

# QA: Static Analysis

Michael Hilton and **Rohan Padhye**

# Administrativa

- HW4 Part D is due tonight
- HW5 will be released soon
  - HW5 Part A is based on static/dynamic analysis tools and CI
  - HW5 Part B is based on ML explainability (covered Thursday)

# Learning goals

- Give a one sentence definition of static analysis. Explain what types of bugs static analysis targets.
- Give an example of syntactic or structural static analysis.
- Construct basic control flow graphs for small examples by hand.
- Give a high-level description of dataflow analysis and cite some example analyses.
- Explain at a high level why static analyses cannot be sound, complete, and terminating; assess tradeoffs in analysis design.
- Characterize and choose between tools that perform static analyses.
- Contrast static analysis tools with software testing and dynamic analysis tools as a means of catching bugs.

# What smells?

```
1  class Foo {  
2      int a; int b;  
3  
4      public boolean equals(Object other) {  
5          Foo foo = (Foo) other;  
6          if (foo != null)  
7              if (foo.a != this.a)  
8                  return false;  
9              if (foo.b == this.b)  
10                 return true;  
11                 else return false;  
12            }  
13  
14            public int a() {  
15                return this.a();  
16            }  
17  
18            public int b() {  
19                return this.b();  
20            }  
21        }
```

# What smells?

```
1 ▾ int dtls1_process_heartbeat(SSL *s)
2   {
3     unsigned char *p = &s->s3->rrec.data[0], *pl;
4     unsigned short hbtype;
5     unsigned int payload;
6     unsigned int padding = 16; /* Use minimum padding */
7
8     /* Read type and payload length first */
9     hbtype = *p++;
10    n2s(p, payload);
11    pl = p;
12
13    if (s->msg_callback)
14      s->msg_callback(0, s->version, TLS1_RT_HEARTBEAT,
15                    &s->s3->rrec.data[0], s->s3->rrec.length,
16                    s, s->msg_callback_arg);
17
18    if (hbtype == TLS1_HB_REQUEST)
19    {
20      unsigned char *buffer, *bp;
21      int r;
22
23      /* Allocate memory for the response, size is 1 byte
24       * message type, plus 2 bytes payload length, plus
25       * payload, plus padding
26       */
27      buffer = OPENSSL_malloc(1 + 2 + payload + padding);
28      bp = buffer;
29
30      /* Enter response type, length and copy payload */
31      *bp++ = TLS1_HB_RESPONSE;
32      s2n(payload, bp);
33      memcpy(bp, pl, payload);
34      bp += payload;
35      /* Random padding */
36      RAND_pseudo_bytes(bp, padding);
37
38      r = dtls1_write_bytes(s, TLS1_RT_HEARTBEAT, buffer, 3 + payload + padding);
```



# Static Analysis

- Try to discover issues by analyzing source code. No need to run.
- Defects of interest may be on uncommon or difficult-to-force execution paths for testing.
- What we really want to do is check the entire possible state space of the program for particular properties.

# Defects Static Analysis can Catch

- 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**

github.com/marketplace?category=code-quality

Search or jump to... Pull requests Issues Marketplace Explore

Marketplace Search results

Types

Search for apps and actions

Types

Apps

Actions

Categories

API management

Chat

Code quality

Code review

Continuous integration

Dependency management

Deployment

IDEs

Learning

Localization

Mobile

Monitoring

Project management

Publishing

Recently added

Security

Support

Testing

Utilities

Filters

Verification

Verified

Unverified

Your items

Purchases

### 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 Benchmark 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

Previous Next

Also recommended for you

https://github.com/marketplace?category=api-management



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

```
public class Test {
```

▼ **Lint** Missing a Javadoc comment.

Java  
1:02 AM, Aug 21

[Please fix](#)

[Not useful](#)

```
    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

[Please fix](#)

Suggested fix attached: [show](#)

[Not useful](#)

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

//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");  
    }  
}
```

Apply

Cancel

# How do they work?

```
1 class Foo {
2     int a; int b;
3
4     public boolean equals(Object obj) {
5         Foo foo = (Foo) obj;
6         if (foo != null)
7             if (foo.a != this.a)
8                 return false;
9             if (foo.b == this.b)
10                return true;
11            else return false;
12        }
13
14        public int a() {
15            return this.a();
16        }
17
18        public int b() {
19            return this.b();
20        }
21 }
```

```
1 int dtls1_process_heartbeat(SSL *s)
2 {
3     unsigned char *p = &s->s3->rrec.data[0], *pl;
4     unsigned short hbtype;
5     unsigned int payload;
6     unsigned int padding = 16; /* Use minimum padding */
7
8     /* Read type and payload length first */
9     hbtype = *p++;
10    n2s(p, payload);
11    pl = p;
12
13    if (s->msg_callback)
14        s->msg_callback(0, s->version, TLS1_RT_HEARTBEAT,
15                        &s->s3->rrec.data[0], s->s3->rrec.length,
16                        s, s->msg_callback_arg);
17
18    if (hbtype == TLS1_HB_REQUEST)
19    {
20        unsigned char *buffer, *bp;
21        int r;
22
23        /* Allocate memory for the response, size is 1 byte
24         * message type, plus 2 bytes payload length, plus
25         * payload, plus padding
26         */
27        buffer = OPENSSL_malloc(1 + 2 + payload + padding);
28        bp = buffer;
29
30        /* Enter response type, length and copy payload */
31        *bp++ = TLS1_HB_RESPONSE;
32        s2n(payload, bp);
33        memcpy(bp, pl, payload);
34        bp += payload;
35        /* Random padding */
36        RAND_pseudo_bytes(bp, padding);
37
38        r = dtls1_write_bytes(s, TLS1_RT_HEARTBEAT, buffer, 3 + payload + padding);
```

# Two fundamental concepts

- Abstraction.
  - Elide details of a specific implementation.
  - Capture semantically relevant details; ignore the rest.
- Programs as data.
  - Programs are just trees/graphs!
  - ...and we know lots of ways to analyze trees/graphs, right?

# Defining Static Analysis

- Systematic examination of an **abstraction** of program **state space**.
  - Does not execute code! (like code review)
- **Abstraction**: A representation of a program that is simpler to analyze.
  - Results in fewer states to explore; makes difficult problems tractable.
- Check if a **particular property** holds over the entire state space:
  - Liveness: “something good eventually happens.”
  - Safety: “this bad thing can’t ever happen.”
  - Compliance with mechanical design rules.

# The Bad News: Rice's Theorem

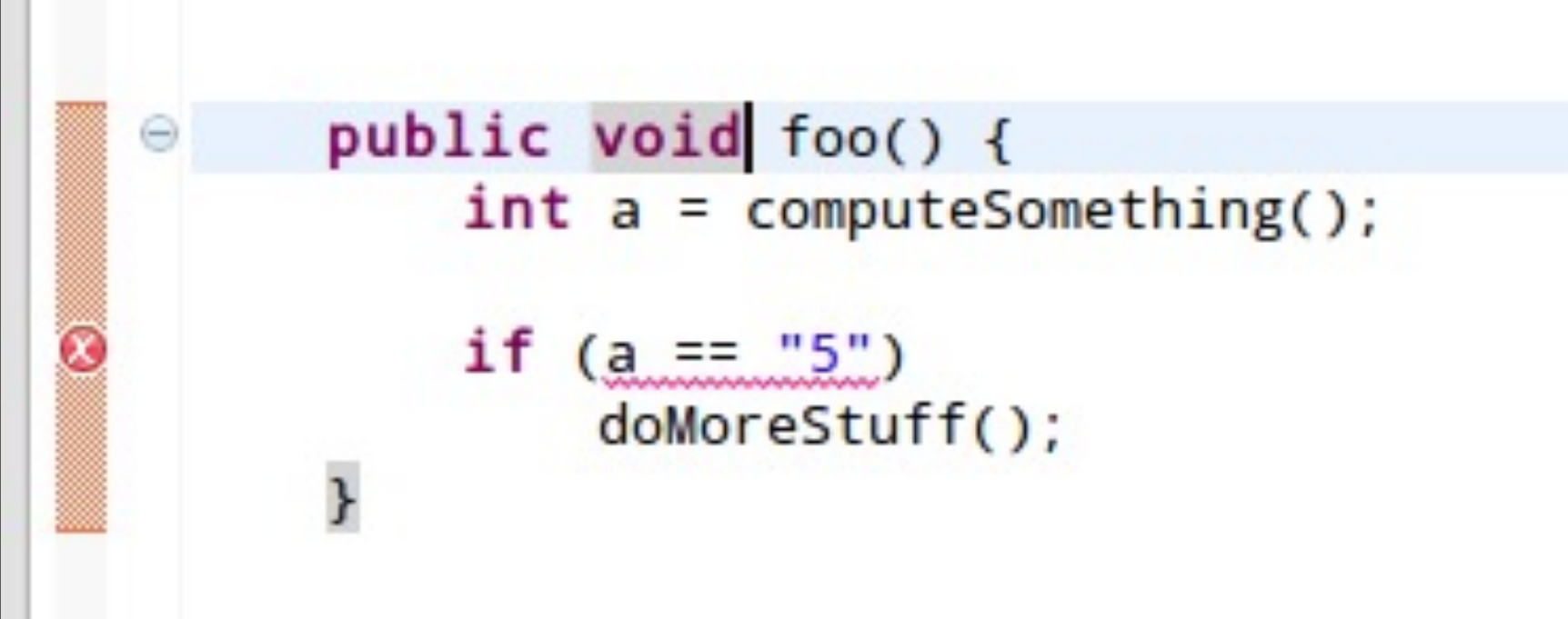
"Any nontrivial property about the language recognized by a Turing machine is undecidable."

Henry Gordon Rice, 1953

Every static analysis is necessarily incomplete or unsound or undecidable (or multiple of these)

# SIMPLE SYNTACTIC AND STRUCTURAL ANALYSES

# Type Analysis



A screenshot of a code editor interface. On the left, there is a vertical orange bar with a red circle containing a white 'X' icon. The code is as follows:

```
public void foo() {  
    int a = computeSomething();  
  
    if (a == "5")  
        doMoreStuff();  
}
```

The code is written in a monospaced font. The first line, `public void foo() {`, is highlighted with a light blue background. The variable `a` in the `if` statement is underlined with a red wavy line, indicating a type mismatch error between the `int` variable and the `"5"` string literal.

# Abstraction: abstract syntax tree

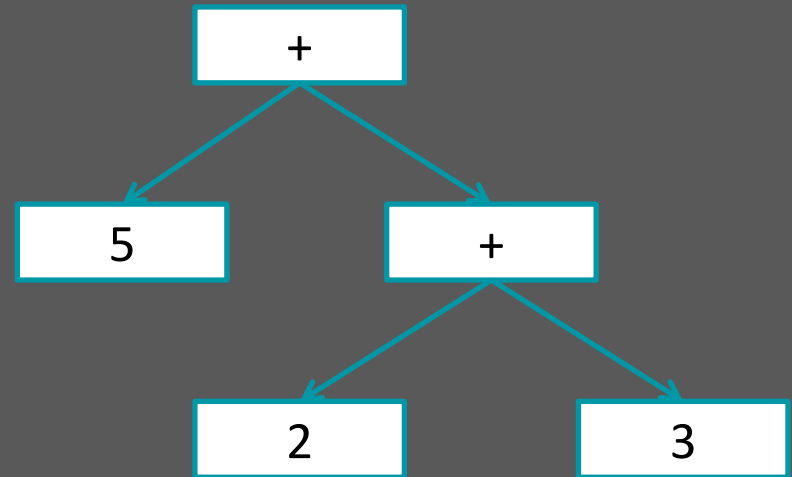
Tree representation of the syntactic structure of source code.

Parsers convert concrete syntax into abstract syntax, and deal with resulting ambiguities.

Records only the semantically relevant information.

Abstract: doesn't represent every detail (like parentheses); these can be inferred from the structure.

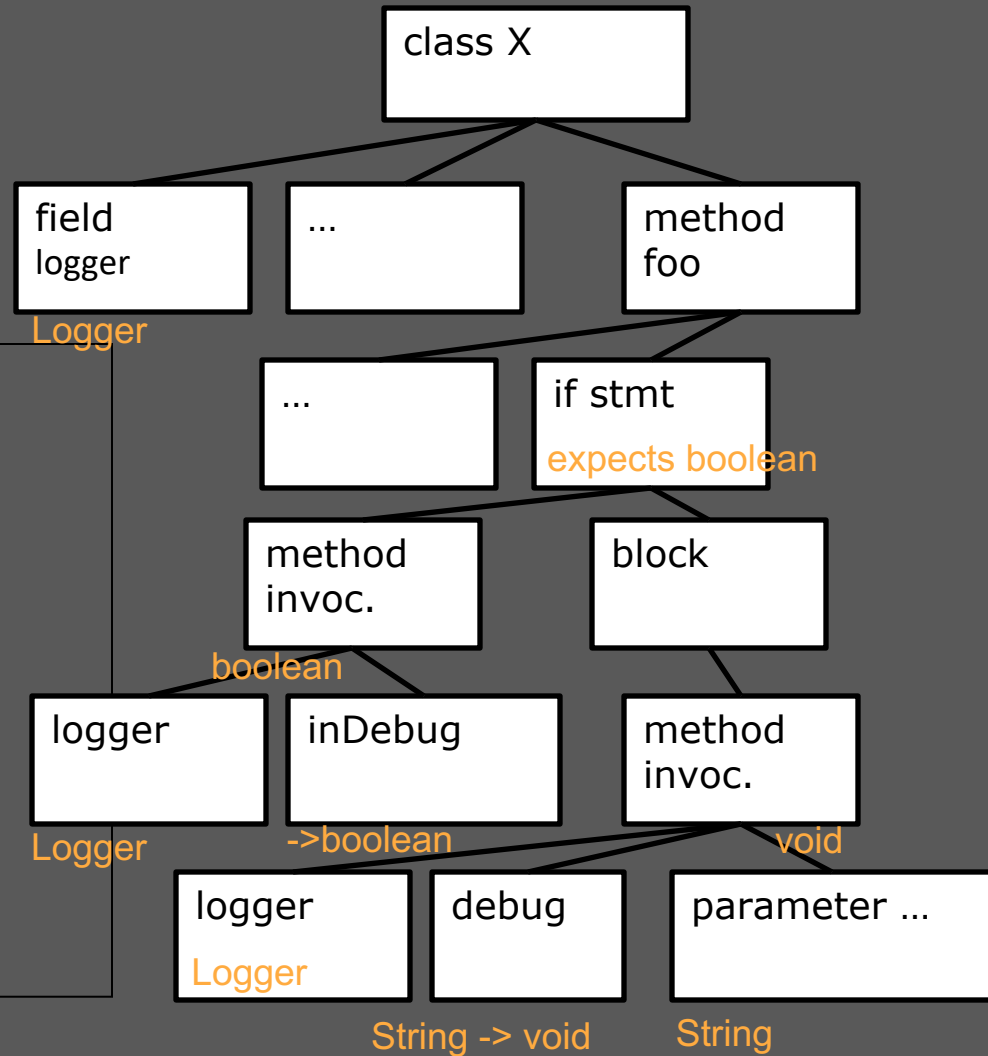
(How to build one? Take compilers!)





# Type checking

```
class X {  
  Logger logger;  
  public void foo() {  
    ...  
    if (logger.isDebugEnabled()) {  
      logger.debug("We have " + conn +  
"connections.");  
    }  
  }  
}  
class Logger {  
  boolean inDebug() {...}  
  void debug(String msg) {...}  
}
```



# Syntactic Analysis

Find every occurrence of this pattern:

```
public foo() {  
    ...  
    logger.debug("We have " + conn + "connections.");  
}
```

```
public foo() {  
    ...  
    if (logger.isDebugEnabled()) {  
        logger.debug("We have " + conn + "connections.");  
    }  
}
```

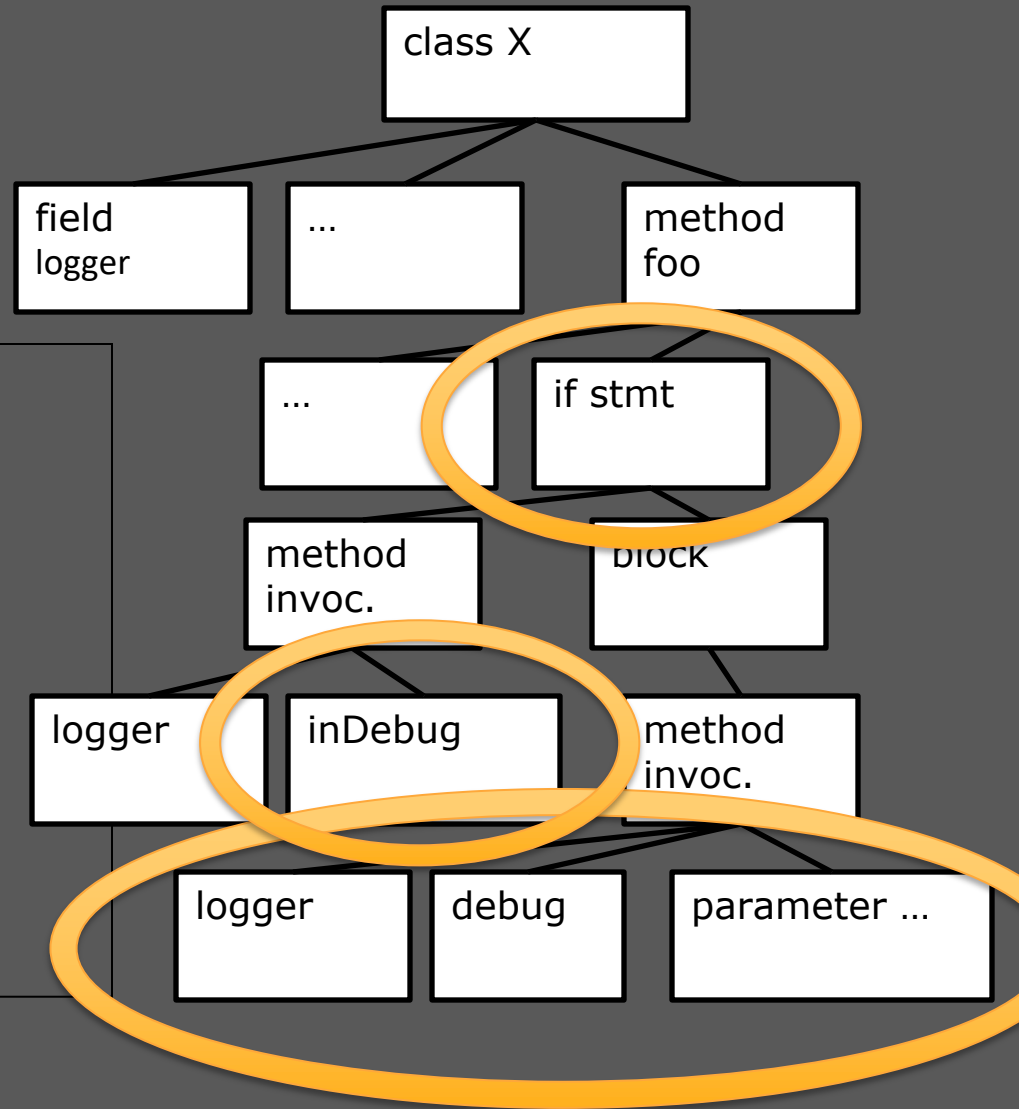
```
grep "if \(logger\.isEnabledForDebug" . -r
```

# Abstract syntax tree walker

- Check that we don't create strings outside of a `Logger.isDebugEnabled` check
- Abstraction:
  - Look only for calls to `Logger.debug ( )`
  - Make sure they're all surrounded by `if (Logger.isDebugEnabled ( ))`
- Systematic: Checks all the code
- Known as an Abstract Syntax Tree (AST) walker
  - Treats the code as a structured tree
  - Ignores control flow, variable values, and the heap
  - Code style checkers work the same way

# Structural Analysis

```
class X {  
  Logger logger;  
  public void foo() {  
    ...  
    if (logger.isDebugEnabled()) {  
      logger.debug("We have " + conn +  
        "connections.");  
    }  
  }  
}  
class Logger {  
  boolean inDebug() {...}  
  void debug(String msg) {...}  
}
```



# Structural analysis for possible NPEs?

```
1  if (foo != null)
2      foo.a();
3  foo.b();
4
```

# Which of these should be flagged for NPE?

Surely safe? Surely bad? Suspicious? // Limitations of structural analysis

A

```
1  if (foo != null)
2      foo.a();
3  foo.b();
```

B

```
1  if (foo == null)
2      foo = new Foo();
3  foo.b();
```

C

```
1  if (foo != null)
2      foo.a();
3  else
4      foo = new Foo();
5
6  foo.b();
```

D

```
1  if (foo != null)
2      foo.a();
3  else
4      foo.b();
```

# CONTROL-FLOW AND DATA-FLOW ANALYSIS

# Control/Dataflow analysis

- **Reason** about all possible executions, via paths through a *control flow graph*.
  - Track information relevant to a property of interest at every *program point*.
- Define an **abstract domain** that captures only the values/states relevant to the property of interest.
- **Track** the abstract state, rather than all possible concrete values, for all possible executions (paths!) through the graph.



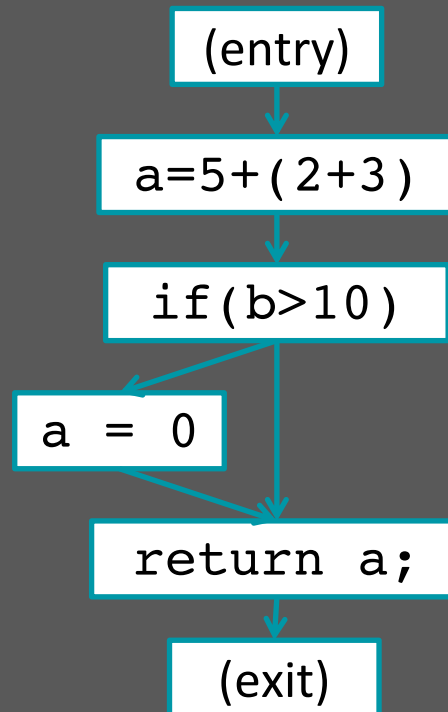
# Control/Dataflow analysis

- Reason about all possible executions, via paths through a *control flow graph*.
  - Track information relevant to a property of interest at every *program point*.
- Define an **abstract domain** that captures only the values/states relevant to the property of interest.
- Track the abstract state, rather than all possible concrete values, for all possible executions (paths!) through the graph.

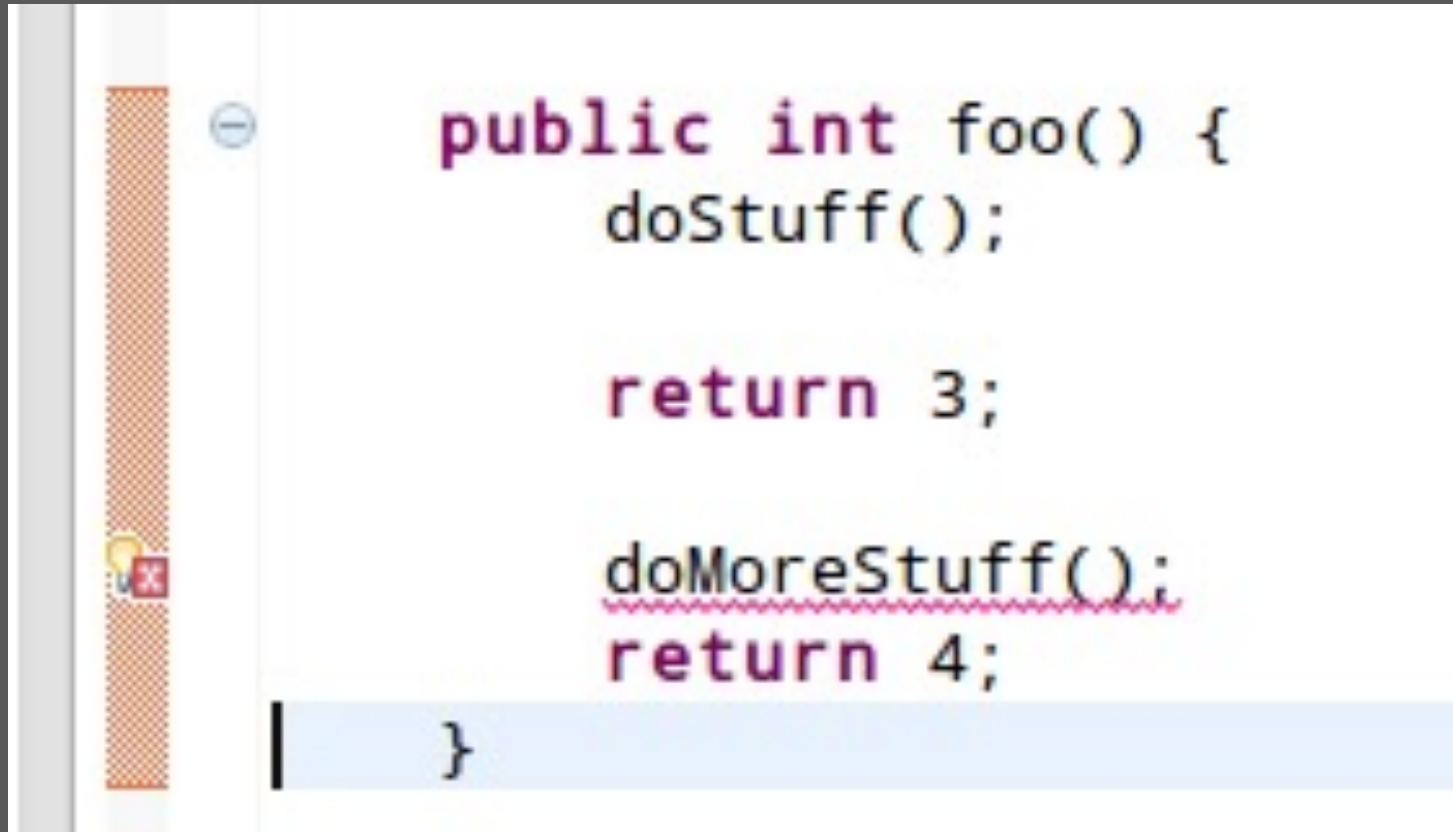
# Control flow graphs

- A tree/graph-based representation of the flow of control through the program.
  - Captures all possible execution paths.
- Each node is a basic block: no jumps in or out.
- Edges represent control flow options between nodes.
- *Intra-procedural*: within one function.
  - cf. inter-procedural

```
1.  a = 5 + ( 2 + 3 )
2.  if ( b > 10 ) {
3.      a = 0;
4.  }
5.  return a;
```



# How can CFG be used to identify this issue?



A screenshot of a code editor showing a Java method. The method is named `foo()` and is of type `public int`. It contains three statements: `doStuff();`, `return 3;`, and `doMoreStuff();`. The `doMoreStuff();` statement is underlined with a red wavy line, indicating a control flow graph issue. The `return 4;` statement is also present, but it is not reached by the `doMoreStuff();` statement. The closing brace `}` is on the last line. The code is displayed in a monospaced font with syntax highlighting: keywords in purple, identifiers in black, and literals in blue. The editor has a light blue background and a vertical scrollbar on the left.

```
public int foo() {  
    doStuff();  
  
    return 3;  
  
    doMoreStuff();  
    return 4;  
}
```

# Control/Dataflow analysis

- Reason about all possible executions, via paths through a *control flow graph*.
  - Track information relevant to a property of interest at every program point.
- Define an **abstract domain** that captures only the values/states relevant to the property of interest.
- Track the abstract state, rather than all possible concrete values, for all possible executions (paths!) through the graph.

# NPE analysis revisited

A

```
1  if (foo != null)
2      foo.a();
3  foo.b();
```

B

```
1  if (foo == null)
2      foo = new Foo();
3  foo.b();
```

C

```
1  if (foo != null)
2      foo.a();
3  else
4      foo = new Foo();
5
6  foo.b();
```

D

```
1  if (foo != null)
2      foo.a();
3  else
4      foo.b();
```

# Abstract Domain for NPE Analysis

- Map of `Var`  $\rightarrow$  `{Null, NotNull, Unknown}`
- For example:
  - `foo`  $\rightarrow$  `Null`
  - `bar`  $\rightarrow$  `NotNull`
  - `baz`  $\rightarrow$  `Unknown`
- Mapping tracked at every program point (before/after each CFG node). Updated across nodes and edges.
- `// let's say foo  $\rightarrow$  Null and bar  $\rightarrow$  NotNull`  
`foo = new Foo();`  
`// at this point, we have foo  $\rightarrow$  NotNull and bar  $\rightarrow$  Null`

# NPE analysis revisited

A

```
1  if (foo != null)
2      foo.a();
3  foo.b();
```

B

```
1  if (foo == null)
2      foo = new Foo();
3  foo.b();
```

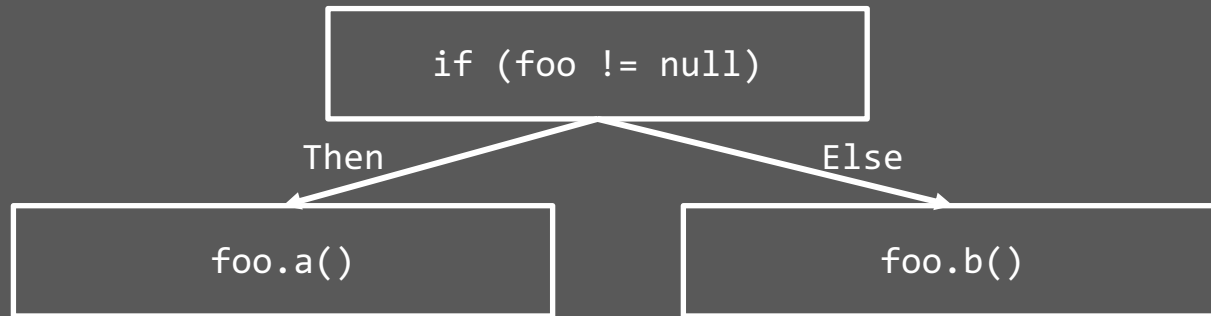
C

```
1  if (foo != null)
2      foo.a();
3  else
4      foo = new Foo();
5
6  foo.b();
```

D

```
1  if (foo != null)
2      foo.a();
3  else
4      foo.b();
```

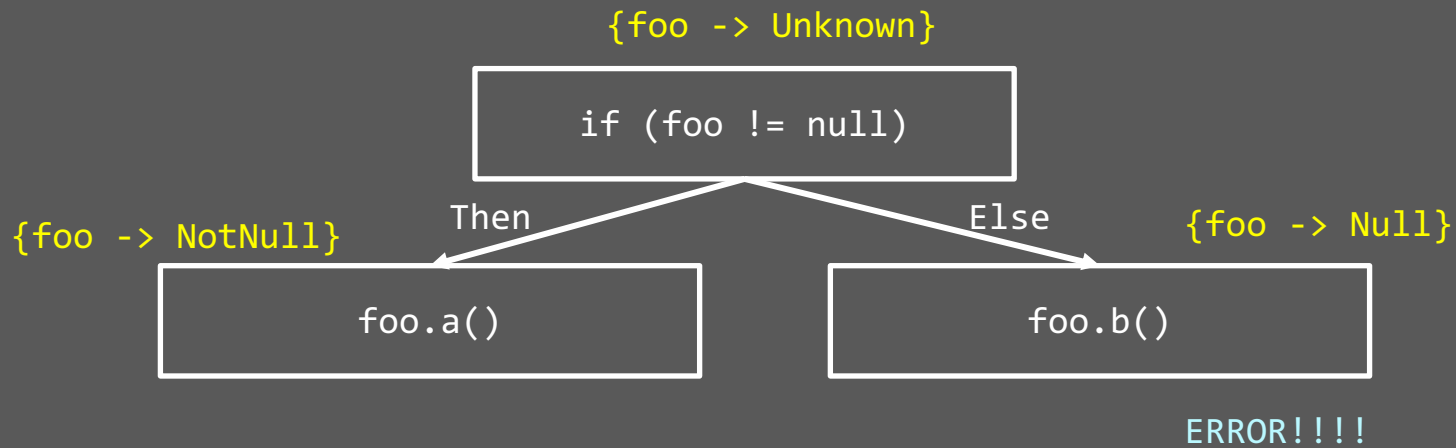
# Data-Flow Analysis Examples



```
1  if (foo != null)
2      foo.a();
3  else
4      foo.b();
```

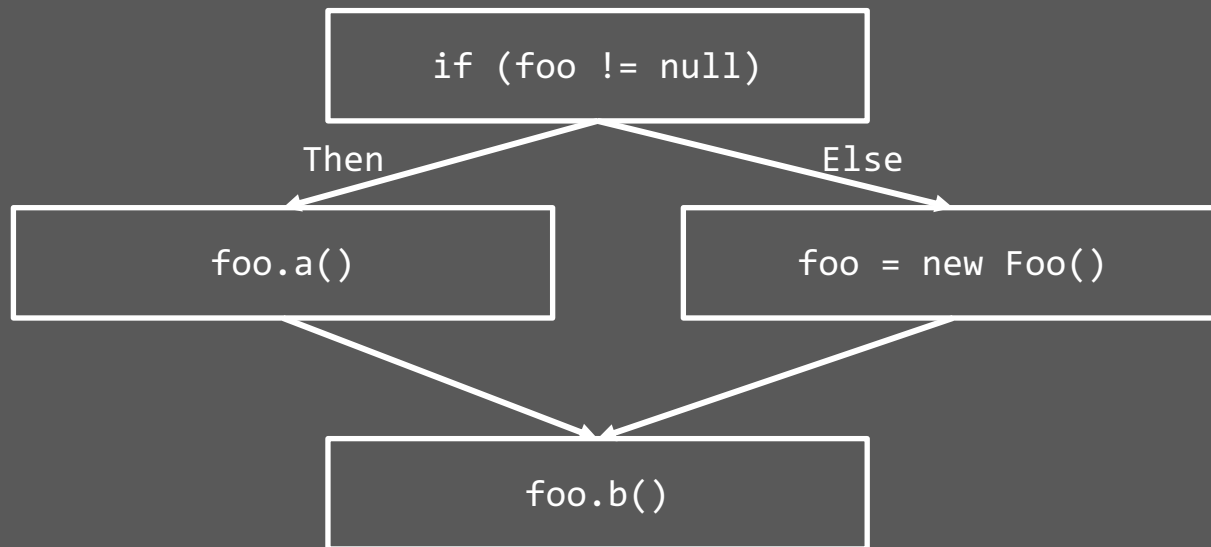


# Data-Flow Analysis Examples



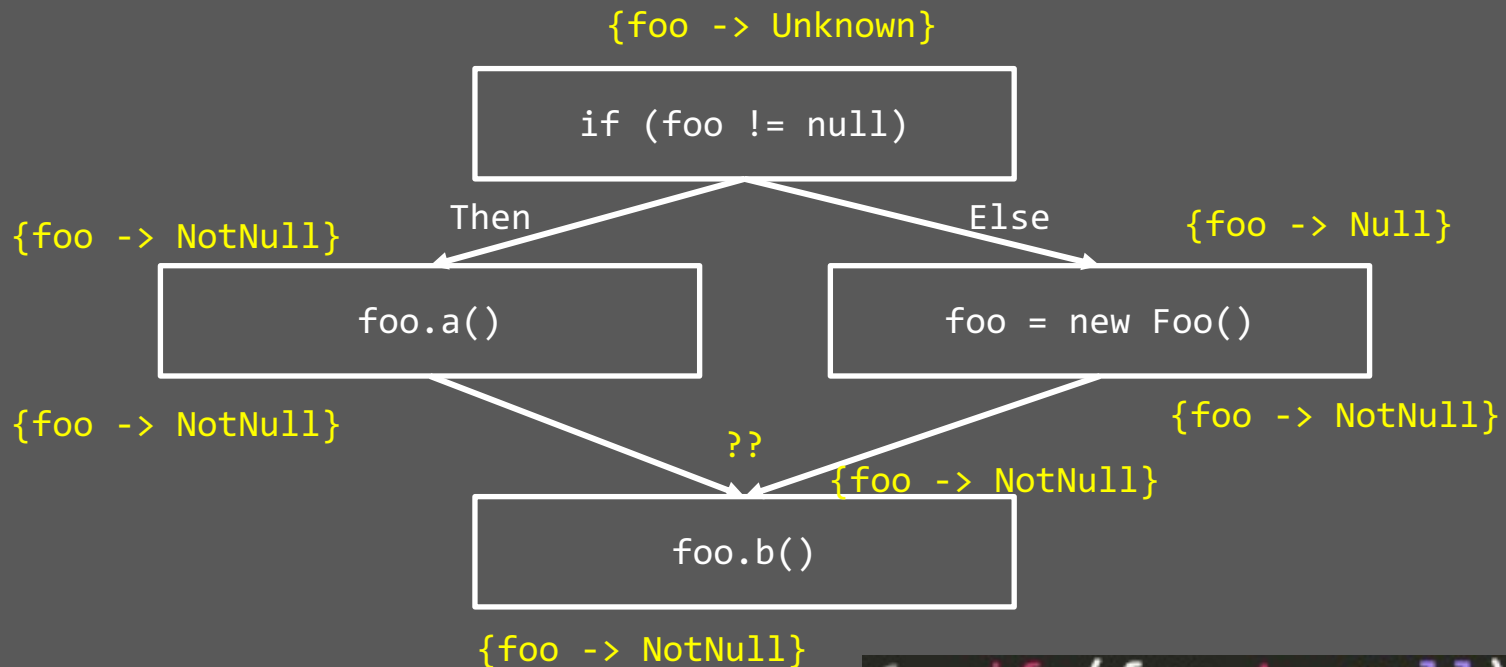
```
1  if (foo != null)
2      foo.a();
3  else
4      foo.b();
```

# Data-Flow Analysis Examples



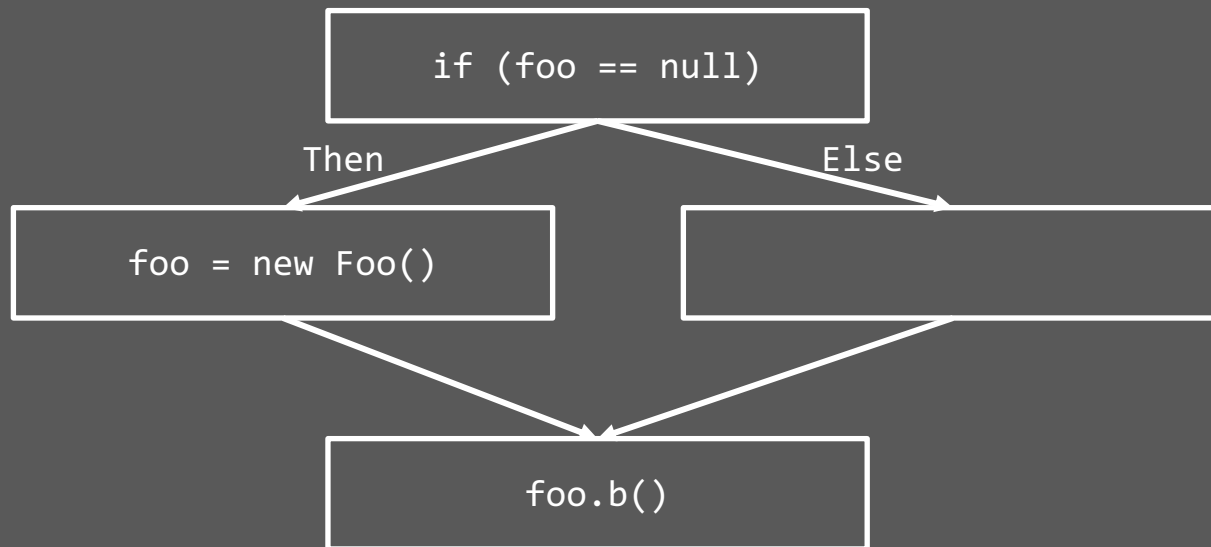
```
1  if (foo != null)
2      foo.a();
3  else
4      foo = new Foo();
5
6  foo.b();
```

# Data-Flow Analysis Examples



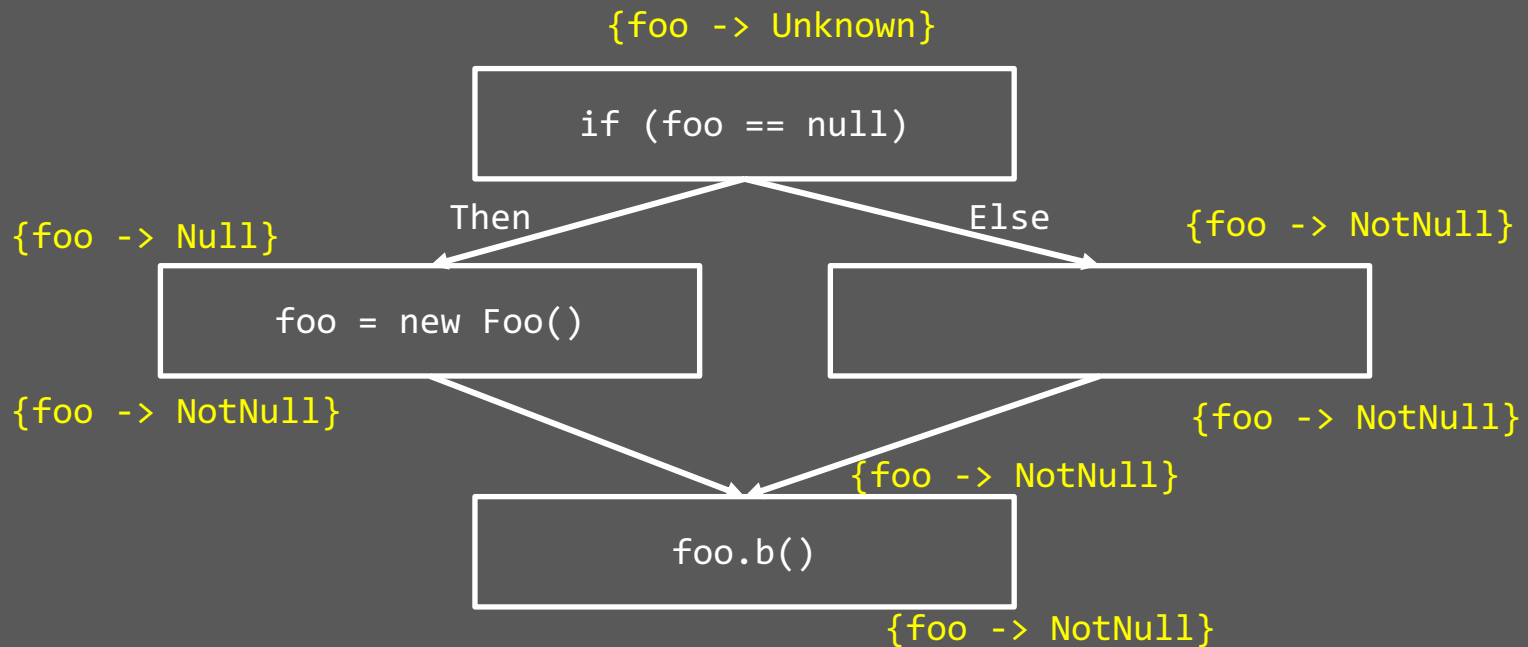
```
1 if (foo != null)
2     foo.a();
3 else
4     foo = new Foo();
5
6 foo.b();
```

# Data-Flow Analysis Examples



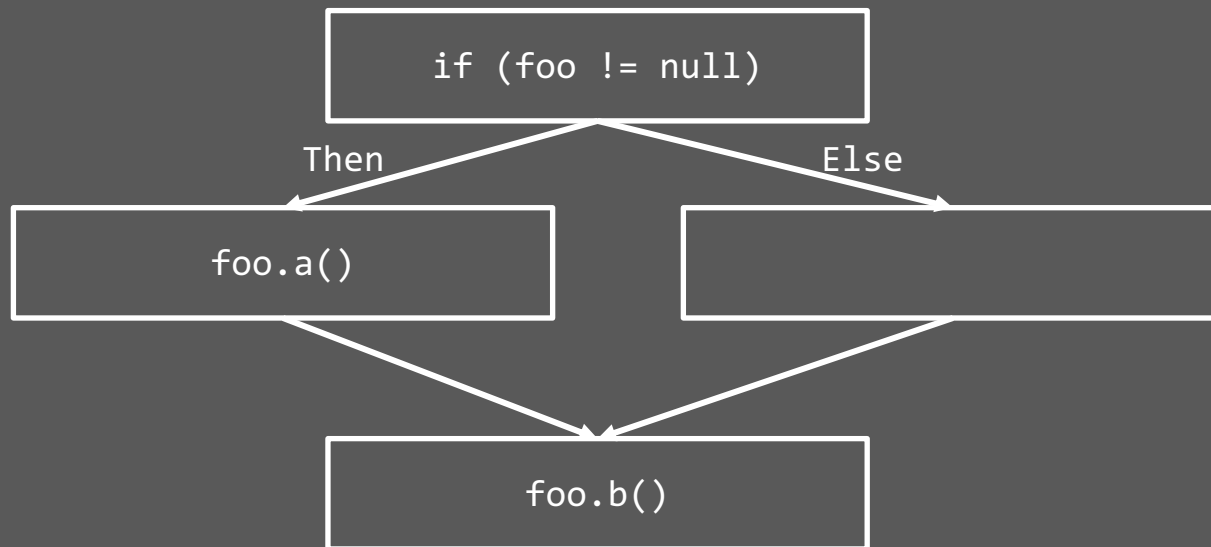
```
1  if (foo == null)
2      foo = new Foo();
3  foo.b();
```

# Data-Flow Analysis Examples



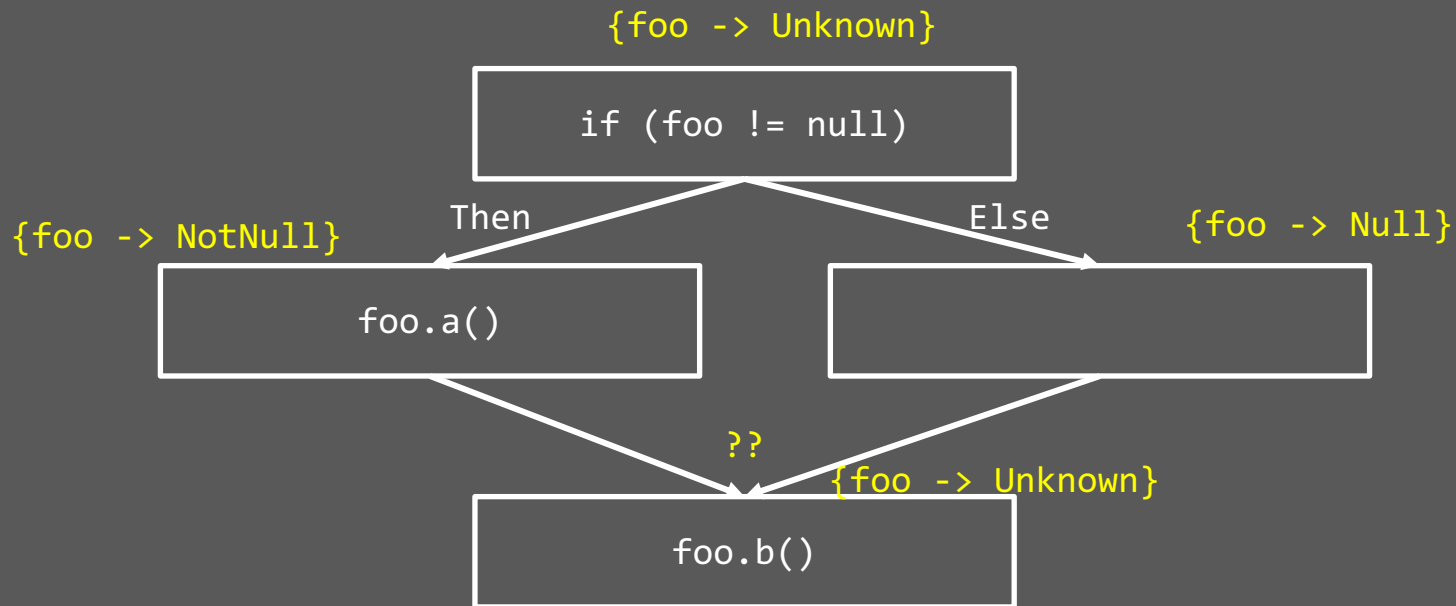
```
1  if (foo == null)
2      foo = new Foo();
3  foo.b();
```

# Data-Flow Analysis Examples



```
1  if (foo != null)
2      foo.a();
3  foo.b();
4
```

# Data-Flow Analysis Examples

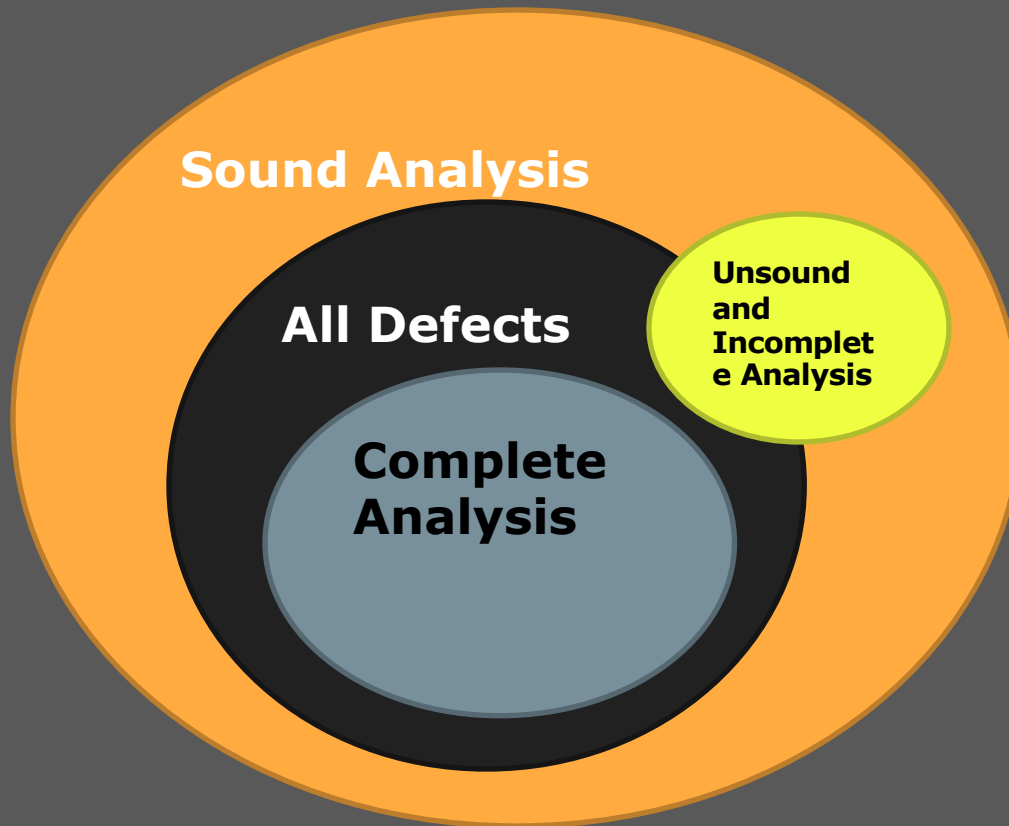


```
1  if (foo != null)
2      foo.a();
3  foo.b();
4
```

# Interpreting abstract states

- “Null” means “must be NULL at this point, regardless of path taken”
- “NotNull” is similar
- “Unknown” means “may be NULL or not null depending on the path taken”
- Unknown must be dealt with due to Rice’s theorem
  - Can make analysis smarter (at the cost of more algorithmic complexity) to reduce Unknowns, but can’t get rid of them completely
- Whether to raise a flag on UNKNOWN access depends on usability/soundness.
  - False positives if warning on UNKNOWN
  - False negatives if no warning on UNKNOWN





# Examples of Data-Flow Analyses

- Null Analysis
  - Var  $\rightarrow$  {Null, NotNull, UNKNOWN}
- Zero Analysis
  - Var  $\rightarrow$  {Zero, NonZero, UNKNOWN}
- Sign Analysis
  - Var  $\rightarrow$  {-, +, 0, UNKNOWN}
- Range Analysis
  - Var  $\rightarrow$  {[0, 1], [1, 2], [0, 2], [2, 3], [0, 3], ..., UNKNOWN}
- Constant Propagation
  - Var  $\rightarrow$  {1, 2, 3, ..., UNKNOWN}
- File Analysis
  - File  $\rightarrow$  {Open, Close, UNKNOWN}
- Tons more!!!

# Data-Flow Analysis: Challenges

- Loops
  - Fixed-point algorithms guarantee termination at the cost of losing information (“Unknown”)
- Functions
  - Analyze them separately or analyze whole program at once
  - “Context-sensitive” analyses specialize on call sites (think: duplicate function body for every call site via inlining)
- Recursion
  - Makes context-sensitive analyses explode (cf. loops)
- Object-oriented programming
- Heap memory
  - Need to abstract mapping keys not just values
- Exceptions

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

- Offered in Spring 2022 (by Rohan Padhye). Sign up now if interested!
- Topics include:
  - Program semantics
  - Dataflow analysis
  - Pointer Analysis
  - Control flow analysis for functional programs
  - Model checking
  - SMT solvers
  - Program verification
  - Symbolic Execution
  - Race Detection for Concurrent Programs
- Beautiful theory 🥰 + building automatic bug-finding tools 🖥️
- Check out <https://cmu-program-analysis.github.io> (2020 website)

# Static Analysis vs. Testing

- Which one to use when?
- Points in favor of Static Analysis
  - Don't need to set up run environment, etc.
  - Can analyze functions/modules independently and in parallel
  - Don't need to think of (or try to generate) program inputs
- Points in favor of Testing / Dynamic Analysis
  - Not deterred by complex program features
  - Can easily handle external libraries, platform-specific config, etc.
  - Ideally no false positives
  - Easier to debug when a failure is identified