Static Program Analysis

Automated Static Analysis

- A static analyzer is a software tool for source code text processing
- They parse the program text and try to discover potentially erroneous conditions and bring these to the attention of the V&V/Testing team
- Very effective as an aid to inspections.
- A supplement to but not a replacement for inspections

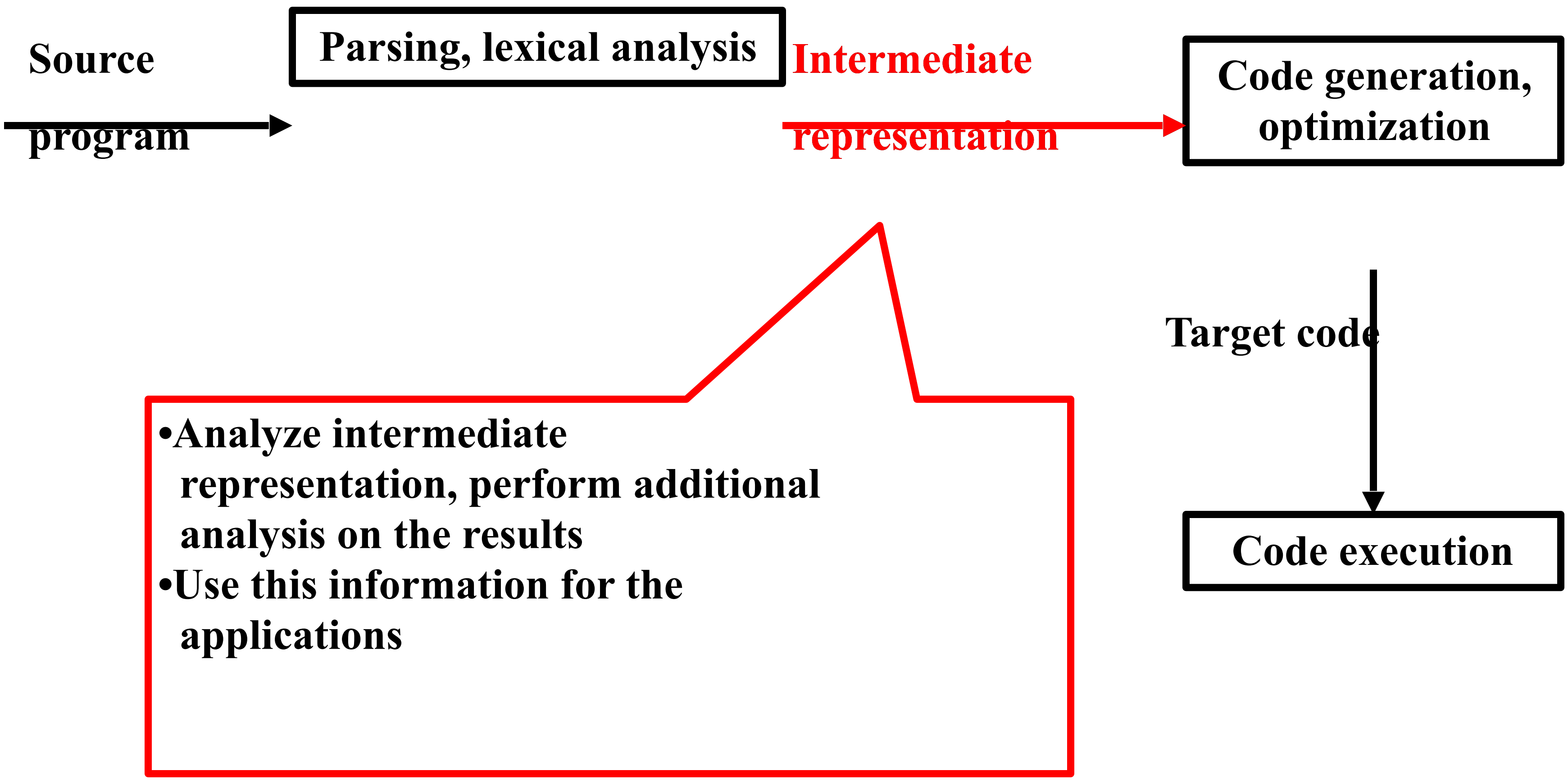
Types of Static Analysis Checks

Fault Type	Static Analysis Check
Data	• Variables used before initialization • Variables declared but never used • Variables assigned twice but never used between assignments • Possible array bound violations • Undeclared variables
Control	• Unreachable code • Unconditional branches into loops
I/O	• Variables output twice with no intervening assignment
Interface	• Parameter type mismatches • Parameter number mismatches • Non-usage of results of functions • Uncalled functions
Storage management	• Unassigned pointers • Pointer arithmetic

Static Models of the Source Code

- Low level
  - Source code text
- Intermediate level
  - Symbol table
  - Parse tree
- High level
  - Control flow
  - Data flow
  - Program Dependency Graph
- Design Level
  - Class diagram
  - Sequence diagram

Starting Point for Static Analysis



Intermediate Representation

- Parse (derivation) Tree & Symbol Table
- Concrete Parse Tree
  - Concrete (derivation) tree shows structure *and* is language-specific issues
  - Parse tree represents concrete syntax
- Abstract Syntax Tree/Graph (AST)/(ASG)
  - Abstract Syntax Tree shows only structure
  - Represents abstract syntax

AST vs Parse Tree

Example

1. a := b + c

2. a = b + c;

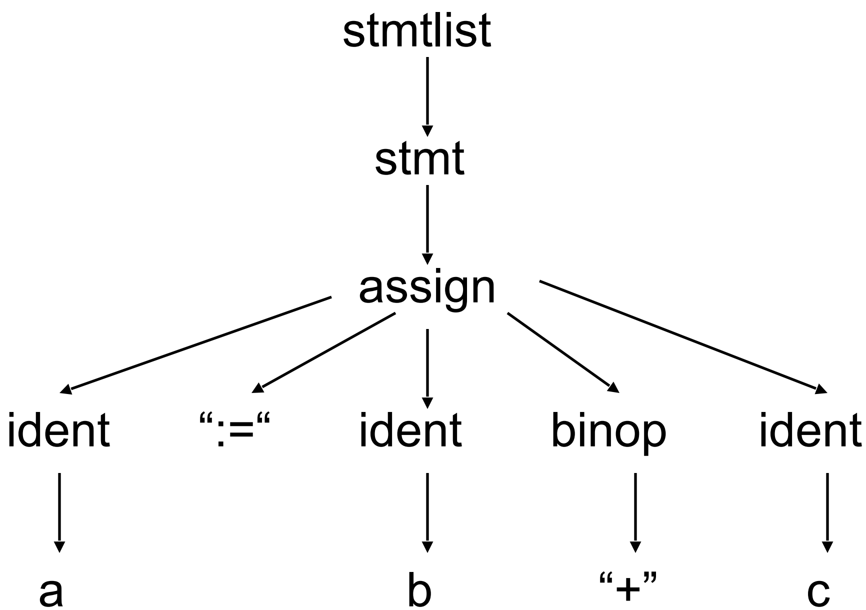
- Grammar for 1
  - stmtlist → stmt | stmt stmtlist
  - stmt → assign | if-then | …
  - assign → ident ":=" ident binop ident
  - binop → "+" | "-" | …
- Grammar for 2
  - stmtlist → stmt ";" | stmt";" stmtlist
  - stmt → assign | if-then | …
  - assign → ident "=" ident binop ident
  - binop → "+" | "-" | …

Parse Trees

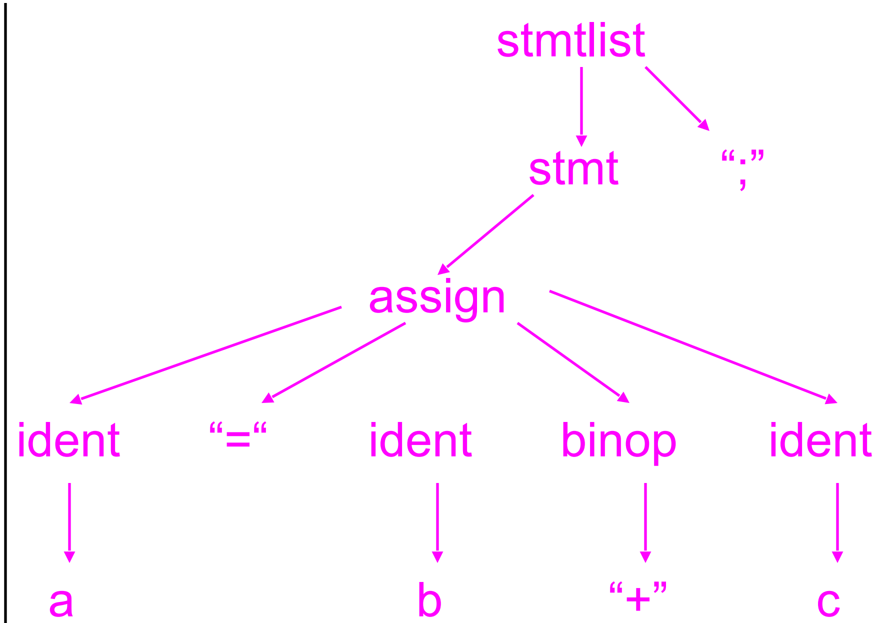
Example

1. a := b + c
2. a = b + c;

Parse Tree for 1



Parse Tree for 2

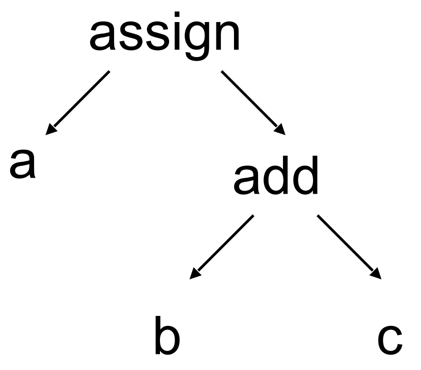


AST

Example

1. a := b + c
2. a = b + c;

Abstract syntax tree for 1 and 2

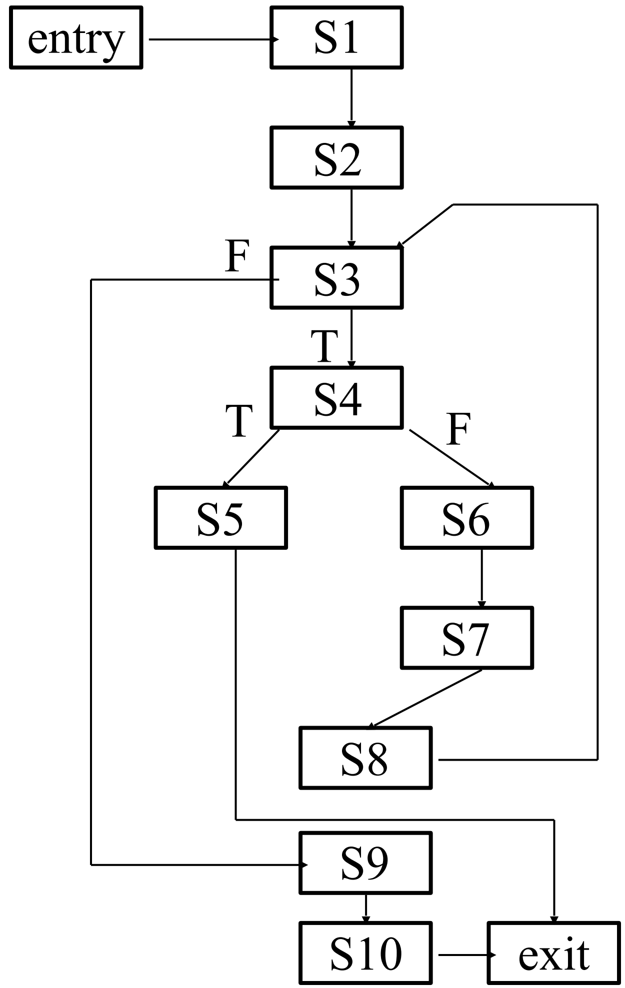


Intermediate to High level

- Given
  - Source code
  - AST
  - Symbol table
- One can construct
  - Call graphs
  - Control flow graph
  - Data flow
  - Slices

Control Flow Analysis (CF)

```
Procedure AVG
S1   count = 0
S2   fread(fptr, n)
S3   while (not EOF) do
S4      if (n < 0)
S5         return (error)
        else
S6         nums[count] = n
S7         count ++
        endif
S8      fread(fptr, n)
       endwhile
S9   avg = mean(nums,count)
S10  return(avg)
```

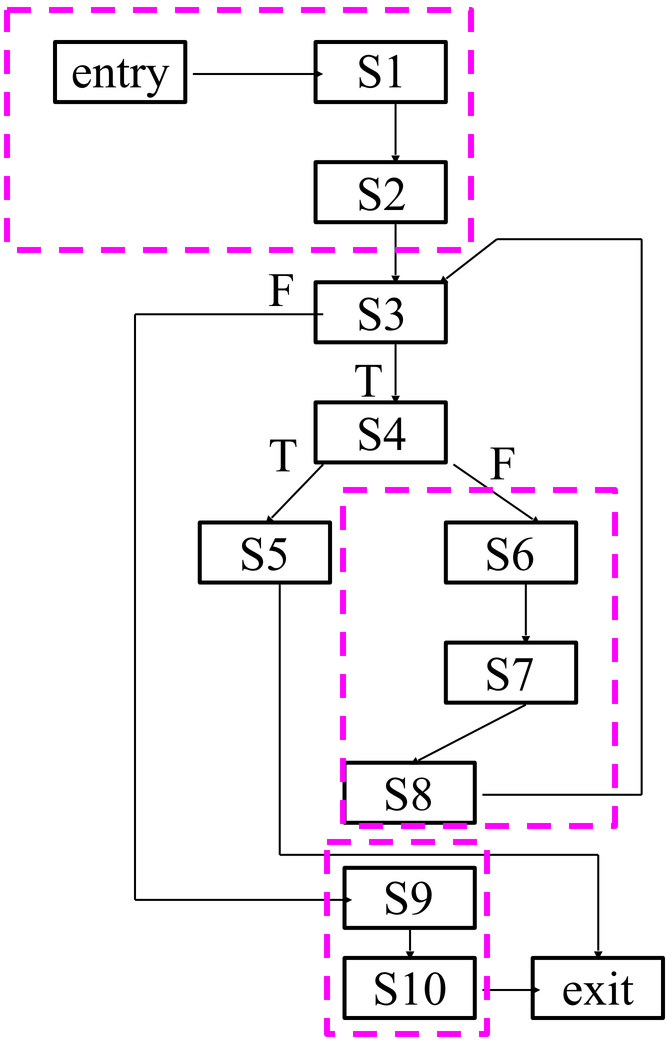


Computing Control Flow

- Basic blocks can be identified in the AST
- Basic blocks are straight line sequence of statements with no branches in or out.
- A basic block may or may not be "maximal"
- For compiler optimizations, maximal basic blocks are desirable
- For software engineering tasks, basic blocks that represent one source code statement are often used

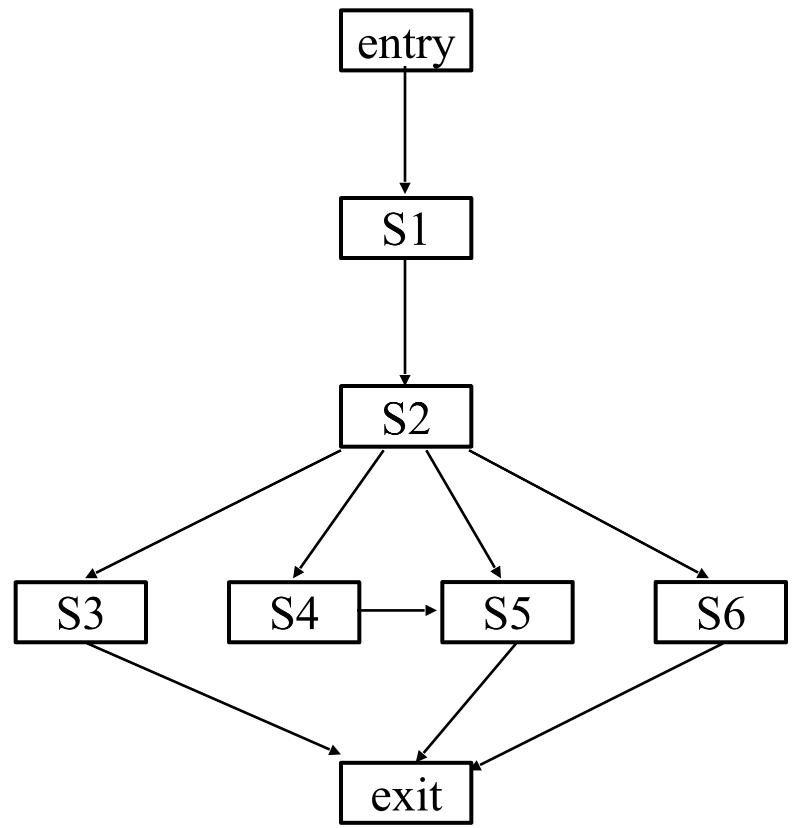
Computing Control Flow

```
Procedure AVG
S1    count = 0
S2    fread(fptr, n)
S3    while (not EOF) do
S4       if (n < 0)
S5          return (error)
         else
S6          nums[count] = n
S7          count ++
         endif
S8       fread(fptr, n)
        endwhile
S9    avg = mean(nums,count)
S10   return(avg)
```



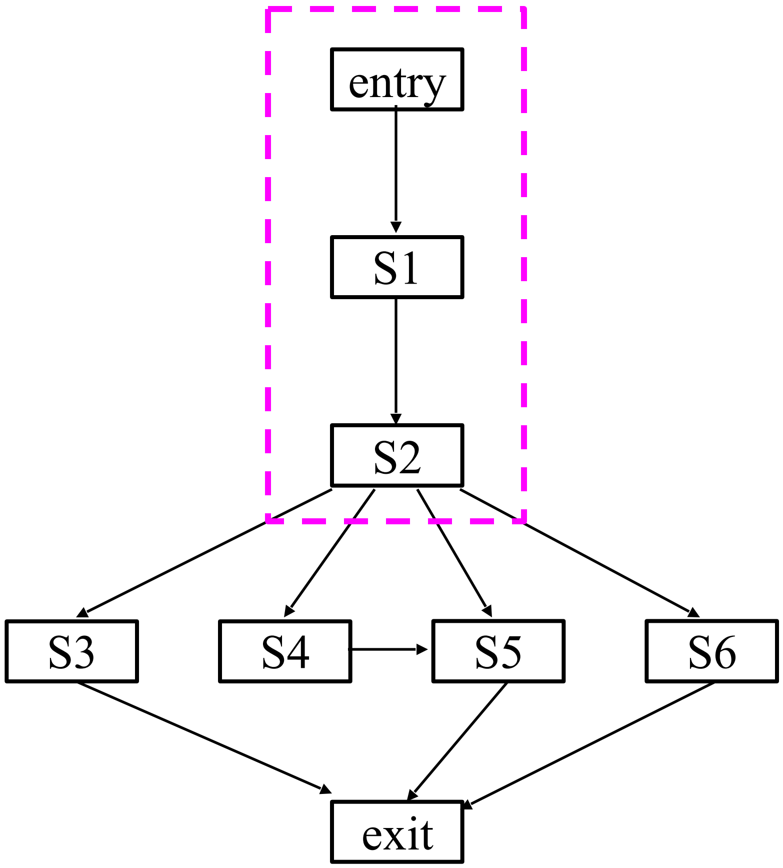
Computing Control Flow

```
Procedure Trivial
S1   read (n)
S2    switch (n)
        case 1:
S3        write ("one")
          break
        case 2:
S4        write ("two")
        case 3:
S5        write ("three")
          break
        default
S6        write ("Other")
       endswitch
end Trivial
```



Computing Control Flow

```
Procedure Trivial
S1    read (n)
S2    switch (n)
        case 1:
S3        write ("one")
          break
        case 2:
S4        write ("two")
        case 3:
S5        write ("three")
          break
        default
S6        write ("Other")
      endswitch
end Trivial
```



Control Flow Graph

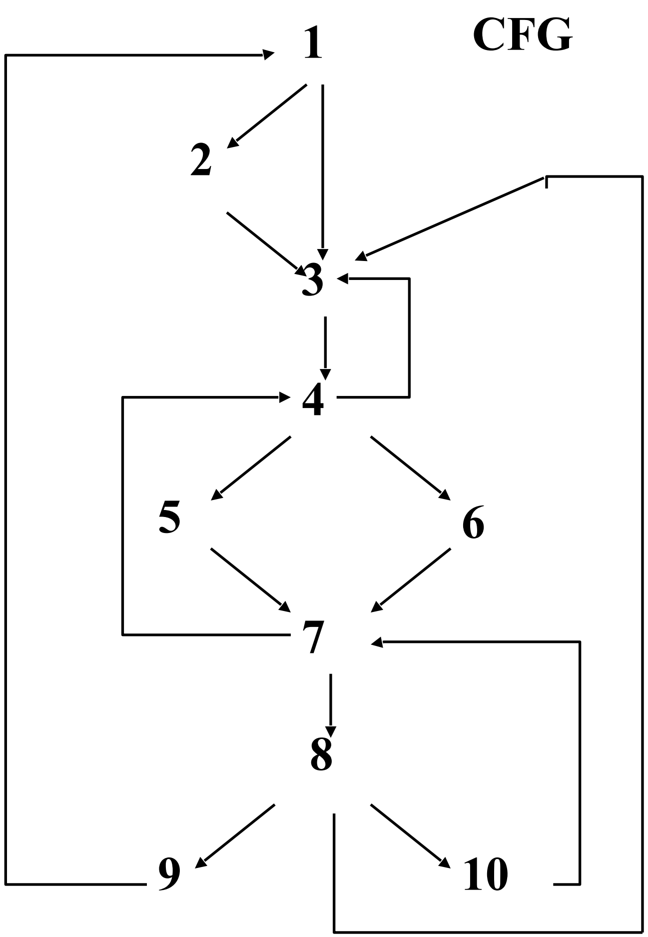
- A control flow graph $CFG = (N, E)$ is a directed graph
- $N = \{n_1, n_2, \ldots n_k\}$ is a finite set of nodes (basic blocks of a program)
- $E = \{(n_i, n_j) \mid n_i, n_j\ N\ \&$ *the flow of control goes from* n_i *to* $n_j\}$

Dominators

- Given a Control Flow Graph (CFG) with nodes D and N:
  - D dominates N if every path from the initial node to N goes through D

- Properties of dominance:
  1. Every node dominates itself
  2. Initial node dominates all others

Dominators - example

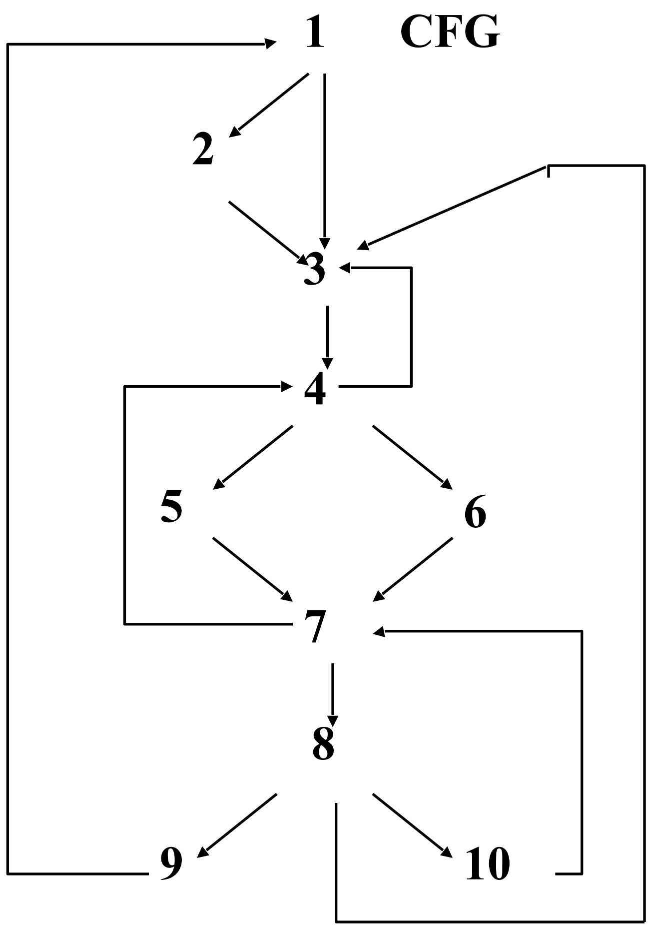


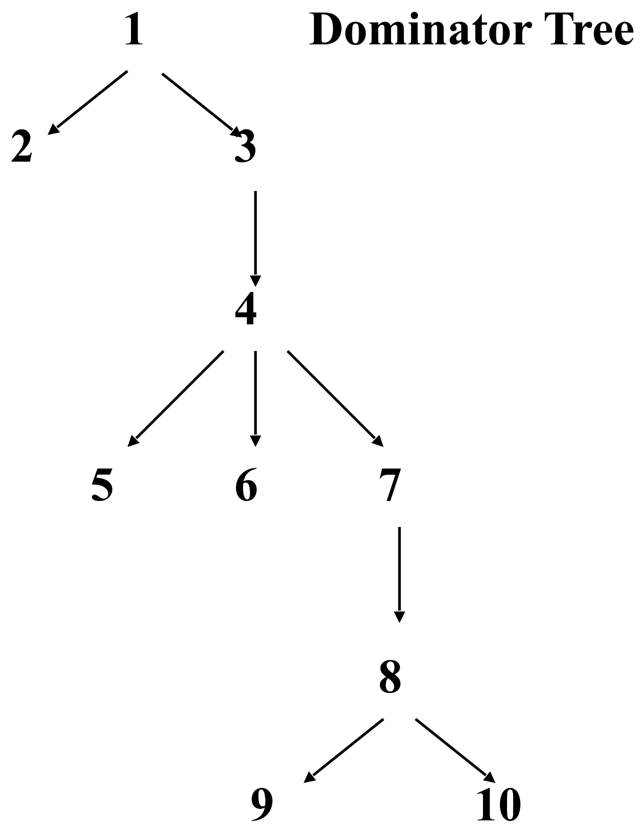
Node	Dominates
1	1,2,…,10
2	2
3	3,4,5,6,7,8,9,10
4	4,5,6,7,8,9,10
5	5
6	6
7	7,8,9,10
8	8,9,10
9	9
10	10

Dominator Trees

- In a dominator tree
  - The initial node *n* is the root of the Control Flow Graph
  - The parent of a node n is its *immediate dominator* (i.e., the last dominator of *n* on any path); the immediate dominator for *n* is unique

Dominators - dominator tree example



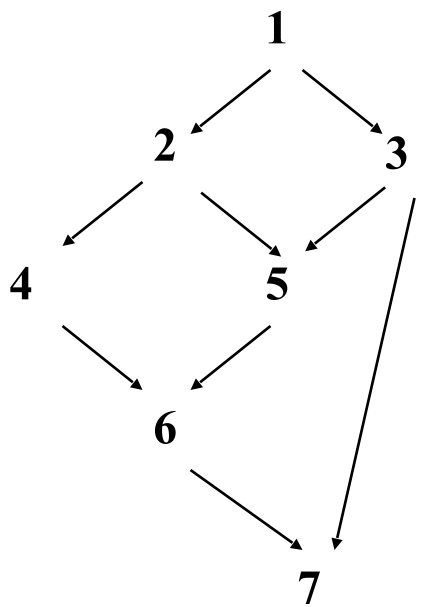


Post-Dominators

- Given a Control Flow Graph with nodes PD and N:
  - PD post dominates N if every path from N to the final nodes goes through PD

Post-Dominators - Example

CFG



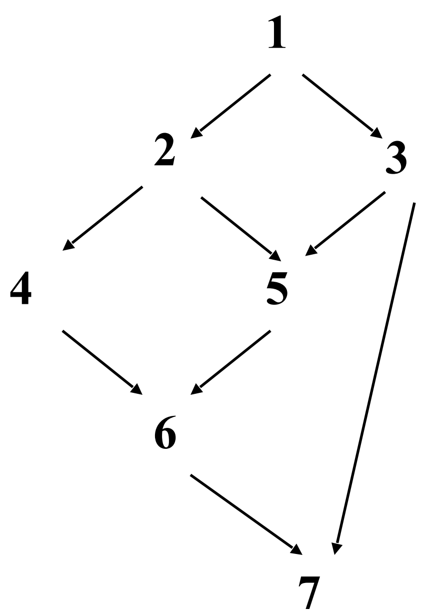
Node	Postdominates
1	--
2	--
3	--
4	--
5	--
6	2,4,5
7	1,2,3,4,5,6

Post Dominators - Dominator Tree

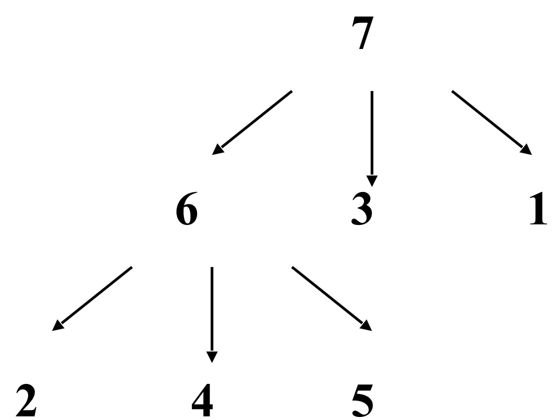
- In a post dominator tree
  - The initial node *n* is the exit node of the Control Flow Graph
  - The parent of a node n is its *immediate post dominator* (i.e., the first post dominator of *n* on any path); the immediate post dominator for *n* is unique

Post Dominators - Dominator Tree Example

CFG

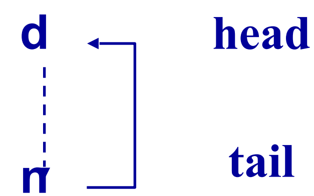


Post dominator Tree



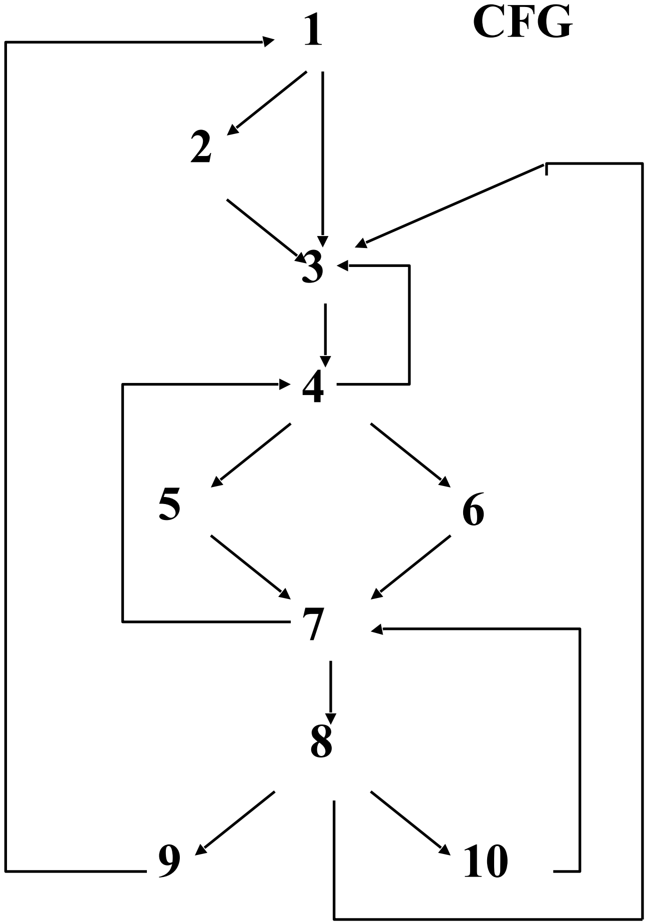
Finding Loops

- We'll consider what are known as natural loops
  - Single entry node (header) that *dominates* all other nodes in the loop
  - The nodes in the loop form a strongly connected component, that is, from every node there is at least one path back to the header
  - There is a way to iterate - there is a back edge (n,d) whose target node d (called the head) dominates its source node n (called the tail)



- If two back edges have the same target, then all nodes in the loop sets for these edges are in the same loop

Loops - Example



CFG

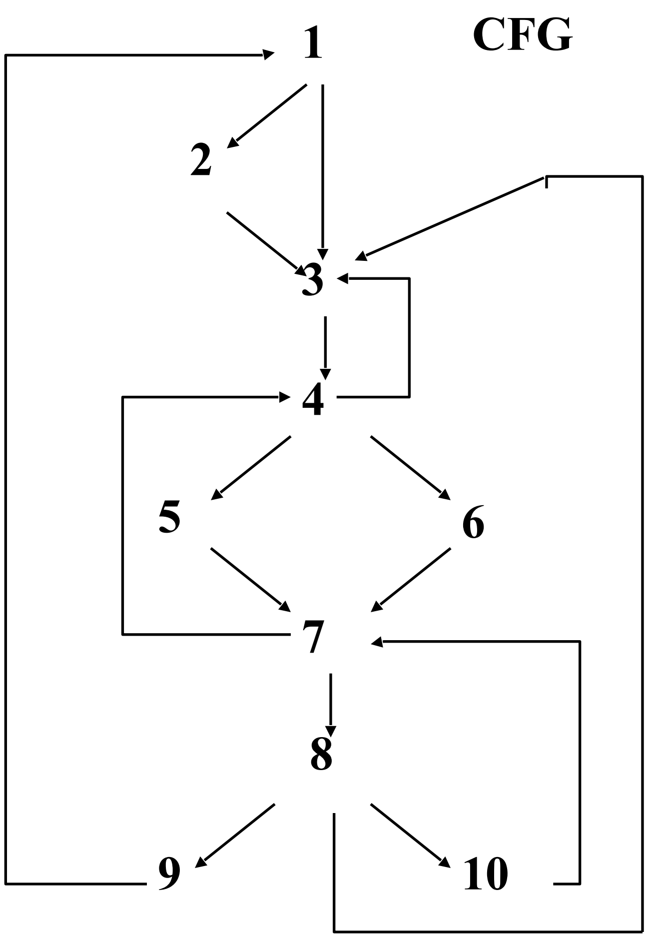
Which edges are back edges?

$4 \rightarrow 3$
$7 \rightarrow 4$
$10 \rightarrow 7$
$9 \rightarrow 1$
$8 \rightarrow 3$

Construction of loops

1. Find dominators in Control Flow Graph
2. Find back edges
3. Traverse back edge in reverse execution direction until the target of the back edge is reached; all nodes encountered during this traversal form the loop. The result is all nodes that can reach the source of the edge without going through the target

Loops - Example



Back Edge	Loop Induced
4 → 3	{3,4,5,6,7,8,10}
7 → 4	{4,5,6,7,8,10}
10 → 7	{7,8,10}
8 → 3	{3,4,5,6,7,8,10}
9 → 1	{1,2,…,10}

Applications of Control Flow

- Complexity
  - Cyclomatic (McCabe's) - Indication of number of test case needed; indication of difficulty of maintaining
- Testing
  - branch, path, basis path
- Program understanding
  - program structure and flow is explicit

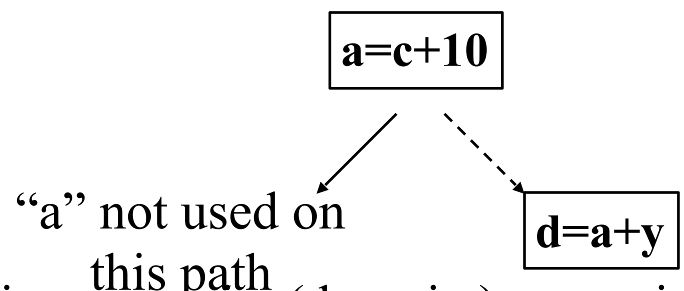
Data Flow Analysis

- Data-flow analysis provides information for compiling and SE tasks by computing the flow of different types of data to points in the program
- For structured programs, data-flow analysis can be performed on an AST
- In general, intra-procedural (global) data-flow analysis is performed on the Control Flow Graph
- Exact solutions to most problems are undecidable
  - May depend on input
  - May depend on outcome of a conditional statement
  - May depend on termination of loop
- We compute approximations of the exact solution

Applications of Data Flow Analysis

Software Engineering Tasks

- *Data-flow testing*
  - suppose that a statement assigns a value but the use of that value is never executed under test



  - need <u>definition-use pairs (du-pairs):</u> associations between definitions and uses of the same variable or memory location

Applications of Data Flow Analysis

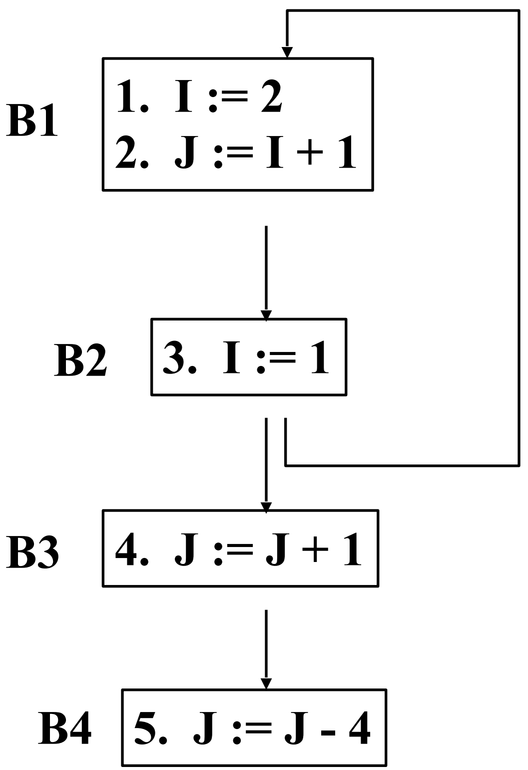
Software Engineering Tasks

- *Debugging*
  - suppose that **a** has the incorrect value in the statement

a=c+y

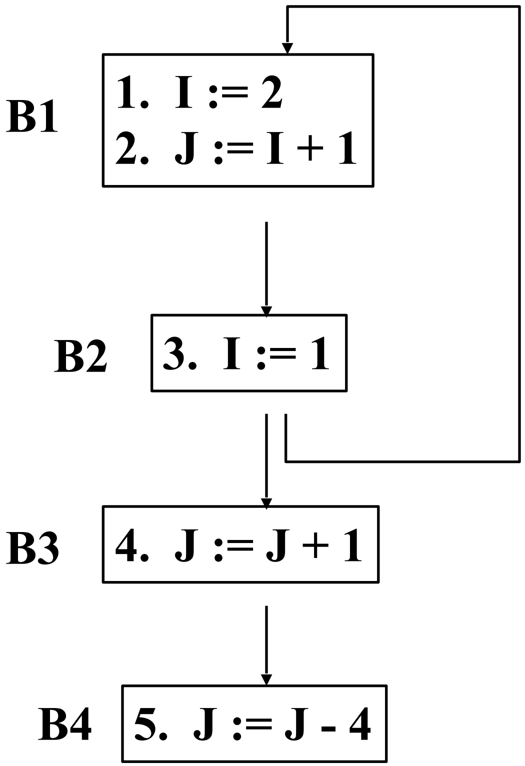
  - need data dependence information: statements that can affect the incorrect value at this point

Data Flow Problems – Reaching Definitions



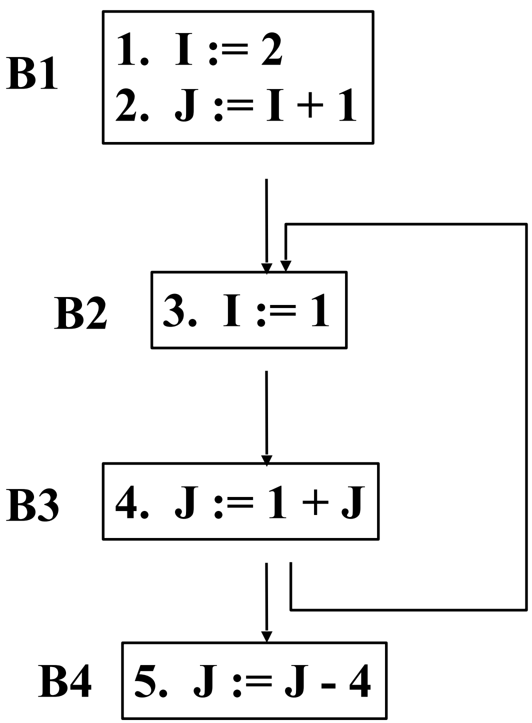
- Compute the flow of data to points in the program - e.g.,
  - Where does the assignment to I in statement 1 reach?
  - Where does the expression computed in statement 2 reach?
  - Which uses of variable J are reachable from the end of B1?
  - Is the value of variable I live after statement 3?
- Interesting points before and after basic blocks or statements

Data Flow Problems – Reaching Definitions



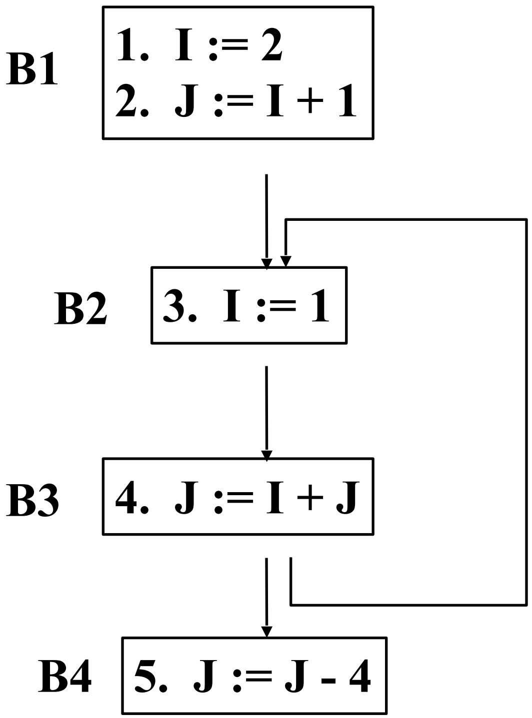
- A *definition* of a variable or memory location is a point or statement where that variable gets a value - e.g., input statement, assignment statement.
- X *reaches* a point P if there exists a control-flow path in the CFG from the definition to P with no other definitions of X on the path (called a *definition-clear path*)
- Such a path may exist in the graph but may not be executable (i.e., there may be no input to the program that will cause it to be executed); such a path is *infeasible*.

Data Flow Problems – Reachable Uses



- A *use* of a variable or memory location is a point or statement where that variable is referenced but not changed - e.g., used in a computation, used in a conditional, output
- Use of X is *reachable* from a point P if there exists a control-flow path in the CFG from the P to the use with no definitions of X on the path
- Reachable uses also called *upwards exposed uses*

Data Flow Problems – Reachable Uses



- Definitions:
  - I: 1, 3
  - J: 2, 4, 5
- Uses:
  - I: 2, 4
  - J: 4, 5
- Reachable Uses:
  - I from 1: 2
  - I from 3: 4
  - J from 2: 4
  - J from 4: 4, 5
  - J from 5:

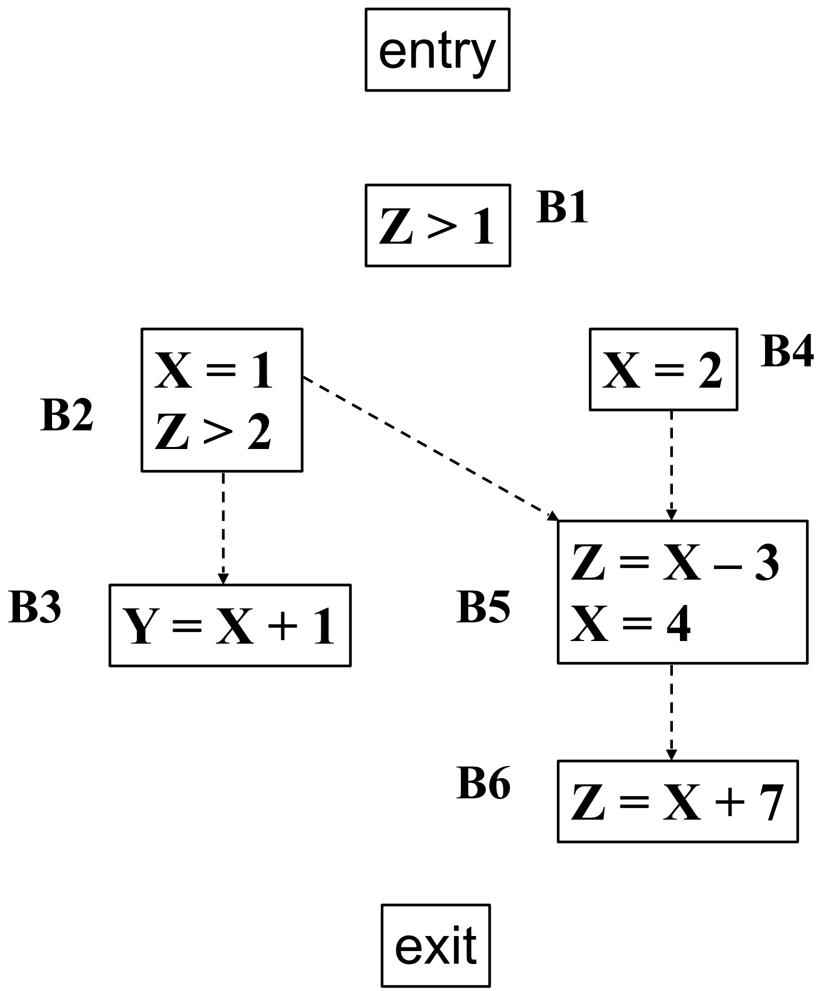
DU-Chains, UD-chains, Webs

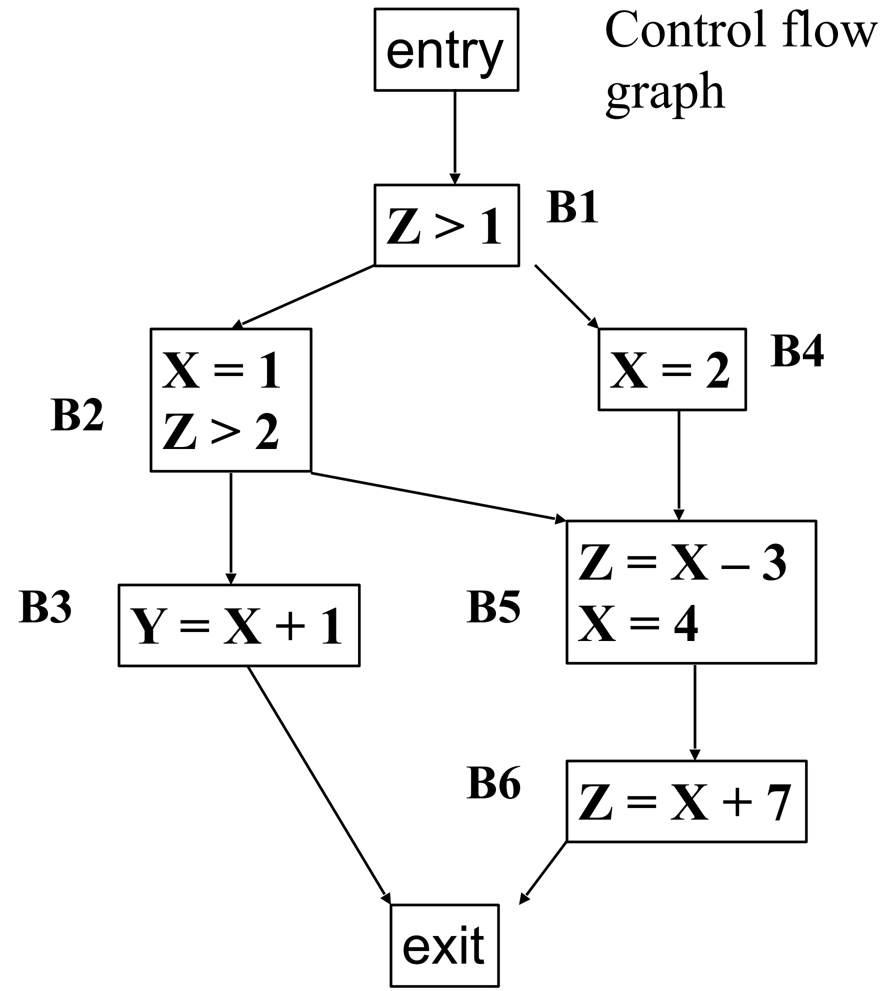
- A *definition-use chain* or DU-chain for a definition D of variable V connects the D to all uses of V that it can reach
- A *use-definition chain* or UD-chain for a use U of variable V connects U to all definitions of V that reach it
- A *web* for a variable is the maximal union of intersecting DU-chains

Data-Dependence

- A *data-dependence graph* has one node for every basic block and one edge representing the flow of data between the two nodes
- X is *data dependent* on Y iff there exists a variable **v** such that:
  - Y has a definition of **v** and
  - X has a use of **v** and
  - There exists a control path from Y to X along which **v** is not redefined
- Different types of data dependence edges can be defined
  - Flow: def to use (most common)
  - Anti: use to def
  - Out: def to def

Data (flow) Dependence Graph





Control Dependence

- A statement S1 is *control dependent* on a statement S2 if the outcome of S2 determines whether S1 is reached in the CFG
- We define control dependence for language constructs
- Control dependencies can be derived for arbitrary control flow using the concept of post dominator of **conditional** instructions

Definitions

`if Y then B1 else B2;`

- X is control dependent on Y iff X is in B1 or B2

`while Y do B;`

- X is control dependent on Y iff X is in B

Program-Dependence Graph

- A *program dependence graph* (PDG) for a program P is the combination of the control-dependence graph for P and the data-dependence graph for P
- Can be used for
  - Redundant code analysis
  - I/O relation analysis
  - Program slicing

Compute a PDG

```
read (n)
i := 1
sum := 0
product := 1
while i <= n do
     sum := sum + i
     product := product * i
     i := i + 1
write (sum)
write (product)
```

Identify control dependencies via CFG and conditionals

Identify data dependencies via definition/uses

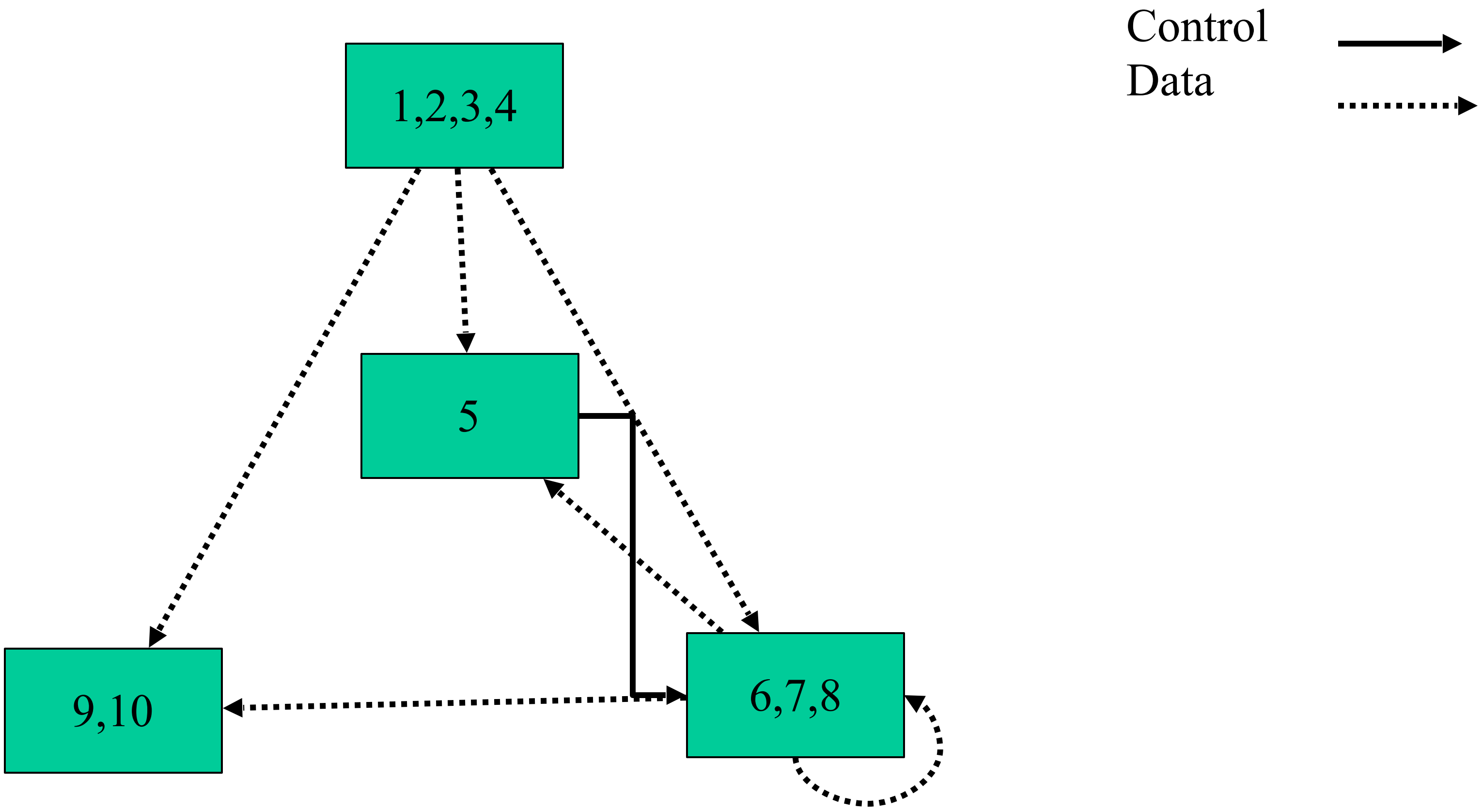
Computing a PDG

```
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)
```

6,7,8 are control dependent on 5

DU-Chains:
(1,5)
(2,5), (2,6), (2,7), (2,8),
(8,5), (8,6), (8,7), (8,8)
(3,6), (3,9), (6,6), (6,6),
(6,9)
(4,7), (4,10), (7,7), (7,10)

PDG



Program Slicing (Weiser ‘82)

- A *program slice* consists of the parts of a program that (potentially) affect the values computed at some point of interest, referred to as a *slicing criterion*
- Typically, a slicing criterion consists of a pair (line-number; variable).
- The parts of a program which have a direct or indirect effect on the values computed at a slicing criterion C are called the program slice with respect to criterion C
- A program slice is computed from the program dependency graph
- The task of computing program slices is called program slicing

Program Slicing Research

Types of slices

- Backward static slice
- Executable slice
- Forward static slice
- Dynamic slice
- Execution slice

Levels of slices

- Intra-procedural
- Inter-procedural

1. Agrawal
2. Binkley
3. Gallagher
4. Gupta
5. Horgan
6. Horwitz
7. Korel
8. Laski
9. K. Ottenstein
10. L. Ottenstein
11. Reps
12. Soffa
13. Tip
14. Weiser

Static Backward Slicing

- 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**>

Static Backward Slicing - Example

Criterion <9, product>

```
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)
```

Static Backward Slicing - Example

Criterion <9, product>

```
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)
```

Executable Slicing

- 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 results of running the program that is the executable slice produces the same result for variables in **V** at **p** for all inputs.

Executable Slicing - Example

```
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 <9, 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)
```

Static Forward Slicing

- 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**>

Static Forward Slicing - Example

Criterion <3, sum>

```
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)
```

Static Forward Slicing - Example

Criterion <3, sum>

```
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)
```

Static Forward Slicing - Example

```
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>

Static Forward Slicing - Example

```
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>

Dynamic Slicing

- 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, v, p**>. The slicing uses the execution history or trajectory for the program with input **i.**

Dynamic Slicing - Example

```
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>

Dynamic Slicing - Example

```
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>

Dynamic Slicing - Example

```
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>

Dynamic Slicing - Example

```
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<1, 10^1, z>

Dynamic Slicing - Example

```
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<1, 10^1, z>

Dynamic Slicing - Example

```
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>

Execution Slicing

- 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.**

Execution Slicing - Example

```
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

Execution Slicing - Example

```
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