CS 6410: Compilers

Fall 2023

Tamara Bonaci

t.bonaci@northeastern.edu

Thank you to UW faculty Hal Perkins. Today lecture notes are a modified version of his lecture notes.

Credits For Course Material

- Big thank you to UW CSE faculty member, Hal Perkins
- Some direct ancestors of this course:
  - UW CSE 401 (Chambers, Snyder, Notkin, Perkins, Ringenburg, Henry, ...)
  - UW CSE PMP 582/501 (Perkins)
  - Cornell CS 412-3 (Teitelbaum, Perkins)
  - Rice CS 412 (Cooper, Kennedy, Torczon)
  - Many books (Appel; Cooper/Torczon; Aho, [[Lam,] Sethi,] Ullman [Dragon Book], Fischer, [Cytron ,] LeBlanc; Muchnick, ...)

Agenda

- Dataflow analysis – review and finish
  - Framework for many common compiler analyses
  - Dataflow analysis for common subexpression elimination
  - Other analysis problems that work in the same framework
  - Some of these are optimizations we've seen, but more formally and with details
- Loops optimizations
  - Dominators – discovering loops
  - Loop invariant calculations
  - Loop transformations
  - A quick look at some memory hierarchy issues
  - Largely based on material in Appel ch. 18, 21; similar material in other books
- Overview of SSA IR
  - Constructing SSA graphs
  - Sample of SSA-based optimizations
  - Converting back from SSA form
  - Sources: Appel ch. 19, also an extended discussion in Cooper-Torczon sec. 9.3, Mike Ringenburg's CSE 401 slides (13wi)

Value Numbering

Review: Value Numbering

- Technique for eliminating redundant expressions:
  - Assign an identifying number VN(n) to each expression
  - VN(x + y) = VN(j) if x+y and j have the same value
  - Use hashing over value numbers for efficiency
- Old idea (Balke 1968, Ershov 1954)
  - Invented for low-level, linear IRs
  - Equivalent methods exist for tree IRs, e.g., build a DAG

Optimization Categories (1)

- *Local methods*
  - Usually confined to basic blocks
  - Simplest to analyze and understand
  - Most precise information

Optimization Categories (2)

- *Superlocal methods*
  - Operate over *Extended Basic Blocks* (EBBs)
    - An EBB is a set of blocks b_1, b_2, …, b_n where b_1 has multiple predecessors and each of the remaining blocks b_i ($2 \leq i \leq n$) have only b_{i-1} as its unique predecessor
    - The EBB is entered only at b_1, but may have multiple exits
    - A single block b_i can be the head of multiple EBBs (these EBBs form a tree rooted at b_i)
  - Use information discovered in earlier blocks to improve code in successors

Optimization Categories (3)

- *Regional methods*
  - Operate over scopes larger than an EBB but smaller than an entire procedure/ function/method
  - Typical example: loop body
  - Difference from superlocal methods is that there may be merge points in the graph (i.e., a block with two or more predecessors)
    - Facts true at merge point are facts known to be true on all possible paths to that point

Optimization Categories (4)

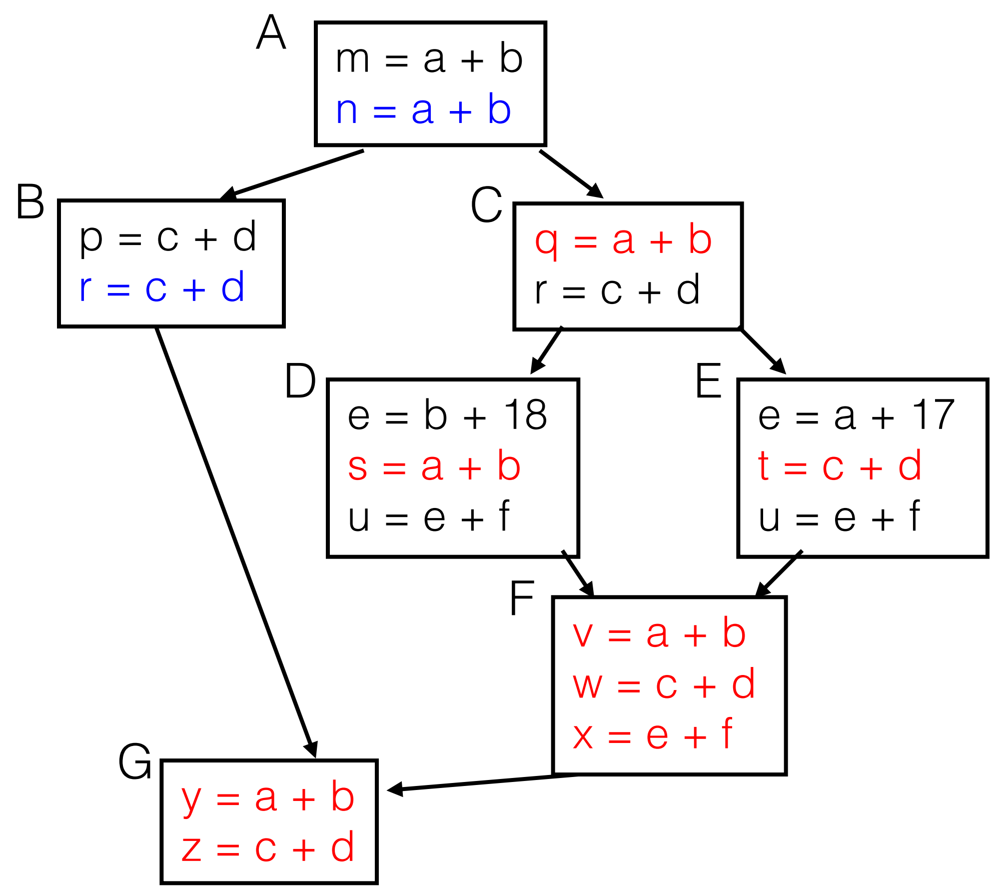
- *Global methods*
  - Operate over entire procedures
  - Sometimes called *intraprocedural* methods
  - Motivation is that local optimizations sometimes have bad consequences in larger context
  - Procedure/method/function is a natural unit for analysis, separate compilation, etc.
  - Almost always need global *data-flow* analysis information for these

Optimization Categories (5)

- *Whole-program methods*
  - Operate over more than one procedure
  - Sometimes called *interprocedural* methods
  - Challenges: name scoping and parameter binding issues at procedure boundaries
  - Classic examples: inline method substitution, interprocedural constant propagation
  - Common in aggressive JIT compilers and optimizing compilers for object-oriented languages

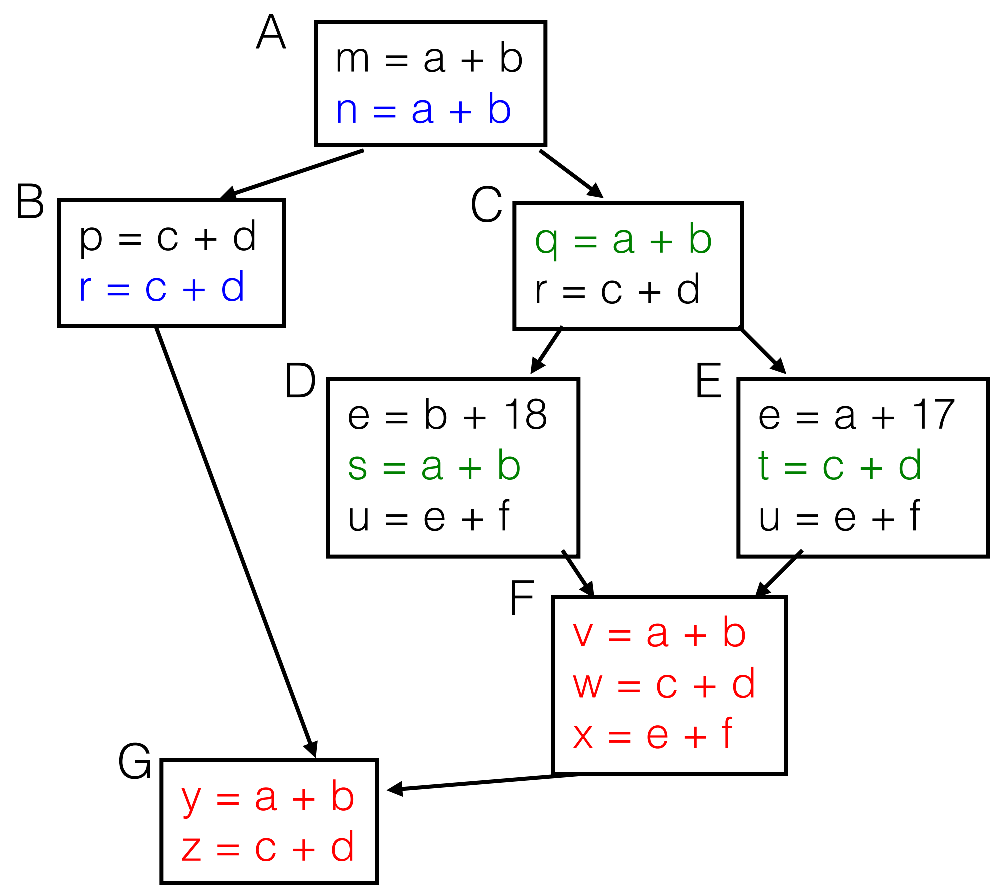
Value Numbering Revisited

- Local Value Numbering
  - 1 block at a time
  - Strong local results
  - No cross-block effects
- Missed opportunities



Superlocal Value Numbering

- Idea: apply local method to EBBs
  - {A,B}, {A,C,D}, {A,C,E}
- Final info from A is initial info for B, C; final info from C is initial for D, E
- Gets reuse from ancestors
- Avoid reanalyzing A, C
- Doesn't help with F, G



SSA Name Space

- Two Principles
  - Each name is defined by exactly one operation
  - Each operand refers to exactly one definition
- Need to deal with merge points
  - Add Φ functions at merge points to reconcile names
  - Use subscripts on variable names for uniqueness

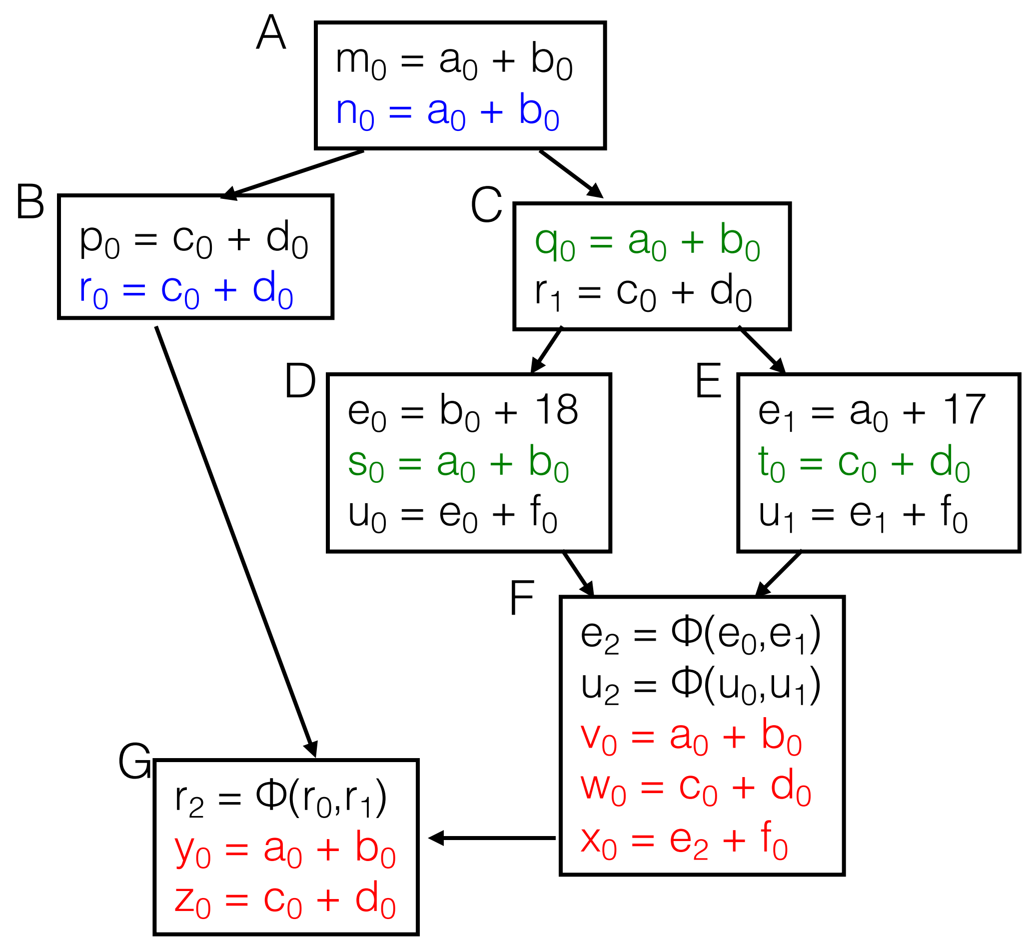
SSA Name Space (from before)

Code	Rewritten
$a_0^3 = x_0^1 + y_0^2$	$a_0^3 = x_0^1 + y_0^2$
$b_0^3 = x_0^1 + y_0^2$	$b_0^3 = a_0^3$
$a_1^4 = 17$	$a_1^4 = 17$
$c_0^3 = x_0^1 + y_0^2$	$c_0^3 = a_0^3$

- Unique name for each definition
- Name ⇔ VN
- a_0^3 is available to assign to c_0^3

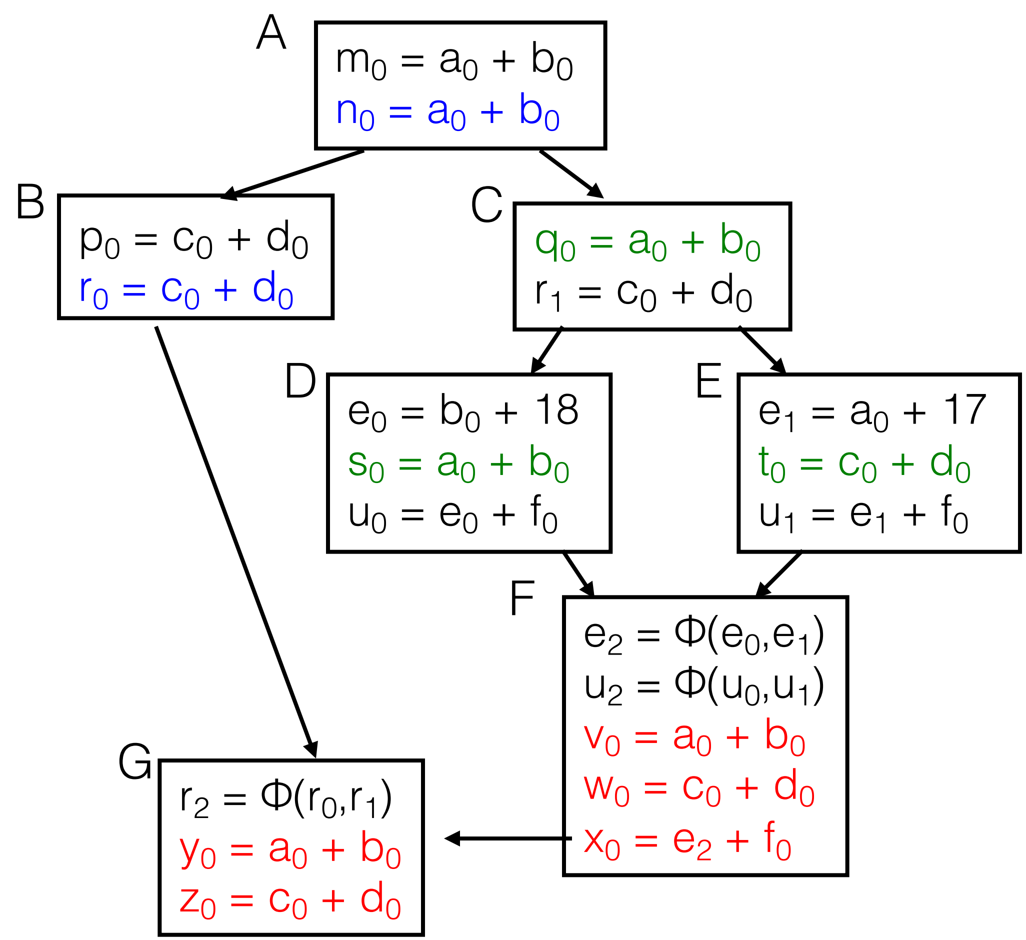
Superlocal Value Numbering with All Bells & Whistles

- Finds more redundancies
- Little extra cost
- Still does nothing for F and G



Larger Scopes

- Still have not helped F and G
- Problem: multiple predecessors
- Must decide what facts hold in F and in G
  - For G, combine B & F?
  - Merging states is expensive
  - Fall back on what we know



Dominators

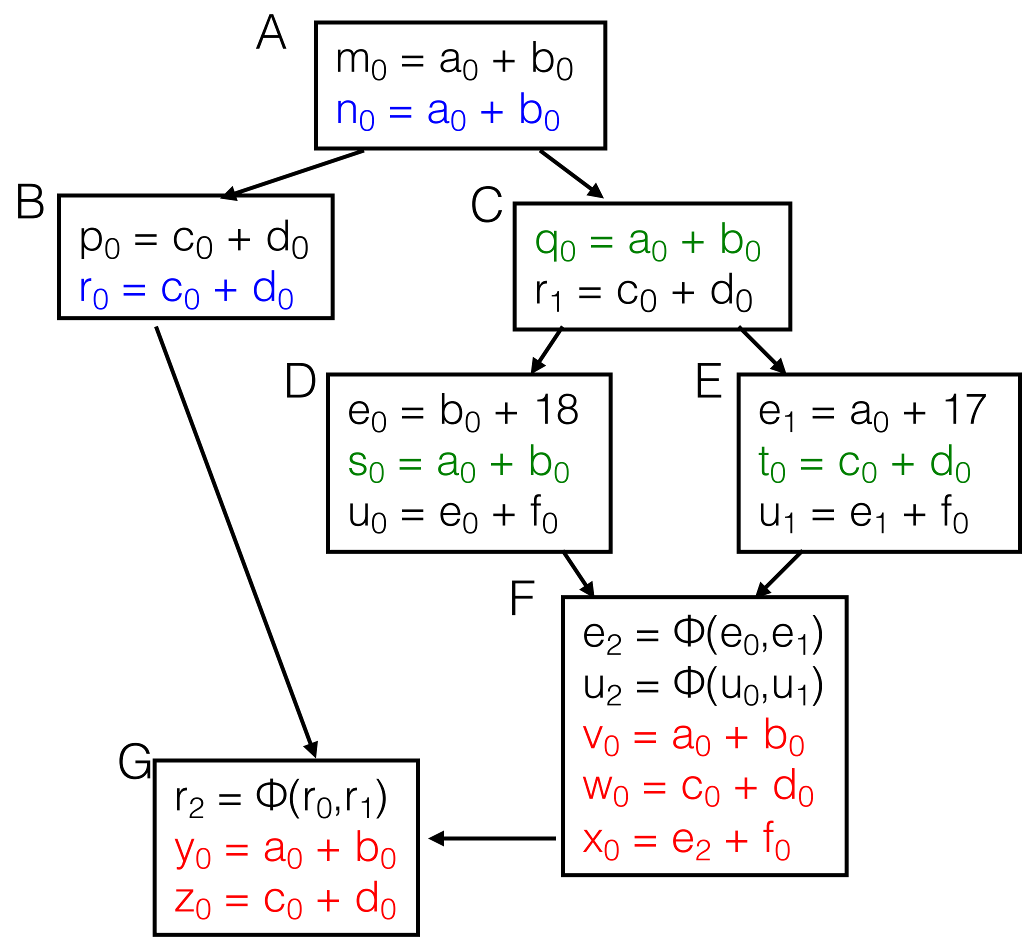
- Definition
  - x *dominates* y if and only if every path from the entry of the control-flow graph to y includes x
- By definition, x dominates x
- Associate a Dom set with each node
  - $|\ \mathrm{Dom}(x)\ | \geq 1$
- Many uses in analysis and transformation
  - Finding loops, building SSA form, code motion

Immediate Dominators

- For any node x, there is a y in Dom(x) closest to x
- This is the *immediate dominator* of x
  - Notation: IDom(x)

Dominator Sets

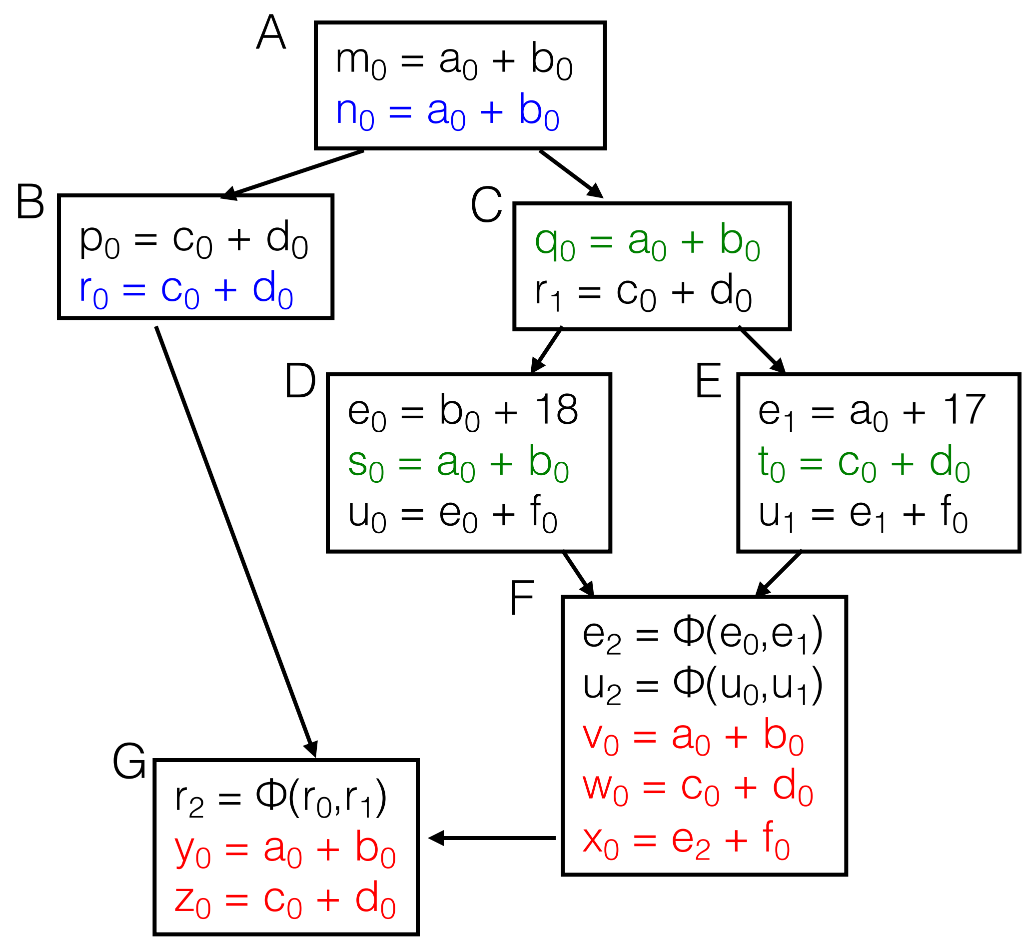
Block	Dom	IDom



Note that the IDOM relation defines a tree!

Dominator Value Numbering

- Still looking for a way to handle F and G
- Idea: Use info from IDom(x) to start analysis of x
  - Use C for F and A for G
- Dominator VN Technique (DVNT)

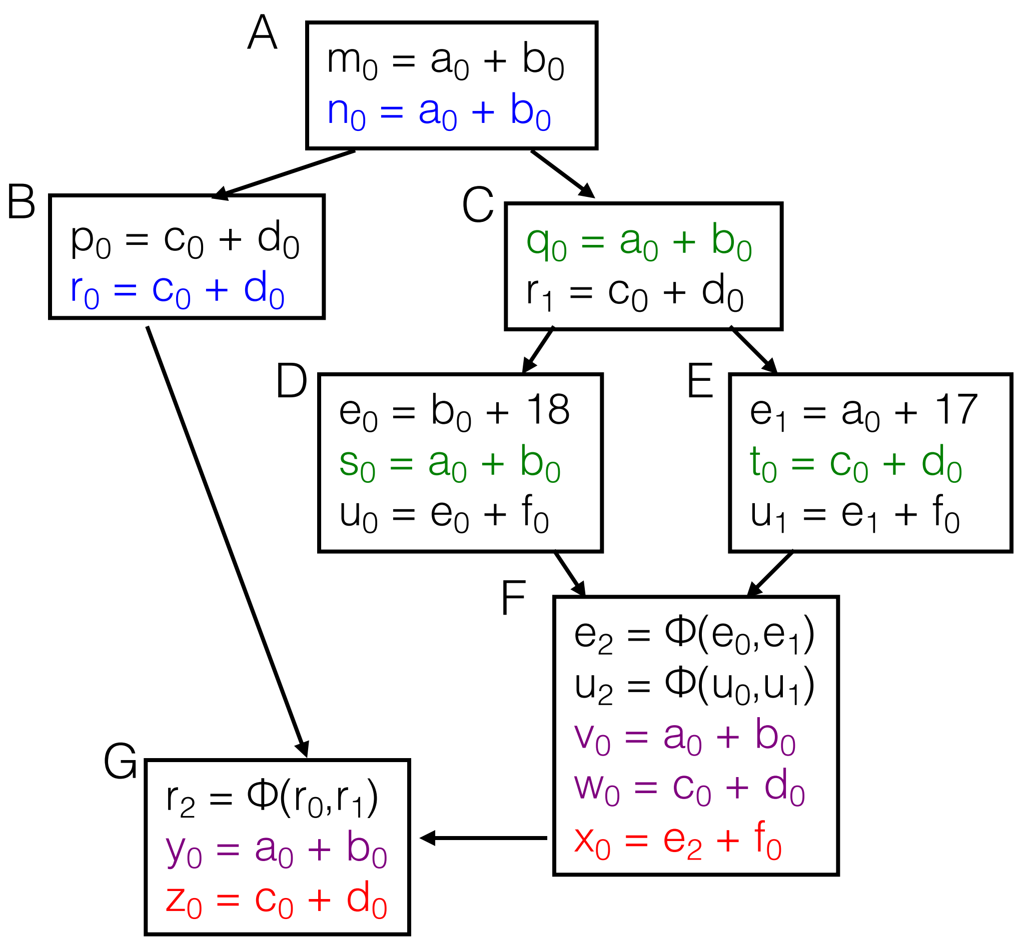


DVNT Algorithm

- Use superlocal algorithm on extended basic blocks
  - Use scoped hash tables & SSA name space as before
- Start each node with table from its IDOM
- No values flow along back edges (i.e., loops)
- Constant folding, algebraic identities as before

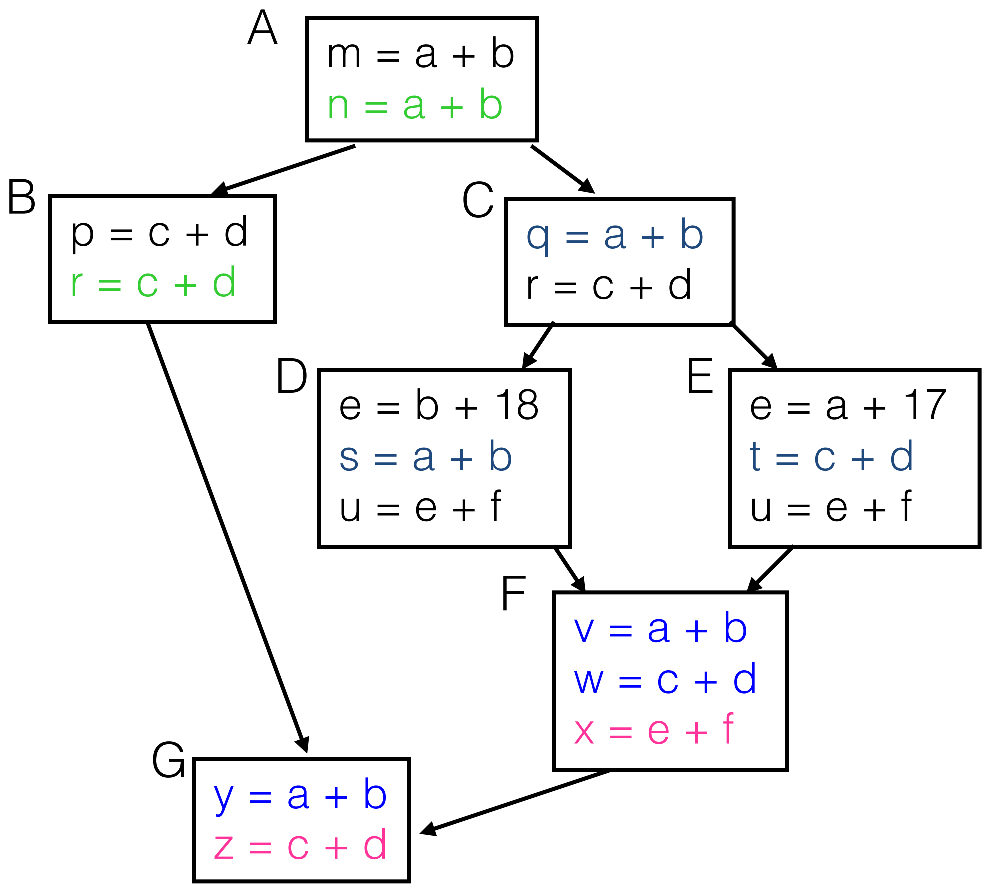
Dominator Value Numbering

- Advantages
  - Finds more redundancy
  - Little extra cost
- Shortcomings
  - Misses some opportunities (common calculations in ancestors that are not IDOMs)
  - Doesn't handle loops or other back edges



Comparing Algorithms

- LVN – Local Value Numbering
- SVN – Superlocal Value Numbering
- DVN – DominatoT-based Value Numbering
- GRE – Global Redundancy Elimination



Comparing Algorithms (2)

- LVN => SVN => DVN form a strict hierarchy – later algorithms find a superset of previous information
- Global RE finds a somewhat different set
  - Discovers e+f in F (computed in both D and E)
  - Misses identical values if they have different names (e.g.,
    a+b and c+d when a=c and b=d)
    - Value Numbering catches this

Scope of Analysis

- Larger context (EBBs, regions, global, interprocedural) sometimes helps
  - More opportunities for optimizations
- But not always
  - Introduces uncertainties about flow of control
  - Usually only allows weaker analysis
  - Sometimes has unwanted side effects
    - Can create additional pressure on registers, for example

The Story So Far…

- Local algorithm
- Superlocal extension
  - Some local methods extend cleanly to superlocal scopes
- Dominator VN Technique (DVNT)
- All of these propagate along forward edges
- None are global

Dataflow Analysis

- A collection of techniques for compile-time reasoning about run-time values
- Almost always involves building a graph
  - Trivial for basic blocks
  - Control-flow graph or derivative for global problems
  - Call graph or derivative for whole-program problems

Dataflow Analysis

- Limitations
  - Precision – “up to symbolic execution”
    - Assumes all paths taken
  - Sometimes cannot afford to compute full solution
  - Arrays – classic analysis treats each array as a single fact
  - Pointers – difficult, expensive to analyze
    - Imprecision rapidly adds up
    - But gotta do it to effectively optimize things like C/C++
- For scalar values we can quickly solve simple problems

Dataflow Analysis

- Many different applications of dataflow analysis:
  - Available expressions
  - Live variables
  - Reaching definitions
  - Very busy expressions

Characterizing Dataflow Analysis

- All of these algorithms involve sets of facts about each basic block b
  - IN(b) – facts true on entry to b
  - OUT(b) – facts true on exit from b
  - GEN(b) – facts created and not killed in b
  - KILL(b) – facts killed in b
- These are related by the equation

  $$OUT(b) = GEN(b) \cup (IN(b) - KILL(b))$$

  - Solve this iteratively for all blocks
  - Sometimes information propagates forward; sometimes backward

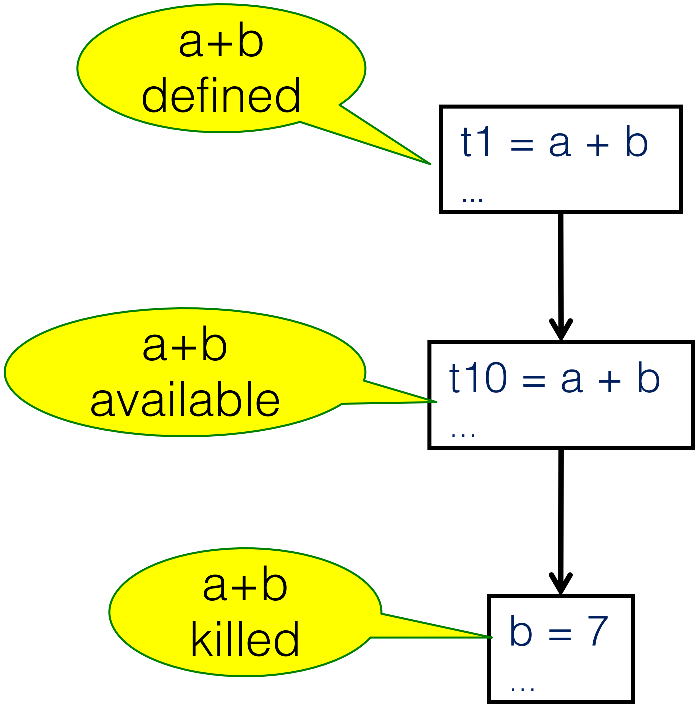
Available Expressions

Available Expressions

- Goal: use dataflow analysis to find common sub-expressions whose range spans basic blocks
- Idea: calculate *available expressions* at beginning of each basic block
- Avoid re-evaluation of an available expression – use a copy operation

“Available” and Other Terms

- An expression *e* is *defined* at point *p* in the CFG if its value is computed at *p*
  - Sometimes called *definition site*
- An expression *e* is *killed* at point *p* if one of its operands is defined at *p*
  - Sometimes called *kill site*
- An expression *e* is *available* at point *p* if every path leading to *p* contains a prior definition of *e* and *e* is not killed between that definition and *p*



Available Expression Sets

- To compute available expressions, for each block *b*, define
  - AVAIL(b) – the set of expressions available on entry to *b*
  - NKILL(b) – the set of expressions not killed in *b*
    - i.e., all expressions in the program *except* for those killed in *b*
  - DEF(b) – the set of expressions defined in *b* and not subsequently killed in *b*

Computing Available Expressions

- Big Picture
  - Build control-flow graph
  - Calculate initial local data – DEF(*b*) and NKILL(*b*)
    - This only needs to be done once for each block *b* and depends only on the statements in *b*
  - Iteratively calculate AVAIL(*b*) by repeatedly evaluating equations until nothing changes
    - Another fixed-point algorithm

Computing Available Expressions

- AVAIL(b) is the set

  $\text{AVAIL}(b) = \cap_{x \in \text{preds}(b)} (\text{DEF}(x) \cup (\text{AVAIL}(x) \cap \text{NKILL}(x)))$

  – preds(b) is the set of b's predecessors in the CFG

  – The set of expressions available on entry to *b* is the set of expressions that were available at the end of *every* predecessor basic block *x*

  – The expressions available on exit from block *b* are those defined in *b* or available on entry to *b* and not killed in *b*

- This gives a system of simultaneous equations – a dataflow problem

Computing DEF and NKILL (1)

- For each block *b* with operations $o_1, o_2, \ldots, o_k$

  KILLED = ∅ // killed *variables*, not expressions

  DEF(b) = ∅

  for i = k to 1 // note: working back to front

      assume o_i is "x = y + z"

      if (y $\notin$ KILLED and z $\notin$ KILLED)

          add "y + z" to DEF(b)

      add x to KILLED

  …

Computing DEF and NKILL (2)

- After computing DEF and KILLED for a block *b*, compute set of all expressions in the program not killed in *b*

NKILL(*b*) = { all expressions }

for each expression *e*

    for each variable $v \in e$

        if $v \in$ KILLED then

            NKILL(*b*) = NKILL(*b*) - *e*

Computing Available Expressions

Once DEF(b) and NKILL(b) are computed for all blocks b

Worklist = { all blocks b_i }

while (Worklist ≠ ∅)

  remove a block *b* from Worklist

  recompute AVAIL(*b*)

  if AVAIL(*b*) changed

    Worklist = Worklist ∪ successors(*b*)

Live Variable Analysis

Live Variable Analysis

- A variable *v* is *live* at point *p* if and only if there is *any* path from *p* to a use of *v* along which *v* is not redefined
- Some uses:
  - Register allocation – only live variables need a register
  - Eliminating useless stores – if variable not live at store, then stored variable will never be used
  - Detecting uses of uninitialized variables – if live at declaration (before initialization) then it might be used uninitialized
  - Improve SSA construction – only need Φ-function for variables that are live in a block (later)

Liveness Analysis Sets

- For each block b, define
  - use[*b*] = variable used in *b* before any def
  - def[*b*] = variable defined in *b* & not killed
  - in[*b*] = variables live on entry to *b*
  - out[*b*] = variables live on exit from *b*

Equations for Live Variables

- Given the preceding definitions, we have:

  $\text{in}[b] = \text{use}[b] \cup (\text{out}[b] - \text{def}[b])$

  $\text{out}[b] = \cup_{s \in \text{succ}[b]} \text{in}[s]$

- Algorithm:
  - Set $\text{in}[b] = \text{out}[b] = \varnothing$
  - Update in, out until no change

Equations for Live Variables v2

- Many problems have more than one formulation. For example, Live Variables…
- Sets:
  - USED(b) – variables used in *b* before being defined in *b*
  - NOTDEF(b) – variables not defined in *b*
  - LIVE(b) – variables live on *exit* from *b*
- Equation:

$$\text{LIVE}(b) = \cup_{s \in succ(b)} \text{USED}(s) \cup (\text{LIVE}(s) \cap \text{NOTDEF}(s))$$

Reaching Definitions

Reaching Definitions

- A definition *d* of some variable *v* *reaches* operation *i* if and only if *i* reads the value of *v* and there is a path from *d* to *i* that does not define *v*
- Uses:
  - Find all of the possible definition points for a variable in an expression

Equations for Reaching Definitions

- Sets:
  - DEFOUT(b) – set of definitions in *b* that reach the end of *b* (i.e., not subsequently redefined in b)
  - SURVIVED(b) – set of all definitions not obscured by a definition in *b*
  - REACHES(b) – set of definitions that reach b
- Equation:

$$\text{REACHES}(b) = \cup_{p \in \text{preds}(b)} \text{DEFOUT}(p) \cup (\text{REACHES}(p) \cap \text{SURVIVED}(p))$$

Very Busy Expressions

Very Busy Expressions

- An expression *e* is considered *very busy* at some point *p* if *e* is evaluated and used along every path that leaves *p*, and evaluating *e* at *p* would produce the same result as evaluating it at the original locations
- Uses
  - Code hoisting – move *e* to *p* (reduces code size; no effect on execution time)

Equations for Very Busy Expressions

- Sets:
  - USED(b) – expressions used in *b* before they are killed
  - KILLED(b) – expressions redefined in *b* before they are used
  - VERYBUSY(b) – expressions very busy on exit from *b*
- Equation:

$$\text{VERYBUSY}(b) = \bigcap_{s \in \text{succ}(b)} \text{USED}(s) \cup (\text{VERYBUSY}(s) - \text{KILLED}(s))$$

Using Dataflow Information

- A few examples of possible transformations that use dataflow information:
  - Common sub-expression elimination
  - Constant propagation
  - Copy propagation
  - Dead code elimination

Classic Common-Subexpression Elimination (CSE)

- In a statement s: t := x op y, if x op y is *available* at s, then it need not be recomputed

- Analysis: compute *reaching expressions* i.e., statements n: v := x op y such that the path from n to s does not compute x op y or define x or y

Classic CSE Transformation

- If x op y is defined at n and reaches s
  - Create new temporary w
  - Rewrite n: v := x op y as

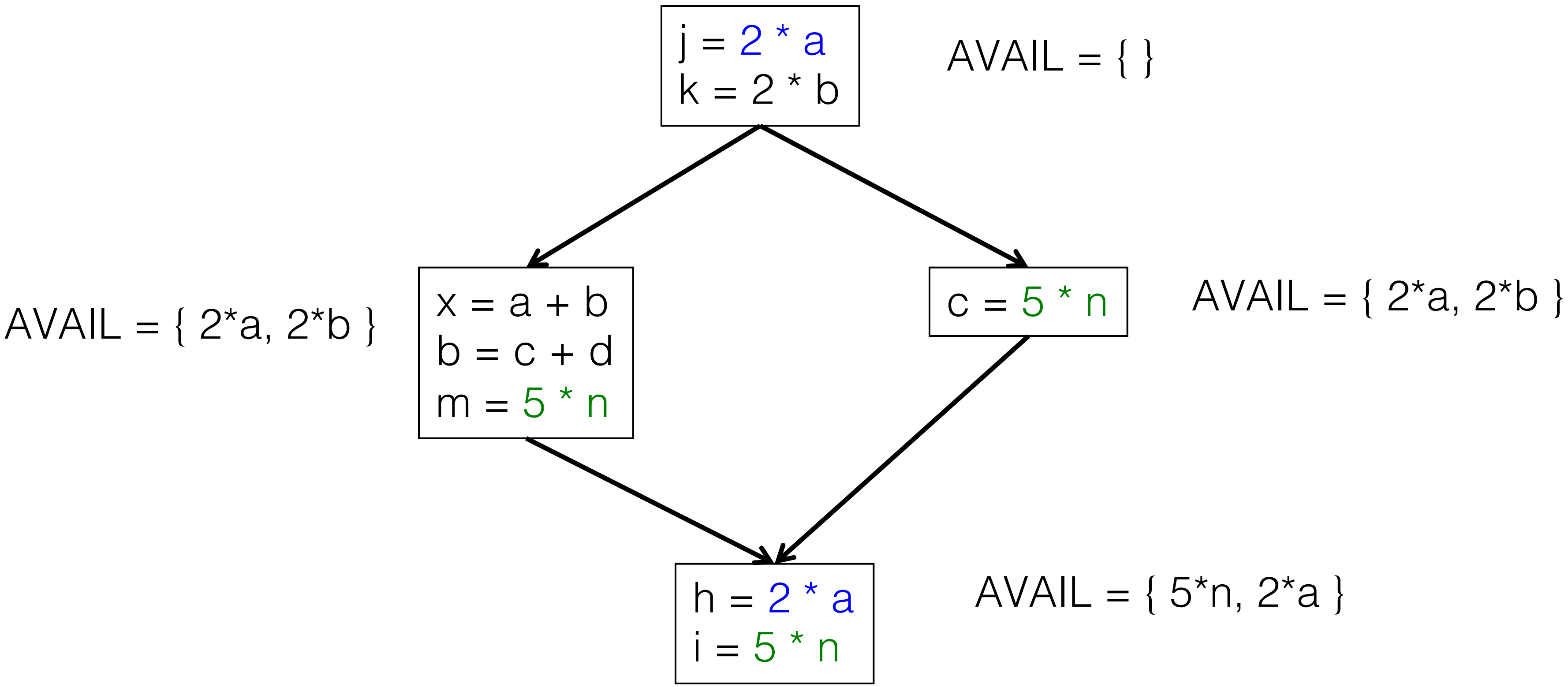
    n: w := x op y

    n': v := w
  - Modify statement s to be

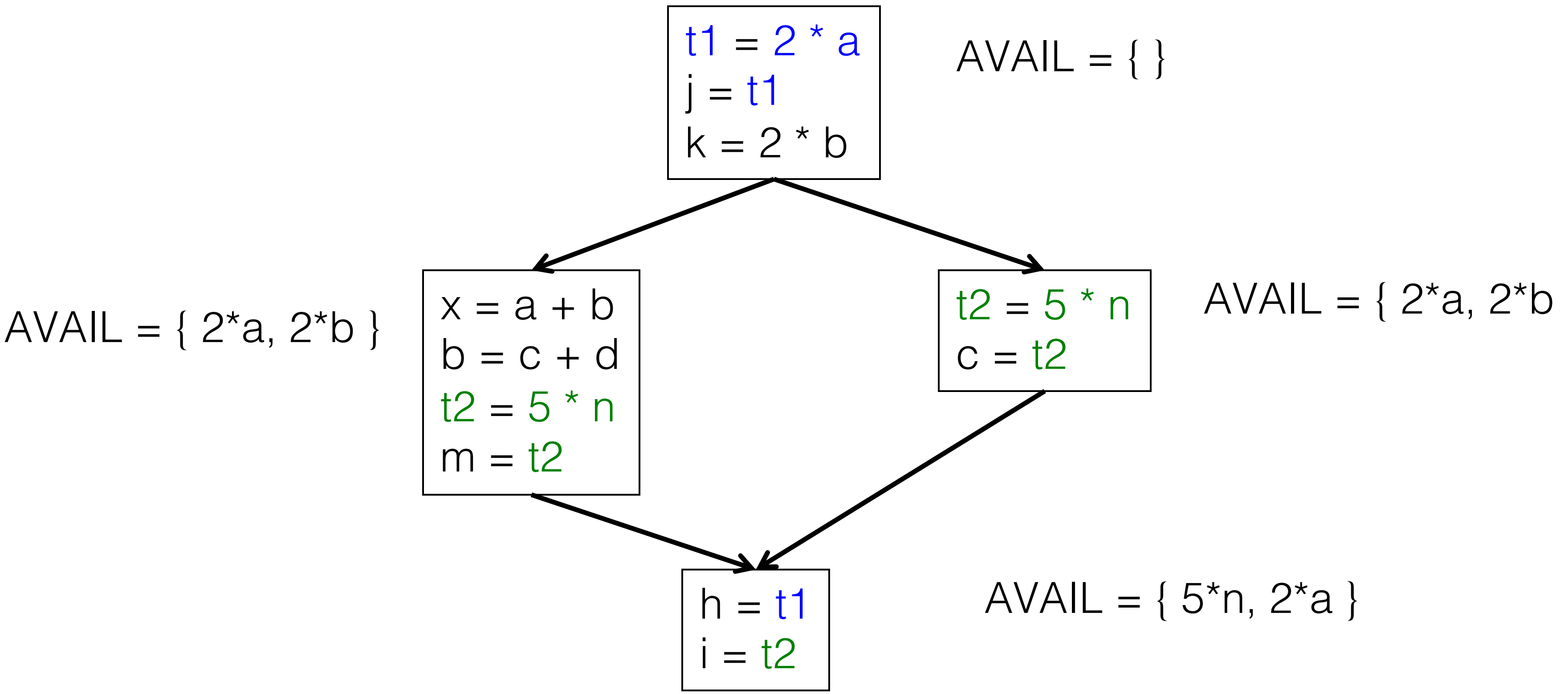
    s: t := w

  - (Rely on copy propagation to remove extra assignments that are not really needed)

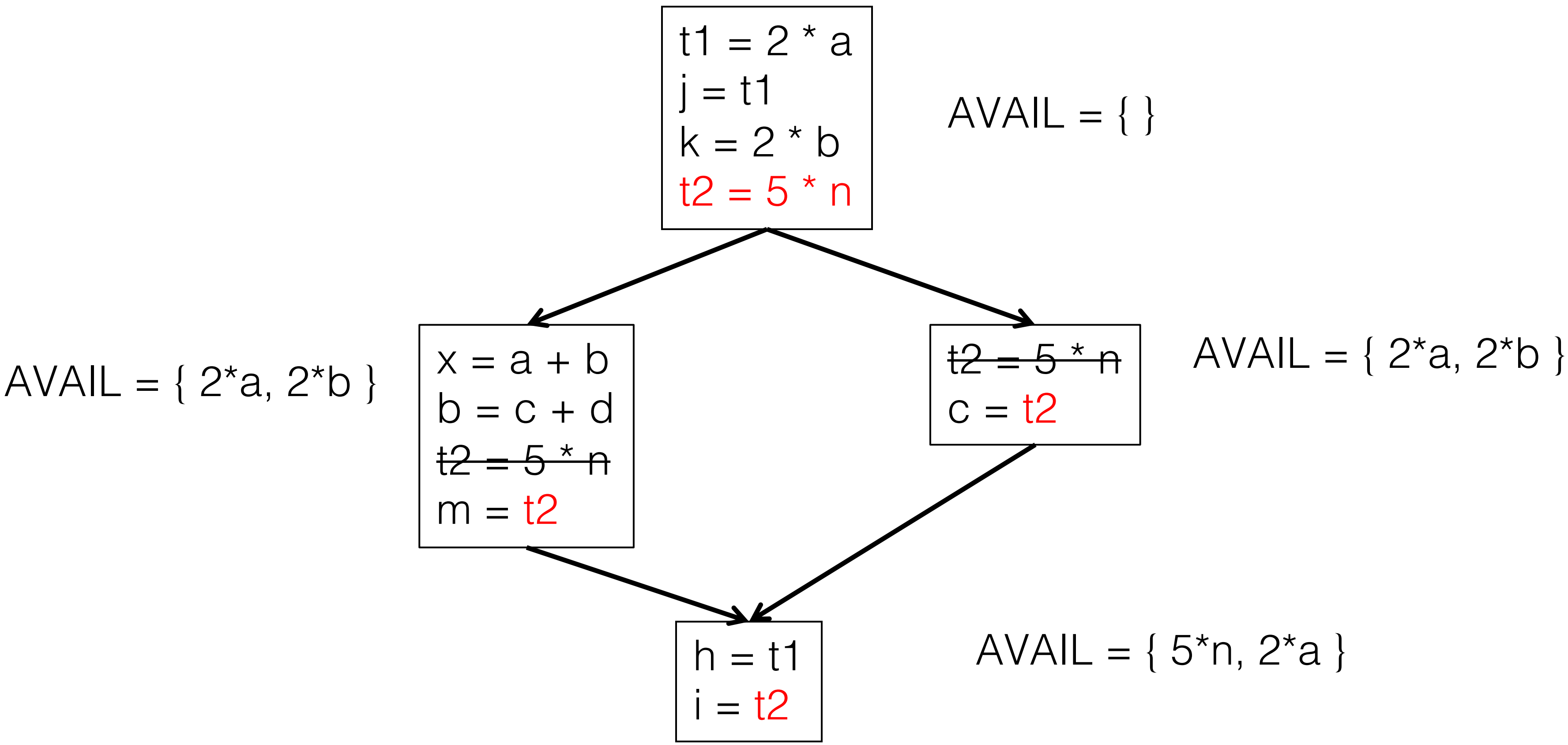
Revisiting Example (w/slight addition)



Revisiting Example (w/slight addition)



Then Apply Very Busy…



Constant Propagation

- Suppose we have
  - Statement d: t := c, where c is constant
  - Statement n that uses t
- If d reaches n and no other definitions of t reach n, then rewrite n to use c instead of t

Copy Propagation

- Similar to constant propagation
- Setup:
  - Statement d: t := z
  - Statement n uses t
- If d reaches n and no other definition of t reaches n, and there is no definition of z on any path from d to n, then rewrite n to use z instead of t
  - Recall that this can help remove dead assignments

Copy Propagation Tradeoffs

- Downside is that this can increase the lifetime of variable z and increase need for registers or memory traffic
- But it can expose other optimizations, e.g.,

```
a := y + z
u := y
c := u + z        // copy propagation makes this y + z
```

  – After copy propagation we can recognize the common subexpression

Dead Code Elimination

- If we have an instruction

  s: a := b op c

  and a is not live-out after s, then s can be eliminated
  - Provided it has no implicit side effects that are visible (output, exceptions, etc.)
    - If b or c are function calls, they have to be assumed to have unknown side effects unless the compiler can prove otherwise

Aliases

- A variable or memory location may have multiple names or *aliases*
  - Call-by-reference parameters
  - Variables whose address is taken (&x)
  - Expressions that dereference pointers (p.x, *p)
  - Expressions involving subscripts (a[i])
  - Variables in nested scopes

Aliases vs Optimizations

- Example:

  p.x := 5; q.x := 7; a := p.x;

  - Does reaching definition analysis show that the definition of p.x reaches a?
  - (Or: do p and q refer to the same variable/object?)
  - (Or: *can* p and q refer to the same thing?)

Aliases vs Optimizations

- Example

```
void f(int *p, int *q) {
  *p = 1; *q = 2;
  return *p;
}
```

  - How do we account for the possibility that p and q might refer to the same thing?
  - Safe approximation: since it’s possible, assume it is true (but rules out a lot)
    - C programmers can use “restrict” to indicate no other pointer is an alias for this one

Types and Aliases (1)

- In Java, ML, MiniJava, and others, if two variables have incompatible types they cannot be names for the same location
  - Also helps that programmer cannot create arbitrary pointers to storage in these languages

Types and Aliases (2)

- Strategy: Divide memory locations into *alias classes* based on type information (every type, array, record field is a class)
- Implication: need to propagate type information from the semantics pass to optimizer
  - Not normally true of a minimally typed IR
- Items in different alias classes cannot refer to each other

Aliases and Flow Analysis

- Idea: Base alias classes on points where a value is created
  - Every new/malloc and each local or global variable whose address is taken is an alias class
  - Pointers can refer to values in multiple alias classes (so each memory reference is to a set of alias classes)
  - Use to calculate “may alias” information (e.g., p “may alias” q at program point s)

Using “may-alias” information

- Treat each alias class as a “variable” in dataflow analysis problems
- Example: framework for available expressions
  - Given statement s: M[a]:=b,

    gen[s] = { }

    kill[s] = { M[x] | a may alias x at s }

May-Alias Analysis

- Without alias analysis, #2 kills M[t] since x and t might be related
- If analysis determines that "x may-alias t" is false, M[t] is still available at #3; can eliminate the common subexpression and use copy propagation

- Code

```
1:  u := M[t]
2:  M[x] := r
3:  w := M[t]
4:  b := u+w
```

Loops

Loops

Much of the execution time of programs is spent here

∴ worth considerable effort to make loops go faster

∴ want to figure out how to recognize loops and figure out how to “improve” them

What Is a Loop?

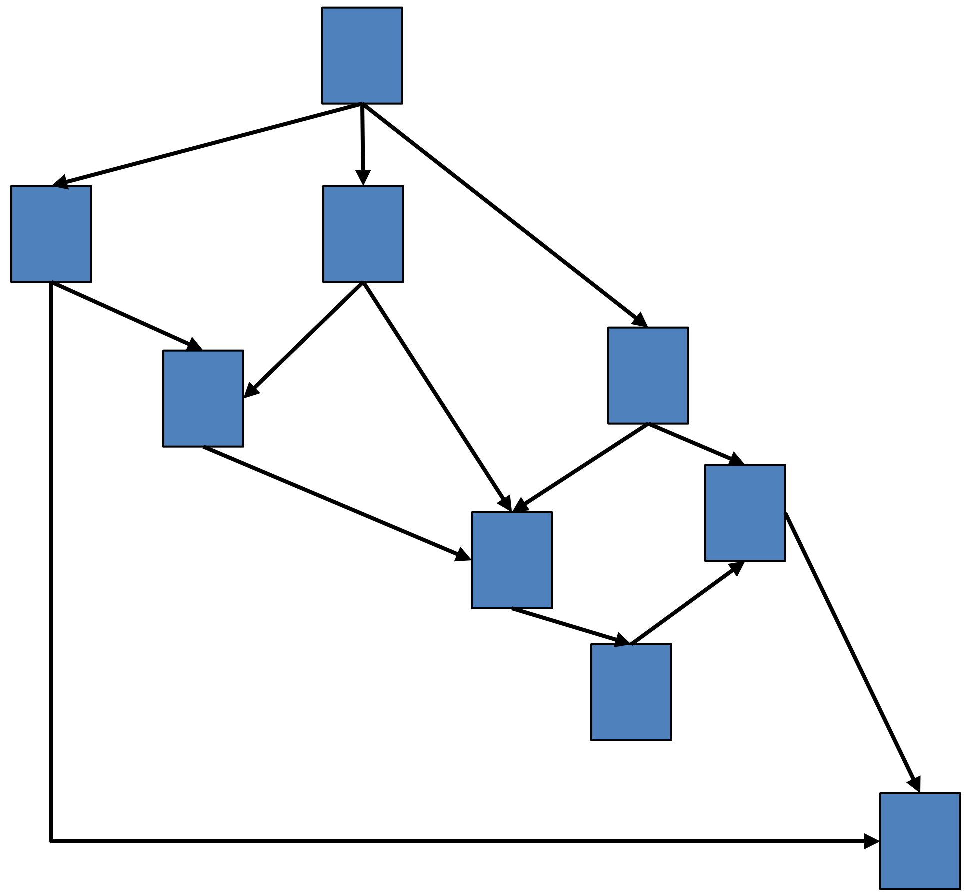
- In source code, a loop is the set of statements in the body of a for/while construct
- But, in a language that permits free use of GOTOs, how do we recognize a loop?
- In a control-flow-graph (node = basic-block, arc = flow-of-control), how do we recognize a loop?

Example: Any Loops in this Code?

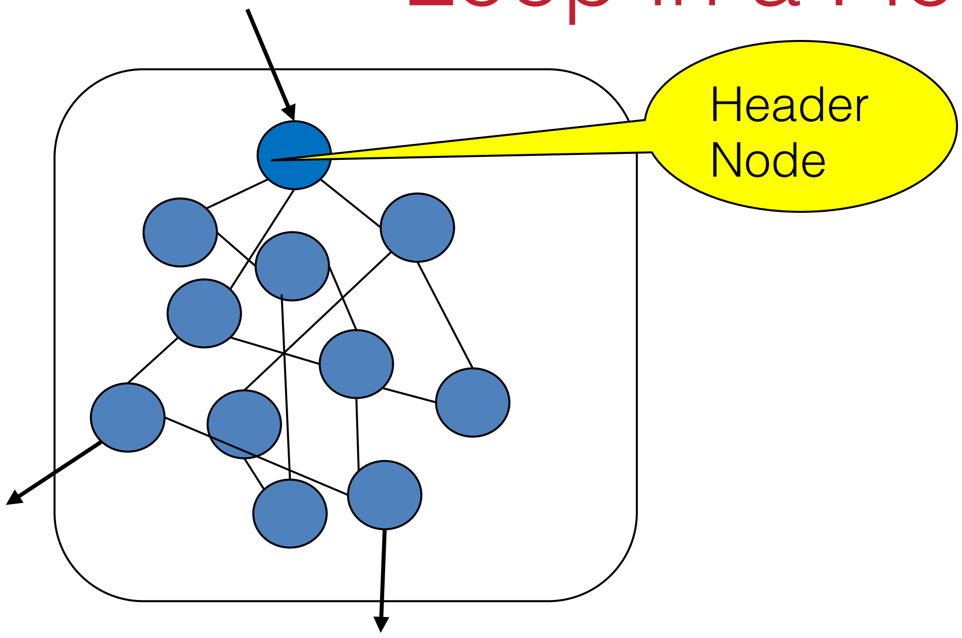
```
     i = 0
     goto L8
L7:  i++
L8:  if (i < N) goto L9
     s = 0
     j = 0
     goto L5
L4:  j++
L5:  N--
     if(j >= N) goto L3
     if (a[j+1] >= a[j]) goto L2
     t = a[j+1]
     a[j+1] = a[j]
     a[j] = t
     s = 1
L2:  goto L4
L3:  if(s != ) goto L1 else goto L9
L1:  goto L7
L9:  return
```

Anyone recognize or guess the algorithm?

Any Loops in this Flowgraph?



Loop in a Flowgraph: Intuition



- Cluster of nodes, such that:

- There's one node called the "header"
- I can reach all nodes in the cluster from the header
- I can get back to the header from all nodes in the cluster
- Only once entrance - via the header
- One or more exits

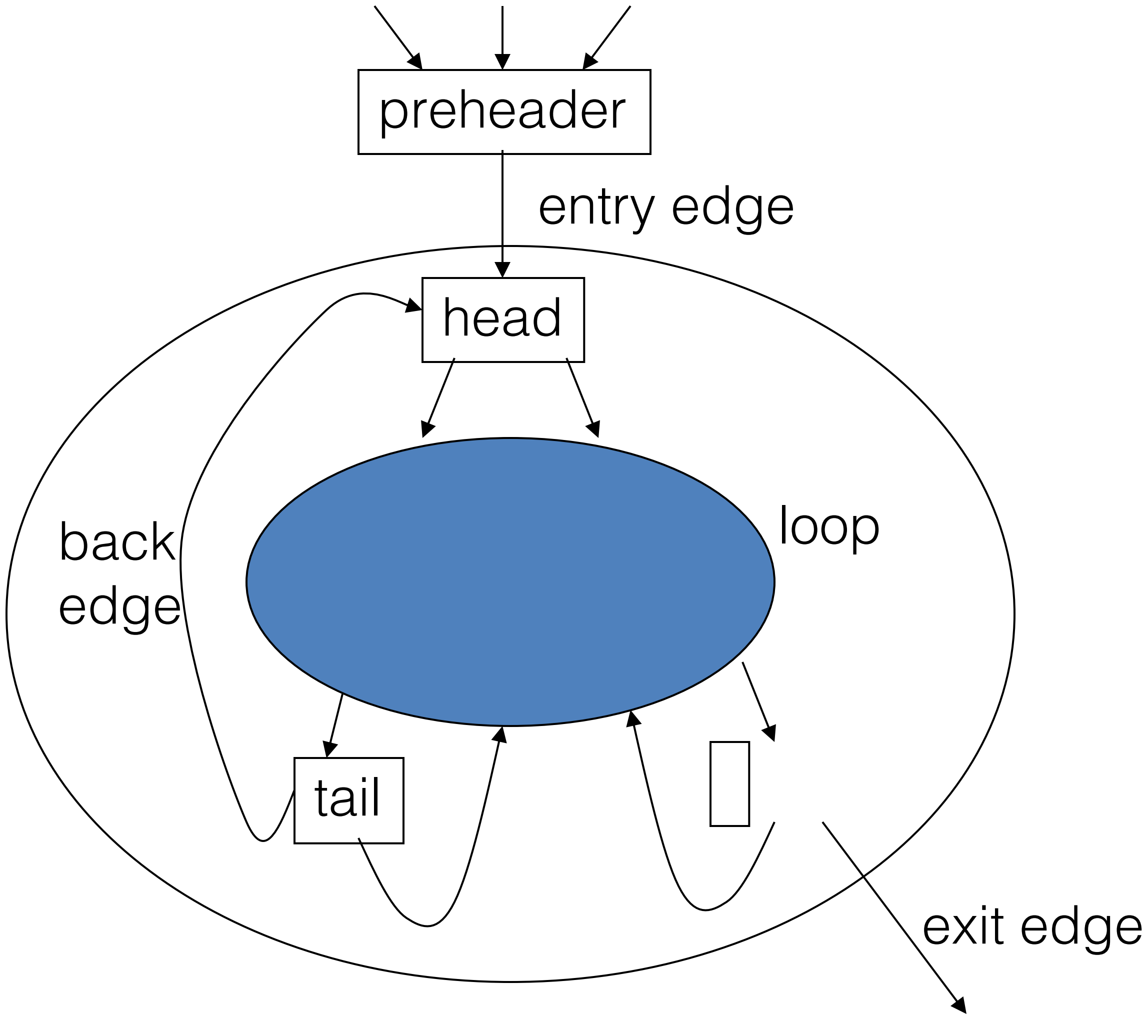
What Is a Loop?

- In a control flow graph, a loop is a set of nodes *S* such that:
  - *S* includes a *header node* h
  - From any node in *S* there is a path of directed edges leading to h
  - There is a path from h to any node in *S*
  - There is no edge from any node outside *S* to any node in *S* other than h

Entries and Exits

- In a loop
  - An *entry node* is one with some predecessor outside the loop
  - An *exit node* is one that has a successor outside the loop
- Corollary: A loop may have multiple exit nodes, but only one entry node

Loop Terminology



Reducible Flow Graphs

- In a reducible flow graph, any two loops are either nested or disjoint
- Roughly, to discover if a flow graph is reducible, repeatedly delete edges and collapse together pairs of nodes (x,y) where x is the only predecessor of y
- If the graph can be reduced to a single node it is reducible
  - Caution: this is the "powerpoint" version of the definition – see a good compiler book for the careful details

Reducible Flow Graphs in Practice

- Common control-flow constructs yield reducible flow graphs
  - if-then[-else], while, do, for, break(!)
- A C function without goto will always be reducible
- Many dataflow analysis algorithms are very efficient on reducible graphs, but…
- We don't need to assume reducible control-flow graphs to handle loops

Finding Loops in Flow Graphs

- We use *dominators* for this
- Recall
  - Every control flow graph has a unique start node s_0
  - Node x dominates node y if every path from s_0 to y must go through x
  - A node x dominates itself

Calculating Dominator Sets

- D[n] is the set of nodes that dominate n
  - $D[s_0] = \{ s_0 \}$
  - $D[n] = \{ n \} \cup (\cap_{p \in pred[n]} D[p])$
- Set up an iterative analysis as usual to solve this
  - Except initially each D[n] must be all nodes in the graph – updates make these sets smaller if changed

Immediate Dominators

- Every node n has a single *immediate dominator* idom(n)
  - idom(n) dominates n
  - idom(n) differs from n – i.e., strictly dominates
  - idom(n) does not dominate any other strict dominator of n
    - i.e., strictly dominates and is nearest dominator
- Fact (er, theorem): If a dominates n and b dominates n, then either a dominates b or b dominates a

  ∴ idom(n) is unique

Dominator Tree

- A *dominator tree* is constructed from a flowgraph by drawing an edge form every node in n to idom(n)
  - This will be a tree. Why?

Back Edges & Loops

- A flow graph edge from a node n to a node h that dominates n is a *back edge*
- For every back edge there is a corresponding subgraph of the flow graph that is a loop

Natural Loops

- If h dominates n and n->h is a back edge, then the *natural loop* of that back edge is the set of nodes x such that
  - h dominates x
  - There is a path from x to n not containing h
- h is the *header* of this loop
- Standard loop optimizations can cope with loops whether they are natural or not

Inner Loops

- Inner loops are more important for optimization because most execution time is expected to be spent there
- If two loops share a header, it is hard to tell which one is "inner"
  - Common way to handle this is to merge natural loops with the same header

Inner (nested) loops

- Suppose
  - A and B are loops with headers a and b
  - $a \neq b$
  - b is in A
- Then
  - The nodes of B are a proper subset of A
  - B is nested in A, or B is the *inner loop*

Loop-Nest Tree

- Given a flow graph G
  1. Compute the dominators of G
  2. Construct the dominator tree
  3. Find the natural loops (thus all loop-header nodes)
  4. For each loop header h, merge all natural loops of h into a single loop: loop[h]
  5. Construct a tree of loop headers s.t. h_1 is above h_2 if h_2 is in loop[h_1]

Loop-Nest Tree Details

- Leaves of this tree are the innermost loops
- Need to put all non-loop nodes somewhere
  - Convention: lump these into the root of the loop-nest tree

Loop Preheader

- Often we need a place to park code right before the beginning of a loop
- Easy if there is a single node preceding the loop header h
  - But this isn't the case in general
- So insert a *preheader* node p
  - Include an edge p->h
  - Change all edges x->h to be x->p

Loop-Invariant Computations

- Idea: If x := a1 op a2 always does the same thing each time around the loop, we'd like to *hoist* it and do it once outside the loop
- But can't always tell if a1 and a2 will have the same value
  - Need a conservative (safe) approximation

Loop-Invariant Computations

- d: x := a1 op a2 is *loop-invariant* if for each a_i
  - a_i is a constant, or
  - All the definitions of a_i that reach d are outside the loop, or
  - Only one definition of a_i reaches d, and that definition is loop invariant
- Use this to build an iterative algorithm
  - Base cases: constants and operands defined outside the loop
  - Then: repeatedly find definitions with loop-invariant operands

Hoisting

- Assume that d: x := a1 op a2 is loop invariant. We can hoist it to the loop preheader if:
  - d dominates all loop exits where x is live-out, and
  - There is only one definition of x in the loop, and
  - x is not live-out of the loop preheader
- Need to modify this if a1 op a2 could have side effects or raise an exception

Hoisting: Possible?

- Example 1

```
L0: t := 0
L1: i := i + 1
d:  t := a op b
    M[i] := t
    if i < n goto L1
L2: x := t
```

- Example 2

```
L0: t := 0
L1: if i ≥ n goto L2
    i := i + 1
d:  t := a op b
    M[i] := t
    goto L1
L2: x := t
```

Hoisting: Possible?

- Example 3

```
L0: t := 0
L1: i := i + 1
d:  t := a op b
    M[i] := t
    t := 0
    M[j] := t
    if i < n goto L1
L2: x := t
```

- Example 4

```
L0: t := 0
L1: M[j] := t
    i := i + 1
d:  t := a op b
    M[i] := t
    if i < n goto L1
L2: x := t
```

Induction Variables

- Suppose inside a loop
  - Variable i is incremented or decremented
  - Variable j is set to i*c+d where c and d are loop-invariant
- Then we can calculate j's value without using i
  - Whenever i is incremented by a, increment j by c*a

Example

- Original

```
     s := 0
     i := 0
L1:  if i ≥ n goto L2
     j := i*4
     k := j+a
     x := M[k]
     s := s+x
     i := i+1
     goto L1
L2:
```

- To optimize, do…
  - Induction-variable analysis to discover i and j are related induction variables
  - Strength reduction to replace *4 with an addition
  - Induction-variable elimination to replace i ≥ n
  - Assorted copy propagation

Result

- Original

```
     s := 0
     i := 0
L1:  if i ≥ n goto L2
     j := i*4
     k := j+a
     x := M[k]
     s := s+x
     i := i+1
     goto L1
L2:
```

- Transformed

```
     s := 0
     k' = a
     b = n*4
     c = a+b
L1:  if k' ≥ c goto L2
     x := M[k']
     s := s+x
     k' := k'+4
     goto L1
L2:
```

Details are somewhat messy – see your favorite compiler book

Basic and Derived Induction Variables

- Variable i is a *basic induction variable* in loop L with header h if the only definitions of i in L have the form i:=i±c where c is loop invariant
- Variable k is a *derived induction variable* in L if:
  - There is only one definition of k in L of the form k:=j*c or k:=j+d where j is an induction variable and c, d are loop-invariant, *and*
  - if j is a derived variable in the family of i, then:
    - The only definition of j that reaches k is the one in the loop, *and*
    - there is no definition of i on any path between the definition of j and the definition of k

Optimizating Induction Variables

- Strength reduction: if a derived induction variable is defined with j:=i*c, try to replace it with an addition inside the loop
- Elimination: after strength reduction some induction variables are not used or are only compared to loop-invariant variables; delete them
- Rewrite comparisons: If a variable is used only in comparisons against loop-invariant variables and in its own definition, modify the comparison to use a related induction variable

Loop Unrolling

- If the body of a loop is small, much of the time is spent in the "increment and test" code
- Idea: reduce overhead by *unrolling* – put two or more copies of the loop body inside the loop

Loop Unrolling

- Basic idea: Given loop L with header node h and back edges s_i->h
  1. Copy the nodes to make loop L’ with header h’ and back edges s_i’->h’
  2. Change all back edges in L from s_i->h to s_i->h’
  3. Change all back edges in L’ from s_i’->h’ to s_i’->h

Unrolling Algorithm Results

- Before

```
L1: x := M[i]
      s := s + x
      i := i + 4
    if i<n goto L1 else L2
L2:
```

- After

```
L1: x := M[i]
      s := s + x
      i := i + 4
      if i<n goto L1’ else L2
L1’:     x := M[i]
      s := s + x
      i := i + 4
    if i<n goto L1 else L2
L2:
```

Hmmmm….

- Not so great – just code bloat
- But: use induction variables and various loop transformations to clean up

After Some Optimizations

- Before

```
L1: x := M[i]
    s := s + x
    i := i + 4
    if i<n goto L1' else L2
L1':    x := M[i]
    s := s + x
    i := i + 4
  if i<n goto L1 else L2
L2:
```

- After

```
L1: x := M[i]
    s := s + x
    x := M[i+4]
    s := s + x
    i := i + 8
    if i<n goto L1 else L2
L2:
```

Still Broken…

- But in a different, better(?) way
- Good code, but only correct if original number of loop iterations was even
- Fix: add an epilogue to handle the “odd” leftover iteration

Fixed

- Before

```
L1: x := M[i]
      s := s + x
      x := M[i+4]
      s := s + x
      i := i + 8
    if i<n goto L1 else L2
L2:
```

- After

```
      if i<n-8 goto L1 else L2
L1: x := M[i]
      s := s + x
      x := M[i+4]
      s := s + x
      i := i + 8
      if i<n-8 goto L1 else L2
L2: x := M[i]
      s := s+x
      i := i+4
      if i < n goto L2 else L3
L3:
```

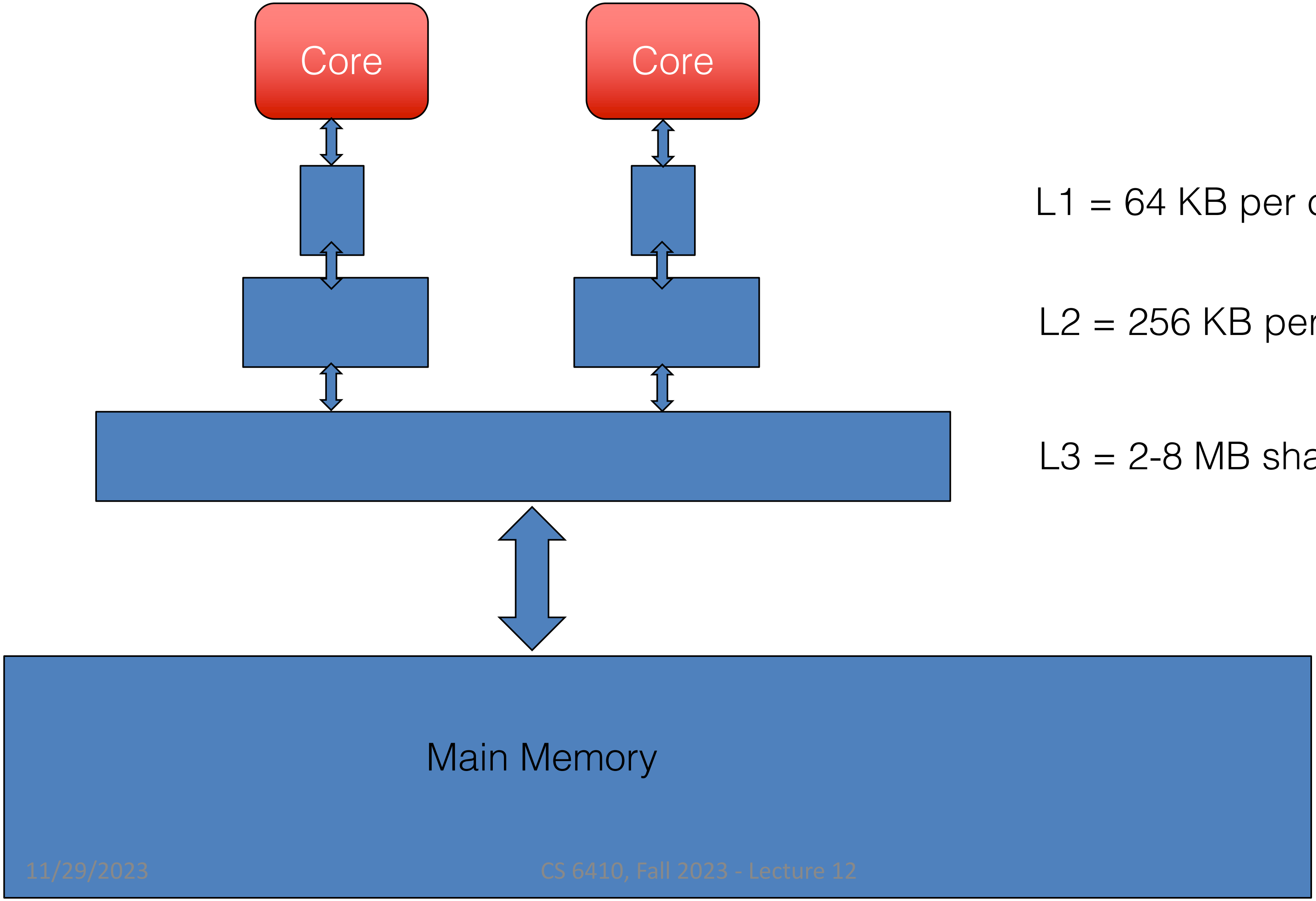
Postscript

- This example only unrolls the loop by a factor of 2
- More typically, unroll by a factor of K
  - Then need an epilogue that is a loop like the original that iterates up to K-1 times

Memory Hierarchies

- One of the great triumphs of computer design
- Effect is a large, fast memory
- Reality is a series of progressively larger, slower, cheaper stores, with frequently accessed data automatically staged to faster storage (cache, main storage, disk)
- Programmer/compiler typically treats it as one large store. (but not always the best idea)
- Hardware maintains cache coherency – most of the time

Intel Haswell Caches



L1 = 64 KB per core

L2 = 256 KB per core

L3 = 2-8 MB shared

Just How Slow is Operand Access?

• Instruction	~5 per cycle
• Register	1 cycle
• L1 CACHE	~4 cycles
• L2 CACHE	~10 cycles
• L3 CACHE (unshared line)	~40 cycles
• DRAM	~100 ns

Memory Issues

- Byte load/store is often slower than whole (physical) word load/store
  - Unaligned access is often extremely slow
- Temporal locality: accesses to recently accessed data will usually find it in the (fast) cache
- Spatial locality: accesses to data near recently used data will usually be fast
  - "near" = in the same cache block
- But – alternating accesses to blocks that map to the same cache block will cause thrashing

- CPU speed increases have out-paced increases in memory access times
- Memory access now often determines overall execution speed
- "Instruction count" is not the only performance metric for optimization

Data Alignment

- Data objects (structs) often are similar in size to a cache block (≈ 64 bytes)
  ∴ Better if objects don't span blocks
- Some strategies:
  - –Allocate objects sequentially; bump to next block boundary if useful
  - –Allocate objects of same common size in separate pools (all size-2, size-4, etc.)
- Tradeoff: speed for some wasted space

Instruction Alignment

- Align frequently executed basic blocks on cache boundaries (or avoid spanning cache blocks)
- Branch targets (particularly loops) may be faster if they start on a cache line boundary
  - Often see multi-byte nops in optimized code as padding to align loop headers
  - How much depends on architecture (current intel 16 bytes, current AMD 32 bytes)
- Try to move infrequent code (startup, exceptions) away from hot code
- Optimizing compiler may perform basic-block ordering

Loop Interchange

- Watch for bad cache patterns in inner loops; rearrange if possible
- Example

```
for (i = 0; i < m; i++)
  for (j = 0; j < n; j++)
    for (k = 0; k < p; k++)
      a[i,k,j] = b[i,j-1,k] + b[i,j,k] + b[i,j+1,k]
```

  – b[i,j+1,k] is reused in the next two iterations, but will have been flushed from the cache by the k loop

Loop Interchange

- Solution for this example: interchange j and k loops

```
for (i = 0; i < m; i++)
  for (k = 0; k < p; k++)
    for (j = 0; j < n; j++)
      a[i,k,j] = b[i,j-1,k] + b[i,j,k] + b[i,j+1,k]
```

  - Now b[i,j+1,k] will be used three times on each cache load
  - Safe here because loop iterations are independent

Loop Interchange

- Need to construct a data-dependency graph showing information flow between loop iterations
- For example, iteration (j,k) depends on iteration (j’,k’) if (j’,k’) computes values used in (j,k) or stores values overwritten by (j,k)
  - If there is a dependency and loops are interchanged, we could get different results – so can’t do it

Blocking

- Consider matrix multiply

```
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++) {
    c[i,j] = 0.0;
    for (k = 0; k < n; k++)
      c[i,j] = c[i,j] + a[i,k]*b[k,j]
  }
```

- If a, b fit in the cache together, great!
- If they don't, then every b[k,j] reference will be a cache miss
- Loop interchange (i<->j) won't help; then every a[i,k] reference would be a miss

Blocking

- Solution: reuse rows of A and columns of B while they are still in the cache
- Assume the cache can hold 2*c*n matrix elements ($1 < c < n$)
- Calculate $c \times c$ blocks of C using c rows of A and c columns of B

Blocking

- Calculating c × c blocks of C

```
for (i = i0; i < i0+c; i++)
  for (j = j0; j < j0+c; j++) {
    c[i,j] = 0.0;
    for (k = 0; k < n; k++)
      c[i,j] = c[i,j] + a[i,k]*b[k,j]
  }
```

Blocking

- Then nest this inside loops that calculate successive c × c blocks

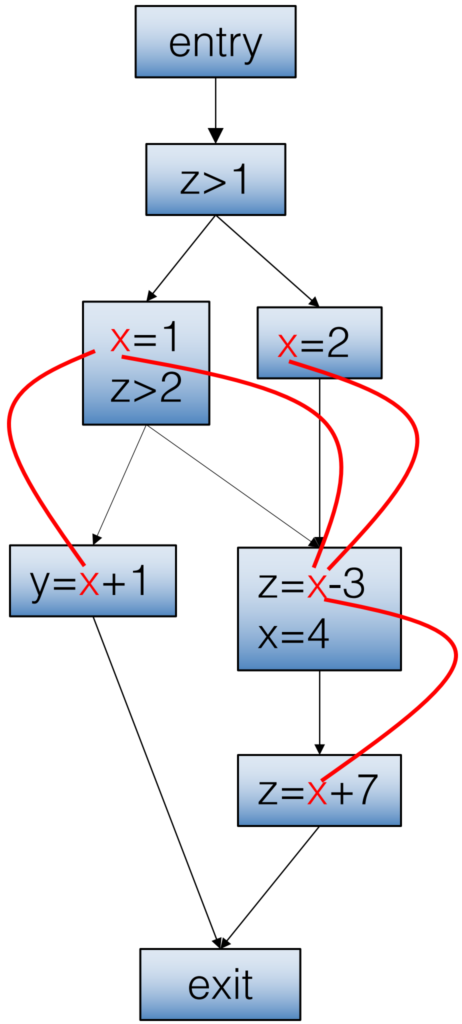
```
for (i0 = 0; i0 < n; i0+=c)
  for (j0 = 0; j0 < n; j0+=c)
    for (i = i0; i < i0+c; i++)
      for (j = j0; j < j0+c; j++) {
        c[i,j] = 0.0;
        for (k = 0; k < n; k++)
          c[i,j] = c[i,j] + a[i,k]*b[k,j]
      }
```

SSA

Def-Use (DU) Chains

- Common dataflow analysis problem: Find all sites where a variable is used, or find the definition site of a variable used in an expression
- Traditional solution: def-use chains – additional data structure on top of the dataflow graph
  - Link each statement defining a variable to all statements that use it
  - Link each use of a variable to its definition

Def-Use (DU) Chains



In this example, two DU chains intersect

DU-Chain Drawbacks

- Expensive: if a typical variable has N uses and M definitions, the total cost *per-variable* is O(N * M), i.e., $O(n^2)$
  - Would be nice if cost were proportional to the size of the program
- Unrelated uses of the same variable are mixed together
  - Complicates analysis – variable looks live across all uses even if unrelated

SSA: Static Single Assignment

- IR where each variable has only one definition in the program text
  - This is a single *static* definition, but that definition can be in a loop that is executed dynamically many times
- Makes many analyses (and associated optimizations) more efficient
- Separates values from memory storage locations
- Complementary to CFG/DFG – better for some things, but cannot do everything

SSA in Basic Blocks

Idea: for each original variable x, create a new variable x_n at the n^{th} definition of the original x. Subsequent uses of x use x_n until the next definition point.

- Original
  - $a := x + y$
  - $b := a - 1$
  - $a := y + b$
  - $b := x * 4$
  - $a := a + b$
- SSA
  - $a_1 := x + y$
  - $b_1 := a_1 - 1$
  - $a_2 := y + b_1$
  - $b_2 := x * 4$
  - $a_3 := a_2 + b_2$

Merge Points

- The issue is how to handle merge points

if (...)
  a = x;
else
  a = y;
b = a;

if (...)
  a_1 = x;
else
  a_2 = y;
b_1 = ??;

Merge Points

- The issue is how to handle merge points

```
if (...)
  a = x;
else
  a = y;
b = a;
```

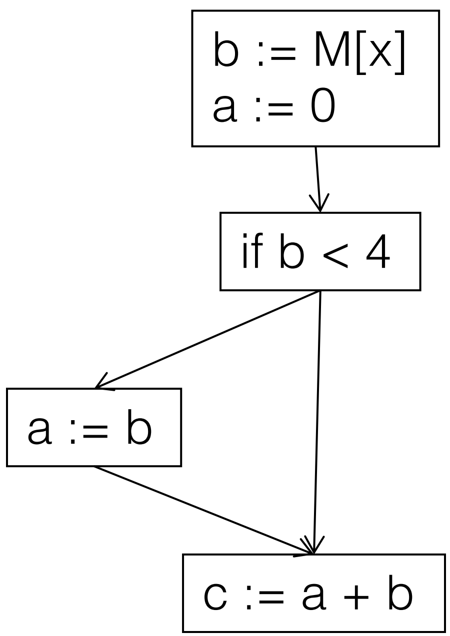
$\rightarrow$

$$
\begin{array}{l}
\text{if } (\ldots) \\
\quad a_1 = x; \\
\text{else} \\
\quad a_2 = y; \\
a_3 = \Phi(a_1, a_2); \\
b_1 = a_3;
\end{array}
$$

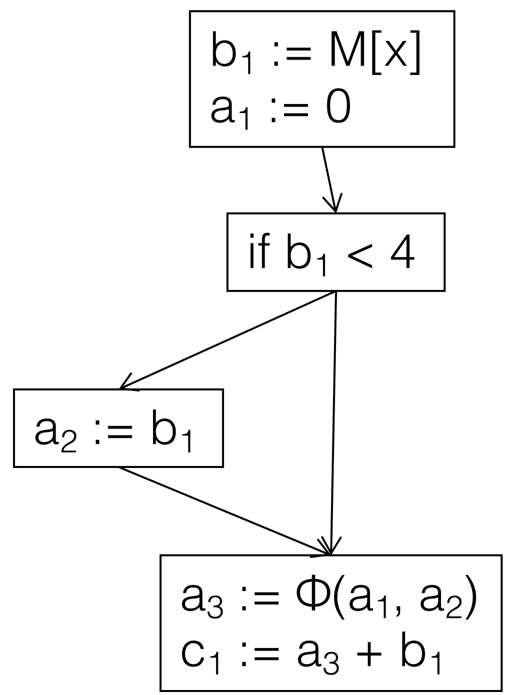
- Solution: introduce a Φ-function
  $a_3 := \Phi(a_1, a_2)$
- Meaning: a_3 is assigned either a_1 or a_2 depending on which control path is used to reach the Φ-function

Another Example

Original



SSA

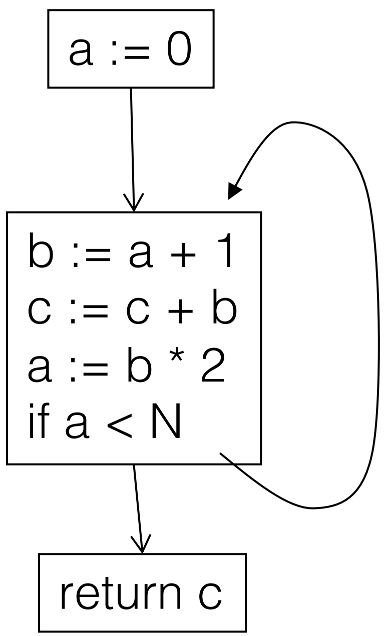


How Does Φ “Know” What to Pick?

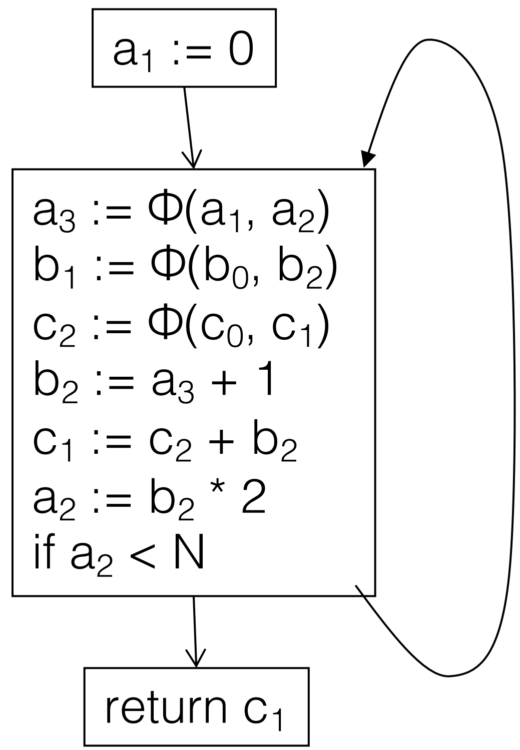
- It doesn’t
- Φ-functions don’t actually exist at runtime
  - When we’re done using the SSA IR, we translate back out of SSA form, removing all Φ-functions
    - Basically by adding code to copy all SSA x_i values to the single, non-SSA, actual x
  - For analysis, all we typically need to know is the connection of uses to definitions – no need to “execute” anything

Example With a Loop

Original



SSA



Notes:

- Loop back edges are also merge points, so require Φ-functions
- a_0, b_0, c_0 are initial values of a, b, c on block entry
- b_1 is dead – can delete later
- c is live on entry – either input parameter or uninitialized

What does SSA “get” us?

- No need for DU or UD chains – implicit in SSA
- Compact representation
- SSA is “recent” (i.e., 80s)
- Prevalent in real compilers for { } languages

Converting To SSA Form

- Basic idea
  - First, add Φ-functions
  - Then, rename all definitions and uses of variables by adding subscripts

Inserting Φ-Functions

- Could simply add Φ-functions for every variable at every join point(!)
- Called “maximal SSA”
- But
  – Wastes *way* too much space and time
  – Not needed in many cases

Path-convergence criterion

- Insert a Φ-function for variable a at point z when:
  - There are blocks x and y, both containing definitions of a, and $x \neq y$
  - There are nonempty paths from x to z and from y to z
  - These paths have no common nodes other than z

Details

- The start node of the flow graph is considered to define every variable (even if "undefined")
- Each Φ-function itself defines a variable, which may create the need for a new Φ-function
  - So we need to keep adding Φ-functions until things converge
- How can we do this efficiently?
  Use a new concept: dominance frontiers

Dominators - Review

- Definition: a block x *dominates* a block y if and only if every path from the entry of the control-flow graph to y includes x
- So, by definition, x dominates x

Dominators and SSA

- One property of SSA is that definitions dominate uses; more specifically:
  - If $x := \Phi(\ldots, x_i, \ldots)$ is in block B, then the definition of x_i dominates the i^{th} predecessor of B
  - If x is used in a non-Φ statement in block B, then the definition of x dominates block B

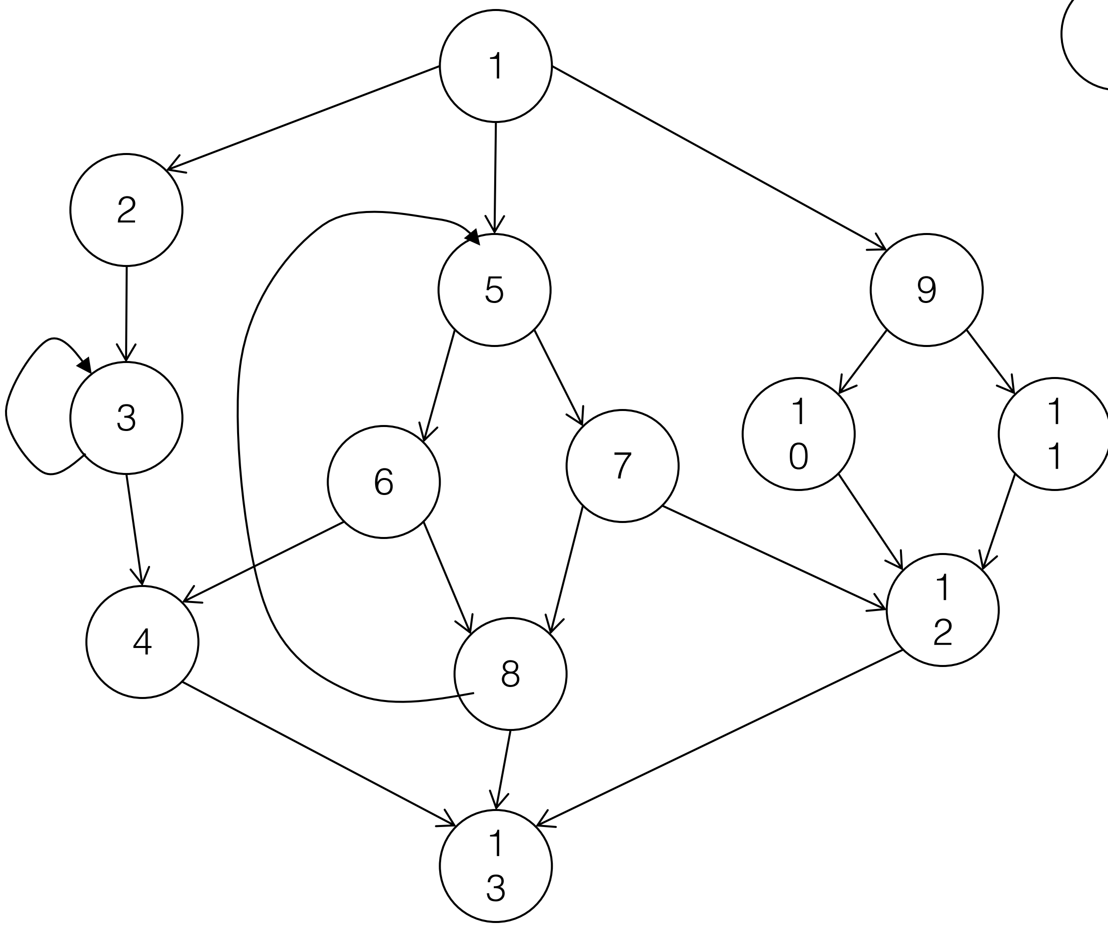
Dominance Frontier (1)

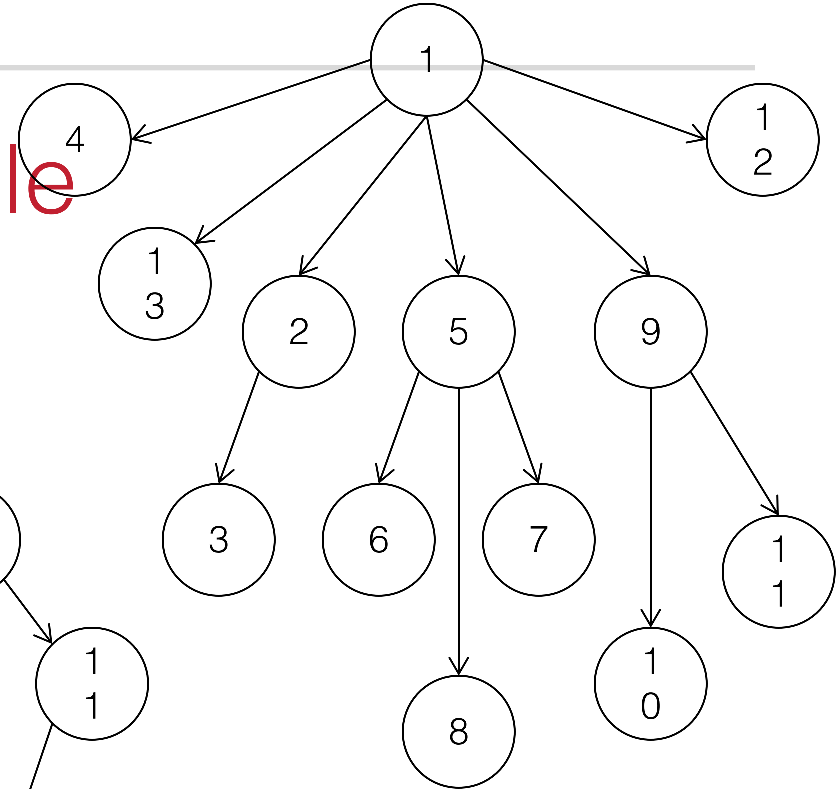
- To get a practical algorithm for placing Φ-functions, we need to avoid looking at all combinations of nodes leading from x to y
- Instead, use the dominator tree in the flow graph

Dominance Frontier (2)

- Definitions
  - x *strictly dominates* y if x dominates y and x ≠ y
  - The *dominance frontier* of a node x is the set of all nodes w such that x dominates a predecessor of w, but x does not strictly dominate w
    - This means that x can be in *it's own* dominance frontier! That can happen if there is a back edge to x (i.e., x is the head of a loop)
- Essentially, the dominance frontier is the border between dominated and undominated nodes

Example



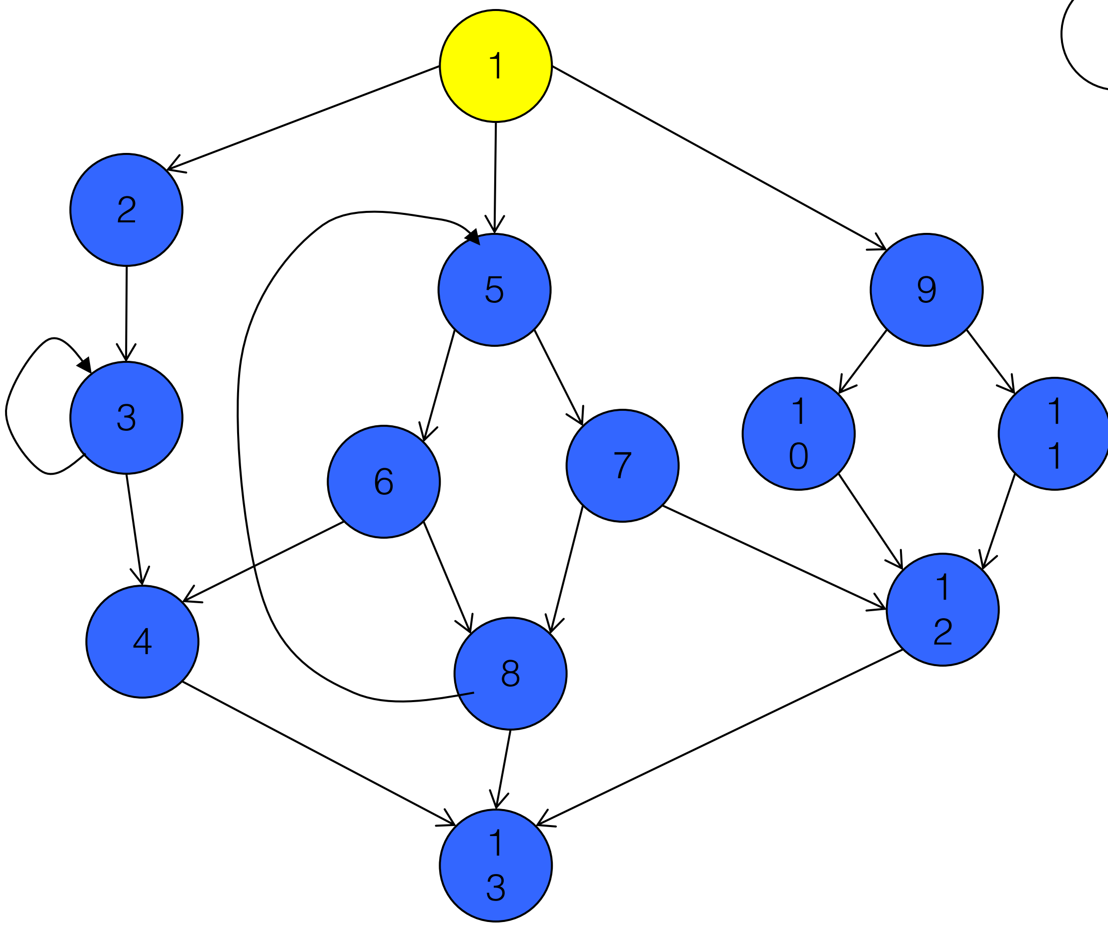


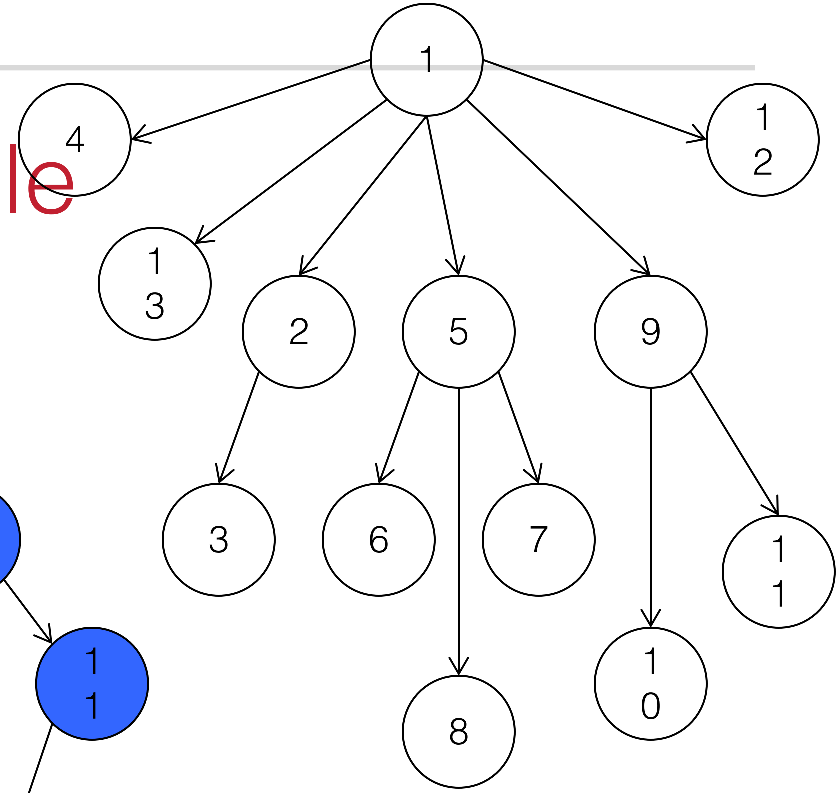
= x

= DomFrontier(x)

= StrictDom(x)

Example



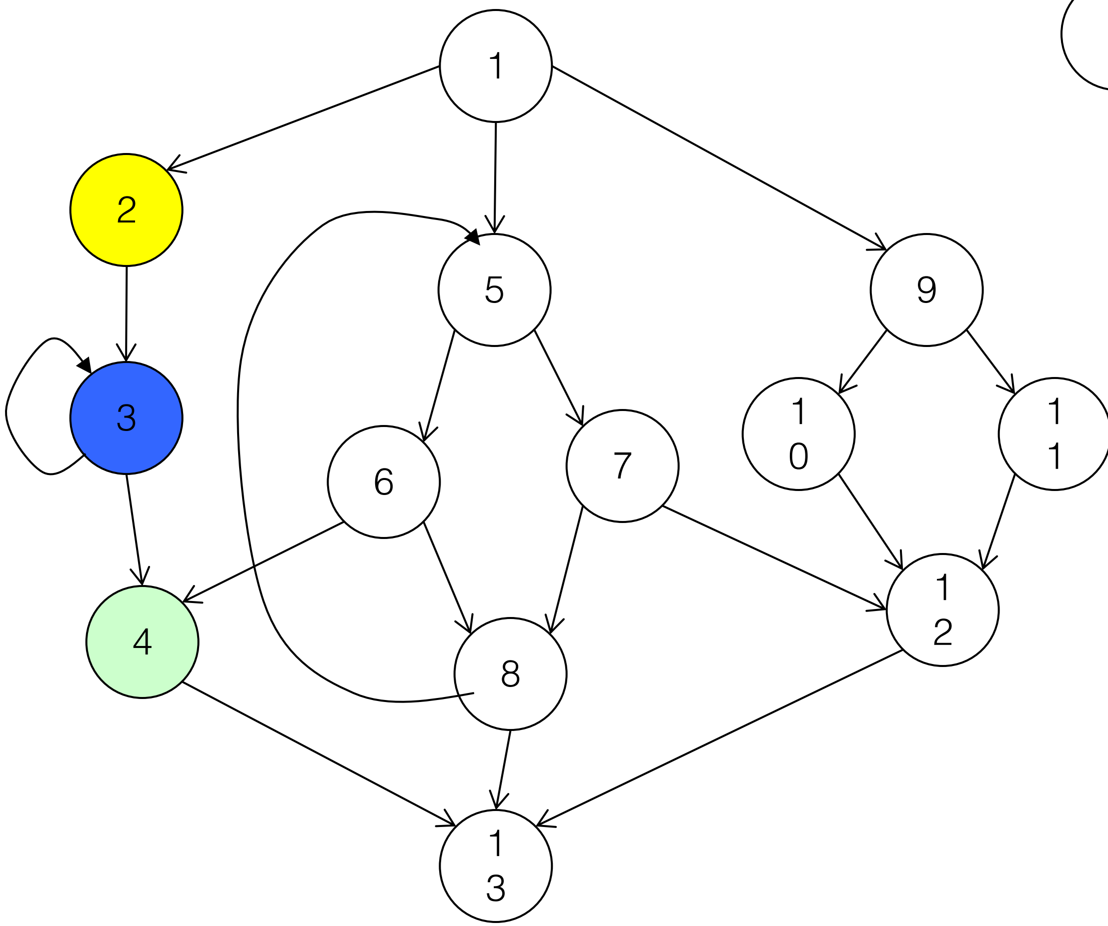


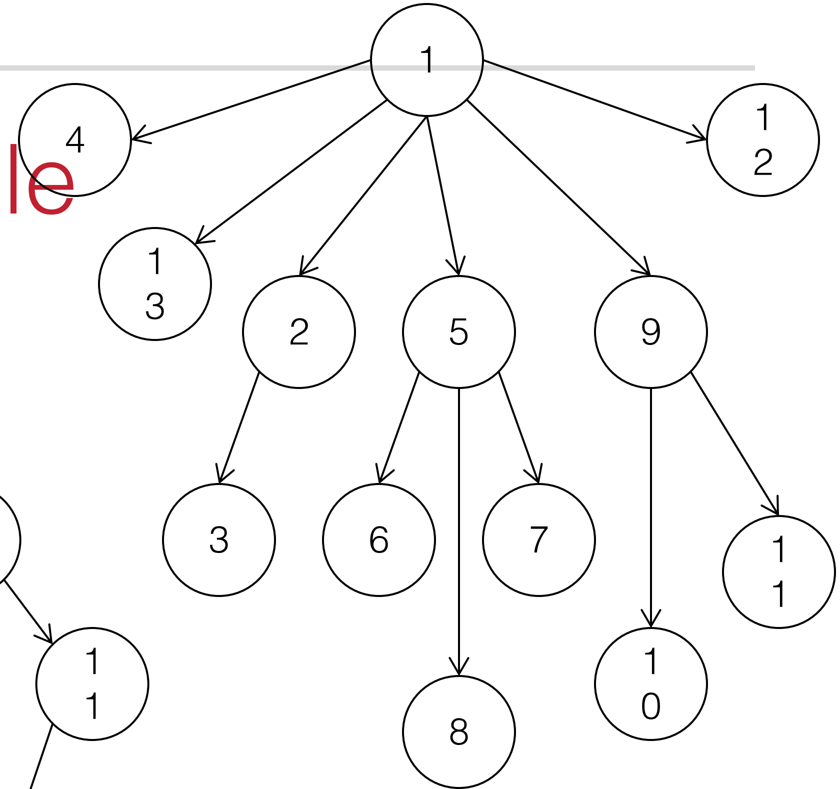
= x

= DomFrontier(x)

= StrictDom(x)

Example



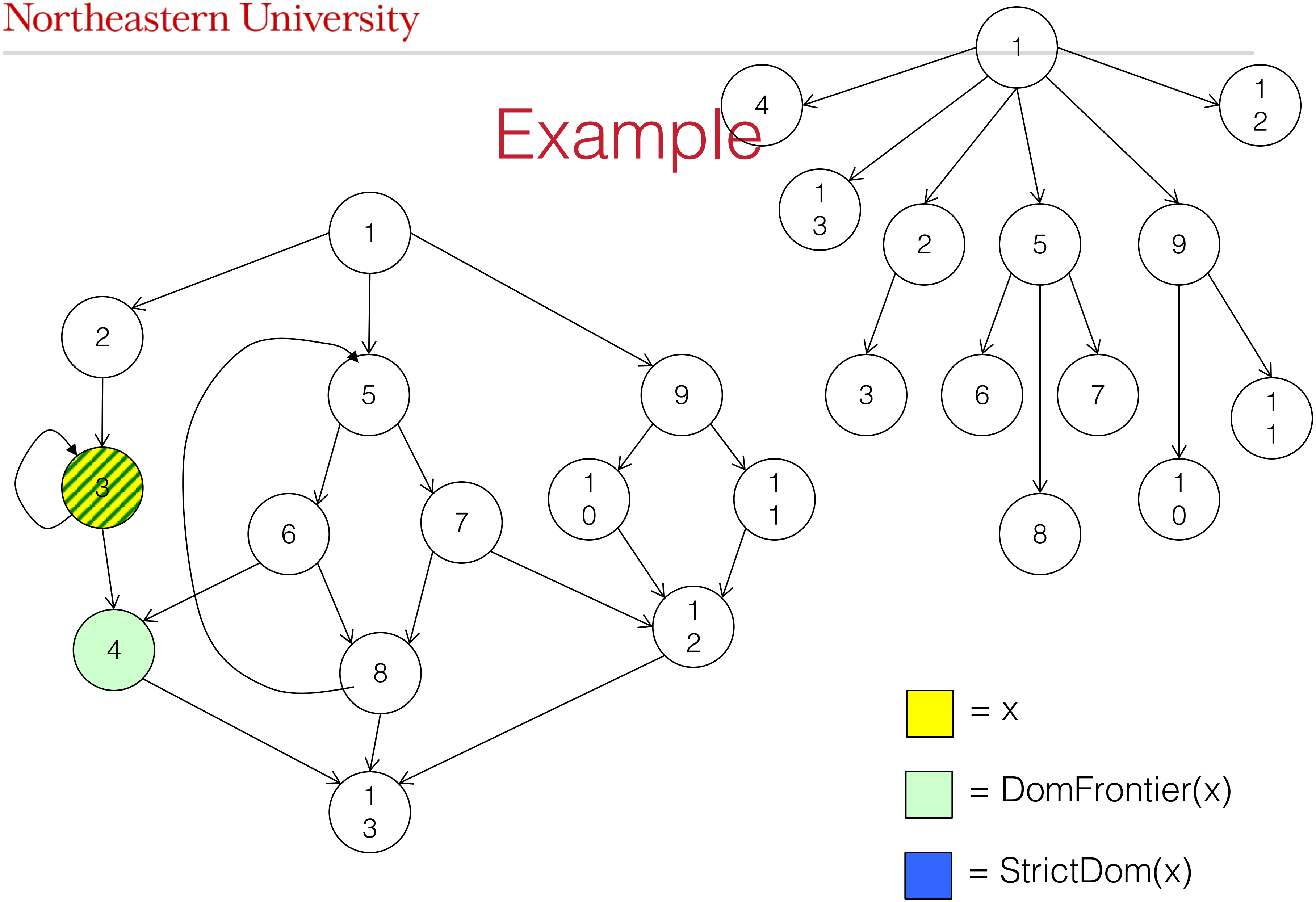


= x

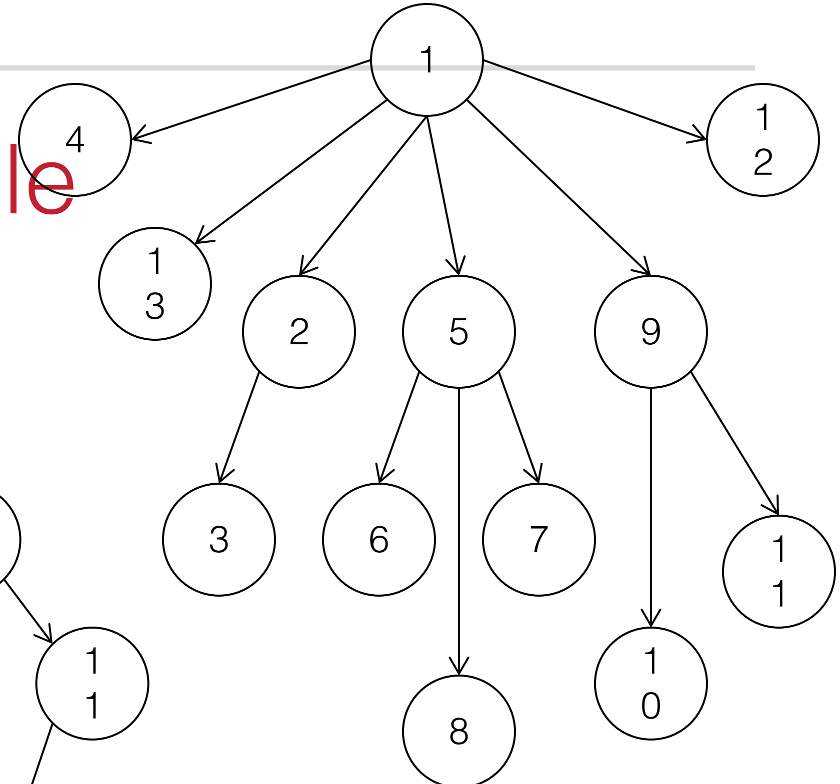
= DomFrontier(x)

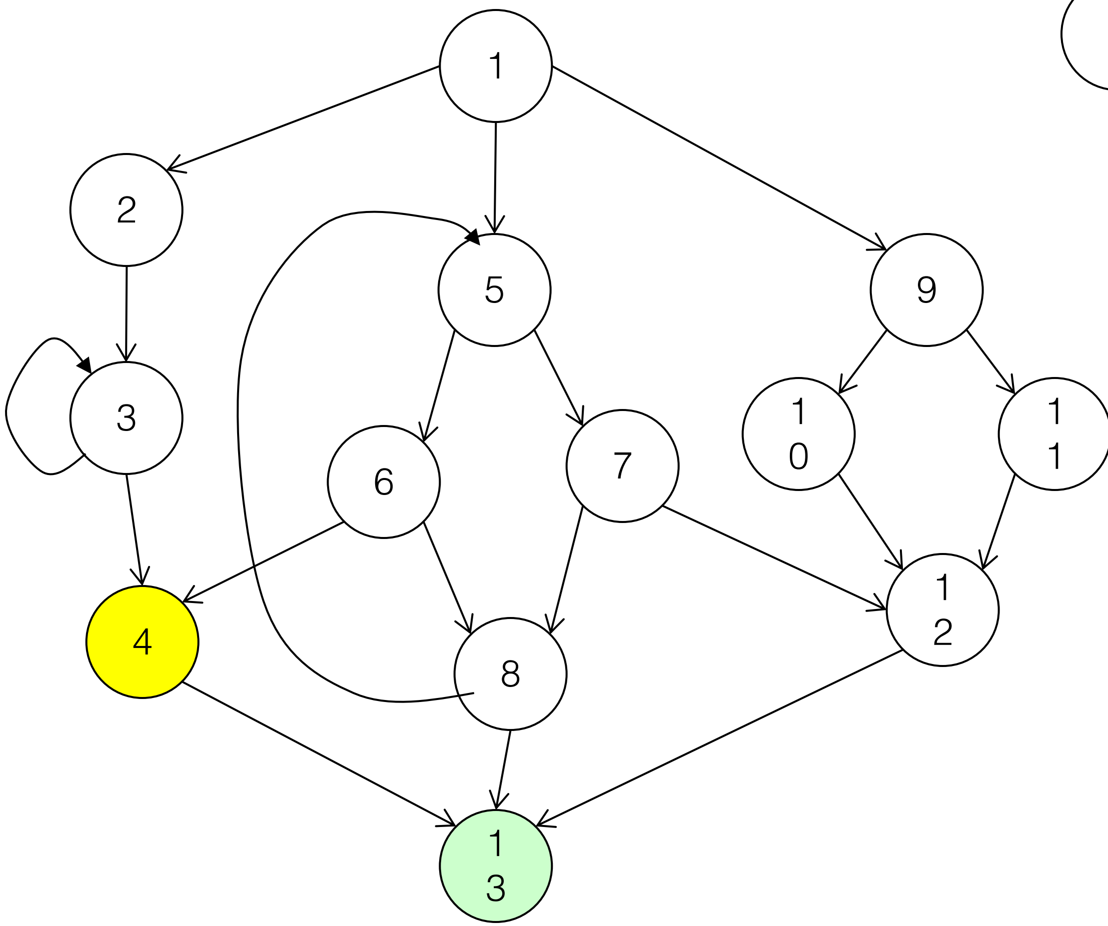
= StrictDom(x)

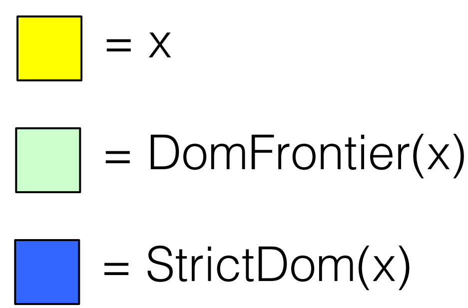
Example



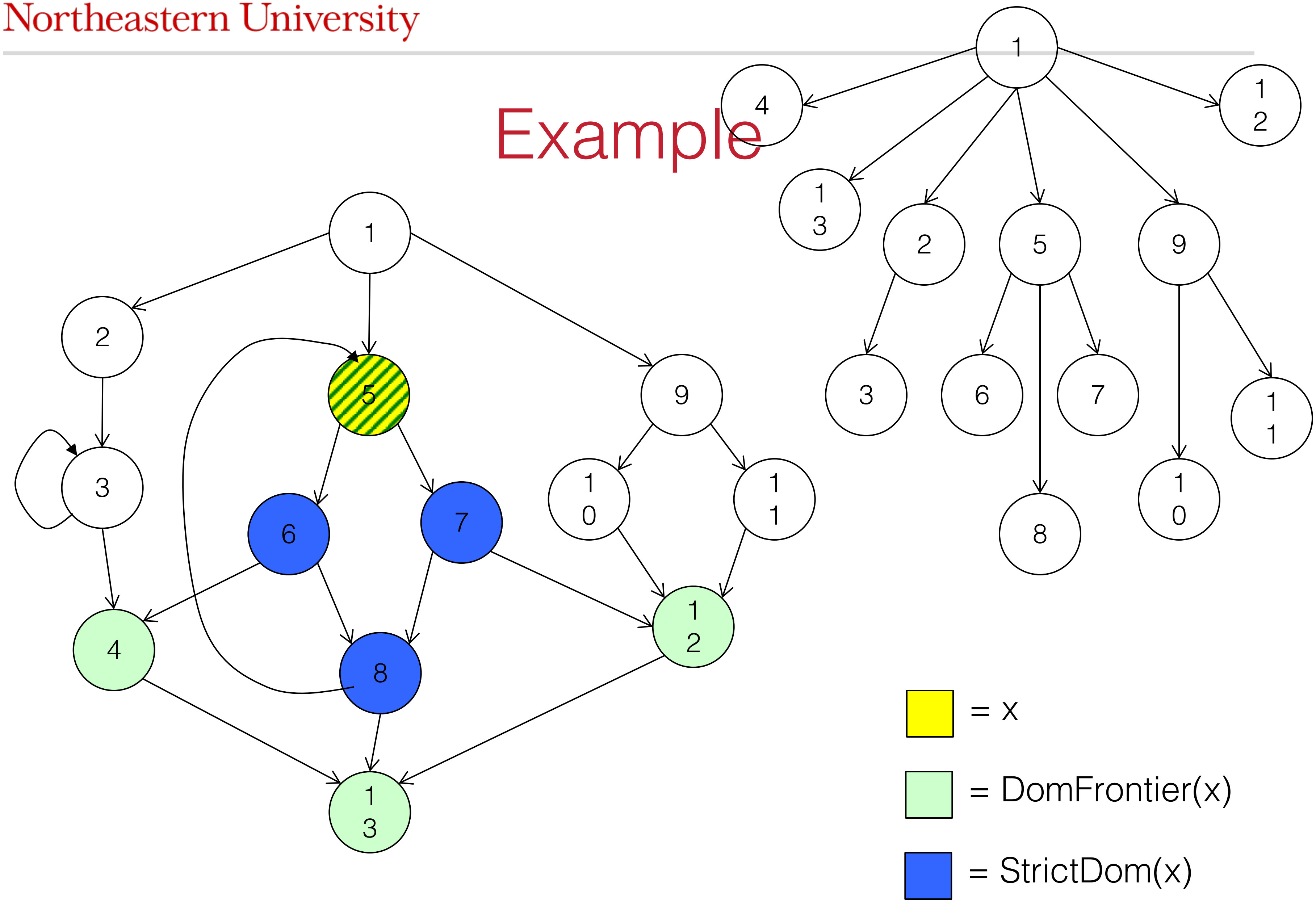
Example


1
4
1 2
1 3
2
5
9
3
6
7
1 1
8
1 0

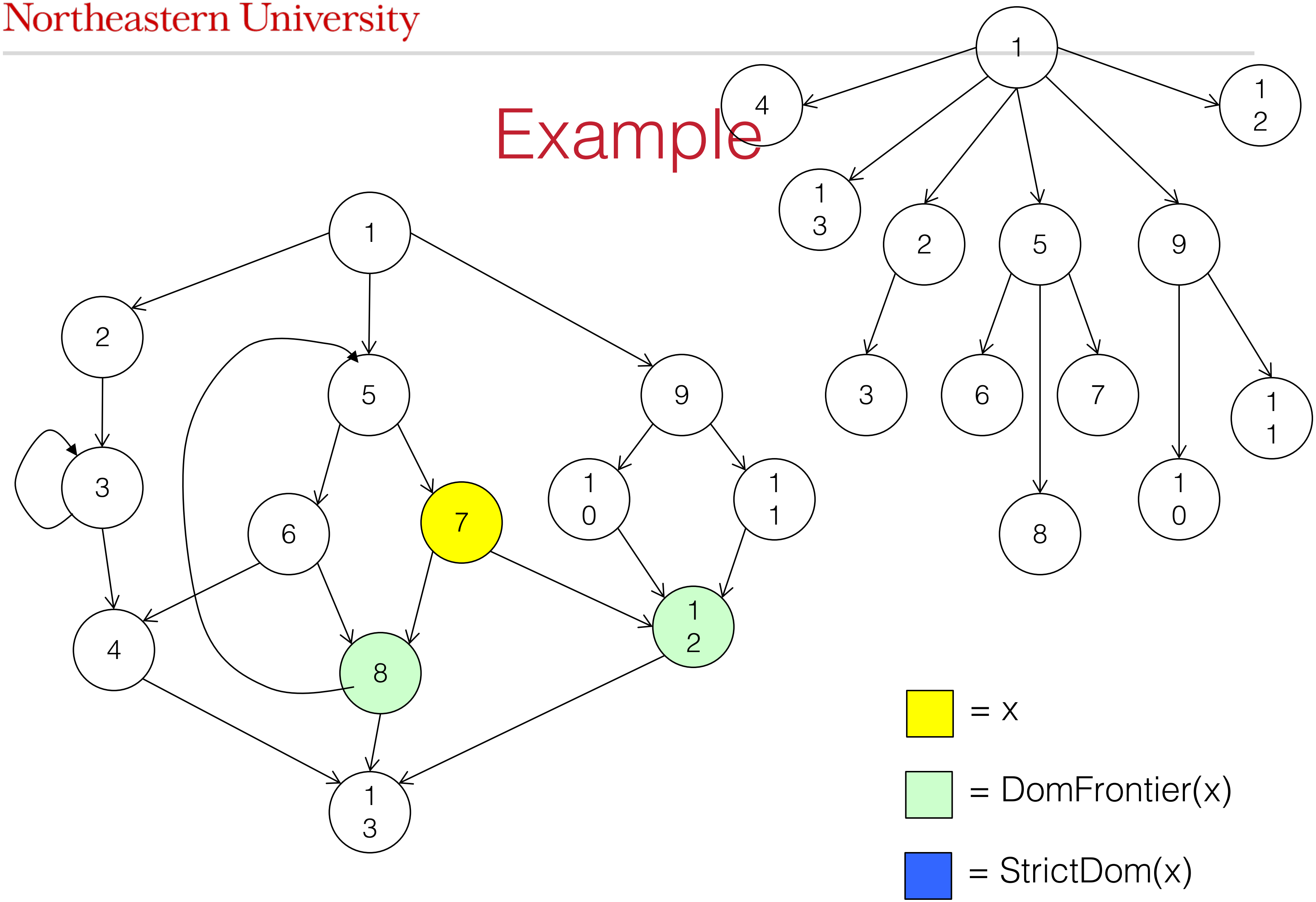

1
2
5
9
3
6
7
1 0
1 1
4
8
1 2
1 3


= x
= DomFrontier(x)
= StrictDom(x)

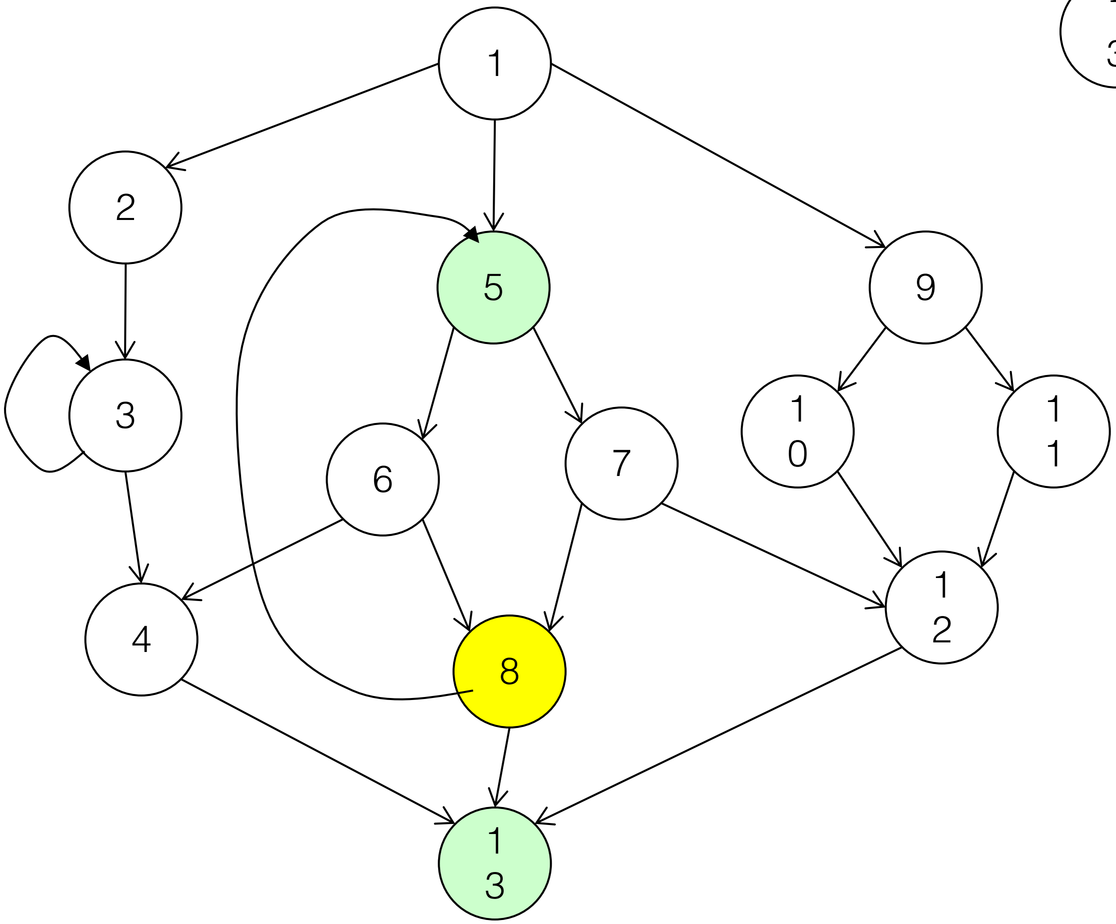
Example

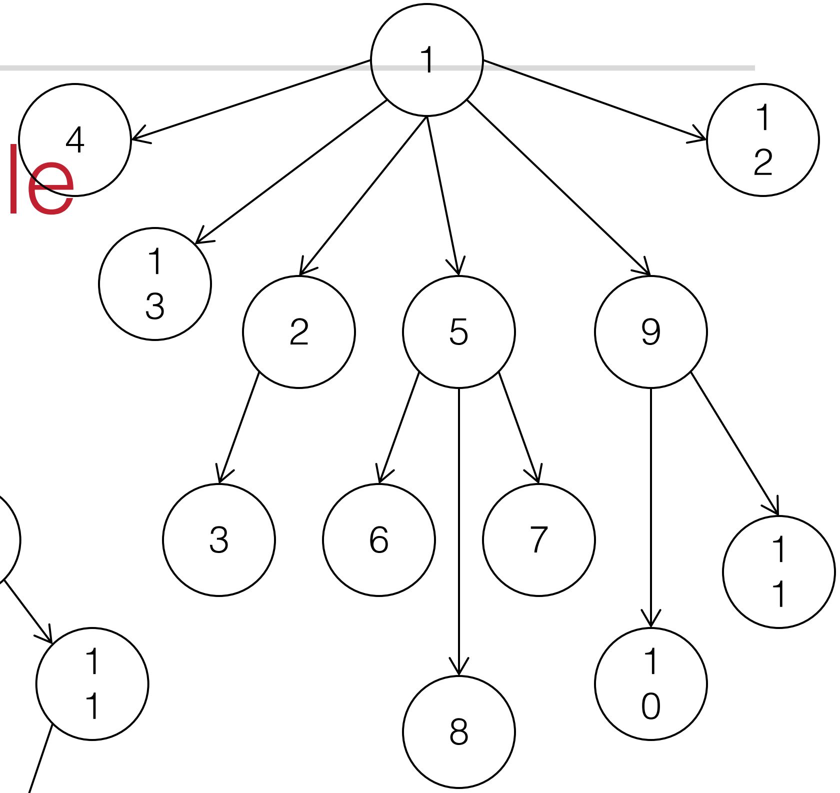


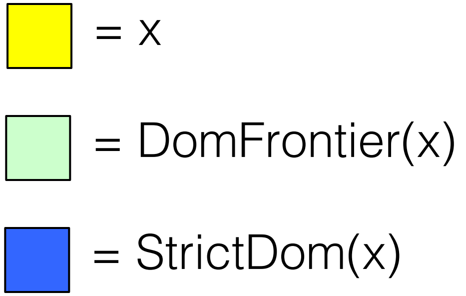
Example


1
2
3
4
5
6
7
8
9
1 0
1 1
1 2
1 3
1
4
1 2
1 3
2
5
9
3
6
7
1 1
8
1 0
= X
= DomFrontier(x)
= StrictDom(x)

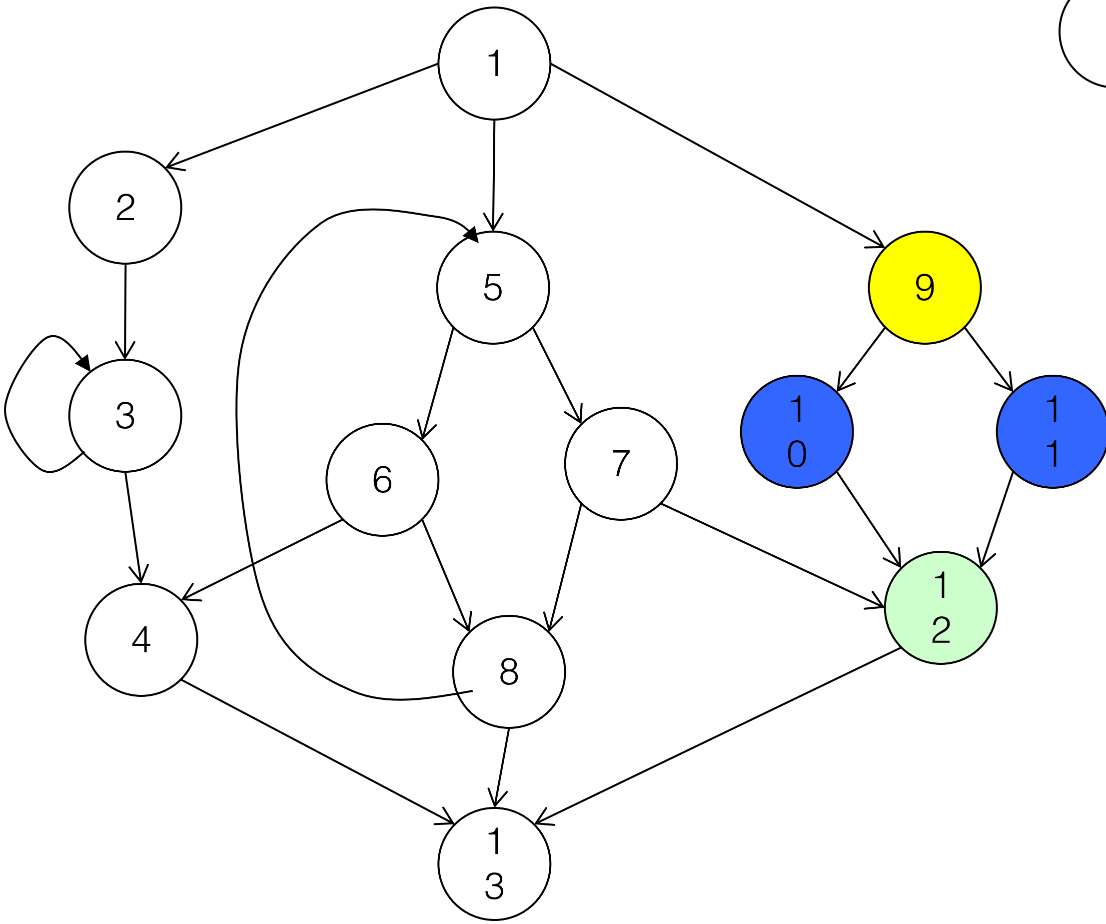
Example

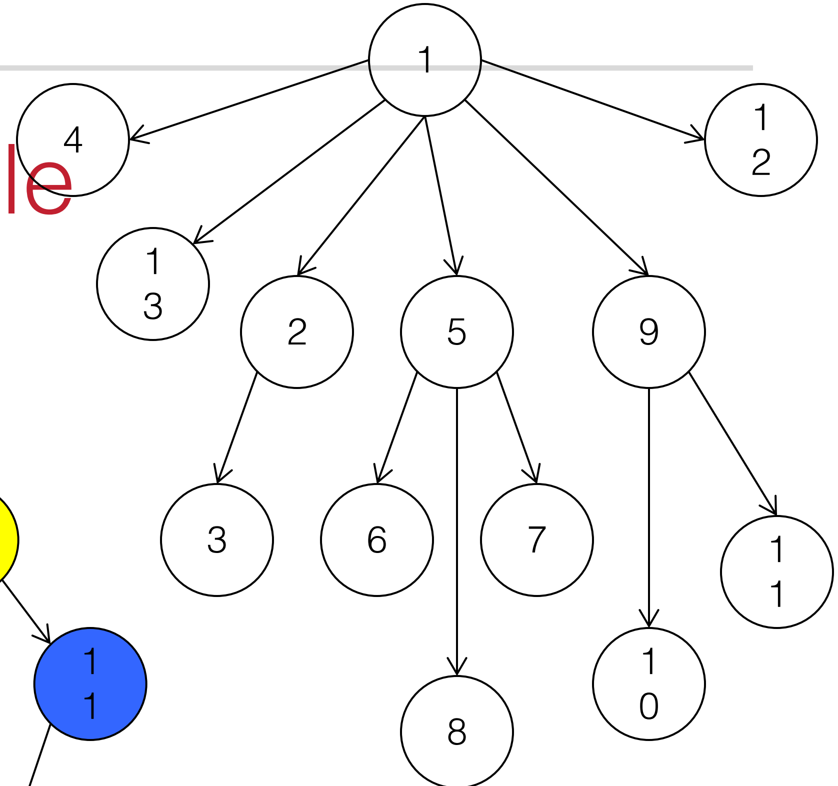

1
2
3
4
5
6
7
8
9
10
11
12
13


1
4
12
13
2
5
9
3
6
7
11
8
10


= X
= DomFrontier(x)
= StrictDom(x)

Example



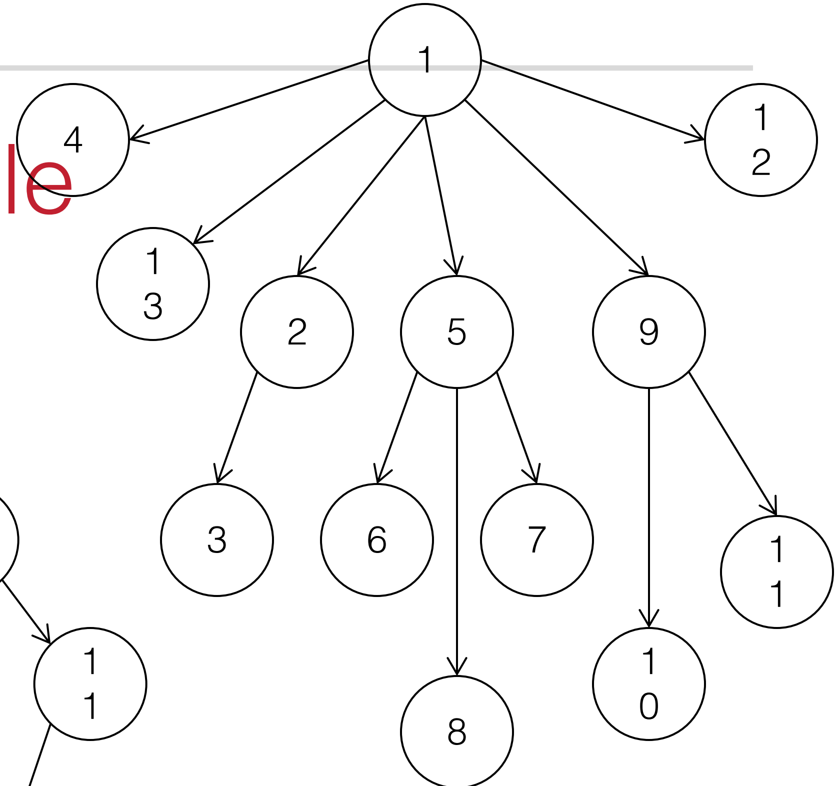


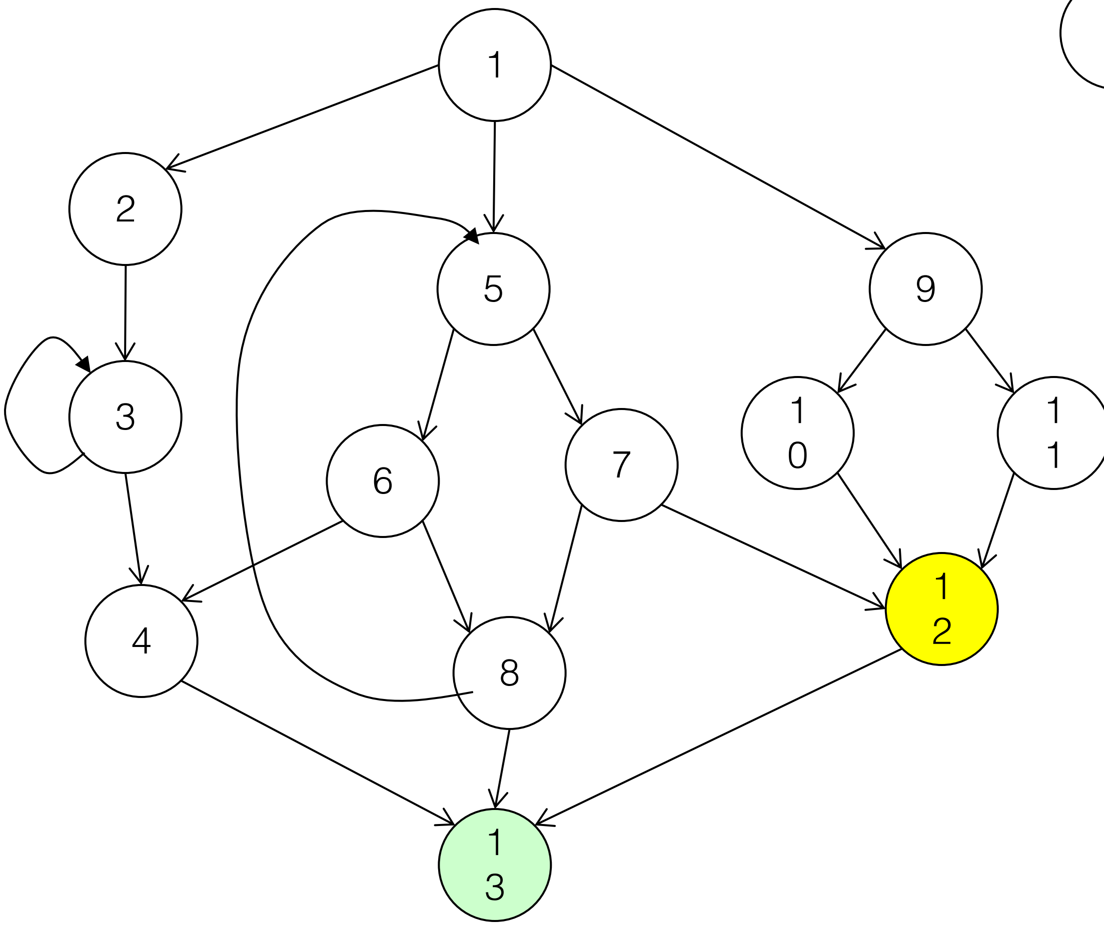
= x

= DomFrontier(x)

= StrictDom(x)

Example



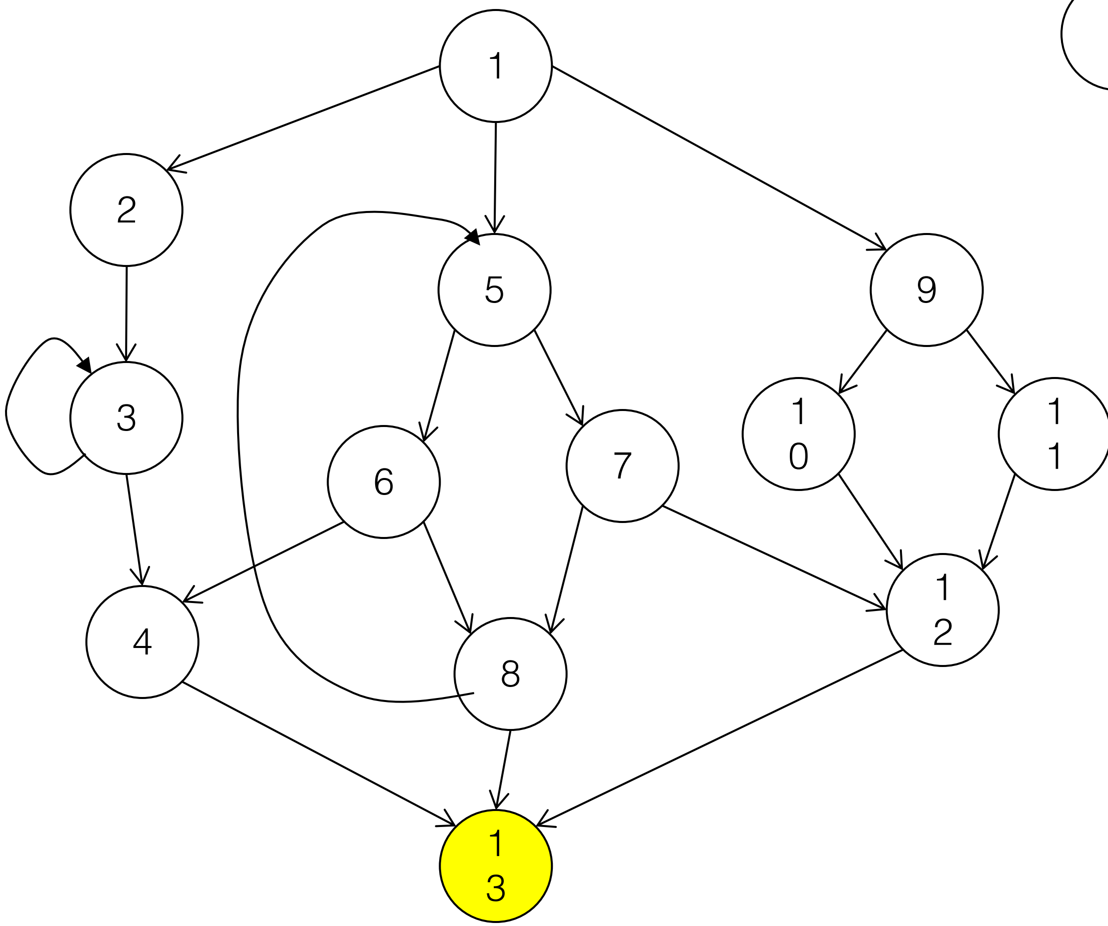


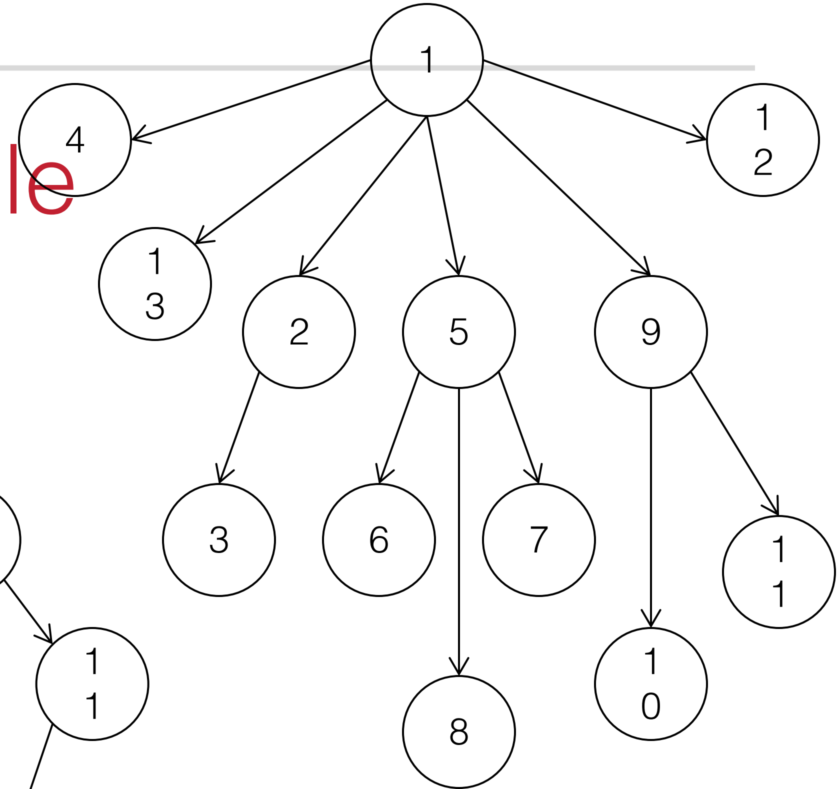
= x

= DomFrontier(x)

= StrictDom(x)

Example





= x

= DomFrontier(x)

= StrictDom(x)

Dominance Frontier Criterion for Placing Φ-Functions

- If a node x contains the definition of variable a, then every node in the dominance frontier of x needs a Φ-function for a
  - Idea: Everything dominated by x will see x's definition of a. The dominance frontier represents the first nodes we could have reached via an alternative path, which *will* have an alternate reaching definition (recall we say the entry node defines everything)
    - Why is this right for loops? Hint: strict dominance…
  - Since the Φ-function itself is a definition, this placement rule needs to be iterated until it reaches a fixed-point
- Theorem: this algorithm places exactly the same set of Φ-functions as the path criterion given previously

Placing Φ-Functions: Details

- See the book for the full construction, but the basic steps are:
  1. Compute the dominance frontiers for each node in the flowgraph
  2. Insert just enough Φ-functions to satisfy the criterion. Use a worklist algorithm to avoid reexamining nodes unnecessarily
  3. Walk the dominator tree and rename the different definitions of each variable a to be a_1, a_2, a_3, …

Efficient Dominator Tree Computation

- Goal: SSA makes optimizing compilers faster since we can find definitions/uses without expensive bit-vector algorithms
- So, need to be able to compute SSA form quickly
- Computation of SSA from dominator trees are efficient, but…

Lengauer-Tarjan Algorithm

- Iterative set-based algorithm for finding dominator trees is slow in worst case
- Lengauer-Tarjan is near linear time
  - Uses depth-first spanning tree from start node of control flow graph
  - See books for details

SSA Optimizations

- Why go to the trouble of translating to SSA?
- The advantage of SSA is that it makes many optimizations and analyses simpler and more efficient
  - We'll give a couple of examples
- But first, what do we know? (i.e., what information is kept in the SSA graph?)

SSA Data Structures

- Statement: links to containing block, next and previous statements, variables defined, variables used.
- Variable: link to its (single) definition and (possibly multiple) use sites
- Block: List of contained statements, ordered list of predecessors, successor(s)

Dead-Code Elimination

- A variable is live ⇔ its list of uses is not empty(!)
  - That's it! Nothing further to compute
- Algorithm to delete dead code:

```
while there is some variable v with no uses
    if the statement that defines v has no
            other side effects, then delete it
```

  - Need to remove this statement from the list of uses for its operand variables – which may cause those variables to become dead

Sparse Simple Constant Propagation

- If c is a constant in v := c, any use of v can be replaced by c
  - Then update every use of v to use constant c
- If the c_i's in $v := \Phi(c_1, c_2, \ldots, c_n)$ are all the same constant c, we can replace this with v := c
- Incorporate copy propagation, constant folding, and others in the same worklist algorithm

Simple Constant Propagation

```
W := list of all statements in SSA program
while W is not empty
  remove some statement S from W
  if S is v:=Φ(c, c, …, c), replace S with v:=c
  if S is v:=c
     delete S from the program
     for each statement T that uses v
       substitute c for v in T
       add T to W
```

Converting Back from SSA

- Unfortunately, real machines do not include a Φ instruction
- So after analysis, optimization, and transformation, need to convert back to a “Φ-less” form for execution

Translating Φ-functions

- The meaning of $x := \Phi(x_1, x_2, \ldots, x_n)$ is "set $x := x_1$ if arriving on edge 1, set $x := x_2$ if arriving on edge 2, etc."
- So, for each i, insert $x := x_i$ at the end of predecessor block i
- Rely on copy propagation and coalescing in register allocation to eliminate redundant copy instructions

SSA Wrapup

- More details needed to fully and efficiently implement SSA, but these are the main ideas
  - See recent compiler books (but not the new Dragon book!)
- Allows efficient implementation of many optimizations
- SSA is used in most modern optimizing compilers (llvm is based on it) and has been retrofitted into many older ones (gcc is a well-known example)
- Not a silver bullet – some optimizations still need non-SSA forms, but very effective for many

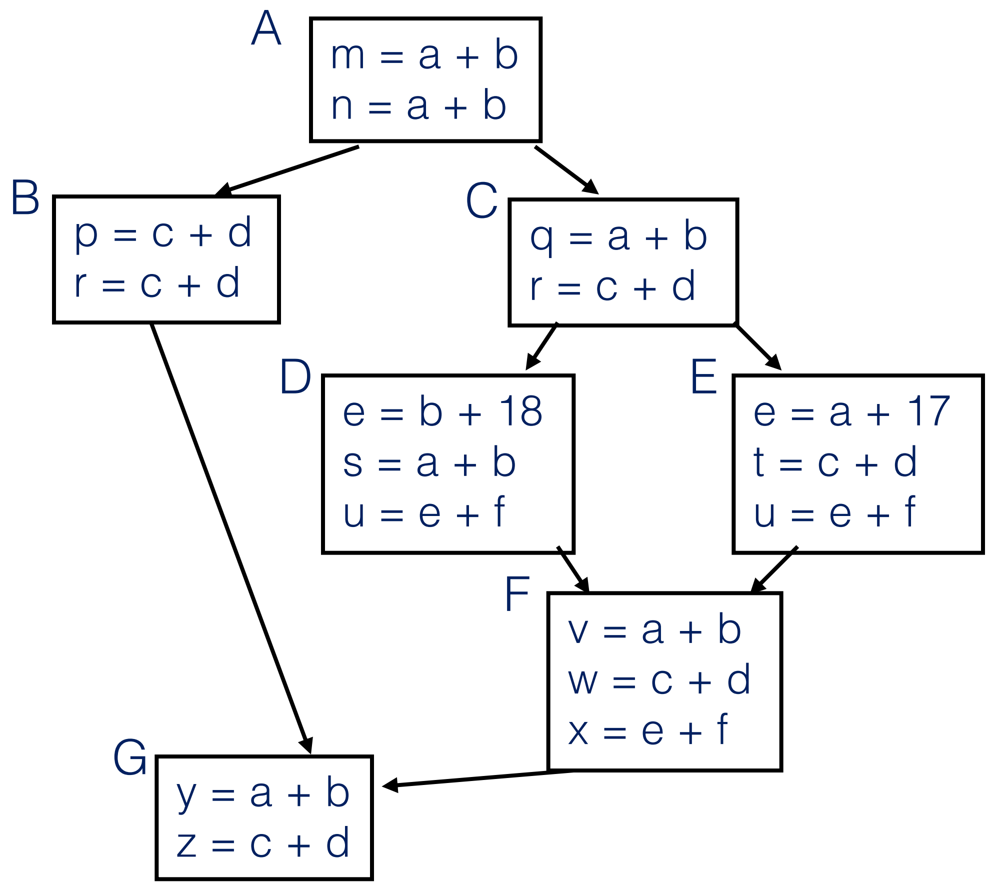
Code Replication

- Sometimes replicating code increases opportunities – modify the code to create larger regions with simple control flow
- Two examples
  - Cloning
  - Inline substitution

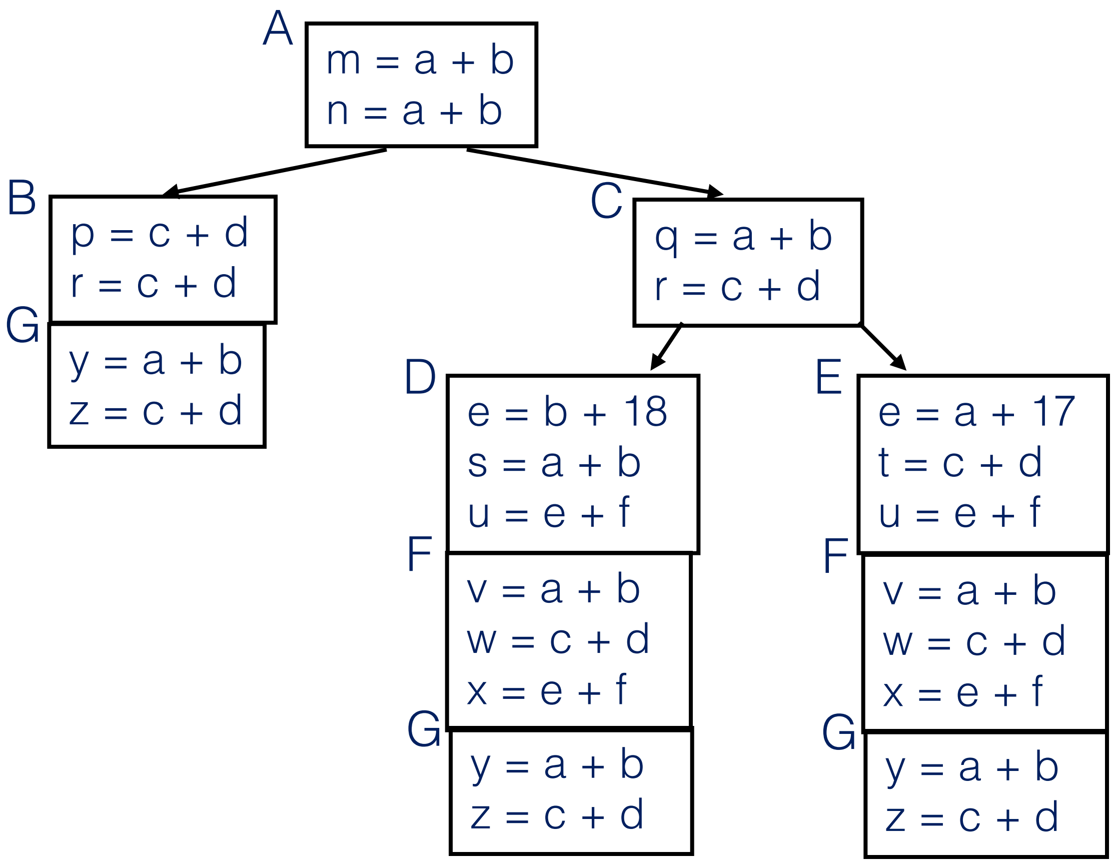
Cloning

- Idea: duplicate blocks with multiple predecessors
- Tradeoff
  - More local optimization possibilities – larger blocks, fewer branches
  - But: larger code size, may slow down if it interacts badly with cache

Original VN Example



Example with Cloning

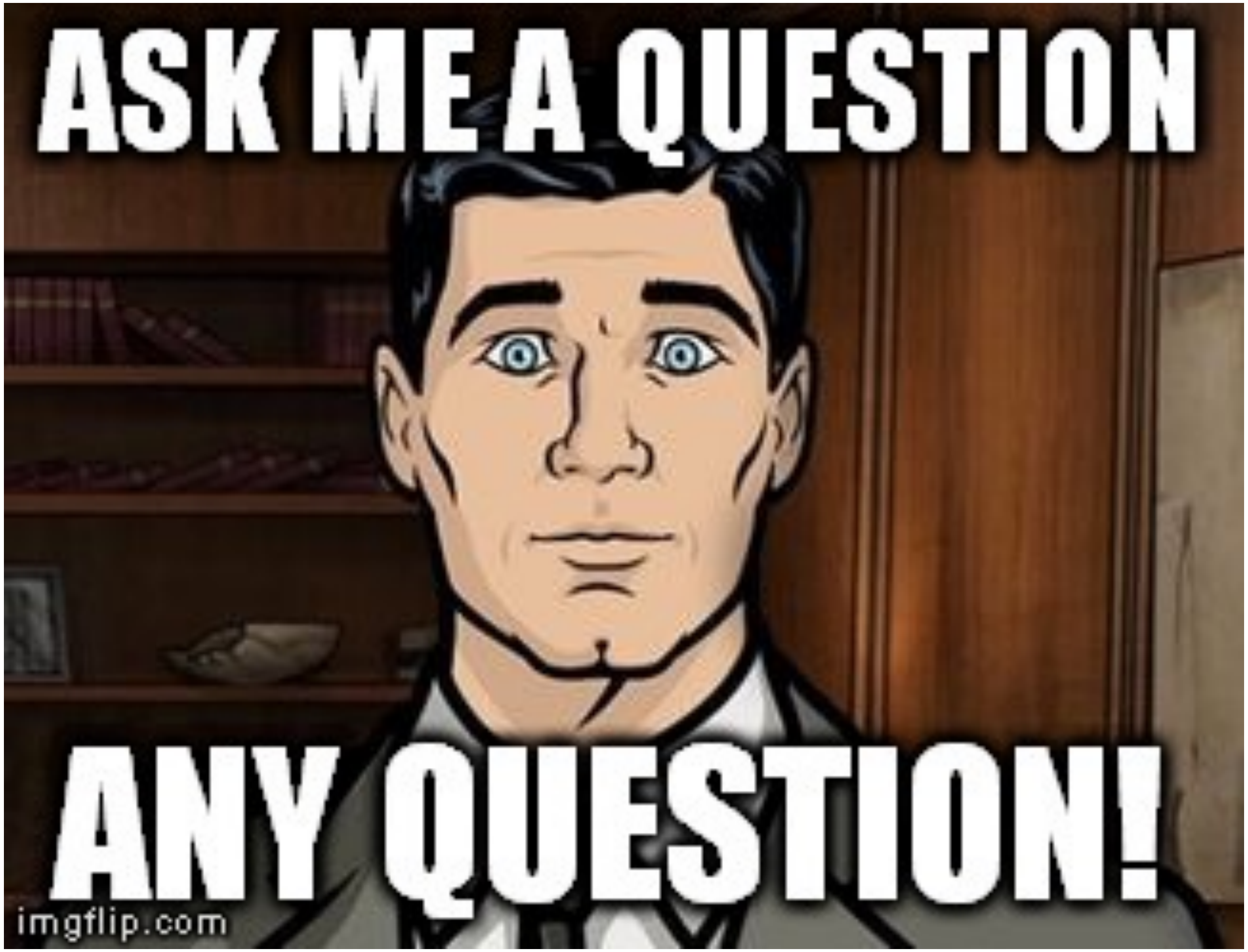


Inline Substitution

- Problem: an optimizer has to treat a procedure call as if it (could have) modified all globally reachable data
  – Plus there is the basic expense of calling the procedure
- Inline Substitution: replace each call site with a copy of the called function body

Inline Substitution Issues

- Pro
  - More effective optimization – better local context and don't need to invalidate local assumptions
  - Eliminate overhead of normal function call
- Con
  - Potential code bloat
  - Need to manage recompilation when either caller or callee changes



[Meme credit: imgflip.com]