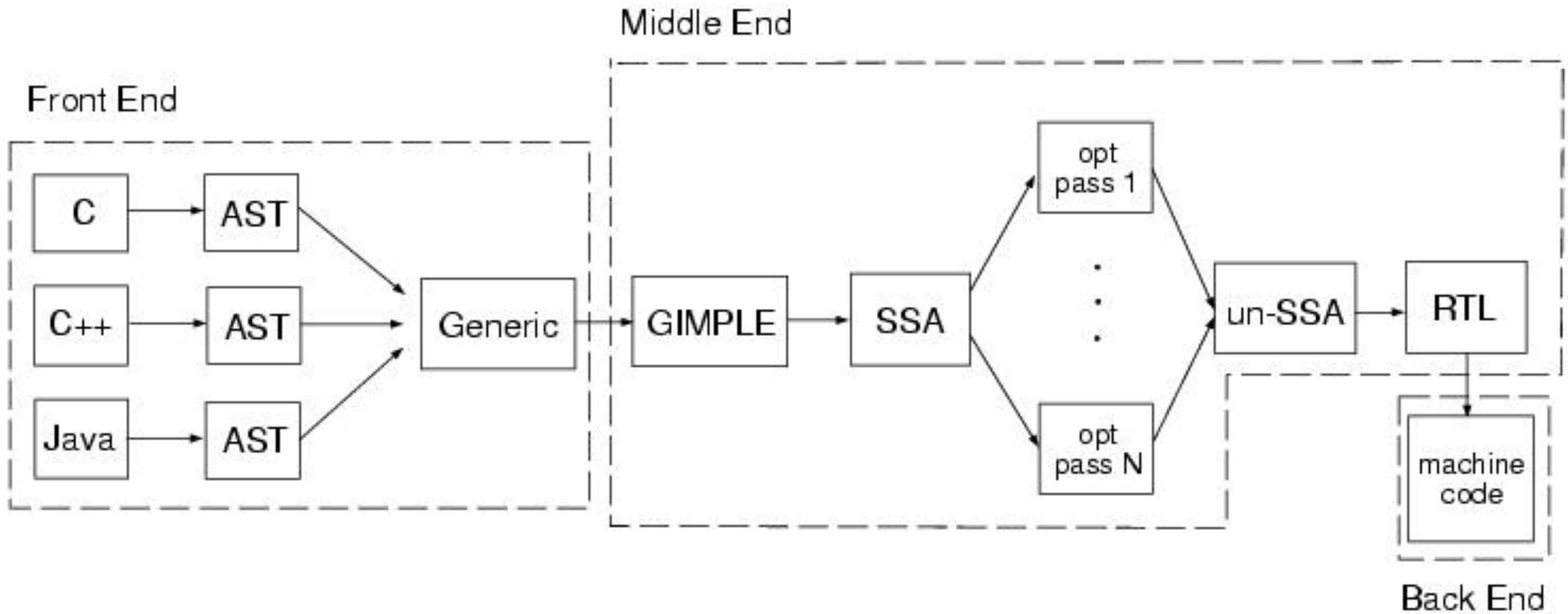
An Overview of GCC Architecture (source: wikipedia)



Control-Flow Analysis and Loop Detection

Reminders

- Assignment 1 resubmit due on D2L by Thursday 11:59pm

Last time

- Lattice-theoretic framework for data-flow analysis

Today

- Finishing up lattice-theoretic framework for data-flow analysis
- Control-flow analysis
- Loops
- Identifying loops using dominators

Tuples of Lattices

Problem

- Simple analyses may require complex lattices (*e.g.,* Reaching constants)

Possible Solutions for reaching constants

- Tuple of lattices, (variable, constant) tuples
- Tuple of lattices, one entry in tuple per variable

$\mathbf{L = (V, \sqcap) \;\equiv\; (L_i = (V_i, \sqcap_i))^N}$

- $V = V_1 \times V_2 \times \ldots \times V_N$
- Meet ($\sqcap$): point-wise application of $\sqcap_T$
- $(\ldots, v_i, \ldots) \sqsubseteq (\ldots, u_i, \ldots) \equiv v_i \sqsubseteq u_i, \forall i$
- Top ($\top$): tuple of tops $(\top_i)^N$
- Bottom ($\bot$): tuple of bottoms $(\bot_i)^N$
- Height (L) = height(L_1) * height(L_2) * … * height(L_N)

Equivalence of Power Set Lattices and Tuple of two-point lattices (bitvectors)

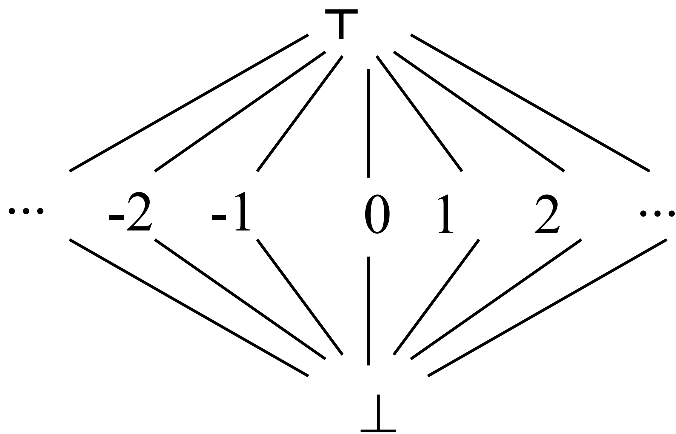
Tuples of Lattices Example

Reaching constants (previously)

- P = v×c, for variables v & constants c
- V: 2^P

Alternatively

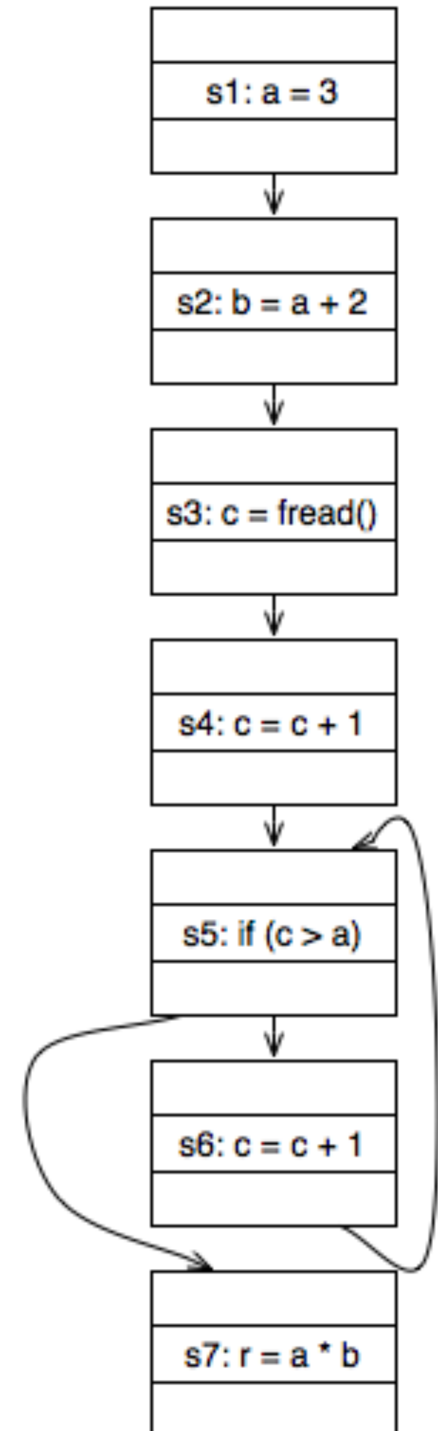
- V = c ∪ {⊤, ⊥}



The whole problem is a tuple of lattices, one for each variable

Tuple of Lattices example

For reaching constants, how big is the tuple with entry per variable for this example?



Reaching Constants, Various Ways to do Tuple of Lattices

Reaching Constants

- V: $2^{(C, C, \ldots, C)}$
- $\sqcap$: $\cap$
- $\sqsubseteq$: $\subseteq$
- Top($\top$): U
- Bottom ($\bot$): $\emptyset$
- F: . . .

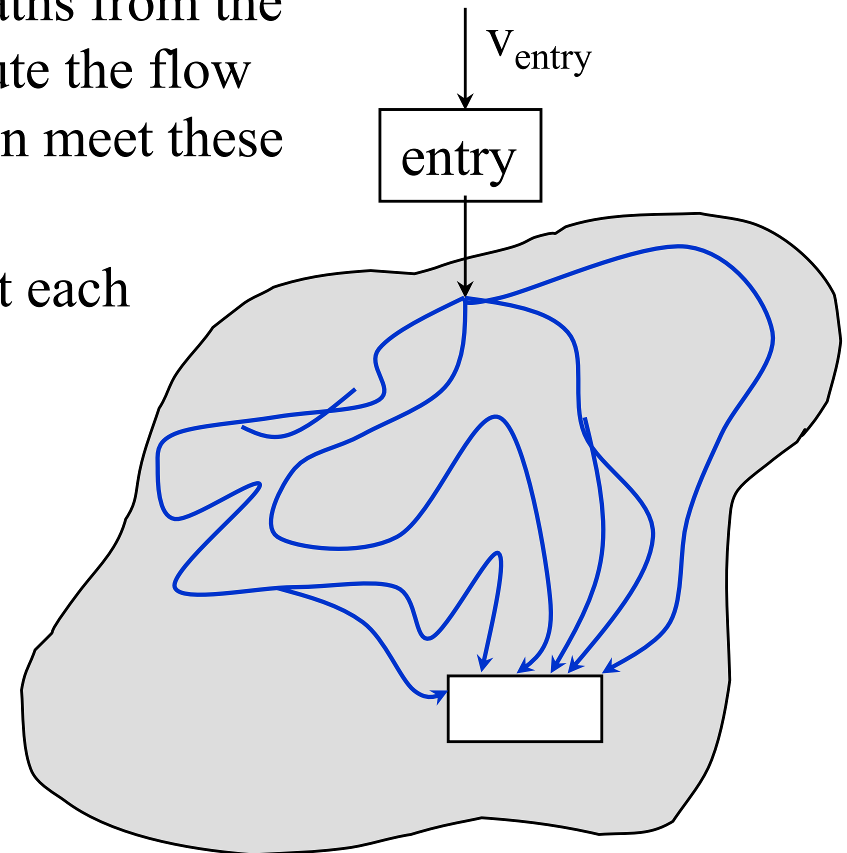
Reaching Constants

- V: $2^{v \times c}$, variables v and constants c
- $\sqcap$: $\cap$
- $\sqsubseteq$: $\subseteq$
- Top($\top$): U
- Bottom ($\bot$): $\emptyset$
- F: . . .

Solving Data-Flow Analyses

Goal

- For a forward problem, consider all paths from the entry to a given program point, compute the flow values at the end of each path, and then meet these values together
- Meet-over-all-paths (MOP) solution at each program point
- $\sqcap_{\text{all paths n1, n2, ..., ni}} (f_{ni}(\ldots f_{n2}(f_{n1}(v_{entry}))))$



Solving Data-Flow Analyses (cont)

Problems

- Loops result in an infinite number of paths
- Statements following merge must be analyzed for all preceding paths
  - Exponential blow-up

Solution

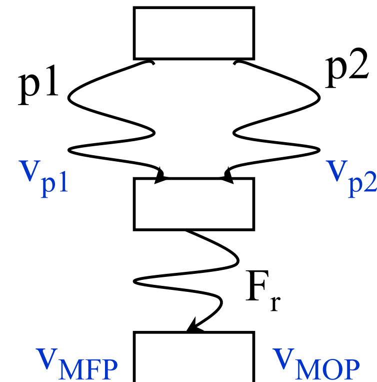
- Compute meets early (at merge points) rather than at the end
- Maximum fixed-point (MFP)

Questions

- Is this correct?
- Is this efficient?
- Is this accurate?

Correctness

"Is v_{MFP} correct?" ≡ "Is $v_{MFP} \sqsubseteq v_{MOP}$?"



Look at Merges

$v_{MOP} = F_r(v_{p1}) \sqcap F_r(v_{p2})$

$v_{MFP} = F_r(v_{p1} \sqcap v_{p2})$

$v_{MFP} \sqsubseteq v_{MOP} \equiv F_r(v_{p1} \sqcap v_{p2}) \sqsubseteq F_r(v_{p1}) \sqcap F_r(v_{p2})$

Observation

$\forall x,y \in V$

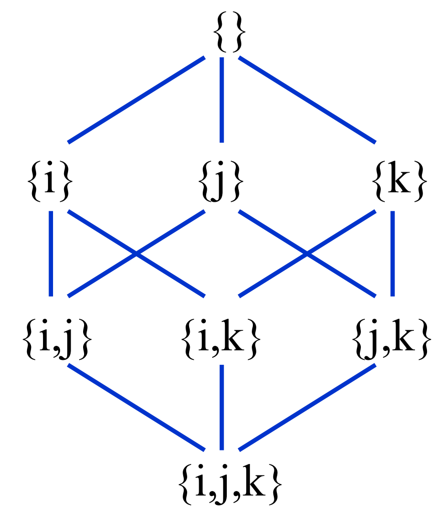
$$f(x \sqcap y) \sqsubseteq f(x) \sqcap f(y) \quad \Leftrightarrow \quad x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y)$$

$\therefore$ v_{MFP} correct when F_r (really, the flow functions) are monotonic

Monotonicity

Monotonicity: $(\forall x,y \in V)[x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y)]$

- If the flow function f is applied to two members of V, the result of applying f to the "lesser" of the two members will be under the result of applying f to the "greater" of the two
- Giving a flow function more conservative inputs leads to more conservative outputs (never more optimistic outputs)



Why else is monotonicity important?

For monotonic F over domain V

- The maximum number of times F can be applied to self w/o reaching a fixed point is height(V) – 1
- IDFA is guaranteed to terminate if the flow functions are monotonic and the lattice has finite height

Efficiency

Parameters

- n: Number of nodes in the CFG
- k: Height of lattice
- t: Time to execute one flow function

Complexity

- O(nkt)

Example

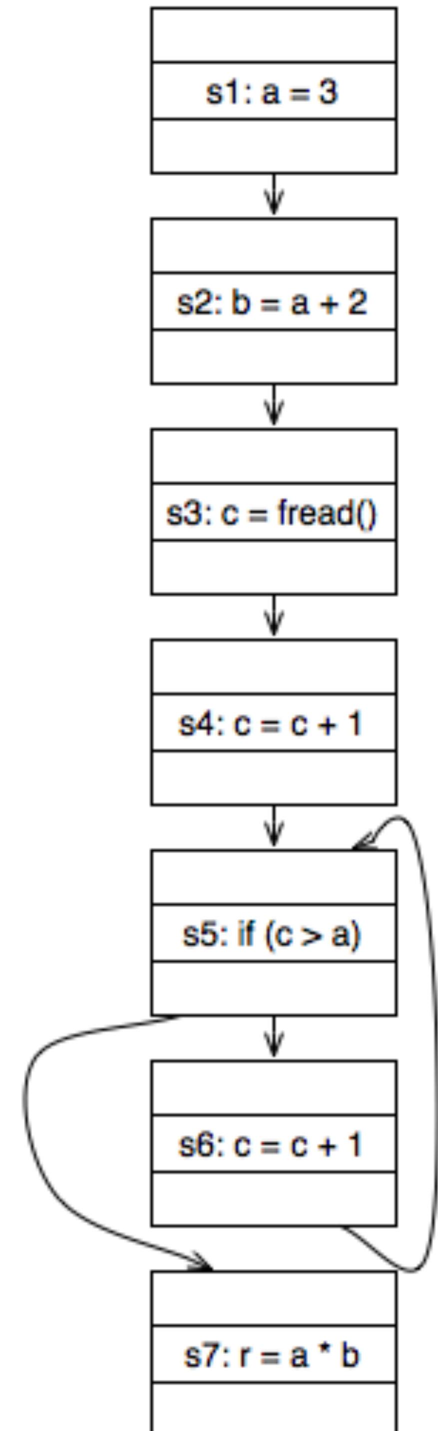
- Reaching definitions?

Reaching Defs Example

What is the height of the lattice?

How many passes over the nodes are necessary?

What if we visit the nodes in a non-optimal order?



Accuracy

Distributivity

- $f(u \sqcap v) = f(u) \sqcap f(v)$
- $v_{MFP} \sqsubseteq v_{MOP} \equiv F_r(v_{p1} \sqcap v_{p2}) \sqsubseteq F_r(v_{p1}) \sqcap F_r(v_{p2})$
- If the flow functions are distributive, MFP = MOP

Examples

- Reaching definitions?
- Reaching constants?

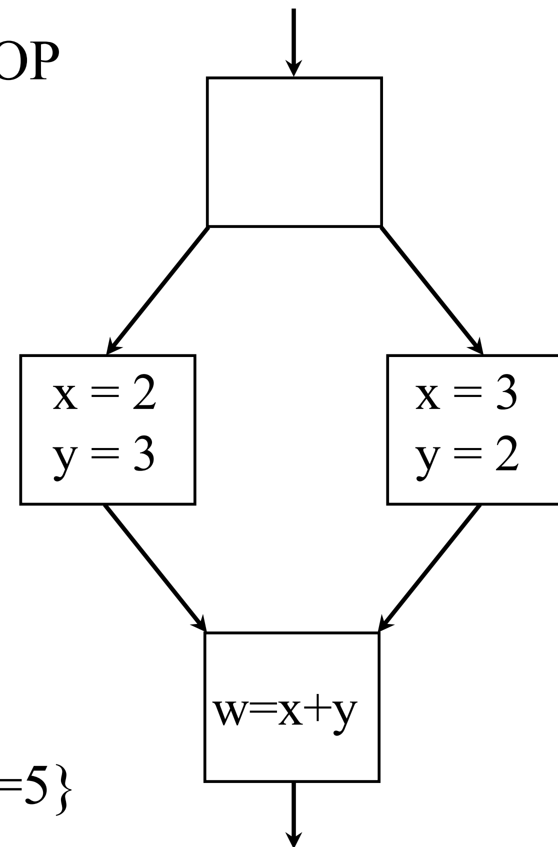
$f(u \sqcap v) = f(\{x=2,y=3\} \sqcap \{x=3,y=2\})$

$= f(\varnothing) = \varnothing$

$f(u) \sqcap f(v) = f(\{x=2,y=3\}) \sqcap f(\{x=3,y=2\})$

$= [\{x=2,y=3,w=5\} \sqcap \{x=2,y=2,w=5\}] = \{w=5\}$

$\Rightarrow$ MFP $\neq$ MOP



Concepts

Lattices

- Conservative approximation
- Optimistic (initial guess)
- Data-flow analysis frameworks
- Tuples of lattices

Lattice Theoretic framework for common subexpression elimination

Data-flow analysis

- Fixed point
- Meet-over-all-paths (MOP)
- Maximum fixed point (MFP)
- Legal/safe/correct (monotonic)
- Efficient
- Accurate (distributive)

Control Flow Analysis

Data-flow

- Flow of data values from defs to uses
- Could alternatively be represented as a data dependence

Control-flow

- Sequencing of operations
- Could alternatively be represented as a control dependence
- *e.g.,* Evaluation of then-code and else-code depends on if-test

Why study control flow analysis?

Finding Loops

- most computation time is spent in loops
- to optimize them, we need to find them

Loop Optimizations

- Loop-invariant code hoisting
- Induction variable elimination
- Array bounds check removal
- Loop unrolling
- Parallelization
- ...

Identifying structured control flow

- can be used to speed up data-flow analysis

Representing Control-Flow

High-level representation

- Control flow is implicit in an AST

Low-level representation:

- Use a **Control-flow graph**
  - Nodes represent statements
  - Edges represent explicit flow of control

Other options

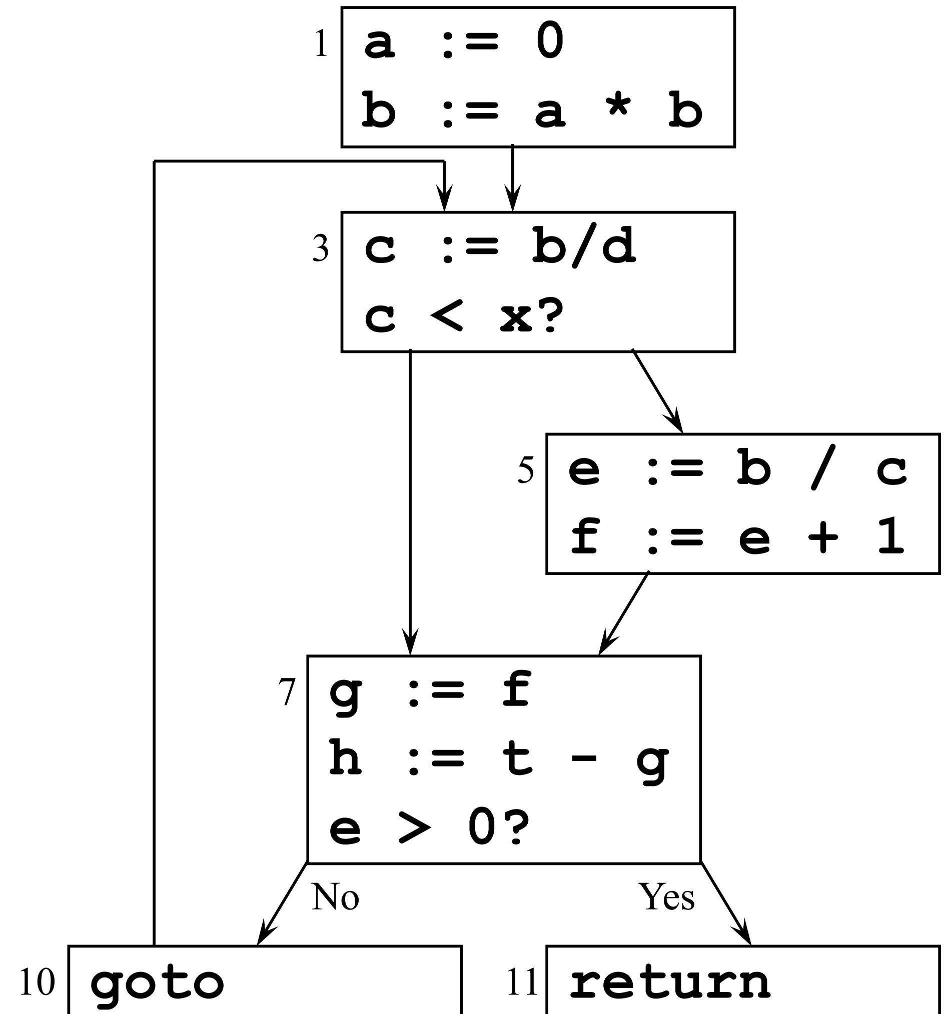
- Control dependences in program dependence graph (PDG) [Ferrante87]
- Dependences on explicit state in value dependence graph (VDG) [Weise 94]

What Is Control-Flow Analysis?

Control-flow analysis discovers the flow of control within a procedure
(*e.g.*, builds a CFG, identifies loops)

Example

```
1        a := 0
2        b := a * b
3  L1:   c := b/d
4        if c < x goto L2
5        e := b / c
6        f := e + 1
7  L2:   g := f
8        h := t - g
9        if e > 0 goto L3
10 goto L1
11 L3:   return
```



Loop Concepts

Loop: Strongly connected subgraph of CFG with a single entry point (header)

Loop entry edge: Source not in loop & target in loop

Loop exit edge: Source in loop & target not in loop

Loop header node: Target of loop entry edge. Dominates all nodes in loop.

Back edge: Target is loop header & source is in the loop

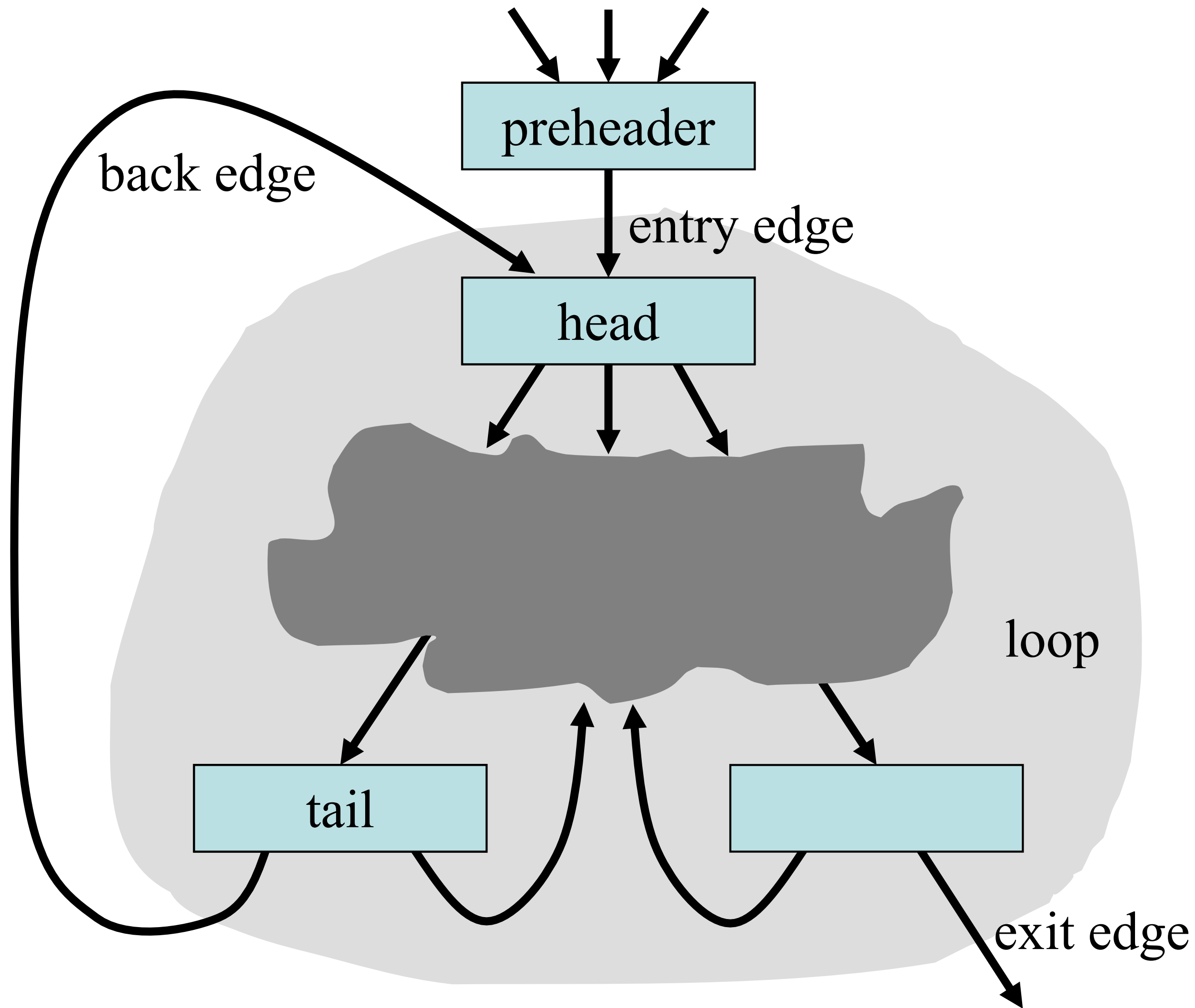
Natural loop: Associated with each back edge. Nodes dominated by header and with path to back edge without going through header

Loop tail node: Source of back edge

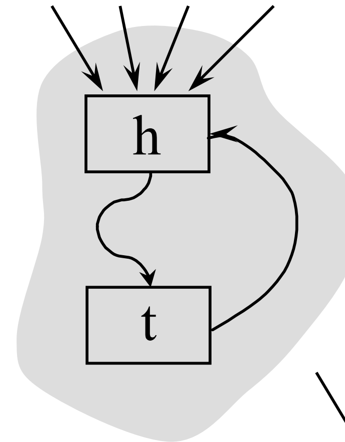
Loop preheader node: Single node that's source of the loop entry edge

Nested loop: Loop whose header is inside another loop

Picturing Loop Terminology



The Value of Preheader Nodes

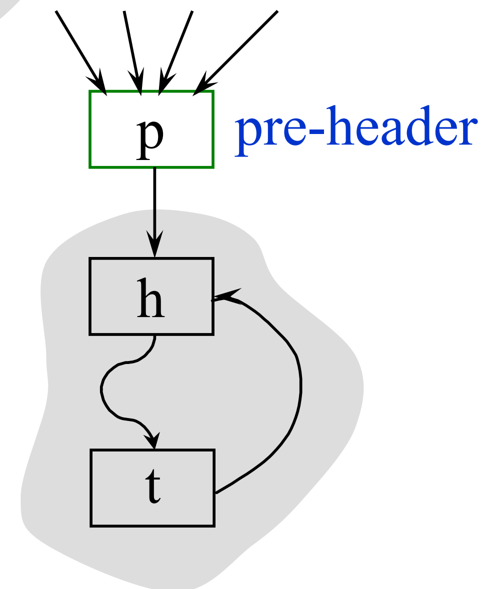


Not all loops have preheaders

- Sometimes it is useful to create them

Without preheader node

- There can be multiple entry edges



With single preheader node

- There is only one entry edge

Useful when moving code outside the loop

- Don't have to replicate code for multiple entry edges

Identifying Loops

Why?

– Most execution time spent in loops, so optimizing loops will often give most benefit

Many approaches

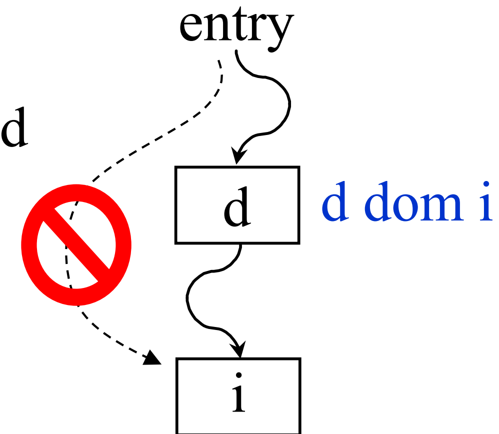
– Interval analysis
  – Exploit the natural hierarchical structure of programs
  – Decompose the program into nested regions called intervals
– Structural analysis: a generalization of interval analysis
– Identify **dominators** to discover loops

We'll focus on the dominator-based approach

Dominator Terminology

Dominators

d **dom** i if all paths from entry to node i include d

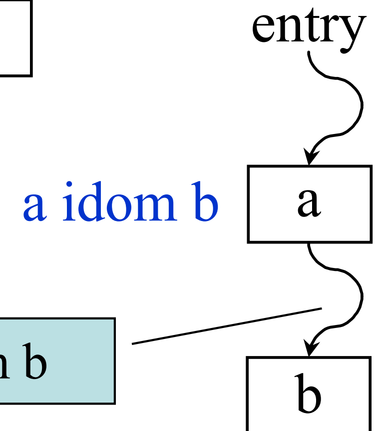


Strict dominators

d **sdom** i if d dom i and d ≠ i

Immediate dominators

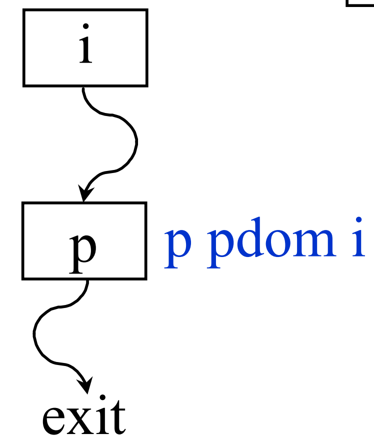
a **idom** b if a **sdom** b and there does not exist a node c such that c ≠ a, c ≠ b, a dom c, and c dom b



not ∃ c, a sdom c and c sdom b

Post dominators

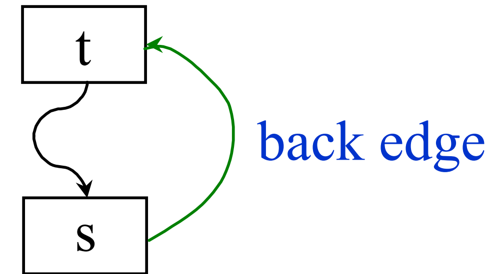
p **pdom** i if every possible path from i to exit includes p (p dom i in the flow graph whose arcs are reversed and entry and exit are interchanged)



Identifying Natural Loops with Dominators

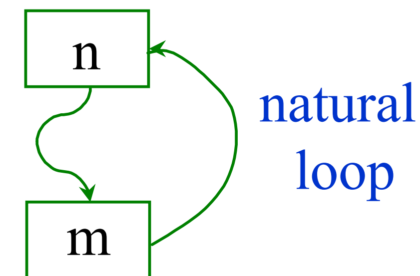
Back edges

A **back edge** of a natural loop is one whose target dominates its source



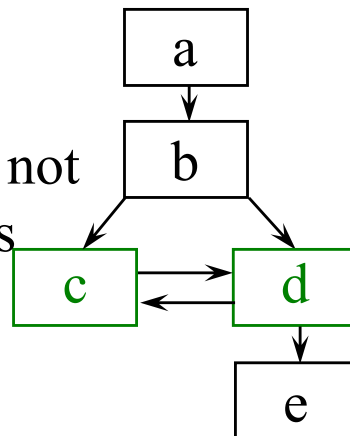
Natural loop

The **natural loop** of a back edge (m→n), where n dominates m, is the set of nodes x such that n dominates x and there is a path from x to m not containing n

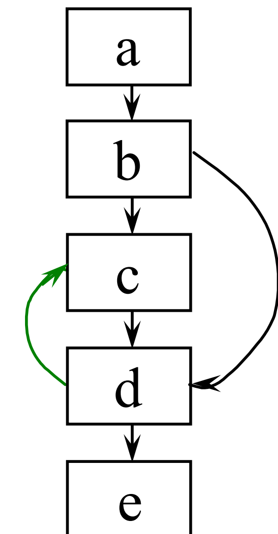


Example

SCC with c and d not a loop because has two entry points



The target, c, of the edge (d→c) does not dominate its source, d, so (d→c) does not define a natural loop



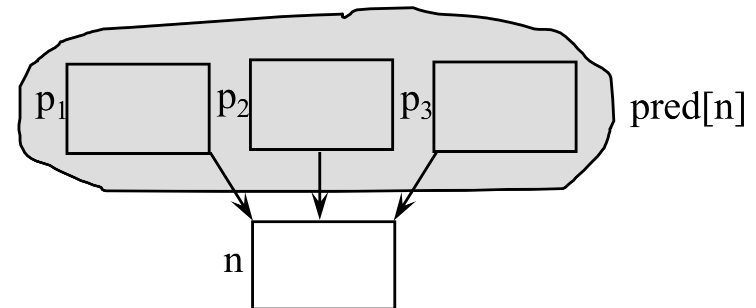
Computing Dominators

Input: Set of nodes N (in CFG) and an entry node s
Output: Dom[i] = set of all nodes that dominate node i

Dom[s] = {s}
for each n $\in$ N – {s}
    Dom[n] = N
repeat
    change = false
    for each n $\in$ N – {s}
        D = {n} $\cup$ ($\cap_{p \in pred(n)}$ Dom[p])
        if D $\neq$ Dom[n]
            change = true
            Dom[n] = D
until !change

Key Idea

If a node dominates all predecessors of node n, then it also dominates node n

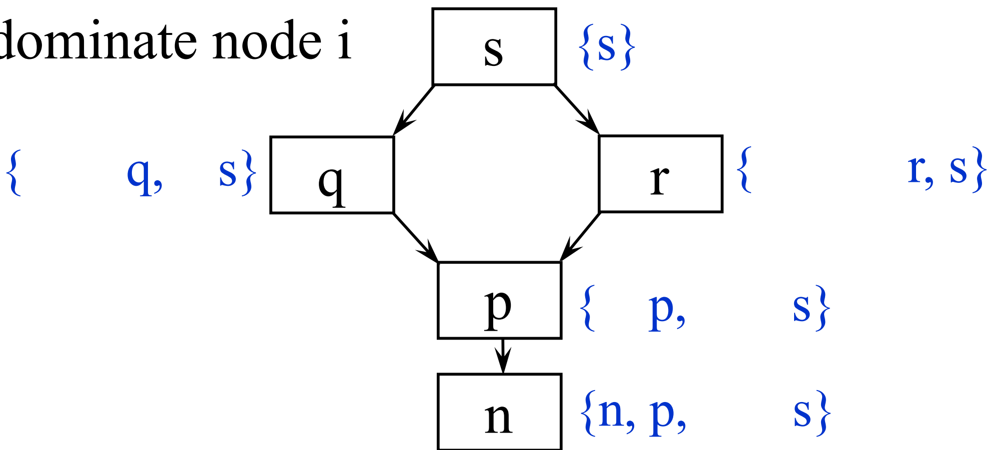


$x \in Dom(p_1) \wedge x \in Dom(p_2) \wedge x \in Dom(p_3) \Rightarrow x \in Dom(n)$

Computing Dominators (example)

Input: Set of nodes N and an entry node s

Output: Dom[i] = set of all nodes that dominate node i



Dom[s] = {s}

for each n ∈ N – {s}

  Dom[n] = N

repeat

  change = false

  for each n ∈ N – {s}

    $D = \{n\} \cup (\cap_{p \in pred(n)} Dom[p])$

    if D ≠ Dom[n]

      change = true

      Dom[n] = D

until !change

Initially

Dom[s] = {s}

Dom[q] = {n, p, q, r, s}. . .

Finally

Dom[q] = {q, s}

Dom[r] = {r, s}

Dom[p] = {p, s}

Dom[n] = {n, p, s}