

Data Flow Analysis

Suman Jana

Adopted From U Penn CIS 570: Modern Programming Language Implementation (Autumn 2006)

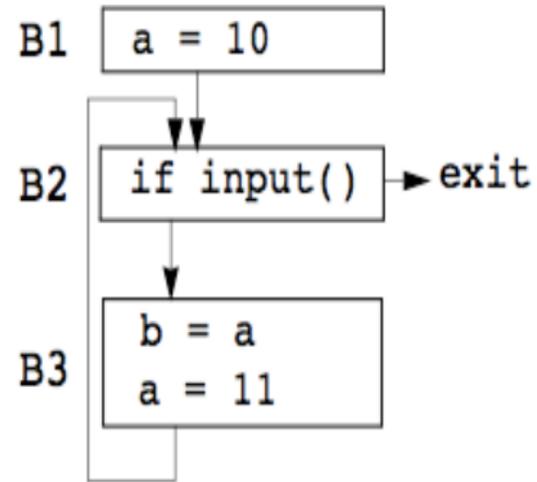
Data flow analysis

- Derives information about the **dynamic** behavior of a program by only examining the **static** code
- Intraprocedural analysis
- Flow-sensitive: sensitive to the control flow in a function
- **Examples**
 - Live variable analysis
 - Constant propagation
 - Common subexpression elimination
 - Dead code detection

```
1  a := 0
2  L1: b := a + 1
3  c := c + b
4  a := b * 2
5  if a < 9 goto L1
6  return c
```

- How many registers do we need?
- Easy bound: # of used variables (3)
- Need better answer

Data flow analysis



- **Statically**: finite program
- **Dynamically**: can have infinitely many paths
- Data flow analysis abstraction
 - For each point in the program, combines information of all instances of the same program point

Example 1: Liveness Analysis

Liveness Analysis

Definition

- A variable is **live** at a particular point in the program if its value at that point will be used in the future (**dead**, otherwise).
- To compute liveness at a given point, we need to look into the future

Motivation: Register Allocation

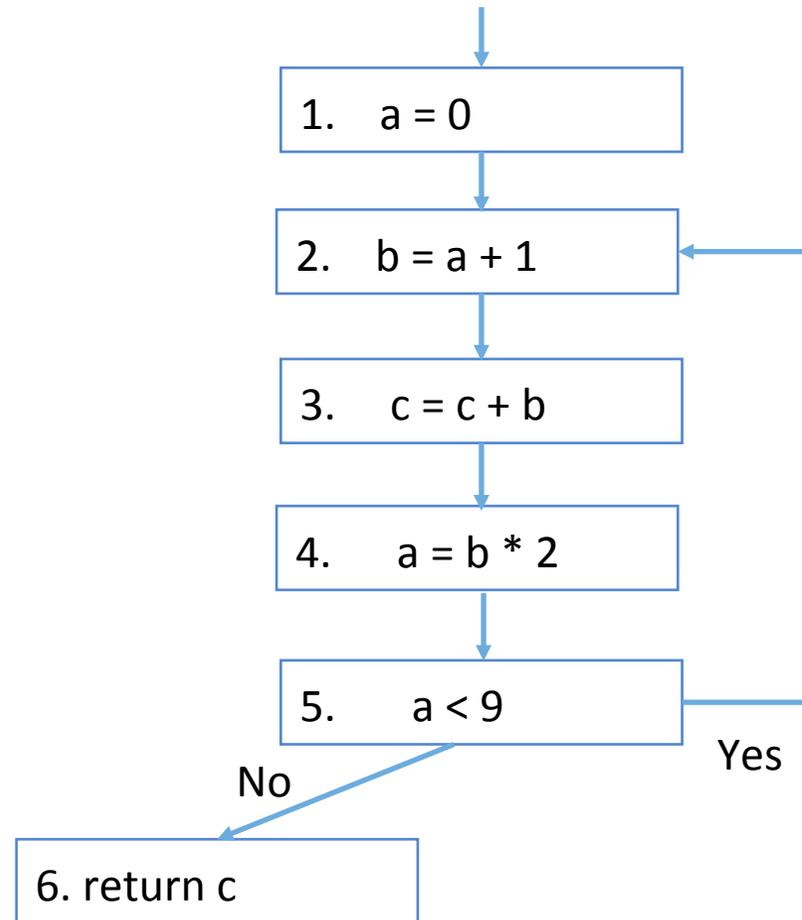
- A program contains an unbounded number of variables
- Must execute on a machine with a bounded number of registers
- Two variables can use the same register if they are never in use at the same time (*i.e.*, never simultaneously live).
- Register allocation uses liveness information

Control Flow Graph

- Let's consider CFG where nodes contain program statement instead of basic block.

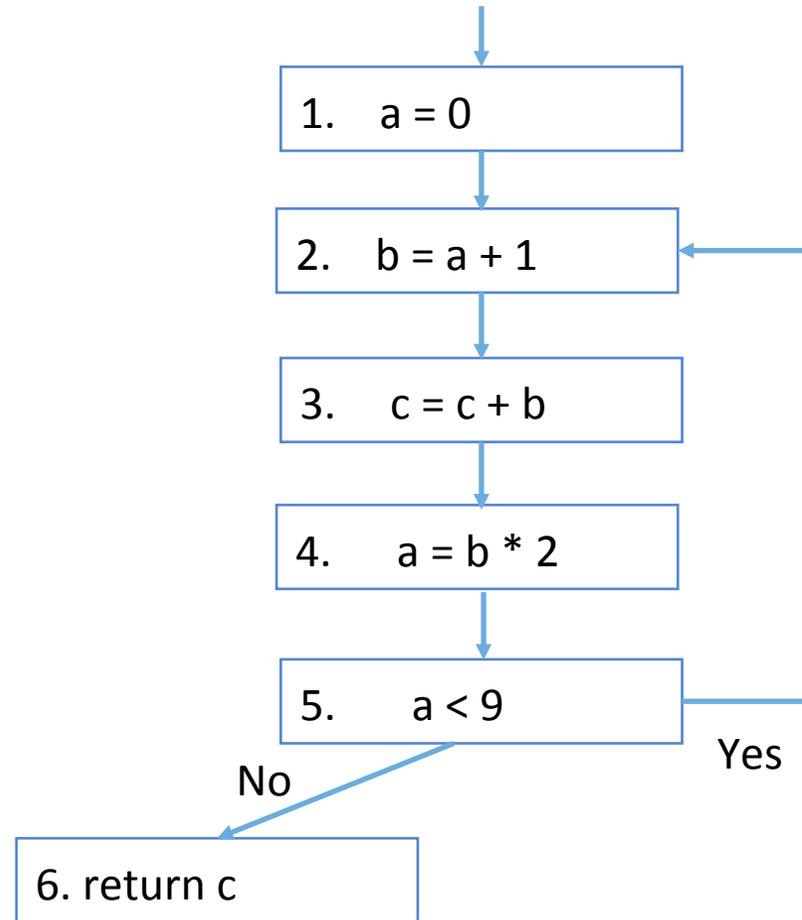
- Example

1. $a := 0$
2. L1: $b := a + 1$
3. $c := c + b$
4. $a := b * 2$
5. if $a < 9$ goto L1
6. return c



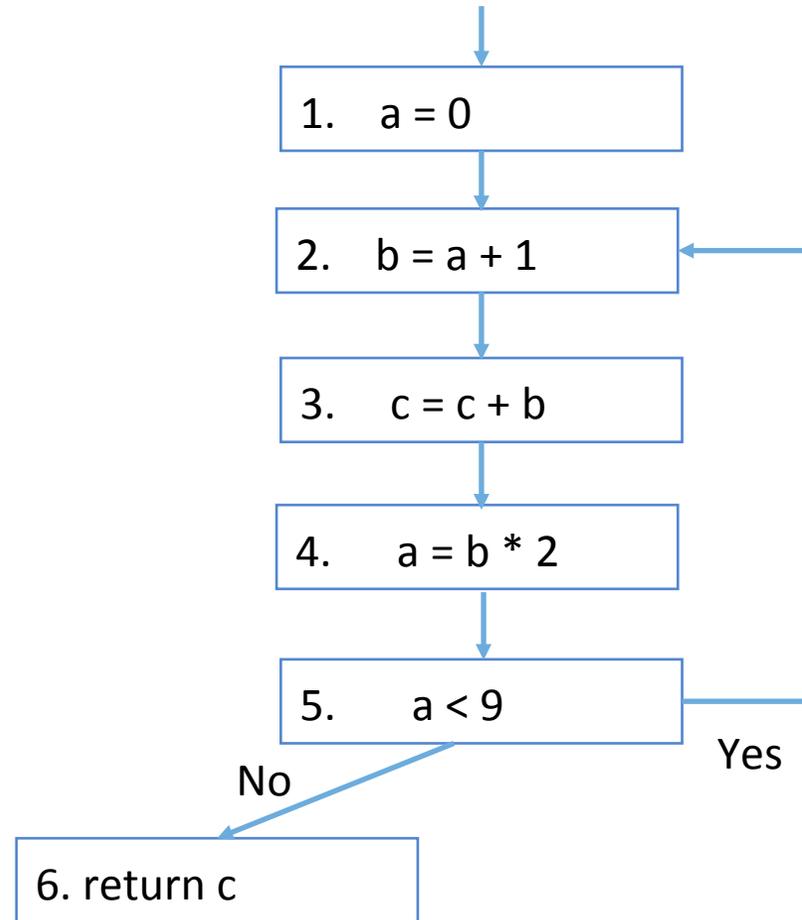
Liveness by Example

- Live range of b
 - Variable b is read in line 4, so b is live on 3-→4 edge
 - b is also read in line 3, so b is live on (2-→3) edge
 - Line 2 assigns b, so value of b on edges (1-→2) and (5-→2) are not needed. So b is **dead** along those edges.
- b's live range is (2-→3-→4)



Liveness by Example

- Live range of a
 - (1->2) and (4->5->2)
 - a is dead on (2->3->4)

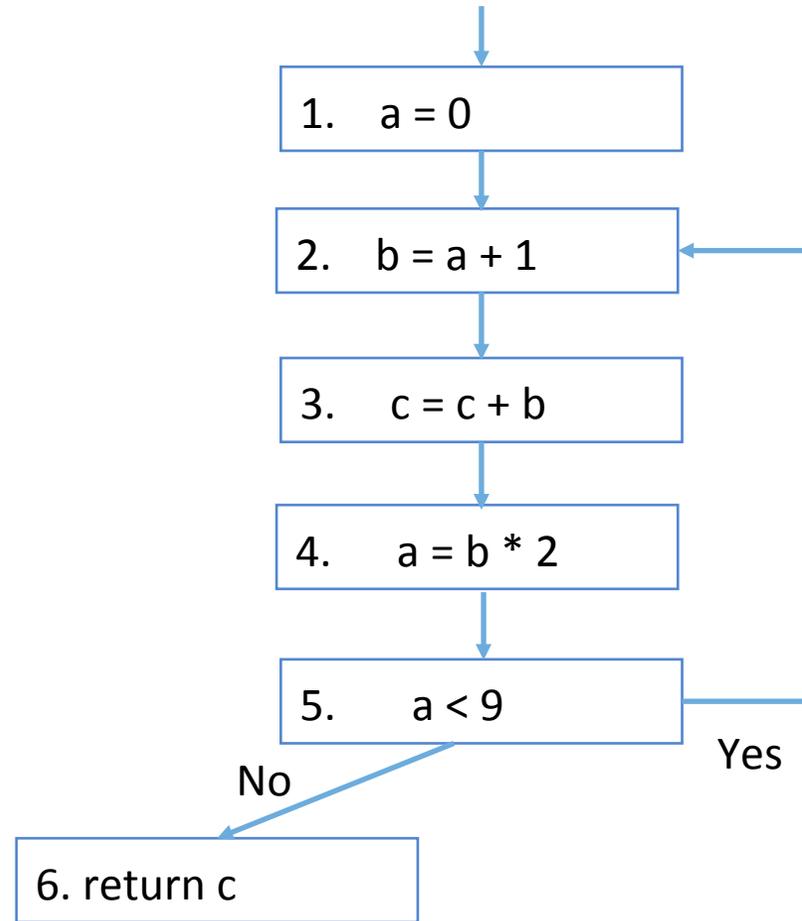


Terminology

- Flow graph terms
 - A CFG node has **out-edges** that lead to **successor** nodes and **in-edges** that come from **predecessor** nodes
 - $\text{pred}[n]$ is the set of all predecessors of node n
 - $\text{succ}[n]$ is the set of all successors of node n

Examples

- Out-edges of node 5: $(5 \rightarrow 6)$ and $(5 \rightarrow 2)$
- $\text{succ}[5] = \{2, 6\}$
- $\text{pred}[5] = \{4\}$
- $\text{pred}[2] = \{1, 5\}$



Uses and Defs

Def (or definition)

- An **assignment** of a value to a variable
- $\text{def}[v]$ = set of CFG nodes that define variable v
- $\text{def}[n]$ = set of variables that are defined at node n

`a = 0`

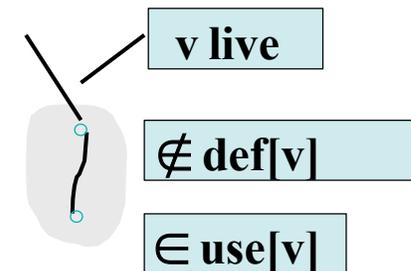
Use

- A **read** of a variable's value
- $\text{use}[v]$ = set of CFG nodes that use variable v
- $\text{use}[n]$ = set of variables that are used at node n

`a < 9`

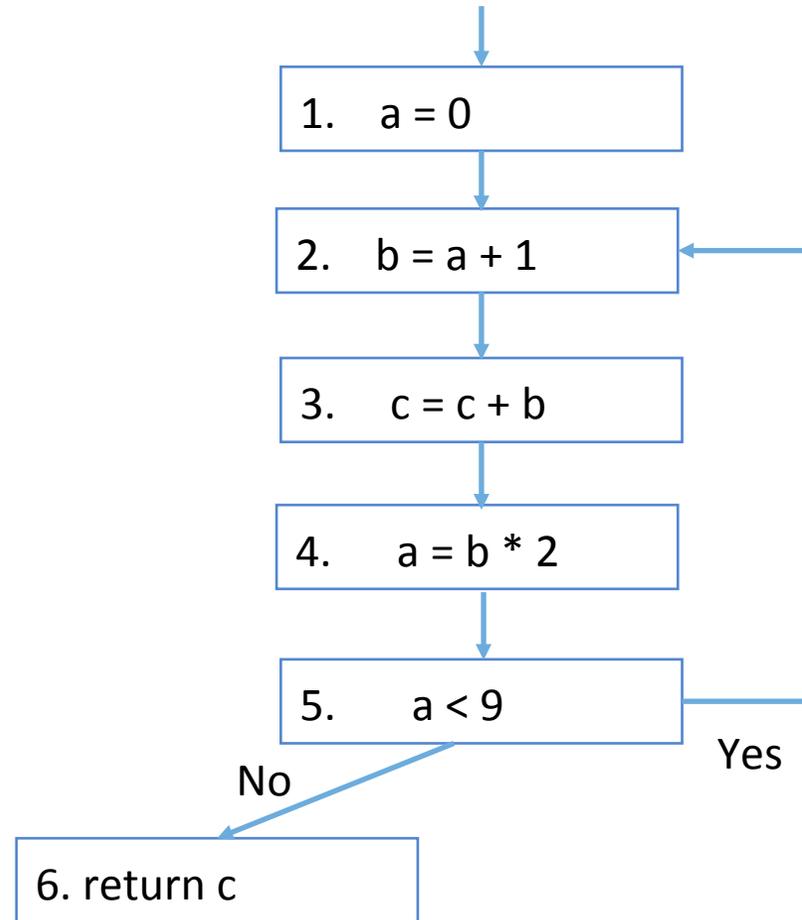
More precise definition of liveness

- A variable v is live on a CFG edge if
 - (1) \exists a directed path from that edge to a use of v (node in $\text{use}[v]$), **and**
 - (2) that path does not go through any def of v (no nodes in $\text{def}[v]$)

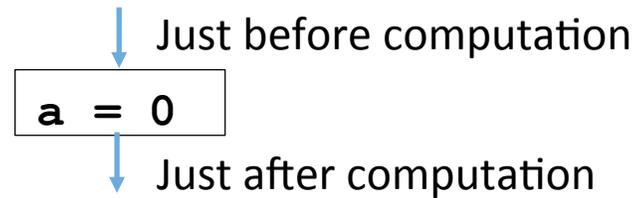


The Flow of Liveness

- Data-flow
 - Liveness of variables is a property that flows through the edges of the CFG
- Direction of Flow
 - Liveness flows backwards through the CFG, because the behavior at future nodes determines liveness at a given node

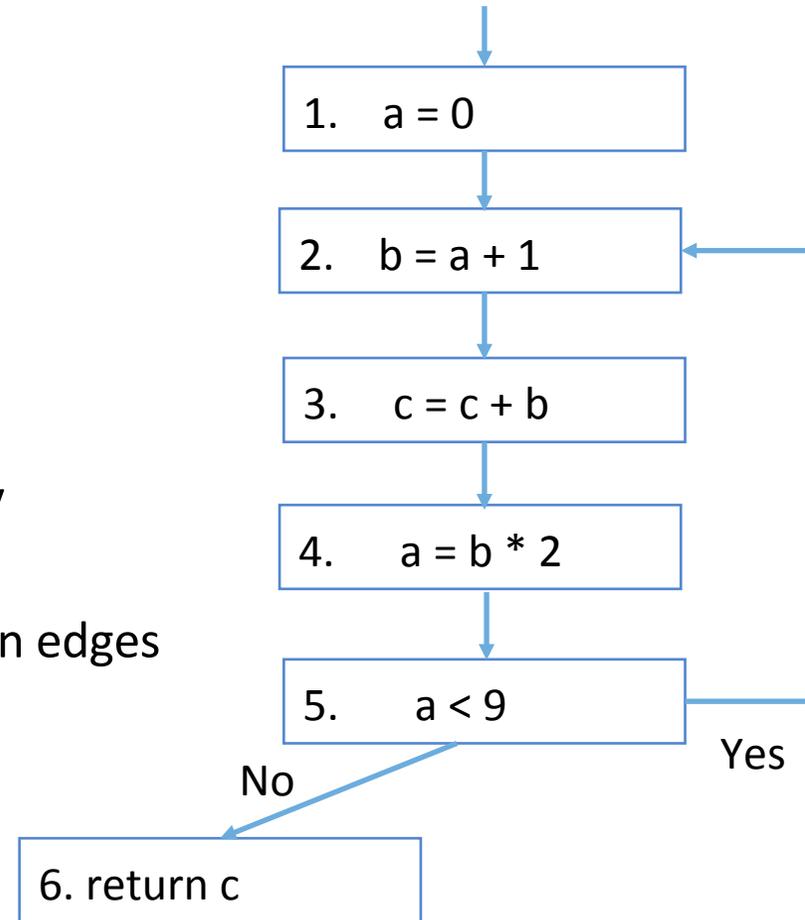


Liveness at Nodes



Two More Definitions

- A variable is **live-out** at a node if it is live on any out edges
- A variable is **live-in** at a node if it is live on any in edges



Computing Liveness

- Generate liveness: If a variable is in $use[n]$, it is live-in at node n
- Push liveness across edges:
 - If a variable is live-in at a node n
 - then it is live-out at all nodes in $pred[n]$
- Push liveness across nodes:
 - If a variable is live-out at node n and not in $def[n]$
 - then the variable is also live-in at n
- Data flow Equation: $in[n] = use[n] \cup (out[n] - def[n])$
 $out[n] = \bigcup_{s \in succ[n]} in[s]$

Solving Dataflow Equation

```
for each node n in CFG  
    in[n] =  $\emptyset$ ; out[n] =  $\emptyset$   
repeat  
    for each node n in CFG  
        in'[n] = in[n]  
        out'[n] = out[n]  
        in[n] = use[n]  $\cup$  (out[n] - def[n])  
        out[n] =  $\cup$  in[s]  
            s  $\in$  succ[n]  
until in'[n]=in[n] and out'[n]=out[n] for all n
```

} Initialize solutions

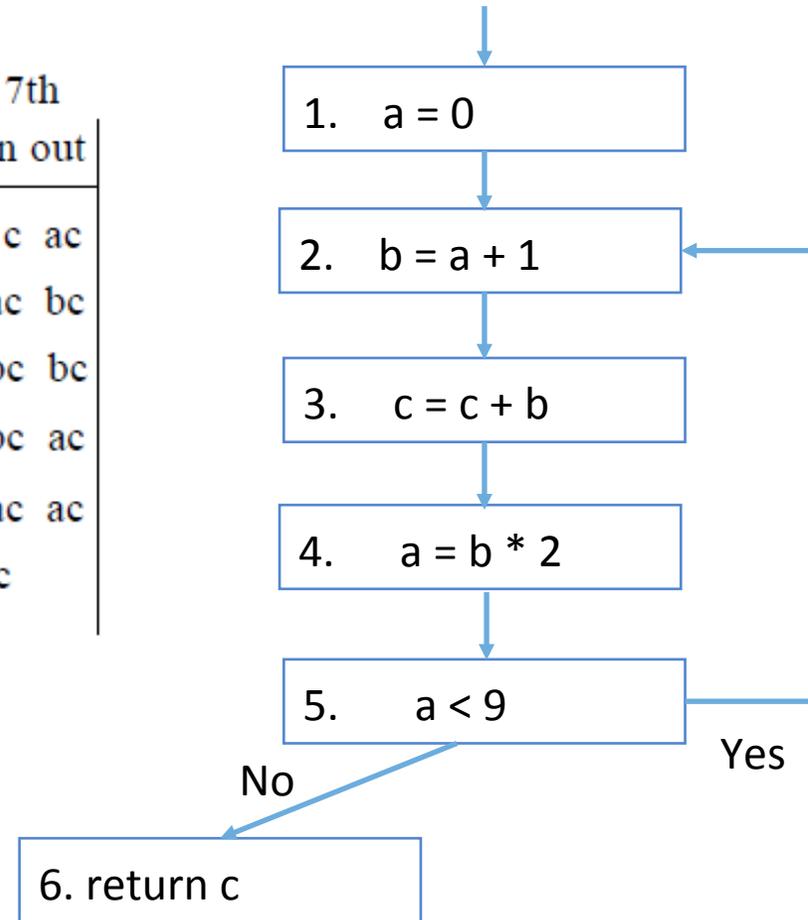
} Save current results

} Solve data-flow equation

} Test for convergence

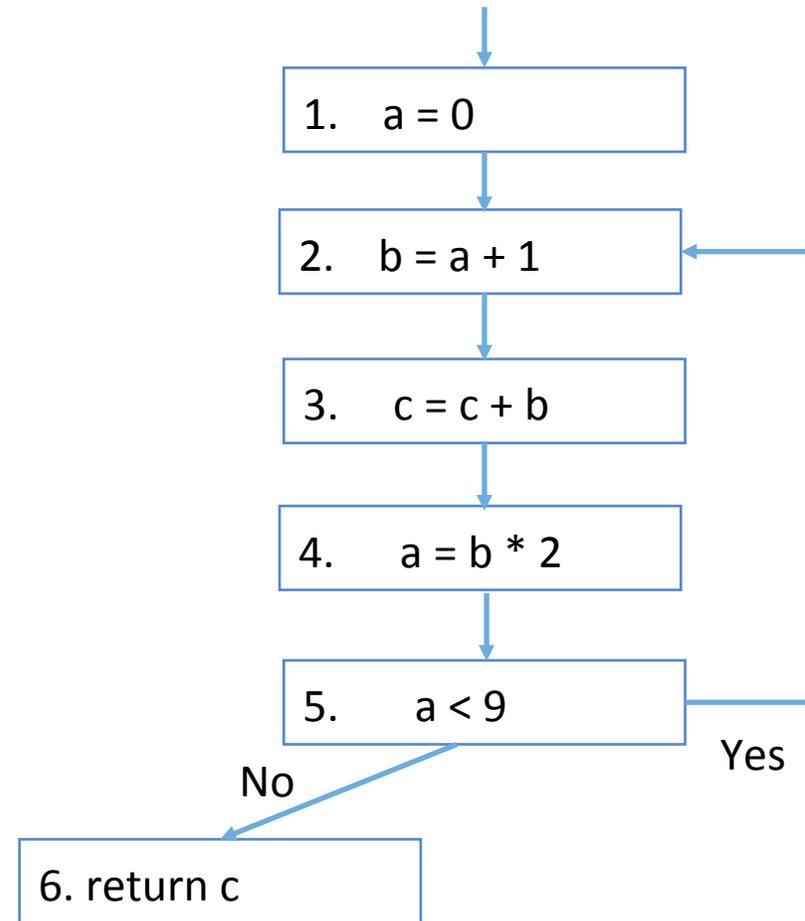
Computing Liveness Example

node #	use	def	1st		2nd		3rd		4th		5th		6th		7th	
			in	out												
1	a				a		a		ac		c	ac	c	ac	c	ac
2	a	b	a		a	bc	ac	bc								
3	bc	c	bc		bc	b	bc	b	bc	b	bc	b	bc	bc	bc	bc
4	b	a	b		b	a	b	a	b	ac	bc	ac	bc	ac	bc	ac
5	a		a	a	a	ac										
6	c		c		c		c		c		c		c		c	



Iterating Backwards: Converges Faster

node #	use	def	1st		2nd		3rd	
			out	in	out	in	out	in
6	c			c		c		
5	a		c	ac	ac	ac	ac	
4	b	a	ac	bc	ac	bc	ac	bc
3	bc	c	bc	bc	bc	bc	bc	bc
2	a	b	bc	ac	bc	ac	bc	ac
1		a	ac	c	ac	c	ac	c



Liveness Example: Round1

A variable is **live** at a particular point in the program if its value at that point will be used in the future (**dead**, otherwise).

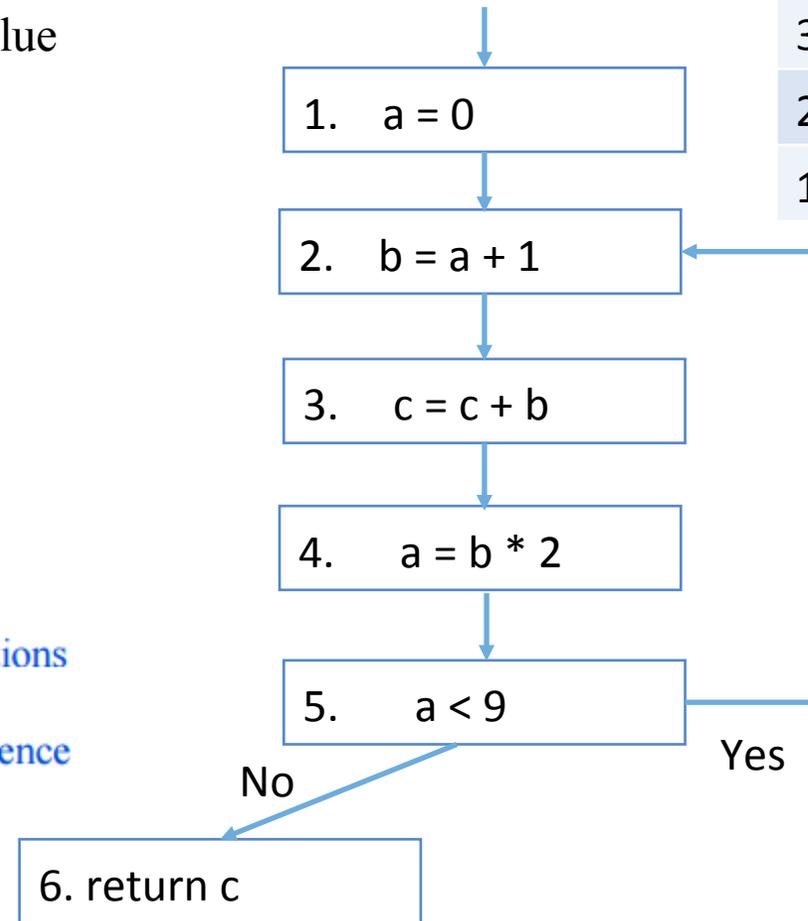
Algorithm

```

for each node n in CFG
    in[n] = ∅; out[n] = ∅
repeat
    for each node n in CFG in reverse topsort order
        in'[n] = in[n]
        out'[n] = out[n]
        out[n] =  $\bigcup_{s \in \text{succ}[n]} \text{in}[s]$ 
        in[n] = use[n]  $\cup$  (out[n] - def[n])
    until in'[n]=in[n] and out'[n]=out[n] for all n
  
```

} Initialize solutions
 } Save current results
 } Solve data-flow equations
 } Test for convergence

Node	use	def
6	c	
5	a	
4	b	a
3	bc	c
2	a	b
1		a



Liveness Example: Round1

Node	use	def
6	c	
5	a	
4	b	a
3	bc	c
2	a	b
1		a

Algorithm

```

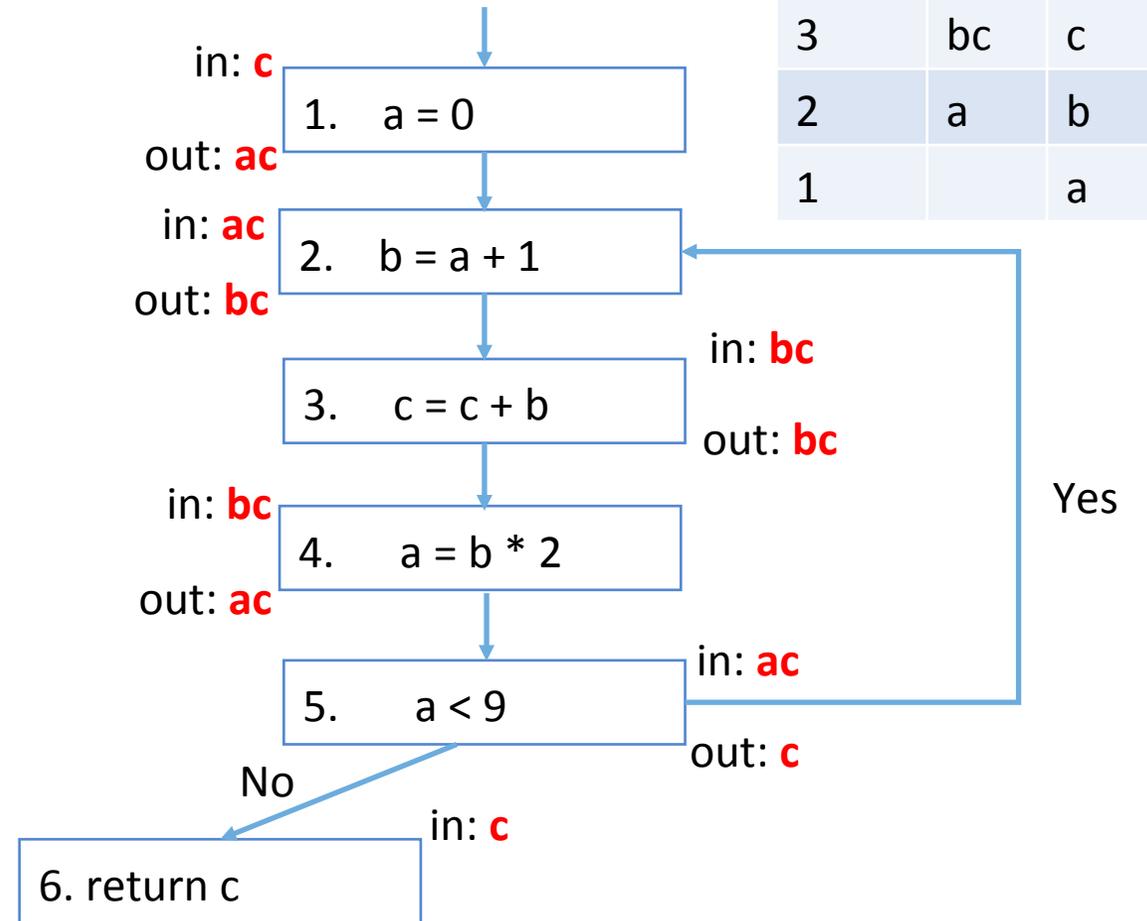
for each node n in CFG
    in[n] = ∅; out[n] = ∅
repeat
    for each node n in CFG in reverse topsort order
        in'[n] = in[n]
        out'[n] = out[n]
        out[n] =  $\bigcup_{s \in \text{succ}[n]} \text{in}[s]$ 
        in[n] = use[n]  $\cup$  (out[n] - def[n])
    until in'[n]=in[n] and out'[n]=out[n] for all n
  
```

Initialize solutions

Save current results

Solve data-flow equations

Test for convergence



Liveness Example: Round1

Node	use	def
6	c	
5	a	
4	b	a
3	bc	c
2	a	b
1		a

Algorithm

```

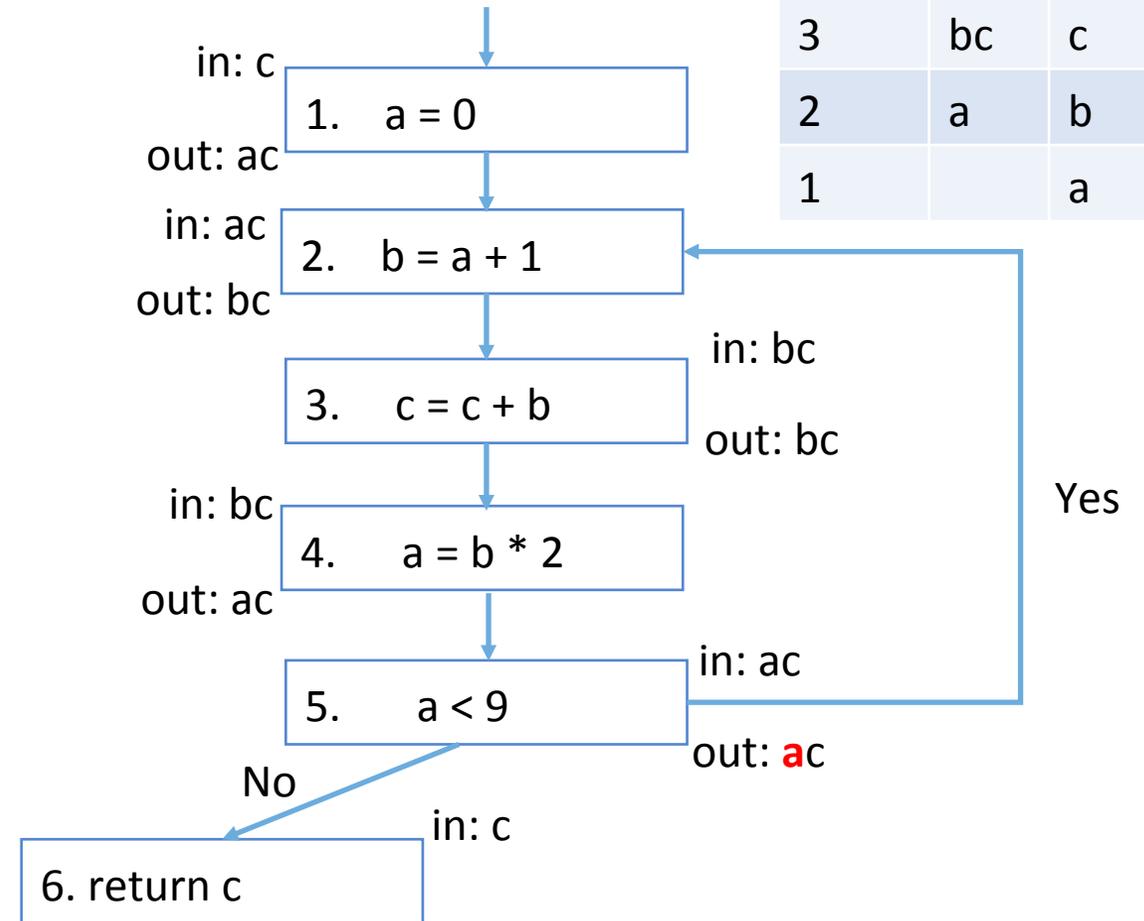
for each node n in CFG
    in[n] = ∅; out[n] = ∅
repeat
    for each node n in CFG in reverse topsort order
        in'[n] = in[n]
        out'[n] = out[n]
        out[n] =  $\bigcup_{s \in \text{succ}[n]} \text{in}[s]$ 
        in[n] = use[n]  $\cup$  (out[n] - def[n])
    until in'[n]=in[n] and out'[n]=out[n] for all n
  
```

Initialize solutions

Save current results

Solve data-flow equations

Test for convergence

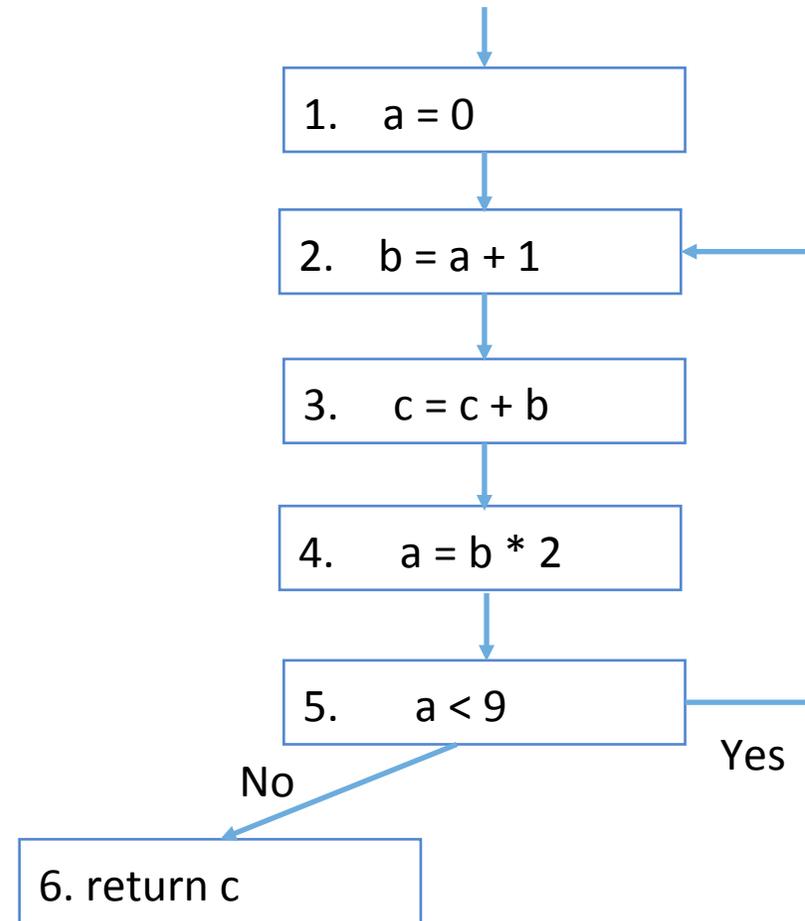


Conservative Approximation

node #	use	def	X		Y		Z	
			in	out	in	out	in	out
1		a	c	ac	cd	acd	c	ac
2	a	b	ac	bc	acd	bcd	ac	b
3	bc	c	bc	bc	bcd	bcd	b	b
4	b	a	bc	ac	bcd	acd	b	ac
5	a		ac	ac	acd	acd	ac	ac
6	c		c		c		c	

Solution X:

- From the previous slide



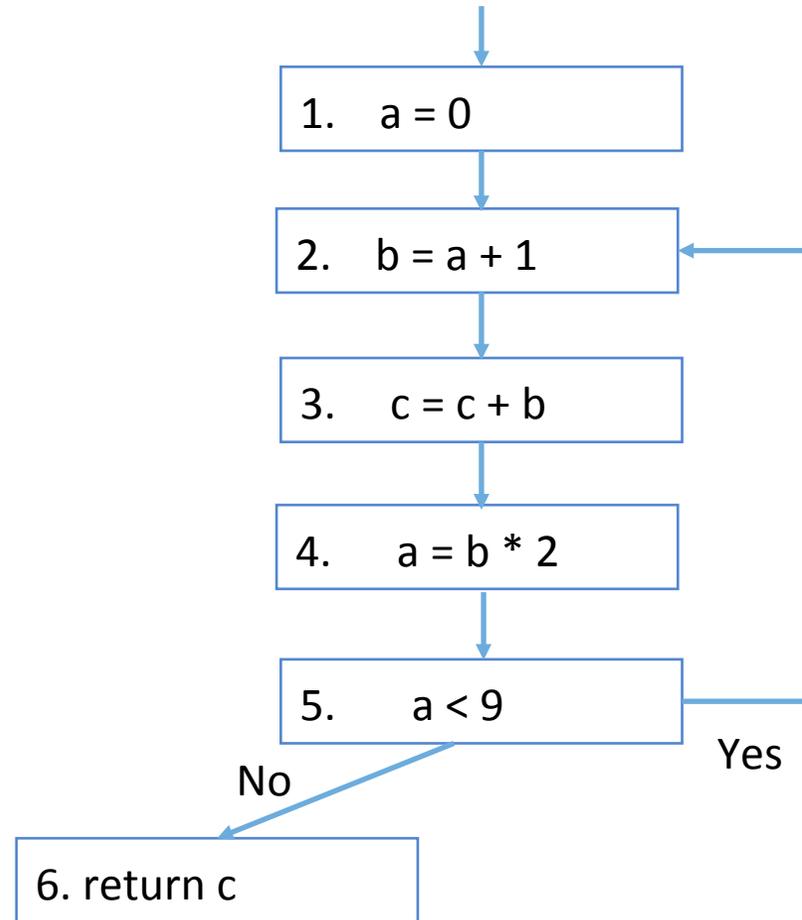
Conservative Approximation

node #	use	def	X		Y		Z	
			in	out	in	out	in	out
1		a	c	ac	cd	acd	c	ac
2	a	b	ac	bc	acd	bcd	ac	b
3	bc	c	bc	bc	bcd	bcd	b	b
4	b	a	bc	ac	bcd	acd	b	ac
5	a		ac	ac	acd	acd	ac	ac
6	c		c		c		c	

Solution Y:

Carries variable d uselessly

– Does Y lead to a correct program?



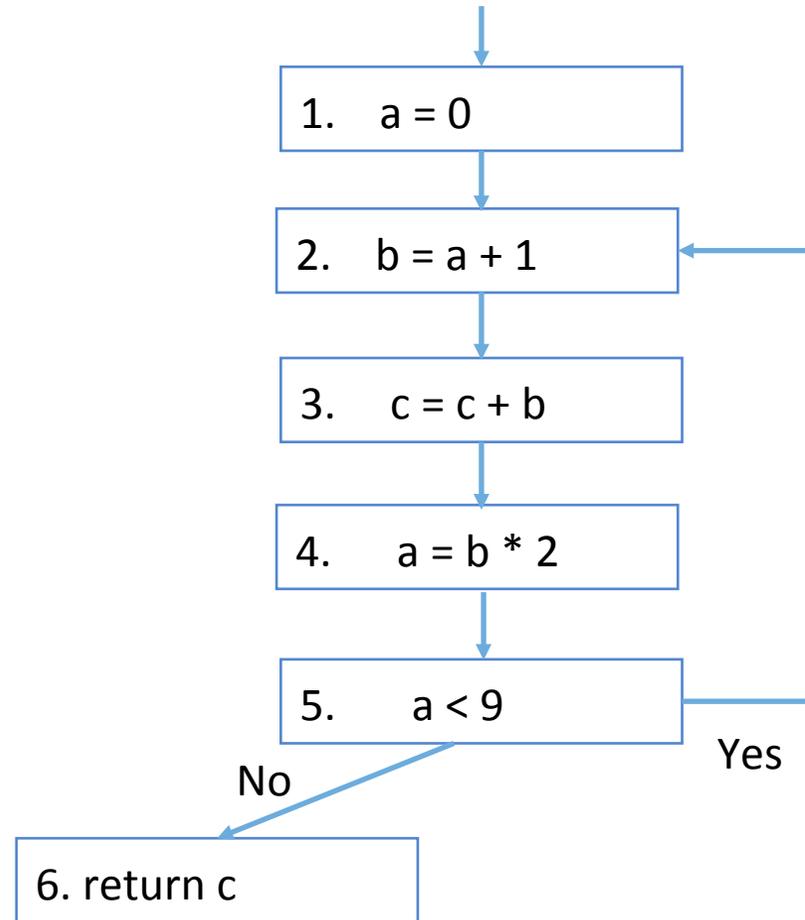
Imprecise conservative solutions \Rightarrow sub-optimal but correct programs

Conservative Approximation

node #	use	def	X		Y		Z	
			in	out	in	out	in	out
1		a	c	ac	cd	acd	c	ac
2	a	b	ac	bc	acd	bcd	ac	b
3	bc	c	bc	bc	bcd	bcd	b	b
4	b	a	bc	ac	bcd	acd	b	ac
5	a		ac	ac	acd	acd	ac	ac
6	c		c		c		c	

Solution Z:

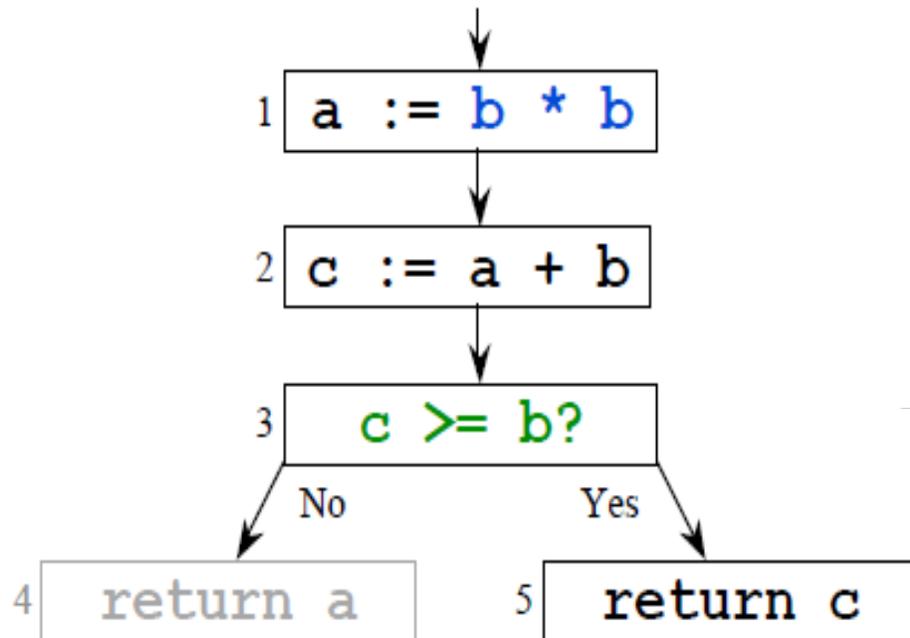
Does not identify c as live in all cases
 – Does Z lead to a correct program?



Non-conservative solutions \Rightarrow incorrect programs

Need for approximation

- Static vs. Dynamic Liveness: $\mathbf{b} * \mathbf{b}$ is always non-negative, so $\mathbf{c} \geq \mathbf{b}$ is always true and \mathbf{a} 's value will never be used after node

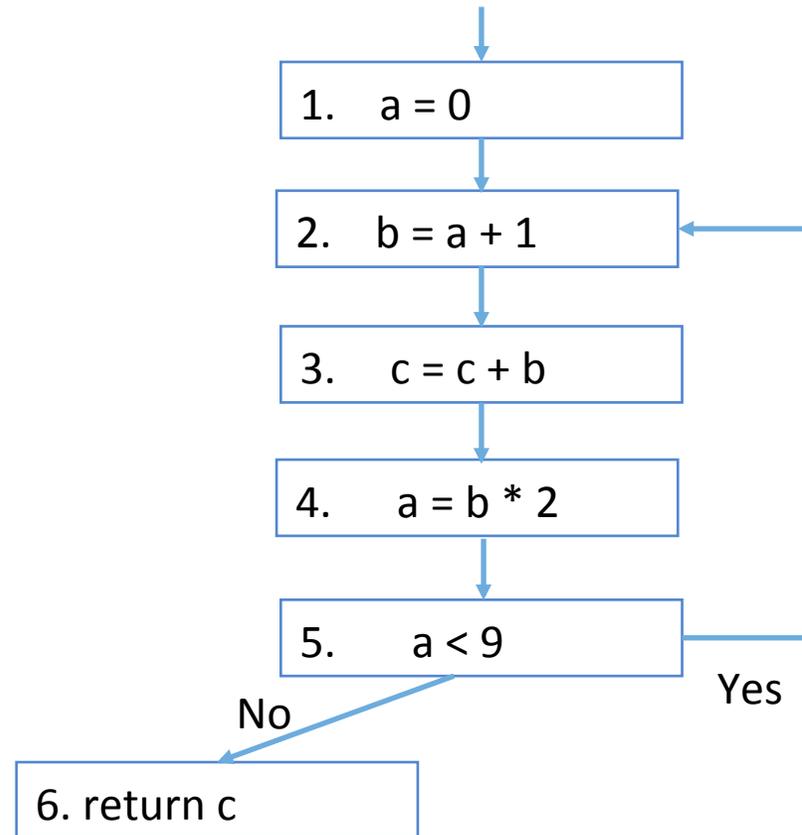


No compiler can statically identify all infeasible paths

Liveness Analysis Example Summary

- Live range of a
 - (1->2) and (4->5->2)
- Live range of b
 - (2->3->4)
- Live range of c
 - Entry->1->2->3->4->5->2, 5->6

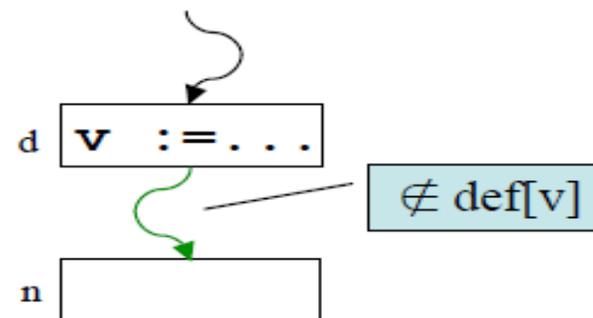
You need **2** registers **Why?**



Example 2: Reaching Definition

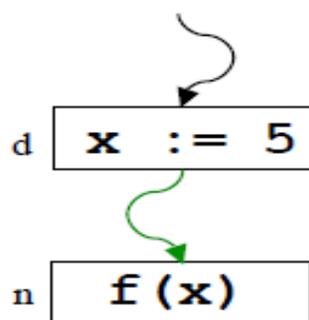
Definition

- A definition (statement) d of a variable v **reaches** node n if there is a path from d to n such that v is not redefined along that path



Uses of reaching definitions

- Build use/def chains
- Constant propagation
- Loop invariant code motion



Does this def of x reach n ?
Can we replace n with $f(5)$?

```
1  a = . . . ;  
2  b = . . . ;  
3  for ( . . . ) {  
4      x = a + b ;  
5      . . .  
6  }
```

Reaching definitions of a and b

To determine whether it's legal to move statement 4 out of the loop, we need to ensure that there are no reaching definitions of a or b inside the loop

Computing Reaching Definition

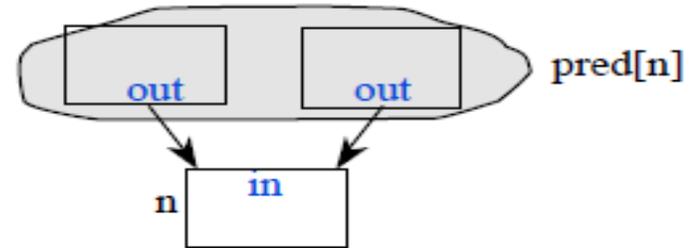
- Assumption: At most one definition per node
- **Gen[n]**: Definitions that are generated by node n (at most one)
- **Kill[n]**: Definitions that are killed by node n

<u>statement</u>	<u>gen's</u>	<u>kills</u>
$x:=y$	$\{y\}$	$\{x\}$
$x:=p(y,z)$	$\{y,z\}$	$\{x\}$
$x:=*(y+i)$	$\{y,i\}$	$\{x\}$
$*(v+i):=x$	$\{x\}$	$\{\}$
$x := f(y_1, \dots, y_n)$	$\{f, y_1, \dots, y_n\}$	$\{x\}$

Data-flow equations for Reaching Definition

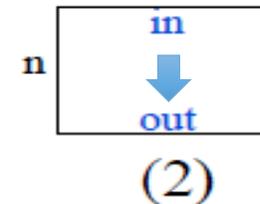
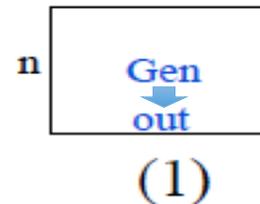
The in set

- A definition reaches the beginning of a node if it reaches the end of **any** of the predecessors of that node



The out set

- A definition reaches the end of a node if (1) the node itself **generates** the definition **or** if (2) the definition reaches the beginning of the node and the node does **not kill** it



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

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

Recall Liveness Analysis

- Data-flow Equation for liveness

$$\text{in}[n] = \text{use}[n] \cup (\text{out}[n] - \text{def}[n])$$

$$\text{out}[n] = \bigcup_{s \in \text{succ}[n]} \text{in}[s]$$

- Liveness equations in terms of Gen and Kill

$$\left. \begin{array}{l} \text{in}[n] = \text{gen}[n] \cup (\text{out}[n] - \text{kill}[n]) \\ \text{out}[n] = \bigcup_{s \in \text{succ}[n]} \text{in}[s] \end{array} \right\} \begin{array}{l} \text{A use of a variable generates liveness} \\ \text{A def of a variable kills liveness} \end{array}$$

Gen: New information that's added at a node

Kill: Old information that's removed at a node

Can define almost any data-flow analysis in terms of Gen and Kill

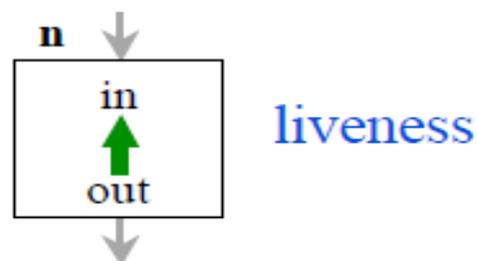
Direction of Flow

Backward data-flow analysis

- Information at a node is based on what happens **later** in the flow graph
i.e., $in[]$ is defined in terms of $out[]$

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

$$out[n] = \bigcup_{s \in succ[n]} in[s]$$

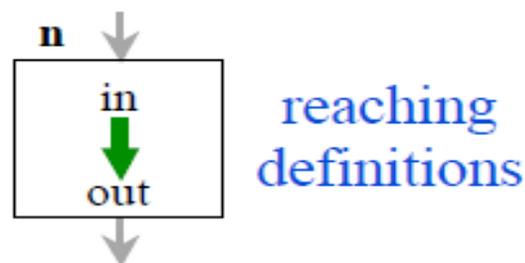


Forward data-flow analysis

- Information at a node is based on what happens **earlier** in the flow graph
i.e., $out[]$ is defined in terms of $in[]$

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

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



Some problems need both forward and backward analysis

- *e.g.*, Partial redundancy elimination (uncommon)

Data-Flow Equation for reaching definition

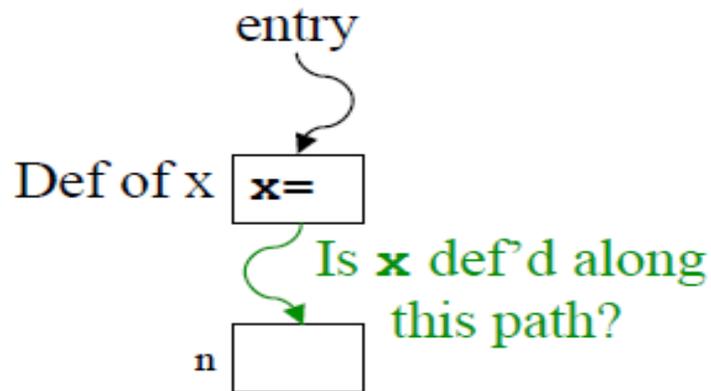
Symmetry between reaching definitions and liveness

- Swap in[] and out[] and swap the directions of the arcs

Reaching Definitions

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

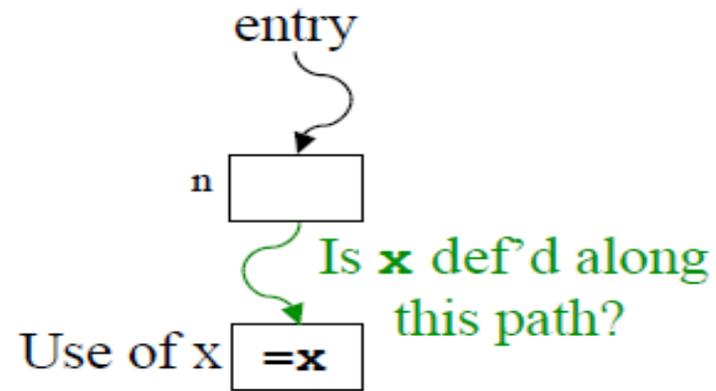
$$\text{out}[n] = \text{gen}[n] \cup (\text{in}[n] - \text{kill}[n])$$



Live Variables

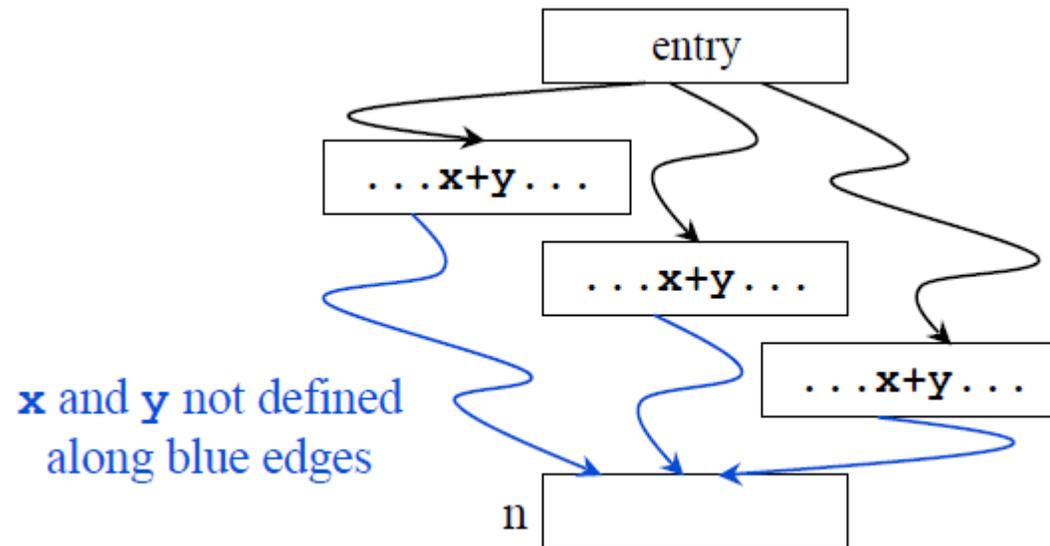
$$\text{out}[n] = \bigcup_{s \in \text{succ}[n]} \text{in}[s]$$

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



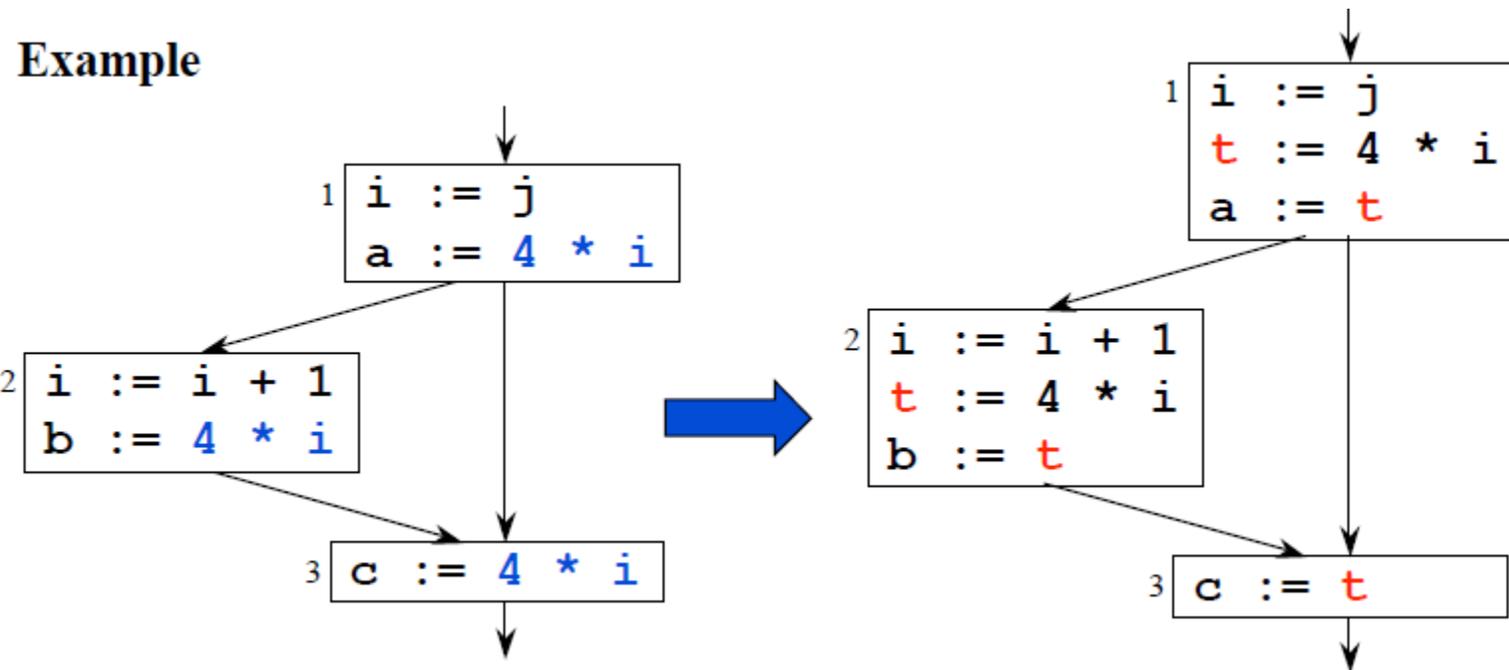
Available Expression

- An expression, $x+y$, is **available** at node n if every path from the entry node to n evaluates $x+y$, and there are no definitions of x or y after the last evaluation.



Available Expression for CSE

- Common Subexpression eliminated
 - If an expression is available at a point where it is evaluated, it need not be recomputed



Must vs. May analysis

- **May information:** Identifies possibilities
- **Must information:** Implies a guarantee

	May	Must
Forward	Reaching Definition	Available Expression
Backward	Live Variables	Very Busy Expression