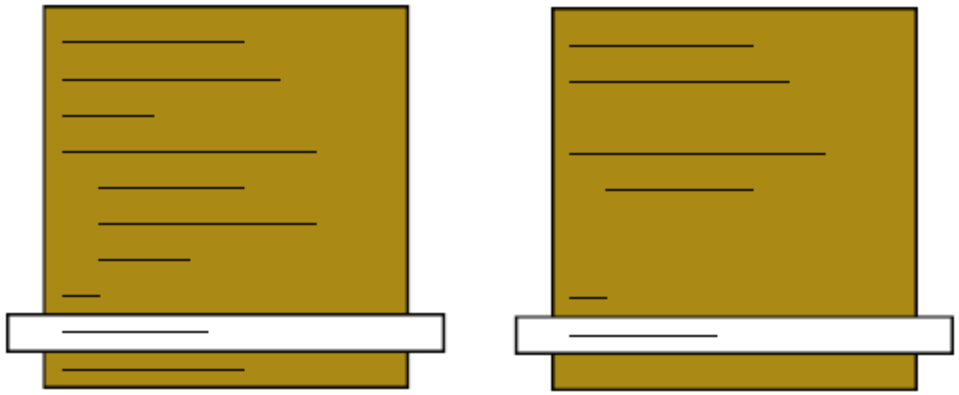
Program Slicing

Amin Ranj Bar

Introduction

- **Slicing:** decomposition technique that extracts relevant statements from a program
- **Executable backward static slicing:** Mark Weiser's 1979 Ph.D. thesis (U.Michigan)
- **Forward slicing:** Susan Horwitz, *et al* 1990 (U.Wisc.) [HRB90]
- **Dynamic slicing:** Bogdan Korel and Janusz Laski 1988



Why Slicing?

- *Software Maintenance*: changing source code without unwanted side effects
- *Testing*: reduce cost of regression testing after modifications (only run those tests that needed)
- *Debugging*: that's how slicing was discovered!
- *Software Quality Assurance*: validate interactions between safety-critical components
- *Reverse Engineering*: comprehending the design by abstracting out of the source code and the design decisions

What use?

- Understanding Programs
  - What is affected by what?
- Restructuring Programs
  - Isolation of separate “computational threads”
- Program Specialization and Reuse
  - Slices = specialized programs
  - Only reuse needed slices
- Program Differencing
  - Compare slices to identify changes
- Testing
  - What new test cases would improve coverage?
  - What regression tests must be rerun after a change?

Slicing

- Identifying statements in a program that may affect or be affected by the value of a variable

Slicing Overview

- Types of slices
  - Backward static slice
  - Executable slice
  - Forward static slice
  - Dynamic slice
  - Execution slice
- Levels of slices
  - Intraprocedural
  - Interprocedural

Types of Slicing (Backward Static)

- A backward slice of a program with respect to a program point **p** and set of program variables **V** consists of all statements and predicates in the program that may affect the value of variables in **V** at **p.**

- The program point **p** and the variables **V** together form the slicing criterion, usually written <**p, V**>

Types of Slicing (Backward Static)

```
1.  read (n)
2.  i := 1
3.  sum := 0
4.  product := 1
5.  while i <= n do
6.      sum := sum + i
7.      product := product * i
8.      i := i + 1
9.  write (sum)
10. write (product)
```

- Criterion <10, product>

Types of Slicing (Backward Static)

```
1.  read (n)
2.  i := 1
3.  sum := 0
4.  product := 1
5.  while i <= n do
6.      sum := sum + i
7.      product := product * i
8.      i := i + 1
9.  write (sum)
10. write (product)
```

- Criterion <10, product>

Types of Slicing (Executable)

- A slice is executable if the statements in the slice form a syntactically correct program that can be executed.

- If the slice is computed correctly (safely), the result of running the program that is the executable slice produces the same result for variables in **V** at **p** for all inputs.

Types of Slicing (Executable)

```
1.  read (n)
2.  i := 1
3.  sum := 0
4.  product := 1
5.  while i <= n do
6.      sum := sum + i
7.      product := product * i
8.      i := i + 1
9.  write (sum)
10. write (product)
```

```
1.  read (n)
2.  i := 1
3.
4.  product := 1
5.  while i <= n do
6.
7.      product := product * i
8.      i := i + 1
9.
10. write (product)
```

Criterion <10, product>

Types of Slicing (Forward Static)

- A forward slice of a program with respect to a program point **p** and set of program variables **V** consists of all statements and predicates in the program that may be affected by the value of variables in **V** at **p.**

- The program point **p** and the variables **V** together form the slicing criterion, usually written <**p, V**>.

Types of Slicing (Forward Static)

```
1.  read (n)
2.  i := 1
3.  sum := 0
4.  product := 1
5.  while i <= n do
6.      sum := sum + i
7.      product := product * i
8.      i := i + 1
9.  write (sum)
10. write (product)
```

- Criterion <3, sum>

Types of Slicing (Forward Static)

```
1.  read (n)
2.  i := 1
3.  sum := 0
4.  product := 1
5.  while i <= n do
6.      sum := sum + i
7.      product := product * i
8.      i := i + 1
9.  write (sum)
10. write (product)
```

- Criterion <3, sum>

Types of Slicing (Forward Static)

```
1.  read (n)
2.  i := 1
3.  sum := 0
4.  product := 1
5.  while i <= n do
6.      sum := sum + i
7.      product := product * i
8.      i := i + 1
9.  write (sum)
10. write (product)
```

- Criterion <1, n>

Types of Slicing (Forward Static)

```
1.  read (n)
2.  i := 1
3.  sum := 0
4.  product := 1
5.  while i <= n do
6.      sum := sum + i
7.      product := product * i
8.      i := i + 1
9.  write (sum)
10. write (product)
```

- Criterion <1, n>

Types of Slicing (Dynamic)

- A dynamic slice of a program with respect to an input value of a variable **v** at a program point **p** for a particular execution **e** of the program is the set of all statements in the program that affect the value of **v** at **p.**

- The program point **p,** the variables **V,** and the input i for **e** form the slicing criterion, usually written <i**, p, v**>. The slicing uses the execution history or trajectory for the program with input **i.**

Types of Slicing (Dynamic)

```
1.  read (n)
2.  for I := 1 to n do
3.      a := 2
4.      if c1 then
5.          if c2 then
6.              a := 4
7.          else
8.              a := 6
9.      z := a
10. write (z)
```

- Input n is 1; c1, c2 both true
- Execution history is

  $1^1, 2^1, 3^1, 4^1, 5^1, 6^1, 9^1, 2^2, 10^1$

- Criterion<1, 10^1, z>

Types of Slicing (Dynamic)

```
1.  read (n)
2.  for I := 1 to n do
3.      a := 2
4.      if c1 then
5.          if c2 then
6.              a := 4
7.          else
8.              a := 6
9.      z := a
10. write (z)
```

- Input n is 1; c1, c2 both true
- Execution history is

  $1^1, 2^1, 3^1, 4^1, 5^1, 6^1, 9^1, 2^2, 10^1$

- Criterion<1, 10^1, z>

Why we don't take a:=2 ?

Types of Slicing (Dynamic)

```
1.  read (n)
2.  for I := 1 to n do
3.      a := 2
4.      if c1 then
5.          if c2 then
6.              a := 4
7.          else
8.              a := 6
9.      z := a
10. write (z)
```

```
1.  read (n)
2.  for I := 1 to n do
3.      a := 2
4.      if c1 then
5.          if c2 then
6.              a := 4
7.          else
8.              a := 6
9.      z := a
10. write (z)
```

Static slice <10, z>

Types of Slicing (Dynamic)

```
1.  read (n)
2.  for I := 1 to n do
3.      a := 2
4.      if c1 then
5.          if c2 then
6.              a := 4
7.          else
8.              a := 6
9.      z := a
10. write (z)
```

- Input n is 2; c1, c2 false on first iteration and true on second iteration
- Execution history is
- Criterion<2, 10^1, z>

Types of Slicing (Dynamic)

```
1.  read (n)
2.  for I := 1 to n do
3.      a := 2
4.      if c1 then
5.          if c2 then
6.              a := 4
7.          else
8.              a := 6
9.      z := a
10. write (z)
```

- Input n is 2; c1, c2 false on first iteration and true on second iteration
- Execution history is

  $1^1, 2^1, 3^1, 4^1, 9^1, 2^2, 3^2, 4^2, 5^1, 6^1, 9^2, 2^3, 10^1$

- Criterion<2, 10^1, z>

Types of Slicing (Dynamic)

```
1.  read (n)
2.  for I := 1 to n do
3.      a := 2
4.      if c1 then
5.          if c2 then
6.              a := 4
7.          else
8.              a := 6
9.      z := a
10. write (z)
```

```
1.  read (n)
2.  for I := 1 to n do
3.      a := 2
4.      if c1 then
5.          if c2 then
6.              a := 4
7.          else
8.              a := 6
9.      z := a
10. write (z)
```

Static slice <10, z>

Types of Slicing (Execution)

- An execution slice of a program with respect to an input value of a variable **v** is the set of statements in the program that are executed with input **v.**

Types of Slicing (Execution)

```
1.  read (n)
2.  for I := 1 to n do
3.      a := 2
4.      if c1 then
5.          if c2 then
6.              a := 4
7.          else
8.              a := 6
9.      z := a
10. write (z)
```

- Input n is 2; c1, c2 false on first iteration and true on second iteration
- Execution history is

  $1^1, 2^1, 3^1, 4^1, 9^1, 2^2, 3^2, 4^2, 5^1, 6^1, 9^2, 2^3, 10^1$

- Execution slice is

  1, 2, 3, 4, 5, 6, 9, 10

Types of Slicing (Execution)

```
1.  read (n)
2.  for I := 1 to n do
3.      a := 2
4.      if c1 then
5.          if c2 then
6.              a := 4
7.          else
8.              a := 6
9.      z := a
10. write (z)
```

- Input n is 2; c1, c2 false on first iteration and true on second iteration
- Execution history is

  $1^1, 2^1, 3^1, 4^1, 9^1, 2^2, 3^2, 4^2, 5^1, 6^1, 9^2, 2^3, 10^1$

- Execution slice is

  1, 2, 3, 4, 5, 6, 9, 10

Why we don't take a:=6 ?

Program Slicing

- A ***backward slice*** with respect to variable *v* at program point *p* is the program subset that may influence the value of variable *v* at point *p*.

- A ***forward slice*** with respect to variable *v* at program point *p* is the program subset that may be influenced by the value of variable *v* at point *p*.

Example

```
proc Main
        sum := 0
        i := 1
        while i < 11 do
                sum := sum + i
                i := i + 1
        od
        output(sum)
        output(i)
end
```

Backward Slice

```
proc Main
        sum := 0
        i := 1
        while i < 11 do
                sum := sum + i
                i := i + 1
        od
        output(sum)
        output(i)
end
```

Backward slice w.r.t. variable `i` in statement "`output (i)`"

Forward Slice

```
proc Main
        sum := 0
        i := 1
        while i < 11 do
                sum := sum + i
                i := i + 1
        od
        output(sum)
        output(i)
end
```

Forward slice w.r.t. variable `sum` in statement "`sum:=sum+1`"

How are slices computed?

- Reachability in a dependence graph
  - Program Dependence Graph (PDG)
    - Dependences within one procedure
    - *Intra*procedural slicing is reachability in one PDG
  - System Dependence Graph (SDG)
    - Dependences within entire system
    - *Inter*procedural slicing is reachability in the SDG

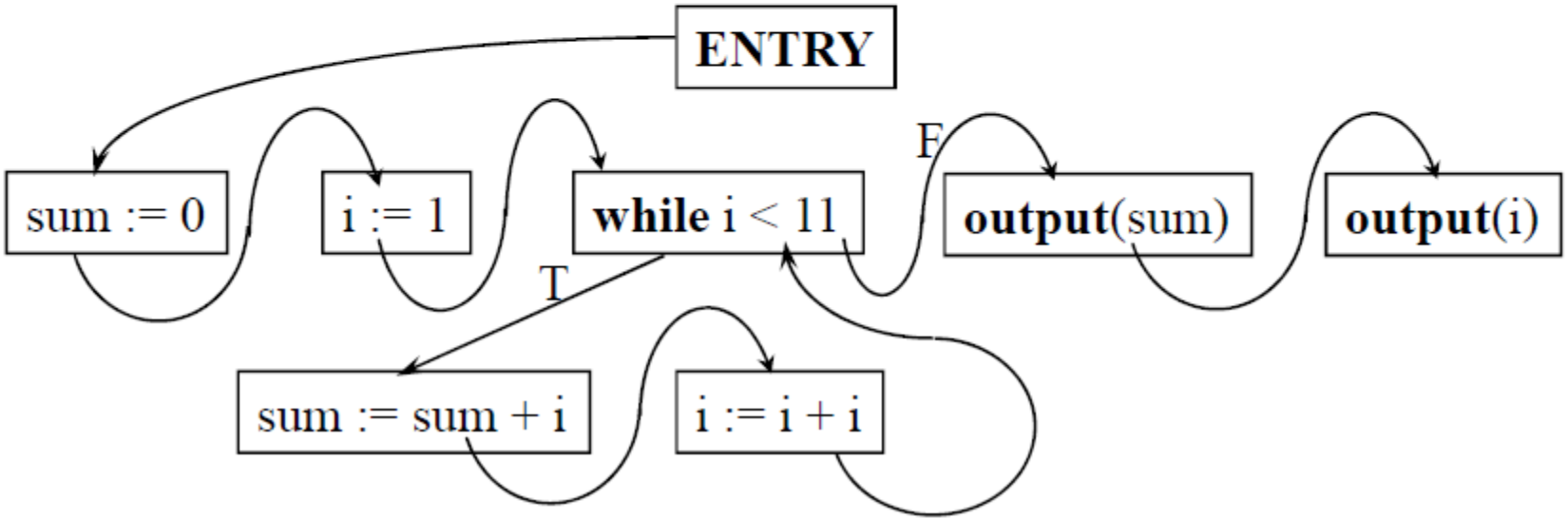
How is a PDG computed?

- Control Flow Graph (CFG)
- PDG is union of:
  - Control Dependence Graph
  - Flow Dependence Graph

  computed from CFG

Control Flow Graph

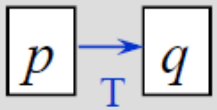
```
proc Main
        sum := 0
        i := 1
        while i < 11 do
                sum := sum + i
                i := i + 1
        od
        output(sum)
        output(i)
end
```



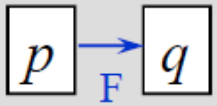
Control Dependence Graph (CDG)

```
proc Main
        sum := 0
        i := 1
        while i < 11 do
                sum := sum + i
                i := i + 1
        od
        output(sum)
        output(i)
end
```

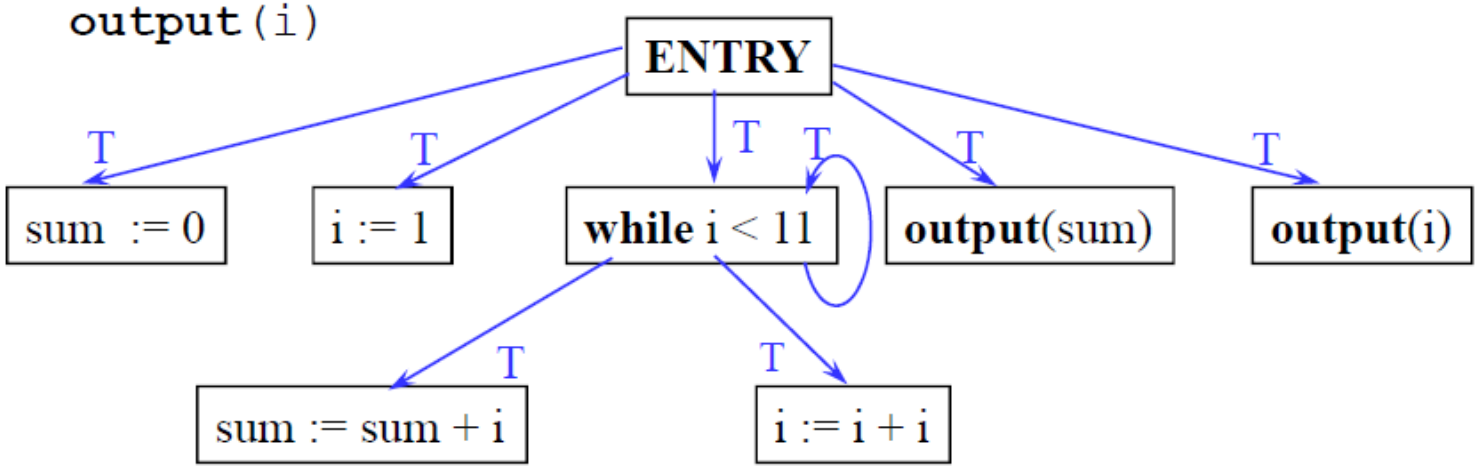
Control dependence



q is reached from *p* if condition *p* is true (T), not otherwise.



Similar for false (F).



A control dependence edge from vertex *vi* to vertex *vj* ⇔

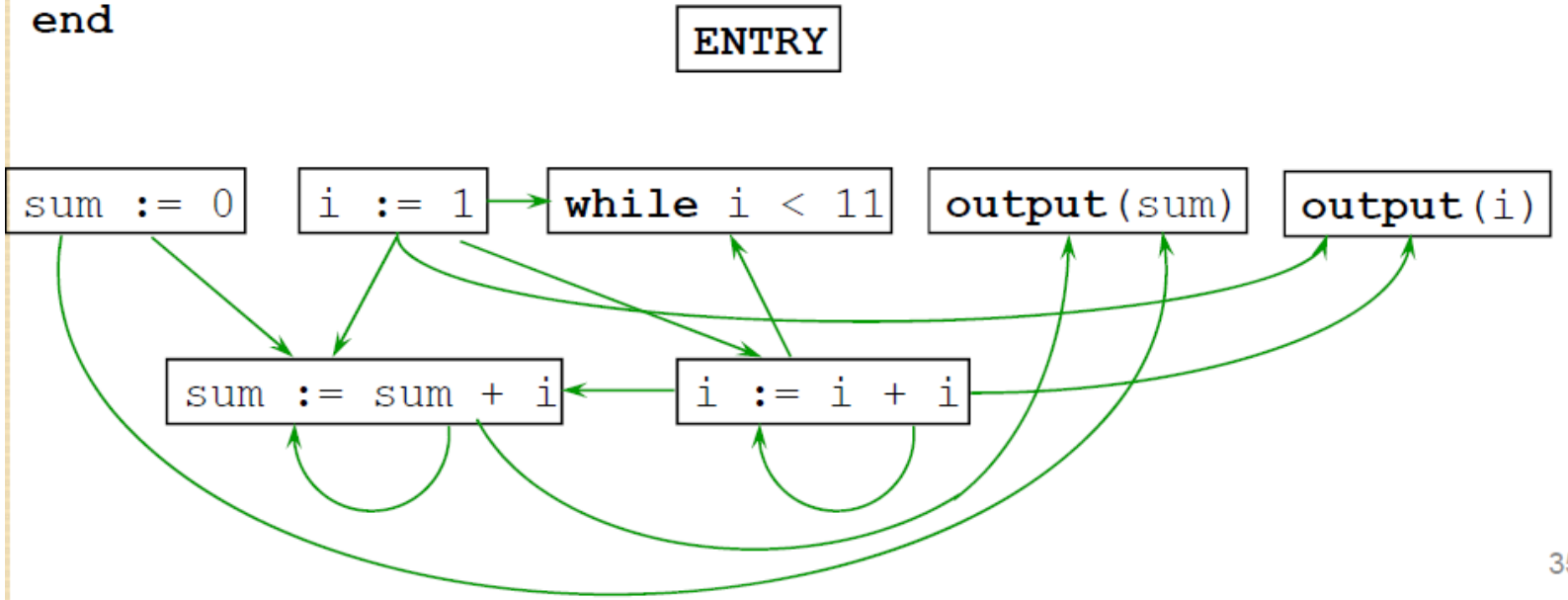
- *vi* is the entry vertex, and *vj* represents a component of P that is not nested in any loop or conditional (labeled true).
- *vi* represents a control predicate, and *vj* represents a component of P immediately nested within the loop or conditional whose predicate is represented by *vi*.

Flow (Data) Dependence Graph (DDG)

```
proc Main
        sum := 0
        i := 1
        while i < 11 do
                sum := sum + i
                i := i + 1
        od
        output(sum)
        output(i)
end
```

Flow dependence

$p \rightarrow q$ Value of variable assigned at p may be used at q.

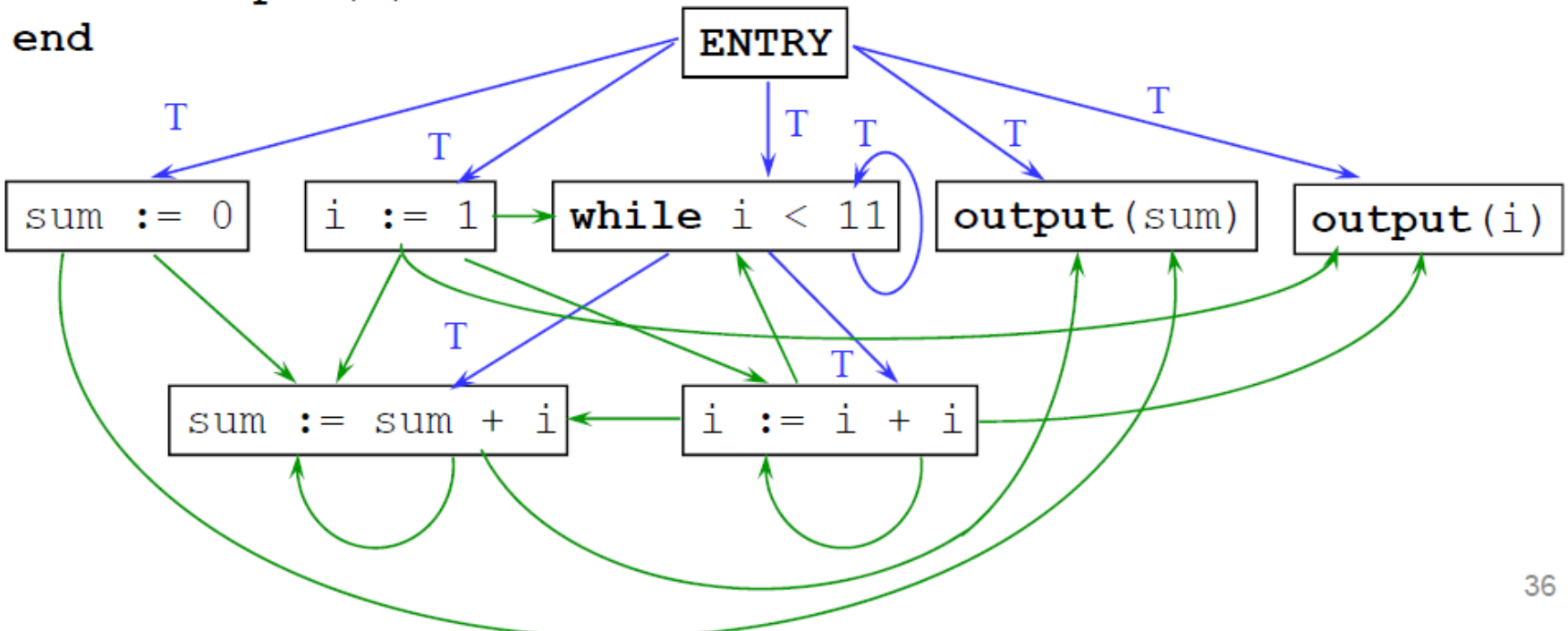


Program Dependence Graph

```
proc Main
        sum := 0
        i := 1
        while i < 11 do
                sum := sum + i
                i := i + 1
        od
        output(sum)
        output(i)
end
```

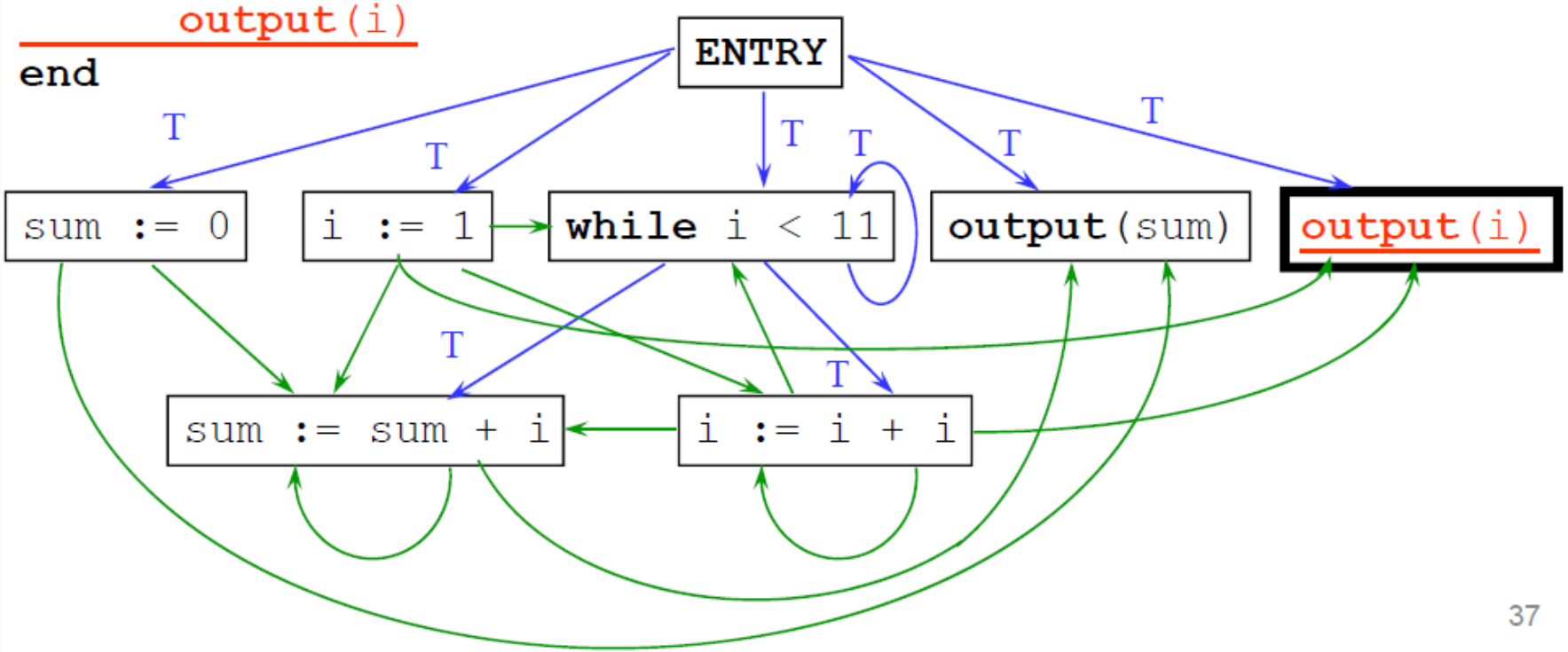
Control dependence

Flow dependence



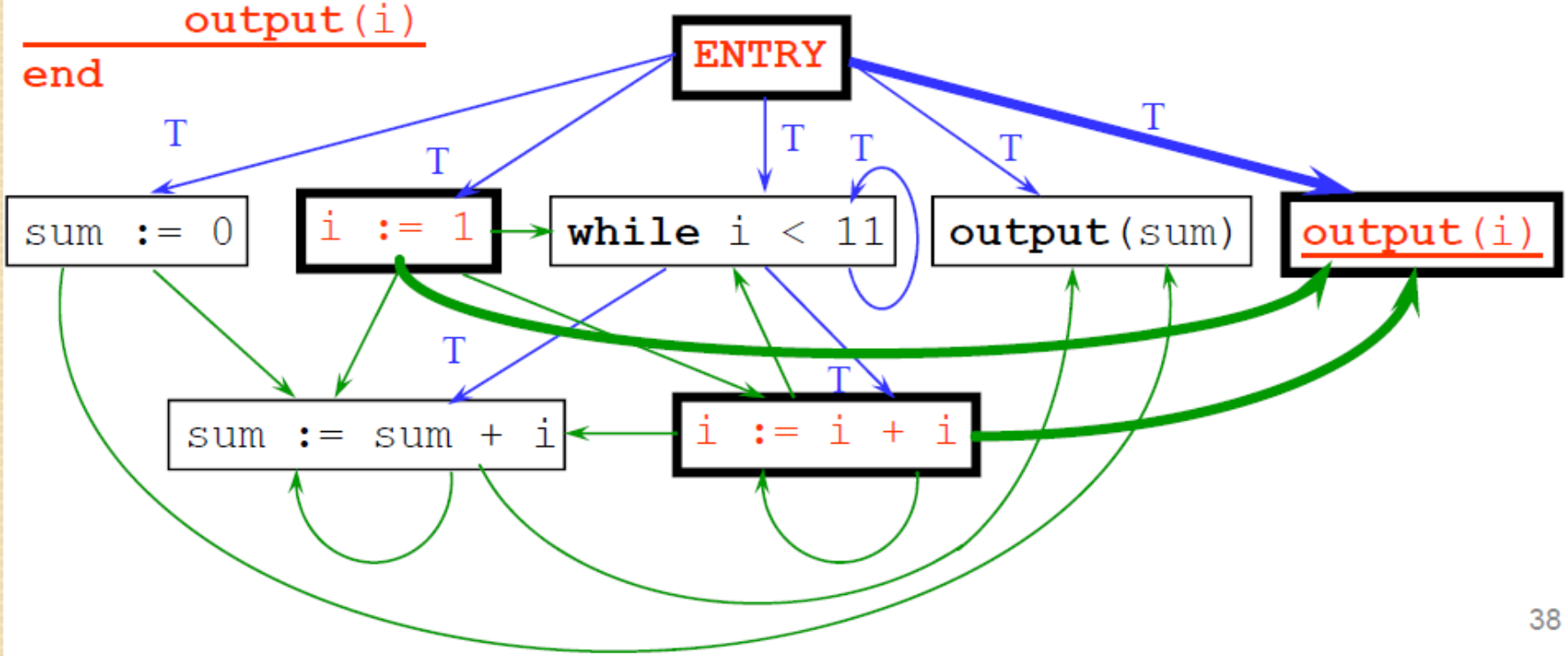
Backward Slice

```
proc Main
        sum := 0
        i := 1
        while i < 11 do
                sum := sum + i
                i := i + 1
        od
        output(sum)
        output(i)
end
```



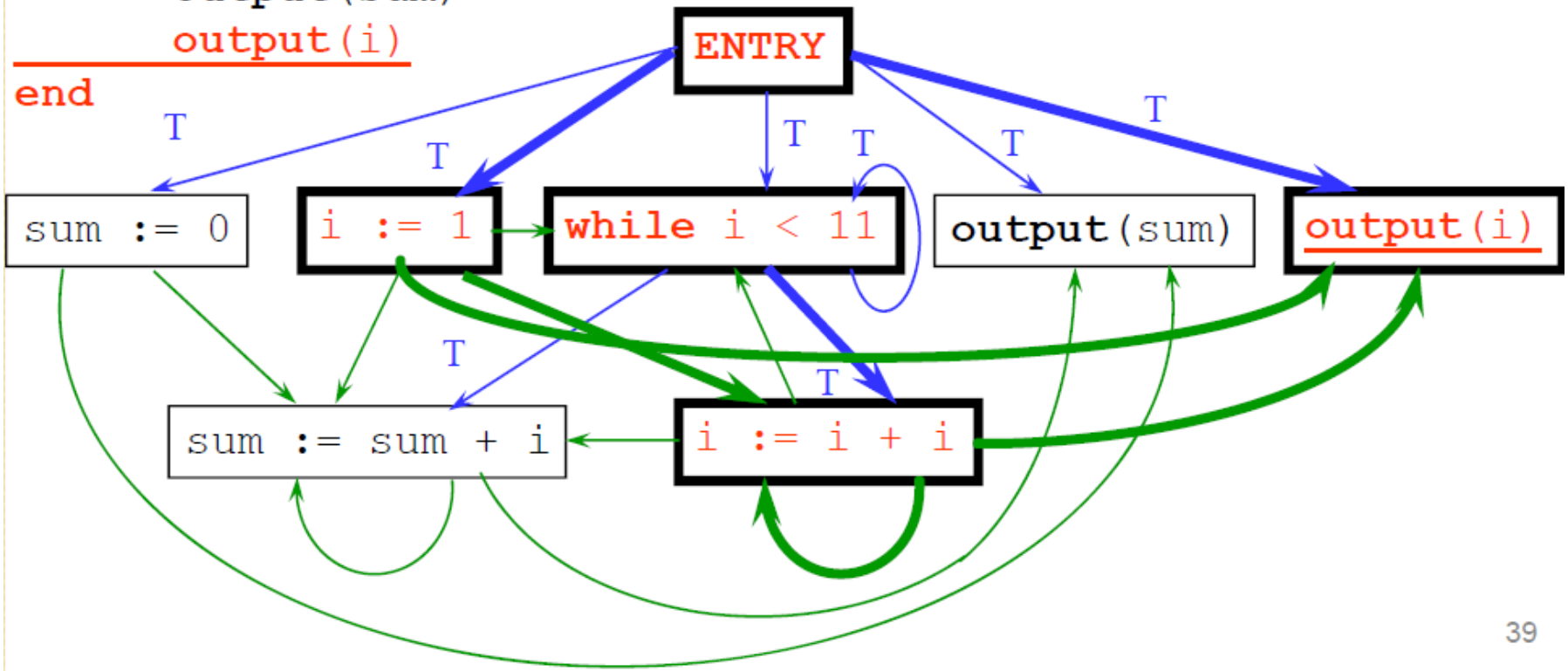
Backward Slice (2)

```
proc Main
        sum := 0
        i := 1
        while i < 11 do
                sum := sum + i
                i := i + 1
        od
        output(sum)
        output(i)
end
```



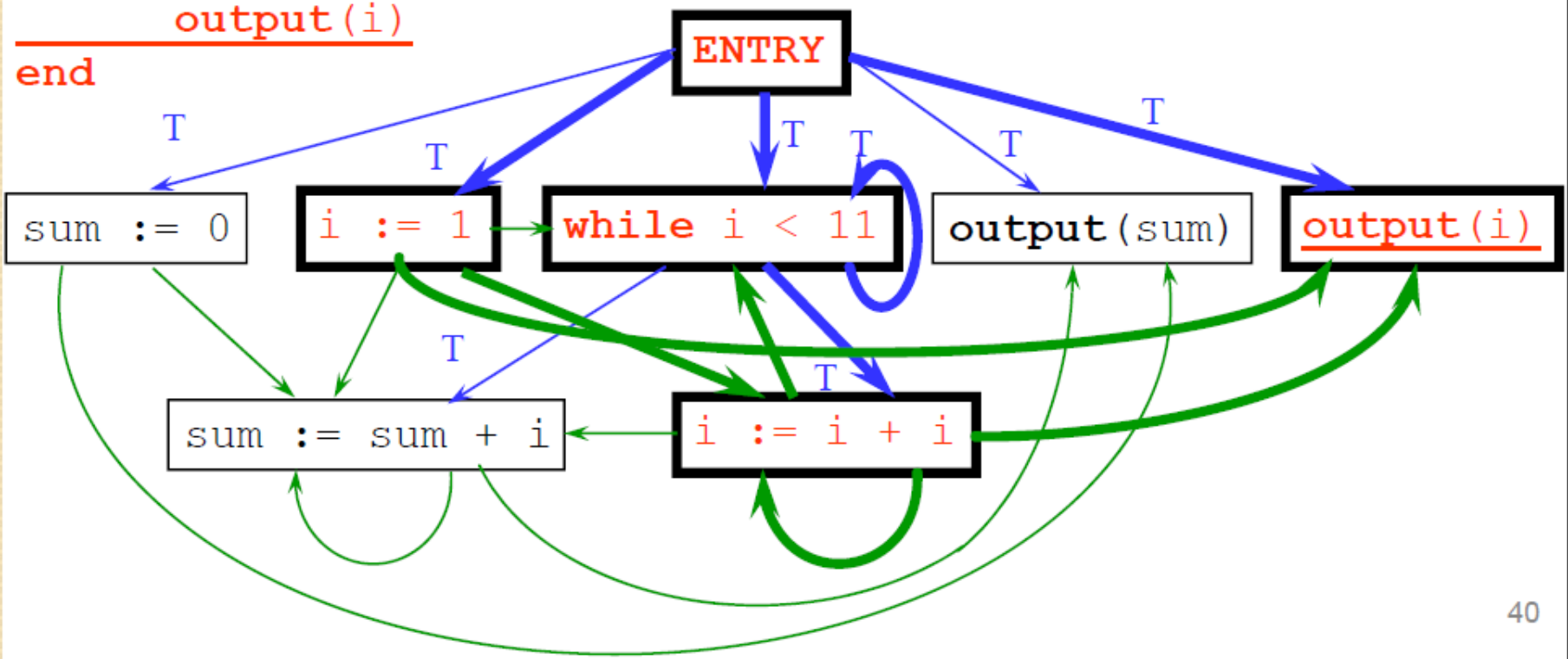
Backward Slice (3)

```
proc Main
        sum := 0
        i := 1
        while i < 11 do
                sum := sum + i
                i := i + 1
        od
        output(sum)
        output(i)
end
```



Backward Slice (4)

```
proc Main
        sum := 0
        i := 1
        while i < 11 do
                sum := sum + i
                i := i + 1
        od
        output(sum)
        output(i)
end
```



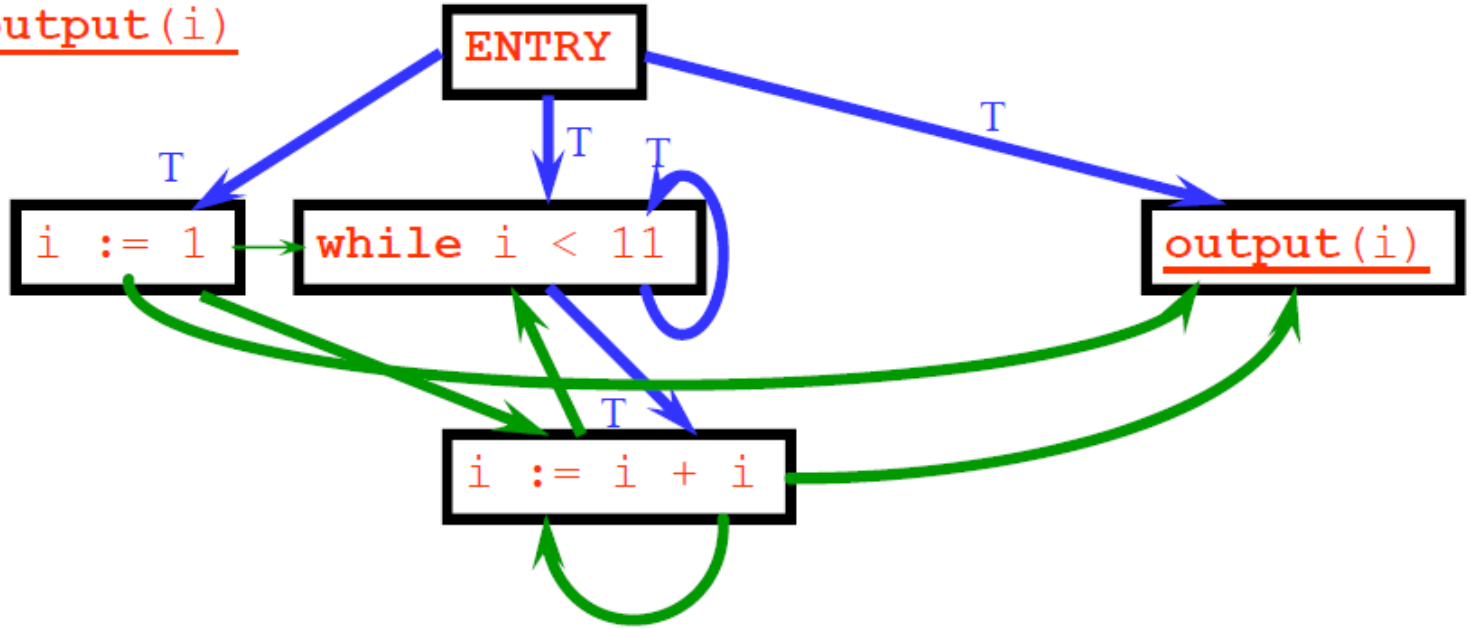
Slice Extraction

```
proc Main

        i := 1
        while i < 11 do

                i := i + 1
        od

        output(i)
end
```



Line-Character-Count Program

```
void line_char_count(FILE *f)
{
int lines = 0;
int chars;
BOOLEAN eof_flag = FALSE;
int n;
extern void scan_line(FILE *f, BOOLEAN *bptr, int *iptr);
scan_line(f, &eof_flag, &n);
chars = n;
while(eof_flag == FALSE){
    lines = lines + 1;
    scan_line(f, &eof_flag, &n);
    chars = chars + n;
    }
printf("lines = %d\n", lines);
printf("chars = %d\n", chars);
}
```

Character-Count Program

```
void char_count(FILE *f)
{
int lines = 0;
int chars;
BOOLEAN eof_flag = FALSE;
int n;
extern void scan_line(FILE *f, BOOLEAN *bptr, int *iptr);
scan_line(f, &eof_flag, &n);
chars = n;
while(eof_flag == FALSE){
    lines = lines + 1;
    scan_line(f, &eof_flag, &n);
    chars = chars + n;
    }
printf("lines = %d\n", lines);
printf("chars = %d\n", chars);
}
```

Line-Count Program

```
void line_count(FILE *f)
{
int lines = 0;
int chars;
BOOLEAN eof_flag = FALSE;
int n;
extern void scan_line2(FILE *f, BOOLEAN *bptr, int *iptr);
scan_line2(f, &eof_flag, &n);
chars = n;
while(eof_flag == FALSE){
    lines = lines + 1;
    scan_line2(f, &eof_flag, &n);
    chars = chars + n;
    }
printf("lines = %d\n", lines);
printf("chars = %d\n", chars);
}
```

*Inter*procedural Slice

```
proc Main
  sum := 0
  i := 1
  while i < 11 do
     call Add(sum, i)
     call Add(i,1)
  od
  output(sum)
  output(i)
end
```

```
proc Add(x: inout, y:
    in)
    x := x + y
end
```

*Inter*procedural Slice

```
proc Main
   sum := 0
   i := 1
   while i < 11 do
      call Add(sum, i)
      call Add(i,1)
   od
   output(sum)
   output(i)
end
```

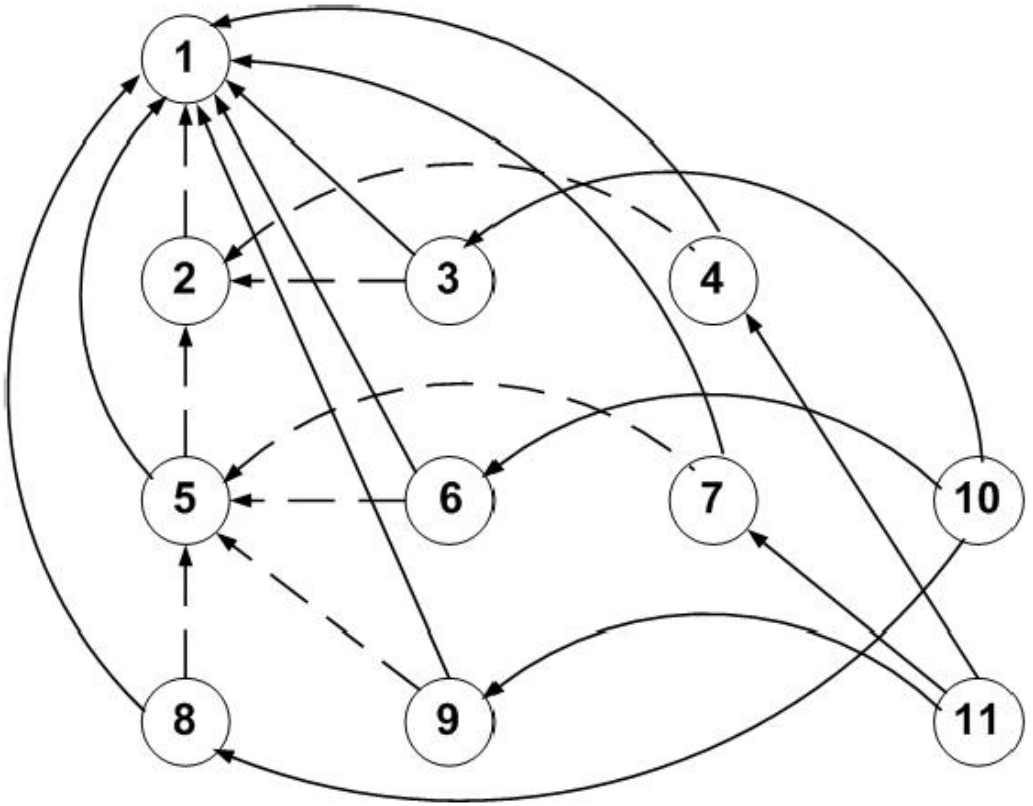
```
proc Add(x: inout, y:
   in)
   x := x + y
end
```

Dynamic Program Slicing

- Static program slicing is concerned with all the statements that *could* influence the value of the variable occurrence for *any* input.
- Dynamic program slicing is concerned with all the statements that *did* affect the value of the variable occurrence for the *current* input.
- It is especially suitable for program debugging.
- Typically, when debugging the value of a variable at some program statement is observed to be incorrect. dynamic program slicing is used to identify relevant subset of the program so as to look for the possible cause of the error.
- Here, four approaches for dynamic program slicing [AH90] are introduced.

Example 1 – PDG and Static Slicing

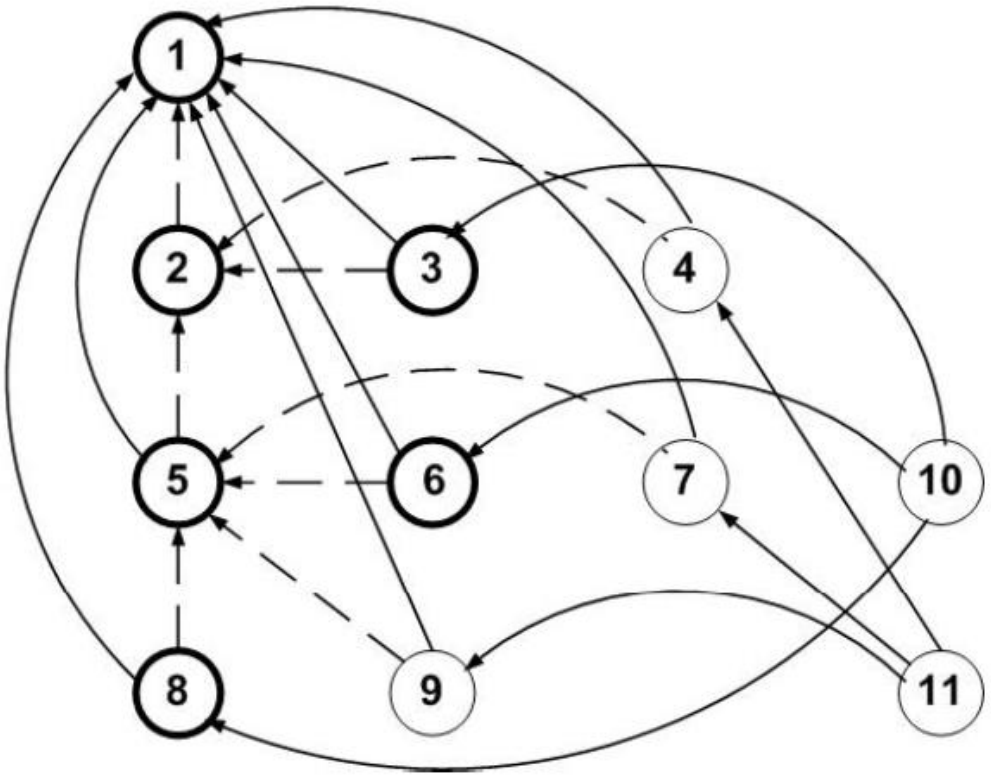
```
        begin
S1:       read(X);
S2:       if (X < 0)
          then
S3:         Y := f1(X);
S4:         Z := g1(X);
          else
S5:         if (X = 0)
              then
S6:             Y := f1(X);
S7:             Z := g2(X);
              else
S8:             Y := f3(X);
S9:             Z := g3(X);
              end_if;
          end_if;
S10:      write(Y);
S11:      write(Z);
        end.
```



What's the static slice with respect to variable Y at statement 10?

Example 1 – PDG and Static Slicing

```
        begin
S1:       read(X);
S2:       if (X < 0)
          then
S3:         Y := f1(X);
S4:         Z := g1(X);
          else
S5:         if (X = 0)
              then
S6:             Y := f1(X);
S7:             Z := g2(X);
              else
S8:             Y := f3(X);
S9:             Z := g3(X);
              end_if;
          end_if;
S10:      write(Y);
S11:      write(Z);
        end.
```



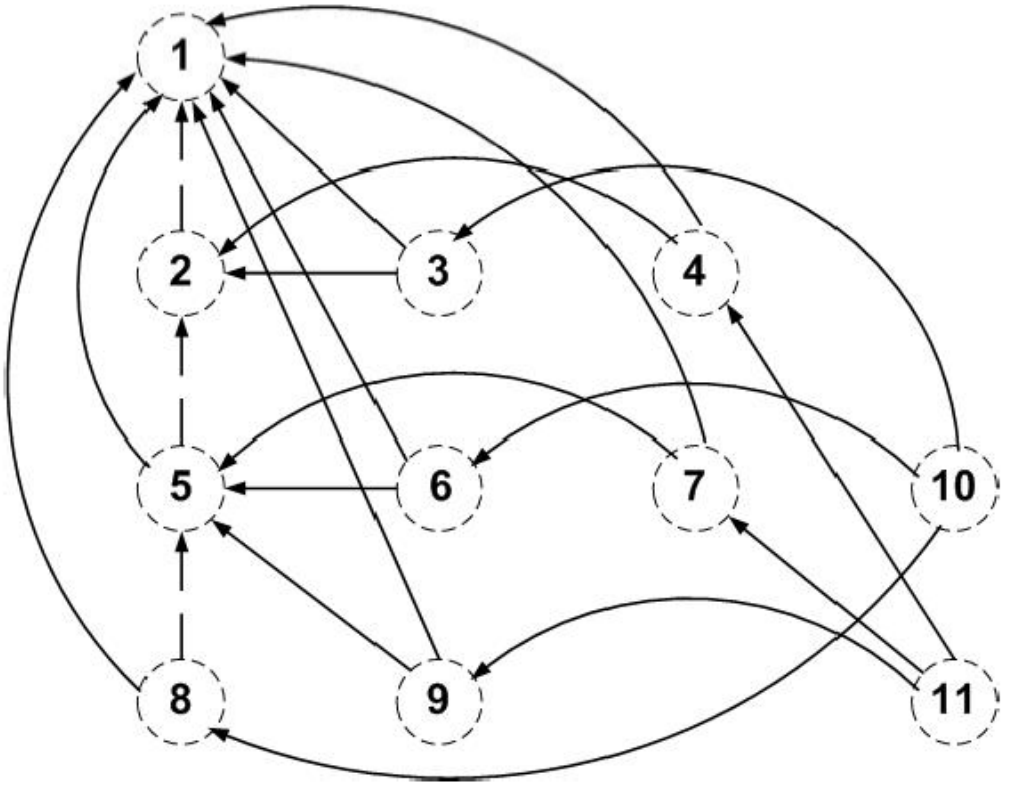
The static slice with respect to variable Y at statement 10 is {1, 2, 3, 5, 6, 8}.

Dynamic Slicing: Approach 1

- *Approach 1*
  - To obtain the dynamic slice with respect to a variable for a given execution history, first take the “projection” of the Program Dependence Graph with respect to the nodes that occur in the execution history, and then use the static slicing algorithm on the projected Dependence Graph to find the desired dynamic slice.

Example 1 – Dynamic Slicing, Approach 1

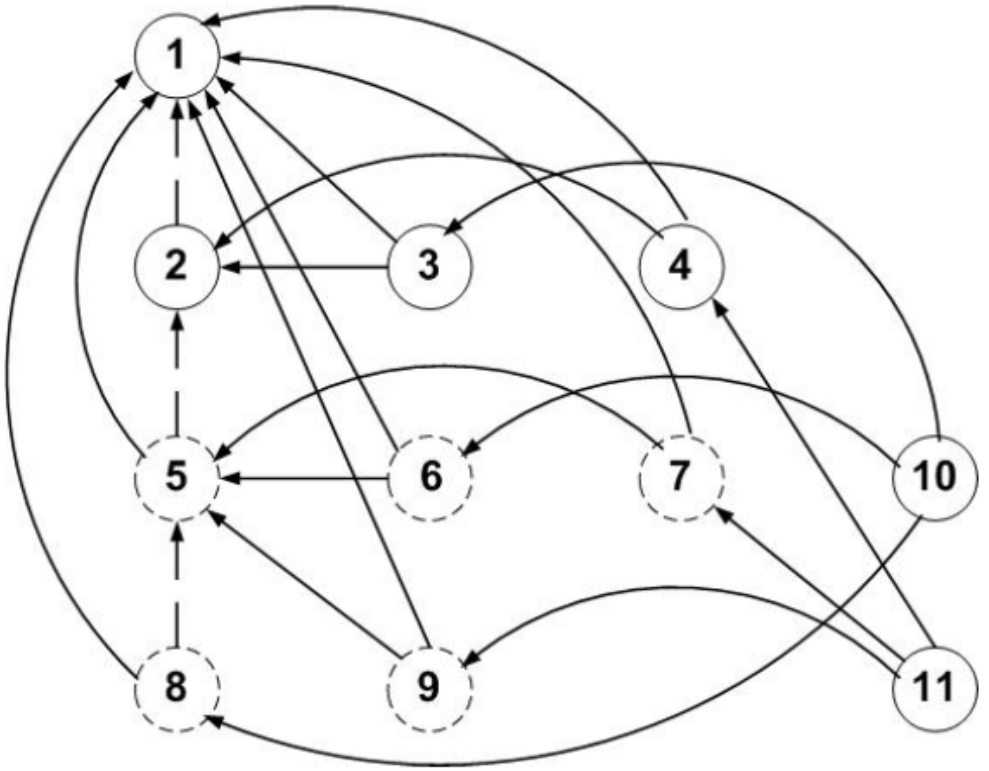
```
        begin
S1:       read(X);
S2:       if (X < 0)
          then
S3:         Y := f1(X);
S4:         Z := g1(X);
          else
S5:         if (X = 0)
              then
S6:             Y := f1(X);
S7:             Z := g2(X);
              else
S8:             Y := f3(X);
S9:             Z := g3(X);
              end_if;
          end_if;
S10:      write(Y);
S11:      write(Z);
        end.
```



What's the dynamic slice with respect to variable Y at statement 10, for test case X = -1?

Example 1
– Dynamic Slicing, Approach 1 (cont.)

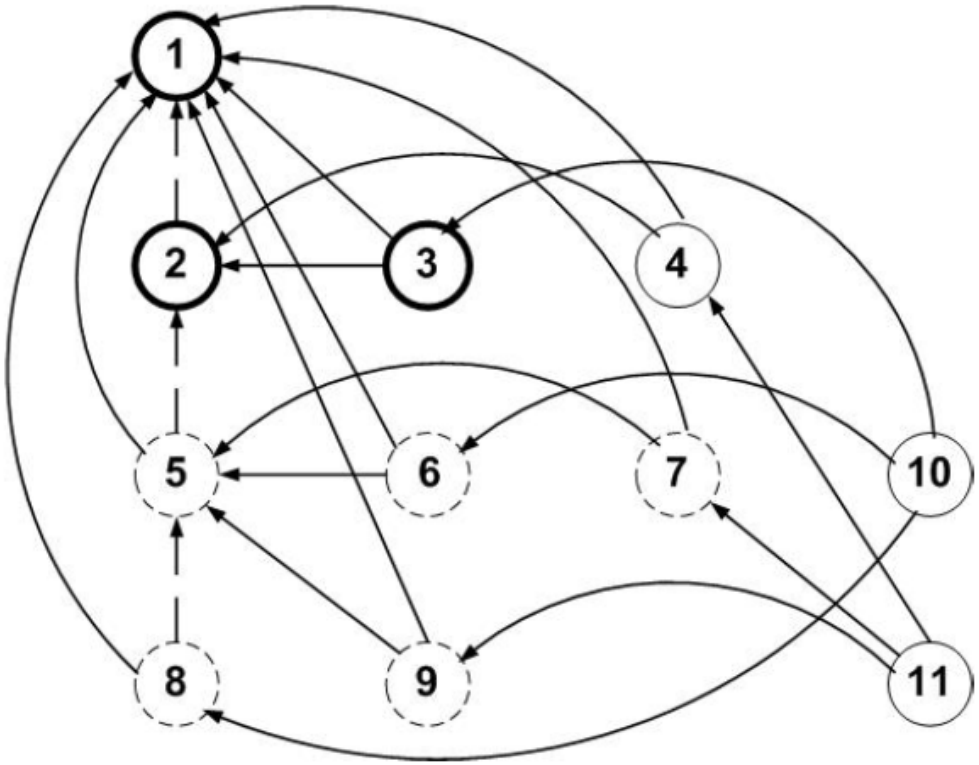
```
        begin
S1:       read(X);
S2:       if (X < 0)
          then
S3:         Y := f1(X);
S4:         Z := g1(X);
          else
S5:         if (X = 0)
              then
S6:             Y := f1(X);
S7:             Z := g2(X);
              else
S8:             Y := f3(X);
S9:             Z := g3(X);
              end_if;
          end_if;
S10:      write(Y);
S11:      write(Z);
        end.
```



- Draw all the nodes in the execution history as solid nodes.
- For test-case X = -1, the execution history is <1, 2, 3, 4, 10, 11>.

Example 1
– Dynamic Slicing, Approach 1 (cont.)

```
        begin
S1:       read(X);
S2:       if (X < 0)
          then
S3:         Y := f1(X);
S4:         Z := g1(X);
          else
S5:         if (X = 0)
              then
S6:             Y := f1(X);
S7:             Z := g2(X);
              else
S8:             Y := f3(X);
S9:             Z := g3(X);
              end_if;
          end_if;
S10:      write(Y);
S11:      write(Z);
        end.
```



- The graph is traversed only for solid nodes, beginning at node 3, the last definition of Y in the execution history.
- The dynamic slice is {1, 2, 3}.

Weakness of Approach 1

- The problem with Approach 1 lies in the fact that,
  - A statement may have multiple reaching definitions of the same variable in the program flow-graph, and hence it may have multiple out-going data dependence edges for the same variable in the Program Dependence Graph.

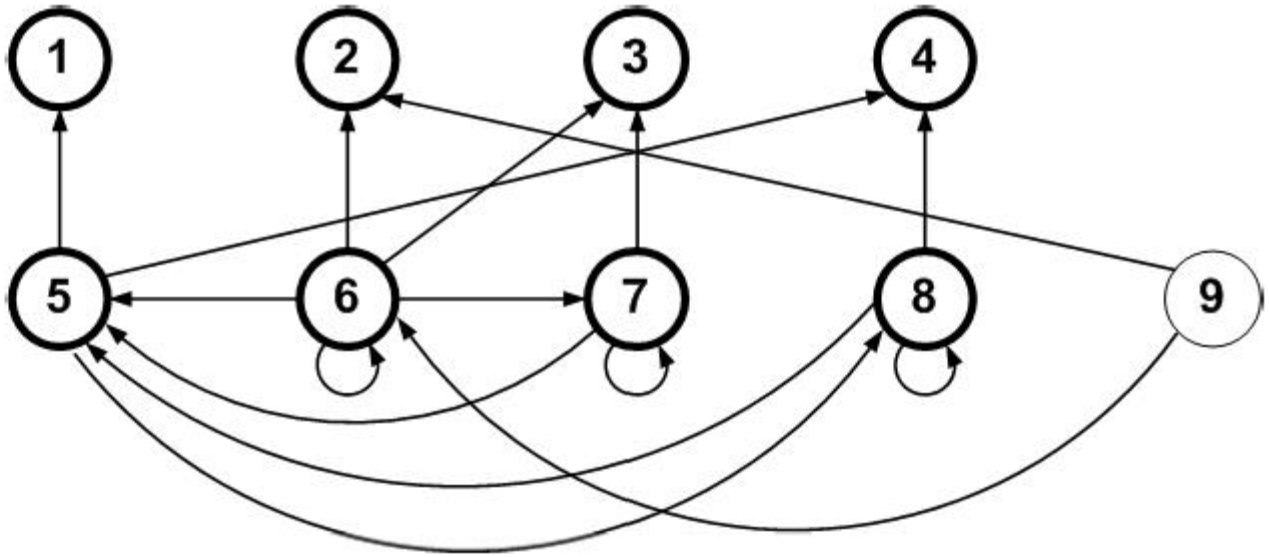
Example 2 – Dynamic Slicing, Approach 1

```
      begin
S1:     read(N);
S2:     Z := 0;
S3:     Y := 0;
S4:     I := 1;
S5:     while (I <= N)
        do
S6:         Z := f1(Z, Y);
S7:         Y := f2(Y);
S8:         I := I + 1;
        end-while;
S9:     write(Z)
      end.
```

What's the dynamic slice with respect to variable Z at the end of the execution, for test case N = 1?

Example 2 – Dynamic Slicing, Approach 1(cont.)

```
      begin
S1:     read(N);
S2:     Z := 0;
S3:     Y := 0;
S4:     I := 1;
S5:     while (I <= N)
        do
S6:         Z := f1(Z, Y);
S7:         Y := f2(Y);
S8:         I := I + 1;
        end-while;
S9:     write(Z)
      end.
```



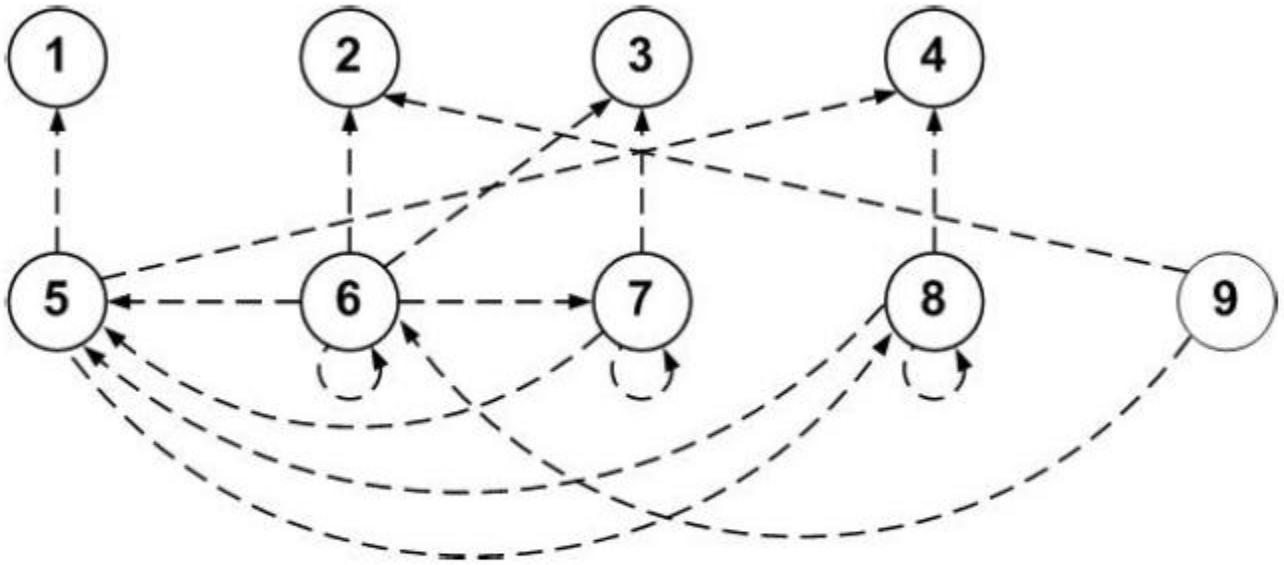
- The dynamic slice is {1, 2, 3, 4, 5, 6, 7, 8}.
- The value assigned to Y at statement 7 is never used later.

Dynamic Slicing: Approach 2

- *Approach 2*
  - Mark the edges of the Program Dependence Graph as the corresponding dependencies arise during the program execution; then traverse the graph only along the marked edges to find the slice.

Example 2 – Dynamic Slicing, Approach 2

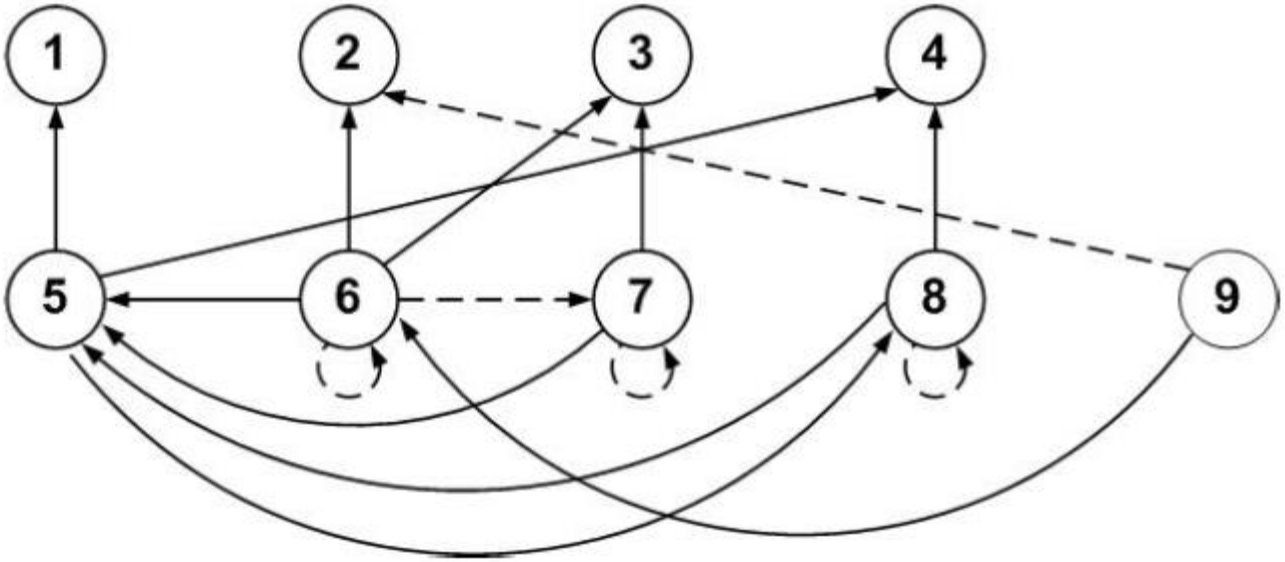
```
      begin
S1:     read(N);
S2:     Z := 0;
S3:     Y := 0;
S4:     I := 1;
S5:     while (I <= N)
        do
S6:         Z := f1(Z, Y);
S7:         Y := f2(Y);
S8:         I := I + 1;
        end-while;
S9:     write(Z)
      end.
```



What's the dynamic slice with respect to variable Z at the end of the execution, for test case N = 1?

Example 2 – Dynamic Slicing, Approach 2

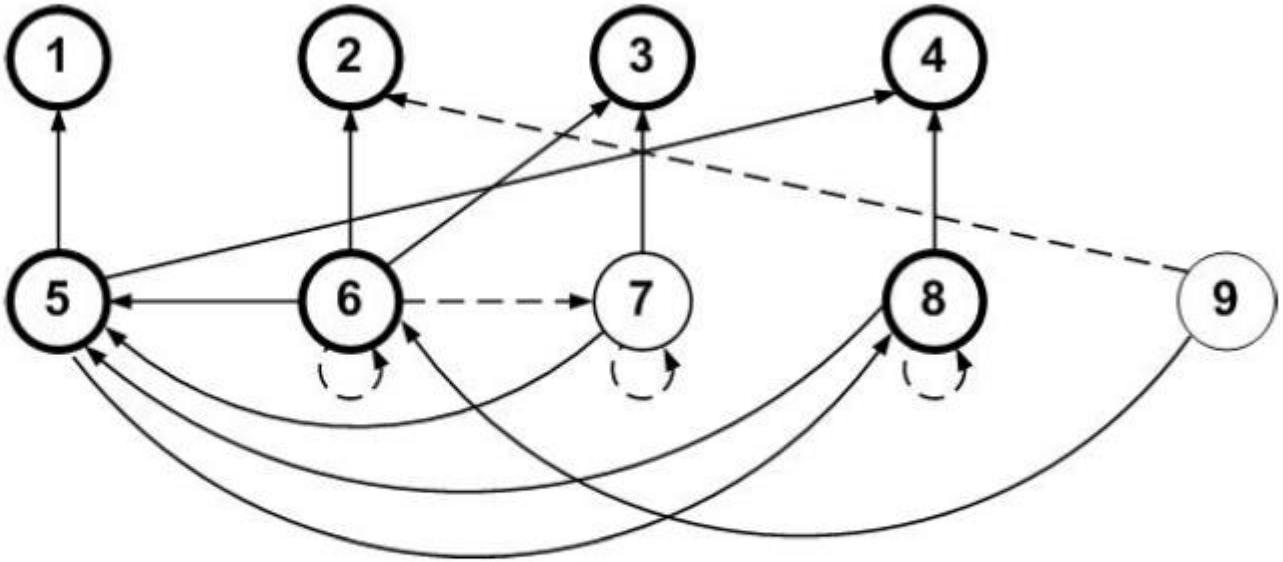
```
        begin
S1:     read(N);
S2:     Z := 0;
S3:     Y := 0;
S4:     I := 1;
S5:     while (I <= N)
        do
S6:         Z := f1(Z, Y);
S7:         Y := f2(Y);
S8:         I := I + 1;
        end-while;
S9:     write(Z)
        end.
```



- As statements are executed, edges corresponding to the new dependencies that occur are changed to solid lines.

Example 2 – Dynamic Slicing, Approach 2

```
      begin
S1:     read(N);
S2:     Z := 0;
S3:     Y := 0;
S4:     I := 1;
S5:     while (I <= N)
        do
S6:         Z := f1(Z, Y);
S7:         Y := f2(Y);
S8:         I := I + 1;
        end-while;
S9:     write(Z)
      end.
```



- The graph is traversed only along solid edges and the nodes reached are made bold.
- The dynamic slice is {1, 2, 3, 4, 5, 6, 8}.

Weakness of Approach 2

- The problem with Approach 2 lies in the fact that,
  - In the presence of loops, the slice may sometimes include more statements than necessary.

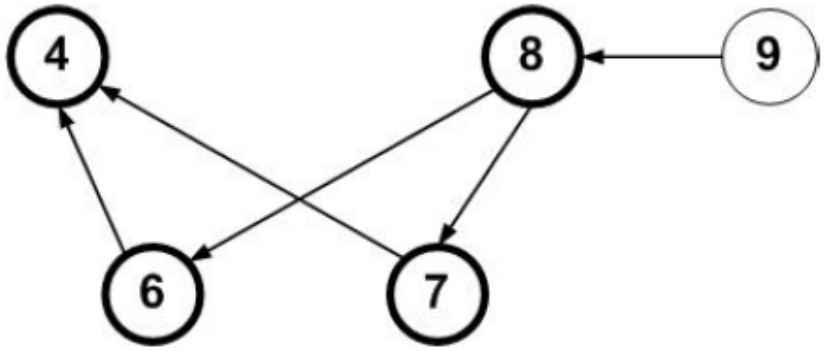
Example 3 – Dynamic Slicing, Approach 2

```
      begin
S1:     read(N);
S2:     I := 1;
S3:     while (I <= N)
        do
S4:         read(X);
S5:         if (X < 0)
            then
S6:             Y := f1(X);
            else
S7:             Y := f2(X);
            end_if;
S8:         Z := f3(Y);
S9:         WRITE(Z);
S10:        I := I + 1;
        end_while;
      end.
```

What's the dynamic slice with respect to variable Z at the end of the execution, for test case (N = 2, X = -4, 3)?

Example 3 – Dynamic Slicing, Approach 2(cont.)

```
      begin
S1:       read(N);
S2:       I := 1;
S3:       while (I <= N)
          do
S4:           read(X);
S5:           if (X < 0)
              then
S6:               Y := f1(X);
              else
S7:               Y := f2(X);
              end_if;
S8:           Z := f3(Y);
S9:           WRITE(Z);
S10:          I := I + 1;
          end_while;
      end.
```



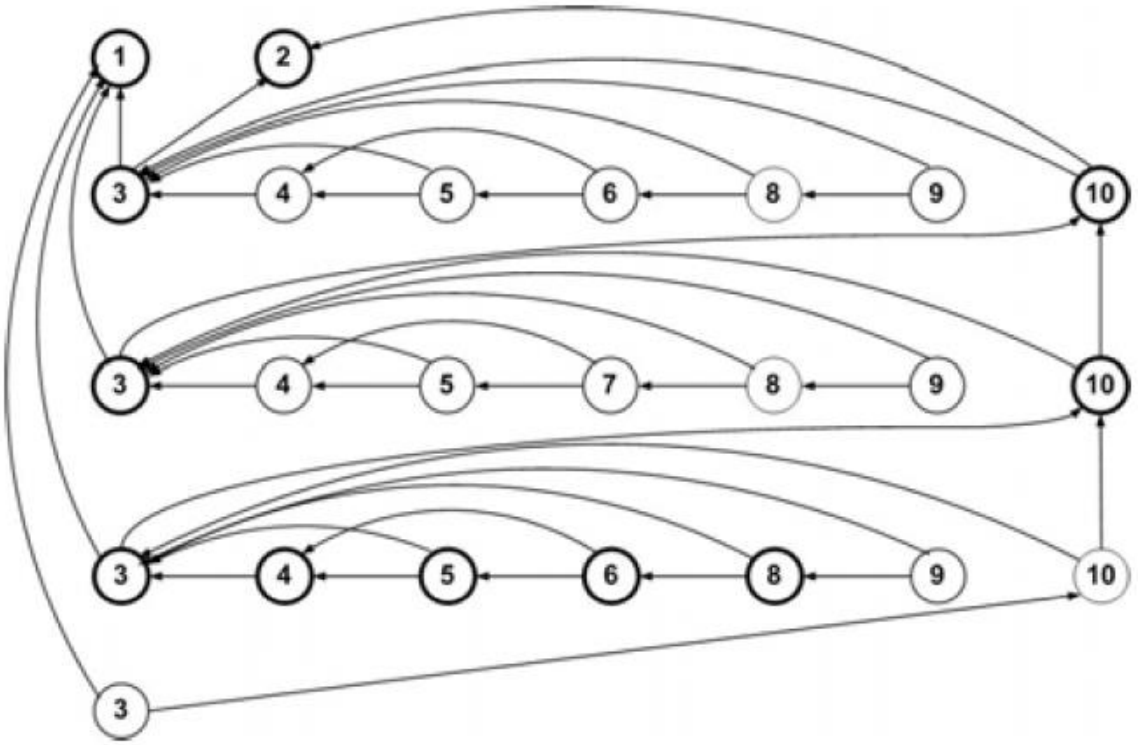
- The data dependence edge from 8 to 6 is marked during the first iteration.
- The data dependence edge from 8 to 7 is marked during the second iteration.
- The value of Z observed at the end of the second iteration is only affected by statement 7.

Dynamic Slicing: Approach 3

- *Approach 3*
  - Create a separate node for each occurrence of a statement in the execution history, with outgoing dependence edges to only those statements (their specific occurrences) on which this statement occurrence is dependent.
  - The resulting graph is called *Dynamic Dependence Graph*.

Example 3 – Dynamic Slicing, Approach 3

```
      begin
S1:     read(N);
S2:     I := 1;
S3:     while (I <= N)
        do
S4:       read(X);
S5:       if (X < 0)
          then
S6:         Y := f1(X);
          else
S7:         Y := f2(X);
          end_if;
S8:       Z := f3(Y);
S9:       WRITE(Z);
S10:      I := I + 1;
        end_while;
      end.
```



The dynamic slice with respect to variable Z at the end of the execution, for test case (N = 3, X = -4, 3, -2).

Weakness of Approach 3

- The problem with Approach 3 lies in the fact that,
  - The size of a Dynamic Dependence Graph (total number of nodes and edges) is, in general, *unbounded*.
  - The number of nodes in the graph is equal to the number of statements in the execution history, which, in general, may depend on values of run-time inputs.

Dynamic Slicing: Approach 4

- *Approach 4*
  - Instead of creating a new node for every occurrence of a statement in the execution history, create a new node only if another node with the same transitive dependencies does not already exist.
  - The resulting graph is called *Reduced Dynamic Dependence Graph*.

Rules Employed in Approach 4

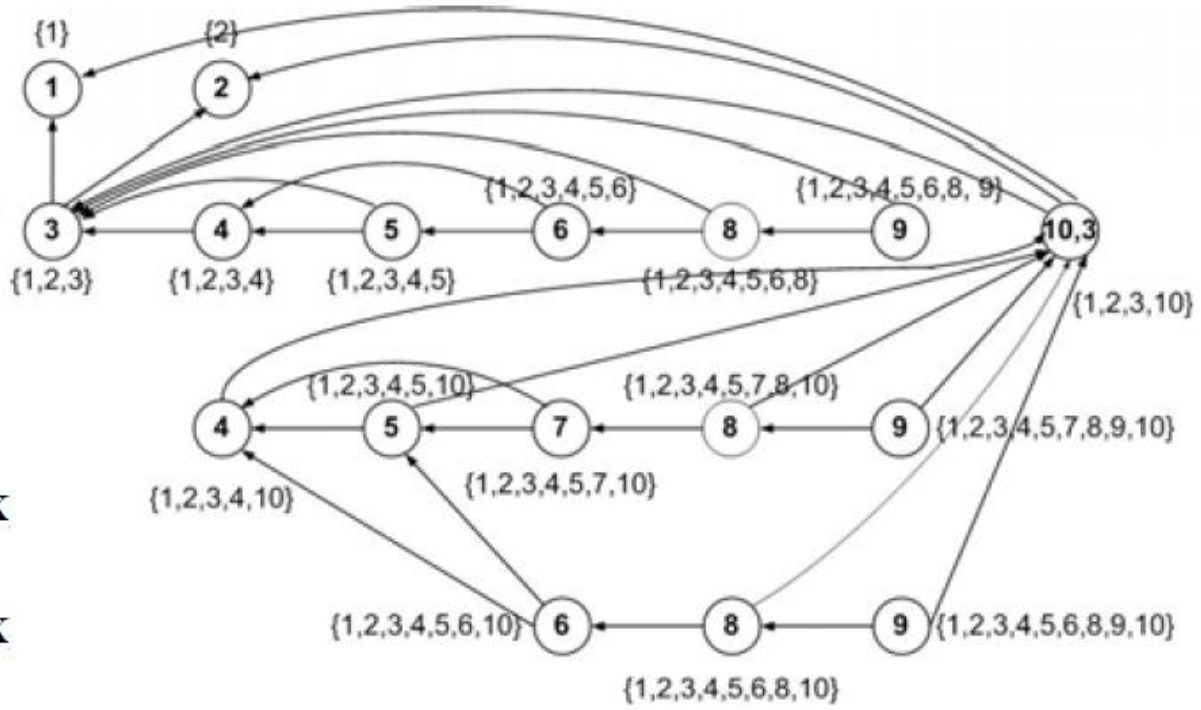
- Every time a statement, Si, gets executed, we determine the set of nodes, D, that last assigned values to the variables used by S_i, and the last occurrence, C, of the control predicate node of the statement.

- If a node, n associated with S_i already exists whose immediate descendants are the same as *DUC*, we associate the new occurrence of S_i with n. Otherwise, we create a new node with outgoing edges to all nodes in *DUC* .

- Whenever we need to create a new node, say for statement S_i, we first determine if any of its immediate descendants say node v already has a dependency on a previous occurrence of S_i and if the other immediate descendants of the occurrences of S_i are also reachable from v.
  - If so, we can merge the new occurrence of S_i with v.

Example 3 – Dynamic Slicing, Approach 4

```
      begin
S1:       read(N);
S2:       I := 1;
S3:       while (I <= N)
          do
S4:           read(X);
S5:           if (X < 0)
              then
S6:               Y := f1(X
              else
S7:               Y := f2(X
              end_if;
S8:           Z := f3(Y);
S9:           WRITE(Z);
S10:          I := I + 1;
          end_while;
      end.
```



References

- **[Wei81]** Mark Weiser, *Program Slicing*. Proceedings of the 5th International Conference on Software Engineering, pp. 439-449, 1981.

- **[HRB90]** Susan Horwitz, Thomas Reps, and Binkley, *Interprocedural Slicing Using Dependence Graphs*. ACM Transactions on Programming Languages and Systems, Vol. 12, No. 1, pp. 26-60, 1990.

- **[AH90]** Hiralal Agrawal and Joseph R. Horgan, *Dynamic Program Slicing*. Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation 1990 implementation, 1990.