15-745 Lecture 4

SSA

CCP, ADCE

Dominance & Minimal SSA

From Before… Def-Use Chains

```
• …
• for (i=0; i++; i<10) {
•     … = … i …;
•     …
• }
• for (i=j; i++; i<20) {
•     …  = i  …
• }
```

How is this related to RA?

Def-Use chains are 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;
  }
  …
```


Def-Use chains are expensive

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

In general,
- N defs
- M uses
- ⇒ O(NM) space and time

A solution is to limit each var to ONE def site

Def-Use chains are expensive

```
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 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;
  }
  …
```

A solution is to limit each var to ONE def site

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:
  - assign to a fresh variable at each stmt.
  - Each use uses the most recently defined var.
  - (Similar to Value Numbering)

Advantages of SSA

- Makes du-chains explicit
- Makes dataflow optimizations
  - Easier
  - faster
- Improves register allocation
  - Automatically builds Webs
  - Makes building interference graphs easier
- For most programs reduces space/time requirements

SSA History

- Developed by Wegman, Zadeck, Alpern, and Rosen in 1988
- New to gcc 4.0, used in ORC, LLVM, used in both IBM and Sun Java JIT compilers
  - and others

Straight-line SSA

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


Straight-line SSA

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

$a_1 \leftarrow x + y$
$b_1 \leftarrow a_1 + x$
$a_2 \leftarrow b_1 + 2$
$c_1 \leftarrow y + 1$
$a_3 \leftarrow c_1 + a_2$

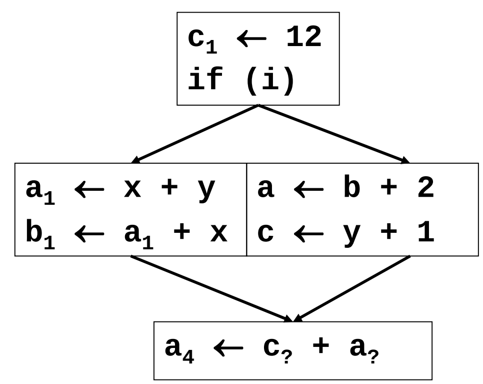
SSA

- 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:
  - assign to a fresh variable at each stmt.
  - Each use uses the most recently defined var.
  - (Similar to Value Numbering)

- What about at joins in the CFG?

Merging at Joins

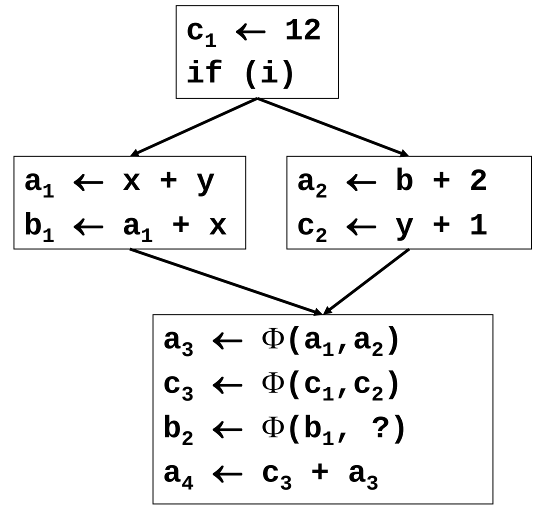
```
c ← 12
if (i) {
  a ← x + y
  b ← a + x
} else {
  a ← b + 2
  c ← y + 1
}
a ← c + a
```



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:
  - assign to a fresh variable at each stmt.
  - Each use uses the most recently defined var.
  - (Similar to Value Numbering)
- What about at joins in the CFG?
  - Use a notional fiction: A Φ function

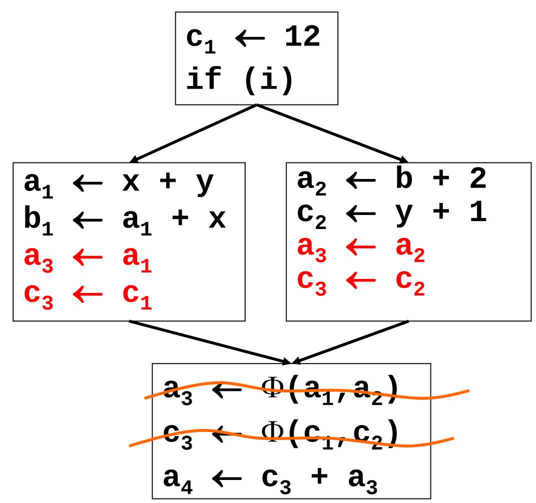
Merging at Joins



The Φ function

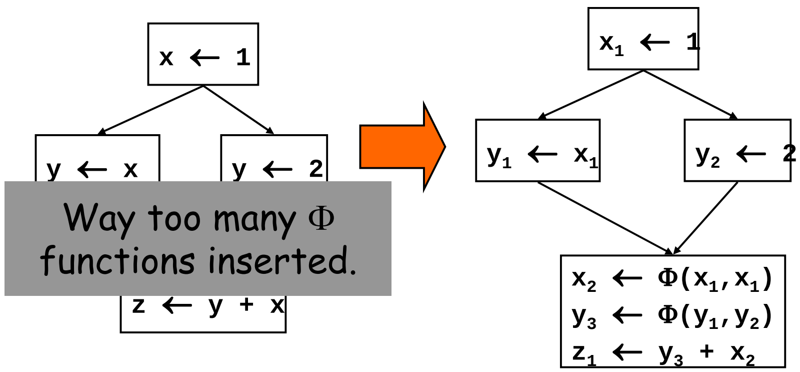
- Φ merges multiple definitions along multiple control paths into a single definition.
- At a BB with p predecessors, there are p arguments to the Φ function.
  $x_{new} \leftarrow \Phi(x_1, x_1, x_1, \ldots, x_p)$
- How do we choose which x_i to use?
  - We don't really care!
  - If we care, use moves on each incoming edge

"Implementing" Φ



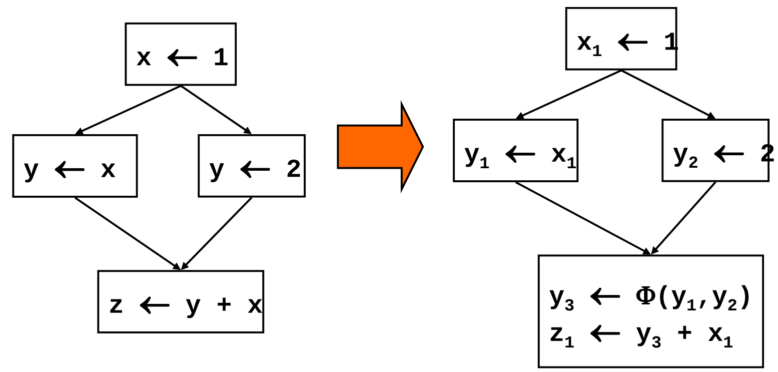
Trivial SSA

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

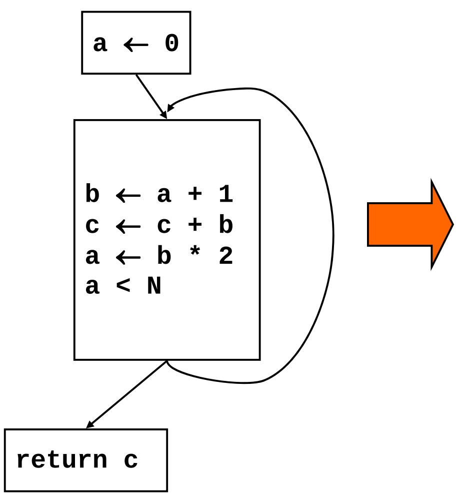


Minimal SSA

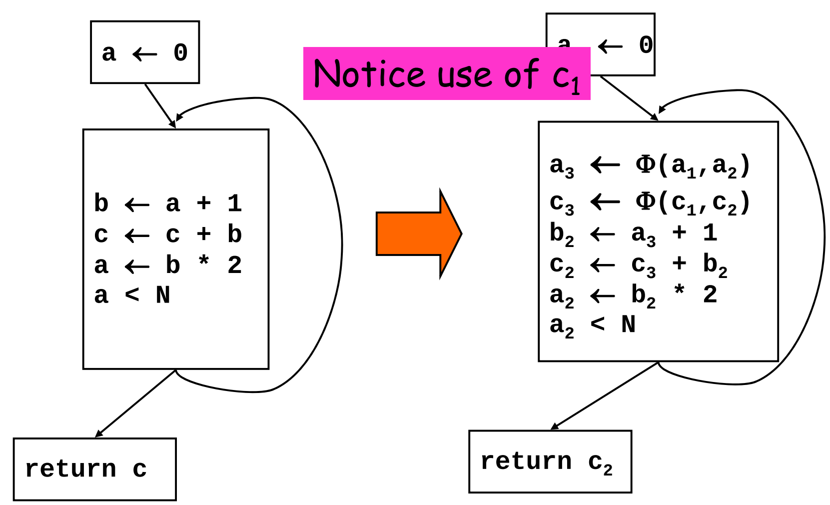
- Each assignment generates a fresh variable.
- At each join point insert Φ functions for all variables with multiple outstanding defs.



Another Example

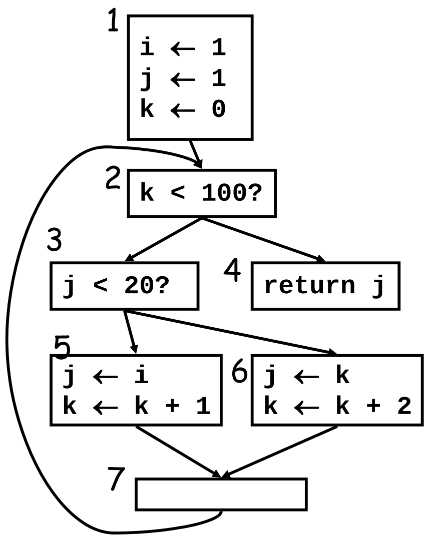


Another Example

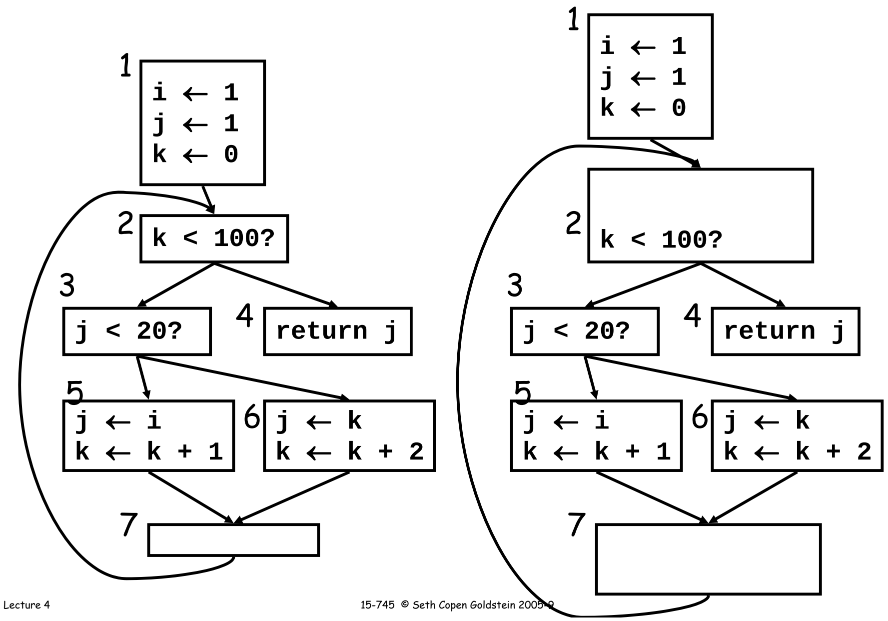


Lets optimize the following:

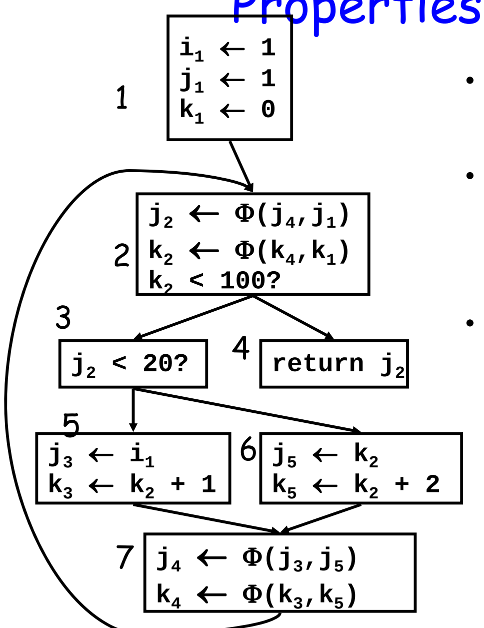
```
i=1;
j=1;
k=0;
while (k<100) {
  if (j<20) {
     j=i;
     k++;
  } else {
     j=k;
     k+=2;
  }
}
return j;
```



First, turn into SSA



Properties of SSA



- Only 1 assignment per variable
- definitions dominate uses

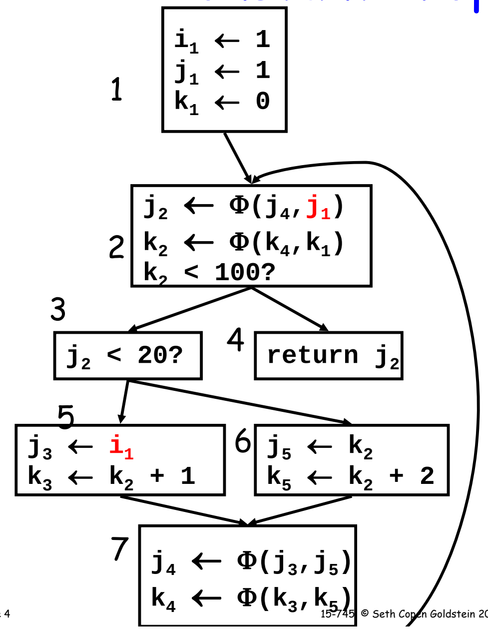
- Can we use this to help with constant propagation?

Constant Propagation

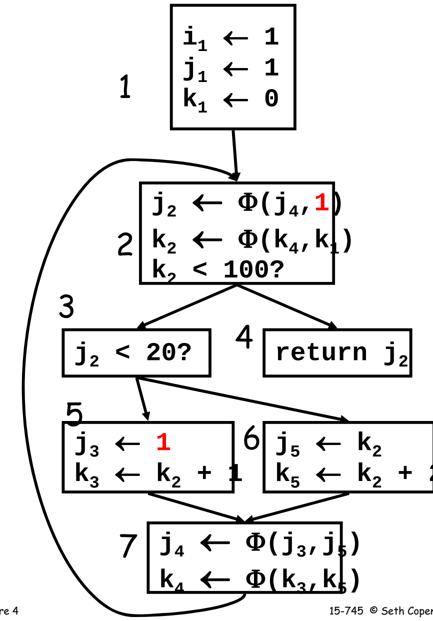
- If "v ← c", replace all uses of v with c
- If "v ← **Φ(c,c,c)**" replace all uses of v with c

```
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)
}
```

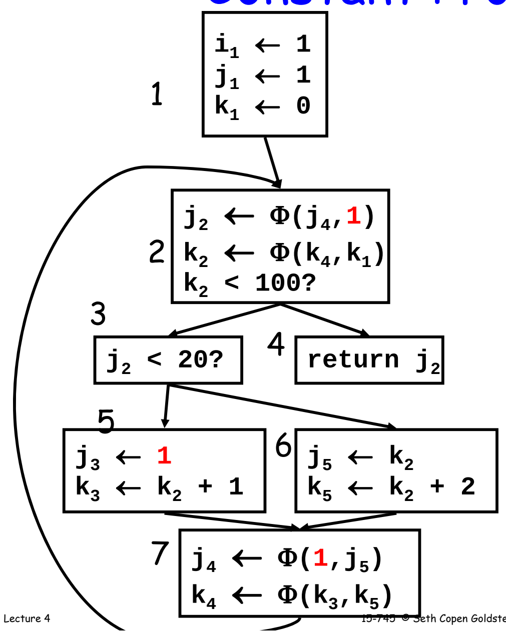
Constant Propagation



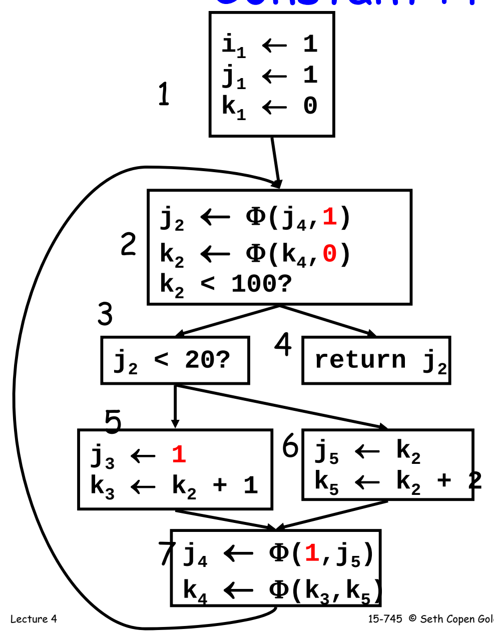
Constant Propagation



Constant Propagation



Constant Propagation

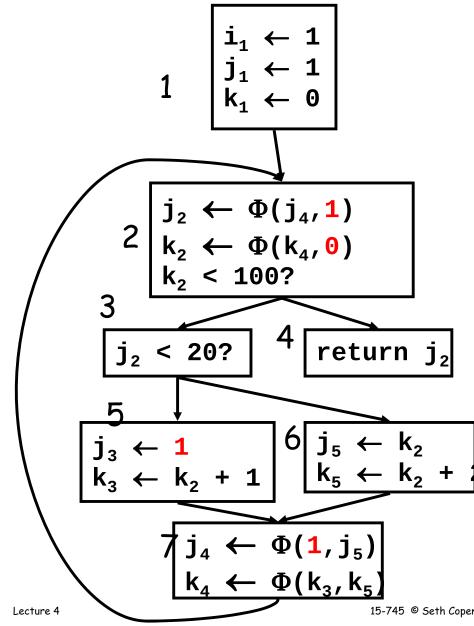


But, so what?

Other stuff we can do?

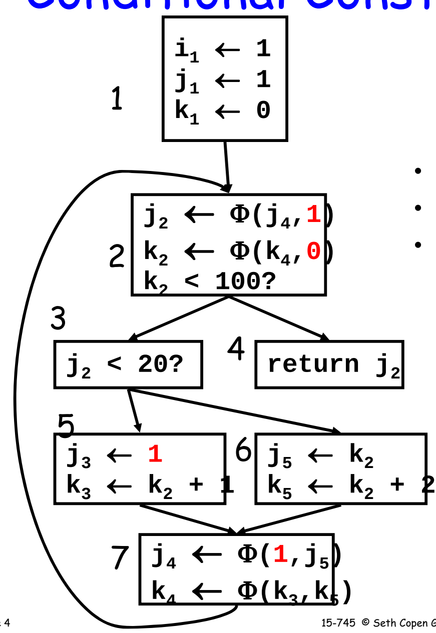
- Copy propagation
  delete "x ← Φ(y)" and replace all x with y
  delete "x ← y" and replace all x with y
- Constant Folding
  (Also, constant conditions too!)
- Unreachable Code
  Remember to delete all edges from unreachable block

Constant Propagation



But, so what?

Conditional Constant Propagation



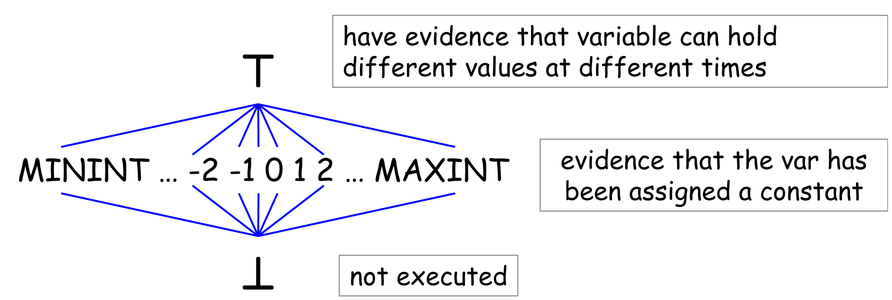
- Does block 6 ever execute?
- Simple CP can't tell
- CCP can tell:
  - Assumes blocks don't execute until proven otherwise
  - Assumes Values are constants until proven otherwise

CCP data structures & lattice

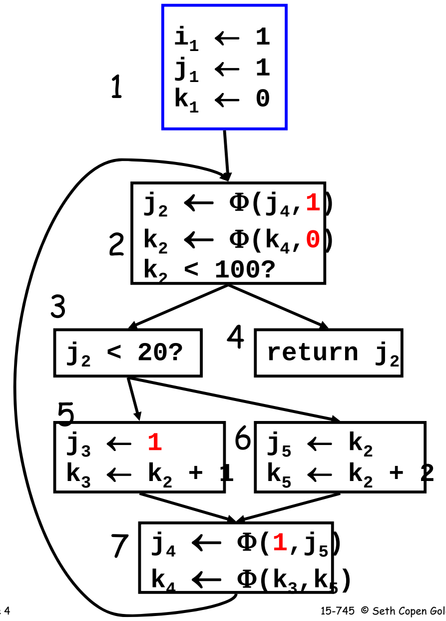
Keep track of:

- 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)

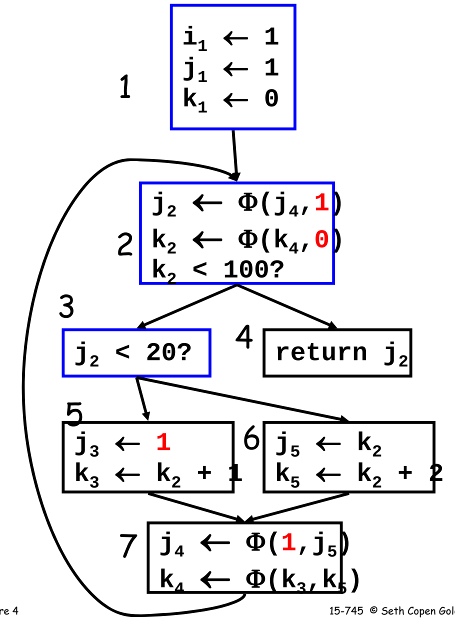
Use a lattice for variables:



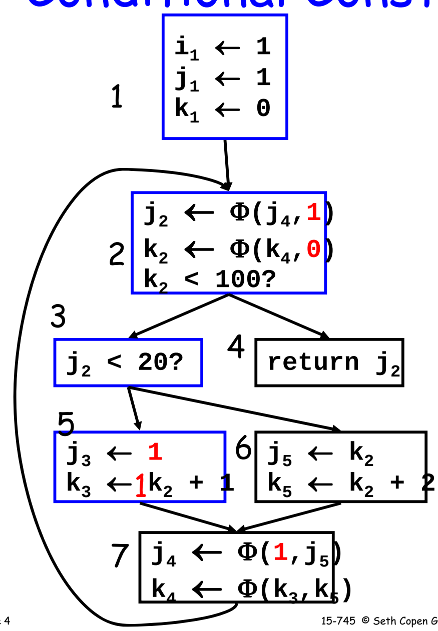
Conditional Constant Propagation



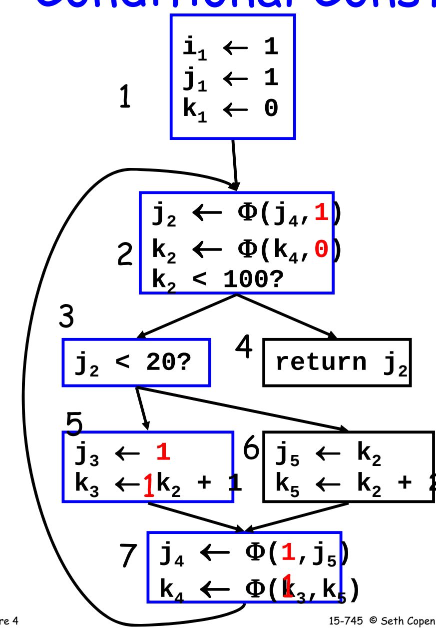
Conditional Constant Propagation



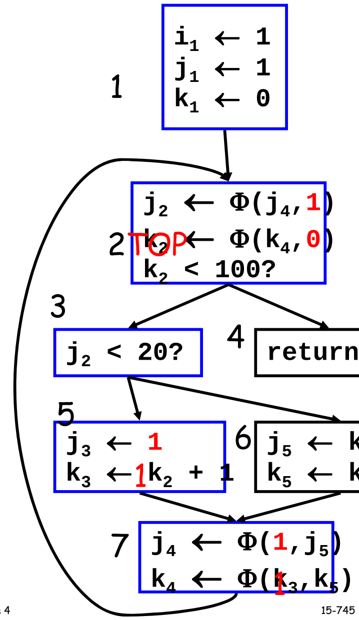
Conditional Constant Propagation



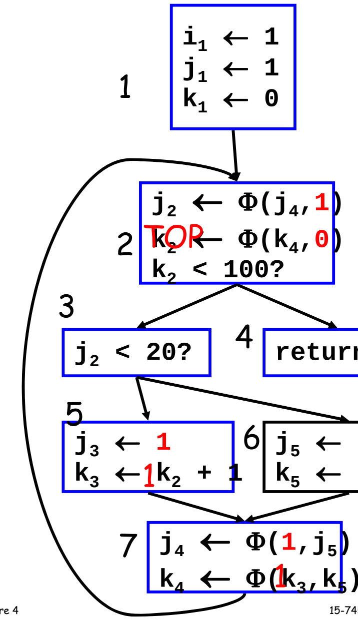
Conditional Constant Propagation



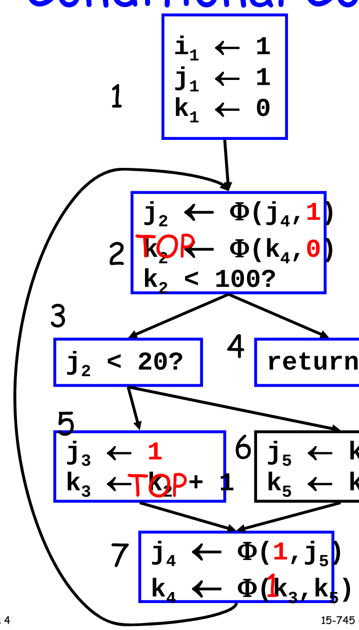
Conditional Constant Propagation



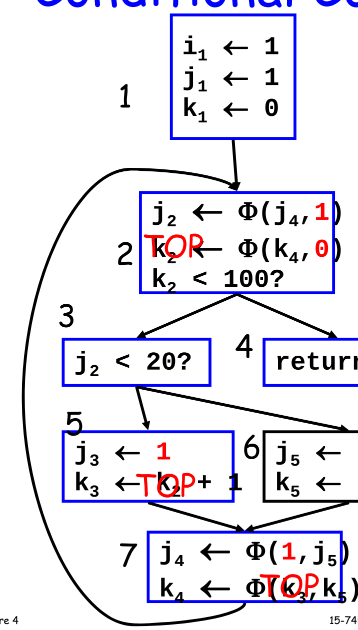
Conditional Constant Propagation



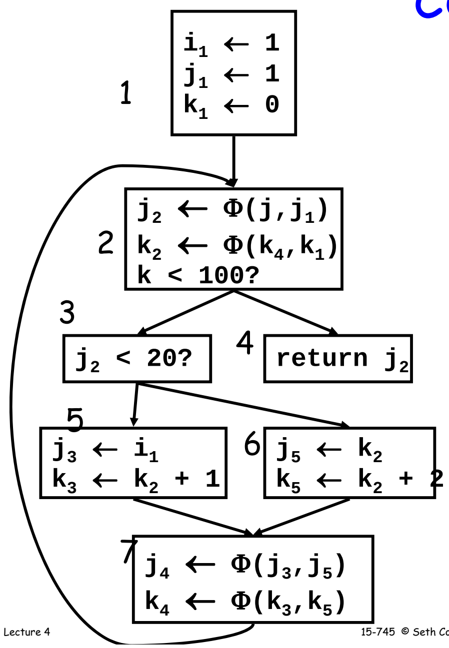
Conditional Constant Propagation

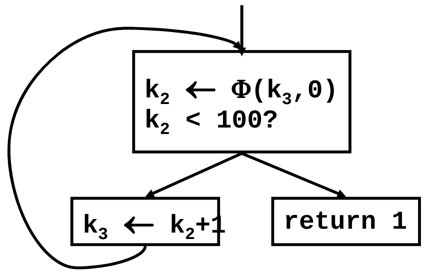


Conditional Constant Propagation



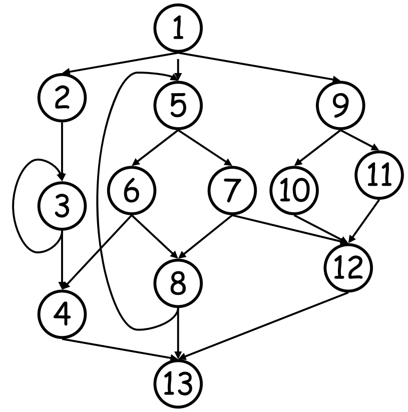
CCP





We can do more …

When do we insert Φ?



CFG

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

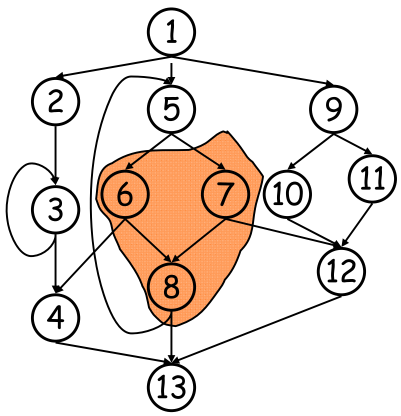
When do we insert Φ?

- 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 from y to z, P_{yz} s.t.
    - $P_{xz} \cap P_{yz} = \{ z \}$
    - $z \notin P_{xq}$ or $z \notin P_{xr}$ where $P_{xz} = P_{xq} \rightarrow z$ and $P_{yz} = P_{xr} \rightarrow z$
  - Entry block contains an implicit def of all vars
  - Note: A = $\Phi(\ldots)$ is a def of A

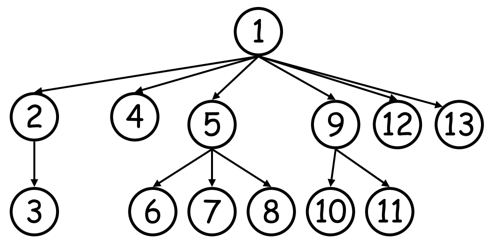
Dominance Property of SSA

- 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

Dominance



CFG

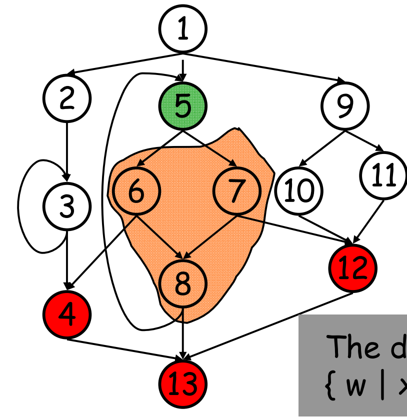


If there is a def of a in block 5, which nodes need a $\Phi()$?

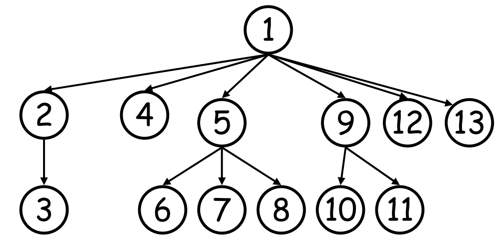
D-Tree

x strictly dominates w (s sdom w) iff x dom w AND x ≠ w

Dominance Frontier



CFG

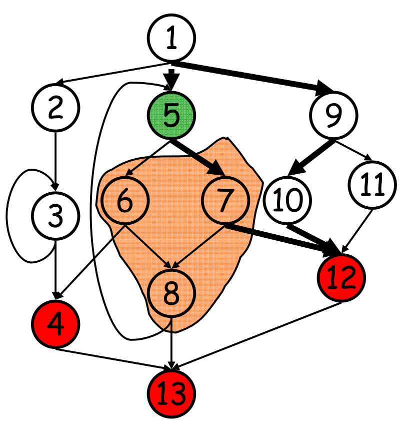


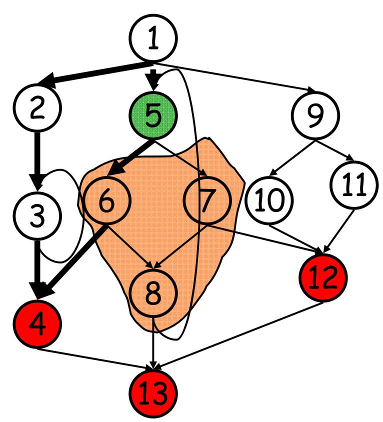
The dominance Frontier of a node x = { w | x dom pred(w) AND !(x sdom w)}

D-Tree

x strictly dominates w (s sdom w) iff x dom w AND x ≠ w

Dominance Frontier & path-convergence





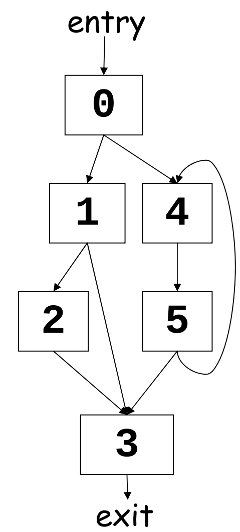
Computing Dominance Frontier

- You've probably already seen a $O(n^3)$ iterative algorithm
- There's also a near linear time algorithm due to Tarjan and Lengauer (Chap 19.2)
  - SSA construction therefore near linear
  - SSA form makes many optimizations linear (no need for iterative data flow)

Side trip: Dominators

Dominators

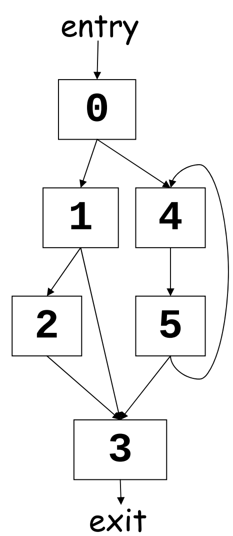
- *a dom b*
  - block *a dominates* block *b* if every possible execution path from *entry* to *b* includes *a*
    - **entry** dominates everything
    - **0** dominates everything but entry
    - **1** dominates **2** and **1**



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 $D[\text{entry}]$) to be the set of all blocks, and then iterate until no $D[n]$ changes:

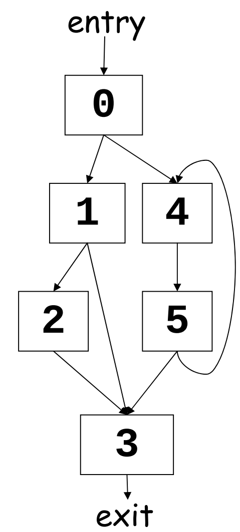
$$D[\text{entry}] = \{\text{entry}\}$$

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

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}

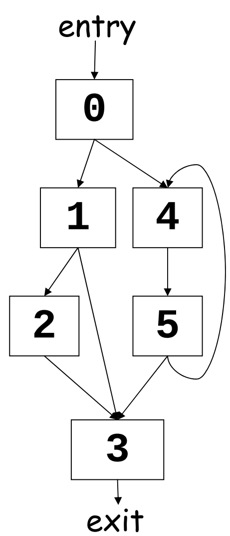


Example

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:

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

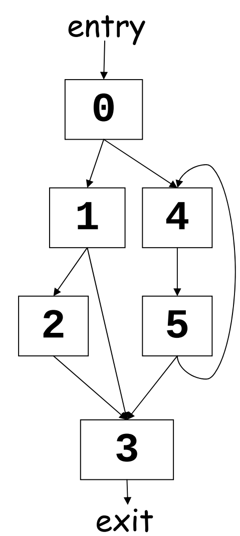


Example

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}

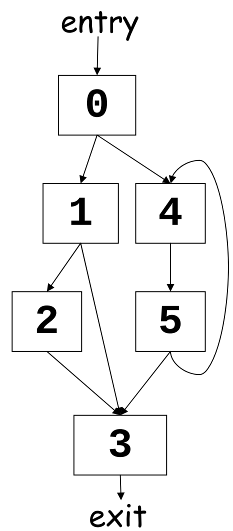
Update rule:

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



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

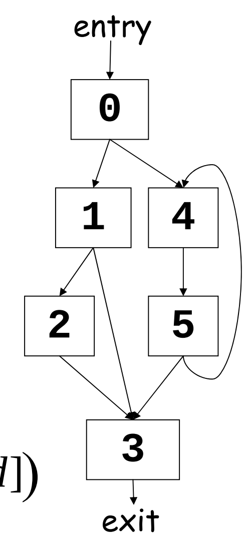
- Iterative algorithm is $O(n^2 e)$
  - assuming bit vector sets
- More efficient algorithm due to Lengauer and Tarjan
  - $O(e \cdot \alpha(e,n))$ *inverse Ackermann*
  - much more complicated
  - your book provides a simple algorithm that is very fast in practice
    - faster than Tarjan algorithm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

Computing dominators

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

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}	

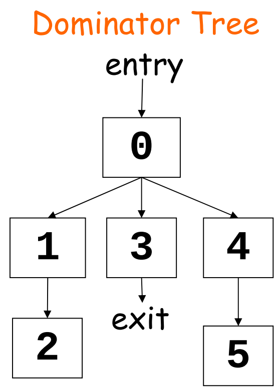


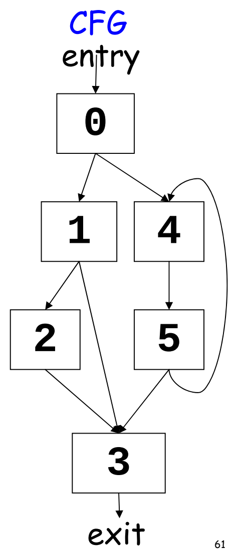
Update rule: $$iD[n] = iD[n] - \left(\bigcup_{d \in iD[n]} sD[d] \right)$$

Dominator Tree

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

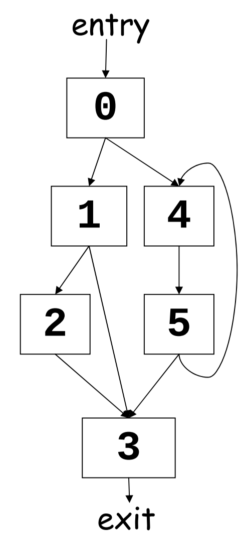
block	iD[n]
entry	{}
0	{entry}
1	{0}
2	{1}
3	{0}
4	{0}
5	{4}
exit	{3}





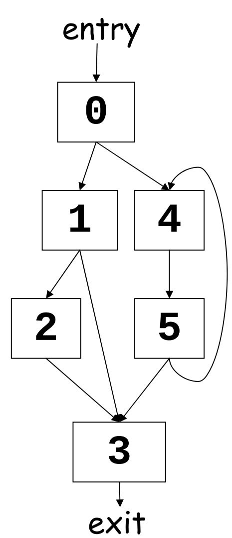
Post-Dominance

- Block *a* post-dominates *b* (a pdom b) if every path from *a* to the exit block includes *b*
- pdom on CFG is the same as dom on the reverse (all edges reversed) CFG
- **0** post-dominates ?
  1 post-dominates ?
  4 post-dominates ?



Dominance Frontier

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



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

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

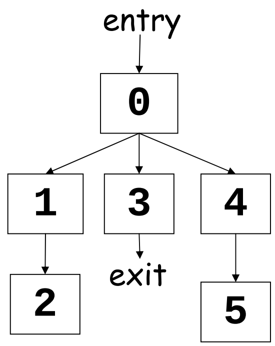
Skip example

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}	

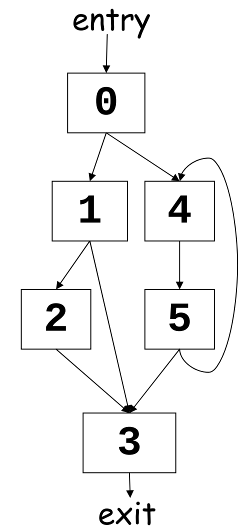
Dominator Tree



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}	{}



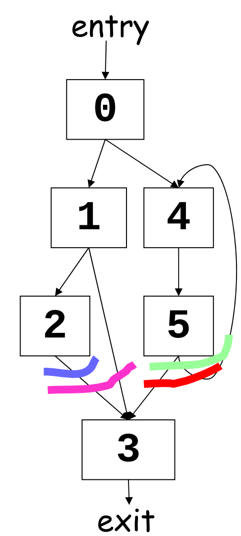
Example

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

Using DF to compute SSA

- place all Φ()
- Rename all variables

Using DF to Place Φ()

- Gather all the defsites of every variable
- Then, for every variable
  - foreach defsite
    - foreach node in DF(defsite)
      - if we haven't put Φ() in node put one in
      - If this node didn't define the variable before: add this node to the defsites

- This essentially computes the Iterated Dominance Frontier on the fly, inserting the minimal number of Φ() 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}
      }
    }
  }
}
```

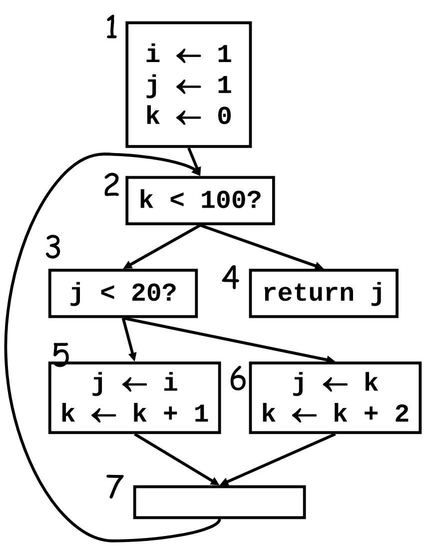
Renaming Variables

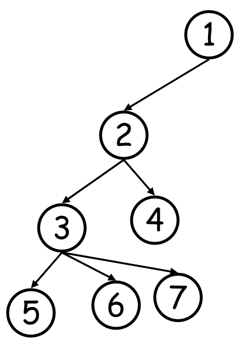
- Walk the D-tree, renaming variables as you go
- Replace uses with more recent renamed def
  - For straight-line code this is easy
  - If there are branches and joins?

Renaming Variables

- Walk the D-tree, renaming variables as you go
- Replace uses with more 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

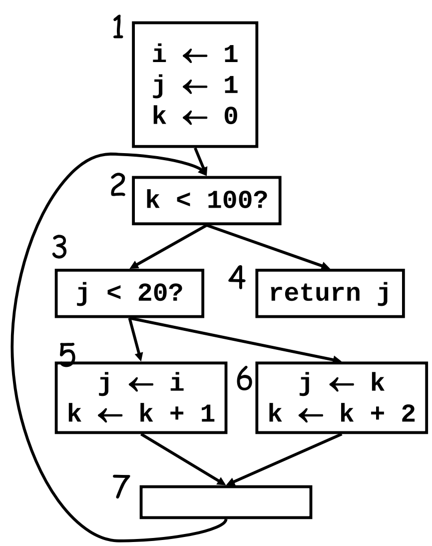
Compute D-tree

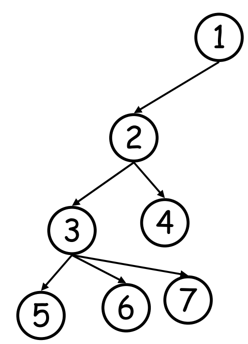




D-tree

Compute Dominance Frontier

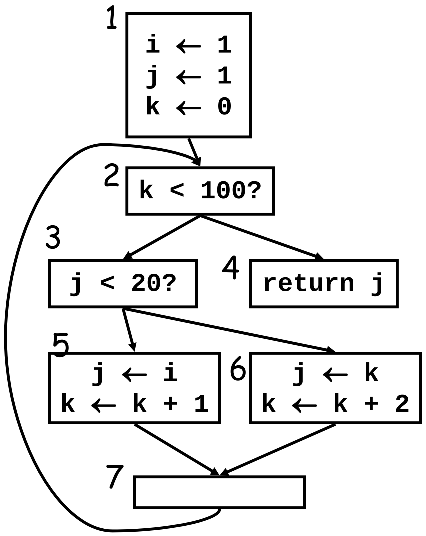




1	{}
2	{2}
3	{2}
4	{}
5	{7}
6	{7}
7	{2}

DFs

Insert Φ()



DFs

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

orig[n]

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

defsites[v]

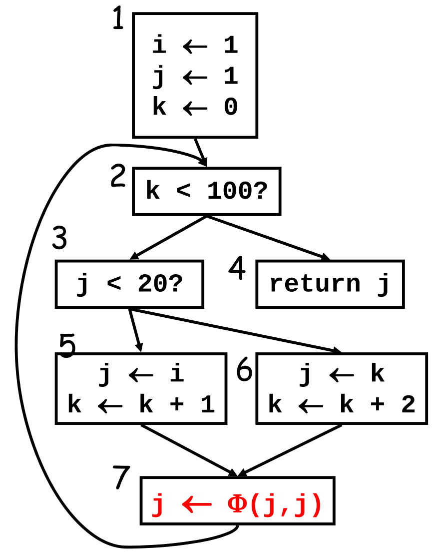
i	{1}
j	{1,5,6}
k	{1,5,6}

var i: W={1}

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

DF{1}, DF{5}

Insert Φ()



DFs

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

orig[n]

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

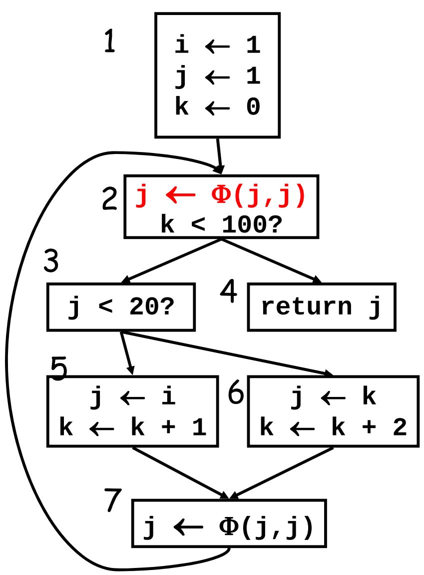
defsites[v]

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

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

DF{1}, DF{5}

Insert Φ()



DFs

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

orig[n]

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

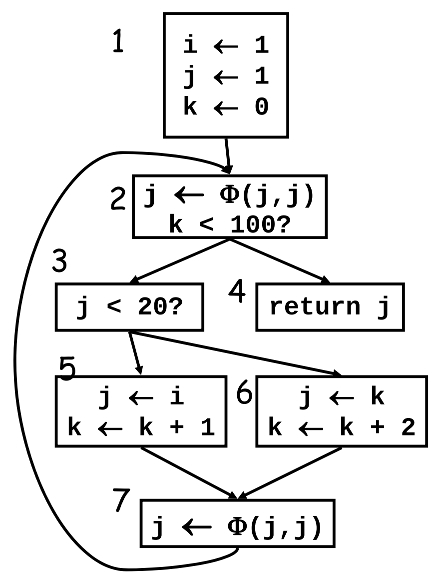
defsites[v]

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

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

DF{1}, DF{5}

Insert Φ()



DFs

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

orig[n]

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

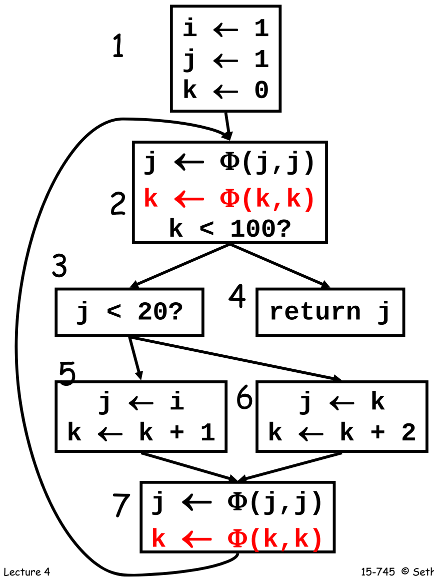
defsites[v]

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

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

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

Insert Φ()



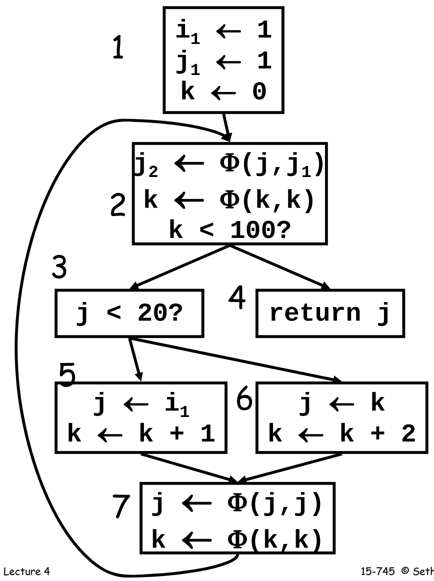
DFs	
1	{}
2	{2}
3	{2}
4	{}
5	{7}
6	{7}
7	{2}

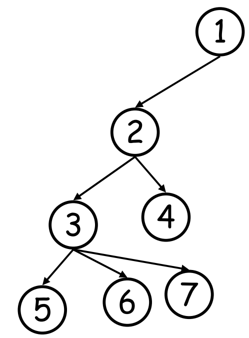
orig[n]	
1	{ i,j,k}
2	{}
3	{}
4	{}
5	{j,k}
6	{j,k}
7	{}

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

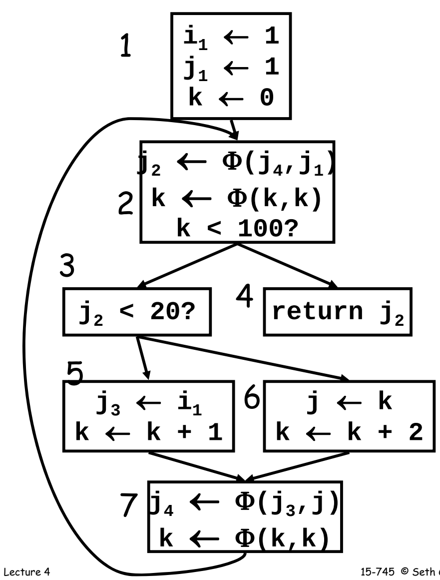
var k: W={1,5,6}

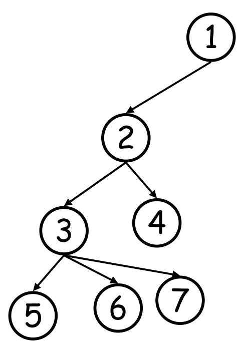
Rename Vars



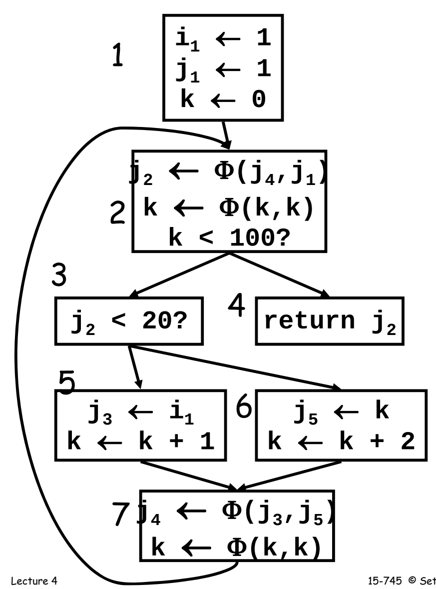


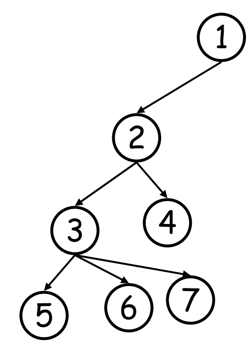
Rename Vars



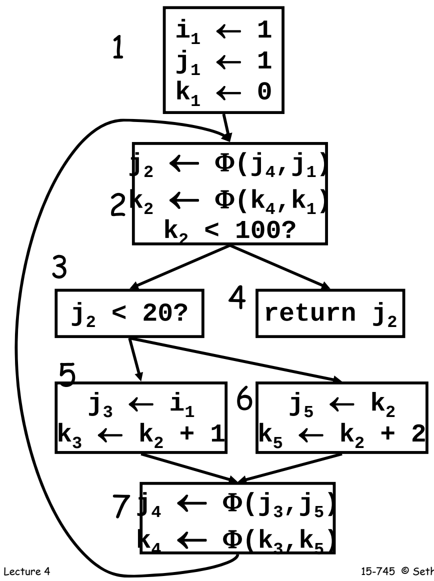


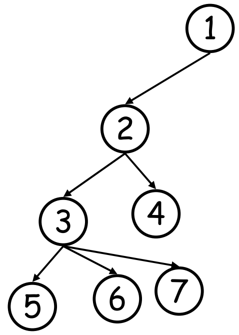
Rename Vars





Rename Vars





SSA Properties

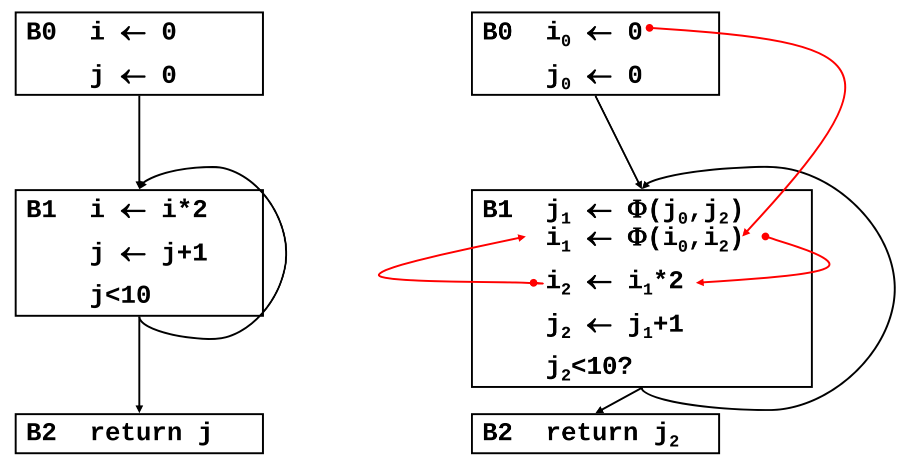
- Only 1 assignment per variable
- definitions dominate uses

Dead Code Elimination

```
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}
          delete S
        }
}
```

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

Example DCE



Standard DCE leaves Zombies!

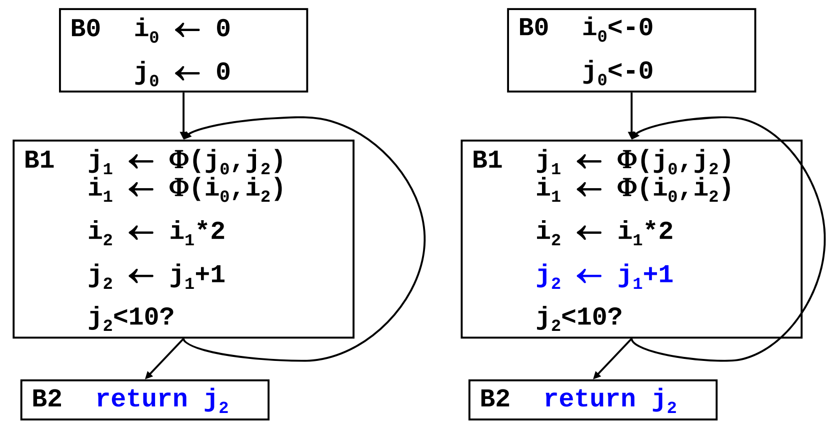
Aggressive Dead Code Elimination

Assume a stmt is dead until proven otherwise.

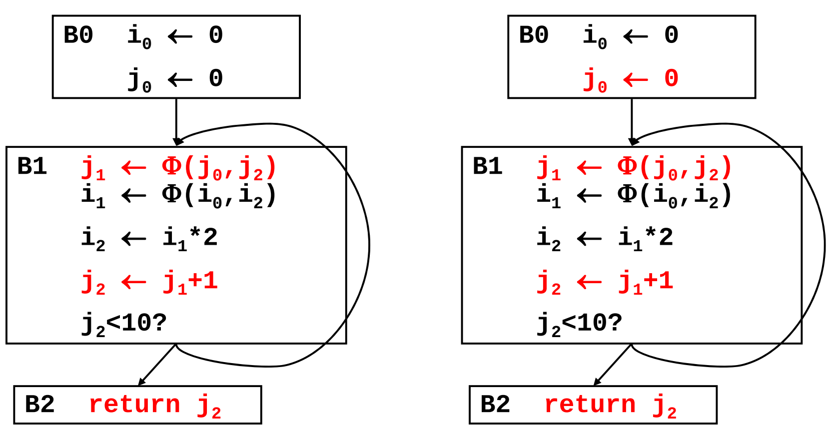
```
init:
   mark as live all stmts that have side-effects:
      - I/O
      - stores into memory
      - returns
      - calls a function that MIGHT have side-effects
   As we mark S alive, insert S.defs into W

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


Example DCE



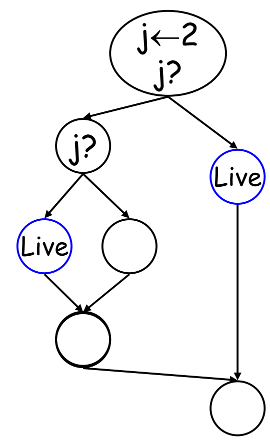
Example DCE



Problem!

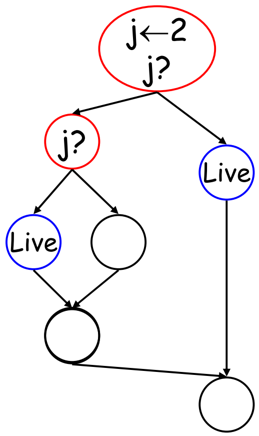
Fixing ADCE

- If S is live, then
  If T determines if S can execute, T should be live



Fixing DCE

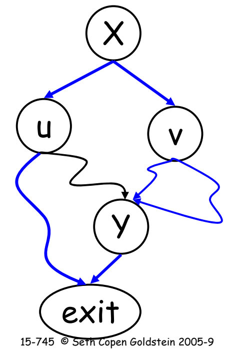
- If S is live, then
  If T determines if S can execute, T should be live



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

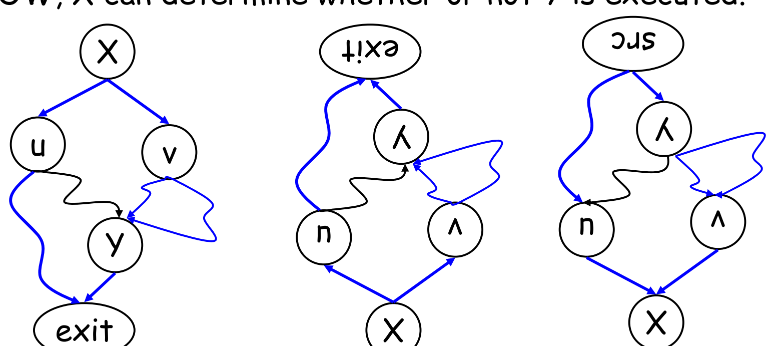
IOW, X can determine whether or not Y is executed.



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

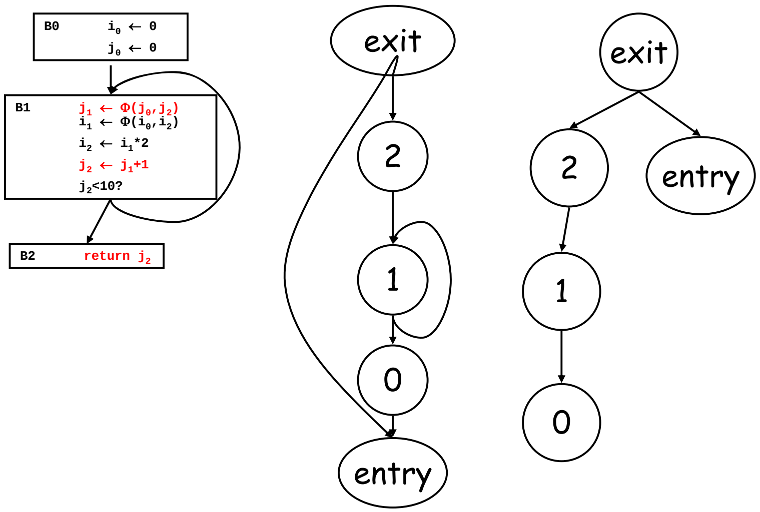
IOW, X can determine whether or not Y is executed.



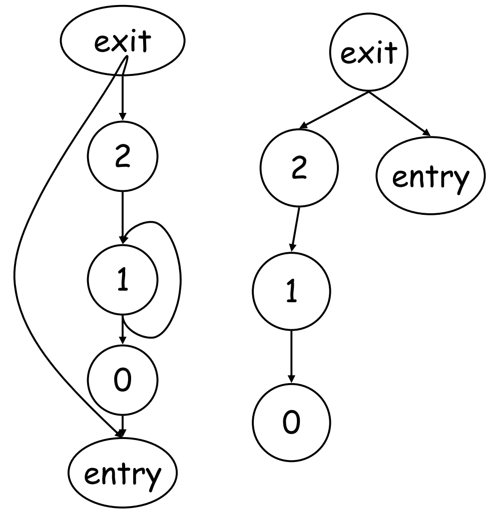
Finding the CDG

- 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 $y \in G'$ (post-DF of G)
- Add $(x,y) \in G$ to CDG if $x \in DF_{G'}(y)$

CDG of example



CDG of example



DF

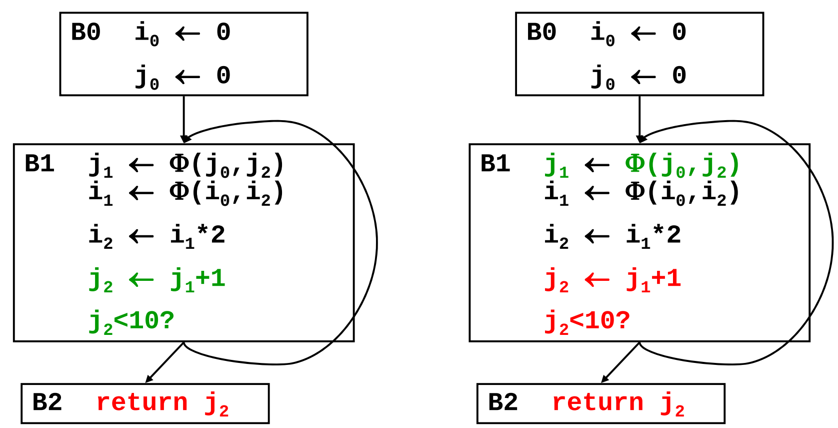
exit: {}
2: {entry}
1: {1,entry}
0: {entry}
entry: {}

Aggressive Dead Code Elimination

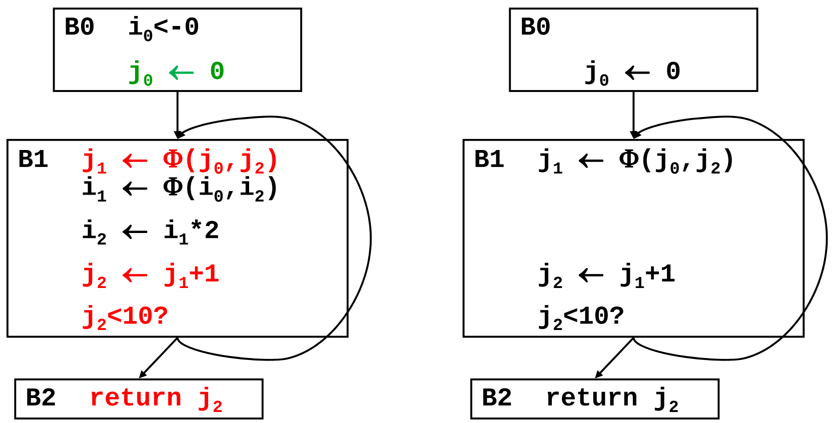
Assume a stmt is dead until proven otherwise.

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

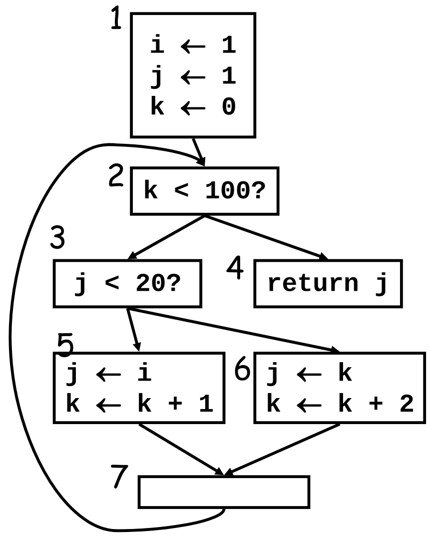

Example DCE



Example DCE

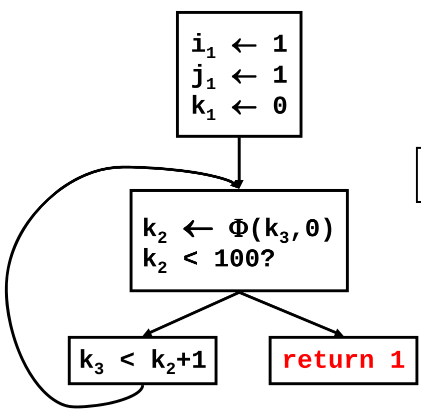


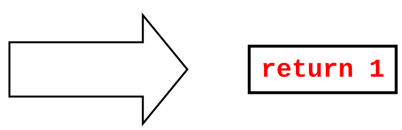
CCP Example



- 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





Problem??

Whew!

- SSA: 1 assignment per variable. Defs dom uses
- Minimal SSA, Phi-functions, variable relabeling
- Dominators, dominator trees, dominance frontier
- CCP
- ADCE
- Control dependence