SSA (2 of 2)

15-411/15-611 Compiler Design

Ben L. Titzer and Seth Goldstein

January 30, 2025

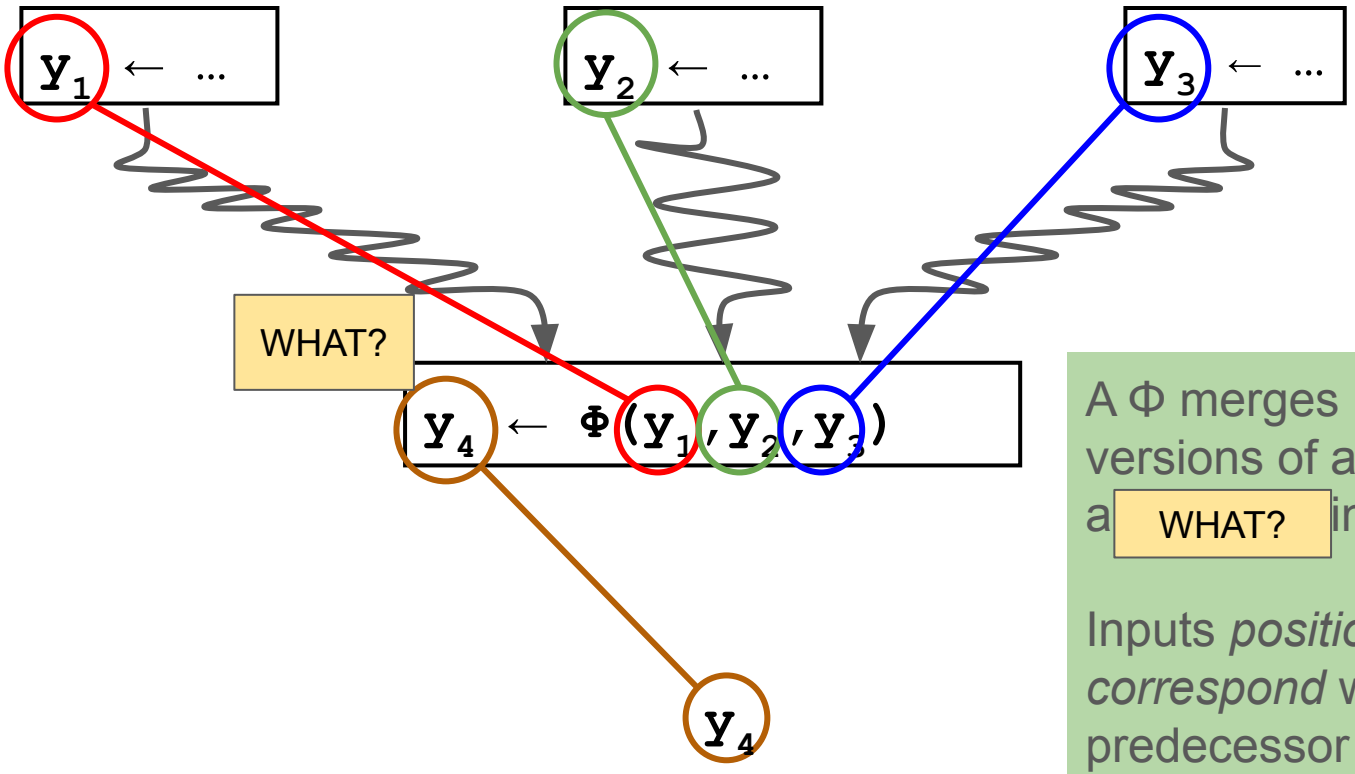
Today

- Thinking in SSA: Local reasoning
- Maintaining SSA properties during optimization
- Deconstructing SSA
  - Two issues: critical edges and ordering moves
- Syntax-directed SSA generation
- Bonus: Virgil compiler's SSA CFG

Recap From Last Time

- SSA is an IR for analysis and optimization
- Every modern optimizing compiler uses it!
- Invariant: every variable defined statically once
- Φ-functions are a notational fiction for merging dataflow at joins
- Can build SSA using the dominator tree
- Iterated dominance frontier guides Φ placement
- Renaming step walks down dominator tree
- General algorithm handles all possible CFGs

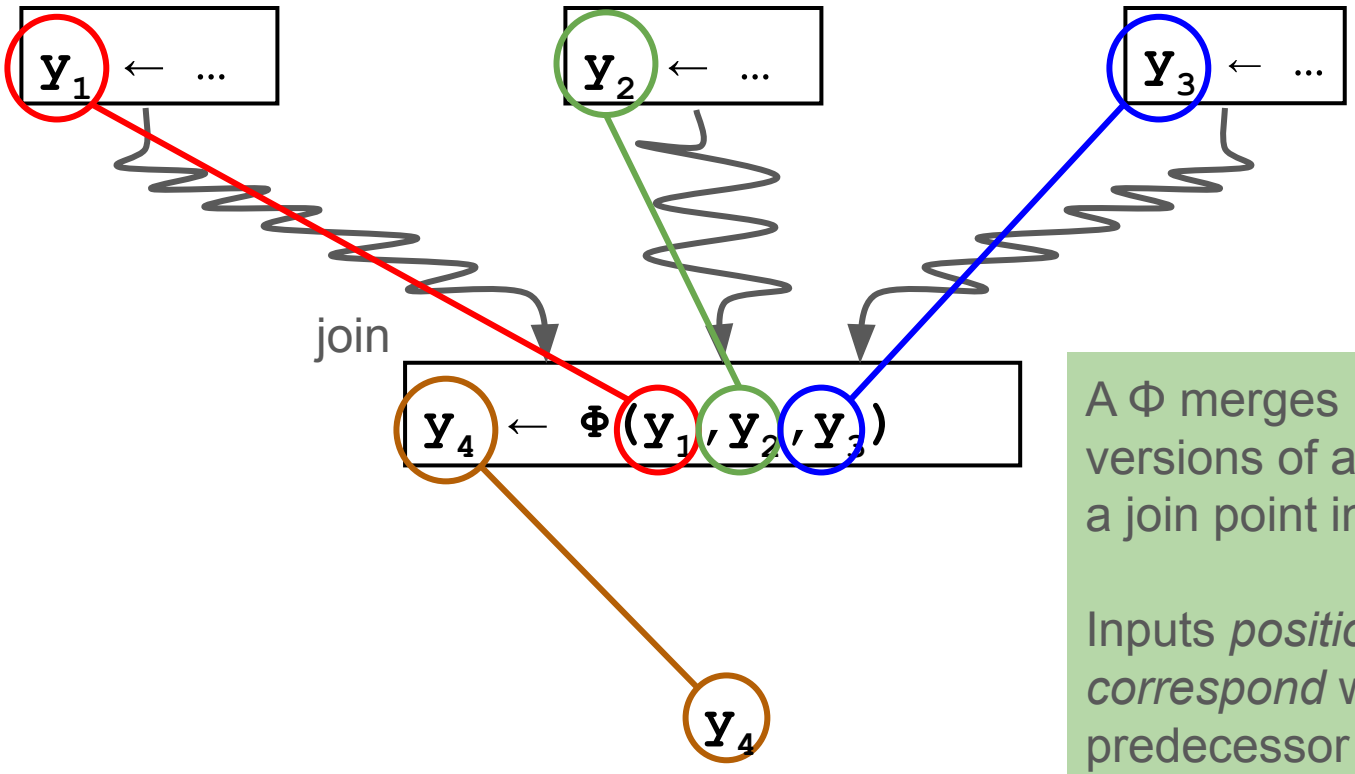
What is a Φ anyway?



A Φ merges multiple versions of a variable at a WHAT? in the CFG.

Inputs *positionally correspond* with predecessor edges.

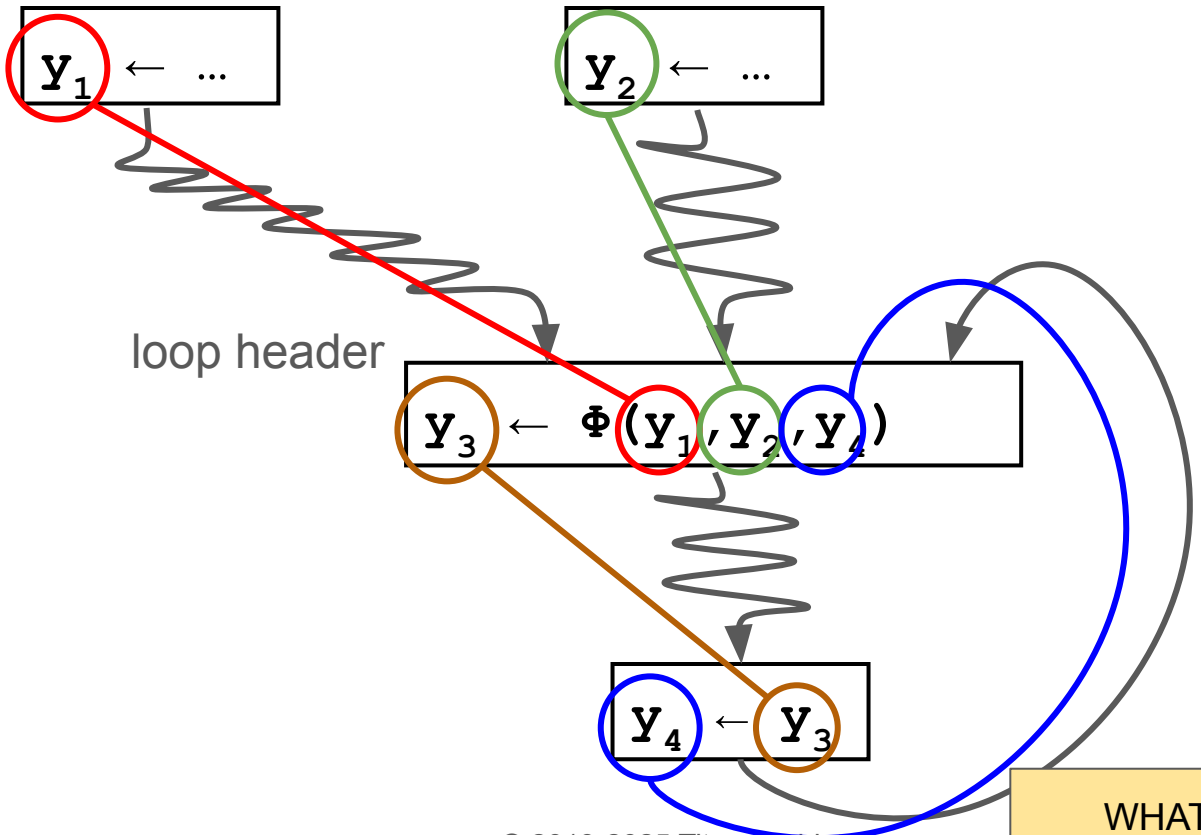
What is a Φ anyway?



A Φ merges multiple versions of a variable at a join point in the CFG.

Inputs *positionally correspond* with predecessor edges.

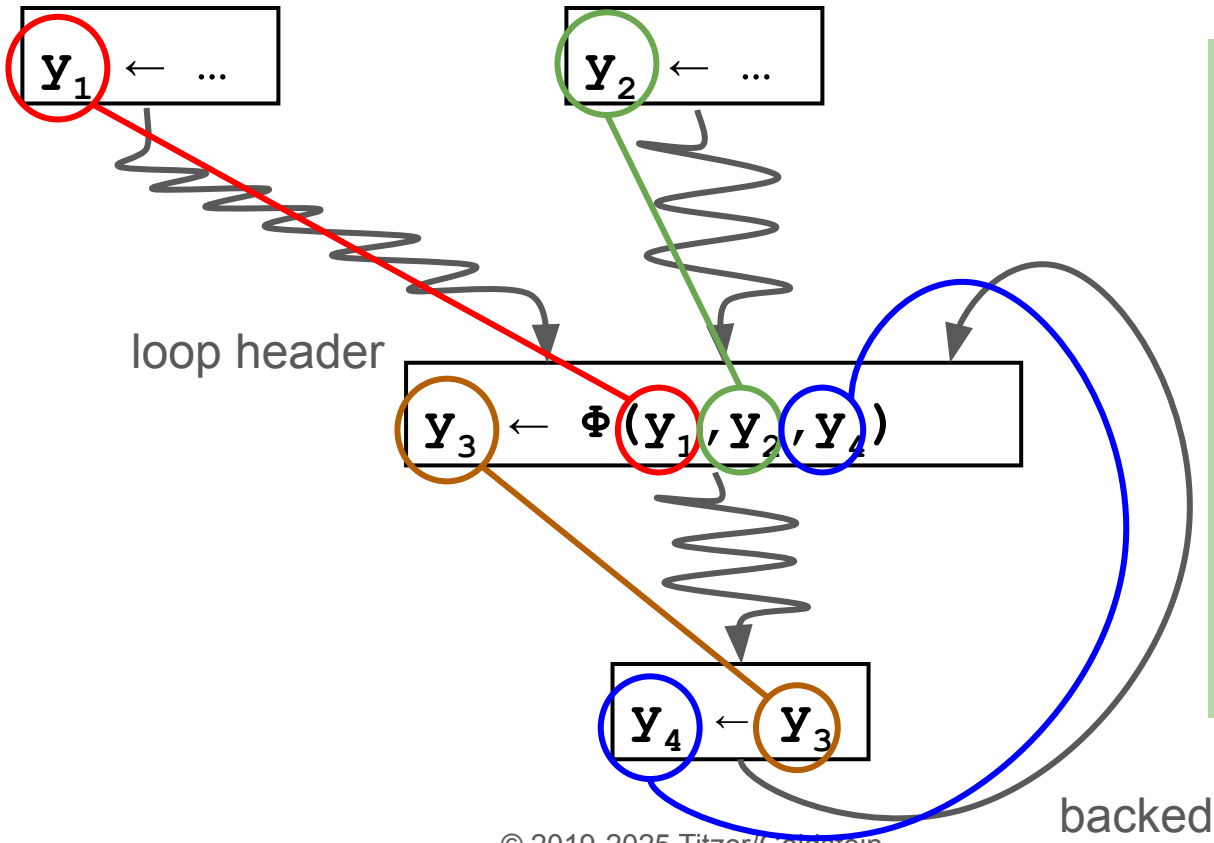
What is a Φ (for a loop) anyway?



Φs at loop headers relate the dataflow on a lo WHAT? with the control flow.

Allows finding induction variables really easily.

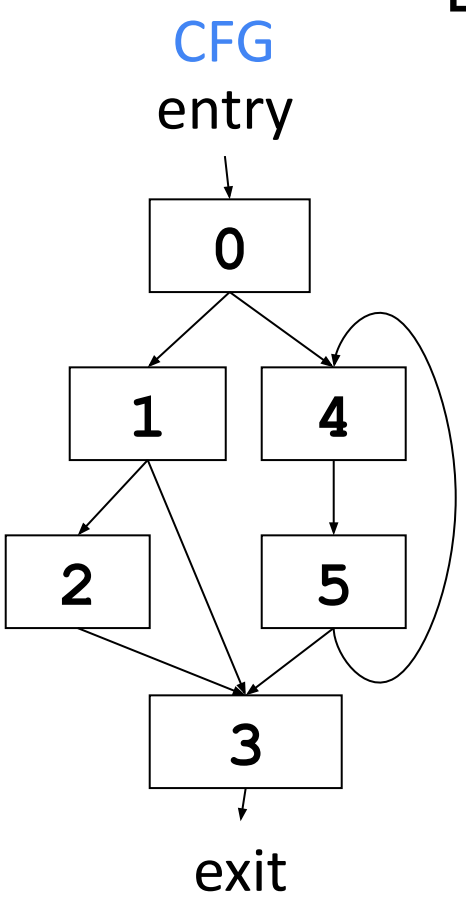
What is a Φ (for a loop) anyway?



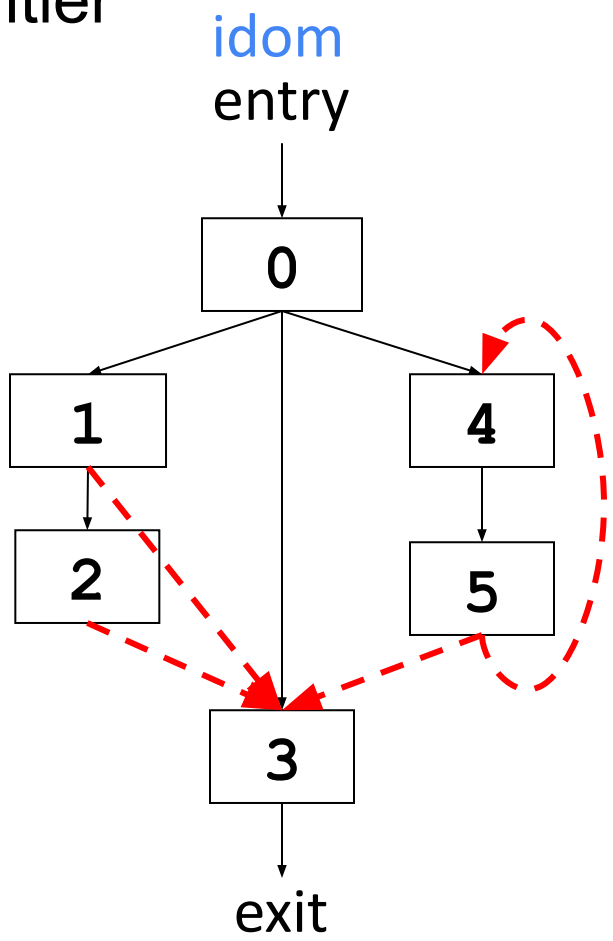
Φs at loop headers relate the dataflow on a loop backedge with the control flow.

Allows finding induction variables really easily.

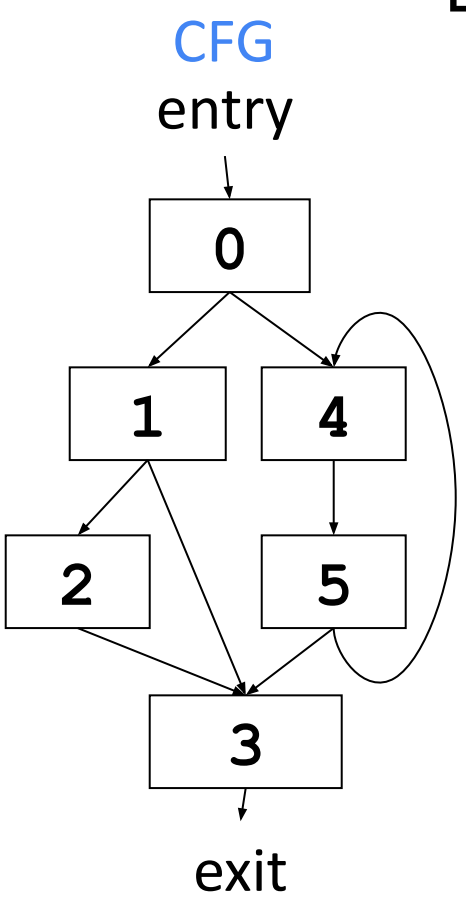
Dominance Frontier



The *dominance frontier* can be determined by looking at CFG edges that ***leave a dominator subtree*** to non-dominated nodes.

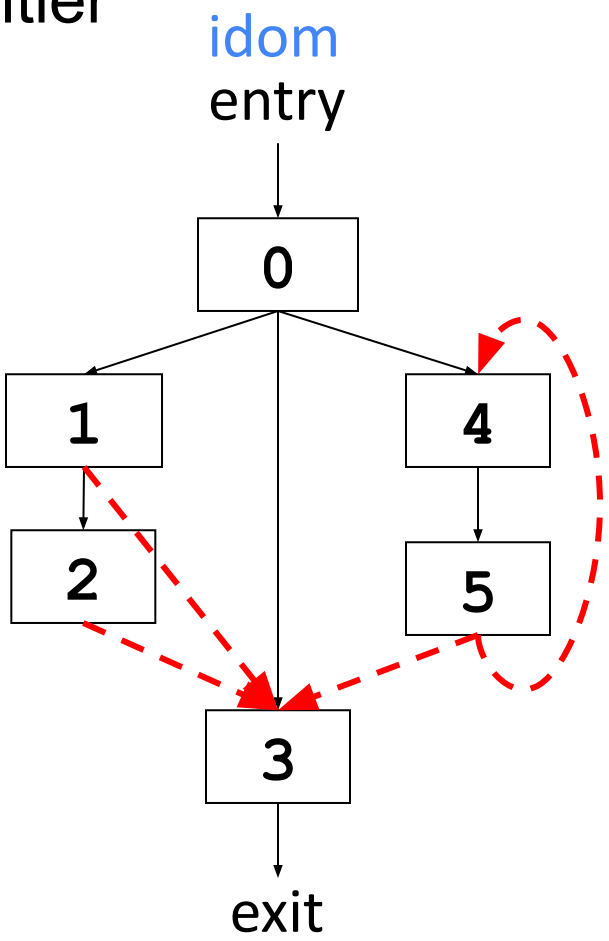


Dominance Frontier



The *dominance frontier* can be determined by looking at CFG edges that

WHAT?



Thinking in SSA: Local Reasoning

- Local Φ simplifications
  - y = Φ(x, x) ↠ x WHAT?
  - y = Φ(x, y) ↠ x WHAT?

Thinking in SSA: Local Reasoning

- Local Φ simplifications
  - y = Φ(x, x) ⇝ x
  - y = Φ(x, y) ⇝ x

A Φ can be simplified if all inputs are the same, or all inputs are the same or the Φ itself.

Thinking in SSA: Local Reasoning

- Constant propagation
  - x = 13;
    y = x + 2;
    z = func(x); ⇝
    w = x - x;

Thinking in SSA: Local Reasoning

- Constant propagation
  - x = 13;
y = x + 2;
z = func(x);
w = x - x;

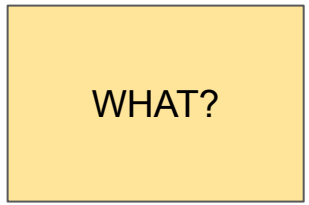
    ⇝

    y = 13 + 2;
z = func(13);
w = 13 - 13;

After SSA construction, any variable which is assigned a constant can be substituted ***everywhere it occurs***.

Thinking in SSA: Local Reasoning

- Copy propagation
  - x = a;
    y = x + 2;
    z = func(x);
    w = x - x;



Thinking in SSA: Local Reasoning

- Copy propagation
  - x = a;
y = x + 2;
z = func(x);
w = x - x;

  ⇝

  y = a + 2;
z = func(a);
w = a - a;

After SSA construction, any variable which is assigned a simple copy can be substituted ***everywhere it occurs***.

Thinking in SSA: Local Reasoning

- Copy propagation
  - x = a;
y = x + 2;
z = func(x);
w = x - x;

y = a + 2;
z = func(a);
w = a - a;

Copies are vestigial.

After SSA construction, any variable which is assigned a simple copy can be substituted ***everywhere it occurs***.

Thinking in SSA: Local Reasoning

- Reasoning about value ranges of Φs
  - x = Φ(3, 7);
    y = x > 0;
    z = x == 12; ⇝
    w = 33 / x;

WHAT?

Thinking in SSA: Local Reasoning

- Reasoning about value ranges of Φs
  - x = Φ(3, 7);
    y = x > 0;
    z = x == 12;
    w = 33 / x;

    ⇝

    y = true;
    z = false;
    w = 33 / x (safe);

Φ inputs represent all the possible values the phi could take on across all control flow. Therefore some conditions are statically decidable.

Thinking in SSA: Local Reasoning

- Dominating conditions
  - if (x == 0) return x; ⇝ if (x == 0) return WHAT?
  - if (x != null) return x.f; ⇝ if (x != null) return WHAT?
  - if (x != 0) return 1000 / x; ⇝ if (x != 0) return WHAT?

Thinking in SSA: Local Reasoning

- Dominating conditions
  - if (x == 0) return x; ⇝ if (x == 0) return 0;
  - if (x != null) return x.f; ⇝ if (x != null) return x.f (safe);
  - if (x != 0) return 1000 / x; ⇝ if (x != 0) return 1000 / x;

Any branch can assume the condition is ***true or false*** on the respective output control flow edges.

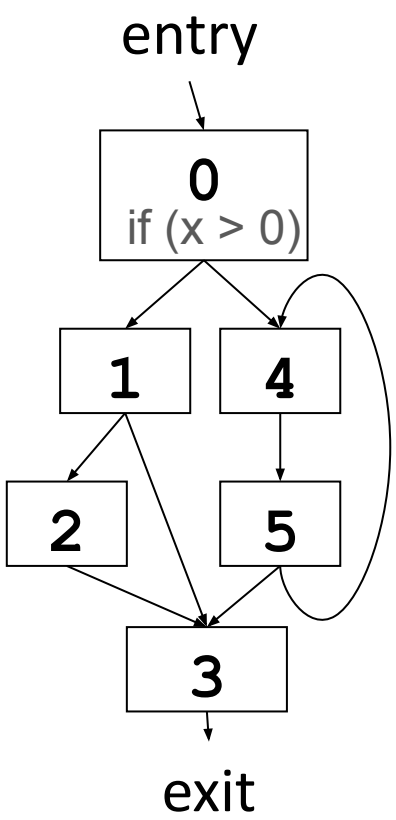
Thinking in SSA: Local Reasoning

- Dominating conditions
  - if (x == 0) return x; ⇝ if (x == 0) return 0;
  - if (x != null) return x.f; ⇝ if (x != null) return x.f (safe);
  - if (x != 0) return 1000 / x; ⇝ if (x != 0) return 1000 / x;

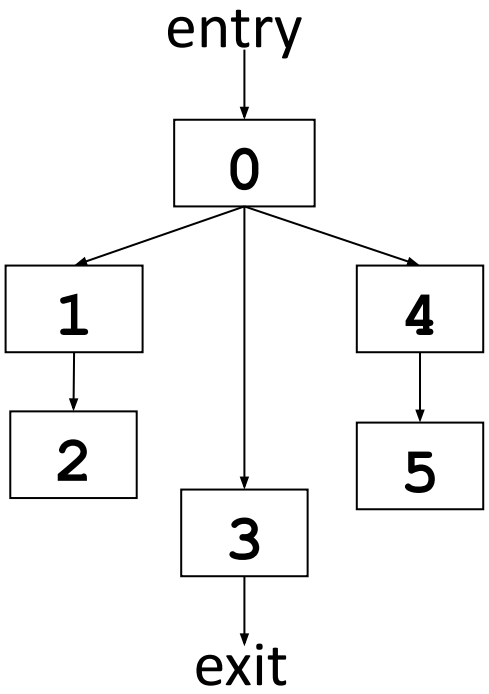
Any branch can assume the condition is ***true or false*** on the respective output control flow edges.

Thinking in SSA: Local Reasoning

CFG



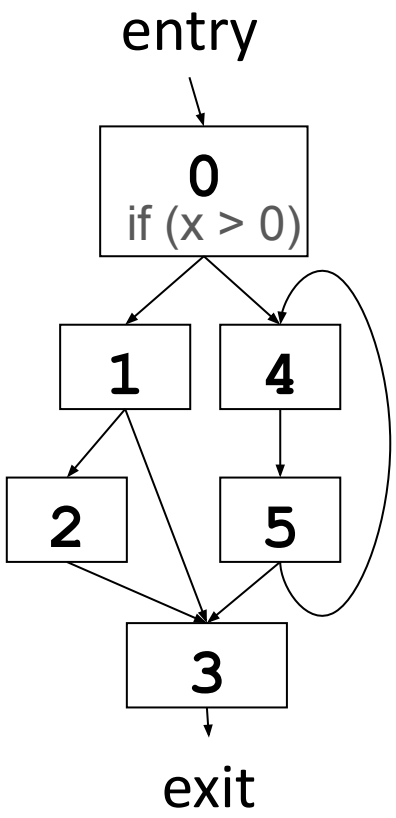
Dominator Tree



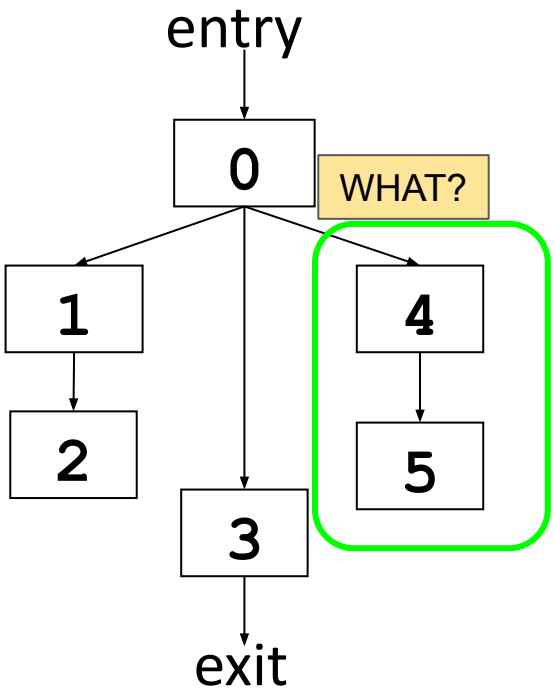
Any branch can assume the condition is true or false on the respective output control flow edges ***and their dominated blocks.***

Thinking in SSA: Local Reasoning

CFG



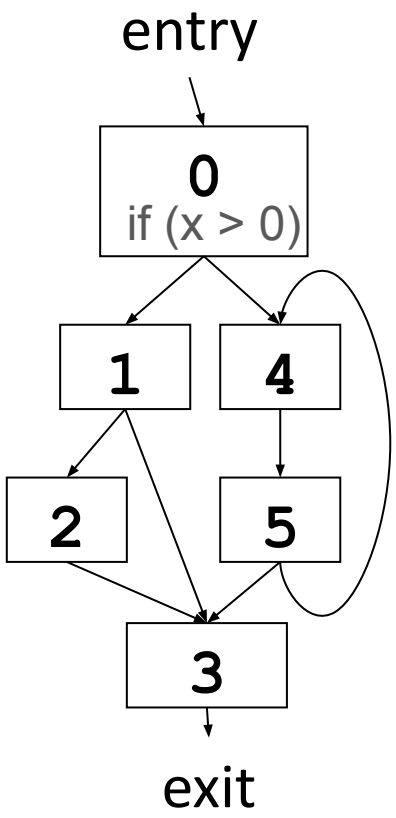
Dominator Tree



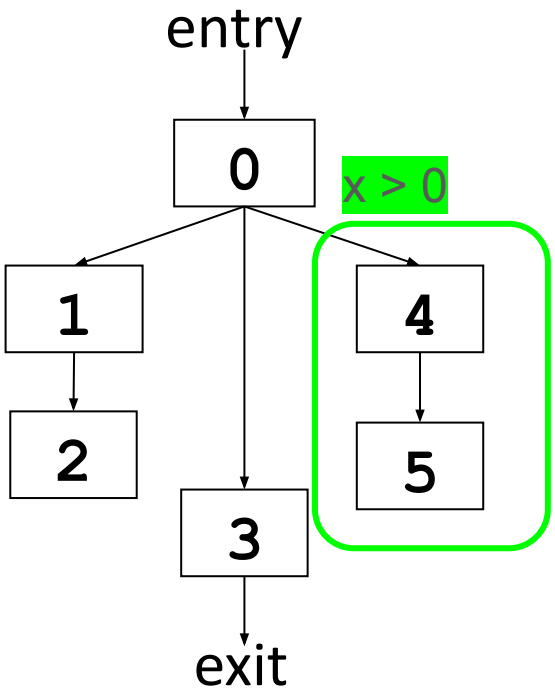
Any branch can assume the condition is true or false on the respective output control flow edges ***and their dominated blocks.***

Thinking in SSA: Local Reasoning

CFG



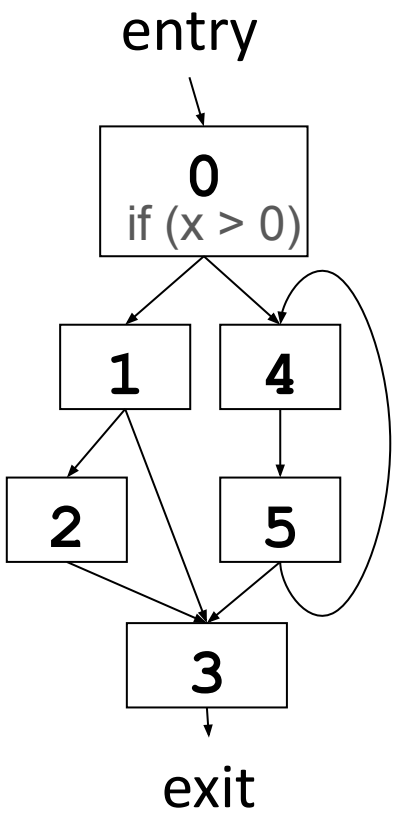
Dominator Tree



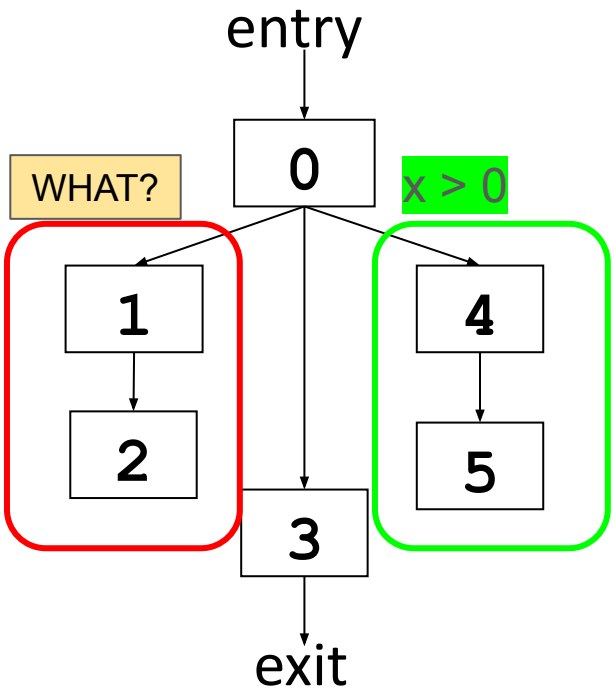
Any branch can assume the condition is true or false on the respective output control flow edges ***and their dominated blocks.***

Thinking in SSA: Local Reasoning

CFG



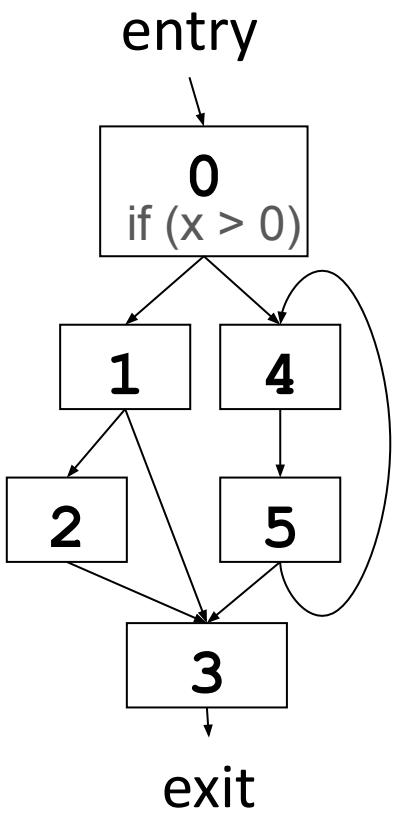
Dominator Tree



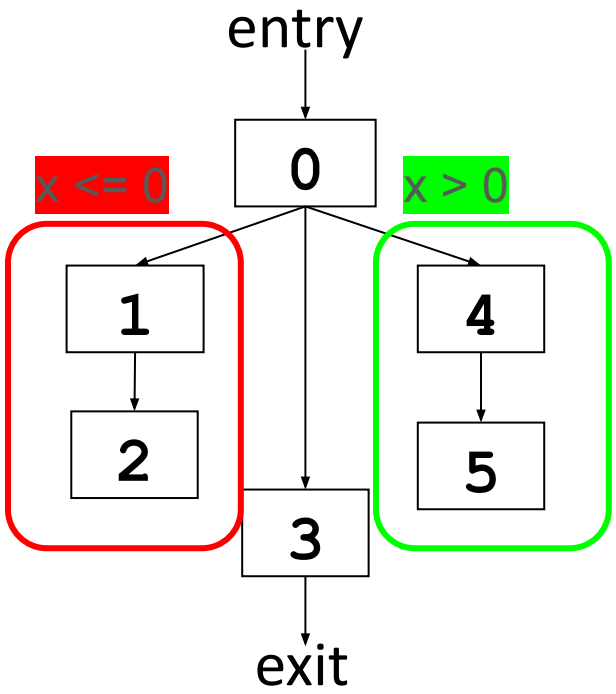
Any branch can assume the condition is true or false on the respective output control flow edges ***and their dominated blocks.***

Thinking in SSA: Local Reasoning

CFG



Dominator Tree



Any branch can assume the condition is true or false on the respective output control flow edges ***and their dominated blocks.***

Thinking in SSA: Local Reasoning

- Branch folding
  - if (x == x) {
    y_1 = a + 3;
    } else {
    y_2 = a + 5;
    }
    $y_3 = \Phi(y_1, y_2)$

WHAT?

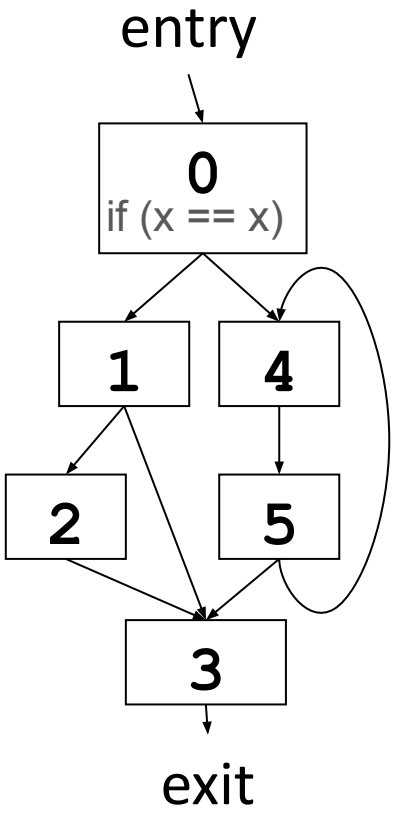
Thinking in SSA: Local Reasoning

- Branch folding
  - if (x == x) {
  y_1 = a + 3;
} else {
  y_2 = a + 5;
}
$y_3 = \Phi(y_1, y_2)$

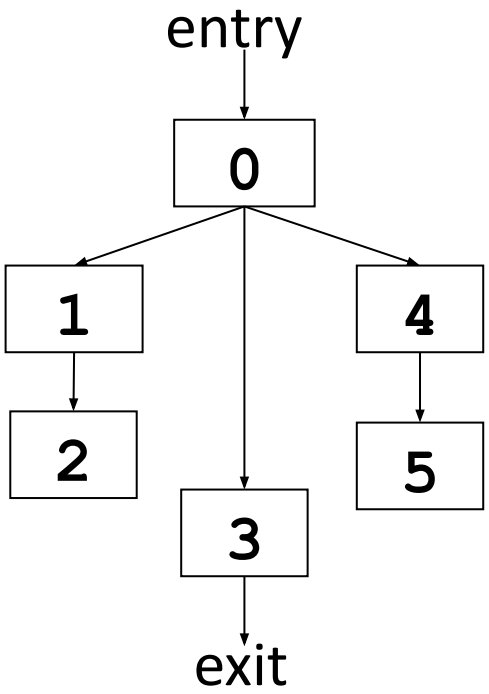
if (true) {
  y_1 = a + 3;
} else {
  y_2 = a + 5;
}
$y_3 = y_1$

Any branch with a constant condition can be folded and edges removed from the CFG.

Thinking in SSA: Local Reasoning

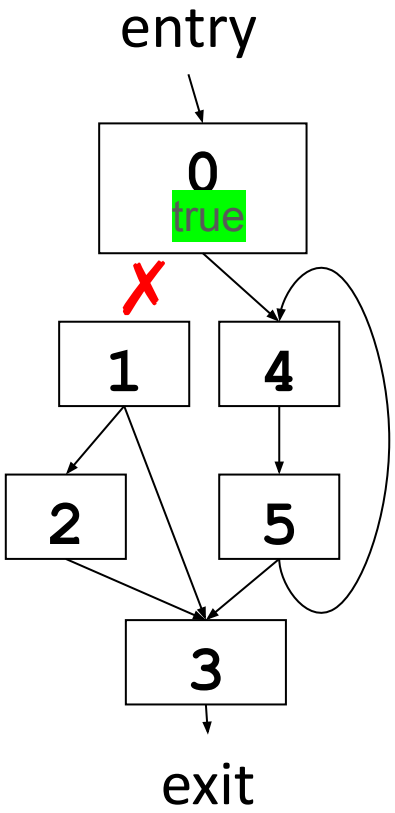


Dominator Tree

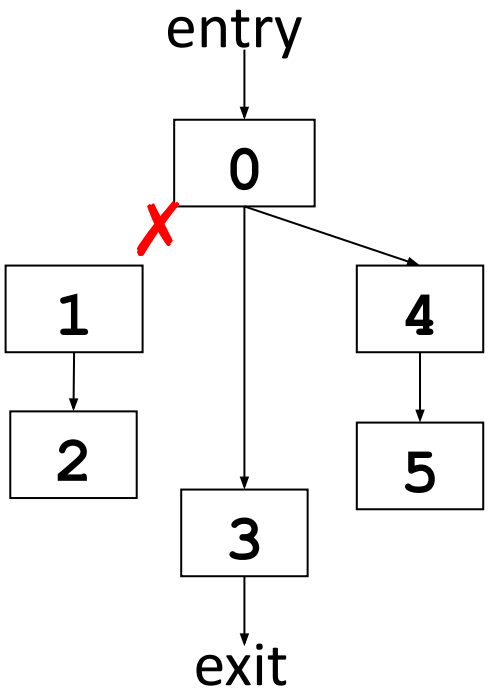


Any branch with a constant condition can be folded and edges removed from the CFG.

Thinking in SSA: Local Reasoning

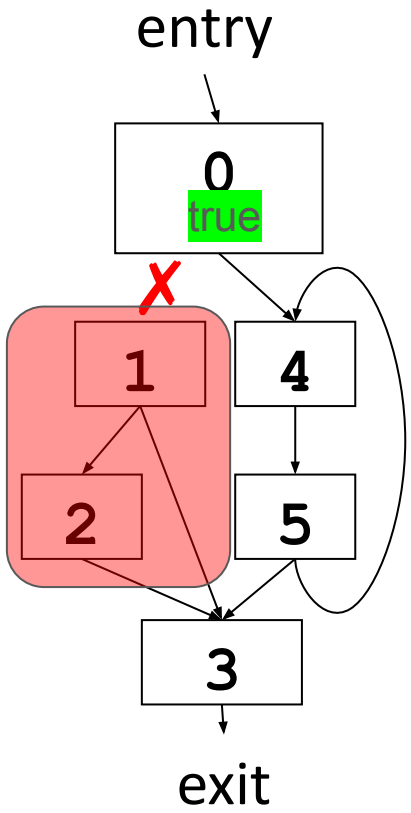


Dominator Tree

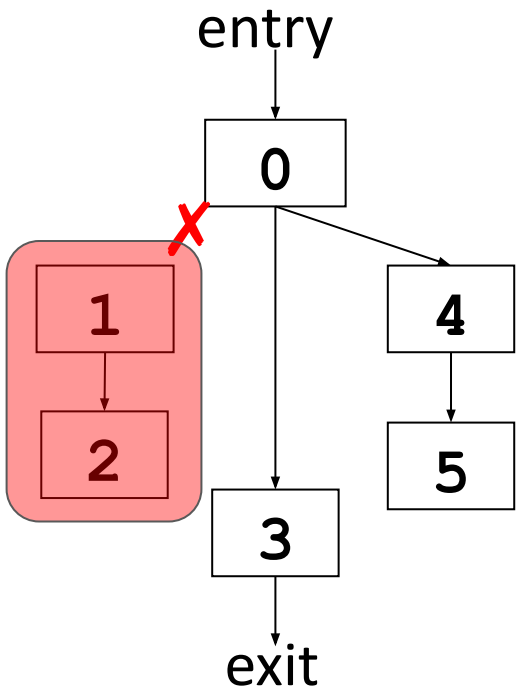


The entire dominator subtree will become dead and can be removed.

Thinking in SSA: Local Reasoning

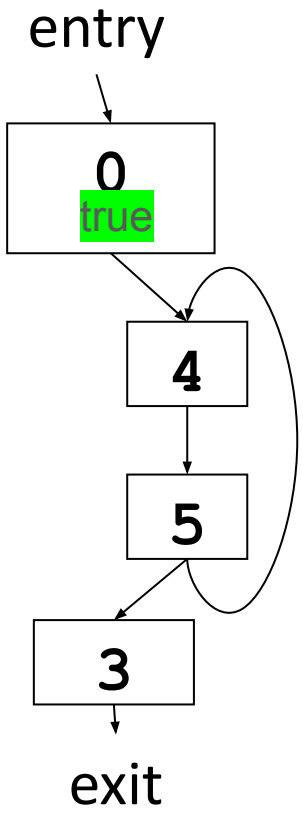


Dominator Tree

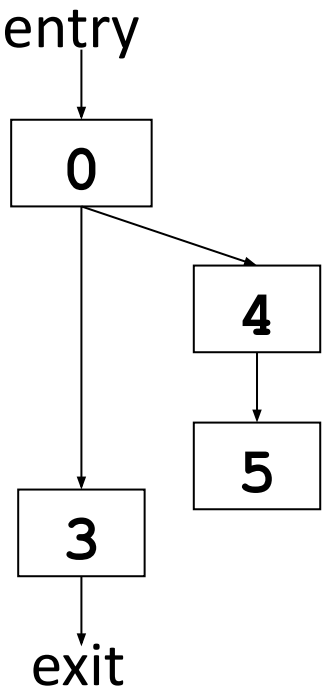


The entire dominator subtree will become dead and can be removed.

Thinking in SSA: Local Reasoning

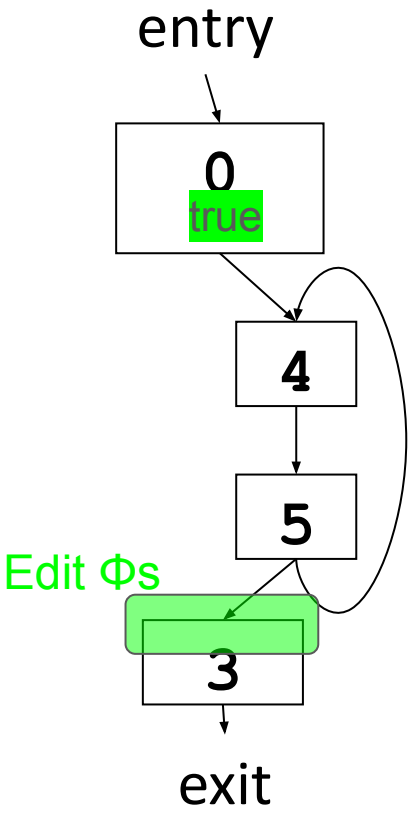


Dominator Tree

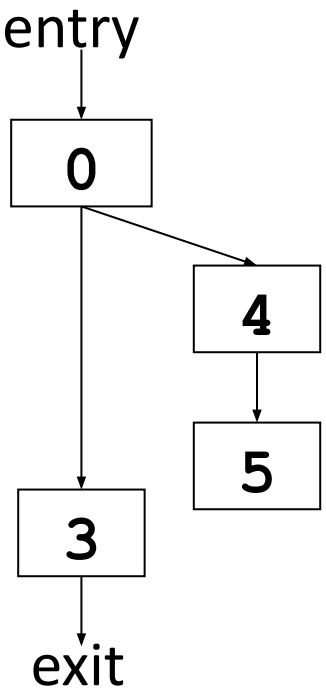


The entire dominator subtree will become dead and can be removed.

Thinking in SSA: Local Reasoning

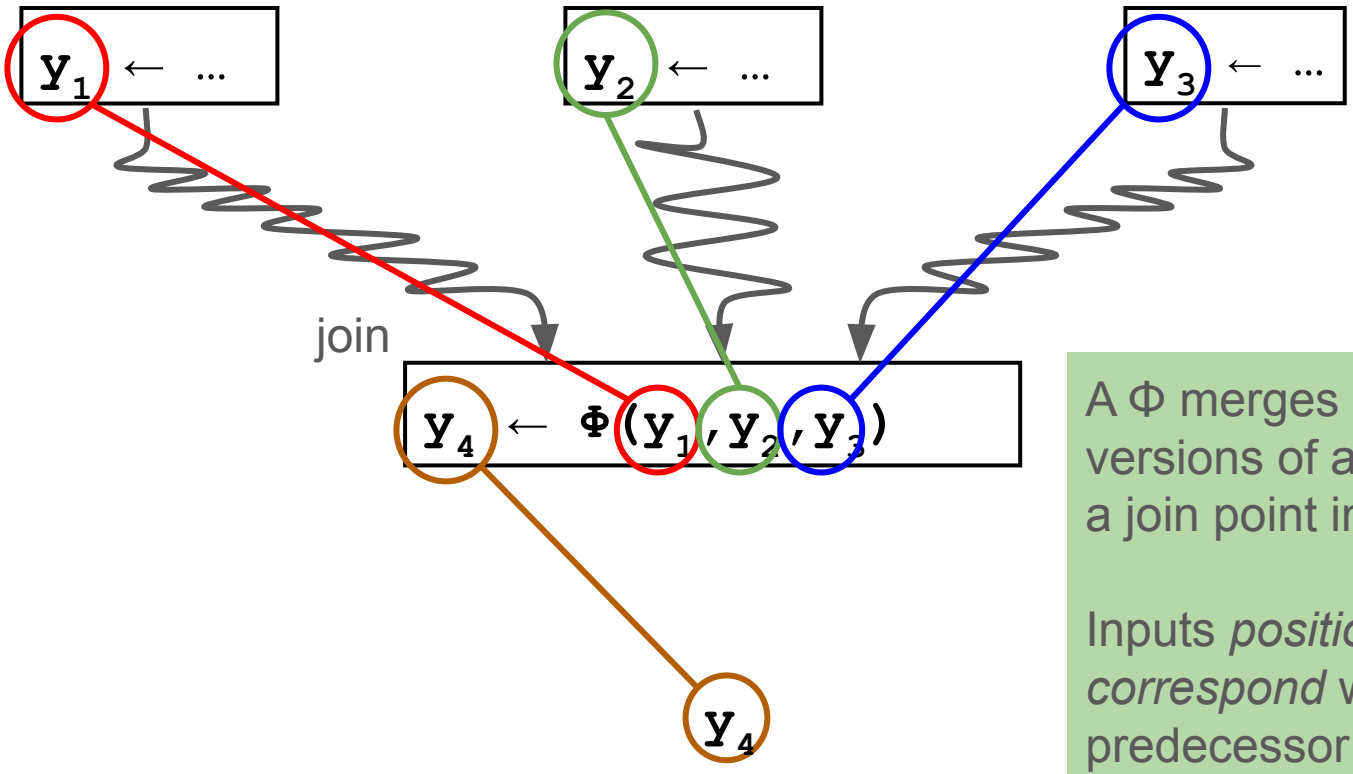


Dominator Tree



To maintain SSA, any Φs at join points with dead predecessors need to be edited and simplified.

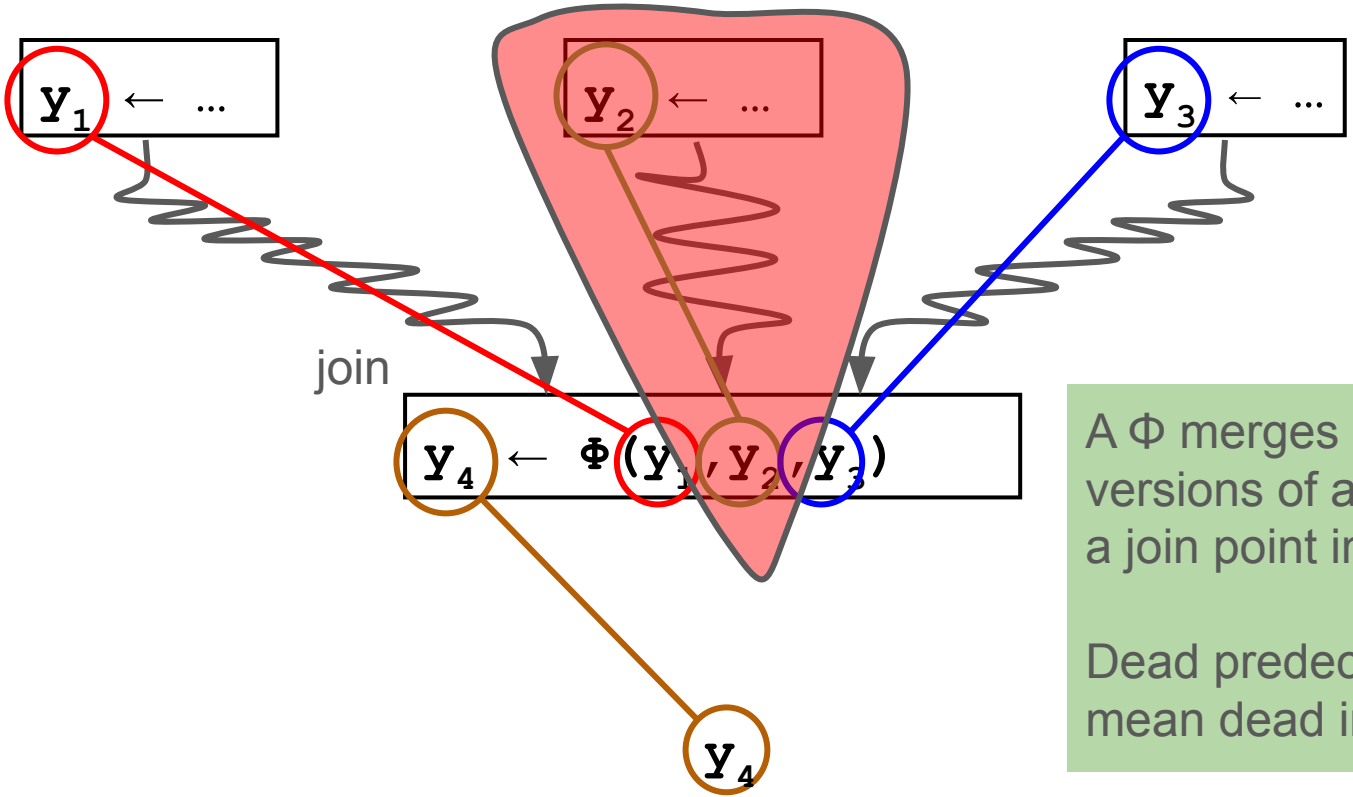
Editing and simplifying Φs



A Φ merges multiple versions of a variable at a join point in the CFG.

Inputs *positionally correspond* with predecessor edges.

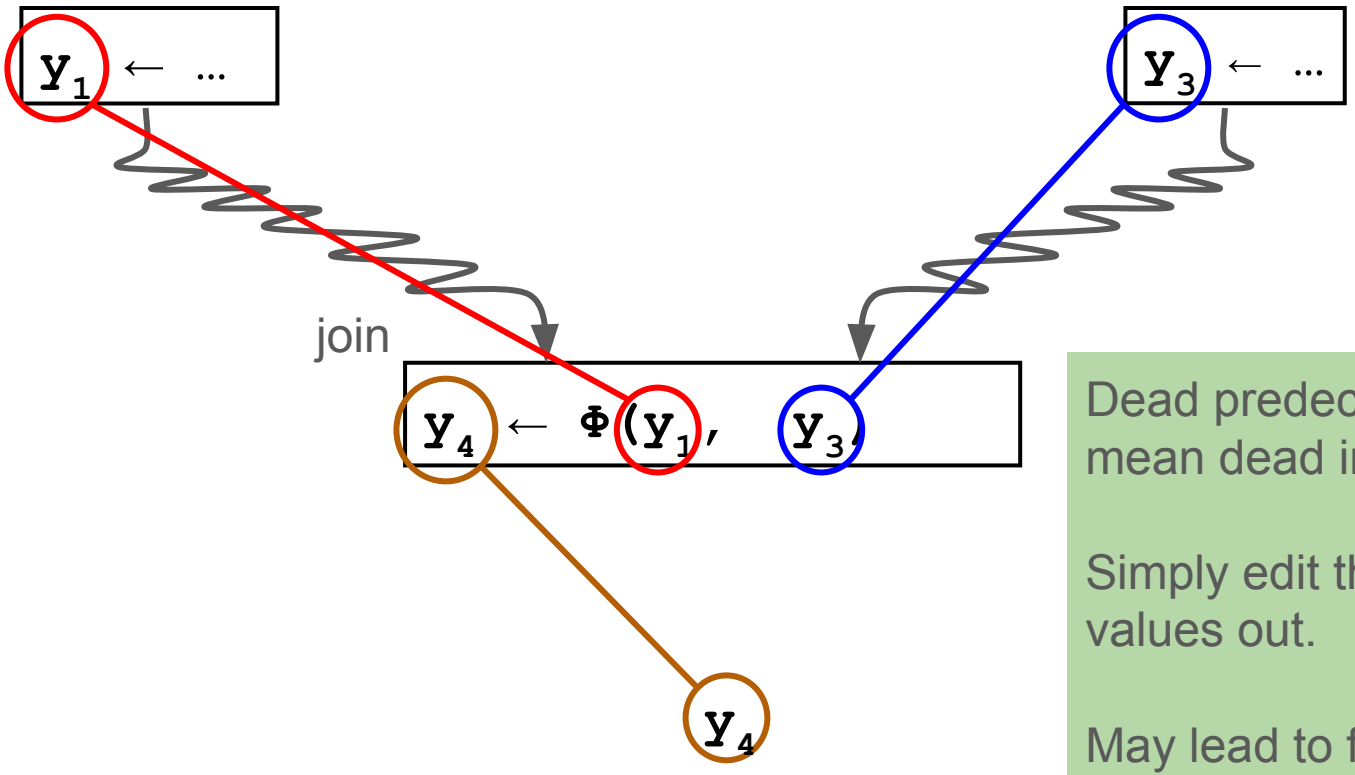
Editing and simplifying Φs



A Φ merges multiple versions of a variable at a join point in the CFG.

Dead predecessors mean dead input values.

Editing and simplifying Φs



Dead predecessors mean dead input values.

Simply edit the input values out.

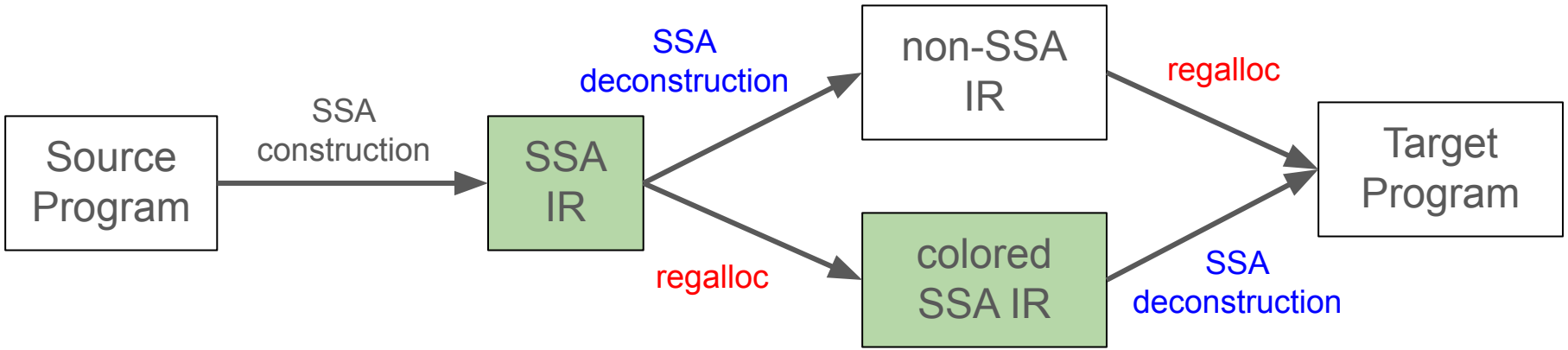
May lead to further simplification.

Deconstructing SSA

- Real machines don’t have Φ functions.
- Have to insert moves at predecessors.
- Mentioned earlier, but with huge caveats.
- We resolve those caveats today.

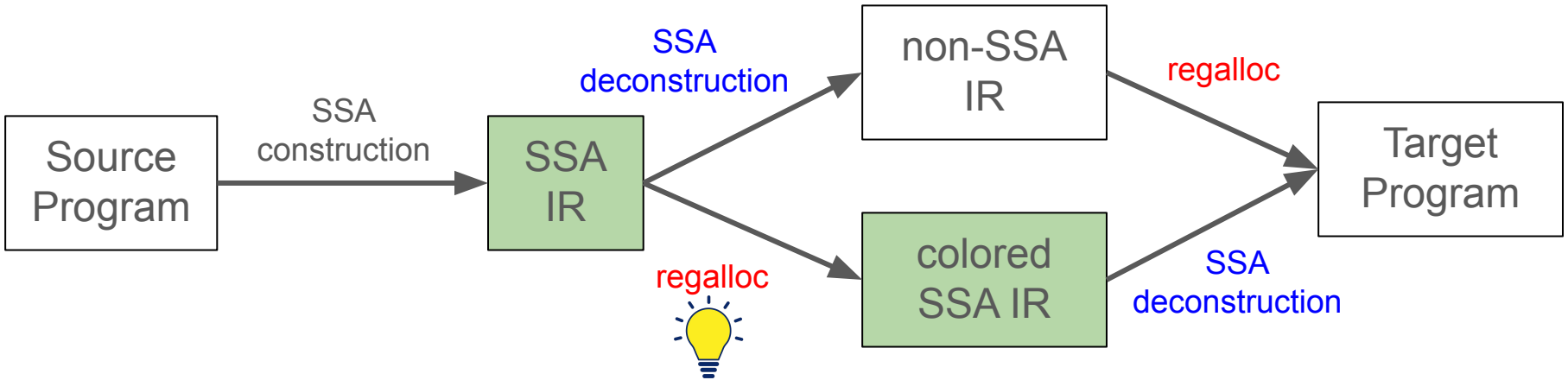
Deconstructing SSA

- When during compilation to deconstruct SSA?
- There are two common choices: before or after regalloc.
- Regalloc before deconstruction is relatively new (2010s).



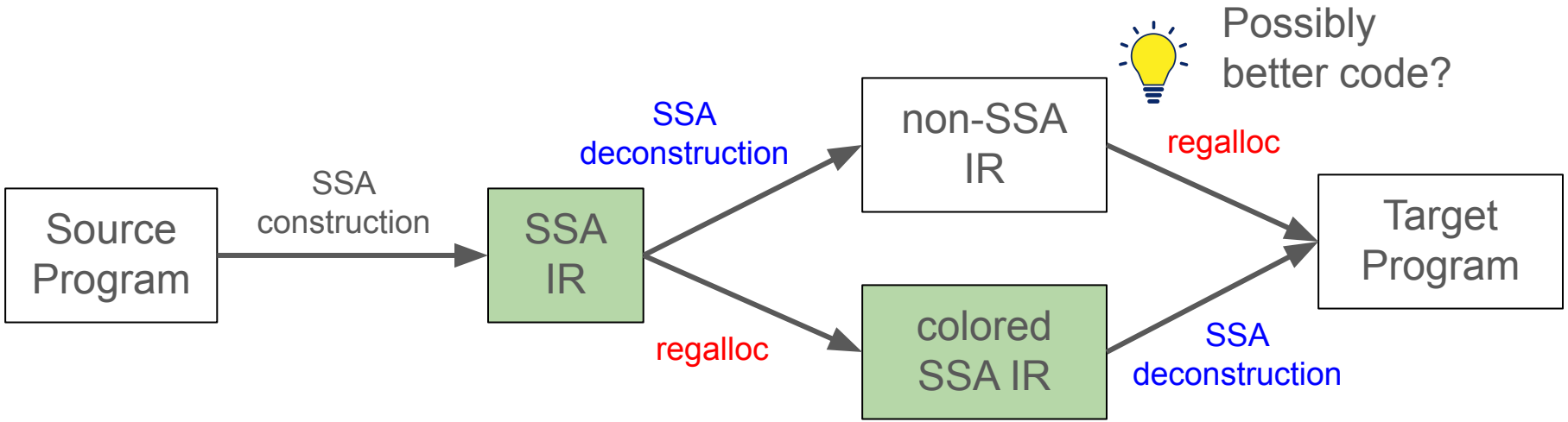
Deconstructing SSA

- When during compilation to deconstruct SSA?
- There are two common choices: before or after regalloc.
- Regalloc before deconstruction is relatively new (2010s).



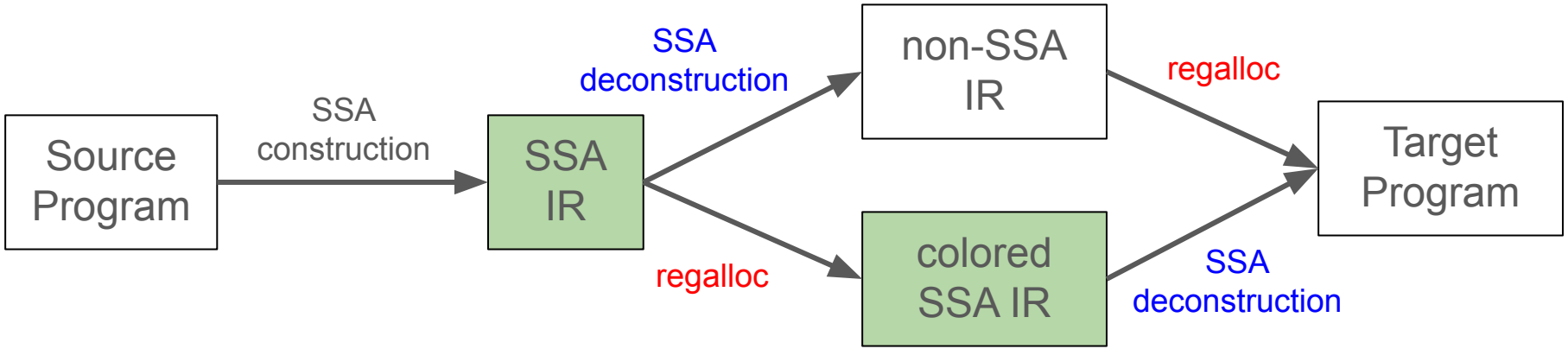
Deconstructing SSA

- When during compilation to deconstruct SSA?
- There are two common choices: before or after regalloc.
- Regalloc before deconstruction is relatively new (2010s).



Deconstructing SSA

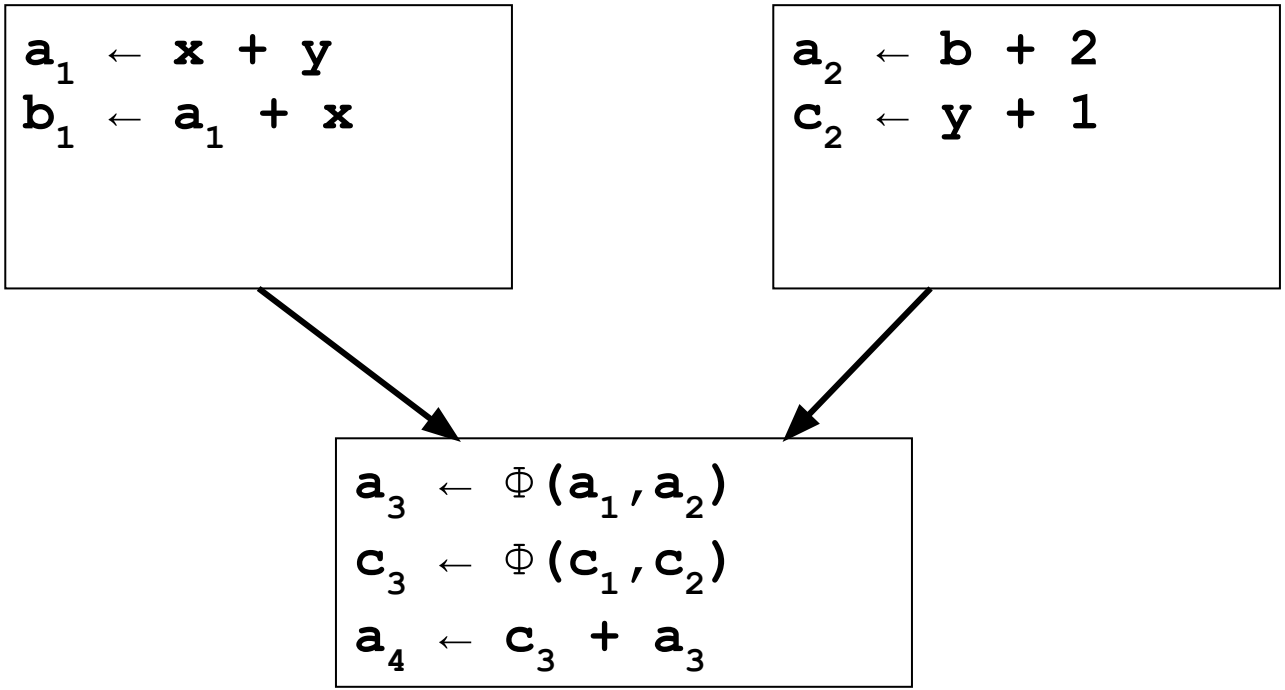
- When during compilation to deconstruct SSA?
- There are two common choices: before or after regalloc.
- Regalloc before deconstruction is relatively new (2010s).



Deconstruction is more or less the same either way.

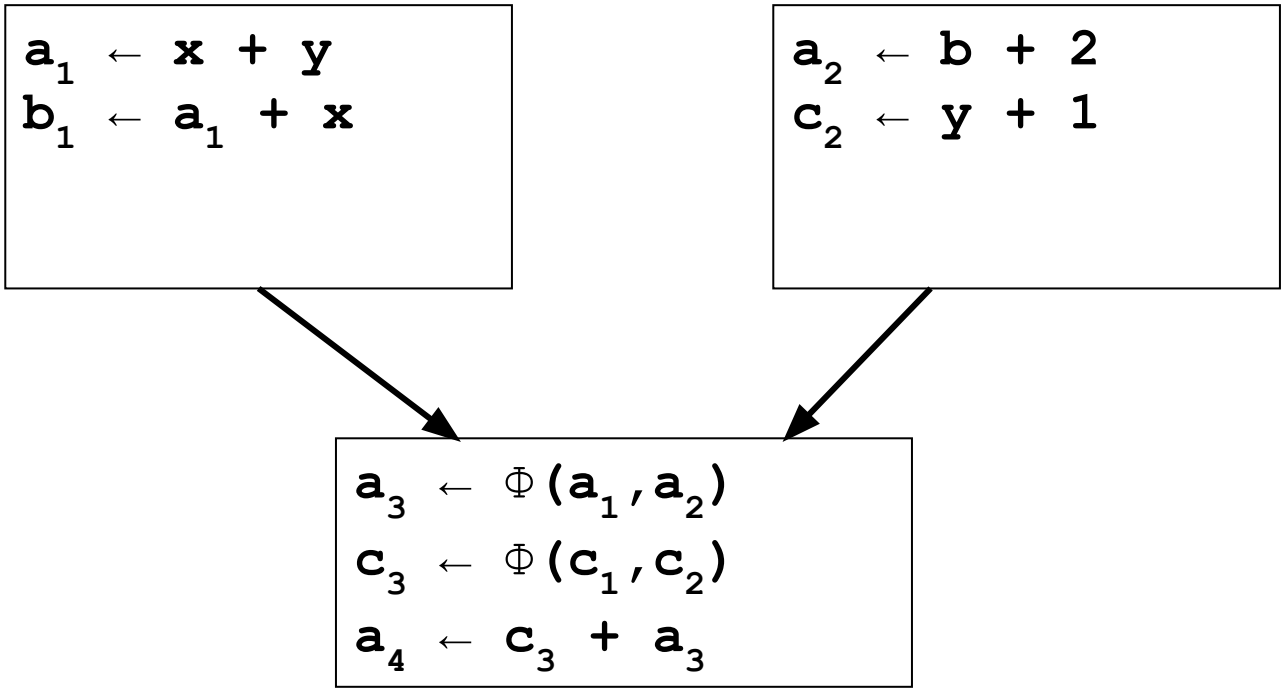
Deconstructing SSA

- Insert moves according to the positional correspondence of inputs.



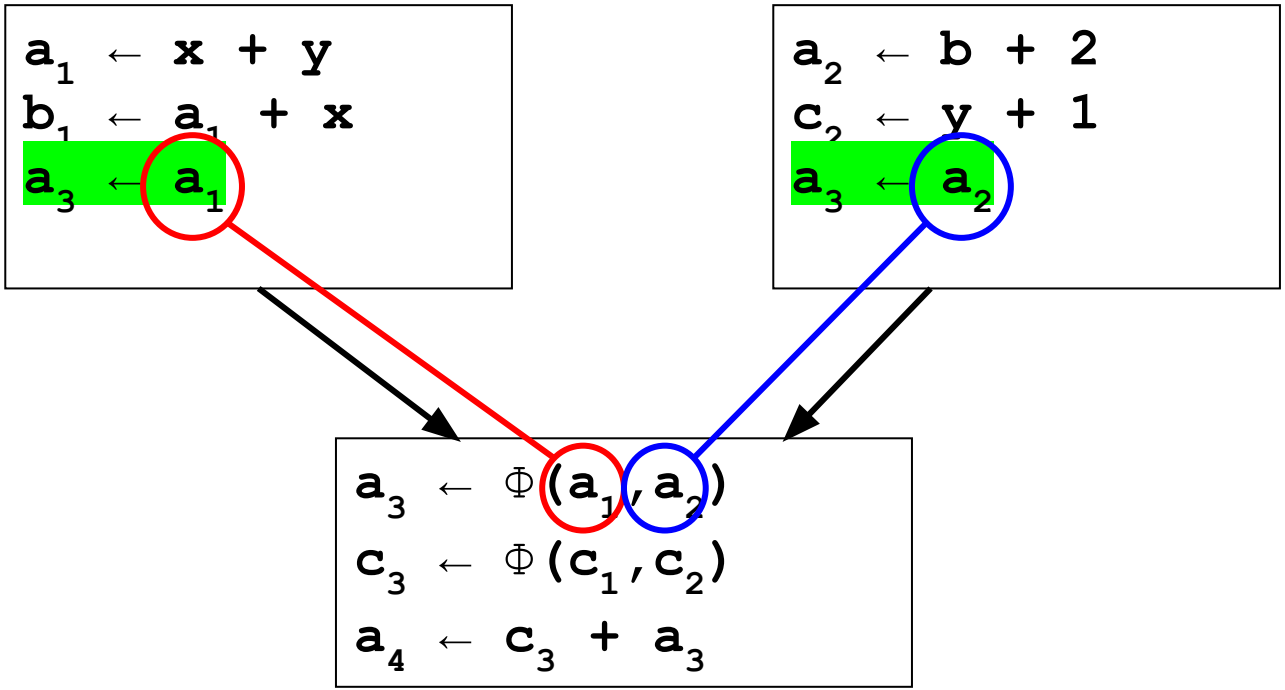
Deconstructing SSA

- Insert *Φ-resolution moves* and remove Φs.



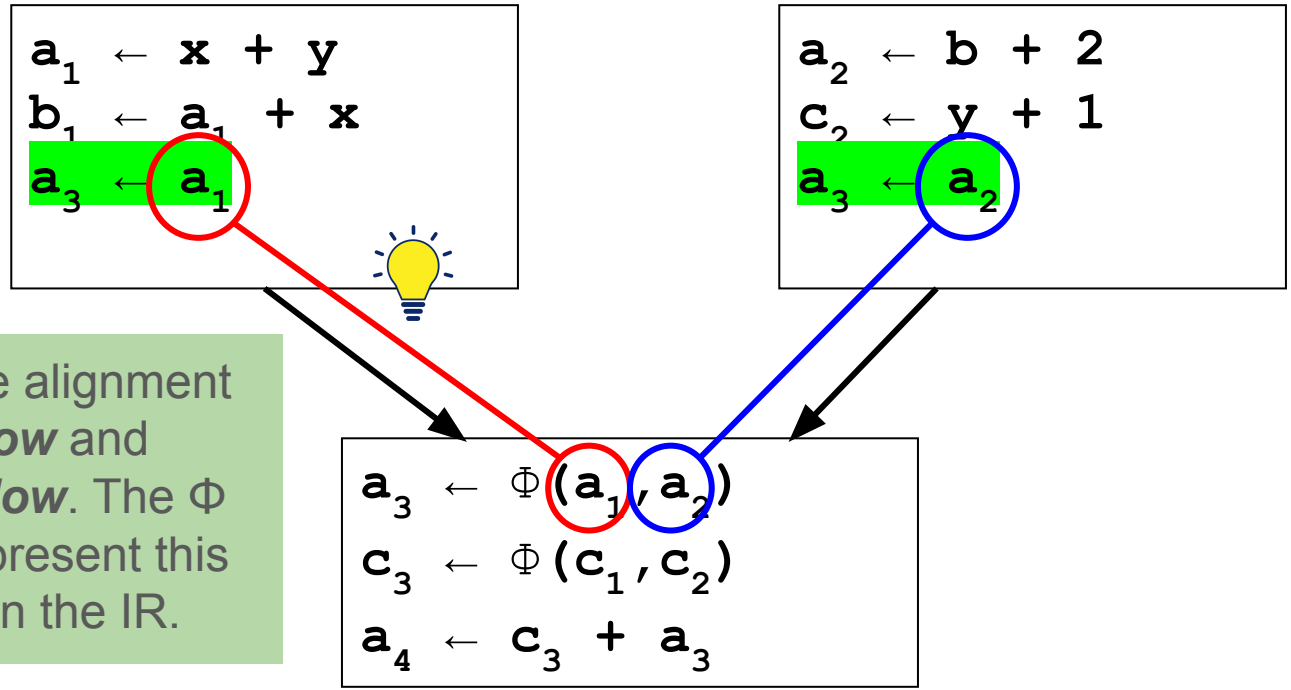
Deconstructing SSA

- Insert moves according to the positional correspondence of inputs.



Deconstructing SSA

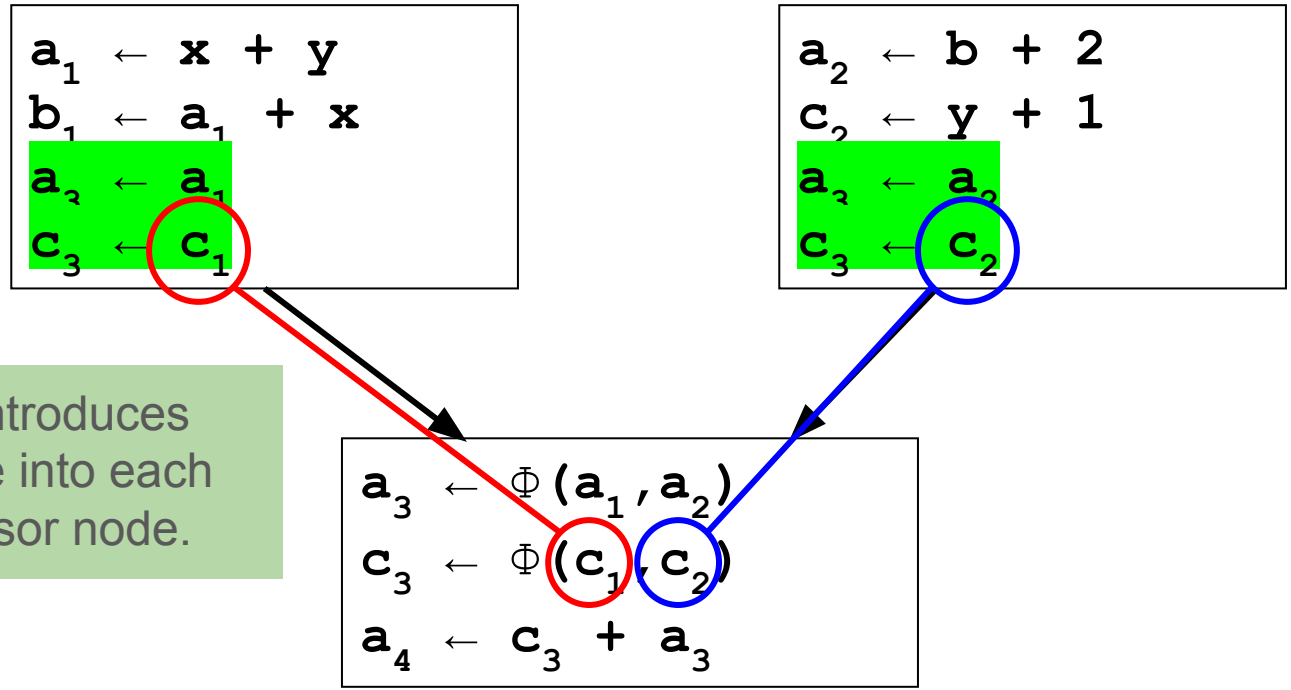
- Insert moves according to the positional correspondence of inputs.



Notice the alignment of ***data flow*** and ***control flow***. The Φ nodes represent this explicitly in the IR.

Deconstructing SSA

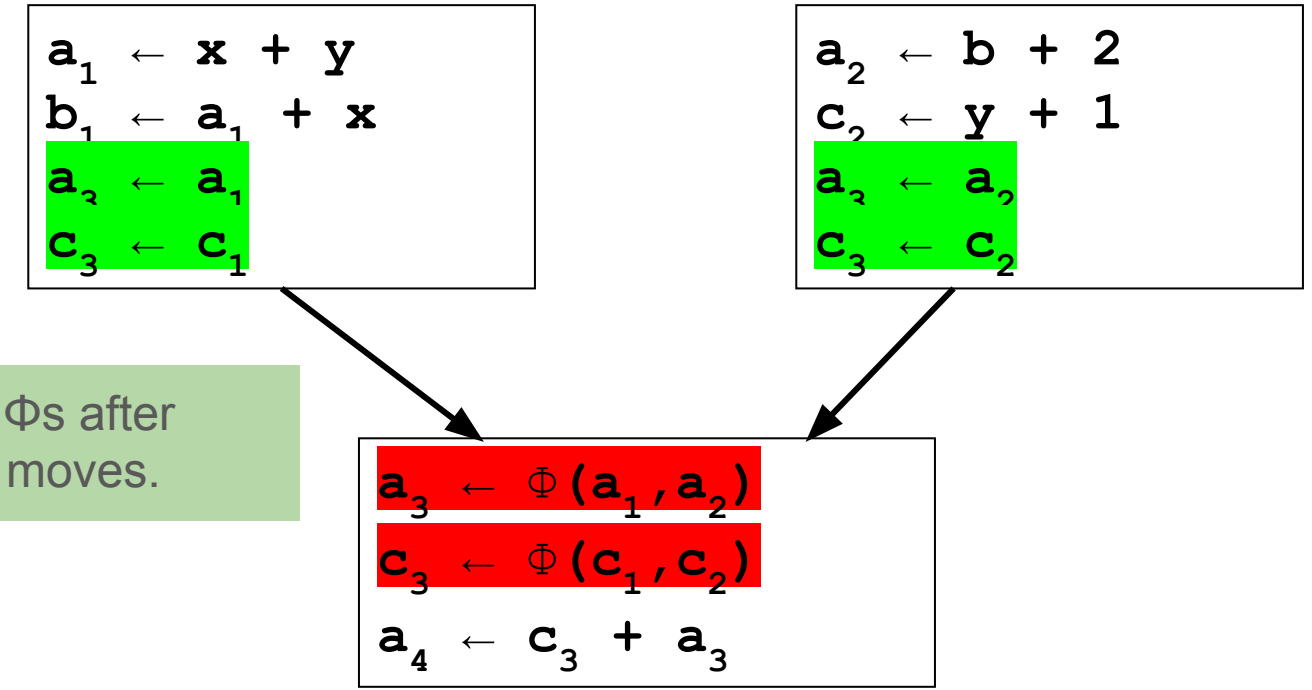
- Insert moves according to the positional correspondence of inputs.



Each Φ introduces one move into each predecessor node.

Deconstructing SSA

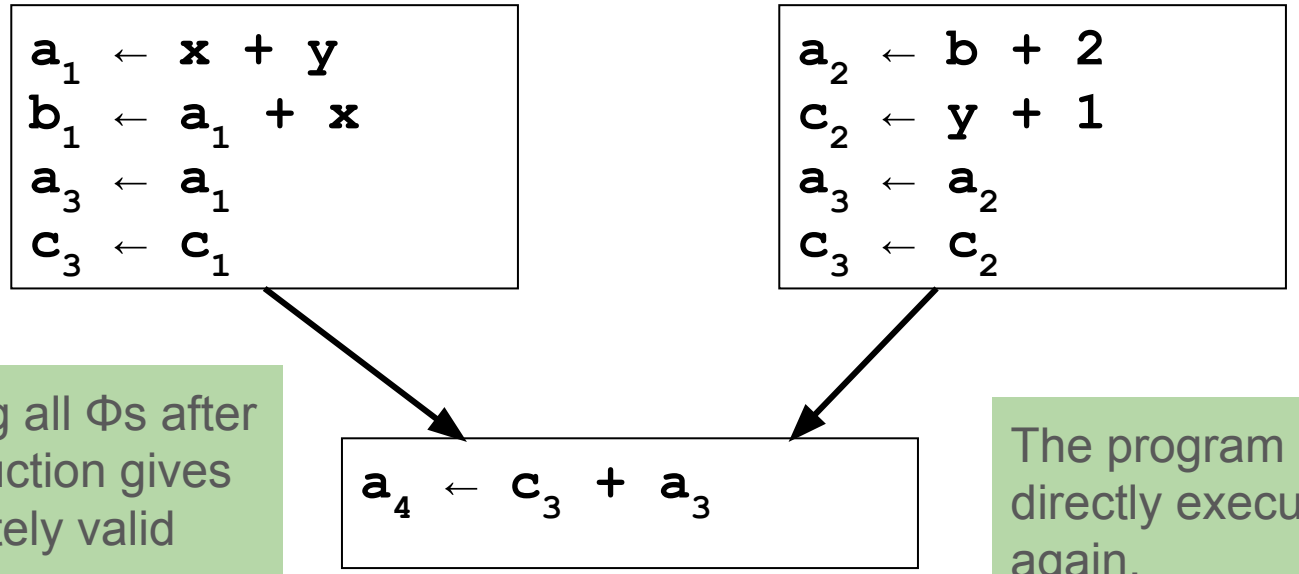
- Insert moves according to the positional correspondence of inputs.



Remove Φs after inserting moves.

Deconstructing SSA

- Insert moves according to the positional correspondence of inputs.

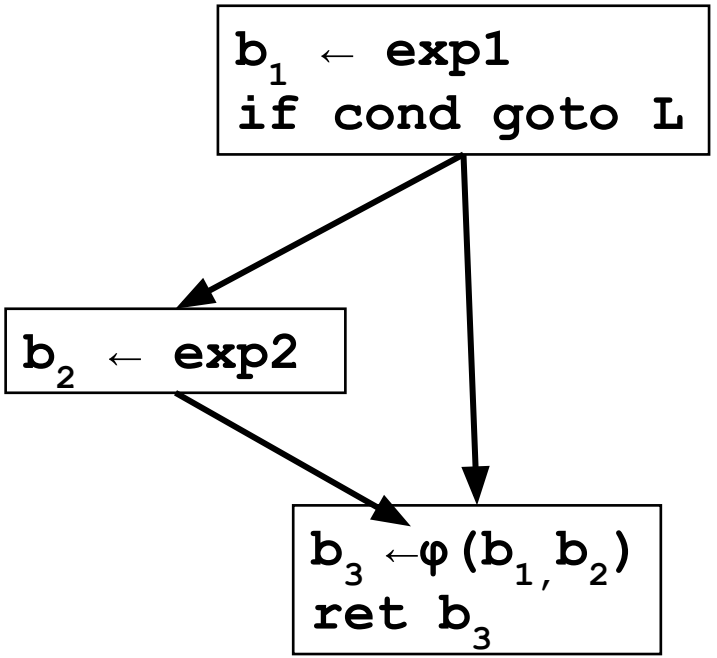


Removing all Φs after deconstruction gives a completely valid non-SSA program.

The program is now directly executable again.

Issue 1: Critical Edges

- Consider a simple triangle CFG.

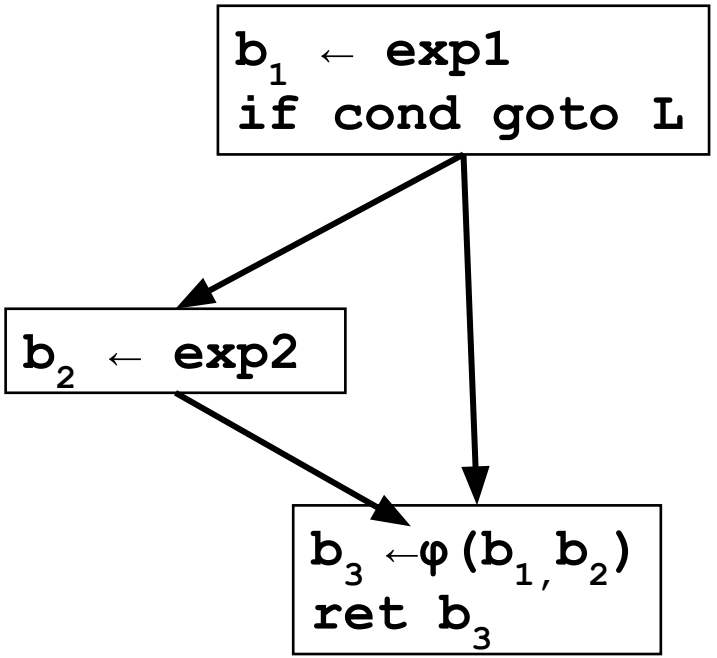


Issue 1: Critical Edges

- Consider a simple triangle CFG.

But wait, there's a bug in the SSA for this program!

What's the bug?

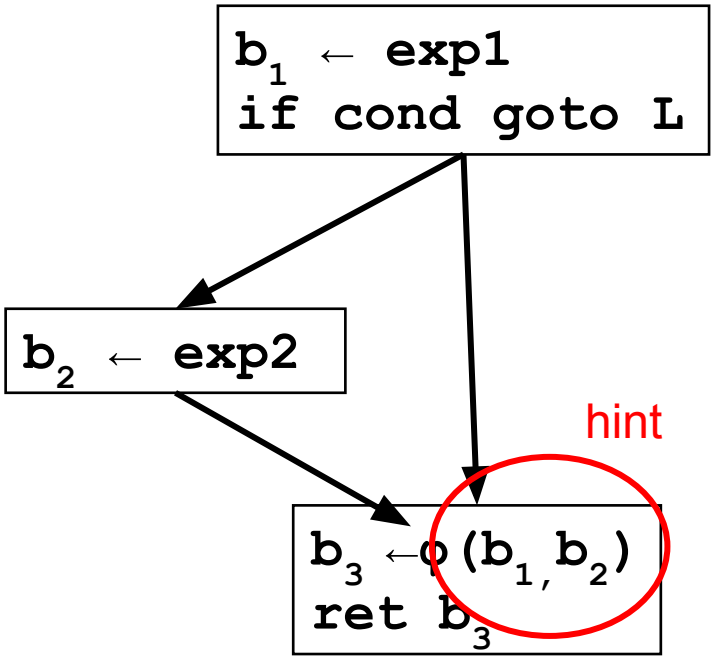


Issue 1: Critical Edges

- Consider a simple triangle CFG.

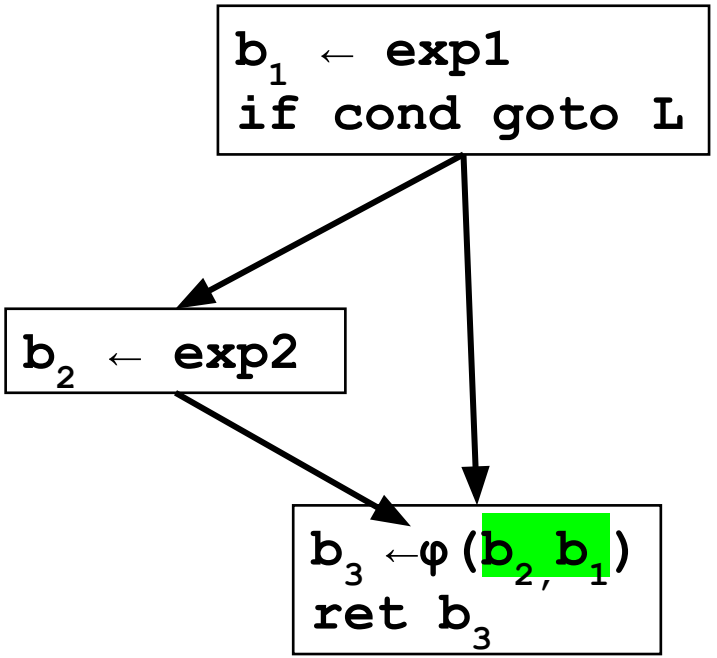
But wait, there's a bug in the SSA for this program!

What's the bug?



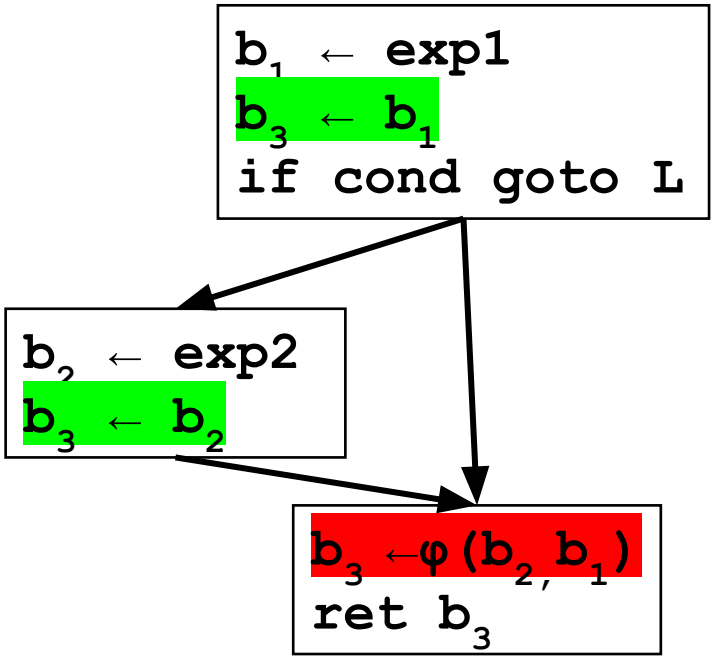
Issue 1: Critical Edges

- Consider a simple triangle CFG.



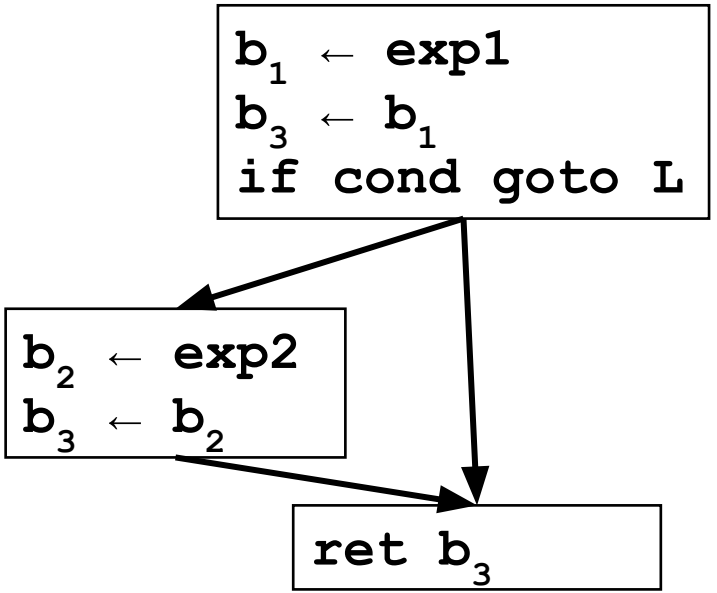
Issue 1: Critical Edges

- Consider a simple triangle CFG.
- We insert moves in both predecessors and remove the Φ.



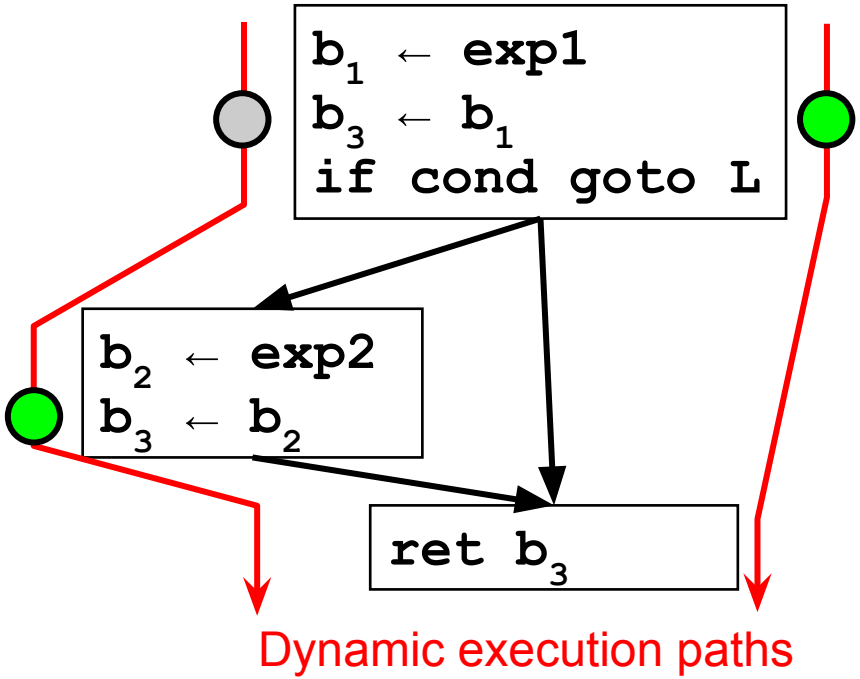
Issue 1: Critical Edges

- Consider a simple triangle CFG.
- We insert moves in both predecessors and remove the Φ.



Issue 1: Critical Edges

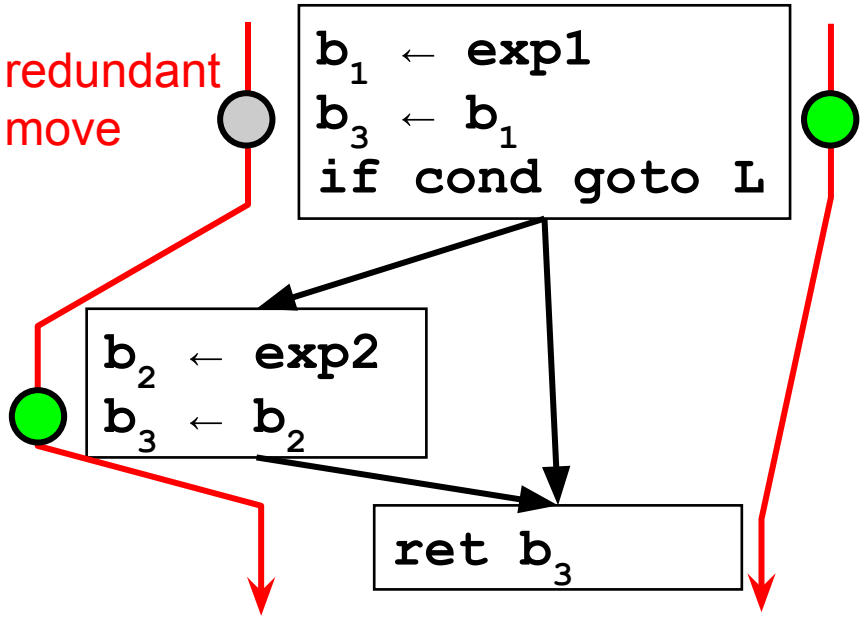
- Consider a simple triangle CFG.
- We insert moves in both predecessors and remove the Φ.



Issue 1: Critical Edges

- Consider a simple triangle CFG.
- We insert moves in both predecessors and remove the Φ.

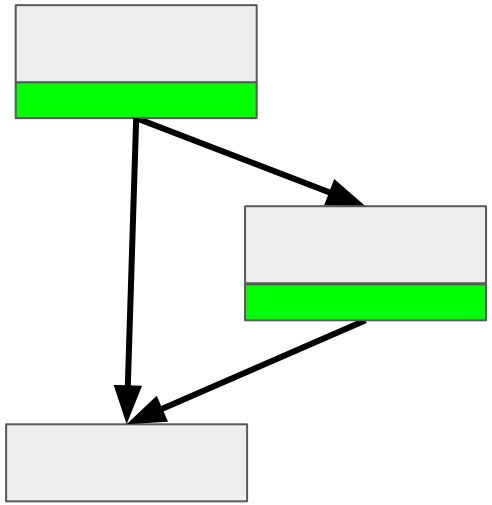
Naïve insertion can introduce redundant code on some execution paths.



Issue 1: Critical Edges

- Consider a simple triangle CFG.
- We insert moves in both predecessors and remove the Φ.

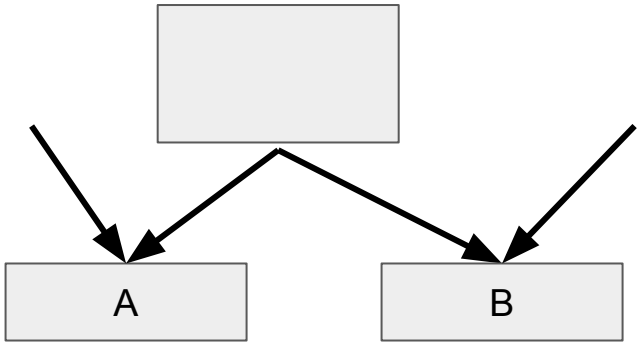
Naïve insertion can introduce redundant code on some execution paths.



Issue 1: Critical Edges

- Consider a *more complicated* CFG.
- We insert moves in *all* predecessors and remove the Φ.

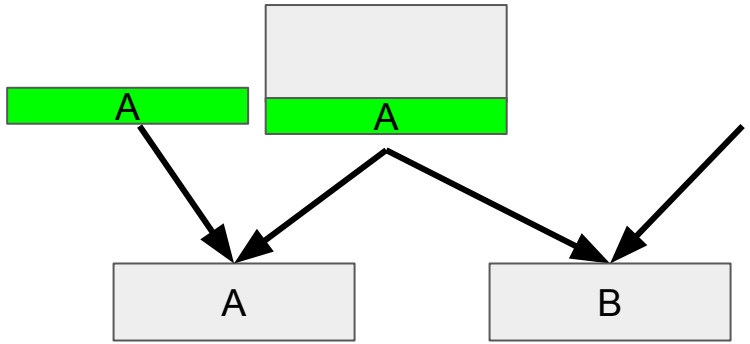
Naïve insertion can introduce redundant code on some execution paths.



Issue 1: Critical Edges

- Consider a *more complicated* CFG.
- We insert moves in *all* predecessors and remove the Φ.

Naïve insertion can introduce redundant code on some execution paths.

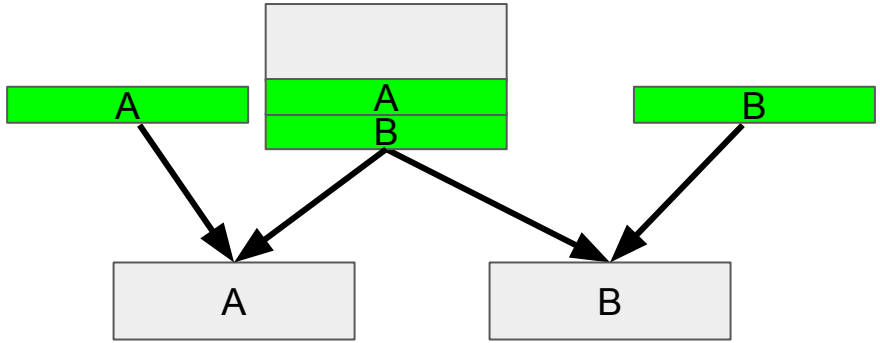


Issue 1: Critical Edges

- Consider a *more complicated* CFG.
- We insert moves in *all* predecessors and remove the Φ.

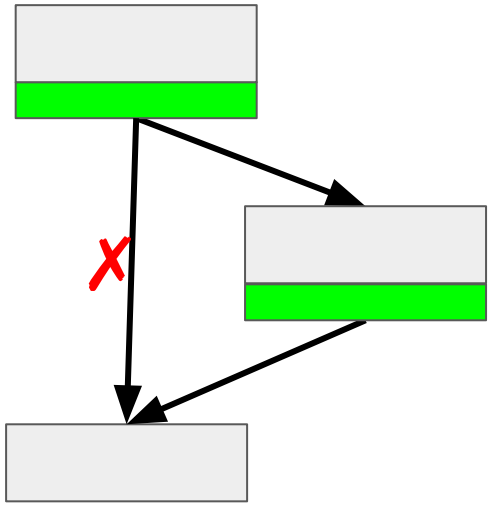
Naïve insertion can introduce redundant code on some execution paths.

Can actually get *really* bad.



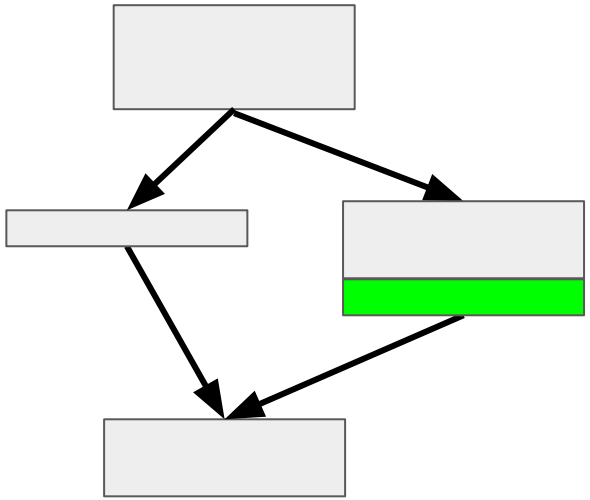
Splitting Critical Edges

- To avoid redundant moves, split *critical edges* by inserting an empty block between.



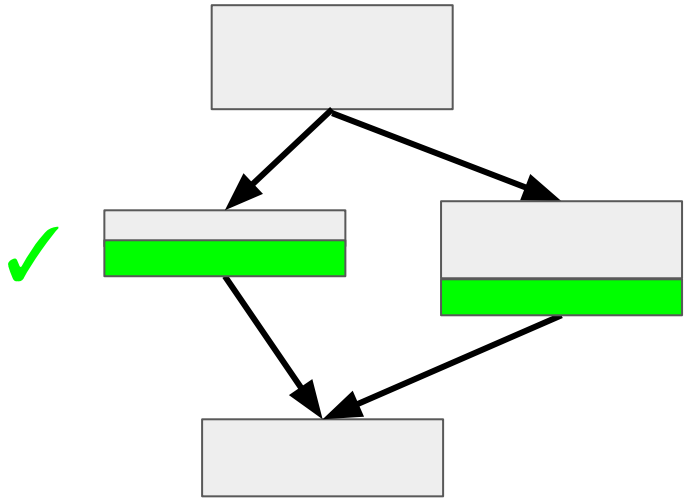
Splitting Critical Edges

- To avoid redundant moves, split *critical edges* by inserting an empty block between.



Splitting Critical Edges

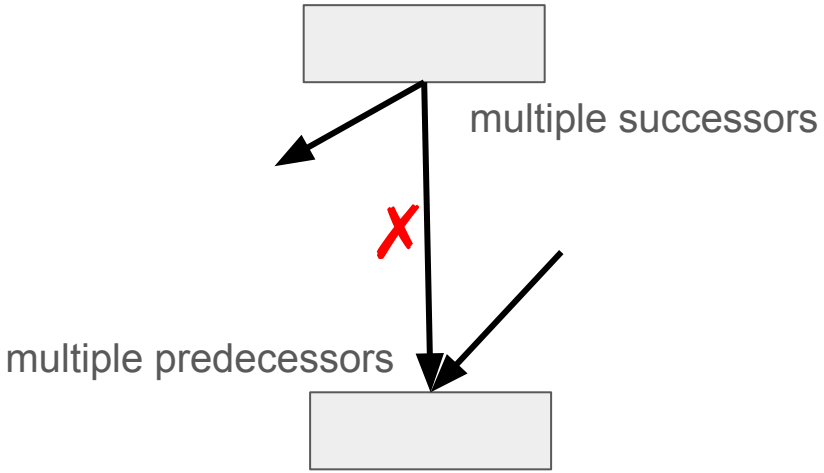
- To avoid redundant moves, split *critical edges* by inserting an empty block between.
  - This block is the proper place for Φ-resolution moves.



Splitting Critical Edges

- To avoid redundant moves, split *critical edges* by inserting an empty block between.
  - This block is the proper place for Φ-resolution moves.

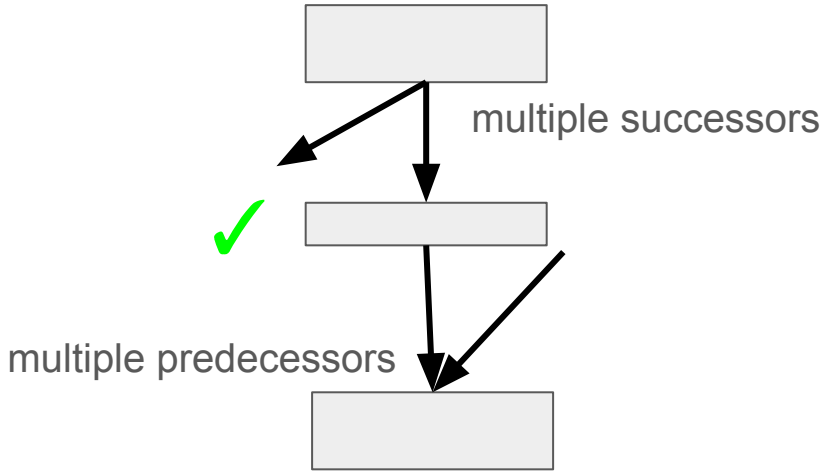
A *critical edge* is any edge that connects a block with multiple successors to a block with multiple predecessors.



Splitting Critical Edges

- To avoid redundant moves, split *critical edges* by inserting an empty block between.
  - This block is the proper place for Φ-resolution moves.

Splitting all critical edges prior to SSA deconstruction is easy.



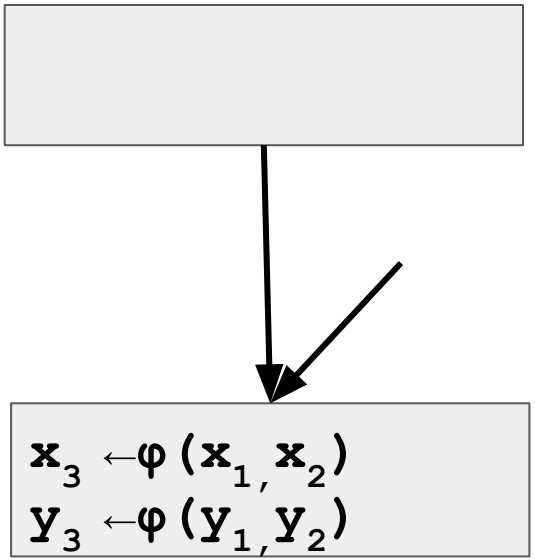
Issue 2: Ordering Moves

- Does the order of Φ-resolution moves matter?
- For CFGs without loops, *no*.
- Let’s convince ourselves.

Issue 2: Ordering Moves

- Does the order of Φ-resolution moves matter?

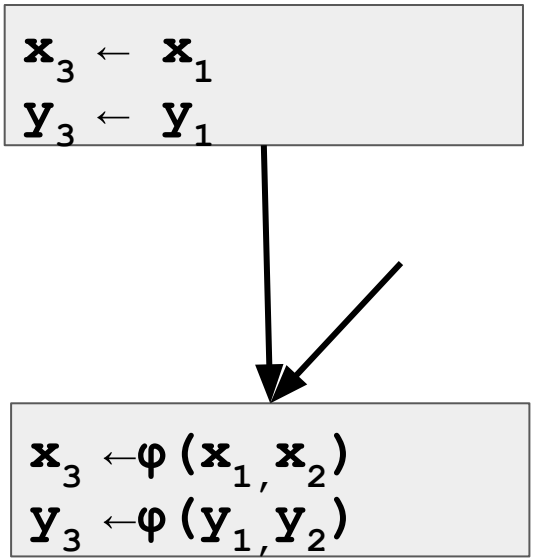
- Consider a join with at least two Φs.



Issue 2: Ordering Moves

- Does the order of Φ-resolution moves matter?

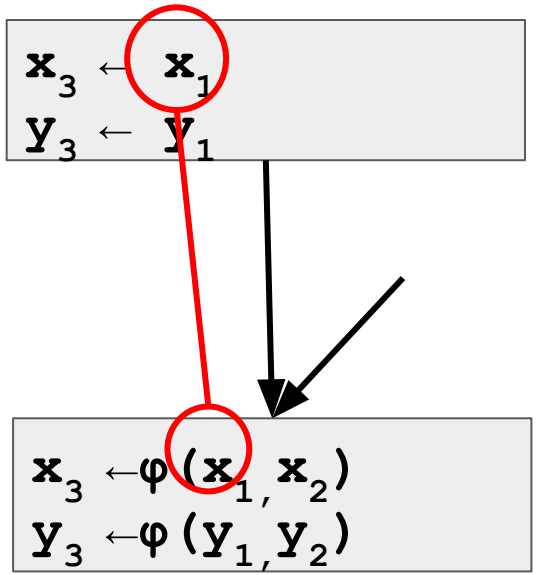
- Consider a join with at least two Φs.
- Moves are inserted into predecessors.



Issue 2: Ordering Moves

- Does the order of Φ-resolution moves matter?

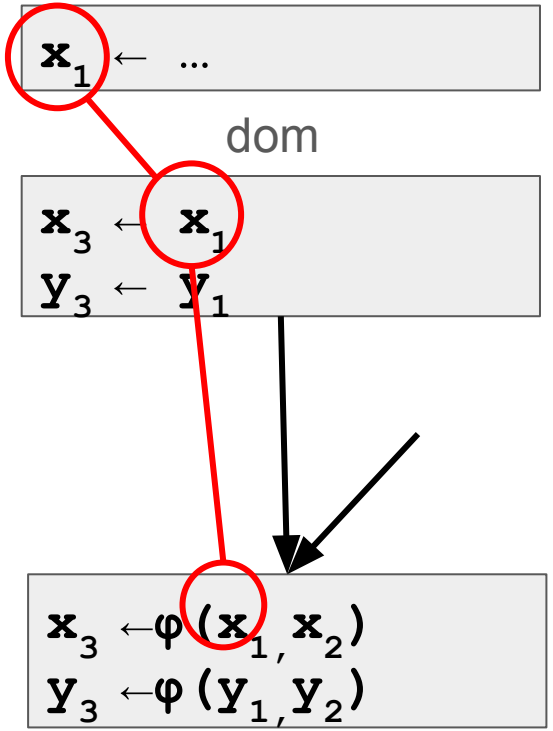
- Consider a join with at least two Φs.
- Moves are inserted into predecessors.



Issue 2: Ordering Moves

- Does the order of Φ-resolution moves matter?

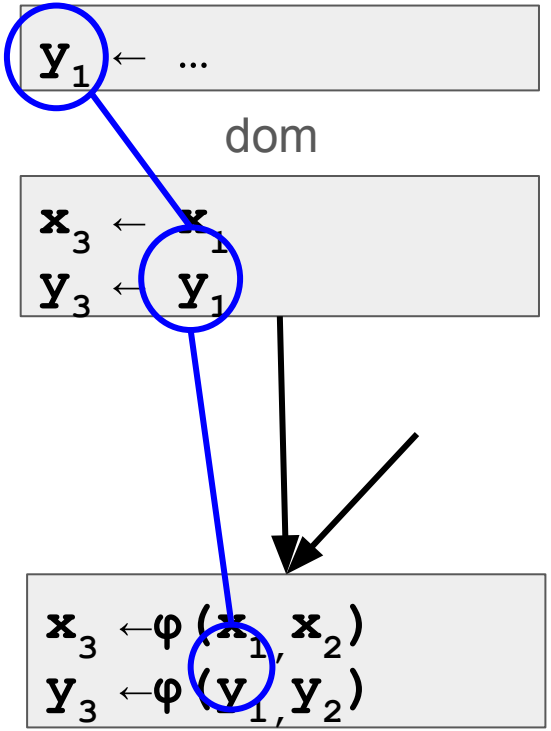
- Consider a join with at least two Φs.
- Moves are inserted into predecessors.
- By SSA invariants, the definition of the RHS of each move dominates the move.



Issue 2: Ordering Moves

- Does the order of Φ-resolution moves matter?

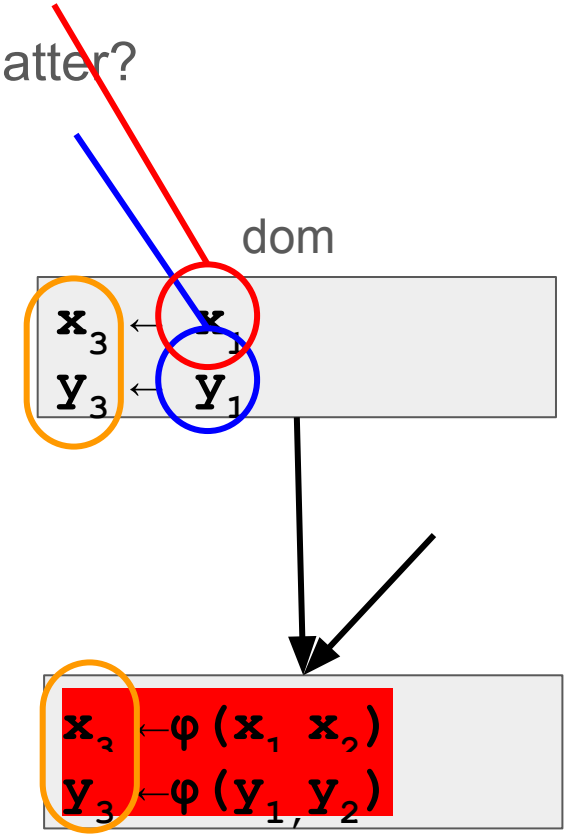
- Consider a join with at least two Φs.
- Moves are inserted into predecessors.
- By SSA invariants, the definition of the RHS of each move dominates the move.



Issue 2: Ordering Moves

- Does the order of Φ-resolution moves matter?

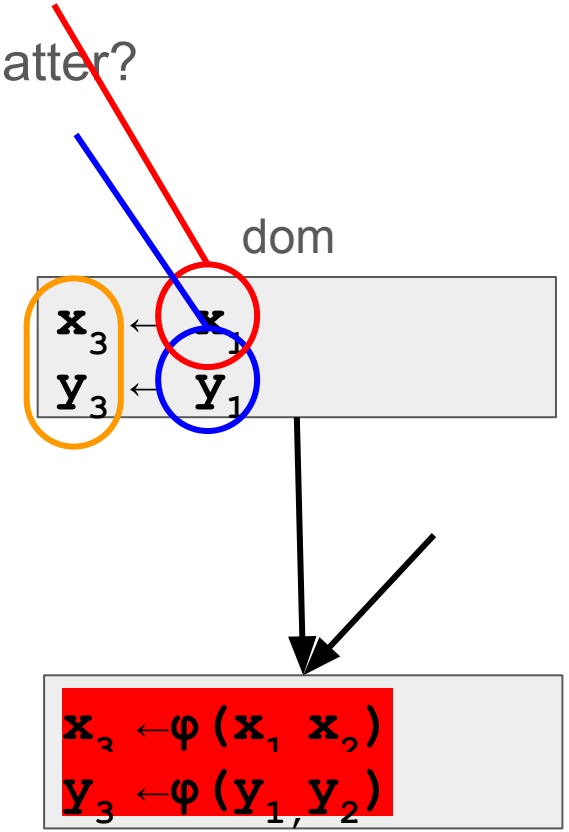
- Consider a join with at least two Φs.
- Moves are inserted into predecessors.
- By SSA invariants, the definition of the RHS of each move dominates the move.
- It cannot be the case that the LHS is live, because previously there was only one definition, below.



Issue 2: Ordering Moves

- Does the order of Φ-resolution moves matter?

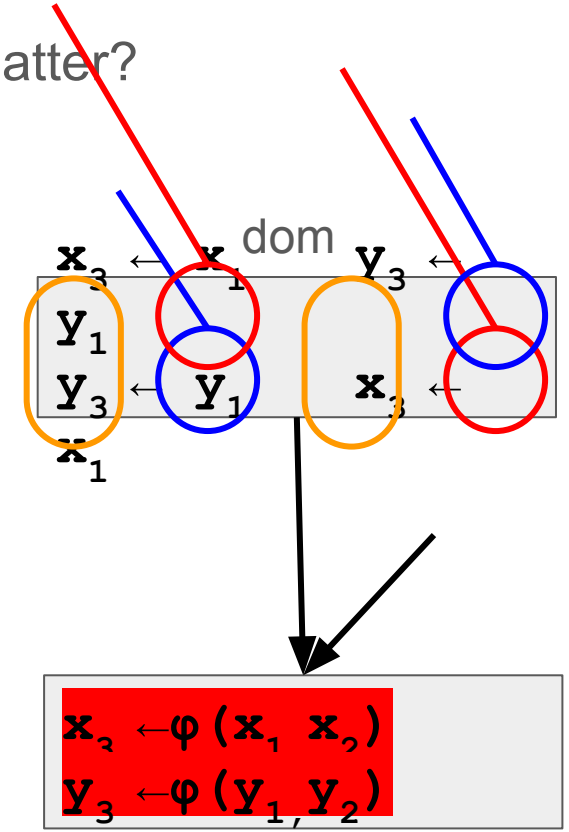
- Consider a join with at least two Φs.
- Moves are inserted into predecessors.
- By SSA invariants, the definition of the RHS of each move dominates the move.
- It cannot be the case that the LHS is live, because previously there was only one definition, below.
- Therefore we are only assigning to fresh variables, and not overwriting anything.



Issue 2: Ordering Moves

- Does the order of Φ-resolution moves matter?

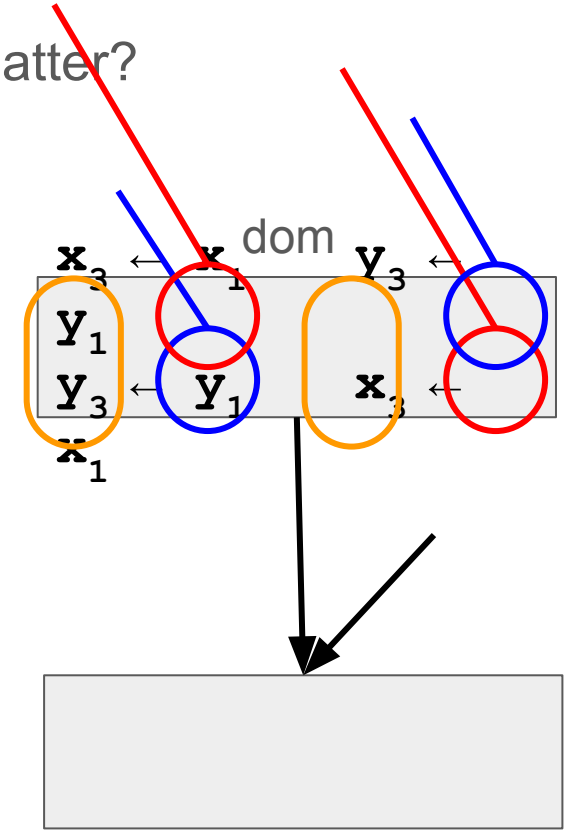
- Consider a join with at least two Φs.
- Moves are inserted into predecessors.
- By SSA invariants, the definition of the RHS of each move dominates the move.
- It cannot be the case that the LHS is live, because previously there was only one definition, below.
- Therefore we are only assigning to fresh variables, and not overwriting anything.
- Therefore any order is fine.



Issue 2: Ordering Moves

- Does the order of Φ-resolution moves matter?

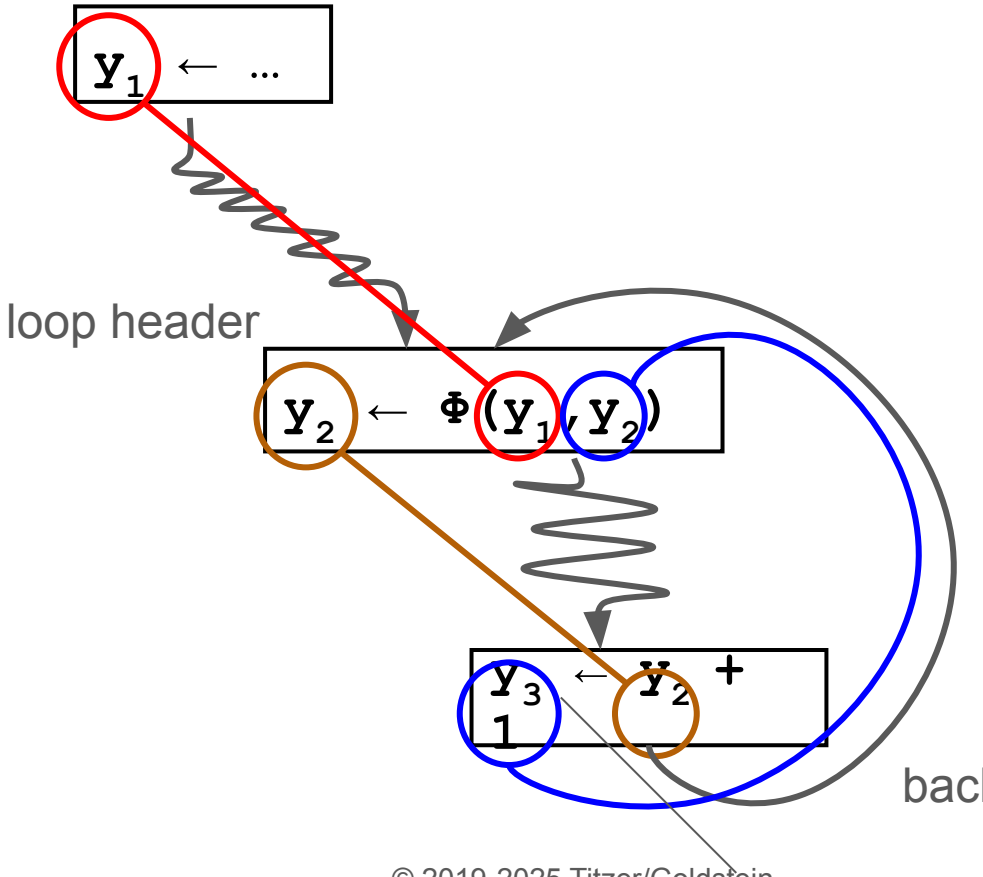
- Consider a join with at least two Φs.
- Moves are inserted into predecessors.
- By SSA invariants, the definition of the RHS of each move dominates the move.
- It cannot be the case that the LHS is live, because previously there was only one definition, below.
- Therefore we are only assigning to fresh variables, and not overwriting anything.
- Therefore any order is fine.



Issue 2: Ordering Moves

- Does the order of Φ-resolution moves matter?
- For CFGs without loops, *no*.
- But what about loops?

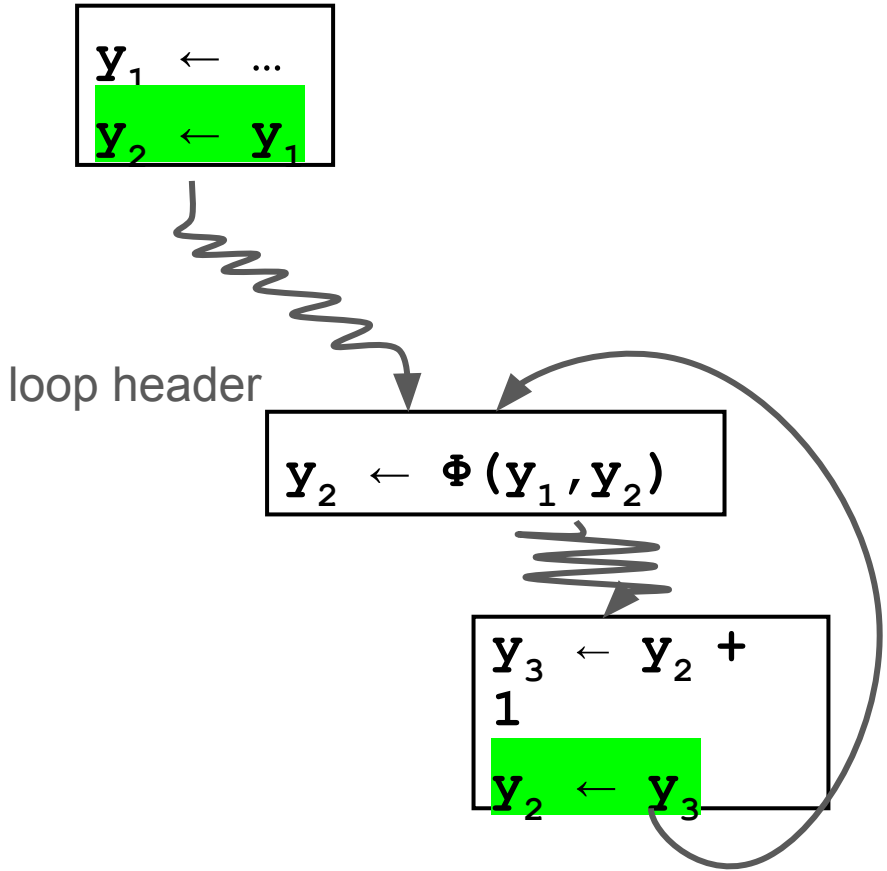
Issue 2: Ordering Moves



Φs at loop headers relate the dataflow on a loop backedge with the control flow.

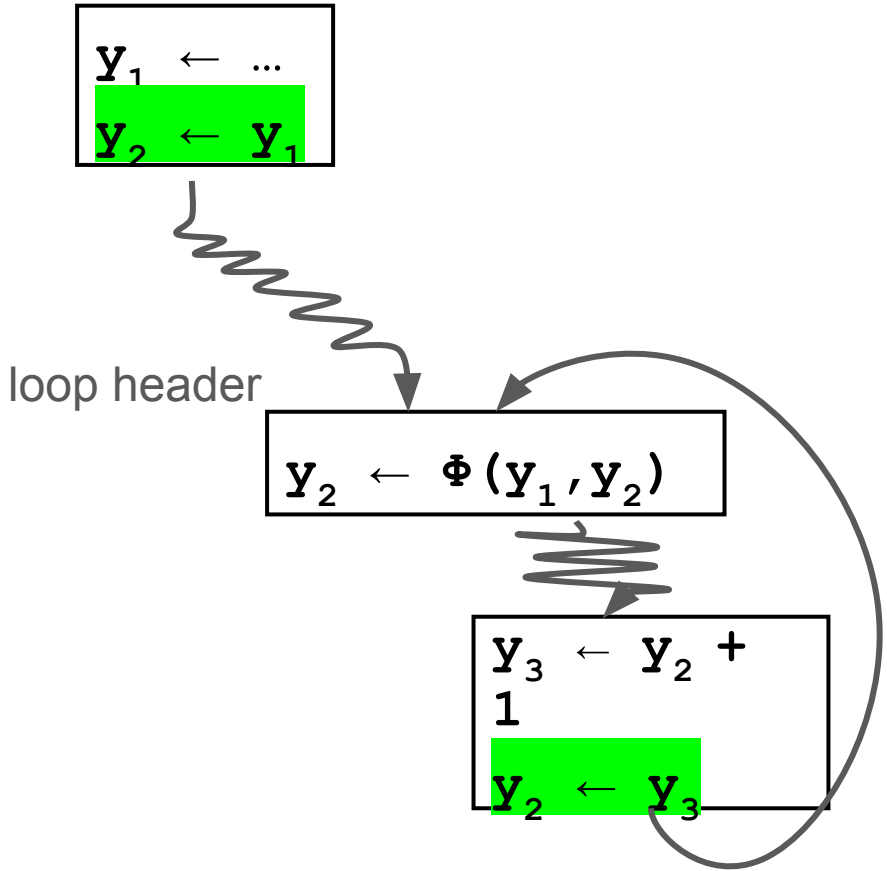
A loop Φ can be defined in terms of itself.

Issue 2: Ordering Moves



Like any other join, we insert Φ-resolution moves at predecessors.

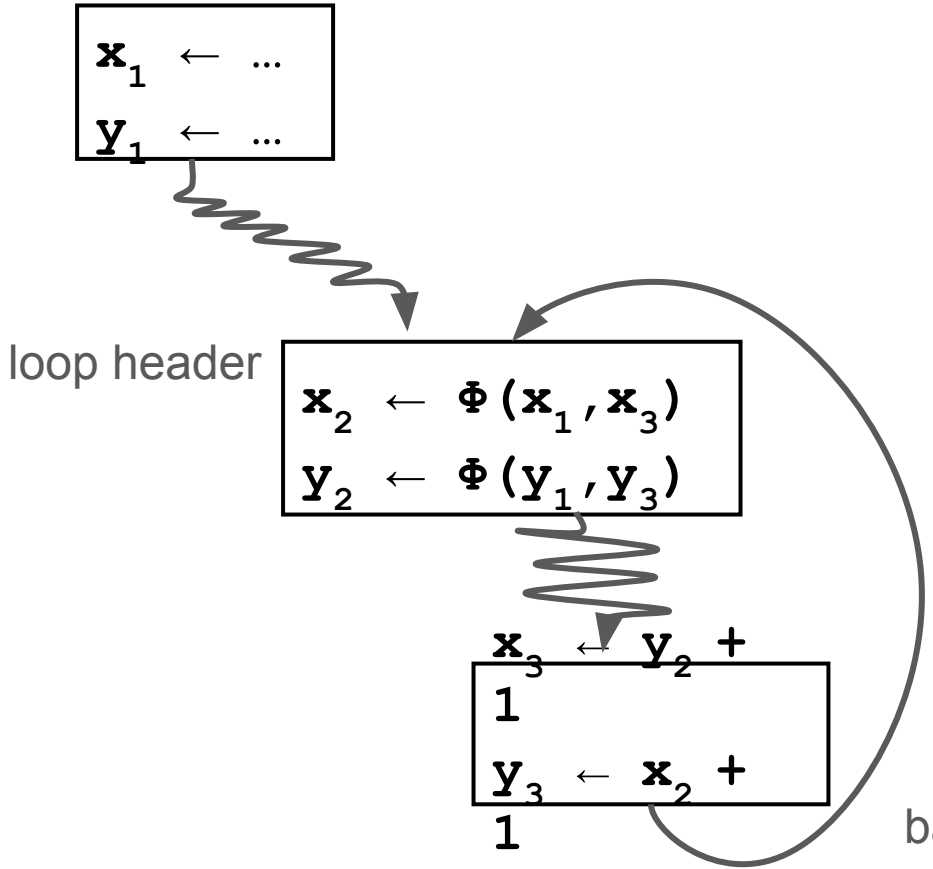
Issue 2: Ordering Moves



Like any other join, we insert Φ-resolution moves at predecessors.

With only one Φ, there is no problem yet.

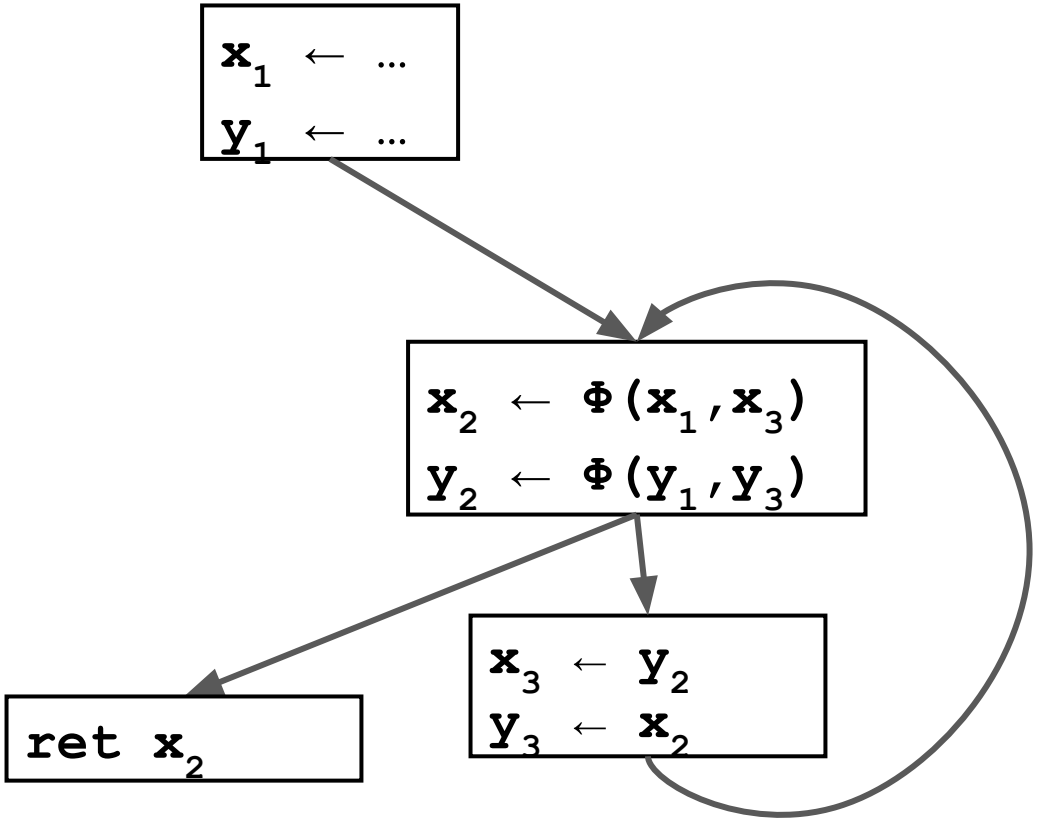
Issue 2: Ordering Moves



Like any join, a loop header can have multiple Φs.

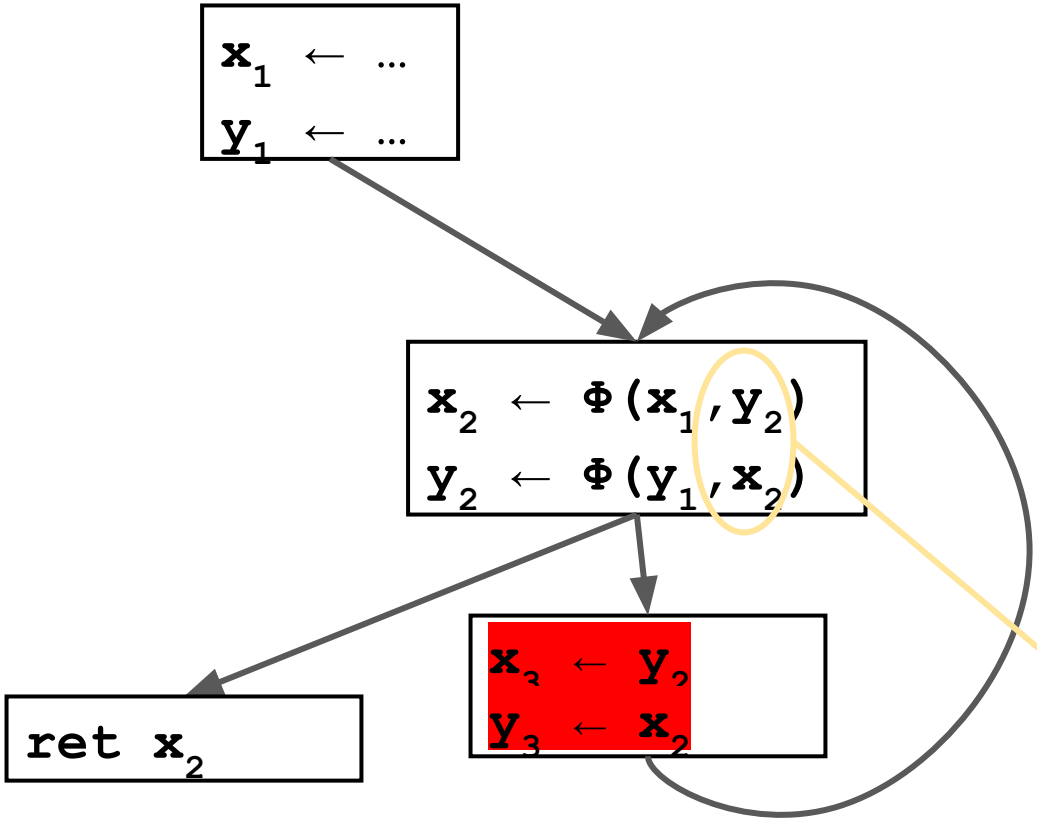
Because Φs can use inductively defined versions of themselves, they can be recursive or even ***mutually recursive***.

Issue 2: Ordering Moves



A simple example: swap of variables in a loop.

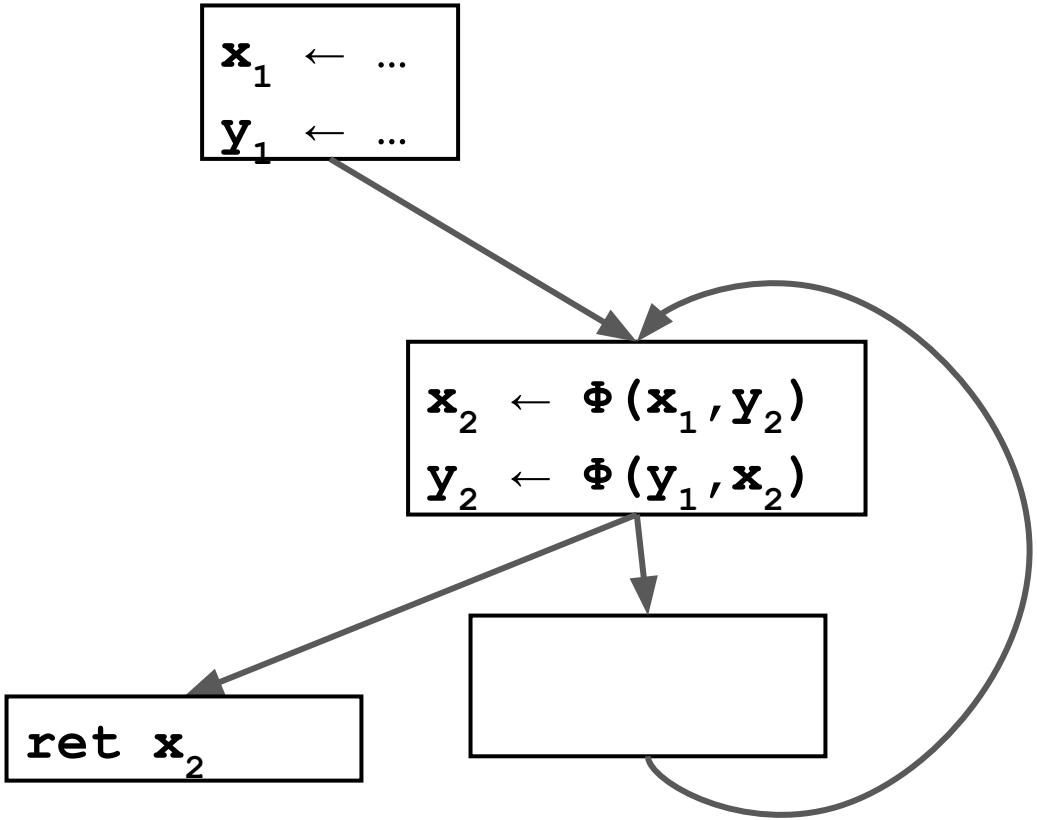
Issue 2: Ordering Moves



After optimizations such as copy propagation, the Φs can be mutually recursive.

Replace
x_3 with y_2
y_3 with x_2

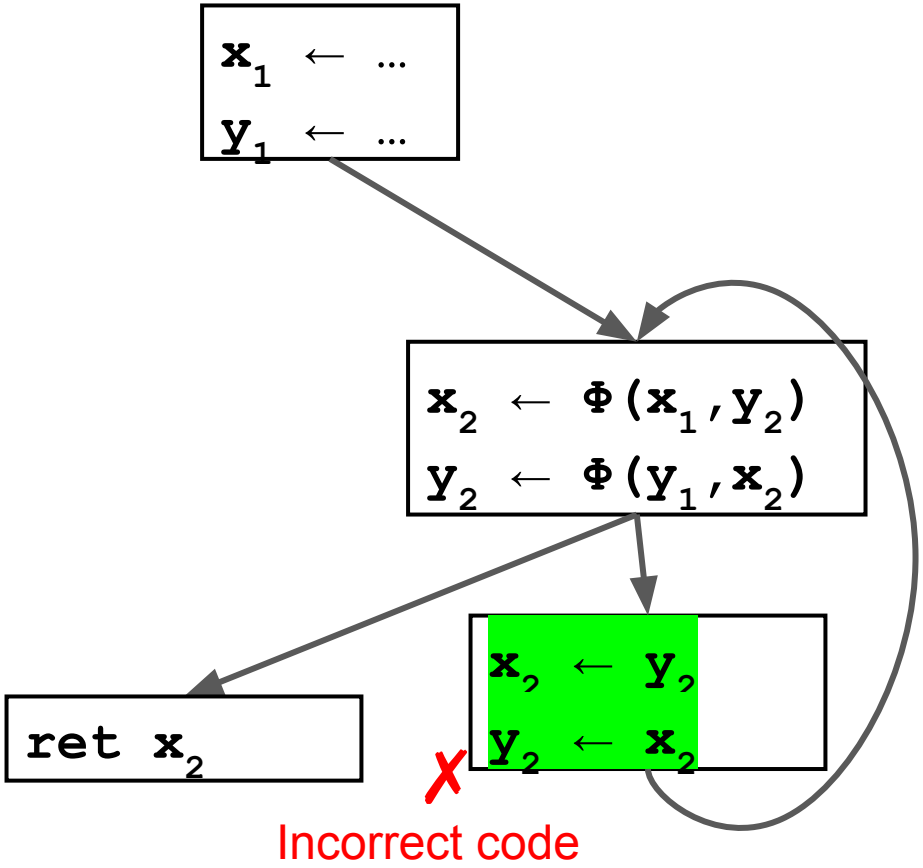
Issue 2: Ordering Moves



After optimizations such as copy propagation, the Φs can be mutually recursive.

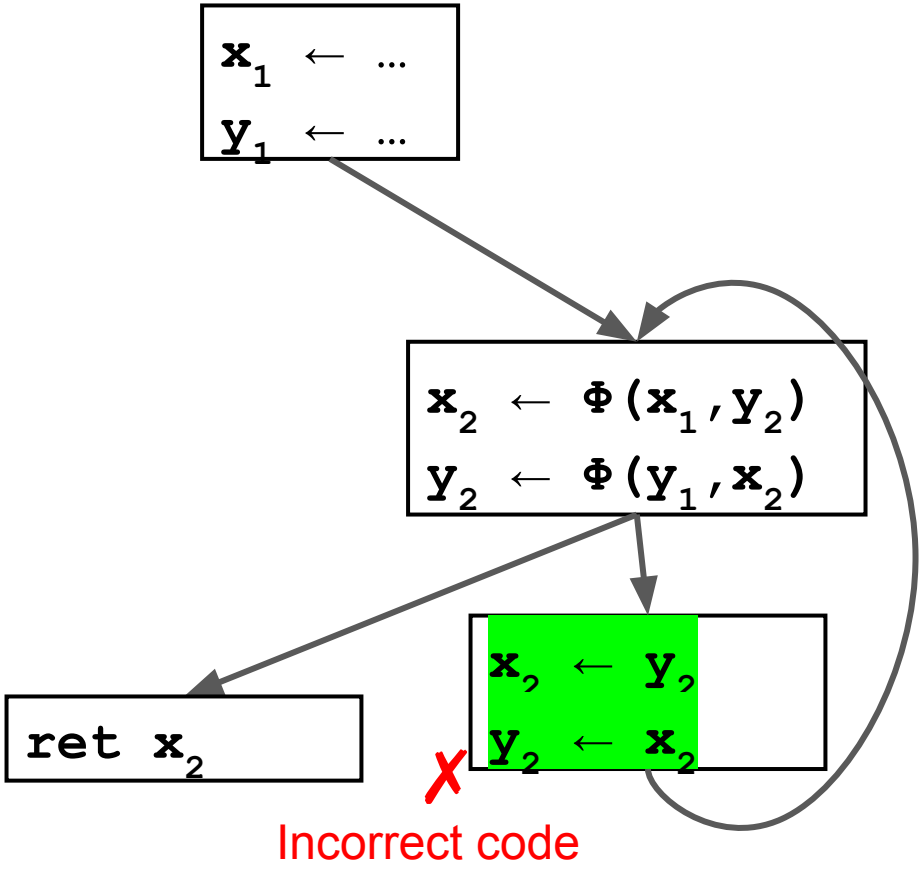
This is totally legal and cool.

Issue 2: Ordering Moves

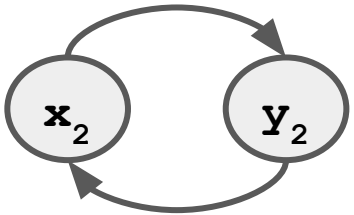


SSA deconstruction using the naïve move insertion will always generate incorrect code, regardless of the order.

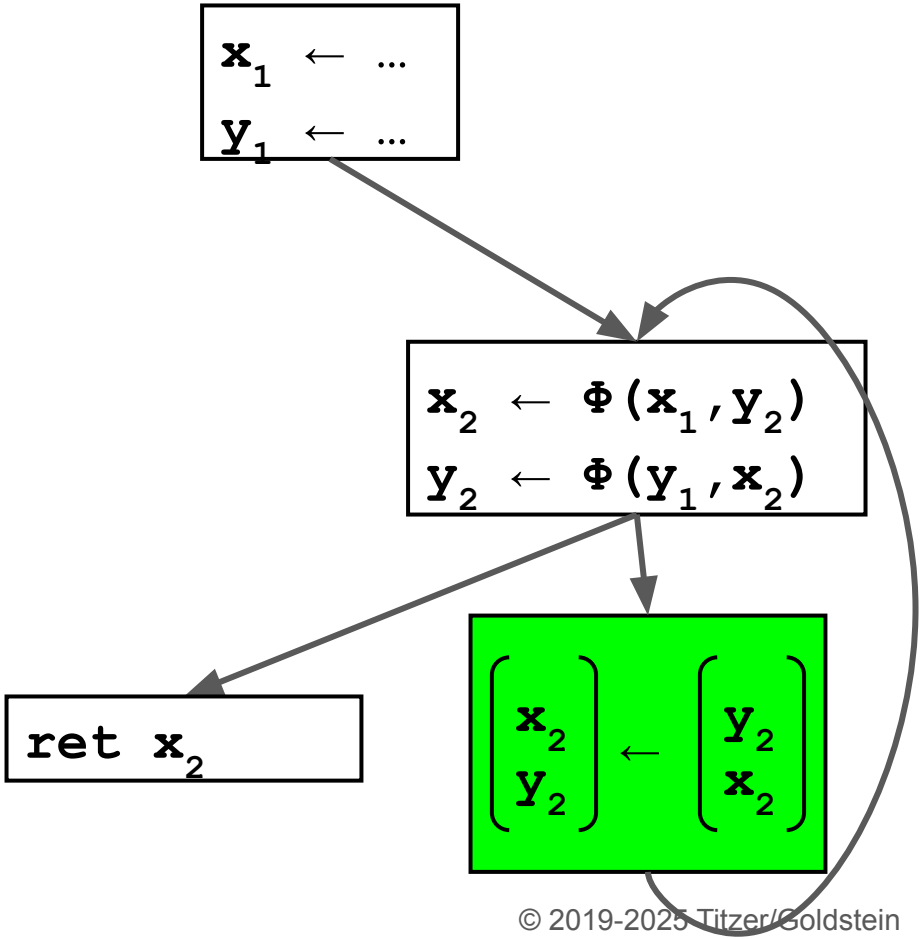
Issue 2: Ordering Moves



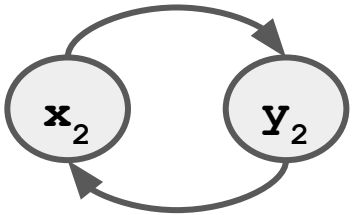
SSA deconstruction using the naïve move insertion will always generate incorrect code, regardless of the order.



Issue 2: Ordering Moves



The reason is that phi resolution moves have ***parallel move*** semantics.



Implementing Parallel Moves

- Φ resolution moves must be done in parallel, without overwriting old versions.
- One simple solution: introduce new temps again.

$$\begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} \leftarrow \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{bmatrix}$$

generates

$$\begin{aligned} t_0 &\leftarrow y_0 \\ t_1 &\leftarrow y_1 \\ t_2 &\leftarrow y_2 \\ t_3 &\leftarrow y_3 \\ x_0 &\leftarrow t_0 \\ x_1 &\leftarrow t_1 \\ x_2 &\leftarrow t_2 \\ x_3 &\leftarrow t_3 \end{aligned}$$

Works every time.

Generates ***a lot*** of temporaries, but maybe the register allocator / copy propagation can clean them up?

Implementing Parallel Moves

- Φ resolution moves must be done in parallel, without overwriting old versions.
- Better solution: order moves more intelligently.

$$\begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix} \leftarrow \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{pmatrix}$$

Implementing Parallel Moves

- Φ resolution moves must be done in parallel, without overwriting old versions.
- Better solution: order moves more intelligently.

$$\begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix} \leftarrow \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{pmatrix}$$

Notice that because parallel moves originate from SSA deconstruction, variables on the LHS appear only once on the LHS.

$$x_0 \neq x_1 \neq x_2 \neq x_3$$

Implementing Parallel Moves

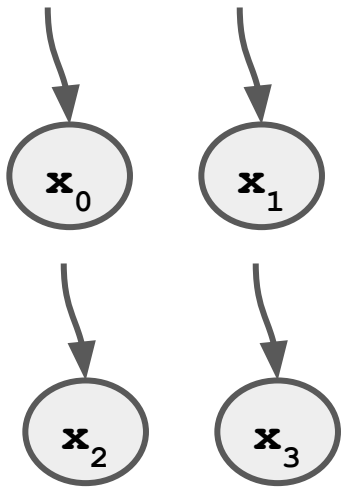
- Φ resolution moves must be done in parallel, without overwriting old versions.
- Better solution: order moves more intelligently ***using LTG***.

$$\begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} \leftarrow \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{bmatrix}$$

We can build a graph where each node in the parallel moves gets a node, and directed edges represent moves.

$$x_0 \neq x_1 \neq x_2 \neq x_3$$

Location Transfer Graph



Implementing Parallel Moves

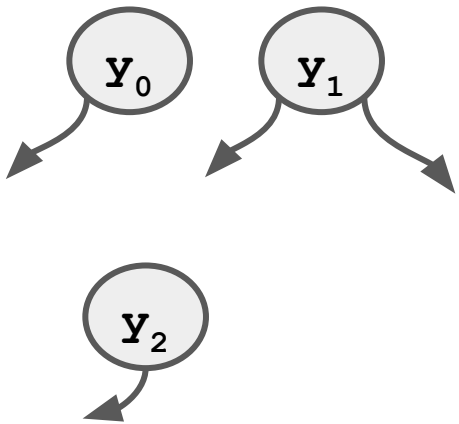
- Φ resolution moves must be done in parallel, without overwriting old versions.
- Better solution: order moves more intelligently ***using LTG***.

$$\begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix} \leftarrow \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_1 \end{pmatrix}$$

Variables may appear ***multiple times*** on the RHS, and may appear on both LHS and RHS.

$$x_0 \neq x_1 \neq x_2 \neq x_3$$

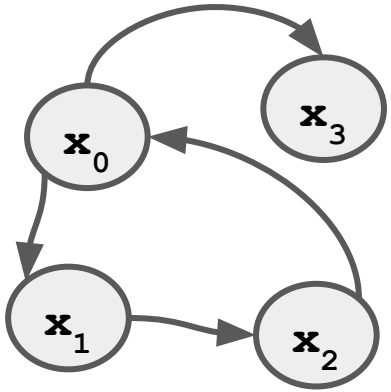
Location Transfer Graph



Location Transfer Graphs

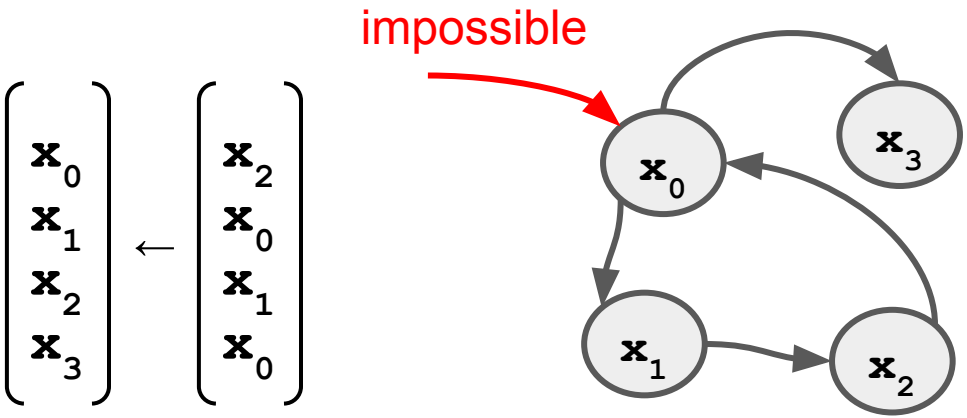
- A location transfer graph represents a set of parallel moves.
- It can be traversed to generate a legal move ordering.
- It's constrained:
  - Every node in the graph has at most one incoming edge.
  - That implies the graph can only have simple cycles.

$$\begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix} \leftarrow \begin{pmatrix} x_2 \\ x_0 \\ x_1 \\ x_0 \end{pmatrix}$$



Location Transfer Graphs

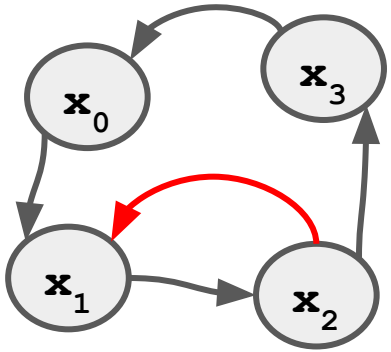
- A location transfer graph represents a set of parallel moves.
- It can be traversed to generate a legal move ordering.
- It's constrained:
  - Every node in the graph has at most one incoming edge.
  - That implies the graph can only have simple cycles.



Location Transfer Graphs

- A location transfer graph represents a set of parallel moves.
- It can be traversed to generate a legal move ordering.
- It's constrained:
  - Every node in the graph has at most one incoming edge.
  - That implies the graph can only have simple cycles.

$$\begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix} \leftarrow \begin{pmatrix} x_3 \\ x_0 \\ x_1 \\ x_2 \end{pmatrix}$$

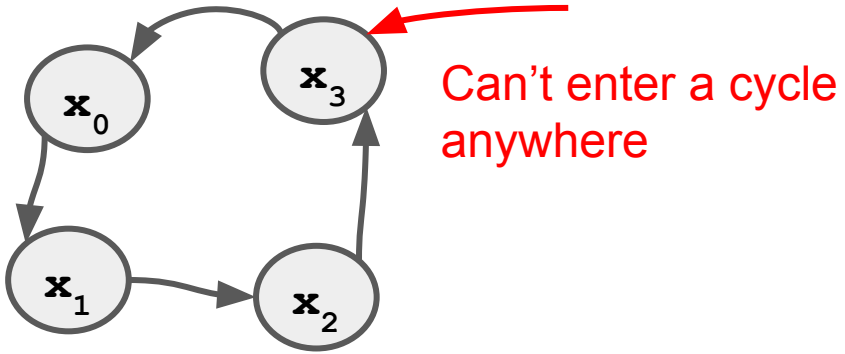


impossible to have nested cycles

Location Transfer Graphs

- A location transfer graph represents a set of parallel moves.
- It can be traversed to generate a legal move ordering.
- It's constrained:
  - Every node in the graph has at most one incoming edge.
  - That implies the graph can only have simple cycles.

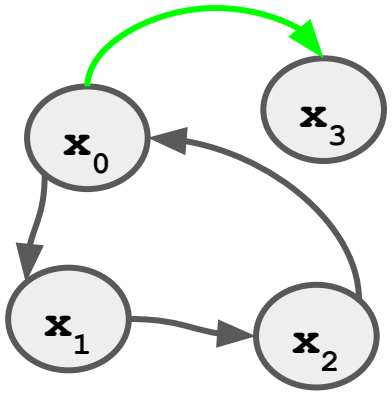
$$\begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix} \leftarrow \begin{pmatrix} x_3 \\ x_0 \\ x_1 \\ x_2 \end{pmatrix}$$



Location Transfer Graphs

- A location transfer graph represents a set of parallel moves.
- It can be traversed to generate a legal move ordering.
- It's constrained:
  - Every node in the graph has at most one incoming edge.
  - That implies the graph can only have simple cycles.

$$\begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} \leftarrow \begin{bmatrix} x_2 \\ x_0 \\ x_1 \\ x_0 \end{bmatrix}$$

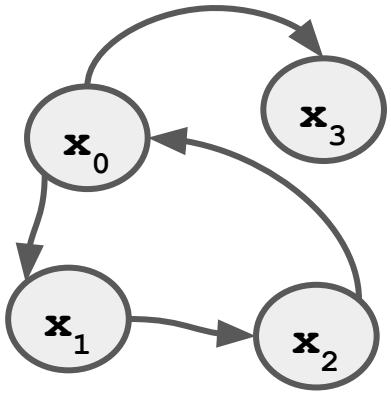


Can leave a cycle though!

Location Transfer Graphs

- A location transfer graph represents a set of parallel moves.
- It can be traversed to generate a legal move ordering.
- It's constrained:
  - Every node in the graph has at most one incoming edge.
  - That implies the graph can only have simple cycles.

$$\begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix} \leftarrow \begin{pmatrix} x_2 \\ x_0 \\ x_1 \\ x_0 \end{pmatrix}$$



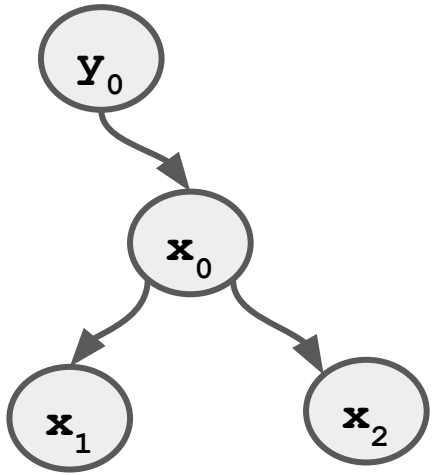
Also known as "windmill graphs" [1]

[1] See "SSA Elimination after Register Allocation" by Pereira and Palsberg, 2009.

Location Transfer Graphs

- Post-order depth-first search (DFS) on this graph yields a legal move ordering.
- Must break cycles with a temporary (or use swaps [1])

$$\begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} \leftarrow \begin{bmatrix} y_0 \\ x_0 \\ x_0 \end{bmatrix}$$

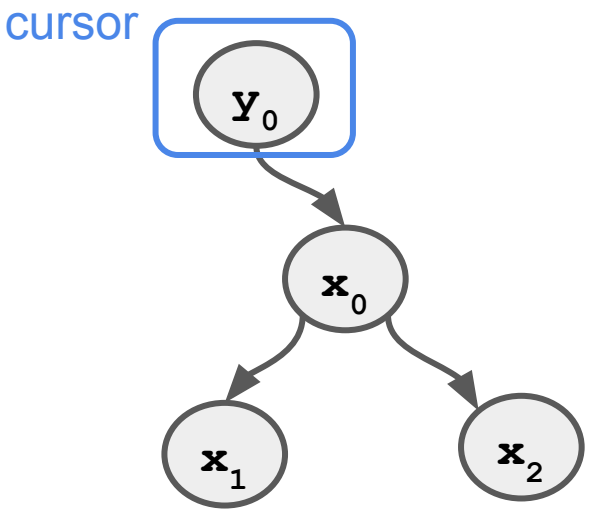


[1] See “SSA Elimination after Register Allocation” by Pereira and Palsberg, 2009.

Location Transfer Graphs

- Post-order depth-first search (DFS) on this graph yields a legal move ordering.
- Must break cycles with a temporary (or use swaps [1])

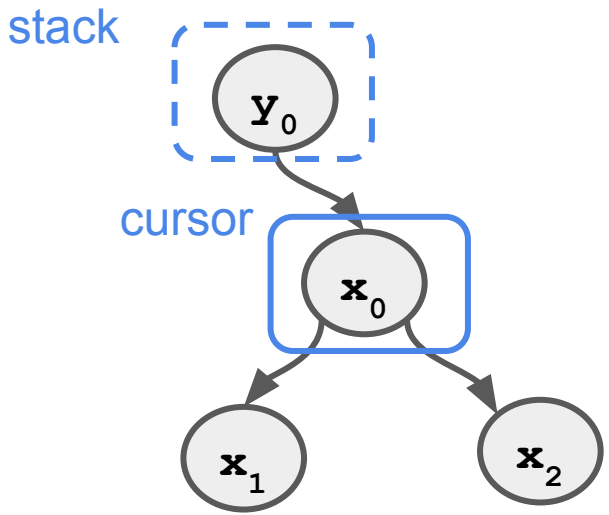
$$\begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} \leftarrow \begin{pmatrix} y_0 \\ x_0 \\ x_0 \end{pmatrix}$$



Location Transfer Graphs

- Post-order depth-first search (DFS) on this graph yields a legal move ordering.
- Must break cycles with a temporary (or use swaps [1])

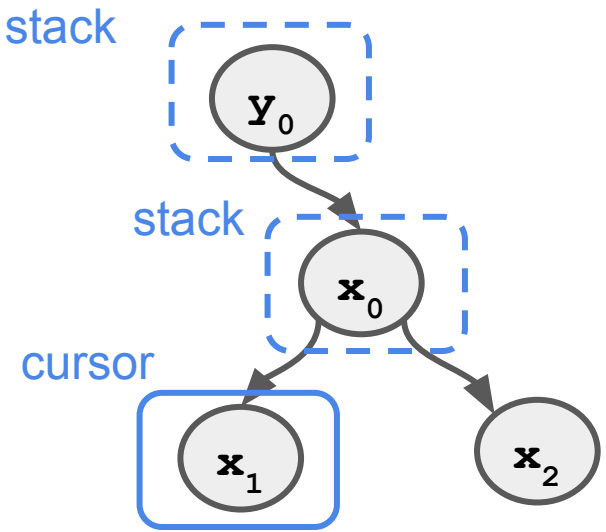
$$\begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} \leftarrow \begin{bmatrix} y_0 \\ x_0 \\ x_0 \end{bmatrix}$$



Location Transfer Graphs

- Post-order depth-first search (DFS) on this graph yields a legal move ordering.
- Must break cycles with a temporary (or use swaps [1])

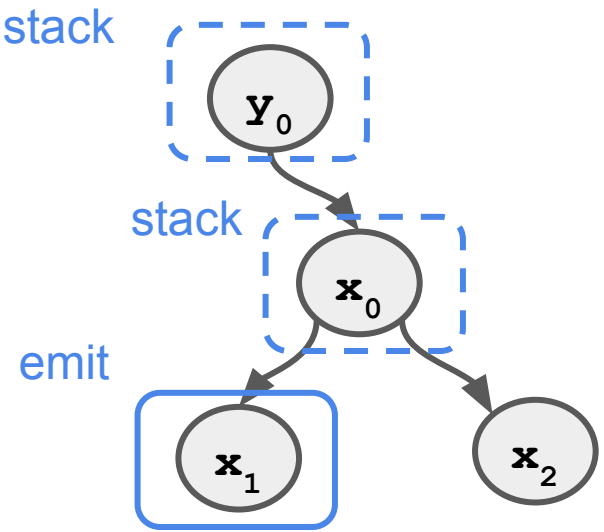
$$\begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} \leftarrow \begin{bmatrix} y_0 \\ x_0 \\ x_0 \end{bmatrix}$$



Location Transfer Graphs

- Post-order depth-first search (DFS) on this graph yields a legal move ordering.
- Must break cycles with a temporary (or use swaps [1])

$$\begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} \leftarrow \begin{pmatrix} y_0 \\ x_0 \\ x_0 \end{pmatrix}$$

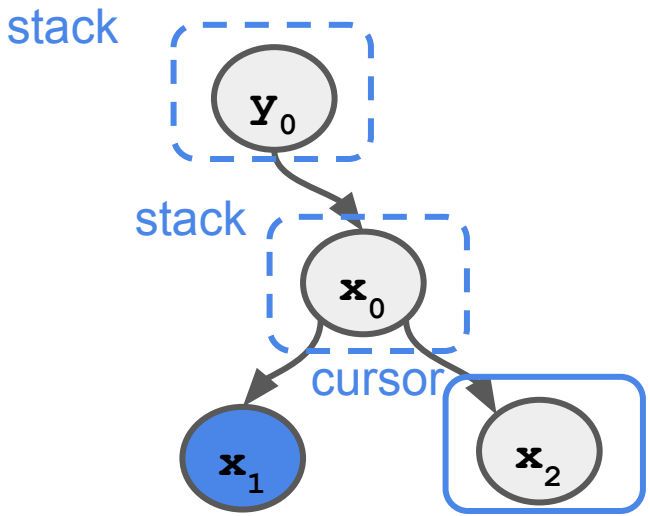


$x_1 \leftarrow x_0$

Location Transfer Graphs

- Post-order depth-first search (DFS) on this graph yields a legal move ordering.
- Must break cycles with a temporary (or use swaps [1])

$$\begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} \leftarrow \begin{pmatrix} y_0 \\ x_0 \\ x_0 \end{pmatrix}$$

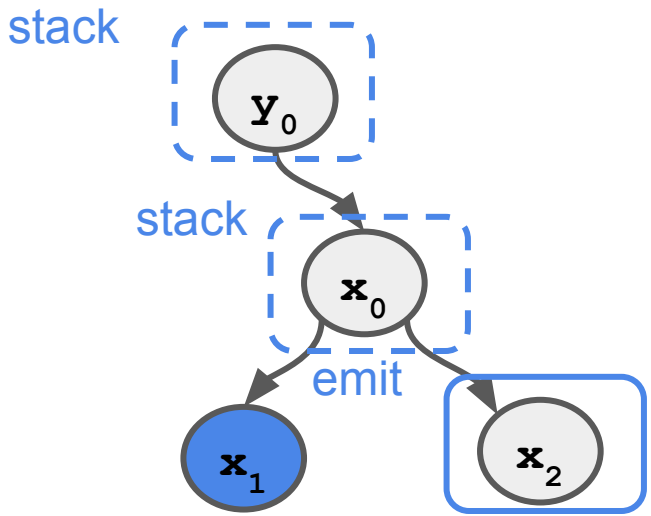


$x_1 \leftarrow x_0$

Location Transfer Graphs

- Post-order depth-first search (DFS) on this graph yields a legal move ordering.
- Must break cycles with a temporary (or use swaps [1])

$$\begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} \leftarrow \begin{pmatrix} y_0 \\ x_0 \\ x_0 \end{pmatrix}$$



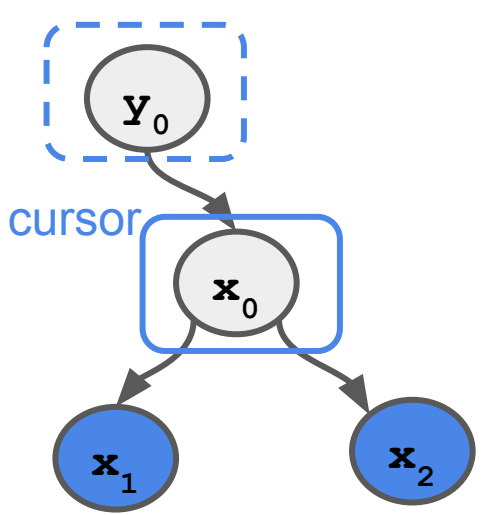
$x_1 \leftarrow x_0$

$x_2 \leftarrow x_0$

Location Transfer Graphs

- Post-order depth-first search (DFS) on this graph yields a legal move ordering.
- Must break cycles with a temporary (or use swaps [1])

$$\begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} \leftarrow \begin{pmatrix} y_0 \\ x_0 \\ x_0 \end{pmatrix}$$



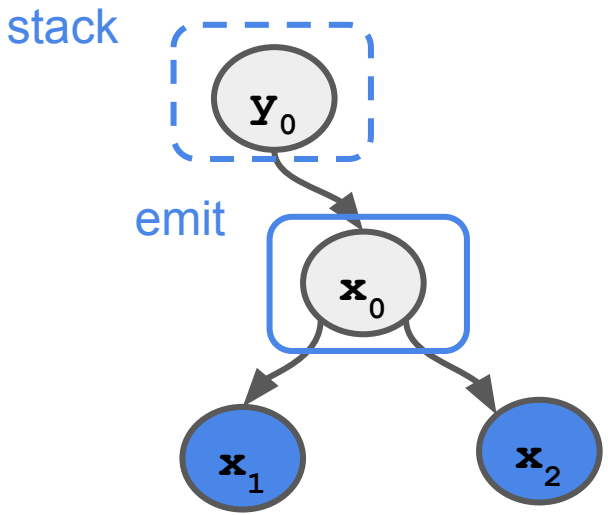
$x_1 \leftarrow x_0$

$x_2 \leftarrow x_0$

Location Transfer Graphs

- Post-order depth-first search (DFS) on this graph yields a legal move ordering.
- Must break cycles with a temporary (or use swaps [1])

$$\begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} \leftarrow \begin{pmatrix} y_0 \\ x_0 \\ x_0 \end{pmatrix}$$

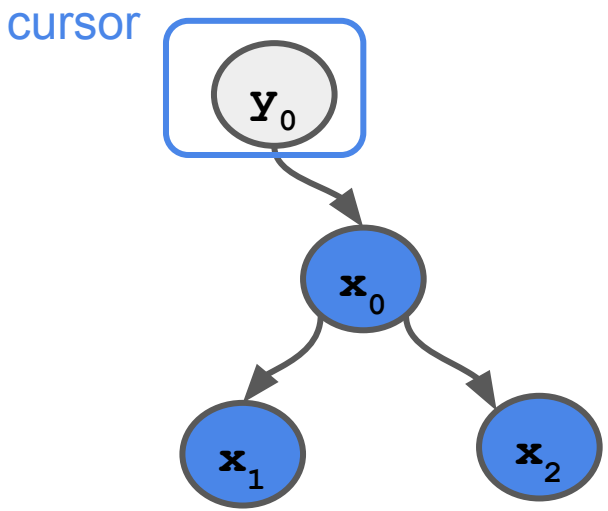


$$x_1 \leftarrow x_0$$
$$x_2 \leftarrow x_0$$
$$x_0 \leftarrow y_0$$

Location Transfer Graphs

- Post-order depth-first search (DFS) on this graph yields a legal move ordering.
- Must break cycles with a temporary (or use swaps [1])

$$\begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} \leftarrow \begin{pmatrix} y_0 \\ x_0 \\ x_0 \end{pmatrix}$$

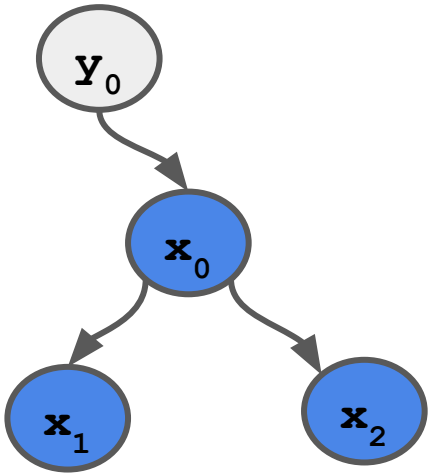


$$x_1 \leftarrow x_0$$
$$x_2 \leftarrow x_0$$
$$x_0 \leftarrow y_0$$

Location Transfer Graphs

- Post-order depth-first search (DFS) on this graph yields a legal move ordering.
- Must break cycles with a temporary (or use swaps [1])

$$\begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} \leftarrow \begin{bmatrix} y_0 \\ x_0 \\ x_0 \end{bmatrix}$$

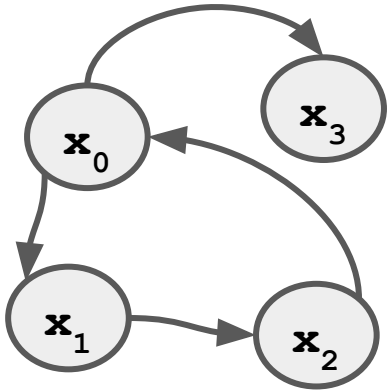


$$x_1 \leftarrow x_0$$
$$x_2 \leftarrow x_0$$
$$x_0 \leftarrow y_0$$

Location Transfer Graphs

- Post-order depth-first search (DFS) on this graph yields a legal move ordering.
- Must break cycles with a temporary (or use swaps [1])

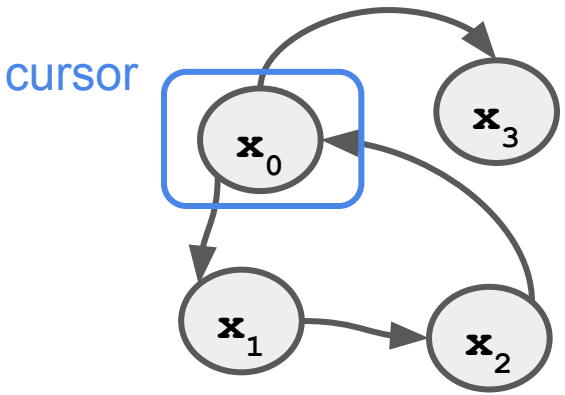
$$\begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix} \leftarrow \begin{pmatrix} x_2 \\ x_0 \\ x_1 \\ x_0 \end{pmatrix}$$



Location Transfer Graphs

- Post-order depth-first search (DFS) on this graph yields a legal move ordering.
- Must break cycles with a temporary (or use swaps [1])

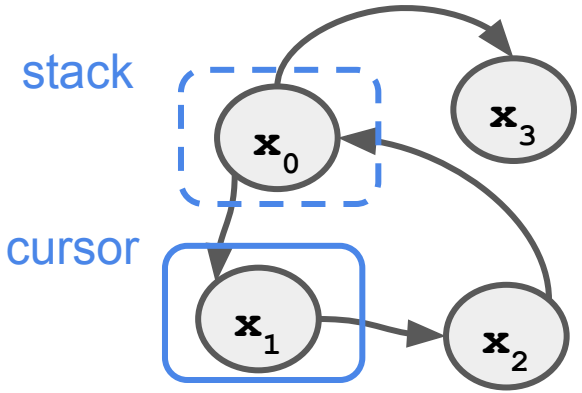
$$\begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} \leftarrow \begin{bmatrix} x_2 \\ x_0 \\ x_1 \\ x_0 \end{bmatrix}$$



Location Transfer Graphs

- Post-order depth-first search (DFS) on this graph yields a legal move ordering.
- Must break cycles with a temporary (or use swaps [1])

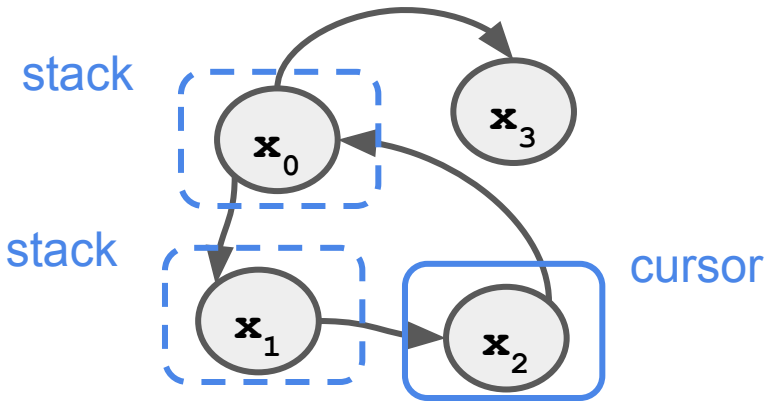
$$\begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix} \leftarrow \begin{pmatrix} x_2 \\ x_0 \\ x_1 \\ x_0 \end{pmatrix}$$



Location Transfer Graphs

- Post-order depth-first search (DFS) on this graph yields a legal move ordering.
- Must break cycles with a temporary (or use swaps [1])

$$\begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix} \leftarrow \begin{pmatrix} x_2 \\ x_0 \\ x_1 \\ x_0 \end{pmatrix}$$

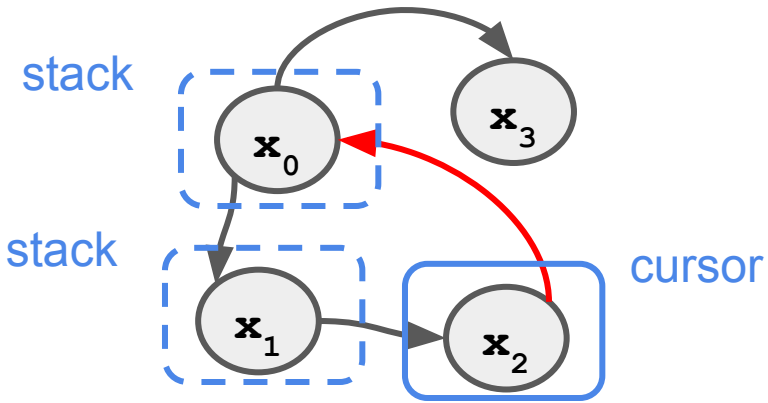


Location Transfer Graphs

- Post-order depth-first search (DFS) on this graph yields a legal move ordering.
- Must break cycles with a temporary (or use swaps [1])

Break cycle by saving in a temp

$$\begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix} \leftarrow \begin{pmatrix} x_2 \\ x_0 \\ x_1 \\ x_0 \end{pmatrix}$$

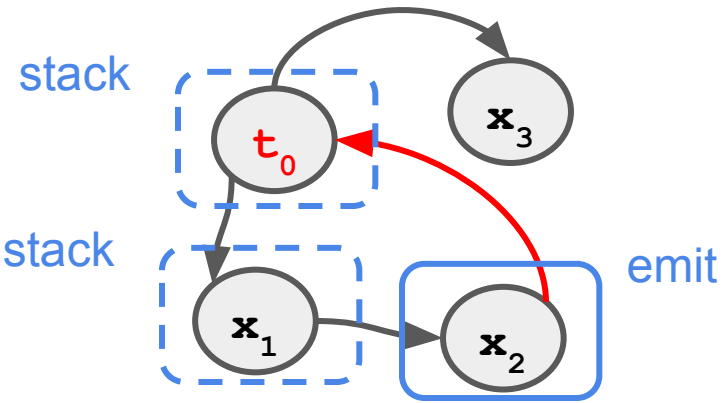


Location Transfer Graphs

- Post-order depth-first search (DFS) on this graph yields a legal move ordering.
- Must break cycles with a temporary (or use swaps [1])

Break cycle by saving in a temp, ***overwriting***, and using the temp instead.

$$\begin{bmatrix} x_2 \\ x_3 \end{bmatrix} \quad \begin{bmatrix} x_1 \\ x_0 \end{bmatrix}$$

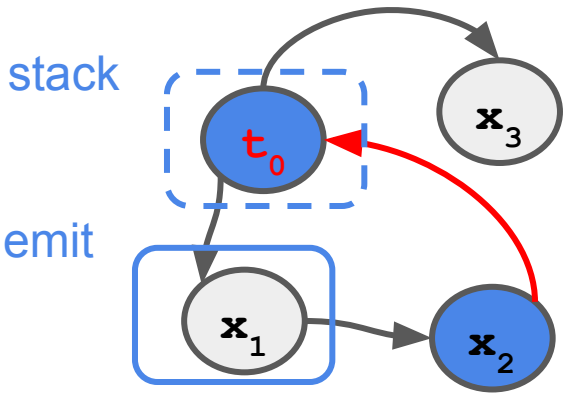


$$t_0 \leftarrow x_0$$
$$x_0 \leftarrow x_2$$

Location Transfer Graphs

- Post-order depth-first search (DFS) on this graph yields a legal move ordering.
- Must break cycles with a temporary (or use swaps [1])

$$\begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix} \leftarrow \begin{pmatrix} x_2 \\ x_0 \\ x_1 \\ x_0 \end{pmatrix}$$

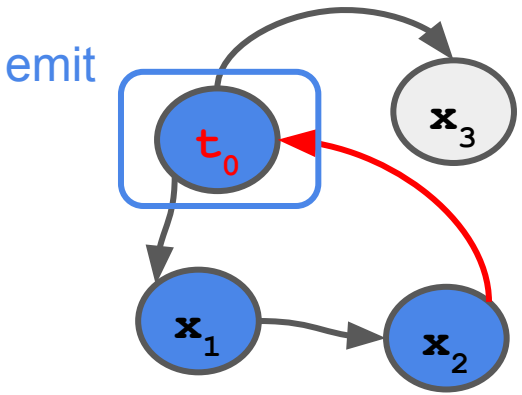


$$t_0 \leftarrow x_0$$
$$x_0 \leftarrow x_2$$
$$x_2 \leftarrow x_1$$

Location Transfer Graphs

- Post-order depth-first search (DFS) on this graph yields a legal move ordering.
- Must break cycles with a temporary (or use swaps [1])

$$\begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} \leftarrow \begin{bmatrix} x_2 \\ x_0 \\ x_1 \\ x_0 \end{bmatrix}$$

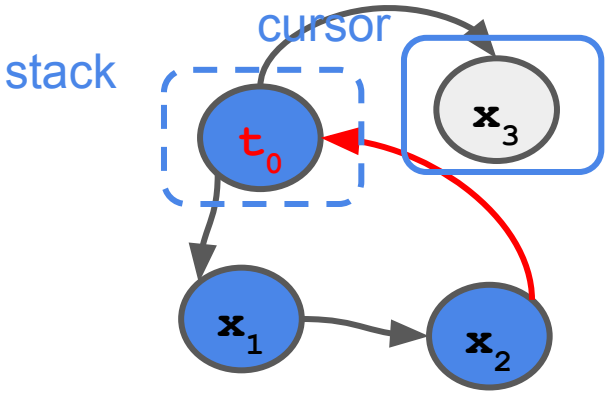


$$t_0 \leftarrow x_0$$
$$x_0 \leftarrow x_2$$
$$x_2 \leftarrow x_1$$
$$x_1 \leftarrow t_0$$

Location Transfer Graphs

- Post-order depth-first search (DFS) on this graph yields a legal move ordering.
- Must break cycles with a temporary (or use swaps [1])

$$\begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix} \leftarrow \begin{pmatrix} x_2 \\ x_0 \\ x_1 \\ x_0 \end{pmatrix}$$

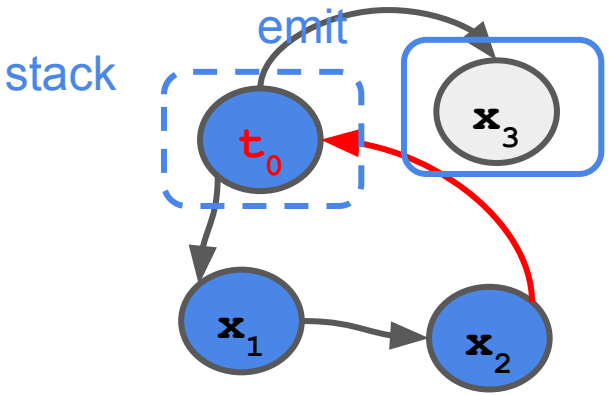


$$t_0 \leftarrow x_0$$
$$x_0 \leftarrow x_2$$
$$x_2 \leftarrow x_1$$
$$x_1 \leftarrow t_0$$

Location Transfer Graphs

- Post-order depth-first search (DFS) on this graph yields a legal move ordering.
- Must break cycles with a temporary (or use swaps [1])

$$\begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix} \leftarrow \begin{pmatrix} x_2 \\ x_0 \\ x_1 \\ x_0 \end{pmatrix}$$

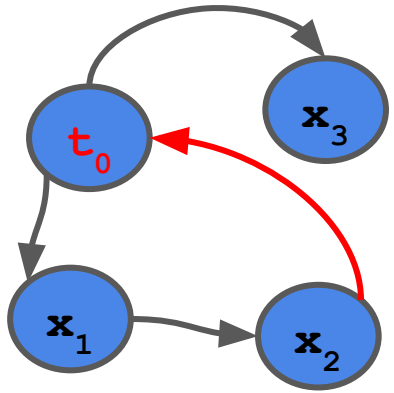


$$\begin{aligned} t_0 &\leftarrow x_0 \\ x_0 &\leftarrow x_2 \\ x_2 &\leftarrow x_1 \\ x_1 &\leftarrow t_0 \\ x_3 &\leftarrow t_0 \end{aligned}$$

Location Transfer Graphs

- Post-order depth-first search (DFS) on this graph yields a legal move ordering.
- Must break cycles with a temporary (or use swaps [1])

$$\begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} \leftarrow \begin{bmatrix} x_2 \\ x_0 \\ x_1 \\ x_0 \end{bmatrix}$$



$$\begin{aligned} t_0 &\leftarrow x_0 \\ x_0 &\leftarrow x_2 \\ x_2 &\leftarrow x_1 \\ x_1 &\leftarrow t_0 \\ x_3 &\leftarrow t_0 \end{aligned}$$

SSA Elimination After Register Allocation

- Breaking cycles in LTG is easy with a temporary
- Before regalloc, temporaries are plentiful
- If deconstructing after register allocation, all regs may be used
  - How do we know? May need to save something from regalloc
- It's possible to implement LTG with swaps and/or sequences of xor's
- Another option is to use a temporary slot
- See paper for details:
  - "SSA Elimination after Register Allocation" by Pereira and Palsberg, 2009.

Syntax-directed SSA Construction

- Is it possible to skip dominators/dominance frontier?
- “Single-Pass Generation of Static Single-Assignment Form for Structured Languages” by Brandis and Mössenbock (1994).
  - https://dl.acm.org/doi/10.1145/197320.197331
- Also done in my Virgil compiler
  - https://github.com/titzer/virgil