SSA

15-411/15-611 Compiler Design

Seth Copen Goldstein

September 15, 2020

Today

- Trivial SSA
- ϕ-functions
- Motivation (CCP)
- Dominators
- Placement & Renaming
- Deconstructing SSA

SSA

- Static single assignment is an **IR** where every variable has only ONE definition in the program text
  - single **static** definition
  - (Could be in a loop which is executed dynamically many times.)
- ϕ-functions used at CFG merge points
- Definitions dominate uses

Advantages of SSA

- Makes du-chains explicit
- Makes dataflow optimizations
  - Easier
  - faster
- 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

One definition for each use

- Key to SSA is single point of definition for each use
- Introduce ϕ-functions to handle joins in CFG

```
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:
```

One definition for each use

- Key to SSA is single point of definition for each use
- Introduce ϕ-functions to handle joins in CFG

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

One definition for each use

- Key to SSA is single point of definition for each use
- Introduce ϕ-functions to handle joins in CFG

$$
\begin{array}{l}
\quad \mathbf{x}_0 \leftarrow \ldots \\
\quad \mathbf{y}_0 \leftarrow \ldots \\
\quad \mathbf{if}\ (\mathbf{x}_0 >= 100)\ \mathbf{goto\ end} \\
\mathbf{loop:} \\
\quad \mathbf{x}_1 \leftarrow \Phi(\mathbf{x}_0, \mathbf{x}_2) \\
\quad \mathbf{y}_1 \leftarrow \Phi(\mathbf{y}_0, \mathbf{y}_2) \\
\quad \mathbf{x}_2 \leftarrow \mathbf{x}_1 + 1 \\
\quad \mathbf{y}_2 \leftarrow \mathbf{y}_1 + 1 \\
\quad \mathbf{if}\ (\mathbf{x}_2 < 100)\ \mathbf{goto\ loop} \\
\mathbf{end:} \\
\quad \mathbf{x}_3 \leftarrow \Phi(\mathbf{x}_0, \mathbf{x}_2) \\
\quad \mathbf{y}_3 \leftarrow \Phi(\mathbf{y}_1, \mathbf{y}_2)
\end{array}
$$

The Φ function

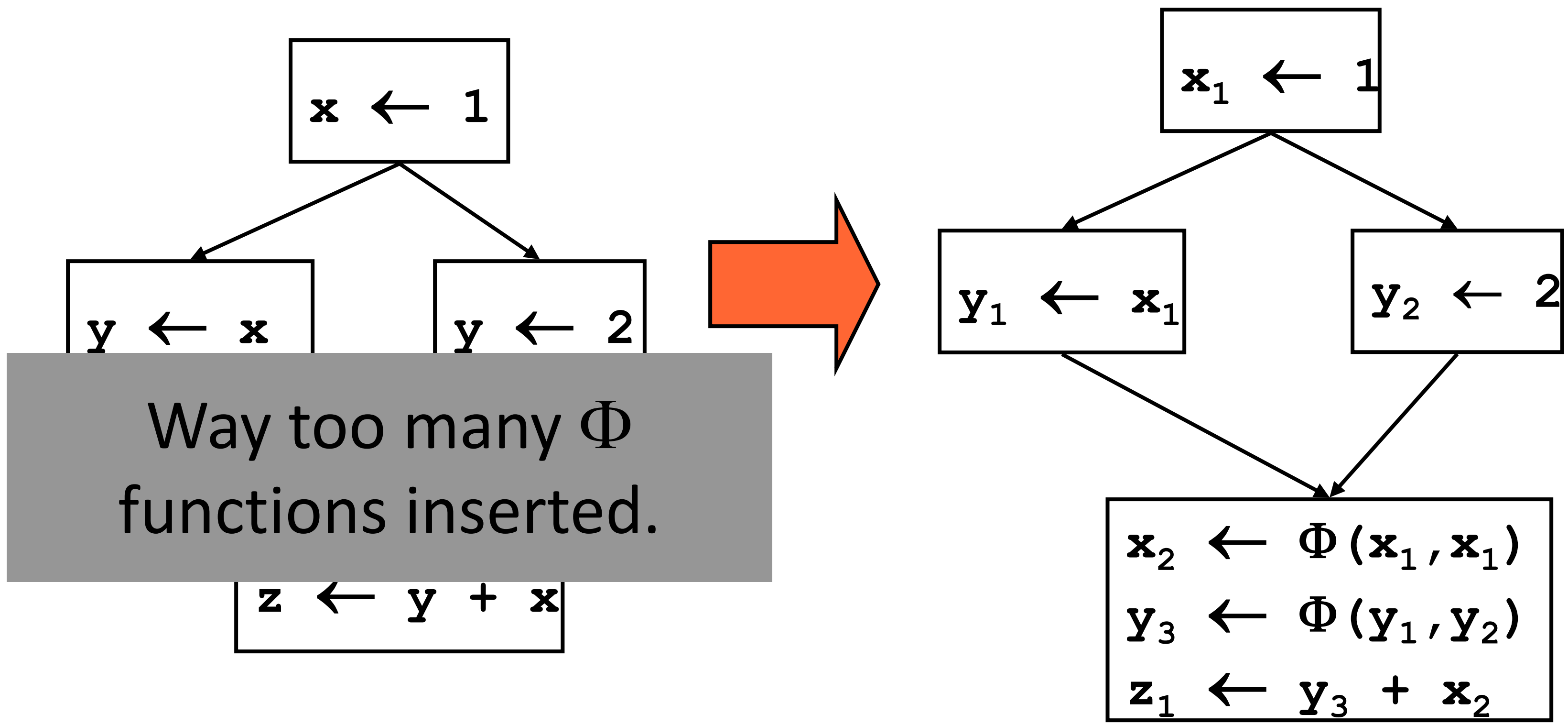
- Φ merges multiple definitions along multiple control paths into a single definition.
- At a BB with p predecessors, there are p arguments to the Φ function.

  $x_{new} \leftarrow \Phi(\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3, \ldots, \mathbf{x}_p)$

- How do we choose which x_i to use?
  - We don't really care!
  - If we care, use moves on each incoming edge

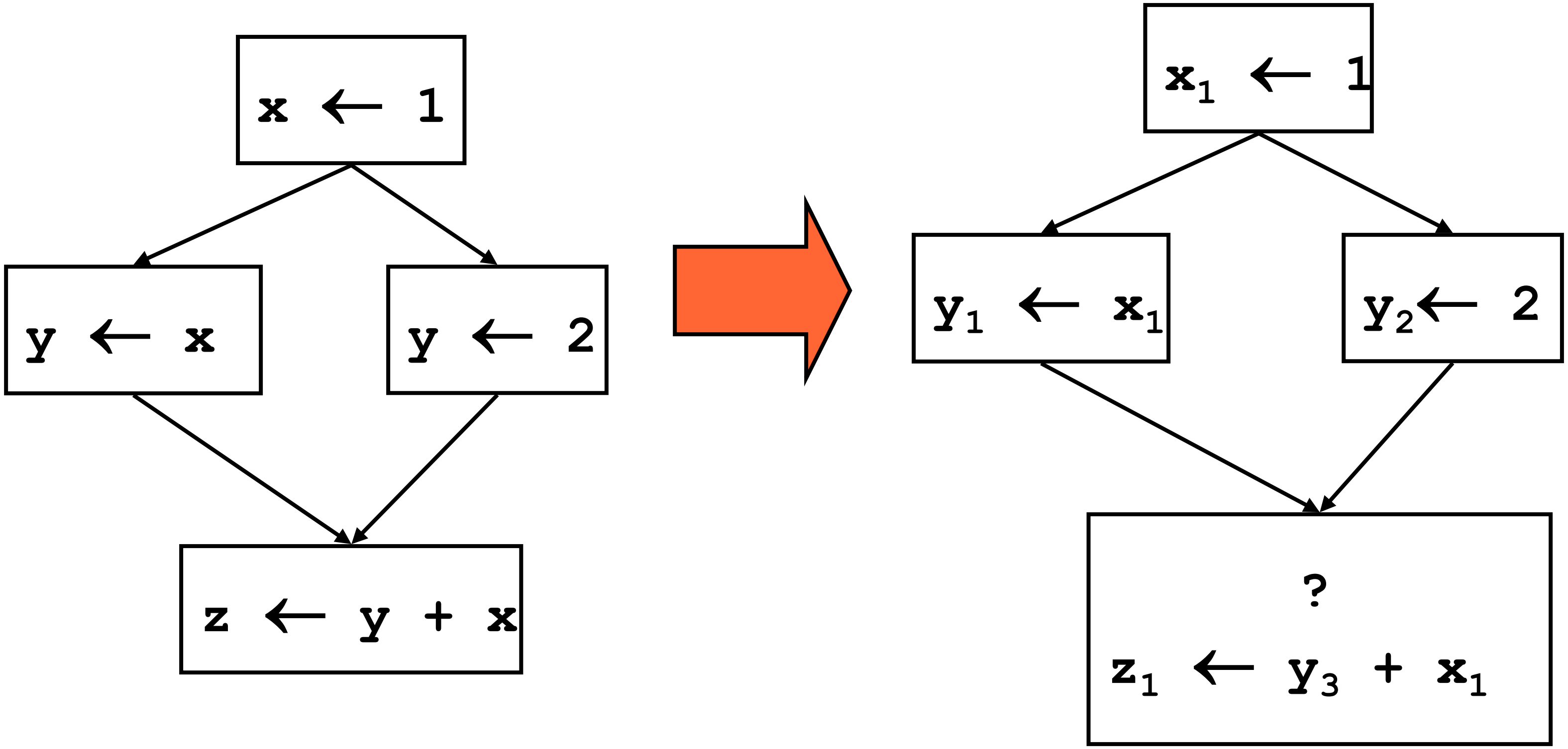
Trivial SSA

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



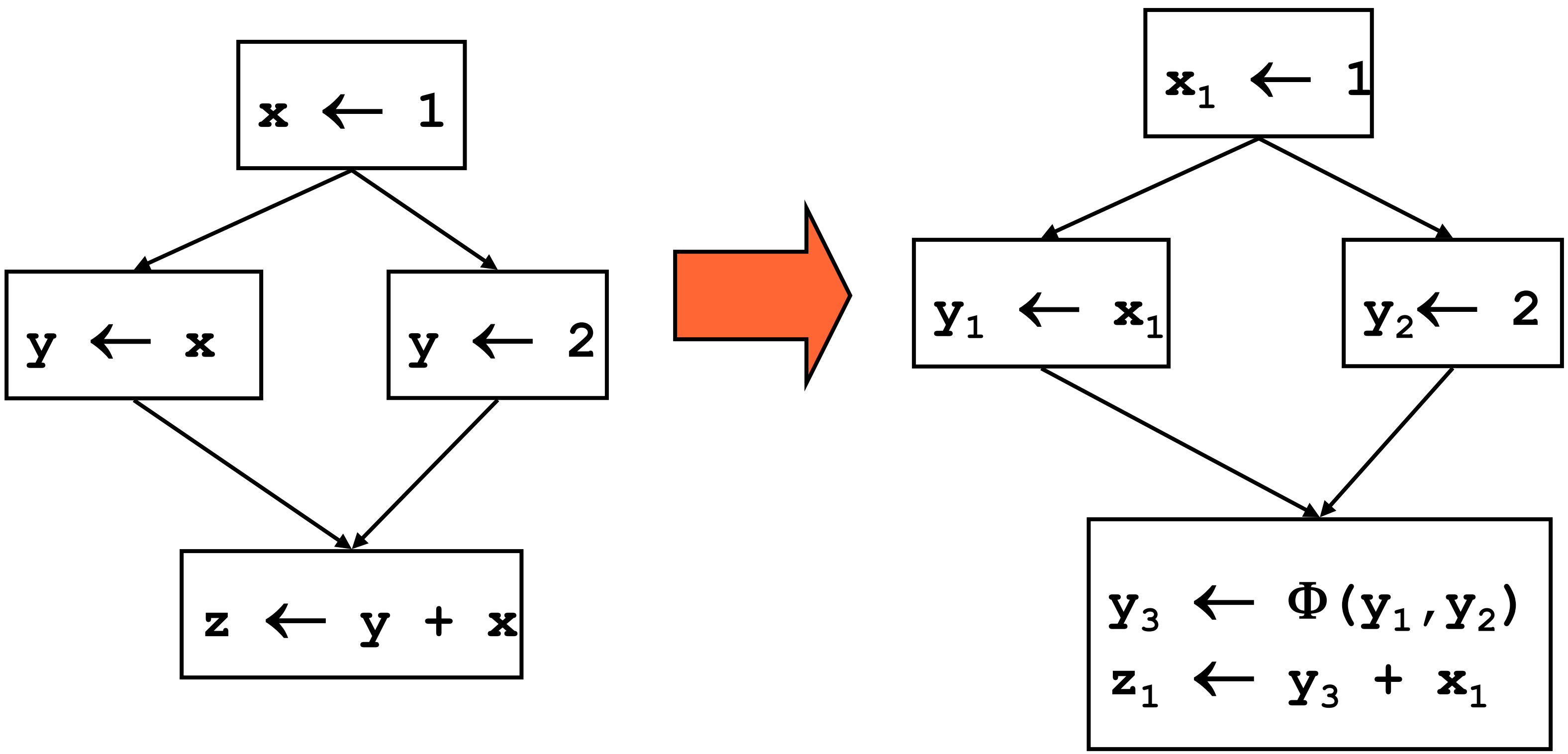
Minimal SSA

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

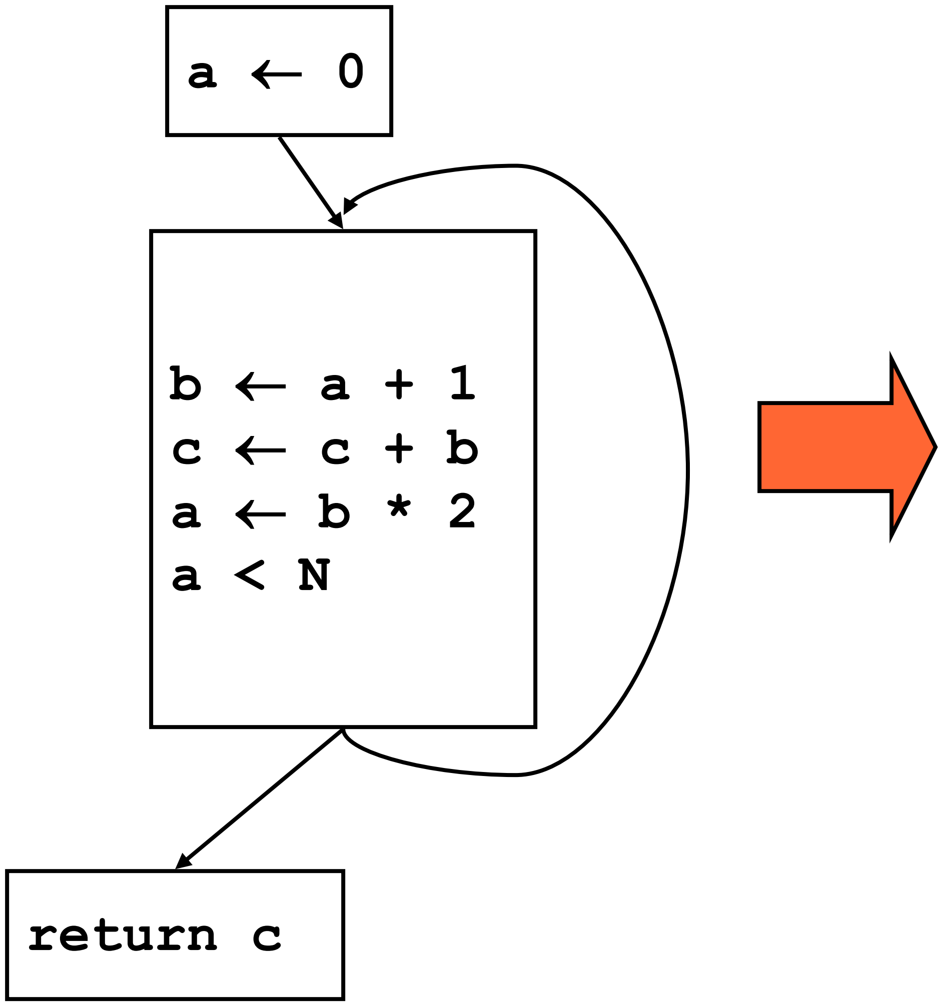


Minimal SSA

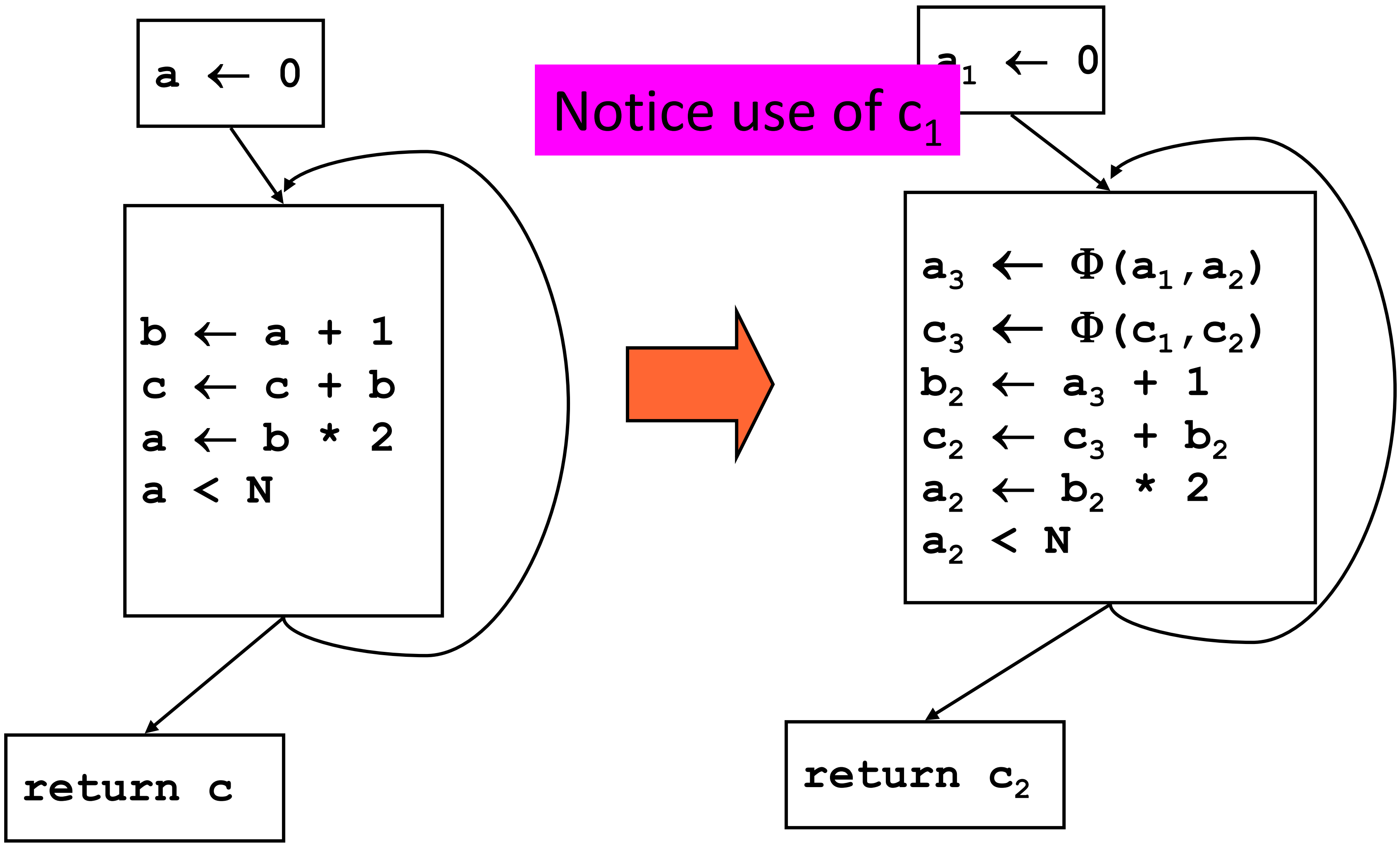
- Each assignment generates a fresh variable.
- At each join point insert Φ functions for all variables with multiple outstanding defs.



Another Example

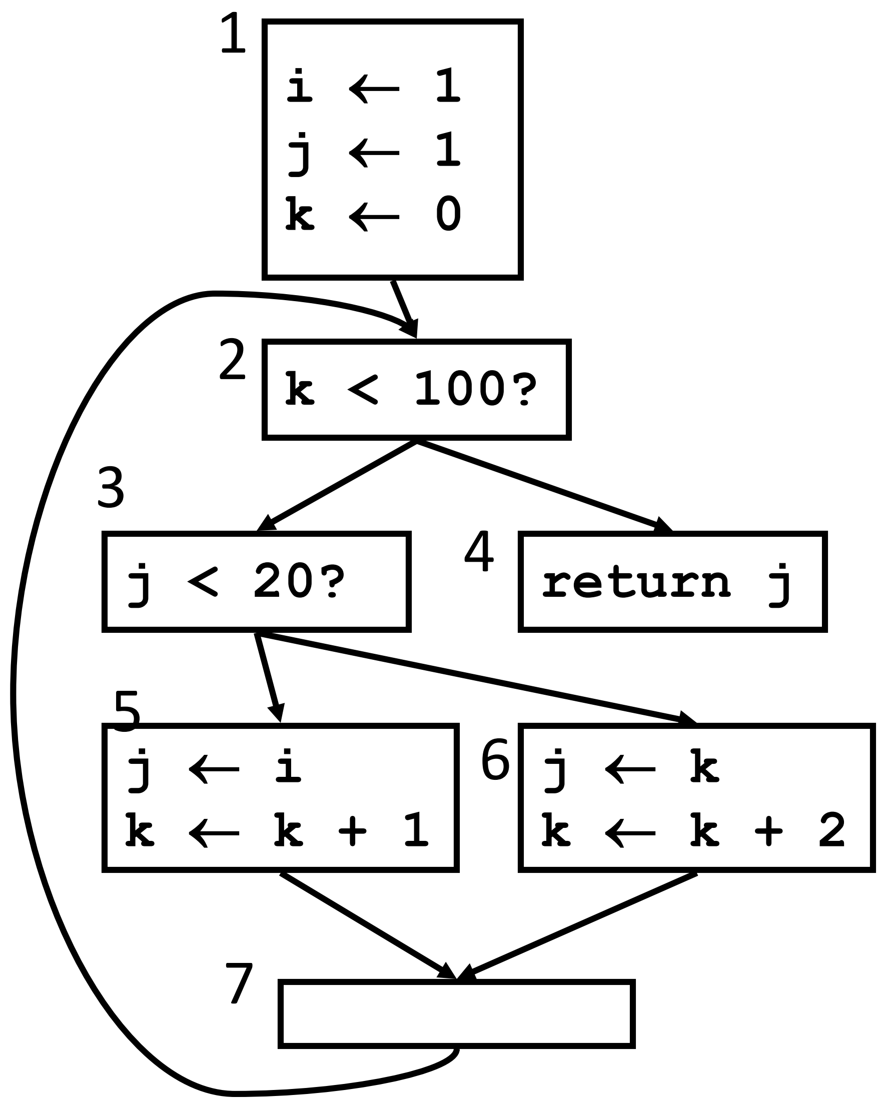


Another Example

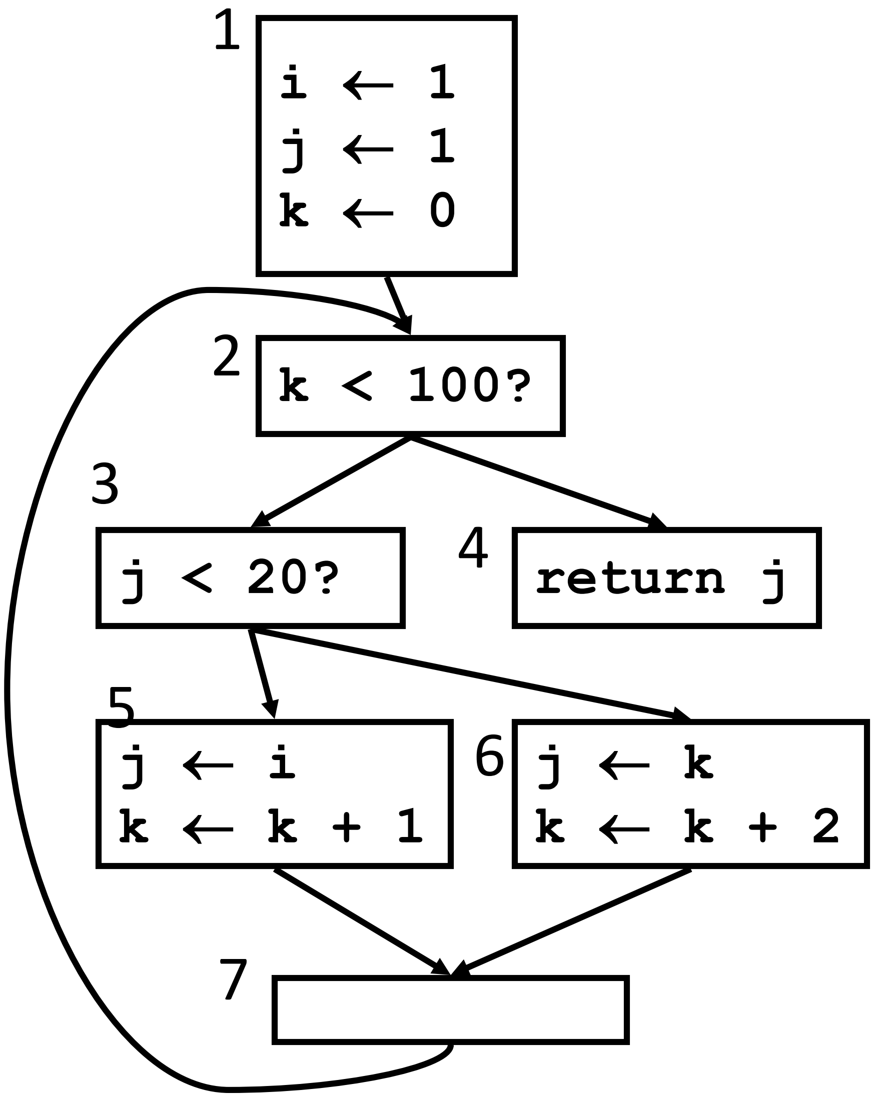


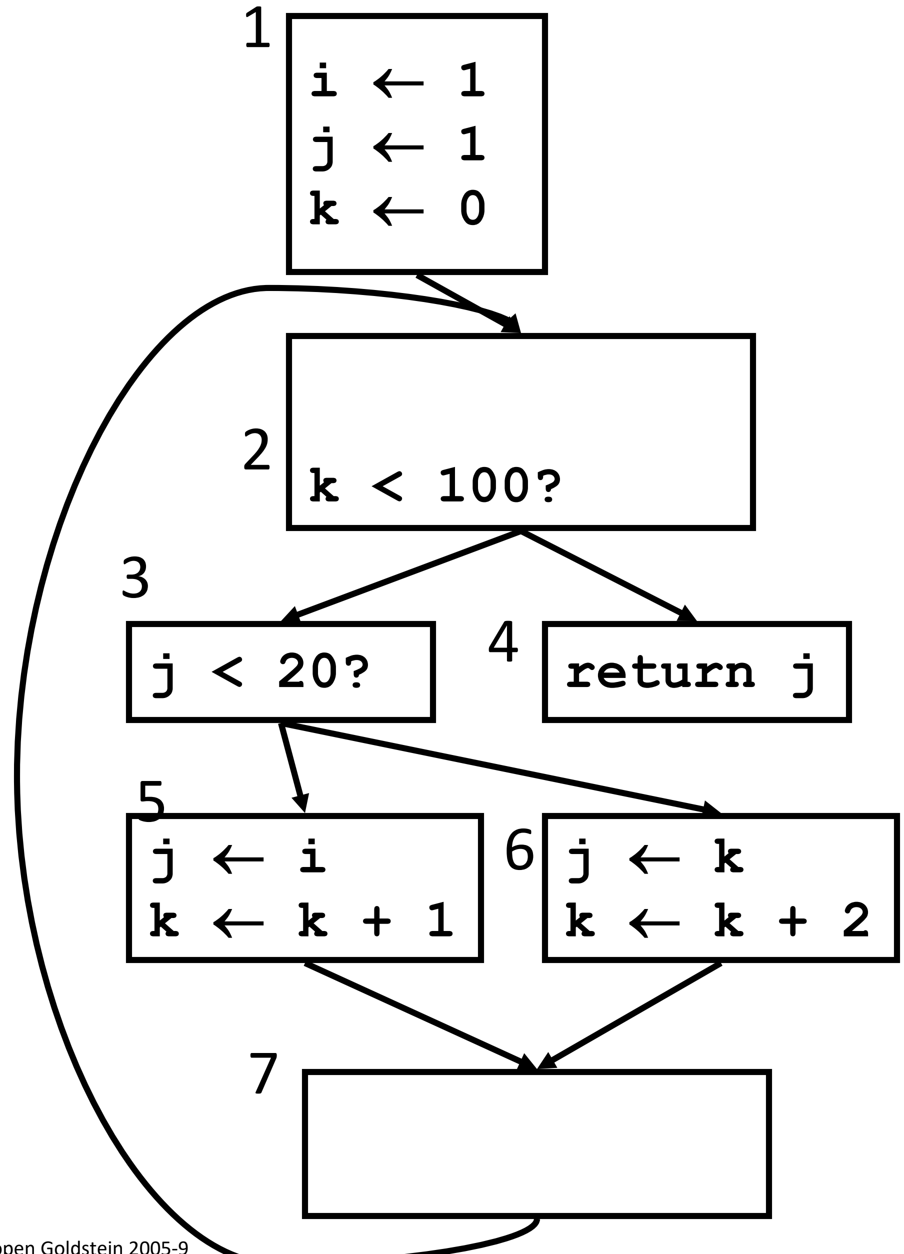
Lets optimize the following:

```
i=1;
j=1;
k=0;
while (k<100) {
  if (j<20) {
      j=i;
      k++;
  } else {
      j=k;
      k+=2;
  }
}
return j;
```

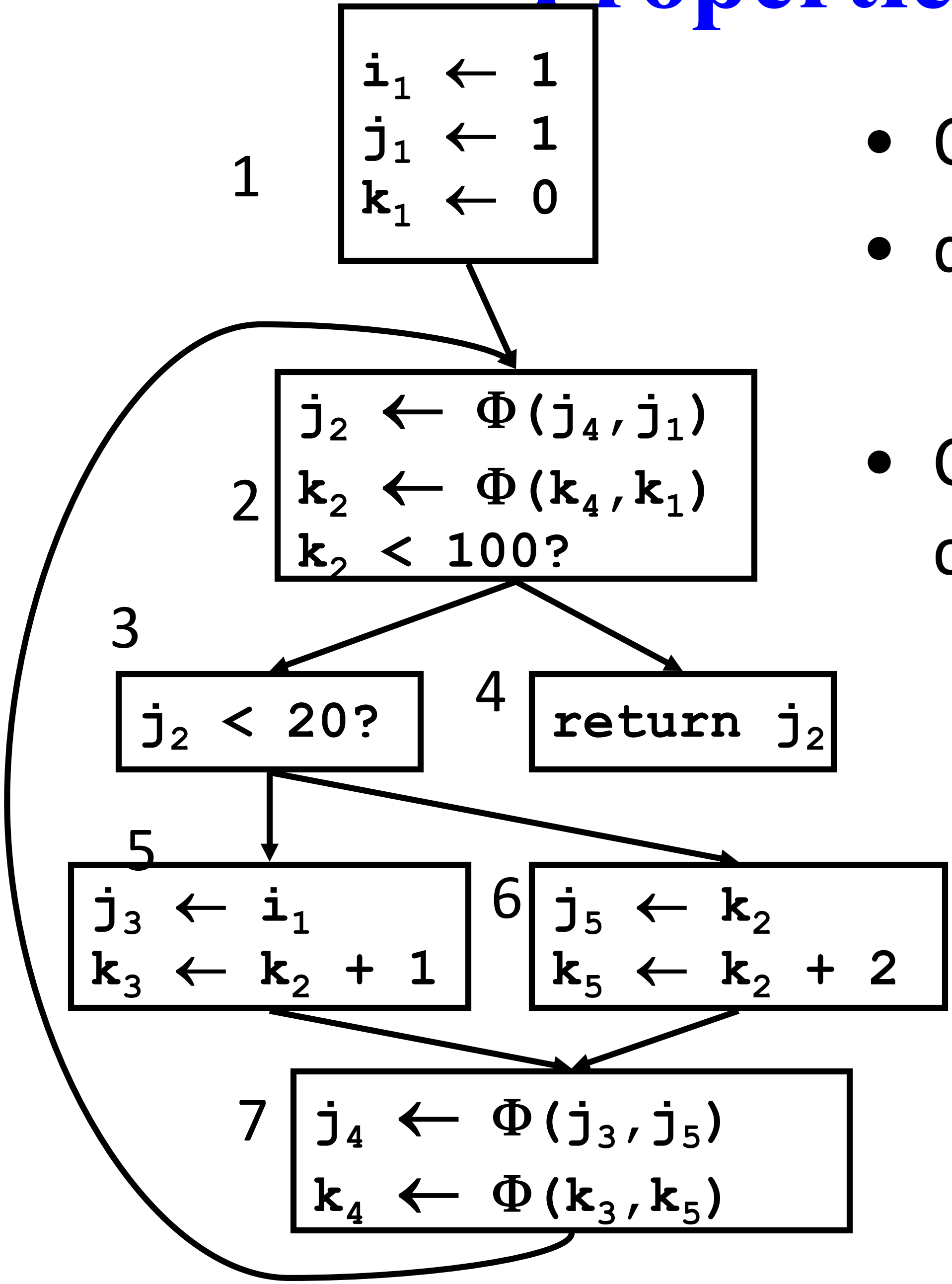


First, turn into SSA





Properties of SSA



- Only 1 assignment per variable
- definitions dominate uses

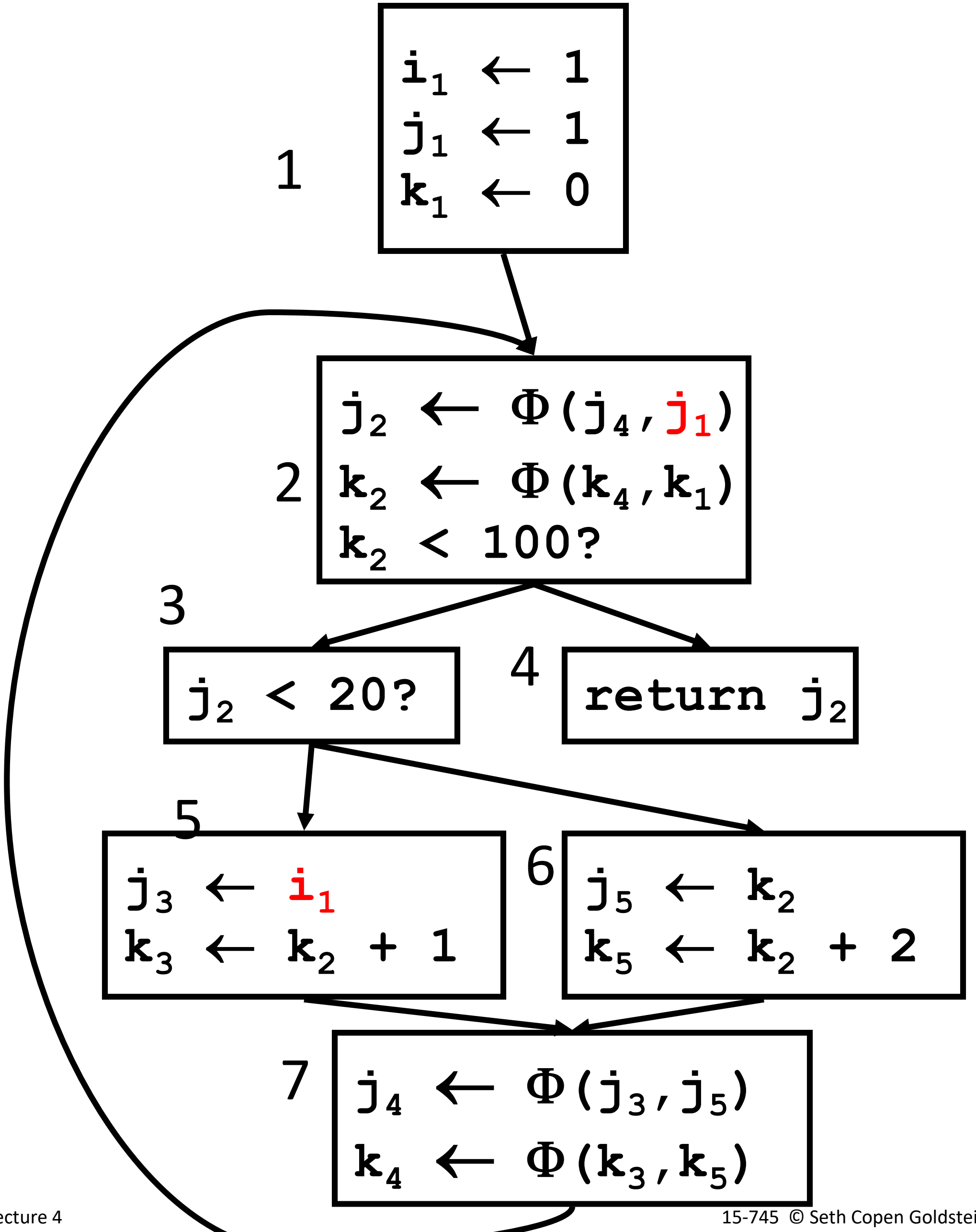
- Can we use this to help with constant propagation?

Constant Propagation

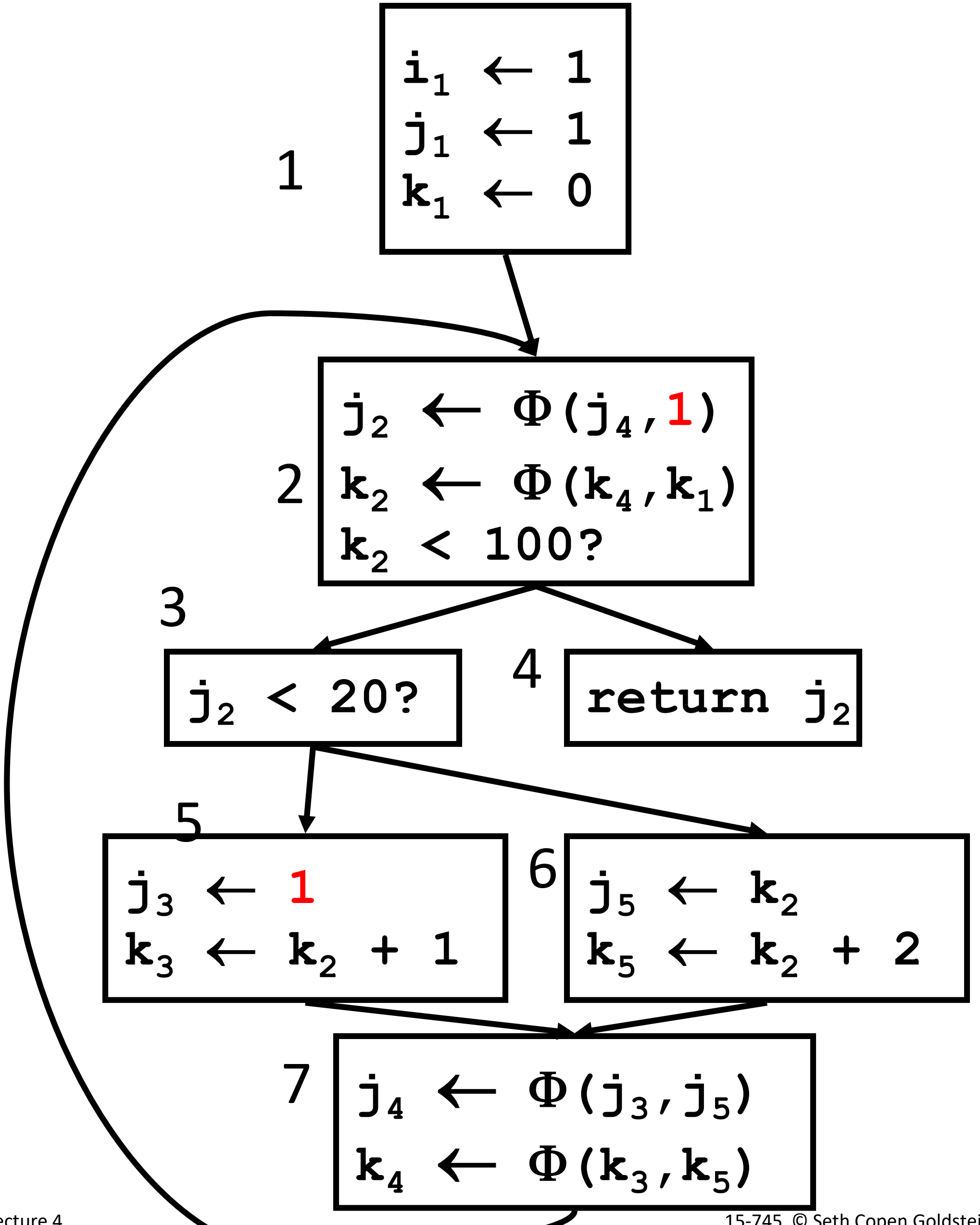
- If "v ← c", replace all uses of v with c
- If "v ← `Φ(c,c,c)`" replace all uses of v with c

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

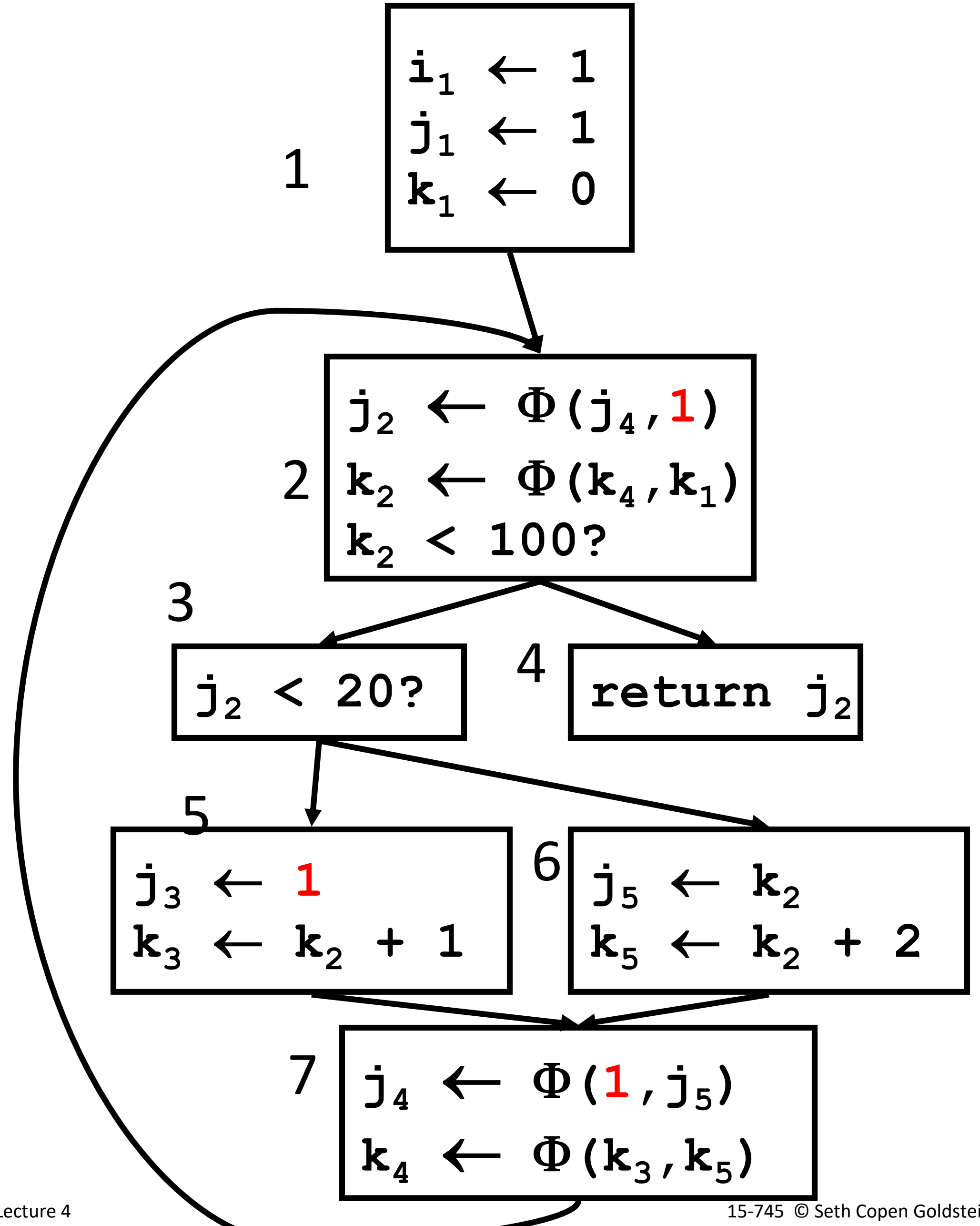
Constant Propagation



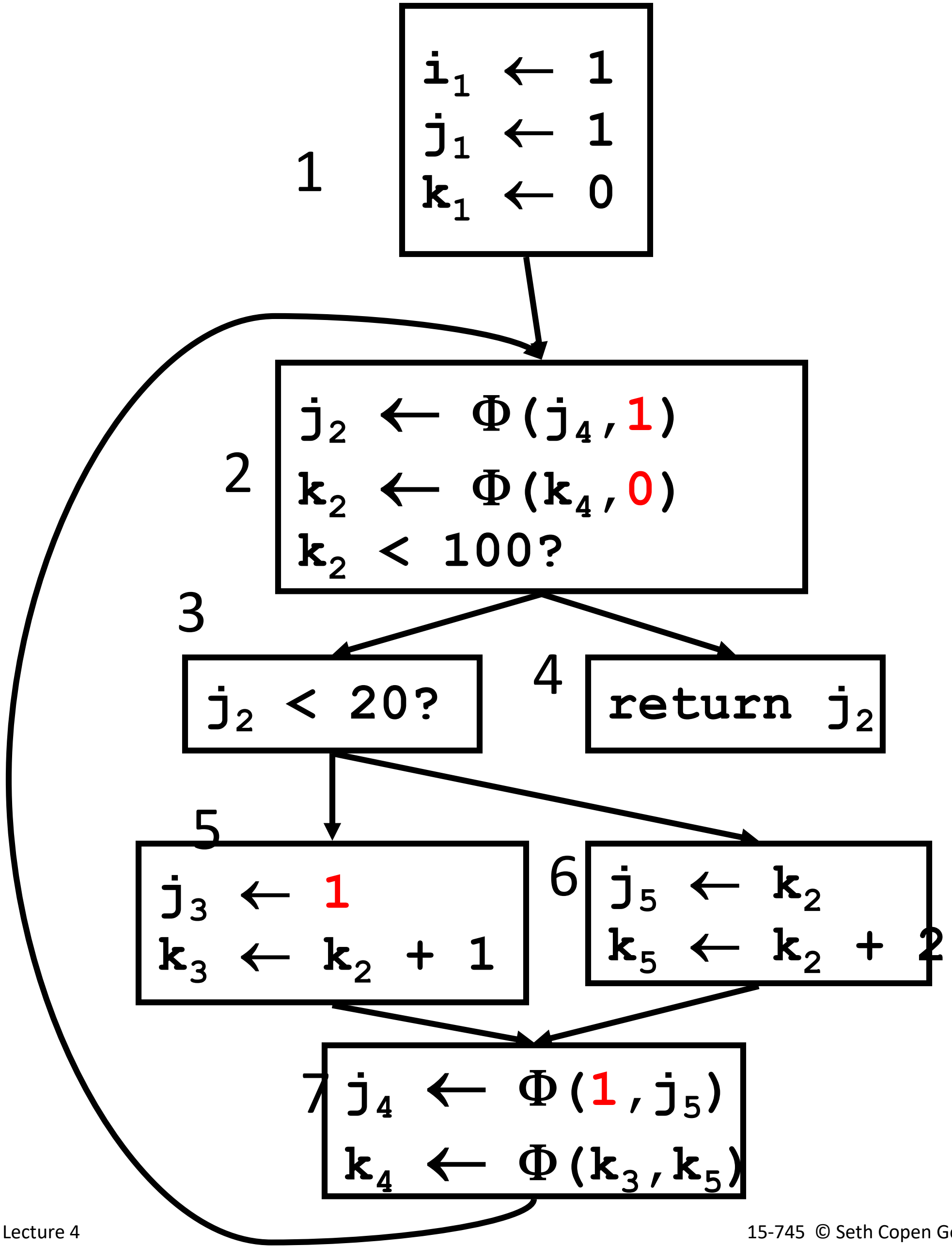
Constant Propagation



Constant Propagation



Constant Propagation

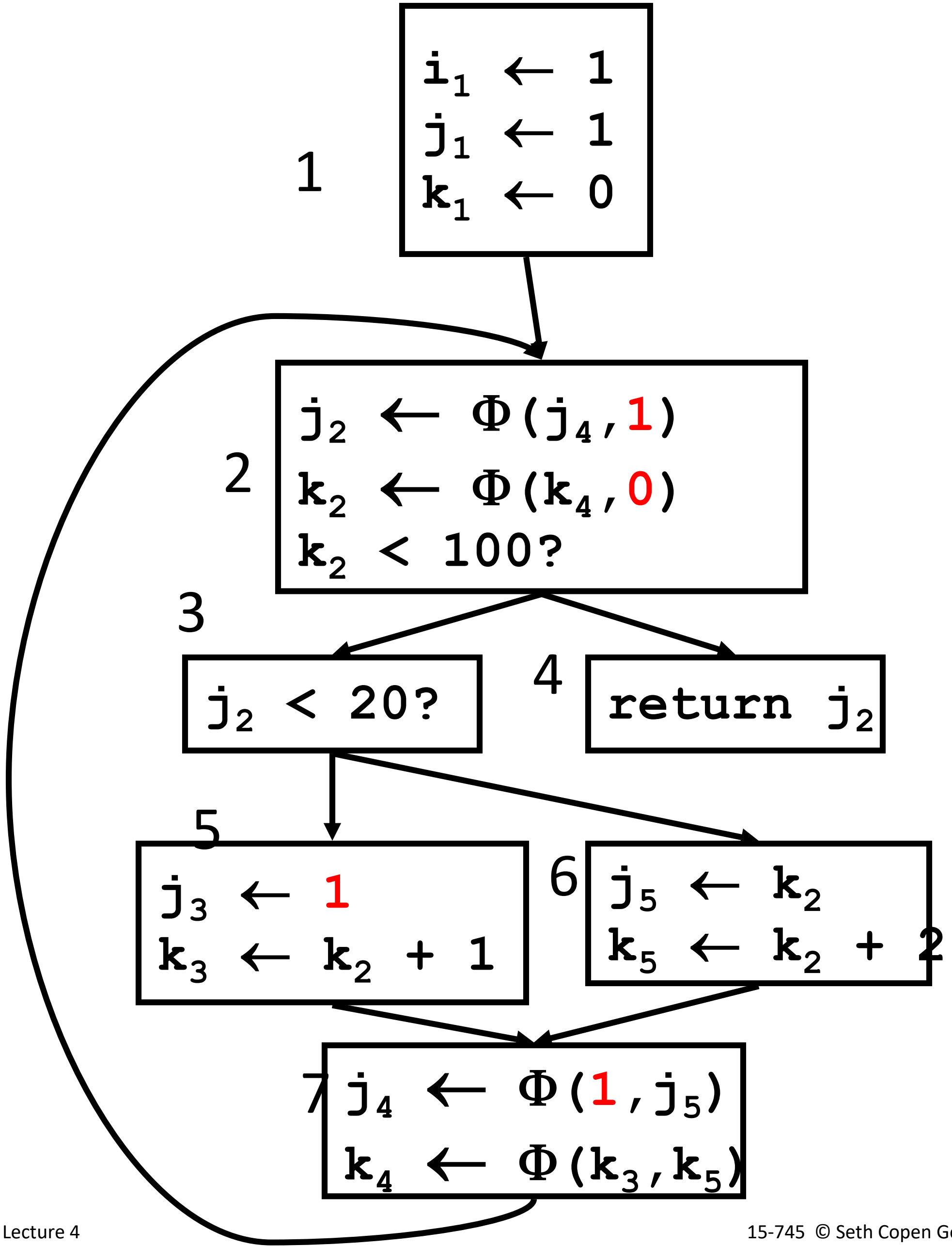


But, so what?

Other stuff we can do?

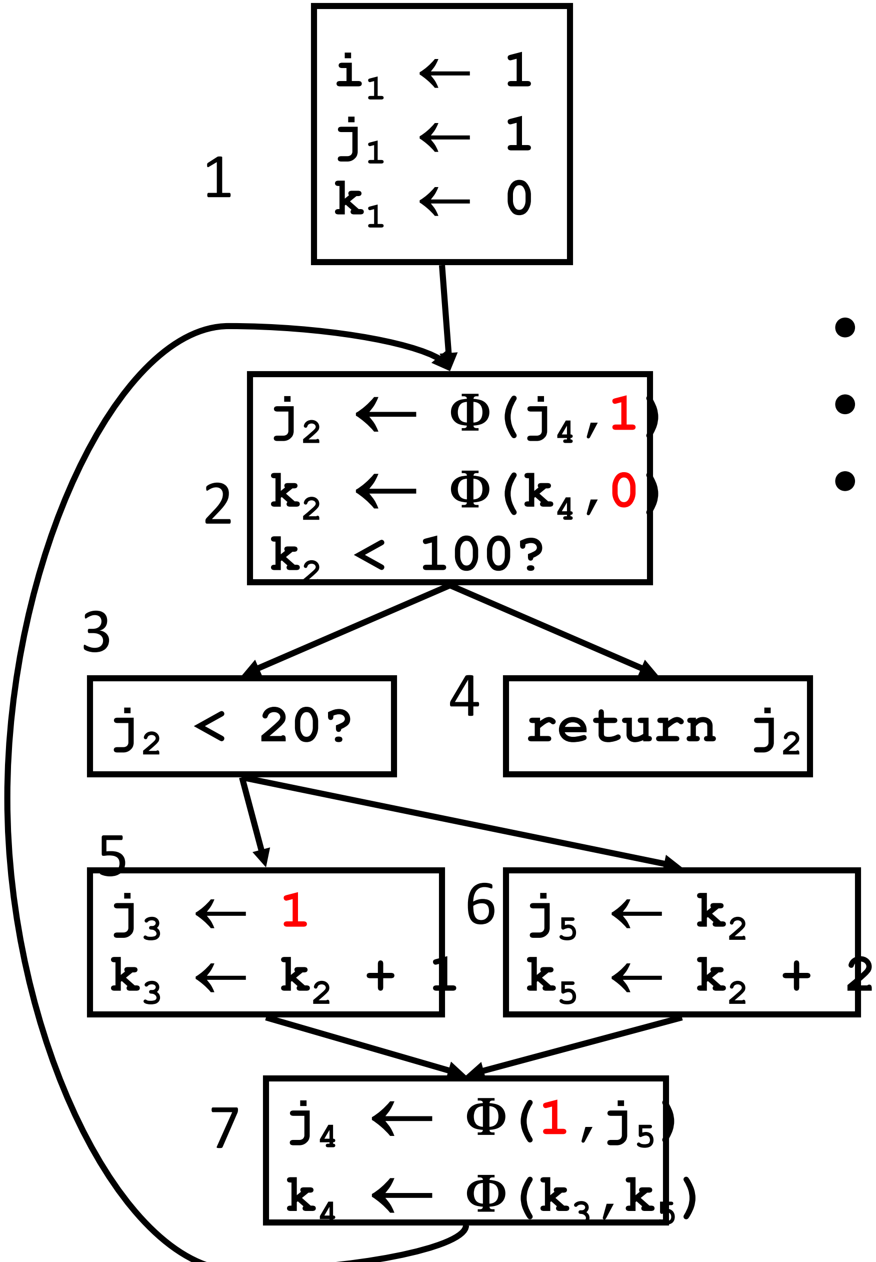
- Copy propagation
  delete “$x \leftarrow \Phi(y)$” and replace all x with y
  delete “$x \leftarrow y$” and replace all x with y
- Constant Folding
  (Also, constant conditions too!)
- Unreachable Code
  Remember to delete all edges from unreachable block

Constant Propagation



But, so what?

Conditional Constant Propagation



- Does block 6 ever execute?
- Simple CP can't tell
- CCP can tell:
  - Assumes blocks don't execute until proven otherwise
  - Assumes Values are constants until proven otherwise

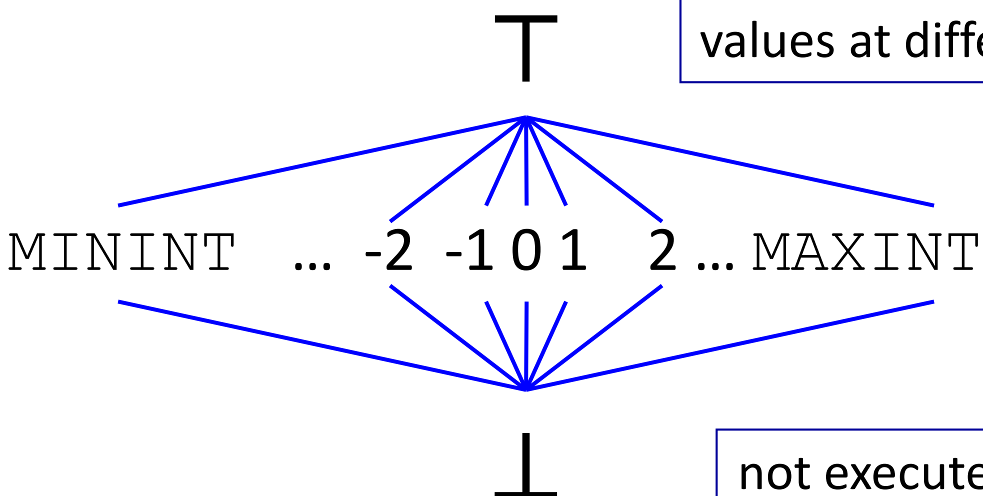
CCP data structures & lattice

Keep 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)

Use a lattice for variables:

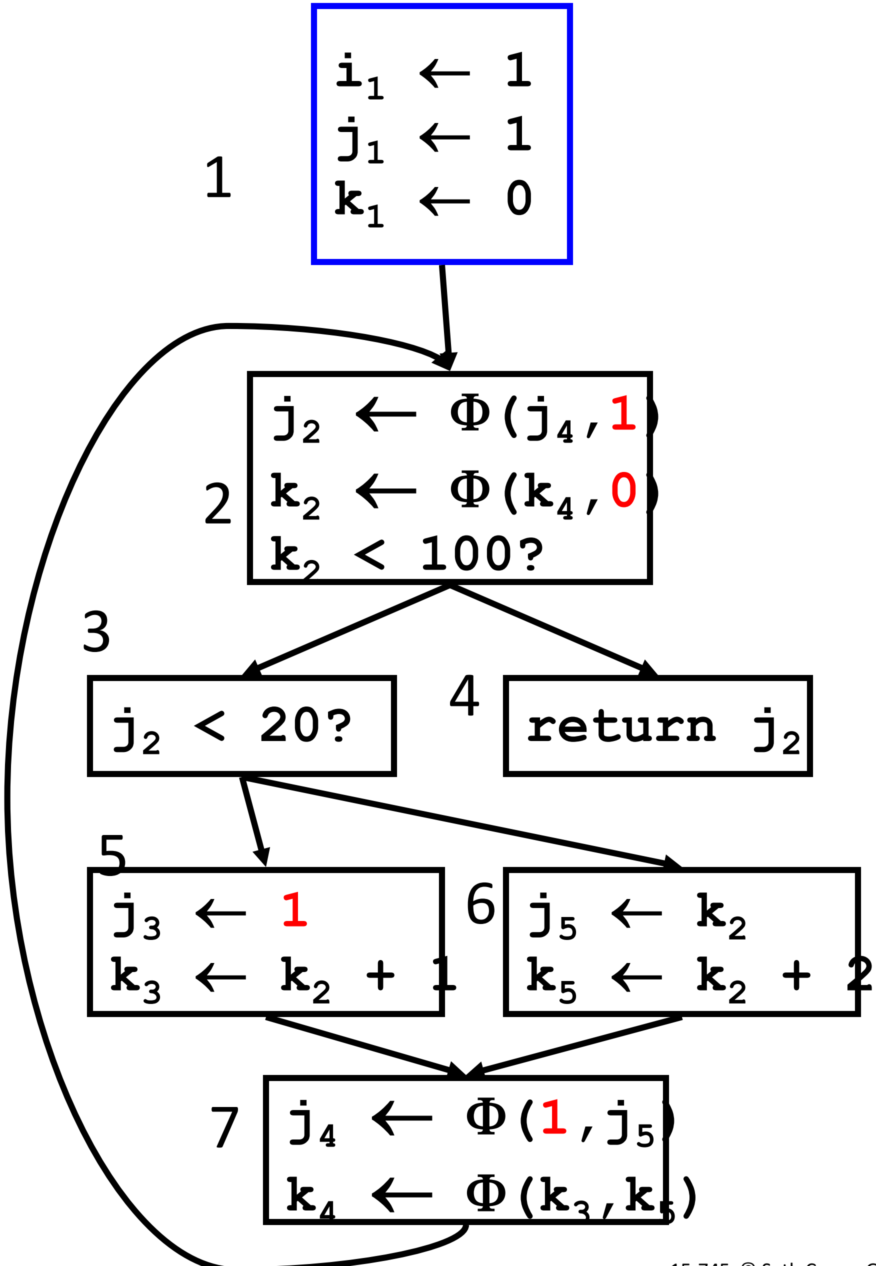


have evidence that variable can hold different values at different times

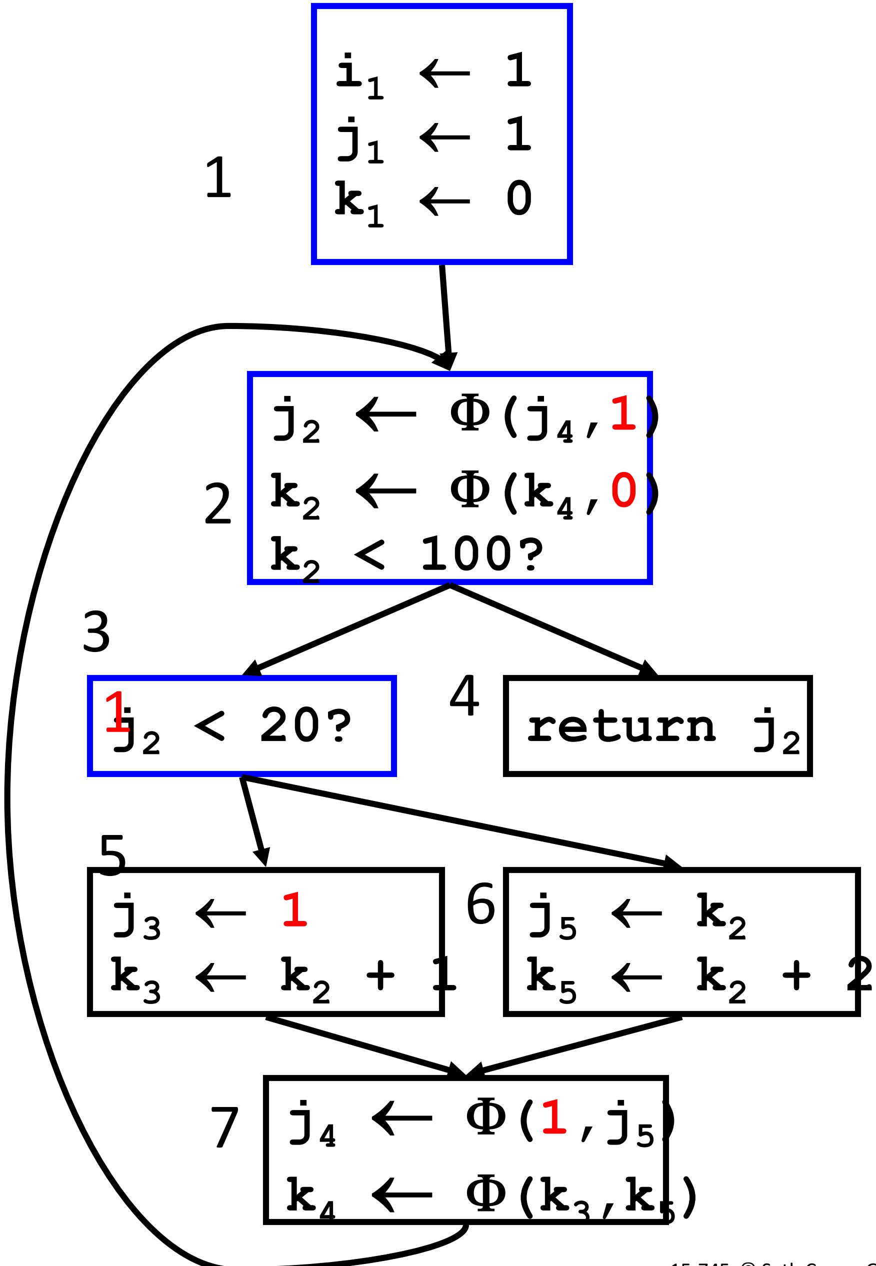
evidence that the var has been assigned a constant

not executed

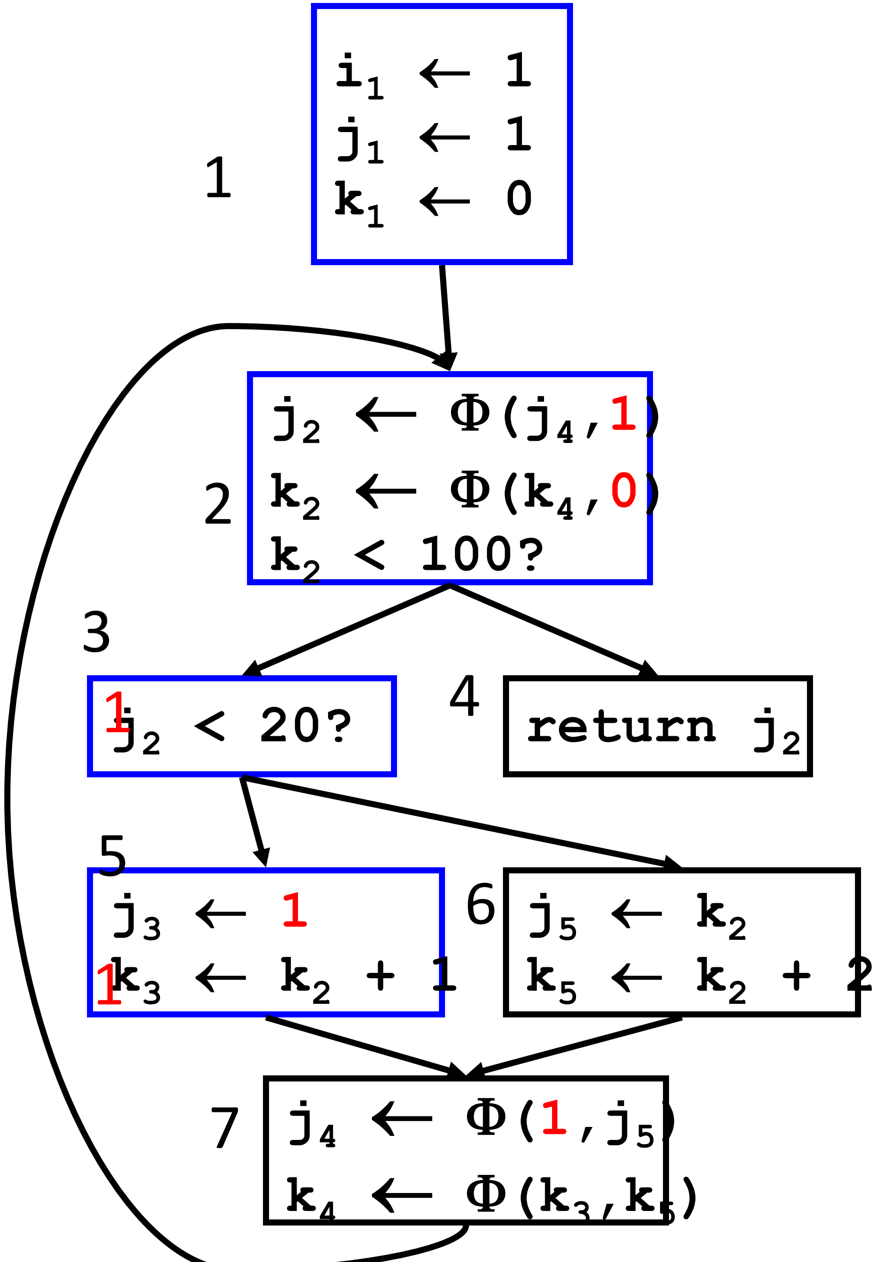
Conditional Constant Propagation



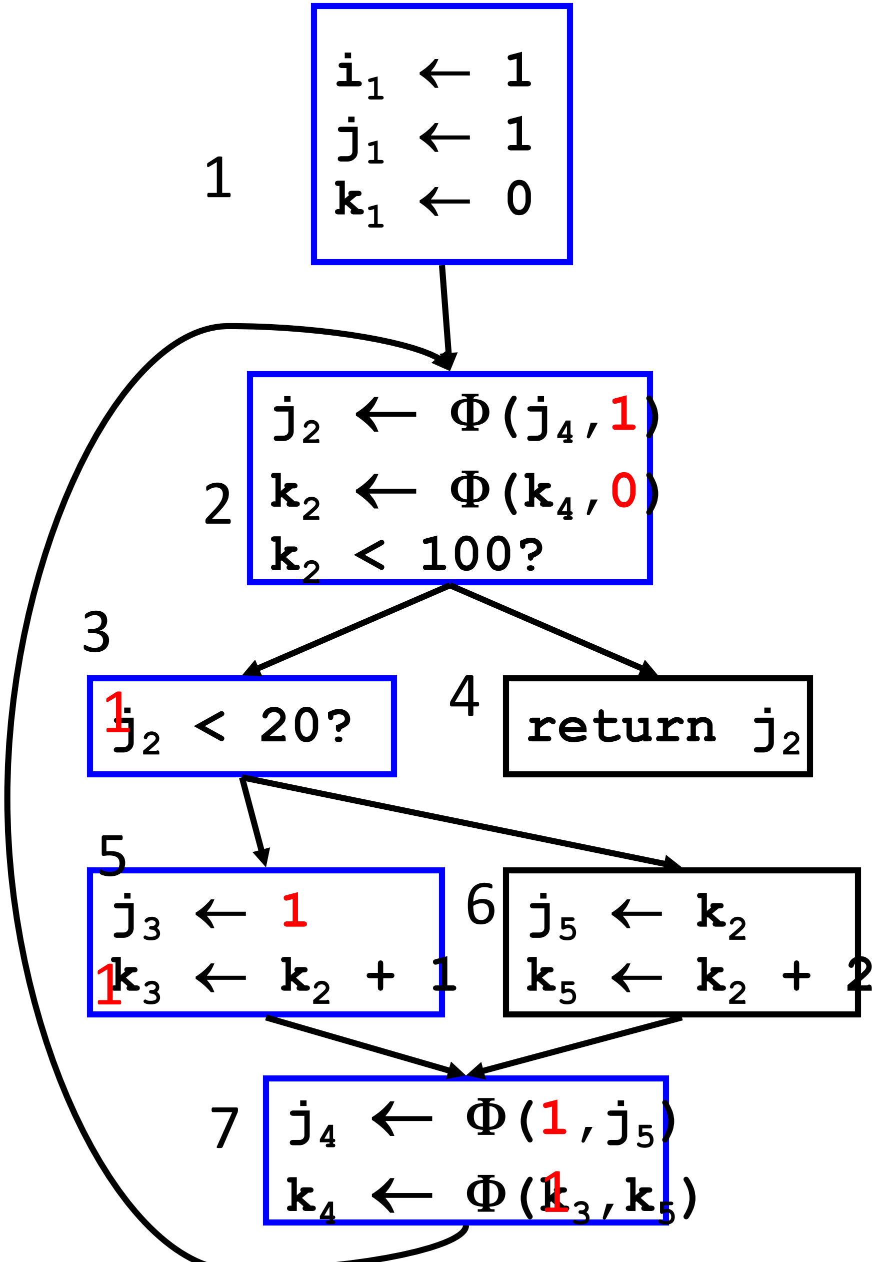
Conditional Constant Propagation



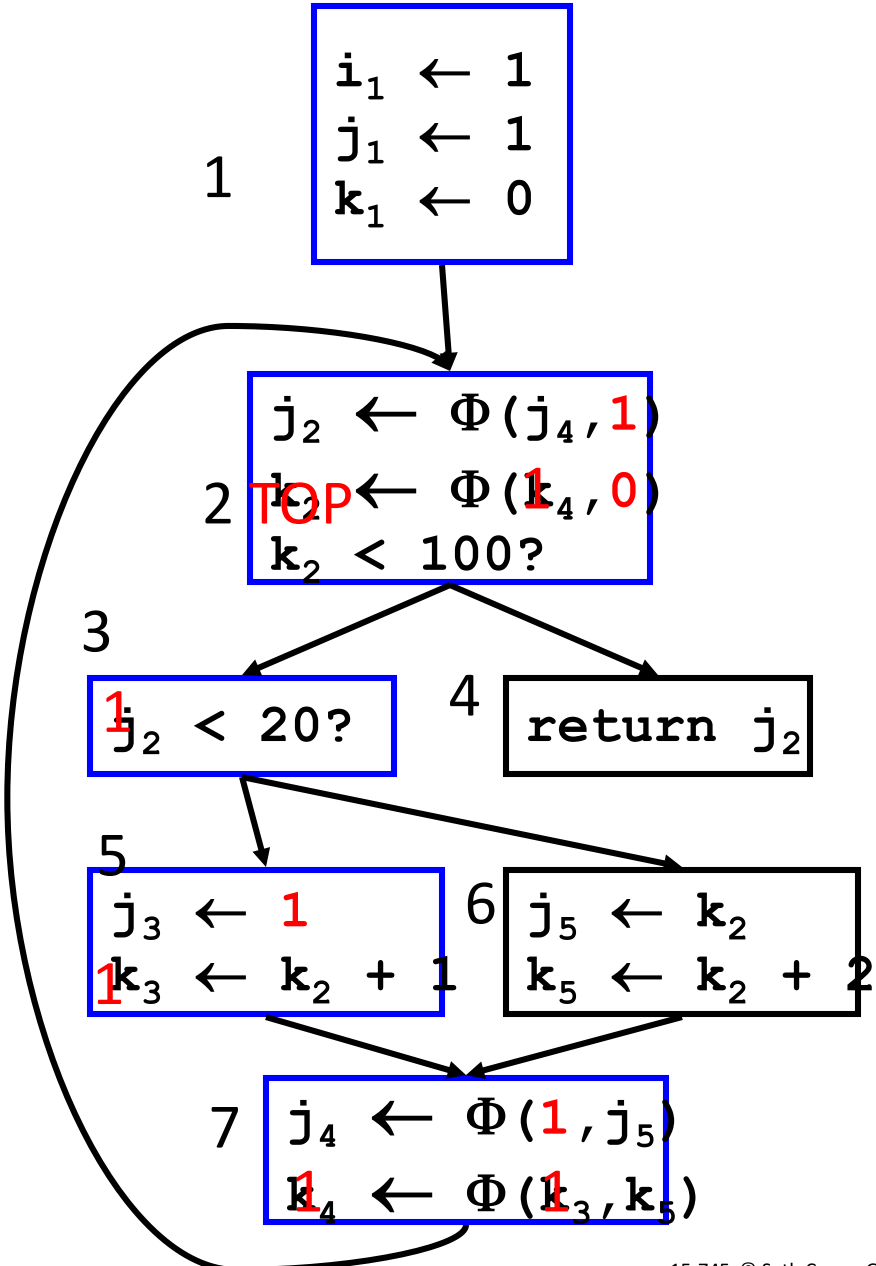
Conditional Constant Propagation



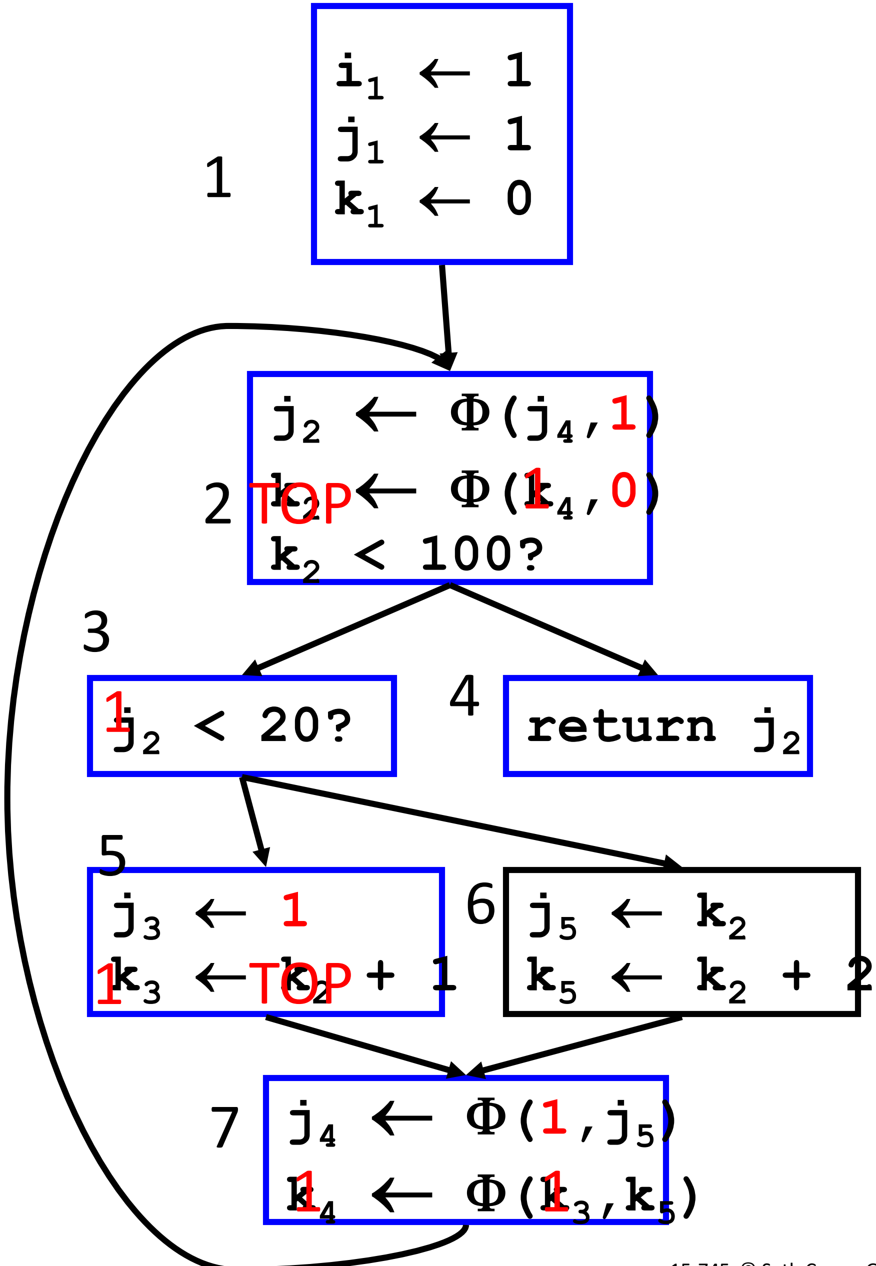
Conditional Constant Propagation



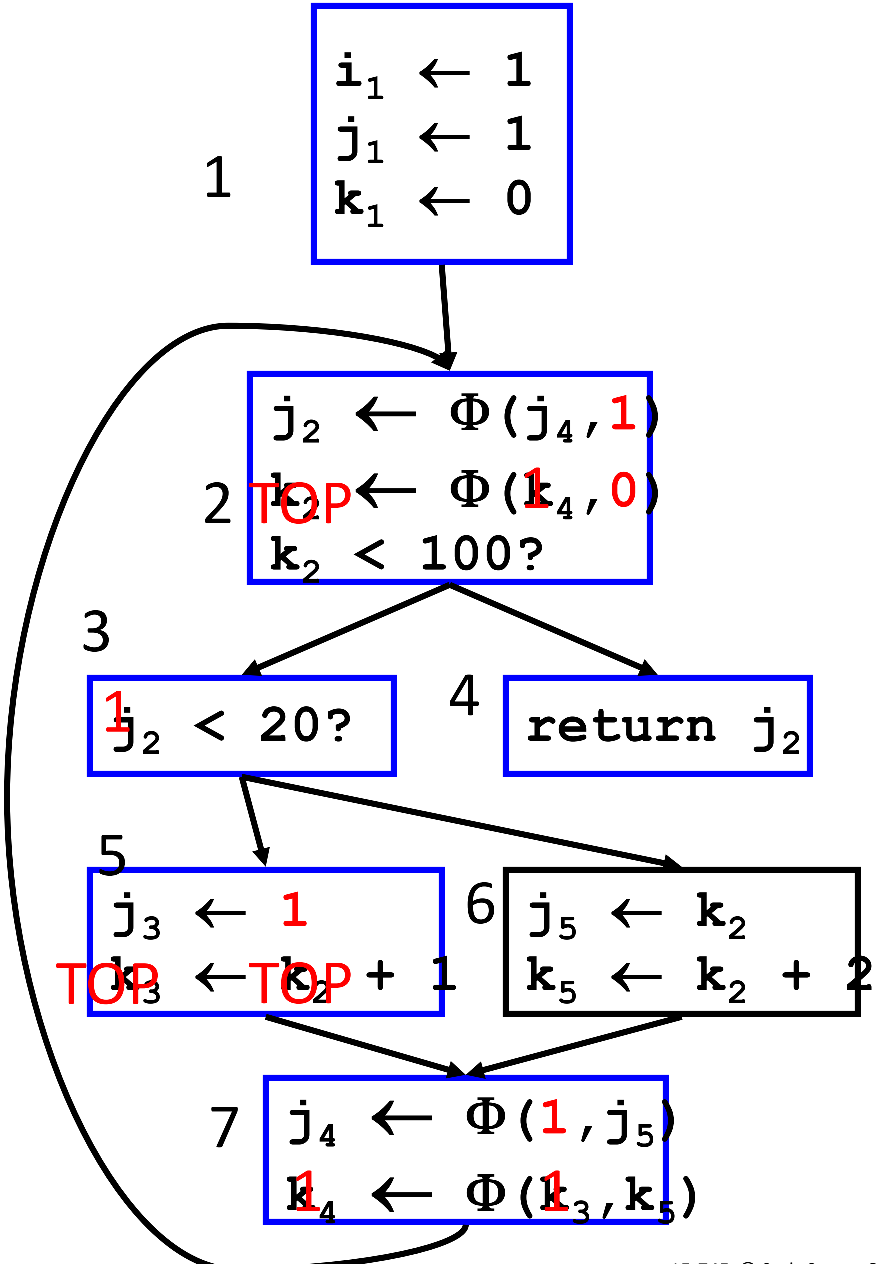
Conditional Constant Propagation



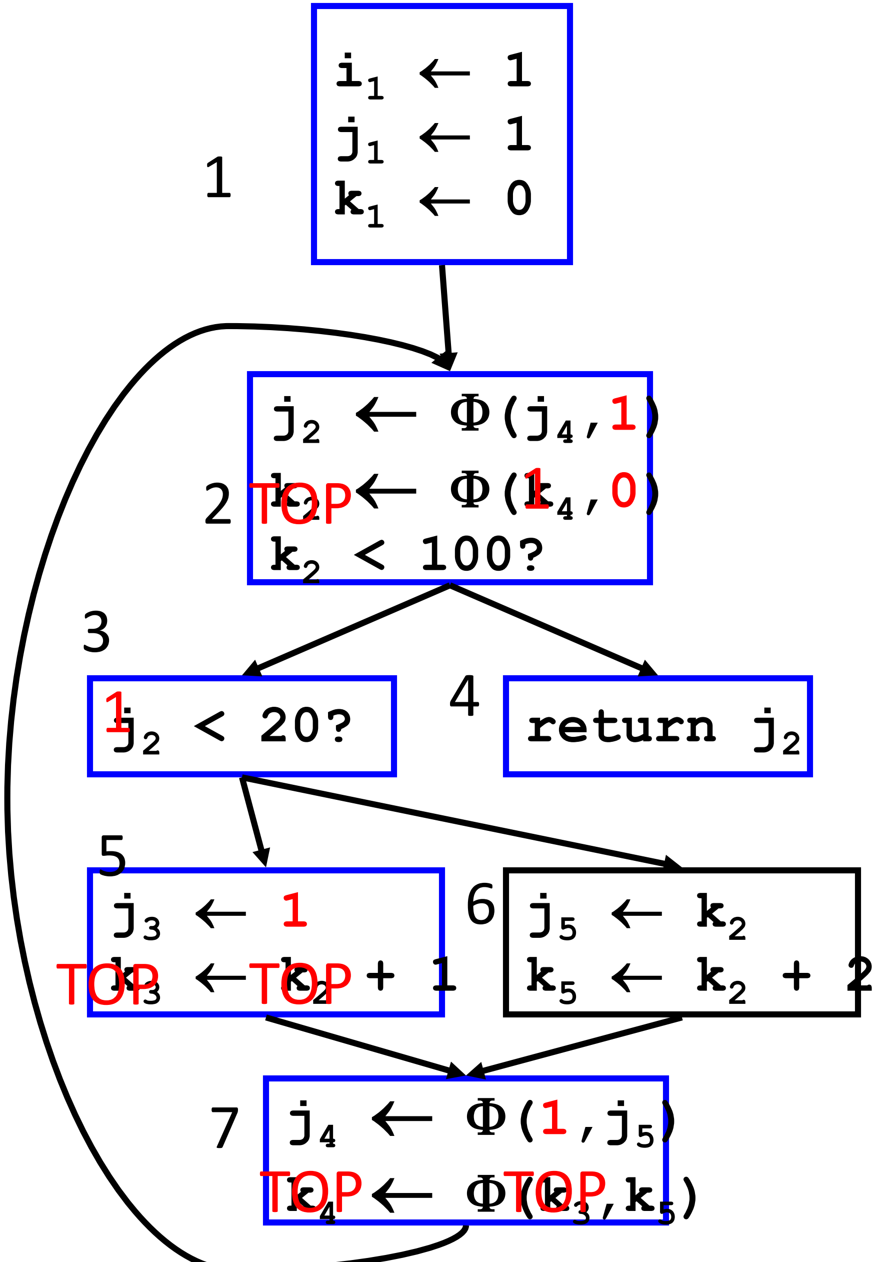
Conditional Constant Propagation



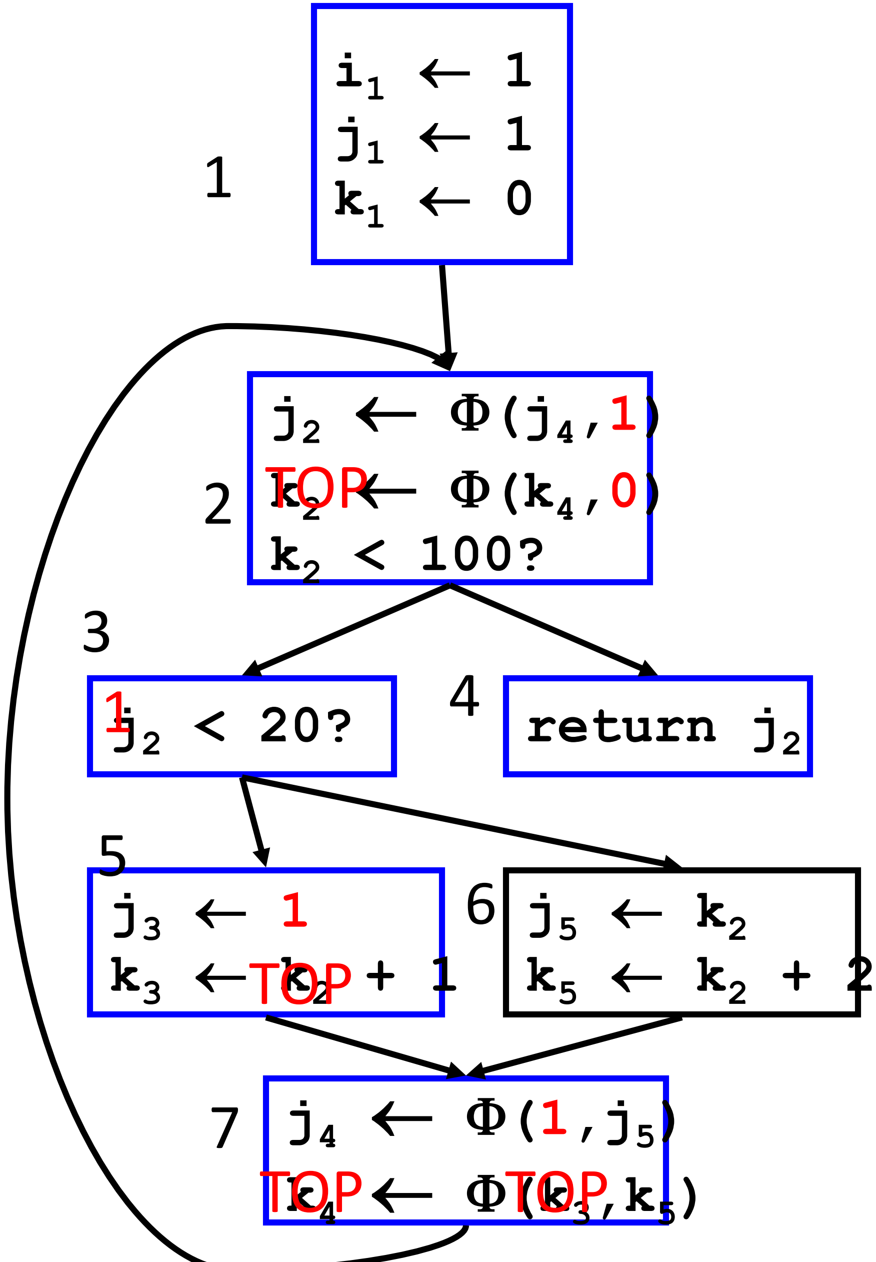
Conditional Constant Propagation



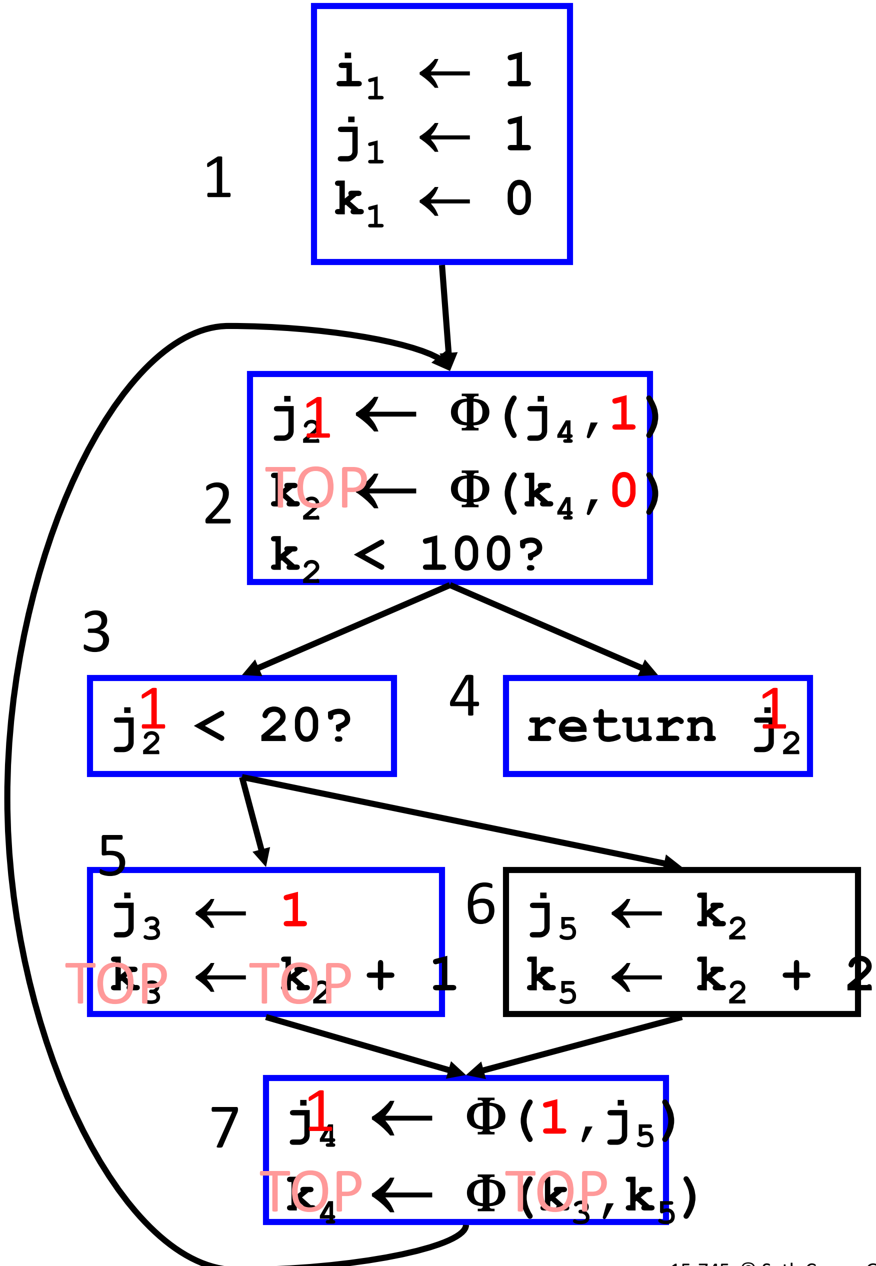
Conditional Constant Propagation



Conditional Constant Propagation

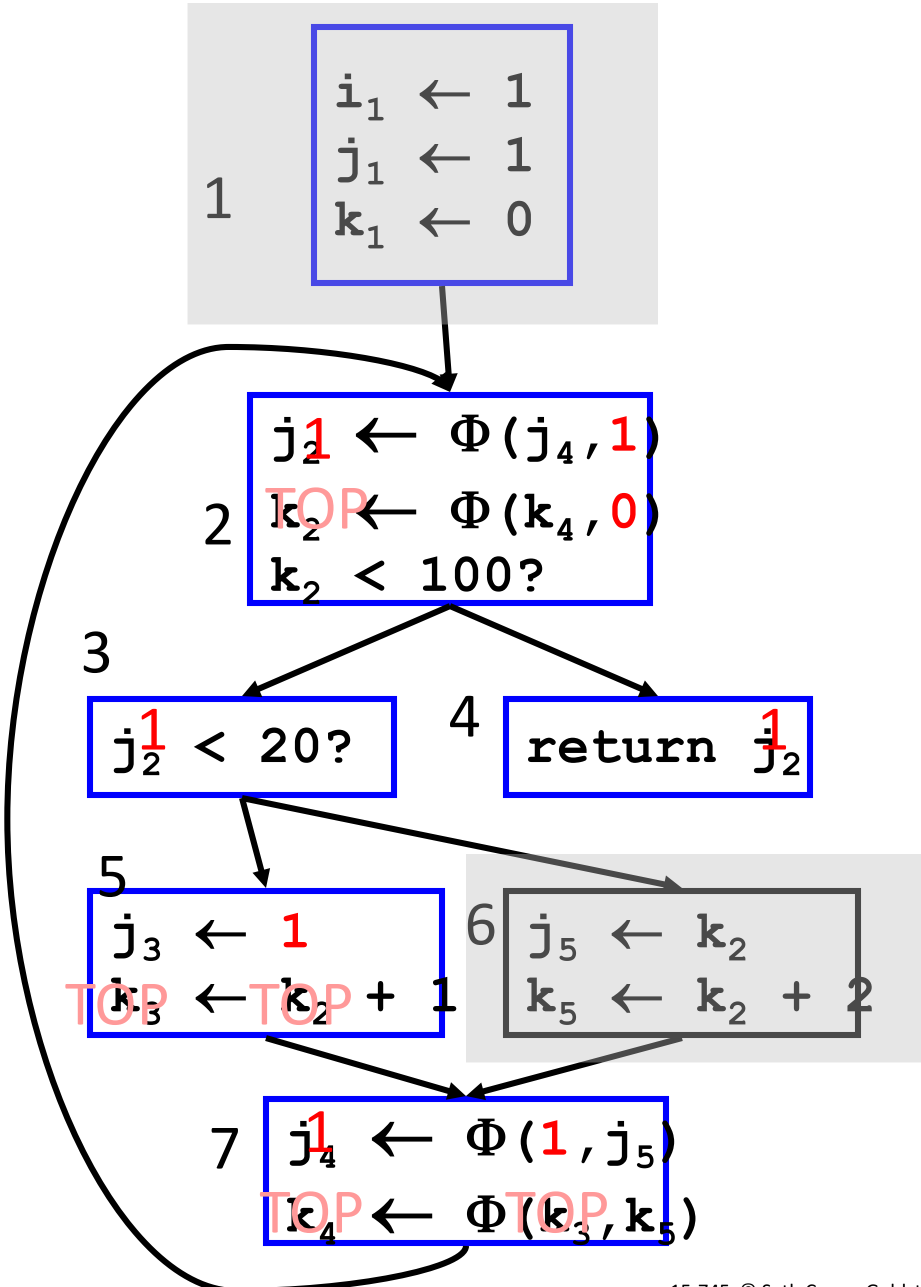


Conditional Constant Propagation



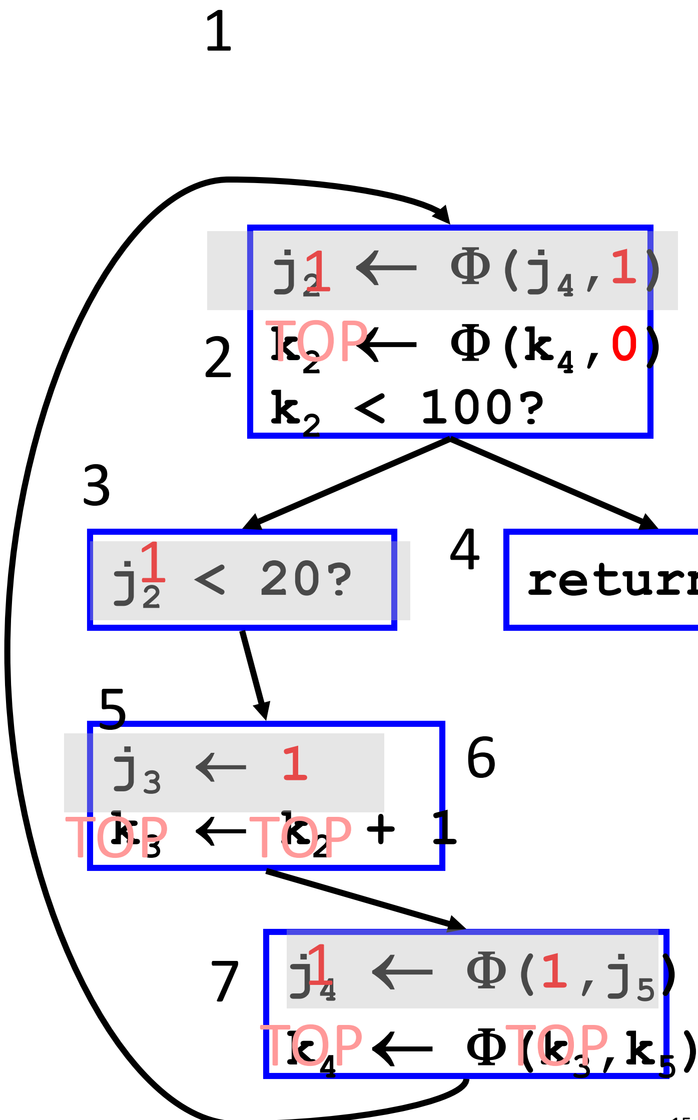
Now What?

Conditional Constant Propagation



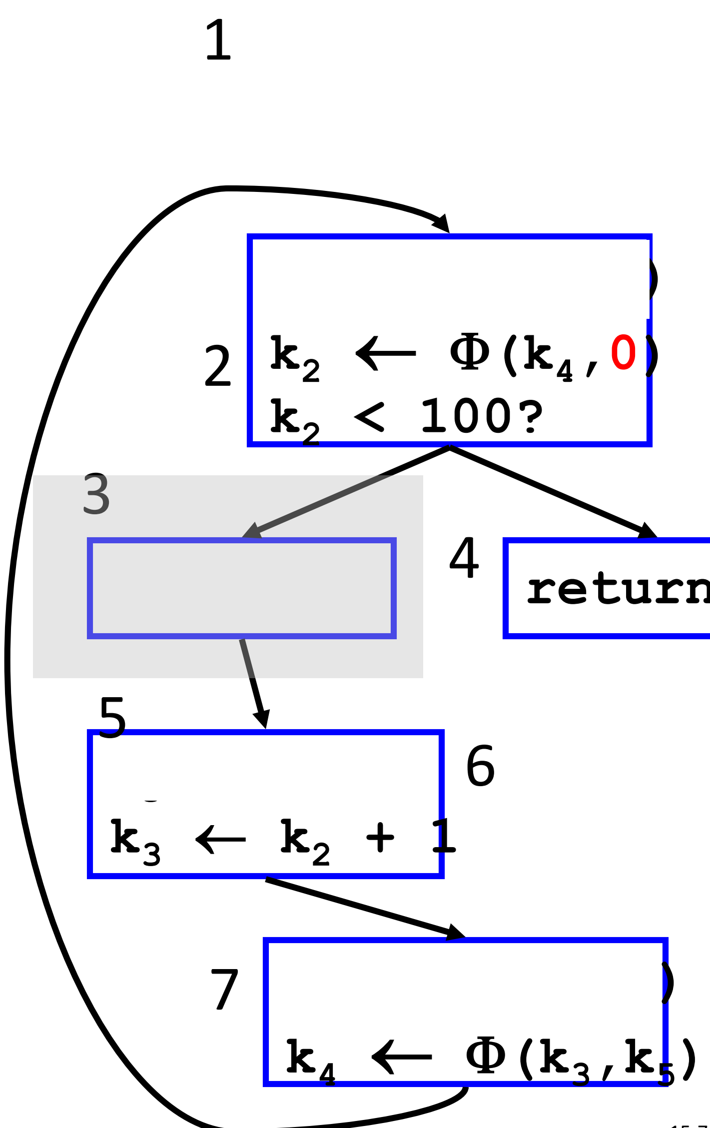
Now What?

Conditional Constant Propagation



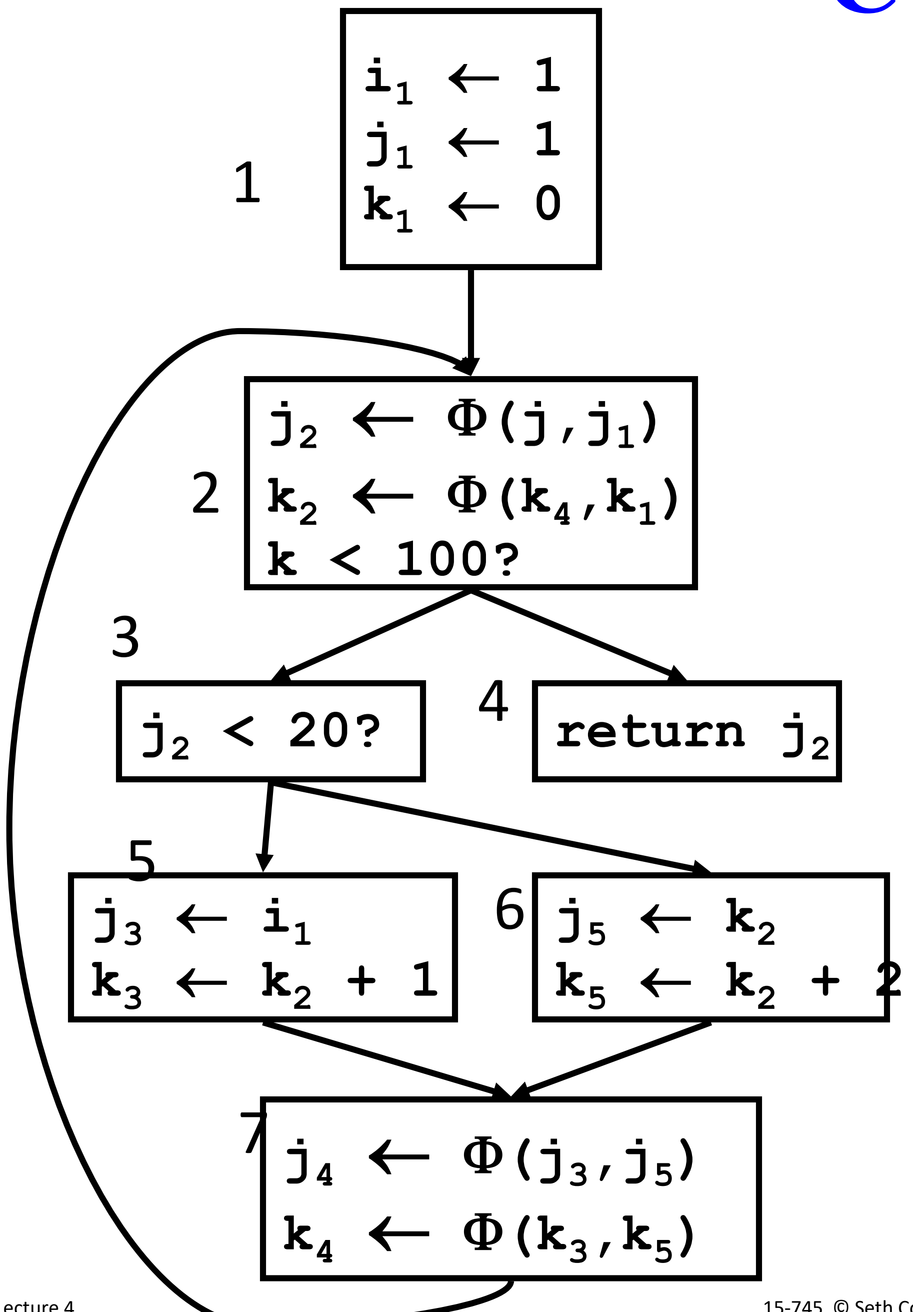
Now What?

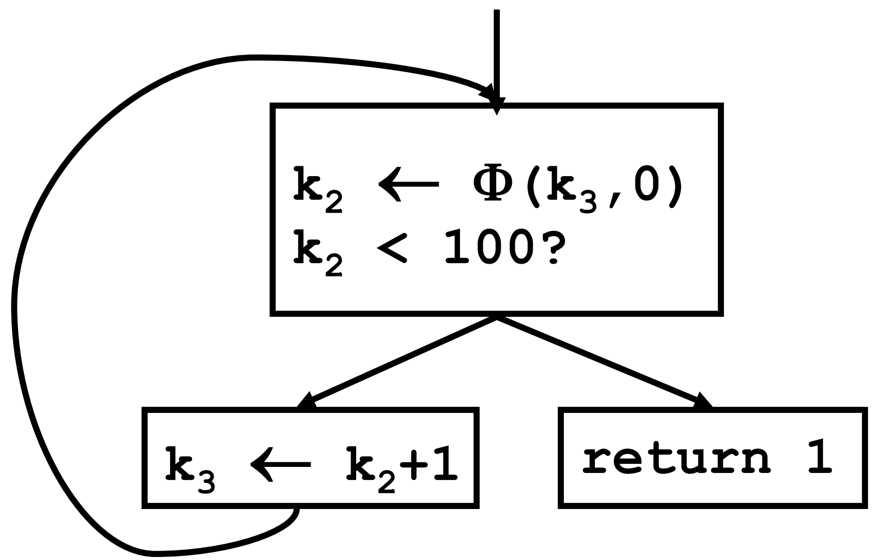
Conditional Constant Propagation



Now What?

CCP

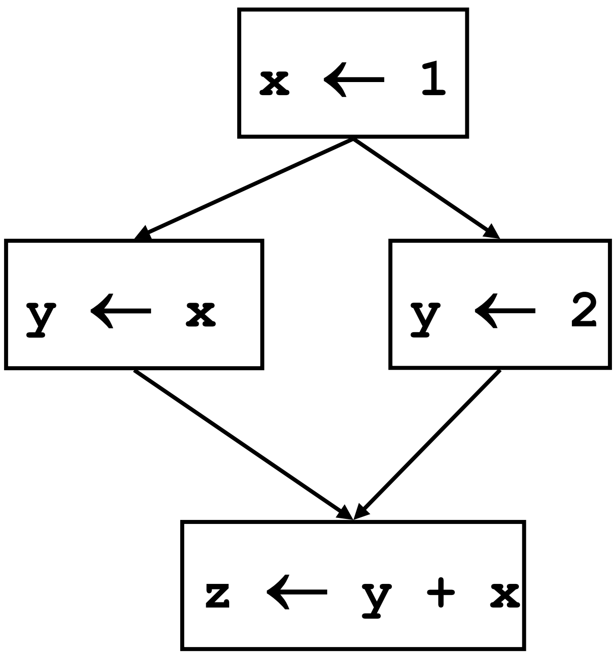


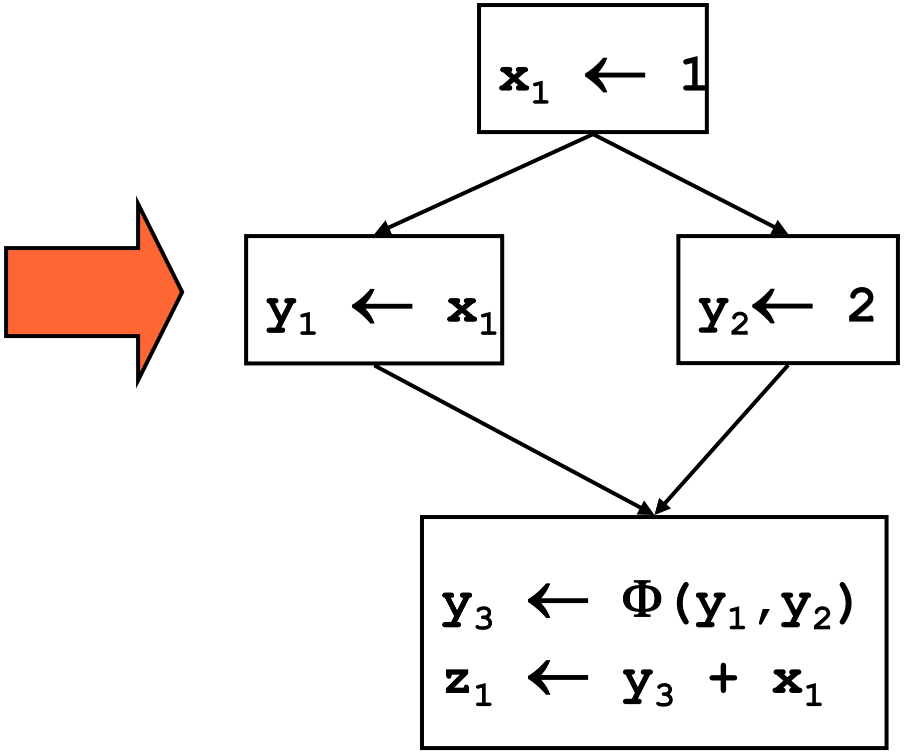


Just a taste of the kinds of optimizations that SSA enables.

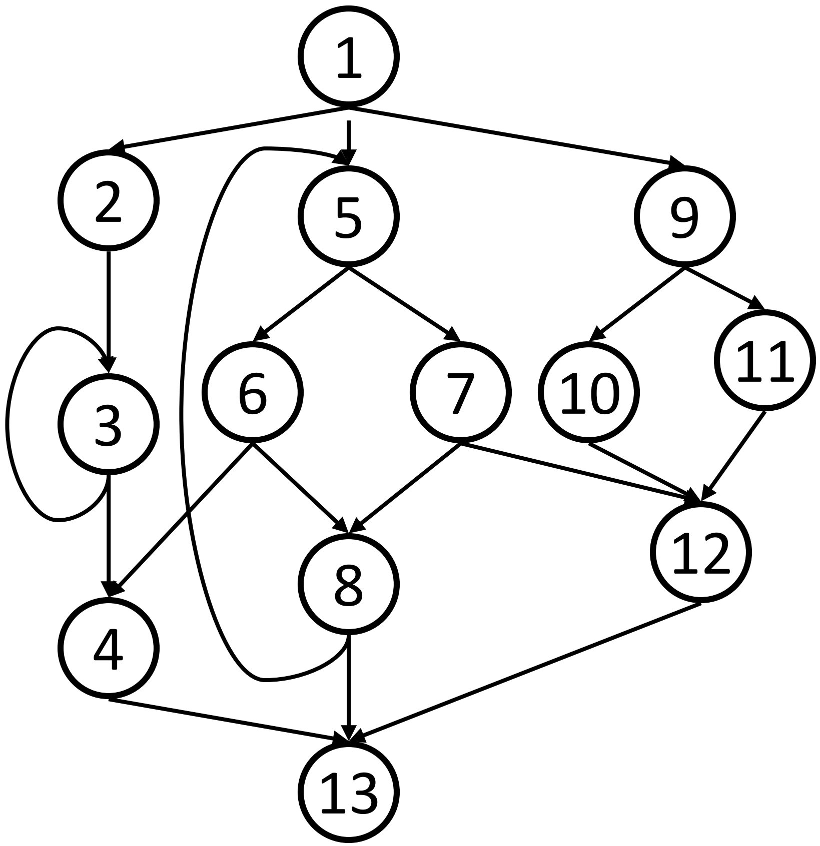
Minimal SSA

- 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?



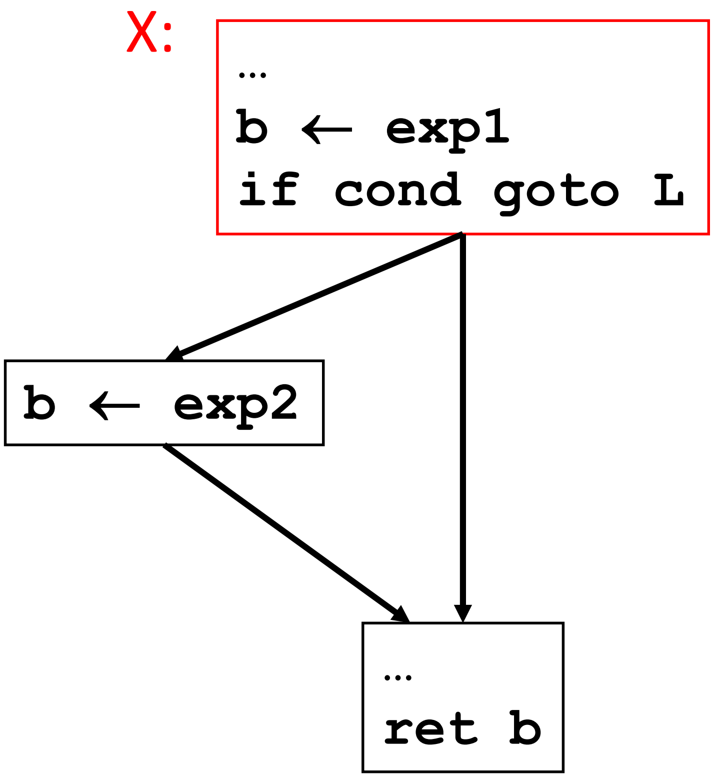
CFG

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

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:

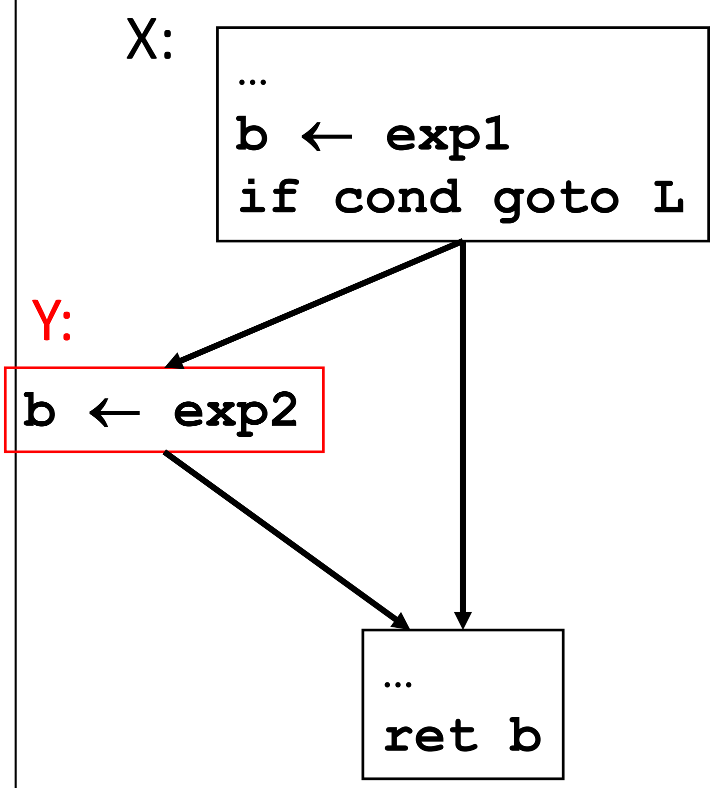
- There is a block x containing a def of b
- There is a block y (with y ≠ x) containing a def of b
- There is a nonempty path P_{xz} of edges from x to z
- There is a nonempty path P_{yz} of edges from y to z
- Paths P_{xz} and P_{yz} do not have any node in common other than z, and…
- The node z does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



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:

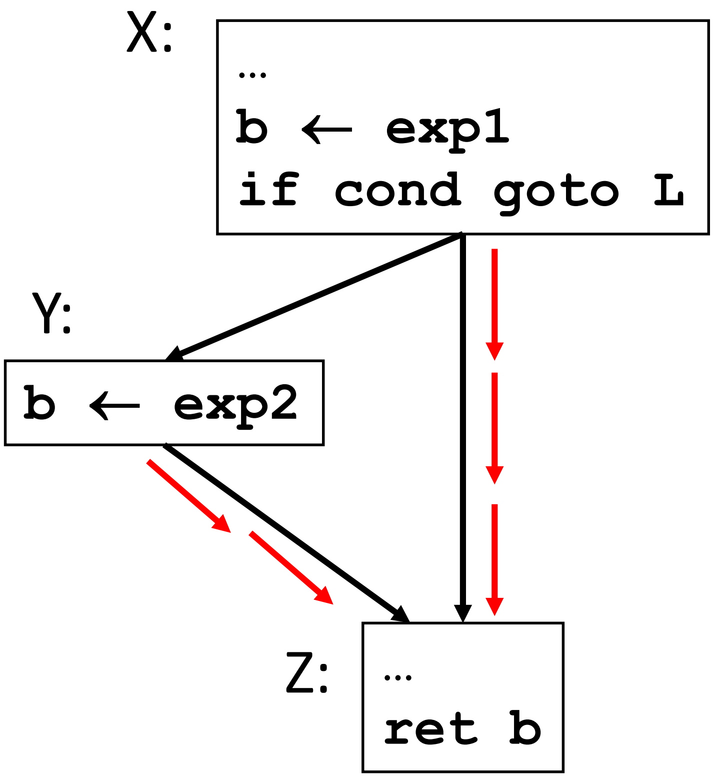
- There is a block x containing a def of b
- There is a block y (with y ≠ x) containing a def of b
- There is a nonempty path P_{xz} of edges from x to z
- There is a nonempty path P_{yz} of edges from y to z
- Paths P_{xz} and P_{yz} do not have any node in common other than z, and…
- The node z does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



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:

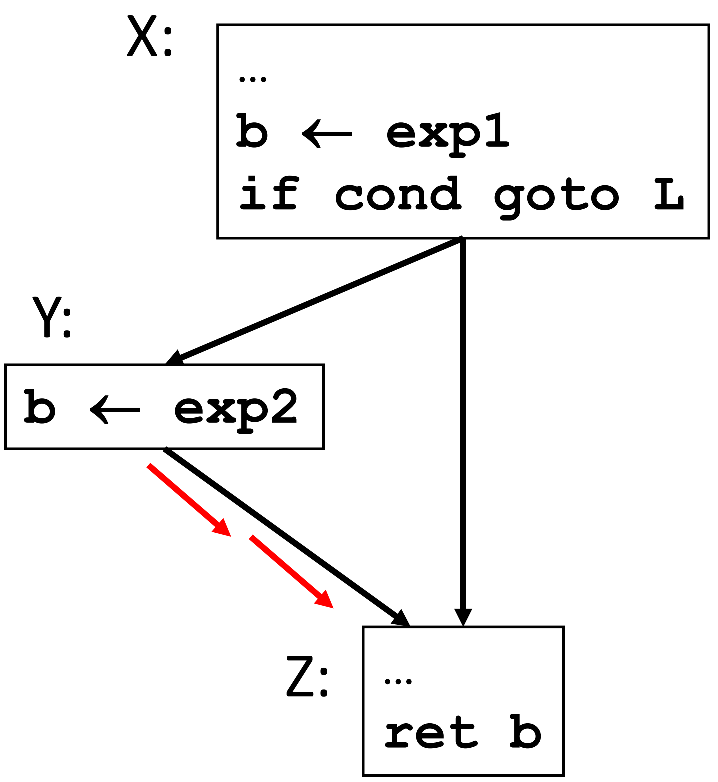
- There is a block x containing a def of b
- There is a block y (with y ≠ x) containing a def of b
- There is a nonempty path P_{xz} of edges from x to z
- There is a nonempty path P_{yz} of edges from y to z
- Paths P_{xz} and P_{yz} do not have any node in common other than z, and…
- The node z does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



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:

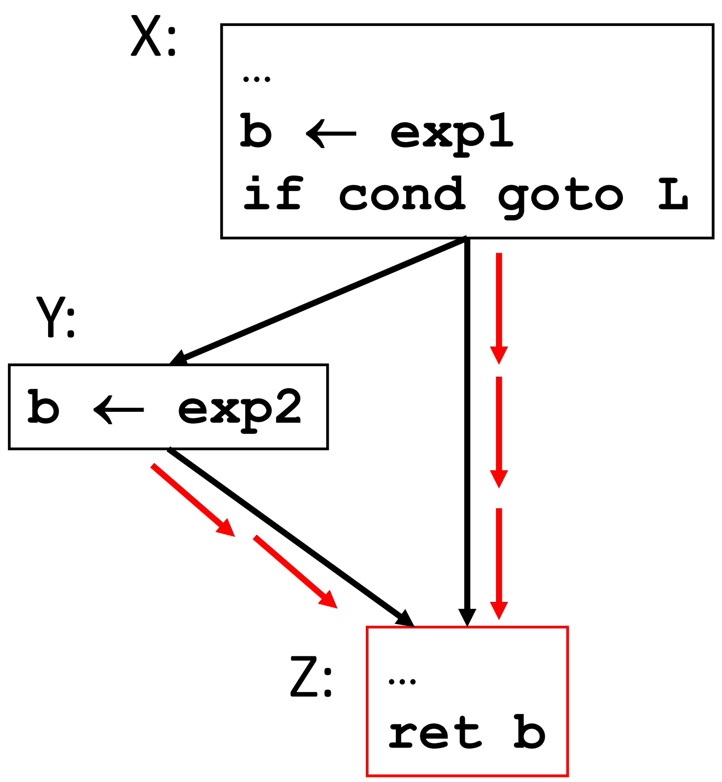
- There is a block x containing a def of b
- There is a block y (with y ≠ x) containing a def of b
- There is a nonempty path P_{xz} of edges from x to z
- There is a nonempty path P_{yz} of edges from y to z
- Paths P_{xz} and P_{yz} do not have any node in common other than z, and…
- The node z does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



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:

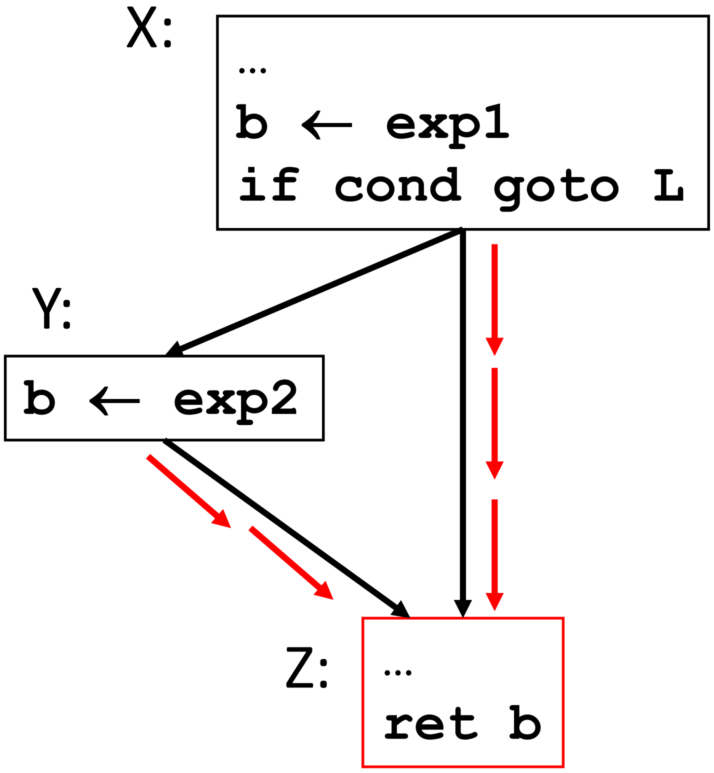
- There is a block x containing a def of b
- There is a block y (with y ≠ x) containing a def of b
- There is a nonempty path P_{xz} of edges from x to z
- There is a nonempty path P_{yz} of edges from y to z
- Paths P_{xz} and P_{yz} do not have any node in common other than z, and…
- The node z does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



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.



Iterative Insertion

- Implicit def of every variable in start node
- Inserting Φ-function creates new definition
- While there $\exists$ x,y,z that
  - satisfy path-convergence critera
  - 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 ith pred 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 alg to convert to SSA

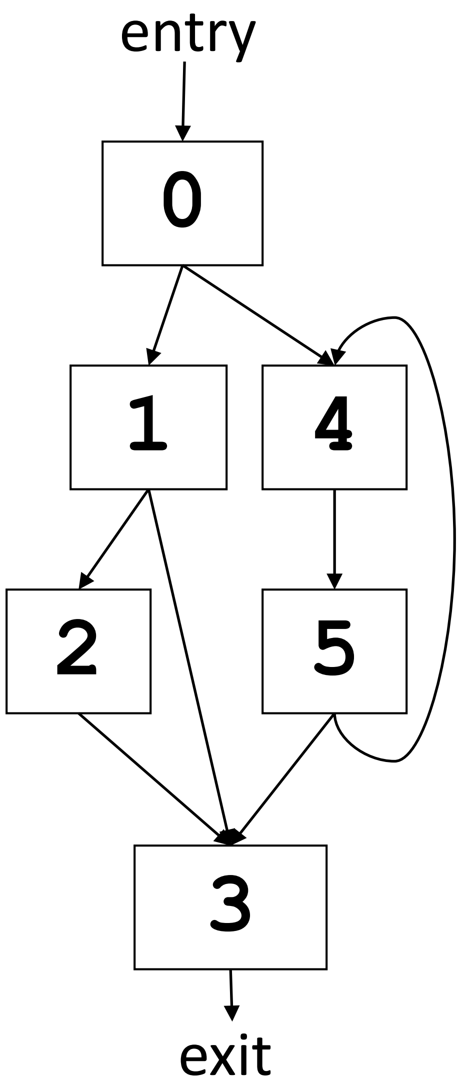
(actually only true for strict SSA, where all variables are defined before they are used.

Side trip: Dominators

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 and

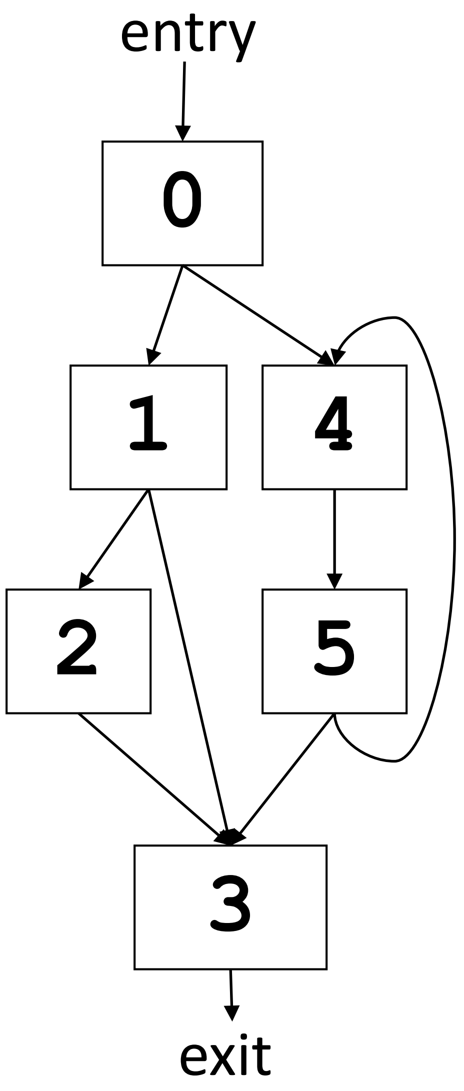
Dominators are useful in indentifying “natural” loops



Dominators

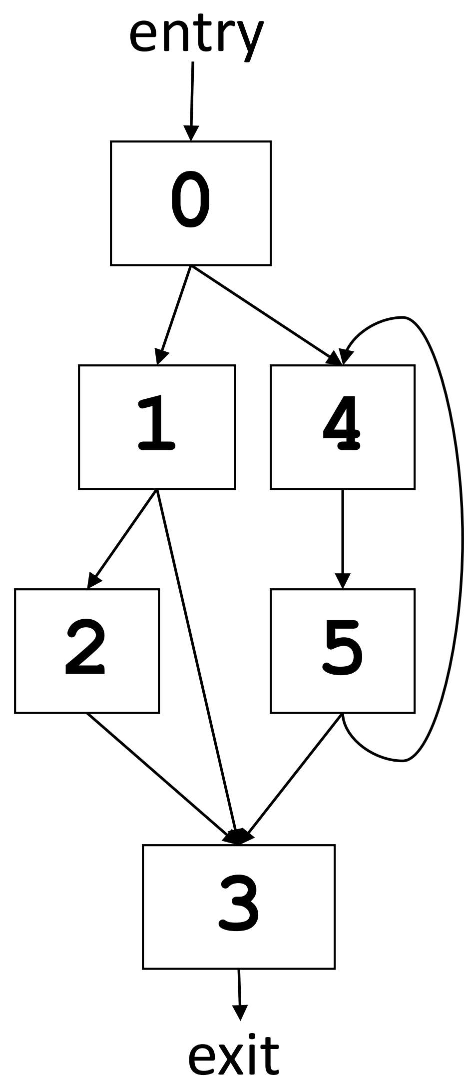
- *a dom b*
  - block *a dominates* block *b* if every possible execution path from *entry* to *b* includes *a*
    - **entry** dominates everything
    - **0** dominates everything but entry
    - **1** dominates **2** and **1**

Dominators are useful in indentifying "natural" loops



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*[entry]) to be the set of all blocks, and then iterate until no *D*[*n*] changes:

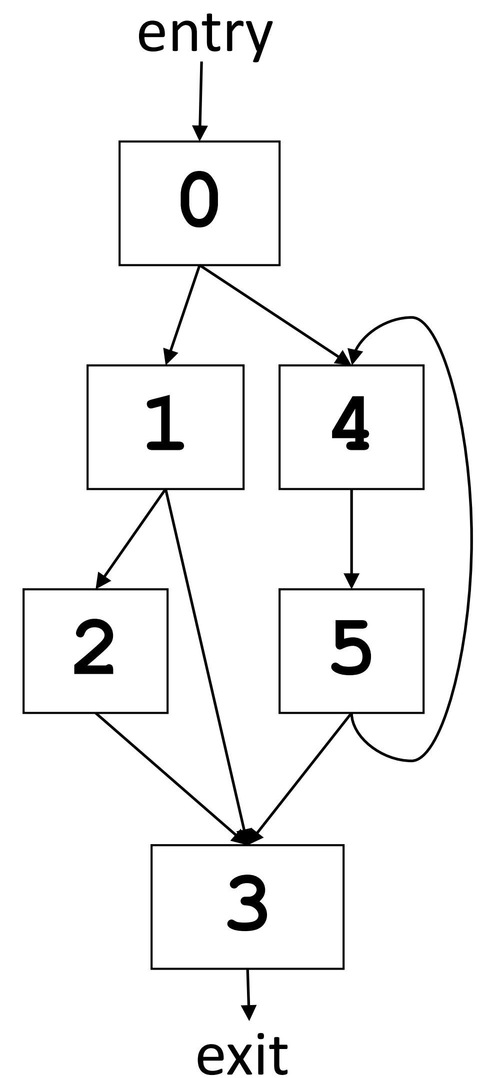
$$D[\text{entry}] = \{\text{entry}\}$$

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

Skip example

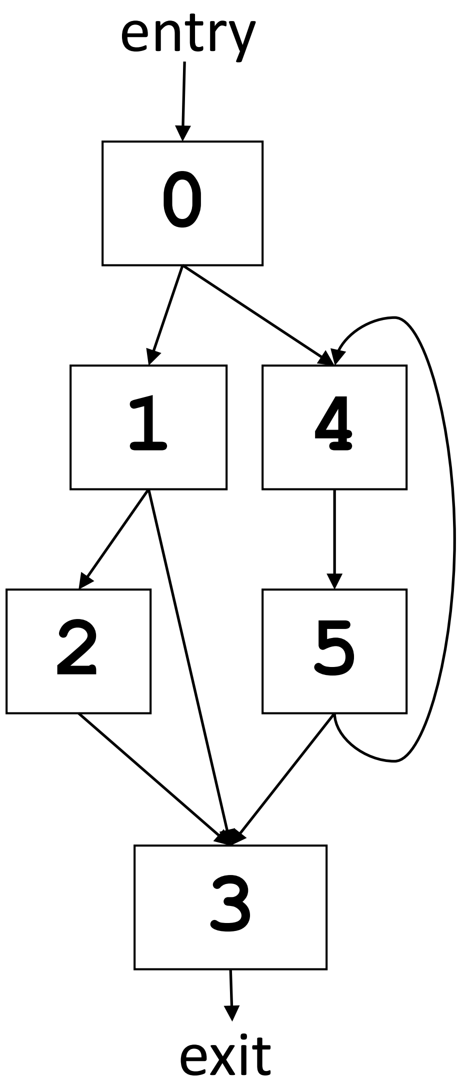
Example

block	Initialization 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}

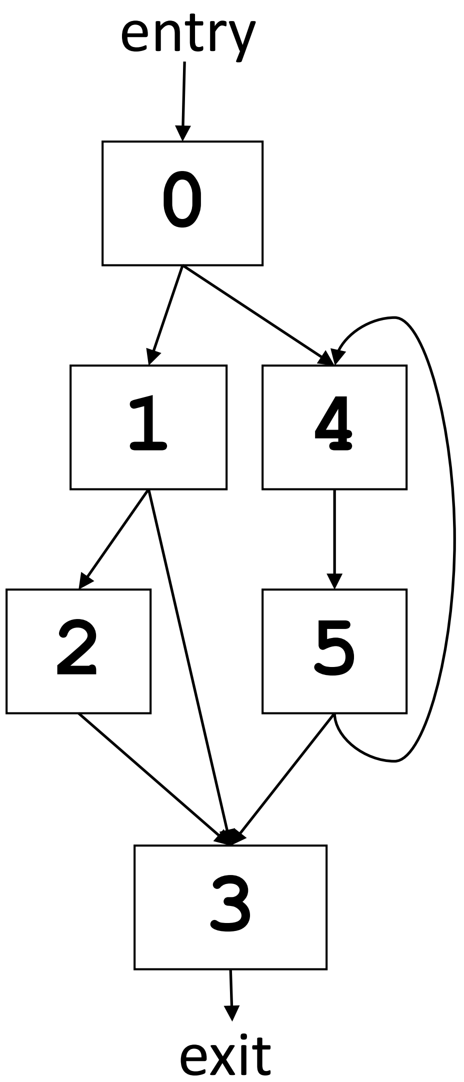


Update rule:

$$D[n] = \{n\} \cup \left(\bigcap_{p \in pred(n)} D[p] \right)$$

Example

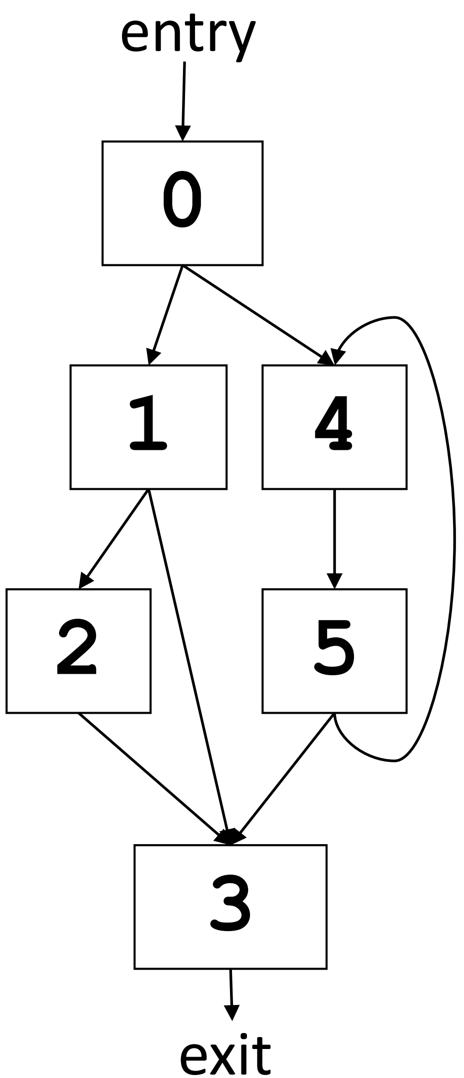
block	First Pass D[n]	Second Pass D[n]
entry	{entry}	{entry}
0	{0,entry}	{0,entry}
1	{1,0,entry}	{1,0,entry}
2	{2,1,0,entry}	{2,1,0,entry}
3	{3,1,0,entry}	{3,0,entry}
4	{4,0,entry}	{4,0,entry}
5	{5,4,0,entry}	{5,4,0,entry}
exit	{exit,3,1,0,entry}	{exit,3,0,entry}



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

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

Computing dominators

- Iterative algorithm is $O(n^2e)$
  - assuming bit vector sets
- More efficient algorithm due to Lengauer and Tarjan
  - $O(e\cdot\alpha(e,n))$
  - much more complicated
  - your book provides a simple algorithm that is very fast in practice
    - faster than Tarjan algorithm for any realistic CFG

$\alpha(e,n)$ is *inverse Ackermann*

Computing iDom(n)

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

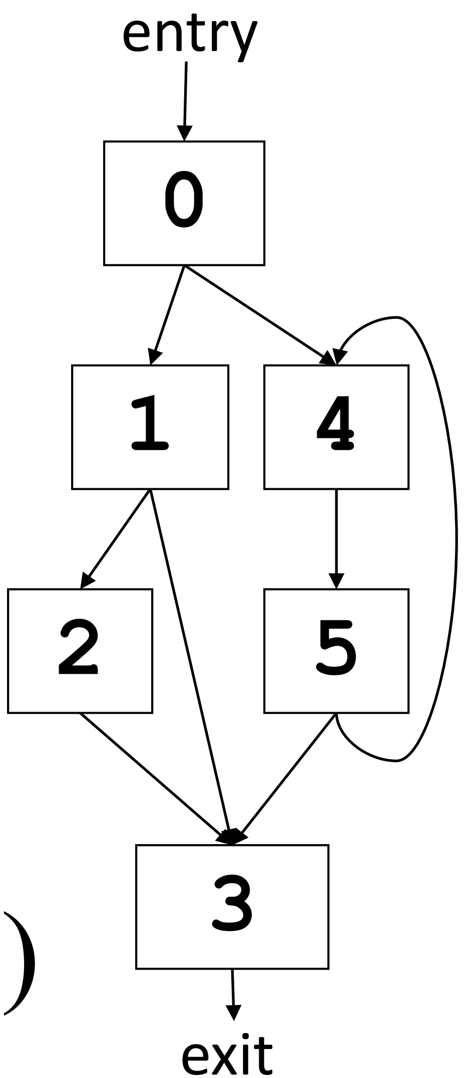
$$sD[n] = D[n] - \{n\}$$

- 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}	
1	{0,entry}	
2	{1,0,entry}	
3	{0,entry}	
4	{0,entry}	
5	{4,0,entry}	
exit	{3,0,entry}	



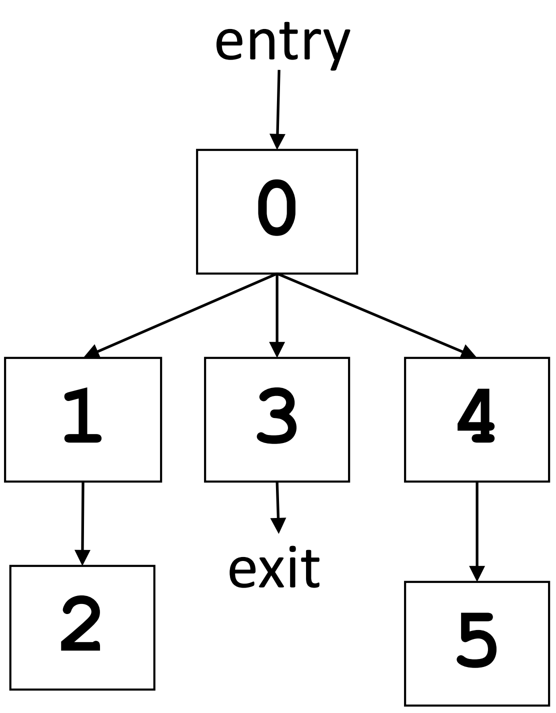
Update rule: $$iD[n] = iD[n] - \bigcup_{d \in iD[n]} (sD[d])$$

Dominator Tree

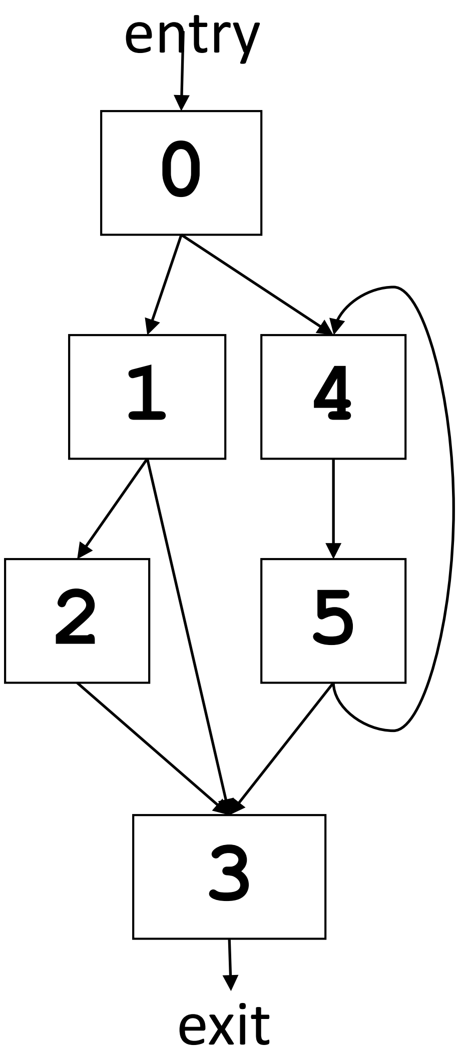
In the dominator tree the initial node is the entry block, and the parent of each other node is its immediate dominator.

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

Dominator Tree

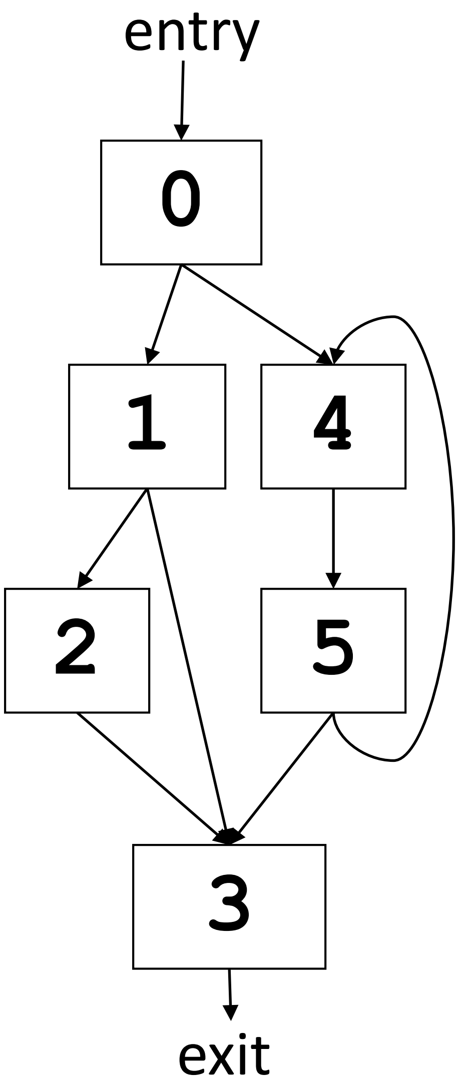


CFG



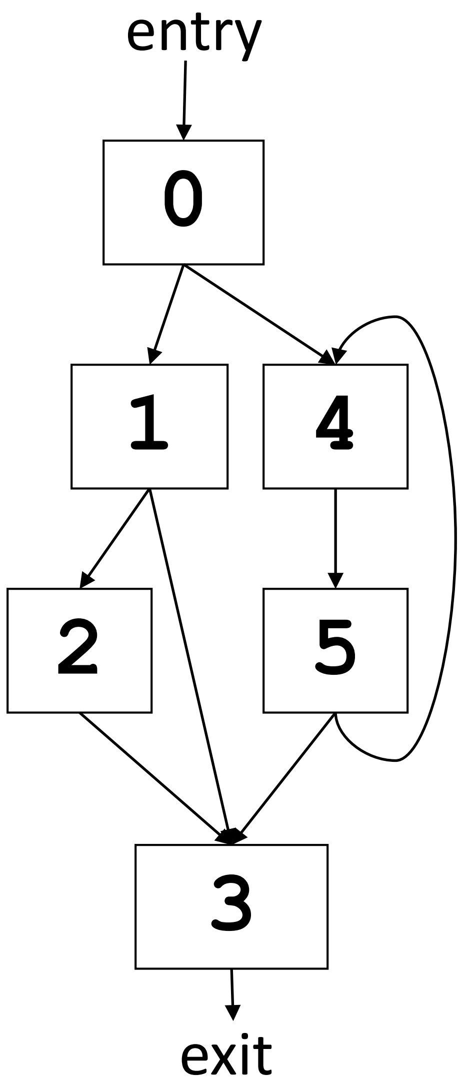
Dominance Frontier

- 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*,
  $x \rightarrow \ldots \rightarrow y \rightarrow z$
  where *x* dom *y* but not *x* sdom *z*.
- Dominance frontier of **1**?
- Dominance frontier of **2**?
- Dominance frontier of **4**?



Dominance Frontier

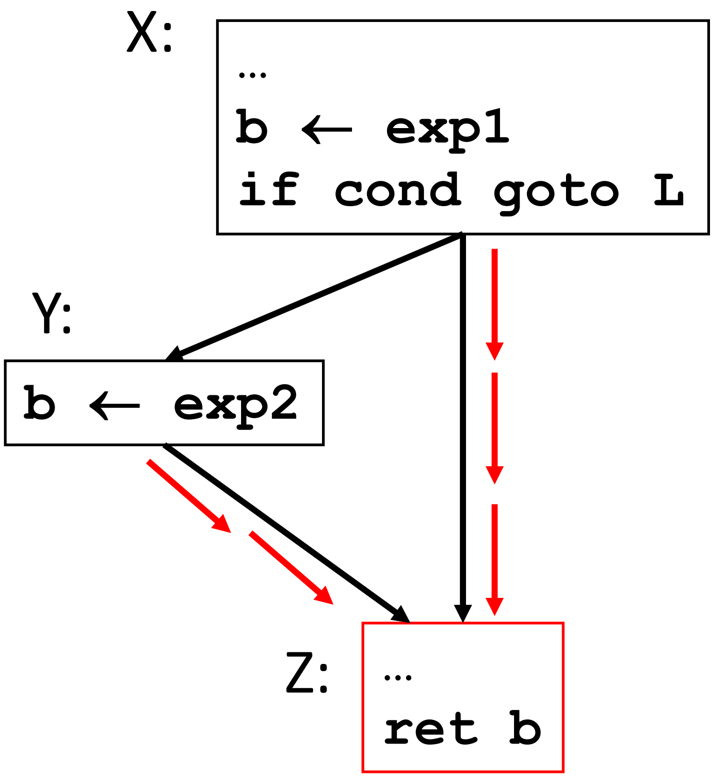
- If z is the first node we encounter on the path from x which x does not *strictly* dominate, z is in the dominance frontier of x
- For some path from node x to z,
  $x \rightarrow \ldots \rightarrow y \rightarrow z$
  where x dom y but not x sdom z.
- Dominance frontier of **1**? {3}
- Dominance frontier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



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)

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

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

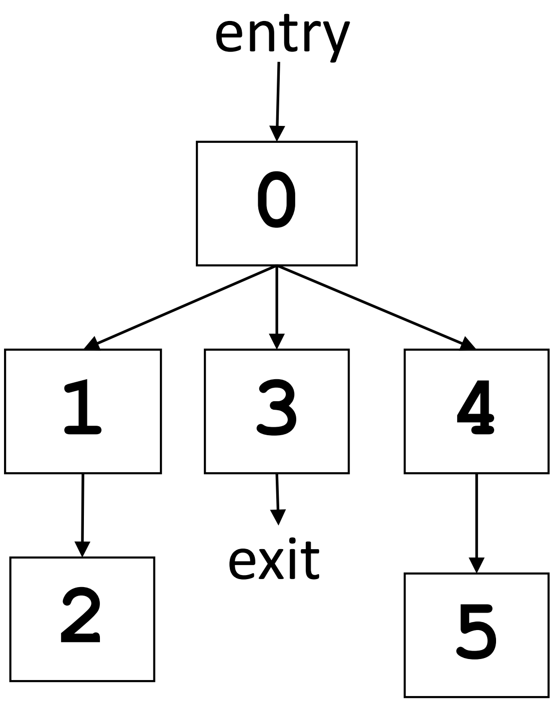
Skip example

Example

First calculate dominates[n] from the dominator tree

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

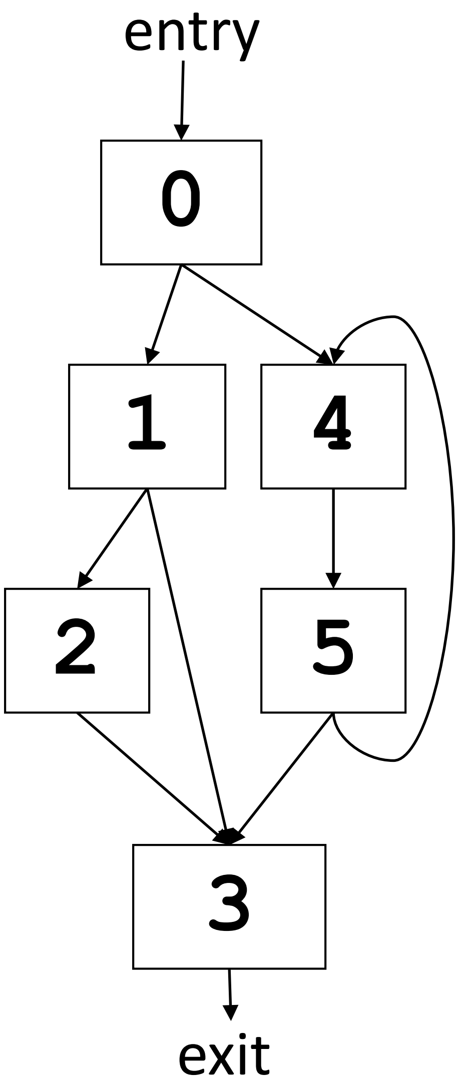
Dominator Tree



Example

Then compute the successor set of dominates[n]

block	dominates[n]	*succ*(dominates[n])
entry	{entry,0,1,2,3,4,5,exit}	
0	{0,1,2,3,4,5,exit}	
1	{1,2}	
2	{2}	
3	{3,exit}	
4	{4,5}	
5	{5}	
exit	{exit}	{}



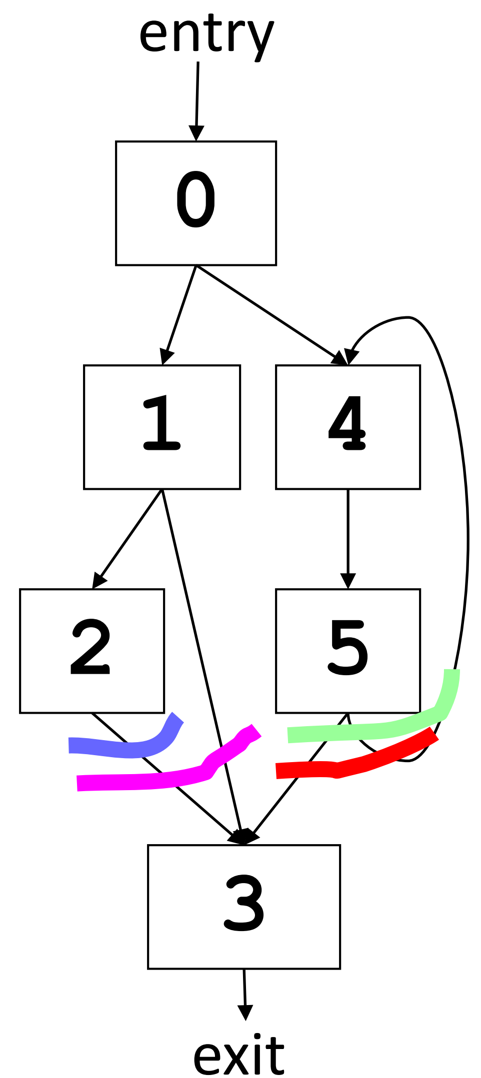
Example

Finally, remove all the blocks from the successor set that are strictly dominated by n to get DF[n]

block	sdominates[n]	*succ*(dominates[n])	DF[n]
entry	{entry,0,1,2,3,4,5,exit}	{0,1,2,3,4,5,exit}	
0	{0,1,2,3,4,5,exit}	{1,2,3,4,5,exit}	
1	{1,2}	{2,3}	
2	{2}	{3}	
3	{3,exit}	{exit}	
4	{4,5}	{3,4,5}	
5	{5}	{3,4}	
exit	{exit}	{}	{}

Example

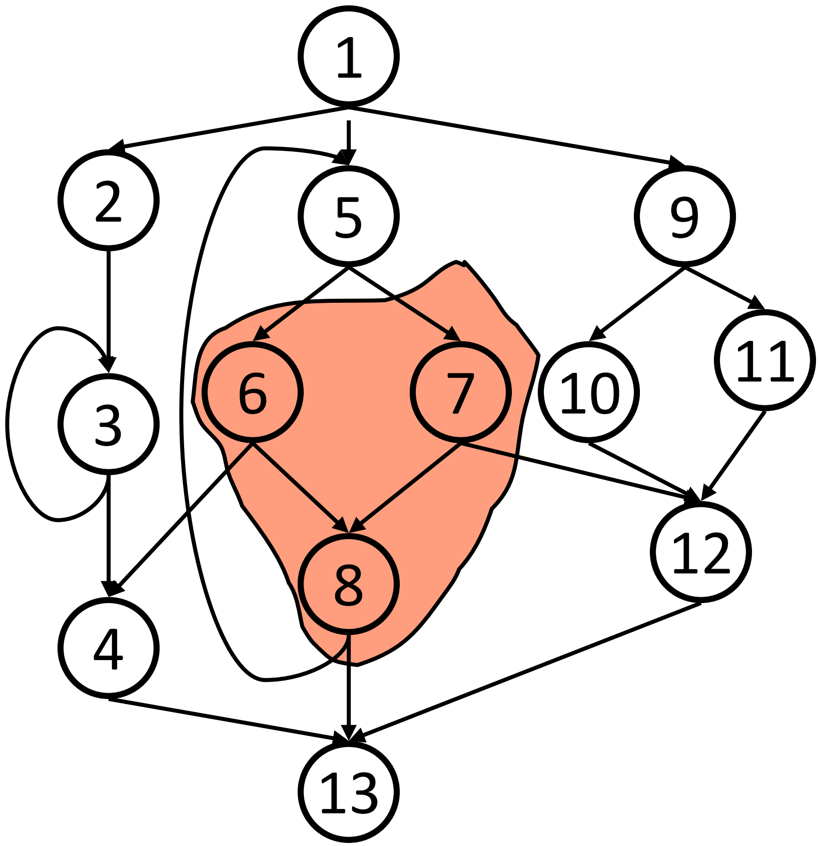
block	DF[n]
entry	{}
0	{}
1	{3}
2	{3}
3	{}
4	{3,4}
5	{3,4}
exit	{}

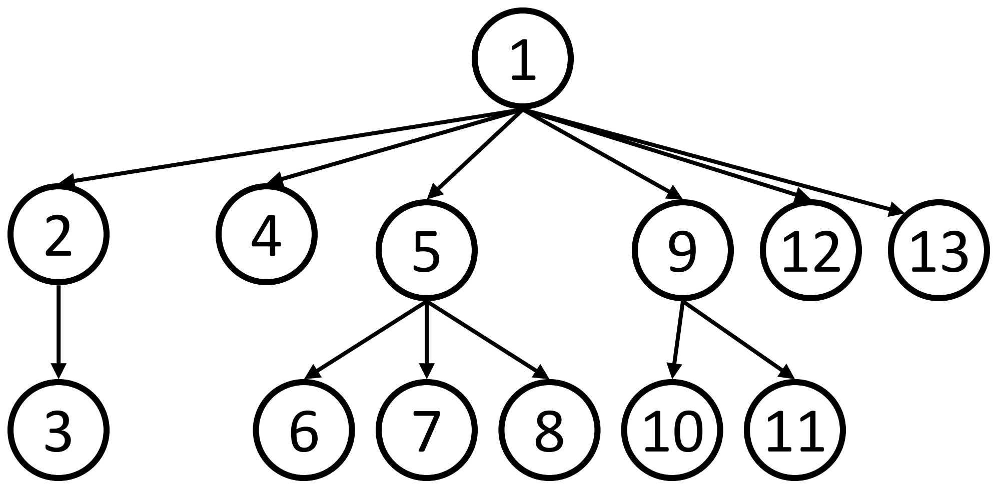


Recap

- *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

Dominance

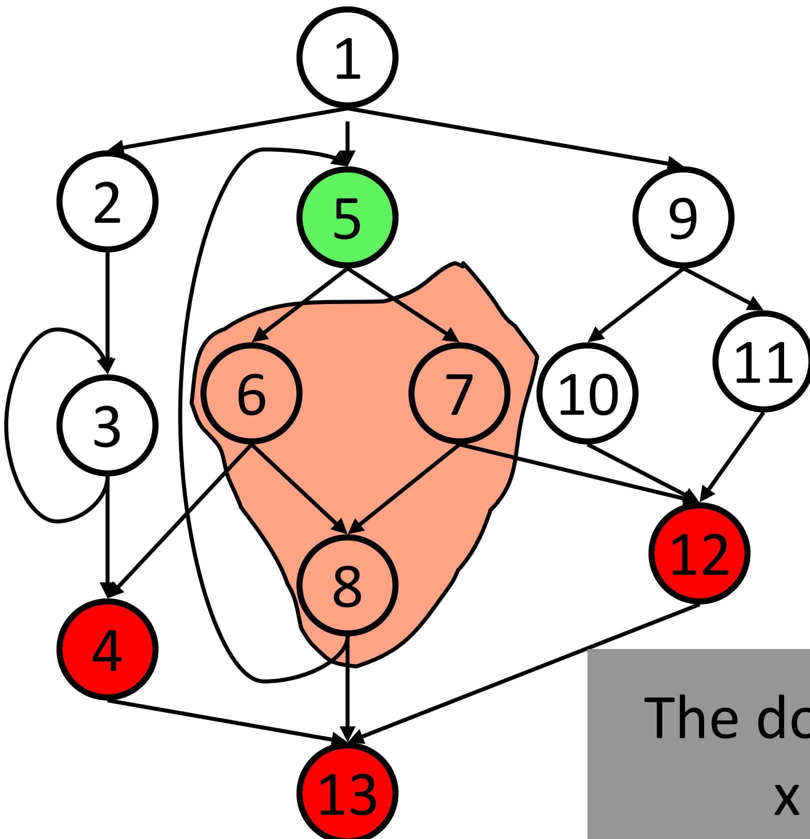




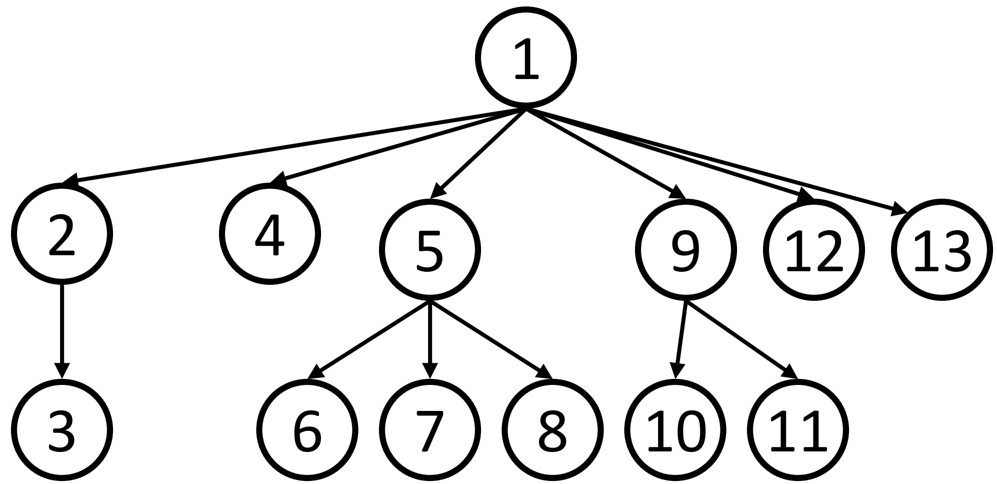
If there is a def of a in block 5, which nodes need a $\Phi()$?

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

Dominance Frontier



CFG

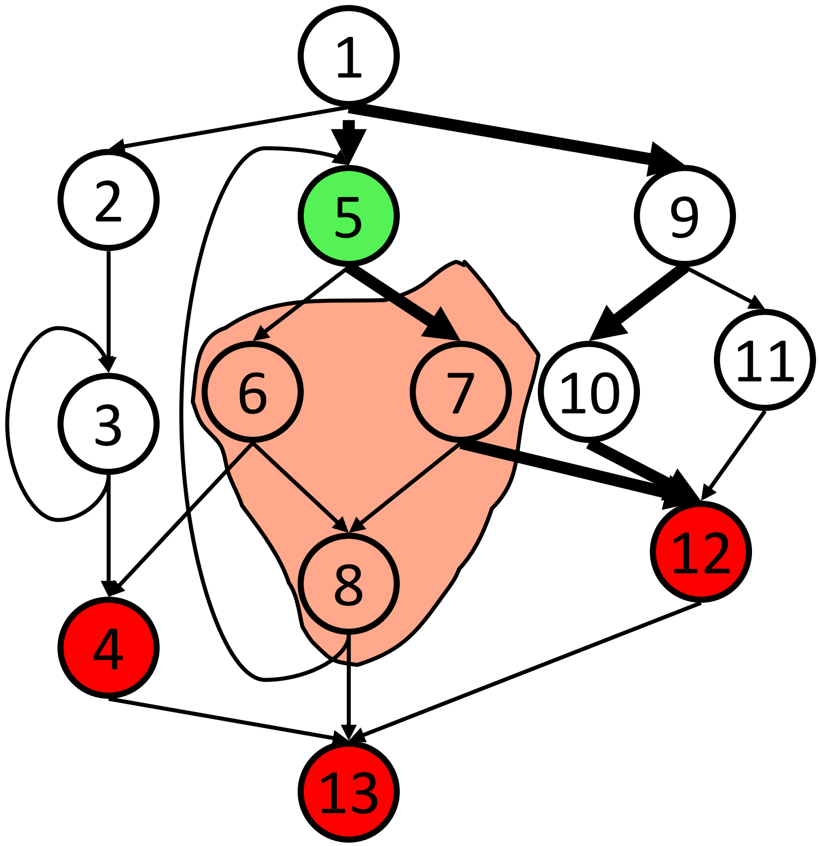


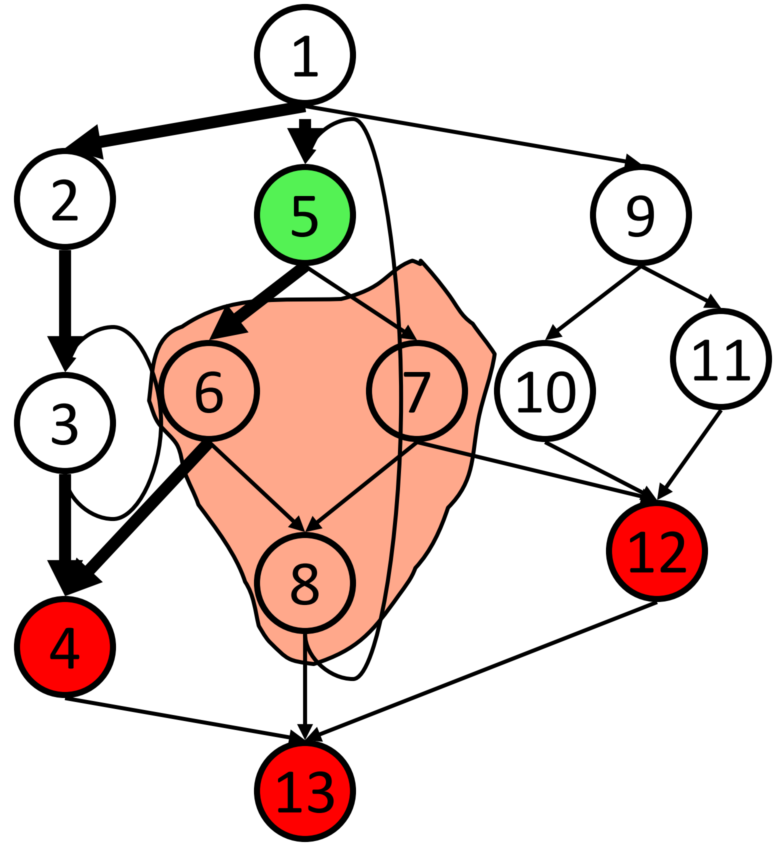
D-Tree

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

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

Dominance Frontier & path-convergence



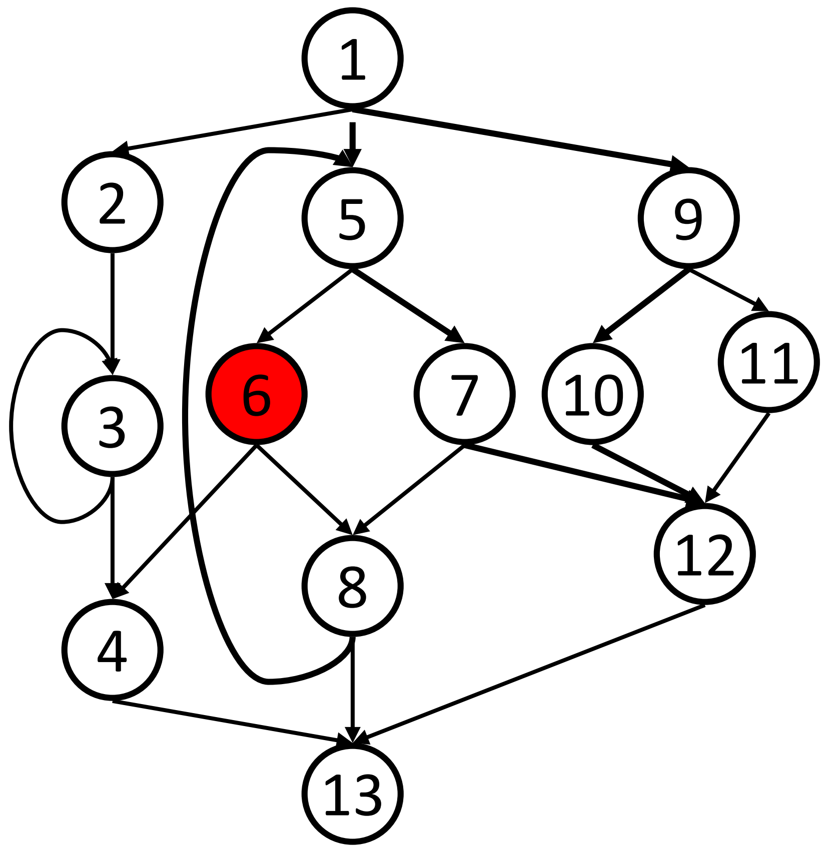


Dominance Frontier Criterion

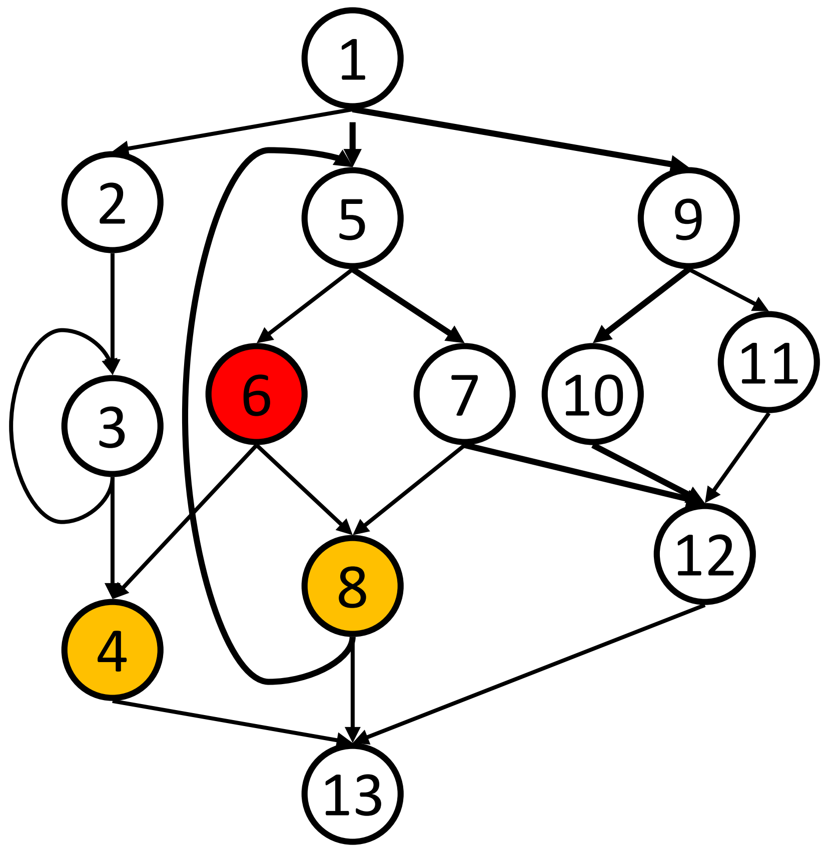
- **Dominance-Frontier Criterion**: Whenever node x contains a definition of some variable a, then any node z $\in$ DF(x), z needs a Φ-function for a.

- **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 Frontier Criterion

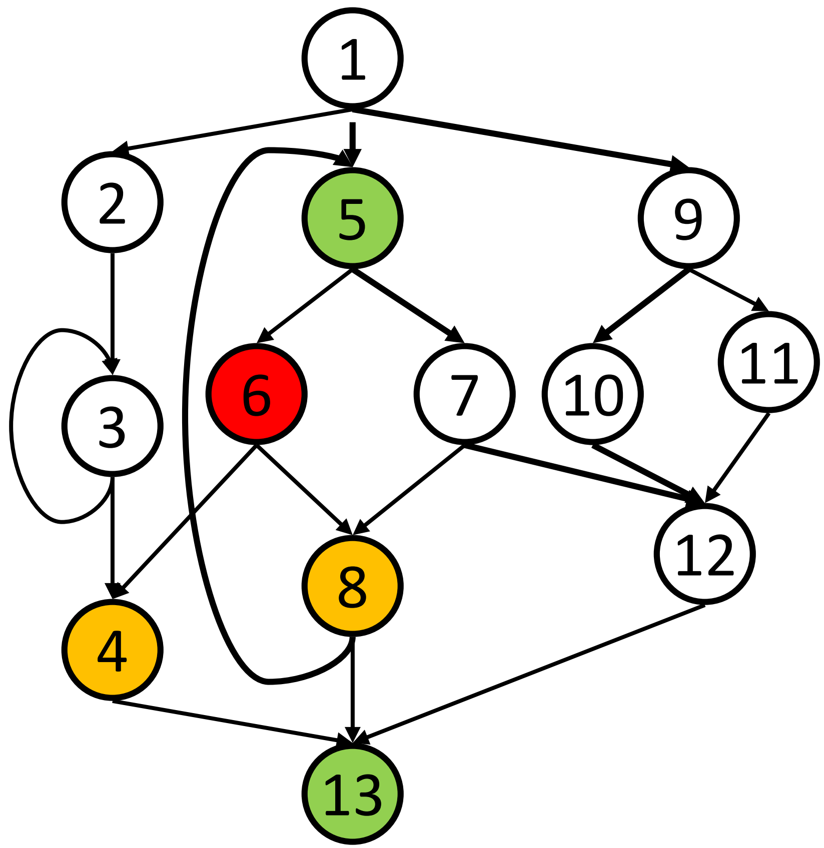


Dominance Frontier Criterion



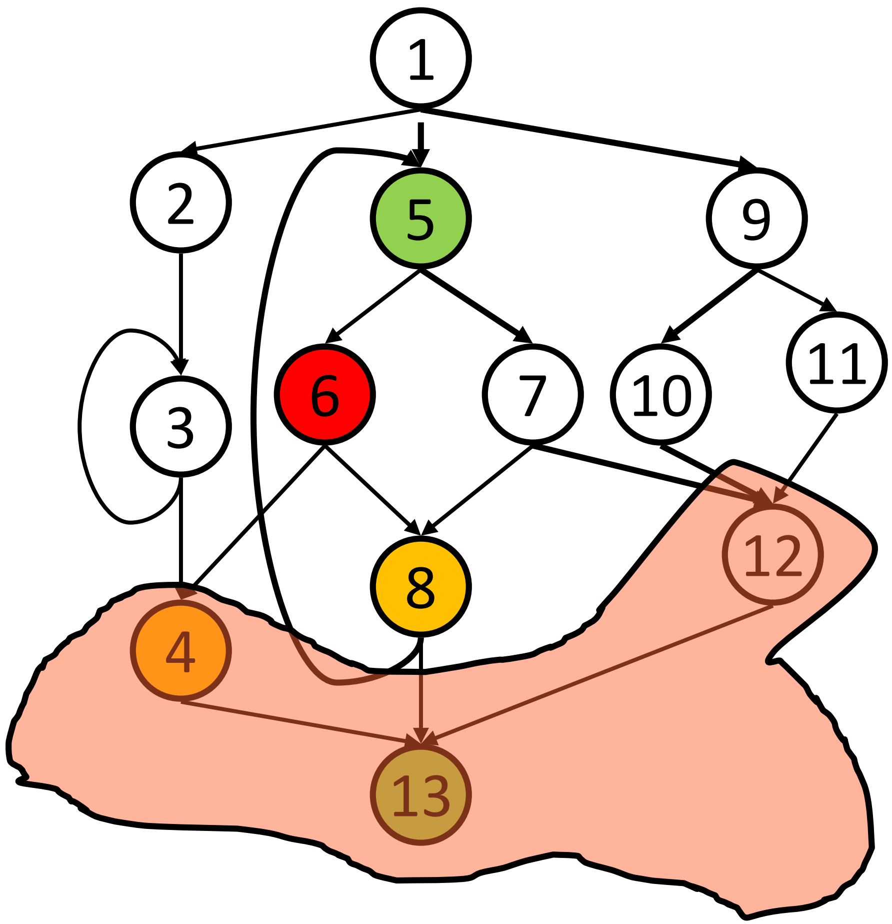
And, Iterating

Dominance Frontier Criterion



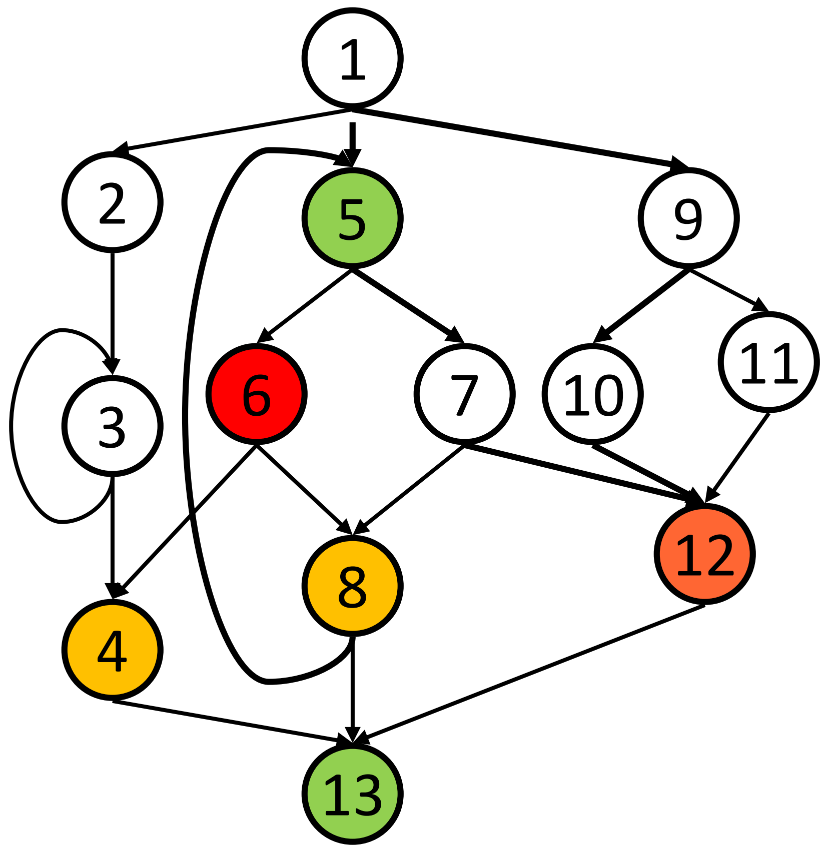
And, Iterating

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 $\Phi()$

- 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] ∪= {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:

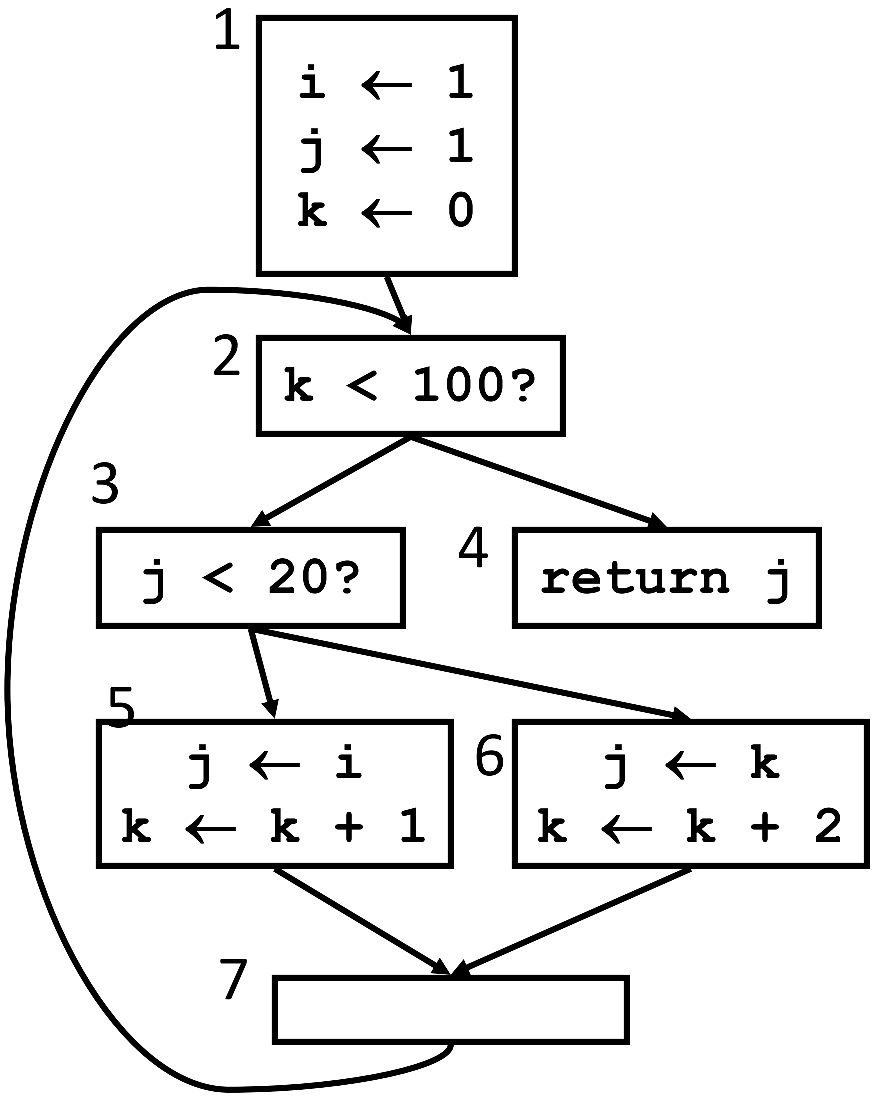
      rename(X)

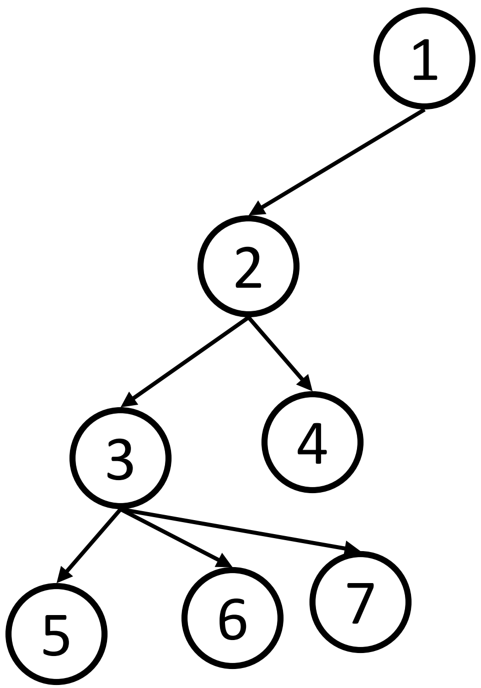
    for each instruction S ∈ n:

      for each variable v that S defines:

        pop Stack[v]

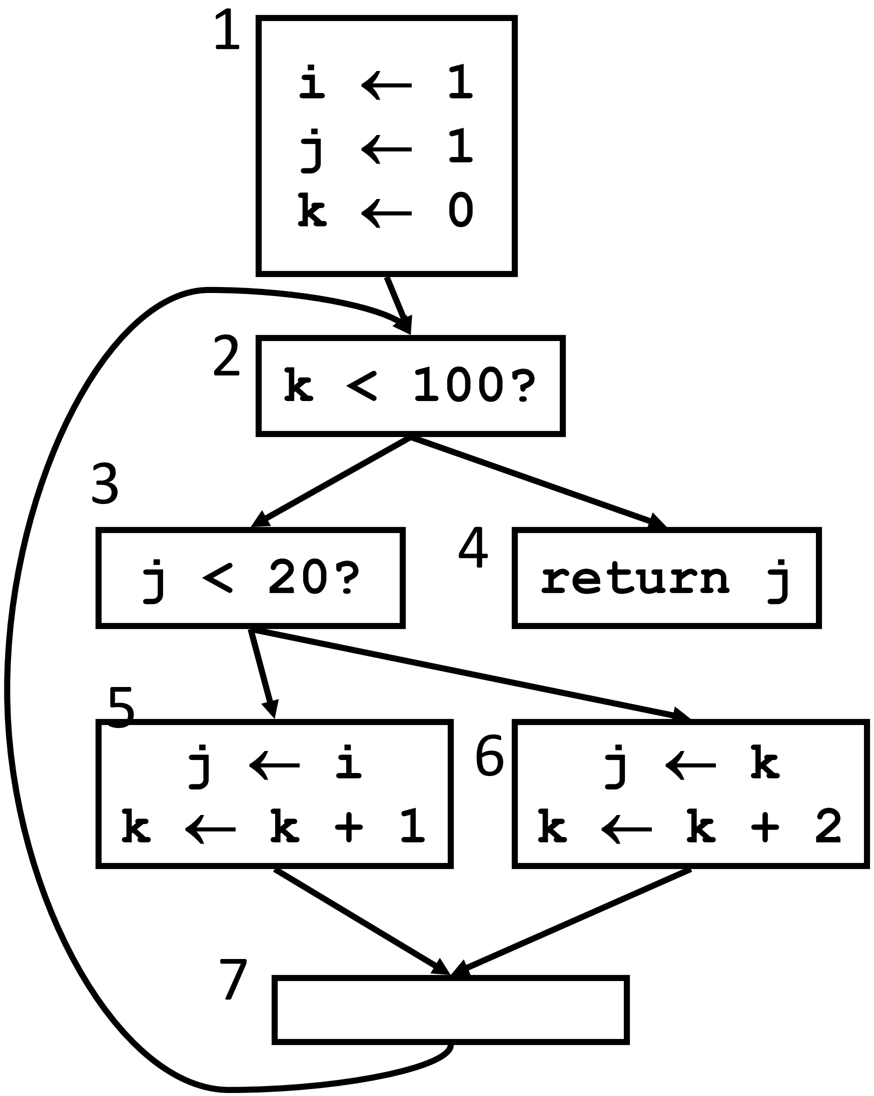
Compute D-tree

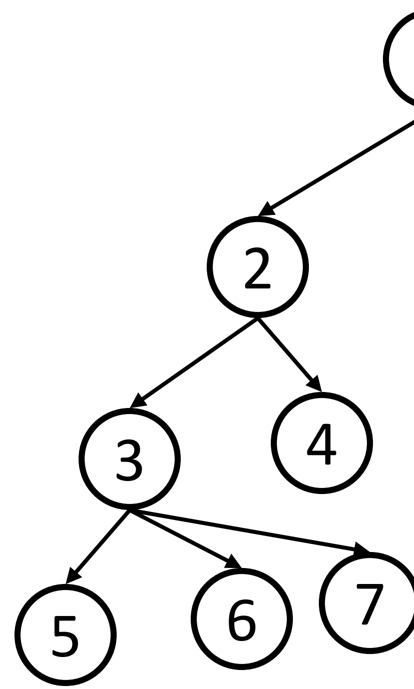




D-tree

Compute Dominance Frontier

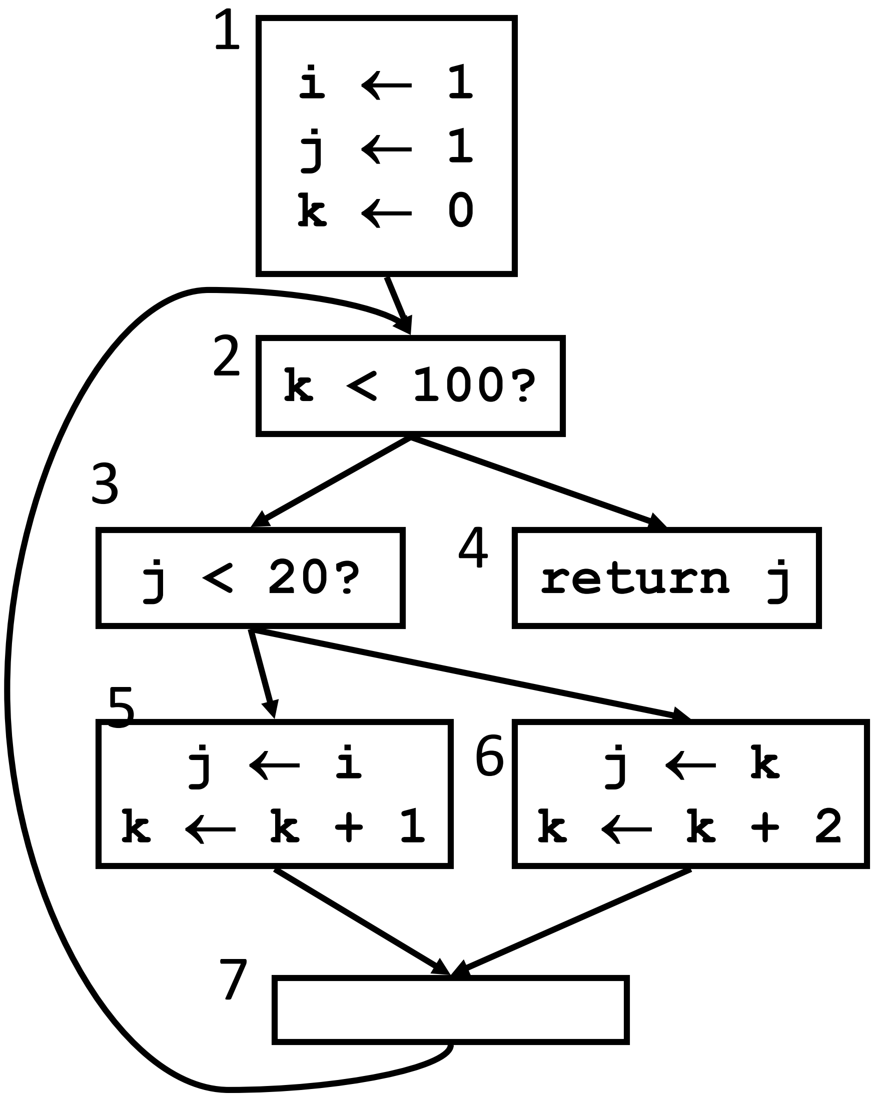




1	{}
2	{2}
3	{2}
4	{}
5	{7}
6	{7}
7	{2}

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	{}

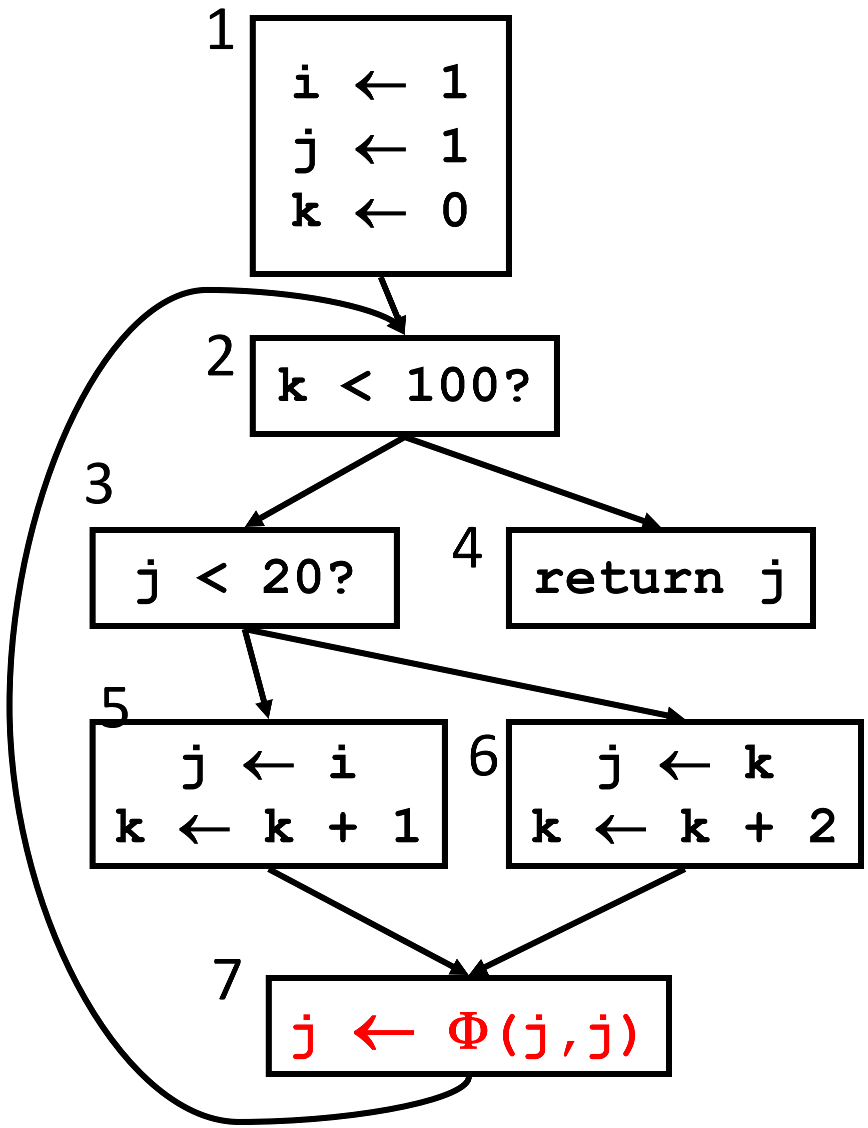
	defsites[v]
i	{1}
j	{1,5,6}
k	{1,5,6}

var i: W={1}

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

DF{1}, DF{5}

Insert Φ()



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

DFs

orig[n]

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

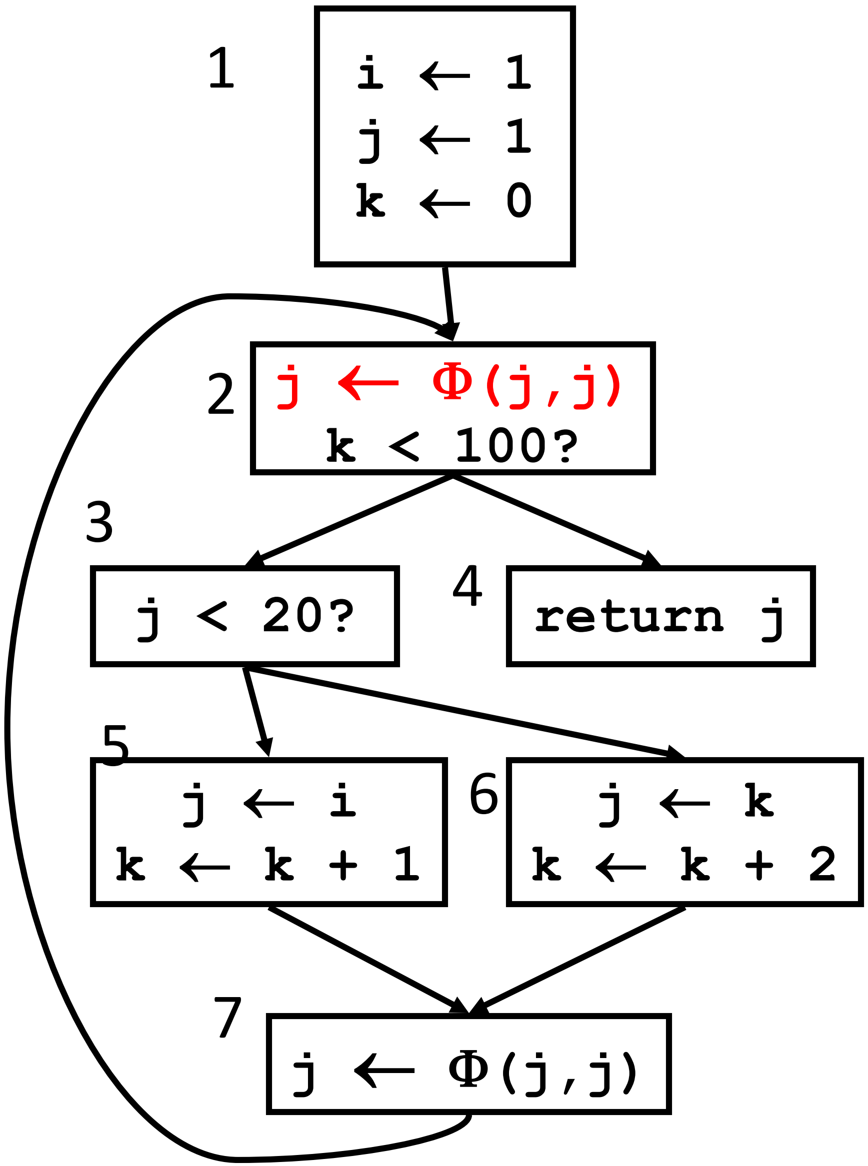
defsites[v]

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}

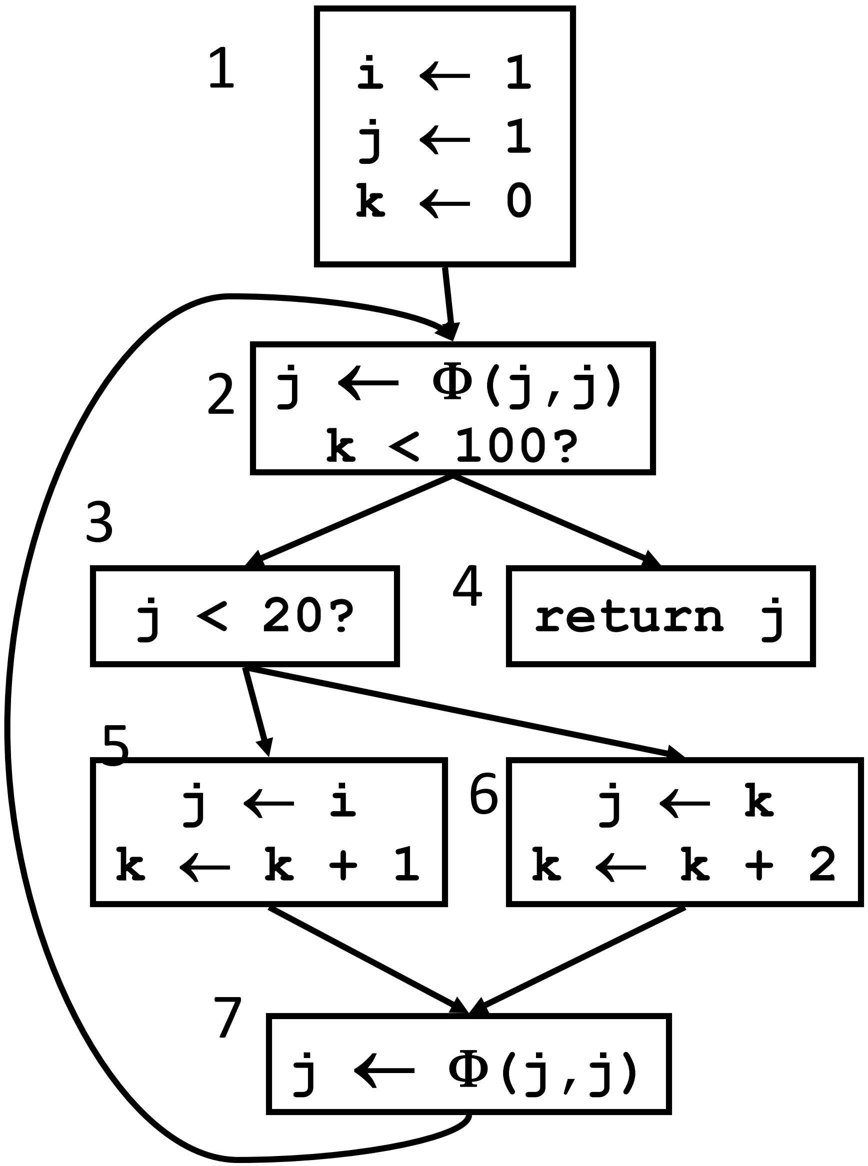
	orig[n]
1	{ i,j,k}
2	{}
3	{}
4	{}
5	{j,k}
6	{j,k}
7	{}

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

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

DF{1}, DF{5}

Insert Φ()



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

DFs

orig[n]

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

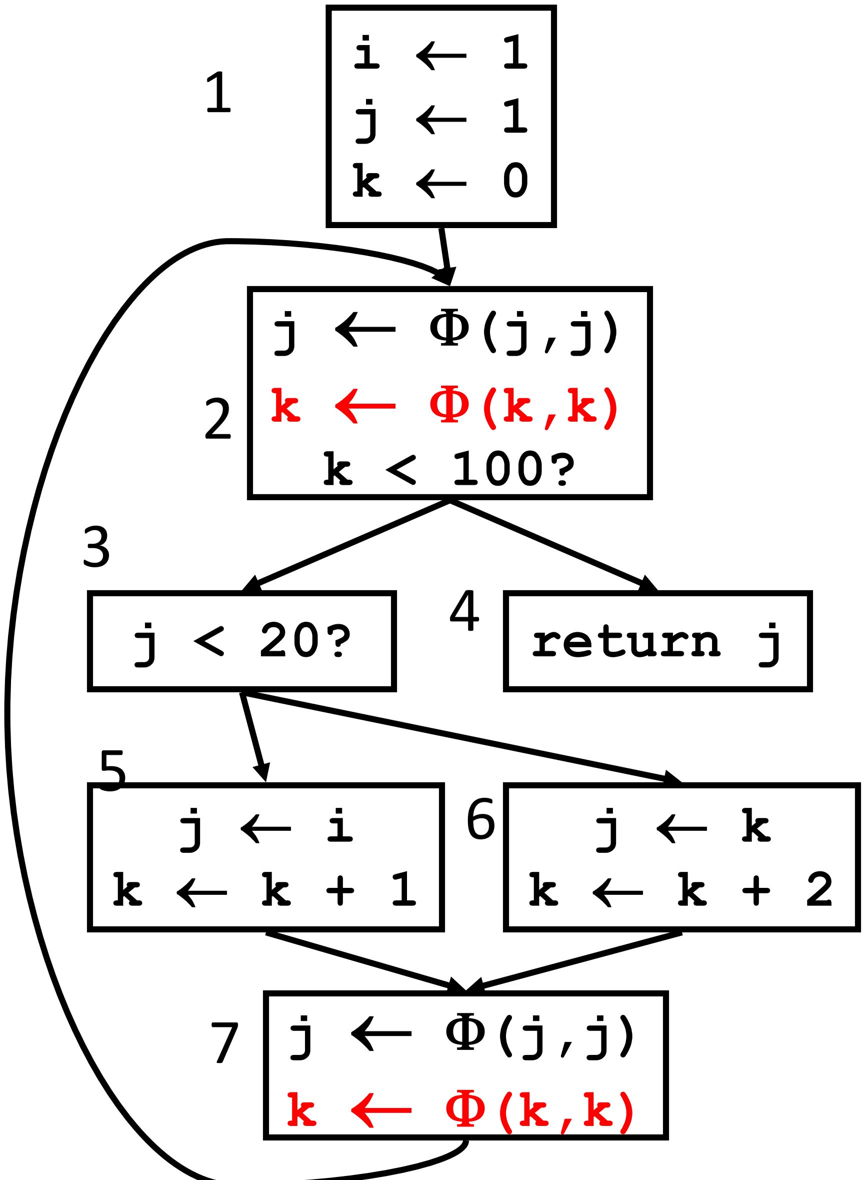
defsites[v]

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

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

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

Insert Φ()



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

DFs

orig[n]

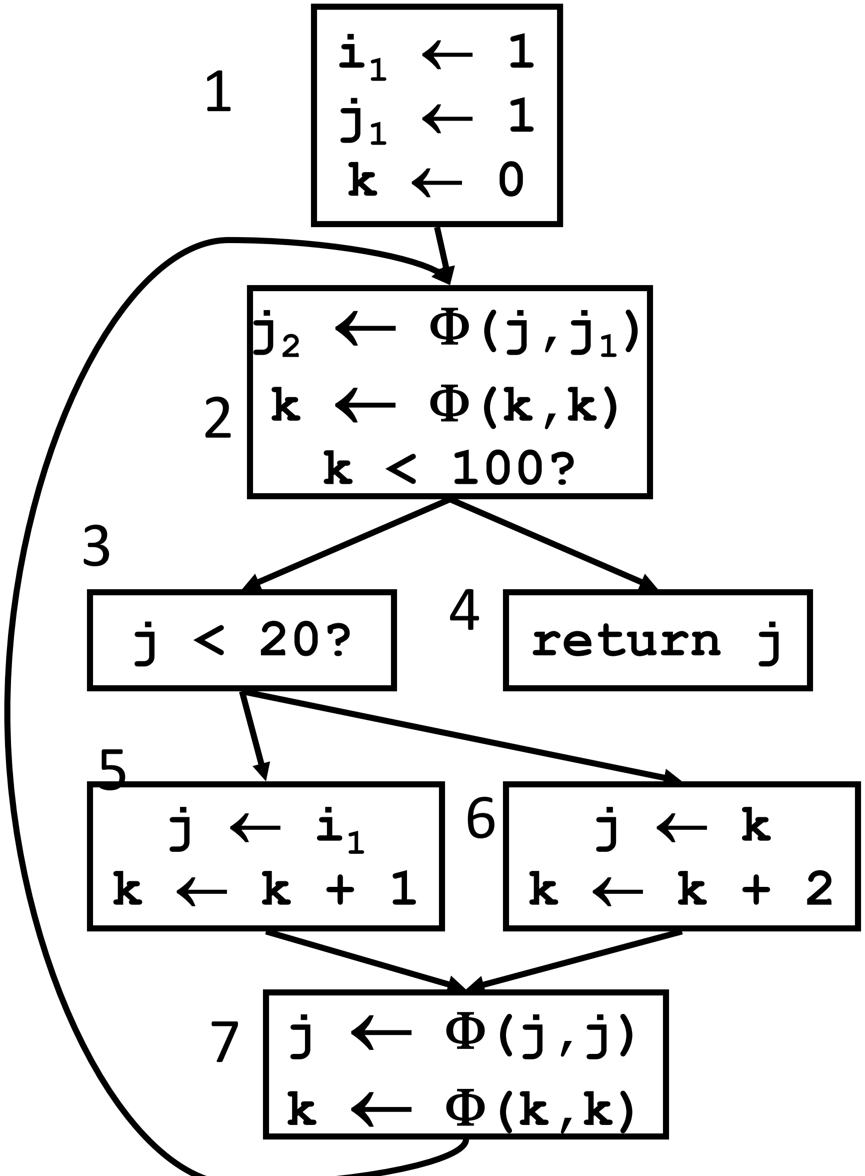
1	{ i,j,k}
2	{}
3	{}
4	{}
5	{j,k}
6	{j,k}
7	{}

defsites[v]

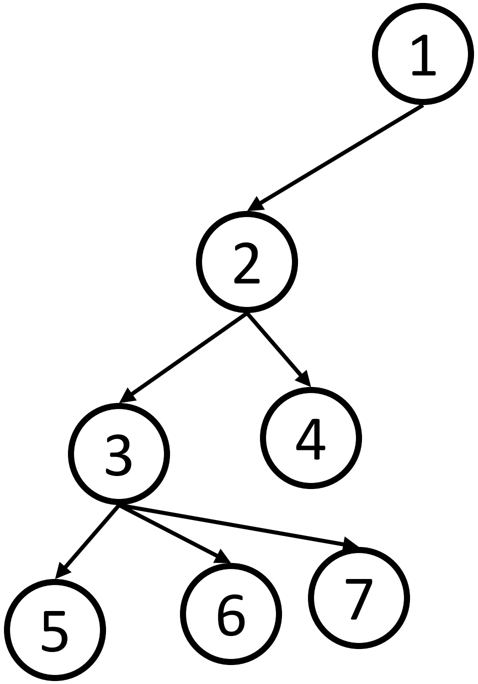
i	{1}
j	{1,5,6}
k	{1,5,6}

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

Rename Vars

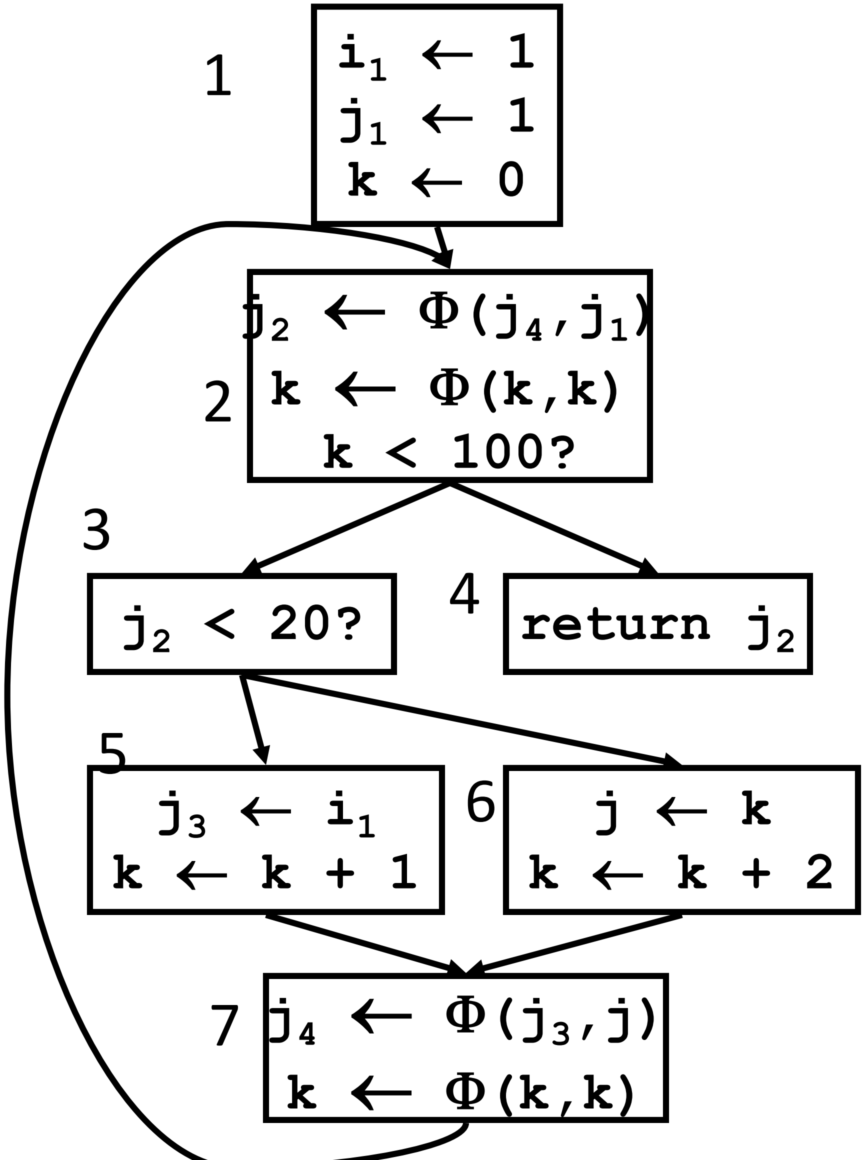


1
i1 ← 1
j1 ← 1
k ← 0
2
j2 ← Φ(j,j1)
k ← Φ(k,k)
k < 100?
3
j < 20?
4
return j
5
j ← i1
k ← k + 1
6
j ← k
k ← k + 2
7
j ← Φ(j,j)
k ← Φ(k,k)

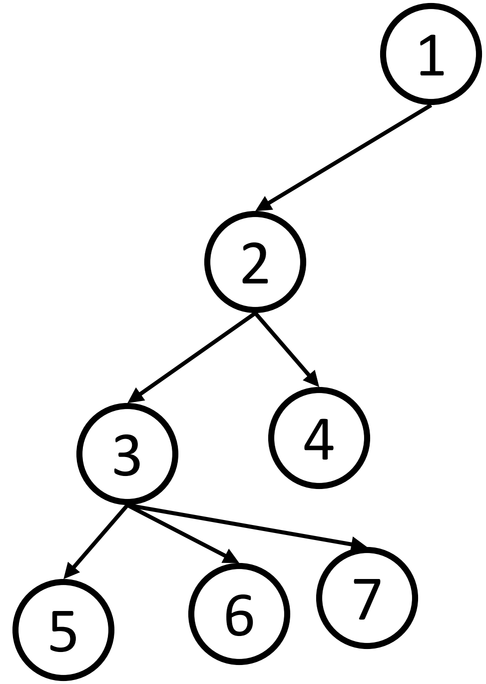


1
2
3
4
5
6
7

Rename Vars

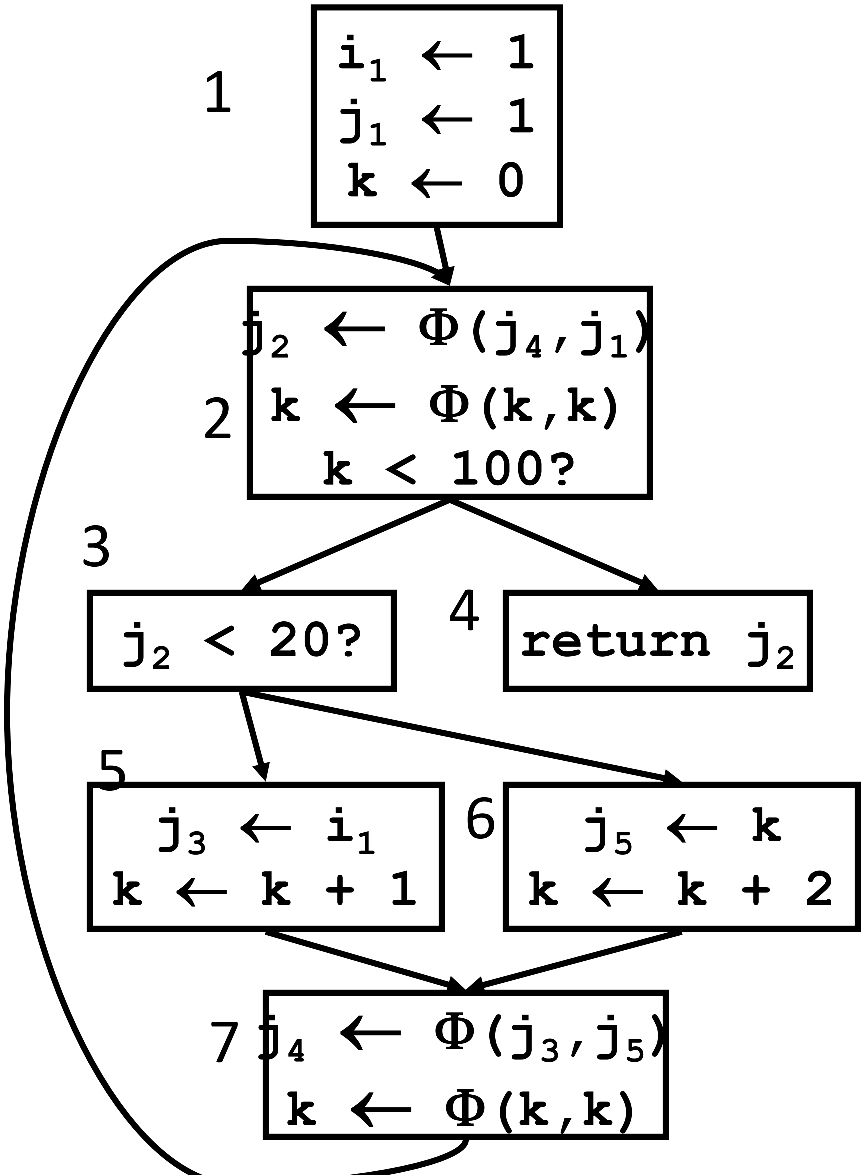


1
$i_1 \leftarrow 1$
$j_1 \leftarrow 1$
$k \leftarrow 0$
2
$j_2 \leftarrow \Phi(j_4,j_1)$
$k \leftarrow \Phi(k,k)$
k < 100?
3
j_2 < 20?
4
return j_2
5
$j_3 \leftarrow i_1$
$k \leftarrow k + 1$
6
$j \leftarrow k$
$k \leftarrow k + 2$
7
$j_4 \leftarrow \Phi(j_3,j)$
$k \leftarrow \Phi(k,k)$

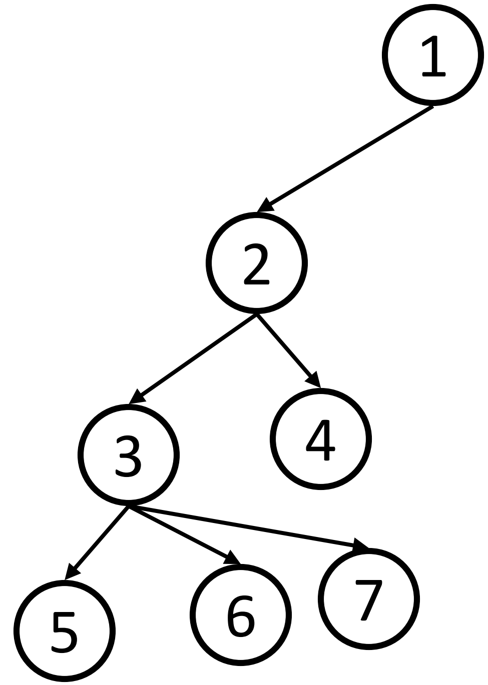


1
2
3
4
5
6
7

Rename Vars

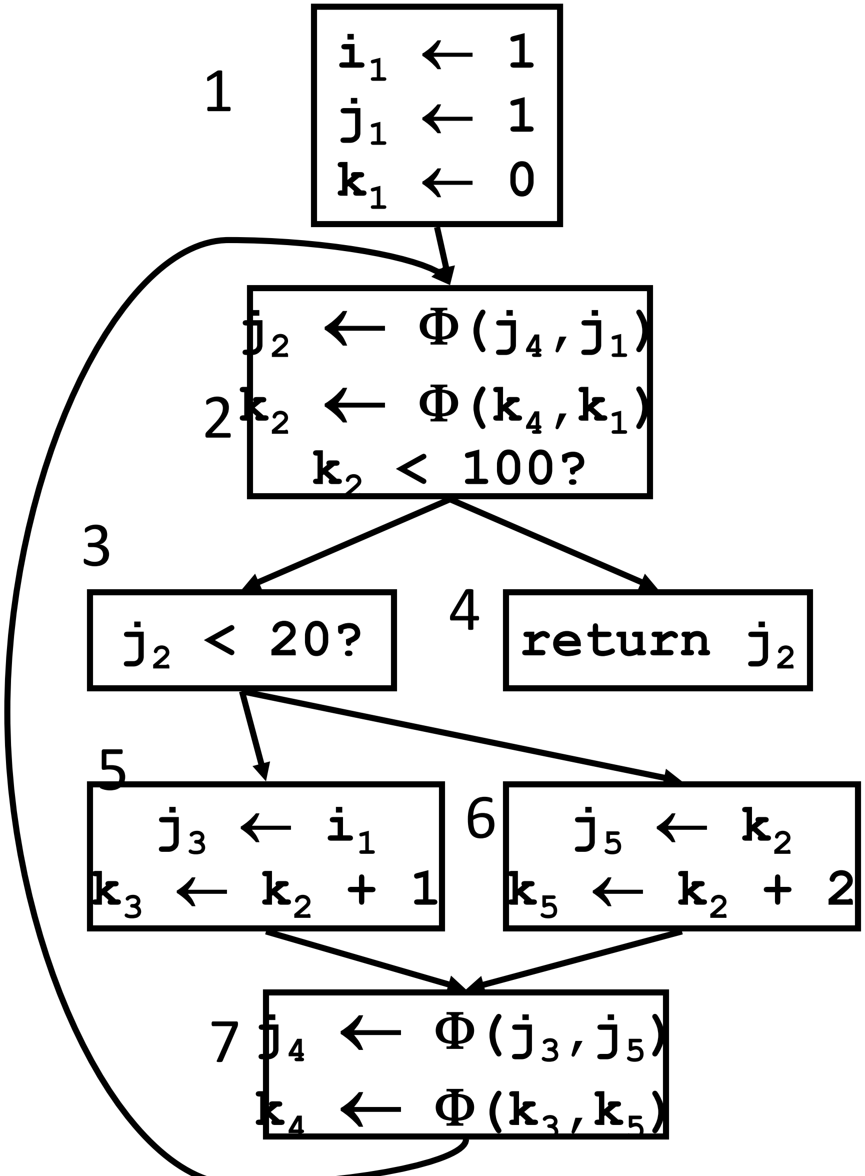


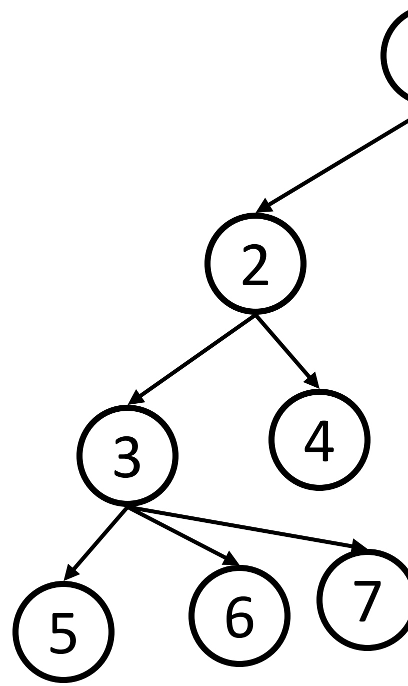
1
$i_1 \leftarrow 1$
$j_1 \leftarrow 1$
$k \leftarrow 0$
2
$j_2 \leftarrow \Phi(j_4, j_1)$
$k \leftarrow \Phi(k, k)$
$k < 100?$
3
$j_2 < 20?$
4
return j_2
5
$j_3 \leftarrow i_1$
$k \leftarrow k + 1$
6
$j_5 \leftarrow k$
$k \leftarrow k + 2$
7
$j_4 \leftarrow \Phi(j_3, j_5)$
$k \leftarrow \Phi(k, k)$



1
2
3
4
5
6
7

Rename Vars


1
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
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

Deconstructing SSA

- Real machines don't have Φ-functions
- Implement with moves (and swaps) to predecessors.

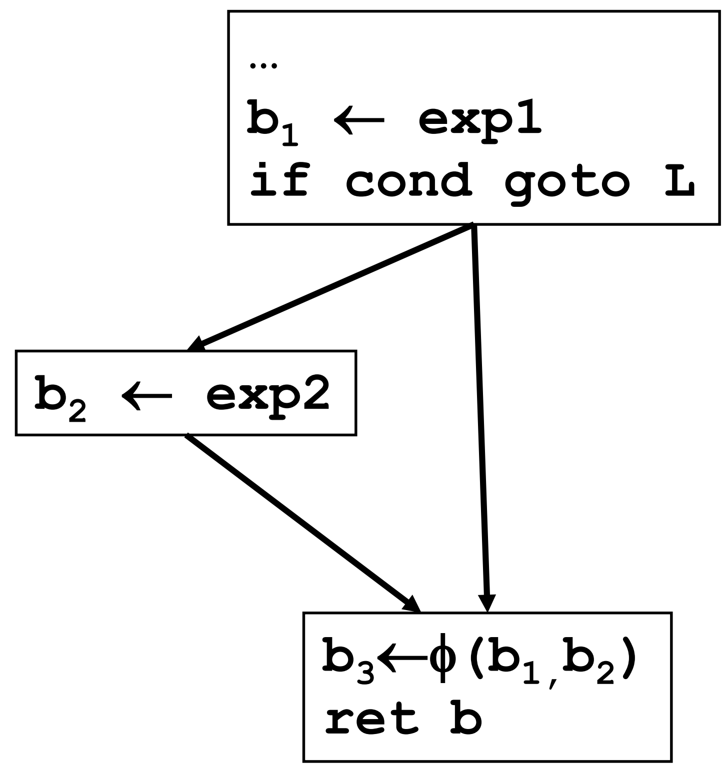
$a_1 \leftarrow x + y$
$b_1 \leftarrow a_1 + x$
$a_3 \leftarrow a_1$
$c_3 \leftarrow c_1$

$a_2 \leftarrow b + 2$
$c_2 \leftarrow y + 1$
$a_3 \leftarrow a_2$
$c_3 \leftarrow c_2$

$\cancel{a_3 \leftarrow \Phi(a_1, a_2)}$
$\cancel{c_3 \leftarrow \Phi(c_1, c_2)}$
$a_4 \leftarrow c_3 + a_3$

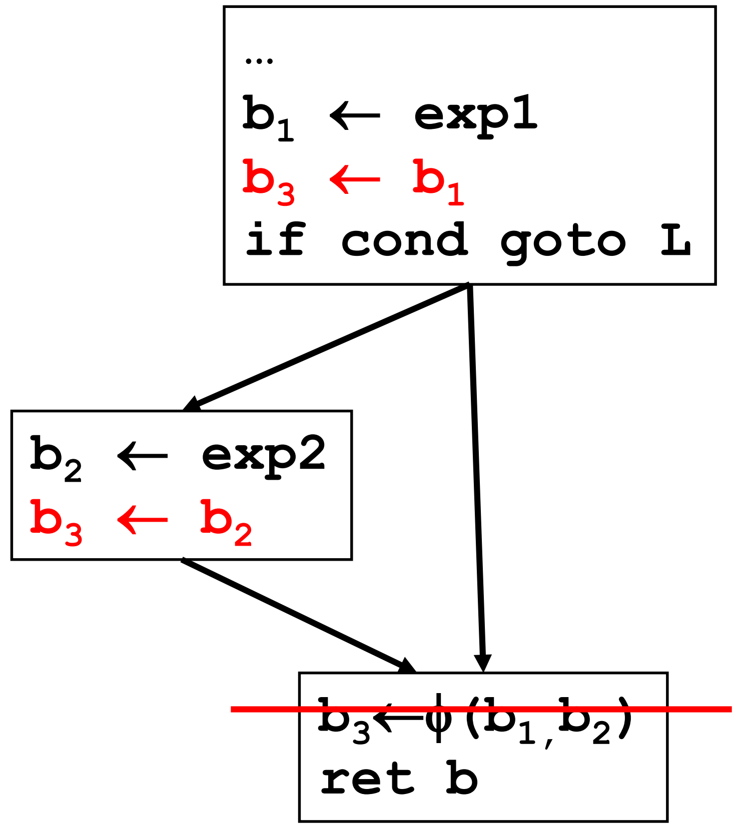
Deconstructing SSA

- Real machines don't have Φ-functions
- Implement with moves (and swaps) to predecessors.
- Issue 1: critical edges



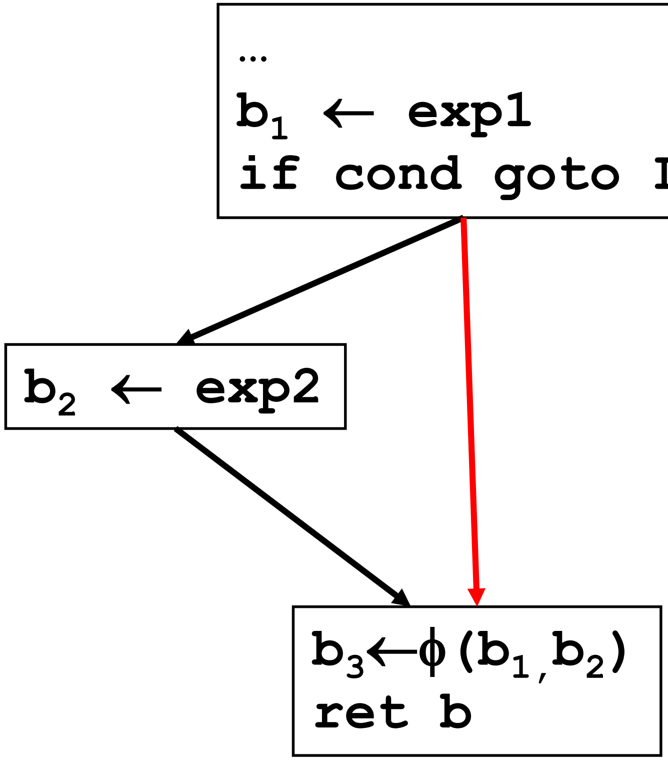
Deconstructing SSA

- Real machines don't have Φ-functions
- Implement with moves (and swaps) to predecessors.
- Issue 1: critical edges
  – unnecc assignment
  – So, first remove critical edges



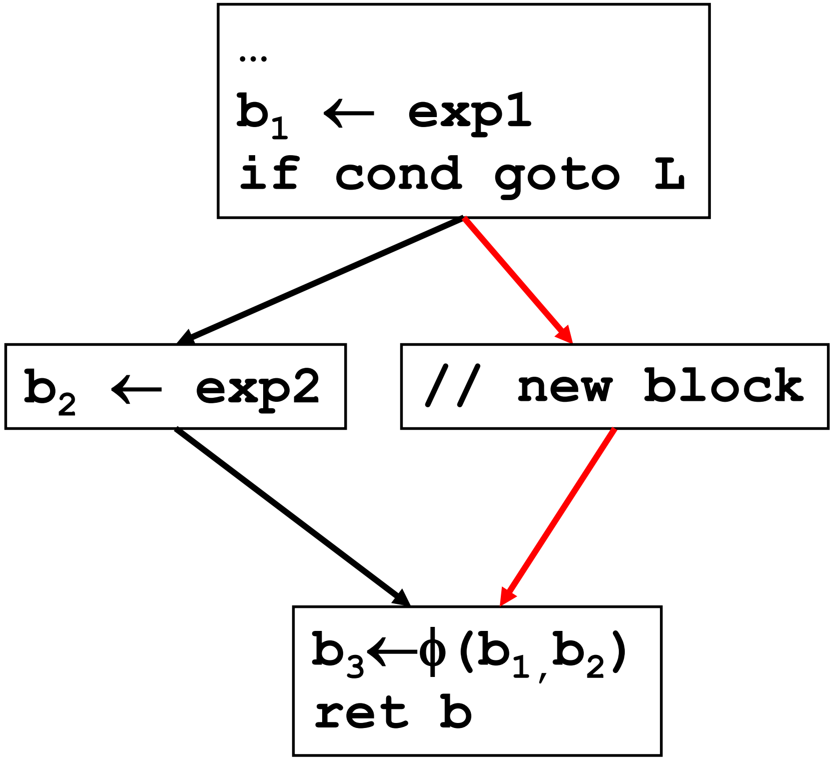
Removing a Critical Edge

- A critical edge is an edge from a to b when 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

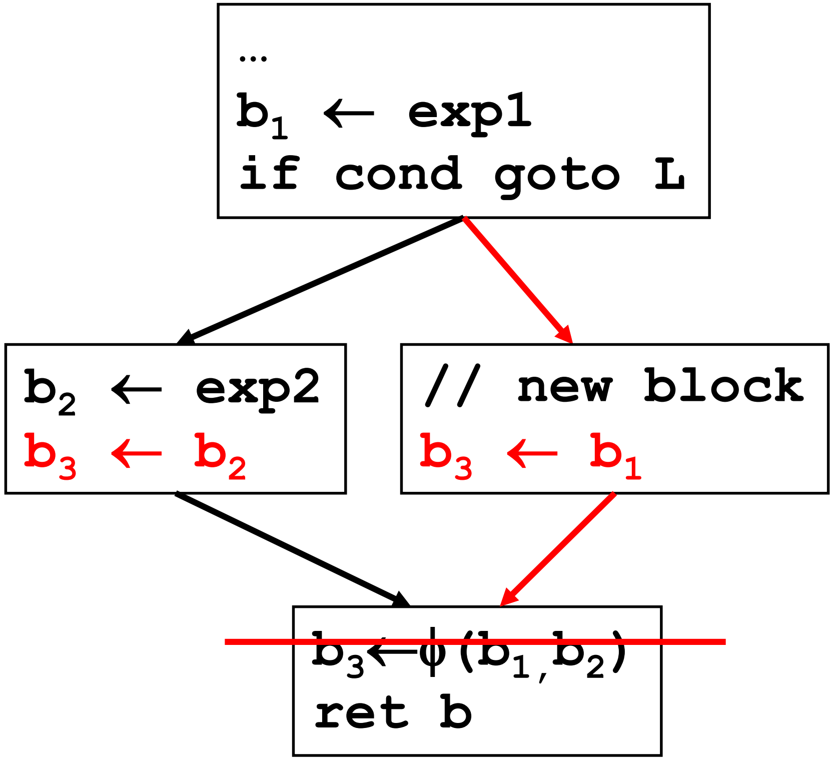
- A critical edge is an edge from a to b when a has > 1 successor and b has > 1 predecessor.
- For each edge (a,b) in CFG where a > 1 succ and
  b > 1 pred
  - Insert new block Z
  - replace (a,b) with
    - (a,z) and (z,b)



Replacing Φ

- Insert **`b`** ← **`b`**$_i$ in predecessor block for each **`b`**$_i$ in Φ
- remove Φ

- There are still some issues, but only if certain optimizations are performed.



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!