An Algorithm for Structuring Flowgraphs

BRENDA S. BAKER

Bell Laboratories, Murray Hill, New Jersey

ABSTRACT This paper describes an algorithm which transforms a flowgraph into a program containing control constructs such as **if then else** statements, **repeat** (**do forever**) statements, multilevel **break** statements (causing jumps out of enclosing **repeats**), and multilevel **next** statements (causing jumps to iterations of enclosing **repeats**) The algorithm can be extended to create other types of control constructs, such as **while** or **until** The program appears natural because the constructs are used according to common programming practices The algorithm does not copy code, create subroutines, or add new variables Instead, **goto** statements are generated when no other available control construct describes the flow of control. The algorithm has been implemented in a program called STRUCT which rewrites Fortran programs using constructs such as **while**, **repeat**, and **if then else** statements The resulting programs are substantially more readable than their Fortran counterparts

KEY WORDS AND PHRASES structured programming, flow of control, flowgraph, program transformations, Fortran

CR CATEGORIES 4 19, 4.43, 5 24

1. *Introduction*

Structured programming emphasizes programming language constructs such as **while** loops, **until** loops, and **if then else** statements. Properly used, these constructs make occurrences of loops and branching of control clear. They are preferable to **goto** statements, which tend to obscure the flow of control and make programs hard to understand [10,11]. This paper describes an algorithm which transforms a flowgraph into a program written using **repeat** (**do forever**) and **if then else** statements, together with multilevel **break** statements (which cause jumps out of enclosing **repeats**) and multilevel **next** statements (which cause jumps to the next iteration of an enclosing **repeat**). The goal of the algorithm is to produce understandable programs, rather than to avoid the use of **goto** statements entirely. **Goto** statements are generated when there is no better way to describe the flow of control.

Repeat and **if then else** statements are selected as target language constructs for the algorithm because they are sufficient to show the organization of the program into loops and branches to alternative blocks of code. **Break** and **next** statements are used because they are easily generated by the algorithm and contain more information than **goto** statements. The algorithm can be easily extended to produce other constructs, such as **while**, **until**, or the constructs described in [16]. This paper concentrates on finding the basic structure of a program rather than on producing the syntax desired for a particular implementation of the algorithm.

A preliminary version of this paper was presented at the Third ACM Symposium on Principles of Programming Languages, Atlanta, Georgia, January 19-21, 1976

Author's address. Bell Laboratories, Murray Hill, New Jersey 07974

The author provided camera-ready copy for this paper using EQN | TROFF [14] on UNIX [20].

A number of techniques for eliminating **goto** statements from programs have been previously published [2,7-9,15,17,19]. These include adding control variables, copying code, creating subroutines, and adding extra levels of **repeat** statements in conjunction with multilevel **break** statements. Each of these methods may be appropriate in some cases. However, these techniques do not necessarily produce clear flow of control [16]. Rather than try to determine when these techniques are appropriate, the algorithm of this paper does not use them.

Instead, the structuring algorithm is based on some principles about how **repeat** and **if then else** statements should be used for best readability. The structuring algorithm and these principles evolved together. That is, applying earlier versions of the algorithm led to refinements of the principles which led in turn to refinements in the algorithm. The principles are discussed at length in [4]. A program which satisfies the principles is called *properly nested*. In a properly nested program, **repeats** reflect iteration in the program, and **if then else** statements reflect branching and merging of control in a reasonable way. The algorithm transforms a flowgraph into a properly nested program, in which the predicates and straight line code statements are the same as those of the flowgraph in both number and execution order. In general, the program may contain **goto** statements. However, the **goto** statements occur only where no other available control construct describes the flow of control.

Section 2 introduces flowgraphs and a simple structured language SL. Some simple ideas concerning the use of **repeat** and **if then else** statements are discussed in Section 3 as motivation for the algorithm. Section 4 describes the algorithm as restricted to "reducible" flowgraphs, i.e. flowgraphs in which each loop can be entered in only one place. The algorithm is extended in Section 5 to include irreducible flowgraphs. Section 6 proves that the algorithm produces a properly nested program. Section 7 applies some results from [4] concerning properly nested programs to show that any properly nested program for a flowgraph must be similar in form to the one generated by the algorithm. Moreover, if a flowgraph has a properly nested program with no **goto** statements, the algorithm generates one.

Section 8 discusses briefly an implementation of the algorithm in a program called STRUCT [3], which translates Fortran programs into RATFOR [13], an extended Fortran language which includes constructs such as **if then else** and **while**. The structured programs generated by STRUCT are much more readable than their Fortran counterparts. It is usually not obvious that they are mechanically generated, since the structuring principles cause them to imitate common programming practice. The structured programs usually contain few **goto** statements. An example of a program structured by STRUCT is included in Appendix B.

De Balbine [5,6] has written a program called the "structuring engine," which also claims to structure Fortran. His algorithm has not been published for comparison with the algorithm of this paper. However, the published output from the structuring engine appears to follow some of the same basic principles as the algorithm of this paper. A major difference is that the structuring engine avoids **goto** statements by creating a kind of argumentless subroutine. It is not clear from the published examples that the artificially created subroutines improve readability.

The structuring algorithm presented in this paper is proposed as a tool for the maintenance of Fortran programs. One of the problems in dealing with Fortran programs is that the lack of convenient control structures makes programs hard to understand Extended Fortran languages such as RATFOR have been developed so that new programs may be written using convenient control structures and translated by a preprocessor to Fortran for compilation. But many existing programs were written in Fortran without the benefit of preprocessors. These programs become dramatically more understandable when they are structured mechanically. Therefore, modification and debugging of these programs is facilitated by structuring.

2. *Definitions*

This section defines a simple structured language SL, the execution order of SL programs, and flowgraphs.

SL contains optionally labeled statements of the following forms:

(1) straight line code (slc) statements (i.e. assignment, read, write, etc.),

(2) **stop**,

(3) **goto** L, where L is a label,

(4) **if (p) then** {S1} **else** {S2}, where S1 and S2 are (possibly null) sequences of optionally labeled SL statements, and **p** is a predicate,

(5) **repeat** {S}, where S is a non-null sequence of optionally labeled SL statements,

(6) **break(i)**, where **i** is a positive integer,[1]

(7) **next(i)**, where **i** is a positive integer.[1]

An SL program is a nonnull sequence of SL statements, such that each label in a **goto** statement labels exactly one statement in the program. Henceforth, *program* means SL program, except where otherwise noted.

Goto, **next(i)**, **stop**, and **break(i)** statements are referred to as *branching* statements; other statements are *nonbranching* statements. Associated with a program is an "exit" which is reached when a **stop** is executed, or when control reaches the bottom of the program. Two statements are *at the same level of nesting* if neither is enclosed in another statement, or if the same statement is the smallest statement enclosing each one. The exit of the program is outside all levels of nesting by definition. The innermost **repeat** (if any) enclosing a statement is its first enclosing **repeat**. For $i>1$, the ith enclosing **repeat** is the **repeat** (if any) enclosing the $(i-1)$-st enclosing **repeat**.

Statements of types 1-4 are interpreted in the standard way. **Repeat** {S} causes the sequence S to be iterated until a **stop** is executed, or until a **goto**, **break(i)**, or **next(i)** (**i** greater than **1**) causes a jump out of the **repeat** statement. **Break(i)** causes a jump to the statement following the **i**th enclosing **repeat** statement. **Next(i)** causes a jump to the next iteration of the **i**th enclosing **repeat** statement, that is, it is equivalent to **goto** L, where L is a label added (if necessary) to the **i**th enclosing **repeat**.

For simplicity, no **elseless if then** statement is provided, but its equivalent is obtained by a null **else** clause. Also, more complex constructs such as **while** and **until** are not provided since they can be expressed in terms of **repeat**, **if then else**, and **break**. For simplicity, **return** is not included; it may be treated like **stop** (but separately) during structuring.

A *flowgraph* is a directed graph with labeled nodes and arcs representing flow of control between nodes. Each node is either a straight line code (slc) node with one outarc, an **exit** node with no outarcs, or an **if** node, with a "true" outarc and a "false" outarc. A flowgraph has exactly one **exit** node, and there is a path to it from every node in the flowgraph. One node of the flowgraph is designated as the **start** node.

An SL program is *flowgraphable* if every loop (created by means of **repeat** and/or **goto** statements) includes either an **slc** or **if** statement. The possible computations performed by a flowgraphable program *P* are determined by the flow of control between **slc** and **if** statements and the exit of the program. This flow of control is described by a flowgraph *COMPUTE*(*P*) which is obtained as follows from *P*. For

[1]Multilevel **break** and **next** statements are included because they happen to be easily identified by the structuring algorithm They are not fundamental to the algorithm; the algorithm has been implemented in STRUCT with nice results using only single level **break** and **next** statements

each **slc** or **if** statement in P, there is a corresponding node in $COMPUTE(P)$. In addition, there is a single **exit** node which represents the exit of the program. There is an arc from node p to node q if control can pass from statement p to q without passing through any other **if** or **slc** statement. The **start** node of $COMPUTE(P)$ is the node which corresponds to the first **slc** or **if** statement reached upon executing the program, or the **exit** node if the exit of the program is reached before any **slc** or **if** statement is executed.

Two flowgraphable SL programs P_1 and P_2 are *equivalent* if $COMPUTE(P_1) = COMPUTE(P_2)$. Note that this is a stronger statement than merely requiring that the set of execution paths be the same. If one program has two copies of an **slc** statement while the other has only one, the programs may have identical sets of execution paths but are not equivalent by this definition. This definition of equivalence was chosen because the algorithm of this paper does not copy code.

A flowgraphable program P is a *structuring* of a flowgraph G if $G = COMPUTE(P)$.

A *loop* in a flowgraph G is a path which begins and ends at the same node and includes at least one arc. A *cycle* is a loop in which only the first node (which is the same as the last node) occurs twice.

Finally, a flowgraph is *reducible* if every cycle in it can be entered in exactly one place, i.e. there is a node p in each cycle such that every path from the **start** node must reach p before any other node in the cycle. A program P is *reducible* if $COMPUTE(P)$ is reducible. The structuring of reducible flowgraphs is discussed in Sections 3 and 4. Irreducible flowgraphs (i.e. flowgraphs which are not reducible) are treated in Section 5.

3. *Basic Requirements for the Algorithm*

The goal of this paper is to present an algorithm which generates readable structured programs from flowgraphs. The first step in developing the algorithm is to determine some basic properties a program must have to be readable. To keep the discussion simple, this section assumes that the programs of interest are reducible. The ideas discussed in this section are generalized in [4] as a set of principles for the use of **repeat** and **if then else** statements in (reducible or irreducible) programs. These principles are listed in their general form in Section 6.

First, **repeat** statements must reflect iteration in the program Obviously, programs such as the following should not be allowed.

```
repeat
    { s = 1
      stop
    }
```

But the following program is also poor because it gives the impression that the whole program can be iterated.

```
repeat
    { if (p) then
           { code segment
             stop
           }
      else
           { x = f(x)
           }
    }
```

The iteration in this program is better represented by the following version.

```
repeat
     { if (p) then { break(1) }
      else { x = f(x) }
     }
code segment
stop
```

Thus a basic requirement is that every statement within a **repeat** should lead to an iteration of the **repeat**.

Each **if then else** statement should reflect branching and merging of flow of control in the program. In particular, a **goto** statement should not jump into the middle of a **then** or **else** clause. Also, a statement should appear within a clause if it can be reached only from the clause and is within the innermost **repeat** (if any) containing the clause. In particular, of the following examples, (a) is reasonable, while (b) and (c) are not.

```
(a)    if (p) then { j = 1 }
       else { j = 2 }
       y = f(j)
       stop

(b)    if (p) then
            { j = 1
             goto 10
            }
       else
            { j = 2
         10  y = f(j)
            }
       stop

(c)    if (p) then {}
       else
            { j = 2
             goto 10
            }
       j = 1
  10   y = f(j)
       stop
```

The above discussion is sufficient as a base from which to develop the algorithm. Obviously, the algorithm makes further decisions about the use of control constructs.

4. *The Structuring Algorithm*

This section presents the algorithm for structuring flowgraphs. To keep discussion simple, this section assumes that the input flowgraph is reducible. Section 5 discusses adaptations to the algorithm which enable it to handle irreducible programs in a reasonable way.

4.1. Finding Structure in a Flowgraph. The input to the structuring algorithm is a flowgraph G in which every node is reached by a path from the **start** node. The first step in structuring G is to locate the loops in G. Loops are identified from a spanning tree which is constructed by a depth-first search [12]. The depth-first search proceeds as follows.

Begin by visiting the **start** node of G and setting NUM to the number of nodes in the flowgraph When visiting a node m, do the following If node m has an arc to a node p not already visited, make p a child of m in the spanning tree, and visit p next Otherwise, number m with NUM, decrement NUM by 1, and return to visit the parent of m (if it exists) again

A *back* arc is an arc from a node to itself or from a descendant to an ancestor in the spanning tree; other arcs are *forward* arcs. If (p,q) is a back arc and $p \neq q$, there is a path from q to p consisting of edges from parent to child in the tree. This path and the back arc form a cycle in G. Each node entered by one or more back arcs will become the first statement within a **repeat** in the final program.

Let L be a list of the nodes of the graph ordered by the numbering assigned during the depth-first search. This list will be used to ensure that all **gotos** in the final program flow downward on the page. Note that an arc (p,q) is a back arc if and only if q appears before p in L. Also, if (p,q) is a back arc, there is a path from q to p which includes only nodes between q and p in L.

Example. A flowgraph and the numbering of nodes produced by a depth-first search are illustrated in Figure 1.

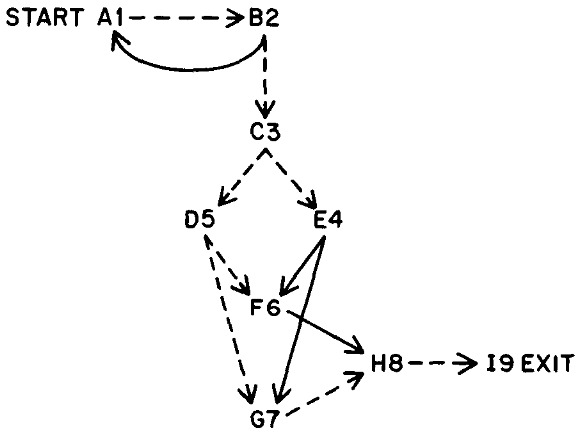

FIG 1 A flowgraph G and the numbering associated with the depth-first search, which traverses "true" (left) arcs before "false" (right) arcs Dashes indicate arcs in the spanning tree The list $L=A,B,C,E,D, F,G,H,I$

At this point, the nodes which will become the first statements within **repeats** have been determined. In particular, they are the nodes entered by back arcs. This information is encoded in the flowgraph as follows. For each node n entered by a back arc, add a single **repeat** node p. Replace each arc (q,n) by an arc (q,p), and add an arc (p,n). Insert the **repeat** node p immediately before n in L. Call the new graph the *extension* of G, $EXT(G)$. The **start** node of $EXT(G)$ is defined to be the first node of L, which may be a **repeat** node rather than the **start** node of G.

Note that the addition of the new nodes does not change the ordering of the nodes already in L. The definition of back arc is extended to $EXT(G)$ by defining an arc (p,q) in $EXT(G)$ to be a back arc if q precedes p in L.

Example. Figure 2 shows the graph $EXT(G)$ generated from G of Figure 1.

A **repeat** node p is the *head* of all loops and cycles which include p but no nodes preceding p in L. If p is the head of a loop containing q, and $p \neq q$, q is in a *loop tail* headed by p. In the final program the **repeat** statement corresponding to p will contain the statements corresponding to nodes in loop tails headed by p. For each node q, the algorithm determines $HEAD(q)$, which is the **repeat** node which will correspond to the smallest **repeat** enclosing q in the final program. In particular, of the **repeat** nodes which are heads of loops containing q, $HEAD(q)$ is the closest one preceding q in L. If no such node exists, $HEAD(q)$ is undefined. For convenience, if $HEAD(q)$ is undefined, all nodes are said to be in the "loop tail" headed by $HEAD(q)$, and this "loop tail" is considered to be the entire program. Also, if

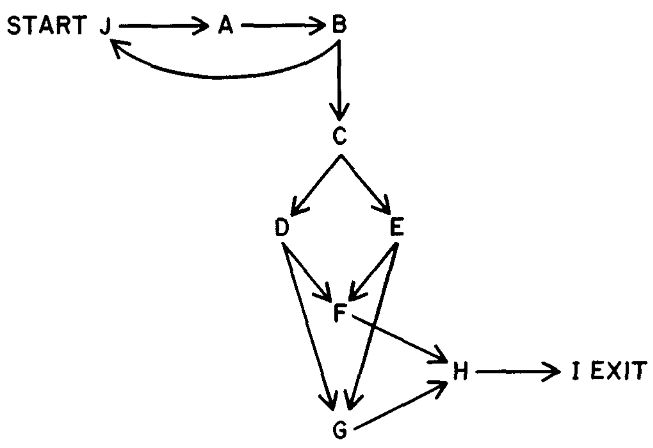

FIG 2 The graph *EXT(G)* generated from the graph *G* of Figure 1 The list *L=J,A,B,C, E,D,F,G,H,I*

HEAD(*p*) and *HEAD*(*q*) are undefined, *HEAD*(*p*)= *HEAD*(*q*) by definition. Note that for a **repeat** node *p*, *HEAD*(*p*) is either a different **repeat** node or is undefined. The **repeat** corresponding to *p* will be nested within the **repeat** corresponding to *HEAD*(*p*) in the final program.

Example. For the graph of Figure 2, *HEAD*(*A*)=*HEAD*(*B*)=*J*. For all other nodes *p*, *HEAD*(*p*) is undefined.

To produce code from *EXT*(*G*), the algorithm needs to know which statements are reachable from which others. For example, for the graph of Figure 2, it needs to know that nodes *F* and *G* can be reached through both branches of *C*, so that neither *F* nor *G* should be placed within a clause of *C*.

Such branching and merging of control can be described by *dominators* in the flowgraph [1]. Node *p dominates* node *q* if every path from the **start** node to node *q* must pass through node *p*. Node *p* is the *immediate dominator* of node *q* if no other dominator of *q* lies "closer" to *q* (that is, if every dominator of *q* other than *p* also dominates *p*). Every node in the flowgraph except the **start** node is dominated by at least one node, the **start** node. Moreover, every node except the **start** node has an immediate dominator. Let *DOM*(*p*) denote the immediate dominator of *p*.

Example. The dominators of the nodes in the graph of Figure 2 are as follows: *DOM*(*J*) is undefined, *DOM*(*A*)=*J*, *DOM*(*B*)=*A*, *DOM*(*C*)=*B*, *DOM*(*I*)=*H*, and the immediate dominator of the other nodes is *C*.

For each node *p*, *HEAD* and *DOM* are used to obtain a set *FOLLOW*(*p*) specifying nodes which belong "after" *p* at the same level of nesting as *p*. For each **if** node *p*, define

$$FOLLOW(p)=\{q \mid q \text{ is entered by 2 or more forward arcs, } p=DOM(q), \text{ and } HEAD(p)=HEAD(q)\}$$

For each **repeat** node p, define

$$FOLLOW(p)=\{q \mid HEAD(q)=HEAD(p) \text{ and } DOM(q) \text{ is in a loop tail headed by } p\}.$$

For each **slc** node *p*, define

$$FOLLOW(p)=\{q \mid HEAD(q)=HEAD(p) \text{ and } p=DOM(q)\}.$$

Note that the sets *FOLLOW*(p) are pairwise disjoint, for all nodes p.

Every node is in a *FOLLOW* set except for the nodes which will correspond to the first statements at each level of nesting. Intuitively, *FOLLOW*(p) is the set of nonbranching statements reachable from p which must follow p at the same level of nesting as p.

Example. For the graph of Figure 2, *FOLLOW* sets are as follows. *FOLLOW*$(J) = \{C\}$, *FOLLOW*$(A) = \{B\}$, *FOLLOW*$(C) = \{F,G,H\}$, *FOLLOW*$(H) = \{I\}$, and all other *FOLLOW* sets are empty.

A recursive procedure **getform** generates the basic form of the structured program. That is, **getform** determines the nesting and order of nonbranching statements. Branching statements are added later. **Getform** is called on the **start** node of *EXT*(G).

```
getform(n)
  {  if (n is an slc node) then
          { print the straight line code }
     else if (n is a repeat node with arc to node q) then
          { print("repeat{")
            call test(q)
            print("}")
          }
     else if (n is an if node with predicate r and a "true" arc to
       node p and a "false" arc to node q) then
          { print("if (r) then {")
            if (the "true" arc is a forward arc) then { call test(p) }
            print("} else {")
            if (the "false" arc is a forward arc) then { call test(q) }
            print("}")
          }
     for each member q of FOLLOW(n) in order of appearance in L
          { call getform(q) }
  }

test(q)
  {  if (q is not in any FOLLOW set) then
          { call getform(q) }
  }
```

Since the *FOLLOW* sets are pairwise disjoint, **getform** is called exactly once on each node in *EXT*(G) The output from **getform** is called *PF*(G), the "program form" generated from G.

Example. **getform** generates the following from the graph of Figure 2:

```
repeat
     { A
       if (B) then {}
       else {}
     }
if (C) then
     { if (D) then {}
       else {}
     }
else
     { if (E) then {}
       else {}
     }
F
G
H
```

The above procedure is responsible for ensuring that the final program has the properties discussed in the preceding section.

4.2. ADDING BRANCHING STATEMENTS. The second part of the algorithm adds branching statements to $PF(G)$ to represent the flow of control in $EXT(G)$. Some decisions must be made at this point as to how to use branching statements. Obviously, a branching statement should not be used if deleting it does not alter the flow of control. Branching statements are not needed in certain places in $PF(G)$ because the flow of control is already correct. The first part of the algorithm guarantees that the first statement within a **repeat** is the node entered by an arc from the **repeat**, and that the first statement within a clause of an **if** statement is the node entered by the appropriate arc of the **if**. Thus, it remains to add branching statements to correct the flow of control out of some **slc** statements and empty **if** clauses

The simplest way of correcting the flow of control in $PF(G)$ is to find every **slc** statement and every empty **if** clause which passes control to the wrong point, and add appropriate branching statements to correct the flow of control. However, it is desirable to make the algorithm somewhat more sophisticated, so that it can generate code such as the following.

```
if (p) then { j = 1 }
else { j = 2 }
break(1)
```

The simple strategy mentioned above would put a **break** statement in each clause, rather than the single **break** statement after the **if** statement. Instead, a reasonable convention is to place a branching statement after any statement p such that control can pass out of the statement to exactly one nonbranching statement q, and q is not reached automatically without a branching statement. Two definitions are needed to identify such statements.

First, for each node p in $EXT(G)$, define $REACH(p)$ to be the set consisting of all nodes q entered by arcs from p or from nodes corresponding to statements nested within p, such that q does not correspond to p or to a statement nested within p. (For this definition, recall that the **exit** node corresponds to the exit of the program which has been defined to be outside all levels of nesting of statements.)

Example. For the graph G of Figure 2 and the code $PF(G)$ of the preceding example, $REACH$ sets are defined as follows: $REACH(A)=\{B\}$, $REACH(B)=\{J,C\}$, $REACH(C)=\{F,G\}$, $REACH(D)=REACH(E)=\{F,G\}$, $REACH(F) = REACH(G) =\{H\}$, $REACH(H)=\{I\}$, $REACH(I)=\emptyset$, $REACH(J)= \{C\}$.

Second, define a branching statement to be *redundant* if it passes control to the same statement which would be reached from that point if the branching statement were deleted. Thus, for each statement p in $PF(G)$ such that $REACH(p)$ contains exactly one node r, p must be followed by a branching statement passing control to r unless that statement would be redundant.

Branching statements are added to the program recursively from outer levels of nesting to inner levels by calling the procedure **addbranch**$(PF(G),G)$.

```
addbranch(F,G)          /* F is a program minus branching statements, G is a flowgraph */
  { compute REACH(p) for every node p
    if (F contains no nonbranching statements) then { add a statement "stop" to F }
    else
        { let p denote the first nonbranching statement in F
          call fixcontrol(p)
        }
  }
```

```
fixcontrol(p)
  {  if (REACH(p) contains a single node r) then
          { add the statement choosebranch(p,r) after r unless this statement would be redundant }
     if (p is a repeat statement whose body begins with a statement q ) then { call fixcontrol(q) }
     else if (p is an if statement) then
          { for each clause of p
               { if (the clause is empty) then
                    { add the statement choosebranch(p,r) to the clause
                              unless this statement would be redundant
                    }
                 else if (the clause begins with a statement q) then { call fixcontrol(q) }
               }
     if (p is not the last nonbranching statement at its level of nesting) then
          { call fixcontrol(q), where q is the next nonbranching statement at  this level of nesting }
  }

choosebranch(p,r)  /* select a branching statement to pass control from p to r */
  {  let n be the number of repeat statements enclosing p
     for each i from 1 to n
          { if (placing "break(i)" after p would pass control to r) then { return("break(i)") } }
     for each i from 1 to n
          { if (r is the ith repeat enclosing p) then { return("next(i)") } }
     if (r is the exit node) then { return("stop") }
     if (r has no label) then
          { label r with a  label L distinct from all labels already in the program }
     return("goto L"), where L is the label of r
  }
```

Note that the procedure **choosebranch** imposes a precedence order upon the possible branching statements which might be used at each point. This order has the desirable property that it ensures that every branching statement is reachable (see Lemma 3). This is not true for all precedence orders. For example, if **break(i)** is preferred to **break(j)**, $i > j$, unreachable **break** statements may be generated. Therefore, the algorithm allows **break** statements to jump to other **break** statements.

When the above procedure is applied to the program form $PF(G)$ generated by the first part of the algorithm, the resulting program is called $ALG(G)$.

Example. For the flowgraph G of Figure 1, $ALG(G)$ is the following.

```
     repeat
         { A
           if (B) then {}
           else { break(1) }
         }
     if (C) then
         { if (D) then { goto 10 }
           else {}
         }
     else
         { if (E) then {}
           else { goto 10 }
         }
     F
     goto 20
10   G
20   H
     stop
```

This example is not an impressive example of a structured program. It was chosen because its peculiarities illustrate most parts of the structuring algorithm.

5. *Structuring Irreducible Flowgraphs*

Most programs are reducible. However, the algorithm must handle irreducible programs if it is to be perfectly general. Since the decision has been made in this paper not to copy code, a program generated from an irreducible flowgraph must contain a **goto** into one or more **repeat** statements. Therefore, the problem is to find a reasonable way to treat **goto** statements which jump into **repeat** statements. The choice made here is to try to make the program well structured at a local level. In particular, the algorithm pretends that each loop is entered at only one point, structures the program accordingly, and adjusts the final flow of control to put the jumps into loops back in. Thus, the algorithm can generate a program of the following form.

```
if (p) then { goto 10 }
else {}
repeat
    { if (q) then { j = 1 }
     else
         {
      10  j = 2
         }
     if (r) then {stop }
     else {}
    }
```

This program contains an ugly jump into an **else** clause. However, this program is better structured within the **repeat** than the following program, in which the jump into the **else** clause is avoided at the expense of an extra **goto** statement.

```
if (p) then {goto 10 }
else {}
repeat
    { if (q) then
        {j = 1
         goto 20
        }
     else {}
 10  j = 2
 20  if (r) then {stop }
     else {}
    }
```

A large example of an irreducible program generated by STRUCT appears in Appendix B.

The way the algorithm "pretends" that the flowgraph is reducible is to construct a reducible flowgraph $REDUCE(EXT(G))$ from the flowgraph $EXT(G)$, and calculate dominators from this reducible graph rather than from $EXT(G)$.

Intuitively, the algorithm pretends that each arc entering a loop at a point other than its head enters the head instead. Let $REDUCE(EXT(G))$ be a flowgraph obtained as follows from $EXT(G)$. If (p,q) is an arc, $p \neq HEAD(q)$, and p is not in a loop tail headed by $HEAD(q)$, the arc (p,q) is replaced in $REDUCE(EXT(G))$ by an arc (p,r), where r is the first **repeat** node in L which is the head of a loop containing q but not the head of any loop containing p. The resulting graph $REDUCE(EXT(G))$ is reducible. Node p *R-dominates* node q if p dominates q in $REDUCE(EXT(G))$. Note that a **repeat** node p R-dominates each node in a loop tail headed by p. For each node p, $RDOM(p)$ is defined to be the immediate dominator of p in the graph $REDUCE(EXT(G))$. Define an arc (p,q) in $REDUCE(EXT(G))$ to be a forward arc if $p<q$.

The description of the algorithm in Section 4 must be modified for the general case by using R-dominators rather than dominators in $EXT(G)$. Also, in defining $FOLLOW(p)$ for an **if** node p, the reference to forward arcs must be to forward arcs in $REDUCE(EXT(G))$ rather than to forward arcs in $EXT(G)$. Note that if $EXT(G)$ is reducible, $REDUCE(EXT(G))=EXT(G)$. Therefore, this general form of the algorithm behaves the same on reducible flowgraphs as the basic algorithm of the preceding section.

6. *Properties of $ALG(G)$*

This section proves that the general algorithm of Section 5 produces a structuring of G, and that this program $ALG(G)$ has reasonable properties.

THEOREM 1. *$ALG(G)$ is a structuring of G.*

PROOF. The procedure **fixcontrol** guarantees that flow of control in $ALG(G)$ between **if**, **slc**, and **repeat** statements and the exit of the program corresponds to $EXT(G)$. Since every loop in $EXT(G)$ includes an **slc** or **if** statement, $ALG(G)$ is flowgraphable and $COMPUTE(ALG(G))$ is defined. The flow of control between **slc** and **if** statements and the exit of $ALG(G)$ is obtained from $EXT(G)$ by applying the inverse of the transformation which generated $EXT(G)$ from G. Thus, $G=COMPUTE(ALG(G))$, and $ALG(G)$ is a structuring of G. □

Section 3 discussed how **repeat** and **if then else** statements should be used in reducible programs. These ideas are generalized in [4] as a set of principles for the use of **repeat** and **if then else** statements in a program (reducible or irreducible). It will be shown that $ALG(G)$ satisfies these principles.

The principles are divided into two parts: proper use of **repeat** statements, and proper use of **if then else** statements.

A program *uses* **repeat** *statements properly* if it has the following properties:

(1) If a nonbranching statement q is nested within a **repeat** statement p, there is an execution path which passes from p to q and back to p without leaving the body of p.

(2) The first statement within a **repeat** is an **slc** or **if** statement and is reached only from the **repeat**.

(3) A **repeat** statement can be entered without first entering its body.

(4) Control can pass to a lexically preceding part of the program only from within the body of a **repeat** to the **repeat**.

A program P *uses* **if** *statements properly* if the following conditions are satisfied:

(1) if **goto** L occurs and is not within a **then** or **else** clause containing L, then the **goto** is also outside the innermost **repeat** containing L.

(2) If a nonbranching statement r in P is nested within the innermost **repeat** containing an **if** statement p, and is not within the **then** (**else**) clause of p, then r is reachable either from the "false" ("true") branch of p or from a nonbranching statement not equal to p which is nested within the **repeat** but not nested within the clause or within r.

A program is *well formed* if it is flowgraphable and every nonbranching statement is accessible from the start of the program. A program is *properly nested* if it is well formed and uses both **repeat** and **if then else** statements properly.

In order to prove that $ALG(G)$ is properly nested, two technical lemmas are needed. Their proofs appear in Appendix A. The first lemma describes the behavior of the procedure **getform**. If **getform**(q) is called during the recursive execution of **getform**(p), p is called a *g-ancestor* of q, and q is called a *g-descendant* of p.

LEMMA 1. *A node q is a g-descendant of a node p if and only if p R-dominates q and q is in a loop tail headed by $HEAD(p)$.*

LEMMA 2. *If (r,s) is a back arc in $EXT(G)$, then s is a* **repeat** *node and r is nested within s in $ALG(G)$. If (r,s) is a forward arc in $EXT(G)$, then either s is nested within r or s is after r in $ALG(G)$.*

THEOREM 2. *$ALG(G)$ is properly nested.*

PROOF. From the proof of Theorem 1, $ALG(G)$ is a structuring of G. Since the flow of control between nonbranching statements and the exit of the program is represented by $EXT(G)$, and every node in G is accessible from the **start** node, $ALG(G)$ is well formed.

Next, it is shown that $ALG(G)$ uses **repeat** statements properly.

(1) First, it is shown that a node p is nested within a **repeat** statement r if and only if p is in a loop tail headed by r.

> Let q be the node entered by an arc from r in $EXT(G)$. It is straightforward to show that q is in a loop tail headed by r and q is nested within r.
>
> Now consider any node p, $p \neq q$ and $p \neq r$. If p is nested within r, p is a g-descendant of q. By Lemma 1, p is in a loop tail headed by $HEAD(q)$. On the other hand, if p is in a loop tail headed by $HEAD(q) = r$, r R-dominates p. Since r R-dominates p and the only outarc of r enters q, q R-dominates p. By Lemma 1, p is a g-descendant of q, and p is nested within r.

Now, control can never jump out of r and back in without passing through a nonbranching statement. Thus, for each statement within the body of r, there is an execution path from r to the statement and back to r, such that the path never passes outside of r.

(2) As a result of the construction of $EXT(G)$ from G, the first statement within a **repeat** is an **slc** or **if** statement and is reached only from the **repeat**.

(3) From the depth-first search and the construction of $EXT(G)$, it follows that a **repeat** node can be reached in $EXT(G)$ without passing through any node in a loop tail headed by the **repeat**. By part (1) of this proof, the **repeat** can be entered without first passing through any nonbranching statement nested within the **repeat**. Since a **goto** can jump only to a nonbranching statement, the **repeat** can be entered without first entering the body of the **repeat**.

(4) If control passes upward in the program to a statement other than an enclosing **repeat**, it does so via a **goto** statement. So suppose a **goto** statement s jumps to a statement r above s or to a statement r enclosing s. By definition of **fixcontrol**, r is a nonbranching statement.

By (2), s is not the first statement within a **repeat**. Suppose s is the first statement within a clause of an **if** statement p. Then there is an arc from p to r in $EXT(G)$. By Lemma 2, this arc is a back arc and r is a **repeat** enclosing p and s.

Suppose s follows a statement p at the same level of nesting. Then r is in $REACH(p)$ and there is an arc from p or a node nested within p to r. By definition of $REACH$ sets, r is not nested within p and $r \neq p$. By Lemma 2, this arc is a back arc and r is a **repeat** enclosing p and s.

Thus, control can flow to a lexically preceding point in the program only to an enclosing **repeat**.

Next, it is shown that $ALG(G)$ uses **if** statements properly.

(5) Let p be an **if** statement with a "true" ("false") arc to q. Suppose the **then** (**else**) clause contains a statement r labeled with **L**. Suppose **goto L** occurs within the innermost **repeat** enclosing p but outside this clause. The **goto** corresponds to an arc (u,r) such that u is outside the clause but within the innermost **repeat** enclosing p. By part (1) of this proof, u is in a loop tail headed by $HEAD(p)$. Lemma 2 implies that the arc is not a back arc. If $r=q$, either the arc (p,q) is a back arc or r is in a $FOLLOW$ set. In the former case, Lemma 2 implies that p is nested within r, and

in the latter case **getform** guarantees that r is not nested within the clause. So $r \neq q$. Now, u is not a g-descendant of q. By Lemma 1, u is not R-dominated by q. Since u is in a loop tail headed by $HEAD(q)=HEAD(p)$, neither is r. By Lemma 1, r is not a g-descendant of q, contradicting the fact that r is in the clause.

(6) Let p be an **if** statement with a "true" ("false") arc to q. If q is not in the **then** (**else**) clause, **getform** implies that q is in a *FOLLOW* set or the arc is a back arc, and that the clause contains no nonbranching statements. Thus, either q is not in the innermost **repeat** enclosing p, or q is reached from the "false" ("true") branch of p, or q is reached from another nonbranching statement not in the clause.

Now consider a node $r \neq q$. Suppose r is within the innermost **repeat** enclosing p but not within the **then** (**else**) clause. If q is not within the clause, the clause contains no nonbranching statements and r is reached from a statement outside the clause or from the "false" ("true") arc of p. So assume q is within the clause. By part (1) of this proof and Lemma 1, r is not R-dominated by q. Since every nonbranching statement within the clause is R-dominated by q, r is reached from the "false" ("true") branch of p or from a nonbranching statement which is within the innermost **repeat** enclosing p but is not within the clause. □

Corollary. *If* **goto L** *occurs in $ALG(G)$, then the statement labeled* **L** *occurs after the* **goto** *statement.*

Proof. The corollary follows from part (4) of the above proof and the fact that the algorithm generates a **next** in preference to a **goto** statement which jumps to an enclosing **repeat**. □

Finally, it is shown that every statement in $ALG(G)$ is reachable. This justifies the comments in Section 4 after the procedure **choosebranch**.

Lemma 3. *Every statement is reachable from the start of the $ALG(G)$.*

Proof. Since G is a flowgraph in which every node is entered by a path from the **start** node, and $G=COMPUTE(ALG(G))$, every nonbranching statement in $ALG(G)$ is reachable from the start of the program. It will be shown that every branching statement is reachable from the start of the program, from a nonbranching statement, or from a branching statement lexically preceding it in the program. Consequently, every statement in the program is reachable.

Let p be any branching statement within $ALG(G)$. If p is the first statement in the program, it is reachable from the start of the program. If p is the first statement within any other level of nesting, it is reachable from the statement enclosing it. Otherwise, p follows a nonbranching statement. The remainder of the proof shows that every branching statement which follows a nonbranching statement r is reachable from r or from a statement nested within r.

For each statement p, define $LEX(p)$ as follows. If p is followed at the same level of nesting by a statement q, $q=LEX(p)$. Otherwise, if p is the last statement within a clause of an **if** statement q, $LEX(p)=LEX(q)$. Otherwise, if p is the last statement within a **repeat** statement q, $LEX(p)=q$. If p is the last statement in the program $LEX(p)$ is the exit of the program.

Define the target of a branching statement to be the first nonbranching statement reached upon executing it.

Next, it is shown that if r is an **if** statement and $\{q\}=REACH(r)$, then $\{q\}=REACH(s)$, where s is the last nonbranching statement to occur at the outermost level within r. Since P is well formed, the exit of the program is reachable from s, and $REACH(s)$ is not empty. Moreover, every member of $REACH(s)$ is outside r (for either it is below both s and r, or by Theorem 2, part(2) it is a **repeat** containing both s and r). Therefore, $REACH(s)=REACH(r)$.

Now, it is shown by induction that if a branching statement p is $LEX(r)$ for a nonbranching statement r, and the target of p is is the unique member of $REACH(r)$, then p is reached from r or from a statement nested within r. The induction is based on the number of levels of nesting within p. For the basis, note that if r is an **slc** statement, then $LEX(r)$ is reached from r. Suppose that the assertion holds whenever r has at most k levels of nesting. Now suppose that r has $k+1$ levels. If r is a **repeat**, p is reached by a **break** from within r because its target is in $REACH(r)$ and lower levels of **break** statements are generated in preference to higher levels. So suppose r is an **if** statement. Consider the **else** clause of r. If the clause is null, $LEX(r)$ is reached from r. Otherwise, let s be the last statement within the clause. If s is a branching statement, its target must be in $REACH(r)$, implying that its target is the same as the target of p, and s is redundant. Therefore, s is a nonbranching statement. Since $REACH(s)$ is nonempty, $REACH(s)= REACH(r)$. Also, $LEX(s)= LEX(r) =p$. By the induction hypothesis, p is reached from r or from within r.

Finally, suppose a branching statement p follows a nonbranching statement r at the same level of nesting. By the definition of **fixcontrol**, the target of p is the unique member of $REACH(r)$. By the preceding paragraph, p is reached from r or from a statement nested within r. □

7. *Properties of Properly Nested Programs*

This section quotes some results concerning proper nesting which show that any properly nested structuring of a flowgraph G must be similar to $ALG(G)$. Moreover, if G has a properly nested structuring with no **goto** statements, $ALG(G)$ has no **goto** statements.

In a program with proper nesting, the nesting of statements can be characterized as follows [4].

THEOREM 3[4]. *If P_1 and P_2 are equivalent properly nested reducible programs, then they are identical in the number of occurrences of each nonbranching statement and in how the nonbranching statements are nested within each other, but not necessarily in the order of nonbranching statements at each level of nesting.*

Note that this theorem does not state that P_1 and P_2 are identical in the *order* of nonbranching statements at each level of nesting. In fact, the order of statements is not uniquely determined. For example, consider the following code.

```
     if (p) then
          { if (q) then { goto 10 }
            else {}
          }
     else
          { if (r) then { goto 10 }
            else {}
          }
     x = 1
     goto 20
10   x = 2
20   y = f(x)
```

This segment could be rewritten by exchanging **x = 2** with **x = 1** and moving the **goto** statements to the **else** clauses.

However, there is no flexibility in order when no **goto** statements occur [4].

THEOREM 4[4]. *If P_1 and P_2 are equivalent properly nested programs with no* **goto** *statements, then P_1 and P_2 are identical in how nonbranching statements are nested and in the order of nonbranching statements at each level of nesting.*

The following theorem shows that $ALG(G)$ generates a properly nested program with no **goto** statements whenever this is possible.

THEOREM 5. *Let P be a properly nested program. The following statements are equivalent:*

(1) *P has an equivalent properly nested program with no* **goto** *statements.*

(2) *P is reducible, and for each* **repeat** *or* **if** *statement p, at most one statement outside of p but within the innermost* **repeat** *enclosing p can be reached from p or from within p.*

(3) *ALG(COMPUTE(P)) is a properly nested program with no* **goto** *statements equivalent to P.*

Proof. (1=>2) Let P' be a properly nested program equivalent to P with no **goto** statements. The lack of **goto** statements implies that P' is reducible.

Let p be any **repeat** or **if** statement. At most one statement within the innermost **repeat** (if any) enclosing p can be reached from p or from within p without a **goto**, since control must pass out of an **if** or **repeat** through the bottom.

(2=>3) For each nonbranching statement p, let $R(p)$ denote the unique nonbranching statement outside of p but within the innermost **repeat** enclosing p which can be reached from p or from within p, if such a statement exists. Apply the algorithm to $COMPUTE(P)$ to obtain a properly nested program P'. By Theorem 3, P_1 and P_2 are identical in the nesting of nonbranching statements but not necessarily in the ordering of statements at each level of nesting.

Suppose p is a nonbranching statement. It will be shown that if $FOLLOW(p)$ is nonempty, then $R(p)$ is defined and is the unique member of $FOLLOW(p)$.

> Let s be the least member of $FOLLOW(p)$. Since s is reachable, s is entered by a forward arc from a node t. Now, s is dominated by p and $HEAD(s)=HEAD(p)$. Therefore, either $t=p$ or t is dominated by p. Also, t is in a loop tail headed by $HEAD(p)$. Either $t=p$ or by Lemma 1, t is a g-descendant of p. By Lemma 2, t precedes s or t encloses s in the program. Thus, either $t=p$ or t is nested within p, implying $s=R(p)$
>
> Now, suppose that $FOLLOW(p)$ has another member q, $q \neq R(p)$. By definition, the immediate dominator of q is p or is nested within p. But $R(p)$ must be a closer dominator to q than p or any node nested within p. Therefore, no such q exists.

Define $LEX(p)$ as in the proof of Lemma 3. It is shown that $R(p)=LEX(p)$ whenever $R(p)$ is defined. The proof is by induction on the number of levels enclosing p. Assume that the assertion holds for all statements (if any) enclosing p.

> If p is the last nonbranching statement at its level of nesting, and $R(p)$ is defined, then $R(p)$ is below p within the innermost **repeat** (if any) enclosing p and is not the exit of the program. Therefore, p is enclosed by an **if** statement t such that $R(t)=R(p)$. By the induction hypothesis, $LEX(t)=R(t)$. Thus, $LEX(p)=LEX(t)=R(p)$.
>
> Otherwise, let q be the next nonbranching statement after p at the same level of nesting Since each $FOLLOW$ set has at most one member, q is in $FOLLOW(p)$, implying $q=R(p)$. Thus, no branching statement follows p and $q=LEX(p)$.

Now, suppose a **goto** statement g with target q occurs in the program. Either g follows a nonbranching statement p or is the first statement within a clause of an **if** statement p. If q is within the innermost **repeat** enclosing p, the above argument implies that $q=LEX(p)$ contradicting the existence of the **goto**. If q is not within the innermost **repeat** containing p, then $q=LEX(r)$ for some **repeat** r enclosing p Thus, **choosebranch** generates a **break** in preference to the **goto**, contradicting the existence of the **goto**. Therefore, no **goto** occurs in the program.

(3=>1) Trivial. □

The results in this section suggest that $ALG(G)$ cannot be greatly improved upon within the context of properly nested programs and the limitations imposed by the definition of structuring. The way in which branching statements are used could be modified without losing the desirable properties described in this section and Section 6. It is possible that restricted use of code copying, creating subroutines, or creation of control variables might improve upon $ALG(G)$ in some cases, if the definition of "structuring" is relaxed to allow these operations. Such extensions of the algorithm are left for further research.

8. *Applying the Algorithm*

The algorithm has been implemented in a program called STRUCT [3], which rewrites Fortran programs in RATFOR [13]. STRUCT consists of about 4000 lines of code in the programming language C [18] and runs on a PDP 11/45 under UNIX [20].

RATFOR has the following statement types in addition to Fortran statements: **repeat**, **repeat until**, **while**, **if else** (the keyword **then** is omitted), and single level **break** and **next** statements. The basic algorithm is extended in STRUCT to generate **while** loops and **elseless if** statements. Predicates are negated when necessary for the generation of **elseless if** statements. STRUCT keeps each comment with the following statement. Since RATFOR has only single-level **break** and **next** statements, STRUCT chooses its branching statements by a modified version of **choosebranch**. Appendix B contains an example of a Fortran program and the RATFOR program generated from it by STRUCT.

The mechanically structured versions of programs are easier to understand than their Fortran counterparts, sometimes dramatically so. Their natural appearance indicates that the structuring principles describe reasonable programming practices. The structured programs usually contain few **goto** statements. Of course, STRUCT does not improve programs; it merely displays their structure. A Fortran program with peculiar flow of control can have a structured version with many **goto** statements. A more extensive discussion of STRUCT, its handling of individual Fortran constructs, and its success in structuring Fortran appears in [3].

It is expected that STRUCT will be a useful tool in the maintenance of existing programs. New programs may be written in RATFOR, while existing Fortran programs may be structured into RATFOR for greater ease of modification and debugging.

Appendix A

This appendix contains the proofs of Lemmas 2 and 3. In order to prove them, another technical lemma is needed to relate $REDUCE(EXT(G))$ to $EXT(G)$ when G is irreducible. The following lemma is trivially true if G is reducible.

LEMMA A.

(*i*) *If u R-dominates v, then $u<v$.*

(*ii*) *If (r,s) is an arc in $EXT(G)$ which is replaced by an arc (r,t) in $REDUCE(EXT(G))$, $s\neq t$, then $r<t<s$.*

PROOF. The following assertion, referred to as Assertion A, is helpful in the proofs of (i) and (ii).

> Suppose p is not a **repeat** node, p is in a loop headed by r, and r has an arc to s in $EXT(G)$. Either $p=s$ or p is a descendant of s in the spanning tree generated by the depth-first search.

This assertion is easily proved from the fact that a back arc passes from a descendant to an ancestor in the spanning tree.

(i) Each path from the **start** node to v in $REDUCE(EXT(G))$ contains every R-dominator of v. Therefore, it suffices to show that there is a path from the **start** node to v in $REDUCE(EXT(G))$ in which each node other than v is $<v$.

If v is a **repeat** node, let s be the node entered by an arc from v; otherwise, let $s=v$. In G, there is a path **start** $=p_0,\ldots,p_n=s$ in which each arc passes from parent to

$s=v$. In G, there is a path **start** $=p_0,...,p_n=s$ in which each arc passes from parent to child in the spanning tree identified by the depth-first search. Since all these arcs are forward arcs, each $p_i<p_{i+1}$. In $EXT(G)$, some edges (p_i,p_{i+1}) may be replaced by edges (p_i,t), (t,p_{i+1}) in which t is a **repeat** node and $p_i<t<p_{i+1}$. Call the new path P'. By Assertion A, P' contains every **repeat** heading a loop tail containing s. Thus, v is on the path whether or not $v=s$. It is easy to show that this path is also a path in $REDUCE(EXT(G))$.

(ii) Let u be the node entered by an arc from t. If s is a **repeat** node, let w be the node entered by an arc from s; otherwise, let $w=s$. Then u immediately follows t in L, and either $s=w$ or w immediately follows s in L. Since t is the head of a loop tail containing s, $t<s$. Suppose by way of contradiction that $t<r$ Then $u<r$, and u is last visited after r is last visited during the depth-first search.

If u is also first visited before r is first visited during the search, then r is a descendant of u in the spanning tree. But then, there is a path from u to r to w which includes no node less than u. Since w is in a loop tail headed by t, there is a path in G from w to u not including any node less than u. Consequently, $EXT(G)$ has a path from t to r and from r to t not including any node less than t. Thus, r is in a loop tail headed by t, and (r,s) is not replaced by (r,t) in $REDUCE(EXT(G))$, contradicting the initial condition on r.

Therefore, u is first visited after r. But then, the arc (r,w) is searched before t is searched, and w is not a descendant of t, contradicting Assertion A. □

LEMMA 1. *A node q is a g-descendant of a node p if and only if p R-dominates q and q is in a loop tail headed by $HEAD(p)$.*

PROOF. $(=>)$ If q is a g-descendant of p, there is a sequence $p=p_0,p_1,...,p_n=q$ such that for each i, **getform**(p_{i+1}) is called during the outermost level of **getform**(p_i). Suppose $n=1$, i.e. **getform**(q) is called during the outermost level of **getform**(p). Either q is in $FOLLOW(p)$ or q is not in any $FOLLOW$ set and is entered by an arc from p.

In the former case, $HEAD(q)=HEAD(p)$ by definition. If p is an **if** node or **slc** node, $p=RDOM(q)$ by definition of $FOLLOW$ sets. If p is a **repeat** node, $RDOM(q)$ is in a loop tail headed by p and p R-dominates $RDOM(q)$. By transitivity, p R-dominates q.

In the latter case, q is entered by only one forward arc in $REDUCE(EXT(G))$. If q is entered by a back arc (s,q) in $REDUCE(EXT(G))$, (s,q) is also a back arc in $EXT(G)$ by Lemma A. Consequently, s is in a loop tail headed by q, and q R-dominates s. Therefore, back arcs need not be considered in finding $RDOM(q)$. Since the only forward arc entering q is (p,q), $p=RDOM(q)$. Moreover, since q is not in the $FOLLOW$ set of any **repeat** node, q is in a loop tail headed by $HEAD(RDOM(q))=HEAD(p)$.

The proof is completed for $n>1$ by applying the above argument inductively and noting the transitivity of the dominance relation and containment in loops.

$(<=)$ Suppose node p R-dominates q and q is in a loop tail headed by $HEAD(p)$. The R-dominance relation provides a sequence $p=r_0,...,r_n=q$ such that for each $j<n$, $r_j=RDOM(r_{j+1})$. Obtain a subsequence $p=s_0,...,s_m=q$ by deleting each r_j such that q is not in a loop tail headed by $HEAD(r_j)$. Obviously, $s_0=p$ and $s_m=q$. By transitivity, each s_j R-dominates s_{j+1}, for $j=0,...,m-1$.

Consider any s_i, $1\leqslant i<m$, for which $HEAD(s_i)$ is defined. Since the loop headed by $HEAD(s_i)$ is entered only through $HEAD(s_i)$ in $REDUCE(EXT(G))$, $HEAD(s_i)$ must be a closer R-dominator to q than any other R-dominator t of q not in this loop, i.e. t R-dominates $HEAD(s_i)$. Therefore, if s_{i-1} is not in a loop tail headed by $HEAD(s_i)$, $s_{i-1}=HEAD(s_i)$. Moreover, s_{i+1} is in a loop tail headed by $HEAD(s_i)$. For otherwise, both s_i and s_{i+1} R-dominate q, implying that s_{i+1} R-dominates $HEAD(s_i)$. But this contradicts the fact that $HEAD(s_i)$ R-dominates s_i which R-

dominates s_{i+1}.

Now, suppose $HEAD(s_i) \neq HEAD(s_{i+1})$ for some i. If $HEAD(s_i)$ is undefined, the inequality guarantees that $HEAD(s_{i+1})$ is defined, and by definition, $HEAD(s_{i+1})$ is in a loop tail headed by $HEAD(s_i)$. If $HEAD(s_i)$ is defined, the preceding paragraph implies that s_{i+1} is in a loop tail headed by $HEAD(s_i)$. Since $HEAD(s_i) \neq HEAD(s_{i+1})$, s_i cannot also be in a loop tail headed by $HEAD(s_{i+1})$. By the preceding paragraph, $s_i = HEAD(s_{i+1})$, and s_i is a **repeat** node. Since q is in a loop tail headed by $s_i = HEAD(s_{i+1})$, the node entered by an arc from s_i cannot have been deleted; this node must be s_{i+1}. Since this node is entered by no other arcs, s_{i+1} is not in any *FOLLOW* set and **getform**(s_{i+1}) is called during **getform**(s_i).

Now suppose $HEAD(s_i) = HEAD(s_{i+1})$. If $s_i \neq RDOM(s_{i+1})$, then s_i is a **repeat** node and $RDOM(s_{i+1})$ is in a loop tail headed by s_i. Therefore, s_{i+1} is in $FOLLOW(s_i)$. On the other hand, suppose $s_i = RDOM(s_{i+1})$. If s_{i+1} is entered by two or more forward arcs in $REDUCE(EXT(G))$, then s_i is an **if** node and s_{i+1} is in $FOLLOW(s_i)$. Otherwise, either s_i is an **slc** node and s_{i+1} is in $FOLLOW(s_i)$, or s_i is an **if** node with an arc to s_{i+1} and s_{i+1} is not in any *FOLLOW* set. In each case, **getform**(s_{i+1}) is called during **getform**(s_i). □

LEMMA 2. *If (r,s) is a back arc in $EXT(G)$, then s is a* **repeat** *node and r is nested within s in $ALG(G)$. If (r,s) is a forward arc in $EXT(G)$, then either s is nested within r or s is after r in $ALG(G)$.*

PROOF. If (r,s) is a back arc in $EXT(G)$, then s is a **repeat** node by construction of $EXT(G)$ and r is in a loop headed by s. By part (1) of the proof of Theorem 2, r is nested within s in $ALG(G)$.

Suppose (r,s) is a forward arc in $EXT(G)$. By definition, $r<s$. Since the construction of $EXT(G)$ eliminates self-loops, $r \neq s$. If s is a g-descendant of r, then **getform**(s) is called before **getform**(r), and either r is nested within s or s is above r. If r is a g-descendant of s, then s R-dominates r by Lemma 1. By Lemma A(i), $s<r$, contradicting the choice of $r<s$.

So suppose neither r nor s is a g-descendant of the other. Let t be the "closest" common g-ancestor of r and s, i.e. there is no g-descendant of t which is a g-ancestor of both r and s. Let u be such that **getform**(t) calls **getform**(u) and either $u=r$ or r is a g-descendant of u. Let v be such that **getform**(t) calls **getform**(v) and either $v=s$ or s is a g-descendant of v.

If r is in a loop tail headed by $HEAD(s)$, let $z=s$. Otherwise, let z be the node such that the arc (r,s) is replaced in $REDUCE(EXT(G))$ by an arc (r,z). In the former case, $r<s=z$ since (r,s) is a forward arc. In the latter case, $r<z$ by Lemma A(ii). The following argument shows that $v=z$:

> Either $z=s$ or z is the head of a loop tail containing s. In the latter case, s is nested within z by part (1) of the proof of Theorem 2, and s is a g-descendant of z. Either $s=v$ or s is a g-descendant of v. Thus, either $z=v$ or z is a g-descendant of v or v is a g-descendant of z.
>
> If v is a g-descendant of z, so are u and r. By Lemma 1, z R-dominates r. By Lemma A(i), $z<r$. But from earlier, $r<z$. The contradiction implies that v is not a g-descendant of z.
>
> Suppose z is a g-descendant of v. Then z is in a loop tail headed by $HEAD(v)$ by part (1) of the proof of Theorem 2. Moreover, so is r. Since r is not a g-descendant of v, Lemma 1 implies that v does not R-dominate r. But then the arc (r,z) prevents v from R-dominating z, which contradicts Lemma 1.
>
> The conclusion is that $v=z$.

Next, it is shown that z is in $FOLLOW(t)$. Suppose z is not in any *FOLLOW* set. Then only one forward arc enters z in $REDUCE(EXT(G))$ and it originates at t. But (r,z) is also a forward arc entering z in $REDUCE(EXT(G))$, and $r \neq t$. Consequently, z is in a *FOLLOW* set. Moreover, it must be in $FOLLOW(t)$ in order for

By Lemma 1, $u \leqslant r$ and from above, $r < z$. Either u is nested within t while z follows t, or u is also in $FOLLOW(t)$ and **getform**(u) is called before **getform**(z). Therefore, **getform**(r) is called before **getform**(z), and z appears after r in $ALG(G)$. Either $s{=}z$ or z is the head of a loop containing s and s is nested within z by part (1) of the proof of Theorem 2. Therefore, s is after r in $ALG(G)$. □

Appendix B

A Fortran subroutine (from R.C. Singleton, Algorithm 347, An efficient algorithm for sorting with minimal storage, *Comm. ACM 12,* 3 (March 1969), p. 186, with some added comments):

```
      subroutine sort(a,ii,jj)
c variation on quicksort sorts array a into increasing order from a(ii) to a(jj)
c arrays iu(k) and il(k) permit sorting up to 2**(k+1)-1 elements
      dimension a(1),iu(16),il(16)
      integer a,t,tt
      m = 1
      i = ii
      j = jj
5     if (i .ge.j) goto 70
c set t to median of a(i), a((i+j)/2), a(j)
10    k = i
      ij = (j+i)/2
      t = a(ij)
      if (a(i) .le. t) goto 20
      a(ij) = a(i)
      a(i) = t
      t=a(ij)
20    l=j
      if (a(j) .ge. t) goto 40
      a(ij) = a(j)
      a(j) = t
      t = a(ij)
      if (a(i) .le. t) goto 40
      a(ij) = a(i)
      a(i) = t
      t = a(ij)
      goto 40
30    a(l) = a(k)
      a(k) = tt
c use t to split segment
40    l = l-1
      if (a(l) .gt. t) goto 40
      tt = a(l)
50    k=k+1
      if (a(k) .lt. t) goto 50
      if (k .le. l) goto 30
c stack one segment to be sorted later
      if (l-i .le. j-k) goto 60
      il(m) = i
      iu(m) = l
      i=k
      m=m+1
      goto 80
60    il(m) = k
      iu(m)=j
      j=l
      m=m+1
      goto 80
c find next segment to be sorted
```

```
70     m=m-1
       if(m.eq. 0) return
       i=il(m)
       j=iu(m)
80     if (j-i .ge. 11) goto 10
c sort small segments
       if (i .eq. ii) goto 5
       i=i-1
90     i=i+1
       if (i .eq. j) goto 70
       t = a(i+1)
       if (a(i) .le. t) goto 90
       k=i
100    a(k+1) = a(k)
       k = k-1
       if (t .lt. a(k)) goto 100
       a(k+1) = t
       goto 90
       end
```

The preceding program as structured by STRUCT is:

```
     subroutine sort(a,ii,jj)
# variation on quicksort sorts array a into increasing order from a(ii) to a(jj)
# arrays iu(k) and il(k) permit sorting up to 2**(k+1)-1 elements
     dimension a(1),iu(16),il(16)
     integer a,t,tt
     m = 1
     i = ii
     j = jj
     repeat
        { if (i<j)
               go to 10
          repeat
             {
# find next segment to be sorted
               m = m-1
               if (m==0)
                    return
               i = il(m)
               j = iu(m)
               while (j-i>=11)
                  {
# set t to median of a(i), a((i+j)/2), a(j)
                 10 k = i
                    ij = (j+i)/2
                    t = a(ij)
                    if (a(i)>t)
                       { a(ij) = a(i)
                         a(i) = t
                         t = a(ij)
                       }
                    l = j
                    if (a(j)<t)
                       { a(ij) = a(j)
                         a(j) = t
                         t = a(ij)
                         if (a(i)>t)
                            { a(ij) = a(i)
                              a(i) = t
                              t = a(ij)
                            }
                       }
```

```
            repeat
              {
# use t to split segment
              l = l-1
              if (a(l)<=t)
                { tt = a(l)
                  repeat
                    { k = k+1
                      if (a(k)>=t)
                          break
                    }
                  if (k>l)
                      break
                  a(l) = a(k)
                  a(k) = tt
                }
              }
# stack one segment to be sorted later
            if (l-i<=j-k)
              { il(m) = k
                iu(m) = j
                j = l
                m = m+1
              }
            else
              { il(m) = i
                iu(m) = l
                i = k
                m = m+1
              }
          }
# sort small segments
          if (i==ii)
            break
          i = i-1
          repeat
            { i = i+1
              if (i==j)
                  break
              t = a(i+1)
              if (a(i)>t)
                { k = i
                  repeat
                    { a(k+1) = a(k)
                      k = k-1
                      if (t>=a(k))
                          break
                    }
                  a(k+1) = t
                }
            }
        }
    }
  return
  end
```

ACKNOWLEDGMENTS The author wishes to thank A.V. Aho, R.A. Becker, S.C. Johnson, B.W. Kernighan, and M.D. McIlroy for many helpful comments on this paper.

REFERENCES

1 AHO, A V , AND ULLMAN, J D *The Theory of Parsing, Translation, and Compiling, Vol. II. Compiling* Prentice-Hall, Englewood Cliffs, N J , 1973

2 ASHCROFT, E , AND MANNA, Z Translating program schemas to while-schemas *SIAM J. Comptg 4,* 2 (1975), 125–146

3. BAKER, B.S Struct, a program which structures Fortran Internal memo , Bell Labs , Murray Hill, N J., 1975.

4 BAKER, B S. Automatic structuring of programs In preparation

5. DE BALBINE, G Better Man Power Utilization Using Automatic Restructuring Caine, Farber & Gordon, Inc., 1974

6 DE BALBINE, G Using the Fortran structuring engine In Proc of Comp Sci and Stat 8th Ann Symp on the Interface, Los Angeles, 1975, pp 297–305

7 BOHM, C , AND JACOPINI, G Flow diagrams, Turing machines and languages with only two formation rules *Comm ACM 9,* 5 (May 1966), 366–371

8 BRUNO, J , AND STEIGLITZ, K The expression of algorithms by charts *J ACM 19,* 3 (July 1972), 366–371

9 COOPER, D C Bohm and Jacopini's reduction of flow charts *Comm ACM 10,* Aug 1967), 463 (Letter)

10 DAHL, O -J , DIJKSTRA, E W , AND HOARE, C A R *Structured Programming* Academic Press, New York, 1972.

11 DIJKSTRA, E W Go to statement considered harmful *Comm ACM 11,* 3 (March 1968), 147–148

12 HECHT, M S , AND ULLMAN, J.D Characterizations of reducible flowgraphs *J ACM 21,* 3 (July 1974), 367–375

13 KERNIGHAN, B W Ratfor — a preprocessor for a rational Fortran *Software Practice and Experience 5,* 4 (1975), 395–406

14 KERNIGHAN, B W , AND CHERRY, L L A system for typesetting mathematics *Comm ACM 18,* 3 (March 1975), 151–156

15 KNUTH, D E , AND FLOYD, R W Notes on avoiding "go to" statements *Infor Proc Letters 1* (1971), 23–31

16 KNUTH, D E Structured programming with go to statements *ACM Comptg Surveys 6,* 4 (1974), 261–302

17. KOSARAJU, S.R. Analysis of structured programs. *J. Comptr. Sys. Sci. 9,* 3 (1974), 232–254

18 LESK, M E , KERNIGHAN, B W , AND RITCHIE, D M The C programming manual Comptg Sci Tech. Rep #31, Bell Labs , Murray Hill, N J

19 PETERSON, W W , KASAMI, T , AND TOKURA, N. On the capabilities of while, repeat and exit statements *Comm ACM 16,* 8 (Aug 1973), 503–512

20 RITCHIE, D M , AND THOMPSON, K The UNIX time-sharing system *Comm ACM 17,* 7 (July 1974), 365–375

RECEIVED APRIL 1976, REVISED JULY 1976