KERNEL EQUIVALENCE OF PROGRAMS AND PROVING KERNEL EQUIVALENCE AND CORRECTNESS BY TEST CASES*

Terrence W. Pratt
University of Texas
Austin, Texas, U.S.A.

Abstract

The statements in a program may be classified as "kernel statements" if they participate directly in the computation of some output and as "control statements" if they participate directly in deciding the control path at branch points. Two programs are kernel equivalent if they always execute identical sequences of kernel statements given the same inputs. Kernel equivalence is defined formally and is shown to be practically decidable in many cases by a procedure of trying test cases. The concept of program kernel may also be used as a basis for proving correctness of programs.

Descriptive Terms

Equivalence, correctness, program analysis, flow of control, flow of data, theory of programming, graph model

I. Introduction.

The most general definition of the equivalence of programs is the purely functional -- a program represents a mapping from input data sets to output data sets and two programs are equivalent if they represent the same function on the same domain of inputs. The other extreme of equivalence is the purely structural. One assumes a particular representation of programs, e.g. as Algol programs, Turing machines, abstract flowchart graphs, or lambda-calculus expressions. Two programs arc then equivalent if they have identical representations, i.e. identical structures. Unfortunately, for any fairly general representation of programs, structural identity is too restricted a form of equivalence. On the other hand, determining functional equivalence is usually undecidable in general, and at best requires a detailed knowledge of the properties of the primitive operations and predicates involved.

In this paper, we develop an intermediate definition of equivalence based on program "kernels" which requires only fairly superficial knowledge of the flow of control and data between statements in a program yet which may be used to prove functional equivalence of programs whose structures are quite different. Proving the kernel equivalence of two programs may often be done by observing the results of executing each program on a finite number of test cases.

*This work supported in part by NSF Grant GJ-778

The definition of "kernel equivalence" is developed formally within the framework of the common directed graph representation of programs presented in the next section, (although the use of this particular representation is not essential). Following sections develop the definition of kernel equivalence in terms of a distinction between kernel and control statements and between kernel and control variables. Some examples of equivalence proofs are given. In the concluding section we also argue the relationship of kernel equivalence to proofs of program "correctness".

11. Programs.

As the basic formal model of a program we shall adopt a representation which has been widely used (see e.g. Luckham, Park and Paterson (1), Kaplan (2)). A program is represented by a finite directed graph (flowchart) with labeled nodes. Nodes may have zero (exit nodes), one (assignment nodes) or two (branch nodes) exiting arrows. There is a unique entry node with no entering arrow and every node lies on a path from the entry node to an exit node. The arrows leaving a branch node are labeled <u>true</u> and <u>false</u>. Assignment nodes are labeled with assignment statements, branch nodes with branch statements. The entry node is labeled "ENTER" and each exit node is labeled "EXIT".

Assignment and branch statements are constructed from a set of variables $\mathcal{V}=\{V_1,V_2,\ldots\}$, a set of single-valued functions $\mathcal{F}=\{F_1^{n_1},F_2^{n_2},\ldots\}$ (where n_i specifies the number of arguments of F_1) and a set of predicates $\mathcal{P}=\{P_1^{m_1},P_2^{m_2},\ldots\}$ (where each m_i specifies the number of arguments of P_i. A branch statement has the form:

$$P_i^{m_i}(U_1,U_2,\ldots U_{m_i}) \text{ where each } U_k\in\mathcal{V},\ P_i^{m_i}\in\mathcal{P}$$

and an assignment statement has the form:

$$U_o:=F_k^{n_k}(U_1,U_2,\ldots U_{n_k}) \text{ where each } U_i\in\mathcal{V}$$
$$\text{and } F_k^{n_k}\in\mathcal{F}.$$

Figure 1 is an example of a program. Execution of a program proceeds according to the usual rules. One begins at the entry node and follows a path through the graph executing assignment statements as they are encountered and choosing the appropriate branch at branch nodes on the basis of the value of the predicate.

The particular representation as a directed graph is not important. Alternatively, we could have used, e.g., an Algol-like program representation. The key items of information needed

are:
(1) The possible sequences of statements which may be executed by the program and (2) the flow of data information given by inspecting the variables occurring in the possible sequence of statements.

Some auxiliary definitions are needed.

Let R be the range of possible values for the variables of $\mathcal{V}$.

Definition (1.1) If H is a program, then the variable set of H, $\mathcal{V}_H$, is

$\mathcal{V}_H = \{V \mid V \in \mathcal{V}$ and V occurs as an argument in some branch or assignment statement or on the left of ":=" in an assignment statement in H. $\}$

(1.2) A value assignment for a program H is a mapping of $\mathcal{V}_H$ into R, which assigns a value from R to each variable of the program. We shall refer to the range of such a function as a value set.

(1.3) The initial value assignments of H are the value assignments which represent valid initial assignments of values to the variables of 11.

(1.4) The execution sequence $\mathcal{S}_I$ of H for a given initial value assignment I is the sequence $S_1S_2 \ldots S_n$ of statements executed when H is executed with its variables initialized to the values specified by the initial value assignment. If H does not terminate for 1 then $\mathcal{S}_I$ may be infinite.

(1.5) The result assignment Q. oi H for a given initial value assignment 1 is the value assignment resulting from execution of the program with variables initialized to the values in I. Q is undefined if H does not terminate for I.

We assume that the functions and predicates are properly defined so as to always give a value, so that Q is always defined if H does terminate for L

III. Approaches to Program Equivalence.

The foregoing definitions give us two views of a program H as a function: (1) As a function from initial value assignments to result assignments, H (I)=Q (if 11 terminates on I) and (2) as a function from initial value assignments to execution sequences 11 (])- $\mathcal{S}_I$. Both views in fact may be used as a basis for definitions of program equivalence, as follows:

(1) Functional equivalence. If H and H^1 are programs then H = H' (H is functionally equivalent to H') iff II and H' have the same variable sets and the same initial value assignments, and for every initial value assignment I, either both $H_f(I)$ and $H'_f(I)$ are undefined (do not terminate) or if $H_f(I)$ = Q and H' (I)=Q^I , then Q =Q' (similar to the "strong equivalence" of Luckham, Park, and Paterson (1)).

(2) Structural equivalence. If H and H^1 are programs then H = H' (H is structurally equivalent to H') iff H and H' have the same variable sets and initial value assignments, and for every initial value assignment 1, II (1) - II' (1), (i.e. each program executes exactly the same sequence of statements given input values I), and at each branch node the branch statement returns the same value in each program (similar to the "strict equivalence" of Orgass (3)). Structural equivalence does not in fact imply that the graphs for H and H' are isomorphic. However, it is clearly a very narrow view of equivalence, requiring that each program execute the same sequence of statements and branches for each input assignment. Functional equivalence, on the other hand, is perhaps the broadest definition of equivalence, requiring only that each program produce the same output (or be undefined) for each input assignment. The following result follows immediately:

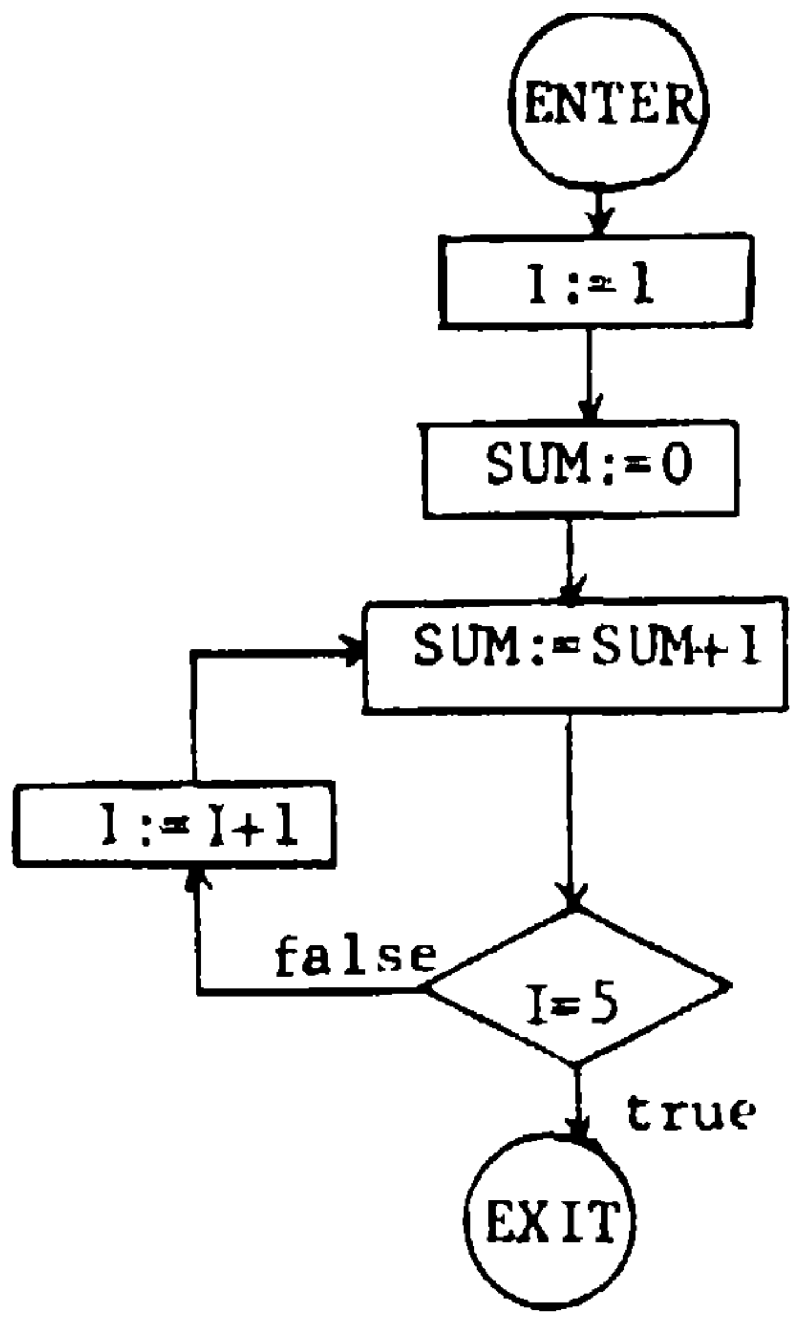
Theorem 1. If H = H' then H - II'.

It is fairly clear that any general scheme for proving functional equivalence immediately runs into basic undecidability results (see Luckham, Park, and Paterson (1)) due to the fact that programs may not terminate for some inputs. For the study of practical programs however, it is of interest to press past this roadblock and ask: Suppose wc assume termination, can we prove equivalence? This seems not unreasonable since for actual programs we can often use specialized arguments to show that particular programs terminate for the initial value assignments of interest. Thus we now assume that the initial value assignments for each program 11 are restricted to those ior which 11 terminates.

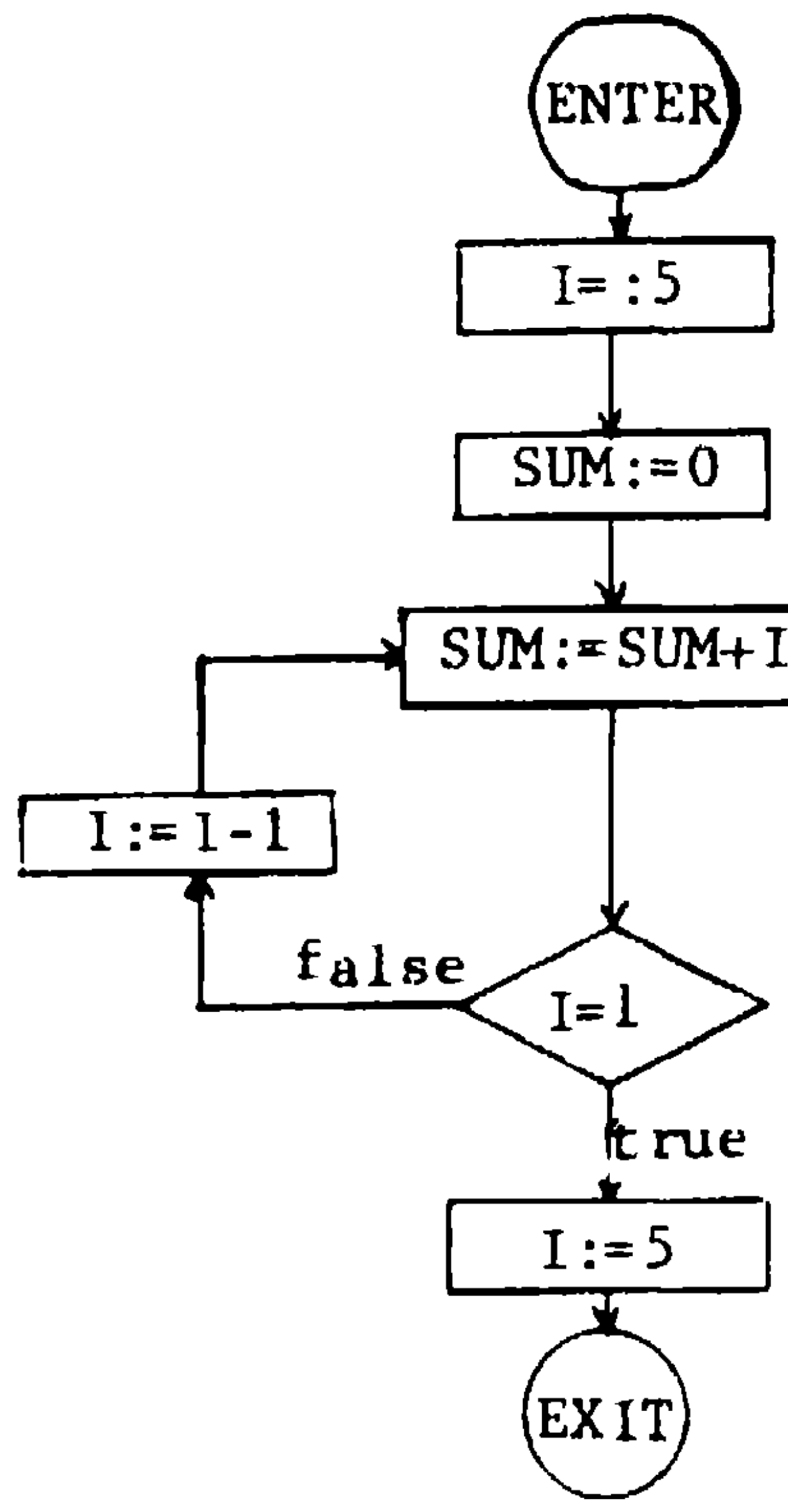
Is functional equivalence decidablc given termination? If the set R of possible initial values for variables is infinite, then it is not clear. Suppose that R is finite? Then immediately the brute force algorithm will suffice theoretically: execute each program on each possible initial value assignment; the programs are equivalent iff they have the same result in each case. But is such a brute force approach ever practical? (Clearly actual programs always have only a finite range oi values possible for each variable.) Unfortunately very seldom, ior although theoretically the set oi input value sets may be finite, it is ordinarily so large as to preclude exhaustive testing. We are unlikely to find testing equivalence by looking at the outputs of the programs for given inputs a practical possibility even when theoretically feasible. We may fall back on the standard debugging tool - try a small set of test cases -

to gain some partial evidence for equivalence, but no proof.

What other possibilities besides this "black box" approach of matching inputs and outputs is feasible? Clearly, some analysis of program internal structure is indicated. At the extreme, isomorphism of programs is a clearly decidable but uninterestingly narrow definition of equivalence. On the other end of the scale, we might go into the properties of the functions and predicates in each program, requiring a set of axioms which allow proving certain functions or sequences of statements to be equivalent. Figure 1 is such a case in point. The program in 1(a) computes SUM:=((((0+l)+2)+3)+4)+5 while that of 1(b) computes SUM:=((((Of5)+4)+3)+2)+l. We can only prove they are equivalent if we know that "+" is commutative and associative.



1 (a)



1 (b)

Figure 1.

Two Functionally Equivalent Programs.

Proving equivalence in cases where detailed knowledge is needed of the functions and predicates requires a substantial theorem-proving apparatus.

Are there any reasonable definitions of program equivalence which are fairly broad yet require only the sort of information found in the abstract programs of the preceding section, namely information about flow of control and data, but no detailed information about the functions and predicates involved? In the next section, we propose one such definition and then argue that not only is it decidable theoretically but in many cases practically.

IV. Program Kernels and Kernel Equivalence

The proposed approach to equivalence may be briefly outlined:

(1) Assume that only certain variables contain significant "output values" on termination. Call these the kernel output variables.

(2) By backtracing control paths through the program from exit nodes, identify and tag (as kernel statements) all program statements which participate in the computation of the iinal values oi the kernel output variables. At the same time identify the kernel input variables, those variables which are used as arguments in some kernel statement but have no prior assignment made to them.

(3) By a similar backtracing from branch nodes, identify and tag (as control statements) all branch statements and all statements which participate in the computation of the arguments to branch statements. At the same time identify control input variables, those variables which are used as arguments in some control statement but have no prior assignment made to them.

It should be clear that given complete information about flow of control and data in a program, the sets of kernel statements, kernel input variables, kernel output variables, control statements and control input variables may be readily determined (algorithms are given below for the model of the preceding section.) Note that neither the sets of statements nor the sets of variables need be disjoint.

Two programs are said to be kernel equivalent iff (a) they have the same kernel input, kernel output and control input variables sets and the same set of kernel statements, and (b) for each initial value assignment 1 to kernel and control variables they execute the same sequence of kernel statements (the kernel execution sequence for 1).

We shall show that kernel equivalence implies functional equivalence relative to the kernel output variables. Moreover, the kernel execution sequence for a given initial value assignment is dependent only on the values assigned to the control variables. Thus to determine kernel equivalence of two programs we need only try as test cases each possible assignment of initial values to the control variables and compare the kernel execution sequences in each case. Finally we shall argue that this technique has a certain practical interest, for ordinarily control input variables in actual programs have only a small range of values, and there are relatively few control input variables at that. It is the kernel input variables which will contain the real data which the program manipulates, and thus which will have a large range of possible values. We may expect often to find only a small number of possible initial assignments to control input variables, thus indicating only a small number of test cases to be tried to determine kernel equivalence which then in turn implies functional equivalence for the variables of interest. Kernel equivalence also has some implications for proving correctness of programs (see the conclusion).

We will now show the rigorous development of these concepts in the model of the preceding section:

Kernel Output Variables. We may assume that the set of kernel output variables (the variables whose linal values are of interest) is given. Alternatively, we may identify such a set by tracing iorward from nodes containing assignment statements. It the assignment statement has the form $U:=F(V_1,\ldots,V_n)$ and there is a path from the node to an exit node which contains no statement in which U is an argument, then U is a kernel out put variable (since U is assigned a value which in some cases is never later used).

Kerne1 Statements and Kernel Input Variables. First tag as a kernel statement each statement in the step above which computes a kernel output. Now begin to iterate: Pick an argument variable V in some kernel statement U:=F(...,V,...). Trace back along each path trom the node containing that kernel statement until an assignment node containing the assignment V := . . . is reached. 'Jag that statement as a kernel statement. If the entry node is reached and no such assignment has been found, then V is a kernel input variable. Repeat the process until all argument variables in all kernel statements have been tested.

Control Statements and Control Input Variables. First tag as control statements all branch statements contained in branch nodes. Now iterate exactly as for kernel statements: Pick an argument variable in a control statement and trace back along each path to a statement which assigns a value to that variable and tag that statement as a control statement. If the entry node is reached then the variable is a control input variable. Clearly all these procedures terminate in a finite number of steps.

Ke mel Execution Sequence. Given an initial value assignment I for the variables of a program H, the kernel execution sequence K_I of H for 1 is the ordinary execution sequence $\&_I$ with all but kernel statements deleted.

Kernel Equivalence. Two programs H and H' arc kernel equivalent, $H \overset{k}{=} H$, iff

(1) H and II^1 have identical sets of kernel statements, kernel input variables, kernel output variables and control input variables, and
(2) for every initial value assignment 1, H and H^1 have the same kernel execution sequence.

Kernel Equivalence Implies Functional Equivalence, The following theorem follows easily:

Theorem 2. If H and H' are programs and $H \overset{k}{=} H^1$, then $H \underset{f}{=} H'$ considered as a function only from the control and kernel input variables to the kernel output variables.

Proof Outline: Since the control statements are the only statements which participate in determining the execution sequence followed for a particular initial value assignment, the initial values for the control input variables uniquely determine a kernel execution sequence. H = H' implies, for a given initial value assignment 1, that $K_1 = S_1S_2, \ldots S_n = K'_I$ But these kernel statements $S_l \ldots S_n$ arc the only statements which participate in determining the values of the kernel outputs, and the initial values of the kernel input variables completely determine the results of executing the statement sequence $S_1 \ldots S_n$. Thus for the same initial values of kernel and control input variables, $H \overset{k}{-} H'$ implies that execution of H and H^1 will produce the same result values in the kernel output variables. H = H' also implies that H terminates on a given initial value set I iff 11' terminates on 1 also. //

Now the following result justifies the use of a set of test cases for deciding kernel equivalence, Theorem 3. If H and H^1 are programs with identical sets of kernel statements and control input, kernel input and kernel output variables, then H - H* iff II and H^1 have identical kernel execution sequences for each initial value assignment to the control input variables.

Proof Outline: The control statements are the only statements which affect the path taken through the program graph during execution. A given initial value assignment to the control input variables determines a unique execution sequence and thus a unique kernel execution sequence. Therefore, if I_o and I_1 are two distinct initial value assignments with identical assignments to the control variables then $K_{I_o} = K_{I_1}$ regardless of the values assigned the other variables.//

by the preceding theorem if we restrict the possible initial values oi the control input variables to be finite sets, then kernel equivalence is decidable by simply trying each possible combination of initial value assignments and looking at the kernel execution sequences which result; The set of possible value assignments to kernel input variables may be Infinite or very large without increasing the number of test cases to be tried.

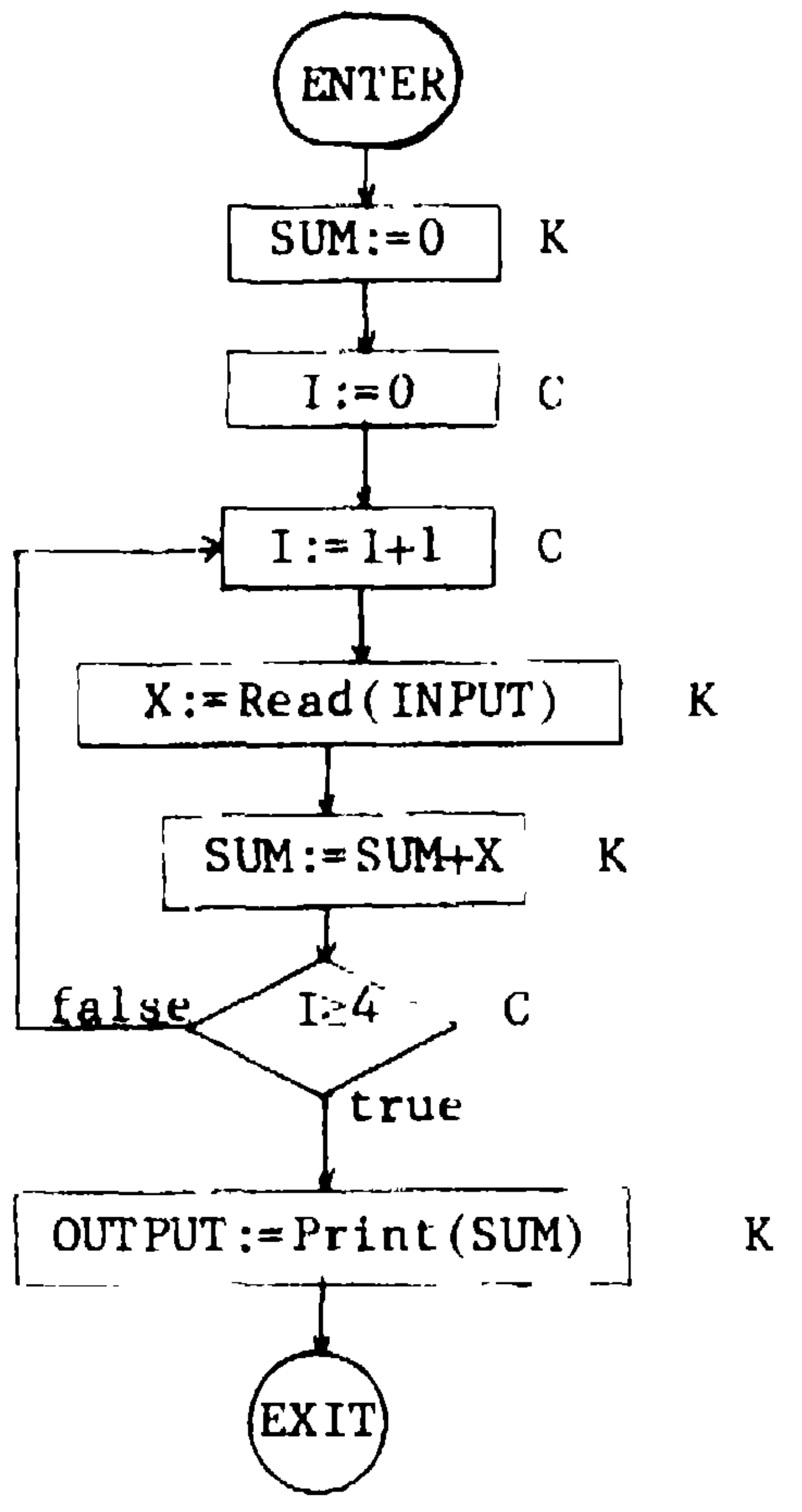
Example 1.

Consider the three programs of Figure 2. Each has a single kernel output variable, OUTPUT, a single kernel input variable, INPUT, no control input variables, and kernel statements:

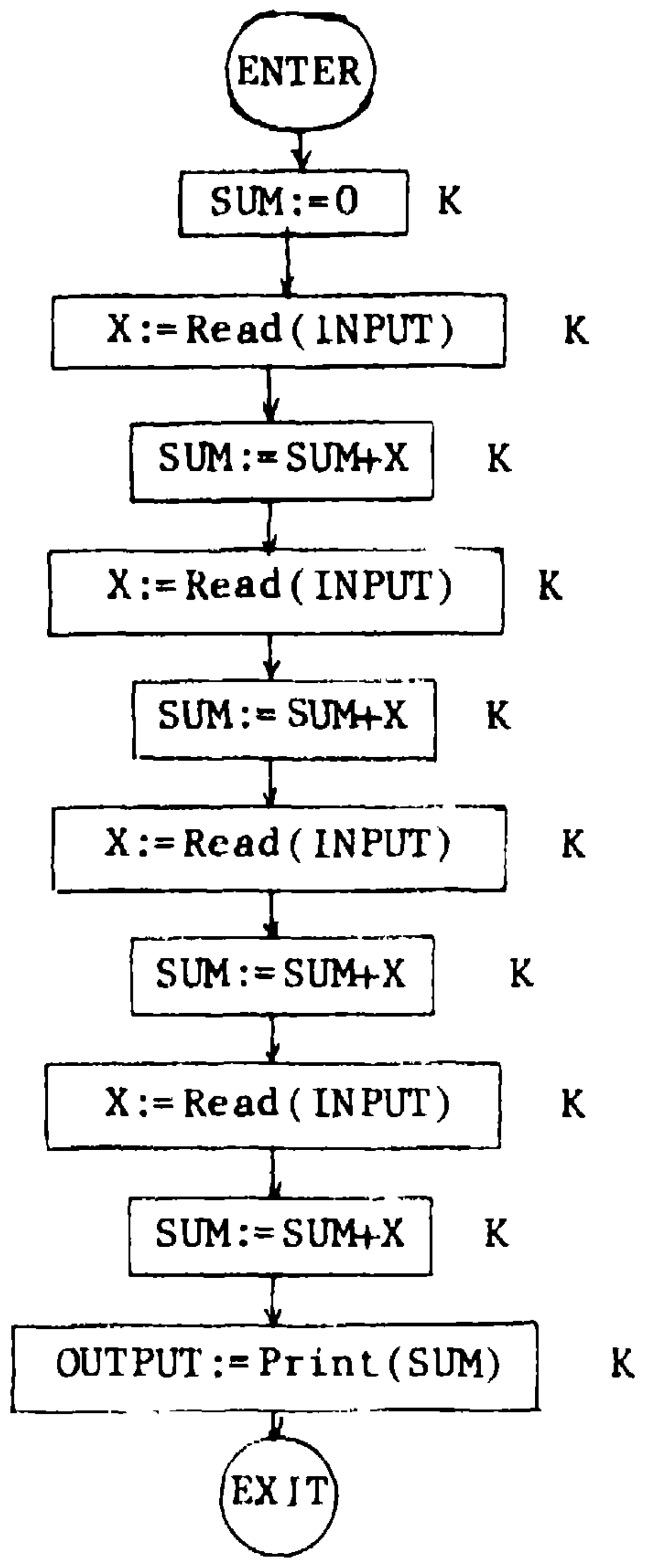
```
     X := Read(INPUT)
OUTPUT := Print(SUM)
   SUM := 0
   SUM := SUM+X
```

In Figure 2, nodes containing kernel statements are tagged with a "K" and those containing control statements with a "C".

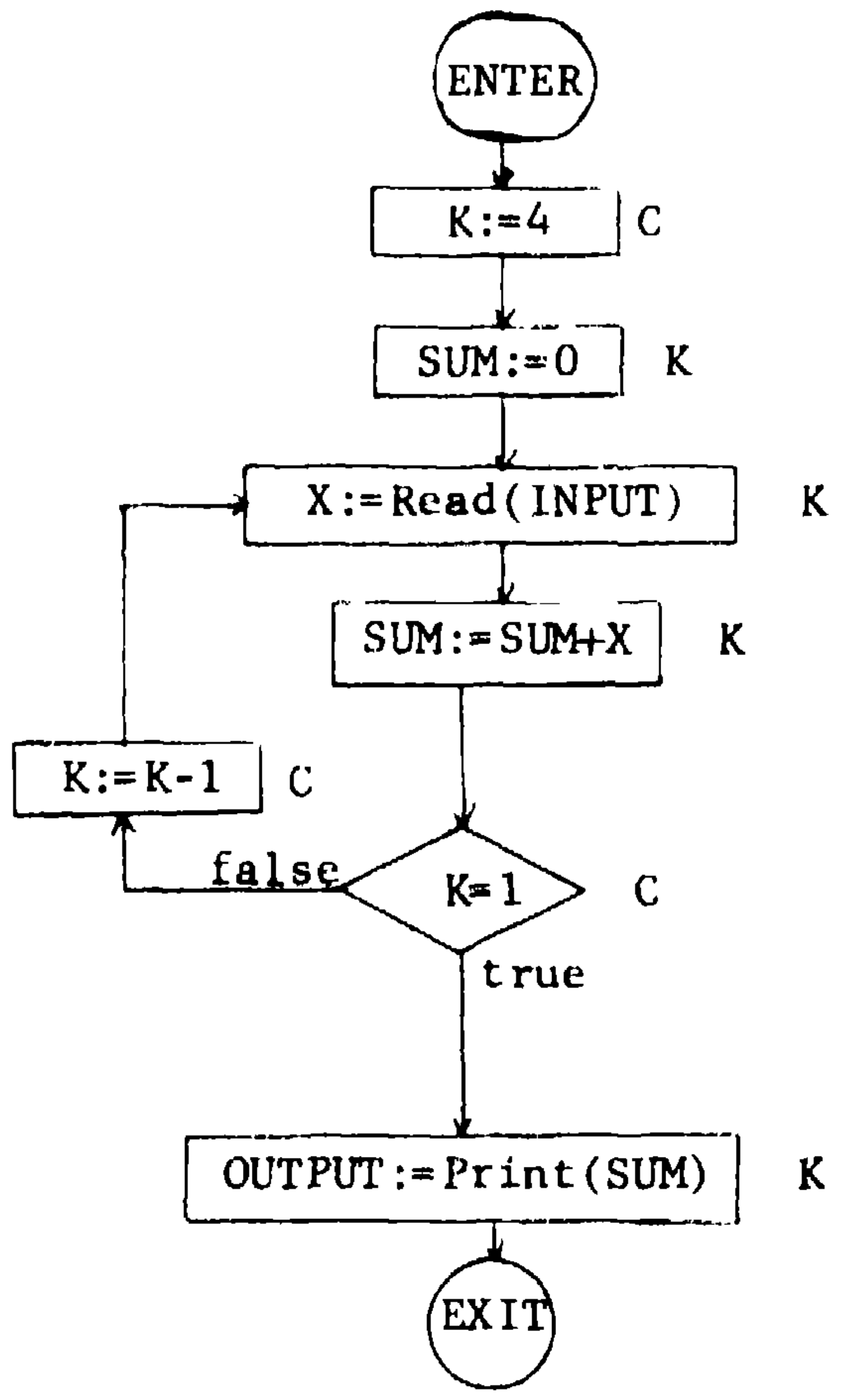
Are the programs kernel equivalent? By Theorem 3, we need only look at the kernel execution sequence for each possible initial assignment of values to the control input variables. Since that set is empty, there is only one execution sequence, and thus only one kernel execution sequence for each program. Figure 3 shows the execution sequence for each program, with non-kernel statements in parentheses. Since the kernel execution sequences for each program are identical, the programs are kernel equivalent. Thus by Theorem 2 they are also functionally equivalent with respect to the input variable INPUT and the output variable OUTPUT. The important point of this proof is that we have used only the simplest arguments to prove functional equivalence of three structurally quite dissimilar programs.



2 (a)



2 (b)



2 (c)

Figure 2.

Three Kernel Equivalent Programs.

```
   SUM := 0
    (I := 0)
    (I := I + 1)
     X := Read(INPUT
   SUM := SUM + X
    (I ≥ 4?)
    (1 := 1 + 1)
     X := Read(INPUT)
   SUM := SUM + X
    (1    4?)
    (I := 1 + 1)
     X := Read(INPUT)
   SUM := SUM+ X
    (I 4. 4?)
    (1 := I + 1)
     X := Read(INPUT)
   SUM := SUM + X
    (I 4  4?)
OUTPUT •  Print(SUM)
```

3(a)

```
   SUM = 0
     X    Read(INPUT)
   SUM    SUM + X
     X    Read(INPUT)
   SUM    SUM + X
     X    Read (INPUT)
   SUM    SUM + X
     X    Read(INPUT)
   SUM    SUM + X
OUTPUT    Print(SUM)
```

3(b)

```
    (K := 4)
   SUM := 0
     X := Read(INPUT)
   SUM := SUM + X
    (K  = 1?)
    (K := K - 1)
     X := Read(INPUT)
   SUM := SUM + X
    (K  = 1?)
    (K := K - 1)
     X := Read(INPUT)
   SUM := SUM + X
    (K := 1?)
    (K := K - 1)
     X := Read(INPUT)
   SUM := SUM -I- X
    (K  = 1?)
OUTPUT    Print(SUM)
```

3 (c)

Figure 3. Execution Sequences for Programs of Figure 2 with Non-kernel Statements in Parentheses.

Example 2.
In Figure 4 two programs are given. It is readily verified that each has a single kernel output variable, OUTPUT, a single kernel input variable, INPUT, the single control input variable MODE, and the kernel statements:

```
     X := Read(INPUT)
OUTPUT := Print(RES)
   RES := RKS+X
   RES := RES*X
   RES := 0
   RES := 1
```

If we assume the range of values for MODE to be the set {'ADD','MULT'), then we have two possible initial assignments of a value to the control variable MODE. Looking at the kernel execution sequences of each program for each assignment to MODE, it is readily verified that they are identical in both cases. Thus the programs are kernel equivalent and also functionally equivalent with respect to the input variables INPUT and MODE and the output variable OUTPUT.

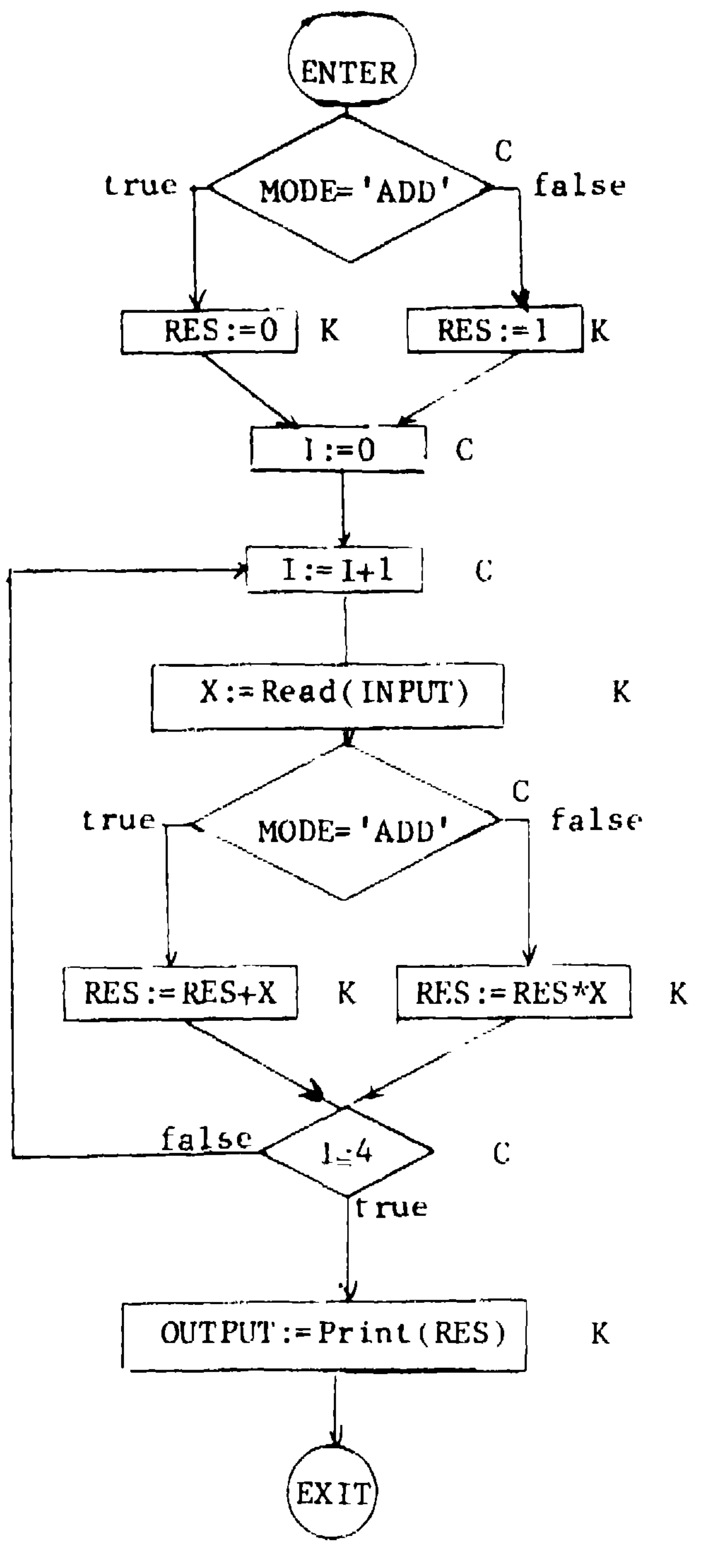
Kerne ls and Proving Correctness of Programs.
Studies of program equivalence are closely related to studies of program correctness. How does the concept of kernel equivalence aid in proving a program correct? We may readily observe the similarity between the "test case" approach to proving kernel equivalence and the "case analysis" approach to proving correctness discussed by London in his survey of techniques for proving correctness (4). If we look at the kernel execution sequence for each case of initial value assignments to the control input variables in a program, then we might reasonably argue that the program is "correct" if it has the "correct" kernel execution sequence in each case. Since each kernel execution sequence is a simple linear sequence of statements, it appears likely that one could more readily prove the "correctness" of such a statement sequence than the "correctness" of the original program taken as a whole, and that thus this technique of proving correctness by "case analysis" would often be useful (and might quite conceivably be mechanized). As a useful debugging tool also, one might envision a "kernel analysis system" which, given a program and a list of its control input variables with their ranges, would simply list the kernel execution sequence for each combination oi input assignments to the control variables. Inspection of the kernel execution sequences should often suffice to show the "correctness" of the program control structure, and thus the correctness of the program as a whole if one were convinced that the kernel execution sequences were correct.

Conclusion

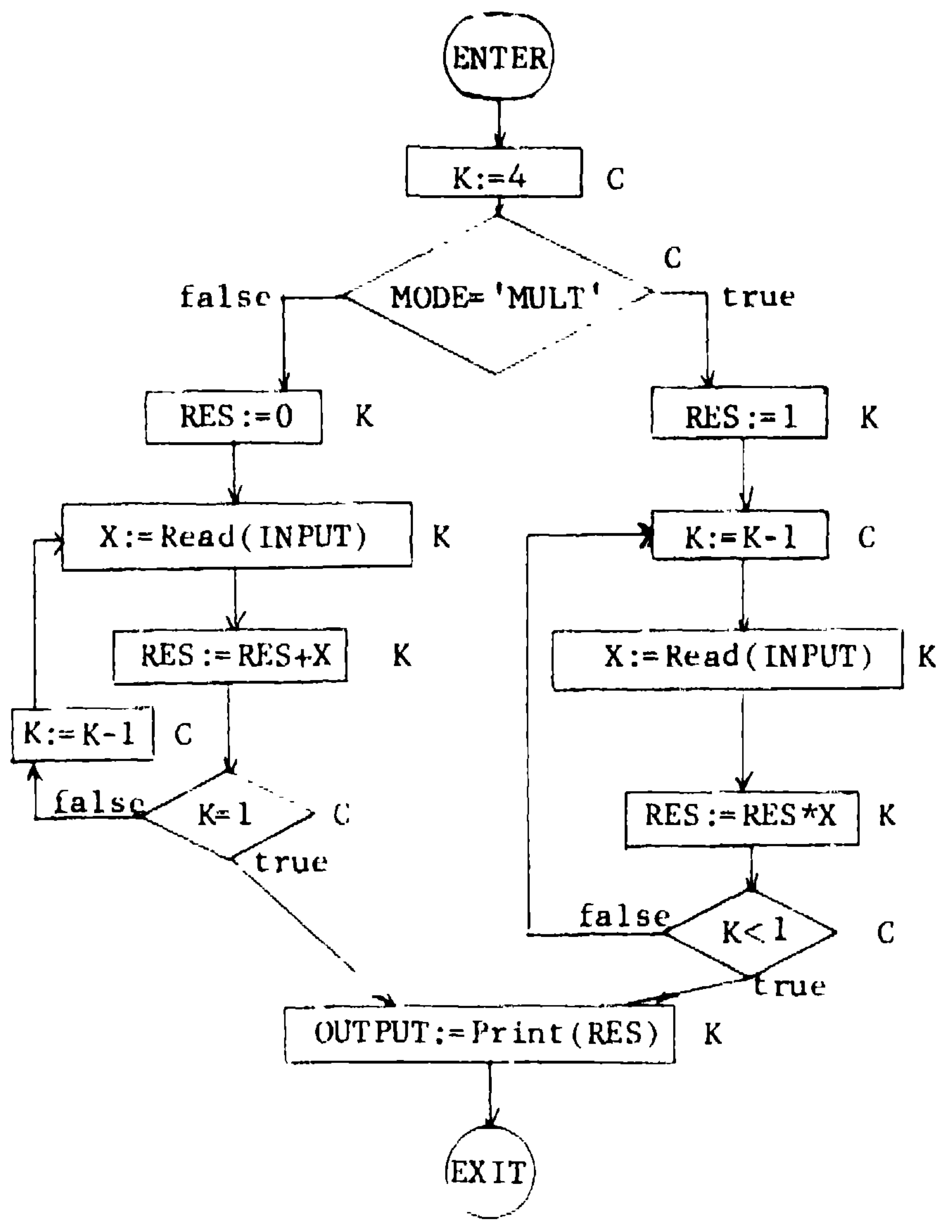
Kernel equivalence is a rather elementary yet not uninteresting form of program equivalence. Because determination of kernel equivalence requires only a superficial analysis of flow of control and data in a program, it lias some potential practical value for application to actual programs.

The analysis of programs in terms of "kernel" versus "control" statements and variables which is used here in an elementary way seems a powerful tool for studying programs.

Clearly there arc many other aspects of programs which are also primarily concerned with control, such as subprograms, argument transmission, iteration statements, and block structure, which often may be radically different between two programs which make the same "kernel" computation. An extension of kernel equivalence to encompass these other structures seems feasible without greatly increasing the depth of program analysis needed.



4 (a)



4 (b)

Figure 4.

Two Kernel Equivalent Programs.

REFERENCES

(1) Luckham, D. C., Park, M. R. and Patcrson, M. S. "On formalized computer programs," Jour. Comp. and System Sciences, 4,3, June 1970.

(2) Kaplan, D. M. "Regular expressions and the equivalence of programs," Jour. Comp. and System Sciences, 3,4, Nov. 1969.

(3) Orgass, R. J. "Some results concerning proofs of statements about programs," Jour. Comp. and System Sciences 4,1, Feb. 1970.

(4) London, R. L., "Computer programs can be proved correct," in Hanerji and Mesarovic (eds) Theoretical Approaches to Non-numerical Prob. Solving, Springer-Vcrlag, 1970.