Lecture 23

EECS 483: COMPILER CONSTRUCTION

Announcements

- HW6: Analysis and Optimization
  - Due on Thursday, May 2
- Final Exam
  - 4-6pm April 29
  - DOW1010, DOW1005, DOW2166
  - Same cheat sheet policy as before
- Guest Lectures
  - Professor Cyrus Omar today
  - Steven Schaefer (GSI last semester) Wednesday
  - Professor New returns next week.

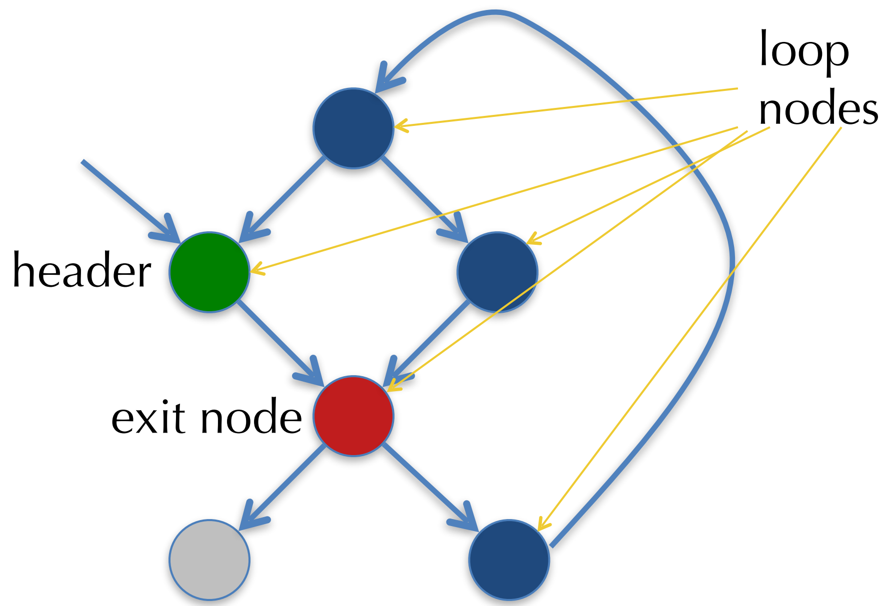
LOOPS AND DOMINATORS

Loops in Control-flow Graphs

- Taking into account loops is important for optimizations.
  - The 90/10 rule (Pareto Principle) 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 / LLVM IR level.
  - Many kinds of loops: while, do/while, for, continue, goto…
- Problem: *How do we identify loops in the control-flow graph?*

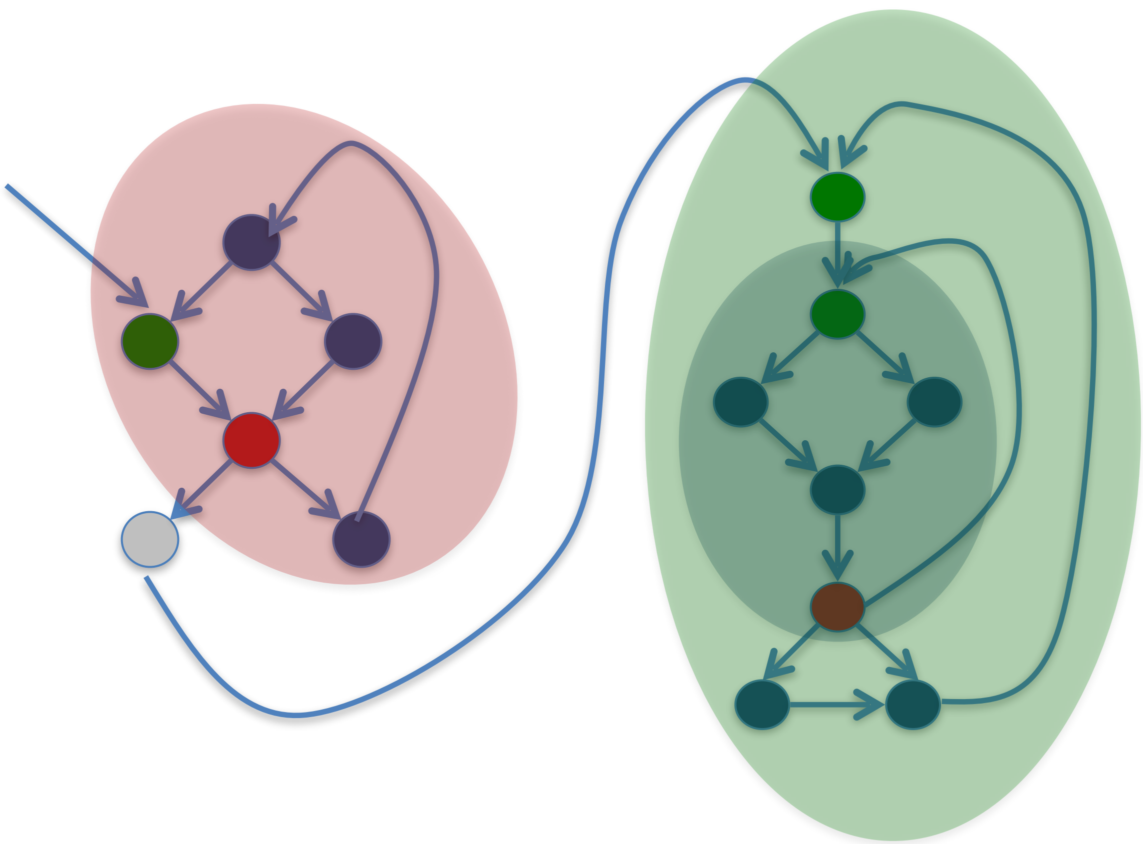
Definition of a (Structured) Loop

- A ***structured 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 (using only nodes in the loop).
  - No edges enter the loop except to the header
- Nodes with outgoing edges are called loop exit nodes
- A loop is a *strongly connected component*

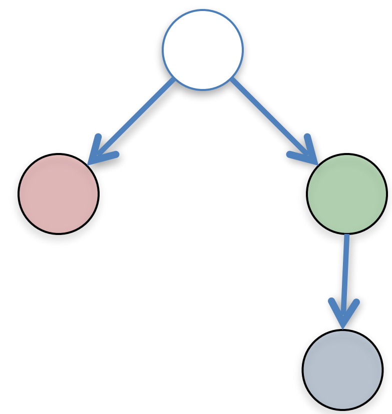


Nested Loops

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



Control Tree:



The control tree depicts the nesting structure of the program.

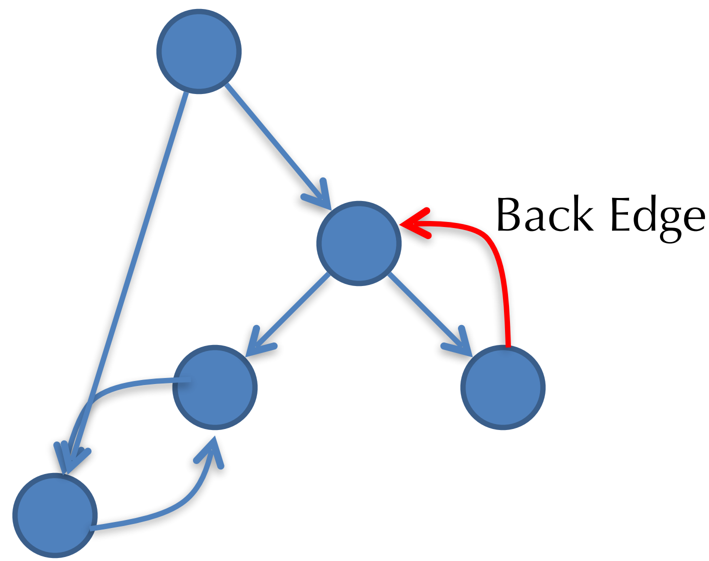
Control-flow Analysis

- 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.

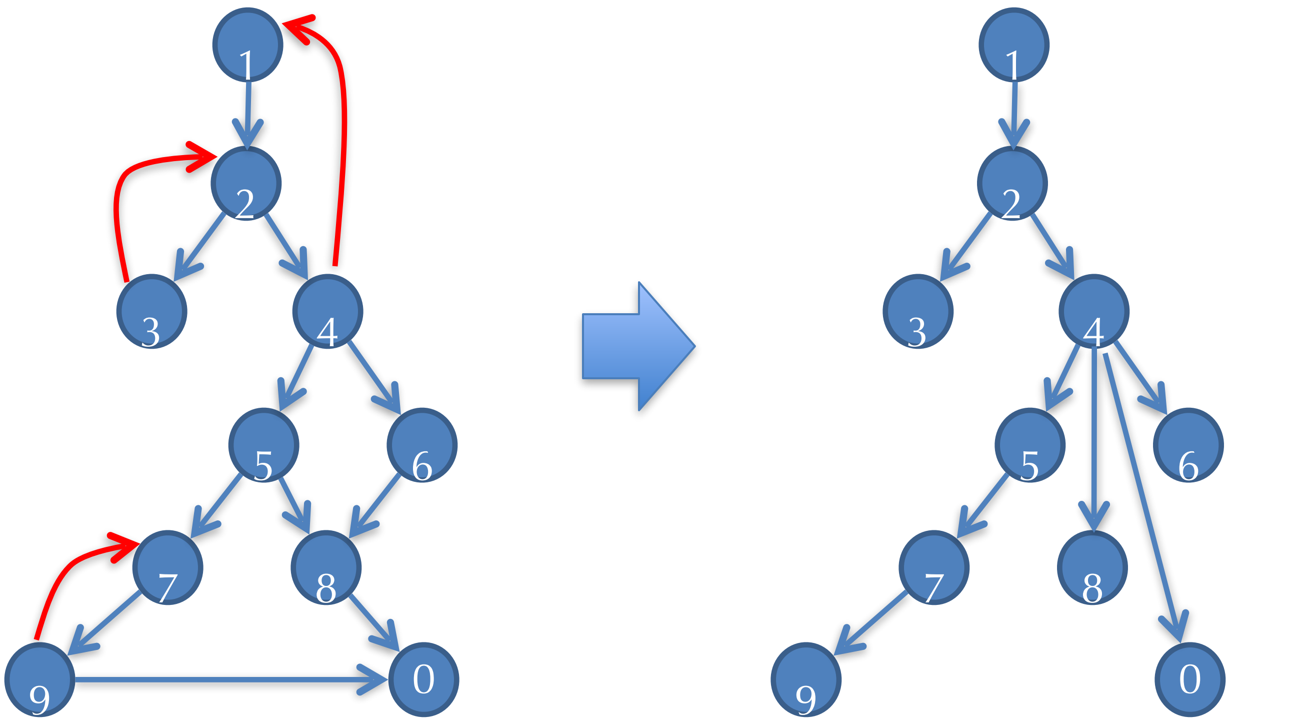
- An edge in the graph is a *back edge* if the target node dominates the source node.

- A loop contains at least one back edge.



Dominator Trees

- Domination is reflexive:
  - A dominates A
- 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



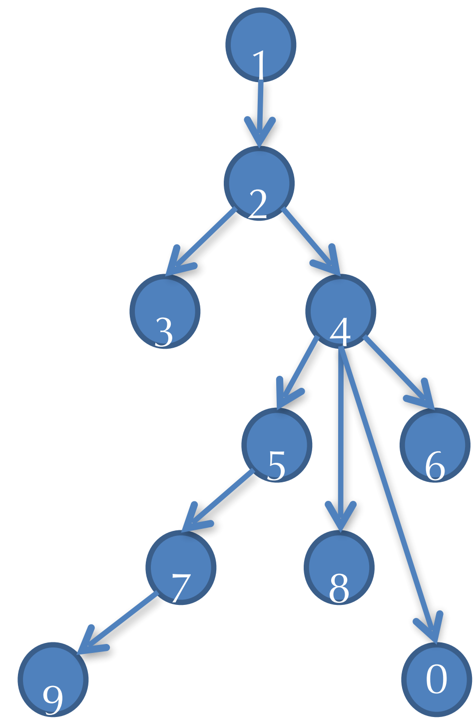
Dominator Dataflow Analysis

- 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."
  - $\text{in}[n] := \bigcap_{n' \in \text{pred}[n]} \text{out}[n']$
- "Every node dominates itself."
  - $\text{out}[n] := \text{in}[n] \cup \{n\}$

- Formally: 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 (meet over all paths)

Improving the Algorithm

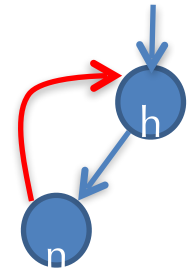
- Dom[b] contains just those nodes along the path in the dominator tree from the root to b:
  - e.g. Dom[8] = {1,2,4,8}, Dom[7] = {1,2,4,5,7}
  - There is a lot of sharing among the nodes
- More efficient way to represent Dom sets is to store the dominator *tree*.
  - doms[b] = immediate dominator of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

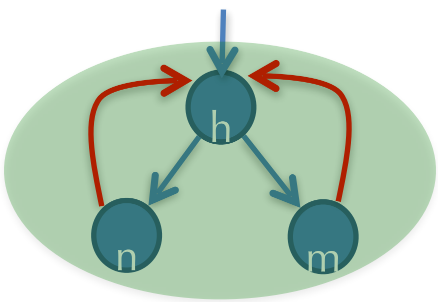
Completing Control-flow Analysis

- 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



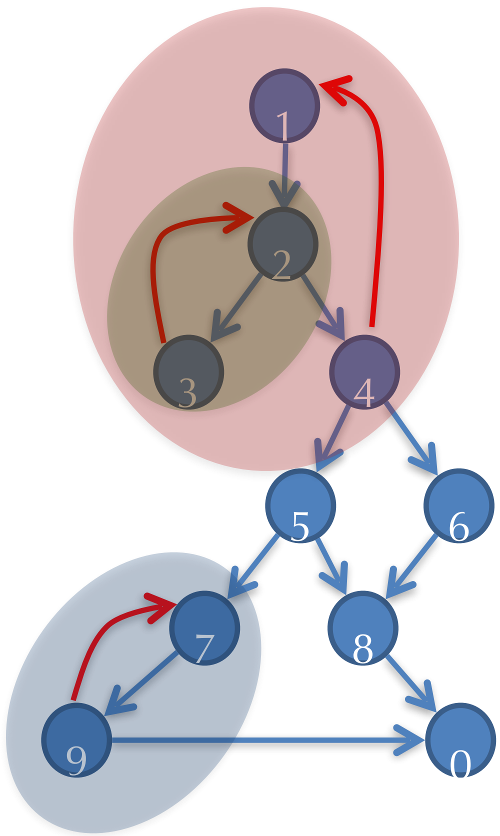
- For each back edge n → h, find the natural loop:
  - {n′ | n is reachable from n′ in G – {h}} ∪ {h}

- Two loops may share the same header: merge them



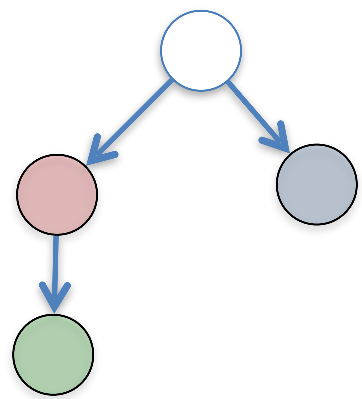
- Nesting structure of loops is determined by set inclusion
  - Can be used to build the control tree

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 backend: 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

- Then: How do we convert SSA code to x86, mapping `%uids` to registers?
  - Register allocation.

Alloca vs. %UID

- Current compilation strategy:

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

```
%x = alloca i64
%y = alloca i64
store i64* %x, 3
store i64* %y, 0
%x1 = load %i64* %x
%tmp1 = add i64 %x1, 1
store i64* %x, %tmp1
%x2 = load %i64* %x
%tmp2 = add i64 %x2, 2
store i64* %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 i64 3, 0
%y1 = add i64 0, 0
%x2 = add i64 %x1, 1
%y2 = add i64 %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 i64 %y1, 1
  br label %merge
else:
  %x3 = mult i64 %y1, 2
merge:
  %z2 = %add i64 ???, 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.

    $\texttt{\%uid = phi}\ \langle\texttt{ty}\rangle\ \texttt{v}_1, \langle\texttt{label}_1\rangle, \ldots, \texttt{v}_n, \langle\texttt{label}_n\rangle$

```
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 i64 %y1, 1
  br label %merge
else:
  %x3 = mult i64 %y1, 2
merge:
  %x4 = phi i64 %x2, %then, %x3, %else
  %z2 = %add i64 %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*

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

body:
  %x2 = phi i64 %x1, %entry, %x3, %body
  %x3 = add i64 %x2, %y1
  %p = icmp slt i64, %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 i64             // %x cannot be promoted
  %y = call malloc(i64 8)
  %ptr = bitcast i8* %y to i64**
  store i65** %ptr, %x        // store the pointer into the heap
```

```
entry:
  %x = alloca i64           // %x cannot be promoted
  %y = call foo(i64* %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
- Place 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.

Phi Placement (Inefficient)

- Less efficient, but easier to understand:

- Place phi nodes "maximally" (i.e. at every node with > 1 predecessors)

- If all values flowing into phi node are the same, then eliminate it:

```
%x = phi t  %y, %pred1  %y  %pred2  … %y %predK
// code that uses %x
⇒
// code with %x replaced by %y
```

- Interleave with other optimizations
  - copy propagation
  - constant propagation
  - etc.

Phi Placement (Efficient)

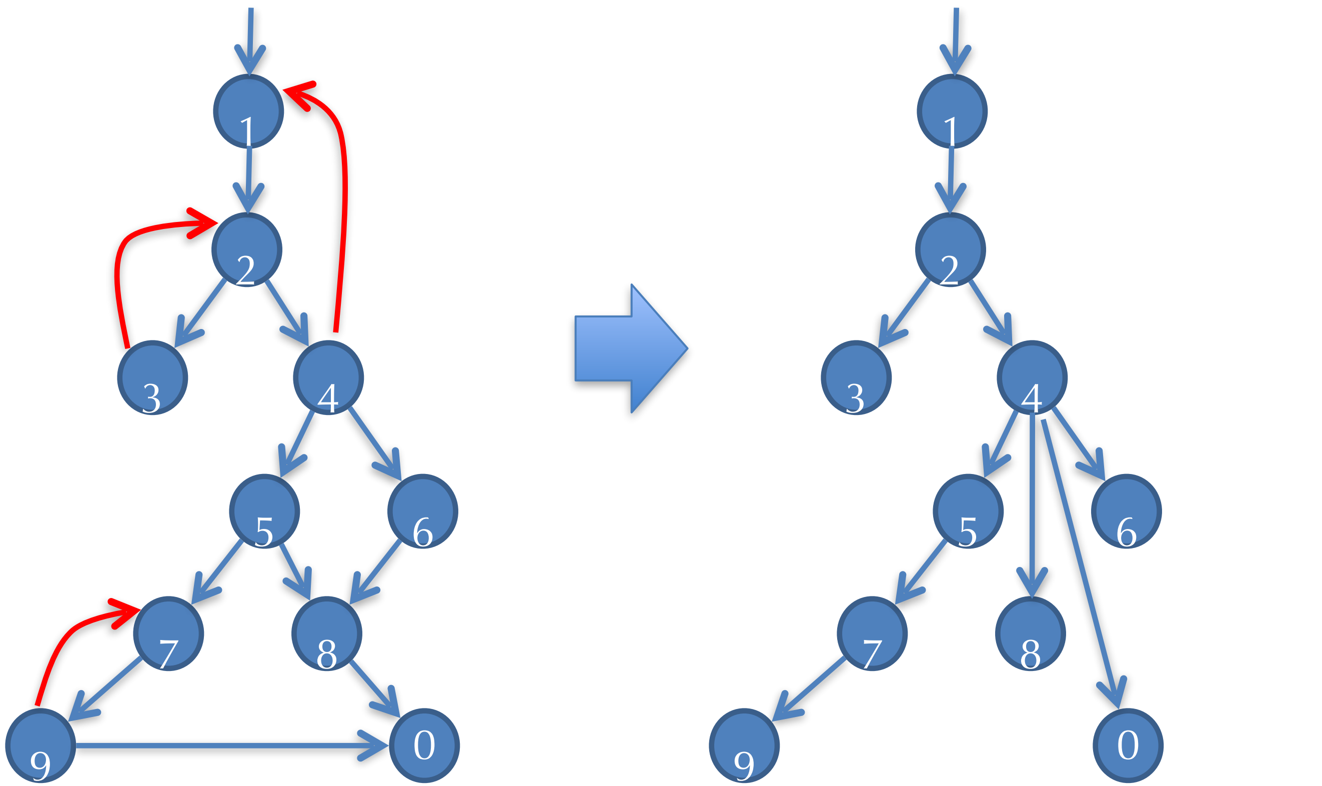
- Need to calculate the "Dominance Frontier"

- Node A ***strictly dominates*** node B if A dominates B and A $\neq$ B.
  - A dominates A but A does not strictly dominate A

- 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[B] for the dominance frontier of node B.
  - Intuitively, which downstream nodes of B **with multiple predecessors** are not strictly dominated by B.

Dominance Frontiers

- Example of a dominance frontier calculation results
- DF[1] = {1},   DF[2] = {1,2},   DF[3] = {2}, DF[4] = {1}, DF[5] = {8,0}, DF[6] = {8}, DF[7] = {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)
- Adds each B to the DF sets to which it belongs

```
for all nodes B
    if #(pred[B]) ≥ 2                    // (just an optimization)
     for each p ∈pred[B] {
          runner := p                    // start at the predecessor of B
          while (runner ≠ doms[B])       // walk up the tree adding 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:

$$DF[N] = \cup_{n \in N} DF[n]$$

- Suppose that a variable x is defined at a set of nodes N.

$DF_0[N] = DF[N]$
$DF_{i+1}[N] = DF[DF_i[N] \cup N]$

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 node in J[N].
  – $x = \phi(x, x, \ldots, x)$; (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, just as in the dataflow solver.

- Intuition:
  – If N is the set of places where x is modified, then DF[N] is the places where phi nodes need to be added, but those also "count" as modifications of x, so we need to insert the phi nodes to capture those modifications too…

Example Join-point Calculation

- Suppose the variable x is modified at nodes 3 and 6
  - Where would we need to add phi nodes?

- $DF_0[\{3,6\}] = DF[\{3,6\}] = DF[3] \cup DF[6] = \{2,8\}$
- $DF_1[\{3,6\}]$

  $= DF[DF_0\{3,6\} \cup \{3,6\}]$

  $= DF[\{2,3,6,8\}]$

  $= DF[2] \cup DF[3] \cup DF[6] \cup DF[8]$

  $= \{1,2\} \cup \{2\} \cup \{8\} \cup \{0\} = \{1,2,8,0\}$
- $DF_2[\{3,6\}]$

  $= \ldots$

  $= \{1,2,8,0\}$

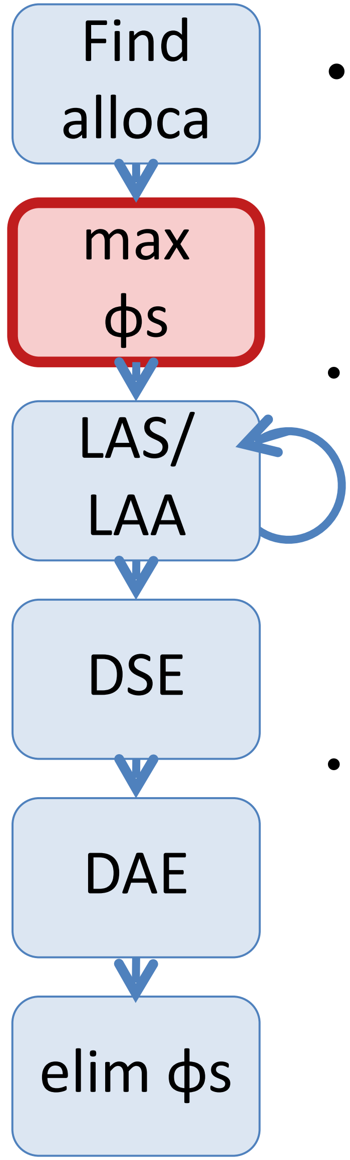
- So $J[\{3,6\}] = \{1,2,8,0\}$ and we need to add phi nodes at those four spots.

Example SSA Optimizations

```
l1: %p = alloca i64
    store 0, %p
    %b = %y > 0

    br %b, %l2, %l3
```

```
l2:


    store 1, %p

    br %l3
```

```
l3:


    %x = load %p
    ret %x
```



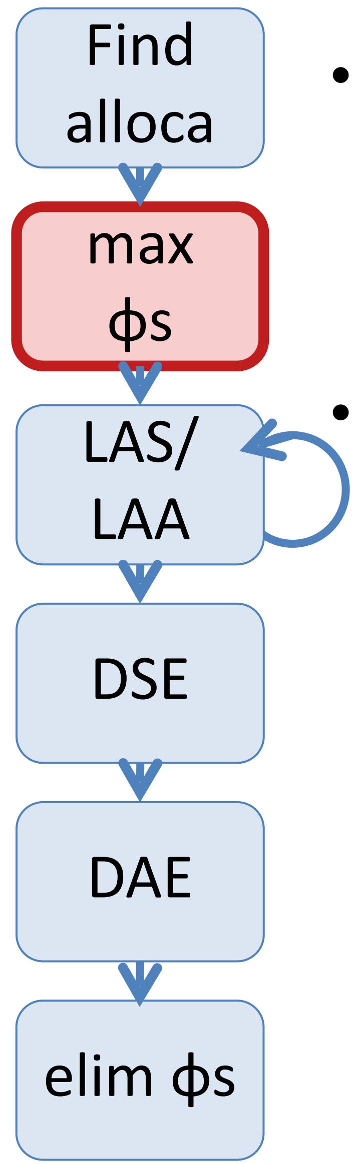
- How to place phi nodes without breaking SSA?
- Note: the "real" implementation combines many of these steps into one pass.
  - Places phis directly at the dominance frontier
- This example also illustrates other common optimizations:
  - Load after store/alloca
  - Dead store/alloca elimination

Example SSA Optimizations

```
l1: %p = alloca i64
    store 0, %p
    %b = %y > 0
    %x1 = load %p
    br %b, %l2, %l3
```

```
l2:

    store 1, %p
    %x2 = load %p
    br %l3
```

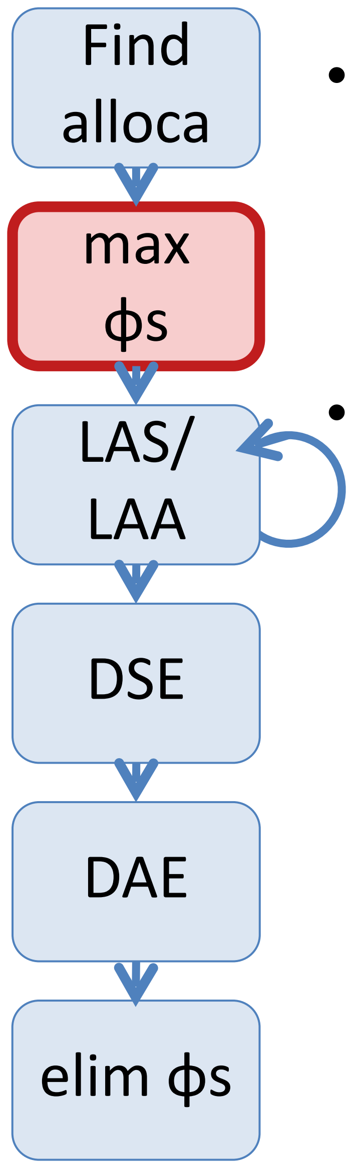
```
l3:

    %x = load %p
    ret %x
```



- How to place phi nodes without breaking SSA?
- Insert
  - Loads at the end of each block

Example SSA Optimizations

```
l1: %p = alloca i64
    store 0, %p
    %b = %y > 0
    %x1 = load %p
    br %b, %l2, %l3

l2: %x3 = φ[%x1,%l1]

    store 1, %p
    %x2 = load %p
    br %l3

l3: %x4 = φ[%x1;%l1, %x2:%l2]

    %x = load %p
    ret %x
```



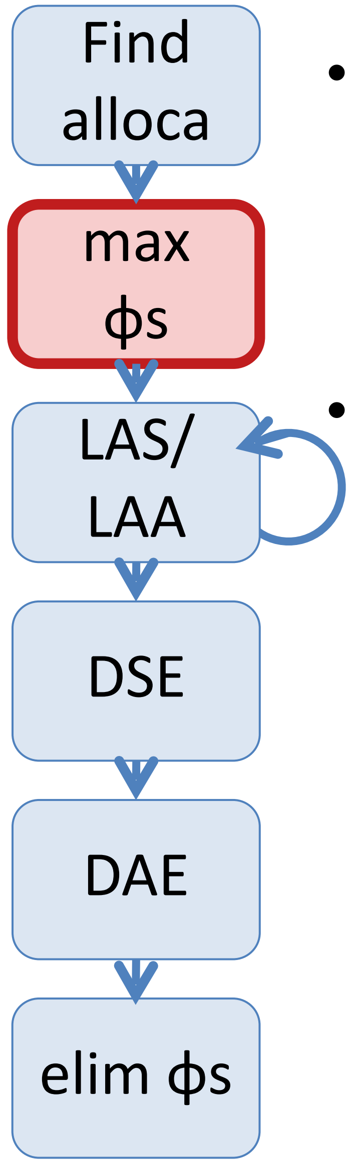
- How to place phi nodes without breaking SSA?
- Insert
  - Loads at the end of each block
  - Insert φ-nodes at each block

Example SSA Optimizations

```
l1: %p = alloca i64
    store 0, %p
    %b = %y > 0
    %x1 = load %p
    br %b, %l2, %l3
```

```
l2: %x3 = ϕ[%x1,%l1]
    store %x3, %p
    store 1, %p
    %x2 = load %p
    br %l3
```

```
l3: %x4 = φ[%x1;%l1, %x2:%l2]
    store %x4, %p
    %x = load %p
    ret %x
```



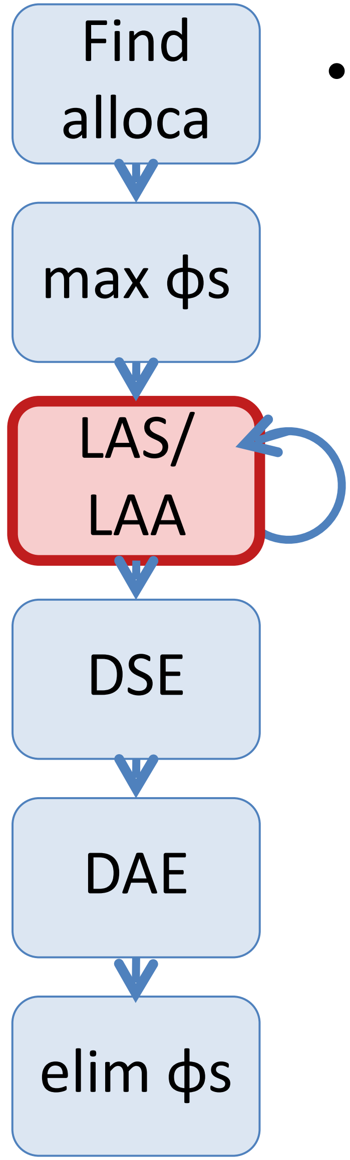
- How to place phi nodes without breaking SSA?
- Insert
  - Loads at the end of each block
  - Insert ϕ-nodes at each block
  - Insert stores after ϕ-nodes

Example SSA Optimizations

```
l1: %p = alloca i64
    store 0, %p
    %b = %y > 0
    %x1 = load %p
    br %b, %l2, %l3
```

```
l2: %x3 = ϕ[%x1,%l1]
    store %x3, %p
    store 1, %p
    %x2 = load %p
    br %l3
```

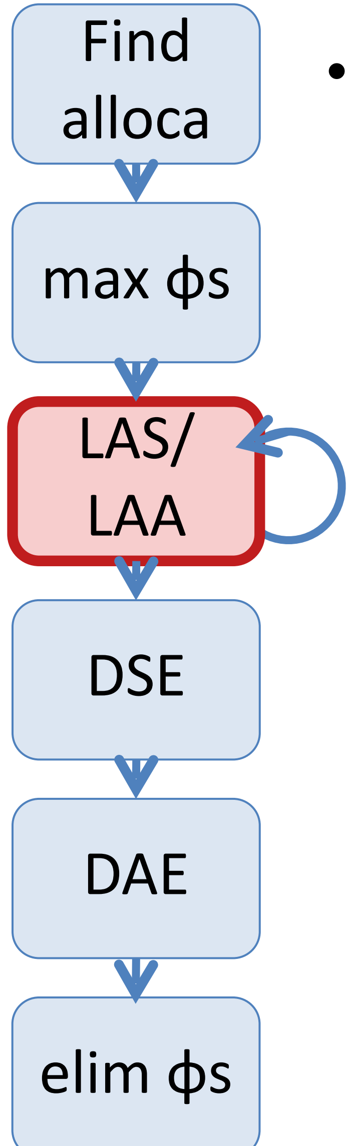
```
l3: %x4 = φ[%x1;%l1, %x2:%l2]
    store %x4, %p
    %x = load %p
    ret %x
```



- For loads after stores (LAS):
  - Substitute all uses of the load by the value being stored
  - Remove the load

Example SSA Optimizations

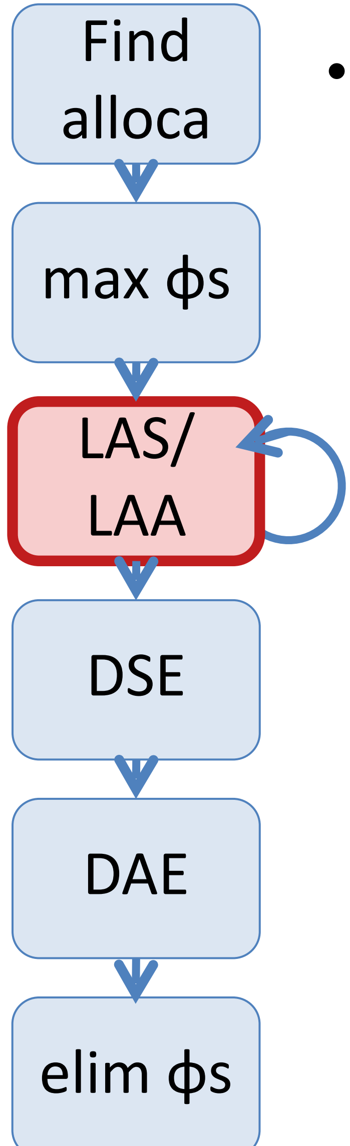
```
l1: %p = alloca i64
    store 0, %p
    %b = %y > 0
    %x1 = load %p
    br %b, %l2, %l3

l2: %x3 = ϕ[%x1:%l1]
    store %x3, %p
    store 1, %p
    %x2 = load %p
    br %l3

l3: %x4 = φ[%x1:%l1, %x2:%l2]
    store %x4, %p
    %x = load %p
    ret %x
```



- For loads after stores (LAS):
  - Substitute all uses of the load by the value being stored
  - Remove the load

Example SSA Optimizations

```
l1: %p = alloca i64
    store 0, %p
    %b = %y > 0
    %x1 = load %p
    br %b, %l2, %l3

l2: %x3 = ϕ[0,%l1]
    store %x3, %p
    store 1, %p
    %x2 = load %p
    br %l3

l3: %x4 = φ[0;%l1, %x2:%l2]
    store %x4, %p
    %x = load %p
    ret %x
```



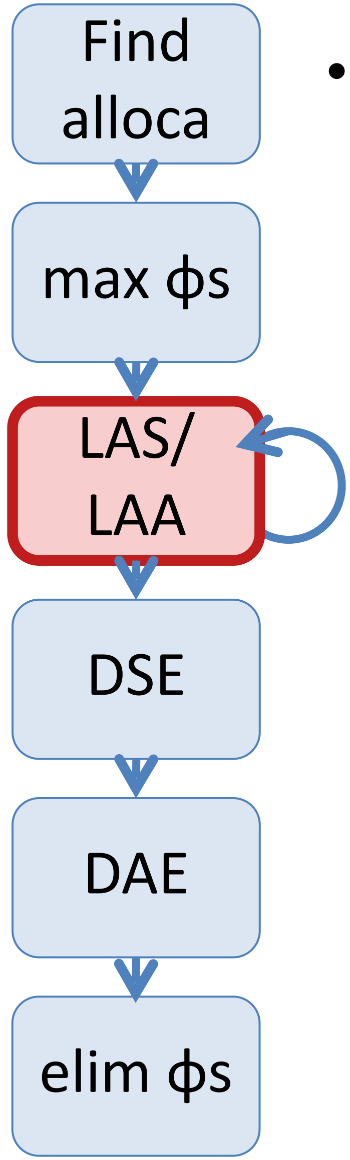
- For loads after stores (LAS):
  - Substitute all uses of the load by the value being stored
  - Remove the load

Example SSA Optimizations

```
l1: %p = alloca i64
    store 0, %p
    %b = %y > 0

    br %b, %l2, %l3
```

```
l2: %x3 = ϕ[0,%l1]
    store %x3, %p
    store 1, %p
    %x2 = load %p
    br %l3
```

```
l3: %x4 = φ[0;%l1, %x2, %l2]
    store %x4, %p
    %x = load %p
    ret %x
```



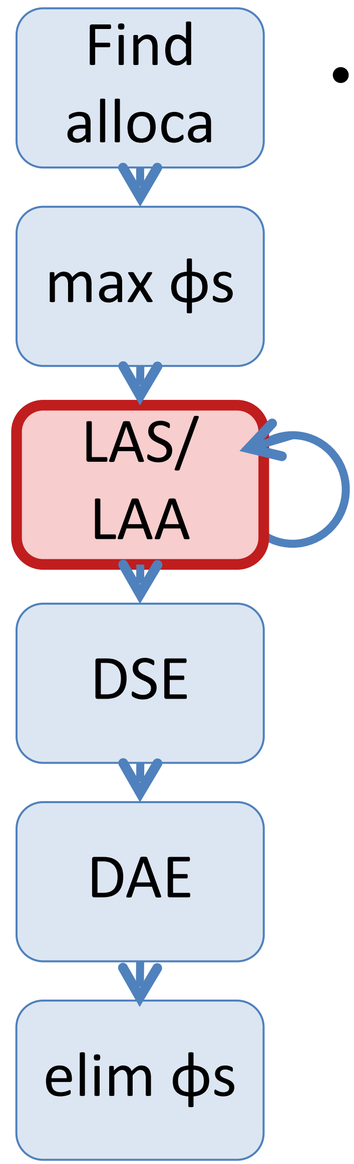
- For loads after stores (LAS):
  - Substitute all uses of the load by the value being stored
  - Remove the load

Example SSA Optimizations

```
l1: %p = alloca i64
    store 0, %p
    %b = %y > 0

    br %b, %l2, %l3
```

```
l2: %x3 = ϕ[0,%l1]
    store %x3, %p
    store 1, %p
    %x2 = load %p
    br %l3
```

```
l3: %x4 = φ[0;%l1, 1:%l2]
    store %x4, %p
    %x = load %p
    ret %x
```



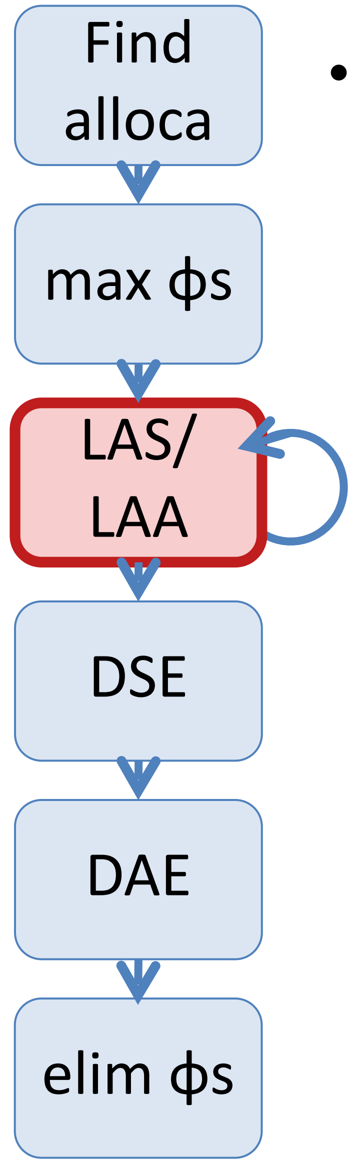
- For loads after stores (LAS):
  - Substitute all uses of the load by the value being stored
  - Remove the load

Example SSA Optimizations

```
l1: %p = alloca i64
    store 0, %p
    %b = %y > 0

    br %b, %l2, %l3
```

```
l2: %x3 = ϕ[0,%l1]
    store %x3, %p
    store 1, %p

    br %l3
```

```
l3: %x4 = φ[0;%l1, 1:%l2]
    store %x4, %p
    %x = load %p
    ret %x
```



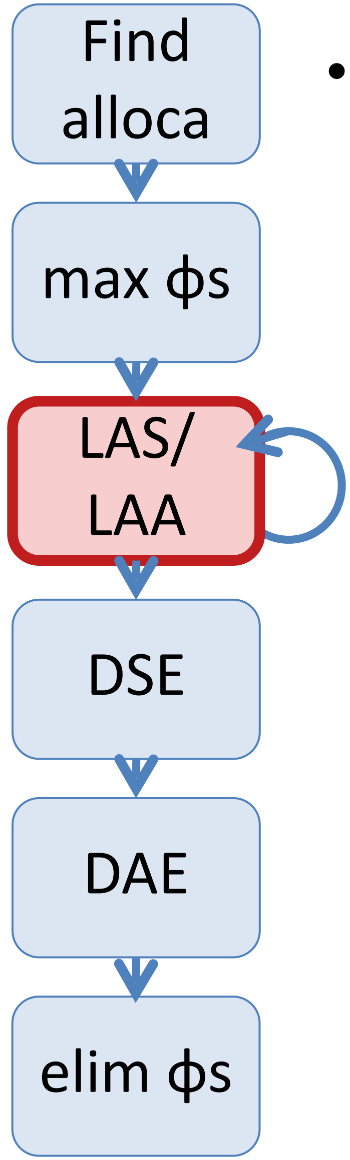
- For loads after stores (LAS):
  - Substitute all uses of the load by the value being stored
  - Remove the load

Example SSA Optimizations

```
l1: %p = alloca i64
    store 0, %p
    %b = %y > 0

    br %b, %l2, %l3
```

```
l2: %x3 = ϕ[0,%l1]
    store %x3, %p
    store 1, %p

    br %l3
```

```
l3: %x4 = φ[0;%l1, 1:%l2]
    store %x4, %p
    %x = load %p
    ret %x4
```



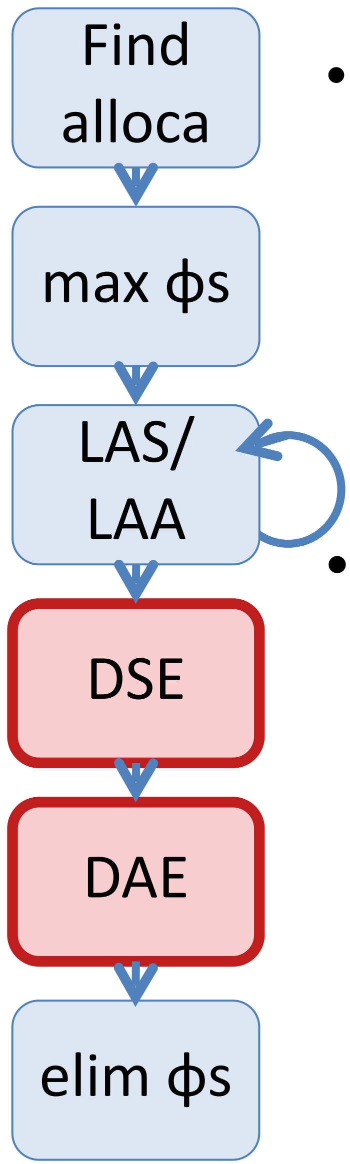
- For loads after stores (LAS):
  - Substitute all uses of the load by the value being stored
  - Remove the load

Example SSA Optimizations

```
l1: %p = alloca i64
    store 0, %p
    %b = %y > 0

    br %b, %l2, %l3
```

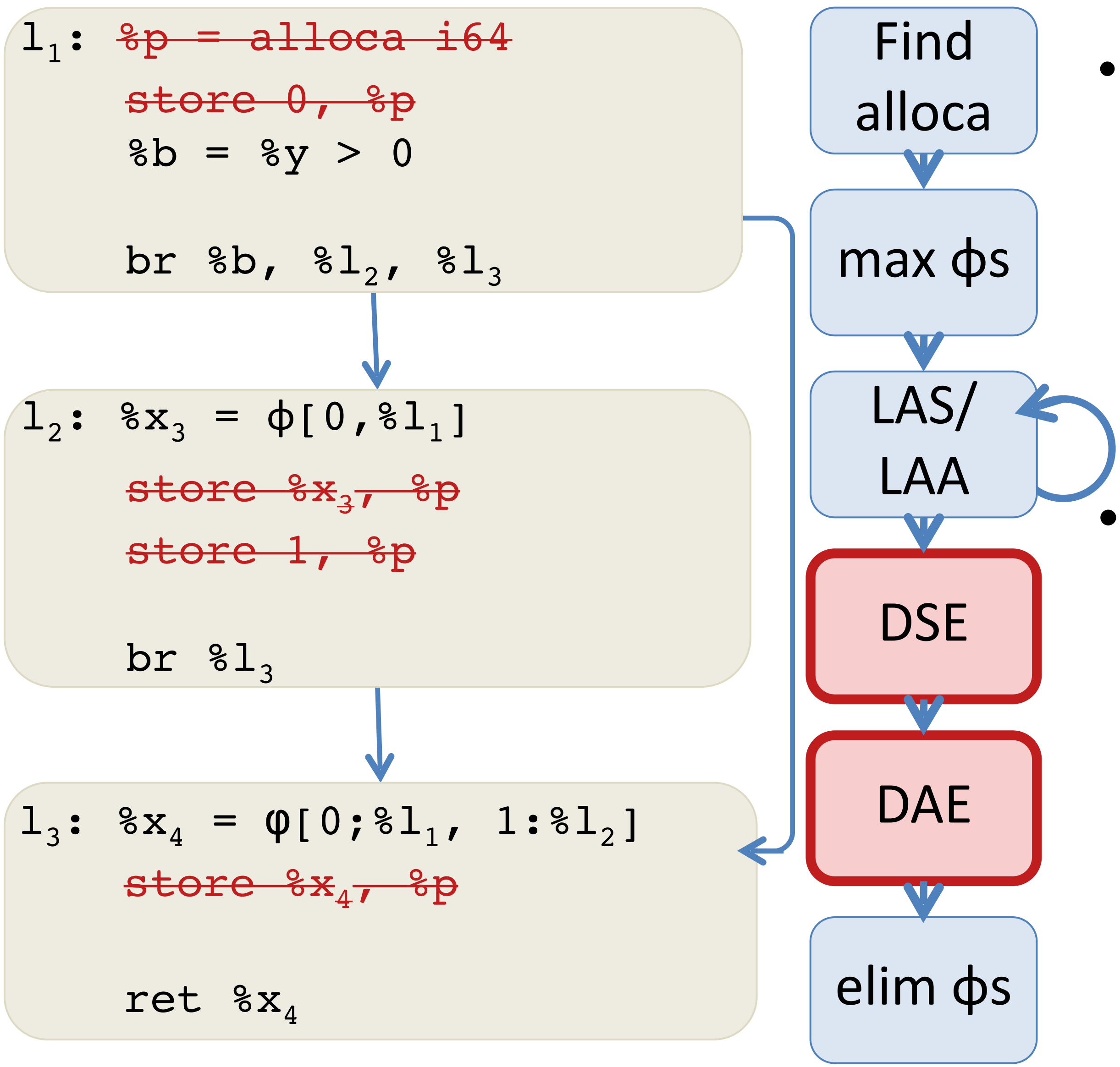
```
l2: %x3 = ϕ[0,%l1]
    store %x3, %p
    store 1, %p

    br %l3
```

```
l3: %x4 = φ[0;%l1, 1:%l2]
    store %x4, %p

    ret %x4
```



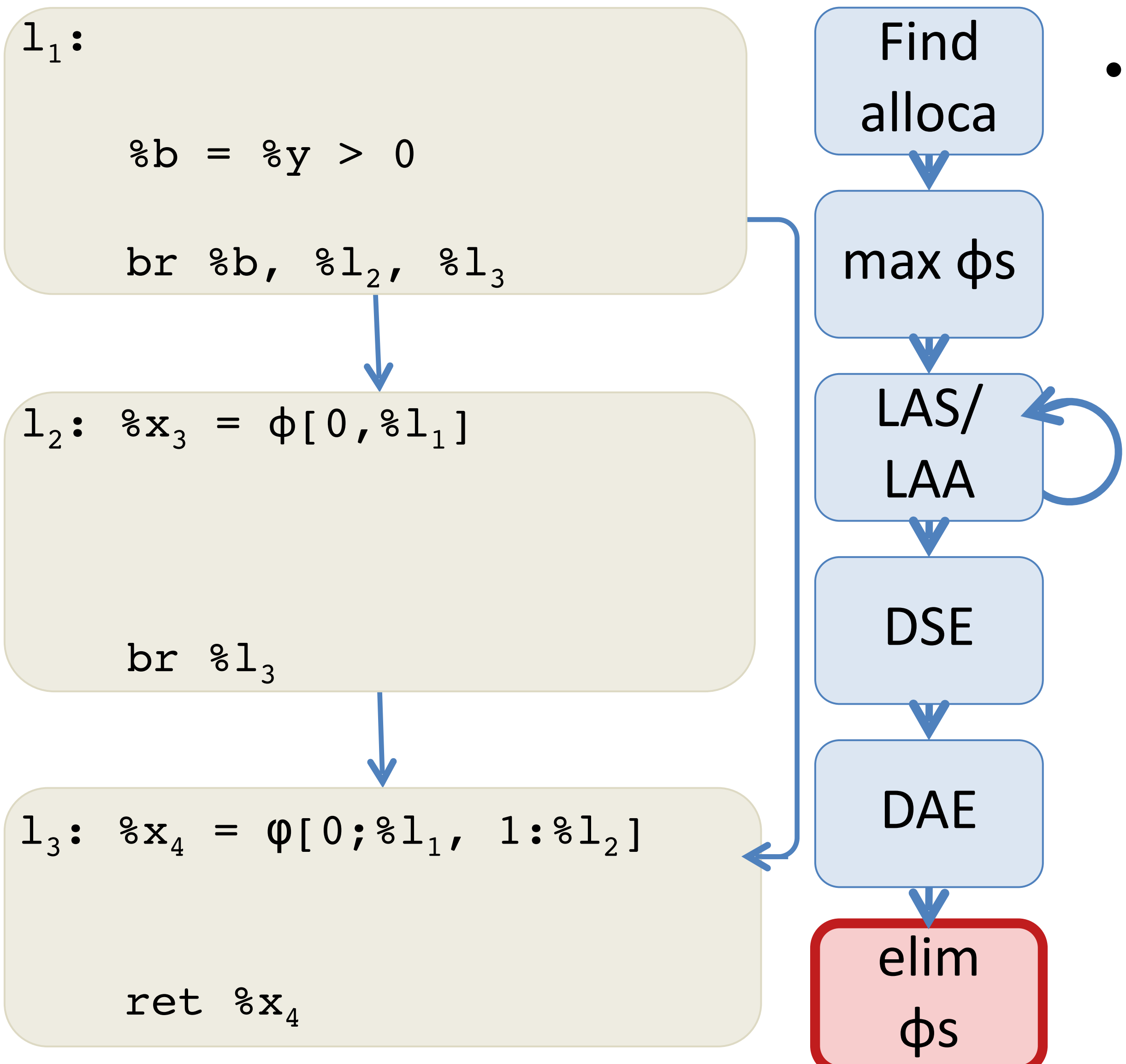
- Dead Store Elimination (DSE)
  - Eliminate all stores with no subsequent loads.
- Dead Alloca Elimination (DAE)
  - Eliminate all allocas with no subsequent loads/ stores.

Example SSA Optimizations



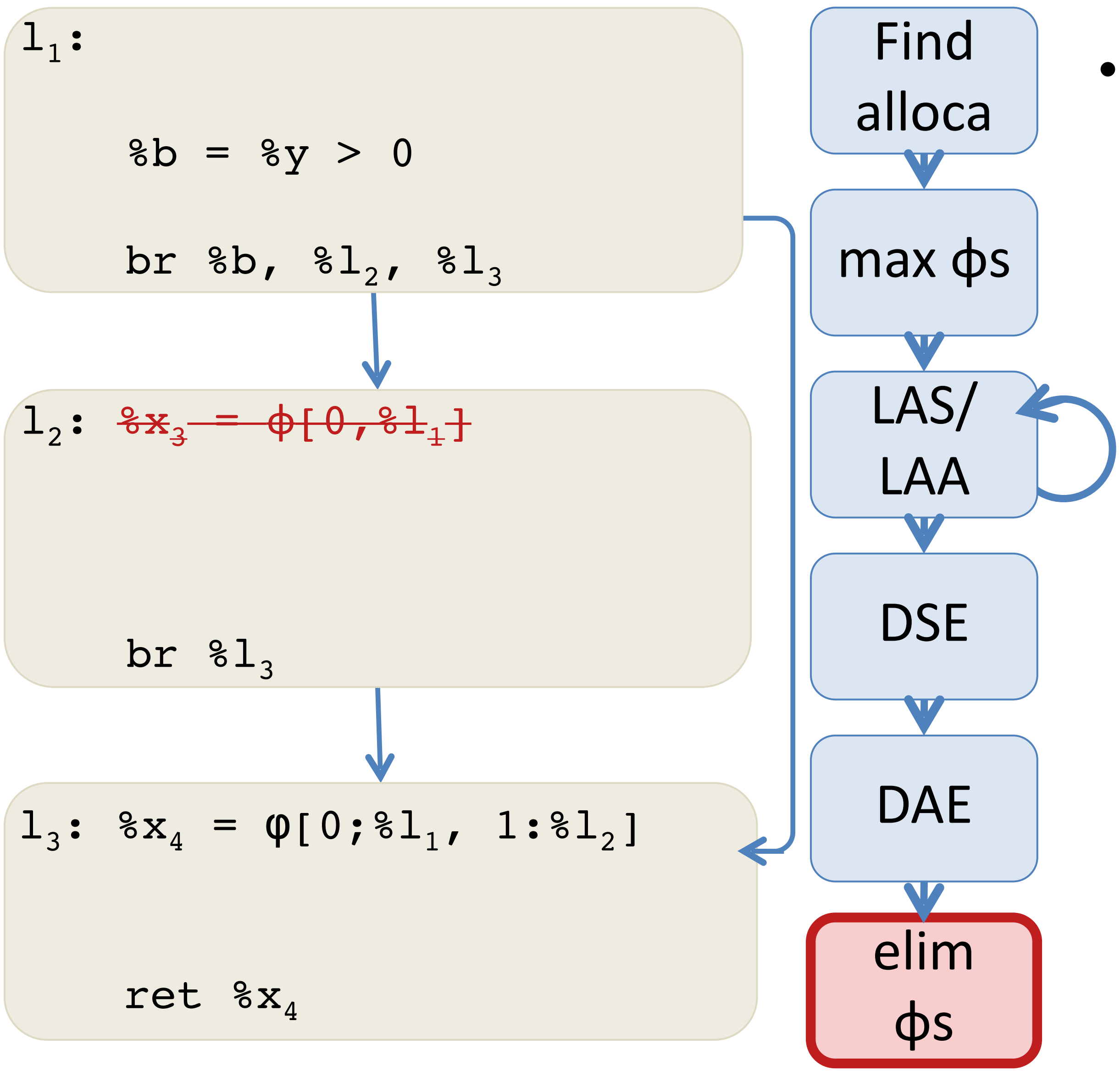
- Dead Store Elimination (DSE)
  - Eliminate all stores with no subsequent loads.
- Dead Alloca Elimination (DAE)
  - Eliminate all allocas with no subsequent loads/ stores.

Example SSA Optimizations



- Eliminate φ nodes:
  - Singletons
  - With identical values from each predecessor
  - See Aycock & Horspool, 2002

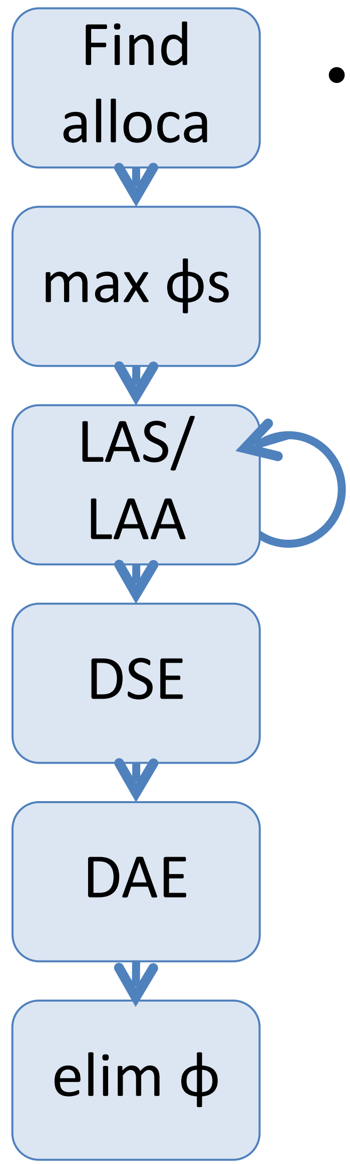
Example SSA Optimizations



- Eliminate φ nodes:
  - Singletons
  - With identical values from each predecessor

Example SSA Optimizations

```
l1:
    %b = %y > 0
    br %b, %l2, %l3

l2:
    br %l3

l3: %x4 = φ[0;%l1, 1:%l2]
    ret %x4
```



- Done!

LLVM Phi Placement

- This transformation is also sometimes called register promotion
  - older versions of LLVM called this "mem2reg" memory to register promotion
- In practice, LLVM combines this transformation with *scalar replacement of aggregates* (SROA)
  - i.e. transforming loads/stores of structured data into loads/stores on register-sized data
- These algorithms are (one reason) why LLVM IR allows annotation of predecessor information in the .ll files
  - Simplifies computing the DF