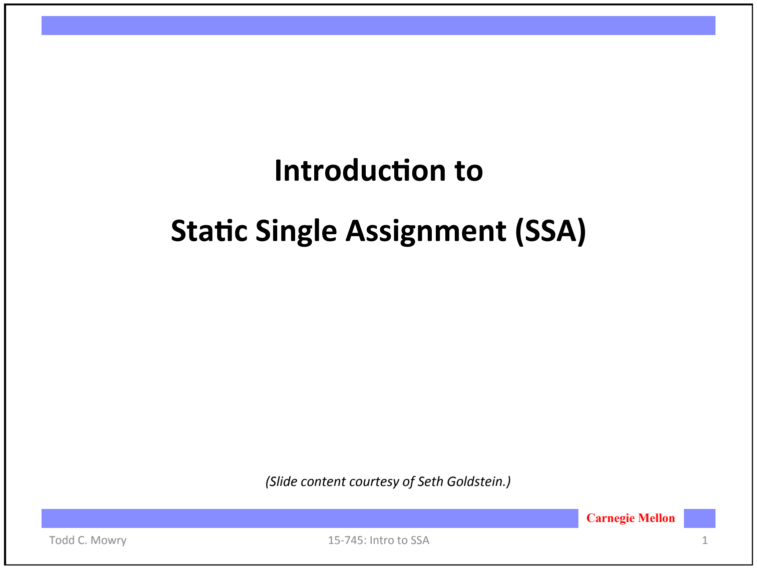
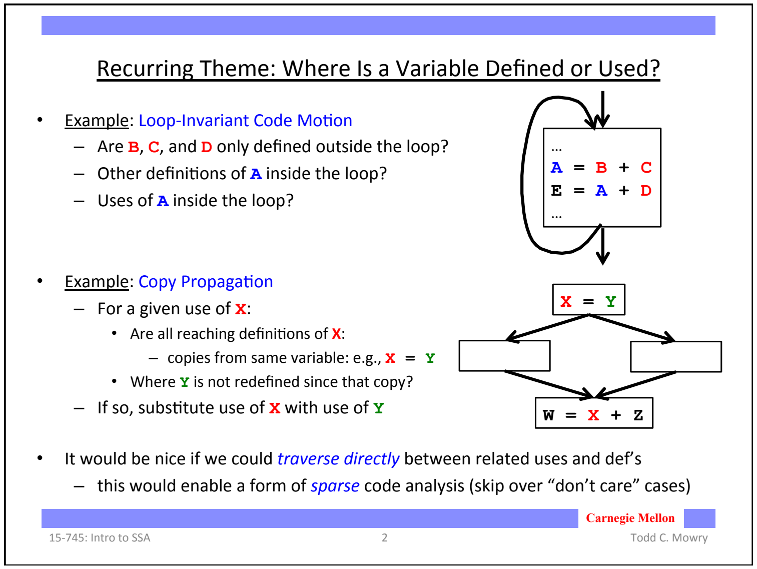


Introduction to

Static Single Assignment (SSA)

(Slide content courtesy of Seth Goldstein.)



Recurring Theme: Where Is a Variable Defined or Used?

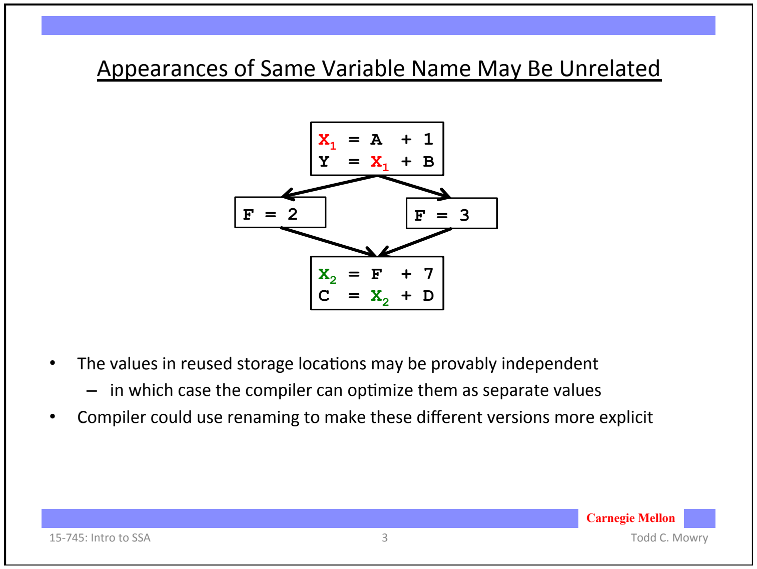
- Example: Loop-Invariant Code Motion
  - Are B, C, and D only defined outside the loop?
  - Other definitions of A inside the loop?
  - Uses of A inside the loop?

```
…
A = B + C
E = A + D
…
```

- 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

```
X = Y
W = X + Z
```

- 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)



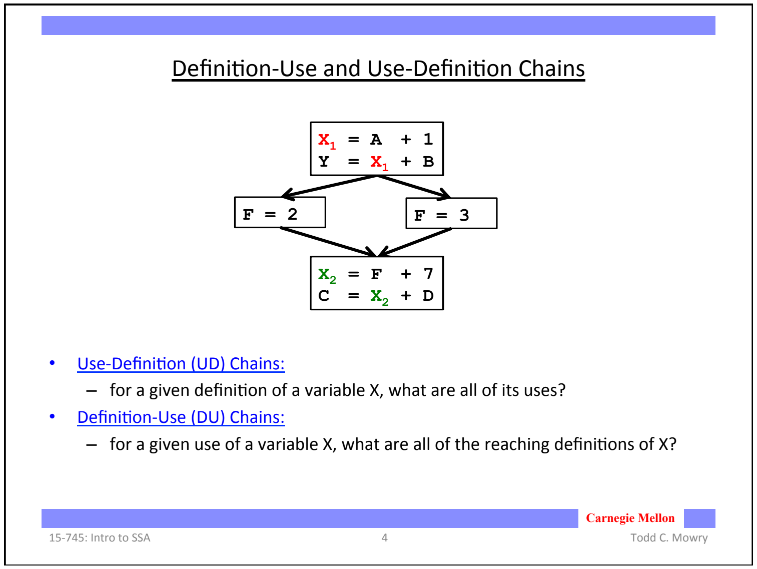
Appearances of Same Variable Name May Be Unrelated

$X_1 = A + 1$
$Y = X_1 + B$

$F = 2$ $F = 3$

$X_2 = F + 7$
$C = X_2 + D$

- 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



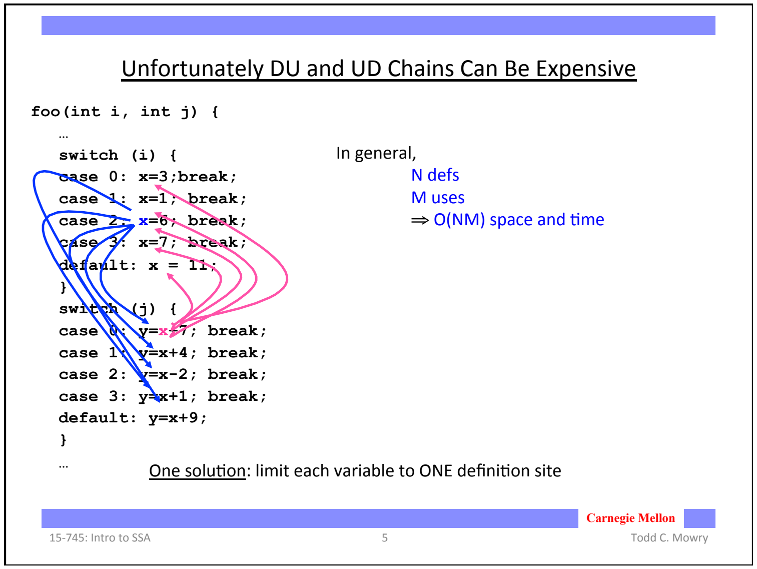
Definition-Use and Use-Definition Chains

$X_1 = A + 1$
$Y = X_1 + B$

$F = 2$ $F = 3$

$X_2 = F + 7$
$C = X_2 + D$

- 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?

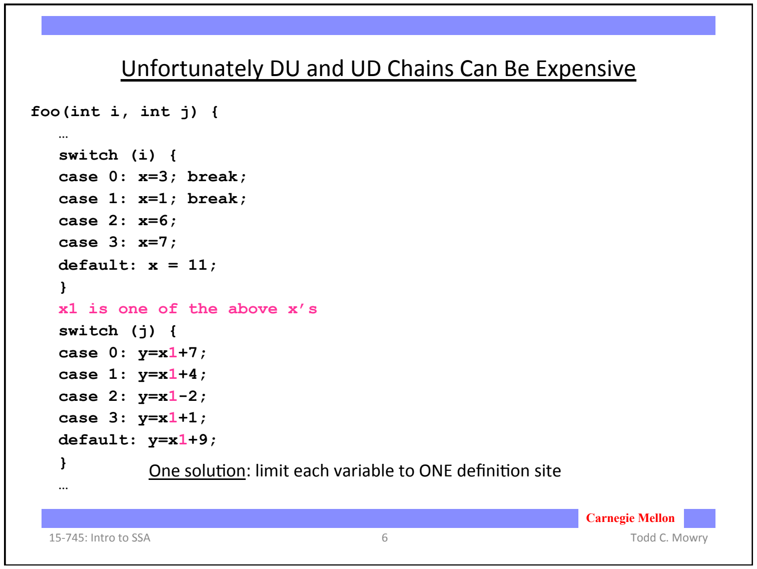


Unfortunately DU and UD Chains Can Be Expensive

```
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=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;
  }
  …
```

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

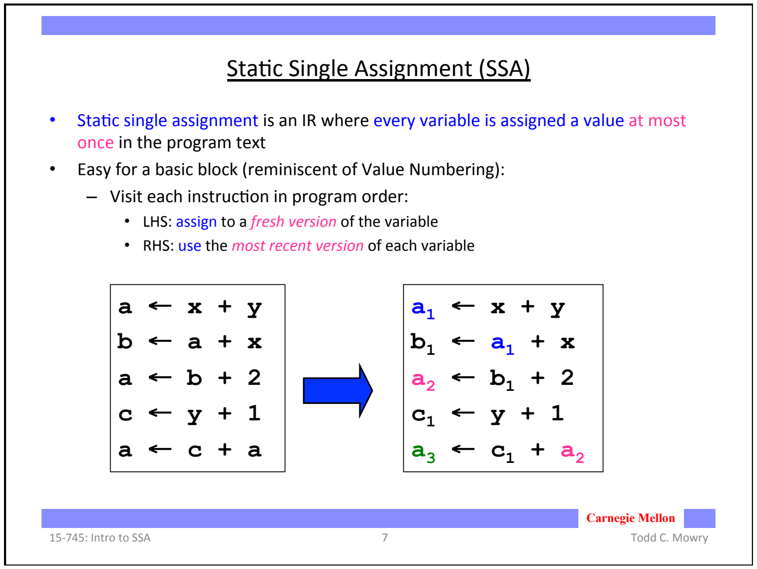
One solution: limit each variable to ONE definition site



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

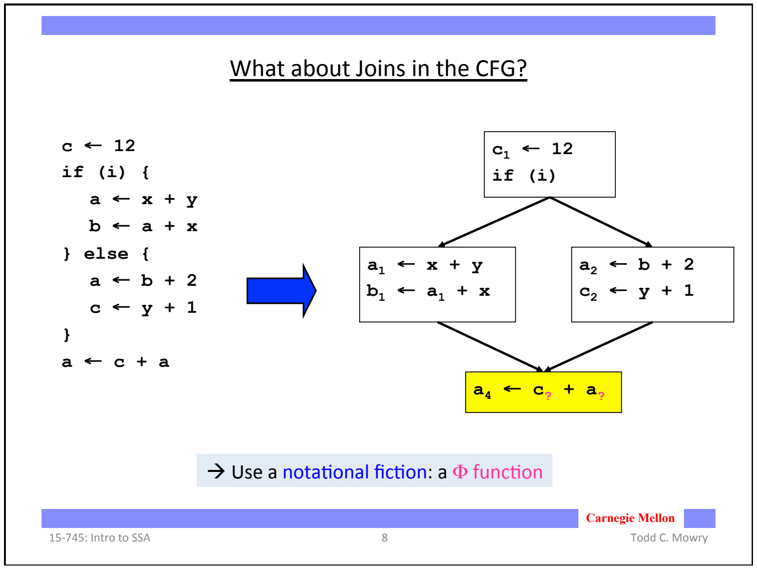


Static Single Assignment (SSA)

- 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

$a \leftarrow x + y$
$b \leftarrow a + x$
$a \leftarrow b + 2$
$c \leftarrow y + 1$
$a \leftarrow c + a$

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



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
```

$c_1 \leftarrow 12$
if (i)

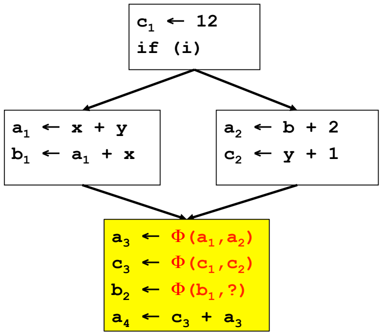
$a_1 \leftarrow x + y$
$b_1 \leftarrow a_1 + x$

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

$a_4 \leftarrow c_? + a_?$

→ Use a notational fiction: a Φ function

Merging at Joins: the Φ function



$c_1 \leftarrow 12$
if (i)

$a_1 \leftarrow x + y$
$b_1 \leftarrow a_1 + x$

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

$a_3 \leftarrow \Phi(a_1, a_2)$
$c_3 \leftarrow \Phi(c_1, c_2)$
$b_2 \leftarrow \Phi(b_1, ?)$
$a_4 \leftarrow c_3 + a_3$

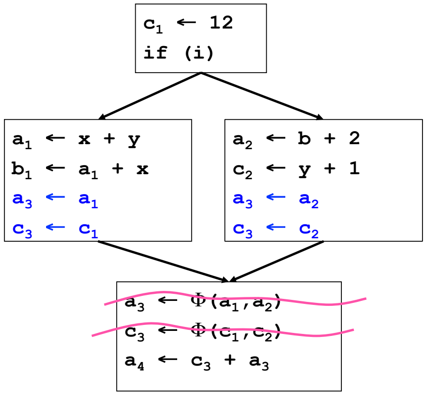
The Φ function

- 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.

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

- 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



$c_1 \leftarrow 12$
if (i)

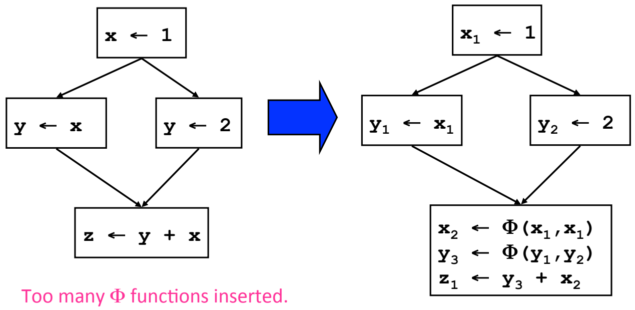
$a_1 \leftarrow x + y$
$b_1 \leftarrow a_1 + x$
$a_3 \leftarrow a_1$
$c_3 \leftarrow c_1$

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

~~$a_3 \leftarrow \Phi(a_1, a_2)$~~
~~$c_3 \leftarrow \Phi(c_1, c_2)$~~
$a_4 \leftarrow c_3 + a_3$

Trivial SSA

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



$x \leftarrow 1$

$y \leftarrow x$

$y \leftarrow 2$

$z \leftarrow y + x$

$x_1 \leftarrow 1$

$y_1 \leftarrow x_1$

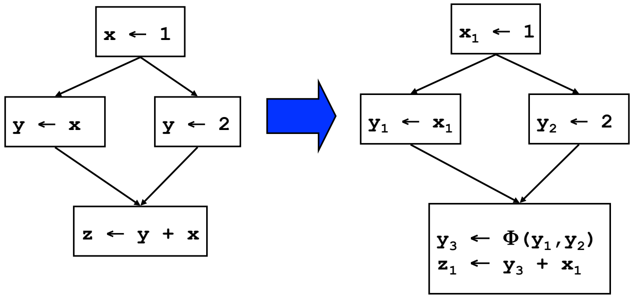
$y_2 \leftarrow 2$

$x_2 \leftarrow \Phi(x_1, x_1)$
$y_3 \leftarrow \Phi(y_1, y_2)$
$z_1 \leftarrow y_3 + x_2$

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.



$x \leftarrow 1$

$y \leftarrow x$

$y \leftarrow 2$

$z \leftarrow y + x$

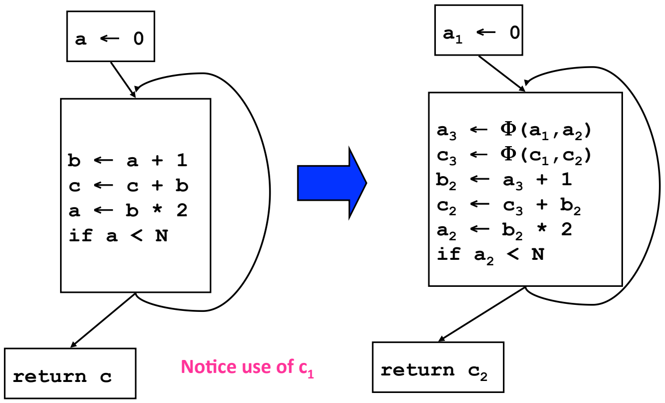
$x_1 \leftarrow 1$

$y_1 \leftarrow x_1$

$y_2 \leftarrow 2$

$y_3 \leftarrow \Phi(y_1, y_2)$
$z_1 \leftarrow y_3 + x_1$

Another Example



$a \leftarrow 0$

```
b ← a + 1
c ← c + b
a ← b * 2
if a < N
```

return c

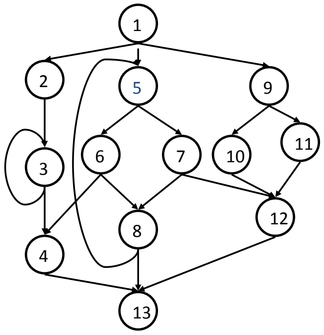
$a_1 \leftarrow 0$

$a_3 \leftarrow \Phi(a_1, a_2)$
$c_3 \leftarrow \Phi(c_1, c_2)$
$b_2 \leftarrow a_3 + 1$
$c_2 \leftarrow c_3 + b_2$
$a_2 \leftarrow b_2 * 2$
if $a_2 < N$

return c_2

Notice use of c_1

When Do We Insert Φ?



CFG

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

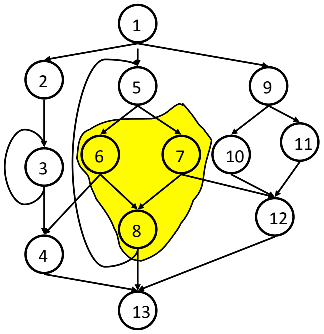
When do we insert Φ?

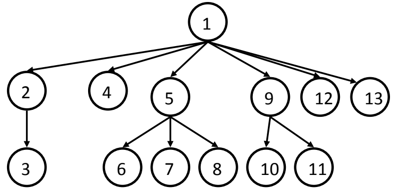
- We insert a Φ function for variable **A** in block Z iff:
  - **A** was defined more than once before
    - (i.e., **A** defined in **X** and Y AND X ≠ Y)
  - There exists a non-empty path from x to z, P_{xz}, and a non-empty path from y to z, P_{yz}, s.t.
    - $P_{xz} \cap P_{yz} = \{ z \}$
    - $z \notin P_{xq}$ or $z \notin P_{yr}$ where $P_{xz} = P_{xq} \to z$ and $P_{yz} = P_{yr} \to z$
- 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 i^{th} predecessor 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 algorithm to convert to SSA

Dominance





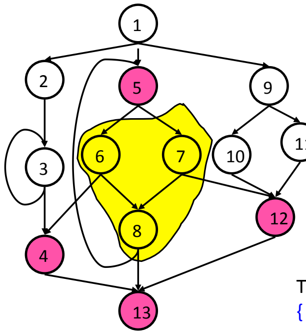
If there is a def of a in block 5, which nodes need a Φ()?

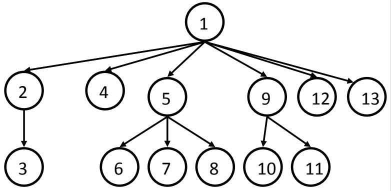
CFG

D-Tree

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

Dominance Frontier





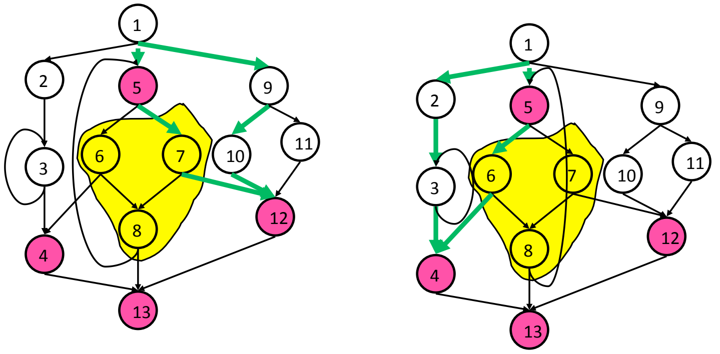
The Dominance Frontier of a node x =
{ w | x dom pred(w) AND !(x sdom w)}

CFG

D-Tree

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

Dominance Frontier and Path Convergence



Using Dominance Frontier to Compute SSA

- place all Φ()
- Rename all variables

Using Dominance Frontier to Place Φ()

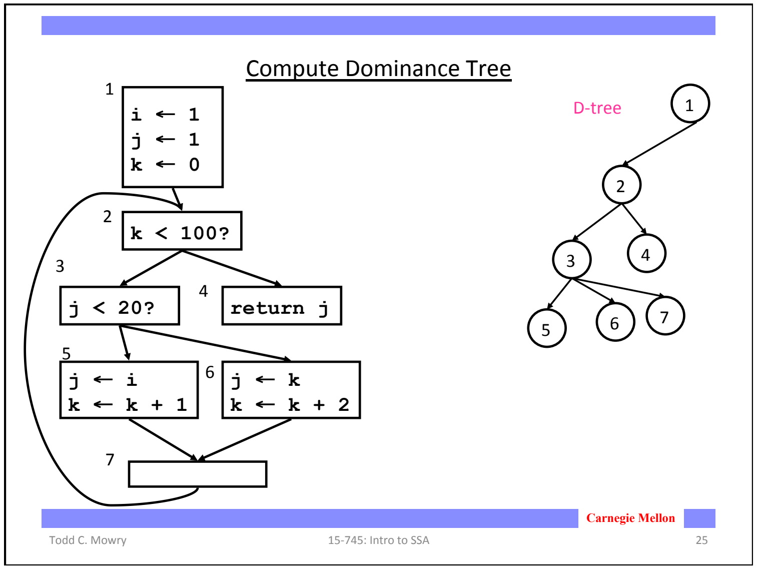
- Gather all the defsites of every variable
- Then, for every variable
  - foreach defsite
    - foreach node in DominanceFrontier(defsite)
      - if we haven't put Φ() in node, then put one in
      - if this node didn't define the variable before, then add this node to the defsites
- This essentially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

Using Dominance Frontier to Place Φ()

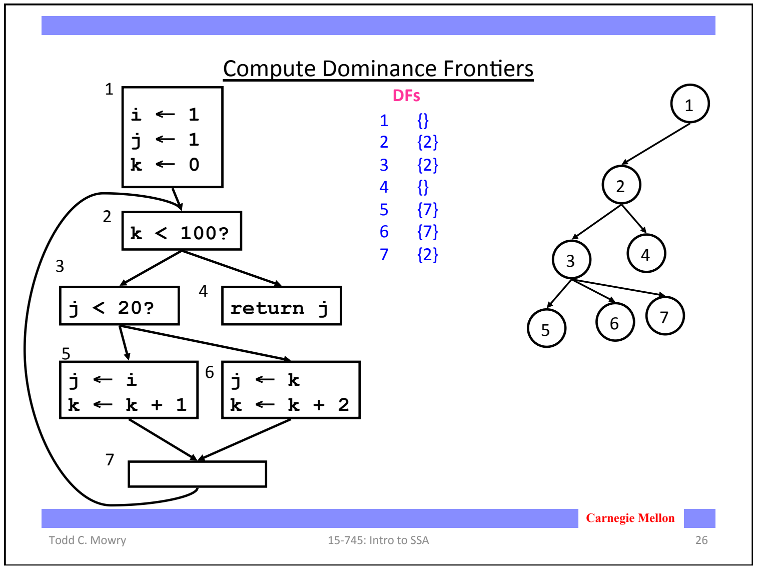
```
foreach node n {
   foreach variable v defined in n {
     orig[n] ∪= {v}
     defsites[v] ∪= {n}
   }
}
foreach variable v {
   W = defsites[v]
   while W not empty {
     n = remove node from W
     foreach y in DF[n]
     if y ∉ PHI[v] {
       insert "v ← Φ(v,v,…)" at top of y
       PHI[v] = PHI[v] ∪ {y}
       if v ∉ orig[y]: W = W ∪ {y}
     }
   }
}
```

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: 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

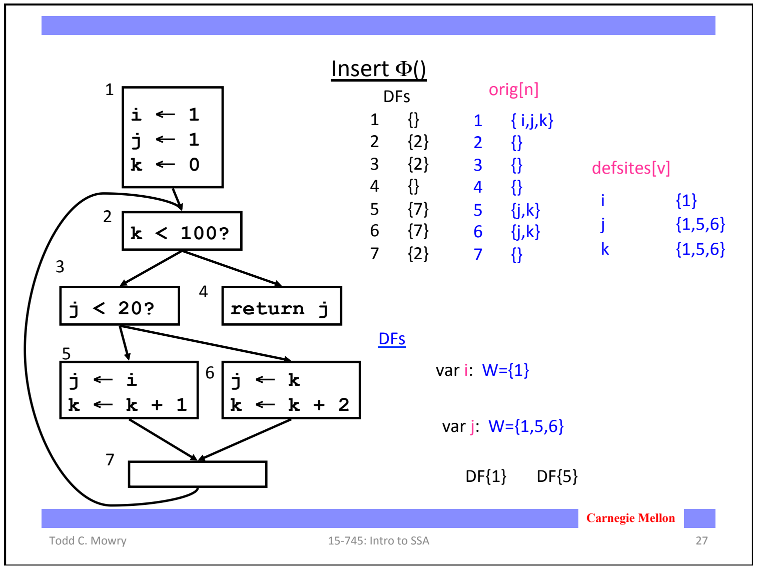


Compute Dominance Tree



Compute Dominance Frontiers

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



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	{}

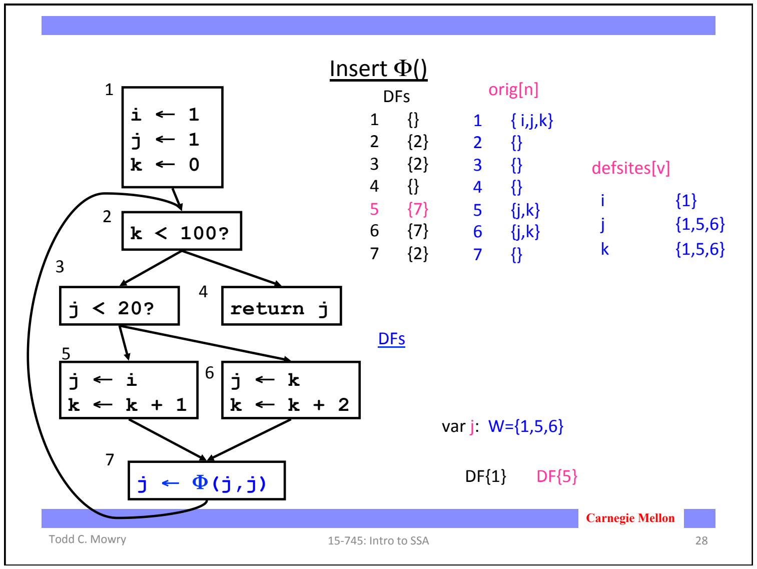
defsites[v]	
i	{1}
j	{1,5,6}
k	{1,5,6}

DFs

var i: W={1}

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

DF{1} DF{5}



Insert Φ()

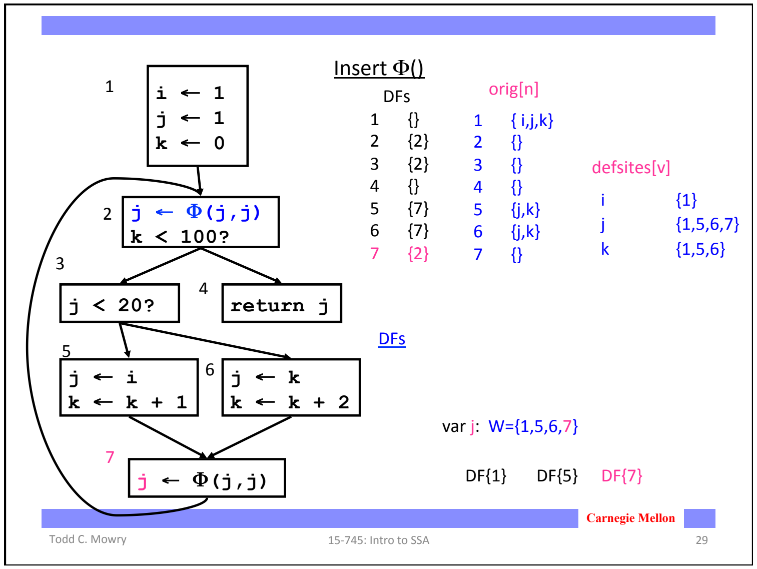
	DFs		orig[n]
1	{}	1	{ i,j,k}
2	{2}	2	{}
3	{2}	3	{}
4	{}	4	{}
5	{7}	5	{j,k}
6	{7}	6	{j,k}
7	{2}	7	{}

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

DFs

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

DF{1} DF{5}



Insert Φ()

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)

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	{}

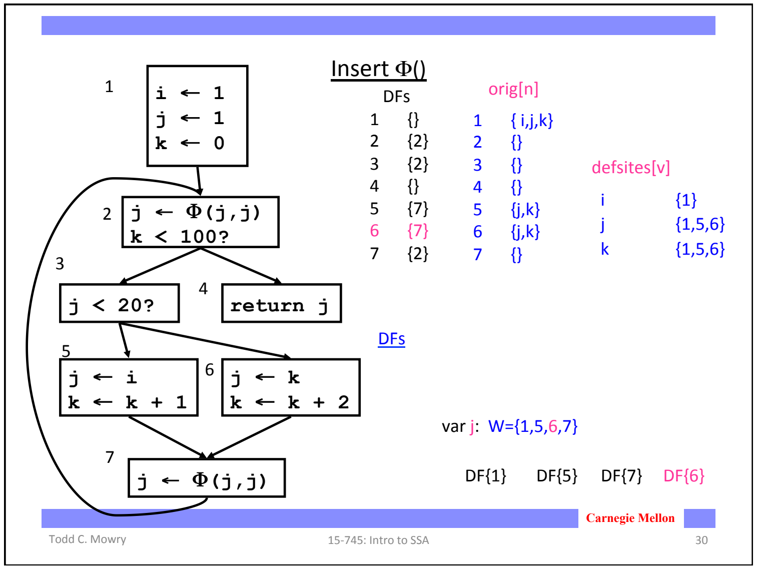
defsites[v]	
i	{1}
j	{1,5,6,7}
k	{1,5,6}

DFs

var j: W={1,5,6,7}

DF{1} DF{5} DF{7}

Carnegie Mellon



Insert Φ()

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)

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	{}

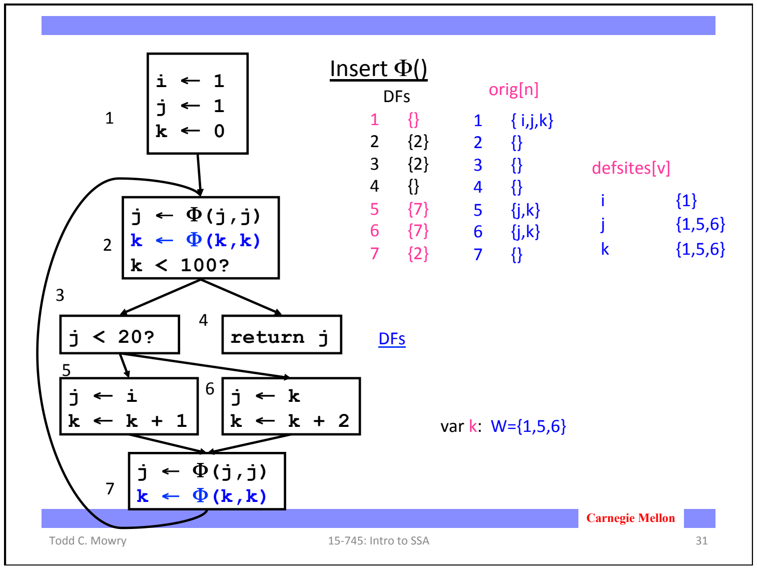
defsites[v]	
i	{1}
j	{1,5,6}
k	{1,5,6}

DFs

var j: W={1,5,6,7}

DF{1} DF{5} DF{7} DF{6}

Carnegie Mellon



Insert Φ()

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)

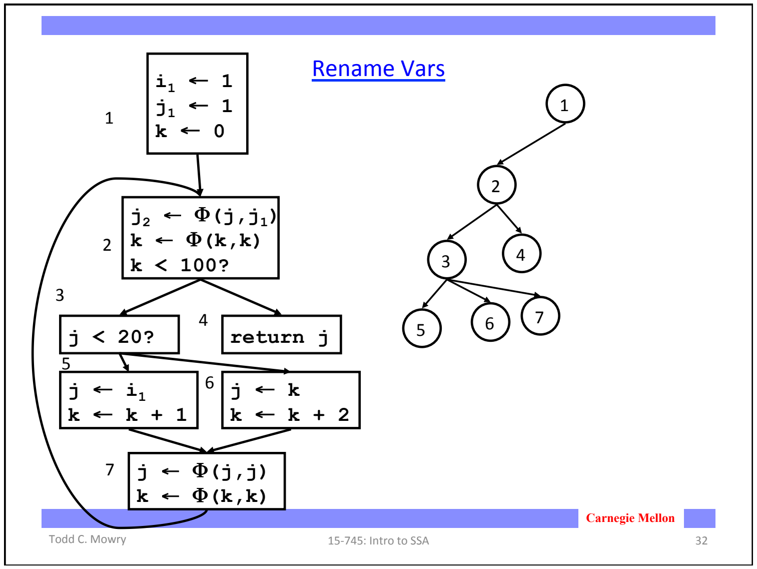
DFs		orig[n]	
1	{}	1	{ i,j,k}
2	{2}	2	{}
3	{2}	3	{}
4	{}	4	{}
5	{7}	5	{j,k}
6	{7}	6	{j,k}
7	{2}	7	{}

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

DFs

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

Carnegie Mellon



Rename Vars

1
i_1 ← 1
j_1 ← 1
k ← 0

2
j_2 ← Φ(j,j_1)
k ← Φ(k,k)
k < 100?

3
j < 20?

4
return j

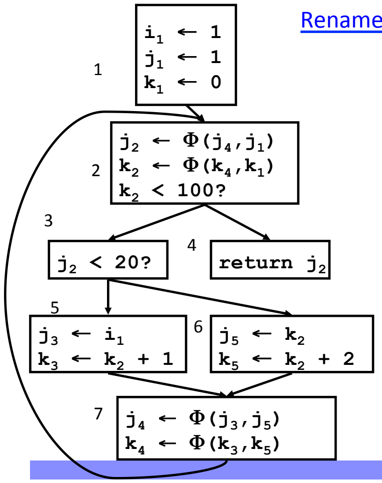
5
j ← i_1
k ← k + 1

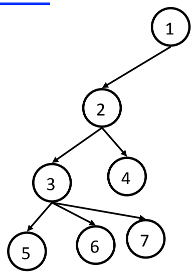
6
j ← k
k ← k + 2

7
j ← Φ(j,j)
k ← Φ(k,k)

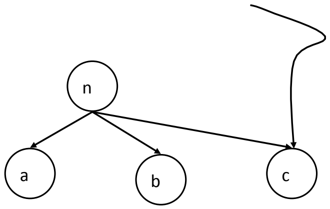
Carnegie Mellon

Rename Vars



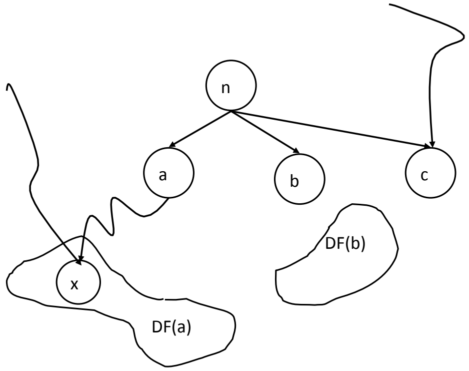


Computing DF(n)



n dom a
n dom b
!n dom c

Computing DF(n)

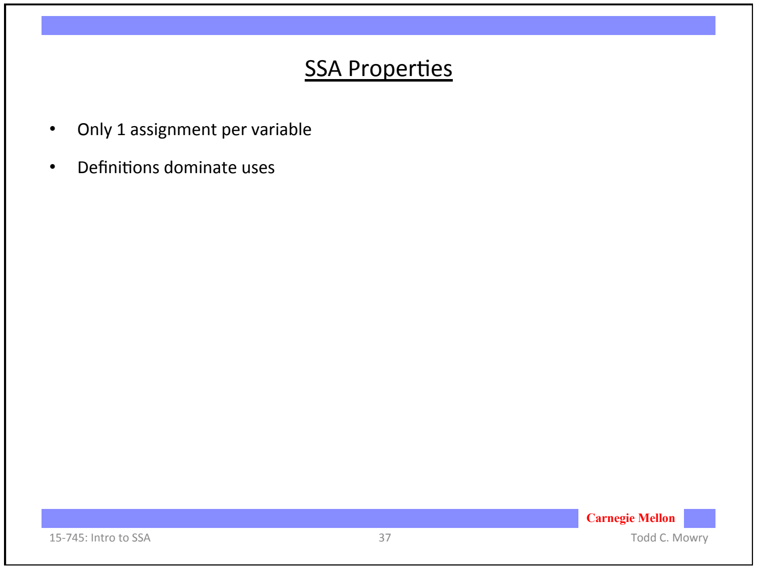


n dom a
n dom b
!n dom c

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 dom w
                S = S ∪ { w }
    DF[n] = S
```



SSA Properties

- Only 1 assignment per variable
- Definitions dominate uses