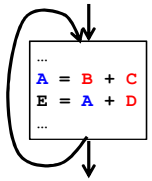
Lecture 8

Static Single Assignment (SSA)

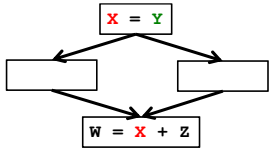
[ALSU 6.2.4]

Recurring Theme: Where Is a Variable Defined or Used?

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

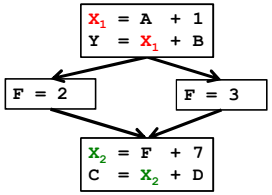


- Example: Copy Propagation
  - For a given use of X:
    - Are all reaching definitions of X:
      - copies from same variable: e.g., X = Y
    - Where Y is not redefined since that copy?
  - If so, substitute use of X with use of Y



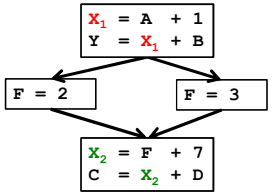
- It would be nice if we could *traverse directly* between related uses and def's
  - this would enable a form of *sparse* code analysis (skip over "don't care" cases)

Appearances of Same Variable Name May Be Unrelated



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

Definition-Use and Use-Definition Chains



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

Unfortunately DU and UD Chains Can Be Expensive

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

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

One solution: limit each variable to ONE definition site

Unfortunately DU and UD Chains Can Be Expensive

```
foo(int i, int j) {
  …
  switch (i) {
  case 0: x=3; break;
  case 1: x=1; break;
  case 2: x=6;
  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;
  }
  …
```

One solution: limit each variable to ONE definition site

Static Single Assignment (SSA)

- Static single assignment is an IR where every variable is assigned a value at most once in the program text
- Easy for a basic block (reminiscent of Value Numbering):
  - Visit each instruction in program order:
    - LHS: assign to a *fresh version* of the variable
    - RHS: use the *most recent version* of each variable

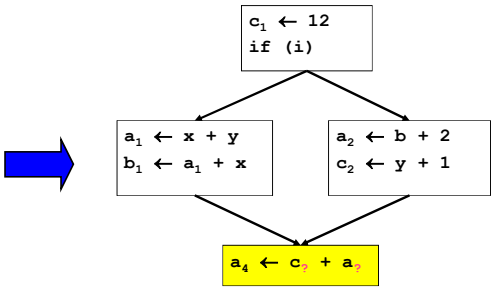
```
a ← x + y
b ← a + x
a ← b + 2
c ← y + 1
a ← c + a
```

⇒

```
a1 ← x + y
b1 ← a1 + x
a2 ← b1 + 2
c1 ← y + 1
a3 ← c1 + a2
```

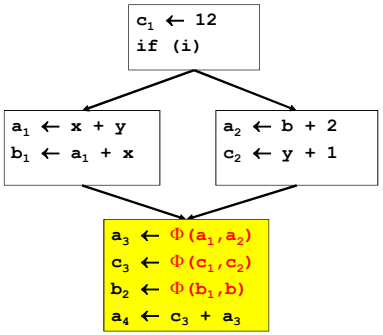

What about Joins in the CFG?

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



→ Use a notational convention (fiction): a Φ function

Merging at Joins: the Φ function



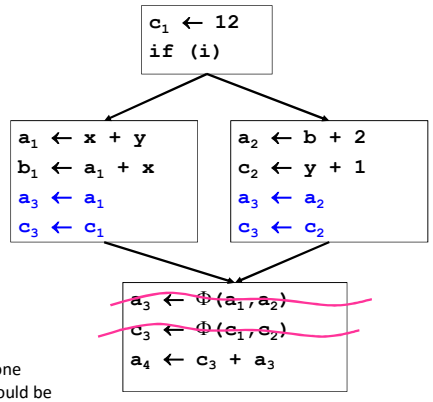
The Φ function

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

$$x_{new} \leftarrow \Phi(x_1, x_2, x_3, \ldots, x_p)$$

- How do we choose which x_i to use?
  - We don't really care!

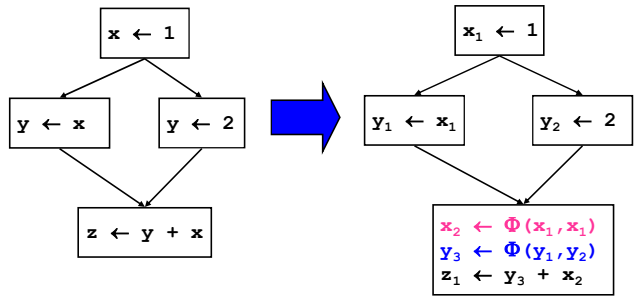
"Implementing" Φ



Never really done this way, but could be

Trivial SSA

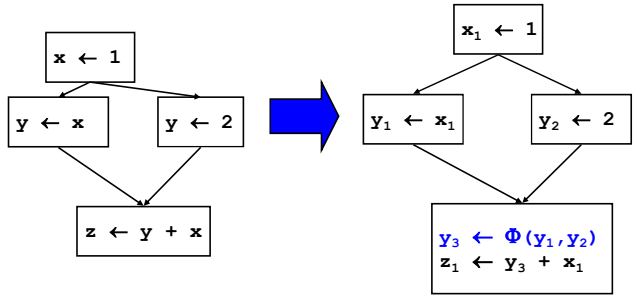
- Each assignment generates a fresh variable
- At each join point insert Φ functions for all live variables



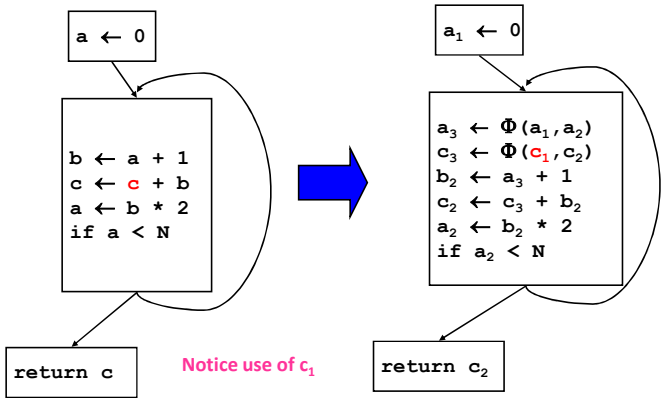
In general, too many Φ functions inserted

Minimal SSA

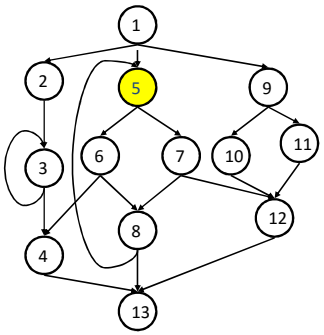
- Each assignment generates a fresh variable
- At each join point insert Φ functions for all live variables with multiple outstanding defs



Another Example



When Do We Insert Φ?

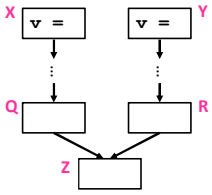


Control Flow Graph (CFG)

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

When do we insert Φ?

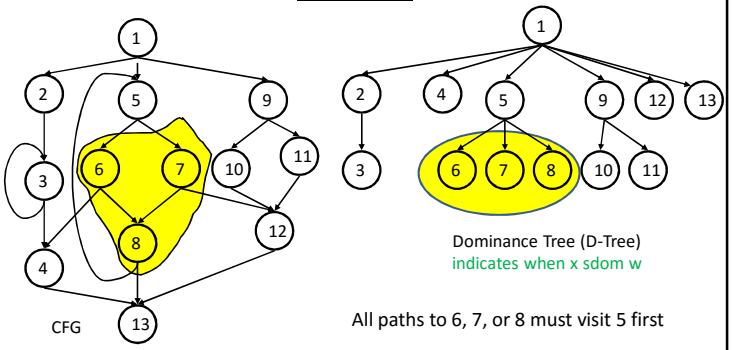
- We insert a Φ function for variable **v** in block Z iff:
  - **v** was defined more than once before
    - (i.e., **v** defined in X and Y AND X ≠ Y)
  - There exists a non-empty path from X to Z, P_{xz}, and a non-empty path from Y to Z, P_{yz}, s.t.
    - $P_{xz} \cap P_{yz} = \{ Z \}$
      (Z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



Path Convergence

- Entry block contains an implicit def of all vars
- Note: **v** = Φ(…) is a def of **v**

Dominance



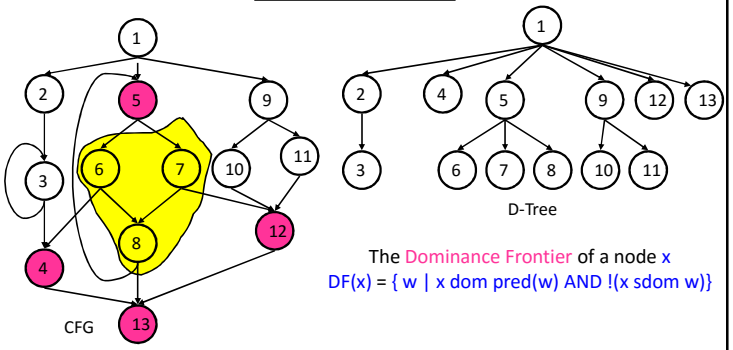
Dominance Tree (D-Tree)
indicates when x sdom w

All paths to 6, 7, or 8 must visit 5 first

x strictly dominates w (x sdom w) iff impossible to reach w without passing through x first

x dominates w (x dom w) iff x sdom w OR x = w

Dominance Frontier

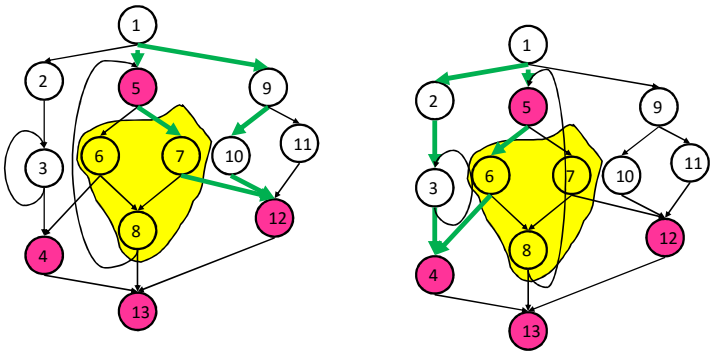


The Dominance Frontier of a node x
DF(x) = { w | x dom pred(w) AND !(x sdom w)}

x strictly dominates w (x sdom w) iff impossible to reach w without passing through x first

x dominates w (x dom w) iff x sdom w OR x = w

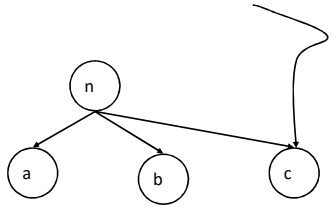
Dominance Frontier and Path Convergence



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

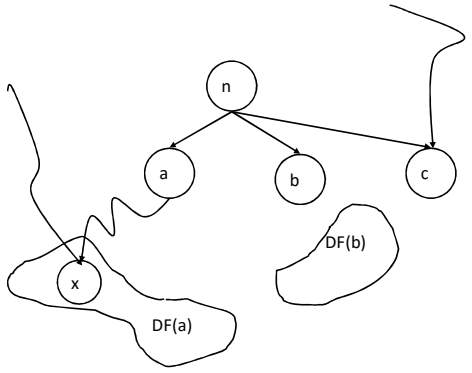
Computing DF(n)

DF(x) = { w | x dom pred(w) AND !(x sdom w)}



n dom n
n dom a
n dom b
!(n dom c)

Computing DF(n)



n dom n
n dom a
n dom b
!(n dom c)

Computing the Dominance Frontier

DF(n) = { w | n dom pred(w) AND !(n sdom w)}

```
compute-DF(n)
        S = {}
        foreach node c in succ[n]
                if !(n sdom c)
                        S = S ∪ { c }          e.g., node c on previous slide
        foreach child a of n in D-tree
                compute-DF(a)
                foreach x in DF[a]
                        if !(n dom x)
                                S = S ∪ { x }      e.g., node x on previous slide
        DF[n] = S
```


Using Dominance Frontier to Compute SSA

- Place all Φ()
- Rename all variables

Using Dominance Frontier to Place Φ()

- Gather all the defsites of every variable
- Then, for every variable
  - foreach defsite
    - foreach node in DominanceFrontier(defsite)
      - if we haven't put Φ() in node, then put one in
      - if this node didn't define the variable before, then add this node to the defsites (because Φ counts as def)
- This essentially computes the Iterated Dominance Frontier on the fly, inserting the minimal number of Φ() neccesary

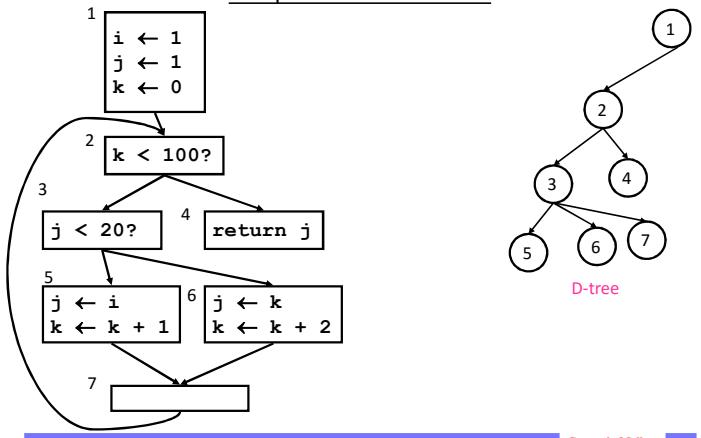
Using Dominance Frontier to Place Φ(): Algorithm

```
foreach node n {
   foreach variable v defined in n {
     orig[n] ∪= {v}          /* variables defined in basic block n */
     defsites[v] ∪= {n}      /* basic blocks that define variable v */
   }
}
foreach variable v {
   W = defsites[v]                          /* work list of basic blocks */
   while W not empty {
     n = remove node from W
     foreach y in DF[n]
       if y ∉ PHI[v] {
         insert "v ← Φ(v,v,…)" at top of y
         PHI[v] = PHI[v] ∪ {y}              /* BBs containing a Φ for v */
         if v ∉ orig[y]: W = W ∪ {y}        /* add BB to work list */
       }
   }
}
```

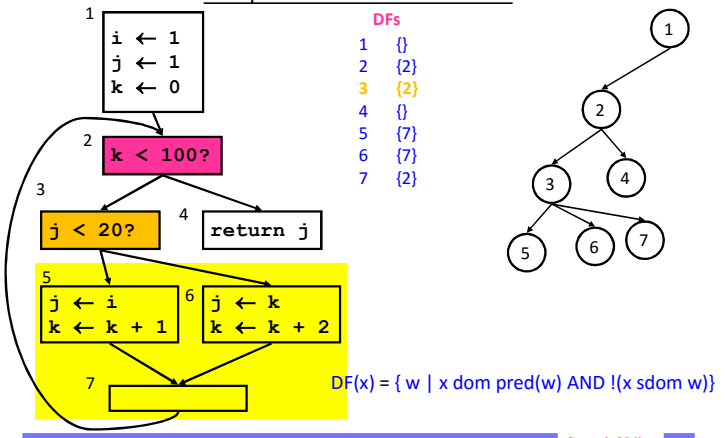

Renaming Variables

- Algorithm:
  - Walk the D-tree, renaming variables as you go
  - Replace uses with more recent renamed def
- For straight-line code this is easy
- What if there are branches and joins?
  - use the closest def such that the def is above the use in the D-tree
- Easy implementation:
  - for each var: call 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 Dominance Tree

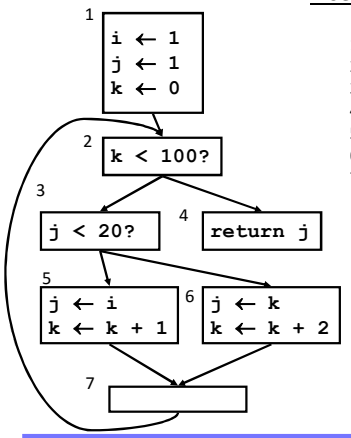


Compute Dominance Frontiers



DF(x) = { w | x dom pred(w) AND !(x sdom w)}

Insert Φ()



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

orig[n] = variables defined in n

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

var i: W={1} DF{}

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

DF{2} DF{5}

Insert Φ()

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

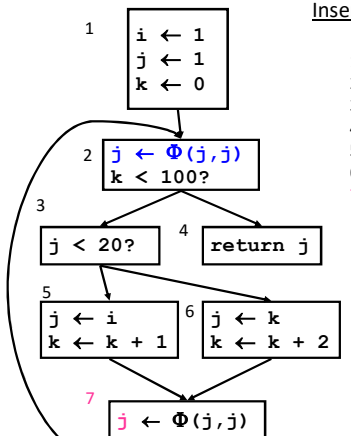
PHI[v]	
j	{7}
k	{}

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

var j: W={5,6}

DF{2} DF{5}

Insert Φ()



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

PHI[v]	
j	{2,7}
k	{}

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

var j: W={7,6}

DF{2} DF{2} DF{7}

Insert Φ()

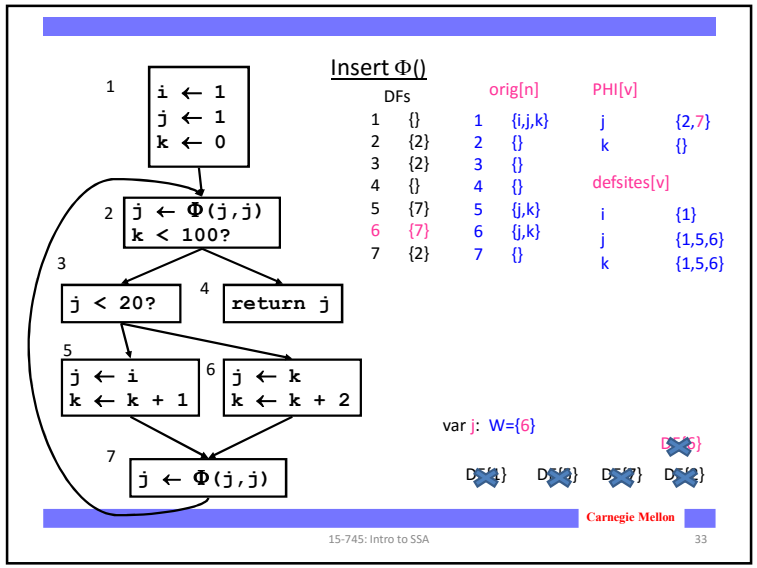
	DFs		orig[n]
1	{}	1	{i,j,k}
2	{2}	2	{}
3	{2}	3	{}
4	{}	4	{}
5	{7}	5	{j,k}
6	{7}	6	{j,k}
7	{2}	7	{}

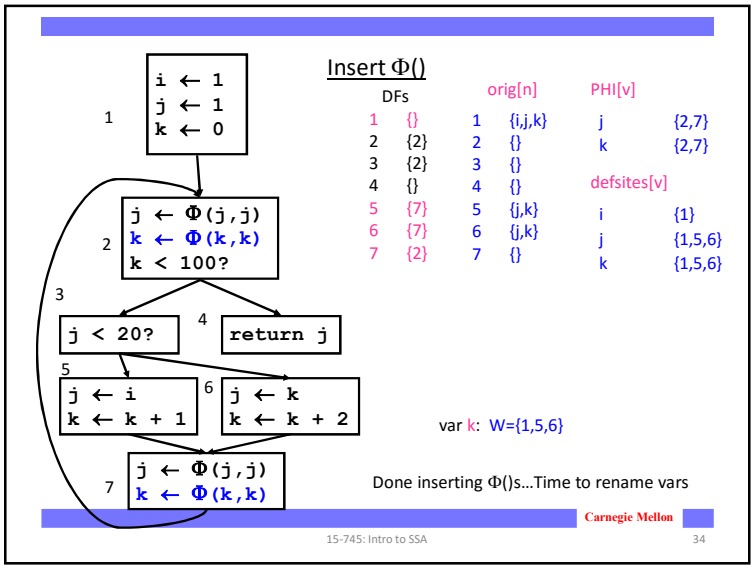
PHI[v]	
j	{2,7}
k	{}

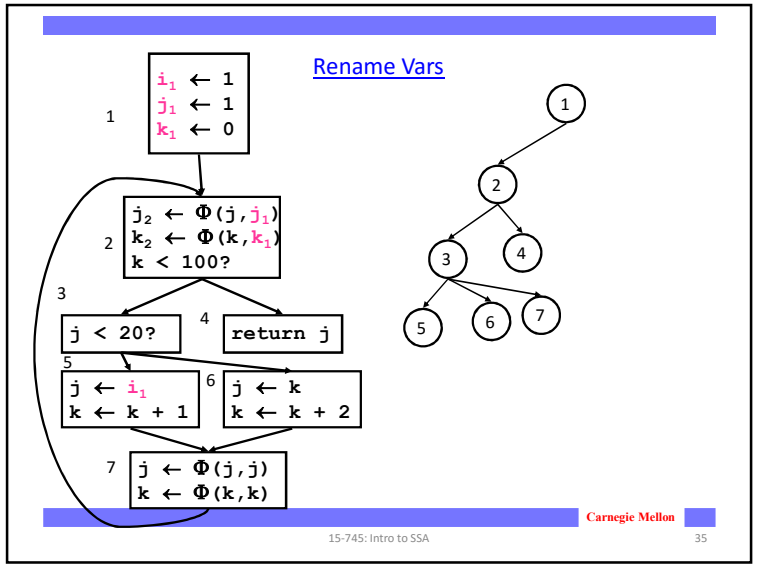
defsites[v]	
i	{1}
j	{1,5,6}
k	{1,5,6}

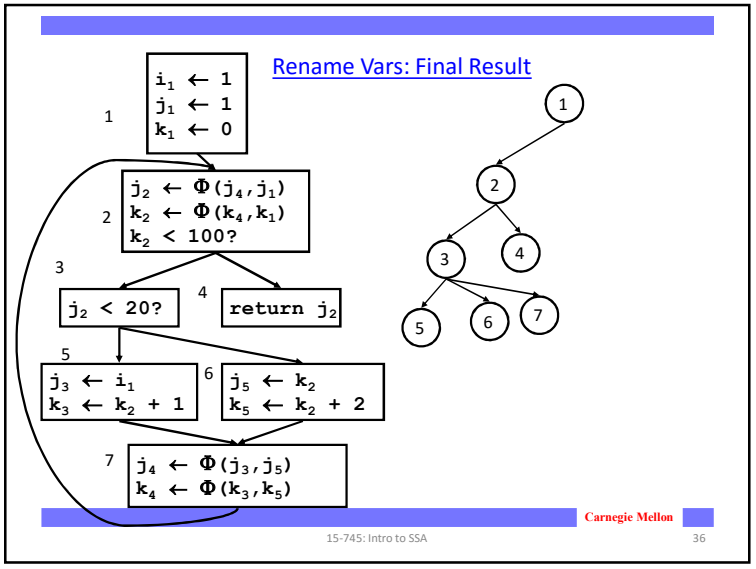
var j: W={2,6}

DF{2} DF{2} DF{7} DF{2}


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 {}
PHI[v]
j {2,7}
k {}
defsites[v]
i {1}
j {1,5,6}
k {1,5,6}
1 i ← 1 j ← 1 k ← 0
2 j ← Φ(j,j) k < 100?
3 j < 20?
4 return j
5 j ← i k ← k + 1
6 j ← k k ← k + 2
7 j ← Φ(j,j)
var j: W={6}
D[1] D[2] D[7] D[2]
Carnegie Mellon
15-745: Intro to SSA
33


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 {}
PHI[v]
j {2,7}
k {2,7}
defsites[v]
i {1}
j {1,5,6}
k {1,5,6}
1 i ← 1 j ← 1 k ← 0
2 j ← Φ(j,j) k ← Φ(k,k) k < 100?
3 j < 20?
4 return j
5 j ← i k ← k + 1
6 j ← k k ← k + 2
7 j ← Φ(j,j) k ← Φ(k,k)
var k: W={1,5,6}
Done inserting Φ()s…Time to rename vars
Carnegie Mellon
15-745: Intro to SSA
34


Rename Vars
1 i1 ← 1 j1 ← 1 k1 ← 0
2 j2 ← Φ(j,j1) k2 ← Φ(k,k1) 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
Carnegie Mellon
15-745: Intro to SSA
35


Rename Vars: Final Result
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
Carnegie Mellon
15-745: Intro to SSA
36

SSA Properties

- Only 1 assignment per variable
- Definitions dominate uses

Wednesday's Class

- Loop Invariant Code Motion [ALSU 9.6]
- Assignment 1 due midnight