CS 6340
Software Analysis and Testing

Mary Jean Harrold

Aristotle Research Group
SPARC/CERCS
College of Computing
Georgia Tech

Class 1

- Introductions; Student Information
- Details (syllabus, etc.)
  - Shown on T-Square (https://t-square.gatech.edu)
- Basic Analyses (1): intermediate representations, control-flow analysis,
- Assign
  - Basic Analyses (1): Be familiar with concepts
  - Representation and Analysis of Software (Sections 1-5) (Schedule has link)
  - Problem Set 1 (Schedule has link): due 8/25/09

Course Overview, Syllabus

- Motivation for studying program analysis and testing
- Course objectives
  - Learn traditional, promising analyses
  - Learn traditional, new applications
  - Explore research areas in analysis, use of artifacts
  - Apply analyses and applications through homework, semester project
- Means for approaching course objectives
  - Class lectures, readings, homework, class presentations
  - Semester project (proposal, oral, written report)
  - Exams

Course Overview, Syllabus

- Your responsibilities
  - Arrive on time, attend all classes
  - Prepare (read papers before class), participate in class
  - Submit homework, projects, etc. at the beginning of class on the due date
- Course evaluation
  - Homework: 30%
  - Semester project (proposal, written, oral): 30%
  - Exams: 30%
  - Class participation: 10%
- Prerequisites
  - CS 4240, graduate-level standing, permission of instructor

Overview of Course

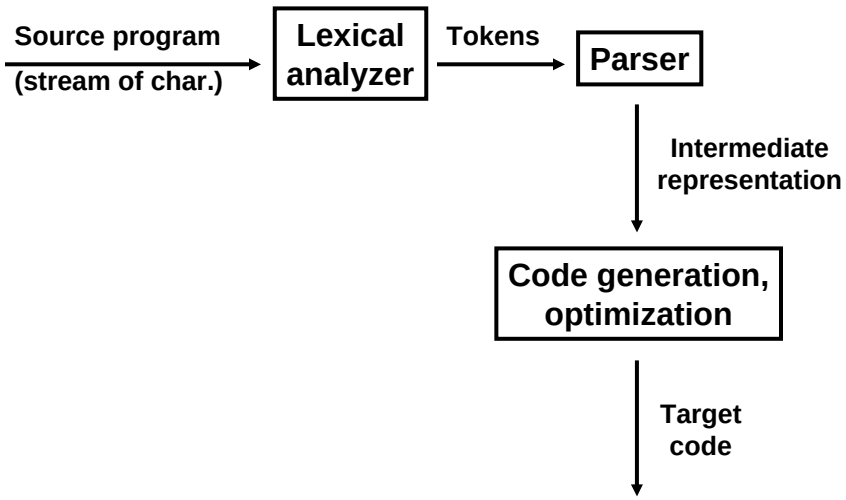
- Static analyses (computed without execution)
  - Intraprocedural (within a single procedure)
    - AST, control-flow, control-dependence, data-flow, etc.
  - Complicating factors
    - Interprocedural (across procedure boundaries), recursion
    - Pointers, references, polymorphism, dynamic binding, etc.
  - Slicing, analysis by reachability, demand analysis
  - Applications
- Dynamic analyses (computed by execution)
  - Instrumentation, profiling
  - Dynamic versions of control-flow, etc.
  - Applications such as testing, debugging,
- Combinations of static and dynamic analyses

Overview of Course

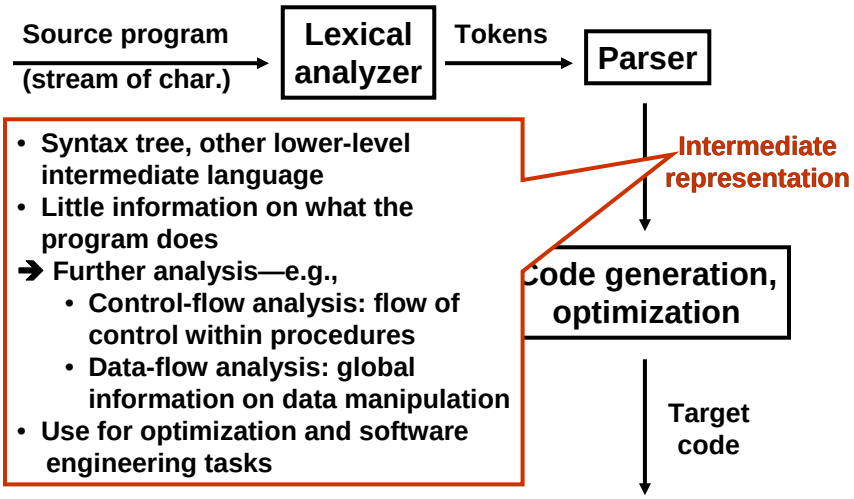
- Static analyses (computed without execution)
  - Intraprocedural (within a single procedure)
    - AST, control-flow, control-dependence, data-flow, etc.
  - Complicating factors
    - Interprocedural (across procedure boundaries), recursion
    - Pointers, references, polymorphism, dynamic binding, etc.
  - Slicing, analysis by reachability, demand analysis
  - Applications
- Dynamic analyses (computed by execution)
  - Instrumentation, profiling
  - Dynamic versions of control-flow, etc.
  - Applications such as testing, debugging,
- Combinations of static and dynamic analyses

Intermediate Representations

Intermediate Representations (traditional)



Intermediate Representations (traditional)



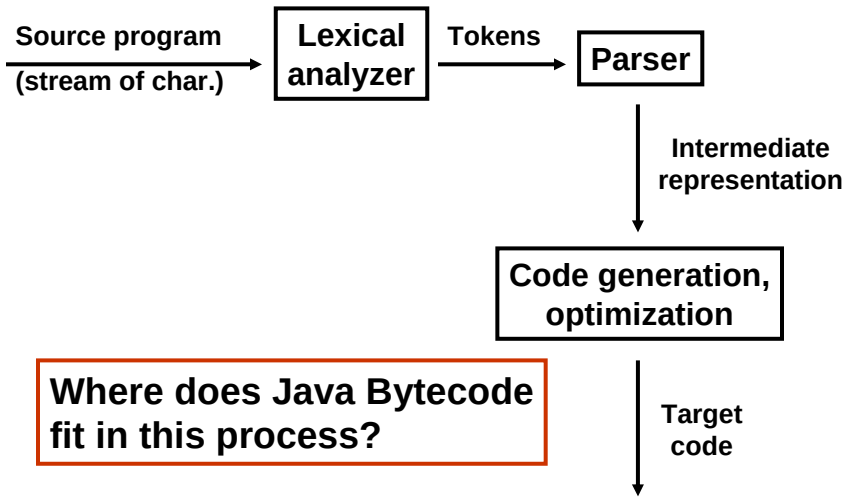
- Syntax tree, other lower-level intermediate language
- Little information on what the program does

➔ Further analysis—e.g.,

- Control-flow analysis: flow of control within procedures
- Data-flow analysis: global information on data manipulation

- Use for optimization and software engineering tasks

Intermediate Representations (traditional)



Where does Java Bytecode fit in this process?

Abstract Syntax Tree (AST)

- Concrete versus abstract syntax
  - Concrete shows structure *and* is language-specific
  - Abstract shows structure
- Representations
  - Parse tree represents concrete syntax
  - Abstract syntax tree represents abstract syntax

Example: Grammar

Examples

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 → "+" | "-" | …

Example: Parse tree and AST

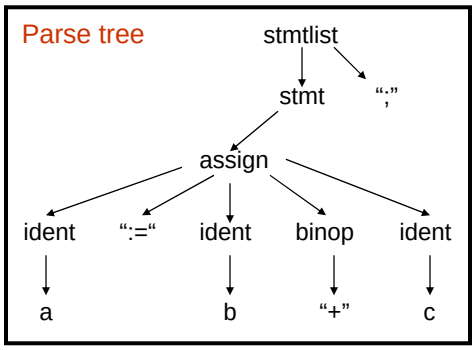
- Example: a := b + c;
- Grammar
  - stmtlist -> stmt “**;**” | stmt “**;**” stmtlist

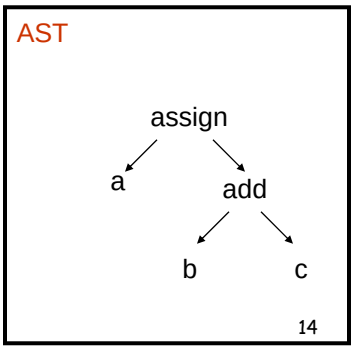
    stmt -> assign | if-then | …

    assign -> ident “**:=**” ident binop ident

    binop -> “**+**” | “**-**”

    …





Three Address Code

- General form: x := y *op* z
- May include temporary variables (intermediate values)
- Types (examples; rest in handout)
  - Assignment
    - Binary x := y *op* z
    - Unary x := *op* y
  - Copy x := y
  - Jumps
    - Unconditional *goto L*
    - Conditional *if x relop y goto L*
  - …

Example: Three Address Code

Source code

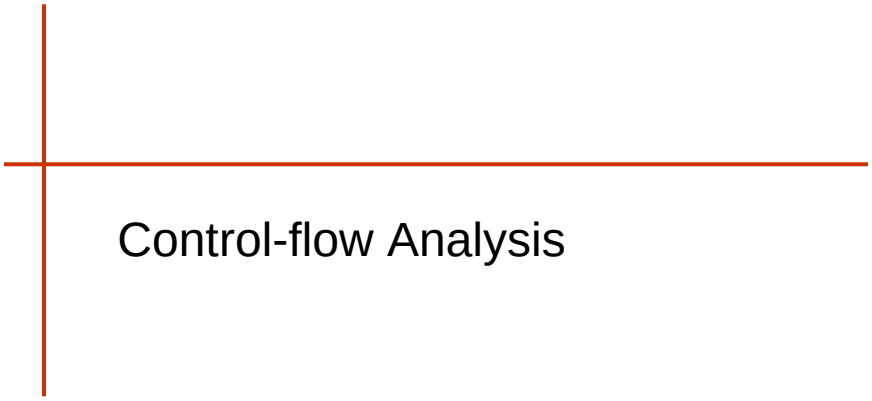
if a > 10 **then**
  x = y + z
else
  x = y – z
…

Corresponding 3-address code

- if a > 10 goto 4
- x = y – z
- goto 5
- x = y + z
- …

Analysis Levels

- Local: within a single basic block or statement
- Global, Intraprocedural: within a single procedure, function, or method (sometimes, intramethod)
- Interprocedural: across procedure boundaries, procedure call, shared globals, etc.
- Intraclass: within a single class
- Interclass: across class boundaries
- Intramodule: within a single module
- …



Control-flow Analysis

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 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)
```

Control flow is a relation (i.e., set of ordered pairs)

- that represents the possible flow of execution in a program
- (a, b) in the relation means that control can flow from node a to node b during execution.

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)
```

Control flow is a relation (i.e., set of ordered pairs)

- that represents the possible flow of execution in a program
- (a, b) in the relation means that control can flow from node a to node b during execution.

What is the control-flow relation for Procedure AVG?

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)
```

Control flow is a relation (i.e., set of ordered pairs)

- that represents the possible flow of execution in a program
- (a, b) in the relation means that control can flow from node a to node b during execution.

{(entry,S1),(S1,S2), (S2,S3), (S3,S4), (S3,S9), (S4,S5), (S5,exit), (S4,S6), (S6,S7), (S7,S8), (S8,S3), (S9,S10), (S10,exit)}

What is the control-flow relation for Procedure AVG?

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)
```

Control-flow Graph (CFG) is a way to represent the control-flow relation:

- *nodes* represent elements in pairs (A,B)
- *edges* represent the relation between A and B
- *labels* represent the conditions that cause that branch to be executed
- *entry* and *exit* nodes added

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)
```

Control-flow Graph (CFG) is a way to represent the control-flow relation:

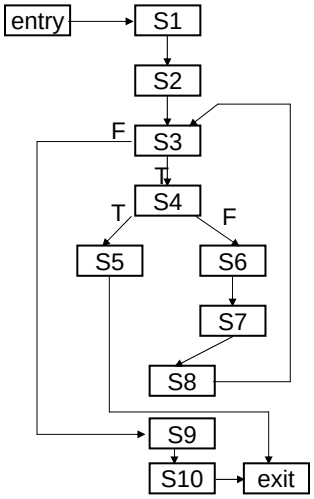
- *nodes* represent elements in pairs (A,B)
- *edges* represent the relation between A and B
- *labels* represent the conditions that cause that branch to be executed
- *entry* and *exit* nodes added

What is the control-flow graph for Procedure AVG?

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)
```



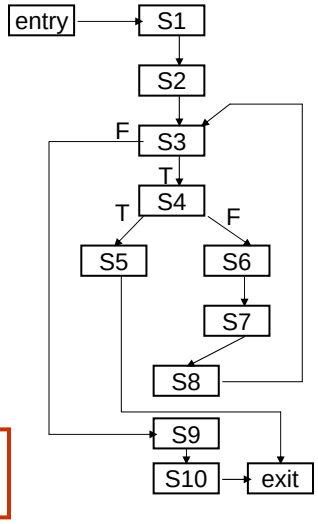
Control Flow: Basic Blocks

- A basic block is a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without halt or possibility of branch except at the end
- 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

CFG with Maximal Basic Blocks

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)
```

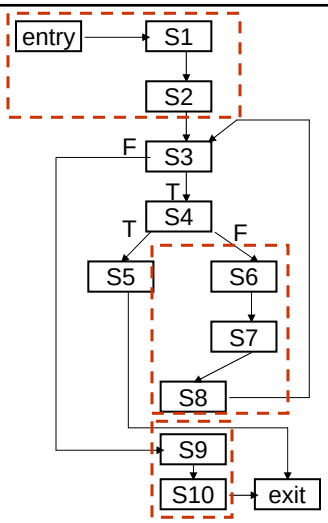


What are the maximal basic blocks for Procedure AVG?

CFG with Maximal Basic Blocks

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: Algorithm

- *Input*: a list of program statements in some form
- *Output*: A list of control-flow graph (CFG) nodes and edges
- *Method*:
  - Construct basic blocks

How?

  - Create entry and exit nodes; create edge (entry, B1); create (Bk, exit) for each Bk that represents an exit from program
  - Add CFG edge from Bi to Bj if Bj can immediately follow Bi in some execution, i.e.,

How do we determine this?

  - Label edges that represent conditional transfers of control

Computing Control Flow: Algorithm

- *Input*: a list of program statements in some form
- *Output*: A list of control-flow graph (CFG) nodes and edges
- *Method*:
  - Construct basic blocks
  - Create entry and exit nodes; create edge (entry, B1); create (Bk, exit) for each Bk that represents an exit from program
  - Add CFG edge from Bi to Bj if Bj can immediately follow Bi in some execution, i.e.,
    - There is conditional or unconditional goto from last statement of Bi to first statement of Bj or
    - Bj immediately follows Bi in the order of the program and Bi does not end in an unconditional goto statement
  - Label edges that represent conditional transfers of control

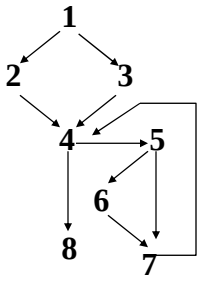
What is the complexity of the algorithm, given n statements in the program?

Control Flow Analysis: Terminology

- CFG = <N, E>, rooted directed graph
  - N = set of nodes
  - E $\subseteq$ N x N = set of edges, some labeled
  - *entry* $\in$ N, *exit* $\in$ N
- Successors/predecessors of a basic block
- Branch node is a node with two or more outgoing edges
- Join node is a node with two or more outgoing edges

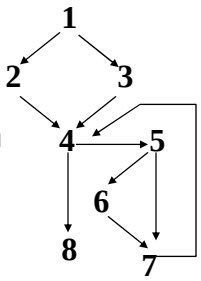
Applications of Control Flow

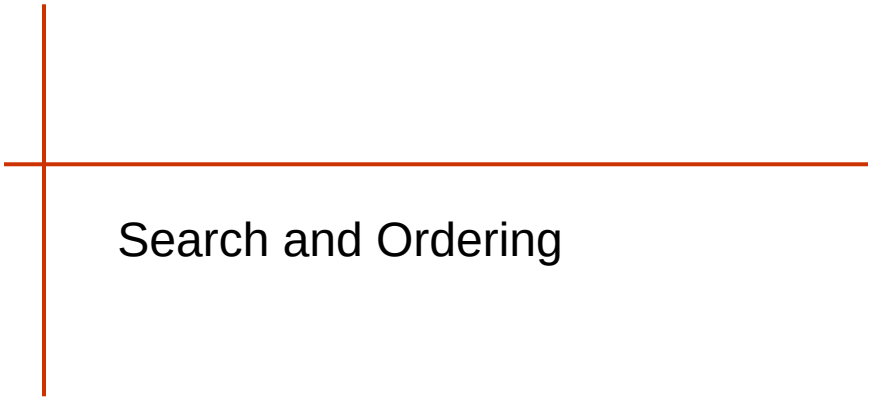
- Program understanding
  - program structure and flow is explicit
- Complexity
  - Cyclomatic (McCabe's)
  - Computed in several ways:
    - Edges – nodes +2
    - Number of regions in CFG
    - Number of decision statements + 1 (if structured)
  - Indication of number of test case needed; indication of difficulty of maintaining



Applications of Control Flow

- Testing: branch, path, basis path
  - Branch: must test 1→2, 1→ 3, 4→5, 4→8, 5→6, 5→7
  - Path: infinite number because of loop
  - Basis path: set of paths such that each path executes at least one more edge (cyclomatic complexity gives max necessary); example: 1,2,4,8; 1,3,4,5,6,7,4,8





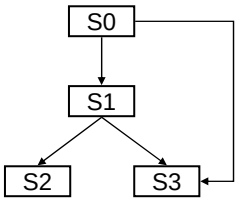
Search and Ordering

Some Useful Concepts

- Depth-First Search (DFS): Visits descendants before visiting siblings
- Depth-first spanning tree: All nodes, only edges traversed in the DFS
- Depth-first presentation: spanning tree + remaining edges (marked)
  - Forward edges: node → direct descendant in the tree
  - Back edges: node → ancestor in the tree
  - Cross edges: node → neither ancestor nor descendant in the tree

Some Useful Concepts

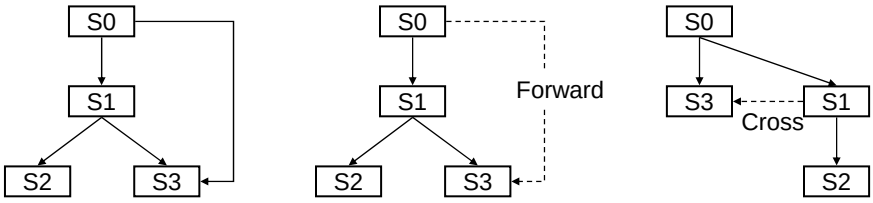
- Depth-First Search (DFS): Visits descendants before visiting siblings
- Depth-first spanning tree: All nodes, only edges traversed in the DFS
- Depth-first presentation: spanning tree + remaining edges (marked)
  - Forward edges: node → direct descendant in the tree
  - Back edges: node → ancestor in the tree
  - Cross edges: node → neither ancestor nor descendant in the tree



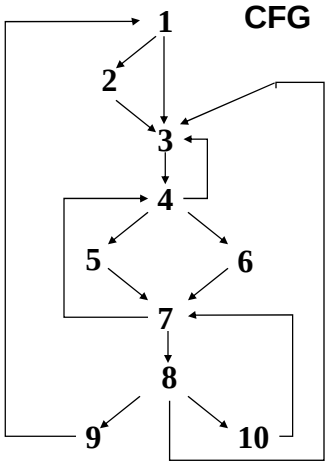
What are some depth-first spanning trees and presentations for this CFG?

Some Useful Concepts

- Depth-First Search (DFS): Visits descendants before visiting siblings
- Depth-first spanning tree: All nodes, only edges traversed in the DFS
- Depth-first presentation: spanning tree + remaining edges (marked)
  - Forward edges: node → direct descendant in the tree
  - Back edges: node → ancestor in the tree
  - Cross edges: node → neither ancestor nor descendant in the tree

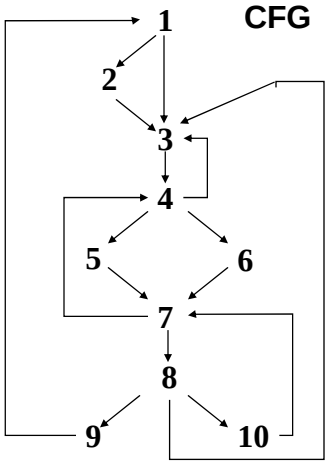


Search and Ordering (depth-first)



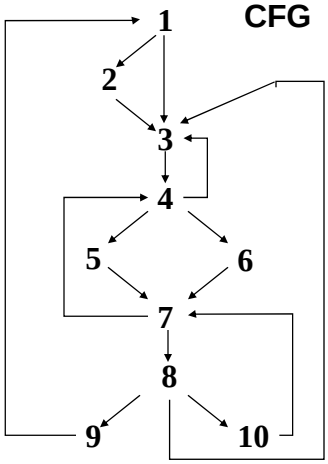
Show one depth-first presentation for the CFG?

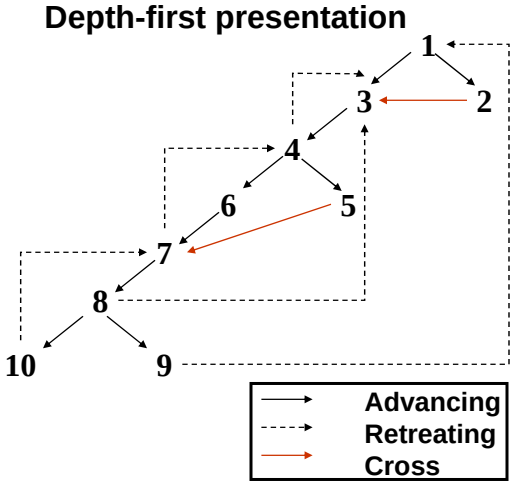
Search and Ordering (depth-first)



- One DFS of CFG is 1→3→4→6→7→8→10, back to 8, →9, back to 8, 7, 6, 4, →5, back to 4, 1, 3, →2, back to 1
- Depth-first ordering of nodes is the reverse of the order in which nodes are visited in the DFS
- For the DFS, nodes are visited 1,3,4,6,7,8,10,8,9,8,7,6,4,5,4,1,3,2,1
- Depth-first ordering is 1,2,3,4,5,6,7,8,9,10

Search and Ordering (depth-first)

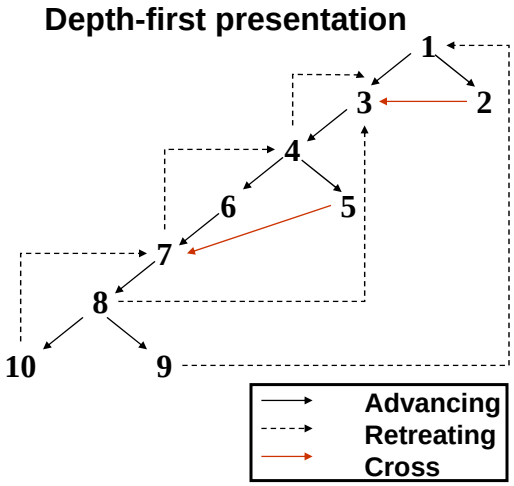




Search and Ordering (depth-first)

Given a depth-first presentation of a CFG, the depth of the CFG is the greatest number of retreating edges on any cycle-free path

What is the depth of this CFG, given this presentation?

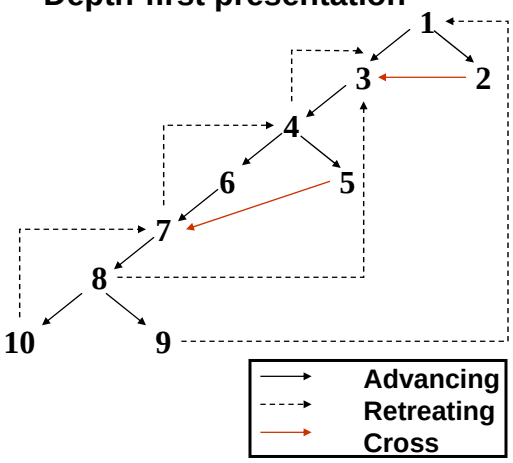


Search and Ordering (depth-first)

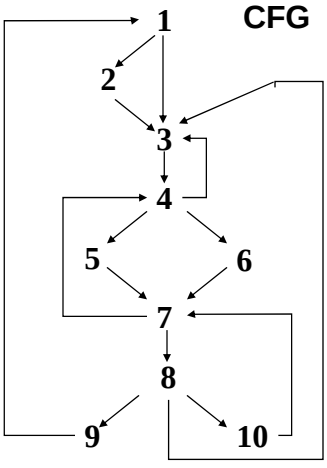
Given a depth-first presentation of a CFG, the depth of the CFG is the greatest number of retreating edges on any cycle-free path

There is a path 10→7→4→3 with three retreating edges; thus the depth of the CFG is 3

Depth-first presentation



Search and Ordering (depth-first)



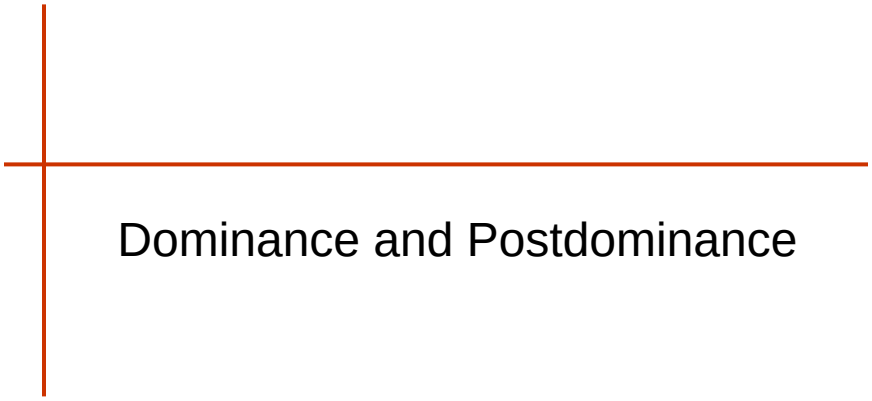
For Thursday:

Is there a depth-first presentation with depth greater than 3?

Some Useful Concepts

- Preorder traversal (reverse postorder): Traversal of the depth-first spanning tree in which each node is processed before its descendants
- Postorder traversal: Traversal of the depth-first spanning tree in which each node is processed after its descendants
- Breadth-First Search (BFS): All of a node's immediate descendants are processed before any of their unprocessed descendants
- Breadth-first order: Order imposed by a BFS

Search and ordering algorithms are review, and you are expected to know them.



Dominance and Postdominance

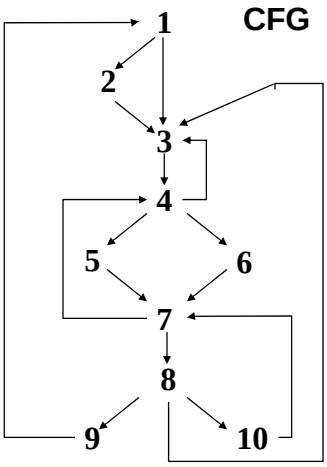
Dominators, Postdominators

Given a CFG with nodes D and N, D dominates N if every path from the initial node to N goes through D

Properties of dominance

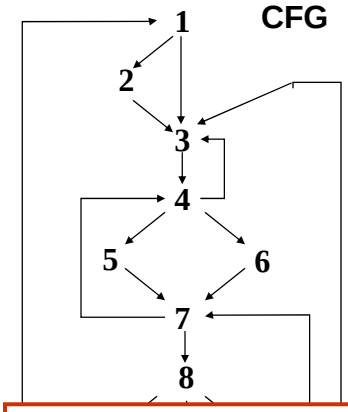
1. Every node dominates itself
2. Initial node dominates all others

Dominators, Postdominators (example)



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

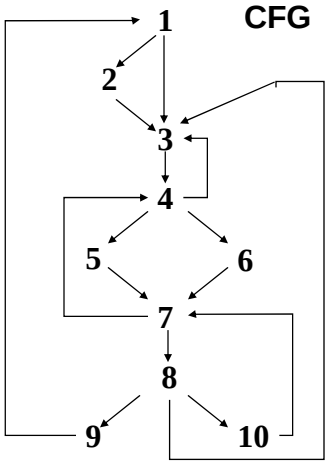
Dominators, Postdominators (example)



What are the dominators for nodes in the CFG?

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

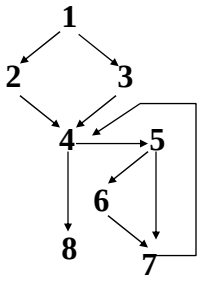
Dominators, Postdominators (example)



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

Dominators, Postdominators (dominator properties)

CFG



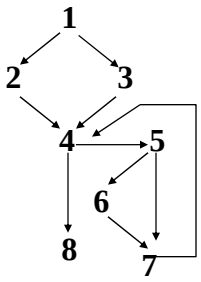
a dom *b* iff

- *a* = *b* or
- *a* is the unique immediate predecessor of *b* or
- *b* has more than one predecessor and for all immediate predecessors *c* of *b*, *a* dominates *c*

dom is reflexive, transitive, and antisymmetric

Dominators, Postdominators (dominator properties)

CFG



a dom *b* iff

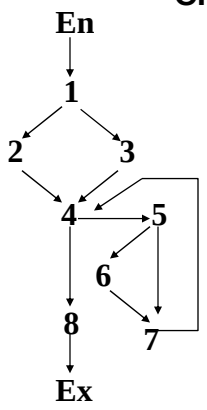
- *a* = *b* or
- *a* is the unique immediate predecessor of *b* or
- *b* has more than one predecessor and for all immediate predecessors *c* of *b*, *a* dominates *c*

dom is reflexive, transitive, and antisymmetric

What these properties mean for the *dom* relation?

Dominators, Postdominators (dominator algorithm)

CFG

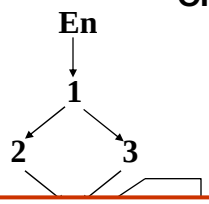


Intuition for algorithm

- N is set of nodes in CFG with En, Ex
- initialize *domin*(En) to {En}, *change* to false
- Initialize *domin*(n) to N for all n != En
- iterate over all n (except En) until no change in *domin* sets
  - assign N to T
  - compute *domin*(n) by first taking the intersection of T and *domin*(p), forall p, a predecessor of n
  - then add n to T (this is new *domin*(n))
  - If T != *domin*(n), a change has occurred
    - assign T to *domin*(n)
    - *change* is true

Dominators, Postdominators (dominator algorithm)

CFG



Intuition for algorithm

- N is set of nodes in CFG with En, Ex
- initialize *domin*(En) to {En}, *change* to false
- Initialize *domin*(n) to N for all n != En
- iterate over all n (except En) until no change in *domin* sets
  - gn N to T
  - bute *domin*(n) by first taking the section of T and *domin*(p), forall p, a ecessor of n
  - add n to T (this is new *domin*(n))
  - = *domin*(n), a change has occurred
    - ssign T to *domin*(n)
    - *hange* is true

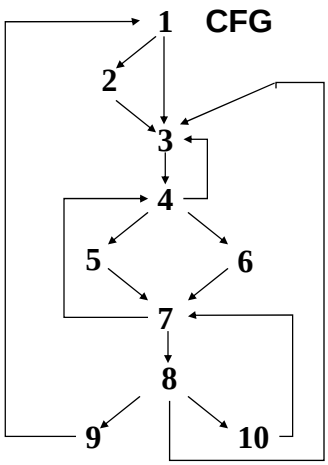
For Thursday:

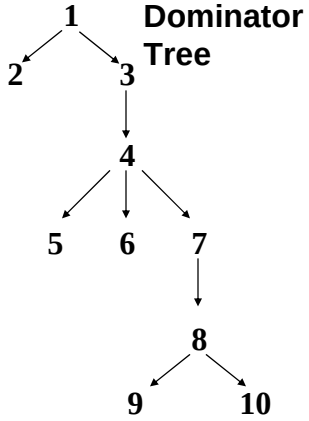
Show iterations of the algorithm over the nodes in the CFG until the result converges?

Dominators, Postdominators (dominator tree)

- In a dominator tree
  - The initial node *n* is the root of the CFG
  - 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, Postdominators (dominator tree)



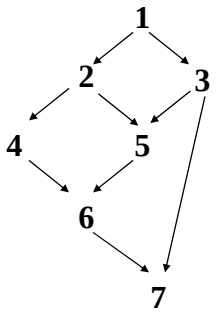


Dominators, Postdominators

Given a CFG with nodes PD and N, PD postdominates N if every path from N to the final nodes goes through PD

Dominators, Postdominators (example)

CFG



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

Dominators, Postdominators (dominator tree)

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

Dominators, Postdominators (dominator tree)