An Algorithm for Structuring Programs:

EXTENDED ABSTRACT

Brenda S. Baker

Bell Laboratories,
Murray Hill, New Jersey 07974

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 obvious. They are preferable to **goto** statements, which tend to obscure the flow of control [DDH,DIJ]. This paper describes an algorithm which transforms a flowgraph into a program written using **repeat** (**do forever**) and **if then else** statements. The goal of the algorithm is to produce readable 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.

A number of techniques for eliminating **goto** statements from programs have been previously published [AM, BJ, BS, COO, KF, KOS, PKT]. However, these techniques do not necessarily produce clear flow of control [KN]. Misuse of control constructs may mislead the reader into expecting patterns of control flow which do not exist in the algorithm. For example, these techniques may use a **repeat** statement when the contained code cannot be executed more than once or add numerous control variables to avoid **goto** statements. They also may avoid **goto** statements by copying segments of code or creating subroutines. The former method results in longer programs and bugs may be introduced when all the identical segments must be modified. The latter method may result in subroutines which appear unnatural.

Therefore, this paper formalizes some common programming practices as a set of principles for the use of basic control constructs: **if then else**, **repeat**, multilevel **break** (a branch to the statement following an enclosing **repeat** statement), multilevel **next** (a branch to the next iteration of an enclosing **repeat** statement), **stop**, and **goto**. The principles fall into two classes: those which concern the nesting of statements and those which concern the use of branching statements (**goto**, **next**, **break**, and **stop**). A program which obeys the structuring principles is called *properly structured.* An algorithm is presented which transforms a flowgraph into a properly structured 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 properly structured program may contain **goto** statements. However, the **goto** statements occur only where no other available control construct describes the flow of control. If a flowgraph can be written as a properly structured program with no **goto** statements, the algorithm does it.

Section 2 defines flowgraphs and introduces a simple structured language **SL**. The principles concerning nesting and ordering of statements are described in Section 3. Section 4 presents the first part of the algorithm. The principles for the use of branching statements and the second part of the algorithm appear in Section 5.

Section 6 studies how the structuring principles limit the possible forms of programs representing the same flowgraph. If a flowgraph can be represented by a properly structured program with no **goto** statements, this program is unique. More generally, if a flowgraph contains no jumps into the middle of loops, *all* properly structured programs representing it have the same nesting (but not necessarily order) of statements other than **goto**, **next**, **break**, and **stop**.

Section 7 discusses briefly the application of the algorithm to structuring real programs. The algorithm has been implemented in a program called **struct**, which translates **fortran** programs into **ratfor** [KER], a **fortran** preprocessor language. The structured programs generated by **struct** are much more readable than their **fortran** counterparts. It is not usually obvious that they are mechanically generated, since the structuring principles cause them to imitate common programming practice. An example of a program structured by **struct** is included at the end of the paper.

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. **fortran** preprocessor languages such as **ratfor** have been developed so that new programs may be written using convenient control structures. But many existing programs were written in **fortran** without the benefit of preprocessors. Mechanically structuring these programs improves the readability dramatically, facilitating modification and debugging.

2. Goals of structuring

This section defines flowgraphs, a simple structured language **SL**, and what it means for an **SL** program to be a **structuring** of a flowgraph.

A flowgraph is a directed graph with labelled nodes representing computational steps and arcs representing flow of control between nodes. Each node is either a straight line code **(slc)** node with one outarc, a **stop** node with no outarcs, or a predicate **(pred)** node, with a "true" outarc and a "false" outarc. A flowgraph has exactly one **stop** 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.

The structuring algorithm presented in this paper translates a flowgraph into a simple structured language **SL**. **SL** contains optionally labelled 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 labelled **SL** statements, and **p** is a predicate,

(5) **repeat {S}**, where **S** is a (possibly null) sequence of optionally labelled **SL** statements,

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

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

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.

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** during structuring.

goto, **next(i)**, **stop**, and **break(i)** statements are referred to as *branching* statements; other statements are *nonbranching* statements.

An **SL** program is *well-formed* if the following conditions are satisfied:

(1) Every statement is accessible from the start of the program,

(2) the program contains at least one stop statement, and a **stop** statement is accessible from every **slc** statement and from both the "true" and "false" evaluations of every **if** predicate.

As a result of condition (2), every loop in a well-formed program has an exit.

A flowgraph *FLOW*(*P*) may be obtained from a well-formed **SL** program *P* to describe the flow of control between **slc** statements, **if** predicates, and **stop** statements. Each **slc** statement in *P* is represented by a distinct **slc** node, each **if** predicate in *P* is represented by a distinct **pred** node, and all **stop** statements are represented by a single **stop** node in *FLOW*(*P*). There is an arc from an **slc** node *p* to a node *q* in *FLOW*(*P*) if after executing the corresponding **slc** statement in *P*, control can pass directly to the statement represented by *q*, i.e. without first executing any other **slc** statement or **if** predicate. There is a "true" ("false") arc from a **pred** node *p* to a node *q* if control passes directly to the statement represented by *q* when the **if** predicate represented by *p* is evaluated to "true" ("false"). The **start** node of *FLOW*(*P*) is the node representing the first **slc** statement, **if** predicate, or **stop** statement executed in *P*.

Two well-formed **SL** programs P_1 and P_2 are *equivalent* if $FLOW(P_1) = FLOW(P_2)$. 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 well-formed **SL** program *P* is a *structuring* of a flowgraph *G* if $G = FLOW(P)$.

The structuring algorithm presented in this paper identifies the basic structure inherent in a flowgraph and writes it as an **SL** program. It has two parts. The first part determines the organization of the final program, that is, it decides how many **repeat** statements to use, how nonbranching statements should be nested, and the ordering of nonbranching statements. The result is a *program form.* i.e. an incomplete program consisting only of **slc**, **if then else**, and **repeat** statements together with a specification of the correspondence between these statements and nodes in the flowgraph. The second part of the algorithm determines where branching statements should be added to produce the proper flow of control. For example, if the final structured program generated by the algorithm is

```
repeat {
      if (p) then {x = x+1 }
      else {
            y = y+1
            if (q) then { goto 10 }
            else {}
            }
      y = f(y)
10    x = g(x)
      if (r) then {break(1) }
      else {}
      }
z = h(x,y)
stop
```

then the first part of the algorithm generates the following:

```
repeat {
      if (p) then {x = x+1 }
      else {
            y = y+1
            if (q) then {}
            else {}
            }
      y = f(y)
      x = g(x)
      if (r) then {}
      else {}
      }
z = h(x,y)
```

The second part of the algorithm adds the **goto 10**, the label **10**, the **break(1)**, and the **stop**.

The program form obtained by deleting all branching statements from an **SL** program *P* is denoted by *FORM*(*P*). Thus, if the algorithm generates an **SL** program *P*, the first part of the algorithm generates only *FORM*(*P*).

3. Deciding on principles for organizing programs

The goal of the algorithm presented in this paper is not to eliminate **goto** statements, since the methods of eliminating all **goto** statements have not been found to produce readable programs [KN]. Instead, the algorithm follows a set of principles for the use of control constructs to ensure that the mechanically structured programs appear natural to the reader. These principles describe some reasonable practices for programming in **SL**. They also appear to be followed (albeit flexibly) by many programmers. Since a number of principles are needed to ensure production of natural **SL** programs, some examples are presented to motivate how this set of principles evolved for use in the structuring algorithm. The principles describe how **repeat** statements, **if then else** statements, and branching statements should be used in **SL** programs.

First, some examples of uses of **repeat** statements are presented. In the following program, the **repeat** is inappropriate because it contains code which does not iterate.

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

The following are some equivalent programs with room for improvement in how **repeat** statements are used.

```
(a)  y = 1
     goto 10
     repeat {
          if (p) then {break(1) }
          else {}
     10   y = f(y)
          }
     x = g(y)
     stop
```

```
(b)  goto 10
     repeat {
          y = f(y)
          if (p) then { break(1) }
          else { next(1) }
     10   y = 1
          }
     x = g(y)
     stop
```

In (a), the **repeat** statement can be entered only by jumping into it. In (b), this problem is compounded because the statement **y = 1** is written inside the **repeat** even though it is executed only once. In particular, this statement cannot be reached after executing the statement **y = f(y)** which is the first statement in the **repeat**. A much clearer way of writing the same computation is the following.

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

The objectionable program segments presented above can be avoided by obeying the following principle.

(1) Every **repeat** statement can be entered through its "head", i.e. not just via a **goto** to a statement nested within it. Every **repeat** statement contains at least one **slc** or **if then else** statement. Every statement within a **repeat** is accessible from the first statement within the **repeat** without exiting from the **repeat**. Every nonbranching statement within the **repeat** can lead to an iteration of the **repeat** without first exiting from the **repeat**.*

Equivalent **SL** programs need not have the same number of **repeat** statements. Consider the following examples.

```
(a) repeat {
        repeat {
            if (p) then { break(1) }
            else {}
            }
        if (q) then {break(1) }
        else { x = x+1 }
        }

(b) repeat {
        if (p) then {
            if (q) then { break(1) }
            else { x = x+1 }
            }
        else {}
        }
```

The first example uses two **repeat** statements where one suffices. The algorithm generates only one **repeat** since a single **repeat** appears simpler.

(2) A **repeat** statement may not be the first nonbranching statement reached upon entering another **repeat** statement.

Next, two examples of peculiar uses of **if then else** statements are presented.

*Programmers often violate this principle in order to avoid **goto** statements when a loop has several pieces of exit code. As implemented in **struct**, this principle has an associated size limit so that small segments of code (but not large segments) may appear in a **repeat** without iterating.

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

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

In (a), the statement **y = f(j)** is placed inside the **else** clause, forcing a jump into the **else** clause from the **then** clause. In (b), the statement **j = 1** is placed after the **if** statement, forcing the **else** clause to contain a **goto** jumping around the statement following the **if** clause. Example (a) could be prevented by forbidding jumps into **then** or **else** clauses. Example (b) could be prevented by requiring that a **then** or **else** clause contain as much as possible without causing a jump into it or violating the conditions on **repeat** statements.

But problems are caused by programs in which a loop may be entered in more than one place. A flowgraph G is *reducible* [HU72] if each cycle has exactly one entry point, that is, if every cycle in G contains a node q such that every path from the **start** node to a node in the cycle must pass through q. Otherwise, a flowgraph is *irreducible*. A well-formed **SL** program P is *reducible* (*irreducible*) if $FLOW(P)$ is reducible (irreducible). Since reducibility is a property of flowgraphs, an irreducible program does not have an equivalent reducible program. (However, an irreducible flowgraph may be transformed into a reducible flowgraph by duplicating part of the graph.) Consider the following equivalent irreducible programs.

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

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

The first example allows a **goto** into the **else** clause from outside the smallest **repeat** enclosing the **else** clause. The second example avoids the **goto** into the **else** clause but forces an additional **goto** to be generated in the **then** clause. Locally, i.e. within the **repeat** statement, the first example is structured better. In particular, it does not look as if the statement **j = 2** can be executed after both the **then** and **else** clauses. Therefore, the following principles are used by the algorithm.

(3a) A **goto** may jump into a **then** or **else** clause only from outside the innermost **repeat** enclosing the clause.

(3b) A statement in the innermost **repeat** enclosing an **if then else** statement p must be placed within the **then** clause of p if this does not force a violation of Principle (3a). The same principle applies to **else** clauses.

The following principle guarantees that loops are created only by **repeat** statements and that each **goto** statement jumps to a statement which occurs after it in the program.

(4) Control may flow to a lexically preceding point in the program only to an iteration of a **repeat**, i.e. by executing a **next(i)** or by reaching the bottom of a **repeat** statement.

A well-formed **SL** program which satisfies principles (1)-(4) has *proper nesting.*

4. The first part of the structuring algorithm

The first step in structuring a flowgraph G is to locate the loops in G. A *loop* is a path of G which begins and ends at the same node. A *cycle* is a loop in which only the first node (which is the same as the last node) occurs twice. Loops can be located by constructing a spanning tree by means of a depth first search [HU74], which 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 descendant to an ancestor in the spanning tree; other arcs are *forward* arcs. Each node entered by one or more back arcs will become the first statement within a **repeat** in the final program. If a cycle has more than one entry point, exactly one entry point is entered by a back arc.

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.

At this point, the nodes which will become the first statements within **repeats** have been determined. 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, $EXT(G)$.

Note that the addition of the new nodes does not change the ordering of the nodes already in L. Therefore, an arc (p,q) is a back arc if and only if q precedes p in L..

A **repeat** node p is the *head* of all loops and cycles which include p but no nodes preceding p in L. In the final program, the corresponding **repeat** statement will contain the statements corresponding to nodes in loops 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. 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.

To produce the following segment of code

```
if (p) then { x = 1 }
else { x = 2 }
y = f(x)
```

the algorithm needs to know that the statements **x = 1** and **x = 2** can be reached only through the true and false branches of the if statement, but that **y = f(x)** can be reached through both branches.

Such branching and merging of control can be described by *dominators* in the flowgraph [AU]. 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.

Because principle (3) implies that the inside of a **repeat** must be structured as if the **repeat** can be entered only at its head, the structuring algorithm uses a modified graph for calculating dominators. Intuitively, it pretends that each arc entering a cycle 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 and p is not in a cycle 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. For each node p, $DOM(p)$ is defined to be the immediate dominator of p in the graph $REDUCE(EXT(G))$.

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 **pred** node p, define

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

For each **repeat** node p, define

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

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 non-branching statements reachable from p

which must follow p at the same level of nesting as p. For example, suppose p is a **repeat** node and q is in $FOLLOW(p)$. Since $HEAD(q) = HEAD(p)$, q must be placed within the smallest **repeat** containing p but not within p. Since $DOM(q)$ is in a loop headed by p, every path to q within this **repeat** must pass through p. By principle (3), q must be at the same level of nesting as p within this **repeat**. Furthermore, q must be below p to avoid an upward **goto**.

The nesting and ordering of nonbranching statements in the program is determined by generating a program form from the flowgraph G, i.e. a sequence $PF(G)$ of nested nonbranching statements which will be $FORM(P)$ for the final program P generated by the algorithm. $PF(G)$ is generated by calling the following recursive routine on the **start** node of the extended flowgraph $EXT(G)$. To be precise, the correspondence between statements of $PF(G)$ and nodes of $EXT(G)$ should be specified; for simplicity, it is merely assumed to exist.

```
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 a pred node with
        predicate r and a true arc to
        node p and a false arc to node q)
        then {
            print("if (r) then {")
            call test(p)
            print("} else {")
            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 resulting program form is $PF(G)$.

5. Branching statements and the second part of the algorithm

Next, the use of branching statements is considered. The first principle for the use of branching statements is the following.

(5) A **goto** statement may not jump to another branching statement. A **goto** may not jump to the first statement inside a **repeat**; it must jump to the **repeat** instead. A branching statement may not appear unless deleting it alters the flow of control in the program.

The above principle does not specify where branching statements should be used. Consider the following example.

```
if (p) then {
      x = 1
      stop
      }
else {
      x = 2
      stop
      }
```

The form of the following program is preferable.

```
if (p) then {x = 1 }
else { x = 2 }
stop
```

On the other hand, when the **then** and **else** clauses jump to different places, it is probably preferable to put the branching statements inside the **then** and **else** clauses. The following principle is followed by the structuring algorithm to determine where branching statements should be added.

(6) A branching statement appears after a nonbranching statement p if and only if there is exactly one node q in the corresponding flowgraph such that both of the following conditions hold:

(a) q does not correspond to an **slc** or **if** statement nested within p

(b) q is the only node satisfying (a) which is entered by an arc from the node corresponding to p or from a node corresponding to an **slc** or **if** statement nested within p.

The next principle assigns a preference order to branching statements to ensure that branching statements following **repeat** statements can be reached.

(7) Branching statements are used in the following order of precedence: **break(i)** for any **i**, **next(i)** for any **i**, **stop**, **goto**. That is, a statement in the list may not be used if a choice earlier in the list may be substituted without altering the flow of control between nonbranching statements.

This principle ensures that a **goto** statement is not used when another branching statement suffices.

A program which follows principles (5)-(7) has *proper branching*.

The second part of the algorithm adds proper branching to $PF(G)$ by computing the statements directly reachable from each nonbranching statement but not nested within it. In particular, for a node p in $EXT(G)$, $REACH(p)$ is 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 a statement nested within p. Branching statements are added to the program recursively from outer levels of nesting to inner levels. A branching statement is added after a nonbranching statement p if $REACH(p)$ contains exactly one node q, and q is not the node corresponding to the statement reached automatically in the program if no branching statement follows p. If a **then** or **else** clause (or the entire program) contains no nonbranching statements, a branching statement is added if it is needed to ensure proper flow of control. The choice of branching statement is determined by principle (7) and $EXT(G)$. A label is added to each statement entered by a **goto**.

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 $ALG(G)$.

A well-formed **SL** program with proper nesting and proper branching is *properly structured.* If P is a structuring of a flowgraph G and P is properly structured, P is a *proper structuring* of G.

Theorem 1. *ALG(G)* is a proper structuring of G.

A nice feature of the algorithm is that it does not generate **goto** statements needlessly.

Theorem 2. If a flowgraph has a proper structuring with no **goto** statements, the algorithm produces one.

A simple analysis of the algorithm yields the following upper bounds for time and space.

Theorem 3. In the worst case, the generation of $ALG(G)$ from G requires at most space $O(n^2)$ and time $O(n^2 \log n)$, where n is the number of nodes in G.

In practice, the implementation of the algorithm in **struct** handles **fortran** programs several hundred lines in length in a reasonable amount of time on a pdp-11/45 with 60K 8-bit bytes.

6. Properly structured programs

In this section, the implications of the structuring principles are investigated.

One question one might ask about the conditions for proper nesting and proper branching is how much flexibility they permit in writing programs. In particular, suppose one is given a computation specified by a flowgraph. In writing an **SL** program to perform this computation, how much flexibility is there in the number and type of control statements, in the nesting of statements, and in the ordering of statements at each level of nesting? In other words, where may differences occur between equivalent properly nested programs? Consider the following example.

```
if (p) then {}
     else { goto 10 }
repeat {
     x = f(y)
10   y = f(x)
     if (q) then { break(1) }
     else {}
     }
```

This code segment could be rewritten as

```
if (p) then {goto 10 }
     else {}
repeat {
     y = f(x)
     if (q) then {break(1) }
     else {}
10   x = f(y) }
```

There is a choice because the loop is entered in two places, that is, the underlying flowgraph is irreducible. However, when the underlying flowgraph is reducible, there is no flexibility in the number or nesting of nonbranching statements.

Theorem 4. If P_1 and P_2 are equivalent properly nested reducible **SL** programs, then $FORM(P_1)$ and $FORM(P_2)$ are identical in the number of occurrences of each nonbranching statement and in how the nonbranching statements are nested within each other.

Note that Theorem 4 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.

Theorem 5. If P_1 and P_2 are equivalent properly nested **SL** programs with no **goto** statements, then $FORM(P_1) = FORM(P_2)$.

Proper branching does not restrict the form of the program.

Lemma 1. For every properly nested **SL** program P_1, there exists an equivalent **SL** program P_2 with proper branching such that $FORM(P_2) = FORM(P_1)$. Moreover, P_2 is unique (except for labels on statements).

From the above theorems and lemma, it follows that equivalent properly structured **SL** programs with no **goto** statements are identical. In terms of the algorithm, this result may be stated as follows.

Corollary 1. If P is a properly structured **SL** program with no **goto** statements, then $P = ALG(FLOW(P))$.

Intuitively, this corollary states that flexibility in writing a gotoless program occurs only in choosing the flowgraph or in choosing to violate the principles of proper structuring. In other words, when a programmer wishes to write a properly structured program without **goto** statements, flexibility lies only in modifying the computation to be performed, i.e. the flowgraph for the program, and not in the way the program itself describes the computation.

It must be noted that an **SL** program without **goto** statements may not have an equivalent properly structured **SL** program because of the restriction that a **repeat** may not be the first statement inside another **repeat**. The following code segment is an example.

```
repeat {
      repeat {
            if (s) then {break(2) }
            else {
                  if (p) then {
                        if (q) then {}
                        else { break(1) }
                        }
                  else {
                        if (r) then {}
                        else {break(1) }
                  x = x + 1
                  }
            }
      x = x + 2
      }
```

If **x = x+2** and **x = x+1** were within a single **repeat**, they would be at the same level of nesting as the **if (p)** statement. But at most one of them could be reached without a **goto** from within the **if (p)** statement. Therefore, there is no equivalent properly nested program without **goto** statements.

7. Applying the algorithm

The algorithm has been implemented in a program called **struct**, which rewrites **fortran** programs in **ratfor**[KER]. The basic algorithm is extended in **struct** to generate (optional) additional constructs such as **while** loops and a form of **case** statement. Predicates are negated by **struct** when necessary for the generation of **if then** statements. **ratfor** has only single-level **break** and **next** statements. Therefore, **struct** does not adhere strictly to the conditions of proper branching. The Appendix 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. A more extensive discussion of **struct** and its success in structuring **fortran** appears in [BAK].

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.

Acknowledgements

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

References

[AU] A. V. Aho and J. D. Ullman, *The Theory of Parsing, Translation, and Compiling, Vol. II - Compiling*, Prentice-Hall, Englewood Cliffs, N.J., 1973.

[AM] E. Ashcroft and Z. Manna, Translating program schemas to while-schemas, *SIAM J. on Comp. 4,2* (1975), 125-146.

[BAK] B. S. Baker, **struct**, a program which structures **fortran**, in preparation.

[BJ] C. Bohm and G. Jacopini, Flow diagrams, Turing machines and languages with only two formation rules, *Comm. ACM 9* (1966), 366-371.

[BS] J. Bruno and K. Steiglitz, The expression of algorithms by charts, *J. ACM 19* (1966),366-371.

[COO] D. C. Cooper, Bohm and Jacopini's reduction of flow charts, *Comm. ACM 10R (1967),463.*

[DDH]O.-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, *Structured Programming*, Academic Press, New York, 1972.

[DIJ] E. W. Dijkstra, Goto statement considered harmful, *Comm. ACM 11* (1968), 147-148.

[HU74]M. S. Hecht and J. D. Ullman, Characterizations of reducible flowgraphs, *J. ACM 21,3* (1974), 367-375.

[HU72]M. S. Hecht and J. D. Ullman, Flow graph reducibility, *SIAM J. Comput. 1* (1972), 188-202.

[KER]B. W. Kernighan, **ratfor** - a preprocessor for a rational **fortran**, *Software Practice and Experience 5,4* (1975), 395-406.

[KF] D. E. Knuth and R. W. Floyd, Notes on avoiding "go to" statements, *Infor. Proc. Letters 1* (1971), 23-31.

[KN] D. E. Knuth, Structured programming with goto statements, *ACM Comp. Surveys 6,4* (1974), 261-302.

[KOS]S. R. Kosaraju, Analysis of structured programs, *J. Comp. Sys. Sci. 9,3* (1974), 232-254.

[PKT]W. W. Peterson, T. Kasami, and N. Tokura, On the capabilities of while, repeat and exit statements, *Comm. ACM 16* (1973), 503-512.

Appendix

A **fortran** subroutine (from R. C. Singleton, Algorithm 347, an efficient algorithm for sorting with minimal storage, *Comm. ACM 12,3* (1969), p. 186):

```
      subroutine sort(a,ii,jj)
c sorts array a into increasing order
c from a(ii) to a(jj)
      dimension a(1),iu(16),il(16)
      integer a,t,tt
      m = 1
      i = ii
      j = jj
5     if (i .ge.j) goto 70
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
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
      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
70    m=m-1
      if(m.eq. 0) return
      i=il(m)
      j=iu(m)
80    if (j-i .ge. 11) goto 10
      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 subroutine as structured by **struct** into **ratfor**:

```
   subroutine sort(a,ii,jj)
# sorts array a into increasing order
# from a(ii) to a(jj)
   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
         {
         m = m-1
         if (m==0)
            return
         i = il(m)
         j = iu(m)
         while (j-i>=11)
            {
         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
               {
               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
                  }
               }
            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
               }
            }
         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
```