Practical Program Analysis: Introduction

Join the course on Ilias! See link on
http://software-lab.org/teaching/winter2019/ppa/

Prof. Dr. Michael Pradel

Software Lab, University of Stuttgart

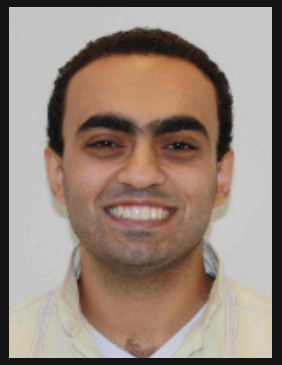
Winter 2019/2020

About Me: Michael Pradel

- **Since 9/2019: Full Professor at University of Stuttgart**
- **Before**
  - Studies at TU Dresden, ECP (Paris), and EPFL (Lausanne)
  - PhD at ETH Zurich, Switzerland
  - Postdoctoral researcher at UC Berkeley, USA
  - Assistant Professor at TU Darmstadt
  - Sabbatical at Facebook, Menlo Park, USA

About the Software Lab



- **My research group since 2014**
- **Focus: Tools and techniques for building reliable, efficient, and secure software**
  - Program testing and analysis
  - Machine learning, security
- **Thesis and job opportunities**

Plan for Today

- **Introduction**
  - What the course is about
  - Why it is interesting
  - How it can help you
- **Organization**
  - Classes
  - Project
- **Dynamic analysis**
  - Basics, examples, implementation

Program Testing & Analysis

What you probably know:

- **Manual testing or semi-automated testing:**
  JUnit, Selenium, etc.
- **Manual ”analysis” of programs:**
  Code inspection, debugging, etc.

Focus of this course:

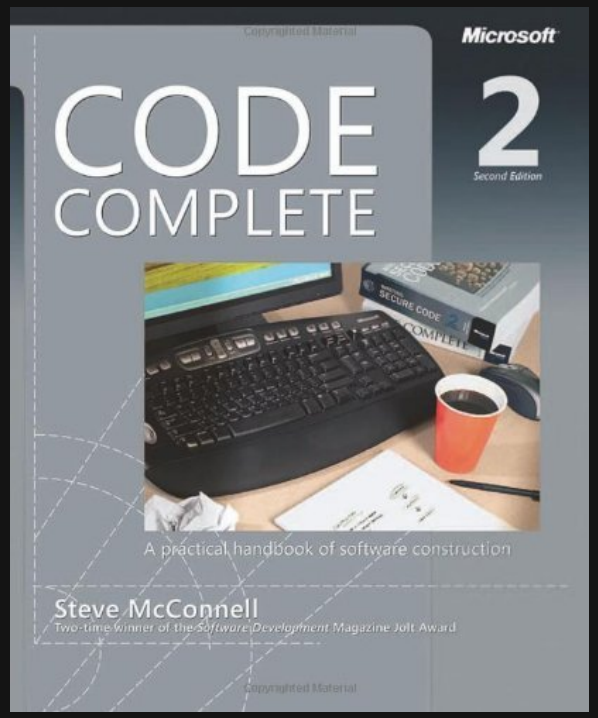
Automated testing and program analysis

Why Do We Need It?

- All software has bugs
- Bugs are hard to find
- Bugs cause serious harm

Why Do We Need It?

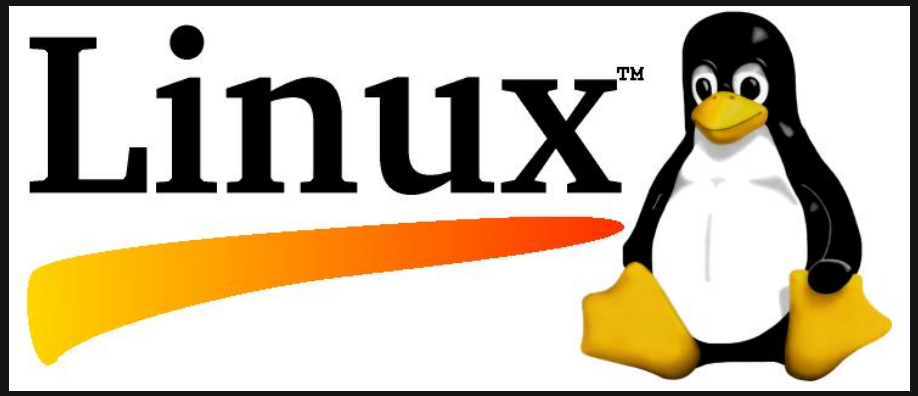
- All software has bugs
- Bugs are hard to find
- Bugs cause serious harm



0.5-25/KLoC in delivered software

Why Do We Need It?

- All software has bugs
- Bugs are hard to find
- Bugs cause serious harm



1.5 years to find a bug

[Palix2011]

Why Do We Need It?

- All software has bugs
- Bugs are hard to find
- Bugs cause serious harm

Ariane 5

Northeast blackout

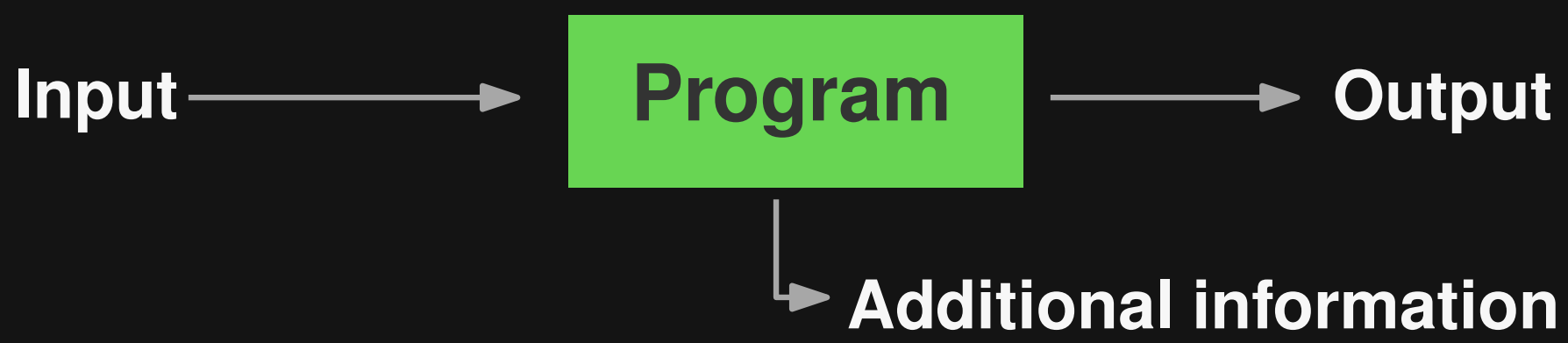
Therac-25

What is Program Analysis?

- **Automated analysis of program behavior, e.g., to**
  - find programming errors
  - optimize performance
  - find security vulnerabilities

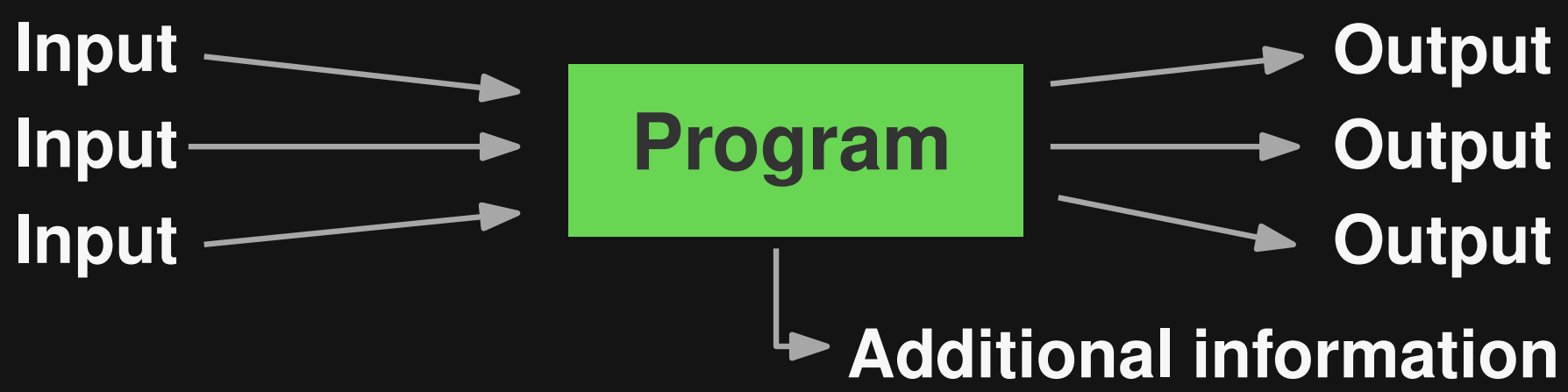
What is Program Analysis?

- **Automated analysis of program behavior, e.g., to**
  - find programming errors
  - optimize performance
  - find security vulnerabilities



What is Program Analysis?

- **Automated analysis of program behavior, e.g., to**
  - find programming errors
  - optimize performance
  - find security vulnerabilities



Static vs. Dynamic Analysis

Static

- Analyse source code, byte code, or binary
- Typically:
  - Consider all inputs
  - Overapproximate possible behavior

Dynamic

- Analyze program execution
- Typically:
  - Consider current input
  - Underapproximate possible behavior

Static vs. Dynamic Analysis

Static

- Analyse source code, byte code, or binary
- Typically:
  - Consider all inputs
  - Overapproximate possible behavior

E.g., compilers, lint-like tools

Dynamic

- Analyze program execution
- Typically:
  - Consider current input
  - Underapproximate possible behavior

E.g., automated testing, profilers

Example

```
// JavaScript
var r = Math.random(); // value in [0,1)
var out = "yes";
if (r < 0.5)
  out = "no";
if (r === 1)
  out = "maybe";
console.log(out);
```

What are the possible outputs?

Example

```
// JavaScript
var r = Math.random(); // value in [0,1)
var out = "yes";
if (r < 0.5)
  out = "no";
if (r === 1)
  out = "maybe"; // infeasible path
console.log(out);
```

Overapproximation: ”yes”, ”no”, ”maybe”

- Consider all paths (that are feasible based on limited knowledge)

Example

```
// JavaScript
var r = Math.random(); // value in [0,1)
var out = "yes";
if (r < 0.5)
  out = "no";
if (r === 1)
  out = "maybe"; // infeasible path
console.log(out);
```

Underapproximation: ”yes”

- Execute the program once

Example

```javascript
// JavaScript
var r = Math.random(); // value in [0,1)
var out = "yes";
if (r < 0.5)
  out = "no";
if (r === 1)
  out = "maybe"; // infeasible path
console.log(out);
```

Sound and complete: ”yes”, ”no”

- For this example: Can explore both feasible paths

Another Example

```
// JavaScript
var r = Math.random(); // value in [0,1)
var out = r * 2;
console.log(out);
```

Another Example

```
// JavaScript
var r = Math.random(); // value in [0,1)
var out = r * 2;
console.log(out);
```

Overapproximation: Any value

- Consider all paths (that are feasible based on limited knowledge about `random()`)

Another Example

```
// JavaScript
var r = Math.random(); // value in [0,1)
var out = r * 2;
console.log(out);
```

Underapproximation:

Some number in [0,2), e.g., 1.234

- Execute the program once

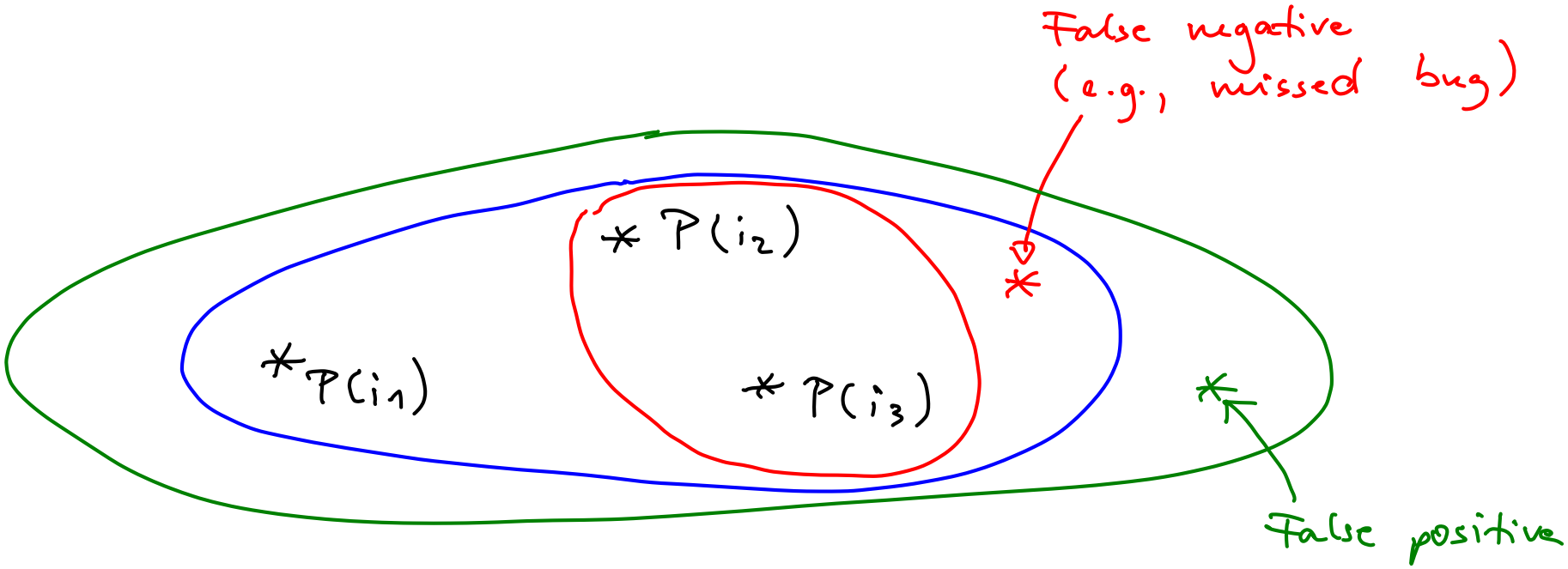
Another Example

```
// JavaScript
var r = Math.random(); // value in [0,1)
var out = r * 2;
console.log(out);
```

Sound and complete?

- Exploring all possible outputs:
  Practically impossible
- This is the case for most real-world programs

Program P, Input i, Behavior $P(i)$



All possible behaviors (what we want, ideally)

Underapproximation (most dyn. analyses)

Overapproximation (most static analyses)

Test Generation

- **Dynamic analysis:**
  Requires input to run the program
- **Test generation:**
  Creates inputs automatically
- **Examples**
  - Generate JUnit tests:
    Input = sequence of method calls
  - UI-level test generation:
    Input = sequence UI events
  - Fuzz-test a compiler: Input = program

How Does All This Help Me?

Improve the quality of your code

- Fewer bugs
- Better performance
- More secure software

Save time during manual testing

Become a better developer

- Get better understanding of program's behavior
- Avoid common pitfalls
- Learn to use and write tools

Plan for Today

- **Introduction**
  - What the course is about
  - Why it is interesting
  - How it can help you
- **Organization**
  - Classes
  - Project
- **Dynamic analysis**
  - Basics, examples, implementation

Organization

- **Six classes**
  - Introducing topic, tools, and project
  - Usually on Friday at 8am

    Exception: Wednesday, Oct 30, 8am
  - Check schedule:

    http://software-lab.org/teaching/winter2019/ppa/
- **Independent work on project**
  - Develop your own program analysis

Hands-on Classes

All classes from next week: Hands-on

- Bring your laptop
- Install the prerequisites
  - Option 1: Install Virtualbox and load a prepared virtual machine image
  - Option 2: Install all prerequisites on your own machine
  - See course page for details

Project

Design and implement your own program analysis

- Concolic execution of WebAssembly code
  - Form of test generation
  - WebAssembly: “Bytecode for the web”
- Based on Wasabi framework
  - Dynamic analysis framework
  - Best paper award at ASPLOS’19
  - See *wasabi.software-lab.org*

Project: Teams

- **Teams of two or three**
  - Larger team $\implies$ higher expectations
- **All teams have the same task**
  - Solutions must your own
  - Any form of plagiarism will be punished
  - Allowed to discuss the problem with your peers, but not to reuse any part of an existing solution

Project: Timeline

- Project published on November 22
- Milestone due on January 9
- Final project due on February 9
- Week of February 10 to 14: Presentation of projects

Ilias

Platform for discussions and announcements

- Please register for the course
- Use it for all questions related to the course
- Messages sent to all students go via Ilias

Link to Ilias course on
software-lab.org/teaching/winter2019/ppa/

Grading

Based on project only (no written exam)

- Quality of implementation
- Automated testing
- Final presentation

Grading

Based on project only (no written exam)

- Quality of implementation → **20%**
- Automated testing → **60%**
- Final presentation → **20%**

Plan for Today

- **Introduction**
  - What the course is about
  - Why it is interesting
  - How it can help you
- **Organization**
  - Classes
  - Project
- **Dynamic analysis** ←
  - Basics, examples, implementation

Executions Paths

- **One program, many execution paths**
  - Path: Sequence of statements executed during an execution
- **One representation to reason about paths: Control flow graph**
  - Nodes: Basic blocks (sequence of statements that are always executed together)
  - Edges: Possible flow of control

Example

Given an array of numbers, sort the array in-place in ascending order.

```
function sort(a) {
  if (!a || a.length < 2)
    return; // array is trivially sorted
  for (var i = 0; i < a.length - 1; i++) {
    if (a[i] < a[i+1]) {
      break;
    }
  }
  if (i >= a.length - 1)
    return; // array is sorted
  qsort(a); // use quicksort to sort array
}
```

Example

Given an array of numbers, sort the array in-place in ascending order.

```
function sort(a) {
  if (!a || a.length < 2)
    return; // array is trivially sorted
  for (var i = 0; i < a.length - 1; i++) {
    if (a[i] < a[i+1]) {
      break;
    }
  }
  if (i >= a.length - 1)
    return; // array is sorted
  qsort(a); // use quicksort to sort array
}
```

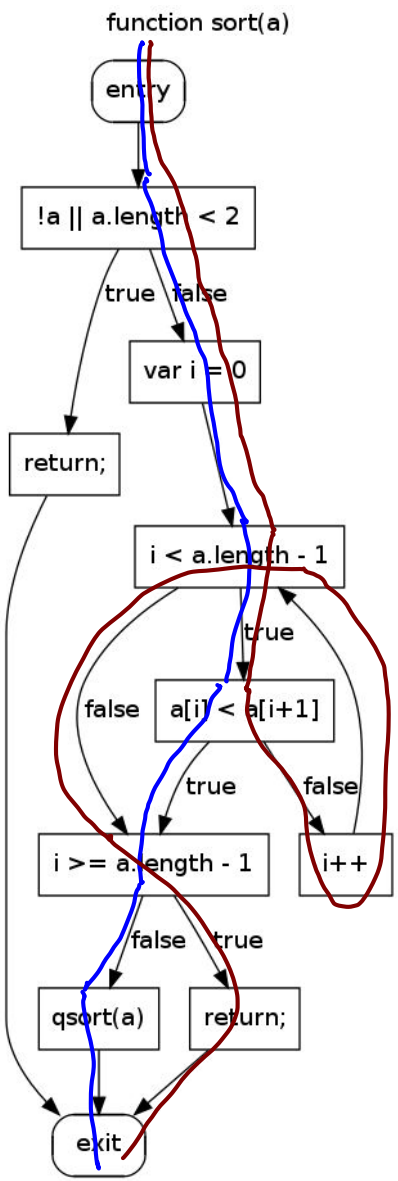
Error: Check for sortedness should use >

Control flow graph

entry

```
function sort(a) {
  if (!a || a.length < 2)
    return; // array is trivially sorted
  for (var i = 0; i < a.length - 1; i++) {
    if (a[i] < a[i+1]) {
      break;
    }
  }
  if (i >= a.length - 1)
    return;
  qsort(a); // use quicksort to sort array
}
```

exit



Input: a = [3, 7, 5]

a = [7, 5]

Execution Trace

- **Information extracted by a dynamic analysis during an execution**
- **For specific operations, log some details**
  - E.g., for each call, log caller and callee
- **Store information about program state**
  - E.g., for each variable, store code location where last written

```
function f() {
        g()
        h()
}
function g() {
        h()
}
function h() {
}
f()
```

Trace:

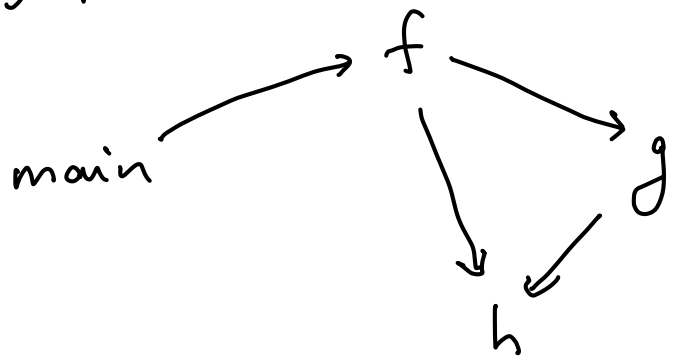
main calls f

f calls g

g calls h

f calls h

Call graph:



Call Graph Analysis

- **Call graph: Functions and their call relations**
  - Nodes: Functions in the programs
  - Edges: Call relations
    - $f_1 \rightarrow f_2$ means f_1 calls f_2
- **Dynamic analysis that tracks all calls: Dynamic call graph analysis**

Implementing Dynamic Analyses

Option 1: Instrument program under analysis

- Add instructions between “normal” instructions
  - Log into execution trace
  - Update state of dynamic analysis

Option 2: Modified runtime environment

- Add instructions to environment that executes the program under test
  - E.g., for interpreted code, add instructions before and after handling “normal” instructions

```
function f() {
    if (userInput()) {
        log("f calls g")
        g()
    } else {
        log("f calls h")
        h()
    }
}
⋮
f()
```

Instrumentation code for dynamic call graph analysis

→ Different dynamic call graph depending on user input

More Complex Dynamic Analyses

- **Various kinds of analyses**
  - Active field of research
- **Two classical examples**
  - Slicing: Find statements relevant for reaching a particular program state
  - Taint analysis: Check if values from “source” reach a “sink”

Dynamic Slicing

- **Extract an executable subset of a program that affects variable v at a program location n**
  - Slicing criterion = program location n and variable v
- **An observer focusing on the slicing criterion cannot distinguish a run of the program from a run of the slice**

Example

```
var x = readInput();
if (x < 0) {
  y = x + 1;
  z = x + 2;
} else {
  if (x === 0) {
    y = x + 3;
    z = x + 4;
  } else {
    y = x + 5;
    z = x + 6;
  }
}
console.log(y);
console.log(z);
```

Example

```
var x = readInput();
if (x < 0) {
  y = x + 1;
  z = x + 2;
} else {
  if (x === 0) {
    y = x + 3;
    z = x + 4;
  } else {
    y = x + 5;
    z = x + 6;
  }
}
console.log(y);
console.log(z);
```

Which parts of the code influence `y` at this location?

Program Dependence Graph

Directed graph representing the data and control dependences between statements

- Nodes:
  - Statements
  - Predicate expressions
- Edges:
  - Data flow dependences: One edge for each definition-use pair
  - Control flow dependences

Dynamic Slice

- **Given: Execution history**
  - Sequence of PDG nodes that are executed
- **Slice for statement n and variable v:**
  - Keep PDG nodes only if there are in history
  - Slice = All Statements from which n is reachable (i.e., all statements on which n depends)

Based on Agrawal & Horgan, PLDI 1990

See paper for a more sophisticated approach

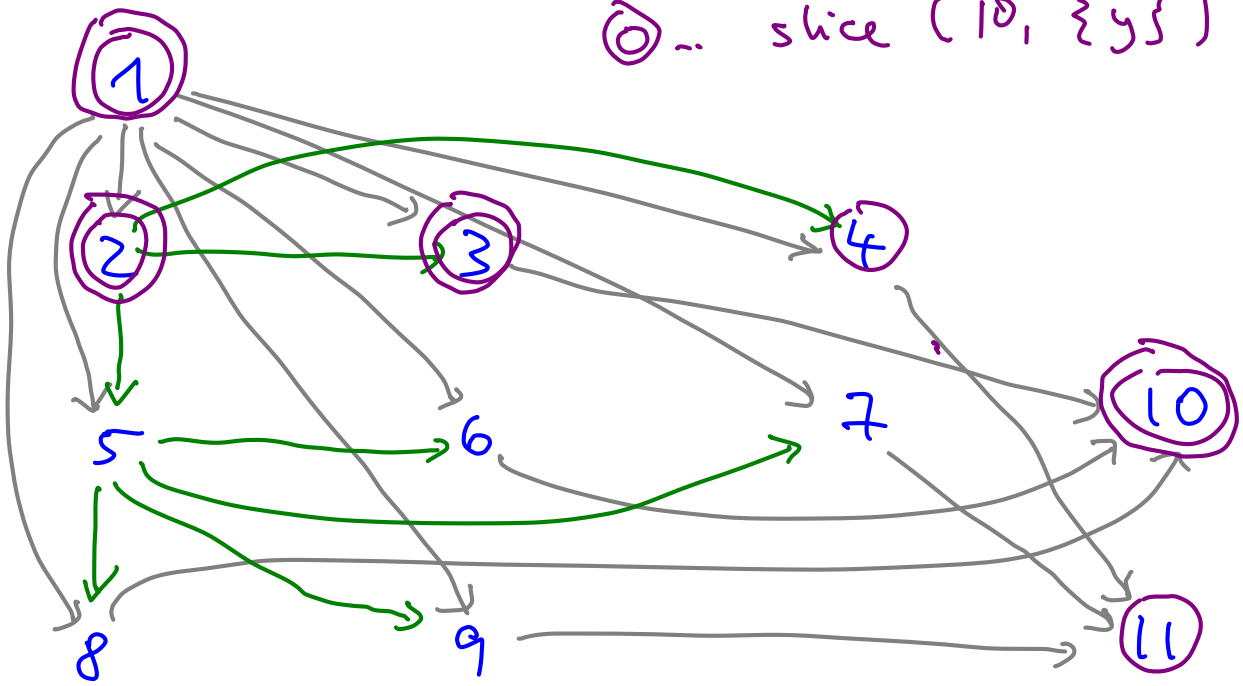
```
1  var x = readInput()
2  if (x < 0) {
3      y = x + 1
4      z = x + 2
   } else {
5      if (x == 0) {
6          y = x + 3
7          z = x + 4
       } else {
8          y = x + 5
9          z = x + 6
       }
   }
10 console.log(y)
11 console.log(z)
```

Input: −1

History: 1, 2, 3, 4, 10, 11

○ .. executed

◎ .. slice (10, {y})



→ data

→ control

Plan for Today

- **Introduction**
  - What the course is about
  - Why it is interesting
  - How it can help you
- **Organization**
  - Classes
  - Project
- **Foundations**
  - Dynamic analysis

Reminder:

1) Bring laptop with prerequisites

2) Next class: Wednesday, Oct 30, 8am, room 0.463

✔