CSE 543: Computer Security

Module: Security Analysis Techniques

Prof. Syed Rafiul Hussain
Department of Computer Science and Engineering
The Pennsylvania State University

Our Goal

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

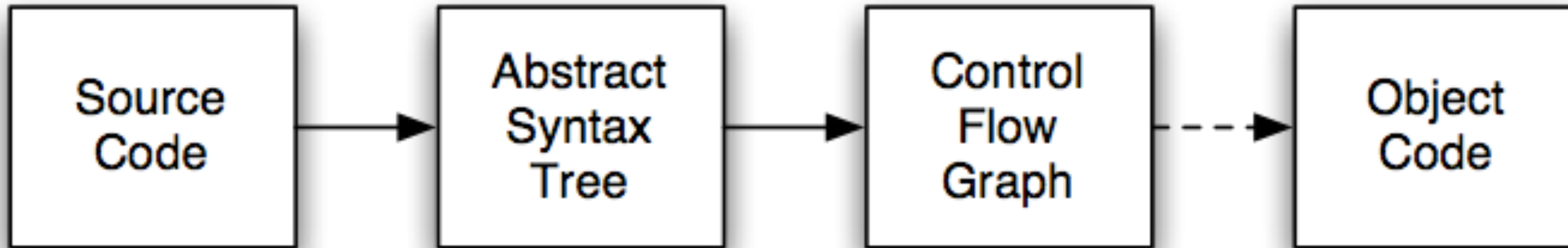
Must perform control and data flow analysis

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

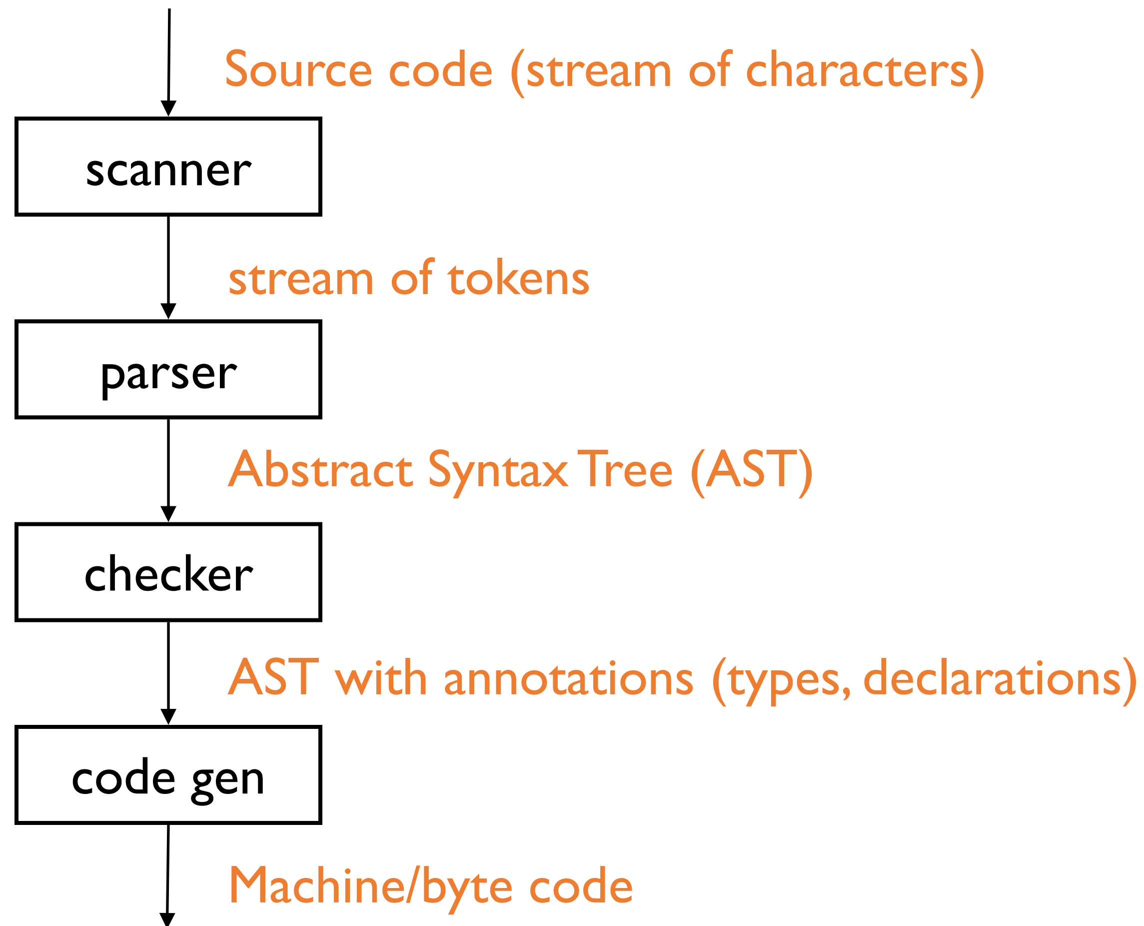
- LLVM (http://llvm.org/) is a highly customizable and modular compiler framework
  - Users can write LLVM passes to perform different types of analysis
  - Clang static analyzer can find several types of bugs
  - Can instrument code for dynamic analysis

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

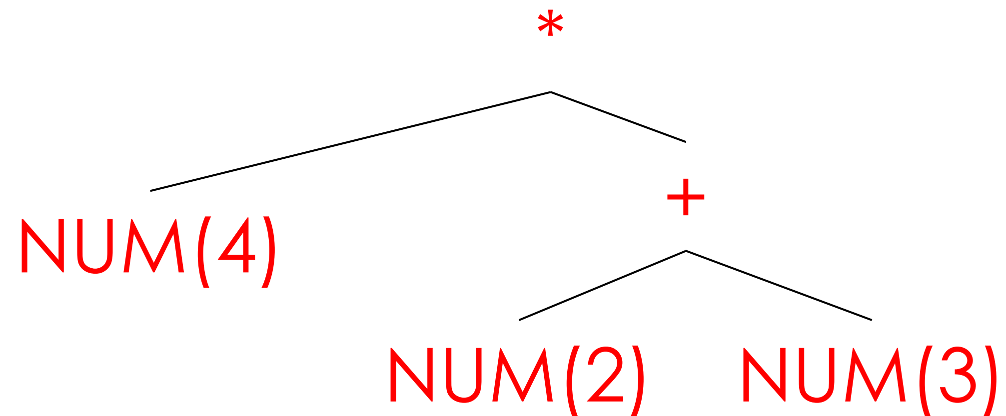
- Source Code

  4*(2+3)

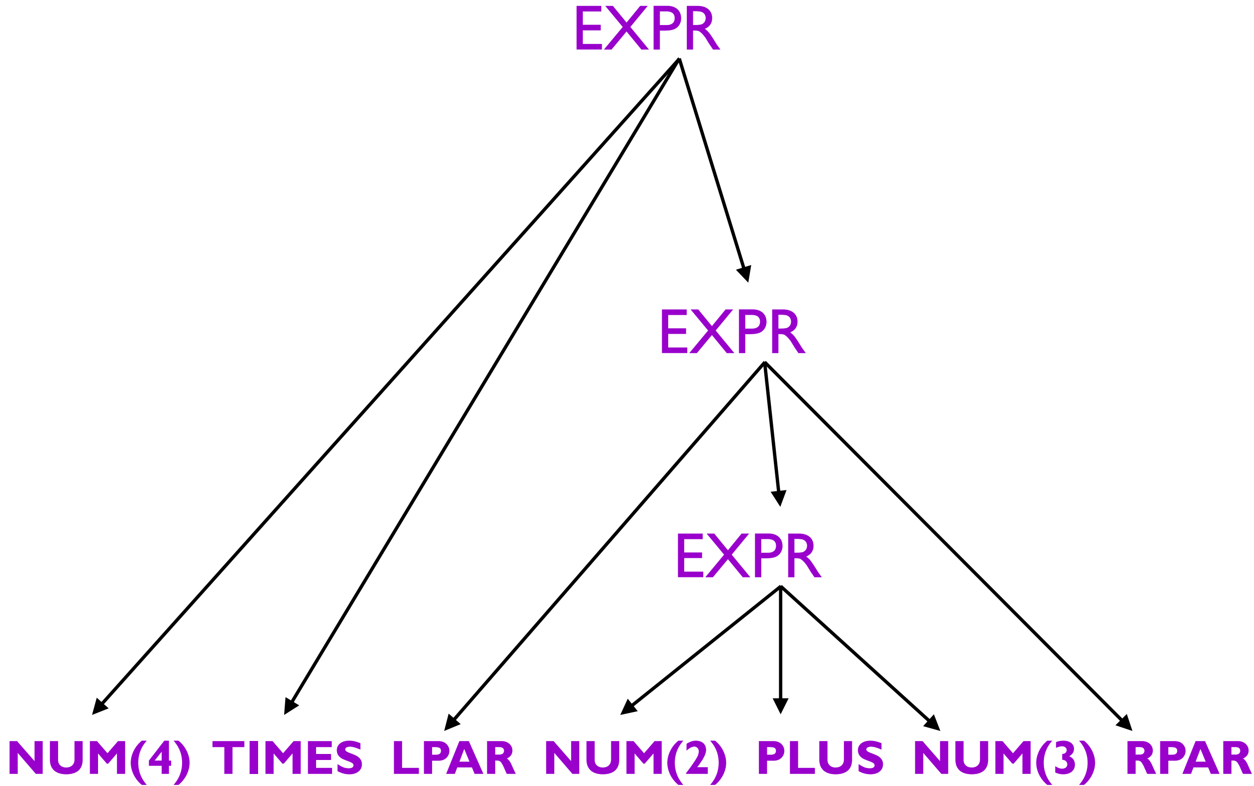
- Parser input

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

- Parser output (AST):



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



leaves are tokens

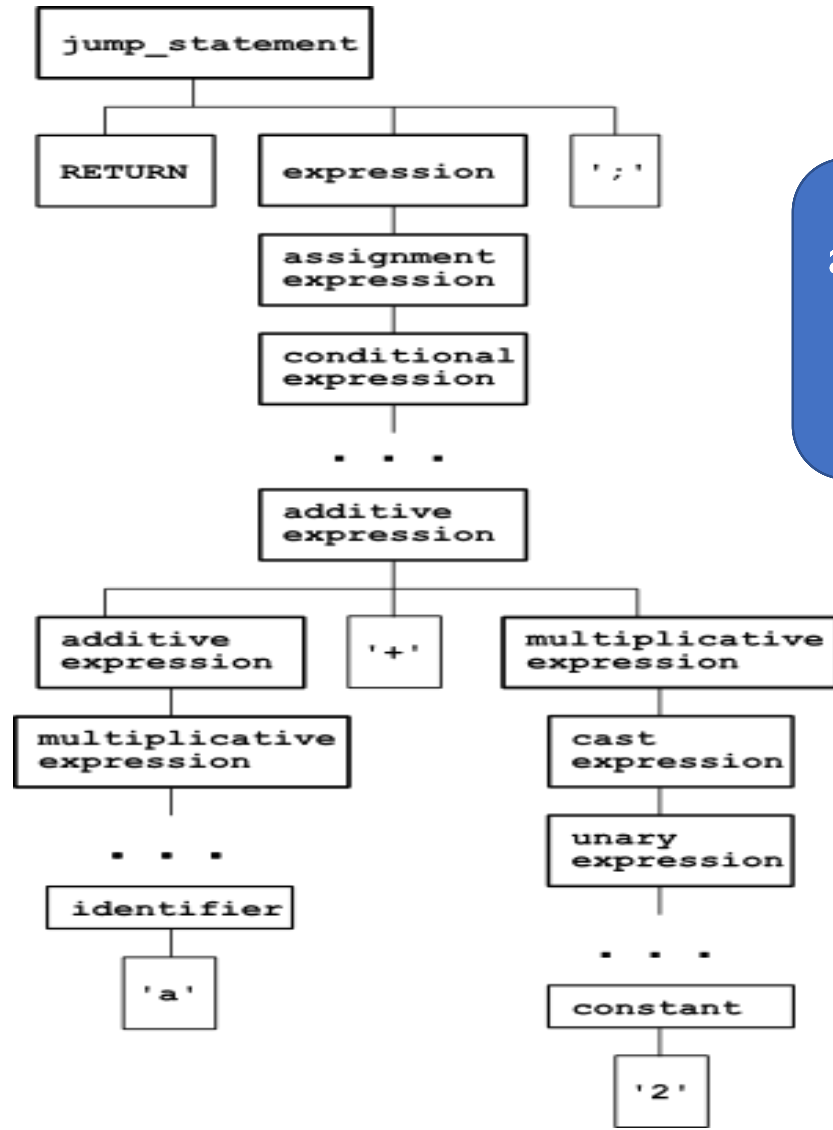
Parse Tree

- 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)

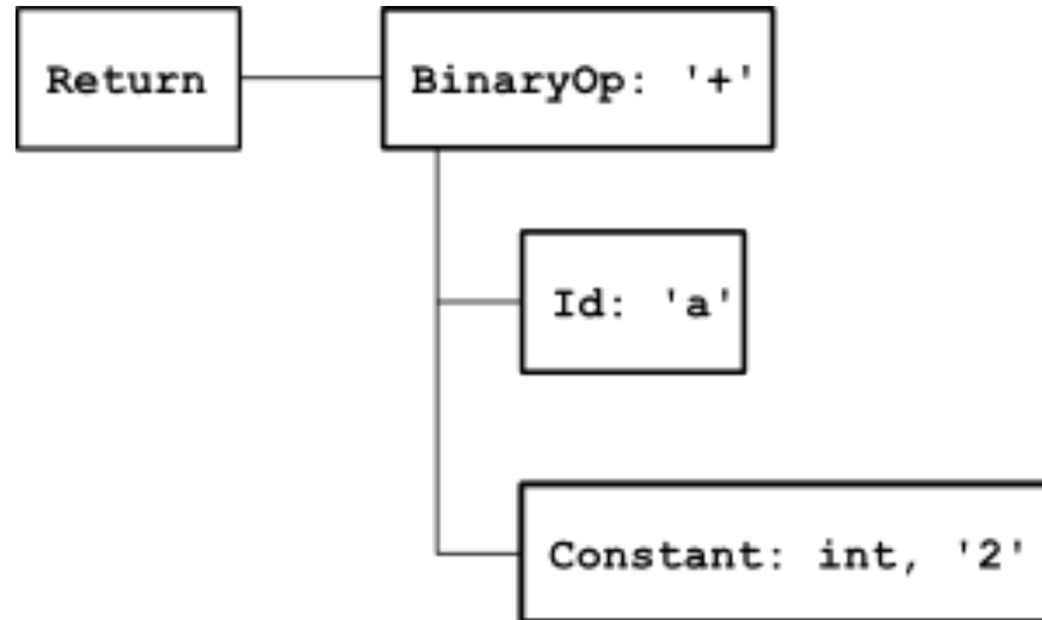
- Simplified syntactic representations of the source code, and they're most often expressed by the data structures of the language used for implementation

> 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:

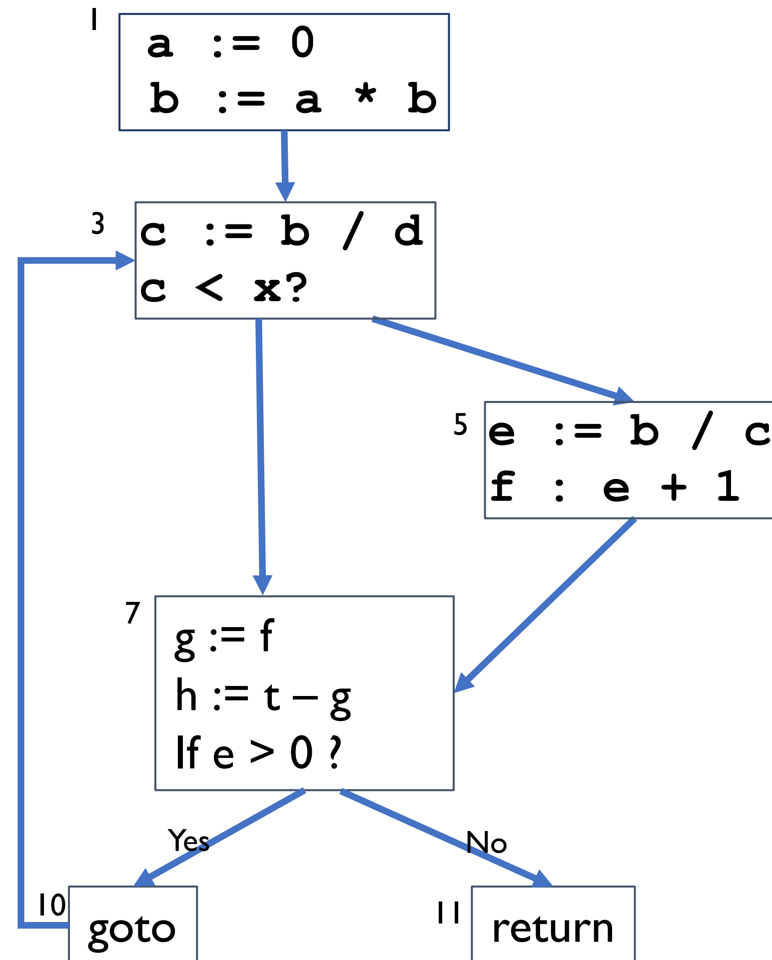
–Use a **Control-flow graph (CFG)**

–Nodes represent statements (low-level linear IR)

–Edges represent explicit flow of control

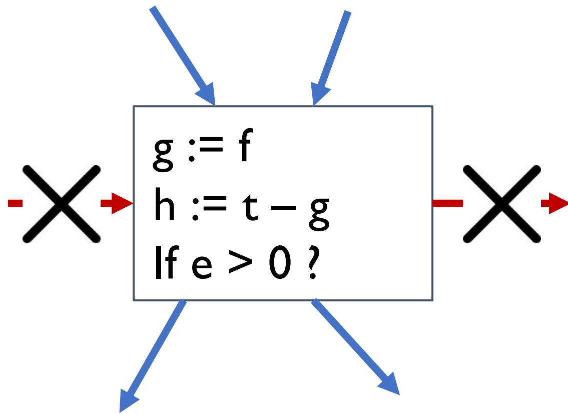
What Is Control-Flow Analysis?

```
1        a := 0
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
```



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}

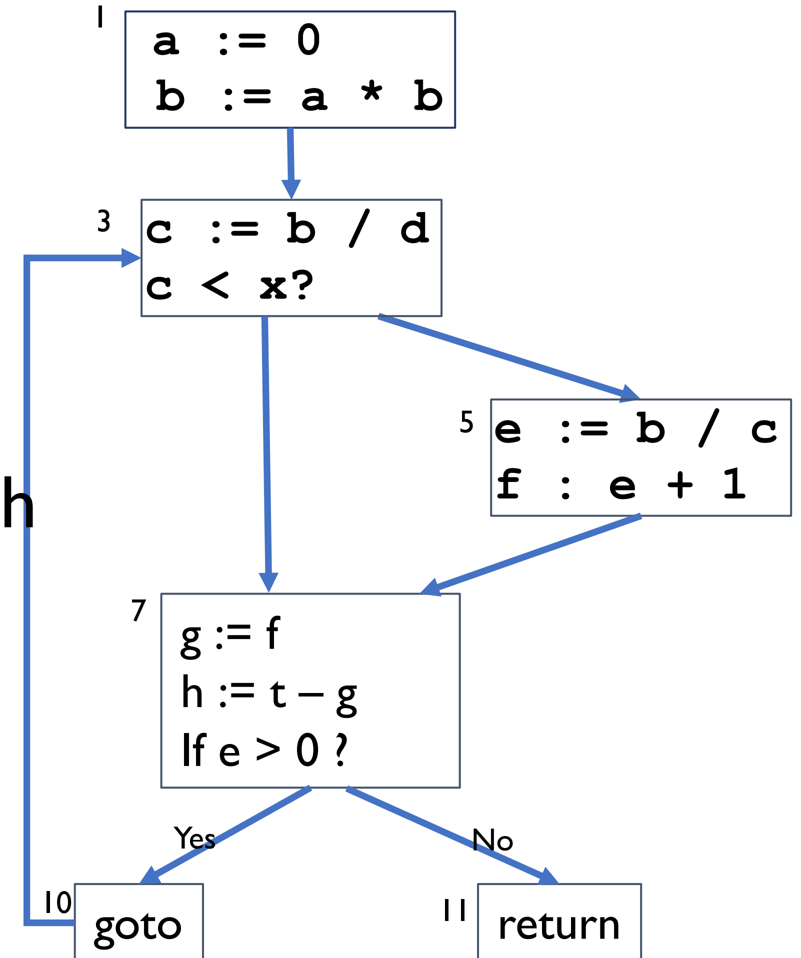
Blocks?

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

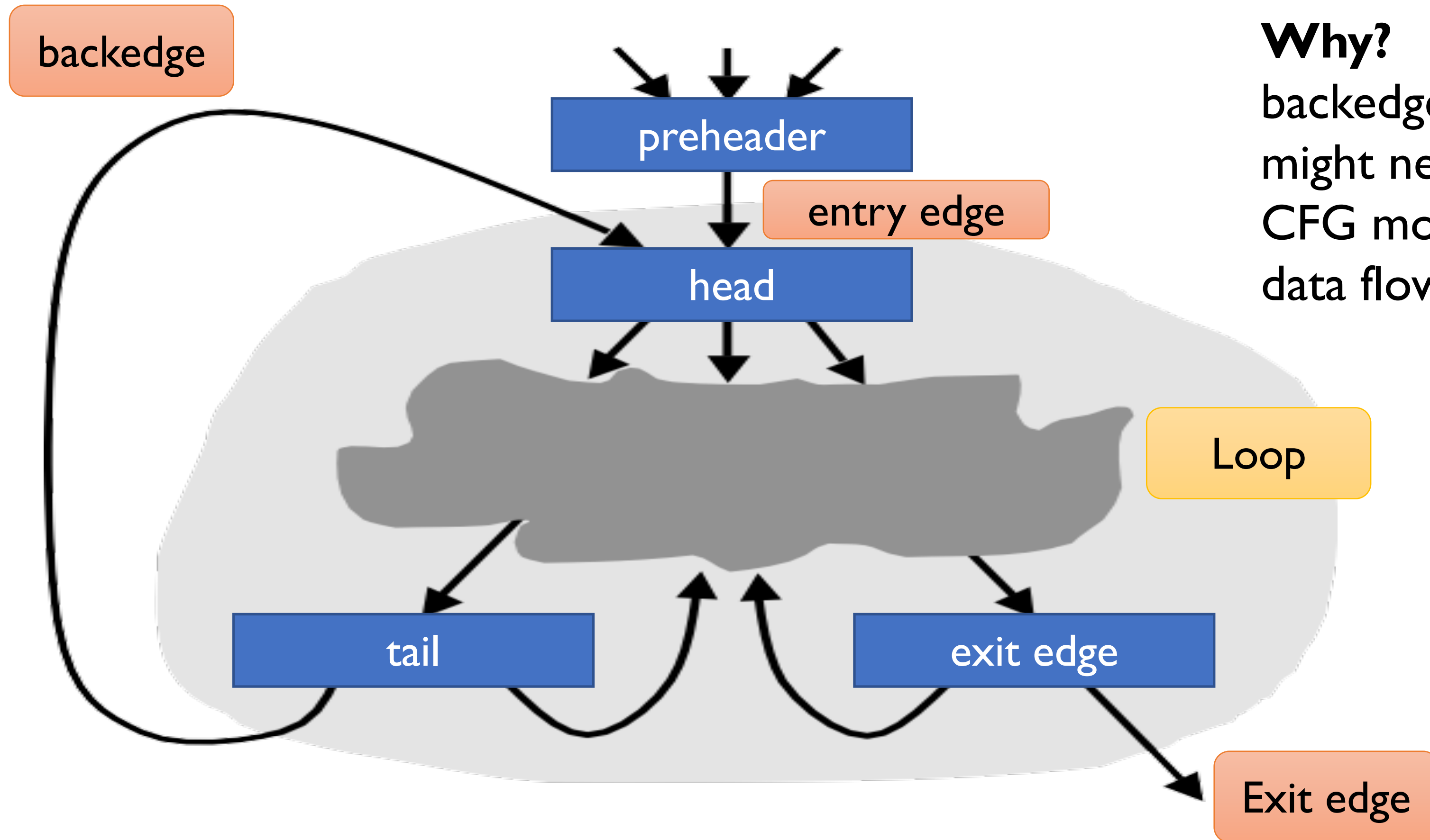
Building a CFG From Basic Block

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)

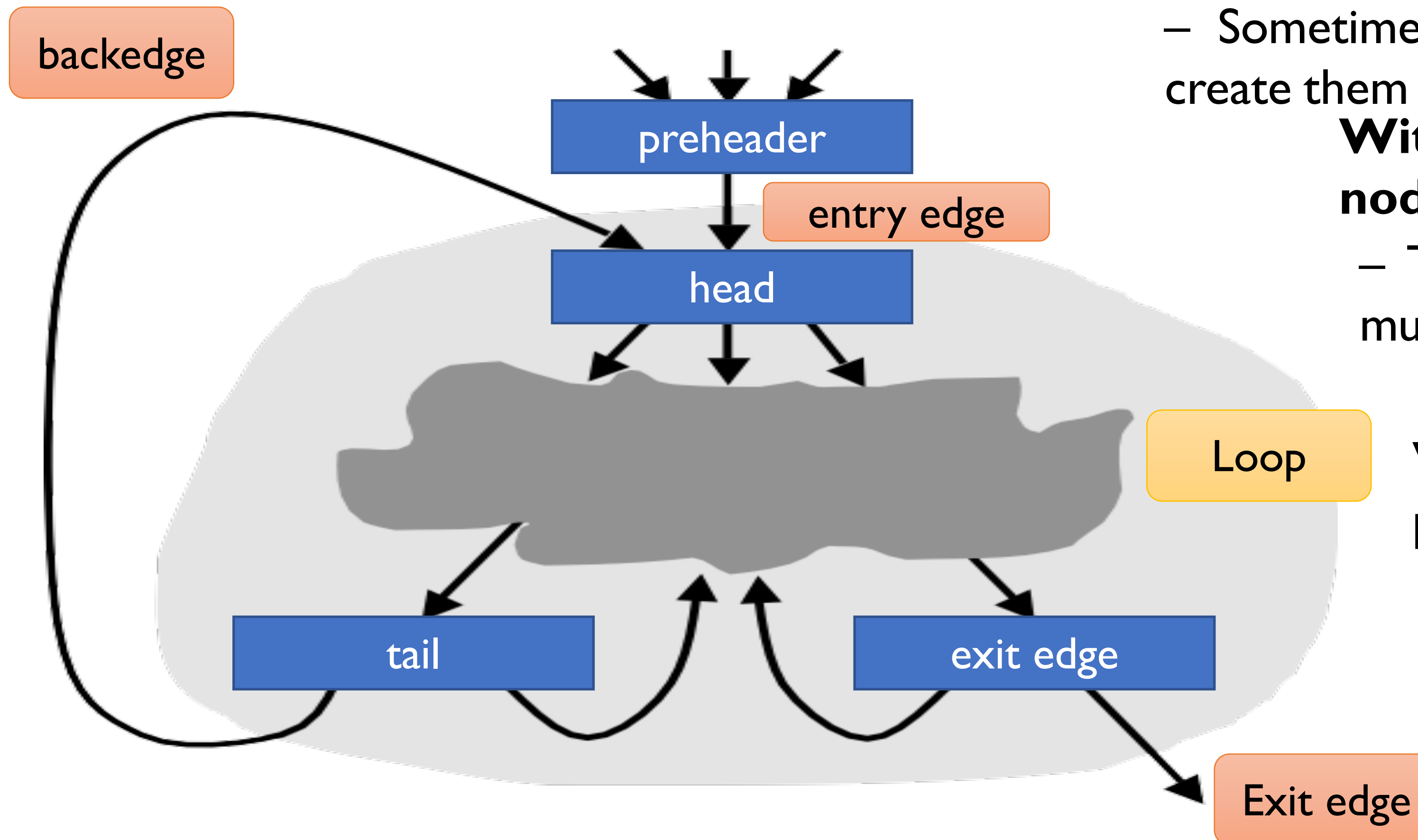


Looping



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

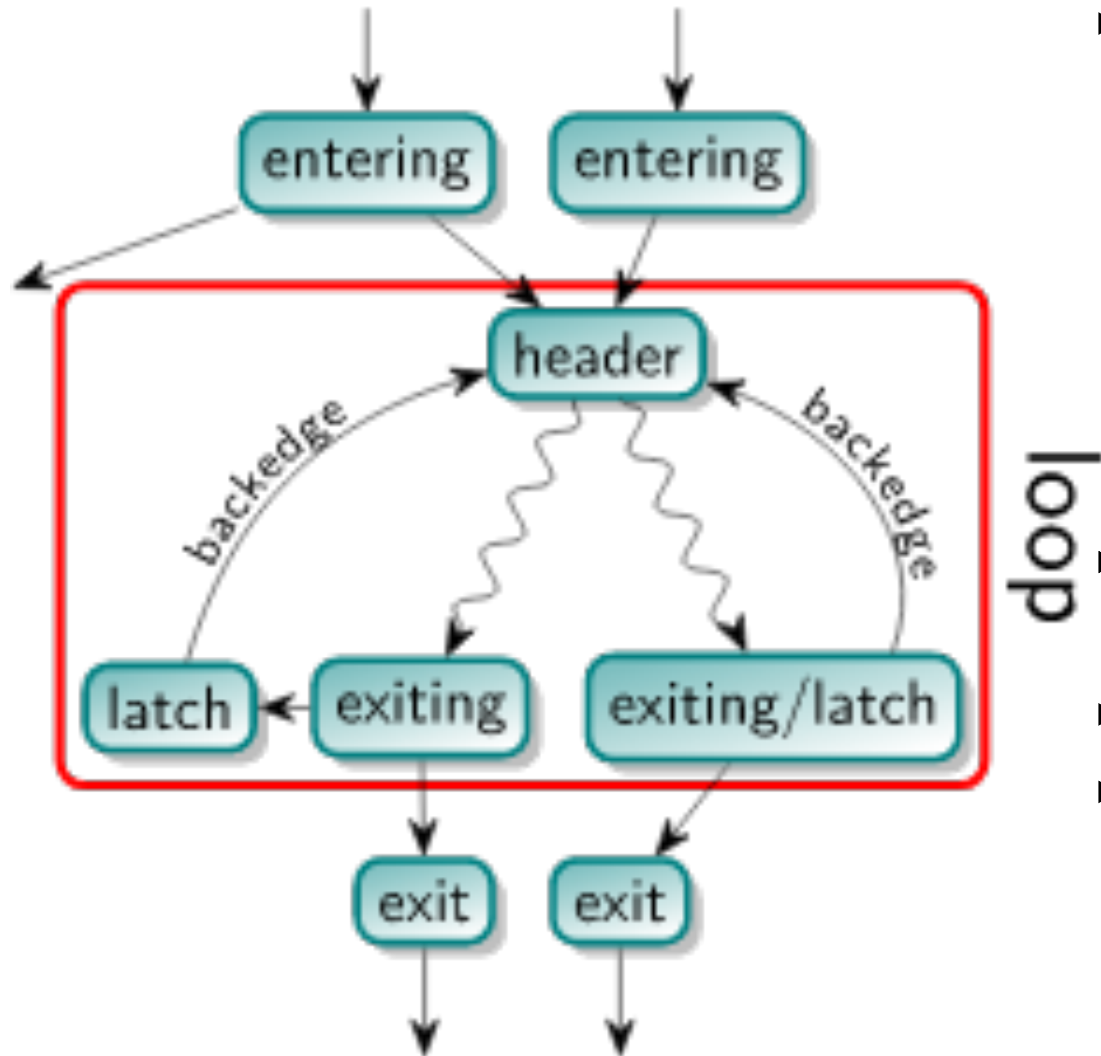
Without preheader node

- 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 ($m \rightarrow n$), 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

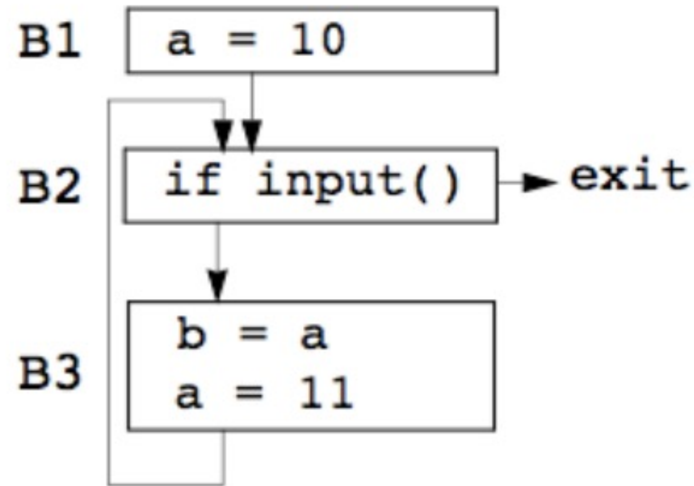
- Derives information about the **dynamic** behavior of a program by only examining the **static** code
- Intraprocedural analysis
- Flow-sensitive: sensitive to the control flow in a function

```
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

- **Examples**
  - Live variable analysis
  - Constant propagation
  - Common subexpression elimination
  - Dead code detection

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

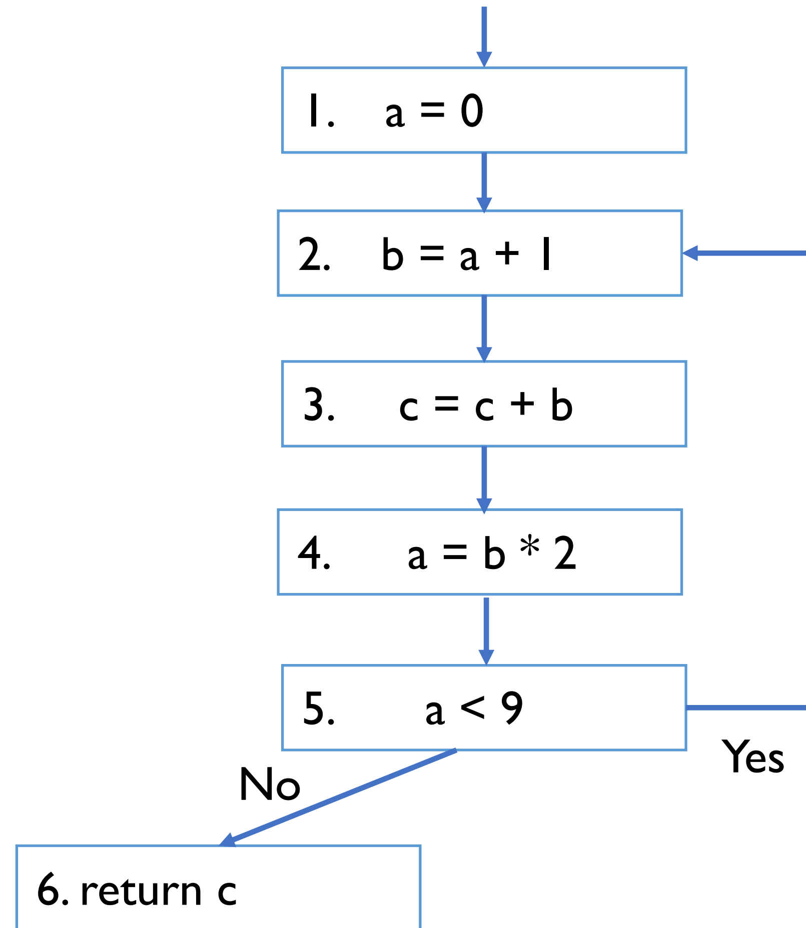
- A program contains an unbounded number of variables
- Must execute on a machine with a bounded number of registers
- Two variables can use the same register if they are never in use at the same time (*i.e*, never simultaneously live).

  –Register allocation uses liveness information

Control Flow Graph

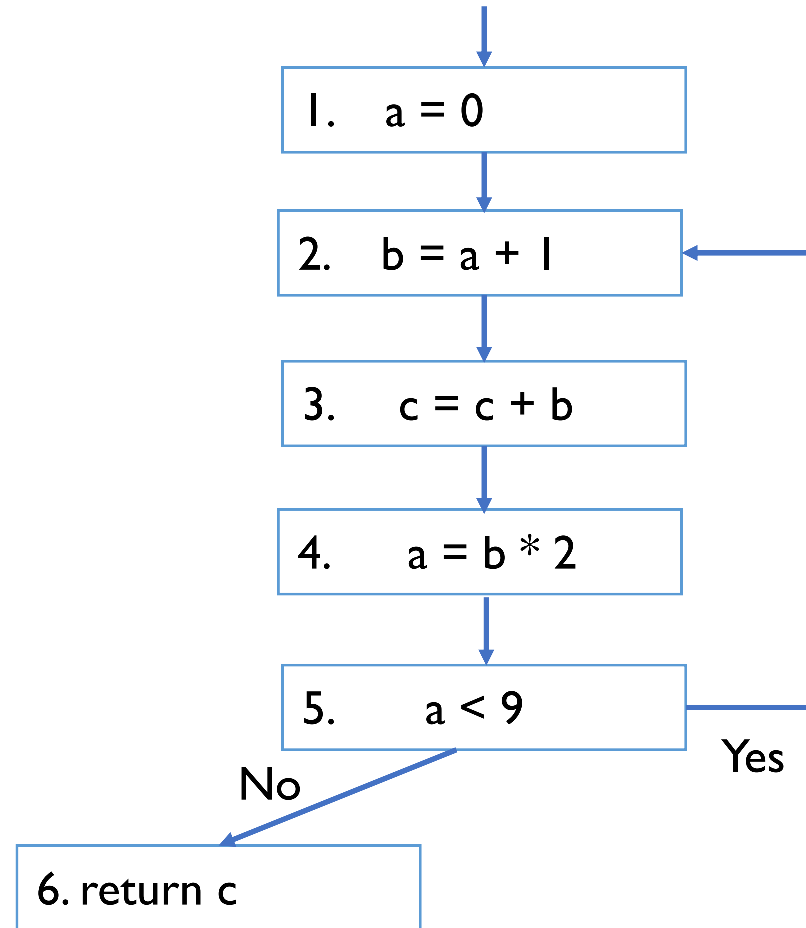
- Let’s consider CFG where nodes contain program statement instead of basic block.
- Example

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



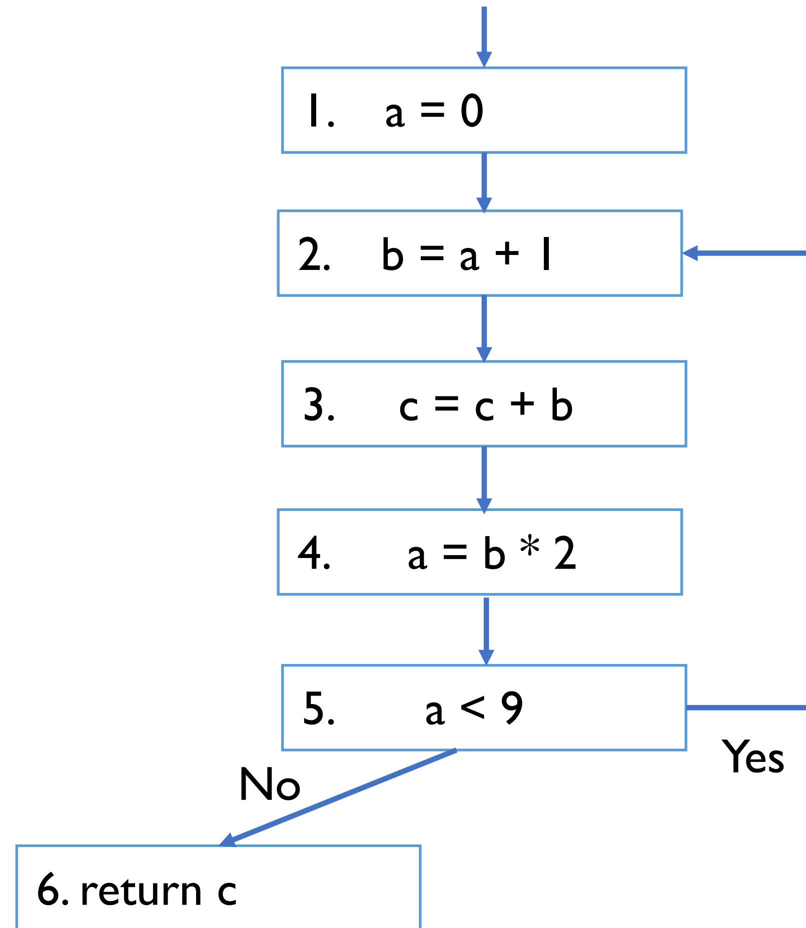
Liveness by Example

- 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)

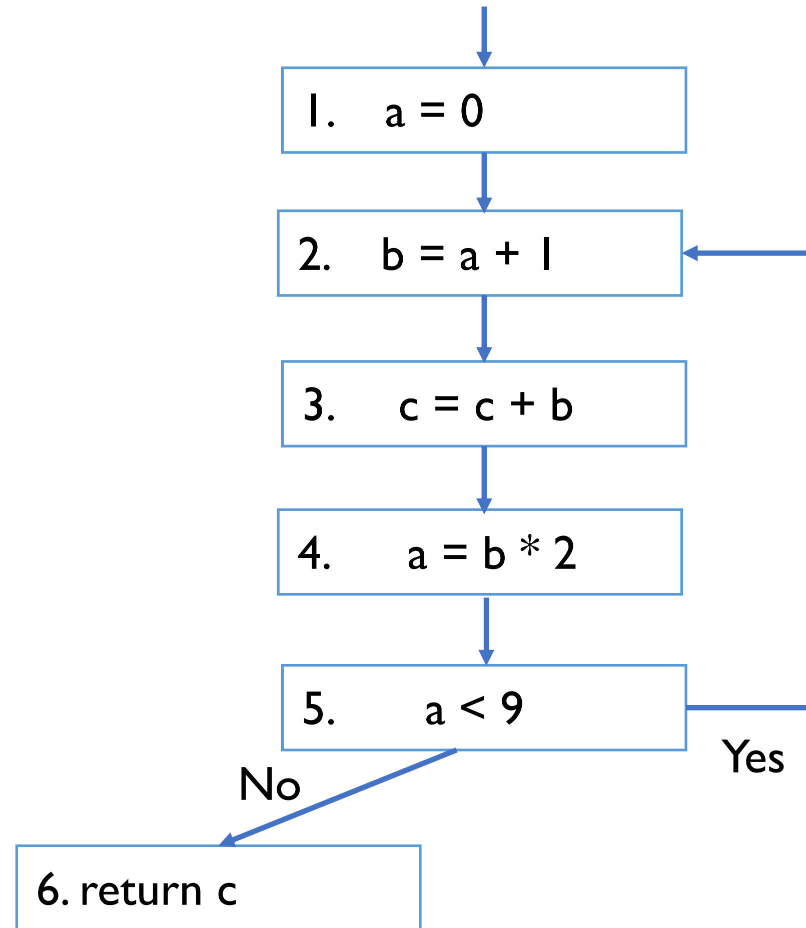


Terminology

- 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
```

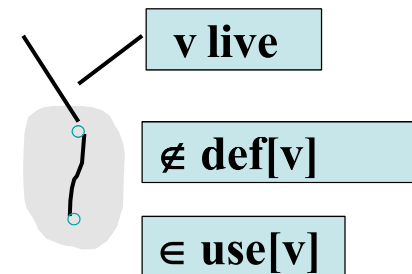
Use

- 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
```

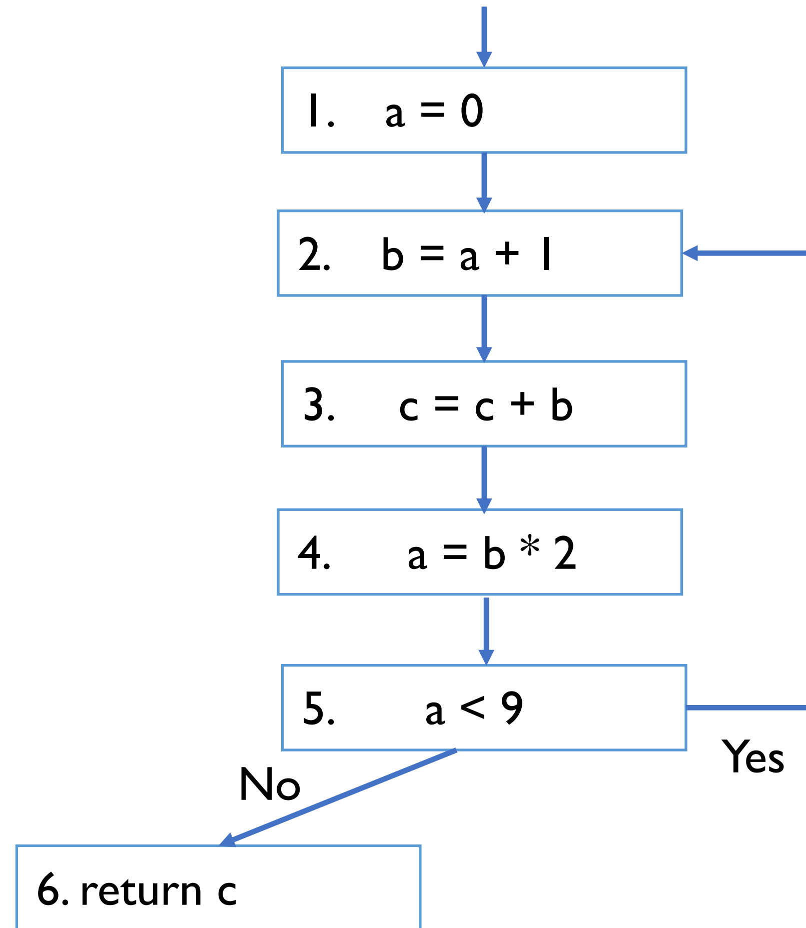
More precise definition of liveness

- 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])

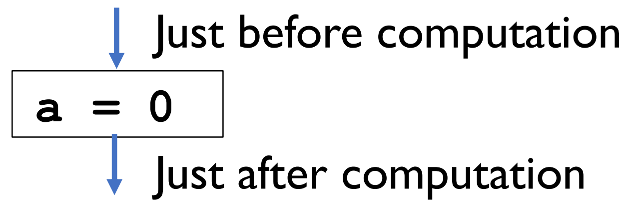


The Flow of Liveness

- 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



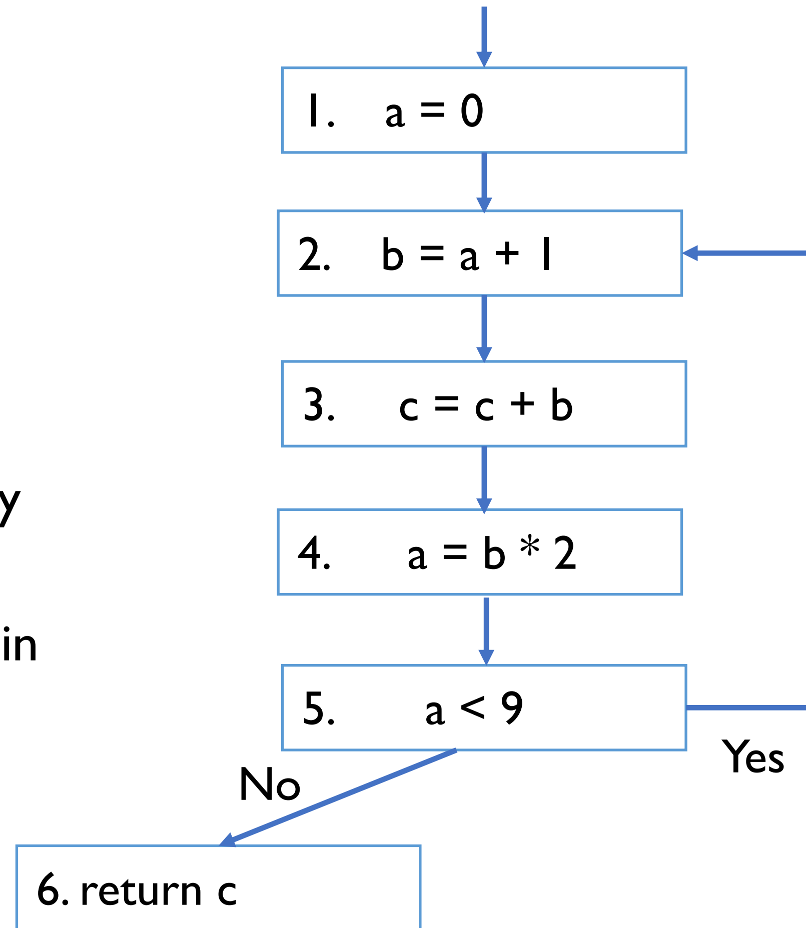
Liveness at Nodes



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

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

Solving Dataflow Equation

for each node n in CFG

$in[n] = \emptyset; out[n] = \emptyset$ } Initialize solutions

repeat

for each node n in CFG

$in'[n] = in[n]$

$out'[n] = out[n]$ } Save current results

$in[n] = use[n] \cup (out[n] - def[n])$ } Solve data-flow equation

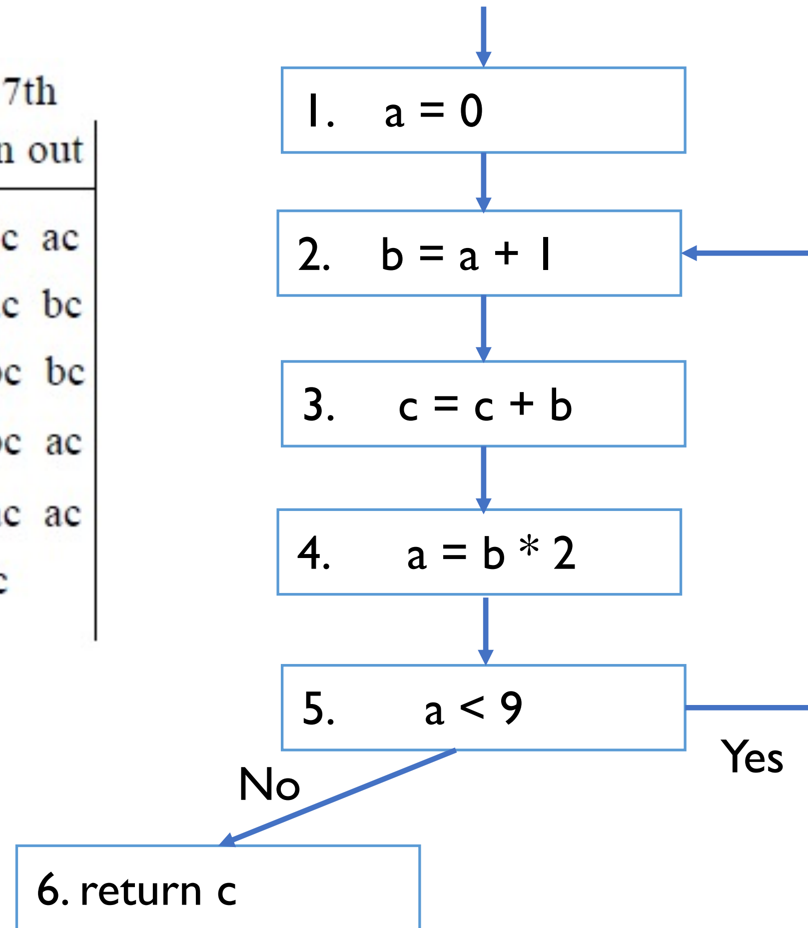
$out[n] = \bigcup_{s \in succ[n]} in[s]$

} Test for convergence

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

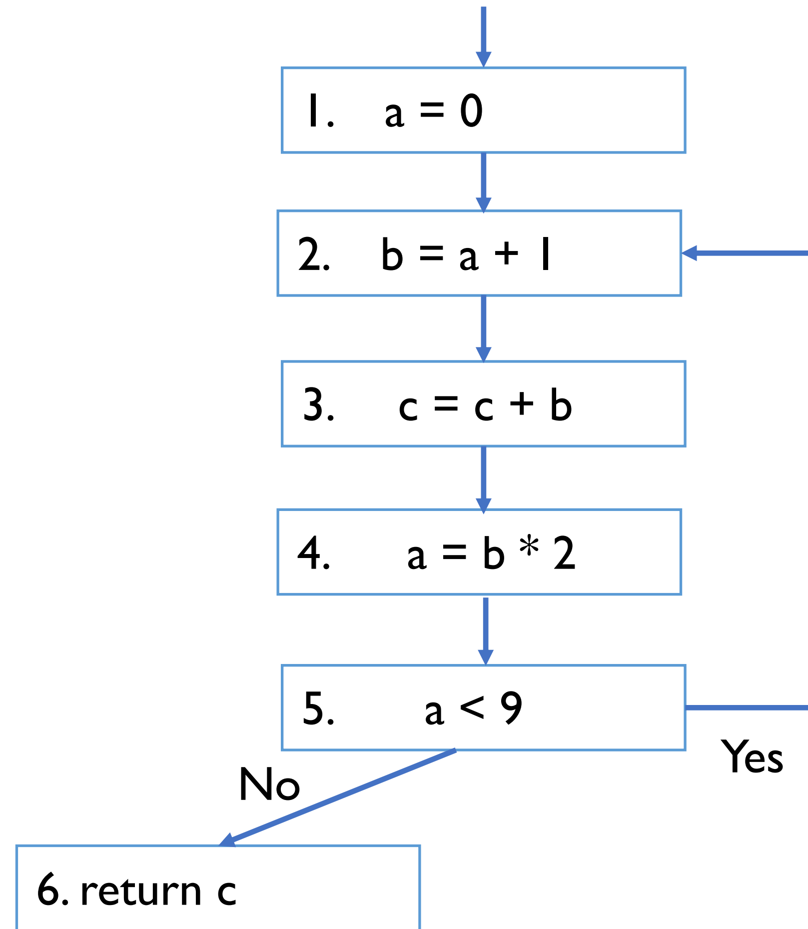
Computing Liveness Example

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

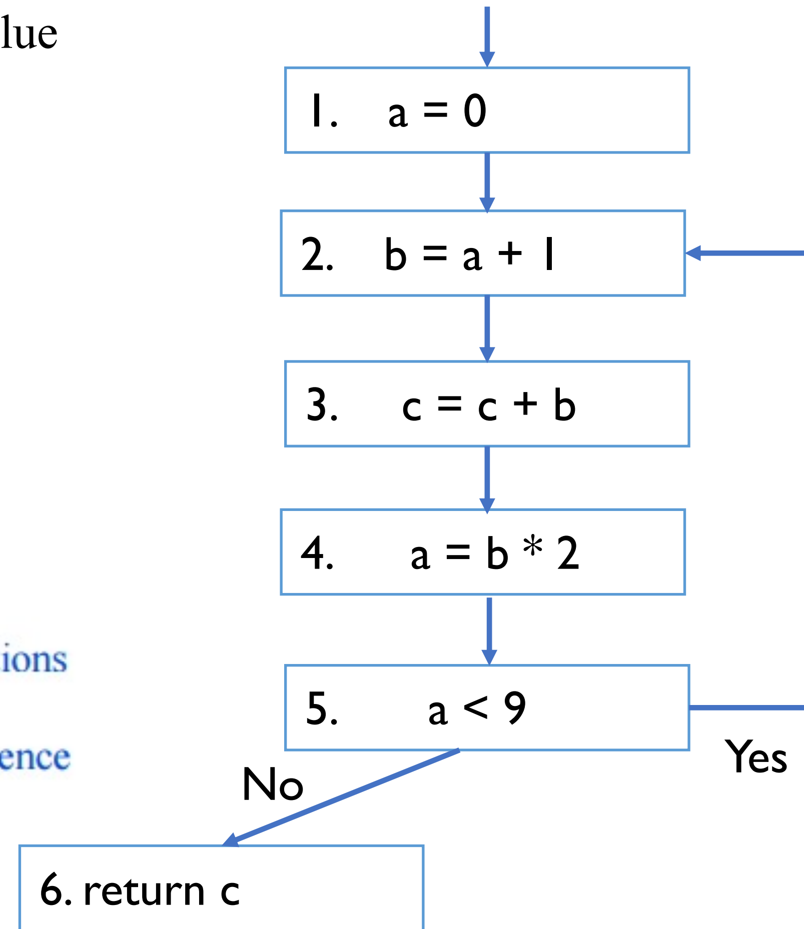


Liveness Example: Round1

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] = ∅ } 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

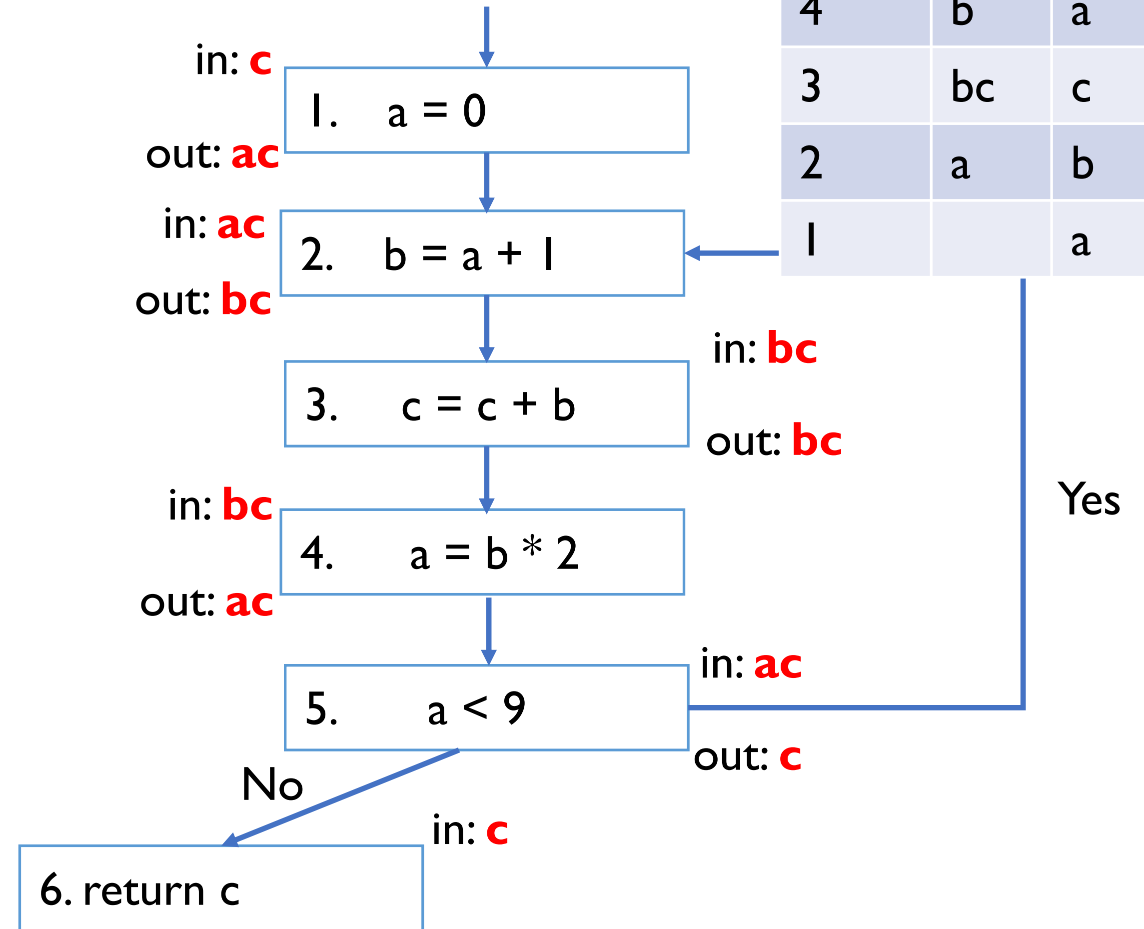
Liveness Example: Round1

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: Round1

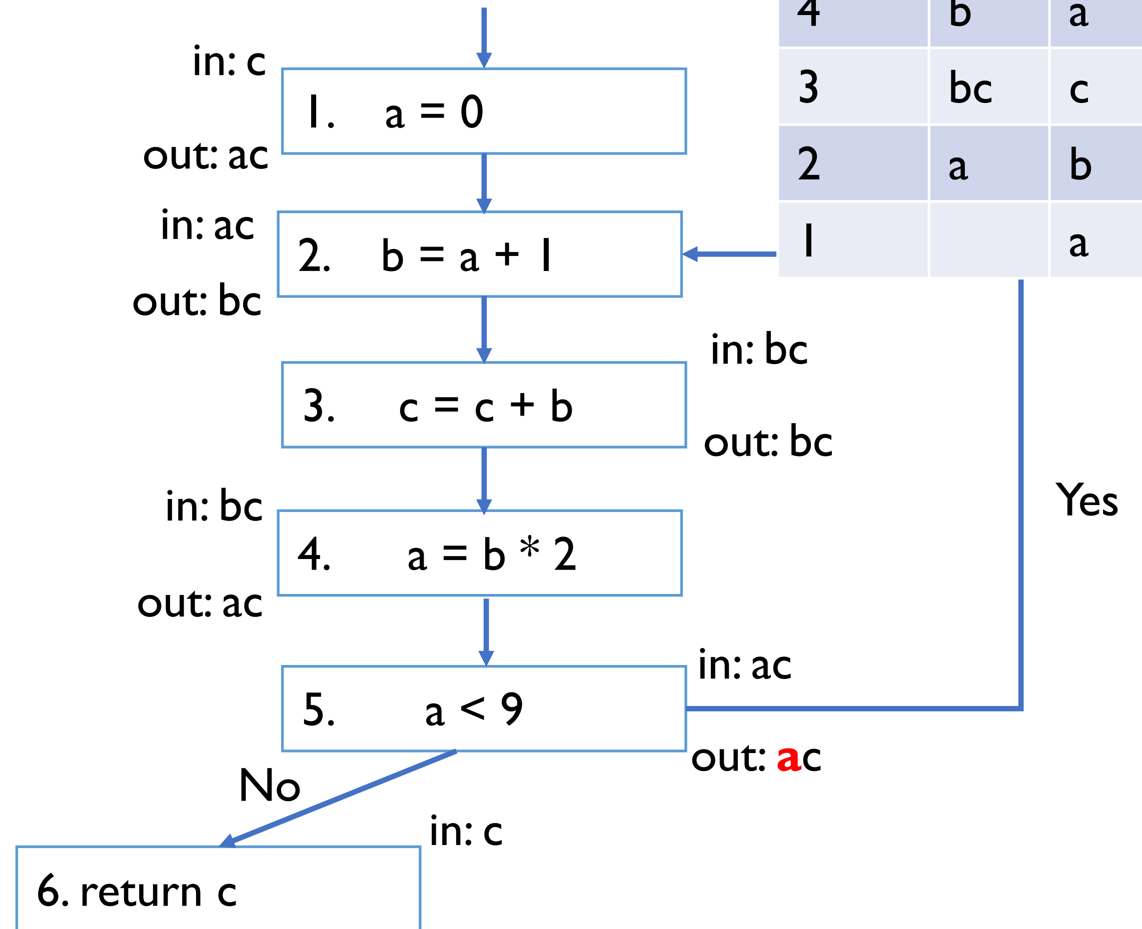
Algorithm

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

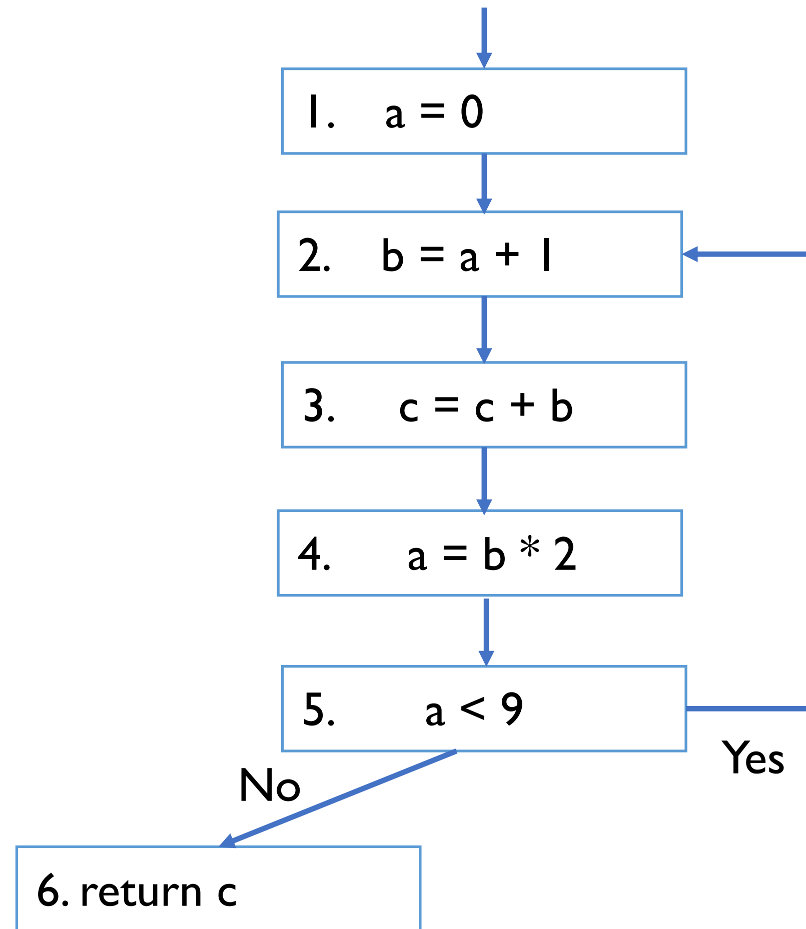
Node	use	def
6	c	
5	a	
4	b	a
3	bc	c
2	a	b
1		a



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	

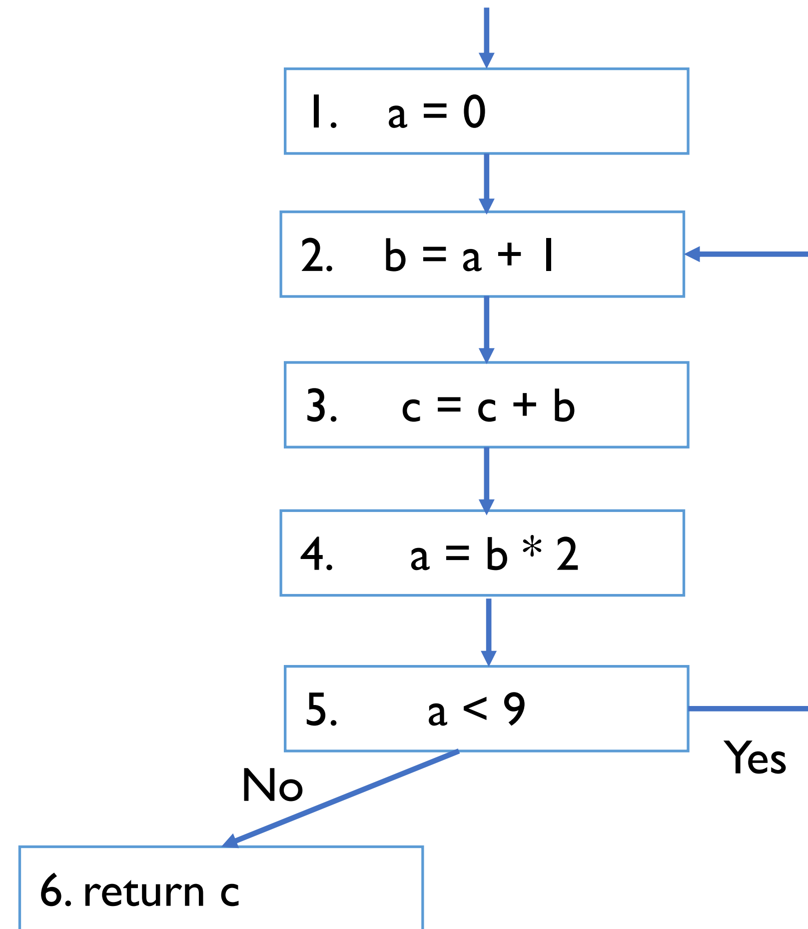
Solution X:
- From the previous slide



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 Y:
Carries variable d uselessly
– Does Y lead to a correct program?



Imprecise conservative solutions ⇒ sub-optimal but correct programs

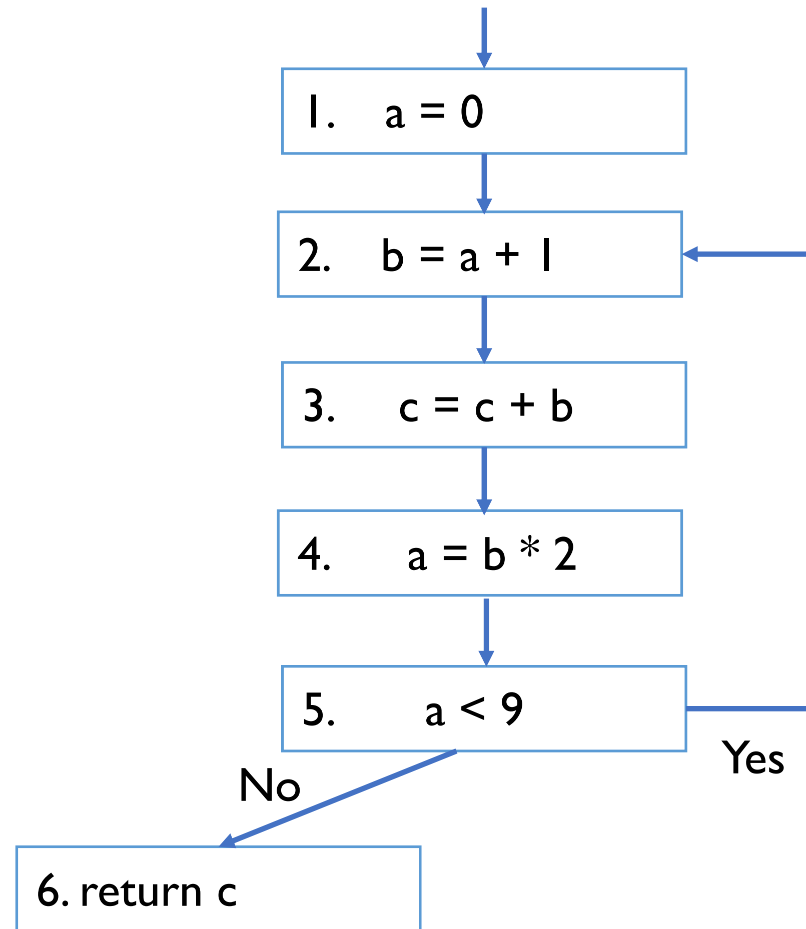
Conservative Approximation

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 Z:

Does not identify c as live in all cases

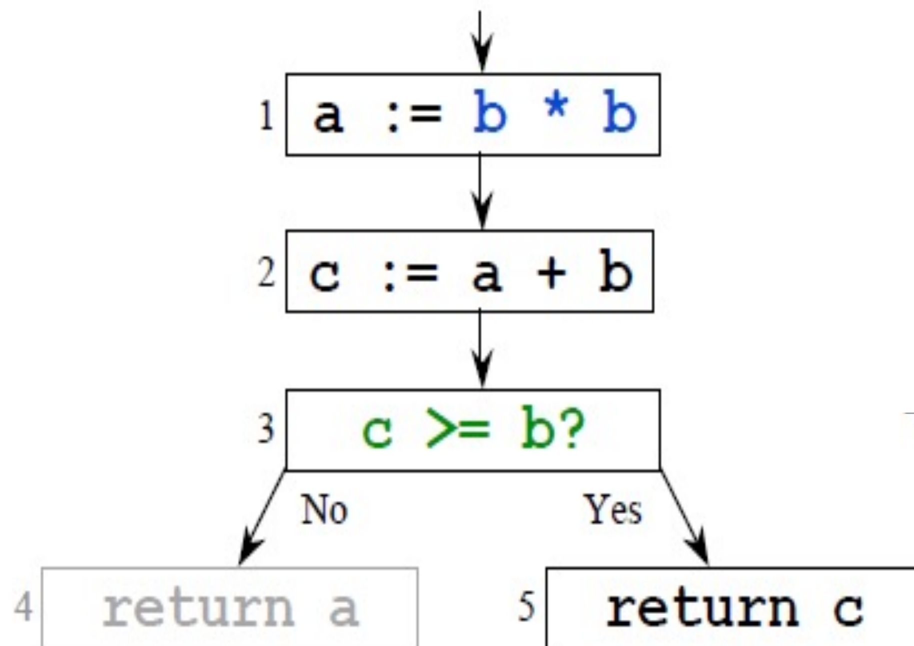
– Does Z lead to a correct program?



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

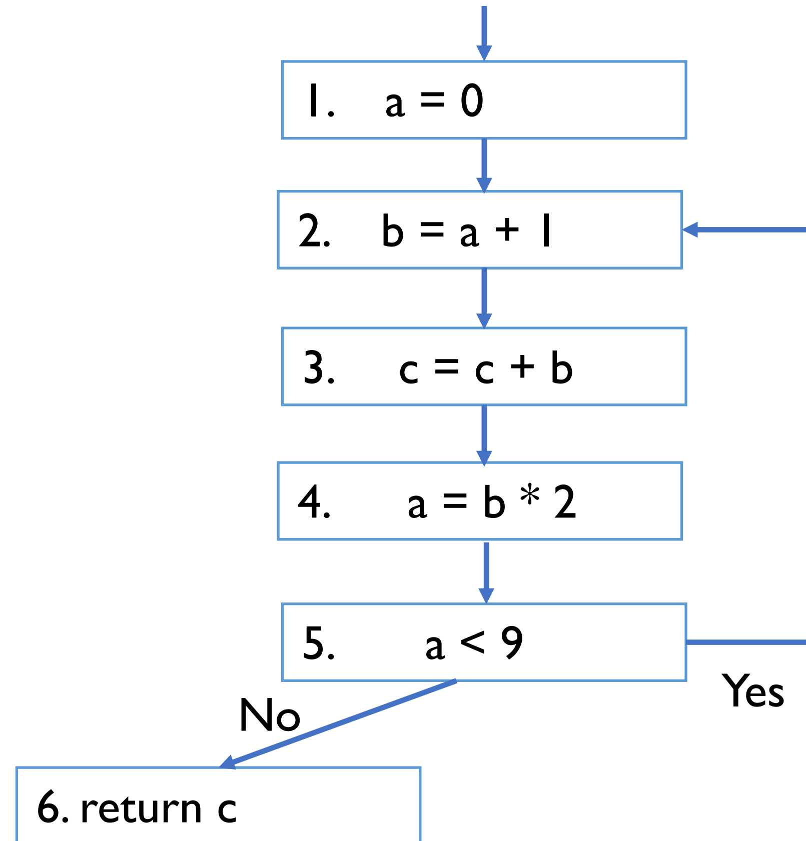


No compiler can statically identify all infeasible paths

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

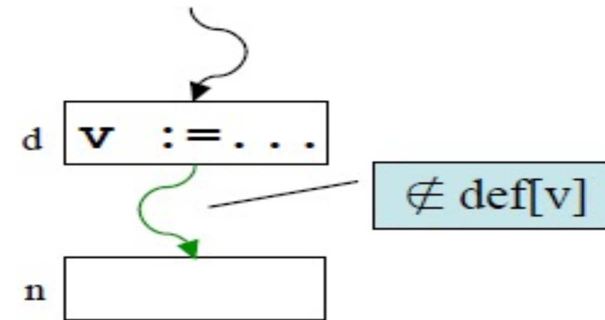
You need **2** registers Why?



Example 2: Reaching Definition

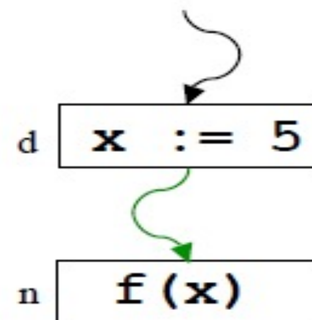
Definition

- 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

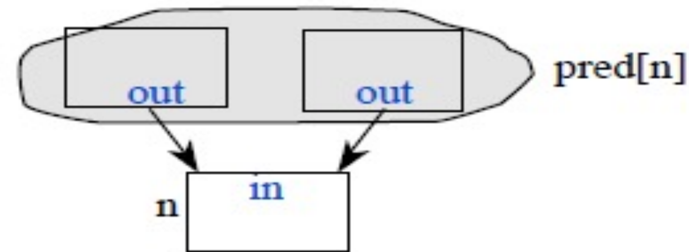
- **Gen[n]:** Definitions that are generated by node n (at most one)
- **Kill[n]:** Definitions that are killed by node n

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 := $f(y_1,\ldots,y_n)$	$\{f,y_1,\ldots,y_n\}$	{x}

Data-flow equations for Reaching Definition

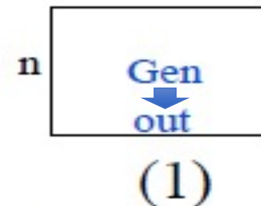
The in set

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

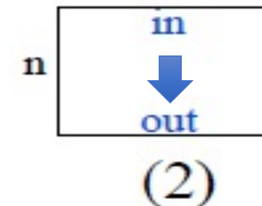


The out set

- 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



(1)



(2)

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

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

Recall Liveness Analysis

- Data-flow Equation for liveness

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

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

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

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

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

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

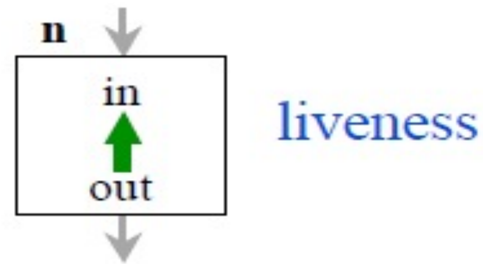
Can define almost any data-flow analysis in terms of Gen and Kill

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[]

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

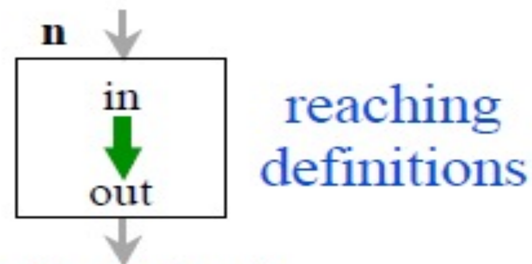


liveness

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[]

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



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

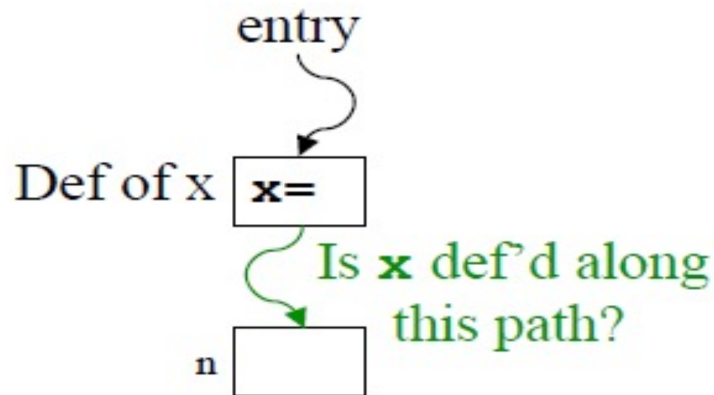
$$in[n] = \bigcup_{p \in pred[n]} out[s]$$

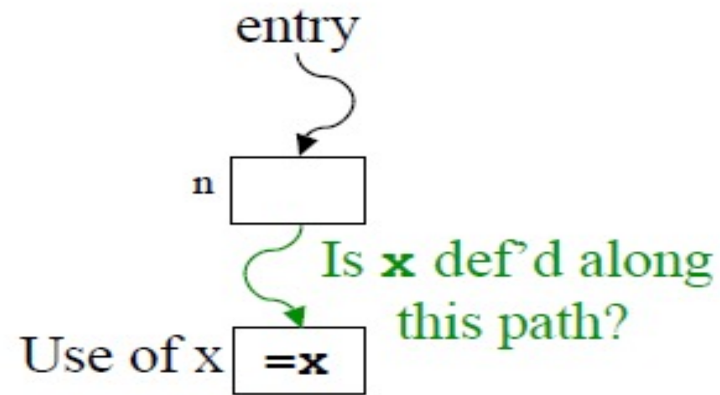
$$out[n] = gen[n] \cup (in[n] - kill[n])$$

Live Variables

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

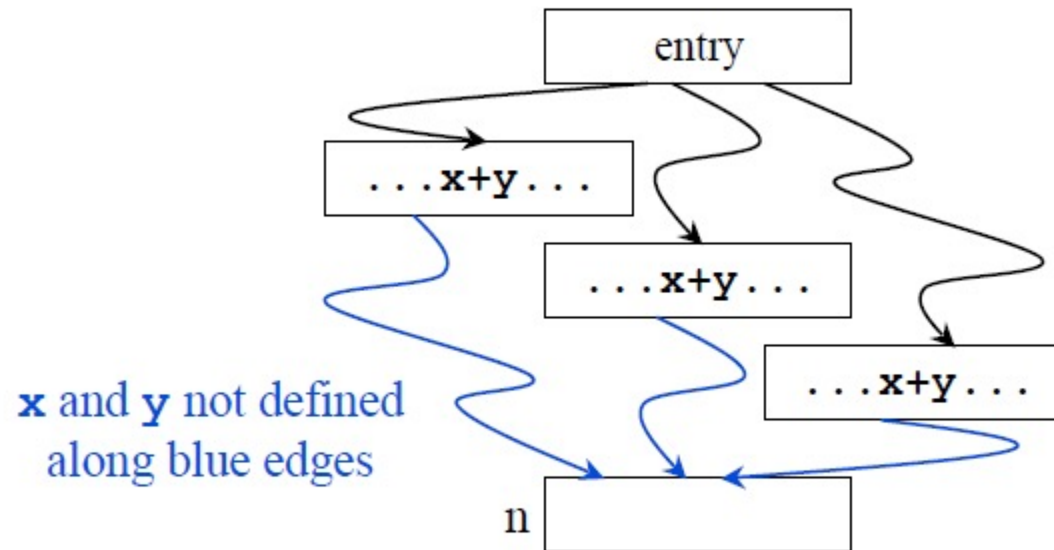
$$in[n] = gen[n] \cup (out[n] - kill[n])$$





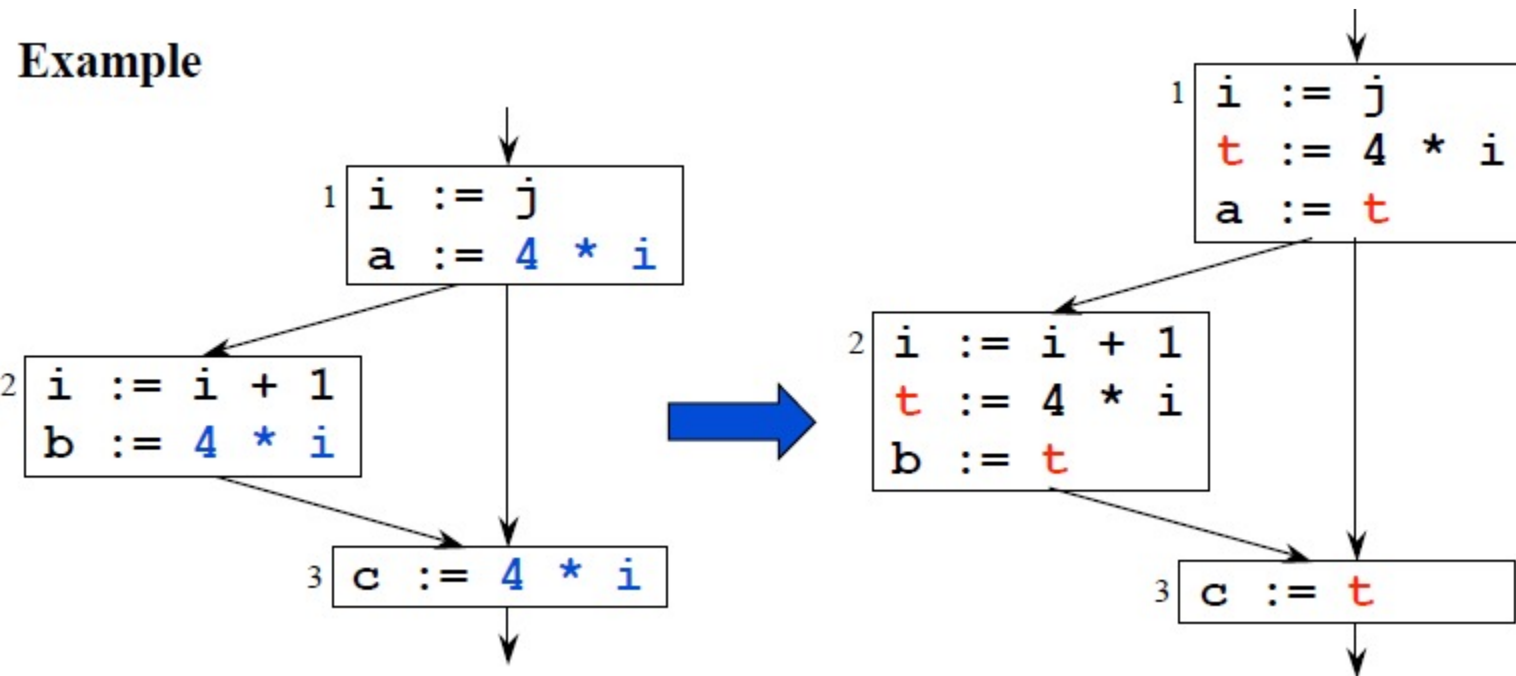
Available Expression

- 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

Security Analysis Techniques

- Testing/Fuzzing
- Static Analysis (Already covered)
- Symbolic Execution
- Concolic Execution
- Formal Verification

Automatic test case generation

Static analysis

Program verification

Fuzzing

Dynamic symbolic execution

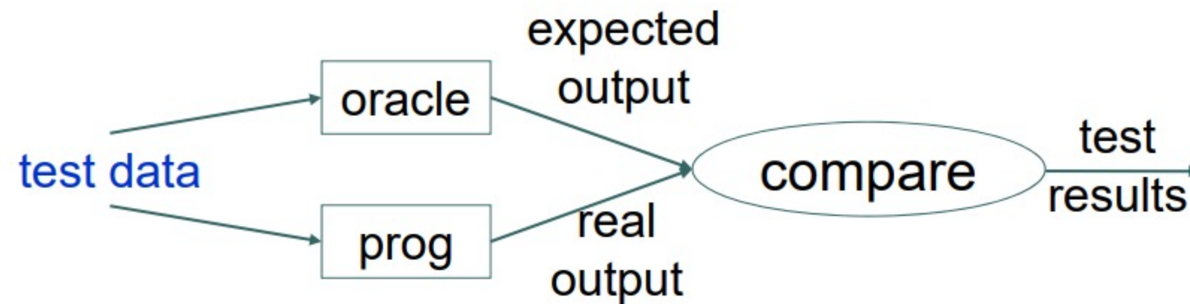
Lower coverage
Lower false positives
Higher false negatives

Higher coverage
Higher false positives
Lower false negatives

Testing

PennState

- Testing: the process of running a program on a set of test cases and comparing the actual results with expected results (according to the specification).



  - For the implementation of a factorial function, test cases could be {0, 1, 5, 10}. What is missing?
  - Can it guarantee correctness?
    - Correctness: For all possible values of n, your factorial program will provide correct output.
    - Verification: High cost!

Fuzz Testing

- Idea proposed by Bart Miller at Wisconsin in 1988 after experiencing an unusual crash while accessing a Unix utility remotely

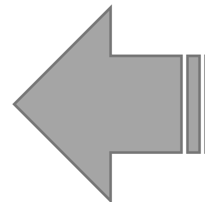
```
format.c (line 276):
... while (lastc != '\n') { //reading line
      rdc(); }

input.c (line 27):
   rdc() {
       do { //skipping space and tab
         readchar();
       } while (lastc == ' ' || lastc == '\t');
       return (lastc);
   }
```

Fuzz Testing

Fuzz Testing

- Idea proposed by Bart Miller at Wisconsin in 1988 after experiencing an unusual crash while accessing a Unix utility remotely

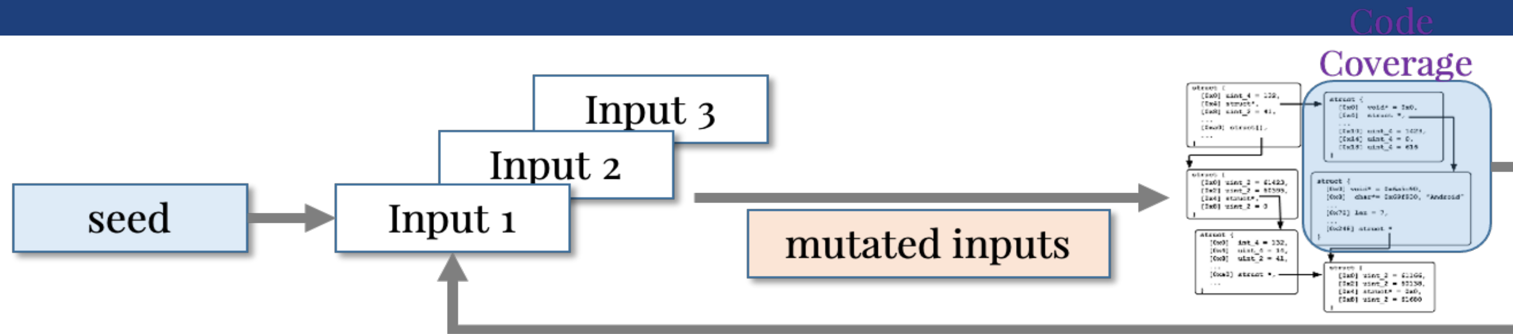
```
format.c (line 276):
... while (lastc != '\n') { //reading line
      rdc(); }

input.c (line 27):
   rdc() {
       do { //reading words
         readchar();
       } while (lastc == ' ' || lastc == '\t');
       return (lastc);
   }
```



When end of file, readchar() sets lastc to be 0; then the program hangs (infinite loop)

Fuzzing

PennState



- Fuzzing is an automated form of testing that runs code on (semi) random and (abnormal) input.
  - Black Box (based on specification): e.g., input is non-negative
  - White Box (source/binary): e.g., if(x>y and y>z) then … else .
- Mutation-based fuzzing generates test cases by mutating existing test cases.
- Generation-based fuzzing generates test cases based on a model of the input (i.e., a specification). It generates inputs "from scratch" rather than using an initial input and mutating.
- Any inputs that crash the program are recorded.
  - Crashes are then sorted, reduced, and bugs are extracted. Bugs are then analyzed individually (is it a security vulnerability?).

Blackbox Fuzzing

- Given a program simply feed random inputs and see whether it exhibits incorrect behavior (e.g., crashes)
- Advantage: easy, low programmer cost
- Disadvantage: inefficient
  - Inputs often require structures, random inputs are likely to be malformed
  - Inputs that trigger an incorrect behavior is a a very small fraction, probably of getting lucky is very low

Fuzzing

- Automatically generate test cases
- Many slightly anomalous test cases are input into a target
- Application is monitored for errors
- Inputs are generally either file based (.pdf, .png, .wav, etc.) or network based (http, SNMP, etc.)

Problem detection

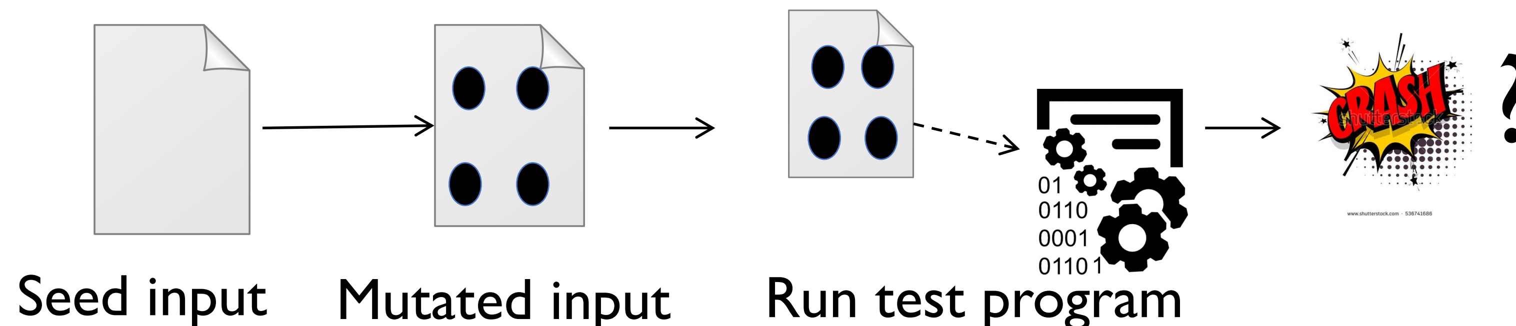
- See if program crashed
  - Type of crash can tell a lot (SEGV vs. assert fail)
- Run program under dynamic memory error detector (valgrind/purify/AddressSanitizer)
  - Catch more bugs, but more expensive per run.
- See if program locks up
- Roll your own dynamic checker e.g. valgrind skins

Regression vs. Fuzzing

	Regression	Fuzzing
Definition	Run program on many normal inputs, look for badness	Run program on many abnormal inputs, look for badness
Goals	Prevent normal users from encountering errors (e.g., assertion failures are bad)	Prevent attackers from encountering exploitable errors (e.g., assertion failures are often ok)

Enhancement 1: Mutation-Based fuzzing

- Take a well-formed input, randomly perturb (flipping bit, etc.)
- Little or no knowledge of the structure of the inputs is assumed
- Anomalies are added to existing valid inputs
  - Anomalies may be completely random or follow some heuristics (e.g., remove NULL, shift character forward)
- Examples: ZZUF, Taof, GPF, ProxyFuzz, FileFuzz, Filep, etc.



Example: fuzzing a PDF viewer

- Google for .pdf (about 1 billion results)
- Crawl pages to build a corpus
- Use fuzzing tool (or script)
  - Collect seed PDF files
  - Mutate that file
  - Feed it to the program
  - Record if it crashed (and input that crashed it)

Mutation-based fuzzing

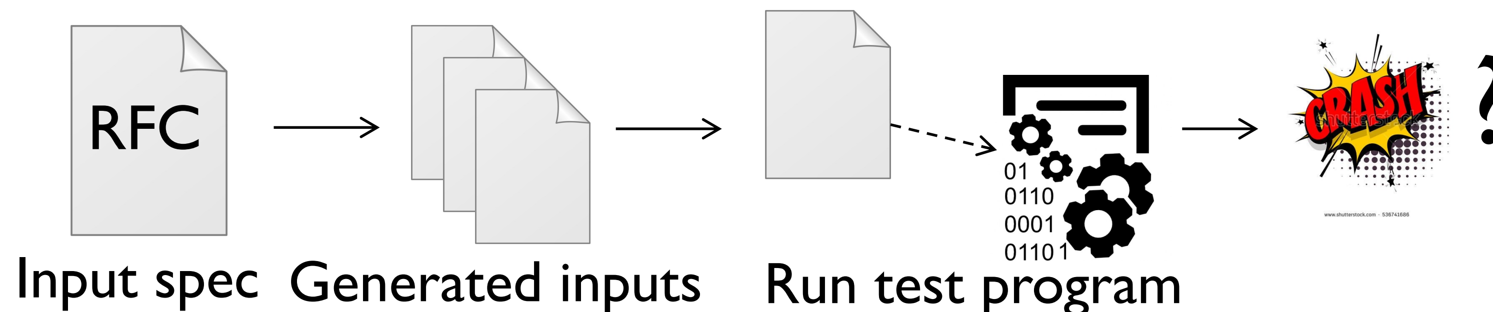
PennState

- Super easy to setup and automate
- Little or no file format knowledge is required
- Limited by initial corpus
- May fail for protocols with checksums, those which depend on challenge

Enhancement II: Generation-Based Fuzzing

PennState

- Test cases are generated from some description of the input format: RFC, documentation, etc.
  - Using specified protocols/file format info
  - E.g., SPIKE by Immunity
- Anomalies are added to each possible spot in the inputs
- Knowledge of protocol should give better results than random fuzzing



Mutation-based vs. Generation-based

- Mutation-based fuzzer
  - Pros: Easy to set up and automate, little to no knowledge of input format required
  - Cons: Limited by initial corpus, may fall for protocols with checksums and other hard checks
- Generation-based fuzzers
  - Pros: Completeness, can deal with complex dependencies (e.g., checksum)
  - Cons: writing generators is hard, performance depends on the quality of the spec

How much fuzzing is enough?

PennState

- Mutation-based-fuzzers may generate an infinite number of test cases. When has the fuzzer run long enough?
- Generation-based fuzzers may generate a finite number of test cases. What happens when they're all run and no bugs are found?

Code coverage

- Some of the answers to these questions lie in *code coverage*
- Code coverage is a metric that can be used to determine how much code has been executed.
- Data can be obtained using a variety of profiling tools. e.g. gcov, lcov

Line coverage

PennState

- **Line/block coverage**: Measures how many lines of source code have been executed.
- For the code on the right, how many test cases (values of pair (a,b)) needed for full(100%) line coverage?

```
if( a >  2  )
      a =  2;
if( b >2  )
      b =  2;
```

Branch coverage

PennState

- Branch coverage: Measures how many branches in code have been taken (conditional jmps)
- For the code on the right, how many test cases needed for full branch coverage?

```
if( a >  2  )
      a =  2;
if( b >2  )
      b =  2;
```

Path coverage

PennState

- Path coverage: Measures how many paths have been taken
- For the code on the right, how many test cases needed for full path coverage?

```
if( a >  2  )
     a =  2;
if( b >2  )
     b =  2;
```

Benefits of Code coverage

PennState

- Can answer the following questions
  - How good is an initial file?
  - Am I getting stuck somewhere?

```
if (packet[0x10] < 7) { //hot path
} else { //cold path }
```

  - How good is fuzzerX vs. fuzzerY
  - Am I getting benefits by running multiple fuzzers?

Enhancement III: Coverage-guided gray-box fuzzing

- Special type of mutation-based fuzzing
  - Run mutated inputs on instrumented program and measure code coverage
  - Search for mutants that result in coverage increase
  - Often use genetic algorithms, i.e., try random mutations on test corpus and only add mutants to the corpus if coverage increases
  - Examples: AFL, libfuzzer

American Fuzzy Lop (AFL)

Seed inputs → Input queue → Next input → Mutation → Execute against instrumented target → branch/edge coverage increased? → Add mutant to the queue

Periodically calls the queue without affecting total coverage

Data-flow-guided fuzzing

- Intercept the data flow, analyze the inputs of comparisons
    - Incurs extra overhead
- Modify the test inputs, observe the effect on comparisons
- Prototype implementations in libFuzzer and go-fuzz

Static Analysis

PennState

- Limitation of dynamic testing:
  - We cannot find all vulnerabilities in a program
- *Can we build a technique that identifies *all* vulnerabilities?*
  - *Turns out that we can: static analysis*
    - Explore all possible executions of a program
      - All possible inputs
      - All possible states
  - *But, it has its own major limitation*
    - *Can identify many false positives (not actual vulnerabilities)*
  - *Can be effective when used carefully*

Static Analysis

PennState

- Provides an approximation of behavior
- "Run in the aggregate"
  - Rather than executing on ordinary states
  - Finite-sized descriptors representing a collection of states
- "Run in non-standard way"
  - Run in fragments
  - Stitch them together to cover all paths
- Various properties of programs can be tracked
- Control flow, Data flow, Types
- Which ones will expose which vulnerabilities

Control Flow Analysis

PennState

Can we detect code with no return check?

```
format.c (line 276):
while (lastc != '\n')
{ //reading line
 rdc();
}
```

```
input.c (line 27):
rdc() {
 do { //reading words
     readchar(); }
while (lastc == ' ' ||
lastc == '\t');
   return (lastc);
}
```

- To find an execution path that does not check the return value of a function
  - ❑ That is actually run by the program
  - ❑ How do we do this? Control Flow Analysis

Static vs. Dynamic

PennState

- Dynamic
  - Depends on concrete inputs
  - Must run the program
  - Impractical to run all possible executions in most cases
- Static
  - Overapproximates possible input values (sound)
  - Assesses all possible runs of the program at once
  - Setting up static analysis is somewhat of an art form
- Is there something that combines best of both?
  - Can’t quite achieve all these, but can come closer

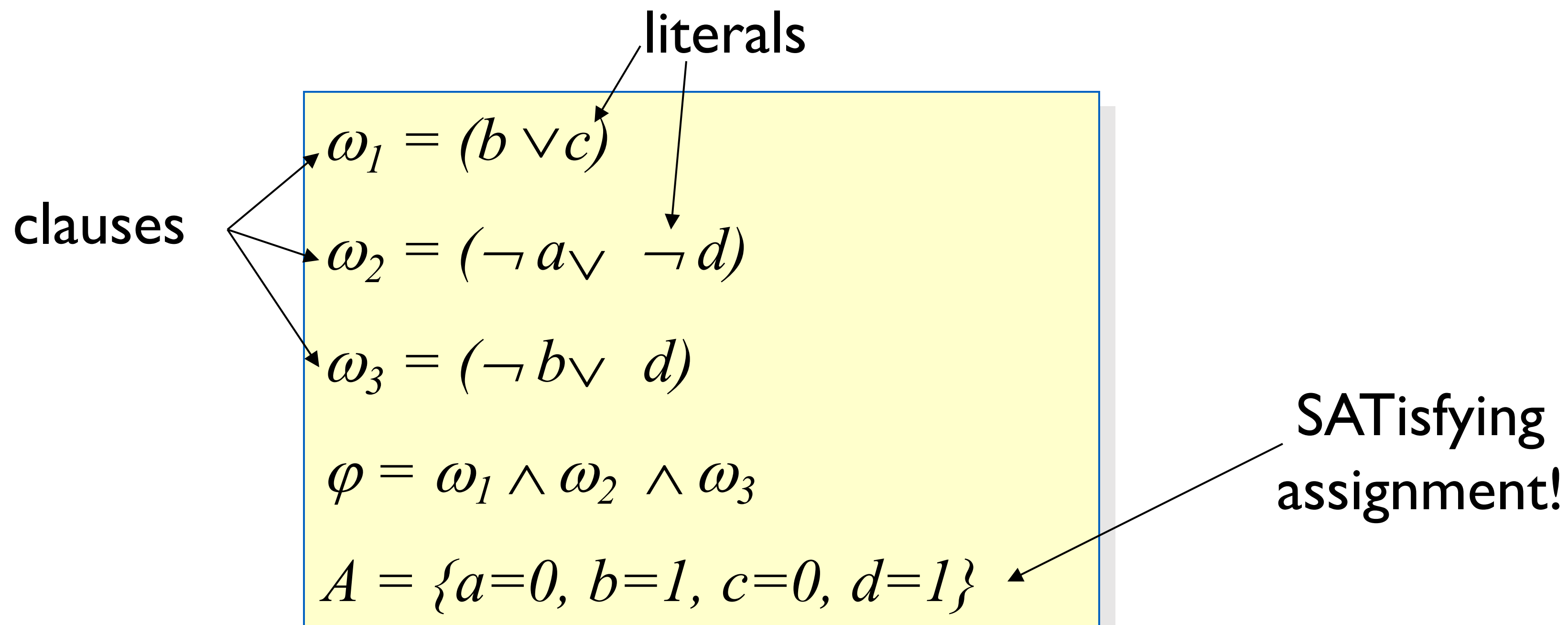
Symbolic Execution

PennState

- Symbolic execution is a method for emulating the execution of a program to learn constraints
  - Assign variables to symbolic values instead of concrete values
  - Symbolic execution tells you what values are possible for symbolic variables at any particular point in your program
- Like dynamic analysis (fuzzing) in that the program is executed in a way – albeit on symbolic inputs
- Like static analysis in that one start of the program tells you what values may reach a particular state

Background: SAT

Given a propositional formula in CNF, find if there exists an assignment to Boolean variables that makes the formula true:



Background: SMT

SMT: Satisfiability Modulo Theories

Input: a first-order formula φ over background theory

Output: is φ satisfiable?

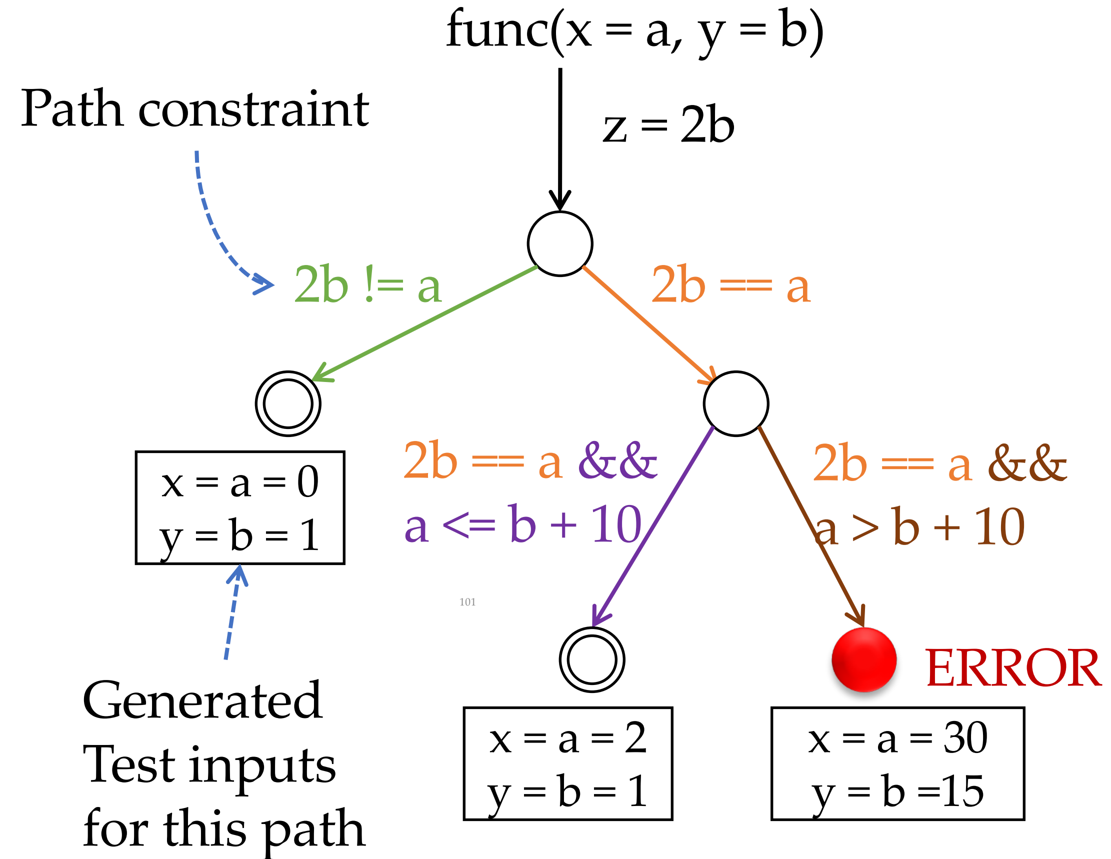
- does φ have a model?
- Is there a refutation of φ = proof of ¬φ?

For most SMT solvers: φ is a ground formula

- Background theories: Arithmetic, Arrays, Bit-vectors, Algebraic Datatypes
- Most SMT solvers support simple first-order sorts

Symbolic Execution

```
Void func(int x, int
  y){
  int z = 2 * y;
  if(z == x){
        if (x > y + 10)
            ERROR
  }
}
int main(){
  int x = sym_input();
  int y = sym_input();
  func(x, y);
  return 0;
}
```

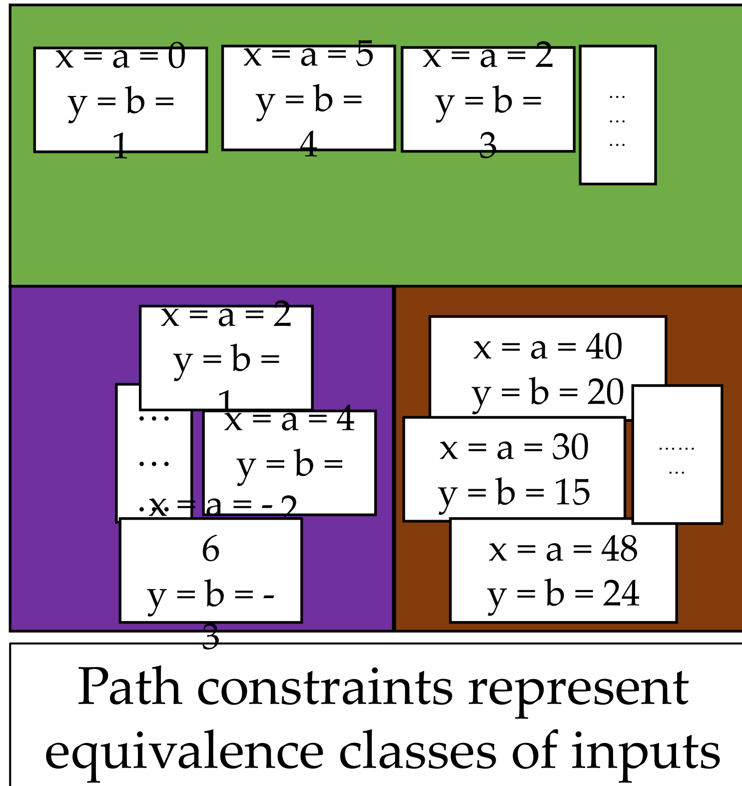


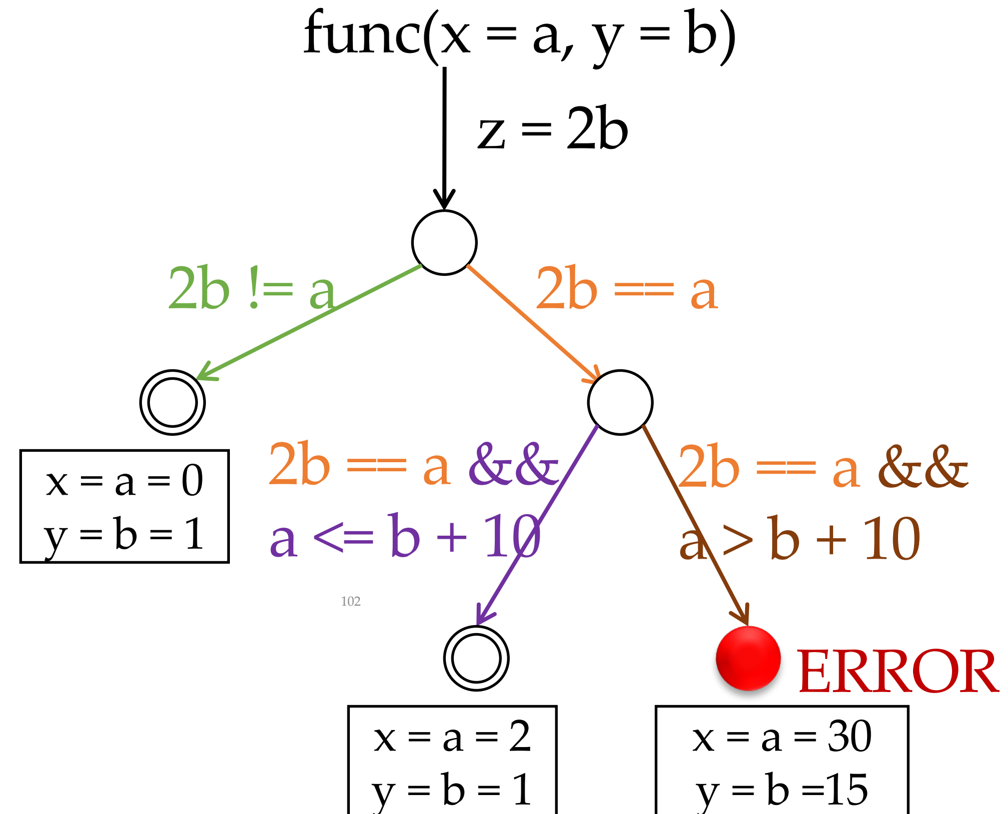
Note: Require inputs to be marked as symbolic

Symbolic Execution

PennState

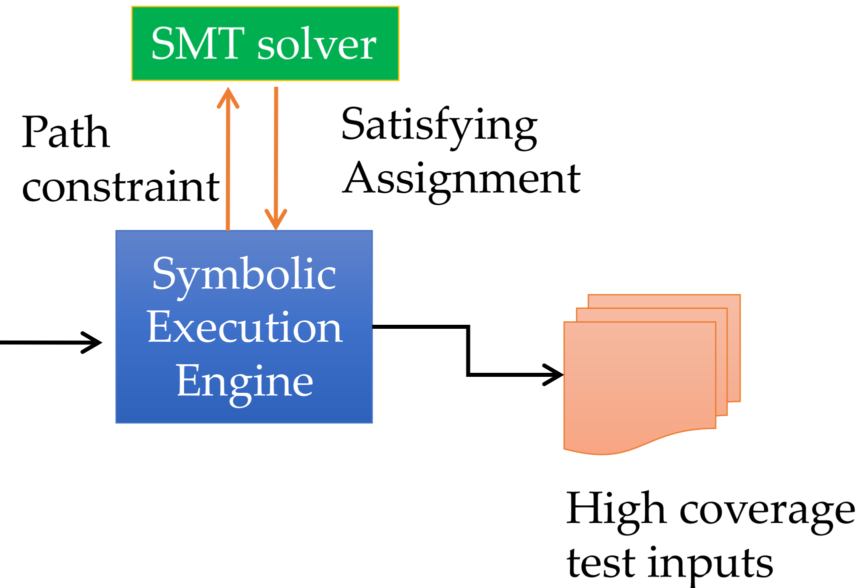
How does symbolic execution work?





Symbolic Execution


```
Void func(int x, int
  y){
  int z = 2 * y;
  if(z == x){
        if (x > y + 10)
              ERROR
  }
}
int main(){
  int x = sym_input();
  int y = sym_input();
  func(x, y);
  return 0;
}
```



Symbolic Execution

PennState

- Execute the program with symbolic valued inputs (**Goal: good path coverage**)
- Represents *equivalence class of inputs* with first order logic formulas (**path constraints**)
- One path constraint abstractly represents all inputs that induces the program execution to go down a specific path
- Solve the path constraint to obtain one representative input that exercises the program to go down that specific path

Symbolic Execution

PennState

- Instead of concrete state, the program maintains symbolic states, each of which maps variables to symbolic values
- Path condition is a quantifier-free formula over the symbolic inputs that encodes all branch decisions taken so far
- All paths in the program form its execution tree, in which some paths are feasible and some are infeasible

Symbolic Execution Tools

PennState

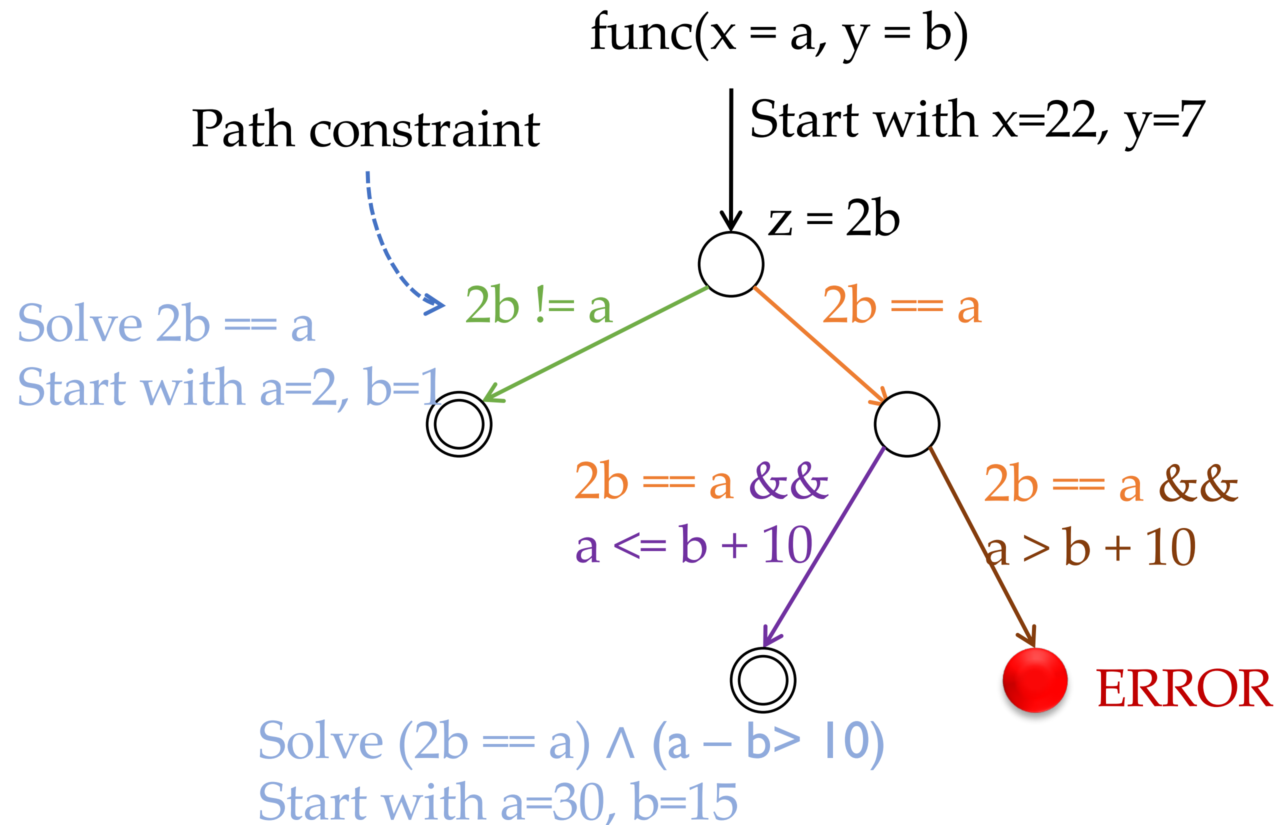
- FuzzBALL:
  - Works on binaries, generic SE engine. Used to, e.g., find PoC exploits given a vulnerability condition.
  - KLEE: Instruments through LLVM-based pass, relies on source code. Used to, e.g., nd bugs in programs.
  - S2E: Selective Symbolic Execution: automatic testing of large source base, combines KLEE with an concolic execution. Used to, e.g., test large source bases (e.g., drivers in kernels) for bugs.
- Efficiency of SE tool depends on the search heuristics and search strategy. As search space grows exponentially, a good search strategy is crucial for efficiency and scalability.

Symbolic Execution Summary

- Symbolic execution is a great tool to find vulnerabilities or to create PoC exploits.
- Symbolic execution is limited in its scalability. An efficient search strategy is crucial.

Concolic Execution


```
Void func(int x, int
  y){
  int z = 2 * y;
  if(z == x){
        if (x > y + 10)
              ERROR
  }
}
int main(){
  int x = sym_input();
  int y = sym_input();
  func(x, y);
  return 0;
}
```



Formal Verification

- Formal verification is the act of using formal methods to proving or disproving the correctness of a certain system given its formal specification.
- Formal verification requires a specification and an abstraction mechanism to show that the formal specification either holds (i.e., its correctness is proven) or fails (i.e., there is a bug).
- Verification is carried out by providing a formal proof on the abstracted mathematical model of the system according to the specification. Many different forms of mathematical objects can be used for formal verification like finite state machines or formal semantics of programming languages (e.g., operational semantics or Hoare logic).

Takeaways

PennState

- Testing is simple but only tests for presence of functionality.
- Fuzzing uses test cases to explore other paths, might run forever.
- Static analysis has limited precision (e.g., aliasing).
- Symbolic execution needs guidance when searching through program.
- Formal verification is precise but arithmetic operations can be difficult.
- All mechanisms (except testing) run into state explosion.

Thanks

Thanks to Omar Choowdhury, Suman Jana and Baishakhi Ray
for some slides.