

15-745 Lecture 2

Dataflow Analysis Basic Blocks Related Optimizations SSA

Copyright © Seth Copen Goldstein 2005-8

Dataflow Analysis

- Last time we looked at code transformations
 - Constant propagation
 - Copy propagation
 - Common sub-expression elimination
 - ...
- Today, dataflow analysis:
 - How to determine if it is **legal** to perform such an optimization
 - (Not doing analysis to determine if it is **beneficial**)

A sample program

```
int fib10(void) {
  int n = 10;
  int older = 0;
  int old = 1;
  ir What are those numbers?
  int i;

  if (n <= 1) return n;
  for (i = 2; i<n; i++) {
    result = old + older;
    older = old;
    old = result;
  }
  return result;
}
```

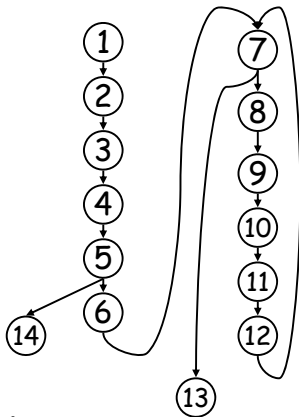
```
1:  n <- 10
2:  older <- 0
3:  old <- 1
4:  result <- 0
5:  if n <= 1 goto 14
6:  i <- 2
7:  if i > n goto 13
8:  result <- old + older
9:  older <- old
10: old <- result
11: i <- i + 1
12: JUMP 7
13: return result
14: return n
```

Simple Constant Propagation

- Can we do SCP?
 - How do we recognize it?
 - What aren't we doing?
 - Metanote:
 - keep opts simple!
 - Use combined power
- ```
1: n <- 10
2: older <- 0
3: old <- 1
4: result <- 0
5: if n <= 1 goto 14
6: i <- 2
7: if i > n goto 13
8: result <- old + older
9: older <- old
10: old <- result
11: i <- i + 1
12: JUMP 7
13: return result
14: return n
```

## Reaching Definitions

- A definition of variable  $v$  at program point  $d$  reaches program point  $u$  if there exists a path of control flow edges from  $d$  to  $u$  that does not contain a definition of  $v$ .



```

1: n <- 10
2: older <- 0
3: old <- 1
4: result <- 0
5: if n <= 1 goto 14
6: i <- 2
7: if i > n goto 13
8: result <- old + older
9: older <- old
10: old <- result
11: i <- i + 1
12: JUMP 7
13: return result
14: return n

```

## Reaching Definitions (ex)

- 1 reaches 5, 7, and 14

14, Really?

Meta-notes:

- (almost) always conservative
- only know what we know
- Keep it simple:
  - What opt(s), if run before this would help
  - What about:
 

```

1: x <- 0
2: if (false) x <- -1
3: ... x ...

```
- Does 1 reach 3?
- What opt changes this?

```

1: n <- 10
2: older <- 0
3: old <- 1
4: result <- 0
5: if n <= 1 goto 14
6: i <- 2
7: if i > n goto 13
8: result <- old + older
9: older <- old
10: old <- result
11: i <- i + 1
12: JUMP 7
13: return result
14: return n

```

## Calculating Reaching Definitions

- A definition of variable  $v$  at program point  $d$  reaches program point  $u$  if there exists a path of control flow edges from  $d$  to  $u$  that does not contain a definition of  $v$ .

- Build up RD stmt by stmt
- Stmt  $s$ , "d:  $v \leftarrow x \text{ op } y$ ", generates  $d$
- Stmt  $s$ , "d:  $v \leftarrow x \text{ op } y$ ", kills all other defs( $v$ )

Or,

- $Gen[s] = \{ d \}$
- $Kill[s] = defs(v) - \{ d \}$

## Gen and kill for each stmt

|                          | Gen | kill |
|--------------------------|-----|------|
| 1: n <- 10               | 1   |      |
| 2: older <- 0            | 2   | 9    |
| 3: old <- 1              | 3   | 10   |
| 4: result <- 0           | 4   | 8    |
| 5: if n <= 1 goto 14     |     |      |
| 6: i <- 2                | 6   | 11   |
| 7: if i > n goto 13      |     |      |
| 8: result <- old + older | 8   | 4    |
| 9: older <- old          | 9   | 2    |
| 10: old <- result        | 10  | 3    |
| 11: i <- i + 1           | 11  | 6    |
| 12: JUMP 7               |     |      |
| 13: return result        |     |      |
| 14: return n             |     |      |

How can we determine the defs that reach a node?

We can use:

- control flow information
- gen and kill info

## Computing in[n] and out[n]

- In[n]: the set of defs that reach the beginning of node n
- Out[n]: the set of defs that reach the end of node n

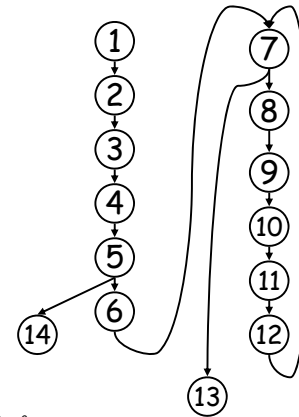
$$in[n] = \bigcup_{p \in pred[n]} out[p]$$

$$out[n] = gen[n] \cup (in[n] - kill[n])$$

- Initialize in[n]=out[n]={} for all n
- Solve iteratively

## What is pred[n]?

- Pred[n] are all nodes that can reach n in the control flow graph.
- E.g., pred[7] = { 6, 12 }



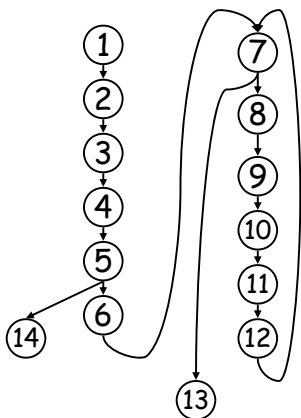
```

1: n <- 10
2: older <- 0
3: old <- 1
4: result <- 0
5: if n <= 1 goto 14
6: i <- 2
7: if i > n goto 13
8: result <- old + older
9: older <- old
10: old <- result
11: i <- i + 1
12: JUMP 7
13: return result
14: return n

```

## What order to eval nodes?

- Does it matter?
- Lets do: 1,2,3,4,5,14,6,7,13,8,9,10,11,12



```

1: n <- 10
2: older <- 0
3: old <- 1
4: result <- 0
5: if n <= 1 goto 14
6: i <- 2
7: if i > n goto 13
8: result <- old + older
9: older <- old
10: old <- result
11: i <- i + 1
12: JUMP 7
13: return result
14: return n

```

## Example:

- Order: 1,2,3,4,5,14,6,7,13,8,9,10,11,12
- $$in[n] = \bigcup_{p \in pred[n]} out[p] \quad out[n] = gen[n] \cup (in[n] - kill[n])$$

|                          | Gen | kill | in    | out |
|--------------------------|-----|------|-------|-----|
| 1: n <- 10               | 1   |      |       | 1   |
| 2: older <- 0            | 2   | 9    | 1     | 1,2 |
| 3: old <- 1              | 3   | 10   | 1,2,3 | 1-4 |
| 4: result <- 0           | 4   | 8    |       |     |
| 5: if n <= 1 goto 14     |     |      |       |     |
| 6: i <- 2                | 6   | 11   |       |     |
| 7: if i > n goto 13      |     |      |       |     |
| 8: result <- old + older | 8   | 4    |       |     |
| 9: older <- old          | 9   | 2    |       |     |
| 10: old <- result        | 10  | 3    |       |     |
| 11: i <- i + 1           | 11  | 6    |       |     |
| 12: JUMP 7               |     |      |       |     |
| 13: return result        |     |      |       |     |
| 14: return n             |     |      |       |     |

## Example (pass 1)

- Order: 1,2,3,4,5,14,6,7,13,8,9,10,11,12

$$in[n] = \bigcup_{p \in pred[n]} out[p] \quad out[n] = gen[n] \cup (in[n] - kill[n])$$

|                          | Gen | kill | in        | out       |
|--------------------------|-----|------|-----------|-----------|
| 1: n <- 10               | 1   |      |           | 1         |
| 2: older <- 0            | 2   | 9    | 1         | 1,2       |
| 3: old <- 1              | 3   | 10   | 1,2       | 1,2,3     |
| 4: result <- 0           | 4   | 8    | 1-3       | 1-4       |
| 5: if n <= 1 goto 14     |     |      | 1-4       | 1-4       |
| 6: i <- 2                | 6   | 11   | 1-4       | 1-4,6     |
| 7: if i > n goto 13      |     |      | 1-4,6     | 1-4,6     |
| 8: result <- old + older | 8   | 4    | 1-4,6     | 1-3,6,8   |
| 9: older <- old          | 9   | 2    | 1-3,6,8   | 1,3,6,8,9 |
| 10: old <- result        | 10  | 3    | 1,3,6,8,9 | 1,6,8-10  |
| 11: i <- i + 1           | 11  | 6    | 1,6,8-10  | 1,8-11    |
| 12: JUMP 7               |     |      | 1,8-11    | 1,8-11    |
| 13: return result        |     |      | 1-4,6     | 1-4,6     |
| 14: return n             |     |      | 1-4       | 1-4       |

## Example (pass 2)

- Order: 1,2,3,4,5,14,6,7,13,8,9,10,11,12

$$in[n] = \bigcup_{p \in pred[n]} out[p] \quad out[n] = gen[n] \cup (in[n] - kill[n])$$

|                          | Gen | kill | in         | out        |
|--------------------------|-----|------|------------|------------|
| 1: n <- 10               | 1   |      |            | 1          |
| 2: older <- 0            | 2   | 9    | 1          | 1,2        |
| 3: old <- 1              | 3   | 10   | 1,2        | 1,2,3      |
| 4: result <- 0           | 4   | 8    | 1-3        | 1-4        |
| 5: if n <= 1 goto 14     |     |      | 1-4        | 1-4        |
| 6: i <- 2                | 6   | 11   | 1-4        | 1-4,6      |
| 7: if i > n goto 13      |     |      | 1-4,6,8-11 | 1-4,6,8-11 |
| 8: result <- old + older | 8   | 4    | 1-4,6,8-11 | 1-3,6,8-11 |
| 9: older <- old          | 9   | 2    | 1-3,6,8-11 | 1,3,6,8-11 |
| 10: old <- result        | 10  | 3    | 1,3,6,8-11 | 1,6,8-11   |
| 11: i <- i + 1           | 11  | 6    | 1,6,8-11   | 1,8-11     |
| 12: JUMP 7               |     |      | 1,8-11     | 1,8-11     |
| 13: return result        |     |      | 1-4,6      | 1-4,6      |
| 14: return n             |     |      | 1-4        | 1-4        |

## An Improvement: Basic Blocks

- No need to compute this one stmt at a time
- For straight line code:
  - In[s1; s2] = in[s1]
  - Out[s1; s2] = out[s2]
- Can we combine the gen and kill sets into one set per BB?

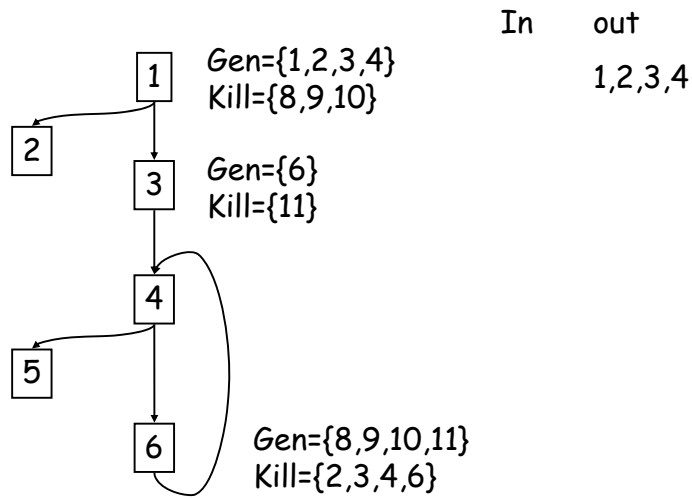
|               | Gen | kill |
|---------------|-----|------|
| 1: i <- 1     | 1   | 8,4  |
| 2: j <- 2     | 2   |      |
| 3: k <- 3 + i | 3   | 11   |
| 4: i <- j     | 4   | 1,8  |
| 5: m <- i + k | 5   |      |

- Gen[BB] = {2,3,4,5}
- Kill[BB] = {1,8,11}
- Gen[s1;s2] =
- Kill[s1;s2] =

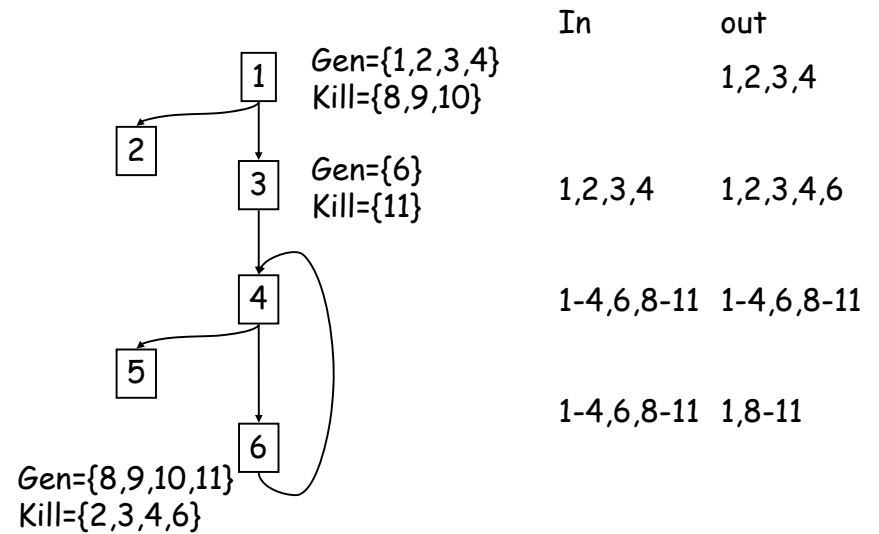
## BB sets

|                          | Gen | kill    |
|--------------------------|-----|---------|
| 1: n <- 10               | 1   |         |
| 2: older <- 0            | 2   | 9       |
| 3: old <- 1              | 3   | 10      |
| 4: result <- 0           | 4   | 8       |
| 5: if n <= 1 goto 14     |     | 1,2,3,4 |
| 6: i <- 2                | 6   | 11      |
| 7: if i > n goto 13      |     | 6       |
| 8: result <- old + older | 8   | 4       |
| 9: older <- old          | 9   | 2       |
| 10: old <- result        | 10  | 3       |
| 11: i <- i + 1           | 11  | 6       |
| 12: JUMP 7               |     | 8-11    |
| 13: return result        |     | 2-4,6   |
| 14: return n             |     |         |

## BB sets



## BB sets



## Forward Dataflow

- Reaching definitions is a forward dataflow problem:
  - It propagates information from preds of a node to the node
- Defined by:
  - Basic attributes: (gen and kill)
  - Transfer function:  $out[b]=F_{bb}(in[b])$
  - Meet operator:  $in[b]=M(out[p])$  for all  $p \in pred(b)$
  - Set of values (a lattice, in this case powerset of program points)
  - Initial values for each node  $b$
- Solve for fixed point solution

## How to implement?

- Values?
- Gen?
- Kill?
- $F_{bb}$ ?
- Order to visit nodes?
- When are we done?
  - In fact, do we know we terminate?

## Implementing RD

- Values: bits in a bit vector
- Gen: 1 in each position generated, otherwise 0
- Kill: 0 in each position killed, otherwise 1
- $F_{bb}$ :  $out[b] = (in[b] \mid gen[b]) \& kill[b]$
- Init  $in[b]=out[b]=0$
- When are we done?
- What order to visit nodes? Does it matter?

## RD Worklist algorithm

Initialize:  $in[B] = out[b] = \emptyset$

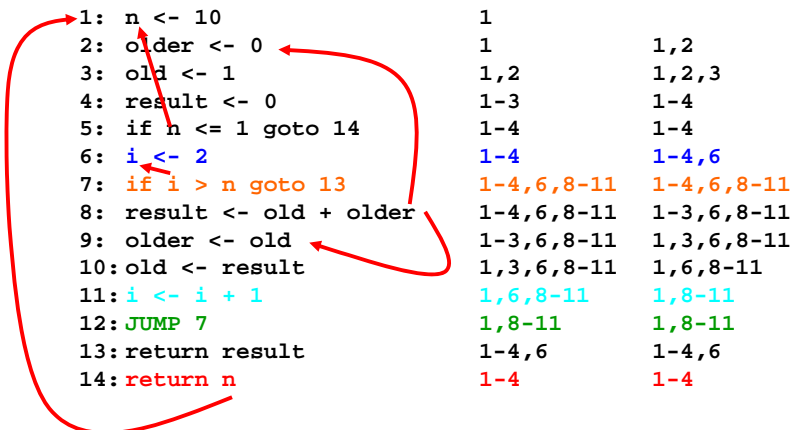
Initialize:  $in[entry] = \emptyset$

Work queue,  $W =$  all Blocks in topological order

```
while (|W| != 0) {
 remove b from W
 old = out[b]
 in[b] = {over all pred(p) ∈ b} ∪ out[p]
 out[b] = gen[b] ∪ (in[b] - kill[b])
 if (old != out[b]) W = W ∪ succ(b)
}
```

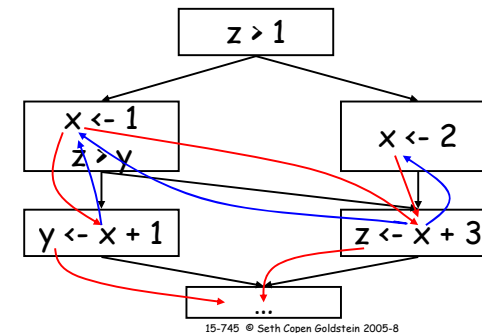
## Storing Rd information

- Use-def chains: for each use of var  $x$  in  $s$ , a list of definitions of  $x$  that reach  $s$



## Def-use chains are valuable too

- Def-use chain: for each definition of var  $x$ , a list of all uses of that definition
- Computed from liveness analysis, a backward dataflow problem
- Def-use and use-def are different



## Using RD for Simple Const. Prop.

|                          |            |            |
|--------------------------|------------|------------|
| 1: n <- 10               | 1          |            |
| 2: older <- 0            | 1          | 1,2        |
| 3: old <- 1              | 1,2        | 1,2,3      |
| 4: result <- 0           | 1-3        | 1-4        |
| 5: if n <= 1 goto 14     | 1-4        | 1-4        |
| 6: i <- 2                | 1-4        | 1-4,6      |
| 7: if i > n goto 13      | 1-4,6,8-11 | 1-4,6,8-11 |
| 8: result <- old + older | 1-4,6,8-11 | 1-3,6,8-11 |
| 9: older <- old          | 1-3,6,8-11 | 1,3,6,8-11 |
| 10: old <- result        | 1,3,6,8-11 | 1,6,8-11   |
| 11: i <- i + 1           | 1,6,8-11   | 1,8-11     |
| 12: JUMP 7               | 1,8-11     | 1,8-11     |
| 13: return result        | 1-4,6      | 1-4,6      |
| 14: return n             | 1-4        | 1-4        |

## Better Constant Propagation

- What about:
 

```

x <- 1
if (y > z)
 x <- 1
a <- x

```

## Better Constant Propagation

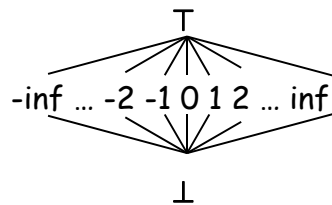
- What about:
 

```

x <- 1
if (y > z)
 x <- 1
a <- x

```

- Lattice



- Meet:
 

```

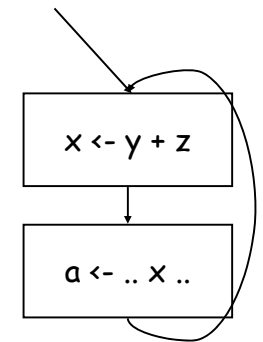
a <- a ∧ T
⊥ <- a ∧ ⊥
c <- c ∧ c
⊥ <- c ∧ d (if c ≠ d)

```

- Init all vars to: bot or top?

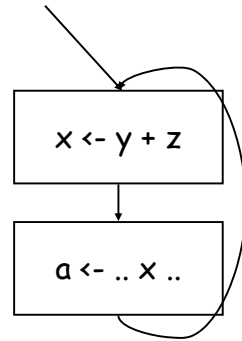
## Loop Invariant Code Motion

- When can expression be moved out of a loop?



## Loop Invariant Code Motion

- When can expression be moved out of a loop?
- When all reaching definitions of operands are outside of loop, expression is loop invariant
- Use ud-chains to detect
- Can du-chains be helpful?



## Liveness (def-use chains)

- A variable  $x$  is live-out of a stmt  $s$  if  $x$  can be used along some path starting a  $s$ , otherwise  $x$  is dead.
- Why is this important?
- How can we frame this as a dataflow problem?

## Liveness as a dataflow problem

- This is a backwards analysis
  - A variable is live out if used by a successor
  - Gen: For a use: indicate it is live coming into  $s$
  - Kill: Defining a variable  $v$  in  $s$  makes it dead before  $s$  (unless  $s$  uses  $v$  to define  $v$ )
  - Lattice is just live (top) and dead (bottom)
- Values are variables
- $In[n] = \text{variables live before } n$   
 $= out[n] - kill[n] \cup gen[n]$
- $Out[n] = \text{variables live after } n$   
 $= \bigcup_{s \in succ(n)} In[s]$

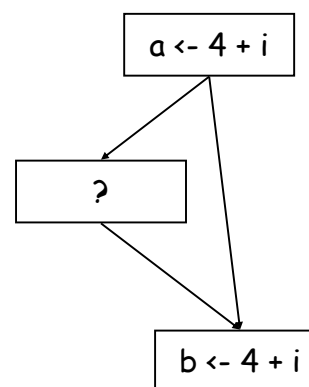
## Dead Code Elimination

- Code is dead if it has no effect on the outcome of the program.
- When is code dead?

## Dead Code Elimination

- Code is dead if it has no effect on the outcome of the program.
- When is code dead?
  - When the definition is dead, and
  - When the instruction has no side effects
- So:
  - run liveness
  - Construct def-use chains
  - Any instruction which has no users and has no side effects can be eliminated

## When can we do CSE?



## Available Expressions

- $X+Y$  is "available" at statement  $S$  if
  - $x+y$  is computed along every path from the start to  $S$  AND
  - neither  $x$  nor  $y$  is modified after the last evaluation of  $x+y$

`a ← b+c`

`b ← a-d`

`c ← b+c`

`d ← a-d`

## Computing Available Expressions

- Forward or backward?
- Values?
- Lattice?
- $gen[b] =$
- $kill[b] =$
- $in[b] =$
- $out[b] =$
- initialization?

## Computing Available Expressions

- Forward
- Values: all expressions
- Lattice: available, not-avail
- $gen[b]$  = if  $b$  evals expr  $e$  and doesn't define variables used in  $e$
- $kill[b]$  = if  $b$  assigns to  $x$ , then all exprs using  $x$  are killed.
- $out[b] = in[b] - kill[b] \cup gen[b]$
- $in[b]$  = what to do at a join point?
- initialization?

## Computing Available Expressions

- Forward
- Values: all expressions
- Lattice: available, not-avail
- $gen[b]$  = if  $b$  evals expr  $e$  and doesn't define variables used in  $e$
- $kill[b]$  = if  $b$  assigns to  $x$ , exprs( $x$ ) are killed  
 $out[b] = in[b] - kill[b] \cup gen[b]$
- $in[b]$  = An expr is avail only if avail on ALL edges, so:  $in[b] = \cap$  over all  $p \in pred(b)$ ,  $out[p]$
- Initialization
  - All nodes, but entry are set to ALL avail
  - Entry is set to NONE avail

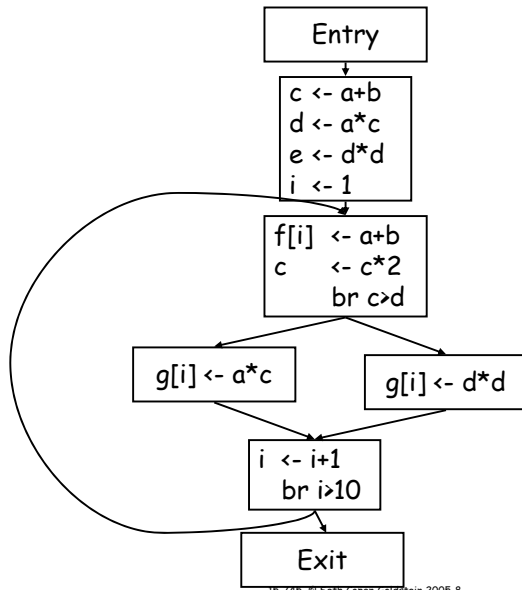
## Constructing Gen & Kill

| Stmt                           | Gen                             | Kill                             |
|--------------------------------|---------------------------------|----------------------------------|
| $t \leftarrow x \text{ op } y$ | $\{x \text{ op } y\} - kill[s]$ | $\{\text{exprs containing } t\}$ |
| $t \leftarrow M[a]$            | $\{M[a]\} - kill[s]$            |                                  |
| $M[a] \leftarrow b$            |                                 |                                  |
| $f(a, \dots)$                  |                                 | $\{M[x] \text{ for all } x\}$    |
| $t \leftarrow f(a, \dots)$     |                                 |                                  |

## Constructing Gen & Kill

| Stmt                           | Gen                             | Kill                                                      |
|--------------------------------|---------------------------------|-----------------------------------------------------------|
| $t \leftarrow x \text{ op } y$ | $\{x \text{ op } y\} - kill[s]$ | $\{\text{exprs containing } t\}$                          |
| $t \leftarrow M[a]$            | $\{M[a]\} - kill[s]$            | $\{\text{exprs containing } t\}$                          |
| $M[a] \leftarrow b$            | $\{\}$                          | $\{\text{for all } x, M[x]\}$                             |
| $f(a, \dots)$                  | $\{\}$                          | $\{\text{for all } x, M[x]\}$                             |
| $t \leftarrow f(a, \dots)$     | $\{\}$                          | $\{\text{exprs containing } t \text{ for all } x, M[x]\}$ |

## Example

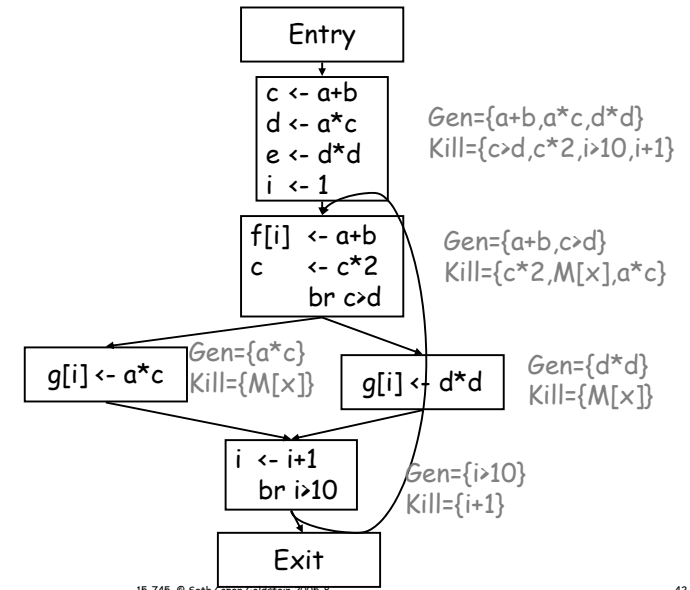


Lecture 2

15-745 © Seth Copen Goldstein 2005-8

41

## Example



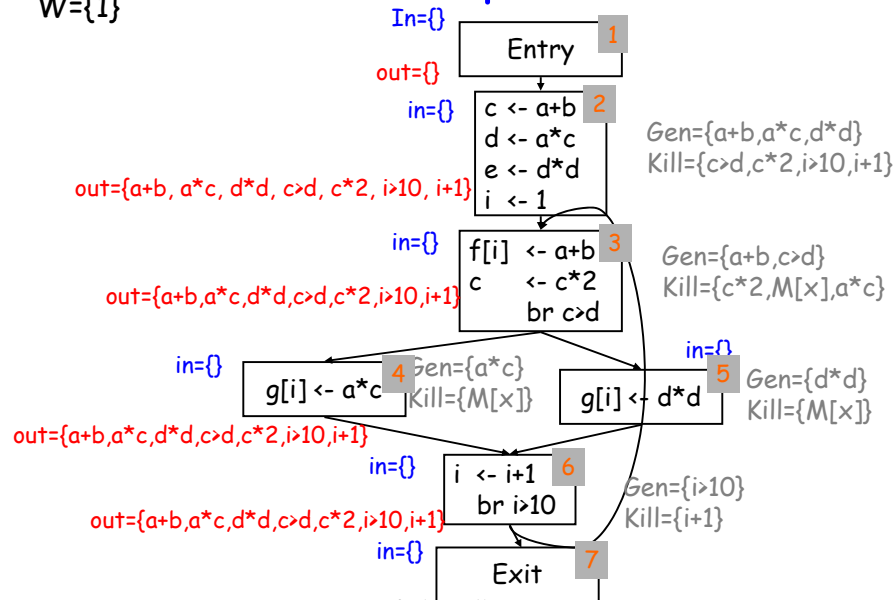
Lecture 2

15-745 © Seth Copen Goldstein 2005-8

42

$W=\{1\}$

## Example



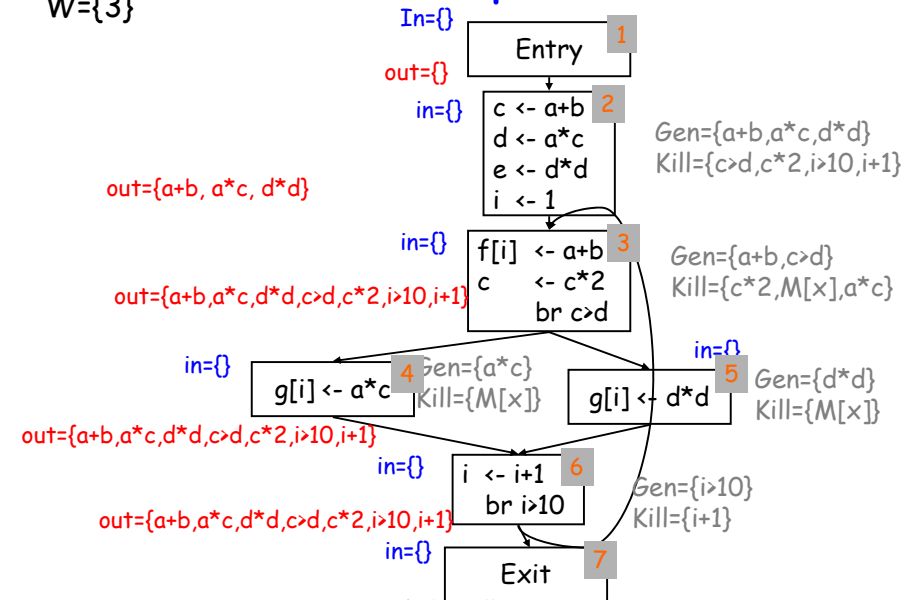
Lecture 2

15-745 © Seth Copen Goldstein 2005-8

43

$W=\{3\}$

## Example



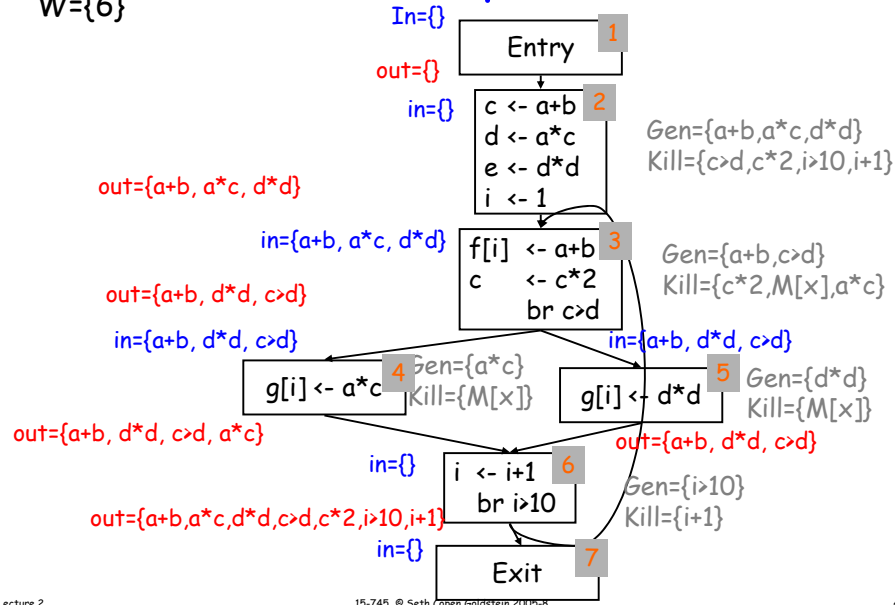
Lecture 2

15-745 © Seth Copen Goldstein 2005-8

44

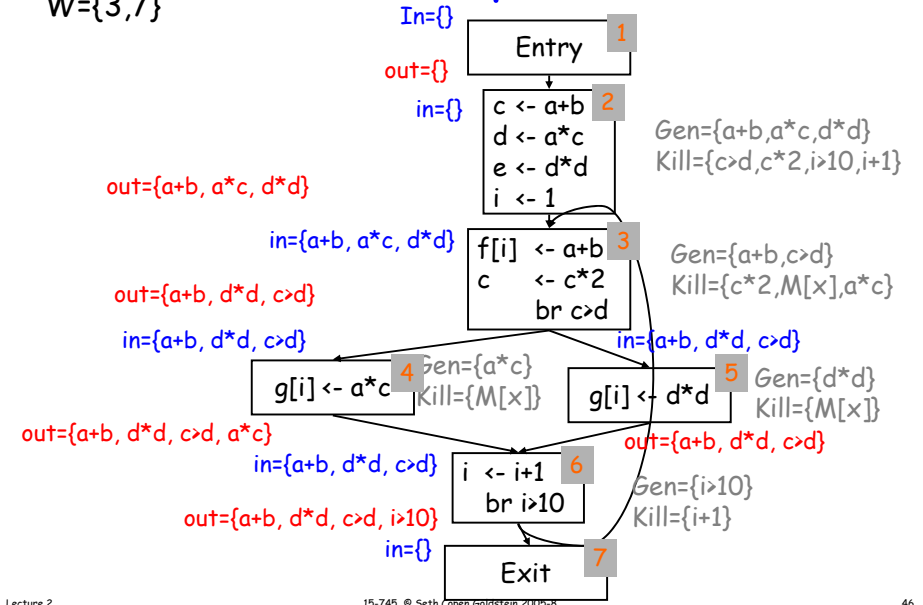
W={6}

### Example



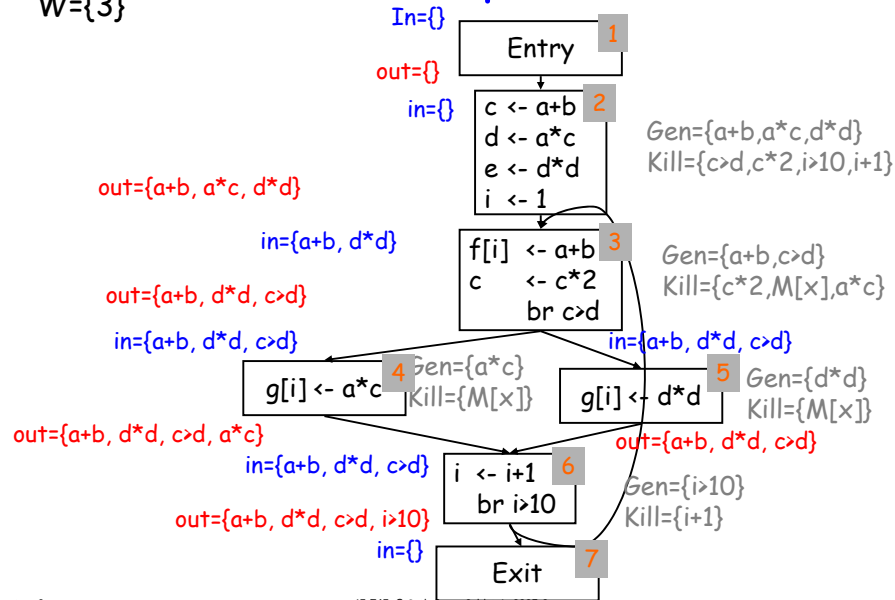
W={3,7}

### Example



W={3}

### Example

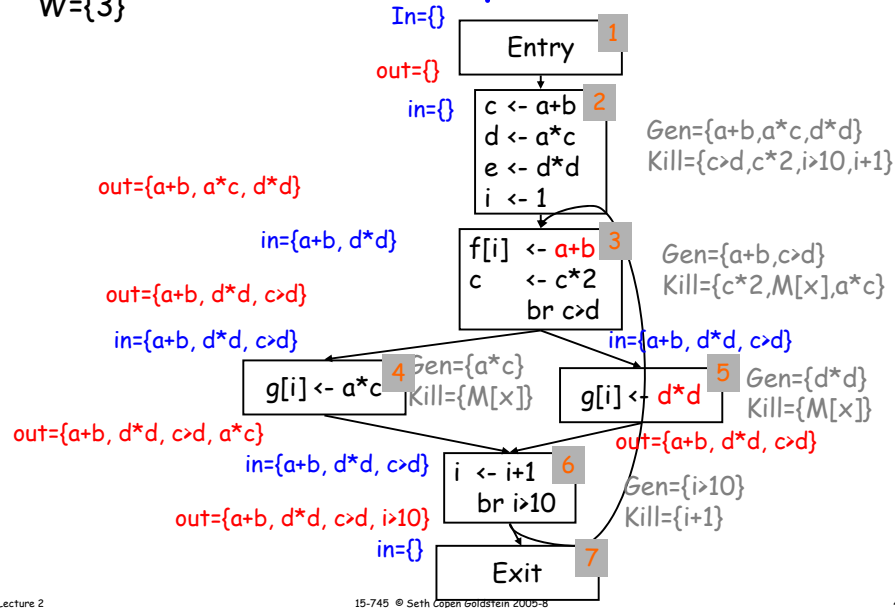


### CSE

- Calculate Available expressions
- For every stmt in program
  - If expression,  $x \text{ op } y$ , is available {
    - Compute reaching expressions for  $x \text{ op } y$  at this stmt
    - foreach stmt in RE of the form  $t \leftarrow x \text{ op } y$ 
      - rewrite at:  $t' \leftarrow x \text{ op } y$
      - $t \leftarrow t'$
- }
  - replace  $x \text{ op } y$  in stmt with  $t'$

W={3}

## Example



Lecture 2

15-745 © Seth Copen Goldstein 2005-8

49

## Calculating RE

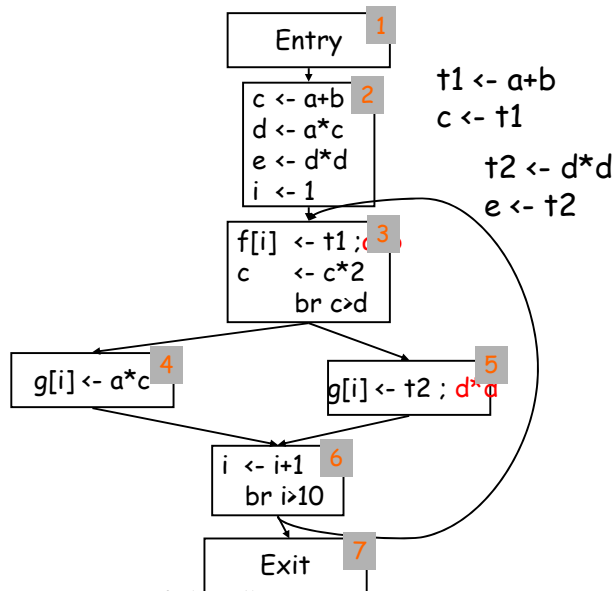
- Could be dataflow problem, but not needed enough, so ...
- To find RE for  $x \text{ op } y$  at stmt  $S$ 
  - traverse cfg backward from  $S$  until
    - reach  $t \leftarrow x + y$  (& put into RE)
    - reach definition of  $x$  or  $y$

Lecture 2

15-745 © Seth Copen Goldstein 2005-8

50

## Example



Lecture 2

15-745 © Seth Copen Goldstein 2005-8

51

## Dataflow Summary

|          | Union          | intersection    |
|----------|----------------|-----------------|
| Forward  | Reaching defs  | Available exprs |
| Backward | Live variables |                 |

Later in course we look at bidirectional dataflow

Lecture 2

15-745 © Seth Copen Goldstein 2005-8

52

# Dataflow Framework

- Lattice
- Universe of values
- Meet operator
- Basic attributes (e.g., gen, kill)
- Traversal order
- Transfer function

# 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

## Advantages of SSA


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

## 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)


## Straight-line SSA

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



## Straight-line SSA

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

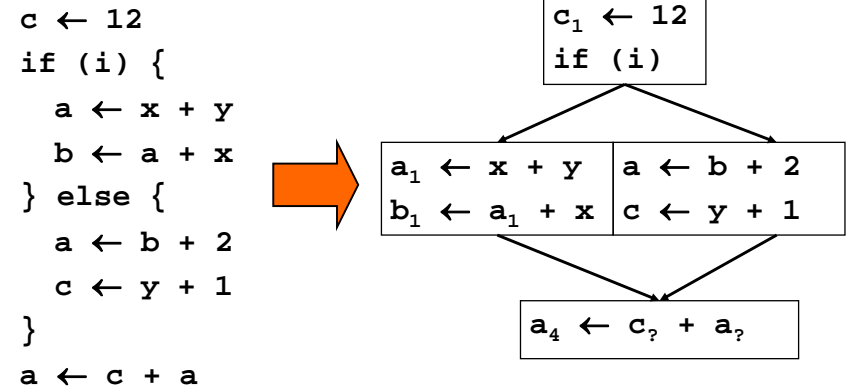


$a_1 \leftarrow x + y$   
 $b_1 \leftarrow a_1 + x$   
 $a_2 \leftarrow b_1 + 2$   
 $c_1 \leftarrow y + 1$   
 $a_3 \leftarrow c_1 + a_2$

# 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?

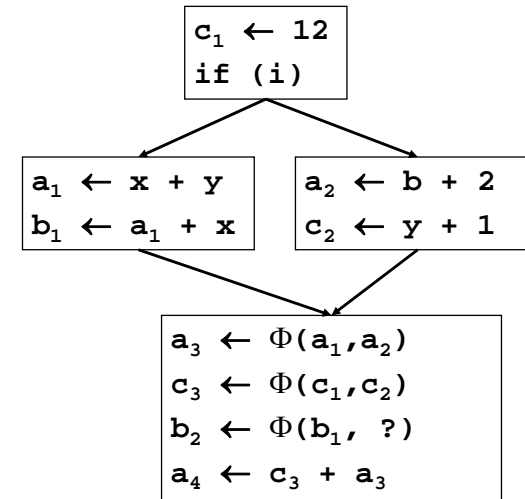
# Merging at Joins



# 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  $\Phi$  function

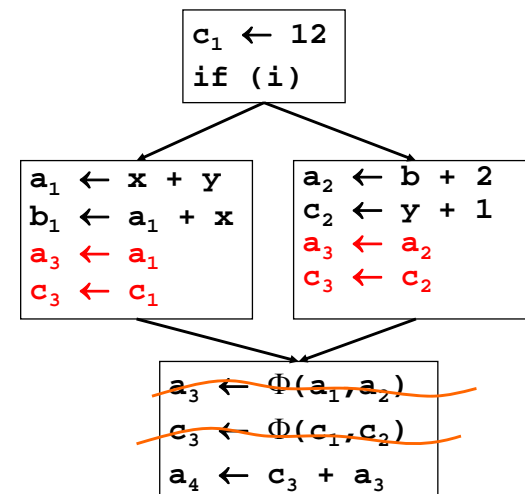
# Merging at Joins



## The $\Phi$ function

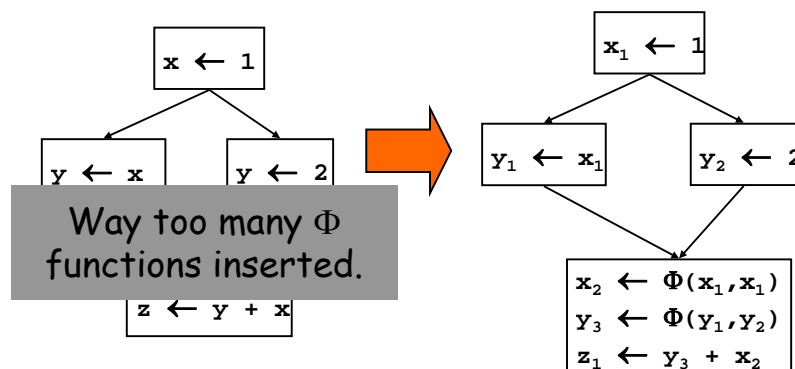
- $\Phi$  merges multiple definitions along multiple control paths into a single definition.
- At a BB with  $p$  predecessors, there are  $p$  arguments to the  $\Phi$  function.  
 $x_{new} \leftarrow \Phi(x_1, x_1, x_1, \dots, x_p)$
- How do we choose which  $x_i$  to use?
  - We don't really care!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

## "Implementing" $\Phi$



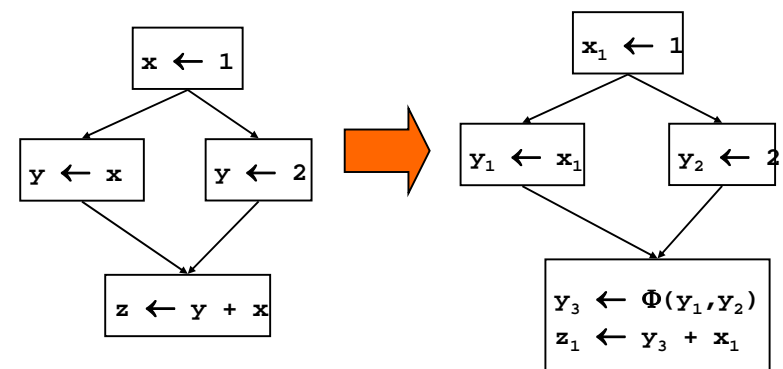
## Trivial SSA

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

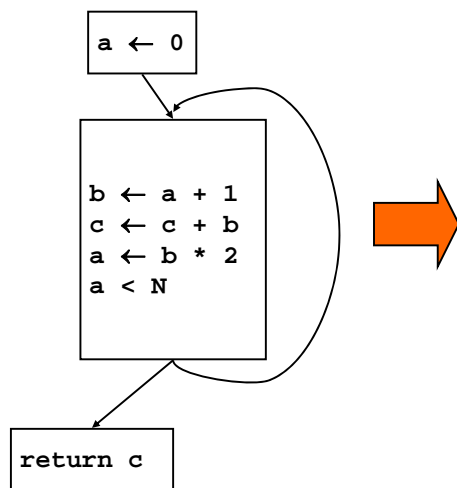


## Minimal SSA

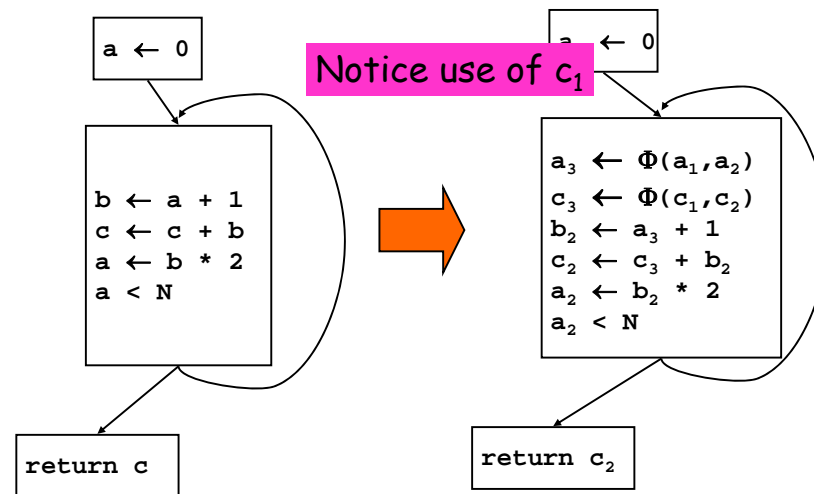
- Each assignment generates a fresh variable.
- At each join point insert  $\Phi$  functions for all variables with **multiple outstanding defs.**



## Another Example



## Another Example

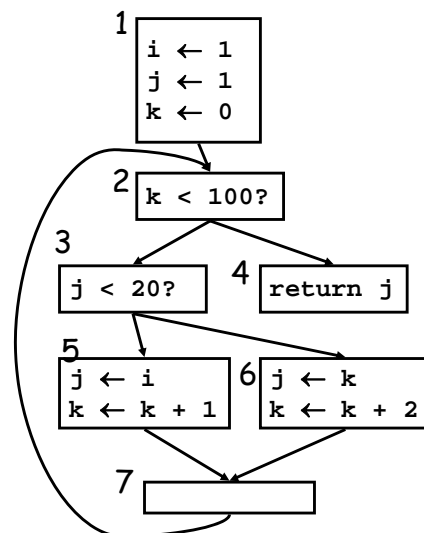


## Lets optimize the following:

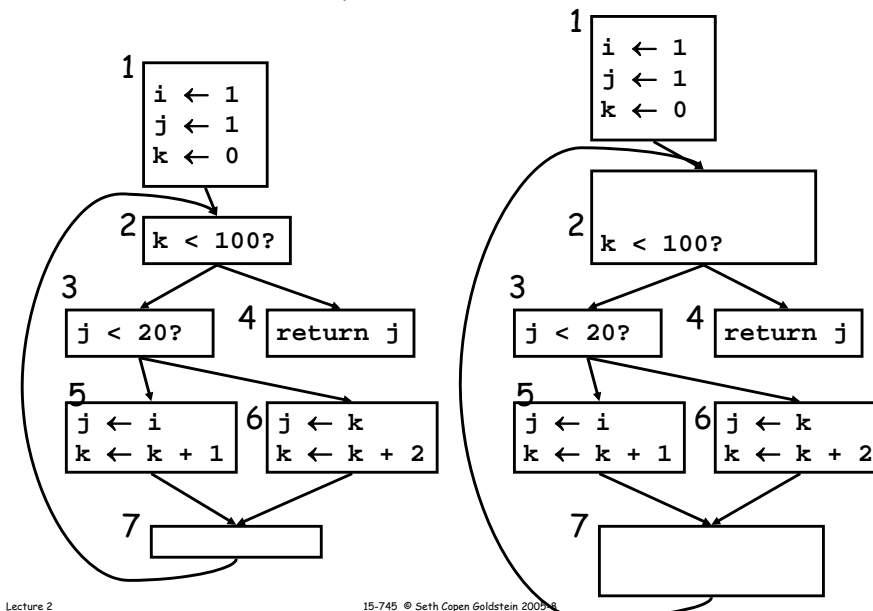
```

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

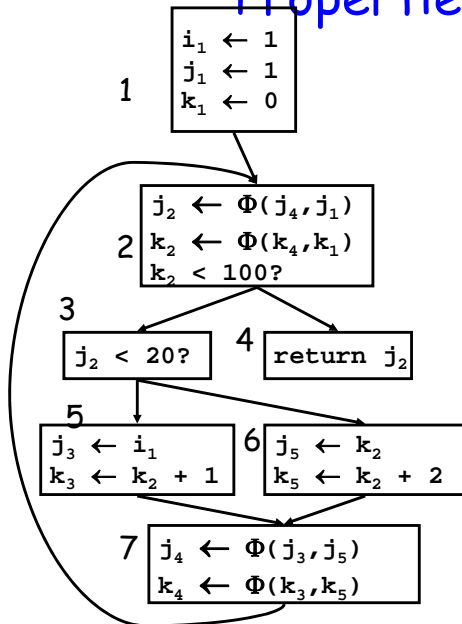
```



## First, turn into SSA



## Properties of SSA



- Only 1 assignment per variable
- definitions dominate uses
- Can we use this to help with constant propagation?

Lecture 2

15-745 © Seth Copen Goldstein 2005-8

73

## Constant Propagation

- If " $v \leftarrow c$ ", replace all uses of  $v$  with  $c$
- If " $v \leftarrow \Phi(c, c, c)$ " replace all uses of  $v$  with  $c$

```

W ← list of all defs
while !W.isEmpty {
 Stmt S ← W.removeOne
 if S has form " $v \leftarrow \Phi(c, \dots, c)$ "
 replace S with $V \leftarrow c$
 if S has form " $v \leftarrow c$ " then
 delete S
 foreach stmt U that uses v,
 replace v with c in U
 W.add(U)
}

```

Lecture 2

15-745 © Seth Copen Goldstein 2005-8

74

## Other stuff we can do?

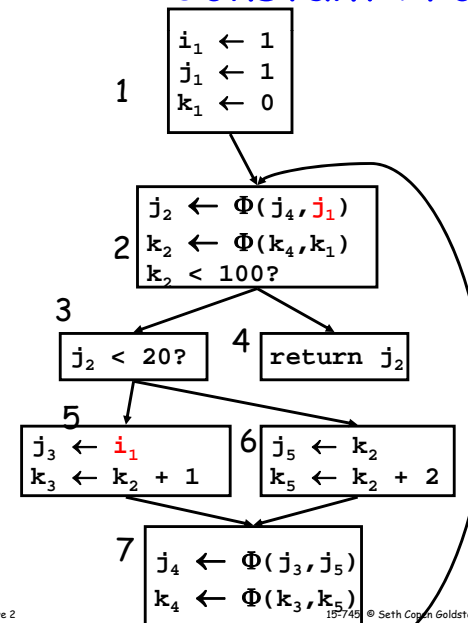
- Copy propagation  
delete " $x \leftarrow \Phi(y)$ " and replace all  $x$  with  $y$   
delete " $x \leftarrow y$ " and replace all  $x$  with  $y$
- Constant Folding  
(Also, constant conditions too!)
- Unreachable Code  
Remember to delete all edges from unreachable block

Lecture 2

15-745 © Seth Copen Goldstein 2005-8

75

## Constant Propagation

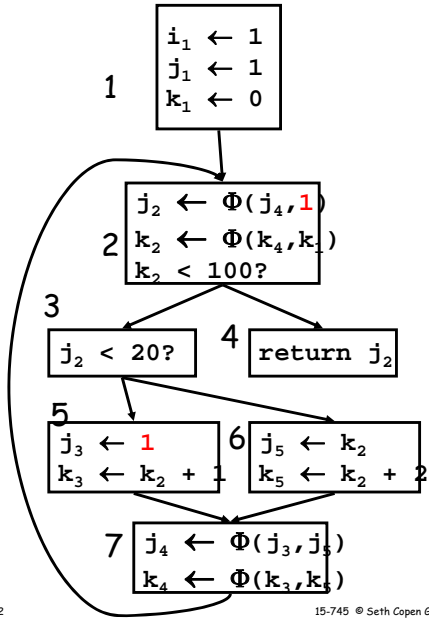


Lecture 2

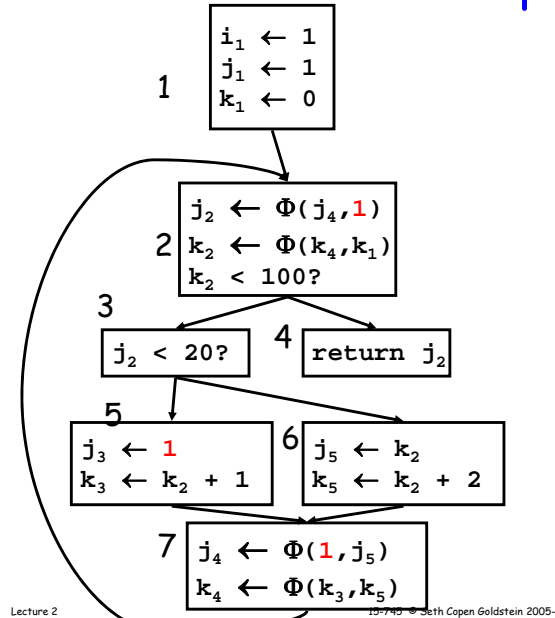
15-745 © Seth Copen Goldstein 2005-8

76

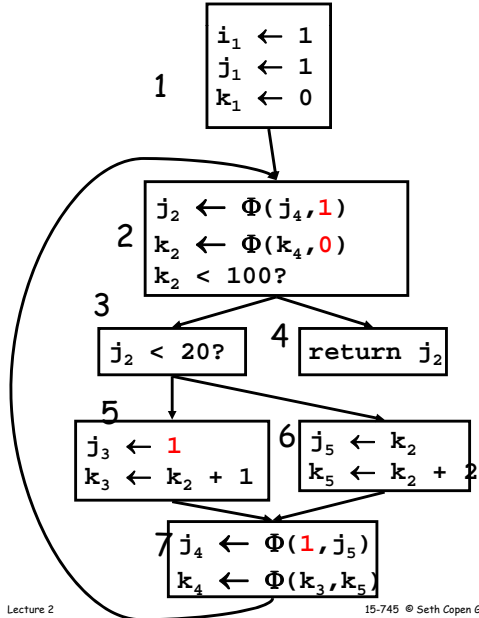
# Constant Propagation



# Constant Propagation



# Constant Propagation



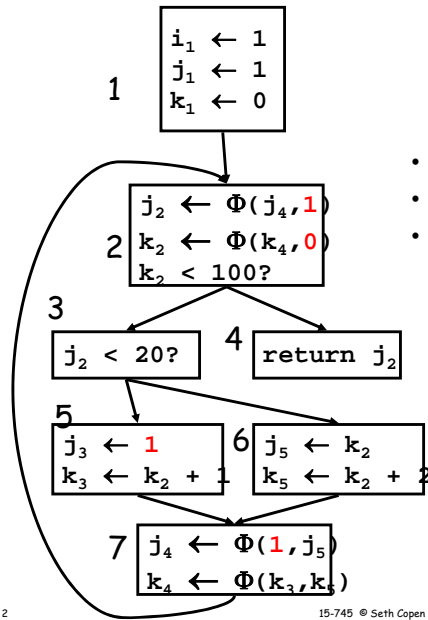
But, so what?

You will have to wait  
til next time :)

# Summary

- Dataflow framework
  - Lattice, meet, direction, transfer function, initial values
- Du-chains, ud-chains
- CSE
- SSA
  - One static definition per variable
  - $\Phi$ -functions

# 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)

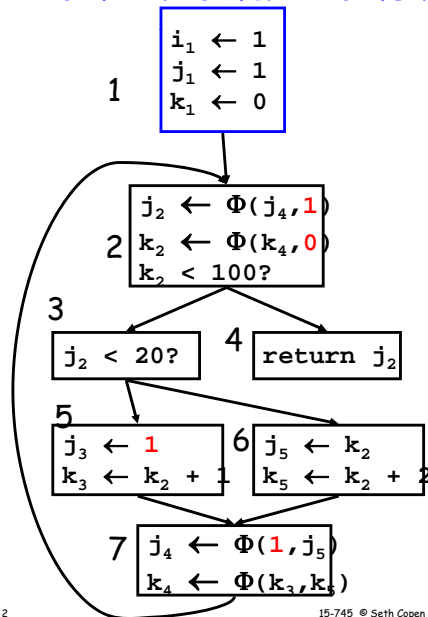
Use a lattice for variables:

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

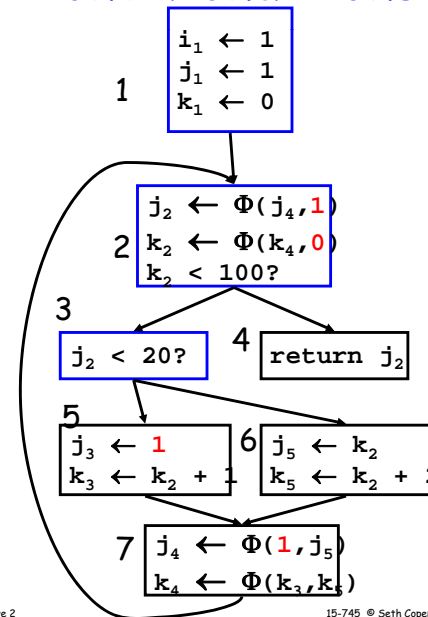
integers = we have seen evidence that the var has been assigned a constant with the value

BOT = not executed

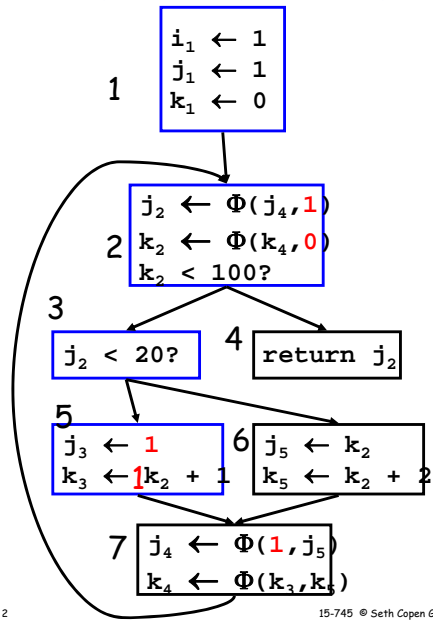
# Conditional Constant Propagation



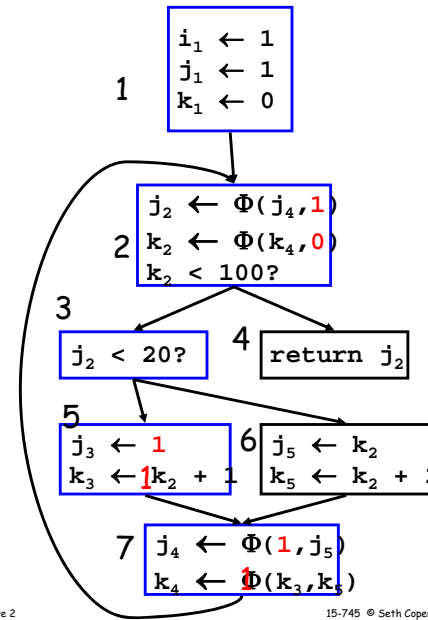
# Conditional Constant Propagation



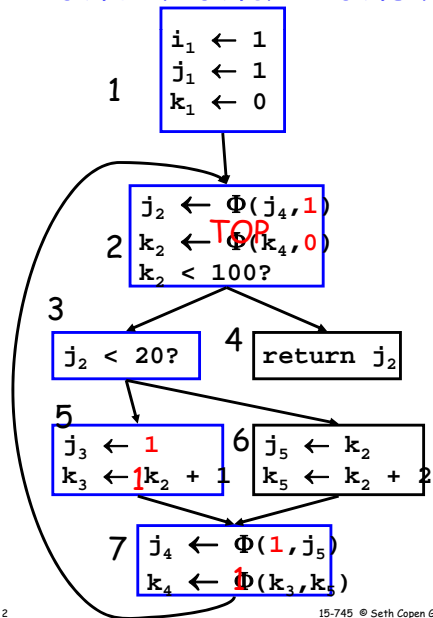
# Conditional Constant Propagation



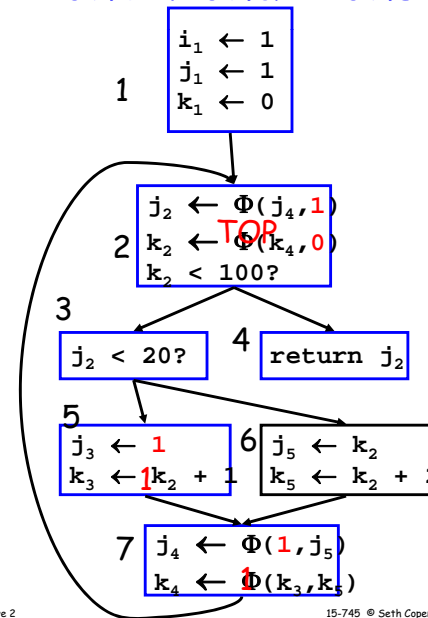
# Conditional Constant Propagation



# Conditional Constant Propagation



# Conditional Constant Propagation



# CCP

