Lecture 11

Static Single Assignment (SSA)

I. Review: Intro to SSA

II. When/Where to Insert Φ

III. Example

IV. Constant Propagation with SSA

ALSU 6.2.4

I. Review: Static Single Assignment (SSA)

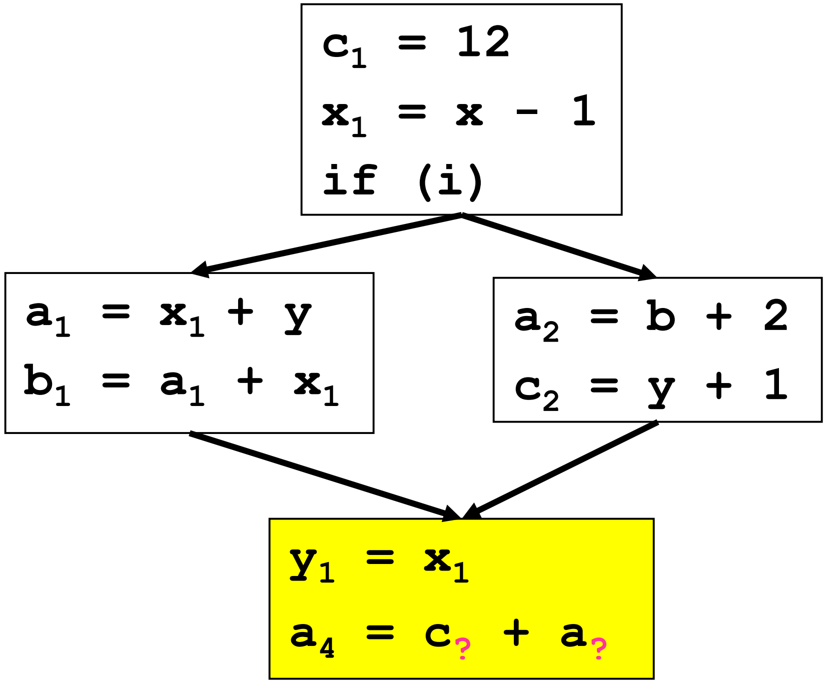
- Static single assignment is an IR where every variable is assigned a value at most once in the program text
- Easy for a basic block (reminiscent of Value Numbering):
  - Visit each instruction in program order:
    - LHS: assign to a *fresh version* of the variable
    - RHS: use the *most recent version* of each variable

```
a = x + y
b = a + x
a = b + 2
c = y + 1
a = c + a
```

$$
\begin{aligned}
a_1 &= x + y \\
b_1 &= a_1 + x \\
a_2 &= b_1 + 2 \\
c_1 &= y + 1 \\
a_3 &= c_1 + a_2
\end{aligned}
$$

Review: What about Joins in the CFG?

```
c = 12
x = x - 1
if (i) {
   a = x + y
   b = a + x
} else {
   a = b + 2
   c = y + 1
}
y = x
a = c + a
```

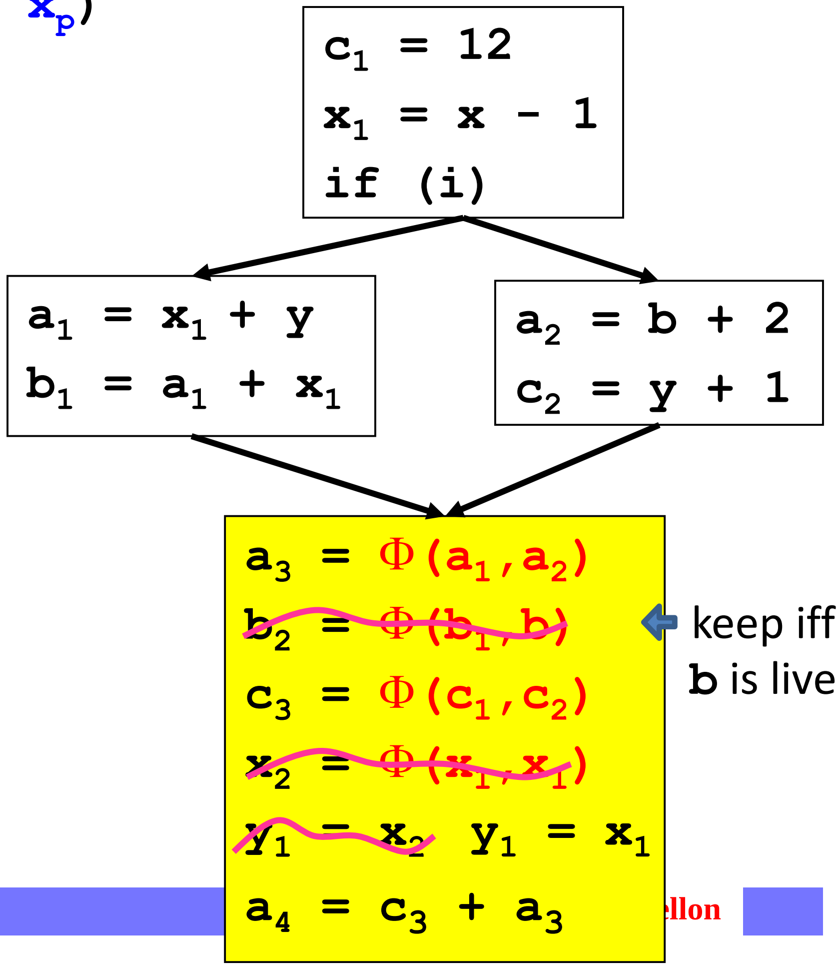


→ Use a notational convention (fiction): a Φ function

Review: The Φ function

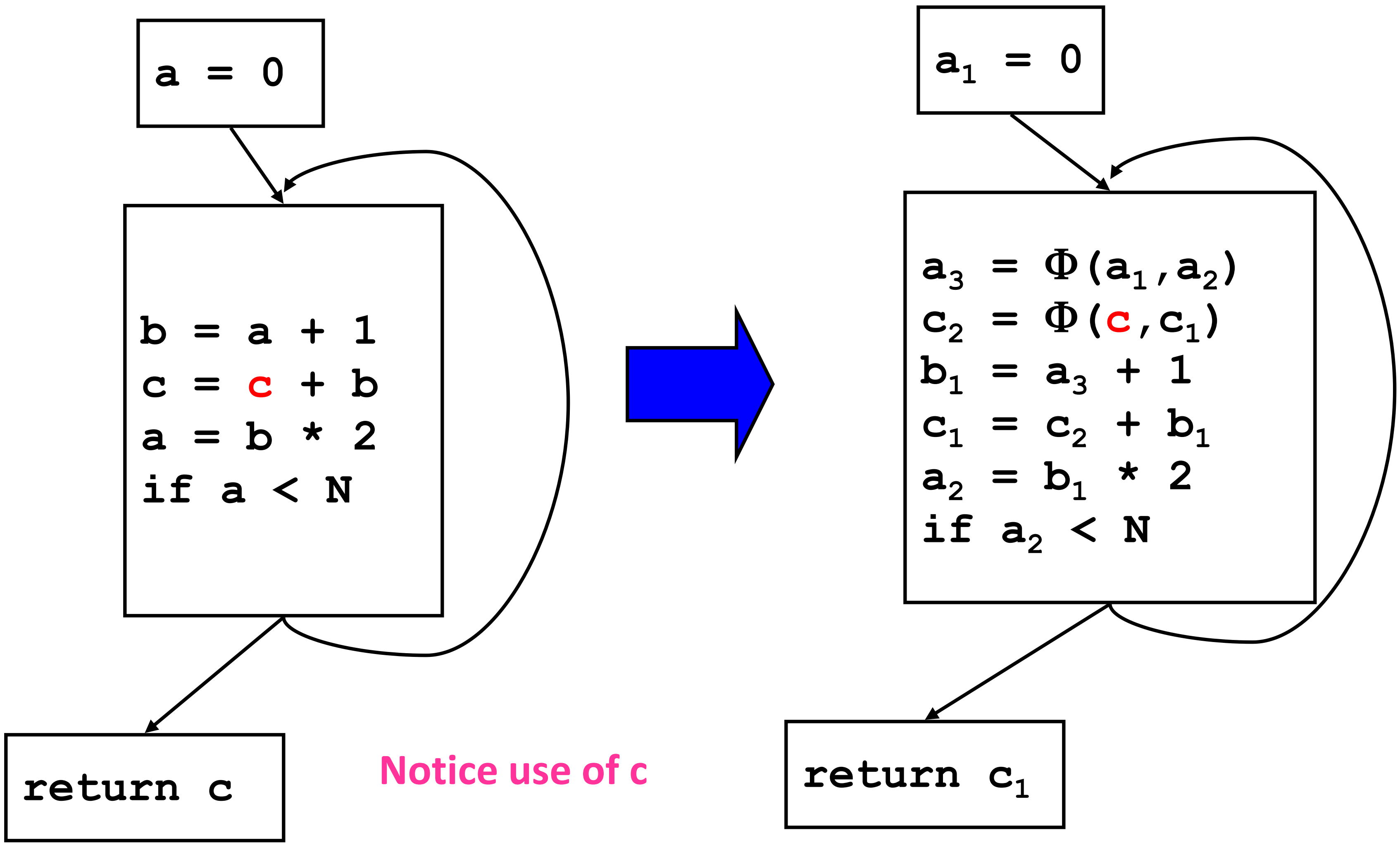
- Φ merges multiple definitions along multiple control paths into a single definition.
- At a basic block with *p* predecessors, there are *p* arguments to the Φ function.

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

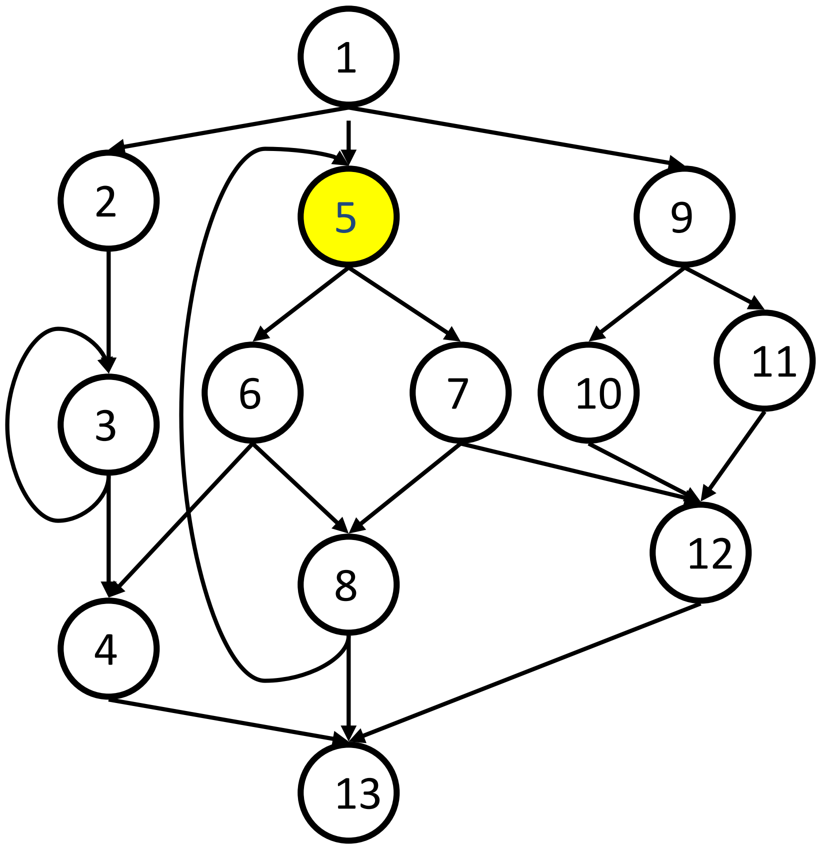


- **Minimal SSA**: At each join point, insert Φ functions for all live variables with multiple outstanding defs

Another Example



II. When/Where to Insert Φ?

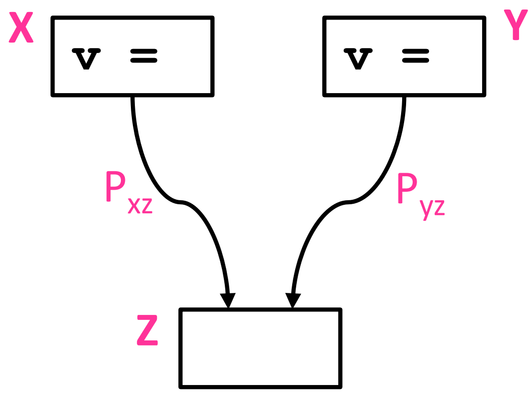


If there is a def of **a** in block 5, which nodes need a Φ()?

Control Flow Graph (CFG)

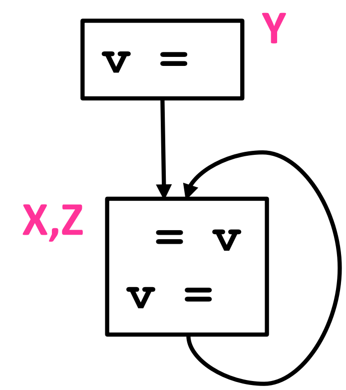
When/Where to insert Φ?

- We insert a Φ function for variable **v** in block Z iff:
  - **v** was defined more than once before
    - (i.e., **v** defined in X and Y AND $X \neq Y$)
  - There exists nonempty paths P_{xz} from X to Z and P_{yz} from Y to Z s.t. Z is the first node common to the two paths
    - Nonempty = at least one edge
    - Note: one of X or Y can be Z



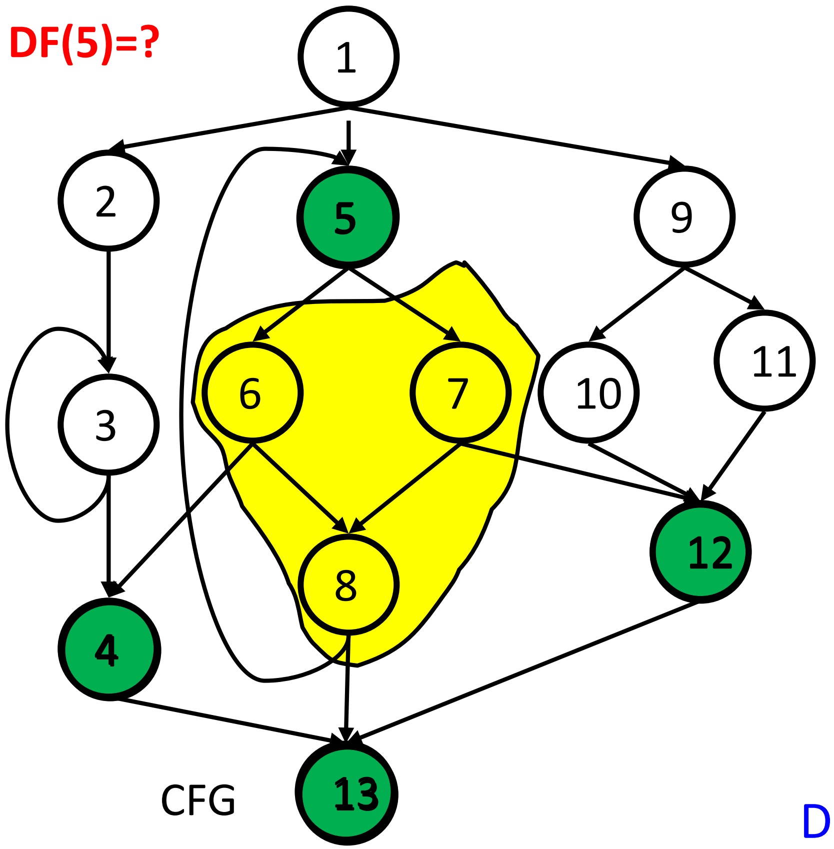
Path Convergence

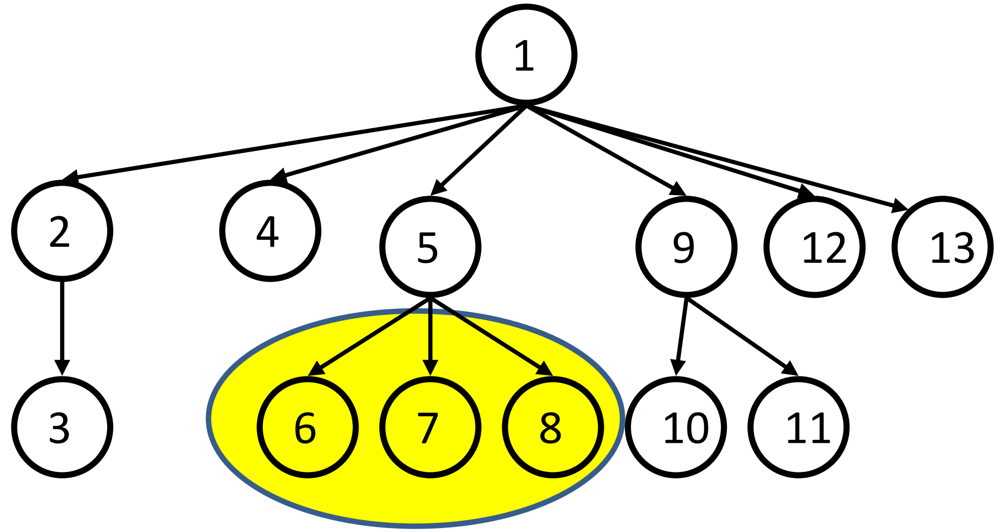
- **Entry block** contains an implicit def of all vars
- Note: **v** = Φ(…) is a def of **v**



Dominance Frontier

DF(5)=?





Dominance Tree

x sdom w iff x is a proper ancestor of w

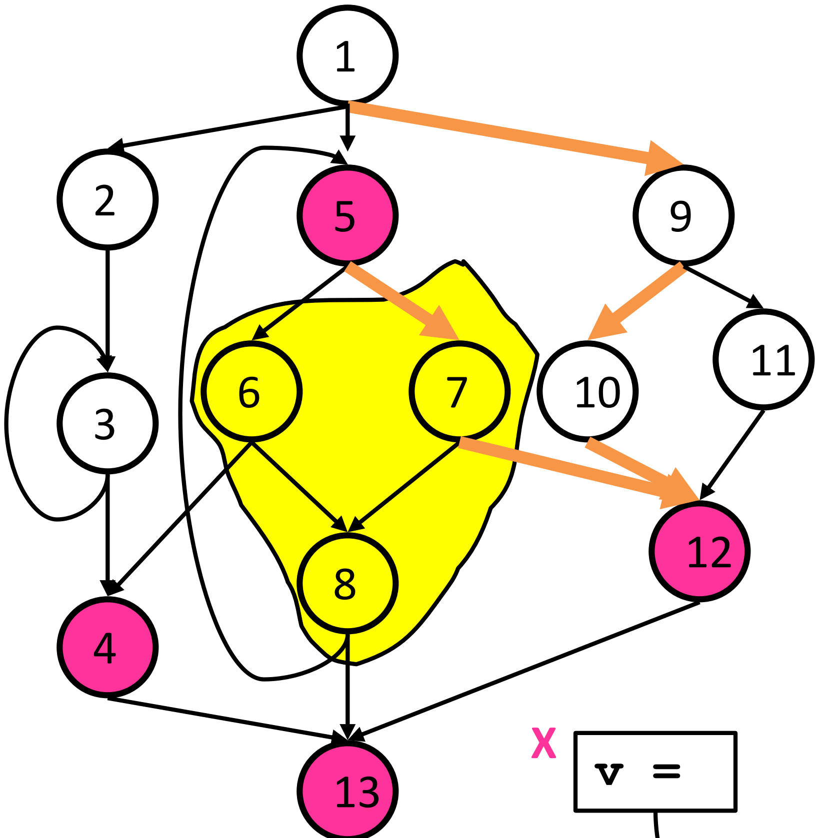
The Dominance Frontier of a node x

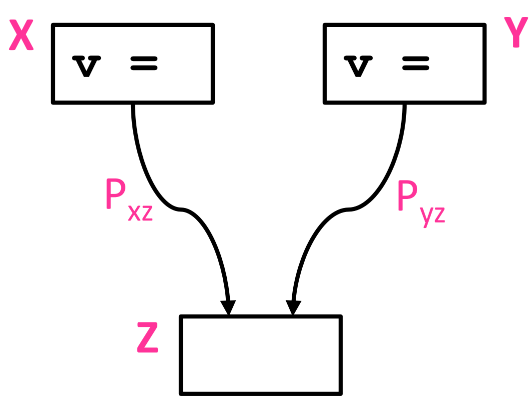
DF(x) = { w | x dom pred(w) AND !(x sdom w)}

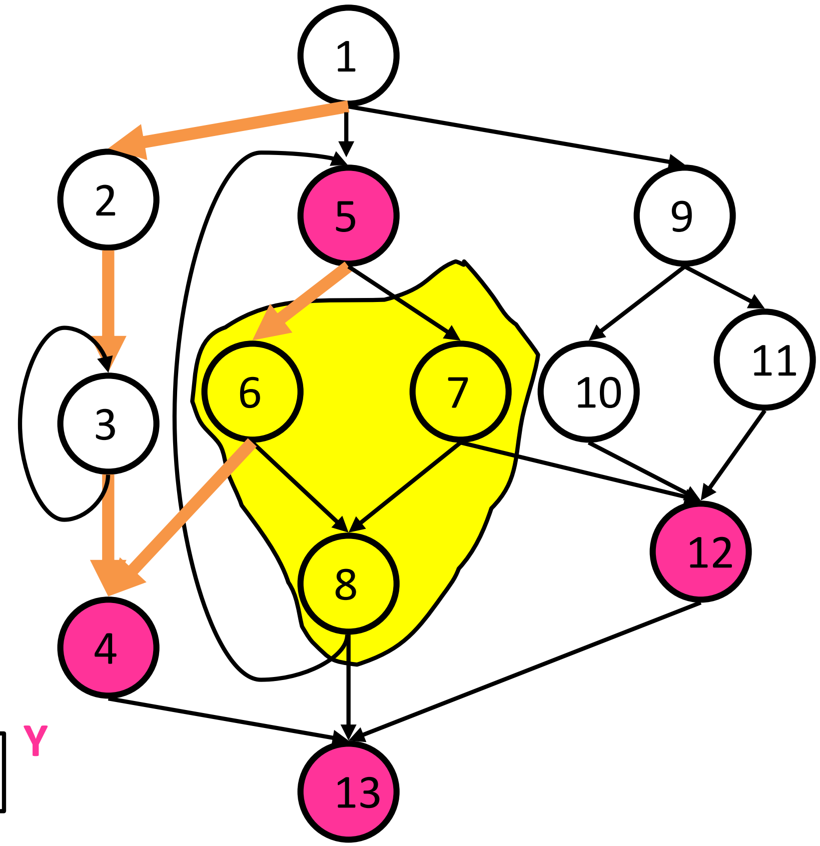
x strictly dominates w (x sdom w) iff impossible to reach w without passing through x first

x dominates w (x dom w) iff x sdom w OR x = w

Dominance Frontier (DF) and Path Convergence



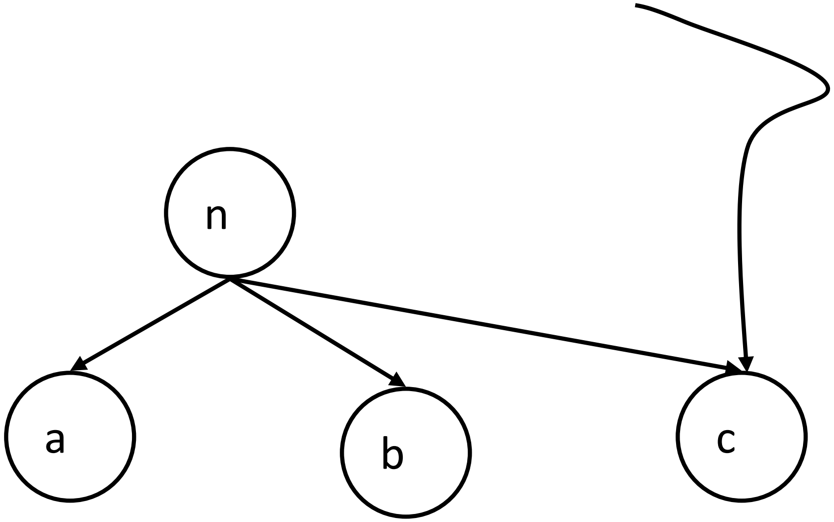




If there is a def of **a** in block **5**, nodes in DF(**5**) need a $\Phi()$ for **a**

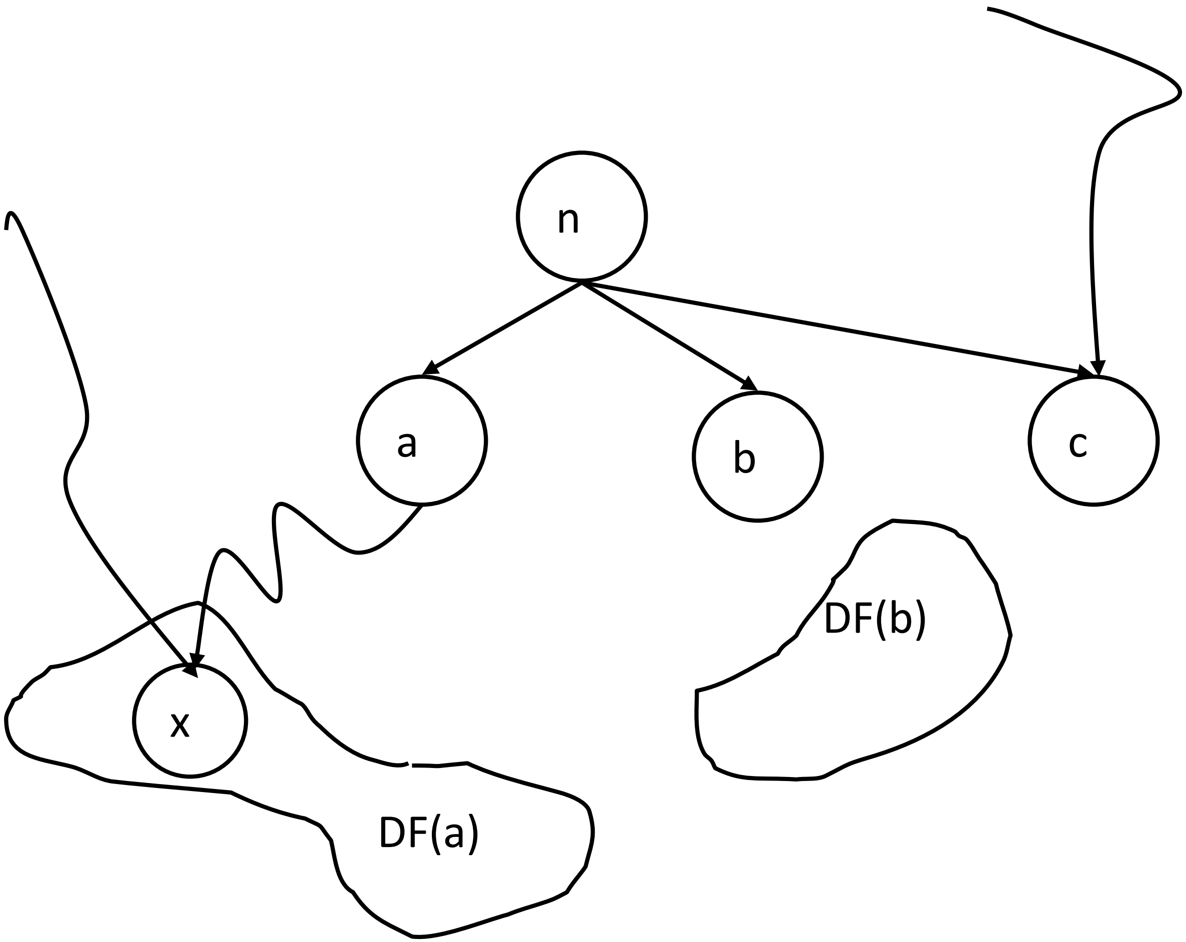
Computing DF(n)

DF(x) = { w | x dom pred(w) AND !(x sdom w)}



n dom n
n dom a
n dom b
!(n dom c)

Computing DF(n)



n dom n
n dom a
n dom b
!(n dom c)

Computing the Dominance Frontier

DF(n) = { w | n dom pred(w) AND !(n sdom w)}

```
compute-DF(n)
        S = {}
        foreach node c in succ[n]
                if !(n sdom c)
                        S = S ∪ { c }        e.g., node c on previous slide
        foreach child a of n in D-tree
                compute-DF(a)
                foreach x in DF[a]
                        if !(n dom x)
                                S = S ∪ { x }     e.g., node x on previous slide
        DF[n] = S
```

Using Dominance Frontier to Compute SSA: Overview

1. Place all $\Phi()$
2. Rename all variables

Using Dominance Frontier to Place $\Phi()$

- Gather all the defsites of every variable
- Then, for every variable
  - foreach defsite
    - foreach node in DominanceFrontier(defsite)
      - if we haven't put $\Phi()$ in node, then put one in
      - if this node didn't define the variable before, then add this node to the defsites (because Φ counts as def)

- This essentially computes the Iterated Dominance Frontier on the fly, inserting the minimal number of $\Phi()$ neccesary

Using Dominance Frontier to Place Φ(): Algorithm

```
foreach node n {
   foreach variable v defined in n {
     orig[n] ∪= {v}            /* variables defined in basic block n */
     defsites[v] ∪= {n}        /* basic blocks that define variable v */
   }
}
foreach variable v {
   W = defsites[v]                              /* work list of basic blocks */
   while W not empty {
     n = remove node from W
     foreach y in DF[n]
       if y ∉ PHI[v] {
         insert "v ← Φ(v,v,…)" at top of y
         PHI[v] = PHI[v] ∪ {y}          /* BBs containing a Φ for v */
         if v ∉ orig[y]: W = W ∪ {y}    /* add BB to work list */
       }
   }
}
```

Renaming Variables

- Algorithm:
  - Walk the D-tree, renaming variables as you go
  - Replace uses with more recent renamed def

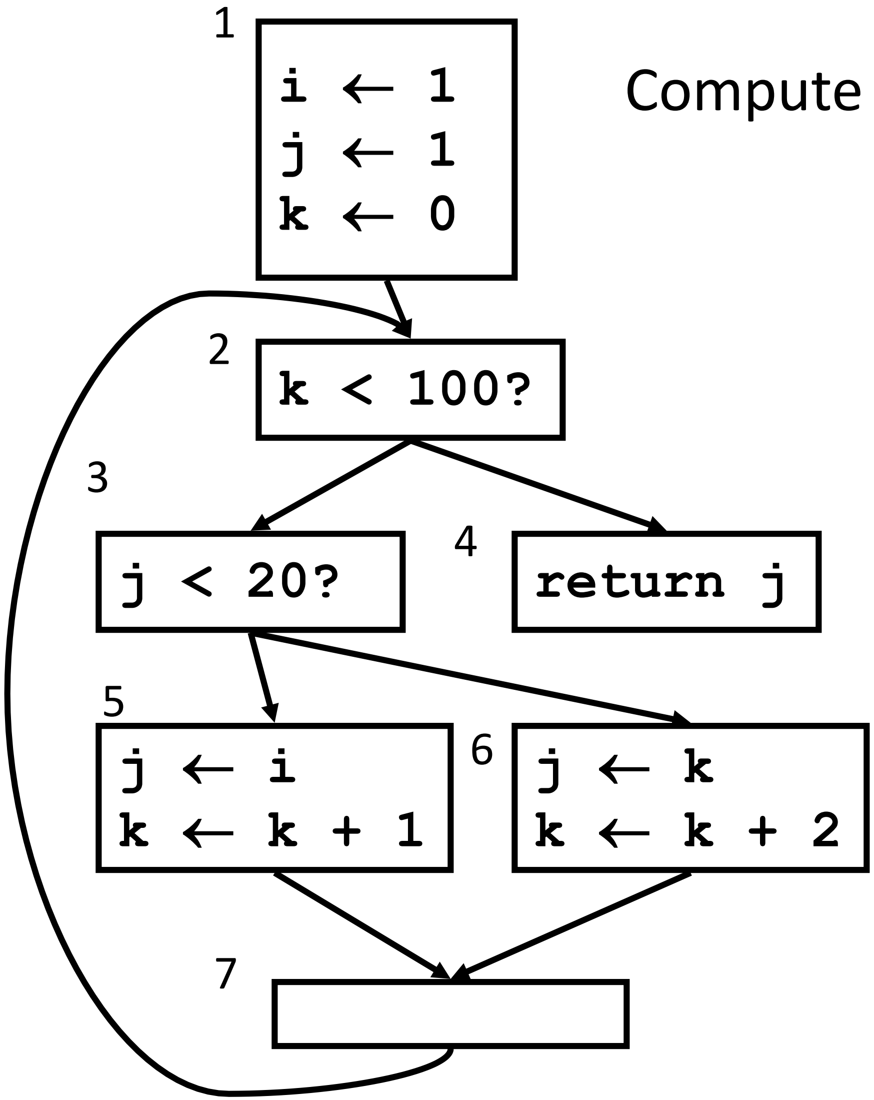
- For straight-line code this is easy
- What if there are branches and joins?
  - use the closest def such that the def is above the use in the D-tree

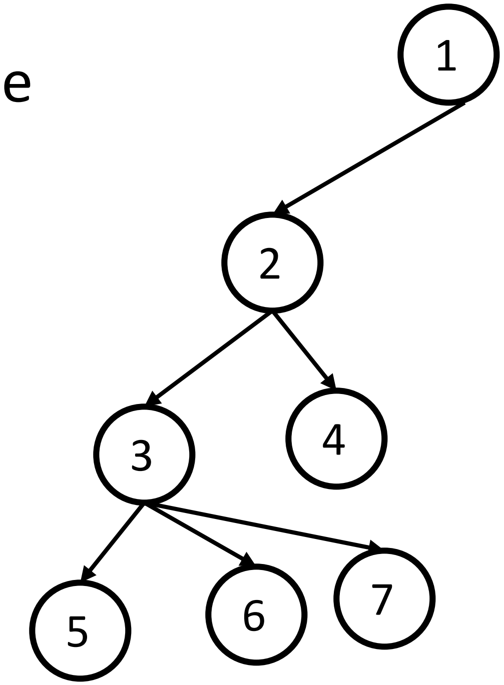
- Easy implementation:
  - Call rename(entry)

```
rename(B):
for each assignment in B:
    replace (non-Φ) use of v with top of stack(v)
    replace def of v with v_new, push v_new onto stack(v)
for each successor S of B in CFG:
    replace k'th arg. of Φ(v,…v) with top of stack(v),
            where B is k'th predecessor of S
call rename(C) on all children C of B in D-tree
pop all defs in B from stacks
```

III. Example

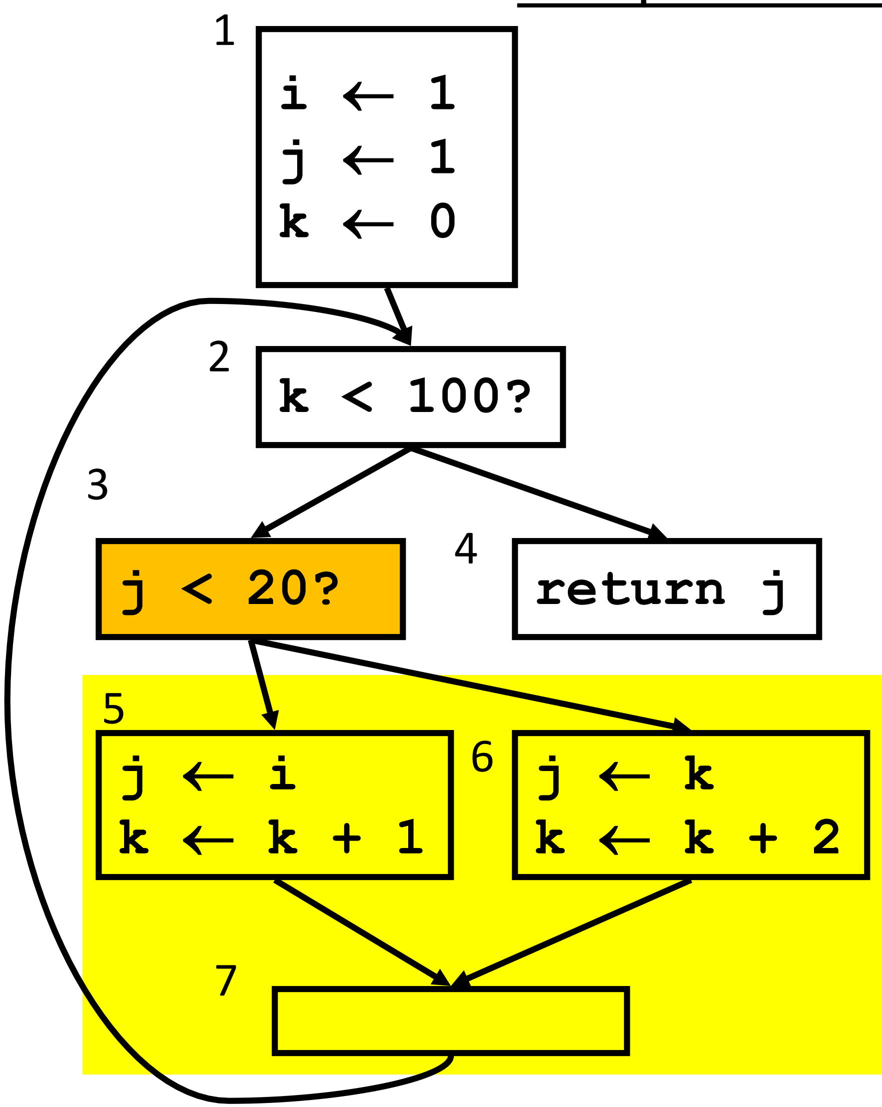
Compute Dominance Tree





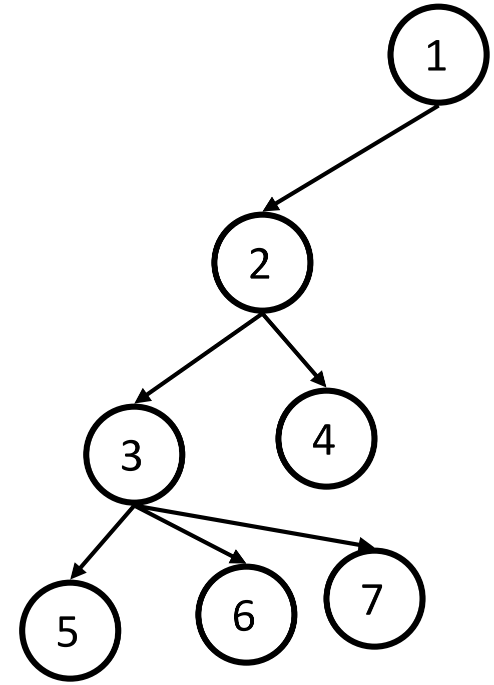
D-tree

Compute Dominance Frontiers



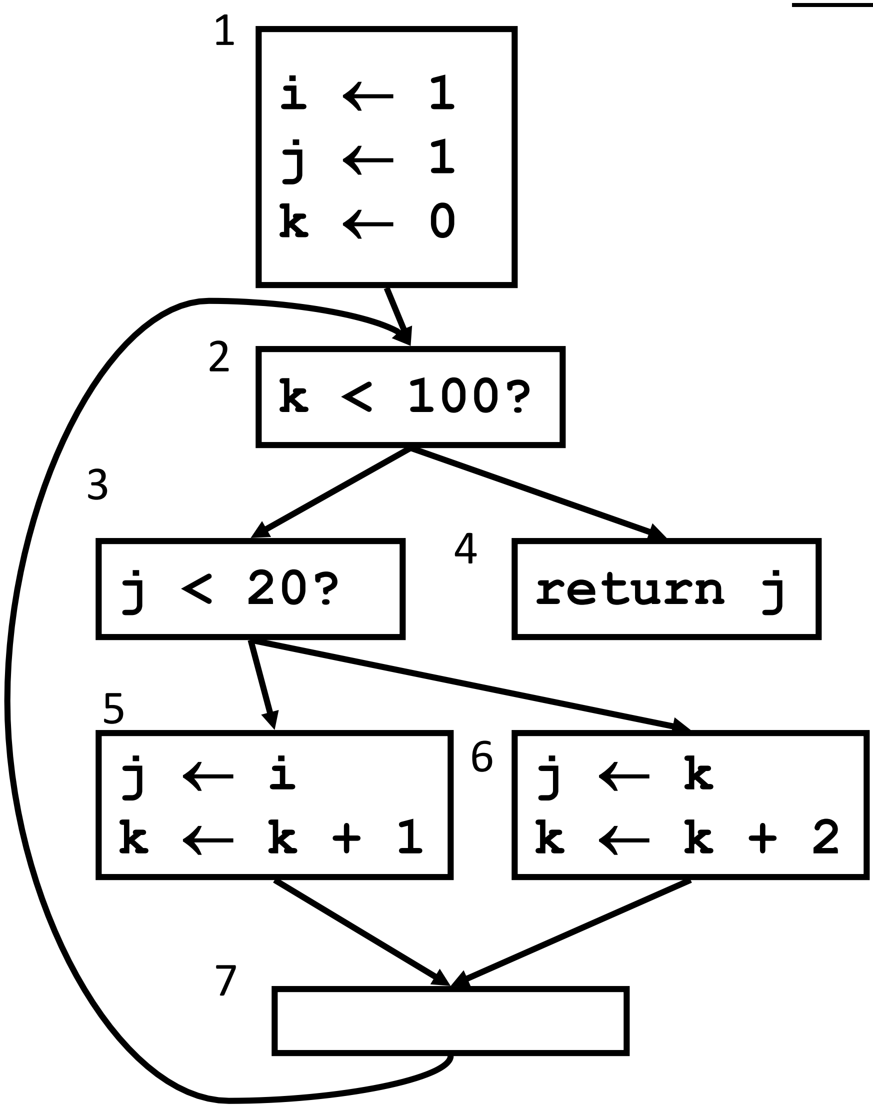
DFs

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



DF(x) = { w | x dom pred(w)
AND !(x sdom w)}

Insert Φ()



orig[n] = variables defined in n

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

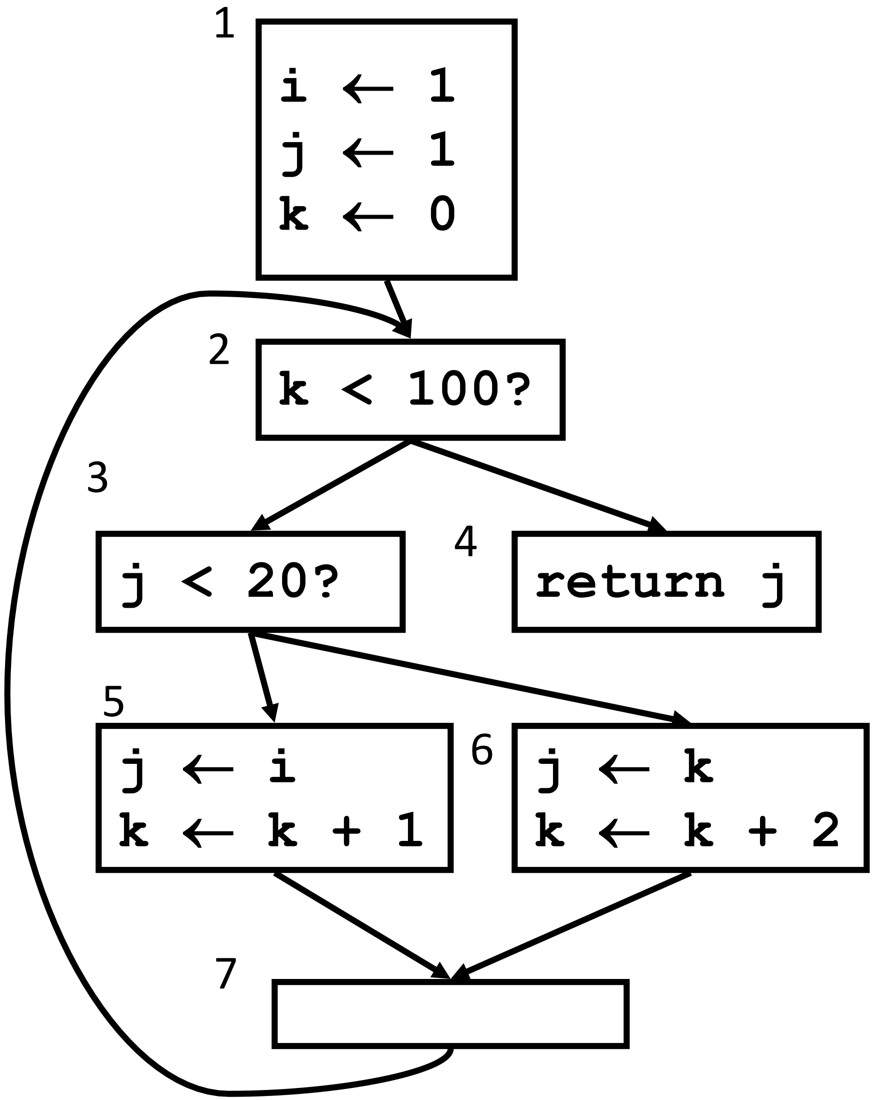
defsites[v]

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

```
foreach variable v {
  W = defsites[v]
  while W not empty {
    n = remove node from W
    …
```

var i: W={1}

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	{}

PHI[v]

j	{}
k	{}

defsites[v]

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

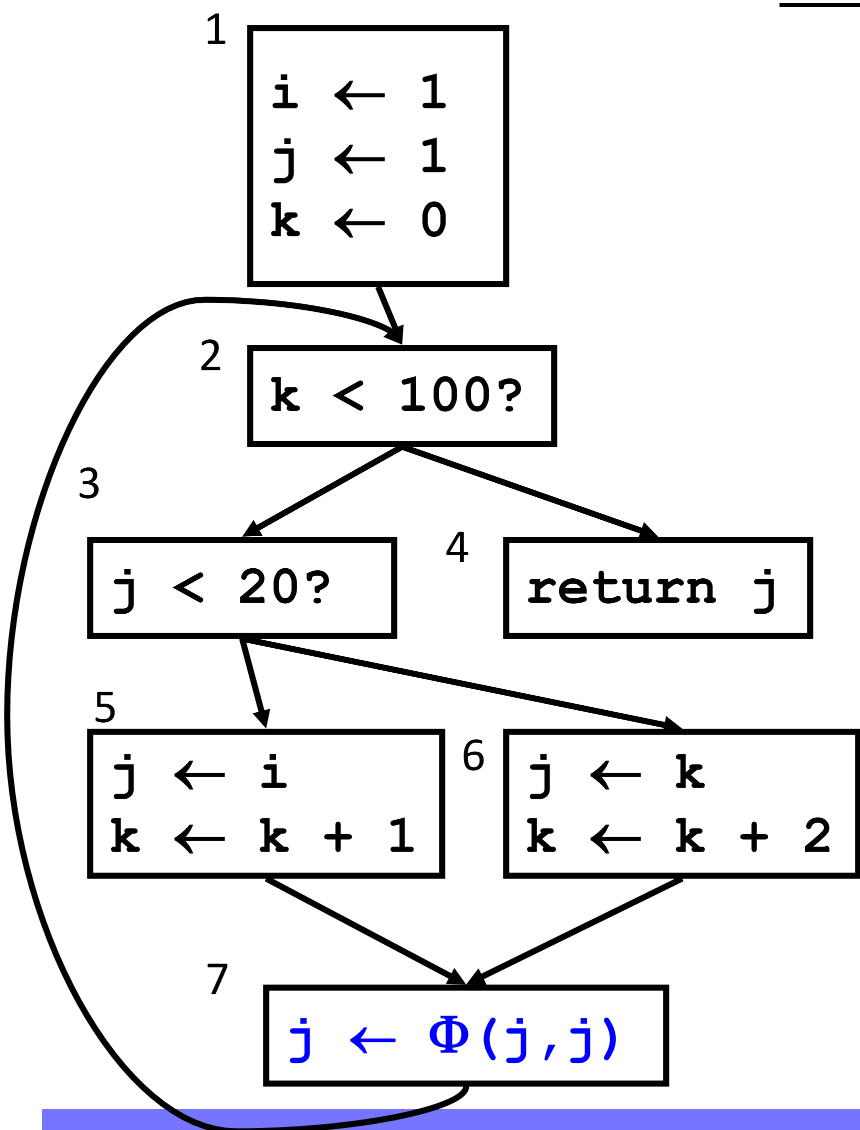
```
foreach y in DF[n]
  if y ∉ PHI[v] {
    insert "v ← Φ(v,v,…)" in y
    PHI[v] = PHI[v] ∪ {y}
    if v ∉ orig[y]: W = W ∪ {y}
  }
```

var i: W={1}   DF{1}

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

DF{1}   DF{5}

Insert Φ()



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

PHI[v]

j	{7}
k	{}

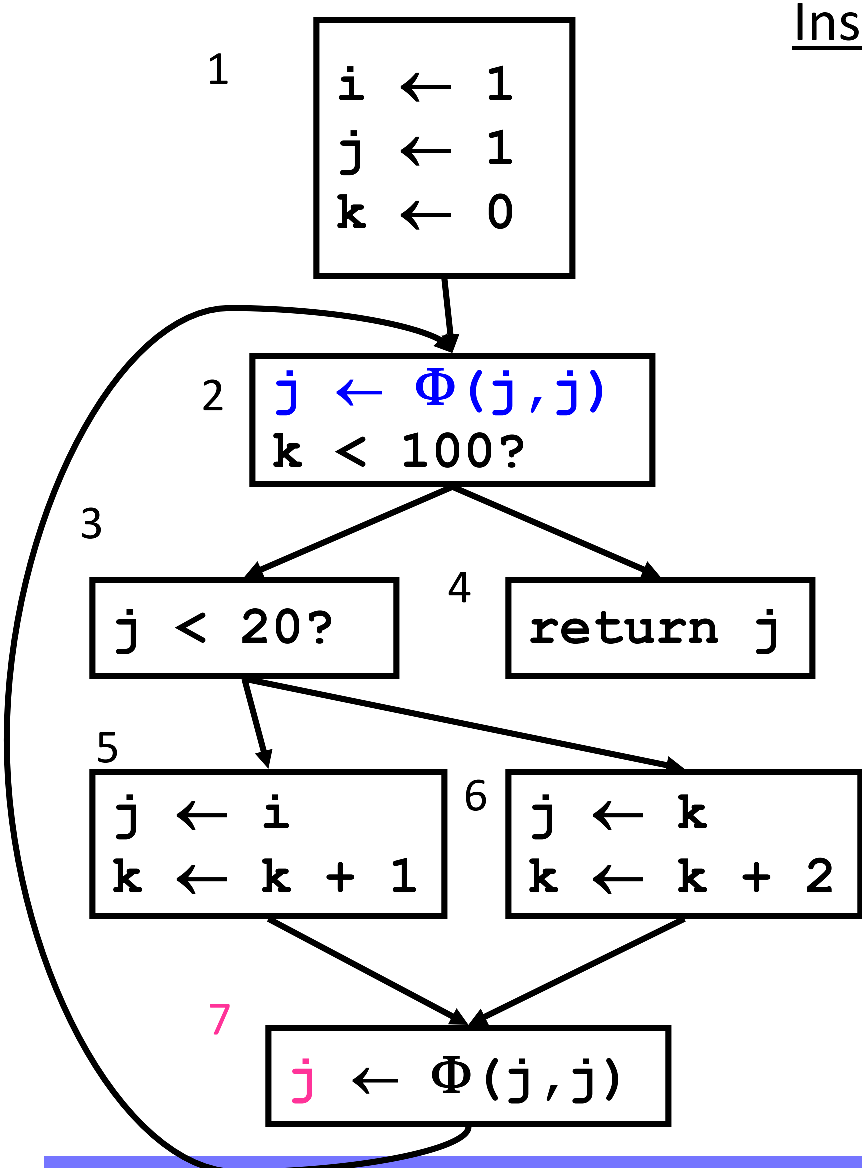
defsites[v]

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

```
foreach y in DF[n]
  if y ∉ PHI[v] {
    insert "v ← Φ(v,v,…)" in y
    PHI[v] = PHI[v] ∪ {y}
    if v ∉ orig[y]: W = W ∪ {y}
  }
```

var j: W={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	{}

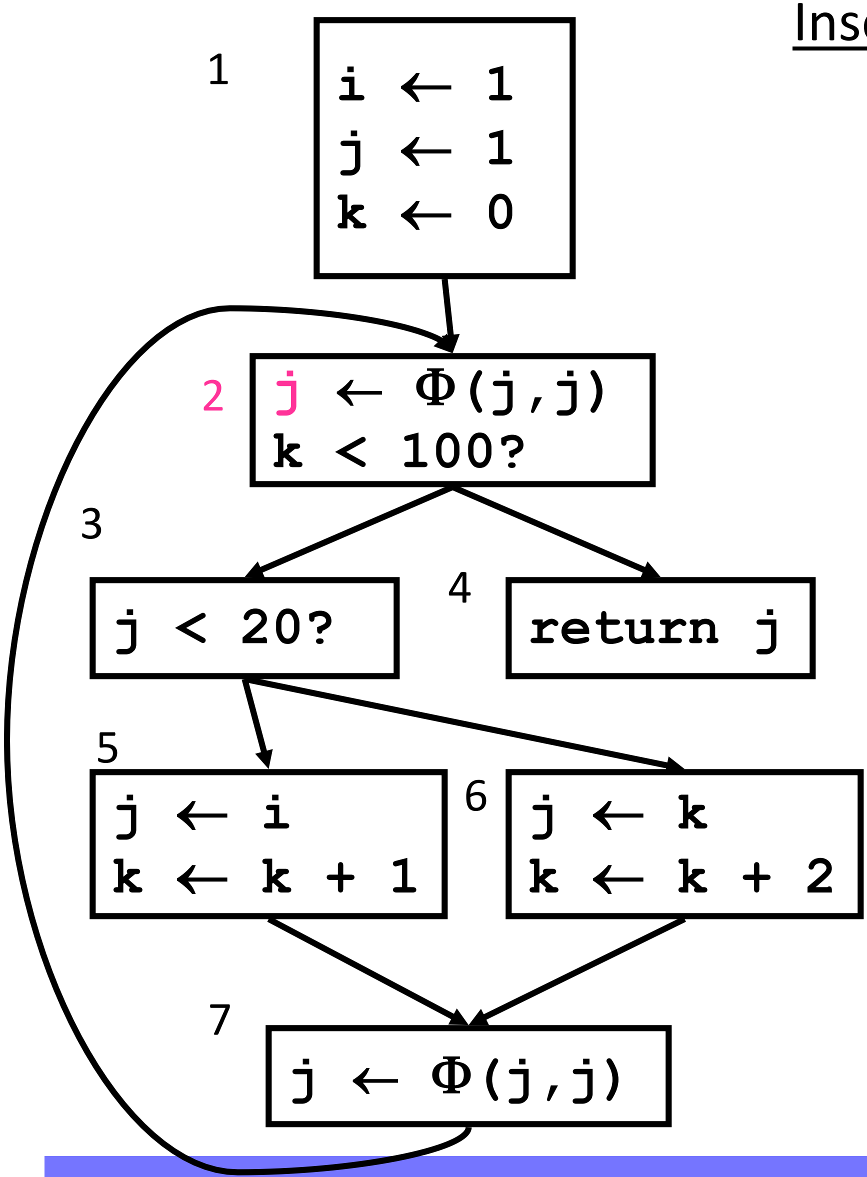
PHI[v]	
j	{2,7}
k	{}

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

```
foreach y in DF[n]
   if y ∉ PHI[v] {
     insert "v ← Φ(v,v,…)" in y
     PHI[v] = PHI[v] ∪ {y}
     if v ∉ orig[y]: W = W ∪ {y}
   }
```

var j: W={7,6}

~~DF{1}~~ ~~DF{[illegible]}~~ DF{7}



Insert Φ()

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

PHI[v]

j	{2,7}
k	{}

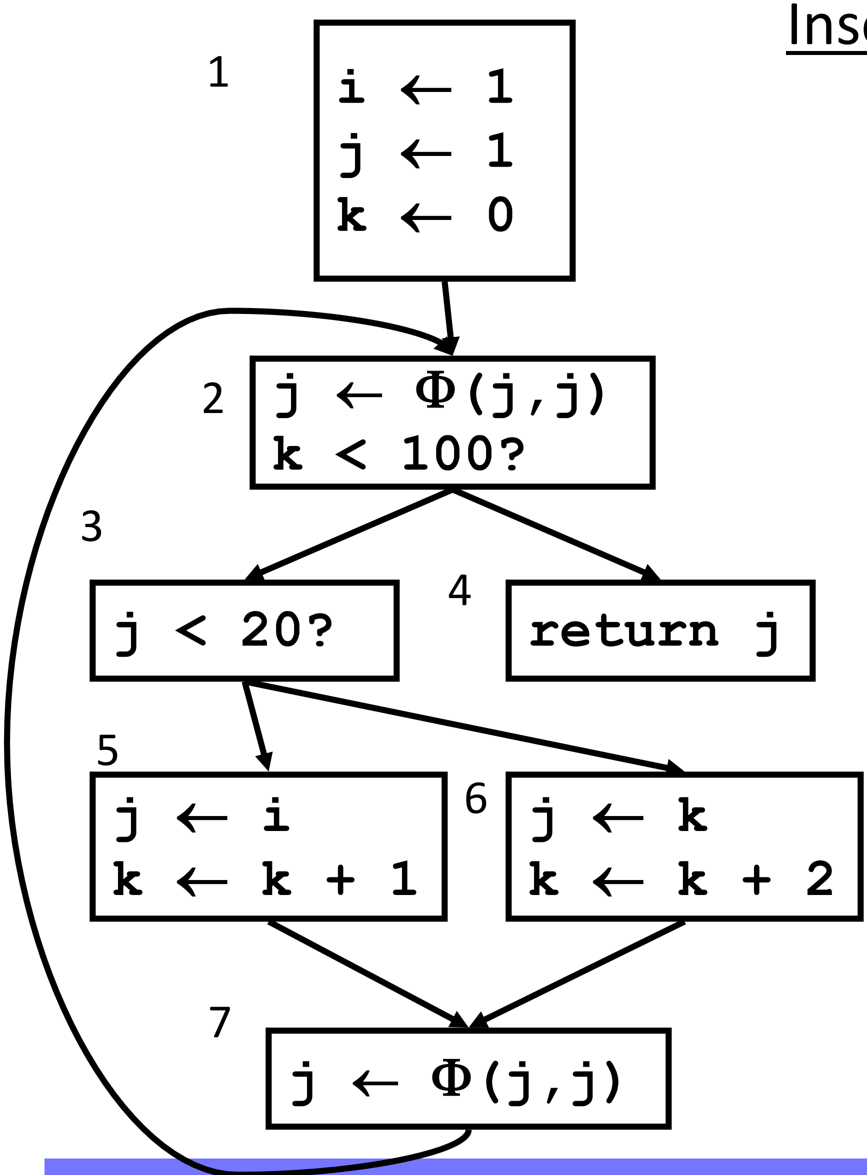
defsites[v]

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

```
foreach y in DF[n]
  if y ∉ PHI[v] {
    insert "v ← Φ(v,v,…)" in y
    PHI[v] = PHI[v] ∪ {y}
    if v ∉ orig[y]: W = W ∪ {y}
  }
```

var j: W={2,6}

DF{1} DF{5} DF{7} DF{2}



Insert Φ()

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

PHI[v]

j	{2,7}
k	{}

defsites[v]

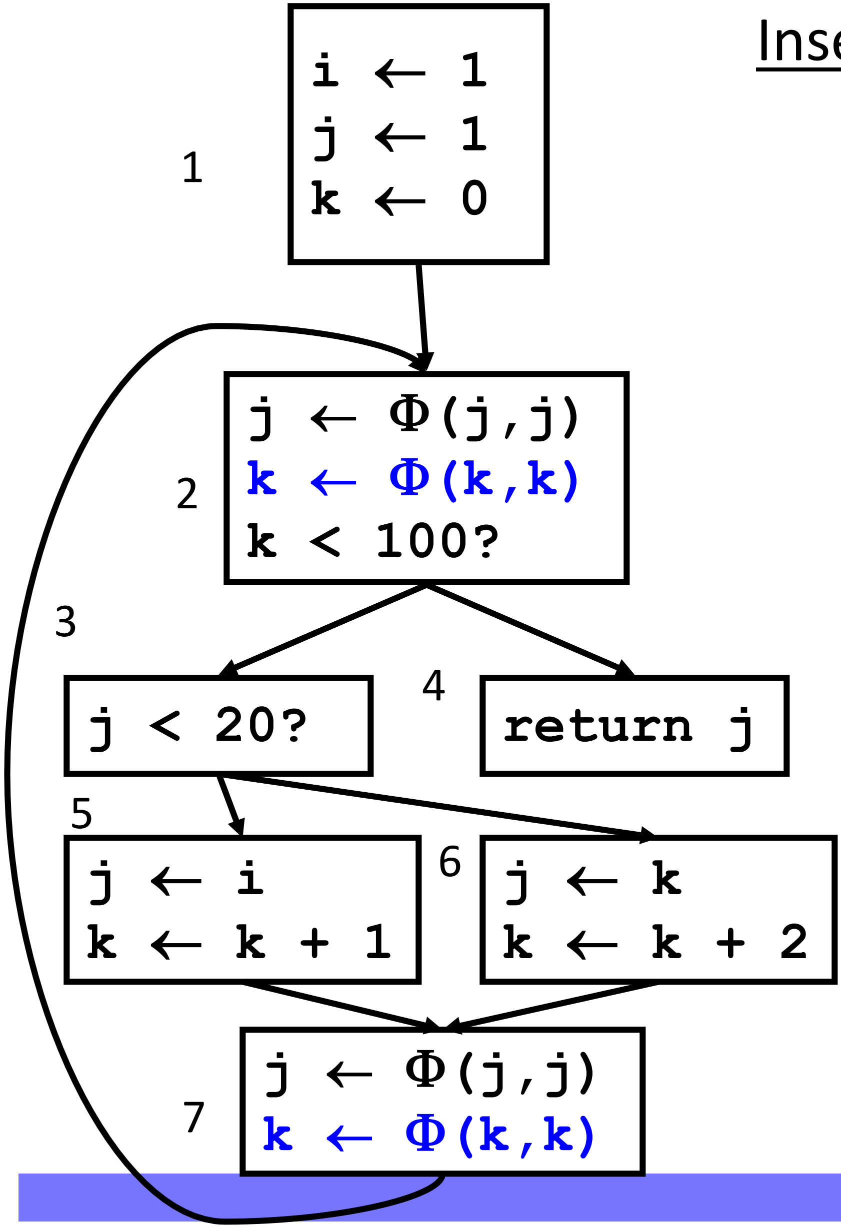
i	{1}
j	{1,5,6}
k	{1,5,6}

```
foreach y in DF[n]
  if y ∉ PHI[v] {
    insert "v ← Φ(v,v,…)" in y
    PHI[v] = PHI[v] ∪ {y}
    if v ∉ orig[y]: W = W ∪ {y}
  }
```

var j: W={6}

DF{6}

DF{1} DF{5} DF{7} DF{2}



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	{}

PHI[v]

j	{2,7}
k	{2,7}

defsites[v]

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

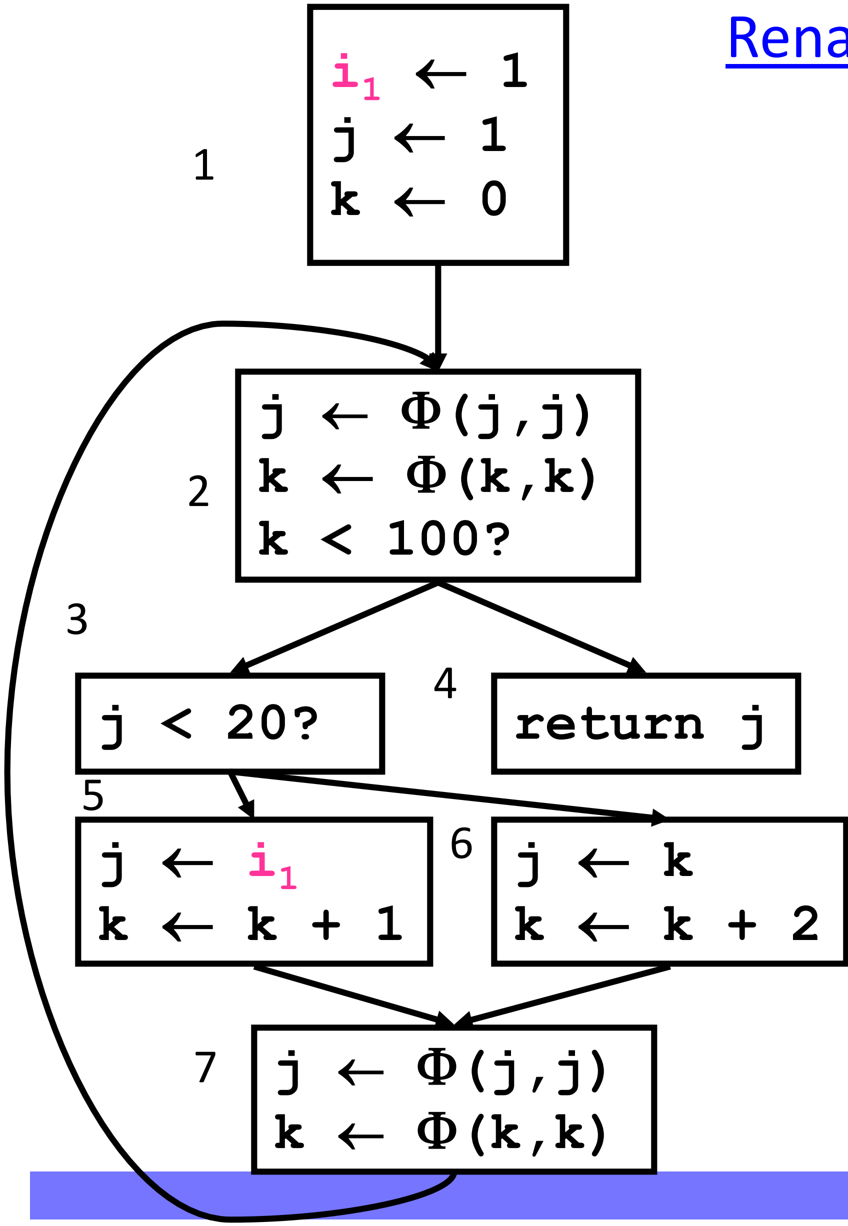
```
foreach y in DF[n]
  if y ∉ PHI[v] {
    insert "v ← Φ(v,v,…)" in y
    PHI[v] = PHI[v] ∪ {y}
    if v ∉ orig[y]: W = W ∪ {y}
  }
```

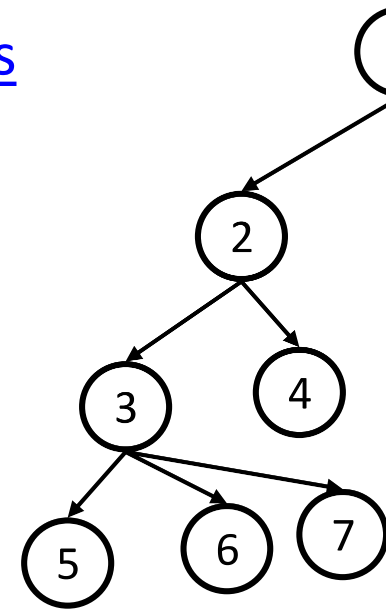
var k: W={1,5,6}

Same as var j: adds Φ to 7 and 2

Done inserting Φ()s…Time to rename vars

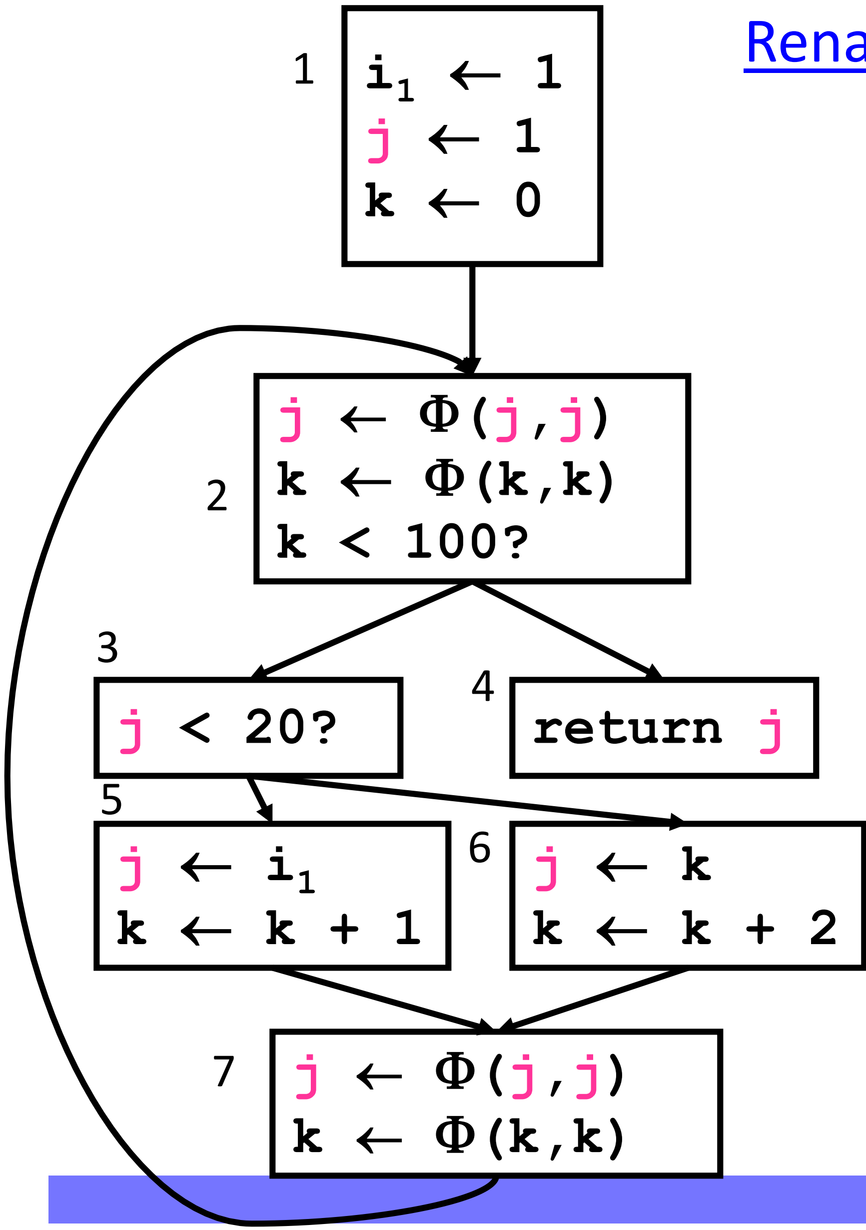
Rename Vars

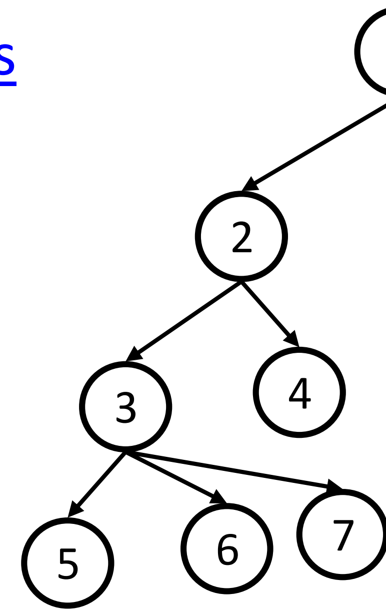




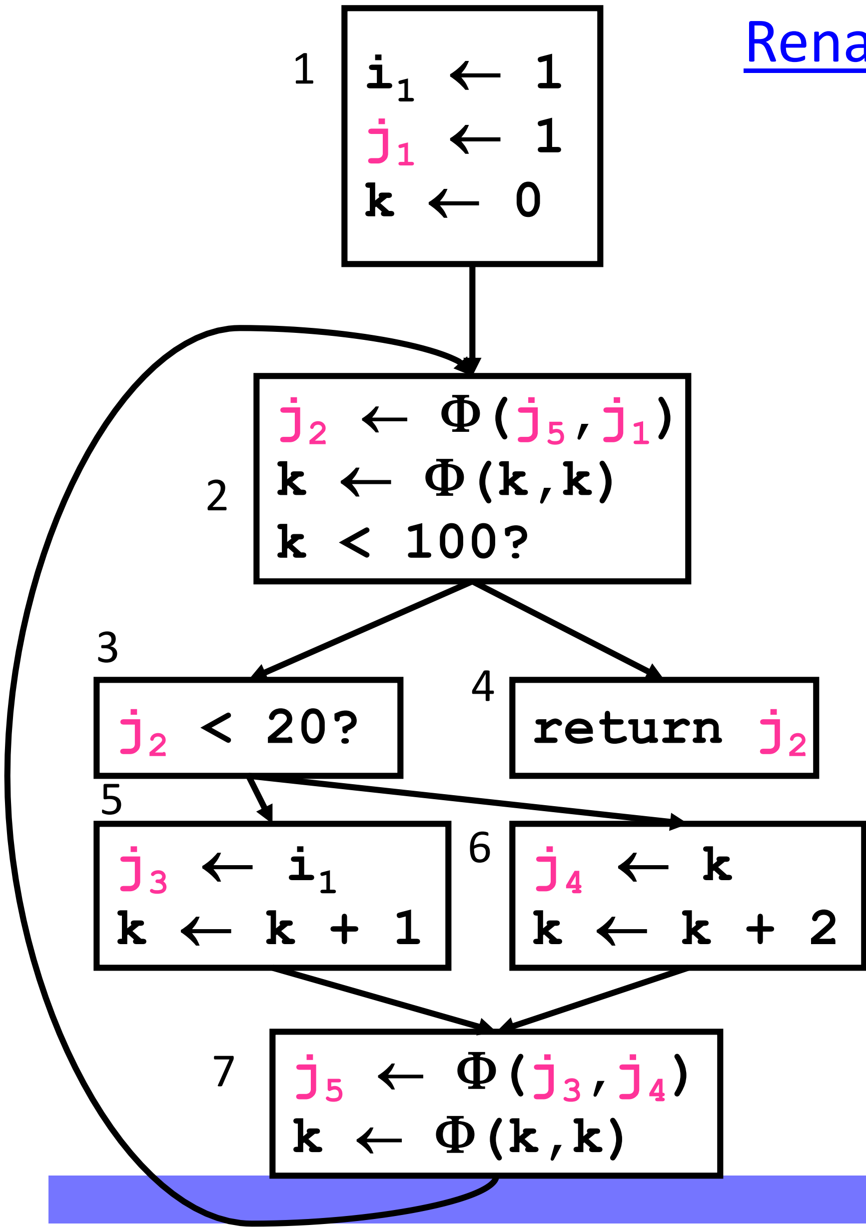
rename(B):
for each assignment in B:
    replace use of v with top of stack(v)
    replace def of v with v_{new},
        push v_{new} onto stack(v)
for each successor S of B in CFG:
    replace k'th arg. of **Φ(v,…v)** with top of
        stack(v), where B is k'th predecessor of S
call rename(C) on all children C of B in D-tree
pop all defs in B from stacks

Rename Vars

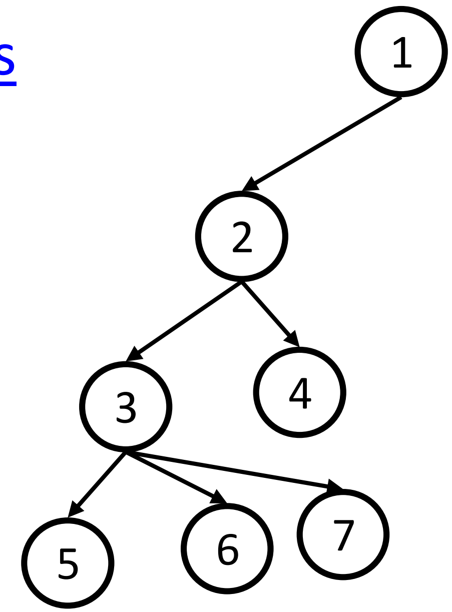





```
rename(B):
for each assignment in B:
    replace use of v with top of stack(v)
    replace def of v with v_new,
        push v_new onto stack(v)
for each successor S of B in CFG:
    replace k'th arg. of Φ(v,…v) with top of
        stack(v), where B is k'th predecessor of S
call rename(C) on all children C of B in D-tree
pop all defs in B from stacks
```

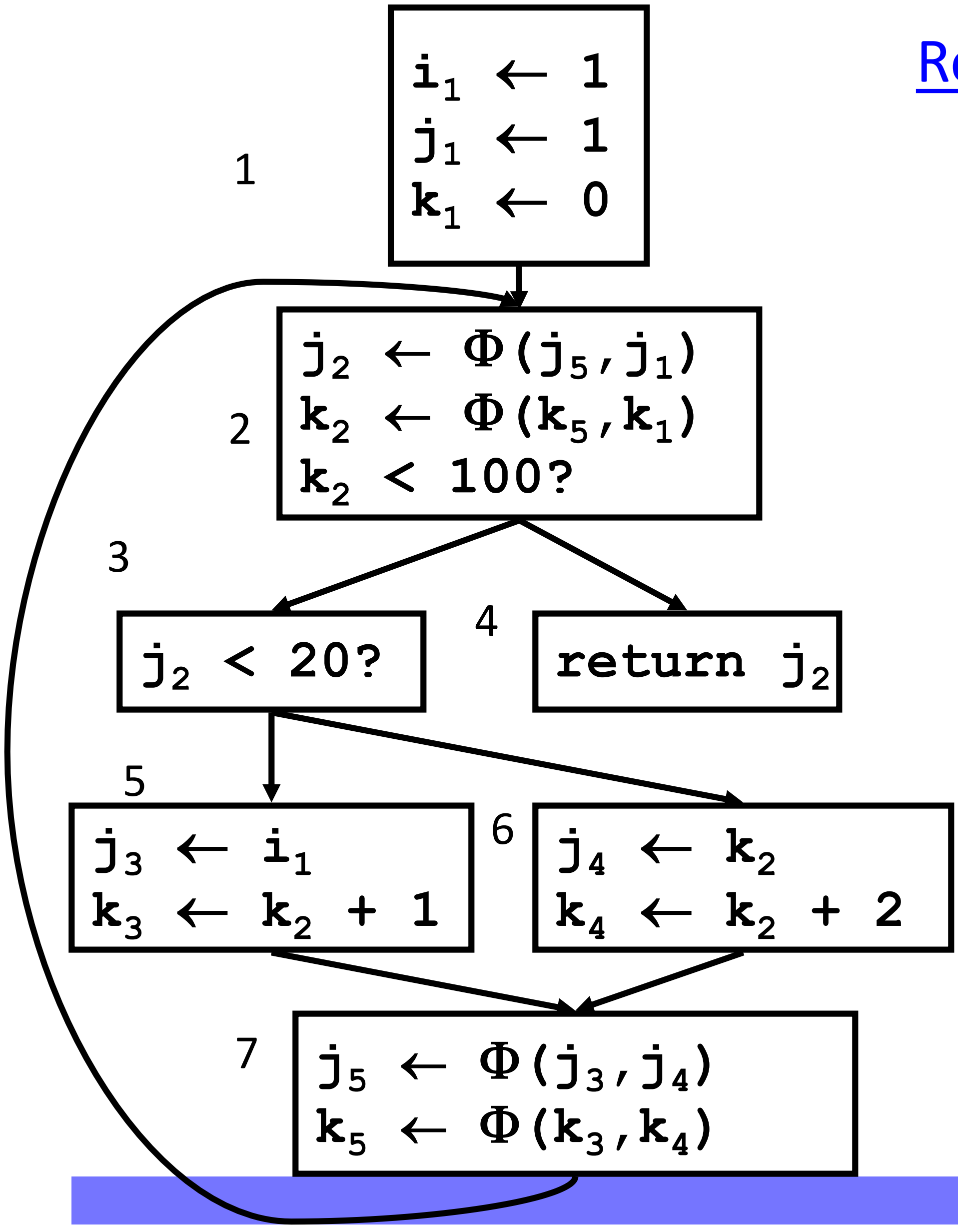


Rename Vars



rename(B):
for each assignment in B:
    replace use of v with top of stack(v)
    replace def of v with v_{new},
        push v_{new} onto stack(v)
for each successor S of B in CFG:
    replace k'th arg. of **Φ(v,…v)** with top of
        stack(v), where B is k'th predecessor of S
call rename(C) on all children C of B in D-tree
pop all defs in B from stacks

Rename Vars: Final Result

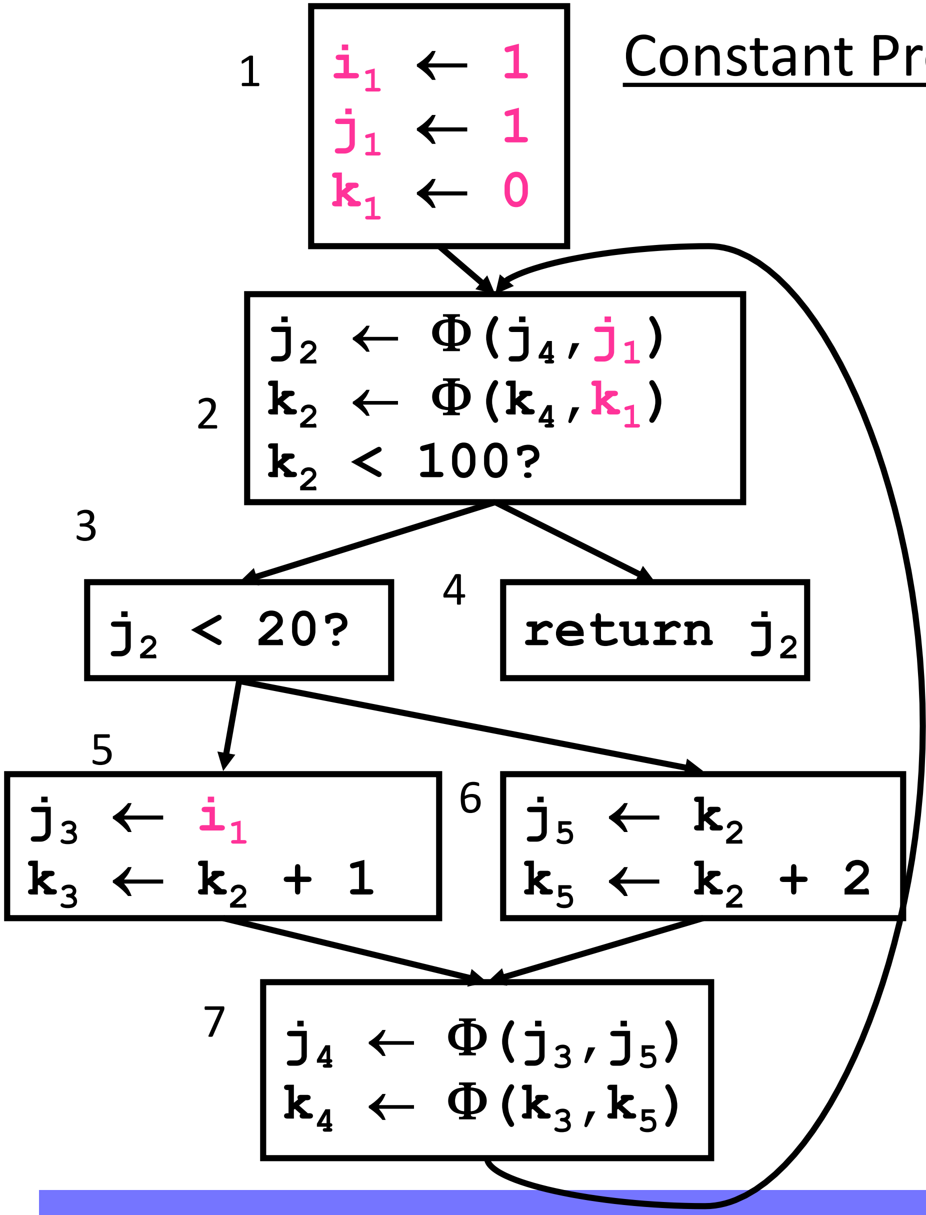


IV. Constant Propagation with SSA

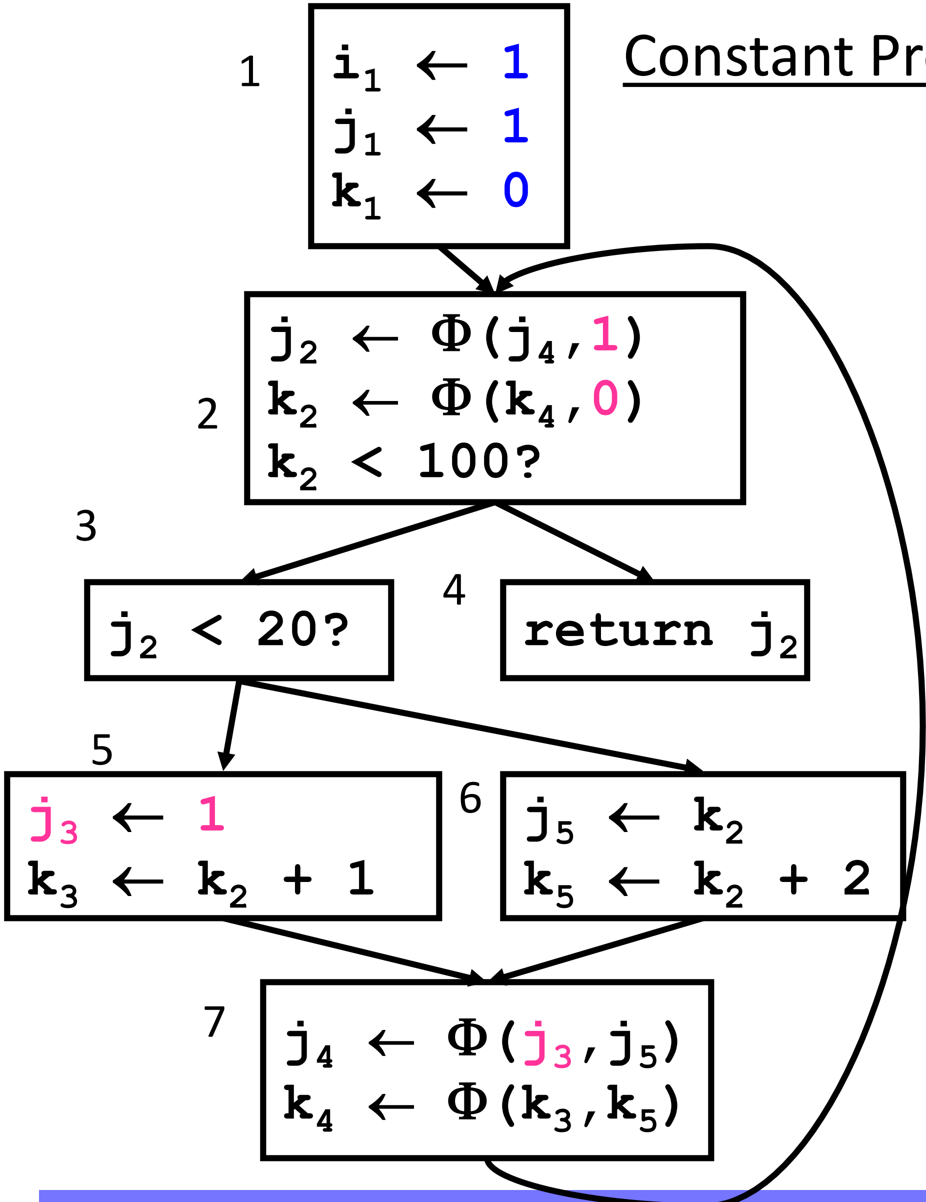
- If "v ← c", replace all uses of v with c
- If "v ← Φ(c,c,c)" (each input is the same constant), replace all uses of v with c

```
W ← list of all defs
while !W.isEmpty {
      Stmt S ← W.removeOne
      if ((S has form "v ← c") ||
          (S has form "v ← Φ(c,…,c)")) then {
          delete S
          foreach stmt U that uses v {
            replace v with c in U
            W.add(U)
          }
      }
}
```

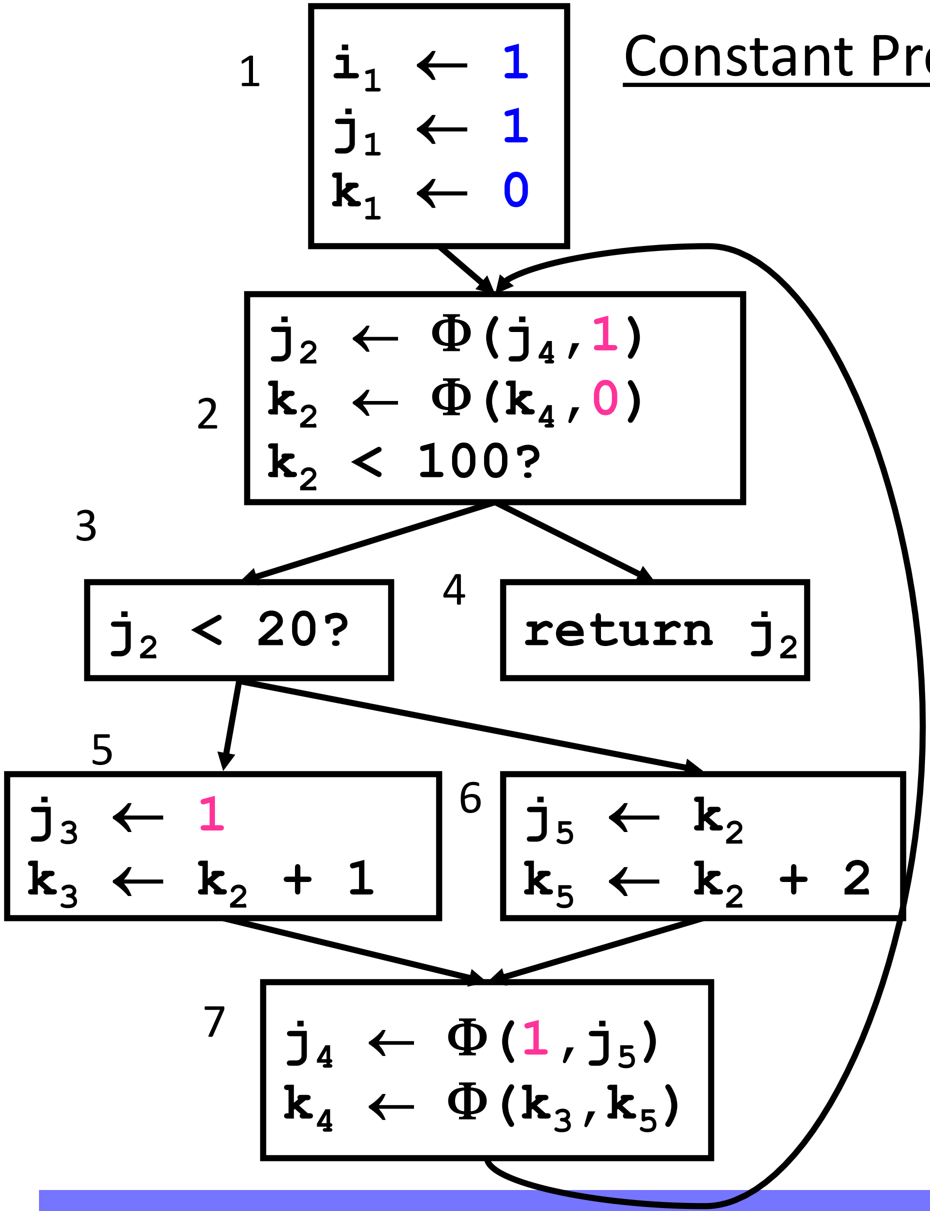
Constant Propagation



Constant Propagation

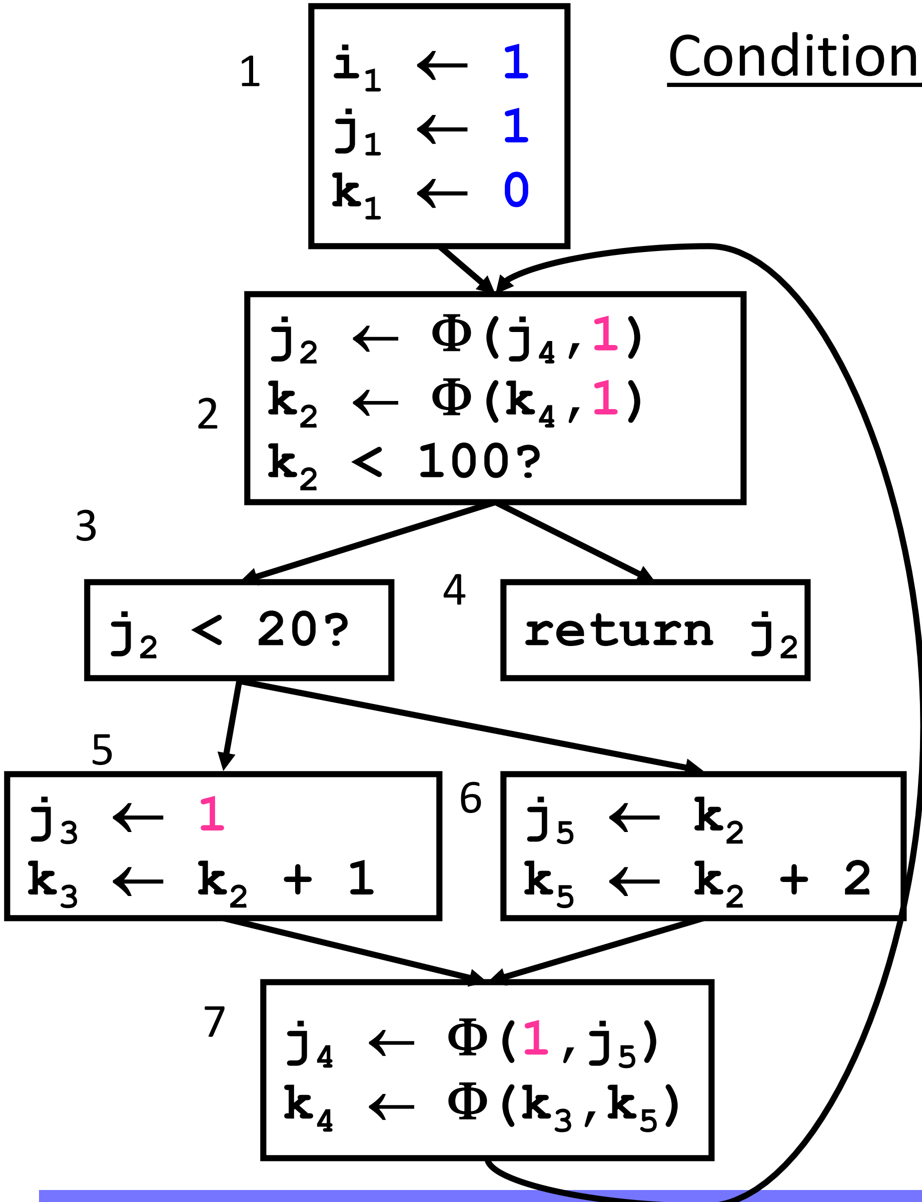


Constant Propagation



Not a very exciting result (yet)…

Conditional Constant Propagation



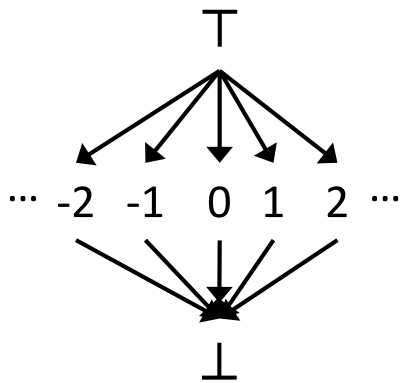
- Does block 6 ever execute?
- Simple Constant Propagation can't tell
- But "Conditional Const. Prop." *can* tell:
  - Assumes blocks don't execute until proven otherwise
  - Assumes values are constants until proven otherwise

Conditional Constant Propagation Algorithm

Keeps track of:

- Blocks
  - assume unexecuted until proven otherwise
- Variables
  - assume not executed (only with proof of assignments of a non-constant value do we assume not constant)

Lattice for representing variables:

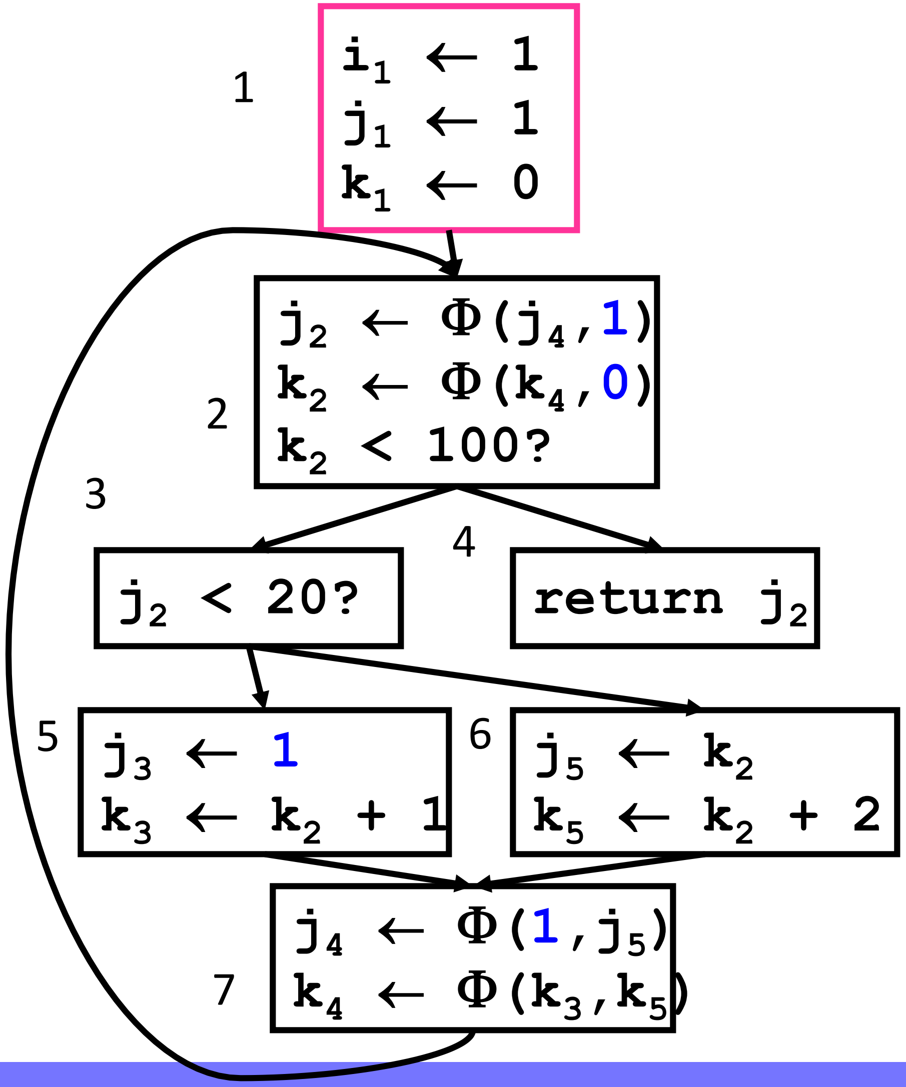


not executed

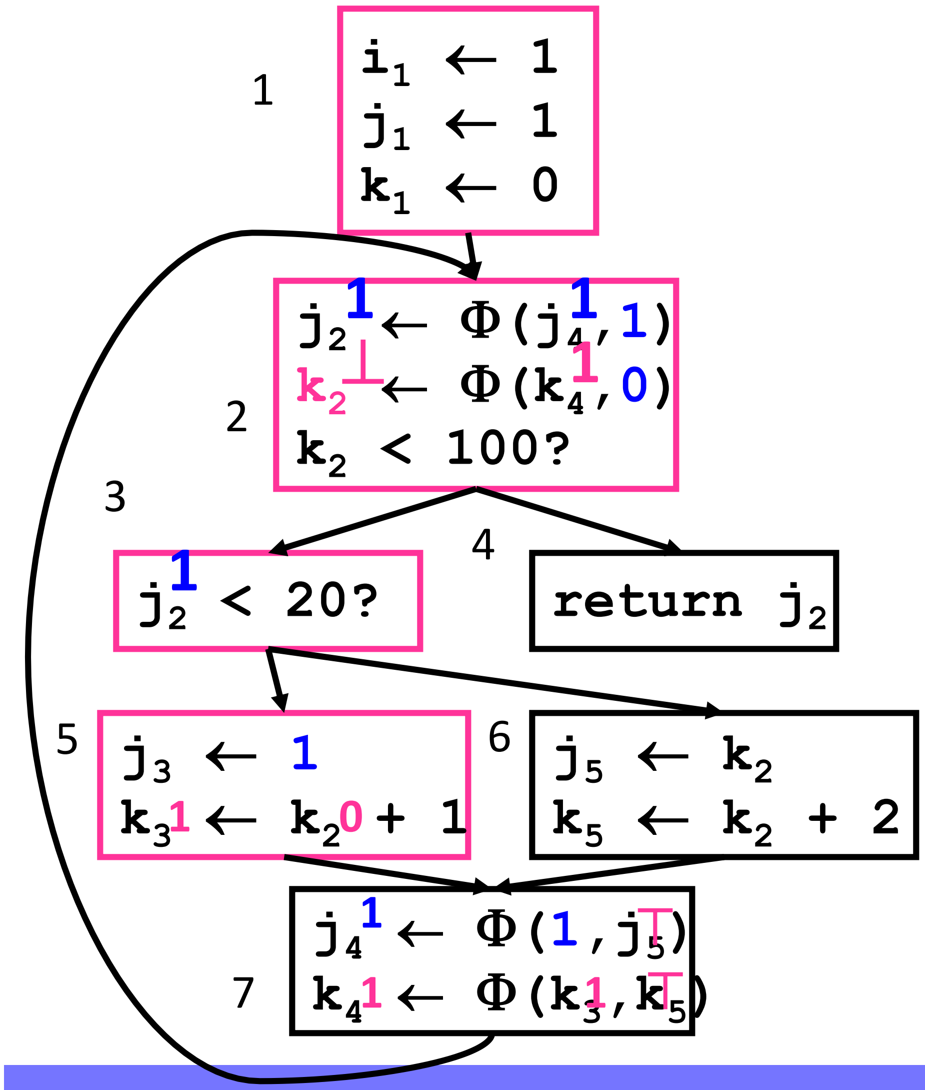
we have seen evidence that the variable has been assigned a constant with the value

we have seen evidence that the variable can hold different values at different times

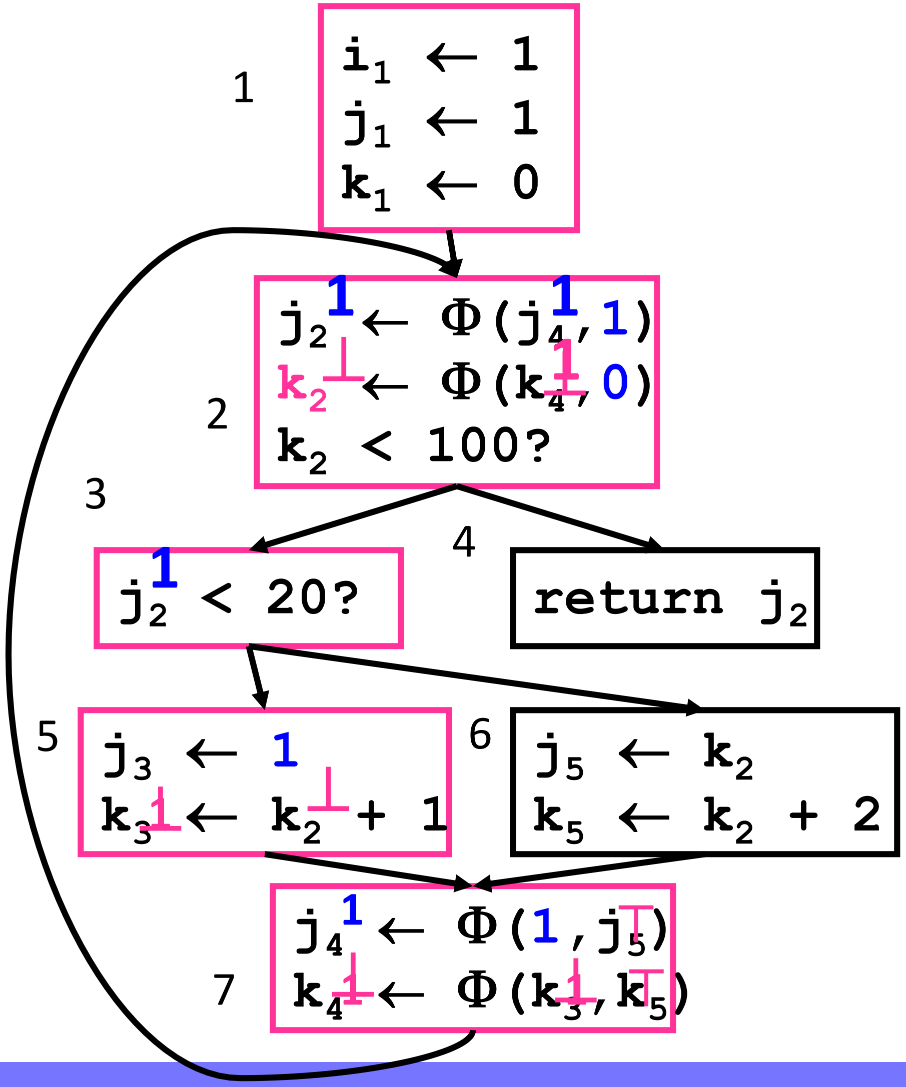
Conditional Constant Propagation



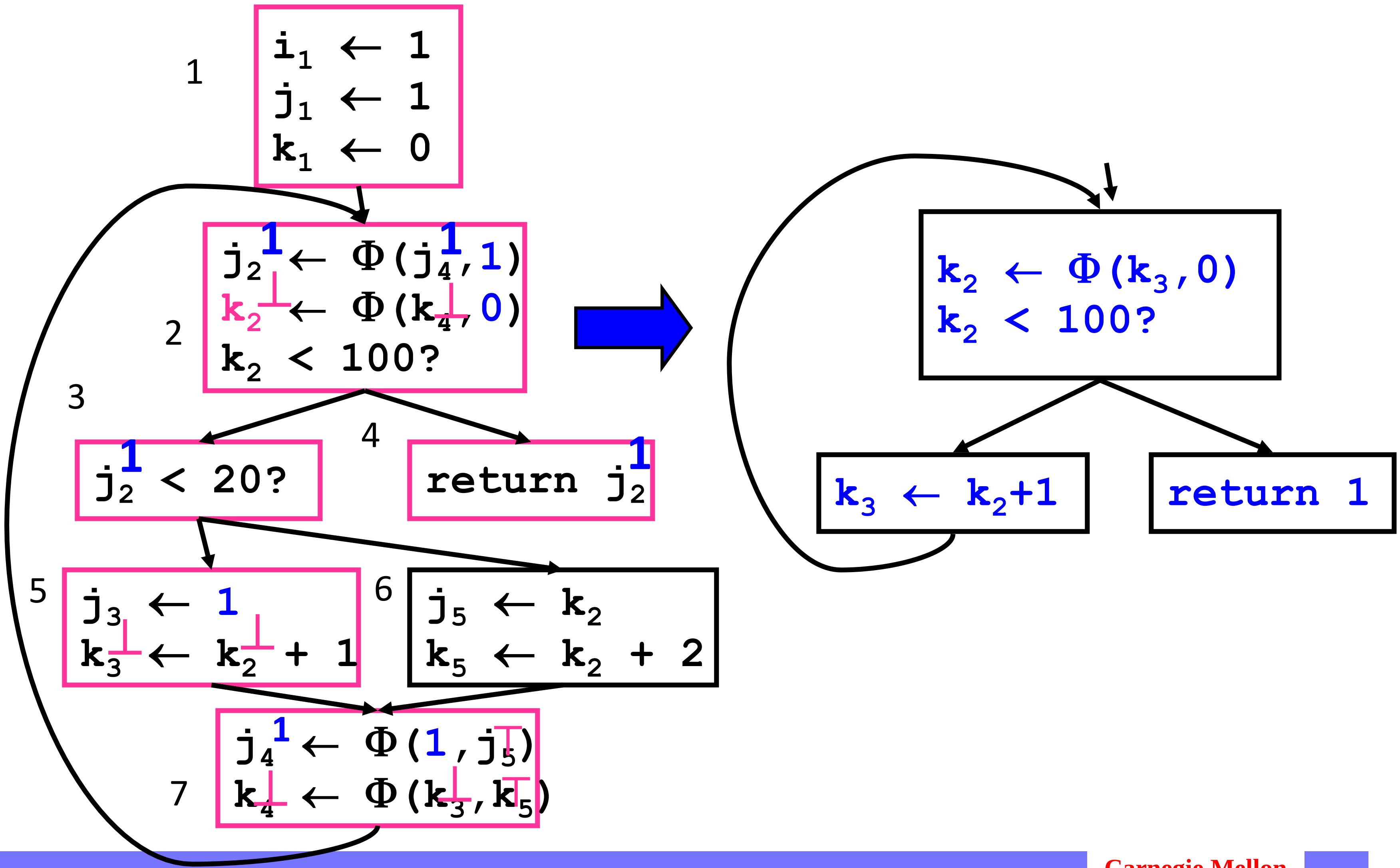
Conditional Constant Propagation



Conditional Constant Propagation



Conditional Constant Propagation



Today's Class

I. Review: Intro to SSA

II. When/Where to Insert Φ

III. Example

IV. Constant Propagation with SSA

Monday's Class

- Register Allocation
  - ALSU 8.8