PennState

CSE 597: Security of Emerging Technologies

Module: Program Analysis

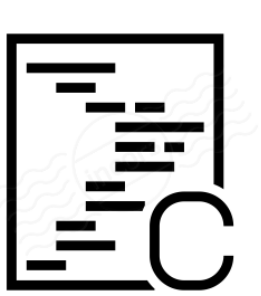
Prof. Syed Rafiul Hussain

Systems and Network Security (SyNSec) Research Group

Department of Computer Science and Engineering

The Pennsylvania State University

Our Goal



Source code

Security bugs

Program analyzer must be able to understand program properties (e.g., can a variable be NULL at a particular program point?)

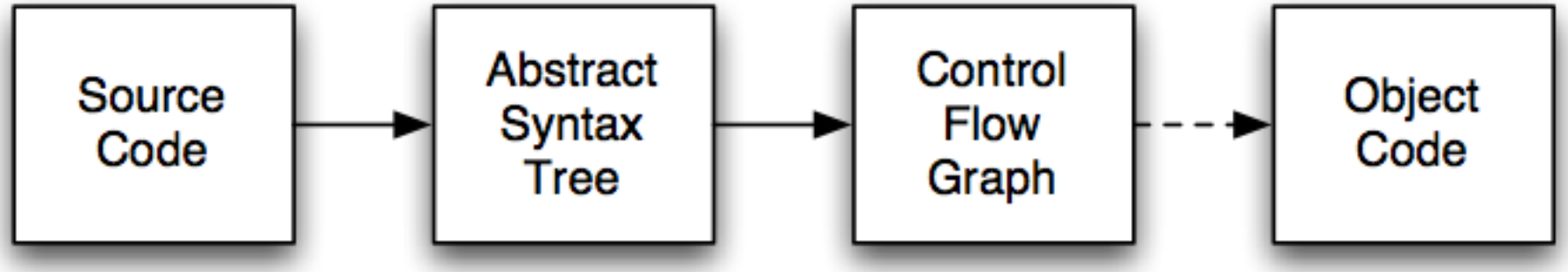
Must perform control and data flow analysis

Do we need to implement control and data flow analysis from scratch?

- Most modern compilers already perform several types of such analysis for code optimization
  - We can hook into different layers of analysis and customize them
  - We still need to understand the details

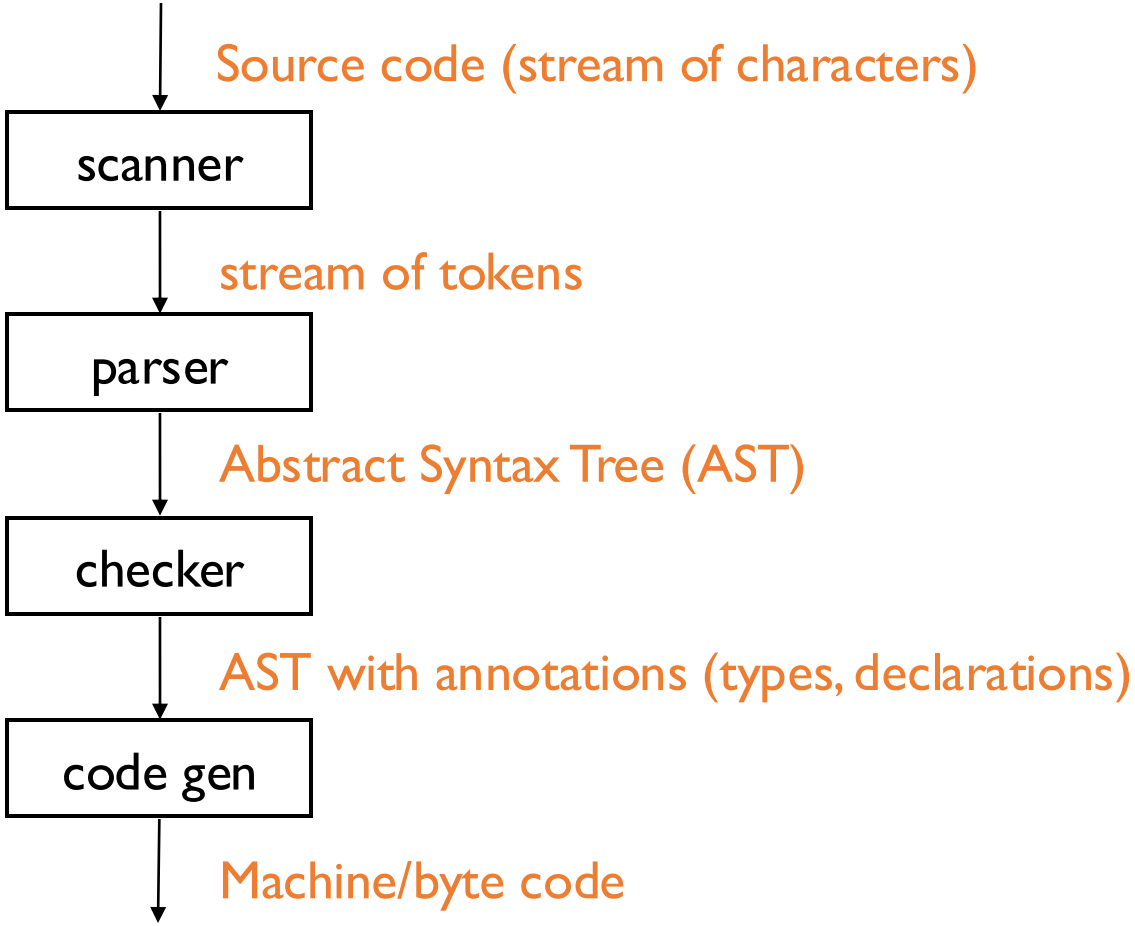
- LLVM (http://llvm.org/) is a highly customizable and modular compiler framework
  - Users can write LLVM passes to perform different types of analysis
  - Clang static analyzer can find several types of bugs
  - Can instrument code for dynamic analysis

Compiler Overview



- Abstract Syntax Tree : Source code parsed to produce AST
- Control Flow Graph: AST is transformed to CFG
- Data Flow Analysis: operates on CFG

The Structure of a Compiler



Syntactic Analysis

- **Input:** sequence of tokens from scanner
- **Output:** abstract syntax tree
- Actually,
  - parser first builds a parse tree, representation of grammars in a tree-like form.
  - AST is then built by translating the parse tree
  - parse tree rarely built explicitly; only determined by, say, how parser pushes stuff to stack

Example

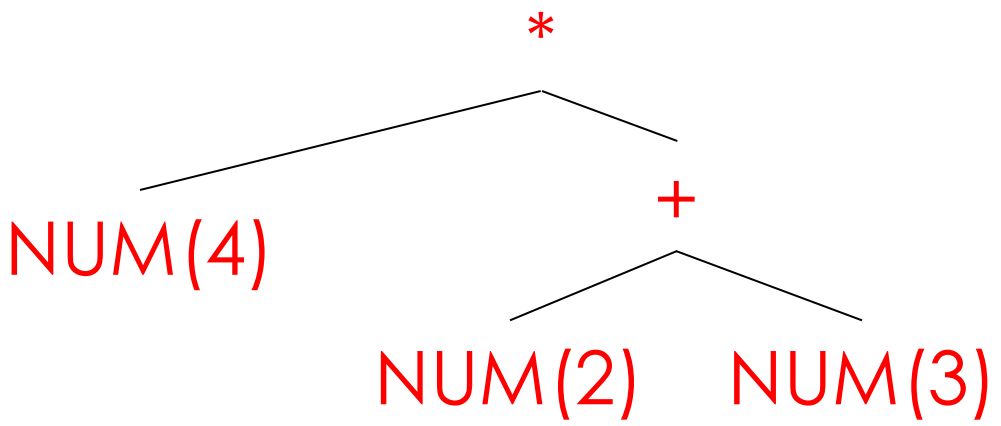
- Source Code

  4*(2+3)

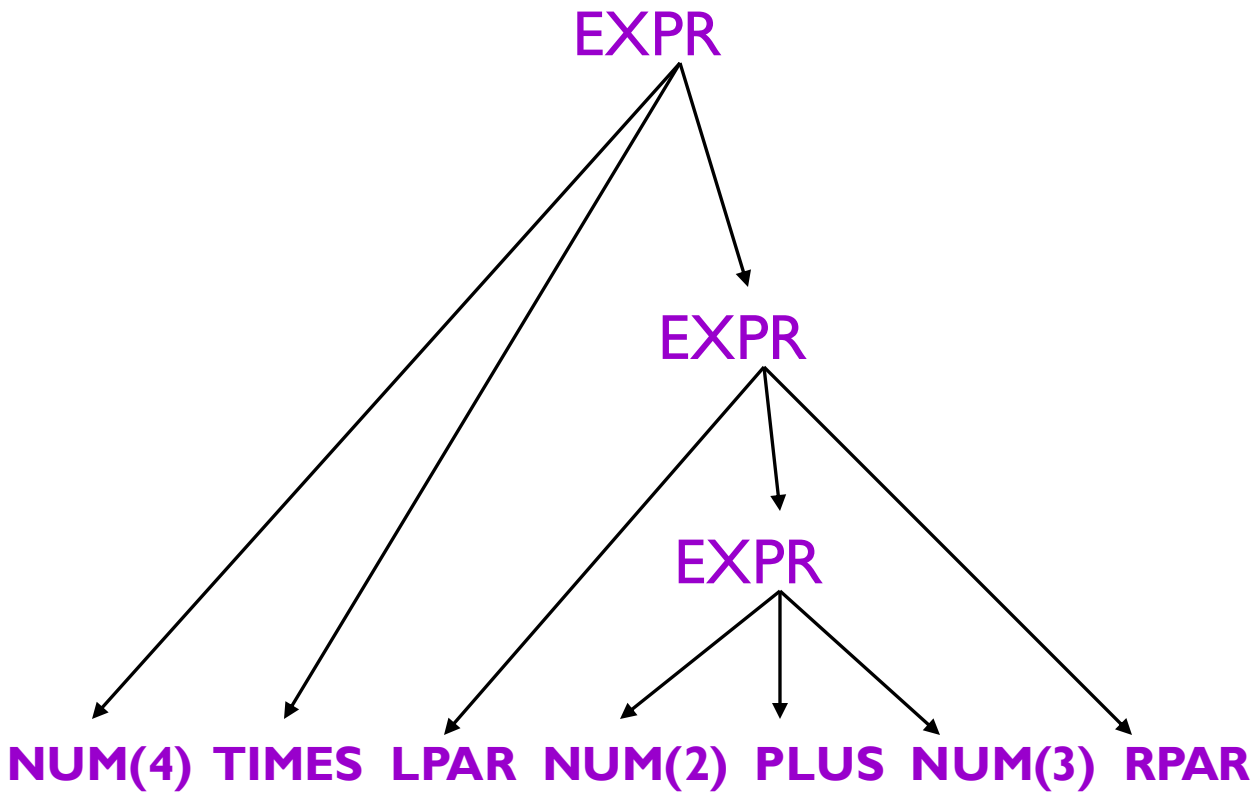
- Parser input

  NUM(4) TIMES LPAR NUM(2) PLUS NUM(3) RPAR

- Parser output (AST):



Parse tree for the example: 4*(2+3)



leaves are tokens

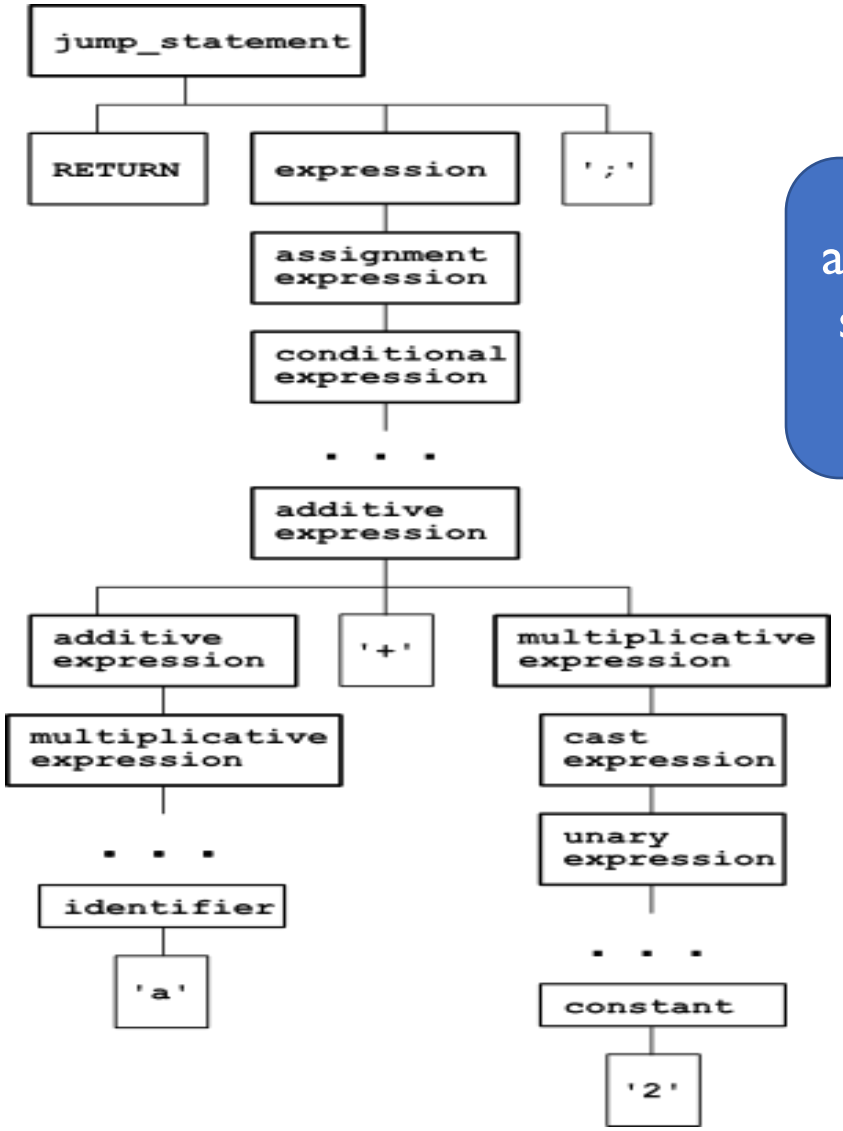
Parse Tree

- Representation of grammars in a tree-like form.

> A parse tree pictorially shows how the start symbol of a grammar derives a string in the language. … Dragon Book

- Is a one-to-one mapping from the grammar to a tree-form.

Parse Tree



C Statement: return a + 2

a very formal representation that strictly shows how the parser understands the statement return a + 2;

Abstract Syntax Tree (AST)

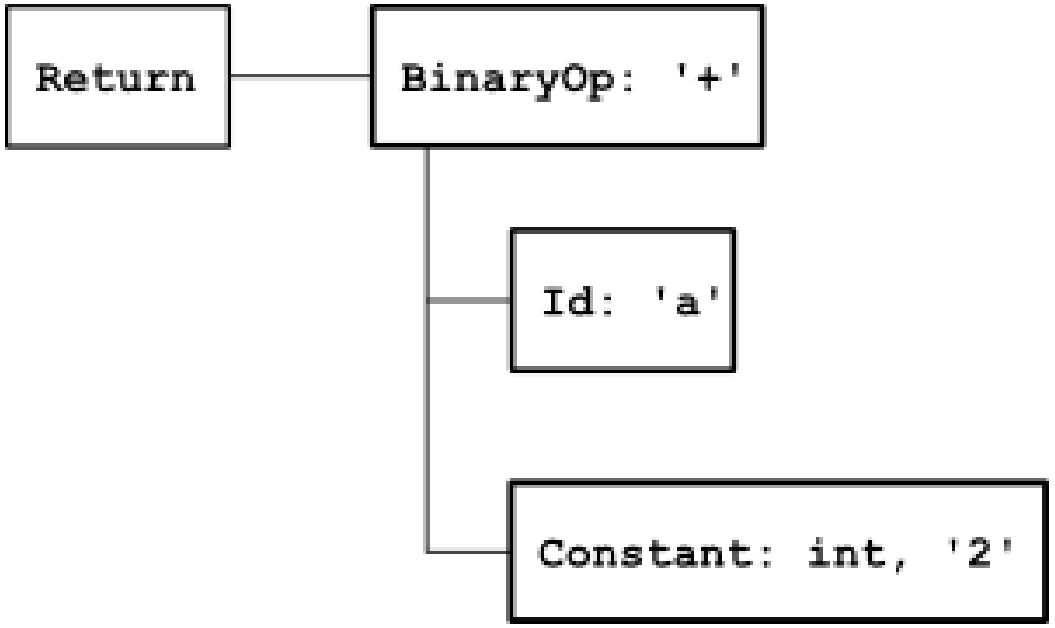
- Simplified syntactic representations of the source code, and they're most often expressed by the data structures of the language used for implementation

> ASTs differ from parse trees because superficial distinctions of form, unimportant for translation, do not appear in syntax trees.. … Dragon Book

- Without showing the whole syntactic clutter, represents the parsed string in a structured way, discarding all information that may be important for parsing the string, but isn't needed for analyzing it.

Abstract Syntax Tree (AST)

C Statement: return a + 2



Disadvantages of ASTs

- AST has many similar forms
  - E.g., for, while, repeat...until
  - E.g., if, ?:, switch

```
int x = 1 // what's the value of x ?
          // AST traversal can give the answer, right?

What about int x; x = 1; or int x= 0; x += 1;  ?
```

- Expressions in AST may be complex, nested
  - (x * y) + (z > 5 ? 12 * z : z + 20)

- Want simpler representation for analysis
  - ...at least, for dataflow analysis

Control Flow Graph & Analysis

High-level representation

–Control flow is implicit in an AST

Low-level representation:

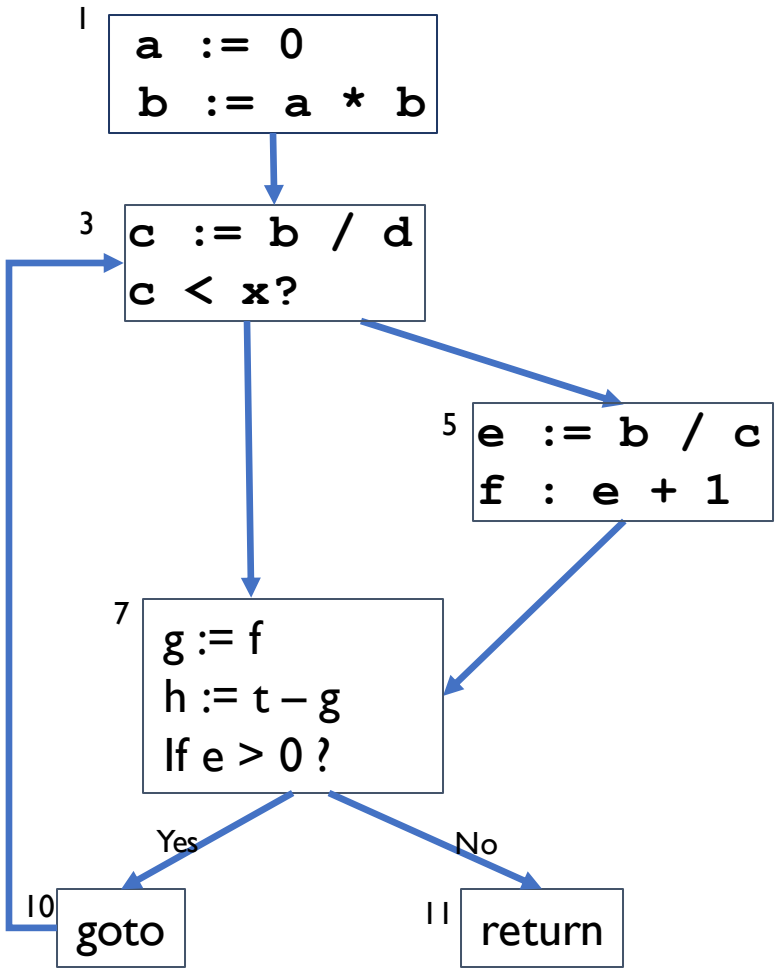
–Use a **Control-flow graph (CFG)**

–Nodes represent statements (low-level linear IR)

–Edges represent explicit flow of control

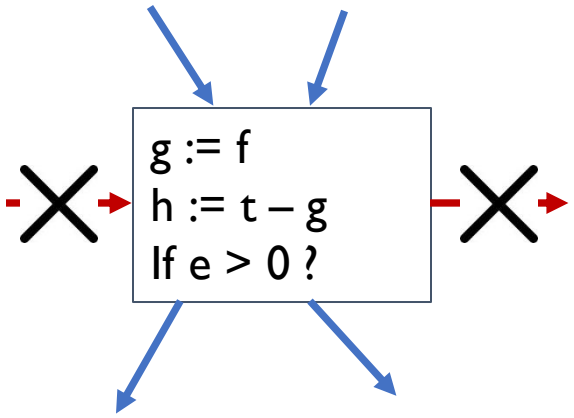
What Is Control-Flow Analysis?

```
1         a := 0
          b := a * b
2
3   L1: c := b/d
4         if c < x goto L2
5         e := b / c
          f := e + 1
6
7   L2: g := f
8         h := t - g
          if e > 0 goto L3
9
10  goto L1
11  L3: return
```



Basic Blocks

- A **basic block** is a sequence of straight line code that can be entered only at the beginning and exited only at the end



- **Building basic blocks**
  - Identify **leaders**
    - The first instruction in a procedure, or
    - The target of any branch, or
    - An instruction immediately following a branch (implicit target)
  - Gobble all subsequent instructions until the next leader

Basic Block Example

```
1         a := 0
2         b := a * b
3   L1: c := b/d
4         if c < x goto L2
5         e := b / c
6         f := e + 1
7   L2: g := f
8         h := t - g
9         if e > 0 goto L3
10  goto L1
11  L3: return
```

Leaders?

Blocks?

Basic Block Example

```
1        a := 0
2        b := a * b
3   L1:  c := b/d
4        if c < x goto L2
5        e := b / c
6        f := e + 1
7   L2:  g := f
8        h := t - g
9        if e > 0 goto L3
10  goto L1
11  L3:  return
```

Leaders?

– {1, 3, 5, 7, 10, 11}

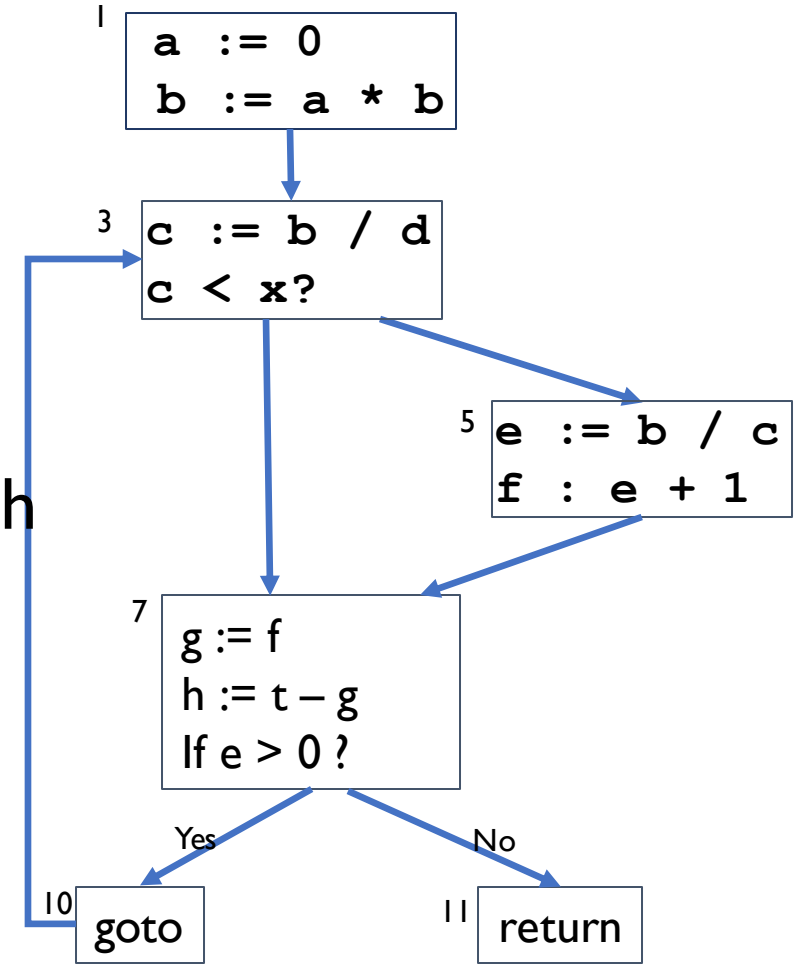
Blocks?

– {1,2}
– {3,4}
– {5,6}
– {7, 8, 9}
– {10}
– {11}

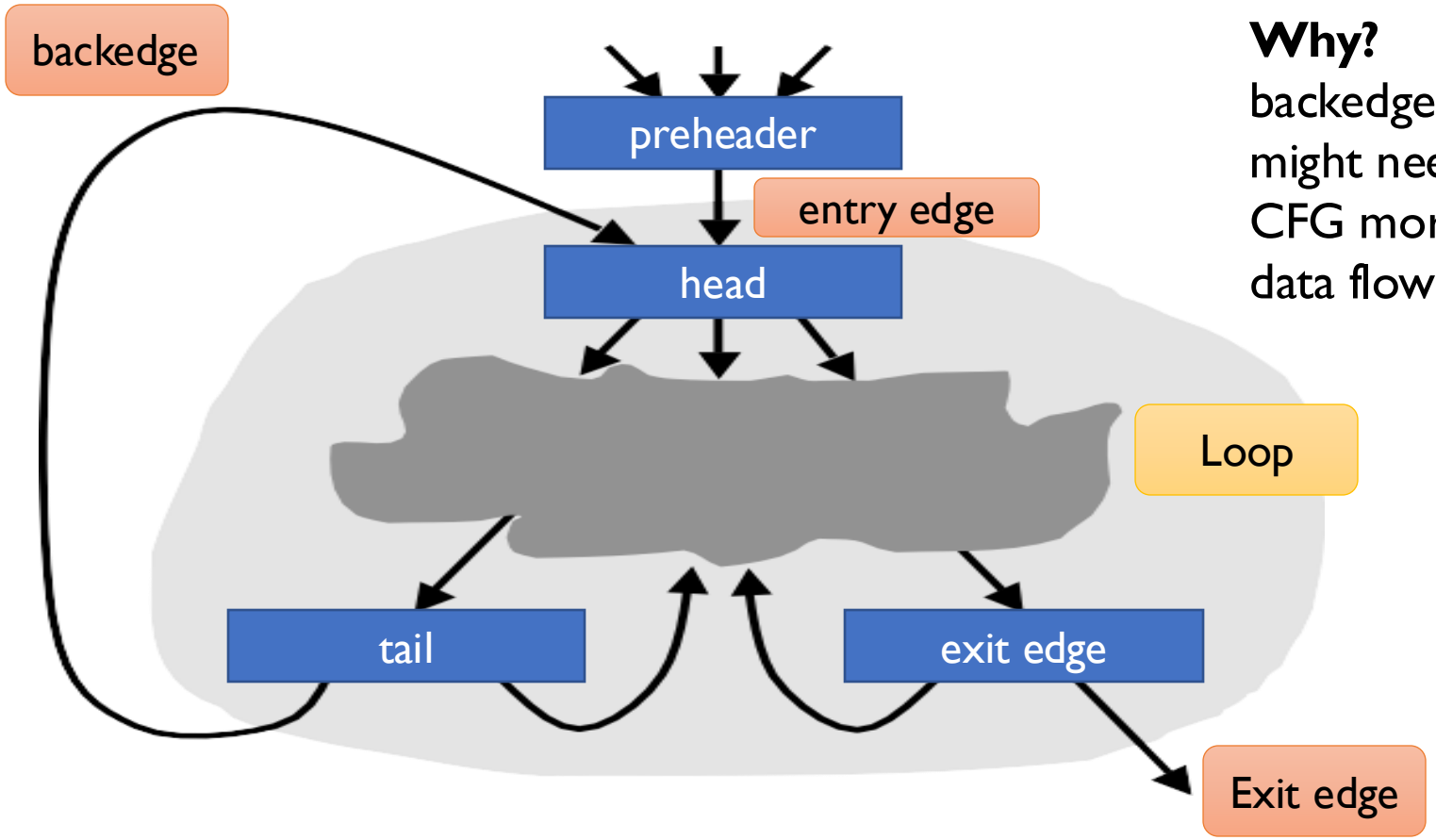
Building a CFG From Basic Block

Construction

- Each CFG node represents a basic block
- There is an edge from node i to j if
  - Last statement of block i branches to the first statement of j, or
  - Block i does **not** end with an unconditional branch and is immediately followed in program order by block j (fall through)

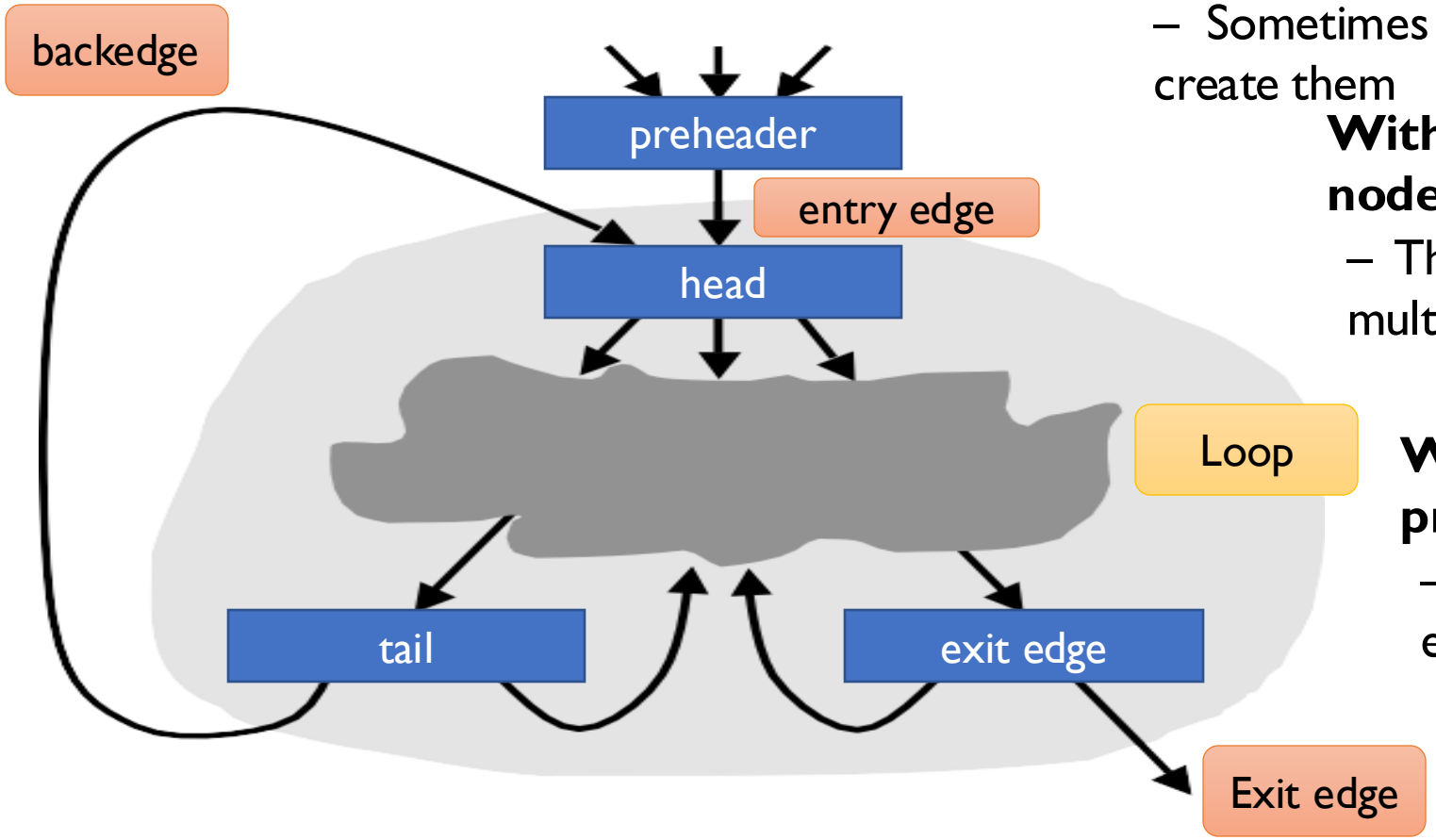


Looping



Why?
backedges indicate that we might need to traverse the CFG more than once for data flow analysis

Looping



Not all loops have preheaders

– Sometimes it is useful to create them

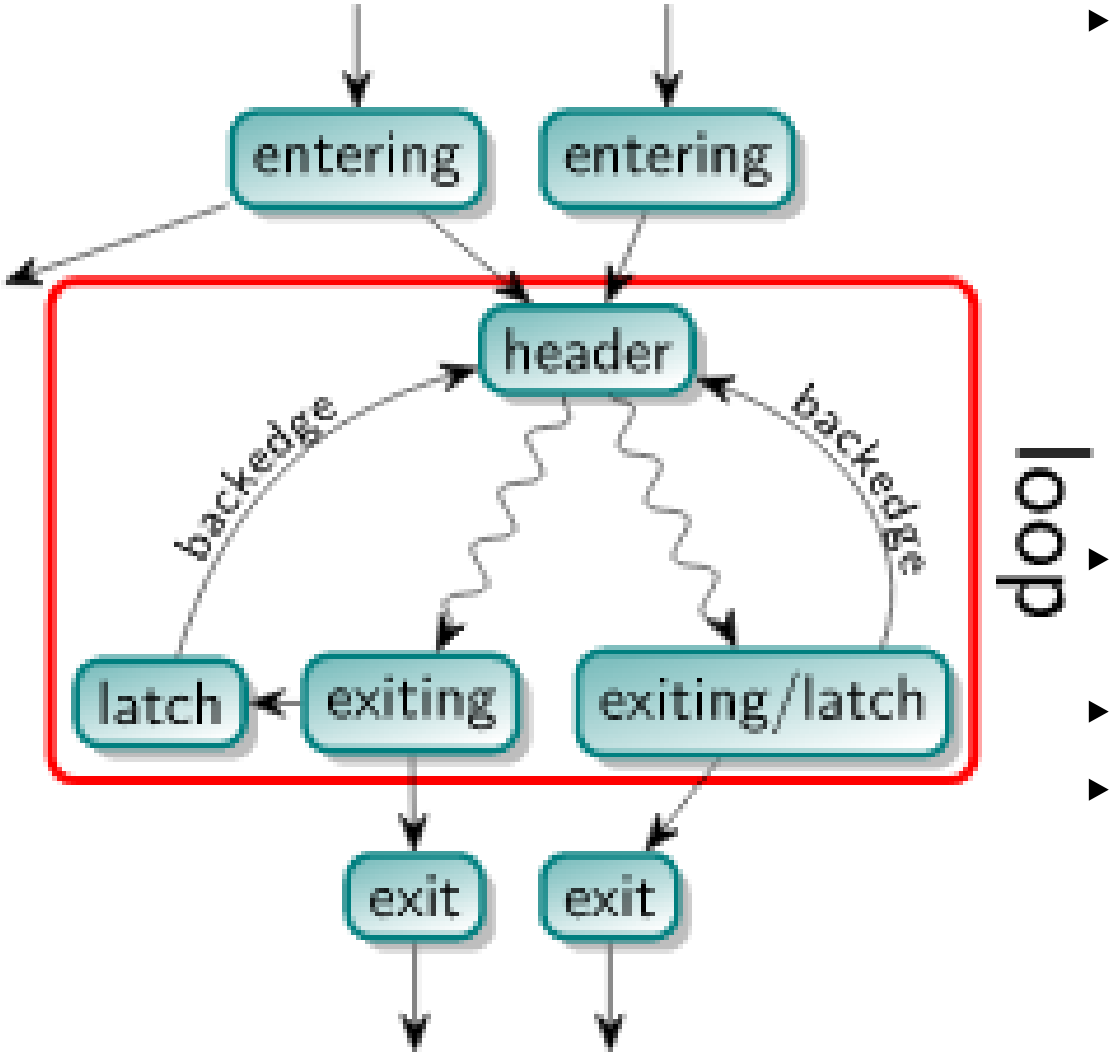
Without preheader node

– There can be multiple entry edges

With single preheader node

– There is only one entry edge

Looping



- An **entering block** (or **loop predecessor**) is a non-loop node that has an edge into the loop (necessarily the header). If there is only one entering block entering block, and its only edge is to the header, it is also called the loop's **preheader**. The **preheader** dominates the loop without itself being part of the loop.
- A **latch** is a loop node that has an edge to the header.
- A **backedge** is an edge from a latch to the header.
- An **exiting edge** is an edge from inside the loop to a node outside of the loop. The source of such an edge is called an **exiting block**, its target is an **exit block**.

Dominators

- d **dom** i if all paths from entry to node i include d
- Strict Dominator (d sdom i)
  - If d dom i, but d != i
- Immediate dominator (a idom b)
  - a sdom b and there does not exist any node c such that a != c, c != b, a dom c, c dom b
- Post dominator (p pdom i)
  - If every possible path from i to exit includes p

Identifying Natural Loops and Dominators

- Back Edge
  - A **back edge** of a natural loop is one whose target dominates its source

- Natural Loop
  - The **natural loop** of a back edge (m→n), where n dominates m, is the set of nodes x such that n dominates x and there is a path from x to m not containing n

Why go through all this trouble?

- **Modern languages provide structured control flow**
  - Shouldn't the compiler remember this information rather than throw it away and then re-compute it?
- **Answers?**
  - We may want to work on the binary code
  - Most modern languages still provide a **goto** statement
  - Languages typically provide multiple types of loops. This analysis lets us treat them all uniformly
  - We may want a compiler with multiple front ends for multiple languages; rather than translating each language to a CFG, translate each language to a canonical IR and then to a CFG

Data flow analysis

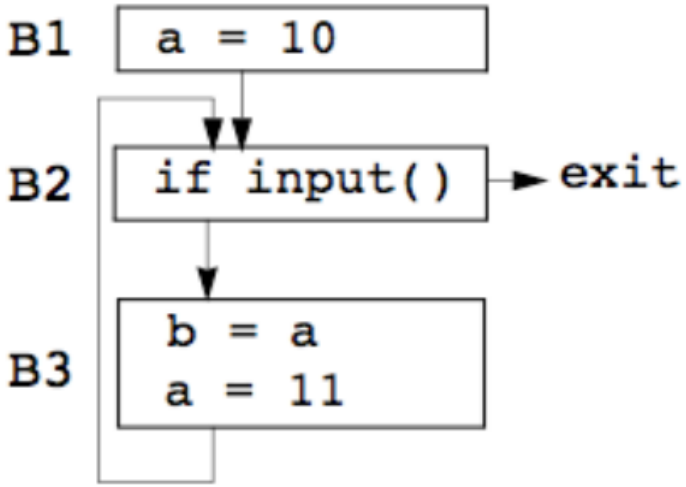
- Derives information about the **dynamic** behavior of a program by only examining the **static** code
- Intraprocedural analysis
- Flow-sensitive: sensitive to the control flow in a function

- **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

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

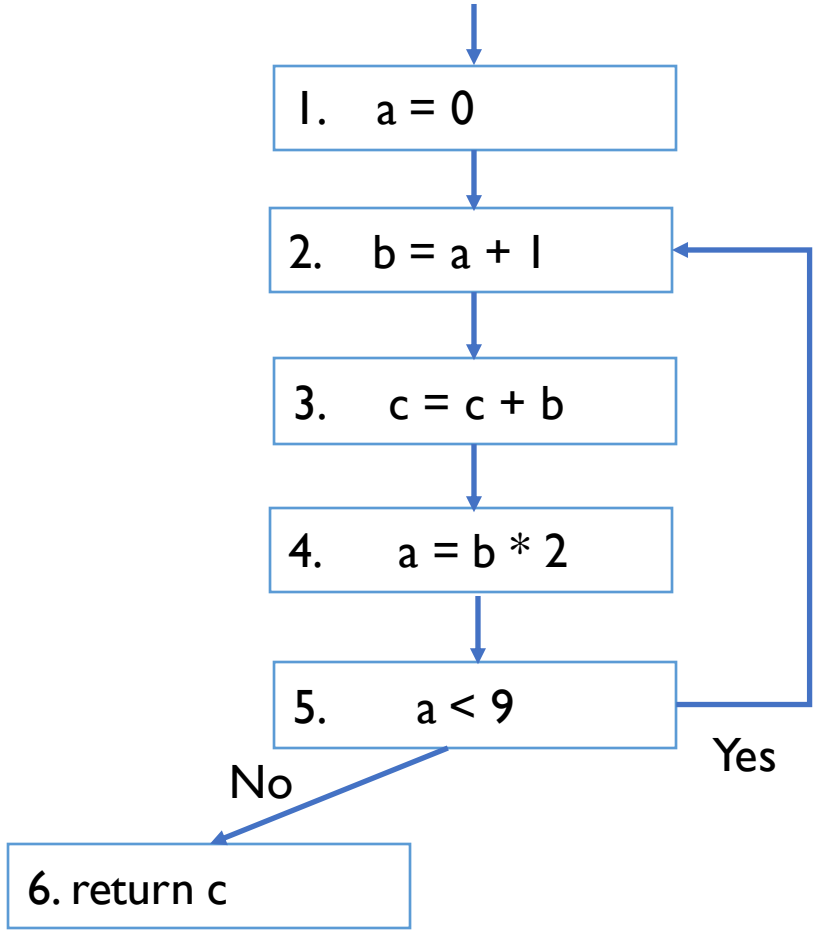
- A program contains an unbounded number of variables
- Must execute on a machine with a bounded number of registers
- Two variables can use the same register if they are never in use at the same time (*i.e*, never simultaneously live).

  –Register allocation uses liveness information

Control Flow Graph

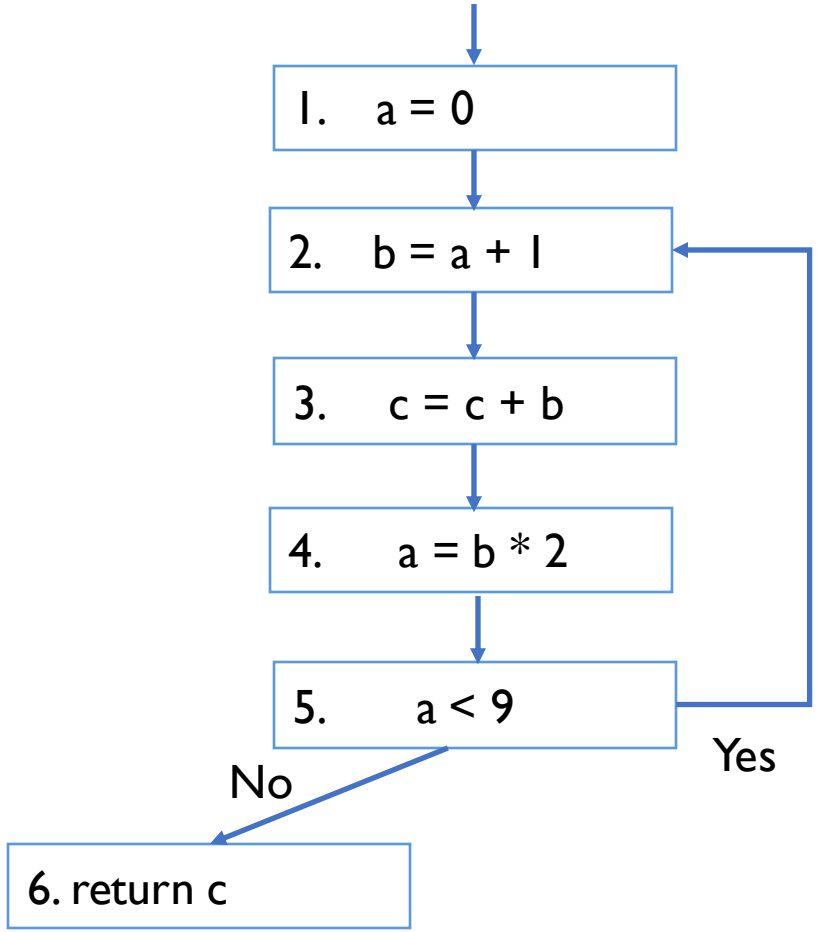
- Let's consider CFG where nodes contain program statement instead of basic block.
- Example

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



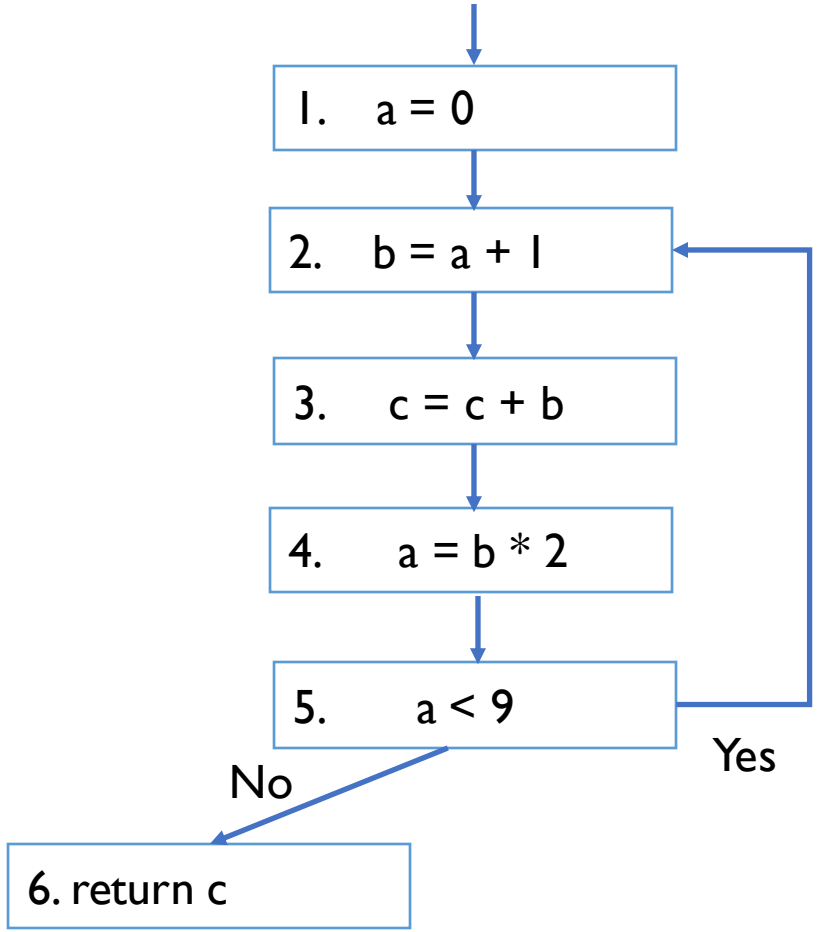
Liveness by Example

- 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)

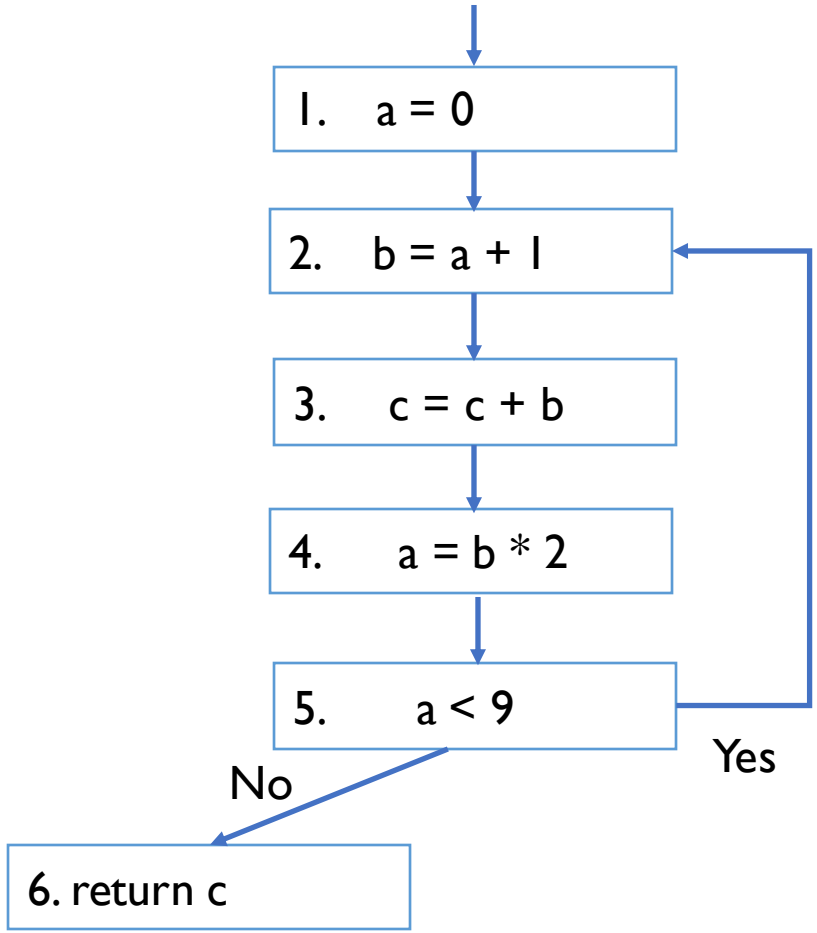


Terminology

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

Examples

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



Uses and Defs

Def (or definition)

–An **assignment** of a value to a variable
–def[v] = set of CFG nodes that define variable v
–def[n] = set of variables that are defined at node n

```
a = 0
```

Use

–A **read** of a variable's value
–use[v] = set of CFG nodes that use variable v
–use[n] = set of variables that are used at node n

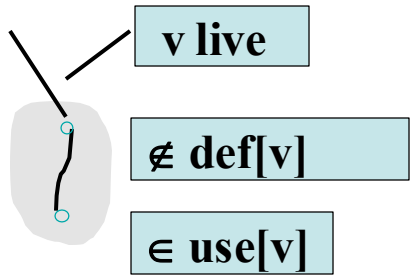
```
a < 9
```

More precise definition of liveness

– A variable v is live on a CFG edge if

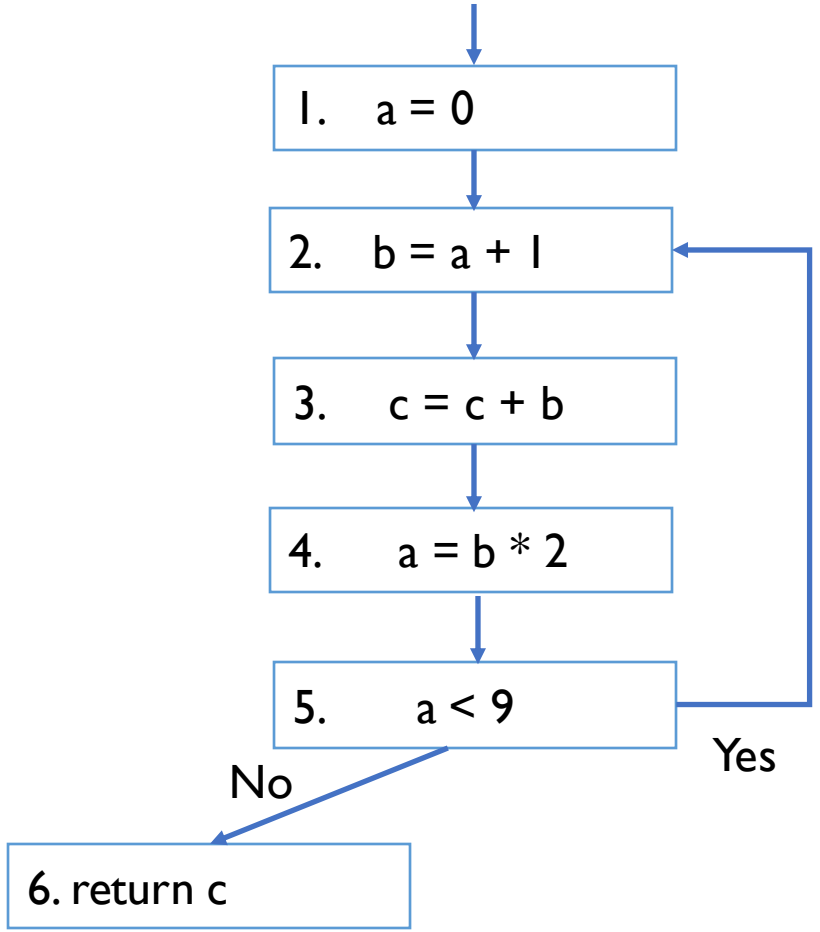
(1)∃ a directed path from that edge to a use of v (node in use[v]), **and**

(2)that path does not go through any def of v (no nodes in def[v])

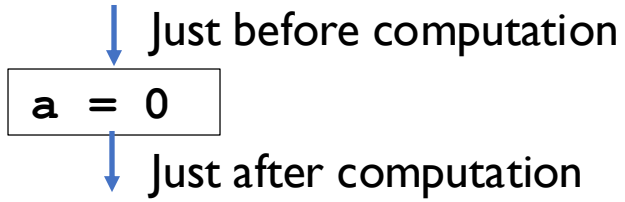


The Flow of Liveness

- 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

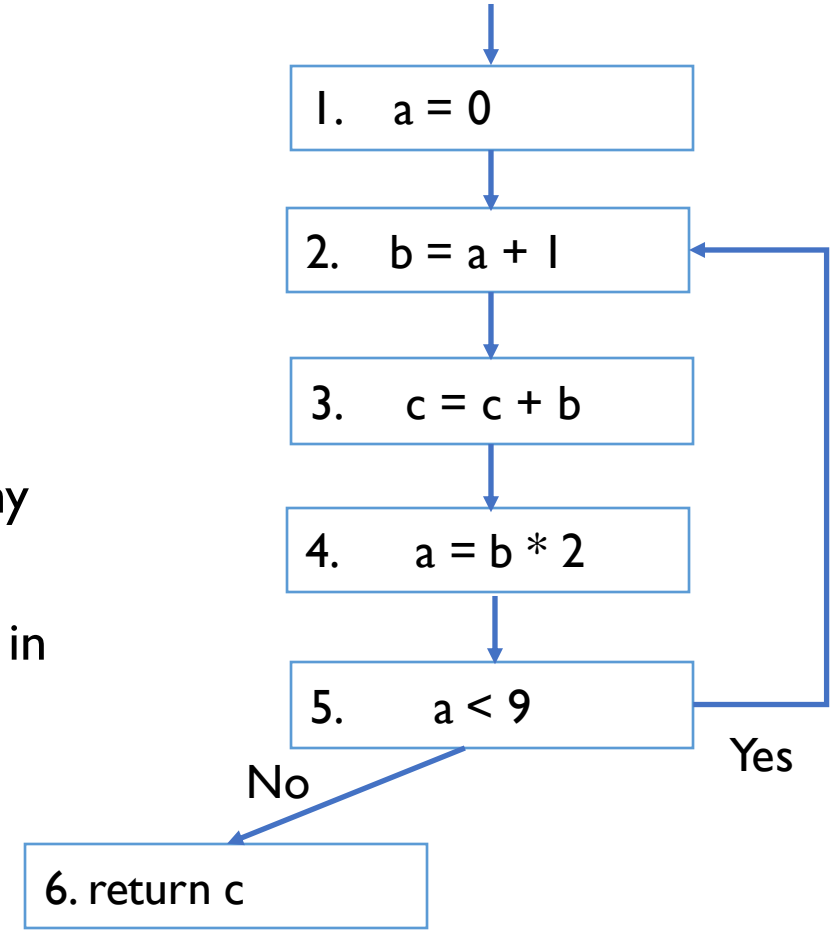


Liveness at Nodes



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

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

Solving Dataflow Equation

for each node n in CFG

in[n] = ∅; out[n] = ∅ } Initialize solutions

repeat

for each node n in CFG

in'[n] = in[n] } Save current results

out'[n] = out[n]

in[n] = use[n] ∪ (out[n] – def[n]) } Solve data-flow equation

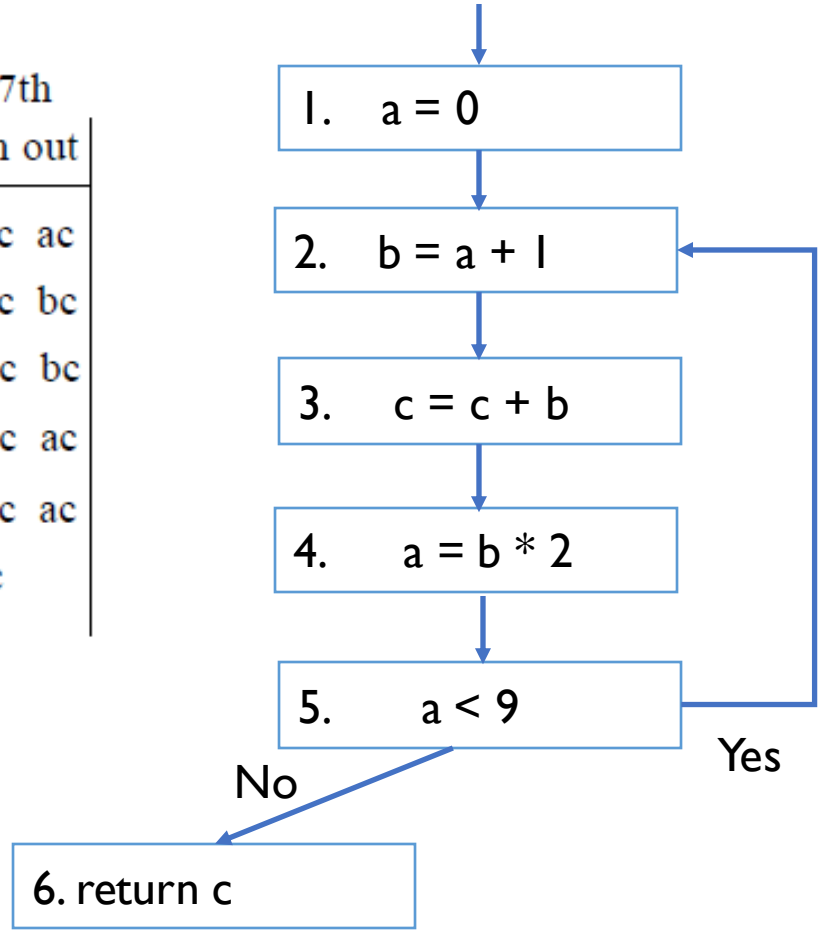
out[n] = ∪ in[s]

s ∈ succ[n] } Test for convergence

until in'[n]=in[n] and out'[n]=out[n] for all n

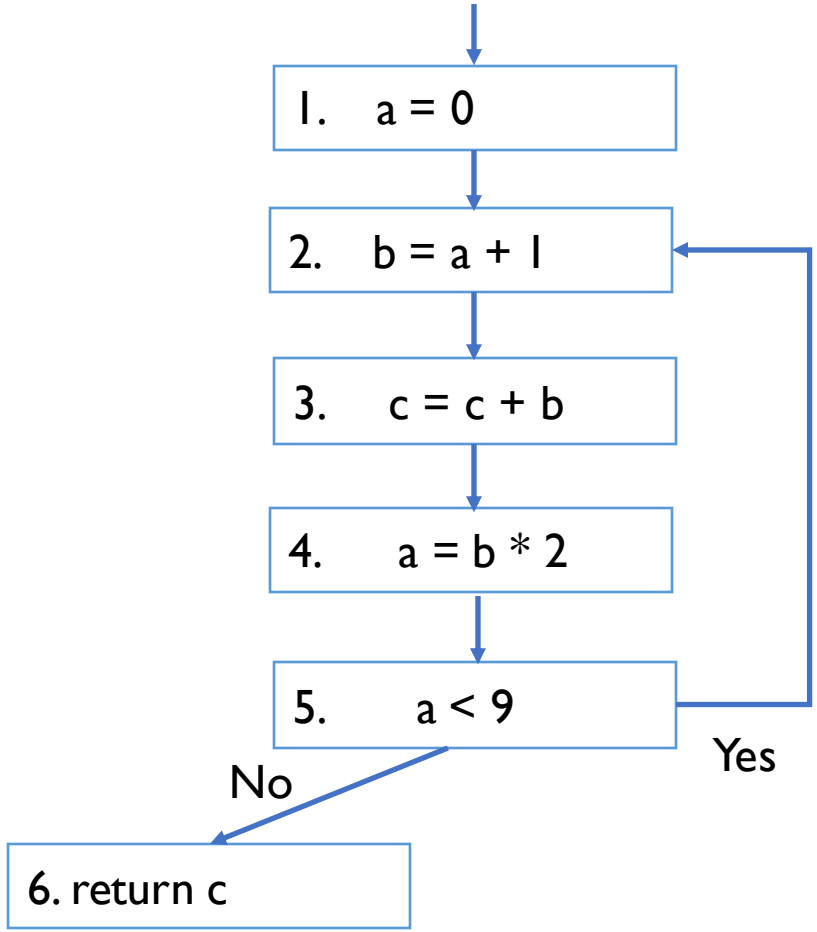
Computing Liveness Example

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



Iterating Backwards: Converges Faster

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

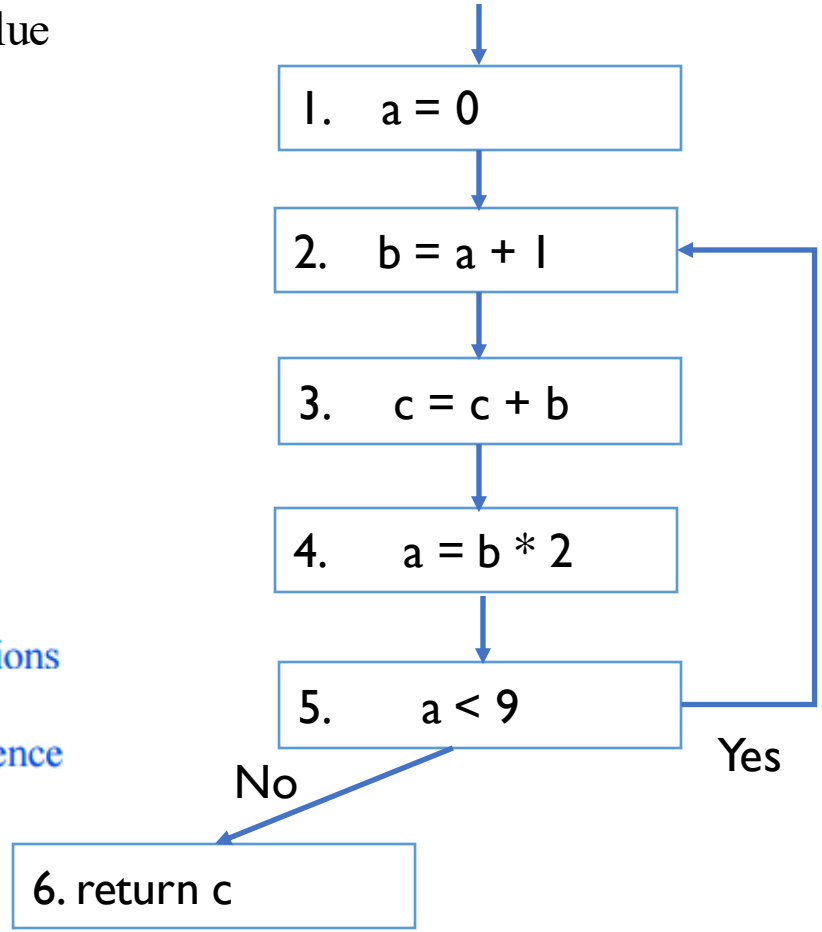


Liveness Example: Round 1

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] = ∅ } Initialize solutions
repeat
  for each node n in CFG **in reverse topsort order**
    in'[n] = in[n]
    out'[n] = out[n] } Save current results
    out[n] = $\bigcup_{s \in succ[n]}$ **in[s]**
    in[n] = use[n] ∪ (out[n] – def[n]) } Solve data-flow equations
until in'[n]=in[n] and out'[n]=out[n] for all n } Test for convergence



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

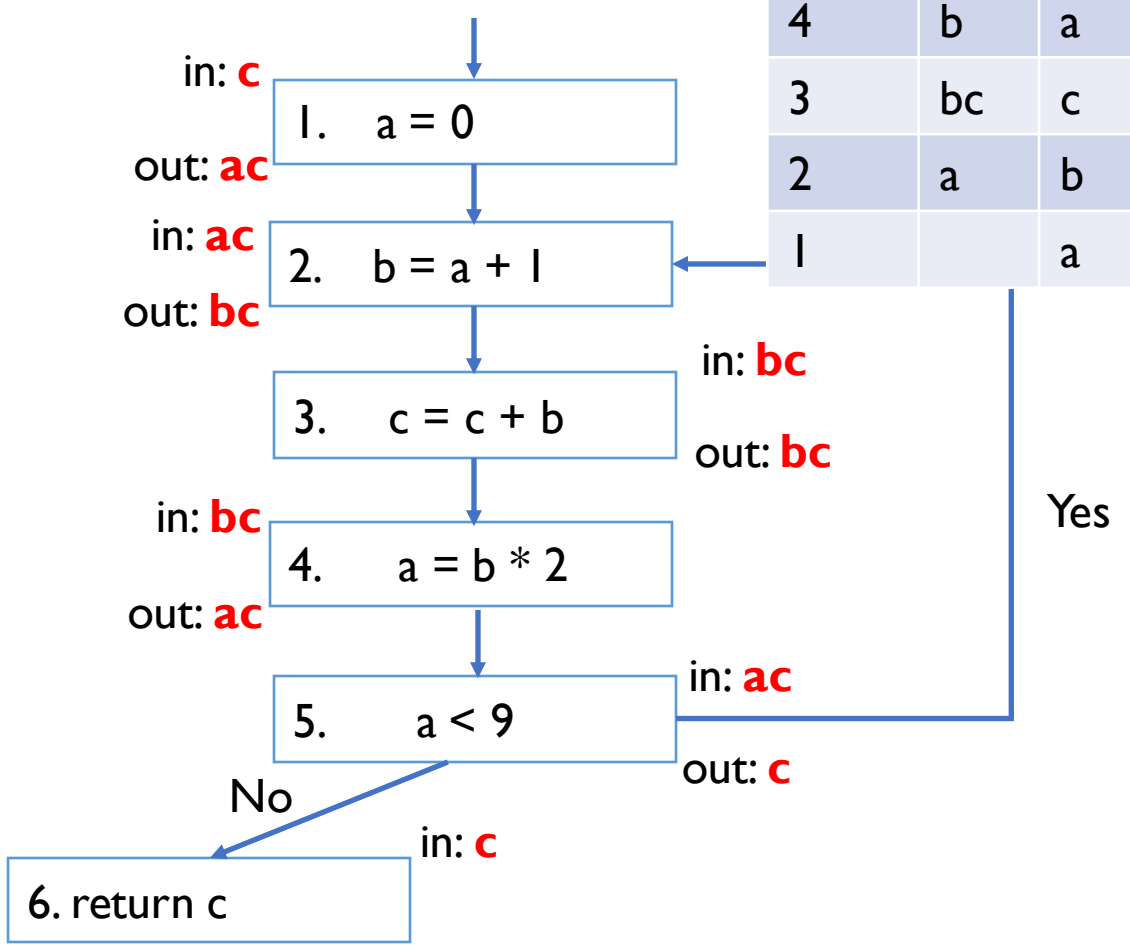
Liveness Example: Round 1

Algorithm

for each node n in CFG
  in[n] = ∅; out[n] = ∅ } Initialize solutions

repeat
  for each node n in CFG **in reverse topsort order**
    in'[n] = in[n]
    out'[n] = out[n] } Save current results
    out[n] = $\bigcup_{s \in succ[n]}$ **in[s]**
    in[n] = use[n] ∪ (out[n] – def[n]) } Solve data-flow equations
until in'[n]=in[n] and out'[n]=out[n] for all n } Test for convergence

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



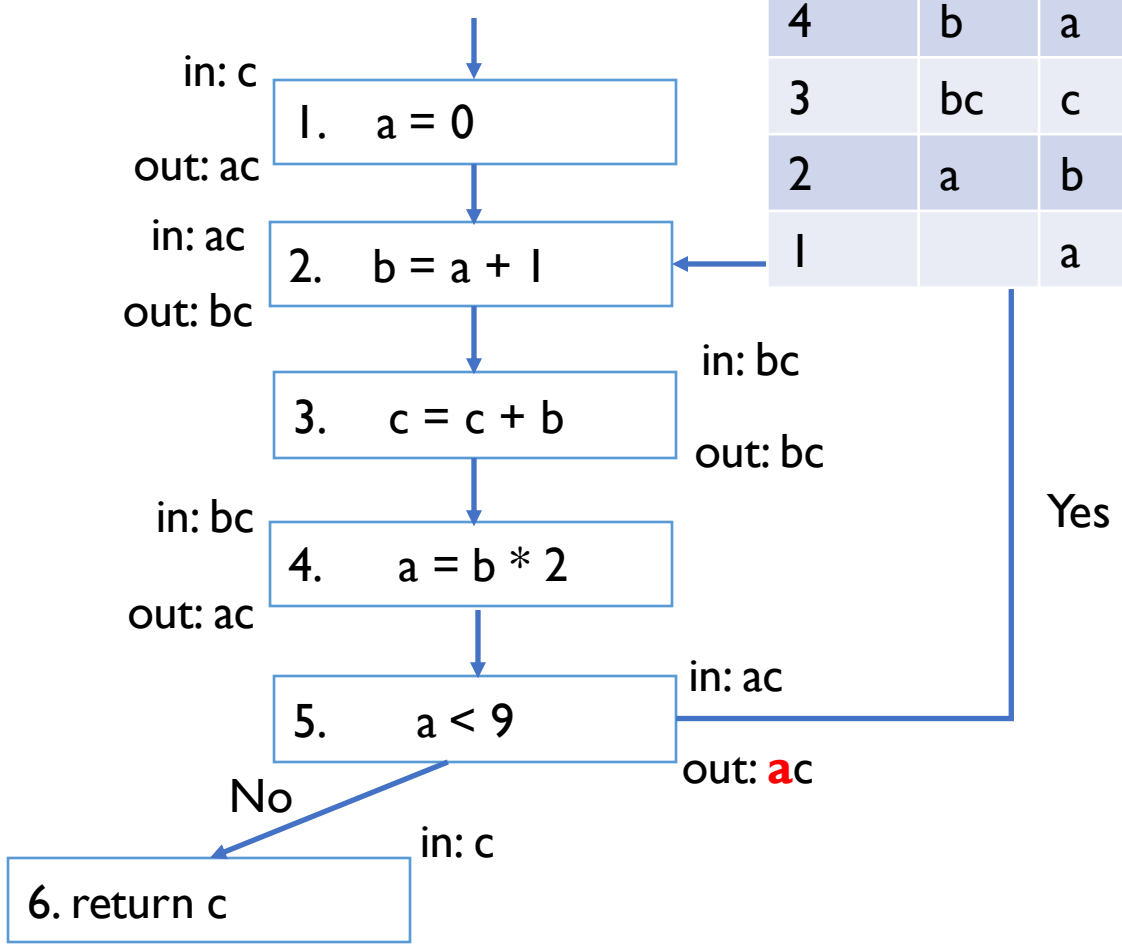
Liveness Example: Round I

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] = ∅ } Initialize solutions

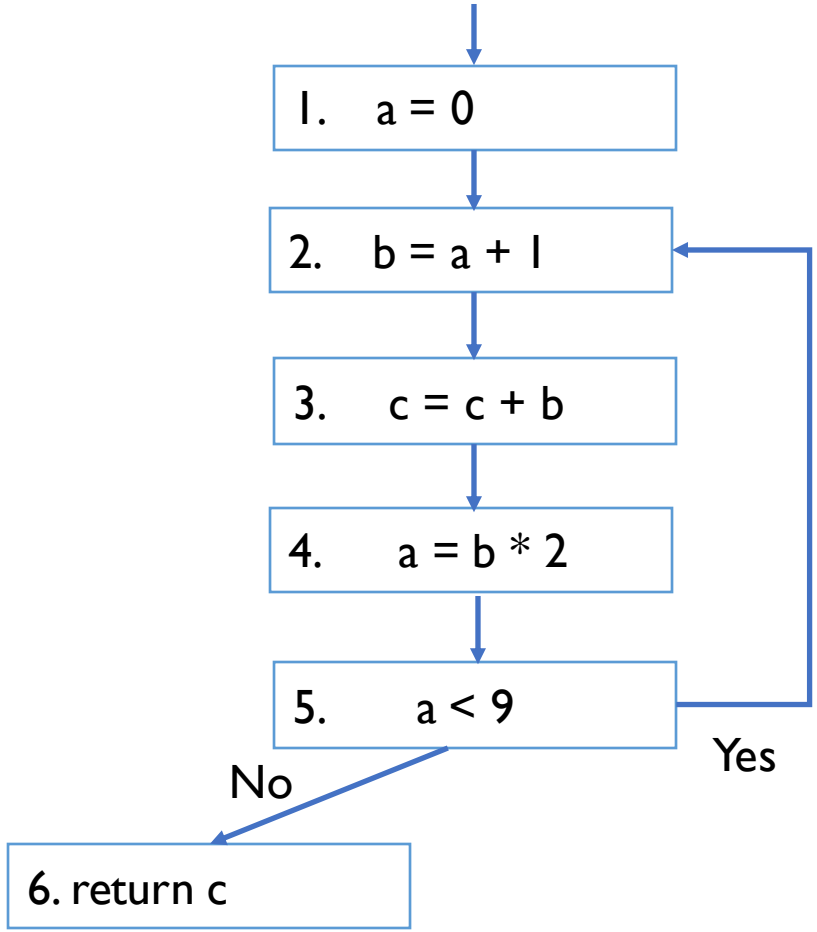
repeat
  for each node n in CFG **in reverse topsort order**
    in'[n] = in[n]
    out'[n] = out[n] } Save current results
    out[n] = $\bigcup_{s \in succ[n]}$ **in[s]**
    in[n] = use[n] ∪ (out[n] – def[n]) } Solve data-flow equations
until in'[n]=in[n] and out'[n]=out[n] for all n } Test for convergence



Conservative Approximation

node #	use	def	X in	X out	Y in	Y out	Z in	Z 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	

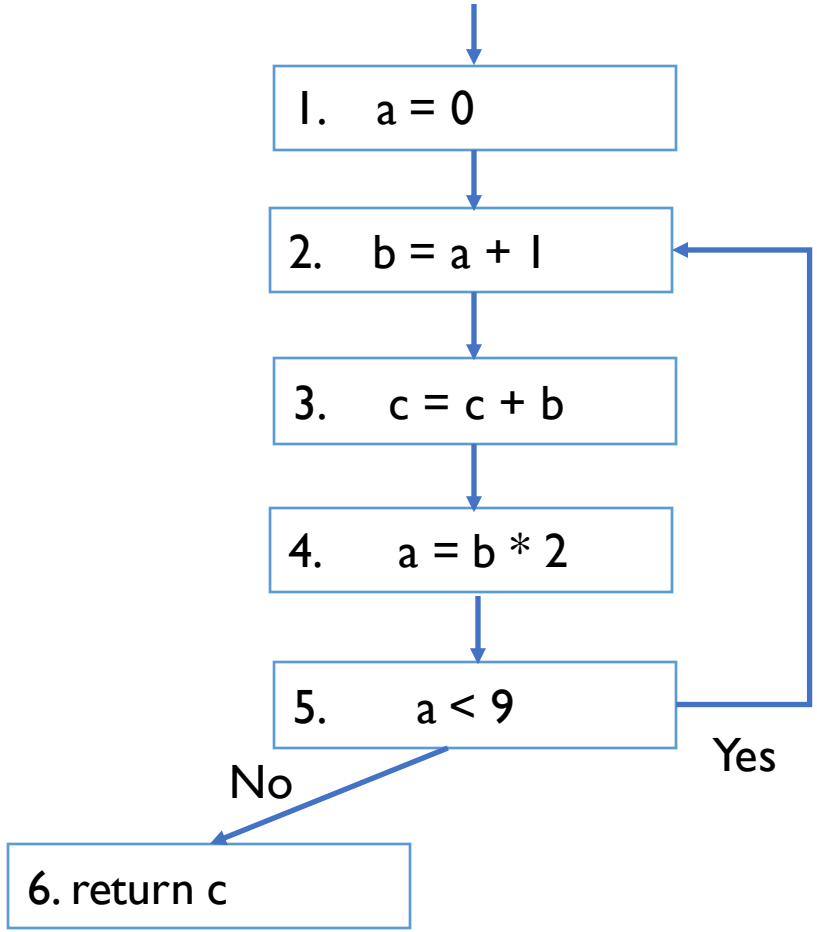
Solution X:
- From the previous slide



Conservative Approximation

			X		Y		Z	
node #	use	def	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?

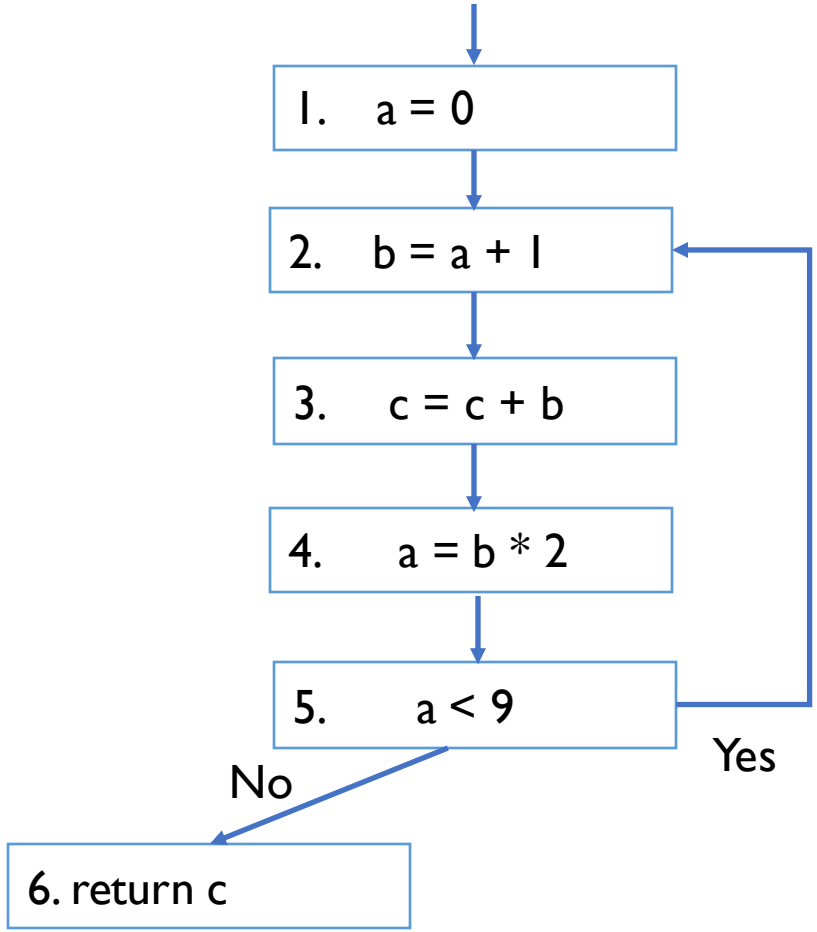


Imprecise conservative solutions ⇒ sub-optimal but correct programs

Conservative Approximation

			X		Y		Z	
node #	use	def	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	

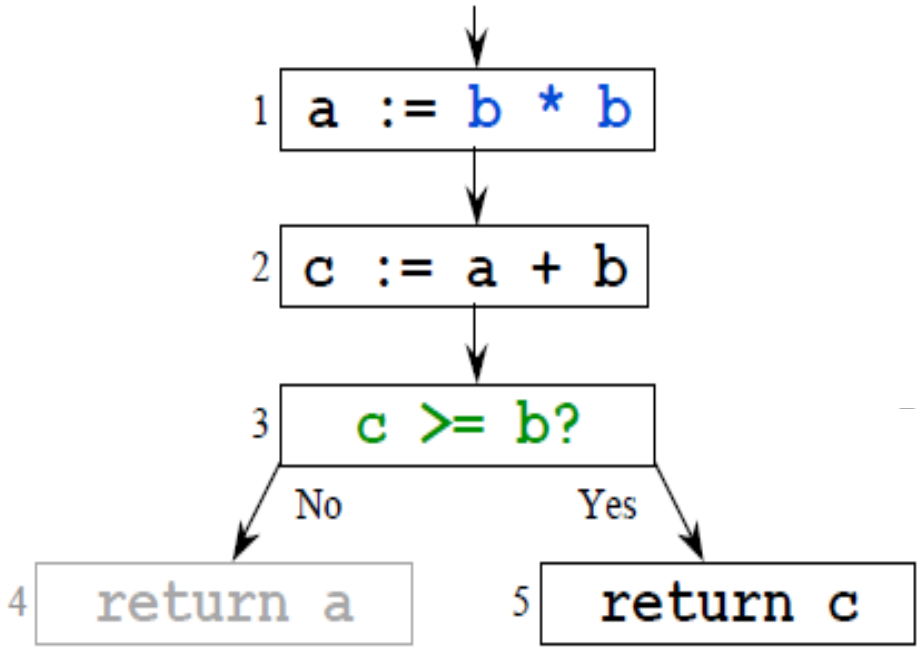
Solution Z:
Does not identify c as live in all cases
– Does Z lead to a correct program?



Non-conservative solutions ⇒ incorrect programs

Need for approximation

- Static vs. Dynamic Liveness: **b*b** is always non-negative, so **c >= b** is always true and **a**'s value will never be used after node

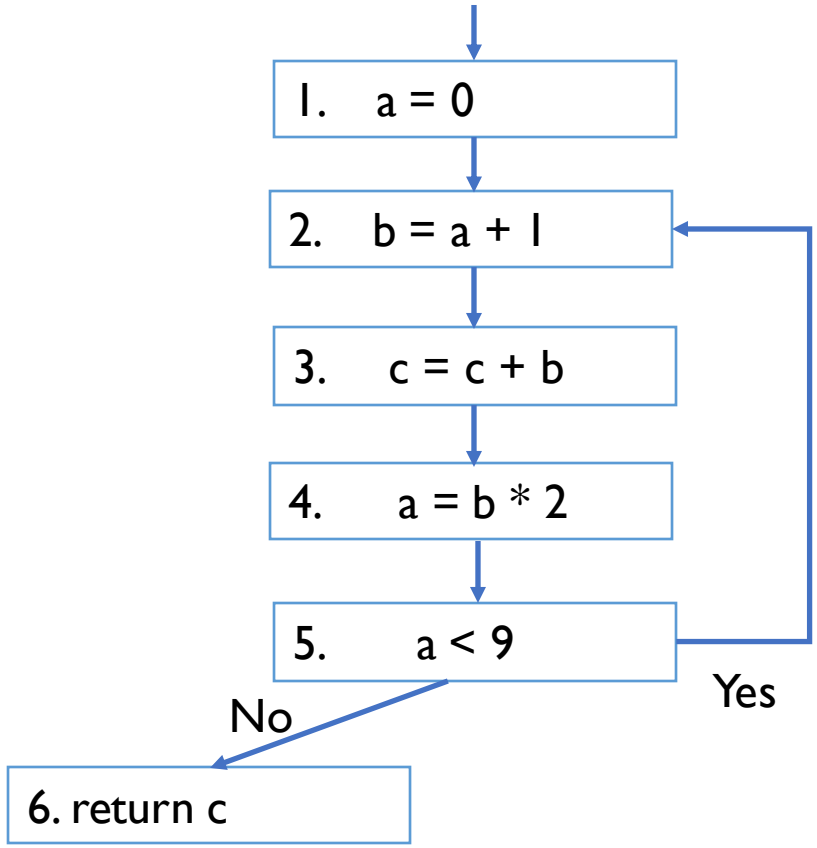


No compiler can statically identify all infeasible paths

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

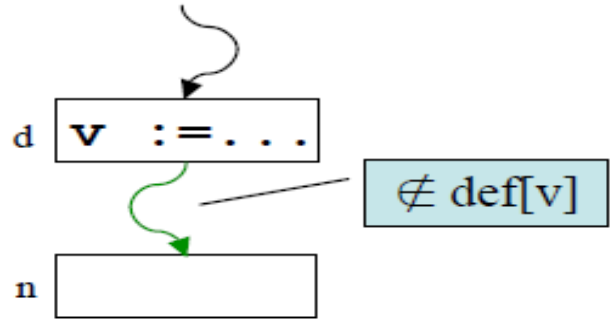
You need **2** registers Why?



Example 2: Reaching Definition

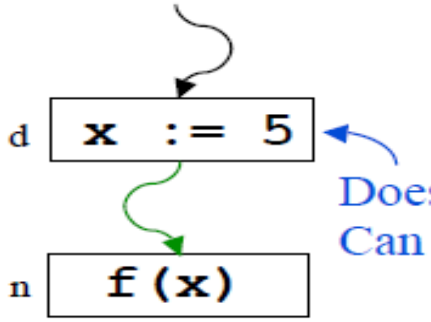
Definition

- 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

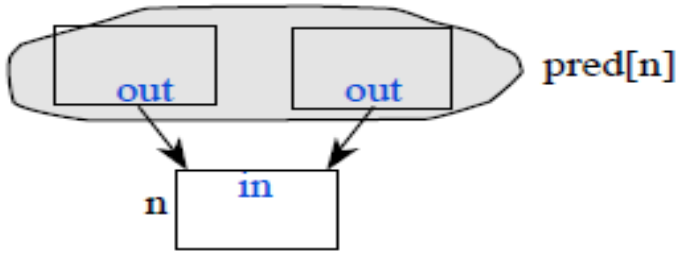
- **Gen[n]:** Definitions that are generated by node n (at most one)
- **Kill[n]:** Definitions that are killed by node n

statement	gen's	kills
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,…,y_n)	{f,y_1,…,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

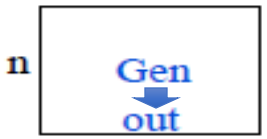


The out set

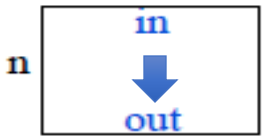
– A definition reaches the end of a node if (1) the node itself **generates** the definition **or** if (2) the definition reaches the beginning of the node and the node does **not kill** it

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

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



(1)



(2)

Recall Liveness Analysis

- Data-flow Equation for liveness

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

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

- **Liver** **s of Gen and Kill**

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

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

A use of a variable generates liveness
A def of a variable kills liveness

Gen: New information that's added at a node
Kill: Old information that's removed at a node

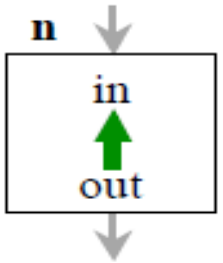
Can define almost any data-flow analysis in terms of Gen and Kill

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[]

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

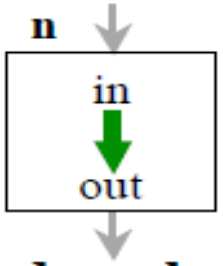


liveness

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[]

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



reaching definitions

Some problems need both forward and backward analysis

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

Data-Flow Equation for reaching definition

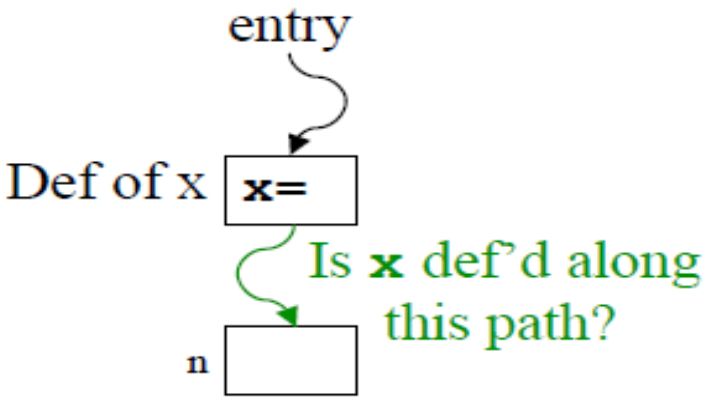
Symmetry between reaching definitions and liveness

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

Reaching Definitions

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

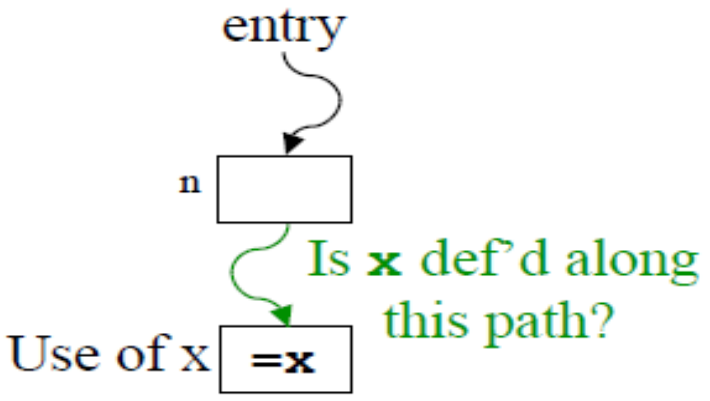
$$out[n] = gen[n] \cup (in[n] - kill[n])$$



Live Variables

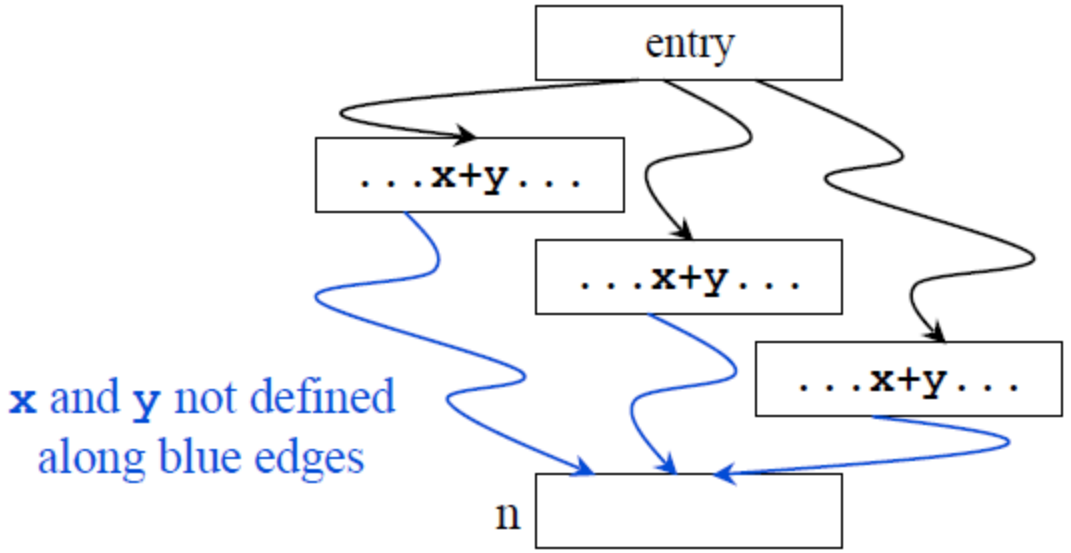
$$out[n] = \bigcup_{s \in succ[n]} in[s]$$

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



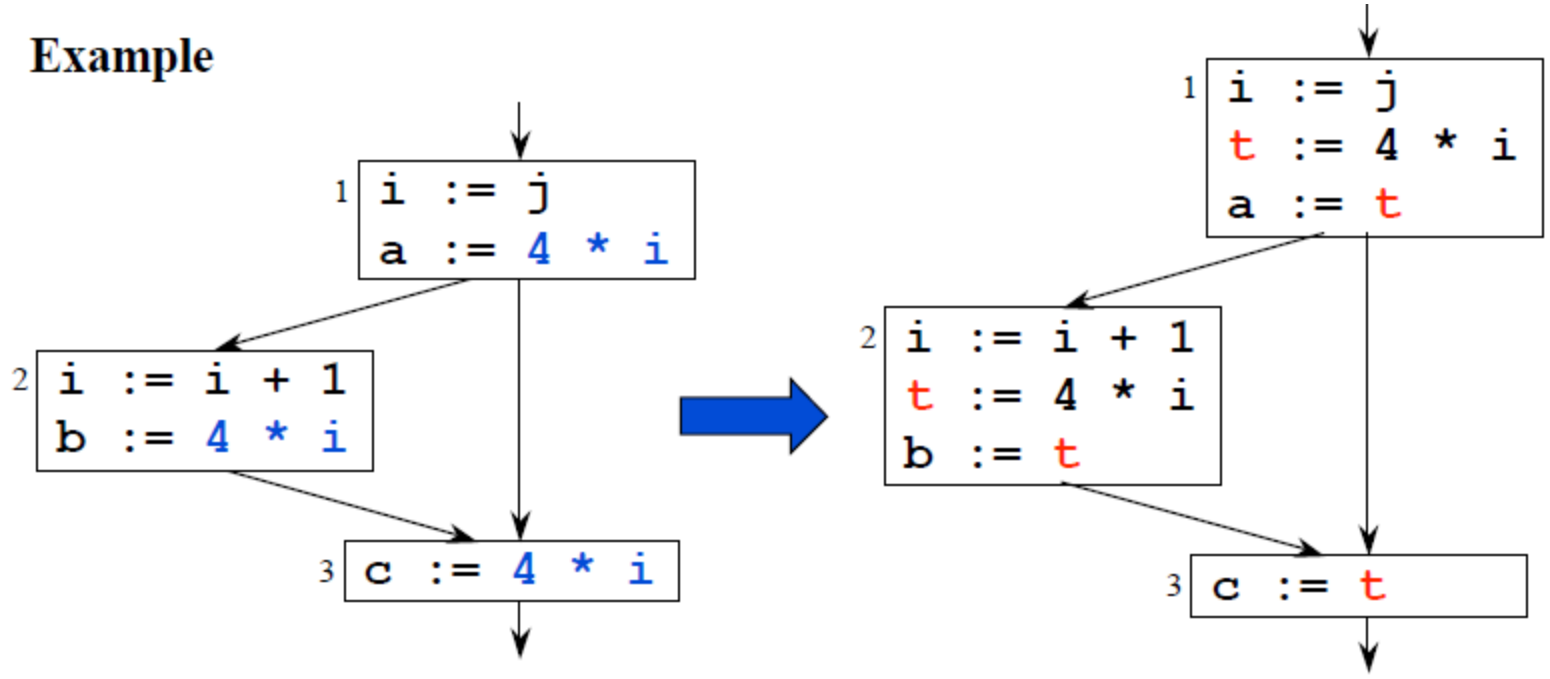
Available Expression

- 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

Thanks

PennState

Thanks to Suman Jana and Baishakhi Ray for some slides.