Program Analysis

Dynamic Analysis Frameworks

Prof. Dr. Michael Pradel

Software Lab, University of Stuttgart

Winter 2021/2022

Warm-up Quiz

```
var a;
var a, a;
var a, a, a = a;
a = eval("var a;")
a = function a(a, a) {
    return a;
}
a = a(null, a);
console.log(a.name);
```

Warm-up Quiz

```
var a;
var a, a;
var a, a, a = a;
a = eval("var a;")
a = function a(a, a) {
    return a;
}
a = a(null, a);
console.log(a.name);
```

Result: a

Warm-up Quiz

```
var a;
var a, a;
var a, a, a = a;
a = eval("var a;")
a = function a(a, a) {
    return a;
}
a = a(null, a);
console.log(a.name);
```

a is a function that returns the second argument, i.e., the function itself

Result: a

Outline

1. **Introduction**
2. **Special-Purpose Dynamic Analysis**
3. **General-Purpose Frameworks**

Relevant papers:

- *Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation*, Nethercote et al., PLDI 2007
- *Jalangi: A Selective Record-Replay and Dynamic Analysis Framework for JavaScript*, Sen et al., FSE 2013
- *Wasabi: A Framework for Dynamically Analyzing WebAssembly*, Lehmann et al., ASPLOS 2019

Dynamic Analysis

- Execute an instrumented program to gather information that can be analyzed to learn about a property of interest
- Precise: All observed behavior actually happens
- Incomplete: Very difficult to cover all possible behaviors

Examples

- Coverage: Track which lines or branches get executed
- Call graph: Track which functions call which other functions
- Slicing: Track dependencies to produce a reduced program
- We’ll see more in upcoming lectures

Examples

- **Coverage**: Track which lines or branches get executed
- **Call graph**: Track which functions call which other functions
- **Slicing**: Track dependencies to produce a reduced program
- We'll see more in upcoming lectures

Different goals, similar challenges: Use a common framework

Outline

1. Introduction

2. Special-Purpose Dynamic Analysis ←

3. General-Purpose Frameworks

Relevant papers:

- *Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation*, Nethercote et al., PLDI 2007
- *Jalangi: A Selective Record-Replay and Dynamic Analysis Framework for JavaScript*, Sen et al., FSE 2013
- *Wasabi: A Framework for Dynamically Analyzing WebAssembly*, Lehmann et al., ASPLOS 2019

Coverage Analysis

Goal: Track which branches are executed

```
x = readInput();
if (x > 0) {
  y = 2;
  y = 3
  while (y > 0) {
    y = y - x;
  }
} else {
  y = 3
}
```

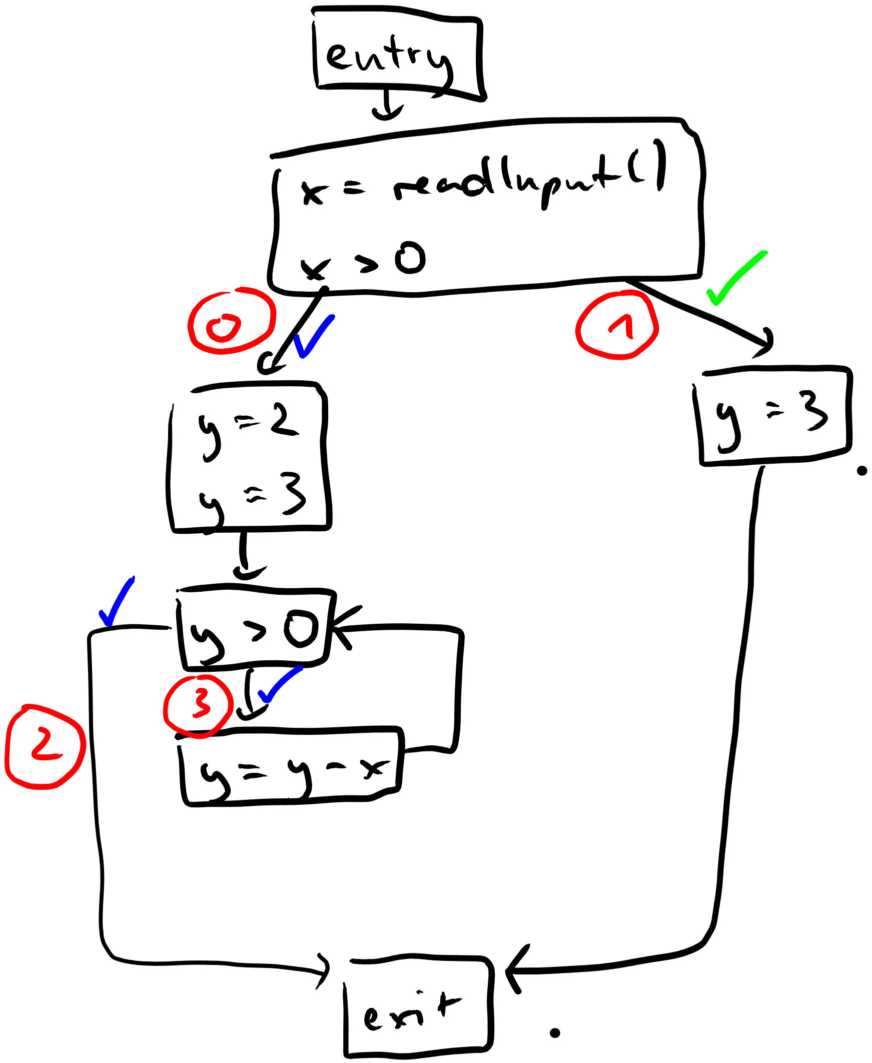
Instrumented Program

Add instrumentation code at **beginning of each basic block**

```
x = readInput();
if (x > 0) {

  y = 2;
  y = 3
  while (y > 0) {

    y = y - x;
  }

} else {

  y = 3
}
```

→

```
cov = [false, false, false, false];
x = readInput();
if (x > 0) {
  cov[0] = true;
  y = 2;
  y = 3
  while (y > 0) {
    cov[3] = true;
    y = y - x;
  }
  cov[2] = true;
} else {
  cov[1] = true;
  y = 3
}
```



Input 1 : 5

Input 2 : -5

3/4 branches covered

1/4 branches covered

Quiz

```
cov = [false, false, false, false];
x = readInput();
if (x > 0) {
  cov[0] = true;
  y = 2;
  y = 3
  while (y > 0) {
    cov[3] = true;
    y = y - x;
  }
  cov[2] = true;
} else {
  cov[1] = true;
  y = 3
}
```

Given the input 1, what’s the branch coverage?

Quiz

```
cov = [false, false, false, false];
x = readInput();
if (x > 0) {
  cov[0] = true;
  y = 2;
  y = 3
  while (y > 0) {
    cov[3] = true;
    y = y - x;
  }
  cov[2] = true;
} else {
  cov[1] = true;
  y = 3
}
```

Given the input 1, what's the branch coverage?

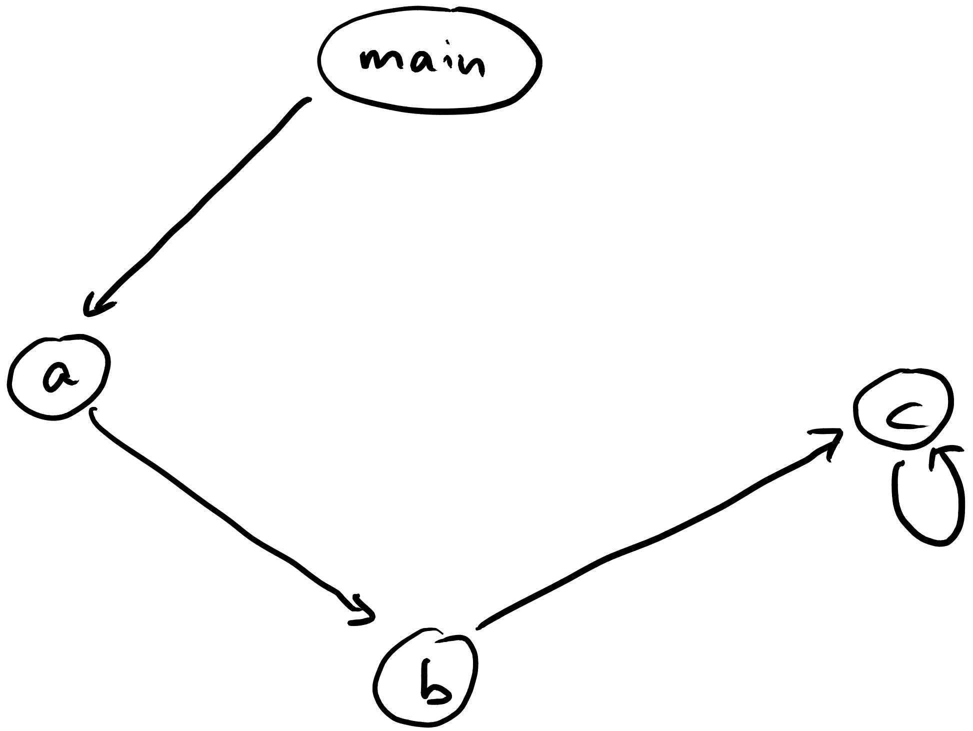
Answer:

`[true, false, true, true]`

Call Graph Analysis

Goal: Track “calls” relationships between functions

```
n = readInput();
function a() {
  b();
}
function b() {
  if (n == 5) {
    c();
  }
}
function c() {
  if (n == 5) {
    c();
    n—;
  }
}
a();
```



Static overapproximation of call graph

Instrumented Program

Add instrumentation code at each call site

```
n = readInput();
function a() {
  b();
}
function b() {
  if (n == 5) {
    c();
  }
}
function c() {
  if (n == 5) {
    c();
    n--;
  }
}
a();
```

→

```
calls = new Set();
n = readInput();
function a() {
  calls.add("a->b"); b();
}
function b() {
  if (n == 5) {
    calls.add("b->c"); c();
  }
}
function c() {
  if (n == 5) {
    calls.add("c->c"); c();
    n--;
  }
}
calls.add("main->a"); a();
```

Outline

1. **Introduction**
2. **Special-Purpose Dynamic Analysis**
3. **General-Purpose Frameworks** ←

Relevant papers:

- *Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation*, Nethercote et al., PLDI 2007
- *Jalangi: A Selective Record-Replay and Dynamic Analysis Framework for JavaScript*, Sen et al., FSE 2013
- *Wasabi: A Framework for Dynamically Analyzing WebAssembly*, Lehmann et al., ASPLOS 2019

Commonalities

Different dynamic analyses, but many commonalities

- Specific runtime events to track
- Analysis updates some state in response to events

Commonalities

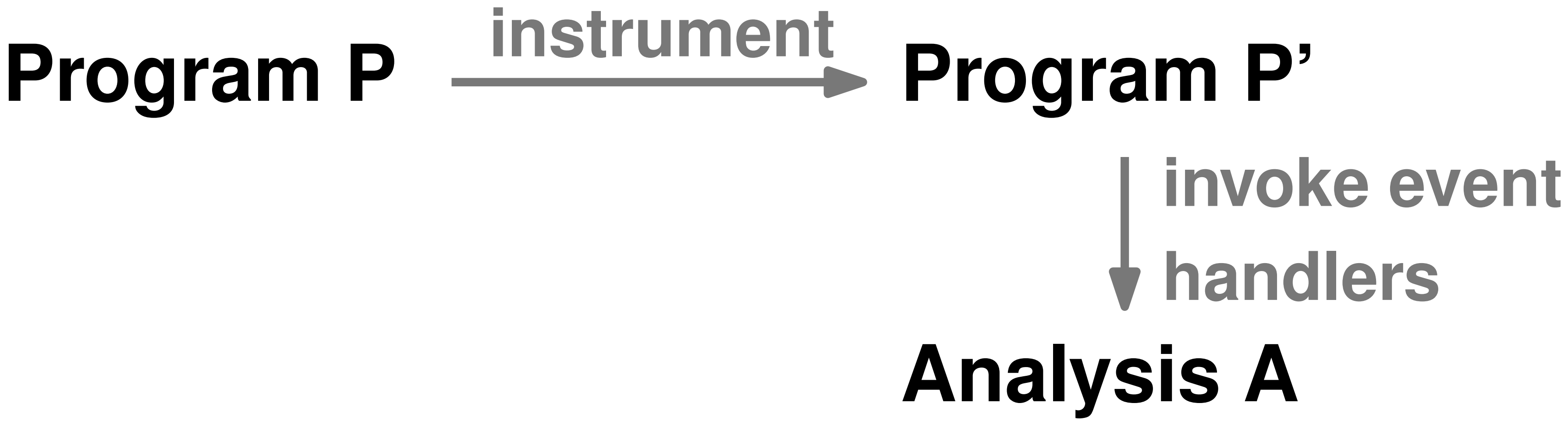
Different dynamic analyses, but many commonalities

- Specific runtime events to track
- Analysis updates some state in response to events

Can avoid re-implementing everything from scratch for each new analysis?

Dynamic Analysis Frameworks

- Set of **kinds of runtime events**
- Analysis can **register** for specific events
- At runtime, instrumented program **invokes event handlers**



Typical Runtime Events

Event	Example
Arithmetic operation	`2+3`
Boolean operation	`a > 0`
Branch	`if (c) ...`
Function call	`g()`
Return from function call	`x = g()`
Write into variable or field	`x.f = z`
Read of variable or field	`x.f = z`

(and many others)

Example

```
a = readInput();
b = a + 3;
if (b == -23) {
  foo();
} else {
  b = 5;
}
```

Runtime events:

- Arithmetic operations
- Boolean operations
- Reads of variables
- Writes into variables
- Function calls

Input: -26

What sequence of events get triggered?

Runtime Events : Example

- call of readInput
- write -26 into a
- read of a (-26)
- arithmetic operation (-26 + 3 = -23)
- write of -23 into b
- read of b (-23)
- boolean operation (-23 == -23 ⇝ true)
- call of foo()

Extended Operational Semantics

- Tracking runtime events: **Additional behavior** performed during program execution
- Formally describe by **extending the operational semantics**

Extending Small-Step Operational Semantics

Events :
- write to variable ⇝ "write 3 to x"
- branch → "true branch taken"

Extending configuration into :

⟨P, s, e⟩ where P, s as before

e .. sequence of events (represented as strings)

Replace all axioms & rules to use triple configuration, e.g.,

$$\frac{}{\langle !\ell, s\rangle \longrightarrow \langle n, s\rangle \qquad \text{if } s(\ell) = n}\ (\text{var})$$

becomes

$$\frac{}{\langle !\ell, s, e\rangle \longrightarrow \langle n, s, e\rangle \qquad \text{if } s(\ell) = n}\ (\text{var})$$

events remain the same

Revise some axioms & rules to create new events

1) writes to variables:

$$\frac{}{\langle \ell := n , s \rangle \longrightarrow \langle \mathsf{skip} , s[\ell \mapsto n] \rangle}\ (:=)$$

becomes

$$\frac{}{\langle \ell := n , s , e \rangle \longrightarrow \langle \mathsf{skip} , s[\ell \mapsto n] , e \underset{\text{append to sequence}}{\circ} \text{"write } n \text{ to } \ell\text{"} \rangle}\ (:=)$$

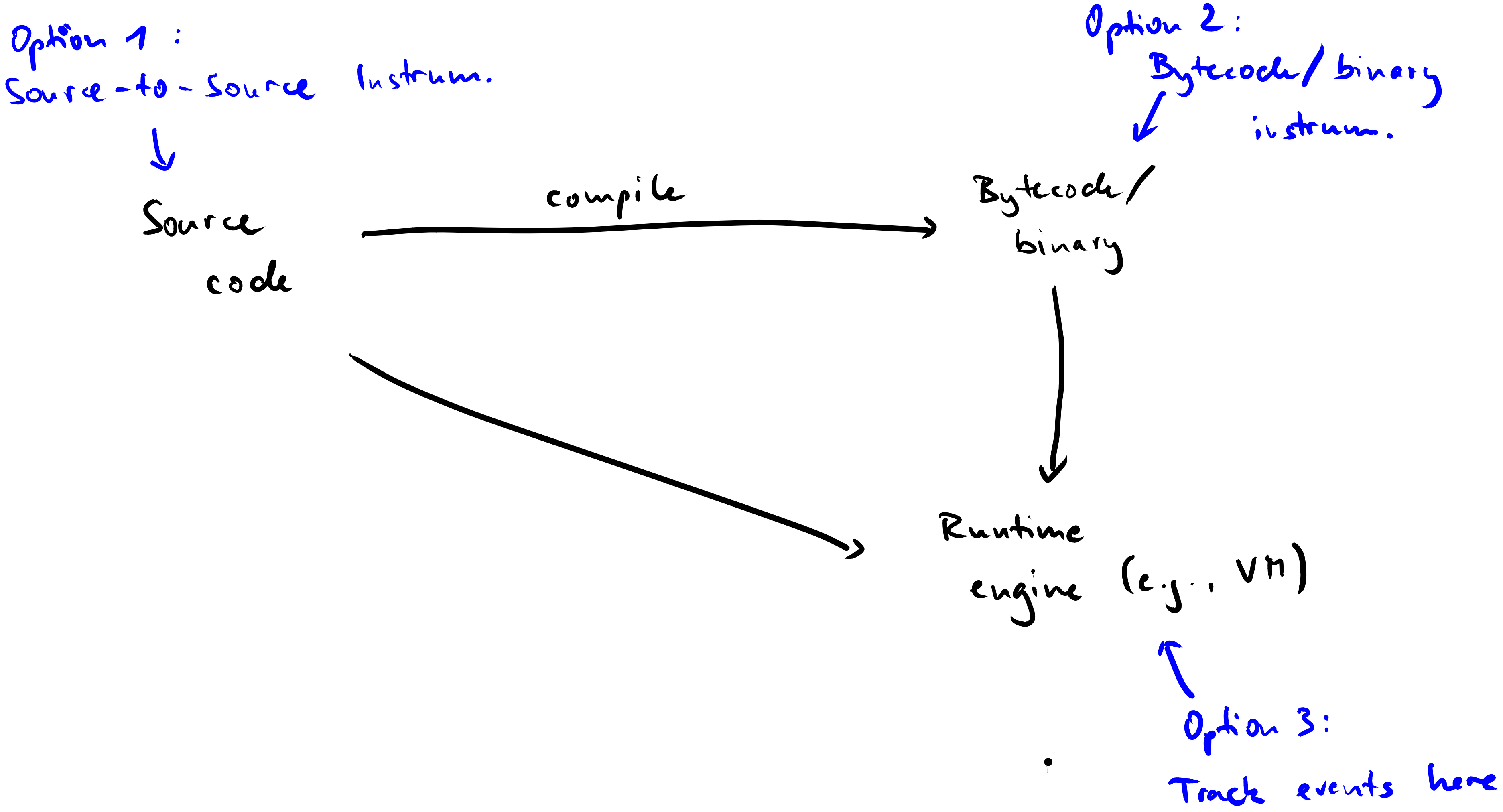
Quiz: Extend axioms & rules for tracking branches

$$\frac{}{\langle \text{if True then } C_1 \text{ else } C_2 , s, e \rangle \to \langle C_1, s, e \circ \text{"true branch taken"} \rangle} \quad (\text{if}_T)$$

(analogous for False)

Implementing Dynamic Analyses

How to **implement a dynamic analysis framework** in practice?


Option 1 :
Source-to-Source Instrum.
Source code
compile
Bytecode/ binary
Option 2 :
Bytecode/ binary instrum.
Runtime engine (e.g., VM)
Option 3:
Track events here

Source Code Instrumentation

Naive approach:
Find and extend particular statements
via regular expressions

Example:

```
// Before: x=y; foo(); a=b;
// After:  x=y; foo(); evt(’call’); a=b;
regex = /; (\w+\(\))/g;
code.replaceAll(regex, "; $1; evt(’call’)")
```

Source Code Instrumentation

Naive approach:
Find and extend particular statements via regular expressions

Example:

Identify function calls

```
// Before: x=y; foo(); a=b;
// After:  x=y; foo(); evt(’call’); a=b;
regex = /; (\w+\(\))/g;
code.replaceAll(regex, "; $1; evt(’call’)")
```

Source Code Instrumentation

Naive approach:
Find and extend particular statements via regular expressions

Example:

Identify function calls

Add call that logs the ’call’ event

```
// Before: x=y; foo(); a=b;
// After:  x=y; foo(); evt(’call’); a=b;
regex = /; (\w+\(\))/g;
code.replaceAll(regex, "; $1; evt(’call’)")
```

Source Code Instrumentation

Naive approach:
Find and extend particular statements via regular expressions

Example:

Identify function calls

Add call that logs the ’call’ event

```
// Before: x=y; foo(); a=b;
// After:  x=y; foo(); evt(’call’); a=b;
regex = /; (\w+\(\))/g;
code.replaceAll(regex, "; $1; evt(’call’)")
```

Cumbersome and extremely brittle:
Don’t do this

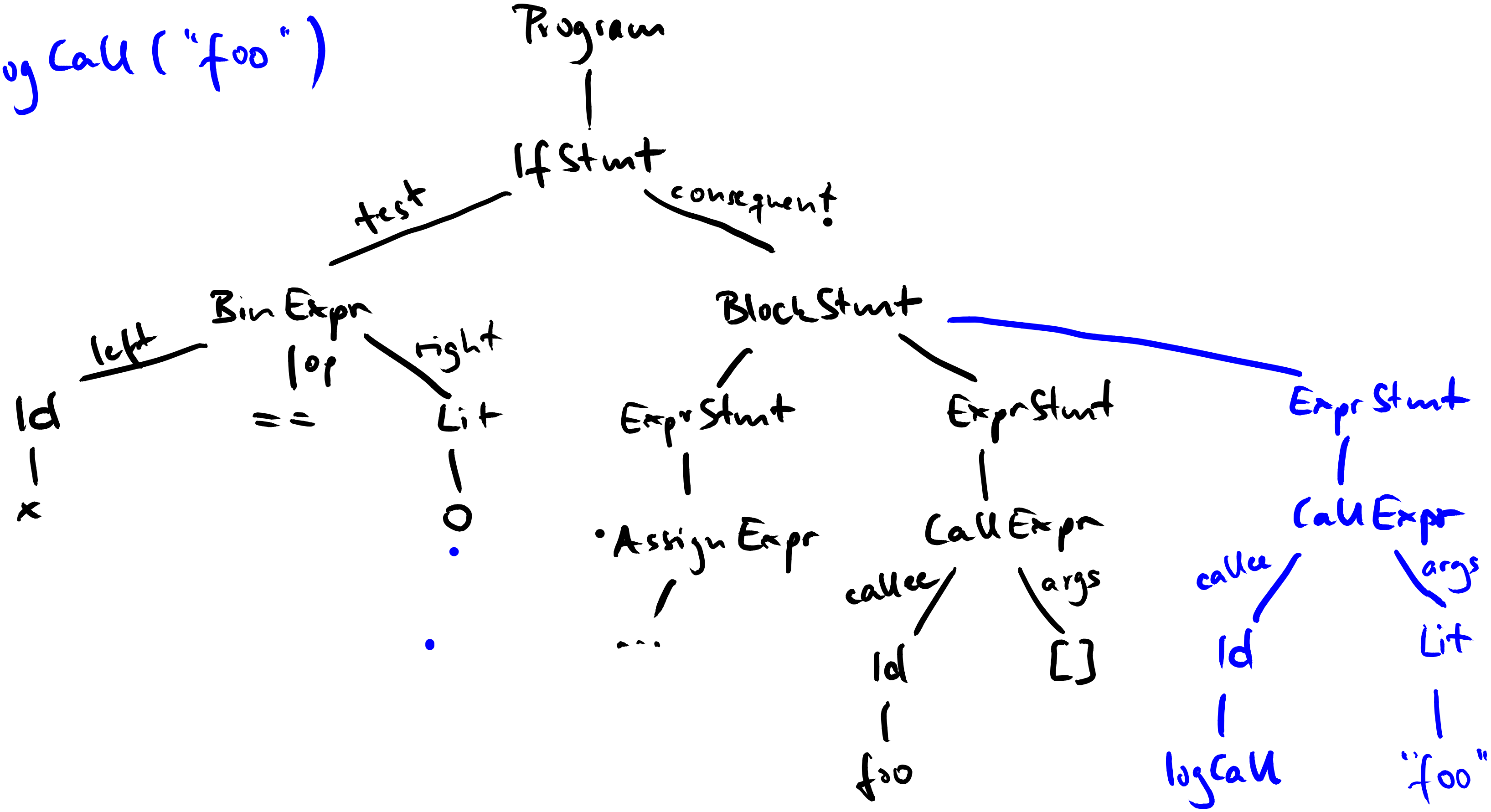
AST-based Instrumentation

More reliable approach:

- **Parse** code into AST
- **Manipulate** AST, e.g., by adding subtrees
- **Pretty-print** AST into code again

AST example

```
if (x == 0) {
    y = 3;
    foo(); ← logCall("foo")
}
```



Real-World Tools

Name	Target language
Pin	x86 binaries
Valgrind	x86 binaries
DiSL	Java
RoadRunner	Java
Jalangi	JavaScript
Wasabi	WebAssembly

Real-World Tools

Name	Target language
Pin	x86 binaries
Valgrind	x86 binaries
DiSL	Java
RoadRunner	Java
Jalangi	JavaScript
Wasabi	WebAssembly

To be used in course project

Developed by my group (main author: Daniel Lehmann)