SSA (1 of 2)

15-411/15-611 Compiler Design

Seth Copen Goldstein

January 27, 2026

Today

- Trivial SSA
- φ-functions
- Dominance
- Placement & Renaming

SSA

- Static single assignment is an **intermediate representation (IR)** where every variable has only *one* definition
    - Single **static** definition
    - (Could be in a loop which is executed dynamically many times.)

- φ-functions used at CFG join points
- All definitions dominate uses
- Variable names don't matter; IR implementation is literally nodes in a graph that point to each other

Advantages of SSA

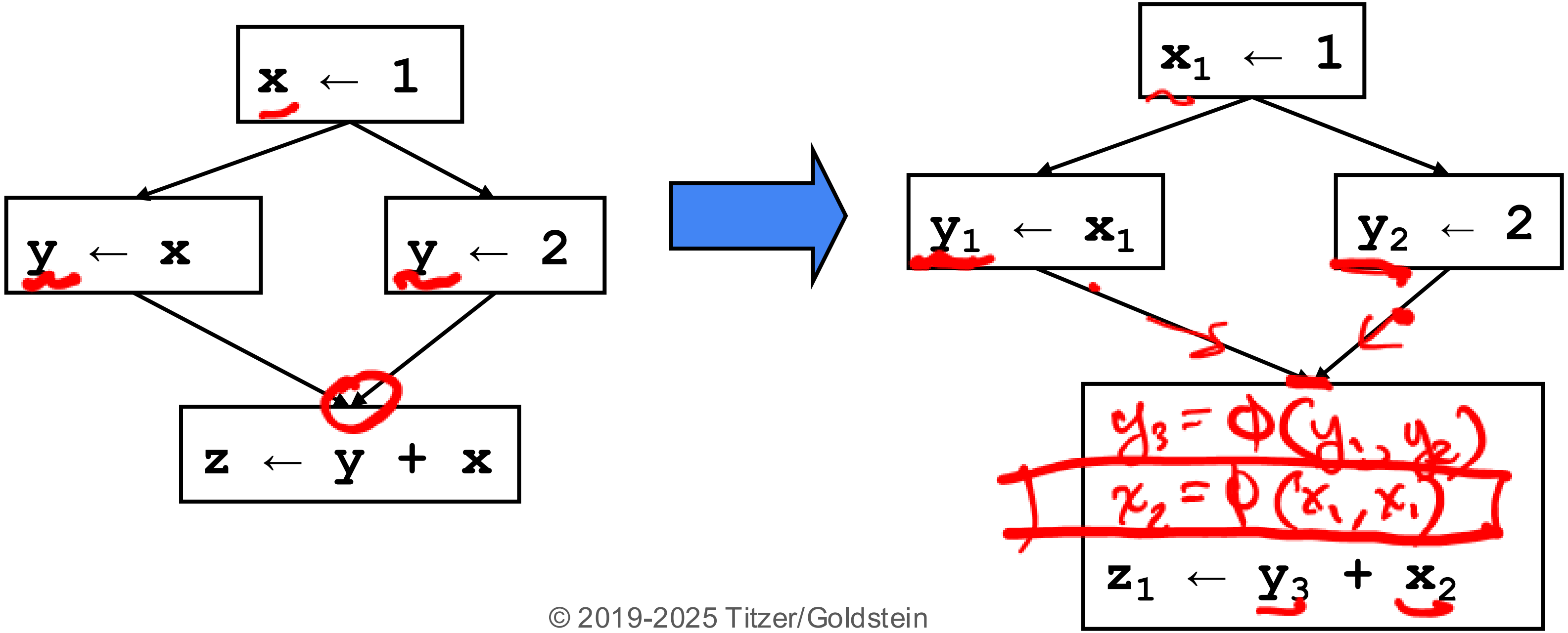
- Makes def-use-chains explicit
- Makes dataflow optimizations more *robust*
  - Easier to get right
  - Multiple optimizations can compose
  - Applies to more places
- Improves register allocation
  - Makes building interference graphs easier
  - Easier register allocation algorithm
  - Decoupling of spill, color, and coalesce
- For most programs reduces space/time requirements
  - Smaller IR, faster optimizations

Implications of single definition

- Never have to worry about a variable being overwritten
  - Before SSA, compilers had to worry about variable names and redefinitions
  - A “node” in SSA IR represents a computation, rather than a storage location
- Improves pattern-matching optimizations
  - Constant propagation (y = 13; x + y ⇝ x + 13)
  - Constant folding (3 + 5 ⇝ 8)
  - Strength reduction (x + 0 ⇝ x)
  - Algebraic simplification (x + y - x ⇝ y)
- Improves reasoning across control flow
- Think of it as a “bulk solution” to many forward dataflow problems

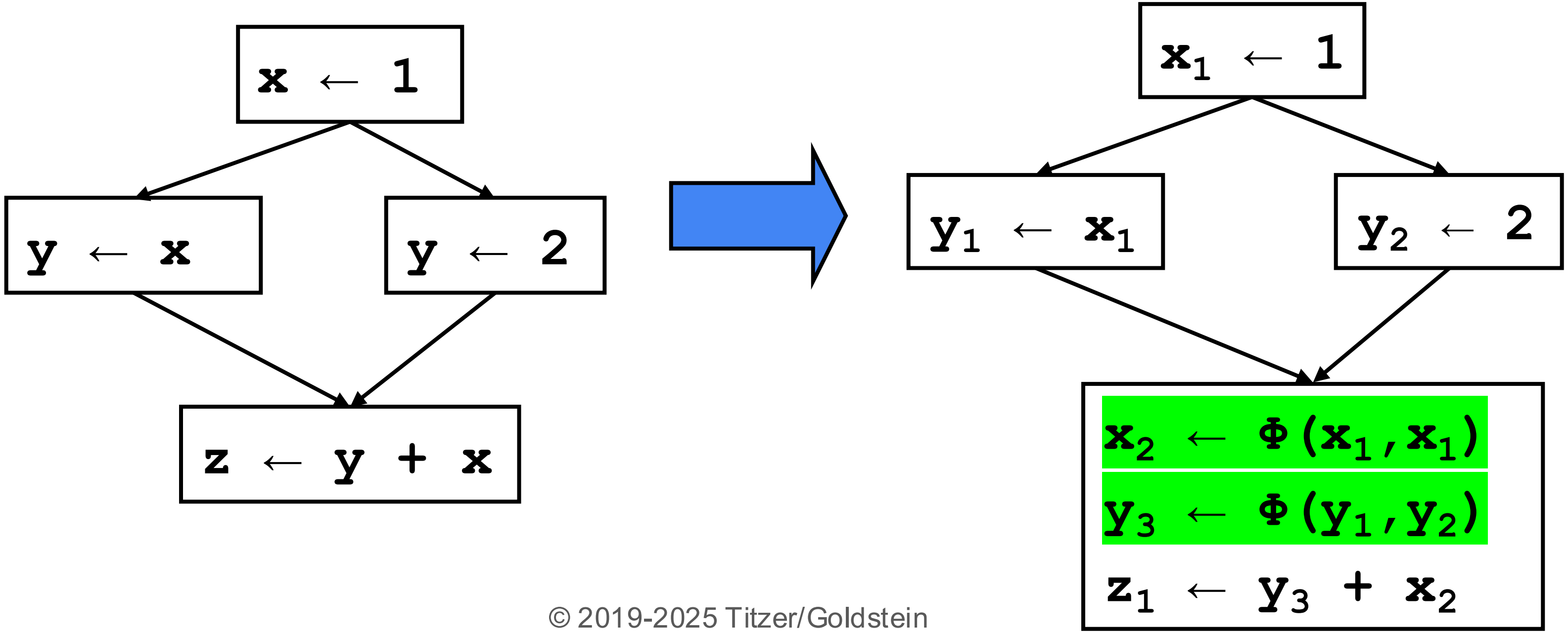
Trivial SSA

- Each assignment generates a fresh variable.
- At each join point insert Φ functions for all live variables.



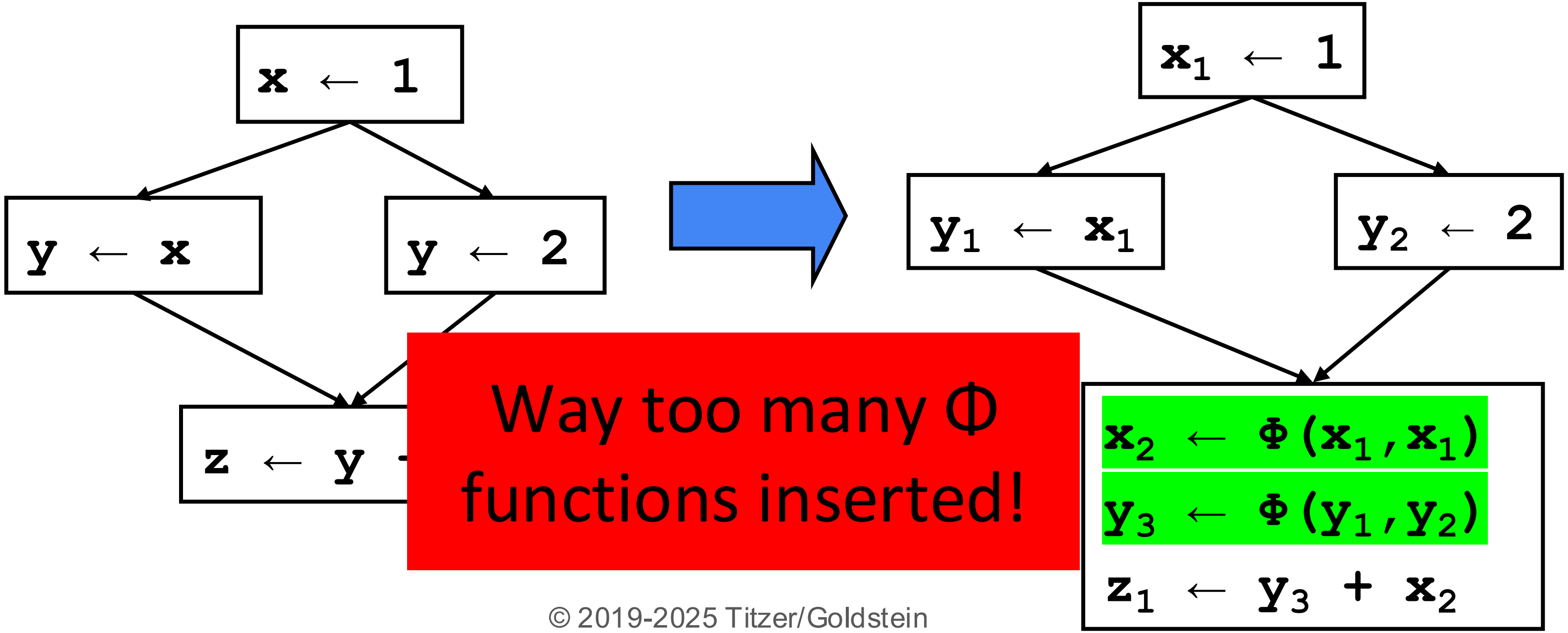
Trivial SSA

- Each assignment generates a fresh variable.
- At each join point insert Φ functions for all live variables.



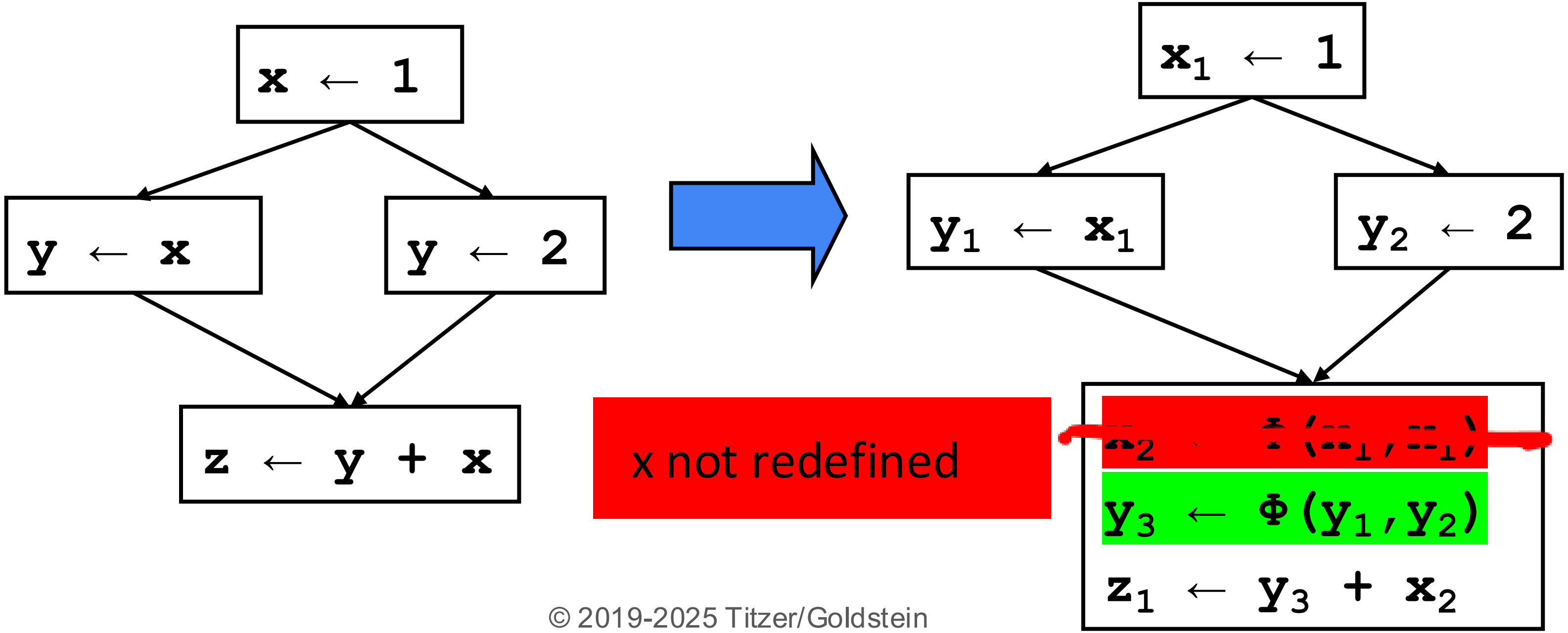
Trivial SSA

- Each assignment generates a fresh variable.
- At each join point insert Φ functions for all live variables.



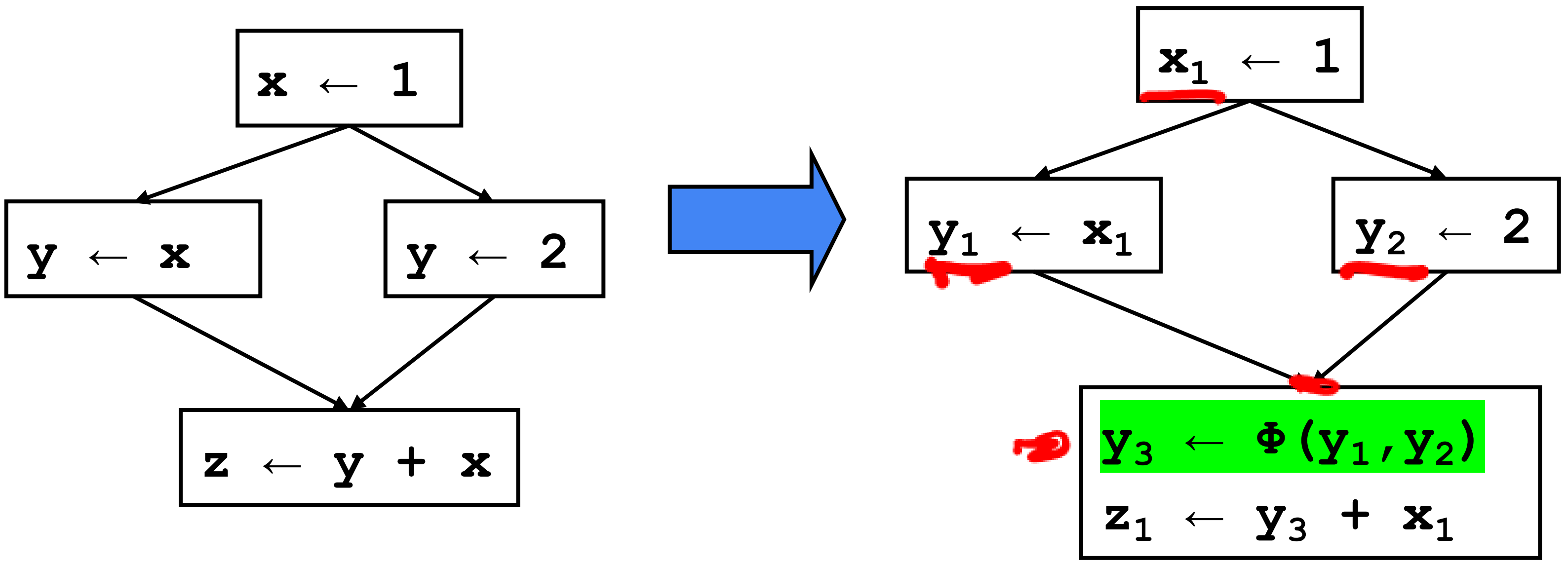
Minimal SSA

- Each assignment generates a fresh variable.
- At each join point insert Φ functions for all variables with multiple outstanding defs.



Minimal SSA

- Each assignment generates a fresh variable.
- At each join point insert Φ functions for all variables with multiple outstanding defs.



Handling cyclic control flow

- Introduce ϕ-functions to handle *joins* in CFG
- Loops have joins too!

```
x ← …
y ← …
while(x < 100){
   x ← x + 1
   y ← y + 1
}
```

```
   x ← …
   y ← …
   if (x >= 100) goto end
loop:
   x ← x + 1
   y ← y + 1
   if (x < 100) goto loop
end:
```

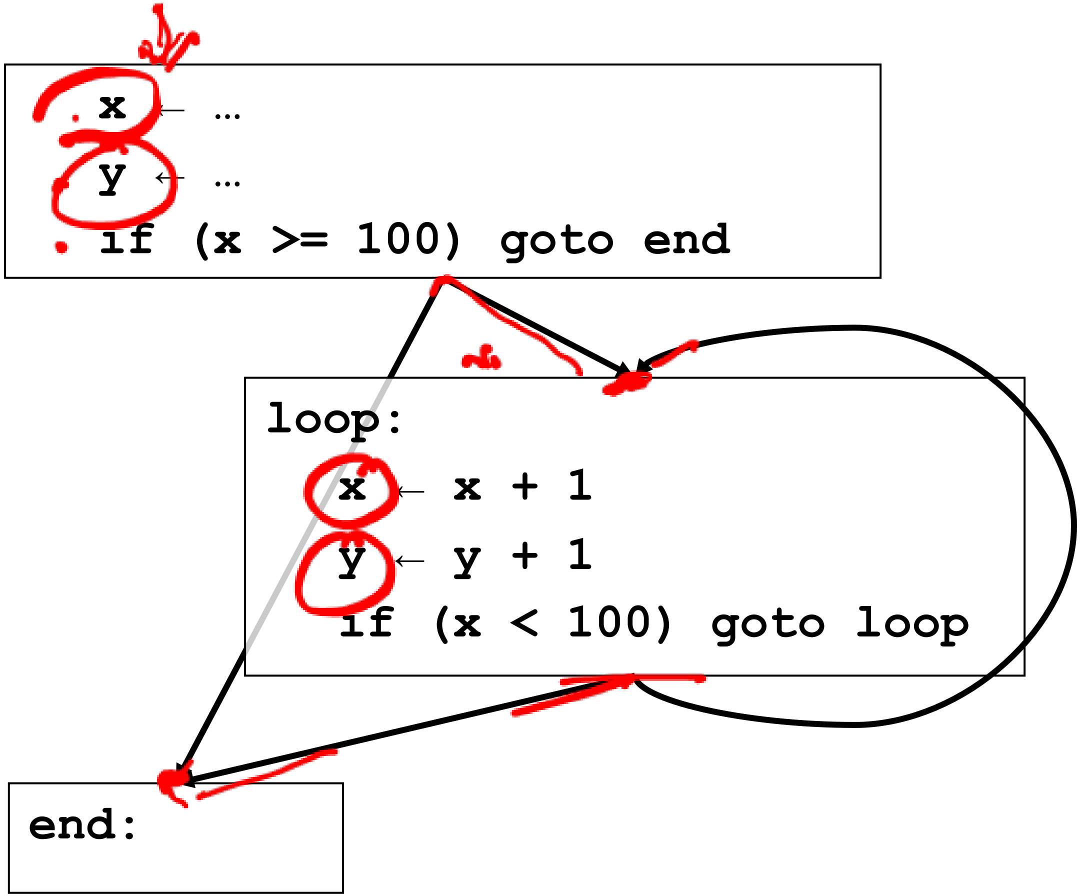
Handling cyclic control flow

- SSA requires single definition for each use
- Introduce φ-functions to handle joins at loop headers too

```
   x ← …
   y ← …
   if (x >= 100) goto end
loop:
   x ← x + 1
   y ← y + 1
   if (x < 100) goto loop
end:
```

Handling cyclic control flow

- SSA requires single definition for each use
- Introduce φ-functions to handle joins at loop headers too

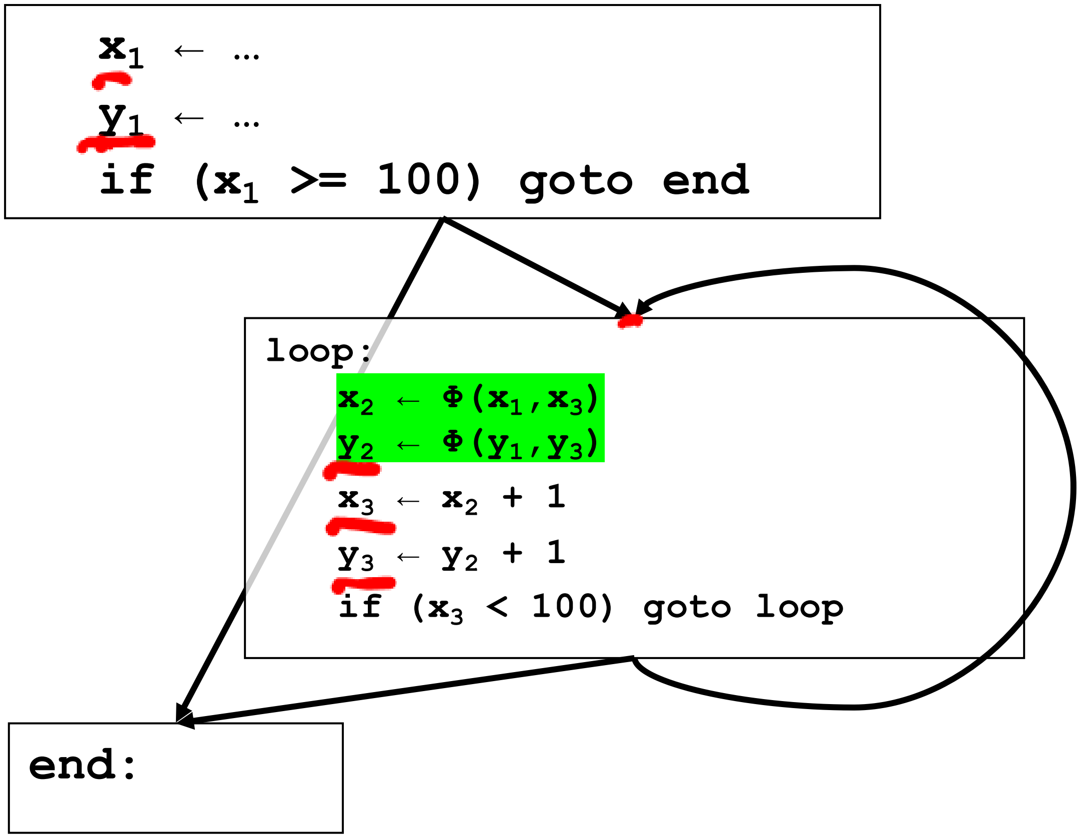



```
   x ← …
   y ← …
   if (x >= 100) goto end
loop:
   x ← x + 1
   y ← y + 1
   if (x < 100) goto loop
end:
```

Handling cyclic control flow

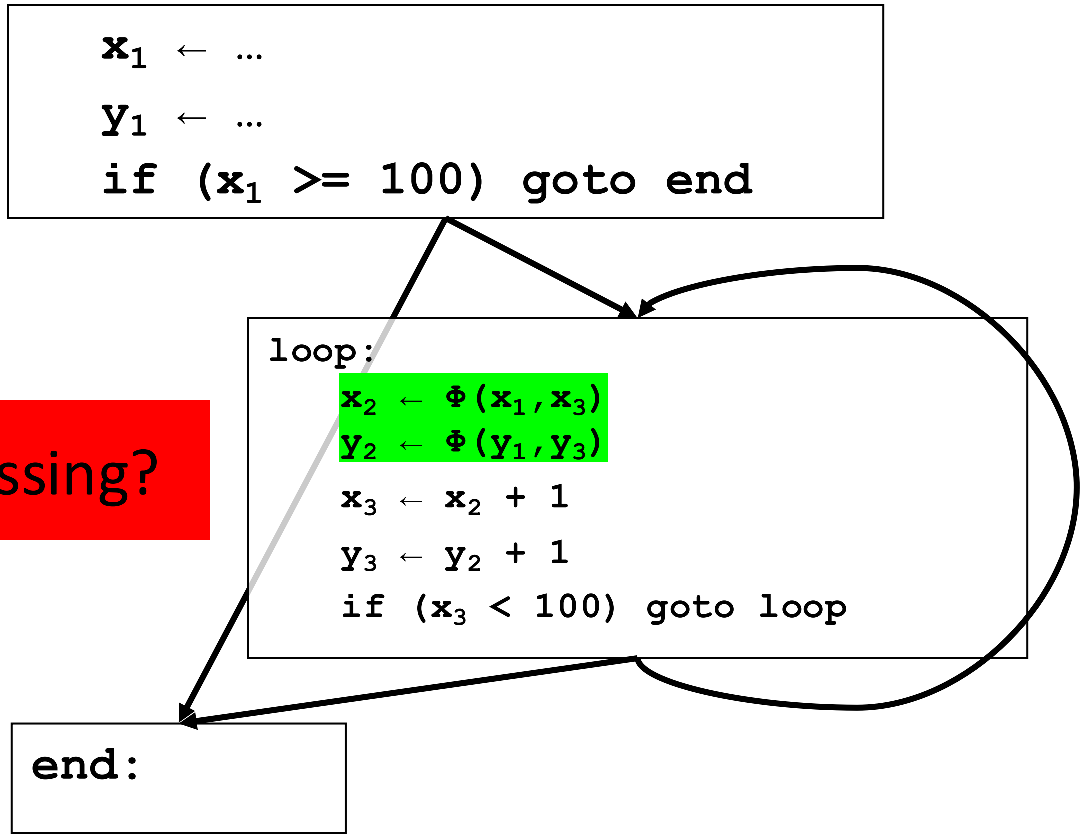
- SSA requires single definition for each use
- Introduce φ-functions to handle joins at loop headers too

```
   x ← …
   y ← …
   if (x >= 100) goto end
loop:
   x ← x + 1
   y ← y + 1
   if (x < 100) goto loop
end:
```



Handling cyclic control flow

- SSA requires single definition for each use
- Introduce φ-functions to handle joins at loop headers too

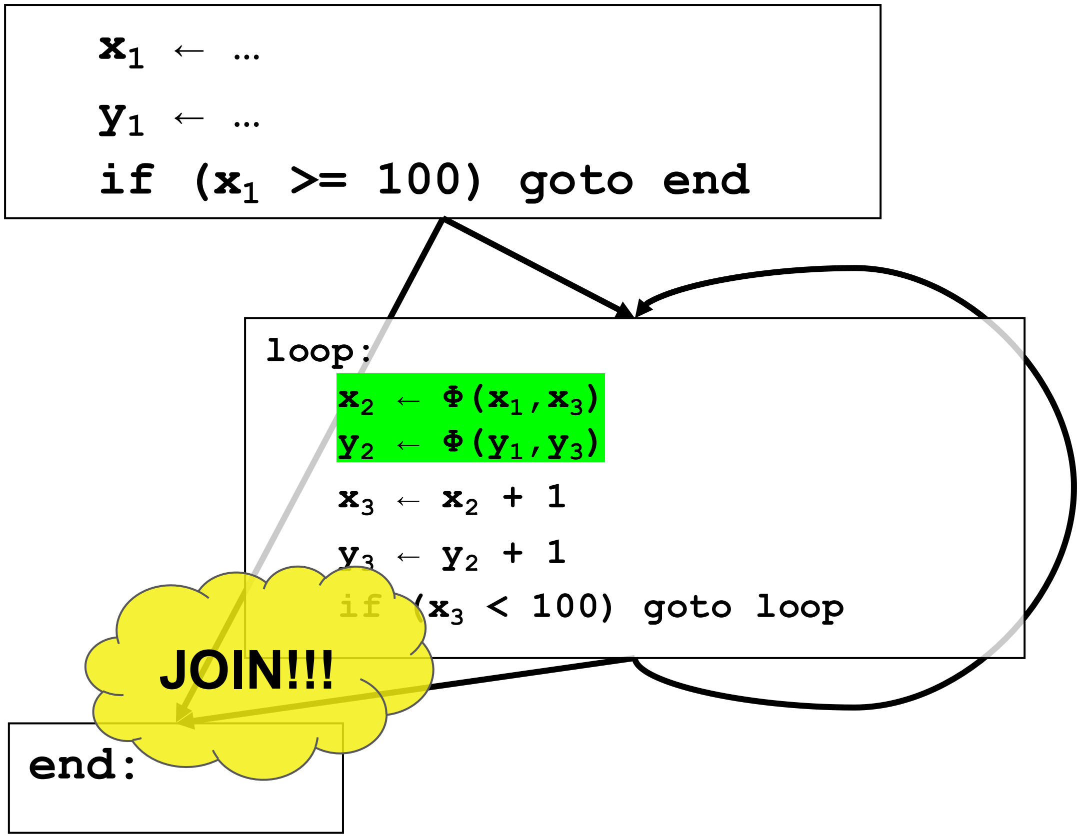


What’s missing?

```
   x ← …
   y ← …
   if (x >= 100) goto end
loop:
   x ← x + 1
   y ← y + 1
   if (x < 100) goto loop
end:
```

Handling cyclic control flow

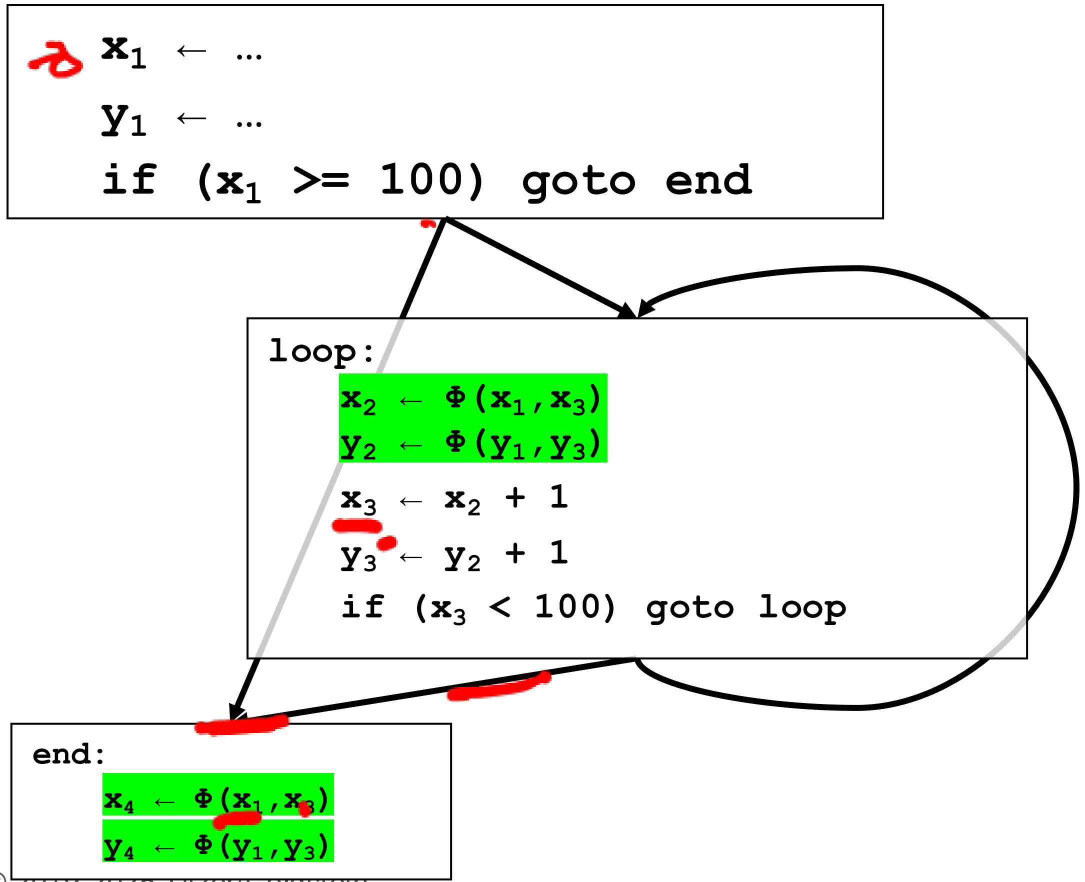
- SSA requires single definition for each use
- Introduce φ-functions to handle joins at loop headers too




```
   x ← …
   y ← …
   if (x >= 100) goto end
loop:
   x ← x + 1
   y ← y + 1
   if (x < 100) goto loop
end:
```

Handling cyclic control flow

- SSA requires single definition for each use
- Introduce φ-functions to handle joins at loop headers too




```
   x ← …
   y ← …
   if (x >= 100) goto end
loop:
   x ← x + 1
   y ← y + 1
   if (x < 100) goto loop
end:
```

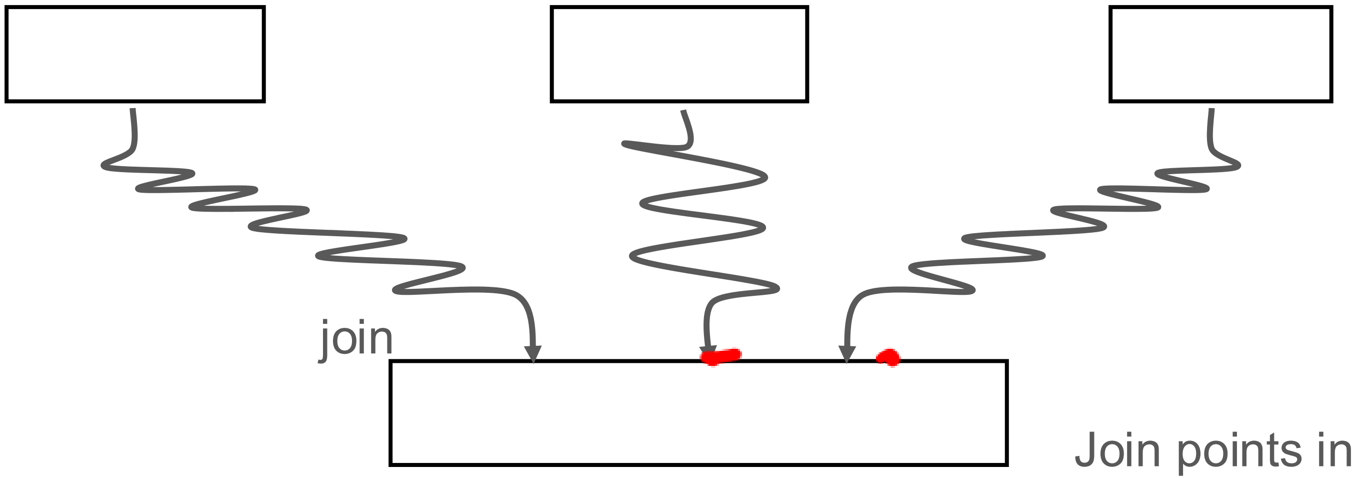
What is a Φ anyway?

- Φ is a fictional operator; it merges multiple definitions into a single definition at a join in the control flow graph.
- At a BB with *p* predecessors, there are *p* inputs to the Φ.

$$x_{new} \leftarrow \Phi(\mathbf{x_1}, \mathbf{x_2}, \mathbf{x_3}, \ldots, \mathbf{x_p})$$

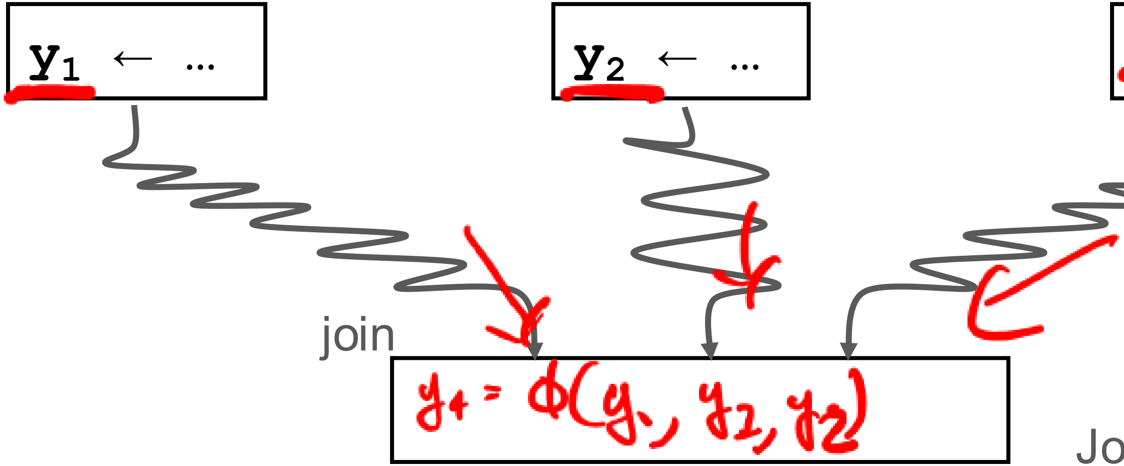
- What do the inputs to a Φ mean?
  - The inputs to φ-functions *positionally correspond* to the incoming control-flow edges.
  - They relate control flow merging and data flow merging.

What is a Φ anyway?



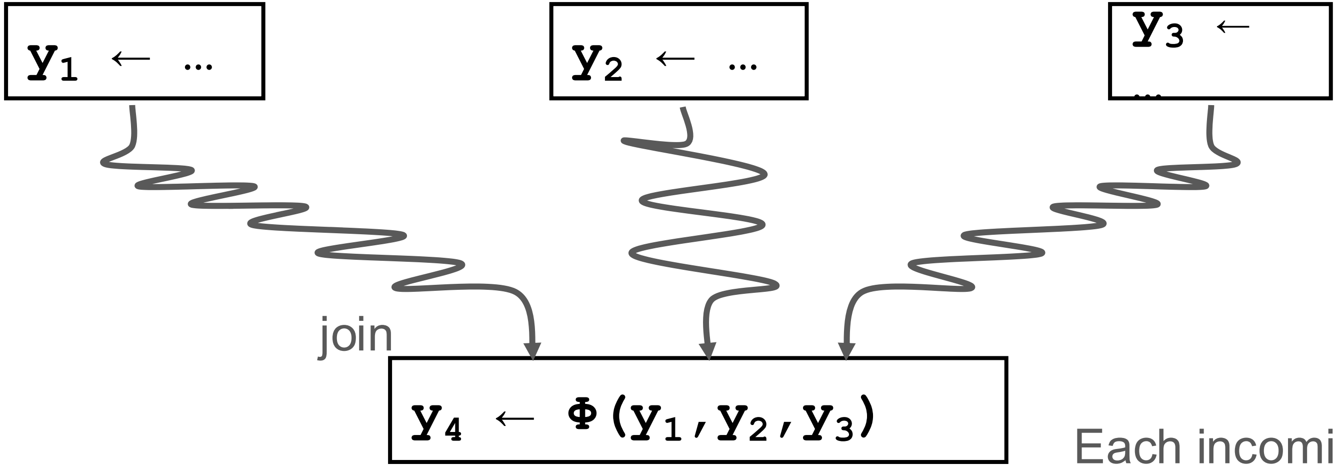
Join points in the control flow graph may require insertion of Φ functions.

What is a Φ anyway?



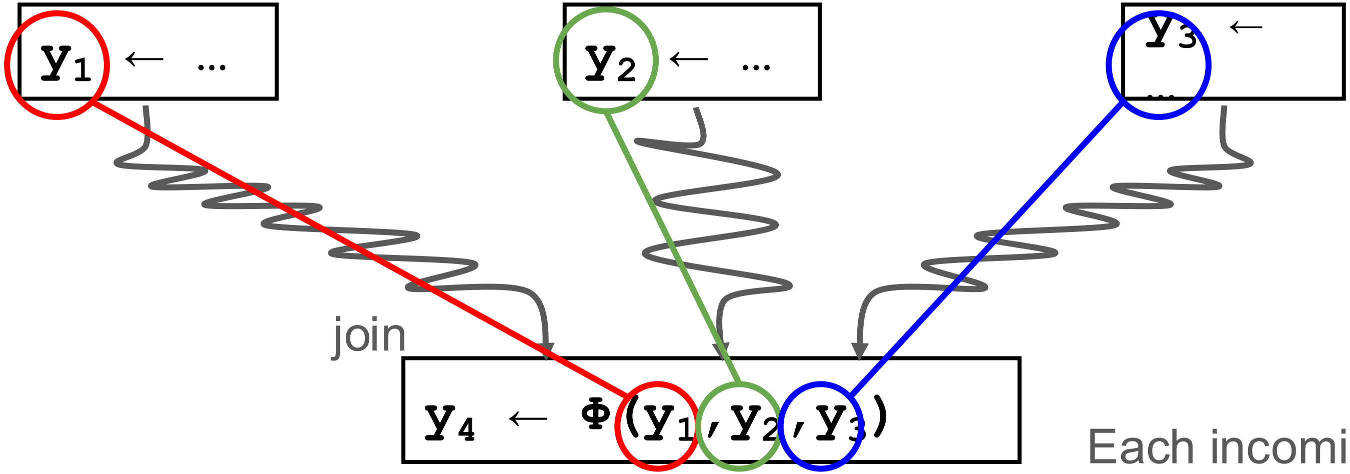
Join points in the control flow graph may require insertion of Φ functions, ***if there are different versions of the variable arriving.***

What is a Φ anyway?



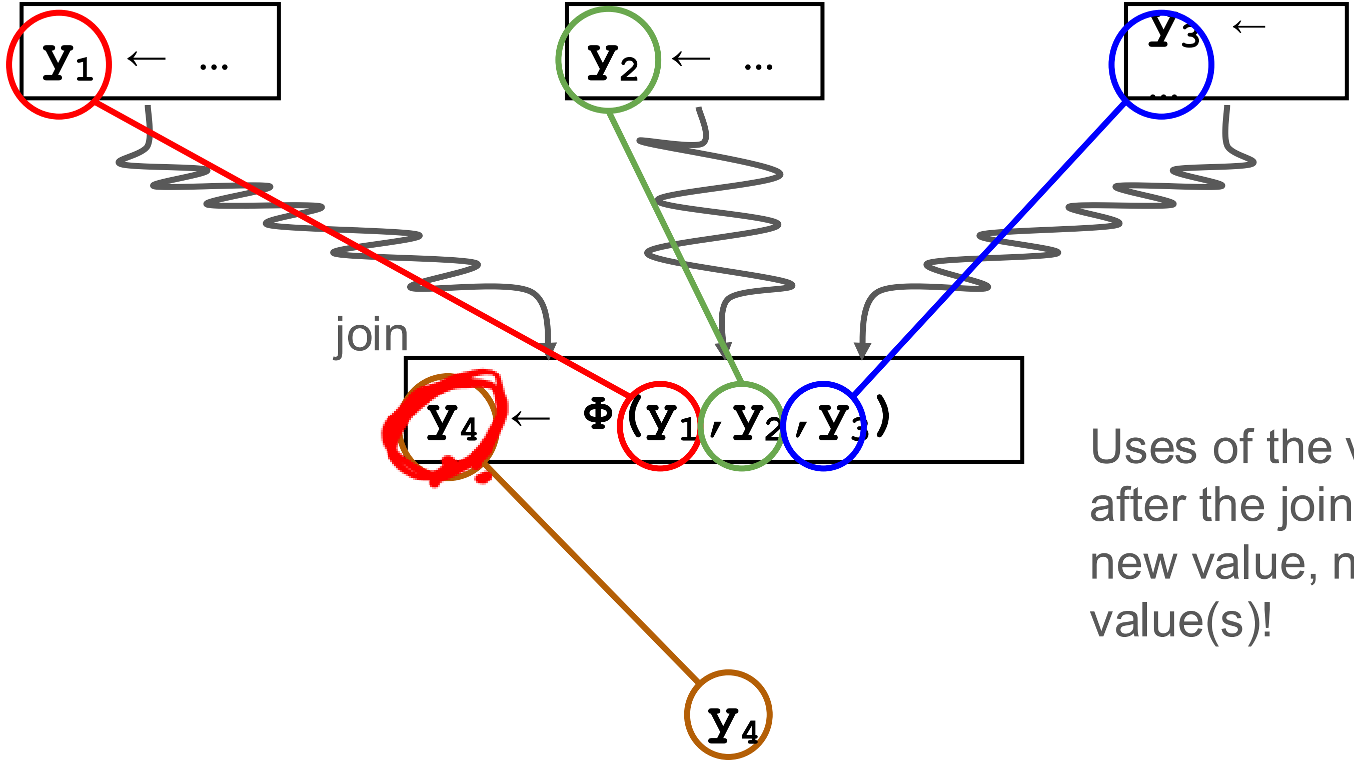
Each incoming control edge supplies a corresponding data value for the Φ from the predecessor.

What is a Φ anyway?



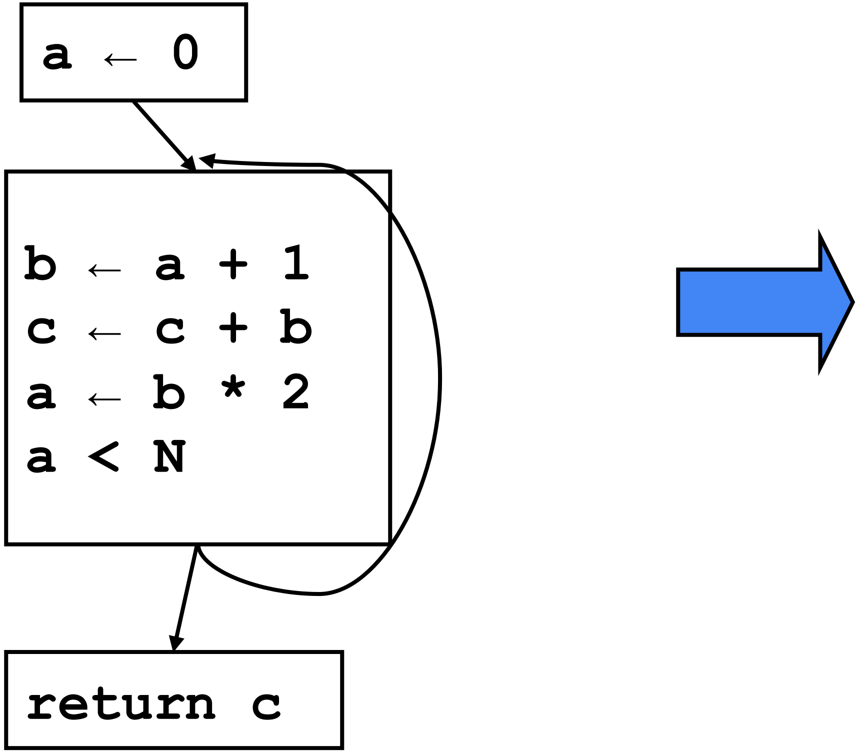
Each incoming control edge supplies a corresponding data value for the Φ from the predecessor.

What is a Φ anyway?

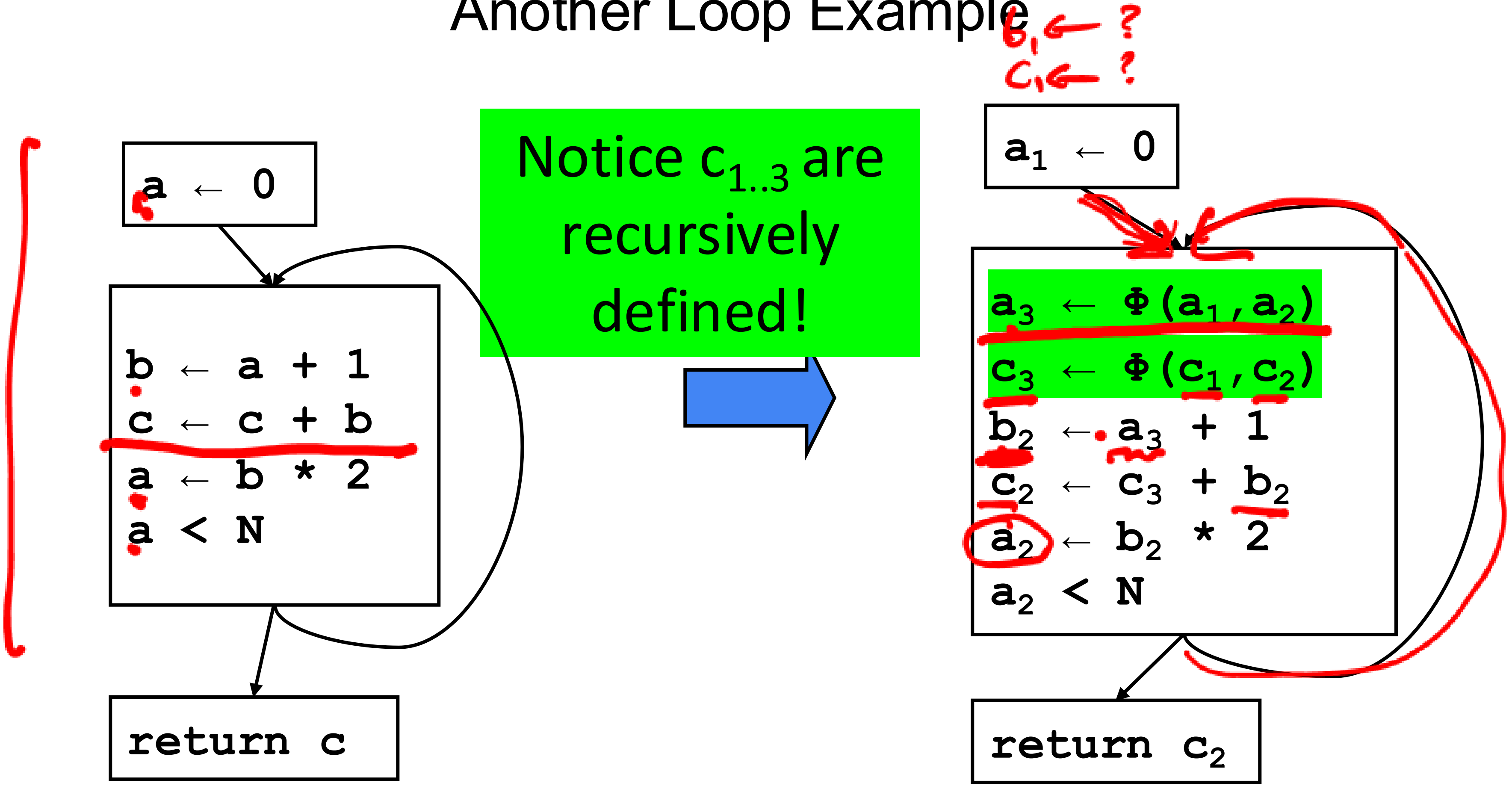


Uses of the variable after the join get the new value, not the old value(s)!

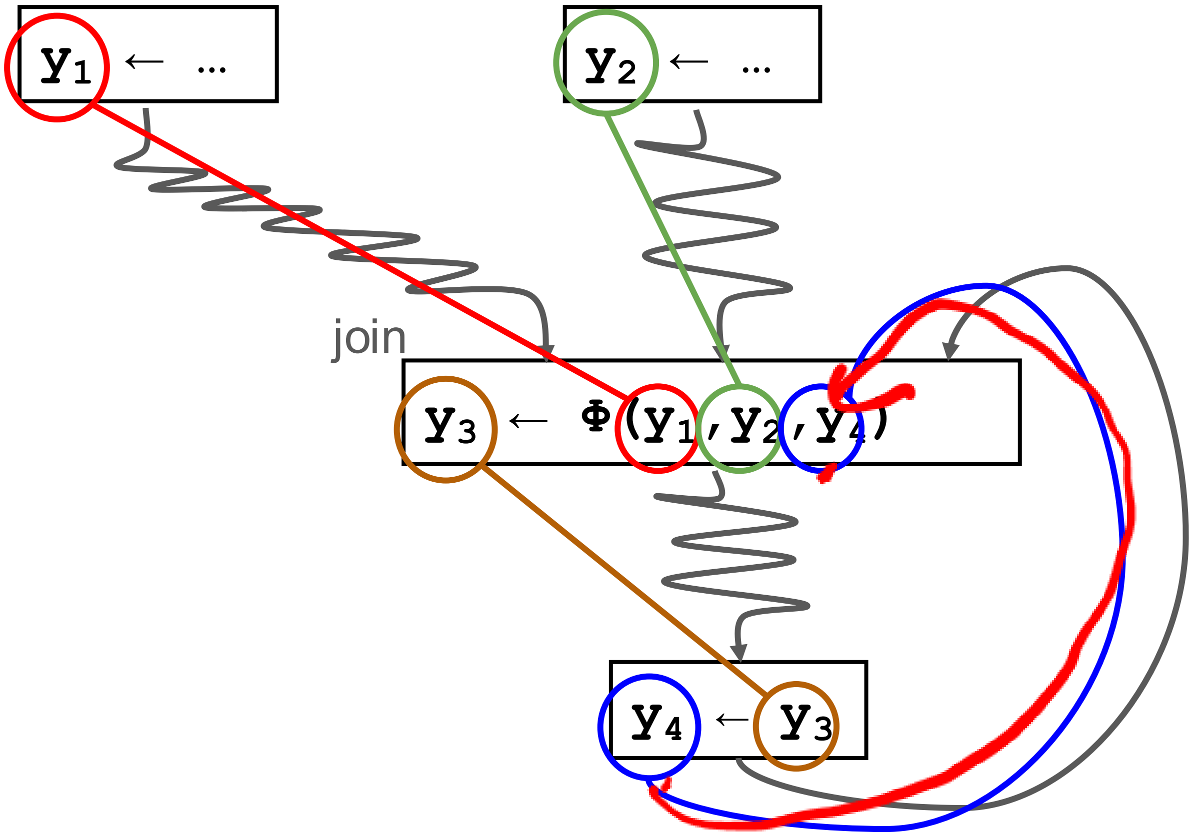
Another Loop Example



Another Loop Example



What is a Φ (for a loop) anyway?

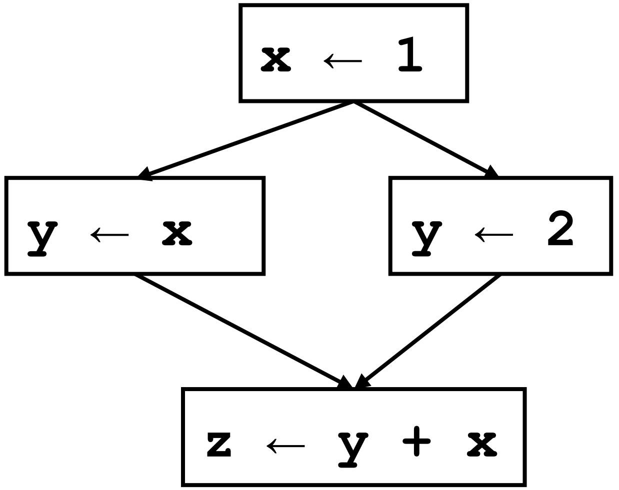


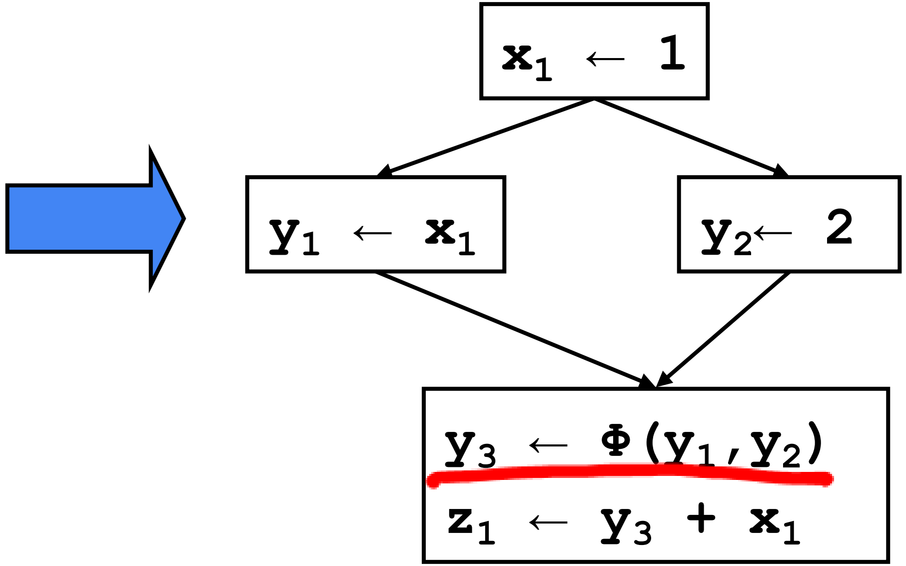
Φs at loop headers relate the dataflow on a loop backedge with the control flow.

Allows finding induction variables really easily.

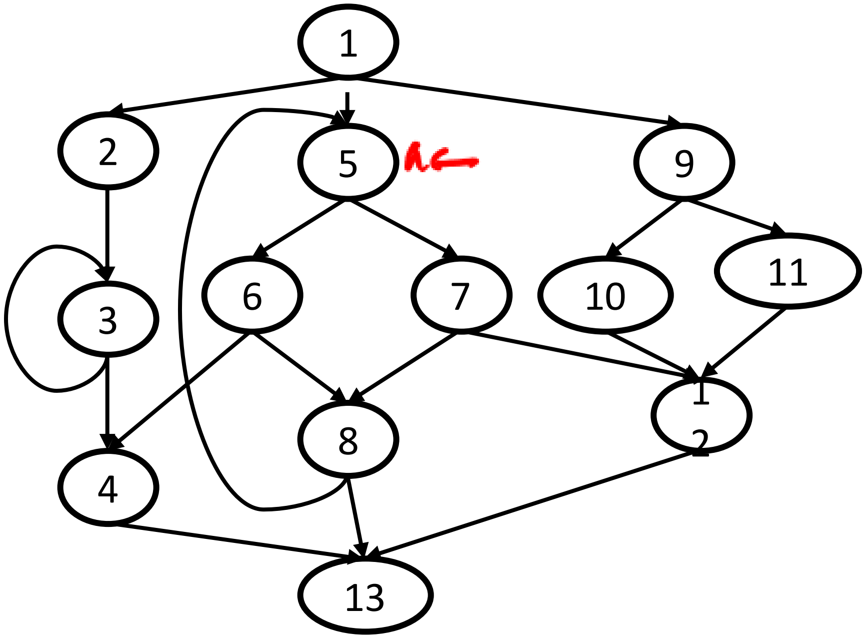
Minimal SSA

- Each assignment generates a fresh variable.
- At each join point insert Φ functions for all variables with multiple outstanding defs.





When do we insert Φ?



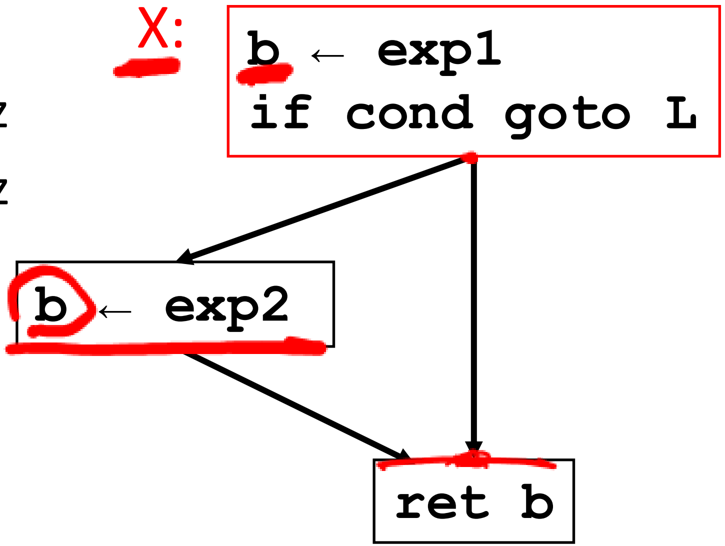
CFG

If there is a def of **a** in block 5, which nodes need a Φ()?

When do we insert Φ?

Require a Φ-function for variable b at node z of the flow graph:

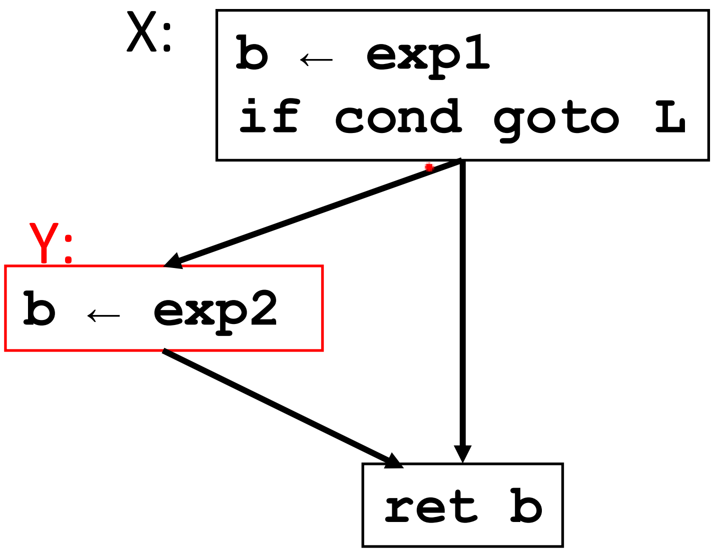
- There is a block x containing a def of b
- There is a block $y \neq x$ containing a def of b
- There is a nonempty path P_{xz} of edges from x to z
- There is a nonempty path P_{yz} of edges from y to z
- Paths P_{xz} and P_{yz} do not have any node in common other than z, and…
- The node z does not appear within both P_{xz} and P_{yz} prior to the end, though it may appear in one or the other.



When do we insert Φ?

Require a Φ-function for variable b at node z of the flow graph:

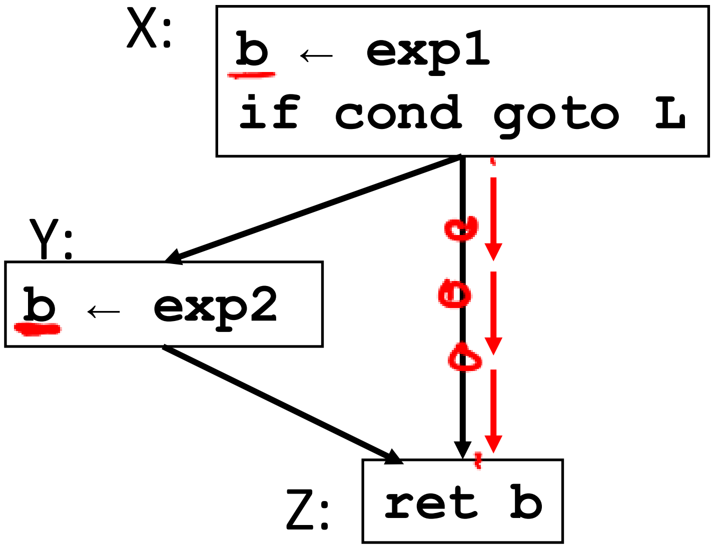
- There is a block x containing a def of b
- There is a block y ≠ x containing a def of b
- There is a nonempty path P_{xz} of edges from x to z
- There is a nonempty path P_{yz} of edges from y to z
- Paths P_{xz} and P_{yz} do not have any node in common other than z, and…
- The node z does not appear within both P_{xz} and P_{yz} prior to the end, though it may appear in one or the other.



When do we insert Φ?

Require a Φ-function for variable b at node z of the flow graph:

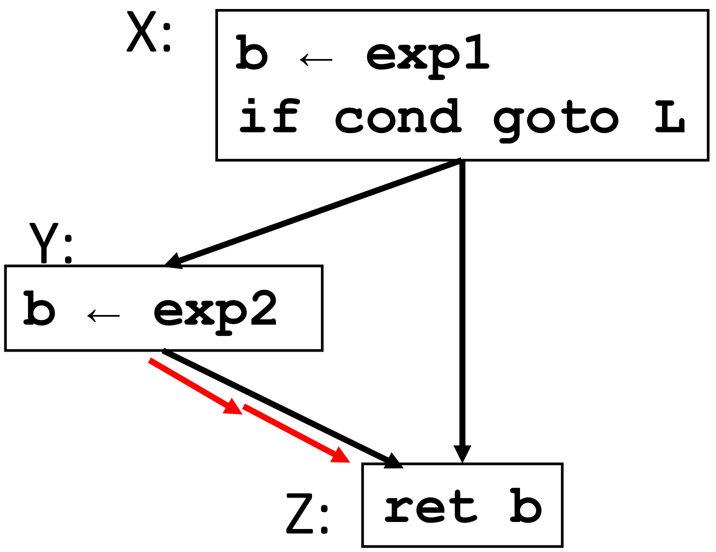
- There is a block x containing a def of b
- There is a block y ≠ x containing a def of b
- There is a nonempty path P_{xz} of edges from x to z
- There is a nonempty path P_{yz} of edges from y to z
- Paths P_{xz} and P_{yz} do not have any node in common other than z, and…
- The node z does not appear within both P_{xz} and P_{yz} prior to the end, though it may appear in one or the other.



When do we insert Φ?

Require a Φ-function for variable b at node z of the flow graph:

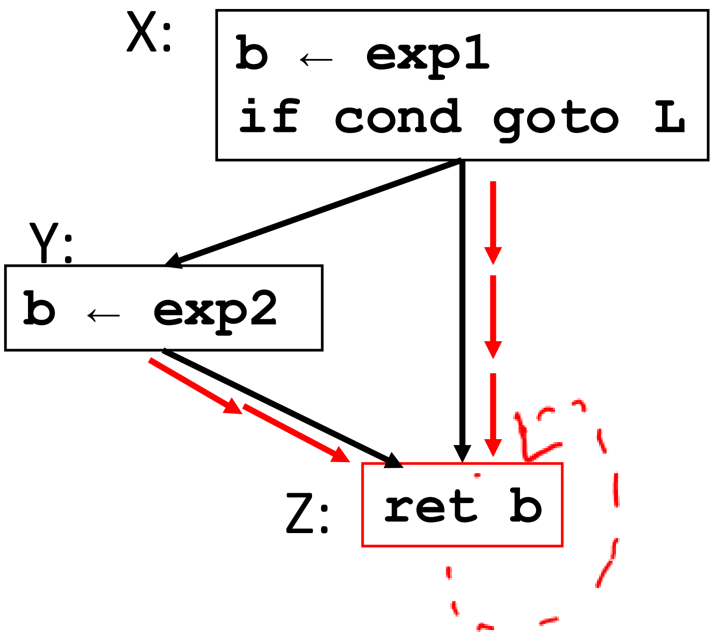
- There is a block x containing a def of b
- There is a block y ≠ x containing a def of b
- There is a nonempty path P_{xz} of edges from x to z
- There is a nonempty path P_{yz} of edges from y to z
- Paths P_{xz} and P_{yz} do not have any node in common other than z, and…
- The node z does not appear within both P_{xz} and P_{yz} prior to the end, though it may appear in one or the other.



When do we insert Φ?

Require a Φ-function for variable b at node z of the flow graph:

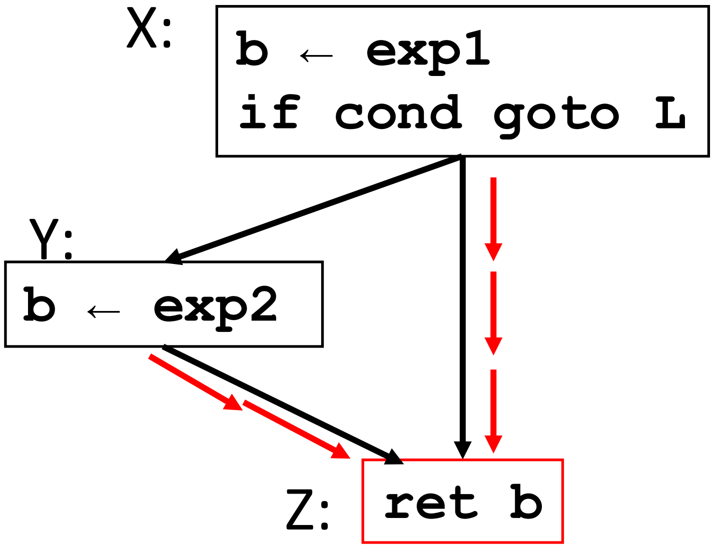
- There is a block x containing a def of b
- There is a block y ≠ x containing a def of b
- There is a nonempty path P_{xz} of edges from x to z
- There is a nonempty path P_{yz} of edges from y to z
- Paths P_{xz} and P_{yz} do not have any node in common other than z, and…
- The node z does not appear within both P_{xz} and P_{yz} prior to the end, though it may appear in one or the other.



When do we insert Φ?

Require a Φ-function for variable b at node z of the flow graph:

- There is a block x containing a def of b
- There is a block y ≠ x containing a def of b
- There is a nonempty path P_{xz} of edges from x to z
- There is a nonempty path P_{yz} of edges from y to z
- Paths P_{xz} and P_{yz} do not have any node in common other than z, and…
- The node z does not appear within both P_{xz} and P_{yz} prior to the end, though it may appear in one or the other.



Iterative Insertion

- Implicit def of every variable in start node
- Inserting Φ-function creates new definition
- While there ∃ x,y,z that
  - satisfy path-convergence criteria
  - and z does not contain Φ-function for b
- do
  - insert $b \leftarrow \Phi(b,b,b,\ldots,b_n)$ at node z, z having n predecessors.

Dominance Property of SSA

- In SSA **definitions dominate uses***.
  - If x_i is used in $x \leftarrow \Phi(\ldots, x_i, \ldots)$, then BB($x_i$) dominates i^{th} predecessor of BB(Φ)
  - If x is used in $y \leftarrow \ldots x \ldots$, then BB(x) dominates BB(y)
- We can use this for an efficient algorithm to convert to SSA

Dominance Property of SSA

- In SSA **definitions dominate uses***.
  - If x_i is used in $x \leftarrow \Phi(\ldots, x_i, \ldots)$, then BB($x_i$) dominates i^{th} predecessor of BB(Φ)
  - If x is used in $y \leftarrow \ldots x \ldots$, then BB(x) dominates BB(y)
- We can use this for an efficient algorithm to convert to SSA

well *akshully, this only true for strict SSA**,
where all variables are defined before they are used.

Dominance Property of SSA

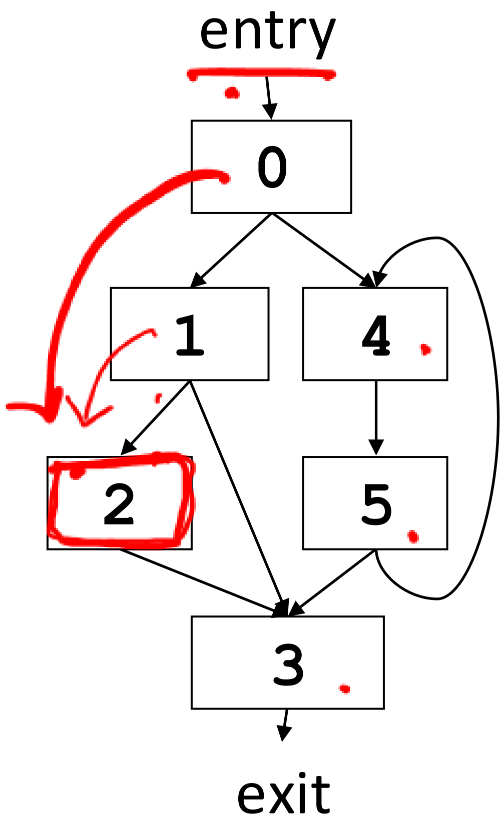
- In SSA **definitions dominate uses***.
  - If x_i is used in $x \leftarrow \Phi(\ldots, x_i, \ldots)$, then BB($x_i$) dominates i^{th} predecessor of BB(Φ)
  - If x is used in $y \leftarrow \ldots x \ldots$, then BB(x) dominates BB(y)
- We can use this for an efficient algorithm to convert to SSA

well *akshully, this only true for strict SSA**,
where all variables are defined before they are used.
****well *double akshully***, we can insert assignments to convert any program to strict SSA

Side trip: Dominators

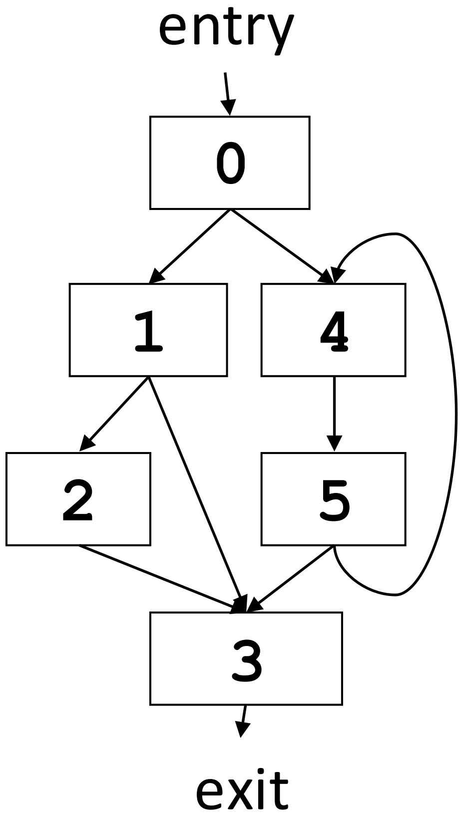
Dominators

- *a dom b*
  - block *a* dominates block *b* if every possible execution path from *entry* to *b* includes *a*



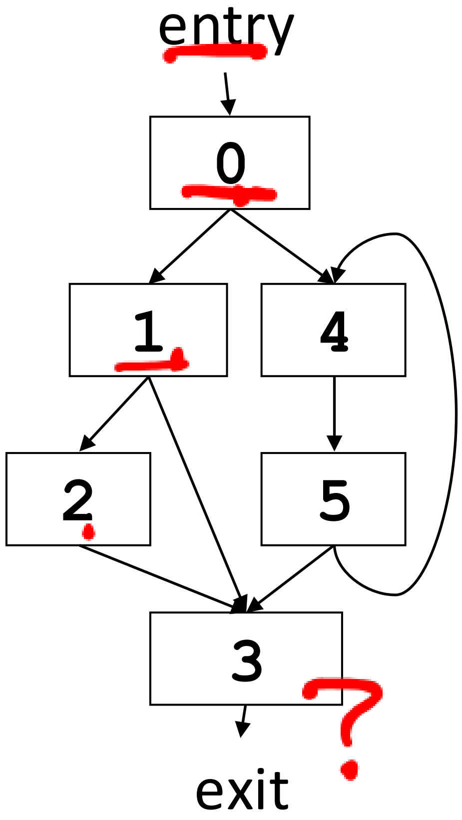
Dominators

- *a dom b*
  - block *a* *dominates* block *b* if every possible execution path from *entry* to *b* includes *a*



Dominators

- *a dom b*
  - block *a* *dominates* block *b* if every possible execution path from *entry* to *b* includes *a*
    - **entry** dominates everything
    - **0** dominates everything but entry
    - **1** dominates **2** and **1**

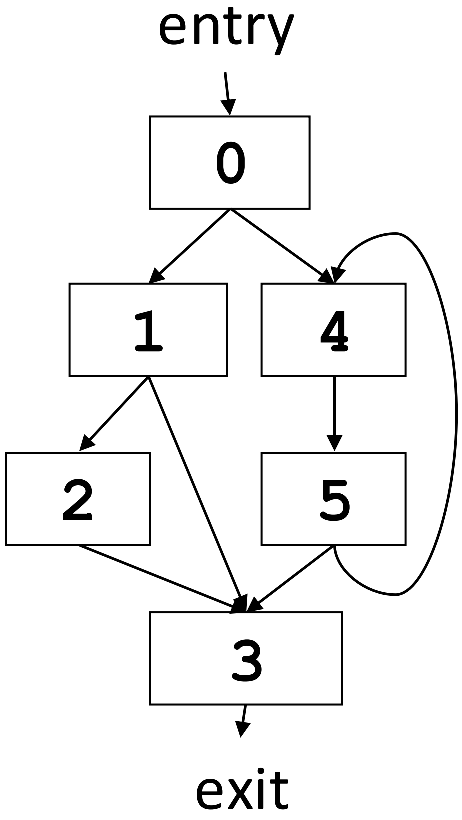


Dominators

- *a dom b*
  - block *a* dominates block *b* if every possible execution path from *entry* to *b* includes *a*

Dominators are useful in:

- *Dataflow analysis*
- *Constructing SSA*
- *Identifying "natural" loops*
- *Code motion*
- *…*



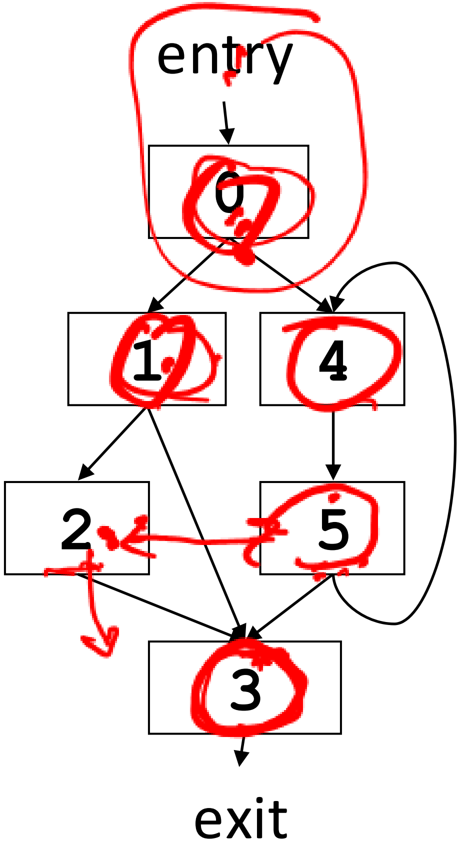
Definitions

- *a sdom b*

  If *a* and *b* are different blocks and *a dom b*, we say that *a* ***strictly dominates*** *b*

- *a idom b*

  If *a sdom b*, and there is no *c* such that *a sdom c* and *c sdom b*, we say that *a* is the ***immediate dominator*** of *b*



Properties of Dom

- Dominance is a partial order on the blocks of the flow graph, i.e.,
  - 1. Reflexivity: a dom a for all a
  - 2. Anti-symmetry: a dom b and b dom a implies a = b
  - 3. Transitivity: a dom b and b dom c implies a dom c
- NOTE: there may be blocks a and b such that neither a dom b or b dom a holds.
- The dominators of each node n are linearly ordered by the dom relation. The dominators of n appear in this linear order on any path from the initial node to n.

Computing dominators

- We want to compute $D[n]$, the set of blocks that dominate n

Initialize each $D[n]$ (except $D[\text{entry}]$) to be the set of all blocks, and then iterate until no $D[n]$ changes:

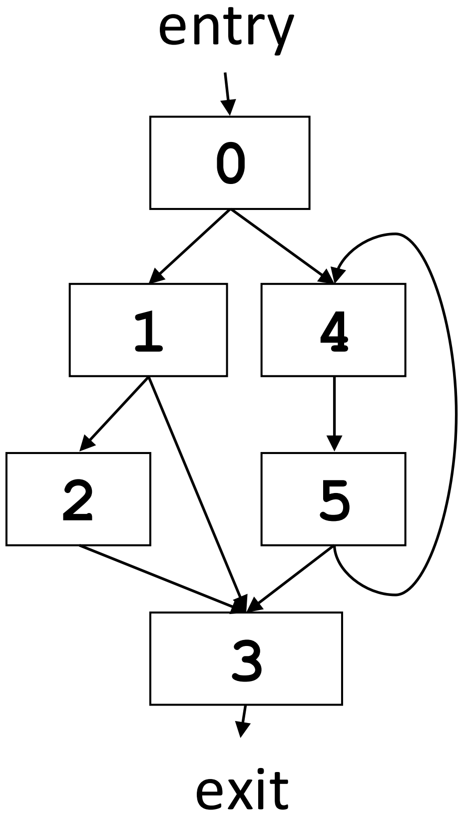
$$D[\text{entry}] = \{\text{entry}\}$$

$$D[n] = \{n\} \cup \left(\bigcap_{p \in \text{pred}(n)} D[p] \right), \quad \text{for } n \neq \text{entry}$$

Example

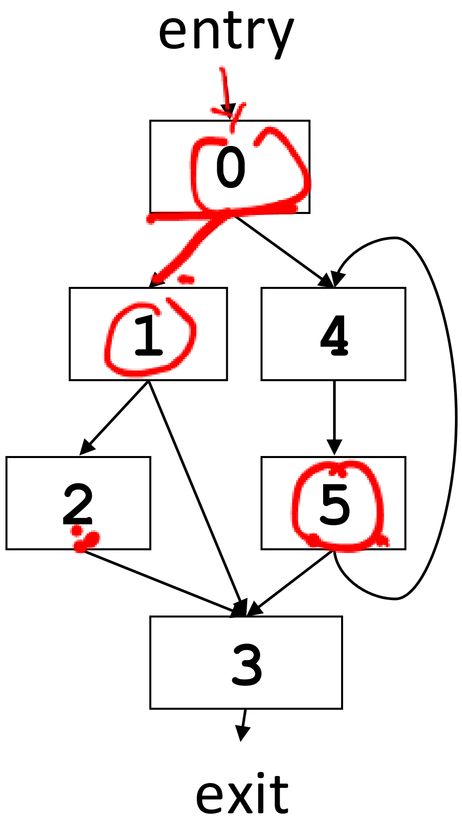
Initialization

block	D[n]
entry	{entry}
0	{entry,0,1,2,3,4,5,exit}
1	{entry,0,1,2,3,4,5,exit}
2	{entry,0,1,2,3,4,5,exit}
3	{entry,0,1,2,3,4,5,exit}
4	{entry,0,1,2,3,4,5,exit}
5	{entry,0,1,2,3,4,5,exit}
exit	{entry,0,1,2,3,4,5,exit}



Example

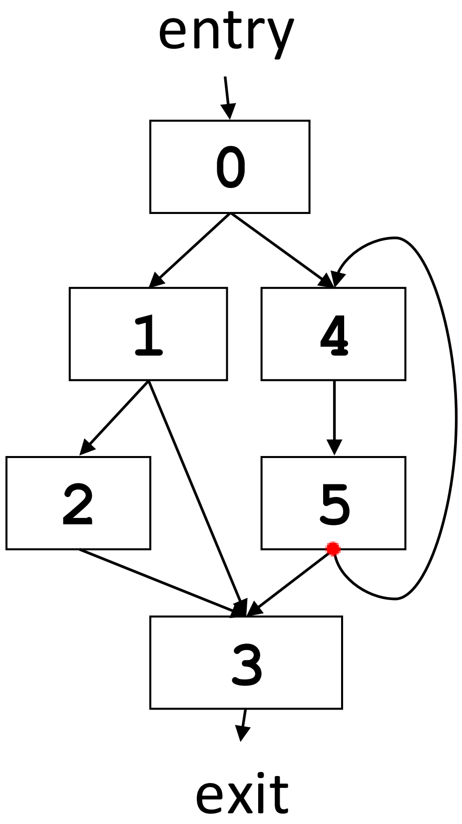
block	Initialization D[n]	First Pass D[n]
entry	{entry}	{entry}
0	{entry,0,1,2,3,4,5,exit}	{0,entry}
1	{entry,0,1,2,3,4,5,exit}	{1,0,entry}
2	{entry,0,1,2,3,4,5,exit}	{2,1,0,entry}
3	{entry,0,1,2,3,4,5,exit}	{3,1,0,entry}
4	{entry,0,1,2,3,4,5,exit}	{4,0,entry}
5	{entry,0,1,2,3,4,5,exit}	{5,4,0,entry}
exit	{entry,0,1,2,3,4,5,exit}	{exit,3,1,0,entry}



Update rule: $D[n] = \{n\} \cup \left(\bigcap_{p \in \text{pred}(n)} D[p] \right)$;

Example

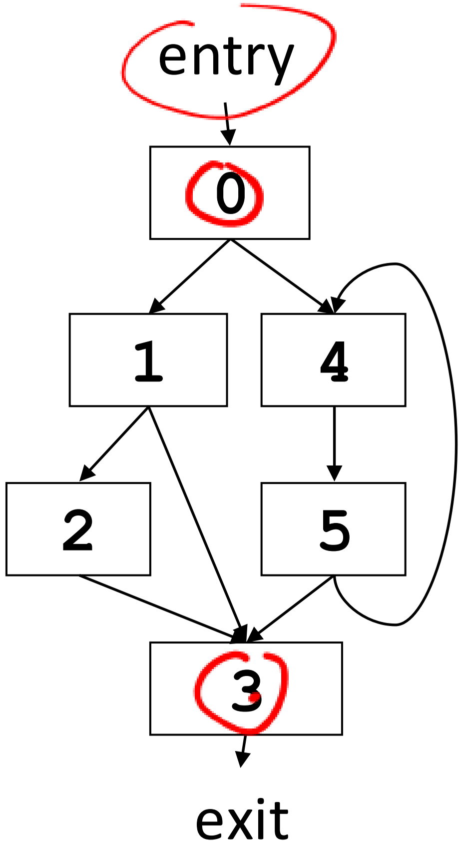
block	First Pass D[n]	Second Pass D[n]
entry	{entry}	{entry}
0	{0,entry}	{0,entry}
1	{1,0,entry}	{1,0,entry}
2	{2,1,0,entry}	{2,1,0,entry}
3	{3,1,0,entry}	{3,0,entry}
4	{4,0,entry}	{4,0,entry}
5	{5,4,0,entry}	{5,4,0,entry}
exit	{exit,3,1,0,entry}	{exit,3,0,entry}



Update rule: $$D[n] = \{n\} \cup \left(\bigcap_{p \in \text{pred}(n)} D[p] \right).$$

Example

block	Second Pass D[n]	Third Pass D[n]
entry	{entry}	{entry}
0	{0,entry}	{0,entry}
1	{1,0,entry}	{1,0,entry}
2	{2,1,0,entry}	{2,1,0,entry}
3	{3,0,entry}	{3,0,entry}
4	{4,0,entry}	{4,0,entry}
5	{5,4,0,entry}	{5,4,0,entry}
exit	{exit,3,0,entry}	{exit,3,0,entry}



Update rule: $$D[n] = \{n\} \cup \left(\bigcap_{p \in \text{pred}(n)} D[p] \right);$$

Computing dominators

- Iterative algorithm is $O(n^2 e)$
  - assuming bit vector set
  - choosing a good iteration order matters
- More efficient algorithm due to Lengauer and Tarjan
  - $O(e \cdot \alpha(e,n))$

    $\alpha(e,n)$ is *inverse Ackermann*
  - much more complicated
  - Books provide simple algorithms that are fast in practice (faster than Tarjan algorithm for realistic CFGs)
  - For a clever algorithm see: "A Simple, Fast Dominance Algorithm" by Cooper, Harvey, and Kennedy"

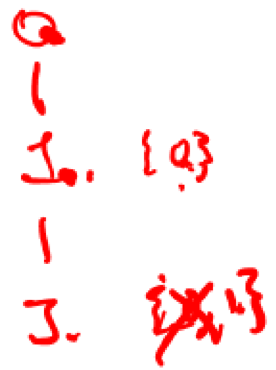
Immediate dominators

- Let *sD*[*n*] be the set of blocks that strictly dominate *n*, then

$$sD[n] = D[n] - \{n\}$$

- To compute *iD*[*n*], the set of blocks (size <= 1) that immediately dominate *n*
- Set $iD[n] = sD[n]$
- Repeat until no iD[n] changes:

$$iD[n] = iD[n] - \bigcup_{d \in iD[n]} (sD[d])$$

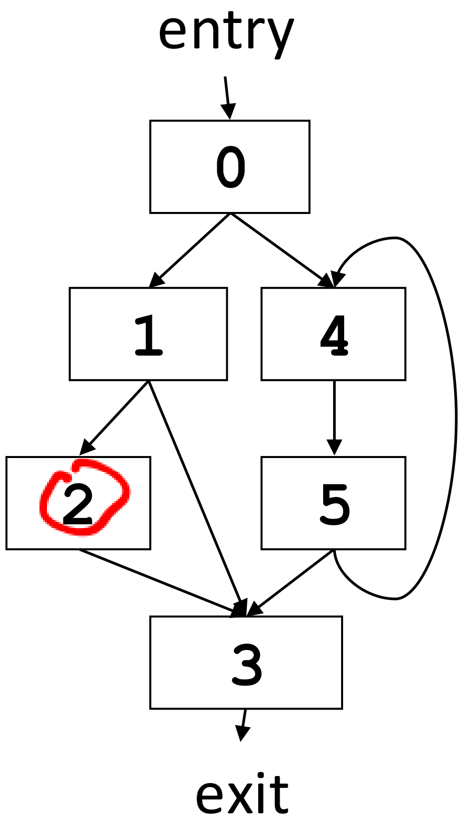


Example

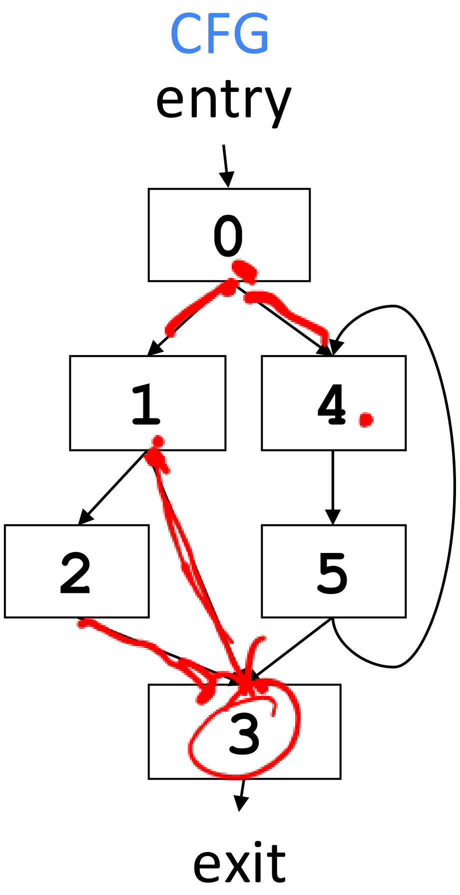
block	Initialization iD[n]=sD[n]	First Pass iD[n]
entry	{}	{}
0	{entry}	{entry}
1	{0,entry}	{0}
2	{1,0,entry}	{1}
3	{0,entry}	{0}
4	{0,entry}	{0}
5	{4,0,entry}	{4}
exit	{3,0,entry}	{3}

Update rule: $$iD[n] = iD[n] - \bigcup_{d \in iD[n]} (sD[d])$$

CFG

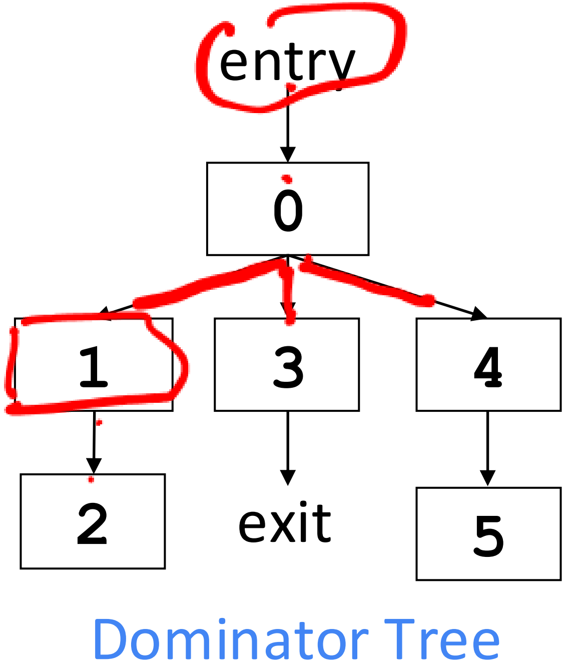


Dominator Tree

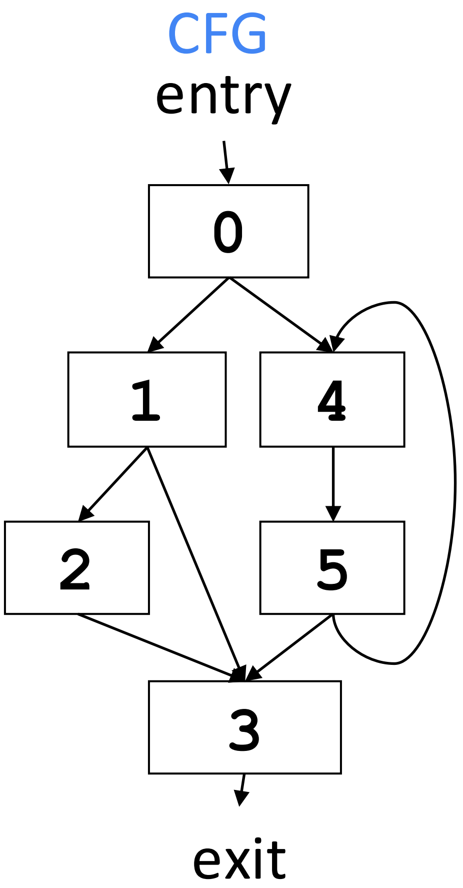


In the *dominator tree* the initial node is the entry block, and the parent of each other node is its immediate dominator.

block	iD[n]
entry	{}
0	{entry}
1	{0}
2	{1}
3	{0}
4	{0}
5	{4}
exit	{3}



Dominance Frontier

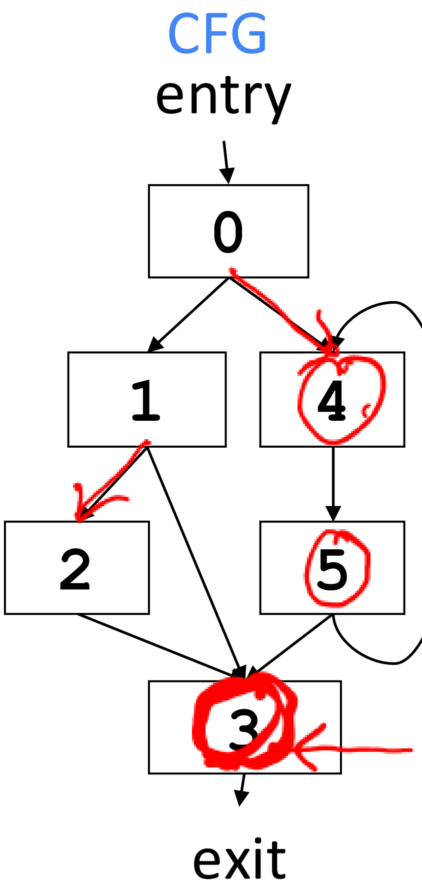


- *z* is in the dominance frontier of *x* If *z* is the first node we encounter on the path from *x* which *x* does not *strictly* dominate.
- For some path from node *x* to *z*,

  $x \to \ldots \to y \to z$

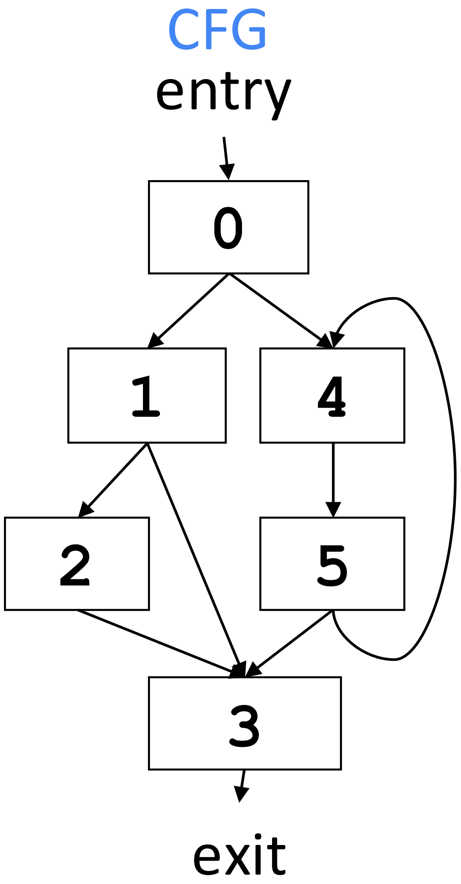
  where *x* dom *y* but not *x* sdom *z*.
- Intuitively, the dominance frontier consists of nodes "just outside the dominator tree"

Dominance Frontier

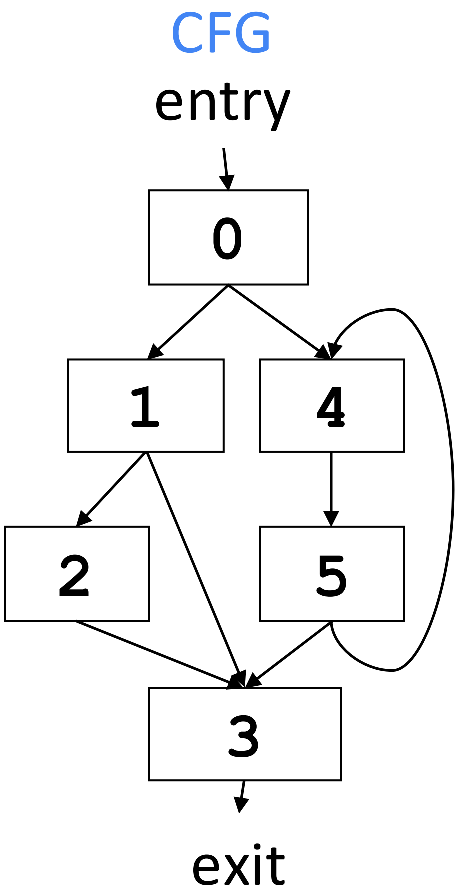


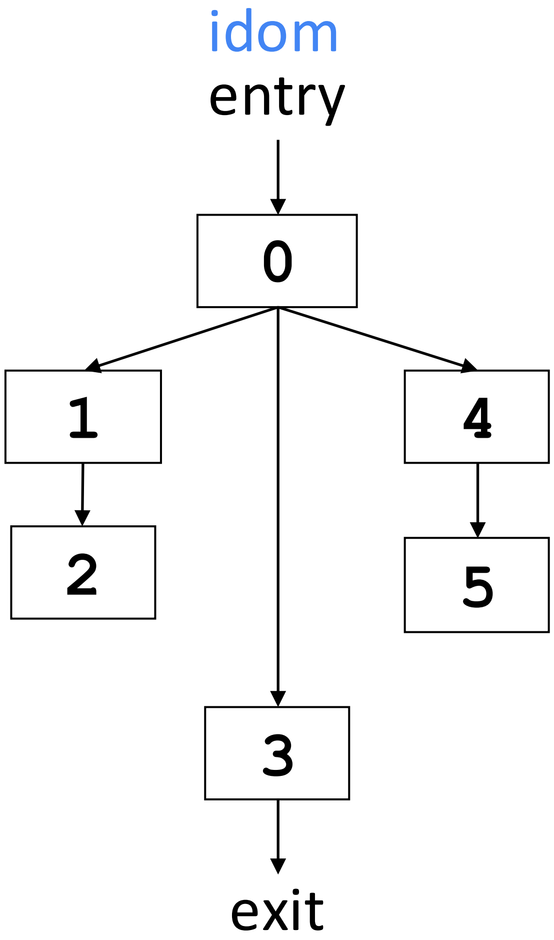
- *z* is in the dominance frontier of *x* If *z* is the first node we encounter on the path from *x* which *x* does not *strictly* dominate.
- For some path from node *x* to *z*,
  $x \rightarrow \ldots \rightarrow y \rightarrow z$
  where *x* dom *y* but not *x* sdom *z*.
- Dominance frontier of **1**?
- Dominance frontier of **2**?
- Dominance frontier of **4**?

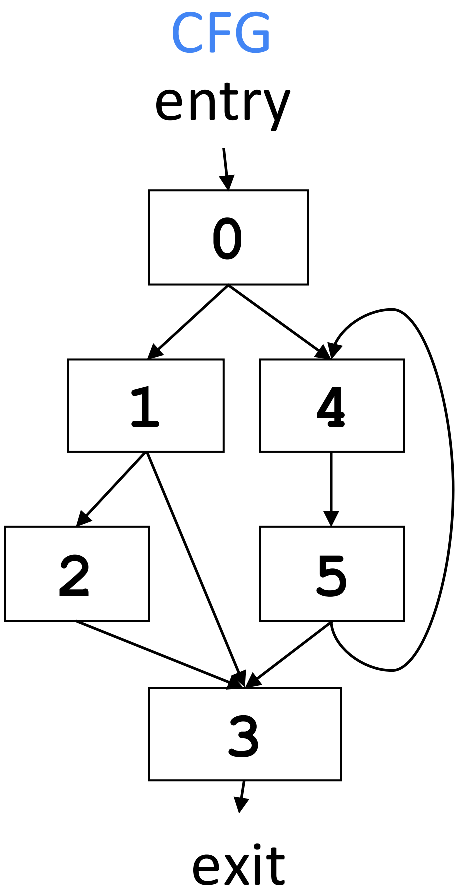
Dominance Frontier

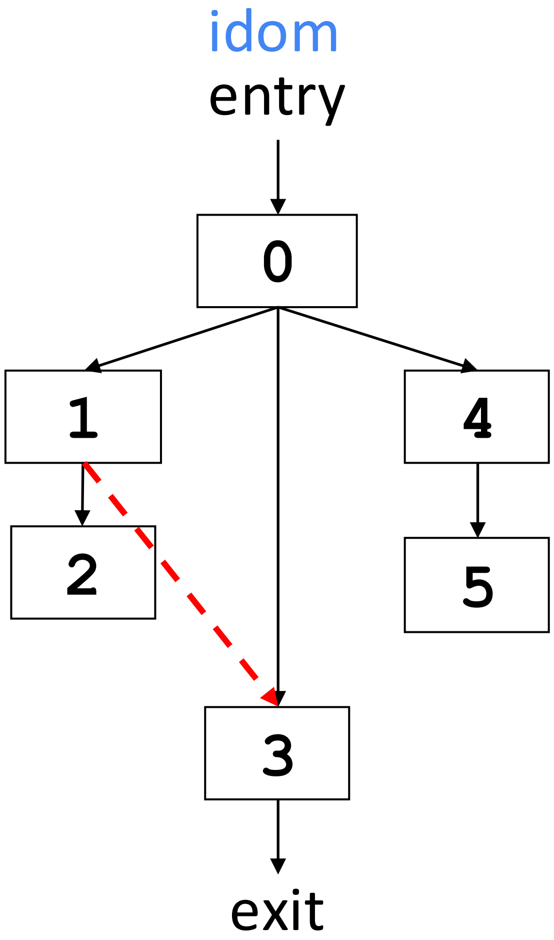


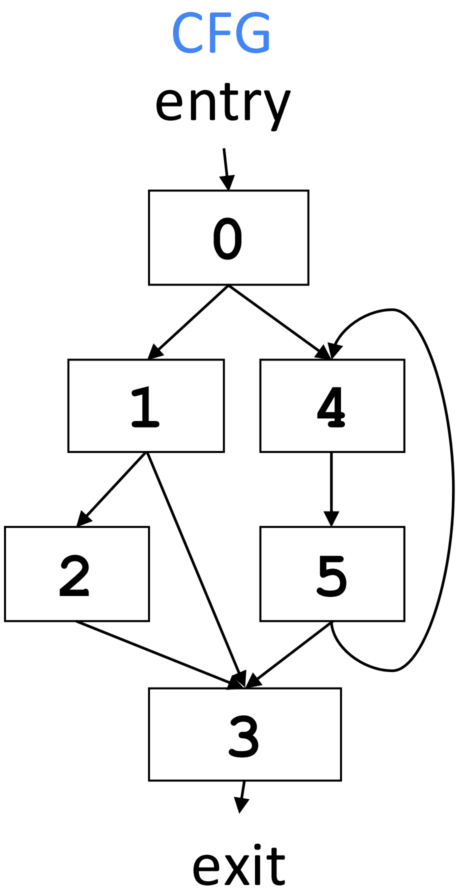
- *z* is in the dominance frontier of *x* If *z* is the first node we encounter on the path from *x* which *x* does not *strictly* dominate.
- For some path from node *x* to *z*,
  $x \rightarrow \ldots \rightarrow y \rightarrow z$
  where *x* dom *y* but not *x* sdom *z*.
- Dominance frontier of **1**? {3}
- Dominance frontier of **2**? {3}
- Dominance frontier of **4**? {3,4}

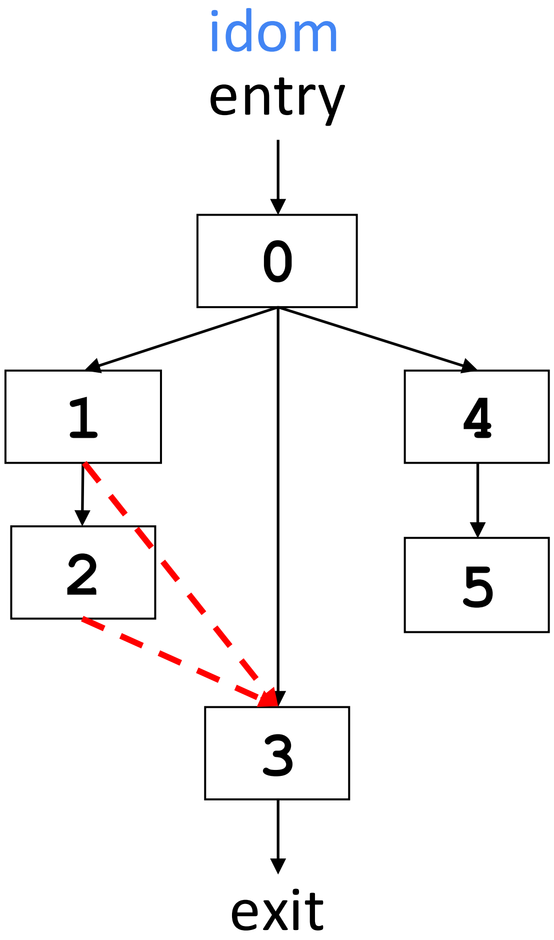

CFG
entry
0
1
4
2
5
3
exit

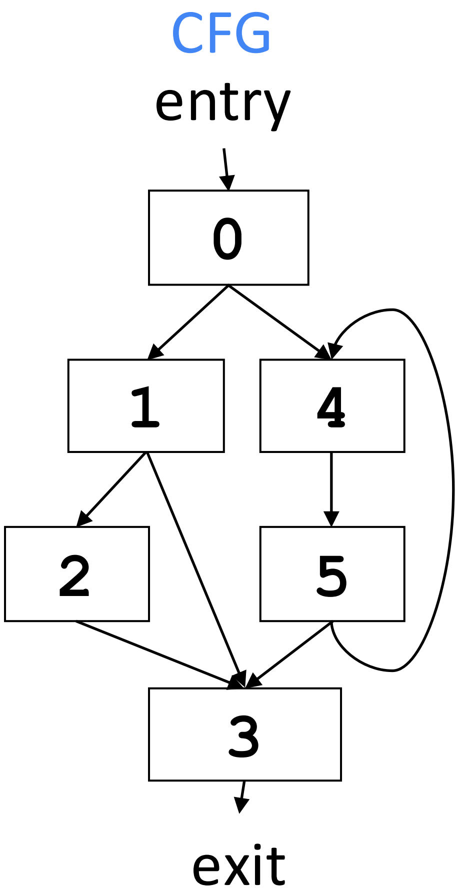

idom
entry
0
1
4
2
5
3
exit

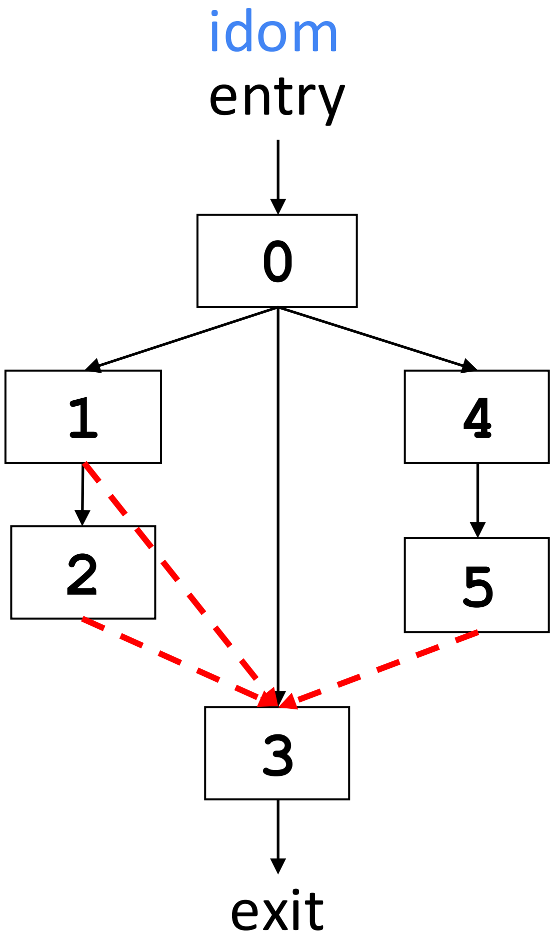

CFG
entry
0
1
4
2
5
3
exit

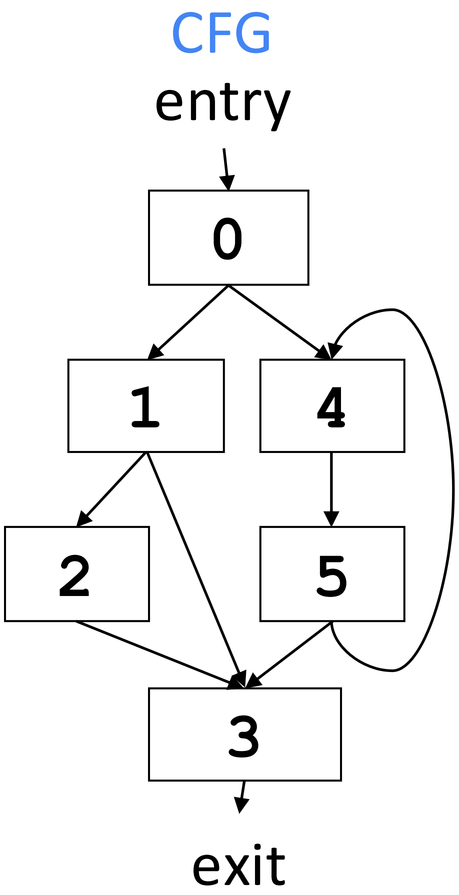

idom
entry
0
1
4
2
5
3
exit


CFG
entry
0
1
4
2
5
3
exit

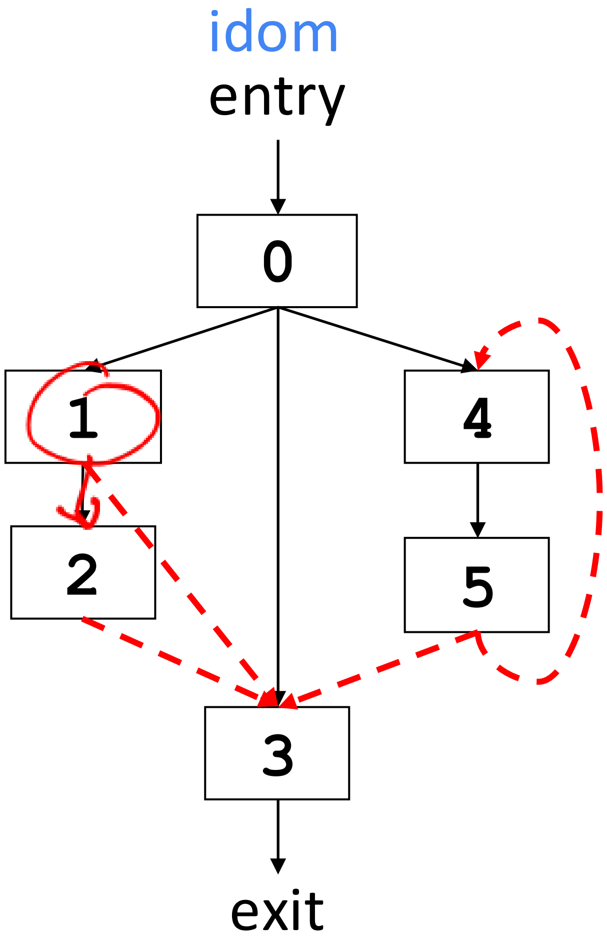

idom
entry
0
1
4
2
5
3
exit


CFG
entry
0
1
4
2
5
3
exit


idom
entry
0
1
4
2
5
3
exit



CFG
entry
0
1
4
2
5
3
exit



idom
entry
0
1
4
2
5
3
exit

Calculating the Dominance Frontier

- Let *dominates[n]* be the set of all blocks which block n dominates
  - subtree of dominator tree with n as the root
- The dominance frontier of n, *DF[n]* is

$$DF[n] = \bigcup_{s \in dominates[n]} succ(s) - dominates[n] - \{n\}$$

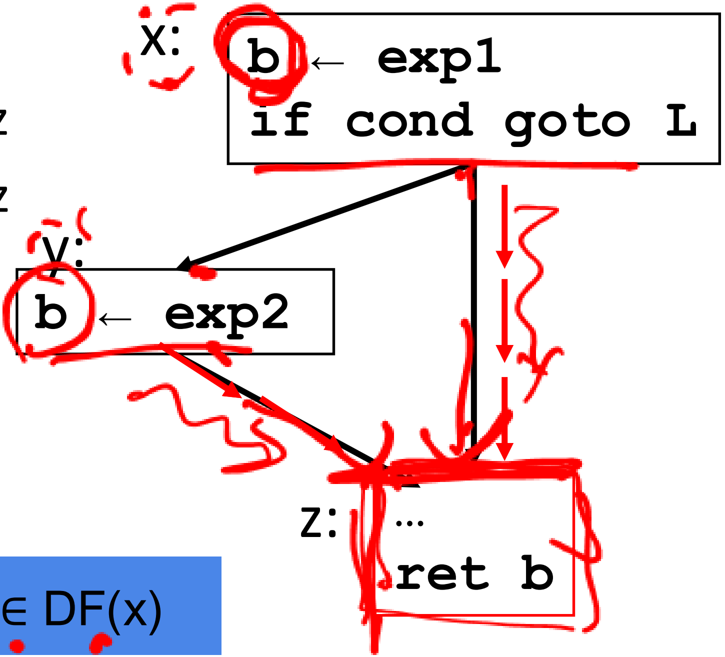
Recap

- *a dom b*
  - every possible execution path from *entry* to *b* includes *a*
- *a sdom b*
  - *a dom b* and *a* != *b*
- *a idom b*
  - *a* is “closest” dominator of *b*
- *a pdom b*
  - every path from *a* to the exit block includes *b*
- Dominator trees
- Dominance frontier

Back to inserting Φs

Require a Φ-function for variable b at node z of the flow graph:

- There is a block x containing a def of b
- There is a block y ≠ x containing a def of b
- There is a nonempty path P_{xz} of edges from x to z
- There is a nonempty path P_{yz} of edges from y to z
- Paths P_{xz} and P_{yz} do not have any node in common other than z, and…
- The node z does not appear within both P_{xz} and P_{yz} prior to the end, though it may appear in one or the other.



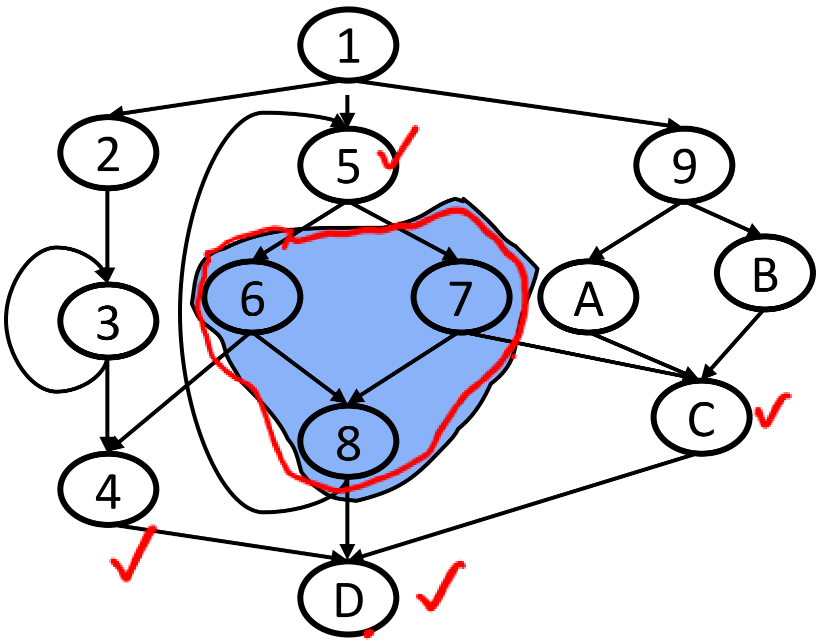
In other words, z ∈ DF(x)

Using Dominance for SSA Construction

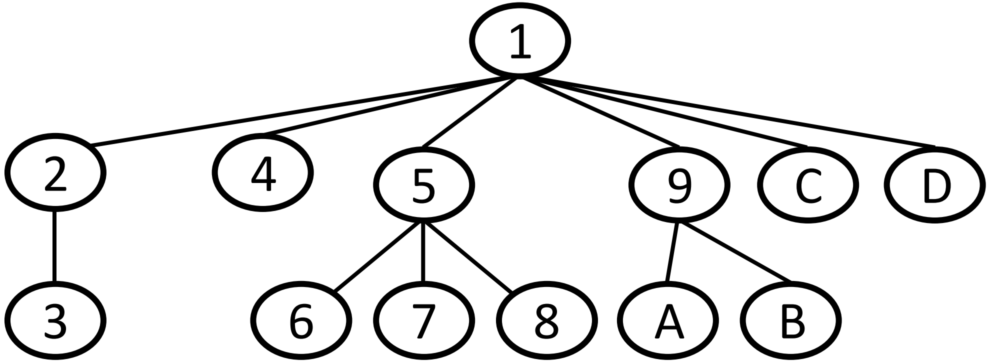
- **Dominance-Frontier Criterion**: Whenever node x contains a definition of some variable a, then any node $z \in DF(x)$, z needs a Φ-function for a.

- **Iterated dominance frontier**: since a Φ-function itself is a definition, we must iterate the dominance-frontier criterion until there are no nodes that need Φ-functions.

Dominance



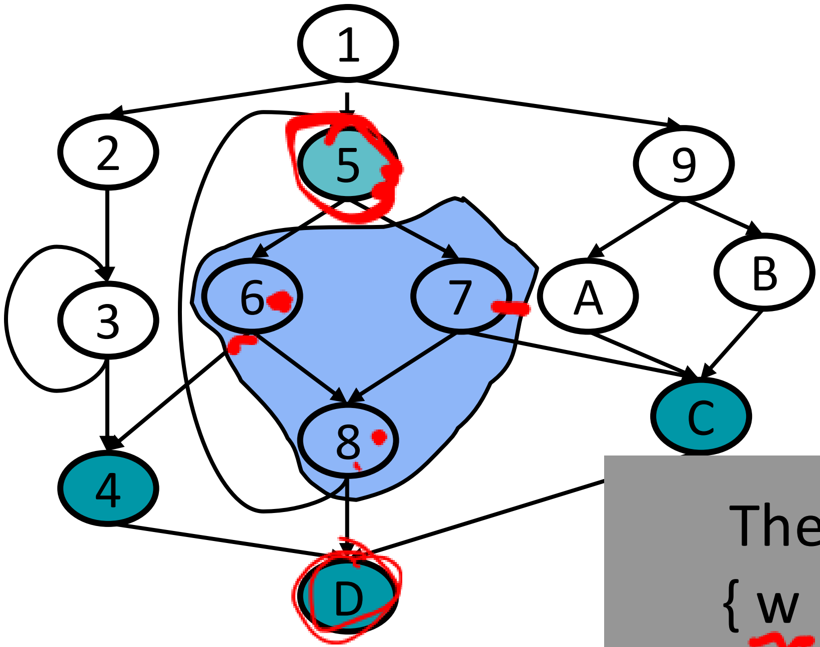
CFG



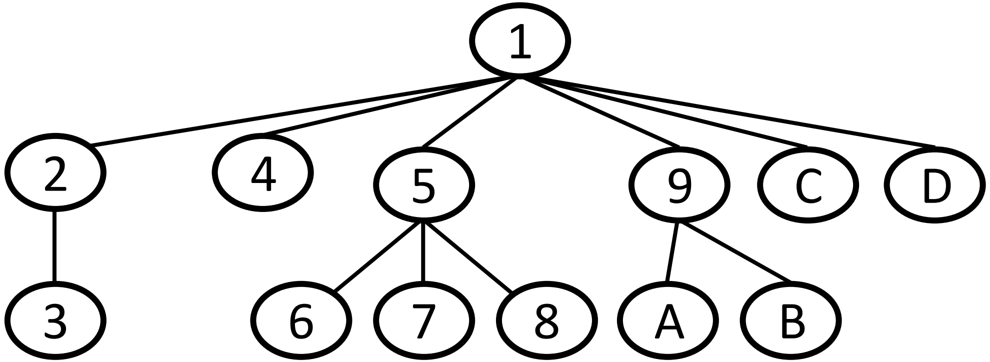
D-Tree

If there is a def of a in block 5, which nodes need a Φ()?

Dominance Frontier



CFG

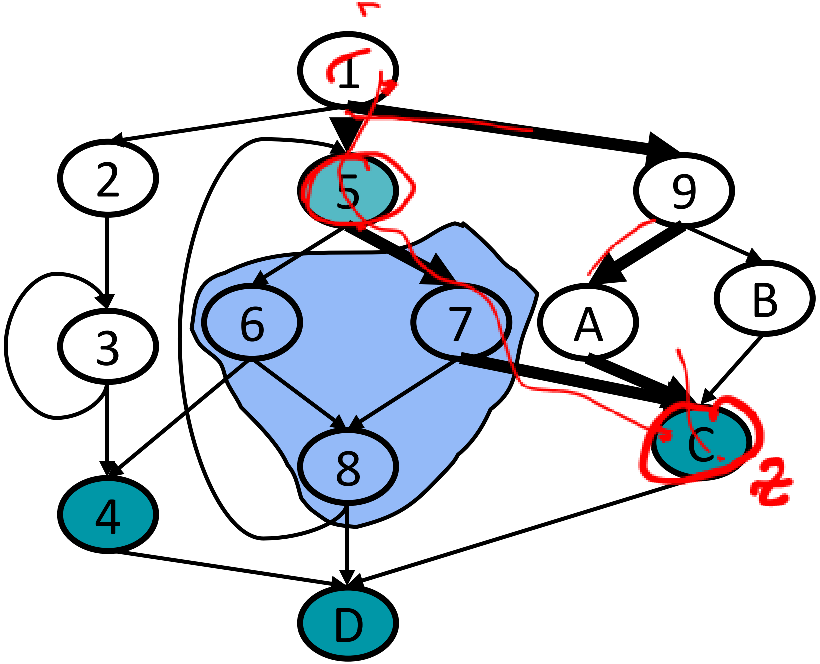


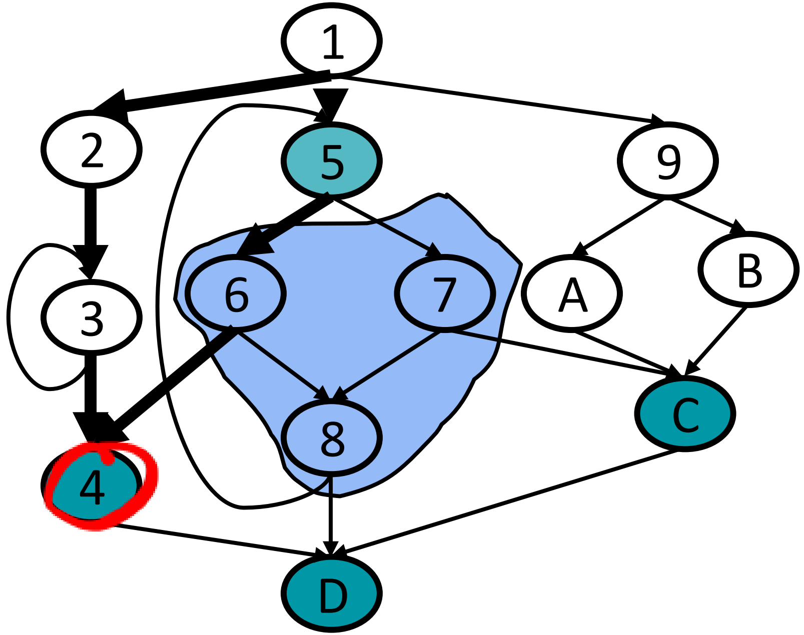
D-Tree

The dominance Frontier of a node x =
{ w | x dom pred(w) AND !(x sdom w)}

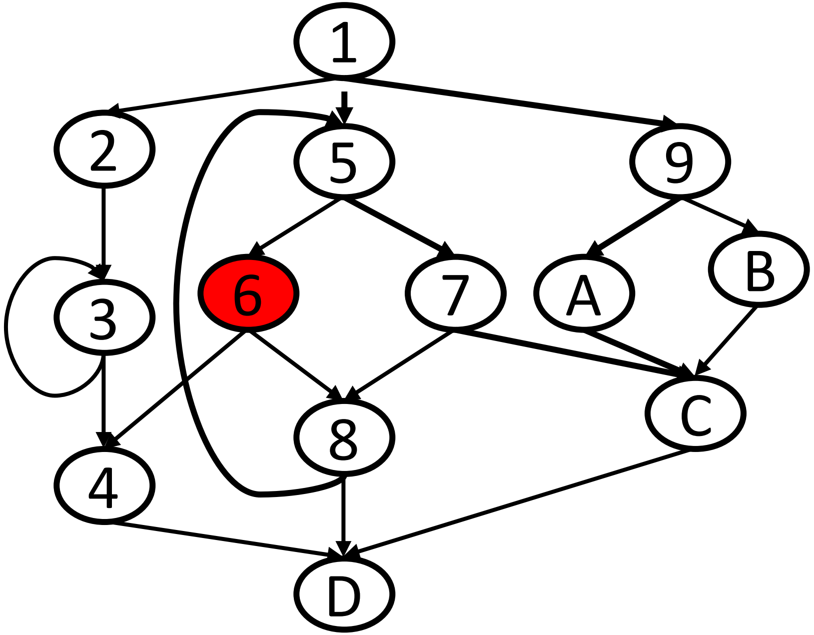
15-411/611
© 2019-21 Goldstein

Dominance Frontier & path-convergence

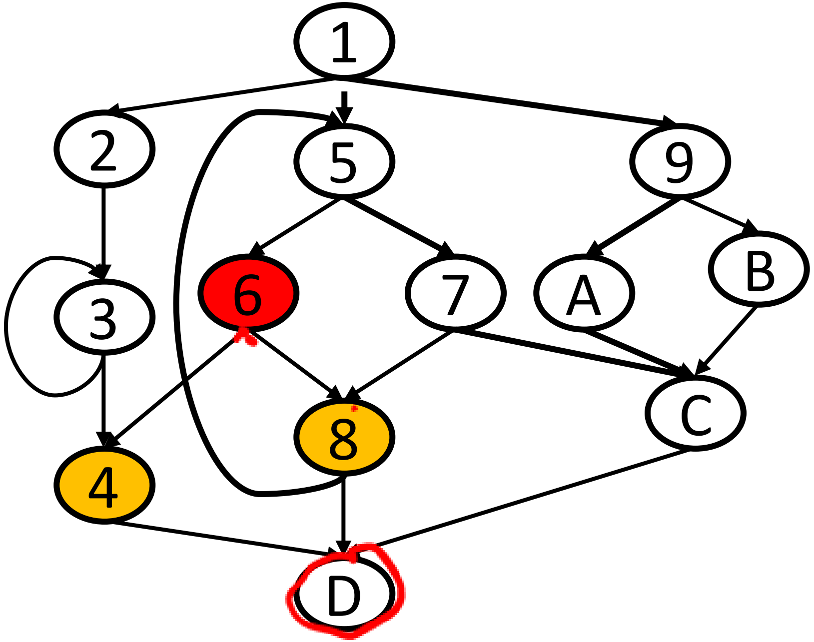




Dominance Frontier Criterion

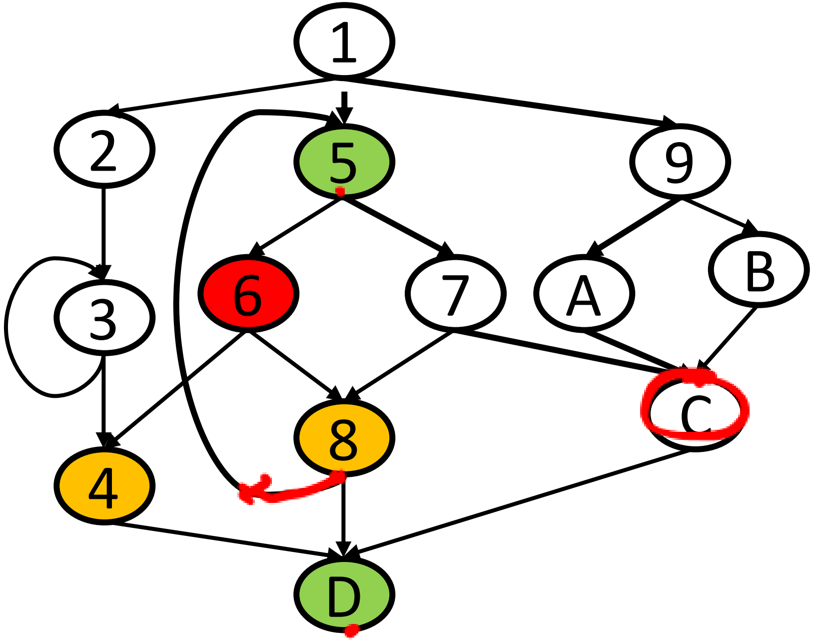


Dominance Frontier Criterion



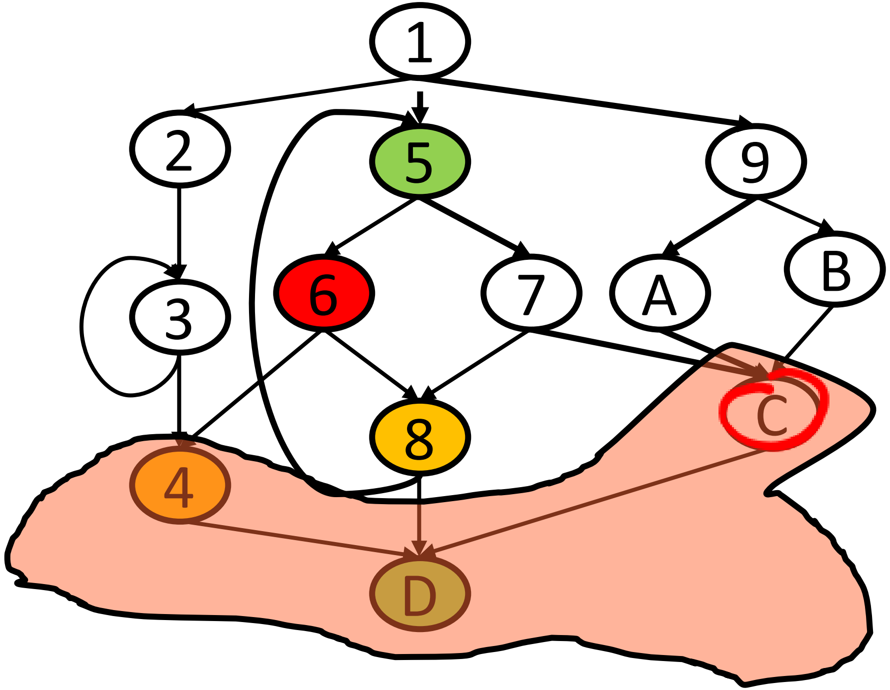
And, Iterating

Dominance Frontier Criterion



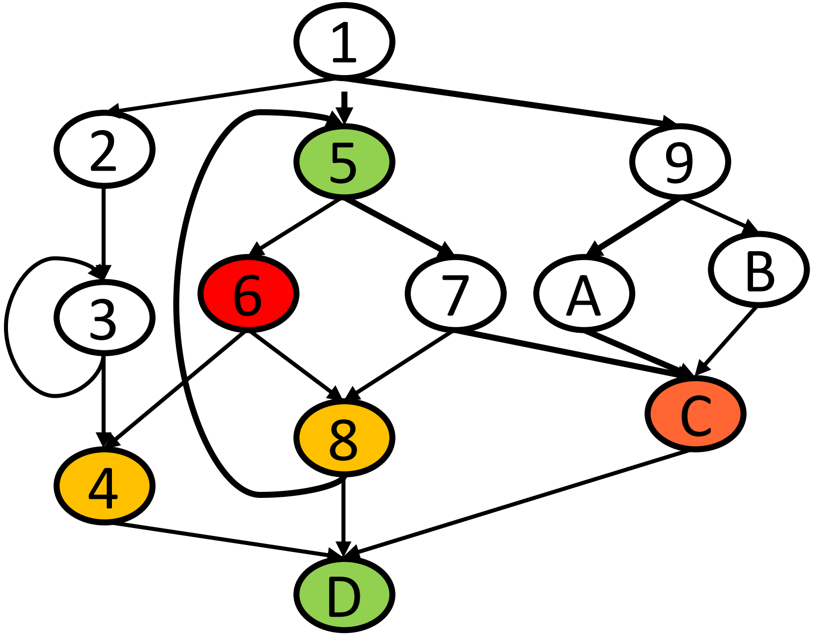
And, Iterating

Dominance Frontier Criterion



And, Iterating

Dominance Frontier Criterion



Done

Using DF to Place $\Phi()$

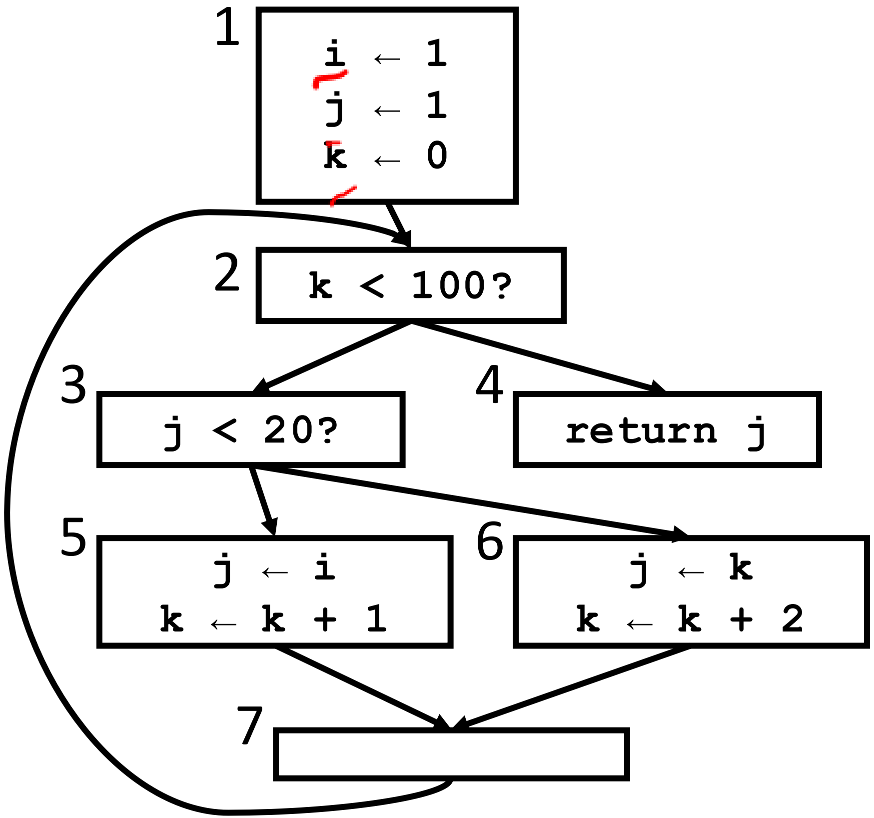
- Gather all the defsites of every variable
- Then, for every variable
  - foreach defsite
    - foreach node in DF(defsite)
      - if we haven't put $\Phi()$ in node put one in
      - If this node didn't define the variable before: add this node to the defsites

- This essentially computes the Iterated Dominance Frontier on the fly, creating minimal SSA

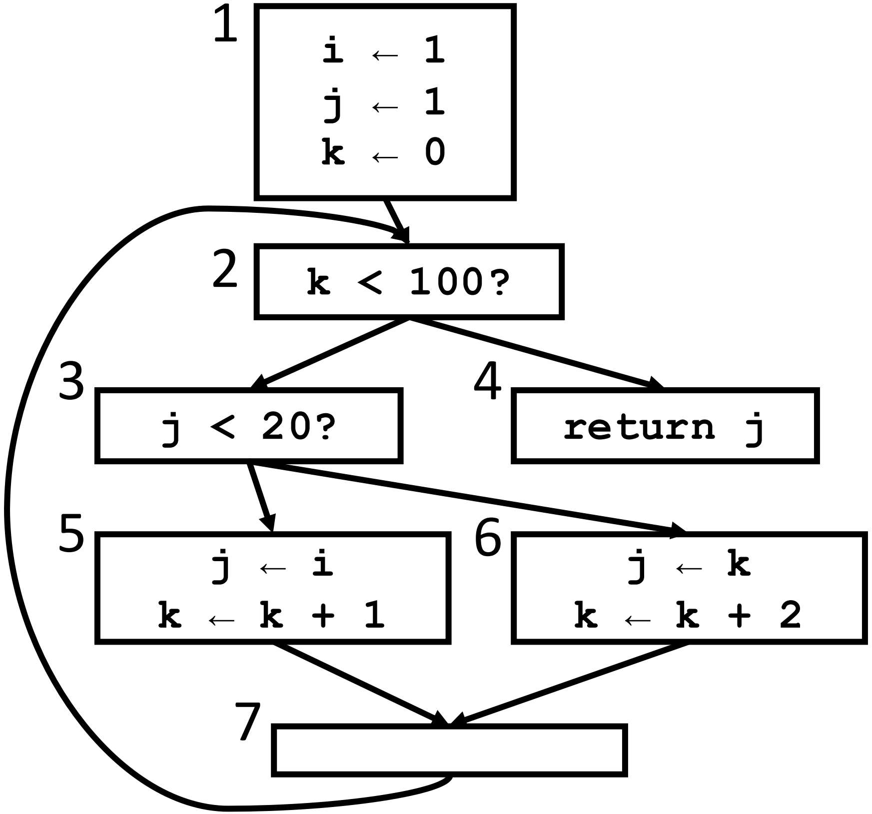
Using DF to Place Φ()

```
foreach node n {
    foreach variable v defined in n {
        orig[n] U= {v}
        defsites[v] U= {n}
    }
    foreach variable v {
        W = defsites[v]
        while W not empty {
            foreach y in DF[n]
            if y ∉ PHI[v] {
                insert "v ← Φ(v,v,…)" at top of y
                PHI[v] = PHI[v] U {y}
                if v ∉ orig[y]: W = W U {y}
            }
        }
    }
}
```

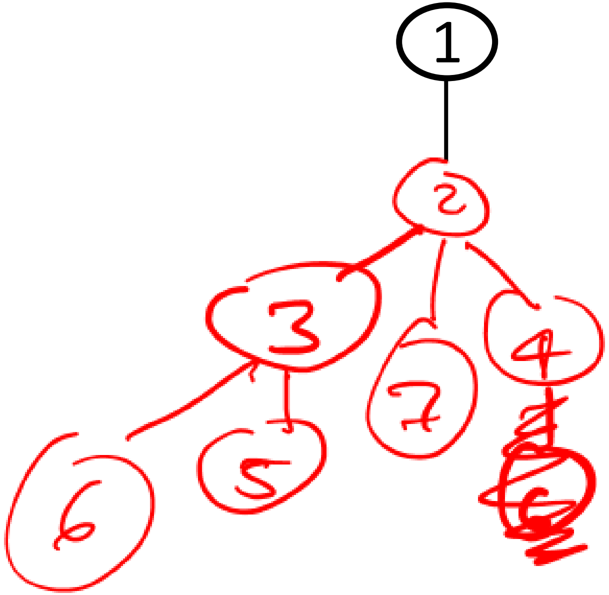
Computing SSA



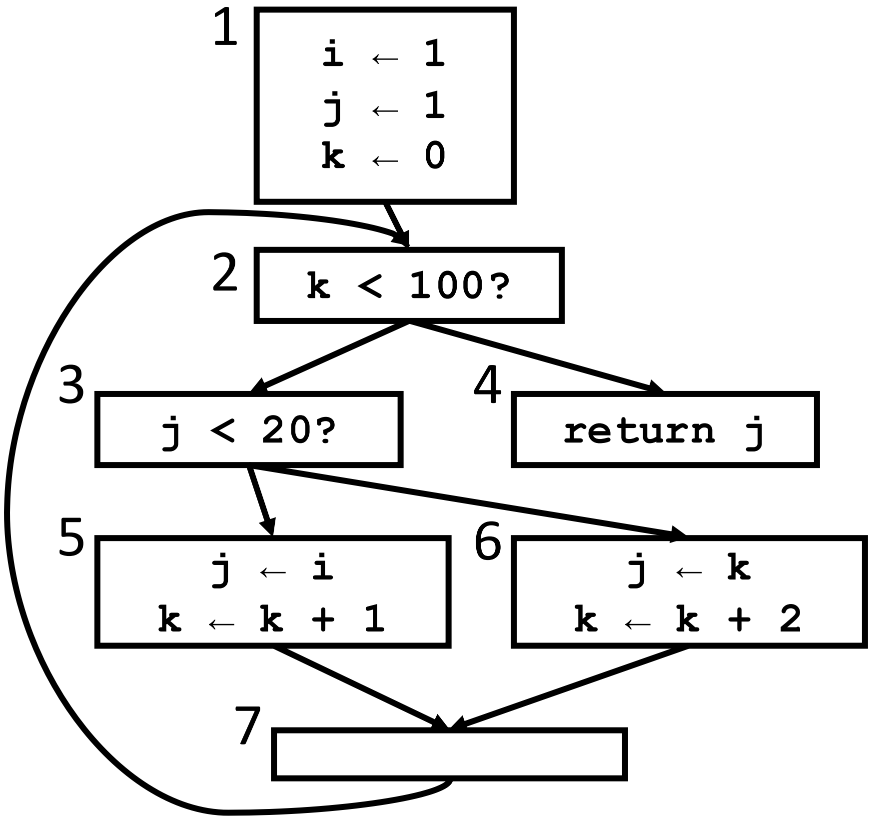
Compute D-tree

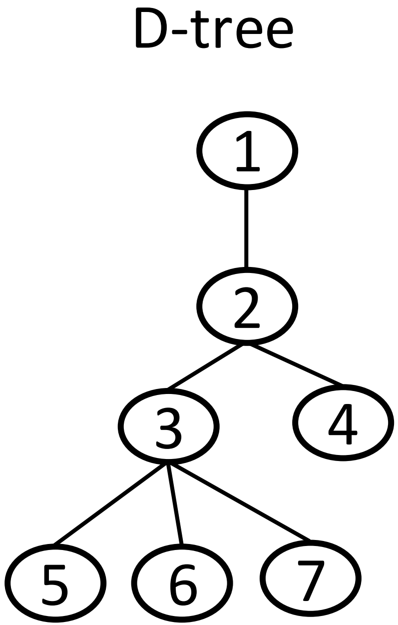


D-tree

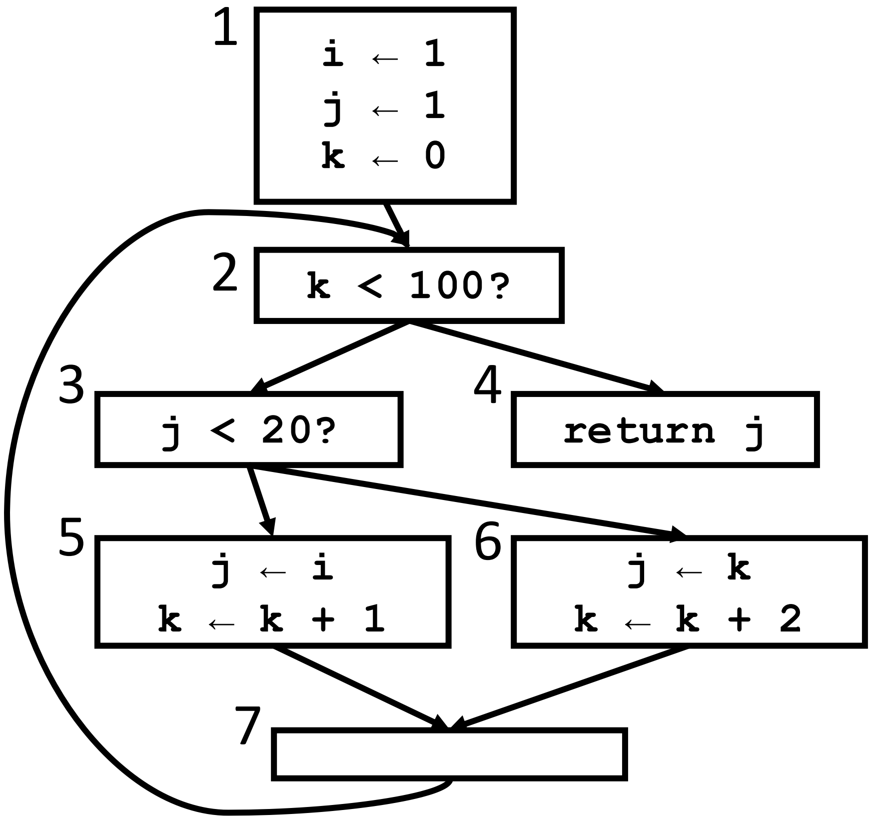


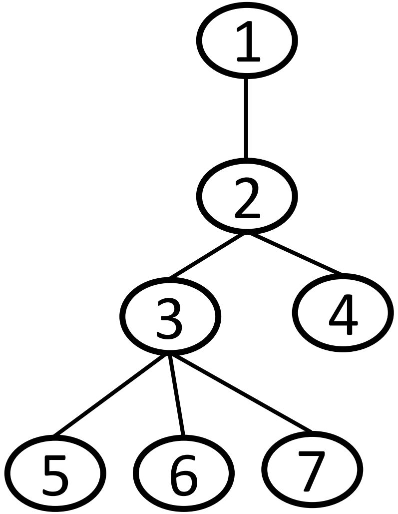
Compute D-tree





Compute Dominance Frontier (DFs)





DFs

1

2

3

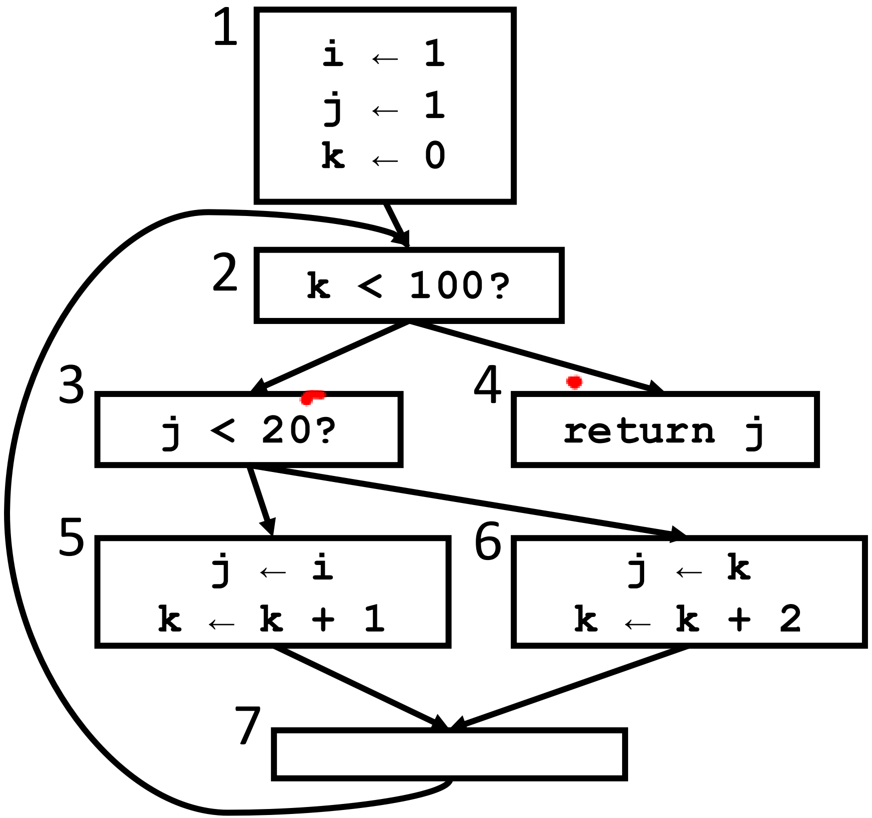
4

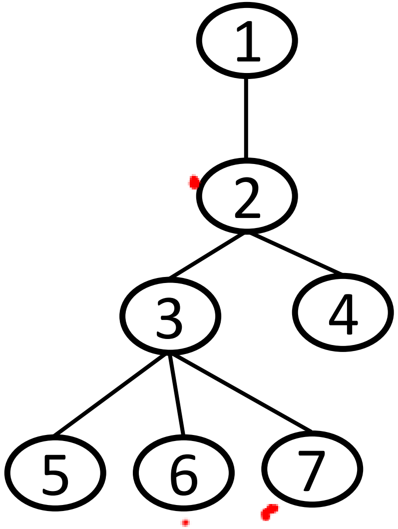
5

6

7

Compute Dominance Frontier (DFs)

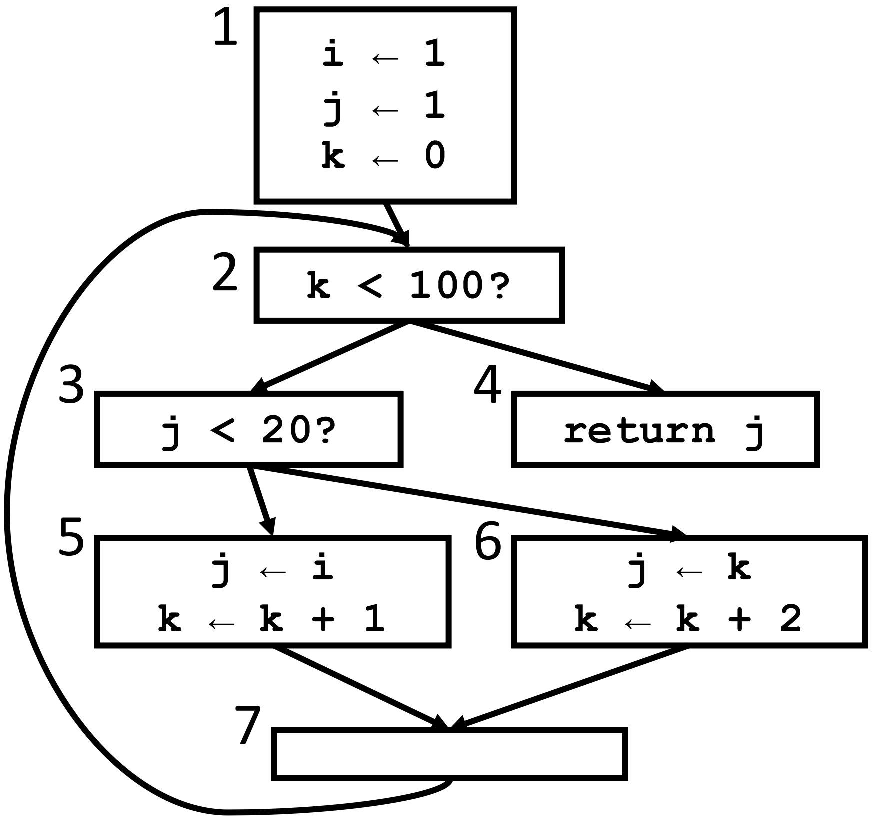




DFs

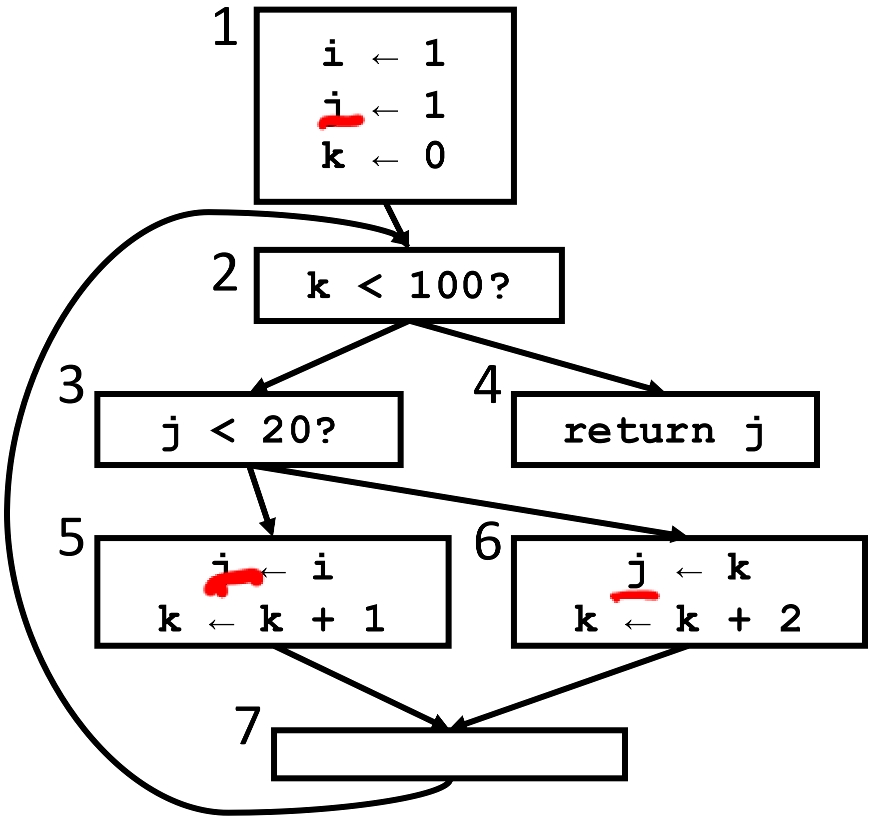
1	{}
2	{2}
3	{2}
4	{}
5	{7}
6	{7}
7	{2}

Compute defsites



DFs		orig[n]		defsites[v]	
1	{}	1	{ i,j,k}	i	{1}
2	{2}	2	{}	j	{1,5,6}
3	{2}	3	{}	k	{1,5,6}
4	{}	4	{}		
5	{7}	5	{j,k}		
6	{7}	6	{j,k}		
7	{2}	7	{}		

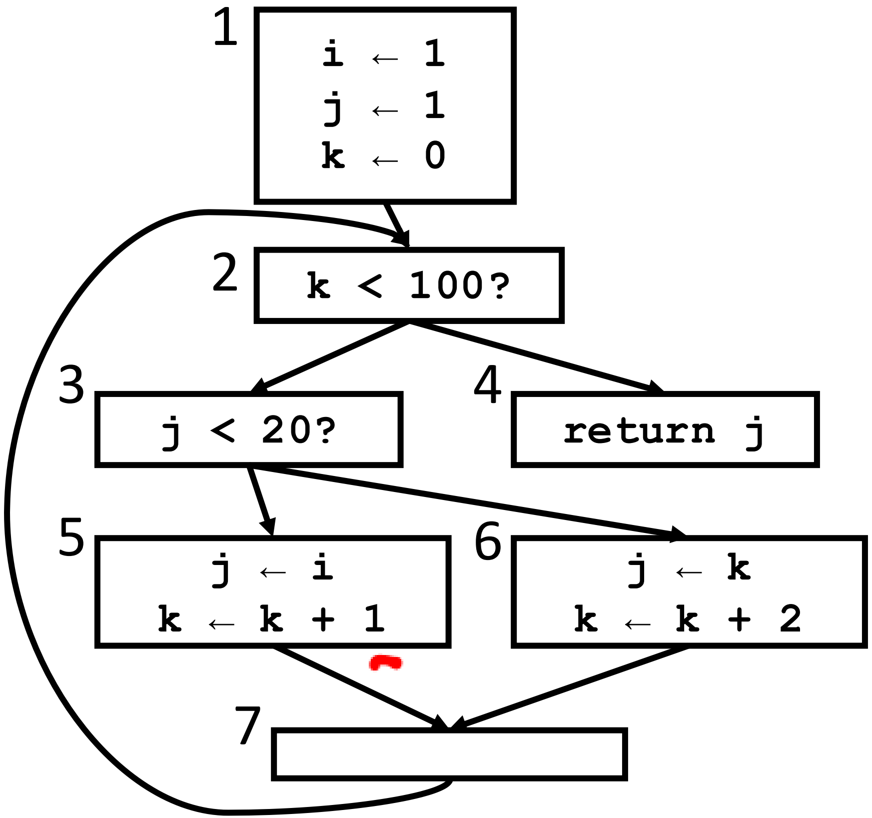
Inspect variables



DFs		orig[n]		defsites[v]	
1	{}	1	{ i,j,k}	i	{1}
2	{2}	2	{}	j	{1,5,6}
3	{2}	3	{}	k	{1,5,6}
4	{}	4	{}		
5	{7}	5	{j,k}		
6	{7}	6	{j,k}		
7	{2}	7	{}		

var j: W={1,5,6}

Insert ϕ for j



DFs	
1	{}
2	{2}
3	{2}
4	{}
5	{7}
6	{7}
7	{2}

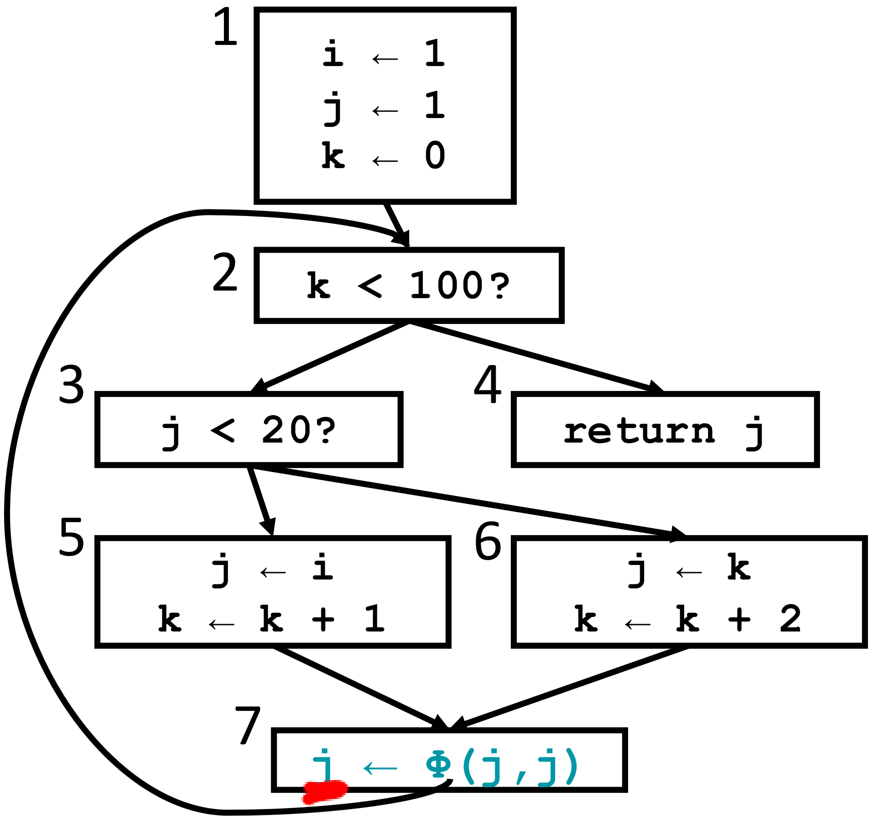
orig[n]	
1	{ i,j,k}
2	{}
3	{}
4	{}
5	{j,k}
6	{j,k}
7	{}

defsites[v]	
i	{1}
j	{1,5,6}
k	{1,5,6}

var j: W={1,5,6}

DF[1] U DF[5] U DF[6] ={7}

Insert ϕ for j



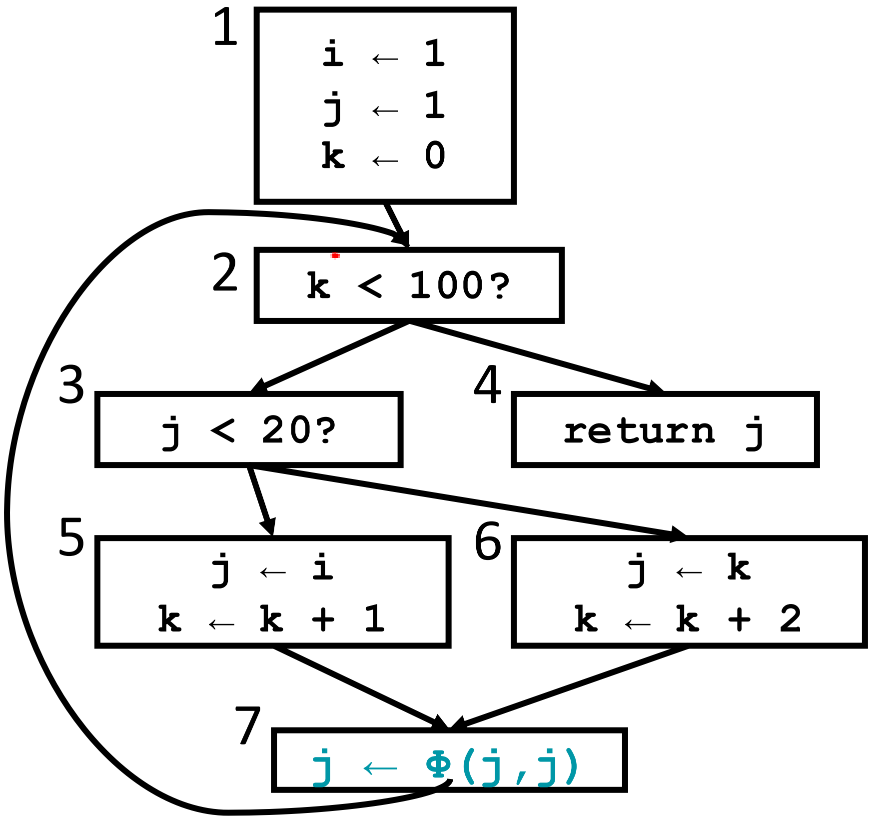
DFs	
1	{}
2	{2}
3	{2}
4	{}
5	{7}
6	{7}
7	{2}

orig[n]	
1	{ i,j,k}
2	{}
3	{}
4	{}
5	{j,k}
6	{j,k}
7	{}

defsites[v]	
i	{1}
j	{1,5,6}
k	{1,5,6}

var j: W={1,5,6}

Handle new write for j

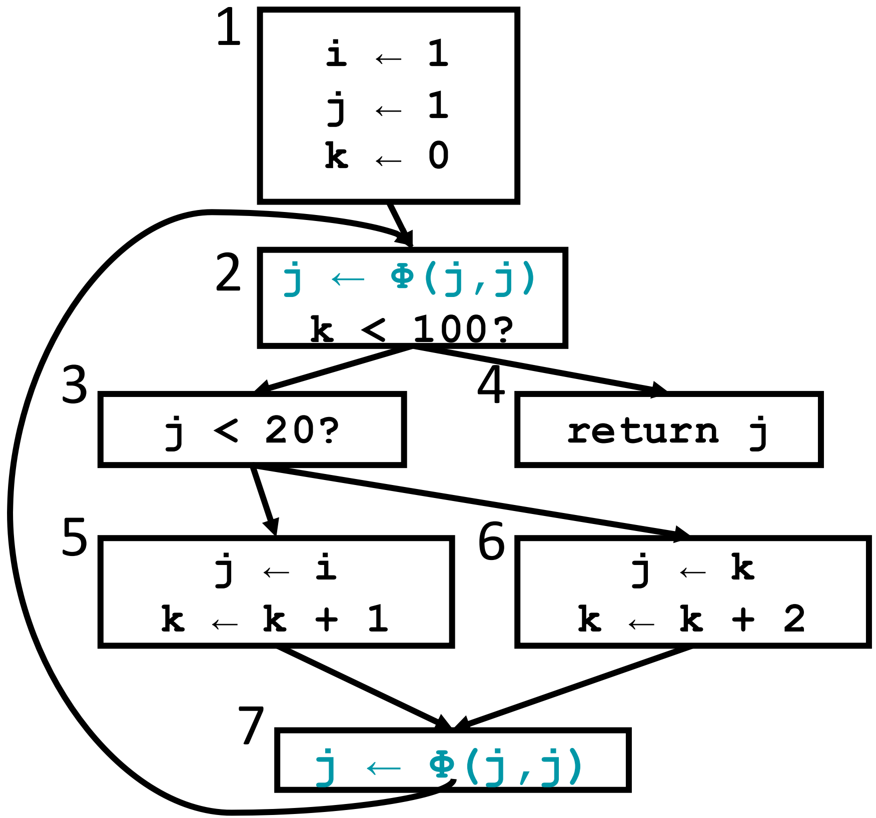


DFs		orig[n]		defsites[v]	
1	{}	1	{ i,j,k}	i	{1}
2	{2}	2	{}	j	{1,5,6}
3	{2}	3	{}	k	{1,5,6}
4	{}	4	{}		
5	{7}	5	{j,k}		
6	{7}	6	{j,k}		
7	{2}	7	{}		

var j: W={1,5,6,7}

DF[1] ∪ DF[5] ∪ DF[6] ∪ DF[7] ={7,2}

Insert more ϕ for j

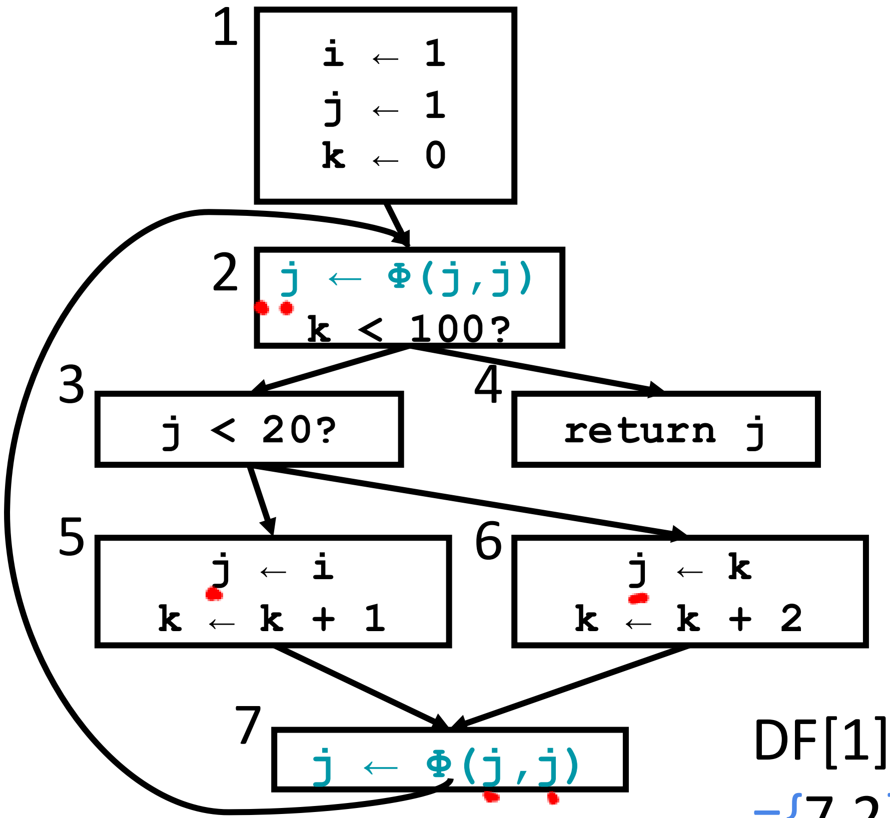


DFs		orig[n]		defsites[v]	
1	{}	1	{ i,j,k}	i	{1}
2	{2}	2	{}	j	{1,5,6}
3	{2}	3	{}	k	{1,5,6}
4	{}	4	{}		
5	{7}	5	{j,k}		
6	{7}	6	{j,k}		
7	{2}	7	{}		

var j: W={1,5,6,7}

DF[1] ∪ DF[5] ∪ DF[6] ∪ DF[7] ={7,2}

Update writes for j



DFs		orig[n]		defsites[v]	
1	{}	1	{ i,j,k}	i	{1}
2	{2}	2	{}	j	{1,5,6}
3	{2}	3	{}	k	{1,5,6}
4	{}	4	{}		
5	{7}	5	{j,k}		
6	{7}	6	{j,k}		
7	{2}	7	{}		

var j: W={1,5,6,7,2}

DF[1] U DF[5] U DF[6] U DF[7] U DF[2]

={7,2}

Renaming Variables

- Placing ϕ is not enough, need to update names
- Walk down the dominator tree, renaming variables incrementally
- Replace uses with most recent renamed def
  - For straight-line code this is easy
  - If there are branches and joins?

Renaming for Straight-Line Code

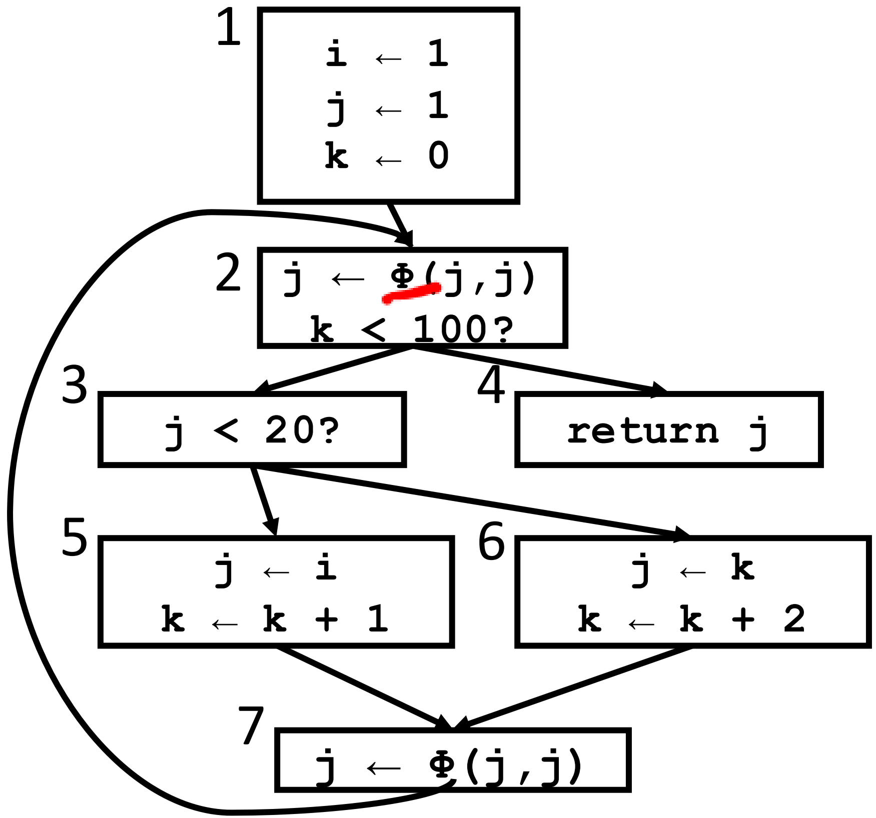
- Need to extend for ϕ-functions.
- Need to maintain property that definitions dominate uses.

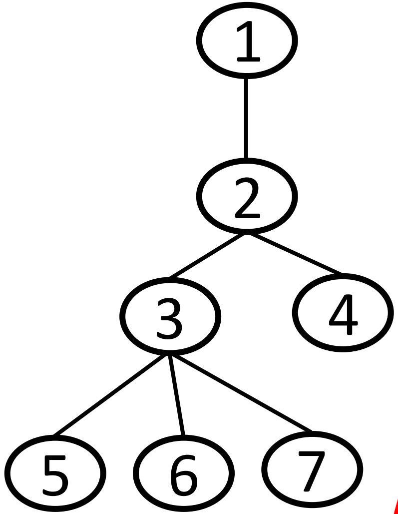
```
for each variable a:
   Count[a] = 0
   Stack[a] = [0]
renameBasicBlock(B):
   for each instruction S in block B:
      for each use of a variable x in S:
         i = top(Stack[x])
         replace the use of x with x_i
      for each variable a that S defines
         count[a] = Count[a] + 1
         i = Count[a]
         push i onto Stack[a]
         replace definition of a with a_i
```

Renaming in CFG

```
rename(n):
    renameBasicBlock(n)
    for each successor Y of n, where n is the jth predecessor of Y:
    for each phi-function f in Y, where the operand of f is 'a'
    i = top(Stack[a])
    replace jth operand with a_i
    for each child of n in D-tree, X:
        rename(X)
    for each instruction S ∈ n:
        for each variable v that S defines:
            pop Stack[v]
```

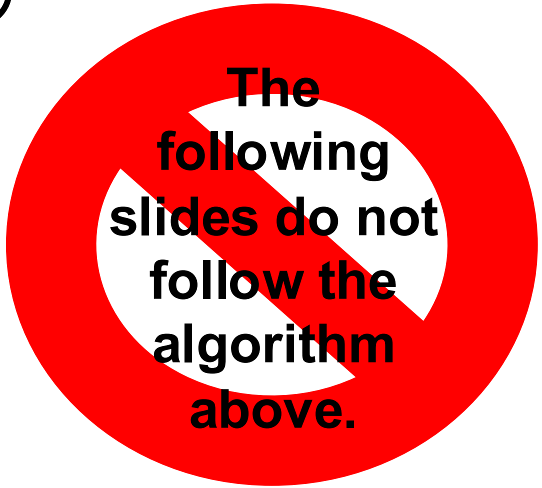
Rename j variables



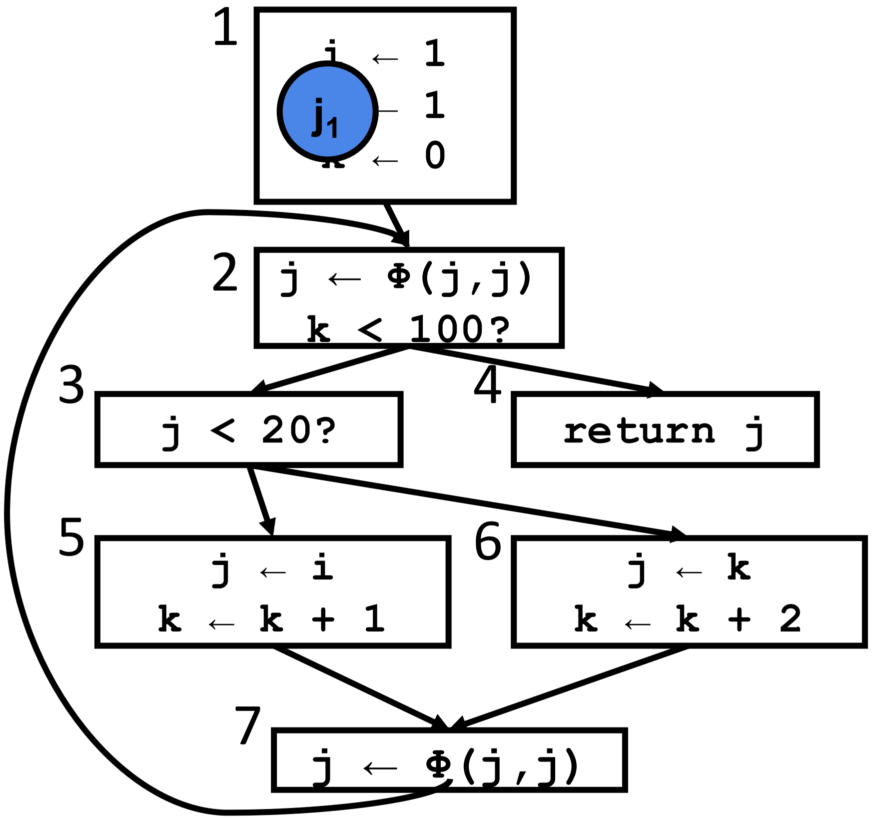


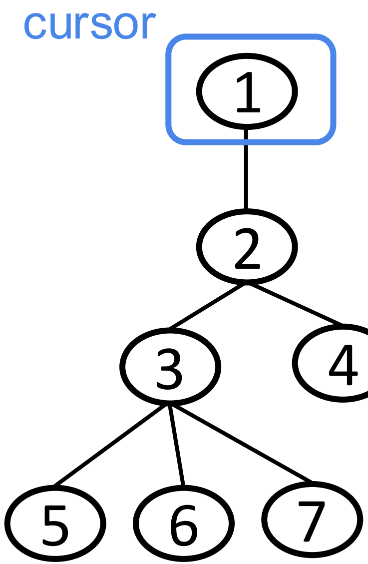
defsites[v]

i	{1}
j	{1,5,6,7,2}
k	{1,5,6}



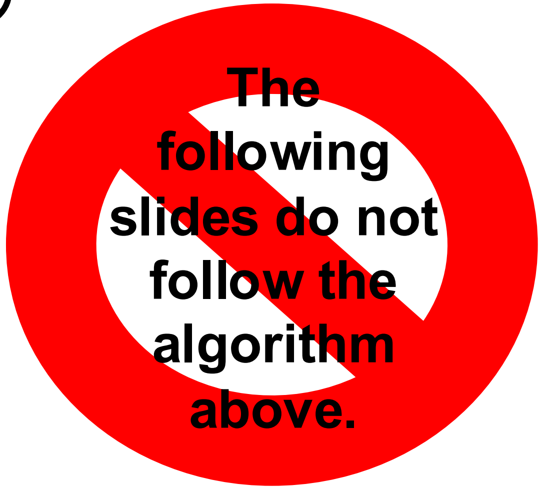
Rename j variables



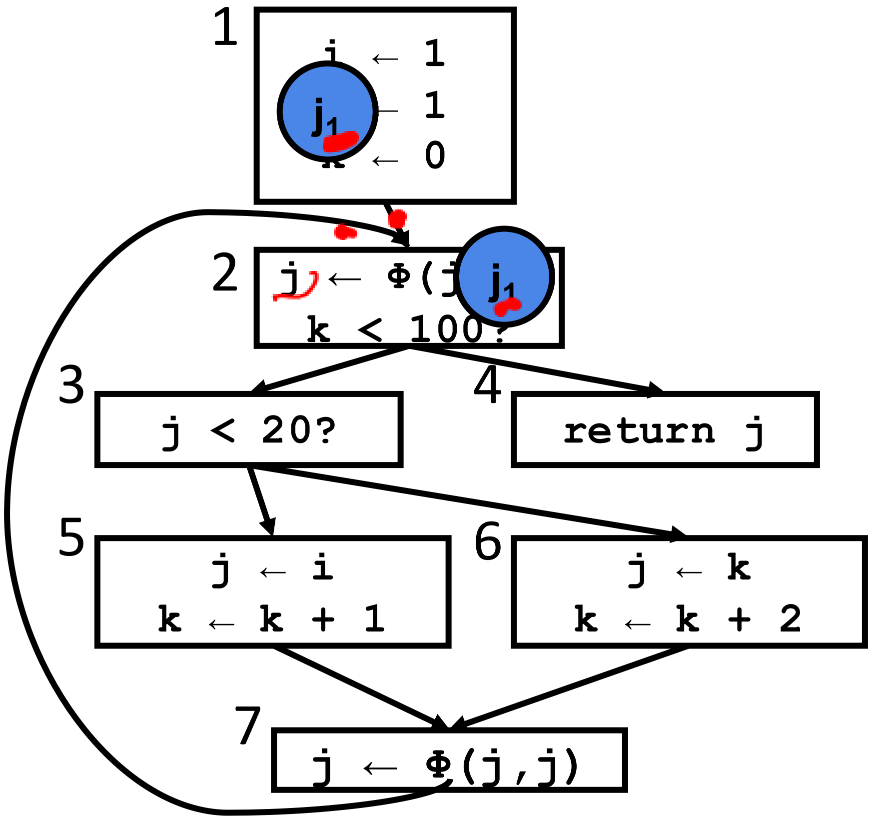


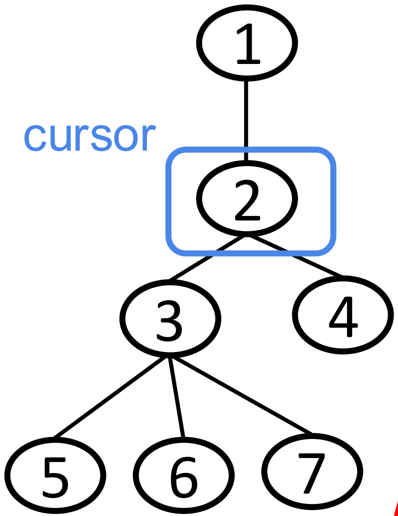
defsites[v]

i	{1}
j	{1,5,6,7,2}
k	{1,5,6}



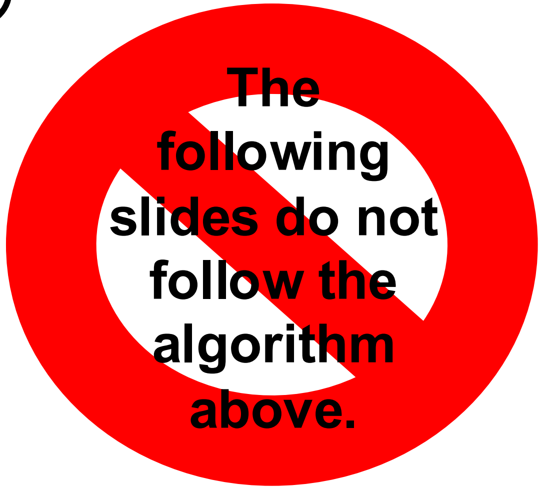
Rename j variables



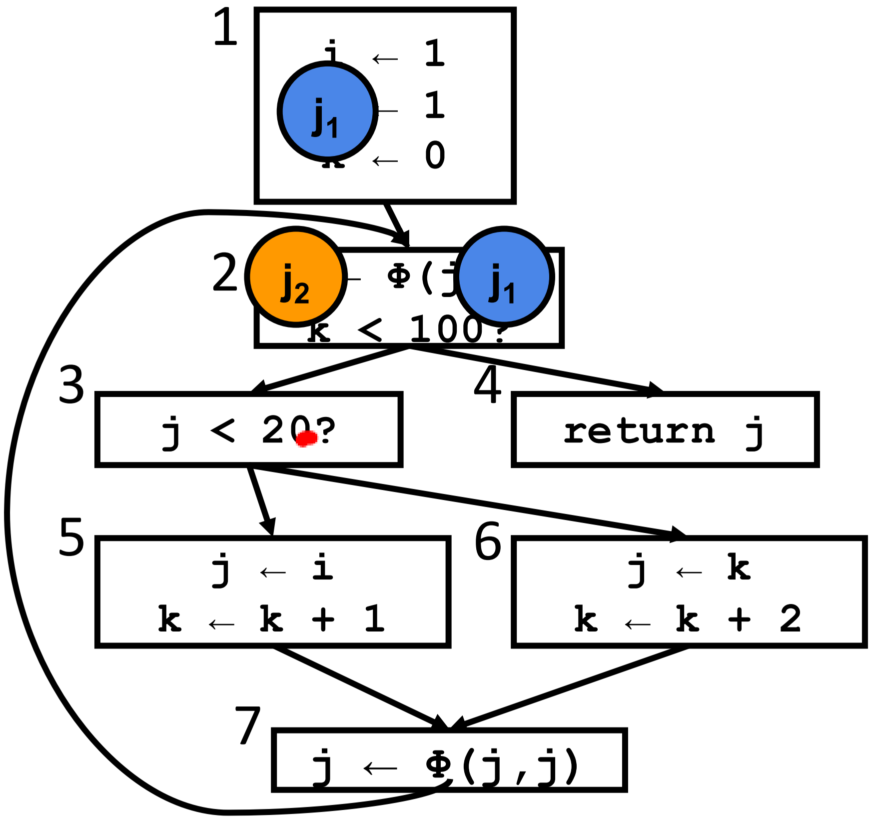


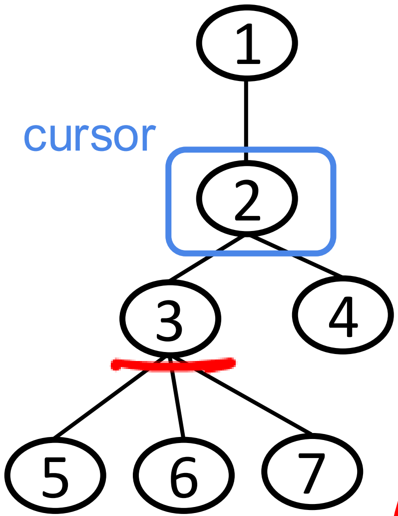
defsites[v]

i	{1}
j	{1,5,6,7,2}
k	{1,5,6}



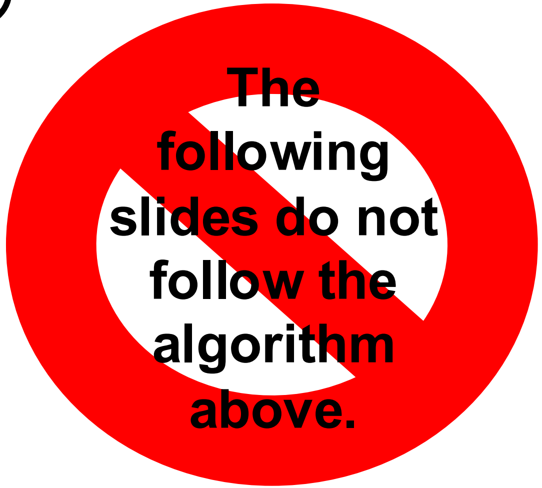
Rename j variables



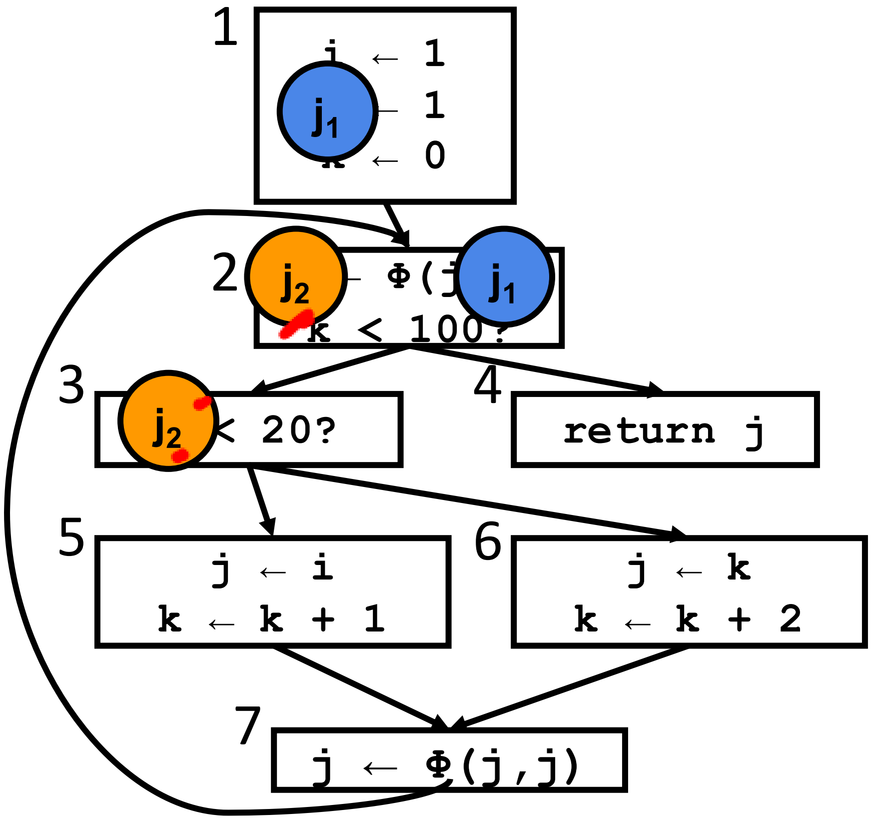


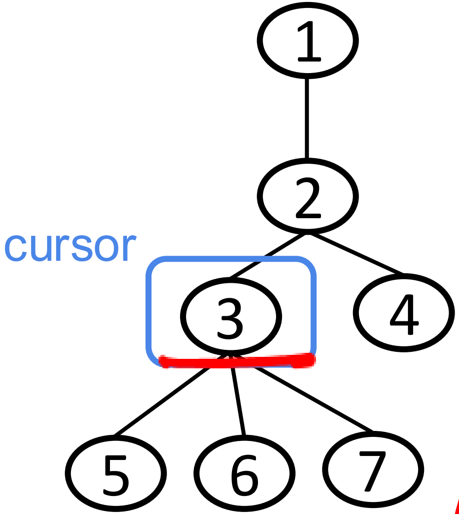
defsites[v]

i	{1}
j	{1,5,6,7,2}
k	{1,5,6}



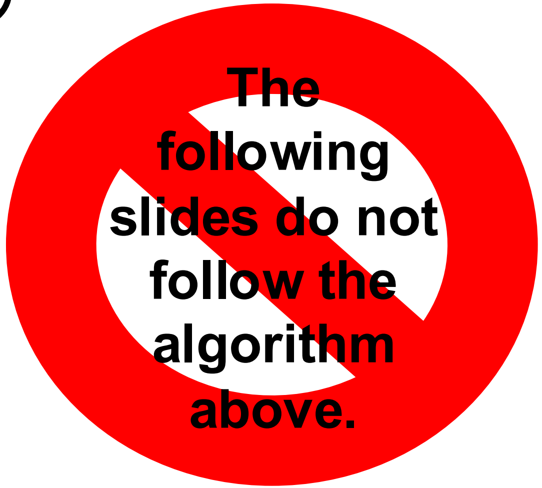
Rename j variables



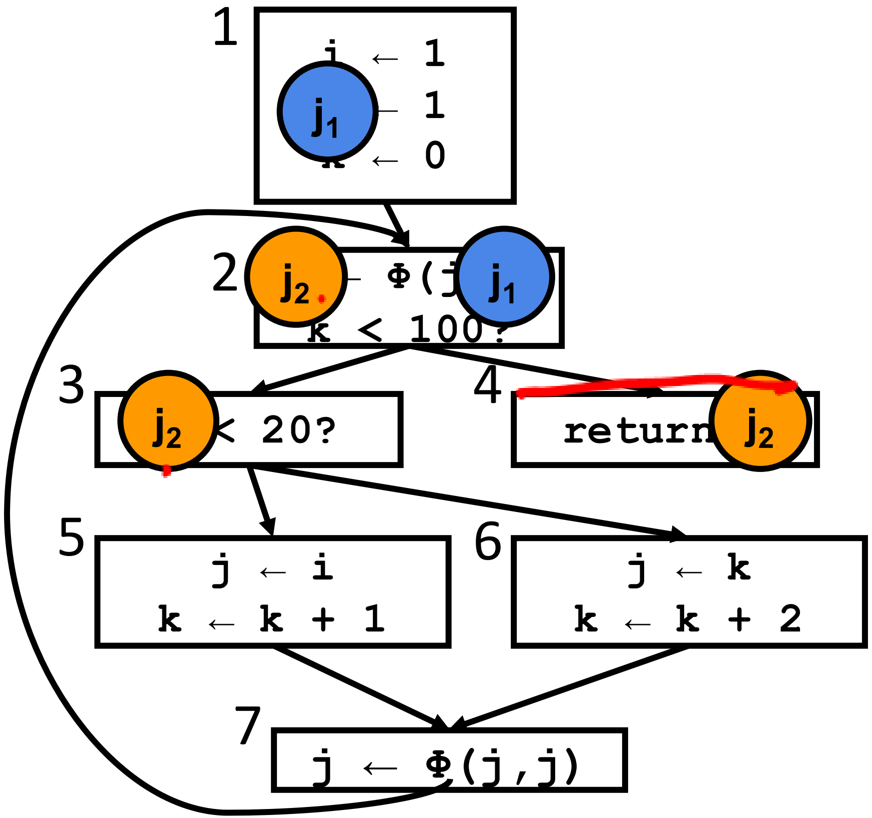


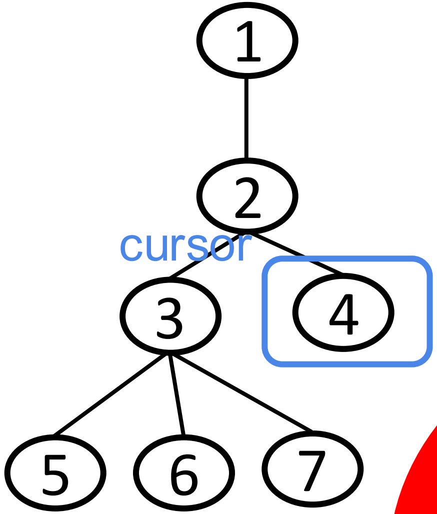
defsites[v]

i	{1}
j	{1,5,6,7,2}
k	{1,5,6}



Rename j variables



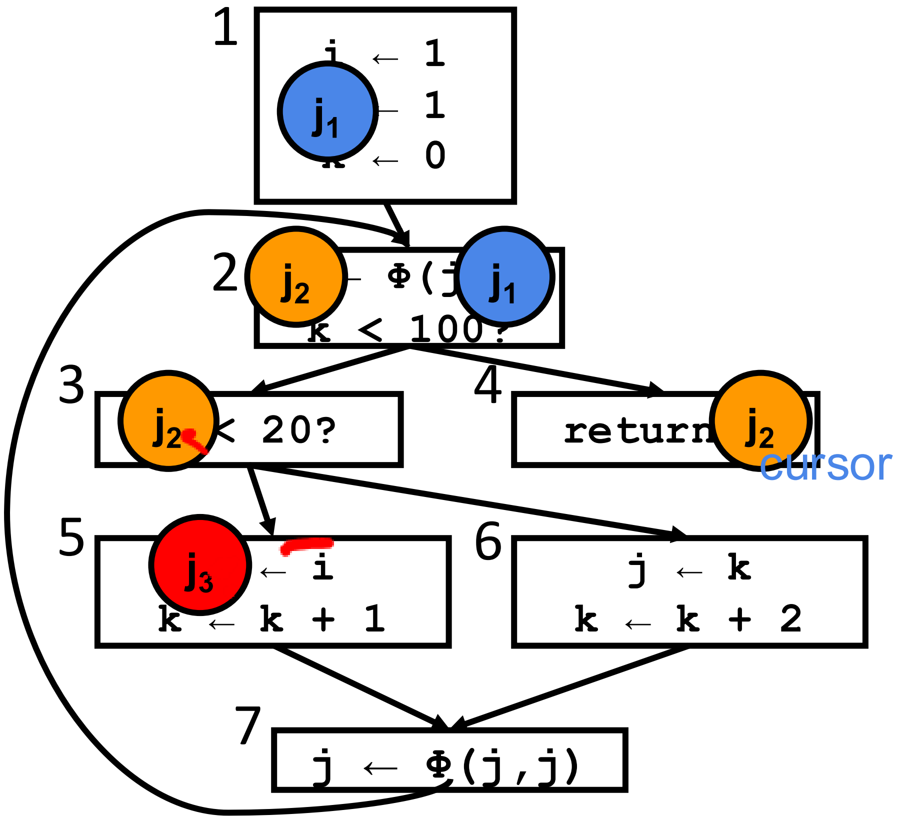


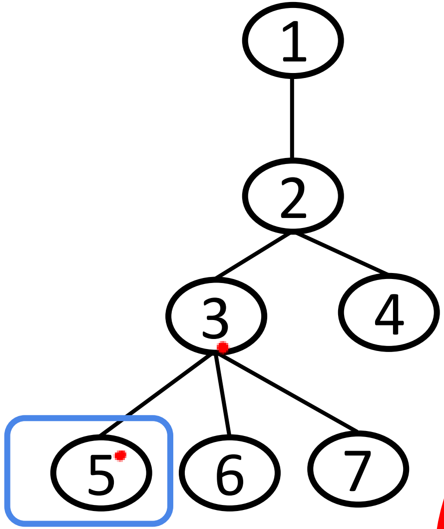
defsites[v]

i	{1}
j	{1,5,6,7,2}
k	{1,5,6}

The following slides do not follow the algorithm above.

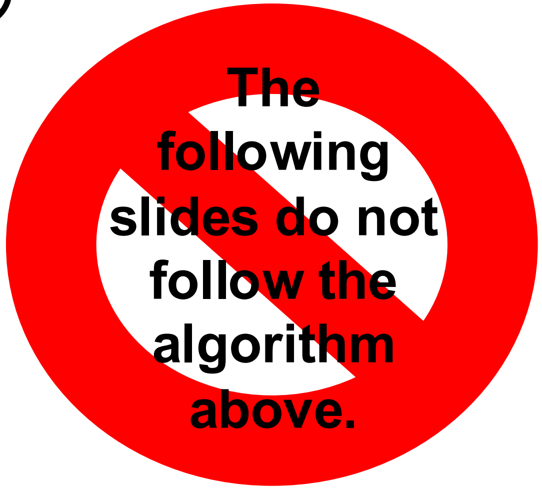
Rename j variables



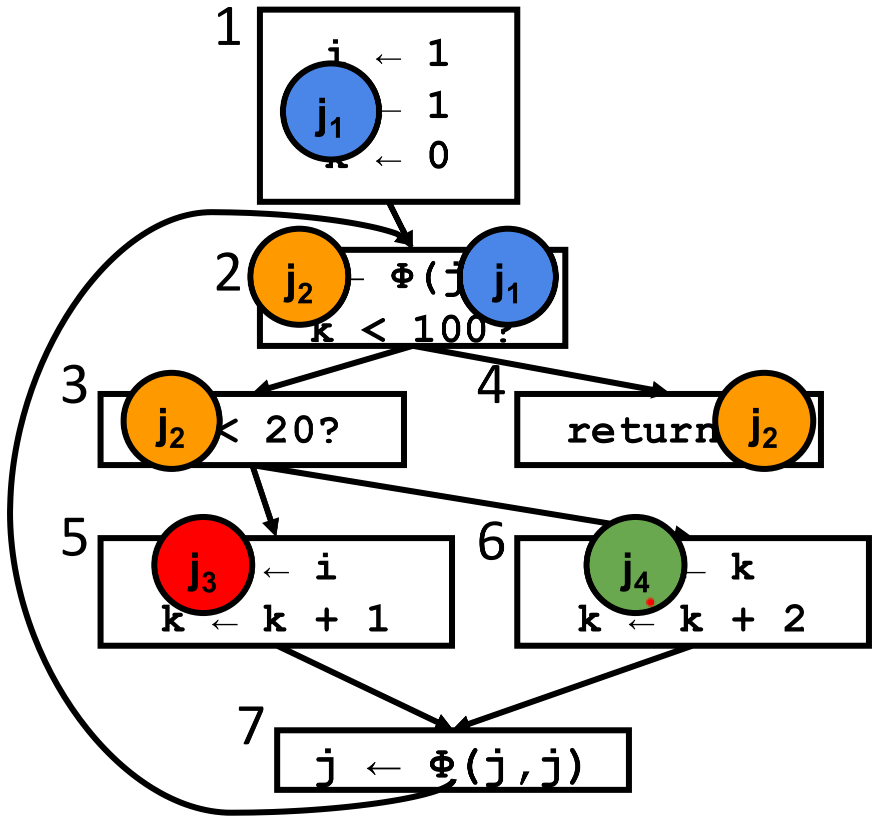


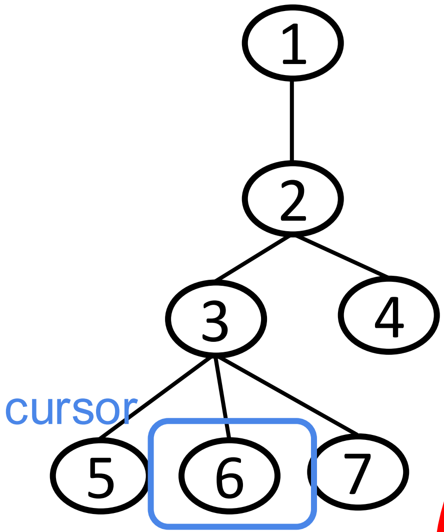
defsites[v]

i	{1}
j	{1,5,6,7,2}
k	{1,5,6}



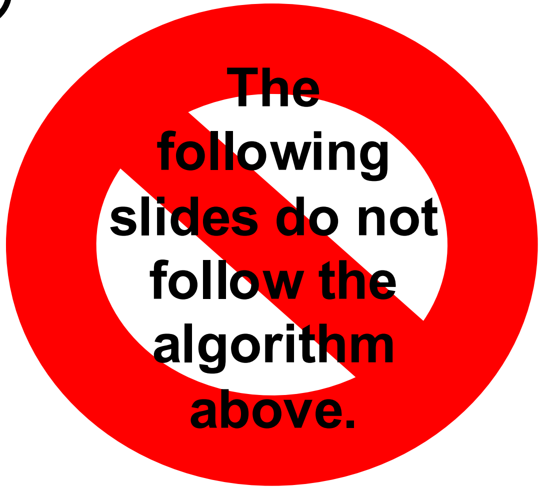
Rename j variables





defsites[v]

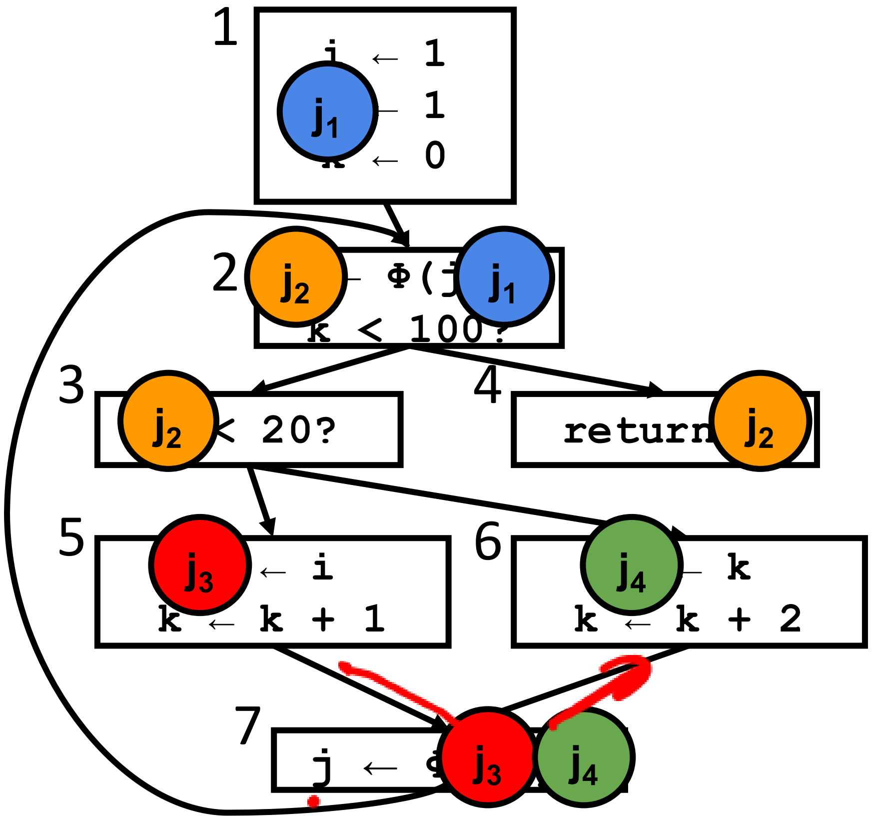
i	{1}
j	{1,5,6,7,2}
k	{1,5,6}

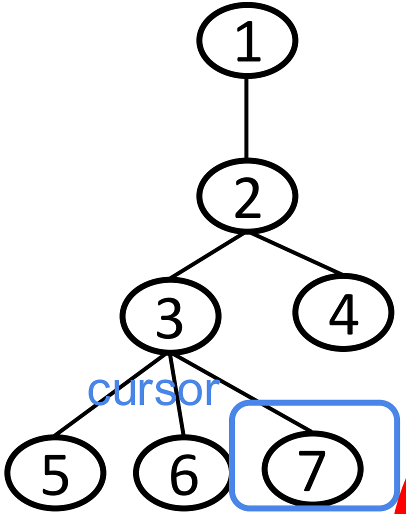


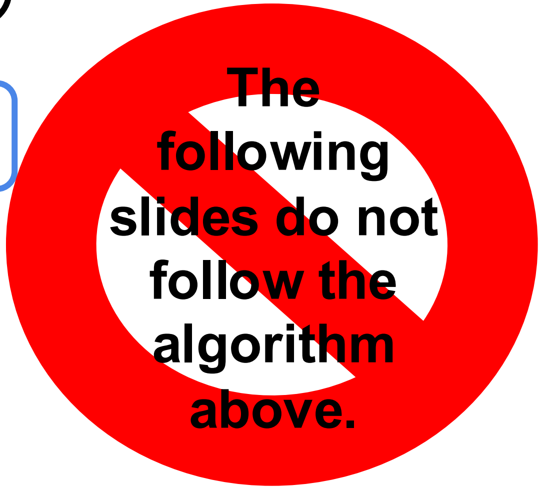
Rename j variables

defsites[v]

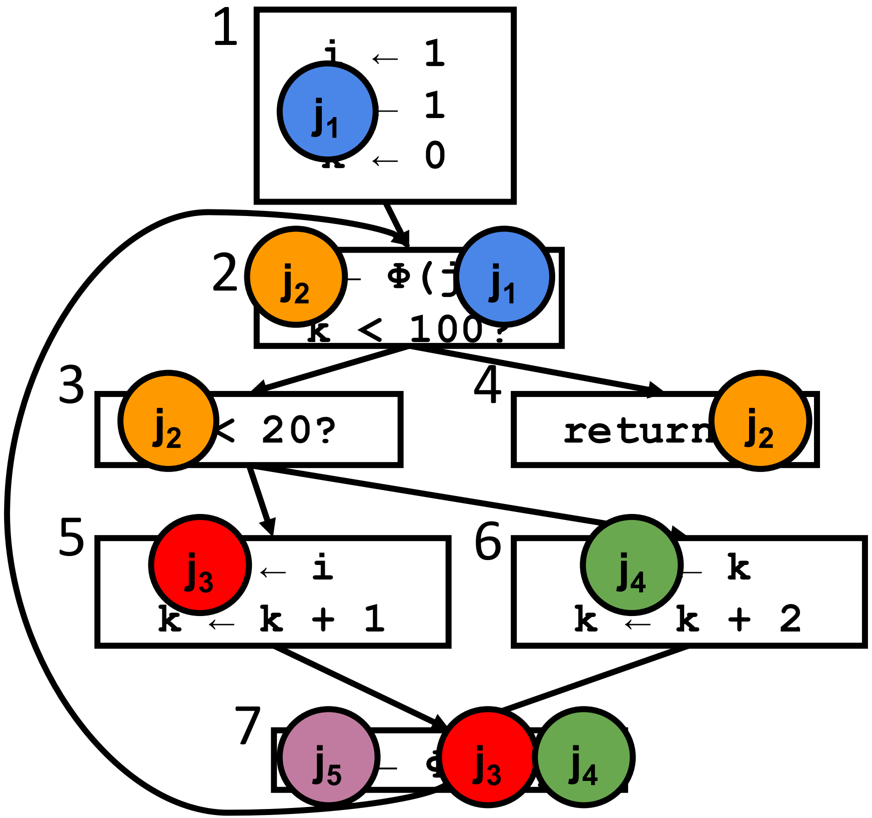
i	{1}
j	{1,5,6,7,2}
k	{1,5,6}

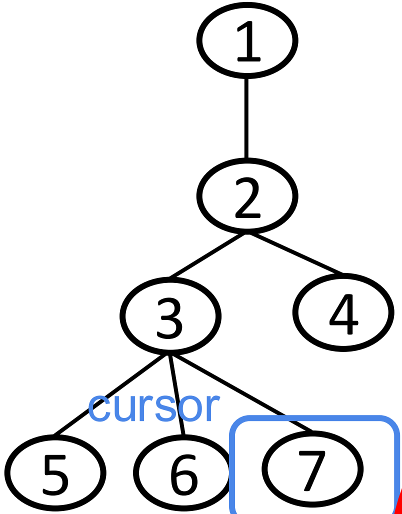






Rename j variables



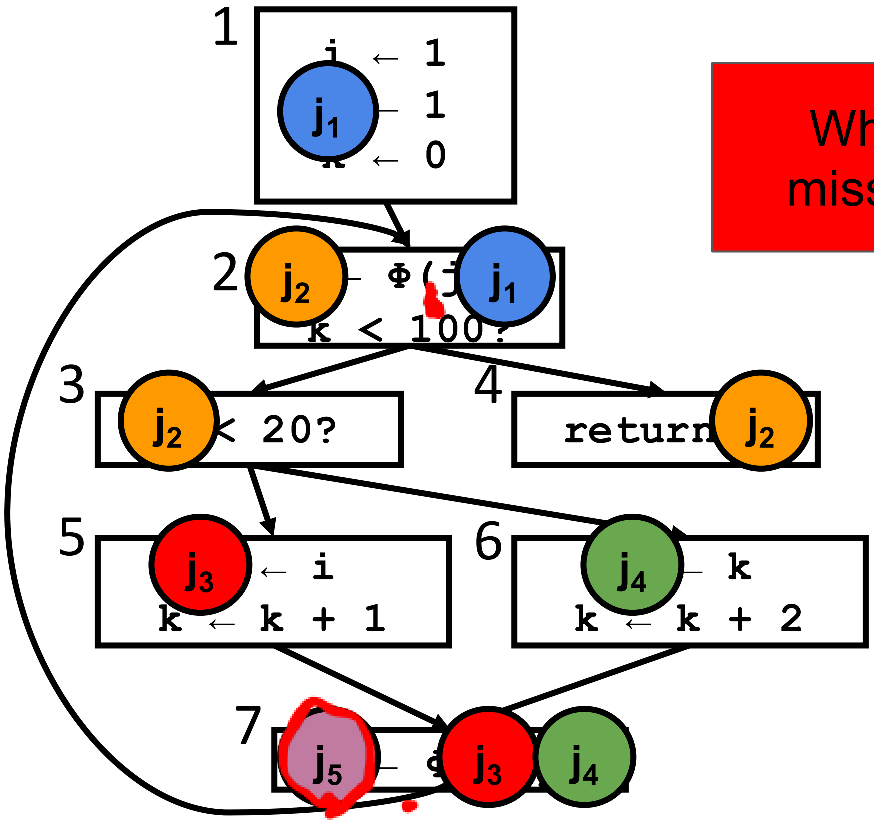


defsites[v]

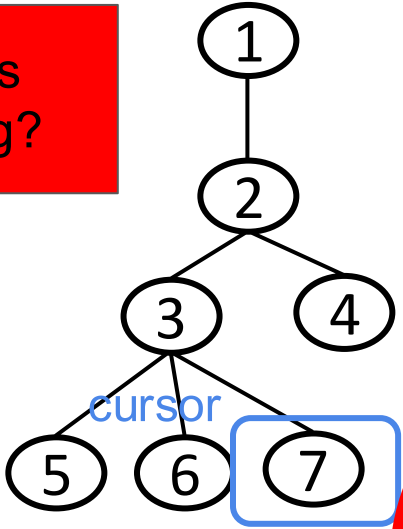
i	{1}
j	{1,5,6,7,2}
k	{1,5,6}

The following slides do not follow the algorithm above.

Rename j variables

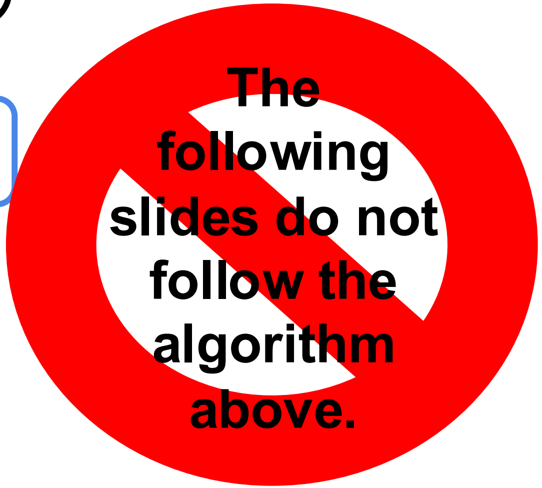


What's missing?

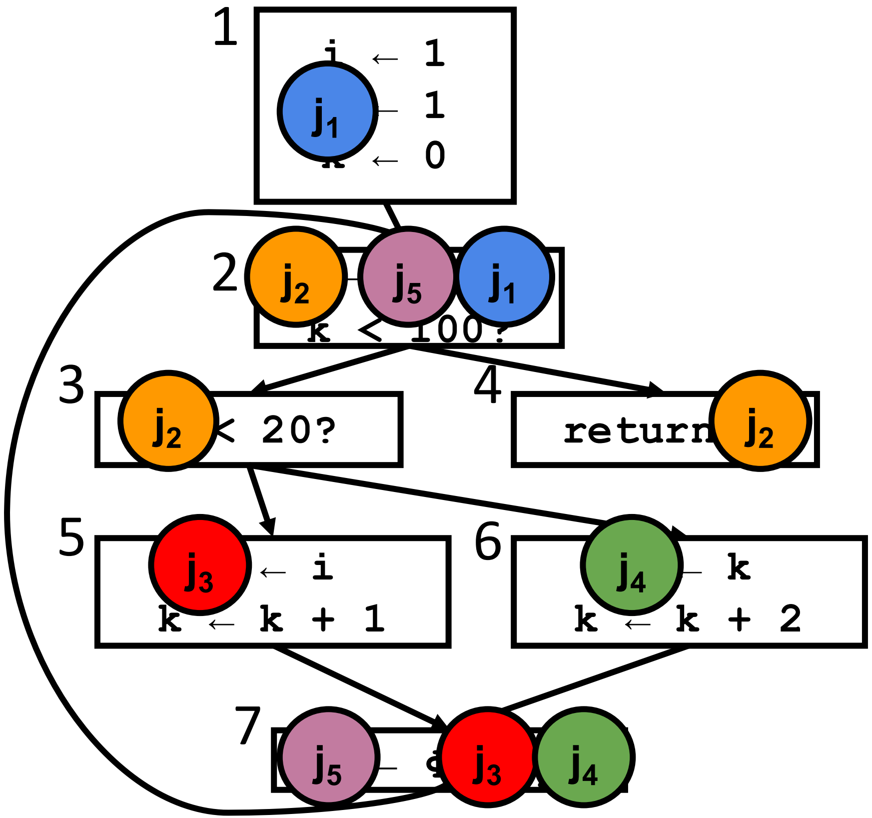


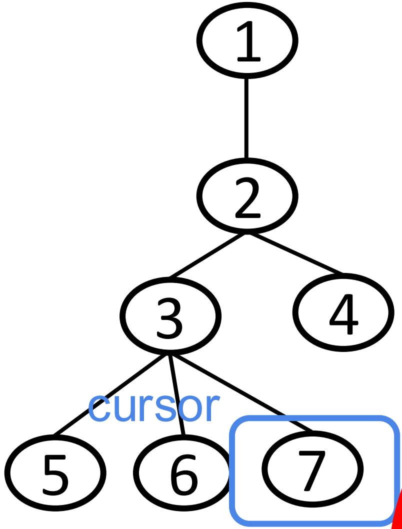
defsites[v]

i	{1}
j	{1,5,6,7,2}
k	{1,5,6}



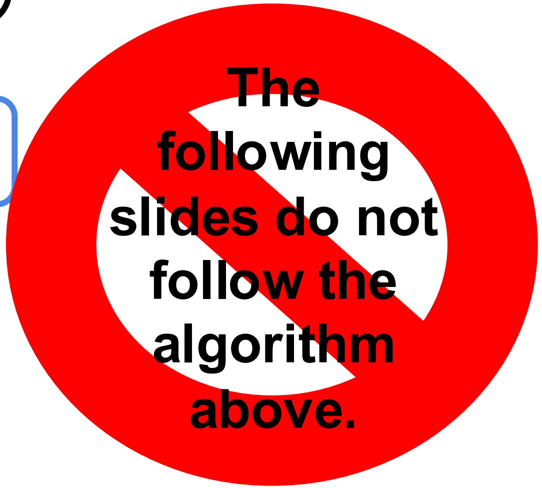
Rename j variables





defsites[v]

i	{1}
j	{1,5,6,7,2}
k	{1,5,6}



Flavors of SSA

- Minimal SSA
  - at each join point with >1 outstanding definition insert a φ-function
  - Some may be dead
- Pruned SSA
  - only add live φ-functions
  - must compute LIVEOUT
- Semi-pruned SSA
  - Same as minimal SSA, but only on names live across more than 1 basic block

Summary

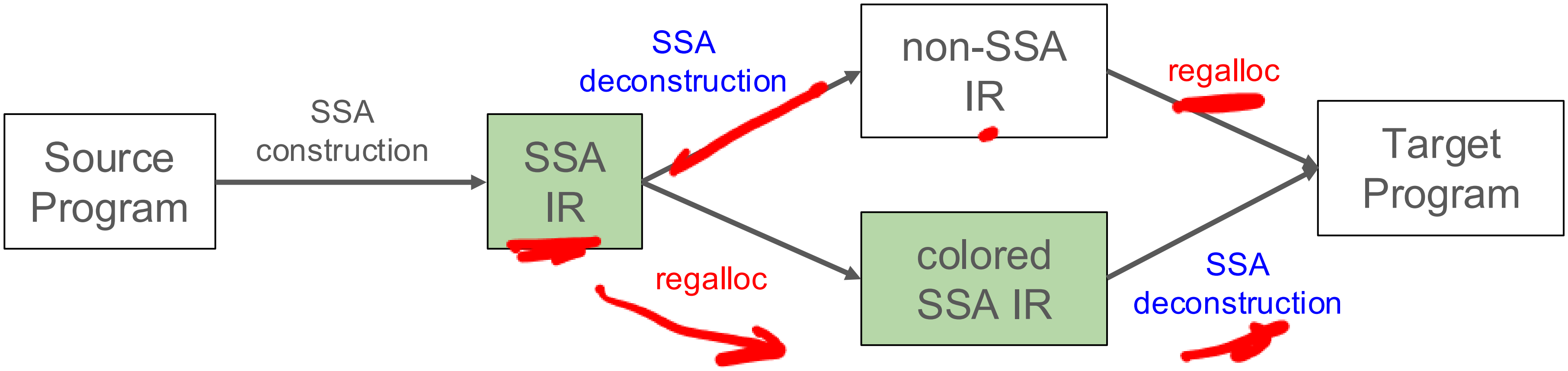
- SSA is a useful and efficient IR.
- Definitions dominate uses
- Constructing SSA can be efficient
  (No need to do Lengaur-Tarjan Algorithm, instead see A Simple, Fast Dominance Algorithm by Cooper, Harvey, and Kennedy)
- Don't do any optimizations yet!

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.

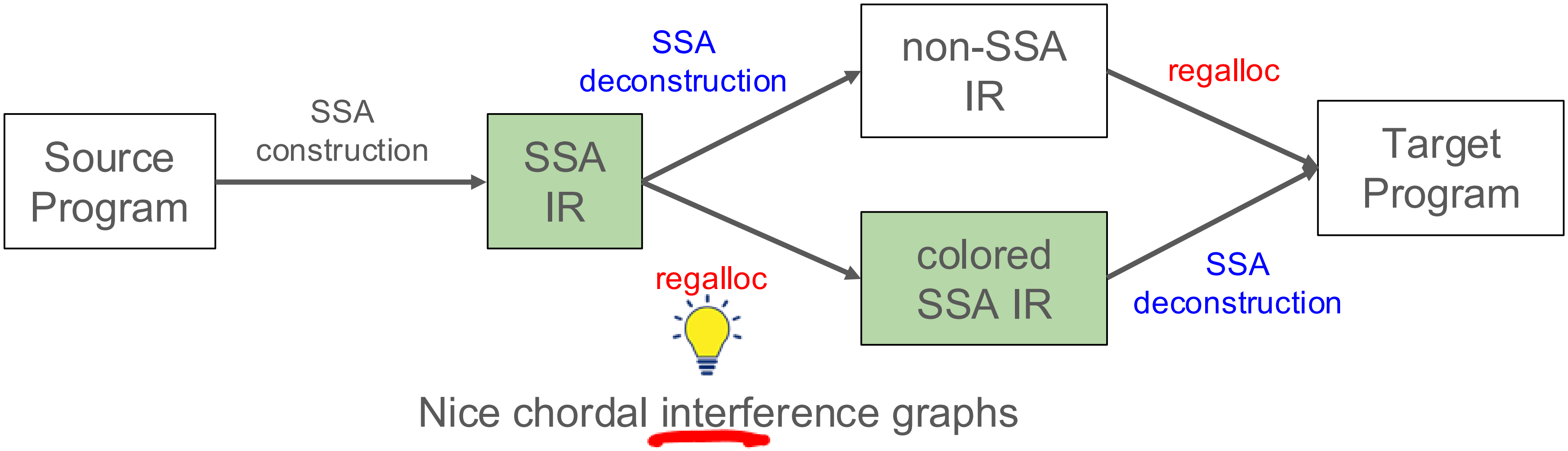
Deconstructing SSA

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



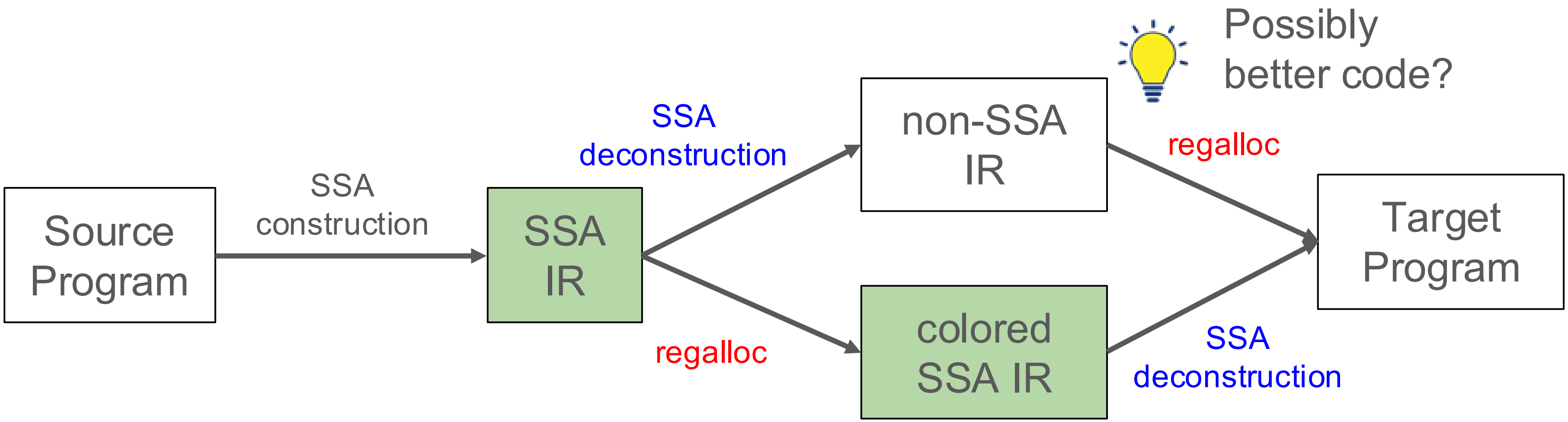
Deconstructing SSA

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



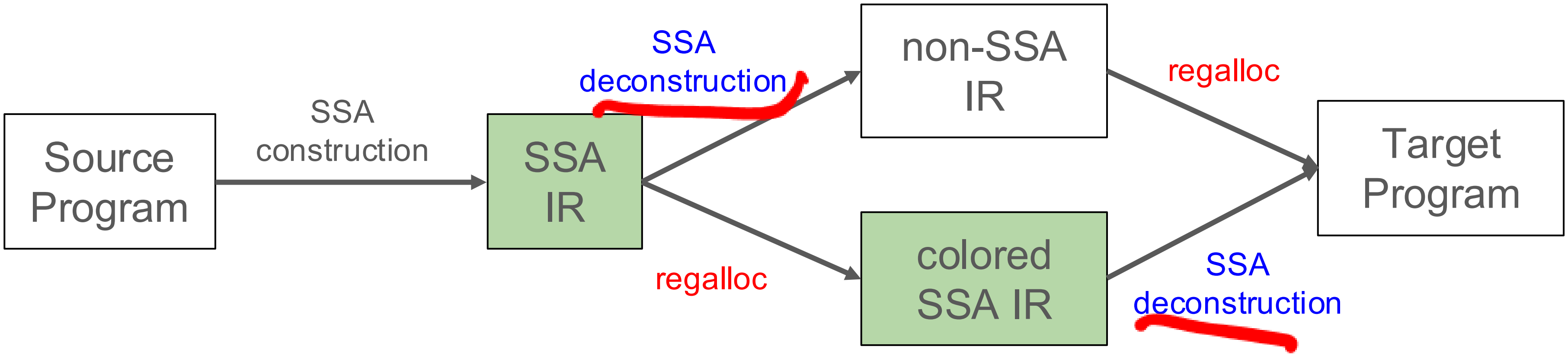
Deconstructing SSA

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



Deconstructing SSA

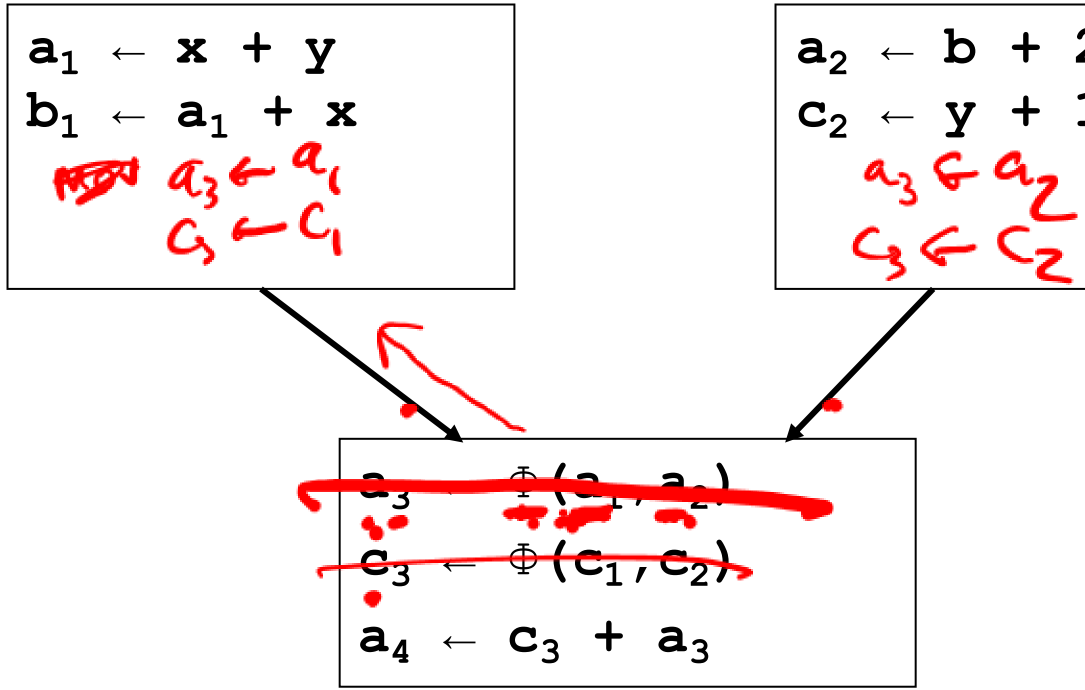
- When during compilation to deconstruct SSA?
- There are two common choices: before or after regalloc.
- Regalloc before deconstruction is relatively new (2010s).



Deconstruction is more or less the same either way.

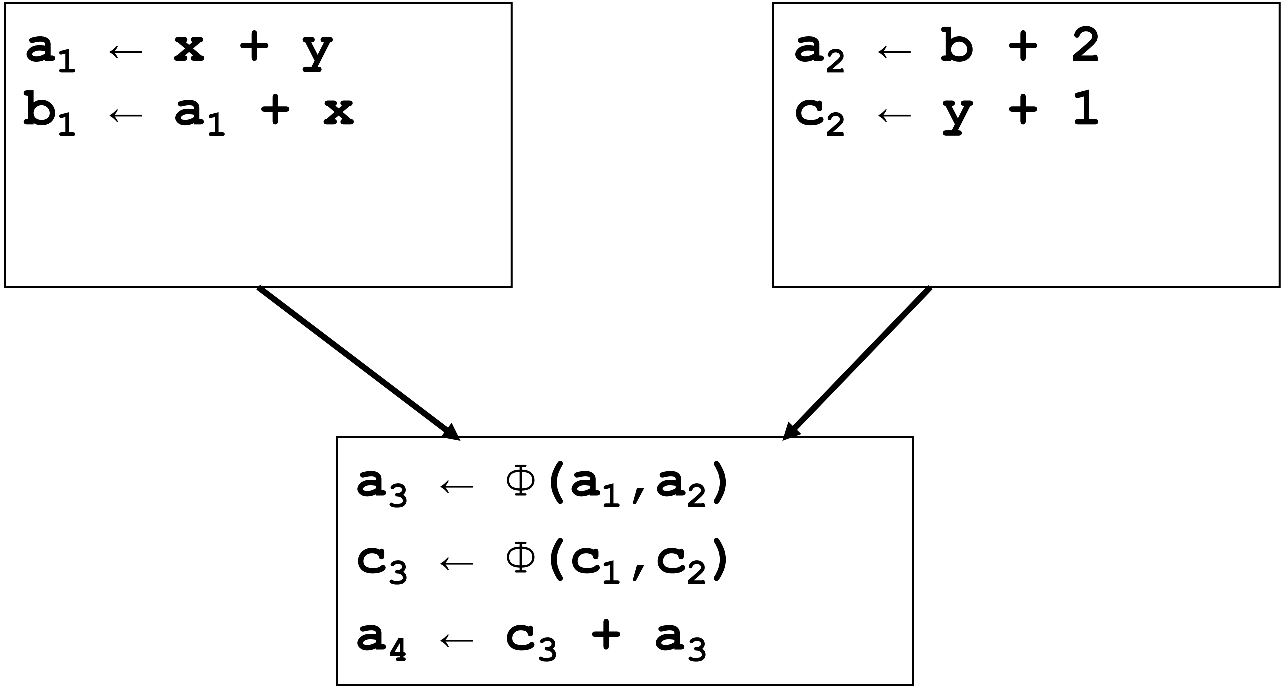
Deconstructing SSA

- Insert moves according to the positional correspondence of inputs.



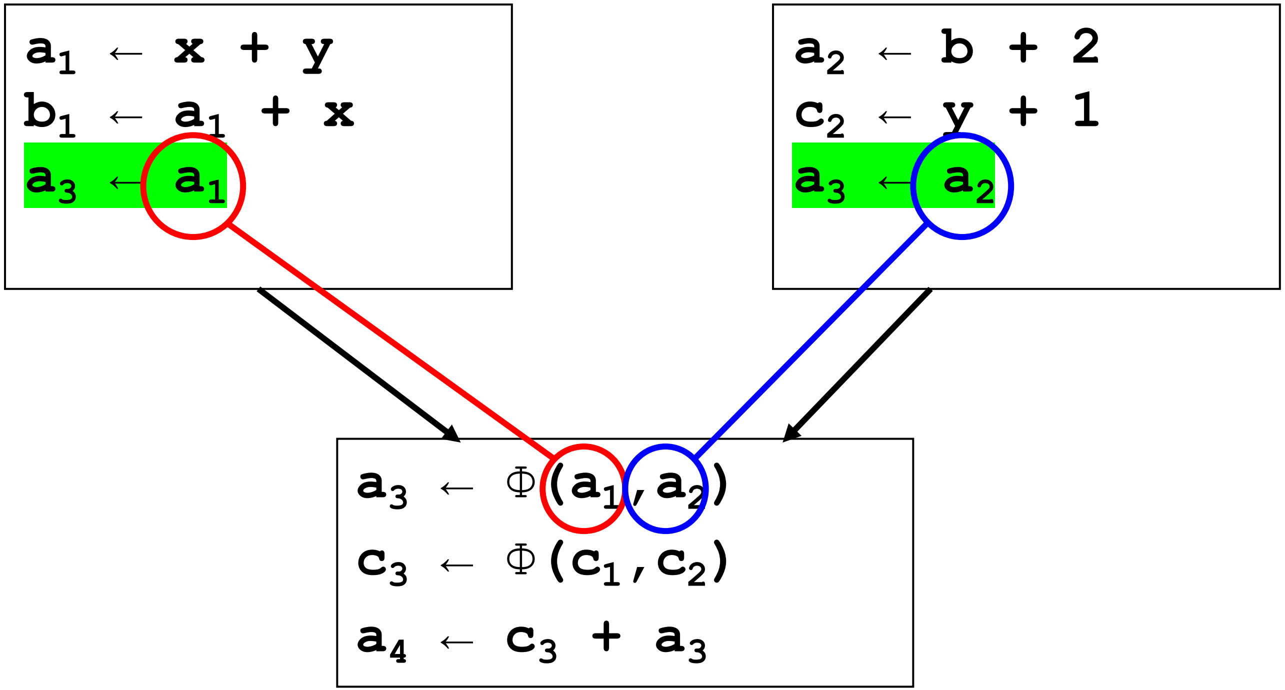
Deconstructing SSA

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



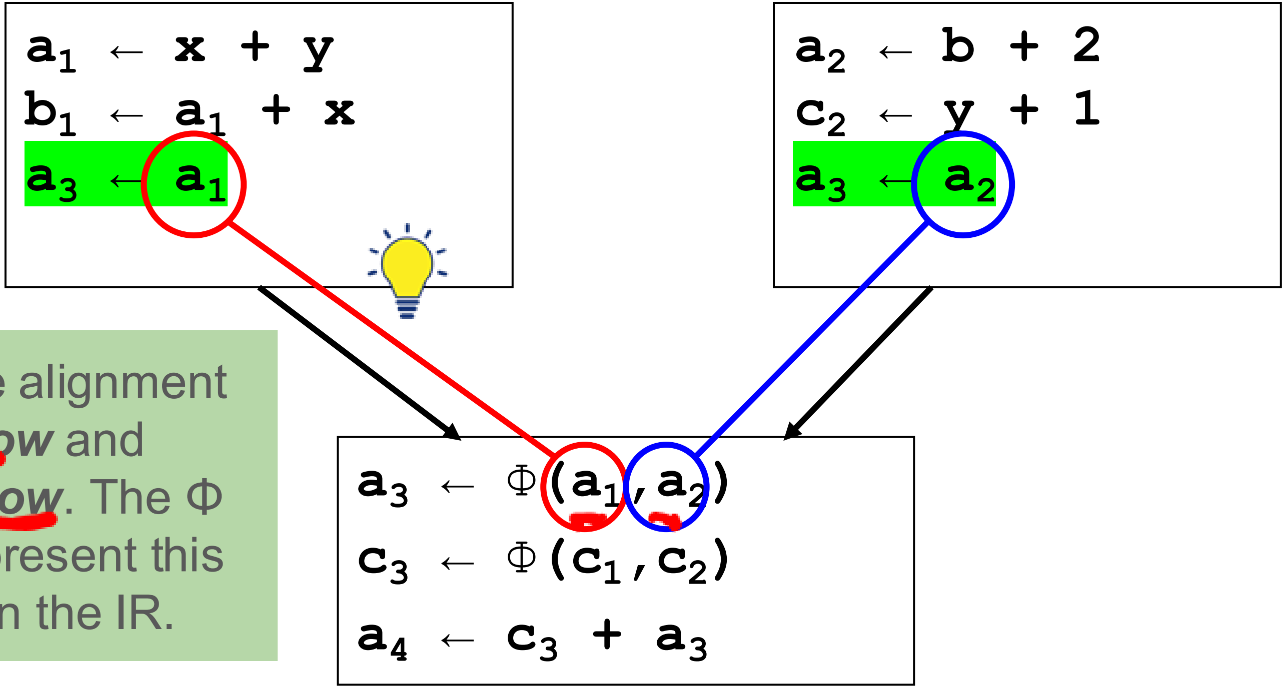
Deconstructing SSA

- 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.

```
a1 ← x + y
b1 ← a1 + x
a3 ← a1
c3 ← c1
```

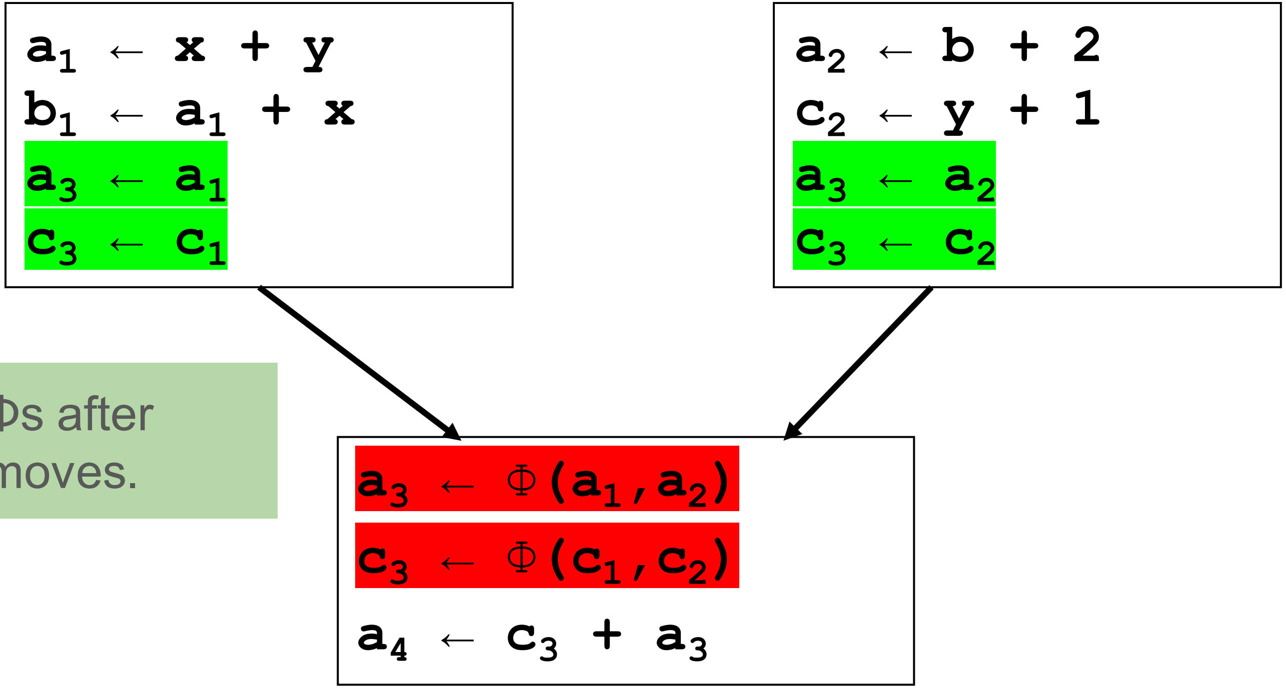
```
a2 ← b + 2
c2 ← y + 1
a3 ← a2
c3 ← c2
```

```
a3 ← Φ(a1,a2)
c3 ← Φ(c1,c2)
a4 ← c3 + a3
```

Each Φ introduces one move into each predecessor node.

Deconstructing SSA

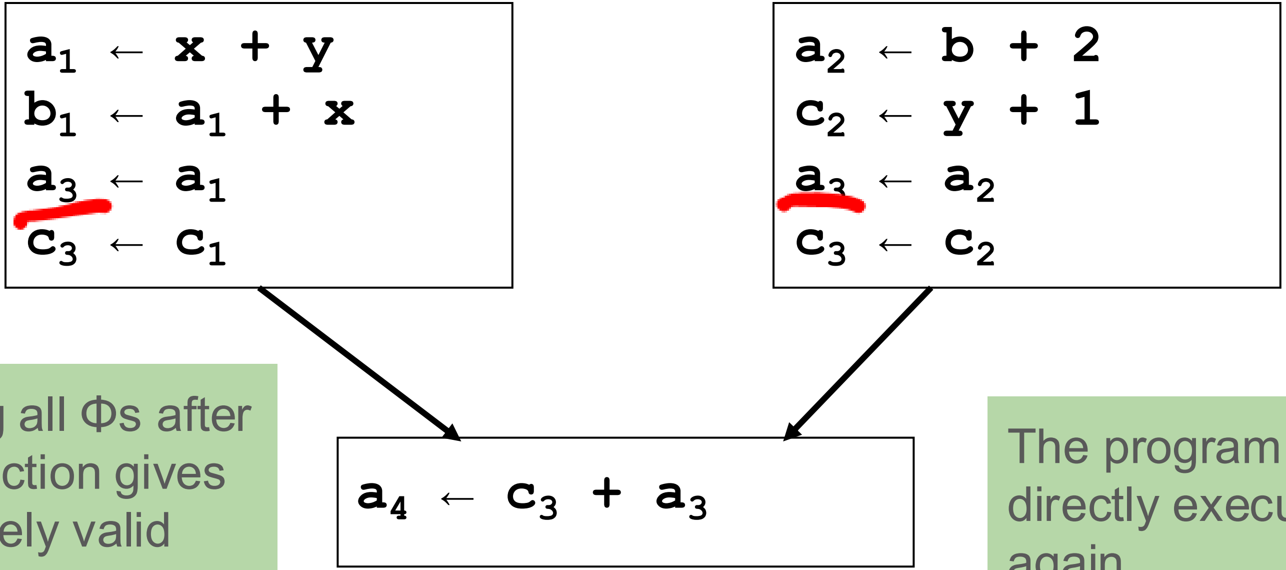
- Insert moves according to the positional correspondence of inputs.



Remove Φs after inserting moves.

Deconstructing SSA

- Insert moves according to the positional correspondence of inputs.

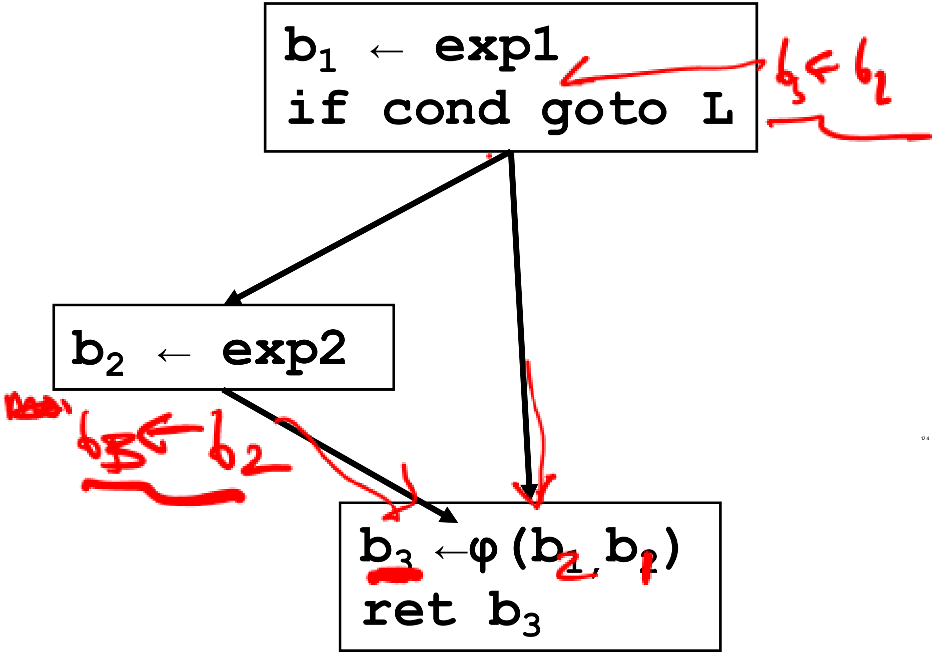


Removing all Φs after deconstruction gives a completely valid non-SSA program.

The program is now directly executable again.

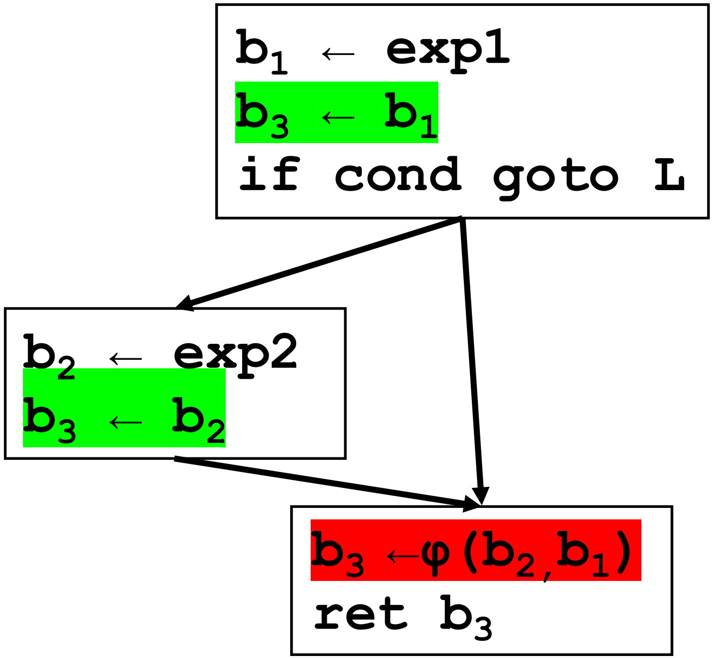
Issue 1: Critical Edges

- Consider a simple triangle CFG.



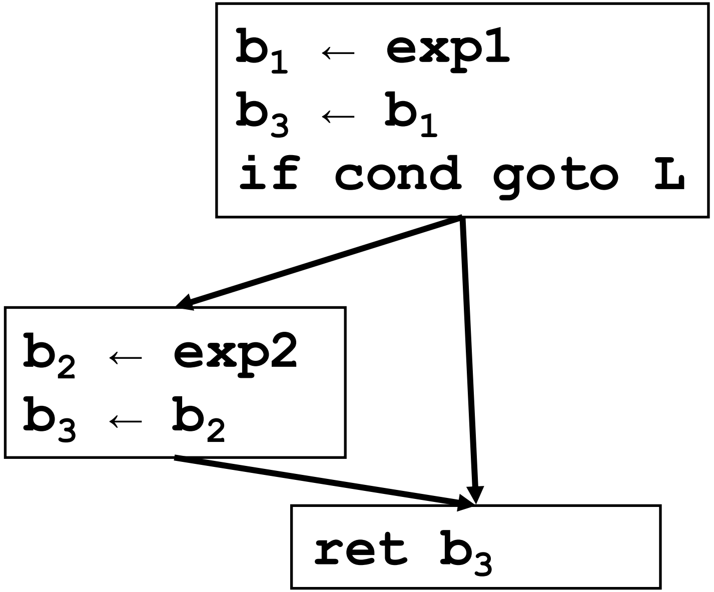
Issue 1: Critical Edges

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



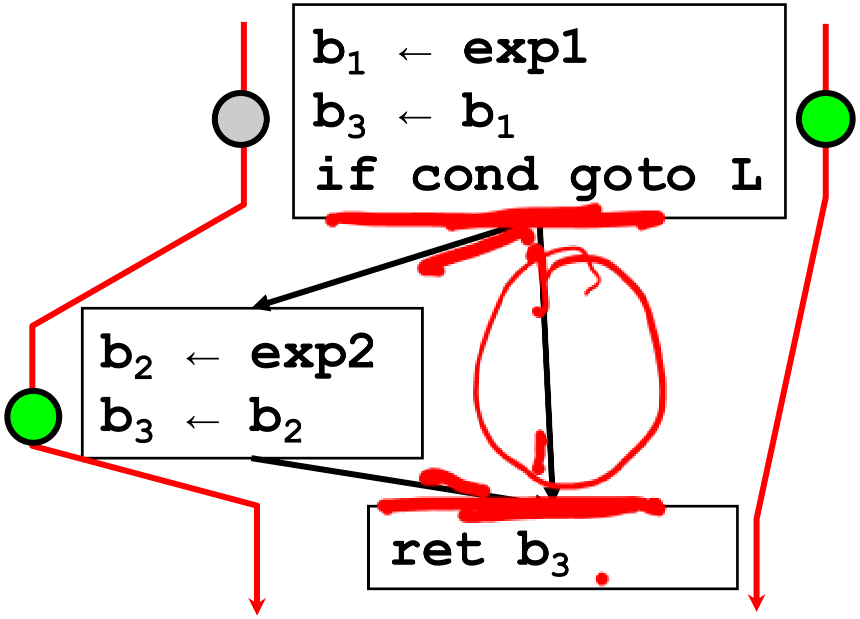
Issue 1: Critical Edges

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



Issue 1: Critical Edges

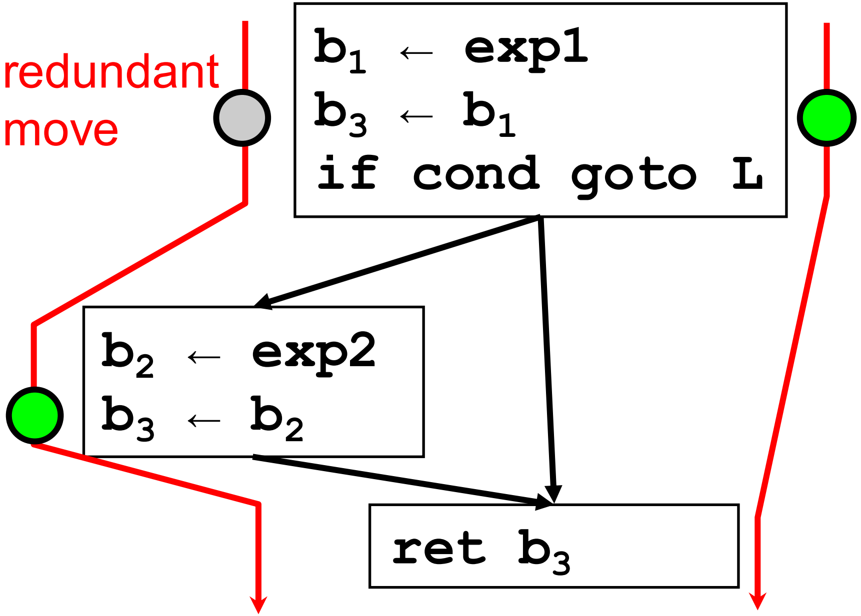
- Consider a simple triangle CFG.
- We insert moves in both predecessors and remove the Φ.



Issue 1: Critical Edges

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

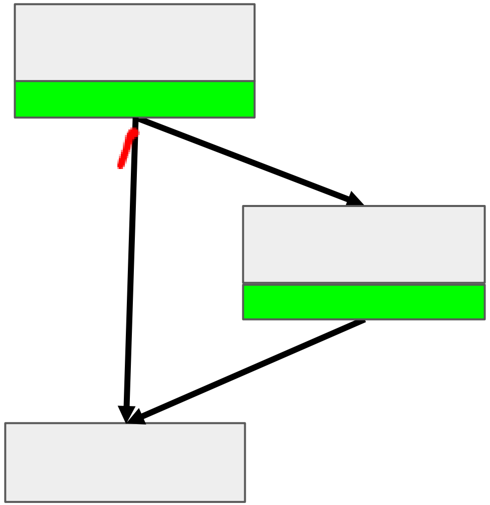
Naïve insertion can introduce redundant code on some execution paths.



Issue 1: Critical Edges

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

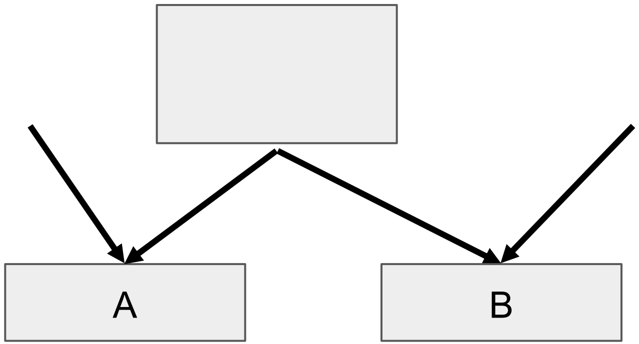
Naïve insertion can introduce redundant code on some execution paths.



Issue 1: Critical Edges

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

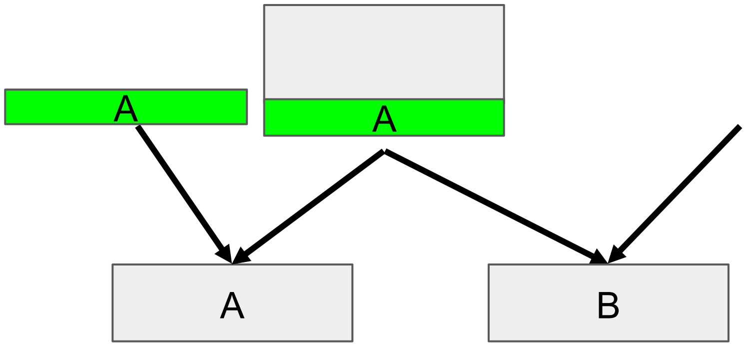
Naïve insertion can introduce redundant code on some execution paths.



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.

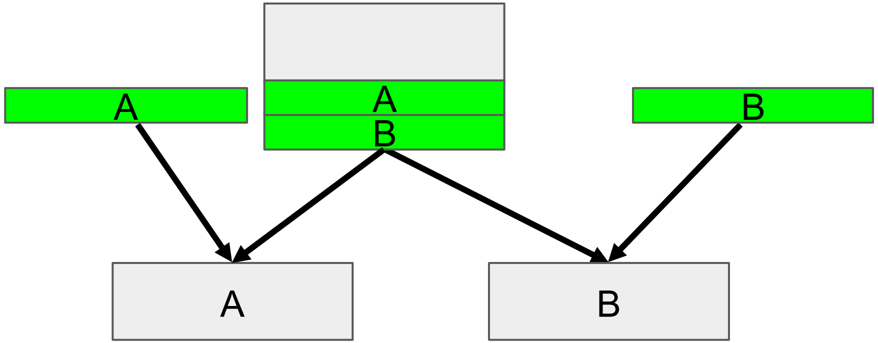


Issue 1: Critical Edges

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

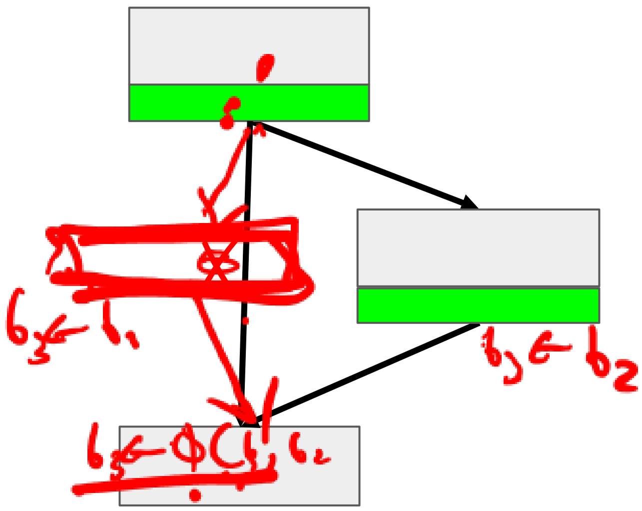
Naïve insertion can introduce redundant code on some execution paths.

Can actually get *really* bad.



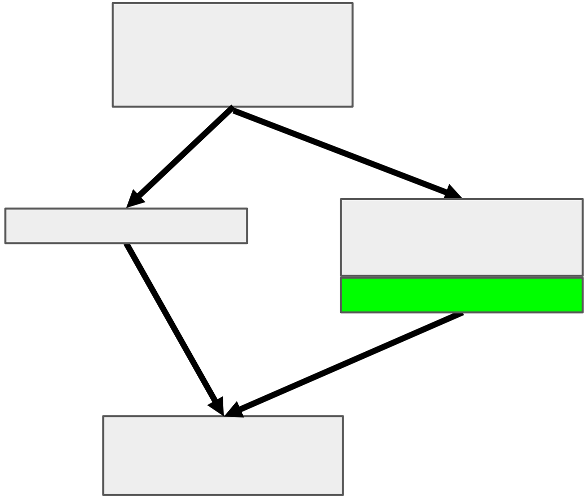
Splitting Critical Edges

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



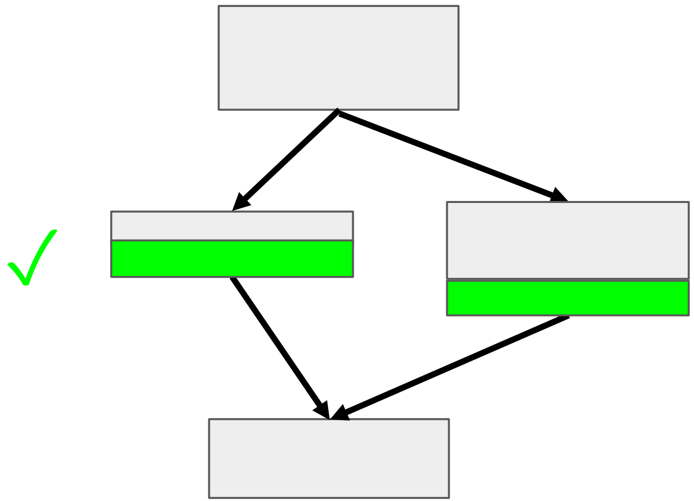
Splitting Critical Edges

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



Splitting Critical Edges

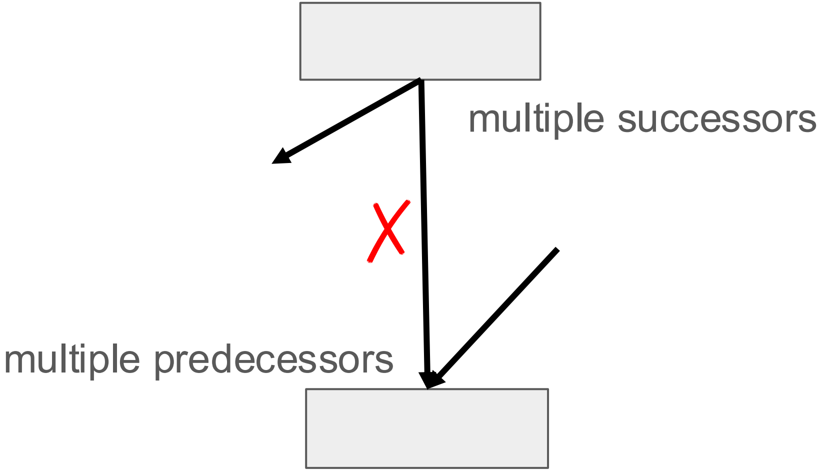
- To avoid redundant moves, split *critical edges* by inserting an empty block between.
- This block is the proper place for Φ-resolution moves.



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.

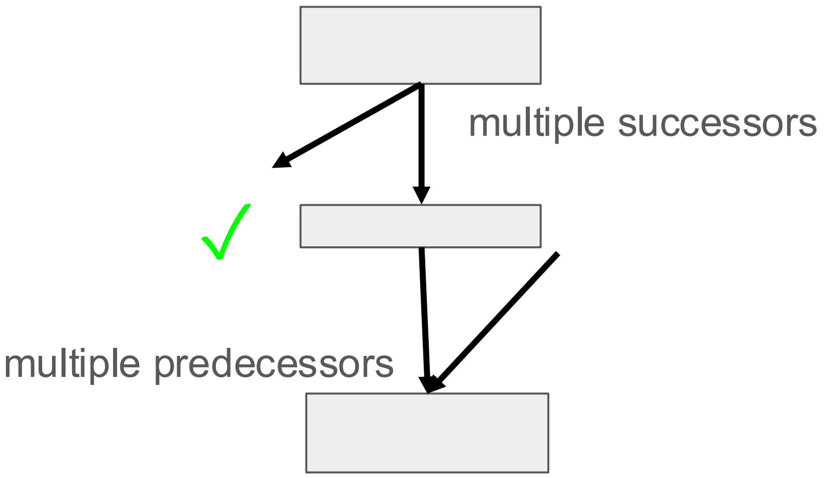
A *critical edge* is any edge that connects a block with multiple successors to a block with multiple predecessors.



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.



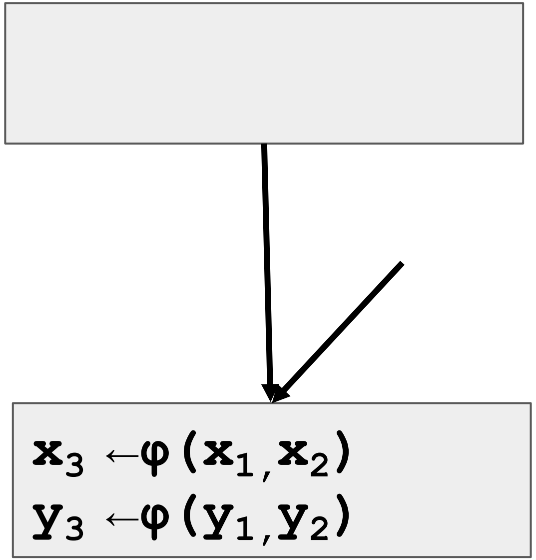
Issue 2: Ordering Moves

- 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?

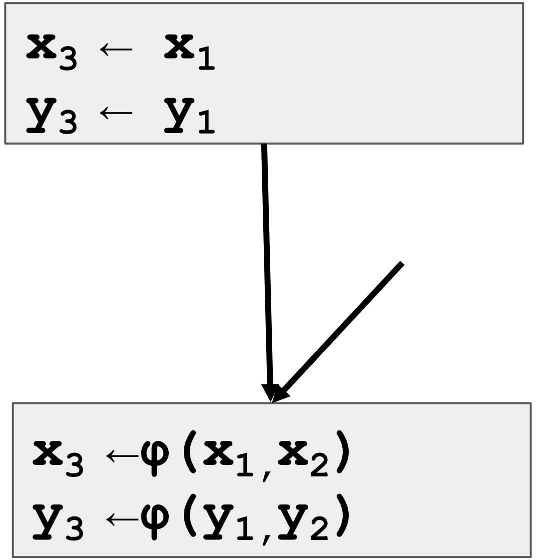
- Consider a join with at least two Φs.



Issue 2: Ordering Moves

- Does the order of Φ-resolution moves matter?

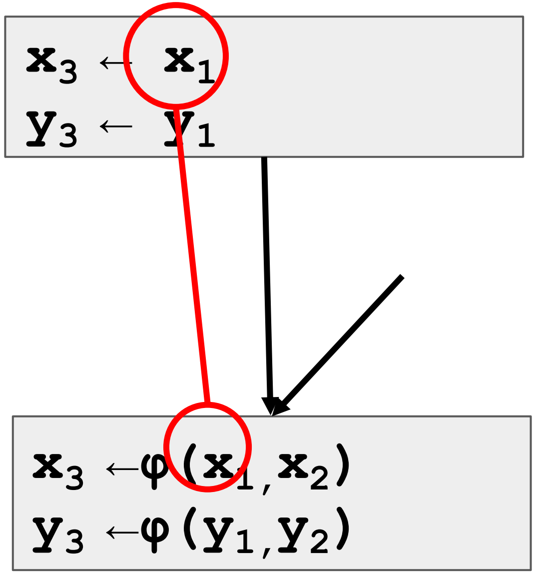
- Consider a join with at least two Φs.
- Moves are inserted into predecessors.



Issue 2: Ordering Moves

- Does the order of Φ-resolution moves matter?

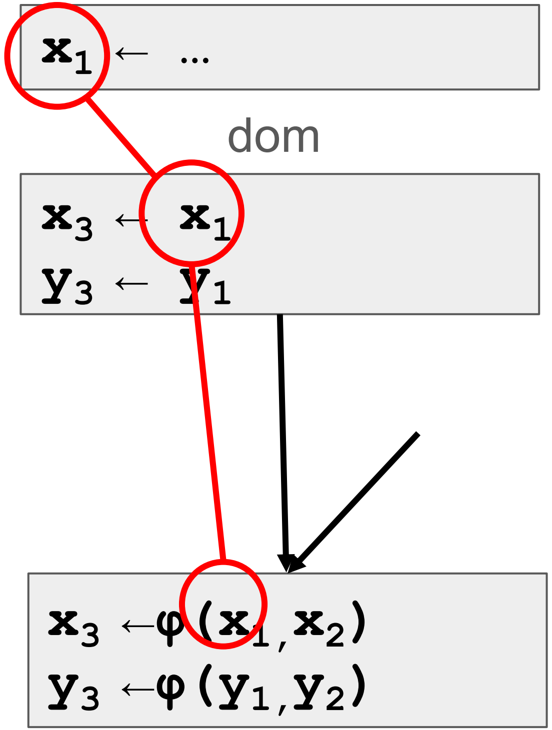
- Consider a join with at least two Φs.
- Moves are inserted into predecessors.



Issue 2: Ordering Moves

- Does the order of Φ-resolution moves matter?

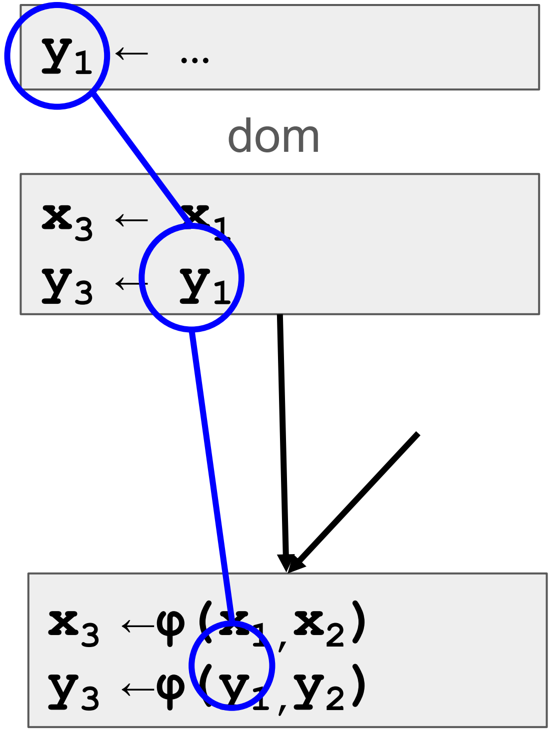
- Consider a join with at least two Φs.
- Moves are inserted into predecessors.
- By SSA invariants, the definition of the RHS of each move dominates the move.



Issue 2: Ordering Moves

- Does the order of Φ-resolution moves matter?

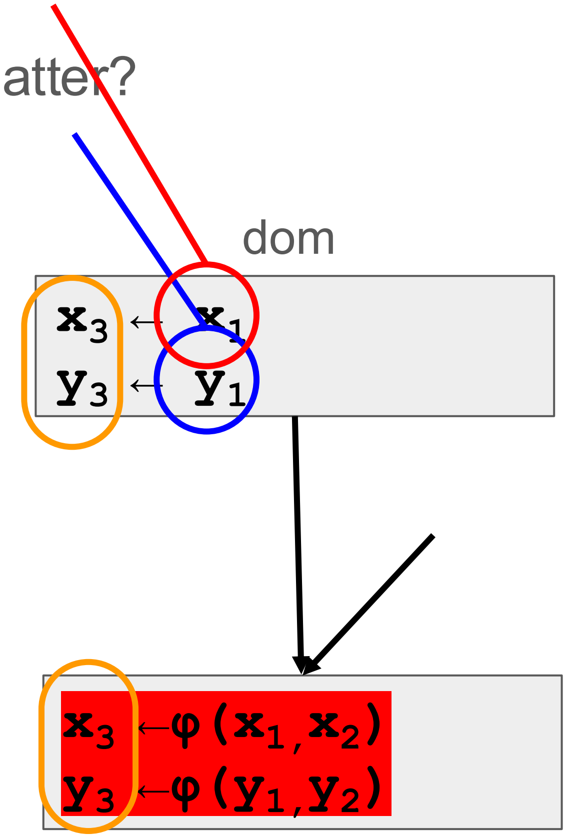
- Consider a join with at least two Φs.
- Moves are inserted into predecessors.
- By SSA invariants, the definition of the RHS of each move dominates the move.



Issue 2: Ordering Moves

- Does the order of Φ-resolution moves matter?

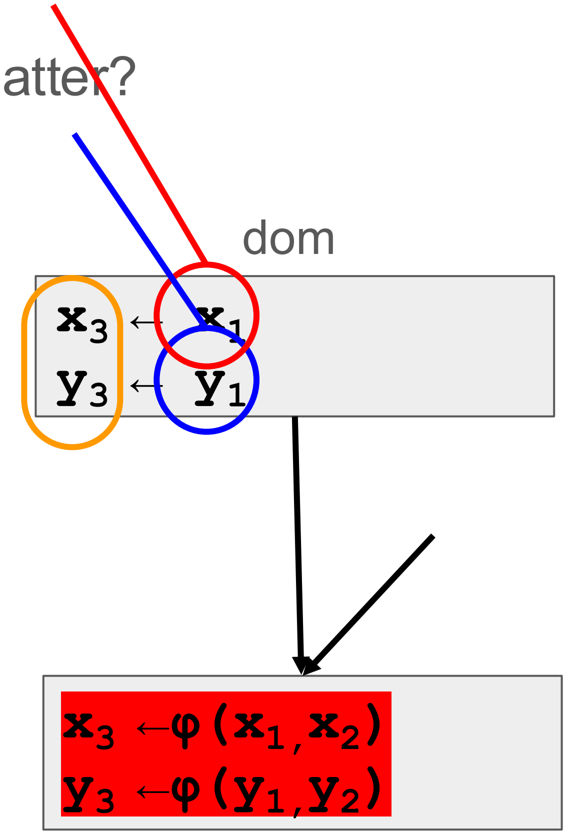
- Consider a join with at least two Φs.
- Moves are inserted into predecessors.
- By SSA invariants, the definition of the RHS of each move dominates the move.
- It cannot be the case that the LHS is live, because previously there was only one definition, below.



Issue 2: Ordering Moves

- Does the order of Φ-resolution moves matter?

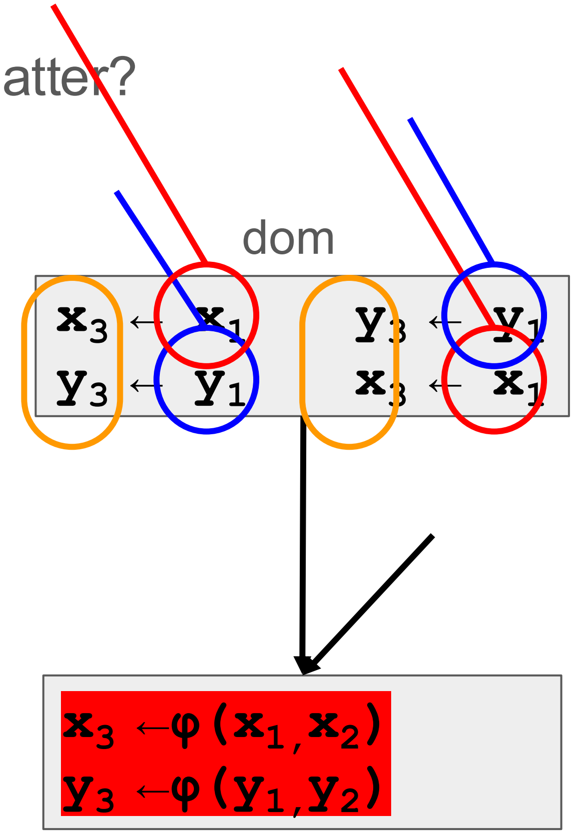
- Consider a join with at least two Φs.
- Moves are inserted into predecessors.
- By SSA invariants, the definition of the RHS of each move dominates the move.
- It cannot be the case that the LHS is live, because previously there was only one definition, below.
- Therefore we are only assigning to fresh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



Issue 2: Ordering Moves

- Does the order of Φ-resolution moves matter?

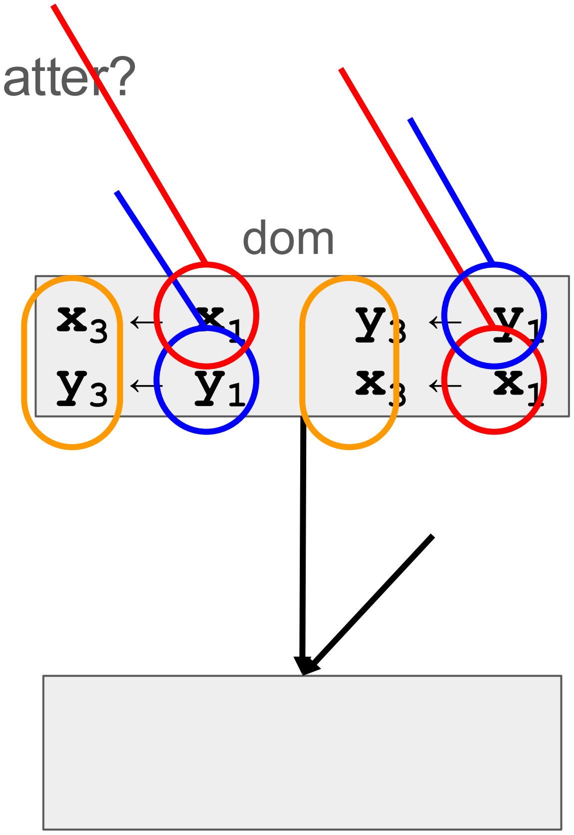
- Consider a join with at least two Φs.
- Moves are inserted into predecessors.
- By SSA invariants, the definition of the RHS of each move dominates the move.
- It cannot be the case that the LHS is live, because previously there was only one definition, below.
- Therefore we are only assigning to fresh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



Issue 2: Ordering Moves

- Does the order of Φ-resolution moves matter?

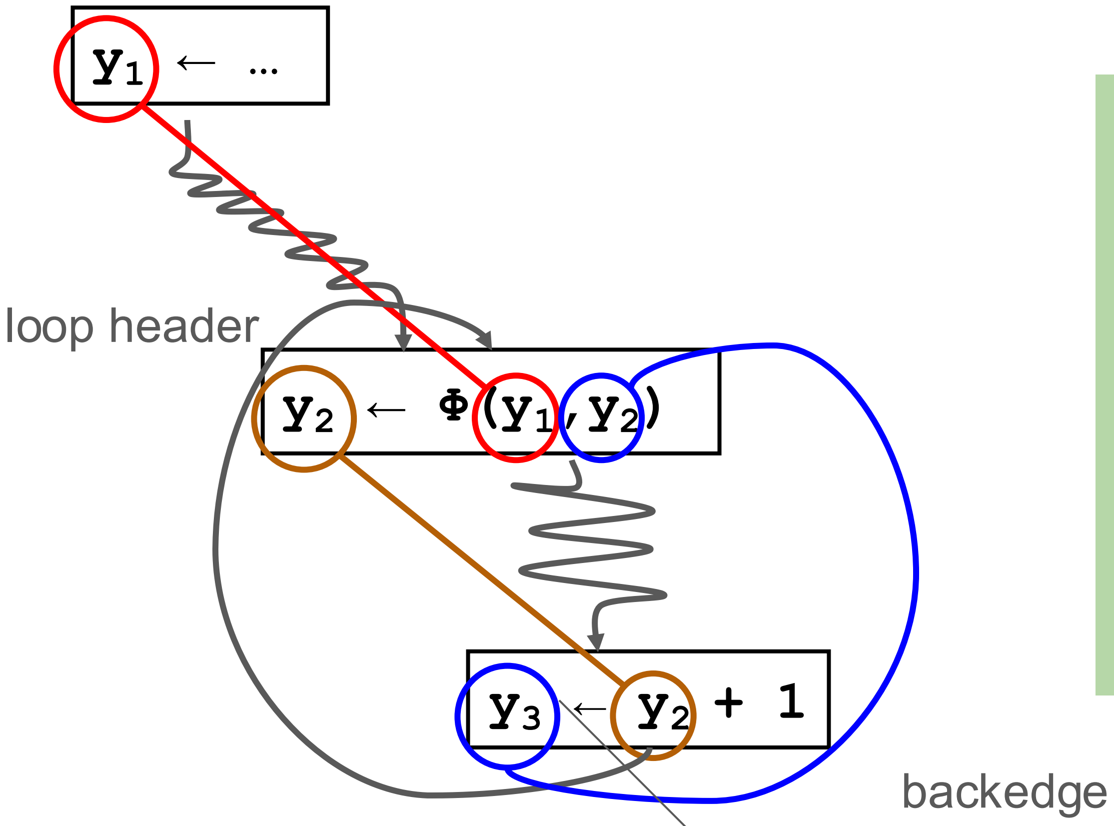
- Consider a join with at least two Φs.
- Moves are inserted into predecessors.
- By SSA invariants, the definition of the RHS of each move dominates the move.
- It cannot be the case that the LHS is live, because previously there was only one definition, below.
- Therefore we are only assigning to fresh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

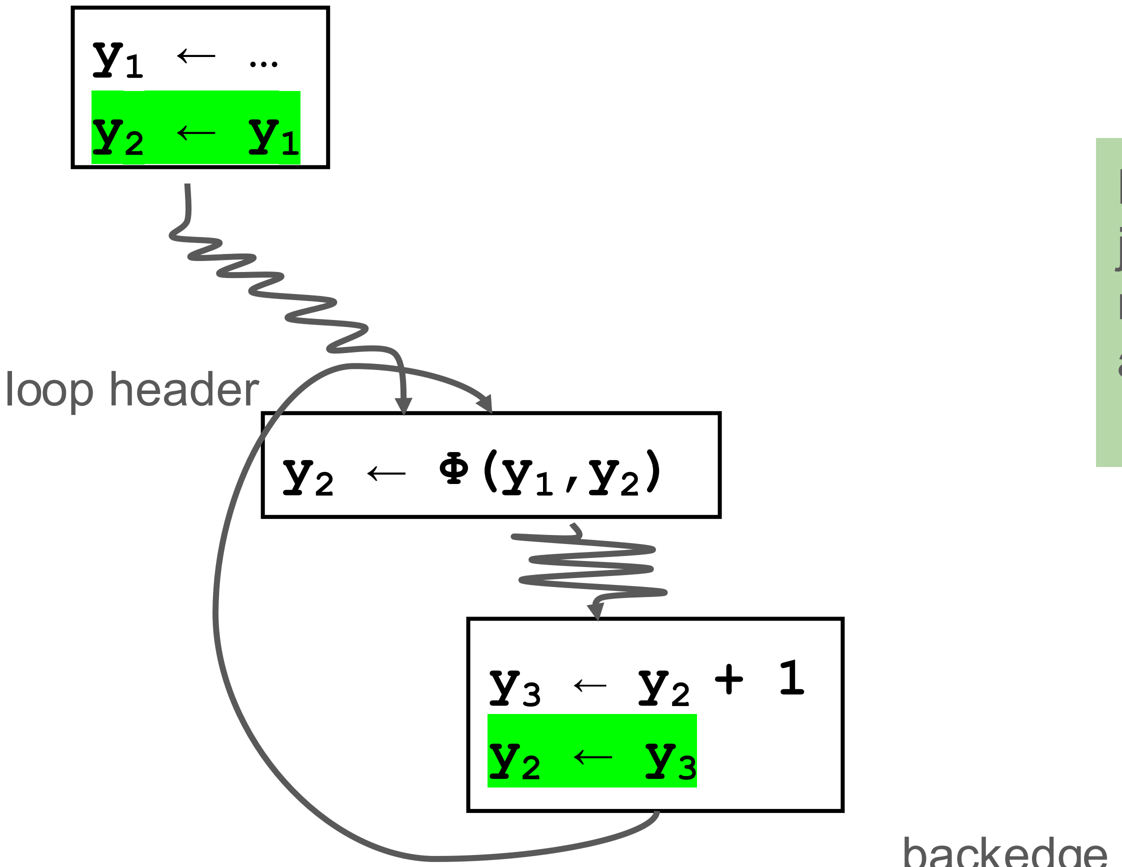
Issue 2: Ordering Moves



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

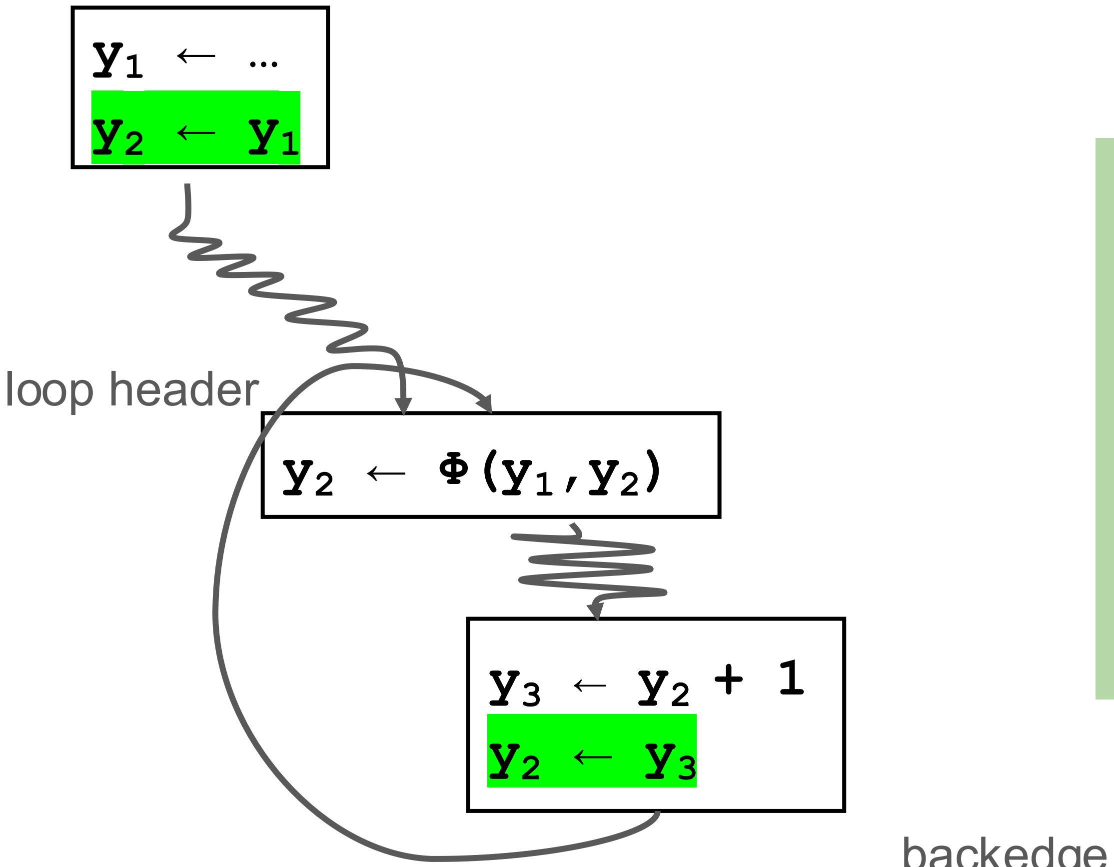
A loop Φ can be defined in terms of itself.

Issue 2: Ordering Moves



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

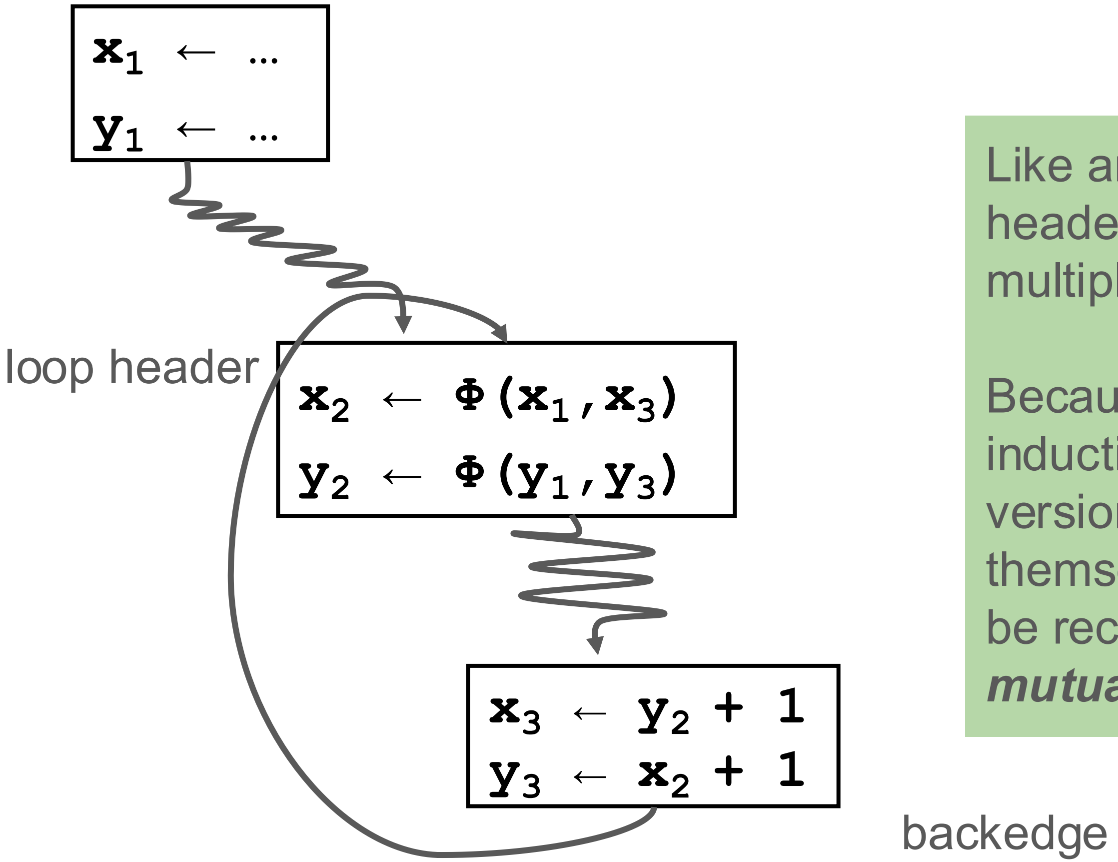
Issue 2: Ordering Moves



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

With only one Φ, there is no problem yet.

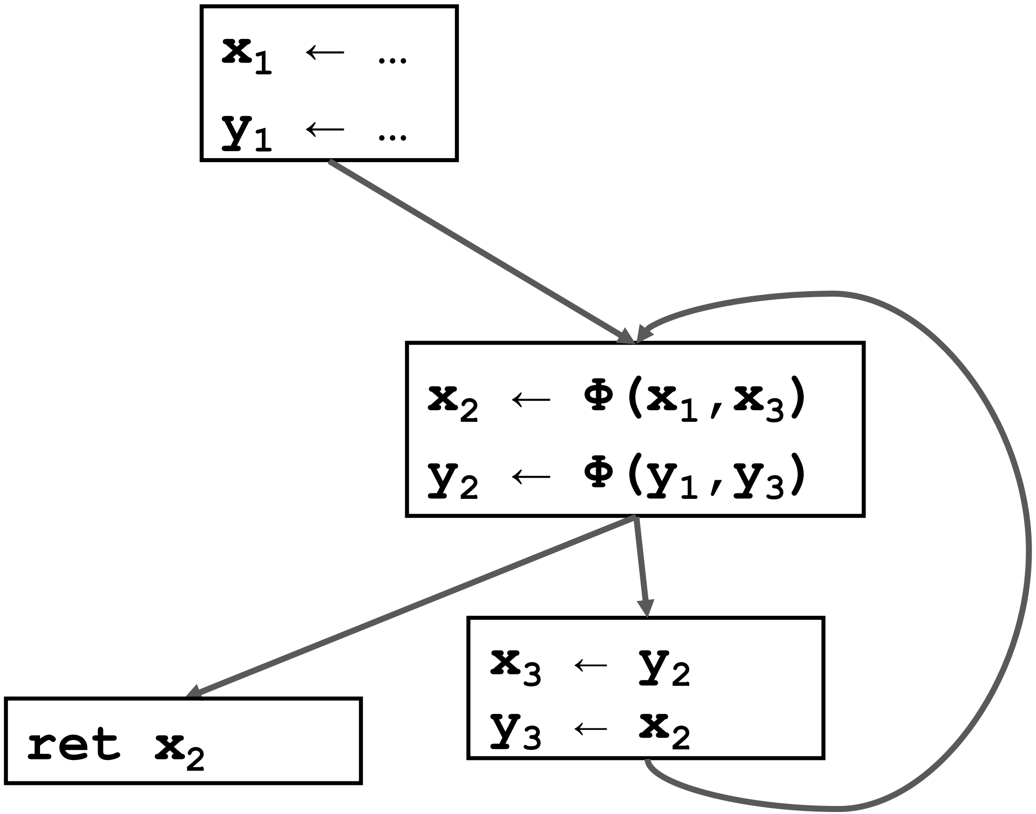
Issue 2: Ordering Moves



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

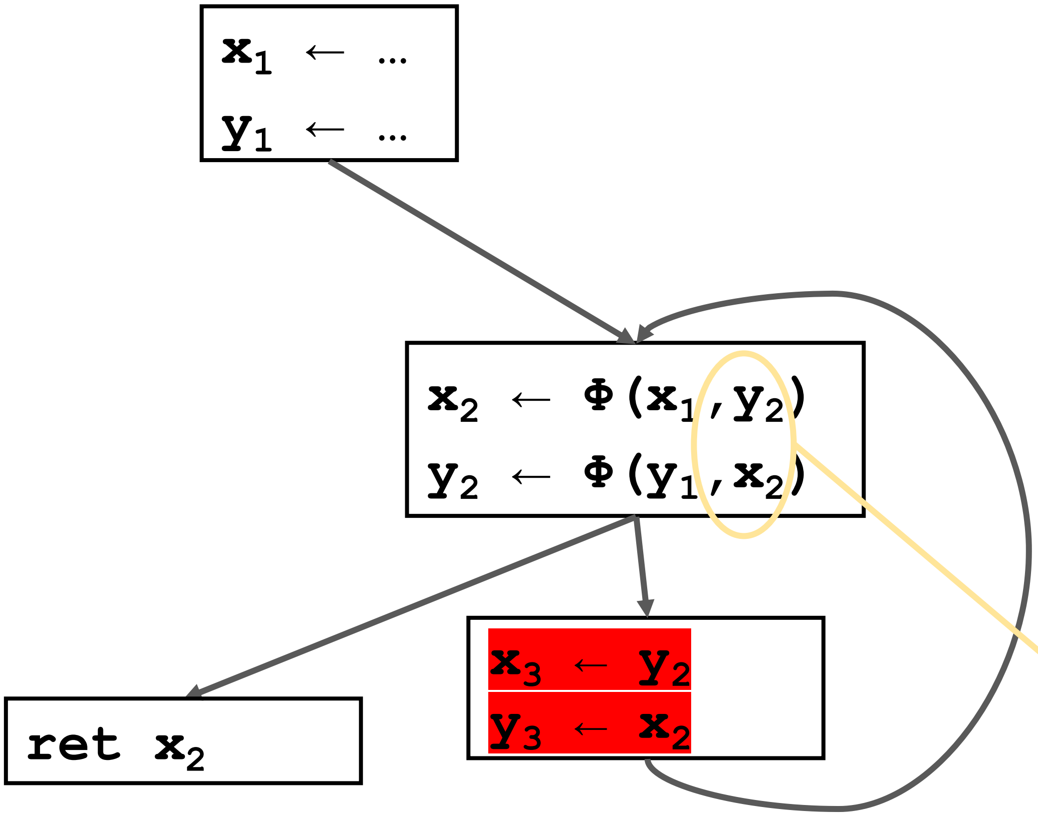
Because Φs can use inductively defined versions of themselves, they can be recursive or even ***mutually recursive***.

Issue 2: Ordering Moves



A simple example: swap of variables in a loop.

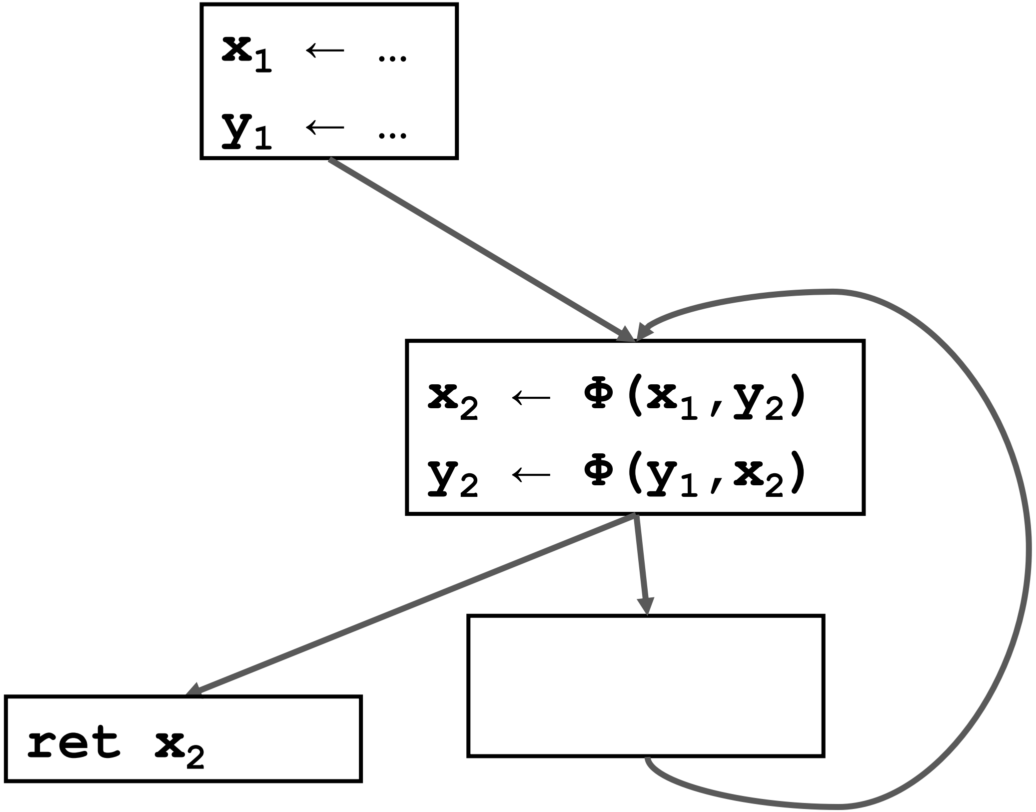
Issue 2: Ordering Moves



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

Replace
x_3 with y_2
y_3 with x_2

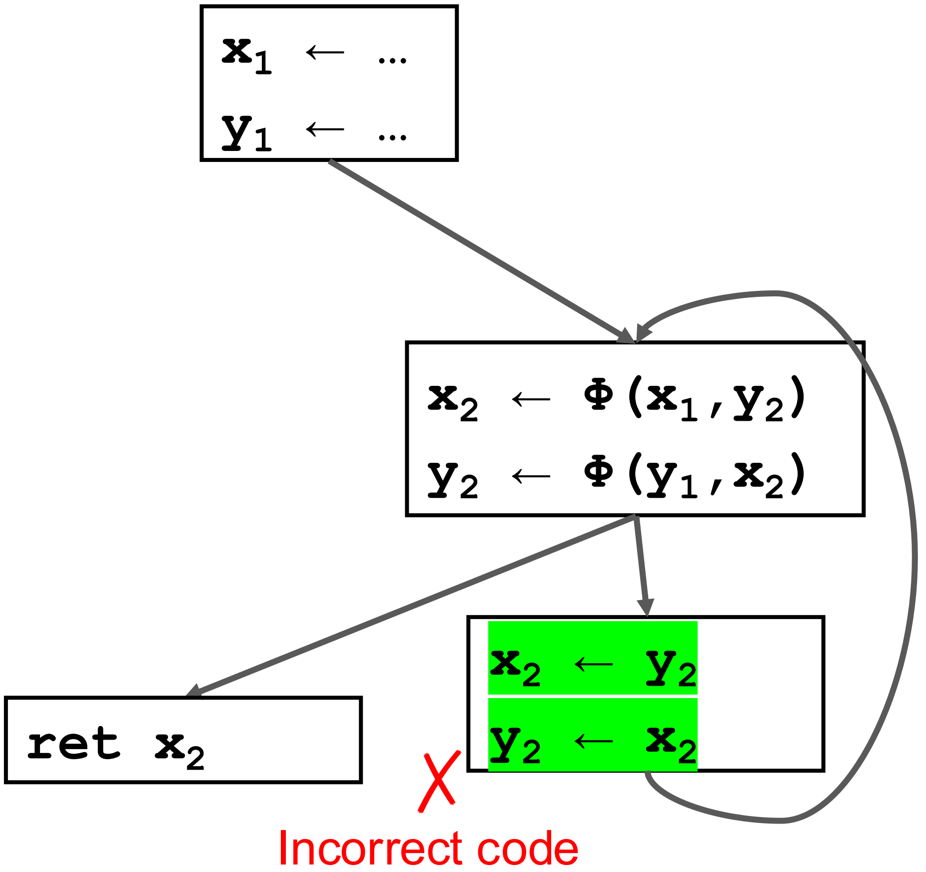
Issue 2: Ordering Moves



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

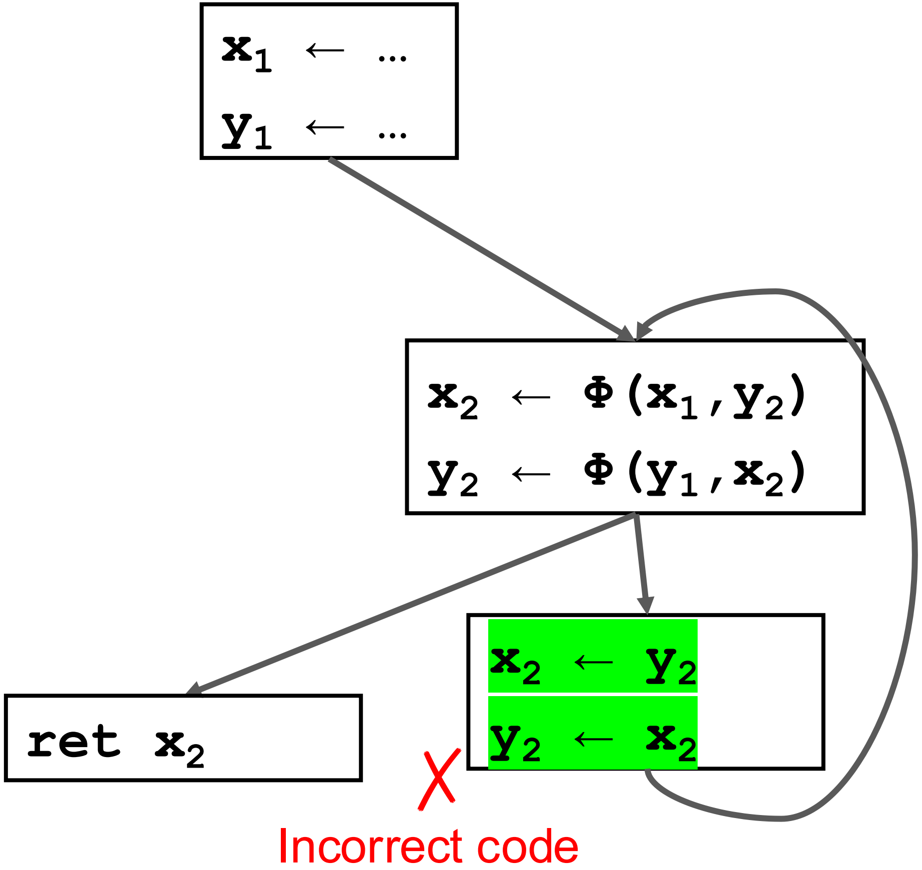
This is totally legal and cool.

Issue 2: Ordering Moves

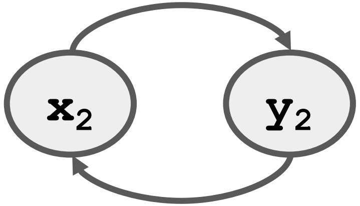


SSA deconstruction using the naïve move insertion will always generate incorrect code, regardless of the order.

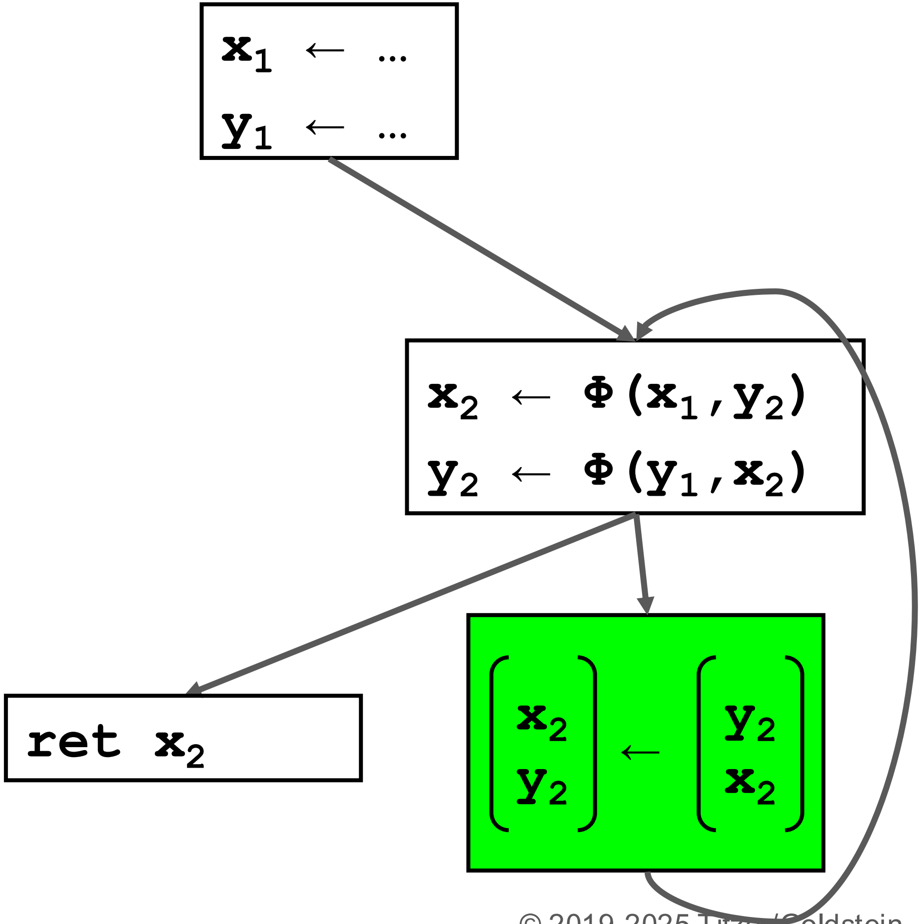
Issue 2: Ordering Moves



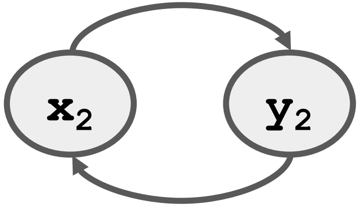
SSA deconstruction using the naïve move insertion will always generate incorrect code, regardless of the order.



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

$t_0 \leftarrow y_0$
$t_1 \leftarrow y_1$
$t_2 \leftarrow y_2$
$t_3 \leftarrow y_3$
$x_0 \leftarrow t_0$
$x_1 \leftarrow t_1$
$x_2 \leftarrow t_2$
$x_3 \leftarrow t_3$

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.

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

Next SSA Lecture

- Finish Deconstructing SSA
- More practice building SSA
- Constant propagation with SSA
- SSA in practice

Implementing Parallel Moves

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

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

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

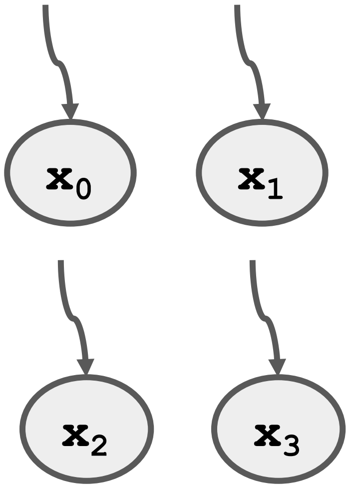
- Φ resolution moves must be done in parallel, without overwriting old versions.
- Better solution: order moves more intelligently ***using LTG***.

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

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

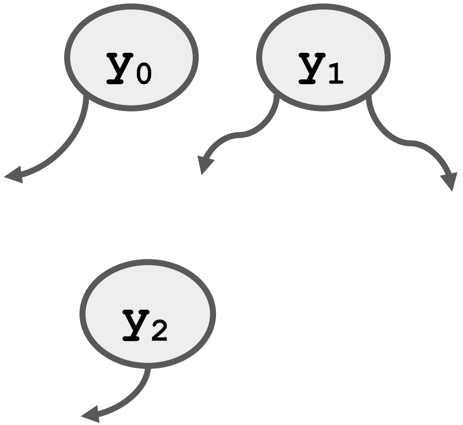
- Φ resolution moves must be done in parallel, without overwriting old versions.
- Better solution: order moves more intelligently ***using LTG***.

Location Transfer Graph

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

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 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.

$$\begin{bmatrix} \mathbf{x}_0 \\ \mathbf{x}_1 \\ \mathbf{x}_2 \\ \mathbf{x}_3 \end{bmatrix} \leftarrow \begin{bmatrix} \mathbf{x}_2 \\ \mathbf{x}_0 \\ \mathbf{x}_1 \\ \mathbf{x}_0 \end{bmatrix}$$