THEORY AND PRACTICE OF ALGORITHM VERIFICATION

by

W. D. Maurer

Memorandum No. ERL-M315

1 January 1972

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Research sponsored by the National Science Foundation, under Grants GJ-821 and GJ-31612.

TABLE OF CONTENTS

1 CORRECTNESS OF PROGRAMS: BASIC PRINCIPLES

This book is an introduction to programming as a science. What distinguishes a science from an art is the fact that in a science we know that certain things are true, instead of merely feeling that they are true.

Debugging, as it is currently practiced, is an example of programming as an art. We write a program, run it a few times, find mistakes, correct them, run it again, and finally emerge with a "working version" of the program. This may still not be correct for all possible input; it is only correct for the input we have tested. If we are more sophisticated, we submit our program to a specialist in programming quality control, who subjects it to as thorough a battery of tests as he can devise. At the end of this process, the program may still have bugs in it; all that is different from before is that the programming manager is now willing to pay for any damage caused in the future by any bugs which remain.

The first chapter of this book will be devoted to the basic principles of proving mathematically that there are no bugs in a given program. The principles, however, will be stated intuitively, in programming terms, without the use of formal mathematical models; these will be deferred until Chapter 3.

1-1 Straight-Line Programs

What does it mean for a program to be correct? To illustrate what should be included in a proper definition of correctness, let us consider the following simple program:

```
B = A x A
C = A x B
B = B x C
B = B X B
B = B X B
B = B X C
```

This program calculates the 23rd power of A. The results at each stage are the square, cube, fifth power, then 10th, 20th, and 23rd powers.

If we are to call this a correct program, it will not be enough to say that it correctly calculates the 23rd power of A. In particular, if we change the last statement to C = B X C, then the new program still calculates the 23rd power of A, and yet the two programs are clearly not equivalent. If this program were part of a larger one, and its result were used later on, we would have to reference C, rather than B. We must, therefore, specify which variable receives the new value. Thus for the program above we would say: After this program is finished, B is equal to the 23rd power of A.

The above statement, however, is still not enough to specify what this program does. In particular, the program

$$A = 1$$
$$B = 1$$

also has the property that, after it is finished, B is equal to the 23rd power of A. The difference between the two programs, of course, is that in the new program the value of A may be changed, whereas it may not be in the original program. If we specify that the value of A is not changed, then the statement that B is equal to the 23rd power of A is enough to specify B completely.

What if we change the last statement of our original program to $A = B \times C$? Now the value of A will be equal to the 23rd power of what it was at the beginning. But if this program is embedded in a larger one, the beginning of the larger program might not be the same as the beginning of this one. The initial value of A must be associated with the beginning of <u>this</u> program, and not some larger one. Thus we might say: <u>If A = X at the beginning of this program, then A will be equal to the 23rd power of X at the end of it</u>. Implicit in this statement is the fact that the value of X is not changed by the program; otherwise, we would encounter the same problem we did before. The device of including a new variable, such as X here, in order to state the conditions of correctness of a program, may be used whenever the purpose of a program is to change a variable rather than to set it to some function of other variables.

A condition such as $A = X$ or $A = X^{23}$ in these examples is known as an <u>assertion</u>. In proving the correctness of a program, there will always be an assertion at the end of the program, and sometimes there will also be one at the beginning. A condition which must hold in order for the program to work properly may be stated as an <u>initial assertion</u>, or an assertion at the beginning.

Thus, for example, in a program to set B equal to the square root of A, the initial assertion would state that A is greater than or equal to zero. The <u>final assertion</u> is then the statement that B is equal to the square root of A. We have identified the following components in a statement of correctness of a program:

(1) an initial assertion;

(2) a final assertion;

(3) a statement of what variables may have their values changed by the program;

(4) a statement that if the initial assertion is true at the beginning of the program, then the final assertion will be true at the end of it.

Are these enough to specify completely what a program does? In general, the answer is still no. A program may get into an endless loop under certain conditions; or it may end in more than one place, thus making it necessary to have more than one final assertion. Let us, however, confine our attention for the moment to <u>straight-line programs</u>, in which the above behavior cannot occur. A straight-line program is one which has no branches at all, such as our program to calculate 23rd powers. We will use this program to show intuitively that in this case the answer to our question is yes.

The program uses three variables, A, B, and C. A complete specification of what this program does will give the new values of A, B, and C after the program has been executed, as functions of their original values. For the moment, let A, B, and C stand for the new values of the variables, and let <u>a</u>, <u>b</u>, and <u>c</u> stand for their original values. Then the equations

$$A = \underline{a}$$
$$B = \underline{a}^{23}$$
$$C = \underline{a}^{3}$$

provide a complete specification of the effect of this program. Normally, the second of these is the only one we will be interested in, although we might need the third, if, for example, this program is embedded in a larger one which uses the cube of $\underline{a}$. If we happen to notice that this is an intermediate result which remains at the end of this program as the final value of C, we can avoid recalculating the cube of $\underline{a}$ if this result is needed.

Let us now regard $\underline{a}$, $\underline{b}$, and $\underline{c}$, not merely as symbols denoting A, B, and C respectively at an earlier point in the computation, but rather as separate variables, much like the variable X in the previous example. We may now construct the initial assertion

$$A = \underline{a},\ B = \underline{b},\ C = \underline{c}$$

and the final assertion

$$A = \underline{a},\ B = \underline{a}^{23},\ C = \underline{a}^{3}$$

Furthermore, B and C are the only variables whose values may be changed by this program. These statements are equivalent to the complete specification given above, and for the same sort of reasons as before; since $\underline{a}$ cannot change, the final assertion defines the values of A, B, and C uniquely in terms of initial values, even though this assertion refers directly only to the final values of A, B, C, $\underline{a}$, $\underline{b}$, and $\underline{c}$.

We have thus shown intuitively that defining the correctness of a straight-line program in terms of initial and final assertions allows the effect of that program to be specified completely. In proving the correctness of a program, however, we are not always interested in a <u>complete</u> specification. We have seen that it is enough, in our first example program, to specify that B is set equal to the 23rd power of A, provided we also know that the value of A cannot change. However, the value of A cannot change because no individual statement of this program can change the value of A; in general, whether or not a program has branches in it, the only variables which can be affected by the program as a whole are those which are affected by individual statements in it.

We shall, therefore, continue to define correctness of a straight-line program in terms of initial and final assertions alone. A complete definition of correctness for straight-line programs may be stated as follows:

<u>Given a straight-line program, an initial assertion about the variables of that program, and a final assertion about these variables, the program is correct (with respect to these two assertions) if, whenever it is started with the initial assertion valid, it ends with the final assertion valid.</u>

We will often abbreviate and say simply that "a program is correct," but we should note carefully that, strictly speaking, a program is always said to be correct <u>with respect to</u> something. Also, whenever it appears that we have no initial assertion for a program, it is understood that the initial assertion is the "null assertion," which is always true. When two or more assertions are connected by commas, "and" is understood; thus, for example, "A = <u>a</u>, B = <u>b</u>, C = <u>c</u>" denotes the single assertion "A = <u>a</u> and B = <u>b</u> and C = <u>c</u>."

1-2 Intermediate Assertions

We shall now prove the correctness of our straight-line program to calculate the 23rd power of A, in its original form. Our final assertion is $B = A^{23}$; there is no initial assertion. In addition, B and C are the only variables whose values may be changed by the program.

Before we proceed to the actual proof, let us consider certain objections which might be made on intuitive grounds to the statement that this program is correct. The first sort of objection concerns itself with the representation of variables within a computer. For example, what if the 23rd power of A is too big a number to fit into a word on the computer being used? In this case, in order for the program to be correct, we must specify that the value of A is small enough so that this will not happen. This then becomes part of our initial assertion. Similarly, if A is a floating point number, there will probably be roundoff error in the calculations, and the answer will not be precisely equal to the 23rd power of the given value of A. In such a case, all of our quantities must be asserted to be subject to roundoff error. Thus our initial assertion might say that A is equal to $X \pm e$, for some error term e. Our final assertion then says that B is now equal to $Y \pm e'$, where Y is the 23rd power of X and the error term e' depends on e, and, in general, on X as well.

It is evident from the above that programs involving floating point quantities may be proved correct with respect to the proper assertions. The formulation of these assertions requires special techniques, which are the province of numerical analysis and are beyond the scope of this book. In the future, we shall always assume, when dealing with integers and real numbers, that these are arbitrary integers and real numbers, not bound by any restrictions as to size or significance. It is worth repeating that these assumptions are made solely in the interests of simplicity; there is nothing to prevent us from dropping them in an expanded theory.

The second sort of objection to the statements which we have made about our example program concerns the variables whose values may be changed. As we saw in the preceding section, we need only look at each statement of the program separately in order to prove, in this case, that B and C are the only such variables. We would infer this from the fact that B and C are the only variables occurring on the left side of the equal sign in any statement in the program. The trouble is that this inference depends on the fact that our statements have no side effects. In some languages, such as ALGOL, any arithmetic statement may have side effects; thus A or B or C might be functions with no parameters. Therefore, this possibility must be explicitly or implicitly excluded as part of our proof of correctness.

Even when side effects are present, it is usually not too difficult to verify the statement that the side effects do not interfere with the variables whose values are being calculated. A side effect is always the effect of some subroutine, or some function coded as a subroutine; any variable whose value is changed by the side effect has its value changed by some statement appearing in the definition of the corresponding subroutine. This statement may, in turn, have a side effect, and the analysis may be carried through several levels, but it never leads to circular arguments, even in the case of recursive side effects. This is because all the correctness statements which we make about programs hold only for programs which terminate in a finite number of steps. If a program terminates in n steps, there can be no more than n recursion levels in the running of that program.

Having met these objections, let us return to the problem of proving that our final assertion will be valid after the given program has been run. One way to do this is to show exactly what goes on at each stage, as follows:

```
          B = A x A
    * B = A^2
          C = A x B
    * B = A^2, C = A^3
          B = B x C
    * B = A^5, C = A^3
          B = B x B
    * B = A^10, C = A^3
          B = B x B
    * B = A^20, C = A^3
          B = B x C
    * B = A^23
```

Instead of two assertions, we now have seven, of which the first is left out since it is always true. The five new assertions may be called <u>intermediate assertions</u>. Each statement in the program now has an assertion before it and after it, and may thus be regarded as a one-statement "program" whose correctness is being proved. Let us write, for example, the third statement of the program above, with its "initial assertion" and "final assertion" in this sense:

```
    * B = A^2, C = A^3
          B = B x C
    * B = A^5, C = A^3
```

To prove this correct, we must verify that, if $B = A^2$ and $C = A^3$, and then B is set equal to $B \times C$, then we will have $B = A^5$ and $C = A^3$. To anyone who knows what assignment statements mean, this is obvious. Let us, however, briefly consider the idea of mechanizing this proof process, that is, writing a computer program which verifies single statements in this fashion, and, it is hoped, larger programs as well. Obviously we would need a convention to distinguish an assertion from a statement; we have used an asterisk preceding each assertion, which allows assertions to look like comments (in most programming languages) so that we can leave them in our programs when we run them. We would also need some other method of specifying exponentiation, such as ** as it is used in FORTRAN. The character strings 'B = A**2' and 'C = A**3', representing assertions, and 'B = B $\times$ C', representing an assignment, must be combined by our program to form 'B = A**5' and 'C = A**3'. This process depends for its validity on the mathematical meaning of assignments and assertions, which will be more fully discussed in Chapter 3. For the present, we shall accept intuitive arguments when they arise; thus in the proof of the one-statement program above we shall simply say that it is correct "by inspection." It is also clear that the entire program is correct if each of the one-statement programs it contains is correct; thus in this case the entire program is correct by inspection.

1-3 Branch-Forward Programs

We now consider programs which contain transfer (or jump, or branch) statements. The simplest of these programs to analyze are the ones in which all transfers are in the forward direction; this is because such programs cannot contain any loops. We shall refer to them as branch-forward programs.

The following FORTRAN II function calculates the absolute value of X:

```
          FUNCTION ABSF(X)
          ABSF = X
          IF (X) 1, 2, 2
    1     ABSF = -X
    2     RETURN
          END
```

We have chosen FORTRAN II because in most algebraic languages this program would not require any branches at all. In FORTRAN IV, for example, we would have written IF (X .LT. 0) ABSF = -X; in ALGOL, we would collapse the whole program into ABSF := if X<0 then -X else X. FORTRAN II, however, forces us, even in this simple case, to consider the issue of branching.

It is clear that only the function variable ABSF can have its value changed by this program. Let us set up intermediate assertions for this program, just as we did in the preceding section:

```
          FUNCTION ABSF(X)
          ABSF = X
    * ABSF = X
          IF (X) 1, 2, 2
```

```
* X < 0, ABSF = X
1     ABSF = -X
* ABSF EQUALS THE ABSOLUTE VALUE OF X
2     RETURN
      END
```

The proof here is a bit harder than the preceding one, because when we consider the IF statement we must distinguish two cases (in general, three). If X was less than zero, we pass to statement 1, and the assertion there is simply that X is less than 0 (and that ABSF is still equal to X, a fact which is not actually used here). If X was greater than or equal to zero, we pass to statement 2. In this case, ABSF will be equal to the absolute value of X because ABSF = X and $X \geq 0$. If statement 1 is executed, ABSF will then be equal to the absolute value of X because ABSF = -X and $X < 0$.

All this is quite straightforward, and we would therefore like to be able to dispense with the intermediate assertions and to say that this program is correct by inspection. In doing this, it is helpful to write out each of the two paths which the program may take:

```
ABSF = X              ABSF = X
(X < 0)               (X ≥ 0)
ABSF = -X             RETURN
RETURN
```

In each of these paths we have written the condition which actually holds whenever that path is taken; these conditions are enclosed in parentheses to identify them as conditions. It is not necessary to include any labels in the specifications of the paths. In each of the two cases above, the final assertion that ABSF is equal to the absolute value of X may be immediately verified.

As another example of a branch-forward program, we consider the sorting of an array of two elements, given here with intermediate conditions at each stage:

```
* A(1) = X, A(2) = Y
        IF (A(1) .LE. A(2)) GO TO 1
* A(1) = X, A(2) = Y, X > Y
        T = A(1)
* A(1) = X, A(2) = Y, X > Y, T = X
        A(1) = A(2)
* A(1) = Y, A(2) = Y, X > Y, T = X
        A(2) = T
* A(1) = min(X, Y), A(2) = max(X, Y)
1       CONTINUE
```

Here we have used FORTRAN IV; *max* and *min* are the usual maximum and minimum functions (which in FORTRAN would be written AMAX and AMIN when applied to real numbers, so as not to begin with I, J, K, L, M, or N). As before, the treatment of the IF statement must be broken up into two cases. The only verification here which is not immediately obvious is that involving the statement A(2) = T, and this becomes clear as soon as we realize that, since $X > Y$, we must have $X = \mathit{max}(X, Y)$ and $Y = \mathit{min}(X, Y)$ in this case. Using the shorter method, we may write out two paths as before. The first is

```
* A(1) = X, A(2) = Y
        (A(1) ≤ A(2))
* A(1) = min(X, Y), A(2) = max(X, Y)
```

and the second is

```
* A(1) = X, A(2) = Y
      (A(1) > A(2))
      T = A(1)
      A(1) = A(2)
      A(2) = T
* A(1) = min(X, Y), A(2) = max(X, Y)
```

Here the values of X and Y cannot change, since indeed they do not appear in the program at all. In each of these paths, there is both an initial and a final assertion. We shall continue to precede these assertions by asterisks, and to write intermediate conditions in parentheses, since intermediate conditions and assertions play quite different rôles in verification. Each intermediate condition must hold at the point at which it is encountered; thus in the second path above we may infer from A(1) > A(2) that we must have had X > Y at the beginning of the path, so that X = max(X, Y), Y = min(X, Y).

Paths in programs, as illustrated above, are subject to certain general rules which we may now state. A path consists of: an initial assertion, which may be omitted (and reads "always true" if it is omitted); followed by any number of assignments and conditions; followed by a final assertion. The path is correct (or verified, or valid) if: whenever it is started with its initial asserti n valid, and if it is actually followed to its end (which means, in particular, that each intermediate assertion is true as it is passed), then when it is finished the final assertion will be valid. The length of such a path is the number of assignments and conditions in it, irrespective of the initial and final assertion.

1-4 Closed Loops

A closed loop in a program is a sequence of statements, such that it is theoretically possible for the program to proceed from the first of them through the sequence to the last and then back to the first. A straight-line program, or, more generally, a branch-forward program, cannot have closed loops. It is possible to write a program which has no closed loops and yet is not a branch-forward program, as shown by the following example (again in FORTRAN II):

```
            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
```

This function computes the sum of the absolute values of X and Y. The statements GO TO 5 and GO TO 9 are branches backward. 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, any program with no closed loops may have its statements ordered in such a way; this follows from the properties of partial orderings. In what follows, we shall assume that any program with no closed loops is in fact a branch-forward program.

As a simple example of a program with a closed loop, let us consider the following program to calculate A^N, in which the variables whose values may be changed are X and I:

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

In order for this program to work, N must be an integer which is either positive or zero. Let us write an assertion before each statement of this program:

```
    * N ≥ 0
          X = 1
    * N ≥ 0, X = 1
          I = 0
    * N ≥ 0, I ≤ N, X = A^I
    1     IF (I .EQ. N) GO TO 2
    * N ≥ 0, I < N, X = A^I
          X = X * A
    * N ≥ 0, I < N, X = A^(I+1)
          I = I + 1
    * N ≥ 0, I ≤ N, X = A^I
          GO TO 1
    * X = A^N
    2     CONTINUE
```

This example raises the question of how to figure out what our assertions should be, especially as our programs get longer. In fact, the determination of assertions in proofs of programs is a task which is comparable to programming itself. In this case, we must know enough about how our program works to be able to assert, for example, that whenever we get to statement number 1 (either the first time or at some later time) then N will still be greater than or equal to zero, I will be less than or equal to N, and the value of X, which is a "partial answer," will be exactly A^I.

Now suppose that we verify what happens at each statement, just as we did before. This verification is obvious at the statement $X = 1$. At the statement $I = 0$, we obtain $I \leq N$ (clear, since $I = 0$ and $N \geq 0$) and $X = A^I$ (clear, since $I = 0$ and $X = 1$).* At statement number 1, we start with $X = A^I$; if $I = N$, then we clearly obtain $X = A^N$. If $I \neq N$, then since we started with $I \leq N$, we get $I < N$, which is the only thing that changes. At the statement $X = X * A$, the only thing that changes is that $X = A^I$ becomes $X = A^{I+1}$, which is clear since $A^I * A = A^{I+1}$. At the statement $I = I + 1$, we start with $I < N$ and obtain $I \leq N$, which is true but depends on the fact that both I and N are integers. We also start with $X = A^{I+1}$ and obtain $X = A^I$, which is clear since I has been increased by one. Finally, the GO TO statement does not change anything.

It is strongly recommended that the reader work through the above verifications before proceeding further, since abbreviated arguments such as those made in the preceding paragraph will be used constantly throughout this chapter. At the risk of over-specification, we shall write out one of these arguments in full. Let us write the statement $X = X * A$ together with the assertions which precede and follow it:

* Throughout this chapter we assume $0^0 = 1$. If 0^0 is left undefined, then we must, for example, include $A \neq 0$ as an initial assertion for this program.

$$* \; N \geq 0, \; I < N, \; X = A^I$$
$$X = X * A$$
$$* \; N \geq 0, \; I < N, \; X = A^{I+1}$$

This is a path of length 1. We must verify that <u>if</u> $N \geq 0$, $I \leq N$, and $X = A^I$, <u>and if</u> we set $X = X * A$, <u>then</u> we will have $N \geq 0$, $I \leq N$, and $X = A^{I+1}$. These three conclusions are verified as follows. We have $N \geq 0$ and $I < N$ at the end because these conditions were both true at the beginning and because neither N nor I has been changed by the assignment $X = X * A$. We have $X = A^{I+1}$ at the end because $X = A^I$ at the beginning and we have multiplied the value of X by A in the given assignment. Thus the path is valid. This entire argument is then abbreviated, as we have done in the preceding paragraph, to "At the statement $X = X * A$, the only thing that changes is that $X = A^I$ becomes $X = A^{I+1}$, which is clear since $A^I * A = A^{I+1}$."

What can we infer from these arguments? Can we, in particular, infer that our program is correct? Suppose that the program takes $\underline{n}$ steps to execute, and suppose that, before the first step, the initial assertion is valid. We may now show inductively that, before the $\underline{i}$th step, which presumably consists in executing some statement, the assertion given before that statement will be valid. In fact, if this is true for $\underline{i} = \underline{k}$, the preceding arguments have shown precisely that it is true for $\underline{i} = \underline{k} + 1$. In particular, it is true for $\underline{i} = \underline{n}$, so that, when the program terminates, the final assertion will be valid. Thus the program is correct. The trouble with this argument is that we have no way of knowing that the program takes a finite number of steps at all. In other words, we now know that <u>either</u> the program is correct <u>or</u> it goes into an endless loop. This situation is so common in proofs of correctness that it has a special name: partial correctness. We now define correctness and partial correctness in a general framework.

Let P be a program and let A_1, ..., A_n be assertions about the variables of P. Let each A_i be associated with a statement F_i of P. Then P is correct with respect to the A_i and the F_i if, whenever it is started at some F_x with A_x true, then it will eventually get to some F_y, and at that point A_y will be true. It is partially correct with respect to the A_i and the F_i if, whenever it is started at some F_x with A_x true, then if it ever gets to some F_y the assertion A_y will be true at that point.

If every statement of P has an assertion associated with it, then the correctness of P with respect to all these assertions is simply the statement that every path in P of length 1 is valid. Such a program, even though it is correct (not just partially correct), may still get into an endless loop. On the other hand, suppose that we choose for the F_i the starting and stopping statements of the program, and for the A_i the initial and final assertions. If we can show that the program never returns to the start, then the correctness of P now means that if the program is started at the beginning with the initial assertion valid, then it will get to the end, and when it does the final assertion will be valid. As we have seen above, the partial correctness in this case may be proved by adding more assertions until every statement has an associated assertion, and then proving correctness (or partial correctness, which is the same in this case) of the result. This is a special case of the theorem of the next section. Note that we may also allow here for P to start and/or stop in more than one place. A straight-line program, of course, always starts at the (physical) beginning and stops at the end, so that reference to associated statements is superfluous in this case.

1-5 Control Points and Control Paths

The exponentiation program of the previous chapter is not only partially correct; it is correct. Before we prove this, however, let us show how its partial correctness may be proved more simply by the use of paths of length greater than 1.

In a branch-forward program, each path from the beginning of the program to the end has a length which is less than or equal to the total number of statements in the program. In a more general situation, however, this is not true. In our exponentiation program, we go around the loop N times, and therefore there is a different path from the beginning of the program to its end for each integer $N \geq 0$. Because this program is so simple, we can, in fact, analyze each of the infinite number of paths of this form; but this is much more difficult for longer programs. There is, however, a method by which we may always restrict ourselves to the consideration of a finite number of paths. This is the method of control points and control paths.

A control point in a program is any statement for which we have made an assertion. A control path is any path from one control point to another, with no control points in between. If we have an assertion attached to every statement in the program, then every statement is a control point, and every control path has length 1. The other extreme is the case in which only the starting and stopping points of our program are control points; in this case, there are an infinite number of control paths unless the given program is a straight-line or branch-forward program. We would like to work with an intermediate case, in which the number of control paths is finite while the number of control points is as small, or nearly as small, as possible.

In our exponentiation program, it is sufficient to consider one control point in addition to the starting and stopping points. Suppose, for example, that statement number 1 is a control point. The assertion at this statement is the same as it was before. Let us write the program with assertions only at the control points we have selected:

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

Remembering that a control path stretches from one control point to another, with no control points in between, we see that there are three control paths, as follows:

```
* N ≥ 0                * N ≥ 0, I ≤ N, X = A^I     * N ≥ 0, I ≤ N, X = A^I
      X = 1                  (I = N)                     (I ≠ N)
      I = 0            * X = A^N                         X = X * A
* N ≥ 0, I ≤ N, X = A^I                                  I = I + 1
                                                   * N ≥ 0, I ≤ N, X = A^I
```

In the third of these control paths, there is a GO TO statement (GO TO 1), which is omitted since it has no bearing on the validity of the path. This path is also an example of the fact that a control path may stretch from one control point back to itself.

Suppose now that we can show each of these paths to be valid. Is the program still partially correct? Let F_1 be the starting statement of the program; as we run it, let F_2 be the first control point we encounter, let F_3 be the second one, and so on. (Some of the F_i may be the same.) Between F_i and F_{i+1} there is a control path, and we see by induction that if the assertion at F_1 is valid then the assertion at F_i is valid for each i. If the program takes a finite number of steps, then there will be a finite number of F_i, and one of them will be a stopping statement, since such a statement is always a control point. The assertion at this point will be valid, and the program is therefore partially correct. This is a special case of the following theorem:

<u>Let Π be a set of pairs (A_i, F_i), where each A_i is an assertion which is associated with the statement F_i in the program P, and all of the F_i are distinct. Then P is correct with respect to any subset $\Pi' \subseteq \Pi$ if it is correct with respect to Π.</u>

Normally, $(A_i, F_i) \in \Pi'$ if and only if F_i is a starting or stopping statement of P. By including a pair $(A_i, F_i) \in \Pi$ for <u>every</u> statement F_i of P, we obtain the method of the preceding section; by including $(A_i, F_i) \in \Pi$ only for control points F_i, we obtain the method of this section.

A collection of control points is <u>sufficient</u> if every closed loop in the program must go through at least one control point. This condition assures us that the total number of control paths is finite, as will be proved in the next paragraph. In our exponentiation program, we have a sufficient collection of control points because there is only one loop in the program, and that loop passes through the statement numbered 1. If we took, instead of the statement numbered 1, any of the three statements which follow it as a control point, we would still have had a sufficient collection of control points.

Let us now consider control paths when the collection of control points is sufficient. Suppose that a control path loops back to some previous point on itself. Then this point must be its starting point, because otherwise there would be a closed loop in the program which contains no control points. Therefore the only two statements in such a control path that can be the same as its start and its end. But since each program contains only a finite number of distinct statements, there can be only a finite number of <u>sequences</u> of distinct statements followed by a single arbitrary statement. Hence there are only finitely many control paths. We have proved that <u>whenever we have a sufficient collection of control points, the total number of control paths relative to these control points is finite</u>. The converse is also true, provided that there are no loops which can never be reached.

In one sense, we have already proved that each control path in our exponentiation program is valid. In fact, by including (A_i, F_i) in Π for every statement F_i, and in Π' for the control points only, we infer from our theorem that the validity of control paths is a consequence of the validity of paths of length 1 (this is also easy to see directly), which we already know in this case. Let us, however, show independently that each control path is valid; as before, we use abbreviated arguments, all of which may be "written out" using the paths of length 1 as above. The first path gives $N \geq 0$ (clear), $I \leq N$ (clear from $I = 0$ and $N \geq 0$), and $X = A^I$ ($= A^0 = 1$, clear). The second path gives us $X = A^N$, which is clear from $X = A^I$ since $I = N$. The third path gives us $N \geq 0$ (clear), $I \leq N$, and $X = A^I$. Since originally $I \leq N$ and $I \neq N$, we must have had $I < N$, so that $I + 1 \leq N$ (since I and N are integers). Also, $X = A^i \cdot A = A^{i+1}$, where i is the old value of I; but $i+1$ is then the new value of I, so that $X = A^I$. This completes the verification.

One method of assuring that we have a sufficient collection of control points, in most computer languages, is to choose the labelled statements as control points. This works because in most languages a closed loop must include at least one labelled statement. In assembly language, in order to use this trick, one must be assured that the given program contains no transfers backward to an unlabelled statement, such as a label plus or minus a constant.

1-6 Controlled Expressions

Let us now finish the proof of correctness of our exponentiation program. We have seen that it is now sufficient to prove that it always terminates in a finite number of steps.

The proof of this fact might have been easier if we had used a DO loop, as follows:

```
          X = 1
          DO 2 I = 1, N
    2     X = X * A
```

This is not quite equivalent to our original program, and is not even correct for N = 0 because of the way FORTRAN works. But the termination of such a program follows immediately from a theorem which may be formulated specifically for FORTRAN. One merely needs to check certain things: that I and N are never changed inside the loop; that there is no inner loop (or that any inner loops terminate); and so on. In any other language, with a different sort of iteration statement, we may prove a similar theorem.

The trouble with this is that DO statements, and iteration statements in general, are not very easy to handle when it comes to proving partial correctness. In fact, when we prove partial correctness by the methods outlined above, iteration statements cannot really be considered as statements at all, in the sense of being associated with assertions. It is possible to revise our methods so that they apply to programs containing iteration statements, but it is simpler to "write out" all iteration statements; that is, we include statements which explicitly initialize, increment, and test the controlled variable.

We must therefore consider the general problem of proving termination without iteration statements. The key to such proofs is usually the quantity which, if we had used an iteration statement, would have played the rôle of the controlled variable. Very often this quantity is in fact a single variable; in our exponentiation program, it is I. The properties of the variable I which we shall need in this program are:

(1) It increases each time we go around the loop;

(2) It is bounded at some statement in the loop.

In this loop, the statement I = I + 1 is the only statement which changes the value of I, and thus I always increases when we go around the loop. The assertion $I \leq N$, among others, appears at statement number 1, and this shows that I is bounded. It is necessary, of course, that the maximum value of I, namely N, be constant inside the loop; It may, however, be changed outside the loop; for example, it may be initialized at the start of the program.

We shall now state and prove, in informal terms, a theorem about the termination of programs which have exactly one closed loop and exactly one control point in that loop. In this case, there will be one control path which goes completely around the loop, which we call the loop control path. In general, a loop control path in a program is any control path which is contained completely within a single closed loop of that program. Our theorem now reads as follows:

A program with one loop and one control point in that loop always terminates if there exists an integer expression whose value always increases as we go around the loop, and is bounded at the control point by a quantity which does not change inside the loop.

Here we have used the phrase "always terminates" to mean "always terminates in a finite number of steps (i. e., takes only a finite number of steps to execute) when started with its initial assertion valid." The statement that the program is bounded at the control point implies that we have already set up an assertion there, and verified it by showing that every control path in the program is valid. That is, when we are proving correctness, it is assumed that partial correctness has already been proved.

The proof of our theorem proceeds as follows. Suppose, in contradiction, that the loop is endless. We start our program and run it until we get to the control point. Suppose that the value of our integer expression is now v_1; we also know that this expression is bounded by a quantity whose value at this point we call b. Thus we have $v_1 \leq b$. We now continue the program until we get back to the control point; suppose that the integer expression now has value v_2. By hypothesis, we have $v_2 > v_1$, and also $v_2 \leq b$ because the value of the bound does not change within the loop. Continuing in this way, we derive a strictly increasing sequence of integers $v_1 < v_2 < \ldots$, all of which are less than or equal to the integer b, which is impossible. Thus the theorem is proved.

Integer expressions such as the one discussed here may be called _controlled expressions_. In general, a program will have several closed loops and one or more controlled expressions. In this case, we must determine how they interact with one another; this topic will be taken up again in section 1-11.

The simplest example of a situation in which a controlled expression does not consist of a single variable is that in which the controlled variable decreases rather than increasing, such as in the following program:

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

This program also calculates A^N. The variable I is not a controlled expression here, in the sense of the theorem above, because it does not increase. We could, of course, prove another theorem about variables which decrease, but it is simpler and more elegant to consider the negative of a "decreasing controlled expression" as an ordinary controlled expression. Thus, in this program, the controlled expression is -I. Let us rewrite the program with assertions placed at control points:

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

We have purposely omitted any assertions about X and A because we are only interested here in the question of termination. (The program is partially correct with respect to the given assertions, but this does not imply, of course, that it computes A^N.) It is now clear that the value of the expression $-I$ always increases whenever we go around the loop; and that it is bounded (by the constant zero) at a statement in the loop, namely statement number 1.

1-7 Assertions in Loops

The art of proving the correctness of programs, as we have suggested, consists to a great degree in finding the right assertions. This involves the idea of formalizing intuitive arguments, which is quite familiar to anyone who has ever learned to program, or, for that matter, to solve word problems in high school algebra. A person who has written a program usually has an intuitive idea of why it works. In proving that program it is merely necessary for him to translate this into a set of assertions at a sufficient collection of control points.

The following examples are meant to give the reader experience in finding such assertions in simple programs. Let us first finish the example of the previous section:

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

Why does this work? To be precise, what is the intermediate value of X at some point in the loop, such as statement number 1? Experienced programmers might make the intuitive statement that "it doesn't matter, because it is obvious that the loop is executed N times anyway." In proofs, however, this is not enough; one must always look for an assertion which may be stated in formal terms. In this case, the variable I goes from N down to zero while the power of A goes from zero up to N, so that I is always equal to N minus the power of A; or, in other words, the power of A is equal to N-I. This may be stated as the assertion $X = A^{N-I}$, and, after taking into account that X and I are the only variables whose values may change, the setup is complete:

```
* $N \geq 0$
      X = 1
      I = N
* $N \geq 0,\ I \geq 0,\ X = A^{N-I}$
1     IF (I .EQ. 0) GO TO 2
      X = X * A
      I = I - 1
      GO TO 1
* $X = A^N$
2     CONTINUE
```

The control paths are now as follows:

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

In the first control path, we have $N \geq 0$ (clear), $I \geq 0$ (clear from $I = N$ and $N \geq 0$), and $X = A^{N-I}$ ($= A^0 = 1$, clear). In the second, we have $X = A^N$, which is clear from $X = A^{N-I}$ and $I = 0$. In the third, we have $N \geq 0$ (clear), $I \geq 0$, and $X = A^{N-I}$. Since at the beginning $I \geq 0$ and $I \neq 0$, we must have had $I > 0$, which means that $I-1 \geq 0$ since I is an integer. Finally, we have $X = A^{N-I} * A = A^{(N-I)+1} = A^{N-(I-1)}$, and subsequently the value of I is decreased by one, giving $X = A^{N-I}$ again.

The following program finds the sum of all the elements of an array A of dimension $N \geq 1$, while changing only I and S:

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

What is the proper assertion here at statement number 1? Clearly it must, among other things, specify that S is equal to some partial sum, which depends on the value of I. A little checking will show, however, that S is not equal to $\sum_{x=1}^{I} A(x)$, but rather to $\sum_{x=1}^{I-1} A(x)$, at statement number 1. (By making the statement immediately following statement number 1 into our control point, we could have used $\sum_{x=1}^{I} A(x)$.) Also at statement number 1, I is less than or equal to N. We rewrite the program with assertions:

```
* $N \geq 1$
          I = 1
          S = 0
* $N \geq 1,\ I \leq N,\ S = \sum_{x=1}^{I-1} A(x)$
1         S = S + A(I)
          I = I + 1
          IF (I .LE. N) GO TO 1
* $S = \sum_{x=1}^{N} A(x)$
          CONTINUE
```

The first control path is

```
* $N \geq 1$
      $I = 1$
      $S = 0$
* $N \geq 1,\ I \leq N,\ S = \sum_{x=1}^{I-1} A(x)$
```

Here $N \geq 1$ is clear, $I \leq N$ is clear from $I = 1$ and $N \geq 1$, and S is asserted to be "a sum from x=1 to 0," which must clearly be defined as zero, which is the value given to S. The second control path is

```
* $N \geq 1,\ I \leq N,\ S = \sum_{x=1}^{I-1} A(x)$
      $S = S + A(I)$
      $I = I + 1$
      $(I \leq N)$
* $N \geq 1,\ I \leq N,\ S = \sum_{x=1}^{I-1} A(x)$
```

Here $N \geq 1$ is clear, $I \leq N$ follows from the intermediate condition $(I \leq N)$, and $S = \sum_{x=1}^{I-1} A(x)$ is preserved because $S = S + A(I)$ adds the I-th term to the first I-1, and then I is increased by one. The third and last control path is

```
* $N \geq 1,\ I \leq N,\ S = \sum_{x=1}^{I-1} A(x)$
      $S = S + A(I)$
      $I = I + 1$
      $(I > N)$
* $S = \sum_{x=1}^{N} A(x)$
```

Here we must have had at the beginning $I + 1 > N$ but $I \leq N$, therefore $I = N$. The N-th term is added to S and the path thus becomes clearly valid.

Termination of this program follows by the same arguments that were used in the preceding section. The controlled expression is I, and, just as before, it is increasing as we go around the loop and is bounded at the statement numbered 1.

1-8 Inductive Assertions

The assertion $S = \sum_{x=1}^{I-1} A(x)$ in the last program above is rather awkward to present to a computer program, such as a verification program, as it stands. Such assertions are, rather, defined inductively in this situation. If we let C(X, K) stand for the assertion $X = \sum_{x=1}^{K} A(x)$, then C(X, 0) is equivalent to X = 0, while C(X, K-1) is equivalent to C(X + A(K), K) for K > 0.

It is remarkable that such inductive definitions often lead to simplified proofs of the validity of paths, even when the given program is being verified by hand. For example, consider the second of the three control paths in the last program above, rewritten to make reference to the assertion C(X, K) above:

```
* N ≥ 1, I ≤ N, C(S, I-1)
      S = S + A(I)
      I = I + 1
      (I ≤ N)
* N ≥ 1, I ≤ N, C(S, I-1)
```

The argument of the previous chapter, using the summation sign, is intuitive and rather tedious to formalize. The argument above, however, is immediate if we rewrite the first instance of C above as C(S + A(I), I). The first assignment in this path now converts this into C(S, I), and the second assignment converts it into C(S, I-1), which is exactly the form given in the final assertion.

We shall give defining relations for conditions at the beginning of the program, preceded by two asterisks. As another example of their use, we prove the correctness of a program to move an array from one place to another. It is equivalent to the simple DO loop

```
          DO 1 I = 1, N
    1     B(I) = A(I)
```

The intermediate result here is that B($\underline{x}$) = A($\underline{x}$) for all $\underline{x}$ ($\geq$ 1) less than or equal to I-1. We call this assertion C(I-1), and define C inductively before listing the program, as follows:

```
    ** C(0) = .TRUE.
    ** C(K) = C(K-1) .AND. (A(K) = B(K))
    * N ≥ 1
          I = 1
    * 1 ≤ I, I ≤ N, C(I-1)
    1     B(I) = A(I)
          I = I + 1
          IF (I .LE. N) GO TO 1
    * C(N)
          CONTINUE
```

We note that if the second statement of this program is replaced by A(I) = B(I), the same assertions still apply; the difference, of course, is that such a program would change the values of elements of the array A, whereas the given program does not. The statement B(I) = A(I) will execute properly only if the subscript I is in its proper range; this is the sole reason for the assertion 1 $\leq$ I at statement number 1. (One must also be assured here that the dimension specified for the arrays A and B is not less than N.) The control paths are now listed at the left, with their corresponding proofs at the right:

```
* N ≥ 1
                            1 ≤ I follows from I = 1, and I ≤ N from
      I = 1
                            I = 1 and N ≥ 1; C(I-1) = C(0) = .TRUE.
* 1 ≤ I, I ≤ N, C(I-1)
* 1 ≤ I, I ≤ N, C(I-1)
      B(I) = A(I)
      I = I + 1
      (I ≤ N)
* 1 ≤ I, I ≤ N, C(I-1)
```

$1 \leq I$ is clear since I increases; $I \leq N$ follows from the intermediate assertion $I \leq N$; C(I-1) at the end is by definition C(I-2) and B(I-1) = A(I-1), each of which follow from the path.

```
* 1 ≤ I, I ≤ N, C(I-1)
      B(I) = A(I)
      I = I + 1
      (I > N)
* C(N)
```

We must have I = N at the beginning in order to have I > N at the end. Therefore C(N-1) holds at the beginning, followed by B(N) = A(N), giving C(N).

The controlled expression for this program is I; the argument is the same as before. This completes the proof of correctness.

We now consider a loop which has two exits. The following program searches the array A for the quantity X; the intermediate result here is that we have not found X yet for as far as we have searched. Specifically, just before we check whether X = A(I), we must know that $X \neq A(x)$ for each $x \leq I-1$. We call this assertion F(I-1), and define it inductively as before.

```
** F(0) = .TRUE.
** F(K) = F(K-1) .AND. (X ≠ A(K))
* N ≥ 1
        I = 1
* 1 ≤ I, I ≤ N, F(I-1)
1       IF (X .EQ. A(I)) GO TO 3
        I = I + 1
        IF (I .LE. N) GO TO 1
```

```
* F(N)
2       CONTINUE
         . . .
* F(I-1), X = A(I)
3       CONTINUE
```

Again we note the assertion $1 \leq I$, included for the same reason as before. Since the program has two exits, correctness is proved with respect to three (sets of) assertions, one at the beginning of the program, one at statement 2, and one at statement 3. We show the control paths to be valid as follows:

* $N \geq 1$ $I = 1$ * $1 \leq I$, $I \leq N$, F(I-1)	$1 \leq I$ follows from $I = 1$, and $I \leq N$ from $I = 1$ and $N \geq 1$; $F(I-1) = F(0) =$ .TRUE.
* $1 \leq I$, $I \leq N$, F(I-1) $(X = A(I))$ * F(I-1), $X = A(I)$	Follows immediately.
* $1 \leq I$, $I \leq N$, F(I-1) $(X \neq A(I))$ $I = I + 1$ $(I \leq N)$ * $1 \leq I$, $I \leq N$, F(I-1)	$1 \leq I$ is clear since I increases; $I \leq N$ follows from the condition $I \leq N$ in the path; F(I-1) is F(I-2) and $(X \neq A(I-1))$, both of which are clear from the path.
* $1 \leq I$, $I \leq N$, F(I-1) $(X \neq A(I))$ $I = I + 1$ $(I > N)$ * F(N)	We must have $I = N$ at the beginning in order to have $I > N$ at the end. Therefore F(N-1) holds at the beginning, and $X \neq A(N)$, giving us F(N).

The third of these paths is the only one which is a loop control path. The termination argument is again just as before; the controlled expression is I. The only variable whose value may be changed by this program is I.

1-9 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 number 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 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 three things remaining to be cleaned 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. From what we now know about assertions, it is clear that we must derive an assertion in the loop, at statement number 1, for example. This assertion says that we have checked all integers less than J (and greater than 1) for being divisors of I; that is, $I \neq \underline{x} \cdot \underline{y}$ for integers $\underline{x}$ and $\underline{y}$ with $2 \leq \underline{x} \leq J-1$. Let us call this condition C(J-1), and write out the loop control path using it:

```
* C(J-1)
        (MOD(I, J) ≠ 0)
        J = J + 1
        (J*J ≤ I)
* C(J-1)
```

This path is valid, since if MOD(I, J) ≠ 0 and C(J-1) is true, then so is C(J), and then J is increased by 1. Furthermore, C(1) is always true, so that the initial control path of the routine is valid whatever the initial assertion is. If C(J) is true for some J with J*J > I, then we may infer that I is prime, and so the path which includes the statement PRIME = .TRUE. is valid. This leaves the path which includes the statement PRIME = .FALSE. . If MOD(I, J) = 0, then I is certainly not prime unless J is equal to 1 or I. How can we show that this will never happen? J is certainly not 1, because it starts at 2 and increases; similarly, J*J is tested to be less than or equal to I, so J should certainly be less than I. Let us incorporate these facts into our proof by adding the assertions J ≥ 2 and J*J ≤ I at statement number 1; then J < I will follow immediately. The loop control path now reads:

```
* C(J-1), J ≥ 2, J*J ≤ I
        (MOD(I, J) ≠ 0)
        J = J + 1
        (J*J ≤ I)
* C(J-1), J ≥ 2, J*J ≤ I
```

This is valid, since J is increased and hence the inequality $J \geq 2$ is preserved, whereas the assertion $J*J \leq I$ follows from the condition $J*J \leq I$ in the path. However, we must now also go back and re-verify the initial control path, which now reads:

```
* I > 0
      J = 2
* C(J-1), J ≥ 2, J*J ≤ I
```

Unfortunately, this path is not valid. C(J-1) is C(1), which is always true, and $J \geq 2$ certainly holds, but $J*J \leq I$ will not hold unless I is at least 4.

Well, that is easy enough. We can verify the program with the initial assertion $I \geq 4$ -- in fact, we have already done so -- and then go back and check the special cases I = 1, 2, and 3. 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 no other positive integer value of I), the program given above is not 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; and, in fact, once the bug is fixed, the proof of partial correctness of the program follows immediately from what we have done up to now. We rewrite the program in correct form:

```
** C(1) = .TRUE.
** C(K) = C(K-1) .AND. (I ≠ K•z for integer z)
         LOGICAL FUNCTION PRIME(I)
* I > 0
         IF (4 .GT. I) GO TO 3
         J = 2
* C(J-1), J ≥ 2, J*J ≤ 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.
* PRIME = .TRUE. and I is prime
         RETURN
2        PRIME = .FALSE.
* PRIME = .FALSE. and I is not prime
         RETURN
         END
```

To finish the proof of correctness, we need only note that J is our controlled expression. It increases around the loop, and the assertion J*J ≤ I, together with J ≥ 2, serves to bound J at the statement numbered 1.

1-10 Multiple Loops

We shall now apply our methods to the verification of a slightly larger program, one which finds all prime numbers from 1 to N using the sieve of Eratosthenes. Because this program includes more than one loop, only its partial correctness will be proved in this section.

The program uses a logical (i. e., Boolean) array whose size will be left unspecified, although it is assumed that this size is greater than or equal to N. The purpose of the program is to set the elements of this array, called A, in such a way that A(I) is .TRUE. if I is prime and .FALSE. otherwise. First, all elements of the array are set to .TRUE.; then we start the sieve by "crossing out" (i. e., setting to .FALSE.) all elements A(I) such that I is a multiple of 2, 3, 4, and so on. In crossing out multiples of J, it may be assumed that multiples of the form J*L with L < J have already been crossed out, since J*L = L*J, so that we first cross out A(4), A(6), A(8), etc., then A(9), A(12), A(15), etc., and continue in this fashion. As in the preceding program, we make use of the fact that if N (or any number less than N) is the product of two integers, at least one of them must be less than or equal to the square root of N; hence the last multiples to be crossed out are the multiples of M, where M*M $\leq$ N but (M+1)*(M+1) > N.

Our first task is to determine precisely what the intermediate assertions should be at all labelled statements. At statement number 1, we have set each A($\underline{x}$) to .TRUE. for 1 $\leq$ $\underline{x}$ $\leq$ I-1; we denote this assertion by U(I-1). At statement number 2, we have for all $\underline{x}$, 1 $\leq$ $\underline{x}$ $\leq$ N, that A($\underline{x}$) = .FALSE. if $\underline{x}$ = $\underline{y}$·$\underline{z}$ for positive integers $\underline{y}$ $\leq$ $\underline{z}$ with 2 $\leq$ $\underline{y}$ $\leq$ I-1, and A($\underline{x}$) = .TRUE. otherwise; we denote this assertion by V(I-1). At statement number 3, we have for all $\underline{x}$, $1 \leq \underline{x} \leq N$, that A($\underline{x}$) = .FALSE. if $\underline{x} = \underline{y} \cdot \underline{z}$ for positive integers $\underline{y} \leq \underline{z}$ with $2 \leq \underline{y} \leq I-1$, or if $\underline{x} \leq J-1$ and $\underline{x} = I \cdot \underline{z}$ for some integer $\underline{z} \geq I$, and A($\underline{x}$) = .TRUE. otherwise; we denote this assertion by W(I, J-1). Finally, at statement 4, A($\underline{x}$) is .TRUE. or .FALSE. according as $\underline{x}$ is prime or not, for $1 \leq \underline{x} \leq N$. With these assertions (for the moment not defined inductively; see also section 2-6), the program reads as follows:

```
* N > 0
      I = 1
* 1 ≤ I, I ≤ N, U(I-1)
1     A(I) = .TRUE.
      I = I + 1
      IF (I .LE. N) GO TO 1
      I = 2
* I ≥ 2, (I-1)*(I-1) ≤ N, V(I-1)
2     J = I*I
      IF (J .GT. N) GO TO 4
* I ≥ 2, I*I ≤ J, J ≤ N, MOD(J, I) = 0, W(I, J-1)
3     A(J) = .FALSE.
      J = J + I
      IF (J .LE. N) GO TO 3
      I = I + 1
      GO TO 2
* A(x) = PRIME(x), 1 ≤ x ≤ N
4     CONTINUE
```

In the last assertion we have used the PRIME function as defined in the previous chapter. There are seven control paths, as follows:

```
Path 1      * N > 0
                  I = 1
            * 1 ≤ I, I ≤ N, U(I-1)
```

The assertion $1 \leq I$ follows from $I = 1$, and $I \leq N$ follows from $I = 1$ and $N > 0$. Also, U(I-1) is U(0), which is vacuously true.

```
Path 2      * 1 ≤ I, I ≤ N, U(I-1)
                  A(I) = .TRUE.
                  I = I + 1
                  (I ≤ N)
            * 1 ≤ I, I ≤ N, U(I-1)
```

Here $1 \leq I$ follows since I is increased, and $I \leq N$ follows from the condition $I \leq N$ in the path. The reference to A(I) may be made because $1 \leq I$ and $I \leq N$. The assertion U(I-1) at the beginning of the path and the assignment A(I) = .TRUE. give U(I), and then I is increased by 1, giving U(I-1) again.

```
Path 3      * 1 ≤ I, I ≤ N, U(I-1)
                  A(I) = .TRUE.
                  I = I + 1
                  (I > N)
                  I = 2
            * I ≥ 2, (I-1)*(I-1) ≤ N, V(I-1)
```

Again $1 \leq I$ and $I \leq N$ legitimize the reference to A(I). The assertion $I \geq 2$ follows from $I = 2$; also, (I-1)*(I-1) = 1, and $1 \leq N$ because at the beginning of the path $1 \leq I$ and $I \leq N$. The assertion V(I-1) = V(1), that is, A(x̲) = .TRUE. for all x̲, $1 \leq$ x̲ $\leq N$. This is the same as U(N), which follows at the end of the path since we must have $I = N$ at the beginning of the path in order to have the intermediate condition $I > N$.

<u>Path 4</u> $* \; I \geq 2, \; (I-1)*(I-1) \leq N, \; V(I-1)$

$J = I*I$

$(J > N)$

$* \; A(\underline{x}) = PRIME(\underline{x}), \; 1 \leq \underline{x} \leq N$

Since $A(\underline{x})$ is .TRUE. for $1 \leq \underline{x} \leq N$ whenever it is not .FALSE., and the same is true of $PRIME(\underline{x})$, it suffices to prove that $PRIME(\underline{x})$ is .FALSE. if and only if $A(\underline{x})$ is. If $PRIME(\underline{x})$ is .FALSE., then $\underline{x} = \underline{y}\cdot\underline{z}$ for $\underline{y}, \underline{z} \neq 1$, and $\underline{y}, \underline{z} \neq \underline{x}$. We may assume $\underline{y} \leq \underline{z}$, so that $2 \leq \underline{y} \leq \underline{z} < \underline{x}$; If $\underline{y} \geq I$, then $\underline{y} \leq \underline{z}$ implies $\underline{z} \geq I$, and $\underline{y}\cdot\underline{z} \geq I*I = J > N$, which contradicts $\underline{y}\cdot\underline{z} = \underline{x} \leq N$. Hence $\underline{y} < I$ or $\underline{y} \leq I-1$, which, by $V(I-1)$, implies that $A(\underline{x})$ is .FALSE.. Conversely, if $A(\underline{x})$ is .FALSE., then $\underline{x} = \underline{y}\cdot\underline{z}$ for positive integers $\underline{y}$ and $\underline{z}$ with $2 \leq \underline{y} \leq I-1$ and $\underline{y} \leq \underline{z}$, which implies $\underline{y} \neq 1$, $\underline{z} \neq 1$, and hence $\underline{y} \neq \underline{x}$, $\underline{z} \neq \underline{x}$, so that $PRIME(\underline{x})$ is .FALSE..

<u>Path 5</u> $* \; I \geq 2, \; (I-1)*(I-1) \leq N, \; V(I-1)$

$J = I*I$

$(J \leq N)$

$* \; I \geq 2, \; I*I = J, \; J \leq N, \; MOD(J, I) = 0, \; W(I, J-1)$

The assertion $I \geq 2$ is unchanged by the path; $I*I = J$ follows from the assignment in the path, and $J \leq N$ from the condition in the path. The assertion $MOD(J, I) = 0$ also follows from $J = I*I$. This leaves $W(I, J-1)$, which is $W(I, I*I-1)$; but if $\underline{x} = I\cdot\underline{z}$ for some integer $\underline{z} \geq I$, then $\underline{x} \geq I*I$, and we cannot have $\underline{x} \leq I*I-1$. Thus $W(I, I*I-1)$ is the same as $V(I-1)$, which is preserved from the beginning of the path since I is unchanged along it.

Path 6

```
* I ≥ 2, I*I ≤ J, J ≤ N, MOD(J, I) = 0, W(I, J-1)
      A(J) = .FALSE.
      J = J + I
      (J ≤ N)
* I ≥ 2, I*I ≤ J, J ≤ N, MOD(J, I) = 0, W(I, J-1)
```

Again $I \geq 2$ is unchanged by the path; $I \geq 2$ implies that J increases along the path, which in turn justifies $I*I \leq J$. The assertion $J \leq N$ follows from the condition $J \leq N$ in the path. Since MOD(J, I) = 0, we have MOD(J+I, I) = 0, which becomes MOD(J, I) = 0 at the end of the path. This leaves W(I, J-1), and the only difference between this at the begin ing of the path and at the end is that J has been increased by I. Since MOD(J, I) = 0, $\underline{x} = J$ satisfies $\underline{x} = I \cdot \underline{z}$ with $\underline{z} \geq I$ since $I*I \leq J$, but since we do not have $x \leq J-1$ for this $\underline{x}$ at the beginning of the path, this does not figure in W(I, J-1) there. When J is increased by I, this is the only value of $\underline{x}$ for which A($\underline{x}$) changes in the description of W(I, J-1); and in the path we indeed set A(J) = .FALSE.. As before, the reference to A(J) may be made because $4 \leq J$ and $J \leq N$.

Path 7

```
* I ≥ 2, I*I ≤ J, J ≤ N, MOD(J, I) = 0, W(I, J-1)
      A(J) = .FALSE.
      J = J + I
      (J > N)
      I = I + 1
* I ≥ 2, (I-1)*(I-1) ≤ N, V(I-1)
```

Here $I \geq 2$ since I is increased along the path, and $(I-1)*(I-1) \leq N$ follows from $I*I \leq N$ at the beginning of the path and the fact that I has increased by 1. The condition V(I) is the same as W(I, N), and this follows from W(I, J-1) at the beginning of the path and the fact that A(J) is set to .FALSE. (just as in the preceding path) and then later J > N. Since I is increased by 1, V(I) becomes V(I-1) again. The reference to A(J) follows in this path in the same way that it does in the previous one. This completes the proof of partial correctness.

1-11 Multiple Controlled Expressions

Now we shall derive some rules for proving termination of a program which has more than one loop. Using these rules, we shall be able to prove the termination, and hence the correctness, of the sieve program of the preceding chapter. In the derivation of these rules, we shall use this program as an example.

It is not difficult to see that, even in the general case, we may restrict our attention to the loop control paths, i. e., those control paths which are part of some closed loop. In our example program, there are seven control paths, and four of these, namely paths 2, 5, 6, and 7, are loop control paths. We shall be seeking rules which generalize those of section 1-5, and thus, for each loop control path, we shall be looking for an expression whose value increases along that path and is bounded at the end of it. The quantities, in this program, which play the rôle of controlled variables are quite obviously I and J; we see that I increases along paths 2 and 7, while J increases along paths 6 and 7. Also, I is bounded at the end of paths 2 and 7, while J is bounded at the end of path 6. Neither I nor J increases along path 5, although we may note that path 5 at least "progresses" in one sense: it does not contain a branch backward.

Suppose we were simply to show that every loop control path in a given program has a corresponding expression which increases along it, and is bounded at the end of it. Is this sufficient to show that the program terminates? The answer is no, as may easily be seen by considering the following program:

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

Considering the numbered statements as control points, I always increases along the first control path, and J along the second; also I and J are bounded everywhere. Yet the program never terminates.

We might say that in this case I increases along the first loop control path, but J decreases along it, and <u>vice versa</u> for the second loop control path. This, however, is not the real reason why this program does not terminate, because exactly the same behavior is shown by the program

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

which does terminate. What is the difference between these two programs, from the point of view of loop control paths? It is this: in the first case the two loop control paths are joined together; in the second, they are not. When two such paths are physically separate, it does not matter what each of them does to the controlled expression of the other. This means, in particular, that path 2 in our sieve program may be considered on an entirely separate basis from the other loop control paths of that program.

It has the controlled expression I, which, as we have seen, is increasing along it and is bounded at the end of it.

Let us now consider what happens when two or more paths are joined together, as our paths 5, 6, and 7. Here we would say intuitively that we have an outer loop and an inner loop. Path 6 is the inner loop, and paths 5 and 7 together are the outer loop. The variable I controls the outer loop, and the variable J controls the inner loop. The variable I is not changed at all in the inner loop; but the variable J is decreased in the outer loop, namely along path 5. It turns out that this behavior is characteristic of inner and outer loops, as we now show by considering a program with three levels of loops:

```
      I = 1
1     J = 1
2     K = 1
3     A(I, J, K) = 0
      K = K + 1
      IF (K .LE. N) GO TO 3
      J = J + 1
      IF (J .LE. N) GO TO 2
      I = I + 1
      IF (I .LE. N) GO TO 1
```

Under the assumption that statement number 3 is the only control point, we have three loop control paths. Along the innermost one, K is increased, and I and J remain constant. Along the outermost, I is increased; J and K are also increased and then later decreased (when they are set to 1). In general, in an outer loop control path, it does not matter what happens to the controlled expressions of inner loops. An inner loop control path, however, is presumably inside some outer loop, and the controlled expression of the outer loop must not be allowed to decrease. This observation is further strengthened by considering the "middle" loop in our last example. Here J increases, while I remains constant but K decreases. The J loop is "inner" to the I loop, but "outer" to the K loop.

Suppose that we have determined all of the inner-outer relationships which exist among the loops of a given program. Then every loop control path is completely inside $\underline{k}$ levels of nested loops, for some $\underline{k}$. If these have the associated controlled expressions $e_1, \ldots, e_k$, then the values of <u>all</u> these expressions must be nondecreasing along this path. Also, any two loops which do not have an inner-outer relationship must be physically separate. Suppose, in particular, that a second loop control path is completely inside $\underline{m}$ levels of nested loops, having the associated controlled expressions $e'_1, \ldots, e'_m$. Then we must have $e_i = e'_i$ for $1 \leq \underline{i} \leq \underline{\min}(\underline{k}, \underline{m})$. If $\underline{k} > \underline{m}$, then the first path is inner to the second; if $\underline{k} < \underline{m}$, then the second path is inner to the first; and if $\underline{k} = \underline{m}$, the two paths are "in the same loop." In our sieve program, paths 2, 5, and 7 are each inside one level of loop, with the controlled expression I, whereas for path 6 we have $\underline{k} = 2$, $e_1 = I$, and $e_2 = J$. In the last example above, the outermost loop is associated with I, the next with I and J, and the innermost with I, J, and K.

There remains the problem of discriminating "primary" paths such as paths 2, 6, and 7 in our sieve program, along which controlled expressions increase, from "secondary" paths such as path 5, along which they do not. We recall that path 5 contains no branch backward; this seems to suggest that the "program counter" of this program (i. e., the "variable" whose "value" is the number of the statement currently being executed) is a controlled expression. In fact, this expression certainly increases along path 5, and it is always bounded. As we gain more experience with proofs of this kind, however, we will see that this use of the location counter, while handy for short programs, becomes increasingly difficult as programs get longer, especially if the controlled expressions involved are increased at the beginning or the middle of a loop rather than at the end.

The real distinction between a primary and a secondary control path is this: every closed loop containing a secondary control path also contains a primary path in the same loop, or some outer-level path. Thus, in our sieve program, every closed loop which contains path 5 must also contain path 7. Both of these paths are in the I loop only, and I does not decrease along either of them. Path 7 is a primary path; I increases along it, and is bounded at the end of it. Neither of these things need be proved for a secondary path, such as path 5; all we need to know is that it is "in the same loop" -- irrevocably -- as some primary path.

1-12 The Termination Theorem

We shall now state and prove, in an informal style, a theorem which embodies our termination rules.

A program terminates (when started with its initial assertion valid) if we may associate with each of its loop control paths a sequence of expressions with integer values, subject to the following rules:

(1) The value of each expression associated with a path must never be strictly less at the end of the path (if started with its initial assertion valid) than it was at the beginning of that path;

(2) For any two paths such that one of them begins where the other one ends, one of the two must be associated with the expressions $e_1, \ldots, e_j$, with $j \leq k$, where the other is associated with $e_1, \ldots, e_k$;

(3) If a path is associated with $e_1, \ldots, e_k$, then either:

(a) the value of e_k must be strictly greater at the end of this path (when started with its initial assertion valid) than it was at the beginning, and the final assertion of this path must include a bound on e_k which cannot change along any path whose associated expressions are $e_1, \ldots, e_m$, with $k \leq m$, or

(b) every closed loop in the program which contains this path must contain a path with property (a) above which is associated with $e_1, \ldots, e_k$, or any path which is associated with $e_1, \ldots, e_j$ for some $j < k$.

As indicated in the preceding section, the expressions $e_1, \ldots, e_k$ which are associated with a given path normally correspond to the k levels of nested loops within which that path is contained.

The expression e_1 controls the outermost of these loops, and the expression e_k the innermost. We have purposely omitted any specific reference to primary and secondary control paths, and to inner and outer loops, in our definition. However, it is clear that control paths satisfying condition 3(a) are primary, in the sense of the preceding chapter, while those satisfying condition 3(b) are secondary. Also, if one path is associated with e_1, ..., e_j, with $j < k$, while another is associated with e_1, ..., e_k, then the first is inner to the second, or the second is outer to the first. We shall employ these terms even when $j = k$, and shall speak of the second path being <u>strictly</u> <u>outer</u> to the first if $j < k$.

Two other points about our termination theorem are worth mentioning. The condition that one path begins where another one ends is not the only way that two paths can have a statement in common, and thus fail to be physically separate; however, it is the only one needed here. Also, the restriction as to where the bounds on controlled expressions can change is quite weak; these bounds may, in fact, be changed in any place outside the loop which is controlled by the given expression (and any of its inner loops).

Before we prove our theorem, let us apply it to our sieve program. With control paths 2, 5, and 7, as mentioned in the previous chapter, we associate the controlled expression I; with path 6, we associate I and J. Then rule 1 is satisfied, and also rule 2 (note that rule 2 would still have been satisfied if, for example, we had used J in path 2 rather than I). The innermost controlled expression is I for paths 2 and 7, and J for path 6; in each of these cases, rule 3(a) is satisfied, and in fact N never changes in the entire algorithm. Path 5, as we have noted before, is secondary to path 7. Thus the sieve program terminates.

We note, in connection with our sieve program, that if we consider only the statements numbered 1, 3, and 4 as control points, then every closed loop in this program still must pass through one of these points. With respect to the new set of control points, there are still seven control paths, but only three of them are loop control paths. In general, although letting all the labelled statements of a program be control points is a safe way of getting a sufficient collection of them, we may often eliminate some of them in this way. This normally makes the termination proof shorter, although it may make the proof of partial correctness longer.

The proof of our theorem now proceeds as follows. Suppose the contrary, and let us start the program for initial conditions (implied in some way by the initial assertion) in such a way that it never terminates. We have mentioned before the fact that our program contains only a finite number of control paths. Therefore, some of these paths must be taken infinitely often. Among all such paths, we choose one (call it π) which is associated with the controlled expressions $e_1, \ldots, e_k$, where k is as small as possible. Thus all paths associated with a smaller number than k expressions are taken only a finite number of times; similarly, all non-loop control paths in the program are taken only a finite number of times (either zero or one). Hence we may run our program until each of these paths has been taken for the last time. Thus, from now until infinity, the only control paths taken will be paths which are associated with at least k controlled expressions.

We may now assume that π is primary. (For the purposes of the remainder of the proof, we use the terms primary, secondary, inner, outer, and strictly outer as indicated in the discussion above.)

For if π is secondary, then any closed loop containing π must contain another path which is outer to π, and this path cannot be strictly outer, or else we would have a contradiction to our choice of k. But rule 3 now specifies that this new path is primary. Of all such new paths, we may choose an arbitrary one which is taken an infinite number of times; this is always possible, because π itself is taken an infinite number of times. We now simply call this new path π; it has the same associated controlled expressions as the original π.

Now we continue the program until π is taken, and note the value of e_k at the end of it; denote this integer value by v_1. Since π is primary, e_k is bounded at the end of π by a quantity which we call b; thus $v_1 \leq b$. We now show that, from now until infinity, the only control paths taken are inner to π. Suppose the contrary, and let π'' be the first control path taken (after this point) which is not inner to π. Let π' be the path immediately preceding π'' in the flow of control (possibly $\pi' = \pi$). Then π'' begins where π' ends, and therefore, by rule 2, π'' is either inner or outer to π'. If π'' is inner to π', then, since π' is inner to π, we would have π'' inner to π, a contradication. Therefore π'' is outer to π'. But π is also outer to π', and therefore π' must be either inner or outer to π. Since it is not inner, it must be strictly outer, and we have a contradiction to our choice of k.

Rule 3(a) now tells us that the maximum value of e_k does not change along any path which is inner to π, and thus, by the preceding paragraph, this maximum value will be equal to b from now until infinity. Furthermore, for the same reason, the controlled expression e_k can never decrease from now until infinity. We now recall that the path π is taken an infinite number of times. If we run the program until π is taken again, then we may denote the value of e_k at this point by v_2. We have $v_2 \leq b$ as before, and $v_1 < v_2$ since the value of e_k can never decrease and must be strictly increasing along the path π. Continuing in this way we obtain a strictly increasing sequence of integers $v_1 < v_2 < \ldots$, each of which is less than or equal to b, which is impossible. This final contradiction completes the proof.

PROBLEMS

1. For each of the following programs, state a final assertion of the form X = e, where e is an expression which involves only variables whose values are not changed by the given program. Assume in each case that there is no initial assertion.

(a)
```
C = B * A
D = C * C
X = D - C
```

(b)
```
A = 6
B = A * A
C = B - A - 28
X = C * C * C
```

(c)
```
X = A + B
C = X * 5
X = X + X
C = X - D
X = C / X
```

2. Consider the statement X = 5 - X as a one-statement program. In each of the following cases, give a final assertion for this program with the given initial assertion.

(a) $X = 4$

(b) $X < 0$

(c) $X < Y$

(d) $Y > Z$

3. Initial and final assertions are given for the following program. Fill in the intermediate assertions:

```
* B² ≥ 4AC, A ≠ 0
       U = B * B
       V = A * C
       W = 4 * V
       U = U - W
       V = √U
       W = 2 * A
       U = V - B
       X = U / W
* X = (-B+√(B²-4AC))/2A
```

4. Initial and final assertions are given for the following program. Fill in the intermediate assertions:

```
* A < B < F < C < D < E
       T = F
       F = E
       E = D
       D = C
       C = T
* A < B < C < D < E < F
```

5. Fill in the intermediate assertions in the following branch-forward program written in FORTRAN IV:

```
* I > 0, I < 4
       IF (I .EQ. 1) GO TO 11
       IF (I .EQ. 2) GO TO 12
```

```
         J = 9
         GO TO 13
   11    J = 1
         GO TO 13
   12    J = 4
   13    CONTINUE
   * J = I2
```

6. (a) Consider all branch-forward programs having n statements, each of which requires an amount of time t to be executed. What is the maximum time which can possibly be required to execute such a program?

(b) What is the maximum value, over all such programs in which every statement except the last is a conditional transfer statement, of the minimum time required to execute the program?

7. Consider the following exponentiation program, slightly modified from the one given in section 1-4:

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

Write out all assertions and prove that this program is partially correct with respect to them, as is done in the text. The final assertion, as before, should be $X = A^N$; the intermediate assertions, however, will have to be quite different, as will the initial assertion (what happens when N = 0?).

8. (a) Show that for every program which has no closed loops, there exists a statement with the property that no other statement in the program immediately precedes it. (Hint: You may construct a closed loop directly if such a statement does not exist.)

(b) Use the result above to show that every statement in a program with no closed loops may be given a number, such that the program becomes a branch-forward program if "forward" refers to increasing statement numbers. (Hint: Use induction on the total number of statements in the program.)

9. Consider the following exponentiation program:

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

Assume that the assertion at statement number 2 is $X = A^N$.

(a) With the initial assertion $N \geq 1$, this program is partially correct. Is it partially correct without that assertion? (Be careful. What, exactly, happens in this program when $N \leq 0$?)

(b) Prove the partial correctness of this program, using only the final assertion given above, plus an initial assertion and an assertion at statement number 1.

10. Consider the following program:

```
        A = B(X)
1       IF (A .EQ. 0) GO TO 4
        C = D(X)
2       IF (C .EQ. 0) GO TO 5
        IF (C .EQ. 1) GO TO 4
3       A = A - 1
        GO TO 5
4       C = C - 1
        GO TO 2
5       E = F(X)
        IF (E .EQ. 0) GO TO 1
```

In each of the following cases, state whether the given statements, together with the beginning and end of this program, form a sufficient collection of control points (see section 1-5).

(a) Statements 1, 4, and 5.

(b) Statements 1, 3, and 5.

(c) Statement 4 alone.

(d) Statement 2 alone.

11. Consider what would happen in the exponentiation program of section 1-4 if we had no initial assertion. Clearly the program would sometimes go into an endless loop. Why is I not a controlled expression in this case?

12. A very common programming error made by beginners which results in endless looping is to include the initialization of the index within the loop. Suppose that we did this in the exponentiation program of section 1-4 by changing the statement numbers so that the first statement of the program is statement number 1. Why is I not a controlled expression in this case?

13. Prove the partial correctness of the following program (slightly modified from that given in section 1.7):

```
* N ≥ 2
          I = 1
          S = A(1)
1         I = I + 1
          S = S + A(I)
          IF (I .LT. N) GO TO 1
```

$$* \; S = \sum_{x=1}^{N} A(x)$$

14. Prove the correctness of the above program.

15. Prove the correctness of the following program (slightly modified from that given in section 1.8):

```
** C(0) = .TRUE.
** C(K) = C(K-1) .AND. (A(K) = B(K))
* N ≥ 1
          I = 0
1         I = I + 1
          B(I) = A(I)
          IF (I .NE. N) GO TO 1
* C(N)
          CONTINUE
```

16. Prove the correctness of the following program (slightly modified from that given in section 1.8):

```
** F(0) = .TRUE.
** F(K) = F(K-1) .AND. (X ≠ A(K))
* N ≥ 1
          I = 0
```

```
1      I = I + 1
       IF (X .EQ. A(I)) GO TO 3
       IF (I .NE. N) GO TO 1
* F(N)
2      CONTINUE
       . . .
* F(I-1), X = A(I)
3      CONTINUE
```

17. The following program, given with assertions, is a modification of that given in problem 15 above:

```
** C(0) = .TRUE.
** C(K) = C(K-1) .AND. (A(K) = B(K))
* N ≥ 1
       I = 0
1      I = I + 2
       B(I-1) = A(I-1)
       B(I) = A(I)
       IF (I .NE. N) GO TO 1
* C(N)
       CONTINUE
```

The point of this modification, of course, is that it takes the computer just as long to do I = I + 1 as I = I + 2, and we need to do I = I + 2 only half as many times as we would need to do I = I + 1. There is, however, a bug in the above program. Try to prove the correctness of this program and show exactly where such a proof breaks down.

18. Fix the bug in the program of the above problem, while retaining the statement I = I + 2 and the property of it mentioned

above. Then prove the correctness of the resulting program.

19. Consider the following program:

```
* N ≥ 0
        M = N
        X = 1
        I = 1
3       IF (I .GT. N) GO TO 5
        I = 2*I
        GO TO 3
5       IF (I .EQ. 1) GO TO 7
        I = I/2
        X = X*X
        IF (I .GT. M) GO TO 5
        X = X*A
        M = M - I
        GO TO 5
* X = A^N
7       CONTINUE
```

This program calculates exponentials for large values of N much more efficiently than the previous exponentiation programs do. Consider this program as being run with N = 13.

(a) What will be the value of I when the program first passes statement number 5?

(b) How many times does the program pass statement number 5?

(c) When the program passes statement number 5 for the third time, what will be the values of I, X, M, and N? Can you see a relationship which holds between I, j, M, and N, where $X = A^j$? Does this relationship still hold when the program passes statement number 5 for the fourth time? For the first time?

20. In the program above, consider that the assertion $X^I = A^{N-M}$ has been inserted before statement number 5.

(a) Show that with this assertion only, none of the paths which start at statement number 5 is necessarily valid.

(b) Add the assertion that I is a power of 2, i. e., $I = 2^J$ for some $J \geq 0$, at statement number 5. Show that with these assertions only, the path from statement number 5 to statement number 7 is not necessarily valid, but the other two paths which start at statement number 5 are valid.

(c) Add the assertions $I > M$ and $M \geq 0$ at statement number 5. Show that all three paths starting from statement number 5 are now valid.

(d) Complete the proof of partial correctness of this program.

21. Consider a control path in a program from a certain control point to itself. Suppose we have an expression which is bounded at the given control point. Suppose also that there exists another statement in this control path, such that the given expression is increasing along the given control path if that path is considered as starting and ending at the new statement. Show that these conditions are <u>not</u> sufficient for the program to terminate, even if this control path is the only loop control path in the program. (Hint: Consider the program

```
1    J = J + 1
     I = J
     I = 1
     GO TO 1
```

as a counterexample.)

22. In all of our discussion of bounded expressions it is assumed that the bound does not itself change. It would seem, however, that it would do no harm for a bound to decrease. Show rigorously that this is true. (Note: This is not as tedious as it might appear. Let I be an increasing expression and let N be a decreasing bound on I. Consider the expression I-N. Generalize to arbitrary expressions.)

23. In the definition of a secondary path (as given in section 1-12) there are two possibilities for a closed loop containing such a path; it may contain a primary path in the same loop, or some outer-level path. Write a program which demonstrates that the second of these conditions is sometimes useful. (Hint: A linear search algorithm inside an outer loop will do nicely. The secondary path is the one along which the quantity being searched for is found.)

24. Prove the statement made in section 1-12 that the conditions of the termination theorem may be weakened so that the bound on a controlled expression may be arbitrarily changed in any place which is not on any control path controlled by that expression.

2 CORRECTNESS OF PROGRAMS: FURTHER TECHNIQUES

The basic principles given in the preceding chapter have still left us a long way from being able to prove the correctness of programs which we encounter in practice. The discovery of the proper assertions and controlled expressions which will prove a program correct requires a considerable amount of experience. Also, there are no mechanical methods which will invariably guide us to the construction of a proof of correctness. We may say that proving programs correct is an art, just as proving theorems in any other branch of mathematics is an art, even though, when a theorem is finally proved, a scientific fact has been established.

This chapter gives further techniques for proving the correctness of a program besides those of the preceding chapter. As before, all techniques are given in intuitive form, without the use of formal mathematical models. An important theme running throughout this chapter is the universality of our methods of proving correctness. In the preceding chapter, the programs which were proved were written in FORTRAN, and had a very simple structure. In this chapter, however, we will see how to prove programs using input-output or written in assembly language, and we will develop special techniques for programs which rearrange data. Even programs which modify themselves are subject, as we shall see, to the same sort of analysis that we have presented for other programs.

2-1 Using Mathematical Facts

When we are proving the correctness of a program which computes the value of a mathematical function, we may need to prove certain facts about this function for use in verifying the program. We saw an example of this in the program of section 1-10 to compute primes by the sieve method, specifically in path 4, where we effectively used the fact that any nonprime has a factor greater than 1 and less than or equal to its square root. A further example is provided by considering a modification of this program to improve its efficiency. Let us give the statement I = I + 1 between statements 3 and 4 the statement number 5, and let us add a new statement just before statement 3, as follows:

```
IF (.NOT. A(I)) GO TO 5
```

The point of this is that if I has already been "crossed out," then so have all its multiples. In particular, using this change, we no longer "cross out" the multiples of 4, 6, 8, 9, etc.; only multiples of primes will be crossed out, and in fact each nonprime should now be crossed out once and only once.

Let us see how the verification of this program is changed by the given modification. It is actually changed very little, mainly because the modification introduces no new assignment statements and because it merely eliminates waste motion, rather than causing things to be done in a different order. All of the assertions given with the program in section 1-10 still apply, and even the verifications of the control paths still hold. The only one of these paths, in fact, which is changed at all is path 5, and even here the proof still works because no new assignments are introduced. (We assume, of course, that statement number 5 is <u>not</u> a control point; and, in fact, it does not need to be.) There is one new path, which we shall call path 8, as follows:

```
* I ≥ 2, (I-1)*(I-1) ≤ N, V(I-1)
        J = I*I
        (J ≤ N)
        (A(I) = .FALSE.)
        I = I + 1
* I ≥ 2, (I-1)*(I-1) ≤ N, V(I-1)
```

Here I ≥ 2 is clearly preserved since I increases; also, (I-1)*(I-1) ≤ N is clear from the condition J ≤ N in the path, with J = I*I, followed by I = I + 1. The only difference between V(I-1) at the beginning of the path and at the end is that, at the end, we must have A($\underline{x}$) = .FALSE. if $\underline{x} = \underline{i}\cdot\underline{w}$, where $\underline{i} \leq \underline{w}$ and $\underline{i}$ is the value of I at the beginning of the path. It is in the verification of this fact that we must resort to a mathematical argument. We know that A($\underline{i}$) = .FALSE., and since V(I-1) -- that is, V($\underline{i}$-1) -- holds at the start of the path, we may infer that $\underline{i} = \underline{y}\cdot\underline{z}$ where $\underline{y} \leq \underline{z}$ and $2 \leq \underline{y} \leq \underline{i}-1$. But then if $\underline{x} = \underline{i}\cdot\underline{w}$ then $\underline{x} = (\underline{y}\cdot\underline{z})\cdot\underline{w} = \underline{y}\cdot(\underline{z}\cdot\underline{w})$, where $\underline{y} \leq \underline{z}\cdot\underline{w}$ (actually, $\underline{y} < \underline{z}\cdot\underline{w}$) and $2 \leq \underline{y} \leq \underline{i}-1$; and V($\underline{i}$-1) implies A($\underline{x}$) = .FALSE. already in this case. This completes the verification of path 8. The proof of termination of the modified program is the same as before, with the addition of the treatment of path 8; it is primary, and is associated with the controlled expression I.

We shall now give another example of the use of mathematical facts to prove correctness of programs. The following program is an extremely simple version of Euclid's algorithm for finding the greatest common divisor of two integers M and N:

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

This program is often very inefficient; for example, it always takes _n_ steps to determine that the greatest common divisor of _n_ and _n_-1 is 1. It works by successive subtraction; faster versions work by successive division and taking remainders, and terminate when a zero remainder is obtained. This version terminates when the two quantities I and J are equal. We have included an intermediate assertion; the point of it is that, as we continually decrease the values of I and J, their greatest common divisor remains unchanged.

The control paths of this program are as follows:

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

Follows by direct substitution.

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

Here we use our first mathematical fact about the GCD function -- namely, that the greatest common divisor of x and x is x.

```
* I > 0, J > 0, GCD(I, J) = GCD(M, N)
      (I < J)
      J = J - I
* I > 0, J > 0, GCD(I, J) = GCD(M, N)
```

The assertion $I > 0$ is preserved along the path since I is unchanged, and the assertion $J > 0$ follows from $I < J$ and $J = J - I$, but for the last assertion we need another mathematical fact. If $I > 0$, $J > 0$, and $J-I > 0$, then $GCD(I, J) = GCD(I, J-I)$. We may prove this by verifying that any positive integer dividing both I and J also divides J-I, and, similarly, any positive integer dividing I and J-I also divides J. Hence the greatest integer which divides both I and J must be the same as the greatest integer dividing both I and J-I.

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

This path is the same as the previous one with I and J reversed, provided that we know $GCD(I, J) = GCD(J, I)$. Thus it, like the previous one, is valid.

To complete the proof of correctness we need to identify the controlled expressions. This may be done in any of three ways. We may associate the expression -J with the third path above, and -J and -I with the fourth; here -J increases along the third path (because $I > 0$) and -I along the fourth (because $J > 0$), and each is bounded at the end of the corresponding path. Or we may associate simply -I with the fourth path, and -I and -J with the third; the arguments may then be constructed correspondingly. A third way to prove termination here is to construct a single controlled expression -(I+J); this expression must increase along each of the two loop control paths of the program, and is bounded at the end of each of these paths.

2-2 Input-Output

Up to now we have assumed that the specification of a program may be made entirely in terms of the variables with which it is concerned. It might seem that this is no longer true when a program accepts input or produces output. We shall, however, adopt a point of view according to which input files and output files are simply collections of variables of a special kind. Thus the general theory developed in the preceding chapter is immediately applicable to file processing programs as well.

There are always a number of different ways in which the same file may be regarded as a collection of variables. In a card file, for example, we may regard every column of every card as a separate variable whose values are the possible characters which may appear in that column. We may also, however, regard each hole (or, rather, each position at which a hole might appear) on each card as a variable whose values are 0 and 1; or, taking a more global view, we might consider a card as a variable whose values are the possible contents of that card. This last point of view usually turns out to be the easiest to work with, although there is a still more global viewpoint in which all of the information in an entire card deck is taken as the value of a single variable representing that deck. All of these viewpoints are equivalent, in a way which is intuitively obvious and which will be made precise in Chapter 3; the multiplicity of models for the same situation is derived from the hierarchical structure of card files. Similarly, we may regard output characters, lines, or entire listings as variables; and the same for tapes, tape files, tape records, and tape squares.

We shall restrict our attention for the moment to <u>sequential files</u>, which are processed from beginning to end. All card, tape, and printer files are sequential files, and so are the disk files which simulate them. Reading and writing a sequential file bear a number of similarities to the processing of an array from one end to the other; indeed, arrays which are processed in this way are often called "internal files." Let us consider, for example, the program of section 1-8 which moves the values in one array to another:

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

Here the arrays A and B are collections of <u>n</u> variables each, for some <u>n</u> not less than the value of N, and each of these variables may take (normally) any floating point value. In a program such as this we may easily substitute a deck of cards for the array A and a printer listing for the array B; the program then becomes one which lists a deck of <u>n</u> cards. Instead of floating point numbers, we have character strings as values of the A(<u>i</u>) and the B(<u>i</u>); the maximum length of these strings is the maximum number of characters which may appear in a punched card or a printed line respectively.

The trouble with this analogy as it stands is that there is no apparent counterpart of the index variable I. That such a counterpart is actually essential may be seen, for example, by considering the operation of reading a card. The question is, <u>which</u> card is to be read? This is the question which is answered for an internal file by reference to the index variable. At any time during the reading of a card file, one card in the deck will be the "current card" which has just been read, or which is to be read next. Since different cards become current at different times during the running of the program, each card may be considered as a value of the variable quantity "current card." This variable quantity could also be thought of as representing the card reader, and its current value as representing the current position of the input deck within the card reader. We shall refer to this quantity as the file position variable for the card reader. Similarly, there will be a file position variable for the printer, whose value tells us which line has just been printed, or is to be printed next.

Every sequential file has a position variable whose value determines which square, record, etc., is currently being read or written, and which is initialized by opening the file. The position variables of sequential input files perform certain other functions similar to those of the index variables of internal files. They may be tested to determine when a file process should terminate, and they may serve as controlled expressions in file processing programs. As an illustration, let us rewrite the program above in equivalent form, together with its file processing analogue:

```
         I = 1                          OPEN INPUT FILE
         J = 1                          OPEN OUTPUT FILE
   1     X = A(I)                 1     READ X
         I = I + 1
         IF (I .GT. N) GO TO 2          IF END OF INPUT GO TO 2
         B(J) = X                       PRINT X
         J = J + 1
         GO TO 1                        GO TO 1
   2     CONTINUE                 2     CONTINUE
```

The variables I and J are here the input and output file position variables respectively. Each statement at the right is the analogue of the statement or statements opposite it at the left.

When a program uses input-output instructions of a variety of types, the effect of each of these types on the file variables and file position variables must be specified if the program is to be proved correct according to our methods. An example of such specification will now be given for a simplified collection of input-output instructions. Let fn denote any file name, and let the position variable of the file named fn be denoted by pv.fn. This file is made up of file variables fv.fn(i); the variable fv.fn(pv.fn) is the one which has just been read or written. The values of pv.fn are assumed to be positive integers, of which the largest is denoted by len.fn; the values of the fv.fn(i) are assumed to be the same as the values of X. Note that, in opening the file, pv.fn must be initialized to zero, rather than 1; also, when we read or write, we perform the incrementation of the file position variable *before* the file reference. In the previous example, A(I) and B(J) denote the elements which are *about to be* read or written, rather than those which have just been read or written. Thus I, for example, assumes the final value N + 1. In the example below, however, the final value of the position variable is the length of the file. The meaning, for verification purposes, of each prototype statement at the left is given opposite it at the right.

OPEN FILE fn	pv.fn = 0
READ fn, X	pv.fn = pv.fn + 1, X = fv.fn(pv.fn)
IF (EOF, fn) THEN ...	IF pv.fn $\geq$ len.fn THEN ...
WRITE fn, X	pv.fn = pv.fn + 1, fv.fn(pv.fn) = X
END FILE fn	len.fn = pv.fn
REWIND fn	pv.fn = 0

BACKSPACE <u>fn</u>	<u>pv.fn</u> = <u>pv.fn</u> - 1
POSITION <u>fn</u>, N	<u>pv.fn</u> = N
POSITION FORWARD <u>fn</u>, N	<u>pv.fn</u> = <u>pv.fn</u> + N
POSITION BACKWARD <u>fn</u>, N	<u>pv.fn</u> = <u>pv.fn</u> - N

These definitions may be extended so as to allow reading of more than one variable at the same time and in various formats, or to assure that the positioning instructions, for example, do not "run off" either end of the file.

If a single variable is used to represent a file, its values are sequences $(x_1, \ldots, x_n)$, for various values of $\underline{n}$, where each x_i is a possible value of X above. The file position variable still has much the same function as before. When a file is processed in one direction only, there is another approach which eliminates the file position variable entirely; this consists of regarding the sequence $(x_1, \ldots, x_n)$ as representing those file elements which are still to be read on input, or which have been written on output. This approach fails, however, in more complex situations, such as when a file is read, rewound, and read again.

2-3 Permutations and Sortedness

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 $B(3) \leq B(4)$. 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 case above 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 $B(1) = A(\underline{x})$, for some value of $\underline{x}$ with $1 \leq \underline{x} \leq 4$; and similarly $B(2) = A(\underline{y})$, $B(3) = A(\underline{z})$, and $B(4) = A(\underline{w})$. Moreover, we must know that $\underline{x}$, $\underline{y}$, $\underline{z}$, and $\underline{w}$ are all distinct.

Let us refer to $\underline{x}$, $\underline{y}$, $\underline{z}$, and $\underline{w}$ as $\underline{f}(1)$, $\underline{f}(2)$, $\underline{f}(3)$, and $\underline{f}(4)$, respectively. Then $B(\underline{i})$ must be equal to $A(\underline{f}(\underline{i}))$, for $1 \leq \underline{i} \leq 4$. The values of the function $\underline{f}$ lie between 1 and 4 inclusive, and they are all distinct, which means that $\underline{f}$ is a <u>permutation</u> 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 $A(\underline{i}) = B(\underline{g}(\underline{i}))$, for

$g(1) = 2 \qquad g(2) = 4 \qquad g(3) = 1 \qquad g(4) = 3$

and $1 \leq \underline{i} \leq 4$. The function $\underline{g}$ is also a permutation; it is the <u>inverse</u> of $\underline{f}$, and $\underline{f}(\underline{g}(\underline{i})) = \underline{g}(\underline{f}(\underline{i})) = \underline{i}$ for $1 \leq \underline{i} \leq 4$.

Now suppose that, instead of placing our sorted results in a new array, we are sorting an array in place. How shall we express the assertion that the new values of the elements of our array are a rearrangement of their old values? What we must do here is exactly what we did in section 1-2: 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 $A(\underline{x}) = A0(\underline{x})$ for $1 \leq \underline{x} \leq N$, where the dimension of A is N. This is then our initial assertion, and then our final assertion says that $A(\underline{x}) = A0(\underline{f}(\underline{x}))$ for some permutation $\underline{f}$ of the integers 1 through N, and also that $A(\underline{x}) \leq A(\underline{x}+1)$ for each $\underline{x}$, $1 \leq \underline{x} < N$. It is to be noted that this subterfuge is necessary only when there is no other known array in the program, 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 initial assertion may simply be that each variable $A(\underline{x})$ has the same value in one place as it does in the other.

Let us denote by PERM(A, B, N) the assertion that $A(\underline{x}) = B(\underline{f}(\underline{x}))$, $1 \leq \underline{x} \leq N$, for some permutation $\underline{f}$ of the integers from 1 to N. If $A(\underline{x}) = B(\underline{x})$, $1 \leq \underline{x} \leq N$, then certainly PERM(A, B, N) holds, because we need only to set $f(\underline{x}) = \underline{x}$ for each $\underline{x}$; this is the <u>identity permutation</u>. Thus PERM(A, B, N) becomes an initial as well as a final assertion in a program to sort the array A. It is not surprising that, in many sort programs, it is also an intermediate assertion; that is, it remains true as the sort progresses. This is, in particular, true of the various interchange sort programs, in which an array is sorted by successive interchanging of various elements of it.

We now prove that PERM(A, B, N) is actually preserved by a properly chosen interchange; that is, we shall show that the path

```
* PERM(A, B, N), 1 ≤ I, I ≤ N, 1 ≤ J, J ≤ N
      T = A(I)
      A(I) = A(J)
      A(J) = T
* PERM(A, B, N)
```

is valid. This prototype situation may then be applied to a wide variety of interchange sort programs. Suppose that $A(\underline{x}) = B(\underline{f}(\underline{x}))$, $1 \leq \underline{x} \leq \underline{n}$, at the beginning of the above path, where $\underline{n}$ is the value of N. Let $\underline{g}$ be the permutation defined as follows:

$$\underline{g}(\underline{i}) = \underline{j}, \quad \underline{g}(\underline{j}) = \underline{i}, \quad \underline{g}(\underline{x}) = \underline{x} \text{ for } 1 \leq \underline{x} \leq \underline{n},\ \underline{x} \neq \underline{i}, \underline{j}$$

where $\underline{i}$ and $\underline{j}$ are the respective values of I and J (note that the values of I, J, and N are not changed in the above path). The function $\underline{h}$ defined by $\underline{h}(\underline{x}) = \underline{f}(\underline{g}(\underline{x}))$, $1 \leq \underline{x} \leq \underline{n}$, is then also a permutation. We shall prove that $A(\underline{x}) = B(\underline{h}(\underline{x}))$, $1 \leq \underline{x} \leq \underline{n}$, at the end of the path, thus verifying PERM(A, B, N) there. In fact, if $\underline{x} \neq \underline{i}, \underline{j}$, then $\underline{h}(\underline{x}) = \underline{f}(\underline{x})$, and $A(\underline{x}) = B(\underline{f}(\underline{x})) = B(\underline{h}(\underline{x}))$ at the beginning of the path; since $A(\underline{x})$ is not changed anywhere in the path, this relation still holds true at the end. This leaves only the cases $\underline{x} = \underline{i}$ and $\underline{x} = \underline{j}$, and since $\underline{h}(\underline{i}) = \underline{f}(\underline{j})$ and $\underline{h}(\underline{j}) = \underline{f}(\underline{i})$, we must show that if $A(\underline{i}) = B(\underline{f}(\underline{i}))$ and $A(\underline{j}) = B(\underline{f}(\underline{j}))$ at the beginning of our path, then $A(\underline{i}) = B(\underline{f}(\underline{j}))$ and $A(\underline{j}) = B(\underline{f}(\underline{i}))$ at the end of it. This is intuitively clear from the fact that A(I) and A(J) have been interchanged; but let us prove it formally. After the first assignment, we have $T = B(\underline{f}(\underline{i}))$ and $A(\underline{j}) = B(\underline{f}(\underline{j}))$. After the second, we have $T = B(\underline{f}(\underline{i}))$ and $A(\underline{i}) = B(\underline{f}(\underline{j}))$; finally, after the third, we have $A(\underline{j}) = B(\underline{f}(\underline{i}))$ and $A(\underline{i}) = B(\underline{f}(\underline{j}))$. The legitimacy of the operations on A(I) and A(J) and the existence of the permutation $\underline{g}$ follow from $1 \leq I \leq N$, $1 \leq J \leq N$.

Many rearrangement programs also pose interesting problems with regard to termination. It is quite common to write an interchange sort in such a way that it simply keeps looking for adjacent pairs of elements to interchange, stopping only when it cannot find any more. The fact that such procedures always terminate may be shown by introducing a concept called <u>sortedness</u>. If A is an array of $\underline{n}$ elements, the sortedness of A is

$$\sum_{1 \leq \underline{i} < \underline{j} \leq \underline{n}} (\underline{\text{if}}\ A(\underline{i}) \leq A(\underline{j})\ \underline{\text{then}}\ 1\ \underline{\text{else}}\ 0)$$

That is, it is the number of pairs $(A(\underline{i}), A(\underline{j}))$ which are in order, i. e., $\underline{i} < \underline{j}$ and $A(\underline{i}) \leq A(\underline{j})$. Its maximum value is $\frac{\underline{n(n-1)}}{2}$, and it takes this value if and only if A is sorted. If its value is zero, A is sorted in reverse order. The statement $A(\underline{i}) \leq A(\underline{j})$ in the definition of sortedness does no imply that these have to be integers or real numbers; any other simple ordering may be substituted, the definition then depending upon the ordering.

The sortedness of an array is an integer expression, and we would like to be able to treat it as a controlled expression. Clearly it is always bounded, since its value can never be greater than $\frac{\underline{n(n-1)}}{2}$. We shall now prove, for the prototype control path of the preceding section, that *the sortedness of the given array is always greater 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 $(\underline{i}, \underline{j})$, which was out of order, is placed in order by the given operation; this increases by one the number of pairs in order. We complete the proof by showing that for any pair $(\underline{i}, \underline{x})$ which is in order and is put out of order by the interchange of $\underline{i}$ and $\underline{j}$, the pair $(\underline{j}, \underline{x})$ will be out of order and will be placed in order; a similar statement may then be proved if we start with a pair-$(\underline{j}, \underline{x})$ which is put out of order, and hence there can be no further net decrease in the sortedness. Let $(\underline{i}, \underline{x})$ be in order; then either $\underline{x} < \underline{i}$ and $A(\underline{x}) \leq A(\underline{i})$, or else $\underline{i} \leq \underline{x}$ and $A(\underline{i}) \leq A(\underline{x})$. In the first instance, $\underline{x} < \underline{j}$ since $\underline{i} < \underline{j}$, and so $(\underline{i}, \underline{x})$ cannot be put out of order by the transposition. In the second, in order for $(\underline{i}, \underline{x})$ to be put out of order, we must have $\underline{x} < \underline{j}$. By hypothesis, $A(\underline{i}) > A(\underline{j})$, and since $A(\underline{i}) \leq A(\underline{x})$ we must have $A(\underline{x}) > A(\underline{j})$. But this shows that $(\underline{j}, \underline{x})$ is out of order before the transposition, and in order after it. As mentioned before, a similar argument holds in reverse, and the proof is thus complete.

2-4 Sort Routines

We now apply the results of the preceding chapter to the proof of correctness of a typical interchange sort routine:

```
      ** ASC(1) = .TRUE.
      ** ASC(K) = ASC(K-1) .AND. (A(K-1) ≤ A(K))
      ** INIT(0) = .TRUE.
      ** INIT(K) = INIT(K-1) .AND. (A(K) = AO(K))
      * N ≥ 2, INIT(N)
            I = 1
      * 1 ≤ I, I < N, PERM(A, AO, N), ASC(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
      * PERM(A, AO, N), ASC(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 each pair (A(<u>x</u>), A(<u>x</u>+1)) is in order for <u>x</u> < I; this condition is denoted by ASC(I). If the routine finds that the pair (A(I), A(I+1)) is in order, then I is increased by 1; if this pair is out of order, it is interchanged and I is decreased by 1. When the program gets to the end of the array, all pairs must be in order.

At the start of the routine, $A(\underline{x}) = AO(\underline{x})$ for all $\underline{x}$; this condition is denoted by INIT(N). The assertion PERM(A, AO, N) was defined and discussed in the preceding chapter. It is clearly preserved here throughout the algorithm; that is, each time we make an interchange, we still have some rearrangement of our original data. The assertion $I < N$, of course, implies $I+1 \leq N$; this and $1 \leq I$ justify the references made to A(I) and A(I+1) at various points in the algorithm.

The control paths are as follows:

<u>Path 1</u>

```
* N ≥ 2, INIT(N)
      I = 1
* 1 ≤ I, I < N, PERM(A, AO, N), ASC(I)
```

Here $1 \leq I$ follows from $I = 1$; $I < N$ follows from $I = 1$ and $N \geq 2$; and ASC(I) is ASC(1), which is always true. The fact that PERM(A, AO, N) is implied by INIT(N) in this form follows from our discussion of the identity permutation in the preceding section.

<u>Path 2</u>

```
* 1 ≤ I, I < N, PERM(A, AO, N), ASC(I)
      (A(I) ≤ A(I+1))
      I = I + 1
      (I ≠ N)
* 1 ≤ I, I < N, PERM(A, AO, N), ASC(I)
```

The inequality $1 \leq I$ is preserved because I increases. The initial assertion $I < N$ becomes $I+1 < N$ or $I \leq N$, and since $I \neq N$ in the path, we must have $I < N$. The assertion PERM(A, AO, N) is unchanged by this path. Finally, ASC(I) at the beginning of the path, together with $A(I) \leq A(I+1)$, give ASC(I+1), and then I is increased by 1, giving ASC(I) again.

<u>Path 3</u>

```
* 1 ≤ I, I < N, PERM(A, AO, N), ASC(I)
      (A(I) ≤ A(I+1))
      I = I + 1
      (I = N)
* PERM(A, AO, N), ASC(N)
```

Again PERM(A, AO, N) is unchanged by the path, and, just as in the previous path, ASC(I) holds at the end; since I = N, this becomes ASC(N).

<u>Path 4</u>

```
* 1 ≤ I, I < N, PERM(A, AO, N), ASC(I)
      (A(I) > A(I+1))
      T = A(I)
      A(I) = A(I+1)
      A(I+1) = T
      (I = 1)
* 1 ≤ I, I < N, PERM(A, AO, N), ASC(I)
```

The assertions 1 ≤ I and I < N are unchanged in this path. The fact that PERM(A, AO, N) is preserved in a sequence like this follows from the discussion in the preceding section (note that I < N implies I+1 ≤ N at the beginning of the path). Finally, ASC(I) is ASC(1), which is always true.

<u>Path 5</u>

```
* 1 ≤ I, I < N, PERM(A, AO, N), ASC(I)
      (A(I) > A(I+1))
      T = A(I)
      A(I) = A(I+1)
      A(I+1) = T
```

```
        (I ≠ 1)
        I = I - 1
    * 1 ≤ I, I < N, PERM(A, AO, N), ASC(I)
```

Here $1 \leq I$ because clearly at the beginning of the path $2 \leq I$ (since $1 \leq I$ and $I \neq 1$), and then I is decreased by 1. The inequality $I < N$ is preserved since I decreases. PERM(A, AO, N) is preserved here just as it was in the preceding path. Finally, let us write ASC(I) at the beginning of the path as ASC(I-1) .AND. (A(I-1) ≤ A(I)). Then ASC(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, ASC(I-1) becomes ASC(I) again. This completes the proof of partial correctness.

To prove termination we need only consider the loop control paths, namely paths 2, 4, and 5. In paths 4 and 5, as we have noted in the preceding section, the sortedness increases. This fact is not changed, of course, by the introduction of conditions or assignments involving I after the interchange has been made. In path 2, the sortedness is constant and I increases and is bounded at the end of the path. The sortedness, of course, is always bounded. Thus the given sort program is correct.

2-5 Merging Routines

A different kind of rearrangement of data takes place in a routine which merges two sorted arrays to produce a single sorted array. Whereas rearrangement of data in place corresponds to a permutation, rearrangement of data out of place corresponds to a more general mapping from one set of variables to another.

In a simple merging routine, a sorted array of length M is being merged with another of length N to form a third sorted array of length M+N. Three pointers are kept, which we shall call I, J, and K. The first i-1 elements of the first array and the first j-1 elements of the second have already been merged to form the first k-1 elements of the result, where i, j, and k are the respective values of I, J, and K. The rearrangement implied by this statement will be denoted by MERGE(I-1, J-1, K-1); this assertion says that there exists a certain function f from variables to variables, such that each variable v in the domain of f has the same current value that $f(v)$ does. The range of f consists of the first k-1 elements of the resulting array, while its domain consists of the first i-1 elements of the first array together with the first j-1 elements of the second. (We could have constructed a similar function for PERM(A, AO, N); its domain and range would have consisted of the variables in the arrays A and AO respectively.)

In order to determine the kth element of the new array, we look at the ith element of the first array and the jth element of the second; the smaller of these two, in the given order, becomes the new kth element, and K is increased by 1. In addition, if the smaller element came from the first array, then I is increased by 1; otherwise, J is increased by 1. Taking into account the special cases in which we are completely finished with either one of the original two arrays, we may write our program as follows:

```
** ASC(Z, 1) = .TRUE.
** ASC(Z, K) = ASC(Z, K-1) .AND. (Z(K-1) ≤ Z(K))
* ASC(A, M), ASC(B, N), M > 0, N > 0
        I = 1
        J = 1
        K = 0
* 1 ≤ I, I ≤ M, 1 ≤ J, J ≤ N, ASC(A, M), ASC(B, N),
* MERGE(I-1, J-1, K), K = I + J - 2,
* (K = 0 .OR. (K > 0, C(K) ≤ A(I), C(K) ≤ B(J), ASC(C, 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
* 1 ≤ I, I ≤ M, J = N + 1, ASC(A, M),
* MERGE(I-1, J-1, K), K = I + J - 2,
* K > 0, C(K) ≤ A(I), ASC(C, 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
```

```
* I = M + 1, 1 ≤ J, J ≤ N, ASC(B, N)
* MERGE(I-1, J-1, K), K = I + J - 2,
* K > 0, C(K) ≤ B(J), ASC(C, K)
4      K = K + 1
       C(K) = B(J)
       J = J + 1
       IF (J .LE. N) GO TO 4
* MERGE(M, N, M+N), ASC(C, M+N)
5      CONTINUE
```

The result of this routine is that the values in the new array are some arrangement of the values in the two original arrays, and also that they are sorted. We denote these two assertions by MERGE(M, N, M+N) and ASC(C, M+N) respectively.

As the values of I, J, and K increase, the function whose existence is implied by MERGE(I-1, J-1, 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:

```
* 1 ≤ I, I ≤ M, 1 ≤ J, J ≤ N, ASC(A, M), ASC(B, N),
* MERGE(I-1, J-1, K), K = I + J - 2,
* (K = 0 .OR. (K > 0, C(K) ≤ A(I), C(K) ≤ B(J), ASC(C, K)))
       K = K + 1
       (A(I) ≥ B(J))
       C(K) = B(J)
       J = J + 1
       (J ≤ N)
* 1 ≤ I, I ≤ M, 1 ≤ J, J ≤ N, ASC(A, M), ASC(B, N),
* MERGE(I-1, J-1, K), K = I + J - 2,
* (K = 0 .OR. (K > 0, C(K) ≤ A(I), C(K) ≤ B(J), ASC(C, K)))
```

This path starts at statement number 1 and returns there. At the beginning of the path, we have MERGE($\underline{i}$-1, $\underline{j}$-1, $\underline{k}$), where $\underline{i}$, $\underline{j}$, and $\underline{k}$ are the current values of I, J, and K. The domain of the corresponding function $\underline{f}$ includes the first $\underline{i}$-1 elements of A and the first $\underline{j}$-1 elements of B. Since J increases by 1 along the path, the domain of the new function, which we shall call $\underline{f}$', includes the domain of $\underline{f}$ plus the element B($\underline{j}$). Likewise, the range of $\underline{f}$ includes the first $\underline{k}$-1 elements of C, while the range of $\underline{f}$' consists of the range of $\underline{f}$ together with C($\underline{k}$). We now construct $\underline{f}$' by simply setting $\underline{f}$'(B($\underline{j}$)) = C($\underline{k}$) and $\underline{f}$'($\underline{v}$) = $\underline{f}$($\underline{v}$) for $\underline{v} \neq$ B($\underline{j}$). Since $\underline{v}$ and $\underline{f}$($\underline{v}$) have the same values, so do $\underline{v}$ and $\underline{f}$'($\underline{v}$) for $\underline{v} \neq$ B($\underline{j}$), while B($\underline{j}$), which is the only element of B whose value changes, receives a new value equal to that of f'(B($\underline{j}$)). Hence the new function $\underline{f}$' preserves the property characteristic of $\underline{f}$ which is necessary for MERGE(I-1, J-1, K) to be true at the end of this path.

The other assertions in this path may be seen to follow by arguments which we have met before. We shall give these arguments now, partly as an example of how to handle alternatives (.OR.) in assertions. In general, any final assertion may be validated using one alternative appearing in the initial assertion, but it must then be re-verified using the other one. Clearly, when the <u>final</u> assertion contains alternatives, only one of them need be verified, although both of them may be under differing circumstances. In this case, we actually will never have K = 0 at the end. In verifying K > 0 at the end, we must treat separately the cases K = 0 and K > 0 at the beginning. In both cases, the argument is clear since K is increased by 1 along the path.

At the beginning of this path, we have MERGE($\underline{i}$-1, $\underline{j}$-1, $\underline{k}$), where $\underline{i}$, $\underline{j}$, and $\underline{k}$ are the current values of I, J, and K. The domain of the corresponding function $\underline{f}$ includes the first $\underline{i}$-1 elements of A and the first $\underline{j}$-1 elements of B. Since I increases by 1 along the path, the domain of the new function, which we shall call $\underline{f}'$, includes the domain of $\underline{f}$ plus the element A($\underline{i}$). Likewise, the range of $\underline{f}$ includes the first $\underline{k}$-1 elements of C, while the range of $\underline{f}'$ consists of the range of $\underline{f}$ together with C($\underline{k}$). We now construct $\underline{f}'$ by simply setting $\underline{f}'$(A($\underline{i}$)) = C($\underline{k}$) and $\underline{f}'$($\underline{v}$) = $\underline{f}$($\underline{v}$) for $\underline{v} \neq$ A($\underline{i}$). Since $\underline{v}$ and $\underline{f}$($\underline{v}$) have the same values, so do $\underline{v}$ and $\underline{f}'$($\underline{v}$) for $\underline{v} \neq$ A($\underline{i}$), while A($\underline{i}$), which is the only element of A whose value changes, receives a new value equal to that of $\underline{f}'$(A($\underline{i}$)). Hence the new function $\underline{f}'$ preserves the property characteristic of $\underline{f}$ which is necessary for MERGE(I-1, J-1, K) to be true at the end of this path.

The other assertions in this path may be seen to follow by arguments which we have met before. The assertions J = N + 1 and ASC(A, M) are unchanged by the path, while $1 \leq$ I and K > 0 are preserved because I and K increase along the path. The assertion I $\leq$ M follows from the condition I $\leq$ M in the path, and the assertion K = I + J - 2 is preserved since both I and J have been increased by 1. The assertion ASC(A, M) at the beginning of the path, together with 1 $\leq$ I and I < M (from I+1 $\leq$ M later on), implies A(I) $\leq$ A(I+1) and thus C(K) $\leq$ A(I) at the end of the path, and also C(K-1) $\leq$ C(K), which, when combined with ASC(C, K-1) (from ASC(C, K) at the beginning of the path), yields ASC(C, K) at the end of the path.

2-6 Assertions Before and After Assignments

Let us recall the fact that in section 1-10 we introduced a sieve program whose assertions were defined in terms of their total behavior, rather than being defined inductively. In fact, we could have defined the assertion U inductively by writing

```
** U(0) = .TRUE.
** U(K) = U(K-1) .AND. (A(K) = .TRUE.)
```

The assertions V and W, however, are not so easy to define inductively. In fact, they cannot be so defined using .AND. as above. Notice that the assertion U(K) implies that U(K-1) is true, and, in addition, something else is true. If V(K) is true, however, V(K-1) is false as it stands. The reason for this is that V(K) and V(K-1) involve the same variables, namely the A($\underline{x}$) for $1 \leq \underline{x} \leq N$, and some of these have different values when V(K) is true than when V(K-1) is. If the value of K is 2, for example, then V(K) implies that A(2$\underline{x}$) is .FALSE. for $\underline{x} \geq 2$, whereas V(K-1) implies that A($\underline{x}$) is .TRUE. for each $\underline{x}$. The assertion U(K), however, involves only the variables A($\underline{x}$) for $1 \leq \underline{x} \leq K$, and, in particular, it involves A(K), which U(K-1) does not. This is what makes it possible to use .AND. in the inductive definition of U.

We shall now introduce two operators, after and before, which will, among other things, enable us to make inductive definitions of V and W above. If A is any assertion and S is any assignment, then (A after S) means the relation among the variables of the program which is true after S is performed, if A was true before S was performed. Similarly, (A before S) is the relation which was true before S is performed if A is true after S is performed. Here are a few examples of the use of these operators:

1. $(I \leq N)$ <u>after</u> $(I = I + 1)$ is $(I \leq N + 1)$.

2. $(I \leq N)$ <u>before</u> $(I = I + 1)$ is $(I \leq N - 1)$, or $(I + 1 \leq N)$.

3. $(S = \sum_{x=1}^{I-1} A(\underline{x}))$ <u>after</u> $(S = S + A(I))$ is $(S = \sum_{x=1}^{I} A(\underline{x}))$.

4. $(S = \sum_{x=1}^{I} A(\underline{x}))$ <u>before</u> $(S = S + A(I))$ is $(S = \sum_{x=1}^{I-1} A(\underline{x}))$, or $(S + A(I) = \sum_{x=1}^{I} A(\underline{x}))$.

5. U(K-1) <u>after</u> (A(K) = .TRUE.) is U(K) (where U is as above).

6. U(K) <u>before</u> (A(K) = .TRUE.) is U(K-1).

It is clear from these examples that <u>before</u> and <u>after</u> are inverse to each other, although this does not hold in both directions. If A is the assertion (U = V .AND. X = Y) and S is the assignment (X = Y), then (A <u>after</u> S) is the same as A, whereas (A <u>before</u> S) is simply (U = V). However, ((A <u>before</u> S) <u>after</u> S) is always the same as A. The last two examples above illustrate the fact that our uses of .AND. in inductive assertions may be replaced by <u>after</u>; thus we could have written

```
** U(0) = .TRUE.
** U(K) = U(K-1) after (A(K) = .TRUE.)
```

as a definition of U above. Notice, however, that (A(K) = .TRUE.) is here an assignment, whereas before it was an assertion.

Examples 2 and 4 above suggest a general rule for calculating the result of <u>before</u>. If <u>v</u> is any simple (non-subscripted) variable and <u>e</u> is any expression, then the assertion (A <u>before</u> (<u>v</u> = <u>e</u>)) may be obtained by substituting <u>e</u> for <u>v</u> wherever <u>v</u> appears in the (complete) description of the assertion A. Thus in example 2 above, A is $(I \leq N)$, <u>v</u> is I, and <u>e</u> is I + 1; if we substitute I + 1 for I in A, we obtain $(I + 1 \leq N)$, which is one form of the result specified in example 2. This is called the <u>back substitution rule</u>; it breaks down, in general, when $\underline{y}$ is a subscripted variable.

Let us now redefine the assertions U, V, and W in our sieve program, using <u>after</u>:

```
** U(0) = .TRUE.
** U(K) = U(K-1) after (A(K) = .TRUE.)
** V(K) = W(K, K)
** W(2, 2) = U(N)
** W(K+1,K+1) = W(K, K*L) .AND. (K*(L+1) > N), for K ≥ 2
** W(K, K*L) = W(K, K*(L-1)) after (A(K*L) = .FALSE.), for K,L≥2
```

Our earlier definitions of U, V, and W may now be proved using these definitions. At the same time, we may prove that V(K), where K*K > N, implies that A($\underline{x}$) = PRIME($\underline{x}$) for each $\underline{x}$, $1 \leq \underline{x} \leq N$.

First let $1 \leq \underline{x} \leq N$; we show that U($\underline{y}$) implies that A($\underline{x}$) = .TRUE. for each $\underline{y}$, $\underline{x} \leq \underline{y} \leq N$. This certainly holds for $\underline{y} = \underline{x}$; if it holds for $\underline{y} = \underline{z} - 1$, $\underline{z} > \underline{x}$, then it holds for $\underline{y} = \underline{z}$, since $\underline{z} \neq \underline{x}$ and A($\underline{z}$) is the only variable involved in the transition from U($\underline{z}$-1) to U($\underline{z}$). In particular, U(N) implies A($\underline{x}$) = .TRUE., for $1 \leq \underline{x} \leq N$. Now let $\underline{x} = \underline{p} \cdot \underline{q}$, where $\underline{p}$, $\underline{q} \neq 1$ and $\underline{p}$, $\underline{q} \neq \underline{x}$. Then W($\underline{p}$, $\underline{p}$*$\underline{q}$) implies that A($\underline{x}$) = .FALSE.; we show that this is likewise implied by W($\underline{p}$, $\underline{p}$*$\underline{r}$) for each $\underline{r} \geq \underline{q}$. In fact, this holds for $\underline{r} = \underline{q}$, and if it holds for $\underline{r} = \underline{s}-1$, where $\underline{s}-1 \geq \underline{q} \geq 2$ and $\underline{p} \geq 2$, then it certainly holds for $\underline{r} = \underline{s}$, since, in the transition, no element of A is set to any value other than .FALSE.. In particular, this holds for $\underline{q}$*$\underline{r}$ > N, so that A($\underline{x}$) = .FALSE. is also implied by W($\underline{p}$+1, $\underline{p}$+1), where, as before, $\underline{p} \geq 2$. Continuing in the same way, we see that A($\underline{x}$) = .FALSE. is implied by W($\underline{p}$+2, $\underline{p}$+2), and in general by W($\underline{q}$, $\underline{q}$) for each $\underline{q} \geq \underline{p}+1$. This is the same as V($\underline{q}$), and, in particular, this shows that V($\underline{q}$) for $\underline{q}$*$\underline{q}$ > N implies that A($\underline{x}$)

= PRIME($\underline{x}$) whenever PRIME($\underline{x}$) = .FALSE., $1 \leq \underline{x} \leq N$. Now suppose that PRIME($\underline{x}$) = .TRUE.; then W(2, 2) (= U(N)) implies that A($\underline{x}$) = .TRUE., as detailed above. Then W(K, K*L) implies that A($\underline{x}$) = .TRUE., for any applicable values of K and L, since in the inductive definition of W the only changes ever made to the array A are to values A($\underline{p}$*$\underline{q}$) for $p \geq 2$, $\underline{q} \geq 2$. This implies that A($\underline{x}$) = .TRUE. follows from V($\underline{q}$) in this case as well, in particular when $\underline{q}$*$\underline{q}$ > N. This completes the proof.

2-7 Assertions About Lists

List processing programs exist on two levels. We may work directly with pointers to construct lists and other data structures, using an algebraic language or an assembly language. Or we may be given a higher-level language which processes lists directly, and in which each statement specifies a list-processing operation which we may assume to be performed correctly. Each of these levels has its own verification techniques.

In our discussion of list processing we shall use certain terminology from the language LISP, although our actual programming examples will not be given in LISP. A list is a sequence of _n_ objects x_1, ..., x_n; any of these may be lists themselves, but if they are not lists, they are called atoms. The notation $(x_1 \ldots x_n)$ to denote the list of objects x_1, ..., x_n is called the S-expression of that list; here any x_i which is itself a list appears as the S-expression of that list. We write $L = (x_1 \ldots x_n)$ to denote the fact that L is the name of the list of objects x_1, ..., x_n. If $L = (x_1 \ldots x_n)$, then _car_(L) = x_1 and _cdr_(L) = $(x_2 \ldots x_n)$; if _n_ = 1, then _cdr_(L) is the empty list, which is denoted by _nil_.

If we are doing list processing without the use of a higher level language, we must first establish certain conventions relating to the representation of lists in memory. For the purposes of verification, our next step is to define certain assertions about lists, using these conventions. Usually what is being asserted is that a certain list is represented in memory according to the conventions. By the nature of list processing, these conventions are not specific; a list may be represented in memory in a variety of ways, but the corresponding assertion will be satisfied if it is represented in <u>some</u> legal way, no matter what that way may be.

The following is an example of a set of conventions for the processing of lists of integers without sublists, using an algebraic language. A free storage area is given as an integer array FS of size $2\underline{m}$. Each item in a list is a pair (FS($2\underline{i}$-1), FS($2\underline{i}$), for $1 \leq \underline{i} \leq \underline{m}$. Here the integer FS($2\underline{i}$) is the content of the given item, and FS($2\underline{i}$-1) is the pointer to the next item; that is, it is of the form $2\underline{j}-1$, where (FS($2\underline{j}$-1), FS($2\underline{j}$)) is the next item. If there is no next item, i. e., if (FS($2\underline{i}$-1), FS($2\underline{i}$)) is the last item in its list, then FS($2\underline{i}$-1) = 0. The list whose S-expression is ($x_1 \dots x_n$) is then represented by $\underline{n}$ such items; if the $\underline{k}$th item, for example, is (FS($2\underline{i}$-1), FS($2\underline{i}$)), and the ($\underline{k}$+1)st item is (FS($2\underline{j}$-1), FS($2\underline{j}$)), then FS($2\underline{i}$-1) = $2\underline{j}-1$ and FS($2\underline{i}$) = x_k.

We may now define, relative to these conventions, an assertion LV(LP, L) (read "the list value of LP is L"), where LP is an ordinary integer variable and L = ($x_1 \dots x_n$) is a list of integers without sublists. This asserts that L is represented as $\underline{n}$ pairs in the manner described above, and that the value of LP is a pointer to the first of these pairs. The assertion LV(LP, L) for L = ($x_1 \dots x_n$) is the assertion that there exist integers $i_1, \dots, i_n$ with LP = $2i_1-1$ and $1 \leq i_j \leq \underline{m}$, FS($2i_j-1$) = $2i_{j+1}-1$, and FS($2i_j$) = x_j for each j, $1 \leq j \leq n$, except that for j = n we have FS($2i_j-1$) = 0. (Having made this definition, it is not hard to prove that all these integers i_j must be distinct.) However, LV(LP, L) may also be defined inductively, using the LISP terminology above. In fact,

LV(LP, L) = (<u>if</u> L = <u>nil</u> <u>then</u> LP = 0 else
    (LP is odd and FS(LP+1) = <u>car</u>(L) and LV(FS(LP), <u>cdr</u>(L))))

This definition is actually more inclusive that the preceding one, since it covers the case in which L is <u>nil</u>.

We emphasize the fact that these definitions represent only one set of list representation rules. The special pointer which marks the end of a list need not be zero; it may be detectable in some other way (as, for example, in LISP). Our pointers might have been taken in the form $\underline{i}$ or $2\underline{i}$, rather than $2\underline{i}-1$; or they could have been the absolute addresses of the corresponding pairs (if these are available in the given language). Our pairs might have been reversed, with the first element of each pair, rather than the second, denoting the content of the item; or two arrays FS1 and FS2 might have been used, with FS1($\underline{i}$) and FS2($\underline{i}$) constituting a single pair; or we might have used a double array, in which a pair consists of FS(1, $\underline{i}$) and FS(2, $\underline{i}$), or FS($\underline{i}$, 1) and FS($\underline{i}$, 2). We might even have concentrated all the information in a list item into a single element of the array FS, as we would probably do, for example, in assembly language. Still further conventions would be needed to handle lists of real numbers, or of complex numbers, or lists with sublists, or list structures with multiple links. With each of these, we would construct an assertion like LV(LP, L).

Let us define $\underline{\text{elt}}(\underline{i}, L)$ as the $\underline{i}$th element of the list L; that is, if $L = (x_1 \ldots x_n)$, then $\underline{\text{elt}}(\underline{i}, L) = x_i$. The LISP definition of $\underline{\text{elt}}$ is

$$\underline{\text{elt}}(\underline{i}, L) = \underline{\text{car}}(\underline{\text{cdrn}}(\underline{i}-1, L))$$

where

$$\underline{\text{cdrn}}(0, L) = L, \ \underline{\text{cdrn}}(\underline{i}, L) = \underline{\text{cdr}}(\underline{\text{cdrn}}(\underline{i}-1, L))$$

The function $\underline{\text{cdrn}}$ is the usual LISP function $\underline{\text{cdd}}\ldots\underline{\text{dr}}$, where the number of occurrences of $\underline{d}$ is the first argument of $\underline{\text{cdrn}}$. Using $\underline{\text{elt}}$ and $\underline{\text{cdrn}}$, we may set up assertions for the proof of correctness of a program which "unwinds" a list, copying its elements one by one into an array, as follows:

```
** E(0) = .TRUE.
** E(K) = E(K-1) .AND. (A(K) = elt(K, L))
* LV(X, L)
      I = 1
      GO TO 2
1     A(I) = FS(X+1)
      I = I + 1
      X = FS(X)
* E(I-1), LV(X, cdrn(I-1, L))
2     IF (X .NE. 0) GO TO 1
* E(length(L))
      CONTINUE
```

Here length(L), the length of the list L, is defined as usual by length(nil) = 0 and length(L) = length(cdr(L)) + 1 if L $\neq$ nil. The control paths here are verified as follows:

```
* LV(X, L)
     I = 1
* E(I-1), LV(X, cdrn(I-1, L))
```

E(I-1) is E(0), which is true; LV(X, cdrn(I-1, L) is LV(X, L) (which holds) because cdrn(I-1, L) = cdrn(0, L) = L.

```
* E(I-1), LV(X, cdrn(I-1, L))
     (X ≠ 0)
     A(I) = FS(X+1)
     I = I + 1
     X = FS(X)
* E(I-1), LV(X, cdrn(I-1, L))
```

The definition of LV shows that cdrn(I-1, L) $\neq$ nil at the beginning, and FS(X+1) = car(cdrn(I-1, L) = elt(I, L), so that E(I-1) and (A(I) = FS(X+1)) give E(I), and then I is increased by 1, giving E(I-1) again. Also X = FS(X) gives LV(X, cdr(cdrn(I-1, L))) = LV(X, cdrn(I, L)), which, when I is increased, becomes LV(X, cdrn(I-1, L)).

* E(I-1), LV(X, *cdrn*(I-1, L)) (X = 0)	Here the definition of LV shows that *cdrn*(I-1, L) = *nil* at the beginning, so that I-1 = *length*(L) (as may be easily shown) and E(I-1) becomes E(*length*(L)).
* E(*length*(L))	

Thus the program is partially correct. The fact that it terminates follows from a consideration of the controlled expression I, which is bounded along the loop control path since it can never become larger than *length*(L). The initial assertion, involving a list L of finite length, is, of course, crucial to the termination of this program.

2-8 Assembly Language

We shall now show how to prove the correctness of assembly language programs, using the methods developed thus far. Such programs have certain special problems, due to the possibility of self-modification; in the next section, we show how these are solved.

Superficially, from the viewpoint of verification, instructions in assembly language are much like very simple statements in an algebraic language. Indeed, each assembly-language statement has its algebraic-language counterpart, provided that we regard each register as a variable with a special name. If we have one accumulator called ac, for example, then "store X" becomes $X = \underline{ac}$, "add X" becomes $\underline{ac} = \underline{ac} + X$, and "shift left by $\underline{k}$ bits" becomes $\underline{ac} = \underline{ac} \cdot 2^k$. This last correspondence is not exact, of course, since we must make allowance for those bits which are shifted off the left-hand end of the accumulator. But rather than writing out the complete (and cumbersome) definition of the shifting operation each time it is used, we may simply use the fact that the above simplified definition holds whenever the leftmost $\underline{k}$ bits of the accumulator are all the same, as they almost always will be if the shift is used for arithmetic purposes. This then becomes an assertion to be associated with the shift instruction, much like the assertion $1 \leq I$ which is required at any FORTRAN IV statement which makes reference to A(I). If we are using a shift for a non-arithmetic purpose, we use a different property of it, such as one which considers the accumulator as being broken up into fields.

Indexed transfers in assembly language cause a problem whose method of solution will help us understand the treatment of self-modification in the next section. One way in which indexed transfers may be used is in analogy with switches, controlled GO TO statements, and the like, in algebraic languages. Consider the FORTRAN statement

```
GO TO (21, 22, 23, 65, 99), K
```

This specifies a transfer to one of five different statements. We may have to consider the possibility that this statement does not transfer at all; this will happen in some versions of FORTRAN when K is unequal to 1, 2, 3, 4, or 5. But at any rate there are at most six different next statements after this one; this is important because it sharply limits the number of control paths in the program. If we were doing this in assembly language, we would load K into an index register and then make an indexed transfer. If statement number 21 corresponds to the label L21, and so on, our program might look like this:

```
LOAD    XR,K
JUMP    *,XR
JUMP    L21
JUMP    L22
JUMP    L23
JUMP    L65
JUMP    L99
```

Here XR stands for the index register (or its equivalent) which we are using; this is loaded with the value of K, and then we jump to * (meaning "the current location") plus the current value contained in XR. This jump is what causes the difficulty, because theoretically it could go to any address in the entire computer. If there are 32,768 words in memory, it would seem that we must consider (at least) 32,768 different control paths in every assembly language program.

One way to avoid this is to add some instructions which test the value of K and then perform the indexed transfer only if this value is 1, 2, 3, 4, or 5. In this way we may directly consider all 32,768 (or however many) paths, and show that only five (or six) of them are ever actually taken. In fact, this is what would be done if this assembly language program were the result of a FORTRAN compilation, at least in some versions of FORTRAN. Our verification theory, however, would not be very general if it did not allow for the possibility of omitting this test in the interests of efficiency. This, after all, is what programmers do in practice; they say intuitively that "the value of K is always going to be in its proper range anyway, so why test?" What we do in this case is to make the assertion that K is equal to 1, 2, 3, 4, or 5, and associate this assertion with the start of the assembly language program above. This assertion is then proved, just like any other assertion.

Let us now consider a slightly more complex case. Suppose that we are using a subroutine call instruction which leaves the return address in the register XR. To return from a subroutine, we would us

```
JUMP    0,XR
```

Suppose now that a certain subroutine is called in n different places in a program, with corresponding return addresses a_1, ..., a_n. At th point at which we return from the subroutine, we must have an assertion that XR contains some one of these return addresses. If the subroutine calls another subroutine, using the same type of call instruction, then it will have to save the return address at the beginning, and restore it at the end. It is clear, however, that whether the return address is kept in XR or in some saved location, the assertion that it is in fact one of the a_i must be verified over every control path of the subroutine. If it is kept in XR, we must make sure that no instruction of the subroutine changes XR. If it is saved, we must make sure that no instruction of the subroutine changes the save area, except for the one which actually does the saving.

It may seem that we have a circular argument here. In order to limit the total number of control paths, we need the assertion that the return address is proper; but in order to verify this assertion, we need to make a complete analysis of all of our control paths. In fact, however, this type of argument is perfectly valid, and is not circular. All assertions at a single point in the program are combined into one, using "and"; among these will be the assertions about return addresses. We must now verify each control path in the program. In verifying each assertion separately at the end of the path, we may use all the assertions at the beginning of that path, in combination if necessary. Thus we may assume, for example, that the number of control paths in our program is properly limited, and that our other intermediate conditions are valid as well; and we may then proceed to prove both these things at once, by verifying each of our assumed control paths and showing that our return address remains proper at the end of each of them.

2-9 Self-Modifying Programs

As long as our programs cannot modify themselves, the control paths in them remain fixed. It would, however, appear that when a program has the capability, for example, of changing an ordinary assignment statement into a transfer statement during execution, then a new control path is created. How do we extend our method of control points and control paths to cover this situation? The answer is that we do not extend it at all; instead, we extend our model of the program to one in which the control paths, as properly interpreted, remain fixed throughout the execution. This will now be illustrated for assembly-language programs; most other languages do not allow true self-modification.

Even when an assembly-language program does not modify itself, this fact must be proved. Otherwise, we would have no way of being sure that the control paths in the program are what we say they are. We shall now indicate how this is done. Suppose that our program has been loaded into memory and is ready to start. Each instruction word in this program will contain a certain pattern of zeroes and ones, which directs the computer to perform some instruction. Now this statement is itself an assertion, albeit a highly complex one. The memory cells in which the instruction words are kept may certainly be thought of as variables, since theoretically their contents may change at any time. Since this program does not modify itself, however, the contents of these cells will not, in fact, change during the running of the program. This means that the assertion that each instruction word contains its own proper instruction code is true throughout the running of the program, and this fact may be proved using the same methods we have used before. Along each control path, we must show that this assertion is preserved; that is, if the instruction words have their indicated contents at the beginning of the path, then they still do at the end of it. Normally this is very simple; all we have to do is to observe that none of our statements affects the contents of any instruction cell. It is important to remember, however, that no assembly-language program is proved correct until this type of verification has been made.

Let us now consider a self-modifying program. Most self-modifying programs modify themselves in only a very few places, so let us first consider those instruction words which are <u>not</u> modified. Each of these has certain contents at the beginning of the program, and, by assumption, these contents are not modified during execution. Consequently this part of the analysis of a self-modifying program proceeds in exactly the same way as if it did not modify itself. That is, we must prove that all of these contents remain constant when we go along each control path of the program. Of course, it is no longer true that our statements do not affect the contents of <u>any</u> instruction cell, but they at least do not modify the contents of those cells which we have identified as never being modified.

It remains to consider the instruction words which are modified during execution, and how these affect the construction of the control paths. Taking a very simple case, suppose that we have an instruction which unconditionally transfers to location Z, and suppose that this instruction is later modified to contain "no operation" or some other statement which never transfers. How do we set up our control paths in the presence of such an instruction? We set them up exactly as if the instruction had been a conditional transfer to Z. In fact, it is a sort of conditional transfer, somewhat like the "zero test" instructions that mean "if X is zero, then skip the next instruction." If X denotes the location of this instruction, and $\underline{t}$ is the instruction code of the transfer to Z, then the meaning, in context, of this instruction is "If X = $\underline{t}$, then transfer to Z."

In a broader sense, this is true of every instruction word in every assembly language program. Suppose that $t_1, \ldots, t_n$ are the various possible contents of the instruction word X, where there are $\underline{b}$ bits in an instruction word and $\underline{n} = 2^b$. Let $I_1, \ldots, I_n$ be the instructions corresponding to $t_1, \ldots, t_n$ respectively. Then the meaning of the instruction word X is "If X = t_j, for any j, then do I_j." Some of the I_j, of course, will be meaningless or illegal. But in any case, if we wish to verify our programs, we will have to be able to prove that, when we get to X, the contents of X will be sharply limited. In the extreme case, we can prove that X can have only one possible value; this is the case we treated earlier in connection with non-self-modifying programs. If X has $\underline{k}$ possible values, then X must be treated as a $\underline{k}$-way conditional instruction. This does not necessarily mean, of course, that there are $\underline{k}$ different control paths to consider here, because some of these instructions may not transfer, and others may all transfer to the same place. The shortness of our proof, however, will depend to a great extent on how well we can limit the number of different possibilities to consider.

The statement that the instruction word X may contain only the instruction codes $u_1, \ldots, u_m$ (depending on X), is, of course, an assertion. In order to prove this assertion at X, we will normally have to strengthen it a bit. The assertion at X must not only say that only the instruction codes u_j will occur; it must also tell us under what conditions (on the other variables in the program) each u_j will actually be the contents of X. In verifying all the assertions of this type in our program, we make valid use of the same seemingly circular argument we encountered in the preceding section. That is, in setting up our control paths, we assume the validity of certain assertions about the instruction words; to prove these assertions, it is enough to analyze only those control paths which we have set up under the assumptions we have made.

Many programmers attempt to avoid self-modification entirely when they write assembly-language programs. The conventions which are used with certain computers for subroutine handling, however, make a certain amount of self-modification unavoidable. On many computers, the standard instruction which calls a subroutine leaves the return address in the address field of an instruction word, and then transfers to the next instruction word. Return from such a subroutine is accomplished by transferring to its first instruction word. If possible, this is done with indirect addressing; otherwise, the operation code field of this word will contain the operation code for an unconditional transfer, which in some cases may be placed there by the calling instruction. In either case, this first instruction word is modified during execution of the program. On other computers, when a return address is saved, it is saved in the address field of an instruction word at the end of its subroutine. This instruction word is later executed, and the result is to transfer control to the return address. Here again we have an instruction word which is modified by the program.

2-10 Subroutines

The existence of subroutines in a program being proved correct gives rise to a problem concerning the proper construction of control paths in that program. We know, even when we are not working in assembly language, that one of the functions of a subroutine call statement is to transfer control to the entry point of the corresponding subroutine. Suppose, however, that we constructed a control path which entered the subroutine when it came to a call statement. The only way that our control path, or some future control path, could get back into the calling program is through the return statement of the subroutine. But if the subroutine is called more than once, the return statement can return control to a number of different places. The only way we can tell where the return statement is going to lead us is by reference to the return address; and in most languages, the return address is kept carefully hidden from the view or concern of the programmer.

In assembly language, of course, our return addresses are in full view. We may, if we wish, construct control paths which lead from call statements into subroutines, and whose final assertions involve the return address. The initial assertion of a control path leading through a return statement will then involve a return address, and this assertion will allow us to determine which way to go after the subroutine is finished. If we do this, however, the assertions in our subroutines will be quite complicated. Consider, for example, the first control point in a subroutine. If this subroutine is called from <u>n</u> different places, we will have <u>n</u> different control paths which end here, and we will have to construct an assertion at this point which is true no matter which control path we have taken. If $A_1, \ldots, A_n$ are assertions which hold at the <u>n</u>

different call statements, and $a_1, \ldots, a_n$ are the corresponding return addresses, then

$$(A_1 \ \underline{\text{and}}\ (r=a_1)) \ \underline{\text{or}}\ (A_2 \ \underline{\text{and}}\ (r=a_2)) \ \underline{\text{or}}\ \ldots \ \underline{\text{or}}\ (A_n \ \underline{\text{and}}\ (r=a_n))$$

is an assertion which will be true at the beginning of the subroutine, provided that each call statement left its return address in the register **r**. Similarly, at the last control path before a return statement in the subroutine, we will have to construct an assertion that will lead correctly to an arbitrary return address. If $A_1', \ldots, A_n'$ are assertions which hold at the return addresses $a_1, \ldots, a_n$ respectively, then

$$(A_1' \ \underline{\text{and}}\ (r=a_1)) \ \underline{\text{or}}\ (A_2' \ \underline{\text{and}}\ (r=a_2)) \ \underline{\text{or}}\ \ldots \ \underline{\text{or}}\ (A_n' \ \underline{\text{and}}\ (r=a_n))$$

will hold at each return statement of the subroutine, provided that it transfers control to the address in the register **r**. This division into cases will then be carried throughout the proof of the subroutine.

Normally, the assertions A_i will have certain similarities to each other, which will tend to reduce somewhat the labor of proof. The difficulty of this approach to proving subroutines correct becomes even more apparent, however, when there are multiple levels of subroutines. If routine A calls B at $\underline{n}$ places and routine B calls C at $\underline{m}$ places, then there are $\underline{n} \cdot \underline{m}$ assertions connected by "or" in the routine C as above. Further levels will lead to further multiplication.

There is another method of proving subroutines correct which introduces into the proof process the same sort of division of labor that the use of subroutines introduces into programming itself. Suppose that we have a subroutine X which does not call any other subroutines at further levels. We may then prove this subroutine correct as a separate program, without reference to any other routine which calls it. We will have to do this with respect to some initial assertion A and some final assertion A'; at the same time, we note that X changes only the values of certain variables $x_1, \ldots, x_k$. Now suppose that the program P calls X at n places. At each of these there is an assertion A_i, and at the next statement following a given call statement, i. e., at its return address, there is an assertion A_i'. If $A_i = A$ and B_i, and $A_i' = A'$ and B_i, where B_i does not involve the variables $x_1, \ldots, x_k$, then the path from the call statement to its return address, through the subroutine, is valid. We may thus regard the call statement as if it were an assignment statement of a special kind. Hence our control paths, in particular, do not need to lead from a call statement to the start of the corresponding subroutine; rather, they lead directly through the call statement to the return address. This considerably simplifies the proof of programs with subroutines.

For a subroutine with parameters, the process is slightly more complex. The assertions A and A' above will now depend, in general, on what the actual parameters are. If a subroutine is meant to be called with m actual parameters, we may denote them by $p_1, \ldots, p_m$, and construct assertions $A(p_1, \ldots, p_m)$ and $A'(p_1, \ldots, p_m)$. In the proof of correctness of the subroutine, certain assertions will involve one or more of its formal parameters. Each such assertion is then understood to be a parametrized assertion as above, in which each formal parameter occurring in this way is replaced by the corresponding actual parameter symbol p_i. When the subroutine is called the validity of a control path through the call statement to its return address in the calling program is established by noting the actual parameters with which the subroutine is called, substituting these into $A(p_1, \ldots, p_m)$ and $A'(p_1, \ldots, p_m)$, and following the implications of the correctness of the corresponding path of length 1 through the call statement.

The decomposition of the proof of correctness of a program with subroutines into the separate proofs of correctness of the program and of the subroutines is a process which must, of course, be rigorously justified and not taken on faith. In fact, it is very easy for this process to be constructed incorrectly. Suppose, for example, that routine A calls B using a calling instruction which leaves the return address in register R, and routine B calls C using the same calling instruction. An experienced programmer will now notice that B must save the return address somewhere; otherwise it will be lost when B calls C, and we will never return to routine A (and in fact we will probably enter an endless loop). But the statement that B saves and restores R is not necessary in order to prove that B is correct. Thus, if B does not save R, the routines A, B, and C may all be correct separately, but not when they are combined. We must therefore state a set of rules which will be used whenever subroutines are called, and we must then <u>prove</u> that, if these rules are followed, the correctness of a collection of routines separately is equivalent to their correctness as a whole.

For programs in algebraic and higher-level languages, the same method may be used. It is clear, in particular, that in order for a program written in such a language to be executed correctly, the compiler or other language processor for the given language must itself be correct. But if the compiler is correct, then the object code must be produced in a correct manner, and this includes the statement that the rules which are followed by all object code in its handling of subroutines are valid in the sense defined above. Since these rules are valid, we do not need to worry about them in writing our programs in such a language. We simply prove each subroutine correct, and trust to the compiler to execute the program correctly as a whole, automatically.

For recursive programs, there is one further consideration. A collection of recursive programs may be separately correct, and a reasonable collection of rules may be strictly followed with respect to communication between subroutines, including the handling of a push-down list, and yet the collection of programs as a whole may be only partially correct. This is because of the problem of unlimited growth of the push-down list, or, what is the same thing, unlimited increase in the recursion level. The fact that this cannot happen must therefore be shown separately; it is a consequence of the following fact, which is usually true for correct recursive routines. Let $P_1, \ldots, P_n$ be routines which call each other mutually, and let the various arguments with which these routines are called be ordered in some way (if there exist an infinite number of possible sets of arguments, these must be well-ordered). Then if P_i is called with a set S_x of arguments, and P_i proceeds to call P_j with a set S_y of arguments, we must have $S_y \leq S_x$ in the ordering of the sets of arguments; in addition, if $i > j$, we must have $S_y < S_x$.

2-11 Matrix Multiplication

As our final example of a slightly longer program, we shall prove the partial correctness of a matrix multiplication routine. This will be used in the next section, where we discuss ways of abbreviating commonly occurring correctness arguments.

Three indices, I, J, and K, are used in this program, and each of them has its corresponding loop. The statement numbered 3 is common to all three loops, and therefore it is the only necessary control point, aside from the starting and stopping points. Our first task, therefore, is to determine precisely what the intermediate assertion should be at statement number 3. It is clear that this assertion depends heavily on I, J, and K. The first I-1 rows of the matrix C have already been filled in at this point; this assertion will be denoted by ROWS(I-1). The first J-1 elements of the I-th row have also been filled in, and this assertion is denoted by ROW(I, J-1). Finally, S contains a partial sum corresponding to the J-th element of the I-th row, this partial sum including all terms below the K-th term. We denote this last assertion by ESUM(S, I, J, K-1). The final assertion of this program is now ROWS(N), and the assertions at statement number 3 include ROWS(I-1), ROW(I, J-1), and ESUM(S, I, J, K-1), together with the statement that each of the indices I, J, and K has an integer value between 1 and N inclusive. As before, the statements that I, J, and K are strictly positive are needed only to insure that the various subscripts in the program will all be in their proper ranges. We give the program with its assertions, first making inductive definitions of ROWS, ROW, and ESUM:

```
** ROWS(0) = .TRUE.
** ROWS(I) = ROWS(I-1) .AND. ROW(I, N)
** ROW(I, 0) = .TRUE.
** ROW(I, J) = ROW(I, J-1) .AND. (C(I, J) = S) .AND.
**          ESUM(S, I, J, N)
** ESUM(S, I, J, 0) = (S = 0)
** ESUM(S, I, J, K-1) = ESUM(S + A(I,K)*B(K,J),
* N ≥ 1
        I = 1
1       J = 1
2       K = 1
        S = 0
* 1 ≤ I, I ≤ N, 1 ≤ J, J ≤ N, 1 ≤ K, K ≤ N,
* ROWS(I-1), ROW(I, J-1), ESUM(S, I, J, K-1)
3       S = S + A(I,K)*B(K,J)
        K = K + 1
        IF (K .LE. N) GO TO 3
        C(I, J) = S
        J = J + 1
        IF (J .LE. N) GO TO 2
        I = I + 1
        IF (I .LE. N) GO TO 1
* ROWS(N)
        CONTINUE
```

This program has five control paths. The first is

```
* N ≥ 1
    I = 1
    J = 1
    K = 1
```

```
        S = 0
    * 1 ≤ I, I ≤ N, 1 ≤ J, J ≤ N, 1 ≤ K, K ≤ N,
    * ROWS(I-1), ROW(I, J-1), ESUM(S, I, J, K-1)
```

Here $1 \leq I$, $1 \leq J$, and $1 \leq K$ follow from $I = 1$, $J = 1$, and $K = 1$ respectively; $I \leq N$, $J \leq N$, and $K \leq N$ follow from these and from $N \geq 1$. The remaining conditions are true by definition; specifically, ROWS(I-1) = ROWS(0) = .TRUE., ROW(I, J-1) = ROW(I, 0) = .TRUE., and ESUM(S, I, J, K-1) = ESUM(S, I, J, 0) = (S = 0), which is clearly true. The next three control paths are the loop control paths; the one for the innermost loop is

```
    * 1 ≤ I, I ≤ N, 1 ≤ J, J ≤ N, 1 ≤ K, K ≤ N,
    * ROWS(I-1), ROW(I, J-1), ESUM(S, I, J, K-1)
        S = S + A(I,K)*B(K,J)
        K = K + 1
        (K ≤ N)
    * 1 ≤ I, I ≤ N, 1 ≤ J, J ≤ N, 1 ≤ K, K ≤ N,
    * ROWS(I-1), ROW(I, J-1), ESUM(S, I, J, K-1)
```

Here $1 \leq K$ is clear because K increases, $K \leq N$ is clear from the condition $K \leq N$ in the path, and ESUM(S, I, J, K-1) at the beginning of the path becomes ESUM(S + A(I,K)*B(K,J), I, J, K), which is successively transformed into ESUM(S, I, J, K) and then ESUM(S, I, J, K-1). The other conditions at the end of this path are unchanged by the path because S and K are not involved in them. The next outer loop control path is

```
    * 1 ≤ I, I ≤ N, 1 ≤ J, J ≤ N, 1 ≤ K, K ≤ N,
    * ROWS(I-1), ROW(I, J-1), ESUM(S, I, J, K-1)
        S = S + A(I,K)*B(K,J)
```

```
            K = K + 1
            (K > N)
            C(I, J) = S
            J = J + 1
            (J ≤ N)
            K = 1
            S = 0
        * 1 ≤ I, I ≤ N, 1 ≤ J, J ≤ N, 1 ≤ K, K ≤ N,
        * ROWS(I-1), ROW(I, J-1), ESUM(S, I, J, K-1)
```

The arguments for $1 \leq K$, $K \leq N$, and ESUM(S, I, J, K-1) here are the same as in the first path, and those for $1 \leq I$, $I \leq N$, and ROWS(I-1) the same as in the second. We have $1 \leq J$ at the end since J increases, and $J \leq N$ from the condition $J \leq N$ in the path. We must have $K = N$ at the beginning of the path in order to have $K > N$ when it appears as a condition, and thus ESUM(S, I, J, N) holds up through C(I, J) = S; when combined with this and ROW(I, J-1), the result (by the definition) is ROW(I, J), and then J is increased by one, giving ROW(I, J-1) again. The outermost loop control path is

```
        * 1 ≤ I, I ≤ N, 1 ≤ J, J ≤ N, 1 ≤ K, K ≤ N,
        * ROWS(I-1), ROW(I, J-1), ESUM(S, I, J, K-1)
            S = S + A(I,K)*B(K,J)
            K = K + 1
            (K > N)
            C(I, J) = S
            J = J + 1
            (J > N)
            I = I + 1
            (I ≤ N)
            J = 1
            K = 1
            S = 0
      * 1 ≤ I, I ≤ N, 1 ≤ J, J ≤ N, 1 ≤ K, K ≤ N,
      * ROWS(I-1), ROW(I, J-1), ESUM(S, I, J, K-1)
```

Here we have $1 \le I$ at the end since I increases, and $I \le N$ from the condition $I \le N$ in the path. As before, we must have $K = N$ at the beginning of the path, and also $J = N$; also as before, we have ROW(I, J), that is, ROW(I, N), valid just before the statement J = J + 1, and combining this with ROWS(I-1) gives ROWS(I) by the definition, following which I is increased by one, giving ROWS(I-1) again. The other conditions in this path follow as in the first path. Finally, there is the path

```
      * 1 ≤ I, I ≤ N, 1 ≤ J, J ≤ N, 1 ≤ K, K ≤ N,
      * ROWS(I-1), ROW(I, J-1), ESUM(S, I, J, K-1)
            S = S + A(I,K)*B(K,J)
            K = K + 1
            (K > N)
            C(I, J) = S
            J = J + 1
            (J > N)
            I = I + 1
            (I > N)
      * ROWS(N)
```

In this path, by arguments resembling the preceding, we must have had $I = N$, $J = N$, and $K = N$ at the beginning. As in the preceding path, ROWS(I) holds just before the statement I = I + 1; but this is ROWS(N), and ROWS(N) continues to hold through the end of the path. Our program has thus been proved partially correct.

2-12 Standard Verification Arguments

The proof developed in the preceding section has made patently obvious the need for efficiency in the proof process. The program whose partial correctness was proved there uses only the most elementary of programming techniques. It consists of twelve statements, plus CONTINUE at the end, and it is equivalent to

```
          DO 1 I = 1, N
          DO 1 J = 1, N
          S = 0
          DO 2 K = 1, N
2         S = S + A(I, K)*B(K, J)
1         C(I, J) = S
```

which contains only six statements. Yet its proof took over three pages, although admittedly half of the proof consisted in listing paths. If we are to be able to verify programs of any reasonable length, we must learn to skip steps; we must identify certain arguments which are very often used in proofs of this kind, and agree not to make such arguments explicitly in the future. We shall now specifically identify four of these standard arguments, and postpone the proof of termination of our matrix multiplication program until the next section.

Each of our four arguments will be given an identifying letter. When this letter appears in parentheses after an assertion at the end of a control path, the corresponding standard argument is assumed to apply. As an illustration, we have constructed a hypothetical control path using all four of these arguments, as follows:

```
* 1 ≤ I, I ≤ N, F(S, T)
      K = 1
      I = I + 1
      (I ≤ N)
* 1 ≤ I (I), I ≤ N (C), K = 1 (A), F(S, T) (U)
```

The meanings of the identifying letters are as follows:

(U) The assertion is unchanged during the path. That is, an assertion at the end of the path is identical to one at the beginning, and none of the variables which appeared in that assertion were changed by any assignment in the path. Thus, in our example, the assertion F(S, T) appears both at the beginning and at the end of the path, and the two given assignments change only K and I. Notice that S and T could have appeared, although they did not, on the right sides of these assignments and in the intermediate condition.

(A) The assertion is the result of an assignment which is made on this path and whose variables are not later changed. If the assignment is $\underline{v} = \underline{e}$, where $\underline{v}$ is any variable and $\underline{e}$ any expression, then the condition may be $(\underline{v} = \underline{e})$, $(\underline{v} \leq \underline{e})$, $(\underline{v} \geq \underline{e})$, $(\underline{e} = \underline{v})$, etc. In the example, K is set to 1 as the first assignment of the path. Note that K could have been set to something else before the statement K = 1 in this path, although not afterward; also K could have occurred in right sides and intermediate conditions.

(C) The assertion is the same as an intermediate condition appearing in the path, whose variables are not later changed. Thus in our example the condition I ≤ N implies the final assertion I ≤ N. Here again, I and/or N could have appeared on the right side of an assignment after the intermediate condition I ≤ N, or in any other intermdiate condition, or on the left side before (I ≤ N), as here in the assignment I = I + 1.

(I) The assertion is an <u>inequality</u> which is "strengthened" by the path. For an inequality of the form *a* < *b*, for example, *b* may increase during the path, or *a* may decrease, but not vice versa. The same inequality is presumed to hold at the beginning of the path and at the end. Thus, here, the inequality $1 \leq I$ is strengthened by the statement I = I + 1.

Of the 34 assertions needed in the proof of our matrix multiplication program, 18 follow directly from these four standard arguments. We shall rewrite the five paths of this program with the standard arguments specified, omitting all proofs:

```
* N ≥ 1
      I = 1
      J = 1
      K = 1
      S = 0
* 1 ≤ I (A), I ≤ N, 1 ≤ J (A), J ≤ N, 1 ≤ K (A),
* K ≤ N, ROWS(I-1), ROW(I, J-1), ESUM(S, I, J, K-1)

* 1 ≤ I, I ≤ N, 1 ≤ J, J ≤ N, 1 ≤ K, K ≤ N,
* ROWS(I-1), ROW(I, J-1), ESUM(S, I, J, K-1)
      S = S + A(I,K)*B(K,J)
      K = K + 1
      (K ≤ N)
* 1 ≤ I (U), I ≤ N (U), 1 ≤ J (U), J ≤ N (U),
* 1 ≤ K (I), K ≤ N (C), ROWS(I-1) (U),
* ROW(I, J-1) (U), ESUM(S, I, J, K-1)
* 1 ≤ I, I ≤ N, 1 ≤ J, J ≤ N, 1 ≤ K, K ≤ N,
* ROWS(I-1), ROW(I, J-1), ESUM(S, I, J, K-1)
      S = S + A(I,K)*B(K,J)
      K = K + 1
      (K > N)
      C(I, J) = S
      J = J + 1
      (J ≤ N)
      K = 1
      S = 0
* 1 ≤ I (U), I ≤ N (U), 1 ≤ J (I), J ≤ N (C),
* 1 ≤ K (A), K ≤ N, ROWS(I-1) (U),
* ROW(I, J-1), ESUM(S, I, J, K-1)

* 1 ≤ I, I ≤ N, 1 ≤ J, J ≤ N, 1 ≤ K, K ≤ N,
* ROWS(I-1), ROW(I, J-1), ESUM(S, I, J, K-1)
      S = S + A(I,K)*B(K,J)
      K = K + 1
      (K > N)
      C(I, J) = S
      J = J + 1
      (J > N)
      I = I + 1
      (I ≤ N)
      J = 1
      K = 1
      S = 0
* 1 ≤ I (I), I ≤ N (C), 1 ≤ J (A), J ≤ N,
* 1 ≤ K (A), K ≤ N, ROWS(I-1), ROW(I, J-1),
* ESUM(S, I, J, K-1)
* 1 ≤ I, I ≤ N, 1 ≤ J, J ≤ N, 1 ≤ K, K ≤ N,
* ROWS(I-1), ROW(I, J-1), ESUM(S, I, J, K-1)
        S = S + A(I,K)*B(K,J)
        K = K + 1
        (K > N)
        C(I, J) = S
        J = J + 1
        (J > N)
        I = I + 1
        (I > N)
* ROWS(N)
* 1 ≤ I, I ≤ N, 1 ≤ J, J ≤ N, 1 ≤ K, K ≤ N,
* ROWS(I-1), ROW(I, J-1), ESUM(S, I, J, K-1)
        S = S + A(I,K)*B(K,J)
        K = K + 1
        (K > N)
        C(I, J) = S
        J = J + 1
        (J ≤ N)
        K = 1
        S = 0
* 1 ≤ I (U), I ≤ N (U), 1 ≤ J (I), J ≤ N (C),
* 1 ≤ K (A), K ≤ N, ROWS(I-1) (U),
* ROW(I, J-1), ESUM(S, I, J, K-1)

* 1 ≤ I, I ≤ N, 1 ≤ J, J ≤ N, 1 ≤ K, K ≤ N,
* ROWS(I-1), ROW(I, J-1), ESUM(S, I, J, K-1)
        S = S + A(I,K)*B(K,J)
        K = K + 1
        (K > N)
        C(I, J) = S
        J = J + 1
        (J > N)
        I = I + 1
        (I ≤ N)
        J = 1
        K = 1
        S = 0
* 1 ≤ I (I), I ≤ N (C), 1 ≤ J (A), J ≤ N,
* 1 ≤ K (A), K ≤ N, ROWS(I-1), ROW(I, J-1),
* ESUM(S, I, J, K-1)
* 1 ≤ I, I ≤ N, 1 ≤ J, J ≤ N, 1 ≤ K, K ≤ N,
* ROWS(I-1), ROW(I, J-1), ESUM(S, I, J, K-1)
      S = S + A(I,K)*B(K,J)
      K = K + 1
      (K > N)
      C(I, J) = S
      J = J + 1
      (J > N)
      I = I + 1
      (I > N)
* ROWS(N)
```

P R O B L E M S

1. The following program computes the square root of X to within a tolerance of 10^{-5}, under the assumption that X is positive. It is given with assertions; ABS denotes the absolute-value function. We repeat, as noted in section 1-2, that roundoff error and other types of error associated with floating point are to be ignored.

```
* X > 0
      A = X
      B = 1
      TOL = 1.0E-05
* A*B = X, A > 0, B > 0
1     A = (A + B)/2.0
      B = X/A
      IF (ABS(A-B) > TOL) GO TO 1
* ABS(A-SQRT(X)) ≤ TOL
```

Prove the partial correctness of this program. (Note that if the final assertion had been ABS(A-B) ≤ TOL , the partial correctness would have been trivial.)

2. (a) Consider the path in the above program which starts and ends at statement number 1. Verify this path with the additional assertion A > B at statement number 1. (Hint: Write A = $\underline{p} - \underline{q}$, B = $\underline{p} + \underline{q}$; then $\underline{p}$ = (A + B)/2.0 and X = A*B = $\underline{p}^2 - \underline{q}^2 < \underline{p}^2$.)

(b) If X < 1, the assertion A > B will not, of course, hold at the end of the control path which starts at the beginning of the program. Therefore, let us use (A = X .OR. A > B) instead. Re-verify the partial correctness of the entire program with this new assertion at statement number 1.

(c) If A > B and A*B = X, show that $\left|\frac{A+B}{2} - \sqrt{X}\right| < \frac{1}{2}\left|A - \sqrt{X}\right|$. (Hint: First show that the absolute value signs may be removed.)

(d) Add to the program the statement I = 0 at the beginning and the statement I = I + 1 somewhere within the loop. Add the assertion ABS(A-B) ≤ ABS(X-1.0)*(2.0**(-I)) at statement number 1. Using (c) above, verify the partial correctness of the expanded program.

(e) Verify the correctness of the expanded program. (The controlled expression is obviously I; therefore all we need to do is to add, and verify, another additional assertion at statement number 1, giving a bound on I.)

(f) Give an intuitive argument to the effect that the correctness of the expanded program is equivalent to the correctness of the original program. (Consider the fact that no new statements were added which may change any variables not present in the original program.)

3. Translate the following program according to the rules given at the end of section 2-2:

```
            OPEN FILE INPUT
            OPEN FILE OUTPUT
      1     READ INPUT, X
            IF (EOF, INPUT) GO TO 2
            WRITE OUTPUT, X
            GO TO 1
      2     END FILE OUTPUT
```

It is assumed that INPUT and OUTPUT here are file names.

4. (a) The program of the preceding problem is an adaptation of the program given in section 2-2, and in this section it was stated that this program is in turn equivalent to one given in section 1-8. Formulate the initial, intermediate, and final assertions for the program of the preceding problem which correspond to those of the program in section 1-8.

(b) Prove the correctness of the program of the preceding problem with respect to the assertions obtained as above.

5. (a) Suppose that A(1) = 3, A(2) = 5, A(3) = 7, A(4) = 5, B(1) = 5, B(2) = 5, B(3) = 7, and B(4) = 3. Is PERM(A, B, 4) true?

(b) Suppose that A(1) = 1, A(2) = 2, A(3) = 3, A(4) = 3, B(1) = 2, B(2) = 3, B(3) = 4, and B(4) = 1. Is PERM(A, B, 4) true?

6. What is the sortedness of each of the following arrays? Assume that the elements of each array are ordered as given.

(a) 6, 3, 8, 10, 2.

(b) 19, 17, 12, 10, 3, 2, 1.

(c) 4, 8, 2, 11, 1, 12.

(d) 1, 2, 3, 4, 5, 6, 7, 8, 9, 10.

7. In the program of section 2-4, suppose that we alter the condition in statement number 1 from .GT. to .LE., and rearrange the other statements accordingly (without changing them), as follows:

```
        I = 1
1       IF (A(I) .LE. A(I+1)) GO TO 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
      2       I = I + 1
              IF (I .NE. N) GO TO 1
      3       CONTINUE
```

(Notice that, in the process, we have reduced the total number of executable statements in the program from 12 to 11.) How does this change affect the proof of correctness of the given program?

8. Suppose that we change statement number 2 (in the <u>original</u> program of section 2-4, not as modified above), together with the following two statements, to

```
T = A(I+1)
A(I+1) = A(I)
A(I) = T
```

How does this change affect the proof of correctness of this program?

9. In the following problems, assume that MERGE(I, J, K) is defined, as in section 2-5, relative to the arrays A, B, and C.

(a) Suppose that the arrays A, B, and C are given by

1, 6, 8, 12, 17

2, 12, 17, 19

1, 2, 6, 8, 12, 17, 19

(that is, A(1) = 1, A(2) = 6, A(3) = 8, etc.) Is MERGE(5, 4, 7) true?

(b) Suppose that the arrays A, B, and C are given by

1, 6, 8, 12, 17

24, 25, 29, 31, 33

1, 6, 8, 12, 17, 24, 25, 29, 31, 33

Is MERGE(5, 5, 10) true?

10. (a) Suppose that $A(1) = A(2) = \ldots = A(10) = 17$. Is ASC(A, 10) true?

(b) Suppose that $A(1) = -4$, $A(2) = -3$, $A(3) = -2$, and $A(4) = -1$. Is ASC(A, 4) true?

11. What is the value of each of the following?

(a) $(K = 2)$ <u>after</u> $(K = K + K)$

(b) $(I \leq J + 5)$ <u>after</u> $(J = 10 - J)$

(c) $(W = Y)$ <u>after</u> $(W = Y*Y)$

(d) $(K \neq L)$ <u>after</u> $(K = K + 5)$

12. Use the back substitution rule to determine the value of each of the following:

(a) (S = SIGMA(A(K), K, 1, I)) <u>before</u> (S = S + A(I))

(b) (MERGE(I-1, J-1, K-1)) <u>before</u> (K = K + 1)

(c) $(T < U + V)$ <u>before</u> $(U = 2*V)$

(d) $(D > E + 5)$ <u>before</u> $(D = E + 3)$

13. Modify the definition

LV(LP, L) = (<u>if</u> L = <u>nil</u> <u>then</u> LP = 0 <u>else</u>
  (LP is odd and FS(LP+1) = <u>car</u>(L) and LV(FS(LP), <u>cdr</u>(L))))

given in section 2-7 for the assertion LV ("list value") as directed by each of the following modifications in the list processing conventions given just before that definition. Some of these modifications are mentioned in the paragraph following the definition; each one is to be considered separately.

(a) The special pointer which marks the end of a list is taken to be a constant called NIL.

(b) Any pointer to the j-th item in the free storage is given by j, rather than $2j-1$.

(c) The first element of each pair denotes the content of the item, and the second denotes a pointer or zero as originally specified for the first.

(d) Two arrays FS1 and FS2 are used. The j-th item consists of FS1(j) and FS2(j); by a pointer to this item we mean the integer j. The content of the item is in FS1(j) and the pointer or zero is in FS2(j).

(e) A double array is used; otherwise, same as (d) above, with FS1(j) replaced by FS(1, j) and FS2(j) replaced by FS(2, j).

14. In each of the five cases above, state in detail what changes must be made in the list unwinding program of section 2-7, and in its proof of correctness, in order for it to remain correct with respect to the assertions given.

15. The following operations and pseudo-operations in an imaginary assembly language are found in Programming: An Introduction To Computer Techniques by Maurer. We shall assume that our computer has an accumulator, which is the only register with which we shall be concerned, plus an unspecified number of memory words which have names I, J, K, ... The pseudo-operation

```
n       RE       1
```

where n is any name, reserves one memory word and gives it the name n. The pseudo-operation

```
n       CO       k
```

where n is any name and k is any constant, reserves one memory word and gives it the name n and the initial contents k. The following instructions are given with their approximate algebraic-language counterparts; here ac denotes the accumulator.

LD	Y	ac = Y
AD	Y	ac = ac + Y
SU	Y	ac = ac - Y
MU	Y	ac = ac * Y
DI	Y	ac = ac / Y
ST	Y	Y = ac
TR	BETA	GO TO BETA
TZ	BETA	IF (ac = 0) GO TO BETA

The following program now represents in this assembly language the exponentiation program of section 1-4:

```
I     RE   1
N     RE   1
A     RE   1
X     RE   1
ONE   CO   1
ZERO  CO   0
      LD   ONE
      ST   X
      LD   ZERO
      ST   I
L1    LD   I
      SU   N
      TZ   L2
      LD   X
      MU   A
```

```
          ST    X
          LD    I
          AD    ONE
          ST    I
          TR    L1
    L2    (continue)
```

Prove the correctness of this program in the same way that the correctness of the original program was proved in Chapter 1. Assume, for the moment, that the approximate algebraic-language equivalents given are in fact exact.

16. Translate the prime-testing program of section 1-9 into the assembly language of the preceding problem, and prove its correctness in this form, under the same assumptions as above. You may use one additional instruction:

```
    TP   BETA         IF (ac ≥ 0) GO TO BETA
```

Your initial assertion should be $I \geq 4$ (as mentioned in section 1-9, the program is not correct for $I = 2$). Note that the MOD function may be translated into an assembly language using a divide instruction which does not compute remainders (as the one defined above) by dividing, then multiplying, and finally subtracting; thus, for example, MOD(7,2) = 7-(7/2*2) = 7-(3*2) = 7-6 = 1.

17. Suppose that a computer with no index registers and a maximum of 4K 18-bit words has the assembly language defined in problem 16 above. There is one instruction per word, and standard (not base-displacement) addressing is used; the first six bits of an instruction are the operation code, and the remaining 12

bits are the address. The operation codes for the instructions which we have introduced are given as follows in octal:

LD -- 21	MU -- 25	TZ -- 50
AD -- 23	DI -- 27	TR -- 60
SU -- 24	ST -- 22	TP -- 52

Suppose that the program of problem 15 above is loaded into memory, starting at octal address 0206. Formulate the precise assertions about the instruction and data words of this program which should be made at its beginning and at L1 and L2 in order to complete the proof of correctness of this program begun in problem 15, taking now into account the fact that it must be proved that this program does not modify itself.

18. The following program represents, in the assembly language of problem 15 above, the summation program of section 1-7. It is assumed that the pseudo-operation

```
n       RE      m
```

where n is any name and m is any integer, reserves m memory words for an array called n (of dimension m). All variables are assumed to be integers.

```
N       RE      1
I       RE      1
S       RE      1
A       RE      100
ONE     CO      1
ZERO    CO      0
INST    AD      A
START   LD      INST
        ST      L2
        LD      ONE
        ST      I
        LD      ZERO
        ST      S
L1      LD      S
L2      RE      1
        ST      S
        LD      L2
        AD      ONE
        ST      L2
        LD      I
        AD      ONE
        ST      I
        LD      N
        SU      I
        TP      L1
```

Suppose that this program is loaded into the computer of problem 17 above, starting at octal address 1051. Prove the correctness of the program with respect to assertions derived from those in section 1-7; the initial assertion must include $N \leq 100$ because of the restriction in the program above on the array A. Note that this program modifies itself; the operation code for an add instruction is such that the sign bit in the instruction word will be zero, so that, as an integer, the instruction word will be positive. (This insures, for example, that adding 1 to

```
        AD      A
```

will produce

```
        AD      A+1
```

no matter how negative integers are represented.)

19. The following FORTRAN subroutine is given with assertions:

```
            SUBROUTINE ROOT
            COMMON A, B, C, N
      * C ≥ 0
            B = SQRT(C)
      * B = √C
            RETURN
            END
```

It is called by the following FORTRAN program with assertions:

```
            REAL N
            COMMON A, B, C, N
      * N > 0
            C = 1.0
            A = 0.0
      * N > 0, C > 0, C ≤ N, A = $\sum_{X=1}^{C-1}\sqrt{X}$
      1     CALL ROOT
      2     A = A + B
            C = C + 1.0
            IF (C .LE. N) GO TO 1
      * A = $\sum_{X=1}^{N}\sqrt{X}$
```

The COMMON statements in the program and in the subroutine, of course, insure us that all variables here are global. Assume that the subroutine is correct (the proof of this would involve the correctness of the further function SQRT called by it).

(a) Prove the correctness of the program under the assumption that the statement CALL ROOT may be replaced by its effect (i. e., by the presumably correct subroutine).

(b) Prove the correctness of the program under the assumption that the statement CALL ROOT is the same as "Set R = 2 and go to ROOT" and that the statement RETURN is the same as "Go to the statement whose statement number is given by the value of R." Assume that R is an integer variable.

20. (a) Suppose that we were to add the statement A = 0.0 after B = SQRT(C) in the subroutine. Does this affect the correctness of the subroutine?

(b) Does it affect the partial correctness of the main program?

(c) Where does the proof of correctness of the main program break down under these conditions?

(d) Suppose, instead, that we were to add the statement C = 1.0 <u>before</u> B = SQRT(C) in the subroutine. Does this affect the correctness of the subroutine?

(e) Does it affect the partial correctness of the main program?

(f) Where does the proof of correctness of the main program break down under these conditions?

21. In the matrix multiplication routine of section 2-11, we used the variable S 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) = S, and replace S by C(I, J) everywhere else in the program. How does this change affect the proof of partial correctness?

22. Complete the proof of correctness of the routine given in section 2-11.

23. Identify the standard verification arguments (see section 2-12) which may be used in the proof of partial correctness given in sections 1-4 and 1-5.

24. Identify the standard verification arguments which may be used in the proof of partial correctness given in section 1-10.

3 STATE VECTOR FUNCTION THEORY

The title of this chapter does not adequately convey its generality. State vector function theory, as we shall develop it, is nothing less than <u>the theory of programs</u> as they are actually run on computers. Various other "theories of programs" or "theories of algorithms" have been formulated, but most of these are about programs for artificially constructed machines, such as Turing machines. A Turing machine has <u>states</u>, which may be thought of as integer values of a single variable which constitutes the entire "innards" of the machine. A digital computer, on the other hand, has <u>state vectors</u>, and so does a program written in an algebraic or higher level language. Each state vector is a set of states or values, one for each variable. <u>State vector functions</u> are then functions on state vectors, i. e., functions f(S), where S is a state vector.

In developing state vector function theory, the fundamental object is the <u>memory</u>, which is a set of variables together with a way of telling what values each variable can take. The reader who is familiar with other mathematical theories, such as group theory or automata theory, may make comparisons between the way in which a group, an automaton, or a memory (in our sense) is defined in terms of sets and functions. Once this is done, we may define the concepts of state vector and state vector function in formal terms. We may then develop certain theorems, then use these to prove more complex theorems, and ultimately build up a theory which may then be applied in proving assertions about arbitrary programs.

3-1 Sets and Functions

In order to develop our formal mathematical models, we shall start with some basic facts about sets and functions. These may also be found in textbooks on elementary set theory, modern algebra, or the like.

The notion of a set is fundamental to all mathematics. A set contains elements, and each of these elements is a member of the set. If the element x is a member of the set X, we say that x is in X, or $x \in X$. If x is not in X, we write $x \notin X$. It is always true that either $x \in X$ or $x \notin X$, for any element x and any set X.

If X and Y are sets, then the union of X and Y, or $X \cup Y$, is the set of all elements which are in either X or Y, and the intersection of X and Y, or $X \cap Y$, is the set of all elements which are in both X and Y. Every set has a certain number of elements, and this number may be finite or infinite. If it is finite, it may be positive or zero. There is exactly one set whose number of elements is zero; this set is denoted by ϕ, and is called the null set. If $X \cap Y = \phi$ for the sets X and Y, then X and Y are called disjoint. If the number of elements in a set is infinite, it is called an infinite set; otherwise, a finite set.

A finite set may be specified by listing all its elements in brackets. Thus $X = \{2, 3, 6, 7\}$ means that X is the set consisting of the four elements 2, 3, 6, and 7. The set of all integers between m and n inclusive will sometimes be abbreviated $\{m, m+1, \ldots, n\}$; if $m = 1$, we write $\{1, \ldots, n\}$.

If every element of the set X is also an element of the set Y, then we say that X is <u>contained</u> in Y, or that Y <u>contains</u> X. These may be abbreviated $X \subset Y$ and $Y \supset X$. If both $X \subset Y$ and $Y \subset X$, then X and Y must be the same set, i. e., $X = Y$. A <u>subset</u> of Y is any set X contained in Y. Any property of the elements of a set defines a subset consisting of all elements which have that property. The set of all elements of Y which are greater than zero, for example, is a subset of Y; it may be denoted by $\{y \in Y: y > 0\}$. A subset of Y is called <u>proper</u> if it is unequal to Y.

A <u>function</u> f with <u>domain</u> X, where X is a set, is a way of associating a <u>value</u>, called f(x), with every element $x \in X$. The set of all values of the function f is called the <u>range</u> of f. If X is any set which contains the domain of f and Y is any set which contains the range of f, we may write $f: X \rightarrow Y$, and we may say that f is a function <u>from</u> X <u>to</u> Y. Such a function is called <u>total</u> (or a <u>total function</u>) if X is equal to its domain, and it is called <u>onto</u> (or <u>onto Y</u>) if Y is equal to its range. If f is not necessarily onto Y, it is called <u>into</u> Y; if f is not necessarily total, it is said to be <u>partial</u>. (Partial functions may be thought of as "partially defined," i. e., f(x) does not necessarily exist for $x \in X$, where $f: X \rightarrow Y$.) A function f is <u>one-to-one</u> if $f(a) = f(b)$ always implies $a = b$.

THEOREM <u>3-1-1</u>. A total function $f: X \rightarrow X$, where X is a finite set, is one-to-one if and only if it is onto. (Such a function is called a <u>permutation</u>.)

<u>PROOF</u>. If f is not one-to-one, then there exist $a \in X$, $b \in X$ such that $f(a) = f(b)$. If X contains <u>n</u> elements, then the range of f consists of f(a) together with the (at most) n-2 values f(x) for $x \neq a$, $x \neq b$. There are thus at most n-1 values in this range, which

shows that X is not equal to the range and therefore f is not onto. Conversely, if f is not onto, the range of f can contain at most n-1 elements; since there are n quantities f(x), for $x \in X$, at least two of them must be the same, which shows that x is not one-to-one.

The restriction of the function $f: X \to Y$ to the subset X' of X is the function $g: X' \to Y$ such that $g(x) = f(x)$ for all $x \in X'$. It is abbreviated $f|X$.

Let $f: X \to Y$ be a function and let X' be a subset of X. The set of all f(x) for $x \in X'$ is a subset of Y. If we denote this subset of Y by Y', we may write $f(X') = Y'$. This notation is very often used, although it would be more accurate to write $f'(X') = Y'$, where f' is the function from the set of all subsets of X to the set of all subsets of Y defined by $f'(X') = \{y \in Y: y = f(x)$ for some $x \in X'\}$, for each subset X' of X.

The inverse image of an element $y \in Y$ under a function $f: X \to Y$ is the set of all $x \in X$ such that $f(x) = y$.

A function may be specified by listing all its values if its domain is finite. In particular, suppose that the domain of a function f is the set $\{1, 2\}$. Then the function is completely specified by giving f(1) and f(2). If $f(1) = x$ and $f(2) = y$, then the function is completely specified by writing (x, y). The expression (x, y) is called an <u>ordered pair</u>, and the set of all such ordered pairs (x, y), for all x in a set X and all y in a set Y, is called the <u>cartesian product</u> of the two given sets X and Y, or $X \times Y$. (It is also possible to define functions in terms of ordered pairs, rather than ordered pairs in terms of functions. In fact, the set of all ordered pairs of the form $(x, f(x))$, for some function f, may be taken as a definition of f.)

Unions, intersections, and cartesian products may be extended to the case of more than two factors. If $X_1, X_2, \ldots, X_n$ are sets, then their <u>union</u> is the set of all elements which are in any of them, and their <u>intersection</u> is the set of all elements which are in all of them. The union of the sets $X_1, X_2, \ldots, X_n$ is denoted by $\bigcup_{i=1}^{n} X_i$, and their intersection by $\bigcap_{i=1}^{n} X_i$. The cartesian product $X_1 \times X_2 \times \ldots \times X_n$ is the set of all <u>ordered</u> <u>n-tuples</u> $(x_1, x_2, \ldots, x_n)$, where each x_i is a member of the corresponding X_i. Following the definition of an ordered pair above, we define an ordered n-tuple as a function whose domain is the set $\{1, \ldots, n\}$, specified by giving the value $f(i)$ for each i. In the ordered n-tuple $(x_1, \ldots, x_n)$, considered as a function f, $x_i = f(i)$ for $1 \leq i \leq n$.

These constructions may be taken a step further. Let I be a set and for each $i \in I$ let X_i be a set. The <u>union</u> of all the X_i is the set of all elements which are in any of the X_i, and is denoted by $\bigcup_{i \in I} X_i$. The <u>intersection</u> of all the X_i is the set of all elements which are in all of the X_i, and is denoted by $\bigcap_{i \in I} X_i$. The <u>cartesian product</u> of all the X_i will be defined as the set of all functions f with domain I which are such that $f(i) \in X_i$ for each $i \in I$. This product may be denoted by $\prod_{i \in I} X_i$.

The <u>identity</u> <u>function</u> on any set X is the total function

$e: X \rightarrow X$ such that $e(x) = x$ for all $x \in X$. The composition of two functions $f: X \rightarrow Y$ and $g: Y \rightarrow Z$ is the function $h: X \rightarrow Z$ such that $h(x) = g(f(x))$ whenever this is defined; we write $h = f \circ g$. A function $f: X \rightarrow Y$ has an inverse $g: Y \rightarrow X$ if and only if $f \circ g = e$.

THEOREM 3-1-2. A function has an inverse if and only if it is one-to-one, and this inverse is itself one-to-one. The inverse of a function is total if and only if the function is onto, and onto if and only if the function is total.

PROOF. If the function $f: X \rightarrow Y$ is not one-to-one, then there exist $a \in X$, $b \in X$ with $f(a) = f(b)$. If f were to have an inverse g, then $a = e(a) = g(f(a)) = g(f(b)) = e(b) = b$. Conversely, if f is one-to-one, we may define $g: Y \rightarrow X$ such that $g(y) = x$ whenever $f(x) = y$. This function is well defined, because otherwise we would have $g(y) = x_1$ and $g(y) = x_2$ for $x_1 \neq x_2$, or $f(x_1) = y = f(x_2)$, and f would not be one-to-one. If g were not one-to-one, we would have $g(y_1) = g(y_2)$ for $y_1 \neq y_2$, i. e., there exists x with $f(x) = y_1$ and $f(x) = y_2$, which is absurd. The function g will be total if and only if, for every $y \in Y$, there exists $g(y)$, i. e., there exists x with $f(x) = y$, and this is true if and only if f is onto. The function g will be onto if and only if, for every $x \in X$, there exists y with $g(y) = x$, i. e., $f(x) = y$, and this is true if and only if f is total.

A total function $f: X \rightarrow Y$ which is one-to-one and onto is called a one-to-one correspondence.

COROLLARY 3-1-1. The inverse of a one-to-one correspondence is a one-to-one correspondence.

A one-to-one correspondence is natural (or sometimes canonical) if there is a particularly easy way to define it. This, of course, is not a precise concept. When we state that there exists a natural one-to-one correspondence between two sets, however, we mean the following: (1) there is a one-to-one correspondence; (2) we are now going to define a particular one-to-one correspondence which will be considered from now on as "the natural one" between these two particular sets. As an example of this, we now define a particular one-to-one correspondence which is very widely used.

<u>THEOREM</u> <u>3-1-3</u>. Let X be any set and let P(X) be the set of all subsets of X (including the null set and X itself). Let B be the set $\{$<u>true</u>, <u>false</u>$\}$ of two elements, and let F(X, B) be the set of all total functions $f: X \rightarrow B$. Then there is a natural one-to-one correspondence between P(X) and F(X, B).

<u>PROOF</u>. For each subset X' of X let the corresponding $f: X \rightarrow B$ be defined by $f(x) =$ <u>true</u> if $x \in X'$ and $f(x) =$ <u>false</u> otherwise. This correspondence is total; it is also one-to-one, because different subsets clearly correspond to different functions. If $f: X \rightarrow B$ is an arbitrary total function, then the set of all $x \in X$ for which $f(x) =$ <u>true</u> is a subset of X from which f is derivable in the above way. Thus the correspondence is also onto.

A <u>binary</u> <u>operation</u> on any set X is a function $f: X \times X \rightarrow X$. For example, the operator + on a set X (if it is defined) corresponds to the binary operation $f: X \times X \rightarrow X$ defined by $f(x, y) = x + y$. The domain of f is the set $X \times X$, i. e., the set of ordered pairs (x, y) with $x \in X$ and $y \in X$, and hence we should properly write $f((x, y))$. However, when the domain of any function is a cartesian product we shall drop the extra pair of parentheses. An <u>n-ary</u> <u>operation</u> on X is a function $f: X^n \quad X$, where X^n is defined to be the cartesian product of <u>n</u> copies of X, i. e., the set of all total functions from $\{1, \ldots, \underline{n}\}$ into X.

A relation on a set X is a map $f: X \times X \to \{\text{true}, \text{false}\}$. For example, the operator $<$ on a set X (if it is defined) corresponds to the relation $f: X \times X \to \{\text{true}, \text{false}\}$ defined by $f(x, y) = \text{true}$ if $x < y$ and $f(x, y) = \text{false}$ otherwise. A relation may also be thought of as a subset of $X \times X$, by the preceding theorem. In general, when we have a natural one-to-one correspondence between two sets and we are given an element of one of these sets, we may regard it just as easily as an element of the other. In this case the set of all subsets of $X \times X$ and the set of all maps $f: X \times X \to \{\text{true, false}\}$ may be thought of interchangeably as the set of all relations on X. As a subset of $X \times X$, a relation is the set of all ordered pairs (x, y) for which the relation holds true ($x < y$, in our example). If a relation is called R, and (x, y) is in the corresponding set (or, thinking of R as a function, $R(x, y) = \text{true}$), we may write xRy.

A relation is reflexive if xRx is always true and irreflexive if xRx is always false. A relation is symmetric if xRy implies yRx and antisymmetric if xRy and yRx imply xRx. (This includes the possibility that xRy and yRx are never true at the same time.) A relation is transitive if xRy and yRz imply xRz. A relation is an equivalence relation if it is reflexive, symmetric, and transitive.

A partition (or a decomposition) of a set X is a set $\mathfrak{D}$ of subsets of X such that $D \cap D' = \phi$ for any two elements D, D' of $\mathfrak{D}$, and also $\bigcup_{D \in \mathfrak{D}} D = X$. That is, each element of X belongs to exactly one $D \in \mathfrak{D}$.

THEOREM 3-1-4. There is a natural one-to-one correspondence between the set of all equivalence relations on any set X and the set of all partitions of X.

PROOF. Let R be an equivalence relation on X. For each $x \in X$, let $D_x = \{y \in X: xRy\}$. (The set D_x is called the equivalence class of x in X.) The corresponding partition $\mathcal{D}$ is now the set of all the D_x for $x \in X$, with the restriction that if two or more of the D_x are the same set then this set is counted only once. Since R is reflexive, $x \in D_x$ and thus $\bigcup_{D \in \mathcal{D}} D = X$. Suppose that $D_x \cap D_y \neq \phi$; we will show that in this case $D_x = D_y$. There exists $z \in X$ such that xRz and yRz; if $a \in D_x$; since R is symmetric, zRy; since R is transitive, xRz and zRy imply xRy, and, again since R is symmetric, yRx. If $a \in D_x$ then xRa; since R is transitive, yRx and xRa imply yRa, and $a \in D_y$. Conversely, if $a \in D_y$, then yRa, and since R is transitive, xRy and yRa imply xRa, so that $a \in D_x$. Thus $D_x = D_y$. Now let $\mathcal{D}$ be any partition, and define the corresponding relation R as a subset of X to be the set of all (x, y) where x and y are both in the same element $D \in \mathcal{D}$. Clearly $(x, x) \in R$ for each x, so that R is reflexive. If x and y are in the same element of $\mathcal{D}$, this is the same as saying that y and x are, so that xRy implies yRx; thus R is symmetric. Finally, if x and y are in the same element of $\mathcal{D}$, and so are y and z, then, clearly, so are x and z. Thus xRy and yRz imply xRz, and R is transitive, which completes the proof that R is an equivalence relation.

A relation is a partial ordering if it is reflexive, antisymmetric, and transitive. It is customary to denote a relation by $\equiv$ if it is an equivalence relation and by $\leq$ if it is a partial ordering. A partially ordered set is a set X together with a partial ordering on X. A partially ordered set has a smallest element x if $x \leq y$ for all y, and a greatest element y if $x \leq y$ for all x. It has a maximal element x if $x \leq y$ implies $x = y$, and a minimal element y if $x \leq y$ implies $x = y$.

A directed graph is a set X together with a set of ordered pairs (x, y), $x \in X$, $y \in X$. If the ordered pair (x, y) is in this set, we write $x \to y$. The elements of X are called the nodes of the graph and the ordered pairs (x, y) are called the links. If x and y are nodes, then we say that there is a directed path of length n from x to y if there exist nodes $x_0, \ldots, x_n$, $x_0 = x$, $x_n = y$, with $x_{i-1} \to x_i$ for $1 \leq i \leq n$. There is always by definition a directed path of length zero from x to x. A directed path of length greater than zero from x to x is called a directed cycle. A graph with no directed cycles is called acyclic. (By "graph" we shall always mean a directed graph.)

THEOREM 3-1-5. On any acyclic graph the relation R, defined by xRy if and only if there exists a directed path (of length ≥ 0) from x to y, is a partial ordering. (Acyclic graphs are accordingly sometimes called ordered graphs.)

PROOF. By the above paragraph, R is certainly reflexive. It is also transitive, because if there exist nodes $x_0, \ldots, x_n$, $x_0 = x$, $x_n = y$, $x_{i-1} \to x_i$ for $1 \leq i \leq n$, and also nodes $y_0, \ldots, y_m$, $y_0 = y$, $y_m = z$, $y_{i-1} \to y_i$ for $1 \leq i \leq n$, then we set $x_{i+n} = y_i$, $1 \leq i \leq m$, and the nodes $x_0, \ldots, x_{n+m}$ then satisfy $x_0 = x$, $x_{n+m} = y_m = z$, and $x_{i-1} \to x_i$ for $1 \leq i \leq n$ and $n+1 \leq i \leq n+m$, i. e., $1 \leq i \leq n+m$. Thus xRy and yRz imply xRz. If xRy and yRx, then the above construction gives a directed path from x to x which can have length zero only if the original two paths do, i. e., x = y. Since the graph is acyclic, this must be the case, which shows that R is antisymmetric.

A node x of any directed graph is initial if $y \to x$ does not hold for any node y. It is <u>terminal</u> if $x \rightarrow y$ does not hold for any node y. These definitions hold in any directed graph; if a graph is acyclic, initial nodes are minimal nodes (in the partial ordering of the above theorem) and terminal nodes are maximal nodes.

There are a number of special set-theoretical conventions which we will use. The notation $A \subset B$ (or $B \supset A$) means that A is a subset of B which is not necessarily proper; that is, it may be that $A = B$ (and we will not use the notation $A \subseteq B$ or $B \supseteq A$). The identity function on any set will be denoted by e. The composition of two functions f and g is denoted by $h = f \circ g$, where $h(x) = g(f(x))$ (and <u>not</u> $f(g(x))$, as used by some authors); thus $f \circ g$ means intuitively "first apply f, then apply g." The inverse of a function f is the function g such that $f \circ g = e$ (not $g \circ f = e$, although this follows immediately). The cartesian product of the sets X_i, for all $i \in I$ (for some set I), will be denoted by $\prod_{i \in I} X_i$ (and not $\bigtimes_{i \in I} X_i$).

3-2 Memories

We shall take the fundamental position that <u>all characteristics of memory may be inferred from the values which are currently assumed by certain variables</u>. As applied to human memory, this statement, of course, is debatable, although there is nothing which prevents us from conjecturing that the human mind, in this way as in so many other ways, is like a computer. If a person's behavior depends on whether he is happy, sad, frustrated, etc., one might postulate a "brain variable" called <u>mood</u>, whose values are "happy," "sad," "frustrated," etc. For the present, however, we shall confine ourselves to considering the memory of a computer, or of an algorithm. Even in this case the statement made above is not at all obvious; one must decide what to do about input-output, currently inactive variables, push-down lists, etc. However, in what follows we shall give evidence indicating that the given statement is in fact applicable to all such situations.

The fundamental relation between variables and their values is that of <u>legality</u>. That is, given a variable $\underline{m}$ and a value $\underline{v}$, one must be able to determine whether $\underline{v}$ is a legal (or allowable or permissible) value of $\underline{m}$. If M is the set of all variables and V is the set of all values, then the set of all ordered pairs $(\underline{m}, \underline{v})$, such that v is a legal value of $\underline{m}$, is a subset of the cartesian product $M \times V$ which we call the <u>legality relation</u> L. It is clear from experience that no finiteness restrictions ought to be imposed on M or V, and, in fact, these may be arbitrary sets. Thus an (ideal) algorithm or computer may have an infinite number of variables, and any of these may have an infinite number of legal values. These considerations are now made precise as follows.

DEFINITION 3-2-1. A memory is an ordered triple $\Gamma = (M, V, L)$, where M and V are arbitrary sets and L is a subset of $M \times V$. The set M is the set of variables of Γ; the set V is the set of values of Γ; and L is the legality relation of Γ.

We have the following simple examples of memories:

(1) $\Gamma = (B, \{0, 1\}, B \times \{0, 1\})$, where B is the set of all bit positions in the registers, core, tapes, etc., of a certain binary digital computer. In this case, every value is always legal.

(2) $\Gamma = (M, V, L)$, where $M = \{a_1, \ldots, a_n, b_1, \ldots, b_m\}$, V is the union of two sets R and I, and L contains the pairs (a_j, r) for $r \in R$ and (b_j, i) for $i \in I$. This is the memory of an algorithm whose declarations are real $a_1, \ldots, a_n$ and integer $b_1, \ldots, b_m$. The sets R and I are sets of real numbers and integers respectively. The legality relation assures us that no real variable may have an integer value, or vice versa.

(3) $\Gamma = (\phi, \phi, \phi)$, the null memory.

Whenever we are given a memory Γ, we may denote its set of variables by Γ^M, its set of values by Γ^V, and its legality relation by Γ^L. Thus in the first example above we may write $\Gamma^M = B$. If one memory is the union of two others, its set of variables, set of values, and legality relation are the respective unions of the corresponding components of those two memories. Thus for the memories Γ_1 and Γ_2, we have

$$\Gamma_1 \cup \Gamma_2 = (\Gamma_1^M \cup \Gamma_2^M, \Gamma_1^V \cup \Gamma_2^V, \Gamma_1^L \cup \Gamma_2^L)$$

In particular, the memory of the second example above is the union of its "real variable memory" $\Gamma_1 = (\{a_1, \ldots, a_n\}, R, \{a_1, \ldots, a_n\} \times R)$ and its "integer variable memory" $\Gamma_2 = (\{b_1, \ldots, b_m\}, I, \{b_1, \ldots,$

$b_m\} \times I)$. This example shows that the property of a memory that every value is always legal is not preserved when we take the union of memories. Another way to specify the definition of the union of two memories is to write

$$(\Gamma_1 \cup \Gamma_2)^M = \Gamma_1^M \cup \Gamma_2^M$$
$$(\Gamma_1 \cup \Gamma_2)^V = \Gamma_1^V \cup \Gamma_2^V$$
$$(\Gamma_1 \cup \Gamma_2)^L = \Gamma_1^L \cup \Gamma_2^L$$

In the interest of brevity, we shall sometimes specify a memory as an ordered pair $\Gamma = (M, V)$, rather than as an ordered triple. In this case we assume that every value is always legal. Thus our three examples above may be written $\Gamma = (B, \{0, 1\})$, $\Gamma = (\{a_1, \ldots, a_n\}, R) \cup (\{b_1, \ldots, b_m\}, I)$, and $\Gamma = (\phi, \phi)$.

The legality relation of a memory may be specified completely by giving a set of (legal) values for each variable. Conversely, such a specification of sets uniquely defines a legality relation. If two variables have the same set of values, we commonly speak of them as being "of the same type"; likewise in programming languages we have quantities called <u>types</u> which specify roughly what values are permitted for a variable. We shall now make this concept precise by defining a type function in an arbitrary memory.

<u>DEFINITION 3-2-2</u>. The <u>type function</u> Γ^T of the memory Γ is the function from Γ^M into the set of all subsets of Γ^V defined by $\Gamma^T(m) = \{v \in \Gamma^V: (m, v) \in \Gamma^L\}$ for each $m \in \Gamma^M$. The set $\Gamma^T(m)$ is called the <u>type</u> of m in Γ.

Thus, in the second example above, R is the type of each variable a_j (i. e., each "real variable") and I is the type of each variable b_j (each "integer variable"). In the first example, $\{0, 1\}$ is the type of every variable.

In practice, the type of a variable depends on whether we are dealing with "ideal" or "actual" situations. In an ideal memory, the type of an integer variable, for example, will be the set of all integers. In an actual memory, this type will be merely the set of all those integers which will fit into a word on some actual computer. This assumes that single precision is used; if multiple precision is allowed, the type will be some larger, but still finite, set. In what follows, we will denote any of these three kinds of type set by integer. All of the key words, in fact, which normally define types in programming languages, such as integer, real, and Boolean, will be used here to denote sets. Of these, only Boolean has a constant, universal meaning, namely Boolean = {true, false}. In particular, real (as used in FORTRAN and ALGOL) may be the set of all real numbers or the set of all allowable floating point numbers. In PL/I, a type is uniquely identified by a particular base, scale, mode, and precision.

3-3 State Vectors

The most important property of the values of variables is that they change during a computation. Every computation goes through a series of "states" T_0, T_1, ..., which may either be infinite (in which case we speak of an "endless loop") or finite. At each point in the computation, the value of every variable is determined; this means that each state corresponds to a function from variables to values. If T is a state and m is a variable, then T(m) is the value of m associated with T. Such states depend on the memory associated with the process; in particular, T(m) must be a legal value of m by the legality relation of the memory. These considerations are formalized as follows.

DEFINITION 3-3-1. A state vector ~~of a~~ memory Γ is a function $T: \Gamma^M \rightarrow \Gamma^V$ such that $(m, T(m)) \in \Gamma^L$ for each $m \in \Gamma^M$. The set of all state vectors of a memory Γ is the state vector domain Γ^D of Γ.

State vectors are also called content functions, and the value T(m), as above, may also be called the contents (or current contents) of m. Our use of the term "state vectors" is partly derived from the fact that, in the following sections of this chapter, we shall be studying various functions whose domain is Γ^D. Such functions will be called state vector functions; it would be slightly awkward to call them "content function functions." State vectors are sometimes also called "snapshots" (from the term "snapshot dump") or simply "states" by analogy with automata theory.

When the set of variables is finite, say $\Gamma^M = \{m_1, \ldots, m_n\}$, every state vector may be identified with an n-tuple, namely $(T(m_1), \ldots, T(m_n))$, where T is the state vector. This construction justifies the use of the term "vector." The domain in this case may be identified with the n-fold cartesian product $X_1 \times \ldots \times X_n$, where $X_i = \Gamma^T(m_i)$, $1 \leq i \leq n$. In general, a state vector domain may be identified in this way with a finite or infinite cartesian product of sets of values of variables.

If $\Gamma = (M, V, L)$ is a memory and M' is a subset of M, the state vector $T: M \rightarrow V$ has the restriction $T': M' \rightarrow V$ defined in the usual way, i. e., $T'(m') = T(m')$ for all $m' \in M'$. As usual, we denote this restriction by $T|M'$. It specifies a legal value for each variable in M', and is therefore itself a state vector of the memory $\Gamma' = (M', V', L')$, where $V' = \{v \in V: \exists\, m \in M' \text{ with } (m, v) \in L\}$ and $L' = L \cap (M' \times V')$. In fact, the domain of Γ' is the set of all restrictions to M' of the domain of Γ. We refer to Γ' as a submemory of Γ and write $\Gamma' \subset \Gamma$. In general,

$$\Gamma_1 \subset \Gamma_2 \Leftrightarrow (\Gamma_1^M \subset \Gamma_2^M, \Gamma_1^V \subset \Gamma_2^V, \Gamma_1^L = \Gamma_2^L \cap (\Gamma_1^M \times \Gamma_1^V))$$

We have defined state vector domains in terms of memories, but there is a sense in which the state vector domain is the more fundamental concept. The situation is analogous to that in physics where density functions, temperature functions, and the like may be defined in terms of coordinate systems, but nevertheless have a nature which is independent of the coordinate system. In the same way, there will generally be a large number of ways of constructing a single memory, and the state vector domain (and all state vector functions) will be essentially the same for all of these ways. To start with, there are two trivial ways in which this is so:

(a) Let $\Gamma = (M, V, L)$ and suppose that certain values in V are always illegal; that is, suppose that $V' = \{v \in V: \exists\, m \in M \text{ with } (m, v) \in L\}$ is a proper subset of V. Then L is a subset of $M \times V'$, and every state vector of Γ is also a state vector of $\Gamma' = (M, V', L)$. In light of this fact we may generally assume, for any memory, that each of its values is always legal for at least one variable. Under this condition it is clear, in particular, that a submemory is completely specified by its set of variables; if $\Gamma_1 \subset \Gamma_2$, then $\Gamma_1^V = \{v \in \Gamma_2^V: \exists m \in \Gamma_1^M$ with $(m, v) \in \Gamma_2^L\}$.

(b) Let $\Gamma = (M, V, L)$ and suppose that some variable in M has no legal values. It is then clear from the definition that this memory has no state vectors at all, and thus becomes uninteresting. But now suppose that there are certain variables, each of which has exactly one legal value. Specifically, let us break up M into two sets, M' and M'', such that if $m \in M''$ there is one and only one $v \in V$ (which we denote by $v(m)$) such that $(m, v) \in L$. If T is a state vector of Γ, then clearly $T(m) = v(m)$ for each $m \in M''$, and T is thus defined solely by its values on M'. If $\Gamma' \subset \Gamma$ is determined by M' as above, then each state vector in the domain of Γ corresponds uniquely to one in the domain of Γ'. Thus the two state vector domains are identical.

In light of this fact we might think that an axiom should be set up specifying at least two legal values for each variable. Such an axiom, however, would not correspond to situations found in practice; read-only memories, for example, are such that each variable in them has one and only one value.

A deeper method of constructing large numbers of memories, all of which have effectively the same state vector domain, may be found by considering ~~some new examples~~ of memories, in addition to the ones mentioned earlier:

(4) $\Gamma = (M_{36}, \{0, 1, \ldots, 2^{36}-1\}) \cup (M_{15}, \{0, 1, \ldots, 2^{15}-1\})$. Here M_{36} is the set of all 36-bit words, registers, etc., in some binary digital computer, and M_{15} is the set of all 15-bit registers.

(5) $\Gamma = (\{A_1, A_2, A_3, B_1, B_2, B_3, C_1, C_2, C_3, D_1, D_2, D_3\}, \underline{\text{real}})$. This represents a collection of four real single arrays, each of dimension 3. Each component of each array is a variable, and its values are real numbers.

(6) $\Gamma = (\{A, B, C, D\}, \underline{\text{real}} \times \underline{\text{real}} \times \underline{\text{real}})$. This also represents four real single arrays of dimension 3. Each array is a variable, and its values are three-dimensional real vectors.

It is easy to see that there is a natural one-to-one correspondence between the state vectors of the last two examples. A state vector of the memory of example 5 gives a real value for each variable; these may be collected into real three-dimensional vectors, one for each array. But this gives us a state vector of the memory of example 6; and the argument clearly works in reverse. The same sort of correspondence may be drawn up between the state vectors of the memory of example 4 and those of example 1, provided that the set B is chosen properly. In fact, B should contain 36 elements (bits) for each element of M_{36}, and 15 elements for each element of M_{15}. A state vector will now give zero or one as a value for each of these bits, and from these we may determine an integer less than 2^{36} as a value for each 36-bit register and less than 2^{15} as a value for each 15-bit register. This gives us a state vector as in example 4.

We shall refer to the passage from example 5 to example 6 above as the process of <u>combining variables</u>. The variables A_1, A_2, and A_3 are combined into a single variable A, and similarly for the other three arrays. In the same way, the bits in the memory of example 1 are combined into words and registers as in example 4. When this is done, the legal values of the variable formed by the combining process are the state vectors of the submemory determined by the variables which were combined. Thus, in example 6 above, $\underline{\text{real}} \times \underline{\text{real}} \times \underline{\text{real}}$

is effectively the set of state vectors of the memory $\Gamma' = (\{A_1, A_2, A_3\}, \underline{\text{real}})$, or of $\Gamma'' = (\{B_1, B_2, B_3\}, \underline{\text{real}})$, etc. The correspondence between the two state vector domains will now be set up in the most general case.

<u>THEOREM</u> <u>3-1-1</u>. Let $\Gamma = (M, V, L)$ be an arbitrary memory and let $\mathfrak{D}$ be a decomposition of M; that is, $\mathfrak{D}$ is a set of disjoint subsets of M whose union is M. For each $D \in \mathfrak{D}$ let D^Γ be the submemory of Γ, as defined above, such that $D^{\Gamma M}$ (i. e., $(D^\Gamma)^M) = D$. Let V' be the union of all $D^{\Gamma D}$ (i. e., $(D^\Gamma)^D$) for all $D \in \mathfrak{D}$, and let $L' \subset \mathfrak{D} \times V'$ be such that $(D, T) \in L'$ if and only if $T \in D^{\Gamma D}$. Then there is a natural one-to-one correspondence between Γ^D and K^D, where $K = (\mathfrak{D}, V', L')$.

<u>PROOF</u>. Let $T \in \Gamma^D$ and define $T' \in K^D$ by $T'(D) = T|D$ for each $D \in \mathfrak{D}$. By the definition of state vectors in D^Γ, we have $T|D \in D^{\Gamma D}$, and thus $(D, T'(D)) \in L'$ for each $D \in \mathfrak{D}$, which shows that T' is actually in K^D. If $T_1, T_2 \in \Gamma^D$ correspond to $T_1', T_2' \in K^D$ in this way, then $T_1' = T_2'$ implies $T_1|D = T_2|D$ for each $D \in \mathfrak{D}$; and thus $T_1 = T_2$ since the union of all $D \in \mathfrak{D}$ is M. If $T' \in K^D$, then we may define $T \in \Gamma^D$ by the following rule: if $x \in M$, then $x \in D$ for some $D \in \mathfrak{D}$, and letting $U = T'(D)$, we define $T(x) = U(x)$. It is then clear that $T|D = U = T'(D)$, so that T' is derived from T by the given correspondence. This completes the proof.

This theorem may be made intuitively clear by considering our state vector domains as cartesian products. The domain Γ^D is the cartesian product of the value sets $\Gamma^T(x)$ for each $x \in M$. The domain K^D is the cartesian product of a collection of subproducts, each of which involves the value sets $\Gamma^T(x)$ for each x in some subset $D \subset M$. The product is the same no matter in what order it is taken.

3-4 Computation Sequences

We shall now take a viewpoint about state vectors which seems, at first glance, to be quite restrictive. Subsequently, however, we shall give evidence indicating that this viewpoint is completely natural in any computing situation, provided only that we take account of <u>all</u> quantities which play the rôle of variables.

When a computation goes through a series of states which are represented by state vectors T_0, T_1, ..., each state vector, obviously, depends on the preceding one. Does it, however, depend on anything else as well? Our viewpoint is that the answer to this question is no; that is, <u>each state vector in a computation is determined by the preceding state vector, and by nothing else</u>. A corollary to this is that <u>anything which might affect the choice of a new state vector must be incorporated into the current state vector</u>. Let us first deal with two points which immediately come to mind:

(1) The values of those quantities which are normally thought of as variables in a program do not, by themselves, determine the next state vector. One must also know which statement is currently being executed. This situation, however, may be treated be admitting a new variable whose values are the statements of the program itself. (Such variables are sometimes called location counters, p-counters, instruction counters, program address registers, etc.) If this variable is called λ, then the value $T(\lambda)$ for any state vector T tells us which statement we are working on. This, in turn, tells us what the next statement is, so that the value of $T'(\lambda)$ for the next state vector T' is determined, as well as all the other values of T'.

(2) If a computational process involves input-output, the next state vector may depend on the values which have been read in. Fur-

thermore, a process may be receiving input from several sources at the same time. This situation is treated by considering each input device as a variable, or, rather, as several variables which may be combined into one by the combining process of the preceding section. Each square on a tape, for example, is a variable, whose legal values are the symbols which may be written on the tape. If all these squares are combined, the legal values of the single resulting "tape variable" are the possible contents of the entire tape. In a different situation, we might wish to combine these variables in a different fashion, so that records or files would be variables, rather than entire tapes. The same considerations hold for card readers, paper tape readers, and all other input devices; each of these is included, when appropriate, as one or more components of the state vector. Note that it does not matter whether our tapes, card decks, etc., are finite or infinite, since the set of variables of a memory may be infinite, and correspondingly an infinite number of variables may be combined into one.

It is also convenient to include output devices as variables, whether or not the information they contain may be read back into the computation. The reason for this is that, when a computation terminates, the resulting output may be determined directly from the final value of the state vector. Output devices, of course, are variables in the same sense that input devices are; that is, each output device may be thought of as a sequence of variables, whose values are characters, printed lines, punched cards, tape squares, etc., or these variables may be combined into a single variable for each output device.

Having justified our point of view, we now make two important definitions.

DEFINITION 3-4-1. An execution function is a total or partial function $f: \Gamma^D \to \Gamma^D$, for some memory Γ which is presumed to contain a location counter λ. Any sequence $T_0, T_1, \ldots$, such that $T_{i+1} = f(T_i)$ for each $i \geq 0$ is called a computation sequence of f.

These definitions imply what we have postulated about computations; that is, each state vector depends on the preceding state vector, and on nothing else. The execution functions are allowed to be incompletely defined; that is, it is not necessarily true that every state vector leads to an identifiable next state vector. The most important case in which this is true is when the value of λ specifies a stopping statement of the program. In this case, the computation sequence terminates. This phenomenon will be taken up in more detail in the next section.

It is clear that every digital computer has an execution function. If we are given the value zero or one for each bit in core, on tape, in the registers, etc., then this determines a state vector. The next state vector is now determined by the current state vector alone. That is, the location counter specifies an instruction; the instruction specifies an address; and the instruction is executed. Simultaneously, any input-output channel has a state, which is part of the current state vector, and the next state vector includes the results of input-output transmission of data determined by this state. It is, of course, true that the timing of input-output introduces some uncertainty into the determination of the next state vector; and we could also imagine a computer being subject to intermittent error, in which the next state vector would be given only in probabilistic terms. Proofs of algorithms under such conditions require special techniques.

Similarly, every program written in a programming language has an execution function. There are a number of features of programming languages which make it seem that the next state vector is determined partially by quantities other than the values of the variables. In each such case, however, these quantities will themselves be considered as variables. Here are a few examples:

(1) A variable whose type can change during the course of a computation has an associated type-variable whose legal values are the applicable types. For example, suppose that we are working with a language in which the statement K = I + J is executed with integer values of K, I, and J, and then executed again in the same program with real (floating point) values of these variables. In order to execute this statement the computer must first interrogate the types of K, I, and J, which are presumably kept in memory at execution time, before deciding whether to use a fixed-point or a floating-point add instruction. In such a situation, our memory will include K, I, and J in its set of variables, and all applicable integers as well as real numbers will be legal values of K, I, and J; but the memory will also contain other variables (let us call them K', I', and J') whose legal values are the sets (real and integer) of all applicable floating point numbers and integers respectively.

(2) A push-down list may be considered as a variable whose values are sequences of quantities which may be pushed down. If Q is the set of all such quantities, then the legal values of the push-down list variable are finite sequences of elements of Q.

(3) Inactive variables in a program are represented by a variable which very closely resembles the one constructed above for a push-down list. Let us consider, for example, a language with block structure. When a block is entered, certain declarations made within the block involve variable names which have already been declared, either in some other block or recursively in the current block.

The current values of these variables, therefore, have to be saved, and this operation is mathematically the same as if they were placed on a push-down list. In practice, this does not actually happen, because it would take too much time; instead, we allocate new storage, and make reference at all times to variables in terms of the current storage allocation. It is quite possible to include storage allocation information itself as variables in memory, and we may construct execution functions on this basis. This, however, has the disadvantage that the meaning of a program is defined in terms of a particular scheme for interpreting or compiling that program.

Another type of inactive variable is found in string processing languages, where the names of those identifiers which are to be given values during a computation may be constructed as strings by the same computation. In such a case, every syntactically valid identifier is potentially a variable in memory. No distinction need be made, however, between potential and actual variables; we may, indeed, consider a memory in which every possible variable name corresponds to a variable.

3-5 State Vector Functions

The execution functions defined in the preceding section are a special example of state vector functions -- that is, functions on some state vector domain. In the next few sections, we shall see how to construct state vector functions that correspond, not only with programs, but with various things that are found in programs such as assignment statements, conditional statements, expressions, terms, and factors.

The correspondence between state vector functions and the constituents of a program provides an answer to what might appear to be a serious problem in our theory: how can we prove anything about execution functions, in light of the fact that most of them which we encounter in practice are so extremely complex? It is clearly no good to have a theory which deals effectively only with small programs, and the execution function of a program is clearly as complex as the program itself. Of even greater complexity is the execution function of a digital computer; indeed, the entire computer operations manual is, in a sense, a description of this function. Much the same thing, of course, could be said about programs themselves; for example, one could say that it takes the entire manual of a programming language to define that language. But when we break up a program into statements, the statements into expressions and the like, the expressions into terms and factors, and so on, we discover that at each individual stage in the analysis the passage from lower to higher complexity is simple and straightforward. A term, for example, is nothing more than a sequence of factors separated by multiplication and division signs. In the same way, when we associate a term, for example, with a state vector function, we do so by combining the state vector functions of its factors in a standard and simply stated way. The fact that a program with a given execution function produces correct results may then be proved in stages, just as would the fact that a program has no syntax errors in it.

These state vector function correspondences will be made precise in the next few sections. For the moment, we give intuitive explanations of seven of the commonest types of state vector functions. In each case we assume that the memory Γ does <u>not</u> include a location counter variable; the domain of each function is Γ^D.

(1) Given a state vector and an <u>assignment statement</u>, we may perform the assignment and obtain a new state vector. An assignment statement therefore corresponds to a function from state vectors to state vectors, i. e., a function $f: \Gamma^D \to \Gamma^D$.

(2) Given a state vector and an <u>arithmetic expression</u>, we can substitute for each variable in the expression its value as given by the state vector, and then evaluate the expression, i. e., find a value for it. An arithmetic expression therefore corresponds to a function from state vectors to values, i. e., a function $f: \Gamma^D \to T$, where T is the set of all possible values of the given expression.

(3) Given a state vector and a <u>statement in a program</u>, we can determine the next statement. If a program P is viewed as a finite set of statements (which may have an additional structure, depending on the programming language) then there thus corresponds to every such statement a function $g: \Gamma^D \to P$ (as well as, in general, a function $f: \Gamma^D \to \Gamma^D$ as above).

(4) Given a state vector and a <u>left side</u> of an assignment, we can determine which variable is changed by that assignment. A left side of an assignment thus corresponds to a function from state vectors to the set of possible variables which may be changed, i. e., a function $f\colon \Gamma^D \to \Gamma^M$. (This assumes that only one variable is being so changed.)

(5) A <u>condition</u> may be viewed as an expression whose set of values is $\{\underline{\text{true}}, \underline{\text{false}}\}$. As in (2) above, there corresponds to the condition a function $f\colon \Gamma^D \to \{\underline{\text{true}}, \underline{\text{false}}\}$. A condition may also be thought of as the set $D' \subset \Gamma^D$ of all state vectors for which the condition is true. This dual view of conditions follows also from Theorem 3-1-3.

(6) Given a state vector and a <u>program</u> (with a designated place to start), we can run the program, and, when it terminates (if it does), we have a new state vector. A program thus corresponds to a function from state vectors to state vectors, i. e., a function $f\colon \Gamma^D \to \Gamma^D$; this function will be partial unless the program always terminates.

(7) Given a state vector and an <u>expression with a side effect</u>, we obtain both a value for the expression and a new state vector which expresses the values of all the variables, some of which may have been changed by the side effect. Such an expression therefore corresponds to two functions -- the expression function as in (2) above, which is a function $f\colon \Gamma^D \to T$, and the side effect, which is a function $s\colon \Gamma^D \to \Gamma^D$ as in (6) above. Conditions may also have side effects; here, as before, we simply set $T = \{\underline{\text{true}}, \underline{\text{false}}\}$.

3-6 Assignments

An assignment in a programming language corresponds to a function from a state vector domain to itself. This is true whether or not it involves a subscripted left side, multiple replacement, type conversion, parameter references, or side effects.

In the simplest type of assignment, there is a single variable on the left side, whose new value is the value of the expression on the right. The new values of all other variables are the same as their old values. These two sentences completely determine the state vector function. As an example, let us consider the memory Γ = $(\{Z, A, B, C\}, \underline{\text{real}})$, and the assignment Z = A + B + C. A state vector corresponds to a 4-tuple (z, a, b, c), where z, a, b, and c are the respective values of Z, A, B, and C. (If the state vector is S, viewed as a function, then the corresponding 4-tuple is (S(Z), S(A), S(B), S(C)).) The function $f: \Gamma^D \to \Gamma^D$ is then defined by

$$f(z, a, b, c) = (a+b+c, a, b, c)$$

In general, provided all variables are real, the memory will contain the above memory as a submemory. In this case we define

$$f(S) = S'$$

where S is an arbitrary state vector and S' is defined in terms of S as follows:

$$S'(Z) = S(A) + S(B) + S(C)$$
$$S'(x) = S(x) \text{ for all } x \neq Z$$

It is immediately apparent that the state vector functions of assignments may be quite complicated. Even in the example above, which is very simple, we have, at the very least, a function of four-dimensional vectors whose value is a four-dimensional vector. Before we go any further, therefore, we need to assure ourselves that the rule, in any situation, for forming the state vector function of an assignment will be simple enough to be easily understood. In order for this to be so, we must proceed step by step, just as indicated in the preceding section. An assignment has a left side and a right side, and each of these has its own corresponding state vector function, which we can assume is known. These functions may now be combined according to a standard rule, which varies from one programming language to another, in order to construct a state vector function for an expression.

Let us review briefly the nature of state vector functions for the left and right sides of an assignment, as indicated in the last section. We assume, first of all, that V_1 is the set of all legal values of the right side, and V_2 is the set of all legal values of the variable on the left side (or of any variable in the array which the left side refers to). Furthermore let $\underline{t}$ be the standard type-conversion function from V_1 to V_2. (As an example, if V_1 = <u>real</u> and V_2 = <u>integer</u>, then $\underline{t}$ is the <u>entier</u> function, with, for example, $t(6.437) = 6$.) If $V_1 = V_2$, then $\underline{t}$ will be the identity function. Let Γ be a memory such that all variables referenced on both sides of the assignment are contained in Γ^M. The left side now corresponds to a function $v: \Gamma^D \rightarrow \Gamma^M$, and the right side corresponds to a function $e: \Gamma^D \rightarrow V_1$. From these we must construct a function $a: \Gamma^D \rightarrow \Gamma^D$ which corresponds to the assignment. This may be done by the following rule:

$$a(S) = S', \ S'(v(S)) = t(e(S)), \ S'(x) = S(x) \text{ for } x \neq v(S)$$

Thus, in the example above, v is the constant function whose value in Γ^M is always Z, while e(S) = S(A) + S(B) + S(C) for each state vector $S \in \Gamma^D$. (The plus sign here refers, of course, to a specific binary operation which must be defined on the set real.) For an example in which v is not a constant function, consider the assignment A[I+1] = A[I], where A is an array. In this case the value of v(S) is the variable A[x] where x = S(I) + 1, while e(S) = S(m) where m is the variable A[S(I)].

The state vector function of an assignment depends not only on the assignment but also on the memory involved. If we add extra variables to our memory, we may immediately derive a new state vector function by assuming that the assignment does not affect the values of any of these new variables. Specifically, if $\Gamma_1 \subseteq \Gamma_2$ and $f_1 : \Gamma_1^D \to \Gamma_1^D$ is the state vector function of an assignment with the memory Γ_1, then the state vector function $f_2 : \Gamma_2^D \to \Gamma_2^D$ of the same assignment with the memory Γ_2 is given by

$$f_2(S) = S', \ S'|\Gamma_1^M = f_1(S|\Gamma_1^M), \ S'(x) = S(x) \text{ for } x \notin \Gamma_1^M$$

We say that f_2 is a memory extension of f_1. In section 2-7, we shall make a very general definition of memory extension, and we shall show that every assignment defined on a finite number of variables has a "primitive memory" containing every memory on which it may be defined; memory extensions then give all state vector functions of that assignment.

When several assignments are executed one after the other, the result may be represented by a single state vector function which is the composition (in the usual sense) of the state vector functions corresponding to the individual assignments. This assumes a single memory large enough to contain all variables which are involved in these assignments. If f_1 and f_2 correspond to the assignments a_1 and a_2, then we shall write $f_1 \circ f_2$ for the function which corresponds to a_1 followed by a_2; here if $g = f_1 \circ f_2$, then $g(x) = f_2(f_1(x))$. Such a composition has the same domain and range as the functions of which it is composed; this corresponds to the fact that a succession of assignments plays the same rôle in a program as a single assignment (for example, one box in a flowchart may contain a succession of assignments). If several assignments have state vector functions on different memories, it is essential first to take the union of all these memories in order to form the state vector function of the sequence of assignments as the composition of all these memory extensions.

It is well known that assignments have different meanings in different programming languages. This fact may be interpreted to mean that <u>the rule for forming the state vector function of an assignment depends on the programming language</u>. The rule we have given assumes that there are no side effects and no references to parameters; these are taken up in sections 3-11 and 3-12 respectively. A separate state vector function rule must be specified for a language allowing multiple assignments. This is one feature of algebraic languages whose meaning is not at all obvious; for example, consider the multiple assignment

```
I = J = M[J] = K            (or I, J, M[J] = K)
```

What does this mean? Do we put K first in I, then in J, and then in M[J]? In this case M[J] will always be M[K]. Or do we put K first in M[J], then in J, and finally in I? This is the sort of question which is most effectively answered by reference to a rule for forming state vector functions. Suppose, for example, that we have already defined the state vector function <u>a</u> for a simple assignment with left side v̲ and right side e̲. Now let v_1, ..., v_n be left sides and let a_1, ..., a_n be the state vector functions of the simple assignments v_1=e, ..., v_n=e. Then the multiple assignment v_1=...=v_n=e (or v_1,...,v_n=e) would be defined, in the absence of side effects, by

$$a_1 \circ \dots \circ a_n$$

in the first case and

$$a_n \circ \dots \circ a_1$$

in the second. If side effects are present, of course, still further rules might be formulated; for example, we might wish to take the side effects from left to right, and then the assignments from right to left.

3-7 Expressions and Conditions

Expressions in programming languages correspond to functions from state vectors to values of the expression. The ways in which these functions are constructed reflect the characteristics of the given programming language.

As an example, consider the expression $A \times B + C$. (As in the preceding section, we defer until later the possibility that A, B, or C might be a parameter, or have a side effect.) If the given memory is $\Gamma = (\{A, B, C\}, \underline{\text{real}})$, so that A, B, and C are real variables, then Γ^D may be identified with $\underline{\text{real}} \times \underline{\text{real}} \times \underline{\text{real}}$. If the values of $A \times B + C$ are also elements of real, then the corresponding state vector function $f: \Gamma^D \to \underline{\text{real}}$ is defined by

$$f(a, b, c) = a \times b + c$$

where $a, b, c \in \underline{\text{real}}$, or by

$$f(S) = S(A) \times S(B) + S(C)$$

This last equation has the advantage that it remains valid when Γ is replaced by some larger memory which contains Γ (as above) as a submemory. These equations, however, are still not precise definitions of the corresponding functions, and, in fact, are subject to some ambiguity. We have not yet specified the precedence of the operators + and $\times$, or, indeed, whether they are subject to any precedence at all. In the language APL, the equation for f would be

$$f(S) = S(A) \times (S(B) + S(C))$$

whereas in FORTRAN, ALGOL, and PL/I it would be

$$f(S) = (S(A) \times S(B)) + S(C)$$

This means that when we are working with expressions, just as with assignments, our state vector functions must be constructed step by step.

An expression may be built up from two subexpressions with an operator between them. The simplest expressions are often called _primary expressions_ or _primaries_; these may be constants, variables, or function references. Parentheses may be handled by considering an arbitrary expression in parentheses to be a primary expression. In APL, the operator following the leftmost primary is applied to that primary and the subexpression to its right; thus A×B+C, for example, is formed from A and B+C. If we know the state vector functions p and q corresponding to A and B+C respectively, we can form from them the function f corresponding to A×B+C. Specifically,

$$p(S) = S(A)$$
$$q(S) = S(B) + S(C)$$
$$f(S) = S(A) \times (S(B) + S(C))$$

and thus

$$f(S) = p(S) \times q(S)$$

This, then, is the general rule for forming _f_ from _p_ and _q_ whenever the operator is ×, assuming that f, p, and q are all defined on the same state vector domain, with the same memory. Any other operator has its corresponding rule.

In FORTRAN, ALGOL, and PL/I, the situation is more complex, because of the existence of precedence. Usually, what is done is to define for each precedence level a specific type of expression which is formed from the expression types at the next lower level using operators at the current level. Thus in ALGOL there are three precedence levels for arithmetic operators and three types of arithmetic expressions, known as *simple arithmetic expressions*, *terms*, and *factors*. (We ignore for the moment the unary minus sign and the use of *if*, *then*, and *else*.) The simple arithmetic expression $A \times B + C$ is made up of the terms $A \times B$ and C, and the term $A \times B$ is in turn made up of the factors A and B. When a simple arithmetic expression is made up of more than two terms, the *rightmost* term is combined with the remaining subexpression, using the operator at the *left* of that term, and similar statements hold for terms and factors. Within this framework, however, the rules for forming state vector functions are much the same as before. If f, p, and q are state vector functions which correspond to $A \times B + C$, $A \times B$, and C respectively, then $f(S) = p(S) + q(S)$; similarly, if f, p, and q correspond to $A \times B$, A, and B, then $f(S) = p(S) \times q(S)$.

All of these rules are subject to slight alteration if the state vector functions to be combined are defined with respect to different memories. If $p: U^D \rightarrow$ *real* and $q: V^D \rightarrow$ *real* are to be combined using the + operator to form $f: W^D \rightarrow$ *real*, then $W = U \cup V$ and

$$f(S) = p(S|U^M) + q(S|V^M)$$

for each state vector $S \in W^D$.

When we have a rule such as $f(S) = p(S) + q(S)$ (or $f(S) = p(S|U^M) + q(S|V^M)$), we would like to write simply $f = p + q$. We shall now justify a form of abbreviation of this kind. Let *plus* be the binary addition function on the set *real*, i. e., *plus*$(x,y) = x + y$; the following construction allows us to extend *plus* to a function *plus**, which is such that we may write $f =$ *plus**(p, q) in either of the above two cases.

DEFINITION 3-7-1. Let $f: X_1 \times \dots \times X_n \rightarrow Y$. Then the *star-extension* $f^*(e_1, \dots, e_n)$, where $e_i: \Gamma_i^D \rightarrow X_i$, $1 \leq i \leq n$, is the function $g: \Gamma^D \rightarrow Y$, where $\Gamma = \Gamma_1 \cup \ldots \cup \Gamma_n$, defined by $g(S) = f(e_1(S|\Gamma_1^M), \ldots, e_n(S|\Gamma_n^M))$ for each state vector $S \in \Gamma^D$.

In the example above, plus is a function from real x real into real, so that $n = 2$ and $X_1 = X_2 = Y =$ real. For an example in which X_1, X_2, and Y are not all the same, consider the definition of a relational expression such as $A+B < C*D$. This is a Boolean expression, and the range of its corresponding state vector function f is therefore {true, false}; the function itself is given by

$$f(S) = \begin{cases} \underline{\text{true}} & \text{if } S(A)+S(B) < S(C)*S(D) \\ \underline{\text{false}} & \text{otherwise} \end{cases}$$

If p and q are state vector functions corresponding to A+B and C*D respectively, then we may write $f = (p < q)$, or, using star-extensions, $f =$ less*(p,q), where less is a function from real x real into {true, false} with less(a,b) = true if $a < b$ and false otherwise. For an example in which $n = 3$, let Y be any set and define cond: {true, false} × Y × Y → Y by cond(true, y, y') = y and cond(false, y, y') = y'. If f and f' are state vector functions with values in Y which correspond respectively to e and e', while g corresponds to the Boolean expression b, then cond*(g, f, f') will correspond to (if b then e else e'). Here if Y = real, we get a conditional real expression; if Y = {true, false}, a conditional Boolean expression; if Y is itself some state vector domain, a conditional assignment; and so on. In this last case, where e and e' are assignments, we can abbreviate by writing (if b then e) if e' is the null assignment, corresponding to the identity state vector function.

There are various miscellaneous rules for forming state vector functions of expressions. Unary operators may be treated using star-extensions, just like binary operations; if neg: $Y \to Y$ is the function given by neg(y) = $-y$, then neg*(f) is the state vector function of $-e$ whenever f is the state vector function of e. In this case we may write more simply $f \circ$ neg, rather than neg*(f). Constants k have constant state vector functions $f(S) \equiv k$; simple variables y have state vector functions $f(S) = S(y)$. Array references $A[e_1, \ldots, e_n]$ have state vector functions $a^*(f_1, \ldots, f_n)$, where the f_i are state vector functions corresponding respectively to the e_i, and $a(i_1, \ldots, i_n)$ stands for a variable (on the left side of an assignment) or its current value (on the right side). Function references $F[e_1, \ldots, e_n]$ have state vector functions $f^*(f_1, \ldots, f_n)$, where the f_i correspond to the e_i as above and f is the function computed by F. This assumes, again, that there are no side effects. We shall see in section 3-11 that a different kind of star-extension must be used when side effects are present.

The state vector functions of expressions are subject to a form of memory extension. If $f: \Gamma_0^D \to Y$ corresponds to an expression and $\Gamma \supseteq \Gamma_0$, the function $g: \Gamma^D \to Y$ corresponding to this same expression may be defined by $g(S) = f(S|\Gamma_0^M)$. If a defining relation such as $f(S) = S(A) * S(B) + S(C)$ holds for f, then a corresponding relation such as $g(S) = S(A) * S(B) + S(C)$ holds for g, since for any variable $V \in \Gamma_0^M$ we have $S(V) = S'(V)$, where $S' = S|\Gamma_0^M$.

Conditions in programming languages, let us repeat, are treated exactly like expressions. They correspond to state vector functions whose values are either true or false.

3-8 Memory Extensions

We now define memory extensions in their complete generality, and show that all state vector functions arise by memory extension from primitive ones, provided that the underlying memory is finite. (A memory Γ is said to be finite if Γ^M is a finite set.) For the state vector functions of expressions $f(x_1, \ldots, x_n)$ and assignments $y = f(x_1, \ldots, x_n)$, the corresponding primitive functions may be identified with f itself, viewed in the proper way as a state vector function.

DEFINITION 3-8-1. Let A, B, U, V be memories with $A \subseteq U$, $B \subseteq V$, and let $f: A^D \to B^D$. Furthermore assume that $V \subseteq B \cup U$. Then the memory extension $g: U^D \to V^D$ of f is defined by $g(S) = S'$, where $S'(x) = S(x)$ for $x \notin B^M$ and $S'(x) = S''(x)$ for $x \in B^M$, with $S'' = f(S|A^M)$.

Up to now we have not treated functions from one state vector domain to a different one. We will now show how an arbitrary function of n arguments may be regarded as just such a function. Let the arguments be x_i, $1 \leq i \leq n$, and let each argument x_i be required to have its values in a set X_i. If A is the union of all the one-element memories $(\{x_i\}, X_i)$, then A^D may be identified with the cartesian product $X_1 \times \ldots \times X_n$, which is the domain of f. Now let Y be the set of all possible values of f, and let B be the one-element memory $(\{y\}, Y)$. Then B^D may be identified with Y, since each state vector in it has exactly one component, which is an element of Y. Since Y is the range of f, we have represented f as a function from A^D to B^D.

Let us now consider memory extensions of this state vector function. We may add new variables to either A or B, or both; the condition $V \subseteq B \cup U$ means that any new variables added to B must either be in A or added to A. This is necessary, since the new values of these variables must be the same as their old values, and the old values must therefore exist. Specifically, S is a function on U, and S' on V; in the definition of S', if $x \notin B^M$, then $x \in U^M$ and thus S(x) is defined. We are particularly concerned with two special cases of the condition $V \subseteq B \cup U$:

(a) V = B. Here V^D may be identified with Y, and the memory extension $g: U^D \rightarrow Y$ of f is defined by $g(S) = f(S|A^M)$. The result is precisely the general form for the state vector function of an expression, as mentioned at the end of the preceding section.

(b) V = U. In this case $V^D = U^D$, and the memory extension $g: U^D \rightarrow U^D$ of f is defined by $g(S) = S'$, where $S'(x) = S(x)$ for $x \neq y$ and $S'(y) = f(S|A^M) = f(S(x_1), \ldots, S(x_n))$. This is precisely the general form for the state vector function of an assignment, such as the one given as an example at the beginning of section 3-6.

We now consider the problem of finding memories A and B, as small as possible, and a function $f: A^D \rightarrow B^D$ which may be extended to a given function $g: U^D \rightarrow V^D$ in the manner described above. If v is any variable in V^M whose value never changes under the action of g, then it is not necessary for v to belong to B^M, since the memory extension will express precisely the fact that the value of v never changes. Hence B^M should consist of those variables in V^M whose values may change, as well as those which have no old values, i. e., which are not in U^M. The memory B will thus include all possible results of the computation represented by g. Likewise, if u is any variable in U^M whose value cannot affect any possible result of g, then u does not need to belong to A^M, since these results, i. e., the values of g(S) on B^M, depend only on the values of S on A^M by the definition of memory extension. Hence A^M should consist of those variables in U^M whose values may affect the results of g. The memory A will thus include everything which is used by the computation represented by g. We may think of g as "taking its input from A" and "leaving its output in B." This is now made precise as follows.

DEFINITION 3-8-2. Let $g: U^D \to V^D$. Then the output region $\mathrm{OR}(g)$ is defined as $\{v \in V^M$: not $(S'(x) = S(x)$ for all $S \in U^D$, where $S' = g(S))\}$ and the input region $\mathrm{IR}(g)$ is defined as $\{u \in U^M: \exists S_1, S_2 \in U^D, y \in \mathrm{OR}(g)$, with $S_1(z) = S_2(z)$ for all $z \neq u$, but $S_1'(y) \neq S_2'(y)$, where $S_1' = g(S_1)$, $S_2' = g(S_2)\}$. The input memory $\mathrm{IM}(g)$ and the output memory $\mathrm{OM}(g)$ are those submemories of U and V respectively which are such that $(\mathrm{IM}(g))^M = \mathrm{IR}(g)$ and $(\mathrm{OM}(g))^M = \mathrm{OR}(g)$. The input domain $\mathrm{ID}(g)$ is defined to be $(\mathrm{IM}(g))^D$; the output domain $\mathrm{OD}(g)$ is defined to be $(\mathrm{OM}(g))^D$.

The rather awkward definition of $\mathrm{OR}(g)$ is meant to cover two cases; $S(x)$ may not exist, or it may exist but be unequal to $S'(x)$ for some $S \in U^D$. Clearly, if $U^D = V^D$, for example, only the second of these cases applies. The definition of $\mathrm{IR}(g)$ depends on that of $\mathrm{OR}(g)$, which may seem like putting the cart before the horse. However, this definition is necessary, as may be seen by the theorems which follow. The first of these is valid with no restrictions on the size of the memory.

THEOREM 3-8-1. Let $g: U^D \to V^D$ be a memory extension of $f: A^D \to B^D$. Then $\mathrm{IR}(g) \subseteq A^M$ and $\mathrm{OR}(g) \subseteq B^M$.

PROOF. If $v \in V^M$ and $v \in B^M$, then $S(v) = S'(v)$ for all $S \in U^D$, where $S' = g(S)$, and this implies that $v \notin \mathrm{OR}(g)$. Hence $\mathrm{OR}(g) \subseteq B^M$. If $u \in U^M$ and $u \notin A^M$, then consider $S_1, S_2 \in U^D$ with $S_1(z) = S_2(z)$ for $z \neq u$. Since $S_1 | A^M = S_2 | A^M$, we have $f(S_1 | A^M) = f(S_2 | A^M)$, and thus $S_1'(v) = S_2'(v)$ for all $v \in B^M$, where $S_1' = g(S_1)$, $S_2' = g(S_2)$. Thus $u \notin \mathrm{IR}(g)$, and therefore $\mathrm{IR}(g) \subseteq A^M$. This completes the proof.

This theorem does not imply that g is always actually a memory extension of f with $\mathrm{IR}(g) = A^M$ and $\mathrm{OR}(g) = B^M$. The proof of this statement for finite memory depends upon the following lemma.

LEMMA 3-8-1. Let $g: U^D \to V^D$, where U is finite (i. e., where U^M is finite), and let $S_1, S_2 \in U^D$. If $S_1|\mathrm{IR}(g) = S_2|\mathrm{IR}(g)$, then $S_1'|\mathrm{OR}(g) = S_2'|\mathrm{OR}(g)$, where $S_1' = g(S_1)$, $S_2' = g(S_2)$.

PROOF. Let $u_1, \ldots, u_n$ be those elements of the finite set U^M for which $S_1(u_i) \neq S_2(u_i)$; by hypothesis, each $u_i \notin \mathrm{IR}(g)$. Let $T_0, \ldots, T_n \in U^D$ be defined by $T_i(u_j) = S_1(u_j)$ for $i < j$, $T_i(u_j) = S_2(u_j)$ for $i \geq j$, $T_i(x) = S_1(x) = S_2(x)$ for x not equal to any u_j. Then $T_0 = S_1$, $T_n = S_2$, and $T_i(z) = T_{i-1}(z)$ for all $z \neq x_i$, $1 \leq i \leq n$. If $U_i = g(T_i)$, $0 \leq i \leq n$, then $U_0 = S_1'$, $U_n = S_2'$, and $U_i(y) = U_{i-1}(y)$ for each $y \in \mathrm{OR}(g)$, $1 \leq i \leq n$, by definition of $\mathrm{IR}(g)$. Hence $U_0(y) = U_n(y)$ for each $y \in \mathrm{OR}(g)$, i. e., $S_1'|\mathrm{OR}(g) = S_2'|\mathrm{OR}(g)$. This completes the proof.

THEOREM 3-8-2. Let $g: U^D \to V^D$, where U is finite. Then there exists a function $f: A^D \to B^D$, where $A^D = \mathrm{ID}(g)$, $B^D = \mathrm{OD}(g)$, such that g is a memory extension of f.

PROOF. Let $S \in \mathrm{ID}(g)$ and consider any $S_1 \in U^D$ such that $S_1|\mathrm{IR}(g) = S$. We define f(S) to be S_1' $\mathrm{OR}(g)$, where $S_1' = g(S_1)$. If S_2 is any other state vector in U^D such that $S_2|\mathrm{IR}(g) = S$, then, by the lemma, $S_2'|\mathrm{OR}(g) = S_1'|\mathrm{OR}(g)$, where $S_2' = g(S_2)$, and thus f is a well-defined function. The memory extension of f to a function $h: U^D \to V^D$ is such that $h(S) = S'$, where $S'(x) = S(x)$ for $x \notin \mathrm{OR}(g)$ and $S'(x) = S''(x)$ for $x \in \mathrm{OR}(g)$, where $S'' = f(S|\mathrm{IR}(g))$. But if we set $S' = g(S)$, then $S'(x) = S(x)$ for $x \notin \mathrm{OR}(g)$ by definition of $\mathrm{OR}(g)$, whereas $S'|\mathrm{OR}(g) = S''|\mathrm{OR}(g)$, where $S'' = f(S|\mathrm{IR}(g))$, by definition of f. Thus g = h, and the theorem is proved.

Neither the preceding theorem nor the lemma holds in the presented form if the memory is allowed to be infinite. A slight extension of the concepts of memory and state vector domain is needed in this case for the theorem to hold. Let M and V be finite sets, let L be a subset of $M \times V$ denoting a legality relation, and let T be a particular state vector. We now define a memory as a 4-tuple (M, V, L, T), and we define its domain as the set of all state vectors which differ from T in only a finite number of places. (For example, M might contain an infinite number of variables denoting tape squares, and T might be the state vector such that T(x), for each tape square x, is a "blank character." The domain is then the set of all state vectors which specify only a finite number of non-blank characters on the tape.) With respect to this extension of the definition, the lemma and theorem above hold true.

3-9 <u>Programs</u>

We may now carry our analysis one step further, and derive the state vector function of a program from the state vector functions of the assignments and other types of statements in it.

In section 3-4, we associated with each program an execution function $f: \Gamma^D \rightarrow \Gamma^D$, for some memory Γ, one of whose variables is presumed to be a location counter λ. The set of legal values of λ represents the finite set of statements of the program. From the definition of f we deduce the statement that

> <u>if we know the values of all variables including λ</u>
> <u>then we know the new values of all these variables</u>

Let us now construct a sentence equivalent to the above:

> <u>if we know the value of λ then</u>
> (<u>if we know the values of all variables</u>
> <u>other than λ</u>, <u>then we know the new values</u>
> <u>of all these variables</u>)
> <u>and if we know the value of λ then</u>
> (<u>if we know the values of all other vari-</u>
> <u>ables</u>, <u>then we know the new value of λ</u>)

This suggests that with each value of λ, i. e., with each statement in our program, we may associate two functions on Γ_0^D, where Γ_0 is the submemory of Γ obtained by excluding λ. One of these has range Γ_0^D and gives us the new values of all variables except λ; the other gives the new value of λ, and its range is the set of statements in the program. This will now be made precise as follows.

<u>DEFINITION 3-9-1</u>. A <u>program</u> on a memory Γ is a finite set P

of statements, such that there is associated with every statement Σ an action $\Sigma^A: \Gamma^D \to \Gamma^D$ and a next-statement function $\Sigma^N: \Gamma^D \to P$.

We have the following simple examples of statements:

(1) An assignment statement consists of an assignment which is placed in a program in such a way that it is clear which statement comes next. If the assignment statement is Σ and the next statement is Σ', then Σ^A is the state vector function corresponding to the assignment as in section 3-6, and Σ^N is the constant function whose value for any state vector is Σ'.

(2) A go-to statement Σ has null action, i. e., Σ^A is the identity function, whereas Σ^N is the constant function whose value for any state vector is the statement which carries the label referenced in Σ.

(3) A conditional go-to statement Σ has null action (unless there are side effects in the condition) and a next-statement function with k different values, for a k-way branch. For an ordinary two-way branch Σ with next statement Σ', Σ^N is the function whose value is the statement to which transfer is made if the given state vector satisfies the given condition, and whose value is Σ' otherwise. As before, Σ^A is the identity function. If the given condition is always true, we obtain the ordinary go-to statement as a special case.

Our definition of a program was motivated by the desire to be able to construct an execution function. Before defining the execution function of an arbitrary program, we consider what happens when either Σ^A or Σ^N, for some statement Σ, is a partial function. This is by no means an uncommon occurrence. If S is a state vector and $\Sigma^A(S)$ is not defined, this means, intuitively, that computation cannot be performed for some reason. For example, Σ might be an assignment statement which makes subscript references, and S might be a state vector which specifies one of the subscripts to be outside its proper range. On the other hand, if $\Sigma^N(S)$ is not defined, this means that there is no next statement, i. e., this is the end of the computation. If the last statement in a program is a conditional transfer statement Σ, then $\Sigma^N(S)$ will be defined if and only if S satisfies the given condition.

Now let P be a program on a memory Γ_0 and let Γ be the memory Γ_0 with the location counter λ added, i. e., $\Gamma = \Gamma_0 \cup (\{\lambda\}, P)$. Then Γ^D may be identified with $\Gamma_0^D \times P$, since a state vector in Γ^D is a function $T: \Gamma^M \rightarrow \Gamma^V$ which is determined completely by $T|\Gamma_0^M \in \Gamma_0^D$ and $T(\lambda) \in P$. Using this correspondence, we would like to define the execution function of P as the function $f: \Gamma^D \rightarrow \Gamma^D$, where $f(S, \Sigma) = (\Sigma^A(S), \Sigma^N(S))$; this is in line with our previous intuitive discussion. However, Σ^A and Σ^N may be partial functions, and it is quite possible for some S and some Σ that $\Sigma^A(S)$ is defined but $\Sigma^N(S)$ is not, or vice versa. We can handle functions which are only partially defined, but we cannot handle functions whose values are themselves functions which are only partially defined. Therefore, in order to define our execution functions, we must be sure that the above-mentioned behavior does not occur. We therefore make the following definition.

DEFINITION 3-9-2. A program P is <u>complete</u> if, for each statement Σ of P, the domain of Σ^A is the same as the domain of Σ^N. If P is a complete program on a memory Γ_0, the <u>execution function</u> of P is the function $f: \Gamma^D \rightarrow \Gamma^D$, where $\Gamma = \Gamma_0 \cup (\{\lambda\}, P)$, defined by $f(T) = T'$ where $T'|\Gamma_0^M = (T(\lambda))^A(T|\Gamma_0^M)$ and $T'(\lambda) = (T(\lambda))^N(T|\Gamma_0^M)$. (By the definition of a complete program, one of these will be defined when and only when the other one is.)

An important special case of completeness is the condition that Σ^A and Σ^N are either (a) both everywhere defined or (b) both nowhere defined, for an arbitrary statement Σ of P. A statement satisfying condition (b) above may be called an <u>exit statement</u>; it serves no purpose in the computation itself, but merely denotes where that computation ends.

3-10 The Effect of a Program

The execution function $\underline{f}$ of a program gives the state vector $f(T)$ which results from executing a single step of the program, given the state vector T. The effect of a program (when started at a given point) is a function $\underline{e}$ which gives the state vector $e(S)$ which results from running the entire program, having started with the state vector S. This is, in general, a partial function, since $e(S)$ is undefined whenever the initial state vector S directs the program to run endlessly.

When a program has an execution function — i. e., when it is complete — its effects are very easy to define. (We will often abbreviate and simply refer to "the effect of a program," but, strictly speaking, an effect of a program is always with respect to some particular starting point.) Let the program P be defined on Γ_0, and let $\Gamma = \Gamma_0 \cup (\{\lambda\}, P)$ and $\Gamma^D = \Gamma_0^D \times P$ as in the preceding section. If $S \in \Gamma_0^D$ and Σ is the designated starting point, then $(S, \Sigma) = T_0 \in \Gamma^D$. We now define T_1, T_2, T_3, etc., by $T_i = e(T_{i-1})$, where e is the execution function of P. The T_i will then be the successive state vectors of Γ^D which constitute the computation sequence (see section 3-4) of P when started at T_0. If this computation sequence is finite, let T_z be its last element; then we define $f(S) = T_z | \Gamma_0^D$, where f is the effect of P with respect to Σ. That is, T_z will be of the form $(f(S), \Sigma')$, for some statement Σ'.

We will now make a slightly broader definition of the effect of a program, which includes the possibility that the program is incomplete. Let P, Γ_0, Γ, S, Σ, and T_0 be as above. Let $S_0 = S$ and $F_0 = \Sigma$; then $T_0 = (S_0, F_0)$. Let $S_{i+1} = F_i^A(S_i)$ and $F_{i+1} = F_i^N(S_i)$ for each $i \geq 0$ for which both S_i and F_i are defined; we set $T_i = (S_i, F_i)$. Let $\mathfrak{e}$ be the function we are seeking to define, i. e., the effect of P when started at Σ. There are now four cases:

(1) S_i and F_i are defined for all i; in this case $\mathfrak{e}(S)$ is undefined.

(2) S_i and F_i are defined only for $i \leq z$, for some z; in this case, $\mathfrak{e}(S) = S_z$. (These are the two cases we discussed above.)

(3) S_i is defined for $i \leq z+1$, and F_i for $i \leq z$. Thus $T_z = (S_z, F_z)$ has the property that $F_z^A(S_z)$ is defined but $F_z^N(S_z)$ is not. In other words, this is the last statement of the program, but it has an action; it is not an exit statement. The result of that action should therefore be the value of $\mathfrak{e}(S)$, i. e., $\mathfrak{e}(S) = S_{z+1}$.

(4) S_i is defined for $i \leq z$, and F_i for $i \leq z+1$; thus $F_z^N(S_z)$ is defined, but $F_z^A(S_z)$ is not. Thus a next statement would be defined if the action were defined, but the action is not in fact defined. This means that the computation sequence has terminated improperly. Since it may be necessary in this case to examine what the program has done so far, we define this as the effect, i. e., $\mathfrak{e}(S) = S_z$.

Noting that in cases (2), (3), and (4) above $\mathfrak{e}(S)$ is defined as the last of the S_i to be defined, we make our formal definition.

<u>DEFINITION</u> <u>3-10-1</u>. Let P be any program on Γ_0 and let $f\colon \Gamma^D \to \Gamma^D$ be its execution function, where $\Gamma = \Gamma_0 \cup (\{\lambda\}, P)$ and Γ^D is identified canonically with $\Gamma_0^D \times P$. Then the <u>effect</u> $e\colon \Gamma_0^D \to \Gamma_0^D$ of P with respect to the starting statement Σ of P, or simply the "effect of P when started at Σ," is defined as follows. Let $S_0 = S$, $F_0 = \Sigma$, and, if S_i and F_i are defined, $i \geq 0$, let $S_{i+1} = F_i^A(S_i)$ and let $F_{i+1} = F_i^N(S_i)$. If each S_i is defined, for $i \geq 0$, then $e(S)$ is undefined; if the S_i are defined for $i \leq z$ only, then we define $e(S) = S_z$.

We now show that any program may be modified very slightly so that it becomes complete, without changing any of its effects. This will justify the restriction of some of our later definitions to the case of a complete program.

THEOREM 3-10-1. Given any program P on Γ_0, there is a complete program P' on Γ_0 (called the completion of P) such that:

(1) The statements of P' are the statements of P plus two exit statements (call them U and V).

(2) The action of each statement of P is a restriction of its action as a statement of P'.

(3) The effect of P when started at any of its statements is the same as the effect of P' when started at that statement.

(4) Whenever P' terminates improperly, it terminates at U.

(5) Whenever P' terminates properly, it terminates at V.

PROOF. We define the action of each statement of P' (except U and V, whose action is nowhere defined) to be the action of P wherever that is defined, and to be the identity whenever the action of P is not defined. We define the next-statement function of each statement of P' to have the value V whenever its value in P is undefined; to have value U when its value in P is defined but the value of the corresponding action is undefined; and to have the same value as it does in P when that value and the value of the corresponding action are both defined. U and V, of course, have next-statement functions which are nowhere defined. Properties (1)-(5) above may now be verified in a straightforward manner.

A program with $\underline{n}$ statements may be replaced by an altered program of at most $\underline{n}^2+2\underline{n}$ statements, each of which is of one of the special forms discussed above, i. e., an assignment (with possible side effects) or conditional transfer, and without changing any of its effects, according to the following theorem.

<u>THEOREM</u> <u>3-10-2</u>. Let P be a program on Γ_0; then there exists a program P' on Γ_0 such that:

(1) Each statement of P corresponds to a statement of P';

(2) The effect of P when started at a given statement is the same as the effect of P' when started at the corresponding statement;

(3) Each statement of P' is either an assignment statement with possible side effects (i. e., its next-statement function has at most one value) or a conditional transfer without side effects (i. e., its action is the identity function).

<u>PROOF</u>. Let $\Sigma_1, \ldots, \Sigma_n$ be the statements of P. The statements of P' are of the forms Σ_i' and Σ_i'', $1 \leq i \leq n$, and Σ_{ij}, $1 \leq i \leq n$, $1 \leq j \leq n$. The statements Σ_i' are conditional transfers without side effects; that is, each $\Sigma_i'^A$ is the identity function, while $\Sigma_i'^N(S) = \Sigma_{ij}$ whenever $\Sigma_i^N(S) = \Sigma_j$, and $\Sigma_i'^N(S) = \Sigma_i''$ whenever $\Sigma_i^N(S)$ is undefined. The statements Σ_i'' are assignments with possible side effects; we have $\Sigma_i''^A = \Sigma_i^A$, while $\Sigma_i''^N$ is nowhere defined. The statements Σ_{ij} are also assignments with possible side effects; we have $\Sigma_{ij}^A = \Sigma_i^A$, and Σ_{ij}^N has the single value Σ_j'. The statements Σ_i' corresponds to the statements Σ_i, and thus conditions (1) and (3) above are satisfied. To verify condition (2), let Σ be any statement of P and let $T_0 = (S, \Sigma)$, $T_0' = (S, \Sigma')$. Let T_i and T_i' be the computation sequences of P and P' starting with T_0 and T_0' respectively. We now show that, if $T_x = (S_x, F_x)$, then $T_{2x}' = (S_x, F_x')$, where F_x' corresponds as above to F_x. This is true for $x = 0$;

so we may assume that it is true for $x = i$. We have $T_{i+1} = (F_i^A(S_i), F_i^N(S_i))$; but, by definition, $T'_{2i+1} = (F_i'^A(S_i), F_i'^N(S_i)) = (S_i, \Sigma_{kj})$ where $F_i = \Sigma_k$ and $F_i^N(S_i) = \Sigma_j$, and thus T'_{2i+2} $(= T'_{2(i+1)}) = (\Sigma_{kj}^A(S_i), \Sigma_{kj}^N(S_i)) = (\Sigma_k^A(S_i), \Sigma'_j) = (F_i^A(S_i), \Sigma'_j)$, where Σ'_j corresponds to $\Sigma_j = F_i^N(S_i)$. Thus the given statement is true for $x = i+1$. Now let e and e' be the effects of P and P' starting with Σ and Σ' respectively. If the sequence T_i continues indefinitely, then so does the sequence T'_i, so that e(S) and e'(S) are both undefined. Otherwise, let $T_z = (S_z, F_z)$ be the last of the T_i, with $T'_{2z} = (S_z, F'_z)$. If $S_{z+1} = F_z^A(S_z)$ is undefined, then $e(S) = S_z$, whereas $e'(S) = S_z$ whether T'_{2z} or T'_{2z+1} is the last of the T'_i. If S_{z+1} is defined, then $e(S) = S_{z+1}$. Since T_z is the last of the T_i, $F_z^N(S_z)$ is undefined, and thus $T'_{2z+1} = (S_z, \Sigma''_k)$, where $F_z = \Sigma_k$; $\Sigma_k''^N(S_z)$ is nowhere defined, but $\Sigma_k''^A(S_z)$ is, so that $e'(S) = \Sigma_k''^A(S_z) = \Sigma_k^A(S_z) = F_z^k(S_z) = S_{z+1}$. Thus in all cases $e(S) = e'(S)$ or both are undefined, and the theorem is proved.

We may specialize still further and allow each of our conditional transfers to be a two-way conditional transfer, since a k-way conditional transfer is the composition of k-1 two-way transfers. It is not possible in general, however, to restrict ourselves to assignment statements _without_ side effects. An exchange statement, for example, is one-to-one as a state vector function, whereas any assignment which is not the identity is not one-to-one, as long as everything is finite (in this case, as long as each variable can take only a finite number of distinct values). It is clear, of course, that a one-to-one, non-identity function cannot be the composition of functions which are not one-to-one.

3-11 Side Effects

We are now in the position to be able to define rigorously what it means for an expression, term, factor, etc., to have a side effect. Let us consider the FORTRAN statement

$$Z = A + F(X) + B$$

Here the memory Γ includes Z, A, B, and X, and also any intermediate variables which are used in computing F. Assuming that all these variables have values in the set <u>real</u>, the expression A + F(X) + B has a state vector function $f: \Gamma^D \rightarrow$ <u>real</u>. It also has the side effect $s: \Gamma^D \rightarrow \Gamma^D$, which is the effect (in the technical sense introduced in the preceding chapter) of the function F, when started at its designated starting point.

Any expression, term, factor, or condition which has a side effect is associated with <u>two</u> state vector functions. The domain of each of these functions is the given state vector domain. The first of these functions represents the expression, term, factor, or condition in the usual way, and its range is the set of its values in the usual sense. The second is the side effect, and its range is the same as its domain. An assignment, even with a side effect, has only one state vector function, since an assignment and the effect of a function have the same form as state vector functions.

When an expression has several side effects, the composition of the effects of the corresponding functions is called <u>the</u> side effect of the given expression. For example, the expression

$$A + F(X) + B + G(Y)$$

has the side effect formed by composing the functions $f: \Gamma^D \rightarrow \Gamma^D$

and $g: \Gamma^D \to \Gamma^D$ which are the effects of the functions F and G respectively. That is, the side effect of the expression is $h: \Gamma^D \to \Gamma^D$, where $h(S) = g(f(S))$ for each $S \in \Gamma^D$. The order of composition might be the other way around under different conditions; for example, consider the expression

$$F(X) + B*G(Y)$$

Here it would seem that the order should be the same as above, but in many compilers an expression of the form A+B*C is compiled as "load B, multiply by C, add A." Unless a special form of compilation is used when there are side effects, we will have $h(S) = f(g(S))$.

The composition of side effect functions in expressions, terms, factors, and conditions may always be specified to be from left to right by using the <u>star-extension with side effects</u>, as follows.

<u>DEFINITION 3-11-1</u>. Let $f: X_1 \times \dots \times X_n \to Y$, and let $s_0: \Gamma_0^D \to \Gamma_0^D$. Then the <u>star-extension</u> $(f, s_0)*((e_1, s_1), \dots, (e_n, s_n))$, where $e_i: \Gamma_i^D \to X_i$ and $s_i: \Gamma_i^D \to \Gamma_i^D$, $1 \leq i \leq n$, is the pair (g, s) for $g: \Gamma^D \to Y$ and $s: \Gamma^D \to \Gamma^D$, where $\Gamma = \Gamma_0 \cup \Gamma_1 \cup \dots \cup \Gamma_n$, defined as follows. Let $s_i': \Gamma^D \to \Gamma^D$ be the memory extension of s_i to Γ^D, for $0 \leq i \leq n$, and for each state vector $S \in \Gamma^D$ let $S_1 = S$ and $S_{i+1} = s_i'(S_i)$, $1 \leq i \leq n$; then $g(S) = f(e_1(S_1|\Gamma_1^M), \dots, e_n(S_n|\Gamma_n^M))$ and $s(S) = s_0'(S_{n+1})$.

In this very general definition, each e_i corresponds to an expression and s_i, $i > 0$, corresponds to its side effect, while s_0 is the side effect of f. Any or all of these side effects may be the identity function, in which case we say that certain expressions "have no side effects"; if all side effects are the identity, this form of star-extension reduces to that of Definition 3-7-1. If e_i and s_i, for some i, are originally defined on different state vector domains, say A_i^D and B_i^D, then we may define $\Gamma_i = A_i \cup B_i$ and make the memory extensions of e_i and s_i to Γ_i^D before proceeding. Note that if we write $S_n = S$, $S_{i-1} = s_i'(S_i)$, and $s(S) = s_0'(S_0)$ in the above, we obtain right-to-left evaluation, rather than left-to-right; in either case, the function evaluation, with its side effect, follows all argument evaluations with their side effects.

The above definition allows us to build up an expression from primaries with and without side effects. If such an expression is the right side of an assignment, there will be two different plausible interpretations for that assignment if its left side is subscripted and the side effect of the expression changes the value(s) of the subscript(s). The problem here, of course, is whether we should use the new value(s) or the old. Thus in the assignment A(I) = F(I), where I is initially 1, A(I) is a subscripted variable, and F is a function which increments I by 1, we obtain A(2) = F(I) according to the first interpretation above and A(1) = F(I) according to the second. If $v: \Gamma^D \rightarrow \Gamma^M$ corresponds to the left side of our assignment and $e: \Gamma^D \rightarrow V$ corresponds to the right side, with the side effect $s: \Gamma^D \rightarrow \Gamma^D$ and the type-conversion function t, then the resulting assignment corresponds to a function $a: \Gamma^D \rightarrow \Gamma^D$, defined by $a(S) = S'$, where

$$S'(v(s(S))) = t(e(S)),\ S'(x) = S''(x) \text{ for } x \neq v(s(S)) \text{ with } S'' = s(S)$$

according to the first interpretation above and

$$S'(v(S)) = t(e(S)),\ S'(x) = S''(x) \text{ for } x \neq v(S) \text{ with } S'' = s(S)$$

according to the second. The practical difference between these two viewpoints is that the first corresponds to a simpler interpretational strategy, according to which the entire right side is handled first, then the result stored in the place indicated by the left side; whereas the second presents the simpler viewpoint to the user, according to which the expression is considered from left to right in the sense that whatever happens on the right does not affect what happens on the left. To implement this strategy, we first calculate and save the subscript(s) on the left, then treat the right side, retrieve the subscript(s) on the left, and finally store the value of the right side in the indicated location. Of course, if v is a constant function, i. e., the left side is not subscripted, then the two viewpoints coincide.

All of this assumes that the left side of our assignment does not itself have a side effect. This will always be true in FORTRAN II, but in most other algebraic languages the left side of an assignment may, for example, be a subscripted variable whose subscript(s) may have side effects. In such a case, we must first calculate the single side effect $u: \Gamma^D \to \Gamma^D$ associated with the left side $v: \Gamma^D \to \Gamma^M$, using the definition of the pair (v, u) as a star-extension with side effects. Once this is done, taking the second of the two viewpoints above, we arrive at the rule

$$a(S) = S', \; S'(v(u(S))) = t(e(u(S))),$$
$$S'(x) = S''(x) \text{ for } x \neq v(u(S)), \text{ where } S'' = e(u(S))$$

We repeat that no matter how many side effects the components of an assignment have, the resulting assignment corresponds to only one function from state vectors to state vectors, rather than two.

It is a well-known fact that many compiler optimization techniques are incompatible with the interpretation of multiple side effects on a strict left-to-right (or right-to-left) basis. Any such technique corresponds to a rule, alternative to that which we have given for the star-extension with side effects, for constructing the state vector function of an arbitrary expression; however, most such rules are difficult to work with in practice. Optimization of generated code for expressions is best carried out either in the complete absence of side effects or under guarantees (as may sometimes be given by the structure of certain languages) that side effects will involve only variables not present in the current expression.

Conditions can also have side effects. In fact, the action of a conditional go-to statement, in the sense of Definition 3-9-1, is precisely the side effect of the condition which it contains. Thus in the FORTRAN II statement

```
IF (A + F(X) + B) 21, 22, 27
```

the action is the side effect of F; if F has no side effect, there is no action, i. e., none of the variables in the program have their values changed by the statement. The next-statement function is not affected by the presence of side effects. Relations such as $A + B(X) > C(X) + D$, of course, have state vector functions and side effects which may be constructed in the general way described above, using the star-extension as given by Definition 3-11-1.

3-12 Parameters

Each reference to a function, with or without actual parameters, constitutes a primary expression and thus has a corresponding state vector function. For example, consider the function GCD(M, N), called by writing (say) GCD(K1+K2, K3*K4). The corresponding state vector function f is then given by f(S) = gcd(S(K1)+S(K2), S(K3)*S(K4)), for each state vector S; it is, in fact, the star-extension gcd*(e_1, e_2), where e_1 and e_2 are the state vector functions corresponding to K1+K2 and K3*K4 respectively. One way to implement this function reference is to perform the assignments M = K1+K2 and N = K3*K4, followed by calling GCD, in this case, we would have f(S) = gcd(S(M), S(N)). The advantage of this is that, when the GCD function is coded, the variables M and N have the same status as any other variable. For example, M is a primary expression, whose state vector function m, within the coding of GCD, is given by m(S) = S(M). The function f, in fact, does not depend at all, in this case, upon the actual parameters.

Let us now consider a function EUCLID(M, N, K) which sets K equal to GCD(M, N). If this function is called by writing EUCLID(K1+K2, K3*K4, K5), then it may not be implemented using the above scheme unless certain modifications are made. We may perform the assignments M = K1+K2 and N = K3*K4, as before, but we cannot then do K = K5 and call EUCLID; we must first call EUCLID, and then we must set K5 by doing K5 = K. If this is done, then the state vector function corresponding to EUCLID(M, N, K) will still depend only on the formal parameters M, N, and K; one must, however, make a distinction between parameters which are used in a routine (such as M and N) and those whose values are returned (such as K). This is done in the JOVIAL language, where, for example, EUCLID(M, N; K) would denote the fact that K is a "returned" parameter, since it follows the semicolon, whereas M and N are "used" parameters.

Formal parameters as described above, which are set to the values of the corresponding actual parameters (or the reverse) when the given routine is called, are said to be <u>called by value</u>. In some computer languages, such as JOVIAL and SNOBOL 4, all parameters are always called by value. There are, however, two other standard ways of calling parameters; we speak of parameters <u>called by reference</u> (sometimes "by location" or "by address") and <u>called by name</u>. In each of these cases, the execution of the given function or subroutine depends in a much more fundamental way upon the form of the actual parameters than if calling by value had been used. This implies that each usage of a formal parameter called by reference or by name within the coding of a routine is to be interpreted in a different way than if it were an ordinary variable.

Calling by reference is very common, being used, for example, in FORTRAN. The formal parameters are here implemented as <u>reference variables</u> (sometimes called "pointer variables"). In general, the value of a reference variable is another variable; if $\Gamma = (M, V, L)$ is a memory, then the legal values of a reference variable in M are themselves in M (besides being in V). If EUCLID(K1+K2, K3*K4, K5) is a call to the subroutine EUCLID(M, N, K) in which the parameters are called by reference, then M, N, and K become reference variables; the value of K becomes K5, whereas the values of M and N become temporary variables whose values are the values of the expressions K1+K2 and K3*K4 respectively (just before EUCLID is called). Such temporary variables are always used unless the corresponding actual parameter consists of a single variable or a single constant. The state vector function m corresponding to the primary expression M in the coding of the subroutine is now given by m(S) = S(S(M)), rather than m(S) = S(M). Here S(M) gives the current value of M, which is another variable, and then S(S(M)) gives the current value of that variable.

The use of call by reference is less efficient in simple cases than the use of call by value, because of the extra calculation necessary whenever a parameter is used. (Some schemes for implementing call by reference do not use any extra time at this point, but they use it elsewhere.) On the other hand, call by reference is much more natural than call by value; in particular, a parameter called by value which is both "used" and "returned" must be mentioned twice. It is also to be noted that call by value in ALGOL allows only "used" parameters, whereas call by reference always allows parameters to be either "used" or "returned." If M, as above, appears on the left side of an assignment statement, the corresponding state vector function is given by m(S) = S(M); its values are themselves variables, as for the state vector function of the left side of any assignment.

Call by name is defined relative to a transformation on the program which results in a new program without any calls at all. Each call to a procedure (= subroutine) is replaced by the entire text of that procedure, in which each formal parameter called by name is replaced by the corresponding actual parameter. When the actual parameter consists of a single constant or a single unsubscripted variable without side effects, call by name is equivalent to call by reference. When an actual parameter consists of a more general expression, however, this expression must be evaluated anew each time the parameter is used. If the values of any of the variables appearing in the expression have changed, the value of the expression will normally change as well.

It is possible to implement call by name according to the original definition; that is, the given program is transformed in such a way as to eliminate procedure calls. The new form of the program, however, would normally take up excessive space, and so what is usually done instead is to treat each usage of a parameter called by name as a function which evaluates the corresponding actual parameter. A reference to this value is then returned by the function; this reference is a temporary variable unless the parameter is a single variable, just as in call by reference. Since a procedure may be called from more than one place, with differing actual parameters, such a parameter usage is a <u>function variable</u>, whose values are state vector functions. For the formal parameter M, called by name, the state vector function of M as a primary expression is given by $m(S) = S(h(S))$ where $h = S(M)$, and its state vector function as the left side of an assignment is given by $m(S) = h(S)$. Any side effect of h becomes the side effect of m whenever $S(M) = h$; the total side effect of m is then the function s whose (state vector) value $s(S)$ is the value of the side effect of $S(M)$.

A mathematical proof of the equivalence of this scheme for handling parameters called by name with that of the original definition is beyond the scope of this book. Such a proof must, of course, be a part of any complete proof of correctness of an ALGOL system. Further examples of the equivalence of parameter handling schemes are quite common in working with programming languages. Most algebraic language processors, for example, use <u>calling sequences</u>, which are sequences of formal parameter values as described above. There is then within each subroutine a single variable whose values are (pointers to) calling sequences. Such a subroutine is called by setting the value of this single variable to a particular calling sequence and transferring control, which is faster than setting several parameter variables and transferring control. In the EUCLID routine, the calling sequences are of the form (μ, ν, κ), where μ, ν and κ are either integers (call by value), integer references (call by reference), or integer-reference-valued functions (call by name). If R is the calling sequence variable, then the state vector function of M, for example, as a primary expression is now given by $m(S) = \mu$ (or $S(\mu)$, or $\mu(S)$) where $(\mu, \nu, \kappa) = S(R)$. This illustrates only the most rudimentary calling sequence scheme; if a language processing system employs such a scheme, then the proof of correctness of the system must include a proof that the scheme is valid -- normally, a proof of the equivalence of the scheme with that by which the given language is defined.

3-13 Correctness

The theory presented in Chapters 1 and 2 on correctness of programs will now be given a formal mathematical basis. Let P be a program on the memory Γ_0, as in Definition 3-9-1, and let $\Gamma = \Gamma_0 \cup (\{\lambda\}, P)$, so that Γ^D may be canonically identified with $\Gamma_0^D \times P$. By Theorem 3-10-2, we may assume that P is complete, and if it is complete then its execution function $f: \Gamma^D \rightarrow \Gamma^D$ is given by Definition 3-9-2. If $T_0 \in \Gamma^D$, then the computation sequence T_0, T_1, T_2, ..., starting from T_0 is given by $T_{i+1} = f(T_i)$ for $i \geq 0$ as in Definition 3-4-1. An assertion about the variables of P is the same as a condition on them; that is, it is a state vector function whose values are true or false. We shall view every assertion about P as a subset of Γ_0^D (or sometimes of Γ^D, if this is explicitly stated); this is justified in section 3-5. A state vector S for which the assertion A is valid is precisely one which belongs to A as a set. We may now make formal definitions of correctness and partial correctness.

DEFINITION 3-13-1. Let Π be a set of pairs (F_i, A_i), where each F_i is a statement of a complete program P on Γ_0 and each $A_i \in \Gamma_0^D$. Let Γ, f, and computation sequences be defined as above. Let $R \in \Gamma^D$ be the set of all (S_j, F_i), for all $S_j \in \Gamma_0^D$ and all $(F_i, A_i) \in \Pi$, and let $R_0 \subseteq R$ be defined by $(S_j, F_i) \in R_0$ if and only if $S_j \in A_i$. Then P is partially correct with respect to Π if no computation sequence T_0, T_1, ..., for $T_0 \in R_0$ contains elements of $R-R_0$. It is correct with respect to Π if it is partially correct with respect to Π and if, in addition, each computation sequence T_0, T_1, ..., for $T_0 \in R_0$ contains elements of R_0.

THEOREM 3-13-1. If P is partially correct with respect to Π, then it is partially correct with respect to any subset $\Pi' \subseteq \Pi$.

PROOF. Let R and R_0 relative to Π' be denoted by R' and R_0'. Then $R' \subseteq R$, $R_0' \subseteq R_0$, and $R'-R_0' \subseteq R-R_0$. If T_0, T_1, ..., is a computation sequence for $T_0 \in R_0'$, then certainly $T_0 \in R_0$, and the sequence contains no elements of $R-R_0$, and therefore no elements of $R'-R_0'$. This completes the proof.

This proof is deceptively short, because we have defined partial correctness relative to entire computation sequences, rather than control paths. The following theorem indicates, as we did intuitively earlier, how partial correctness is proved.

THEOREM 3-13-2. Using notation as in the definition above, suppose that whenever T_0, T_1, ..., is a computation sequence with $T_0 \in R_0$, and $T_k \in R$ but each $T_j \notin R$ for $0 < j < k$, we have $T_k \in R_0$. Then P is partially correct with respect to Π.

PROOF. Suppose the contrary; then there exists a computation sequence T_0, T_1, ..., with $T_0 \in R_0$, containing elements of $R-R_0$. Let T_z be that element of $R-R_0$ in this sequence whose index z is as small as possible, so that $T_j \notin R-R_0$ for $0 < j < z$. Let T_y, $y < z$, be that element of R_0 in this sequence whose index is as large as possible; certainly T_y exists, since $T_0 \in R_0$. We then have $T_j \notin (R-R_0) \cup R_0 = R$ for $y < j < z$. Consideration of the computation sequence beginning with T_y leads us to a contradiction of the hypothesis. This completes the proof. (The converse is obvious.)

COROLLARY 3-13-1. If for each statement F_i in P we have $(F_i, A_i) \in \Pi$ for some A_i, then P is partially correct with respect to Π if and only if $f(R_0) \subseteq R_0$ (that is, $T \in R_0$ implies $f(T) \in R_0$ if $f(T)$ exists).

This follows immediately from the fact that $R = \Gamma^D$ in this case.

DEFINITION 3-13-2. Using notation as in the definition above, a path in P is a sequence $(\Sigma_0, \ldots, \Sigma_n)$ of statements of P such that for each i, $1 \leq i \leq n$, there exist S, S' $\in \Gamma_0^D$ with $f((S, \Sigma_{i-1})) = (S', \Sigma_i)$. A loop in P is a path $(\Sigma_0, \ldots, \Sigma_n)$ in P with $\Sigma_0 = \Sigma_n$. A control path of Π is a path of P such that $(\Sigma_i, A_i) \in \Pi$ for some A_i if and only if i = 0 or n. A loop control path of Π is a control path of Π contained in some loop in P. A control path of Π is verified if for each computation sequence $(S_0, \Sigma_0), \ldots, (S_n, \Sigma_n)$ of P (for the path $(\Sigma_0, \ldots, \Sigma_n)$) we have $S_0 \in A_0$ implies $S_n \in A_n$, for $(\Sigma_0, A_0), (\Sigma_n, A_n) \in \Pi$. The set Π is sufficient if each loop in P contains some Σ_i with $(\Sigma_i, A_i) \in \Pi$ for some A_i.

THEOREM 3-13-3. If every control path of Π is verified, then P is partially correct with respect to Π.

PROOF. Let $T_0 = (S_0, \Sigma_0)$, $T_1 = (S_1, \Sigma_1)$, ..., be a computation sequence for some $\Sigma_0, \Sigma_1, \ldots$; then for each k, $(\Sigma_0, \ldots, \Sigma_k)$ is a path in P, by definition of a computation sequence. If $T_0 \in R_0$, $T_k \in R$, and each $T_j \notin R$ for $0 < j < k$, then $(\Sigma_0, \ldots, \Sigma_k)$ is a control path of Π by the definitions of R and R_0. By hypothesis, it is therefore verified; since $T_0 \in R_0$, we have $S_0 \in A_0$, and thus $S_k \in A_k$. This, combined with $T_k \in R$, yields $T_k \in R_0$, and thus, by Theorem 3-13-2, P is partially correct with respect to Π.

THEOREM 3-13-4. If Π is sufficient, then Π has only a finite number of control paths, and these are of bounded length.

PROOF. Let $(\Sigma_0, \ldots, \Sigma_k)$ be such a path. We first show that $\Sigma_1, \ldots, \Sigma_{k-1}$ are all distinct. Suppose the contrary, so that $\Sigma_i = \Sigma_j$, where we may assume $i < j$. Then by considering $(\Sigma_i, \Sigma_{i+1}, \ldots, \Sigma_j)$, we would have a contradiction to the statement that Π is sufficient. If there are n statements in the program, then there are at most n! choices for $(\Sigma_1, \ldots, \Sigma_{k-1})$ if these are all distinct, and at most n choices for each of Σ_0 and Σ_k. Also, the maximum length of each such path is clearly n+1. This completes the proof.

THEOREM 3-13-5. Suppose that Π is sufficient and that each control path of Π is verified, and let J be the set of all integers. Suppose that for each loop control path $C = (\Sigma_0, \ldots, \Sigma_k)$ we may associate $b_C: \Gamma^D \to J$, $e_1: \Gamma^D \to J$, ..., $e_n: \Gamma^D \to J$, such that:

(1) For each computation sequence (S_0, Σ_0), ..., (S_k, Σ_k) with $S_0 \in A_0$ (for $(\Sigma_0, A_0) \in \Pi$) we have

(a) $e_i(S_k) \geq e_i(S_0)$, $1 \leq i \leq n$,

(b) $e_n(S_k) > e_n(S_0)$ if C is primary (see (3) below),

(c) $b_K(S_k) = b_K(S_0)$ for each b_K associated with each loop control path K of Π (including K = C);

(2) If e_1, ..., e_n is associated with $C = (\Sigma_0, \ldots, \Sigma_k)$ and e'_1, ..., e'_m with $C' = (\Sigma'_0, \ldots, \Sigma'_k)$, then $\Sigma_k = \Sigma'_0$ implies $e_i = e'_i$ for $1 \leq i \leq \min(m, n)$ (in which case C is called <u>inner</u> to C' if $m \leq n$, <u>outer</u> to C' if $m \geq n$, and <u>strictly outer</u> to C' if $m > n$);

(3) Each $C = (\Sigma_0, \ldots, \Sigma_k)$ associated with e_1, ..., e_n is either

(a) <u>primary</u> (see also (1) above), in which case $S \in A_k$ (for $(\Sigma_k, A_k) \in \Pi$) implies $e_n(S) \leq b_C(S)$;

(b) <u>secondary</u>, in which case each loop in P containing C contains some C' which is primary and outer to C, or which is strictly outer to C.

Then P is correct with respect to each subset Π' of Π involving all termination statements of P (i. e., $(\Sigma, A) \in \Pi - \Pi'$ implies $f((S, \Sigma))$ is always defined for $S \in A$).

<u>PROOF</u>. Since each control path of Π is verified, P is partially correct with respect to Π, and therefore with respect to Π'. Let (S_0, Σ_0), (S_1, Σ_1), ..., with $(\Sigma_0, A_0) \in \Pi'$, $S_0 \in A_0$, be a computation sequence; it is sufficient to derive a contradiction, assuming that the sequence is infinite. (If it must be finite, then it ends at some (S_k, Σ_k), with $(\Sigma_k, A_k) \in \Pi'$ by the hypothesis on Π'; since P is partially correct with respect to Π', we have $S_k \in A_k$,

and therefore P is correct with respect to Π'.)

Let i_0 (= 0) $< i_1 < i_2 < \dots$ be those indices for which $(\Sigma_{i_j}, A_{i_j}) \in \Pi$; the control paths which this sequence traverses are $C_j = (\Sigma_{i_{j-1}}, \Sigma_{i_{j-1}+1}, \dots, \Sigma_{i_j})$ for each $j > 0$. Since Π is sufficient, there are infinitely many i_j and hence infinitely many C_j. Since there are only a finite number of distinct C_j, some must appear an infinite number of times. Of these, let C_x be associated with $e_1, \dots, e_n$ for n as small as possible. If C_x is primary, take $C_y = C_x$; otherwise, any loop in P containing C_x contains some C' which must be primary and associated with $e_1, \dots, e_n$ by the choice of x; we choose such a C' which appears an infinite number of times and call it C_y. There are a finite number of C_j each of which appears a finite number of times, ending at Σ_{i_j} for various i_j; let C_u end at the largest of these i_j, and let j_0 (= $v > u$) $< j_1 < j_2 < \dots$ be those indices for which $C_{j_i} = C_y$. Such indices exist because C_y appears infinitely often. Let $v_k = e_n(S_k)$, $b_k = b_{C_k}(S_k)$ for each $k \geq 0$; by hypothesis (3b), $v_{j_i} \leq b_{j_i}$ for each i, since C_{j_i} is primary.

We now show that each C_z, $z > v$, is inner to C_v. Suppose the contrary, and let z be the smallest integer with $z > v$ but C_z not inner to C_v. Then $z-1 \geq v$ and C_{z-1} is inner to C_v. By hypothesis (2), either C_{z-1} is outer to C_z (impossible, because then C_z would also be inner to C_w) or C_{z-1} is inner to C_z. But then C_z is either inner or outer to C_w; since it is not inner, it must be strictly outer, and we have a contradiction to our choice of C_x. Hence each C_z, $z > v$, is inner to C_v, and this implies immediately by induction $v_a \leq v_b$ and and $b_a = b_b$ for all $b > a > v$ by hypotheses (1a), (1c). By hypothesis (1b), $v_{j_i - 1} < v_{j_i}$ for each i, and since $v_{j_i - 1} \leq v_{j_{i-1}}$, we have $v_{j_0} < v_{j_1} < v_{j_2} < \dots$ and each $v_{j_i} \leq b_{j_i} = b_{j_0}$, which is impossible. This completes the proof.

PROBLEMS

1. (a) Let A be a set of $\underline{m}$ elements and let B be a set of $\underline{n}$ elements. How many total functions $f\colon A \rightarrow B$ are there?

(b) How many of these are one-to-one?

(c) How many partial functions $f\colon A \rightarrow B$ are there?

(d) How many of these are one-to-one?

2. A relation is called a <u>strict ordering</u> if it is irreflexive, antisymmetric, and transitive. Show that there is a natural one-to-one correspondence between the set of all strict orderings on a given set and the set of all partial orderings on that set. (Note: If a partial ordering is denoted by $\leq$ or $\geq$, the corresponding strict ordering is often denoted by $<$ or $>$.)