Lecture 1: Introduction to Program Analysis

17-355/17-665/17-819: Program Analysis

Rohan Padhye

Jan 18, 2022

* Course materials developed with Jonathan Aldrich and Claire Le Goues

Introductions

Prof. Rohan Padhye

TA Bella Laybourn

My Background

IBM Research

Berkeley
UNIVERSITY OF CALIFORNIA

SAMSUNG
RESEARCH AMERICA

Carnegie Mellon University

Microsoft Research

- Involved with program analysis for ~10 years.
- PhD from UC Berkeley, Masters from IIT Bombay (India)
- Worked at IBM Research, Microsoft Research, and Samsung Research America
- Advising PhD students at CMU’s Institute for Software Research
- Developed tools for improving developer productivity, finding input-validation software bugs, identifying security vulnerabilities in mobile systems, discovering concurrency issues in distributed systems, etc.
- Contributed to research on fuzz testing, static interprocedural analysis, dynamic performance analysis, etc.

Learning objectives

- Provide a high level definition of program analysis and give examples of why it is useful.
- Sketch the explanation for why all analyses must approximate.
- Understand the course mechanics, and be motivated to read the syllabus.
- Describe the function of an AST and outline the principles behind AST walkers for simple bug-finding analyses.
- Recognize the basic WHILE demonstration language and translate between WHILE and While3Addr.

What is this course about?

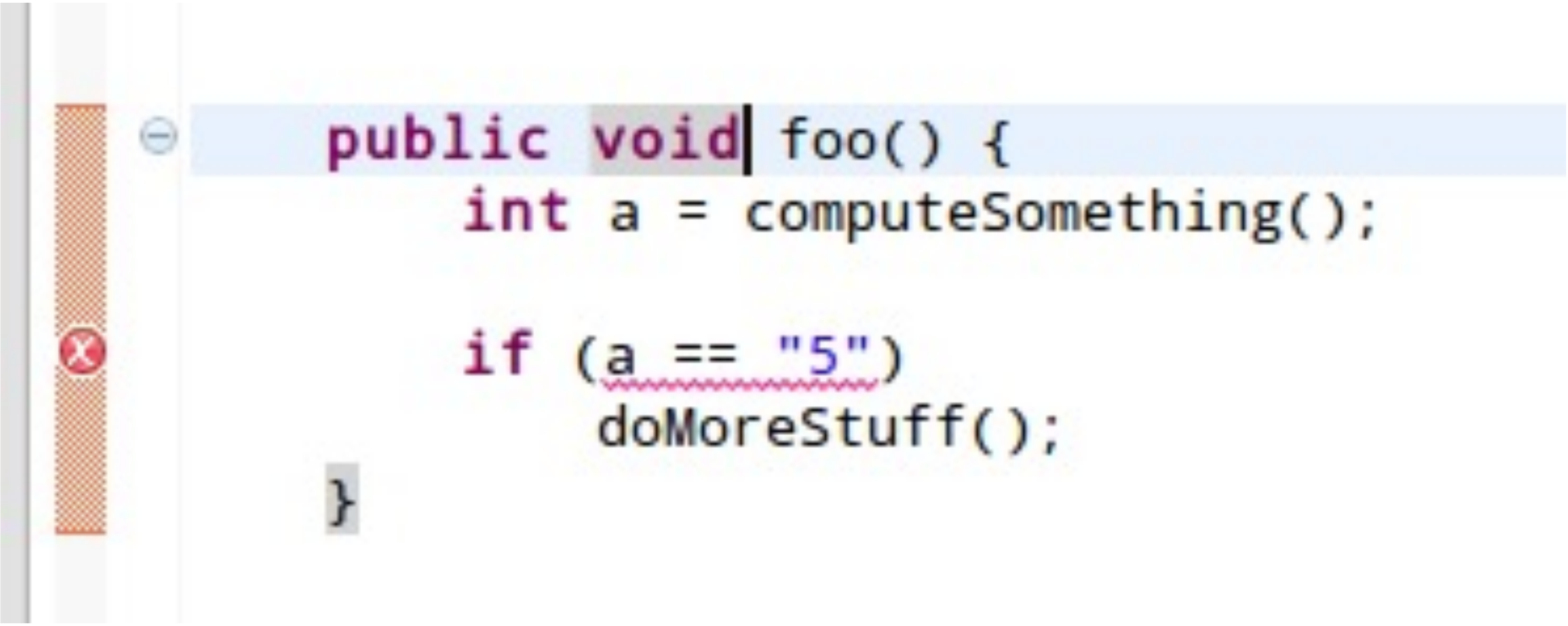
- Program analysis is the systematic examination of a program to determine its properties.
- From 30,000 feet, this requires:
  - Precise program representations
  - Tractable, systematic ways to reason over those representations.
- We will learn:
  - How to unambiguously define the meaning of a program, and a programming language.
  - How to prove theorems about the behavior of particular programs.
  - How to use, build, and extend tools that do the above, automatically.

Why might you care?

Program analysis, and the skills that underlie it, have implications for:

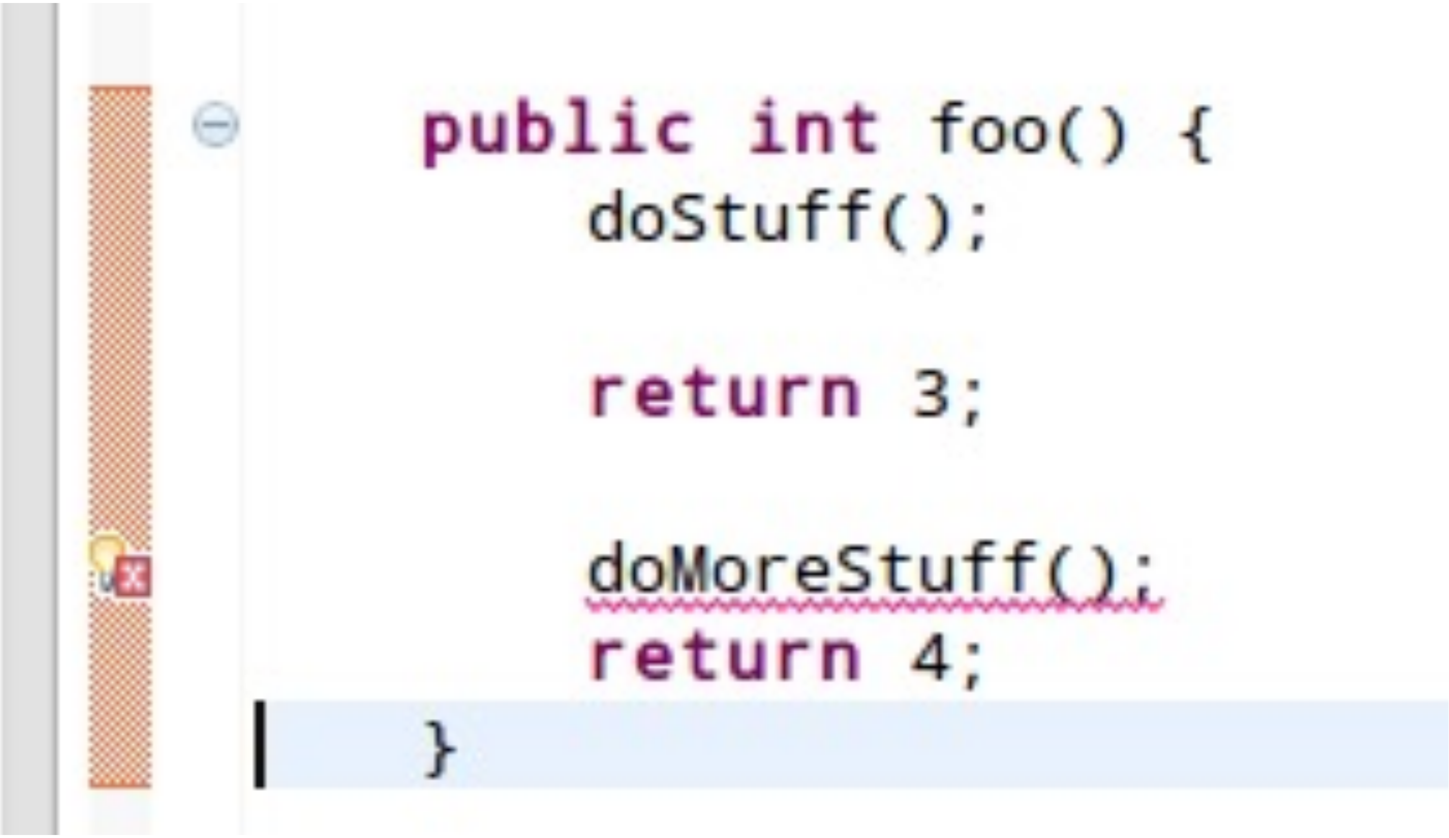
- Automatic bug finding
- Language design and implementation (compilers, VMs)
- Program transformation (refactoring, optimization, repair)
- Program synthesis

You’ve seen it before



```
public void foo() {
    int a = computeSomething();

    if (a == "5")
        doMoreStuff();
}
```

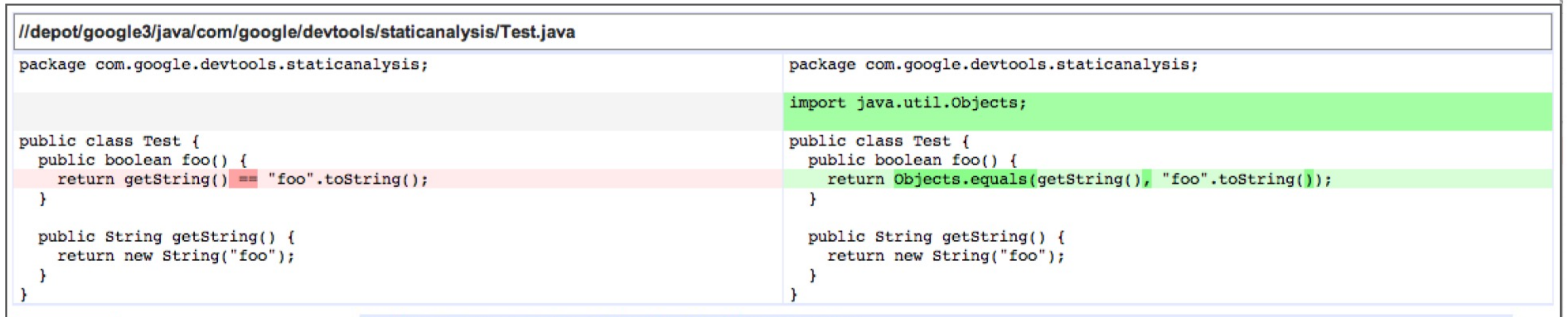
You’ve seen it before



```
public int foo() {
    doStuff();

    return 3;

    doMoreStuff();
    return 4;
}
```

Lots of tools available

Lint



//depot/google3/java/com/google/devtools/staticanalysis/Test.java

```
package com.google.devtools.staticanalysis;


public class Test {
  public boolean foo() {
    return getString() == "foo".toString();
  }

  public String getString() {
    return new String("foo");
  }
}
```

```
package com.google.devtools.staticanalysis;

import java.util.Objects;

public class Test {
  public boolean foo() {
    return Objects.equals(getString(), "foo".toString());
  }

  public String getString() {
    return new String("foo");
  }
}
```

```
package com.google.devtools.staticanalysis;

public class Test {
```

▾ Lint — Missing a Javadoc comment.
Java
1:02 AM, Aug 21

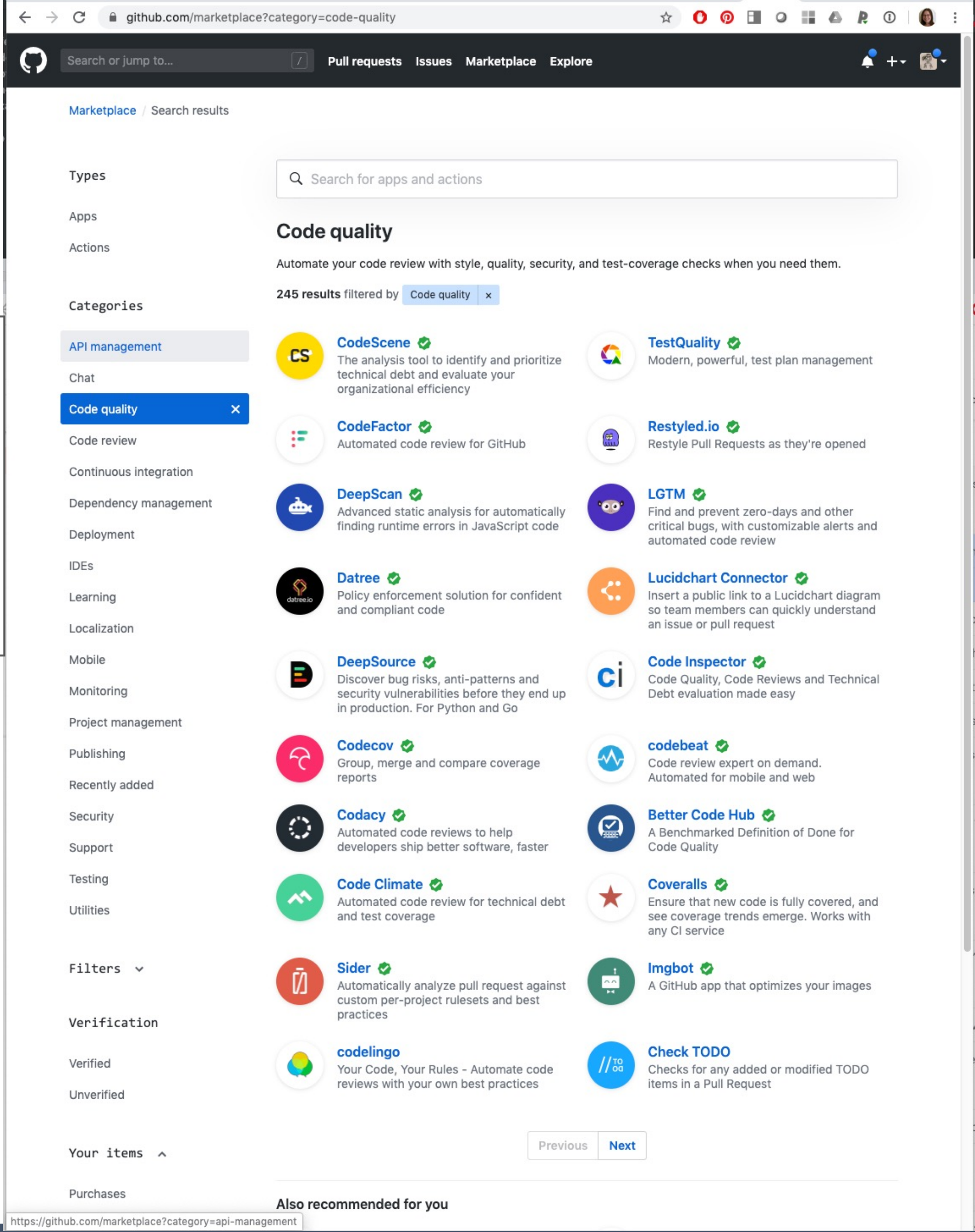
```
  public boolean foo() {
    return getString() == "foo".toString();
```

▾ ErrorProne — String comparison using reference equality instead of value equality (see http://code.google.com/p/error-prone/wiki/StringEquality)
StringEquality
1:03 AM, Aug 21

Suggested fix attached: show

```
  }

  public String getString() {
    return new String("foo");
  }
}
```

ErrorProne



Code quality

Automate your code review with style, quality, security, and test-coverage checks when you need them.

245 results filtered by Code quality ×

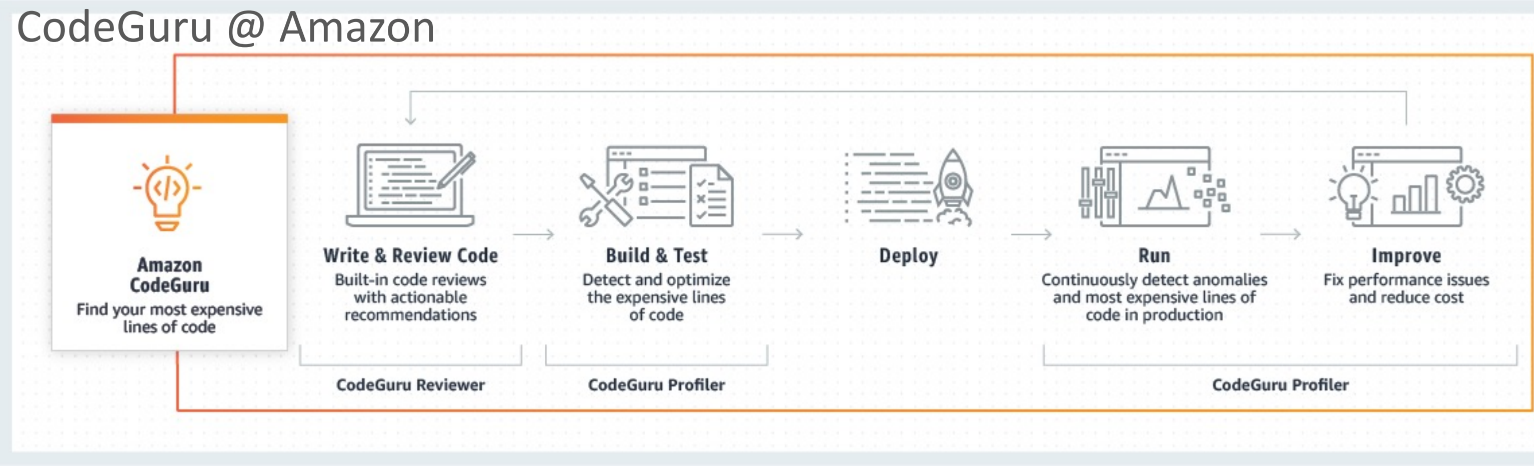
CodeScene The analysis tool to identify and prioritize technical debt and evaluate your organizational efficiency	**TestQuality** Modern, powerful, test plan management
CodeFactor Automated code review for GitHub	**Restyled.io** Restyle Pull Requests as they're opened
DeepScan Advanced static analysis for automatically finding runtime errors in JavaScript code	**LGTM** Find and prevent zero-days and other critical bugs, with customizable alerts and automated code review
Datree Policy enforcement solution for confident and compliant code	**Lucidchart Connector** Insert a public link to a Lucidchart diagram so team members can quickly understand an issue or pull request
DeepSource Discover bug risks, anti-patterns and security vulnerabilities before they end up in production. For Python and Go	**Code Inspector** Code Quality, Code Reviews and Technical Debt evaluation made easy
Codecov Group, merge and compare coverage reports	**codebeat** Code review expert on demand. Automated for mobile and web
Codacy Automated code reviews to help developers ship better software, faster	**Better Code Hub** A Benchmarked Definition of Done for Code Quality
Code Climate Automated code review for technical debt and test coverage	**Coveralls** Ensure that new code is fully covered, and see coverage trends emerge. Works with any CI service
Sider Automatically analyze pull request against custom per-project rulesets and best practices	**Imgbot** A GitHub app that optimizes your images
codelingo Your Code, Your Rules - Automate code reviews with your own best practices	**Check TODO** Checks for any added or modified TODO items in a Pull Request

Also recommended for you

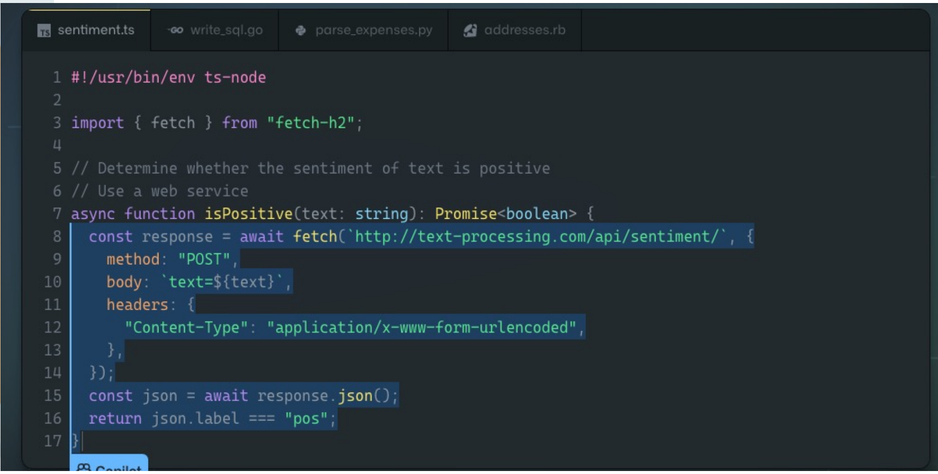
https://github.com/marketplace?category=code-quality

Advanced examples from industry

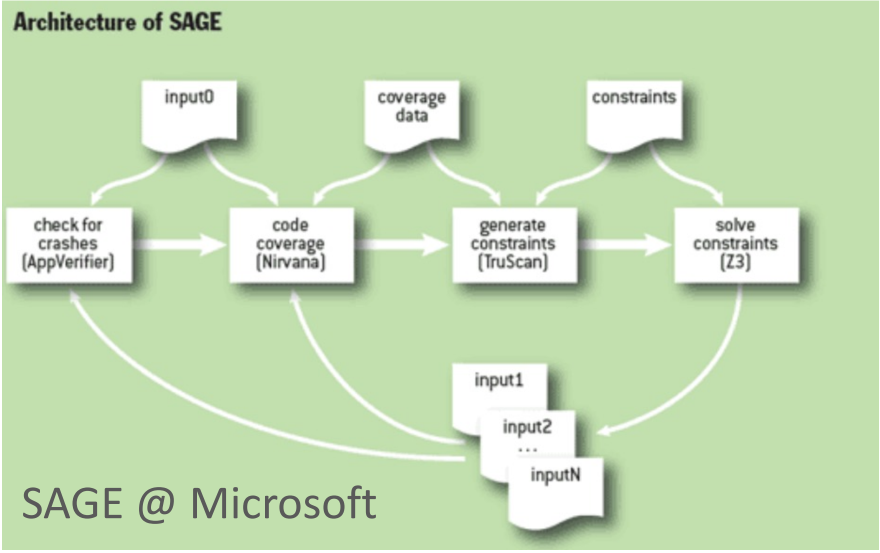
CodeGuru @ Amazon



GitHub CoPilot

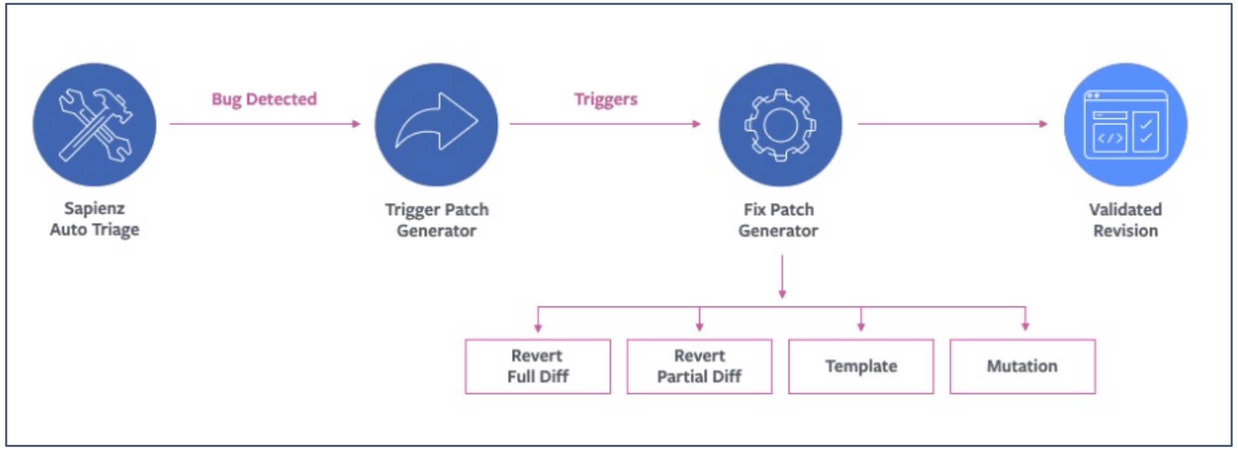


```
#!/usr/bin/env ts-node

import { fetch } from "fetch-h2";

// Determine whether the sentiment of text is positive
// Use a web service
async function isPositive(text: string): Promise<boolean> {
  const response = await fetch(`http://text-processing.com/api/sentiment/`, {
    method: "POST",
    body: `text=${text}`,
    headers: {
      "Content-Type": "application/x-www-form-urlencoded",
    },
  });
  const json = await response.json();
  return json.label === "pos";
}
```



SAGE @ Microsoft

Sapienz and SapFix @ Facebook



Common types of issues found using automated program analysis

- Defects that result from inconsistently following simple design rules.
    - **Security:** Buffer overruns, improperly validated input.
    - **Memory safety:** Null dereference, uninitialized data.
    - **Resource leaks:** Memory, OS resources.
    - **API Protocols:** Device drivers; real time libraries; GUI frameworks.
    - **Exceptions:** Arithmetic/library/user-defined
    - **Encapsulation:** Accessing internal data, calling private functions.
    - **Data races:** Two threads access the same data without synchronization

Key: check compliance to simple, mechanical design rules

IS THERE A BUG IN THIS CODE?

```c
/* from Linux 2.3.99 drivers/block/raid5.c */
static struct buffer_head *
get_free_buffer(struct stripe_head * sh,
                int b_size) {
  struct buffer_head *bh;
  unsigned long flags;
  save_flags(flags);
  cli(); // disables interrupts
  if ((bh = sh->buffer_pool) == NULL)
    return NULL;
  sh->buffer_pool = bh -> b_next;
  bh->b_size = b_size;
  restore_flags(flags); // re-enables interrupts
  return bh;
}
```

Part of the spec:
Interrupts should not be disabled upon function return

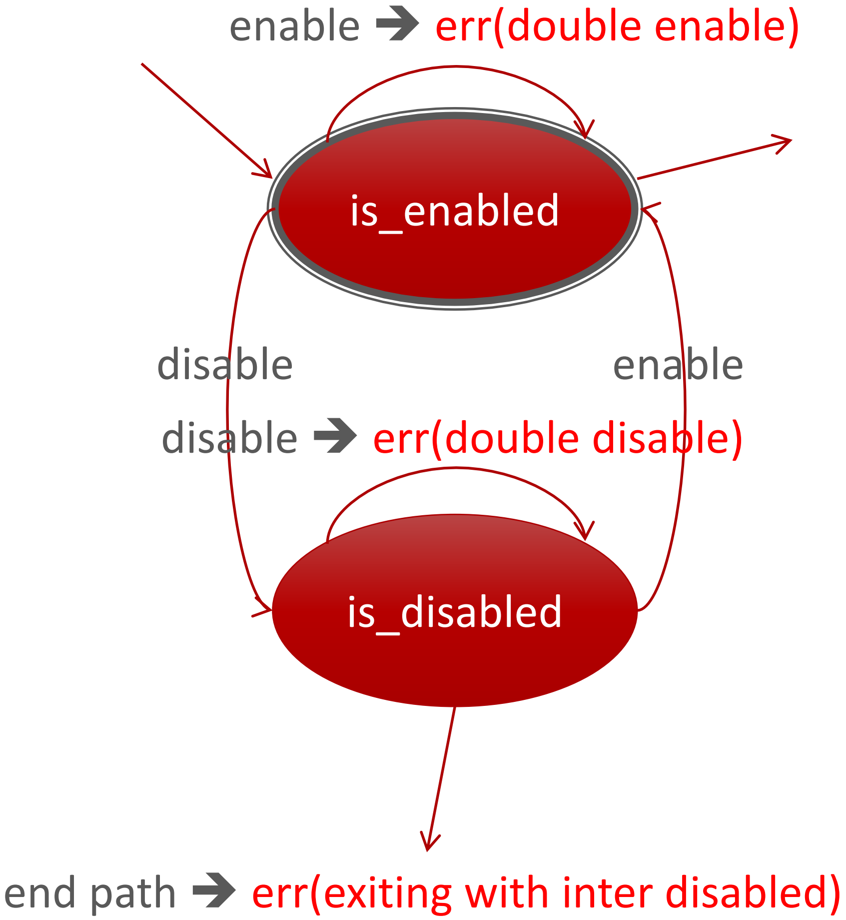
Example from Engler et al., *Checking system rules Using System-Specific, Programmer-Written Compiler Extensions*, OSDI '000

```c
/* from Linux 2.3.99 drivers/block/raid5.c */
static struct buffer_head *
get_free_buffer(struct stripe_head * sh,
                int b_size) {
  struct buffer_head *bh;
  unsigned long flags;
  save_flags(flags);
  cli(); // disables interrupts
  if ((bh = sh->buffer_pool) == NULL)
    return NULL;
  sh->buffer_pool = bh -> b_next;
  bh->b_size = b_size;
  restore_flags(flags); // re-enables interrupts
  return bh;
}
```

ERROR: function returns with interrupts disabled!

Example from Engler et al., *Checking system rules Using System-Specific, Programmer-Written Compiler Extensions*, OSDI '000

Abstract Model



enable ➔ err(double enable)

is_enabled

disable

enable

disable ➔ err(double disable)

is_disabled

end path ➔ err(exiting with inter disabled)

```c
/* from Linux 2.3.99 drivers/block/raid5.c */
static struct buffer_head *
get_free_buffer(struct stripe_head * sh,
                int b_size) {
  struct buffer_head *bh;
  unsigned long flags;
  save_flags(flags);
  cli(); // disables interrupts
  if ((bh = sh->buffer_pool) == NULL)
    return NULL;
  sh->buffer_pool = bh -> b_next;
  bh->b_size = b_size;
  restore_flags(flags); // re-enables interrupts
  return bh;
}
```

Initial state: is_enabled

Example from Engler et al., *Checking system rules Using System-Specific, Programmer-Written Compiler Extensions*, OSDI ‘000

```
/* from Linux 2.3.99 drivers/block/raid5.c */
static struct buffer_head *
get_free_buffer(struct stripe_head * sh,
                int b_size) {
  struct buffer_head *bh;
  unsigned long flags;
  save_flags(flags);
  cli(); // disables interrupts
  if ((bh = sh->buffer_pool) == NULL)
    return NULL;
  sh->buffer_pool = bh -> b_next;
  bh->b_size = b_size;
  restore_flags(flags); // re-enables interrupts
  return bh;
}
```

Transition to: is_disabled

Example from Engler et al., *Checking system rules Using System-Specific, Programmer-Written Compiler Extensions*, OSDI ‘000

```c
/* from Linux 2.3.99 drivers/block/raid5.c */
static struct buffer_head *
get_free_buffer(struct stripe_head * sh,
                int b_size) {
  struct buffer_head *bh;
  unsigned long flags;
  save_flags(flags);
  cli(); // disables interrupts
  if ((bh = sh->buffer_pool) == NULL)
    return NULL;
  sh->buffer_pool = bh -> b_next;
  bh->b_size = b_size;
  restore_flags(flags); // re-enables interrupts
  return bh;
}
```

Final state: is_disabled

Example from Engler et al., *Checking system rules Using System-Specific, Programmer-Written Compiler Extensions*, OSDI '000

```c
/* from Linux 2.3.99 drivers/block/raid5.c */
static struct buffer_head *
get_free_buffer(struct stripe_head * sh,
                int b_size) {
  struct buffer_head *bh;
  unsigned long flags;
  save_flags(flags);
  cli(); // disables interrupts
  if ((bh = sh->buffer_pool) == NULL)
     return NULL;
   sh->buffer_pool = bh -> b_next;
   bh->b_size = b_size;
   restore_flags(flags); // re-enables interrupts
   return bh;
}
```

Transition to: is_enabled

Example from Engler et al., *Checking system rules Using System-Specific, Programmer-Written Compiler Extensions*, OSDI '000

```
/* from Linux 2.3.99 drivers/block/raid5.c */
static struct buffer_head *
get_free_buffer(struct stripe_head * sh,
                int b_size) {
  struct buffer_head *bh;
  unsigned long flags;
  save_flags(flags);
  cli(); // disables interrupts
  if ((bh = sh->buffer_pool) == NULL)
    return NULL;
  sh->buffer_pool = bh -> b_next
  bh->b_size = b_size;
  restore_flags(flags); // re-enables interrupts
  return bh;
}
```

Final state: is_enabled

Example from Engler et al., *Checking system rules Using System-Specific, Programmer-Written Compiler Extensions*, OSDI '000

Behavior of interest…

- Is on uncommon execution paths.
    - Hard to exercise when testing.
- Executing (or analyzing) all paths is infeasible
- **Instead: (abstractly) check the entire possible state space of the program.**

What is this course about?

- Program analysis is *the systematic examination of a program to determine its properties*.
- From 30,000 feet, this requires:
  - Precise program representations
  - Tractable, systematic ways to reason over those representations.
- We will learn:
  - How to unambiguously define the meaning of a program, and a programming language.
  - How to prove theorems about the behavior of particular programs.
  - How to use, build, and extend tools that do the above, automatically.

What is this course about?

- Program analysis is *the systematic examination of a program to determine its properties*.
- Principal techniques:
  - **Dynamic:**
    - **Testing:** Direct execution of code on test data in a controlled environment.
    - **Analysis:** Tools extracting data from test runs.
  - **Static:**
    - **Inspection:** Human evaluation of code, design documents (specs and models), modifications.
    - **Analysis:** Tools reasoning about the program without executing it.
  - …and their combination.

The Bad News: Rice's Theorem

> "Any nontrivial property about the language recognized by a Turing machine is undecidable.“

Henry Gordon Rice, 1953

Proof by contradiction (sketch)

Assume that you have a function that can determine if a program *p* has some nontrivial property (like `divides_by_zero`):

```
int silly(program p, input i) {
  p(i);
  return 5/0;
}
bool halts(program p, input i) {
  return divides_by_zero(`silly(p,i)`);
}
```

	Error exists	**No error exists**
Error Reported	True positive (correct analysis result)	False positive
No Error Reported	False negative	True negative (correct analysis result)

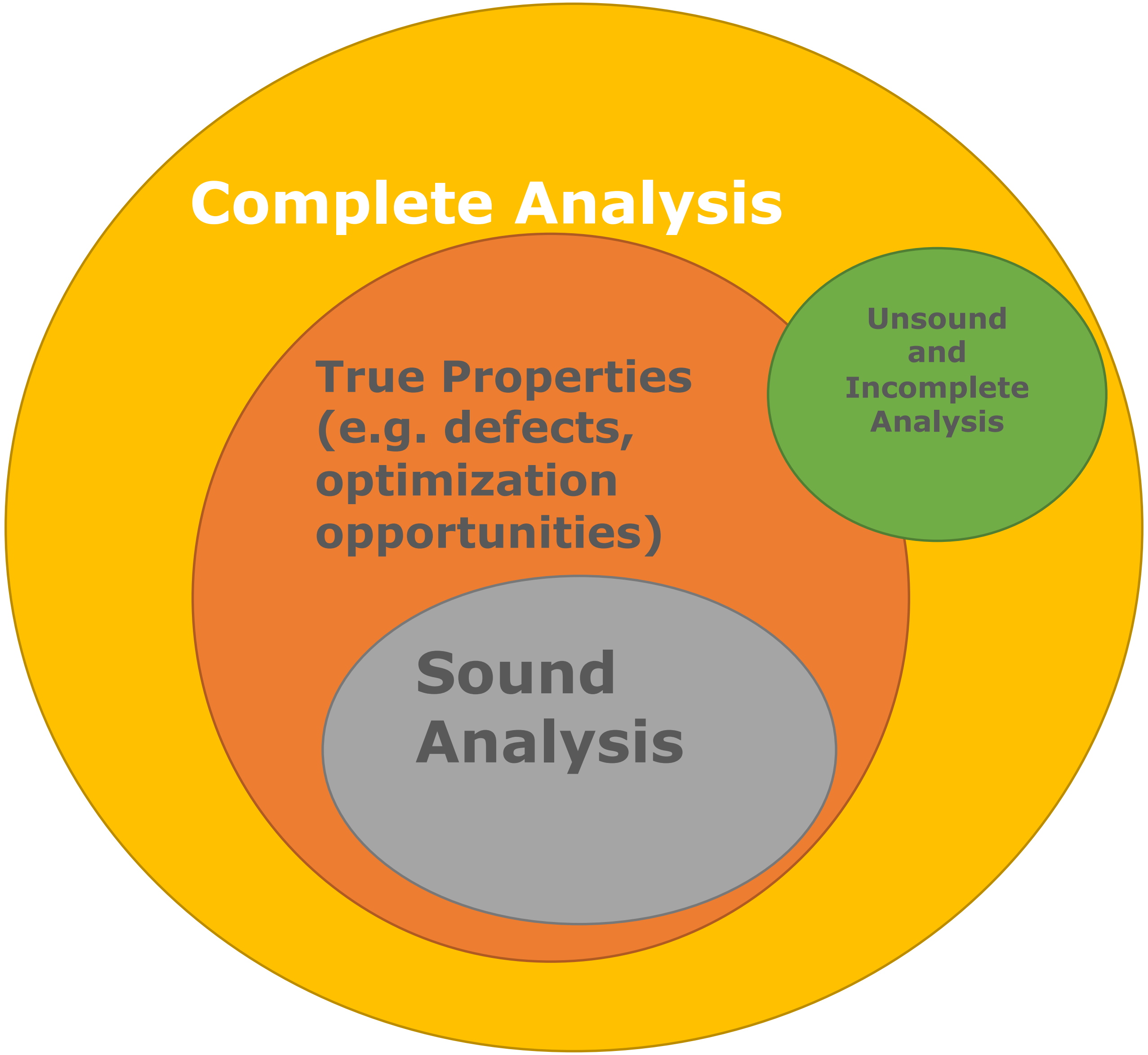
Over-approximate analysis:
- reports all potential defects
- -> no false negatives
- -> subject to false positives

Under-approximate analysis:
- every reported defect is an actual defect
- -> no false positives
- -> subject to false negatives

Soundness and Completeness

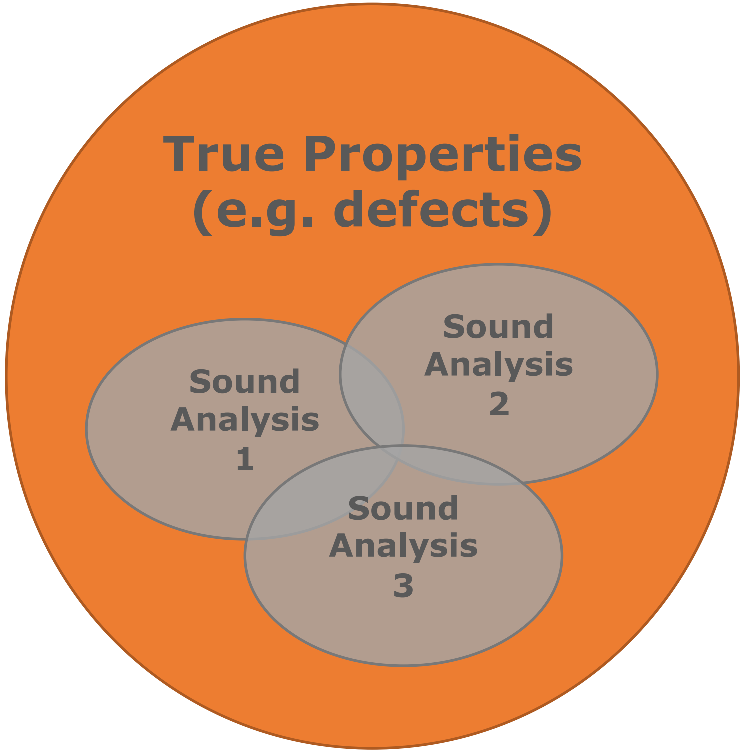
- An analysis is “sound” if every claim it makes is true
- An analysis is “complete” if it makes every true claim

- Soundness/Completeness correspond to under/over-approximation depending on context.
  - E.g. compilers and verification tools treat “soundness” as over-approximation since they make claims over all possible inputs
  - E.g. code quality tools often treat “sound” analyses as under-approximation because they make claims about existence of bugs


Complete Analysis
True Properties (e.g. defects, optimization opportunities)
Unsound and Incomplete Analysis
Sound Analysis

Soundness and Completeness Tradeoffs

- Sound + Complete is impossible in general (Rice's theorem)
- Most practical tools attempt to be either sound or complete for some specific application, using approximation
- Multiple classes of sound/complete techniques may exist, with trade-offs for accuracy and performance.
- Program analysis is a rich field because of the constant and never-ending battle to balance these trade-offs with ever-increasing software complexity



Course topics

- Program representation
- Abstract interpretation: Use abstraction to reason about possible program behavior.
  - Operational semantics.
  - Dataflow Analysis
  - Termination, complexity
  - Widening, collecting
  - Interprocedural analysis
  - Pointer analysis
  - Control flow analysis
- Hoare-style verification: Make logical arguments about program behavior.
  - Axiomatic semantics
- Model checking (briefly) : reason about all possible program states.
  - Take 15-414 if you want the full treatment!
- SAT/SMT solvers
- Symbolic execution: test all possible executions paths simultaneously.
  - Concolic execution
  - Test generation
- Grey-box analysis for fuzz testing
- Dynamic analysis for race detection
- Program synthesis
- Program repair
- We will basically *not* cover types.

Fundamental concepts

- Abstraction
    - Elide details of a specific implementation.
    - Capture semantically relevant details; ignore the rest.
- The importance of semantics.
    - We prove things about analyses with respect to the semantics of the underlying language.
- Program proofs as inductive invariants.
- Implementation
    - You do not understand analysis until you have written several.

Course mechanics

What to expect

- Beautiful and elegant theory (15-251 is a soft pre-req)
    - Mostly discrete mathematics, symbolic reasoning, inductive proofs
    - This is traditionally a “white-board” course [using slides while we’re on Zoom]
- Build awesome tools
    - Engineering of program analyses, compilers, and bug finding tools make great use of many fundamental ideas from computer science and software engineering
- New way to think about programs (15-150 or 15-214 soft pre-reqs)
    - Representations, control/data-flow, input state space
- Appreciate the limits and achievements in the space
    - What tools are *impossible* to build?
    - What tools are *impressive* that they exist at all?
    - When is it appropriate to use a particular analysis tool versus another?
    - How to interpret the results of a program analysis tool?

When/what

- Lectures 2x week (T,Th – in GHC 4101 from Feb 1; now on Zoom).
  - Active learning exercise(s) in every class
  - Lecture notes for review --- get latest PDF from website
- Recitation 1x week (Fr – in MI 348 from Feb 1; now on Zoom).
  - Lab-like, very helpful for homework.
  - Be ready to work
- Homework, midterm exam, project.
- There is an optional physical textbook. (“PPA”)

Communication

- Course website: https://cmu-program-analysis.github.io
- We also use Canvas, Piazza, Gradescope (see website for links)
  - Canvas: In-class exercises, some assignments, Zoom links, grades tally
  - Gradescope: For written assignments
  - Piazza: Please use public posts for any course related questions as much as possible, unless the matter is sensitive. Feel free to respond to other posts and engage in discussion.
- We have office hours! Or, by appointment.

“How do I get an A?”

- 10% in-class participation and exercises
- 50% homework assignments
  - Both written (proof-y) and coding (implementation-y).
  - First one (mostly coding) to be released by Friday!
- 20% midterm exam
- 20% final project
  - There will be some options here.
- No final exam; exam slot used for project presentations.
- We have late days and a late day policy; read the syllabus.
  - *tl;dr*: 3 late days per HW, with 5 total late days before penalties kick in

Slight variations in expectations

- If you’re taking the undergraduate version of the course (17-355)
  - Recitation attendance is expected and part of participation grade.
- If you’re taking the graduate version of the course (17-665/819)
  - Recitation attendance is encouraged.
  - Higher bar for final course project.
    - Master’s students: Expected to engage with large codebases (either frameworks or targets)
    - PhD students: Expected to engage with research questions
- You are welcome to move up your expectations to be assessed differently (email me)

CMU can be a pretty intense place.

- A 12-credit course is expected to take ~12 hours a week.
- We aim to provide a rigorous but tractable course.
  - More frequent assignments rather than big monoliths
  - Midterm exam to cover core material from first half of course
- Please let us know how much time the class is *actually taking*.
  - We have no way of knowing if you have three midterms in one week.
  - Sometimes, we misjudge assignment difficulty.
- If it’s 2 am and you’re panicking…put the homework down, send us an email, and go to bed.

Let’s get started

What is this course about?

- Program analysis is *the systematic examination of a program to determine its properties*.
- From 30,000 feet, this requires:
  - Precise program representations
  - Tractable, systematic ways to reason over those representations.
- We will learn:
  - How to unambiguously define the meaning of a program, and a programming language.
  - How to prove theorems about the behavior of particular programs.
  - How to use, build, and extend tools that do the above, automatically.

Our first representation: Abstract Syntax

- A tree representation of source code based on the language grammar.
- **Concrete syntax**: The rules by which programs can be expressed as strings of characters
  - E.g. “if (x * (a + b)) { foo(a); }”
  - Use finite automata and context-free grammars, automatic lexer/parser generators
- **Abstract syntax**: a subset of the parse tree of the program.
  - Only care about statements, expressions and their relationship with constituent operands.
  - Don’t care about parenthesis, semicolons, keywords, etc.
- (The intuition is fine for this course; take compilers if you want to learn how to parse for real.)

The WHILE language – Example program

```
y := x;
z := 1;
if y > 0 then
  while y > 1 do
    z := z * y;
    y := y – 1
else
  skip
```

- Sample program computes z = *x!* using *y* as a temp variable.
- WHILE uses assignment statements, if-then-else, while loops.
- All vars are integers.
- Expressions only arithmetic (for vars) or relational (for conditions).
- No I/O statements. Inputs and outputs are implicit.
  - Later on, we may use extensions with explicit `read x` and `print x`.

WHILE abstract syntax

Concrete syntax is similar, but adds things like (parentheses) for disambiguation during parsing

- Categories:

S ∈ **Stmt**	statements
a ∈ **Aexp**	arithmetic expressions
x, y ∈ **Var**	variables
n ∈ **Num**	number literals
P ∈ **BExp**	boolean predicates
l ∈ **labels**	statement addresses (line numbers)

- Syntax:
  - $S \;::=\; x := a \mid \texttt{skip} \mid S_1 \;;\; S_2$
    $\mid \texttt{if } P \texttt{ then } S_1 \texttt{ else } S_2 \mid \texttt{while } P \texttt{ do } S$
  - $a \;::=\; x \mid n \mid a_1 \; op_a \; a_2$
  - $op_a \;::=\; + \mid - \mid * \mid / \mid \ldots$
  - $P \;::=\; \texttt{true} \mid \texttt{false} \mid \texttt{not } P \mid P_1 \; op_b \; P_2 \mid a1 \; op_r \; a2$
  - $op_b \;::=\; \texttt{and} \mid \texttt{or} \mid \ldots$
  - $op_r \;::=\; < \mid \leq \mid = \mid > \mid \geq \mid \ldots$

Exercise: Building an AST

```
y := x;
z := 1;
if y > 0 then
  while y > 1 do
    z := z * y;
    y := y – 1
else
  skip
```

Ex 1: Building an AST for C code

```
void copy_bytes(char dest[], char source[], int n) {
        for (int i = 0; i < n; ++i)
                dest[i] = source[i];
}
```

Our first static analysis: AST walking

- One way to find “bugs” is to walk the AST, looking for particular patterns.
  - Traverse the AST, look for nodes of a particular type
  - Check the neighborhood of the node for the pattern in question.
  - Basically, a glorified “grep” that knows about the syntax but not semantics of a language.

Example: shifting by more than 31 bits.

Assume we want to find code patterns of the following form:

```
x << -3
```

```
z >> 35
```

For 32-bit integer vars, these operations may signal unintended typos, since it doesn't makes sense to shift by a number outside the range (0, 32).

Example: shifting by more than 31 bits.

```
For each instruction I in the program
  if I is a shift instruction
     if (type of I’s left operand is int
          && I’s right operand is a constant
          && value of constant < 0 or > 31)
        warn(“Shifting by less than 0 or more
              than 31 is meaningless”)
```

Our first static analysis: AST walking

- One way to find “bugs” is to walk the AST, looking for particular patterns.
  - Traverse the AST, look for nodes of a particular type
  - Check the neighborhood of the node for the pattern in question.
- Various frameworks, some more language-specific than others.
  - Tradeoffs between language agnosticism and semantic information available.
  - Consider “grep”: very language agnostic, not very smart.
  - Python’s ”astor” package designed for Python ASTs. Clean API; highly specific.
- One common architecture based on Visitor pattern:
  - class Visitor has a visitX method for each type of AST node X
  - Default Visitor code just descends the AST, visiting each node
  - To do something interesting for AST element of type X, override visitX
- Other more recent approaches based on semantic search, declarative logic programming, or query languages.

CodeQL

- A language for querying code. Developed by GitHub.
- Supports many common languages.
- Library of common programming patterns and optimizations.

Inefficient empty string test

Created by Documentation team, last modified on Mar 28, 2019

Name: Inefficient empty string test

Description: Checking a string for equality with an empty string is inefficient.

ID: java/inefficient-empty-string-test

Kind: problem

Severity: recommendation

Precision: high

Query: InefficientEmptyStringTest.ql

When checking whether a string s is empty, perhaps the most obvious solution is to write something like `s.equals("")` (or `"".equals(s)`). However, this actually carries a fairly significant overhead, because `String.equals` performs a number of type tests and conversions before starting to compare the content of the strings.

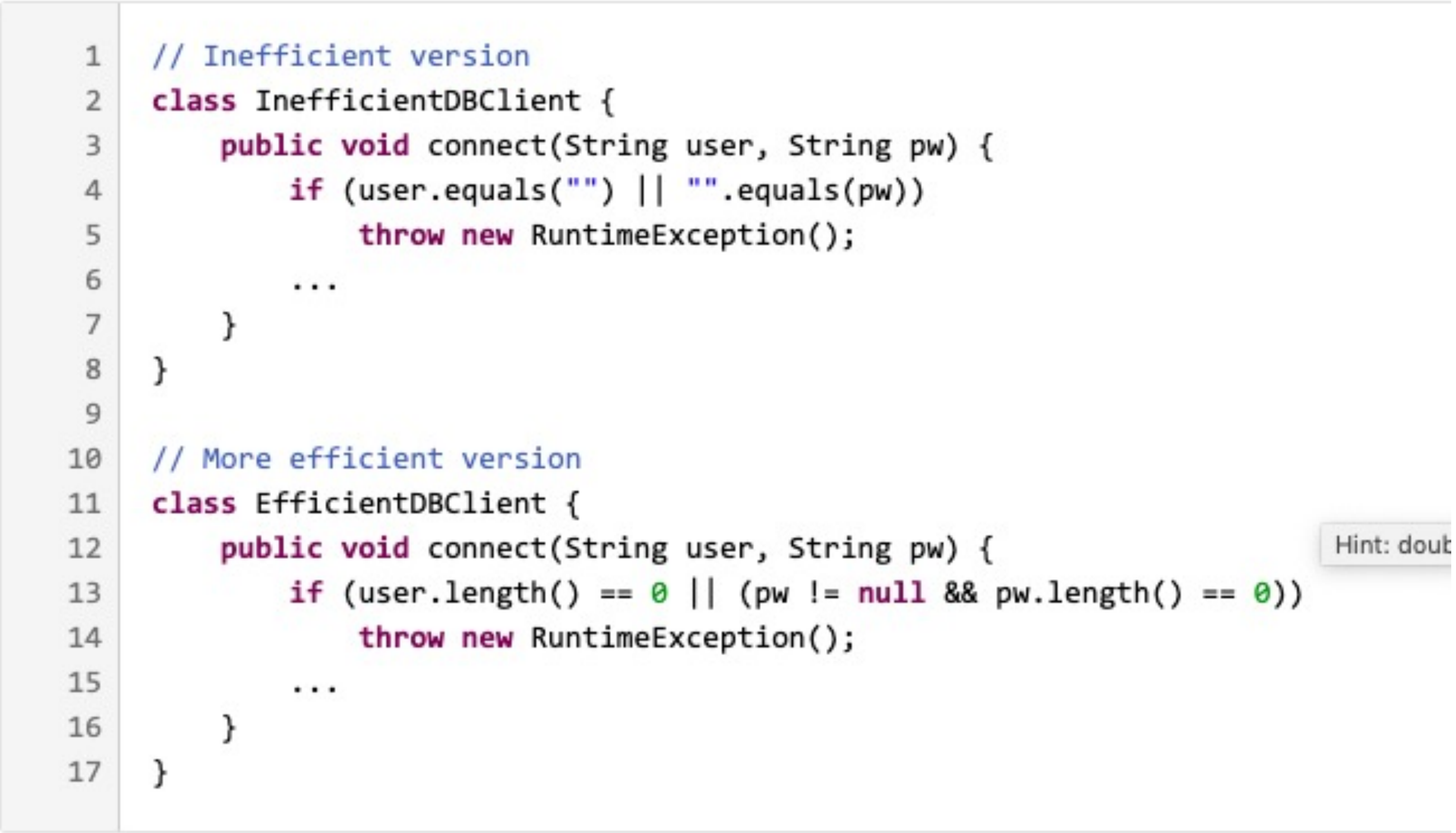
Recommendation

The preferred way of checking whether a string s is empty is to check if its length is equal to zero. Thus, the condition is `s.length() == 0`. The `length` method is implemented as a simple field access, and so should be noticeably faster than calling `equals`.

Note that in Java 6 and later, the `String` class has an `isEmpty` method that checks whether a string is empty. If the codebase does not need to support Java 5, it may be better to use that method instead.

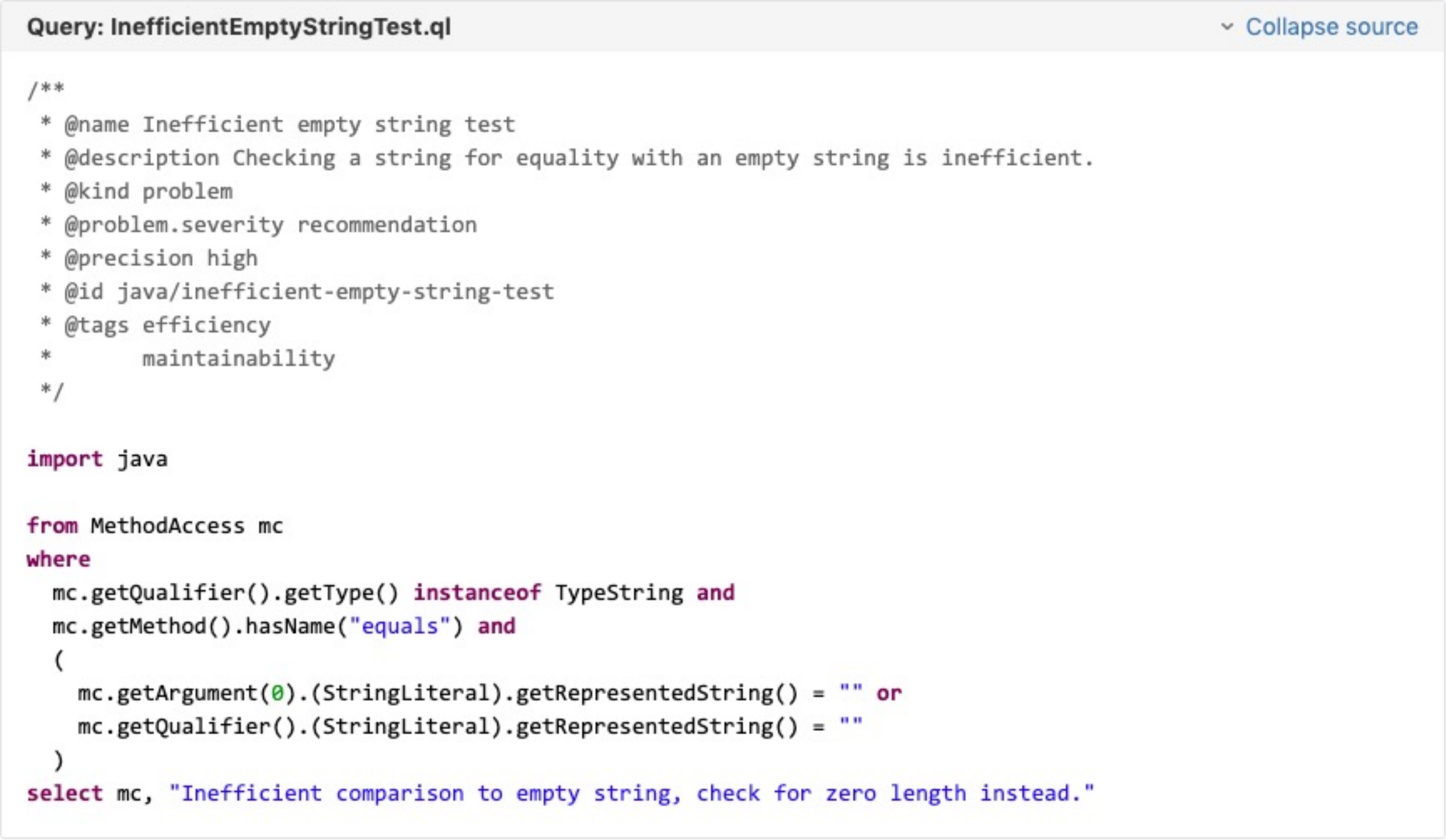
https://help.semmle.com/wiki/display/JAVA/Inefficient+empty+string+test

Example: Java string compare with “”



```java
// Inefficient version
class InefficientDBClient {
    public void connect(String user, String pw) {
        if (user.equals("") || "".equals(pw))
            throw new RuntimeException();
        ...
    }
}

// More efficient version
class EfficientDBClient {
    public void connect(String user, String pw) {
        if (user.length() == 0 || (pw != null && pw.length() == 0))
            throw new RuntimeException();
        ...
    }
}
```

CodeQL query for empty string comparison



Query: InefficientEmptyStringTest.ql

```
/**
 * @name Inefficient empty string test
 * @description Checking a string for equality with an empty string is inefficient.
 * @kind problem
 * @problem.severity recommendation
 * @precision high
 * @id java/inefficient-empty-string-test
 * @tags efficiency
 *       maintainability
 */

import java

from MethodAccess mc
where
  mc.getQualifier().getType() instanceof TypeString and
  mc.getMethod().hasName("equals") and
  (
    mc.getArgument(0).(StringLiteral).getRepresentedString() = "" or
    mc.getQualifier().(StringLiteral).getRepresentedString() = ""
  )
select mc, "Inefficient comparison to empty string, check for zero length instead."
```

Ex 2: String concatenation in a loop

- Write pseudocode for a simple syntactic analysis that warns **when string concatenation occurs in a loop**
    - Why? In Java and .NET it may be more efficient to use a StringBuffer
    - Assume any appropriate AST elements

For next time

- Get on Piazza and Canvas
- Read lecture notes and the course syllabus
- Homework 1 will be released later this week, and is due next Thursday.