EFFICIENCY CONSIDERATIONS IN IMPLEMENTING DIJKSTRA'S GUARDED COMMAND CONSTRUCTS

Marguerite Elaine McGuire

TR79-382

Department of Computer Science
Cornell University
Ithaca, NY 14853

EFFICIENCY CONSIDERATIONS IN IMPLEMENTING DIJKSTRA'S GUARDED COMMAND CONSTRUCTS

A Thesis

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment for the Degree of

Master of Science

by

Marguerite Elaine McGuire

May, 1979

Abstract

The guarded command alternative and iterative constructs proposed by E.W. Dijkstra subsume the conventional alternative and iterative constructs. The extra flexibility of these guarded command constructs enables the programmer to express his ideas more directly and clearly. Moreover, Dijkstra has developed a calculus for the derivation of correct programs that utilizes these guarded command constructs.

This thesis addresses the problem of efficiently implementing these guarded command constructs. Several new optimizations that are particularly well suited to the guarded command constructs are described. The most useful is the elimination of redundant boolean expressions. This optimization provides a means of implementing the guarded command alternative statement with efficiency comparable to the IF-THEN-ELSE statement.

The main contribution of this thesis is a detailed description of an algorithm for eliminating redundant boolean expressions. The algorithm itself is presented in a program written in a PASCAL supplemented with the guarded command constructs. The basic method involves considering individual execution paths through the guarded command construct and applying rules of inference to recognize and avoid evaluation of many boolean expressions. It is shown that the number of execution paths through a guarded command construct remains small enough to make this method practical.

Biographical Sketch

Marguerite Elaine McGuire was born in Brockton, Massachusetts on November 28, 1955. She received her Bachelor of Arts degree in Mathematics from Vassar College in May of 1977, awarded general and departmental honors. She is a member of the Phi Beta Kappa Honor Society and the Association for Computing Machinery.

To my parents, Frederick and Leona

and to a dear friend, Walt

Acknowledgements

I am very grateful to my advisor, David Gries, for suggesting this interesting topic. His suggestions and constructive criticisms greatly improved this thesis and my programming style.

I would like to give special thanks to Walter O'Brian III. Many long discussions with him led to importants insights. His constant support and encouragement helped me over the rough spots to the completion of this thesis.

I would also like to thank Gary Levin and Bob Constable for their interest in my work and their valuable suggestions.

Finally, I am deeply grateful to Bell Laboratories for supporting me during my graduate studies at Cornell.

Table of Contents

Chapter 1

Introduction

E. W. Dijkstra has introduced alternative and iterative statements that differ somewhat from the corresponding conventional IF- and DO-constructs. The alternative statement has the form

$$\underline{\text{if}}\ B_1 \rightarrow SL_1 \ \square\ B_2 \rightarrow SL_2 \ \square\ \dots\ \square\ B_n \rightarrow SL_n\ \underline{\text{fi}}\ .$$

The boolean expressions preceding the arrows are called <u>guards</u>. Following each arrow is a list of statements, SL_i, that may be executed only if the corresponding guard, B_i, is initially true. Thus the guard may prevent execution of the statements whenever this execution is undesirable. Hence the syntactic entity $B_i \rightarrow SL_i$ is called a <u>guarded</u> <u>command</u>.

To execute the alternative statement, one of the guarded commands whose guard is true, say B_j, is selected. The statement sequence SL_j is then executed and execution of the alternative statement is complete. If all the B_i are false, the program aborts.

The iterative statement has the form

$$\underline{\text{do}}\ B_1 \rightarrow SL_1 \ \square\ B_2 \rightarrow SL_2 \ \square\ \dots\ \square\ B_n \rightarrow SL_n\ \underline{\text{od}}\ .$$

One of the guarded commands whose guard is true, say B_j, is selected and SL_j is executed. This process is repeated until all the B_i are simultaneously false. At this point, execution of the iterative statement is complete.

The conventional <u>while</u> B <u>do</u> SL is easily expressed by the guarded command construct <u>do</u> $B \rightarrow SL$ <u>od</u>, and <u>if</u> B <u>then</u> SL_1 <u>else</u> SL_2.

can be written as <u>if</u> B → SL_1 [] <u>not</u> B → SL_2 <u>fi</u>; hence the guarded command constructs subsume the conventional constructs. Furthermore, the guarded commands are preferable to the conventional constructs for handling problems whose solutions must be given in terms of several cases. The following problem, presented by Dijkstra in 1975, illustrates this.

Consider three monotonically nondecreasing functions f, g, and h defined on the nonnegative integers. There exists at least one value x and three integers i, j, and k such that $x = f(i) = g(j) = h(k)$. Call the least such value $\bar{x}$ and the corresponding smallest possible integers $\bar{i}$, $\bar{j}$, and $\bar{k}$. The problem is to find $\bar{x}$ - that is, to establish the truth of

$$i = \bar{i} \ \underline{\text{and}} \ j = \bar{j} \ \underline{\text{and}} \ k = \bar{k},$$

where i, j, and k are program variables whose values are to be determined.

A solution using the conventional constructs is:

```
i, j, k := 0, 0, 0;
while f(i) ≠ g(j) or g(j) ≠ h(k) do
      if f(i) < g(j) then i := i + 1
         else if g(j) < h(k) then j := j + 1
                else k := k + 1 .
```

A solution using the guarded command statements is:

```
i, j, k := 0, 0, 0;
do f(i) < g(j) → i := i + 1
[] g(j) < h(k) → j := j + 1
[] h(k) < f(i) → k := k + 1
od
```

with loop invariant ($0 \leq i \leq \bar{i}$ and $0 \leq j \leq \bar{j}$ and $0 \leq k \leq \bar{k}$).

The latter program concisely expresses the solution with one iterative statement, whereas the former uses a WHILE-loop and two nested IF-statements. The semantics of the guarded command constructs provide a means of expressing the solution in terms of cases that are not necessarily disjoint and permits the reader to consider the three cases independently. This keeps the number of details the reader must deal with at any moment down to the essential minimum.

The nested IF structure of the first program, on the other hand, misleads the reader into believing that the ordering of these cases is important. In order for the third statement list, i.e., $k := k + 1$, to be executed, the conjunction of the negations of the surrounding booleans, i.e., $f(i) \geq g(j)$ and $g(j) \geq h(k)$, must be true. This conjunction together with the truth of the loop boolean implies that $h(k) < f(i)$. This boolean alone indicates the need to increment k, but this first program forces additional requirements. Not only does this first program confuse the reader and force him to reason about the implications of all the booleans combined, but also it introduces complexity in the proof of program correctness.

This example illustrates that the extra flexibility provided by the guarded command constructs enables the programmer to express his intentions more directly and clearly. Moreover, Dijkstra has provided a calculus for deriving correct programs that uses these guarded command constructs [2]. Arguments supporting the superiority of the guarded command iterative statement with respect to program correctness are given by Gries [4].

In the straightforward implementation of the guarded command constructs, guards are evaluated in their order of occurrence until a true guard is found, which can cause execution of a guarded command construct to take longer than it need take. Consider, for example, the following program for searching a two-dimensional array b for a value x.

$i, j := 0,0;$

$\underline{\text{do}}\ i \neq n\ \underline{\text{cand}}\ j \neq m\ \underline{\text{cand}}\ x \neq b(i,j) \rightarrow j := j + 1$

$\square\ i \neq n\ \underline{\text{cand}}\ j = m \qquad\qquad\qquad \rightarrow i, j := i + 1, 0$

$\underline{\text{od}}$

$\{(0 \leq i < n\ \underline{\text{and}}\ 0 \leq j < m\ \underline{\text{and}}\ x{=}b(i,j))\ \underline{\text{or}}\ (i{=}n\ \underline{\text{and}}\ x{\notin}b)\}$

Notice that the value of $i \neq n$ in the first guard completely determines the value of $i \neq n$ in the second guard. Thus the time spent evaluating this second occurrence of $i \neq n$ is wasted. Similarly, the value of $j \neq m$ may be used to determine the value of $j = m$. Thus $j = m$ need not be evaluated whenever it is preceded by the evaluation of $j \neq m$.

For another savings in time, consider an iteration of this statement in which the first guard is true and j := j + 1 is executed. The value of $i \neq n$ is true before executing this assignment statement and none of its operands are changed by it. Hence $i \neq n$ must still be true and thus need not be reevaluated during the next execution of the iterative statement.

These and other ways of improving the execution time of the guarded command constructs will be presented and discussed in subsequent chapters. Most of the existing time optimizations can be applied to the guarded command constructs, but new optimizations will also be shown to be very useful. In particular, one optimization is essential in providing a means of implementing the alternative statement with efficiency comparable to the IF-THEN-ELSE statement. A large part of this thesis deals with presenting, in general and in detail, the new ideas needed to implement these optimizations.

Chapter 2

Existing Code Optimization

The idea of improving the execution time of the object language program corresponding to a given source language program is certainly not new. Several techniques for implementing certain optimizations already exist in the literature [1,3,6]. The specific optimizations that have been addressed may be divided into three areas. One is local optimizations, those performed on blocks of straight-line code that have only one entrance and one exit - at the beginning and end of the block, respectively. Hereafter such blocks will be called basic blocks. The second is loop optimizations, those performed on the existing iterative constructs. Finally there are global optimizations, those performed using information provided by data-flow analysis.

2.1 Internal Forms

Optimizations may be performed within the semantic routines that generate the object code for the input source program. Often, however, the source program is converted into an intermediate internal form over which one or more optimizing passes are made. Polish notation, triples and quadruples are some of the internal representations currently used [1,3]. This latter approach often makes it easier to illustrate specific optimizations. Hence it is used in some of the following examples of optimizations. Table 2.1 gives the subset of quadruples used in the examples. Any parameter - R, V, Vi - can be an indirect variable. Dereferencing of such variables is indicated by <u>ref</u> - e.g., <u>ref</u> R.

TABLE 2.1. Quadruples to be used.

	Operator	Operands	Meaning
1.	B	, ,(i)	Branch to quadruple i.
2.	BTF	BOOL,(i),(j)	If BOOL is "true", then branch to quadruple i. If BOOL is "false", then branch to quadruple j.
3.	CVIR	V, ,R	Convert the value of V from integer to real and store the result in R.
4.	<[≤,=,≥,>,≠]	V1,V2,R	If V1 < V2 [V1 ≤ V2, V1 = V2, etc], then set R to "true." Otherwise, set R to "false."
5.	+[-,*,/,**]	V1,V2,R	Add [subtract, multiply, divide, exponentiate] the value of V1 to the value of V2 and store the result in R.
6.	:=	V1, ,V2	Store the value of V1 in the location of V2.

2.2 Local Optimization

2.2.1 Constant Folding

Two local optimizations are implemented in almost all optimizing compilers. One, called <u>constant folding</u>, is the replacement of runtime operations by compile-time computations. It can only be done when the value of each of the operands of the operation is known at compile-time. Constant folding is most frequently applied to arithmetic operations, but should also be applied to conversion operations if they explicitly appear in the internal representation.

Consider the program segment given in Figure 2.1a and its internal representation in Figure 2.1b. Clearly, quadruple (2) may be executed at compile-time. Moreover, since the value of PI to be used in quadruple (3) is always known at compile time, this quadruple may also be executed at compile-time. References to the temporaries, T1 and T2, that hold the results of these quadruples may then be replaced by the corresponding compile-time computed constant. Finally, quadruples (2) and (3) may be removed to yield the result given in Figure 2.1c.

PI:= 3.14159	(1) (:= 3.14159, ,PI)	(1) (:= 3.14159, , PI)
AREA:= 4*PI*(R**2)	(2) (CVIR 4, , T1)	(2) (** R,2,T3)
	(3) (* T1,PI,T2)	(3) (* 12.56636,T3,T4)
	(4) (** R,2,T3)	(4) (:= T4, ,AREA)
	(5) (* T2,T3,T4)	
	(6) (:= T4, , AREA)	
(a) Program segment	(b) Original quadruples	(c) Optimized quadruples

Figure 2.1 Illustration of constant folding.

2.2.2 Redundant Operation Elimination

The second common local optimization is the elimination of redundant operations. An operation ρ_2 is <u>redundant</u> if there exists an identical operation ρ_1 preceding ρ_2 and the values of ρ_2's operands are not changed along the path from ρ_1 to ρ_2. Even the best programs have redundant operations, for two reasons. First, redundancy may be introduced purposefully to make the program easier to read and understand. Secondly, certain redundant operations are invisible in the source program but clearly appear in the internal representation, and hence are out of the programmer's control. Such redundant operations often arise when referencing subscripted variables.

In a byte-addressable machine, an offset such as 4*I must be computed in order to access the I^{th} element of an array A. This offset must then be added to the base location of the array A to yield the actual address of A[I]. Assuming the use of a byte-addressable machine, consider the program segment given in Figure 2.2a. It looks deceptively simple. The internal form of this program segment, given in Figure 2.2b, more accurately indicates the operations that must be performed in executing this program segment. Looking at this initial internal representation, the second reference to A[I] is easily seen to produce two redundant operations. Clearly, quadruples (1) and (3) compute the same value; so do quadruples (2) and (4). Hence we may replace all occurrences of T3 and T4 by T1 and T2, respectively, and remove quadruples (3) and (4). The final result is given in Figure 2.2c. (Multiplication by a power of 2 is usually implemented by a shift operation; but for clarity, the multiplication is used here.)

A[I]:=A[I]+1	(1) (* 4,I,T1)	(1) (* 4,I,T1)
	(2) (+ BASELOC_A,T1,T2)	(2) (+ BASELOC_A,T1,T
	(3) (* 4,I,T3)	(3) (+ 1,ref T2,ref T
	(4) (+ BASELOC_A,T3,T4)	
	(5) (+ 1,ref T4,ref T2)	
(a) Program segment	(b) Original quadruples	(c) Optimized quadrupl

Figure 2.2. Illustration of redundant operation elimination.

2.3 Loop Optimization

2.3.1 Factoring Out Loop Invariant Operations

An operation ρ is invariant in a loop if and only if the value of each of its operands is not changed within the body of that loop.

Invariant operations may be factored out of the loop and placed within the basic block immediately preceding the loop without altering the semantics of the program. The more heavily used the loop is, the greater the saving in execution time. Consider, for example, the program segment given in Figure 2.3a. The value of the variable PI is not changed within the body of the loop; hence the operation 4*PI is invariant in the loop. Factoring this operation out of the loop, as shown in Figure 2.3b, saves the time of 499 executions of this operation.

```
for RADIUS := 1 to 500 do
   begin
     FOUR-PI := 4*PI;
     AREA := FOUR_PI*(RADIUS**2);
     writeln (RADIUS,AREA)
   end
```

(a) Original Program segment

```
FOUR_PI := 4*PI;
 for RADIUS := 1 to 500 do
  begin
    AREA := FOUR_PI*(RADIUS**2);
    writeln (RADIUS,AREA)
  end
```

(b) Optimized program segment

Figure 2.3. Illustration of factoring out loop invariant operations.

2.3.2 Strength Reduction

Strength reduction refers to the replacing of an operation within a loop by some equivalent operation that executes faster. The two most common strength reductions are: (1) changing a multiplication in which one of the factors is the loop index to an addition, and (2) changing an exponentiation in which the loop index

is the exponent to a multiplication. Figure 2.4 illustrates the latter strength reduction. Variable T is a temporary introduced by the optimizing compiler.

```
for I := 1 to N do          T := 1;
    A[I] := X**I             for I := 1 to N do
                                 begin
                                     T := X*T;
                                     A[I] := T
                                 end
```

(a) Original program segment (b) Optimized program segment

Figure 2.4. Illustration of strength reduction.

2.3.3 Induction Variable Elimination

An <u>induction variable</u> is one whose values form an arithmetic progression. When multiples of an induction variable are also computed within the loop, it may be possible to replace the induction variable by one of its multiples. This replacement process is called <u>induction variable elimination</u>. An induction variable may be eliminated only if the use of its loop value is local to the loop.

One of the most common applications of this optimization concerns loops in which the loop index is used to reference an array element. Figure 2.4a gives such a loop. Examining the corresponding internal representation given in Figure 2.4b, we see that I controls the iteration of the loop, while 4*I is the offset needed to locate the array elements. Both are reevaluated upon each iteration of the loop, even though the value of one completely determines the value of the other. Instead, the temporary variable holding the offset of the array element currently being accessed may be used to control the loop iteration. Thus the loop index I is no longer needed and all references to it may be eliminated. The final result of this optimization is given in Figure 2.4c.

<u>for</u> I := 1 <u>to</u> 10 <u>do</u>	(1) (:= 1, , I)	(1) (:= 4, ,T0)
A[I] := B[I]	(2) (> I,10,T1)	(2) (> T0,40,T1)
	(3) (BTF T1,(10),(4))	(3) (BTF T1,(9),(4))
	(4) (* 4,I,T2)	(4) (+ BASELOC_A,T0,T3)
	(5) (+ BASELOC_A,T2,T3)	(5) (+ BASELOC_B,T0,T4)
	(6) (+ BASELOC_B,T2,T4)	(6) (:= <u>ref</u> T4, ,<u>ref</u> T3)
	(7) (:= <u>ref</u> T4, ,<u>ref</u> T3)	(7) (+ T0, 4,T0)
	(8) (+ I,1,I)	(8) (B , ,(2))
	(9) (B , ,(2))	(9)
	(10)	
(a) Program segment	(b) Original quadruples	(c) Optimized quadruples

Figure 2.4. Illustration of induction variable elimination.

2.3.4 Loop Unrolling

If the number of iterations of a loop is constant, then the number of tests for termination may be reduced by inserting copies of the body of the loop. This replication process is called <u>loop unrolling</u>. The number of copies inserted within the loop may be any divisor of the number of iterations, however the trade-off between time and space must be considered in choosing this number. In the loop of Figure 2.5, 50 tests for termination are avoided by copying the loop body just once.

```
I := 1;                        I := 1;
while I ≤ 100 do               while I ≤ 100 do
  begin                          begin
    A[I] := 0;                     A[I] := 0;
    I := I + 1                     I := I + 1;
  end                              A[I] := 0;
                                   I := I + 1
                                 end
```

(a) Original program segment (b) Optimized program segment

Figure 2.5. Illustration of loop unrolling.

2.3.5 Loop Fusion

Loop fusion refers to merging the bodies of two loops. In order for two successive loops to be merged, they must have the same loop index and be iterated the same number of times. Moreover, to maintain the semantics of the original program, no value computed in the i^{th} iteration of the first loop may be computed in the j^{th} iteration of the second loop, $j \leq i$, and no value computed in the i^{th} iteration of the first loop may be used in the j^{th} iteration of the second loop, $j \leq i$. By merging two loops, all the tests for termination of one loop are eliminated.

```
for R := 1 to N do                for R := 1 to N do
    C[R] := 2*PI*R;                   begin
for R := 1 to N do                        C[R] := 2*PI*R;
    A[R]: = 2*C[R]*R                      A[R] := 2*C[R]*R
                                      end
```

(a) Original program segment (b) Optimized program segment.

Figure 2.6. Illustration of loop fusion.

2.3.6 Loop Splicing

The unconditional branch at the end of a loop may be eliminated by moving the test for termination to the end of the loop and jumping to that test to start execution of the loop. This process is called <u>loop splicing</u> and is usually applied to the innermost loops. It saves the execution time of one unconditional branch for each iteration of the loop. Figure 2.7c indicates the rearranging that must be done in the original internal form, given in Figure 2.7b, for loop splicing.

I := 0;	(1) (:= 0, ,I)	(1) (:= 0, ,I)
while I ≤ N and A[I] ≠ X	(2) (≤ I,N,T1)	(2) (B , ,(5))
do	(3) (BTF T1,(4),(10))	(3) (+ I,1,I)
I := I + 1	(4) (* 4,I,T2)	(4) (≤ I,N,T1)
	(5) (+ LOC_A,T2,T3)	(5) (BTF T1,(6),(10))
	(6) (≠ ref T3,X,T4)	(6) (* 4,I,T2)
	(7) (BTF T4,(8),(10))	(7) (+ LOC_A,T2,T3)
	(8) (+ I,1,I)	(8) (≠ ref T3,X,T4)
	(9) (B , ,(2))	(9) (BTF T4,(3),(10))
	(10)	(10)
(a) Program segment	(b) Original quadruples	(c) Optimized quadruples

Figure 2.7. Illustration of loop splicing.

2.4 Global Optimization

Certain optimizations can only be performed by looking at the program as a whole. Information from various parts of the program must be accumulated at the desired point of optimization to determine whether optimization is possible. This information is gathered via data-flow analysis.

Given a statement s at some point p of the program, data-flow analysis routines examine all paths of a program passing through p and gather the information pertinent to s and the particular optimization being considered. Using this information, the optimization is either performed or rejected.

A loop optimization that requires some data-flow analysis has already been mentioned - induction variable elimination. The entire program must be examined to determine which induction variables have their loop values referenced outside the loop. Those that do cannot be eliminated.

Global constant folding also requires data-flow analysis. Figure 2.8 indicates the various paths passing through the basic block B1. We would like to replace variable PI by a constant. In order to legally do this (i.e., maintain the program's semantics), the value of PI must be the same constant along all paths - P1, P2 and P3 - passing through B1. Data-flow analysis routines provide the information needed to determine this.

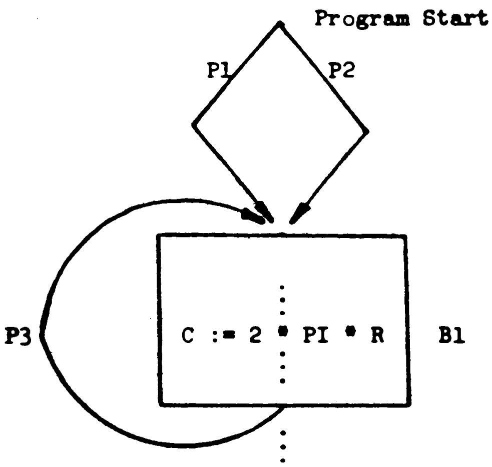


Figure 2.8. Illustration of data flow.

Redundant operation elimination may also be performed on a global level. The concept is the same as described before - avoid evaluating an expression if its value is already known. Again, as was the case for constant folding, the difference between local and global redundant operation elimination is the manner in which they are implemented. It is important to note that these local and global optimizations are implemented by looking at the program on different levels of detail. Thus neither type of optimization replaces the other.

Chapter 3

New Code Optimizations

Each of the optimizations discussed in the preceding chapter may be applied to some program segment expressed in terms of the guarded command constructs. There are, however, optimizations that have not yet been addressed in the literature and that are particularly well suited to the guarded command constructs. In this chapter, we shall examine the semantics of the guarded command constructs and discuss the optimizations to which it naturally leads.

3.1 Ordering Guarded Commands

The exact syntax and semantics of the guarded command constructs have already been given in Chapter 1. Recall that the body of both the alternative and iterative statements is a <u>guarded command set</u> (abbreviated GCS) - i.e., a finite number of guarded commands. Each guarded command consists of a guard and a corresponding statement list. Common to the execution of both the alternative and iterative statements is the process of (1) selecting one of the true guards and executing the corresponding statement list or (2) determining that all the guards are false. This process is referred to as the <u>execution of the guarded command set</u>.

There are several ways of implementing the execution of a guarded command set. The most straightforward manner is as follows. Each of the guards are evaluated in the order of their occurrence within the guarded command set until a true guard is found. The statement list associated with this true guard is executed and execution of the guarded command set is complete. None of the guards following this first true guard are evaluated. If no guard is true, then all the guards are evaluated and found to be false.

Although this manner of implementation is perfectly legitimate, it does not take advantage of the nondeterminism with which a true guard may be chosen. This nondeterminism makes it possible to consider the guarded commands in any order. Hence it makes sense to order the guarded commands using some criterion that, in most cases, minimizes the execution time of the guarded command set.

One criterion that may be used is the amount of time required to evaluate each guard. Of course, no exact evaluation time can be given; however, a useful relative time measure can be derived. A relative time cost can be assigned to each of the operations that may occur within a guard - e.g., scalar and subscripted variable references, arithmetic and relational operations, functions calls, etc. Except for the function call cost, each of these time costs should be finite. Since there is no way of determining the amount of time spent executing the called function, the cost of a function call should be ∞. A relative time measure can be derived using the operation time costs. Clearly, such a measure ensures that a guard that contains a function call is evaluated last. If all the guards contain function calls, this measure provides no useful information. In general however, guards are short, simple, and free of function calls. Thus, the guard evaluation time given by such a measure is a useful criterion for ordering the guarded commands. Implementing this ordering is straightforward once the cost measure is defined.

The execution time of the statement list is rejected as a criterion for ordering the guarded commands. Statement lists are often long and frequently contain procedure calls, function calls and loops - each of which must be assigned a time cost of ∞. Hence estimation of execution time would often be costly itself and yield no useful information. If the programmer knows that a certain statement list requires much execution time, he may force it to be executed only when all the other guards are false by making the guarded command set deterministic. Other criteria for ordering the guarded commands will be suggested after the optimizations on which they depend are introduced.

3.2 Redundant Operation Elimination Within the Guards

3.2.1 Obvious Redundant Operations

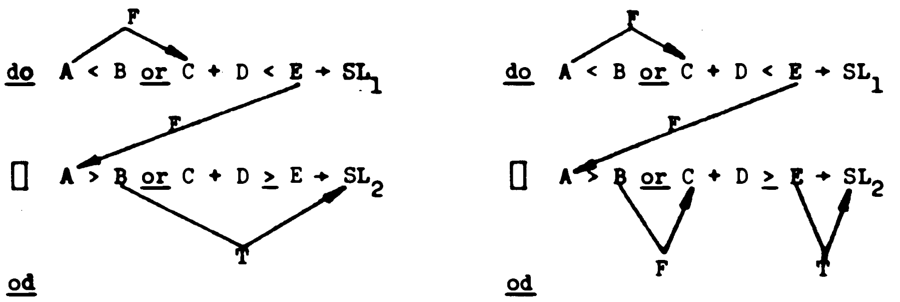
Regardless of the criterion used for ordering the guarded commands, each execution of a guarded command set may require the evaluation of more than one guard. Often an operation ρ within one of these guards g_i (say) is repeated in another guard g_j. The occurrence of ρ in g_j is redundant if:

(1) g_j is evaluated after g_i is evaluated,

(2) ρ is evaluated during this evaluation of g_i, and

(3) ρ is evaluated during this evaluation of g_j.

Consider, for example, the execution of an iterative statement depicted in Figure 3.1a. Both the first and second guards are evaluated. The operation C + D is present in both guards, but it is only evaluated in the evaluation of the first guard. Hence the occurrence of C + D in the second guard is not redundant in this execution of the guarded command set. Figure 3.1b, on the other hand, depicts an execution of the guarded command set in which C + D is executed for both guards. Thus in this execution of the guarded command set, the C + D in the second guard is redundant and should be replaced by a reference to the result of evaluated C + D in the first guard.



(a) Execution with no redundance (b) Execution with redundance

Figure 3.1 Two executions of a guarded command set.

3.2.2 The Use of Arithmetic Reasoning in Redundant Operation Recognition

In general, an operation is redundant if the information yielded by its evaluation may be obtained from existing information. Clearly, if the identical operation ρ occurs twice within a guarded command set and is evaluated twice within one execution of the guarded command set, the information obtained by the second evaluation of ρ is already known. This repetitions evaluation of identical operations is the only type of redundance addressed in the literature. (See Section 2.2.2.) Within the guarded command sets however, more redundant operations both occur and can be recognized.

The guards of a guarded command set are boolean expressions. Each boolean expression is built from smaller boolean expressions that are combined with the logical connectors - <u>and</u>, <u>or</u>, <u>cand</u>, and <u>cor</u>. Those boolean expressions that contain no logical connector are called <u>boolean expression factors</u>, abbreviated BE-factors. These BE-factors are either relations or boolean-valued variables. Using some elementary arithmetic reasoning and rules of inference, the known values of certain BE-factors can be used to determine, without further evaluation, the value of another BE-factor. Hence the definition of a redundant operation within an execution of a guarded command set may be broadened to the following. The occurrence of an operation ρ in a guard g_j is redundant within a particular execution of a guarded command set if:

(1) g_j is evaluated after guards g_{i_k}, $1 \leq k \leq n$, are evaluated,

(2) operations δ_ℓ, $1 \leq \ell \leq m$, are evaluating during this evaluation of guards g_{i_k}, $1 \leq k \leq n$,

(3) ρ is evaluated during this evaluation of g_j, and

(4) the value of ρ may be deduced from the values of δ_ℓ, $1 \leq \ell \leq m$.

Note that ρ may be a long boolean expression that is composed of BE-factors whose values can be deduced.

The types of deductions that should be considered in determining the values of BE-factors may be divided into five categories. First, the value of certain BE-factors may be deduced from the value of one other BE-factor. For example, if a < b is known to be true, then we may deduce that a $\leq$ b is true. Figure 3.2 gives a list of some of these single hypothesis deductions. The second category are those deductions that require a two-part hypothesis, but no use of transitivity. Three instances of these deductions are: (1) a $\leq$ b and a $\neq$ b $\Rightarrow$ a < b, (2) a $\geq$ b and a $\neq$ b $\Rightarrow$ a > b, and (3) a $\leq$ b and a $\geq$ b $\Rightarrow$ a = b.

The third and fourth types of deductions require the use of transitivity. Suppose that the following relations are known to be true: a = b, b = c, c = d, d = e. Using transitivity, we may deduce a = e. This type of deduction is called equality reasoning. Along the same line, suppose that the following relations are known to be true: a < b, b $\leq$ c, c $\leq$ d, d < e. Using transitivity, we may deduce a < e. This type of deduction is called linear order reasoning.

Finally, when the operands of relations are restricted to variables of type integer, the following monotonicity axioms may be

$a<b$ true $\Rightarrow$ $a<b$ true	$a>b$ true $\Rightarrow$ $a<b$ false	$a=b$ true $\Rightarrow$ $a<b$ false
$a\leq b$ true	$a\leq b$ false	$a\leq b$ true
$a=b$ false	$a=b$ false	$a=b$ true
$a\geq b$ false	$a\geq b$ true	$a\geq b$ true
$a>b$ false	$a>b$ true	$a>b$ false
$a\neq b$ true	$a\neq b$ true	$a\neq b$ false

$a\geq b$ true $\Rightarrow$ $a\geq b$ true	$a\leq b$ true $\Rightarrow$ $a\leq b$ true	$a\neq b$ true $\Rightarrow$ $a\neq b$ true
$a<b$ false	$a>b$ false	$a=b$ false

β true $\Rightarrow$ β true	not β true $\Rightarrow$ not β true
not β false	β false

where a and b are real- or integer-valued variables and β is a boolean-valued variable.

Figure 3.2 Single-hypothesis deductions

applied in making deductions:

(1) $x \geq y$ **and** $c_1 \geq c_2 \Rightarrow x + c_1 \geq y + c_2$ (monotonicity of +)

(2) $x \geq 0$ **and** $c_1 \geq c_2 \Rightarrow x * c_1 \geq x * c_2$ (monotonicity of *)

where c_1 and c_2 are integer constants.

Consider the conventional IF-statement given in Figure 3.3a. One representation of this IF-statement in the guarded command construct is given in Figure 3.3b. Notice that the IF-statement requires the evaluation of only one boolean expression. Assuming the guards are evaluated in the given order, the guarded command alternative statement requires the evaluation of two boolean expressions whenever the first guard is false. However using the single hypothesis deductions, the falseness of $x < y$ implies the truth of $x \geq y$ and the falseness of $y < z$ implies the truth of $y \geq z$. Thus without further evaluation, the truth of the second guard may be deduced from the falseness of the first guard. Hence SL_2 may be executed immediately upon determining that the first guard is false. This illustrates the importance of using arithmetic reasoning in recognizing redundant operations. Without it, the execution of the guarded command constructs could not be implemented as efficiently as the execution of the conventional constructs.

<u>if</u> $x < y$ <u>or</u> $y < z$	<u>if</u> $x < y$ <u>or</u> $y < z \rightarrow SL_1$
<u>then</u> SL_1	$\square\ x \geq y$ <u>and</u> $y \geq z \rightarrow SL_2$
<u>else</u> SL_2	<u>fi</u>
(a) Conventional representation	(b) Guarded command representation

Figure 3.3. Two representations of the IF-THEN-ELSE construct.

3.2.3 Further Redundant Operation Recognition in the Iterative Statement

Execution of an iterative statement requires repeated execution of the guarded command set until all the guards are found to be false. In all but the last execution of the guarded command set, several guards may be evaluated and one statement list SL_k (say) is executed. Some of the BE-factors of each of these guards are evaluated in evaluating the guards. If the operands of a BE-factor are not changed in executing SL_k, then the value of the BE-factor is not changed and need not be reevaluated during the next execution of the guarded command set. In general, the value of a BE-factor β is known after completing an execution of the guarded command set if and only if:

(1) the value of β is known before the execution of the guarded command set or is evaluated during this execution, and

(2) none of the operands of β are changed by the statement list executed during this execution of the guarded command set.

Consider the following program segment:

```
i, j, k := 0, 0, 0;
do f(i) < g(j) → i := i + 1
[] g(j) < h(k) → j := j + 1
[] h(k) < f(i) → k := k + 1
od
```

Suppose that in some execution of the guarded command set in the iterative statement the first two guards are determined to be false, but the third is true. When the statement list k := k + 1 is executed and the guarded command set is executed again. Note that neither of the operands in the BE-factor $f(i) < g(j)$ are changed in executing the third statement list. Hence the value of this BE-factor must still be false and we may skip over the entire first guard in the next execution of the guarded command set.

3.3 More on Ordering

Two more criteria for ordering the guards of a guarded command set may be suggested. First, the guards may be ordered to maximize the number of BE-factors whose values may be deduced using the arithmetic reasoning of Section 3.2.2. Secondly, the guards of an iterative statement may be ordered to maximize the number of BE-factors whose values in one execution of the guarded command set are known from the preceding executions of the guarded command set. (See Section 3.2.3). Measures for ordering the guards according to either of these two criteria are dependent upon the implementation of the optimizations discussed in Sections 3.2.2 and 3.2.3.

Chapter 4

The Problem

In the preceding chapter, the redundant operations that occur within the guards of a GCS are divided into three types. Type 1 are redundant operations that are identical repetitions of operations already evaluated in the current execution of the GCS. Type 2 refers to BE-factors whose values may be deduced, using arithmetic reasoning, from the values of BE-factors already evaluated in the current execution of the GCS. Finally, Type 3 refers to the BE-factors in the GCS of an iterative statement whose values in the current execution of the GCS are known from previous executions of the GCS.

Each of these three types have one important point in common. Each is defined in terms of a single execution of the GCS. This is one major difference between these types of redundant operations and the conventional type discussed in the literature. The conventional type is not defined in terms of a single execution path through the program. Rather, all execution paths through an occurrence of an operation must be considered in determining whether this occurrence is redundant. Specifically, an operation ρ at some point P'_ρ of a program is redundant if and only if:

(1) ρ is evaluated at some point P^i_ρ on each execution path ξ^i through P'_ρ, and

(2) the operands of ρ are not changed between each P^i_ρ and P'_ρ.

If ρ at P'_ρ is redundant according to this conventional definition, then the result of evaluating ρ at each point P^i_ρ may be saved in some common temporary T and the evaluation of ρ at P'_ρ may be replaced by a reference to T. In this manner, a conventional redundant operation is eliminated from the program. Unfortunately, this method of elimination does not work for the three new types of redundant operations.

In Section 3.2.1 we noted that a single occurrence of an operation ρ in a guard g_j could be redundant in one execution ξ^R of the GCS, but nonredundant in another execution ξ^N of the same GCS. A new method must be developed to eliminate an operation that is redundant in some particular execution of the GCS while maintaining the semantics of the guarded command construct for all possible executions of the GCS. The remaining chapters present and discuss such a method.

The method to be discussed addresses the particular problem of recognizing and eliminating BE-factors whose values may be deduced using the single-hypothesis deductions. The reason for choosing this particular subset of the new types of redundant operations is two-fold. First of all, the redundant operation of Types 1 and 3 may be recognized and eliminated by a simplified version of the method to be presented or by a combination of this simplified version with existing data flow analysis techniques. Thus, in presenting this method, all the new ideas needed for eliminating redundant operations of any of the three types from the guards of a GCS are introduced. Secondly, recognizing and eliminating this particular subset makes it possible to implement the guarded command alternative statement with efficiency comparable to any implementation of the conventional IF-THEN-ELSE construct.

Chapter 5

The Basic Approach

5.1 The DAG Representation of a GCS

A rooted binary directed acyclic graph (hereafter called a DAG) is a useful data structure for representing a GCS because it makes each possible execution of the GCS readily discernible. The root of the DAG is labeled with the first BE-factor of the first guard to be evaluated. Each of the other interior nodes are labeled with the other BE-factors of the GCS. Each leaf is labeled with one of the statement lists or the guarded command construct terminator - i.e., fi or od.

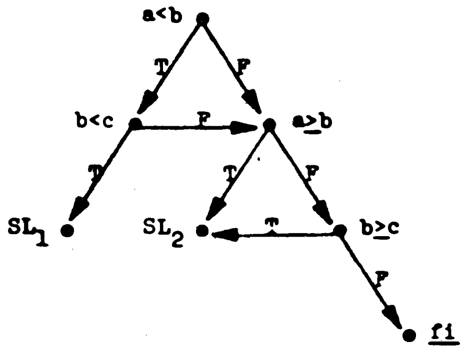
Let n be an interior node labeled with the BE-factor β. The left son of n (hereafter called the true son) is labeled with the part of the GCS that must be considered immediately after β when β is true. Similarly, the right son (hereafter called the false son) is labeled with the GCS part that must be considered immediately after β when β is false. Figure 5.1 illustrates the DAG corresponding to a guarded command construct.

Each possible execution of the OCS is represented by some path from the root of the DAG to some leaf. The nodes of such a path are labeled with precisely those BE-factors considered in the execution of the GCS and in the same order they are evaluated. Moreover, the value of each BE-factor β within this execution is indicated by the path by virtue of which son of β (i.e., the true or false son) is on the path.

<u>**if**</u> $a < b$ <u>**and**</u> $b < c \rightarrow SL_1$

$\square$ $a \geq b$ <u>**or**</u> $c \geq d \rightarrow SL_2$

<u>**fi**</u>



(a) Guarded command construct (b) DAG representation of the GCS

Figure 5.1. Illustration of the DAG representation of a GCS.

Some paths in the DAG, however, do not represent possible executions of the GCS. The rightmost path in Figure 5.1b is such a path, because it is not possible for both $a < b$ and $a \geq b$ to be false. Many of the paths corresponding to impossible executions will be eliminated from the final DAG representation of the GCS.

5.2 Splitting Paths

Let α^i be the i^{th} occurrence of a BE-factor α preceding the j^{th} occurrence β^j of the BE-factor β in some GCS. We would like to eliminate β^j from all executions of the GCS in which (1) α^i is evaluated before β^j and (2) the value of β can be deduced from the value of α. To understand how these eliminations may be realized, consider the DAG representing this GCS. The path through β^i may be divided into three types:

(1) paths along which the value of α^i implies the truth of β^j,

(2) paths along which the value of α^i implies the falseness of β^j, and

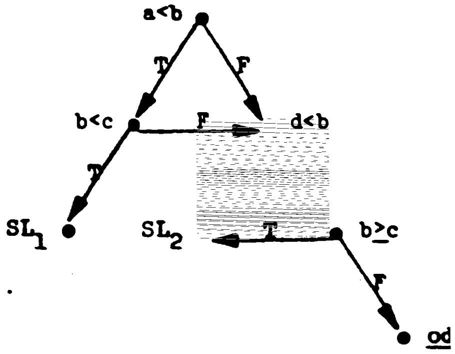
(3) paths along which the value of α^i does not imply a value for β^j or paths that do not pass through α^i.

Type 1 paths should be changed to skip over β^j to the true son of β^j. Type 2 paths should be changed to skip over β^j to its false son. Type 3 paths should remain unchanged. Unfortunately, these changes cannot always be applied to the initial DAG representation of the GCS. Paths of different types may merge before passing through β^j and thus no one of these changes to the initial DAG yields the desired result. Hence the initial DAG must be changed by splitting apart paths of different types so that each type may be dealt with separately. This splitting is accomplished simply by making a copy of each node between α^i and β^j, not including β^j, that lies on paths of different types.

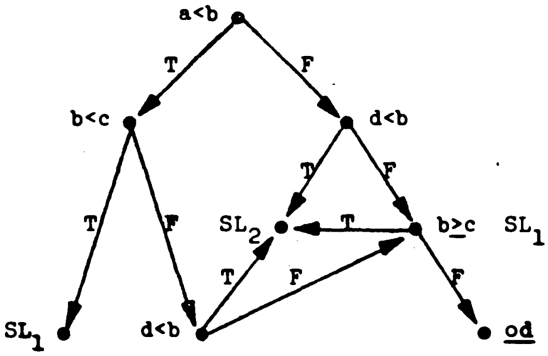
Consider, for example, the DAG given in Figure 5.2b, which represents the GCS of the construct given in Figure 5.2a. Let α^i be the only occurrence of BE-factor $b < c$ and β^j be $b \geq c$. Let p_1 be the path passing through $a < b$, $b < c$, $d < b$, $b \geq c$, SL_2; let p_2 be the path passing through $a < b$, $d < b$, $b \geq c$, od. Path p_1 is a Type 1 path and p_2 is a Type 3 path. These paths merge at $d < b$ and must be split apart by copying this node, as indicated in Figure 5.2c. Now p_1 can be changed to skip over $b \geq c$ to the true son SL_2. Path p_2 must remain unchanged since the value of $b \geq c$ is not yet known. The final result of making all possible deductions about the value of $b \geq c$ from the value of $b < c$ is given in Figure 5.2d. (Note that the node $d < b$ on path p_1 can also be eliminated later.)

```
do  a < b and b < c → SL1
[]  d < b or  b ≥ c → SL2
od
```

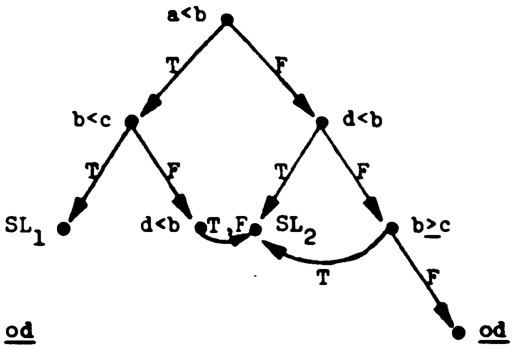
(a) Guarded command construct



(b) Initial DAG representation



(c) DAG with paths split



(d) DAG with redundant b≥c remove

Figure 5.2. Illustration of redundant operation elimination by splitting paths

5.3 A General Description of the Redundant Operation Recognition and Elimination Algorithm

Each path of the DAG that corresponds to a valid execution of the GCS is traversed to locate and eliminate redundant operations. Let p_i be the path currently being traversed. Let n_c be the node on p_i currently being considered. A truth table, called TT, contains a description of this path, together with the values of the nonredundant BE-factors that have been "evaluated" along p_i between the root and n_c.

If n_c is labeled with a statement list or a guarded command terminator, then traversal of p_i is complete. Otherwise n_c is labeled with some BE-factor β. If the value of β cannot be deduced from any BE-factor in TT, then either β or not β is added to TT, depending on whether the true son or false son of n_c is on p_i, respectively. If the value of β can be deduced from some BE-factor α in TT, then the DAG is changed to reflect all possible deductions about the value of β at n_c from this occurrence of α. If the value of β along p_i agrees with the value that α implies for β, then traversal of p_i is continued. If, however, the value of β contradicts the value implied by α, then p_i does not represent a valid execution of the GCS. In this case, the changes just made to the DAG have eliminated p_i as a possible path; traversal of p_i is terminated.

Chapter 6

General Time Analysis

The method of eliminating redundant operations just described may be subdivided into two processes. One is the process of finding all applications of the single-hypothesis deductions. The other is the process of changing the DAG to reflect these deductions. The first process requires traversing paths through the DAG. A constant bounds the amount of work that must be done at each node of each path in order to determine whether a single-hypothesis deduction may be applied. Since the length of each path in the DAG must be less than or equal to the total number of BE-factors n (say) in the GCS, a generous upper bound for this process is $C \cdot n \cdot P_{DAG}$, where P_{DAG} is the number of paths through the DAG and C is some constant.

Along a path of length m, at most m-1 deductions can be made. Thus a very generous upper time bound on the second process is $D \cdot n \cdot P_{DAG}$, where D is an upper bound on the amount of time spent changing the DAG to reflect one deduction. The method presented for changing the DAG is bounded by a function that is linear in n. Hence the upper time bounds of both processes are largely determined by the number of paths through the DAG.

6.1 Upper Bounds on the Number of Paths Through the DAG

It can be shown that the number of paths through <u>most</u> DAGs is bounded by a polynomial in the total number of BE-factors in the corresponding GCS. Moreover, the following theorem enables us to derive a bound on the number of paths through any DAG representing a GCS.

Theorem 6.1 If the length of the longest path in a binary rooted DAG is h, then there are at most 2^h distinct paths through the DAG.

Proof A rooted binary DAG can be represented as a binary tree simply by splitting apart all merging paths through the DAG. The number of paths through this binary tree, as well as their lengths, are the same as in the original DAG. Each path of a binary tree terminates at a distinct leaf. Since a complete binary tree of height h has 2^h leaves, a binary tree with the longest path of length h has at most 2^h leaves and paths. Hence a binary rooted DAG with the longest path of length h has at most 2^h paths.

Each BE-factor whose value may be deduced along a path through the DAG is immediately removed from that path. Thus the length of the longest path through the DAG is at most n_{ND}, where n_{ND} is the number of BE-factors in the GCS whose values cannot be determined using the single-hypothesis deductions and the value of another BE-factor of the GCS. Hence the number of paths through the DAG is bounded by a single exponential in n_{ND}. The value of n_{ND} in any reasonable guard is very small and so the total number of paths remains small enough to make the method practical.

6.2 Upper Bounds on the Runtime

In summary, the method for eliminating redundant operations described in the preceding chapter can be implemented to run in $O(n^c)$ time for most GCSs, where n is the number of BE-factors in the GCS and c is some constant. Furthermore, we have shown that the implementation can run in $O(2^{n_{ND}})$ time for any GCS, where n_{ND} is the number of BE-factors whose values cannot be deduced from the value of another BE-factor in the GCS. Both values n and n_D are usually so small that implementation is practical. The details of such an implementation are given in subsequent chapters.

Chapter 7

A More Detailed Description of the Algorithm

7.1 Pre-Processing

A compiler usually translates the source program into some internal form, e.g., quadruples. When translating guarded command sets, however, the compiler must perform some additional processing before it may invoke REMOVE_REDUNDANCY. It must build the DAG representation of the GCS. Each node in the DAG corresponds to a GCS-part and the GCS-part is described by a PASCAL record. The fields within this record are: NODE_NUM, TIPE, TRUE_SON, FALSE_SON, CANON[TTRUE], CANON[FFALSE], MARKERS, FATHERS, FIRST_QUAD and BTF_QUAD. We assume that these variables are initialized for each GCS-part in the guarded command set before REMOVE_REDUNDANCY is invoked.

Each node α of the DAG (and hence each GCS-part) is assigned a node number n such that n is larger than the node number n' of any ancestor α' of α. This ordering of nodes is called a <u>topological</u> ordering. NODE_NUM is the topological node number assigned to the corresponding GCS-part. TIPE indicates whether this GCS-part is a BE-factor, statement list or terminator, i.e., <u>fi</u> or <u>od</u>. TRUE_SON (FALSE_SON) is a pointer to the DAG node encountered upon following the true (false) branch from this GCS-part. CANON[TTRUE] (CANON[FFALSE]) is a pointer to a record describing the canonical representation of the GCS-part when it is true (false). MARKERS is a set of markers that is initially empty. When a BE-factor is removed from some paths through the DAG, the markers in this set indicate the type(s) of path(s) on which this GCS-part lies.

FATHERS is a pointer to a linked list of the fathers of this GCS-part within the DAG. FIRST_QUAD is a pointer to the first quadruple in a sequence of quadruples representing this GCS-part. BTF_QUAD is a pointer to the quadruple associated with this GCS-part whose operator is BTF.

In addition to building and initializing this DAG, the compiler must also provide: (1) the number of BE-factors in the GCS, (2) the number of GCS-parts, (3) a pointer to the root of the DAG, and (4) the size of the table in which BE-factors on the current path are stored (to be described subsequently).

7.2 Path Traversal and Description

Paths through the DAG are traversed in search of redundant operations. The root R of the DAG is assumed true, i.e., the BE-factor associated with R is assumed true, and the paths that lie within the subDAG whose root is the true son R_T of R are traversed. After these paths are traversed, R is assumed false and the paths that lie within the subDAG whose root is the false son R_F of R are traversed. To traverse the paths in the subDAG with root R_T, R_T is assumed true and the paths within the subDAG whose root is the true son of R_T are traversed. Then R_T is assumed false and the paths within the subDAG whose root is the false son of R_T are traversed. Similarly, traversal proceeds in this recursive manner.

As a path is traversed, a description of the path from the root to the node currently being traversed is maintained in the hash table TT. For each path node corresponding to a nonredundant BE-factor β (say), TT contains the truth value of β on this path and a pointer to the record that describes β. The location within TT at which this information is stored is determined by hashing the canonical form that represents β and its truth value on the current path. For example, if β is $a \leq b$ and β is true, then the canonical form of $a \leq b$ is hashed into TT. If β is false, then $a > b$ is hashed into TT.

Consider the DAG given in Figure 7.2. It represents the GCS given in Figure 7.1 which is an abbreviation of the algorithm used in procedure DELETE_MIN_FROM_HEAP in Chapter 8. The leftmost path is the first path traversed. Hence TT contains pointers to the records describing the nodes labeled $RS \leq N$, $A \leq B$ and $A \leq C$ and notes that these BE-factors are true on this current path. SL_1 indicates the end of this path. The paths that lie within the subDAG with root $LS = N$ are traversed next. Before they may be traversed, however, the description of the path within TT must be updated. Notice that only the value of the last BE-factor on the path, i.e., $A \leq C$, changes. Hence only the entry in TT that indicates $A \leq C$ is true must be removed and $A \leq C$ is false must be added. Thus this recursive traversal of the DAG minimizes the updating of the path description. The next path to be traversed passes through nodes labeled with $RS \leq N$, $A \leq B$, $A \leq C'$, $LS = N$, $B \leq A$, and $B \leq C$.

$$
\begin{array}{ll}
\underline{\text{do}}\ RS \leq N\ \underline{\text{cand}}\ A \leq B\ \underline{\text{cand}}\ A \leq C & \rightarrow SL_1 \\
\square\ (LS = N\ \underline{\text{or}}\ RS \leq N)\ \underline{\text{cand}}\ B \leq A\ \underline{\text{cand}}\ B \leq C & \rightarrow SL_2 \\
\underline{\text{od}} &
\end{array}
$$

Figure 7.1 Guarded command for reestablishing the heap property.

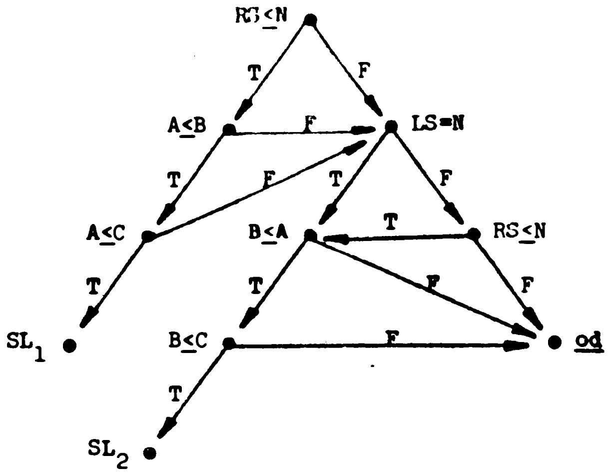


Figure 7.2 DAG representation of GCS in Figure 7.1.

7.3 Looking For an Implier

As each node α on the current path is traversed, we must determine whether the BE-factor β associated with α is redundant. Procedure LOOK_FOR_IMPLIER determines this by searching TT for BE-factors β_i that do imply a truth value for β. The canonical form of each β_i is hashed into TT and compared with the BE-factors, if any, at this location of TT. If β_{i_0} (say) is identical to one of the BE-factors β' (say) described in TT, then:

(1) IMPLIER_FOUND is set to TRUE to indicate that an implier has been found,

(2) IMPLIER is set to point to the record describing β',

(3) VALUE_IMPLIED is set to TTRUE or FFALSE, depending upon whether β' implies the truth or falseness of β,

(4) MAIN_SUBDAG is set to TTRUE or FFALSE, depending upon whether the current path follows the true or false branch out of the node labeled β', and

(5) SECOND_SUBDAG is set to TTRUE, FFALSE or NEITHER, depending upon the value of MAIN_SUBDAG and whether not β' implies a truth value for β.

Otherwise, if none of the β_i are identical to any BE-factor described in TT, then IMPLIER_FOUND is set to FALSE.

Suppose, for example, β is a boolean-valued variable B (say). There are only two BE-factors that imply a truth value for β - namely B and not B. First B is hashed into TT. If B is identical to any BE-factor described at this location, then the variables described above are initialized accordingly. Otherwise, not B is hashed into TT. If not B is identical to any BE-factor described at this location, then the variables described are initialized accordingly. Otherwise, IMPLIER_FOUND is set to FALSE.

As another more complex problem, suppose β is the relation $a \leq b$. The following BE-factors imply a truth value for this relation: $a \leq b$, $a < b$, $a = b$ and $a > b$. The canonical form of each of these BE-factors is hashed into TT until one is found to be identical to a BE-factor described in TT or it is established that none of them are identical to any BE-factor described in TT. The variables described above are initialized accordingly.

7.4 Marking

Assume that β is the node on the path that is currently being traversed and that β' is the BE-factor on the current path that implies a truth value for β. Henceforth, β' will be called CONSEQUENCE↑ and β called IMPLIER↑, as they are referred to in the procedures.

CONSEQUENCE† must be eliminated from all paths through IMPLIER† along which the value of CONSEQUENCE† may be deduced from the value of IMPLIER†. In order to do this, three types of paths through CONSEQUENCE† must be made distinguishable:

(1) paths along which the value of IMPLIER† implies the truth of CONSEQUENCE†,

(2) paths along which the value of IMPLIER† implies the falseness of CONSEQUENCE†, and

(3) paths along which the value of IMPLIER† does not imply a value for CONSEQUENCE† or paths that do not pass through IMPLIER†.

This is done by marking nodes on each of these paths with markers that indicate the type of the path. This marking of nodes is mainly done by procedures MARK_IMPLIER, MARK_IMPLYING_SUBDAGS, MARK and MARK_IMPLYING_REACHERS; however, for efficiency reasons, some marking is postponed until splitting time.

Before describing the marking that is done, a few terms must be introduced. The <u>subDAG between a node α_1 and one of its descendants α_2</u> is the set of all nodes α such that α is a descendant of α_1 and an ancestor of α_2 together with the set of DAG edges that connect these nodes. (Note that this subDAG necessarily includes α_1 and α_2.) The subDAG between the true (false) son of IMPLIER† and CONSEQUENCE† is called the true-subDAG (false-subDAG).

If the value of MAIN_SUBDAG is TTRUE, then the true-subDAG is the <u>main-subDAG</u>; otherwise MAIN_SUBDAG is FFALSE and the false-subDAG is the main-subDAG. Each node in the main-subDAG is marked with MAIN and REACHES. That is, these markers are added to the set of markers associated with each node in the main-subDAG. Similarly, if SECOND_SUBDAG is TTRUE (FFALSE), then the true-subDAG (false-subDAG) is the second-subDAG and each node in this subDAG is marked with SECOND and REACHES. If SECOND_SUBDAG is NEITHER, then no subDAG is marked with SECOND. In this last case, if MAIN_SUBDAG is TTRUE (FFALSE), then the false (true) son of IMPLIER↑ is marked with UNKNOWN.

Each descendant of the root of the main-subDAG that may be an ancestor of CONSEQUENCE↑ is marked with MAIN. Since any node with a node number greater than the node number of CONSEQUENCE↑ cannot be an ancestor of CONSEQUENCE↑, we mark with MAIN only those descendants whose node number is less than or equal to the node number of CONSEQUENCE↑. Similarly, we could mark with SECOND those descendants of the root of the second-subDAG whose node number is less than or equal to the node number of CONSEQUENCE↑. Conceptually, this is what is done. However, for efficiency reasons, only a subset of these nodes are actually marked with SECOND at this time. This subset determines all the other nodes that must be marked with SECOND, which are marked later when paths are split.

Each ancestor of CONSEQUENCE↑ that is marked MAIN and/or SECOND is marked with REACHES. Thus all and only those nodes marked with both MAIN and REACHES are in the main-subDAG; and conceptually, all and only those nodes marked with SECOND and REACHES are in the second-subDAG. Furthermore, we mark with UNKNOWN each ancestor α of CONSEQUENCE↑ that is marked with MAIN and/or SECOND and has a father f_α, $f_\alpha \neq$ IMPLIER↑, that is not marked at all.

If a node α, $\alpha \neq$ IMPLIER↑, is marked with MAIN and REACHES, then there is a path through IMPLIER↑, α and CONSEQUENCE↑ along which the value of CONSEQUENCE↑ is VALUE_IMPLIED. If α is marked with SECOND and REACHES, then there is a path through IMPLIER↑, α and CONSEQUENCE↑ along which the value of CONSEQUENCE↑ is not VALUE_IMPLIED. Finally, if α is marked with UNKNOWN, then there is a path through α and CONSEQUENCE↑ along which the value of CONSEQUENCE↑ cannot be deduced from the value of IMPLIER↑.

Figure 7.3 indicates how the DAG given in Figure 7.2 is marked when IMPLIER↑ is the root, labeled RS≤N, and CONSEQUENCE↑ is the interior node labeled RS≤N. The dashed line indicates the current path and the nodes are labeled with the appropriate markers instead of the BE-factors. M, S and R abbreviate MAIN, SECOND and REACHES, respectively.

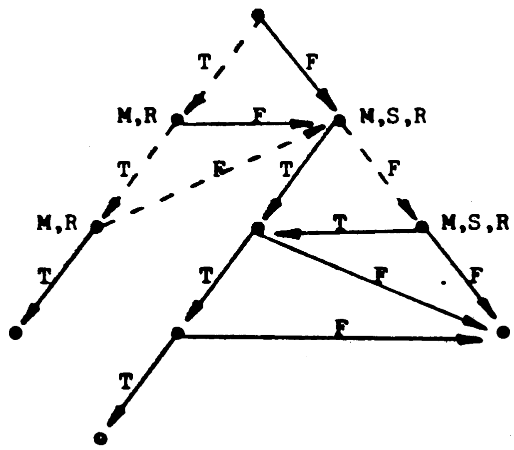


Figure 7.3 Illustration of marking nodes.

7.5 Splitting Paths by Splitting Nodes

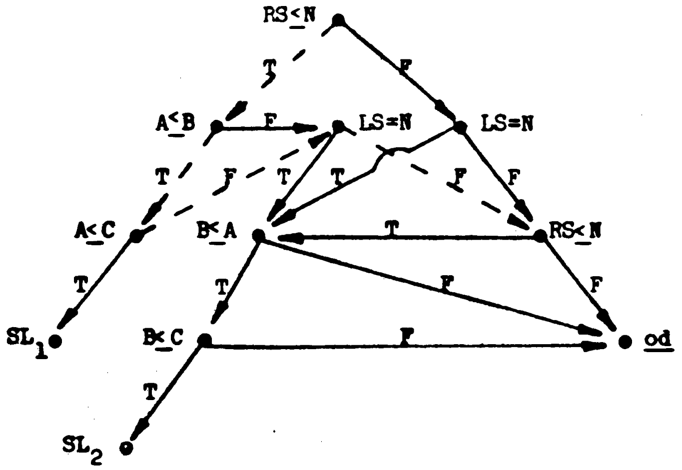
The markers [illegible], SECOND and UNKNOWN indicate the type of path(s) on which a node lies; hence they are called the path indicating markers. Each node α, $\alpha \neq$ CONSEQUENCE↑, that is marked with more than one path indicating marker lies on paths of different types. These nodes are split in order of increasing node number by procedure SPLIT_PATHS. The splitting process consists of two steps. First, a copy of α is made for each of its path indicating markers in excess of one. Each instance of α (i.e., the original and each copy) is marked with one of these path indicating markers. Hence each instance of α is marked with only one path indicating marker and each of the path indicating markers initially associated with α is now associated with some instance of α. Second, the DAG and quadruples must be changed so that each father f_α of α branches to the appropriate instance of α. Each f_α that is marked with m is made the father of that instance of α that is marked m and this instance of α is made the son of f_α in place of the original α. (f_α = IMPLIER↑ is handled separately.)

Consider the DAGs given in Figures 7.2 and 7.3. There are two types of paths from IMPLIER↑, i.e., the root labeled $RS \le N$, to CONSEQUENCE↑, i.e., the interior node labeled $RS \le N$, that pass through the node labeled $LS=N$. This is indicated by the fact that the node labeled $LS=N$ is marked with both M (for MAIN) and S (for SECOND). Along some paths, e.g., ones passing through nodes labeled $RS \le N$, $A \le B$, $A \le C$, $LS=N$ and $RS \le N$, the value of the interior $RS \le N$ may be deduced to be true. Along other paths, e.g., ones passing through $RS \le N$, $LS=N$ and $RS \le N$, the value of the interior $RS \le N$ may be deduced to be false. We split apart these two types of paths by splitting the node labeled LS=N into two nodes, one for each type of path. The DAG edges (as well as the corresponding quadruples) are changed so that all paths labeled with M pass through the original node labeled LS=N and all paths labeled S pass through the copy node labeled LS=N. These changes to the DAG are indicated in Figure 7.4. The dashed line indicates the current path.

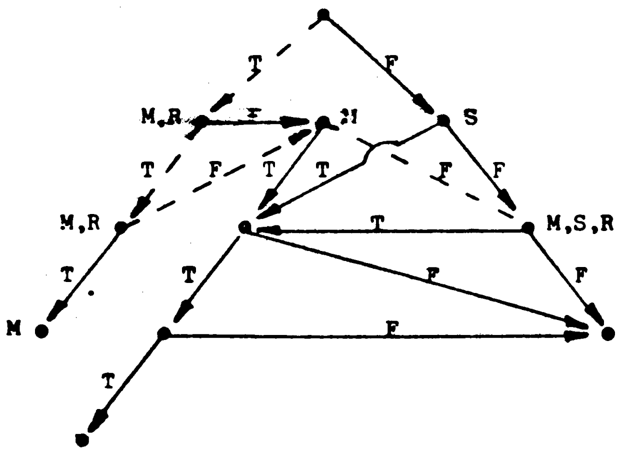
Once all paths of different types have been split apart, CONSEQUENCE↑ may be removed from all paths along which the value of CONSEQUENCE↑ may be deduced from IMPLIER↑. This is done by making each father of CONSEQUENCE↑ that is marked MAIN the father of the VALUE_IMPLIED son of CONSEQUENCE↑ instead of the father of CONSEQUENCE↑. Similarly, each father of CONSEQUENCE↑ that is marked SECOND is made the father of the <u>not</u> VALUE_IMPLIED son of CONSEQUENCE↑ instead of the father of CONSEQUENCE↑. (The case where IMPLIER↑ is a father of CONSEQUENCE↑ is handled separately.) These final changes to the DAG and the corresponding quadruples are done by procedure SKIP_OVER. Figure 7.5 depicts the DAG given in Figure 7.2 after the interior node labeled RS≤N is removed from all paths along which its value is strictly determined by the value of the RS≤N at the root of the DAG.

7.6 Cleanup

During the process of removing a redundant BE-factor from one or more execution paths, several nodes of the DAG are marked with markers. These markers must be removed before traversal of a path may continue. Procedure REMOVE_MARKS does this by traversing that subDAG that is marked and setting the set of markers associated with each node in this subDAG to [ ], i.e., the null set.



(a) DAG with nodes marked with corresponding GCS-parts



(b) DAG with nodes marked with corresponding markers

Figure 7.4 Illustration of splitting a node.

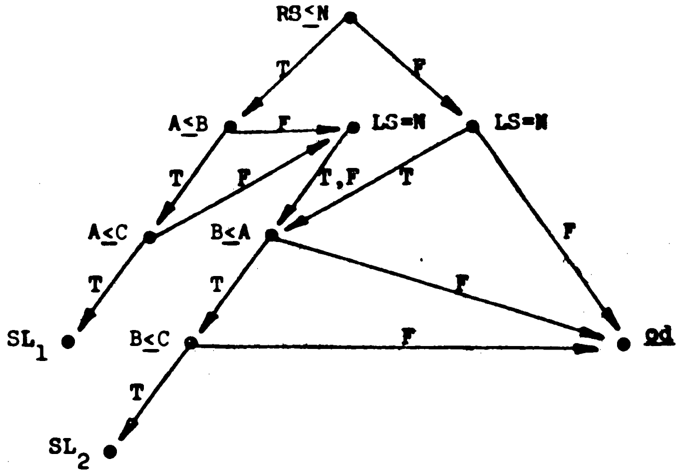


Figure 7.5 Illustration of removing a redundant BE-factor.

Chapter 8

The Algorithms

The following algorithms are written in a PASCAL that is supplemented with the guarded command constructs.

```
program  REMOVE_REDUNDANCY(input);

   constant  MAX_REP_LENGTH = {Maximum length of canonical representation
                               of a BE-factor.};

   type
      {NODE_NUM is topological node number of GCS_PART.  TIPE indicates
       whether GCS_PART is a BE-factor, statement list or terminator
       (i.e., fi or od).  TRUE_SON (FALSE_SON) is a pointer to the DAG
       node next encountered when GCS_PART is true (false).  CANON[TTRUE]
       (CANON[FFALSE]) is a pointer to a record describing the canonical
       representation of GCS_PART when it is true (false).  MARKERS is
       a set of markers that indicate the type(s) of path(s) on which
       this GCS_PART lies.  FATHERS is a pointer to a linked list of the
       fathers of GCS_PART within the DAG.  FIRST_QUAD is pointer to the
       first quadruple of the ordered quadruples representing GCS_PART.
       BTF_QUAD is the quadruple associated with GCS_PART whose operator
       is BTF.}

        GCS_PART = record
                     NODE_NUM:  1..NUM_GCS_PARTS;
                     case TIPE:  GCS_PART_TYPE of
                          BE_FACTOR:
                            (TRUE_SON, FALSE_SON: GCS_PART_PTR;
                             CANON: array [TRUTH_VALUE] of CANON_PTR;
                             MARKERS: MARKS;
                             FATHERS: LINK;
                             FIRST_QUAD, BTF_QUAD: QUAD_PTR);
                          SL, TERMINATOR:
                          end;

      GCS_PART_TYPE = (BE_FACTOR, SL, TERMINATOR);
      GCS_PART_PTR = ↑GCS_PART;
      TRUTH_VAL_OR_NEITHER = (TTRUE, FFALSE, NEITHER);
      TRUTH_VALUE = (TTRUE..FFALSE);
      CANON_PTR = ↑CANON_FORM;
```

```
    {REP is the canonical representation of GCS_PART.  FACTOR_TYPE
     indicates whether this GCS_PART is a relation or a boolean-valued
     variable.}
       CANON_FORM = record
                      REP: array [1..MAX_REP_LENGTH] of CHAR;
                      case FACTOR_TYPE: BE_FACTOR_TYPE of
                           REL: (OPER: RELOP);
                           BOOL_VAR:
                           end;
     BE_FACTOR_TYPE = (REL, BOOL_VAR);
     RELOPS_WITH_ENDMARK = (LT, LE, EQ, GE, GT, NE, ENDMARK);
     RELOPS = (LT..NE);
     MARKS = (MAIN, SECOND, REACHES, UNKNOWN);
     MARKS_SUBSET = (MAIN..SECOND);
     LINK = ↑LINKED_LIST;
     LINKED_LIST = record
                     VALUE: GCS_PART_PTR;
                     NEXT:  LINK
                     end;
     QUAD_PTR = ↑QUAD;
     QUAD = record
              OPERATOR: QUAD_OP;
              OPERAND1: SYMBOL_TABLE_ENTRY_PTR;
              case  QT: QUAD_TYPE of
                    BTF_TYPE: (BT, BF: QUAD_PTR);
                    NON_BTF:  (OPERAND2, RESULT:
                                 SYMBOL_TABLE_ENTRY_PTR)
                    end;
     QUAD_OP = (B, BTF, CVIR, LESS_THAN, LESS_EQUAL, EQUAL,
                GREATER_EQUAL, GREATER, NOT_EQUAL, PLUS, MINUS,
                TIMES, DIVIDE, EXPO, ASSIGN);
     QUAD_TYPE = (BTF_TYPE, NON_BTF);

var NUM_BE_FACTORS {Number of BE-factors in the GCS},
    NUM_GCS_PARTS  {Sum of the number of BE-factors and the number of
                    statement lists in the GCS, plus one for the
                    terminator},
    SIZE_HASH_TABLE
                   : INTEGER;

    DAG_ROOT {Pointer to the root of the DAG representing the GCS}
           : GCS_PART_PTR;

begin
  {Retrieve values of variables that are calculated by the compiler.}
    GET_FROM_COMPILER(NUM_BE_FACTORS, NUM_GCS_PARTS, DAG_ROOT);

  {Compute initial size of hash table.}
    COMPUTE_SIZE(NUM_GCS_PARTS, SIZE_HASH_TABLE);

  {Perform redundancy elimination.}
    ALL_PATHS(DAG_ROOT);
```

```
procedure ALL_PATHS(CURR_GCS_PART: GCS_PART_PTR);

  {Search all consistent paths from CURR_GCS_PART↑ to the SLs and the
   terminator of the GCS for BE-factors whose values may be deduced from
   the values of preceding BE-factors.  Change the GCS quadruples and DAG
   to branch around such BE-factors where possible.}

  type TT_ENTRY = record
                    DAG_LOCATION: BE_FACTOR_PTR;
                    CURR_VALUE: TRUTH_VALUE;
                    LINK_FIELD: TT_INDEX
                    end;

       TT_INDEX = (Index of CURR_BE_FACTOR in TT)
                    (1..SIZE_HASH_TABLE);

  var  IMPLIER_FOUND (Boolean value returned by LOOK_FOR_IMPLIER
                      indicating whether an IMPLIER has been found.)
                     : BOOLEAN;

       IMPLIER (If IMPLIER_FOUND, then a pointer into the DAG to a
                BE-factor that implies a truth value for CURR_GCS_PART↑.
                Otherwise undefined.)
               : BE_FACTOR_PTR;

       MAIN_SUBDAG (If IMPLIER_FOUND, then indicates whether the true or
                    false son of IMPLIER↑ is the root of the subDAG to
                    be marked with MAIN.  Otherwise undefined.)
                   : TRUTH_VALUE;

       SECOND_SUBDAG (If IMPLIER_FOUND, then indicates whether the true
                      or false son or neither of these is the root of
                      subDAG to be marked with SECOND.  A value of
                      NEITHER indicates that no subDAG is to be marked
                      with SECOND.  If not IMPLIER_FOUND, then undefined.)
                     : TRUTH_VAL_OR_NEITHER;

       VALUE_IMPLIED (If IMPLIER_FOUND, then a truth value for
                      CURR_GCS_PART known at each BE-factor to be
                      marked MAIN.  Otherwise undefined.)
                     : TRUTH_VALUE;

       TT (Each entry in this hash table contains: (1) a pointer into
           the DAG to a BE-factor β on the current path, (2) TTRUE or
           FFALSE depending on whether β is true or false along the
           current path, and (3) a link field if a bucket hash is used.)
          : array [1..SIZE_HASH_TABLE] of TT_ENTRY;
```

```
begin
  if CURR_GCS_PART↑.TIPE = BE_FACTOR

     → (Look in TT for a BE-factor IMPLIER↑ that implies a truth value
        for CURR_GCS_PART↑.  If such a BE-factor is found, change the
        quadruples and DAG to reflect this and continue searching appro-
        priate subDAG.  Otherwise, update TT to indicate the presence
        of CURR_GCS_PART↑ along the current path and search the true
        subDAG of CURR_GCS_PART↑ for deducible BE-factors.  Then change
        TT to indicate the presence of not CURR_GCS_PART↑ and search
        the false subDAG.  Finally return TT to its form upon entry.)

          LOOK_FOR_IMPLIER(CURR_GCS_PART, IMPLIER_FOUND, IMPLIER,
                           VALUE_IMPLIED, MAIN_SUBDAG, SECOND_SUBDAG);

          if IMPLIER_FOUND cand VALUE_IMPLIED = TTRUE

             → (IMPLIER↑ implies the truth of CURR_GCS_PART↑.  Change
                quads and DAG to reflect this and then process the true
                subDAG of CURR_GCS_PART↑.)
                  CHANGE_DAG(CURR_GCS_PART, IMPLIER, VALUE_IMPLIED,
                             MAIN_SUBDAG, SECOND_SUBDAG);
                  ALL_PATHS(CURR_GCS_PART↑.TRUE_SON)

          [] IMPLIER_FOUND cand VALUE_IMPLIED = FFALSE

             → (not IMPLIER↑ implies the falseness of CURR_GCS_PART↑.
                Change quads and DAG to reflect this and then process
                the false subDAG of CURR_GCS_PART↑.)
                  CHANGE_DAG(CURR_GCS_PART, IMPLIER, VALUE_IMPLIED,
                             MAIN_SUBDAG, SECOND_SUBDAG);
                  ALL_PATHS(CURR_GCS_PART↑.FALSE_SON)

          [] not IMPLIER_FOUND

             → (No implying BE-factor has been found.  Put
                CURR_GCS_PART into TT with its current truth value
                TTRUE and process the true subDAG.  Remove this last
                entry from TT, put CURR_GCS_PART into TT with its
                current truth value FFALSE and process the false
                subDAG.  Remove this last entry from TT.)
                  PUT_IN_TT(CURR_GCS_PART, TTRUE, TT_INDEX);
                  ALL_PATHS(CURR_GCS_PART↑.TRUE_SON);
                  REMOVE_FROM_TT(TT_INDEX);
                  PUT_IN_TT(CURR_GCS_PART, FFALSE, TT_INDEX);
                  ALL_PATHS(CURR_GCS_PART↑.FALSE_SON);
                  REMOVE_FROM_TT(TT_INDEX)
          fi

  [] CURR_GCS_PART↑.TIPE ≠ BE_FACTOR
     → skip  (end of current path through DAG)
fi
```

```
procedure PUT_IN_TT(CURR_BE_FACTOR: BE_FACTOR_PTR; VALUE: TRUTH_VALUE;
                    var TT_INDEX: INTEGER);

   begin

     {If VALUE is TTRUE, hash CURR_BE_FACTOR↑ into hash table TT.
      Otherwise if VALUE is FFALSE, hash not CURR_BE_FACTOR↑ into hash
      table TT.  Return the location of this entry in TT in variablo
      TT_INDEX.}

   end;  {PUT_IN_TT}



procedure REMOVE_FROM_TT(TT_INDEX);

   begin

     {Remove the entry in hash table TT that is located at TT_INDEX.}

   end;  {REMOVE_FROM_TT}



procedure LOOK_IN_TT(CANON_REP: CANON_PTR; var IMPLIER_FOUND: BOOLEAN;
                     var IMPLIER: BE_FACTOR_PTR;
                     var MAIN_SUBDAG: TRUTH_VALUE);

   begin

     {Hash CANON_REP↑ into TT.  If CANON_REP↑ does not match a boolean
      expression stored at this location of TT, set IMPLIER_FOUND to
      FALSE and return.  If CANON_REP↑ matches a boolean expression
      at this location of TT, set IMPLIER_FOUND to TRUE.  Set IMPLIER to
      point to the location within the DAG of this matching boolean
      expression.  Set MAIN_SUBDAG to TTRUE or FFALSE depending on
      whether the true son or false son of IMPLIER↑ is on the path
      through the DAG that is currently being traversed.}

   end  {LOOK_IN_TT}

   end;  {ALL_PATHS}
```

```
procedure LOOK_FOR_IMPLIER(CURR_BE_FACTOR: BE_FACTOR_PTR;
                           var IMPLIER_FOUND: BOOLEAN;
                           var IMPLIER: BE_FACTOR_PTR;
                           var VALUE_IMPLIED, MAIN_SUBDAG: TRUTH_VALUE;
                           var SECOND_SUBDAG: TRUTH_VAL_OR_NEITHER);

  {Set IMPLIER_FOUND to "there exists a BE-factor in TT that implies
   CURR_BE_FACTOR↑ or not CURR_BE_FACTOR↑." If IMPLIER_FOUND, then set:
   (1) IMPLIER to point to the BE-factor within the DAG that implies the
   truth value, (2) VALUE_IMPLIED to the truth value implied for
   CURR_BE_FACTOR↑ along the current path, (3) MAIN_SUBDAG to TTRUE or
   FFALSE, depending on the truth value of IMPLIER↑ represented in TT, and
   (4) SECOND_SUBDAG to NEITHER if the truth value of IMPLIER↑ not
   represented in TT does not imply a value for CURR_BE_FACTOR↑;
   otherwise set SECOND_SUBDAG to TTRUE or FFALSE, depending on the
   value of not MAIN_SUBDAG.}

   var IMPLIES_TRUE {Let y_x represent a relation y with relational
                     operation x. IMPLIES[OPER] is the set of all
                     relational operations x such that y_x implies the
                     truth of y_OPER.},

       IMPLIES_FALSE {Analogous to IMPLIES_TRUE, except y_x implies the
                      falseness of y_OPER.}
                     : array [RELOPS] of set of RELOPS;

       OP {If CURR_BE_FACTOR↑ is a relation, the relational operator
           in this relation.}
          : RELOP;

       IMPLYING_OPS {If CURR_BE_FACTOR↑ is a relation, the set union
                     of IMPLIES_TRUE[OP] and IMPLIES_FALSE[OP].}
                    : set of RELOPS;

       IMPLYING_CANON {If CURR_BE_FACTOR↑ is a relation, pointer to
                       the canonical form of a relation R that implies
                       a truth value for CURR_BE_FACTOR↑.}
                      : CANON_PTR;

       CURR_OP {If CURR_BE_FACTOR↑ is a relation, the relational
                operator currently being considered for or used as
                the operator in IMPLYING_CANON.}
               : RELOPS_WITH_ENDMARK;
```

```
begin
  IMPLIER_FOUND := FALSE;
  if CURR_BE_FACTOR↑.FACTOR_TYPE = BOOL_VAR

     → {If CURR_BE_FACTOR↑ or not CURR_BE_FACTOR↑ are represented in
       TT, set IMPLIER_FOUND to TRUE and set the values of IMPLIER,
       VALUE_IMPLIED, MAIN_SUBDAG and SECOND_SUBDAG.}
         LOOK_IN_TT(CURR_BE_FACTOR↑.CANON[TTRUE], IMPLIER_FOUND,
                    IMPLIER, MAIN_SUBDAG);

         if IMPLIER_FOUND cand MAIN_SUBDAG = TTRUE
           → VALUE_IMPLIED := TTRUE;  SECOND_SUBDAG := FFALSE

         [] IMPLIER_FOUND cand MAIN_SUBDAG = FFALSE
           → VALUE_IMPLIED := TTRUE;  SECOND_SUBDAG := TTRUE

         [] not IMPLIER_FOUND
           → LOOK_IN_TT(CURR_BE_FACTOR↑.CANON[FFALSE], IMPLIER_FOUND,
                        IMPLIER, MAIN_SUBDAG);

             if IMPLIER_FOUND cand MAIN_SUBDAG = TTRUE
               → VALUE_IMPLIED := FFALSE;  SECOND_SUBDAG := FFALSE

             [] IMPLIER_FOUND cand MAIN_SUBDAG = FFALSE
               → VALUE_IMPLIED := FFALSE; SECOND_SUBDAG := TTRUE

             [] not IMPLIER_FOUND
               → skip
             fi
         fi

  [] CURR_BE_FACTOR↑.FACTOR_TYPE = REL

     → {Initialize IMPLIES_TRUE and IMPLIES_FALSE.}
         IMPLIES_TRUE[LT] := [LT];
         IMPLIES_TRUE[LE] := [LE,LT,EQ];
         IMPLIES_TRUE[EQ] := [EQ];
         IMPLIES_TRUE[GE] := [GE,GT,EQ];
         IMPLIES_TRUE[GT] := [GT];
         IMPLIES_TRUE[NE] := [NE,LT,GT];
         IMPLIES_FALSE[LT] := [EQ,GE,GT];
         IMPLIES_FALSE[LE] := [GT];
         IMPLIES_FALSE[EQ] := [NE,LT,GT];
         IMPLIES_FALSE[GE] := [LT];
         IMPLIES_FALSE[GT] := [EQ,LE,LT];
         IMPLIES_FALSE[NE] := [EQ];

       {Look in TT for the relations R that imply a truth value for
        CURR_BE_FACTOR↑.  If some such R is in TT, set IMPLIER_FOUND
        to true and set the values of VALUE_IMPLIED, MAIN_SUBDAG and
        SECOND_SUBDAG.}
         OP := CURR_BE_FACTOR↑.CANON[TTRUE]↑.OPER;
         IMPLYING_OPS := IMPLIES_TRUE[OP] + IMPLIES_FALSE[OP];
```

```
        COPY_CANON(CURR_BE_FACTOR↑.CANON[TTRUE], IMPLYING_CANON);
        CURR_OP := LT;  {Initialize CURR_OP to smallest RELOP.}

       {For each relational operator x in IMPLYING_OPS, set the
        operator in IMPLYING_CANON↑ to x.  Look in TT for
        IMPLYING_CANON↑ and if it is found, then set the values
        of IMPLIER_FOUND, IMPLIER, VALUE_IMPLIED, MAIN_SUBDAG and
        SECOND_SUBDAG.}

          do CURR_OP in IMPLYING_OPS and CURR_OP ≠ ENDMARK and
             not IMPLIER_FOUND
             → IMPLYING_CANON↑.OPER := CURR_OP;
               LOOK_IN_TT(IMPLYING_CANON, IMPLIER_FOUND, IMPLIER,
                          MAIN_SUBDAG);

              {If IMPLIER_FOUND, then determine value of
               SECOND_SUBDAG.}

                 if IMPLIER_FOUND cand MAIN_SUBDAG = TTRUE cand
                    IMPLIER↑.CANON[FFALSE]↑.OPER in IMPLYING_OPS
                    → SECOND_SUBDAG := FFALSE

                 [] IMPLIER_FOUND cand MAIN_SUBDAG = FFALSE cand
                    IMPLIER↑.CANON[TTRUE]↑.OPER in IMPLYING_OPS
                    → SECOND_SUBDAG := TTRUE

                 [] IMPLIER_FOUND cand
                    not (IMPLIER↑.CANON[TTRUE]↑.OPER in IMPLYING_OPS
                    cand IMPLIER↑.CANON[FFALSE]↑.OPER in
                    IMPLYING_OPS)
                    → SECOND_SUBDAG := NEITHER

                 [] not IMPLIER_FOUND
                      skip
                 fi;

              {If IMPLIER_FOUND, determine value of VALUE_IMPLIED.
               Otherwise set CURR_OP to next RELOP.}

                 if IMPLIER_FOUND and CURR_OP in IMPLIES_TRUE[OP]
                    → VALUE_IMPLIED := TTRUE
                 [] IMPLIER_FOUND and CURR_OP in IMPLIES_FALSE[OP]
                    → VALUE_IMPLIED := FFALSE
                 [] not IMPLIER_FOUND
                    → CURR_OP := SUCC[CURR_OP]
                 fi

          [] not (CURR_OP in IMPLIES_VALUE) and CURR_OP ≠ ENDMARK
             and not IMPLIER_FOUND
             → CURR_OP := SUCC[CURR_OP]
          od
   fi
end;  {LOOK_FOR_IMPLIER}
```

```
procedure CHANGE_DAG(CONSEQUENCE, IMPLIER: BE_FACTOR_PTR;
                     VALUE_IMPLIED, MAIN_SUBDAG: TRUTH_VALUE;
                     SECOND_SUBDAG: TRUTH_VAL_OR_NEITHER);

  {CONSEQUENCE↑ is known to be true or false, depending upon
   VALUE_IMPLIED, at each BE-factor X↑ reachable from the MAIN_SUBDAG
   son of IMPLIER↑ with X↑.NODE_NUM ≤ CONSEQUENCE↑.NODE_NUM.  If
   SECOND_SUBDAG is not NEITHER, CONSEQUENCE↑ is known to be true or
   false, depending upon not VALUE_IMPLIED, at each BE-factor Y↑ reachable
   from the SECOND_SUBDAG son of IMPLIER↑ with
   Y↑.NODE_NUM ≤ CONSEQUENCE↑.NODE_NUM.  Change DAG to delete BE-factor
   CONSEQUENCE↑ on all possible paths from IMPLIER↑ through CONSEQUENCE↑.}

   begin
     MARK_IMPLYING_SUBDAGS(IMPLIER, CONSEQUENCE, MAIN_SUBDAG,
                           SECOND_SUBDAG);
     MARK_IMPLYING_REACHERS(IMPLIER, CONSEQUENCE);
     SPLIT_PATHS(IMPLIER, CONSEQUENCE, MAIN_SUBDAG);
     SKIP_OVER(CONSEQUENCE, IMPLIER, VALUE_IMPLIED, MAIN_SUBDAG,
               SECOND_SUBDAG);
     REMOVE_MARKS(IMPLIER);
   end;  {CHANGE_DAG}



procedure MARK_IMPLYING_SUBDAGS(IMPLIER, CONSEQUENCE: BE_FACTOR_PTR;
                                MAIN_SUBDAG: TRUTH_VALUE;
                                SECOND_SUBDAG: TRUTH_VAL_OR_NEITHER);

  {Mark with MAIN all BE-factors X↑ reachable from the MAIN_SUBDAG son of
   IMPLIER↑ with X↑.NODE_NUM ≤ CONSEQUENCE↑.NODE_NUM.  If SECOND_SUBDAG is
   not NEITHER, mark with SECOND all BE-factors Y↑ reachable from the
   SECOND_SUBDAG son of IMPLIER↑ with X↑.NODE_NUM ≤ CONSEQUENCE↑.NODE_NUM.
   If SECOND_SUBDAG is NEITHER, mark with UNKNOWN that son of IMPLIER↑
   that is not marked with MAIN.}

   begin
     if MAIN_SUBDAG = TTRUE  → MARK(IMPLIER↑.TRUE_SON, CONSEQUENCE, MAIN)

     [] MAIN_SUBDAG = FFALSE → MARK(IMPLIER↑.FALSE_SON, CONSEQUENCE, MAIN)
     fi;

     if SECOND_SUBDAG = TTRUE
        → MARK(IMPLIER↑.TRUE_SON, CONSEQUENCE, SECOND)
     [] SECOND_SUBDAG = FFALSE
        → MARK(IMPLIER↑.FALSE_SON, CONSEQUENCE, SECOND)
     [] SECOND_SUBDAG = NEITHER and MAIN_SUBDAG = TTRUE
        → IMPLIER↑.FALSE_SON↑.MARKERS := [UNKNOWN]
     [] SECOND_SUBDAG = NEITHER and MAIN_SUBDAG = FFALSE
        → IMPLIER↑.TRUE_SON↑.MARKERS := [UNKNOWN]
     fi
   end; {MARK_IMPLYING_SUBDAGS}
```

```
procedure MARK(ROOT, CONSEQUENCE: BE_FACTOR_PTR; MARKER: MARKS_SUBSET);

  {Mark with MARKER each BE-factor X↑ reachable from ROOT with
   X↑.NODE_NUM ≤ CONSEQUENCE↑.NODE_NUM.}

   var CONSIDER_FOR_MARK {Pointer to the first element in a linked list
                          of pointers to GCS-parts to be considered for
                          marking},

       NEW_ENTRY {Pointer to space allocated for a new entry in the
                  CONSIDER_FOR_MARK list.}
                 : LINK;

       CURR_GCS_PART {Pointer to the GCS-part currently being considered
                      for marking.}
                     : GCS_PART_PTR;

   begin
     CONSIDER_FOR_MARK := nil; {CONSIDER_FOR_MARK is initially empty.}
     CURR_GCS_PART := ROOT;

     {Mark each GCS-part X↑ that is a BE-factor with X↑.NODE_NUM ≤
      CONSEQUENCE↑.NODE_NUM where X = CURR_GCS_PART or X is in
      CONSIDER_FOR_MARK. Also mark any BE-factor Y↑ reachable from
      such X↑ with Y↑.NODE_NUM ≤ CONSEQUENCE↑.NODE_NUM.}

       do CURR_GCS_PART↑.NODE_NUM ≤ CONSEQUENCE↑.NODE_NUM and
          CURR_GCS_PART↑.TIPE = BE_FACTOR cand
          card(CURR_GCS_PART↑.MARKERS) = 0

          → {CURR_GCS_PART has not yet been marked. Mark it with MARKER
             and indicate that its sons must be considered for marking.}

               CURR_GCS_PART↑.MARKERS := [MARKER];
               NEW(NEW_ENTRY);
               NEW_ENTRY↑.NEXT := CONSIDER_FOR_MARK;
               NEW_ENTRY↑.VALUE := CURR_GCS_PART↑.TRUE_SON;
               CONSIDER_FOR_MARK := NEW_ENTRY;
               CURR_GCS_PART := CURR_GCS_PART↑.FALSE_SON

       [] CURR_GCS_PART↑.NODE_NUM ≤ CONSEQUENCE↑.NODE_NUM and
          CURR_GCS_PART↑.TIPE = BE_FACTOR cand
          card(CURR_GCS_PART↑.MARKERS) ≠ 0 cand
          not (MARKER in CURR_GCS_PART↑.MARKERS)

          → {CURR_GCS_PART has been marked but not with MARKER. Mark it
             with MARKER.}

               CURR_GCS_PART↑.MARKERS := CURR_GCS_PART↑.MARKERS +
                                          [MARKER]
```

```
    [] (CURR_GCS_PART↑.NODE_NUM > CONSEQUENCE↑.NODE_NUM or
       (CURR_GCS_PART↑.TIPE = BE_FACTOR cand
        MARKER in CURR_GCS_PART↑.MARKERS) or
        CURR_GCS_PART↑.TIPE ≠ BE_FACTOR) and
        CONSIDER_FOR_MARK ≠ nil

        → {CURR_GCS_PART↑ does not qualify for marking.  Change
           CURR_GCS_PART to point to next GCS-part to be considered
           for marking.}

             CURR_GCS_PART := CONSIDER_FOR_MARK↑.VALUE;
             CONSIDER_FOR_MARK := CONSIDER_FOR_MARK↑.NEXT
    od

end;  {MARK}
```

```
procedure MARK_IMPLYING_REACHERS(IMPLIER, CONSEQUENCE: BE_FACTOR_PTR);

  {Mark with REACHES all BE-factors X↑ that reach CONSEQUENCE↑ and
   are marked MAIN and/or SECOND.  Mark with UNKNOWN all such BE-factors
   X↑ at which the value of CONSEQUENCE↑ may not be known.}

  var CONSIDER_FOR_MARK {Pointer to the first element of linked list
                         of pointers to BE-factors to be considered
                         for marking.},

      NEW_ENTRY {Pointer to space allocated for a new entry of the
                 CONSIDER_FOR_MARK list.},

      FATHER {Pointer to an entry in the FATHERS list associated with
              CURR_BE_FACTOR↑.}
             : LINK;

      CURR_BE_FACTOR {Pointer to the BE-factor currently being considered
                      for marking.}
                     : BE_FACTOR_PTR;

  begin
    CONSIDER_FOR_MARK := nil; {CONSIDER_FOR_MARK is initially empty.}
    CURR_BE_FACTOR := CONSEQUENCE;
```

```
{Mark with REACHES each BE-factor X↑ that is not already marked with
 REACHES, where X = CURR_BE_FACTOR or X is in list CONSIDER_FOR_MARK.
 Also mark with REACHES any BE-factor Y↑ that reaches such an X↑
 and is marked with MAIN and/or SECOND but not REACHES.}

   do not (REACHES in CURR_BE_FACTOR↑.MARKERS)

      → {CURR_BE_FACTOR↑ has not yet been marked REACHES.  Mark it and
         put into CONSIDER_FOR_MARK a pointer to each of its fathers
         that is marked MAIN and/or SECOND but not REACHES.}
          CURR_BE_FACTOR↑.MARKERS := CURR_BE_FACTOR↑.MARKERS +
                                     [REACHES];
          FATHER := CURR_BE_FACTOR↑.FATHERS;

          do FATHER ≠ nil
             → if card(FATHER↑.VALUE↑.MARKERS) ≥ 1 and
                  not (REACHES in FATHER↑.VALUE↑.MARKERS)

                  → {BE-factor FATHER↑.VALUE↑ qualifies for being
                     marked REACHES.  Put a pointer to it into
                     list CONSIDER_FOR_MARK.}
                      NEW(NEW_ENTRY);
                      NEW_ENTRY↑.NEXT := CONSIDER_FOR_MARK;
                      NEW_ENTRY↑.VALUE := FATHER↑.VALUE;
                      CONSIDER_FOR_MARK := NEW_ENTRY

               [] card(FATHER↑.VALUE↑.MARKERS) = 0 and
                  FATHER↑.VALUE ≠ IMPLIER

                  → {The value of CONSEQUENCE↑ is not known at
                     BE-factor FATHER↑.VALUE↑ and thus it may not be
                     known at its son CURR_BE_FACTOR↑.  Indicate
                     this by marking CURR_BE_FACTOR↑ with UNKNOWN.}
                      CURR_BE_FACTOR↑.MARKERS :=
                      CURR_BE_FACTOR↑.MARKERS + [UNKNOWN]

               [] REACHES in FATHER↑.VALUE↑.MARKERS

                  → {FATHER↑.VALUE↑ is already appropriately
                     marked REACHES.}
                      skip
               fi;
               FATHER := FATHER↑.NEXT
          od

   [] REACHES in CURR_BE_FACTOR↑.MARKERS and CONSIDER_FOR_MARK ≠ nil

      → {CURR_BE_FACTOR has already been marked REACHES.  Access
         next BE-factor to be considered for marking.}
          CURR_BE_FACTOR := CONSIDER_FOR_MARK↑.VALUE;
          CONSIDER_FOR_MARK := CONSIDER_FOR_MARK↑.NEXT
   od
end; {MARK_IMPLYING_REACHERS}
```

```
procedure SPLIT_PATHS(IMPLIER, CONSEQUENCE: BE_FACTOR_PTR;
                      MAIN_SUBDAG: TRUTH_VALUE);

  {Each path through CONSEQUENCE↑ may be one of three types - a path along
   which the value of CONSEQUENCE↑ is  (1) known to be true, (2) known to
   be false or (3) unknown.  Split apart paths of different types that
   merge by copying all common nodes between the merging point and
   CONSEQUENCE↑, not including CONSEQUENCE↑.}

  var CURR_BE_FACTOR (Pointer to the BE-factor currently being checked
                      for merging paths.}
                     : BE_FACTOR_PTR;

      CHECK_FOR_MERGE {A heap of pointer to BE-factors that must be
                       checked for merging paths.  The elements of the
                       heap are ordered by increasing node number.}
                      : array [1..NUM_BE_FACTORS] of BE_FACTOR_PTR;

      NUM_IN_HEAP {Number of elements in heap CHECK_FOR_MERGE.}
                 : INTEGER;

  begin
    NUM_IN_HEAP := 0;  {Heap initially empty.}

    {Add to CHECK_FOR_MERGE the son(s) of IMPLIER↑ that reach CONSEQUENCE↑
     and at which a value for CONSEQUENCE↑ is known.  Set CURR_BE_FACTOR
     to an entry in the heap with smallest node number.}

      ADD_APPROPRIATE_SONS_TO_HEAP(IMPLIER);
      CURR_BE_FACTOR := CHECK_FOR_MERGE[1];
      DELETE_MIN_FROM_HEAP;

    if CURR_BE_FACTOR ≠ CONSEQUENCE

      → {Mark IMPLIER↑ with MAIN or SECOND so that CONSIDER_FOR_SPLIT
         can make IMPLIER↑ branch to the appropriate copy of
         CURR_BE_FACTOR↑.  Split CURR_BE_FACTOR↑, if necessary, and make
         each father of CURR_BE_FACTOR↑ branch to the appropriate  copy
         of CURR_BE_FACTOR↑.  Then mark IMPLIER↑ so that
         CONSIDER_FOR_SPLIT can make IMPLIER↑ branch to the appropriate
         copy of X↑, where X↑ is a son of IMPLIER↑ and
         X ≠ CURR_BE_FACTOR.}

        if (CURR_BE_FACTOR = IMPLIER↑.TRUE_SON and MAIN_SUBDAG = TTRUE)
           or (CURR_BE_FACTOR = IMPLIER↑.FALSE_SON and
           MAIN_SUBDAG = FFALSE)

          → IMPLIER↑.MARKERS := [MAIN];
            CONSIDER_FOR_SPLIT(CURR_BE_FACTOR);
            IMPLIER↑.MARKERS := [SECOND]
```

```
        [] not ((CURR_BE_FACTOR = IMPLIER↑.TRUE_SOF and
           MAIN_SUBDAG = TTRUE) or (CURR_BE_FACTOR = IMPLIER↑.FALSE_SOF
           and MAIN_SUBDAG = FFALSE))

           → IMPLIER↑.MARKERS := [SECOND];
             CONSIDER_FOR_SPLIT(CURR_BE_FACTOR);
             IMPLIER↑.MARKERS := [MAIN]
        fi;

        CURR_BE_FACTOR := CHECK_FOR_MERGE[1];
        DELETE_MIN_FROM_HEAP

[] CURR_BE_FACTOR = CONSEQUENCE

   → skip
fi;

{BE-factor X↑, where X = CURR_BE_FACTOR or X is CHECK_FOR_MERGE,
 must be checked for merging paths.  All BE-factors Y↑ that are
 reachable from such X↑ and are marked REACHES must also be
 checked for merging paths.  If paths of different types do merge
 at any of these BE-factors, generate the appropriate copy(ies)
 and change the father BE-factors to branch to the proper copy.}

   do CURR_BE_FACTOR ≠ CONSEQUENCE

      → CONSIDER_FOR_SPLIT(CURR_BE_FACTOR);
        CURR_BE_FACTOR := CHECK_FOR_MERGE[1];
        DELETE_MIN_FROM_HEAP

   od;

IMPLIER↑.MARKERS := [ ];
```

```
procedure CONSIDER_FOR_SPLIT(CURR_BE_FACTOR: BE_FACTOR_PTR);

  {Add to CHECK_FOR_MERGE those sons of CURR_BE_FACTOR↑ that reach
   CONSEQUENCE↑.  If paths of different types merge at CURR_BE_FACTOR↑,
   generate the appropriate copy(ies) and change father BE-factors to
   branch to the proper copy.}

  var SECOND_COPY {Pointer to the copy of CURR_BE_FACTOR↑ that replaces
                   CURR_BE_FACTOR↑ on the paths marked with SECOND.},

      UNKNOWN_COPY {Pointer to the copy of CURR_BE_FACTOR↑ that replaces
                    CURR_BE_FACTOR↑ on the paths marked with UNKNOWN.}
                   : BE_FACTOR_PTR;

      CURR_FATHER {Pointer into the list of original fathers of
                   CURR_BE_FACTOR↑.  CURR_FATHER↑.VALUE points to the
                   father of CURR_BE_FACTOR↑ currently being added to the
                   fathers list of the appropriate copy of
                   CURR_BE_FACTOR↑.},

      FATHER {Pointer to the list of original fathers of CURR_BE_FACTOR↑
              that have yet to be added to the fathers list of the
              appropriate copy of CURR_BE_FACTOR↑.}
             : LINK;

  begin
    ADD_APPROPRIATE_SONS_TO_HEAP(CURR_BE_FACTOR);

    if card(CURR_BE_FACTOR↑.MARKERS) > 1

         {Make appropriate copies.}

           if UNKNOWN in CURR_BE_FACTOR↑.MARKERS
              → CREATE_COPY(CURR_BE_FACTOR, UNKNOWN);
                UNKNOWN_COPY↑.MARKERS := [UNKNOWN];
                UNKNOWN_COPY↑.FATHERS := nil
           [] not (UNKNOWN in CURR_BE_FACTOR↑.MARKERS)
                skip
           fi;

           if SECOND in CURR_BE_FACTOR↑.MARKERS and
              MAIN in CURR_BE_FACTOR↑.MARKERS
              → CREATE_COPY(CURR_BE_FACTOR, SECOND_COPY);
                SECOND_COPY↑.MARKERS := [SECOND];
                SECOND_COPY↑.FATHERS := nil;
                CURR_BE_FACTOR↑.MARKERS := [MAIN]
           [] not (MAIN in CURR_BE_FACTOR↑.MARKERS)
              → CURR_BE_FACTOR↑.MARKERS := [SECOND];
                SECOND_COPY := CURR_BE_FACTOR
           [] not (SECOND in CURR_BE_FACTOR↑.MARKERS)
                CURR_BE_FACTOR↑.MARKERS := [MAIN]
           fi;
```

```
{Change fathers of CURR_BE_FACTOR↑ to branch to copy with same
 marker.}
   FATHER := CURR_BE_FACTOR↑.FATHERS;
   CURR_BE_FACTOR↑.FATHERS := nil;

 do FATHER ≠ nil
      CURR_FATHER := FATHER;
      FATHER := FATHER↑.NEXT;

      {Add CURR_FATHER to the list of fathers for the appropriate
       copy.  Change the QUADS and DAG to reflect this addition.}

         if MAIN in CURR_FATHER↑.VALUE↑.MARKERS

            → {Put CURR_FATHER into list of fathers of
              CURR_BE_FACTOR↑.}
                CURR_FATHER↑.NEXT := CURR_BE_FACTOR↑.FATHERS;
                CURR_BE_FACTOR↑.FATHERS := CURR_FATHER

         [] SECOND in CURR_FATHER↑.VALUE↑.MARKERS

            → {Put CURR_FATHER into list of fathers of
              SECOND_COPY↑ and update branches in DAG and QUADS.}
                CURR_FATHER↑.NEXT := SECOND_COPY↑.FATHERS;
                SECOND_COPY↑.FATHERS := CURR_FATHER;
                CHANGE_BRANCH(CURR_FATHER↑.VALUE, CURR_BE_FACTOR,
                              SECOND_COPY)

         [] not (MAIN in CURR_FATHER↑.VALUE↑.MARKERS or
            SECOND in CURR_FATHER↑.VALUE↑.MARKERS)

            → {Put CURR_FATHER in list of fathers of UNKNOWN_COPY↑
              and update branches in DAG and QUADS.}
                CURR_FATHER↑.NEXT := UNKNOWN_COPY↑.FATHERS;
                UNKNOWN_COPY↑.FATHERS := CURR_FATHER;
                CHANGE_BRANCH(CURR_FATHER↑.VALUE, CURR_BE_FACTOR,
                              UNKNOWN_COPY)
         fi
   od

 [] card(CURR_BE_FACTOR↑.MARKERS) = 1
    → skip
 fi;

end;  {CONSIDER_FOR_SPLIT}
```

```
procedure ADD_APPROPRIATE_SONS_TO_HEAP(CURR_BE_FACTOR: BE_FACTOR_PTR);

  {Add to heap CHECK_FOR_MERGE those sons of CURR_BE_FACTOR↑ that reach
   CONSEQUENCE↑ and at which a value for CONSEQUENCE↑ is known.  Add
   the markers associated with CURR_BE_FACTOR↑ to the markers associated
   with these sons and remove REACHES.}

   begin

     if REACHES in CURR_BE_FACTOR↑.TRUE_SON↑.MARKERS

        → ADD_TO_HEAP(CURR_BE_FACTOR↑.TRUE_SON);
          CURR_BE_FACTOR↑.TRUE_SON↑.MARKERS :=
          (CURR_BE_FACTOR↑.TRUE_SON↑.MARKERS + CURR_BE_FACTOR↑.MARKERS)
                                                          - [REACHES]

     [] not (REACHES in CURR_BE_FACTOR↑.TRUE_SON↑.MARKERS)

        → skip
     fi;

     if REACHES in CURR_BE_FACTOR↑.FALSE_SON↑.MARKERS

        → ADD_TO_HEAP(CURR_BE_FACTOR↑.FALSE_SON);
          CURR_BE_FACTOR↑.FALSE_SON↑.MARKERS :=
          (CURR_BE_FACTOR↑.FALSE_SON↑.MARKERS + CURR_BE_FACTOR↑.MARKERS)
                                                            - [REACHES]

     [] not (REACHES in CURR_BE_FACTOR↑.FALSE_SON↑.MARKERS)

        → skip
     fi

   end;  {ADD_APPROPRIATE_SONS_TO_HEAP}
```

```
procedure ADD_TO_HEAP(NEW_ENTRY: BE_FACTOR_PTR);

  {Add BE-factor NEW_ENTRY to heap CHECK_FOR_MERGE and reestablish
   the heap property.}

   var TEMP {Variable used to temporarily hold an entry in heap
            CHECK_FOR_MERGE while a swapping of entries is being
            performed.}
           : BE_FACTOR_PTR;

       INDEX {Index of NEW_ENTRY within heap CHECK_FOR_MERGE.},

       FATHER_INDEX {Index of father of NEW_ENTRY within heap
                    CHECK_FOR_MERGE.}
                   : INTEGER;

   begin
     {Add NEW_ENTRY to heap CHECK_FOR_MERGE.}

        NUM_IN_HEAP := NUM_IN_HEAP + 1;
        CHECK_FOR_MERGE[NUM_IN_HEAP] := NEW_ENTRY;

     {Reestablish heap property.}

        INDEX := NUM_IN_HEAP;
        FATHER_INDEX := NUM_IN_HEAP div 2;

        do FATHER_INDEX ≠ 0 cand
           CHECK_FOR_MERGE[FATHER_INDEX]↑.NODE_NUM >
           CHECK_FOR_MERGE[INDEX]↑.NODE_NUM

           → {Swap CHECK_FOR_MERGE[FATHER_INDEX] with
              CHECK_FOR_MERGE[INDEX].}

                TEMP := CHECK_FOR_MERGE[FATHER_INDEX];
                CHECK_FOR_MERGE[FATHER_INDEX] := CHECK_FOR_MERGE[INDEX];
                CHECK_FOR_MERGE[INDEX] := TEMP;
                INDEX := FATHER_INDEX;
                FATHER_INDEX := INDEX div 2
        od

   end; {ADD_TO_HEAP}
```

```
procedure DELETE_MIN_FROM_HEAP;

  {Delete CHECK_FOR_MERGE[1] from heap and reestablish the heap
   property.}

   var TEMP {Variable used to temporarily hold an entry in heap
             CHECK_FOR_MERGE while entries are being swapped.}
            : BE_FACTOR_PTR;

       INDEX {Index of the heap entry currently being checked if it
              upholds the heap property.},

       LEFT_SON {Index of the left son of heap entry currently being
                 checked for heap property.},

       RIGHT_SON {Index of the right son of heap entry currently being
                  checked for heap property.}
                 : INTEGER;

   begin

     {Delete CHECK_FOR_MERGE[1] from heap.}
       CHECK_FOR_MERGE[1] := CHECK_FOR_MERGE[NUM_IN_HEAP];
       NUM_IN_HEAP := NUM_IN_HEAP - 1;

     {Reestablish heap property.}
       INDEX := 1;
       LEFT_SON := 2;
       RIGHT_SON := 3;

     {CHECK_FOR_MERGE[I] ≤ min(CHECK_FOR_MERGE[2*I], CHECK_FOR_MERGE[2*I+1])
      for all I such that 1 ≤ I ≤ NUM_IN_HEAP and I ≠ INDEX.}

       do RIGHT_SON ≤ NUM_IN_HEAP cand
          CHECK_FOR_MERGE[RIGHT_SON]↑.NODE_NUM ≤
          CHECK_FOR_MERGE[LEFT_SON]↑.NODE_NUM cand
          CHECK_FOR_MERGE[RIGHT_SON]↑.NODE_NUM ≤
          CHECK_FOR_MERGE[INDEX]↑.NODE_NUM

          → {Swap CHECK_FOR_MERGE[RIGHT_SON] with CHECK_FOR_MERGE[INDEX].}

               TEMP := CHECK_FOR_MERGE[RIGHT_SON];
               CHECK_FOR_MERGE[RIGHT_SON] := CHECK_FOR_MERGE[INDEX];
               CHECK_FOR_MERGE[INDEX] := TEMP;
               INDEX := RIGHT_SON;
               LEFT_SON := 2 * INDEX;
               RIGHT_SON := LEFT_SON + 1
```

```
    [] (LEFT_SON = NUM_IN_HEAP or RIGHT_SON ≤ NUM_IN_HEAP) cand
         CHECK_FOR_MERGE[LEFT_SON]↑.NODE_NUM ≤
         CHECK_FOR_MERGE[RIGHT_SON]↑.NODE_NUM cand
         CHECK_FOR_MERGE[LEFT_SON]↑.NODE_NUM ≤
         CHECK_FOR_MERGE[INDEX]↑.NODE_NUM

         → {Swap CHECK_FOR_MERGE[LEFT_SON] with
            CHECK_FOR_MERGE[INDEX].}

              TEMP := CHECK_FOR_MERGE[LEFT_SON];
              CHECK_FOR_MERGE[LEFT_SON] := CHECK_FOR_MERGE[INDEX];
              CHECK_FOR_MERGE[INDEX] := TEMP;
              INDEX := LEFT_SON;
              LEFT_SON := 2 * INDEX;
              RIGHT_SON := LEFT_SON + 1
    od
  end {DELETE_MIN_FROM_HEAP}
  end; {SPLIT_PATHS}


procedure CREATE_COPY(ORIG: BE_FACTOR_PTR; var COPY: BE_FACTOR_PTR);

  {Create a copy of BE-factor ORIG↑ and the quadruples representing it.
   Return a pointer to this new BE-factor in COPY.}

  var ORIG_BTF_QUAD {Pointer to the BTF-quad in the sequence of quads
                     representing ORIG↑.},

      ORIG_QUAD {Pointer to the quad within the sequence of quads
                 representing ORIG↑ that is currently being copied
                 or considered for copying.},

      COPY_QUAD {Pointer to the copy of ORIG_QUAD↑ currently being built.},

      PRED_QUAD {Either (1) COPY_QUAD itself or (2) a pointer to the
                 quad created just prior to COPY_QUAD.}
                : QUAD_PTR;

  begin

    {Obtain space for the BE-factor COPY↑ and initialize the fields
     whose values are the same as for ORIG↑.}
      NEW(COPY, BE_FACTOR);
      COPY↑.NODE_NUM := ORIG↑.NODE_NUM;
      COPY↑.TRUE_SON := ORIG↑.TRUE_SON;
      COPY↑.FALSE_SON := ORIG↑.FALSE_SON;
      COPY↑.TIPE := BE_FACTOR

    {Put COPY in the fathers list of each son of ORIG↑ that is a
     BE-factor.}
      CREATE_ENTRY_IN_FATHERS(ORIG↑.TRUE_SON, COPY);
      CREATE_ENTRY_IN_FATHERS(ORIG↑.FALSE_SON, COPY);
```

```
{Create a copy of each of the quads representing BE-factor ORIG↑
 and set COPY↑.FIRST_QUAD and COPY↑.BTF_QUAD to point to the
 first and last or these copies, respectively.}

   ORIG_BTF_QUAD := ORIG↑.BTF_QUAD;
   ORIG_QUAD := ORIG↑.FIRST_QUAD;

  {Create a copy of the first quad in the sequence of quads
   representing ORIG↑ and set COPY↑.FIRST_QUAD to point at this
   quad.  If this quad is also the BTF-quad, then set COPY↑.BTF_QUAD
   to point to it.}

     if ORIG_QUAD ≠ ORIG_BTF_QUAD
        → NEW(COPY_QUAD, NON_BTF);
          COPY↑.FIRST_QUAD := COPY_QUAD;
          COPY_NON_BTF(COPY_QUAD, ORIG_QUAD);
          PRED_QUAD := COPY_QUAD;
          ORIG_QUAD := ORIG_QUAD↑.NEXT

     [] ORIG_QUAD = ORIG_BTF_QUAD
        → NEW(COPY_QUAD, BTF);
          COPY↑.FIRST_QUAD := COPY_QUAD;
          COPY↑.BTF_QUAD := COPY_QUAD;
          COPY_BTF(COPY_QUAD, ORIG_QUAD)
     fi;

  {Create a copy of each quad X↑ in the sequence of quads representing
   ORIG↑, where X ≠ ORIG↑.FIRST_QUAD and X ≠ ORIG_BTF_QUAD.}

     do ORIG_QUAD ≠ ORIG_BTF_QUAD
        → NEW(COPY_QUAD, NON_BTF);
          PRED_QUAD↑.NEXT := COPY_QUAD;
          COPY_NON_BTF(COPY_QUAD, ORIG_QUAD);
          PRED_QUAD := COPY_QUAD;
          ORIG_QUAD := ORIG_QUAD↑.NEXT
     od;

  {If ORIG_FIRST_QUAD ≠ ORIG_BTF_QUAD, create a copy of ORIG_BTF_QUAD↑
   and set COPY↑.BTF_QUAD to point to this copy.}

     if ORIG↑.FIRST_QUAD ≠ ORIG_BTF_QUAD
        → NEW(COPY_QUAD, BTF);
          COPY↑.BTF_QUAD := COPY_QUAD;
          PRED_QUAD↑.NEXT := COPY_QUAD;
          COPY_BTF(COPY_QUAD, ORIG_QUAD)

     [] ORIG↑.FIRST_QUAD = ORIG_BTF_QUAD
        → skip
     fi;

end; {CREATE_COPY}
```

```
procedure CREATE_ENTRY_IN_FATHERS(SON: GCS_PART_PTR;
                                  FATHER_VALUE: BE_FACTOR_PTR);

  {If SON↑ is a BE-factor, make FATHER_VALUE↑ a father of SON↑.}

  var FATHER {Pointer to a new entry in a fathers list.}
           : LINK;

  begin

    if SON↑.TIPE = BE_FACTOR

       → {Create a new entry in the fathers list of SON↑ that indicates
          FATHER_VALUE↑ is a father.}

            NEW(FATHER);
            FATHER↑.VALUE := FATHER_VALUE;
            FATHER↑.NEXT := SON↑.FATHERS;
            SON↑.FATHERS := FATHER

    [] SON↑.TIPE ≠ BE_FACTOR

       → skip
    fi

  end;  {CREATE_ENTRY_IN_FATHERS}




procedure ADD_TO_FATHERS(SON: BE_FACTOR_PTR; FATHER: LINK);

  {If SON↑ is a BE-factor, make FATHER↑ and element of the fathers list
   of SON↑.}

  begin

    if SON↑.TIPE =BE_FACTOR
       → FATHER↑.NEXT := SON↑.FATHERS;
         SON↑.FATHERS := FATHER

    [] SON↑.TIPE ≠ BE_FACTOR
       → skip
    fi

  end;  {ADD_TO_FATHERS}
```

```
procedure COPY_BTF(COPY_QUAD, ORIG_QUAD: QUAD_PTR);

  {Copy the data from BTF-quad ORIG_QUAD↑ into COPY_QUAD↑.}

  begin

    COPY_QUAD↑.OPERATOR := ORIG_QUAD↑.OPERATOR;
    COPY_QUAD↑.OPERAND1 := ORIG_QUAD↑.OPERAND1;
    COPY_QUAD↑.BT := ORIG_QUAD↑.BT;
    COPY_QUAD↑.BF := ORIG_QUAD↑.BF

  end;  {COPY_BTF}



procedure COPY_NON_BTF(COPY_QUAD, ORIG_QUAD: QUAD_PTR);

  {Copy the data from non-BTF-quad ORIG_QUAD↑ into COPY_QUAD↑.}

  begin

    COPY_QUAD↑.OPERATOR := ORIG_QUAD↑.OPERATOR;
    COPY_QUAD↑.OPERAND1 := ORIG_QUAD↑.OPERAND1;
    COPY_QUAD↑.OPERAND2 := ORIG_QUAD↑.OPERAND2;
    COPY_QUAD↑.RESULT := ORIG_QUAD↑.RESULT

  end;  {COPY_NON_BTF}



procedure CHANGE_BRANCH(CURR_FATHER, OLD_SON, NEW_SON: GCS_PART_PTR);

  {For each son S↑ of CURR_FATHER↑, if S = OLD_SON then set S = NEW_SON
   and change the quads associated with CURR_FATHER↑ to branch to
   NEW_SON↑.FIRST_QUAD instead of to OLD_SON↑.FIRST_QUAD.}

  begin

    do CURR_FATHER↑.TRUE_SON = OLD_SON
       → CURR_FATHER↑.TRUE_SON := NEW_SON;
         CURR_FATHER↑.BTF_QUAD↑.BT := NEW_SON↑.FIRST_QUAD

    [] CURR_FATHER↑.FALSE_SON = OLD_SON
       → CURR_FATHER↑.FALSE_SON := NEW_SON;
         CURR_FATHER↑.BTF_QUAD↑.BF := NEW_SON↑.FIRST_QUAD
    od

  end;  {CHANGE_BRANCH}
```

```
procedure SKIP_OVER(CONSEQUENCE, IMPLIER: BE_FACTOR_PTR;
                    VALUE_IMPLIED, MAIN_SUBDAG: TRUTH_VALUE;
                    SECOND_SUBDAG: TRUTH_VAL_OR_NEITHER);

 {Make each father of CONSEQUENCE↑ that is marked MAIN the father of the
  VALUE_IMPLIED son of CONSEQUENCE↑.  Make each father of CONSEQUENCE↑
  that is marked SECOND the father of the not VALUE_IMPLIED son of
  CONSEQUENCE↑.  If CONSEQUENCE↑ is a son of IMPLIER↑, make IMPLIER↑ the
  father of CONSEQUENCE↑ and/or the father of the true and/or false son(s
  of CONSEQUENCE↑, depending upon the value of MAIN_SUBDAG, SECOND_SUBDAG
  VALUE_IMPLIED, IMPLIER↑.TRUE_SON and IMPLIER↑.FALSE_SON.}

  var CURR_FATHER {Pointer into the list of original fathers of
                   CONSEQUENCE↑.  CURR_FATHER↑.VALUE points to the fathe
                   of CONSEQUENCE↑ currently being considered for skippi
                   over CONSEQUENCE↑.},

      FATHER {Pointer to the list of original fathers of CONSEQUENCE↑ th
              have yet to be considered for skipping over CONSEQUENCE↑.}
             : LINK;

      SON {If CURR_FATHER↑.VALUE↑ is marked MAIN or SECOND, that son of
           CONSEQUENCE↑ to which this father skips.}
          : GCS_PART_PTR;

  begin

    FATHER := CONSEQUENCE↑.FATHERS;
    CONSEQUENCE↑.FATHERS  := nil;

    do FATHER ≠ nil

      → CURR_FATHER := FATHER;
        FATHER := FATHER↑.NEXT;

       {Make CONSEQUENCE↑, CONSEQUENCE↑.TRUE_SON↑ or
        CONSEQUENCE↑.FALSE_SON↑ the son of CURR_FATHER↑.VALUE depending
        upon VALUE_IMPLIED and whether CURR_FATHER↑.VALUE↑ is marked
        MAIN, SECOND or UNKNOWN.  If this GCS-part is a BE-factor, add
        CURR_FATHER to its fathers list.}

          if (MAIN in CURR_FATHER↑.VALUE↑.MARKERS and
              VALUE_IMPLIED = TTRUE) or (SECOND in
              CURR_FATHER↑.VALUE↑.MARKERS and VALUE_IMPLIED = FFALSE)

            → SON := CONSEQUENCE↑.TRUE_SON;
              ADD_TO_FATHERS(SON, CURR_FATHER);
              CHANGE_BRANCH(CURR_FATHER↑.VALUE, CONSEQUENCE, SON)
```

```
        □ (MAIN in CURR_FATHER↑.VALUE↑.MARKERS and
           VALUE_IMPLIED = FFALSE) or
          (SECOND in CURR_FATHER↑.VALUE↑.MARKERS and
           VALUE_IMPLIED = TTRUE)

           → SON := CONSEQUENCE↑.FALSE_SON;
             ADD_TO_FATHERS(SON, CURR_FATHER);
             CHANGE_BRANCH(CURR_FATHER↑.VALUE, CONSEQUENCE, SON)

        □ UNKNOWN in CURR_FATHER↑.VALUE↑.MARKERS or
          (card(CURR_FATHER↑.VALUE↑.MARKERS) = 0 and
           CURR_FATHER↑.VALUE ≠ IMPLIER)

           → CURR_FATHER↑.NEXT := CONSEQUENCE↑.FATHERS;
             CONSEQUENCE↑.FATHERS := CURR_FATHER

        □ CURR_FATHER↑.VALUE = IMPLIER

           → {Remove IMPLIER↑ from fathers list of CONSEQUENCE↑.}

                skip
        fi
   od;

{If CONSEQUENCE↑ is a son of IMPLIER↑, make IMPLIER↑ the father of
CONSEQUENCE↑, CONSEQUENCE↑.TRUE_SON↑ and/or CONSEQUENCE↑.FALSE_SON↑,
depending upon the value of MAIN_SUBDAG, SECOND_SUBDAG, VALUE_IMPLIED,
IMPLIER↑.TRUE_SON and IMPLIER↑.FALSE_SON.}

   FIX_IMPLIER_SONS(IMPLIER, CONSEQUENCE, VALUE_IMPLIED, MAIN_SUBDAG,
                    SECOND_SUBDAG)

end; {SKIP_OVER}
```

```
procedure FIX_IMPLIER_SONS(IMPLIER, CONSEQUENCE: BE_FACTOR_PTR;
                           VALUE_IMPLIED, MAIN_SUBDAG: TRUTH_VALUE;
                           SECOND_SUBDAG: TRUTH_VAL_OR_NEITHER);

 {If CONSEQUENCE↑ is the MAIN_SUBDAG son of IMPLIER↑, make the
  VALUE_IMPLIED son of CONSEQUENCE↑ the MAIN_SUBDAG son of IMPLIER↑.  If
  CONSEQUENCE↑ is the SECOND_SUBDAG son of IMPLIER↑, make the not
  VALUE_IMPLIED son of CONSEQUENCE↑ the SECOND_SUBDAG son of IMPLIER↑.  If
  CONSEQUENCE↑ is a son of IMPLIER↑ but not the MAIN_SUBDAG son and
  SECOND_SUBDAG is NEITHER, then put IMPLIER↑ back on the fathers list
  of CONSEQUENCE↑.}

  begin
    if IMPLIER↑.TRUE_SON = CONSEQUENCE and
      ((MAIN_SUBDAG = TTRUE and VALUE_IMPLIED = TTRUE) or
       (MAIN_SUBDAG = FFALSE and SECOND_SUBDAG = TTRUE and
        VALUE_IMPLIED = FFALSE))
       → IMPLIER↑.TRUE_SON := CONSEQUENCE↑.TRUE_SON;
         IMPLIER↑.BTF_QUAD↑.BT := CONSEQUENCE↑.TRUE_SON↑.FIRST_QUAD;
         CREATE_ENTRY_IN_FATHERS(CONSEQUENCE↑.TRUE_SON, IMPLIER)

    [] IMPLIER↑.TRUE_SON = CONSEQUENCE and
      ((MAIN_SUBDAG = TTRUE and VALUE_IMPLIED = FFALSE) or
       (MAIN_SUBDAG = FFALSE and SECOND_SUBDAG = TTRUE and
        VALUE_IMPLIED = TTRUE))
       → IMPLIER↑.TRUE_SON := CONSEQUENCE↑.FALSE_SON;
         IMPLIER↑.BTF_QUAD↑.BT := CONSEQUENCE↑.FALSE_SON↑.FIRST_QUAD;
         CREATE_ENTRY_IN_FATHERS(CONSEQUENCE↑.FALSE_SON, IMPLIER)

    [] IMPLIER↑.TRUE_SON = CONSEQUENCE and
      (MAIN_SUBDAG = FFALSE and SECOND_SUBDAG = NEITHER)
       → CREATE_ENTRY_IN_FATHERS(CONSEQUENCE↑.TRUE_SON, IMPLIER)

    [] IMPLIER↑.TRUE_SON ≠ CONSEQUENCE
       →  skip
    fi;

    if IMPLIER↑.FALSE_SON = CONSEQUENCE and
      ((MAIN_SUBDAG = FFALSE and VALUE_IMPLIED = TTRUE) or
       (MAIN_SUBDAG = TTRUE and SECOND_SUBDAG = FFALSE and
        VALUE_IMPLIED = FFALSE))
       → IMPLIER↑.FALSE_SON := CONSEQUENCE↑.TRUE_SON;
         IMPLIER↑.BTF_QUAD↑.BF := CONSEQUENCE↑.TRUE_SON↑.FIRST_QUAD;
         CREATE_ENTRY_IN_FATHERS(CONSEQUENCE↑.TRUE_SON, IMPLIER)

    [] IMPLIER↑.FALSE_SON = CONSEQUENCE and
      ((MAIN_SUBDAG = FFALSE and VALUE_IMPLIED = FFALSE) or
       (MAIN_SUBDAG = TTRUE and SECOND_SUBDAG = FFALSE and
        VALUE_IMPLIED = TTRUE))
       → IMPLIER↑.FALSE_SON := CONSEQUENCE↑.FALSE_SON;
         IMPLIER↑.BTF_QUAD↑.BF := CONSEQUENCE↑.FALSE_SON↑.FIRST_QUAD;
         CREATE_ENTRY_IN_FATHERS(CONSEQUENCE↑.FALSE_SON, IMPLIER)
```

```
    [] IMPLIER↑.FALSE_SON = CONSEQUENCE and
       (MAIN_SUBDAG = TTRUE and SECOND_SUBDAG = NEITHER)
        → CREATE_ENTRY_IN_FATHERS(CONSEQUENCE↑.FALSE_SON, IMPLIER)

    [] IMPLIER↑.FALSE_SON ≠ CONSEQUENCE
       →  skip
    fi
  end;  {FIX_IMPLIER_SONS}


procedure REMOVE_MARKS(IMPLIER: BE_FACTOR_PTR);

  {Set X↑.MARKERS to [ ] for all BE-factors X↑ reachable from IMPLIER↑
   that have been marked.}

  var UNMARK_LIST {Pointer to first element of a linked list of pointers
                   to BE-factors that must be checked for unmarking.},
      NEW_ENTRY {Pointer to space obtained for a new entry into linked
                 list UNMARK_LIST↑.}
                 : LINK;

      CURR_BE_FACTOR {Pointer to the BE-factor currently being considered
                      for unmarking and as a link to other BE-factors
                      that must be considered for unmarking.}
                     : BE_FACTOR_PTR;

  begin
    UNMARK_LIST := nil;
    CURR_BE_FACTOR := IMPLIER;
    {All X↑ that are marked BE-factors, where X = CURR_BE_FACTOR or X is
     in UNMARK_LIST↑, and all marked BE-factors reachable from such X↑
     must be unmarked.}

      do card(CURR_BE_FACTOR↑.MARKERS) ≥ 1

         → {Unmark CURR_BE_FACTOR↑ and indicate its sons must be
            considered for unmarking.}
              CURR_BE_FACTOR↑.MARKERS := [ ];
              NEW(NEW_ENTRY);
              NEW_ENTRY↑.NEXT := UNMARK_LIST;
              NEW_ENTRY↑.VALUE := CURR_BE_FACTOR↑.TRUE_SON;
              UNMARK_LIST := NEW_ENTRY;
              CURR_BE_FACTOR := CURR_BE_FACTOR↑.FALSE_SON

      [] card(CURR_BE_FACTOR↑.MARKERS) = 0 and UNMARK_LIST ≠ nil

         → {Access next BE-factor to be checked for unmarking.}
              CURR_BE_FACTOR := UNMARK_LIST↑.VALUE;
              UNMARK_LIST := UNMARK_LIST↑.NEXT
      od
  end  {REMOVE_MARKS}
  end; {REMOVE_REDUNDANCY}
```

Chapter 9

Some Examples

A few examples are now given to illustrate the usefulness of applying the algorithm just described. In each example, the original DAG representation of the GCS (see Section 5.1) and the optimized DAG are given. Most paths through the optimized DAO are considerably shorter than the corresponding paths through the original DAG. Thus, needless evaluation of certain boolean expressions is avoided in many executions of the GCS and hence considerable time is often saved in executing the entire guarded command construct.

1. Find the maximum m of two values x and y.

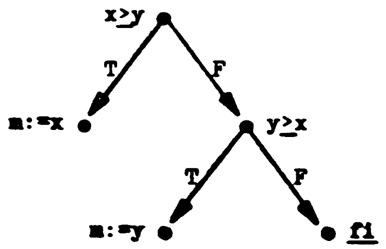
Program segment

<u>if</u> $x \geq y \rightarrow m := x$

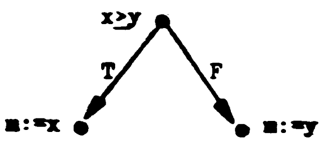
$\square$ $y \geq x \rightarrow m := y$

<u>fi</u>

Original DAG



Optimized DAG



2. Search a 2-dimensional array b for a value x.

Program segment

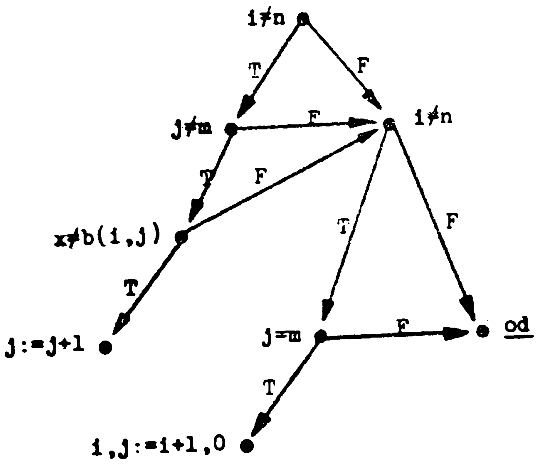
i, j := 0, 0;

<u>do</u> i ≠ n <u>cand</u> j ≠ m <u>cand</u> x ≠ b(i,j) → j := j + 1

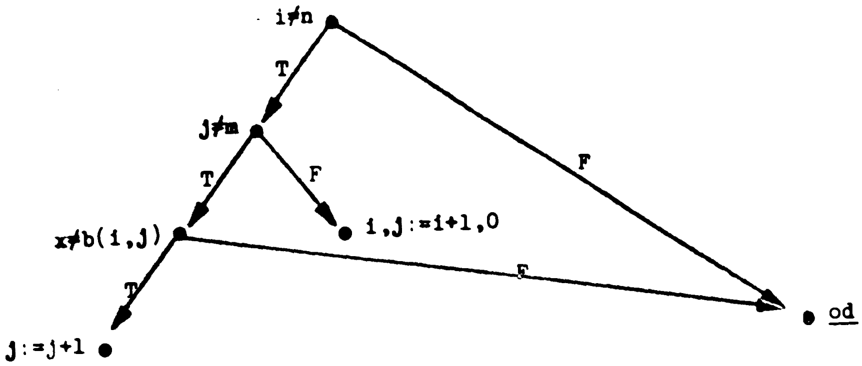
□ i ≠ n <u>cand</u> j = m → i, j := i + 1, 0

<u>od</u>

Original DAG



Optimized DAG



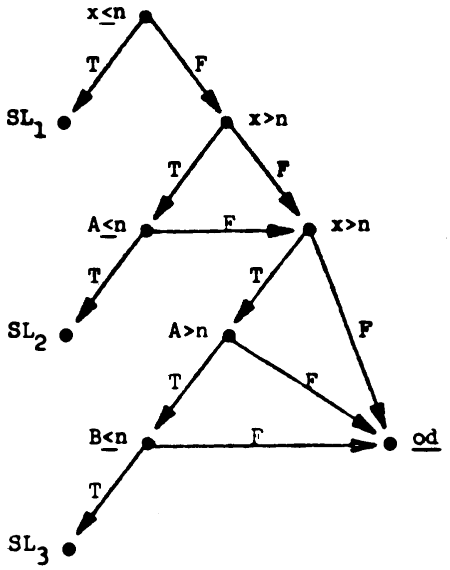
3. **The following is a linear sieve algorithm for finding prime numbers [5].**

```
{n ≥ 4}
p, q, k, x, S := 2, 2, 1, 4, (2, ..., n};
do  x ≤ n
      → remove(S,x);  k, x := k + 1, p * x
[]  x > n and p * next(S,q) ≤ n
      → q := next(S,q);  k, x := 1, p * q
[]  x > n and p * next(S,q) > n and next(S,p)^2 ≤ n
      → p := next(S,p);  q, k, x := p, 1, p * p
od
{S = {y | 2 ≤ y ≤ n and y is prime})
```

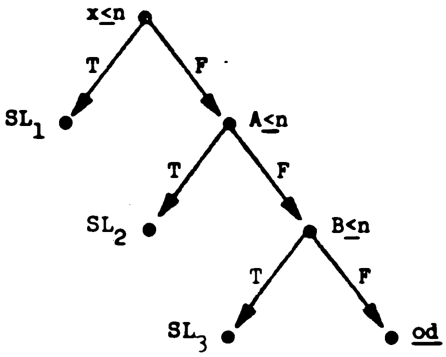
This iterative statement is abbreviated as:

```
do  x ≤ n                            → SL1
[]  x > n and A ≤ n                  → SL2
[]  x > n and A > n and B ≤ n        → SL3
od
```

Original DAG



Optimized DAG



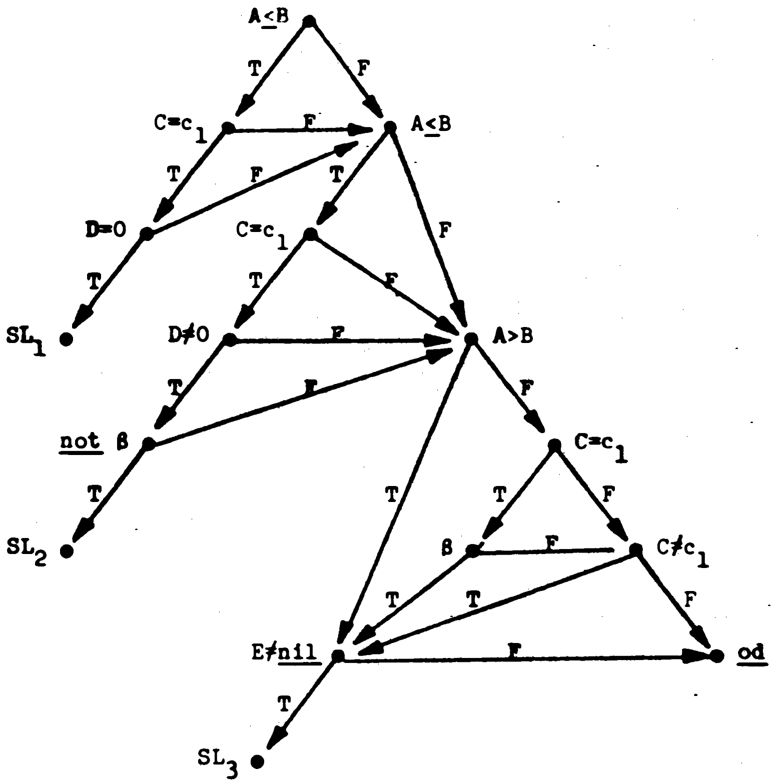
4. The following iterative statement is taken from procedure MARK. (See Chapter 8.)

```
do  CURR_GCS_PART↑.NODE_NUM ≤ CONSEQUENCE↑.NODE_NUM and
    CURR_GCS_PART↑.TIPE = BE_FACTOR cand
    card (CURR_GCS_PART↑.MARKERS) = 0
    → SL1
□   CURR_GCS_PART↑.NODE_NUM ≤ CONSEQUENCE↑.NODE_NUM and
    CURR_GCS_PART↑.TIPE = BE_FACTOR cand
    card (CURR_GCS_PART↑.MARKERS) ≠ 0 cand
    not (MARKER in CURR_GCS_PART↑.MARKERS)
    → SL2
□   (CURR_GCS_PART↑.NODE_NUM > CONSEQUENCE↑.NODE_NUM or
    (CURR_GCS_PART↑.TIPE = BE_FACTOR cand MARKER in
     CURR_GCS_PART↑.MARKERS)
     or CURR_GCS_PART↑.TIPE ≠ BE_FACTOR) and
     CONSIDER_FOR_MARK ≠ nil
    → SL3
od
```

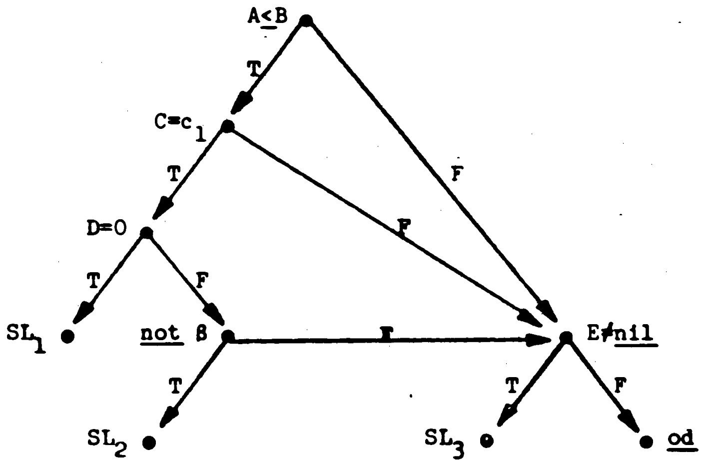
This iterative statement is abbreviated as:

```
do  A ≤ B and C = c1 cand D = 0                          → SL1
□   A ≤ B and C = c1 cand D ≠ 0 cand not β               → SL2
□   (A > B or (C = c1 cand β) or C ≠ c1) and E ≠ nil     → SL3
od
```

Original DAG



Optimized DAG



5. **The following alternative statement is taken from procedure SKIP_OVER. (See Chapter 8.)**

```
if  (MAIN in CURR_FATHER↑.VALUE↑.MARKERS and
     VALUE_IMPLIED = TTRUE)
    or (SECOND in CURR_FATHER↑.VALUE↑.MARKERS and
        VALUE_IMPLIED = FFALSE)
    → SL1

[]  (MAIN in CURR_FATHER↑.VALUE↑.MARKERS and
     VALUE_IMPLIED = FFALSE)
    or (SECOND in CURR_FATHER↑.VALUE↑.MARKERS and
        VALUE_IMPLIED = TTRUE)
    → SL2

[]  UNKNOWN in CURR_FATHER↑.VALUE↑.MARKERS
    → SL3

fi
```

This is abbreviated as:

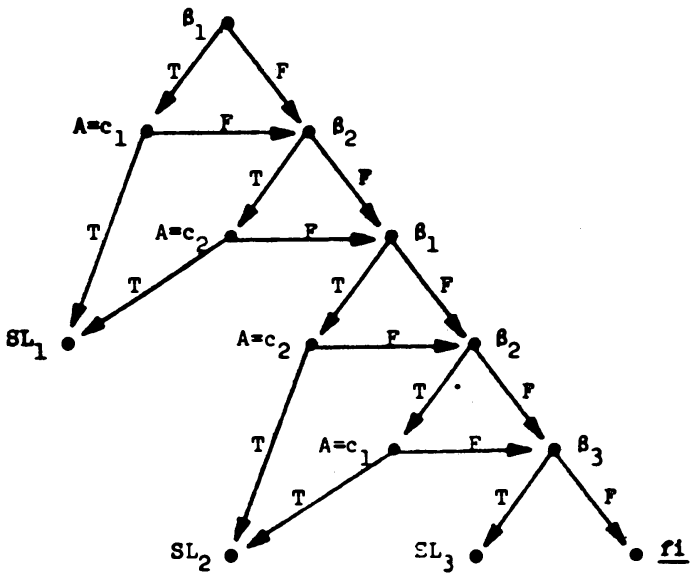
$\underline{\text{if}}$ $(\beta_1$ $\underline{\text{and}}$ $A = c_1)$ $\underline{\text{or}}$ $(\beta_2$ $\underline{\text{and}}$ $A = c_2)$ $\rightarrow SL_1$

$\square$ $(\beta_1$ $\underline{\text{and}}$ $A = c_2)$ $\underline{\text{or}}$ $(\beta_2$ $\underline{\text{and}}$ $A = c_1)$ $\rightarrow SL_2$

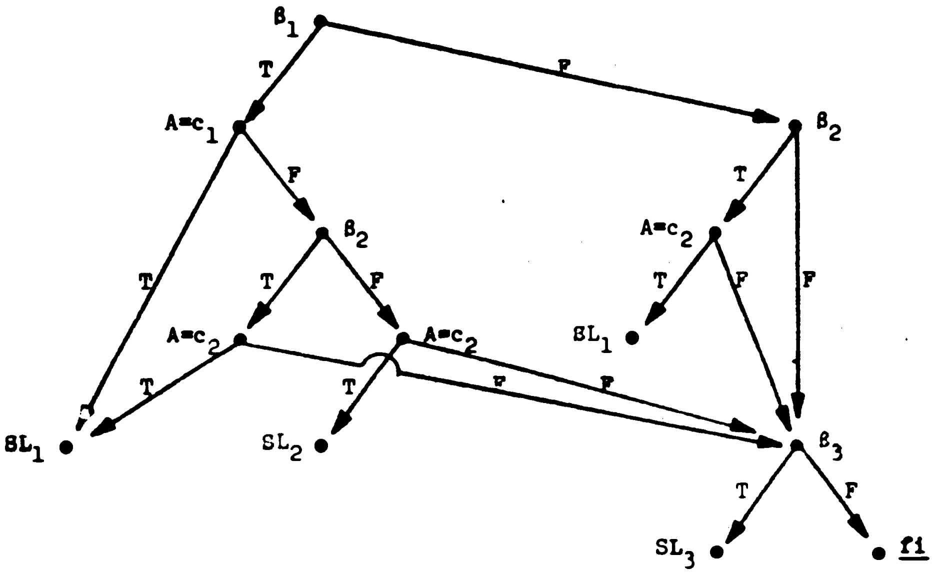
$\square$ β_3 $\rightarrow SL_3$

$\underline{\text{fi}}$

Original DAG



Optimized DAG



6. The following iterative statement is taken form procedure DELETE_MIN_FROM_HEAP. (See Chapter 8.) It reestablishes the heap property.

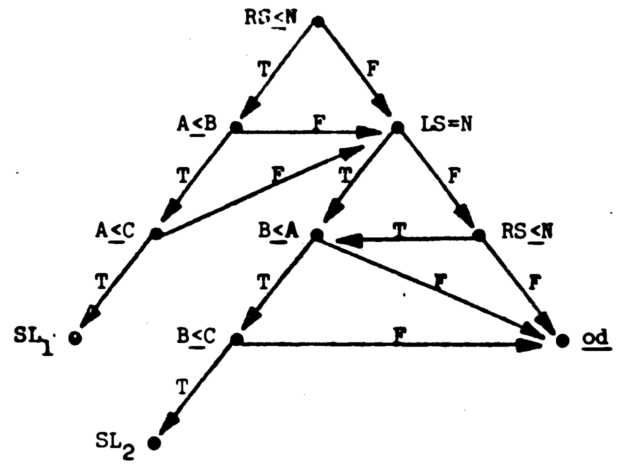
```
do  RIGHT_SON ≤ NUM_IN_HEAP cand
    CHECK_FOR_MERGE[RIGHT_SON]↑.NODE_NUM ≤
    CHECK_FOR_MERGE[LEFT_SON]↑.NODE_NUM
    cand CHECK_FOR_MERGE[RIGHT_SON]↑.NODE_NUM ≤
    CHECK_FOR_MERGE[INDEX]↑.NODE_NUM
    → SL1

[]  (LEFT_SON = NUM_IN_HEAP or RIGHT_SON ≤ NUM_IN_HEAP) cand
    CHECK_FOR_MERGE[LEFT_SON]↑.NODE_NUM ≤
    CHECK_FOR_MERGE[RIGHT_SON]↑.NODE_NUM
    cand CHECK_FOR_MERGE[LEFT_SON]↑.NODE_NUM ≤
    CHECK_FOR_MERGE[INDEX]↑.NODE_NUM
    → SL2
od
```

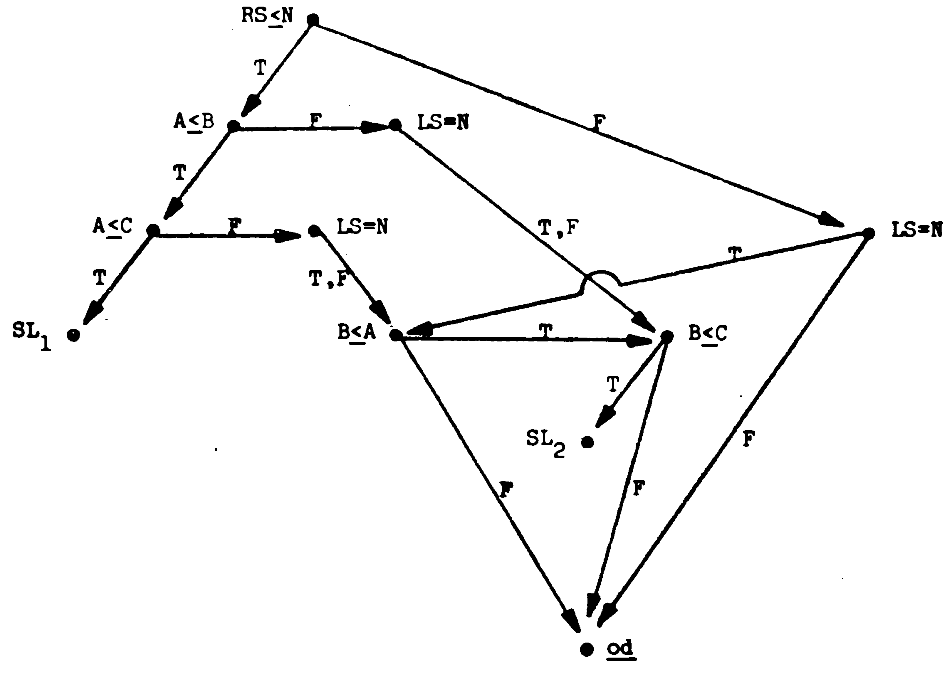
This is abbreviated as:

```
do  RS ≤ N cand A ≤ B cand A ≤ C                 → SL1
[]  (LS = N or RS ≤ N) cand B ≤ A cand B ≤ C   → SL2
od
```

Original DAG



Optimized DAG



Note that two occurrences of LS = N branch to one BE-factor - B $\leq$ A in one case, B $\leq$ C in another - regardless of its truth value. Hence these two occurrences may be removed to further optimize the DAG.

Chapter 10

Conclusion

Two types of optimizations that improve the execution time of a straightforward implementation of the guarded command constructs have been described. One is to order the guarded commands using some criterion that minimizes the execution time of the guarded command set. A criterion that may be used is the amount of time required to evaluate each guard. The other type, and by far the most useful, is the elimination of redundant boolean expressions from the guards. An algorithm for recognizing and eliminating several redundant boolean expressions has been presented. This algorithm makes it possible to implement the guarded command alternative construct with efficiency comparable to the IF-THEN-ELSE construct.

Bibliography

1. Aho, A. V., Ullman, J. D., Principles of Compiler Design, Addison-Wesley, Reading, Mass, 1977.
2. Dijkstra, E. W., A Discipline of Programming, Prentice Hall, Englewood Cliffs, N.J., 1976.
3. Gries, D., Compiler Construction for Digital Computers, John Wiley & Sons, Inc., New York, N.Y., 1971.
4. Gries, D., "A Note on Iteration," TR-77-323, Department of Computer Science, Cornell University, 1977.
5. Gries, D., Misra, J., "A Linear Sieve Algorithm for Finding Prime Numbers," TR-77-313, Department of Computer Science, Cornell University, 1977.
6. Wulf, W., Johnsson, R. K., Weinstock, C. B., Hobbs, S. O., Geschke, C. M., The Design of an Optimizing Compiler, American Elsevier, New York, N.Y., 1975.