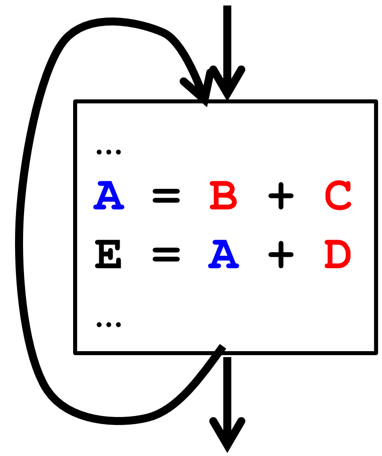
Static Single Assignment (SSA)

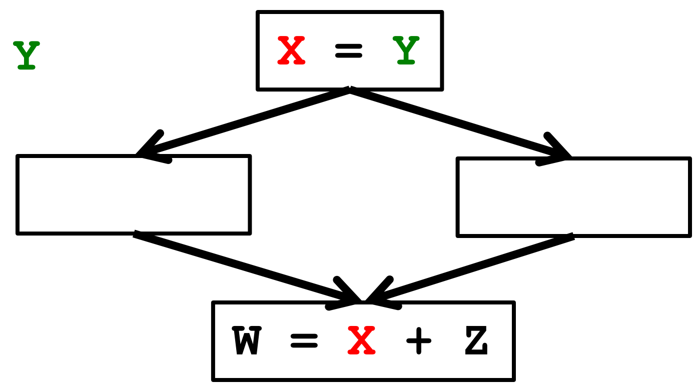
The content of this lecture is adapted from the lectures of Gennady Pekhimenko, Todd Mowry and Phillip Gibbons

Where Is a Variable Defined or Used?

- Example: Loop-Invariant Code Motion
  - Are **B**, **C**, and **D** only defined outside the loop?
  - Other definitions of **A** inside the loop?
  - Uses of **A** inside the loop?

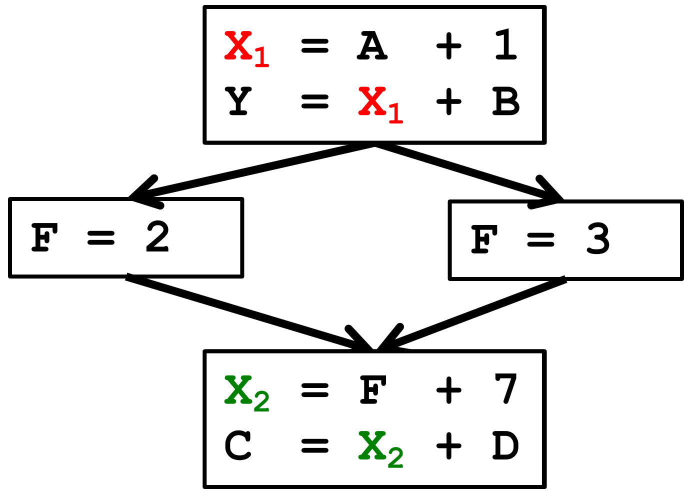


- Example: Copy Propagation
  - For a given use of **X**:
    - Are all reaching definitions of **X**:
      - copies from same variable: e.g., **X = Y**
    - Where **Y** is not redefined since that copy?
  - If so, substitute use of **X** with use of **Y**



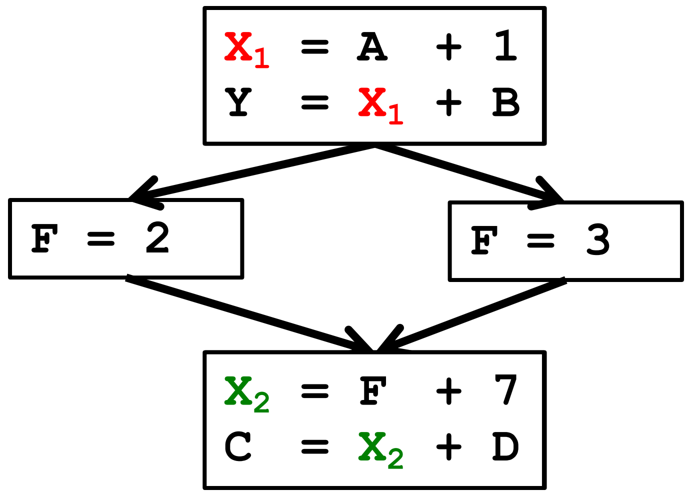
- It would be nice if we could *traverse directly* between related uses and def's
  - this would enable a form of *sparse* code analysis (skip over "don't care" cases)

Appearances of Same Variable Name May Be Unrelated



- The values in reused storage locations may be provably independent
  - in which case the compiler can optimize them as separate values
- Compiler could use renaming to make these different versions more explicit

Definition-Use and Use-Definition Chains



- Use-Definition (UD) Chains:
  - for a given definition of a variable X, what are all of its uses?
- Definition-Use (DU) Chains:
  - for a given use of a variable X, what are all of the reaching definitions of X?

DU and UD Chains Can Be Expensive

```
foo(int i, int j) {
   …
   switch (i) {
   case 0: x=3;break;
   case 1: x=1; break;
   case 2: x=6; break;
   case 3: x=7; break;
   default: x = 11;
   }
   switch (j) {
   case 0: y=x+7; break;
   case 1: y=x+4; break;
   case 2: y=x-2; break;
   case 3: y=x+1; break;
   default: y=x+9;
   }
   …
```

In general,

N defs

M uses

$\Rightarrow$ O(NM) space and time

<u>One solution</u>: limit each variable to ONE definition site

DU and UD Chains Can Be Expensive (2)

```
foo(int i, int j) {
   …
   switch (i) {
   case 0: x=3; break;
   case 1: x=1; break;
   case 2: x=6;
   case 3: x=7;
   default: x = 11;
   }
   x1 = x; // x1 is one of the above x's
   switch (j) {
   case 0: y=x1+7;
   case 1: y=x1+4;
   case 2: y=x1-2;
   case 3: y=x1+1;
   default: y=x1+9;
   }
   …
```

<u>One solution</u>: limit each variable to ONE definition site

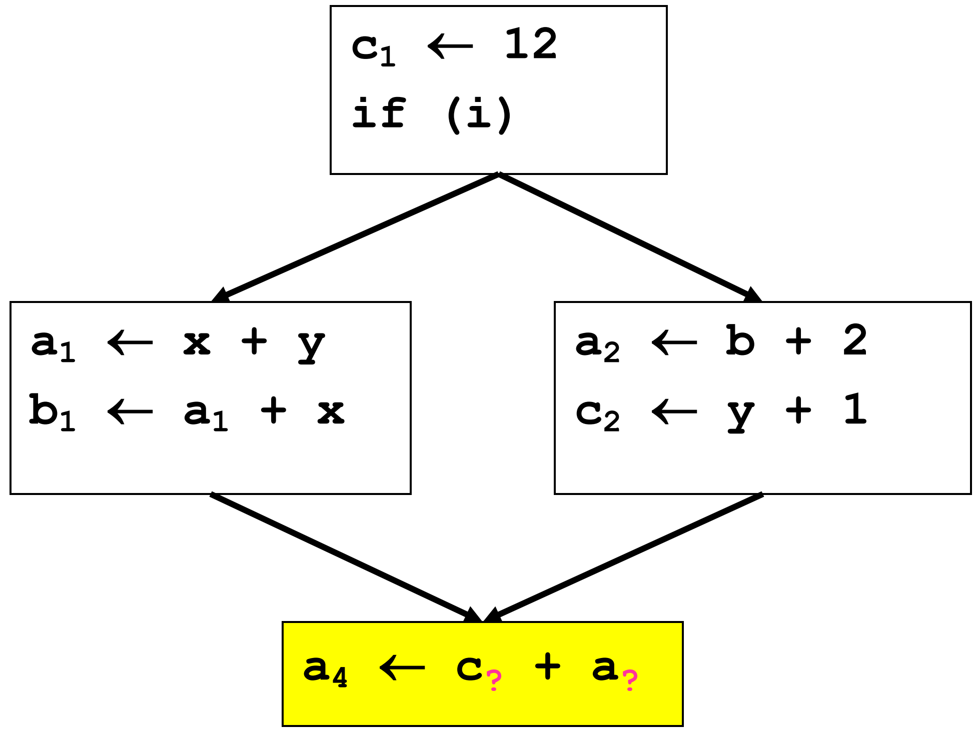
Static Single Assignment (SSA)

- 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

$$
\begin{array}{l}
\texttt{a} \leftarrow \texttt{x} + \texttt{y} \\
\texttt{b} \leftarrow \texttt{a} + \texttt{x} \\
\texttt{a} \leftarrow \texttt{b} + \texttt{2} \\
\texttt{c} \leftarrow \texttt{y} + \texttt{1} \\
\texttt{a} \leftarrow \texttt{c} + \texttt{a}
\end{array}
\quad \Rightarrow \quad
\begin{array}{l}
\texttt{a}_1 \leftarrow \texttt{x} + \texttt{y} \\
\texttt{b}_1 \leftarrow \texttt{a}_1 + \texttt{x} \\
\texttt{a}_2 \leftarrow \texttt{b}_1 + \texttt{2} \\
\texttt{c}_1 \leftarrow \texttt{y} + \texttt{1} \\
\texttt{a}_3 \leftarrow \texttt{c}_1 + \texttt{a}_2
\end{array}
$$

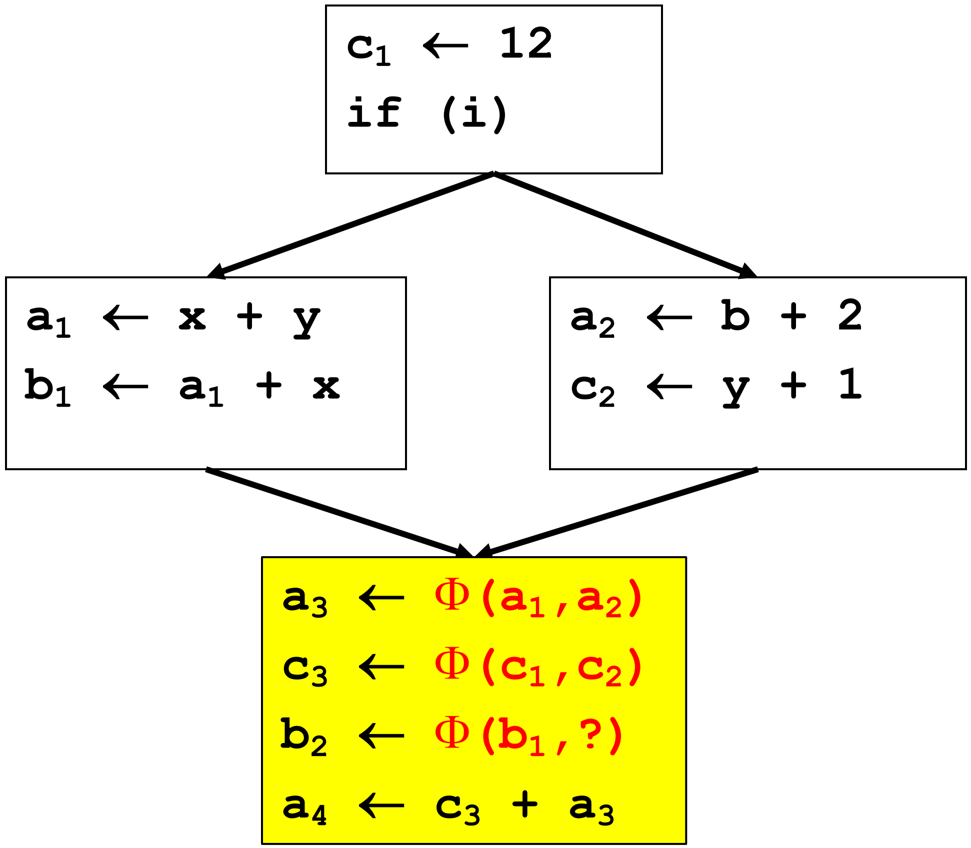
What about Joins in the CFG?

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



→ Use a notational fiction: a Φ function

Merging at Joins: the Φ function



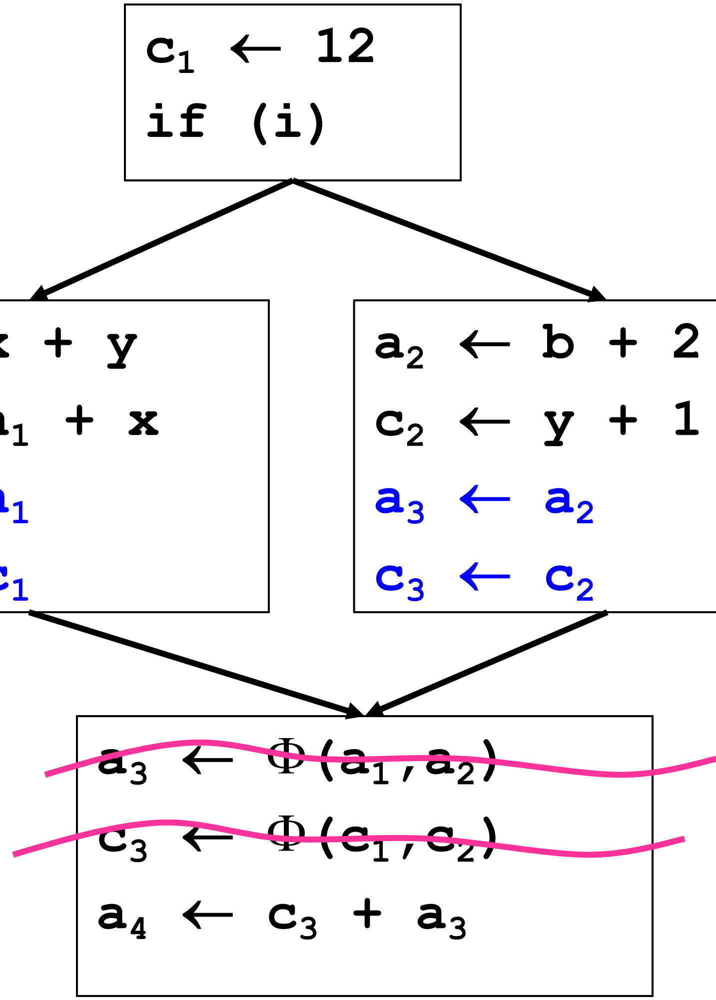
The Φ function

- 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.

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

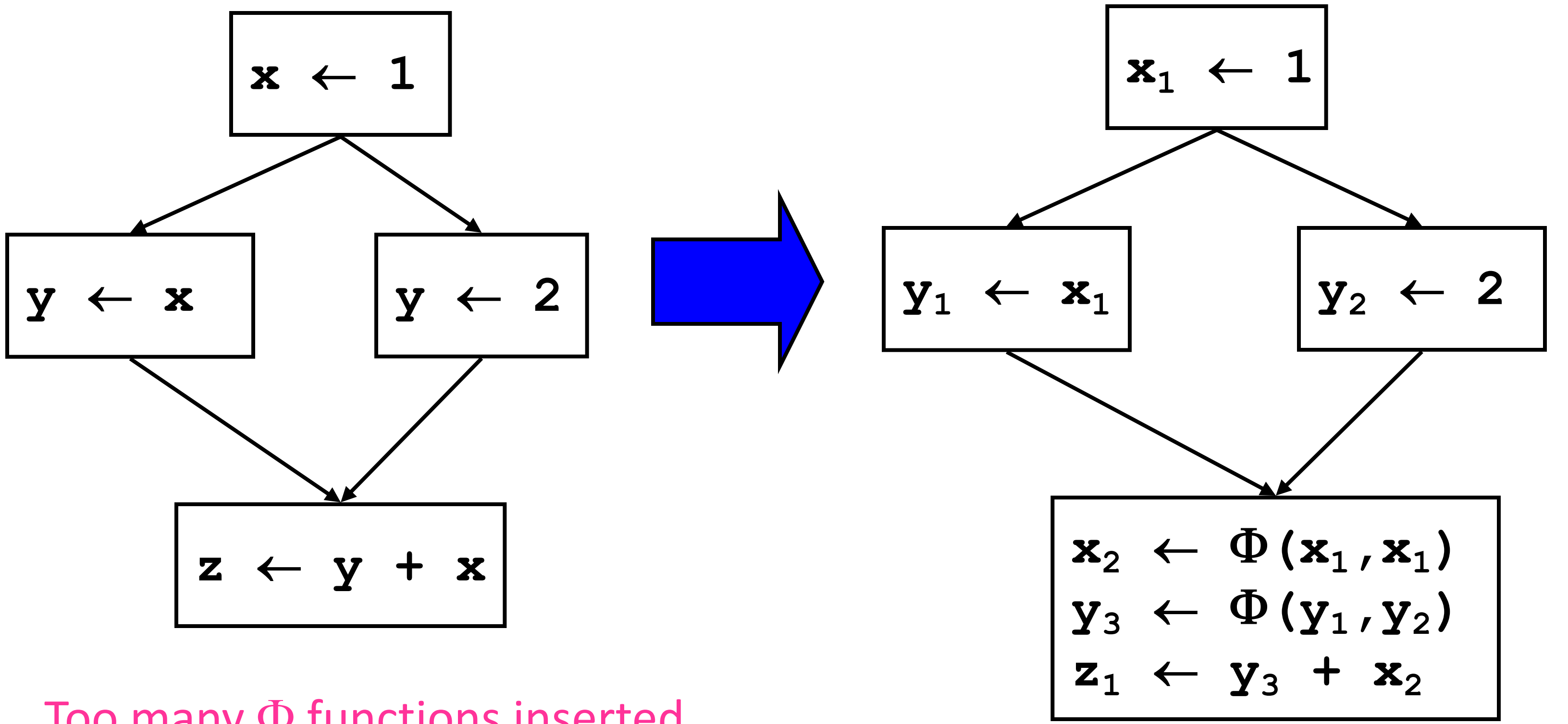
- How do we choose which x_i to use as value of Φ?
  - For static analysis we just use $\sqcup$
  - For code gen, add moves on each incoming edge

“Implementing” Φ



Trivial SSA

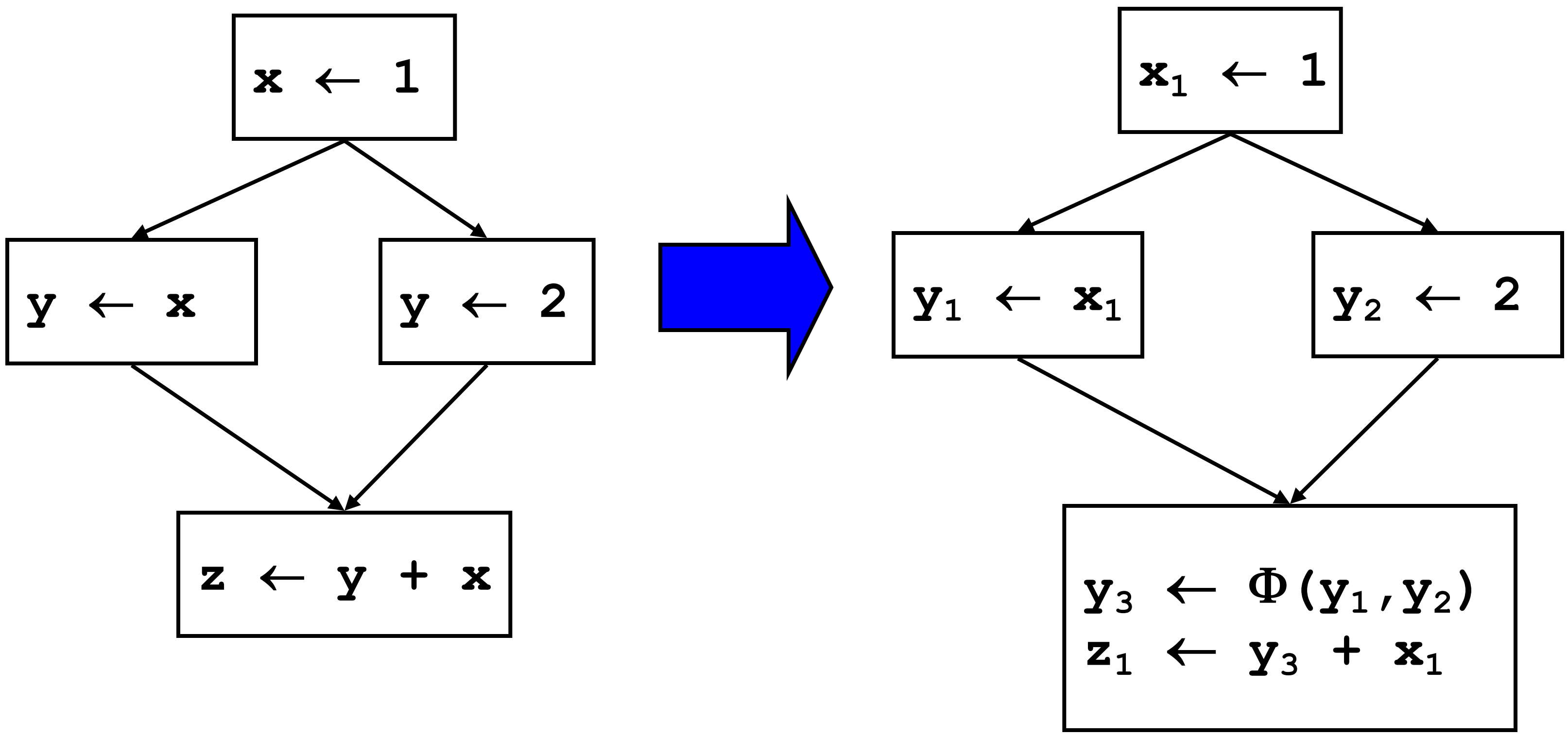
- Each assignment generates a fresh variable.
- At each join point insert Φ functions for all live variables.



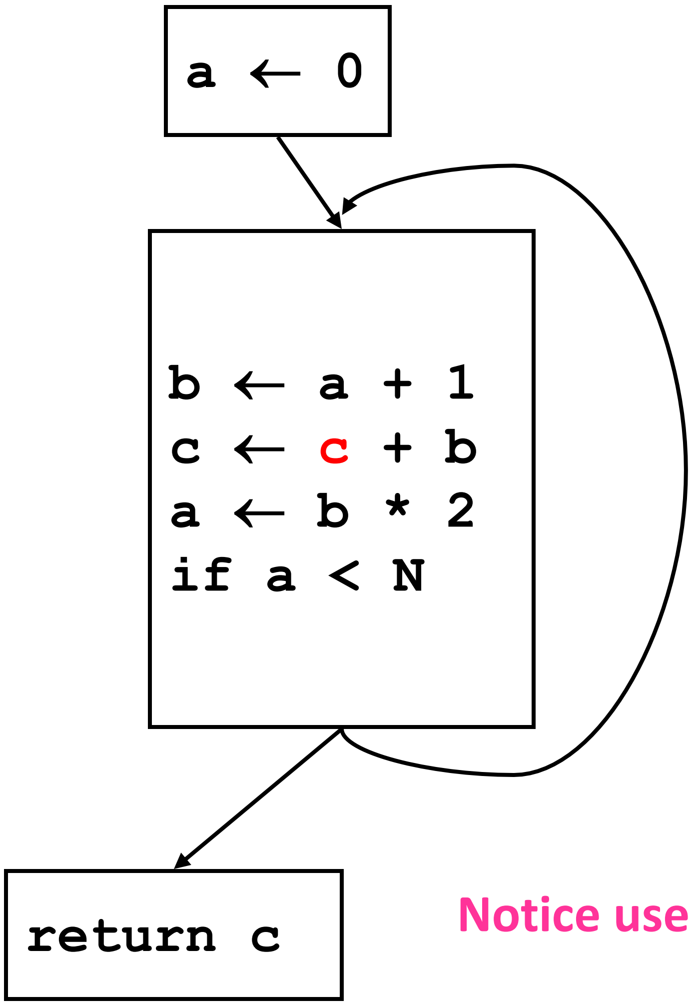
Too many Φ functions inserted.

Minimal SSA

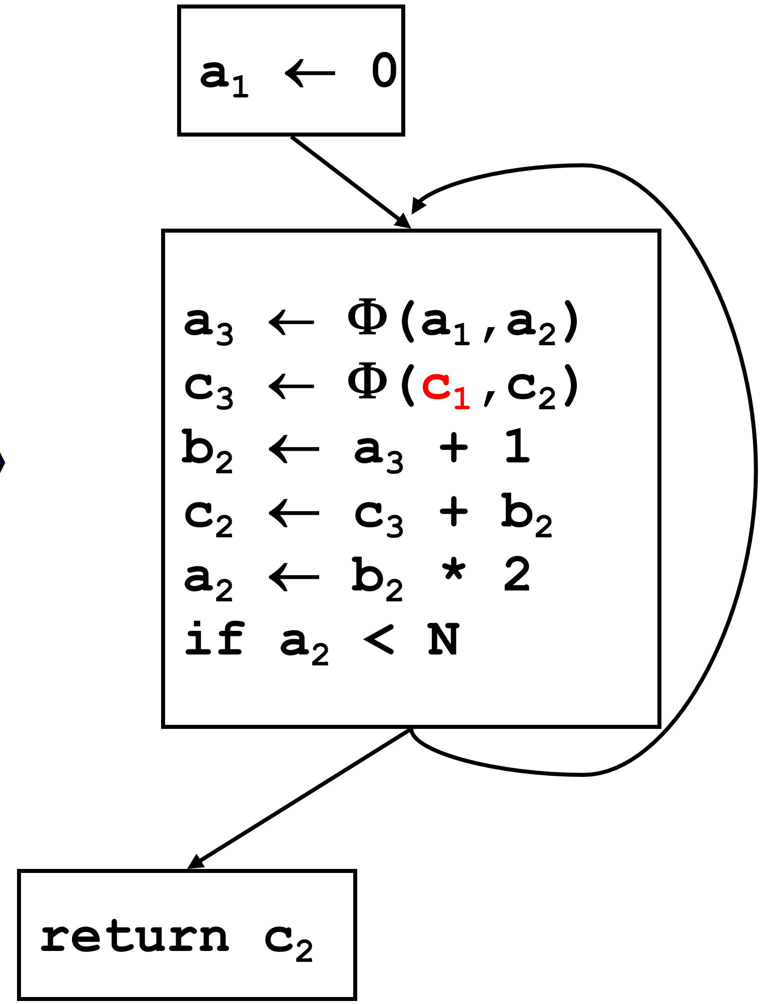
- Each assignment generates a fresh variable.
- At each join point insert Φ functions for all live variables with multiple outstanding defs.



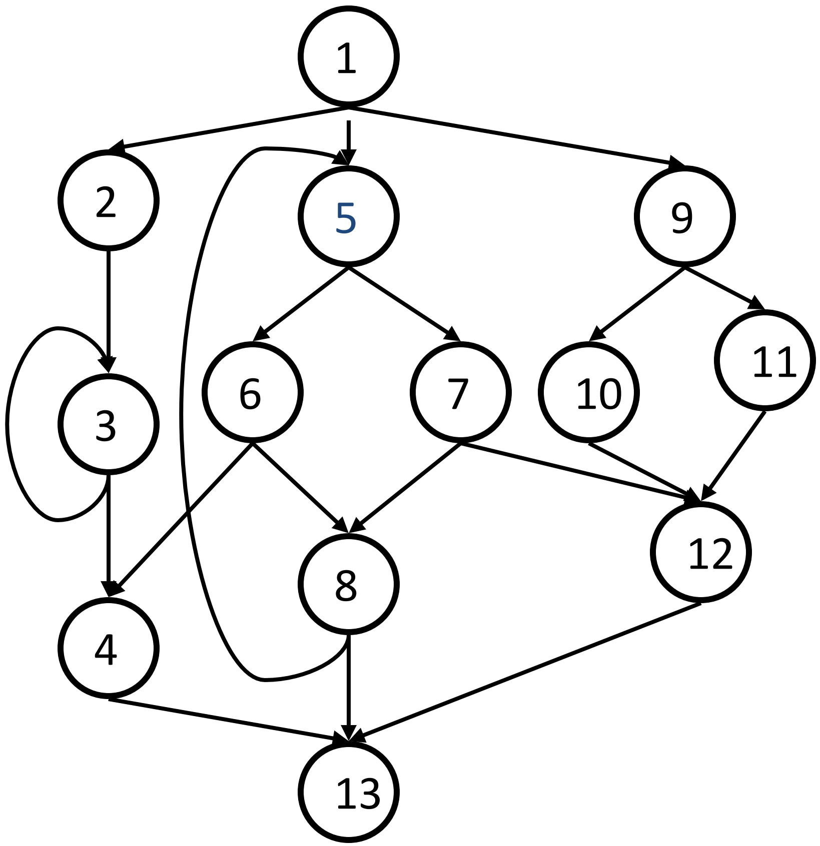
Another Example



Notice use of c_1



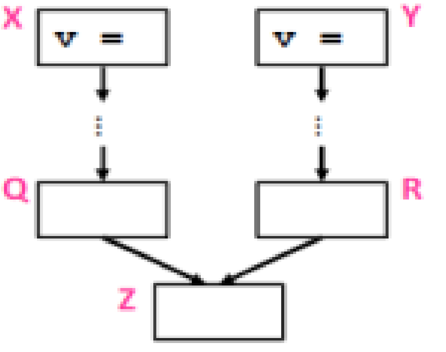
When Do We Insert Φ?



CFG

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

When do we insert Φ?



- We insert a Φ function for variable **A** in block Z iff:
  - **A** was defined more than once before
    - (i.e., **A** defined in **X** and Y AND $X \neq Y$)
  - There exists a non-empty path from x to z, P_{xz},
  - and a non-empty path from y to z, P_{yz}, s.t.
    - $P_{xz} \cap P_{yz} = \{ z \}$
      (Z is only common block along paths)
    - $z \notin P_{xq}$ or $z \notin P_{yr}$ where $P_{xz} = P_{xq} \rightarrow z$ and $P_{yz} = P_{yr} \rightarrow z$
      (at least one path reaches Z for first time)
- Entry block contains an implicit def of all vars
- Note: v = Φ(…) is a def of v

Control flow dominators

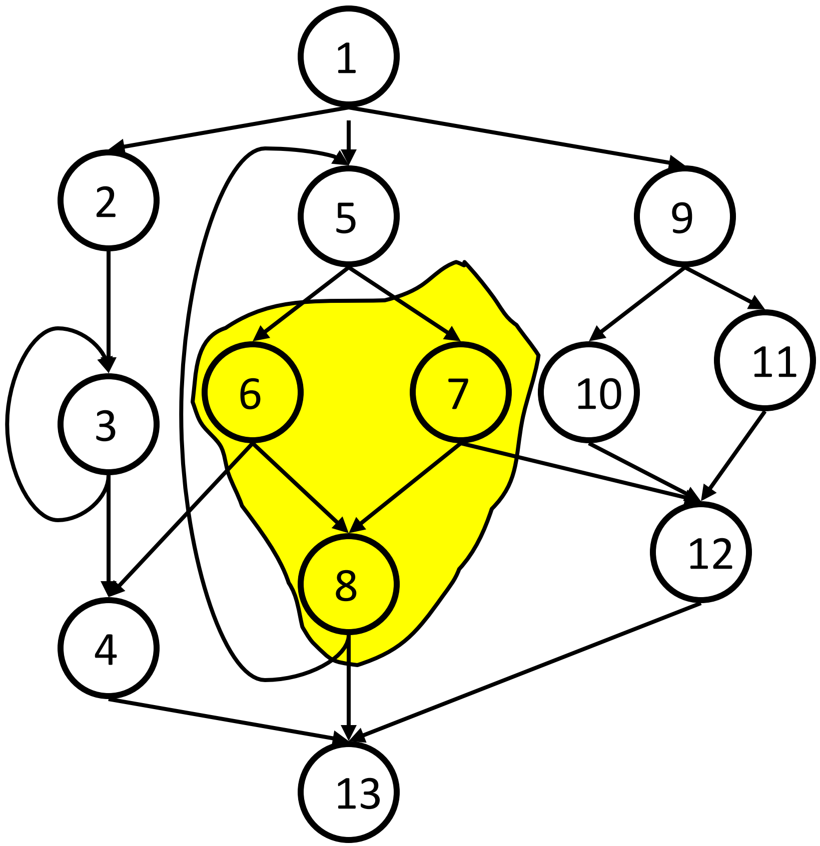
- Given two nodes, n and m, in a CFG
- **n dominates m** (aka **n dom m**)
  - if every path from the entry to **m** passes through **n**
- Every node trivially dominates itself
  - **n strictly dominates m** (aka **n sdom m**)
  - If **n** dom **m** and **n** $\neq$ **m**
- Dominance relation induces a tree-shaped ordering on CFG nodes

Dominance Property of SSA

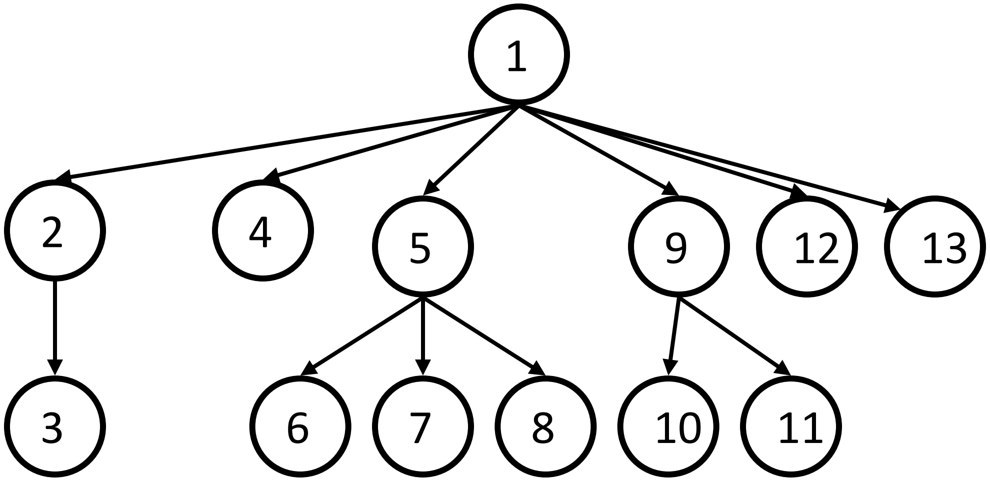
- In SSA, definitions dominate uses.
  - If x_i is used in $x \leftarrow \Phi(\ldots, x_i, \ldots)$, then $BB(x_i)$ dominates i^{th} predecessor of BB(PHI)
  - If x is used in $y \leftarrow \ldots x \ldots$, then BB(x) dominates BB(y)

- We can use this for an efficient algorithm to convert to SSA

Dominance



CFG

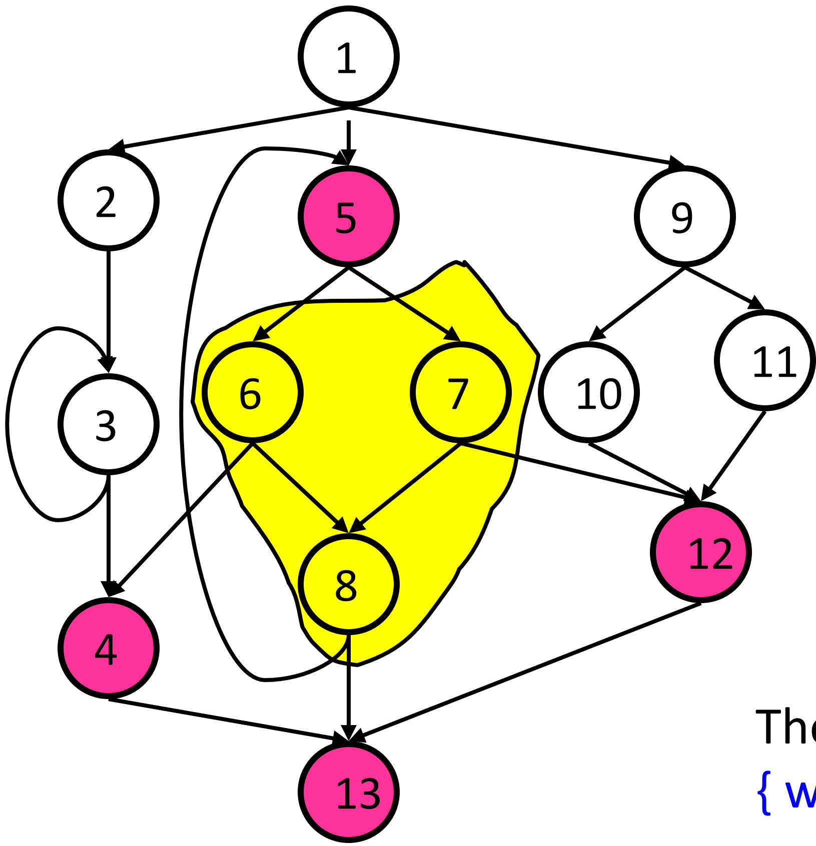


If there is a def of a in block 5, which nodes need a $\Phi()$?

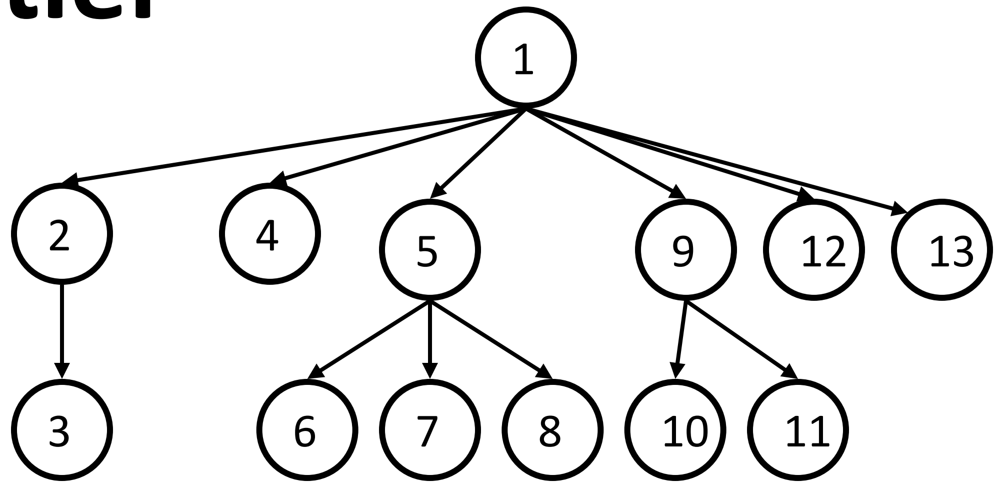
D-Tree

x strictly dominates w (x sdom w) iff x dom w AND $x \neq w$

Dominance Frontier



CFG

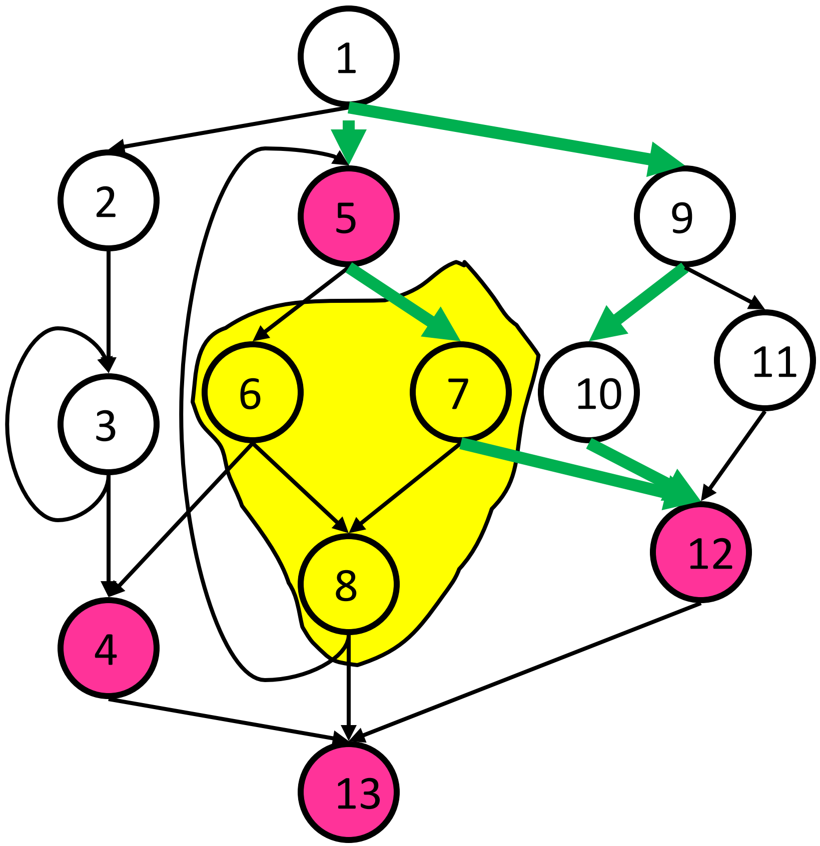


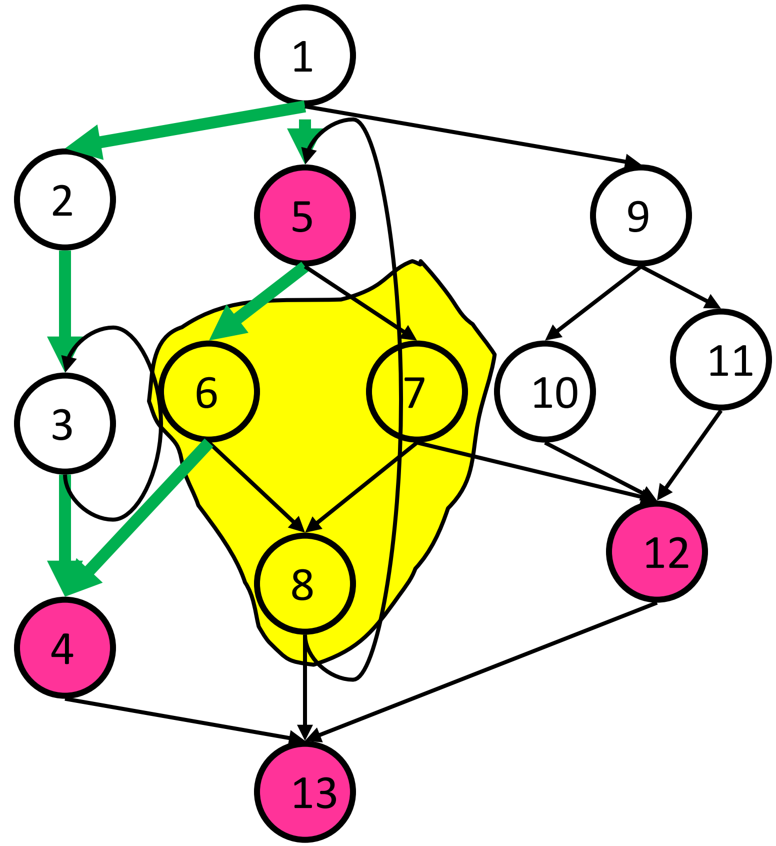
D-Tree

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

x strictly dominates w (x sdom w) iff x dom w AND x $\neq$ w

Dominance Frontier and Path Convergence





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

Using Dominance Frontier to Compute SSA

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

Using Dominance Frontier to Place $\Phi()$

- Gather all the defsites, i.e., assignments, 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

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

Using Dominance Frontier to Place Φ()

```
foreach node n {
   foreach variable v defined in n {
     varsdef[n] ∪= {v}
     defsites[v] ∪= {n}
   }
}
foreach variable v {
   W = defsites[v]
   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}
         if v ∉ varsdef[y]: W = W ∪ {y}
       }
     }
   }
}
```

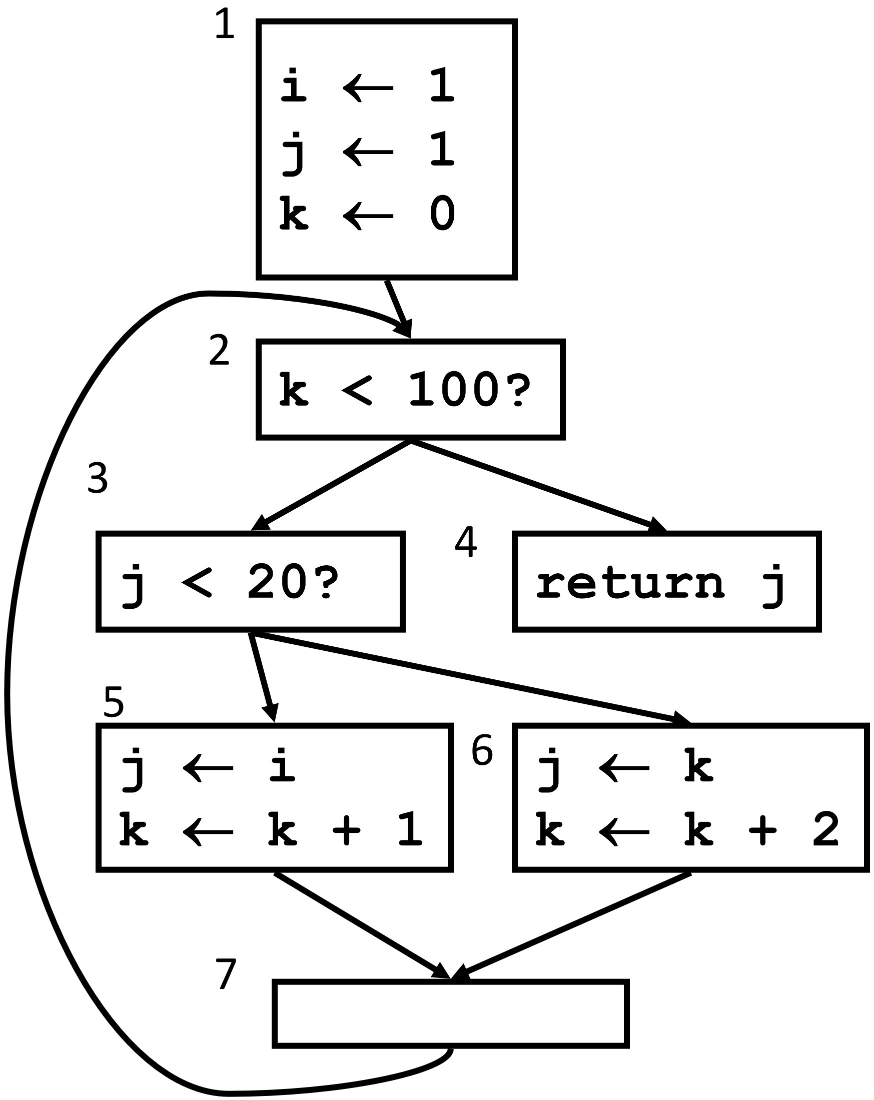
Renaming Variables

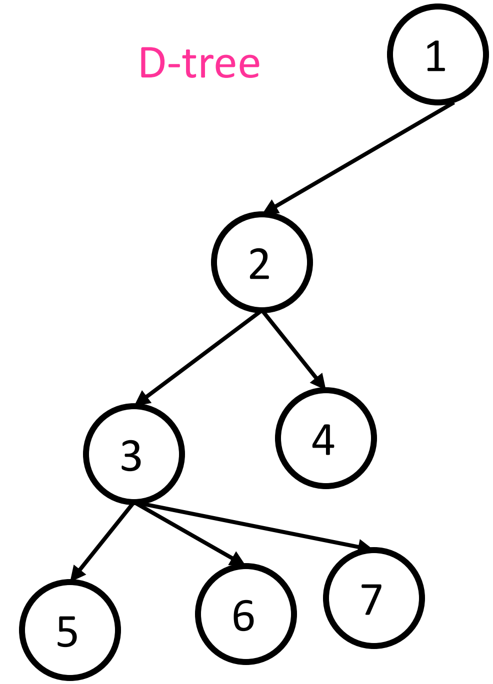
- Algorithm:
  - Walk the D-tree, renaming variables as you go
  - Replace uses with most recent renamed def

- 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

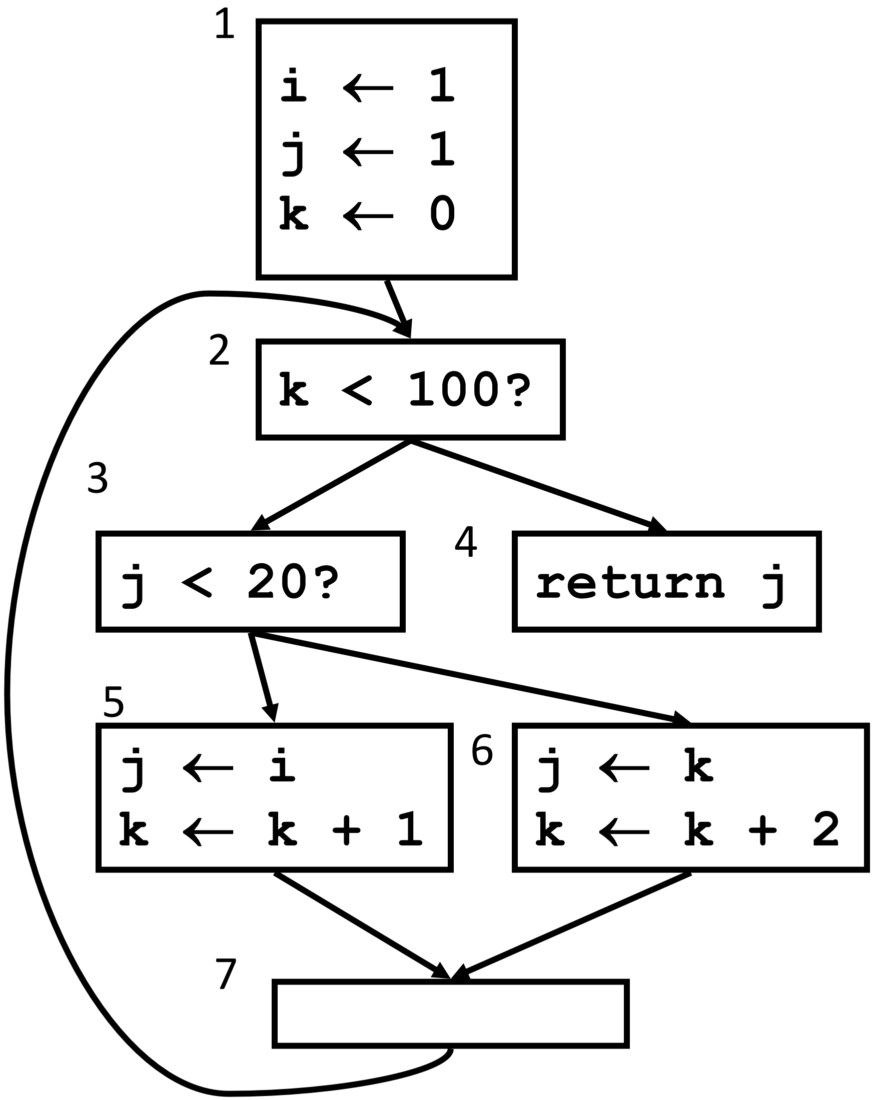
- Easy implementation:
  - for each var: rename (v)
  - rename(v): replace uses with top of stack
    at def: push onto stack
    call rename(v) on all children in D-tree
    if D-tree leaf, replace uses of CFG successors
    for each def in this block pop from stack

Compute Dominance Tree



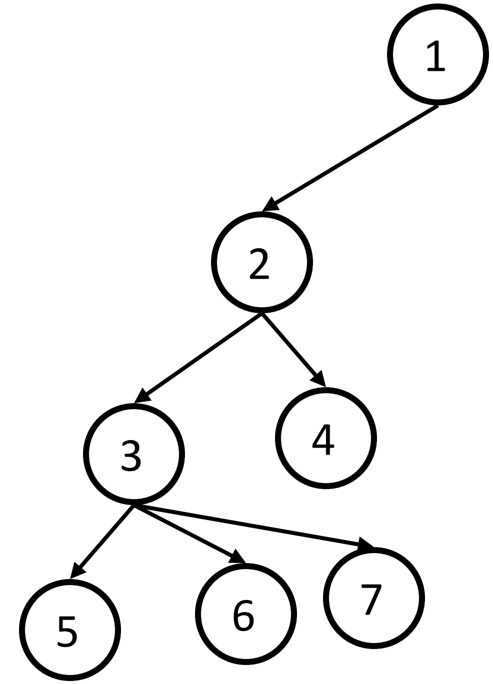


Compute Dominance Frontiers

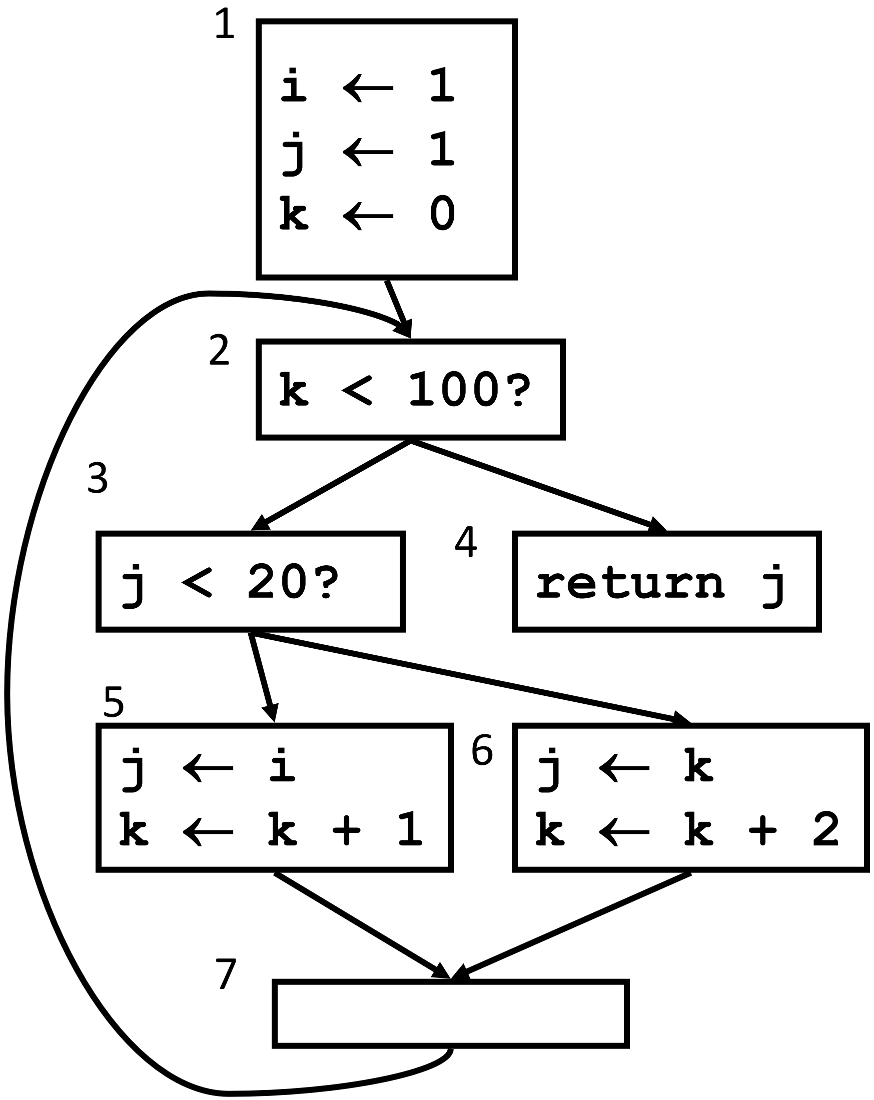


DFs

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



Insert $\Phi()$



DFs		varsdef[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}

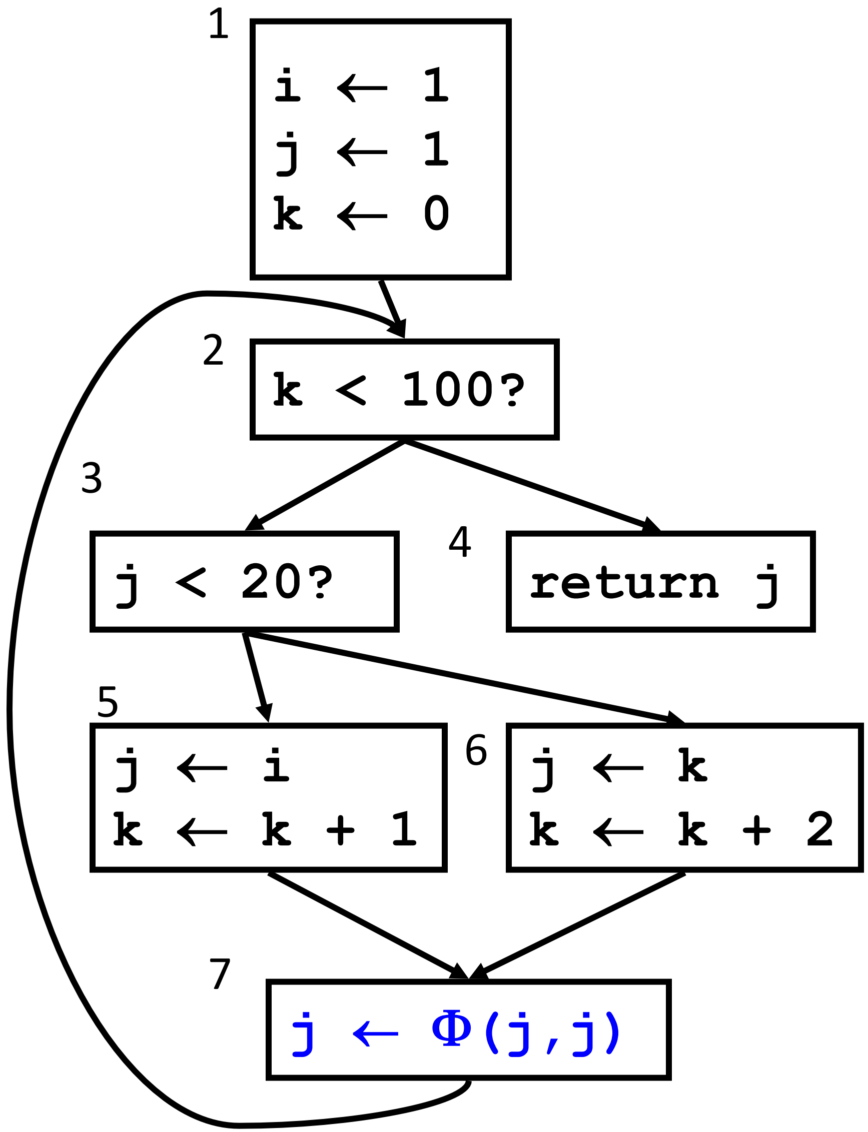
DFs

var i: W={1}

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

DF{1} DF{5}

Insert Φ()



DFs		varsdef[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	{}

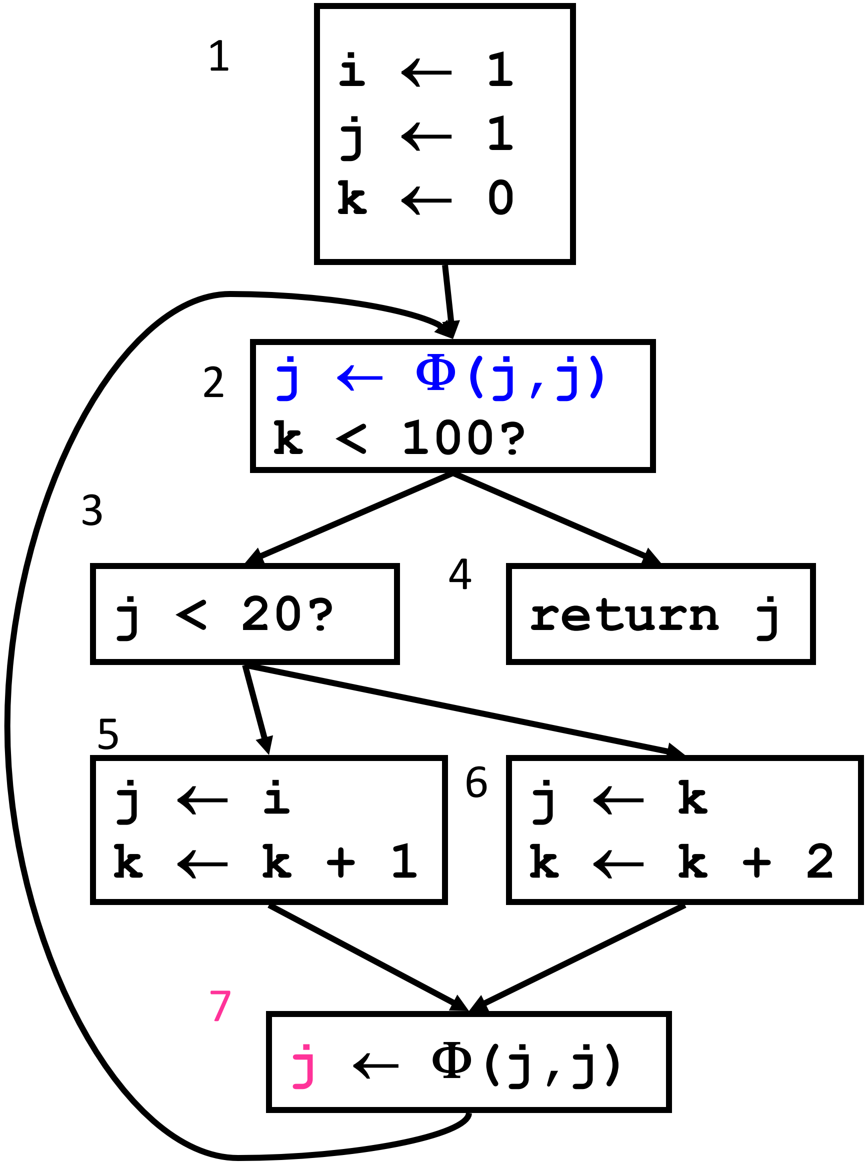
defsites[v]

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

DFs

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

DF{1} DF{5}



DFs		varsdef[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	{}

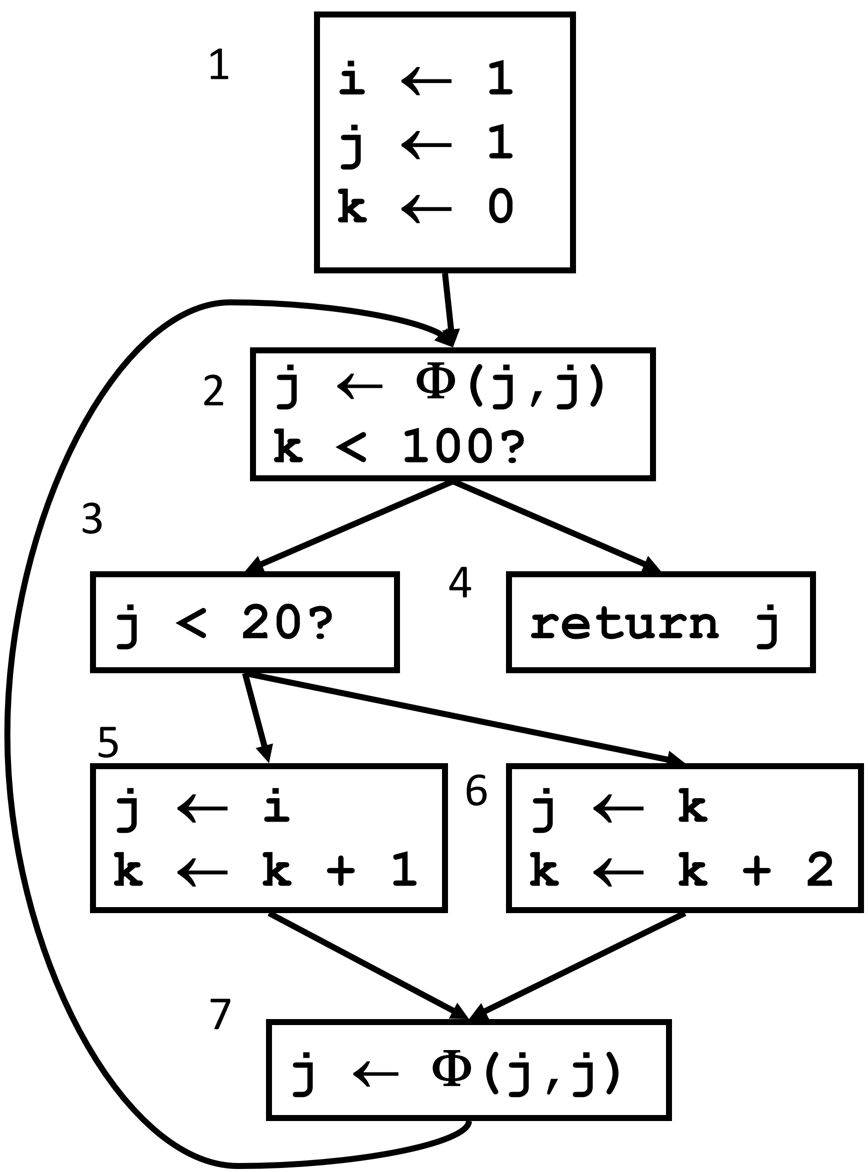
defsites[v]

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

DFs

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

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



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

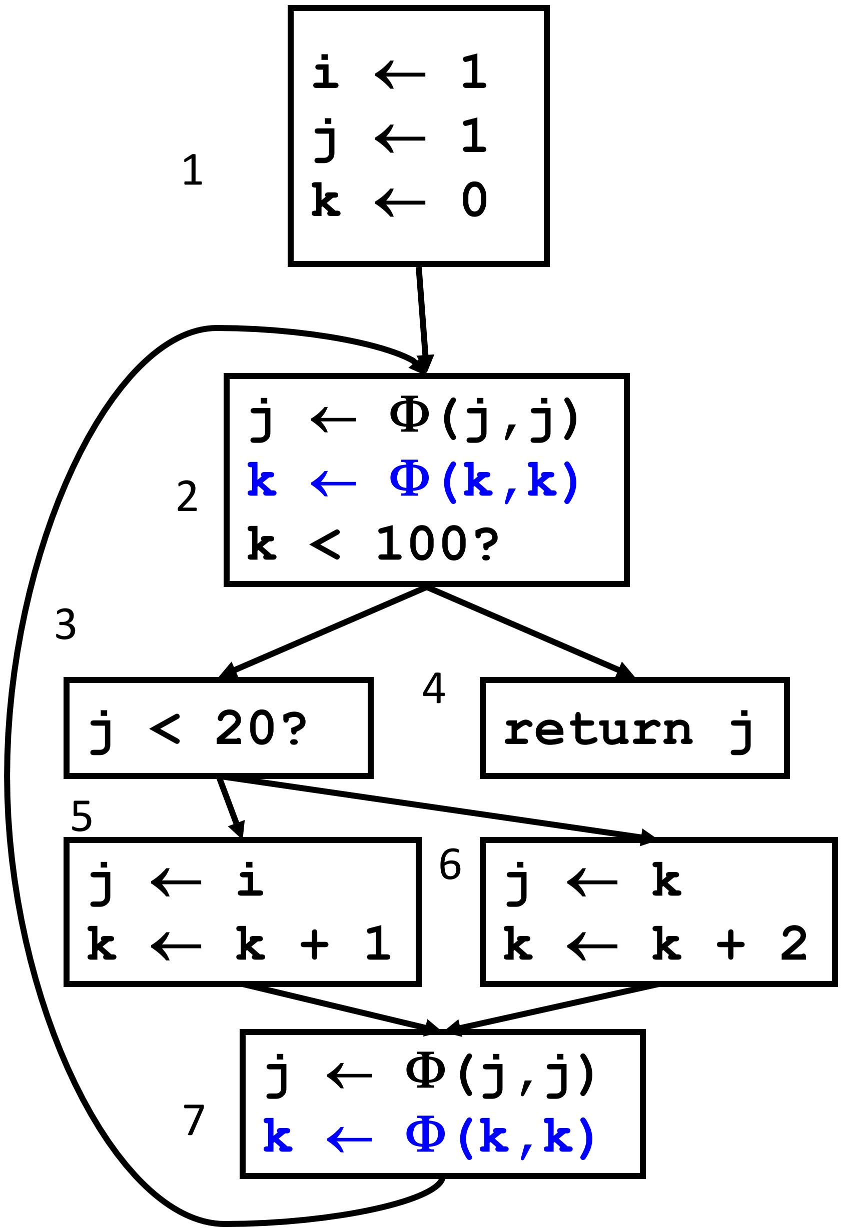
varsdef[n]	
1	{ i,j,k}
2	{}
3	{}
4	{}
5	{j,k}
6	{j,k}
7	{}

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

DFs

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

DF{1} DF{5} DF{7} DF{6}



DFs		varsdef[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	{}

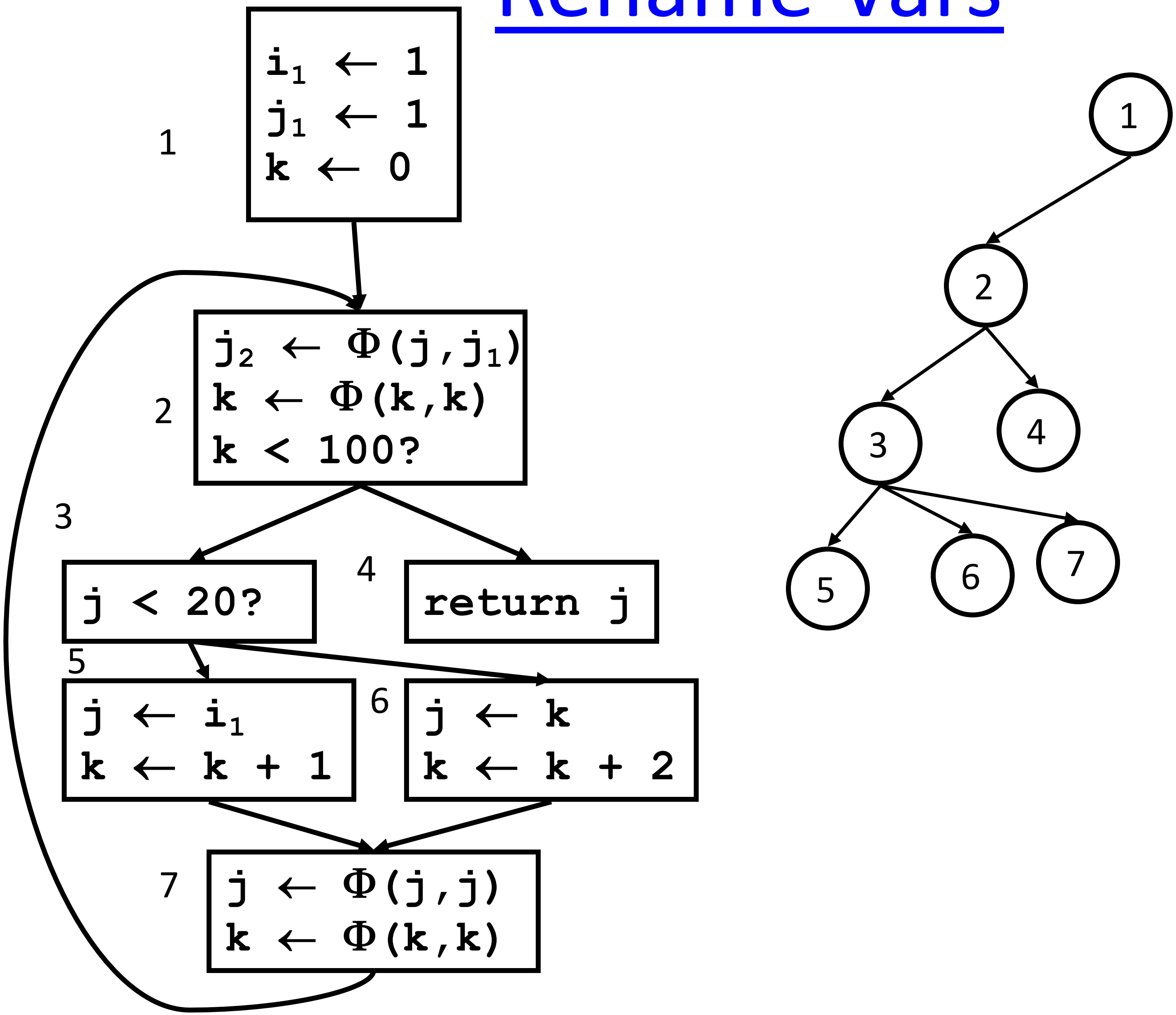
Def sites[v]

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

DFs

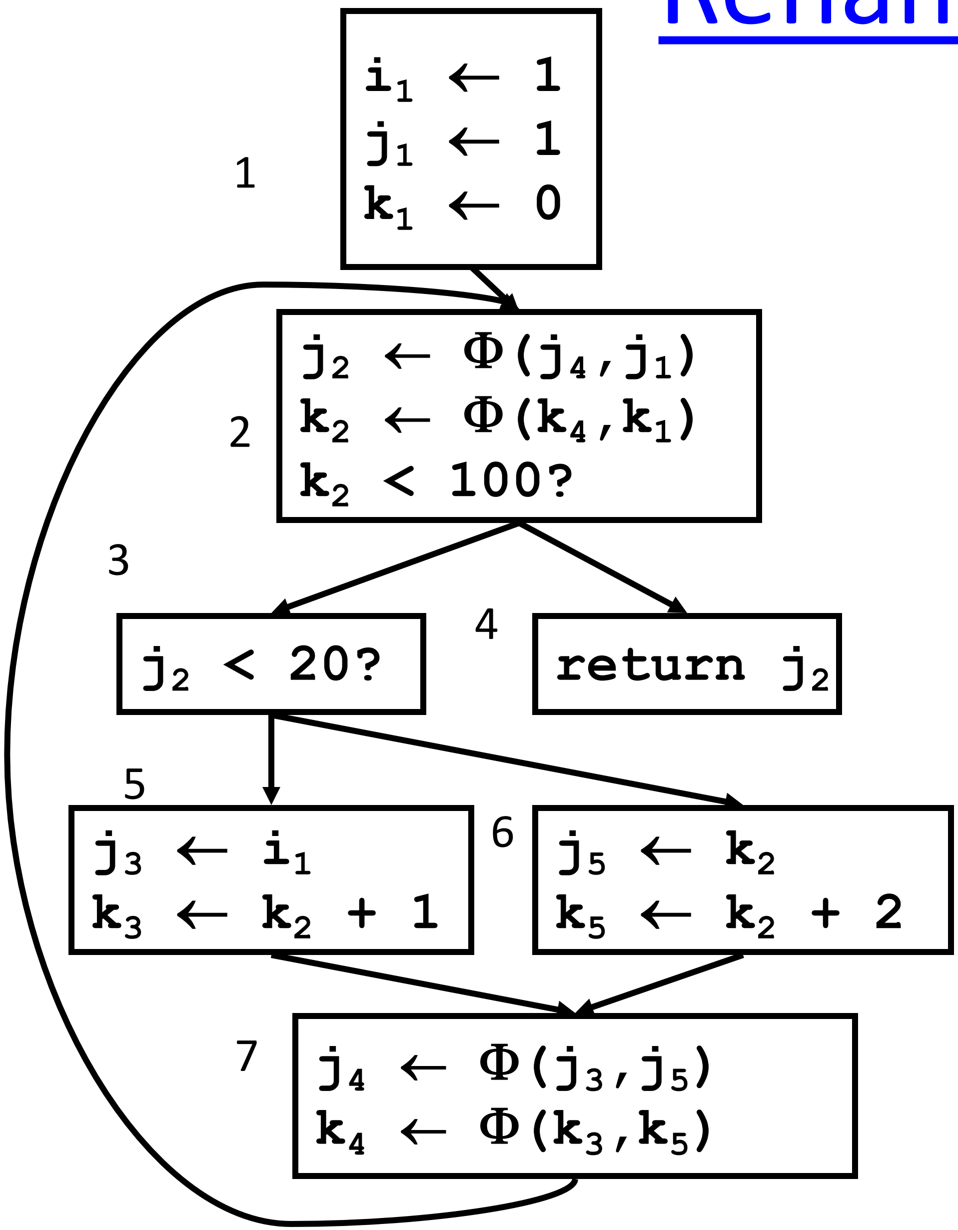
var k: W={1,5,6}

Rename Vars

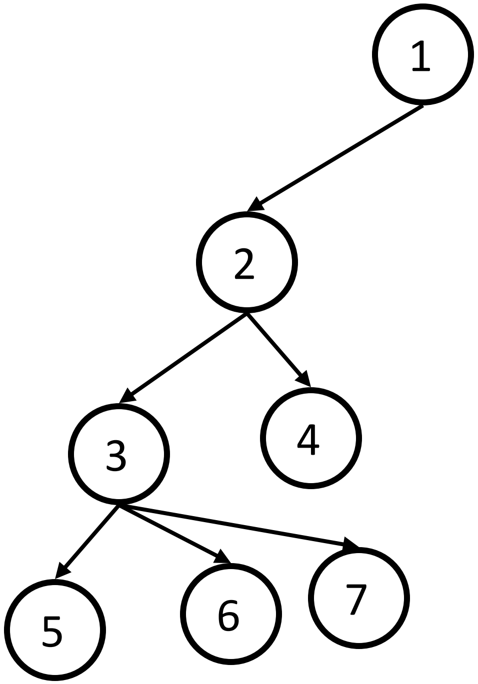


1
$i_1 \leftarrow 1$
$j_1 \leftarrow 1$
$k \leftarrow 0$
2
$j_2 \leftarrow \Phi(j, j_1)$
$k \leftarrow \Phi(k, k)$
$k < 100?$
3
$j < 20?$
4
return j
5
$j \leftarrow i_1$
$k \leftarrow k + 1$
6
$j \leftarrow k$
$k \leftarrow k + 2$
7
$j \leftarrow \Phi(j, j)$
$k \leftarrow \Phi(k, k)$
1
2
3
4
5
6
7

Rename Vars



1
$i_1 \leftarrow 1$
$j_1 \leftarrow 1$
$k_1 \leftarrow 0$
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

SSA Properties

- Only 1 assignment per variable
- Definitions dominate uses

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

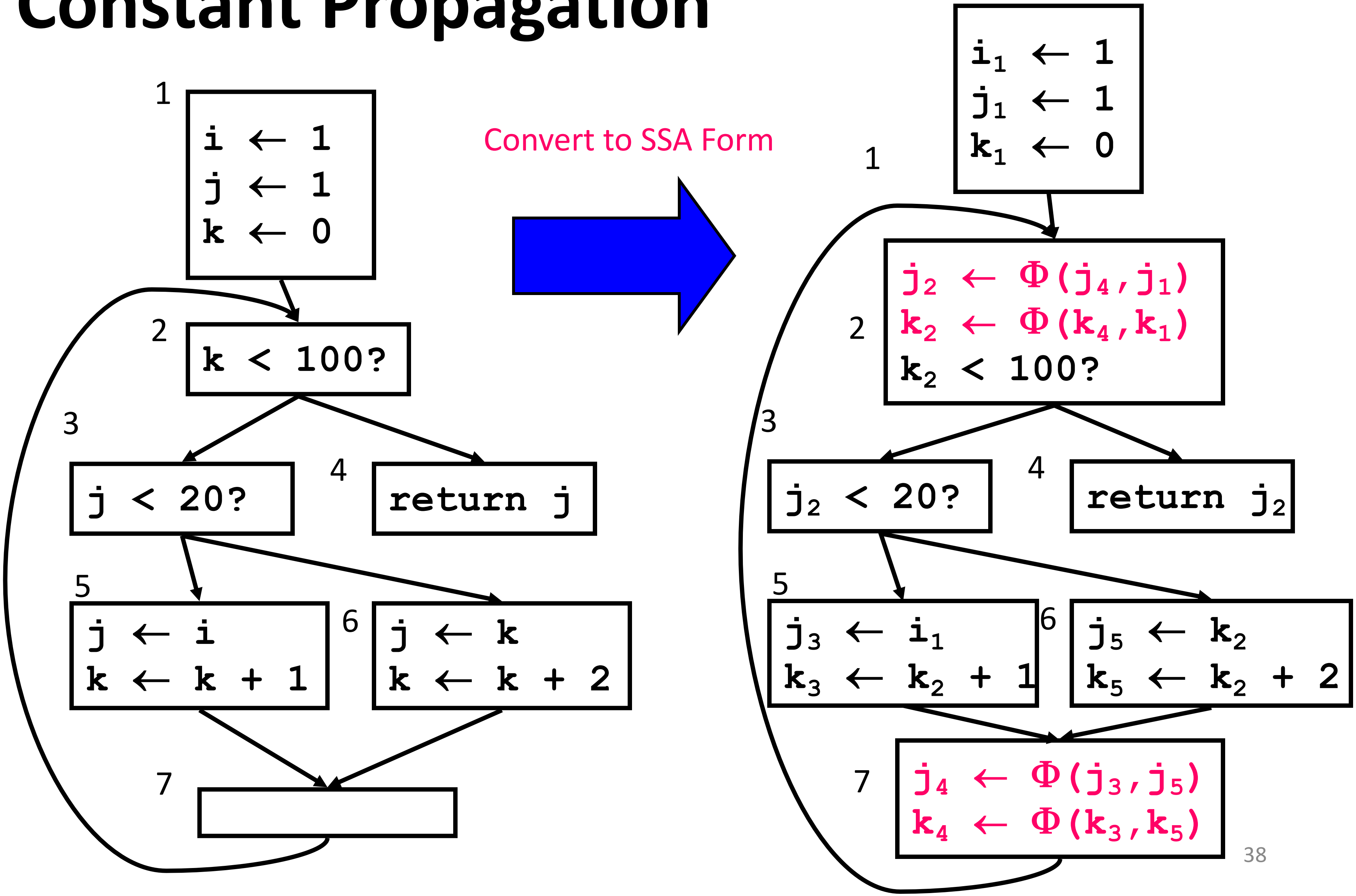
```
W ← list of all defs
while !W.isEmpty {
     Stmt S ← W.removeOne
     if ((S has form "v ← c") ||
         (S has form "v ← Φ(c,…,c)")) then {
         delete S
         foreach stmt U that uses v {
           replace v with c in U
           W.add(U)
         }
     }
}
```

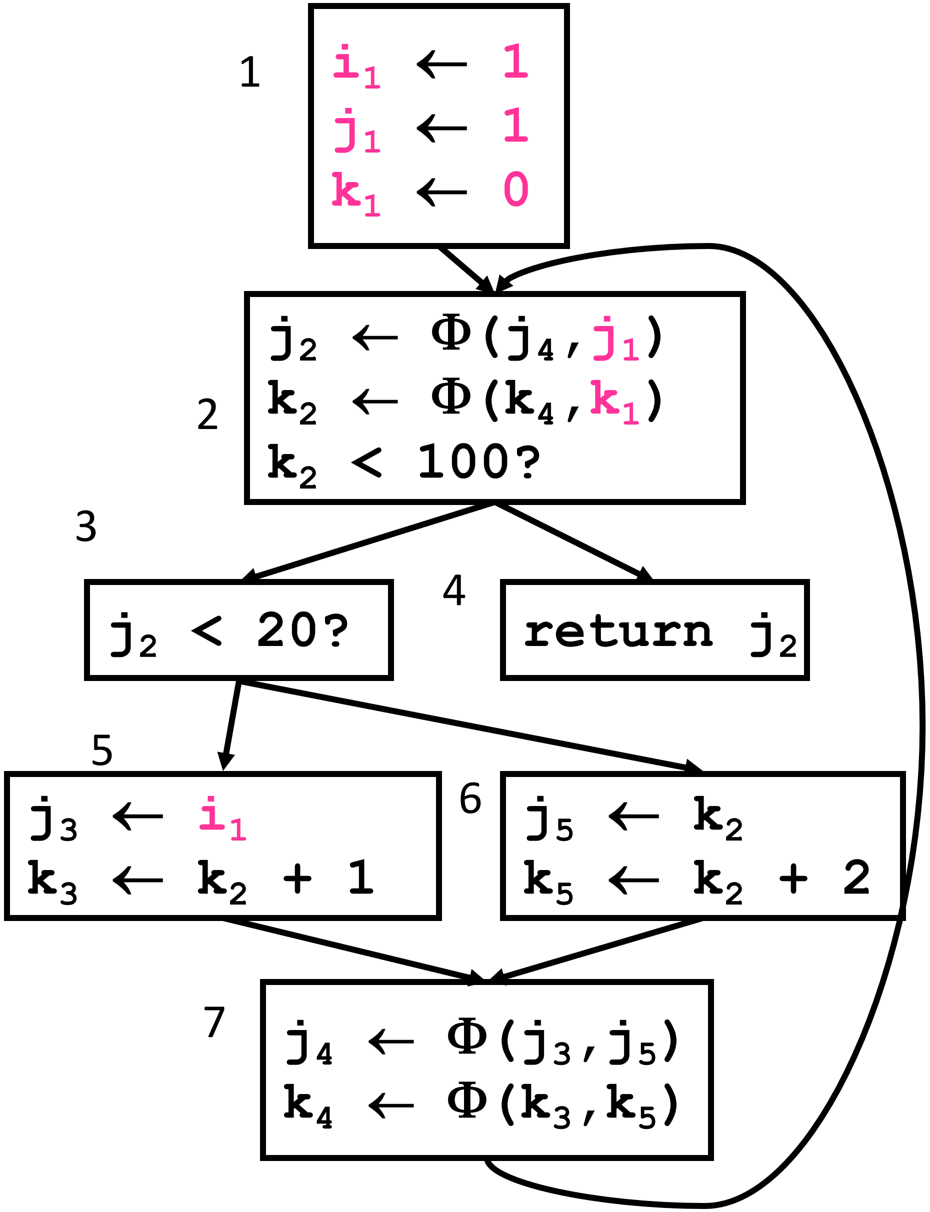
Other Optimizations with SSA

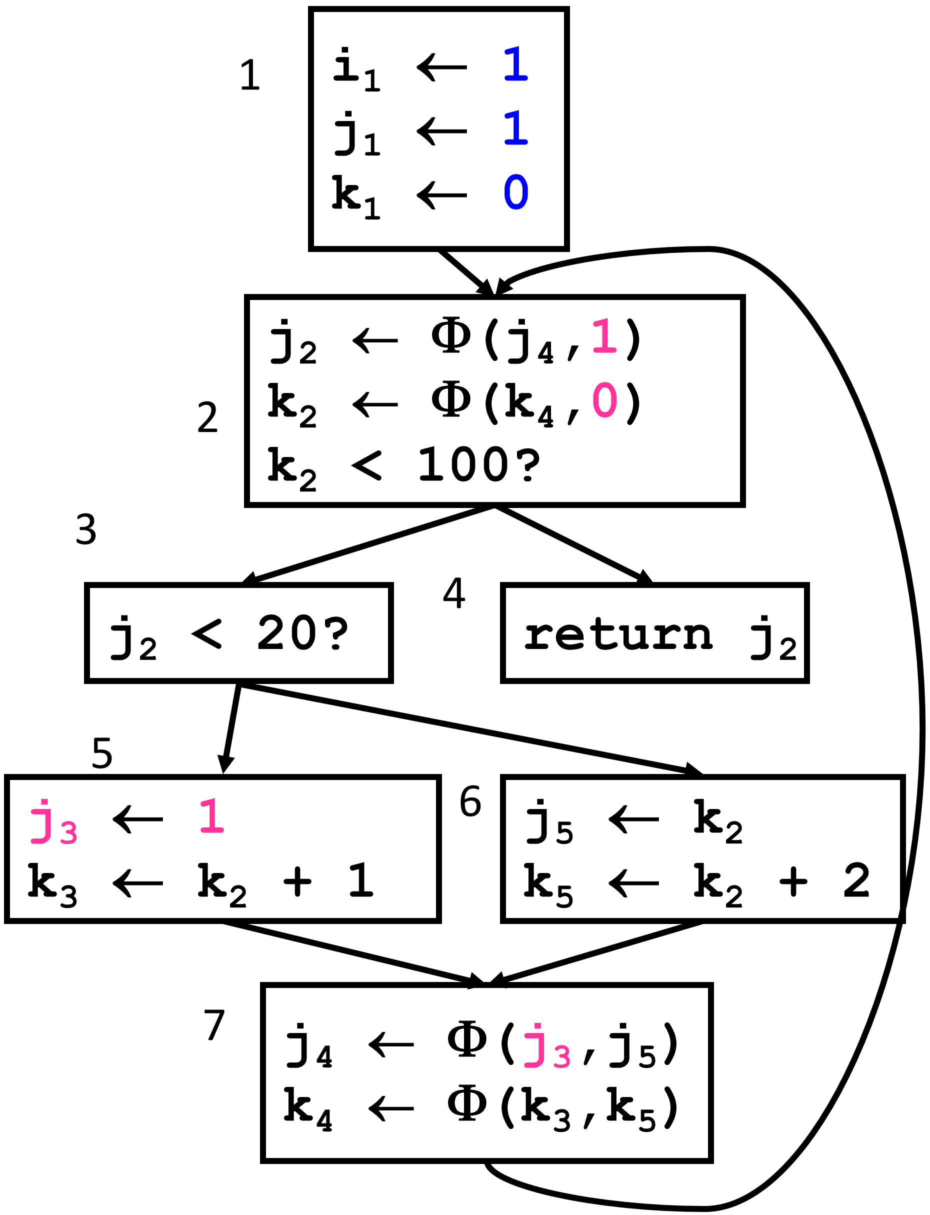
- Copy propogation
  - delete “$x \leftarrow \Phi(y,y,y)$” and replace all x with y
  - delete “$x \leftarrow y$” and replace all x with y

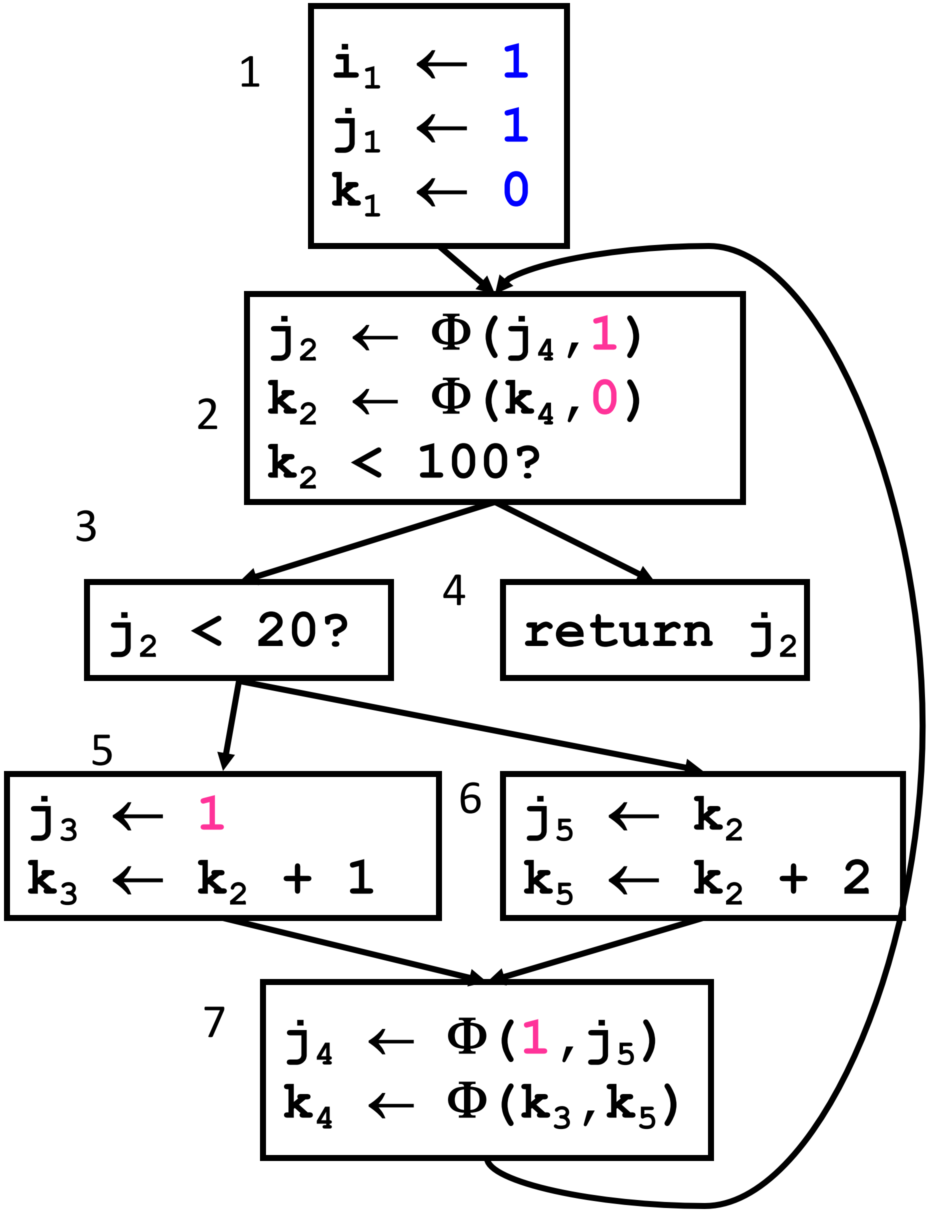
- Constant Folding
  - (Also, constant conditions too!)

Constant Propagation



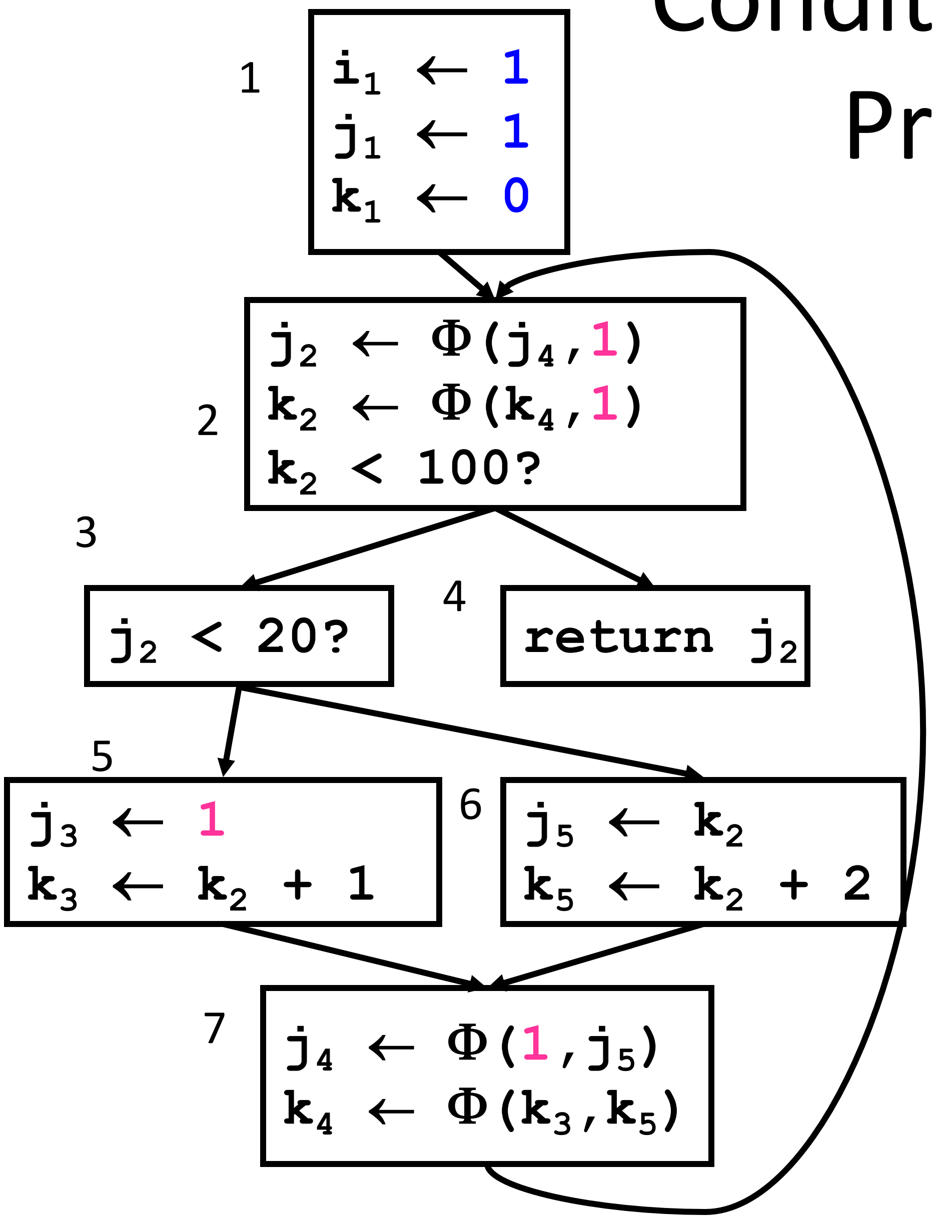

1
i1 ← 1
j1 ← 1
k1 ← 0
2
j2 ← Φ(j4,j1)
k2 ← Φ(k4,k1)
k2 < 100?
3
j2 < 20?
4
return j2
5
j3 ← i1
k3 ← k2 + 1
6
j5 ← k2
k5 ← k2 + 2
7
j4 ← Φ(j3,j5)
k4 ← Φ(k3,k5)


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



Not a very exciting result (yet)…

Conditional Constant Propagation

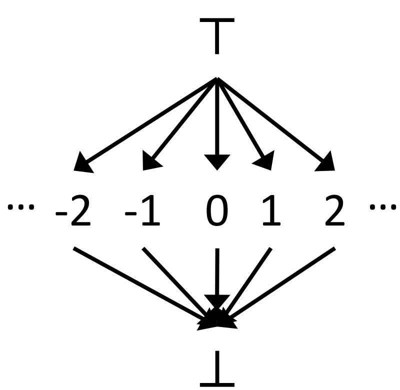


- Does block 6 ever execute?
- Simple Constant Propagation can't tell
- But "Conditional Const. Prop." *can* tell:
  - Assumes blocks don't execute until proven otherwise
  - Assumes values are constants until proven otherwise

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:

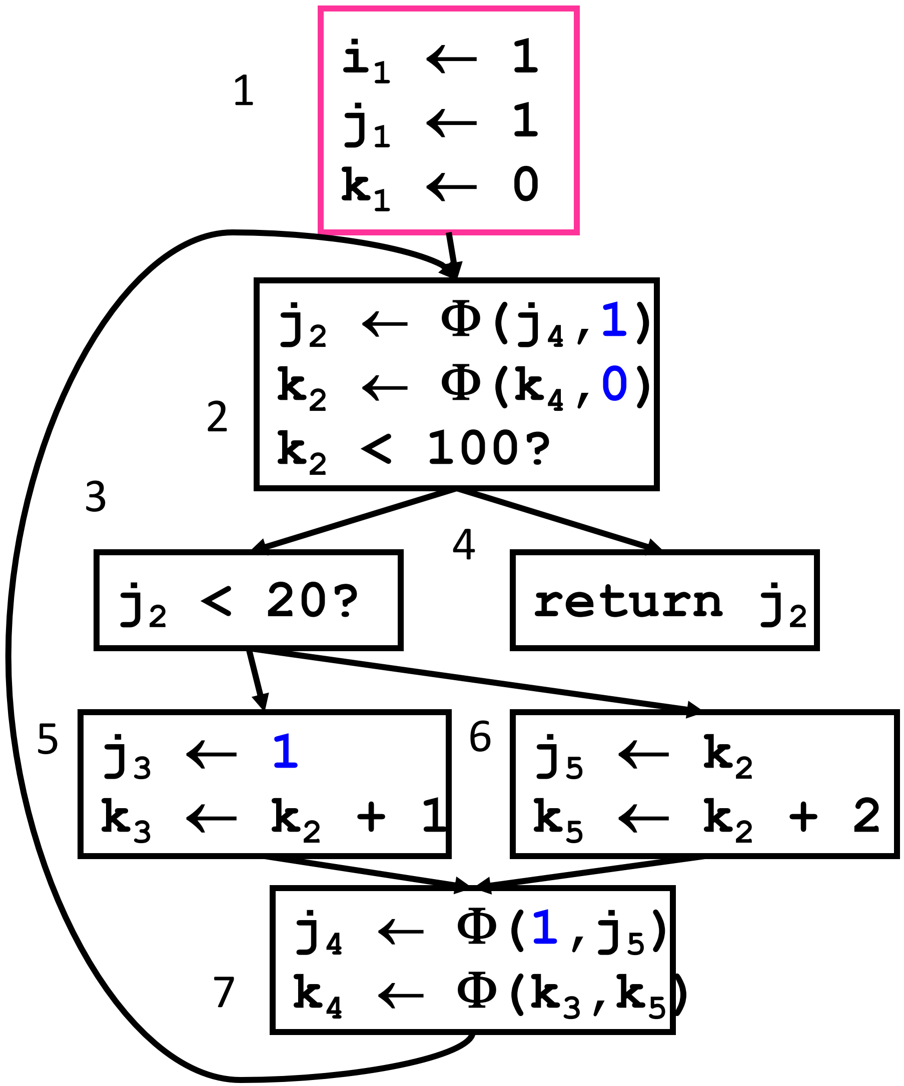


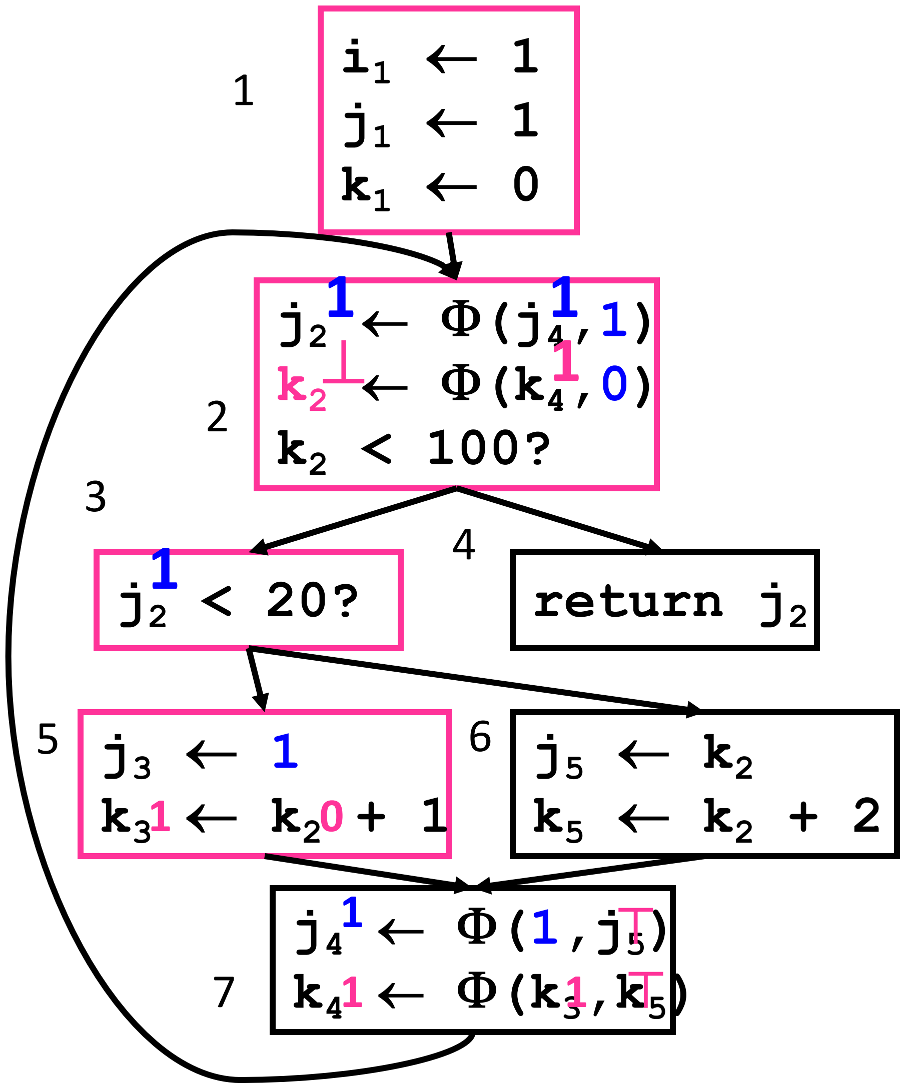
not executed

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




1
i1 ← 1
j1 ← 1
k1 ← 0
2
j2 ← Φ(j4,1)
k2 ← Φ(k4,0)
k2 < 100?
3
j2 < 20?
4
return j2
5
j3 ← 1
k3 ← k2 + 1
6
j5 ← k2
k5 ← k2 + 2
7
j4 ← Φ(1,j5)
k4 ← Φ(k3,k5)

Conditional Constant Propagation