SSA

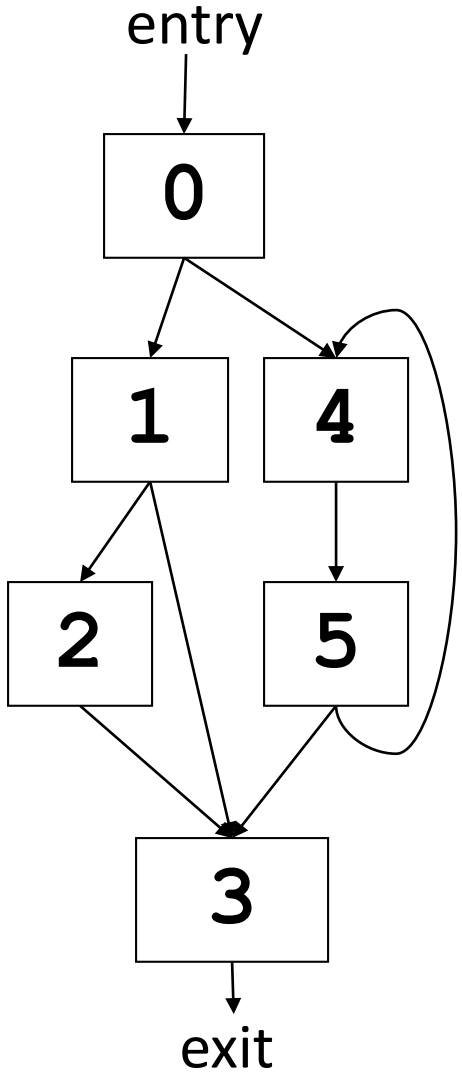
15-411/15-611 Compiler Design

Seth Copen Goldstein

September 17, 2020

Dominance Frontier

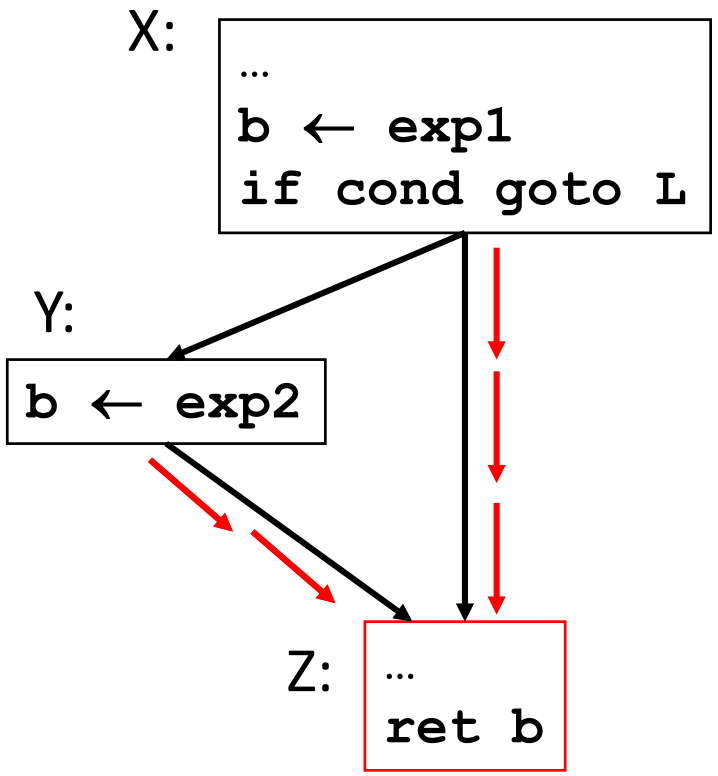
- If *z* is the first node we encounter on the path from *x* which *x* does not *strictly* dominate, *z* is in the dominance frontier of *x*
- For some path from node *x* to *z*,
  x → … → *y* → *z*
  where *x* dom *y* but not *x* sdom *z*.
- Dominance frontier of **1**? {3}
- Dominance frontier of **2**?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



When do we insert Φ?

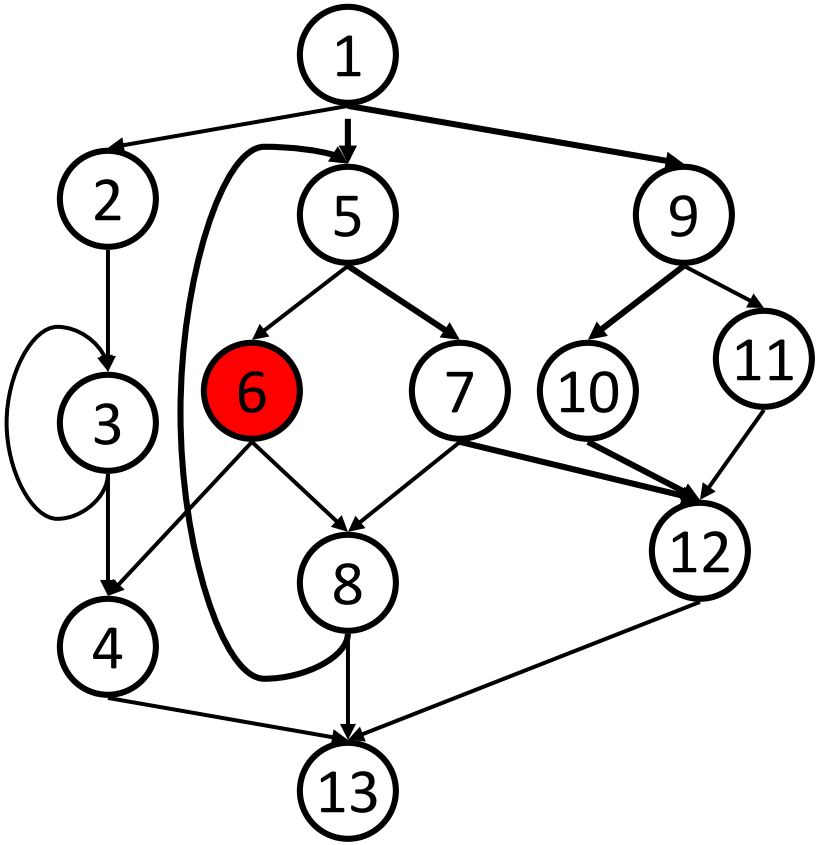
There should be a Φ-function for variable b at node z of the flow graph exactly when all of the following are true:

- There is a block x containing a def of b
- There is a block y (with 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.

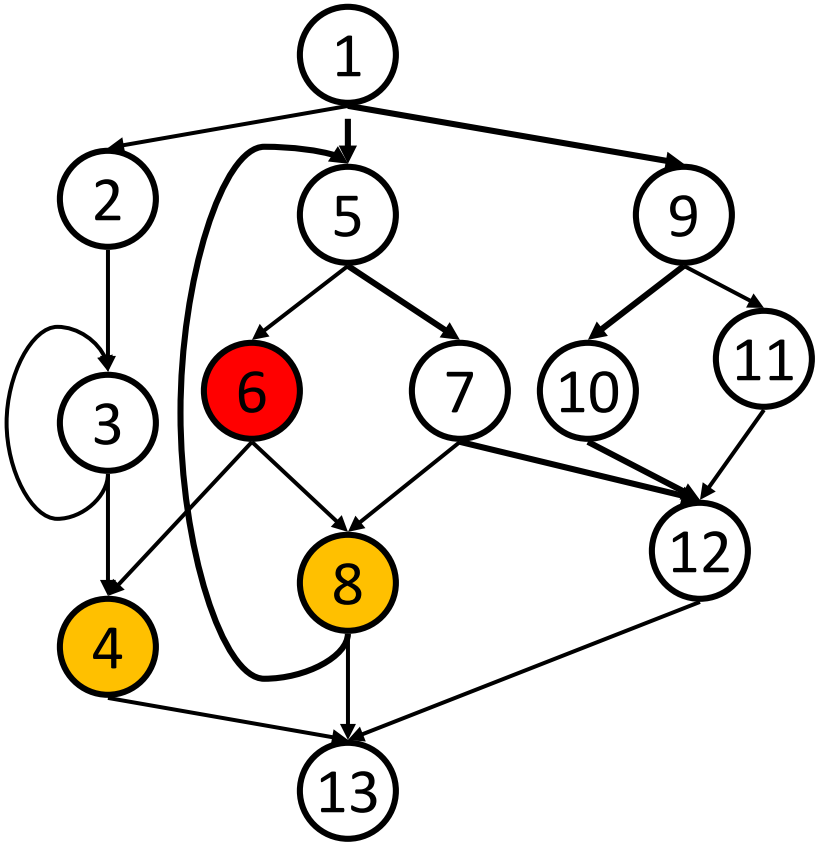


IOW, Z ∈ DF(X)

Dominance Frontier Criterion

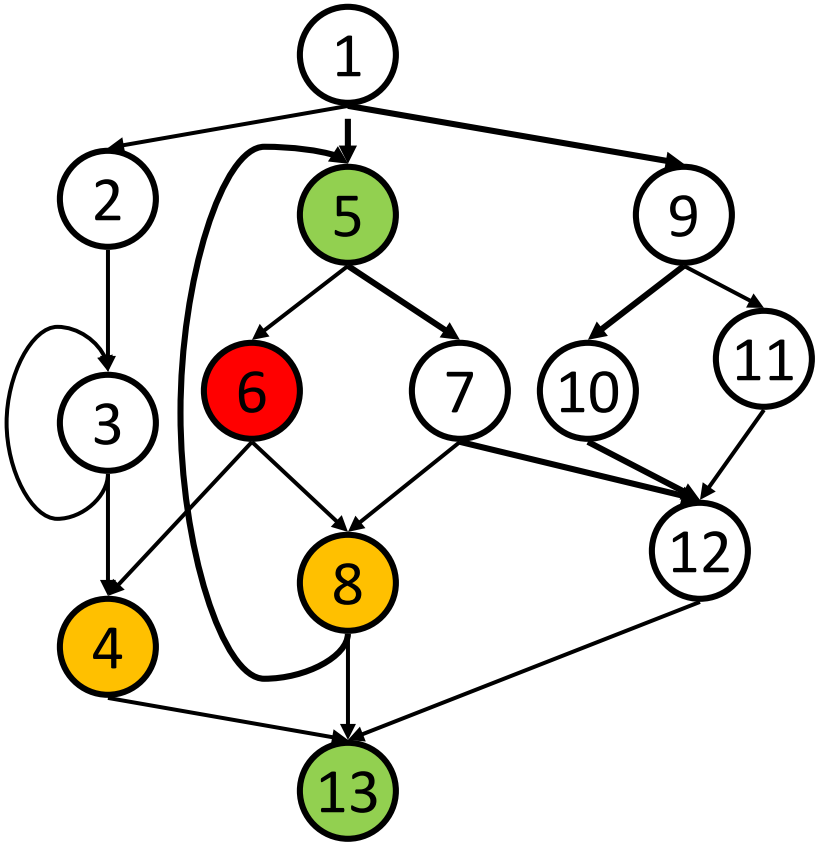


Dominance Frontier Criterion



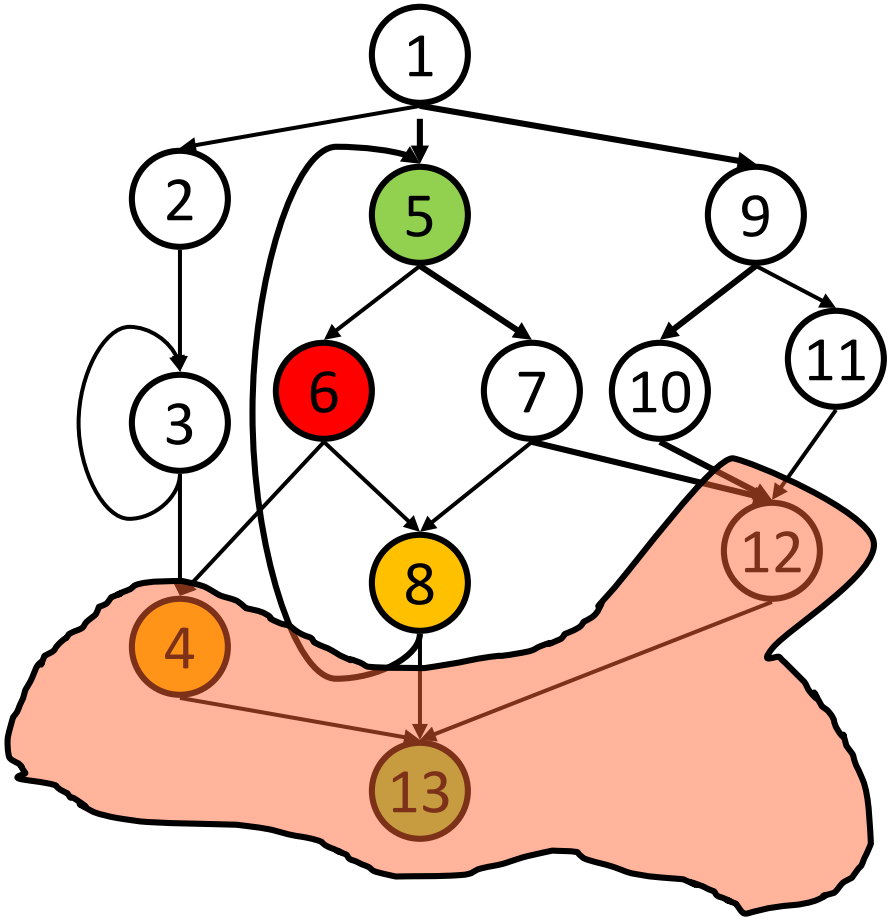
And, Iterating

Dominance Frontier Criterion



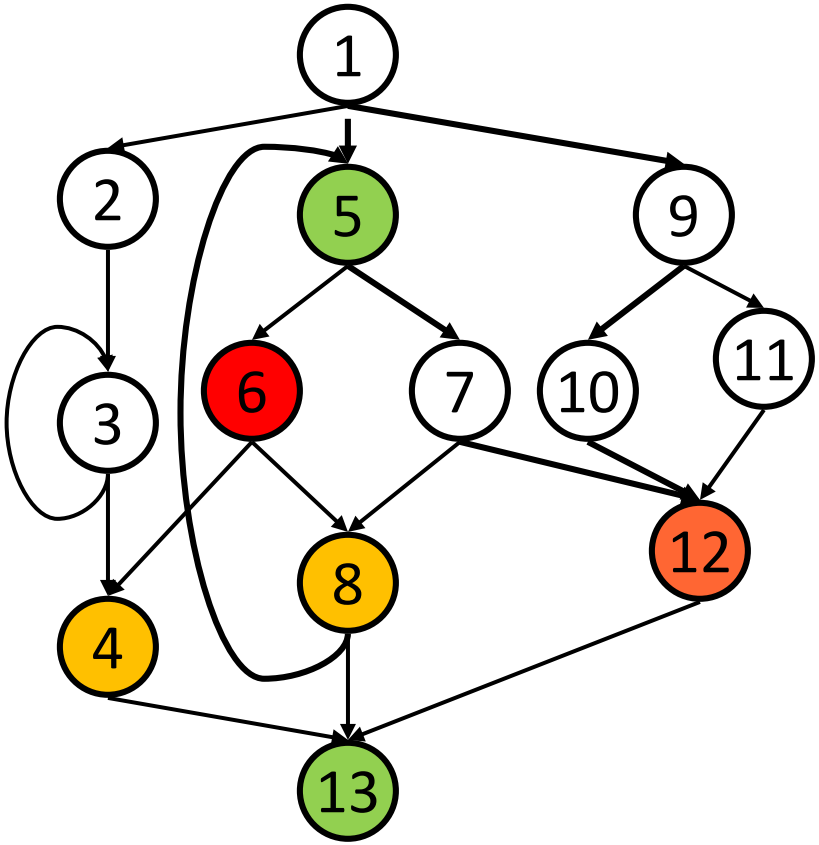
And, Iterating

Dominance Frontier Criterion



And, Iterating

Dominance Frontier Criterion



Done

Using DF to compute minimal SSA

- place all $\Phi()$
- Rename all variables

minimal, but not pruned

Using DF to Place Φ()

- Gather all the defsites of every variable
- Then, for every variable
  - foreach defsite
    - foreach node in DF(defsite)
      - if we haven't put Φ() 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] ∪= {v}
    defsites[v] ∪= {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] ∪ {y}
        if v ∉ orig[y]: W = W ∪ {y}
      }
    }
  }
}
```

Renaming Variables

- Walk the D-tree, renaming variables as you go
- Replace uses with more 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 j^{th} predecessor of Y:

for each phi-function f in Y, **where** the operand of f is 'a'

i = top(Stack[a])

replace j^{th} operand with a_i

for each child of n in D-tree, X:

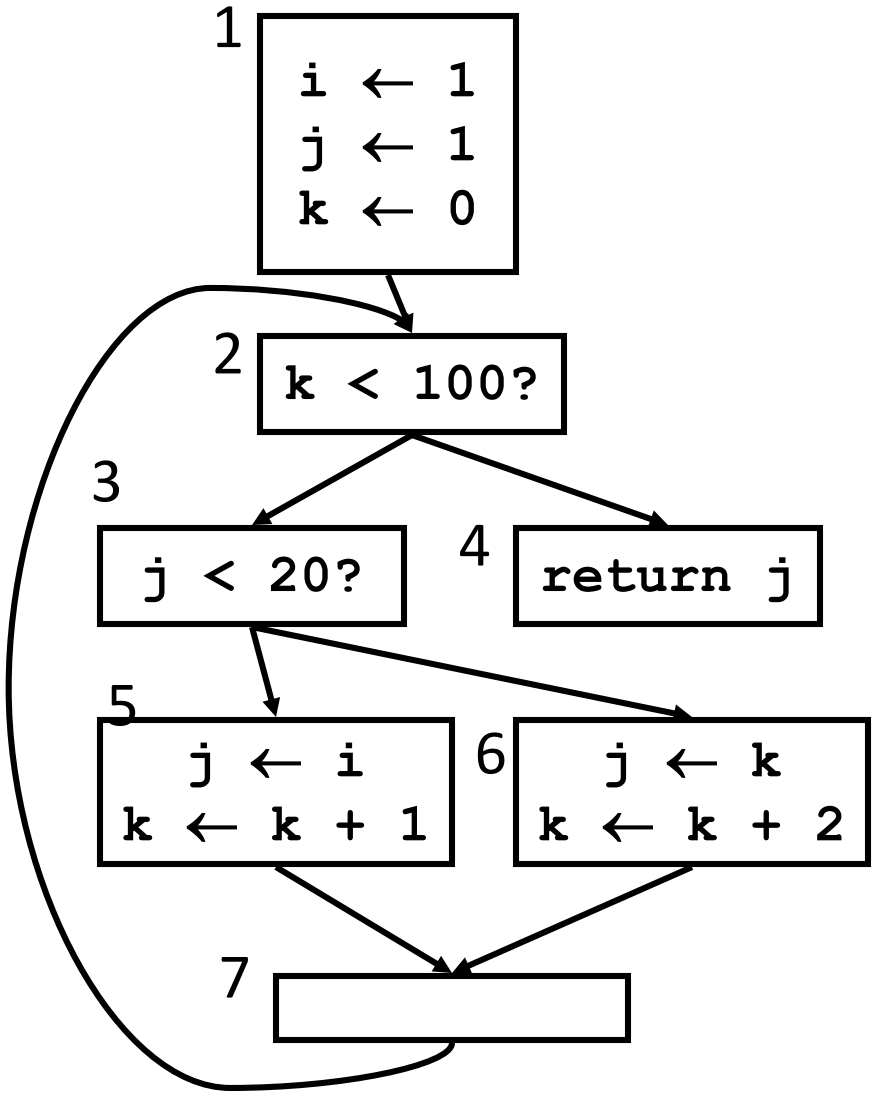
rename(X)

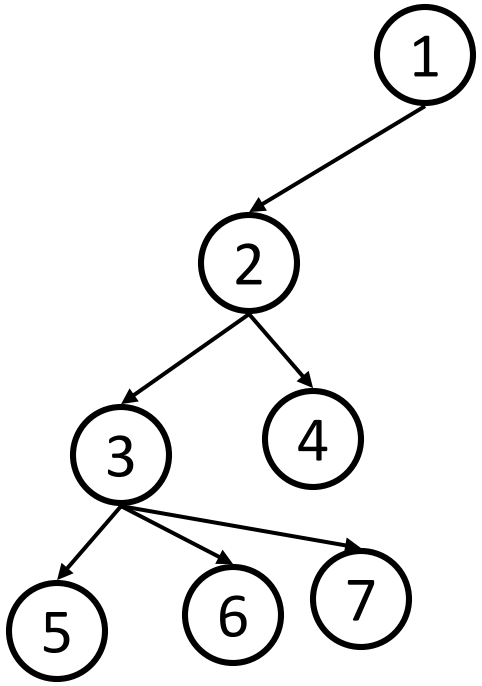
for each instruction S ∈ n:

for each variable v that S defines:

pop Stack[v]

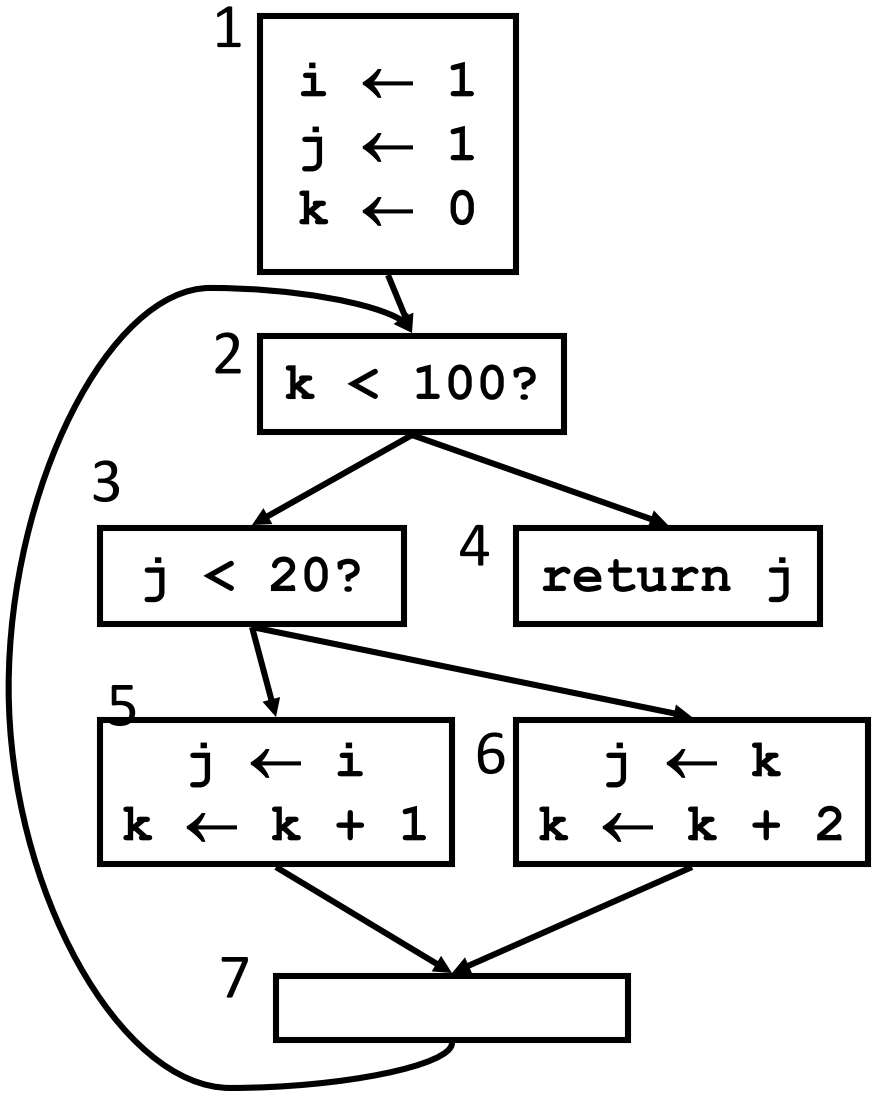
Compute D-tree

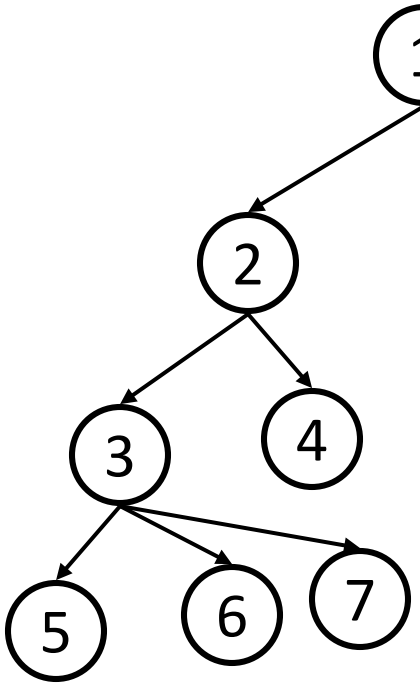




D-tree

Compute Dominance Frontier

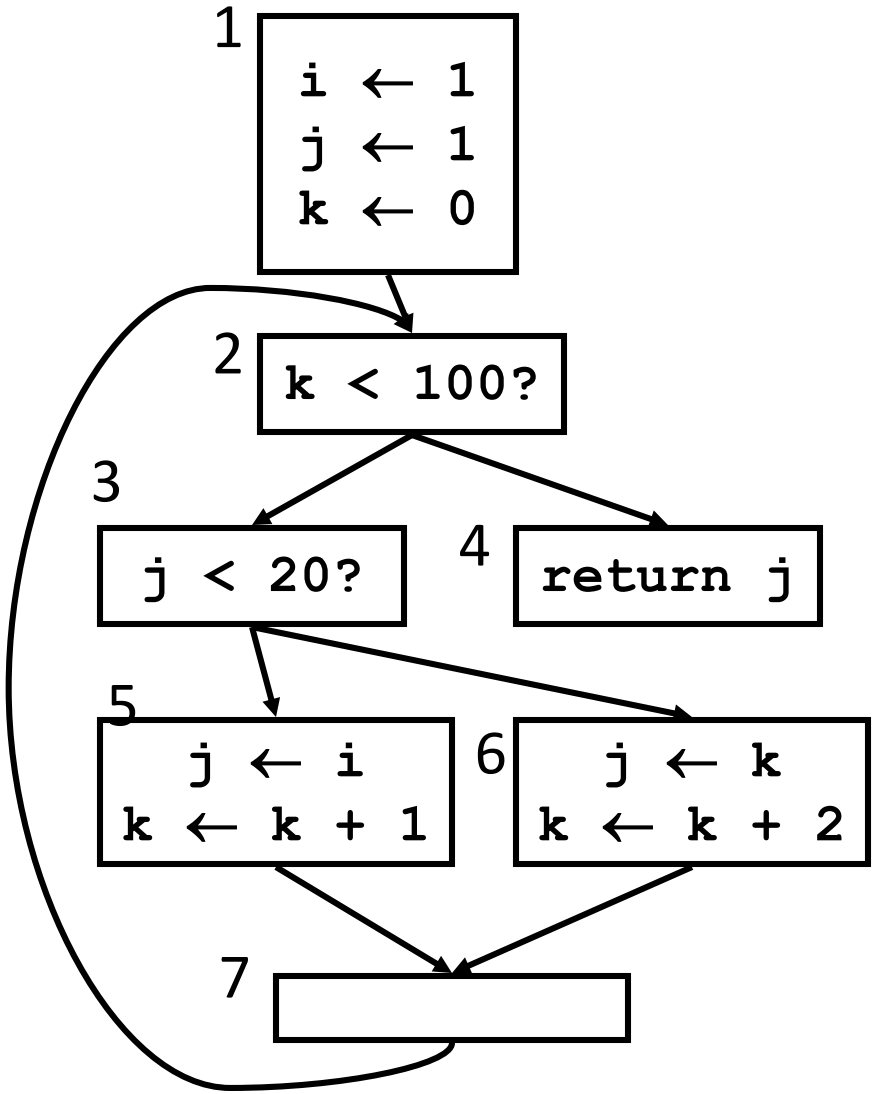




1	{}
2	{2}
3	{2}
4	{}
5	{7}
6	{7}
7	{2}

DFs

Insert Φ()



	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]

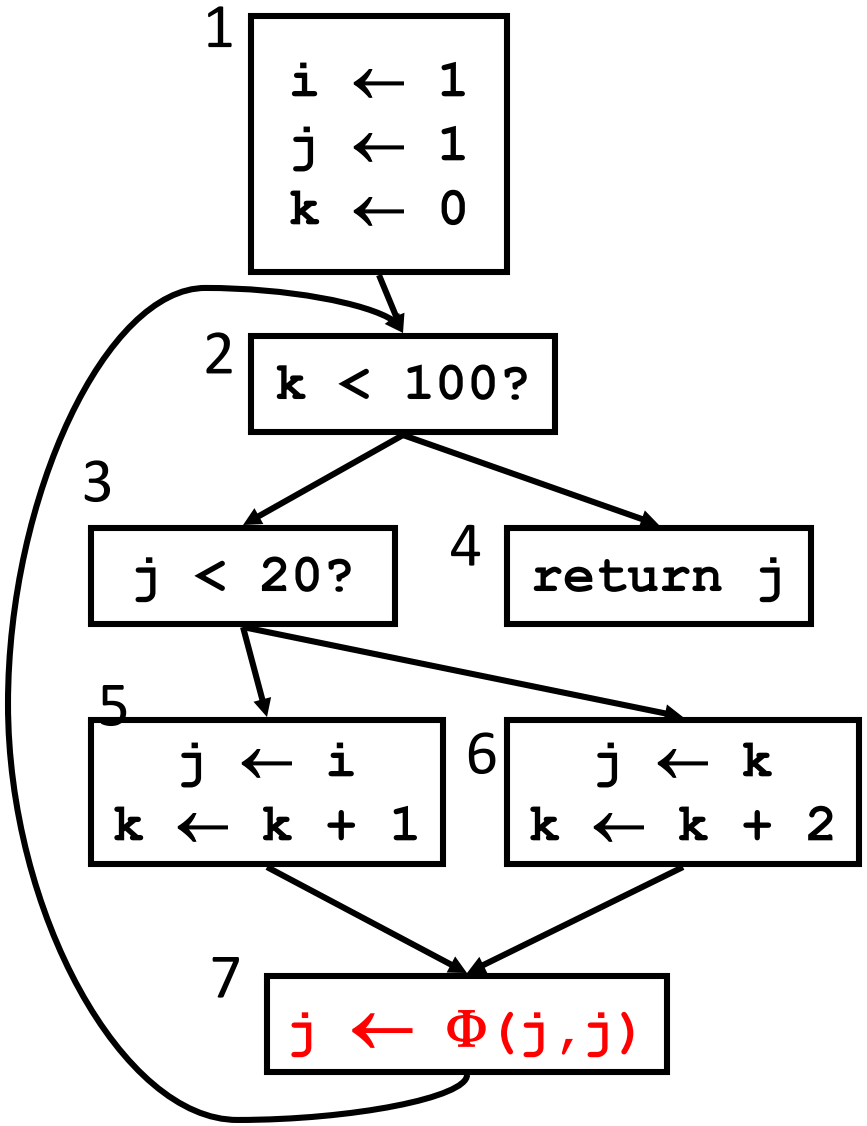
i	{1}
j	{1,5,6}
k	{1,5,6}

var i: W={1}

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

DF{1}, DF{5}

Insert Φ()



	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	{}

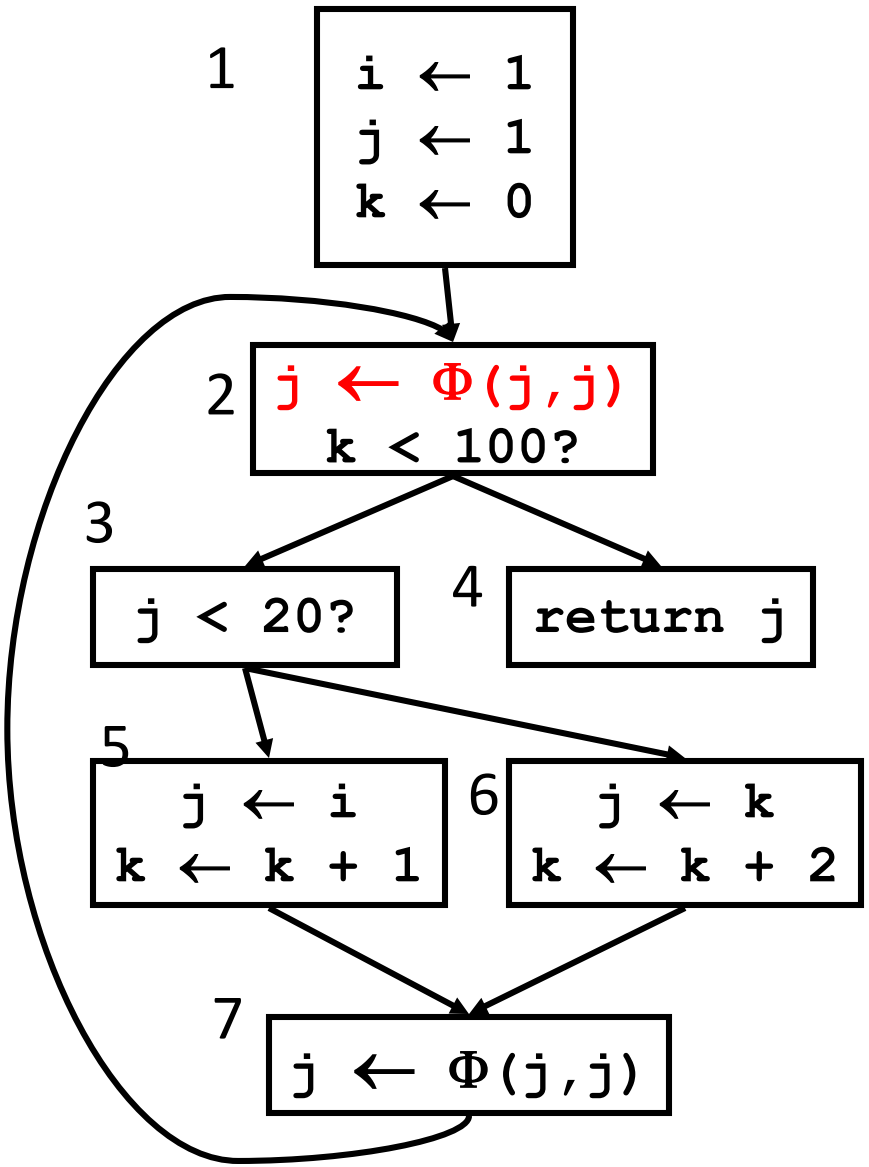
defsites[v]

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

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

DF{1}, DF{5}

Insert Φ()



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

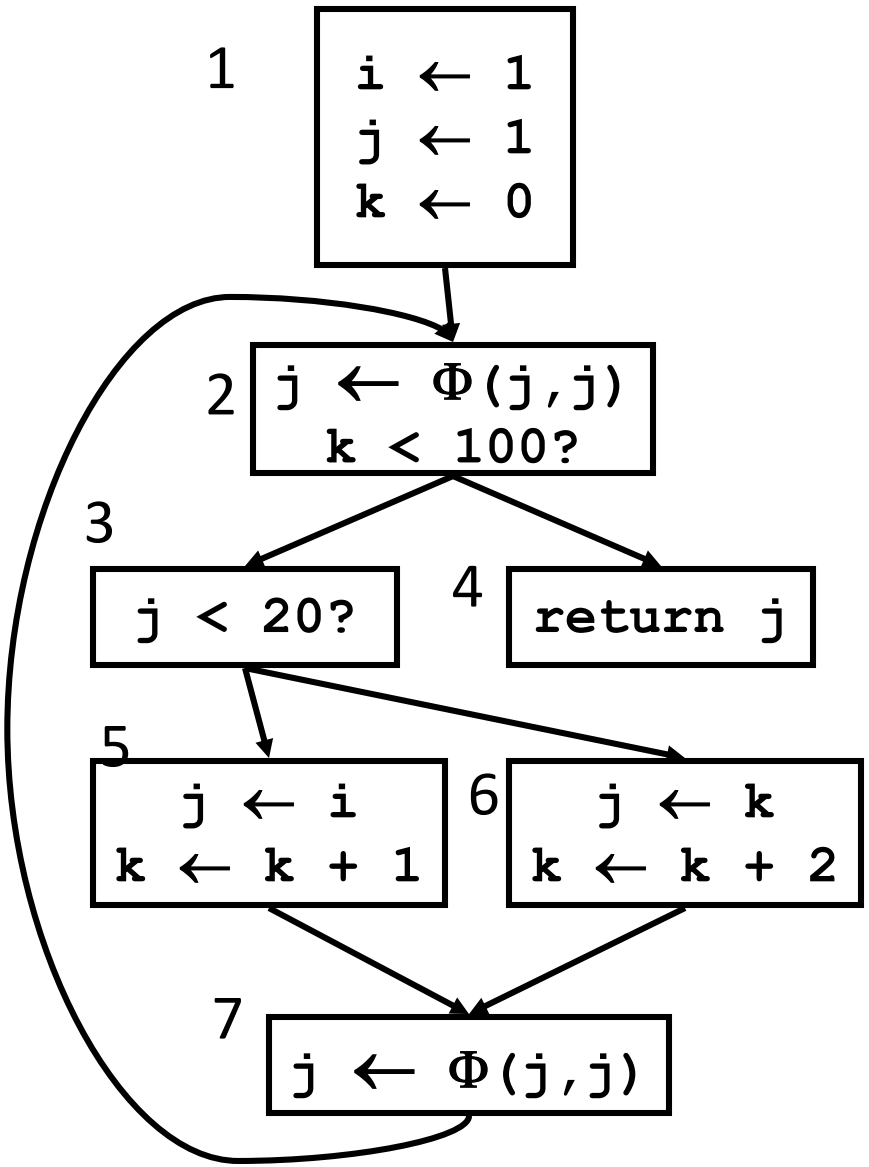
	orig[n]
1	{ i,j,k}
2	{}
3	{}
4	{}
5	{j,k}
6	{j,k}
7	{}

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

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

DF{1}, DF{5}

Insert Φ()



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

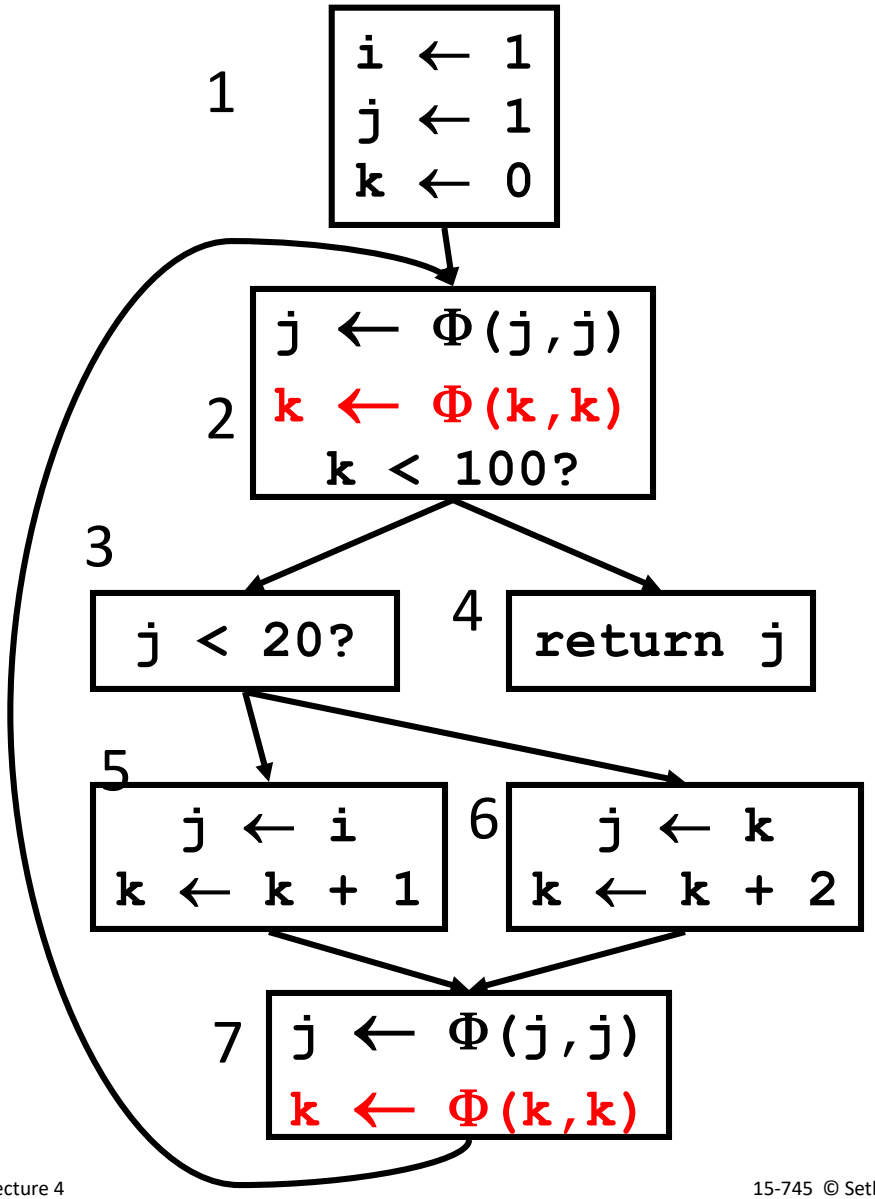
	orig[n]
1	{ i,j,k}
2	{}
3	{}
4	{}
5	{j,k}
6	{j,k}
7	{}

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

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

DF{1}, DF{5}, DF{6}

Insert Φ()



DFs

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

orig[n]

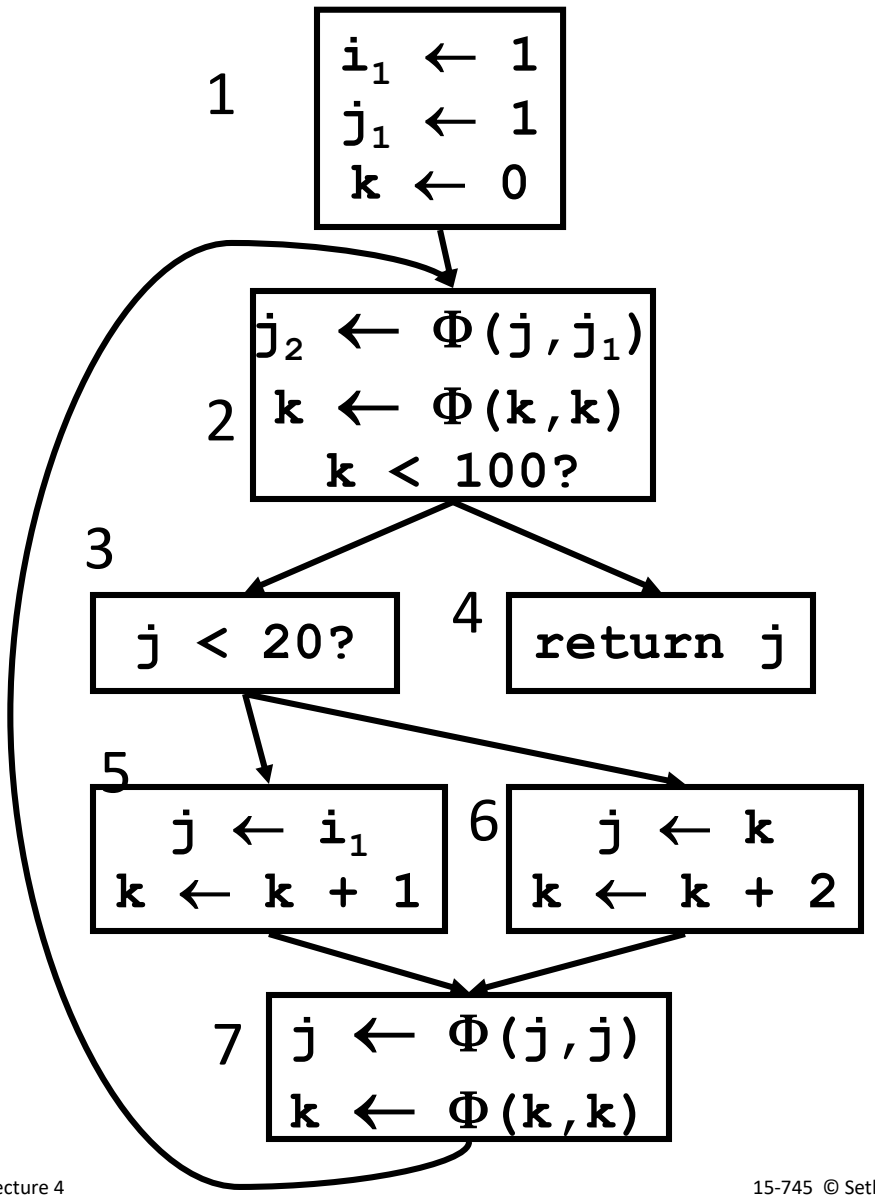
1	{ i,j,k}
2	{}
3	{}
4	{}
5	{j,k}
6	{j,k}
7	{}

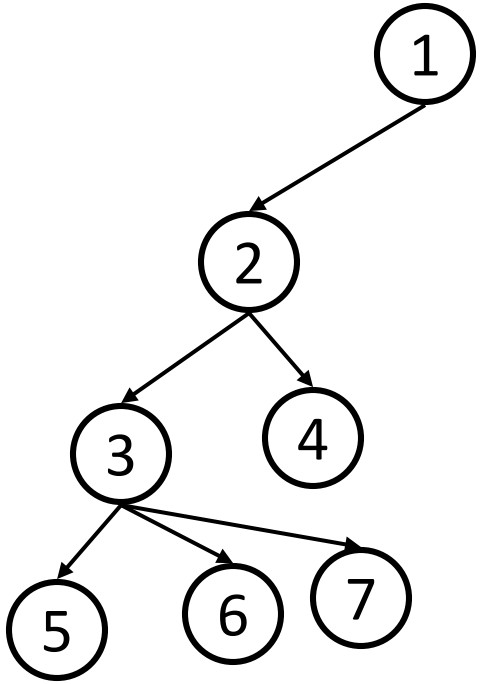
defsites[v]

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

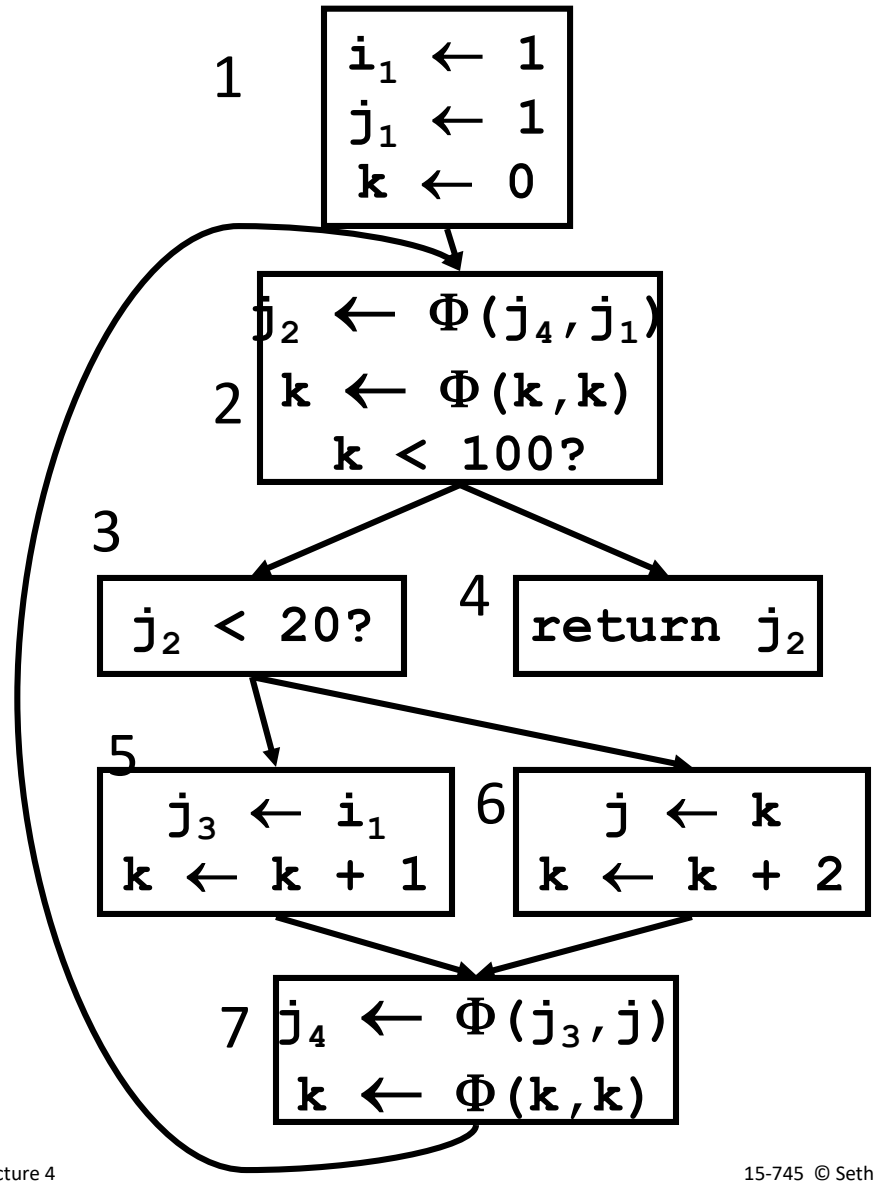
var k: W={1,5,6}

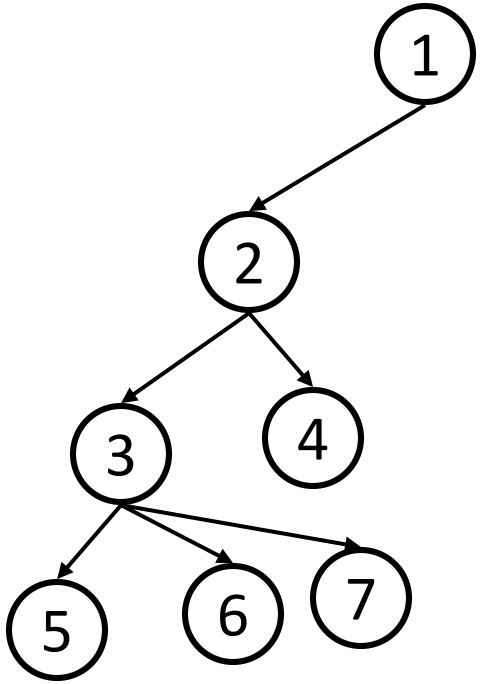
Rename Vars


1
$i_1 \leftarrow 1$
$j_1 \leftarrow 1$
$k \leftarrow 0$
2
$j_2 \leftarrow \Phi(j, j_1)$
$k \leftarrow \Phi(k, k)$
$k < 100?$
3
$j < 20?$
4
return j
5
$j \leftarrow i_1$
$k \leftarrow k + 1$
6
$j \leftarrow k$
$k \leftarrow k + 2$
7
$j \leftarrow \Phi(j, j)$
$k \leftarrow \Phi(k, k)$

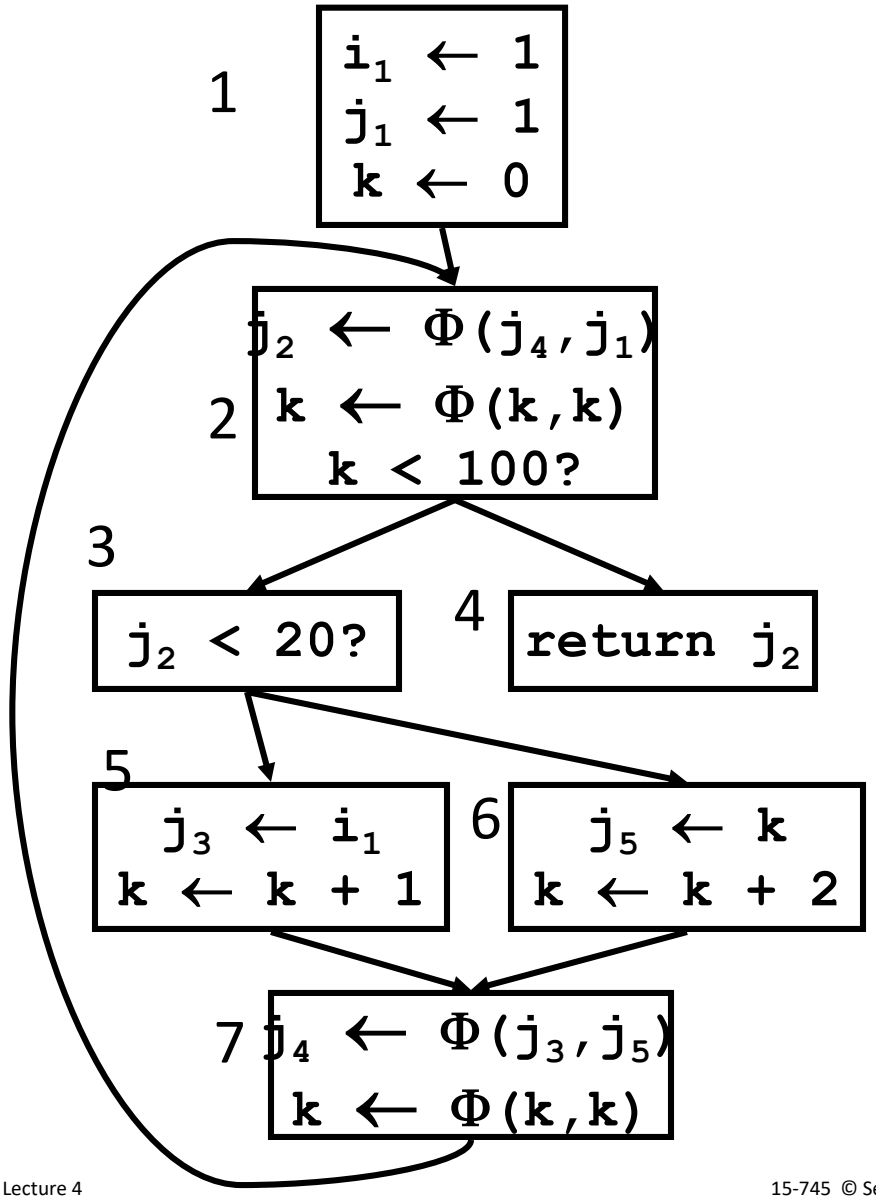

1
2
3
4
5
6
7

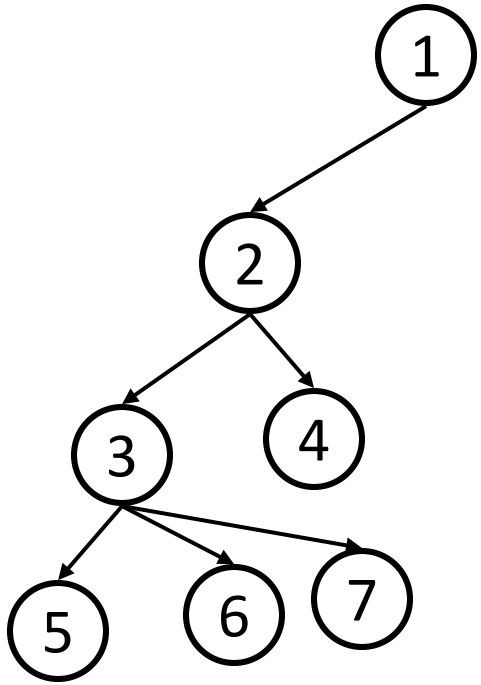
Rename Vars


1
$i_1 \leftarrow 1$
$j_1 \leftarrow 1$
$k \leftarrow 0$
2
$j_2 \leftarrow \Phi(j_4, j_1)$
$k \leftarrow \Phi(k,k)$
$k < 100?$
3
$j_2 < 20?$
4
return j_2
5
$j_3 \leftarrow i_1$
$k \leftarrow k + 1$
6
$j \leftarrow k$
$k \leftarrow k + 2$
7
$j_4 \leftarrow \Phi(j_3, j)$
$k \leftarrow \Phi(k,k)$

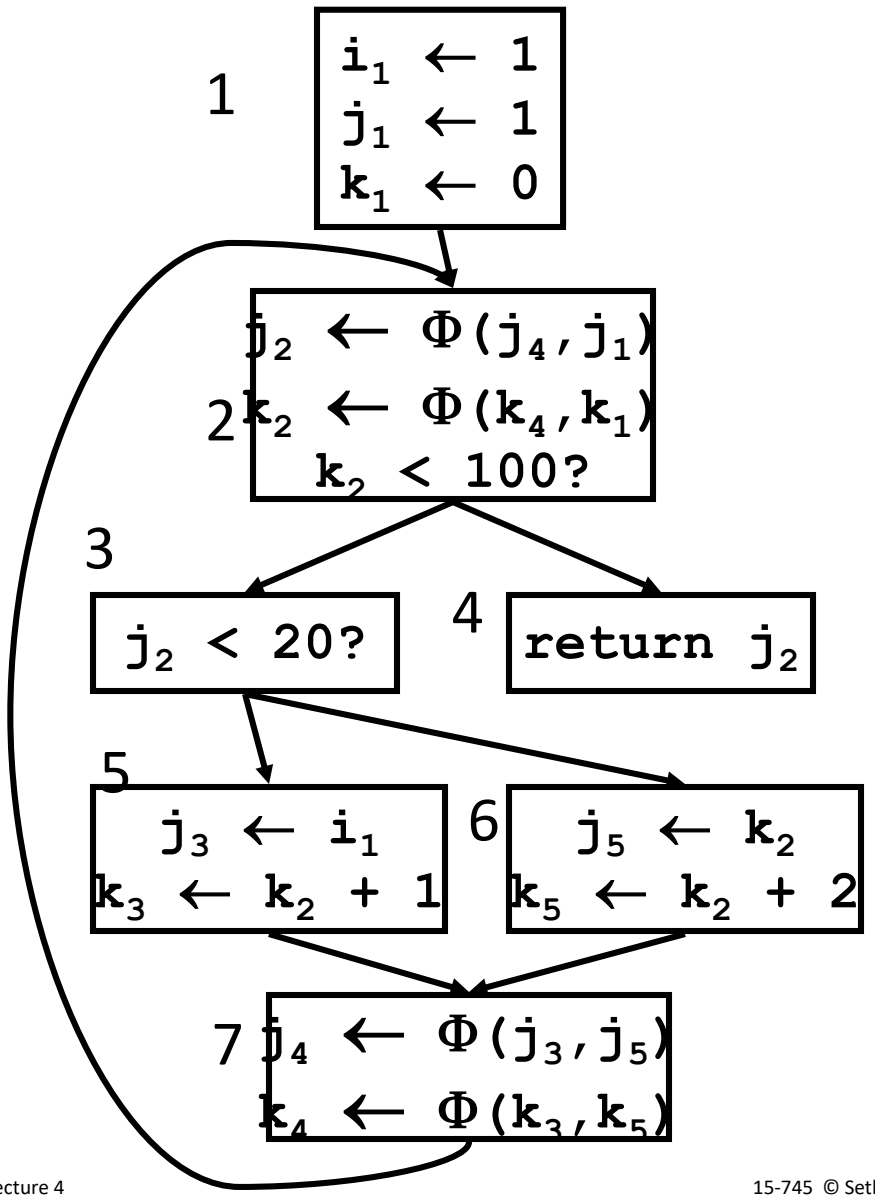

1
2
3
4
5
6
7

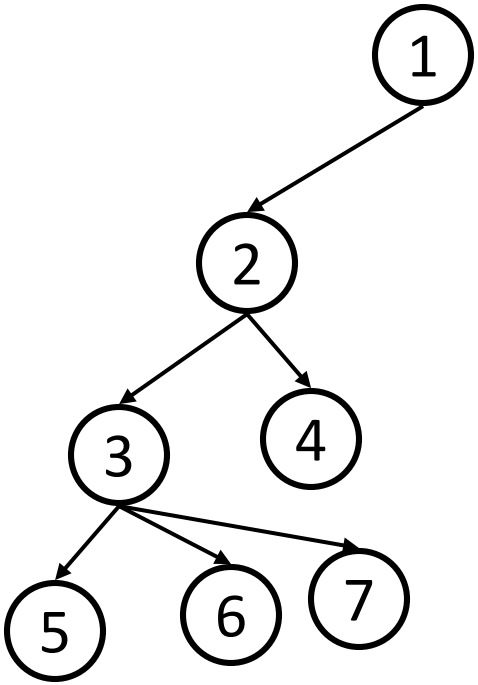
Rename Vars


1
$i_1 \leftarrow 1$
$j_1 \leftarrow 1$
$k \leftarrow 0$
2
$j_2 \leftarrow \Phi(j_4, j_1)$
$k \leftarrow \Phi(k,k)$
$k < 100?$
3
$j_2 < 20?$
4
return j_2
5
$j_3 \leftarrow i_1$
$k \leftarrow k + 1$
6
$j_5 \leftarrow k$
$k \leftarrow k + 2$
7
$j_4 \leftarrow \Phi(j_3, j_5)$
$k \leftarrow \Phi(k,k)$


1
2
3
4
5
6
7

Rename Vars


1
$i_1 \leftarrow 1$
$j_1 \leftarrow 1$
$k_1 \leftarrow 0$
2
$j_2 \leftarrow \Phi(j_4, j_1)$
$k_2 \leftarrow \Phi(k_4, k_1)$
$k_2 < 100?$
3
$j_2 < 20?$
4
return j_2
5
$j_3 \leftarrow i_1$
$k_3 \leftarrow k_2 + 1$
6
$j_5 \leftarrow k_2$
$k_5 \leftarrow k_2 + 2$
7
$j_4 \leftarrow \Phi(j_3, j_5)$
$k_4 \leftarrow \Phi(k_3, k_5)$


1
2
3
4
5
6
7

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

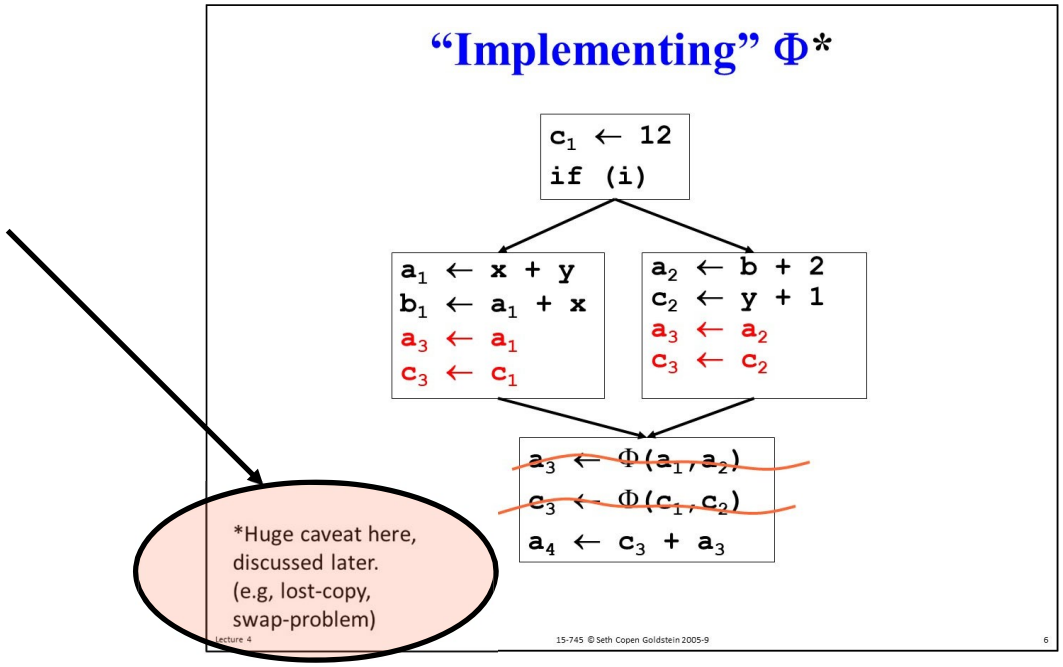
Summary – getting into

- 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!

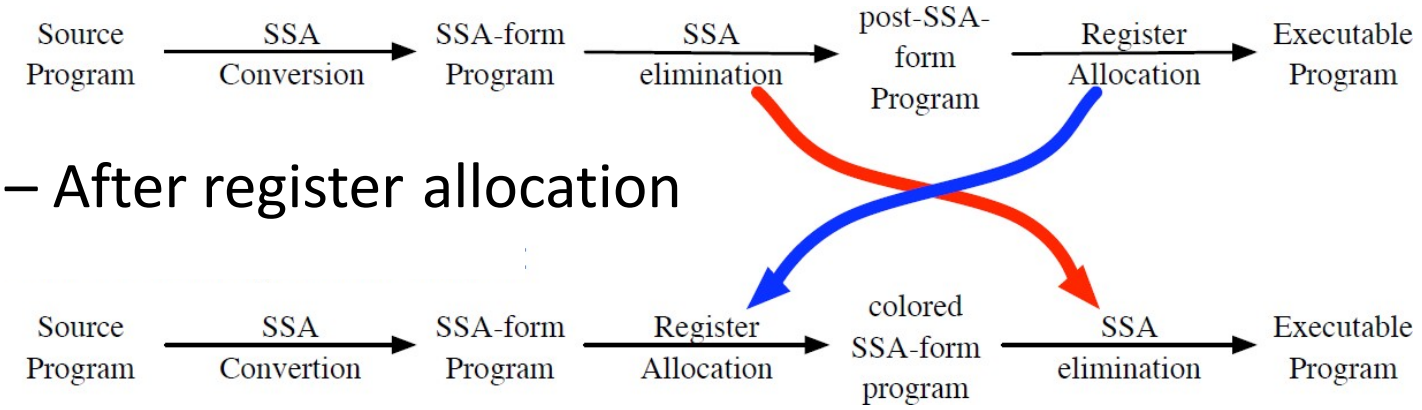
But, …

- Eventually, have to get out of SSA and deconstruct all the Φ-functions
- Recall from Lecture 4



But, …

- Eventually, have to get out of SSA and deconstruct all the Φ-functions
- Two choices:
  - Before register allocation
  - After register allocation



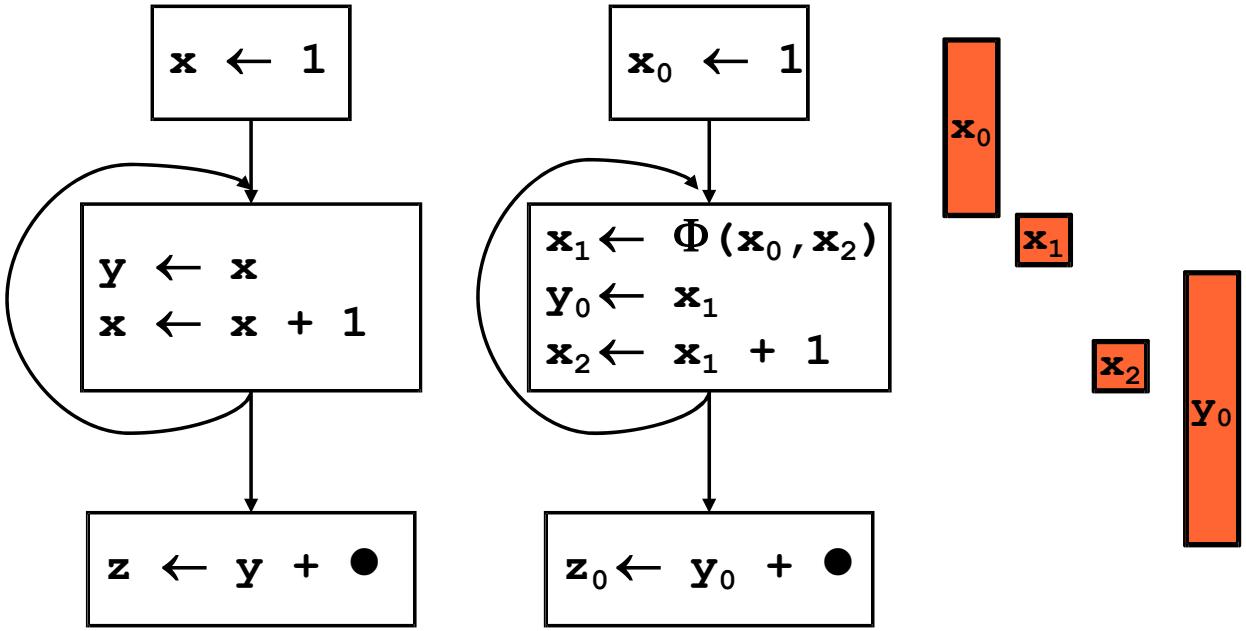
When to deconstruct SSA?

- Before register allocation
  - deconstructing SSA can introduce lots of copies which are easier to eliminate without register constraints
- After register allocation ☺
  - Enables decoupled register allocation
    - spill, color, coalesce
  - Φ-functions may have sources which are registers and memory.
  - Complicated by code-motion optimizations

Conventional-SSA

- Initial Conversion to SSA creates
  "Conventional SSA"
- Main feature:
  - variables involved in a Φ-function never interfere
  - Thus, can allocate to a single **resource**
    - the same register
    - the same frame slot (in case of spill)
- However, code motion can destroy this property

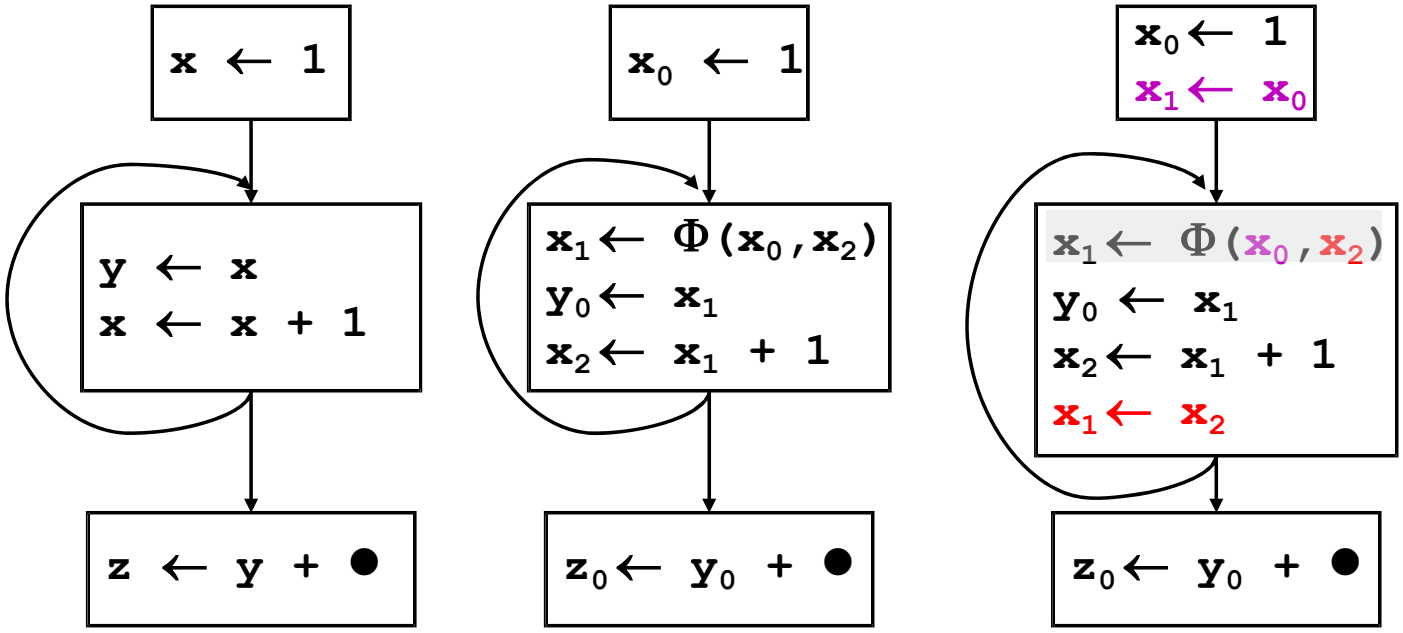
→ SSA



Original

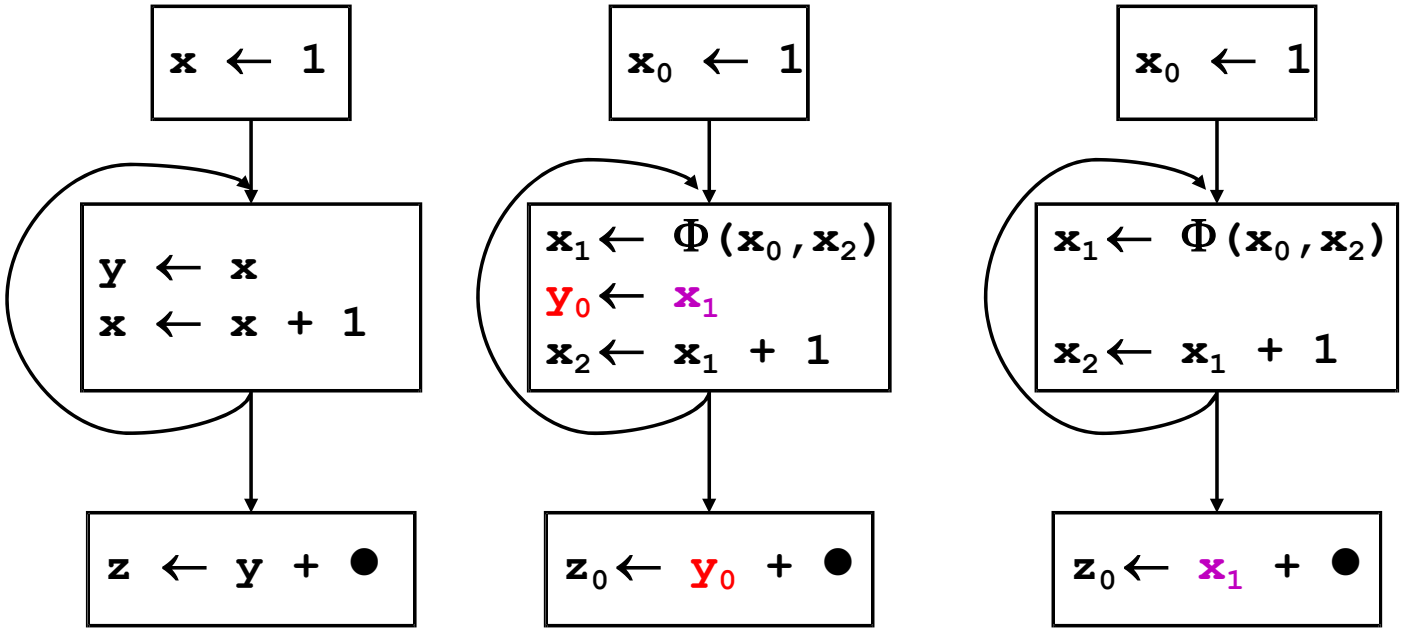
In SSA

Leaving SSA



Original In SSA Out of SSA

Copy Propagation

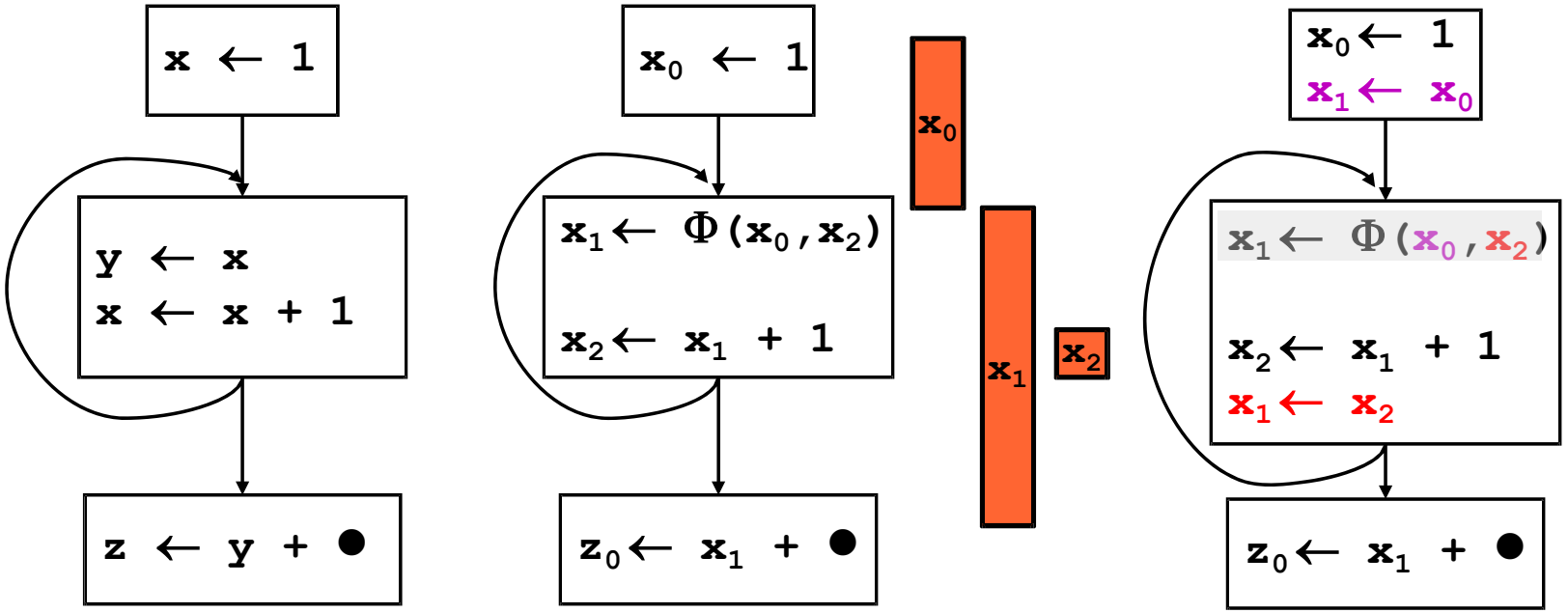


Original

In SSA

Post Propagate

Leaving SSA After Copy Folding



Original

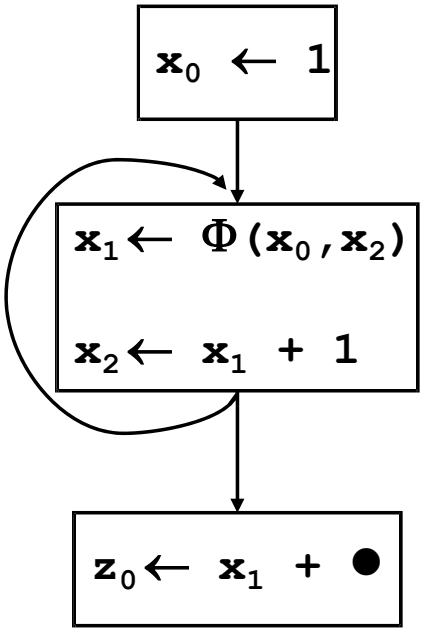
Post Prop

Out of SSA

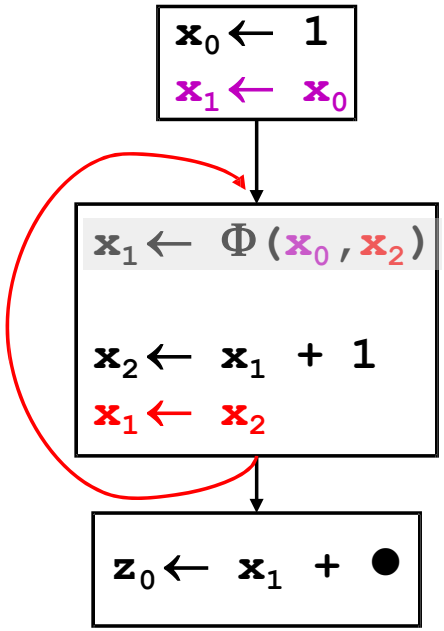
"Lost Copy" Problem

Critical Edges

critical edge



Copy Propagated



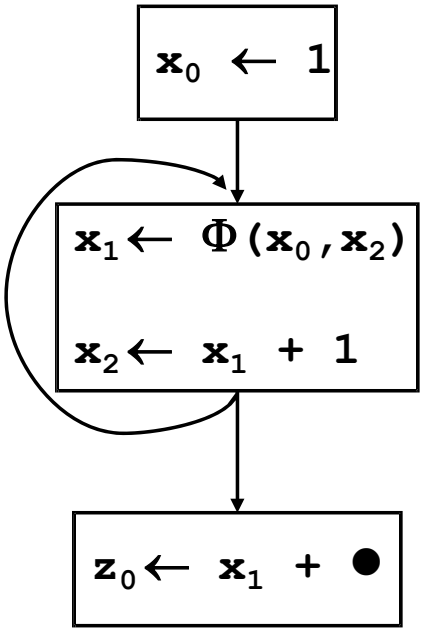
INCORRECT

A critical edge is an edge a→b where

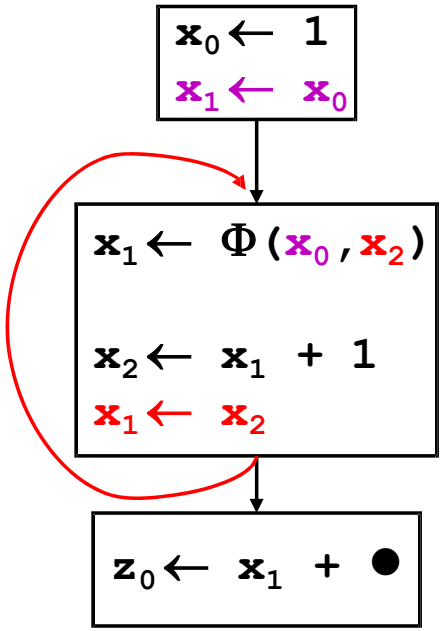
- a has > 1 successor and
- b has > 1 predecessor.

Critical Edges

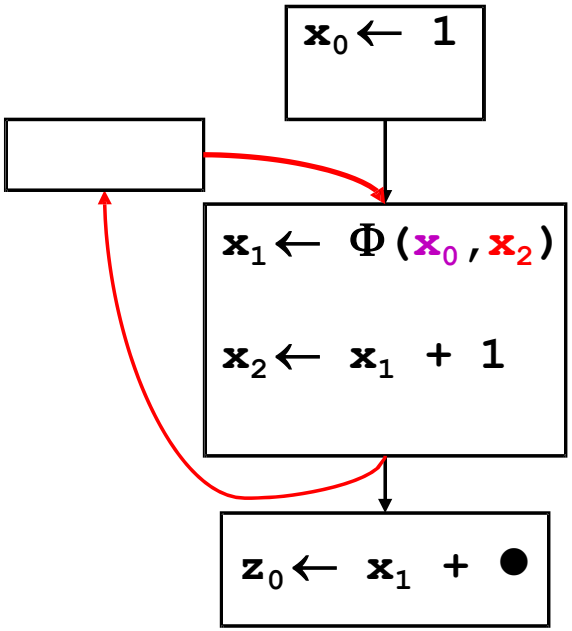
critical edge



Copy Folded



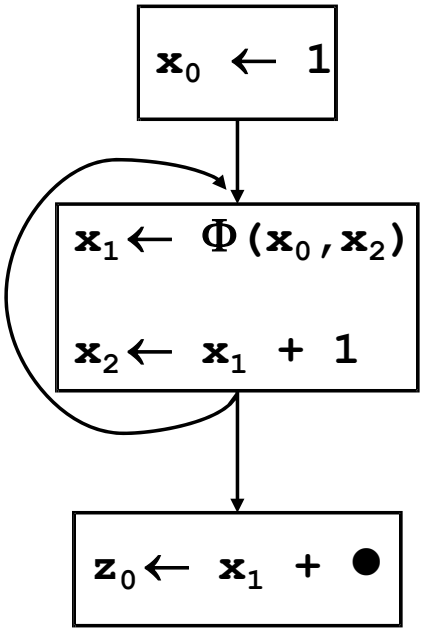
INCORRECT



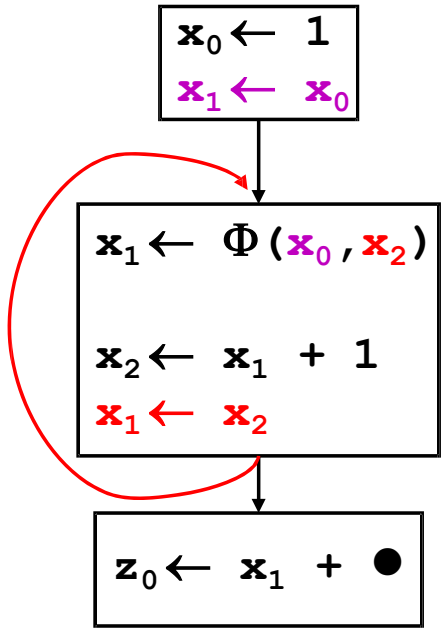
Inserting block on critical edge

Critical Edges

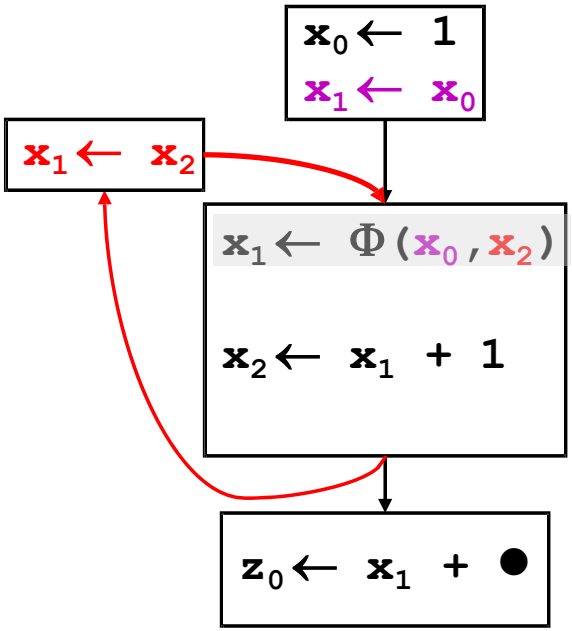
critical edge



Copy Folded



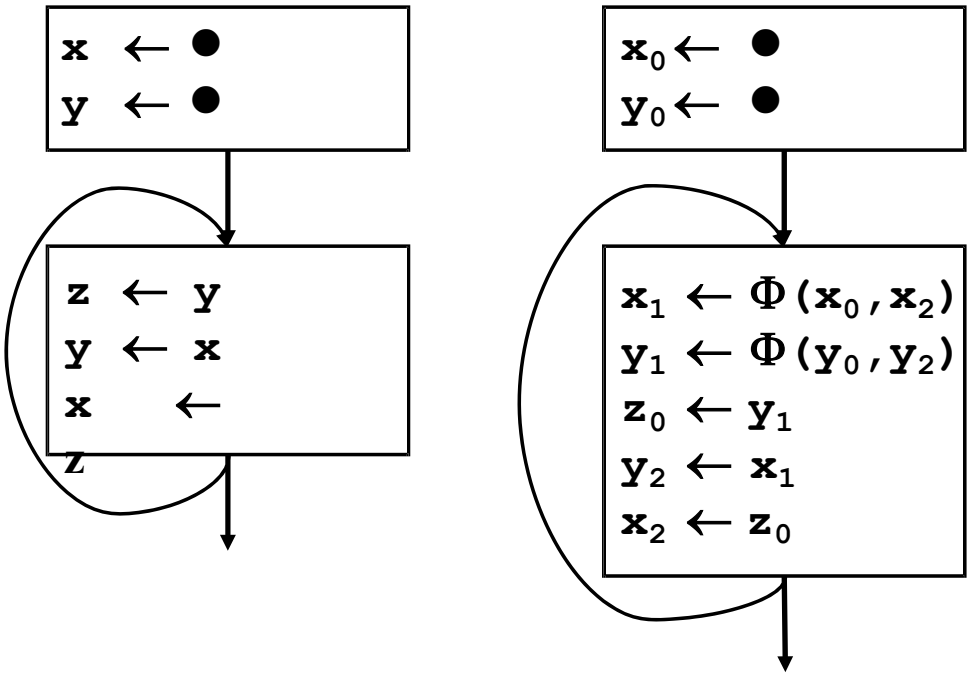
INCORRECT



Inserting block on critical edge, Then deconstruct

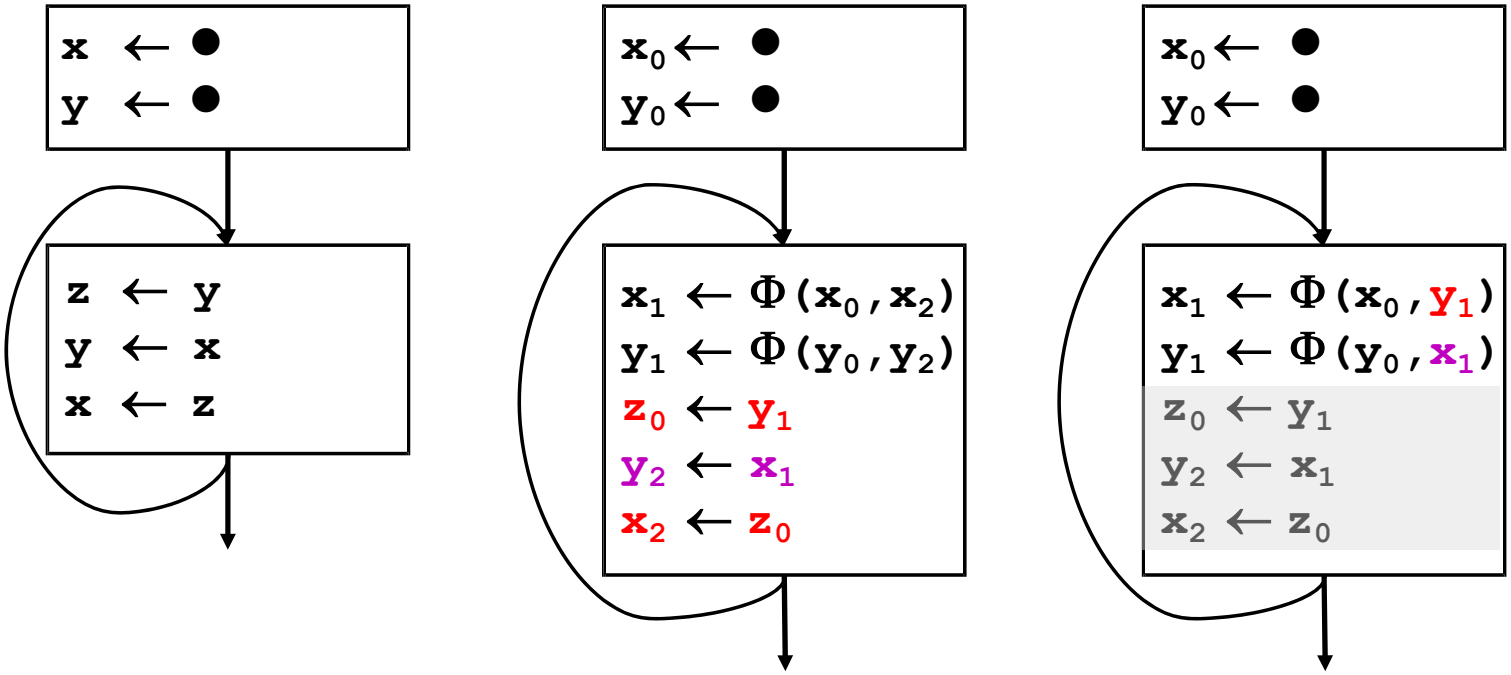
Semantics of Φ-functions

- Semantics of Φ-functions requires copies to be done in parallel.



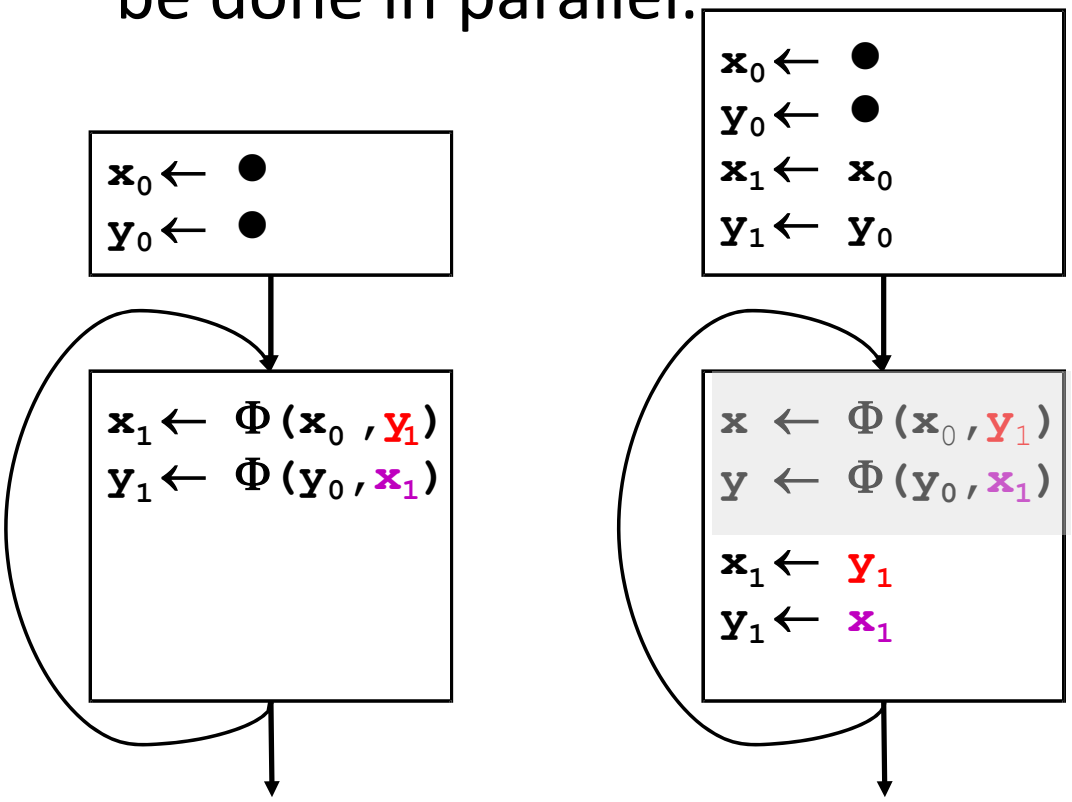
Semantics of Φ-functions

- Semantics of Φ-functions requires copies to be done in parallel.



Semantics of Φ-functions

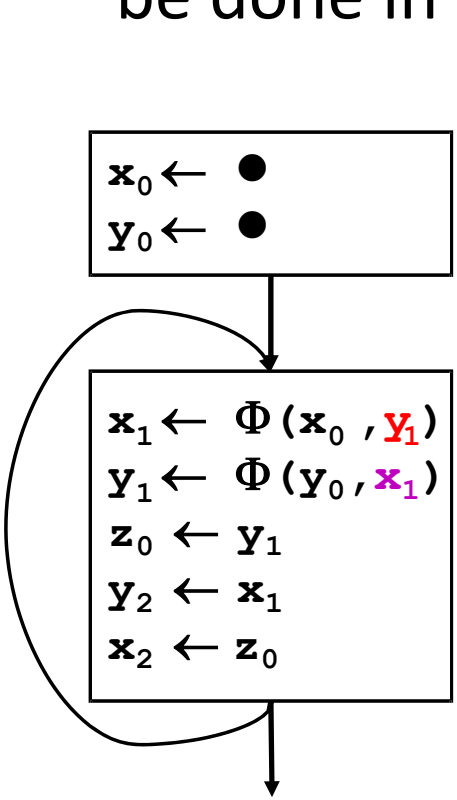
- Semantics of Φ-functions requires copies to be done in parallel.

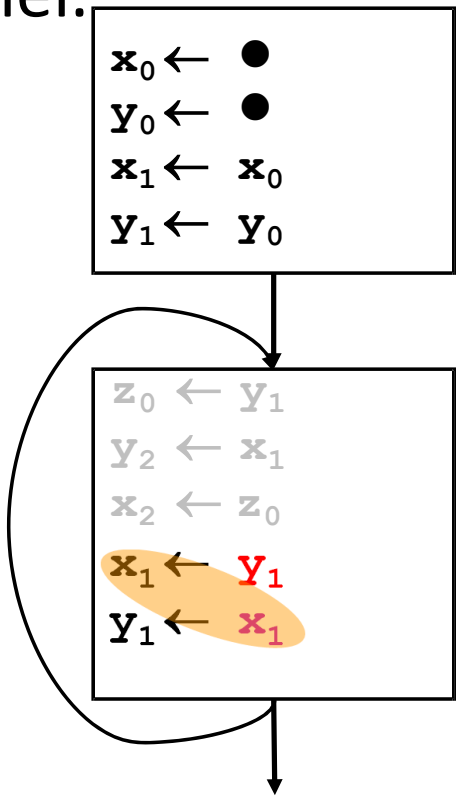


INCORRECT

Semantics of Φ-functions

- Semantics of Φ-functions requires copies to be done in parallel.



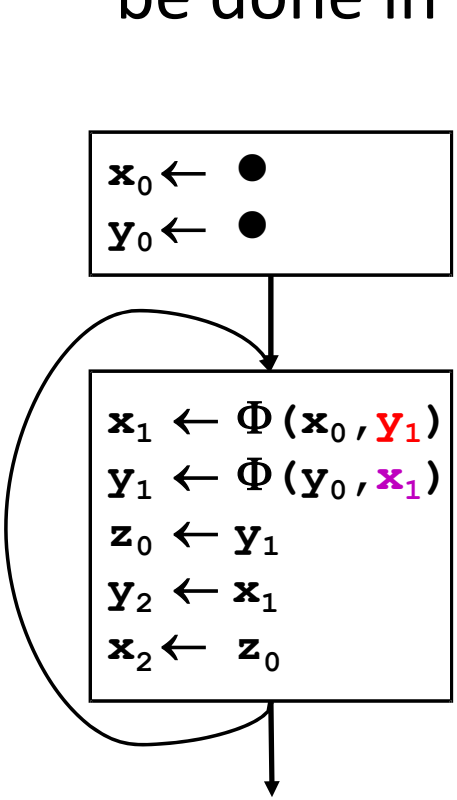


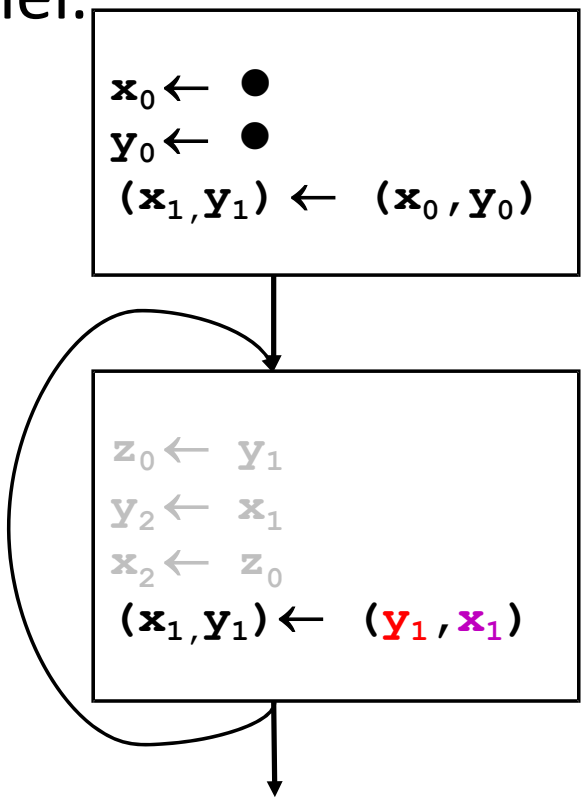
INCORRECT

Lost value of x_1! because did Φ assignments sequentially

Semantics of Φ-functions

- Semantics of Φ-functions requires copies to be done in parallel.





Using Parallel copies

Impact of Spilling

- What happens when we spill a Φ related variable?
- For example:
  - $\mathbf{r \leftarrow \Phi(r, m_0)}$
  - $\mathbf{m_1 \leftarrow \Phi(r, m_0)}$
- Could require memory-memory move after deconstructing SSA

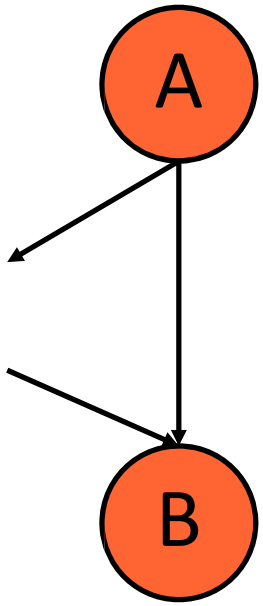
Solution

- Critical Edge Splitting
- Convert back to Conventional-SSA (CSSA)
- Register Allocation
  - Build interference graph
  - pre-spilling
  - coloring
- Deconstruct SSA
  - put parallel-copies in predecessors
  - Eliminate parallel copies
- Coalescing

Note: we changed traditional register allocation sequence

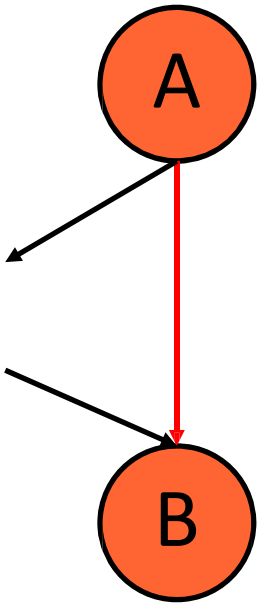
Removing a Critical Edge

- A critical edge is an edge a→b where
  - a has > 1 successor and
  - b has > 1 predecessor.
- For each edge (a,b) in CFG where a > 1 succ and b > 1 pred
  - Insert new block Z
  - replace (a,b) with
    - (a,z) and (z,b)



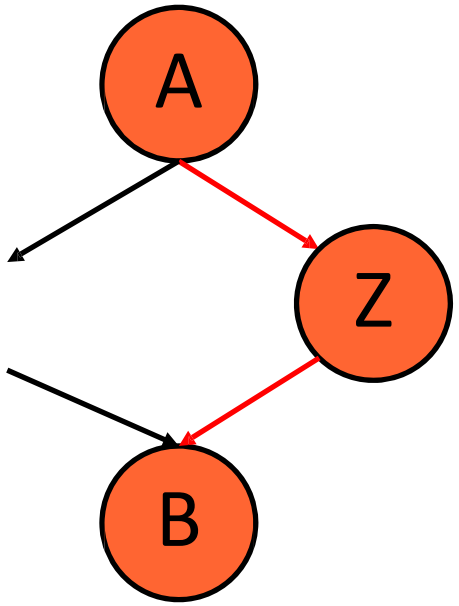
Removing a Critical Edge

- A critical edge is an edge a→b where
  - a has > 1 successor and
  - b has > 1 predecessor.
- For each edge (a,b) in CFG where a > 1 succ and b > 1 pred
  - Insert new block Z
  - replace (a,b) with
    - (a,z) and (z,b)



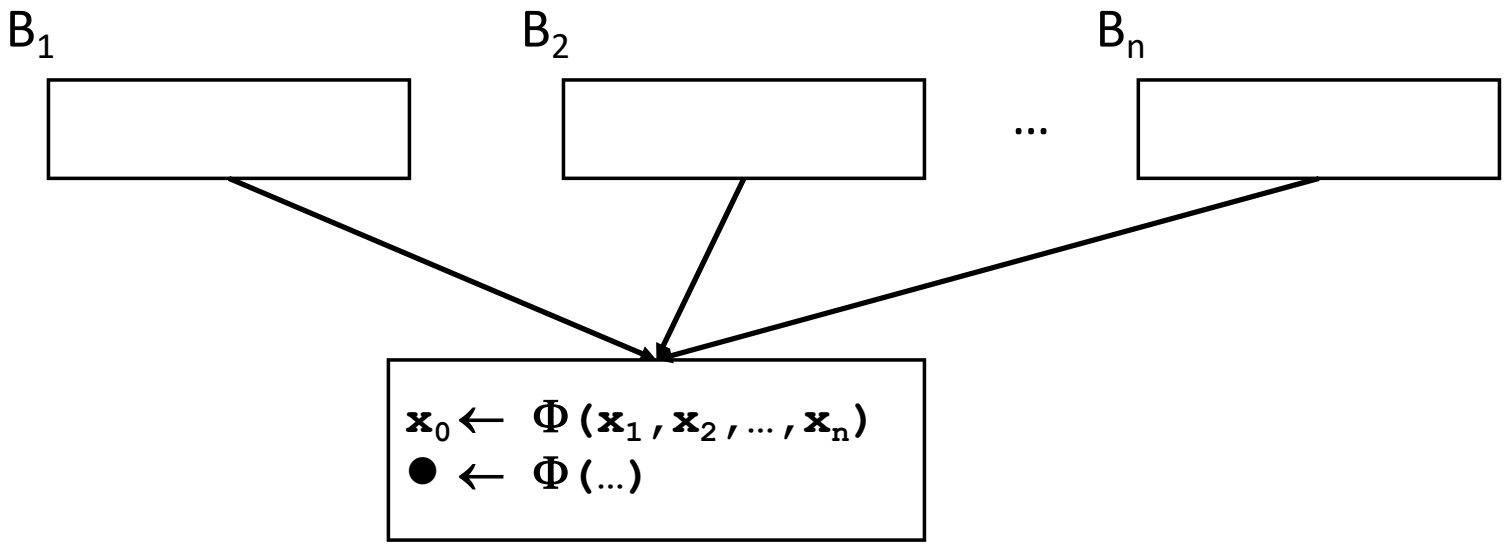
Removing a Critical Edge

- A critical edge is an edge a→b where
  - a has > 1 successor and
  - b has > 1 predecessor.
- For each edge (a,b) in CFG where a > 1 succ and b > 1 pred
  - Insert new block Z
  - replace (a,b) with
    - (a,z) and (z,b)



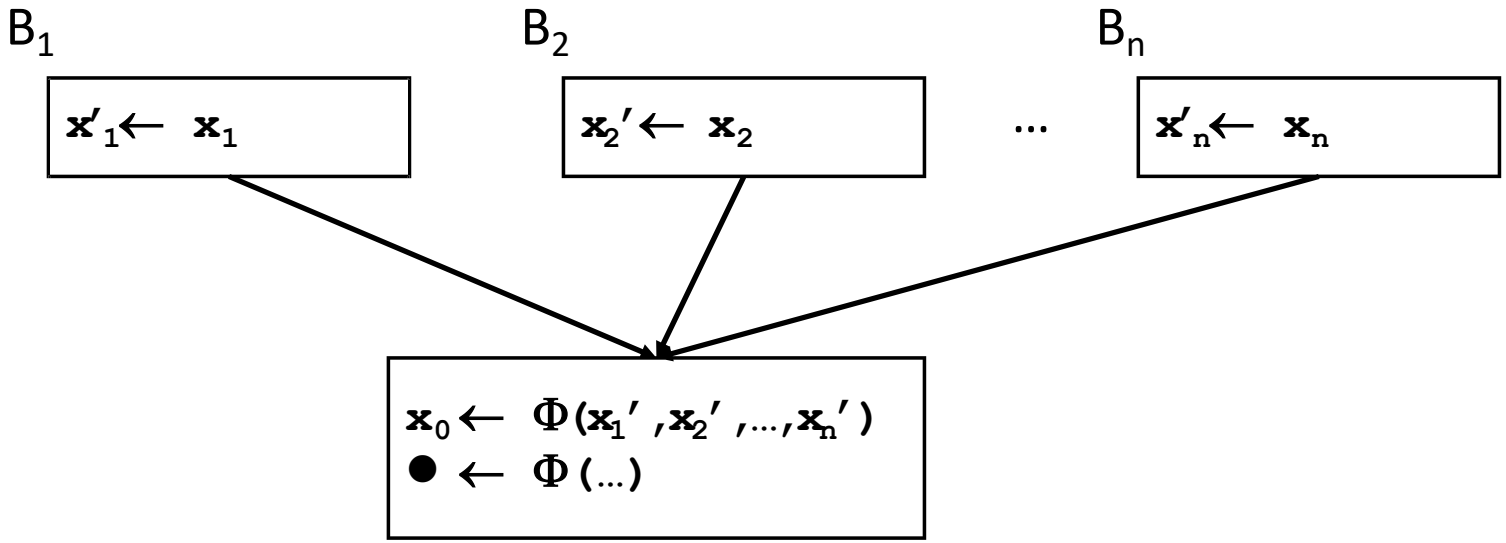
Converting to CSSA

- Goal is to ensure that all Φ related variables do NOT interfere
  - insert copies to (possibly) split live ranges



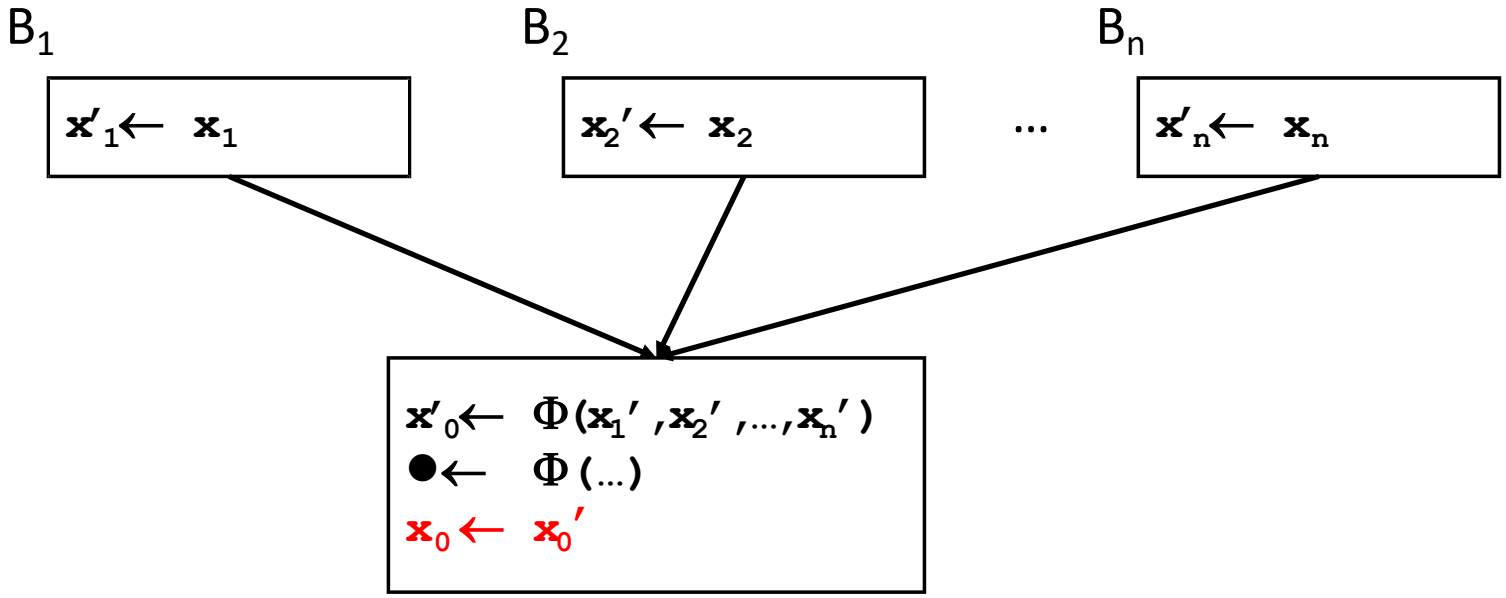
Converting to CSSA

- Goal is to ensure that all Φ related variables do NOT interfere
  - For each argument, insert copy at end of predecessor block and use copy in Φ-function



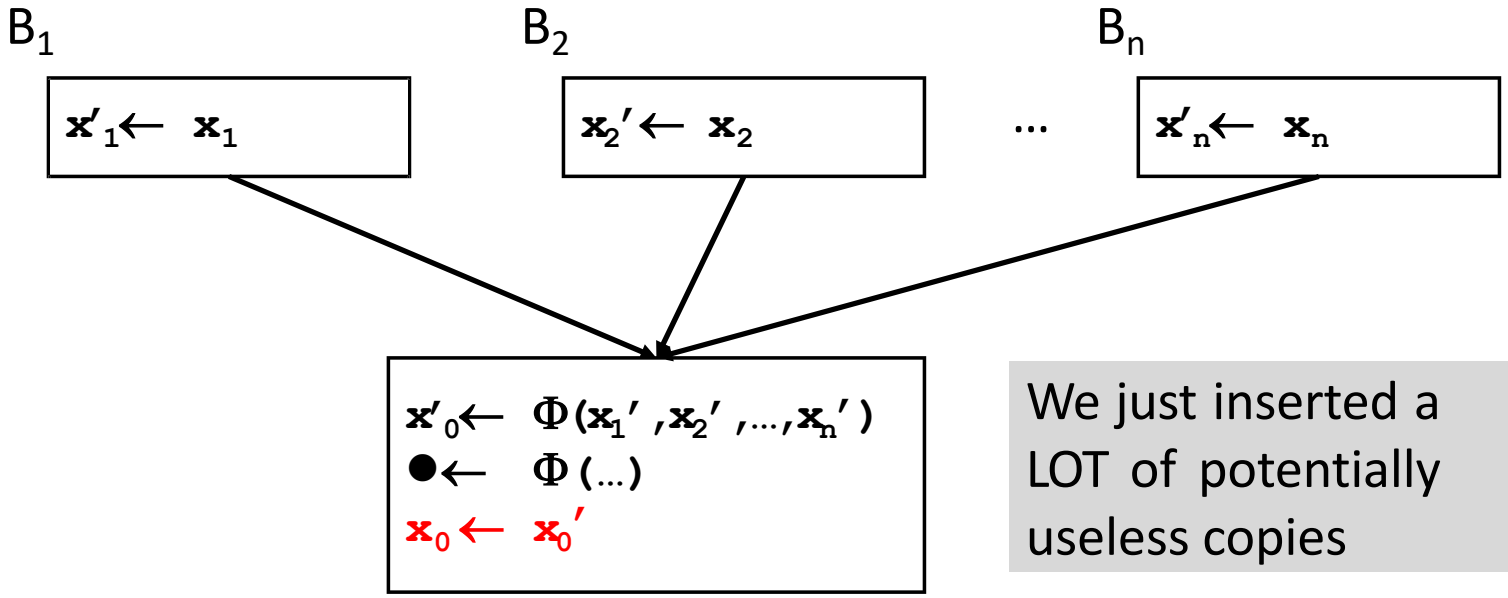
Converting to CSSA

- Goal is to ensure that all Φ related variables do NOT interfere
  - For each argument, insert copy at end of predecessor block and use copy in Φ-function
  - Rename destination
  - Insert copy AFTER all Φ-functions in block



Converting to CSSA

- Goal is to ensure that all Φ related variables do NOT interfere
  - For each argument, insert copy at end of predecessor block and use copy in Φ-function
  - Rename destination
  - Insert copy AFTER all Φ-functions in block



We just inserted a LOT of potentially useless copies

Register Allocation on CSSA

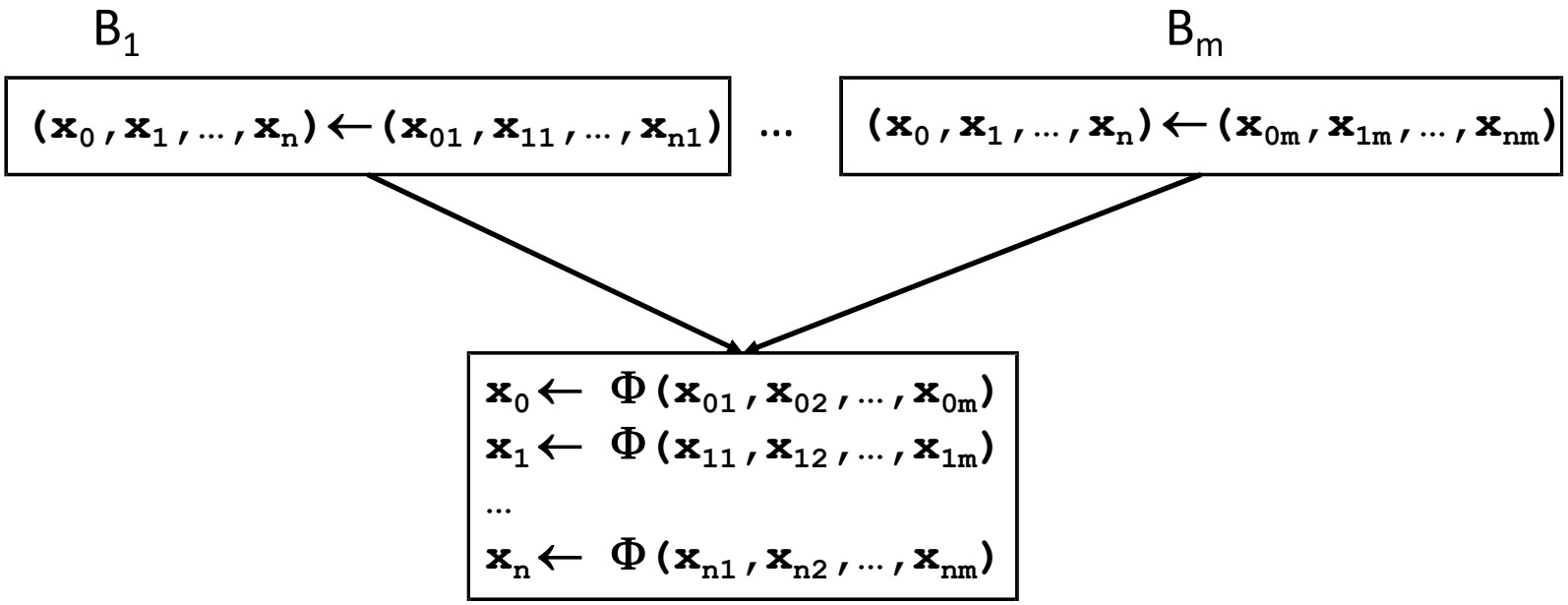
- Build interference graph
- Pre-spill to make it colorable
  - If spill a Φ-related variable, make sure all from same Φ-function use same memory slot!
  - Why do we know this is ok?
  - [Cheat 1: if you spill one, spill them all]
- Color using SEO

Elimination of Φ-functions

- Put parallel copies in predecessor blocks
- Sequentialize the parallel copies

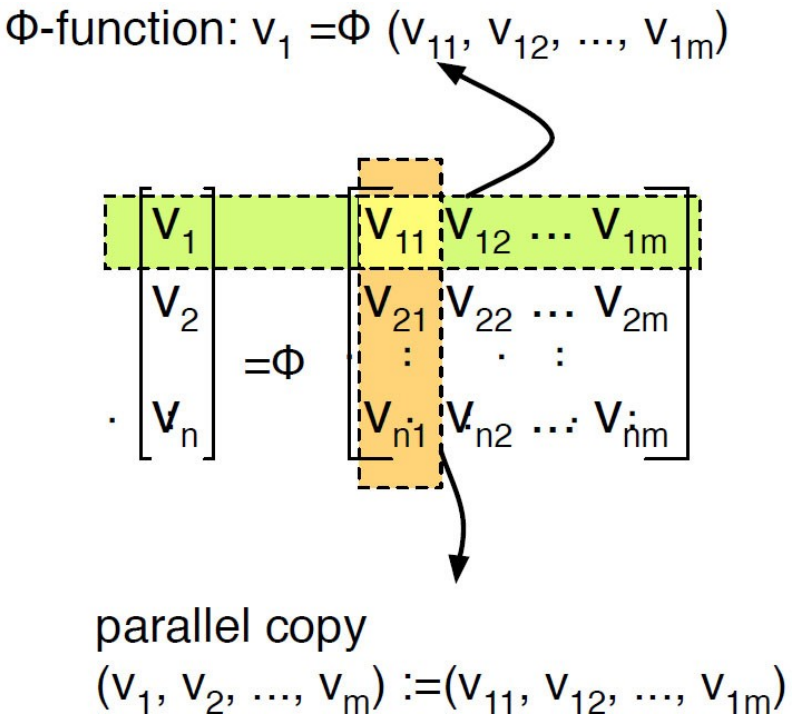
Elimination of Φ-functions

- Put parallel copies in predecessor blocks
- Sequentialize the parallel copies



Elimination of Φ-functions

- Put parallel copies in predecessor blocks
- Sequentialize the parallel copies



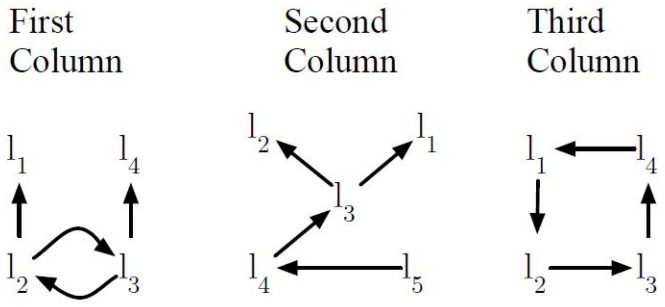
[Pereira&Palsberg 2010]

Parallel Copies

- $(x_0, x_1, \ldots, x_n) \leftarrow (x_{01}, x_{11}, \ldots, x_{n1})$
- Each parallel copy forms a "location transfer graph" [Pereira&Palsberg 2010]
  - edges in graph are the pairwise copies that need to be performed
- In LTG, in-degree is at most 1

Φ-matrix

$$\begin{bmatrix} l_1 \\ l_2 \\ l_3 \\ l_4 \end{bmatrix} = \Phi \begin{bmatrix} l_2 & l_3 & l_4 \\ l_3 & l_3 & l_1 \\ l_2 & l_4 & l_2 \\ l_3 & l_5 & l_3 \end{bmatrix}$$



Parallel Copies

- $(x_0, x_1, \ldots, x_n) \leftarrow (x_{01}, x_{11}, \ldots, x_{n1})$
- Each parallel copy forms a "location transfer graph" [Pereira&Palsberg 2010]
  - edges in graph are the pairwise copies that need to be performed
- In LTG, in-degree is at most 1
- If we spilled correctly (e.g., all Φ-related variables are spilled to same slot), then also:
  - out-degree of any node is at most 1
  - if node in graph is memory location, then

Spartan Transfer Graphs

- If we spilled correctly (e.g., all Φ-related variables are spilled to same slot), then also:
  - in-degree of any node is at most 1
  - out-degree of any node is at most 1
  - if node in graph is memory location, then
    - in-degree + out-degree is at most 1, or
    - edge on node is self-loop
- These graphs are "Spartan Transfer Graphs" [PP10]

Sequentializing Parallel Copies

- Each connected component forms
  - A Cycle
    (Then, all nodes are registers)
  - A Path
    (Then 1st may be memory store and/or last node may be memory load)
- Can implement as sequential code:
  - cycles use register swap
  - Paths use moves (mov, ld, st as appropriate)

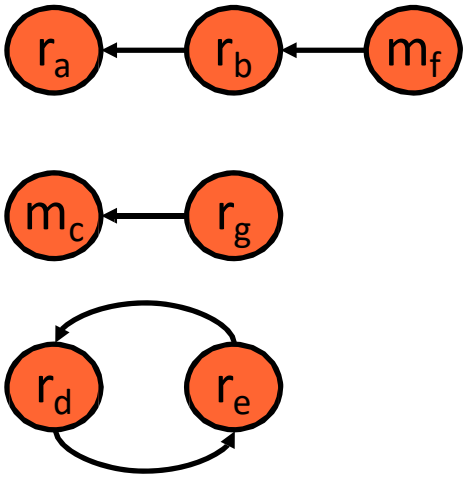
Parallel Copies

- $(x_0, x_1, \ldots, x_n) \leftarrow (x_{01}, x_{11}, \ldots, x_{n1})$
- Each parallel copy forms a "location transfer graph" [Pereira&Palsberg 2010]
  - edges in graph are the pairwise copies that need to be performed
- If we spilled correctly (e.g., all Φ-related variables are spilled to same slot), then LTG is either cycle or path
  - If cycle, only registers involved
  - If path and memory involved, then
    - 1st copy may be store and last copy may be load

Example LTG to code

$$\begin{bmatrix} r_a \\ r_b \\ m_c \\ r_d \\ r_e \end{bmatrix} = \Phi \begin{bmatrix} \cdots & r_b & \cdots \\ \cdots & m_f & \cdots \\ \cdots & r_g & \cdots \\ \cdots & r_e & \cdots \\ \cdots & r_d & \cdots \end{bmatrix}$$

Creates LTG with 3 connected components



mov $r_a \leftarrow r_b$
ld $r_b \leftarrow m_f$

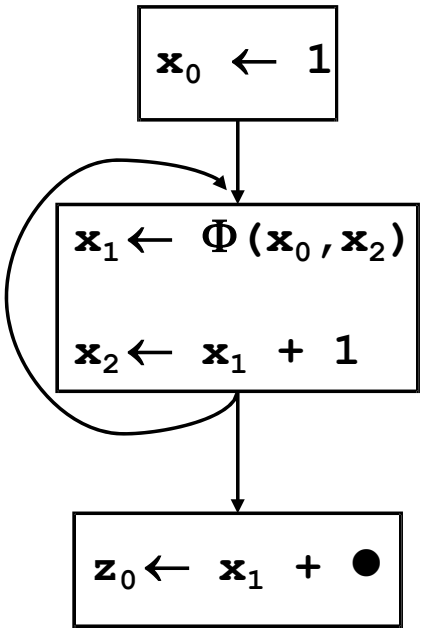
st $m_c \leftarrow r_g$

xchg $r_d \leftrightarrow r_e$

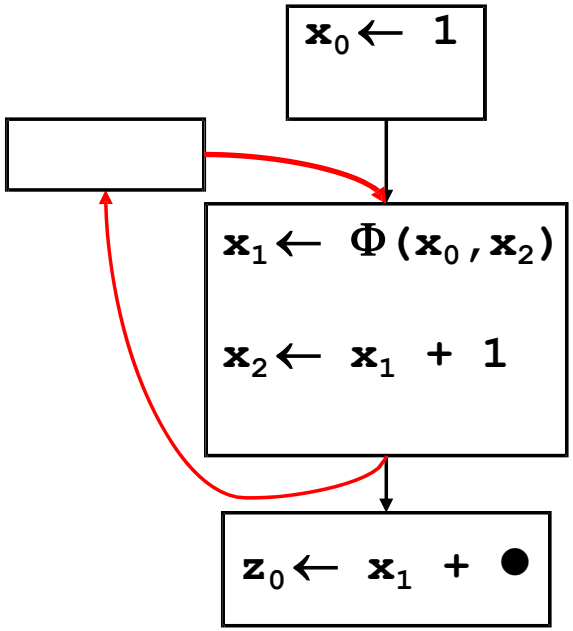
Putting it all together

- Critical Edge Splitting
- Convert back to Conventional-SSA (CSSA)
- Register Allocation
  - Build interference graph
  - pre-spilling
  - coloring
  - coalescing
- Deconstruct SSA
  - put parallel-copies in predecessors
  - Eliminate parallel copies

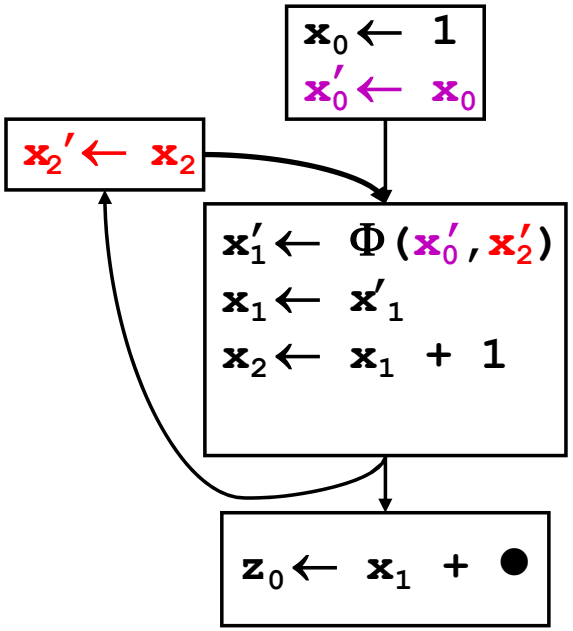
Example 1



Copy Folded

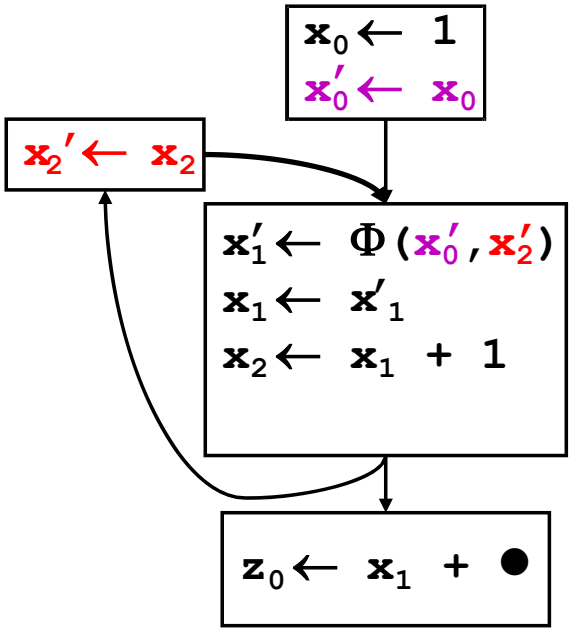


split critical edge

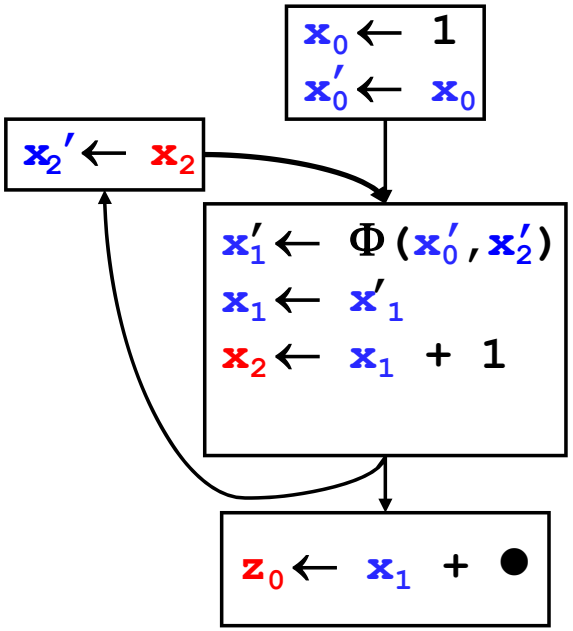


Convert to CSSA

Example 1



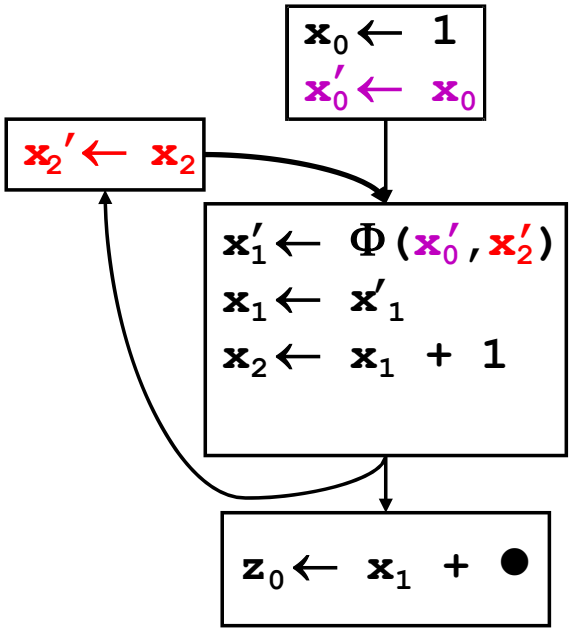
Convert to CSSA



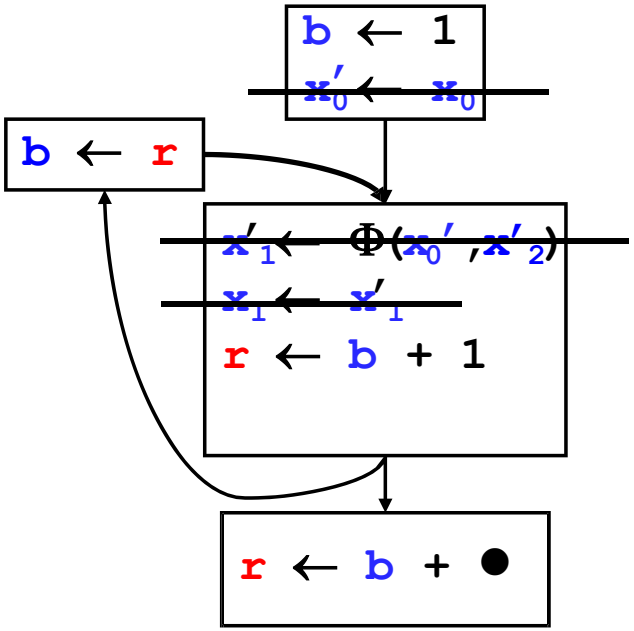
register allocation

Done, since $x \leftarrow \Phi(x,x)$ can simply be eliminated.

Well, we can clean up

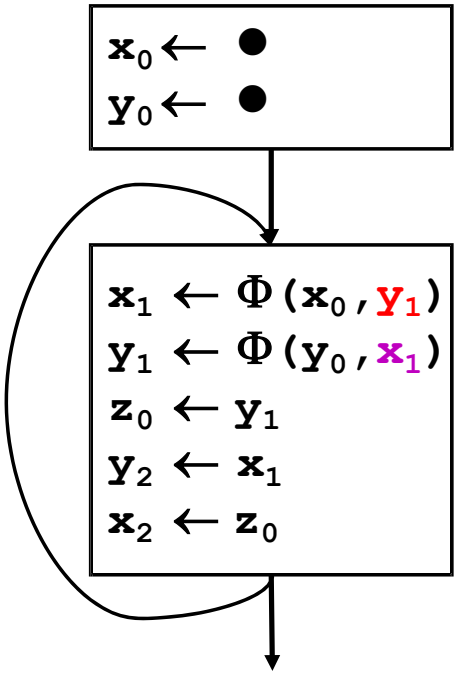


Convert to CSSA



register allocation

Example 2


$x_0 \leftarrow \bullet$
$y_0 \leftarrow \bullet$
$x_1 \leftarrow \Phi(x_0, y_1)$
$y_1 \leftarrow \Phi(y_0, x_1)$
$z_0 \leftarrow y_1$
$y_2 \leftarrow x_1$
$x_2 \leftarrow z_0$

Some Fine Tuning

- We added LOTS and LOTS of copies.
- Can reduce added copies when
  - creating CSSA
  - Introducing parallel copies
- Can rely on coalescing, but also …

Reducing Copies Going to CSSA

- Only need to introduce copies if there is interference!
- As building interference graph mark nodes which are Φ-related. If edge between them, introduce copies and update interference graph.
- Can do even better if also do liveness checking, see [Sreedhar et al, 1999]

Reducing Stores for Spilling

- Every path from LTG that ends in a memory slot will produce a store.
  E.g., $a \rightarrow r_1 \rightarrow r_2 \ldots r_x \rightarrow m$ will create
  `st` $\mathbf{r_x \rightarrow m}$ at the end.
- But, only needs to be done once, e.g., at point of definition.
- So, eliminate store and change register allocator to insert store at definition point
- Similar elimination of loads possible. See [Pereira&Palsberg 2010]

Coalescing

- Coalescing becomes even more important.
- Perform before SSA deconstruction (focus on Φ-related variables)
- (See [Boissinot et.al. 2009])