Control-Flow Analysis

"Dragon book" [Ch. 8, Section 8.4; Ch. 9, Section 9.6]

Compilers: Principles, Techniques, and Tools, 2nd ed. by

Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jerey D. Ullman

on reserve in 18th Ave Library (ask for CSE 5343 textbook)

URL from the course web page [copyrighted material, please do not link to it or distribute it]

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
  - Across basic blocks: global code optimizations; done as part of the code optimization phase
  - 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?We cannot simply say “whatever CFG subgraph is generated by *while*, *do-while*, and *for* statements” – need a general graph-theoretic definition
- Part 4: Static single assignment form (SSA)
- Part 5: Finding control dependences
  - Necessary as part of constructing the program dependence graph (PDG), a popular IR for software tools for slicing, refactoring, testing, and debugging

Part 1: Constructing a CFG

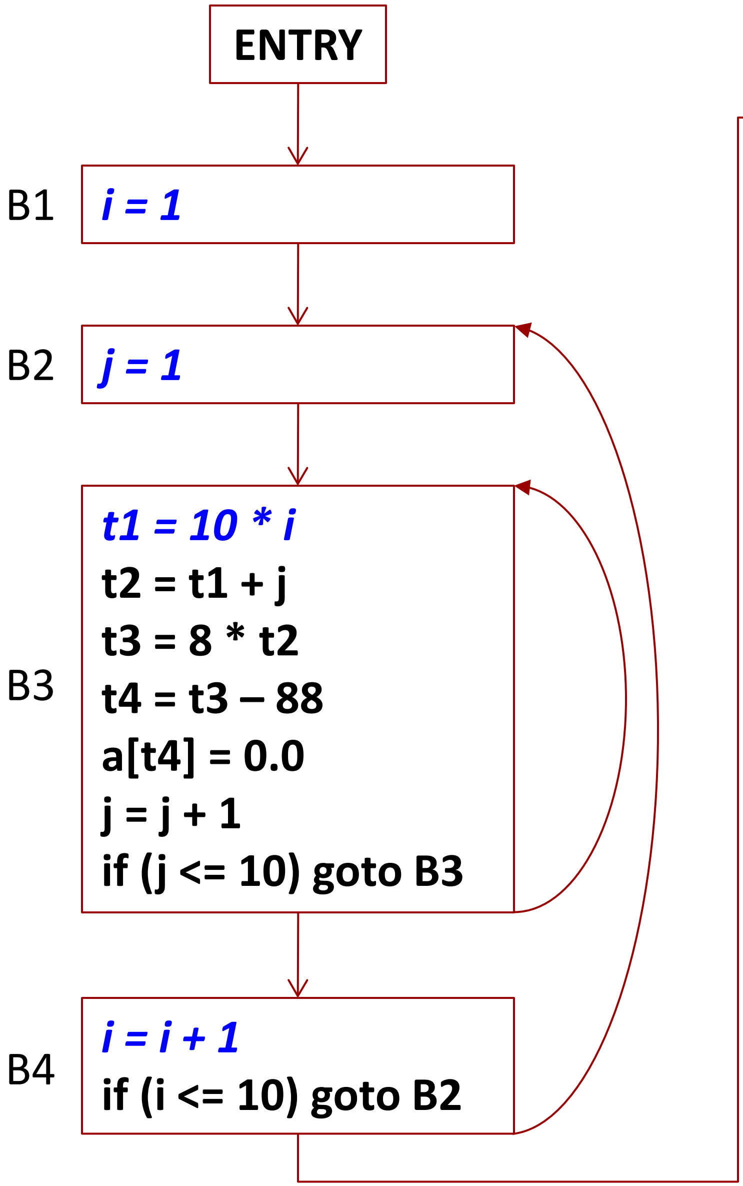
- Basic block: maximal sequence of consecutive three-address instructions such that
  - The flow of control can enter only through the first instruction (i.e., no jumps into the middle of the block)
  - The flow of control can exit only at the last instruction
- Given: the entire sequence of instructions
- 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

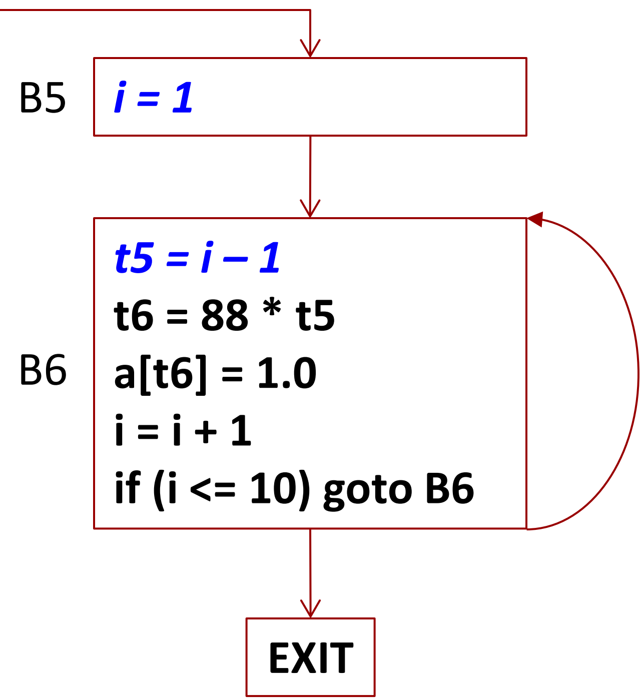
Constructing a CFG

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

Instruction	Leader
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.

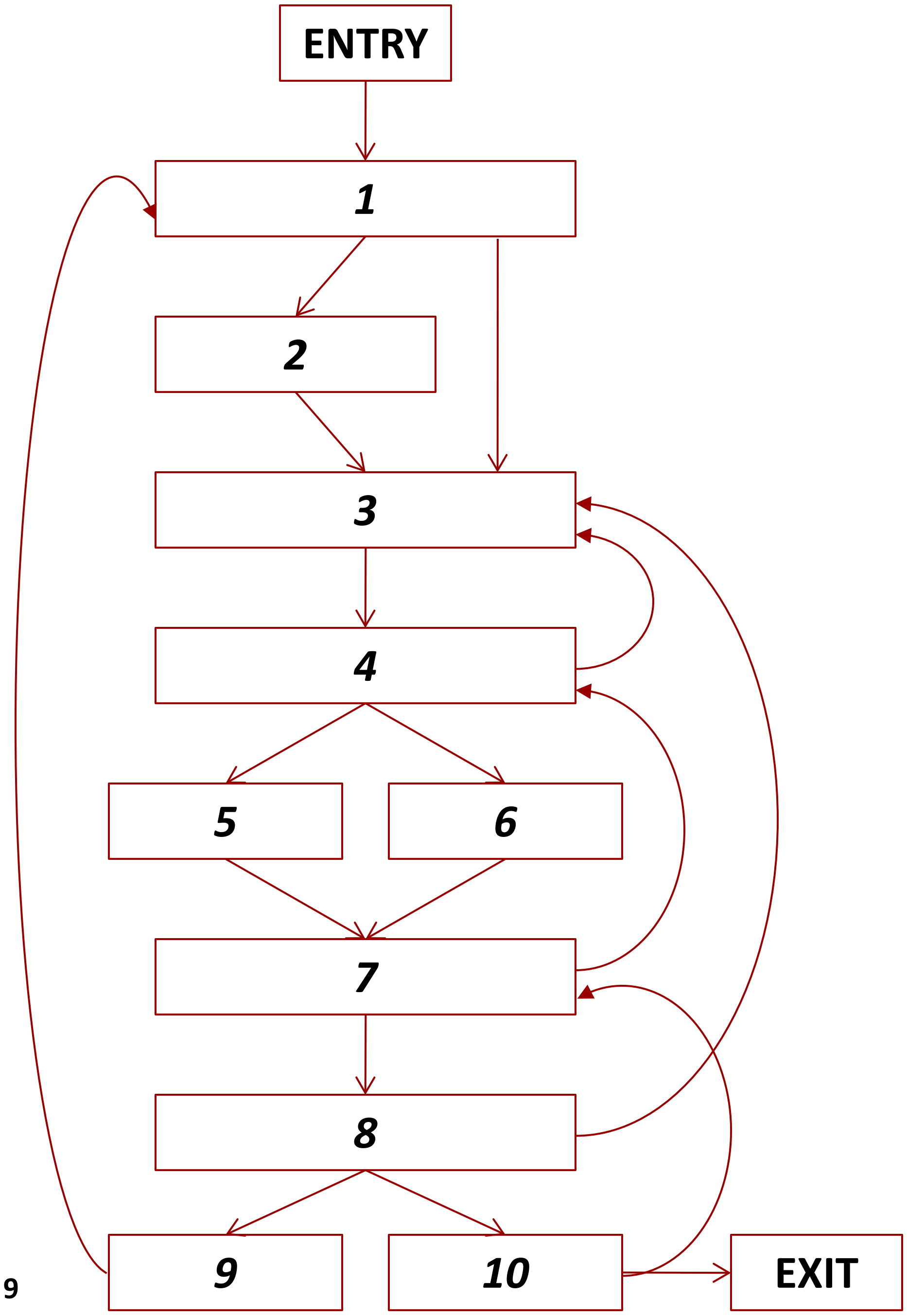
There is an edge from B_p to B_q if it is possible for the first instruction of B_q to be executed immediately after the last instruction of B_p. This is conservative: e.g., **if (3.14 > 2.78)** still generates two edges.

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 * 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 $dom \subseteq$ Nodes × 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:

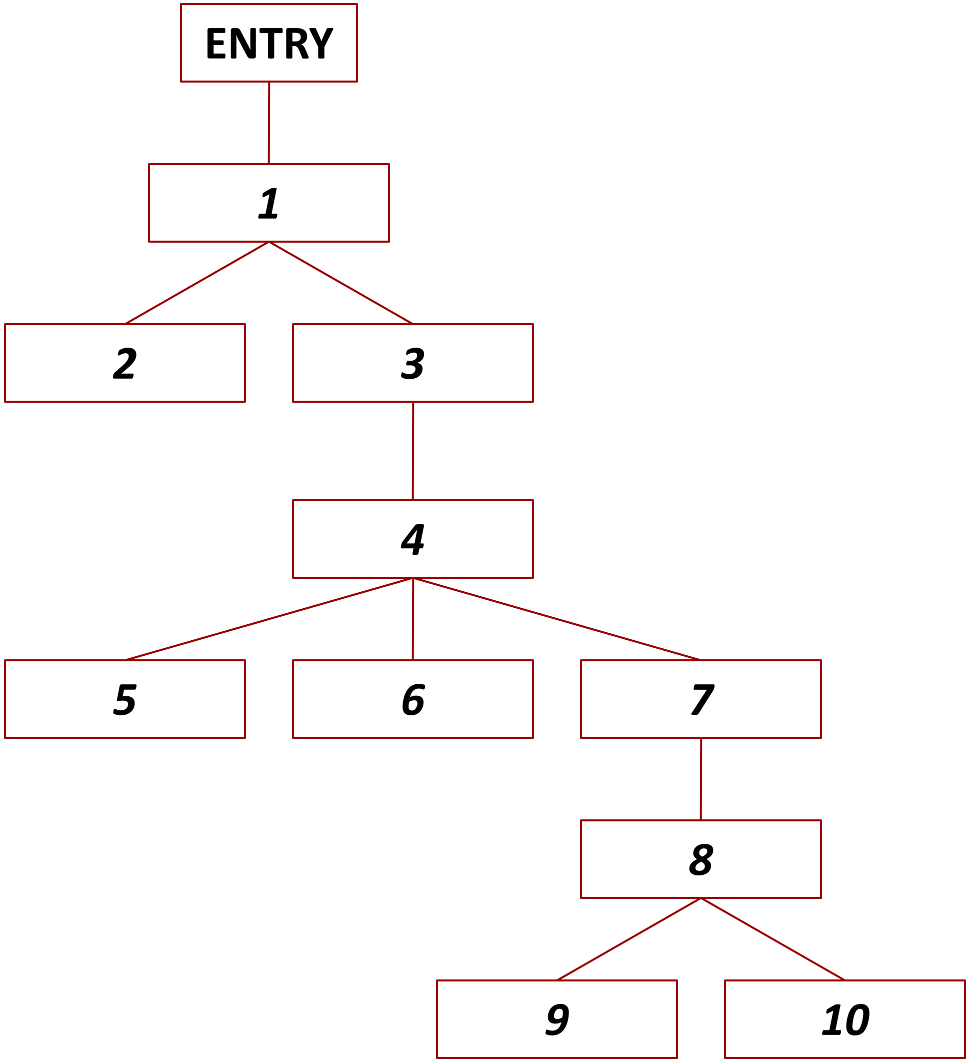
1 → ENTRY	2 → 1
3 → 1	4 → 3
5 → 4	6 → 4
7 → 4	8 → 7
9 → 8	10 → 8
	EXIT → 10

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



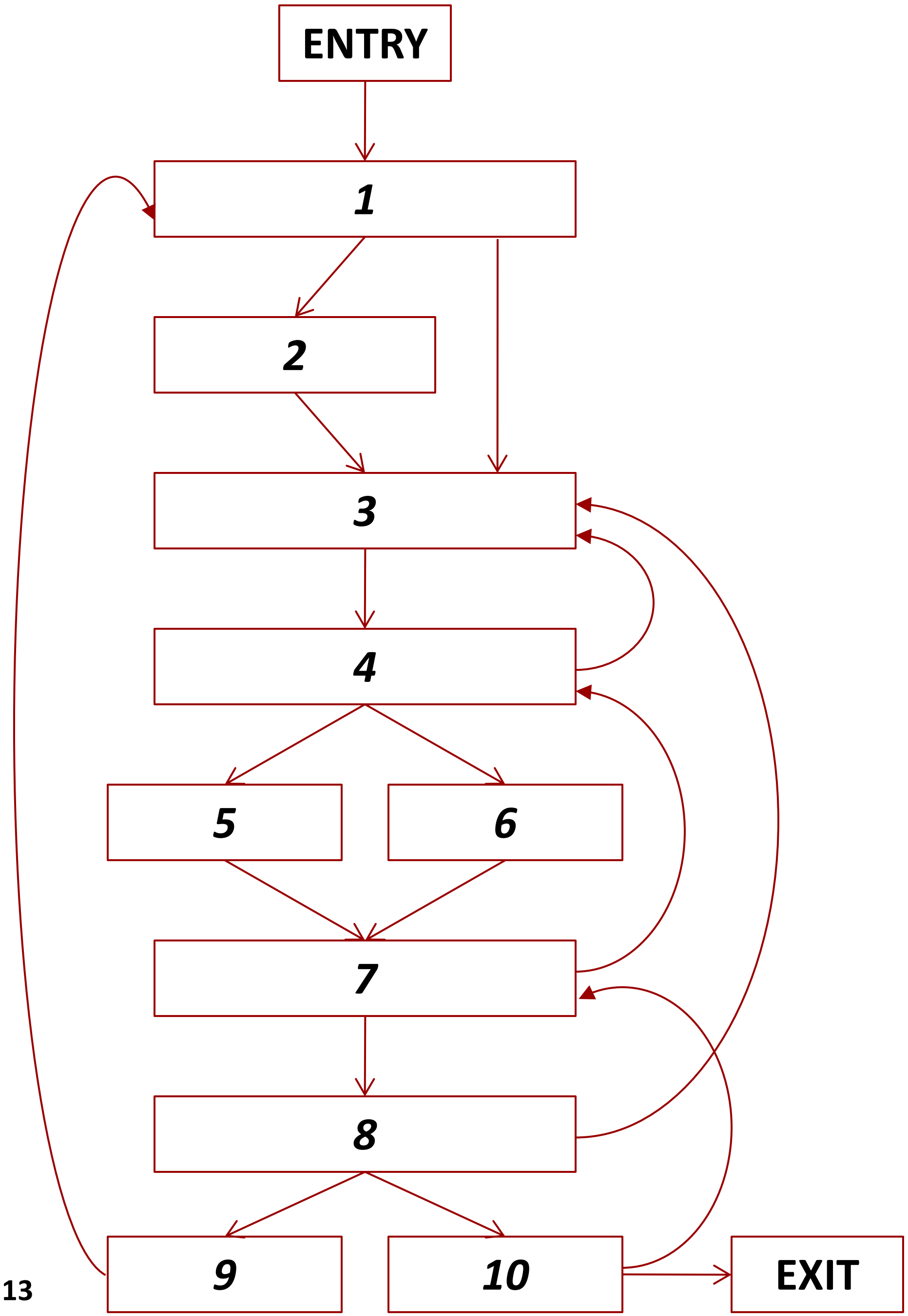
The path from *n* to the root contains all and only dominators of *n*

Constructing the dominator tree: the classic O(Nα(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*

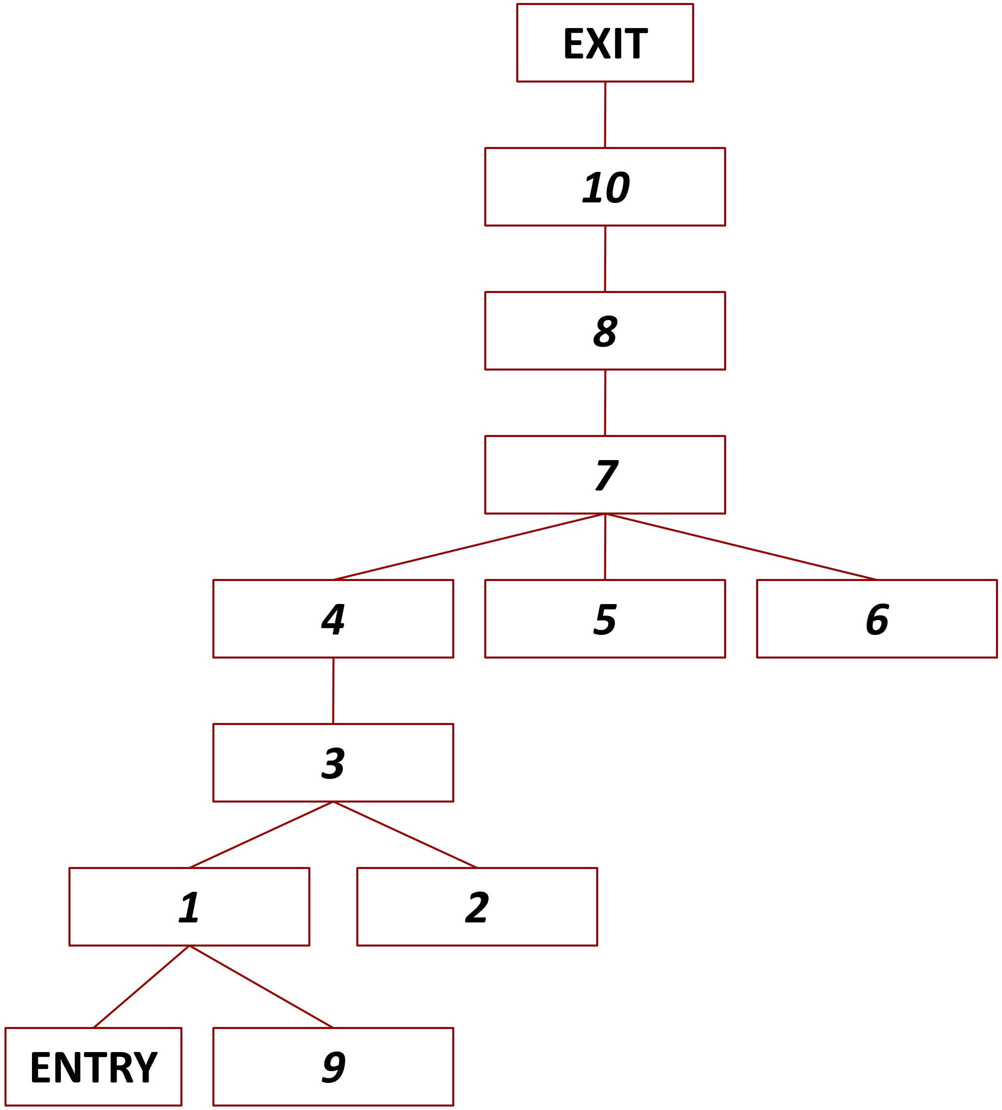
10 *pdom* ENTRY, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10

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

Computing the Dominator Tree

- Theoretically superior algorithms are not necessarily the most desirable in practice
- Our choice: Cooper et al., 2001
- Formulation and algorithm based on insights from dataflow analysis
  - Essentially, solving a system of mutually-recursive equations – more later …
- You should read the paper carefully and implement the algorithm for computing the dominator tree

Details on the Algorithm

- Given: CFG $G=(N,E,n_0)$, compute DOM(n) for each n
  - All nodes dominating n, including n itself
- Assumption: all nodes are reachable from n_0
  - Issue: unreachable code in *catch(Exception e) …*
- Visit the nodes in reverse postorder
  - Recall depth-first search: it grows a DFS spanning tree
    - Postorder in this DSF spanning tree
    - During DSF, whenever a node becomes “black” (p. 604 of CLRS-3), it is postorder-visited
  - Do DFS from ENTRY, put the nodes on a list (e.g., ArrayList in Java) in the reverse of this order

Details on the Algorithm

$$\textsc{Dom}(n_0) = \{n_0\}$$

$$\textsc{Dom}(n) = \left(\bigcap_{p \in preds(n)} \textsc{Dom}(p) \right) \cup \{n\}$$

Details on the Algorithm

$$
\begin{array}{l}
\textit{for all nodes, } n \\
\quad \textsc{Dom}[n] \leftarrow \{1 \ldots N\} \\
\textit{Changed} \leftarrow \textit{true} \\
\textit{while (Changed)} \\
\quad \textit{Changed} \leftarrow \textit{false} \\
\quad \textit{for all nodes, } n, \textit{ in reverse postorder} \\
\quad\quad \textit{new\_set} \leftarrow \left(\bigcap_{p \in preds(n)} \textsc{Dom}[p] \right) \bigcup \; \{n\} \\
\quad\quad \textit{if } (\textit{new\_set} \neq \textsc{Dom}[n]) \\
\quad\quad\quad \textsc{Dom}[n] \leftarrow \textit{new\_set} \\
\quad\quad\quad \textit{Changed} \leftarrow \textit{true}
\end{array}
$$

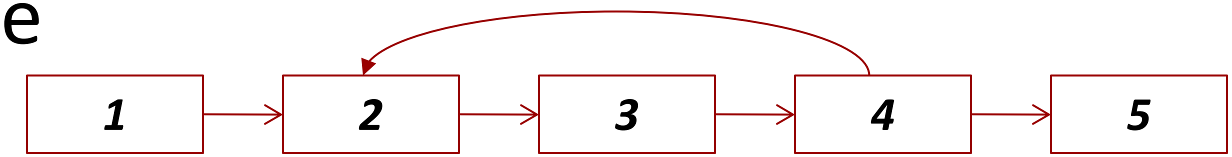
Note: DOM(ENTRY) = { ENTRY } and this node is never processed by the algorithm (so, it should be *"for all nodes other than ENTRY …."*)

Details on the Algorithm

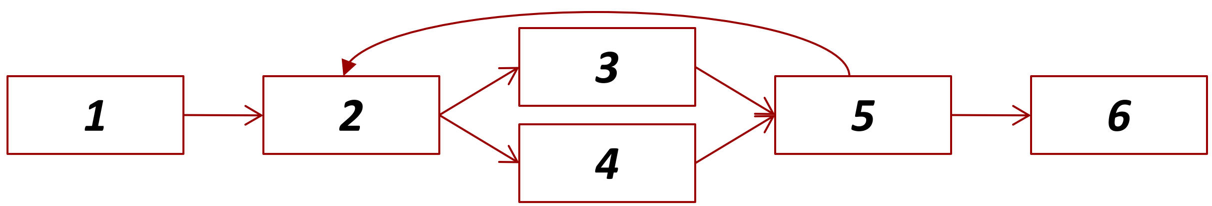
- (Re-)compute DOM(n) as the intersection of DOM(m) for all predecessor nodes m, union { n }
  - If any DOM set changes, recompute everything
- Reverse postorder guarantees efficient algorithm
  - d(G)+2 iterations of the *while(Changed)* loop; d(G) = max number “retreating” edges on any acyclic path
- **Problem**: representation and intersection of sets are expensive, in terms of time and memory
  - Also, we do not find the immediate dominators
- **Solution**: careful algorithm design (Fig 3 in the paper)
  - You will implement this algorithm in a project

Part 3: Loops in CFGs

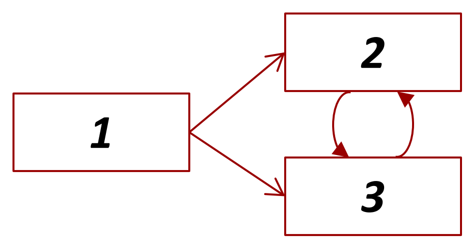
- Cycle: sequence of edges that starts and ends at the same node
  - Example:



- 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:

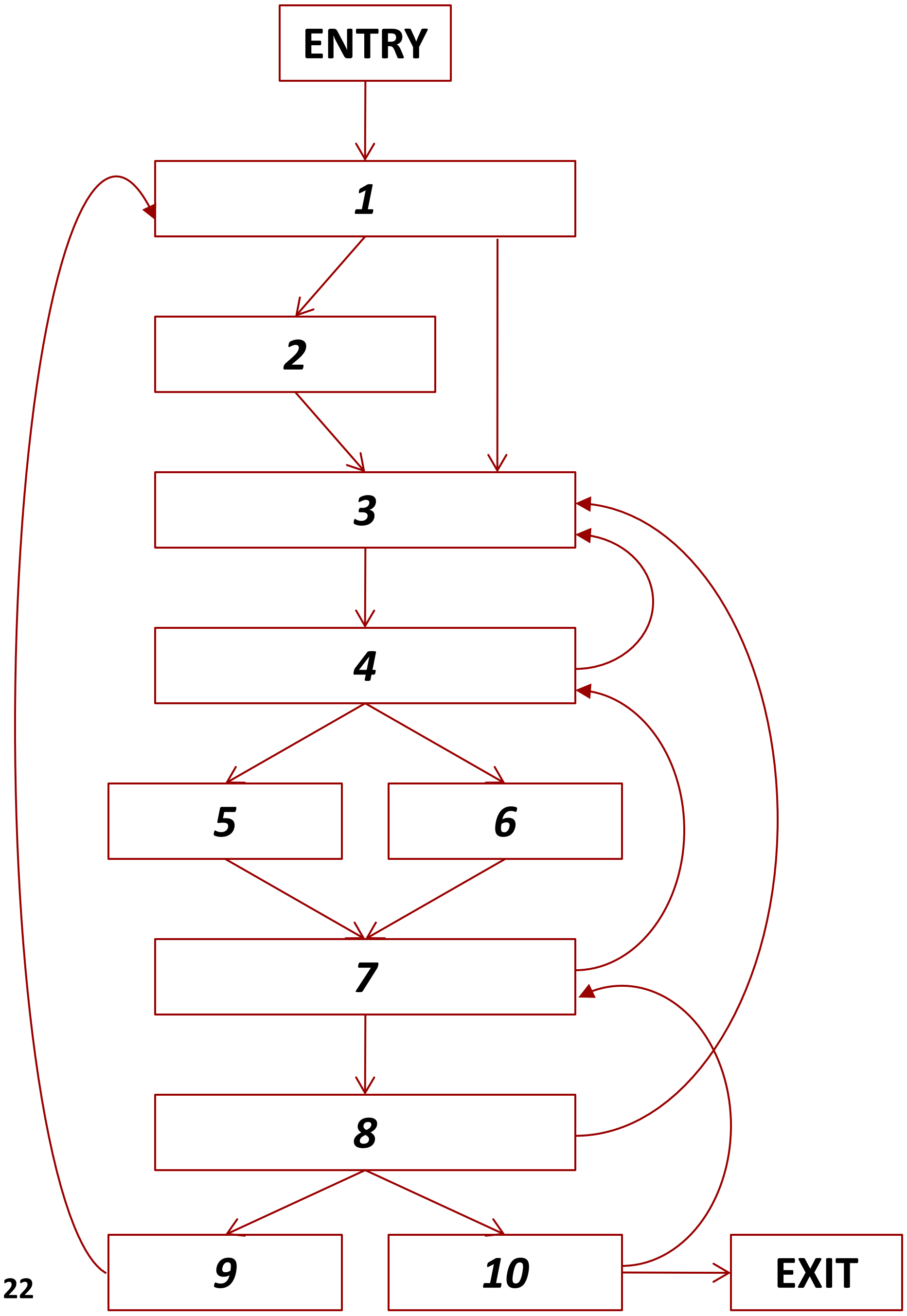


- Loop: informally, a strongly-connected component with a single entry point
  - An SCC that is not a loop:



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 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 }

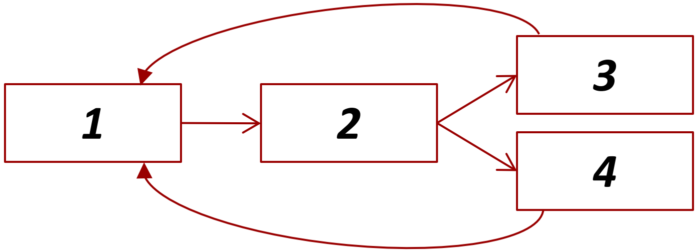
Note: Loop(**$8 \rightarrow 3$**) = Loop(**$4 \rightarrow 3$**)

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



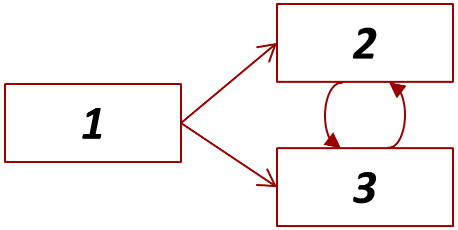
- For two CFG loops L_1 and L_2
  - *header*(L_1) is different from *header*(L_2)
  - *body*(L_1) and *body*(L_2) are either disjoint, or one is a proper subset of the other (nesting – inner/outer)

Graph Algorithms

- DFS again (p. 604 of CLRS-3)
  - 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
- 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 we already saw
- 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

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

Part 4: Static Single Assignment (SSA) Form

- Source: Cytron et al., ACM TOPLAS, Oct. 1991
  - Section 1 (ignore Section 1.1)
  - Section 2
  - Section 3 (ignore Section 3.1)
  - Section 4 (ignore the detailed proofs in Section 4.3)
- Key ideas
  - Insert ϕ-functions at join points (Sections 3 and 4)
    - Based on **dominance frontiers**
  - Rename the variables so that **each use (read)** of a variable is reached by **exactly one definition (write)** of that variable – i.e., by a **single assignment**
    - Section 5.2 discusses this issue, but we will not

Examples

```
V ← 4
  ← V + 5
V ← 6
  ← V + 7
```

$$\begin{aligned} V_1 &\leftarrow 4 \\ &\leftarrow V_1 + 5 \\ V_2 &\leftarrow 6 \\ &\leftarrow V_2 + 7 \end{aligned}$$

Fig. 2. Straight-line code and its single assignment version.

```
if P
   then V ← 4
   else V ← 6

/* Use V several times. */
```

```
if P
   then V1 ← 4
   else V2 ← 6
V3 ← φ(V1, V2)
/* Use V3 several times. */
```

Fig. 3. **if-then-else** and its single assignment version.

```
I ← 1
J ← 1
K ← 1
L ← 1
repeat




  if (P)
    then do
      J ← I
      if (Q)
        then L ← 2
        else L ← 3

      K ← K + 1
    end
    else K ← K + 2




  print(I,J,K,L)
  repeat

    if (R)
      then L ← L + 4

  until (S)
  I ← I + 6
until (T)
```

```
I1 ← 1
J1 ← 1
K1 ← 1
L1 ← 1
repeat
  I2 ← φ(I3,I1)
  J2 ← φ(J4,J1)
  K2 ← φ(K5,K1)
  L2 ← φ(L9,L1)
  if (P)
    then do
      J3 ← I2
      if (Q)
        then L3 ← 2
        else L4 ← 3
      L5 ← φ(L3,L4)
      K3 ← K2 + 1
    end
    else K4 ← K2 + 2
  J4 ← φ(J3,J2)
  K5 ← φ(K3,K4)
  L6 ← φ(L2,L5)
  print(I2,J4,K5,L6)
  repeat
    L7 ← φ(L9,L6)
    if (R)
      then L8 ← L7 + 4
    L9 ← φ(L8,L7)
  until (S)
  I3 ← I2 + 6
until (T)
```

Placement of ϕ Functions

- ϕ functions are used in “fake” assignments of the form $V_k \leftarrow \phi(V_i,V_j)$
  - Along one edge, variable V has the value of V_i; along the other, the value of V_j
  - If multiple incoming edges: $\phi(V_i,V_j,\ldots, V_m)$
- Naïve: for each V, check each pair of assignments to V; do they reach a common join point?
- Better: for each V, consider each assignment to V and find its **dominance frontier**; place ϕ for V
  - And then find the dominance frontier of each ϕ, and place ϕ there as well, and so on …

Dominance Frontier (DF)

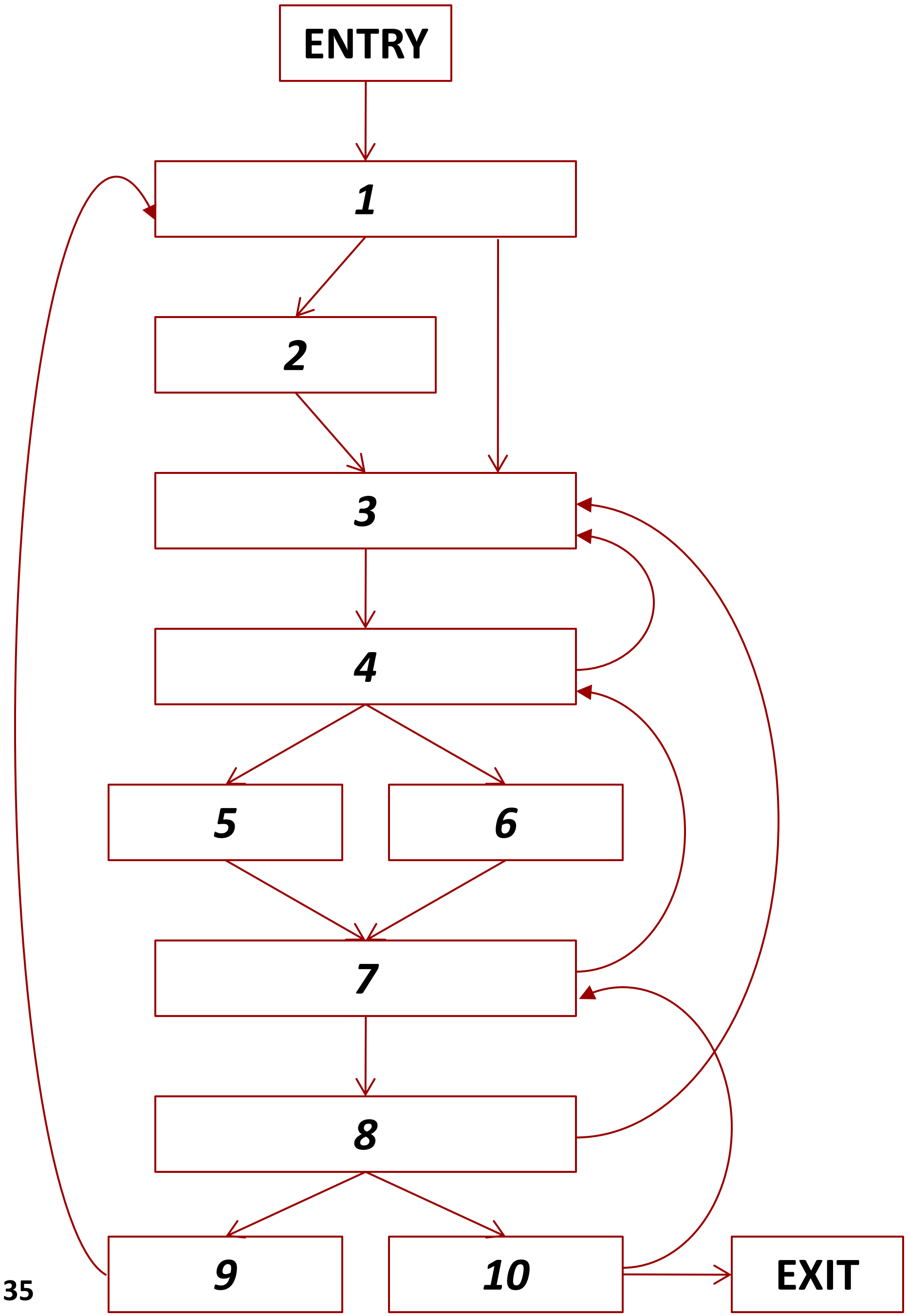
- Suppose node x is an assignment to V
- DF(**x**) = { **y** | for some edge **z** → **y**, **x** dominates **z** but **x** does not strictly dominate **y** }
- A few observations
  – **y** must be a join point. Why?
  – If the flow of control reaches **y** from **z**, the value of V is either the one assigned at **x**, or at some node "between" **x** and **z**
  – If the flow of control reaches **y** from some other predecessor (not **z**), the value of V may come from a different assignment to V
- DF algorithm: Sec 5; alternative: Cooper et al. 2001

Part 5: 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
- See Cytron et al., 1991, Section 6 for more details
  - *c* belongs to DF(*n*) but computed on the *reverse* CFG



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)
- Other algorithms exist, but this one is simple and works quite well