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

Middle/Back end

- Code optimization: machine-independent optimization of three-address code
- 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; done as part of the code generation phase (e.g., Section 8.5)
- Across basic blocks: global code optimizations; done as part of the code optimization phase
- Other aspects of code generation: e.g., global register allocation

Control-Flow Analysis

Part 1: Constructing a CFG

Part 2: Finding dominators and post-dominators

Part 3: Finding loops in a CFG

- What exactly is a loop? Cannot simply say “whatever CFG subgraph is generated by *while*, *do-while*, and *for* statements” – need a general graph-theoretic definition

Part 4: Finding control dependences in a CFG

- Needed for optimizations: cannot violate dependences
- Needed for analyses in software tools: e.g., program slicing

Part 1: Constructing a CFG

Nodes: basic blocks; edges: possible control flow

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 to the middle of the block)
- Can exit only at the last instruction

Advantages of using basic blocks

- Reduces the cost of compile-time analysis
- Intra-BB optimizations are relatively easy

CFG Construction

Given: the entire sequence of instructions

First, find the leaders (starting instructions of all basic blocks)

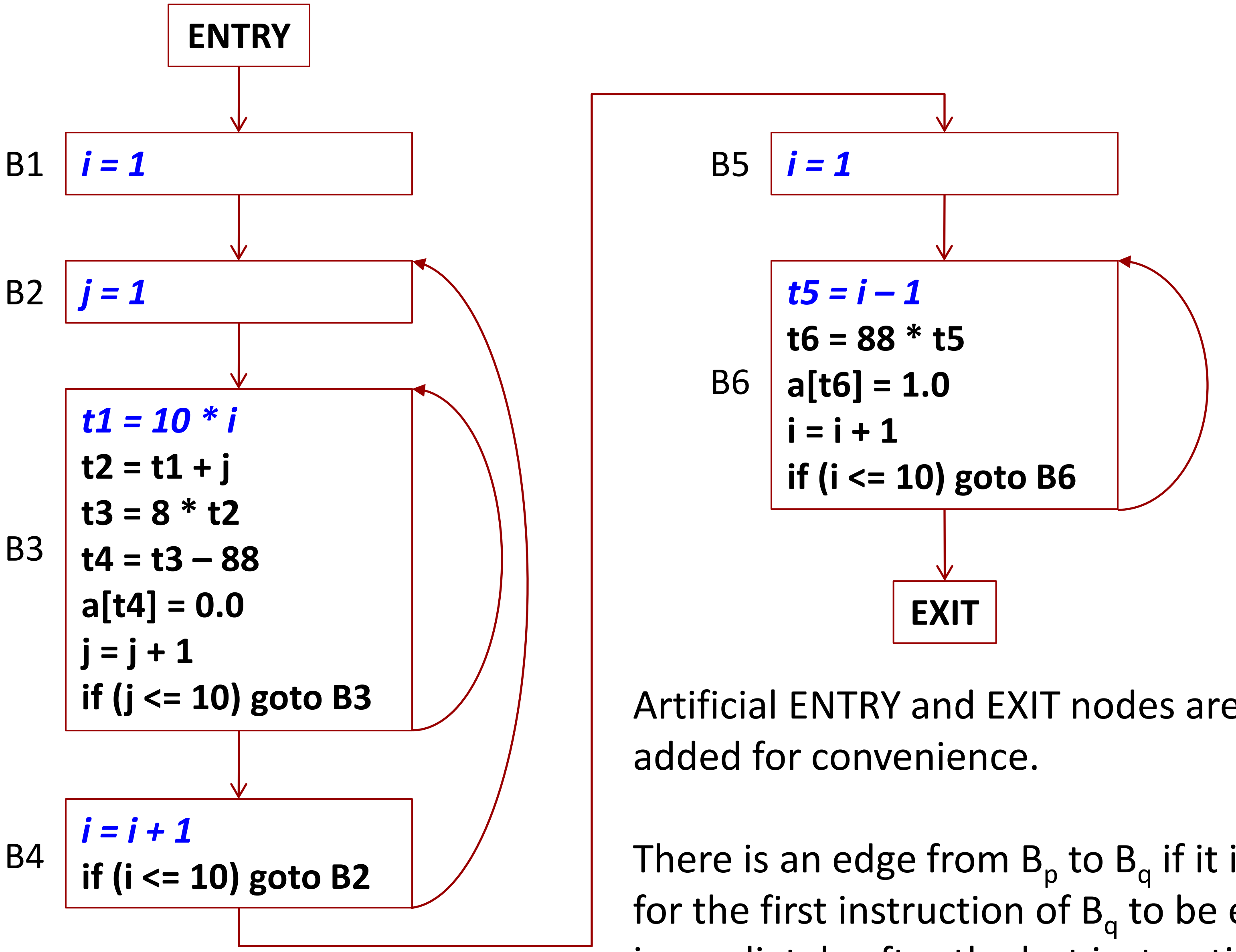
- The first instruction
- The target of any conditional/unconditional jump
- Any instruction that immediately follows a conditional or unconditional jump

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

Example

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

Note: this example sets array elements a[i][j] to 0.0, for 1 <= i,j <= 10 (instructions 1-11). It then sets a[i][i] to 1.0, for 1 <= i <= 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.

Single Exit Node

Single-exit CFG
- If there are multiple exits (e.g., multiple return statements), redirect them to the artificial EXIT node
- Use an artificial compiler-created return variable ***ret***
- ***return expr;*** becomes ***ret = expr; goto exit;***

It gets ugly with exceptions
- Java: e.g., *throw new X()* or null pointer exception
- C: setjmp and longjmp
- We will ignore these

Common assumption
- Every node is reachable from the entry node
- The exit node is reachable from every node
  - Not always true: e.g., a server thread could be *while(true) …*

Practical Considerations [relevant for Project 6]

The usual data structures for graphs can be used

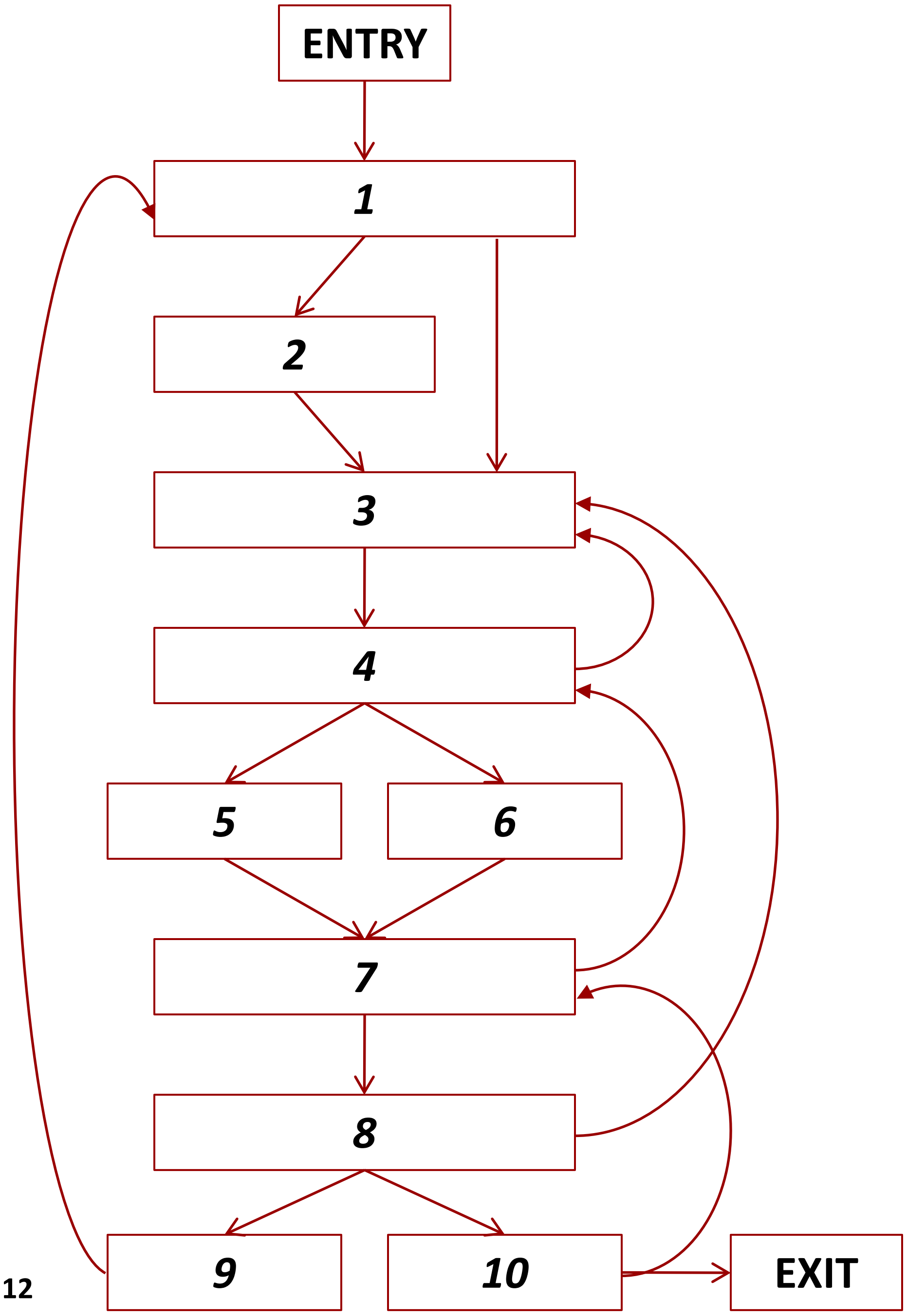
- The graphs are sparse (i.e., have relatively few edges), so an adjacency list representation is the usual choice
  - Number of edges is at most 2 * number of nodes

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 dom n*
  - The relation is trivially reflexive: *d dom d*
- Node *m* is the immediate dominator of *n* if
  - $m \neq n$
  - *m dom n*
  - For any $d \neq n$ such *d dom n*, we have *d dom m*
- Every node has a unique immediate dominator
  - Except ENTRY, which is dominated only by itself



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, EXIT

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

5 does not dominate any other node

6 does not dominate any other node

7 *dom* 7, 8, 9, 10, EXIT

8 *dom* 8, 9, 10, EXIT

9 does not dominate any other node

10 *dom* 10, EXIT

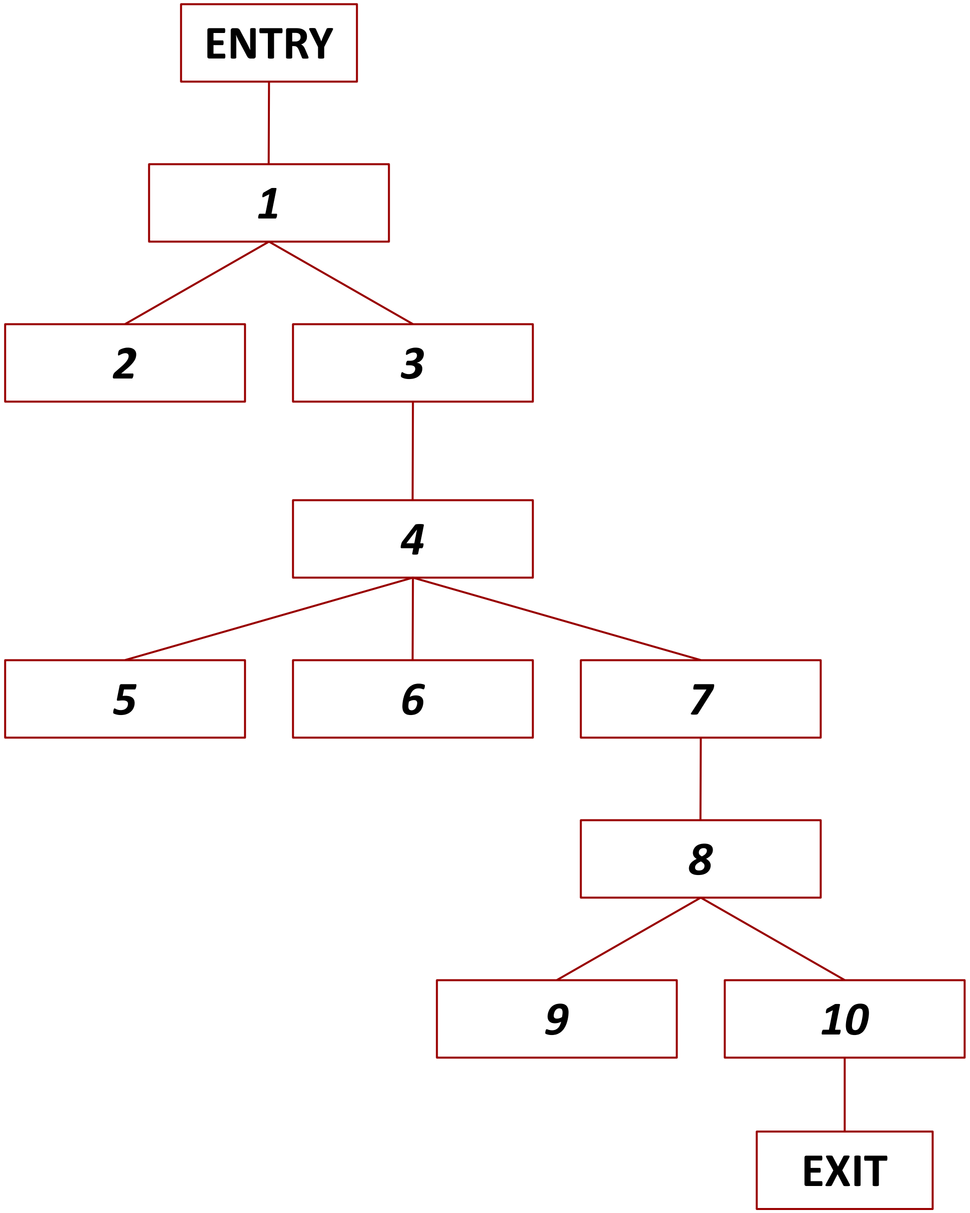
Immediate dominators:

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

A Few Observations

- Dominance is a transitive relation: *a dom b* and *b dom c* means *a dom c*
- Dominance is an anti-symmetric relation: *a dom b* and *b 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 dom b* or *b dom a*
  - Therefore, *dom* is a **total order** for *n*'s dominator set
  - Corollary: for any acyclic path from ENTRY to *n*, all dominators of *n* appear along the path, always in the same order; the last one is the immediate dominator

Dominator Tree



The parent of *n* is its immediate dominator

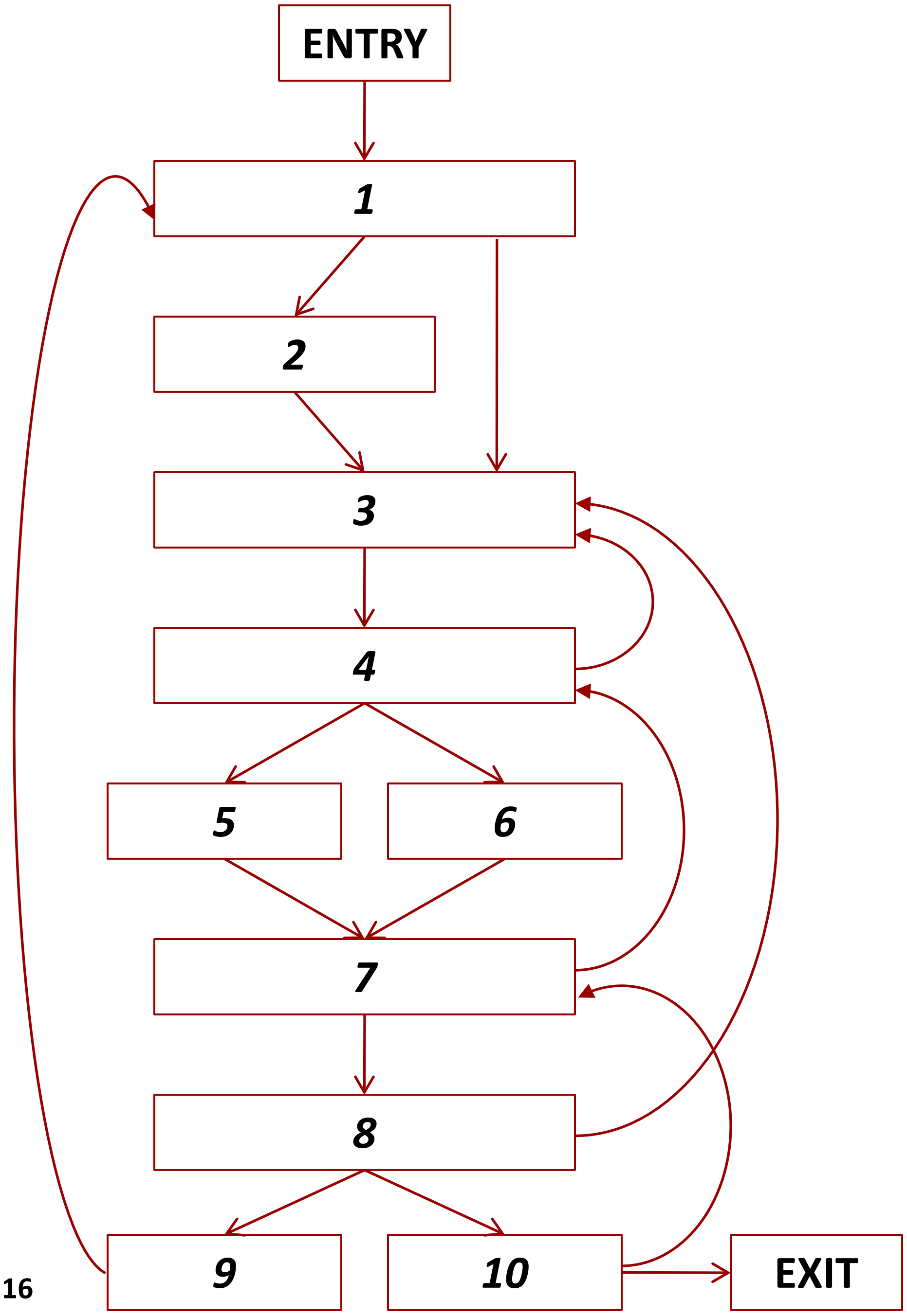
The path from *n* to the root contains all and only dominators of *n*

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* $\subseteq$ Nodes × Nodes: *d pdom n*
  - The relation is trivially reflexive: *d pdom d*
- Node *m* is the immediate post-dominator of *n* if
  - $m \neq n$; $m\ pdom\ n$; $\forall d \neq n.\ \ d\ pdom\ n \Rightarrow d\ pdom\ m$
  - 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



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*

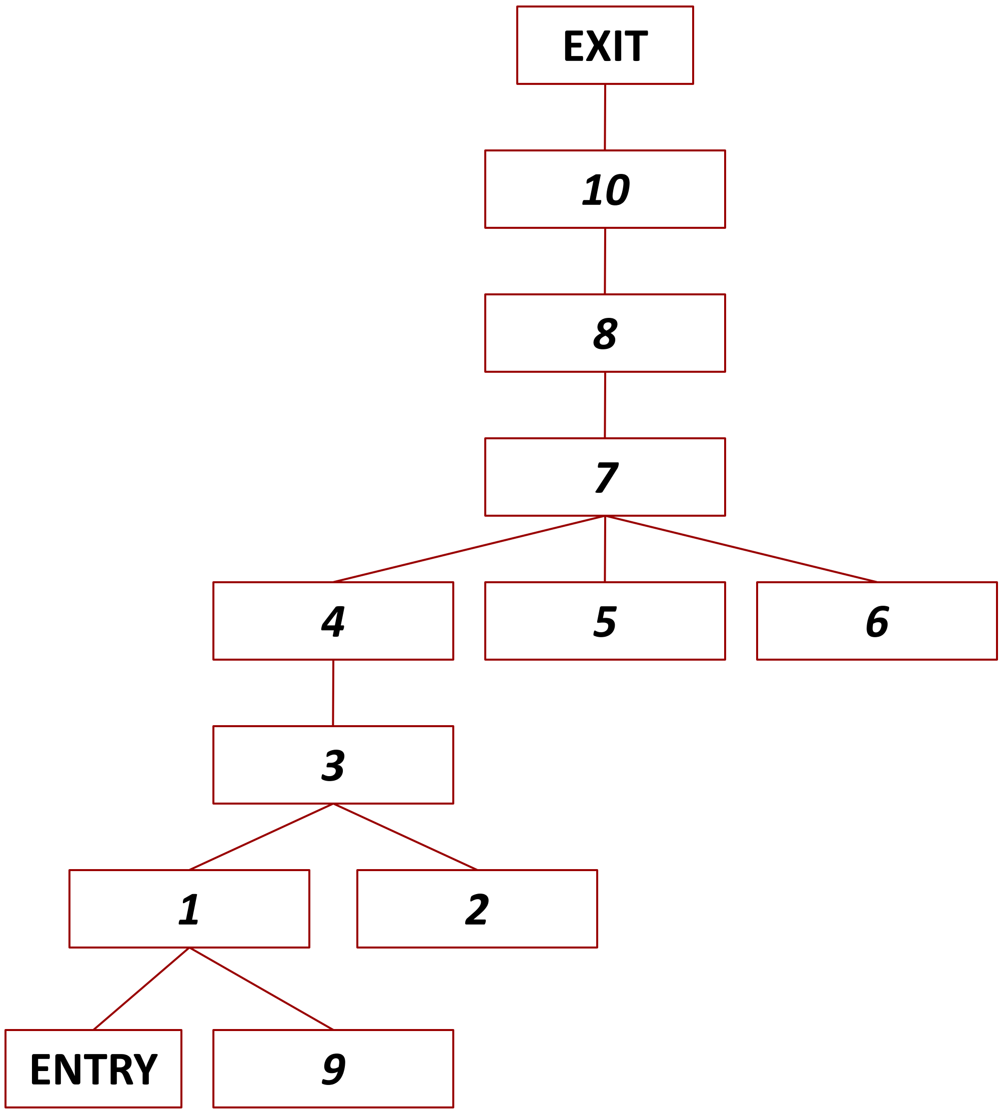
10 *pdom* ENTRY, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10

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

Immediate post-dominators:

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

Post-Dominator Tree

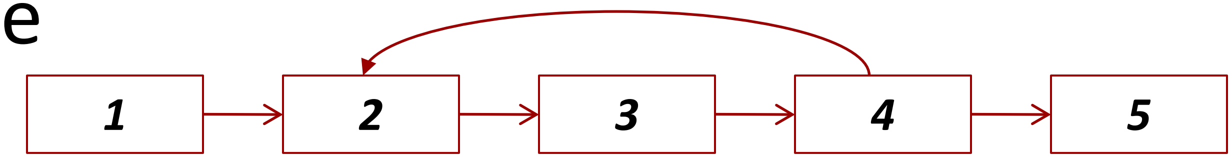


The path from *n* to the root contains all and only post-dominators of *n*

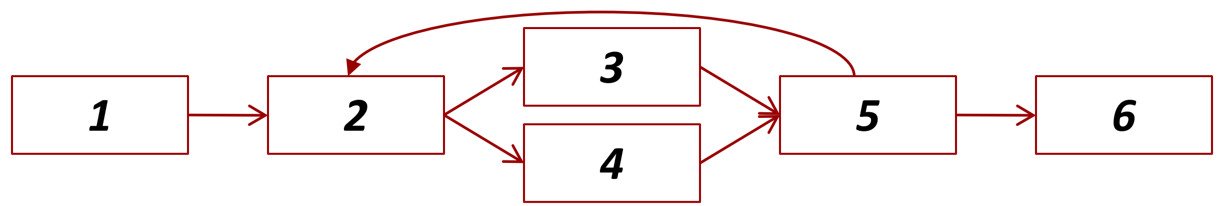
Constructing the post-dominator tree: use any algorithm for constructing the dominator tree; just “pretend” that the edges are reversed

Part 3: Loops in CFGs

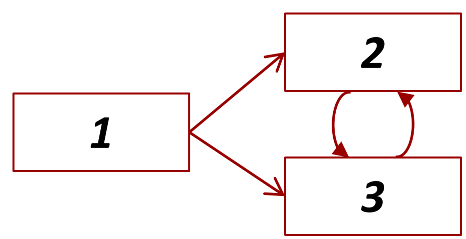
- Cycle: sequence of edges that starts and ends at the same node
  - Example:



- Strongly-connected (induced) subgraph: each node in the subgraph is reachable from every other node in the subgraph
  - Example:

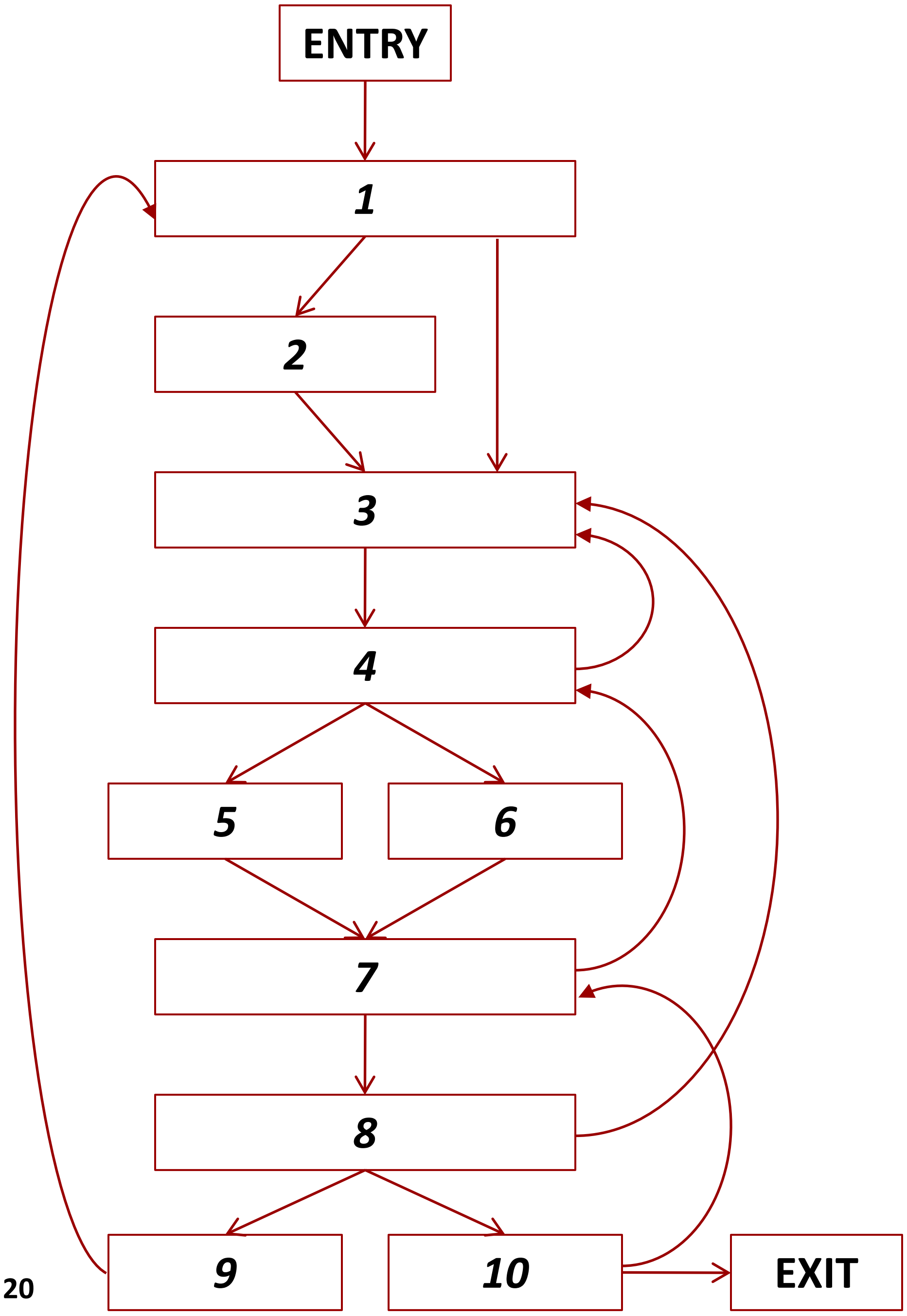


- Loop: informally, a strongly-connected subgraph with a single entry point
  - Not a loop:



Back Edges and Natural Loops

- Back edge: a CFG edge (*n*,*h*) where *h* dominates *n*
- 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
- Simple algorithm to find the natural loop of (*n*,*h*)
  - Mark *h* as visited
  - Perform depth-first search (or breadth-first) starting from *n*, but follow the CFG edges in *reverse* direction
  - All and only visited nodes are in the natural loop



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	EXIT $\rightarrow$ 10	

Back edges: **4 $\rightarrow$ 3**, **7 $\rightarrow$ 4**, **8 $\rightarrow$ 3**, **9 $\rightarrow$ 1**, **10 $\rightarrow$ 7**

Loop(**10 $\rightarrow$ 7**) = { 7, 8, 10 }

Loop(**7 $\rightarrow$ 4**) = { 4, 5, 6, 7, 8, 10 }
Note: Loop(**10 $\rightarrow$ 7**) $\subseteq$ Loop(**7 $\rightarrow$ 4**)

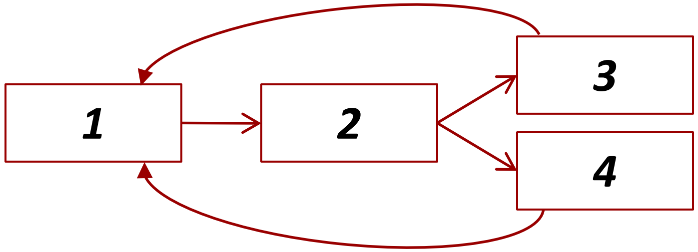
Loop(**4 $\rightarrow$ 3**) = { 3, 4, 5, 6, 7, 8, 10 }
Note: Loop(**7 $\rightarrow$ 4**) $\subseteq$ Loop(**4 $\rightarrow$ 3**)

Loop(**8 $\rightarrow$ 3**) = { 3, 4, 5, 6, 7, 8, 10 }
Note: Loop(**8 $\rightarrow$ 3**) = Loop(**4 $\rightarrow$ 3**)

Loop(**9 $\rightarrow$ 1**) = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 }
Note: Loop(**4 $\rightarrow$ 3**) $\subseteq$ Loop(**9 $\rightarrow$ 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)* ∈ *body(L)*
- Example: this is a single loop with header node 1



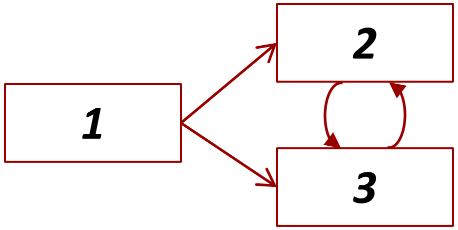
- For any two CFG loops L_1 and L_2
  - *header*(L_1) is different from *header*(L_2)
  - *body*(L_1) and *body*(L_2) are either disjoint, or one is a proper subset of the other (nesting – inner/outer)

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 *gray*
    - For each successor *m*: if color is *white*, call DFS(*m*)
    - Set the color of *n* to *black*
- Inside DFS(*n*), seeing a gray 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
- 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

- A number of equivalent definitions
  - One of them is on the previous page
- Another definition: the graph can be reduced to a single node with the application of the following two rules
  - Given a node *n* with a single predecessor *m*, merge *n* into *m*; all successors of *n* become successors of *m*
  - Remove an edge n → n
- Try this on the graphs from the previous slides
- More details: p. 677 in the textbook

Reducibility

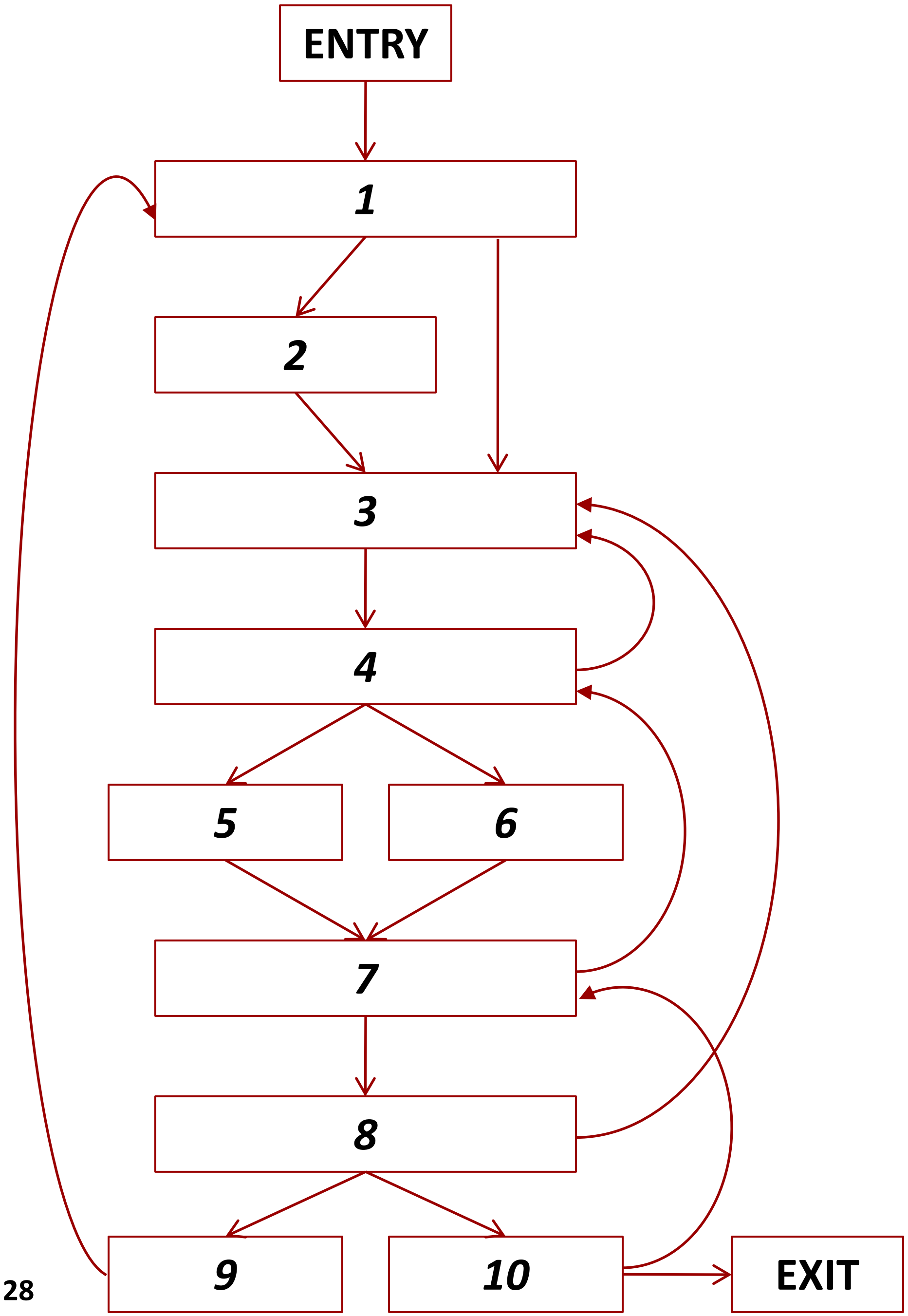
- The essence of irreducibility: a strongly-connected subgraph with multiple possible entry points
  - If the original program was written using **if-then**, **if-then-else**, **while-do**, **do-while**, **break**, and **continue**, the resulting CFG is always reducible
  - If **goto** was used by the 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

Part 4: Control Dependence: Informally

- The decision made at branch node *c* affects whether node *n* gets executed
  - Thus, *n* is control dependent on *c* – the control-flow leading to *n* depends on what *c* does
- 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*
- Informally: *n* postdominates some successor of *c*, but does not postdominate *c* itself

Control Dependence: Formally

- (part 1) *n* is control dependent on *c* if
  - $n \neq c$
  - *n* does not post-dominate *c*
  - there is an edge $c \rightarrow m$ such that *n* post-dominates *m*
- (part 2) *n* is control dependent on *n* if
  - there exists a path (with at least one edge) from *n* to *n* such that *n* post-dominates every node on the path
    - this happens in the presence of loops; *n* is the source node of a loop exit edge



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

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* != 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)

Why Does This Work? [no need to study this proof]

- Given: edge (*c*,*x*) such that *x* does not post-dominate *c*
- 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)