Lecture 23

CIS 341: COMPILERS

Announcements

- Projects 6
  - Available later today
  - Due Next Thursday

- Project 7
  - Available early next week (Monday)
  - Due: May 5th

- Final Exam:
  - Tuesday, April 30th noon-2:00 pm
  - Moore 216

Dataflow Analysis: Summary

- Many dataflow analyses fit into a common framework.
- Key idea: *Iterative solution* of a system of equations over a *lattice* of constraints.
  - Iteration terminates if flow functions are monotonic.
  - Solution is equivalent to meet-over-paths answer if the flow functions distribute over meet ($\sqcap$).

- Dataflow analyses as presented work for an "imperative" intermediate representation.
  - The values of temporary variables are updated ("mutated") during evaluation.
  - Such mutation complicates calculations
  - SSA = "Single Static Assignment" eliminates this problem, by introducing more temporaries – each one assigned to only once.
  - Next up: Converting to SSA, finding loops and dominators in CFGs

LOOPS AND DOMINATORS

Loops in Control-flow Graphs

- Taking into account loops is important for optimizations.
  - The 90/10 rule applies, so optimizing loop bodies is important

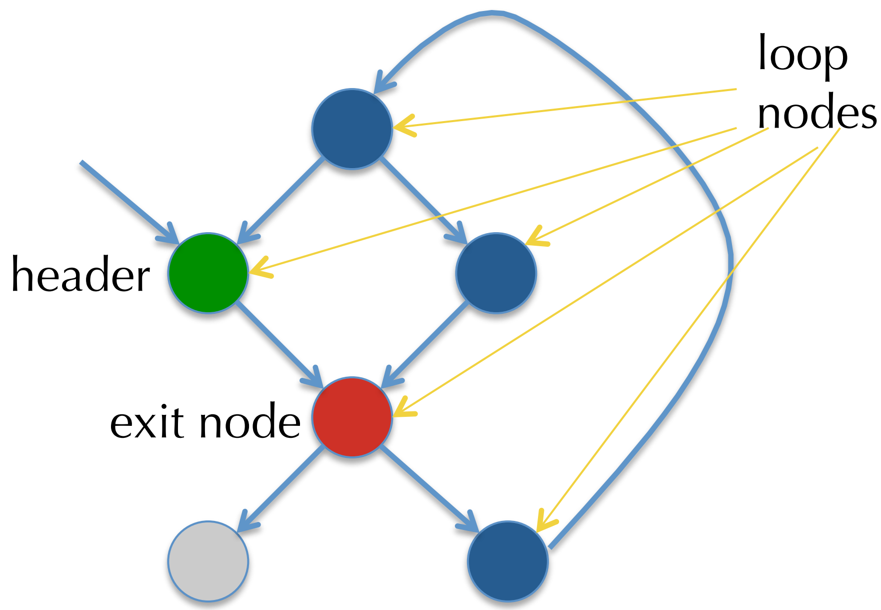
- Should we apply loop optimizations at the AST level or at a lower representation?
  - Loop optimizations benefit from other IR-level optimizations and vice-versa, so it is good to interleave them.

- Loops may be hard to recognize at the quadruple IR level.
  - Many kinds of loops: while, do/while, for, continue, goto…

- Problem: *How do we identify loops in the control-flow graph?*

Definition of a Loop

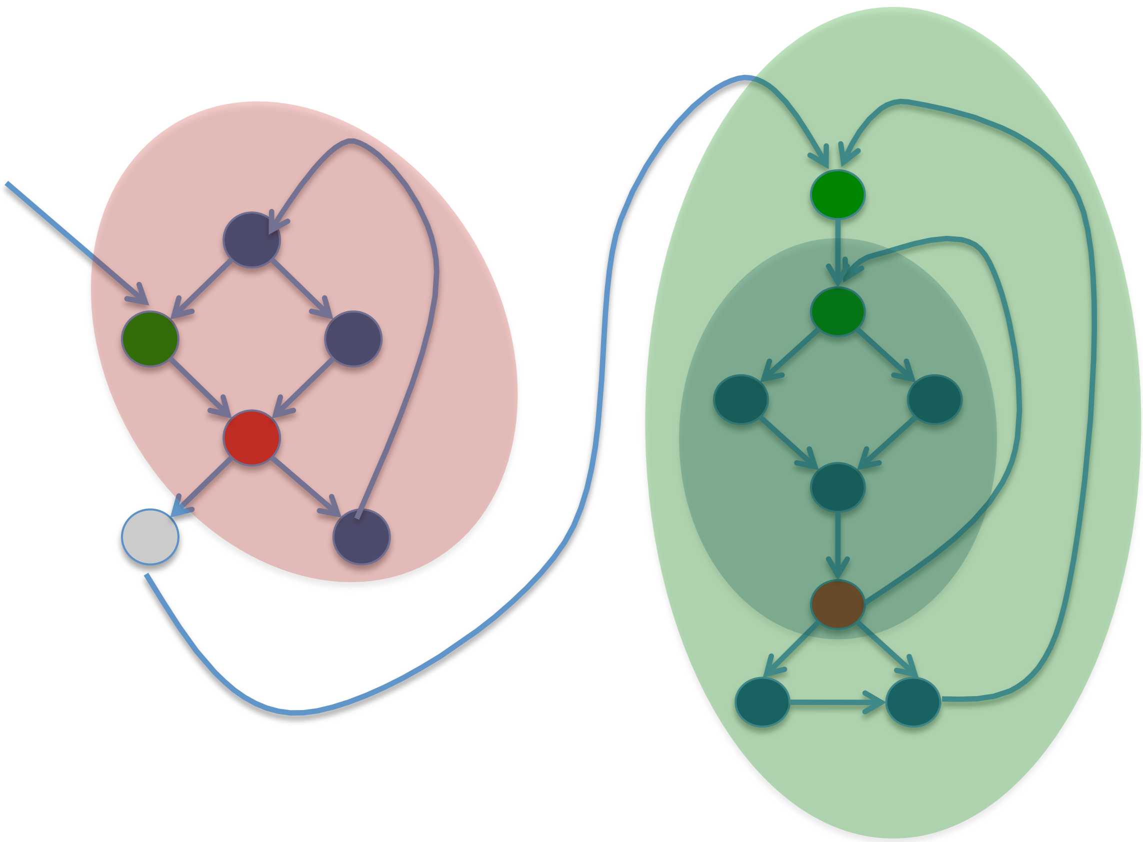
- A *loop* is a set of nodes in the control flow graph.
  - One distinguished entry point called the *header*
- Every node is reachable from the header & the header is reachable from every node.
  - A loop is a *strongly connected component*



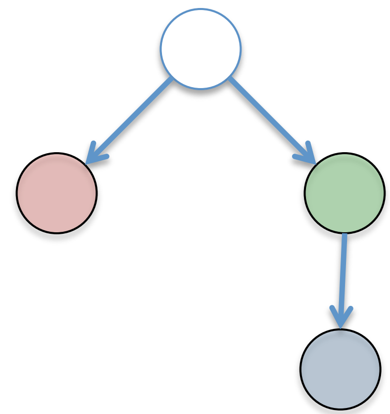
- No edges enter the loop except to the header
- Nodes with outgoing edges are called loop exit nodes

Nested Loops

- Control-flow graphs may contain many loops
- Loops may contain other loops:



Control Tree:



The control tree depicts the nesting structure of the program.

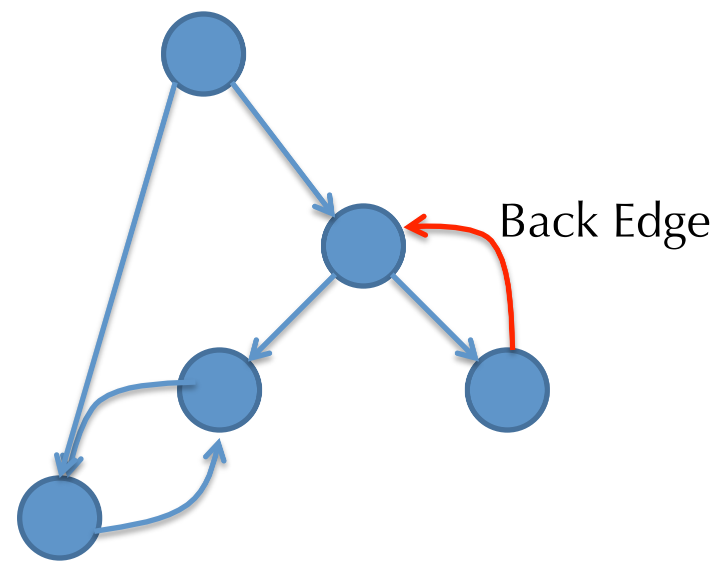
Control-flow Analysis

- Goal: Identify the loops and nesting structure of the CFG.

- Control flow analysis is based on the idea of *dominators*:
- Node A *dominates* node B if the only way to reach B from the start node is through node A.

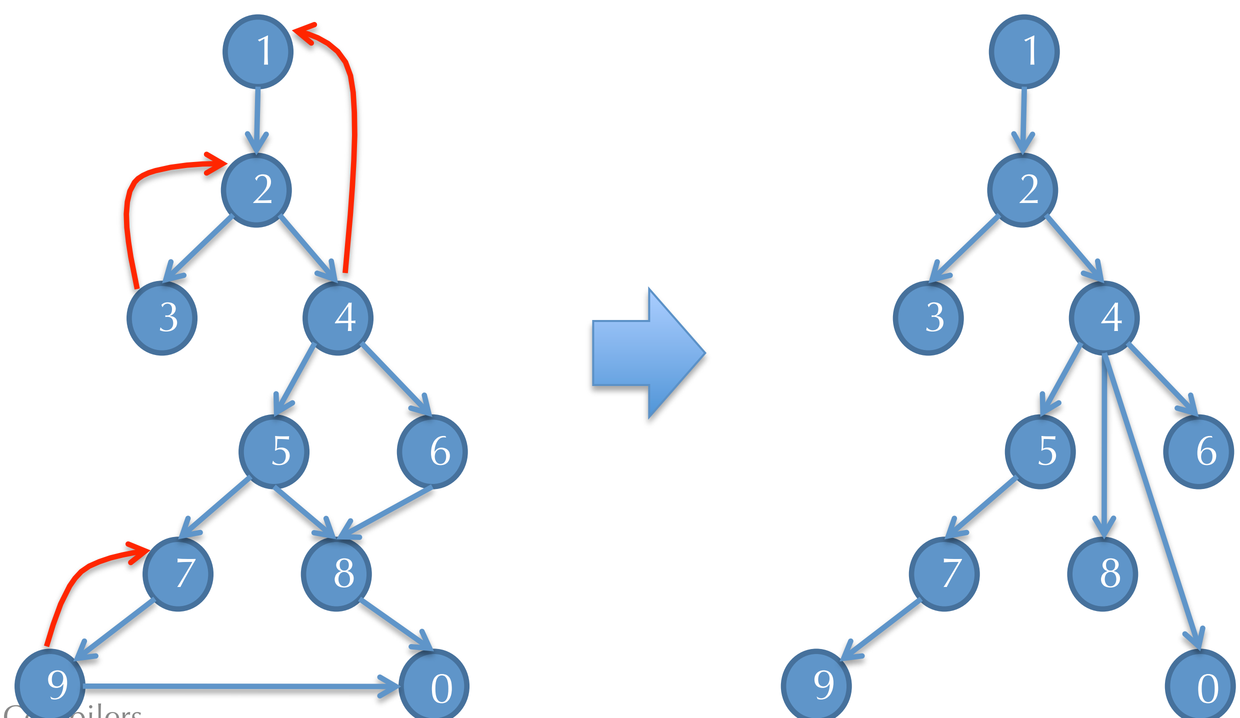
- An edge in the graph is a *back edge* if the target node dominates the source node.

- A loop contains at least one back edge.



Dominator Trees

- Domination is transitive:
  - if A dominates B and B dominates C then A dominates C
- Domination is anti-symmetric:
  - if A dominates B and B dominates A then A = B
- Every flow graph has a dominator tree
  - The Hasse diagram of the dominates relation



Dominator Dataflow Analysis

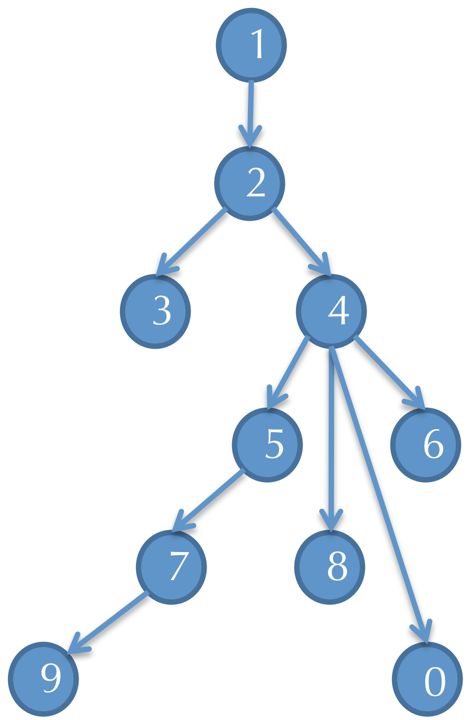
- We can define Dom[n] as a forward dataflow analysis.
  - Using the framework we saw earlier: Dom[n] = out[n] where:
- "A node B is dominated by another node A if A dominates *all* of the predecessors of B."
  - $in[n] := \bigcap_{n' \in pred[n]} out[n']$
- "Every node dominates itself."
  - $out[n] := in[n] \cup \{n\}$

- Formally: $\mathcal{L}$ = set of nodes ordered by $\subseteq$
  - $\top = \{\text{all nodes}\}$
  - $F_n(x) = x \cup \{n\}$
  - $\sqcap$ is $\cap$
- Easy to show monotonicity and that F_n distributes over meet.
  - So algorithm terminates and is MOP

Improving the Algorithm

- Dom[b] contains just those nodes along the path in the dominator tree from the root to b:
  - e.g. Dom[8] = {1,2,4,8}, Dom[7] = {1,2,4,5,7}
  - There is a lot of sharing among the nodes

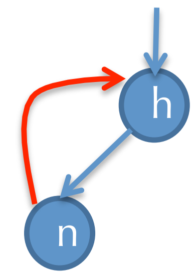
- More efficient way to represent Dom sets is to store the dominator *tree*.
  - doms[b] = immediate dominator of b
  - doms[8] = 4, doms[7] = 5
- To compute Dom[b] walk through doms[b]
- Need to efficiently compute intersections of Dom[a] and Dom[b]
  - Traverse up tree, looking for least common ancestor:
  - Dom[8] ∩ Dom[7] = Dom[4]



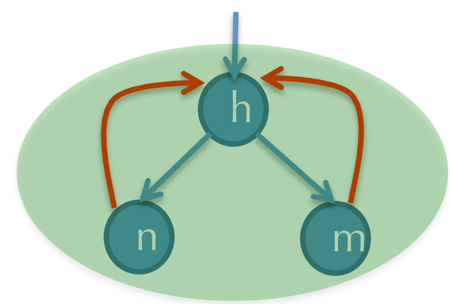
- See: "A Simple, Fast Dominance Algorithm" Cooper, Harvey, and Kennedy

Completing Control-flow Analysis

- Dominator analysis identifies *back edges*:
  - Edge n → h where h dominates n
- Each back edge has a *natural loop*:
  - h is the header
  - All nodes reachable from h that also reach n without going through h

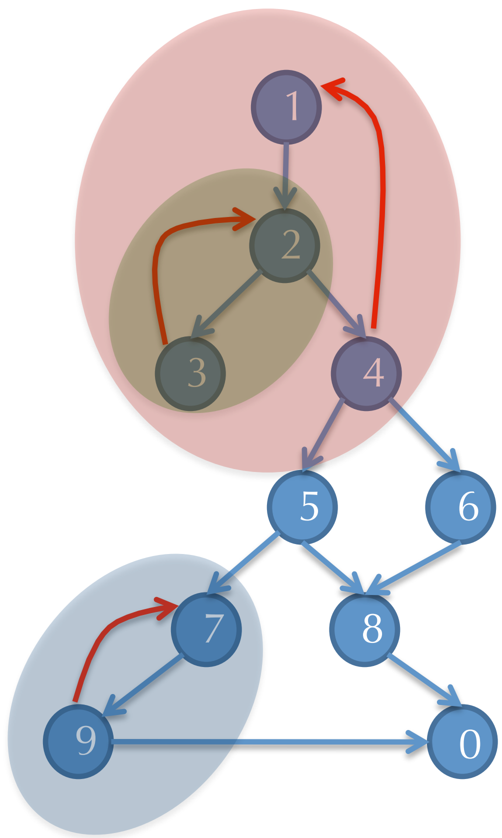


- For each back edge n → h, find the natural loop:
  - {n′ | n is reachable from n′ in G – {h}} ∪ {h}
- Two loops may share the same header: merge them



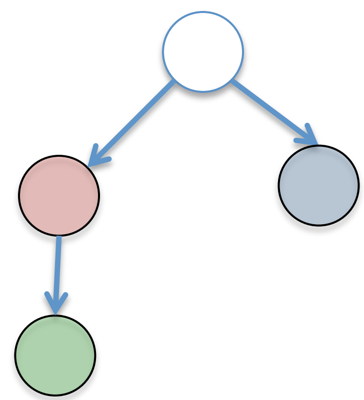
- Nesting structure of loops is determined by set inclusion
  - Can be used to build the control tree

Example Natural Loops



Natural Loops

Control Tree:



The control tree depicts the nesting structure of the program.

Uses of Control-flow Information

- Loop nesting depth plays an important role in optimization heuristics.
  - Deeply nested loops pay off the most for optimization.

- Need to know loop headers / back edges for doing
  - loop invariant code motion
  - loop unrolling

- Dominance information also plays a role in converting to SSA form
  - Used internally by LLVM to do register allocation.

Phi nodes
Alloc "promotion"
Register allocation

REVISITING SSA

Single Static Assignment (SSA)

- LLVM IR names (via `%uids`) *all* intermediate values computed by the program.
- It makes the order of evaluation explicit.
- Each `%uid` is assigned to only once
  - Contrast with the mutable quadruple form
  - Note that dataflow analyses had these kill[n] sets because of updates to variables…
- Naïve implementation of phase2: map `%uids` to stack slots
- Better implementation: map `%uids` to registers (as much as possible)

- Question: How do we convert a source program to make maximal use of `%uids`, rather than alloca-created storage?
  - two problems: control flow & location in memory

Alloca vs. %UID

- Current compilation strategy:

```
int x = 3;
int y = 0;
x = x + 1;
y = x + 2;
```

```
%x = alloca i32
%y = alloca i32
store i32* %x, 3
store i32* %y, 0
%x1 = load %i32* %x
%tmp1 = add i32 %x1, 1
store i32* %x, %tmp1
%x2 = load %i32* %x
%tmp2 = add i32 %x2, 2
store i32* %y, %tmp2
```

- Directly map source variables into `%uids`?

```
int x = 3;
int y = 0;
x = x + 1;
y = x + 2;
```

```
int x1 = 3;
int y1 = 0;
x2 = x1 + 1;
y2 = x2 + 2;
```

```
%x1 = add i32 3, 0
%y1 = add i32 0, 0
%x2 = add i32 %x1, 1
%y2 = add i32 %x2, 2
```

- Does this always work?

What about If-then-else?

- How do we translate this into SSA?

```
int y = …
int x = …
int z = …
if (p) {
  x = y + 1;
} else {
  x = y * 2;
}
z = x + 3;
```

```
entry:
  %y1 = …
  %x1 = …
  %z1 = …
  %p = icmp …
  br i1 %p, label %then, label %else
then:
  %x2 = add i32 %y1, 1
  br label %merge
else:
  %x3 = mult i32 %y1, 2
merge:
  %z2 = %add i32 ???, 3
```

- What do we put for `???`

Phi Functions

- Solution: ϕ functions
  - Fictitious operator, used only for analysis
    - implemented by Mov at x86 level
  - Chooses among different versions of a variable based on the path by which control enters the phi node.
    %uid = phi <ty> v_1, <$label_1$>, … , v_n, <$label_n$>

```
int y = …
int x = …
int z = …
if (p) {
  x = y + 1;
} else {
  x = y * 2;
}
z = x + 3;
```

```
entry:
  %y1 = …
  %x1 = …
  %z1 = …
  %p = icmp …
  br i1 %p, label %then, label %else
then:
  %x2 = add i32 %y1, 1
  br label %merge
else:
  %x3 = mult i32 %y1, 2
merge:
  %x4 = phi i32 %x2, %then, %x3, %else
  %z2 = %add i32 %x4, 3
```

Phi Nodes and Loops

- Importantly, the `%uids` on the right-hand side of a phi node can be defined "later" in the control-flow graph.
  - Means that `%uids` can hold values "around a loop"
  - Scope of `%uids` is defined by dominance (discussed soon)

```
entry:
  %y1 = …
  %x1 = …
  br label %body

body:
  %x2 = phi i32 %x1, %entry, %x3, %body
  %x3 = add i32 %x2, %y1
  %p = icmp slt i32, %x3, 10
  br i1 %p, label %body, label %after

after:
  …
```

Alloca Promotion

- Not all source variables can be allocated to registers
  - If the address of the variable is taken (as permitted in C, for example)
  - If the address of the variable "escapes" (by being passed to a function)
- An alloca instruction is called promotable if neither of the two conditions above holds

```
entry:
  %x = alloca i32          // %x cannot be promoted
  %y = call malloc(i32 4)
  store i32** %y, %x       // store the pointer into the heap
```

```
entry:
  %x = alloca i32          // %x cannot be promoted
  %y = call foo(i32* %x) // foo may store the pointer into the heap
```

- Happily, most local variables declared in source programs are promotable
  - That means they can be register allocated

Converting to SSA: Overview

- Start with the ordinary control flow graph that uses allocas
  - Identify "promotable" allocas
- Compute dominator tree information
- Calculate def/use information for each such allocated variable
- Insert ϕ functions for each variable at necessary "join points"
- Replace loads/stores to alloc′ed variables with freshly-generated %uids
- Eliminate the now unneeded load/store/alloca instructions.

Where to Place ϕ functions?

- Need to calculate the "Dominance Frontier"

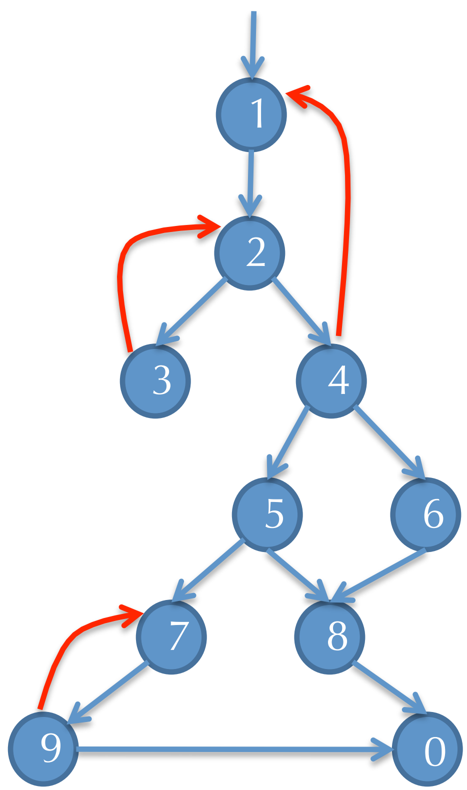
- Node A *strictly dominates* node B if A dominates B and A ≠ B.

- The *dominance frontier* of a node B is the set of all CFG nodes y such that B dominates a predecessor of y but does not strictly dominate y

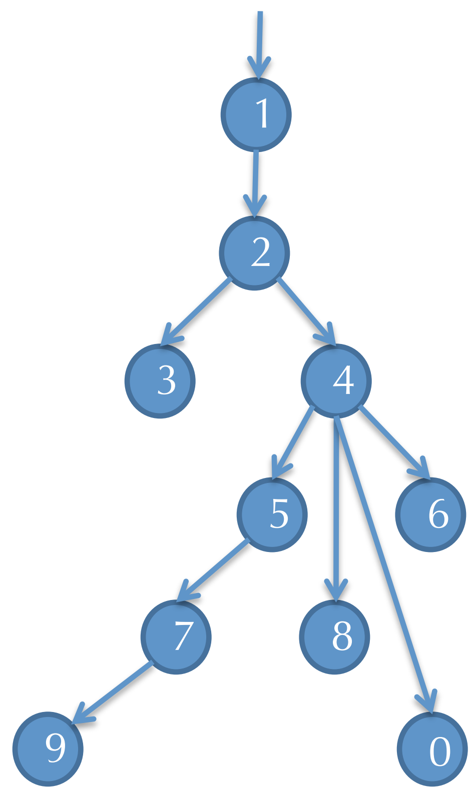
- Write DF[n] for the dominance frontier of node n.

Dominance Frontiers

- Example of a dominance frontier calculation results
- DF[1] = {}, DF[2] = {2}, DF[3] = {2}, DF[4] = {1}, DF[5] = {8,0}, DF[6] = {8}, DF[7] = {0}, DF[8] = {0}, DF[9] = {7,0}, DF[0] = {}



Control-flow Graph



Dominator Tree

Algorithm For Computing DF[n]

- Assume that doms[n] stores the dominator tree (so that doms[n] is the *immediate dominator* of n in the tree)

```
for all nodes b
    if #(pred[b]) ≥ 2
     for each p ∈pred[b]
         runner := p
         while (runner ≠ doms[b])
             DF[runner] := DF[runner] ∪ {b}
             runner := doms[runner]
```

Insert ϕ at Join Points

- Lift the DF[n] to a set of nodes N in the obvious way:
  $DF[N] = \bigcup_{n \in N} DF[n]$
- Suppose that at variable x is defined at a set of nodes N.
- $DF_0[N] = N$
  $DF_i[N] = DF[DF_{i-1}[N] \cup N]$
- Let J[N] be the *least fixed point* of the sequence:
  $DF_0[N] \subseteq DF_1[N] \subseteq DF_2[N] \subseteq DF_3[N] \subseteq \ldots$
  - That is, $J[N] = DF_k[N]$ for some k such that $DF_k[N] = DF_{k+1}[N]$
- J[N] is called the "join points" for the set N
- We insert ϕ functions for the variable x at each such join point.
  - $x = \phi(x, x, \ldots, x)$; (one "x" argument for each predecessor of the node)
  - In practice, J[N] is never directly computed, instead you use a worklist algorithm that keeps adding nodes for $DF_k[N]$ until there are no changes.