Programming Languages
T.A. Standish Editor

# On the Capabilities of While, Repeat, and Exit Statements

W.W. Peterson
University of Hawaii
and
T. Kasami and N. Tokura
Osaka University

**A well-formed program is defined as a program in which loops and if statements are properly nested and can be entered only at their beginning. A corresponding definition is given for a well-formed flowchart. It is shown that a program is well formed if and only if it can be written with if, repeat, and multi-level exit statements for sequence control. It is also shown that if, while, and repeat statements with single-level exit do not suffice. It is also shown that any flowchart can be converted to a well-formed flowchart by node splitting. Practical implications are discussed.**

**Key Words and Phrases: well-formed program, while statement, repeat statement, exit statement, go to statement, flowchart, node splitting, software reliability**

**CR Categories: 4.39, 4.49, 5.29**



Peterson's work was supported in part by the U.S. Army Research Office, Durham, under Grant No. DA-ARO-D-31-124-71-G43 and the National Science Foundation under Grant No. GJ-596.
Authors' addresses: Department of Information and Computer Sciences, University of Hawaii, 2565 The Mall, Honolulu, HA 96822; Osaka University, Osaka, Japan.

## Introduction

Recently there has been considerable interest in the possibility of replacing the use of **go to** statements in programs with iterative constructions such as **for** and **while** because the latter are more easily understood, less error prone, and more easily proved correct. In recent papers, Knuth and Floyd [1] and Ashcroft and Manna [2] and Wulf [3] have presented some theoretical results on what can and what cannot be done. This paper presents further results along the line of Knuth and Floyd's investigation. It is shown here that if we require that neither the program length nor the execution time be increased, the program constructions **if**, **for**, **while**, and even a repeat statement with a multi-level exit statement, are not always sufficient. Thus it appears unlikely that any construction simpler than **go to** suffices with conditions this strict.

Then it is shown that if we relax the condition on program length but retain the condition on execution sequence, an **if** statement and a repeat statement with single-level exit are not always sufficient, but an **if** statement and a repeat statement with a multi-level exit are always sufficient. Finally some discussion of practical implications is given.

## Programs with the Same Length and Execution Sequence

We will assume that a program is represented by a flowchart with a single starting node $S$ and terminal node $T$, and that two flowcharts represent the same computation and are equivalent in that sense if the set of all possible paths from the starting node $S$ to the terminal node $T$ represents the same set of execution sequences. We shall consider the execution sequence to include tests, but not other sequence control. We shall say that two programs with the same execution sequence have the same execution time—this is approximately but not precisely true because of the exclusion of **go to** statements. By "all possible paths" through the flowchart, we mean assuming that all possible combinations of test outcomes are possible. This is the idea behind Ianov's "program schema" [4]. Note that if a program has no input data, for example, there is in fact only one path through the flowchart that can actually occur; but in spite of that, we consider the entire usually infinite set of paths that could occur if all possible combinations of test outcomes could occur. An equivalent statement is that, considering the flowcharts to be transition graphs, we consider that two flowcharts are equivalent and represent the same computation if and only if the associated regular sets are equal.

Three examples of flowcharts are shown in Figure 1, and corresponding programs are shown. Note that lowercase letters here represent some calculation that has one entry point and one exit point, such as an assignment statement or an ordinary subroutine call or a

number of such statements. In the examples, these statements are assumed to include the calculation of the predicates used in **if** statements, which are indicated by the letter $P$ with a subscript in the programs. The same notation will be used in representing the test in an execution sequence. Thus $sP_Aa_1P_Bb_1c$ is a possible path through the flowchart in Figure 1(a).

It is known that if two flowcharts are equivalent and both have the minimum possible number of nodes, they are isomorphic [5]. Furthermore, if a flowchart does not have the minimum number of nodes, it is possible to merge at least one pair of nodes. In order for it to be possible to merge two nodes, it is necessary (but not sufficient) that the branches going out of those nodes represent the same computations. Thus if every branch represents a different computation, then a flowchart has a minimum number of nodes. Thus, for example, in Figure 1(a), if $s$, $a_1$, $a_2$, $b$, $c_1$, and $c_2$ all represent different computations, then this flowchart has a minimum number of nodes. Every flowchart equivalent to this one is isomorphic to it unless it has more nodes and more branches.

If two flowcharts have the same set of tests and branches, we will say that the programs have the same length because the coding for each test and branch will appear once in the program. Again, the flowchart does not include a representation of **go to** statements, and therefore this definition of "same length" does not mean that the actual programs will have precisely the same length.

Now let us ask, Can the flowcharts in Figure 1 be implemented in programs that use only **if** and **while** statements for sequence control? Intuitively, the answer is no. Figure 1(a) needs no loop, and we can see by trying all possibilities with two **if** statements that they will not do. Figures 1(b) and 1(c) represent loops. They cannot be implemented with **while** statements because any loop implemented with a **while** statement has exactly one entry point and one exit point, while the loop in Figure 1(b) has two exit points and the one in Figure 1(c) has two entry points. This can be proved rigorously.

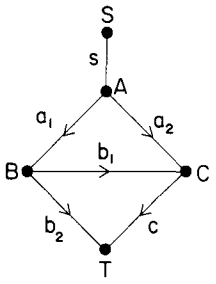
THEOREM 1. *There exist flowcharts that cannot be translated into if-while programs without increasing their lengths and/or changing their execution sequence.*

PROOF. We will show that the flowcharts in Figure 1 cannot be translated into if-while programs. For any of these flowcharts, if we assume that each branch represents a distinct computation, then this flowchart is minimal, and any equivalent flowchart with the same number of branches is isomorphic to it. Thus assuming the same execution sequence and program length implies using essentially the same flowchart.

Suppose one of these programs could be implemented using **if** and **while** statements for sequence control. Then in that program, either there is an **if** statement that does not contain an **if** or **while** statement in either its **then** or **else** clause, or else there is a

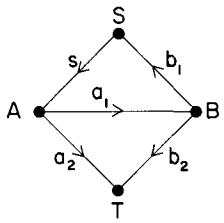
Fig. 1. Some flowchart and program examples.

(a)



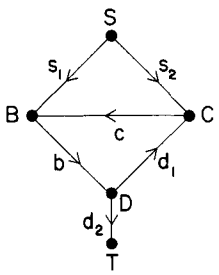

*S*: *s*;
**if** $P_A$ **then do**;
$a_1$ ;
**if** $P_B$ **then do**; $b_1$ ; **go to** *C*; **end**;
**else do**; $b_2$ ; **go to** *T*; **end**;
**end**; **else do**;
$a_2$ ;
*C*: *c*;
**end**;
*T*: **stop**;

(b)



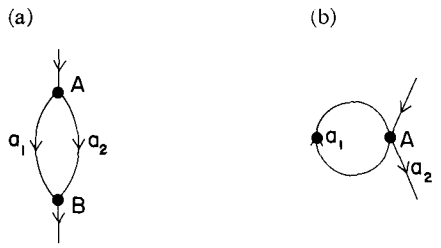

*S*: *s*;
**if** $P_A$ **then do**;
$a_1$ ;
**if** $P_B$ **then do**; $b_1$ ; **go to** *C*; **end**;
**else do**; $b_2$ ; **go to** *T*; **end**;
**end**; **else do**;
$a_2$ ;
*C*: *c*;
**end**;
*T*: **stop**;

(c)




*S*: **if** $P_S$ **then do** $S_1$ ; **go to** *B*; **end**;
**else do**; $s_2$ ; **end**;
*C*: *c*;
*B*: *b*;
**if** $P_D$ **then do** $d_1$ ; **go to** *C*; **end**;
**else do**; $d_2$ ; **end**;
*T*: **stop**;

Fig. 2. (a) Flowchart of **if** $P_A$ **then** $a_1$ ; **else** $a_2$ . (b) Flowchart of **while** $P_A$ ; $a_1$ ; **end**; $a_2$ ;.




**while** statement that does not contain an **if** or **while** statement within its loop. The flowcharts for such **if** or **while** statements are shown in Figures 2(a) and 2(b), respectively.

Actually, the branch $a_1$ or $a_2$ in Figure 2(a), or the branch $a_1$ in Figure 2(b), or any combination of them might be replaced by several branches connected by nodes with only one branch coming in and one going out, corresponding to a compound statement containing no **if** or **while** statement. Then one or the other of these must appear as a subgraph of any flowchart corresponding to a program using only **if** and **while** for controlling execution sequence. (We consider $H$ to be a subgraph of a graph $G$ if $H$ can be derived from $G$ by deleting some nodes and/or branches. Note that a subgraph may contain a branch without containing both nodes that it connects in the graph.) But neither of the subgraphs in Figure 2 appears as a subgraph in any flowchart in Figure 1. It follows that these flowcharts cannot be implemented with **if** and **while** only as sequence control. QED

Figures 1(b) and 1(c) are topologically equivalent to the programs considered by Knuth and Floyd as examples. They give a proof whose meaning is a little obscure, that 1(b) cannot be implemented with **if** and **while** even if node splitting is allowed. In fact their proof is related closely to the fact that a **while** clause has only one exit, but Figure 1(b) has two. Thus in Figure 1(b), the statements $a_1$ and $b_1$ are both in the loop, but their alternatives are both outside. This kind of situation is impossible with a **while** clause; since there is only one exit, only one test can have one consequence in the loop and one outside. If there is another test result in the loop, its alternative must also be in the loop.

Knuth and Floyd show that Figure 1(c) can be implemented using **if** and **while,** if the flowchart is changed by node splitting. Note that the resulting flowchart seems to indicate a longer program—it has one more branch than the old program. However, they used the semantics of their specific program to simplify it. The computation associated with the one new branch could be modified slightly without changing the result, and with this additional change the program is no longer than the original. This could not be expected in general—in general, node splitting will result in a longer program.

## Repeat and Exit Statements

Knuth and Floyd observed that the program represented by Figure 1(b) can be implemented with a repeat statement and an exit statement that exits from the loop formed by the repeat statement. This arrangement allows additional flexibility compared to the **while** statement in that the exit can be placed at any point in the loop, and also multiple exits from a loop are possible. This is just exactly what is needed for Figure 1(b), but it would not seem to help with Figures 1(a) or 1(c). Now we examine this question more carefully.

We wish to consider not only the single-level exit statement considered by Knuth and Floyd but also a multi-level exit statement. For that purpose, the exit statement must show which loop is being exited. We have chosen to use as a repeat statement the statement **do forever** that is found in XPL[6]. We will require a label on the **do forever**, and the same label on the matching **end** statement, and we will write an exit from that loop by **exit** followed by the label. The following example and its equivalent, written with **go to** statements, should make these ideas clear:

```
A: do forever;                    A:
     s1 ;                             s1 ;
     if PA then exit A;               if PA then go to A1;
B:   do forever;                  B:
       s2 ;                             s2 ;
       if PB then exit B;               if PB then go to B1;
       if PC then exit A;               if PC then go to A1;
       s3 ;                             s3 ;
     end B;                           go to B;
     s4 ;                         B1: s4 ;
   end A;                             go to A;
   stop;                          A1: stop;
```

Now let us consider whether the programs in Figures 1(a) and 1(c) can be implemented without increasing the program length and without changing the execution sequence, using the **do forever** and **exit** statements. It would appear at first sight that they cannot because Figure 1(a) doesn't even have a loop and Figure 1(c) requires multiple entry to a loop, which is not possible with the **do forever**. That the problem is not quite so simple is illustrated by the following program, which corresponds exactly to Figure 1(a).

```
     S: s;
T:   do forever;
       C:  do forever;
           if PA then do;
             a1 ;
             if PB then do; b1 ; exit C; end;
             else do; b2 ; exit T; end;
           end; else do; a2 ; exit C; end;
           end C;
           c;
           exit T;
       end T;
       stop;
```

Neither of the "loops" $T$ and $C$ can ever loop, but the multiple exit feature provides enough flexibility to provide the sequence control for this program.

The program represented by Figure 1(c) is actually impossible with the multi-level exit statements, however.

THEOREM 2. *There exist flowcharts which cannot be translated into if-repeat programs without changing their execution sequence or increasing their length even with multi-level exit allowed.*

PROOF. Assume that we have a program for Figure 1(c) using **if**, **do forever**, and **exit** for sequence control. Since there is a loop in Figure 1(c), there is at least one place on the program where control proceeds upward in the program. That means that there is at least one **do forever** statement in which control can reach the matching **end** statement. The set of statements including the **do forever**, the **end**, and all statements between them corresponds to a subgraph of the graph in Figure 1(c) that has the following properties.

1. It contains a closed path and therefore contains at least one node and one branch.
2. There is one node of the subgraph, the entry point, which is on a closed path contained in the subgraph, and there is a path from the entry point to each node in the subgraph.
3. There may be any number of nodes in the subgraph with branches going to nodes outside the subgraph, but there is only one node of the subgraph with branches coming in from nodes outside the subgraph, namely, the entry point.

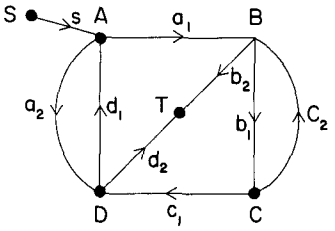
For the flowchart in Figure 1(c), these three conditions are not satisfied by the whole flowchart or any subgraph. It follows that there is no program using **do forever,** multi-level **exit**, and **if** alone for sequence control for the program in Figure 1(c). QED

Now let us relax the condition that the program length be the same and require only that the execution sequence be the same. This means that we will consider flowcharts that can be derived from the original flowchart by node splitting. We show in the next section that for any flowchart we can, by node splitting, make an equivalent flowchart that can be implemented with **if**, **do forever**, and multi-level **exit**. First we show that the multi-level exit is needed.

THEOREM 3. *There exist flowcharts that cannot be translated into if-repeat programs with single-level exits, even if node splitting is allowed.*

PROOF. Consider the flowchart in Figure 3. Assuming all the branches represent distinct computations, no node merging is possible and this flowchart is minimal. Any flowchart equivalent to it can be derived from this one by node splitting. On such a flowchart, each node can be associated with a node on the original graph—the node from which it was derived. Then, for example, any node on the new flowchart associated with node $A$ on the original graph, has two branches going out, one to a node associated with $B$ and one to a node associated with $D$.

Fig. 3. A flowchart with nested loops.




Now let us assume that we have a flowchart equivalent to that in Figure 3 and a program which uses only **if**, **do forever**, and single level **exit** for sequence control. Since the flowchart in Figure 3 has closed paths, control must proceed upward in the program at some point, and therefore there is at least one **do forever** loop in which control reaches the end of the loop and returns to the beginning. Then there must be one such **do forever** loop that has no other such **do forever** loop within it. Consider the subgraph $H$ that consists of the entry point of this loop and all statements within the loop, but not the exit node. Then it must have the following properties.

1. Every closed path must go through the entry point because we have assumed that this **do forever** loop contains no other closed **do forever** loop.
2. There is only one exit node because only single-level exit is permitted.

Now some node in $H$ must be the entry node. Let us assume it is associated with $A$ and call it $A_1$. It has two branches going out to nodes $B_1$ and $D_1$ associated with $B$ and $D$, respectively. At most, one is the exit node, so at least, one is in $H$. If $B_1$ is in $H$, then branches go from $B_1$ to $C_1$ and $T$. Since $T$ is the end of the program it cannot be in $H$, so it must be the exit node. Thus $C_1$, and also $D_1$, must be in $H$. If we assume $D_1$ rather than $B_1$ is in $H$, then we will be led to conclude in the same way that $B_1$ and $C_1$ are in $H$ also. Then there must be branches going out from $C_1$ to nodes associated with $D$ and $B$, respectively. The one associated with $B$ cannot be $B_1$ because that would make a closed path that doesn't go through the entry node, so let us call it $B_2$. $B_2$ connects to nodes associated with $C$ and $T$, and the one associated with $C$ cannot be $C_1$ because that would result in a closed path not going through the entry node. Call it $C_2$. Clearly then, there is no end to the number of nodes associated with $B$ and $C$; and since the flowchart must be finite, we are led to conclude that the entry node cannot be associated with $A$. A similar argument shows that it cannot be associated with $B$ either. By the symmetry of the flowchart, the situation must be the same for $C$ as for $A$ and the same for $D$ as for $B$. Since no node can be the entry node, it is impossible to find a flowchart equivalent to Figure 3 that can be implemented with **if**, **do forever**, and single-level **exit.** QED

## Well-formed Flowcharts

In this section a well-formed flowchart is defined, and it is shown that any flowchart can be transformed into a well-formed flowchart by node splitting. Later sections show that a flowchart is well formed if and only if the corresponding program can be written with properly nested **if**, **do forever**, and multi-level **exit** statements.

Intuitively, the necessary and sufficient condition for a flowchart to correspond to a **do forever** program is simply that no loop should have multiple entry nodes. The precise definition of entry node to a loop in a complicated flowchart is slightly difficult, however.

Let us define two nodes $A$ and $B$ as connected if there is a path from $A$ to $B$ and a path from $B$ to $A$, i.e. if there is a closed path that goes through both points. A subgraph consisting of all nodes connected to a given node, and all branches joining two nodes in this set, is called a component [7, 8]. Any node in a component which has a branch coming to it from outside is an entry node, and if the starting node is in a component, it is also an entry node.

We will call a flowchart well formed if it satisfies the following conditions: every component must have exactly one entry node. It is also necessary to remove all branches going into the entry node, thus opening all closed paths through the entry node, and to examine any new components that occur. Again, multiple entry nodes must not occur. This process must be continued until there are no more closed paths.

The first step in our program-writing algorithm of the next section is to open all closed paths. We present here an algorithm that examines a component for multiple entry points, removes extra entry points by node splitting, and opens the closed paths through the entry node. If this algorithm is applied repeatedly, the result will be a well-formed flowchart with the closed paths opened, ready for the program-writing algorithm of the next section.

THEOREM 4. *Every flowchart can be transformed into an equivalent well-formed flowchart by node splitting.*

PROOF. Suppose we have a flowchart that contains a component $U$ with multiple entry nodes. Choose one, $X$, to become the unique entry node; the others, $Y_1, Y_2, \ldots$, to be removed by node splitting. Next, introduce new nodes $Y_1', Y_2', \ldots$ and remove any entry branches from $Y_1, Y_2, \ldots$ and connect them to $Y_1', Y_2', \ldots$, respectively. Now for each primed node $Z'$, including the ones introduced in this step, if the original node $Z$ connects to a node $W$ outside $U$, place a branch representing the same processing from $Z'$ to $W$. If $Z$ connects to $X$, then connect $Z'$ to $X$ with a branch representing the same processing. If $Z$ connects to any other node $W$ of $U$, then make a new node $W'$ if this hasn't already been done, and connect $Z'$ to $W'$ with a branch representing the same processing as the branch from $Z$ to $W$.

It can be seen that the new flowchart and the old one have the same set of execution sequences, and the old flowchart results if the corresponding primed and unprimed nodes are merged in the new one. In the new flowchart, the component containing $X$ is exactly $U$, as in the old flowchart; for if there were a closed path through $X$ not entirely contained in $U$ on the new graph, it must exit from $U$ to another part of the graph, and return to $X$ through one of the nodes $Y_1', Y_2', \ldots$. Then there would also be one of the original flowchart exiting from $U$, and returning to $U$ via one of the nodes $Y_1, Y_2, \ldots$. This path would have been included in $U$, so it is impossible. Finally, it can be seen that the component $U$, which includes $X$, contains only $X$ as an entry point in the new flowchart.

The next step is to open the closed paths through $X$, introduce a new node $X^*$, and for each branch in the component $Y$ coming into $X$, remove the head of the branch from $X$ and connect it to $X^*$ instead.

The process of removing entry points and opening loops should be repeated until the resulting flowchart has no loops. An important question is, Will the process terminate or can it continue indefinitely? Let us define the complexity of a flowchart by a pair of integers $[N, M]$ where $N$ is the number of nodes in a largest multi-entry component and $M$ is the number of multi-entry components with $N$ nodes. We define $[N_1, M_1] < [N_2, M_2]$ to mean that either $N_1 < N_2$ or $N_1 = N_2$ and $M_1 < M_2$. The process just described for removing multiple entries from one component will decrease $M$ by 1 if $M > 1$, or will cause $N$ to decrease if $M = 1$, thus always decreasing complexity $[N, M]$. Note that although in the process of eliminating multiple entries from a component $U$ node splitting may introduce new components, none of the new ones will contain as many nodes as $U$. Thus repeated elimination of multiple entries from the most complicated components will eventually reduce the complexity to [0, 0], i.e., the resulting flowchart will be well formed. QED

As examples, Figure 1(a) has no closed paths, and therefore is well formed. In Figure 1(b), nodes $A$, $B$, and $S$ form a component, and $S$ must be considered an entry node. However, since this is the only entry, this flowchart also is well formed. In Figure 1(c), nodes $B$, $C$, and $D$ form a component, and $B$ and $C$ are entry nodes. If $B$ is chosen as the entry node and $C$ is split, the well-formed flowchart shown in Figure 4 results. This turns out to be the better choice. In the flowchart of Figure 3, $A$, $B$, $C$, and $D$ make a component, and $A$ is the only entry node. If the closed paths through $A$ are opened, then $B$ and $C$ form a component, and $B$ is the only entry point. There are no other components, so this flowchart is well formed. It is shown in Figure 5 with its closed paths opened, ready for the program-writing algorithm which is given in the next section.

## Well-formed Programs

In this section, the concept of a well-formed program is defined, and it is proved that from a well-formed flowchart a well-formed program using **if** and **go to** statements for sequence control can be produced. It is also shown that such a program can be converted into a program using **if**, **do forever**, and multi-level **exit**, but no **go to** statements for sequence control.

First let us define a **go to** statement to be upward if the target label precedes the **go to** statement, and otherwise downward. We will define the program segment of an upward **go to** statement to be the set of all statements between the label and the **go to** statement excluding the label and the **go to** statement. Note that a complete **if** statement consists of a predicate and two clauses—the **then** clause and the **else** clause—and either clause may be empty or may contain a simple or compound statement in ALGOL-like or PL/I-like languages. We will call the set of statements between a **do forever** statement and its matching **end,** a **do forever loop.**

We shall say that a program which may include **if, go to, do forever**, and **exit** statements is **well formed** provided that the following hold.

1. Whenever any combination of two **if** statements, **do forever** loops, or upward **go to** segments, has a statement in common, one is entirely contained in the other.
2. If any **if** statement, **do forever** loop, or upward **go to** segment is contained in an **if** statement, it is contained within one clause of that **if** statement.
3. If a label is in an upward **go to** segment, every **go to** statement to that label must be in that same upward **go to** segment. Similarly if a label is in a **do forever** loop, every **go to** to that label must be in that loop; and finally, if a label is in one clause of an **if** statement, all **go to**'s to that label must be in the same clause.

Conditions 1 and 2 require that **do forever** loops, upward **go to** segments, and **if** statements be nested, and 3 requires that **do forever** loops, upward **go to** segments, and **if** statements be entered only at their beginning. It follows from this last condition that the flowchart for a well-formed program has no multiple-entry component, and is thus well formed. Given a well-formed flowchart, if one writes a program using **if** and **go to** statements, it may or may not be well formed. Considerable care must be taken in the order in which the program is written. However, given any well-formed flowchart, the following algorithm produces a well-formed program.

Let us assume that a well-formed flowchart has its closed paths opened as described in the preceding section. We will refer to the node $X^*$ resulting from opening the closed paths through the entry node $X$ as the end node corresponding to $X$. We will say that a node is a *merge node* if more than one branch enters it.

We will also assume that in the original flowchart,

Fig. 4. A flowchart equivalent to Figure 1(c) but with no multiple-entry loops.

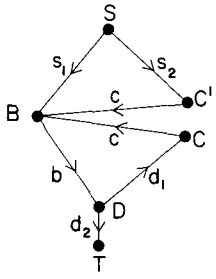
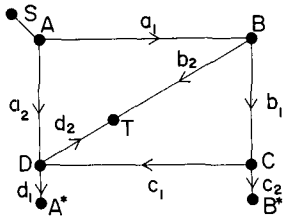



Fig. 5. The flowchart of Figure 3 with closed paths opened.




for any node $X$, there is a path from the starting node $S$ to $X$, and there is a path from $X$ to the terminal node $T$. If there were no path from $S$ to $X$, then $X$ could never be reached and hence could be eliminated without affecting the execution. If there were no path from $X$ to $T$, and if execution reached $X$, the program could not terminate. Thus these are reasonable assumptions. Next we introduce a partial order on nodes—we say $A$ is above $B$, or $B$ is below $A$, if there is a path from $A$ to $B$. Note that since we assume the closed paths have been opened, then if $A$ is above $B$, $B$ is not above $A$. Finally we say that a node $A$ covers a subgraph $G$ if every path from the starting node $S$ to a node or a branch in $G$ goes through $A$, and $A$ is not contained in $G$. If $A$ and $B$ both cover $G$, then every path from the starting node to $G$ goes through both $A$ and $B$, and either $A$ is above $B$ or $B$ is above $A$. Thus of all the nodes that cover $G$, there is a unique lowest cover.

The algorithm, which requires the use of a last-in-first-out stack, is this: Initially let $X$ be the starting node $S$, and let this be Case 1. Do the following repeatedly:

If Case 1, then do the following. (The next thing to be processed is a node.)

Step 1. If $X$ is the lowest cover for any node $Y$ which is a merge node but not an end node nor the terminal node, then stack $Y$. If $X$ is the starting node $S$, then stack $T$. If $X$ is an entry node, then stack $X^*$. Note that this implies that no node will be stacked more than once. If more than one node is stacked on one application of this step, then if $Y_1$ is above $Y_2$, then stack $Y_2$ first. Otherwise any order is acceptable.

Step 2. Write the label "$X$:".

Step 3. If $X$ is the terminal node $T$, then write "**stop**", and terminate the algorithm—the program is complete.

Step 4. If $X$ is an end node $Y^*$, then write "**go to** $Y$;" and let this be Case 3.

Step 5. If $X$ has two branches going out, then write "**if** $P_X$ **then do**;" and stack $XT$ to indicate that the **then** branch coming out of $X$ is being processed. Take this branch as the next value of $X$ and let this be Case 2.

Step 6. Otherwise this node has one branch going out. Take it as the next value of $X$, and let this be Case 2.

If Case 2, then do the following. (The next thing to be processed is a branch. Let $Y$ be the node which this branch enters.)

Step 1. Write the coding for the processing indicated for this branch.

Step 2. If $Y$ is a merge node or the terminal node $T$ or an end node, then write "**go to** $Y$" and let this be Case 3.

Step 3. Otherwise take $Y$ as the next value of $X$, and let this be Case 1.

If Case 3, then do the following. (The next thing to be processed is determined by the stack.)

Step 1. If the top of the stack is of the form $YT$ for some $Y$, then write "**end**; **else do**;", take the **else** branch at $Y$ as the next value of $X$, replace $YT$ on the stack by $YF$ to indicate that the **then** clause starting at $Y$ has been completed and the **else** clause is now being processed, and let this be Case 2.

Step 2. If the top of the stack is of the form $YF$ for some $Y$, then write "**end**;" and remove $YF$ from the stack to indicate that the **else** clause starting at $Y$ has been completed. Let this be Case 3.

Step 3. Otherwise the top of the stack must be a node name. Remove it, take it as the next $X$, and let this be Case 1.

THEOREM 5. *This algorithm produces a correct well-formed program from a well-formed flowchart.*

PROOF. Corresponding to every node in the flowchart there is a label in the program. If a node $X$ has only one branch going out, the coding for that branch immediately follows the label "$X$:". If there are two branches going out, then the coding corresponding to the **true** branch immediately follows "$X$: **if** $P_x$ **then do**;" and the coding for the **false** branch immediately follows "**end**; **else do**;" and the "**end**" matches the **do** for the **then** clause. If the node $Z$ follows the branch $Y$, then either "**go to** $Z$" or the label $Z$ immediately follows the coding for the branch $Y$. Thus the execution of the program indeed matches the flowchart.

Next we prove the following about the algorithm.

LEMMA. *If $X$ is a node or a branch and $Y$ is a node or branch in the flowchart and $X$ is above $Y$, then $X$ is processed before $Y$ by the algorithm.* (This means that the corresponding coding for $X$ will precede that for $Y$ in the program, also.)

PROOF. It follows directly from the algorithm that a branch is never processed before the node which it leaves. If a node is not an end node, a merge node, or the terminal node, then Case 2 Step 3 applies, and the node is processed after the branch which enters it. If we can prove that for merge nodes, end nodes, and the terminal node, that the node isn't processed before all entering branches are processed, then the lemma follows.

Let $X$ denote any end node, merge node, or the terminal node, and let $Y$ denote the node at which $X$ is stacked. If $X$ is a merge node but not an end node or the terminal, $Y$ is the lowest cover of $X$. If $X$ is an end node, then $Y$ is the corresponding entry node. Since the flowchart is well formed, this component containing $X$ and $Y$ has only one entry node, $Y$, and thus every path from $S$ to $X$ goes through $Y$. If $X$ is the terminal node, then $Y$ is the starting node $S$, and trivially every path from $S$ to $X$ goes through $Y$. Thus in each case $Y$ is a cover for $X$, but not necessarily the lowest.

Such a node $X$ is processed when reference is made to the stack (Case 3 in the algorithm) and when $X$ is on the top of the stack. Let us assume that (a) a reference to the stack is about to be made, and (b) there is at least one branch $B$ entering $X$ which has not yet been processed. If $Y$ has not yet been processed, then $X$ hasn't yet even been put on the stack (Case 1, Step 2 of the algorithm) and so cannot be on top. Now let us assume $Y$ has been processed. There must exist a path from $S$ to $B$. Since $B$ goes to $X$, that is also a path from $S$ to $X$ and hence goes through $Y$ because $Y$ is a cover for $X$. Let $Z$ be the first node or branch below $Y$ on that path that hasn't been processed. We must consider two cases—$Z$ is a branch and $Z$ is a node.

If $Z$ is a branch, then the node $W$ which $Z$ goes out of is processed. $W$ must not have been the last thing processed, for in that case no reference to the stack would be required at this point. The only possibility is that $W$ has two branches going out and $Z$ is the **else** branch. In that case $WT$ must be on the stack, and since either $W$ is $Y$ or $W$ is below $Y$, $WT$ must be above $X$ on the stack. Thus $X$ is certainly not on top.

Now suppose $Z$ is a node. Since there must be a branch going out of $Z$, $Z$ cannot be an end node or the terminal node. If $Z$ were not a merge node, then it would have been processed immediately after the branch going into it was processed (Case 1 Step 5) without reference to the stack. Thus $Z$ is a merge node. Since $Z$ is on a path from $Y$ to $X$, every path from $S$ to $Z$ can be extended to $X$. Since $Y$ is a cover for $X$, every path from $S$ to $X$, and hence every path from $S$ to $Z$, goes through $Y$. Thus $Y$ is a cover for $Z$. The lowest cover for $Z$ is either $Y$ or below $Y$. In either case, the algorithm (Case 1 Step 2) will stack $Z$ after it stacks $X$, and $Z$ is still on the stack. Thus $X$ is not on top.

We are forced to conclude that any stack reference made before all branches going into $X$ are processed cannot result in processing $X$, and thus $X$ is not processed until after all branches entering it have been processed, and thus the lemma is proved. QED

The algorithm will terminate only when the terminal node $T$ is processed. Since we have assumed that there is a path from every node to $T$, every node and every branch is above $T$, and it follows directly from the lemma that every other branch and node will be proc-

essed before $T$, and when $T$ is processed, processing is indeed complete.

Let $X$ be an entry node and $X^*$ the corresponding end node. Since the flowchart is well formed, every path from $S$ to $X^*$ must go through $X$, and thus $X$ is above $X^*$. Thus the **go to** written in Case 1 Step 4 of the algorithm is an upward **go to**. It follows from the lemma that all **go to**'s produced by Case 2 Step 2 are downward.

Every time an upward **go to** segment, **then** clause, or **else** clause is started, an item of the form $X^*$, $XT$, or $XF$, respectively, is stacked. Thus at any point in the writing of a program, the stack indicates which segments contain that point in the program. At the end of any such segment, the corresponding item is removed from the stack. Since it is a last-in-first-out stack, these segments must be properly nested. Now suppose $X$ is the target of one or more downward **go to**'s. The **go to** must have been produced by Case 2 Step 2, and thus $X$ must be an end node, a merge node, or the terminal node. Let $Y$ be the node at which $X$ is stacked. It was shown in the proof of the lemma that $Y$ is a cover for $X$. Every upward **go to** segment or **then** or **else** clause containing $X$ is indicated by an item on the stack below $X$. Thus any such item was stacked no later than the time at which $Y$ was processed. The **go to** $X$ statements are written whenever a branch entering $X$ is processed. By the lemma, these will always occur after $Y$ is processed and before $X$ is processed because every branch entering $X$ is below $Y$ and above $X$. Thus these **go to** $X$ statements will be written while $X$ is stacked and hence will be within every upward **go to** segment or **if** clause containing $X$. Thus the program resulting from the algorithm is well formed, and the theorem is proved. QED

As an example, the flowchart of Figure 3, shown in Figure 5 with its closed paths opened, has $A$ and $B$ as entry nodes, $A^*$ and $B^*$ as end nodes, and $T$ and $D$ as merge nodes. For both $T$ and $D$, $A$ is the lowest cover. The program resulting from the application of the algorithm, along with the stack contents after each step that affects the stack, is as follows:

| Program | Stack (after this step) |
|---|---|
| $S$: $s$; | $T$ |
| $A$: **if** $P_A$ **then do**; | $T, A^*, D, AT$ |
| $a_1$ ; | |
| $B$: **if** $P_B$ **then do**; | $T, A^*, D, AT, B^*, BT$ |
| $b_1$ ; | |
| $C$: **if** $P_C$ **then do**; $c_1$ ; | $T, A^*, D, AT, B^*, BT, CT$ |
| **go to** $D$; | |
| **end**; **else do**; $c_2$ ; | $T, A^*, D, AT, B^*, BT, CF$ |
| **go to** $B^*$; | |
| **end**; | $T, A^*, D, AT, B^*, BT$ |
| **end**; **else do**; $b_2$ ; | $T, A^*, D, AT, B^*, BF$ |
| **go to** $T$; | |
| **end**; | $T, A^*, D, AT, B^*$ |
| $B^*$: **go to** $B$; | $T, A^*, D, AT$ |
| **end**; **else do**; $a_2$; | $T, A^*, D, AF$ |
| **go to** $D$ ; | |
| **end**; | $T, A^*, D$ |
| $D$: **if** $P_D$ **then do**; $d_1$ ; | $T, A^*, DT$ |
| **go to** $A^*$; | |
| **end**; **else do**; $d_2$ ; | $T, A^*, DF$ |
| **go to** $T$; | |
| **end**; | $T, A^*$ |
| $A^*$: **go to** $A$; | $T$ |
| $T$: **stop**; | *empty* |

This program can be simplified. For example, the statements "**go to** $B^*$:" "**go to** $A^*$:", and the second "**go to** $D$;" could simply be omitted with no effect on execution. It is clearly possible to refine the algorithm to eliminate these unneeded **go to**'s, but all our attempts so far have resulted in considerably more complicated algorithms and proofs.

Next we show that every well-formed program can be written without **go to** statements.

THEOREM 6. *A well-formed program can be transformed into an equivalent program by deleting the* **go to**'s *and inserting repeat and exit statements and labels, but without rearranging or altering other statements.*

PROOF. For each upward **go to** statement, **go to** $X$, place a **do forever** statement immediately following the target label $X$ and replace the statement **go to** $X$ by **end** $X$. This changes upward **go to** segments into **do forever** loops. It follows directly from the definition of a well-formed program that if the original program was well formed, the new one is also.

Next, if there are any labels which serve as targets for downward **go to** statements and simultaneously serve some other function such as labeling a **do forever** statement, insert a new distinct label preceding the existing label, and relabel all the downward **go to** statements using the new label.

Now the last target label $X$ of downward **go to** statements and those **go to** statements can be replaced by **do forever** and **exit** statements as follows.

1. Replace the label $X$ by "**exit** $X$; **end** $X$;" and remove the label.
2. Find the smallest **do forever** loop or **if** clause that contains $X$. If it is a **do forever** loop labeled $Y$, place "$X$: **do forever**" immediately following the statement "$Y$: **do forever**;". If it is an **if** clause, place "$X$: **do forever**;" immediately following "**then do**;" "**else do**;" or if no "**do**;" occurs, immediately following the "**then**" or "**else**". If there is none, place "$X$: **do forever**;" at the beginning of the program.
3. Replace each statement **go to** $X$; by **exit** $X$;.

It follows directly from the definition of the statements "**do forever**" and "**exit**" that this doesn't affect the sequence of execution. It follows from the assumption that the program is well formed, that any statement **go to** $X$ is inside the smallest **do forever** loop or **if** clause that contained the label $X$. Since the **go to**'s are forward, they, and hence the exit statements which replace them, are in the new **do forever** loop labeled $X$. Furthermore, the new loop, being completely inside the smallest **do forever** loop or **if** clause that contains $X$, will be properly nested with all existing

**do forever** loops and **if** statements. Thus the resulting program is well formed.

By repeatedly eliminating the last label of downward **go to** statements we can eventually eliminate all **go to** statements, and the resulting program is a well-formed program using **if**, **do forever**, and **exit** statements but no **go to**'s.

The previous example of a well-formed program for the flowchart of Figure 3 (with the three redundant statements **go to** $B^*$, **go to** $A^*$, and **go to** $D$ omitted) is shown here with the **go to**'s removed and replaced by **do forever** and **exit** statements:

```
T:  do forever;
S:    s;
A:    do forever;
D:      do forever;
          if P_A then do;
            a_1 ;
B:          do forever;
              if P_B then do;
                b_1 ;
                if P_C then do; c_1 ; exit D;
                end; else do; c_2 ; end;
              end; else do; b_2 ; exit T; end;
B*;         end B;
          end; else do; a_2 ; end;
          exit D;
        end D;
        if P_D then do; d_1 ; end;
        else do;  d_2 ;  exit T;  end;
A*:   end A;
      exit T;
    end T;
    stop;
```

Note that this program also is not the simplest possible. For example, the statements "$T$: **do forever**;" and "**end** $T$;" could be omitted, and then each "**exit** $T$;" could be replaced by "**exit** $A$;".

### Relation to Practical Software

The motivation for studying the implications of having a programming language with no **go to** statements is primarily software reliability, and therefore these two questions arise: (1) Will use of the ideas presented here improve software reliability? and (2) Are there disadvantages that will make the use of other sequence-control statements in place of **go to** undesirable? Let us consider the second question first. We have shown that, in general, if we insist on the same execution sequence, the flowchart may have to be altered by node splitting, which implies that copies of certain codes may have to appear at several points in the program. This means a longer program, which is undesirable. However, it is certainly possible to keep the increase in program length within reasonable bounds. If the code that must be repeated is short, it is not serious. If it is long, then it can be written as a procedure and called at several places. This will cause a small increase in execution time, but will keep the increase in program length modest. Ashcroft and Manna [2] give another method for writing programs with **while** statements by introducing additional Boolean variables. Their method will result in a program that is slightly longer and slower than the original program. Together, these results seem to show that eliminating the use of **go to** statements will, at worst, result in modest increases in program length and execution time.

Next is the question, Is there a benefit that offsets this apparently modest cost? This question is certainly debatable. A few observations can be made, however.

First, it is easy to write a program that is obscure, complicated, and difficult to understand without using **go to** statements. Certainly any algorithms that we have now do not generally transform a complicated program with **go to** statements into a simple program with no **go to**'s. In fact, the program shown for Figure 1(a) without **go to** statements is more difficult to understand than the one using **go to**'s, and the very idea of introducing **do forever** loops that cannot actually loop merely for the purpose of eliminating **go to** statements is the kind of gimmick that we ought to avoid in programming.

Yet it is certainly true that programs can be written most clearly and simply when full advantage is taken of any constructs like **for** and **while** that may be available. Why do such programs seem clear and simple? Perhaps the most important reason is that they are well formed. It appears that it is not the **go to** statements themselves that are harmful, but rather their unrestricted use, especially in ways prohibited in our definition of a well-formed program. And on the other hand, a well-formed program written with **go to** statements may be at least as clear and easy to understand as the corresponding program with **do forever** and **exit** statements—especially if the upward and downward **go to**'s are distinguished in some way in the program listings.

For practical purposes there is another perhaps more convincing "proof" that well-formed programs can be used in practical programming—a demonstration.

Wulf has reported extensive programming experience with a **go to**—less language designed for systems programming [3, 10] with no serious problems and general enthusiasm about its effectiveness. A less extensive experience, but one for which complete documentation is readily available has been published by McKeeman, Horning, and Wortman [6]. The programs shown in their book comprise about 5000 statements typical of systems programming. There seems to be one **go to** statement, and that certainly could be eliminated. They use XPL, a PL/I-like language with **do** and **while** statements like those in PL/I. They make generous use of procedures, and there is one feature that they use which allows multiple exits from a loop—within a procedure, any number of **return** statements may appear, and they may be within nested loops, of course.

The most important observation to be made is that the program listings published by McKeeman are unusually clear and easy to understand, and there seems to be no need for flowcharts. Furthermore, the programming is natural and straightforward. There is no point at which it is obvious that they were inconvenienced by their choice not to use **go to**'s. (There are some places in their programs that are obscure, but these result from unrelated weaknesses in XPL—especially the lack of floating-point arithmetic and double-word boundaries.)

It seems very likely that there were, in fact, times when the first way to solve a problem that occurred to the authors involved the use of **go to**, and they had to think to find an alternative approach. If the result of that thinking is a clearer, better organized program, it was effort well spent.

There are other potential benefits to writing well-formed programs. The best optimizing compilers must analyze the flow of control in the program, and this may prove to be easier in well-formed programs [8]. It has proved helpful in at least one program verification system [11]. In well-formed programs the flow is always forward except between very clearly defined and paired endpoints of loops.

Some objections might be raised to the suggested notation for multi-level exits. There are other possibilities, of course. But the following points should be considered also.

Again, the motivation for the work presented here is software reliability—making programs less error prone and easier to understand. One very error-prone feature of ALGOL, PL/I, and similar languages is the fact that pairing of **do**'s and **end**'s can easily go awry. When it does, the compiler may give no diagnostic message, or it may give a profusion of diagnostic messages not obviously related to the true problem. And what ALGOL or PL/I programmer hasn't spent much time checking the pairings of **do**'s and **end**'s? Yet this problem appears to have escaped discussion in the literature. Elspas [9] has suggested that some discipline should be imposed on the programmer to promote software reliability. We suggest that requiring all matching **do**'s and **end**'s to be labeled is a discipline on the programmer which would definitely contribute to making programs less error prone and more easily understood. It would also make possible accurate diagnosis of this type of error by compilers.

## Conclusion

We have shown that for any given flowchart, a program can be written using if statements, repeat statements, and multi-level exit statements. The flowchart may have to be modified by node splitting. We have also shown that if one insists on precisely the same execution sequence as represented by the original flowchart, node splitting is sometimes necessary, and **if**, **while**, and repeat statements with only a single-level exit are not sufficient.

This study has led us to define a "well-formed program" as one in which loops and conditional statements are properly nested and entered only at their beginning. As a prerequisite to program clarity, requiring a program to be well formed, thus restricting the ways in which **go to** statements can be used, appears much more important than merely avoiding **go to** statements. The results presented in this paper, together with related work by other authors, indicate that it is certainly possible with modest sacrifice in execution time and/or program length to write well-formed programs. It also appears that we have not yet found the best language for sequence control in programs, and further discussion and research is indicated.

*Acknowledgment.* The idea of explicitly defining a "well-formed" program and several important improvements in our proofs were suggested to us by one reviewer.

Received October 1971, revised November 1972

**References**

1. Knuth, D.E., and Floyd, R.W. Notes on avoiding 'GO TO' statements. *Inf. Proc. Letters 1* (1971), 23–31.
2. Ashcroft, E., and Manna, Z. The translation of "go to" programs into "while" programs. Comput. Sci. Dep. Rep. CS 188, Stanford U., Jan. 1971.
3. Wulf, W.A. Programming without the GOTO. IFIP 71 TA-3-84.
4. Ianov, Y.I. The logical schemes of algorithms. In *Problems of Cybernetics, Vol. 1*, English ed., Pergamon Press, New York, 1960, pp. 82–140.
5. Hopcroft, J.E., and Ullman, J.D. *Formal Languages and Their Relation to Automata.* Addison-Wesley, Reading, Mass., 1969.
6. McKeeman, W.M., Horning, J.J., and Wortman, D.B. *A Compiler Generator.* Prentice-Hall, Englewood Cliffs, N.J., 1970.
7. Harary, F. *Graph Theory.* Addison-Wesley, Reading, Mass., 1968.
8. Cocke, J., and Miller, R. Some analysis techniques for optimizing computer programs. Proc. 2nd Int. Conf. of Syst. Sci., Hawaii, Jan. 1969.
9. Elspas, B., Green, M.W., and Levitt, K.N. Software reliability. *Computer 4* (Jan. 1971), 21–27.
10. Wulf, W.A. A case against the GOTO. Proc. ACM 72, pp. 791–796.
11. Lyons, T., and Bruno, J. An interactive system for program verification. Tech. Rep. No. 91, Dept. of E.E., Princeton U., Princeton, N.J., 1971.