclic programs always terminate.

All branch-forward programs are acyclic; but the converse is not generally true. For example, the following program to calculate the sum of the absolute values of X and Y is acyclic, but it is not a branch-forward program because of the statements GO TO 5 and GO TO 9:

```
      FUNCTION SUMABS(X, Y)
1     A = X
2     IF (A) 3, 5, 5
5     B = Y
6     IF (B) 7, 9, 9
9     SUMABS = A + B
10    RETURN
3     A = -A
4     GO TO 5
7     B = -B
8     GO TO 9
      END
```

We have purposely included a statement number on every statement in this program to show that it is in fact a branch-forward program if we interpret "forward" to mean "in increasing order of statement numbers." In fact, as we will now show, <u>any</u> acyclic program may have its statements ordered in such a way that it becomes a branch-forward program in this sense.

THEOREM 8-1. The following three properties of programs are equivalent:

(a) The statements of the program may be numbered in such a way that all transfers are in the forward direction.

(b) The program contains no cycles.

(c) Every subset of the statements of the program contains an initial statement of the subset (that is, a statement to which no other statement of the subset makes exit).

PROOF: (a) → (b) -- Obvious.

(b) → (c) -- Suppose, on the contrary, that there was a subset containing no initial statement. Choose any element of this subset; call it C_0. Since C_0 is not initial, there is another statement -- call it C_1 -- which makes exit to it. There is then still another statement, C_2, which makes exit to C_1. Continue this process until a statement C_j is reached which is the same as statement C_i, $i < j$. This will always happen because the total number of statements in the program is finite. But then $C_j \rightarrow C_{j-1} \rightarrow \cdots \rightarrow C_i$ $(= C_j)$ is a cycle of the program.

(c) → (a) -- Let C_1 be any initial statement of the program. Since C_1 is initial, there are no transfers to it. Remove C_1 from the program and consider an initial statement of the subset that remains; call this C_2. Clearly there can be no transfers to C_2 except possibly from C_1. Now remove C_2 and repeat the process. Continuing in this way, it is clear that, at each stage, the current statement C_j can have no transfers to it except from statements C_i with $i < j$. Since this is true of every statement in the program, it becomes a branch-forward program relative to the constructed ordering of its statements. This completes the proof.

Next we consider another important property that every program which always terminates may be assumed to have.

DEFINITION 8-3. A program is <u>program-reduced</u> if every one of its executable statements lies on some path from an initial statement to a terminal statement.

Other names for program-reduced programs are programs with eligible flowcharts or with well-formed flowcharts.

Given any program, we may define its program reduction as the program obtained by deleting all statements which are not on some path from an initial node to a terminal node. Clearly the program reduction is itself program reduced. The process by which the program reduction is obtained consists of a finite number of steps, as follows:

1. Construct the set X_0 of all initial nodes of the program.

2. At each stage, construct the set X_i of all nodes to which the nodes of X_{i-1} make exit, and which are not in X_0, ..., X_{i-1}.

3. At some stage, some X_i will be null (since the total number of statements in a program is finite). At that point, all statements not in one of the X_i cannot be reached from an initial node, and may be deleted from the program. (Clearly such a statement in a program will never be executed, if the program is always started from some initial node.)

4. Construct the set Y_0 of all terminal nodes of the program. (We are now doing the previous three steps in reverse.)

5. At each stage, construct the set Y_i of all nodes which make exit to nodes of Y_{i-1}, and which are not in Y_0, ..., Y_{i-1}.

6. At some stage, just as before, some Y_i will be null. At that point, all statements not in one of the Y_i do not lead to any terminal node, and may be deleted. (Such a statement will never be executed in a finite computation, although it might be executed as part of an endless loop.)

In what follows, we shall assume, whenever it is necessary, that every program which we consider is program reduced.

We shall now define the closed loop condition of a program

relative to an arbitrary subset of its statements. Our definition coincides with that given informally in section 7-5, if the subset is taken to be the set of all control points of the program.

DEFINITION 8.4. A program is said to satisfy the closed loop condition, relative to a subset of its executable statements, if every cycle of the program must contain at least one statement in that subset.

Likewise, we need a definition of "control path" in a more general context.

DEFINITION 8.5. A control path of a program, relative to a subset of its executable statements, is a path from one statement in that subset to another one (possibly the same one) which does not contain any other statements of that subset.

This definition agrees with that given informally in section 6-1, if we take for our subset the set of all control points. We are now in a position to prove the statement made about the closed loop condition in section 7-5, together with a partial converse.

THEOREM 8-2. Let there be given a subset of the executable statements of a program which contains all of its initial and terminal statements. Then, relative to that subset, the total number of control paths of the program is finite if it satisfies the closed loop condition; and the converse holds if the program is program-reduced.

PROOF. Suppose the closed loop condition is satisfied. Then any control path is completely determined by its first statement, its last statement, and the set of statements between the two; none of the elements of this set may be repeated (that is, a con-

trol path cannot contain the same intermediate statement more than once) because otherwise we would be able to get a cycle from such a statement to itself, not containing any of the statements in the subset. Since the total number of statements and the total number of subsets of statements in a program are both finite, the total number of control paths is finite. Conversely, suppose that the closed loop condition is not satisfied. Let $C_1, \ldots, C_{k-1}, C_k = C_1$ be a cycle containing no elements of the given subset. Since the program may be assumed reduced, C_1 is on some path from some initial node to some terminal node. On this path, let the two statements within the subset that are closest to C_1 be denoted by C_1' and C_y', so that $C_1', C_2', \ldots, C_x' = C_1$, $C_{x+1}', \ldots, C_y'$ is a control path. Clearly C_1' and C_y' must exist, because the initial and terminal statements are all in the subset; also, by definition, C_1 is not in the subset, so that $x \neq 1$ and $x \neq y$. But we can now obtain an infinite number of control paths, one for each positive integral value of n, by going from C_1' to C_1, then n times around the cycle, ending at C_1, and finally to C_y'. This completes the proof.

The importance of this theorem should be obvious. Our assertion method does not work unless we have a finite number of control paths and thus a finite number of verification conditions. If our set of control points (including initial and terminal points) satisfies the closed loop condition, there will always be a finite number of control paths; and, if the program is reduced (certainly a reasonable enough restriction), the closed loop condition must be satisfied for there to be a finite number of control points. Hence we may always assume, when we are given a proof of partial correctness involving control points and assertions, that the

control points have been chosen in such a way that the closed loop condition is satisfied.

There are a number of mechanical ways in which this may be done. We have already mentioned that, if we choose for our intermediate control points every point to which a backward transfer is made, the closed loop condition will always be satisfied. Further methods of guaranteeing this will be deferred to Exercise 5 at the end of this section.

8-4 Loop Expression Sequences

Now we are ready to begin to formulate conditions for the termination of programs with arbitrary numbers of loops. As a first example, let us consider a program which multiplies two matrices:

```
      SUBROUTINE MATMUL(A, B, C, N)
      DIMENSION A(N, N), B(N, N), C(N, N)
      I = 1
1     J = 1
2     K = 1
      SUM = 0
3     SUM = SUM + A(I, K)*B(K, J)
      K = K + 1
      IF (K .LE. N) GO TO 3
      C(I, J) = SUM
      J = J + 1
      IF (J .LE. N) GO TO 2
      I = I + 1
      IF (I .LE. N) GO TO 1
      RETURN
      END
```

The final assertion of this program is that C(I, J) is equal to the sum of all A(I, K)*B(K, J), $1 \leq K \leq N$, for each I and J in the range $1 \leq I \leq N$, $1 \leq J \leq N$. We shall choose a single intermediate control point, namely statement number 3; clearly every closed loop in this program passes through this statement. At this point, the assertion is that:

(1) For all i less than I, the i-th row of the array C (that is, $C(i, j)$ for $1 \leq j \leq N$) has been calculated.

(2) For all j less than J, the j-th element of the I-th row of C (that is, $C(I, j)$ has been calculated.

(3) SUM is equal to the partial sum, taken over all k less than K, for the element C(I, J) of C.

(4) The indices I, J, and K are in range (that is, between 1 and N inclusive).

Let us rewrite the program with these assertions, using our notation, omitting the SUBROUTINE, DIMENSION, RETURN, and END statements, and with the initial assertion $N > 0$:

```
C  N > 0
       I = 1
1      J = 1
2      K = 1
       SUM = 0
C  1 ≤ I, I ≤ N, 1 ≤ J, J ≤ N, 1 ≤ K, K ≤ N,
C  (C(II, JJ) = SUM(A(II, KK)*B(KK, JJ), KK .IN.
C  (1..N)),  II .IN. (1..I-1), JJ .IN. (1..N)),
C  (C(I, JJ) = SUM(A(I, KK)*B(KK, JJ), KK .IN.
C  (1..N)),  JJ .IN. (1..J-1)),
C  SUM = SUM(A(I, KK)*B(KK, J), KK .IN. (1..K-1))
3      SUM = SUM + A(I, K)*B(K, J)
       K = K + 1
       IF (K .LE. N) GO TO 3
       C(I, J) = SUM
       J = J + 1
       IF (J .LE. N) GO TO 2
```

```
      I = I + 1
      IF (I .LE. N) GO TO 1
C   (C(II, JJ) = SUM(SET(A(II, KK)*B(KK, JJ), KK .IN.
C   (1..N))), II .IN. (1..N), JJ .IN. (1..N))
```

Let us first prove that this program is partially correct. It has five control paths.

The first path starts at the beginning of the program and goes to statement number 3. At the end of this path, $1 \leq I$, $1 \leq J$, and $1 \leq K$ follow from $I = 1$, $J = 1$, and $K = 1$; also $I \leq N$, $J \leq N$, and $K \leq N$ follow from $I = 1$, $J = 1$, $K = 1$, and $N > 0$. The other three parts of the intermediate assertion involve either the set (1..I-1), the set (1..J-1), or the set (1..K-1). All these sets are null in this case, which means that there is nothing to prove (that is, the assertion may be assumed to be true).

The next three paths start and end at the intermediate assertion. The second path leads from statement number 3 through the next two statements and back to statement number 3. This path changes the values of only two variables, SUM and K. The only parts of the intermediate assertion that involve these variables are $1 \leq K$, $K \leq N$, and SUM = SUM(A(I, KK)*B(KK, J), KK .IN. (1..K-1)). The other parts of the intermediate assertion, therefore, remain true at the end of the path if they were true at the beginning. We still have $1 \leq K$ at the end of the path, because K has been increased, and we have $K \leq N$ because of the IF statement (this branch is not taken unless $K \leq N$). The set (1..K) is the disjoint union of the sets (1..K-1) and (K), and the sum of all F(KK) for all KK in the disjoint union

of two sets is the sum of the two sums; that is,

SUM(F(KK), K .IN. (S1 .U. S2)) =

SUM(F(KK), K .IN. S1) + SUM(F(KK), K .IN. S2)

if S1 .I. S2 = (). (Note that this last condition -- that S1 and S2 are disjoint -- is necessary. If S1 = S2, for example, then S1 .U. S2 = S1, and the above relation clearly does not hold unless SUM(F(KK), K .IN. S1) = 0.) This implies that when A(I, K)*B(K, J) is added to SUM at statement number 3, we obtain SUM = SUM(A(I, KK)*B(KK, J), KK .IN. (1..K)); then K is increased by 1, leaving us again with SUM = SUM(A(I, KK)*B(KK, J), KK .IN. (1..K-1)).

We now skip to the fourth path, since the third path will then be more easily understood. The fourth path leads from statement number 3 through the next seven statements, then back to statement number 1, and finally forward to statement number 3 again. At the end of this path, we have $1 \leq J$, $J \leq N$, $1 \leq K$, and $K \leq N$ for the same reasons as in the first path. Also, when this path started, we must have had J = N and K = N. This is because $J \leq N$ and $K \leq N$ by the intermediate assertion, together with the fact that, after increasing both J and K by 1, we had J > N and K > N. By substituting N for K in the intermediate assertion, we obtain that we had SUM = SUM(SET(A(I, KK)*B(KK, J), KK .IN. (1..N-1))) at that time, and thus, by an argument similar to that used for the second path, we see that SUM = SUM(A(I, KK)*B(KK, J), KK .IN. (1..N)) just before the statement C(I, J) = SUM. Just after this statement, therefore, we have C(I, J) = SUM(A(I, KK)*B(KK, J), KK .IN. (1..N)). Now, by substituting N for J in the intermediate as-

sertion and in the preceding sentence, we have

```
(C(I, JJ) = SUM(A(I, KK)*B(KK, JJ), KK .IN. (1..N)),
    JJ .IN. (1..N-1))  and
C(I, N) = SUM(A(I, KK)*B(KK, N), KK .IN. (1..N))
```

These two assertions, taken together, are equivalent to

```
(C(I, JJ) = SUM(A(I, KK)*B(KK, JJ), KK .IN. (1..N)),
    JJ .IN. (1..N))
```

as may be shown formally by using the fact that (1..N) is the disjoint union of (1..N-1) and (N), together with the fact that

```
SUM(F(X), X .IN. (S1 .U. S2)) =
   SUM(F(X), X .IN. S1) + SUM(F(X), X .IN. S2)
```

and that

```
(P(X), X .IN. (S1 .U. S2)) =
   (P(X), X .IN. S1) and (P(X), X .IN. S2)
```

for any two disjoint sets S1 and S2, function F, and predicate P. In turn, the assertion just determined, together with

```
(C(II, JJ) = SUM(A(II, KK)*B(KK, JJ), KK .IN. (1..N)),
      II .IN. (1..I-1), JJ .IN. (1..N))
```

is equivalent to

```
(C(II, JJ) = SUM(A(II, KK)*B(KK, JJ), KK .IN. (1..N)),
      II .IN. (1..I), JJ .IN. (1..N))
```

for the same reasons. This is then true just before the statement I = I + 1, after which we recover the original intermediate assertion. (The remainder of this assertion is now immediate;

we have $1 \leq I$ because I has increased, we have $I \leq N$ because of the final IF statement, and the rest of the intermediate assertion involves (1..J-1) and (1..K-1), both of which, as before, are null.)

The third path in this program is, in a sense, a cross between the second path and the fourth. It leads from statement number 3 through the next five statements, then back to statement number 2, and forward to statement number 3 again. The handling of the final assertions $1 \leq I$, $I \leq N$, and (C(II, JJ) = SUM(A(II, KK)*B(KK, JJ), KK .IN. (1..N)), II .IN. (1..I-1), JJ .IN. (1..N)) of this path resembles their handling in the second path, while the handling of the final assertions $1 \leq K$, $K \leq N$, and SUM = SUM(A(I, KK)*B(KK, J), KK .IN. (1..K-1)) of this path resembles their handling in the fourth path. The remaining assertions, involving J, are handled somewhat as the analogous ones involving I were handled in the fourth path.

The fifth and last path leads from statement number 3 to the end of the program. As in the fourth path, we must have had J = N and K = N when this path started, and also I = N. Thus the final assertion of this path holds, since it is obtained by substituting N for I in one of the intermediate assertions which was proved to hold at the end of the fourth path. This completes the proof of partial correctness. (The assertions $1 \leq I$, $I \leq N$, $1 \leq J$, $J \leq N$, $1 \leq K$, and $K \leq N$ at statement number 3 assure us that each subscripted variable is always referenced with both subscripts in range.)

How can we prove that the program terminates? There are three loops, with loop expressions N-I, N-J, and N-K respec-

tively. All of these are integer expressions, bounded below by zero at statement number 3, and each one decreases as we go around its respective loop. However, these facts are not sufficient to prove termination. Consider the following program:

```
1     IF (K .EQ. 0) GO TO 2
      K = 0
      I = I + 1
      J = J - 1
      GO TO 1
2     K = 1
      I = I - 1
      J = J + 1
      GO TO 1
```

Here there are two loops; as we go around one of them, I decreases, and around the other one J decreases. Also, by the logic of the program, I and J keep "bouncing" up and down by one, and hence are bounded. Yet the loops never terminate.

To show what further conditions we should impose on loop expressions in the presence of multiple loops, let us examine the behavior of <u>all three</u> loop expressions in our matrix program, as we go around the three loops:

	Expression N-I	Expression N-J	Expression N-K
Inner loop	Remains the same	Remains the same	Decreases
Middle loop	Remains the same	Decreases	Reset
Outer loop	Decreases	Reset	Reset

This behavior is exactly the same as the behavior of the digits in a three-digit integer when that integer is decreased. There

will always be one digit which is decreased; the digits, if any, to the left of that digit will remain the same, and those to the right of it are reset (to 9, for decimal integers). In general, if the elements of any set may be placed in order, the n-tuples of elements of that set may be placed in lexicographical order. The n-tuple $(a_1, \ldots, a_n)$ precedes the n-tuple $(b_1, \ldots, b_n)$ if $a_1 < b_1$, or if $a_1 = b_1$ and $a_2 < b_2$, or, in general, if $a_i = b_i$ for $1 \leq i \leq k-1$ and $a_k < b_k$, for some k, $1 \leq k \leq n$.

Lexicographical ordering is normally applied to either the digits in an integer or the letters in a word. We may, however, apply it to n-tuples of integers. If we consider the n-tuple of current values of N-I, N-J, and N-K, then, as we go around any of our three loops, this n-tuple is decreased -- if by "decreased" we mean that the new n-tuple, after we have gone around the loop, is less than it in lexicographical order. We are thus led to the concept of a loop expression sequence -- a sequence of integer expressions which are bounded from below by constants at a given control point in a loop, and such that the sequence decreases, in the above sense, every time we go around the loop from that control point to itself, assuming that the assertion given at that point is valid. As before, if a program has a single intermediate control point, it always terminates when started with its initial assertion valid if there is a sequence of integer expressions which is a loop expression sequence for all loops in the program. In particular, our matrix multiplication program has been shown to terminate.

8-5 General Termination Conditions

So far, we have restricted our discussion of termination to programs having only a single intermediate control point. In such a program, each loop corresponds to a single control path from that point to itself. In general, there will be several intermediate control points, and a control path may extend either from one of these points to itself, or to another such point. In this case, the method of loop expression sequences, as presented in the preceding chapter, generalizes in the following way. There will be a sequence of expressions associated with each control point, and we must consider the behavior of these expressions as we go along an arbitrary control path, rather than around an arbitrary loop. Such a control path is now associated with two sequences -- one at the beginning and one at the end. Our conditions are now that:

(1) The sequence associated with the control point at the end of a control path must be less, in lexicographical order, than the sequence associated with the control point at the beginning, whenever that control path is actually taken.

(2) Each expression in each sequence must be bounded below, by a constant, at its associated control point.

It is not necessary that all sequences have the same length. Lexicographical ordering may be applied to two sequences of different lengths (as, for example, two words in a dictionary). The sequence $(a_1, \ldots, a_n)$ is less than the sequence $(b_1, \ldots, b_m)$ if there exists k, $k \leq n$ and $k \leq m$, with $a_k < b_k$ and $a_i = b_i$ for $1 \leq i \leq k-1$, or if $n < m$ and $a_i = b_i$ for $1 \leq i \leq n$. (This last rule is what, for example, makes the word HIGH

precede the word HIGHWAY in the dictionary.)

Before showing how one would construct such sequences in practical cases, let us prove that, if the above conditions are satisfied, the given program must always terminate when started with its initial assertion valid. Suppose the contrary, and pick some initial condition under which the program does not terminate. The execution of the program follows the control paths; at each control point, we may evaluate the expressions in the sequence given at that point, and in this way we obtain an infinite sequence of sequences which is decreasing in lexicographical order. This, however, cannot happen if the expressions in the sequences are bounded from below. (The mathematical reason for this is that the set of all such sequences is a well-ordered set under lexicographical order.)

We will now give a general method for constructing these sequences which is applicable to programs containing more or less "conventional" loops -- those, for example, which are equivalent to FORTRAN DO loops. We shall assume that each loop has a loop expression, and that the given program is arranged in such a way that there are no backward transfers, except for those associated in the natural way with the ends of loops. The method is as follows.

1. For every <u>innermost</u> loop, do the following:

   a. Associate the loop expression of the loop with every control point in the loop (as one element of the sequence).

   b. As the next element of the sequence, associate a small integer, as follows. Start at the control point in the loop which is nearest to, but preceding, the point at which the loop expression is decreased. If there are

no such control points, start at the last control point in the loop. Associate with this point the integer 1. Continue backwards through the loop and around it, and associate with the other control points in the loop the integers 2, 3, 4, etc., in the order in which these control points are encountered. Note that there must be at least one control point in every loop; if there is only one, this step may be skipped for that particular loop.

2. Find a loop containing only loops which have already been treated (if there are none such, skip to step 3), and, for this loop, do the following:

a. Associate the loop expression of the loop with every control point in the loop, including those in inner loops (these already have sequences, which will be appended later).

b. As the next element of the sequence, associate a small integer, as follows. Start as in step 1b and associate with the point thus determined the integer 1. If this point is within an inner loop, append the loop expression (from the previous step) and the integer 1 onto the beginning of the sequence already determined for that inner loop. Now proceed backwards around the loop, again as in step 1b. If the next control point encountered is in the same inner loop, associate with it the same integer; otherwise, the next higher integer. In either case, if that control point is within any inner loop, append the loop expression and this integer onto the beginning of the sequence already determined. Continue in this fashion all the way around the loop.

c. Find another loop containing only loops which have already been treated, and return to step 2a.

3. Finally, start at the physical end of the program, assigning to the control point there the integer 1, and proceeding backwards through the program as in step 2b, assigning a small integer to each control point and putting this small integer on the front of the sequence already found at that point. This process ends when we reach the beginning of the program.

This process will now be illustrated for the sieve program of section 7-5. In order to provide a better illustration, we will assume initially that every statement in this program is a control point, and determine sequences at each of these points. Later, we will show how to determine sequences at only the control points given in section 7-5.

There are two inner loops in this program. The first reads

```
1     A(I) = .TRUE.
      I = I + 1
      IF (I .LE. N) GO TO 1
```

The loop expression of this loop is N-I, so we associate this with every control point. Let us agree to associate the sequence of expressions $\alpha_1, \ldots, \alpha_n$ with a control point by writing

```
C  SEQ(α1, ..., αn)
```

just before that control point. Our inner loop thus reads

```
C  SEQ(N-I)
1     A(I) = .TRUE.
```

```
C  SEQ(N-I)
       I = I + 1
C  SEQ(N-I)
       IF (I .LE. N) GO TO 1
```

The loop expression is decreased at the statement I = I + 1. Therefore, just before that statement, we place the integer 1 as part of our sequence. Working backwards and around the loop, we place the next higher integer, namely 2, at the beginning of the loop, and 3 at the end, as follows:

```
C  SEQ(N-I, 2)
1      A(I) = .TRUE.
C  SEQ(N-I, 1)
       I = I + 1
C  SEQ(N-I, 3)
       IF (I .LE. N) GO TO 1
```

Let us see why this works. When we execute A(I) = .TRUE., the sequence is decreasing -- that is, the value of (N-I, 1), the sequence after this statement, is less, in lexicographical order, than the value of (N-I, 2), the sequence before this statement. This is because N and I do not change, so the value of N-I remains the same, but 1 is less than 2. When we execute I = I + 1, the sequence is decreasing, because the value of N-I after this statement is less than its value beforehand. (It should be clear that the fact that 3 is <u>greater</u> than 1 does not matter, because of the definition of lexicographical order.) Finally, when we execute the IF statement, if we go back to the beginning of the loop, the sequence is decreasing, since 2 is less than 3 and neither N nor I is changed when the IF statement is executed.

Hence, for this loop, the sequence is decreasing in all cases.

Treating the other inner loop in exactly the same way, one obtains:

```
C  SEQ(N-J, 2)
3      A(J) = .FALSE.
C  SEQ(N-J, 1)
           J = J + I
C  SEQ(N-J, 3)
           IF (J .LE. N) GO TO 3
```

This time, in order to show that the sequence is decreasing when J = J + I is executed, we need the assertion that I > 0 at that point. This assertion, of course, is proved as part of the proof of partial correctness.

We now have to find a loop that contains only loops that have already been treated, if there is one. In this program, there is only one more loop, namely the loop which starts at statement number 2. (In the matrix multiplication program of the preceding section, we would be obliged, by this rule, to treat the loops in the order "inner, middle, outer.") What is the loop expression of this loop? In fact, either N-I or N-I*I will do; the latter seems to be slightly easier to verify, so let us use it. Leaving out, for the moment, the sequences we have already determined for the inner loop, our outer loop looks like this:

```
C  SEQ(N-I*I)
2      J = I*I
C  SEQ(N-I*I)
           IF (J .GT. N) GO TO 4
C  SEQ(N-I*I)
3      A(J) = .FALSE.
```

```
C  SEQ(N-I*I)
       J = J + I
C  SEQ(N-I*I)
       IF (J .LE. N) GO TO 3
C  SEQ(N-I*I)
       I = I + 1
C  SEQ(N-I*I)
       GO TO 2
```

Now, as before, we start just above the statement (in this case it is I = I + 1) that decreases the value of the loop expression. At this point we assign the integer 1. At the next point (just before the IF statement) we assign the integer 2, and we append the two quantities thus found at this point -- namely N-I*I and 2 -- onto the beginning of the sequence already determined at that point in the preceding step, which was (N-J, 3). The result is (N-I*I, 2, N-J, 3). We do the same thing in the other statements of the inner loop, remembering not to increase the integer 2 until we get out of the inner loop. The result is that this integer 2 is assigned to the next two points, then the integers 3 and 4, and finally, as we go around the loop (just before GO TO 2), the integer 5. The result is:

```
C  SEQ(N-I*I, 4)
2      J = I*I
C  SEQ(N-I*I, 3)
       IF (J .GT. N) GO TO 4
C  SEQ(N-I*I, 2, N-J, 2)
3      A(J) = .FALSE.
C  SEQ(N-I*I, 2, N-J, 1)
       J = J + I
```

```
C  SEQ(N-I*I, 2, N-J, 3)
      IF (J .LE. N) GO TO 3
C  SEQ(N-I*I, 1)
      I = I + 1
C  SEQ(N-I*I, 5)
      GO TO 2
```

This works for the same reason as before; when a statement is executed, the sequence always decreases. The controlled expression of the inner loop is not affected within the outer loop. (We can see why we held the integer 2 constant over the inner loop if we consider what happens when we execute the IF statement in the inner loop and transfer to statement number 3.)

Finally, we start at the end of the program, assigning the integer 1; then the integer 2 to all the statements of the loop we just analyzed; the integer 3 to the statement I = 2; the integer 4 to all the statements of the other inner loop; and finally the integer 5 to the beginning of the program. As before, we append these integers onto the front of the already existing sequences. The final result is:

```
C  SEQ(5)
      I = 1
C  SEQ(4, N-I, 2)
1     A(I) = .TRUE.
C  SEQ(4, N-I, 1)
      I = I + 1
C  SEQ(4, N-I, 3)
      IF (I .LE. N) GO TO 1
C  SEQ(3)
      I = 2
```

```
C  SEQ(2, N-I*I, 4)
2     J = I*I
C  SEQ(2, N-I*I, 3)
      IF (J .GT. N) GO TO 4
C  SEQ(2, N-I*I, 2, N-J, 2)
3     A(J) = .FALSE.
C  SEQ(2, N-I*I, 2, N-J, 1)
      J = J + I
C  SEQ(2, N-I*I, 2, N-J, 3)
      IF (J .LE. N) GO TO 3
C  SEQ(2, N-I*I, 1)
      I = I + 1
C  SEQ(2, N-I*I, 3)
      GO TO 2
C  SEQ(1)
4     CONTINUE
```

Suppose now that we wish to have control points only at the beginning and at the numbered statements. We would start with the inner loops as before, and would assign the sequences (N-I) and (N-J); step 1b may be skipped in both cases. Now we would do the outer loop, and assign the integer 1 at statement number 3 and the integer 2 at statement number 2, in addition to the loop expression of that loop. Finally, we would assign the integer 1 at the end of the program, 2 at statements 2 and 3, 3 at statement 1, and 4 at the beginning. The sequences would be:

At the beginning -- (4)

At statement number 1 -- (3, N-I)

At statement number 2 -- (2, N-I*I, 2)

At statement number 3 -- (2, N-I*I, 1, N-J)

At statement number 4 -- (1)

NOTES

The fact that termination of programs must be shown separately, if one is using Floyd's basic method of proving partial correctness, appears in [Floyd 67]. He very briefly indicates a method of proving termination which is essentially the one given here, and illustrates it for a one-loop program.

A much more thorough study of termination of algorithms was carried out by Manna in his thesis [Manna 68]; the basic results are given in [Manna 70]. Three examples are given in this paper; in one of these, there is only a single intermediate control point, while the other two each involve three nested loops, as in our matrix multiplication program. The methods which Manna illustrates are essentially our methods of loop expressions and loop expression sequences. The method of section 8-5, for more general loops, was conveyed to the author in a private communication by Manna.

In another series of papers (see [Manna 69] and [Manna and Pnueli 70], for example), both the general problem of partial correctness and the general problem of termination are shown equivalent to certain problems in second-order logic. The term "partial correctness" first occurs in [Manna 69].

The term "eligible flowchart" in section 8-3 (referring to a flowchart with no unreachable statements, etc.) appears in [Karp 60]; the term "well-formed flowchart," for somewhat the same thing, appears in [Bjorner 70]. The closed loop condition is briefly suggested in [Floyd 67] and used in King's program verifier [King 69].

EXERCISES

1. Prove the correctness (partial correctness and termination) of the following program (slightly modified from one given in section 6-5):

```
      K = 1
1     B(K) = A(K)
      K = K + 1
      IF (K .LE. N) GO TO 1
```

Be sure to include a proof that the subscripts are in range every time a subscripted variable is referenced.

2. Prove the correctness of the following program (slightly modified from one given in section 7-2):

```
      I = 0
1     I = I + 1
      IF (X .EQ. A(I)) GO TO 3
      IF (I .NE. N) GO TO 1
```

Assume that the assertion after the last statement of the program is (X .NE. A(I), I .IN. (1..N)), while the assertion at statement number 3 is X = A(I).

3. Prove the correctness of the following program (slightly modified from one given in section 7-2):

```
      I = 1
      S = A(1)
1     I = I + 1
      S = S + A(I)
      IF (I .LT. N) GO TO 1
```

Be sure you choose the initial assertion properly.

4. The following program, given with assertions, is a modification of that given in problem 1 above:

```
      K = 0
1     K = K + 2
      B(K-1) = A(K-1)
      B(K) = A(K)
      IF (K .NE. N) GO TO 1
```

The point of the modification, of course, is that it takes the computer just as long to do K = K + 1 as K = K + 2, and we need to do K = K + 2 only half as many times as we would need to do K = K + 1. There is, however, a bug in the above program.

(a) Try to prove the correctness of this program and show exactly where a proof breaks down.

(b) Fix the bug, while retaining the statement K = K + 2 and the property of it mentioned above.

(c) Prove the correctness of the resulting program.

5. Show informally, using the arguments of section 8-3, that the total number of control paths in a program will be finite if we take as our control points the initial and terminal statements of the program, together with

(a) all labeled statements of the program;

(b) all IF statements and other conditional statements of the program;

(c) all backward transfer (conditional and unconditional) statements in the program. (Note: Exercises 5a, 5b, and 5c are three separate exercises. It is not meant that all statements of all three types should be considered simultaneously.)

6. Give an example to show that a program with a finite total number of control paths need not have a control point in each closed loop unless it is program-reduced.

7. In the matrix multiplication routine of section 8-4, we used the variable SUM as a partial sum, rather than C(I, J), for the sake of efficiency. Suppose, however, that we do not do this; that is, we eliminate the statement C(I, J) = SUM, and replace SUM by C(I, J) everywhere else in the program. How does this change affect the proof of partial correctness?

8. In the matrix multiplication routine of section 8-4, we used I as the outer loop index and J as the middle loop index. A good FORTRAN programmer might notice that, if we used J as the outer loop index and I as the middle loop index, an optimizing FORTRAN compiler might produce faster code, since passing from C(I, J) to C(I+1, J) (rather than to C(I, J+1)) involves incrementing an index register by 1 (or by 4 on the IBM 360 and 370), due to the way in which FORTRAN arrays are stored. How would this change affect the proof of partial correctness?

9. Determine sequences of expressions, using the method of section 8-5, for the matrix multiplication program of section 8-4, assuming that every statement in this program is a control point.

10. Consider the sequences of expressions determined at the end of section 8-5 for the initial statement and the labeled statements in the sieve program. Complete the proof of termination of the sieve program by showing that, for each control path of this program, the sequence is decreasing (in the sense used in this section).

C H A P T E R N I N E

M A C H I N E L A N G U A G E

9-1 Pure Procedures

In order to make use of our program correctness techniques in practical situations, we shall have to be able to prove the correctness of machine language programs. In particular, we shall have to be able to prove that the machine language object code of a compiler is equivalent to the corresponding source code. Clearly, unless this is true, a "correct" program written in an algebraic or other higher level language may still give wrong answers when compiled and executed.

In studying machine-language program correctness, we must deal with self-modification and integer and floating point arithmetic. As we shall see, the concept of a restricted command, as introduced in section 6-5, provides us with a powerful general method of treating all of the above phenomena. Of these, self-modification is the most fundamental. Many machine-language programs perform no arithmetic operations, and are thus not subject to roundoff, truncation, or overflow considerations; on the other hand, even when a machine language program does not modify itself -- that is, when it is a pure procedure -- that fact must be proved.

The basic principle behind proving that a procedure is pure is quite simple. Suppose that a program P does in fact modify itself; then it must be modified by one of its own instructions.

(Even a pure procedure, of course, might be modified by some other procedure; this possibility is discussed in section 20-..) If we wish to prove that P does not modify itself, therefore, we look at each instruction of P and prove that that instruction cannot modify P. Moreover, we may assume, by induction, that the given instruction has not itself been modified yet. Let us now cast this intuitive argument in formal terms.

Let a be an instruction word in some computer. (That is, a denotes the space taken up by one instruction. This will be the same as a data word on the UNIVAC 1108 or the PDP-10; it will be two, four, or six bytes on the IBM 370; either one half or one fourth of a word on the CDC 6600, and so on.) We may regard a as a variable, whose value w is presumably the code for some instruction I. Since a is a variable, we may regard a=w as an assertion. The instruction I will be performed only if the assertion a=w holds just before it is to be performed. Thus every instruction in machine language may be viewed as a restricted command. To say that the instruction I has been modified is precisely to say that the assertion a=w fails to hold.

Now let the set of all instruction words in the program P be $a_1, \ldots, a_n$, and let the corresponding values be $w_1, \ldots, w_n$, and the corresponding instructions $I_1, \ldots, I_n$. The assertion

$$(a_K = w_K,\ K \text{ .IN. } (1..n))$$

says that "so far, the program has not been modified"; or, in other words, every instruction word in the program has the same contents as it had when the program started. Let us now include this as an assertion at every control point in the program. The verification

condition of each control path must now include a verification of this assertion at the end of each path, under the hypothesis that it is satisfied at the beginning of that path.

In many machine language programs, the above verification may be performed by reference to the individual instructions of the program. Most of the instructions in the repertoire of a typical computer do not change the contents of words in memory. These include loading, shifting, most arithmetic and logical instructions, and conditional transfers. Even those which do alter memory, such as store instructions and "add to memory" or "increment memory" instructions, very often alter only certain fixed cells in memory (if, in particular, they are not indexed). If all of the instructions in a program are of the above types, then the program cannot modify itself, no matter how it works or what it does. We shall refer to such a program as an absolutely pure procedure.

We shall now give an example of the proof of correctness of an absolutely pure procedure. We shall use the machine of section 3-5, and the GCD program of section 6-1 as it might be compiled for that machine. Our program, with assertions, is as follows:

Address (hexadecimal)	Contents (hexadecimal)	Mnemonics and assertions
100	----	I RE 1
101	----	J RE 1
102	----	M RE 1
103	----	N RE 1

```
                                * M > 0, N > 0
104             1102                LD  M
105             7100                ST  I
106             1103                LD  N
107             7101                ST  J
                                * GCD(I,J) = GCD(M, N),
                                * I > 0, J > 0
108             1100            U   LD  I
109             5101                SU  J
10A             C110                TM  V
10B             6001                SUI 1
10C             C114                TM  W
10D             4001                ADI 1
10E             7100                ST  I
10F             A108                TR  U
110             1101            V   LD  J
111             5100                SU  I
112             7101                ST  J
113             A108                TR  U
                                * I = GCD(M, N)
114             ----            W
```

This program uses the instructions LD, ST, SU, TM, SUI, ADI, and TR. Of these, only ST (store) alters the contents of any word in memory. The four ST instructions in this program alter only the contents of I and J, which are declared by means of the pseudo-operation RE (RE n reserves n words of memory). Thus the above program is an absolutely pure procedure, and the proof of its correctness proceeds along standard lines. Let us consider in de-

tail one of the control paths, namely the one that proceeds from address 108 to address 10F, and back to address 108:

```
* (ICA), GCD(I, J) = GCD(M, N), I > 0, J > 0
      LD  I              ac = I
      SU  J              ac = ac - J
      (ac ≥ 0)
      SUI 1              ac = ac - 1
      (ac ≥ 0)
      ADI 1              ac = ac + 1
      ST  I              I = ac
* (ICA), GCD(I, J) = GCD(M, N), I > 0, J > 0
```

We have explicitly introduced here an instruction constancy assertion (ICA). This is the assertion, mentioned earlier, which states that the instruction words of the program -- in this case, the sixteen cells with hexadecimal addresses 104 through 113 -- have the same contents they had at the beginning of the program, which in this case are given by the table in hexadecimal. We must first show that ICA is preserved by the path. Now ICA implies, in particular, that cell 108 contains the instruction code for LD I. This instruction changes only the accumulator (denoted, as before, by ac), and thus ICA remains true after it is executed. In turn, since ICA implies that cell 109 contains the instruction code for SU J , this instruction is executed properly, and so on through the end of the path. The last instruction, ST I, alters cell 100, which is not one of the cells (104 through 113) in the definition of ICA, and thus, again, ICA is preserved.

We have given each instruction in the path with its algebraic

language equivalent, as in section 3-5. Note that we have employed a programming "trick" to circumvent the fact that our computer has no test for zero -- we test for minus, then subtract one and test for minus again. If the first test fails and the second succeeds, the accumulator must have been zero before subtracting 1. In this case, however, I - J was strictly positive, and we added the 1 back on again to get the original result (I - J) to store in I. Forward substitution produces the following table:

Variable	I	J	ac
Initial value	I	J	ac
Value after LD I	I	J	I
Value after SU J	I	J	I-J
(ac $\geq$ 0) becomes (I - J $\geq$ 0)			
Value after SUI 1	I	J	I-J-1
(ac $\geq$ 0) becomes (I - J - 1 $\geq$ 0)			
Value after ADI 1	I	J	I-J
Final value	I-J	J	I-J

The final assertion becomes

(ICA), GCD(I-J, J) = GCD(M, N), I-J > 0, J > 0

and this must be implied by the initial assertion and the two intermediate conditions as modified above by substitution, that is,

(ICA), GCD(I, J) = GCD(M, N), I > 0, J > 0, I-J $\geq$ 0, I-J-1 $\geq$ 0

Of course, since I and J are assumed to be integers, I-J-1 $\geq$ 0 is equivalent to I-J > 0, or I > J. The proof now proceeds much as in section 6-1.

9-2 Verified Purity

Absolutely pure procedures may not contain any instructions which are capable of altering arbitrary cells in memory. Among these are the indexed store instructions, necessary for implementing assignments to subscripted variables. However, there are other classes of pure procedures in which such instructions may appear. Consider, for example, the class of programs produced as object code by a compiler with a subscript range checking feature. (Most ALGOL compilers have this feature; most FORTRAN compilers do not. With PL/I, we would consider only those object programs corresponding to PL/I source programs in which the SUBSCRIPTRANGE condition is valid over the entire program.) Given such a program, even if we do not know how it works or what it does, we may assume, provided the compiler is correct, that the program does not modify itself. In particular, every assignment to a subscripted variable is always within the range of that variable, or else the object program stops at a run-time error exit.

The method by which it is proved that no member of such a class of programs can modify itself will be called a local verification algorithm. There are many local verification algorithms, each one corresponding to some class of locally verifiable pure procedures. A typical such algorithm proceeds as follows. Let us first assume that all (conditional and unconditional) transfer instructions in a program transfer to fixed addresses in the program -- that is, there are no indexed transfer instructions, such as computed GO TO or subroutine return instructions. In this case we may identify all locations in the program to which transfer may be

made. Now suppose that the instruction words of the program are a_1, ..., a_n, and that a_j is an indexed store instruction. We assume that no instruction in the program can transfer to a_j. Let i be the largest integer with $i < j$ such that transfer may be made to a_i. Then all of the instructions a_i, a_{i+1}, ..., a_{j-1} must always be executed just before a_j. Suppose that, if these instructions are executed in the given order, a_j cannot modify any instruction word of the program. If we can prove this for each instruction like a_j in the program, we have proved that the program is a pure procedure.

As an example of such a program, let us consider the example of section 6-5 as it might be compiled for an 18-bit machine whose instruction words have 2-bit index register fields, and which are otherwise like those of the 16-bit machine of section 3-5. We shall assume that the fields of the instruction word are in the following order: index register, 2 bits; operation code, 4 bits; address, 12 bits. All operation codes are the same as before; an indexed address is followed by a comma and then the index register number, from 1 to 3, whereas zero in the index register field, as usual, specifies no index modification. The index registers 1 to 3 are addressable and occupy cells 1 to 3 respectively. The program is as follows:

Address (octal)	Contents (octal)	Mnemonics and assertions
1000	------	A RE 100
1144	------	B RE 100
1310	------	N RE 1
		* N > 0, N ≤ 100
1311	020000	LDI 0
1312	070001	ST 1

```
                                    * (A(K)=B(K), K .IN. (1..X1-1)),
                                    * 0 ≤ X1, X1 < N, N ≤ 100
1313        150001          U   IN  1
1314        010001              LD  1
1315        060001              SUI 1
1316        141327              TM  E
1317        060144              SUI 100
1320        131327              TP  E
1321        210777              LD  A-1,1
1322        271143              ST  B-1,1
1323        010001              LD  1
1324        051310              SU  N
1325        141313              TM  U
                                    * (A(K) = B(K), K .IN. (1..N))
1326        ------
1327                            E
```

This routine has two exits -- the normal exit, at address 1326, and the error exit, at address 1327. In fact, the error exit is never taken; but we do not have to know this in order to prove that this is a pure procedure. The points to which transfer may be made are U (address 1313) and E (address 1327). The only indexed store instruction in the program is at address 1322, and thus all instructions from 1313 through 1321 must be executed just before this store instruction. Let us write these instructions as if they were a control path with no initial assertion:

```
IN  1       X1 = X1 + 1
LD  1       ac = X1
```

SUI 1	ac = ac - 1
(ac ≥ 0)	
SUI 100	ac = ac - 100
(ac < 0)	
LD A-1,1	ac = A(X1)
ST B-1,1	B(X1) = ac

Here and in the assertions of the program, X1, of course, refers to index register 1. Forward substitution applied to this path yields the following table:

Variable	ac	X1	A(X1+1)	B(X1+1)
Initial value	ac	X1	A(X1+1)	B(X1+1)
Value after IN 1	ac	X1+1	A(X1+1)	B(X1+1)
Value after LD 1	X1+1	X1+1	A(X1+1)	B(X1+1)
Value after SUI 1	X1	X1+1	A(X1+1)	B(X1+1)
(ac ≥ 0) becomes (X1 ≥ 0)				
Value after SUI 100	X1-100	X1+1	A(X1+1)	B(X1+1)
(ac < 0) becomes (X1-100 < 0)				
Value after LD A-1,1	A(X1+1)	X1+1	A(X1+1)	B(X1+1)
Value after ST B-1,1	A(X1+1)	X1+1	A(X1+1)	A(X1+1)

The indexed store instruction, ST B-1,1 , alters address 1143 modified by the contents of register X1. The symbol X1 above, of course, refers to the original contents of this register; its current contents are X1+1. The two conditions, X1 ≥ 0 and X1-100 < 0, are equivalent to 1 ≤ X1+1 ≤ 100; that is, at the time ST B-1,1 is executed, the index will be between 1 and 100, inclusive. This, of course, is the point of the instructions U+1 through U+5 -- to verify that the subscript is in this range before making reference to the subscrip-

ted variables. In particular, the address of the store instruction is always outside the program's instruction word area, and the program is therefore a pure procedure.

We may summarize our local verification algorithm as follows:

(1) Determine all points in the program to which transfer may be made.

(2) For each indexed store instruction in the program, write a control path directly to it from the nearest of these points.

(3) Verify that, if this path is followed, the indexed store instruction will never modify any of the program's instruction words. Do this for each indexed store instruction in the program.

Note that we have given assertions for our program, and may proceed with an analysis of its control paths, just as in the preceding section. However, such an analysis is not necessary in order to prove that the program is a pure procedure.

What if there are indexed transfers in a program? In this case, we must have some other way of determining all points in the program to which transfer may be made. (Otherwise we might, for example, transfer directly to an indexed store instruction without knowing anything about the contents of the index register.) This leads us to further local verification algorithms, depending upon the particular type of indexed transfer we have in mind. Suppose, for example, that we wish to admit an implementation of the computed GO TO in FORTRAN (or the use of a switch in ALGOL). This is a construction which may transfer to one of n locations, dependent upon whether the value of a certain integer variable (let us call it k) is 1, 2, ..., n. Let us implement such a construction by testing to make sure that $1 \leq k \leq n$ before the indexed transfer is made. If this test is always carried out in a fixed way, we may

scan the code immediately preceding the indexed transfer in order to determine the value of n, which in turn allows us to obtain exactly n points to which indexed transfer may be made. Another slightly more efficient local verification algorithm assumes that every one of these n points, except perhaps the last one, will itself be a transfer (this is normally the case). Hence these points may be omitted in our consideration of all points to which transfer may be made, except that any point immediately following an unconditional transfer is then counted as such a point.

Local verification algorithms for subroutine return instructions depend in an essential way upon the subroutine call and return conventions. This subject will be taken up in section - .

It is, of course, possible to write a pure procedure whose purity cannot be shown by local algorithms. For example, let us remove from the preceding example the instructions U+1 through U+5, obtaining the following program:

Address (octal)	Contents (octal)	Mnemonics and assertions
1000	------	A RE 100
1144	------	B RE 100
1310	------	N RE 1
		* N > 0, N ≤ 100
1311	020000	LDI 0
1312	070001	ST 1
		* (A(K) = B(K), K .IN. (1..X1-1))
		* 0 ≤ X1, X1 < N, N ≤ 100
1313	150001	U IN 1
1314	210777	LD A-1,1

1315	271143	ST	B-1,1
1316	010001	LD	1
1317	051310	SU	N
1320	141313	TM	U

* (A(K) = B(K), K .IN. (1..N))

1321 ------

Let us consider a typical control path of this program, namely the one which extends from address 1313 to address 1320 and back to address 1313:

* (ICA), (A(K) = B(K), K .IN. (1..X1-1)), 0 ≤ X1, X1 < N, N ≤ 100

IN	1	X1 = X1 + 1
LD	A-1,1	ac = A(X1)
ST	B-1,1	B(X1) = ac
LD	1	ac = X1
SU	N	ac = ac - N

(ac < 0)

* (ICA), (A(K) = B(K), K .IN. (1..X1-1)), 0 ≤ X1, X1 < N, N ≤ 100

As before, we have explicitly included (ICA), the instruction constancy assertion, which in this case states that the eight instruction words with addresses 1311 through 1320 have the contents given in the table. This time, however, the validity of (ICA) at the end of the path depends upon the validity of certain other assertions at the beginning of the path -- in this case 0 ≤ X1 and X1 < 100 (which follows from X1 < N and N ≤ 100). After increasing X1 by 1, we will have 1 ≤ X1 ≤ 100, and it is this fact which keeps the array references in the third instruction in the path (and the second one as well, for that matter) within their prescribed bounds.

We shall refer to the above program as a globally verified pure procedure. This is the most general form of pure procedure with which we shall be concerned. A globally verified pure procedure may contain arbitrary indexed store and indexed transfer instructions, and these are kept within limits by the assertions given with the procedure. The idea of proving that a program does not modify itself by using the control paths in that program, when the fact that the program does not modify itself must be used in setting up these very same control paths, may seem like a circular argument; but it is not. Remember that both the instruction constancy assertion and the other assertions at the beginning of a path may be assumed to hold in proving the validity of both the instruction constancy assertion and the other assertions at the end of that path.

9-3 Self-Modification

Even when a machine language program does modify itself, an extension of the methods of the preceding section may be used to prove its correctness. Such methods are necessary, for example, when working with a computer having neither index registers nor indirect addressing; all array references in any program written for such a computer must be made by modifying instruction words. Also, when the subroutine call instruction of a computer stores a subroutine return instruction in memory -- as, for example, on the CDC 6600 -- this must be considered as a form of instruction modification.

Suppose that $a_1, \ldots, a_n$ are the instruction words of some program. If this program modifies itself, there may, in general, be several codes $w_{i1}, w_{i2}, \ldots$, for instructions $I_{i1}, I_{i2}, \ldots$, which may, in the course of the program, occupy the single instruction word a_i. Since the total number of instruction codes is finite, there will, in general, be a finite number k_i of these for each a_i. If $k_i = 1$, then a_i is never modified. In order to prove the correctness of a self-modifying program, we first determine the values of each k_i and all the w_{ij}; that is, we determine, for each instruction word, how many different instruction codes may be stored there, and what these codes are. Relative to this information, we may formulate an instruction behavior assertion (IBA) for the program; this is the assertion that the current value of each a_i, $1 \leq i \leq n$, is some w_{ij}, $1 \leq j \leq k_i$.

The instruction behavior assertion is the natural generalization of the instruction constancy assertion (ICA) of the pre-

ceding section; it reduces to that assertion if each $k_i = 1$. Like the constancy assertion, the instruction behavior assertion must be shown to be preserved along each control path of the program. In order to show this, of course, we will need other assertions that constrict the behavior of modified instructions more precisely.

How do we set up our control paths when some of our instructions may be modified? Suppose that some instruction word a_i can contain k_i different instruction codes. Then we must treat a_i as a k_i-way conditional instruction. If the codes are w_{i1}, ..., w_{ik_i}, and the corresponding instructions are I_{i1}, ..., I_{ik_i}, then the meaning of a_i within this program is "If $a_i = w_{ij}$, then perform the instruction I_{ij}." The instruction behavior assertion, which we are assuming to hold at each point, tells us, of course, that the contents of a_i must always be some w_{ij}. If each I_{ij} is a transfer instruction, then a_i becomes a k_i-way conditional transfer instruction, and there are k_i different directions for a control path to go. On the other hand, it may happen that none of the I_{ij} (for this particular value of i) is a transfer instruction; in that case, all control paths through the given instruction word proceed forward in the normal manner.

We illustrate these notions by writing the program of the preceding chapter for our original 16-bit computer of section 3-5, as follows:

Address (hexadecimal)	Contents (hexadecimal)	Mnemonics and assertions
100	----	A BSS 100
164	----	B BSS 100

```
1C8     ----    N   BSS  1
1C9     ----    J   BSS  1
1CA     1100    W   LD   A
1CB     7164    X   ST   B
                    * N > 0, N ≤ 100
1CC     11C8    S   LD   N
1CD     6001        SUI  1
1CE     71C9        ST   J
1CF     11CA        LD   W
1D0     71D3        ST   Y
1D1     11CB        LD   X
1D2     71D4        ST   Z
                    * (A(K) = B(K), K .IN. (1..N-J-1))
                    * 0 ≤ J, J < N, N ≤ 100,
                    * Y = (LD A(N-J)), Z = (ST B(N-J))
1D3     1100    Y   LD   A
1D4     7164    Z   ST   B
1D5     D1D3        IN   Y
1D6     D1D4        IN   Z
1D7     E1C9        DE   J
1D8     11C9        LD   J
1D9     B1D3        TP   Y
                    * (A(K) = B(K), K .IN. (1..N))
1DA     ----
```

There are two instructions in this program which may be modified, namely Y and Z. The instruction at Y may contain any of the 101 instruction codes for "load A(i)," $1 \leq i \leq 101$; the instruction at Z may contain any of the instruction codes for "store B(i),"

$1 \leq i \leq 101$. (The case $i = 101$ happens only the last time through the loop, as a result of incrementation, and therefore Y and Z are never actually executed under these conditions.) The instruction behavior assertion here, then, states that each of the sixteen instructions with addresses 1CA through 1D9 has contents as given by the above table, except for Y and Z (addresses 1D3 and 1D4), for which we have $1100 \leq Y \leq 1164$ and $1164 \leq Z \leq 11C8$ (hexadecimal).

We now give a complete proof of the partial correctness of this program. (Termination will follow immediately from considering the controlled expression J. Note that, since the original index I is no longer needed, due to the address modification, we have chosen to employ a decreasing index J for efficiency reasons.) The first control path is

```
* (IBA), N > 0, N ≤ 100
        LD   N          ac = N
        SUI  1          ac = ac - 1
        ST   J          J = ac
        LD   W          ac = W
        ST   Y          Y = ac
        LD   X          ac = X
        ST   Z          Z = ac
* (IBA), (A(K) = B(K), K .IN. (1..N-J-1)), 0 ≤ J, J < N,
* N ≤ 100, Y = (LD A(N-J)), Z = (ST B(N-J))
```

We must, of course, explicitly introduce (IBA) at the beginning and the end of every path such as this, just as we did before with (ICA).

There are three store instructions in this path. The first stores J, which is not an instruction, and therefore this does not

affect IBA. The other two store instructions initialize Y and Z to the initial contents of W and X respectively, and these are given by IBA as (LD A) and (ST B) respectively. From this we may conclude that IBA continues to hold at the end of the path. The assertion about A(K) and B(K) is vacuously true, since J is set equal to N-1 in the path and thus N-J-1 = 0. The other assertions at the end of the path are verified as follows: $0 \leq J$, since $N > 0$ and $J = N-1$; $J < N$, since $J = N-1$; $N \leq 100$, since N is unchanged; Y = (LD A) = (LD A(1)) = (LD A(N-J)), since J = N-1; Z = (ST B) = (ST B(1)) = (ST B(N-J)), also since J = N-1.

The second control path is

* (IBA), (A(K) = B(K), K .IN. (1..N-J-1)), $0 \leq J$, $J < N$,
* $N \leq 100$, Y = (LD A(N-J)), Z = (ST B(N-J))

Perform Y		ac = A(N-J)
Perform Z		B(N-J) = ac
IN	Y	Y = Y + 1
IN	Z	Z = Z + 1
DE	J	J = J - 1
LD	J	ac = J
(ac $\geq$ 0)		

* (IBA), (A(K) = B(K), K .IN. (1..N-J-1)), $0 \leq J$, $J < N$,
* $N \leq 100$, Y = (LD A(N-J)), Z = (ST B(N-J))

There are two instructions here which modify other instructions, namely (IN Y) and (IN Z). (The instructions Y and Z themselves, by IBA, cannot modify other instructions.) Since $0 \leq J < N$ at the beginning of the path, we have (LD A(1)) $\leq$ Y $\leq$ (LD A(N)) at that time, and similarly for Z, so that (IBA) holds even after modification. The first thing this control path does is to set B(N-J) =

A(N-J); we may infer this from that part of the initial assertion which deals with the contents of Y and Z. This, combined with (A(K) = B(K), K .IN. (1..N-J-1)), gives (A(K) = B(K), K .IN. (1..N-J)); later, J is decreased by 1, giving (A(K) = B(K), K .IN. (1..N-J-1)) again. Similarly, (Y = (LD A(N-J))) becomes (Y = (LD A(N-J+1))) when Y is incremented, and (Y = (LD A(N-J))) again when J is decremented; and similarly for Z. The other assertions at the end of the path follow much as before: $0 \leq J$ (from ac = J and ac $\geq$ 0 near the end of the path); $J < N$ (from $J < N$ at the start, since J is decreased and N is unchanged); and $N \leq 100$ (since N is unchanged).

Finally, the third and last control path is

```
* (IBA), (A(K) = B(K), K .IN. (1..N-J-1)), 0 ≤ J, J < N,
* N ≤ 100, Y = (LD A(N-J)), Z = (ST B(N-J))
      Perform Y          ac = A(N-J)
      Perform Z          B(N-J) = ac
      IN   Y             Y = Y + 1
      IN   Z             Z = Z + 1
      DE   J             J = J + 1
      LD   J             ac = J
      (ac < 0)
* (IBA), (A(K) = B(K), K .IN. (1..N))
```

We note first that we had $0 \leq J$ at the beginning of the path, and then J was decreased by one and the result was negative. Therefore we must have had J = 0 at the beginning of the path, and hence (A(K) = B(K), K .IN. (1..N-1)), Y = (LD A(N)), and Z = (ST B(N)). The preservation of (IBA) now follows just as it did in the second path, although, as we have already noted, the final

values of Y and Z must be specified with care when (IBA) is constructed. Also, just as before, by setting B(N) = A(N) when (A(K) = B(K), K .IN. (1..N-1)) is true, we obtain the final assertion (A(K) = B(K), K .IN. (1..N)). This completes the proof.

We may note that our program has been globally verified, in the terminology of the preceding chapter. Is there an analogue, for self-modifying programs, of the absolutely pure procedure -- a sort of "absolutely not-quite-pure procedure"? More specifically, if we specify an instruction behavior assertion with particular choices of the instruction codes w_{ij} and corresponding instructions I_{ij}, if we note that none of these are indexed store instructions and that no store instruction can change the contents of a cell which is not to be modified, can we infer that the given instruction behavior assertion is always preserved? In general, the answer is no. The trouble is that we have no way of knowing what might be stored in the modified instructions. In particular, we might store something there that would proceed to modify some other instruction in an uncontrolled manner the next time it was executed. Even if a store instruction is always immediately preceded by a load instruction, so that we presumably know what is in the accumulator or other such register at the given time, we have no general way of knowing that some other instruction cannot transfer around that load instruction. If instruction modifications are of certain specified types, a local verification algorithm may be set up; this is, in particular, true when considering subroutine call instructions which store a subroutine return instruction in memory. This subject will be taken up again in section 20- .

9-4 Integer Arithmetic

The special problems of integer arithmetic on computers include size specifications, overflow (including negative overflow and overflow as a result of shifting), division by zero, and the use of special properties of arithmetic instructions (such as integer subtraction of floating-point numbers as a test for equality). Properties of programs which involve any of these phenomena may be proved by further application of our notion of a restricted command.

Every computer has some smallest negative value and some largest positive value for the one-word integers which it can store. We shall refer to these bounds, in the remainder of this section, as *min* and *max* respectively, with the understanding that they will be different on different computers. If negative numbers are represented in two's complement, we normally have $\underline{min} = -2^{b-1}$, where b is the number of bits per word. If negative numbers are represented in one's complement or by signed magnitude, we have $\underline{min} = -2^{b-1}+1$; finally, in all cases, we have $\underline{max} = 2^{b-1}-1$.

When any final result or intermediate quantity in a calculation is an integer which is larger than *max*, we say that *overflow* has occurred; for an integer which is smaller than *min*, we say that *negative overflow* (or "underflow") has occurred. In most programs, we shall be interested in proving that overflow never occurs, provided that the data which the program processes is within certain limits. On the other hand, there are some programs -- such as double precision addition and subtraction routines -- which make use of the specific results calculated by the add and subtract instructions of a computer when overflow does occur.

Before taking up the question of overflow, however, we must consider the even more basic question of size specifications on the constants and data of a program that are determined by min and max. These affect all programs which process integer data, even if overflow never occurs. As we shall see, our GCD program of section 6-1 never produces overflow or underflow in any situation; but this does not mean that this program will correctly calculate the GCD of any two positive integers. The problem is that an integer larger than max or smaller than min can never be stored in M or N in the first place. In general, if I is any integer variable in a program, the assertions max $\geq$ I and I $\geq$ min must hold at all points within the program. They must, in particular, be implied by the initial assertions of the program. In the GCD program, we have the initial assertions M > 0 and N > 0, which imply M $\geq$ min and N $\geq$ min; but max $\geq$ M and max $\geq$ N must be explicitly or implicitly specified as initial assertions also.

Sometimes, in the presence of finite-precision arithmetic, stronger assertions than the above must be given. Thus in the routine of section 7-1 to calculate the 23rd power of a number A, if this number is taken as an integer, we must initially have A1 $\geq$ A and A $\geq$ A2, where $A1^{23}$ = max and $A2^{23}$ = min. In the GCD program, however, because the variables I and J continue to decrease, the program will actually run for all M and N which are between 1 and max, inclusive.

Let us now see what we can prove about overflow. The addition, subtraction, and multiplication operations on actual computers are not defined so as always to coincide with the mathematical operations of integer addition, subtraction, and multiplication. Thus when we write N = I + J in an algebraic language, or

when we write its equivalent in machine language instructions, we are not actually -- always -- adding I and J; we are performing an operation which sometimes is identical to ordinary addition, but sometimes (when overflow or underflow occurs) is not. What result is actually calculated depends upon the computer being used. The same is true if the plus sign were replaced by a subtraction or multiplication sign. (Integer division, with a single-precision dividend, cannot give rise to overflow or underflow.)

The above facts may seem to imply that, when we want to prove the correctness of a program on an actual machine, we must replace each addition operation by the use of a function giving the result of machine addition on that machine (and the same for subtraction and multiplication) and prove the correctness of the resulting program. Thus, for example, every use of the statement N = I + J would have to be replaced by N = ADD(I, J), where ADD(I, J) is defined as I + J if $\underline{max} \geq I + J$ and $I + J \geq \underline{min}$, and as something else (dependent upon the particular machine) otherwise. Such a requirement would significantly increase the cumbersomeness of proofs of most programs as run on actual machines. Our first application of restricted commands serves to make such a requirement unnecessary for the vast majority of programs, in which it is to be assumed (and proved) that overflow will never occur. Any addition operation is viewed as a restricted command, which works properly only if both of its operands and the result are in the range defined by min and max, and the same is true for subtraction and multiplication. Thus N = I + J, for example, is a restricted command with the restrictions $\underline{max} \geq I$, $I \geq \underline{min}$, $\underline{max} \geq J$, $J \geq \underline{min}$, $\underline{max} \geq N$, and $N \geq \underline{min}$. As long as I, J, and N are in these ranges, the actual addition function of the given machine will coincide

with ordinary addition, and thus we may assume that the restricted command N = I + J always performs ordinary addition.

When an integer expression is evaluated by means of such restricted commands, the assertion before the start of evaluation must imply that all intermediate results, as well as the final result, are within the bounds determined by min and max. The following program, written in the assembly language of section 3-5, executes the FORTRAN assignment Z = A + B - C:

```
LD  A
AD  B
SU  C
ST  Z
```

As a control path (with initial and final assertions unspecified for the moment), this reads:

```
C  INITIAL ASSERTION
     [max ≥ A, A ≥ min]
     ac = A
     [max ≥ B, B ≥ min, max ≥ ac, ac ≥ min, max ≥ ac+B, ac+B ≥ min]
     ac = ac + B
     [max ≥ C, C ≥ min, max ≥ ac, ac ≥ min, max ≥ ac-C, ac-C ≥ min]
     ac = ac - C
     [max ≥ Z, Z ≥ min]
     Z = ac
C  FINAL ASSERTION
```

where the restrictions, as before, are contained in brackets, and where ac denotes the accumulator. We may now see that the initial assertion

$$\underline{max} \geq A,\ A \geq \underline{min},\ \underline{max} \geq B,\ B \geq \underline{min},$$
$$\underline{max} \geq C,\ C \geq \underline{min},\ \underline{max} \geq A+B-C,\ A+B-C \geq \underline{min}$$

-- that is, the assertion that all variables in the given expression, as well as the expression itself, are within range -- is not sufficient to make this control path valid. (Consider A > 0 and B = C = max, for example.) We must also require that the intermediate result A+B is within range. When the statement Z = A + B - C is executed in an algebraic language, where Z, A, B, and C are integers, and the execution is presumed to take place on an actual computer (rather than an abstract machine whose cells can hold numbers of arbitrary size), similar restrictions must be given. (It is conceivable that a language might be defined in which these restrictions are relaxed -- by specifying, for example, that all intermediate computations take place in double precision, with the final result being re-converted to single precision.)

Now suppose we have a program in which use is made of the properties of the computer's addition and subtraction functions in the presence of overflow. In such a case restricted commands cannot be used; we must use the functions defining the actual arithmetic operations of the given machine. The following double-precision addition routine illustrates what may be done in such a case. It is assumed that the double-precision quantity A is stored as an array of size 2, with the high-order part in A(1) and the low-order part in A(2), and similarly for B and C. The routine treats A and B as unsigned integers, and similarly A(1), A(2), B(1) and B(2) as unsigned quantities (nonnegative and less than 2^b), so that we may regard A as $2^b*A(1)+A(2)$, and similarly B as $2^b*B(1)+B(2)$ and C as $2^b*C(1)+C(2)$. The routine sets C equal to A+B:

* $0 \le A(1),\ A(1) < 2^b,\ 0 \le A(2),\ A(2) < 2^b,$

* $0 \le B(1),\ B(1) < 2^b,\ 0 \le B(2),\ B(2) < 2^b,$

* $(2^b*A(1)+A(2)) + (2^b*B(1)+B(2)) < 2^{2b}$

```
          LD    A(2)
          AD    B(2)
          TC    ALPHA
          ST    C(2)
          LDI   0
          TR    BETA
ALPHA     ST    C(2)
          LDI   1
BETA      AD    A(1)
          AD    B(1)
          ST    C(1)
```

* $0 \le C(1),\ C(1) < 2^b,\ 0 \le C(2),\ C(2) < 2^b,$

* $(2^b*C(1)+C(2)) = (2^b*A(1)+A(2)) + (2^b*B(1)+B(2))$

The initial assertions are that A(1), A(2), B(1), and B(2) are in range, and that the resulting value of C will fit into a double word. The instruction TC Z transfers control to Z if carry is 1; carry is set by the AD instruction, and specifically AD Y sets carry = if ac + Y < 2^b then 0 else 1, followed by ac = if ac + Y < 2^b then ac + Y else ac + Y - 2^b. This program has two control paths; we consider only the first (the one which passes through the statement labeled ALPHA):

* $0 \le A(1),\ A(1) < 2^b,\ 0 \le A(2),\ A(2) < 2^b,$

* $0 \le B(1),\ B(1) < 2^b,\ 0 \le B(2),\ B(2) < 2^b,$

* $(2^b*A(1)+A(2)) + (2^b*B(1)+B(2)) < 2^{2b}$

ac = A(2)

carry = if ac + B(2) < 2^b then 0 else 1

ac = if ac + B(2) < 2^b then

    ac + B(2) else ac + B(2) - 2^b

(carry = 1)

C(2) = ac

ac = 1

carry = if ac + A(1) < 2^b then 0 else 1

ac = if ac + A(1) < 2^b then

    ac + A(1) else ac + A(1) - 2^b

carry = if ac + B(1) < 2^b then 0 else 1

ac = if ac + B(1) < 2^b then

    ac + B(1) else ac + B(1) - 2^b

C(1) = ac

* $0 \leq C(1)$, $C(1) < 2^b$, $0 \leq C(2)$, $C(2) < 2^b$,
* $(2^b*C(1)+C(2)) = (2^b*A(1)+A(2)) + (2^b*B(1)+B(2))$

The condition carry = 1 in the path implies that we must have had $A(2)+B(2) \geq 2^b$ at the start of this path; carry is initially set to 1 and ac to ac + B(2) - 2^b. Furthermore, the initial assertion $(2^b*A(1)+A(2)) + (2^b*B(1)+B(2)) < 2^{2b}$ implies, as a few calculations will show, that each of the other two instructions sets carry to 0. Thus the above path reduces to

* $0 \leq A(1)$, $A(1) < 2^b$, $0 \leq A(2)$, $A(2) < 2^b$,
* $0 \leq B(1)$, $B(1) < 2^b$, $0 \leq B(2)$, $B(2) < 2^b$,
* $(2^b*A(1)+A(2)) + (2^b*B(1)+B(2)) < 2^{2b}$,
* $A(2)+B(2) \geq 2^b$

ac = A(2)

carry = 1

ac = ac + B(2) - 2^b

(carry = 1)

C(2) = ac

ac = 1

carry = 0

ac = ac + A(1)

carry = 0

ac = ac + B(1)

C(1) = ac

* $0 \le C(1), C(1) < 2^b, 0 \le C(2), C(2) < 2^b,$

* $(2^b * C(1) + C(2)) = (2^b * A(1) + A(2)) + (2^b * B(1) + B(2))$

At the end of this path, we have $C(1) = 1+A(1)+B(1)$ and $C(2) = A(2)+B(2)-2^b$, implying that the last of the final assertions is valid; also, C(1) is in range because of what we have said about carry in the last two additions, while C(2) is in range because $A(2)+B(2) \ge 2^b$ implies $C(2) \ge 0$ and $A(2) < 2^b$ and $B(2) < 2^b$ imply $C(2) < 2^b$. Thus the given path is valid.

Other integer arithmetic instructions may also be viewed as restricted commands in this way. For a divide instruction, the restriction says that the divisor is unequal to zero. If our program uses the special result (normally zero) to which the given register is set when the divisor is actually zero, then "divide by Y" must be interpreted as ac = if Y=0 then 0 else ac/Y. Similarly, for a shift instruction which is being used as a multiplication or division by a power of 2, the restriction says that overflow cannot occur in this process, and also, quite often, that the initial contents of the given register were nonnegative. If possibly negative quantities are being shifted, the arithmetic interpretation of the shift instruction must be given with care; it will depend on whether one's complement or two's complement is being used, and whether the shift is circular, sign-extending, etc.

9-5 Floating Point Arithmetic

From the viewpoint of correctness, floating point arithmetic differs from integer arithmetic in two fundamental respects. The first is that real numbers are, in general, represented only approximately by floating point numbers, whereas integers are represented exactly. The second is that, even if two real numbers are represented exactly, their sum, difference, product, or quotient might not be; and if they are represented inaccurately, it is possible for such arithmetic results to be represented even more inaccurately. Worse than this, these two types of error can be arbitrarily large; cases may be constructed, on typical digital computers, for which they are as large as 2^{100}.

Given these dismal facts, are we completely at a loss to prove anything about floating point calculations? Perhaps surprisingly, the answer is no. Many programs which perform floating point arithmetic may be proved correct almost as easily as those which do not. Several specialized techniques, however, are necessary in proving correctness in this case; and there are a great number of simple floating point calculations which, although they are correct, are quite difficult to prove correct.

The first thing we have to realize about floating point addition, subtraction, multiplication, and division is that <u>it is only subtraction</u> which causes unbounded increase in floating point inaccuracy. (Strictly speaking, it is only addition of quantities of opposite signs, and subtraction of quantities of the same sign.) So long as we restrict ourselves to addition of positive numbers, and arbitrary multiplication and division, the total error in a calculation is roughly proportional to the number of operations

performed. This number may be determined by our standard assertion method in many cases. Let us now make the above statements precise.

In the remainder of this section, we shall assume that a floating point number consists of a 1-bit sign, an s-bit exponent, and a t-bit fraction. Thus the total length of the word is $1+s+t$. The bias, which shall be called β, is an exponent field quantity consisting of 1 followed by all 0's, or, in other words, 2^{s-1}. We shall denote by PFW(E, F) the positive floating point word with exponent field E and fraction field F; its value is $F \cdot 2^{E-\beta}$. Here E is assumed to be an integer and F a fraction with binary point at the left. It is assumed that $2^{-t} \leq F < 1$; if $1/2 \leq F < 1$, then PFW(E, F) is said to be normalized. The floating point sum, product, difference, and quotient of U and V will be denoted by FLA(U, V), FLS(U, V), FLM(U, V), and FLD(U, V) respectively; if U = PFW(E, F) and V = PFW(E', F') are both normalized and $U \geq V$ (that is, $F \cdot 2^{E-\beta} \geq F' \cdot 2^{E'-\beta}$) then

FLA(U, V) = FLA(V, U) = if W(t) ≥ 1 then PFW(E+1, W(t-1)) else PFW(E, W(t)), where $W(A) = [(F + F'/2^{E-E'}) \cdot 2^A + 1/2]/2^t$

FLS(U, V) = -FLS(V, U) = if $2^{-G-1} \leq W \leq 2^{-G}$ then PFW(E-G, $W \cdot 2^G$), $0 \leq G \leq t$, where $W = [(F - F'/2^{E-E'}) \cdot 2^t + 1/2]/2^t$

while (regardless of whether $U \geq V$ or $U < V$) we have

FLM(U, V) = if W(t+1) < 1 then PFW(E+E'-β-1, W(t+1)) else PFW(E+E'-β, W(t)), where $W(A) = [F \cdot F' \cdot 2^A + 1/2]/2^t$

FLD(U, V) = if W(t) ≥ 1 then PFW(E-E'+β+1, W(t-1)) else PFW(E-E'+β, W(t)), where $W(A) = [2^A \cdot F/F' + 1/2]/2^t$

and FLA(U, V), FLS(U, V), FLM(U, V), and FLD(U, V) are all normalized. (Here $[\alpha]$ is the greatest integer in α; $\alpha - 1 < [\alpha] \leq \alpha$.)

We must note at this point that these conventions are by no means universal for typical digital computers. On the CDC 6000 series, for example, for which (using the notation above) $\underline{s} = 11$ and $\underline{t} = 48$, the "fraction" is not a fraction at all, but an integer, and the exponent is adjusted accordingly. In this case PFW(E, F) is called normalized if $2^{t-1} \leq F < 2^t$, and it is always assumed that $1 \leq F < 2^t$ (for a <u>positive</u> floating point word). Thus 1, for example, is represented in normalized form by PFW(β-47, 2^{47}), whereas, using our conventions, it would be PFW(β+1, 1/2). On the IBM 360 and 370, where $\underline{s}$ is 7 and $\underline{t}$ is either 24 ("short," single precision) or 56 ("long," double precision), the exponent is a power of 16, rather than of 2, and the value of PFW(E, F) is $F \cdot 16^{E-\beta}$. It is to be understood, however, that, although the details of the statements and proofs of the theorems given below will be different for different computers, the basic ideas remain the same.

Every floating point number may be represented with a <u>relative inaccuracy</u> of at most 2^{-t}; that is, if R is a real number and FPA(R) is its best floating point approximation, we have

$$R(1 - 2^{-t}) \leq FPA(R) \leq R(1 + 2^{-t})$$

provided that R is positive and in the floating point range, that is, provided that

$$PFW(0, 1/2) \leq R \leq PFW(2\beta-1, 1-2^{-t})$$

or, in terms of values, $2^{-\beta-1} \leq R \leq 2^{\beta-1} \cdot (1-2^{-t})$. (We shall refer to $2^{-\beta-1}$ as <u>fmin</u> and $2^{\beta-1} \cdot (1-2^{-t})$ as <u>fmax</u>, so that <u>fmin</u> $\leq R \leq$ <u>fmax</u>. A few unnormalized floating point numbers less than <u>fmin</u> exist; these will be ignored. If R is negative, $R(1-2^{-t}) \geq FPA(R) \geq R(1+2^{-t})$.) In fact, let $\gamma = [\log_2 R]$, so that $2^{\gamma} \leq R <$

$2^{\gamma+1}$. Then $-\beta-1 \le \gamma < \beta-1$, and $1/2 \le S \cdot 2^{-\gamma-1} < 1$. Let $N = R \cdot 2^{-\gamma-1}$, and let $N' = [N \cdot 2^t + 1/2]$, so that $2^{t-1} \le N' \le 2^t$. Then $N \cdot 2^t - 1/2 < N' \le N \cdot 2^t + 1/2$, and $1/2 \le N'/2^t \le 1$. We set FPA(R) = *if* $N'/2^t = 1$ *then* PFW($\beta+\alpha+2$, 1/2) *else* PFW($\beta+\kappa+1$, $N'/2^t$). Note that both 1/2 and $N'/2^t$ are in the proper form to be the fraction part of a normalized floating point number. Also, $0 \le \beta+\gamma+1 < 2\beta$, and we cannot have $\beta+\gamma+2 = 2\beta$ and $N'/2^t = 1$ (because then $R = N \cdot 2^{\beta-1}$ would be greater than fmax), showing that the exponent of FPA(R) is in its proper range. Finally, since $N \ge 1/2$, we have $N \cdot 2^t(1 - 2^{-t}) = N \cdot 2^t - N \le N \cdot 2^t - 1/2 < N' \le N \cdot 2^t + 1/2 \le N \cdot 2^t + N = N \cdot 2^t(1 + 2^{-\gamma})$, so that $N(1 - 2^{-t}) < N'/2^t \le N(1 + 2^{-t})$ and $R(1 - 2^{-t}) = N \cdot 2^{\gamma+1}(1 - 2^{-t}) < FPA(R) \le N \cdot 2^{\gamma+1}(1 + 2^{-t}) = R(1 + 2^{-t})$, since the value of FPA(R) is in all cases $2^{\beta+\gamma+1-\beta} \cdot N'/2^t = 2^{\gamma+1} \cdot N'/2^t$. This completes the proof of the above statement.

The floating point sum of two numbers, provided that it is properly normalized and rounded (as it is in the definition of FLA(U, V) above), should be the best floating point approximation of the actual sum, provided that the actual sum is in range. Thus we should expect, from the equation above, that

$$(U + V)(1 - 2^{-t}) \le FLA(U, V) \le (U + V)(1 + 2^{-t})$$

Let us prove this formula for U = PFW(E, F), V = PFW(E', F'), and fmin $\le$ U+V $\le$ fmax. We may assume $U \ge V$; the value of U+V is $2^{E-\beta} \cdot F + 2^{E'-\beta} \cdot F' = 2^{E-\beta} \cdot F + (2^{E-\beta}/2^{E-E'}) \cdot F' = 2^{E-\beta}(F + F'/2^{E-E'})$. For convenience, we set $G = F + F'/2^{E-E'}$, so that $2^{E-\beta} \cdot G$ is the value of U+V, and W(A) (in the definition of FLA(U, V)) is $[G \cdot 2^A + 1/2]/2^t$. Assume first that $W(t) \ge 1$; then we must have $G \ge 1 - 2^{-t-1}$, since otherwise $G \cdot 2^t + 1/2 < 2^t$ and hence $W(t) < 1$. Since, in general, $\alpha-1 < [\alpha] \le \alpha$, we have $G \cdot 2^{t-1} - 1/2 < [G \cdot 2^{t-1} + 1/2]$

$\leq G\cdot 2^{t-1} + 1/2$, so that $G/2 - 2^{-t-1} < W(\underline{t}-1) \leq G/2 + 2^{-t-1}$. There are now two subcases. If $1 - 2^{-t-1} \leq G \leq 1$, then $2^{t-1} + 1/4 \leq G\cdot 2^{t-1} + 1/2 \leq 2^{t-1} + 1/2$; since 2^{t-1} is an integer, we have $[G\cdot 2^{t-1} + 1/2] = 2^{t-1}$, so that $W(\underline{t}-1) = 1/2$ and $G/2(1 - 2^{-\underline{t}}) \leq 1/2(1 - 2^{-t}) < W(\underline{t}-1) < 1/2(1 + 2^{-t-1} - 2^{-2t-1}) = 1/2(1 - 2^{-t-1})(1 + 2^{-t}) \leq G/2(1 + 2^{-t})$. If, on the other hand, $G > 1$, then $G/2(1 - 2^{-t}) = G/2 - G\cdot 2^{-t-1} < G/2 - 2^{-\underline{t}-1} < W(\underline{t}-1) \leq G/2 + 2^{-t-1} < G/2 + G\cdot 2^{-t-1} = G/2(1 + 2^{-t})$. In either case, $2^{E-\beta}\cdot G\cdot(1 - 2^{-t}) < 2^{E+1-\beta}\cdot W(\underline{t}-1) \leq 2^{E-\beta}\cdot G\cdot(1 + 2^{-t})$, that is, $(U + V)(1 - 2^{-\underline{t}}) < FLA(U, V) \leq (U + V)(1 + 2^{-\underline{t}})$. Now suppose that $W(\underline{t}) < 1$. Again from $\alpha-1 < [\alpha] \leq \alpha$, we have $G\cdot 2^{\underline{t}} - 1/2 < [G\cdot 2^{\underline{t}} + 1/2] \leq G\cdot 2^{\underline{t}} + 1/2$, so that $G - 2^{-\underline{t}-1} < W(\underline{t}) \leq G + 2^{-\underline{t}-1}$. Since $F \geq 1/2$ and $F' \geq 1/2$, we have $G \geq 1/2$; thus $G\cdot(1 - 2^{-t}) = G - G\cdot 2^{-\underline{t}} \leq G - 2^{-t-1} < W(\underline{t}) \leq G + 2^{-\underline{t}-1} \leq G + G\cdot 2^{-t} = G\cdot(1 + 2^{-\underline{t}})$, so that $2^{E-\beta}\cdot G\cdot(1 - 2^{-\underline{t}}) < 2^{E-\beta}\cdot W(\underline{t}) \leq 2^{E-\beta}\cdot G\cdot(1 + 2^{-t})$, that is, just as before, $(U + V)(1 - 2^{-\underline{t}}) < FLA(U, V) \leq (U + V)(1 + 2^{-\underline{t}})$. This completes the proof.

Let us now prove similar formulas for multiplication and division. If $U = PFW(E, F)$ and $U' = PFW(E', F')$ are normalized, then

$$U\cdot V\cdot(1 - 2^{-t}) \leq FLM(U, V) \leq U\cdot V\cdot(1 + 2^{-\underline{t}})$$

if $\underline{fmin} \leq U\cdot V \leq \underline{fmax}$, and similarly

$$(U/V)\cdot(1 - 2^{-t}) \leq FLD(U, V) \leq (U/V)\cdot(1 + 2^{-\underline{t}})$$

if $\underline{fmin} \leq U/V \leq \underline{fmax}$. As before, these formulas merely reaffirm that floating point multiplication and division provide the best possible approximation to actual multiplication and division.

For multiplication, suppose first that $W(\underline{t}+1) < 1$. Since $\alpha-1 < [\alpha] \leq \alpha$, we have $F\cdot F'\cdot 2^{t+1} - 1/2 < [F\cdot F'\cdot 2^{t+1} + 1/2] \leq$

$F \cdot F' \cdot 2^{t+1} + 1/2$, so that $2 \cdot F \cdot F' - 2^{-t-1} < W(t+1) \leq 2 \cdot F \cdot F' + 2^{-t-1}$. Since $F \geq 1/2$ and $F' \geq 1/2$, we have $2 \cdot F \cdot F' \cdot (1 - 2^{-t}) = 2 \cdot F \cdot F' - 2 \cdot F \cdot F' \cdot 2^{-t} \leq 2 \cdot F \cdot F' - 2^{-t-1} < W(t+1) \leq 2 \cdot F \cdot F' + 2^{-t-1} \leq 2 \cdot F \cdot F' + 2 \cdot F \cdot F' \cdot 2^{-t} = 2 \cdot F \cdot F' (1 + 2^{-t})$, so that $2^{E-\beta} \cdot F \cdot 2^{E'-\beta} \cdot F' \cdot (1 - 2^{-t}) < 2^{E+E'-\beta-1-\beta} \cdot W(t+1) \leq 2^{E-\beta} \cdot F \cdot 2^{E'-\beta} \cdot F' \cdot (1 + 2^{-t})$, that is, $U \cdot V \cdot (1 - 2^{-t}) < FLM(U, V) \leq U \cdot V \cdot (1 + 2^{-t})$. Now suppose that $W(t+1) \geq 1$; we must then have $F \cdot F' \geq 1/2 - 2^{-t-2}$, since otherwise $F \cdot F' \cdot 2^{t+1} + 1/2 < 2^t$ and thus $W(t+1) < 1$. There are now two subcases. If $1/2 - 2^{-t-2} \leq F \cdot F' \leq 1/2$, then $2^{t-1} + 1/4 \leq F \cdot F' \cdot 2^t + 1/2 \leq 2^{t-1} + 1/2$; since 2^{t-1} is an integer, we have $[F \cdot F' \cdot 2^t + 1/2] = 2^{t-1}$, so that $W(t) = 1/2$ and $F \cdot F' (1 - 2^{-t}) \leq 1/2 (1 - 2^{-t}) < W(t) < 1/2 + 2^{-t-2} - 2^{-2t-2} = (1/2 - 2^{-t-2})(1 + 2^{-t}) \leq F \cdot F' (1 + 2^{-t})$. If, on the other hand, $F \cdot F' > 1/2$, then $F \cdot F' (1 - 2^{-t}) = F \cdot F' - F \cdot F' \cdot 2^{-t} \leq F \cdot F' - 2^{-t-1} < W(t) \leq F \cdot F' + 2^{-t-1} \leq F \cdot F' + F \cdot F' \cdot 2^{-t} = F \cdot F' \cdot (1 + 2^{-t})$. In either case, $2^{E-\beta} \cdot F \cdot 2^{E'-\beta} \cdot F' \cdot (1 - 2^{-t}) < 2^{E+E'-\beta-\beta} \cdot W(t) \leq 2^{E-\beta} \cdot F \cdot 2^{E'-\beta} \cdot F' \cdot (1 + 2^{-t})$, that is, $U \cdot V \cdot (1 - 2^{-t}) < FLM(U, V) \leq U \cdot V \cdot (1 + 2^{-t})$, just as before.

For division, note that $W(t)$ (as defined in connection with $FLD(U, V)$) ≥ 1 if and only if $F/F' \geq 1$ (that is, $F \geq F'$). For clearly $W(t) \geq F/F'$; whereas if $F/F' < 1$, then $(2^t \cdot F)/(2^t \cdot F') < 1$, and, since $2^t \cdot F$ and $2^t \cdot F'$ are both integers, they must differ by at least 1, that is, $2^t \cdot F + 1 \leq 2^t \cdot F'$ or $2^t F/F' + 1/F' \leq 2^t$, so that $W(t) = [2^t \cdot F/F' + 1/2]/2^t < 1$. First suppose, then, that $F \geq F'$; since $\alpha - 1 < [\alpha] \leq \alpha$, we have $2^{t-1} \cdot F/F' - 1/2 < [2^{t-1} \cdot F/F' + 1/2] \leq 2^{t-1} \cdot F/F' + 1/2$, so that $1/2 \cdot F/F' - 2^{-t-1} < W(t-1) \leq 1/2 \cdot F/F' + 2^{-t-1}$. Since $F \geq F'$, we have $1/2 \cdot F/F' \cdot (1 - 2^{-t}) = 1/2 \cdot F/F' - 1/2 \cdot F/F' \cdot 2^{-t} \leq 1/2 \cdot F/F' - 2^{-t-1} < W(t-1) \leq 1/2 \cdot F/F' + 2^{-t-1} \leq 1/2 \cdot F/F' + 1/2 \cdot F/F' \cdot 2^{-t} = 1/2 \cdot F/F' \cdot (1 + 2^{-t})$, so that $((2^{E-\beta} \cdot F)/(2^{E'-\beta} \cdot F')) \cdot (1 - 2^{-t}) < 2^{E-E'+\beta+1-\beta} \cdot W(t-1) \leq ((2^{E-\beta} \cdot F)/(2^{E'-\beta} \cdot F')) \cdot (1 + 2^{-t})$, that is, $(U/V) \cdot (1 - 2^{-t})$

$< FLD(U, V) \leq (U/V)\cdot(1 + 2^{-t})$. Now suppose that $F < F'$; again from $\alpha - 1 < [\alpha] \leq \alpha$, we have $2^t \cdot F/F' - 1/2 < [2^t \cdot F/F' + 1/2] \leq 2^t \cdot F/F' + 1/2$, so that $F/F' - 2^{-t-1} < W(\underline{t}) \leq F/F' + 2^{-t-1}$. Since $1/2 \leq F, F' < 1$, we have $2F > F'$; thus $F/F' \cdot (1 - 2^{-t}) = F/F' - F/F' \cdot 2^{-t} < F/F' - 2^{-t-1} < W(\underline{t}) \leq F/F' + 2^{-t-1} < F/F' + F/F' \cdot 2^{-t} = F/F' \cdot (1 + 2^{-t})$, so that $((2^{E-\beta} \cdot F)/(2^{E'-\beta} \cdot F'))\cdot(1 - 2^{-t}) < 2^{E-E'+\beta-\beta} \cdot W(\underline{t}) \leq ((2^{E-\beta} \cdot F)/(2^{E'-\beta} \cdot F'))\cdot(1 + 2^{-t})$, that is, $(U/V)\cdot(1 - 2^{-t}) < FLD(U, V) \leq (U/V)\cdot(1 + 2^{-t})$ as before. Thus both of the above statements are proved.

We are now ready for the fundamental definition of this section.

DEFINITION 9-1. Let $N > 0$. The assertion

$$\underline{\text{if}}\ R \geq 0\ \underline{\text{then}}\ R\cdot(1 - 2^{-t})^N \leq R' \leq R\cdot(1 - 2^{-t})^{-N}$$
$$\underline{\text{else}}\ R\cdot(1 - 2^{-t})^{-N} \leq R' \leq R\cdot(1 - 2^{-t})^N$$

will be denoted by FEQ(R, R', N) (read "R floating-equals R' with tolerance N units").

It should be clear that, if $R \geq 0$ and N is not too large ($N \leq 2^{t/2}$, say) that FEQ(R, R', N) implies $R\cdot(1 - (N+1)\cdot 2^{-t}) \leq R' \leq R\cdot(1 + (N+1)\cdot 2^{-t})$; if the tolerance is N units, the relative inaccuracy is at most $(N+1)\cdot 2^{-t}$. Other properties of FEQ(R, R', N) are:

(1) If FEQ(R, R', N), then FEQ(R', R, N).

(2) If FEQ(R, R', N), then R and R' are either both positive, both zero, or both negative.

(3) If FEQ(R, R', N), then FEQ(-R, -R', N).

(4) If FEQ(R, R', N), then FEQ(1/R, 1/R', N).

(5) If FEQ(X, Y, N) and FEQ(Y, Z, M), then FEQ(X, Z, N+M). In fact, X, Y, and Z are either all positive, all zero, or all

negative; if they are all positive, we have $X \cdot (1 - 2^{-t})^{N+M} = X \cdot (1 - 2^{-t})^{N} \cdot X \cdot (1 - 2^{-t})^{M} \leq Y \cdot (1 - 2^{t})^{M} \leq Z \leq Y \cdot (1 - 2^{-t})^{-M} \leq X \cdot (1 - 2^{-t})^{-N} \cdot X \cdot (1 - 2^{-t})^{-M} = X \cdot (1 - 2^{-t})^{-(N+M)}$. (The proof in the other two cases is similar.)

(6) If U and V are both positive or both negative, then FEQ(U+V, FLA(U, V), 1). In fact, $1 + 2^{-t} < (1 - 2^{-t})^{-1}$ (since $(1 + 2^{-t})(1 - 2^{-t}) = 1 - 2^{-2t} < 1$), so that, by what we have already proved, $(U+V)(1 - 2^{-t}) \leq FLA(U, V) \leq (U+V)(1 + 2^{-t}) \leq (U+V)(1 - 2^{-t})^{-1}$. If U and V are both negative, the proof is similar.

(7) FEQ(U•V, FLM(U, V), 1). (Same argument as above.)

(8) FEQ(U/V, FLD(U, V), 1). (Same argument as above.)

(9) If FEQ(U, U', N) and FEQ(V, V', M), and U and V are either both positive or both negative, then FEQ(U+V, U'+V', max(N, M)). In fact, if U and V (and thus U' and V') are both positive, and if $U \cdot (1 - 2^{-t})^{N} \leq U' \leq U(1 - 2^{-t})^{-N}$, $V \cdot (1 - 2^{-t})^{M} \leq V' \leq V \cdot (1 - 2^{-t})^{-M}$, then, letting P = max(M, N), we have $(U+V)(1 - 2^{-t})^{P} = U \cdot (1 - 2^{-t})^{P} + V \cdot (1 - 2^{-t})^{P} \leq U \cdot (1 - 2^{-t})^{N} + V \cdot (1 - 2^{-t})^{M} \leq U'+V' \leq U \cdot (1 - 2^{-t})^{-N} + V \cdot (1 - 2^{-t})^{-M} \leq U \cdot (1 - 2^{-t})^{-P} + V \cdot (1 - 2^{-t})^{-P} = (U+V)(1 - 2^{-t})^{-P}$. If U and V are both negative, the proof is similar.

(10) If FEQ(U, U', N) and FEQ(V, V', M), then FEQ(U•V, U'•V', N+M). In fact, if U and V are both positive and $U \cdot (1 - 2^{-t})^{N} \leq U' \leq U \cdot (1 - 2^{-t})^{-N}$, $V \cdot (1 - 2^{-t})^{M} \leq V' \leq V \cdot (1 - 2^{-t})^{-M}$, then $U \cdot V \cdot (1 - 2^{-t})^{N+M} = U \cdot (1 - 2^{-t})^{N} \cdot V \cdot (1 - 2^{-t})^{M} \leq U' \cdot V' \leq U \cdot (1 - 2^{-t})^{-N} \cdot V \cdot (1 - 2^{-t})^{-M} = U \cdot V \cdot (1 - 2^{-t})^{-(N+M)}$. There are three other cases — U and V both negative, U positive and V negative, and U negative and V positive -- and the proofs are all similar.

(11) If FEQ(U, U', N) and FEQ(V, V', M), then FEQ(U/V, U'/V', N+M). (This follows directly from (4) and (10) above.)

(12) If U and V are both positive or both negative, and if FEQ(U, U', N) and FEQ(V, V', M), then FEQ(U+V, FLA(U', V'), max(N, M) + 1). (This follows directly from (9), (6), and (5) above.)

(13) If FEQ(U, U', 2N-1) and FEQ(V, V', 2M-1), then FEQ(U•V, FLM(U', V'), 2(N+M)-1). (This follows directly from (10), (7), and (5).)

(14) If FEQ(U, U', 2N-1) and FEQ(V, V', 2M-1), then FEQ(U/V, FLD(U', V'), 2(N+M)-1). (This follows directly from (11), (8), and (5).)

It is understood, of course, that all given quantities must be in range for each of these properties to hold; that is, $\underline{fmin} \leq |\alpha| \leq \underline{fmax}$ for each quantity α mentioned in connection with each property.

Roughly speaking, the result of an operation which requires N multiplications and divisions will be equal to what it is supposed to be, with tolerance 2N-1 units, by properties (13) and (14) above. The same (actually a little more) is true if additions of quantities of like sign, or subtractions of quantities of opposite sign, are allowed, by property (12) above. We shall now illustrate property (13) by investigating the partial correctness of an assembly-language program which performs exponentiation, using the algorithm which was programmed in FORTRAN at the beginning of section 7-2. In addition to FEQ, we use the assertion INRA(α) (that is, α is in range), meaning that $\underline{fmin} \leq |\alpha| \leq \underline{fmax}$, and the assertion NORM(α), meaning that α is in normalized form. The program is as follows:

```
* N ≥ 0, (N ≠ 0 or A ≠ 0), NORM(A), INRA(A^N)
        LD    ONE
        ST    X
        LDI   0
        ST    I
* FEQ(X, A^I, 2I-1), NORM(X), NORM(A), INRA(A^N), I ≤ N
LOOP    LD    I
        SU    N
        TZ    DONE
        LD    X
        FM    A
        ST    X
        IN    I
        TR    LOOP
* FEQ(X, A^N, 2N-1)
DONE
```

The cell ONE is presumed to contain the floating point number 1.0; the instruction mnemonic FM means "floating multiply." Note that, whereas in section 7-2 we had the intermediate assertion $X = A^I$, here it is only FEQ(X, A^I, 2I-1); that is, X is only approximately equal to A^I. The term 2I-1 is suggested by the term 2N-1 in property (13) above. In fact, whenever we pass the beginning of the loop, the value of I will be equal to the number of multiplications thus far performed, and property (13) then tells us that 2I-1 is the tolerance.

Let us look at the control path in this program which goes around the loop:

* FEQ(X, A^I, 2I+1), NORM(X), NORM(A), INRA(A^N), $I \leq N$

ac = I

ac = ac - N

(ac ≠ 0)

ac = X

ac = FLM(ac, X)

X = ac

I = I + 1

* FEQ(X, A^I, 2I+1), NORM(X), NORM(A), INRA(A^N), $I \leq N$

Since multiplication is in floating point, we write ac = FLM(ac, X), rather than ac = ac * X. The verification condition of this path is

(FEQ(X, A^I, 2I+1), NORM(X), NORM(A), INRA(A^N), $I \leq N$, I-N ≠ 0) implies (FEQ(FLM(X, A), A^{I+1}, 2(I+1)+1), NORM(FLM(X, A)), NORM(A), INRA(A^N), $I+1 \leq N$)

Since $I \leq N$ but I-N ≠ 0, we must have had $I < N$ at the beginning of the path, implying $I+1 \leq N$. As we have defined FLM(X, A), if X and A are normalized, so is FLM(X, A). The assertion FEQ(FLM(X, A), A^{I+1}, 2(I+1)+1) follows from property (13) given above, and also property (1), under the hypotheses of this path, provided that we can show INRA(A^{I+1}). It is assumed that fmin < 1 and fmax > 1; if $|A| > 1$, then certainly fmin < $|A^{I+1}|$, while $|A^{I+1}| \leq |A^N| \leq$ fmax since $I+1 \leq N$. If $|A| = 1$, then fmin < $|A^{I+1}| = 1 <$ fmax; while if $|A| < 1$, then fmin ≤ $|A^N| \leq |A^{I+1}|$, since $I+1 \leq N$, and certainly $|A^{I+1}| <$ fmax. Thus the verification condition of this path is valid.

NOTES

The first detailed treatment of machine language program correctness seems to be that of [Painter 67]. Painter is concerned with proving the correctness of a compiler whose source language is algebraic, and whose destination language is an assembly-like language. The machine for which this assembly language is designed, however, is assumed to be free from the difficulties discussed in this chapter, such as finite precision, overflow, and the possibility of self-modification.

The concept of a pure procedure is well known in the field of operating systems. A program which is to be used very often in a time-sharing or other multiprocessing environment should, if possible, be coded as a pure procedure, because then a single copy of the instruction words of this program may be used by several users without swapping. When a compiler is so constructed that it always generates a pure procedure as object code, this pure procedure is normally, in our sense, locally verifiable; references to subscripted variables are allowed, but the object code checks that each subscript is in range at the time of use.

All of the considerations of this chapter apply, of course, not only to assembly-language programs, but to compiled versions of algebraic-language programs as well. An algebraic-language program may be proved correct and yet not run properly when compiled, perhaps because the compiler is not correct (many compilers, of course, still have uncorrected errors in them), but more often because the proof of correctness was an "ideal" proof which did not take finite precision into account. This point is made in

[Redish 71], with specific reference to the proof given in [London 70] of the correctness of TREESORT III, an in-place sort written in ALGOL.

There are several approaches to the problem of proving the correctness of programs which perform floating point calculations. One is to use interval arithmetic, in which each real number z is represented by two floating point numbers x and y, for which it is known that $x \leq z \leq y$. Addition of two real numbers z_1 and z_2, with corresponding floating point numbers x_1, y_1, x_2, and y_2, is then performed (if all these numbers are positive) by calculating the largest legal floating point number which is less than x_1+x_2, and then the smallest such number which is greater than y_1+y_2. The other arithmetic operations are handled similarly. A subroutine package performing interval arithmetic is proved correct in [Good and London 68] (see also [Good and London 70]). The advantage of interval arithmetic is that we no longer need worry about subtraction or about the total number of operations performed; any algorithm, so long as it is correct in the ideal sense, is correct when executed using interval arithmetic. One disadvantage of interval arithmetic is that it is slow; this, however, could be remedied by building interval arithmetic into the hardware of some computer. Another, more serious, disadvantage is that, at the end of execution of a program using interval arithmetic, we are given an interval within which our answers are guaranteed to be contained, but we have no way of knowing, in advance, what the size of that interval will be.

For an excellent discussion of rounding errors, see [Wilkinson 63]. (Wilkinson uses fl(X1+X2) for our FLA(X1, X2); fl(X1*X2) for our FLM(X1, X2); and so on.)

EXERCISES

1. Complete the proof of partial correctness of the program of section 9-1.

2. Prove that the program of section 9-1 terminates. In what specific ways does this proof use the assertion ICA?

3. Complete the proof of partial correctness of the first program of section 9-2.

4. Complete the proof of partial correctness of the second program of section 9-2.

5. Consider the final program of section 7-2, which performs, in FORTRAN, a linear search of an array. Translate this program into the machine language of the original 16-bit computer of section 3-5, in much the same way as is illustrated in section 9-3. (Instruction modification is to be used in this program in the same way as illustrated in section 9-3, in order to traverse all the elements of the array to be searched.) Formulate initial, final, and intermediate assertions for this program, including the assertion IBA, and describe the assertion IBA precisely.

6. Prove the partial correctness of the program of problem 5 above.

7. Write a program in the assembly language of section 3-5 to calculate the value of Z according to the formula Z = C*C - A*A - B*B. Formulate initial assertions involving max and min, as in section 9-4, which must be satisfied for this program to be correct, and then prove its correctness.

8. Show the validity of the verification condition of the second control path of the program of section 9-4.

9. Verify properties (1) through (4) of the assertion FEQ defined in section 9-5.

10. Complete the proof of partial correctness of the program of section 9-5.

CHAPTER TEN

SEARCHING AND SORTING

10-1 Linear Searching and Timing

In any search routine, there is a table of values of some kind, and we are looking for some element of that table which satisfies some given condition. In the search routines to be considered in this chapter, our table is an array with subscripts extending from 1 to N, and our condition is that the element of this array which we are trying to find be equal to some fixed quantity, sometimes called the target. We shall call the target X, and the array A; thus, if the search is unsuccessful, the assertion which is true after the search has been completed is

$$(X \neq A(II),\ II\ .IN.\ (1..N))$$

while if the search is successful, the assertion is X = A(I), where I is an integer variable used by the program. Normally we need to have I set properly at the end of the search; that is, we need to know not only that X is equal to some element of A, but specifically which element.

The easiest method of searching an array is the linear search method, a program for which was given in section 7-2. We may write this program with assertions as:

```
      DIMENSION A(n)
C  N > 0, N ≤ n
      I = 1
C  1 ≤ I, I ≤ N, N ≤ n, (X ≠ A(II), II .IN. (1..I-1))
1     IF (X .EQ. A(I)) GO TO 2
      I = I + 1
      IF (I .LE. N) GO TO 1
C  (X ≠ A(II), II .IN. (1..N))
      . . .
C  X = A(I)
2     CONTINUE
```

This program has two stopping points, the first for an unsuccessful search and the second for a successful search. Each of these has its own assertion. (We recall that all stopping points of a program must be control points of it.) The intermediate assertion given here is the one that was given with this program in section 7-2, rewritten according to the conventions of section 7-3.

As we have seen, the proof of partial correctness of a program involves determining its various control paths, and then finding and proving the verification condition of each path. In most of the programs which follow, we shall not attempt to give the complete proof of partial correctness. Sometimes we shall merely list the verification conditions of all paths, as they would be obtained by either the forward or the back substitution methods of sections 6-4 and 6-5; the paths themselves are considered in the order in which they would be found by the method of section 6-3. At other times, we shall give a detailed analysis of a single path in a program, and leave the consideration of its other paths to the reader.

For the program above, there are four control paths, of which we shall consider only the third one:

C $1 \leq I$, $I \leq N$, $N \leq \underline{n}$, ($X \neq A(II)$, II .IN. (1..I-1))

[$1 \leq I$, $I \leq \underline{n}$]

($X \neq A(I)$)

$I = I + 1$

($I \leq N$)

C $1 \leq I$, $I \leq N$, $N \leq \underline{n}$, ($X \neq A(II)$, II .IN. (1..I-1))

The assertion $1 \leq I$ is preserved here because I is increased; $I \leq N$ at the end follows from the condition $I \leq N$ in the path; and $N \leq \underline{n}$ is preserved because N is unchanged. The assertion $X \neq A(I)$ is equivalent to ($X \neq A(II)$, II .IN. (I)), and this, together with ($X \neq A(II)$, II .IN. (1..I-1)), is equivalent to ($X \neq A(II)$, II .IN. (1..I)). Then I is increased by 1, giving ($X \neq A(II)$, II .IN. (1..I-1)) again. Formally, we have used the following general facts:

1. (F(I), I .IN. (J)) is the same as F(J)

2. (F(I), I .IN. S1) and (F(I), I .IN. S2), taken together, are the same as (F(I), I .IN. (S1 .U. S2))

3. (I..J) = (I..J-1) .U. (J), if $I \leq J$

In a program such as this, which has more than one stopping point, it is sometimes desirable to set up initial conditions under which the program must terminate at some one particular point. This may be done by placing the assertion .FAISE. at all <u>other</u> stopping points of the program, setting up the initial conditions at the beginning, and proving the correctness of the resulting program. In the above program, if it is to terminate at statement num-

ber 2, there must exist an integer J, $1 \leq J \leq N$, with $X = A(J)$. We may write this as

$$EI(J, (1..N)), X = A(J)$$

Using this as part of the initial assertion, we may rewrite our program with assertions as

```
      DIMENSION A(n)
C  N > 0, N ≤ n, EI(J, (1..N)), X = A(J)
      I = 1
C  1 ≤ I, I ≤ N, N ≤ n, EI(J, (I..N)), X = A(J)
1     IF (X .EQ. A(I)) GO TO 2
      I = I + 1
      IF (I .LE. N) GO TO 1
C  .FALSE.
      . . .
C  X = A(I)
2     CONTINUE
```

This time, we assert at statement number 1 that the integer J, which is known to exist, has not been checked so far -- that is, it must lie between I and N, inclusive. Let us examine the control path ending at the assertion .FALSE.:

```
C  1 ≤ I, I ≤ N, N ≤ n, EI(J, (I..N)), X = A(J)
      [1 ≤ I, I ≤ n]
      (X ≠ A(I))
      I = I + 1
      (I > N)
C  .FALSE.
```

The verification condition of this path is that, if the path is

taken, .FALSE. is true. In other words, we must prove that the path can never be taken when started with its initial assertion valid. Since $I \leq N$ at the beginning, but $I + 1 > N$, we must have had $I = N$, and the integer J is in (N..N). This means that $J = N$ and thus $X = A(N)$. But then $X = A(I)$, since $I = N$, and the path is never taken because of the condition $X \neq A(I)$ in it.

Now let us assume that the array A is sorted in ascending order. This may be expressed in two ways:

$$(A(K) \leq A(K+1),\ K\ .IN.\ (1..N-1))$$

(which says that each element of the array is not less than the immediately preceding one), or

$$(A(K) \leq A(L),\ K\ .IN.\ (1..N),\ L\ .IN.\ (1..N)\ .ST.\ (K < L))$$

(which says, more generally, that each element of the array is not less than any element which precedes it. Recall that we are using .ST. to mean "such that.") The equivalence of these two, for $N \geq 2$, may be shown by induction on N. For $N = 2$, the equivalence is straightforward, while if

$$(A(K) \leq A(K+1),\ K\ .IN.\ (1..N-1)) \Leftrightarrow$$
$$(A(K) \leq A(L),\ K\ .IN.\ (1..N),\ L\ .IN.\ (1..N)\ .ST.\ (K < L))$$

then

$$(A(K) \leq A(K+1),\ K\ .IN.\ (1..(N+1)-1) \Leftrightarrow$$
$$(A(K) \leq A(K+1),\ K\ .IN.\ (1..N-1))\ \underline{\text{and}}\ A(N) \leq A(N+1) \Leftrightarrow$$
$$(A(K) \leq A(L),\ K\ .IN.\ (1..N),\ L\ .IN.\ (1..N)\ .ST.\ (K < L))$$
$$\underline{\text{and}}\ (A(N) \leq A(L),\ L\ .IN.\ (N+1)) \Leftrightarrow$$
$$(A(K) \leq A(L),\ K\ .IN.\ (1..N+1),\ L\ .IN.\ (1..N+1)\ .ST.\ (K < L))$$

The last of these equivalences follows by noting that, if K = N+1, we cannot have K < L, whereas if L = N+1, then A(K) $\leq$ A(L) since A(K) $\leq$ A(N) and A(N) $\leq$ A(L).

We shall define

ASC(A, I1, I2) = (A(K) $\leq$ A(K+1), K .IN. (I1..I2-1))

-- that is, ASC(A, I1, I2) means that the elements A(I1) through A(I2) are sorted in ascending order. When necessary, we shall use the fact, proved in the same way as above, that

ASC(A, I1, I2) = (A(K) $\leq$ A(L), K .IN. (I1..I2),
L .IN. (I1..I2) .ST. (K < L))

A linear search of a sorted array can stop when we pass the point where the searched-for element "ought to be." The following program, given with assertions, makes this kind of a search:

```
      DIMENSION A(n)
C  N > 0, N ≤ n, ASC(A, 1, N)
      I = 1
C  1 ≤ I, I ≤ N, N ≤ n, ASC(A, 1, N), (X ≠ A(II), II .IN. (1..I-1))
1     IF (X-A(I)) 4, 3, 2
2     I = I + 1
      GO TO 1
C  X = A(I)
3     CONTINUE
      . . .
C  (X ≠ A(II), II .IN. (1..N))
4     CONTINUE
```

Let us see what happens at the end of an unsuccessful search by

considering the control path ending at statement number 4:

C $1 \leq I$, $I \leq N$, $N \leq n$, ASC(A, 1, N), $(X \neq A(II)$, II .IN. $(1..I-1))$

$[1 \leq I, I \leq n]$

$(X-A(I) < 0)$

C $(X \neq A(II)$, II .IN. $(1..N))$

Since ASC(A, 1, N) implies that $(A(K) \leq A(L)$, K .IN. $(1..N)$, L .IN. $(1..N)$ .ST. $(K < L))$, we have, in particular, that $(A(I) \leq A(II)$, II .IN. $(I+1..N))$. Since $X < A(I)$, this gives $(X < A(II)$, II .IN. $(I+1..N))$, which implies $(X \neq A(II)$, II .IN. $(I+1..N))$; and this, together with $(X \neq A(II)$, II .IN. $(1..I-1))$, from the beginning of the path, and $X \neq A(I)$ (implied by $X-A(I) < 0$), gives the assertion at the end of the path.

The proof of termination of the above two routines is very simple. In each case, N-I is a loop expression, as may easily be checked.

It is well known that the linear search is a relatively slow searching method. We shall now show how our methods may be used to prove exactly how many steps an algorithm takes. Let NSTEPS be a pseudo-variable whose value, at any time in a given program, is the total number of steps taken so far. If by "steps" we mean "executable statements," then each executable statement increases the value of NSTEPS by 1. We may now include assertions in our program which involve NSTEPS; the initial assertion is NSTEPS = 0. If the program with these new assertions can be proved correct, then the total number of steps taken by the algorithm is given by the value of NSTEPS as specified in the final assertion.

For the first search program of this section, we may place the assertion NSTEPS = 3*I-2 at statement number 1, the assertion

NSTEPS = 3*N+1 at the failure exit, and the assertion NSTEPS = 3*I-1 at the success exit. With these assertions, the program may again be proved correct. Thus, if the search is successful and A(I) = X, the program takes 3*I-1 steps (this, of course, depends on I, that is, on whether X is found near the front of the table or near the back). If the search is unsuccessful, the program always takes 3*N+1 steps.

Most programs do not always take the same number of steps each time. In such a program, the final (and often the intermediate) assertions about NSTEPS are not equalities, but only upper and lower bounds. In the program above which searches a sorted array, we may put NSTEPS = 0 at the beginning, NSTEPS = 3*I-2 at statement number 1, and NSTEPS = 3*I-1 at the success exit, as before, but the assertions at the failure exit are $2 \leq$ NSTEPS and NSTEPS $\leq 3*N+1$. Thus this program is "at most linear" -- it takes no more than a number of steps which is some multiple of the size of the table -- but it can take much less than that. (For a program to be strictly linear in a variable N, we would have to have final assertions of the form $aN+b \leq$ NSTEPS and NSTEPS $\leq cN+d$.)

Other facts about the timing of programs may also be determined in this way. The number of times a certain individual statement is executed, for example, may also be considered as a pseudo-variable whose value is initialized to zero, and which is increased by one each time that statement is executed. The number of multiplications which take place in a program, the number of times a particular subroutine is called, and the total number of assignment statements executed, are further examples of quantities representable as pseudo-variables in this way.

10-2 Binary and Hash Table Search

A binary, or logarithmic, search of a sorted table is carried out by repeatedly dividing the table in half. We shall study a version of the binary search which makes a two-way test each time the table is divided, rather than a three-way test; that is, it does not check, each time the table is divided, whether the middle element of the current table A is equal to the target X. The use of a two-way test increases the efficiency of the binary search on most computers.

The integer variables IFIRST and ILAST will denote the first and last indices, respectively, of that part of the table in which we are currently looking. Our key assertion is that, if the element X is anywhere in A, its index must be between IFIRST and ILAST, inclusive. This may be expressed as

```
(X ≠ A(II), II .IN. (1..N)) .OR.
     (EI(J, (IFIRST..ILAST)), X = A(J))
```

Note that this is not quite the same as

```
(X ≠ A(II), II .IN. ((1..N) .D. (IFIRST..ILAST)))
```

because of the possibility of duplicate items in the table. In particular, we might have A(IFIRST-1) = A(IFIRST) = X or A(ILAST) = (A(ILAST+1) = X, either of which would make the first of the above two conditions true and the second one false.

Initially, IFIRST and ILAST are set to 1 and N, respectively. To divide the table in half, we calculate the average of IFIRST and ILAST, that is, (IFIRST+ILAST)/2, and call this IMIDL. Since the division is performed without rounding, either IMIDL or

IMIDL + 1/2 is the exact average of IFIRST and ILAST. This suggests that IMIDL should be taken to be the final index of the first half of that part of the array between IFIRST and ILAST, inclusive. If X is larger than A(IMIDL), it belongs in the second half of the table; the new IFIRST will be IMIDL+1, and ILAST remains unchanged. Otherwise, X belongs in the first half of the table, and, in this case, IFIRST remains unchanged and IMIDL becomes the new ILAST.

The process of dividing the table in half terminates when we have a table of length 1, that is, when IFIRST and ILAST are equal. At this point, we simply check to see if X is equal to A(IFIRST). The program with assertions is as follows:

```
      DIMENSION A(n)
C  N > 0, N ≤ n, ASC(A, 1, N)
      IFIRST = 1
      ILAST = N
      GO TO 17
C  1 ≤ IFIRST, IFIRST < ILAST, ILAST ≤ N, N ≤ n,
C  ASC(A, 1, N), ((X ≠ A(II), II .IN. (1..N)) .OR.
C  (EI(J, (IFIRST..ILAST)), X = A(J)))
11    IMIDL = (IFIRST+ILAST)/2
      IF (X .GT. A(IMIDL)) GO TO 15
      ILAST = IMIDL
      GO TO 17
15    IFIRST = IMIDL+1
C  1 ≤ IFIRST, IFIRST ≤ ILAST, ILAST ≤ N, N ≤ n,
C  ASC(A, 1, N), ((X ≠ A(II), II .IN. (1..N)) .OR.
C  (EI(J, (IFIRST..ILAST)), X = A(J)))
17    IF (IFIRST .NE. ILAST) GO TO 11
```

```
      IF (X .EQ. A(IFIRST)) GO TO 19
C  (X ≠ A(II), II .IN. (1..N))
      . . .
C  X = A(IFIRST)
19    CONTINUE
```

The assertion ASC(A, 1, N) is as in the preceding section. We have used one more control point than is absolutely necessary, in order to shorten the analysis of control paths. The assertions at the two intermediate control points are the same, except for IFIRST < ILAST at statement number 11 versus IFIRST ≤ ILAST at statement number 17.

There are six control paths, as follows:

```
Path 1  C  N > 0, N ≤ n, ASC(A, 1, N)
           IFIRST = 1
           ILAST = N
        C  1 ≤ IFIRST, IFIRST ≤ ILAST, ILAST ≤ N, N ≤ n,
        C  ASC(A, 1, N), ((X ≠ A(II), II .IN. (1..N)) .OR.
        C  (EI(J, (IFIRST..ILAST)), X = A(J)))
```

The verifications here are straightforward; in particular, if IFIRST = 1 and ILAST = N, the two conditions connected by .OR. are exact opposites of each other.

```
Path 2  C  1 ≤ IFIRST, IFIRST < ILAST, ILAST ≤ N, N ≤ n,
        C  ASC(A, 1, N), ((X ≠ A(II), II .IN. (1..N)) .OR.
        C  (EI(J, (IFIRST..ILAST)), X = A(J)))
           IMIDL = (IFIRST+ILAST)/2
           [1 ≤ IMIDL, IMIDL ≤ n]
           (X > A(IMIDL))
```

```
            IFIRST = IMIDL + 1
      C  1 ≤ IFIRST, IFIRST ≤ ILAST, ILAST ≤ N, N ≤ n,
      C  ASC(A, 1, N), ((X ≠ A(II), II .IN. (1..N)) .OR.
      C  (EI(J, (IFIRST..ILAST)), X = A(J)))
```

When .OR. appears in the initial assertion of a path, as here, we must consider separately the two assertions connected by it. If (X ≠ A(II), II .IN. (1..N)) at the beginning of the path, then this will still be true at the end. If (EI(J, (IFIRST..ILAST)), X = A(J)) at the beginning of the path, and if the path is actually followed, then we may show that this will still be true at the end, as follows. Since X > A(IMIDL), we have at the end X > A(IFIRST+1), and from ASC(A, 1, N) we then obtain (X > A(II), II .IN. (1..IFIRST-1)) and hence (X ≠ A(II), II .IN. (1..IFIRST-1)). Hence the J that existed at the beginning of the path must have been in the set (IFIRST..ILAST) as that set exists at the end of the path.

As for the other assertions at the end of this path, ILAST ≤ N, N ≤ n, and ASC(A, 1, N) are unchanged during the path. Since IFIRST < ILAST and ILAST+ILAST is even, (IFIRST+ILAST)/2 must be strictly less than (ILAST+ILAST)/2 -- that is, at the end, IMIDL < ILAST and hence IFIRST ≤ ILAST. Finally, since IFIRST < ILAST, we have IFIRST ≤ IMIDL after calculation of IMIDL, and hence 1 ≤ IFIRST at the end. The verification that the restriction [1 ≤ IMIDL, IMIDL ≤ n] holds within the path is straightforward.

```
Path 3  C  1 ≤ IFIRST, IFIRST < ILAST, ILAST ≤ N, N ≤ n,
        C  ASC(A, 1, N), ((X ≠ A(II), II .IN. (1..N)) .OR.
        C  (EI(J, (IFIRST..ILAST)), X = A(J)))
              IMIDL = (IFIRST+ILAST)/2
```

```
                [1 ≤ IMIDL, IMIDL ≤ n]
                (X ≤ A(IMIDL))
                ILAST = IMIDL
          C  1 ≤ IFIRST, IFIRST ≤ ILAST, ILAST ≤ N, N ≤ n,
          C  ASC(A, 1, N), ((X ≠ A(II), II .IN. (1..N)) .OR.
          C  (EI(J, (IFIRST..ILAST)), X = A(J)))
```

Here, as before, (X ≠ A(II), II .IN. (1..N)) is true at the end if it is true at the beginning. Otherwise, there are two possibilities. If X = A(IMIDL) in the path, then (EI(J, (IFIRST..ILAST)), X = A(J)) is certainly satisfied after ILAST is set equal to IMIDL. If X < A(IMIDL), then X < A(II) for all II greater than IMIDL, again by the ascending-order condition, and thus the integer J at the beginning of the path must lie between the final values of IFIRST and ILAST, inclusive. Since IFIRST < ILAST, we have at the end IFIRST ≤ IMIDL and hence IFIRST ≤ ILAST (equality might hold, because the division is performed without rounding). Also, the value of ILAST cannot increase along this path, so we still have at the end ILAST ≤ N. The final assertions 1 ≤ IFIRST, N ≤ n, and ASC(A, 1, N) remain unchanged along the path.

The remainder of the proof is straightforward. Path 4 goes from statement number 17 to statement number 11; its verification condition is immediate. Path 5 goes from statement number 17 to statement number 19; the final assertion in this path follows immediately from the condition in the path brought about by the final IF statement. Path 6 goes from statement number 17 to the "unsuccessful search" exit. Since IFIRST = ILAST, J (if it exists) must be equal to IFIRST, and X = A(IFIRST); but we know this is false, and hence the first of the two original alternatives, namely

(X ≠ A(II), II .IN. (1..N)), must have been true at the beginning of the path. The loop expression of this program is ILAST-IFIRST; each time around the loop, either IFIRST increases or ILAST decreases, and ILAST-IFIRST never becomes negative. Thus the program is correct.

We now consider a hash table H of length L, in which there are currently N entries, leaving L-N empty spaces in the table. Let Z be the special word (often, but not always, zero) which denotes an empty space in the hash table. There is then some set of indices of entries in the table, and, for any index K which is not in this set, we have H(K) = Z. The set of indices is determined by the entries themselves, which we shall denote by T(1), ..., T(N). (There is nothing in this notation to distinguish T from a linear array of entries, and, in fact, we may consider it as such, even though there would normally not be such an array in memory.) All the T(I) are assumed to belong to some set ENT of all possible entries in the table, and, for each element Q of ENT, we are given HASH(Q), the hash index of Q, which is between 1 and L inclusive. If no collisions exist in the hash table, we have

$$\{(T(K) = H(HASH(T(K))),\ K\ .IN.\ (1..N)),\ (H(K) = Z,$$
$$K\ .IN.\ ((1..N)\ .D.\ SET(HASH(T(K)),\ K\ .IN.\ (1..N))))\}$$

If collisions do exist -- that is, if HASH(T(I)) = HASH(T(J)) for some I ≠ J -- then we must have a collision handling method. Let us assume that each T(K) is always kept in H, rather than in some auxiliary list space, and that there is a function COLL(Q, J) giving the index of the J-th "try" at placing Q in the table. That is, if H(HASH(Q)) ≠ Z, then we try H(COLL(Q, 1)), H(COLL(Q, 2)), and so on in this order, until we find J such that H(COLL(Q, J)) =

Z, and this is the point at which Q is placed. For convenience, we let COLL(Q, 0) = HASH(Q) for each Q. Our assertion is now

```
((EI(J, (1..L)), T(K) = H(COLL(T(K), J)), (H(COLL(T(K), II))
    ≠ Z, II .IN. (0..J))), K .IN. (1..N))
```

Let us call this assertion HTA(H, L, T, N). It is preserved by hash table insertion routines, and is assumed to be true whenever a hash table search is performed. For the simplest collision handling method (linear with wraparound) we have COLL(Q, J) = HASH(Q) + (if HASH(Q) + J ≤ L then J else (J-L)); the following routine searches a hash table under these conditions:

```
C  HTA(H, L, T, N), L > N
       I = HASH(X)
C  HTA(H, L, T, N), (X ≠ H(COLL(X, J)), Z ≠ H(COLL(X, J))),
C  J .IN. (0..I-1-HASH(X)+(if I .LT. HASH(X) then L else 0)))
1      IF (X .EQ. H(I)) GO TO 2
       IF (X .EQ. Z) GO TO 3
       I = I + 1
       IF (I .LE. L) GO TO 1
       I = 1
       GO TO 1
C  X = H(I)
2      CONTINUE
       . . .
C  (X ≠ T(K), K .IN. (1..N))
3      CONTINUE
```

When a hash table, as above, is used to associate information with the quantity being hashed, then the basic hash table as-

sertion must be modified. For example, there may be a second hash table H2, also of length L, as well as N further entries T2(K), $1 \leq K \leq N$; each T2(K) is a quantity which we wish to associate, in some way, with the corresponding T(K), and this is done by placing T2(K) in H2 as H2(KK) where T(K) has been placed as H(KK). The assertion that all these quantities have been properly placed is

```
((EI(J, (1..L)), T(K) = H(COLL(T(K), J)),
  T2(K) = H2(COLL(T(K), J)), (H(COLL(T(K), II))
  ≠ Z, II .IN. (0..J))), K .IN. (1..N))
```

We may now substitute this assertion for HTA(H, L, T, N) in programs such as the one displayed above.

10-3 Permutations and Unsortedness

In proving the correctness of programs which rearrange data, such as sorting and merging programs, certain assertions are made whose properties depend on a study of permutations. The simplest of these is the assertion that an array has been sorted. Let us consider the following array of four elements with values given:

$$A(1) = 13 \qquad A(2) = 19 \qquad A(3) = 11 \qquad A(4) = 16$$

If we were to sort this array and place the sorted results in a new array B, without changing the original array, we would have

$$B(1) = 11 \qquad B(2) = 13 \qquad B(3) = 16 \qquad B(4) = 19$$

Here we clearly must have $B(1) \leq B(2)$, $B(2) \leq B(3)$, and $B(3) \leq B(4)$, or, in the terminology of section 10-1, ASC(B, 1, 4). But to prove the correctness of a program which performs this sorting, it would be a mistake to regard the above relations as the entire final assertion. If we did that, then a "sorting program" which terminated in the above case with

$$B(1) = 1 \qquad B(2) = 2 \qquad B(3) = 3 \qquad B(4) = 4$$

would be just as "correct." We also need to know that $B(1) = A(x)$, for some value of x with $1 \leq x \leq 4$; and similarly $B(2) = A(y)$, $B(3) = A(z)$, and $B(4) = A(w)$. Moreover, we must know that x, y, z, and w are all distinct.

Let us refer to x, y, z, and w as $f(1)$, $f(2)$, $f(3)$, and $f(4)$, respectively. Then $B(i)$ must be equal to $A(f(i))$, for $1 \leq i \leq 4$. The values of the function f lie between 1 and 4 inclusive, and

they are all distinct, which means that f is a permutation on the integers 1 through 4. In the case above we have

$$f(1) = 3 \qquad f(2) = 1 \qquad f(3) = 4 \qquad f(4) = 2$$

We could just as easily have written here $A(i) = B(g(i))$, for

$$g(1) = 2 \qquad g(2) = 4 \qquad g(3) = 1 \qquad g(4) = 3$$

and $1 \leq i \leq 4$. The function g is also a permutation; it is the inverse of f, and $f(g(i)) = g(f(i)) = i$ for $1 \leq i \leq 4$.

Using the terminology of section 7-3, we may write

(EP(P, (1..N)), (A(K) = B(P(K)), K .IN. (1..N)))

as the assertion that the elements of the array B are a rearrangement of the elements of A, where N is the number of elements. That is, "there exists a permutation, call it P, on the integers 1 through N, such that A(K) = B(P(K)) for each K, $1 \leq K \leq N$." We denote this assertion by PERM(A, B, N). If $A(x) = B(x)$, $1 \leq x \leq N$, then certainly PERM(A, B, N) holds, because we need only to set $P(x) = x$ for each x; this is the identity permutation. Thus PERM(A, B, N) becomes an initial as well as a final assertion in a program to sort the array A into the new array B. It is not surprising that, in many sort programs, it is also an intermediate assertion; that is, it remains true as the sort progresses.

Now suppose that, instead of placing our sorted results in a new array, we are sorting an array in place. In this case, just as in section 6-2, we must introduce new variables into our program for the sole purpose of allowing us to state our assertions properly. If the array A is being sorted, we may invent a new array -- call it A0 -- with $A(x) = A0(x)$ for $1 \leq x \leq N$, where the dimension

of A is N. This is then our initial assertion, and our final assertion is PERM(A, A0, N) (as before, this is also true at the start of the program), as well as ASC(A, 1, N). It is to be noted that this subterfuge is necessary only when the program contains no other known array, or its equivalent, which contains the values of the elements of A. For example, if a file is read into memory and sorted, we have the same data in two places -- in memory and on the file -- and so our assertion may simply be that each variable A(x) is at some unique place within the file.

We now prove that PERM(A, B, N) is preserved by an interchange of two elements of the array A; that is, we shall show that the verification condition of the path

```
C  PERM(A, B, N), 1 ≤ I, I ≤ N, 1 ≤ J, J ≤ N
       T = A(I)
       A(I) = A(J)
       A(J) = T
C  PERM(A, B, N)
```

is valid. This prototype situation may then be applied to an arbitrary interchange sort program, in which an array is sorted by successive interchanges. (For an in-place sort, we take B to be the same as A0 above.) Let f be the permutation whose existence is implied by PERM(A, B, N) at the beginning of this path; we wish to find a permutation g which will make PERM(A, B, N) true at the end of the path. The definition of g in terms of f is as follows:

$$g(i) = f(j),\ g(j) = f(i),\ g(k) = f(k) \text{ for } k \neq i, j\ (1 \leq k \leq N)$$

where i and j are the respective values of I and J (note that the values of I, J, and N are not changed in the above path). If $x \neq$

i, j, then $A(x) = B(f(x)) = B(g(x))$ at the beginning of the path; since $A(x)$ is not changed anywhere in the path, this relation still holds true at the end. If $A(I) = B(f(I))$ and $A(J) = B(f(J))$ at the beginning of the path, then by forward substitution we obtain $A(I) = B(f(J)) = B(g(I))$ and $A(J) = B(f(I)) = B(g(J))$ at the end of the path. This completes the proof.

Let us now prove the correctness of a typical interchange sort routine. We use PERM and ASC as above; PERM(A, AO, N) will be preserved throughout the algorithm (except during the interchanges themselves). The routine is as follows:

```
      DIMENSION A(n)
C  N .GE. 2, N .LE. n, (A(K) = AO(K), K .IN. (1..N))
      I = 1
C  1 .LE. I, I .LT. N, N .LE. n, PERM(A, AO, N), ASC(A, 1, I)
1     IF (A(I) .GT. A(I+1)) GO TO 2
      I = I + 1
      IF (I .NE. N) GO TO 1
      GO TO 3
2     T = A(I)
      A(I) = A(I+1)
      A(I+1) = T
      IF (I .EQ. 1) GO TO 1
      I = I - 1
      GO TO 1
C  PERM(A, AO, N), ASC(A, 1, N)
3     CONTINUE
```

This routine looks through the array A in the forward direction for an adjacent pair of elements (A(I), A(I+1)) which are out of order. It is always assumed that the elements A(1) through A(I)

are in ascending order, so that, if I becomes equal to N, the array is sorted. If the pair (A(I), A(I+1)) is found to be in order, then I is increased by 1; if this pair is out of order, it is interchanged and I is decreased by 1.

There are five control paths, as follows:

Path 1 C $N \geq 2$, $N \leq \underline{n}$, (A(K) = AO(K), K .IN. (1..N))

$I = 1$

C $1 \leq I$, $I < N$, $N < \underline{n}$, PERM(A, AO, N), ASC(A, 1, I)

Here $1 \leq I$ follows from $I = 1$; $I < N$ follows from $I = 1$ and $N \geq 2$; $N \leq \underline{n}$ is unchanged by the path; and ASC(A, 1, I) is ASC(A, 1, 1), which is always true. (Formally, ASC(A, 1, 1) is (A(K) $\leq$ A(K+1), K .IN. (1..0)), which follows since (1..0) is the null set.) The fact that PERM(A, AO, N) is implied by the initial assertion here follows from our discussion of the identity permutation.

Path 2 C $1 \leq I$, $I < N$, $N \leq \underline{n}$, PERM(A, AO, N), ASC(A, 1, I)

[$1 \leq I$, $I \leq \underline{n}$, $1 \leq I+1$, $I+1 \leq \underline{n}$]

(A(I) $\leq$ A(I+1))

$I = I + 1$

($I \neq N$)

C $1 \leq I$, $I < N$, $N \leq \underline{n}$, PERM(A, AO, N), ASC(A, 1, I)

The restrictions in the path are all clearly implied by the initial assertion. The inequality $1 \leq I$ is preserved because I increases, and $N \leq \underline{n}$ is also preserved. The initial assertion $I < N$ becomes $I+1 < N$, or $I \leq N$, and since $I \neq N$ in the path, we must have $I < N$ at the end. The assertion PERM(A, AO, N) is unchanged by this path. Finally, ASC(A, 1, I) at the beginning of the path,

together with $A(I) \leq A(I+1)$, give ASC(A, 1, I+1) (by the first definition of ASC, given above), and then I is increased by 1, giving ASC(A, 1, I) again.

Path 3 C $1 \leq I,\ I < N,\ N \leq \underline{n}$, PERM(A, AO, N), ASC(A, 1, I)

$[1 \leq I,\ I \leq \underline{n},\ 1 \leq I+1,\ I+1 \leq \underline{n}]$

$(A(I) \leq A(I+1))$

$I = I + 1$

$(I = N)$

C PERM(A, AO, N), ASC(A, 1, N)

Again, the restrictions in the path are implied by the initial assertion, and PERM(A, AO, N) is unchanged by the path; and, just as in the previous path, ASC(A, 1, I) holds at the end. Since I = N, this becomes ASC(A, 1, N).

Path 4 C $1 \leq I,\ I < N,\ N \leq \underline{n}$, PERM(A, AO, N), ASC(A, 1, I)

$[1 \leq I,\ I \leq \underline{n},\ 1 \leq I+1,\ I+1 \leq \underline{n}]$

$(A(I) > A(I+1))$

$[1 \leq I,\ I \leq \underline{n}]$

$T = A(I)$

$[1 \leq I,\ I \leq \underline{n},\ 1 \leq I+1,\ I+1 \leq \underline{n}]$

$A(I) = A(I+1)$

$[1 \leq I+1,\ I+1 \leq \underline{n}]$

$A(I+1) = T$

$(I = 1)$

C $1 \leq I,\ I < N,\ N \leq \underline{n}$, PERM(A, AO, N), ASC(A, 1, I)

There are four restricted commands in this path, and all the restrictions are implied by the initial assertion, since I does not change during the path. The assertions $1 \leq I$, $I < N$, and $N \leq \underline{n}$ are likewise unchanged during the path. The fact that PERM(A, AO,

N) is preserved in a sequence like this follows from our discussion earlier in this section. Finally, ASC(A, 1, I) is ASC(A, 1, 1), which, as we noted before, is always true.

Path 5 C $1 \leq I$, $I < N$, $N \leq \underline{n}$, PERM(A, AO, N), ASC(A, 1, I)

$[1 \leq I,\ I \leq \underline{n},\ 1 \leq I+1,\ I+1 \leq \underline{n}]$

$(A(I) > A(I+1))$

$[1 \leq I,\ I \leq \underline{n}]$

$T = A(I)$

$[1 \leq I,\ I \leq \underline{n},\ 1 \leq I+1,\ I+1 \leq \underline{n}]$

$A(I) = A(I+1)$

$[1 \leq I+1,\ I+1 \leq \underline{n}]$

$A(I+1) = T$

$(I \neq 1)$

$I = I - 1$

C $1 \leq I$, $I < N$, $N \leq \underline{n}$, PERM(A, AO, N), ASC(A, 1, I)

Here we must have had $2 \leq I$ at the beginning of the path (since $1 \leq I$ and $I \neq 1$), and therefore $1 \leq I$ at the end. The inequality $I < N$ is preserved since I decreases; the inequality $N \leq \underline{n}$ is also preserved. PERM(A, AO, N) is preserved here just as it was in the preceding path. Finally, we may write ASC(A, 1, I) at the beginning of the path as $A(I-1) \leq A(I)$ together with ASC(A, 1, I-1). Then ASC(A, 1, I-1) is preserved by the first three assignments in this path, since it is a condition only on values $A(\underline{x})$ for $\underline{x} \leq I-1$. When I is decreased by 1, this becomes ASC(A, 1, I) again. Thus the sort program is partially correct.

The loop expression for this sort program is a bit more complex than the loop expressions we have considered so far. It is

the total number of pairs (A(I), A(J)) which are currently out of order. We shall call this the unsortedness of the array A; it is very commonly a loop expression in sort programs. Formally, it may be expressed as

```
SUM(if(I < J, A(I) > A(J)) then 1 else 0,
        I .IN. (1..N), J .IN. (1..N))
```

Notice that this is an integer expression which cannot be negative. We shall now prove, for the prototype control path discussed earlier in this section (in connection with preserving PERM(A, B, N)), that the unsortedness of the given array is always less at the end of the path than it was at the beginning, provided that A(I) and A(J) were out of order to begin with. That is, we shall prove this with the additional assertions $I < J$ and $A(I) > A(J)$ at the beginning of the path.

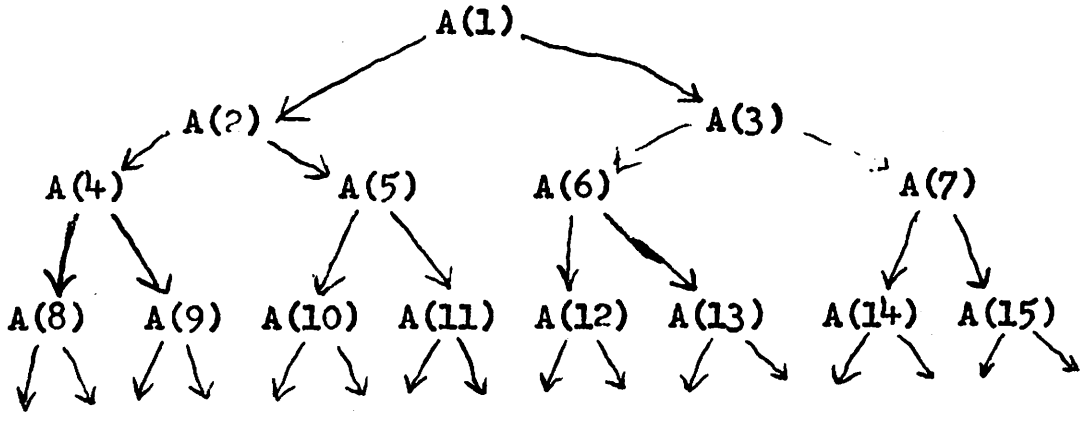
Clearly the pair (I, J), which was out of order, is placed in order by the given operation; this decreases by one the number of pairs out of order. We complete the proof by showing that for any pair (I, $\underline{x}$) which is in order and is put out of order by the interchange of I and J, the pair (J, $\underline{x}$) will be out of order and will be placed in order; a similar statement may then be proved if we start with a pair (J, $\underline{x}$) which is put out of order, and hence there can be no further net increase in the unsortedness. Let (I, $\underline{x}$) be in order; then either $\underline{x} < I$ and $A(\underline{x}) \leq A(I)$, or else $I < \underline{x}$ and $A(I) \leq A(\underline{x})$. In the first instance, $\underline{x} < J$ since $I < J$, and so (I, $\underline{x}$) cannot be put out of order by the transposition. In the second, in order for (I, $\underline{x}$) to be put out of order, we must have $\underline{x} < J$. By hypothesis, $A(I) > A(J)$, and since $A(I) \leq A(\underline{x})$ we must have $A(\underline{x}) > A(J)$. But this shows that (J, $\underline{x}$) is out of order be-

fore the transposition, and in order after it. As mentioned before, a similar argument holds in reverse, and the proof is thus complete.

10-4 The Tree Sort

Most of the more advanced sorting methods in existence have timings proportional to $n \log_2 n$, where n is the number of items being sorted. One of these methods is the tree sort, which has the additional advantage of being an in-place sort requiring no recursion and relying entirely on interchanges.

The tree sort works with what may be called a "simulated tree." That is, there is no actual tree structure in memory, and no pointers; the elements of the array (which we shall again call A) to be sorted are simply considered as belonging to an abstract tree, as follows:



and so on. Note that there is an arrow from each A(i) to A(2i) and to A(2i+1), unless 2i (or 2i+1, respectively), is larger than the size of the array. Putting it another way, there is an arrow from each A(i/2) to A(i), for $i \geq 2$, where the division is performed without rounding.

The first step in the tree sort is to rearrange the array A, by means of interchanges, in such a way that every path in the above tree is sorted in descending order. That is, we must have $A(i/2) \geq A(i)$ for each $i \geq 2$. This is done by calling a subroutine whose correctness we will now prove. The remainder of the

tree sort is postponed until section 20-5, when we shall be able to apply certain facts about subroutine parameters. In fact, the correctness of the main routine which calls this subroutine depends in an essential way upon the method by which the parameters are called.

The subroutine is called SIFTUP. It has two parameters, I and N. When it is called, it assumes that all paths in the above graph involving A(I+1) through A(N) are sorted in descending order; after it is done, all paths involving A(I) through A(N) are sorted in descending order. This means that after the main program calls SIFTUP(I, N), it decreases I by 1 and calls SIFTUP(I, N) again. The initial value of I in the main routine is N/2, because there are no paths in the graph involving A(N/2) through A(N); after SIFTUP(1, N) has been called, all paths in the entire graph will be in descending order, and, in particular, A(1) will be the largest element of A.

We shall not attempt to explain how SIFTUP works, but shall simply give the program with its assertions:

```
      SUBROUTINE SIFTUP(I, N)
C  1 ≤ IO, I ≤ N, N ≤ n, I = IO,
C  (A(K/2) ≥ A(K), K .IN. (2*IO+1..N)),
C  EP(P, (1..N)), (A(II) = AO(P(II)), II .IN. (1..N))
      COPY = A(I)
1     J = 2*I
C  ((I = IO, (A(K/2) ≥ A(K), K .IN. (2*IO+1..N))) .OR.
C  (I ≥ 2*IO, A(I/2) > COPY, A(I/2) ≥ A(I),
C  (A(K/2) ≥ A(K), K .IN. (2*IO..N)))), I ≤ N, 2*I = J,
C  (EP(P, (1..N)), COPY = AO(P(I)), (A(II) = AO(P(II)),
C  II .IN. ((1..N) .D. (I)))), 1 ≤ IO, N ≤ n
      IF (N-J) 4, 3, 2
```

```
2       IF (A(J+1) .LE. A(J)) GO TO 3
        J = J + 1
C   ((I = IO, (A(K/2) ≥ A(K), K .IN. (2*IO+1..N))) .OR.
C   (I ≥ 2*IO, A(I) ≥ A(J), A(I/2) > COPY, A(I/2) ≥ A(I),
C   (A(K/2) ≥ A(K), K .IN. (2*IO..N)))), I = J/2, 2*I ≤ J, J ≤ N,
C   ((2*I < N, A(J) = MAX(A(2*I), A(2*I+1))) .OR.
C   (2*I = N, A(J) = A(N))), (EP(P, (1..N)), COPY = AO(P(I)),
C   (A(II) = AO(P(II)), II .IN. ((1..N) .D. (I)), 1 ≤ IO, N ≤ n
3       IF (A(J) .LE. COPY) GO TO 4
        A(I) = A(J)
        I = J
        GO TO 1
4       A(I) = COPY
C   EP(P, (1..N)), (A(II) = AO(P(II)), II .IN. (1..N)),
C   (A(K/2) ≥ A(K), K .IN. (2*IO..N))
        RETURN
        END
```

Since the array A is being sorted in place, we must, as before, introduce another array AO, such that the elements of A are a rearrangement of the elements of AO both before and after SIFTUP is executed. In addition, we have introduced a variable IO to stand for the initial value of I, since SIFTUP may change the value of I. As before, $\underline{n}$ denotes the total amount of space reserved for the array A.

In order to simplify the proof, we make the following abbreviations:

A1 = (1 ≤ IO)

A2 = (I = IO)

A3 = (I ≤ N)

A4 = (N $\leq$ n)

A5 = (A(K/2) $\geq$ A(K), K .IN. (2*IO+1..N))

A6 = (EP(P, (1..N)), (A(II) = AO(P(II)), II .IN. (1..N))

A7 = (I $\geq$ 2*IO)

A8 = (A(I/2) $\geq$ A(I))

A9 = (A(K/2) $\geq$ A(K), K .IN. (2*IO..N))

A10 = (2*I = J)

A11 = (A(I/2) > COPY)

A12 = (EP(P, (1..N)), COPY = AO(P(I)), (A(II) = AO(P(II)), II .IN. ((1..N) .D. (I))))

A13 = (J > N)

A14 = (EP(P, (1..N)), ((if II = I then COPY else A(II)) = AO(P(II)), II .IN. (1..N)))

A15 = ((if K/2 = I then COPY else A(K/2)) $\geq$ (if K = I then COPY else A(K)), K .IN. (2*IO..N))

A16 = (J = N)

A17 = (A(I) $\geq$ A(J)

A18 = (I = J/2)

A19 = (2*I $\leq$ J)

A20 = (J $\leq$ N)

A21 = (2*I < N)

A22 = (A(J) = MAX(A(2*I), A(2*I+1)))

A23 = (2*I = N)

A24 = (A(J) = A(N))

A25 = (J < N)

A26 = (A(J+1) $\leq$ A(J))

A27 = (A(J+1) > A(J))

A28 = (A(I) $\geq$ A(J+1)

A29 = (I = (J+1)/2)

A30 = (2*I ≤ J+1)

A31 = (J+1 ≤ N)

A32 = (A(J+1) = MAX(A(2*I), A(2*I+1))

A33 = (A(J+1) = A(N))

A34 = (A(J) ≤ COPY)

A35 = (A(J) > COPY)

A36 = (J ≥ 2*IO)

A37 = ((if J/2 = I then A(J) else A(J/2)) > COPY)

A38 = ((if J/2 = I then A(J) else A(J/2)) ≥ A(J))

A39 = ((if K/2 = I then A(J) else A(K/2)) ≥ (if K = I then A(J) else A(K)), K .IN. (2*IO..N))

A40 = (EP(P, (1..N)), COPY = AO(P(J)), ((if II = I then A(J) else A(II)) = AO(P(II)), II .IN. ((1..N) .D. (J))))

(These are initial and final assertions of control paths in this program. The ones which are final assertions have been obtained by substitution.)

There are seven control paths. The first one starts at the beginning of the program; its hypotheses are the assertions A1, A2, A3, A4, A5, and A6. Its conclusions are:

((A2 and A5) or Z), where we shall not spell out Z because A2 and A5 are true anyway;

A3, which is preserved;

2*I = 2*I, which is obvious;

(EP(P, (1..N)), A(I) = A0(P(I)), (A(II) = A0(P(II)), II .IN. ((1..N) .D. (I)))), which reduces to A6;

and A1 and A4, which are preserved.

The second control path starts just following statement number 1 and proceeds immediately to statement number 4. Its hypotheses are ((A2 and A5) or (A7, A11, A8, and A9)), A3, A10, A12, A1, A4, and A13. Its conclusions are A14, which reduces to A12, and A15. If A2 and A5 are true, then A2 and A10 give (2*I0..N) = (J..N), and this is the null set by A13, so that there is nothing to prove in A15. Otherwise, we cannot have K/2 = I for K in (2*I0..N), because then K = 2*I or K = 2*I+1, and 2*I = J > N by A10 and A13. Hence A15 reduces to (A(I/2) ≥ COPY, (A(K/2) ≥ A(K), K .IN. ((2*I0..N) .D. (I)))), and this is implied by A11 and A9.

The third control path starts at the same point as the second, and proceeds immediately to statement number 3. Its hypotheses are ((A2 and A5) or (A7, A11, A8, and A9)), A3, A10, A12, A1, A4, and A16. Its conclusions are:

((A2 and A5) or (A7, A17, A11, A8, and A9)). Here if A2 and A5 are true initially, they are preserved. Otherwise A7, A11, A8, and A9 are preserved, and it only remains to prove A17. By A10, A17 becomes A(J/2) ≥ A(J), and this becomes A(N/2) ≥ A(N) by A16. But this is true by A9;

A18, which is true by A10;

A19, which is true by A10;

A20, which is true by A16;

((A21 and A22) or (A23 and A24)). A23 is true by A10 and A16, and A24 is true by A16;

and A12, A1, and A4, which are preserved.

The fourth control path starts at the same point as the second and third, and proceeds to statement 2 and then to statement 3 (without executing J = J + 1). Its hypotheses are ((A2 and A5) or (A7, A11, A8, and A9)), A3, A10, A12, A1, A4, A25, and A26. Its conclusions are:

((A2 and A5) or (A7, A17, A11, A8, and A9)), which is true just as in the preceding path, except that A16 no longer holds, so that A(J/2) $\geq$ A(J) does not become A(N/2) $\geq$ A(N), but A(J/2) $\geq$ A(J) is still true by A9, A25, A7, and A10;

A18 and A19, which are true by A10, just as before;

A20, which is true by A25;

((A21 and A22) or (A23 and A24)). A21 is true by A10 and A25. We have A(2*I+1) $\leq$ A(2*I) by A26 and A10, so that A22 reduces to A(J) = A(2*I), which is true by A10;

and A12, A1, and A4, which are preserved.

The fifth control path starts at the same point as the second, third, and fourth, and is the same as the fourth except that J = J + 1 is executed. Its hypotheses are ((A2 and A5) or (A7, A11, A8, and A9)), A3, A10, A12, A1, A4, A25, and A27. Its conclusions are:

((A2 and A5) or (A7, A17, A11, A8, and A9)), which is true just as in the preceding path;

A29, which is true by A10 (recalling that the division is performed without rounding);

A30, which is true by A10;

A31, which is true by A25;

((A21 and A32) or (A23 and A33)). A21 is true by A10 and A25. We have A(2*I+1) > A(2*I) by A27 and A10, so that A32 reduces to A(J+1) = A(2*I+1), which is true by A10;

and A12, A1, and A4, which are preserved.

The sixth control path starts at statement number 3 and goes immediately to statement number 4. Its hypotheses are ((A2 and A5) or (A7, A17, A11, A8 and A9)), A18, A19, A20, ((A21 and A22) or (A23 and A24)), A12, A1, A4, and A34. Its conclusions are A14, which reduces to A12, and A15. If A22 is true, then COPY $\geq$ A(2*I) and COPY $\geq$ A(2*I+1) by A22 and A34; otherwise, J = N by A19, A20, and A23, so that COPY $\geq$ A(2*I) by A34 and A23, and 2*I+1 > N. If A2 is true, then I is not in (2*I0..N), and otherwise A(I/2) > COPY by A11. This takes care of the cases K/2 = I and K = I in A15, and the other cases all follow from A5 or from A9, one or the other of which must be true.

Finally, the seventh control path starts at statement number 3 and goes back to statement number 1. Its hypotheses are ((A2 and A5) or (A7, A17, A11, A8, and A9)), A18, A19, A20, ((A21 and A22) or (A23 and A24)), A12, A1, A4, and A35. Its conclusions are:

(Z or (A36, A37, A38, and A39)), where we shall not spell out Z because the other alternative will always hold. In fact, A36 follows from A19 together with either A2 or A7; A37 reduces to A35 by A18; and A38 becomes A(J) $\geq$ A(J) (which is obvious), again by A18. If A22 is true, then A(J) $\geq$ A(2*I) and A(J) $\geq$ A(2*I+1) by A22; otherwise, A(J) $\geq$ A(2*I) by A23 and A24, and 2*I+1 > N. If A2 is true, then I is not in (2*I0..N), and otherwise A(I/2) $\geq$ A(J) by A8 and A17. This takes care of the cases K/2 = I and K =

I in A39, and the other cases all follow from A5 or from A9;

A20, which is preserved;

2*J = 2*J, which is obvious;

A40, which is derived from A12 by taking P in A40 (call it PP) to be PP(I) = P(J), PP(J) = P(I), and (PP(K) = P(K), K .IN. ((1..N) .D. (I, J))). We have COPY = AO(P(I)) = AO(PP(J)), A(J) = AO(P(J)) = AO(PP(II)), and (A(K) = AO(P(K)) = AO(PP(K)), K .IN. ((1..N) .D. (I, J))), from A12, which implies A40;

and A1 and A4, which are preserved.

In all of these paths except the third, there are restrictions due to the use of subscripted variables. In the first path, these restrictions follow from assertions A1, A2, A3, and A4. In all the other paths, we make use of the fact that either A2 or A7 must be true. In the second path, A1, A3, and A4 are also needed; in the fourth and fifth paths, A25, A10, and A1; and in the sixth and seventh paths (which use both A(I) and A(J)), assertions A1, A19, A20, and A4. This completes the proof of partial correctness.

The termination of this program follows from the fact that N-J is a loop expression when loops are measured from statement number 3. Every loop must pass through statement number 3; the expression N-J is bounded at that statement by assertion A20; and its value must decrease every time we go around the loop, since N does not change while J can only strictly increase -- at statement number 1, which must be executed, and at J = J + 1. In fact, at statement number 1, J is doubled, and it is therefore increased (since $J = I \geq I0 > 0$ by A1 and either A2 or A7). Thus our program is correct.

The SIFTUP program is short (only ten executable statements), but of high complexity. It was obtained only after considerable thought, and the complexity of its proof should not be surprising.

10-5 Merging and Exchanging

A different kind of permutation is used in proving the correctness of a program which merges two sorted arrays A and B to produce a single sorted array C. Three pointers, which we shall call I, J, and K, are kept in such a routine. The elements A(1) through A(I-1) and the elements B(1) through B(J-1) have presumably already been merged to form the elements C(1) through C(K), and K = (I-1) + (J-1), or I+J-2. Our permutation P will be on the set (1..K), and, for each element X of the set (1..I-1) we will have A(X) = C(P(X)); that is, each element of A, below the I-th element, is contained in C somewhere below the K-th element. Each element of B below the J-th element is also contained in C, and these J-1 elements of B are keyed to the permutation P by writing B(X-(I-1)) for X in (I..K). Thus our assertion is

$$EP(P, (1..K)), (A(X) = C(P(X)), X .IN. (1..I-1)),$$
$$(B(X-(I-1)) = C(P(X)), X .IN. (I..K)), K = I+J-2$$

We shall denote this assertion by MERGE(A, B, C, I, J, K).

The following program merges the sorted arrays A and B to form the sorted array C, in the manner suggested above. The length of A is M and the length of B is N. In order to determine the K-th element of the new array, we look at the I-th element of A and the J-th element of B; the smaller of these two, in the given order, becomes the new K-th element of C. In addition, if the smaller element came from A, then I is increased by 1; otherwise, J is increased by 1. As soon as we are finished with A or with B, we copy the rest of B (or, respectively, A) into C. The program may be written as follows:

```
      DIMENSION A(m), B(n), C(p)
C  ASC(A, 1, M), ASC(B, 1, N), M > 0, M ≤ m, N > 0, N ≤ n, M+N ≤ p
      I = 1
      J = 1
      K = 0
C  1 ≤ I, I ≤ M, M ≤ m, 1 ≤ J, J ≤ N, N ≤ n, M+N ≤ p,
C  ASC(A, 1, M), ASC(B, 1, N), MERGE(A, B, C, I, J, K),
C  (K = 0 .OR. (K > 0, C(K) ≤ A(I), C(K) ≤ B(J), ASC(C, 1, K)))
1     K = K + 1
      IF (A(I) < B(J)) GO TO 3
      C(K) = B(J)
      J = J + 1
      IF (J .LE. N) GO TO 1
C  1 ≤ I, I ≤ M, M ≤ m, ASC(A, 1, M), J = N + 1, N ≤ n, M+N ≤ p
C  MERGE(A, B, C, I, J, K), K > 0, C(K) ≤ A(I), ASC(C, 1, K)
2     K = K + 1
      C(K) = A(I)
      I = I + 1
      IF (I .LE. M) GO TO 2
      GO TO 5
3     C(K) = A(I)
      I = I + 1
      IF (I .LE. M) GO TO 1
C  1 ≤ J, J ≤ N, N ≤ n, ASC(B, 1, N), I = M + 1, M ≤ m, M+N ≤ p
C  MERGE(A, B, C, I, J, K), K > 0, C(K) ≤ B(J), ASC(C, 1, K)
4     K = K + 1
      C(K) = B(J)
      J = J + 1
      IF (J .LE. N) GO TO 4
C  MERGE(A, B, C, M, N, M+N), ASC(C, 1, M+N)
5     CONTINUE
```

As the value of K increases, the permutation whose existence is implied by MERGE(A, B, C, I, J, K) expands its domain and range and takes on new values, although its old values remain the same. To see how this happens, let us examine a typical control path in the above routine:

```
C  1 ≤ I, I ≤ M, M ≤ m, 1 ≤ J, J ≤ N, N ≤ n, M+N ≤ p,
C  ASC(A, 1, M), ASC(B, 1, N), MERGE(A, B, C, I, J, K),
C  (K = 0 .OR. (K > 0, C(K) ≤ A(I), C(K) ≤ B(J), ASC(C, 1, K)))
       K = K + 1
       [1 ≤ I, I ≤ m, 1 ≤ J, J ≤ n]
       (A(I) ≥ B(J))
       [1 ≤ J, J ≤ n, 1 ≤ K, K ≤ p]
       C(K) = B(J)
       J = J + 1
       (J ≤ N)
C  1 ≤ I, I ≤ M, M ≤ m, 1 ≤ J, J ≤ N, N ≤ n, M+N ≤ p,
C  ASC(A, 1, M), ASC(B, 1, N), MERGE(A, B, C, I, J, K),
C  (K = 0 .OR. (K > 0, C(K) ≤ A(I), C(K) ≤ B(J), ASC(C, 1, K)))
```

This control path involves statement number 1 and the four statements which immediately follow it. The restrictions in the path all follow from the initial assertion (remembering that MERGE(A, B, C, I, J, K) implies K = I+J-2). Back substitution yields the following conclusion (omitting the case K = 0, which never happens):

```
1 ≤ I, I ≤ M, M ≤ m, 1 ≤ J+1, J+1 ≤ N, N ≤ n,
M+N ≤ p, ASC(A, 1, M), ASC(B, 1, N), EP(P, (1..K+1)),
(A(X) = (if P(X) = K+1 then B(J) else C(P(X))), X .IN.
(1..I-1)), (B(X-(I-1)) = (if P(X) = K+1 then B(J) else
C(P(X))), X .IN. (I..K+1)), K+1 = I+J+1-2,
```

$$K+1 > 0,\ B(J) \leq A(I),\ B(J) \leq B(J+1),$$
$$(K = 1\ .OR.\ (ASC(C, 1, K),\ C(K) \leq B(J)))$$

to be derived from the initial assertion and the conditions $A(I) \geq B(J)$ and $J+1 \leq N$ (using the modified value of J) occurring in the path. The last condition arises from writing ASC(C, 1, K) in the final assertion as ASC(C, 1, K-1) and $C(K-1) \leq C(K)$, and then applying back substitution, remembering to separate out the case $K = 1$.

Of all these final assertions, $1 \leq I$, $I \leq M$, $M \leq \underline{m}$, $N \leq \underline{n}$, $M+N \leq \underline{p}$, ASC(A, 1, M), ASC(B, 1, N), and (K = 1 .OR. (ASC(C, 1, K), $C(K) \leq B(J)$)) are clearly preserved, while $B(J) \leq A(I)$ and $J+1 \leq N$ are conditions found in the path. Also, $1 \leq J+1$ follows from $1 \leq J$; $K+1 = I+J+1-2$ follows from $K = I+J-2$; $K+1 > 0$ follows from either $K = 0$ or $K > 0$, one of which must be true; and $B(J) \leq B(J+1)$ follows from ASC(B, 1, N), $1 \leq J$, and $J+1 \leq N$. This leaves only the assertion about the existence of a permutation with certain properties. If the assertion in the hypothesis (on the set (1..K)) is denoted by P, and the assertion in the conclusion (on the set (1..K+1)) by PP, then we may determine PP in terms of P as follows:

$$PP(K+1) = K+1,\ (PP(KK) = P(KK),\ KK\ .IN.\ (1..K))$$

The conclusion about the permutation PP then reduces to

$$(A(X) = C(P(X)),\ X\ .IN.\ (1..I-1)),\ (B(X-(I-1)) = C(P(X)),$$
$$X\ .IN.\ (I..K)),\ B(K+1-(I-1)) = B(J)$$

and this reduces to the initial assertion. In particular, $K+1-(I-1) = J$, because $K = I+J-2$ initially.

Another sorting-related process is that of exchanging. We shall use this term, as it is often used, to refer to a process whereby two pointers, initialized to point to the beginning and the end of a table respectively, move toward each other, ultimately meeting in the middle of the table, and, every so often, the elements of the table to which they point are exchanged. We shall present the exchange process which is used as the first stage in the radix exchange sort, another fast in-place sorting method.

The effect of our exchange process will be to get all negative elements of the array A in front of all positive elements. Successive stages of the program then sort the negative elements and the positive elements separately. (The sort actually acts on a single bit of the sort key; the first stage acts on the leftmost bit, which is taken to be the sign bit. If unsigned quantities are being sorted, this first stage of the sort would be different.) The program is as follows:

```
      DIMENSION A(n)
C  N ≥ 1, N ≤ n
      I = 1
      J = N+1
C  1 ≤ I, I < J, J ≤ N+1, N ≤ n, (A(K) <,0, K .IN.
C  (1..I-1)), (A(K) ≥ 0, K .IN. (J..N))
1     IF (A(I) .GE. 0) GO TO 3
2     I = I + 1
      IF (I .NE. J) GO TO 1
      GO TO 4
C  1 ≤ I, I < J, J ≤ N+1, N ≤ n, (A(K) <,0, K .IN.
C  (1..I-1)), (A(K) ≥ 0, K .IN. (J..N)), A(I) ≥ 0
```

```
3       J = J - 1
        IF (I .EQ. J) GO TO 4
        IF (A(J) .GE. 0) GO TO 3
        T = A(I)
        A(I) = A(J)
        A(J) = T
        GO TO 2
C   (A(K) < 0, K .IN. (1..I-1)), (A(K) ≥ 0, K .IN. (I..N))
4       CONTINUE
```

Note the slightly unsymmetrical treatment of I and J. This is necessary because, among other things, when I and J are equal, A(I) must be either the last negative element or (as in this version of the algorithm) the first positive element of A.

Let us abbreviate our assertions here as in the preceding section:

A1 = (N ≥ 1)

A2 = (N ≤ n)

A3 = (1 < N+1)

A4 = (1 ≤ I)

A5 = (I < J)

A6 = (J ≤ N+1)

A7 = (A(K) < 0, K .IN. (1..I-1))

A8 = (A(K) ≥ 0, K .IN. (J..N))

A9 = (A(I) ≥ 0)

A10 = (A(I) < 0)

A11 = (I + 1 ≠ J)

A12 = (1 ≤ I + 1)

A13 = (I + 1 < J)

A14 = (A(K) < 0, K .IN. (1..I))

A15 = (I + 1 = J)

A16 = (A(K) ≥ 0, K .IN. (I+1..N))

A17 = (A(K) ≥ 0, K .IN. (I..N))

A18 = (A(J-1) ≥ 0)

A19 = (J - 1 ≤ N + 1)

A20 = (A(K) ≥ 0, K .IN. (J-1, N))

A21 = (A(J-1) < 0)

A22 = (I + 1 ≠ J - 1)

A23 = ((if K = I then A(J-1) else if K = J - 1 then A(I) else A(K)) < 0, K .IN. (1..I))

A24 = ((if K = I then A(J-1) else if K = J - 1 then A(I) else A(K)) ≥ 0, K .IN. (J-1..N))

A25 = (I + 1 = J - 1)

A26 = ((if K = I then A(J-1) else if K = J - 1 then A(I) else A(K)) ≥ 0, K .IN. (I+1..N))

For convenience, we list only the hypotheses and the conclusions in the verification condition of each control path; showing that the conclusions actually follow from the hypotheses will be left to the reader.

Control path 1 starts at the beginning of the program and ends at statement number 1. Hypotheses are A1 and A2. Conclusions are A3, A2, and the tautologies 1 ≤ 1, N+1 ≤ N+1, (A(K) < 0, K .IN. (1..0)), and (A(K) ≥ 0, K .IN. (N+1..N)).

Control path 2 starts at statement number 1 and ends at statement number 3. Hypotheses are A4, A5, A6, A2, A7, A8, and A9. Conclusions are the same as the hypotheses.

Control path 3 starts and ends at statement number 1. Hypotheses are A4, A5, A6, A2, A7, A8, A10, and A11. Conclusions

are A12, A13, A6, A2, A14, and A8.

Control path 4 starts at statement number 1 and ends at statement number 4. Hypotheses are A4, A5, A6, A2, A7, A8, A10, and A15. Conclusions are A14 and A16.

Control path 5 starts at statement number 3 and goes immediately to statement number 4. Hypotheses are A4, A5, A6, A2, A7, A8, A9, and A15. Conclusions are A7 and A17.

Control path 6 starts and ends at statement number 3. Hypotheses are A4, A5, A6, A2, A7, A8, A9, A11, and A18. Conclusions are A4, A13, A19, A2, A7, A20, and A9.

Control path 7 starts at statement number 3, passes through statement number 2, and ends at statement number 1. Hypotheses are A4, A5, A6, A2, A7, A8, A9, A11, A21, and A22. Conclusions are A12, A13, A6, A2, A23, and A24.

Finally, control path 8 starts at statement number 3, passes through statement number 2, and ends at statement number 4. Hypotheses are A4, A5, A6, A2, A7, A8, A9, A11, A21, and A25. Conclusions are A23 and A26.

There are restricted commands in control paths 2, 3, 4, 6, 7, and 8. In each case, the restrictions follow from assertions A4, A5, A6, and A2. Thus the program is partially correct.

To prove that the program is correct, we notice that J-I is a loop expression. This program is noteworthy in that, although there are several loops in it, there is one loop expression for all of them.

NOTES

The remarks on timing in section 10-1 are related to the idea of termination, although not in a straightforward manner. That is, if we prove the partial correctness of a program, using a final asssertion which says that the program terminates in e steps (where e is some expression in the variables of the program), this still does not prove that the program terminates; it only proves that if the program terminates, it does so in e steps. Timing proofs may, however, be modified so as to constitute valid proofs of termination, as is done in section 3.8 of [Good 70].

The binary search program of section 10-2 was used in [Floyd 71] as an example illustrating the operation of an interactive program verifier; Deutsch has written a verifier which proves the correctness of this program [Deutsch 73].

The fact that a sorting program is not proved correct until it is proved that the sorted values are a rearrangement of the original values was noted in [London 70] and in [Hoare 71b]. The use of an array of initial values of the variables to be sorted, not appearing explicitly in the original program to be proved correct, appears in [London 70]; the theorem concerning the preservation of the PERM assertion when an interchange is performed was first noted (although not formally proved) in [Hoare 71b].

The tree sort of section 10-4 was originally written by Floyd [Floyd 64] and proved correct by London [London 70]. Exercise 10, below, is based on [Hoare 71b].

EXERCISES

1. Complete the proof of correctness of the linear search routine given at the beginning of section 10-1.

2. Insert into this linear search routine the assertions concerning its timing which are given near the end of section 10-1, and prove the correctness of the resulting program, thus proving that this routine is linear in N.

3. Prove, using the methods of section 10-1, that the binary search routine of section 10-2 is logarithmic. (In other words, show that the total number of steps taken by this routine is bounded from above by an expression of the form $\underline{a} \log N + \underline{b}$, and from below by another such expression.)

4. Prove the correctness of the hash table routine of section 10-2.

5. Prove the correctness of the following program (commonly known as a "bubble sort") using the methods of section 10-3:

```
      DIMENSION A(n)
C  N .GE. 2, N .LE. n, (A(K) = AO(K), K .IN. (1..N))
      J = 1
C  1 .LE. J, J .LT. N, N .LE. n, PERM(A, AO, N), ASC(A, 1, J)
1     I = J
      Q = A(I+1)
C  1 .LE. I, I .LE. J, J .LT. N, N .LE. n,
C  Q .LE. A(I+1), ASC(A, 1, J), (I = J .OR. A(J) .LE. A(J+1)),
C  EP(P, (1..N)), Q = AO(P(I+1)),
C  (A(K) = AO(P(K)), K .IN. ((1..N) .D. (I+1)))
2     IF (Q .GE. A(I)) GO TO 3
```

```
          A(I+1) = A(I)
          I = I - 1
          IF (I .NE. 0) GO TO 2
C  0 ≤ I, I ≤ J, 1 ≤ J, J < N, N ≤ N, (I = 0 .OR. Q ≥ A(I)),
C  Q ≤ A(I+1), ASC(A, 1, J), (I = J .OR. A(J) ≤ A(J+1)),
C  EP(P, (1..N)), Q = AO(P(I+1)),
C  (A(K) = AO(P(K)), K .IN. ((1..N) .D. (I+1)))
3         A(I+1) = Q
          J = J + 1
          IF (J .NE. N) GO TO 1
C  PERM(A, AO, N), ASC(A, 1, N)
          CONTINUE
```

(Note that it is not necessary to use unsortedness to prove that this program terminates.)

6. Find lower and upper bounds for the number of steps (executable statements) taken by the program above, and prove that these are in fact the bounds. (Hint: The program is fastest when the given array is already sorted; it is slowest when the array is sorted in reverse order.)

7. The loop in the SIFTUP program of section 10-4 contains an unconditional GO TO statement. Such a loop can usually be shortened; in fact, we could rewrite this program as

```
          SUBROUTINE SIFTUP(I, N)
          COPY = A(I)
          GO TO 7
5         IF (A(J+1) .LE. A(J)) GO TO 6
```

```
      J = J + 1
6     IF (A(J) .IE. COPY) GO TO 8
      A(I) = A(J)
      I = J
7     J = 2*I
      IF (N-J) 8, 6, 5
8     A(I) = COPY
      RETURN
      END
```

Produce a proof of the correctness of this program by modifying, as little as possible, the proof of correctness of the original program.

8. Prove that the modified program (in the preceding problem) is faster than the unmodified program, by determining the number of statements executed in each program according to the method of section 10-1.

9. Complete the proof of the merge program of section 10-5.

10. The exchange process of section 10-5 may be expanded into a program which finds the F-th element in order in an (originally unsorted) array, and simultaneously performs interchanges in such a way that A(F) > A(G) implies F > G and A(F) < A(G) implies F < G, $1 \leq G \leq N$, as in the program below. For convenience, let

A1 = (A(P) .LE. A(Q), P .IN. (1..M-1), Q .IN. (M..N))

A2 = (A(P) .LE. A(Q), P .IN. (1..NN-1), Q .IN. (NN..N))

A3 = (A(P) .LE. R, P .IN. (1..I-1))

A4 = (A(Q) .GE. R, Q .IN. (J+1..N))

Then the program is as follows. (Note: Not all of the necessary intermediate assertions have been given.)

```
      SUBROUTINE FIND(A, N, F)
      DIMENSION A(N)
      INTEGER F, R, W
C  1 ≤ F, F ≤ N, (A(K) = AO(K), K .IN. (1..N))
      M = 1
      NN = N
C  PERM(A, AO, N), M ≤ F, F ≤ NN, A1, A2
1     IF (M .GE. NN) GO TO 8
      R = A(F)
      I = M
      J = NN
C  PERM(A, AO, N), M ≤ F, F ≤ NN, M ≤ I, J ≤ N, A1, A2, A3, A4
2     IF (I .GT. J) GO TO 6
C  PERM(A, AO, N), M ≤ F, F ≤ NN, M ≤ I, J ≤ N, A1, A2, A3, A4
3     IF (A(I) .GE. R) GO TO 4
      I = I + 1
      GO TO 3
C  PERM(A, AO, N), M ≤ F, F ≤ NN, M ≤ I, J ≤ N, A1, A2, A3, A4
4     IF (R .GE. A(J)) GO TO 5
      J = J - 1
      GO TO 4
5     IF (I .GT. J) GO TO 6
      W = A(I)
      A(I) = A(J)
      A(J) = W
      I = I + 1
```

```
      J = J - 1
      GO TO 2
6     IF (F .GT. J) GO TO 7
      NN = J
      GO TO 1
7     IF (I .GT. F) GO TO 8
      M = I
      GO TO 1
C  PERM(A, AO, N), (A(P) ≤ A(F), P .IN. (1..F)),
C  (A(F) ≤ A(Q), Q .IN. (F..N))
8     CONTINUE
```

Prove its correctness.

REFERENCES

Bjorner 70 — Bjorner, D., Flowchart machines, BIT 10, 4 (1970), pp. 415-442.

Cooper 69 — Cooper, D. C., Program scheme equivalences and second-order logic, Machine Intelligence 4 (1969), pp. 3-15.

Deutsch 73 — Deutsch, L. P., An interactive program verifier, Ph. D. Thesis, Department of Computer Science, University of California, Berkeley, May 1973.

Floyd 64 — Floyd, R. W., Algorithm 245, TREESORT III, Communications of the ACM, 7, 12 (December 1964), p. 701.

Floyd 67 — Floyd, R. W., Assigning meanings to programs, Proc. Symp. Applied Math. 19 (Mathematical Aspects of Computer Science), American Mathematical Society, Providence, R. I., 1967, pp. 19-32.

Floyd 71 — Floyd, R. W., Toward interactive design of correct programs, Information Processing 71, Proceedings of IFIP Congress 71 (Ljubljana), C. V. Freiman, ed., North-Holland, Amsterdam, 1972, pp. 7-11.

Goldstine and von Neumann 47 — Goldstine, H. H., and von Neumann, J., Planning and coding problems for an electronic computing instrument, reprinted in The Collected Works of John von Neumann, vol. 5, pp. 80-235, Pergamon Press, 1963.

Good 70 — Good, D. I., Toward a man-machine system for proving program correctness, Ph. D. Thesis, Computer Science Department, University of Wisconsin, 1970.

Good and London 68 — Good, D. I., and London, R. L., Interval arithmetic for the Burroughs B5500: Four ALGOL procedures and proofs of their correctness, Computer Sciences Technical Report No. 26, University of Wisconsin, 1968.

Good and London 70 — Good, D. I., and London, R. L., Computer interval arithmetic: definition and proof of correct implementation, Journal of the ACM 17, 4 (October 1970), pp. 603-612.

Hoare 69 — Hoare, C. A. R., An axiomatic basis for computer programming, Communications of the ACM 12, 10 (1969), pp. 576-580, 583.

Hoare 71a — Hoare, C. A. R., Procedures and parameters: an axiomatic approach, in Engeler, E., ed., Lecture Notes in Mathematics 188: Symposium on Semantics of Algorithmic Languages, Springer-Verlag, Berlin-Heidelberg-New York, 1971, pp. 102-116.

Hoare 71b — Hoare, C. A. R., Proof of a program: FIND, Communications of the ACM, 14, 1 (January 1971), pp. 39-45.

Hoare and Wirth 72 — Hoare, C. A. R., and Wirth, N., An axiomatic definition of the programming language PASCAL, Berichte der Fachgruppe Computer-Wissenschaften, Nr. 6, ETH, Zurich, November 1972.

Karp 60 — Karp, R. M., A note on the application of graph theory to digital computer programming, Inf. and Control 3, 2 (1960), pp. 179-190.

King 69 — King, J. C., A program verifier, Ph. D. Thesis, Department of Computer Science, Carnegie-Mellon University, 1969.

King 71 — King, J. C., Proving programs to be correct, IEEE Transactions on Computers C20, 11 (1971), pp. 1331-1336.

Knuth 68 — Knuth, D. E., The Art Of Computer Programming, Vol. 1: Fundamental Algorithms, Addison-Wesley, Reading, Mass., 1968.

Lauer 71 — Lauer, P., Consistent formal theories of the semantics of programming languages, Technical Report TR25.088, IBM Laboratory Vienna, 1968.

London 70 — London, R. L., Proof of algorithms -- a new kind of certification (Certification of Algorithm 245, TREESORT III), Communications of the ACM 13, 6 (June 1970), pp. 371-373.

London 71 — London, R. L., Experience with inductive assertions for proving programs correct, in Engeler, E., ed., Lecture Notes in Mathematics 188: Symposium on Semantics of Algorithmic Languages, Springer-Verlag, Berlin-Heidelberg-New York, 1971, pp. 236-251.

London 72 — London, R. L., The current state of proving programs correct, Proc. ACM Annual Conf., August 1972, pp. 39-46.

Manna 68 — Manna, Z., Termination of algorithms, Ph. D. Thesis, Department of Computer Science, Carnegie-Mellon University, 1968.

Manna 69 — Manna, Z., The correctness of programs, J. Computer and Systems Sci. 3, 2 (1969), pp. 119-127.

Manna 70 — Manna, Z., Termination of programs represented as interpreted graphs, Proc. 1970 Spring Joint Computer Conference, pp. 83-89.

Manna and Pnueli 70 — Manna, Z., and Pnueli, A., Formalization of properties of functional programs, Journal of the ACM, 17, 3 (July 1970), pp. 555-569.

Maurer 72 — Maurer, W. D., A semantic extension of BNF, International Journal of Computer Mathematics, Sept. 1972.

Naur 66 — Naur, P., Proof of algorithms by general snapshots, BIT 6, 4 (1966), pp. 310-316.

Painter 67 — Painter, J. A., Semantic correctness of a compiler for an ALGOL-like language, Ph. D. Thesis, Computer Science Department, Stanford University, 1967.

Redish 71 — Redish, K. A., Comment on London's certification of Algorithm 245 (with reply by London), Communications of the ACM, 14, 1 (January 1971), pp. 50-51.

Turing 50 — Turing, A., Checking a large routine, in Report of a Conference on High-Speed Automatic Calculating Machines, University Mathematical Laboratory, Cambridge, Jan. 1950, pp. 67-69.

Wilkinson 63 — Wilkinson, J. H., Rounding Errors in Algebraic Processes, Prentice-Hall, 1963.

INDEX