AN ALGORITHM FOR DETECTING UNEXECUTABLE PATHS THROUGH PROGRAM FLOW GRAPHS

by

Lee A. Bollacker
Department of Computer Science
University of Colorado at Boulder
Boulder, Colorado 80309

#CU-CS-112-78 January, 1978

Introduction

Data flow analysis techniques such as those employed by DAVE [1] can be very useful in discovering certain anomalies in data flow through a program. When such an anomaly is discovered by DAVE, a message is issued along with a description of a path containing the anomaly.

One unfortunate aspect of these techniques, however, is that there is no guarantee that the path on which an anomaly occurs is executable. This phenomenon effectively reduces the credibility of a system such as DAVE in the eyes of the user, because each path output by DAVE must be inspected by hand in order to verify its executability. In addition, messages for anomalies which occur only on unexecutable paths represent a waste of the user's (and of DAVE's) time.

In order to reduce the work required for DAVE and the user, and to increase the quality of information provided by DAVE, it would be useful, then, to eliminate all unexecutable paths from consideration in the data flow analyses. For the general case, the problem of determining the executability of a given path has no solution, being equivalent to the Halting Problem [2]. It is possible, however, to detect certain classes of paths which are unexecutable because of the branching conditions which must be satisfied in order to traverse them.

The paths with which we will concern ourselves here are those which contain pairs of edges for which traversal of the first precludes traversal of the second. These pairs of edges fall into two categories. First, if the above condition regarding traversal of the edges holds for all paths between the edges, then the pair of edges is called an unconditionally impossible pair (UIP). If the condition holds only for a subset of all paths between the edges, then the pair is called a pathwise-impossible pair (PIP). An example of each type of pair is shown in Figure 1.

An algorithm based on techniques presented in [3] for detecting UIP's and PIP's from the branching conditions on the flow graph is described below. A high-level language description of the algorithm is contained in the Appendix.

```
I = 0          a
J = 1          b
IF(X .EQ. Y)   c      I = I + 1   d
IF(I .EQ. 0)   e      L = L + 1   f
IF(J .EQ. 0)   g      K = K + 1   h
CONTINUE       i
```

Figure 1a. A segment of FORTRAN code containing a UIP and a PIP. The pair of statements, J=1 and K=K+1 can never be executed, while the pair, I=0 and L=L+1 cannot be executed if X=Y.

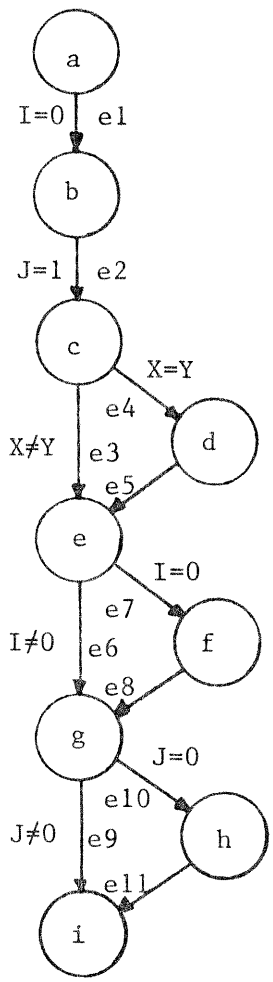

Figure 1b. Flow graph representation of the code segment of Figure 1a. The conflicting branching conditions are shown on the edges. Edges e1 and e6 are a PIP, while edges e2 and e10 are a UIP.

Algorithm

The algorithm takes as input a program unit represented as a flow graph and a set of token-lists (which represent the syntactic composition of each statement), and a set of descriptions of paths to be checked for unexecutability. Then, in four major processing phases (ANNOTATE_GRAPH, DETECT_PIPS, DETECT_UIPS, and TEST_PATHS) and two initialization phases (INIT_PIPS and INIT_UIPS), the algorithm produces 1) a table of PIP's, 2) information as to which PIP's are also UIP's, and 3) for each path read in, an indication as to whether it is unexecutable based on the PIP/UIP analysis.

Phase One: ANNOTATE_GRAPH

The first phase of processing takes the flow graph and annotates it in a manner similar to the flow graph segment in Figure 1b. (We will refer to this flow graph throughout the discussion which follows.) This involves parsing each statement (in token-list form) and deriving from it the conditions (called predicates) required for traversal of the various edges leaving the node representing the statement. These predicates are attached to the appropriate edges for further processing by INIT_PIPS.

Phase Two: INIT_PIPS

Using the predicates generated during ANNOTATE_GRAPH, the second phase of processing creates bit vectors which represent the state of affairs at each node and edge in the flow graph. These vectors are the kill and gen vectors. A description of each vector follows. Selected vectors created for the example in Figure 1 are shown in Table 1.

For each node a bit vector called a kill vector is created and attached to the node. Each component in the kill vector corresponds to an edge. The i-th component of the kill vector for node n, then, is set to one if execution of the statement at node n assigns a value to one or more of the variables in the predicate attached to edge ei. In the kill vector for node a, for example, bit 6 is set to one because execution of node a causes one of the variables in the predicate on edge e6 to be assigned a value.

Table 1.
Selected Bit Vectors
Created by INIT_PIPS

gen(e1) = (1,0,0,0,0,0,0,0,0,0,0)

inc(e1) = (0,0,0,0,0,1,0,0,0,0,0)

gen(e2) = (0,1,0,0,0,0,0,0,0,0,0)

inc(e2) = (0,0,0,0,0,0,0,0,0,1,0)

gen(e6) = (0,0,0,0,0,0,1,0,0,0,0)

inc(e6) = (1,0,0,0,0,0,1,0,0,0,0)

kill(a) = (1,0,0,0,0,1,1,0,0,0,0)

kill(b) = (0,1,0,0,0,0,0,0,1,1,0)

kill(d) = (1,0,0,0,0,1,1,0,0,0,0)

Table 2.
Selected Bit Vectors
Created by DETECT_PIPS

live(e1) = (0,0,1,1,1,1,1,1,0,0,1)

live(e2) = (0,0,1,1,1,1,1,1,1,1,1)

live(e3) = (0,0,0,0,0,1,1,1,1,1,1)

live(e4) = (0,0,0,0,1,0,0,1,1,1,1)

live(e6) = (0,0,0,0,0,0,0,0,1,1,1)

pip(e1) = live(e1) and inc(e1)
= (0,0,1,1,1,1,1,1,0,0,1) and (0,0,0,0,0,1,0,0,0,0,0)
= (0,0,0,0,0,1,0,0,0,0,0)

pip(e2) = (0,0,0,0,0,0,0,0,0,1,0)

pip(e6) = (0,0,0,0,0,0,0,0,0,0,0)

PIP[1] = (1,6)

PIP[2] = (2,10)

For each edge a bit vector called a gen vector is created and attached to the edge. The i-th component in the gen vector corresponds to edge ei. For an edge, the predicate attached to the edge is said to be gen'd at that edge. To indicate this, the algorithm sets a one in the i-th component of the gen vector for edge ei.

One more vector, the inc vector, is created and attached to each edge. Like the gen vector, each component of the inc vector corresponds to an edge. The j-th component of the inc vector for edge ei is set to one if the predicate on edge ej is mutually inconsistent with the predicate on edge ei. A pair of predicates is said to be mutually inconsistent if, when the predicates are compared lexically, they cannot both be true. An example of such inconsistency is the pair of predicates on edges e1 (I=0) and e6 (I≠0) in Figure 1b. Table 1 shows that inc(e1) has a one-bit in position 6.

Phase Three: DETECT_PIPS

The purposes of the third processing phase are 1) to determine which pairs of edges in the flow graph are PIP's, and 2) to build a table of PIP's for use in later processing phases. For this, the bit vectors created by INIT_PIPS are used, and some new bit vectors are created. Examples of the vectors created are in Table 2.

Before describing the processing performed by DETECT_PIPS, let us first consider the conditions which must hold in order for a pair of edges (for example, edges e1 and e6 in Figure 1b) to be a PIP. First, edges e1 and e6 must have mutually inconsistent predicates attached to them. Second, it must be possible to reach edge e6 from edge e1 via at least one path on which no variable in either predicate is assigned a value. The procedure used in DETECT_PIPS, then, must determine the pairs of edges for which these conditions hold.

The processing begins by determining the LIVE sets [4, p.314] for the flow graph using the kill and gen vectors attached to the nodes and edges, respectively. For each edge ei the LIVE algorithm, in effect, moves down the flow graph from edge ei looking for a gen (i.e., a predicate attached to an edge ej) without first encountering a kill (i.e., an assignment to a variable in edge ej's predicate). If it succeeds in finding a gen without an intervening kill on at least one

path starting at edge ei, then edge ej is said to be "live" at edge ei. In Table 2, the live vector for edge e1 has bit 6 set to one indicating that there is at least one path from e1 to e6 (namely, through nodes a, b, c, and e) on which no variable in the predicate on edge e6 is assigned a value (i.e., edge e6 is live at edge e1). On the other hand, bit 9 of live(e1) is set to zero because the value of J is changed at node b on all paths from e1 to e9.

Once the live vectors have been created, it is possible to detect PIP's. This is done for each edge by anding together the live and inc vectors for the edge. The pip vector for edge ei has a one-bit in position j if and only if edge ej is both live at, and mutually inconsistent with, edge ei. Note that these properties satisfy the conditions stated above for PIP's.

Finally, the pairs of edges which are PIP's are stored in the PIP table for use in subsequent processing phases. This is done by examining the pip vector for each edge. For edge ei, each one-bit in pip(ei) represents an edge which is the second edge of a PIP (edge ei being the first edge). The analysis for the flow graph in Figure 1b shows that edges e1 and e6, and edges e2 and e10, are PIP's.

Phase Four: INIT_UIPS

During this phase, the flow graph is marked in a different fashion from that used to detect PIP's. Instead of creating bit vectors containing information about predicates, the bit vectors used for detecting UIP's contain information about PIP's. As a result, the bits in the vectors created by INIT_UIPS correspond to PIP's instead of edges or nodes. The placement of the various vectors on the flow graph differs from that of earlier phases as well. Table 3 contains descriptions of the vectors created by INIT_UIPS for the flow graph of Figure 1b.

First, gen and kill vectors are created for each edge. In these vectors each bit corresponds to a PIP: in the kill vector for edge ei, bit p is set to one if edge ei is the first edge of PIP p; similarly, in the gen vector for edge ei, bit p is set to one if edge ei is the second edge of PIP p.

Next, the killers of each PIP are determined. For each node, a killed vector is created. As in the gen and kill vectors, each bit in the killed vector corresponds to a PIP. In the killed vector for node n, bit p is set to one if a variable in either of the predicates of the pair is assigned a value at node n, i.e. if PIP p is "killed" at node n. In the example, bit one of killed(a) is set to one because execution of the statement at node a causes a variable (namely, I) in PIP[1] to be assigned a value.

Finally, a second set of gen vectors is created. These vectors are attached to the nodes of the flow graph, rather than to the edges. For the start node, the gen vector contains all ones; for all other nodes, the gen vector contains all zeros. In the example, we are assuming that node a is the start node.

Table 3.
Selected Bit Vectors
Created by INIT_UIPS

gen(a) = (1,1)
killed(a) = (1,0)

gen(b) = (0,0)
killed(b) = (0,1)

gen(d) = (0,0)
killed(d) = (1,0)

For all other nodes, gen = (0,0) and killed = (0,0).

kill(e1) = (1,0)
gen(e1) = (0,0)

kill(e2) = (0,1)
gen(e2) = (0,0)

kill(e6) = (0,0)
gen(e6) = (1,0)

kill(e10) = (0,0)
gen(e10) = (0,1)

For all other edges, kill = (0,0) and gen = (0,0).

Phase Five: DETECT_UIPS

This phase determines which of the PIP's found by DETECT_PIPS are also UIP's. To do this, it must be shown that the conditions giving rise to a PIP, namely 1) mutual inconsistency and 2) no variable of either predicate assigned a value, hold for all paths between the pair of edges. This determination is made by using the LIVE and AVAIL algorithms [4] and by combining the resulting vectors. These vectors are listed in Table 4.

The LIVE algorithm determines the LIVE sets for each node in the flow graph, using the gen and kill vectors on the edges as created by INIT_UIPS. In the live vector for node n, bit p is set to one if and only if node n is situated such that, on at least one path starting at node n, the second edge of PIP p is reached without traversing the first edge. (Node "c" in Figure 1b is such a node.) This means that if node n is reachable from the first edge of the PIP, then on some path node n falls between the first edge and the second in PIP p. Whether or not node n is reachable from the first edge will be determined next.

After the live vectors have been created, the AVAIL algorithm is run on the flow graph, using the gen vectors on the nodes and the kill vectors on the edges, as created by INIT_UIPS. This algorithm creates an avail vector for each node defined as follows: bit p of the avail vector for node n is zero if and only if node n is reachable from the first edge of PIP p. The AVAIL algorithm, in effect, searches the flow graph upward from each node n, looking for a gen on all paths leading into node n. If, on all paths, a gen (which is only on the start node) is found without first encountering a kill (i.e., the first edge of a PIP), then the algorithm sets a one-bit in the avail vector for node n. If a kill is seen on any path leading into node n, then the bit is set to zero: if this is the case, it means that the node is reachable from the first edge of a PIP.

Once the live and avail vectors have been created, it is possible to determine for which PIP's a killer node can lie on a path from the first edge to the second. If no killers can lie on any path, then the PIP is also a UIP. The procedure for making this determination produces a vector, called nip, for each node. The nip vector for a node n is the result of anding together 1) the complement of avail(n), 2) killed(n),

and 3) live(n).

Table 4.
Selected Bit Vectors
Created by DETECT_UIPS

live(a) = (0,0)	avail(a) = (1,1)
live(b) = (1,0)	avail(b) = (0,1)
live(c) = (1,1)	avail(c) = (0,0)
live(d) = (1,1)	avail(d) = (0,0)
live(e) = (1,1)	avail(e) = (0,0)
live(f) = (0,1)	avail(f) = (0,0)
live(g) = (0,1)	avail(g) = (0,0)
live(h) = (0,0)	avail(h) = (0,0)
live(i) = (0,0)	avail(i) = (0,0)

nip(a) = (not avail(a) and killed(a) and live(a))
= (not (1,1) and (1,0) and (0,0))
= ((0,0) and (1,0) and (0,0))
= (0,0)

nip(b) = (0,0)

nip(d) = (not avail(d) and killed(d) and live(d))
= (not (0,0) and (1,0) and (1,1))
= ((1,1) and (1,0) and (1,1))
= (1,0)

For all other nodes, nip = (0,0), since for all other nodes killed = (0,0).

nipairs = (1,0)

uip = (0,1)

The nip vector for node n has bit p set to one if and only if 1) node n is reachable from the first edge of PIP p; 2) PIP p is killed by node n; and 3) node n is between the first and second edges of PIP p on at least one path. This means that if bit p is set to one, PIP p cannot be a UIP because at least one path from the first to the second edge must pass through the killer node n.

If the nips for all nodes are then ored together, the resulting vector has a one-bit in position p if and only if PIP p cannot be a UIP. Therefore if this vector is complemented, the resulting vector, called uip, has a one-bit in position p if and only if PIP p is also a UIP. This information is stored for later use in detecting unexecutable paths. In the example in Figure 1, edges e2 and e10 (PIP[2]) are shown to be a UIP through this analysis.

Phase Six: TEST_PATHS

The TEST_PATHS processing phase performs unexecutability analyses on the paths supplied to it. The method makes use of the information obtained by the foregoing processing phases to determine for each input path whether or not it is unexecutable based on the presence of PIP's or UIP's on the path. (Note that the presence of a UIP alone is enough to declare the path to be unexecutable; however, in the case of a PIP occurring on the path it must also be determined that the given path does not pass through a killer node on the way from the first edge of the PIP to the second).

The sixth processing phase begins by creating a bit vector, called killers, for each edge. In the killers vector for edge ei, bit n is set to one if node n kills (i.e., causes assignment of a value to a variable in) the predicate on edge ei. The paths are then read in, one at a time, and inspected.

The procedure used in determining the unexecutability of a path is as follows: first, two bit vectors are created to facilitate manipulation of the path description. These vectors are called nodesreached and edgesreached. In the nodesreached vector, bit n is set to one if the path passes through node n. Similarly, in the edgesreached vector, bit i is set if edge ei is traversed on the path.

Next, the path is examined edge by edge to determine if any two edges on the path are a PIP. This is done by anding the edgesreached vector and the pip vector for each edge ei. If the resulting vector has a one-bit in position j, then the PIP (ei,ej) is on the path being tested. Since the existence of a PIP implies that one or more paths between edges ei and ej are unexecutable, it must be determined whether the given path contains one of those paths.

To make this determination, the following procedure is used: first, the nodesreached vector is anded with killers(ej). If the result of this operation is a vector of all zeros, then the given path does not pass through any of the nodes which kill the predicate on edge ei. This means that the path is unexecutable.

If, however, there are one-bits in the resulting vector, this indicates that the path passes through one or more of edge ei's killers. For the path to be declared unexecutable, then, it must be shown that none of the killer nodes fall between edges ei and ej. The simplest way to make this determination, it appears, is through inspection of the path, node by node. If a killer node is found to lie between edges ei and ej, then no definitive statement can be made about the unexecutability of the path; it may be executable, or unexecutable for reasons which are beyond the scope of this analysis.

A path which does not traverse any PIP's, of course, cannot be declared to be unexecutable by this analysis. Similarly, a path containing a UIP ("abceghi", for example) is always unexecutable and the above analysis concerning PIP's need not be performed.

To illustrate and further clarify the above discussion, let us follow the processing which would be done for an actual path, say "abcegi". Figure 5 contains the bit vectors which would be created for the processing of this path.

First, the algorithm builds the edgesreached and nodesreached vectors. Next, the path is inspected edge by edge. The first edge to be considered, edge e1, is the first edge of a PIP, as can be seen by the result of edgesreached and pip(e1). The vector has a one-bit in position 6, indicating that the PIP (e1,e6) is on the path. Furthermore, the result of nodesreached and killers(e1) contains a one-bit in the first bit position (indicating node a). The implication of

these two results is that the path is unexecutable provided that node a does not lie between edges e1 and e6. Since this is the case, the path "abcegi" is declared to be unexecutable by the algorithm.

That the path is indeed unexecutable can easily be seen by visual inspection of the FORTRAN code in Figure 1a. If variable I is set to zero at node a and is not reset (by passing through node d), then any path which does not pass through node f (the true branch of IF(I.EQ.0)) must be unexecutable. The given path, "abcegi", is such a path.

Table 5.
Selected Bit Vectors
Created by TEST_PATHS

killers(e1) = (1,0,0,1,0,0,0,0,0)
killers(e2) = (0,1,0,0,0,0,0,0,0)
killers(e6) = (1,0,0,1,0,0,0,0,0)
killers(e7) = (1,0,0,1,0,0,0,0,0)
killers(e9) = (0,1,0,0,0,0,0,0,0)
killers(e10)= (0,1,0,0,0,0,0,0,0)

Given path:
a b c e g i

nodesreached = (1,1,1,0,1,0,1,0,1)
edgesreached = (1,1,1,0,0,1,0,0,1,0,0)

edgesreached and pip(e1) = (0,0,0,0,0,1,0,0,0,0,0)

nodesreached and killers(e1) = (1,0,0,0,0,0,0,0,0)

Predicate Characteristics

The algorithm presented here is based largely on the concepts and algorithms presented in [3]. As noted there, the methods used for detecting PIP's and UIP's represent a heuristic approach based on what can reasonably be expected to occur in a typical program.

One major assumption made by this approach is that the predicates which are involved in PIP's and UIP's are simple enough that the determination of mutual inconsistency (which is, in the general case, an unsolvable problem [3, p.16]) can easily be made by means of a simple truth table (see INC in the Appendix).

This assumption seems to be borne out by empirical evidence, collected through examination of a considerable amount of program code [5,6], which indicates that most of the predicates found in an ordinary program are indeed quite simple, often in the form "variable relop constant", where relop stands for a relational operator, such as ".EQ.". In light of this evidence, and in order to keep the determination of mutual inconsistency as simple as possible, the algorithm contains a function which will decide whether a particular predicate is "useful" to the analyses to be performed. This function will declare as useful only simple predicates such as those noted above and will reject all others, which will be replaced with the predicate true (which cannot be inconsistent with any other predicate). This raises the possibility of missing some PIP's and UIP's, but will not introduce fallacious PIP's or UIP's.

The canonical form to be used for storing and comparing predicates is also, to a certain extent, affected by the assumed simplicity of predicates. The form to be used is patterned after the expected form of most predicates, stated above. This form will have the variable highest in alphanumeric order to the left of the relational operator, and the remainder of the predicate on the right. This will produce a predicate of the most desirable form (i.e., "variable relop constant") in most cases.

Future Work

It is currently proposed that the algorithm described here be implemented and integrated into the DAVE system for testing. During this testing phase, it will be determined whether the assumptions made concerning the predicates in an ordinary program are borne out by empirical evidence. It is also hoped that data can be collected concerning the frequency of PIP's and UIP's in programs, and the efficacy of the heuristic static analysis techniques in detecting these phenomena.

If experience with the implemented algorithm shows that the techniques used are inadequate, there are several courses of action which may be taken.

Should the predicates in a typical program prove to be more complex than expected, the algorithm could be changed to allow these more complex predicates to be analyzed and the inconsistency determination made more sophisticated by taking more special cases into account. These cases can be determined by using the data collected during the testing phase.

Other possible improvements and additions to the algorithm described here include 1) an analysis of conditions which hold on all paths into the first node in a path being tested; 2) additional sophistication in handling predicates of the form "variable = variable + constant", including folding of constants and using known values for variables; and 3) the ability to perform the analyses across program unit boundaries. Each of these improvements would result in some increase in the power to detect unexecutable paths, but the cost of adding such features should be carefully weighed against the potential benefits. The information needed to make decisions concerning these additions will be provided, at least in part, by the basic algorithm described in this paper.

Conclusion

An algorithm for detecting impossible pairs of edges and for determining the unexecutability of some paths through a program flow graph based on the branching conditions of the graph has been presented. The technique, while simple, is believed to be sufficiently powerful to improve the quality of the data flow analyses performed by the DAVE system by eliminating many unexecutable paths from consideration.

Acknowledgements

The author wishes to thank Lee Osterweil for giving much-needed advice on how to go about constructing the algorithm presented here. Thanks also go to Lloyd Fosdick for his help in reading and correcting several drafts of this paper.

References

1. Osterweil, L. J. and Fosdick, L. D., "DAVE--A Validation Error Detection and Documentation System for Fortran Programs", Software--Practice and Experience, 6(4) (1976), pp. 473-486.

2. Hopcroft, J. E. and Ullman, J. D., Formal Languages and Their Relation to Automata, Addison Wesley, Reading, Mass., 1969, pp. 108-109.

3. Osterweil, L. J., "The Detection of Unexecutable Program Paths Through Data Flow Analysis", University of Colorado Department of Computer Science Technical Report No. CU-CS-110-77.

4. Fosdick, L. D. and Osterweil, L. J., "Data Flow Analysis in Software Reliability", ACM Computing Surveys, 8(3) (September, 1976), pp. 305-330.

5. Knuth, D. E., "An Empirical Study of FORTRAN Programs", Software--Practice and Experience, 1(2) (1971), pp. 105-135.

6. Elshoff, J. L., "A Numerical Profile of Commercial PL/I Programs", Software--Practice and Experience, 6(4) (1976), pp. 505-525.

Appendix

Algorithm for Detecting Unexecutable Paths

Notation:

The language used in the following description is a mixture of ALGOL 60 and PASCAL. Where appropriate, constructs from both languages have been used. The following conventions have been used in the code:

1. Procedure names which are defined in this description are printed in upper case; other procedures, printed in lower case, are not defined but are simple enough that the reader should have no trouble supplying an appropriate mental definition.
2. A single entry in a table is referenced with the construct, "table[n]"; a field within the table entry is referenced by "table[n].field".
3. Variables are not declared. Their types are assumed to be whatever is necessary to fit the situation (e.g, integers for loop indices, vectors to hold bit vectors, etc.)
4. All predicates are assumed to fit into one storage location, but the tokens making up a predicate may be addressed by their position in the predicate. For example, in the predicate "I=0", predicate[2] is "=".

Input:

1. Source program, reduced to a flow graph and token lists
2. Path descriptions for testing for unexecutablity

Output:

1. Impossible pairs
2. Executablity information (i.e., "unexecutable" or "can't tell") for each input path

Procedure:

```
begin

   ANNOTATE_GRAPH;
   INIT_PIPS;
   DETECT_PIPS;
   INIT_UIPS;
   DETECT_UIPS;
   TEST_PATHS

end.
```

ANNOTATE_GRAPH
Generates predicates and
attaches them to flow graph

Input:

1. Token-list representation of a program unit
2. Node table, containing the following fields:
   1) Statement label associated with this node (if any)
   2) Ptrs to predecessor nodes
   3) Ptrs to successor nodes
3. Edge table, containing the following fields:
   1) Ptr to head node of this edge
   2) Ptr to tail node of this edge
   3) Predicate (unset)

Output:

1. Edge table fields:
   1) Predicate

2. Node table, possibly with new nodes added

Procedure:

```
begin

   while  stmt type ≠ 'END' do

      begin

         get statement;

         case  stmt type  of
           assignment:  ASGMNT;
           data:  DATA;
           arith-if:  ARITHIF;
           logical-if:  LOGIF;
           comp. goto:  CGOTO;
           assigned goto:  AGOTO;
           assign:  ASSIGN;
           do:  DO;
           goto:  GOTO;
           else:  OTHER;
         end

      end

end.
```

ASGMNT
Handles predicate generation for
assignment statements

Input:

1. Token-list representation of an assignment statement
2. Node # ["N"] of node representing the statement
3. Edge and Node tables

Output:

1. Predicate in canonical form, attached to the appropriate edge

Syntax of Assignment Statement:

```
stmt ::= var '=' expr
var ::= simple var | subscripted var
simple var ::= ident
subscripted var ::= ident '(' subscript list ')'
subscript list ::= subscript |
                   subscript ',' subscript |
                   subscript ',' subscript ',' subscript
subscript ::= int const | int const '*' sum | sum
sum ::= ident | ident '+' int const | ident '-' int const
```

Procedure:

```
begin

   parse statement;
   predicate := statement;
   get edge(N -> N+1);
   CHECKANDSTORE(predicate,edge)

end.
```

DATA
Handles predicate generation for
DATA statements

Input:

1. Token-list representation of a DATA statement
2. Node # ["N"] of node representing the statement
3. Edge and Node tables

Output:

1. Predicate(s) in canonical form, attached to the appropriate edge(s). Note: new edges may be constructed to hold the predicates, since only one predicate is allowed per edge.

Syntax of DATA Statement:

```
stmt ::= 'DATA' datalist
datalist ::= varlist '/' valuelist '/' | datalist ',' datalist
varlist ::= var | var ',' varlist
valuelist ::= constant | repcount '*' constant | valuelist ',' valuelist
var ::= simple var | subscripted var
simple var ::= ident
subscripted var ::= ident '(' subscript list ')'
subscript list ::= subscript |
                   subscript ',' subscript |
                   subscript ',' subscript ',' subscript
subscript ::= int const | int const '*' sum | sum
sum ::= ident | ident '+' int const | ident '-' int const
repcount ::= int const
```

Procedure:

```
begin

   set ptr1 to token following 'DATA';
   set ptr2 to token following '/';

   while token[ptr2] ≠ eos do

      begin

         while token[ptr2] ≠ '/' do

            begin
              parse value;
              if token[ptr2] = '*' then
                 begin
                    repcount := value;
                    parse value;
                 end
```

```
                else
                   repcount := 1;
                for i := 1 to repcount do
                   begin
                      parse var;
                      construct predicate "var=value";
                      construct edge(N -> N+1);
                      CHECKANDSTORE(predicate,edge)
                   end
             end;

          if token[ptr2+1] ≠ eos then
             begin
                ptr1 := ptr2 + 2;
                while token[ptr2] ≠ '/' do
                   ptr2 := ptr2 +1;
                ptr2 := ptr2 + 1
             end

       end

end.
```

ARITHIF
Handles predicate generation for
arithmetic IF statements

Input:

1. Token-list representation of an arithmetic IF statement
2. Node # ["N"] of node representing the statement
3. Edge and Node tables

Output:

1. Predicates in canonical form, attached to the appropriate edges

Syntax of Arithmetic IF Statement:

stmt ::= 'IF' '(' expr ')' int const ',' int const ',' int const

Procedure:

```
begin

   parse expr;

   construct predicate "expr < 0";
   set ptr to first integer constant;
   get edge(N -> node labelled with token[ptr]);
   CHECKANDSTORE(predicate,edge);

   construct predicate "expr = 0";
   set ptr to second integer constant;
   get edge(N -> node labelled with token[ptr]);
   CHECKANDSTORE(predicate,edge);

   construct predicate "expr > 0";
   set ptr to third integer constant;
   get edge(N -> node labelled with token[ptr]);
   CHECKANDSTORE(predicate,edge)

end.
```

LOGIF
Handles predicate generation for
logical IF statements

Input:

1. Token-list representation of a logical IF statement
2. Node # ["N"] of node representing the statement
3. Edge and node tables

Output:

1. Predicates in canonical form, attached to the appropriate edges

Syntax of Logical IF Statement:

```
stmt ::= 'IF' '(' boolean expr ')' statement
```

Procedure:

```
begin

   parse boolean expr;

comment  true branch;

   get edge(N -> node representing "statement");
   predicate := boolean expr;
   CHECKANDSTORE(predicate,edge);

comment  false branch;

   get edge(N -> N+1);
   predicate := not boolean expr;
   CHECKANDSTORE(predicate,edge)

end.
```

CGOTO
Handles predicate generation for
computed GOTO statements

Input:

1. Token-list representation of a computed GOTO statement
2. Node # ["N"] of node representing the statement
3. Edge and node tables

Output:

1. Predicates in canonical form, attached to the appropriate edges

Syntax of Computed GOTO Statement:

```
stmt ::= 'GOTO' '(' label list ')' ',' ident
label list ::= integer const | integer const ',' label list
```

Procedure:

```
begin

   set ptr to token following '(';
   counter := 0;
   parse ident;

   while token[ptr] ≠ ')' do

      begin
         counter := counter + 1;
         construct predicate "ident = counter";
         get edge(N -> node labelled with token[ptr]);
         CHECKANDSTORE(predicate,edge);

         ptr := ptr + 1;
         if token[ptr] = ',' then
            ptr := ptr + 1;
      end

end.
```

AGOTO
Handles predicate generation for
assigned GOTO statements

Input:

1. Token-list representation of an assigned GOTO statement
2. Node # ["N"] of node representing the statement
3. Edge and node tables

Output:

1. Predicates in canonical form, attached to the appropriate edges

Syntax of Assigned GOTO Statement:

```
stmt ::= 'GOTO' ident ',' '(' label list ')'
label list ::= integer const | integer const ',' label list
```

Procedure:

```
begin

   parse ident;
   set ptr to token following '(';

   while token[ptr] ≠ ')' do

      begin
         construct predicate "ident = token[ptr]";
         get edge(N -> node labelled with token[ptr]);
         CHECKANDSTORE(predicate,edge);

         ptr := ptr + 1;
         if token[ptr] = ',' then
            ptr := ptr + 1;
      end

end.
```

ASSIGN
Handles predicate generation for
ASSIGN statements

Input:

1. Token-list representation of an ASSIGN statement
2. Node # ["N"] of node representing the statement
3. Edge and node tables

Output:

1. Predicate in canonical form, attached to the appropriate edge

Syntax of ASSIGN Statement:

```
stmt ::= 'ASSIGN' integer const 'TO' ident
```

Procedure:

```
begin

   parse integer constant, ident;
   construct predicate "ident = const";
   get edge(N -> N+1);
   CHECKANDSTORE(predicate,edge);

end.
```

DO
Handles predicate generation for
DO statements

Input:

1. Token-list representation of a DO statement
2. Node # ["N"] of node representing the statement
3. Edge and node tables

Output:

1. Predicates in canonical form, attached to the appropriate edges
2. New entries in the Node table, for test and increment nodes

Syntax of DO Statement:

```
stmt ::= 'DO' termstmt index '=' paramlist
termstmt ::= integer const
index ::= ident
paramlist ::= initial ',' final | initial ',' final ',' increment
initial ::= integer const | simple var
final ::= integer const | simple var
increment ::= integer const | simple var
simple var ::= ident
```

Procedure:

```
begin

   parse termstmt, index, initial, final;
   if token[ptr] = ',' then
      parse increment
   else
      increment := 1;

comment  edge from DO to 1st stmt in loop;

   construct predicate "index = initial";
   get edge(N -> N+1);
   CHECKANDSTORE(predicate,edge);

comment  fall-through edge;

   create node(y);
   node := node labelled with termstmt;
   get edge(node -> node+1);
   insert node(y,edge);
   if increment = 1 then
      construct predicate "index = final"
   else
      construct predicate "index + increment > final";
   get edge(y -> node+1);
   CHECKANDSTORE(predicate,edge);
```

```
comment  edge from termstmt node to increment node;

   create node(z);
   get edge(y -> N+1);
   insert node(z,edge);
   if increment = 1 then
      construct predicate "index < final"
   else
      construct predicate "index + increment ≤ final";
   get edge(y -> z);
   CHECKANDSTORE(predicate,edge);

comment  edge from increment node back to top of loop;

   get edge(z -> N+1);
   construct predicate "index = index + increment";
   CHECKANDSTORE(predicate,edge)

end.
```

GOTO
Handles predicate generation for
GOTO statements

Input:

1. Token-list representation of a GOTO statement
2. Node # ["N"] of node representing the statement
3. Edge and Node tables

Output:

1. Predicate true, attached to the appropriate edge.

Syntax of GOTO Statement:

```
stmt ::= 'GOTO' integer const
```

Procedure:

```
begin

   parse integer const;
   get edge(N -> node labelled with integer const);
   CHECKANDSTORE(true,edge);

end.
```

OTHER
Handles predicate generation for
statements not handled elsewhere

Input:

1. Token-list representation of a statement
2. Node # ["N"] of node representing the statement
3. Edge & Node tables

Output:

1. Predicate true, attached to the appropriate edge.

Procedure:

```
begin

   get edge(N -> N+1);
   CHECKANDSTORE(true,edge);

end.
```

USEFUL(predicate)
Determines the "usefulness" of a predicate
for use in later processing

Input:

1. Predicate, in token-list form

Output:

1. true if predicate is usable for detecting PIPs, UIPs;
   false otherwise.

Procedure:

```
begin

   if length of predicate ≤ 3 then
      USEFUL := true
   else
      begin
         if length of predicate = 4 then
            case form of predicate of
               'v relop -c', '.NOT. v relop v', '.NOT. v relop c',
               '.NOT. c relop v', '.NOT. c relop c':  USEFUL := true;
               else: USEFUL := false
         else
            USEFUL := false
      end

end.
```

CANON(predicate)
Converts a predicate to
canonical form

Input:

1. "Useful" predicate, in any form

Output:

1. Predicate, in canonical form

Procedure:

```
begin

   case length of predicate of
      1:  pred := predicate + '= true';
      2:  pred := predicate[2] + '≠ true';
      3:  case predicate[1].type of
             variable :  if predicate[3].type = variable
                            and predicate[1] > predicate[3] then
                               pred := REVERSE(predicate)
                         else
                               pred := predicate;
             constant :  if predicate[3].type = variable then
                               pred := predicate
                         else
                               if predicate[1] > predicate[3] then
                                  pred := REVERSE(predicate)
                               else
                                  pred := predicate;
                    '(' :  pred := CANON(predicate[2]);
          end;
      4:  case form of predicate of
             '.NOT. v relop v', '.NOT. v relop c', '.NOT. c relop v',
             '.NOT c relop c':  begin
                                   case predicate[2] of
                                      '=' : relop := '≠';
                                      '≠' : relop := '=';
                                      '>' : relop := '≤';
                                      '≥' : relop := '<';
                                      '<' : relop := '≥';
                                      '≤' : relop := '>';
                                   end;
                                   pred := predicate[2] + relop + predicate[4];
                                   pred := CANON(pred)
                                end;
                    'v relop -c':  begin
                                      c := predicate[4]; c := -c;
                                   pred := predicate[1] + predicate[2] + c;
                                   pred := CANON(pred)
                                end;
      end;

      CANON := pred

end.
```

REVERSE(predicate)
Reverses the order of the two operands
in a relational expression

Input:

1. Predicate of length 3, of one of the following forms:
   ′ v relop v′ ′v relop c′ ′c relop v′ ′c relop c′
   ′bv bop bv′ ′bv bop bc′ ′bc bop bv′ ′bc bop bc′
   where v, c are arithmetic variable, constant and relop is a relational operator; and bv, bc are boolean variable, constant and bop is a boolean operator.

Output:

1. Predicate with tokens 1 and 3 interchanged and relop (if present) changed to reflect the interchange.

Procedure:

```
begin

   case predicate[2] of
      '>' : relop := '<';
      '≥' : relop := '≤';
      '<' : relop := '>';
      '≤' : relop := '≥';
      '=' : relop := '=';
      '≠' : relop := '≠';
      '.OR.' : relop := '.OR.';
      '.AND.' : relop := '.AND.';
   end;

   REVERSE := predicate[3] + relop + predicate[1]

end.
```

INIT_PIPS
Builds bit vectors needed
for DETECT_PIPS

Input:

1. Node table from ANNOTATE_GRAPH
2. Edge table from ANNOTATE_GRAPH

Output:

1. Node table, with the following fields set:
   1) Ptr to KILL vector

2. Edge table, with following fields set:
   1) Ptr to GEN vector
   2) Ptr to INC vector

Procedure:

```
begin

comment  N = # nodes in the flow graph.  E = # edges;

   for n := 1 to N do
      SET_KILL(n);

   for e := 1 to E do
      begin
         SET_GEN(e);
         SET_INC(e);
      end;

end.
```

SET_KILL(n)
Creates KILL vector for a node

Input:

1. Ptr to node ["n"].
2. Edge table

Output:

1. Bit vector, ptr stored in Node table

Procedure:

```
begin

   v := newvector(E);

   for e := 1 to E do
      if node n resets a variable in edgetable[e].predicate then
         setbit(e,v);

   nodetable[n].kill := v

end.
```

SET_GEN(e)
Creates GEN vector for an edge

Input:

1. Ptr to edge ["e"]

Output:

1. Bit vector, ptr stored in Edge table

Procedure:

```
begin

   v := newvector(E);
   setbit(e,v);
   edgetable[e].gen := v

end.
```

SET_INC(e)
Creates INC vector for an edge

Input:

1. Ptr to edge ["e"]

Output:

1. Bit vector, ptr stored in Edge table

Procedure:

```
begin

   v := newvector(E);

   for i := 1 to E do
      if INC(e,i) then
         setbit(i,v);

   edgetable[e].inc := v

end.
```

INC(e1,e2)
Determines mutual inconsistency
of a pair of predicates

Input:

1. Ptrs to edges ["e1","e2"]
2. Edge table fields:
   1) Predicate attached to the edge

Output:

1. true, if predicates on edges e1 and e2 are mutually inconsistent;
   false, if not inconsistent or can't tell.

Procedure:

```
begin

   INC := false;

   p1 := edgetable[e1].predicate;
   p2 := edgetable[e2].predicate;

   if variables used in p1 = variables used in p2 then
      case form of p1 of
         'v relop c':  begin
                          if p1.c = p2.c then
                             INC := table1[p1.relop,p2.relop];
                          if p1.c > p2.c then
                             INC := table2[p1.relop,p2.relop];
                          if p1.c < p2.c then
                             INC := table3[p1.relop,p2.relop];
                       end;
         'v relop v':  INC := table1[p1.relop,p2.relop];
         'bv relop bc': begin
                          if p1.bc = p2.bc then
                             case p1.relop of
                                '=':  if p2.relop = '≠' then INC := true;
                                '≠':  if p2.relop = '=' then INC := true;
                             end;
                          if p1.bc ≠ p2.bc then
                             case p1.relop of
                                '=':  if p2.relop = '=' then INC := true;
                                '≠':  if p2.relop = '≠' then INC := true;
                             end;
                       end;
         'bv relop bv': case p1.relop of
                          '=':  if p2.relop = '≠' then INC := true;
                          '≠':  if p2.relop = '=' then INC := true;
                       end;
      end

end.
```

Table 1.

p1.relop \ p2.relop	=	≠	<	≤	>	≥
=	F	T	T	F	T	F
≠	T	F	F	F	F	F
<	T	F	F	F	T	T
≤	F	F	F	F	T	F
>	T	F	T	T	F	F
≥	F	F	T	F	F	F

Table 2.

p1.relop \ p2.relop	=	≠	<	≤	>	≥
=	T	F	T	T	F	F
≠	F	F	F	F	F	F
<	F	F	F	F	F	F
≤	F	F	F	F	F	F
>	T	F	T	T	F	F
≥	T	F	T	T	F	F

Table 3.

p1.relop \ p2.relop	=	≠	<	≤	>	≥
=	T	F	F	F	T	T
≠	F	F	F	F	F	F
<	T	F	F	F	T	T
≤	T	F	F	F	T	T
>	F	F	F	F	F	F
≥	F	F	F	F	F	F

DETECT_PIPS
Creates a table of PIPs

Input:

1. Edge table fields:
   1) Ptr to GEN vector
   2) Ptr to INC vector

2. Node table fields:
   1) Ptr to KILL vector

Output:

1. Edge table fields:
   1) Ptr to LIVE vector
   2) Ptr to PIP vector

2. PIP table fields:
   1) Edge # of first edge of PIP
   2) Edge # of second edge of PIP

Procedure:

```
begin

   LIVE(1);

   for ei := 1 to E do
      begin
         edgetable[ei].pip := edgetable[ei].live and edgetable[ei].inc;
         for ej := each 1-bit in edgetable[ei].pip do
            store (ei,ej) in piptable;
      end

end.
```

INIT_UIPS
Builds bit vectors needed
for DETECT_UIPS

Input:

1. Node table from ANNOTATE_GRAPH
2. Edge table from ANNOTATE_GRAPH
3. PIP table from DETECT_PIPS

Output:

1. Node table, with the following fields set:
   1) Ptr to GEN vector
   2) Ptr to KILL vector
   3) Ptr to KILLED vector

2. Edge table, with following fields set:
   1) Ptr to GEN vector
   2) Ptr to KILL vector

Procedure:

```
begin

comment  P = # entries in PIP table;

   for e := 1 to E do
      begin
         edgetable[e].gen := newvector(P);
         edgetable[e].kill := newvector(P);
      end;

   for p := 1 to P do
      begin
         setbit(p,edgetable[piptable[p].first].kill);
         setbit(p,edgetable[piptable[p].second].gen);
      end;

   for n := 1 to N do
      begin
         MARK_PIPS_KILLED(n);
         if n = start node then
            nodetable[n].gen := not newvector(P)
         else
            nodetable[n].gen := newvector(P)
      end;

end.
```

MARK_PIPS_KILLED(n)
Creates KILLED vector for a node

Input:

1. Ptr to node ["n"].

Output:

1. Bit vector, ptr stored in Node table. Each 1-bit in KILLED represents a PIP which is "killed" by the node n (i.e., a variable in either the first edge or the second edge's predicate is reset).

Procedure:

```
begin

   nodetable[n].killed := newvector(P);

   for p := 1 to P do
      begin
         i := piptable[p].first;
         j := piptable[p].second;
         if bit i or bit j of nodetable[n].kill is set then
            setbit(p,nodetable[n].killed)
      end

end.
```

DETECT_UIPS
Determines which of the PIPs are
unconditionally impossible pairs

Input:

1. Node table fields:
   1) Ptr to KILLED vector
   2) Ptr to GEN vector

2. Edge table fields:
   1) Ptr to KILL vector
   2) Ptr to GEN vector

Output:

1. PIP table fields:
   1) UIP -- 1 if PIP is a UIP, otherwise 0

Procedure:

```
begin

   LIVE(2);
   AVAIL;

   nipairs := newvector(P);

   for n := 1 to N do
      begin
         nip[n] := ( not nodetable[n].avail and nodetable[n].killed and
                     nodetable[n].live);
         nipairs := nipairs or nip[n];
      end;

   uip := not nipairs;

   for i := 1 to P do
      piptable[i].uip := bit i of uip;

end.
```

LIVE(i)
Determines LIVE sets for a flow graph

Input:

1. Flag ("i") indicating how the flow graph is marked:
   1: KILLs on nodes, GENs and LIVEs on edges;
   2: KILLs and GENs on edges, LIVEs on nodes
2. Edge table
3. Node table

Output:

1. LIVE vectors, placed on edges or nodes

Procedure:

```
begin

comment  The procedure used here is adapted from the algorithm of the same
         name in Fosdick, L.D., and Osterweil, L.J., "Data Flow Analysis
         in Software Reliability", Comp. Surveys 8(3) (Sept. 1976) 305-330;

case i of

  1: begin

       for e := 1 to E do
          edgetable[e].live := newvector(E);

       change := true;
       while change do
          begin
             change := false;
             for e := 1 to E do
                begin
                   previous := edgetable[e].live;
                   v := newvector(E);
                   for k := each exit edge from edgetable[e].head do
                      v := v or (edgetable[k].live or edgetable[k].gen);
                   v := v and not nodetable[edgetable[e].head].kill;
                   edgetable[e].live := v;
                   if v ≠ previous then
                      change := true;
                end
          end

     end;
```

```
2:  begin

      for n := 1 to N do
         nodetable[n].live := newvector(P);

      change := true;
      while change do
         begin
            change := false;
            for n := 1 to N do
               begin
                  previous := nodetable[n].live;
                  v := newvector(P);
                  for k := each exit edge from node n do
                     v := v or ((nodetable[edgetable[k].head].live and
                                 not edgetable[k].kill) or edgetable[k].gen);
                  nodetable[n].live := v;
                  if v ≠ previous then
                     change := true;
               end
         end

   end

end.
```

AVAIL
Determines the AVAIL sets for
a flow graph

Input:

1. Node table
2. Edge table

Output:

1. Node table fields:
   1) Ptr to AVAIL vector

Procedure:

```
begin

comment  The procedure used here is adapted from the algorithm of the same
         name in Fosdick, L.D., and Osterweil, L.J., "Data Flow Analysis
         in Software Reliability", Comp. Surveys 8(3) (Sept. 1976) 305-330;

   for n := 2 to N do
      nodetable[n].avail := not newvector(P);

   nodetable[1].avail := newvector(P);

   change := true;
   while change do
      begin
         change := false;
         for n := 2 to N do
            begin
               previous := nodetable[n].avail;
               v := not newvector(P);
               for k := each in-edge of node n do
                  v := v and ((nodetable[edgetable[k].tail].avail and
                                 not edgetable[k].kill)
                                 or nodetable[edgetable[k].tail].gen);
               nodetable[n].avail := v;
               if v ≠ previous then
                  change := true;
            end
      end

end.
```

TEST_PATHS
Tests paths for unexecutablity

Input:

1. Descriptions of paths to be tested

Output:

1. Statement of results of test, either "unexecutable" or "can't tell"

Procedure:

```
begin

   for e := 1 to E do
      MARK_EDGE_KILLERS(e);

   while more paths do
      begin
         read(path);
         if path contains UIP or UNEXEC(path) then
            Output("path is unexecutable:",path)
         else
            Output("cannot determine executablity of path:",path)
      end

end.
```

MARK_EDGE_KILLERS(e)
Creates KILLERS vector for an edge

Input:

1. Ptr to edge ["e"]

Output:

1. Bit vector, ptr stored in Edge table

Procedure:

```
begin

   v := newvector(N);

   for i := 1 to N do
      if bit e of nodetable[i].kill is set then
         setbit(i,v);

   edgetable[e].killers := v

end.
```