

# Control-Flow Analysis

---

Chapter 8, Section 8.4

Chapter 9, Section 9.6

# Phases of the Compilation Process

- Front end
  - Lexical analysis
  - Syntax analysis
  - Semantic analysis (e.g., type checking)
  - Generation of three-address code
- Back end
  - **Code optimization**: machine-independent optimization of three-address code (optional phase)
  - **Code generation**: target code (e.g., assembly)

# Control-Flow Graphs

- Control-flow graph (CFG) for a procedure/method
  - A node is a **basic block**: a single-entry-single-exit sequence of three-address instructions
  - An edge represents the potential flow of control from one basic block to another
- Uses of a control-flow graph
  - Inside a basic block: **local code optimizations** (e.g., Section 8.5)
  - Across basic blocks: **global code optimizations**
  - Aspects of code generation: e.g., **global register allocation**

# Control-Flow Analysis

- Part 1: Constructing a CFG
  - A very simple algorithm
- Part 2: Finding **dominators** and **post-dominators** in a CFG
- Part 3: Finding **control dependences** in a CFG
  - Necessary as part of constructing the **program dependence graph** (PDG), a popular IR for software tools for slicing, refactoring, testing, and debugging
- Part 4: Finding **loops** in a CFG
  - What exactly is a loop? We cannot simply say “whatever CFG subgraph is generated by *while*, *do-while*, and *for* statements” – need a general graph-theoretic definition

# Part 1: Constructing a CFG

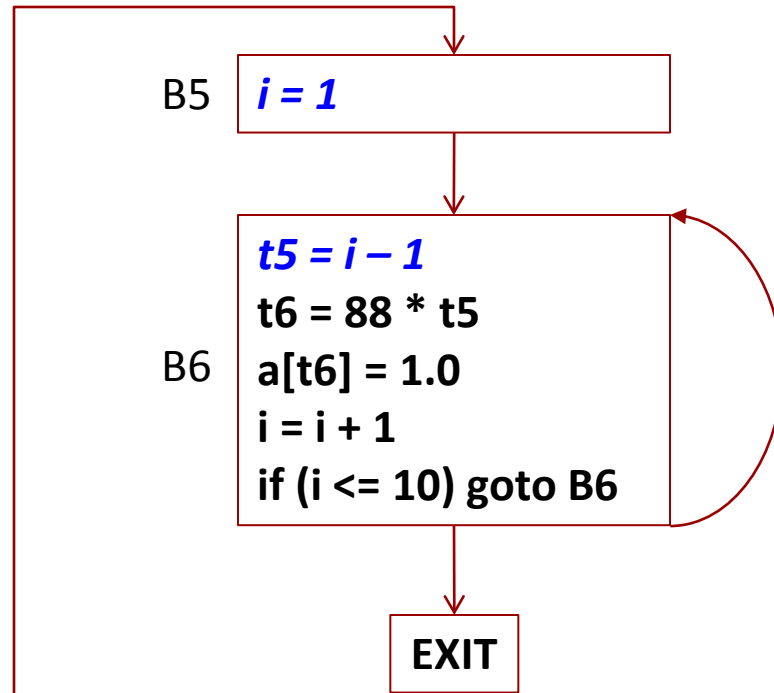
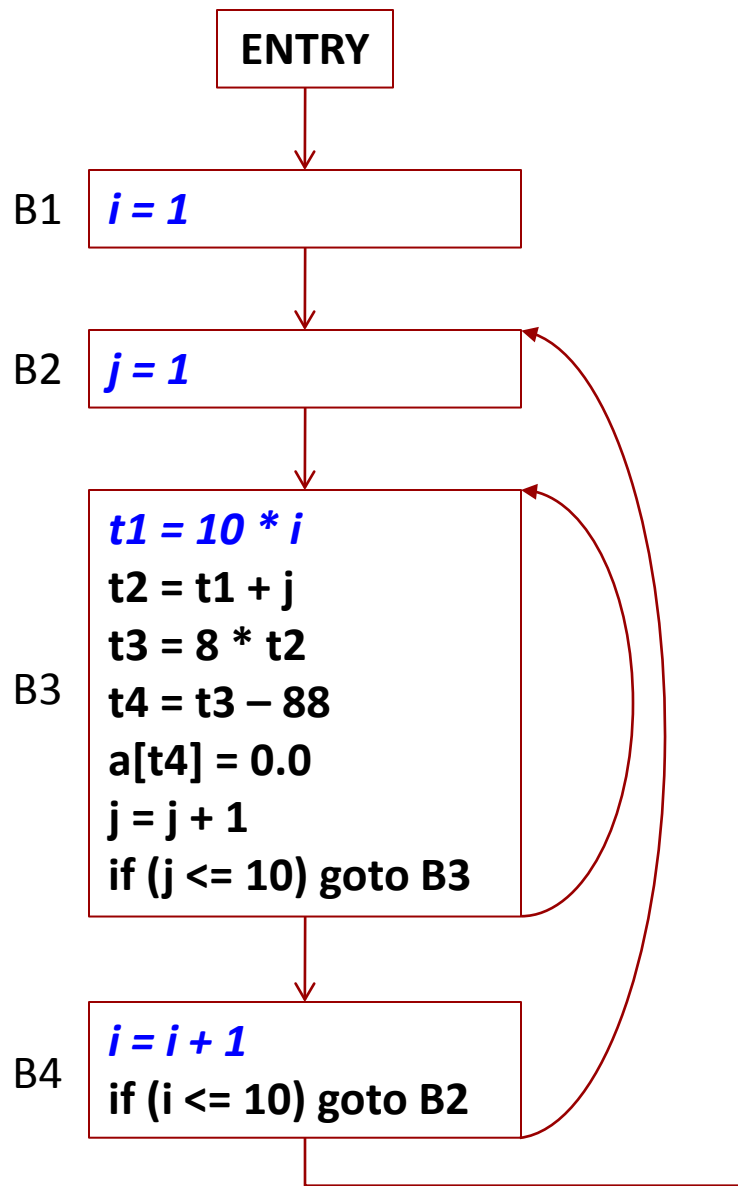
- Basic block: maximal sequence of consecutive three-address instructions such that
  - The flow of control can enter only through the first instruction (i.e., no jumps into the middle of the block)
  - The flow of control can exit only at the last instruction
- Given: the entire sequence of instructions
- First, find the **leaders** (starting instructions of all basic blocks)
  - The first instruction
  - The target of any conditional/unconditional jump
  - Any instruction that immediately follows a conditional or unconditional jump

# Constructing a CFG

- Next, find the basic blocks: for each leader, its basic block contains itself and all instructions up to (but not including) the next leader

<b>1. <math>i = 1</math></b>	→ First instruction
<b>2. <math>j = 1</math></b>	→ Target of 11
<b>3. <math>t1 = 10 * i</math></b>	→ Target of 9
<b>4. <math>t2 = t1 + j</math></b>	
<b>5. <math>t3 = 8 * t2</math></b>	
<b>6. <math>t4 = t3 - 88</math></b>	
<b>7. <math>a[t4] = 0.0</math></b>	
<b>8. <math>j = j + 1</math></b>	
<b>9. if (<math>j \leq 10</math>) goto (3)</b>	
<b>10. <math>i = i + 1</math></b>	→ Follows 9
<b>11. if (<math>i \leq 10</math>) goto (2)</b>	
<b>12. <math>i = 1</math></b>	→ Follows 11
<b>13. <math>t5 = i - 1</math></b>	→ Target of 17
<b>14. <math>t6 = 88 * t5</math></b>	
<b>15. <math>a[t6] = 1.0</math></b>	
<b>16. <math>i = i + 1</math></b>	
<b>17. if (<math>i \leq 10</math>) goto (13)</b>	

Note: this example sets array elements  $a[i][j]$  to 0.0, for  $1 \leq i, j \leq 10$  (instructions 1-11). It then sets  $a[i][i]$  to 1.0, for  $1 \leq i \leq 10$  (instructions 12-17). The array accesses in instructions 7 and 15 are done with offsets computed as described in Section 6.4.3, assuming row-major order, 8-byte array elements, and array indexing that starts from 1, not from 0.



Artificial ENTRY and EXIT nodes are often added for convenience.

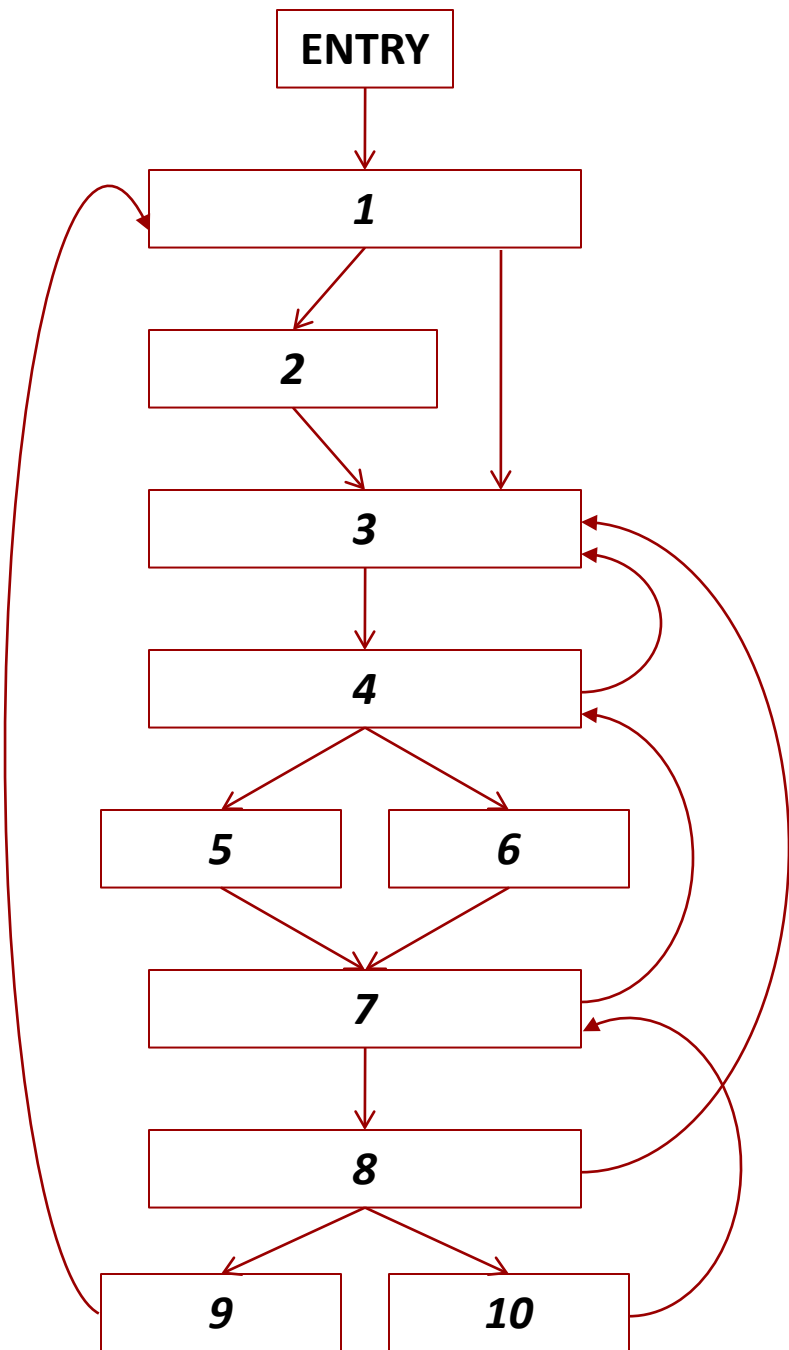
There is an edge from  $B_p$  to  $B_q$  if it is possible for the first instruction of  $B_q$  to be executed immediately after the last instruction of  $B_p$ . This is **conservative**: e.g., **if (3.14 > 2.78)** still generates two edges.

# Practical Considerations

- The usual data structures for graphs can be used
  - The graphs are sparse (i.e., have relatively few edges), so an **adjacency list** representation is the usual choice
    - Number of edges is at most  $2 * \text{number of nodes}$
- Nodes are basic blocks; edges are between basic blocks, not between instructions
  - Inside each node, some additional data structures for the sequence of instructions in the block (e.g., a linked list of instructions)
  - Often convenient to maintain both a list of **successors** (i.e., outgoing edges) and a list of **predecessors** (i.e., incoming edges) for each basic block

## Part 2: Dominance

- A CFG node  $d$  **dominates** another node  $n$  if every path from ENTRY to  $n$  goes through  $d$ 
  - Implicit assumption: every node is reachable from ENTRY (i.e., there is no dead code)
  - A dominance relation  $dom \subseteq \text{Nodes} \times \text{Nodes}$ :  $d \text{ dom } n$
  - The relation is trivially reflexive:  $d \text{ dom } d$
- Node  $m$  is the **immediate dominator** of  $n$  if
  - $m \neq n$
  - $m \text{ dom } n$
  - For any  $d \neq n$  such  $d \text{ dom } n$ , we have  $d \text{ dom } m$
- Every node has a unique immediate dominator
  - Except ENTRY, which is dominated only by itself



This example is artificial: it does not have an EXIT node; nodes 4 and 8 have more than 2 outgoing edges

ENTRY *dom*  $n$  for any  $n$

1 *dom*  $n$  for any  $n$  except ENTRY

2 does not dominate any other node

3 *dom* 3, 4, 5, 6, 7, 8, 9, 10

4 *dom* 4, 5, 6, 7, 8, 9, 10

5 does not dominate any other node

6 does not dominate any other node

7 *dom* 7, 8, 9, 10

8 *dom* 8, 9, 10

9 does not dominate any other node

10 does not dominate any other node

Immediate dominators:

1  $\rightarrow$  ENTRY

2  $\rightarrow$  1

3  $\rightarrow$  1

4  $\rightarrow$  3

5  $\rightarrow$  4

6  $\rightarrow$  4

7  $\rightarrow$  4

8  $\rightarrow$  7

9  $\rightarrow$  8

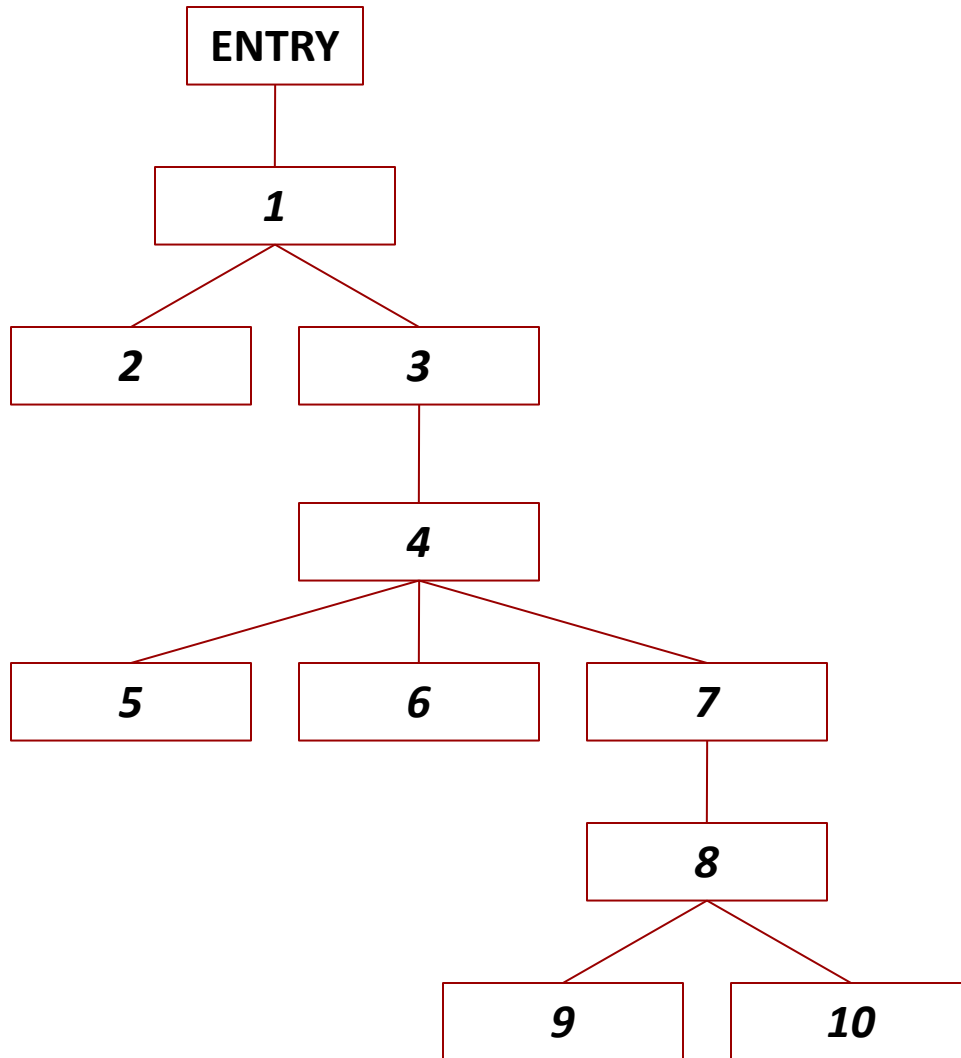
10  $\rightarrow$  8

## A Few Observations

- For any acyclic path from ENTRY to  $n$ , all dominators of  $n$  appear along the path, always in the same order (for all such paths)
- Dominance is a **transitive** relation:  $a \text{ dom } b$  and  $b \text{ dom } c$  means  $a \text{ dom } c$
- Dominance is an **anti-symmetric** relation:  $a \text{ dom } b$  and  $b \text{ dom } a$  means that  $a$  and  $b$  must be the same
  - Reflexive, anti-symmetric, transitive: **partial order**
- If  $a$  and  $b$  are two dominators of some  $n$ , either  $a \text{ dom } b$  or  $b \text{ dom } a$
- Minor exercise: prove these properties for yourself

# Dominator Tree

- The parent of  $n$  is its immediate dominator



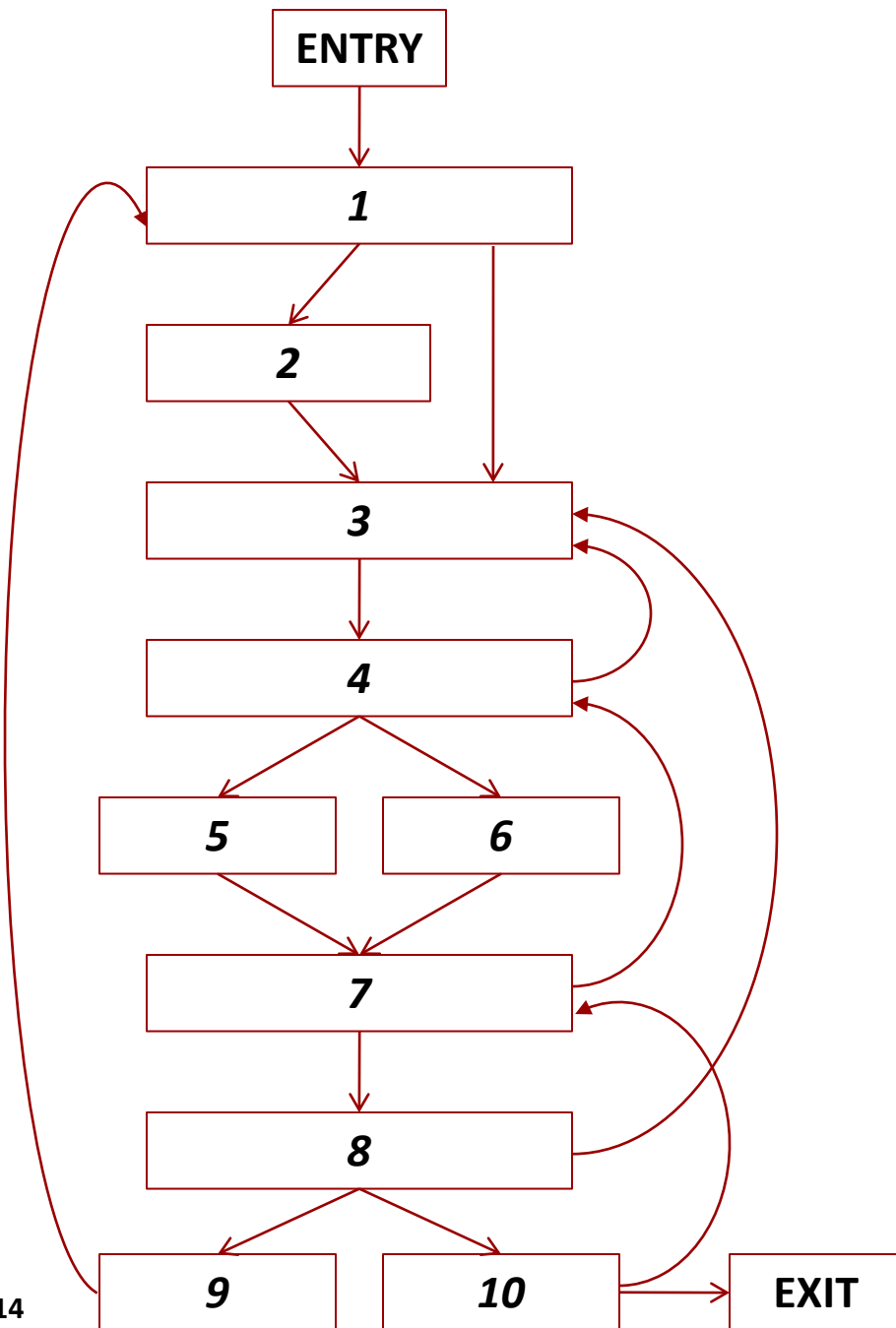
The path from  $n$  to the root contains all and only dominators of  $n$

Constructing the dominator tree: the classic  $O(N\alpha(N))$  approach is from *T. Lengauer and R. E. Tarjan. A fast algorithm for finding dominators in a flowgraph. ACM Transactions on Programming Languages and Systems, 1(1): 121–141, July 1979.*

Many other algorithms: e.g., see *K. D. Cooper, T. J. Harvey and K. Kennedy. A simple, fast dominance algorithm. Software – Practice and Experience, 4:1–10, 2001.*

# Post-Dominance

- A CFG node  $d$  **post-dominates** another node  $n$  if every path from  $n$  to EXIT goes through  $d$ 
  - Implicit assumption: EXIT is reachable from every node
  - A relation  $pdom \subseteq \text{Nodes} \times \text{Nodes}$ :  $d \text{ } pdom \text{ } n$
  - The relation is trivially reflexive:  $d \text{ } pdom \text{ } d$
- Node  $m$  is the **immediate post-dominator** of  $n$  if
  - $m \neq n$ ;  $m \text{ } pdom \text{ } n$ ;  $\forall d \neq n. d \text{ } pdom \text{ } n \Rightarrow d \text{ } pdom \text{ } m$
  - Every  $n$  has a unique immediate post-dominator
- Post-dominance on a CFG is equivalent to dominance on the reverse CFG (all edges reversed)
- **Post-dominator tree**: the parent of  $n$  is its immediate post-dominator; root is EXIT



Extend the previous example with EXIT

ENTRY does not post-dominate any other  $n$

1 *pdom* ENTRY, 1, 9

2 does not post-dominate any other  $n$

3 *pdom* ENTRY, 1, 2, 3, 9

4 *pdom* ENTRY, 1, 2, 3, 4, 9

5 does not post-dominate any other  $n$

6 does not post-dominate any other  $n$

7 *pdom* ENTRY, 1, 2, 3, 4, 5, 6, 7, 9

8 *pdom* ENTRY, 1, 2, 3, 4, 5, 6, 7, 8, 9

9 does not post-dominate any other  $n$

10 *pdom* ENTRY, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10

EXIT *pdom*  $n$  for any  $n$

Immediate post-dominators:

ENTRY  $\rightarrow$  1      1  $\rightarrow$  3

2  $\rightarrow$  3      3  $\rightarrow$  4

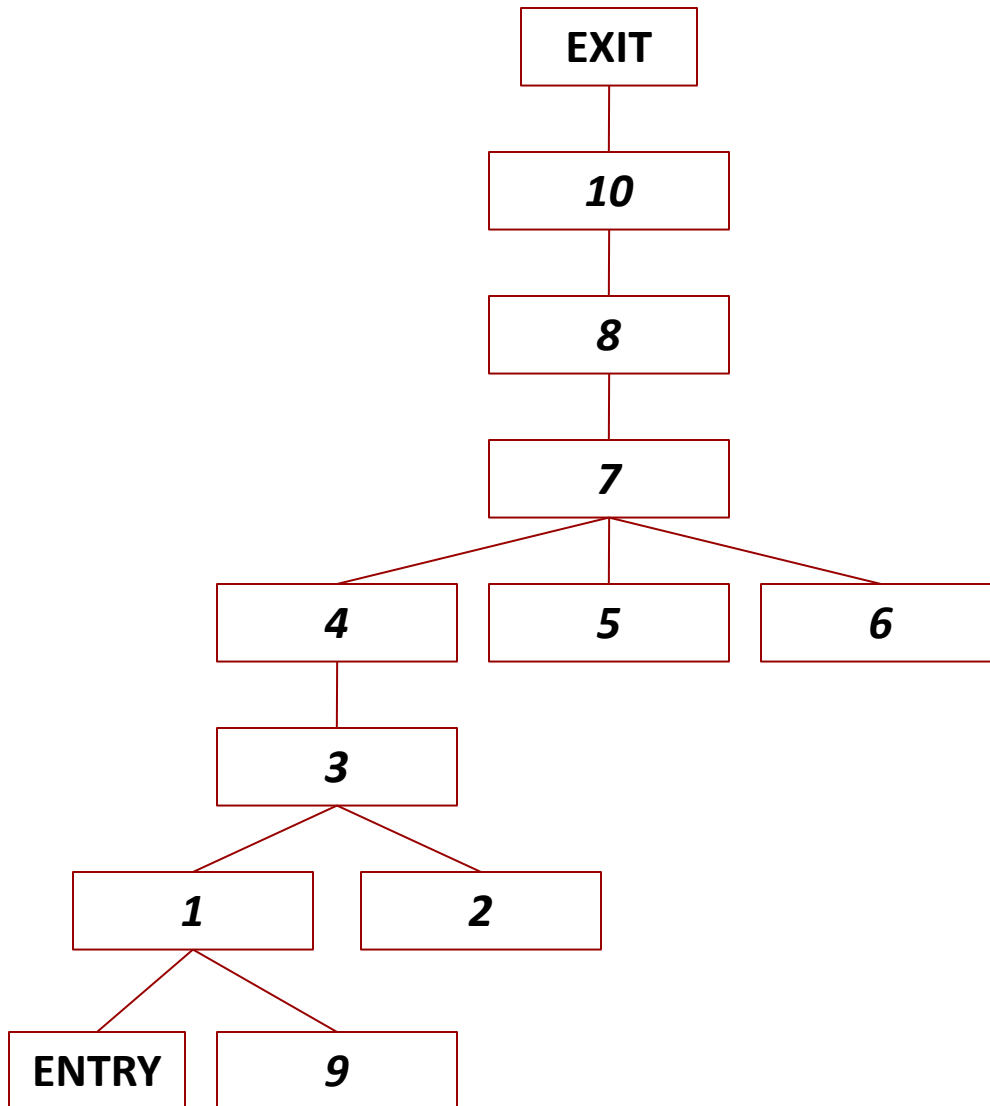
4  $\rightarrow$  7      5  $\rightarrow$  7

6  $\rightarrow$  7      7  $\rightarrow$  8

8  $\rightarrow$  10      9  $\rightarrow$  1

10  $\rightarrow$  EXIT

# Post-Dominator Tree



The path from  $n$  to the root contains all and only post-dominators of  $n$

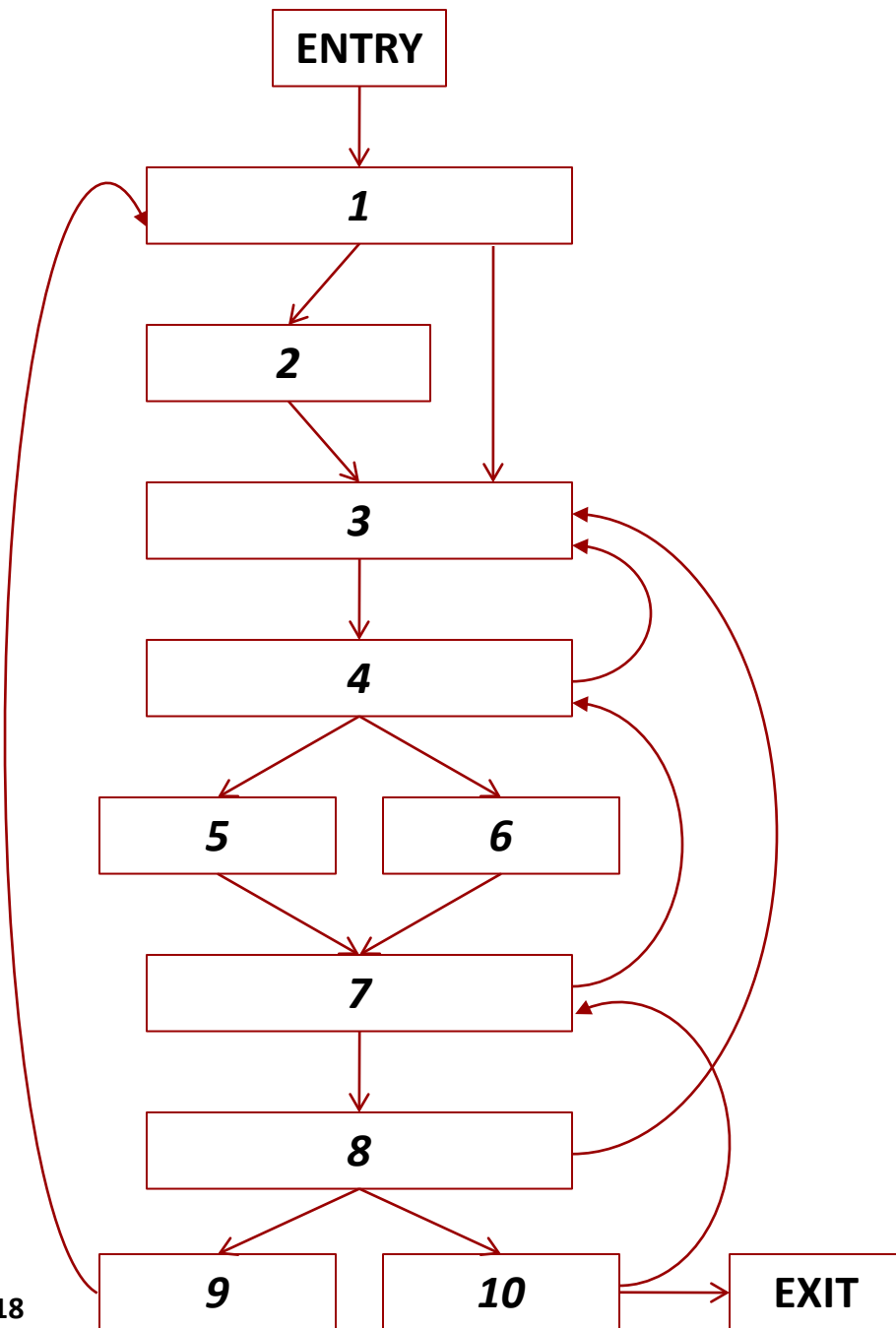
Constructing the post-dominator tree: use any algorithm for constructing the dominator tree; just “pretend” that the edges are reversed

## Part 3: Control Dependence: Informally

- A node  $n$  is control dependent on a node  $c$  if
  - There exists an edge  $e_1$  coming out of  $c$  that definitely causes  $n$  to execute
  - There exists some edge  $e_2$  coming out of  $c$  that is the start of some path that avoids the execution of  $n$
- The decision made at  $c$  affects whether  $n$  gets executed: if  $e_1$  is followed,  $n$  definitely is executed; if  $e_2$  is followed, there is the possibility that  $n$  is not executed at all
  - Thus,  $n$  is **control dependent** on  $c$  – the control-flow leading to  $n$  depends on what  $c$  does

# Control Dependence: Formally

- (part 1)  $n$  is control dependent on  $c$  if
  - $n \neq c$
  - $n$  does **not** post-dominate  $c$
  - there exists a path from  $c$  to  $n$  such that  $n$  post-dominates every node on the path except  $c$
- (part 2)  $n$  is control dependent on  $n$  if
  - there exists a path from  $n$  to  $n$  (with at least one edge) such that  $n$  post-dominates every node on the path
    - this implies that  $n$  has two outgoing edges
    - this case applies to the header of a loop (more later)



Consider all branch nodes  $c$ : 1, 4, 7, 8, 10

[replicated for convenience]

ENTRY does not post-dominate any other  $n$

1 *pdom* ENTRY, 1, 9

2 does not post-dominate any other  $n$

3 *pdom* ENTRY, 1, 2, 3, 9

4 *pdom* ENTRY, 1, 2, 3, 4, 9

5 does not post-dominate any other  $n$

6 does not post-dominate any other  $n$

7 *pdom* ENTRY, 1, 2, 3, 4, 5, 6, 7, 9

8 *pdom* ENTRY, 1, 2, 3, 4, 5, 6, 7, 8, 9

9 does not post-dominate any other  $n$

10 *pdom* ENTRY, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10

EXIT *pdom*  $n$  for any  $n$

2 is control dependent on 1

3, 4, 5, 6 are control dependent on 4

4, 7 are control dependent on 7

9, 1, 3, 4, 7, 8 are control dependent on 8

7, 8, 10 are control dependent on 10

# Finding All Control Dependences

- Consider all CFG edges  $(c,x)$  such that  $x$  does **not** post-dominate  $c$  (therefore,  $c$  is a branch node)
- Traverse the post-dominator tree bottom-up
  - $n = x$
  - while ( $n \neq$  parent of  $c$  in the post-dominator tree)
    - report that  $n$  is control dependent on  $c$
    - $n =$  parent of  $n$  in the post-dominator tree
  - Example: for CFG edge  $(8,9)$  from the previous slide, traverse and report 9, 1, 3, 4, 7, 8 (stop before 10)
- Other algorithms exist, but this one is simple and works quite well

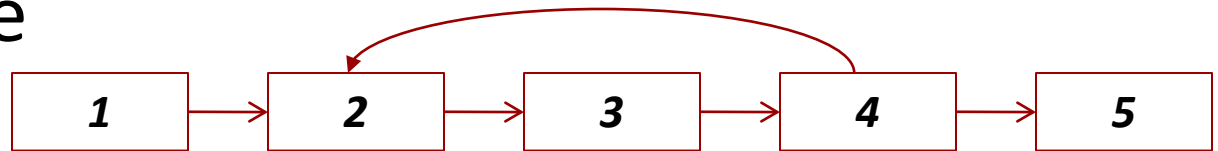
# Why Does This Work?

- Given: edge  $(c,x)$  such that  $x$  does not post-dominate  $c$  (and thus  $c$  is a branch node)
- For any traversed node  $n \neq c$ , we know that
  - $n$  **does not** post-dominate  $c$ 
    - This is why we stop before the parent of  $c$
  - $n$  **does** post-dominate  $x$ : thus, if we follow the  $(c,x)$  edge, we are guaranteed to execute  $n$
  - Easy to show that this is equivalent to part 1 of the definition of control dependence given earlier
- If we traverse  $c$  itself, this means that  $c$  post-dominates  $x$  (thus, part 2 of the definition holds)

## Part 4: Loops in CFGs

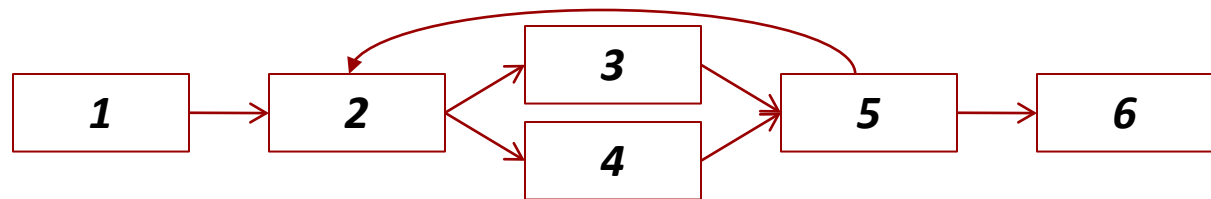
- **Cycle**: sequence of edges that starts and ends at the same node

– Example:



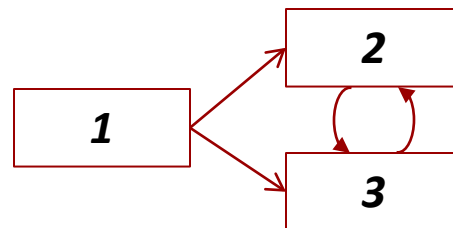
- **Strongly-connected component (SCC)**: a maximal set of nodes such as each node in the set is reachable from every other node in the set

– Example:



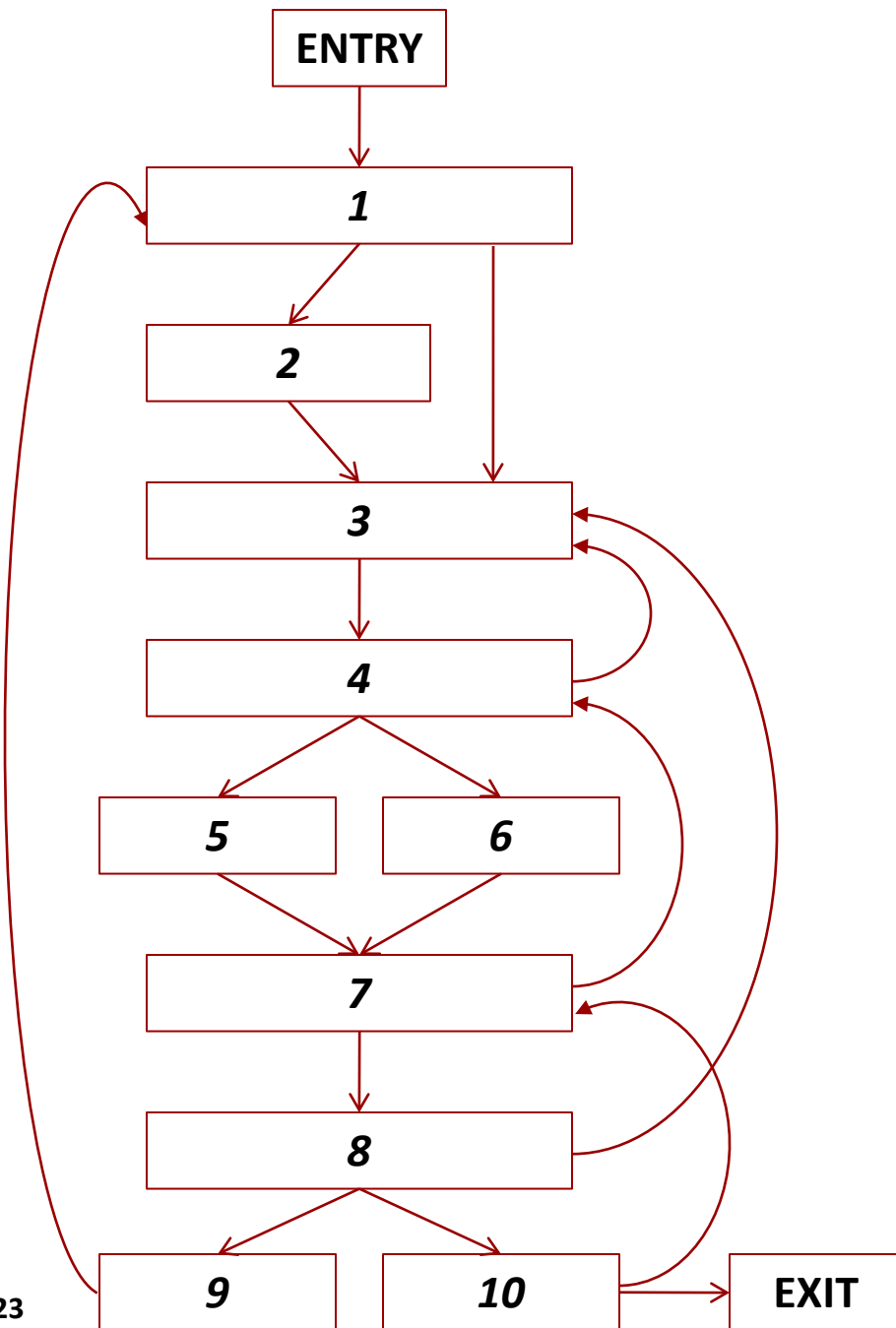
- **Loop**: informally, a strongly-connected component with a single entry point

– An SCC that is not a loop:



# Back Edges and Natural Loops

- Back edge: a CFG edge  $(n, h)$  where  $h$  dominates  $n$ 
  - Easy to see that  $n$  and  $h$  belong to the same SCC
- Natural loop for a back edge  $(n, h)$ 
  - The set of all nodes  $m$  that can reach node  $n$  without going through node  $h$  (trivially, this set includes  $h$ )
  - Easy to see that  $h$  dominates all such nodes  $m$
  - Node  $h$  is the **header** of the natural loop
- Trivial algorithm to find the natural loop of  $(n, h)$ 
  - Mark  $h$  as visited
  - Perform graph traversal starting from  $n$ , but CFG edges are in reverse direction; stop at already-visited nodes
  - All and only visited nodes are in the natural loop



*[replicated for convenience]*

Immediate dominators:

1 → ENTRY	2 → 1	3 → 1
4 → 3	5 → 4	6 → 4
7 → 4	8 → 7	9 → 8
10 → 8	EXIT → 10	

Back edges: **4 → 3**, **7 → 4**, **8 → 3**, **9 → 1**, **10 → 7**

Loop(**10 → 7**) = { 7, 8, 10 }

Loop(**7 → 4**) = { 4, 5, 6, 7, 8, 10 }

Note: Loop(**10 → 7**) ⊆ Loop(**7 → 4**)

Loop(**4 → 3**) = { 3, 4, 5, 6, 7, 8, 10 }

Note: Loop(**7 → 4**) ⊆ Loop(**4 → 3**)

Loop(**8 → 3**) = { 3, 4, 5, 6, 7, 8, 10 }

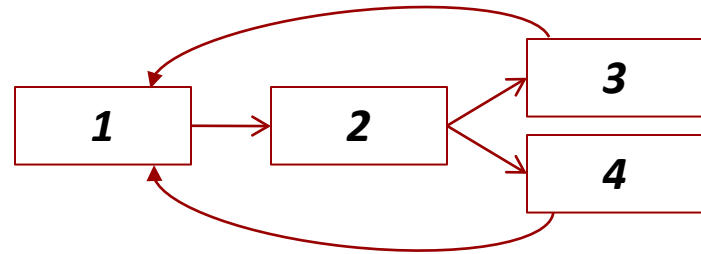
Note: Loop(**8 → 3**) = Loop(**4 → 3**)

Loop(**9 → 1**) = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 }

Note: Loop(**4 → 3**) ⊆ Loop(**9 → 1**)

# Loops in the CFG

- Find all back edges; each target  $h$  of at least one back edge defines a loop  $L$  with  $header(L) = h$
- $body(L)$  is the union of the natural loops of all back edges whose target is  $header(L)$ 
  - Note that  $header(L) \in body(L)$
- Example: this is a single loop with header node 1
- For two CFG loops  $L_1$  and  $L_2$ 
  - $header(L_1)$  is different from  $header(L_2)$
  - $body(L_1)$  and  $body(L_2)$  are either disjoint, or one is a proper subset of the other (nesting – inner/outer)



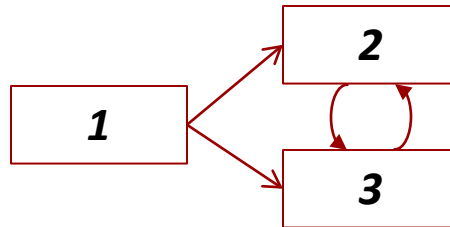
# Flashback to Graph Algorithms

- Depth-first search in the CFG [Cormen et al. book]
  - Set each node's color as *white*
  - Call DFS(ENTRY)
  - DFS( $n$ )
    - Set the color of  $n$  to *grey*
    - For each successor  $m$ : if color is *white*, call DFS( $m$ )
    - Set the color of  $n$  to *black*
- Inside DFS( $n$ ), seeing a grey successor  $m$  means that  $(n, m)$  is a *retreating edge*
  - Note:  $m$  could be  $n$  itself, if there is an edge  $(n, n)$
- The order in which we consider the successors matters: the set of retreating edges depends on it

# Reducible Control-Flow Graphs

- For **reducible** CFGs, the **retreating** edges discovered during DFS are all and only **back** edges
  - The order during DFS traversal is irrelevant: all DFS traversals produce the same set of retreating edges
  - Why the term “reducible”? Details on p. 677
- For **irreducible** CFGs: a DFS traversal may produce retreating edges that are not back edges

- Each traversal may produce different retreating edges
- Example:



- No back edges
- One traversal produces the retreating edge  $3 \rightarrow 2$
- The other one produces the retreating edge  $2 \rightarrow 3$

# Reducibility

- The essence of irreducibility: a SCC with multiple possible entry points
  - If the original program was written using **if-then**, **if-then-else**, **while-do**, **do-while**, **break**, and **continue**, the resulting CFG is always reducible
  - If **goto** was used by the programmer, the CFG could be irreducible (but, in practice, it typically is reducible)
- Optimizations of the intermediate code, done by the compiler, could introduce irreducibility
- Code obfuscation: e.g., Java bytecode can be transformed to be irreducible, making it impossible to reverse-engineer a valid Java source program

# Project 6

- Input: the output of your Project 5
  - A C program that is essentially three-address code
  - Parsed into ROSE to generate the Sage AST
- Goal 1: analyze the AST to generate the CFG
- Goal 2: analyze the CFG to find
  - Back edges (slide 25)
  - Loops (slide 24)
  - We will assume that the CFG is reducible
- Goal 3: print statistics