TRANSLATION OF SIMPLE C PROGRAMS TO PROGRAM DEPENDENCE WEBS

By

PREMKUMAR JOHN

Bachelor of Engineering

Regional Engineering College

Trichy, Tamil Nadu, India

1992

Submitted to the faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
July 1994

OKLAHOMA STATE UNIVERSITY

TRANSLATION OF SIMPLE C PROGRAMS TO

PROGRAM DEPENDENCE WEBS

Thesis Approved:

Mansur Samadzadeh
Thesis Advisor

Huizhu Lu

Blayne E. Mayfield

Thomas C. Collins
Dean of the Graduate College

ACKNOWLEDGEMENTS

I thank my graduate advisor, Dr. Mansur H. Samadzadeh, for his advice, assistance, guidance, and continuous encouragement that helped me to successfully complete my degree. His constructive criticism and his interest helped me in the completion of my thesis. I also thank him for educating me in technical writing. I appreciate his moral support.

I would like to express my gratitude to my parents for their support, encouragement, and confidence in my ability to continue my education. I would also like to thank my brother, his wife, and my uncle for their moral and financial support. I also thank my friends for their moral and financial support.

My sincere thanks to Drs. Blayne Mayfield and Huizhu Lu for serving on my graduate committee. I also want to thank Mr. Jim McGee, Mr. Andy Adsit, Mr. Matthew Link, and my colleagues at the University Computer Center, OSU, for allowing flexible working hours.

Thank You God for making the impossible possible in my life. Thank You for everything You have given to me. Thank You for all You have given to me in my life.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER I

INTRODUCTION

Pictorial representation has been one of the oldest forms of communication among humans. Over time, it has found its place in all fields of science including the field of computer science. State diagrams, Petri nets, and flow graphs are some pictorial representations that are used in representing the static or dynamic aspects of programs. Among them, flow graphs have been used in the optimization of programs, specifically in the compiling process. They are used in the field of software engineering to depict and analyze the structural complexity of a program. They also help in the maintenance process of software systems.

The three types of flow graphs that are widely used in optimization are control flow graphs (CFG), data flow graphs (DFG), and program dependence graphs (PDG). These graphs are used as intermediate program representations to optimize a program. Another intermediate program representation is static single assignment form (also referred to as SSA form), which is used to optimize programs [Cytron89].

Program dependence web (PDW) is a relatively recent form of intermediate program representation. Program dependence web was introduced by Maccabe, Ballance, and Ottenstien [Ballance90]. Program dependence web, which combines the ideas of SSA form and PDG, can be directly interpreted using three models of execution. The three models are control-, data-, and demand-driven models of execution. A PDW could be

used to construct the compositional semantics of a program and to analyze the performance of existing programs. It is also used in cross-compilation and provides an insight into the algorithm-to-architecture mapping problem for parallel architectures using different execution disciplines. A PDW incorporates both the SSA form and the control flow properties, which makes it a more powerful representation than a PDG.

The origin of flow graphs can be traced to flow charts. Much research has been done in the area of flow graphs [Aho73] [Hecht77] [Dennis80] [Ferrante83] [Ferrante87] [Cytron89] [Ballance90] [Cytron91] [Pingali91].

Most representations in the general area of data flow identification and analysis treat all dependencies as data dependencies, control dependencies being converted to data dependencies when necessary [Ottenstein84]. Constructions of data dependence subgraphs have also received much attention [Ottenstein78] [Ottenstein81]. DFGs represent the global data dependence at the statement level. Control dependence can be defined in terms of control flow graphs. The transformations that involve both control and data dependencies could be specified in PDGs [Ferrante87].

Lowry, Cytron, and Zadeck, in their discussion of code motion [Lowry86], came up with the concept of associating a unique variable name v_i to a variable v, each time it is referenced. An efficient method of representing data flow and control flow properties of programs was proposed by Cytron [Cytron89]. This method involves the translation of simple FORTRAN programs into CDG, which is then translated to an SSA form.

An extension of a PDG, which combines the SSA form of a program with explicit operators that manage the flow of data values, is program dependence web [Ballance90].

Translation of a program to a PDW involves three steps [Ballance90]. The first step is to convert the source program to an SSA form. This form serves as an input to

the second stage, where it gets translated to a gated single assignment form (GSA form). Finally, the GSA form is converted to a PDW.

A tool was developed which translates simple C programs to PDWs. This tool uses a new algorithm that differs to a certain extent from the translation process discussed earlier. The main difference is that CFGs, instead of CDGs, form the basis of the translation.

This thesis discusses the translation process and the implementation details of the tool. The remainder of this report is organized as follows. Chapter II introduces the definitions of intermediate representations that are necessary for the translation of a source program to a PDW. This chapter also introduces the basics of PDWs. Chapter III describes the translation process used in the development of this tool, and the difference between the algorithm used and the algorithms that have been suggested in the literature [Ballance90]. Chapter IV discusses the implementation details and Chapter V summarizes the research work and elaborates on the future work.

CHAPTER II

INTERMEDIATE REPRESENTATIONS

The various intermediate representations of a program, used in the literature for different purposes, are defined in this chapter. Control flow graphs (CFGs) and data flow graphs (DFGs) are used in compilers for optimization. Data dependence graphs (DDGs) can be used to measure data dependency complexity. Other intermediate representations, such as static single assignment form (SSA form), gated single assignment form (GSA form), and program dependence web (PDW) are also defined in this chapter. CFGs and DDGs play an important role in the translation of programs to program dependence webs (PDWs). This chapter also discusses their abstract representations and their use.

2.1 Control Flow Graph

A CFG is a directed graph G with the nodes representing the basic blocks (a sequence of instructions with no branches) of a program. A CFG has two additional nodes, called the START and the STOP, which form the entry and exit of a CFG, respectively. An assumption is made that, for any node N in G, there exists a path from START to N and from N to STOP.

A CFG is also defined as a two-dimensional representation of a program that displays the flow of control of a program [Aho73]. Formally, the CFG of a program is a 4-tuple, F = (N, E, a, z), where a is the START node whose indegree is zero and z is

the STOP node whose outdegree is zero [Regson93]. Each node N represents a basic block and each edge in E represents a possible block to block transfer.

In the example shown in Figure 1. a program (1.a) and its equivalent control flow graph in 1.b are given.

```
scanf("%d %d",&x,&y);                    /*  1  x1  */
if  ( y  >=  x ) {                       /*  2  Predicate P */
      if  ( y  ==  0 )  then             /*  3  Predicate Q */
            x = 0                        /*  4  x2  */
            }
else
      x =  -1                            /*  5  x3  */

. . . . . .  Use  x  . . . . .           /*  6   */
```

Figure 1.a A sample C program

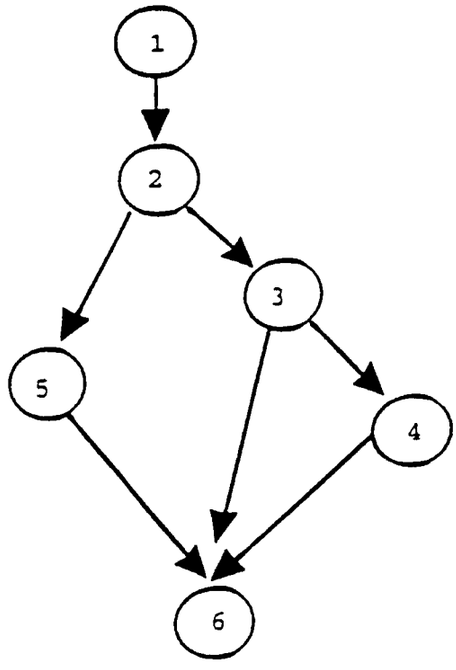

Figure 1.b Control flow graph of the program given in Figure 1.a

2.2 Control Dependence Graph

A CDG formalizes the notion that the execution of one node in the graph may conditionally depend upon the execution of other nodes in the graph. A few preliminary definitions are given below before introducing the definition of a CDG.

The following definitions are mostly based on [Ballance92]. Let X and Y be nodes in a CFG. If Y appears on every path from X to STOP, then Y *postdominates* X. If Y postdominates X and $X \neq Y$, then Y is the closest postdominator of X.

A CFG node Y is *immediately control dependent* on a CFG node X if both of the following hold:

1. There is a non-null path $p: X \Rightarrow^{+} Y$ such that Y postdominates every node after X on p.
2. The node Y does not strictly postdominate the node X.

A CFG node Y is *control dependent* on a CFG node X, if either Y is immediately

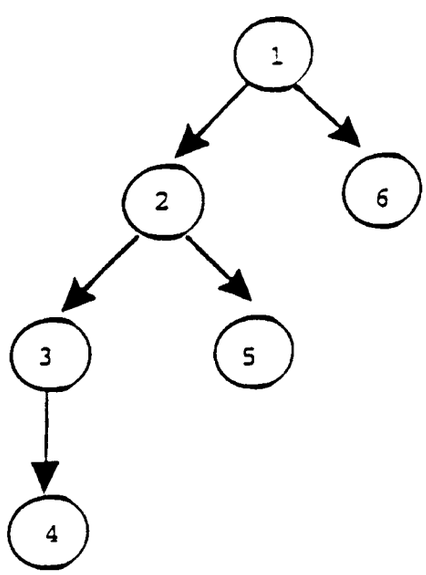

Figure 2. Control dependence graph of the program given in Figure 1.a

control dependent upon X or if there is a node Z, such that Y is immediately control dependent upon Z, and Z is control dependent upon X. An example of a CDG is shown in Figure 2 for the sample program given in Figure 1.a.

2.3 Data Dependence Graph

A DDG represents global data dependence at the statement level where the value of a variable may be referenced or modified. A DDG is a directed graph in which nodes represent statements where variables are modified and edges represent possible data dependencies [Regson93].

The two types of data dependencies are *flow-order* and *def-order*. Let X and Y be nodes in the CFG of a program. Let variable v be defined in node X and used in node

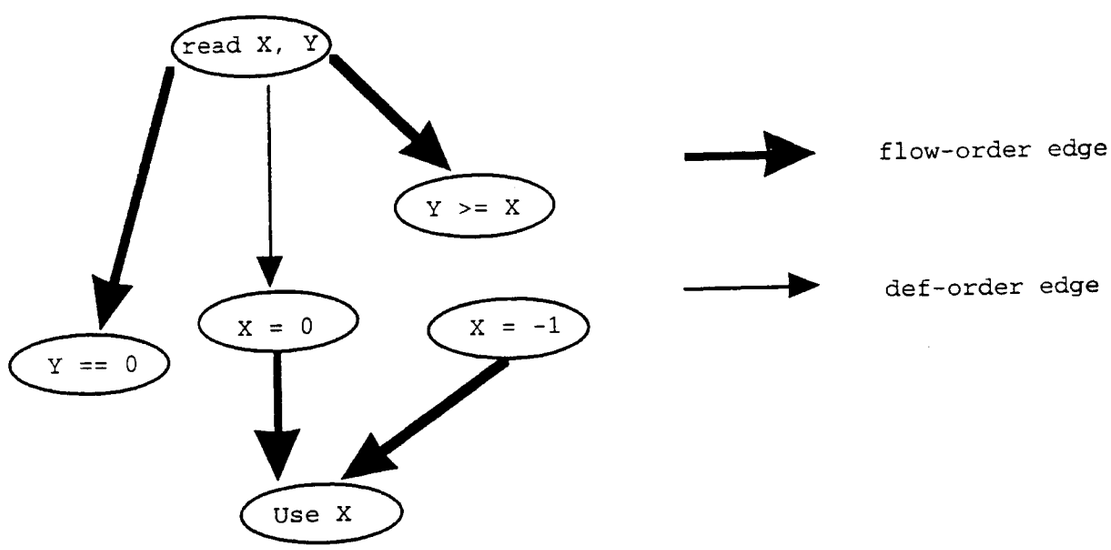

Figure 3. Data dependence graph of the program given in Figure 1.a

Y. There is a flow-order dependence edge from node X to node Y, if there exists a path in the corresponding CFG from X to Y. A set of conditions need to hold for a def-order

dependence edge to exist between nodes X and Y: a. both X and Y must define the same variable, b. X and Y must be on the same path in the corresponding CFG, c. another node Z exists in the corresponding CFG such that there exists a flow-order dependence between X and Z, and between Y and Z, and d. X occurs to the left of the abstract syntax tree of the program. An example of a DDG is given in Figure 3 for the sample program listed in Figure 1.a. The bold faced arrows indicate flow-order edges and the solid arrows represent def-order arrows in Figure 3 [Regson93].

2.4 Program Dependence Graph

A PDG is a graph of a program in which the statements and the predicate

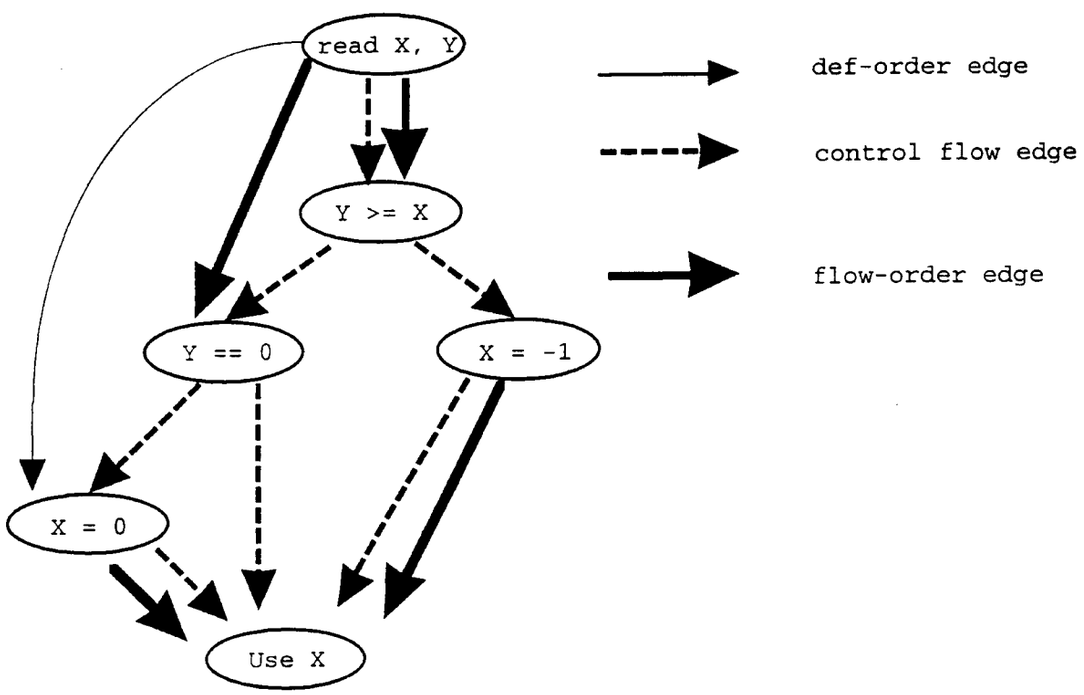

Figure 4. Program dependence graph of the program given in Figure 1.a

expressions are nodes, and the edges incident to the nodes represent both the data values

on which the nodes' operations depend and the control conditions on which the execution of the operations depend [Ferrante87].

A PDG contains both the control and data dependence information embedded in it using directed edges connecting its nodes. Though a PDG unifies control dependence and data dependence, techniques for constructing a PDG rely on the presence of a CFG. In the construction of a PDW also, as it could be seen from the literature, much depends on the construction of the corresponding CFG.

An example of a PDG is shown in Figure 4 for the sample program given in Figure 1.a. The bold faced arrows indicate the flow-order dependence edges and the thin arrows represent the def-order edges. The dashed arrows indicate the flow of control in a program.

2.5 Static Single Assignment Form

The SSA form is an intermediate program representation that is used to represent data and control flow properties. In an SSA form, each occurrence of a variable is replaced by a new variable name. A function (Φ-function) is introduced at appropriate positions to indicate that the variable may have more than one value at the corresponding positions [Cytron89].

A *Φ-function* has the form $u \leftarrow \Phi(v, w, ...)$, where u, v, w, ... are instances of the same variable and the operands v, w, ... are the control flow predecessors of the point where a Φ-function appears.

```
1  scanf("%d",&v);
2  scanf("%d",&y);
3  if (y == 5)
4       v = 10;
5  t = v+y;
```

For example, an equivalent SSA form of the above program is written as follows:

```
scanf("%d",&v1);
scanf("%d",&y1);
if (y1 == 5)
      v2 = 10;
v3 = Φ(v1,v2);
```

At statement number 5, variable v can have the value of v at statement number 1 or the value at statement number 4. Variable v at statement number 1 is expressed as v_1 and variable v at statement number 4 is denoted as v_2. A Φ-function, $v_3 = \Phi(v_1, v_2)$ can be introduced before statement number 5 indicating that variable v at that instant could take the value of either v_1 or v_2.

The *SSA form* for the CFG of a program is defined as follows. For every variable v in the source program, Φ-functions for v are inserted and each mention of v is changed to a mention of a new name v_i, such that the following conditions hold.

1. At every CFG node Z, where two non-null paths converge and in each of those paths a variable v has been assigned a value, a Φ-function for v is inserted immediately before node Z.

2. Each new name v_i for v is the target of exactly one assignment statement or a statement in which the value of v may be changed after the execution of that statement in the program text.

```
scanf("%d %d",&x##1,&y##1);                    /* x1  */
if  ( y##1  >=  x##1 ) {                       /* Predicate P */
      if  ( y##1 == 0 )  then                  /* Predicate Q */
            x##2 = 0                           /* x2  */
            }
else
      x##3 =  -1                               /* x3  */
x##4 = Φ(x##2, x##3)
. . . . . .   Use  x##4   . . . . . .
```

Figure 5. Static single assignment form of the program listed in Figure 1.a

3. The value of the variable name v_i should be the same as v along any control flow path. An example of SSA form is shown in Figure 5 for the sample program given in Figure 1.a

2.6 Gated Single Assignment Form

The GSA form, like the SSA form, depicts the value(s) reaching a variable. The GSA form also depicts the path taken by a value to reach a variable which is used [Ottenstein89].

A γ-*function*, which is used to for a conditional statement, is defined with three variables as γ (P, v^{true}, v^{false}), where P is a predicate, v^{true} is the definition for 'true', and v^{false} is the definition for 'false'. For example, a predicate P can be the *condition* in the statement **If** *condition* **then** A **else** B, v^{true} can be A, and v^{false} can be B.

A μ-*function*, which is used to depict loops, is defined as μ(P, v^{init}, v^{iter}), where the predicate P determines if the control passes into the loop, v^{init} represents those values that enter the head of the loop, and v^{iter} is returned on all executions by the loop.

A η-*function*, which is used to pass the values of variables outside the loops, is defined as η^T or η^F. An η^T(P, v) function returns the value v when the predicate P is true and consumes v otherwise. An η^F(P, v) function returns the value v when the predicate P is false and consumes v otherwise.

The GSA form is defined using gating functions instead of Φ-functions as in the SSA form [Ottenstien89]. Gating functions capture the control conditions that determine which of the definitions reaching a given Φ-function will provide the value for that function. There are three types of gating functions: γ-functions that control forward flow, μ-functions that control the mixing of "loop carried" flow with loop initialization

flows, and η-functions that control the passage of values out of loop bodies into the computations following the loops.

An example of a GSA form is shown in Figure 6 for the sample program given in Figure 1.a.

```
scanf("%d %d",&x##1,&Y##1);                    /* x1  */
if  ( y##1  >=  x##1 ) {                       /* Predicate P */
      if  ( y##1 == 0 )  then                  /* Predicate Q */
            x##2 = 0                           /* x2  */
      }
else
      x##3 =  -1                               /* x3  */
x##4 = γ(y##1 >= x##1, x##2, x##3)
. . . . . .   Use  x##4   . . . . . .
```

Figure 6. Gated single assignment form of the program given in Figure 1.a

2.7 Program Dependence Web

The PDW is an intermediate program representation that is used to interpret the three models of execution namely the control-, data-, or demand-driven models of execution [Ballance90]. It is also used in the construction of compositional semantics and to map existing programs to dataflow architectures [Ballance90] [Campbell93]. It combines the concept of static single assignment version along with PDG to explicitly depict the flow of control and data in a program. The PDW is also called as augmented PDG [Campbell93] [Ballance90].

The PDW is a directed graph with nodes representing the gating functions, and the data and the arcs representing the flow of control and data. There are switches which control the flow of values into a region. The PDW is also defined as a GSA form with the gating functions controlling the flow of data and control through and out of a loop.

The PDW also has switches that transmit data out of the loops.

A *switch* in a PDW is a binary function S(*p, v*) with two output ports denoted by S^T and S^F. The value *v* is transmitted to either one of S^T or S^F according to the truth value of the predicate *p*.

More formally, the PDW is defined [Campbell93] as a directed graph G = (V,A), where

V = {START, STOP. Operators, Predicates, Read/Write, Switches, γ, μ, η}, and

A = {control dependence, data dependence, other data dependencies}.

An example of a PDW is shown in Figure 6 for the sample program listed in Figure 1.a.

```
scanf("%d %d",&x##1,&y##1);                    /*  x1  */
if  ( y##1  >=  x##1 ) {                       /*  Predicate P */
      Predicate P is TRUE
      if  ( y##1 == 0 )  then                  /* Predicate Q */
            Predicate Q is TRUE
            x##2 = 0                           /*  x2  */
      }
else
      Predicate P is FALSE
      x##3 =  -1                               /*  x3  */
x##4 = γ(y##1 >= x##1, x##2, x##3)
. . . . . .   Use  x##4   . . . . . .
```

Figure 7. Program dependence web of the program given in Figure 1.a

CHAPTER III

TRANSLATION TO PDW

According to Maccabe, Ballance, and Ottenstien [Ballance90], three steps are involved in the translation of a simple C program to PDW as follows.

a. Conversion of a source program to the SSA form.

b. Translation of the SSA form to the GSA form.

c. Conversion of the GSA form to the PDW.

During the implementation of the translation process for this thesis, it was discovered that only two steps were necessary for the translation. However, for better understanding, the translation process is discussed in first three sections of this chapter. The two-step translation process is discussed in Section 3.4.

3.1 Source Program to SSA Form

The edges of a program dependence graph consist of control dependence and data dependence edges [Cytron90]. A control flow graph is first drawn for a given source program. To draw a control flow graph, the basic blocks must be identified. Using the CFG, a CDG is drawn. The CFG is used to compute the dominance frontier for each node X, (which is a node in the CDG) using the algorithm presented by Cytron [Cytron89]. According to Cytron [Cytron89], two steps are necessary to translate to the SSA form: a. each variable v is given several new names v_i, and b. special assignment

statements called Φ-functions are inserted at certain points in the program. Using the dominance frontier. Φ-functions for the variables are introduced in the program.

An alternative algorithm to the one proposed by Cytron [Cytron89] is to introduce the Φ-functions directly into the CFG without computing the CDG. Though the CDG may have its advantages [Ballance92] in generating the SSA form, the CFG has an added advantage in the translation of the SSA form to the PDW. The data structure of a node in a CFG plays an important role in the translation of the SSA form to the PDW.

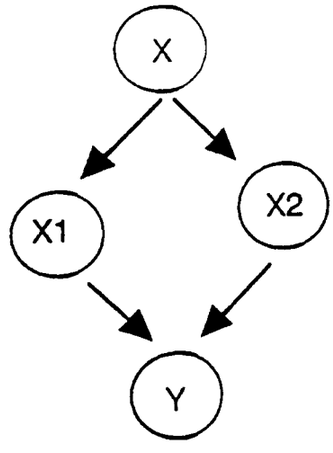

Figure 8.a Graph with a node having indegree greater than one

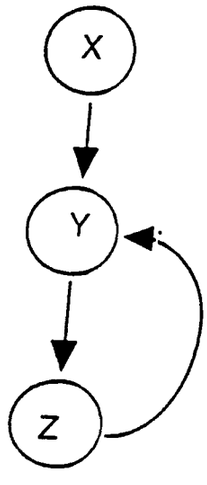

Figure 8.b Graph with a node in a loop having indegree greater than one

The algorithm works as follows. The basic blocks of a program are identified and the CFG is drawn. The Φ-functions are introduced at nodes with indegrees greater than one. To illustrate this, consider Figure 8.a where node Y has indegree greater than one. This means that node Y could be reached by more than one path. In Figure 8.a, where more than one path exists from X to Y, let a variable v be assigned a value along the path X, X1, and Y, and another value be assigned to v along the path X, X2, and Y. At node Y, a variable v may have its value assigned along the path X, X1, and Y or X, X2, and Y, depending on the flow of control at X. So a Φ-function is to be inserted at Y for variable v.

Another possibility, shown in Figure 8.b, is a loop. In Figure 8.b, there exists a path from X to Y, Y to Z, and a path from Z to Y. Therefore, the principle is that the values of one or more variables, $v_1, v_2, \ldots, v_n$, reaching node Y, with indegree greater than one, depend on the path taken to reach node Y. Hence, Φ-functions are introduced at nodes with indegrees greater than one.

```
#include <stdio.h>
main()
{
int i,j,k,l;
int h1,h2,h3,h4,h5;
scanf("%d %d %d %d",&i,&j,&k,&l);
while(i<10) {
      if (k == 1) {
            j = i;
            if (k == j) l = 5;
            else
                  l=6;
            }
      else k = k+2;
      while(k < 10) {
            if (k == i) l = l+4;
            }
      i = i+3;
      }
}
```

Figure 9. A sample C program

```
#include <stdio.h>
main()
{
int i,j,k,l;
int h1,h2,h3,h4,h5;
scanf("%d %d %d %d",&i1,&j1,&k1,&l1);
 i2 = Φ(i1, i3)
 j2 = Φ(j1, j4)
 l2 = Φ(l1, l6)
 k2 = Φ(k1, k3)
while(i2<10) {
      if (k2 == l2) {
            j3 = i2;
            if (k2 == j3) l3 = 5;
            else l9=6;
            l4 =  Φ(l3, l9)
            }
      else k4 = k2+2;
       j4 = Φ(j3, j2)
       k3 = Φ(k2, k4)
       l5 = Φ(l4, l2)
       l6 = Φ(l5, l8)
      while(k3 < 10) {
            if (k3 == i2) l7 = l6+4;
             l8 = Φ(l6, l7)
            }
      i3 = i2+3;
      }
}
```

Figure 10. SSA form of the program given in Figure 9

During two traversals of the flow graph of a program, the information such as start of loops, end of loops, if-then statements, assignment statements, running variables in loops, and predicates are identified. For the sample program given in Figure 9, an equivalent SSA form is shown in Figure 10.

3.2 SSA Form to GSA Form

The input to this phase is the SSA form generated by using the algorithm mentioned in Section 3.1. This step involves the substitution of the Φ-functions by gating functions. Each one of the Φ-functions is replaced by one of the three gating functions: the μ-function, the γ-function, or the η-function.

```
#include <stdio.h>
main()
{
int i,j,k,l;
int h1,h2,h3,h4,h5;

scanf("%d %d %d %d",&i1,&j1,&k1,&l1);
i2 = μ(i2<10,i1, i3)
j2 = μ(i2<10, j1, j4)
l2 = μ(i2<10, l1, l6)
k2 = μ(i2<10, k1, k3)
while(i2<10) {
      if (k2 == l2) {
            j3 = i2;
            if (k2 == j3) l3 = 5;
            else l9=6;
             l4 =  γ(k2 == j3, l3, l9)
            }
      else k4 = k2+2;
      j4 = γ(k2 == l2, j3, j2)
      k3 = γ(k2 == l2, k2, k4)
      l5 = γ(k2 == l2, l4, l2)
      l6 = μ(k3 < 10, l5, l8)
      while(k3 < 10) {
            if (k3 == i2) l7 = l6+4;
             l8 = γ(k3 == i2, l6, l7)
            }
      i3 = i2+3;
      i2 = η^T(i2<10, i3)
      j2 = η^T(i2<10, j4)
      k2 = η^T(i2<10, k3)
      l2 = η^T(i2<10, l6)
      }
}
```

Figure 11. GSA form of the program given in Figure 9

Using the information collected as a result of the two traversals through the control flow graph in the first step, and traversing through the SSA form, the Φ-functions are replaced by the gating functions. The GSA form for the program given in Figure 9 is shown in Figure 11.

3.3 GSA Form to PDW

The last step in the translation of a program to the PDW is the introduction of switches. The algorithm used to translate a program to the PDW uses the information collected in the translation of a program to the SSA form, to introduce switches. An

equivalent PDW for the sample program given in Figure 9 is given in Figure 12.

```
#include <stdio.h>
main()
{
int i,j,k,l;
int h1,h2,h3,h4,h5;

scanf("%d %d %d %d",&i1,&j1,&k1,&l1);
i2 = μ(i2<10,i1, i3)
j2 = μ(i2<10, j1, j4)
l2 = μ(i2<10, l1, l6)
k2 = μ(i2<10, k1, k3)
while(i2<10) {                          /* Predicate P */
      if (k2 == l2) {                   /* Predicate Q */
            O is TRUE
            j3 = i2;
            if (k2 == j3)               /* Predicate R */
                  R is TRUE
                  l3 = 5;
            else
                  R is FALSE
                  l9=6;
             l4 =  γ(k2 == j3, l3, l9)
            }
      else
            Q is TRUE
            k4 = k2+2;
      j4 = γ(k2 == l2, j3, j2)
      k3 = γ(k2 == l2, k2, k4)
      l5 = γ(k2 == l2, l4, l2)
      l6 = μ(k3 < 10, l5, l8)
      while(k3 < 10) {                  /* Predicate S */
            if (k3 == i2)               /* Predicate T */
                  T is TRUE
                  l7 = l6+4;
             l8 = γ(k3 == i2, l6, l7)
            }
      i3 = i2+3;
      i2 = η^T(i2<10, i3)
      j2 = η^T(i2<10, j4)
      k2 = η^T(i2<10, k3)
      l2 = η^T(i2<10, l6)
      }
}
```

Figure 12. PDW of the program given in Figure 9

3.4 Two-Step Translation Process

The first step is the translation of a simple C program to the SSA form was explained in Section 3.1. The second step is the introduction of gating functions and

switches based on the information collected in the first step. The problem rests in the collection of information. In other words, with the proper node structure, the translation of the PDW can be a two-step translation process. The disadvantage is that as the input program increases by size, the amount of memory required to translate also increases depending on the number of nodes with indegree greater than one. The simplicity of the algorithm is its only advantage.

CHAPTER IV

IMPLEMENTATION DETAILS

A tool to translate a simple C program (expressed by the grammar in Section 4.1) to the PDW was developed. The grammar of the input C programs, the node structure that was used, the algorithms used in the translation, the complexity of the algorithms, and the implementation platform are the issues that are discussed in this chapter.

4.1 Grammar for the Input Program

The grammar for the simple input C programs is given below.

```
<program>              ::=   <statement>
<statement>            ::=   <statement> <statement>
<statement>            ::=   if <predicate>
                             then <statement>
                             else <statement>
<statement>            ::=   while <predicate>
                             do <statement>
<statement>            ::=   for(initial,predicate,incremental value)
                             <statement>
<statement>            ::=   switch(<variable>)
                             <case statement>
<statement>            ::=   <input statement>
<statement>            ::=   <output statement>
<case statement>       ::=   case(value):      <statement>
                                               break;
                             <case statement>
<case statement>       ::=   default: <statement>
<statement>            ::=   <variable> ← <expression>
```

The C programs that are input to the package are restricted to the above grammar. An operational definition exists to convert a FORTRAN program expressed as in a subset of the grammar given above to the PDW.

4.2 Node Structure of the Flow Graph

As an input C program is parsed, the CFG is drawn. The node structure of the flow graph is defined as follows.

```
/* The following structure is used to build a directed graph depicting
   CFG, SSA form, GSA form, and PDW  */

    typedef struct cfg_n {
            int flag;
            int flag_end;
            int st_no;
            int no_in_edges;      /* Number of incoming edges */
            int no_of_tr;         /* This is used in building the
                                     SSA form  */
            char *stmt;           /* Statement */
            int what_stmt;        /* Indicates the type of statement */
            int start_stmt;       /* The first statement in a loop
                                     or the if statement */
            struct cfg_n *left, *right;
                                  /* The pointer to the next node */
                                  /* If this node is an if_statement
                                     then the left and right pointers indicate
                                     true and false, respectively */
            struct ssa_ins_list *f_list,*l_list;
                                  /* The ssa-ins_list gives a linked list
                                     of variables for which Phi
                                     variables are to be inserted  */
            struct phi_var_list *f_phi,*l_phi;
                                  /* This list is used in the building
                                     of the SSA form during the first
                                     two traversals */
            struct ssa_var *sec_list;
                                  /* This list is used in the building
                                     of the GSA form */
            struct int_list *f_int;
    }cfg_n;
```

The CFG is represented by a binary-tree-structured directed graph with the left pointer indicating that the control flows towards the left and a right pointer indicating that the control flows towards the right. In the case of an *if condition then else statement*, the

left pointer indicates the flow of control in the event of the condition being true and the right pointer indicates the flow of control in the event of the condition being false. When the flow of control is linear, the right pointer is made NULL and the left pointer points to the next node or the next statement.

A recursive algorithm was used to traverse the directed tree-structured graph. The different fields of the data structure used to implement a node in the flow graph were defined as follows. There are two flag variables *flag* and *flag_end*, one used to check whether the node has been visited and the other to keep track of the phi variables that have been introduced in the first traversal of the CFG (as explained in the algorithm described in Section 4.3). Each statement is stored using a character pointer and the statement number is defined by an integer variable. As defined in Section 2.5, the SSA form introduces a unique variable name for each occurrence of a variable in the program. It also introduces phi functions for the variables. The tool defines the unique variable names in the SSA form as variable##number instead of $variable_{number}$ as used in the literature [see Appendix D for an example and description]. Two types of linked lists were used to maintain the status of variables at each node. One linked list named ssa_var contains the variable name and the variable number. Another linked list contains the variable name, the last referenced variable name, when the variable name was read, and the last referenced variable name, when the variable was assigned a value. Two other integer variables maintain the number of incoming edges and the number of edges along which the node has been reached during a traversal.

The above node structure plays an important role during the translation process of the second and third steps.

4.3 Algorithms

Given the proper node structure, it is possible to convert a program to the SSA form by using a recursive algorithm. The algorithm used traverses the CFG twice with phi functions being introduced in both traversals. The terminating condition in the recursive traversal algorithms differs as it could be seen in Algorithms 1 and 2 given below.

Algorithm 1:

```
Traverse_graph(current_node)
{
if (current_node == NULL) return;
If (current_node->flag is Visited) return;
Traverse_graph(current_node->left_ptr);
current_node->flag = Visited;
Traverse_graph(current_node->right_ptr);
}
```

Algorithm 2:

```
Traverse_g(current_node)
{
if (current_node == NULL) return;
if (current_node->flag is Visited) return;
Traverse_g(current_node->left_ptr);
Traverse_g(current_node->right_ptr);
current_node->flag = Visited;
}
```

The algorithms that enable the introduction of phi functions are in Algorithms 3 and 4 given below.

Algorithm 3:

```
intro_phi_func_1(node, linked_list_var)
{
if (node == NULL) return;
if (node->flag is Visited) return;
update_linked_list_var(node, linked_list_var);
if (node->left_ptr != NULL) {
      Update the dependencies that exist between the current node
      and the node accessed by the left pointer
      If (indegree of the left node is greater than one) {
```

```
                /*  This node is a prospective node for the
                introduction of Phi function(s) */
                if  (there does not exist the status of the variable)
                      The status of the variables is stored
                else
                      The status of the variable is updated.
                      Introduce Phi functions for updated variables
                }
        }
copy the linked list var to new_linked list_var;
first_pass(node->left_ptr, new_linked_list_var);
node->flag = Visited;
if (node->right_ptr != NULL) {
      Update the dependencies that exist between the current node
      and the node accessed by the right pointer
      If (indegree of the right node is greater than one) {
            /*  This node is a prospective node for the
            introduction of Phi function(s) */
            if  (there does not exist the status of the variable)
                  The status of the variables is stored
            else
                  The status of the variable is updated.
                  Introduce Phi functions for updated variables
            }
      }
copy the linked list var to new_linked list_var;
first_pass(node->right_ptr, new_linked_list_var);
}
```

```
#include <stdio.h>
main()
{
int i,j,k,l;
int h1,h2,h3,h4,h5;
scanf("%d %d %d %d",&i1,&j1,&k1,&l1);
 i2 = Φ(i1, i3)
 j2 = Φ(j1, j4)
 l2 = Φ(l1, l6)
 k2 = Φ(k1, k3)
while(i2<10) {
      if (k2 == l2) {
            j3 = i2;
            if (k2 == j3) l3 = 5;
            else l9=6;
            l4 =  Φ(l3, l9)
            }
      else k4 = k2+2;
       j4 = Φ(j3, j2)
       k3 = Φ(k2, k4)
       l6 = Φ(l5, l8)
      while(k3 < 10) {
            if (k3 == i2) l7 = l6+4;
             l8 = Φ(l6, l7)
            }
      i3 = i2+3;
      }
}
```

Figure 13. SSA form of the program given in Figure 9 after execution of Algorithm 3

Figure 13 depicts the output of Algorithm 3 for the sample program given in Figure 9. An equivalent SSA form for the program given in Figure 9 is shown in Figure 10. As it could be noted, the phi function, $l_5 = \Phi(l_4, l_2)$, in Figure 10 does not exist in Figure 13. This is because the new variable names assigned to the phi function is not propagated to other nodes. Algorithm 4 traverses the CFG, which has phi functions partially introduced into it, and propagates the variables for which the phi functions were introduced by Algorithm 3.

Algorithm 4

```
gen_ssa_form(node, linked_list_stat)
{
if (node == NULL) return;
if (node is Visited) return;
update_phi_var(node, linked_list_var);
update_linked_list_var(node, linked_list_var);

if (node->left_ptr != NULL) {
      Update the dependencies that exist between the current node
      and the node accessed by the left pointer
      If (indegree of the left node is greater than one) {
            /*  This node is a prospective node for the
            introduction of Phi function(s) */

            if  (there does not exist the status of the variable)
                  The status of the variables is stored
            else
                  The status of the variable is updated.
                  Introduce Phi functions for updated variables.
            }
      }
copy the linked list var to new_linked list_var;
gen_ssa_form(node->left_ptr, new_linked_list_var);

if (node->right_ptr != NULL) {
            Update the dependencies that exist between the current
            node and the node accessed by the right pointer
            If (indegree of the right node is greater than one) {
            /*  This node is a prospective node for the
            introduction of Phi function(s) */

            if  (there does not exist the status of the variable)
                  The status of the variables is stored
            else
                  The status of the variable is updated.
                  Introduce Phi functions for updated variables

            }
      }
copy the linked list var to new_linked list_var;
```

```
gen_ssa_form(node->left_ptr, new_linked_list_var);
node->flag = Visited;
}
```

The two functions used in algorithms 3 and 4 are defined below.

```
update_phi_var(node, linked_list_var)
{
For each variable with a unique identification number appearing in
the phi variable list of each node
     Update  the number in the linked_list_var corresponding to
that variable.
}

update_linked_list_var(node, linked_list_var)
{
For each variable with a unique identification number appearing in
the statement in node
     Update the number in the linked_list_var corresponding to
that variable.
}
```

4.4 Complexity

The complexity of translating simple C programs to the PDW can be discussed in terms of execution time and memory usage.

The execution complexity can be defined as the number of traversals of a control flow graph. The maximum number of traversals needed for the translation of simple C programs to the PDW is four. As the size of the input program increases, the time needed for the translation generally increases because of the increase in the size of the corresponding flow graph.

Space complexity in terms of storage usage is defined as the amount of memory necessary for the translation. This depends on the number of nodes with indegrees greater than one, the number of variables in the program, and the number of Φ-functions introduced at each node. The amount of memory storage increases linearly with the number of nodes with indegrees greater than one.

4.5 Implementation Platform and Environment

4.5.1 Lex and Yacc

Lex and Yacc are tools that are used to create C routines to analyze and interpret an input stream [Brown90]. Lex reads a specification file containing regular expressions for pattern matching, and generates a C routine that performs lexical analysis (i.e., identifies streams of characters and generates matching sequences called tokens). The lexical analyzer reads the input and produces a sequence of tokens to be used by the parser.

Yacc reads a grammatical specification file of a language and generates a parsing routine. This routine groups tokens into meaningful sequences and invokes action routines to act upon them. Yacc is a tool that generates C code to parse the input. The C code is then compiled for an executable file. Yacc repeatedly calls the lexical analyzer for the tokens, recognizes the input using the tokens, and performs the code associated with each rule recognizing the token. There are three important specifications needed in using Lex and Yacc to develop a PDW:

a. a lexical analyzer is needed to scan the input and break it into meaningful chunks (i.e., tokens) for the parser,

b. a grammar that specifies the syntax of the input languages, and

c. code associated with each rules or actions associated with the grammar.

4.5.2 Sequent Symmetry S/81

The Symmetry S/81 is a powerful mainframe-class multiprocessor system developed by Sequent Computer System, Inc. [Sequent90]. Sequent S/81 is a shared

memory, tightly-coupled multiprocessor that runs the DYNIX/ptx operating system. It also has hardware supporting mutual exclusion. The load is balanced and the tasks are distributed in a multi-user environment to increase throughput and response time. UNIX compatible software can run on the Symmetry S/81 without modification or with slight modification.

CHAPTER V

SUMMARY AND FUTURE WORK

5.1 Summary

The main purpose of this thesis was to implement the ideas proposed by Maccabe, Ballance and Ottenstien [Ballance 90]. A tool was developed that converts simple C programs to the PDW. Instead of using CDGs as an intermediate step to develop the SSA form, CFGs were used to develop the SSA form. Various algorithms were investigated and a feasible algorithm was developed. The complexities of the algorithms in terms of the number of passes over a program and the storage required were also analyzed.

The major contributions of this thesis are listed below.

a. Implementation of the proposed intermediate representation, the PDW.

b. Development of a new algorithm to translate simple C programs to the PDW.

c. Use of the CFG in the translation process.

5.2 Future Work

Possible future work includes extending the tool for any input C program inclusive of pointers, structures, procedures, and recursion. The tool could be changed to incorporate the redefined PDW proposed recently [Campbell93]. Another area of future

work would be to identify and to use the PDW for various applications. Finally, the tool could be extended using X-windows depicting the PDW pictorially instead of as a binary-tree structure.

REFERENCES

[Aho73] Alfred V. Aho and Jeffrey D. Ullman, *The Theory of Parsing, Translation and Compiling, Volume 1: Parsing*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1973.

[Ballance90] Robert A. Ballance, Arthur B. Maccabe, and Karl J. Ottenstein, "The Program Dependence Web: A Representation Supporting Control-, Data-, and Demand-Driven Interpretation of Imperative Languages", *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, pp. 257-271, White Plains, NY, June 1990.

[Ballance 92] Robert A. Ballance and Arthur B. Maccabe, "Program Dependence Graphs for the Rest of Us", Technical Report 92-10, Department of Computer Science, The University of New Mexico, Albuquerque, NM, August 1992.

[Brown90] Doug Brown and Tony Mason, *Lex and Yacc*, O'Reilly & Associates, Inc., Sebastopol, CA, May 1990.

[Campbell93] Philip L. Campbell, Kesheerabdhi Krishna, and Robert A. Ballance, "Refining and Defining the Program Dependence Web", Technical Report 93-6, Department of Computer Science, The University of New Mexico, Albuquerque, NM, March 1993.

[Cytron89] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck, "An Efficient Method of Computing Static Single Assignment Form", *Conference Record of the 16th Annual ACM Symposium on Principles of Programming Languages*, pp. 25-35, Austin, TX, January 1989.

[Cytron90] Ron Cytron, Jeanne Ferrante, and Vivek Sarkar, "Compact Representations for Control Dependence", *Proceedings of the SIGPLAN'90 Conference on Programming Language Design and Implementation*, pp. 337-351, White Plains, NY, June 1990.

[Cytron91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck, "Efficiently Computing Static Single Assignment Form and the Control Dependence Graph", *ACM Transactions on Programming Languages and Systems*. vol. 13, no. 2, pp. 451-490, October 1991.

[David82] A.L. David and R.M. Keller, "Data Flow Program Graphs", *IEEE Computer*, vol. 15, no.2, pp. 26-41, February 1982.

[Dennis80] J.B. Dennis, "Data Flow Supercomputers", *IEEE Computer*, vol. 13, no. 3, pp. 48-56, November 1980.

[Ferrante83] Jeanne Ferrante and Karl J. Ottenstein, "A Program Form Based on Data Dependency in Predicate Regions", *Conference Record of the Tenth ACM Symposium on Principles of Programming Languages*, pp. 217-236, Austin, TX, January 1983.

[Ferrante87] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren, "The Program Dependence Graph and Its Use in Optimization", *ACM Transactions on Programming Languages and Systems*, vol. 9, no. 3, pp. 319-347, July 1987.

[Hecht77] Matthew S. Hecht, *Flow Analysis of Computer Programs*, Elsevier North-Holland, Inc., 1977.

[Johnson93] Richard Johnson and Ksehav Pingali, "Dependence-Based Program Analysis", *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation*, pp. 78-89, Albuquerque, NM, June 1993.

[Knuth68] Donald E. Knuth, *The Art of Computer Programming, Volume I: Fundamental Algorithms*, Addison-Wesley Publishing Company, CA, 1968.

[Lowry86] Ron Cytron, Andy Lowry, and Ken Zadeck, "Code Motion of Control Structures in High-Level Languages", *Proceedings of the Thirteenth Annual ACM Symposium on the Principles of Programming Languages*, pp. 70-85, St. Petersburg Beach, FL, January 1986.

[Ottenstein78] Karl J. Ottenstein, "Data-Flow Graphs as an Intermediate Form", Ph.D. Dissertation, Computer Science Department, Purdue University, Lafayette, IN, August 1978.

[Ottenstein81] Karl J. Ottenstein, "An Intermediate form Based on a Cyclic Data Dependence Graph", CS-TR 81-1, Department of Computer Science, Michigan Technological University, Houghton, MI, October 1981.

[Ottenstein84] K.J. Ottenstein and L.M. Ottenstein, "The Program Dependence Graph in a Software Development Environment", *ACM SIGPLAN Notices*, vol. 19, no. 5, pp. 177-184, May 1984.

[Ottenstien89] Karl J. Ottenstein, Robert A. Ballance, and Arthur B. Maccabe, "Gated Single Assignment Form: Dataflow Interpretation for Imperative Languages", LA-UR-89-3654, Los Alamos, NM, 23 pages, October 1989.

[Pingali91] K. Pingali, M. Beck. R. Johnson, M. Moudgill, and P. Stodghill, "Dependence Flow Graphs: An Algebraic Approach to Program Dependencies", In *Advances in Languages and Compilers for Parallel Processing*, MIT press, Cambridge, MA, pp. 445-467, 1991.

[Regson93] C.P. Regson, "Program Flow Graph Decomposition as a Model of Software Comprehension", MS Thesis, Computer Science Department, Oklahoma State University, Stillwater, OK, 1993

[Sequent90] *DYNIX/ptx User's Guide*, Sequent Computer, Inc., 1990.

APPENDICES

APPENDIX A

GLOSSARY AND TRADEMARK INFORMATION

ANSI: American National Standards Institute.

Basic Blocks: Single-entry/Single-exit regions in a program.

CFG: A Control Flow Graph is a directed graph with nodes representing the basic blocks in a program. Two additional nodes, called the START node and the STOP node, form the entry and exit to the CFG. The assumption is that each node is on a path from START and on a path to STOP.

Control Dependence: A control dependence exists between a statement and the predicate whose value immediately controls the execution of the statement.

Data Dependence: A data dependence exists between two statements whenever a variable appearing in one statement may have an incorrect value if the two statements are reversed.

DFG: A Data Flow Graph represents global data dependence at the operator level.

Dominance Frontier: Dominance frontier of a CFG node X is the set of all CFG nodes Y such that X appears on every path from START to Y, and X may be equal to Y. The *Dominance Frontier* DF(X) of a control flow graph node X is the set of all CFG nodes Y such that X dominates the predecessor of Y but does not strictly dominate Y:

$$DF(X) = \{Y \mid (P \in Pred(Y))\ (X >>= P \text{ and } X << >> Y)\}$$

where Pred(Y) is the predecessor set of Y, X >>= P stands for X dominates P, and X << >>Y means X does not strictly dominate Y.

Flowchart: A pictorial representation of the algorithm of a program.

Gating Function: Gating functions are functions that capture the control conditions defining the values reaching a Φ-function (see Section 2.6 for a definition of the Φ-function).

GSA: Gated Single Assignment form is defined by substituting the gating functions by the Φ-functions in an SSA form.

Lex: A lexical specification tool provided on UNIX machines.

PDG: Program Dependence Graph is a directed graph in which the nodes represent assignment statements and control predicates that occur in a program. The edges represent control and data dependencies.

PDW: Program Dependence Web is a directed graph with nodes representing gating functions and switches. The edges represent the flow of control and data through the program.

Region: A subset of the statements in a program.

SSA Form: Static Single Assignment form redefines each occurrence of a variable by a new variable name and introduces a new function at certain positions in a program.

Switch: A switch, which is used for a loop, is a binary function that decides whether the control flows into the loop or out of the loop.

Yacc: Yet Another Compiler Compiler is a parser generator which, using a grammar and the actions associated with the rules of the grammar, generates a set of tables for simple automaton which implements a parsing algorithm to parse the input.

TRADEMARK INFORMATION

APPENDIX B

PROGRAM LISTING

The following are the files that were used in the development of the tool.

lspec - A lexical specification file. This is the input to Lex which generates the lexical analyzer.

yspec.c - This file is the specification for the parser. This file contains the grammar for the input C program that is the input to Yacc which generates the parser.

ssa.h - This file contains procedures that translate a CFG to the corresponding SSA form

gsa.h - This file contains procedures that translate an SSA form to the corresponding GSA form and PDW.

proc.h - This file contains procedures that enable the translation process.

print.h - This file contains procedures that enable the printing of the CFG, SSA form, GSA form, and PDW onto the screen.

The following is the lspec file.

```
dgt [0-9]
alpha [a-z]
%%
[']             return(APOS);
"#include"      return(INCLUDE);
"<stdio.h>"     return(FILENAME);
"<string.h>"    return(FILENAME);
"<math.h>"      return(FILENAME);
"<ctype.h>"     return(FILENAME);
"<conio.h>"     return(FILENAME);
"int"           return(TYPE_DEC);
"float"         return(TYPE_DEC);
"char"          return(TYPE_DEC);
```

```
 "main()"        return(MAIN);
"if"             return(IF);
"then"           return(THEN);
"else"           return(ELSE);
"for"            return(FOR);
"while"          return(WHILE);
"printf"         return(PRINTF);
"scanf"          return(SCANF);
"case"           return(CASE);
"switch"         return(SWITCH);
"default"        return(DEFAULT);
"break"          return(BREAK);
"&"              return(AMPER);
":"              return(CCLON);
"%c"             return(FORMATVAR);
"%d"             return(FORMATVAR);
"%f"             return(FORMATVAR);
"%x"             return(FORMATVAR);
"&&"             return(AND);
"||"             return(OR);
"=="             return(EQ);
">="             return(GEQ);
"<="             return(LEQ);
"<>"             return(NEQ);
">"              return(GT);
"<"              return(LT);
"++"             return(PLUSPLUS);
"--"             return(MINUSMINUS);
"{"              return('{');
"}"              return('}');
"("              return('(');
")"              return(')');
"+"              return('+');
"-"              return('-');
"*"              return('*');
"/"              return('/');
"="              return('=');
"("              return('(');
")"              return(')');
\n                return(ENTER);
";"              return(SEMICOLON);
","              return(COMMA);
{alpha}+({dgt}|{alpha})*     {
                 yylval.sval=yytext;
                 return(VAR);
                 }
{dgt}+           {
                 yylval.sval = yytext;
                 return(NO);
                 }
```

The following is the yspec.c file.

```
%{
#include <stdio.h>
#include <string.h>
#include <malloc.h>
#include <ctype.h>
#define READ_MODE 1
#define WRITE_MODE 2
#define NEWNAME 100
#define OLDNAME 1

        typedef struct phi_var_list {
                char sivar[20];
                char var[20];
                int no;
                struct phi_var_num *f_num, *l_num;
                struct phi_var_list *next;
        } phi_var_list;

        typedef struct int_list {
                int no;
                struct int_list *next;
        } int_list;

        typedef struct phi_var_num {
                int no;
                char svar[20];
                int stmt_no;
                struct phi_var_num *next;
        }phi_var_num;

/*      A linked list is maintained giving the status of
        each variable at that node */


        typedef struct ssa_var_stat{
                int read_no; /* The variable name if it is referenced  */
                int write_no; /* The name if it is assigned a new value*/
                char var[20]; /* Variable Name */
                int stmt_no;  /* Statement Number last read */
                int wrst_no;  /* Statement Number Last Written */
                struct ssa_var_stat *next;
                }ssa_var_stat;

/*  As the CFG is built using a doubly linked list the following linked
    list is used to maintain the statement and other things in a stored
    fashion */

        typedef struct stmt_list {
                int st_no;
                char *stmt;
                int what_stmt;
                int start_stmt;
                struct stmt_list *next;
        }stmt_list;

/* The following structure is a double linked list that is used to build the
   Control Flow Graph. This list is used to build the directed graph which
   is similar to a tree structure used to depict PDW */

        typedef struct cfg {
                int from,to;
                struct cfg *next,*prev;
        }cfg;

        typedef struct ssa_ins_list {
                struct ssa_var *ssa_list;
                struct ssa_ins_list *next;
        }ssa_ins_List;

/* The following structure is used to build a directed graph depicting
    Control Flow Graph, SSA-form, GSA-form and the PDW  */

        typedef struct cfg_n {
                int flag;
                int flag_end;
                int st_no;
                int no_in_edges;        /* Number of incoming edges */
                int no_of_tr;           /* This is used in building the
```

```
                                        SSA-form  */
        char *stmt;                  /* Statement */
        int what_stmt;               /* Indicates the statement
                                        for_loop, assignment .. */
        int start_stmt;                  /* The first statement in a loop
                                        or the if statement */
        struct cfg_n *left, *right;
                                     /* The pointer to the next node */
                                     /* If this node is a if_statement
                                        then left and right pointer indicates
                                        the true and false respectively */
        struct ssa_ins_list *f_list,*l_list;
                                     /* The ssa-ins_list gives a linked list
                                        of variables that for which SIGH
                                        variables are to be inserted  */
        struct phi_var_list *f_phi,*l_phi;
                                     /* This list is used in the building
                                        of the SSA form during the first
                                        two passes */
        struct ssa_var *sec_list;
                                     /* This list is used in the building
                                        of the GSA form */
        struct int_list *f_int;
}cfg_n;


typedef struct var_occcr {
        int st_num;
        struct var_occcr *next;
}var_occcr;

/*      The following typedef struct variable_n is used in the assignment
        of a new variable name to during the parsing of the input C program */

typedef struct variable_n {
        int no;
        int latest;
        char var[20];
        struct var_occcr *occcr_list;
        struct var_occcr *last_occcr;
        struct variable_n *next, *prev;
}variable_n;


typedef struct ssa_var{
        int no;
        char var[20];
        struct ssa_var *next, *prev;
}ssa_var;


typedef struct ssa_intro{
        int no;
        struct var_occcr *f_occcr, *l_occcr;
}ssa_intro;

typedef struct ssa_int_occcr{
        int no;
        struct ssa_int_occcr *next;
}ssa_int_occcr;

int     ret_val;
int     stmt_no = 1;
/*      stmt_no --  Used to keep track of the statements in the
        input  program */

int     case_st;
/*      Keeps track of the statement number */

int     len, control_t_st = 0;
/*      Control transfer to statement Number */
struct          cfg *first_cfg, *cfg_graph, *prev_cfg, *last_cfg;
/*      First node, A current node and the previous CFG in a linked list
*/
struct stmt_list *first_stmt, *stmt_node, *last_stmt, *curr_stmt;
struct          cfg *cfg_switch, *else_cfg;
char    switch_var[15];
int switch_val=-100;
int cre_list = 0;
struct variable_n *var_list,*first_var, *var_tmp, *org_first_var;
struct variable_n *nfirst_var;
struct var_occcr *split_varf, *split_varl;
```

```
        struct var_ouccr *tmp_var_ouccr;
        struct var_ouccr *tvo_list;
        %}
%union {
        int ival;
        char *sval;
        struct cfg *cfgval;
        struct variable_n *varval;
}
%token          <sval> NO VAR
%token          OR AND EQ GEQ LEQ NEQ GT LT
%token          SCANF PRINTF AMPER SWITCH CASE DEFAULT BREAK
%token          APOS FORMATVAR INCLUDE FILENAME ENTER MAIN
%token          COLON WHILE FOR PLUSPLUS MINUSMINUS
%token          TYPE_DEC SEMICOLON COMMA IF THEN ELSE
%nonassoc "{" "}" "[" "]" "="
%left   "+" "-"
%right  "*" "/"
%left   UMINUS
%type   <sval> left_h_side variable expression assignment cond condition
%type   <sval> variables init_val final_cond inc_val ot_format ip_format
%type   <sval> inp_variable log_opr relaopr
%type   <sval> ot_variables store_switch case_val number for_1
%type   <ival>  track_of_crtl tra_else_crtl st_val if_cond stat_track_crtl
%type   <ival>  while_cond
%type   <cfgval> switch_brace else
%type   <varval> switch svar_list
%start  program
%%
/*  The following is the grammar of the input C program */
program                 :
init_tables header enters  global_dec enters main_body
{
/* Start Symbol is program */
/* init_tables initialize the linked lists necessary to create the
   control flow graphs */
/* header involves the include files and header files and other files */
/* enters accepts one or more number of carriage returns */
/* global_dec is the global declarations that occur in the program */
/* main_body describes the grammar for the program involving the defined
   specifications */

        printf("\n%d  END",stmt_no);
        curr_stmt->next = (struct stmt_list *) malloc(sizeof(struct stmt_list));
        curr_stmt = curr_stmt->next;
        curr_stmt->what_stmt = 100;
        curr_stmt->start_stmt = -1;
        curr_stmt->next = NULL;
        curr_stmt->stmt = NULL;
        curr_stmt->st_no = stmt_no;

};

init_tables     :
enters
{
        /*      The first Node of the CFG is initialized */
        first_cfg = (struct cfg *) malloc(sizeof(struct cfg));
        cfg_graph = first_cfg;
        cfg_graph->from = -1;
        cfg_graph->to = 0;
        cfg_graph->prev = NULL;
        cfg_graph->next = NULL;
        prev_cfg = cfg_graph;
        last_cfg = cfg_graph;
        var_list = NULL;
        first_var = NULL;
        /*      Initialize the linked list */
        first_stmt = (struct stmt_list *) malloc(sizeof(struct stmt_list));
        first_stmt->stmt=NULL;
        first_stmt->next = NULL;
        curr_stmt = first_stmt;
}
header          :
head header
|       head ENTER header
/*      There could be one header file or more with one or more carriage
        returns between them */
|       head
;

head            :
```

```
INCLUDE FILENAME ENTER
{
}
/*  INCLUDE  is a token that is passed to yacc by lex */
/*  FILENAME and ENTER are the same too.  For more information on their
    format check in filename lspec to see for more information */
};


global_dec      :
|       declaration  ENTER global_dec
/*      The above states the grammar for declarations that appear
        in any C program  */
/*      A declaration may be followed by one or more carriage return
        before another carriage return occurs */
;


declaration     :
type_dec variables  SEMICOLON
{
/*      A declaration may be an integer, float, or character declarations
        An integer declaration would have a keyword int followed by one
        or more variables separated by commas between them.
        */
        free($2);

/*      The cre_list is used to build a variable list that could only
        be used in the program.
        */

        cre_list = 0;
};

type_dec        :
TYPE_DEC
{
        cre_list = 1;
}
variables       :
variable COMMA variables
{
        $$ = (char *) malloc(200);
        strcpy($$,$1);
        strcat($$,",");
        strcat($$,$3);

/*      The list is produced by the pro_var_list */

        if(cre_list == 1)
                pro_var_list($1);
        free($1);
        free($3);
}
|       variable
{
        $$ = $1;

/*      The list is produced by the pro_var_list */

        if(cre_list == 1)
                pro_var_list($1);
};


variable        :
VAR
{
        $$ = (char *) malloc(20);
        strcpy($$,$1);
};
/*
The following main_body defines the grammar for the main body of the
input C program.

        As required by any C program there open and close parenthesis
        appears int the main_body.

        Inbetween the open and closed parenthesis the following
        may or may not appear in the input C program:

                (a) Declarations given by global_dec
                (b) any number of carriage returns
```

```
                (c) statements... i/o statements, if-then-else statements
                        for-loops, while-loops, and switch statements
*/

main_body       :
MAIN open_bracket global_dec enters statements '}' enters
{
        ;
};

open_bracket    :
enters '{' ENTER
;
enters          :
|       enter
;

enter           :
enter  ENTER
|       ENTER
;
/*      There may be one or more number of statements        */
statements      :
|       statement enter statements
|       statement statements
;
/*      Statement could be an Assignment Statement    */
statement       :
assignment
{
        printf("\n%d  %s",stmt_no,$1);
        curr_stmt->next = (struct stmt_list *) malloc(sizeof(struct stmt_list));
        curr_stmt = curr_stmt->next;
        curr_stmt->what_stmt = 1;
        curr_stmt->start_stmt = -1;
        curr_stmt->next = NULL;
        len = strlen($1)+5;
        curr_stmt->stmt = (char *) malloc(sizeof(char) * len);
        strcpy(curr_stmt->stmt,$1);
        curr_stmt->st_no = stmt_no;
        free($1);
        stmt_no++;
}
/*      OR Statement could be an for Statement        */
|       for_loop
{
        var_list->no = var_list->latest;
}
/*      OR Statement could be an if then else Statement      */
|       if_then_else
{
        update_list();
}
/*      OR Statement could be while loop      */
|       while_loop
{
        var_list->no = var_list->latest;
}
/*      OR Statement could be an output statement     */
|       output_stmt
{
        var_list->no = var_list->latest;
}
/*      OR Statement could be an input statement      */
|       input_stmt
{
        update_list();
}
/*      OR Statement could be an switch statement     */
|       switch_stmt
{
        update_list();
};

/*      Assignment is defined by
        Left Hand Side followed by an equal sign and the Right Hand Side
        */
assignment      :
left_h_side '=' expression SEMICOLON
{
        $$ = (char *) malloc(200);
        ret_val = find_var_nam($1,NEWNAME);
```

```
        if (ret_val == 0) {
                        printf("\n ERROR: Problem in finding New Variable ");
                        exit(0);
                        }

/*      For the variable in the left hand side (or) the variable assigned a
        value in the left hand side a new variable is assigned.
        The new variable is of the form variable##number */

/*      find_var_nam - Finds the new name for the variable passed as the
                first parameter.  On return the node or the value of
                the var_list (a linked list maintain the last name
                given to each variable) points or contains the
                information of the variable passed */

/*      Here the var_list for the variable for which a new variable name
        is given is Updated */

        tmp_var_ouccr = (struct var_ouccr *) malloc(sizeof(struct var_ouccr));
        tmp_var_ouccr->st_num = stmt_no;
        tmp_var_ouccr->next = NULL;

/*      if the variable had not been referenced before then
        var_list->ouccr_list is NULL
        If it is NULL the information has to be inserted else
        the last_ouccr is updated
*/

        if (var_list->ouccr_list == NULL){
                var_list->last_ouccr = tmp_var_ouccr;
                var_list->ouccr_list = tmp_var_ouccr;
        }
        else {
                var_list->last_ouccr = tmp_var_ouccr;
        }


        strcpy($$,$1);
        strcat($$,"=");
        strcat($$,$3);
        free($1);
        free($3);

/*      Since it is an assignment statement control flows in one direction
        From current statement to next statement.
        Hence in the following build up of CFG the stmt_no is in from
        and st_no+1 is the to statement
        In other words control flows from st_no to st_no+1
*/

        cfg_graph->next = (struct cfg *) malloc(sizeof(struct cfg));
        prev_cfg = cfg_graph;
        cfg_graph = cfg_graph->next;
        last_cfg = cfg_graph;
        cfg_graph->from = stmt_no;
        cfg_graph->to = stmt_no+1;
        cfg_graph->next = NULL;
        cfg_graph->prev = prev_cfg;
        var_list->no = var_list->latest;
}
|       expression PLUSPLUS
{
        $$ = (char *) malloc(200);
        strcpy($$,var_list->var);
        ret_val = find_var_nam($$,NEWNAME);

/*      For the variable in the left hand side (or) the variable assigned a
        value in the left hand side a new variable is assigned.
        The new variable is of the form variable##number */

/*      find_var_nam - Finds the new name for the variable passed as the
                first parameter.  On return the node or the value of
                the var_list (a linked list maintain the last name
                given to each variable) points or contains the
                information of the variable passed */

/*      Here the var_list for the variable for which a new variable name
        is given is Updated */

        tmp_var_ouccr = (struct var_ouccr *) malloc(sizeof(struct var_ouccr));
        tmp_var_ouccr->st_num = stmt_no;
        tmp_var_ouccr->next = NULL;
```

```
        if (var_list->ouccr_list == NULL){
                var_list->last_ouccr = tmp_var_ouccr;
                var_list->ouccr_list = tmp_var_ouccr;
        }
        else {
                var_list->last_ouccr = tmp_var_ouccr;
        }
        strcat($$,"=");
        strcat($$,$1);
        strcat($$,"+1");
        free($1);
}
|       expression MINUSMINUS
{
        $$ = (char *) malloc(60);
        strcpy($$,var_list->var);
        ret_val = find_var_nam($$,NEWNAME);

/*      For the variable in the left hand side (or) the variable assigned a
        value in the left hand side a new variable is assigned.
        The new variable is of the form variable##number */

/*      find_var_nam - Finds the new name for the variable passed as the
                first parameter.  On return the node or the value of
                the var_list (a linked list maintain the last name
                given to each variable) points or contains the
                information of the variable passed */

/*      Here the var_list for the variable for which a new variable name
        is given is Updated */

        tmp_var_ouccr = (struct var_ouccr *) malloc(sizeof(struct var_ouccr));
        tmp_var_ouccr->st_num = stmt_no;
        tmp_var_ouccr->next = NULL;
        if (var_list->ouccr_list == NULL){
                var_list->last_ouccr = tmp_var_ouccr;
                var_list->ouccr_list = tmp_var_ouccr;
        }
        else {
                var_list->last_ouccr->next = tmp_var_ouccr;
                var_list->last_ouccr = tmp_var_ouccr;
        }
        strcat($$,"=");
        strcat($$,$1);
        strcat($$,"-1");
        free($1);
};

/*      The left hand side of the assignment statement is defined by a variable
        as given as follows */
left_h_side             :
variable
{
        $$ = $1;
};

/*      The following grammar defines the grammar for the expression   */


expression      :
expression '+' expression
{
        $$ = (char *) malloc(200);
        strcpy($$,$1);
        strcat($$,"+");
        strcat($$,$3);
        free($1);
        free($3);
}
|       expression '-' expression
{
        $$ = (char *) malloc(200);
        strcpy($$,$1);
        strcat($$,"-");
        strcat($$,$3);
        free($1);
        free($3);
}
|       expression '/' expression
{
        $$ = (char *) malloc(200);
```

```
        strcpy($$,$1);
        strcat($$,"/");
        strcat($$,$3);
        free($1);
        free($3);
}
|       expression '*' expression
{
        $$ = (char *) malloc(200);
        strcpy($$,$1);
        strcat($$,"*");
        strcat($$,$3);
        free($1);
        free($3);
}
|       '(' expression ')'
{
        $$ = (char *) malloc(200);
        strcpy($$,"(");
        strcat($$,$2);
        strcat($$,")");
}
|       '-' expression %prec UMINUS
{
        $$ = (char *) malloc(200);
        strcpy($$,"-");
        strcat($$,$2);
        free($2);
}
|       variable
{
        ret_val = find_var_nam($1,OLDNAME);
        $$ = $1;
}
|       number
{
        $$ = $1;
};


number          :
NO
{
        $$ = (char *) malloc(10);
        strcpy($$,$1);
}

/*      The following grammar defines the syntax for if then else statement */

/*      The problem in a if then else statement is that at the end of
        if then else statement control merges from two different paths

        After an if condition is encountered one or more than one
        statement is encountered.
        After which the false statements may or may not appear

        So the problem is forward referencing so that after the last
        statement in the if true_cond statements is executed
        control is transferred  to the statement after the false statements

        The problem is solved by using an algorithm similar
        to the backtracking algorithm
        */

if_then_else    :
if_cond track_of_crtl true_state else enters tra_else_crtl false_state
{
/*      The following statements would be executed only if the
        the above defined grammar is satisfied  */

/*      The main function of the following statements is to make to
        control of the last statement to point to flow into the merge
        node */

/*      track_of_control stores the statement number of the last
        statement in the statements that get executed as a result of
        the true condition */

/*      tra_else_crtl  indicates the statement number the control is to
        be transferred if the true condition fails  */
```

```
/*      The following 8 statements allows the control to flow to the
        execution of the false statements if the true condition fails.

        These statements creates the flow of control in two directions
        when an if statement is encountered
        */

        cfg_graph->next = (struct cfg *) malloc(sizeof(struct cfg));
        prev_cfg = cfg_graph;
        cfg_graph = cfg_graph->next;
        last_cfg = cfg_graph;
        cfg_graph->prev = prev_cfg;
        cfg_graph->from = $2;
        cfg_graph->to = $6;
        cfg_graph->next = NULL;


        /*      Merge Node */
/*      A Merge node is created and the control flows into this node
        when the nodes merge following the if condition. */
/*      The following sets the control to flow from the merge node to the
        next node.
        */
        cfg_graph->next = (struct cfg *) malloc(sizeof(struct cfg));
        prev_cfg = cfg_graph;
        cfg_graph = cfg_graph->next;
        last_cfg = cfg_graph;
        cfg_graph->prev = prev_cfg;
        cfg_graph->from = stmt_no;
        cfg_graph->to = stmt_no+1;
        cfg_graph->next = NULL;

/*      The following sets the control from the last statement of the if
        condition then statements to flow into the merge node.
        */
        else_cfg = $4;
        else_cfg->to = stmt_no;

/*      The following statements set the MERGE NODE so the the control
        could flow into this node after executing either one of the
        statements depending on the condition
        */

        curr_stmt->next = (struct stmt_list *) malloc(sizeof(struct stmt_list));
        curr_stmt = curr_stmt->next;
        curr_stmt->next = NULL;
        curr_stmt->what_stmt = 20;
        curr_stmt->start_stmt = $1;
        curr_stmt->stmt = NULL,
        curr_stmt->st_no = stmt_no;
        stmt_no++;
}
|       if_cond track_of_crtl true_state
{

/*      In the IF cond STATEMENTS the problem of forward referencing is much
        simpler.
        If the condition is TRUE control should be transferred to the
        first statement in the IF condition statements
        If the condition is FALSE control should be transferred to the
        statement after the last statement of the IF condition statements.
        */

/*      track_of_crtl indicates a statement number from where the control
        should also point to this current statement number which is the
        statement after the last statement of the IF condition statements.
        */


        cfg_graph->next = (struct cfg *) malloc(sizeof(struct cfg));
        prev_cfg = cfg_graph;
        cfg_graph = cfg_graph->next;
        last_cfg = cfg_graph;
        cfg_graph->prev = prev_cfg;
        cfg_graph->from = $2;
        cfg_graph->to = stmt_no;
        cfg_graph->next = NULL;
        /*      Merge Node */
/*      A Merge node is created and the control flows into this node
        when the nodes merge following the if condition. */
/*      The following sets the control to flow from the merge node to the
        next node.
```

```
        */
/*
        */
        cfg_graph->next = (struct cfg *) malloc(sizeof(struct cfg));
        prev_cfg = cfg_graph;
        cfg_graph = cfg_graph->next;
        last_cfg = cfg_graph;
        cfg_graph->prev = prev_cfg;
        cfg_graph->from = stmt_no;
        cfg_graph->to = stmt_no+1;
        cfg_graph->next = NULL;


/*      The following statements set the MERGE NODE so the the control
        could flow into this node after executing either one of the
        statements depending on the condition
        */

        curr_stmt->next = (struct stmt_list *) malloc(sizeof(struct stmt_list));
        curr_stmt = curr_stmt->next;
        curr_stmt->next = NULL;
        curr_stmt->what_stmt = 20;

/*      what_stmt = 20 MEANS that the current node is a MERGE NODE */

        curr_stmt->start_stmt = $1;
        curr_stmt->stmt = NULL,
        curr_stmt->st_no = stmt_no;
        stmt_no++;
};


else    :
ELSE
{
/*      This returns the pointer to the structure where the control needs
        to be transferred to the Merge node */

        $$ = cfg_graph;
};

tra_else_crtl   :
{

/*      tra_else_crtl  indicates the statement number the control is to
        be transferred if the true condition fails  */

        $$ = stmt_no;
};


if_cond                 :
IF cond
{
        printf("\n%d if %s",stmt_no,$2);
        $$ = stmt_no;

/*      curr_stmt - is a list of the nodes contain the data and the
        information about a statement */

/*      The following 9 statements is used to build a node for the
        IF CONDITION */

        curr_stmt->next = (struct stmt_list *) malloc(sizeof(struct stmt_list));
        curr_stmt = curr_stmt->next;
        curr_stmt->next = NULL;
        curr_stmt->what_stmt = 8;

/*      what_stmt = 8 MEANS the node contains an IF CONDITION */

        curr_stmt->start_stmt = -1;
        len = strlen($2)+5;
        curr_stmt->stmt = (char *) malloc(sizeof(char) * len);
        strcpy(curr_stmt->stmt,$2);
        curr_stmt->st_no = stmt_no;

/*      The split_varf keeps track of the statements after which
        there is a possibility of control to flow in more than one direction
        The split_varl indicates the last statement where the control
        could possibly flow in more than one direction
        */
```

```
        if (split_varf == NULL){
                split_varf = (struct var_ouccr *) malloc(sizeof(struct var_ouccr));
                split_varf->st_num = stmt_no;
                split_varf->next = NULL;
                split_varl = split_varf;
        }
        else {
                split_varl->next = (struct var_ouccr *) malloc(sizeof(struct var_ouccr));
                split_varl = split_varl->next;
                split_varl->st_num = stmt_no;
                split_varl->next = NULL;
        }

        control_t_st = stmt_no;

/*      The following statements initialize the TRUE condition causing the
        control to flow to the first statement of the TRUE condition
        statements */

        cfg_graph->next = (struct cfg *) malloc(sizeof(struct cfg));
        prev_cfg = cfg_graph;
        cfg_graph = cfg_graph->next;
        last_cfg = cfg_graph;
        cfg_graph->prev = prev_cfg;
        cfg_graph->from = control_t_st;
        cfg_graph->to = stmt_no+1;
        cfg_graph->next = NULL;
        stmt_no++;
};

/*      The following grammars define the syntax of the condition statement
        */

cond            :
'(' condition ')'
{
        $$ = $2;
}
|       '(' cond log_opr cond ')'
{
        $$ = (char *) malloc(200);
        strcpy($$,$2);
        strcat($$,$3);
        strcat($$,$4);
        free($2);
        free($3);
        free($4);
};


condition       :
expression relaopr expression
{
        $$ = (char *) malloc(200);
        strcpy($$,$1);
        strcat($$,$2);
        strcat($$,$3);
        free($1);
        free($2);
        free($3);
}


log_opr                 :
AND
{
        $$ = (char *) malloc(10);
        strcpy($$,"&&");
}
|       OR
{
        $$ = (char *) malloc(10);
        strcpy($$,"||");
};


relaopr                 :
EQ
{
        $$ = (char *) malloc(10);
        strcpy($$,"==");
}
```

```
|       GEQ
{
        $$ = (char *) malloc(10);
        strcpy($$,">=");
}
|       LEQ
{
        $$ = (char *) malloc(10);
        strcpy($$,"<=");
}
|       NEQ
{
        $$ = (char *) malloc(10);
        strcpy($$,"<>");
}
|       GT
{
        $$ = (char *) malloc(10);
        strcpy($$,">");
}
|       LT
{
        $$ = (char *) malloc(10);
        strcpy($$,"<");
};


true_state      :
'{'enters  statements '}' enters
|       enters statement enters
{
};


end_brace       :
'}'
{
};

false_state     :
true_state
;

stat_track_crtl :       { $$ = stmt_no-1; }
                ;

/*      The following gives the grammar for the FOR LOOP statement */
/*      In a for loop the the initial value, the final condition and the
        running variable are the three statements inside a for loop
        statement.
        */


for_loop        :
for_l stat_track_crtl track_of_crtl loop_body
{
/*      for_l - Contains a string which is the running variable

        track_of_crtl -  Keeps track of where the control should flow
                in the event of a FALSE on the condition i.e. out of the loop

        loop_body -     defines the body of the loop.

        So after the body of the loop is recognized the running variable
        should appear and after which control is transferred to the
        condition statement or track_of_crtl here which contains the statement
        number of the condition statement.
        */

        strip_string($1);

/*      strip_string(string) - updates the variable that occurs in the
                string with a new variable name which has the following
                format variable##number         */

/*      The running variable is allocated a node which contains the
        information about the loop */

        printf("\n%d %s",stmt_no,$1);
        curr_stmt->next = (struct stmt_list *) malloc(sizeof(struct stmt_list));
        curr_stmt = curr_stmt->next;
        curr_stmt->next = NULL;
```

```
        curr_stmt->what_stmt = -1;
        curr_stmt->start_stmt = $2;
        len = strlen($1)+5;

/*      When the if CONDITION or when the CONDITION in the for loop fails
        control is transferred out of the loop.
        */

        curr_stmt->stmt = (char *) malloc(sizeof(char) * len);
        strcpy(curr_stmt->stmt,$1);
        curr_stmt->st_no = stmt_no;
        cfg_graph->next = (struct cfg *) malloc(sizeof(struct cfg));
        prev_cfg = cfg_graph;
        cfg_graph = cfg_graph->next;
        last_cfg = cfg_graph;
        cfg_graph->prev = prev_cfg;

/*      The following statements transfer the control outside the loop */

        cfg_graph->to = stmt_no+2;
        cfg_graph->from = $3;


/*      Control from the running variable is transferred to the statement
        where a eta node is likely to be introduced in a later stage */

        cfg_graph->next = (struct cfg *) malloc(sizeof(struct cfg));
        prev_cfg = cfg_graph;
        cfg_graph = cfg_graph->next;
        last_cfg = cfg_graph;
        cfg_graph->prev = prev_cfg;
        cfg_graph->from = stmt_no;
        cfg_graph->to = stmt_no+1;
        cfg_graph->next = NULL;
        stmt_no++;

/*      The node where the eta function is to be introduced is created */


        curr_stmt->next = (struct stmt_list *) malloc(sizeof(struct stmt_list));
        curr_stmt = curr_stmt->next;
        curr_stmt->next = NULL;
        curr_stmt->what_stmt = 21;
        curr_stmt->start_stmt = $2;
        curr_stmt->stmt = NULL;
        curr_stmt->st_no = stmt_no;
/* */
/*      The control is transferred to the if condition in the for loop */

        cfg_graph->next = (struct cfg *) malloc(sizeof(struct cfg));
        prev_cfg = cfg_graph;
        cfg_graph = cfg_graph->next;
        last_cfg = cfg_graph;
        cfg_graph->prev = prev_cfg;
        cfg_graph->from = stmt_no;
        cfg_graph->to = $3;
        cfg_graph->next = NULL;
        stmt_no++;
};

/*  The format of the for loop is shown below */

for_l           :
FOR  '('init_val final_cond inc_val ')'
{
        control_t_st = 0;
        printf("\n%d %s",stmt_no,$3);

/*      The initial variable is initialized.  The initial value is
        initialized.*/

        curr_stmt->next = (struct stmt_list *) malloc(sizeof(struct stmt_list));
        curr_stmt = curr_stmt->next;
        curr_stmt->next = NULL;
        curr_stmt->what_stmt = 3;

/*      what_stmt = 3 MEANS IT is an assignment statement in the for loop
        It also means that the current statement is a initial assignment
        statement of the for loop */

        curr_stmt->start_stmt = stmt_no+2;
        len = strlen($3)+5;
```

```
        curr_stmt->stmt = (char *) malloc(sizeof(char) * len);
        strcpy(curr_stmt->stmt,$3);
        curr_stmt->st_no = stmt_no;
        stmt_no++;

/*      A loop in a later stage (i.e.) at the GSA stage here the Lamda is
        to be inserted */

        curr_stmt->next = (struct stmt_list *) malloc(sizeof(struct stmt_list));
        curr_stmt = curr_stmt->next;
        curr_stmt->next = NULL;
        curr_stmt->what_stmt = 51;

/*
        */

        curr_stmt->start_stmt = stmt_no+1;
        curr_stmt->stmt = NULL;
        curr_stmt->st_no = stmt_no;

/* This Block is for the SWITCH which occurs in the pdw * Final Phase */

        control_t_st = stmt_no;
        cfg_graph->next = (struct cfg *) malloc(sizeof(struct cfg));
        prev_cfg = cfg_graph;
        cfg_graph = cfg_graph->next;
        last_cfg = cfg_graph;
        cfg_graph->prev = prev_cfg;
        cfg_graph->from = control_t_st;
        cfg_graph->to = stmt_no+1;
        cfg_graph->next = NULL;
        stmt_no++;

/*      The condition in the for loop is stored in a node */

        printf("\n%d %s",stmt_no,$4);
        curr_stmt->next = (struct stmt_list *) malloc(sizeof(struct stmt_list));
        curr_stmt = curr_stmt->next;
        curr_stmt->next = NULL;
        curr_stmt->what_stmt = 5;
        curr_stmt->start_stmt = -1;
        len = strlen($4)+5;
        curr_stmt->stmt = (char *) malloc(sizeof(char) * len);
        strcpy(curr_stmt->stmt,$4);
        curr_stmt->st_no = stmt_no;


/*      The split_varf keeps track of the statements after which
        there is a possibility of control to flow in more than one direction
        The split_varl indicates the last statement where the control
        could possibly flow in more than one direction
        */


        if (split_varf == NULL){
                split_varf = (struct var_ouccr *) malloc(sizeof(struct var_ouccr));
                split_varf->st_num = stmt_no;
                split_varf->next = NULL;
                split_varl = split_varf;
        }
        else {
                split_varl->next = (struct var_ouccr *) malloc(sizeof(struct var_ouccr));
                split_varl = split_varl->next;
                split_varl->st_num = stmt_no;
                split_varl->next = NULL;
        }

/*      Control is transferred to the next node after the initialization
        of a node or an indication where the GSA function may be
        introduced
        */

        control_t_st = stmt_no;
        cfg_graph->next = (struct cfg *) malloc(sizeof(struct cfg));
        prev_cfg = cfg_graph;
        cfg_graph = cfg_graph->next;
        last_cfg = cfg_graph;
        cfg_graph->prev = prev_cfg;
        cfg_graph->from = control_t_st;
        cfg_graph->to = stmt_no+1;
        cfg_graph->next = NULL;
        stmt_no++;
```

```
/*      Possible introduction of the SWITCH in a PDW is created */


        curr_stmt->next = (struct stmt_list *) malloc(sizeof(struct stmt_list));
        curr_stmt = curr_stmt->next;
        curr_stmt->next = NULL;
        curr_stmt->what_stmt = 52;
        curr_stmt->start_stmt = stmt_no-1;
        curr_stmt->stmt = NULL;
        curr_stmt->st_no = stmt_no;


/*      Control is passed from the node which indicates the position
        where a SWITCH may be introduced */

        cfg_graph->next = (struct cfg *) malloc(sizeof(struct cfg));
        prev_cfg = cfg_graph;
        cfg_graph = cfg_graph->next;
        last_cfg = cfg_graph;
        cfg_graph->prev = prev_cfg;
        cfg_graph->from = stmt_no;
        cfg_graph->to = stmt_no+1;
        cfg_graph->next = NULL;
        stmt_no++;

/*       The step value or the increment statement is returned */


        $$ = $5;
};



/*      track_of_control stores the statement number of the last
        statement in the statements that get executed as a result of
        the true condition */

track_of_crtl   :
{
        $$ = control_t_st;
};


init_val        :
assignment
{
        $$ = $1;
}
|       SEMICOLON
{
        $$ = NULL;
};


final_cond      :
condition SEMICOLON
{
        $$ = $1;
}
|       SEMICOLON
{
        $$ = NULL;
};


inc_val                 :
assignment
{
        $$ = $1;
        var_list->latest = var_list->no;
}
|       SEMICOLON
{
        $$ = NULL;
};


loop_body       :
'{' enters statements end_brace
|       enters statement
;
```

```
while_loop      :
while_cond track_of_crtl loop_body
{

/*
        After the execution of the last statement of the while loop
        control is transferred to the condition of the loop.
        track_of_crtl - Stores the statement number of the condition of the
                while loop
        */

        cfg_graph->next = (struct cfg *) malloc(sizeof(struct cfg));
        prev_cfg = cfg_graph;
        cfg_graph = cfg_graph->next;
        last_cfg = cfg_graph;
        cfg_graph->prev = prev_cfg;
        cfg_graph->from = stmt_no;
        cfg_graph->to = $2;
        cfg_graph->next = NULL;

/*      A possible introduction if a GSA function is given by a 21 in the
        what statement.
        Hence a GSA node is introduced.  This node is used to be more as
        an indication than an actual introduction.
        */

        curr_stmt->next = (struct stmt_list *) malloc(sizeof(struct stmt_list));
        curr_stmt = curr_stmt->next;
        curr_stmt->next = NULL;
        curr_stmt->what_stmt = 21;
        curr_stmt->start_stmt = $2;
        curr_stmt->stmt = NULL;
        curr_stmt->st_no = stmt_no;
        stmt_no++;

/*      The control from the while CONDITION is allowed to flow out of the loop
        by the following statements
        */

        cfg_graph->next = (struct cfg *) malloc(sizeof(struct cfg));
        prev_cfg = cfg_graph;
        cfg_graph = cfg_graph->next;
        last_cfg  = cfg_graph;
        cfg_graph->prev = prev_cfg;
        cfg_graph->from = $2;
        cfg_graph->to = stmt_no;
        cfg_graph->next = NULL;
};


while_cond      :
WHILE cond
{

/*      The following node indicates the possible introduction of a GSA
        function into the code */

        curr_stmt->next = (struct stmt_list *) malloc(sizeof(struct stmt_list));
        curr_stmt = curr_stmt->next;
        curr_stmt->next = NULL;
        curr_stmt->what_stmt = 51;
        curr_stmt->start_stmt = stmt_no+1;
        curr_stmt->stmt = NULL;
        curr_stmt->st_no = stmt_no;

/*      The flow of control from the GSA function node to the next node
        */

        cfg_graph->next = (struct cfg *) malloc(sizeof(struct cfg));
        prev_cfg = cfg_graph;
        cfg_graph = cfg_graph->next;
        last_cfg = cfg_graph;
        cfg_graph->prev = prev_cfg;
        cfg_graph->from = stmt_no;
        cfg_graph->to = stmt_no+1;
        cfg_graph->next = NULL;
        stmt_no++;

        printf("\n%d %s",stmt_no,$2);

/*      The CONDITION in the while loop is stored in the following
        statements */
```

```
        curr_stmt->next = (struct stmt_list *) malloc(sizeof(struct stmt_list));
        curr_stmt = curr_stmt->next;
        curr_stmt->next = NULL;
        curr_stmt->what_stmt = 6;
        curr_stmt->start_stmt = -1;
        $$ = stmt_no;
        len = strlen($2)+5;
        curr_stmt->stmt = (char *) malloc(sizeof(char) * len);
        strcpy(curr_stmt->stmt,$2);
        curr_stmt->st_no = stmt_no;


/*      The split_varf keeps track of the statements after which
        there is a possibility of control to flow in more than one direction
        The split_varl indicates the last statement where the control
        could possibly flow in more than one direction
        */

        if (split_varf == NULL){
                split_varf = (struct var_ouccr *) malloc(sizeof(struct var_ouccr));
                split_varf->st_num = stmt_no;
                split_varf->next = NULL;
                split_varl = split_varf;
        }
        else {
                split_varl->next = (struct var_ouccr *) malloc(sizeof(struct var_ouccr));
                split_varl = split_varl->next;
                split_varl->st_num = stmt_no;
                split_varl->next = NULL;
        }


/*      Control flows into the SWITCH indicator node */

        control_t_st = stmt_no;
        cfg_graph->next = (struct cfg *) malloc(sizeof(struct cfg));
        prev_cfg = cfg_graph;
        cfg_graph = cfg_graph->next;
        last_cfg = cfg_graph;
        cfg_graph->prev = prev_cfg;
        cfg_graph->from = stmt_no;
        cfg_graph->to = stmt_no+1;
        cfg_graph->next = NULL;
        stmt_no++;

/*      Possible introduction of SWITCH in a PDW is guessed and hence
        a node indicating the possible introduction is created */

        curr_stmt->next = (struct stmt_list *) malloc(sizeof(struct stmt_list));
        curr_stmt = curr_stmt->next;
        curr_stmt->next = NULL;
        curr_stmt->what_stmt = 52;
        curr_stmt->start_stmt = stmt_no-1;
        curr_stmt->stmt = NULL;
        curr_stmt->st_no = stmt_no;

/*      Control flows into the SWITCH indicator node */

        cfg_graph->next = (struct cfg *) malloc(sizeof(struct cfg));
        prev_cfg = cfg_graph;
        cfg_graph = cfg_graph->next;
        last_cfg = cfg_graph;
        cfg_graph->prev = prev_cfg;
        cfg_graph->from = stmt_no;
        cfg_graph->to = stmt_no+1;
        cfg_graph->next = NULL;
        stmt_no++;
};

/*      The following grammar gives the syntax for the output statement */

output_stmt     :
PRINTF ot_format
{
        printf("\n%d Output %s",stmt_no,$2);

/*      The output statement is stored in the current statement and all
        information regarding the output statement is also stored */

        curr_stmt->next = (struct stmt_list *) malloc(sizeof(struct stmt_list));
        curr_stmt = curr_stmt->next;
        curr_stmt->next = NULL;
```

```
        curr_stmt->what_stmt = 7;

/*      what_stmt = 7 indicates that the statement is an output statement */

        curr_stmt->start_stmt = -1;
        len = strlen($2)+5;
        curr_stmt->stmt = (char *) malloc(sizeof(char) * len);
        strcpy(curr_stmt->stmt,$2);
        curr_stmt->st_no = stmt_no;

/*      From the output statement control flows linearly out into the
        next statement and is accomplished by the following statements */


        cfg_graph->next = (struct cfg *) malloc(sizeof(struct cfg));
        prev_cfg = cfg_graph;
        cfg_graph = cfg_graph->next;
        last_cfg = cfg_graph;
        cfg_graph->prev = prev_cfg;
        cfg_graph->from = stmt_no;
        cfg_graph->to = stmt_no+1;
        cfg_graph->next = NULL;
        stmt_no++;
};


ot_format       :
first_half COMMA ot_variables ')' SEMICOLON
{
        $$ = $3;
};

ot_variables    :
variable COMMA ot_variables
{
        $$ = (char *) malloc(200);
        ret_val = find_var_nam($1,OLDNAME);
        strcpy($$,$1);
        strcat($$,",");
        strcat($$,$3);
        free($1);
        free($3);
}
|       variable
{
        ret_val = find_var_nam($1,OLDNAME);
        $$ = $1;
};


first_half      :
'(' APOS var_format APOS
{
};


var_format      :
FORMATVAR COMMA var_format
|       FORMATVAR var_format
|       FORMATVAR
;

/*      The following grammar defines the syntax of the input statement */

input_stmt      :
SCANF ip_format
{
        printf("\n%d Input %s",stmt_no,$2);

/*      The input statement is stored in the linked list along with
        the other information */

        curr_stmt->next = (struct stmt_list *) malloc(sizeof(struct stmt_list));
        curr_stmt = curr_stmt->next;
        curr_stmt->next = NULL;
        curr_stmt->what_stmt = 2;

/*      what_stmt = 2 indicates that the current statement is an
        input statement */

        curr_stmt->start_stmt = -1;
        len = strlen($2)+5;
```

```
        curr_stmt->stmt = (char *) malloc(sizeof(char) * len);
        strcpy(curr_stmt->stmt,$2);
        curr_stmt->st_no = stmt_no;

/*      Control from an input statement flows out into the next statement
        following the input statement */

        cfg_graph->next = (struct cfg *) malloc(sizeof(struct cfg));
        prev_cfg = cfg_graph;
        cfg_graph = cfg_graph->next;
        last_cfg = cfg_graph;
        cfg_graph->prev = prev_cfg;
        cfg_graph->from = stmt_no;
        cfg_graph->to = stmt_no+1;
        cfg_graph->next = NULL;
        stmt_no++;
};


ip_format       :
first_half COMMA inp_variable ')' SEMICOLON
{
        $$ = $3;
};

/*      The format in the inp_list is given by the inp_variable where
        an ampersign should occur before every input variable defined
        as an integer */

inp_variable    :
{
        $$ =  (char *) malloc(2);
        strcpy($$,"");
}
|       AMPER variable COMMA inp_variable
{
        $$ = (char *) malloc(200);
        ret_val = find_var_nam($2,NEWNAME);

/*      For each variable occurring in the input statement anew variable
        is assigned.
        The new variable is of the form variable##number */

/*      find_var_nam - Finds the new name for the variable passed as the
                first parameter.  On return the node or the value of
                the var_list (a linked list maintain the last name
                given to each variable) points or contains the
                information of the variable passed */

/*      Here the var_list for the variable for which a new variable name
        is given is Updated */


        tmp_var_ouccr = (struct var_ouccr *) malloc(sizeof(struct var_ouccr));
        tmp_var_ouccr->st_num = stmt_no;
        tmp_var_ouccr->next = NULL;

/*      if the variable had not been referenced before then
        var_list->ouccr_list is NULL
        If It is NULL the information has to be inserted else
        the information has to be updated
        */


        if (var_list->ouccr_list == NULL){
                var_list->last_ouccr = tmp_var_ouccr;
                var_list->ouccr_list = tmp_var_ouccr;
        }
        else {
                var_list->last_ouccr = tmp_var_ouccr;
        }
        strcpy($$,$2);
        strcat($$,",");
        strcat($$,$4);
        free($2);
}
|       variable COMMA inp_variable
{
        $$ = (char *) malloc(200);
        ret_val = find_var_nam($1,NEWNAME);

/*      For each variable occurring in the input statement anew variable
```

```
        is assigned.
        The new variable is of the form variable##number */

/*      find_var_nam - Finds the new name for the variable passed as the
                first parameter.  On return the node or the value of
                the var_list (a linked list maintain the last name
                given to each variable) points or contains the
                information of the variable passed */

/*      Here the var_list for the variable for which a new variable name
        is given is Updated */


        tmp_var_ouccr = (struct var_ouccr *) malloc(sizeof(struct var_ouccr));
        tmp_var_ouccr->st_num = stmt_no;
        tmp_var_ouccr->next = NULL;

/*      if the variable had not been referenced before then
        var_list->ouccr_list is NULL
        If it is NULL the information has to be inserted else
        the information has to be updated
        */

        if (var_list->ouccr_list == NULL){
                var_list->last_ouccr = tmp_var_ouccr;
                var_list->ouccr_list = tmp_var_ouccr;
        }
        else {
                var_list->last_ouccr->next = tmp_var_ouccr;
                var_list->last_ouccr = tmp_var_ouccr;
        }
        strcpy($$,$1);
        strcat($$,",");
        strcat($$,$3);
        free($1);
}
|       AMPER variable
{
        ret_val = find_var_nam($2,NEWNAME);

/*      For each variable occurring in the input statement anew variable
        is assigned.
        The new variable is of the form variable##number */

/*      find_var_nam - Finds the new name for the variable passed as the
                first parameter.  On return the node or the value of
                the var_list (a linked list maintain the last name
                given to each variable) points or contains the
                information of the variable passed */

/*      Here the var_list for the variable for which a new variable name
        is given is Updated */


        tmp_var_ouccr = (struct var_ouccr *) malloc(sizeof(struct var_ouccr));
        tmp_var_ouccr->st_num = stmt_no;
        tmp_var_ouccr->next = NULL;

/*      if the variable had not been referenced before then
        var_list->ouccr_list is NULL
        If it is NULL the information has to be inserted else
        the information has to be updated
        */

        if (var_list->ouccr_list == NULL){
                var_list->last_ouccr = tmp_var_ouccr;
                var_list->ouccr_list = tmp_var_ouccr;
        }
        else {
                var_list->last_ouccr->next = tmp_var_ouccr;
                var_list->last_ouccr = tmp_var_ouccr;
        }
        $$ = $2;
}
|       variable
{
        ret_val = find_var_nam($1,NEWNAME);

/*      For each variable occurring in the input statement anew variable
        is assigned.
        The new variable is of the form variable##number */
```

```
/*      find_var_nam - Finds the new name for the variable passed as the
                first parameter.  On return the node or the value of
                the var_list (a linked list maintain the last name
                given to each variable) points or contains the
                information of the variable passed */

/*      Here the var_list for the variable for which a new variable name
        is given is Updated */

        tmp_var_ouccr = (struct var_ouccr *) malloc(sizeof(struct var_ouccr));
        tmp_var_ouccr->st_num = stmt_no;
        tmp_var_ouccr->next = NULL;

/*      if the variable had not been referenced before then
        var_list->ouccr_list is NULL
        If it is NULL the information has to be inserted else
        the information has to be updated
        */

        if (var_list->ouccr_list == NULL){
                var_list->last_ouccr = tmp_var_ouccr;
                var_list->ouccr_list = tmp_var_ouccr;
        }
        else {
                var_list->last_ouccr->next = tmp_var_ouccr;
                var_list->last_ouccr = tmp_var_ouccr;
        }
        $$ = $1;
};

/*      The following defines the grammar for the switch statement */

switch_stmt     :
switch switch_body
{
};


switch          :
SWITCH '(' variable ')' enters
{
        ret_val = find_var_nam($3,OLDNAME);
        strcpy(switch_var,$3);

/*      switch_var is a string variable that keeps track of the variable
                that accrues in the switch statement
        */

        switch_val--;
        $$ = first_var;
};


switch_body     :
switch_brace enters tra_else_crtl case_struct '}'
{
/*
        switch_brace -->  This is used to store the pointer to a structure
                which is used to trace back and reset all dangling pointers
                point to the Merge node.

        tra_else_crtl --> To keep track of the beginning of the switch
                statement
*/

        cfg_switch = $1;
        printf("\n%d Merge Node ",stmt_no);

/*      A merge node is created so that irrespective of which statements
        execute control is transferred after the execution */

        curr_stmt->next = (struct stmt_list *) malloc(sizeof(struct stmt_list));
        curr_stmt = curr_stmt->next;
        curr_stmt->next = NULL;
        curr_stmt->what_stmt = 12;
        curr_stmt->start_stmt = $3;
        curr_stmt->stmt = NULL,
        curr_stmt->st_no = stmt_no;

/*      Control flows from the Merge node to the next node following the
        merge node */
```

```
        cfg_graph->next = (struct cfg *) malloc(sizeof(struct cfg));
        prev_cfg = cfg_graph;
        cfg_graph = cfg_graph->next;
        last_cfg = cfg_graph;
        cfg_graph->prev = prev_cfg;
        cfg_graph->from = stmt_no;
        cfg_graph->to = stmt_no+1;
        cfg_graph->next = NULL;

/*      Since for each statement a statement number is introduced to
        make the creation of the CFG from there the PDW it becomes
        very difficult to make the control flow to the merge node */

/*      So an imaginary number is stored into the flow of control
        as each time the switch statement is encountered

        This imaginary numbers are later reset to the merge node

        The following while loop performs the above operation
        */

/*      Since nested switch statements are allowed the switch_val initialize
        a new imaginary number for each switch statement encountered */

        while(cfg_switch != NULL){
                if(cfg_switch->to == switch_val) {
                        cfg_switch->to = stmt_no;
                }
                cfg_switch = cfg_switch->next;
        }
        stmt_no++;
};


switch_brace    :
'{'
{
/*
        switch_brace -->  This is used to store the pointer to a structure
                which is used to trace back and reset all dangling pointers
                point to the Merge node.

        */
        cfg_switch = cfg_graph;
        $$ = cfg_switch;
};


case_struct     :
case_stmts  svar_list df_stmt

case_stmts      :
cs_stmt case_stmts
|       cs_stmt
;

cs_stmt                 :
svar_list case_stmt
{
        nfirst_var = $1;
        update_var_list();

/*      The variable list should be updated.
                    X
                  /   \
                Y       Z
                  \     /
                    W

                Consider X is a node with an IF condition

                Let there be a path from X to W through Y and another path
                from X to W through Z.

                Let a variable a be initialized a value along Y and referenced
                along Z.

                The variable would be given a name a##2 along Y and it would
                be referenced as a##1 assuming a##1 is the name at X.

                To accomplish this effect there is a list that needs to be
                updated which is done using update_var_list
```

```
                copy_var_list() is also used in the process of addressing
                the above described problem.
        */

};


svar_list       :
{
        $$ = first_var;
        copy_var_list();
};


df_stmt                 :
svar_list default_stmt
{
        nfirst_var = $1;
        update_var_list();
}

default_stmt    :
default store_switch track_of_crtl st_val enters  statements BREAK SEMICOLON enters
{
        cfg_graph->to = switch_val;
        cfg_graph->next = NULL;
};


default :  DEFAULT COLON
{
        printf("\n%d else",stmt_no);

/*      The default statement has the ELSE condition and hence 10 is
        stored in the what_stmt indicating this node is the default
        of the switch function */


        curr_stmt->next = (struct stmt_list *) malloc(sizeof(struct stmt_list));
        curr_stmt = curr_stmt->next;
        curr_stmt->next = NULL;
        curr_stmt->what_stmt = 10;
        curr_stmt->start_stmt = -1;
        curr_stmt->stmt = NULL,
        curr_stmt->st_no = stmt_no;

/*      Since this node is a default node control flows in one direction
        and hence to the immediate next statement */

        cfg_graph->next = (struct cfg *) malloc(sizeof(struct cfg));
        prev_cfg = cfg_graph;
        cfg_graph = cfg_graph->next;
        last_cfg = cfg_graph;
        cfg_graph->prev = prev_cfg;
        cfg_graph->from = stmt_no;
        cfg_graph->to = stmt_no+1;
        cfg_graph->next = NULL;
        stmt_no++;
};

case_stmt       :
tra_else_crtl case_val store_switch track_of_crtl st_val enters statements BREAK SEMICOLON
enters
{

/*      Most of the above non-terminals used above are used because of the
        presence of nested Switch statements.

        tra_else_crtl --> is used to store the statement number of the
                condition indicating the beginning of the switch statement
        case_val --> contains the condition (i.e.) the switch_var == value int
                        the case statement
        track_of_crtl -->  Make the left pointer or Make to control
                flow in the event of a FALSE occurring
        st_val -->  The imaginary value to make the control flow after
                executing the statements following the an occurrence
                of TRUE condition

        */

/*      The following statements allow the transfer of control from
        the merge node to the immediate next statement */
```

```
        prev_cfg = cfg_graph;

        cfg_graph->next = (struct cfg *) malloc(sizeof(struct c
        cfg_graph = cfg_graph->next;
        last_cfg = cfg_graph;
        cfg_graph->prev = prev_cfg;
        cfg_graph->from = stmt_no;
        cfg_graph->to = stmt_no+1;

/*      The following allows the creation of the Merge node */

        cfg_graph->next = NULL;
        printf("\n%d Each case Merge Node",stmt_no);
        curr_stmt->next = (struct stmt_list *) malloc(sizeof(struct stmt_list));
        curr_stmt = curr_stmt->next;
        curr_stmt->next = NULL;
        curr_stmt->what_stmt = 10;
        curr_stmt->start_stmt = $1;
        len = strlen($2)+5;
        curr_stmt->stmt = NULL,
        curr_stmt->st_no = stmt_no;
        stmt_no++;

/*      If there exits another switch statement within this switch statement
        the switch_var which stores the variable used  in this switch statement
        and the imaginary switch value given for forward referencing are
        restored */

        strcpy(switch_var,$3);
        switch_val = $5;
        cfg_graph->to = switch_val;
        cfg_graph->next = NULL;
        cfg_graph->next = (struct cfg *) malloc(sizeof(struct cfg));
        prev_cfg = cfg_graph;
        cfg_graph = cfg_graph->next;
        last_cfg = cfg_graph;
        cfg_graph->prev = prev_cfg;
        cfg_graph->from = $4;
        cfg_graph->to = stmt_no;
        cfg_graph->next = NULL;
        free($3);
};


st_val          :
{
        $$ = switch_val;
};


store_switch    :
{
        $$ = (char *) malloc(20);
        strcpy($$,switch_var);
};


case_val        :
CASE number COLON
{
        $$ = (char *) malloc(10);
        strcpy($$,$2);

/*      A node is created where the switch statement is stored */

        printf("\n%d %s == %s",stmt_no,switch_var,$$);
        curr_stmt->next = (struct stmt_list *) malloc(sizeof(struct stmt_list));
        curr_stmt = curr_stmt->next;
        curr_stmt->next = NULL;
        curr_stmt->what_stmt = 11;
        curr_stmt->start_stmt = -1;
        len = strlen($2) + strlen(switch_var) + 5;
        curr_stmt->stmt = (char *) malloc(sizeof(char) * len);
        strcpy(curr_stmt->stmt,switch_var);
        strcat(curr_stmt->stmt,"==");
        strcat(curr_stmt->stmt,$$);
        curr_stmt->st_no = stmt_no;

/*      Control is transferred to the next statement assuming that the
        condition is true.  Hence in the other grammar where this is called
        the FALSE condition is taken care of*/
```

```
        cfg_graph->next = (struct cfg *) malloc(sizeof(struct cfg));
        prev_cfg = cfg_graph;
        cfg_graph = cfg_graph->next;
        last_cfg = cfg_graph;
        cfg_graph->prev = prev_cfg;
        cfg_graph->from = stmt_no;
        cfg_graph->to = stmt_no+1;
        cfg_graph->next = NULL;
        control_t_st = stmt_no;
        stmt_no++;
};

%%
#include "lex.yy.c"
#include "procedures.h"
#include "print.h"
#include "ssa.h"
#include "gsa.h"

main()
{
        int flag_control = 0;
        struct cfg_n *root_cfg;
        struct ssa_var_stat *first_ssa_stat, *final_stat;
        struct ssa_var *fssa_var;
        split_varf = NULL;
        split_varl = NULL;

/*      As the input is scanned the control flow graph is built on a linked
        list */

        yyparse();
        printf("\n Variable List");

/*      The linked list is transferred to a tree type directed graph so that
        the other three conversions would be easier */

/*      find_con_edges(); */
        root_cfg = convert_graph();
        printf("\n\n\n Control Flow Graph \n\n\n");
        print_graph(root_cfg);

        init_var_stat(&first_ssa_stat);
        final_stat = final_ins_phi(root_cfg,flag_control,first_ssa_stat);

        flag_control = change_flag_val(flag_control);
        new_trav(root_cfg,flag_control,0);

/*      The control flow graph is converted to an SSA-form */

        flag_control = change_flag_val(flag_control);
        first_pass_phi(root_cfg,flag_control,NULL,final_stat);

        flag_control = change_flag_val(flag_control);
        print_ssa_graph(root_cfg,flag_control);

        flag_control = change_flag_val(flag_control);
        new_trav(root_cfg,flag_control,0);

/*      The SSA-form is converted to a GSA-form */

        flag_control = change_flag_val(flag_control);
        fssa_var = initialize_first_ssa();
        gen_ssa_form(root_cfg,flag_control,fssa_var,final_stat);

/*      The GSA-form is converted to PDW */

        init_var_stat(&first_ssa_stat);
        flag_control = change_flag_val(flag_control);
        final_ins_phi(root_cfg,flag_control,first_ssa_stat);

        free_ssa_stat(first_ssa_stat);

        printf("\n\n\n\n Static Single Assignment Form \n\n\n");
        flag_control = change_flag_val(flag_control);
        print_SSA_form(root_cfg,flag_control);

        flag_control = change_flag_val(flag_control);
        intro_gsa_form(root_cfg,flag_control);

        printf("\n\n\n\n Gated Single Assignment Form\n\n\n");
        flag_control = change_flag_val(flag_control);
```

```
        print_final_GSA(root_cfg,flag_control);


        printf("\n\n\n\n Program Dependence Web \n\n\n");
        flag_control = change_flag_val(flag_control);
        print_PDW(root_cfg,flag_control);
}
```

The following file is proc.h

```
void print_cfg_gra();
void pro_var_list();
int find_var_nam();
void update_list();
void strip_string();
void copy_var_list();
void update_var_list();
void find_con_edges();
struct cfg_n *search_cfg();
void print_graph();
void update_node();
struct cfg_n *insert_node();
struct cfg_n *convert_graph();
void print_cfg_graph();
void print_list_st();
void gen_ssa_form();
void print_ssa_graph();
void free_ssa_sta();
void update_write_stat();

/* ************************************************************************ */
/* ************************************************************************ */
/* ************************************************************************ */
/* ************************************************************************ */

int change_flag_val(number)
int number;
{
if (number == 0) return(1);
else return(0);
}


/* ************************************************************************ */
/* ************************************************************************ */
/*                                                                          */
/*      The following procedure has been used in yspec.c                    */
/*      PRO_VAR_LIST is used to produce a variable list that                */
/*      that could be used to produce a variable list used in               */
/*      the program                                                         */
/*      The string that has to be inserted is passed to this function       */
/*                                                                          */
/* ************************************************************************ */
/* ************************************************************************ */
void pro_var_list(l)
char *l;
{
        int i,j;

/*      first_var is a global variable that is use to store the first
        element of the linked list as defined by the declarations */

        if (first_var == NULL) {
/*      If the current variable is the first variable then initialize
        the list */
                var_list = (struct variable_n *) malloc(sizeof(struct variable_n));
                strcpy(var_list->var,l);
                var_list->no = -1;
                var_list->latest = -1;
                var_list->next = NULL;
                var_list->prev= NULL;
                var_list->ouccr_list = NULL;
                var_list->last_ouccr = NULL;
                first_var = var_list;
        }
        else {
/*      Insert into the linked list depending on the position of the
        variable in an increasing order */
/*      Based on the current pointer of the linked list and the variable to
        be inserted the pointer is moved forward or in the backward
        direction */

                if((j=strcmp(var_list->var,l)) >0) {
/*      The pointer is moved in the backward direction in other words the
        position of the variable is before the current pointer position
        in the linked list */

/*      Find the position of the string in the linked list */
```

```
                        while((j=(strcmp(var_list->var,l) > 0)) && (var_list->prev !=
NULL))
                                var_list = var_list->prev;
/*      If the string is not the first string then insert the string into
        the list */
                        if((var_list -> prev != NULL) || (j=(strcmp(var_list->var,l) < 0)))
{
                                var_tmp = (struct variable_n *) malloc(sizeof(struct
variable_n));
                                strcpy(var_tmp->var,l);
                                var_tmp->no = -1;
                                var_list->latest = -1;
                                var_tmp->next = var_list->next;
                                var_tmp->prev = var_list;
                                var_list->next = var_tmp;
                                var_tmp->ouccr_list = NULL;
                                var_tmp->last_ouccr = NULL;
                                if(var_tmp->next != NULL)
                                        var_tmp->next->prev = var_tmp;
                        }
                        else {
/*      The string to be inserted forms the first element in the list */
                                var_tmp = (struct variable_n *) malloc(sizeof(struct
variable_n));
                                strcpy(var_tmp->var,l);
                                var_tmp->no = -1;
                                var_list->latest = -1;
                                var_tmp->next = var_list;
                                var_tmp->prev = NULL;
                                var_tmp->ouccr_list = NULL;
                                var_tmp->last_ouccr = NULL;
                                first_var = var_tmp;
                                var_list->prev = var_tmp;
                        }
                }
                else {
/*      The variable has to be inserted after the current position in the
        linked list /

                        while((j=(strcmp(var_list->var,l) < 0)) && (var_list->next !=
NULL))
                                var_list = var_list->next;
/*      If the string is not the last string then insert the string into
        the list */
                        if((var_list ->next != NULL) || (j=(strcmp(var_list->var,l) > 0)))
{
                                var_tmp = (struct variable_n *) malloc(sizeof(struct
variable_n));
                                strcpy(var_tmp->var,l);
                                var_tmp->no = -1;
                                var_list->latest = -1;
                                var_tmp->next = var_list;
                                var_tmp->prev = var_list->prev;
                                var_tmp->ouccr_list = NULL;
                                var_tmp->last_ouccr = NULL;
                                var_list->prev= var_tmp;
                                if(var_tmp->prev!= NULL)
                                        var_tmp->prev->next = var_tmp;
                        }
                        else {
/*      Insert the string as the last element in the linked list */
                                var_tmp = (struct variable_n *) malloc(sizeof(struct
variable_n));
                                strcpy(var_tmp->var,l);
                                var_tmp->no = -1;
                                var_list->latest = -1;
                                var_tmp->next = NULL;
                                var_tmp->prev = var_list;
                                var_tmp->ouccr_list = NULL;
                                var_tmp->last_ouccr = NULL;
                                var_list->next = var_tmp;
                        }
                }
        }
}

/* ********************************************************************** */
/* ********************************************************************** */
/*      FIND_VAR_NAM finds the name for the variable depending
        on the information in the linked list

        This function returns 0 if the variable has not been defined
```

```
        in the declaration section
                                                                        */
/* ************************************************************************ */
/* ************************************************************************ */
int find_var_nam(varn,flag)
char *varn;
int flag;
{
        int j,num;
        char number[5];
/*      Find the string or the variable in the linked list */

        if ((j=strcmp(var_list->var,varn)) >0) {
                while((j = strcmp(var_list->var,varn) > 0) && (var_list ->prev != NULL))
                        var_list = var_list->prev;
        }
        else {
                while((j = strcmp(var_list->var,varn) < 0) && (var_list->next !=

                    NULL))
                        var_list = var_list->next;
        }

        if ((j=strcmp(var_list->var,varn)) == 0) {
/*      If the variable is fund in the linked list */
                num = var_list->no;
                if(flag == NEWNAME){
/*      Depending on the flag a new name for the variable is to be assigned */
                        if (var_list->no == -1) {
/*      If this is the first time the variable is accessed or the variable
        is initialized then the no field is -1.  Therefore the var_list
        is updated to the new number */
                                var_list->no = 2;
                                var_list->latest = 2;
                                num = 2;
                        }
                        else {
/*      If the variable has already been accessed then the latest is updated
        by one.  Or in other words a new number is generated which is
        an increment by the old number */

                                var_list->latest = var_list->latest+1;
                                num = var_list->latest;
                        }
                }
/*      if flag == OLDNAME or in other words if the variable is referenced
        for the first time then the value is one */
                if (var_list->no == -1){
/*      If this is the first time the variable is accessed or the variable
        is initialized then the no field is -1.  Therefore the var_list
        is updated to the new number */
                        var_list->no = 1;
                        var_list->latest = 1;
                        num = 1;
                }
                strcat(varn,"##");
                sprintf(number,"%d",num);
                strcat(varn,number);
                return(1);
        }
        else {
                printf("\n %s : Variable Not Defined",varn);
                return(0);
        }
}

/* ************************************************************************ */
/* ************************************************************************ */
/*
        UPDATE_LIST - Updates the list of variables in the linked list

                The no and latest are two fields in the structure
                that keep track of the variables referenced and the
                variables initialized.

*/
/* ************************************************************************ */
/* ************************************************************************ */
void update_list()
{
        struct variable_n *tmp_list;
        tmp_list = first_var;
```

```
        while(tmp_list != NULL) {
                tmp_list->no = tmp_list->latest;
                tmp_list = tmp_list->next;
        }
}

/* ************************************************************************ */
/* ************************************************************************ */
/*

        STRIP_STRING () --  The string contain a statement in which
                variables have been given some name is passed on to this
                procedure.
*/
/* ************************************************************************ */
/* ************************************************************************ */


void strip_string(str1)
char *str1;
{
        int i;
        char tmp[40];
        char st[50];
        strcpy(tmp,strstr(str1,"="));
        i = 0;
        while(*(str1+i) != '#') {
                st[i] = *(str1+i);
                i++;
        }
        st[i] = '\0';
        find_var_nam(st,NEWNAME);

/*      For the variable in the left hand side (or) the variable assigned a
        value in the left hand side a new variable is assigned.
        The new variable is of the form variable##number */

/*      find_var_nam - Finds the new name for the variable passed as the
                first parameter.  On return the node or the value of
                the var_list (a linked list maintain the last name
                given to each variable) points or contains the
                information of the variable passed */

/*      Here the var_list for the variable for which a new variable name
        is given is Updated */

        tmp_var_ouccr = (struct var_ouccr *) malloc(sizeof(struct var_ouccr));
        tmp_var_ouccr->st_num = stmt_no;
        tmp_var_ouccr->next = NULL;

/*      if the variable had not been referenced before then
        var_list->ouccr_list is NULL
        If it is NULL the information has to be inserted else
        the last_ouccr is updated
*/

        if (var_list->ouccr_list == NULL){
                var_list->last_ouccr = tmp_var_ouccr;
                var_list->ouccr_list = tmp_var_ouccr;
        }
        else {
                var_list->last_ouccr = tmp_var_ouccr;
        }
        strcpy(str1,st);
        strcat(str1,tmp);
}

/* ************************************************************************ */
/* ************************************************************************ */
/*
        COPY_VAR_LIST()  - Copies the variable list into a temporary list
                This is used in the parsing of the input C program
*/
/* ************************************************************************ */
/* ************************************************************************ */
void copy_var_list()
{
        struct variable_n *tmp_first,*tmp_list, *cp_list;

        cp_list = first_var;
```

```
        tmp_first = (struct variable_n *) malloc(sizeof(struct variable_n));

        tmp_list = tmp_first;

        tmp_list->prev = NULL;
        tmp_list->no = cp_list->no;
        tmp_list->latest = cp_list->latest;
        strcpy(tmp_list->var,cp_list->var);
        tmp_list->last_ouccr = cp_list->last_ouccr;
        tmp_list->ouccr_list = cp_list->ouccr_list;
        tmp_list->next = NULL;


        cp_list = cp_list->next;
        while(cp_list != NULL) {
                tmp_list->next=(struct variable_n *) malloc(sizeof(struct variable_n));
                tmp_list->next->prev = tmp_list;
                tmp_list = tmp_list->next;
                tmp_list->next = NULL;

                tmp_list->no = cp_list->no;
                tmp_list->latest = cp_list->latest;
                strcpy(tmp_list->var,cp_list->var);
                tmp_list->last_ouccr = cp_list->last_ouccr;
                tmp_list->ouccr_list = cp_list->ouccr_list;

                cp_list = cp_list->next;
        }
        first_var = tmp_first;
        var_list = tmp_first;
}

/* ************************************************************************ */
/* ************************************************************************ */
/*
        UPDATE_VAR_LIST () is used to update the latest in the
                variable_n structure.
                In the linked list containing the variable information
                the next variable name needs to be updated.
*/
/* ************************************************************************ */
/* ************************************************************************ */
void update_var_list()
{
        struct variable_n *free_var, *tmp_list;

        var_list = first_var;
        tmp_list = nfirst_var;

        while(var_list != NULL) {
                if(var_list->latest > tmp_list->latest){
                        tmp_list->last_ouccr = var_list->last_ouccr;
                        tmp_list->latest = var_list->latest;
                }
                free_var = var_list;
                var_list = var_list->next;
                tmp_list = tmp_list->next;
                free_var->last_ouccr = NULL;
                free_var->ouccr_list = NULL;
                free(free_var);
        }
        first_var = nfirst_var;
        var_list = first_var;
}

/* ************************************************************************ */
/* ************************************************************************ */
/*
        FIND_CON_EDGES () -->  This procedure goes through the linked list
                and creates the CFG like the tree structure
                on a new structure named ssa_intro which is an array
                of characters

*/
/* ************************************************************************ */
/* ************************************************************************ */
void find_con_edges()
{
        struct cfg  *cfg_list;
        struct var_ouccr *tmp_ouccr;

        int i,j=0,k;
```

```
        struct ssa_intro *ssa_st;
        ssa_st = (struct ssa_intro *) malloc((stmt_no+2) * sizeof(struct ssa_intro));
        for(i=0;i<stmt_no+2;i++){
                ssa_st[i].no=0;
                ssa_st[i].l_ouccr = NULL;
        }

        cfg_list = first_cfg->next;

        while(cfg_list != NULL){
                k = cfg_list->to;
                ssa_st[k].no++;

                tmp_ouccr=(struct var_ouccr *)malloc(sizeof(struct var_ouccr));
                tmp_ouccr->st_num = cfg_list->from;
                tmp_ouccr->next = NULL;
                if (ssa_st[k].l_ouccr == NULL){
                        ssa_st[k].l_ouccr = tmp_ouccr;
                        ssa_st[k].f_ouccr = tmp_ouccr;
                }
                else {
                        ssa_st[k].l_ouccr->next = tmp_ouccr;
                        ssa_st[k].l_ouccr = tmp_ouccr;
                }
                cfg_list = cfg_list->next;
        }
}

/* ********************************************************************** */
/* ********************************************************************** */
/*

        CONVERT_GRAPH() --> This procedure converts the linked list
                to the tree structured directed graph.
                The procedure uses cfg lined list to produce
                a tree structured directed graph cfg_n.

*/
/* ********************************************************************** */
/* ********************************************************************** */

struct cfg_n * convert_graph()
{
        struct cfg *tmp_cfg, *cfg_list, *tmp1;

        struct cfg_n *root, *node_cfg, *tmp2, *tmp3;

/*      Store the first element of the linked list in the tmp_cfg */

        tmp_cfg = first_cfg->next;

/*      The root node of the tree is allocated */

        root = (struct cfg_n *) malloc(sizeof(struct cfg_n));
        root->st_no = tmp_cfg->from;
        root->flag = 0;
        root->no_in_edges = 0;
        root->left = NULL;
        root->right = NULL;

/*      For every from and to in a node of the linked list the pointer
        in the tree structure is built. */

/*      One of the rules is that for control cannot go to more than two
        directions after the execution of a statement */

        node_cfg = (struct cfg_n *) malloc(sizeof(struct cfg_n));
        node_cfg->flag = 0;
        node_cfg->no_in_edges = 0;
        node_cfg->st_no = tmp_cfg->to;
        node_cfg->left = NULL;
        node_cfg->right = NULL;

        root->left = node_cfg;
/*      After the first statement the pointer is made to point to the left */

/*      If the program does not have an CONDITION all right pointers would
        be assigned a NULL values */

        node_cfg->no_in_edges++;
        tmp_cfg = tmp_cfg->next;
        while (tmp_cfg != NULL) {
```

```
/*      If the one end of the link present in the linked list is found
        in the newly formed node in the flow graph insert the other node in
        the graph thereby creating the link */

                if (tmp_cfg->from == node_cfg->st_no)
                        node_cfg = insert_node(root,node_cfg,tmp_cfg);
                else {

/*      Find the suitable position to insert the node or make the link
        in the newly formed CFG */

                        if (tmp_cfg->from > node_cfg->st_no) {
                                tmp1 = tmp_cfg;
/*      Search the linked list for the link between the newly created node in
        the new CFG and the next node */

                                while( (tmp_cfg->from != node_cfg->st_no) && (tmp_cfg !=
NULL));
                                        tmp_cfg = tmp_cfg->next;

/*      If found then insert or make the link in the newly formed CFG
        and insert the node */

        if ((tmp_cfg != NULL)&&(tmp_cfg->from==node_cfg->st_no))
                                        node_cfg=insert_node(root,node_cfg,tmp_cfg);
                                else {

/*      If there exists no node in the linked list or if the link could not
        be made then the newly formed CFG is searched for the node in the
        CFG tree */

                                        tmp_cfg = tmp1;
                                        tmp2 = search_cfg(tmp_cfg->from,root);
                                        if (tmp2 != NULL) {

/*      If the from element or the from statement number is found in the tree
        the to statement number is searched in the tree */

                                                tmp3 = search_cfg(tmp_cfg->to,root);
                                                if (tmp3 == NULL) {
/*      If the to element is not found in the tree the to node is created
        and the link is made */
                                                        tmp3= (struct cfg_n *)
malloc(sizeof(struct cfg_n));
                                                        tmp3->no_in_edges = 0;
                                                        tmp3->flag = 0;
                                                        tmp3->st_no = tmp_cfg->to;
                                                        tmp3->left = NULL;
                                                        tmp3->right = NULL;
                                                }
/*      If the to element is also found in the tree the the link between
        the from element and the to element is made depending on the
        existence of the left pointer of the tree */

                                                if (tmp2->left == NULL) {
                                                        tmp2->left = tmp3;
                                                        tmp3->no_in_edges++;
                                                }
                                                else if (tmp2->right == NULL){
                                                        tmp2->right=tmp3;
                                                        tmp3->no_in_edges++;
                                                }
                                                else {
                                                        if ((tmp2->left->st_no !=
tmp3->st_no) && (tmp2->right->st_no != tmp3->st_no)){
                                                                printf("\n  Pointer Error:
Control Error");
                                                                exit(0);
                                                        }
                                                }
                                        }
/*      if tmp2 is also NULL it means that such a to statement number and
        a from statement number is yet to be in the CFG tree.
        So a connection between the an existing node and the from node
        of the linked list is to be found */
/*      find_un_edge --> performs the above function */
                                        else {
                                                tmp1 = tmp_cfg;
                                                find_un_edge(tmp_cfg,root,node_cfg);
                                                tmp1 = tmp1->prev;
                                        }
                                }
```

```
                                tmp_cfg = tmp1;
                        }
                        else {
/*      This condition exists if the node is already present in the tree */

/*      The tree is searched for the current from node */
                                tmp2 = search_cfg(tmp_cfg->from,root);
                                if (tmp2 != NULL) {
/*      The tree is searched for the current to node */
                                        tmp3 = search_cfg(tmp_cfg->to,root);
                                        if (tmp3 == NULL) {
/*      If to node not found the to node is created and the link is established
        */
                                                tmp3= (struct cfg_n *) malloc(sizeof(struct
cfg_n));
                                                tmp3->no_in_edges = 0;
                                                tmp3->flag = 0;
                                                tmp3->st_no = tmp_cfg->to;
                                                tmp3->left = NULL;
                                                tmp3->right = NULL;
                                        }
/*      If the to node has also been found depending on whether the connection
        has already been established or not, the new connection is established
        */
                                        if (tmp2->left == NULL) {
                                                tmp2->left = tmp3;
                                                tmp3->no_in_edges++;
                                        }
                                        else if (tmp2->right == NULL){
                                                tmp2->right=tmp3;
                                                tmp3->no_in_edges++;
                                        }
                                        else {
                                                if ((tmp2->left->st_no != tmp3->st_no) &&
(tmp2->right->st_no != tmp3->st_no)){
                                                        printf("\n  Pointer Error: Control
Error");
                                                        exit(0);
                                                }
                                        }
                                }
                                else {
/*      if both the nodes are not in the tree a new link is established by the
        find_un_edge so that from that node this link could be formed */
                                        tmp1 = tmp_cfg;
                                        find_un_edge(tmp_cfg,root,node_cfg);
                                        tmp1 = tmp1->prev;
                                        tmp_cfg = tmp1;
                                }
                        }
                }
                tmp_cfg = tmp_cfg->next;
        }
        return(root);
}

/* ********************************************************************** */
/* ********************************************************************** */
/*
        INSERT_NODE() --> Inserts a node into the CFG tree and establishes
                a link in the tree.
*/
/* ********************************************************************** */
/* ********************************************************************** */
struct cfg_n  *insert_node(root, cfg_node,tmp_cfg)
struct cfg_n   *root, *cfg_node;
struct cfg     *tmp_cfg;
{
        struct cfg_n   *tmp;

/*      Searches the tree, newly formed CFG tree, for the statement number
        to be equal to the to statement number */

        tmp = search_cfg(tmp_cfg->to,root);

        if (tmp == NULL) {
/*      if the CFG tree does not have the statement number then a
        node is formed and the link is established */
                tmp= (struct cfg_n *) malloc(sizeof(struct cfg_n));
                tmp->no_in_edges = 0;
                tmp->flag = 0;
                tmp->st_no = tmp_cfg->to;
```

```
                tmp->left = NULL;
                tmp->right = NULL;
                if (cfg_node->left == NULL) {
                        cfg_node->left = tmp;
                        tmp->no_in_edges++;
                }
                else if (cfg_node->right == NULL) {
                        cfg_node->right = tmp;
                        tmp->no_in_edges++;
                }
                else {
                        if ((cfg_node->left->st_no != tmp->st_no) &&
(cfg_node->right->st_no != tmp->st_no)){
                                printf("\n  Pointer Error: Control Error");
                                exit(0);
                        }
                }
                return(tmp);
        }
        else {
/*      If the link has already been established then the establishment of
        the new link is forgotten */
                if ((cfg_node->left == tmp) || (cfg_node->right == tmp))
                        return(NULL);
/*      if the node already exists then the link is established */
                else {
                        if (cfg_node->left == NULL) {
                                cfg_node->left = tmp;
                                tmp->no_in_edges++;
                        }
                        else if (cfg_node->right == NULL) {
                                cfg_node->right = tmp;
                                tmp->no_in_edges++;
                        }
                        else {
                                if ((cfg_node->left->st_no != tmp->st_no) &&
(cfg_node->right->st_no != tmp->st_no)){
                                        printf("\n  Pointer Error: Control Error");
                                        exit(0);
                                }
                        }
                }
                return(tmp);
        }
}

/* ********************************************************************** */
/* ********************************************************************** */
/*
        SEARCH_CFG() --> searches the CFG tree structured directed graph
                for the statement whose number is similar to the one
                passed on to this procedure
*/
/* ********************************************************************** */
/* ********************************************************************** */

struct cfg_n *search_cfg(st_number, node_cfg)
int st_number;
struct cfg_n *node_cfg;
{
        struct cfg_n *tmpc;
        if (node_cfg == NULL) return(NULL);
        if (node_cfg->st_no == st_number) {
                return(node_cfg);
        }
        else{
                tmpc = NULL;
                if (node_cfg->left != NULL){
                        if(node_cfg->left->st_no > node_cfg->st_no)
                                tmpc = search_cfg(st_number,node_cfg->left);
                }
                if(tmpc == NULL){
                        if (node_cfg->left != NULL){
                                if(node_cfg->left->st_no > node_cfg->st_no)
                                        tmpc = search_cfg(st_number,node_cfg->right);
                        }
                }
                else {
                        return(tmpc);
                }
        }
        return(tmpc);
```

```
}


/* ********************************************************************** */
/* ********************************************************************** */
/*
        NEW_TRAV () --> is used to free the structure inside the CFG
                node of the tree.
*/
/* ********************************************************************** */
/* ********************************************************************** */
void new_trav(node_cfg,flag_val,flag)
struct cfg_n *node_cfg;
int flag_val,flag;
{
        int froml=0,tol=0,fromr=0,tor=0;
        if (node_cfg == NULL) return;
        if (node_cfg->left != NULL) {
                froml = node_cfg->st_no;
                tol = node_cfg->left->st_no;
                if (node_cfg->flag != flag_val){
                        /*      Free the structure containing the SSA
                                introducing variables */
                        if (flag == 0) free_f_list(node_cfg);
                        else {
                                printf("\n %d ----->  %d ",froml,tol);
                                printf("  **** %d ",node_cfg->no_in_edges);
                                printf(" %d %s",node_cfg->st_no,node_cfg->stmt);
                                print_f_phi(node_cfg);
                                free_f_list(node_cfg);
                                }
                        }
        }
        node_cfg->flag = flag_val;
        node_cfg->flag_end = flag_val;
        if (froml <= tol)
                new_trav(node_cfg->left,flag_val,flag);
        if (node_cfg->right != NULL) {
                fromr = node_cfg->st_no;
                tor = node_cfg->right->st_no;
                if (node_cfg->flag != flag_val){
                        /*      Free the structure containing the SSA
                                introducing variables */
                        if (flag == 0) free_f_list(node_cfg);
                        else {
                                printf("\n %d ----->  %d ",fromr,tor);
                                printf("  **** %d ",node_cfg->no_in_edges);
                                printf(" %d %s",node_cfg->st_no,node_cfg->stmt);
                                print_f_phi(node_cfg);
                                free_f_list(node_cfg);
                                }
                        }
        }
        if (fromr <= tor)
                new_trav(node_cfg->right,flag_val,flag);
}


/* ********************************************************************** */
/* ********************************************************************** */
/*
        UPDATE_NODE () --> is used to initialize the node in the CFG
                        with additional information of the node.  Each node
                        is initialized completely.

*/
/* ********************************************************************** */
/* ********************************************************************** */
void update_node(node_cfg)
struct cfg_n *node_cfg;
{
        struct stmt_list *tmpstmt;

        tmpstmt = first_stmt;
        while((tmpstmt != NULL) && (tmpstmt->st_no != node_cfg->st_no))
                tmpstmt = tmpstmt->next;
        node_cfg->stmt = tmpstmt->stmt;
        node_cfg->what_stmt = tmpstmt->what_stmt;
        node_cfg->start_stmt = tmpstmt->start_stmt;
/*      The SSA list or the linked list used in defining the linked list
        is initialized */
```

```
        node_cfg->f_list = NULL;
        node_cfg->l_list = NULL;
        node_cfg->f_phi = NULL;
        node_cfg->l_phi = NULL;
        node_cfg->sec_list = NULL;
        node_cfg->f_int = NULL;
}

/* ********************************************************************** */
/* ********************************************************************** */
/*
        FIND_UN_EDGE()  -->  This procedure plays an important role
                in the creation of the tree structured CFG.

                In a linked list there may be a node where the link
                in that node could not be established because of
                the statements may not be present in the newly formed
                CFG tree.
                In such a case this procedure is called which establishes
                the link by tracing the link to the nodes present in this
                node of the linked list.
*/
/* ********************************************************************** */
/* ********************************************************************** */
find_un_edge(tmp_cfg, root, node_cfg)
struct cfg *tmp_cfg;
struct cfg_n *root,*node_cfg;
{
        struct cfg *temper1;
        struct cfg_n *tmp2, *tmp3;
        int tmp_to, tmp_from;

        temper1 = tmp_cfg;
        tmp_from = tmp_cfg->from;
        tmp2 = NULL;
        do {
/*      Search the linked list to identify a TO node which is similar to
        the from statement number of the current linked list for which
        the link could not be established */
                while((tmp_from != tmp_cfg->to) && (tmp_cfg != NULL))
                        tmp_cfg = tmp_cfg->next;

/*      Once the link between this unestablished link and the possible
        link to this unestablished link is identified,  the possible link
        is tried to be established */

/*      The new node in the list which could give the link to this
        unestablished link is searched in the CFG */
                tmp2 = search_cfg(tmp_cfg->from,root);
                if (tmp2 != NULL) {
/*      If the first node of the possible link is in the CFG, then to
        statement number is also searched in the CFG to double check */

                        tmp3 = search_cfg(tmp_cfg->to,root);
                        if (tmp3 == NULL) {
/*      If not found the link is established */
                                tmp3= (struct cfg_n *) malloc(sizeof(struct cfg_n));
                                tmp3->no_in_edges = 0;
                                tmp3->flag = 0;
                                tmp3->st_no = tmp_cfg->to;
                                tmp3->left = NULL;
                                tmp3->right = NULL;
                        }
                        if (tmp2->left == NULL) tmp2->left = tmp3;
                        else if (tmp2->right == NULL) tmp2->right=tmp3;
                        else {
                                if ((tmp2->left->st_no != tmp3->st_no) &&
(tmp2->right->st_no != tmp3->st_no)){
                                        printf("\n  Pointer Error: Control Error");
                                        exit(0);
                                }
                        }
                }
        }while((tmp2 == NULL) && (tmp_cfg != NULL));
}

/* ********************************************************************** */
/* ********************************************************************** */
/*
        UPDATE_F_SIGHS () -->
*/
/* ********************************************************************** */
```

```
/* ********************************************************************** */
update_f_phis(node_cfg,fssa_var,f_stat,flag_val)
struct cfg_n *node_cfg;
struct ssa_var **fssa_var;
struct ssa_var_stat *f_stat;
int flag_val;
{
struct ssa_var *tvar;
char str1[200],*st,str[10];
int i,j,len,number, what_st, leng;
struct ssa_var *f_var, *l_var;
struct phi_var_list *tphi;
struct ssa_var_stat  *tmpstat;

f_var = *fssa_var;
tphi = node_cfg->f_phi;
if (tphi == NULL) return;
while (tphi != NULL) {
        st = tphi->sivar;
        number = tphi->no;
        tvar = f_var;
        while ((tvar != NULL) && ((j=strcmp(tvar->var,st)) != 0)){
                l_var = tvar;
                tvar = tvar->next;
                }
        if (tvar == NULL) {
                l_var->next = (struct ssa_var *) malloc(sizeof(struct ssa_var));
                l_var = l_var->next;
                l_var->next = NULL;
                strcpy(l_var->var,st);
                l_var->no = number;
                }
        if ((j=strcmp(tvar->var,st)) == 0) {
                        if (node_cfg->flag != flag_val) {
/*      For the variable for which the phi function is introduced
        a new variable is introduced for that phi function */
                                tmpstat = f_stat;
                                while((tmpstat != NULL) && ((j=strcmp(tmpstat->var,st)) !=
0))
                                                        tmpstat = tmpstat->next;
/*      The write status of the variable in the list that maintains
        the status is also updated */
                                if (tmpstat != NULL) {
                                        number = tmpstat->write_no;
                                        tmpstat->write_no++;
                                        tphi->no = number;
                                        }
                                }
                        tvar->no = number;
                        }
        tphi = tphi->next;
        }
}


/* ********************************************************************** */
/* ********************************************************************** */
/*
        UPDATE_SSA_VAR () ---> Updates the status of the linked list
                that maintains the status of the linked list

*/
/* ********************************************************************** */
/* ********************************************************************** */
update_ssa_var(fssa_var,node_cfg)
struct ssa_var **fssa_var;
struct cfg_n *node_cfg;
{
        struct ssa_var *tvar;
        char str1[200],st[100],str[10];
        int i,j,len,number, what_st, leng;
        struct ssa_var *f_var, *l_var;

        if(node_cfg == NULL) return;
        if (node_cfg->stmt == NULL) return;
/*      If the current node is NULL then there are no variables in the current
        statement */
        f_var = *fssa_var;

        strcpy(str1,node_cfg->stmt);
        len = strlen(str1);
        what_st = node_cfg->what_stmt;
```

```
        i = 0;
        leng = len;
        do {
                j = 0;
/*      Separate the variable and the number of the variable */
                while(str1[i] != '#') {
                        st[j] = str1[i];
                        i++;
                        j++;
                        }
                st[j] = '\0';
                j = 0;
                i = i+2;
                len = strlen(str1);
                while((i<len) && (isdigit(str1[i]))) {
                        str[j] = str1[i];
                        i++;
                        j++;
                }
                i = i+1;
                str[j] = '\0';
                number = atoi(str);
/*      If the variable status list has not been formed then initialize it */
                if (f_var == NULL) {
                        l_var=(struct ssa_var *) malloc(sizeof(struct ssa_var));
                        l_var->next = NULL;
                        strcpy(l_var->var,st);
                        l_var->no = number;
                        *fssa_var = l_var;
                        f_var = l_var;
                        if (what_st != 2)
                                return;
                }
                tvar = f_var;
                while ((tvar != NULL) && ((j=strcmp(tvar->var,st)) != 0)){
                        l_var = tvar;
                        tvar = tvar->next;
                }
                if (tvar == NULL) {
                        l_var->next = (struct ssa_var *) malloc(sizeof(struct ssa_var));
                        l_var = l_var->next;
                        l_var->next = NULL;
                        strcpy(l_var->var,st);
                        l_var->no = number;
                }
/*      If variable already present in the list update it */
                if ((j=strcmp(tvar->var,st)) == 0) {
                        tvar->no = number;

                }
        }while((what_st == 2) && (i < leng));
}


/* ********************************************************************** */
/* ********************************************************************** */
/*
        COPY_LIST_SSA() -> Copies a list of the status variable list
                to preserve one copy for later use
*/
/* ********************************************************************** */
/* ********************************************************************** */
copy_list_ssa(fssa1,fssa2)
struct ssa_var *fssa1,**fssa2;
{
        struct ssa_var *tmps,*ssa_tmp;
        if (fssa1 == NULL) {
                *fssa2 = fssa1;
                return;
        }
        tmps = (struct ssa_var *) malloc(sizeof(struct ssa_var));
        strcpy(tmps->var,fssa1->var);
        tmps->no = fssa1->no;
        *fssa2 = tmps;
        tmps->next = NULL;
        ssa_tmp = fssa1->next;
        while(ssa_tmp != NULL) {
                tmps->next = (struct ssa_var *) malloc(sizeof(struct ssa_var));
                tmps = tmps->next;
                strcpy(tmps->var,ssa_tmp->var);
                tmps->no = ssa_tmp->no;
```

```
                tmps->next = NULL;
                ssa_tmp= ssa_tmp->next;
        }
}


/* ********************************************************************** */
/* ********************************************************************** */
/*
        UPDATE_SIGH_L () -->  This procedure is used to update the phi
                variable list in the second pass.  As it could be noted
                there exists two status variable lists in the transformation
                of the input C program to the SSA-form.
                Comparing the two status of variable lists the variables
                are updated in the variable list.
*/
/* ********************************************************************** */
/* ********************************************************************** */
update_phi_l(node_g,it_ssa,f_stat)
struct cfg_n *node_g;
struct ssa_var *it_ssa;
struct ssa_var_stat *f_stat;
{
        int j,down_flag;
        struct phi_var_list *t_phi_var;
        struct ssa_var *t_ssa,*fssa_var, *travel_list, *prev_node;
        char *tmpvar;
        struct ssa_int_occr *t_intro;
        struct ssa_var_stat *tmp_stat;

        t_ssa = it_ssa;

        fssa_var = node_g->f_list->ssa_list;
/*      Keeping one of the status of the information standard the
        other list is searched for the current variable */
        while(t_ssa != NULL) {
                travel_list = fssa_var;
                while((travel_list != NULL) && (j=strcmp(travel_list->var,t_ssa->var)
!=0)){
                        prev_node = travel_list;
                        travel_list = travel_list->next;
                }
                if(travel_list == NULL) {
                        prev_node->next = (struct ssa_var*) malloc(sizeof(struct ssa_var));
                        prev_node = prev_node->next;
                        prev_node->next = NULL;
                        strcpy(prev_node->var,t_ssa->var);
                        prev_node->no = t_ssa->no;
                }
                else {
                        if ((travel_list->no < t_ssa->no) && (node_g->no_in_edges > 1)){
/*      If the status of the variables differs and the number does not
        match a new SIGH variable has to be introduced for the current
        variable */

                                tmpvar = travel_list->var;
                                travel_list->no = t_ssa->no;
/*      The phi variable is appended to the list of phi variables */
                                if (node_g->f_phi == NULL) {
/*      If the phi variable list is empty then the phi variable list
        is initialized */
                                        node_g->f_phi = (struct phi_var_list *)
malloc(sizeof(struct phi_var_list));
                                        strcpy(node_g->f_phi->sivar,tmpvar);
                                        tmp_stat = f_stat;
                                        while((tmp_stat != NULL) &&
((j=strcmp(tmpvar,tmp_stat->var)) != 0))
                                                tmp_stat=tmp_stat->next;
                                        node_g->f_phi->no = tmp_stat->write_no;
                                        tmp_stat->write_no++;
                                        node_g->f_phi->next = NULL;
                                        node_g->f_phi->f_num = NULL;
                                        node_g->f_phi->l_num = NULL;
                                        node_g->l_phi = node_g->f_phi;
                                }
                                else {
/*      If the phi variable list is already present the phi variable
        is introduced at the end of the list */
                                        t_phi_var = node_g->f_phi;
                                        while((t_phi_var != NULL) &&
(j=strcmp(t_phi_var->sivar,tmpvar) != 0)) {
                                                t_phi_var = t_phi_var->next;
```

```
                                        }
                                        if (t_phi_var == NULL) {
                                                node_g->l_phi->next = (struct phi_var_list *)
malloc(sizeof(struct phi_var_list));
                                                node_g->l_phi=node_g->l_phi->next;

strcpy(node_g->l_phi->sivar,travel_list->var);
                                                tmp_stat = f_stat;
                                                while((tmp_stat != NULL) &&
((j=strcmp(tmpvar,tmp_stat->var)) != 0))
                                                        tmp_stat=tmp_stat->next;
                                                node_g->l_phi->no = tmp_stat->write_no;
                                                tmp_stat->write_no++;
                                                node_g->l_phi->next = NULL;
                                                node_g->l_phi->f_num = NULL;
                                                node_g->f_phi->l_num = NULL;
                                        }
                                }
                        }
                }
                t_ssa = t_ssa->next;
        }
}


/* ********************************************************************** */
/* ********************************************************************** */
/*
        COPY_VAR_STAT --> The status of the variable list is copied on
                to another position as temporary storage
                This is used in the second phase of the translation
                of a C program to the SSA-form
*/
/* ********************************************************************** */
/* ********************************************************************** */
copy_var_stat(f_stat,ret_stat)
struct ssa_var_stat *f_stat, **ret_stat;
{
        struct ssa_var_stat *tmpstat, *tst;
        tmpstat = f_stat;
        if (f_stat == NULL) {
                *ret_stat = NULL;
                return;
        }
        tst = (struct ssa_var_stat *)malloc(sizeof(struct ssa_var_stat));
        tst->read_no = tmpstat->read_no;
        tst->write_no = tmpstat->write_no;
        tst->stmt_no = tmpstat->stmt_no;
        tst->wrst_no = tst->wrst_no;
        tst->next = NULL;
        *ret_stat = tst;
        strcpy(tst->var,tmpstat->var);
        tmpstat = tmpstat->next;
        while(tmpstat != NULL) {
                tst->next = (struct ssa_var_stat *) malloc(sizeof(struct ssa_var_stat));
                tst = tst->next;
                tst->read_no = tmpstat->read_no;
                tst->write_no = tmpstat->write_no;
                tst->stmt_no = tmpstat->stmt_no;
                tst->wrst_no = tst->wrst_no;
                tst->next = NULL;
                strcpy(tst->var,tmpstat->var);
                tmpstat = tmpstat->next;
        }
}

/* ********************************************************************** */
/* ********************************************************************** */
/*
        UPDATE_WRITE_STAT () --> This procedure updates the write information
                in the maintenance of the status of each variable in the
                status list
*/
/* ********************************************************************** */
/* ********************************************************************** */
void update_write_stat(fvar_stat,svar_stat)
struct ssa_var_stat *fvar_stat, *svar_stat;
{
        while (fvar_stat != NULL) {
                if (fvar_stat->write_no < svar_stat->write_no){
                        fvar_stat->write_no = svar_stat->write_no;
```

```
                        fvar_stat->wrst_no = svar_stat->wrst_no;
                }
                fvar_stat = fvar_stat->next;
                svar_stat = svar_stat->next;
        }
}

/* ********************************************************************** */
/* ********************************************************************** */
/*
        UPDATE_READ_STAT () --> This procedure updates the read information
                of the list that maintains the status of the variables
                at the end of each node
*/
/* ********************************************************************** */
/* ********************************************************************** */
update_read_stat(fstat,str)
struct ssa_var_stat *fstat;
char *str;
{
        struct ssa_var_stat *fvar_stat;
        int j;
        fvar_stat = fstat;
        while((fvar_stat != NULL) && ((j=strcmp(fvar_stat->var,str)) != 0))
                                        fvar_stat = fvar_stat->next;
        if (fvar_stat == NULL) return;
        if (fvar_stat->read_no < fvar_stat->write_no){
                        fvar_stat->read_no = fvar_stat->write_no-1;
                        fvar_stat->stmt_no = fvar_stat->wrst_no;
                        }
}


/* ********************************************************************** */
/* ********************************************************************** */
/*
        FINAL_INS_SIGH() -->  This procedure is used to initialize the
                variables in the program with a new variable name
                corresponding to each variable.
                Each variable of the form VAR is assigned a new name
                VAR##NUM where Num is the number assigned to the
                variable each time it is referenced in the program
                This procedure also initialize the Sigh variables.
*/
/* ********************************************************************** */
/* ********************************************************************** */
struct ssa_var_stat *final_ins_phi(node_cfg,flag_val,var_stat)
struct cfg_n *node_cfg;
int flag_val;
struct ssa_var_stat *var_stat;
{
        struct cfg_n *lnode,*rnode;
        struct ssa_var_stat *rec_var_stat, *fvar_stat, *ssa_stat;
        struct ssa_var_stat *ssa_r_stat, *ssa_l_stat;
        struct cfg_n *lenode, *rinode;
/*      var_stat --> maintains the status of the variables */
        if (node_cfg == NULL) return(NULL);
        if (node_cfg->flag == flag_val) return(NULL);
/*      If this node has already been visited then return */

        replace_var_nvar(node_cfg,var_stat);
        lenode = node_cfg->left;
        rinode = node_cfg->right;
        if (lenode != NULL) {
                if (lenode->f_phi != NULL) {
                        update_in_phi(lenode,var_stat);
                }
        }
        if (rinode != NULL) {
                if (rinode->f_phi != NULL) {
                        update_in_phi(rinode,var_stat);
                }
        }
        node_cfg->flag = flag_val;
        copy_var_stat(var_stat,&rec_var_stat);
        ssa_l_stat = final_ins_phi(node_cfg->left,flag_val,rec_var_stat);
        copy_var_stat(var_stat,&rec_var_stat);
        if (ssa_l_stat != NULL)
                update_write_stat(rec_var_stat,ssa_l_stat);
        ssa_r_stat = final_ins_phi(node_cfg->right,flag_val,rec_var_stat);
        if (ssa_r_stat != NULL)
                update_write_stat(rec_var_stat,ssa_r_stat);
```

```
        free_ssa_stat(ssa_l_stat);
        free_ssa_stat(ssa_r_stat);
        return(rec_var_stat);
}

/* ************************************************************************ */
/* ************************************************************************ */
/* ************************************************************************ */
/* ************************************************************************ */

void init_var_stat(f_var_stat)
struct ssa_var_stat **f_var_stat;
{
        struct ssa_var_stat *tmpstat;
        struct variable_n *tmpvar;

        tmpvar = first_var;
        tmpstat = (struct ssa_var_stat *) malloc(sizeof(struct ssa_var_stat));
        *f_var_stat = tmpstat;
        if (first_var != NULL) {
                strcpy(tmpstat->var,first_var->var);
                tmpstat->read_no = 0;
                tmpstat->write_no = 1;
                tmpstat->stmt_no = 0;
                tmpstat->next = NULL;
        }
        tmpvar = tmpvar->next;
        while (tmpvar != NULL) {
                tmpstat->next = (struct ssa_var_stat *) malloc(sizeof(struct
ssa_var_stat));
                tmpstat = tmpstat->next;
                strcpy(tmpstat->var,tmpvar->var);
                tmpstat->read_no = 0;
                tmpstat->write_no = 1;
                tmpstat->stmt_no = 0;
                tmpstat->next = NULL;
                tmpvar = tmpvar->next;
        }
}

/* ************************************************************************ */
/* ************************************************************************ */
/*
        FIND_ST_NEWNAME () -->  This procedure finds a new name for
                the variable passed on to this procedure.
                This procedure uses the status of the variables, and the
                flag to produce the new variable.
*/
/* ************************************************************************ */
/* ************************************************************************ */


int find_st_newname(st,flag,f_stat,st_number)
char *st;
int flag;
struct ssa_var_stat *f_stat;
int st_number;
{
        int i,j,k;
        char str1[20], number;

        while ((f_stat != NULL) && ((j=strcmp(f_stat->var,st)) != 0))
                f_stat = f_stat->next;
/*      Check for the variable in the linked list */
/*      If variable not found print error */
        if(f_stat == NULL) {
                printf("\n Error: %s Variable Not Defined ");
                exit(1);
        }
        else {
                if (flag == READ_MODE) {
/*      If variable is referenced then the variable needs to be read */
                        if (f_stat->read_no == 0) {
                                printf("\n Warning: %s Variable Used Without
Initialization");
                        }
                        sprintf(str1,"%d",f_stat->read_no);
                        strcat(st,"##");
                        strcat(st,str1);
                        number = f_stat->read_no;
                        f_stat->stmt_no = st_number;
                }
```

```
                        strcat(newstr,st);
                }
/*      If the variable is an assignment variable then after the first
        variable is given a new name the other variables need to be
        given the old referenced name */
                if (node_cfg->what_stmt != 2)
                        flag = READ_MODE;

        }
        free(node_cfg->stmt);
        node_cfg->stmt = newstr;
        if ((node_cfg->what_stmt < 4) && (node_cfg->what_stmt != 2)){
                        update_read_stat(f_stat,write_str);
                        }
}

/* ********************************************************************** */
/* ********************************************************************** */
/*
        UPDATE_IN_SIGH() --> This function initialize the phi variables
                var##num1 = SIGH (var##num2, var##num3, ..., var#numn);
                This function captures the num2, num3, ..., numn
*/
/* ********************************************************************** */
/* ********************************************************************** */

update_in_phi(node_cfg,f_stat)
struct cfg_n *node_cfg;
struct ssa_var_stat *f_stat;
{
        char st[20],str1[10];
        char *newstr;
        int len,i,j,k,flag,new_num;
        struct phi_var_list *phitmp;
        struct phi_var_num *ntmp;
        struct ssa_var_stat *t_stat;

        phitmp = node_cfg->f_phi;
        if (phitmp == NULL) return;
        while(phitmp != NULL) {
                strcpy(st,phitmp->sivar);
                t_stat = f_stat;
                while ((t_stat != NULL)&&((j=strcmp(t_stat->var,st))!=0))
                        t_stat = t_stat->next;
                if (phitmp->f_num == NULL) {
                        ntmp=(struct phi_var_num *)malloc(sizeof(struct phi_var_num));
                        phitmp->f_num = ntmp;
                        phitmp->l_num = ntmp;
                        sprintf(str1,"%d",t_stat->read_no);
                        strcat(st,"##");
                        strcat(st,str1);
                        ntmp->no = t_stat->read_no;
                        ntmp->stmt_no = t_stat->stmt_no;
                        strcpy(ntmp->svar,st);
                        ntmp->next = NULL;
                }
                else {
                        ntmp=(struct phi_var_num *)malloc(sizeof(struct phi_var_num));
                        phitmp->l_num->next= ntmp;
                        phitmp->l_num = ntmp;
                        sprintf(str1,"%d",t_stat->read_no);
                        strcat(st,"##");
                        strcat(st,str1);
                        ntmp->no = t_stat->read_no;
                        ntmp->stmt_no = t_stat->stmt_no;
                        ntmp->next = NULL;
                        strcpy(ntmp->svar,st);
                }
                phitmp = phitmp->next;
        }
}

/* ********************************************************************** */
/* ********************************************************************** */
/*
        UPDATE_SSA_VAR_STAT() --> This procedure is used to update
                the linked list that maintains the status of the variables

*/
/* ********************************************************************** */
/* ********************************************************************** */
update_ssa_var_stat(fssa_var,node_cfg)
```

```
struct ssa_var_stat **fssa_var;
struct cfg_n *node_cfg;
{
        struct ssa_var_stat *tvar;
        char str1[200],st[100],str[10];
        int i,j,len,number, what_st, leng, rd_flag = 0;
        struct ssa_var_stat *f_var, *l_var;

        if(node_cfg == NULL) return;
        if (node_cfg->stmt == NULL) return;
/*      If current statement is NULL then this node is a Merge Node */
/*      Hence the values need not be updated as far as the variables are
        concerned */
        f_var = *fssa_var;

        strcpy(str1,node_cfg->stmt);
        len = strlen(str1);
        what_st = node_cfg->what_stmt;
        if (what_st > 3) rd_flag = 1;
/*      If the current statement is not an assignment statement then
        all variables are only referenced or all the variables used
        in this statement are read only
*/
        i = 0;
        leng = len;
        do {
                j = 0;
/*      Each variable is identified and stored in st */
                while(str1[i] != '#') {
                        st[j] = str1[i];
                        i++;
                        j++;
                }
                st[j] = '\0';
                j = 0;
                i = i+2;
                len = strlen(str1);
                while((i<len) && (isdigit(str1[i]))) {
                        str[j] = str1[i];
                        i++;
                        j++;
                }
/*      The number in each variable is also split */
                i = i+1;
                str[j] = '\0';
                number = atoi(str);
                while((i < len) && (!(isalpha(str1[i])))) i++;
/*      If there exists no list which keeps track of the
        status of the variables the such a list is created */
                if (f_var == NULL) {
                        l_var=(struct ssa_var_stat *) malloc(sizeof(struct ssa_var_stat));
                        l_var->next = NULL;
                        strcpy(l_var->var,st);
                        l_var->write_no = number;
                        l_var->read_no = number;
                        *fssa_var = l_var;
                        f_var = l_var;
/*      The number of the variable as it appears is updated on to the list */
                }
                tvar = f_var;

/*      If such a list that keeps track of the status of the variables
        already exist then the variable that occurs in the current
        statement is stored.  Its position in the list is identified */

                while ((tvar != NULL) && ((j=strcmp(tvar->var,st)) != 0)){
                        l_var = tvar;
                        tvar = tvar->next;
                }
/*      If for the current variable there does not exists a node in the
        linked list maintaining the status then a node is created in the
        list so that the information is stored in the list */

                if (tvar == NULL) {
                        l_var->next = (struct ssa_var_stat *) malloc(sizeof(struct
ssa_var_stat));
                        l_var = l_var->next;
                        l_var->next = NULL;
                        strcpy(l_var->var,st);
                        if (rd_flag == 0) {
                                l_var->write_no = number;
                                l_var->read_no = number;
```

```
                                if (what_st != 2) rd_flag = 1;
                                }
                        else
                                l_var->read_no = number;
                }
                else {
                if ((j=strcmp(tvar->var,st)) == 0) {

/*      If for the given variable there exists information in the table then
        depending on the statement the rd_flag, the information is updated
        in the linked list */

                        if (rd_flag == 0) {
                                tvar->write_no = number;
                                tvar->read_no = number;
                                if (what_st != 2) rd_flag = 1;
                                }
                        else
                                tvar->read_no = number;
                        }
                }
        }while((what_st == 2) && (i < leng));
}



/* ********************************************************************** */
/* ********************************************************************** */
/*
        COPY_LIST_SSA_STAT() -->  The list maintain the status of each
                variable is copied on to some intermediate linked list
                so that the data at this instant can be stored.
*/
/* ********************************************************************** */
/* ********************************************************************** */
copy_list_ssa_stat(fssa1,fssa2)
struct ssa_var_stat *fssa1,**fssa2;
{
        struct ssa_var_stat *tmps,*ssa_tmp;
        if (fssa1 == NULL) {
                *fssa2 = fssa1;
                return;
        }
        tmps = (struct ssa_var_stat *) malloc(sizeof(struct ssa_var_stat));
        strcpy(tmps->var,fssa1->var);
        tmps->read_no = fssa1->read_no;
        tmps->write_no = fssa1->write_no;
        *fssa2 = tmps;
        tmps->next = NULL;
        ssa_tmp = fssa1->next;
        while(ssa_tmp != NULL) {
                tmps->next = (struct ssa_var_stat *) malloc(sizeof(struct ssa_var_stat));
                tmps = tmps->next;
                strcpy(tmps->var,ssa_tmp->var);
                tmps->read_no = ssa_tmp->read_no;
                tmps->write_no = ssa_tmp->write_no;
                tmps->next = NULL;
                ssa_tmp= ssa_tmp->next;
        }
}

/* ********************************************************************** */
/* ********************************************************************** */
/*
        CHANGE_SSA_TO_STAT () -->  The status of the variables is maintained
                in the linked list. At places where the Sigh function needs
                to be introduced the phi variables are identified by another
                structure which is similar to the one in which the status
                of the variables are stored.
                The other structure using which the phi variables are
                identified is done using
*/
/* ********************************************************************** */
/* ********************************************************************** */

change_ssa_to_stat(f_ssa_stat, fssa)
struct ssa_var_stat *f_ssa_stat;
struct ssa_var **fssa;
{
struct ssa_var_stat *tstat;
struct ssa_var *tssa;
```

```
tstat = f_ssa_stat;

if (tstat == NULL) {
        *fssa = NULL;
        return;
        }
tssa = (struct ssa_var *) malloc(sizeof(struct ssa_var));
*fssa = tssa;
tssa->no = tstat->read_no;
strcpy(tssa->var,tstat->var);
tssa->next = NULL;
tstat = tstat->next;
while (tstat != NULL) {
        tssa->next = (struct ssa_var *) malloc(sizeof(struct ssa_var));
        tssa = tssa->next;
        tssa->no = tstat->read_no;
        strcpy(tssa->var,tstat->var);
        tssa->next = NULL;
        tstat = tstat->next;
        }
tssa = *fssa;
}
```

The following file is ssa.h

```
/* ************************************************************************ */
/* ************************************************************************ */
/*
        FIRST_PASS_SIGH() - SIGH is the name of the function that is used
                to indicate the SSA -form.
                To build the SSA form there are basically two passes
                each with its own function.
                This function is the first pass in the building up of
                the SSA form
*/
/* ************************************************************************ */
/* ************************************************************************ */
void first_pass_phi(node_cfg,flag_val,fssa_var,f_stat)
struct cfg_n *node_cfg;
int flag_val;
struct ssa_var_stat *fssa_var;
struct ssa_var_stat *f_stat;
{
        int froml=0,tol=0,fromr=0,tor=0,lflag,rflag;
        struct cfg_n *lnode,*rnode;
        struct ssa_var_stat *rec_ssa, *tssa;
        struct ssa_var *first_ssa;
        struct ssa_ins_list *tmpssa;

        if (node_cfg == NULL) return;
        if (node_cfg->flag == flag_val) return;

/*      The status of the variables that appear in the current node
        is updated on to the list */
        update_ssa_var_stat(&fssa_var,node_cfg);

        if ((node_cfg->flag != flag_val) &&(node_cfg->left != NULL)) {

/*      If the this node in the CFG has not been visited and if the
        exists a left child also then */

/*      The condition about the existence of the left child is due to
        the fact that the phi variables may or may not be introduced in
        the child of the current node */

                froml = node_cfg->st_no;
                lnode = node_cfg->left;
                tol = lnode->st_no;
/*      no_of_tr - NO of Times This node Reached is not equal to zero */
/*      Because each time this node is reached this no_of_tr is decremented */

                if (lnode->no_of_tr != 0 ){
                        lnode->no_of_tr--;
                        if (lnode->no_in_edges > 1){
/*      If this node could be reached by more than one path */
                                change_ssa_to_stat(fssa_var,&first_ssa);
                                if (lnode->f_list == NULL) {
/*      Store the status of the variables at the node where more than
        one edge converge */
                                        lnode->f_list = (struct ssa_ins_list *)
malloc(sizeof(struct ssa_ins_list));
                                        lnode->f_list->ssa_list = first_ssa;
                                        lnode->f_list->next = NULL;
                                        lnode->l_list = lnode->f_list;
                                }
                                else {
/*      If there already exists the status of variables in the node update
        it */
                                        lnode->l_list->next = (struct ssa_ins_list *)
malloc(sizeof(struct ssa_ins_list));
                                        lnode->l_list = lnode->l_list->next;
                                        lnode->l_list->next = NULL;
                                        lnode->l_list->ssa_list = first_ssa;
/*      Once the node is updated with the latest status find the phi
        variables or the variables for which a phi function needs to
        be inserted */
                                        first_intro_phi(lnode,first_ssa,f_stat);
                                }
                        }
                }
        }
/*      Before going to the left sub-tree store the current status of the
        variables */
```

```
/*      This is done because by entering the left subtree the status of the
        referenced variables may change depending upon the assignment or
        the initialization of the variables */

        copy_list_ssa_stat(fssa_var,&rec_ssa);

        if (froml <= tol)
                first_pass_phi(node_cfg->left,flag_val,rec_ssa,f_stat);

/*      Then Look into the right subtree */

        if ((node_cfg->flag != flag_val) &&(node_cfg->right != NULL)) {

/*      If the this node in the CFG has not been visited and if the
        exists a right child also then */

/*      The condition about the existence of the right child is due to
        the fact that the phi variables may or may not be introduced in
        the child of the current node */

                fromr = node_cfg->st_no;
                rnode = node_cfg->right;
                tor = rnode->st_no;
                if (rnode->no_of_tr != 0 ){
/*      If this node could be reached by more than one path */
                        rnode->no_of_tr--;
                        if (rnode->no_in_edges > 1){
/*      Store the status of the variables at the node where more than
        one edge converge */
/*      Before such a storage convert the structure in the form such
        that it could be stored */
                                change_ssa_to_stat(fssa_var,&first_ssa);
                                if (rnode->f_list == NULL) {
                                        rnode->f_list = (struct ssa_ins_list *)
malloc(sizeof(struct ssa_ins_list));
                                        rnode->f_list->ssa_list = first_ssa;
                                        rnode->f_list->next = NULL;
                                        rnode->l_list = rnode->f_list;
                                }
                                else {
                                        rnode->l_list->next = (struct ssa_ins_list *)
malloc(sizeof(struct ssa_ins_list));
                                        rnode->l_list = rnode->l_list->next;
                                        rnode->l_list->next = NULL;
                                        rnode->l_list->ssa_list = first_ssa;
/*      Once the node is updated with the latest status find the phi
        variables or the variables for which a phi function needs to
        be inserted */
                                        first_intro_phi(rnode,first_ssa,f_stat);
                                }
                        }
                }
        }
/*      Make the current node visited */
        node_cfg->flag = flag_val;

/*      Before going to the left sub-tree store the current status of the
        variables */

/*      This is done because by entering the left subtree the status of the
        referenced variables may change depending upon the assignment or
        the initialization of the variables */
        copy_list_ssa_stat(fssa_var,&rec_ssa);
        if (fromr <= tor)
                first_pass_phi(node_cfg->right,flag_val,rec_ssa,f_stat);
        free_ssa_stat(rec_ssa);
}


/* ************************************************************************ */
/* ************************************************************************ */
/*
        This procedure is the implementation of algorithm 4.
        This procedure traverses CFG and introduces Phi functions
        where necessary
*/
/* ************************************************************************ */
/* ************************************************************************ */
void gen_ssa_form(node_cfg,flag_val,fssa_var,f_stat)
struct cfg_n *node_cfg;
int flag_val;
```

```
struct ssa_var *fssa_var;
struct ssa_var_stat *f_stat;
{
        int froml=0,tol=0,fromr=0,tor=0,lflag,rflag,up_flag;
        struct cfg_n *lnode,*rnode;
        struct ssa_var *first_ssa, *rec_ssa, *tssa;
        struct ssa_ins_list *tmpssa;

        if (node_cfg == NULL) return;
        if (node_cfg->flag_end == flag_val) return;
/*      Update the status of the variables where the introduction of the phi
        variables has taken place */
/*      The introduced phi variables also need to propagate to other places
        */

        update_f_phis(node_cfg,&fssa_var,f_stat,flag_val);

        if (node_cfg->what_stmt < 4)
                update_ssa_var(&fssa_var,node_cfg);
/*      The status of the variables that appear in the current node
        is updated on to the list */

/*      Update the left hand side of the current node */
        if (node_cfg->left != NULL) {
                froml = node_cfg->st_no;
                lnode = node_cfg->left;
                tol = lnode->st_no;
                copy_list_ssa(fssa_var,&first_ssa);
/*      The condition about the existence of the left child is due to
        the fact that the phi variables may or may not be introduced in
        the child of the current node */

/*      A copy of the current status is stored in the status of the other
        variables */
                if (lnode->f_list == NULL) {
                        lnode->f_list = (struct ssa_ins_list *) malloc(sizeof(struct
ssa_ins_list));
                        lnode->f_list->ssa_list = first_ssa;
                        lnode->f_list->next = NULL;
                        lnode->l_list = lnode->f_list;
                        }
                else {
                        lnode->l_list->next = (struct ssa_ins_list *) malloc(sizeof(struct
ssa_ins_list));
                        lnode->l_list = lnode->l_list->next;
                        lnode->l_list->next = NULL;
                        lnode->l_list->ssa_list = first_ssa;
/*      Once the node is updated with the latest status find the phi
        variables or the variables for which a phi function needs to
        be inserted */
                        update_phi_l(lnode,first_ssa,f_stat);
                        }
                }
/*      A copy of the status of the variables are stored before the
        left subtree is traversed */
        copy_list_ssa(fssa_var,&rec_ssa);
        if (froml <= tol)
                gen_ssa_form(node_cfg->left,flag_val,rec_ssa,f_stat);

        node_cfg->flag = flag_val;
        update_f_phis(node_cfg,&fssa_var,f_stat,flag_val);
        if (node_cfg->what_stmt < 4)
                update_ssa_var(&fssa_var,node_cfg);

        if (node_cfg->right != NULL) {
                fromr = node_cfg->st_no;
                rnode = node_cfg->right;
                tor = rnode->st_no;
                first_ssa = NULL;
                copy_list_ssa(fssa_var,&first_ssa);
                if (rnode->f_list == NULL) {
                        rnode->f_list = (struct ssa_ins_list *) malloc(sizeof(struct
ssa_ins_list));
                        rnode->f_list->ssa_list = first_ssa;
                        rnode->f_list->next = NULL;
                        rnode->l_list = rnode->f_list;
                        }
                else {
                        rnode->l_list->next = (struct ssa_ins_list *) malloc(sizeof(struct
ssa_ins_list));
                        rnode->l_list = rnode->l_list->next;
                        rnode->l_list->next = NULL;
```

```
                        rnode->l_list->ssa_list = first_ssa;
                        update_phi_l(rnode,first_ssa,f_stat);
                        }
                }
        copy_list_ssa(fssa_var,&rec_ssa);
        if (fromr <= tor){
                gen_ssa_form(node_cfg->right,flag_val,rec_ssa,f_stat);
                }
        node_cfg->flag_end = flag_val;
}


/* ********************************************************************** */
/* ********************************************************************** */
/*
        FIRST_INTRO_SIGH () -->  This function introduces the phi
                variables depending upon the linked list containing the
                status variables that has come by another path.

                This function compares the status of the variables that
                have been stored in this node and the status of the
                variables that has arrived to this node by a different
                path.

                If the status of a variable differs then that variable
                needs a phi function to be introduced

                This function performs the above mentioned task

*/
/* ********************************************************************** */
/* ********************************************************************** */
first_intro_phi(node_g,t_ssa,f_stat)
struct cfg_n *node_g;
struct ssa_var *t_ssa;
struct ssa_var_stat *f_stat;
{
        int j;
        struct ssa_var_stat *tmp_vstat;
        struct phi_var_list *t_phi_var;
        struct ssa_var *fssa_var, *travel_list, *prev_node;
        char *tmpvar;
        struct ssa_int_occcr *t_intro;


        fssa_var = node_g->f_list->ssa_list;
        tmp_vstat = f_stat;
        while(t_ssa != NULL) {

/*      Keeping one list standard and tracing the other list for the
        variable present in the standard list is the algorithm followed here */

                travel_list = fssa_var;
                while((travel_list != NULL) && (j=strcmp(travel_list->var,t_ssa->var)
!=0)){
                        prev_node = travel_list;
                        travel_list = travel_list->next;
                }
                if(travel_list == NULL) {
                        prev_node->next = (struct ssa_var*) malloc(sizeof(struct ssa_var));
                        prev_node = prev_node->next;
                        prev_node->next = NULL;
                        strcpy(prev_node->var,t_ssa->var);
                        prev_node->no = t_ssa->no;
                }
                else {
                        tmpvar = travel_list->var;
                        if (travel_list->no != t_ssa->no){
                                while ((tmp_vstat != NULL)
&&((j=strcmp(tmp_vstat->var,tmpvar)) != 0))
                                        tmp_vstat = tmp_vstat->next;
/*      If the status of a variable differs then that variable needs a
        phi function */
                                travel_list->no = t_ssa->no;
/*      The variable is appended to a list of variables for which the phi
        function needs to be assigned */
                                if (node_g->f_phi == NULL) {
/*      If the phi list is not present then the list is initialized */
                                        node_g->f_phi = (struct phi_var_list *)
malloc(sizeof(struct phi_var_list));
                                        strcpy(node_g->f_phi->          sivar,tmpvar);
                                        if (tmp_vstat != NULL)
```

```
                                        node_g->f_phi->no = tmp_vstat->write_no;
                                    else {
                                        printf("\n SORRY");
                                        }
                                    tmp_vstat->write_no++;
                                    node_g->f_phi->next = NULL;
                                    node_g->f_phi->f_num = NULL;
                                    node_g->f_phi->l_num = NULL;
                                    node_g->l_phi = node_g->f_phi;
                                }
                                else {
                                    t_phi_var = node_g->f_phi;
                                    while((t_phi_var != NULL) &&
(j=strcmp(t_phi_var->sivar,tmpvar) != 0)) {
                                        t_phi_var = t_phi_var->next;
                                    }
                                    if (t_phi_var == NULL) {
                                        node_g->l_phi->next = (struct phi_var_list *)
malloc(sizeof(struct phi_var_list));
                                        node_g->l_phi=node_g->l_phi->next;

strcpy(node_g->l_phi->sivar,travel_list->var);
                                    if (tmp_vstat != NULL)
                                        node_g->l_phi->no = tmp_vstat->write_no;
                                    else {
                                        printf("\n SORRY");
                                        }
                                        tmp_vstat->write_no++;
                                        node_g->l_phi->next = NULL;
                                        node_g->l_phi->f_num = NULL;
                                        node_g->f_phi->l_num = NULL;
                                    }
                                }
                            }
                        }
                        t_ssa = t_ssa->next;
                }
}

/* ********************************************************************** */
/* ********************************************************************** */
/*
        INITIALIZE_FIRST_SSA() --> is used to initialize the status
                of each variable in the beginning of the second pass
                of the SSA-form.
*/
/* ********************************************************************** */
/* ********************************************************************** */
struct ssa_var *initialize_first_ssa()
{
struct ssa_var *f_ssa, *tmp_ssa;
struct variable_n *tmp_var;

tmp_var = first_var;
tmp_ssa = (struct ssa_var *) malloc(sizeof(struct ssa_var));
tmp_ssa->no = 0;
strcpy(tmp_ssa->var,tmp_var->var);
tmp_ssa->next = NULL;
f_ssa = tmp_ssa;
while(tmp_var != NULL) {
        tmp_ssa->next = (struct ssa_var *) malloc(sizeof(struct ssa_var));
        tmp_ssa = tmp_ssa->next;
        strcpy(tmp_ssa->var,tmp_var->var);
        tmp_ssa->no = 0;
        tmp_ssa->next = NULL;
        tmp_var = tmp_var->next;
        }
return(f_ssa);
}
```

The following file is gsa.h

```
/* ********************************************************************** */
/* ********************************************************************** */
/*
        INTRO_GSA_FORM -->  This function introduces the GSA function
                or the ETA function
*/
/*              Here the problem is to introduce Eta Function                */
/* ********************************************************************** */
/* ********************************************************************** */
void intro_gsa_form(node_cfg,flag_val)
struct cfg_n *node_cfg;
int flag_val;
{
        int froml=0,tol=0,fromr=0,tor=0;
        struct cfg_n *lnode,*rnode, *tnode;
        struct cfg_n *int_node;
        struct phi_var_list *node_phi, *tnode_phi;
        struct phi_var_num *tnum, *earnum;
        struct int_list *st_list, *end_l, *int_l;

        if (node_cfg == NULL) return;
        if (node_cfg->flag_end == flag_val) return;
        if (node_cfg->left != NULL) {
                froml = node_cfg->st_no;
                lnode = node_cfg->left;
                tol = lnode->st_no;
                node_phi = lnode->f_phi;
                if ((node_cfg->what_stmt == 21)&&(node_phi != NULL)) {
                        tnode_phi = (struct phi_var_list *) malloc(sizeof(struct
phi_var_list));
                        node_cfg->f_phi = tnode_phi;
                        tnode_phi->next = NULL;
                        strcpy(tnode_phi->var, node_phi->var);
                        strcpy(tnode_phi->sivar, node_phi->sivar);
                        tnum = NULL;
                        if (node_phi->f_num != NULL) {
                                earnum = node_phi->f_num->next;
                                if (earnum != NULL) {
                                        tnum = (struct phi_var_num *) malloc(sizeof(struct
phi_var_num));
                                        tnum->no = earnum->no;
                                        strcpy(tnum->svar,earnum->svar);
                                        tnum->stmt_no = earnum->stmt_no;
                                }
                        }
                        tnode_phi->no = node_phi->no;
                        node_cfg->f_phi = tnode_phi;
                        node_cfg->l_phi = tnode_phi;
                        tnode_phi->no = node_phi->no;
                        tnode_phi->f_num = tnum;
                        tnode_phi->l_num = tnum;
                        tnode_phi->next = NULL;
                        node_phi = node_phi->next;
                        while (node_phi != NULL) {
                                tnode_phi->next = (struct phi_var_list *)
malloc(sizeof(struct phi_var_list));
                                tnode_phi = tnode_phi->next;
                                strcpy(tnode_phi->var, node_phi->var);
                                strcpy(tnode_phi->sivar, node_phi->sivar);
                                tnode_phi->no = node_phi->no;
                                tnum = NULL;
                                if (node_phi->f_num != NULL) {
                                        earnum = node_phi->f_num->next;
                                        if (earnum != NULL) {
                                                tnum = (struct phi_var_num *)
malloc(sizeof(struct phi_var_num));
                                                tnum->no = earnum->no;
                                                strcpy(tnum->svar,earnum->svar);
                                                tnum->stmt_no = earnum->stmt_no;
                                        }
                                }
                                tnode_phi->f_num = tnum;
                                tnode_phi->l_num = tnum;
                                tnode_phi->next = NULL;
                                node_phi = node_phi->next;
                        }
                }
                if (lnode->what_stmt == 12) {
```

```
                /*      Insert the path along which this Gamma Function
                        node is reached */
                        st_list = lnode->f_int;
                        if (st_list == NULL) {
                                st_list = (struct int_list *) malloc(sizeof(struct
int_list));
                                st_list->no = node_cfg->start_stmt;
                                st_list->next = NULL;
                                lnode->f_int = st_list;
                                }
                        else {
                                while (st_list->next != NULL)
                                        st_list = st_list->next;
                                st_list->next = (struct int_list *) malloc(sizeof(struct
int_list));
                                st_list= st_list->next;
                                st_list->no = node_cfg->start_stmt;
                                st_list->next = NULL;
                                }
                        }
        }
        node_cfg->flag = flag_val;
        if (froml <= tol)
                intro_gsa_form(node_cfg->left,flag_val);
        if (node_cfg->right != NULL) {
                fromr = node_cfg->st_no;
                rnode = node_cfg->right;
                tor = rnode->st_no;
                node_phi = rnode->f_phi;
                if ((node_cfg->what_stmt == 21)&&(node_phi != NULL)) {
                        tnode_phi = (struct phi_var_list *) malloc(sizeof(struct
phi_var_list));
                        node_cfg->f_phi = tnode_phi;
                        tnode_phi->next = NULL;
                        strcpy(tnode_phi->var, node_phi->var);
                        strcpy(tnode_phi->sivar, node_phi->sivar);
                        tnum = NULL;
                        if (node_phi->f_num != NULL) {
                                earnum = node_phi->f_num->next;
                                if (earnum != NULL) {
                                        tnum = (struct phi_var_num *) malloc(sizeof(struct
phi_var_num));
                                        tnum->no = earnum->no;
                                        strcpy(tnum->svar,earnum->svar);
                                        tnum->stmt_no = earnum->stmt_no;
                                }
                        }
                        tnode_phi->no = node_phi->no;
                        tnode_phi->no = node_phi->no;
                        tnode_phi->f_num = tnum;
                        tnode_phi->l_num = tnum;
                        node_cfg->f_phi = tnode_phi;
                        node_cfg->l_phi = tnode_phi;
                        node_phi = node_phi->next;
                        while (node_phi != NULL) {
                                tnode_phi->next = (struct phi_var_list *)
malloc(sizeof(struct phi_var_list));
                                tnode_phi = tnode_phi->next;
                                strcpy(tnode_phi->var, node_phi->var);
                                strcpy(tnode_phi->sivar, node_phi->sivar);
                                tnum = NULL;
                                if (node_phi->f_num != NULL) {
                                        earnum = node_phi->f_num->next;
                                        if (earnum != NULL) {
                                                tnum = (struct phi_var_num *)
malloc(sizeof(struct phi_var_num));
                                                tnum->no = earnum->no;
                                                strcpy(tnum->svar,earnum->svar);
                                                tnum->stmt_no = earnum->stmt_no;
                                        }
                                }
                                strcpy(tnode_phi->sivar,node_phi->sivar);
                                tnode_phi->no = node_phi->no;
                                tnode_phi->f_num = tnum;
                                tnode_phi->l_num = tnum;
                                tnode_phi->next = NULL;
                                node_phi = node_phi->next;
                        }
                }
                if (rnode->what_stmt == 12) {
                /*      Insert the path along which this Gamma Function
                        node is reached */
```

```
                        st_list = lnode->f_int;
                        if (st_list == NULL) {
                                st_list = (struct int_list *) malloc(sizeof(struct
int_list));
                                st_list->no = node_cfg->start_stmt;
                                st_list->next = NULL;
                              lnode->f_int = st_list;
                                }
                        else {
                                while (st_list->next != NULL)
                                        st_list = st_list->next;
                                st_list->next = (struct int_list *) malloc(sizeof(struct
int_list));
                                st_list= st_list->next;
                                st_list->no = node_cfg->start_stmt;
                                st_list->next = NULL;
                                }
                        }
        }
        if (fromr <= tor)
                intro_gsa_form(node_cfg->right,flag_val);
        node_cfg->flag_end = flag_val;
}
```

The following file is print.h

```
/* ************************************************************************ */
/* ************************************************************************ */
/*      PRINT_CFG_GRA () --> Prints the Control flow graph in
                as found in the linked list */
/* ************************************************************************ */
/* ************************************************************************ */
void print_cfg_gra()
{
        cfg_graph = first_cfg;

        while(cfg_graph != NULL){
                printf("\n %d ------ %d",cfg_graph->from,cfg_graph->to);
                cfg_graph = cfg_graph->next;
        }
}


/* ************************************************************************ */
/* ************************************************************************ */
/*
        PRINT_GRAPH() --> Print the Control Flow graph as per the
                tree structured CFG method
                This is also used to initialize certain variables in
                the nodes of the CFG
*/
/* ************************************************************************ */
/* ************************************************************************ */
void print_graph(node_cfg)
struct cfg_n *node_cfg;
{
        int froml=0,tol=0,fromr=0,tor=0;
        if (node_cfg == NULL) return;
        if (node_cfg->flag == 1) return;
        update_node(node_cfg);
/* Most of the pointers in the node structure get updated here */
        if ((node_cfg->flag != 1) &&(node_cfg->left != NULL)) {
                froml = node_cfg->st_no;
                tol = node_cfg->left->st_no;
                node_cfg->no_of_tr = node_cfg->no_in_edges;
        }
        if ((node_cfg->flag != 1) &&(node_cfg->right != NULL)) {
                fromr = node_cfg->st_no;
                tor = node_cfg->right->st_no;
                node_cfg->no_of_tr = node_cfg->no_in_edges;
        }
        node_cfg->flag = 1;
        if (froml <= tol)
                print_graph(node_cfg->left);
        if (fromr <= tor)
                print_graph(node_cfg->right);
}


/* ************************************************************************ */
/* ************************************************************************ */
/*
        FREE_F_LIST ()  -  Frees the memory of a linked list in
                the tree structured node in the CFG
*/
/* ************************************************************************ */
/* ************************************************************************ */

free_f_list(node_cfg)
struct cfg_n *node_cfg;
{
struct ssa_ins_list *t_insl;

t_insl = node_cfg->f_list;
while(t_insl != NULL) {
        free_rec_ssa(t_insl->ssa_list);
        t_insl = t_insl->next;
        }
node_cfg->f_list = NULL;
}


/* ************************************************************************ */
/* ************************************************************************ */
```

```
/*
        PRINT_LIST_ST ()  --->  Print the list of statements that
                occur in the input code
*/
/* ************************************************************************ */
/* ************************************************************************ */

void print_list_st()
{
        struct stmt_list *tmp_stlist;
        int j;
        tmp_stlist = first_stmt->next;
        while(tmp_stlist != NULL) {
                if ((j = strcmp(tmp_stlist->stmt,NULL)) != 0)
                        printf("\n %d %s",tmp_stlist->st_no,tmp_stlist->stmt);
                else printf("\n %d MERGE ",tmp_stlist->st_no);
                tmp_stlist = tmp_stlist->next;
        }
}

/* ************************************************************************ */
/* ************************************************************************ */
/*
        PRINT_EACH_VAR () --> Prints the variables in the maintenance
                of the status of the variables
*/
/* ************************************************************************ */
/* ************************************************************************ */
print_each_var(f_var)
struct ssa_var *f_var;
{
        struct ssa_var *tvar;
        struct ssa_int_occr *t_in;

        tvar = f_var;
        while(tvar != NULL) {
                printf("*** %s %d",tvar->var,tvar->no);
                tvar = tvar->next;
        }
}


/* ************************************************************************ */
/* ************************************************************************ */
/*
        FREE_REC_SSA () -->  Free the linked list containing the
                the status of the variables
*/
/* ************************************************************************ */
/* ************************************************************************ */
free_rec_ssa(tmp_ssa)
struct ssa_var *tmp_ssa;
{
        struct ssa_var *free_ssa;
        free_ssa = tmp_ssa;
        while(tmp_ssa != NULL) {
                tmp_ssa = tmp_ssa->next;
                free(free_ssa);
                free_ssa = tmp_ssa;
        }
}



/* ************************************************************************ */
/* ************************************************************************ */
/*
        PRINT_SSA_GRAPH () --> Print the SSA -form of the CFG graph
*/
/* ************************************************************************ */
/* ************************************************************************ */
void print_ssa_graph(node_cfg,flag_val)
struct cfg_n *node_cfg;
int flag_val;
{
        int froml=0,tol=0,fromr=0,tor=0;
        if (node_cfg == NULL) return;
        if (node_cfg->flag == flag_val) return;
        if ((node_cfg->flag != flag_val) &&(node_cfg->left != NULL)) {
                froml = node_cfg->st_no;
                tol = node_cfg->left->st_no;
```

```
                printf("\n %d ----->  %d ",froml,tol);
                printf("  **** %d ",node_cfg->no_in_edges);
                if (node_cfg->stmt == NULL)
                        printf("  NODE: %d - - - - - - ",node_cfg->st_no);
                else
                        printf("  NODE: %d %s ",node_cfg->st_no,node_cfg->stmt);
        }
        if ((node_cfg->flag != flag_val) &&(node_cfg->right != NULL)) {
                fromr = node_cfg->st_no;
                tor = node_cfg->right->st_no;
                printf("\n %d ----->  %d ",fromr,tor);
                printf("  **** %d ",node_cfg->no_in_edges);
                if (node_cfg->stmt == NULL)
                        printf("  NODE: %d - - - - - - ",node_cfg->st_no);
                else
                        printf("  NODE: %d %s ",node_cfg->st_no,node_cfg->stmt);
        }
        node_cfg->flag = flag_val;
        if (froml <= tol)
                print_ssa_graph(node_cfg->left,flag_val);
        if (fromr <= tor)
                print_ssa_graph(node_cfg->right,flag_val);
}

/* ********************************************************************** */
/* ********************************************************************** */
/*
        PRINT_FLIST () -->  Print the list of variables and their
                variable names as appears in the PHI function
*/
/* ********************************************************************** */
/* ********************************************************************** */
/* ********************************************************************** */

print_flist(node)
struct cfg_n *node;
{
        struct ssa_var *tmpssa;
        struct ssa_ins_list *tmpl;

        tmpl = node->f_list;

        while (tmpl != NULL) {
                tmpssa = tmpl->ssa_list;
                while(tmpssa != NULL) {
                        printf(" %s %d",tmpssa->var,tmpssa->no);
                        tmpssa = tmpssa->next;
                }
                tmpl = tmpl->next;
                printf("\n");
        }
}

/* ********************************************************************** */
/* ********************************************************************** */
/*
        PRINT_F_PHI () --> Print the first assignment variable of a
                PHI function
*/
/* ********************************************************************** */
/* ********************************************************************** */


print_f_phi(node_cfg)
struct cfg_n *node_cfg;
{
struct phi_var_list *tmp;
tmp = node_cfg->f_phi;
while(tmp != NULL) {
        printf(" %d  %s  ",tmp->no,tmp->sivar);
        tmp = tmp->next;
        }
}


/* ********************************************************************** */
/* ********************************************************************** */
/*
        PRINT_PHI_VAR () --->  This function is used to print the phi
                variables.  This function was mainly used to debug the code
*/
/* ********************************************************************** */
```

```
/* ********************************************************************** */
print_phi_var(phi_var)
struct phi_var_list *phi_var;
{

        printf("\n VARIABLES TO BE GIVEN NEW NAME \n");
        while(phi_var != NULL) {
                printf(" %s %d",phi_var->sivar,phi_var->no);
                phi_var = phi_var->next;
        }
        printf("\n");
}


/* ********************************************************************** */
/* ********************************************************************** */
/*
        PRINT_INIT_VARST() -->  Prints the initialized status of the variables
*/
/* ********************************************************************** */
/* ********************************************************************** */
print_init_varst(f_stat)
struct ssa_var_stat *f_stat;
{
        struct ssa_var_stat *tmpstat;
        tmpstat = f_stat;
        while(tmpstat != NULL){
                printf("\n %s %d %d",tmpstat->var,tmpstat->read_no,tmpstat->write_no);
                tmpstat = tmpstat->next;
        }
}
/* ********************************************************************** */
/* ********************************************************************** */
/*
        FREE_SSA_STAT () -->  Free the status of the variables
*/
/* ********************************************************************** */
/* ********************************************************************** */
free_ssa_stat(freestat)
struct ssa_var_stat *freestat;
{
        struct ssa_var_stat *tstat;

        tstat = freestat;

        while (tstat != NULL) {
                tstat = tstat->next;
                free(freestat);
                freestat = tstat;
        }
}

/* ********************************************************************** */
/* ********************************************************************** */
/*
        PRINT_SSA_FORM () --> Print the SSA form
*/
/* ********************************************************************** */
/* ********************************************************************** */
void print_SSA_form(node_cfg,flag_val)
struct cfg_n *node_cfg;
int flag_val;
{
        int froml=0,tol=0,fromr=0,tor=0;
        if (node_cfg == NULL) return;
        if (node_cfg->flag == flag_val) return;

        if (node_cfg->f_phi != NULL) print_phis(node_cfg->f_phi);

        if ((node_cfg->flag != flag_val) &&(node_cfg->left != NULL)) {
                froml = node_cfg->st_no;
                tol = node_cfg->left->st_no;
                printf("\n %d -----> %d ",froml,tol);
                printf("  **** %d ",node_cfg->no_in_edges);
                if (node_cfg->stmt == NULL)
                        printf("  NODE: %d - - - - - - ",node_cfg->st_no);
                else
                        printf("  NODE: %d %s ",node_cfg->st_no,node_cfg->stmt);
                if (node_cfg->start_stmt != -1)
                        printf(" **%d",node_cfg->start_stmt);
        }
```

```
        if ((node_cfg->flag != flag_val) &&(node_cfg->right != NULL)) {
                fromr = node_cfg->st_no;
                tor = node_cfg->right->st_no;
                printf("\n %d -----> %d ",fromr,tor);
                printf("  **** %d ",node_cfg->no_in_edges);
                if (node_cfg->stmt == NULL)
                        printf("  NODE: %d - - - - - - ",node_cfg->st_no);
                else
                        printf("  NODE: %d %s ",node_cfg->st_no,node_cfg->stmt);
                if (node_cfg->start_stmt != -1)
                        printf(" **%d",node_cfg->start_stmt);
        }
        node_cfg->flag = flag_val;
        if (froml <= tol)
                print_SSA_form(node_cfg->left,flag_val);
        if (fromr <= tor)
                print_SSA_form(node_cfg->right,flag_val);
}

/* ********************************************************************** */
/* ********************************************************************** */
/*
        PRINT_PHIS() --> Print the phi functions
*/
/* ********************************************************************** */
/* ********************************************************************** */
print_phis(node_phi)
struct phi_var_list *node_phi;
{
        struct phi_var_num *tmpnum;
        while(node_phi != NULL) {
                printf("\n %s = PHI(",node_phi->var);
                tmpnum = node_phi->f_num;
                while(tmpnum != NULL) {
                        printf("%s %d %d",tmpnum->svar,tmpnum->no,tmpnum->stmt_no);
                        tmpnum = tmpnum->next;
                        if (tmpnum == NULL) printf(")");
                        else printf(",");
                }
                node_phi = node_phi->next;
        }
}
/* ********************************************************************** */
/* ********************************************************************** */
/*
        PRINT_FINAL_GAMMA() --> Print the Gamma functions in the GSA form
*/
/* ********************************************************************** */
/* ********************************************************************** */
print_final_gamma(node_phi,node_g)
struct phi_var_list *node_phi;
struct cfg_n *node_g;
{
        struct phi_var_num *tmpnum;
        while(node_phi != NULL) {
                printf("\n %s = GAMMA(%d,",node_phi->var,node_g->start_stmt);
                tmpnum = node_phi->f_num;
                while(tmpnum != NULL) {
                        printf("%s",tmpnum->svar);
                        tmpnum = tmpnum->next;
                        if (tmpnum == NULL) printf(")");
                        else printf(",");
                }
                node_phi = node_phi->next;
        }
}

/* ********************************************************************** */
/* ********************************************************************** */
/*
        PRINT_FINAL_GAMMA() --> Print the switches in the GSA form
*/
/* ********************************************************************** */
/* ********************************************************************** */

print_switch_gamma(node_phi,node_g)
struct phi_var_list *node_phi;
struct cfg_n *node_g;
{
        struct phi_var_num *tmpnum;
        struct int_list *st_list;
        int number = 0,i;
```

```
        st_list = node_g->f_int;
        while(node_phi != NULL) {
                printf("\n %s = GAMMA(%d,",node_phi->var,st_list->no);
                number++;
                st_list = st_list->next;
                tmpnum = node_phi->f_num;
                while(tmpnum->next != NULL) {
                        printf("%s,",tmpnum->svar);
                        tmpnum = tmpnum->next;
                        if (tmpnum->next != NULL) {
                                printf(" GAMMA(%d,",st_list->no);
                                number++;
                                st_list = st_list->next;
                                }
                }
                printf("%s",tmpnum->svar);
                for(i=0; i<number;i++) printf(")");
                node_phi = node_phi->next;
                st_list = node_g->f_int;
                number = 0;
        }
}


/* ********************************************************************** */
/* ********************************************************************** */
/*
        PRINT_FINAL_MU () --> Print the MU functions in the GSA form
*/
/* ********************************************************************** */
/* ********************************************************************** */
print_final_mu(node_phi,node_g)
struct phi_var_list *node_phi;
struct cfg_n *node_g;
{
        struct phi_var_num *tmpnum;
        while(node_phi != NULL) {
                printf("\n %s = MU.L(%d,",node_phi->var,node_g->st_no);
                tmpnum = node_phi->f_num;
                while(tmpnum != NULL) {
                        printf("%s",tmpnum->svar);
                        tmpnum = tmpnum->next;
                        if (tmpnum == NULL) printf(")");
                        else printf(",");
                }
                node_phi = node_phi->next;
        }
}

/* ********************************************************************** */
/* ********************************************************************** */
/*
        PRINT_FINAL_ETA () --> Print the Eta functions in the GSA-form
*/
/* ********************************************************************** */
/* ********************************************************************** */
print_final_eta(node_phi,node_g)
struct phi_var_list *node_phi;
struct cfg_n *node_g;
{
        struct phi_var_num *tmpnum;
        while(node_phi != NULL) {
                printf("\n %s = ETA.T(%d,",node_phi->var,node_g->start_stmt);
                tmpnum = node_phi->f_num;
                printf("%s",tmpnum->svar);
                printf(")");
                node_phi = node_phi->next;
                }
}
/* ********************************************************************** */
/* ********************************************************************** */
/* ********************************************************************** */
/*
        PRINT_FINAL_GSA() --> Print the GSA form
*/
/* ********************************************************************** */
/* ********************************************************************** */
void print_final_GSA(node_cfg,flag_val)
struct cfg_n *node_cfg;
int flag_val;
{
        int froml=0,tol=0,fromr=0,tor=0;
```

```
        if (node_cfg == NULL) return;
        if (node_cfg->flag == flag_val) return;

        if (node_cfg->what_stmt == 51) {
                        printf("\n L = .TRUE");
                        }
        if (node_cfg->what_stmt == 52) {
                        printf("\n L = .FALSE");
                        }
        if (node_cfg->f_phi != NULL) {
                if (node_cfg->what_stmt == 20)
                        print_final_gamma(node_cfg->f_phi,node_cfg);
                else if (node_cfg->what_stmt == 5)
                        print_final_mu(node_cfg->f_phi,node_cfg);
                else if (node_cfg->what_stmt == 6)
                        print_final_mu(node_cfg->f_phi,node_cfg);
                else if (node_cfg->what_stmt == 21)
                        print_final_eta(node_cfg->f_phi,node_cfg);
                else if (node_cfg->what_stmt == 12){

                        print_switch_gamma(node_cfg->f_phi,node_cfg);
                        }
                else
                        print_phis(node_cfg->f_phi);
                }

        if ((node_cfg->flag != flag_val) &&(node_cfg->left != NULL)) {
                froml = node_cfg->st_no;
                tol = node_cfg->left->st_no;
                printf("\n %d -----> %d ",froml,tol);
                printf("  **** %d ",node_cfg->no_in_edges);
                if (node_cfg->stmt == NULL)
                        printf("  NODE: %d - - - - - - ",node_cfg->st_no);
                else
                        printf("  NODE: %d %s ",node_cfg->st_no,node_cfg->stmt);
                if (node_cfg->start_stmt != -1)
                        printf(" **%d",node_cfg->start_stmt);
        }
        if ((node_cfg->flag != flag_val) &&(node_cfg->right != NULL)) {
                fromr = node_cfg->st_no;
                tor = node_cfg->right->st_no;
                printf("\n %d -----> %d ",fromr,tor);
                printf("  **** %d ",node_cfg->no_in_edges);
                if (node_cfg->stmt == NULL)
                        printf("  NODE: %d - - - - - - ",node_cfg->st_no);
                else
                        printf("  NODE: %d %s ",node_cfg->st_no,node_cfg->stmt);
                if (node_cfg->start_stmt != -1)
                        printf(" **%d",node_cfg->start_stmt);
        }
        node_cfg->flag = flag_val;
        if (froml <= tol)
                print_final_GSA(node_cfg->left,flag_val);
        if (fromr <= tor)
                print_final_GSA(node_cfg->right,flag_val);
}


/* ********************************************************************** */
/* ********************************************************************** */
/* ********************************************************************** */
/*
        PRINT_PDW() --> Print the PDW
*/
/* ********************************************************************** */
/* ********************************************************************** */
void print_PDW(node_cfg,flag_val)
struct cfg_n *node_cfg;
int flag_val;
{
        int froml=0,tol=0,fromr=0,tor=0;
        if (node_cfg == NULL) return;
        if (node_cfg->flag == flag_val) return;

        if (node_cfg->what_stmt == 51) {
                        printf("\n L = .TRUE");
                        }
        if (node_cfg->what_stmt == 52) {
                        printf("\n L = .FALSE");
                        }
        if (node_cfg->f_phi != NULL) {
```

```
                if (node_cfg->what_stmt == 20)
                        print_final_gamma(node_cfg->f_phi,node_cfg);
                else if (node_cfg->what_stmt == 5)
                        print_final_mu(node_cfg->f_phi,node_cfg);
                else if (node_cfg->what_stmt == 6)
                        print_final_mu(node_cfg->f_phi,node_cfg);
                else if (node_cfg->what_stmt == 21)
                        print_final_eta(node_cfg->f_phi,node_cfg);
                else if (node_cfg->what_stmt == 12){

                        print_switch_gamma(node_cfg->f_phi,node_cfg);
                        }
                else
                        print_phis(node_cfg->f_phi);
                }

        if ((node_cfg->flag != flag_val) &&(node_cfg->left != NULL)) {
                froml = node_cfg->st_no;
                tol = node_cfg->left->st_no;
                printf("\n %d -----> %d ",froml,tol);
                printf("  **** %d ",node_cfg->no_in_edges);
                if (node_cfg->stmt == NULL)
                        printf("  NODE: %d - - - - - - ",node_cfg->st_no);
                else
                        printf("  NODE: %d %s ",node_cfg->st_no,node_cfg->stmt);
                if (node_cfg->start_stmt != -1)
                        printf(" **%d",node_cfg->start_stmt);
        }
        if ((node_cfg->flag != flag_val) &&(node_cfg->right != NULL)) {
                fromr = node_cfg->st_no;
                tor = node_cfg->right->st_no;
                printf("\n %d -----> %d ",fromr,tor);
                printf("  **** %d ",node_cfg->no_in_edges);
                if (node_cfg->stmt == NULL)
                        printf("  NODE: %d - - - - - - ",node_cfg->st_no);
                else
                        printf("  NODE: %d %s ",node_cfg->st_no,node_cfg->stmt);
                if (node_cfg->start_stmt != -1)
                        printf(" **%d",node_cfg->start_stmt);
        }
        node_cfg->flag = flag_val;
        if (node_cfg->what_stmt == 8)
                printf("\n Predicate %s is TRUE ",node_cfg->stmt);
        if (froml <= tol)
                print_PDW(node_cfg->left,flag_val);
        if (node_cfg->what_stmt == 8)
                printf("\n Predicate %s is FALSE ",node_cfg->stmt);
        if (fromr <= tor)
                print_PDW(node_cfg->right,flag_val);
}
```

APPENDIX C

INPUT/OUTPUT

The order of the input/output listing is as follows:

test1.c - This forms the input to the tool. This program accepts four variables and performs some random operations on them.
out1 - This forms the output produced by the tool.

test2.c - This is another test data for the tool. This program illustrates the switch statements used as an input.
out2 - This is the output of the test input test2.c.

test3.c - This is the third test data for the tool.
out3 - This is the output of the test input test3.c.

INPUT NO 1 : TEST1.c

```
#include <stdio.h>
main()
{
int i,j,k,l;
int h1,h2,h3,h4,h5;

scanf("%d %d %d %d",&i,&j,&k,&l);
while(i<10) {
      if (k == 1) {
            j = i;
            if (k == j) l = 5;
            else
                  l=6;
            }
      else k = k+2;
      while(k < 10) {
            if (k == i) l = l+4;
            l = l+34;
            }
      i = i+3;
      k = k+2;
      }
}
```

OUTPUT 1 : out1 -- This is the output to the above C program which was given as input to the tool. Each statement in test1.c is assigned a statement number as it can be seen below. Some of the nodes are the connecting or merge nodes, e.g., 2, 10, 12, and 13 (see Appendix D for a detailed example including graphical representations).

```
1 Input i##2,j##2,k##2,l##2
3 i##2<10
5 if k##2==l##2
6  j##3=i##2
7 if k##2==j##3
8  l##3=5
9  l##4=6
11  k##3=k##2+2
14 k##3<10
16 if k##3==i##2
17  l##5=l##4+4
19  l##6=l##5+34
21  i##3=i##2+3
22  k##4=k##3+2
24  END
```

The control flow graph as a linked list for the program test1.c follows. The numbers represent the nodes corresponding to the statement numbers and -1 represents the START node (see Appendix D for a detailed example including graphical representations).

```
-1 ------ 1
1 ------ 2
2 ------ 3
3 ------ 4
4 ------ 5
5 ------ 6
6 ------ 7
7 ------ 8
8 ------ 10
9 ------ 10
7 ------ 9
10 ------ 12
11 ------ 12
5 ------ 11
12 ------ 13
13 ------ 14
14 ------ 15
15 ------ 16
16 ------ 17
17 ------ 18
16 ------ 18
18 ------ 19
19 ------ 20
20 ------ 14
14 ------ 21
21 ------ 22
22 ------ 23
23 ------ 3
3 ------ 24
```

The control flow graph as a tree-structured directed graph for the program test1.c follows. The numbers represent the nodes corresponding to the statement numbers and -1 represent the START node (see Appendix D for a detailed example including graphical representations).

```
1 -----> 2   **** 0   NODE: 1 i##1,j##1,k##1,l##1
2 -----> 3   **** 1   NODE: 2 - - - - - -
3 -----> 4   **** 2   NODE: 3 i##1<10
3 -----> 24   **** 2   NODE: 3 i##1<10
4 -----> 5   **** 1   NODE: 4 - - - - - -
5 -----> 6   **** 1   NODE: 5 k##1==l##1
5 -----> 11   **** 1   NODE: 5 k##1==l##1
6 -----> 7   **** 1   NODE: 6 j##2=i##1
7 -----> 8   **** 1   NODE: 7 k##1==j##2
7 -----> 9   **** 1   NODE: 7 k##1==j##2
8 -----> 10   **** 1   NODE: 8 l##2=5
10 -----> 12   **** 2   NODE: 10 - - - - - -
12 -----> 13   **** 2   NODE: 12 - - - - - -
13 -----> 14   **** 1   NODE: 13 - - - - - -
14 -----> 15   **** 2   NODE: 14 k##1<10
14 -----> 21   **** 2   NODE: 14 k##1<10
15 -----> 16   **** 1   NODE: 15 - - - - - -
16 -----> 17   **** 1   NODE: 16 k##1==i##1
16 -----> 18   **** 1   NODE: 16 k##1==i##1
17 -----> 18   **** 1   NODE: 17 l##3=l##2+4
18 -----> 19   **** 2   NODE: 18 - - - - - -
19 -----> 20   **** 1   NODE: 19 l##4=l##3+34
20 -----> 14   **** 1   NODE: 20 - - - - - -
21 -----> 22   **** 1   NODE: 21 i##2=i##1+3
22 -----> 23   **** 1   NODE: 22 k##2=k##1+2
23 -----> 3   **** 1   NODE: 23 - - - - - -
9 -----> 10   **** 0   NODE: 9 l##5=6
11 -----> 12   **** 0   NODE: 11 k##3=k##1+2
```

The static single assignment form for the program test1.c

```
1 -----> 2   **** 0   NODE: 1 i##1,j##1,k##1,l##1
2 -----> 3   **** 1   NODE: 2 - - - - - -  **3
i##2 = PHI(i##1 1 1,i##3 3 21)
j##2 = PHI(j##1 1 1,j##4 4 12)
k##2 = PHI(k##1 1 1,k##4 4 22)
l##2 = PHI(l##1 1 1,l##6 6 14)
3 -----> 4   **** 2   NODE: 3 i##2<10
3 -----> 24   **** 2   NODE: 3 i##2<10
4 -----> 5   **** 1   NODE: 4 - - - - - -  **3
5 -----> 6   **** 1   NODE: 5 k##2==l##2
5 -----> 11   **** 1   NODE: 5 k##2==l##2
6 -----> 7   **** 1   NODE: 6 j##3=i##2
7 -----> 8   **** 1   NODE: 7 k##2==j##3
7 -----> 9   **** 1   NODE: 7 k##2==j##3
8 -----> 10   **** 1   NODE: 8 l##3=5
l##4 = PHI(l##3 3 8,l##10 10 9)
10 -----> 12   **** 2   NODE: 10 - - - - - -  **7
j##4 = PHI(j##3 3 7,j##2 2 3)
k##3 = PHI(k##2 2 7,k##5 5 11)
l##5 = PHI(l##4 4 10,l##2 2 5)
12 -----> 13   **** 2   NODE: 12 - - - - - -  **5
13 -----> 14   **** 1   NODE: 13 - - - - - -  **14
```

```
l##6 = PHI(l##5 5 12,l##9 9 19)
14 -----> 15   **** 2   NODE: 14 k##3<10
14 -----> 21   **** 2   NODE: 14 k##3<10
15 -----> 16   **** 1   NODE: 15 - - - - - - -  **14
16 -----> 17   **** 1   NODE: 16 k##3==i##2
16 -----> 18   **** 1   NODE: 16 k##3==i##2
17 -----> 18   **** 1   NODE: 17 l##7=l##6+4
l##8 = PHI(l##6 6 14,l##7 7 17)
18 -----> 19   **** 2   NODE: 18 - - - - - - -  **16
19 -----> 20   **** 1   NODE: 19 l##9=l##8+34
20 -----> 14   **** 1   NODE: 20 - - - - - - -  **14
21 -----> 22   **** 1   NODE: 21 i##3=i##2+3
22 -----> 23   **** 1   NODE: 22 k##4=k##3+2
23 -----> 3   **** 1   NODE: 23 - - - - - - -  **3
9 -----> 10   **** 0   NODE: 9 l##10=6
11 -----> 12   **** 0   NODE: 11 k##5=k##2+2
```

The gated single assignment form for the program test1.c

```
1 -----> 2   **** 0   NODE: 1 i##1,j##1,k##1,l##1
L = .TRUE
2 -----> 3   **** 1   NODE: 2 - - - - - - -  **3
i##2 = MU.L(3,i##1,i##3)
j##2 = MU.L(3,j##1,j##4)
k##2 = MU.L(3,k##1,k##4)
l##2 = MU.L(3,l##1,l##6)
3 -----> 4   **** 2   NODE: 3 i##2<10
3 -----> 24   **** 2   NODE: 3 i##2<10
L = .FALSE
4 -----> 5   **** 1   NODE: 4 - - - - - - -  **3
5 -----> 6   **** 1   NODE: 5 k##2==l##2
5 -----> 11   **** 1   NODE: 5 k##2==l##2
6 -----> 7   **** 1   NODE: 6 j##3=i##2
7 -----> 8   **** 1   NODE: 7 k##2==j##3
7 -----> 9   **** 1   NODE: 7 k##2==j##3
8 -----> 10   **** 1   NODE: 8 l##3=5
l##4 = GAMMA(7,l##3,l##10)
10 -----> 12   **** 2   NODE: 10 - - - - - - -  **7
j##4 = GAMMA(5,j##3,j##2)
k##3 = GAMMA(5,k##2,k##5)
l##5 = GAMMA(5,l##4,l##2)
12 -----> 13   **** 2   NODE: 12 - - - - - - -  **5
L = .TRUE
13 -----> 14   **** 1   NODE: 13 - - - - - - -  **14
l##6 = MU.L(14,l##5,l##9)
14 -----> 15   **** 2   NODE: 14 k##3<10
14 -----> 21   **** 2   NODE: 14 k##3<10
L = .FALSE
15 -----> 16   **** 1   NODE: 15 - - - - - - -  **14
16 -----> 17   **** 1   NODE: 16 k##3==i##2
16 -----> 18   **** 1   NODE: 16 k##3==i##2
17 -----> 18   **** 1   NODE: 17 l##7=l##6+4
l##8 = GAMMA(16,l##6,l##7)
18 -----> 19   **** 2   NODE: 18 - - - - - - -  **16
19 -----> 20   **** 1   NODE: 19 l##9=l##8+34
l##6 = ETA.T(14,l##9)
20 -----> 14   **** 1   NODE: 20 - - - - - - -  **14
21 -----> 22   **** 1   NODE: 21 i##3=i##2+3
22 -----> 23   **** 1   NODE: 22 k##4=k##3+2
i##2 = ETA.T(3,i##3)
```

```
j##2 = ETA.T(3,j##4)
k##2 = ETA.T(3,k##4)
l##2 = ETA.T(3,l##6)
23 -----> 3   **** 1   NODE: 23 - - - - - -  **3
9 -----> 10   **** 0   NODE: 9 l##10=6
11 -----> 12   **** 0   NODE: 11 k##5=k##2+2
```

The program dependence web for the program test1.c

```
1 -----> 2   **** 0   NODE: 1 i##1,j##1,k##1,l##1
L = .TRUE
2 -----> 3   **** 1   NODE: 2 - - - - - -  **3
i##2 = MU.L(3,i##1,i##3)
j##2 = MU.L(3,j##1,j##4)
k##2 = MU.L(3,k##1,k##4)
l##2 = MU.L(3,l##1,l##6)
3 -----> 4   **** 2   NODE: 3 i##2<10
3 -----> 24   **** 2   NODE: 3 i##2<10
L = .FALSE
4 -----> 5   **** 1   NODE: 4 - - - - - -  **3
5 -----> 6   **** 1   NODE: 5 k##2==l##2
5 -----> 11   **** 1   NODE: 5 k##2==l##2
Predicate k##2==l##2 is TRUE
6 -----> 7   **** 1   NODE: 6 j##3=i##2
7 -----> 8   **** 1   NODE: 7 k##2==j##3
7 -----> 9   **** 1   NODE: 7 k##2==j##3
Predicate k##2==j##3 is TRUE
8 -----> 10   **** 1   NODE: 8 l##3=5
l##4 = GAMMA(7,l##3,l##10)
10 -----> 12   **** 2   NODE: 10 - - - - - -  **7
j##4 = GAMMA(5,j##3,j##2)
k##3 = GAMMA(5,k##2,k##5)
l##5 = GAMMA(5,l##4,l##2)
12 -----> 13   **** 2   NODE: 12 - - - - - -  **5
L = .TRUE
13 -----> 14   **** 1   NODE: 13 - - - - - -  **14
l##6 = MU.L(14,l##5,l##9)
14 -----> 15   **** 2   NODE: 14 k##3<10
14 -----> 21   **** 2   NODE: 14 k##3<10
L = .FALSE
15 -----> 16   **** 1   NODE: 15 - - - - - -  **14
16 -----> 17   **** 1   NODE: 16 k##3==i##2
16 -----> 18   **** 1   NODE: 16 k##3==i##2
Predicate k##3==i##2 is TRUE
17 -----> 18   **** 1   NODE: 17 l##7=l##6+4
l##8 = GAMMA(16,l##6,l##7)
18 -----> 19   **** 2   NODE: 18 - - - - - -  **16
19 -----> 20   **** 1   NODE: 19 l##9=l##8+34
l##6 = ETA.T(14,l##9)
20 -----> 14   **** 1   NODE: 20 - - - - - -  **14
Predicate k##3==i##2 is FALSE
21 -----> 22   **** 1   NODE: 21 i##3=i##2+3
22 -----> 23   **** 1   NODE: 22 k##4=k##3+2
i##2 = ETA.T(3,i##3)
j##2 = ETA.T(3,j##4)
k##2 = ETA.T(3,k##4)
l##2 = ETA.T(3,l##6)
23 -----> 3   **** 1   NODE: 23 - - - - - -  **3
Predicate k##2==j##3 is FALSE
9 -----> 10   **** 0   NODE: 9 l##10=6
```

```
Predicate k##2==l##2 is FALSE
11 -----> 12   **** 0   NODE: 11 k##5=k##2+2
```

INPUT NO 2 : TEST2.c

```
#include <stdio.h>
main()
{
      int i,j,k,l;
      int h1,h2,h3,h4,h5;
      h1 = 10;
      h2 = 0;
      h3 = 0;
      scanf("%d %d %d %d",&i,&j,&k,&l);
      while(i<10) {
            for(k=0;k<10;k++)
                  h1 = h1 + 10;
            if (h1 > 100)
                  h2 = 1000;
            else
                  h3 = 500;

            switch(j)  {
            case 0 :
                  h4 = (h2 + h3) / h1;
                  break;

            case 1 :
                  h4 = (k + h2) / h1;
                  break;

            case 2 :
                  h4 = (i + k) / h1;
                  break;

            case 3 :
                  scanf("%d %d %d %d",&i,&j,&k,&l);
                  while(i<10) {
                        if (k == l) {
                              j = i;
                              if (k == j) l = 5;
                              else
                                    l=6;
                        }
                        else k = k+2;
                        while(k < 10) {
                              if (k == i) l = l+4;
                        }
                        i = i+3;
                  }
                  h4 = (i * k + h2 + h3) / h1;
                  break;

            default :
                  h4 = 10000;
                  break;
            }
            i = i+5;
      }
      printf("%d %d %d",i,j,h4);
}
```

OUTPUT 2 : out2 -- This is the output to the above C program which was given as input to the tool.

Each statement in test2.c is assigned a statement number as it could be seen below.

```
1  h1##2=10
2  h2##2=0
3  h3##2=0
4 Input i##2,j##2,k##2,l##2
6 i##2<10
8 k##3=0
10 k##3<10
12  h1##3=h1##2+10
13 k##4=k##3+1
15 if h1##3>100
16  h2##3=1000
17  h3##3=500
19 j##2 == 0
20  h4##2=(h2##3+h3##3)/h1##3
21 Each case Merge Node
22 j##2 == 1
23  h4##2=(k##4+h2##3)/h1##3
24 Each case Merge Node
25 j##2 == 2
26  h4##2=(i##2+k##4)/h1##3
27 Each case Merge Node
28 j##2 == 3
29 Input i##3,j##3,k##5,l##3
31 i##3<10
33 if k##5==l##3
34  j##4=i##3
35 if k##5==j##4
36  l##4=5
37  l##5=6
39  k##6=k##5+2
42 k##6<10
44 if k##6==i##3
45  l##6=l##5+4
48  i##4=i##3+3
50  h4##3=(i##4*k##6+h2##3+h3##3)/h1##3
51 Each case Merge Node
52 else
53  h4##2=10000
54 Merge Node
55  i##5=i##4+5
57 Output i##5,j##4,h4##3
58  END
```

The control flow graph as a linked list for the program test2.c

```
-1 ------ 1
1 ------ 2
2 ------ 3
3 ------ 4
4 ------ 5
5 ------ 6
6 ------ 7
```

7 ------ 8
8 ------ 9
9 ------ 10
10 ------ 11
11 ------ 12
12 ------ 13
10 ------ 15
13 ------ 14
14 ------ 10
15 ------ 16
16 ------ 18
17 ------ 18
15 ------ 17
18 ------ 19
19 ------ 20
20 ------ 21
21 ------ 54
19 ------ 22
22 ------ 23
23 ------ 24
24 ------ 54
22 ------ 25
25 ------ 26
26 ------ 27
27 ------ 54
25 ------ 28
28 ------ 29
29 ------ 30
30 ------ 31
31 ------ 32
32 ------ 33
33 ------ 34
34 ------ 35
35 ------ 36
36 ------ 38
37 ------ 38
35 ------ 37
38 ------ 40
39 ------ 40
33 ------ 39
40 ------ 41
41 ------ 42
42 ------ 43
43 ------ 44
44 ------ 45
45 ------ 46
44 ------ 46
46 ------ 47
47 ------ 42
42 ------ 48
48 ------ 49
49 ------ 31
31 ------ 50
50 ------ 51
51 ------ 54
28 ------ 52
52 ------ 53
53 ------ 54
54 ------ 55
55 ------ 56
56 ------ 6
6 ------ 57
57 ------ 58

The control flow graph for the program test2.c

```
Warning: text Variable Used Without Initialization
1 -----> 2   **** 0   NODE: 1 h1##1=10
2 -----> 3   **** 1   NODE: 2 h2##1=0
3 -----> 4   **** 1   NODE: 3 h3##1=0
4 -----> 5   **** 1   NODE: 4 i##1,j##1,k##1,l##1
5 -----> 6   **** 1   NODE: 5 - - - - - -
6 -----> 7   **** 2   NODE: 6 i##1<10
6 -----> 57   **** 2   NODE: 6 i##1<10
7 -----> 8   **** 1   NODE: 7 - - - - - -
8 -----> 9   **** 1   NODE: 8 k##2=0
9 -----> 10   **** 1   NODE: 9 - - - - - -
10 -----> 11   **** 2   NODE: 10 k##2<10
10 -----> 15   **** 2   NODE: 10 k##2<10
11 -----> 12   **** 1   NODE: 11 - - - - - -
12 -----> 13   **** 1   NODE: 12 h1##2=h1##1+10
13 -----> 14   **** 1   NODE: 13 k##3=k##2+1
14 -----> 10   **** 1   NODE: 14 - - - - - -
15 -----> 16   **** 1   NODE: 15 h1##1>100
15 -----> 17   **** 1   NODE: 15 h1##1>100
16 -----> 18   **** 1   NODE: 16 h2##2=1000
18 -----> 19   **** 2   NODE: 18 - - - - - -
19 -----> 20   **** 1   NODE: 19 j##1==0
19 -----> 22   **** 1   NODE: 19 j##1==0
20 -----> 21   **** 1   NODE: 20 h4##1=(h2##2+h3##1)/h1##1
21 -----> 54   **** 1   NODE: 21 - - - - - -
54 -----> 55   **** 5   NODE: 54 - - - - - -
55 -----> 56   **** 1   NODE: 55 i##2=i##1+5
56 -----> 6   **** 1   NODE: 56 - - - - - -
22 -----> 23   **** 1   NODE: 22 j##1==1
22 -----> 25   **** 1   NODE: 22 j##1==1
23 -----> 24   **** 1   NODE: 23 h4##2=(k##2+h2##2)/h1##1
24 -----> 54   **** 1   NODE: 24 - - - - - -
25 -----> 26   **** 1   NODE: 25 j##1==2
25 -----> 28   **** 1   NODE: 25 j##1==2
26 -----> 27   **** 1   NODE: 26 h4##3=(i##1+k##2)/h1##1
27 -----> 54   **** 1   NODE: 27 - - - - - -
28 -----> 29   **** 1   NODE: 28 j##1==3
28 -----> 52   **** 1   NODE: 28 j##1==3
29 -----> 30   **** 1   NODE: 29 i##3,j##2,k##4,l##2
30 -----> 31   **** 1   NODE: 30 - - - - - -
31 -----> 32   **** 2   NODE: 31 i##3<10
31 -----> 50   **** 2   NODE: 31 i##3<10
32 -----> 33   **** 1   NODE: 32 - - - - - -
33 -----> 34   **** 1   NODE: 33 k##4==l##2
33 -----> 39   **** 1   NODE: 33 k##4==l##2
34 -----> 35   **** 1   NODE: 34 j##3=i##3
35 -----> 36   **** 1   NODE: 35 k##4==j##3
35 -----> 37   **** 1   NODE: 35 k##4==j##3
36 -----> 38   **** 1   NODE: 36 l##3=5
38 -----> 40   **** 2   NODE: 38 - - - - - -
40 -----> 41   **** 2   NODE: 40 - - - - - -
41 -----> 42   **** 1   NODE: 41 - - - - - -
42 -----> 43   **** 2   NODE: 42 k##4<10
42 -----> 48   **** 2   NODE: 42 k##4<10
43 -----> 44   **** 1   NODE: 43 - - - - - -
44 -----> 45   **** 1   NODE: 44 k##4==i##3
44 -----> 46   **** 1   NODE: 44 k##4==i##3
45 -----> 46   **** 1   NODE: 45 l##4=l##3+4
46 -----> 47   **** 2   NODE: 46 - - - - - -
47 -----> 42   **** 1   NODE: 47 - - - - - -
48 -----> 49   **** 1   NODE: 48 i##4=i##3+3
49 -----> 31   **** 1   NODE: 49 - - - - - -
```

```
37 -----> 38   **** 0   NODE: 37 l##5=6
39 -----> 40   **** 0   NODE: 39 k##5=k##4+2
50 -----> 51   **** 1   NODE: 50 h4##4=(i##3*k##4+h2##2+h3##1)/h1##1
51 -----> 54   **** 1   NODE: 51 - - - - - - -
52 -----> 53   **** 1   NODE: 52 - - - - - - -
53 -----> 54   **** 1   NODE: 53 h4##5=10000
17 -----> 18   **** 0   NODE: 17 h3##2=500
57 -----> 58   **** 1   NODE: 57 i##1,j##1,h4##0
```

The static single assignment form for the program test2.c

```
1 -----> 2   **** 0   NODE: 1 h1##1=10
2 -----> 3   **** 1   NODE: 2 h2##1=0
3 -----> 4   **** 1   NODE: 3 h3##1=0
4 -----> 5   **** 1   NODE: 4 i##1,j##1,k##1,l##1
5 -----> 6   **** 1   NODE: 5 - - - - - - -  **6
h2##2 = PHI(h2##1 1 2,h2##4 4 20)
i##2 = PHI(i##1 1 4,i##4 4 55)
k##2 = PHI(k##1 1 4,k##6 6 54)
h1##2 = PHI(h1##1 1 1,h1##3 3 20)
h3##2 = PHI(h3##1 1 3,h3##3 3 20)
h4##1 = PHI(h4##0 0 0,h4##3 3 54)
j##2 = PHI(j##1 1 4,j##3 3 54)
l##2 = PHI(l##1 1 4,l##3 3 54)
6 -----> 7   **** 2   NODE: 6 i##2<10
6 -----> 57   **** 2   NODE: 6 i##2<10
7 -----> 8   **** 1   NODE: 7 - - - - - - -  **6
8 -----> 9   **** 1   NODE: 8 k##3=0  **10
9 -----> 10   **** 1   NODE: 9 - - - - - - -  **10
h1##3 = PHI(h1##2 2 6,h1##4 4 12)
k##4 = PHI(k##3 3 8,k##5 5 13)
10 -----> 11   **** 2   NODE: 10 k##4<10
10 -----> 15   **** 2   NODE: 10 k##4<10
11 -----> 12   **** 1   NODE: 11 - - - - - - -  **10
12 -----> 13   **** 1   NODE: 12 h1##4=h1##3+10
13 -----> 14   **** 1   NODE: 13 k##5=k##4+1  **11
14 -----> 10   **** 1   NODE: 14 - - - - - - -  **11
15 -----> 16   **** 1   NODE: 15 h1##3>100
15 -----> 17   **** 1   NODE: 15 h1##3>100
16 -----> 18   **** 1   NODE: 16 h2##3=1000
h2##4 = PHI(h2##3 3 16,h2##2 2 6)
h3##3 = PHI(h3##2 2 6,h3##4 4 17)
18 -----> 19   **** 2   NODE: 18 - - - - - - -  **15
19 -----> 20   **** 1   NODE: 19 j##2==0
19 -----> 22   **** 1   NODE: 19 j##2==0
20 -----> 21   **** 1   NODE: 20 h4##2=(h2##4+h3##3)/h1##3
21 -----> 54   **** 1   NODE: 21 - - - - - - -  **19
h4##3 = PHI(h4##2 2 20,h4##4 4 23,h4##5 5 26,h4##6 6 50,h4##7 7 53)
i##3 = PHI(i##2 2 6,i##2 2 6,i##2 2 26,i##6 6 50,i##2 2 6)
j##3 = PHI(j##2 2 19,j##2 2 22,j##2 2 25,j##5 5 31,j##2 2 28)
k##6 = PHI(k##4 4 10,k##4 4 23,k##4 4 26,k##8 8 50,k##4 4 10)
l##3 = PHI(l##2 2 6,l##2 2 6,l##2 2 6,l##5 5 31,l##2 2 6)
54 -----> 55   **** 5   NODE: 54 - - - - - - -  **19
55 -----> 56   **** 1   NODE: 55 i##4=i##3+5
56 -----> 6   **** 1   NODE: 56 - - - - - - -  **6
22 -----> 23   **** 1   NODE: 22 j##2==1
22 -----> 25   **** 1   NODE: 22 j##2==1
23 -----> 24   **** 1   NODE: 23 h4##4=(k##4+h2##4)/h1##3
24 -----> 54   **** 1   NODE: 24 - - - - - - -  **22
25 -----> 26   **** 1   NODE: 25 j##2==2
25 -----> 28   **** 1   NODE: 25 j##2==2
```

```
26 -----> 27   **** 1   NODE: 26 h4##5=(i##2+k##4)/h1##3
27 -----> 54   **** 1   NODE: 27 - - - - - - -  **25
28 -----> 29   **** 1   NODE: 28 j##2==3
28 -----> 52   **** 1   NODE: 28 j##2==3
29 -----> 30   **** 1   NODE: 29 i##5,j##4,k##7,l##4
30 -----> 31   **** 1   NODE: 30 - - - - - - -  **31
i##6 = PHI(i##5 5 29,i##7 7 48)
j##5 = PHI(j##4 4 29,j##7 7 40)
l##5 = PHI(l##4 4 29,l##9 9 42)
k##8 = PHI(k##7 7 29,k##9 9 42)
31 -----> 32   **** 2   NODE: 31 i##6<10
31 -----> 50   **** 2   NODE: 31 i##6<10
32 -----> 33   **** 1   NODE: 32 - - - - - - -  **31
33 -----> 34   **** 1   NODE: 33 k##8==l##5
33 -----> 39   **** 1   NODE: 33 k##8==l##5
34 -----> 35   **** 1   NODE: 34 j##6=i##6
35 -----> 36   **** 1   NODE: 35 k##8==j##6
35 -----> 37   **** 1   NODE: 35 k##8==j##6
36 -----> 38   **** 1   NODE: 36 l##6=5
l##7 = PHI(l##6 6 36,l##12 12 37)
38 -----> 40   **** 2   NODE: 38 - - - - - - -  **35
j##7 = PHI(j##6 6 35,j##5 5 31)
k##9 = PHI(k##8 8 35,k##10 10 39)
l##8 = PHI(l##7 7 38,l##5 5 33)
40 -----> 41   **** 2   NODE: 40 - - - - - - -  **33
41 -----> 42   **** 1   NODE: 41 - - - - - - -  **42
l##9 = PHI(l##8 8 40,l##11 11 46)
42 -----> 43   **** 2   NODE: 42 k##9<10
42 -----> 48   **** 2   NODE: 42 k##9<10
43 -----> 44   **** 1   NODE: 43 - - - - - - -  **42
44 -----> 45   **** 1   NODE: 44 k##9==i##6
44 -----> 46   **** 1   NODE: 44 k##9==i##6
45 -----> 46   **** 1   NODE: 45 l##10=l##9+4
l##11 = PHI(l##9 9 42,l##10 10 45)
46 -----> 47   **** 2   NODE: 46 - - - - - - -  **44
47 -----> 42   **** 1   NODE: 47 - - - - - - -  **42
48 -----> 49   **** 1   NODE: 48 i##7=i##6+3
49 -----> 31   **** 1   NODE: 49 - - - - - - -  **31
37 -----> 38   **** 0   NODE: 37 l##12=6
39 -----> 40   **** 0   NODE: 39 k##10=k##8+2
50 -----> 51   **** 1   NODE: 50 h4##6=(i##6*k##8+h2##4+h3##3)/h1##3
51 -----> 54   **** 1   NODE: 51 - - - - - - -  **28
52 -----> 53   **** 1   NODE: 52 - - - - - - -
53 -----> 54   **** 1   NODE: 53 h4##7=10000
17 -----> 18   **** 0   NODE: 17 h3##4=500
57 -----> 58   **** 1   NODE: 57 i##2,j##2,h4##1
```

The gated single assignment form for the program test2.c

```
1 -----> 2   **** 0   NODE: 1 h1##1=10
2 -----> 3   **** 1   NODE: 2 h2##1=0
3 -----> 4   **** 1   NODE: 3 h3##1=0
4 -----> 5   **** 1   NODE: 4 i##1,j##1,k##1,l##1
L = .TRUE
5 -----> 6   **** 1   NODE: 5 - - - - - - -  **6
h2##2 = MU.L(6,h2##1,h2##4)
i##2 = MU.L(6,i##1,i##4)
k##2 = MU.L(6,k##1,k##6)
h1##2 = MU.L(6,h1##1,h1##3)
h3##2 = MU.L(6,h3##1,h3##3)
h4##1 = MU.L(6,h4##0,h4##3)
```

```
 j##2 = MU.L(6,j##1,j##3)
 l##2 = MU.L(6,l##1,l##3)
 6 -----> 7   **** 2   NODE: 6 i##2<10
 6 -----> 57   **** 2   NODE: 6 i##2<10
 L = .FALSE
 7 -----> 8   **** 1   NODE: 7 - - - - - -  **6
 8 -----> 9   **** 1   NODE: 8 k##3=0  **10
 L = .TRUE
 9 -----> 10   **** 1   NODE: 9 - - - - - -  **10
 h1##3 = MU.L(10,h1##2,h1##4)
 k##4 = MU.L(10,k##3,k##5)
 10 -----> 11   **** 2   NODE: 10 k##4<10
 10 -----> 15   **** 2   NODE: 10 k##4<10
 L = .FALSE
 11 -----> 12   **** 1   NODE: 11 - - - - - -  **10
 12 -----> 13   **** 1   NODE: 12 h1##4=h1##3+10
 13 -----> 14   **** 1   NODE: 13 k##5=k##4+1  **11
 h1##3 = ETA.T(11,h1##4)
 k##4 = ETA.T(11,k##5)
 14 -----> 10   **** 1   NODE: 14 - - - - - -  **11
 15 -----> 16   **** 1   NODE: 15 h1##3>100
 15 -----> 17   **** 1   NODE: 15 h1##3>100
 16 -----> 18   **** 1   NODE: 16 h2##3=1000
 h2##4 = GAMMA(15,h2##3,h2##2)
 h3##3 = GAMMA(15,h3##2,h3##4)
 18 -----> 19   **** 2   NODE: 18 - - - - - -  **15
 19 -----> 20   **** 1   NODE: 19 j##2==0
 19 -----> 22   **** 1   NODE: 19 j##2==0
 20 -----> 21   **** 1   NODE: 20 h4##2=(h2##4+h3##3)/h1##3
 21 -----> 54   **** 1   NODE: 21 - - - - - -  **19
 h4##3 = GAMMA(19,h4##2, GAMMA(22,h4##4, GAMMA(25,h4##5,
GAMMA(28,h4##6,h4##7))))
 i##3 = GAMMA(19,i##2, GAMMA(22,i##2, GAMMA(25,i##2,
GAMMA(28,i##6,i##2))))
 j##3 = GAMMA(19,j##2, GAMMA(22,j##2, GAMMA(25,j##2,
GAMMA(28,j##5,j##2))))
 k##6 = GAMMA(19,k##4, GAMMA(22,k##4, GAMMA(25,k##4,
GAMMA(28,k##8,k##4))))
 l##3 = GAMMA(19,l##2, GAMMA(22,l##2, GAMMA(25,l##2,
GAMMA(28,l##5,l##2))))
 54 -----> 55   **** 5   NODE: 54 - - - - - -  **19
 55 -----> 56   **** 1   NODE: 55 i##4=i##3+5
 h2##2 = ETA.T(6,h2##4)
 i##2 = ETA.T(6,i##4)
 k##2 = ETA.T(6,k##6)
 h1##2 = ETA.T(6,h1##3)
 h3##2 = ETA.T(6,h3##3)
 h4##1 = ETA.T(6,h4##3)
 j##2 = ETA.T(6,j##3)
 l##2 = ETA.T(6,l##3)
 56 -----> 6   **** 1   NODE: 56 - - - - - -  **6
 22 -----> 23   **** 1   NODE: 22 j##2==1
 22 -----> 25   **** 1   NODE: 22 j##2==1
 23 -----> 24   **** 1   NODE: 23 h4##4=(k##4+h2##4)/h1##3
 24 -----> 54   **** 1   NODE: 24 - - - - - -  **22
 25 -----> 26   **** 1   NODE: 25 j##2==2
 25 -----> 28   **** 1   NODE: 25 j##2==2
 26 -----> 27   **** 1   NODE: 26 h4##5=(i##2+k##4)/h1##3
 27 -----> 54   **** 1   NODE: 27 - - - - - -  **25
 28 -----> 29   **** 1   NODE: 28 j##2==3
 28 -----> 52   **** 1   NODE: 28 j##2==3
 29 -----> 30   **** 1   NODE: 29 i##5,j##4,k##7,l##4
 L = .TRUE
 30 -----> 31   **** 1   NODE: 30 - - - - - -  **31
 i##6 = MU.L(31,i##5,i##7)
```

```
j##5 = MU.L(31,j##4,j##7)
l##5 = MU.L(31,l##4,l##9)
k##8 = MU.L(31,k##7,k##9)
31 -----> 32   **** 2   NODE: 31 i##6<10
31 -----> 50   **** 2   NODE: 31 i##6<10
L = .FALSE
32 -----> 33   **** 1   NODE: 32 - - - - - -  **31
33 -----> 34   **** 1   NODE: 33 k##8==l##5
33 -----> 39   **** 1   NODE: 33 k##8==l##5
34 -----> 35   **** 1   NODE: 34 j##6=i##6
35 -----> 36   **** 1   NODE: 35 k##8==j##6
35 -----> 37   **** 1   NODE: 35 k##8==j##6
36 -----> 38   **** 1   NODE: 36 l##6=5
l##7 = GAMMA(35,l##6,l##12)
38 -----> 40   **** 2   NODE: 38 - - - - - -  **35
j##7 = GAMMA(33,j##6,j##5)
k##9 = GAMMA(33,k##8,k##10)
l##8 = GAMMA(33,l##7,l##5)
40 -----> 41   **** 2   NODE: 40 - - - - - -  **33
L = .TRUE
41 -----> 42   **** 1   NODE: 41 - - - - - -  **42
l##9 = MU.L(42,l##8,l##11)
42 -----> 43   **** 2   NODE: 42 k##9<10
42 -----> 48   **** 2   NODE: 42 k##9<10
L = .FALSE
43 -----> 44   **** 1   NODE: 43 - - - - - -  **42
44 -----> 45   **** 1   NODE: 44 k##9==i##6
44 -----> 46   **** 1   NODE: 44 k##9==i##6
45 -----> 46   **** 1   NODE: 45 l##10=l##9+4
l##11 = GAMMA(44,l##9,l##10)
46 -----> 47   **** 2   NODE: 46 - - - - - -  **44
l##9 = ETA.T(42,l##11)
47 -----> 42   **** 1   NODE: 47 - - - - - -  **42
48 -----> 49   **** 1   NODE: 48 i##7=i##6+3
i##6 = ETA.T(31,i##7)
j##5 = ETA.T(31,j##7)
l##5 = ETA.T(31,l##9)
k##8 = ETA.T(31,k##9)
49 -----> 31   **** 1   NODE: 49 - - - - - -  **31
37 -----> 38   **** 0   NODE: 37 l##12=6
39 -----> 40   **** 0   NODE: 39 k##10=k##8+2
50 -----> 51   **** 1   NODE: 50 h4##6=(i##6*k##8+h2##4+h3##3)/h1##3
51 -----> 54   **** 1   NODE: 51 - - - - - -  **28
52 -----> 53   **** 1   NODE: 52 - - - - - -
53 -----> 54   **** 1   NODE: 53 h4##7=10000
17 -----> 18   **** 0   NODE: 17 h3##4=500
57 -----> 58   **** 1   NODE: 57 i##2,j##2,h4##1
```

The program dependence web for the program test2.c

```
1 -----> 2   **** 0   NODE: 1 h1##1=10
2 -----> 3   **** 1   NODE: 2 h2##1=0
3 -----> 4   **** 1   NODE: 3 h3##1=0
4 -----> 5   **** 1   NODE: 4 i##1,j##1,k##1,l##1
L = .TRUE
5 -----> 6   **** 1   NODE: 5 - - - - - -  **6
h2##2 = MU.L(6,h2##1,h2##4)
i##2 = MU.L(6,i##1,i##4)
k##2 = MU.L(6,k##1,k##6)
h1##2 = MU.L(6,h1##1,h1##3)
```

```
h3##2 = MU.L(6,h3##1,h3##3)
h4##1 = MU.L(6,h4##0,h4##3)
j##2 = MU.L(6,j##1,j##3)
l##2 = MU.L(6,l##1,l##3)
6 -----> 7   **** 2   NODE: 6 i##2<10
6 -----> 57   **** 2   NODE: 6 i##2<10
L = .FALSE
7 -----> 8   **** 1   NODE: 7 - - - - - -  **6
8 -----> 9   **** 1   NODE: 8 k##3=0  **10
L = .TRUE
9 -----> 10   **** 1   NODE: 9 - - - - - -  **10
h1##3 = MU.L(10,h1##2,h1##4)
k##4 = MU.L(10,k##3,k##5)
10 -----> 11   **** 2   NODE: 10 k##4<10
10 -----> 15   **** 2   NODE: 10 k##4<10
L = .FALSE
11 -----> 12   **** 1   NODE: 11 - - - - - -  **10
12 -----> 13   **** 1   NODE: 12 h1##4=h1##3+10
13 -----> 14   **** 1   NODE: 13 k##5=k##4+1  **11
h1##3 = ETA.T(11,h1##4)
k##4 = ETA.T(11,k##5)
14 -----> 10   **** 1   NODE: 14 - - - - - -  **11
15 -----> 16   **** 1   NODE: 15 h1##3>100
15 -----> 17   **** 1   NODE: 15 h1##3>100
Predicate h1##3>100 is TRUE
16 -----> 18   **** 1   NODE: 16 h2##3=1000
h2##4 = GAMMA(15,h2##3,h2##2)
h3##3 = GAMMA(15,h3##2,h3##4)
18 -----> 19   **** 2   NODE: 18 - - - - - -  **15
19 -----> 20   **** 1   NODE: 19 j##2==0
19 -----> 22   **** 1   NODE: 19 j##2==0
20 -----> 21   **** 1   NODE: 20 h4##2=(h2##4+h3##3)/h1##3
21 -----> 54   **** 1   NODE: 21 - - - - - -  **19
h4##3 = GAMMA(19,h4##2, GAMMA(22,h4##4, GAMMA(25,h4##5,
GAMMA(28,h4##6,h4##7))))
i##3 = GAMMA(19,i##2, GAMMA(22,i##2, GAMMA(25,i##2,
GAMMA(28,i##6,i##2))))
j##3 = GAMMA(19,j##2, GAMMA(22,j##2, GAMMA(25,j##2,
GAMMA(28,j##5,j##2))))
k##6 = GAMMA(19,k##4, GAMMA(22,k##4, GAMMA(25,k##4,
GAMMA(28,k##8,k##4))))
l##3 = GAMMA(19,l##2, GAMMA(22,l##2, GAMMA(25,l##2,
GAMMA(28,l##5,l##2))))
54 -----> 55   **** 5   NODE: 54 - - - - - -  **19
55 -----> 56   **** 1   NODE: 55 i##4=i##3+5
h2##2 = ETA.T(6,h2##4)
i##2 = ETA.T(6,i##4)
k##2 = ETA.T(6,k##6)
h1##2 = ETA.T(6,h1##3)
h3##2 = ETA.T(6,h3##3)
h4##1 = ETA.T(6,h4##3)
j##2 = ETA.T(6,j##3)
l##2 = ETA.T(6,l##3)
56 -----> 6   **** 1   NODE: 56 - - - - - -  **6
22 -----> 23   **** 1   NODE: 22 j##2==1
22 -----> 25   **** 1   NODE: 22 j##2==1
23 -----> 24   **** 1   NODE: 23 h4##4=(k##4+h2##4)/h1##3
24 -----> 54   **** 1   NODE: 24 - - - - - -  **22
25 -----> 26   **** 1   NODE: 25 j##2==2
25 -----> 28   **** 1   NODE: 25 j##2==2
26 -----> 27   **** 1   NODE: 26 h4##5=(i##2+k##4)/h1##3
27 -----> 54   **** 1   NODE: 27 - - - - - -  **25
28 -----> 29   **** 1   NODE: 28 j##2==3
28 -----> 52   **** 1   NODE: 28 j##2==3
29 -----> 30   **** 1   NODE: 29 i##5,j##4,k##7,l##4
```

```
L = .TRUE
30 -----> 31   **** 1   NODE: 30 - - - - - -  **31
i##6 = MU.L(31,i##5,i##7)
j##5 = MU.L(31,j##4,j##7)
l##5 = MU.L(31,l##4,l##9)
k##8 = MU.L(31,k##7,k##9)
31 -----> 32   **** 2   NODE: 31 i##6<10
31 -----> 50   **** 2   NODE: 31 i##6<10
L = .FALSE
32 -----> 33   **** 1   NODE: 32 - - - - - -  **31
33 -----> 34   **** 1   NODE: 33 k##8==l##5
33 -----> 39   **** 1   NODE: 33 k##8==l##5
Predicate k##8==l##5 is TRUE
34 -----> 35   **** 1   NODE: 34 j##6=i##6
35 -----> 36   **** 1   NODE: 35 k##8==j##6
35 -----> 37   **** 1   NODE: 35 k##8==j##6
Predicate k##8==j##6 is TRUE
36 -----> 38   **** 1   NODE: 36 l##6=5
l##7 = GAMMA(35,l##6,l##12)
38 -----> 40   **** 2   NODE: 38 - - - - - -  **35
j##7 = GAMMA(33,j##6,j##5)
k##9 = GAMMA(33,k##8,k##10)
l##8 = GAMMA(33,l##7,l##5)
40 -----> 41   **** 2   NODE: 40 - - - - - -  **33
L = .TRUE
41 -----> 42   **** 1   NODE: 41 - - - - - -  **42
l##9 = MU.L(42,l##8,l##11)
42 -----> 43   **** 2   NODE: 42 k##9<10
42 -----> 48   **** 2   NODE: 42 k##9<10
L = .FALSE
43 -----> 44   **** 1   NODE: 43 - - - - - -  **42
44 -----> 45   **** 1   NODE: 44 k##9==i##6
44 -----> 46   **** 1   NODE: 44 k##9==i##6
Predicate k##9==i##6 is TRUE
45 -----> 46   **** 1   NODE: 45 l##10=l##9+4
l##11 = GAMMA(44,l##9,l##10)
46 -----> 47   **** 2   NODE: 46 - - - - - -  **44
l##9 = ETA.T(42,l##11)
47 -----> 42   **** 1   NODE: 47 - - - - - -  **42
Predicate k##9==i##6 is FALSE
48 -----> 49   **** 1   NODE: 48 i##7=i##6+3
i##6 = ETA.T(31,i##7)
j##5 = ETA.T(31,j##7)
l##5 = ETA.T(31,l##9)
k##8 = ETA.T(31,k##9)
49 -----> 31   **** 1   NODE: 49 - - - - - -  **31
Predicate k##8==j##6 is FALSE
37 -----> 38   **** 0   NODE: 37 l##12=6
Predicate k##8==l##5 is FALSE
39 -----> 40   **** 0   NODE: 39 k##10=k##8+2
50 -----> 51   **** 1   NODE: 50 h4##6=(i##6*k##8+h2##4+h3##3)/h1##3
51 -----> 54   **** 1   NODE: 51 - - - - - -  **28
52 -----> 53   **** 1   NODE: 52 - - - - - -
53 -----> 54   **** 1   NODE: 53 h4##7=10000
Predicate h1##3>100 is FALSE
17 -----> 18   **** 0   NODE: 17 h3##4=500
57 -----> 58   **** 1   NODE: 57 i##2,j##2,h4##1
```

INPUT NO 3: TEST3.C

```
#include <stdio.h>
main()
{
      int i,j,k,l;
      int h1,h2,h3,h4,h5;
      h1 = 10;
      h2 = 0;
      h3 = 0;
      h4 = 0;
      scanf("%d %d %d %d",&i,&j,&k,&l);
      while(i<10) {
            for(k=0;k<10;k++)
                  h1 = h1 + 10;
            if (h1 > 100)
                  h2 = 1000;
            else
                  h3 = 500;

            switch(j)  {
            case 0 :
                  h4 = (h2 + h3) / h1;
                  break;

            case 1 :
                  h4 = (k + h2) / h1;
                  break;

            case 2 :
                  h4 = (i + k) / h1;
                  for(i=0;i<10;i++)
                        for(j=0;j<10;j++)
                              for(k = 0;k<19;k++) {
                                    h1 = i+j;
                                    h2 = k+j;
                                    h3 = k*j*i;
                                    }
                  break;
            case 3 :
                  scanf("%d %d %d %d",&i,&j,&k,&l);
                  while(i<10) {
                        if (k == 1) {
                              j = i;
                              if (k == j) l = 5;
                              else
                                    l=6;
                        }
                        else k = k+2;
                        while(k < 10) {
                              if (k == i) l = l+4;
                        }
                        i = i+3;
                  }
                  h4 = (i * k + h2 + h3) / h1;
                  break;

            default :
                  h4 = 10000;
                  break;
            }
            i = i+5;
      }
      printf("%d %d %d",i,j,h4);
}
```

OUTPUT 3: out3 - The following is the output to the above C program which was given as input to the tool.

```
1  h1##2=10
2  h2##2=0
3  h3##2=0
4  h4##2=0
5 Input i##2,j##2,k##2,l##2
7 i##2<10
9 k##3=0
11 k##3<10
13  h1##3=h1##2+10
14 k##4=k##3+1
16 if h1##3>100
17  h2##3=1000
18  h3##3=500
20 j##2 == 0
21  h4##3=(h2##3+h3##3)/h1##3
22 Each case Merge Node
23 j##2 == 1
24  h4##4=(k##4+h2##3)/h1##3
25 Each case Merge Node
26 j##2 == 2
27  h4##5=(i##2+k##4)/h1##3
28 i##3=0
30 i##3<10
32 j##3=0
34 j##3<10
36 k##5=0
38 k##5<19
40  h1##4=i##3+j##3
41  h2##4=k##5+j##3
42  h3##4=k##5*j##3*i##3
43 k##6=k##5+1
45 j##4=j##3+1
47 i##4=i##3+1
49 Each case Merge Node
50 j##2 == 3
51 Input i##5,j##5,k##7,l##3
53 i##5<10
55 if k##7==l##3
56  j##6=i##5
57 if k##7==j##6
58  l##4=5
59  l##5=6
61  k##8=k##7+2
64 k##8<10
66 if k##8==i##5
67  l##6=l##5+4
70  i##6=i##5+3
72  h4##6=(i##6*k##8+h2##4+h3##4)/h1##4
73 Each case Merge Node
74 else
75  h4##7=10000
76 Merge Node
77  i##7=i##6+5
79 Output i##7,j##6,h4##7
80  END
```

The control flow graph as a linked list for the program test3.c

```
-1 ------ 1
1 ------ 2
2 ------ 3
3 ------ 4
4 ------ 5
5 ------ 6
6 ------ 7
7 ------ 8
8 ------ 9
9 ------ 10
10 ------ 11
11 ------ 12
12 ------ 13
13 ------ 14
11 ------ 16
14 ------ 15
15 ------ 11
16 ------ 17
17 ------ 19
18 ------ 19
16 ------ 18
19 ------ 20
20 ------ 21
21 ------ 22
22 ------ 76
20 ------ 23
23 ------ 24
24 ------ 25
25 ------ 76
23 ------ 26
26 ------ 27
27 ------ 28
28 ------ 29
29 ------ 30
30 ------ 31
31 ------ 32
32 ------ 33
33 ------ 34
34 ------ 35
35 ------ 36
36 ------ 37
37 ------ 38
38 ------ 39
39 ------ 40
40 ------ 41
41 ------ 42
42 ------ 43
38 ------ 45
43 ------ 44
44 ------ 38
34 ------ 47
45 ------ 46
46 ------ 34
30 ------ 49
47 ------ 48
48 ------ 30
49 ------ 76
26 ------ 50
50 ------ 51
51 ------ 52
52 ------ 53
53 ------ 54
```

```
54 ------ 55
55 ------ 56
56 ------ 57
57 ------ 58
58 ------ 60
59 ------ 60
57 ------ 59
60 ------ 62
61 ------ 62
55 ------ 61
62 ------ 63
63 ------ 64
64 ------ 65
65 ------ 66
66 ------ 67
67 ------ 68
66 ------ 68
68 ------ 69
69 ------ 64
64 ------ 70
70 ------ 71
71 ------ 53
53 ------ 72
72 ------ 73
73 ------ 76
50 ------ 74
74 ------ 75
75 ------ 76
76 ------ 77
77 ------ 78
78 ------ 7
7 ------ 79
79 ------ 80
```

The control flow graph for the program test3.c

```
1 -----> 2    **** 0   NODE: 1 h1##1=10
2 -----> 3    **** 1    NODE: 2 h2##1=0
3 -----> 4    **** 1    NODE: 3 h3##1=0
4 -----> 5    **** 1    NODE: 4 h4##1=0
5 -----> 6    **** 1    NODE: 5 i##1,j##1,k##1,l##1
6 -----> 7    **** 1    NODE: 6 - - - - - -
7 -----> 8    **** 2    NODE: 7 i##1<10
7 -----> 79    **** 2    NODE: 7 i##1<10
8 -----> 9    **** 1   NODE: 8 - - - - - -
9 -----> 10    **** 1    NODE: 9 k##2=0
10 -----> 11    **** 1    NODE: 10 - - - - - -
11 -----> 12    **** 2    NODE: 11 k##2<10
11 -----> 16    **** 2    NODE: 11 k##2<10
12 -----> 13    **** 1    NODE: 12 - - - - - -
13 -----> 14    **** 1    NODE: 13 h1##2=h1##1+10
14 -----> 15    **** 1    NODE: 14 k##3=k##2+1
15 -----> 11    **** 1    NODE: 15 - - - - - -
16 -----> 17    **** 1    NODE: 16 h1##1>100
16 -----> 18    **** 1    NODE: 16 h1##1>100
17 -----> 19    **** 1    NODE: 17 h2##2=1000
19 -----> 20    **** 2    NODE: 19 - - - - - -
20 -----> 21    **** 1    NODE: 20 j##1==0
20 -----> 23    **** 1    NODE: 20 j##1==0
21 -----> 22    **** 1    NODE: 21 h4##2=(h2##2+h3##1)/h1##1
```

```
22 -----> 76   **** 1   NODE: 22 - - - - - -
76 -----> 77   **** 5   NODE: 76 - - - - - -
77 -----> 78   **** 1   NODE: 77 i##2=i##1+5
78 -----> 7   **** 1   NODE: 78 - - - - - -
23 -----> 24   **** 1   NODE: 23 j##1==1
23 -----> 26   **** 1   NODE: 23 j##1==1
24 -----> 25   **** 1   NODE: 24 h4##3=(k##2+h2##2)/h1##1
25 -----> 76   **** 1   NODE: 25 - - - - - -
26 -----> 27   **** 1   NODE: 26 j##1==2
26 -----> 50   **** 1   NODE: 26 j##1==2
27 -----> 28   **** 1   NODE: 27 h4##4=(i##1+k##2)/h1##1
28 -----> 29   **** 1   NODE: 28 i##3=0
29 -----> 30   **** 1   NODE: 29 - - - - - -
30 -----> 31   **** 2   NODE: 30 i##3<10
30 -----> 49   **** 2   NODE: 30 i##3<10
31 -----> 32   **** 1   NODE: 31 - - - - - -
32 -----> 33   **** 1   NODE: 32 j##2=0
33 -----> 34   **** 1   NODE: 33 - - - - - -
34 -----> 35   **** 2   NODE: 34 j##2<10
34 -----> 47   **** 2   NODE: 34 j##2<10
35 -----> 36   **** 1   NODE: 35 - - - - - -
36 -----> 37   **** 1   NODE: 36 k##4=0
37 -----> 38   **** 1   NODE: 37 - - - - - -
38 -----> 39   **** 2   NODE: 38 k##4<19
38 -----> 45   **** 2   NODE: 38 k##4<19
39 -----> 40   **** 1   NODE: 39 - - - - - -
40 -----> 41   **** 1   NODE: 40 h1##3=i##3+j##2
41 -----> 42   **** 1   NODE: 41 h2##3=k##4+j##2
42 -----> 43   **** 1   NODE: 42 h3##2=k##4*j##2*i##3
43 -----> 44   **** 1   NODE: 43 k##5=k##4+1
44 -----> 38   **** 1   NODE: 44 - - - - - -
45 -----> 46   **** 1   NODE: 45 j##3=j##2+1
46 -----> 34   **** 1   NODE: 46 - - - - - -
47 -----> 48   **** 1   NODE: 47 i##4=i##3+1
48 -----> 30   **** 1   NODE: 48 - - - - - -
49 -----> 76   **** 1   NODE: 49 - - - - - -
50 -----> 51   **** 1   NODE: 50 j##1==3
50 -----> 74   **** 1   NODE: 50 j##1==3
51 -----> 52   **** 1   NODE: 51 i##5,j##4,k##6,l##2
52 -----> 53   **** 1   NODE: 52 - - - - - -
53 -----> 54   **** 2   NODE: 53 i##5<10
53 -----> 72   **** 2   NODE: 53 i##5<10
54 -----> 55   **** 1   NODE: 54 - - - - - -
55 -----> 56   **** 1   NODE: 55 k##6==l##2
55 -----> 61   **** 1   NODE: 55 k##6==l##2
56 -----> 57   **** 1   NODE: 56 j##5=i##5
57 -----> 58   **** 1   NODE: 57 k##6==j##5
57 -----> 59   **** 1   NODE: 57 k##6==j##5
58 -----> 60   **** 1   NODE: 58 l##3=5
60 -----> 62   **** 2   NODE: 60 - - - - - -
62 -----> 63   **** 2   NODE: 62 - - - - - -
63 -----> 64   **** 1   NODE: 63 - - - - - -
64 -----> 65   **** 2   NODE: 64 k##6<10
64 -----> 70   **** 2   NODE: 64 k##6<10
65 -----> 66   **** 1   NODE: 65 - - - - - -
66 -----> 67   **** 1   NODE: 66 k##6==i##5
66 -----> 68   **** 1   NODE: 66 k##6==i##5
67 -----> 68   **** 1   NODE: 67 l##4=l##3+4
68 -----> 69   **** 2   NODE: 68 - - - - - -
69 -----> 64   **** 1   NODE: 69 - - - - - -
70 -----> 71   **** 1   NODE: 70 i##6=i##5+3
71 -----> 53   **** 1   NODE: 71 - - - - - -
59 -----> 60   **** 0   NODE: 59 l##5=6
61 -----> 62   **** 0   NODE: 61 k##7=k##6+2
72 -----> 73   **** 1   NODE: 72 h4##5=(i##5*k##6+h2##2+h3##1)/h1##1
```

```
73 -----> 76   **** 1   NODE: 73 - - - - - -
74 -----> 75   **** 1   NODE: 74 - - - - - -
75 -----> 76   **** 1   NODE: 75 h4##6=10000
18 -----> 19   **** 0   NODE: 18 h3##3=500
79 -----> 80   **** 1   NODE: 79 i##1,j##1,h4##1
```

The static single assignment form for the program test3.c

```
1 -----> 2   **** 0   NODE: 1 h1##1=10
2 -----> 3   **** 1   NODE: 2 h2##1=0
3 -----> 4   **** 1   NODE: 3 h3##1=0
4 -----> 5   **** 1   NODE: 4 h4##1=0
5 -----> 6   **** 1   NODE: 5 i##1,j##1,k##1,l##1
6 -----> 7   **** 1   NODE: 6 - - - - - -  **7
h2##2 = PHI(h2##1 1 2,h2##5 5 76)
h4##2 = PHI(h4##1 1 4,h4##4 4 76)
i##2 = PHI(i##1 1 5,i##4 4 77)
k##2 = PHI(k##1 1 5,k##6 6 76)
h1##2 = PHI(h1##1 1 1,h1##5 5 76)
h3##2 = PHI(h3##1 1 3,h3##4 4 76)
j##2 = PHI(j##1 1 5,j##3 3 76)
l##2 = PHI(l##1 1 5,l##3 3 76)
7 -----> 8   **** 2   NODE: 7 i##2<10
7 -----> 79   **** 2   NODE: 7 i##2<10
8 -----> 9   **** 1   NODE: 8 - - - - - -  **7
9 -----> 10   **** 1   NODE: 9 k##3=0  **11
10 -----> 11   **** 1   NODE: 10 - - - - - -  **11
h1##3 = PHI(h1##2 2 7,h1##4 4 13)
k##4 = PHI(k##3 3 9,k##5 5 14)
11 -----> 12   **** 2   NODE: 11 k##4<10
11 -----> 16   **** 2   NODE: 11 k##4<10
12 -----> 13   **** 1   NODE: 12 - - - - - -  **11
13 -----> 14   **** 1   NODE: 13 h1##4=h1##3+10
14 -----> 15   **** 1   NODE: 14 k##5=k##4+1  **12
15 -----> 11   **** 1   NODE: 15 - - - - - -  **12
16 -----> 17   **** 1   NODE: 16 h1##3>100
16 -----> 18   **** 1   NODE: 16 h1##3>100
17 -----> 19   **** 1   NODE: 17 h2##3=1000
h2##4 = PHI(h2##3 3 17,h2##2 2 7)
h3##3 = PHI(h3##2 2 7,h3##9 9 18)
19 -----> 20   **** 2   NODE: 19 - - - - - -  **16
20 -----> 21   **** 1   NODE: 20 j##2==0
20 -----> 23   **** 1   NODE: 20 j##2==0
21 -----> 22   **** 1   NODE: 21 h4##3=(h2##4+h3##3)/h1##3
22 -----> 76   **** 1   NODE: 22 - - - - - -  **20
h4##4 = PHI(h4##3 3 21,h4##5 5 24,h4##6 6 27,h4##7 7 72,h4##8 8 75)
i##3 = PHI(i##2 2 7,i##2 2 7,i##6 6 30,i##9 9 72,i##2 2 7)
j##3 = PHI(j##2 2 20,j##2 2 23,j##4 4 30,j##9 9 53,j##2 2 50)
k##6 = PHI(k##4 4 11,k##4 4 24,k##7 7 30,k##13 13 72,k##4 4 11)
l##3 = PHI(l##2 2 7,l##2 2 7,l##2 2 7,l##5 5 53,l##2 2 7)
h1##5 = PHI(h1##3 3 21,h1##3 3 24,h1##6 6 30,h1##3 3 72,h1##3 3 16)
h2##5 = PHI(h2##4 4 21,h2##4 4 24,h2##6 6 30,h2##4 4 72,h2##4 4 19)
h3##4 = PHI(h3##3 3 21,h3##3 3 19,h3##5 5 30,h3##3 3 72,h3##3 3 19)
76 -----> 77   **** 5   NODE: 76 - - - - - -  **20
77 -----> 78   **** 1   NODE: 77 i##4=i##3+5
78 -----> 7   **** 1   NODE: 78 - - - - - -  **7
23 -----> 24   **** 1   NODE: 23 j##2==1
23 -----> 26   **** 1   NODE: 23 j##2==1
24 -----> 25   **** 1   NODE: 24 h4##5=(k##4+h2##4)/h1##3
25 -----> 76   **** 1   NODE: 25 - - - - - -  **23
26 -----> 27   **** 1   NODE: 26 j##2==2
26 -----> 50   **** 1   NODE: 26 j##2==2
27 -----> 28   **** 1   NODE: 27 h4##6=(i##2+k##4)/h1##3
```

```
28 -----> 29   **** 1   NODE: 28 i##5=0  **30
29 -----> 30   **** 1   NODE: 29 - - - - - -  **30
i##6 = PHI(i##5 5 28,i##7 7 47)
j##4 = PHI(j##2 2 26,j##6 6 34)
h1##6 = PHI(h1##3 3 27,h1##7 7 34)
h2##6 = PHI(h2##4 4 19,h2##7 7 34)
h3##5 = PHI(h3##3 3 19,h3##6 6 34)
k##7 = PHI(k##4 4 27,k##8 8 34)
30 -----> 31   **** 2   NODE: 30 i##6<10
30 -----> 49   **** 2   NODE: 30 i##6<10
31 -----> 32   **** 1   NODE: 31 - - - - - -  **30
32 -----> 33   **** 1   NODE: 32 j##5=0  **34
33 -----> 34   **** 1   NODE: 33 - - - - - -  **34
j##6 = PHI(j##5 5 32,j##7 7 45)
k##8 = PHI(k##7 7 30,k##10 10 38)
h1##7 = PHI(h1##6 6 30,h1##8 8 38)
h2##7 = PHI(h2##6 6 30,h2##8 8 38)
h3##6 = PHI(h3##5 5 30,h3##7 7 38)
34 -----> 35   **** 2   NODE: 34 j##6<10
34 -----> 47   **** 2   NODE: 34 j##6<10
35 -----> 36   **** 1   NODE: 35 - - - - - -  **34
36 -----> 37   **** 1   NODE: 36 k##9=0  **38
37 -----> 38   **** 1   NODE: 37 - - - - - -  **38
h1##8 = PHI(h1##7 7 34,h1##9 9 40)
h2##8 = PHI(h2##7 7 34,h2##9 9 41)
h3##7 = PHI(h3##6 6 34,h3##8 8 42)
k##10 = PHI(k##9 9 36,k##11 11 43)
38 -----> 39   **** 2   NODE: 38 k##10<19
38 -----> 45   **** 2   NODE: 38 k##10<19
39 -----> 40   **** 1   NODE: 39 - - - - - -  **38
40 -----> 41   **** 1   NODE: 40 h1##9=i##6+j##6
41 -----> 42   **** 1   NODE: 41 h2##9=k##10+j##6
42 -----> 43   **** 1   NODE: 42 h3##8=k##10*j##6*i##6
43 -----> 44   **** 1   NODE: 43 k##11=k##10+1  **39
44 -----> 38   **** 1   NODE: 44 - - - - - -  **39
45 -----> 46   **** 1   NODE: 45 j##7=j##6+1  **35
46 -----> 34   **** 1   NODE: 46 - - - - - -  **35
47 -----> 48   **** 1   NODE: 47 i##7=i##6+1  **31
48 -----> 30   **** 1   NODE: 48 - - - - - -  **31
49 -----> 76   **** 1   NODE: 49 - - - - - -  **26
50 -----> 51   **** 1   NODE: 50 j##2==3
50 -----> 74   **** 1   NODE: 50 j##2==3
51 -----> 52   **** 1   NODE: 51 i##8,j##8,k##12,l##4
52 -----> 53   **** 1   NODE: 52 - - - - - -  **53
i##9 = PHI(i##8 8 51,i##10 10 70)
j##9 = PHI(j##8 8 51,j##11 11 62)
l##5 = PHI(l##4 4 51,l##9 9 64)
k##13 = PHI(k##12 12 51,k##14 14 64)
53 -----> 54   **** 2   NODE: 53 i##9<10
53 -----> 72   **** 2   NODE: 53 i##9<10
54 -----> 55   **** 1   NODE: 54 - - - - - -  **53
55 -----> 56   **** 1   NODE: 55 k##13==l##5
55 -----> 61   **** 1   NODE: 55 k##13==l##5
56 -----> 57   **** 1   NODE: 56 j##10=i##9
57 -----> 58   **** 1   NODE: 57 k##13==j##10
57 -----> 59   **** 1   NODE: 57 k##13==j##10
58 -----> 60   **** 1   NODE: 58 l##6=5
l##7 = PHI(l##6 6 58,l##12 12 59)
60 -----> 62   **** 2   NODE: 60 - - - - - -  **57
j##11 = PHI(j##10 10 57,j##9 9 53)
k##14 = PHI(k##13 13 57,k##15 15 61)
l##8 = PHI(l##7 7 60,l##5 5 55)
62 -----> 63   **** 2   NODE: 62 - - - - - -  **55
63 -----> 64   **** 1   NODE: 63 - - - - - -  **64
l##9 = PHI(l##8 8 62,l##11 11 68)
```

```
64 -----> 65   **** 2   NODE: 64 k##14<10
64 -----> 70   **** 2   NODE: 64 k##14<10
65 -----> 66   **** 1   NODE: 65 - - - - - -  **64
66 -----> 67   **** 1   NODE: 66 k##14==i##9
66 -----> 68   **** 1   NODE: 66 k##14==i##9
67 -----> 68   **** 1   NODE: 67 l##10=l##9+4
l##11 = PHI(l##9 9 64,l##10 10 67)
68 -----> 69   **** 2   NODE: 68 - - - - - -  **66
69 -----> 64   **** 1   NODE: 69 - - - - - -  **64
70 -----> 71   **** 1   NODE: 70 i##10=i##9+3
71 -----> 53   **** 1   NODE: 71 - - - - - -  **53
59 -----> 60   **** 0   NODE: 59 l##12=6
61 -----> 62   **** 0   NODE: 61 k##15=k##13+2
72 -----> 73   **** 1   NODE: 72 h4##7=(i##9*k##13+h2##4+h3##3)/h1##3
73 -----> 76   **** 1   NODE: 73 - - - - - -  **50
74 -----> 75   **** 1   NODE: 74 - - - - - -
75 -----> 76   **** 1   NODE: 75 h4##8=10000
18 -----> 19   **** 0   NODE: 18 h3##9=500
79 -----> 80   **** 1   NODE: 79 i##2,j##2,h4##2
```

The gated single assignment form for the program test3.c

```
1 -----> 2   **** 0   NODE: 1 h1##1=10
2 -----> 3   **** 1   NODE: 2 h2##1=0
3 -----> 4   **** 1   NODE: 3 h3##1=0
4 -----> 5   **** 1   NODE: 4 h4##1=0
5 -----> 6   **** 1   NODE: 5 i##1,j##1,k##1,l##1
L = .TRUE
6 -----> 7   **** 1   NODE: 6 - - - - - -  **7
h2##2 = MU.L(7,h2##1,h2##5)
h4##2 = MU.L(7,h4##1,h4##4)
i##2 = MU.L(7,i##1,i##4)
k##2 = MU.L(7,k##1,k##6)
h1##2 = MU.L(7,h1##1,h1##5)
h3##2 = MU.L(7,h3##1,h3##4)
j##2 = MU.L(7,j##1,j##3)
l##2 = MU.L(7,l##1,l##3)
7 -----> 8   **** 2   NODE: 7 i##2<10
7 -----> 79   **** 2   NODE: 7 i##2<10
L = .FALSE
8 -----> 9   **** 1   NODE: 8 - - - - - -  **7
9 -----> 10   **** 1   NODE: 9 k##3=0  **11
L = .TRUE
10 -----> 11   **** 1   NODE: 10 - - - - - -  **11
h1##3 = MU.L(11,h1##2,h1##4)
k##4 = MU.L(11,k##3,k##5)
11 -----> 12   **** 2   NODE: 11 k##4<10
11 -----> 16   **** 2   NODE: 11 k##4<10
L = .FALSE
12 -----> 13   **** 1   NODE: 12 - - - - - -  **11
13 -----> 14   **** 1   NODE: 13 h1##4=h1##3+10
14 -----> 15   **** 1   NODE: 14 k##5=k##4+1  **12
h1##3 = ETA.T(12,h1##4)
k##4 = ETA.T(12,k##5)
15 -----> 11   **** 1   NODE: 15 - - - - - -  **12
16 -----> 17   **** 1   NODE: 16 h1##3>100
16 -----> 18   **** 1   NODE: 16 h1##3>100
17 -----> 19   **** 1   NODE: 17 h2##3=1000
h2##4 = GAMMA(16,h2##3,h2##2)
h3##3 = GAMMA(16,h3##2,h3##9)
19 -----> 20   **** 2   NODE: 19 - - - - - -  **16
20 -----> 21   **** 1   NODE: 20 j##2==0
```

```
 20 -----> 23   **** 1   NODE: 20 j##2==0
 21 -----> 22   **** 1   NODE: 21 h4##3=(h2##4+h3##3)/h1##3
 22 -----> 76   **** 1   NODE: 22 - - - - - - -  **20
 h4##4 = GAMMA(20,h4##3, GAMMA(23,h4##5, GAMMA(26,h4##6,
GAMMA(50,h4##7,h4##8)))
 i##3 = GAMMA(20,i##2, GAMMA(23,i##2, GAMMA(26,i##6,
GAMMA(50,i##9,i##2)))
 j##3 = GAMMA(20,j##2, GAMMA(23,j##2, GAMMA(26,j##4,
GAMMA(50,j##9,j##2)))
 k##6 = GAMMA(20,k##4, GAMMA(23,k##4, GAMMA(26,k##7,
GAMMA(50,k##13,k##4)))
 l##3 = GAMMA(20,l##2, GAMMA(23,l##2, GAMMA(26,l##2,
GAMMA(50,l##5,l##2)))
 h1##5 = GAMMA(20,h1##3, GAMMA(23,h1##3, GAMMA(26,h1##6,
GAMMA(50,h1##3,h1##3)))
 h2##5 = GAMMA(20,h2##4, GAMMA(23,h2##4, GAMMA(26,h2##6,
GAMMA(50,h2##4,h2##4)))
 h3##4 = GAMMA(20,h3##3, GAMMA(23,h3##3, GAMMA(26,h3##5,
GAMMA(50,h3##3,h3##3)))
 76 -----> 77   **** 5   NODE: 76 - - - - - -  **20
 77 -----> 78   **** 1   NODE: 77 i##4=i##3+5
 h2##2 = ETA.T(7,h2##5)
 h4##2 = ETA.T(7,h4##4)
 i##2 = ETA.T(7,i##4)
 k##2 = ETA.T(7,k##6)
 h1##2 = ETA.T(7,h1##5)
 h3##2 = ETA.T(7,h3##4)
 j##2 = ETA.T(7,j##3)
 l##2 = ETA.T(7,l##3)
 78 -----> 7    **** 1   NODE: 78 - - - - - -  **7
 23 -----> 24   **** 1   NODE: 23 j##2==1
 23 -----> 26   **** 1   NODE: 23 j##2==1
 24 -----> 25   **** 1   NODE: 24 h4##5=(k##4+h2##4)/h1##3
 25 -----> 76   **** 1   NODE: 25 - - - - - -  **23
 26 -----> 27   **** 1   NODE: 26 j##2==2
 26 -----> 50   **** 1   NODE: 26 j##2==2
 27 -----> 28   **** 1   NODE: 27 h4##6=(i##2+k##4)/h1##3
 28 -----> 29   **** 1   NODE: 28 i##5=0  **30
 L = .TRUE
 29 -----> 30   **** 1   NODE: 29 - - - - - -  **30
 i##6 = MU.L(30,i##5,i##7)
 j##4 = MU.L(30,j##2,j##6)
 h1##6 = MU.L(30,h1##3,h1##7)
 h2##6 = MU.L(30,h2##4,h2##7)
 h3##5 = MU.L(30,h3##3,h3##6)
 k##7 = MU.L(30,k##4,k##8)
 30 -----> 31   **** 2   NODE: 30 i##6<10
 30 -----> 49   **** 2   NODE: 30 i##6<10
 L = .FALSE
 31 -----> 32   **** 1   NODE: 31 - - - - - -  **30
 32 -----> 33   **** 1   NODE: 32 j##5=0  **34
 L = .TRUE
 33 -----> 34   **** 1   NODE: 33 - - - - - -  **34
 j##6 = MU.L(34,j##5,j##7)
 k##8 = MU.L(34,k##7,k##10)
 h1##7 = MU.L(34,h1##6,h1##8)
 h2##7 = MU.L(34,h2##6,h2##8)
 h3##6 = MU.L(34,h3##5,h3##7)
 34 -----> 35   **** 2   NODE: 34 j##6<10
 34 -----> 47   **** 2   NODE: 34 j##6<10
 L = .FALSE
 35 -----> 36   **** 1   NODE: 35 - - - - - -  **34
 36 -----> 37   **** 1   NODE: 36 k##9=0  **38
 L = .TRUE
 37 -----> 38   **** 1   NODE: 37 - - - - - -  **38
```

```
h1##8 = MU.L(38,h1##7,h1##9)
h2##8 = MU.L(38,h2##7,h2##9)
h3##7 = MU.L(38,h3##6,h3##8)
k##10 = MU.L(38,k##9,k##11)
38 -----> 39   **** 2   NODE: 38 k##10<19
38 -----> 45   **** 2   NODE: 38 k##10<19
L = .FALSE
39 -----> 40   **** 1   NODE: 39 - - - - - -  **38
40 -----> 41   **** 1   NODE: 40 h1##9=i##6+j##6
41 -----> 42   **** 1   NODE: 41 h2##9=k##10+j##6
42 -----> 43   **** 1   NODE: 42 h3##8=k##10*j##6*i##6
43 -----> 44   **** 1   NODE: 43 k##11=k##10+1  **39
h1##8 = ETA.T(39,h1##9)
h2##8 = ETA.T(39,h2##9)
h3##7 = ETA.T(39,h3##8)
k##10 = ETA.T(39,k##11)
44 -----> 38   **** 1   NODE: 44 - - - - - -  **39
45 -----> 46   **** 1   NODE: 45 j##7=j##6+1  **35
j##6 = ETA.T(35,j##7)
k##8 = ETA.T(35,k##10)
h1##7 = ETA.T(35,h1##8)
h2##7 = ETA.T(35,h2##8)
h3##6 = ETA.T(35,h3##7)
46 -----> 34   **** 1   NODE: 46 - - - - - -  **35
47 -----> 48   **** 1   NODE: 47 i##7=i##6+1  **31
i##6 = ETA.T(31,i##7)
j##4 = ETA.T(31,j##6)
h1##6 = ETA.T(31,h1##7)
h2##6 = ETA.T(31,h2##7)
h3##5 = ETA.T(31,h3##6)
k##7 = ETA.T(31,k##8)
48 -----> 30   **** 1   NODE: 48 - - - - - -  **31
49 -----> 76   **** 1   NODE: 49 - - - - - -  **26
50 -----> 51   **** 1   NODE: 50 j##2==3
50 -----> 74   **** 1   NODE: 50 j##2==3
51 -----> 52   **** 1   NODE: 51 i##8,j##8,k##12,l##4
L = .TRUE
52 -----> 53   **** 1   NODE: 52 - - - - - -  **53
i##9 = MU.L(53,i##8,i##10)
j##9 = MU.L(53,j##8,j##11)
l##5 = MU.L(53,l##4,l##9)
k##13 = MU.L(53,k##12,k##14)
53 -----> 54   **** 2   NODE: 53 i##9<10
53 -----> 72   **** 2   NODE: 53 i##9<10
L = .FALSE
54 -----> 55   **** 1   NODE: 54 - - - - - -  **53
55 -----> 56   **** 1   NODE: 55 k##13==l##5
55 -----> 61   **** 1   NODE: 55 k##13==l##5
56 -----> 57   **** 1   NODE: 56 j##10=i##9
57 -----> 58   **** 1   NODE: 57 k##13==j##10
57 -----> 59   **** 1   NODE: 57 k##13==j##10
58 -----> 60   **** 1   NODE: 58 l##6=5
l##7 = GAMMA(57,l##6,l##12)
60 -----> 62   **** 2   NODE: 60 - - - - - -  **57
j##11 = GAMMA(55,j##10,j##9)
k##14 = GAMMA(55,k##13,k##15)
l##8 = GAMMA(55,l##7,l##5)
62 -----> 63   **** 2   NODE: 62 - - - - - -  **55
L = .TRUE
63 -----> 64   **** 1   NODE: 63 - - - - - -  **64
l##9 = MU.L(64,l##8,l##11)
64 -----> 65   **** 2   NODE: 64 k##14<10
64 -----> 70   **** 2   NODE: 64 k##14<10
L = .FALSE
65 -----> 66   **** 1   NODE: 65 - - - - - -  **64
```

```
66 -----> 67   **** 1   NODE: 66 k##14==i##9
66 -----> 68   **** 1   NODE: 66 k##14==i##9
67 -----> 68   **** 1   NODE: 67 l##10=l##9+4
l##11 = GAMMA(66,l##9,l##10)
68 -----> 69   **** 2   NODE: 68 - - - - - -  **66
l##9 = ETA.T(64,l##11)
69 -----> 64   **** 1   NODE: 69 - - - - - -  **64
70 -----> 71   **** 1   NODE: 70 i##10=i##9+3
i##9 = ETA.T(53,i##10)
j##9 = ETA.T(53,j##11)
l##5 = ETA.T(53,l##9)
k##13 = ETA.T(53,k##14)
71 -----> 53   **** 1   NODE: 71 - - - - - -  **53
59 -----> 60   **** 0   NODE: 59 l##12=6
61 -----> 62   **** 0   NODE: 61 k##15=k##13+2
72 -----> 73   **** 1   NODE: 72 h4##7=(i##9*k##13+h2##4+h3##3)/h1##3
73 -----> 76   **** 1   NODE: 73 - - - - - -  **50
74 -----> 75   **** 1   NODE: 74 - - - - - -
75 -----> 76   **** 1   NODE: 75 h4##8=10000
18 -----> 19   **** 0   NODE: 18 h3##9=500
79 -----> 80   **** 1   NODE: 79 i##2,j##2,h4##2
```

The program dependence web for the program test3.c

```
1 -----> 2   **** 0   NODE: 1 h1##1=10
2 -----> 3   **** 1   NODE: 2 h2##1=0
3 -----> 4   **** 1   NODE: 3 h3##1=0
4 -----> 5   **** 1   NODE: 4 h4##1=0
5 -----> 6   **** 1   NODE: 5 i##1,j##1,k##1,l##1
L = .TRUE
6 -----> 7   **** 1   NODE: 6 - - - - - -  **7
h2##2 = MU.L(7,h2##1,h2##5)
h4##2 = MU.L(7,h4##1,h4##4)
i##2 = MU.L(7,i##1,i##4)
k##2 = MU.L(7,k##1,k##6)
h1##2 = MU.L(7,h1##1,h1##5)
h3##2 = MU.L(7,h3##1,h3##4)
j##2 = MU.L(7,j##1,j##3)
l##2 = MU.L(7,l##1,l##3)
7 -----> 8   **** 2   NODE: 7 i##2<10
7 -----> 79   **** 2   NODE: 7 i##2<10
L = .FALSE
8 -----> 9   **** 1   NODE: 8 - - - - - -  **7
9 -----> 10   **** 1   NODE: 9 k##3=0  **11
L = .TRUE
10 -----> 11   **** 1   NODE: 10 - - - - - -  **11
h1##3 = MU.L(11,h1##2,h1##4)
k##4 = MU.L(11,k##3,k##5)
11 -----> 12   **** 2   NODE: 11 k##4<10
11 -----> 16   **** 2   NODE: 11 k##4<10
L = .FALSE
12 -----> 13   **** 1   NODE: 12 - - - - - -  **11
13 -----> 14   **** 1   NODE: 13 h1##4=h1##3+10
14 -----> 15   **** 1   NODE: 14 k##5=k##4+1  **12
h1##3 = ETA.T(12,h1##4)
k##4 = ETA.T(12,k##5)
15 -----> 11   **** 1   NODE: 15 - - - - - -  **12
16 -----> 17   **** 1   NODE: 16 h1##3>100
16 -----> 18   **** 1   NODE: 16 h1##3>100
Predicate h1##3>100 is TRUE
17 -----> 19   **** 1   NODE: 17 h2##3=1000
h2##4 = GAMMA(16,h2##3,h2##2)
```

```
 h3##3 = GAMMA(16,h3##2,h3##9)
 19 -----> 20   **** 2   NODE: 19 - - - - - -  **16
 20 -----> 21   **** 1   NODE: 20 j##2==0
 20 -----> 23   **** 1   NODE: 20 j##2==0
 21 -----> 22   **** 1   NODE: 21 h4##3=(h2##4+h3##3)/h1##3
 22 -----> 76   **** 1   NODE: 22 - - - - - -  **20
 h4##4 = GAMMA(20,h4##3, GAMMA(23,h4##5, GAMMA(26,h4##6,
GAMMA(50,h4##7,h4##8))))
 i##3 = GAMMA(20,i##2, GAMMA(23,i##2, GAMMA(26,i##6,
GAMMA(50,i##9,i##2))))
 j##3 = GAMMA(20,j##2, GAMMA(23,j##2, GAMMA(26,j##4,
GAMMA(50,j##9,j##2))))
 k##6 = GAMMA(20,k##4, GAMMA(23,k##4, GAMMA(26,k##7,
GAMMA(50,k##13,k##4))))
 l##3 = GAMMA(20,l##2, GAMMA(23,l##2, GAMMA(26,l##2,
GAMMA(50,l##5,l##2))))
 h1##5 = GAMMA(20,h1##3, GAMMA(23,h1##3, GAMMA(26,h1##6,
GAMMA(50,h1##3,h1##3))))
 h2##5 = GAMMA(20,h2##4, GAMMA(23,h2##4, GAMMA(26,h2##6,
GAMMA(50,h2##4,h2##4))))
 h3##4 = GAMMA(20,h3##3, GAMMA(23,h3##3, GAMMA(26,h3##5,
GAMMA(50,h3##3,h3##3))))
 76 -----> 77   **** 5   NODE: 76 - - - - - -  **20
 77 -----> 78   **** 1   NODE: 77 i##4=i##3+5
 h2##2 = ETA.T(7,h2##5)
 h4##2 = ETA.T(7,h4##4)
 i##2 = ETA.T(7,i##4)
 k##2 = ETA.T(7,k##6)
 h1##2 = ETA.T(7,h1##5)
 h3##2 = ETA.T(7,h3##4)
 j##2 = ETA.T(7,j##3)
 l##2 = ETA.T(7,l##3)
 78 -----> 7   **** 1   NODE: 78 - - - - - -  **7
 23 -----> 24   **** 1   NODE: 23 j##2==1
 23 -----> 26   **** 1   NODE: 23 j##2==1
 24 -----> 25   **** 1   NODE: 24 h4##5=(k##4+h2##4)/h1##3
 25 -----> 76   **** 1   NODE: 25 - - - - - -  **23
 26 -----> 27   **** 1   NODE: 26 j##2==2
 26 -----> 50   **** 1   NODE: 26 j##2==2
 27 -----> 28   **** 1   NODE: 27 h4##6=(i##2+k##4)/h1##3
 28 -----> 29   **** 1   NODE: 28 i##5=0  **30
 L = .TRUE
 29 -----> 30   **** 1   NODE: 29 - - - - - -  **30
 i##6 = MU.L(30,i##5,i##7)
 j##4 = MU.L(30,j##2,j##6)
 h1##6 = MU.L(30,h1##3,h1##7)
 h2##6 = MU.L(30,h2##4,h2##7)
 h3##5 = MU.L(30,h3##3,h3##6)
 k##7 = MU.L(30,k##4,k##8)
 30 -----> 31   **** 2   NODE: 30 i##6<10
 30 -----> 49   **** 2   NODE: 30 i##6<10
 L = .FALSE
 31 -----> 32   **** 1   NODE: 31 - - - - - -  **30
 32 -----> 33   **** 1   NODE: 32 j##5=0  **34
 L = .TRUE
 33 -----> 34   **** 1   NODE: 33 - - - - - -  **34
 j##6 = MU.L(34,j##5,j##7)
 k##8 = MU.L(34,k##7,k##10)
 h1##7 = MU.L(34,h1##6,h1##8)
 h2##7 = MU.L(34,h2##6,h2##8)
 h3##6 = MU.L(34,h3##5,h3##7)
 34 -----> 35   **** 2   NODE: 34 j##6<10
 34 -----> 47   **** 2   NODE: 34 j##6<10
 L = .FALSE
 35 -----> 36   **** 1   NODE: 35 - - - - - -  **34
```

```
36 -----> 37   **** 1   NODE: 36 k##9=0  **38
L = .TRUE
37 -----> 38   **** 1   NODE: 37 - - - - - -  **38
h1##8 = MU.L(38,h1##7,h1##9)
h2##8 = MU.L(38,h2##7,h2##9)
h3##7 = MU.L(38,h3##6,h3##8)
k##10 = MU.L(38,k##9,k##11)
38 -----> 39   **** 2   NODE: 38 k##10<19
38 -----> 45   **** 2   NODE: 38 k##10<19
L = .FALSE
39 -----> 40   **** 1   NODE: 39 - - - - - -  **38
40 -----> 41   **** 1   NODE: 40 h1##9=i##6+j##6
41 -----> 42   **** 1   NODE: 41 h2##9=k##10+j##6
42 -----> 43   **** 1   NODE: 42 h3##8=k##10*j##6*i##6
43 -----> 44   **** 1   NODE: 43 k##11=k##10+1  **39
h1##8 = ETA.T(39,h1##9)
h2##8 = ETA.T(39,h2##9)
h3##7 = ETA.T(39,h3##8)
k##10 = ETA.T(39,k##11)
44 -----> 38   **** 1   NODE: 44 - - - - - -  **39
45 -----> 46   **** 1   NODE: 45 j##7=j##6+1  **35
j##6 = ETA.T(35,j##7)
k##8 = ETA.T(35,k##10)
h1##7 = ETA.T(35,h1##8)
h2##7 = ETA.T(35,h2##8)
h3##6 = ETA.T(35,h3##7)
46 -----> 34   **** 1   NODE: 46 - - - - - -  **35
47 -----> 48   **** 1   NODE: 47 i##7=i##6+1  **31
i##6 = ETA.T(31,i##7)
j##4 = ETA.T(31,j##6)
h1##6 = ETA.T(31,h1##7)
h2##6 = ETA.T(31,h2##7)
h3##5 = ETA.T(31,h3##6)
k##7 = ETA.T(31,k##8)
48 -----> 30   **** 1   NODE: 48 - - - - - -  **31
49 -----> 76   **** 1   NODE: 49 - - - - - -  **26
50 -----> 51   **** 1   NODE: 50 j##2==3
50 -----> 74   **** 1   NODE: 50 j##2==3
51 -----> 52   **** 1   NODE: 51 i##8,j##8,k##12,l##4
L = .TRUE
52 -----> 53   **** 1   NODE: 52 - - - - - -  **53
i##9 = MU.L(53,i##8,i##10)
j##9 = MU.L(53,j##8,j##11)
l##5 = MU.L(53,l##4,l##9)
k##13 = MU.L(53,k##12,k##14)
53 -----> 54   **** 2   NODE: 53 i##9<10
53 -----> 72   **** 2   NODE: 53 i##9<10
L = .FALSE
54 -----> 55   **** 1   NODE: 54 - - - - - -  **53
55 -----> 56   **** 1   NODE: 55 k##13==l##5
55 -----> 61   **** 1   NODE: 55 k##13==l##5
Predicate k##13==l##5 is TRUE
56 -----> 57   **** 1   NODE: 56 j##10=i##9
57 -----> 58   **** 1   NODE: 57 k##13==j##10
57 -----> 59   **** 1   NODE: 57 k##13==j##10
Predicate k##13==j##10 is TRUE
58 -----> 60   **** 1   NODE: 58 l##6=5
l##7 = GAMMA(57,l##6,l##12)
60 -----> 62   **** 2   NODE: 60 - - - - - -  **57
j##11 = GAMMA(55,j##10,j##9)
k##14 = GAMMA(55,k##13,k##15)
l##8 = GAMMA(55,l##7,l##5)
62 -----> 63   **** 2   NODE: 62 - - - - - -  **55
L = .TRUE
63 -----> 64   **** 1   NODE: 63 - - - - - -  **64
```

```
l##9 = MU.L(64,l##8,l##11)
64 -----> 65   **** 2   NODE: 64 k##14<10
64 -----> 70   **** 2   NODE: 64 k##14<10
L = .FALSE
65 -----> 66   **** 1   NODE: 65 - - - - - -  **64
66 -----> 67   **** 1   NODE: 66 k##14==i##9
66 -----> 68   **** 1   NODE: 66 k##14==i##9
Predicate k##14==i##9 is TRUE
67 -----> 68   **** 1   NODE: 67 l##10=l##9+4
l##11 = GAMMA(66,l##9,l##10)
68 -----> 69   **** 2   NODE: 68 - - - - - -  **66
l##9 = ETA.T(64,l##11)
69 -----> 64   **** 1   NODE: 69 - - - - - -  **64
Predicate k##14==i##9 is FALSE
70 -----> 71   **** 1   NODE: 70 i##10=i##9+3
i##9 = ETA.T(53,i##10)
j##9 = ETA.T(53,j##11)
l##5 = ETA.T(53,l##9)
k##13 = ETA.T(53,k##14)
71 -----> 53   **** 1   NODE: 71 - - - - - -  **53
Predicate k##13==j##10 is FALSE
59 -----> 60   **** 0   NODE: 59 l##12=6
Predicate k##13==l##5 is FALSE
61 -----> 62   **** 0   NODE: 61 k##15=k##13+2
72 -----> 73   **** 1   NODE: 72 h4##7=(i##9*k##13+h2##4+h3##3)/h1##3
73 -----> 76   **** 1   NODE: 73 - - - - - -  **50
74 -----> 75   **** 1   NODE: 74 - - - - - -
75 -----> 76   **** 1   NODE: 75 h4##8=10000
Predicate h1##3>100 is FALSE
18 -----> 19   **** 0   NODE: 18 h3##9=500
79 -----> 80   **** 1   NODE: 79 i##2,j##2,h4##2
```

APPENDIX D

GRAPHS OF THE OUTPUT GENERATED BY THE TOOL

The following program is an input to the tool, which translates simple C programs to PDWs. For the following program, the various graphical representations of the output are discussed. This should facilitate the interpretation of the output generated by the tool for an input C program given by the grammar discussed in Section 4.1.

```
#include <stdio.h>
main()
{
int i,j,k,x,m;
int h1,h2,h3,h4,h5;

scanf("%d %d %d %d",&i,&j,&k,&x);
i = 0;

while(i < 10) {
      if (i < 7) {
            if (i < 5) x = x + 5;
            else
                  x = x +6;
            }
      else x = x + 2;
      m = k;
      while(k < j) {
            if (x < 1000) x = x + 1;
            k = k + 1;
            }
      k = m;
      i = i + 1;
      }
printf("%d", x);
}
```

Figure 14. A sample C program used as an input to the tool

The above program adds 5 to the input variable x five times, 6 twice and 2 thrice. Each time if the value of x is less than 1000, the program adds 1 to x the difference of k and j times.

The following is the output to the C program listed in Figure 14 which was given as input to the tool. Each statement of the program listed in Figure 14 is assigned a statement number as it can be seen below.

```
1 Input i##2,j##2,k##2,x##2
2  i##3=0
4 i##3<10
6 if i##3<7
7 if i##3<5
8  x##3=x##2+5
9  x##4=x##3+6
11  x##5=x##4+2
13  m##2=k##2
15 k##2<j##2
17 if x##5<1000
18  x##6=x##5+1
20  k##3=k##2+1
22  k##4=m##2
23  i##4=i##3+1
25 Output x##6
26  END
```

The following is the control flow graph as a linked list for the program listed in Figure 14. The pictorial representation of the CFG is shown in Figure 15.

```
-1 ------ 1
1 ------ 2
2 ------ 3
3 ------ 4
4 ------ 5
5 ------ 6
6 ------ 7
7 ------ 8
8 ------ 10
9 ------ 10
7 ------ 9
10 ------ 12
11 ------ 12
6 ------ 11
12 ------ 13
13 ------ 14
14 ------ 15
15 ------ 16
16 ------ 17
17 ------ 18
18 ------ 19
17 ------ 19
19 ------ 20
20 ------ 21
21 ------ 15
15 ------ 22
22 ------ 23
23 ------ 24
24 ------ 4
4 ------ 25
25 ------ 26
```

The following is the control flow graph as a tree-structured directed graph for the program given in Figure 14. The pictorial representation of the CFG is shown in Figure 16.

```
1 -----> 2   **** 0   NODE: 1 i##1,j##1,k##1,x##1
2 -----> 3   **** 1   NODE: 2 i##2=0
3 -----> 4   **** 1   NODE: 3 - - - - - -
4 -----> 5   **** 2   NODE: 4 i##2<10
4 -----> 25   **** 2   NODE: 4 i##2<10
5 -----> 6   **** 1   NODE: 5 - - - - - -
6 -----> 7   **** 1   NODE: 6 i##2<7
6 -----> 11   **** 1   NODE: 6 i##2<7
7 -----> 8   **** 1   NODE: 7 i##2<5
7 -----> 9   **** 1   NODE: 7 i##2<5
8 -----> 10   **** 1   NODE: 8 x##2=x##1+5
10 -----> 12   **** 2   NODE: 10 - - - - - -
12 -----> 13   **** 2   NODE: 12 - - - - - -
13 -----> 14   **** 1   NODE: 13 m##1=k##1
14 -----> 15   **** 1   NODE: 14 - - - - - -
15 -----> 16   **** 2   NODE: 15 k##1<j##1
15 -----> 22   **** 2   NODE: 15 k##1<j##1
16 -----> 17   **** 1   NODE: 16 - - - - - -
17 -----> 18   **** 1   NODE: 17 x##2<1000
17 -----> 19   **** 1   NODE: 17 x##2<1000
18 -----> 19   **** 1   NODE: 18 x##3=x##2+1
19 -----> 20   **** 2   NODE: 19 - - - - - -
20 -----> 21   **** 1   NODE: 20 k##2=k##1+1
21 -----> 15   **** 1   NODE: 21 - - - - - -
22 -----> 23   **** 1   NODE: 22 k##3=m##1
23 -----> 24   **** 1   NODE: 23 i##3=i##2+1
24 -----> 4   **** 1   NODE: 24 - - - - - -
9 -----> 10   **** 0   NODE: 9 x##4=x##1+6
11 -----> 12   **** 0   NODE: 11 x##5=x##1+2
25 -----> 26   **** 1   NODE: 25 x##1
```

The following is the static single assignment form for the program given in Figure 14. The pictorial representation for this program is given in Figure 17 and 17.a.

```
1 -----> 2   **** 0   NODE: 1 i##1,j##1,k##1,x##1
2 -----> 3   **** 1   NODE: 2 i##2=0
3 -----> 4   **** 1   NODE: 3 - - - - - -  **4
i##3 = PHI(i##2 2 2,i##4 4 23)
k##2 = PHI(k##1 1 1,k##5 5 22)
x##2 = PHI(x##1 1 1,x##6 6 15)
m##1 = PHI(m##0 0 0,m##2 2 22)
4 -----> 5   **** 2   NODE: 4 i##3<10
4 -----> 25   **** 2   NODE: 4 i##3<10
5 -----> 6   **** 1   NODE: 5 - - - - - -  **4
6 -----> 7   **** 1   NODE: 6 i##3<7
6 -----> 11   **** 1   NODE: 6 i##3<7
7 -----> 8   **** 1   NODE: 7 i##3<5
7 -----> 9   **** 1   NODE: 7 i##3<5
8 -----> 10   **** 1   NODE: 8 x##3=x##2+5
x##4 = PHI(x##3 3 8,x##9 9 9)
10 -----> 12   **** 2   NODE: 10 - - - - - -  **7
x##5 = PHI(x##4 4 10,x##10 10 11)
12 -----> 13   **** 2   NODE: 12 - - - - - -  **6
13 -----> 14   **** 1   NODE: 13 m##2=k##2
14 -----> 15   **** 1   NODE: 14 - - - - - -  **15
k##3 = PHI(k##2 2 13,k##4 4 20)
```

```
x##6 = PHI(x##5 5 12,x##8 8 19)
15 -----> 16   **** 2   NODE: 15 k##3<j##1
15 -----> 22   **** 2   NODE: 15 k##3<j##1
16 -----> 17   **** 1   NODE: 16 - - - - - - -  **15
17 -----> 18   **** 1   NODE: 17 x##6<1000
17 -----> 19   **** 1   NODE: 17 x##6<1000
18 -----> 19   **** 1   NODE: 18 x##7=x##6+1
x##8 = PHI(x##6 6 17,x##7 7 18)
19 -----> 20   **** 2   NODE: 19 - - - - - - -  **17
20 -----> 21   **** 1   NODE: 20 k##4=k##3+1
21 -----> 15   **** 1   NODE: 21 - - - - - - -  **15
22 -----> 23   **** 1   NODE: 22 k##5=m##2
23 -----> 24   **** 1   NODE: 23 i##4=i##3+1
24 -----> 4    **** 1   NODE: 24 - - - - - - -  **4
9 -----> 10    **** 0   NODE: 9 x##9=x##2+6
11 -----> 12   **** 0   NODE: 11 x##10=x##2+2
25 -----> 26   **** 1   NODE: 25 x##2
```

The following is the gated single assignment form for the program given in Figure 14. The pictorial representation of the if-then-else in the program is shown in Figure 18.

```
1 -----> 2    **** 0   NODE: 1 i##1,j##1,k##1,x##1
2 -----> 3    **** 1   NODE: 2 i##2=0
L = .TRUE
3 -----> 4    **** 1   NODE: 3 - - - - - - -  **4
i##3 = MU.L(4,i##2,i##4)
k##2 = MU.L(4,k##1,k##5)
x##2 = MU.L(4,x##1,x##6)
m##1 = MU.L(4,m##0,m##2)
4 -----> 5    **** 2   NODE: 4 i##3<10
4 -----> 25   **** 2   NODE: 4 i##3<10
L = .FALSE
5 -----> 6    **** 1   NODE: 5 - - - - - - -  **4
6 -----> 7    **** 1   NODE: 6 i##3<7
6 -----> 11   **** 1   NODE: 6 i##3<7
7 -----> 8    **** 1   NODE: 7 i##3<5
7 -----> 9    **** 1   NODE: 7 i##3<5
8 -----> 10   **** 1   NODE: 8 x##3=x##2+5
x##4 = GAMMA(7,x##3,x##9)
10 -----> 12  **** 2   NODE: 10 - - - - - - -  **7
x##5 = GAMMA(6,x##4,x##10)
12 -----> 13  **** 2   NODE: 12 - - - - - - -  **6
13 -----> 14  **** 1   NODE: 13 m##2=k##2
L = .TRUE
14 -----> 15  **** 1   NODE: 14 - - - - - - -  **15
k##3 = MU.L(15,k##2,k##4)
x##6 = MU.L(15,x##5,x##8)
15 -----> 16  **** 2   NODE: 15 k##3<j##1
15 -----> 22  **** 2   NODE: 15 k##3<j##1
L = .FALSE
16 -----> 17  **** 1   NODE: 16 - - - - - - -  **15
17 -----> 18  **** 1   NODE: 17 x##6<1000
17 -----> 19  **** 1   NODE: 17 x##6<1000
18 -----> 19  **** 1   NODE: 18 x##7=x##6+1
x##8 = GAMMA(17,x##6,x##7)
19 -----> 20  **** 2   NODE: 19 - - - - - - -  **17
20 -----> 21  **** 1   NODE: 20 k##4=k##3+1
k##3 = ETA.T(15,k##4)
x##6 = ETA.T(15,x##8)
21 -----> 15  **** 1   NODE: 21 - - - - - - -  **15
22 -----> 23  **** 1   NODE: 22 k##5=m##2
```

```
23 -----> 24   **** 1   NODE: 23 i##4=i##3+1
i##3 = ETA.T(4,i##4)
k##2 = ETA.T(4,k##5)
x##2 = ETA.T(4,x##6)
m##1 = ETA.T(4,m##2)
24 -----> 4    **** 1   NODE: 24 - - - - - -  **4
9 -----> 10    **** 0   NODE: 9 x##9=x##2+6
11 -----> 12    **** 0   NODE: 11 x##10=x##2+2
25 -----> 26    **** 1   NODE: 25 x##2
```

The following is the program dependence web for the program given in Figure 14. The pictorial representation of the if-then-else statement is given in Figure 19.

```
1 -----> 2    **** 0   NODE: 1 i##1,j##1,k##1,x##1
2 -----> 3    **** 1   NODE: 2 i##2=0
L = .TRUE
3 -----> 4    **** 1   NODE: 3 - - - - - -  **4
i##3 = MU.L(4,i##2,i##4)
k##2 = MU.L(4,k##1,k##5)
x##2 = MU.L(4,x##1,x##6)
m##1 = MU.L(4,m##0,m##2)
4 -----> 5    **** 2   NODE: 4 i##3<10
4 -----> 25    **** 2   NODE: 4 i##3<10
L = .FALSE
5 -----> 6    **** 1   NODE: 5 - - - - - -  **4
6 -----> 7    **** 1   NODE: 6 i##3<7
6 -----> 11    **** 1   NODE: 6 i##3<7
Predicate i##3<7 is TRUE
7 -----> 8    **** 1   NODE: 7 i##3<5
7 -----> 9    **** 1   NODE: 7 i##3<5
Predicate i##3<5 is TRUE
8 -----> 10    **** 1   NODE: 8 x##3=x##2+5
x##4 = GAMMA(7,x##3,x##9)
10 -----> 12    **** 2   NODE: 10 - - - - - -  **7
x##5 = GAMMA(6,x##4,x##10)
12 -----> 13    **** 2   NODE: 12 - - - - - -  **6
13 -----> 14    **** 1   NODE: 13 m##2=k##2
L = .TRUE
14 -----> 15    **** 1   NODE: 14 - - - - - -  **15
k##3 = MU.L(15,k##2,k##4)
x##6 = MU.L(15,x##5,x##8)
15 -----> 16    **** 2   NODE: 15 k##3<j##1
15 -----> 22    **** 2   NODE: 15 k##3<j##1
L = .FALSE
16 -----> 17    **** 1   NODE: 16 - - - - - -  **15
17 -----> 18    **** 1   NODE: 17 x##6<1000
17 -----> 19    **** 1   NODE: 17 x##6<1000
Predicate x##6<1000 is TRUE
18 -----> 19    **** 1   NODE: 18 x##7=x##6+1
x##8 = GAMMA(17,x##6,x##7)
19 -----> 20    **** 2   NODE: 19 - - - - - -  **17
20 -----> 21    **** 1   NODE: 20 k##4=k##3+1
k##3 = ETA.T(15,k##4)
x##6 = ETA.T(15,x##8)
21 -----> 15    **** 1   NODE: 21 - - - - - -  **15
Predicate x##6<1000 is FALSE
22 -----> 23    **** 1   NODE: 22 k##5=m##2
23 -----> 24    **** 1   NODE: 23 i##4=i##3+1
i##3 = ETA.T(4,i##4)
k##2 = ETA.T(4,k##5)
x##2 = ETA.T(4,x##6)
m##1 = ETA.T(4,m##2)
```

```
24 -----> 4   **** 1   NODE: 24 - - - - - -  **4
Predicate i##3<5 is FALSE
9 -----> 10   **** 0   NODE: 9 x##9=x##2+6
Predicate i##3<7 is FALSE
11 -----> 12   **** 0   NODE: 11 x##10=x##2+2
25 -----> 26   **** 1   NODE: 25 x##2
```

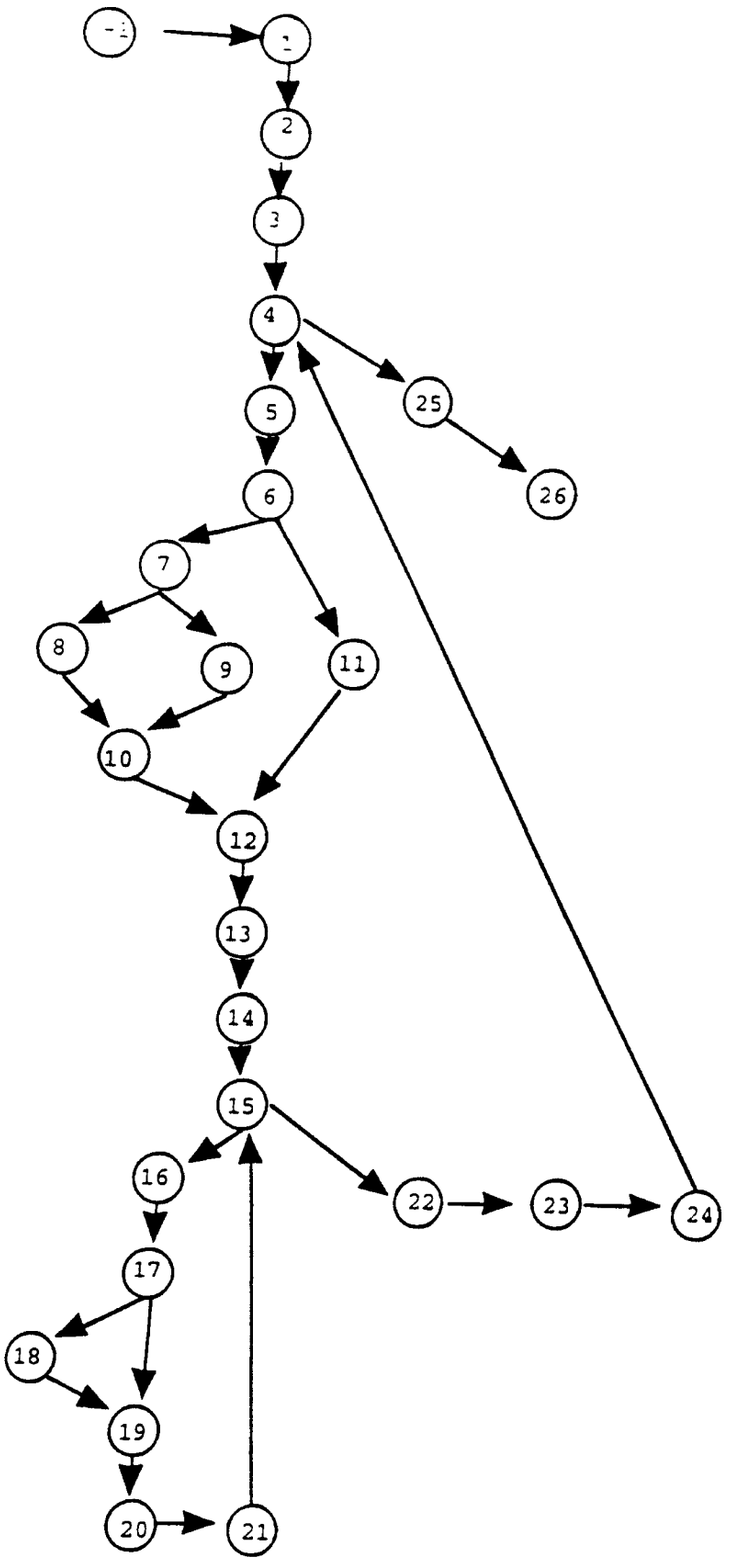

Figure 15. CFG as a linked list for the program given in Figure 14

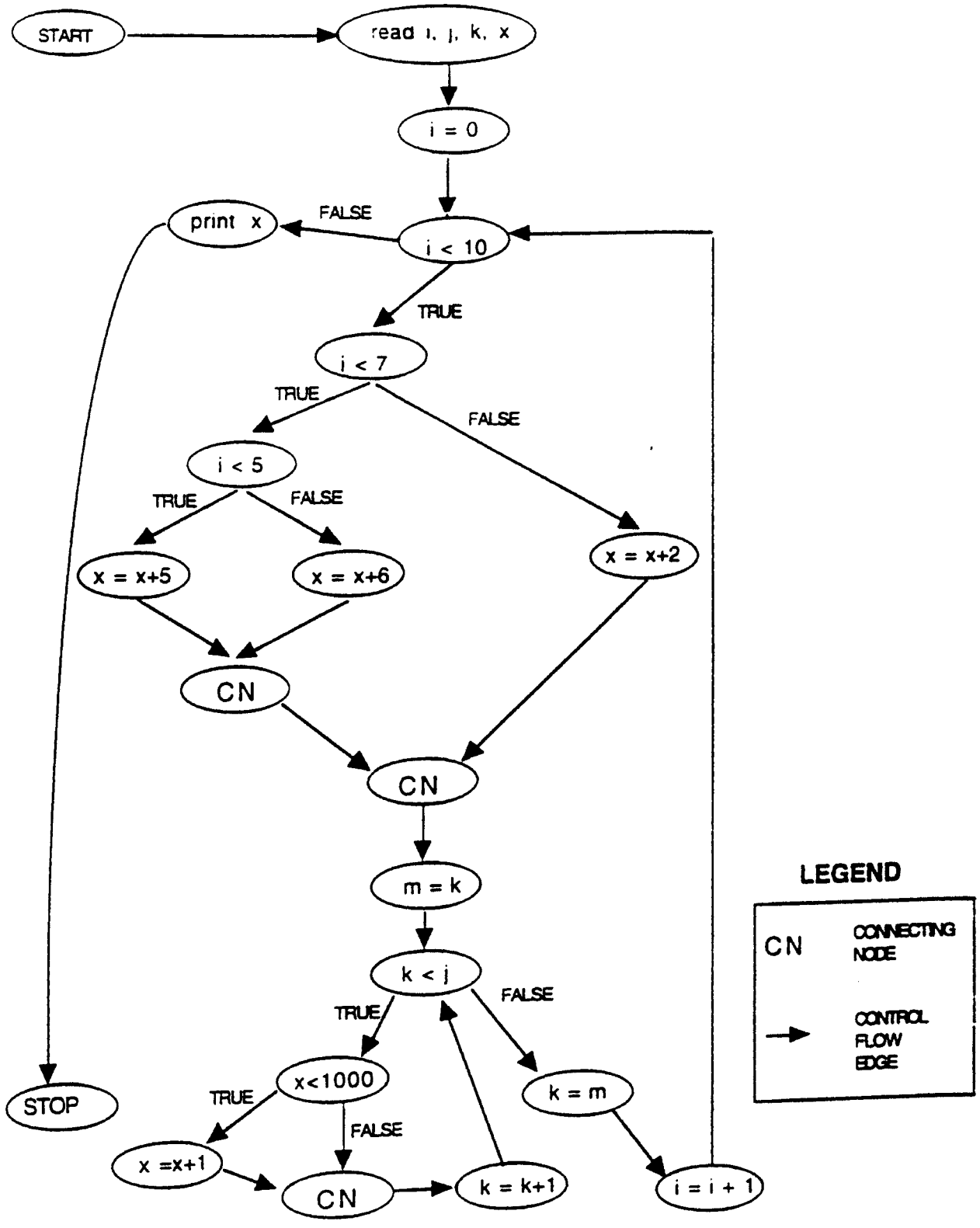

Figure 16. CFG as a tree structured directed graph for the program given in Figure 14

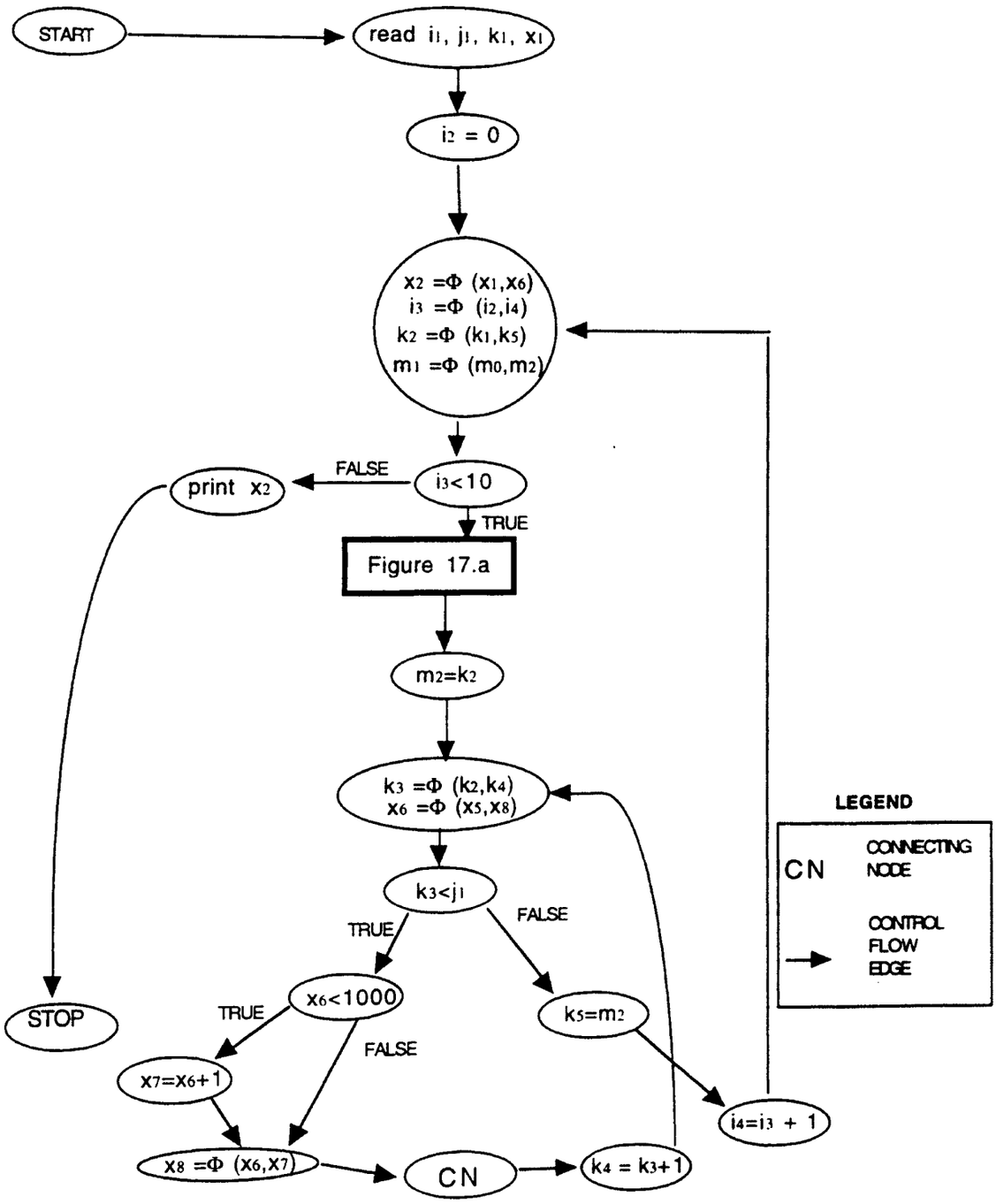

Figure 17. SSA form for the program given in Figure 14

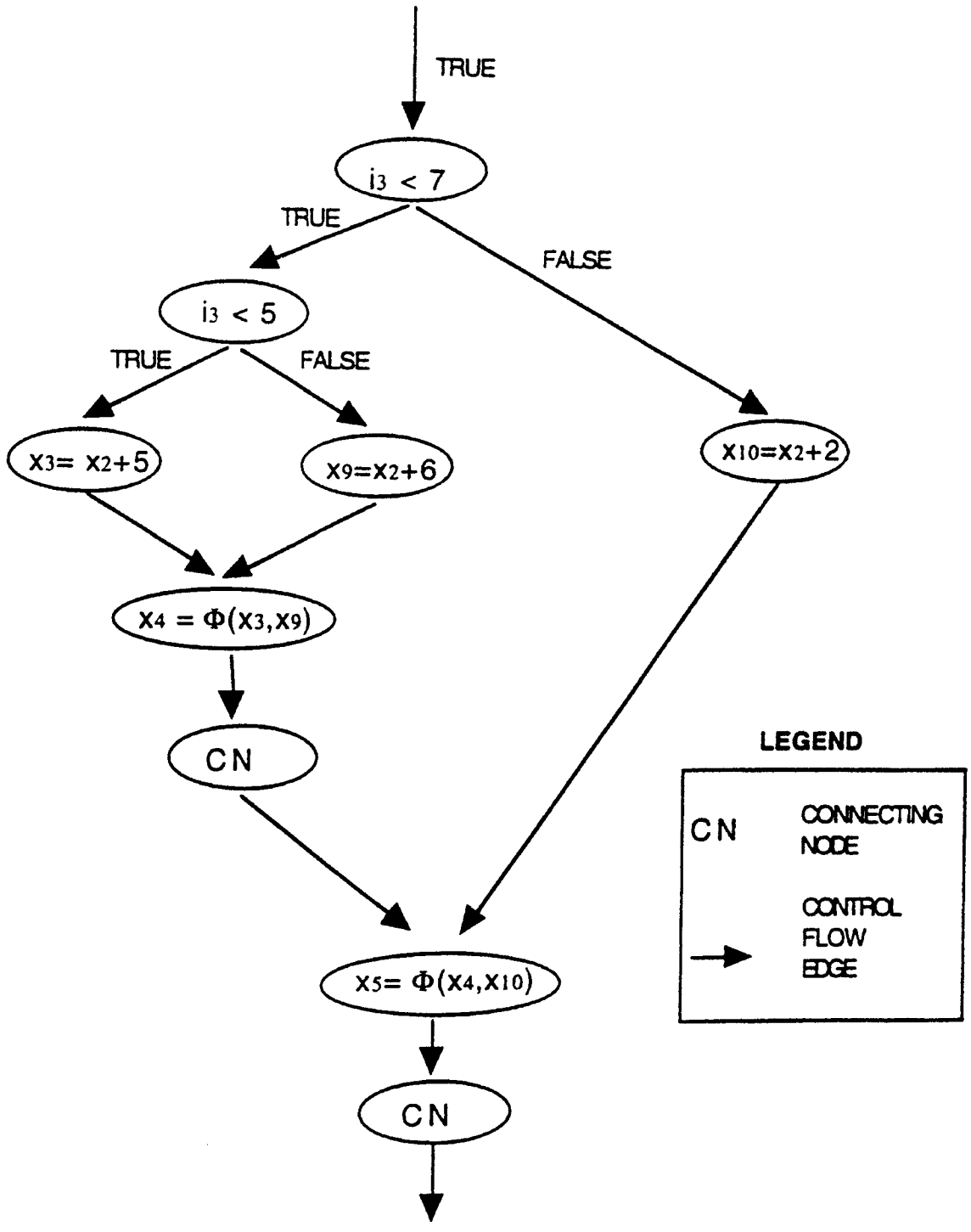

Figure 17.a SSA form for the if-then-else statement in Figure 14

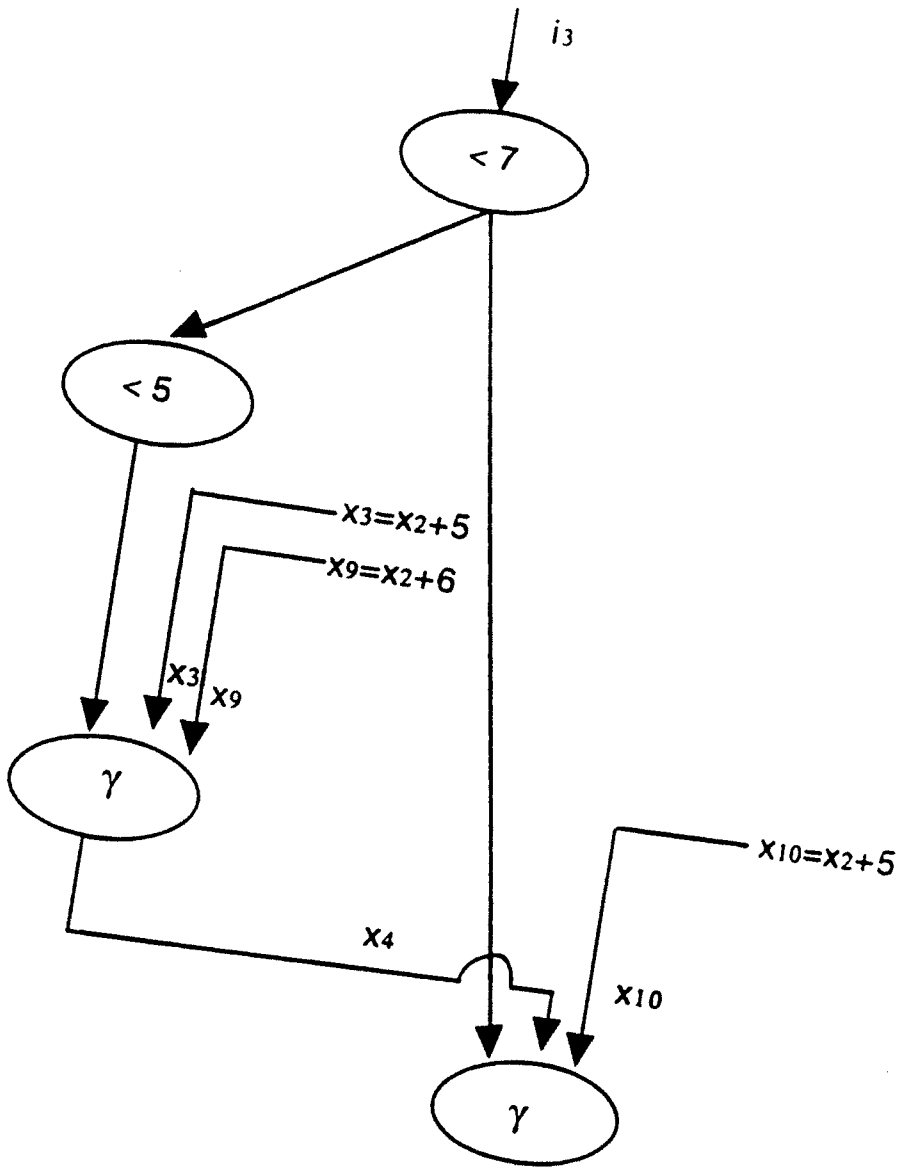

Figure 18. GSA form for the SSA form shown in Figure 17.a

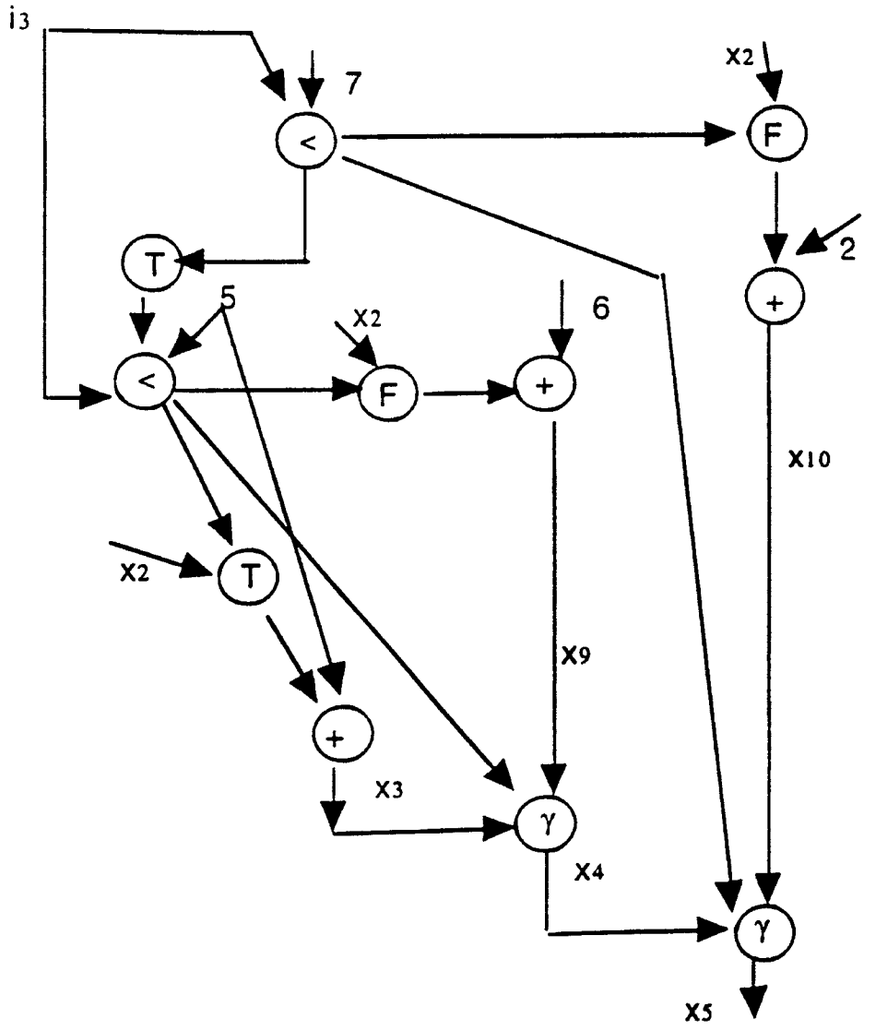

Figure 19. PDW for the GSA form shown in Figure 18

VITA

Premkumar John

Candidate for the Degree of

Master of Science

Thesis: TRANSLATION OF SIMPLE C PROGRAMS TO PROGRAM DEPENDENCE WEBS

Major Field: Computer Science

Biographical:

Personal Data: Born in Trichy, Taminl Nadu, India, August 16, 1970, son of Jainus D. John and Kesala John

Education: Graduated high school from St. John's Vestry Anglo-Indian Higher Secondary School, Trichy, India, May 1988; received Bachelor of Engineering Degree in Computer Science from Regional Engineering College, Trichy, Tamil Nadu, India, May 1992; completed requirements for the Master of Science degree in Computer Science at Oklahoma State University in July 1994.

Professional Experience: Para-professional, Operations, University Computer Center, Oklahoma State University, July 1993 to May 1994.