15-745

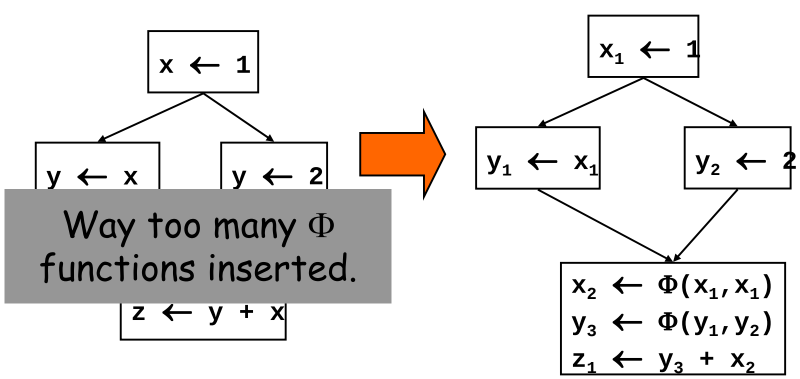
SSA
Dominators

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.
  $x_{new} \leftarrow \Phi(x_1, x_1, x_1, \ldots, x_p)$
- How do we choose which x_i to use?
  - Most compiler writers don't really care!
  - If we care, use moves on each incoming edge (Or, as in pegasus use a mux)

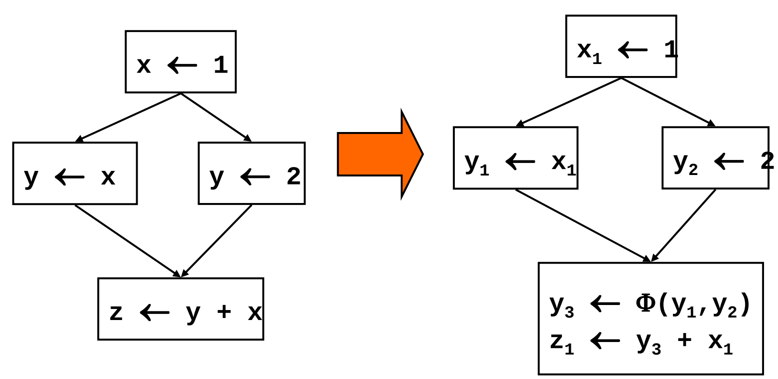
Trivial SSA

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

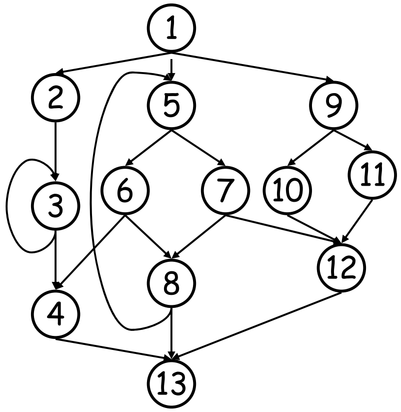


Minimal SSA

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



When do we insert Φ?



CFG

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

Note: a is implicitly defined in block 1

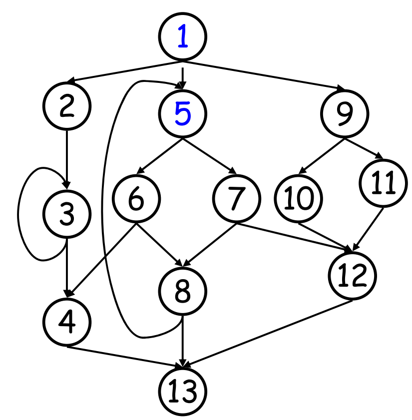
When do we insert Φ?

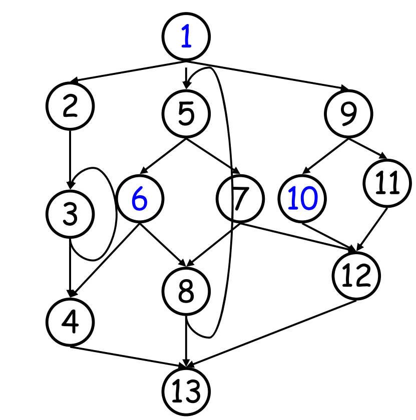
- Insert a Φ function for variable A in block Z iff:
  - A was defined more than once before (i.e., A defined in X and Y AND $X \neq Y$)
  - Z is the first block that joins the paths from X to Z and Y to Z
- Entry block implicitly defines of all vars
- Note: A = Φ(…) is a def of A

When do we insert Φ?

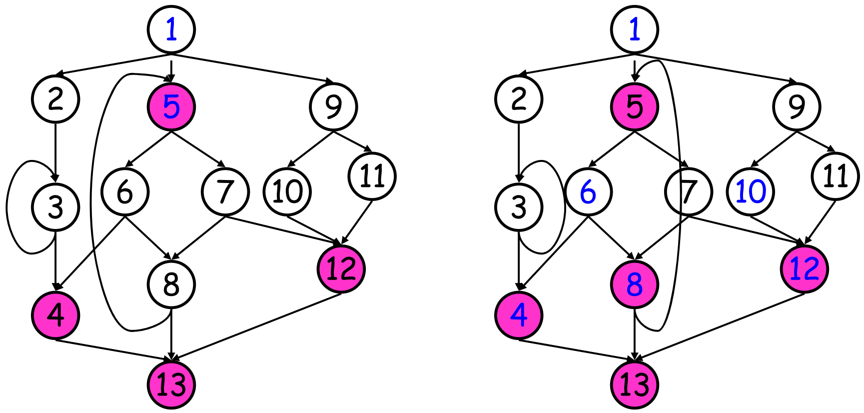
- Insert a Φ function for variable A in block Z iff:
  - A was defined more than once before (i.e., A defined in X and Y AND $X \neq Y$)
  - There exists a non-empty path from x to z, P_{xz}, and a non-empty from y to z, P_{yz} s.t.
    - $P_{xz} \cap P_{yz} = \{ z \}$
    - $z \notin P_{xq}$ or $z \notin P_{yr}$ where $P_{xz} = P_{xq} \rightarrow z$ and $P_{yz} = P_{yr} \rightarrow z$
- Entry block implicitly defines all vars
- Note: A = Φ(…) is a def of A

When do we insert Φ?

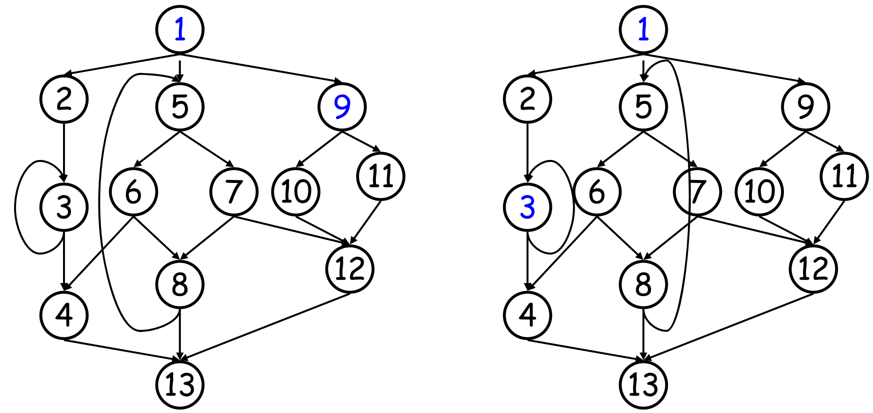




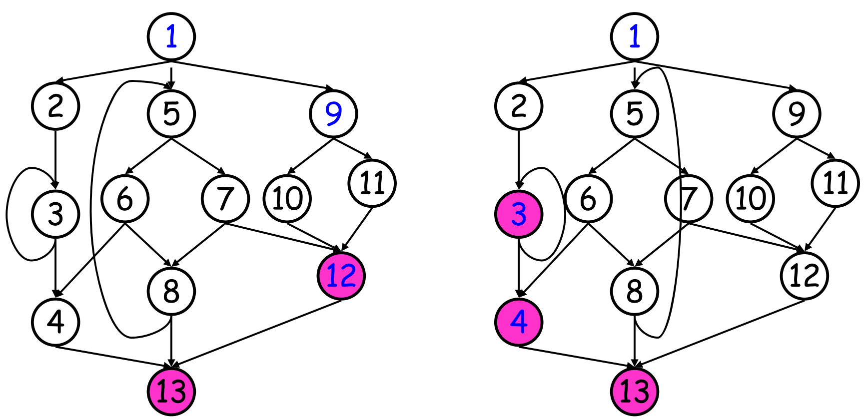
When do we insert Φ?



When do we insert Φ?

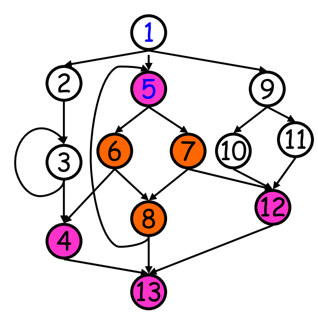


When do we insert Φ?



Def-use property of SSA

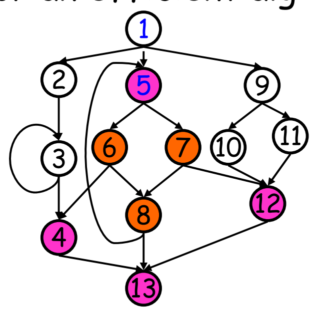
- If x_i is used in $x \leftarrow \Phi(\ldots, x_i, \ldots)$, then NO BBs in any path from BB(x_i) to BB(Φ) include def of x except BB(X_i) and BB(Φ)
- If x is used in $y \leftarrow \ldots x \ldots$, then no BBs in path from BB(x) to BB(y) define x except BB(x)



Another way to say this:
Definitions dominate uses

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)
- Use this for an efficient alg to convert to SSA

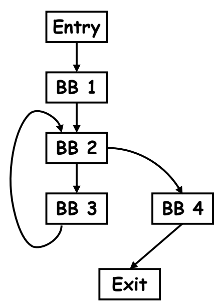


A little side trip

- Computing dominators
- d dom n iff every path from s to n goes through d
- n dom n for all n
- Some definitions:
  - immediate dominator: d idom n iff
    - $d \neq n$
    - d dom n
    - d doesn't dominate any other dominator of n
  - strictly dominates: s sdom n iff
    - s dom n
    - $s \neq n$

Examples

- d dom n iff every path from Entry to n contains d.
  1 dom 1 ; 1 dom 2 ; 1 dom 3 ; 1 dom 4 ;
  2 dom 2 ; 2 dom 3 ; 2 dom 4 ; 3 dom 3 ;
  4 dom 4



- s strictly dominates n, (s sdom n), iff s dom n and $s \neq n$.
  1 sdom 2 ; 1 sdom 3 ; 1 sdom 4 ;
  2 sdom 3 ; 2 sdom 4
- d immediately dominates n, d=idom(n), iff d sdom n and there is no node x such that d dom x and x dom n.
  1 idom 2 ; 2 idom 3; 2 idom 4

(adapted from: http://www.eecg.toronto.edu/~voss/ece540/

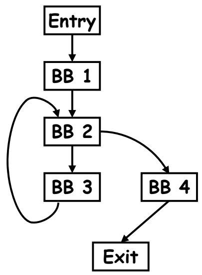
Properties of dominators

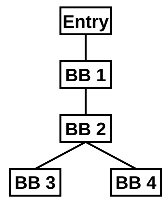
- idom(n) is unique
- The dominance relation is a partial ordering; that is, it is reflexive, anti-symmetric and transitive:
  - reflexive:
    x dom x
  - anti-symmetric:
    x dom y and y dom x $\rightarrow$ x = y
  - transitive :
    x dom y and y dom z $\rightarrow$ x dom z

(adapted from: http://www.eecg.toronto.edu/~voss/ece540/

The dominator tree

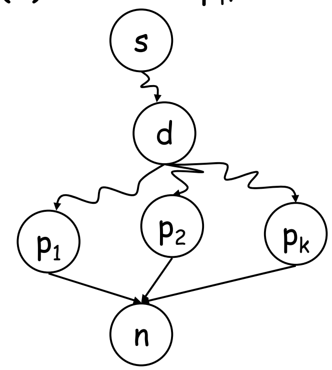
- One can represent dominators in a cfg as a tree of immediate dominators.
- In dominator tree, edge from parent to child if parent idom child in the cfg
- The set of dominators of a node are the nodes from the root to the node.





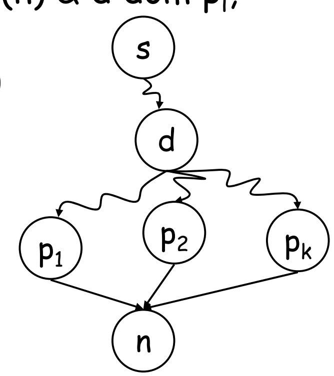
Computing Dominators

- d dom n iff every path from s to n goes through d
- Note: n dom n for all n
- If s dom d & d $\neq$ n & $p_i \in$ pred(n) & d dom p_i, then d dom n
-
- How can we use this?



Computing Dominators

- d dom n iff every path from s to n goes through d
- Note: n dom n for all n
- If s dom d & d $\neq$ n & $p_i \in$ pred(n) & d dom p_i, then d dom n
- $dom(n) = \{n\} \bigcup \bigcap_{p \in pred(n)} dom(p)$

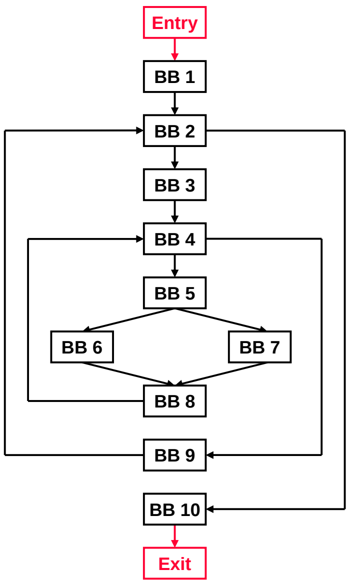


Simple iterative alg

- dom(Entry) = Entry
  for all other nodes, n, dom(n) = all nodes
  changed = true
  while (changed) {
      changed = false
      for each n, n$\neq$Entry {
          old = dom(n)
          $dom(n) = \{n\} \bigcup \bigcap_{p \in pred(n)} dom(p)$
          if (dom(n) != old) changed = true
      }
  }

(adapted from: http://www.eecg.toronto.edu/~voss/ece540/

Example



DOM (Entry) = {Entry}
DOM (1) = {Entry,1}
DOM (2) = {Entry,1,2}
DOM (3) = {Entry,1,2,3}
DOM (4) = {Entry,1,2,3,4}
DOM (5) = {Entry,1,2,3,4,5}
DOM (6) = {Entry,1,2,3,4,5,6}
DOM (7) = {Entry,1,2,3,4,5,7}
DOM (8) = {Entry,1,2,3,4,5,8}
DOM (9) = {Entry,1,2,3,4,9}
DOM (10) = {Entry,1,2,10)

(Borrowed from: http://www.eecg.toronto.edu/~voss/ece540/

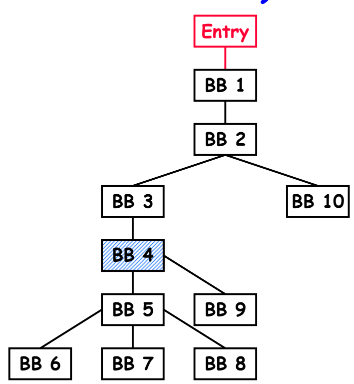
Finding immediate dominators

- idom(n) dominates n, isn't n, and, doesn't strictly dominate any other sdom n
- Init idom(n) to nodes which sdom n
- foreach x ∈ idom(n)
  foreach y ∈ idom(n) – {x}
    if (y ∈ sdom(x)) idom(n)=idom(n)-{y}

Example (immediate dominators)

DOM_d (1) = {Entry}
DOM_d (2) = {Entry,1}
DOM_d (3) = {Entry,1,2}
DOM_d (4) = {Entry,1,2,3}
DOM_d (5) = {Entry,1,2,3,4}
DOM_d (6) = {Entry,1,2,3,4,5}
DOM_d (7) = {Entry,1,2,3,4,5}
DOM_d (8) = {Entry,1,2,3,4,5}
DOM_d (9) = {Entry,1,2,3,4}
DOM_d (10) = {Entry,1,2)

DOM_d (1) = {Entry}
DOM_d (2) = {1}
DOM_d (3) = {2}
DOM_d (4) = {3}
DOM_d (5) = {4}
DOM_d (6) = {5}
DOM_d (7) = {5}
DOM_d (8) = {5}
DOM_d (9) = {4}
DOM_d (10) = {2}



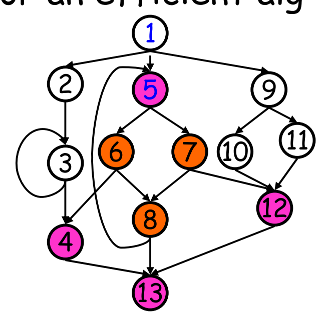
Entry: {1,2,3} ⇒ {Entry,1,2,3}

1: {Entry,2,3} ⇒ {1,2,3}
2: {1,3} ⇒ {2,3}
3: {2} ⇒ {3}

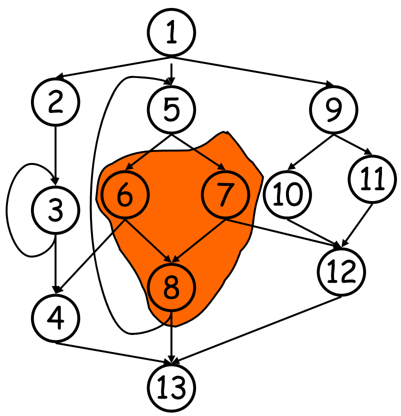
(Borrowed from: http://www.eecg.toronto.edu/~voss/ece540/

Dominance Property of SSA

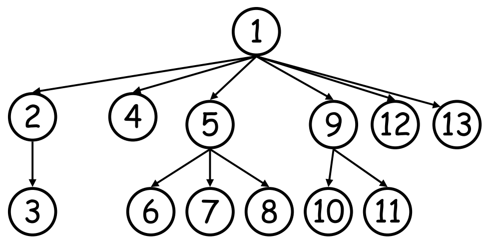
- In SSA definitions dominate uses.
  - If x_i is used in $x \leftarrow \Phi(\ldots, x_i, \ldots)$, then $BB(x_i)$ dominates ith pred of BB(PHI)
  - If x is used in y ← … x …, then BB(x) dominates BB(y)
- Use this for an efficient alg to convert to SSA



Dominance



CFG

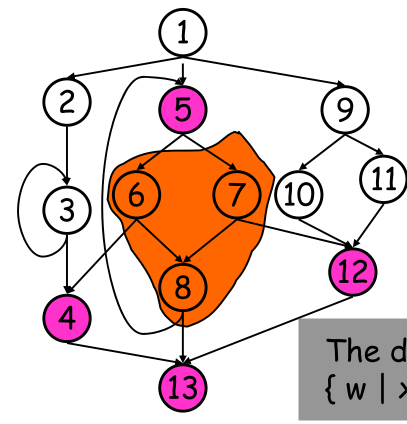


D-Tree

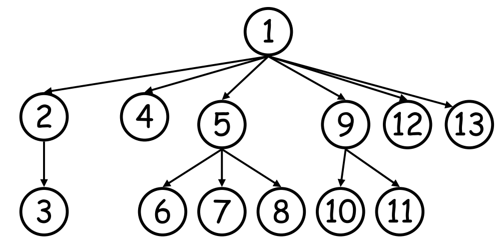
If there is a def of a in block 5, which nodes need a $\Phi()$?

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

Dominance Frontier



CFG

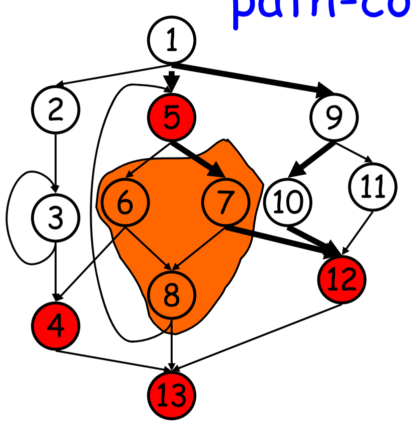


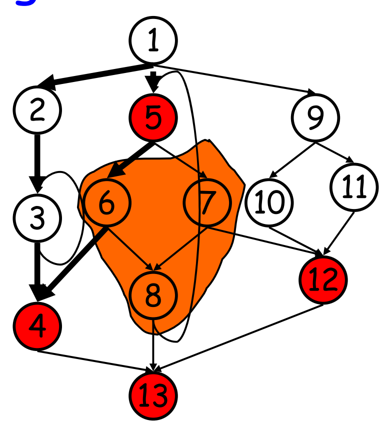
D-Tree

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

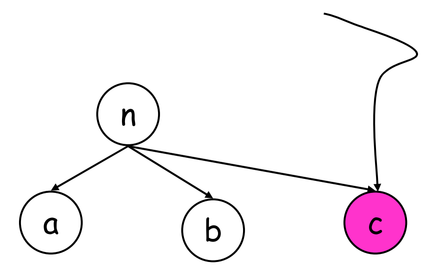
x strictly dominates w (s sdom w) iff x dom w AND x $\neq$ w

Dominance Frontier & path-convergence





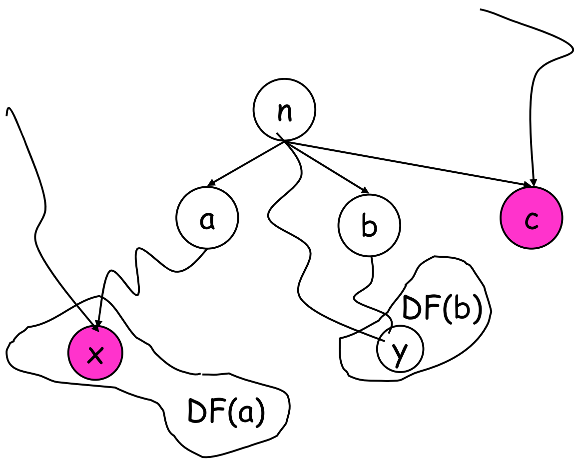
Computing DF(n)



c is an example of the successors of n not strictly dominated by n

n idom a
n idom b
!n idom c

Computing DF(n)



n idom a
n idom b
!n dom c

x is in DF[a] and !(n dom x)

Computing the Dominance Frontier

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

```
compute-DF(n)
    S = {}
    foreach node y in succ[n]
        if idom(y) ≠ n
            S = S ∪ { y }
    foreach child of n, c, in D-tree
        compute-DF(c)
        foreach w in DF[c]
            if !n sdom w
                S = S ∪ { w }
    DF[n] = S
```


Using DF to compute SSA

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

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, inserting the minimal number of $\Phi()$ neccesary

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 Variables

- Walk the D-tree, renaming variables as you go
- Replace uses with most recent renamed def
  - For straight-line code this is easy
  - 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
    for each def in this block pop from stack

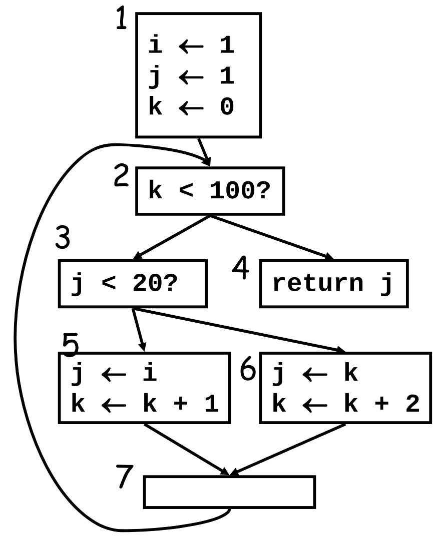
rename

```
foreach var a
    a.count = 0
    a.stack = empty
    a.stack.push(0)
rename(entry)
rename(n) {
   foreach s in block n
       if s isn't Φ
              foreach use of x in S
                     replace x with x_stack.top()
       foreach def of x in S
              i = ++x.count
              x.stack.push(i)
              replace x with x_i
```

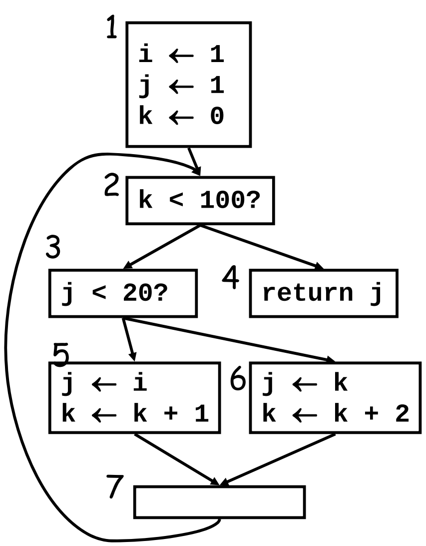
rename

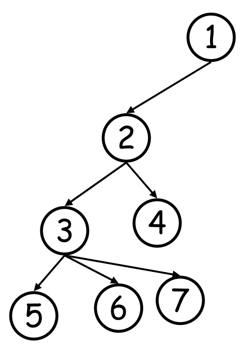
```
rename(n) {
   foreach s in block n
      if s isn't Φ
           foreach use of x in S
                replace x with x_stack.top()
      foreach def of x in S
           i = ++x.count
           x.stack.push(i)
           replace x with x_i
   foreach y ∈ succ(n)
      j = pred # of n in y
      foreach Φ in y
           i <- var-j.stack.top()
           replace var-j with var-j_i
   foreach child X of n in D-tree: rename(X)
   foreach def, x, in S: x.stack.pop()
```


Compute D-tree



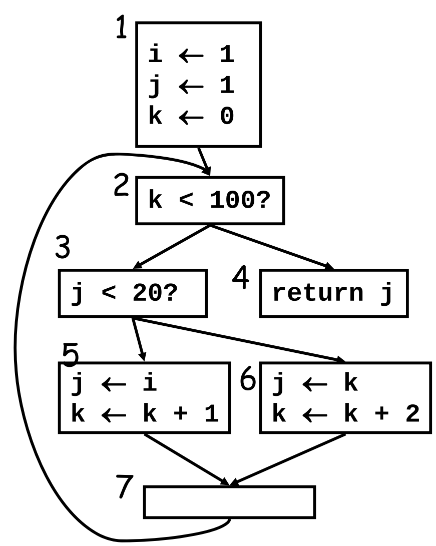
Compute D-tree

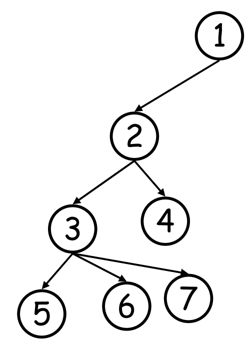




D-tree

Compute Dominance Frontier

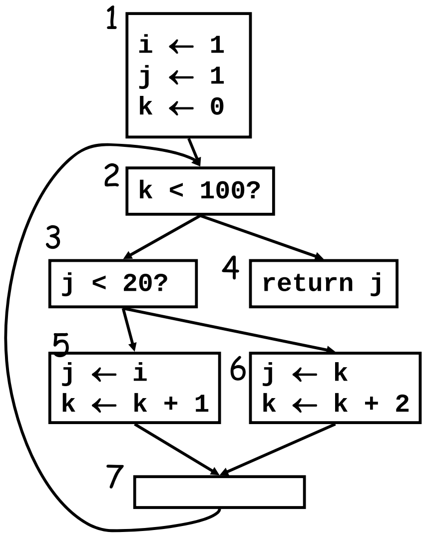




1	{}
2	{2}
3	{2}
4	{}
5	{7}
6	{7}
7	{2}

DFs

Insert Φ()



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

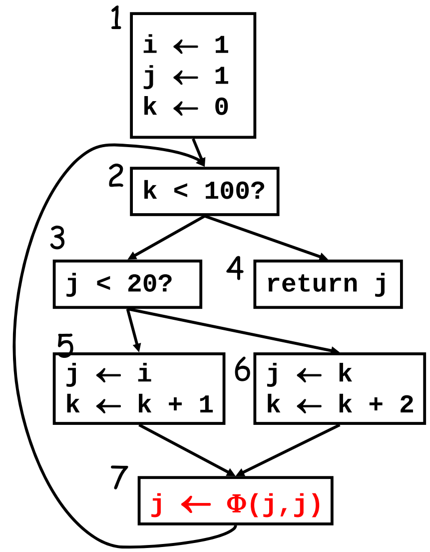
defsites[v]	
i	{1}
j	{1,5,6}
k	{1,5,6}

var i: W={1}

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

DF{1}, DF{5}

Insert Φ()



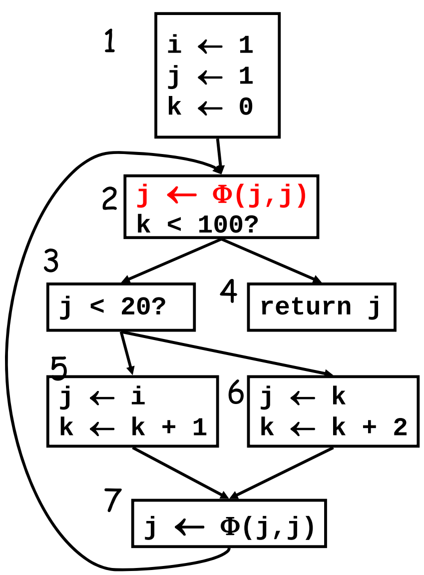
n	DFs	orig[n]
1	{}	{ i,j,k}
2	{2}	{}
3	{2}	{}
4	{}	{}
5	{7}	{j,k}
6	{7}	{j,k}
7	{2}	{}

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

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

DF{1}, DF{5}

Insert Φ()



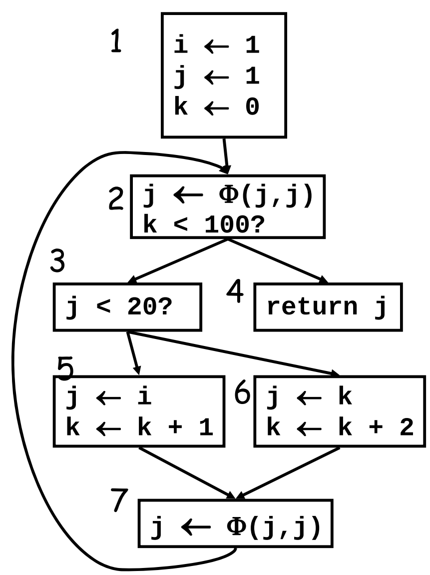
n	DFs	orig[n]
1	{}	{ i,j,k}
2	{2}	{}
3	{2}	{}
4	{}	{}
5	{7}	{j,k}
6	{7}	{j,k}
7	{2}	{}

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

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

DF{1}, DF{5}

Insert Φ()



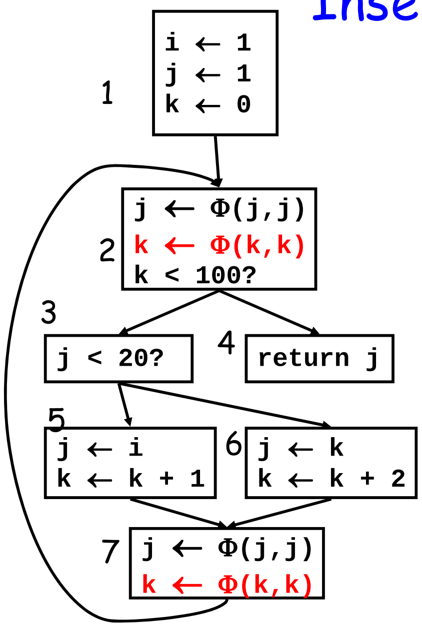
n	DFs	orig[n]
1	{}	{ i,j,k}
2	{2}	{}
3	{2}	{}
4	{}	{}
5	{7}	{j,k}
6	{7}	{j,k}
7	{2}	{}

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()



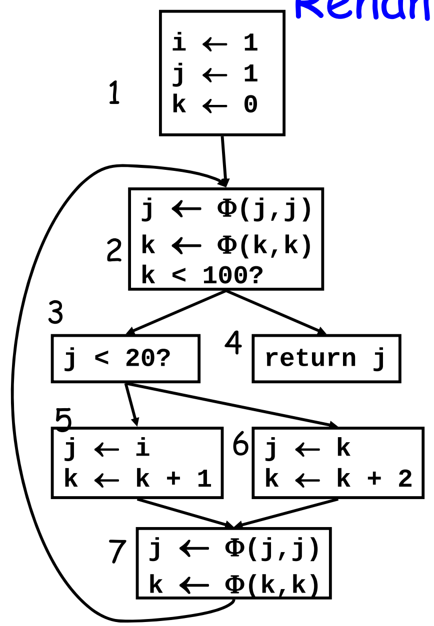
	DFs
1	{}
2	{2}
3	{2}
4	{}
5	{7}
6	{7}
7	{2}

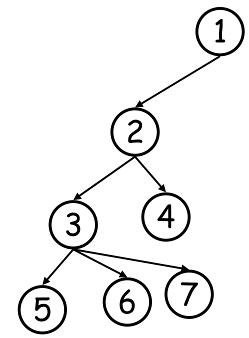
	orig[n]
1	{ i,j,k}
2	{}
3	{}
4	{}
5	{j,k}
6	{j,k}
7	{}

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

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

Rename Vars

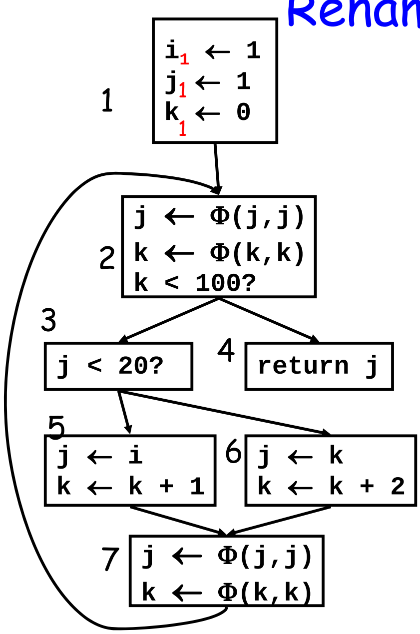


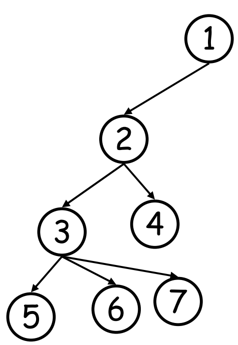


init

Var	Count	stack
i	0	0
j	0	0
k	0	0

Rename Vars

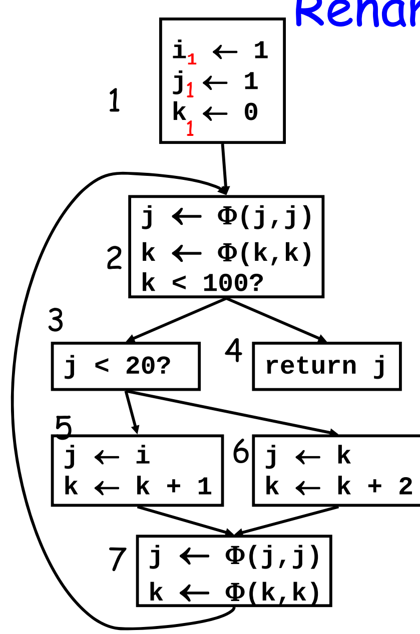


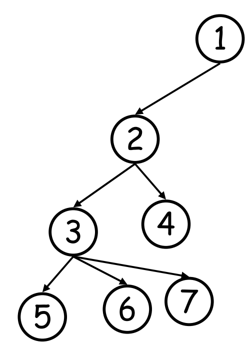


rename(1)
- defs & uses

Var	Count	stack
i	~~0~~ 1	0 1
j	~~0~~ 1	0 1
k	~~0~~ 1	0 1

Rename Vars

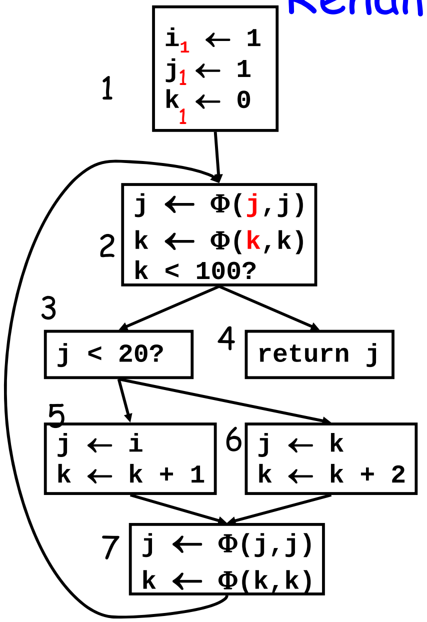


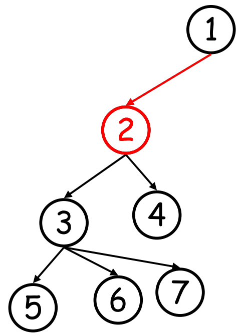


rename(1)
- defs & uses
- succs

Var	Count	stack
i	~~0~~ 1	0 1
j	~~0~~ 1	0 1
k	~~0~~ 1	0 1

Rename Vars

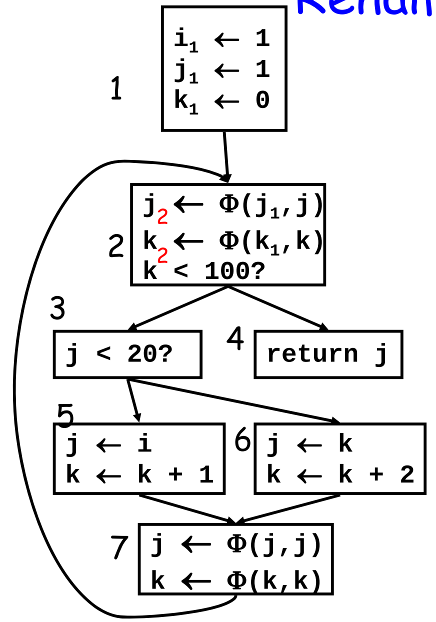


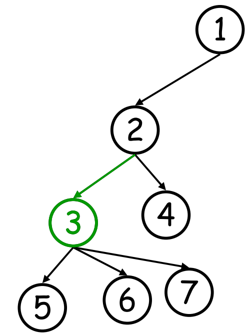


rename(1)
- defs & uses
- succs
- rename(2)

Var	Count	stack
i	~~0~~1	0 1
j	~~0~~1	0 1
k	~~0~~1	0 1

Rename Vars

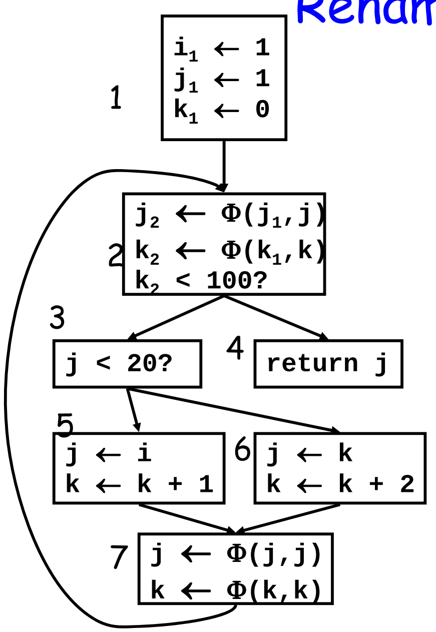


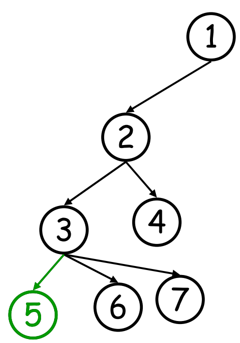


rename(2)
- defs & uses
- succs
- rename(3)

Var	Count	stack
i	1	0 1
j	~~1~~2	0 1 2
k	~~1~~2	0 1 2

Rename Vars

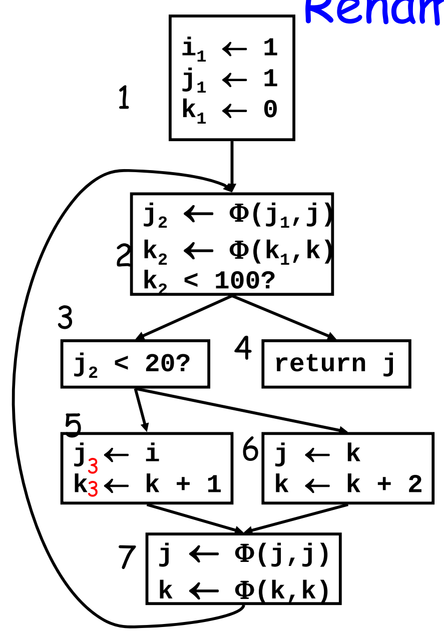


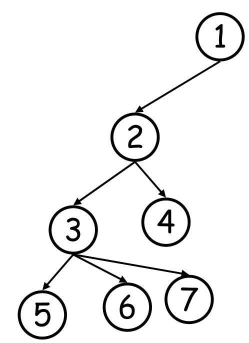


rename(3)
- defs & uses
- succs
- rename(5)

Var	Count	stack
i	1	0 1
j	2	0 1 2
k	2	0 1 2

Rename Vars

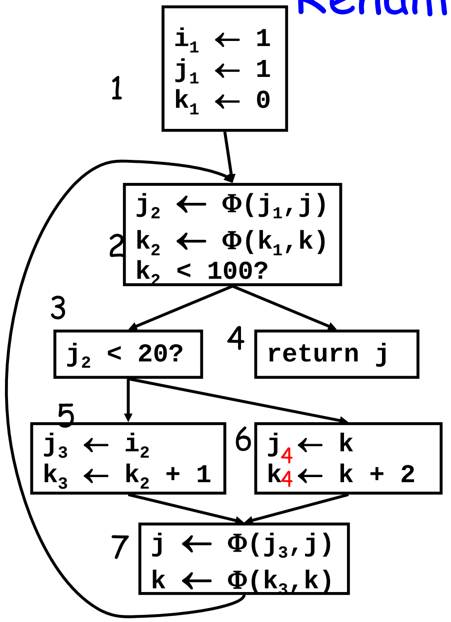


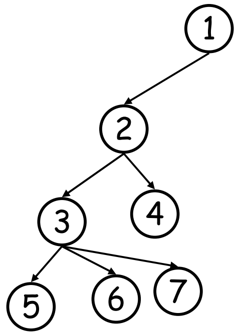


rename(5)
- defs & uses
- succs
- return
- next child of 3 is 6

Var	Count	stack
i	1	0 1
j	~~2~~3	0 1 2
k	~~2~~3	0 1 2

Rename Vars

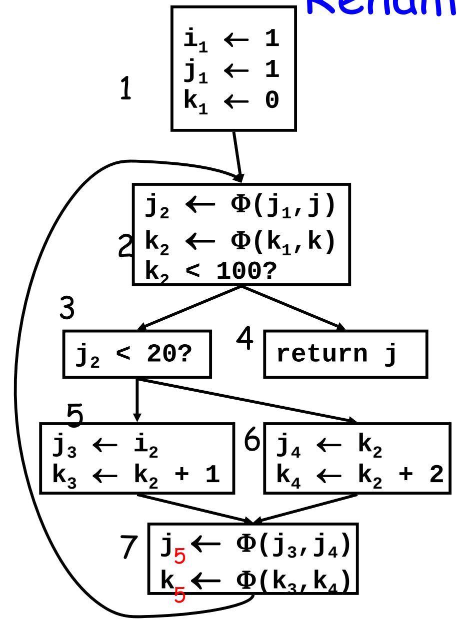


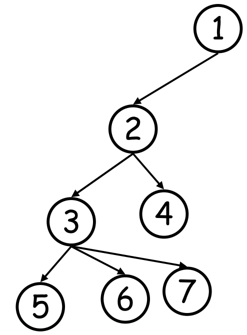


rename(6)
- defs & uses
- succs
- return
- next child of 3 is 7

Var	Count	stack
i	1	0 1
j	~~3~~ 4	0 1 2
k	~~3~~ 4	0 1 2

Rename Vars

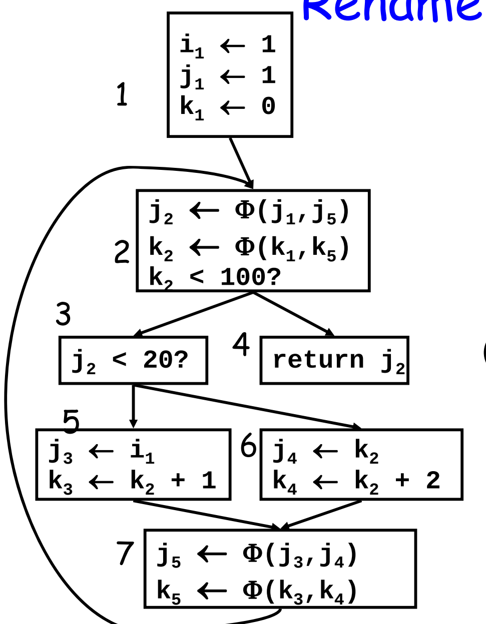


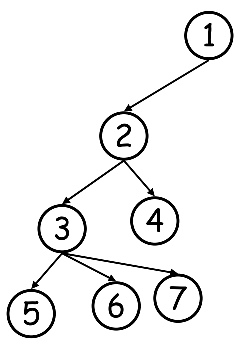


rename(7)
- defs & uses
- succs
- return
- return
- next child of 2 is 4

Var	Count	stack
i	1	0 1
j	~~4~~ 5	0 1 2 5
k	~~4~~ 5	0 1 2 5

Rename Vars





SSA Properties

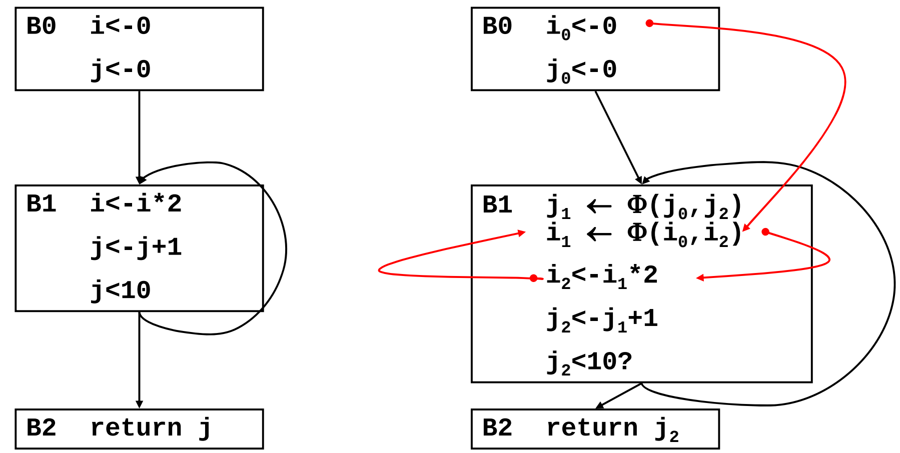
- Only 1 assignment per variable
- definitions dominate uses

Dead Code Elimination

Since we are using SSA, this is just a list of all variable assignments.

```
W <- list of all defs
while !W.isEmpty {
        Stmt S <- W.removeOne
        if |S.users| != 0 then continue
        if S.hasSideEffects() then continue
        foreach def in S.definers {
          def.users <- def.users - {S}
          if |def.uses| == 0 then
            W <- W UNION {def}
        }
}
```


Example DCE



Standard DCE leaves Zombies!

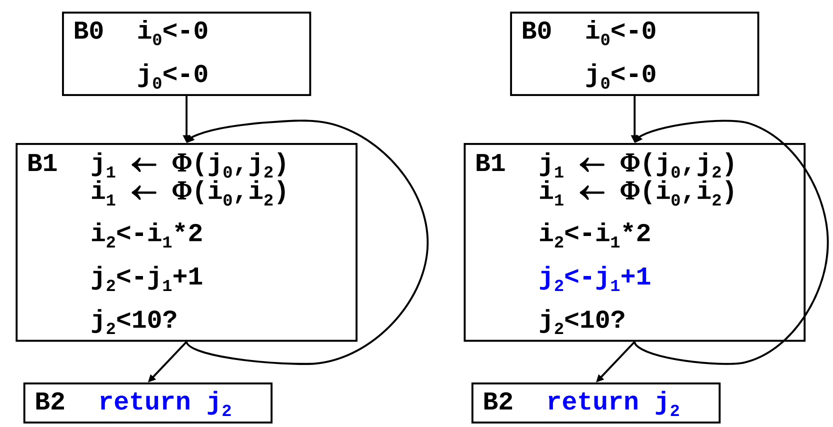
Aggressive Dead Code Elimination

Assume a stmt is dead until proven otherwise.

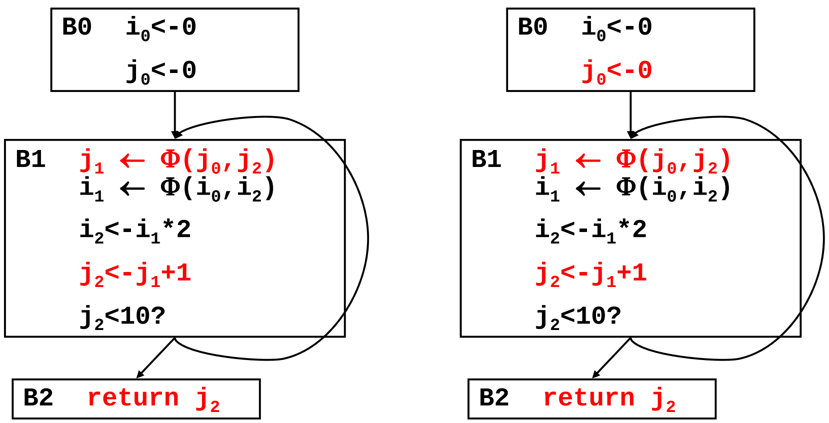
```
init:
   mark as live all stmts that have side-effects:
      - I/O
      - stores into memory
      - returns
      - calls a function that MIGHT have side-effects
   As we mark S live, insert S.defs into W

while (|W| > 0) {
   S <- W.removeOne()
   if (S is live) continue;
   mark S live, insert S.defs into W
}
```


Example DCE



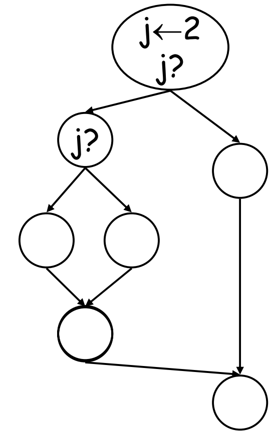
Example DCE



Problem!

Fixing DCE

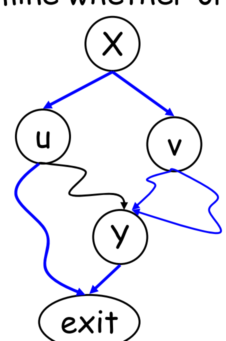
If S is live, then
  forall users of S.def
    if user is a branch -> mark user as live



Control Dependence

Y is control-dependent on X if
- X branches to u and v
- ∃ a path u→exit which does not go through Y
- ∀ paths v→exit go through Y

IOW, X can determine whether or not Y is executed.

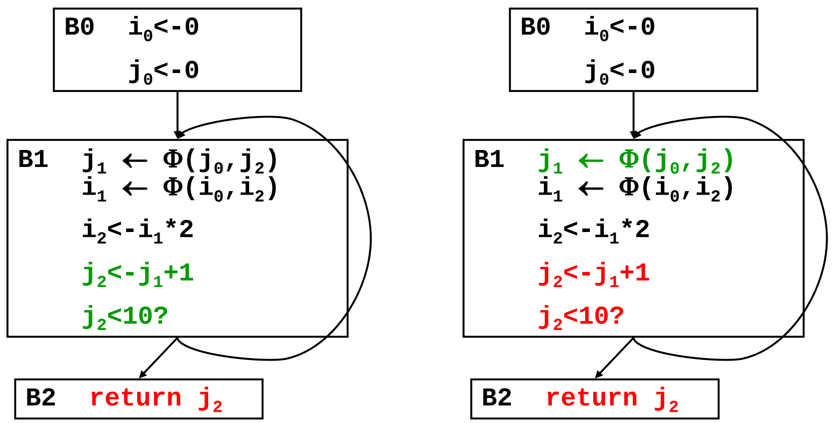


Aggressive Dead Code Elimination

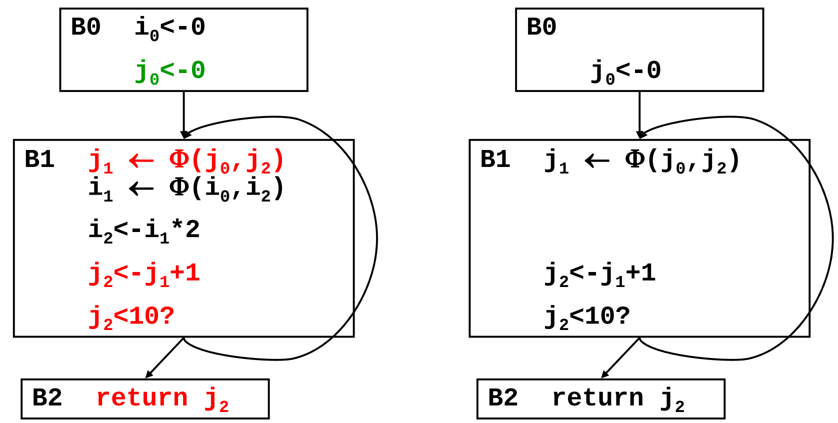
Assume a stmt is dead until proven otherwise.

```
while (|W| > 0) {
   S <- W.removeOne()
   if (S is live) continue;
   mark S live, insert
       - forall operands, S.operand.definers into W
       - S.CD-1 into W
}
```

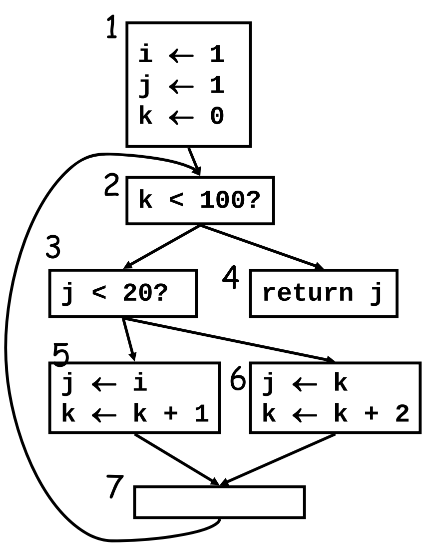
Example DCE



Example DCE

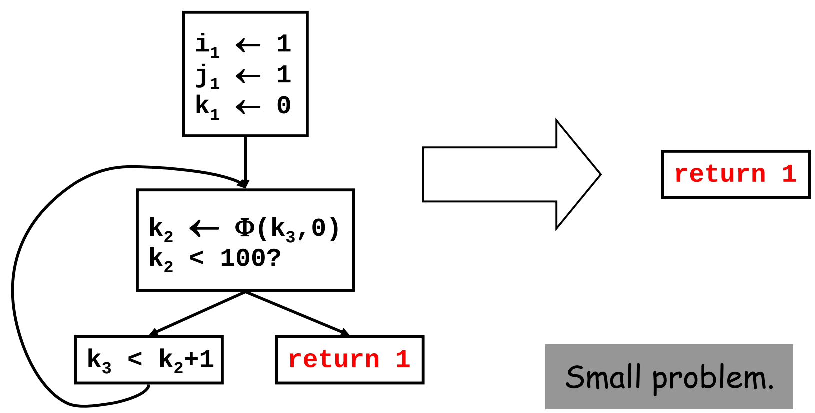


CCP Example



- 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 -> DCE

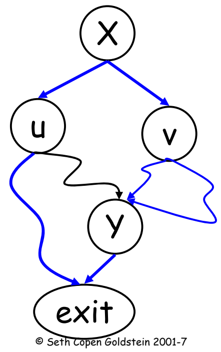


Small problem.

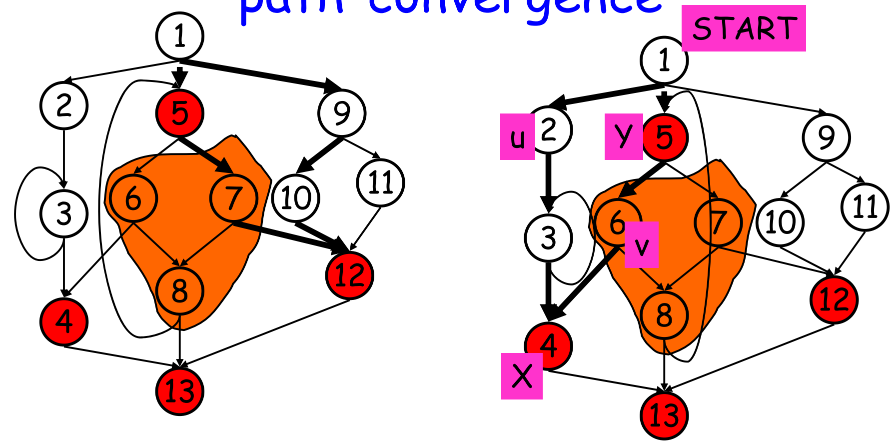
Finding the CDG

Y is control-dependent on X if
- X branches to u and v
- ∃ a path u→exit which does not go through Y
- ∀ paths v→exit go through Y

IOW, X can determine whether or not Y is executed.



Dominance Frontier & path-convergence

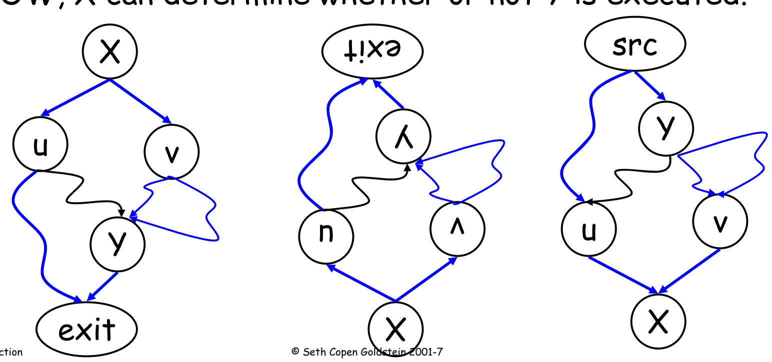


Any ideas?

Finding the CDG

Y is control-dependent on X if
- X branches to u and v
- ∃ a path u→exit which does not go through Y
- ∀ paths v→exit go through Y

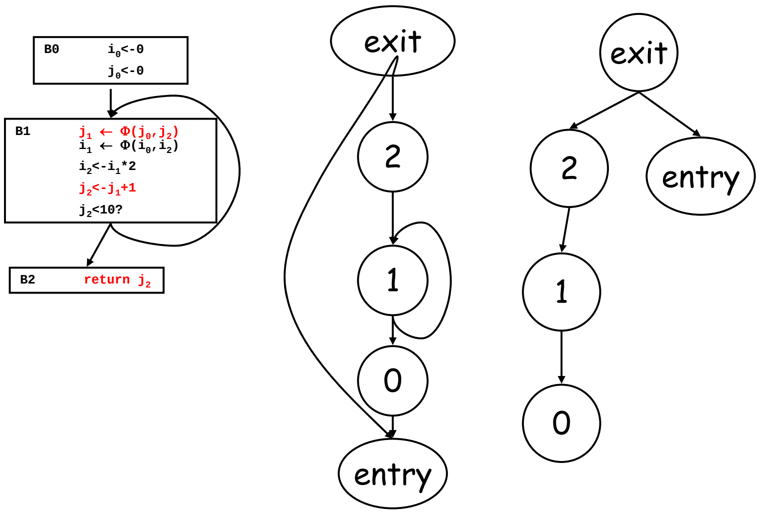
IOW, X can determine whether or not Y is executed.



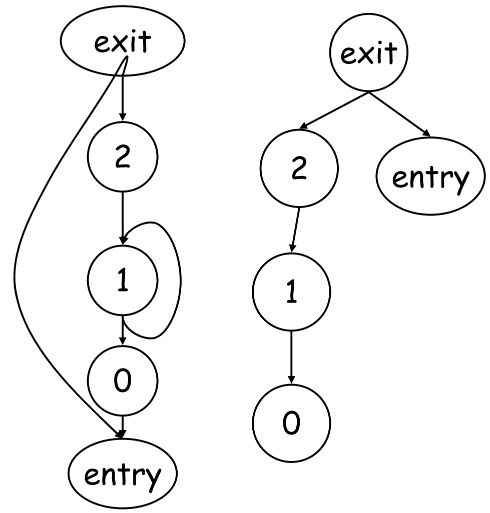
Finding the CDG

- Construct CFG
- Add entry node and exit node
- Add (entry,exit)
- Create G′, the reverse CFG
- Compute D-tree in G′ (post-dominators of G)
- Compute $DF_{G'}(y)$ for all y ∈ G′ (post-DF of G)
- Add (x,y) ∈ G to CDG if x ∈ $DF_{G'}(y)$

CDG of example



CDG of example



exit: {}
2: {entry}
1: {1,entry}
0: {entry}
entry: {}

Summary

- In SSA definitions dominate uses
- Use Dominance Frontier to create minimal SSA
  - Compute Dominance Tree
  - Compute DF
  - Compute Iterated Dominance Frontier
- Use D-Tree to rename variables
- Control Dependence can be computed by inspecting post-dominators, IOW, DF on reverse graph