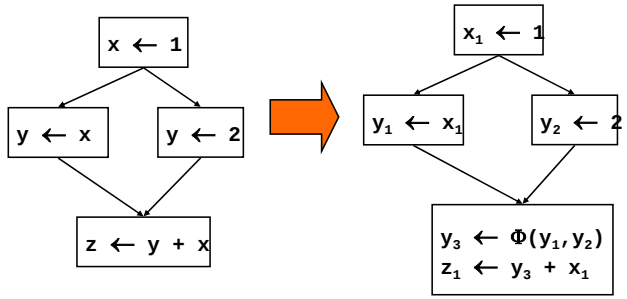
15-745

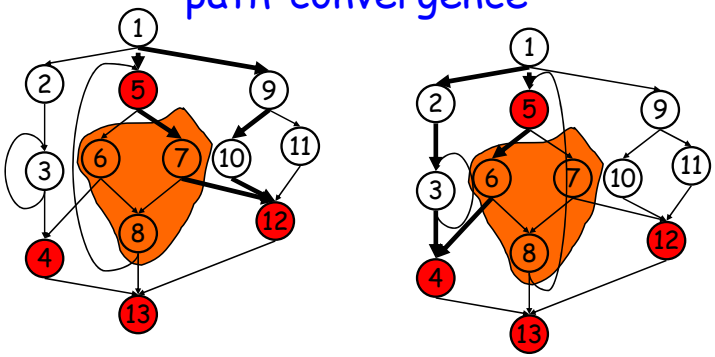
SSA & CCP & DCE & CDG

Review: Minimal SSA

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



Review: Dominance Frontier & path-convergence



Constant Propagation

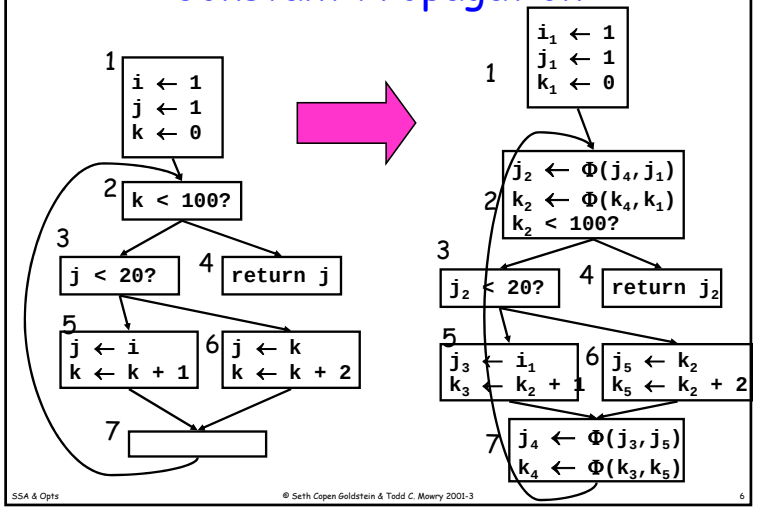
- If "v ← c", replace all uses of v with c
- If "v ← **Φ(c,c,c)**" replace all uses of v with c

```
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)
}
```

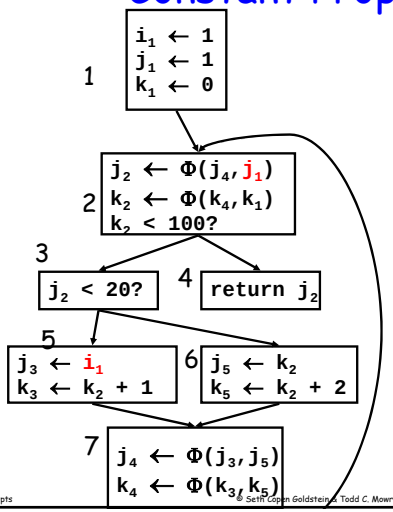
Other stuff we can do?

- Copy propogation
  - 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

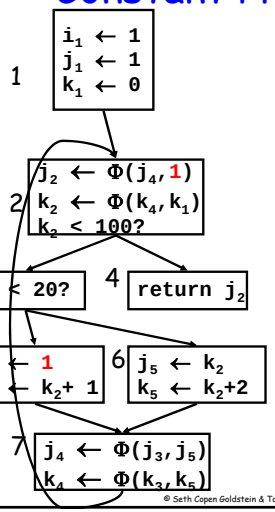
Constant Propagation



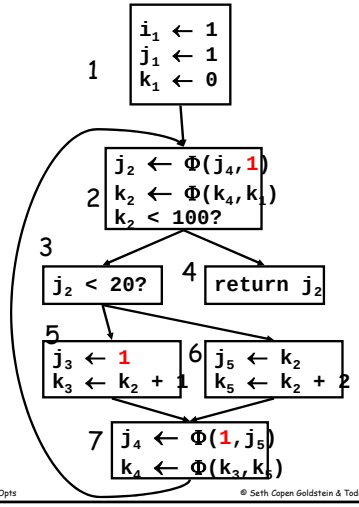
Constant Propagation



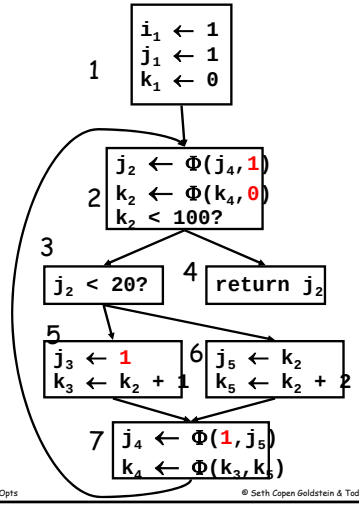
Constant Propagation



Constant Propagation

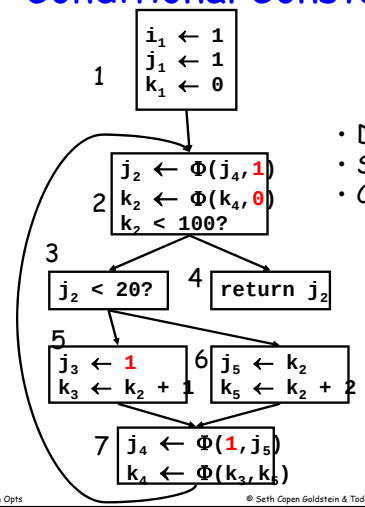


Constant Propagation



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

Tracks:

- 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)

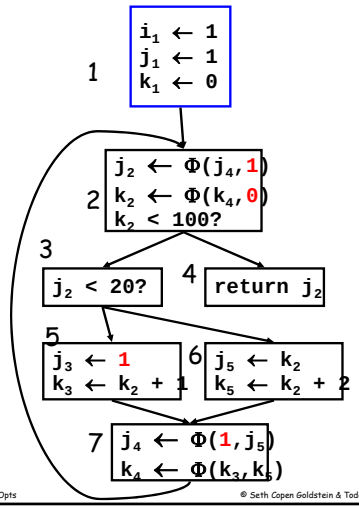
Use a lattice for variables:

TOP = not executed

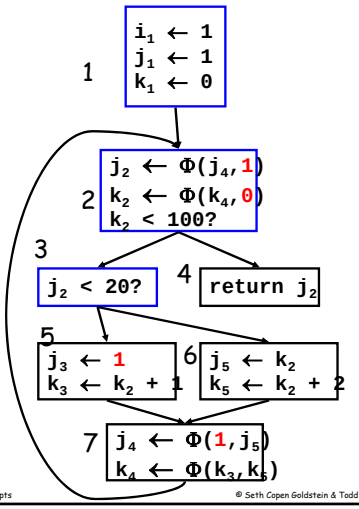
integers = we have seen evidence that the var has been assigned a constant with the value

BOT = we have evidence that variable can hold different values at different times

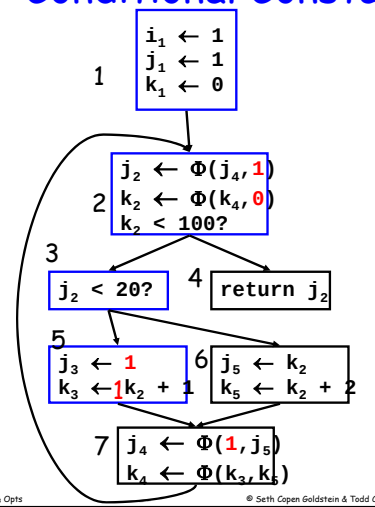
Conditional Constant Propagation



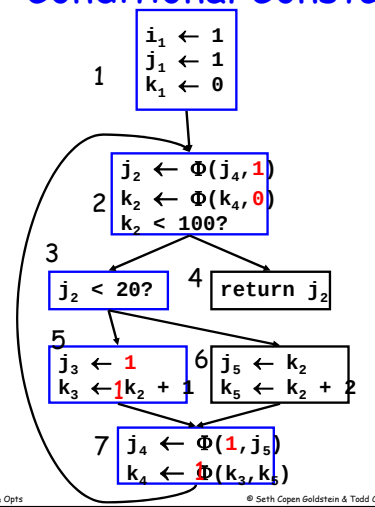
Conditional Constant Propagation



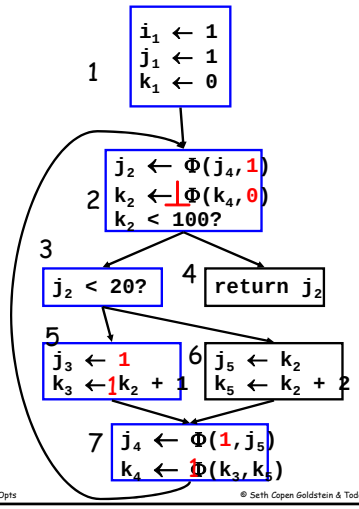
Conditional Constant Propagation



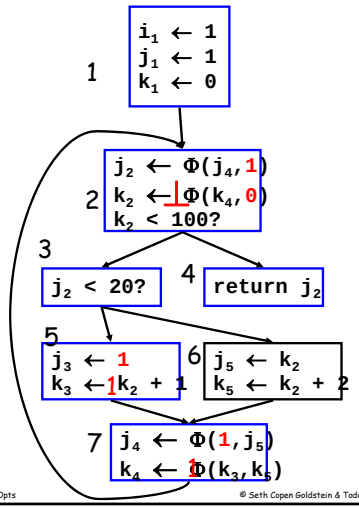
Conditional Constant Propagation



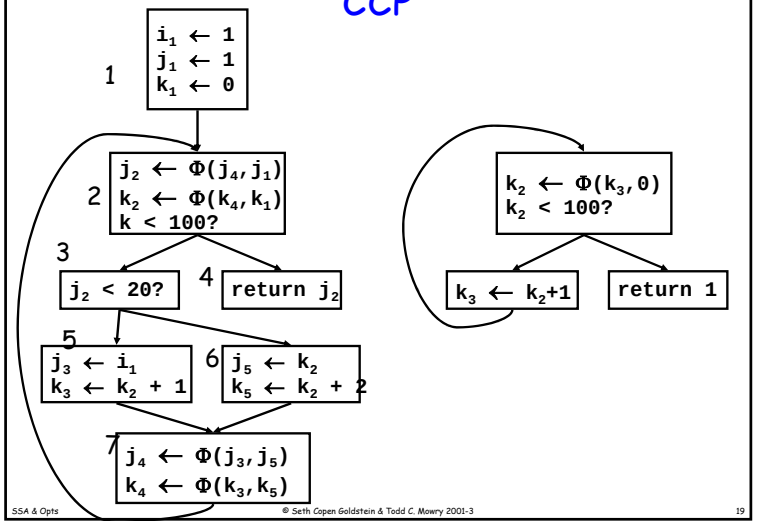
Conditional Constant Propagation



Conditional Constant Propagation



CCP

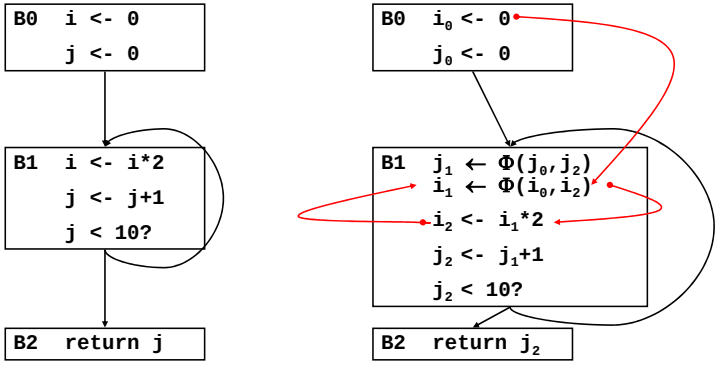


Dead Code Elimination

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.users| == 0 then
            W <- W UNION {def}
        }
        delete S
}
```

Example DCE



Standard DCE leaves Zombies!

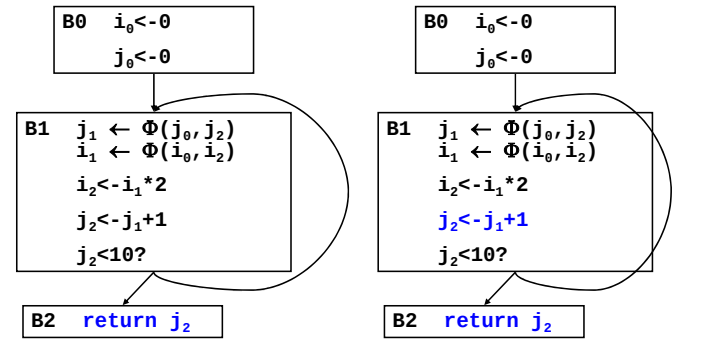
Aggressive Dead Code Elimination

Assume a stmt is dead until proven otherwise.

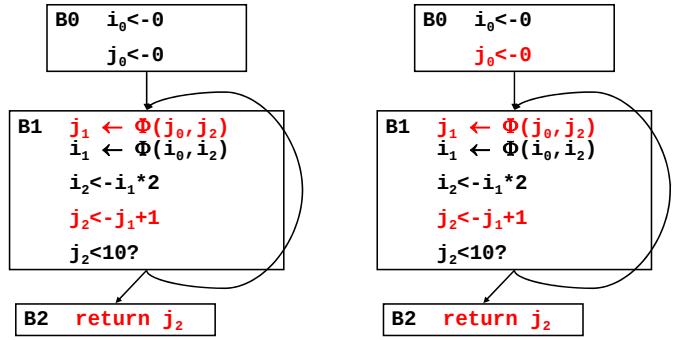
```
init:
   mark as live all stmts that have side-effects:
       - I/O
       - stores into memory
       - returns
       - calls a function that MIGHT have side-effects
   As we mark S live, insert S.defs into W

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


Example DCE



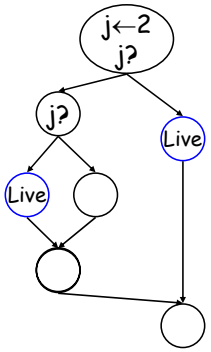
Example DCE



Problem!

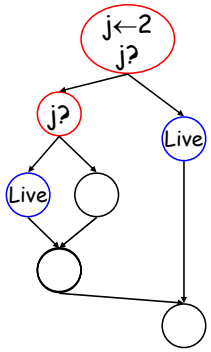
Fixing DCE

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



Fixing DCE

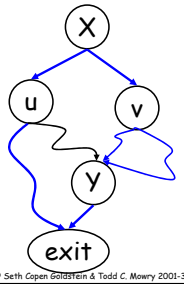
If S is live, then
  If T determines if S can execute, T should be live



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.

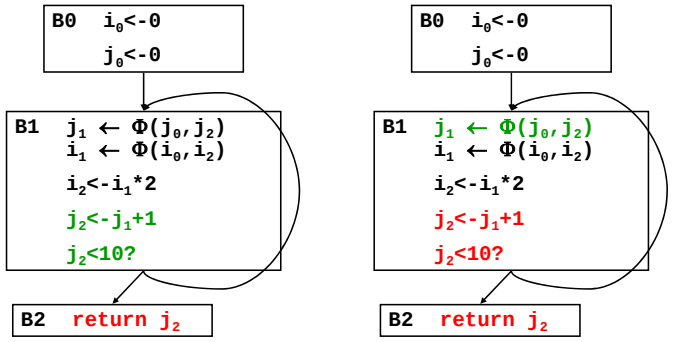


Aggressive Dead Code Elimination

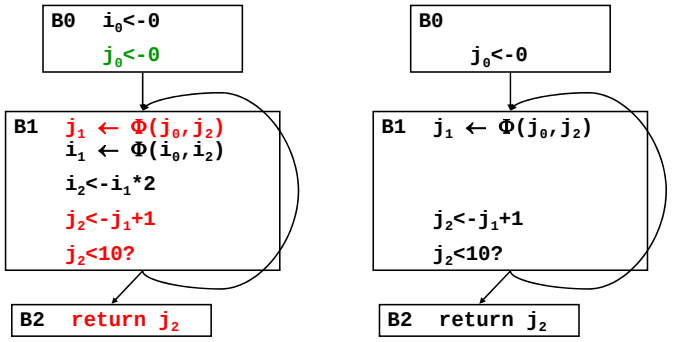
Assume a stmt is dead until proven otherwise.

```
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
}
```

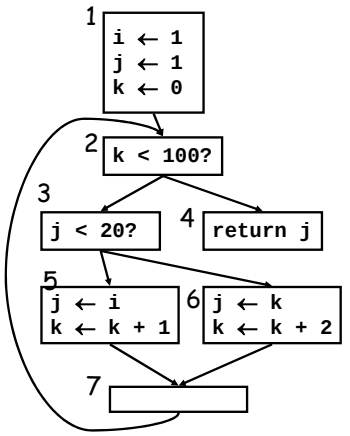
Example DCE



Example DCE

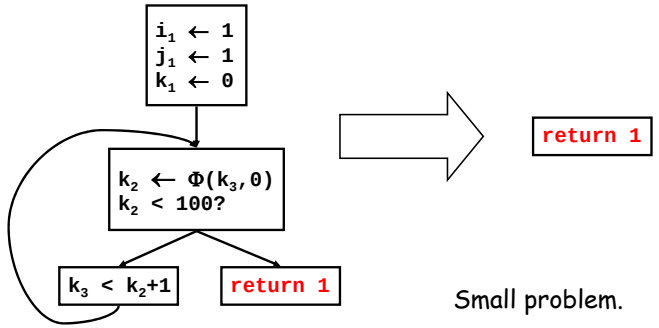


CCP Example



- 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

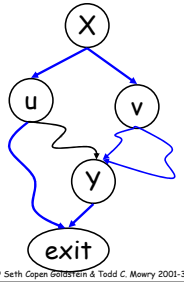


Small problem.

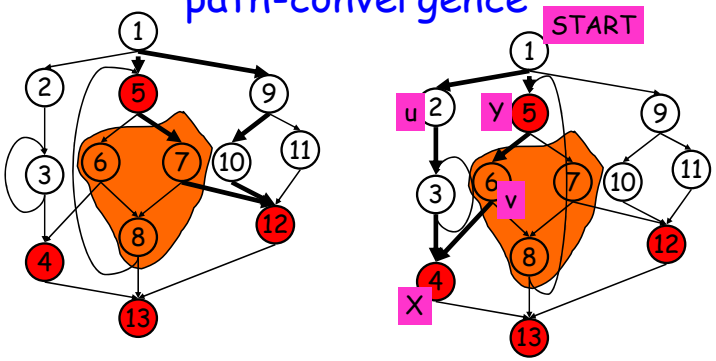
Finding the CDG

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

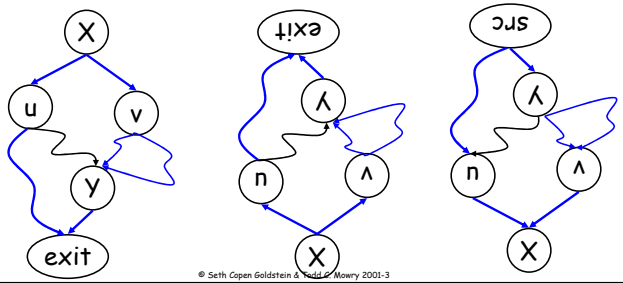


Any ideas?

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

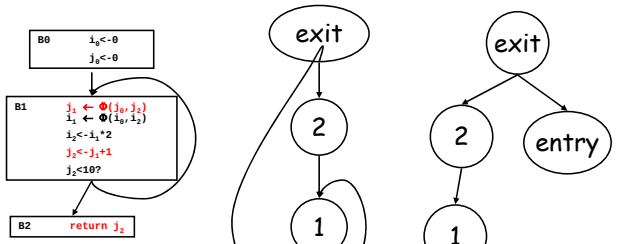
IOW, X can determine whether or not Y is executed.



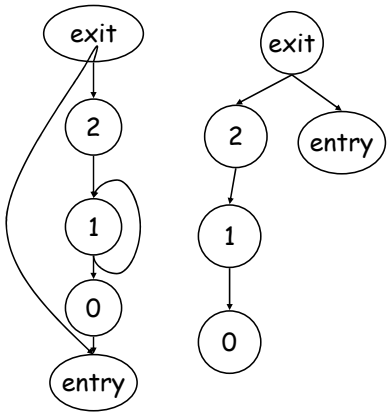
Finding the CDG

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

CDG of example



CDG of example



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

