

Foundations of Software Engineering

Static analysis

Christian Kaestner

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?

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.
- Distinguish between control- and data-flow analyses; define and then step through on code examples simple control and data-flow analyses.
- Implement a dataflow analysis.
- 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.

```
goto fail;
```

```

1. static OSStatus
2. SSLVerifySignedServerKeyExchange(SSLContext *ctx, bool isRsa,
3.                                SSLBuffer signedParams,
4.                                uint8_t *signature,
5.                                UInt16 signatureLen) {
6.     OSStatus err;
7.     ....
8.     if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
9.         goto fail;
10.    if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
11.        goto fail;
12.        goto fail;
13.    if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
14.        goto fail;
15.    ...
16.fail:
17.    SSLFreeBuffer(&signedHashes);
18.    SSLFreeBuffer(&hashCtx);
19.    return err;
20.}

```

```

1. /* from Linux 2.3.99 drivers/block/raid5.c */
2. static struct buffer_head *
3. get_free_buffer(struct stripe_head * sh,
4.                int b_size) {
5.     struct buffer_head *bh;
6.     unsigned long flags;
7.     save_flags(flags);
8.     cli(); // disables interrupts
9.     if ((bh = sh->buffer_pool, == NULL)
10.        return NULL;
11.     sh->buffer_pool = bh -> b_next;
12.     bh->b_size = b_size;
13.     restore_flags(flags); // re-enables interrupts
14.     return bh;
15. }

```

ERROR: function returns with
interrupts disabled!

With thanks to Jonathan Aldrich; example from Engler et al., *Checking system rules Using System-Specific, Programmer-Written Compiler Extensions*, OSDI '000

Could you have found them?

- How often would those bugs trigger?
- Driver bug:
 - What happens if you return from a driver with interrupts disabled?
 - Consider: that's one function
 - ...in a 2000 LOC file
 - ...in a module with 60,000 LOC
 - ...IN THE LINUX KERNEL
- **Moral:** *Some defects are very difficult to find via testing, inspection.*

Klocwork: Our source code analyzer caught Apple's 'gotofail' bug

If Apple had used a third-party source code analyzer on its encryption library, it could have avoided the "gotofail" bug.



by Declan McCullagh | February 28, 2014 1:13 PM PST

Follow



57



223



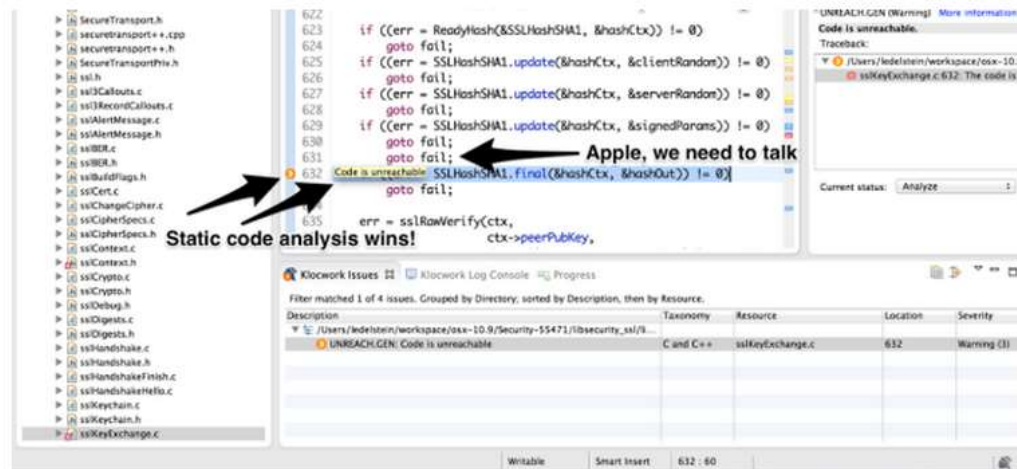
23



5

More +

Comments 25



Klocwork's Larry Edelstein sent us this screen snapshot, complete with the arrows, showing how the company's product would have nabbed the "goto fail" bug.

(Credit: Klocwork)

It was a single repeated line of code -- "goto fail" -- that left millions of Apple users vulnerable to Internet attacks until the company finally [fixed it Tuesday](#).

Featured Posts

Google unveils Android wearables
Internet & Media



Motorola Powerbeat Internet smartwatch



OK, Glass in my face Cutting Edge



Apple iPad product



iPad with camera

Most Popular



Giant 3D house 6k Facebook



Exclusive Doesch 716 Twitter



Google+ four can 771 Google+

Connect With CNET



Facebook Like Us



Google+

http://news.cnet.com/8301-1009_3-57619754-83/klocwork-our-source-code-analyzer-caught-apples-gotofail-bug/

Defects of interest...

- Are on uncommon or difficult-to-force execution paths. (vs testing)
- Executing (or interpreting/otherwise analyzing) all paths concretely to find such defects is infeasible.
- **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, mechanical 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

DEFINING STATIC ANALYSIS

What is Static Analysis?

- **Systematic** examination of an **abstraction** of program **state space**.
 - Does not execute code! (like code review)
- **Abstraction**: produce 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.”

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)

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

Sound Analysis:

- reports all defects
- > no false negatives
- typically overapproximated

Complete Analysis:

- every reported defect is an actual defect
- > no false positives
- typically underapproximated

How does testing relate? And formal verification?

SIMPLE SYNTACTIC AND STRUCTURAL ANALYSES

Syntactic Analysis

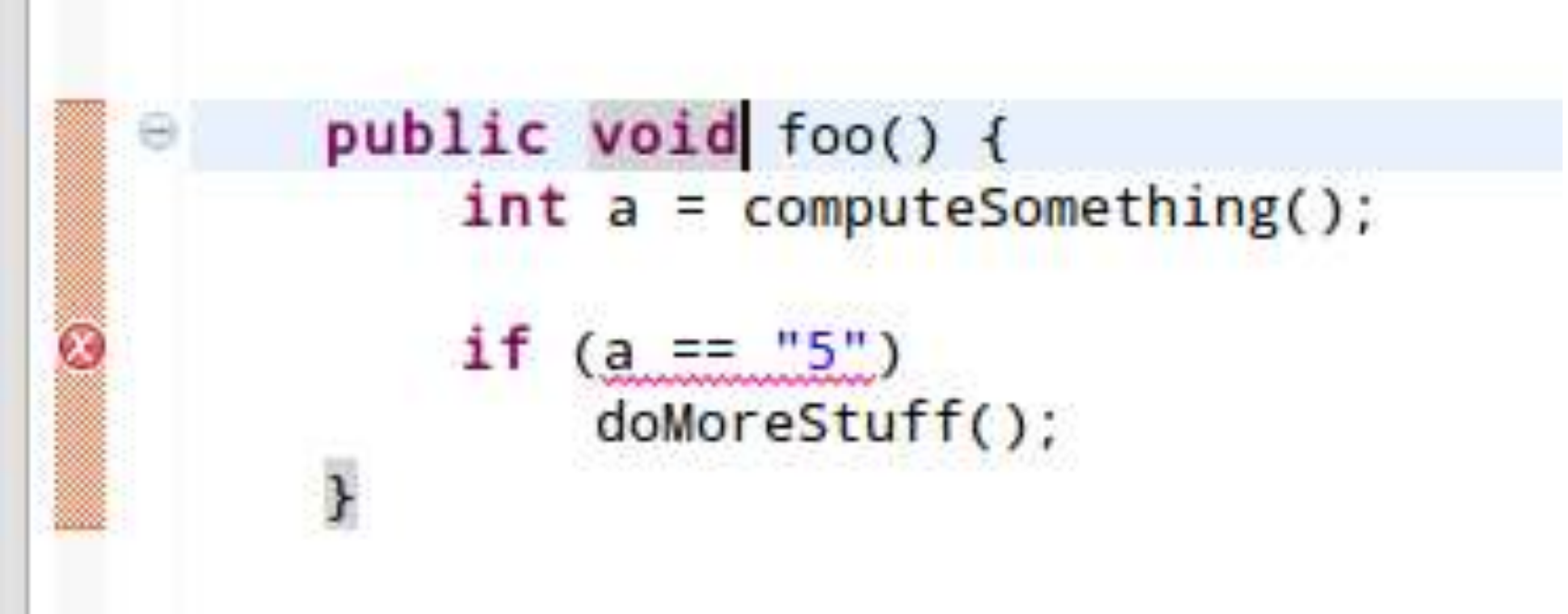
Find every occurrence of this pattern:

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

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

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

Type Analysis



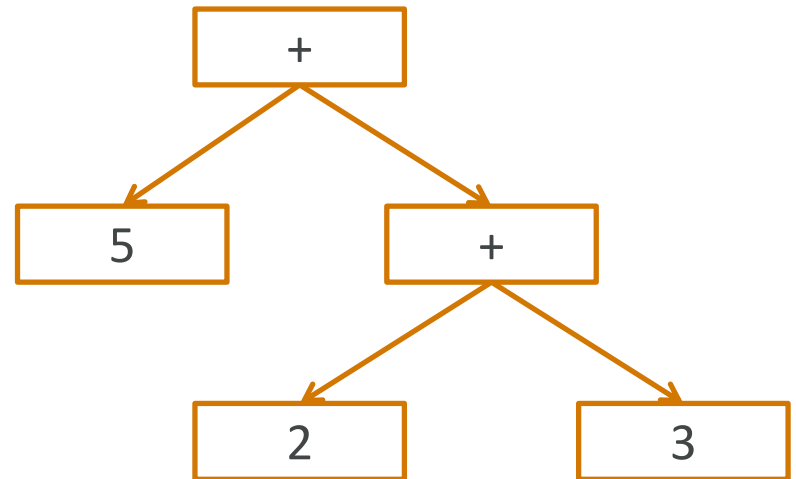
A code editor snippet showing a Java method. The method signature is `public void foo() {`. The first line inside the method is `int a = computeSomething();`. The second line is `if (a == "5")`, where the string `"5"` is highlighted in blue and has a red squiggly line underneath it, indicating a type error. The third line is `doMoreStuff();`. The method is closed with a closing brace `}`. On the left side of the code editor, there is a vertical toolbar with a red 'X' icon, suggesting a compilation or runtime error.

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

Abstraction: abstract syntax tree

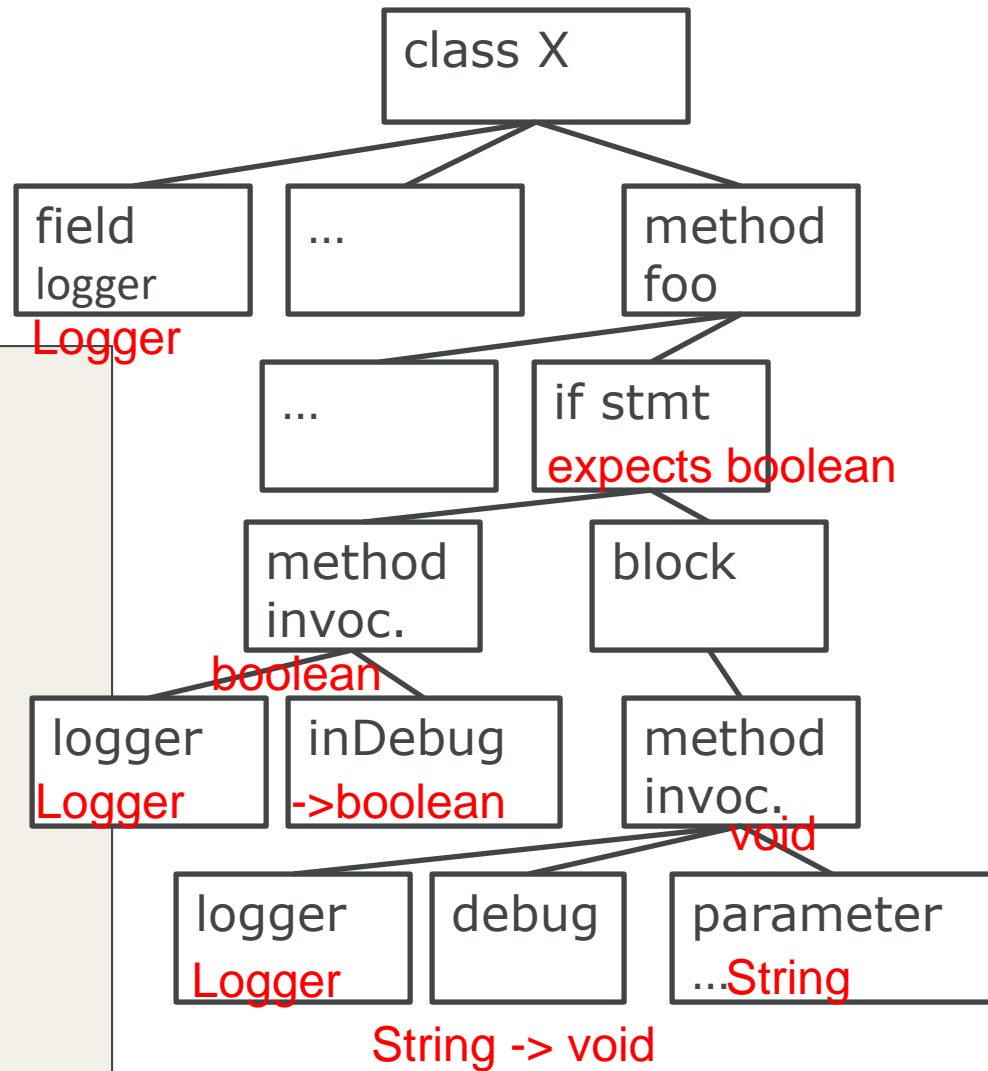
- Tree representation of the syntactic structure of source code.
 - Parsers convert concrete syntax into abstract syntax, and deal with resulting ambiguities.
- Records only the semantically relevant information.
 - Abstract: doesn't represent every detail (like parentheses); these can be inferred from the structure.
- (How to build one? Take compilers!)

- Example: $5 + (2 + 3)$

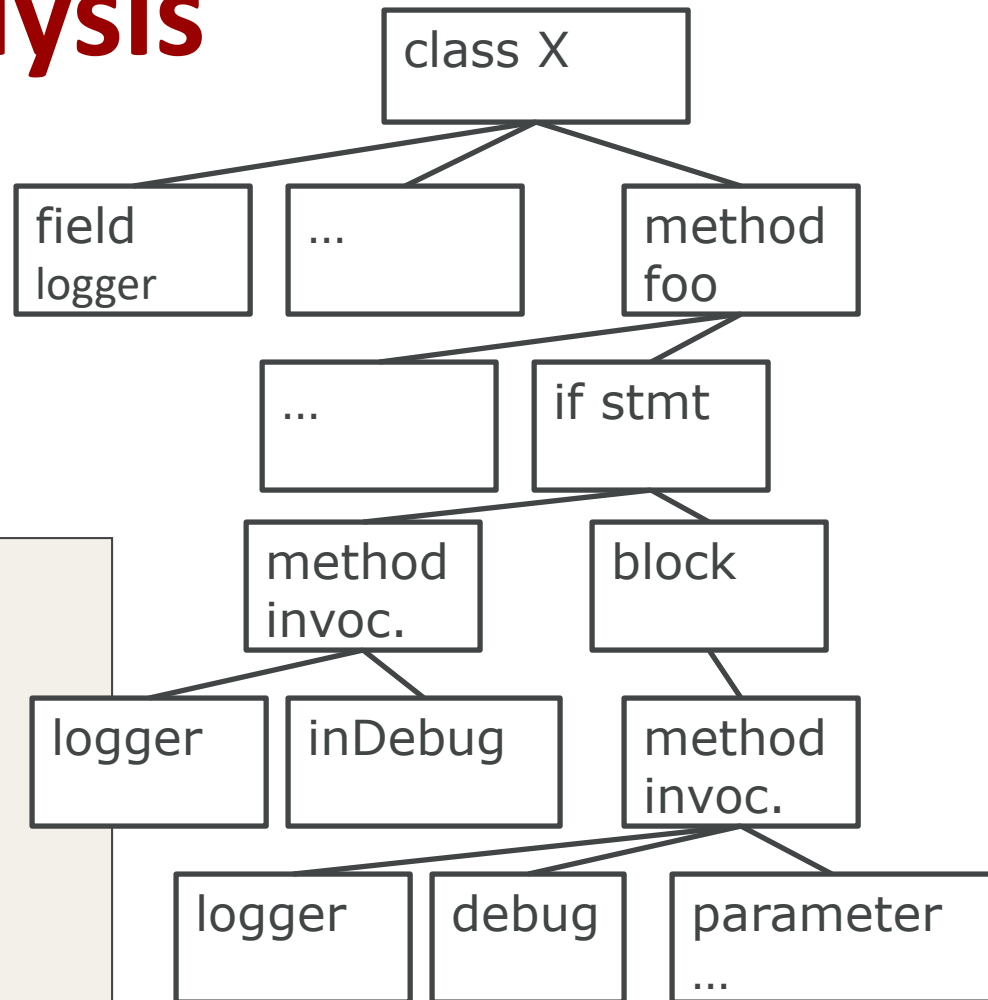


Type checking

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



Structural Analysis



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

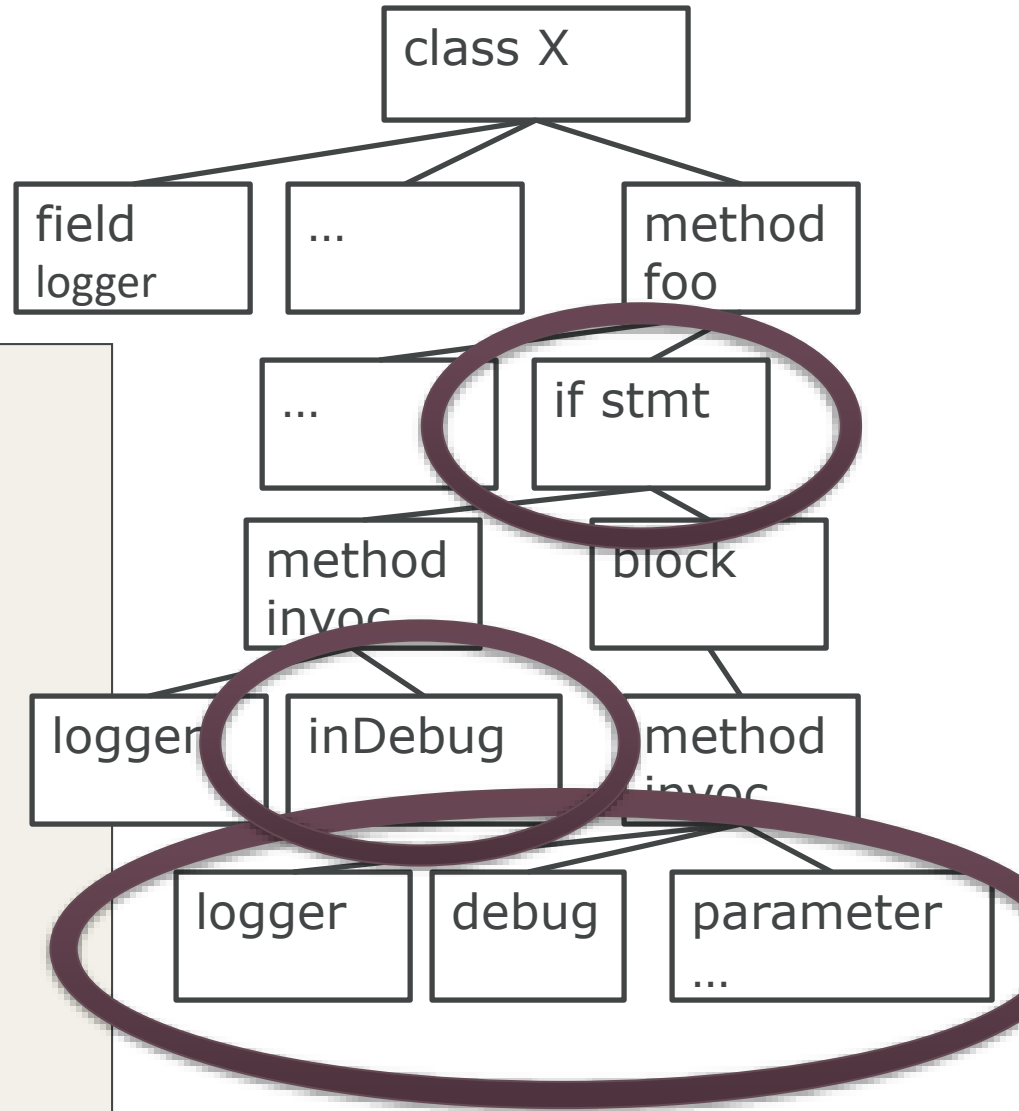
Abstract syntax tree walker

- 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

```

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

```



Bug finding

```
public Boolean decide() {  
    if (computeSomething()==3)  
        return Boolean.TRUE;  
    if (computeSomething()==4)  
        return false;  
    return null;  
}
```

overag History Bug Info Bug Expl



A.java: 69

Navigation

Bug: FBTest.decide() has Boolean return type and returns explicit null

A method that returns either Boolean.TRUE, Boolean.FALSE or null is an accident waiting to happen. This method can be invoked as though it returned a value of type boolean, and the compiler will insert automatic unboxing of the Boolean value. If a null value is returned, this will result in a NullPointerException.

Confidence: Normal, **Rank:** Troubling (14)

Pattern: NP_BOOLEAN_RETURN_NULL

Type: NP, **Category:** BAD_PRACTICE (Bad practice)

Structural Analysis to Detect Goto Fail?

```
1. static OSStatus
2. SSLVerifySignedServerKeyExchange(SSLContext *ctx, bool isRsa,
3.                                  SSLBuffer signedParams,
4.                                  uint8_t *signature,
5.                                  UInt16 signatureLen) {
6.     OSStatus err;
7.     ....
8.     if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
9.         goto fail;
10.    if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
11.        goto fail;
12.        goto fail;
13.    if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
14.        goto fail;
15.    ...
16.fail:
17.    SSLFreeBuffer(&signedHashes);
18.    SSLFreeBuffer(&hashCtx);
19.    return err;
20. }
```

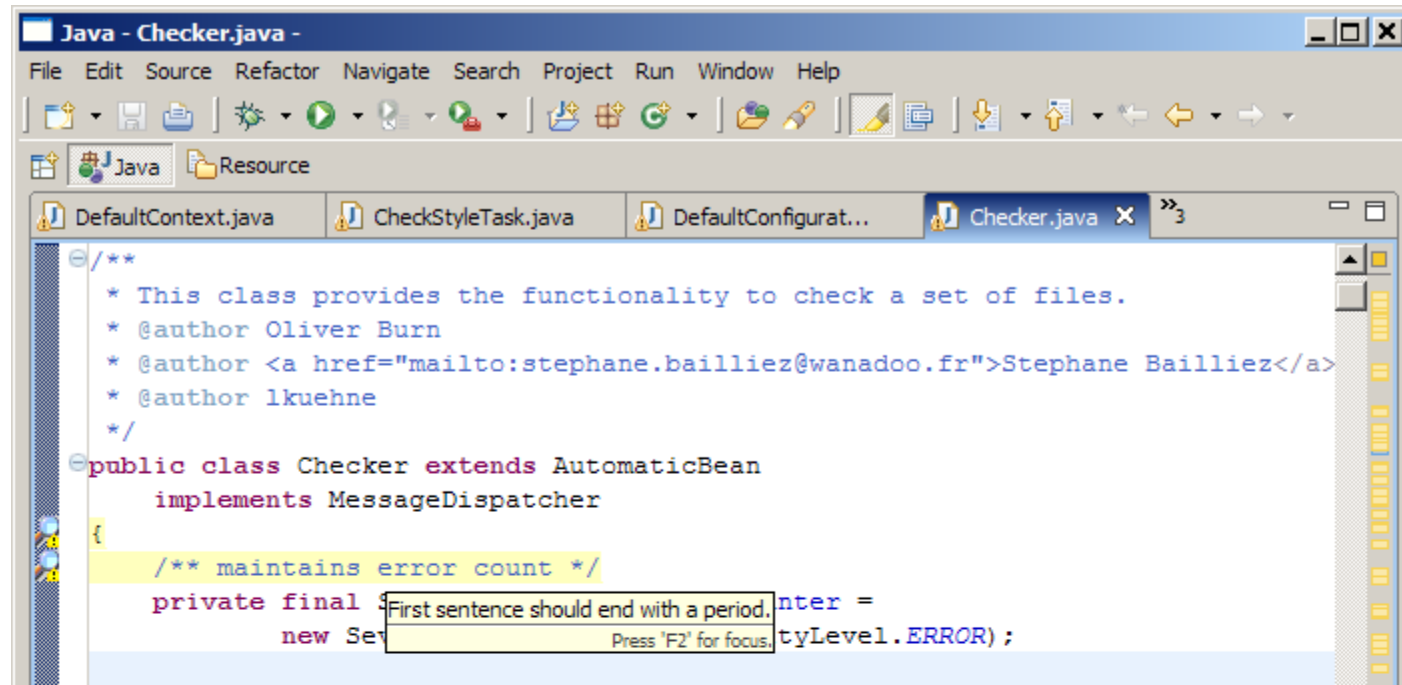
Summary:

Syntactic/Structural Analyses

- Analyzing token streams or code structures (ASTs)
- Useful to find patterns
- Local/structural properties, independent of execution paths

Tools

- Checkstyle
- Many linters (C, JS, Python, ...)
- Findbugs (some analyses)



Tools: Compilers

- Type checking, proper initialization API, correct API usage

Program	Compiler output
<pre>int add(int x,int y) { return x+y; } void main() { add(2); }</pre>	<pre>\$> error: too few arguments to function 'int add(int, int)'</pre>

- Compile at a high warning level
 - `$>gcc -Wall`

CONTROL-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*.
 - Including exception handling, function calls, etc
- 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.

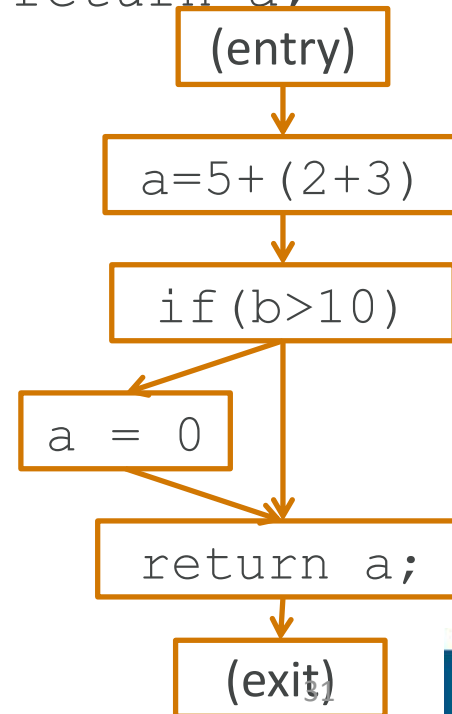
Control/Dataflow analysis

- **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

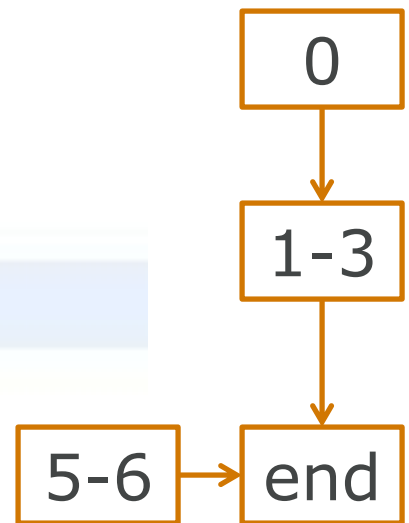
```
1. a = 5 + (2 + 3)
2. if (b > 10) {
3.     a = 0;
4. }
5. return a;
```



More on representation

- Basic definitions:
 - Nodes N – statements of program
 - Edges E – flow of control
 - $\text{pred}(n)$ = set of all predecessors of n
 - $\text{succ}(n)$ = set of all successors of n
 - Start node, set of final nodes (or one final node to which they all flow).
- **Program points:**
 - One program point before each node
 - One program point after each node
 - Join point: point with multiple predecessors
 - Split point: point with multiple successors

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



```

1. /* from Linux 2.3.99 drivers/block/raid5.c */
2. static struct buffer_head *
3. get_free_buffer(struct stripe_head * sh,
4.                int b_size) {
5.     struct buffer_head *bh;
6.     unsigned long flags;
7.     save_flags(flags);
8.     cli(); // disables interrupts
9.     if ((bh = sh->buffer_pool) == NULL)
10.        return NULL;
11.     sh->buffer_pool = bh -> b_next;
12.     bh->b_size = b_size;
13.     restore_flags(flags); // re-enables interrupts
14.     return bh;
15. }

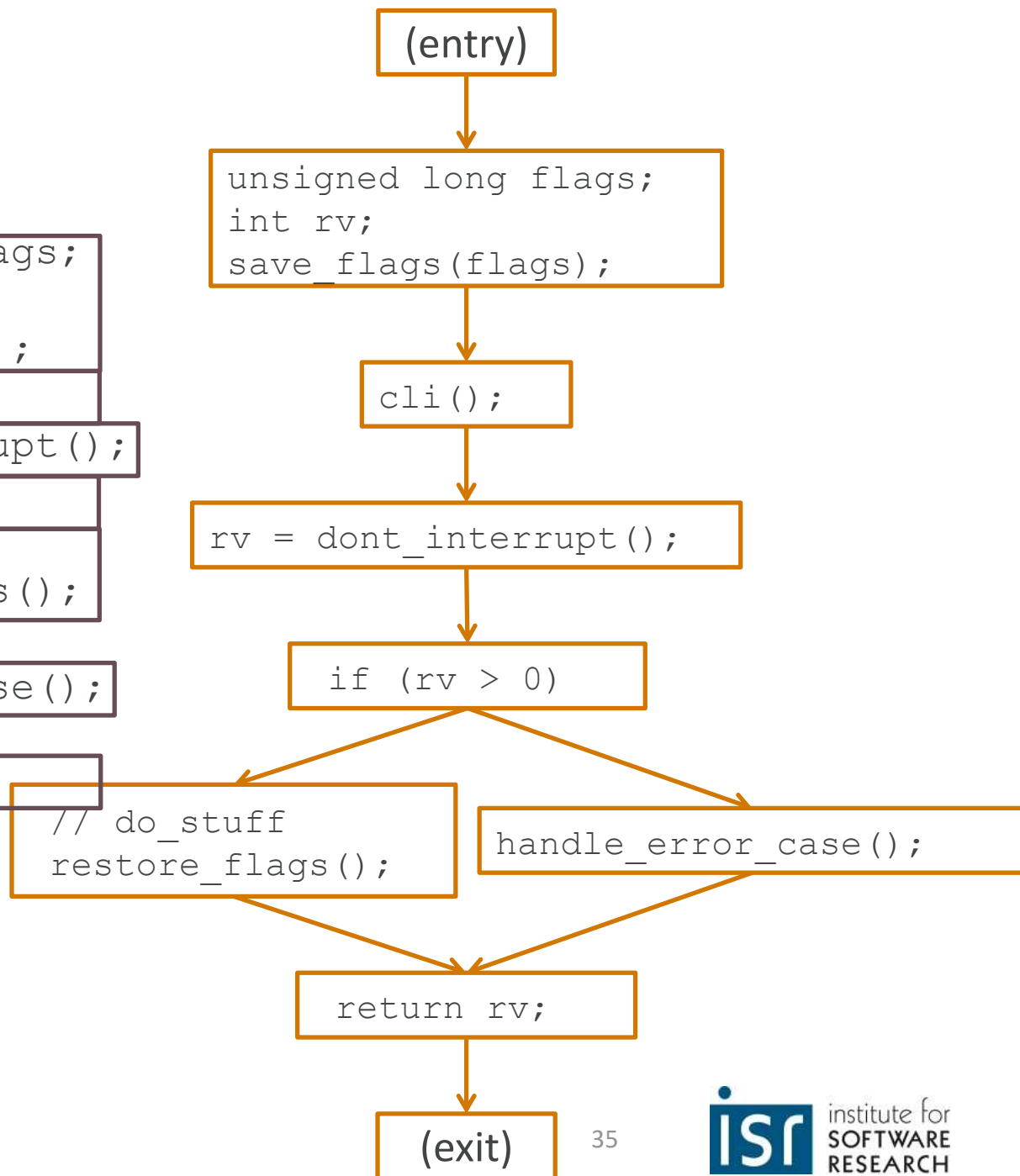
```

Draw control-flow graph
for this function

```

1. int foo() {
2.     unsigned long flags;
3.     int rv;
4.     save_flags(flags);
5.     cli();
6.     rv = dont_interrupt();
7.     if (rv > 0) {
8.         // do_stuff
9.         restore_flags();
10.    } else {
11.        handle_error_case();
12.    }
13.    return rv;
14. }

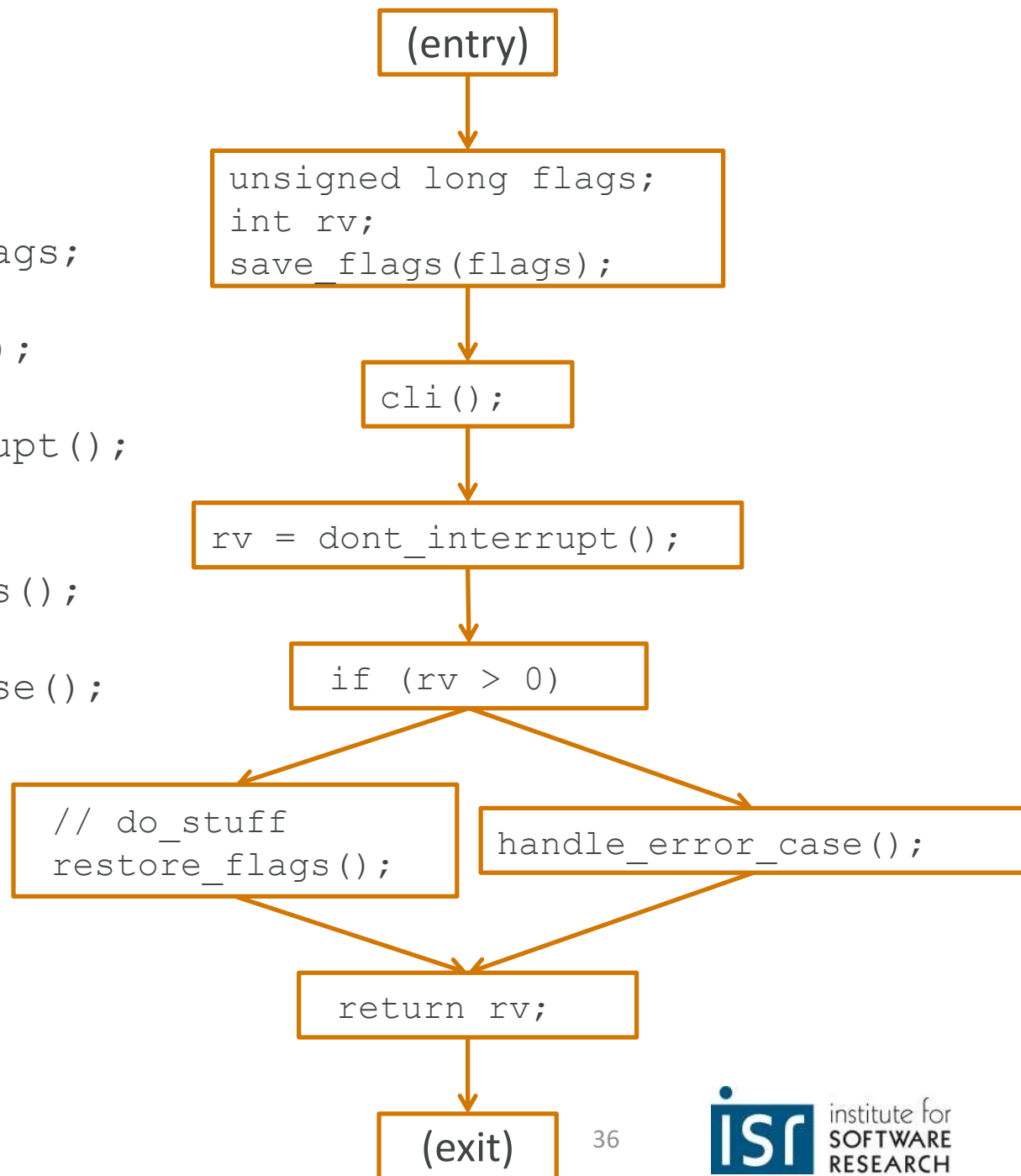
```



```

1.  int foo() {
2.      unsigned long flags;
3.      int rv;
4.      save_flags(flags);
5.      cli();
6.      rv = dont_interrupt();
7.      if (rv > 0) {
8.          // do_stuff
9.          restore_flags();
10.     } else {
11.         handle_error_case();
12.     }
13.     return rv;
14. }

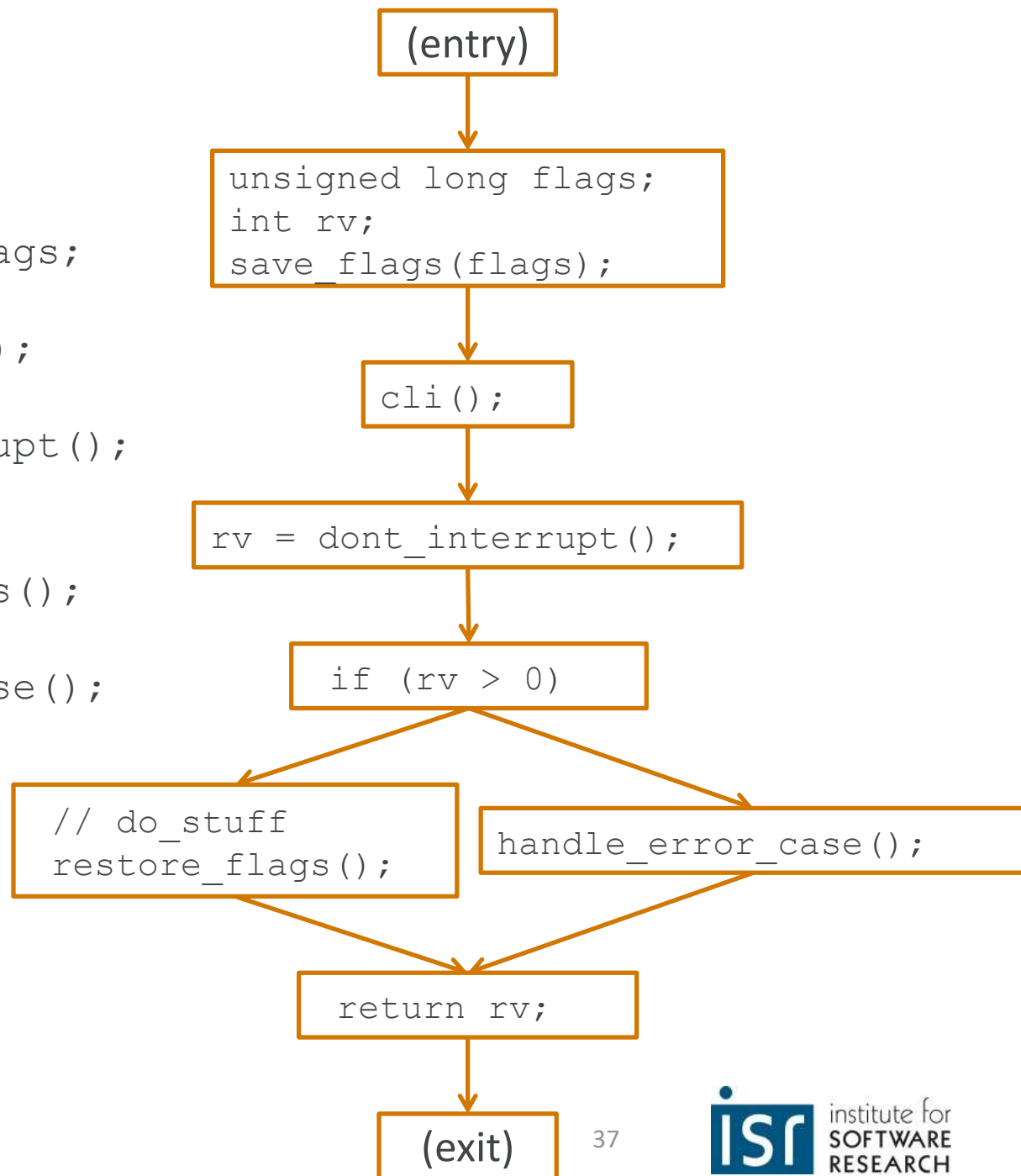
```



```

1.  int foo() {
2.      unsigned long flags;
3.      int rv;
4.      save_flags(flags);
5.      cli();
6.      rv = dont_interrupt();
7.      while (rv > 0) {
8.          // do_stuff
9.          restore_flags();
10.     } else {
11.         handle_error_case();
12.     }
13.     return rv;
14. }

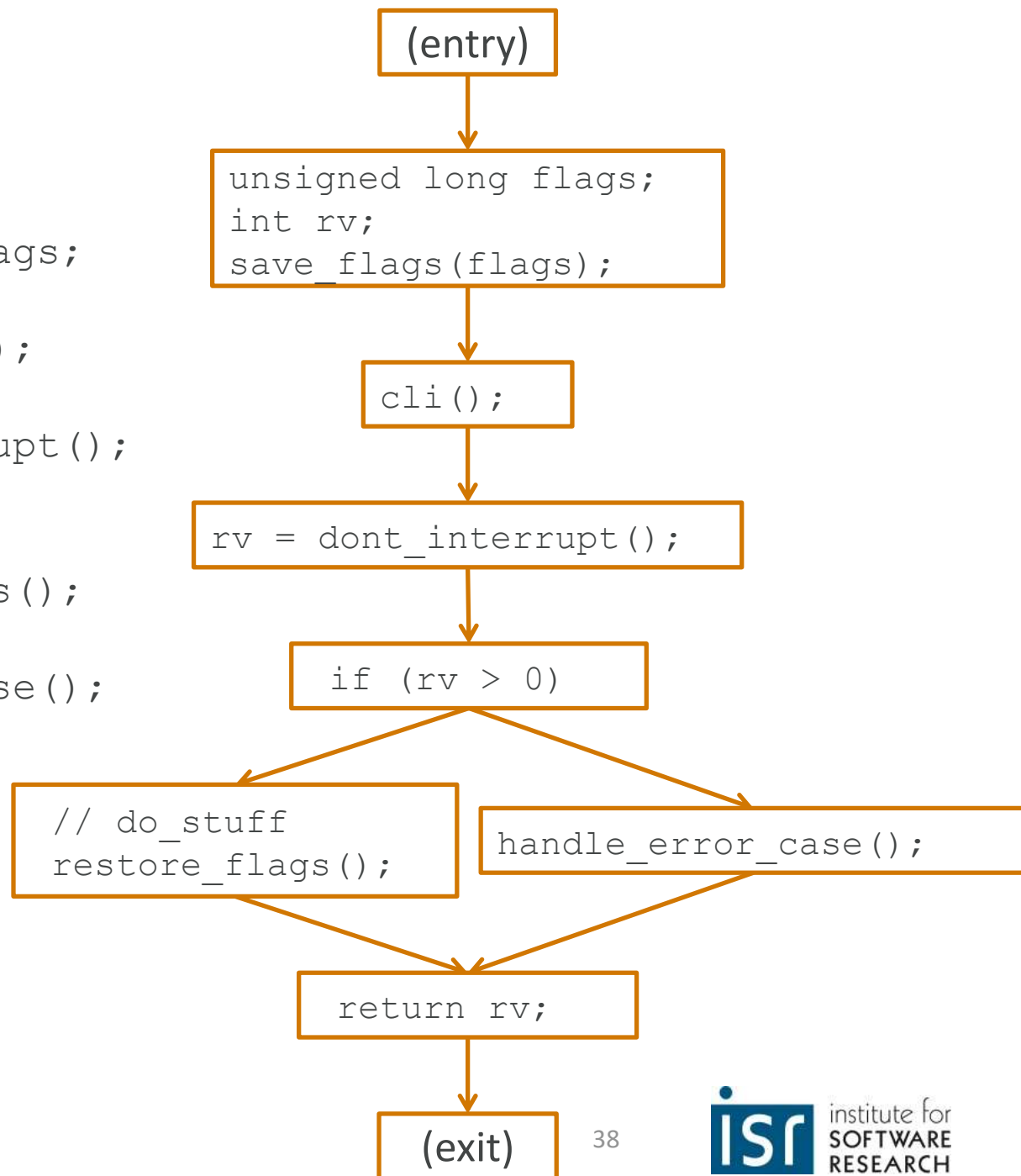
```



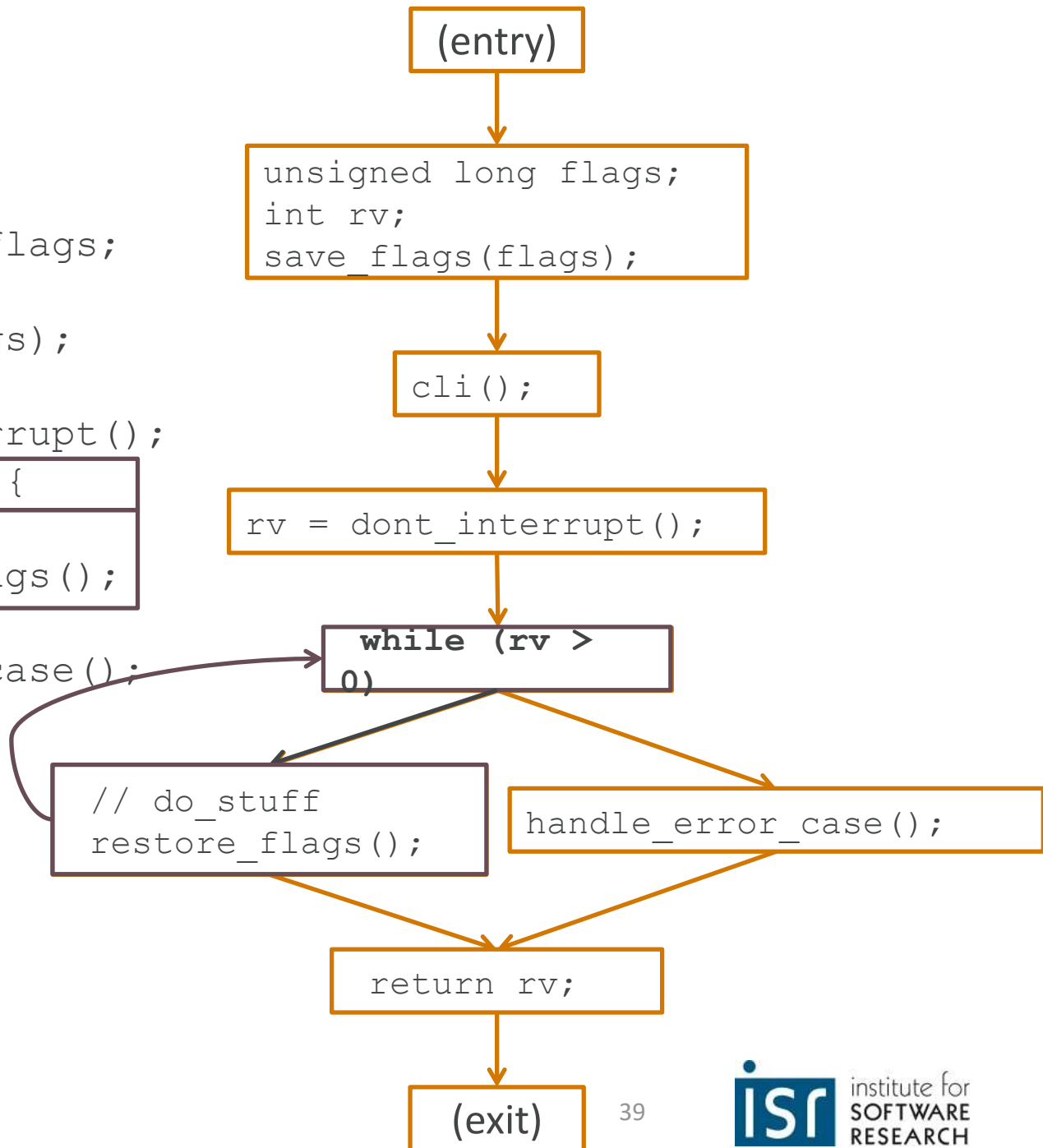
```

1.  int foo() {
2.      unsigned long flags;
3.      int rv;
4.      save_flags(flags);
5.      cli();
6.      rv = dont_interrupt();
7.      while (rv > 0) {
8.          // do_stuff
9.          restore_flags();
10.     } else {
11.         handle_error_case();
12.     }
13.     return rv;
14. }

```



```
1. int foo() {
2.     unsigned long flags;
3.     int rv;
4.     save_flags(flags);
5.     cli();
6.     rv = dont_interrupt();
7.     while (rv > 0) {
8.         // do_stuff
9.         restore_flags();
10.    }
11.    handle_error_case();
12.
13.    return rv;
14. }
```



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.

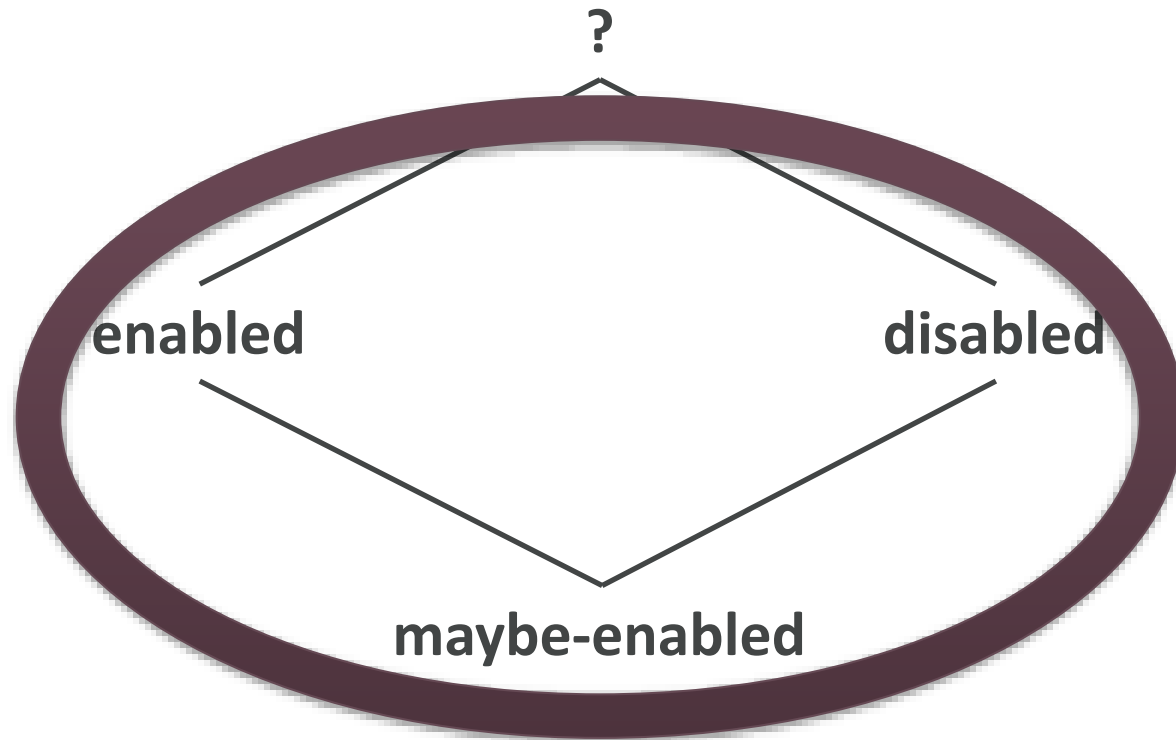
Abstract domain: lattices

- Lattice $D = (S, r)$
 - D is domain of program properties
 - S is a (possibly infinite) set of elements. Must contain unique *largest* (top) and *smallest* elements (bottom).
 - r is a binary relation over elements of S
- Required properties for r :
 - Is a partial order (reflexive, transitive, and anti-symmetric)
 - Every pair of elements has a unique **greatest lower bound** (*meet*) and a unique **least upper bound** (*join*)

Say wha?

- We are tracking all possible values related to a property of interest at every program point.
- Possible values---the information we're tracking---modeled as an element of the lattice that defines the domain.
- Use the lattice to compute information, by building constraints that describe how the information changes through the program:
 - **Transfer function:** Effect of instructions on state
 - **Meet/join:** effect of control flow

Abstract Domain: interrupt checker



Reasoning about a CFG

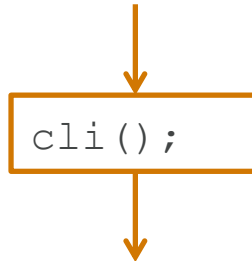
- Analysis updates state at *program points*: points between nodes.
- For each node:
 - determine state on entry by examining/combining state from predecessors.
 - evaluate state on exit of node based on effect of the operations (*transfer*).
- *Iterate through successors and over entire graph until the state at each program point stops changing.*
- **Output: state at each program point**

An interrupt checker

- **Abstraction**
 - Three abstract states: enabled, disabled, maybe-enabled
 - Warning if we can reach the end of the function with interrupts disabled.
- **Transfer function:**
 - If a basic block includes a call to `cli()`, then it moves the state of the analysis from **disabled** to **enabled**.
 - If a basic block includes a call to `restore_flags()`, then it moves the state of the analysis from **enabled** to **disabled**.

Transfer function

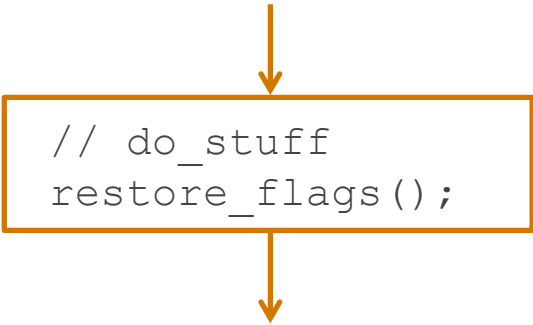
assume: pre-block program point: interrupts disabled



post-block program point: interrupts enabled

Transfer function

assume: pre-block program point: interrupts enabled



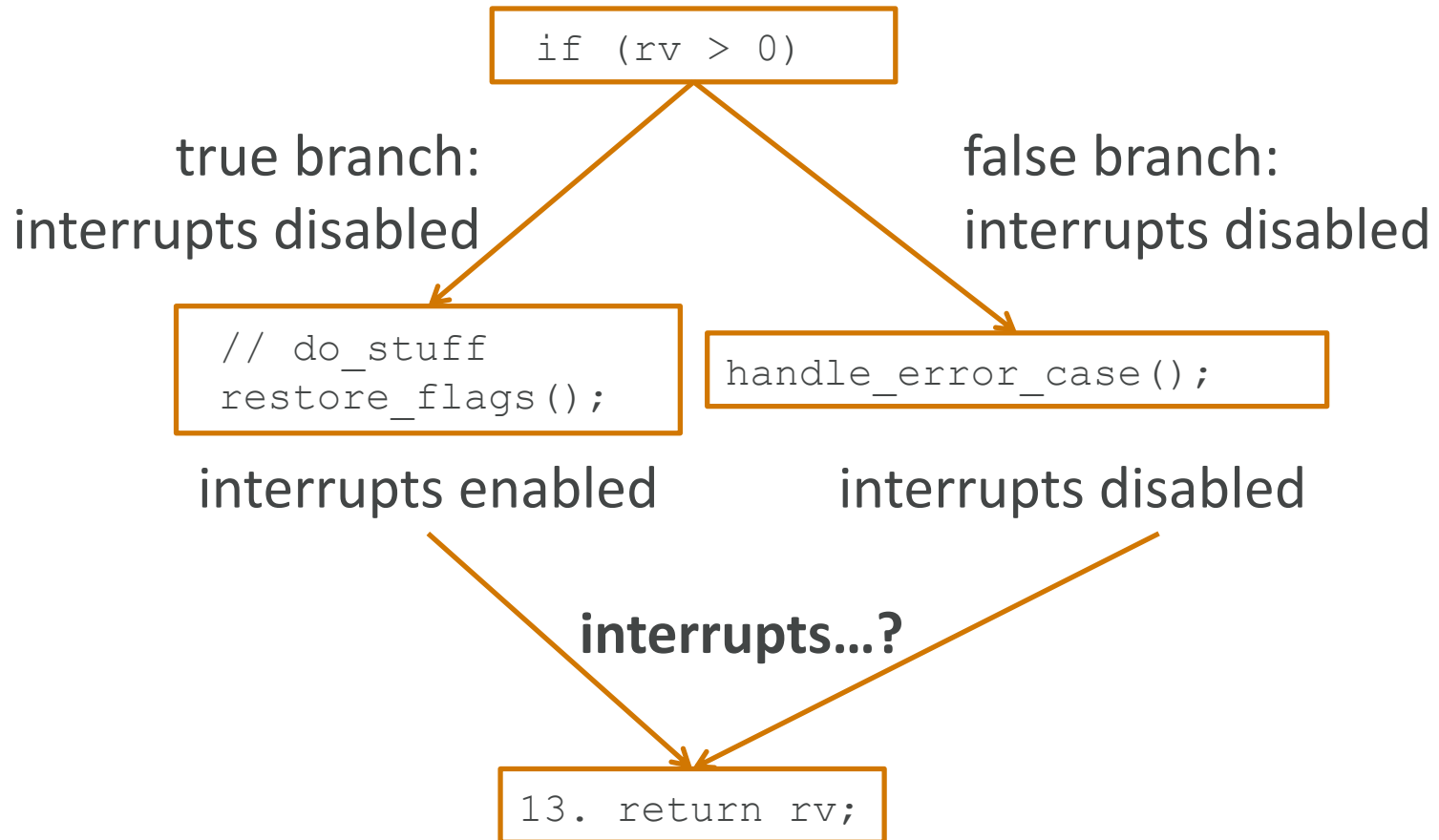
```
// do_stuff  
restore_flags();
```

post-block program point: interrupts disabled

(Note that, in graphs, I leave out some intermediate program points when they're not interesting; you'll see what I mean in a second.)

Join

assume: pre-block program point: interrupts disabled



Join/branching

- What to do with information that comes to/from multiple previous states?
- When we get to a branch, what should we do?
 1. explore each path separately
 - Most exact information for each path
 - But—how many paths could there be?
 - Leads to state explosion, loops add an infinity problem. join paths back together
 2. Join!
 - Less exact, loses information (...Rice's theorem...)
 - But no state explosion, and terminates (more in a bit)
- Not just conditionals!
 - Loops, switch, and exceptions too!

Interrupt analysis: join function

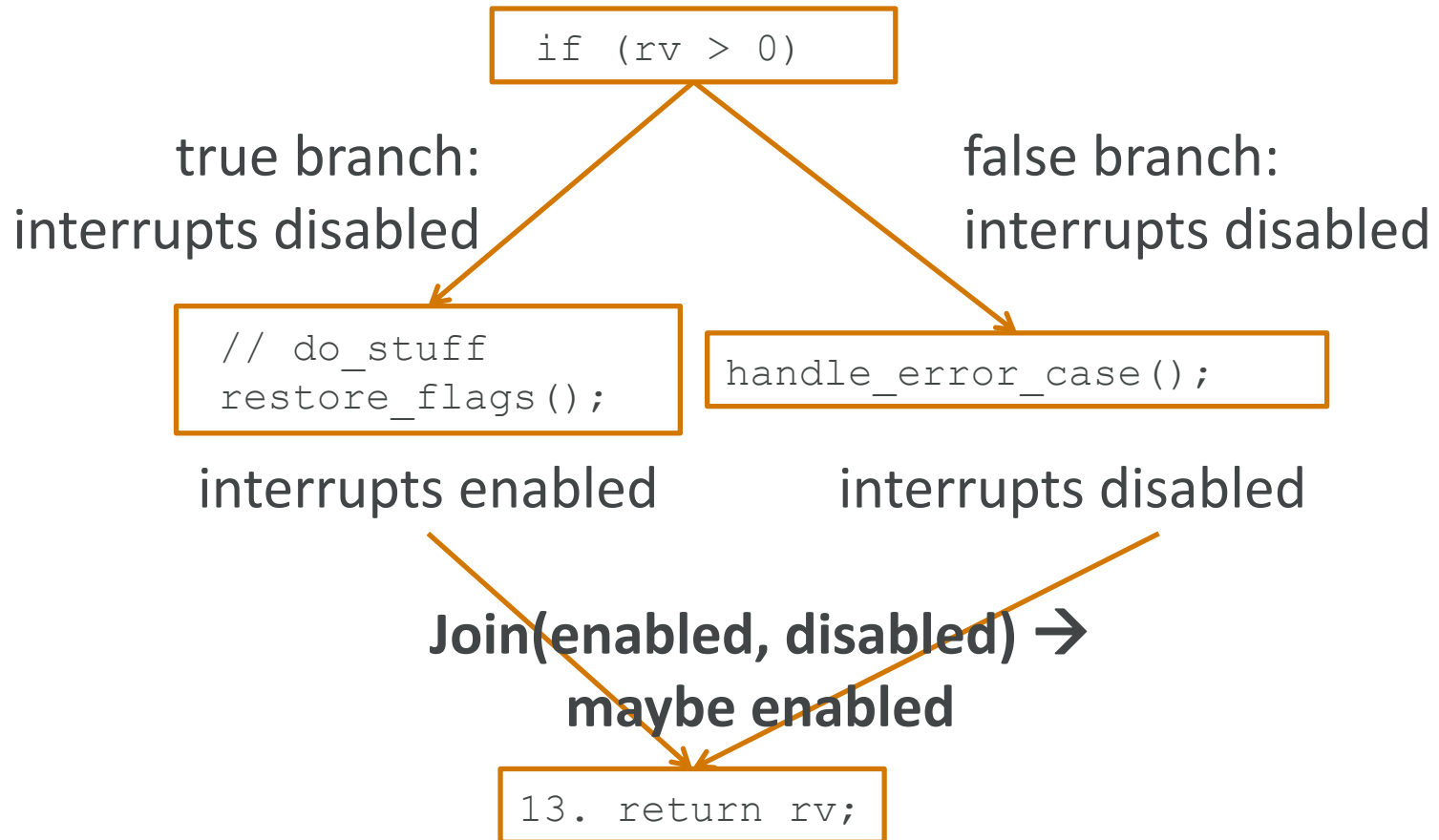
- Abstraction
 - 3 states: enabled, disabled, maybe-enabled
 - Program counter
- **Join:** If at least one predecessor to a basic block has interrupts enabled and at least one has them disabled...

Join

- $\text{Join}(\text{enabled}, \text{enabled}) \rightarrow \text{enabled}$
- $\text{Join}(\text{disabled}, \text{disabled}) \rightarrow \text{disabled}$
- $\text{Join}(\text{disabled}, \text{enabled}) \rightarrow \text{maybe-enabled}$
- $\text{Join}(\text{maybe-enabled}, *) \rightarrow \text{maybe-enabled}$

Join: abstract!

assume: pre-block program point: interrupts disabled

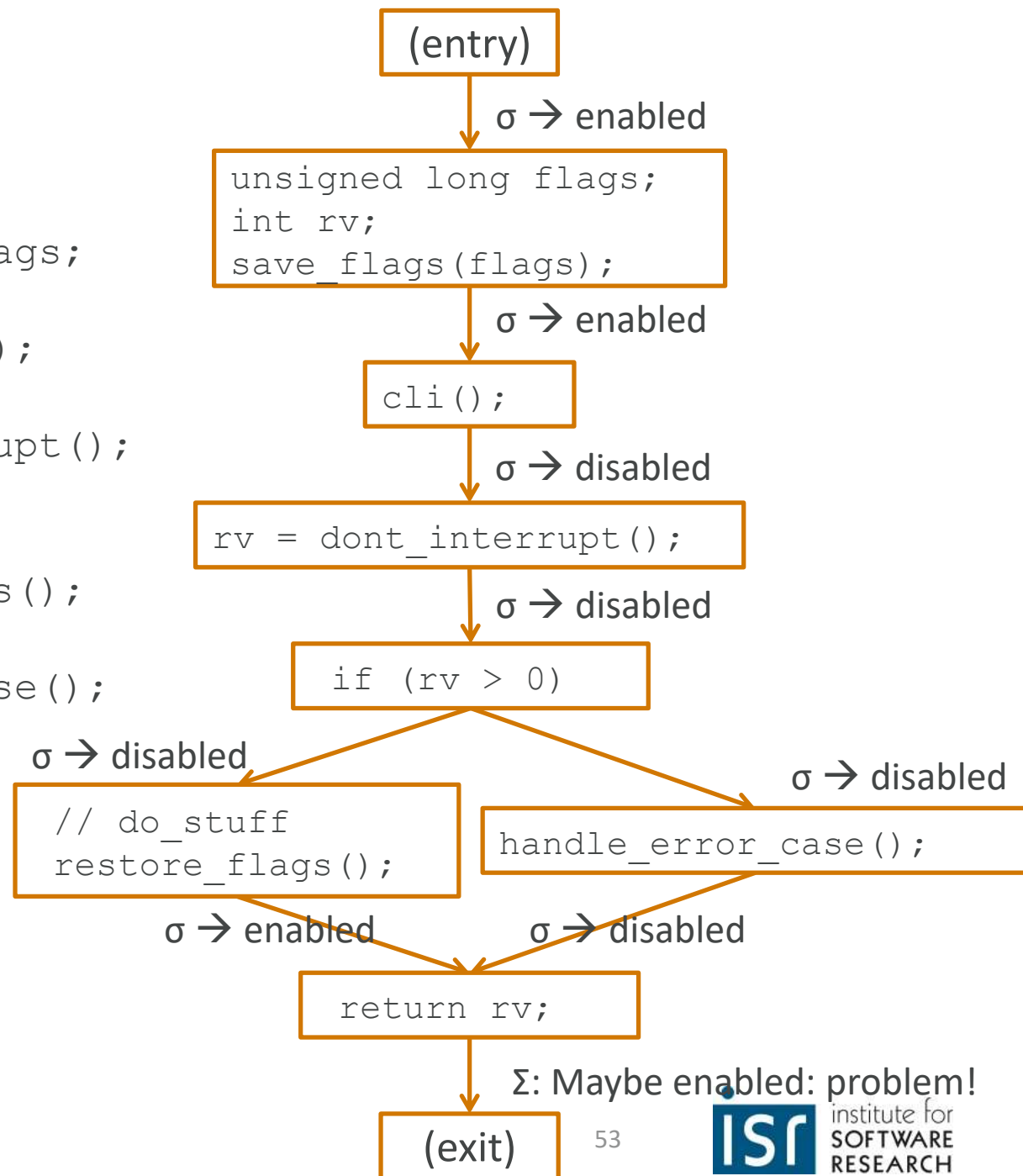


(Note: this is where information gets “lost.”)

```

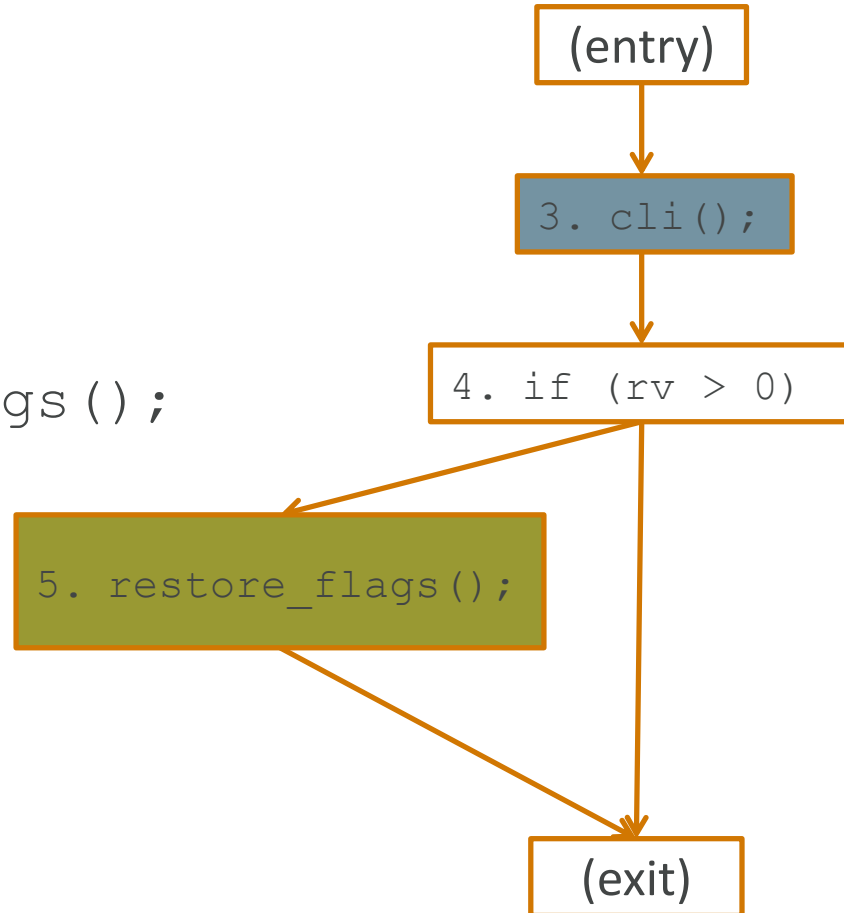
1.  int foo() {
2.      unsigned long flags;
3.      int rv;
4.      save_flags(flags);
5.      cli();
6.      rv = dont_interrupt();
7.      if (rv > 0) {
8.          // do_stuff
9.          restore_flags();
10.     } else {
11.         handle_error_case();
12.     }
13.     return rv;
14. }

```



Abstraction

```
1. void foo() {  
2.     ...  
3.     cli();  
4.     if (a) {  
5.         restore_flags();  
6.     }  
7. }
```



Tools

- Dead-code detection in many compilers (e.g. Java)
- Instrumentation for dynamic analysis before and after decision points; loop detection
- Decompilation

DATA-FLOW ANALYSIS

Data- vs. control-flow

- Dataflow: tracks abstract values for each of (some subset of) the **variables** in a program.
- Control flow: tracks state **global** to the function in question.

Example:

Zero/Null-pointer Analysis

- Could a variable x ever be 0?
 - (what kinds of errors could this check for?)
- Original domain: N maps every variable to an integer.
- Abstraction: every variable is non zero (NZ), zero (Z), or maybe zero (MZ)

Zero analysis transfer

- What operations are relevant?

Zero analysis join (per variable)

- $\text{Join}(\text{zero}, \text{zero}) \rightarrow \text{zero}$
- $\text{Join}(\text{not-zero}, \text{not-zero}) \rightarrow \text{not-zero}$
- $\text{Join}(\text{zero}, \text{not-zero}) \rightarrow \text{maybe-zero}$
- $\text{Join}(\text{maybe-zero}, *) \rightarrow \text{maybe-zero}$

Example

- Consider the following program:

```
x = 10;  
y = x;  
z = 0;  
while (y > -1) {  
    x = x/y;  
    y = y-1;  
    z = 5;  
}
```

- Use **zero analysis** to determine if y could be zero at the division.

Reminder:

x: Join(NZ,NZ) \rightarrow NZ

y: Join(MZ,NZ) \rightarrow MZ

Z: Join(NZ, Z) \rightarrow MZ

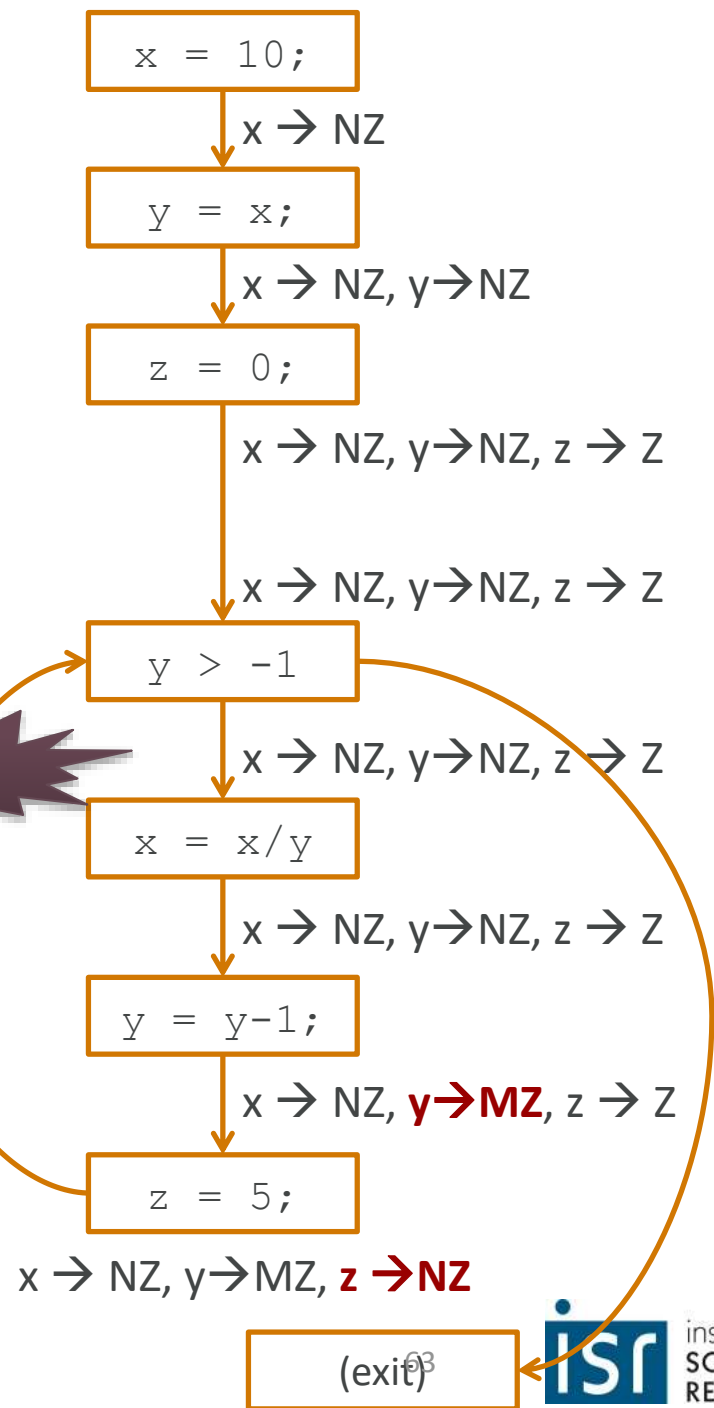
Reminder:

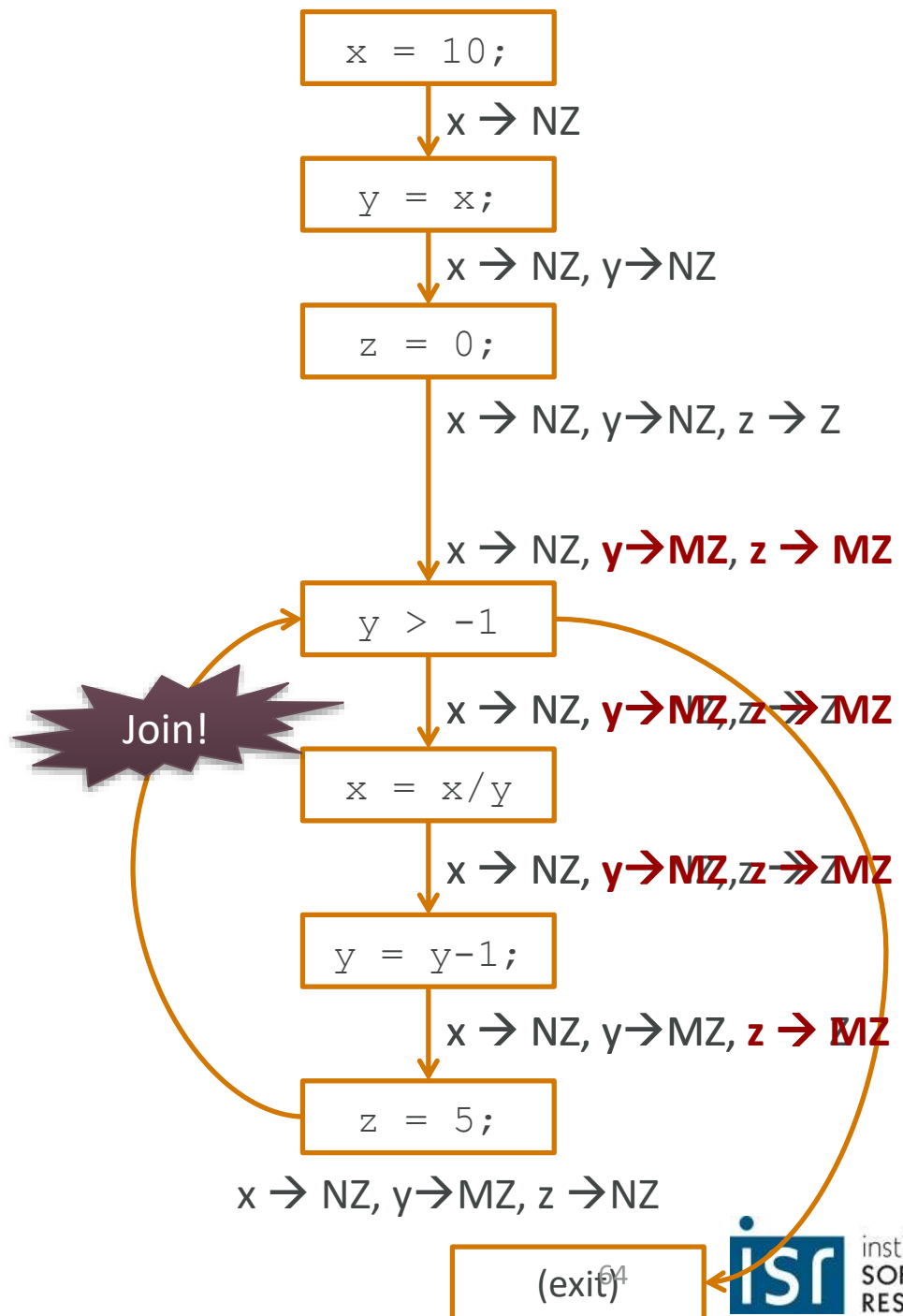
x: Join(NZ,NZ) \rightarrow NZ

y: Join(MZ,NZ) \rightarrow MZ

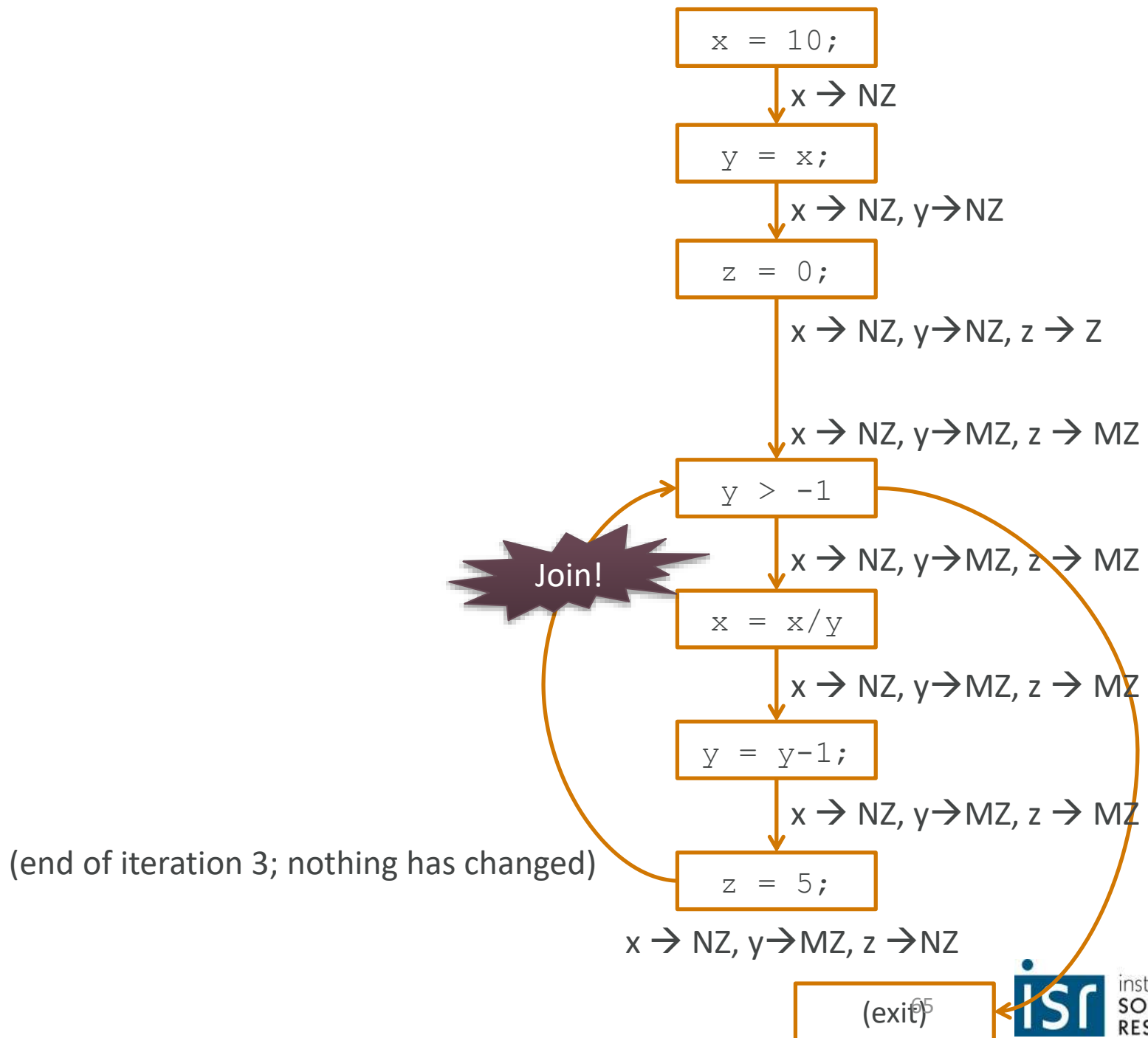
z: Join(NZ, Z) \rightarrow MZ

```
x = 10;  
y = x;  
z = 0;  
while (y > -1) {  
    x = x/y;  
    y = y-1;  
    z = 5;  
}
```

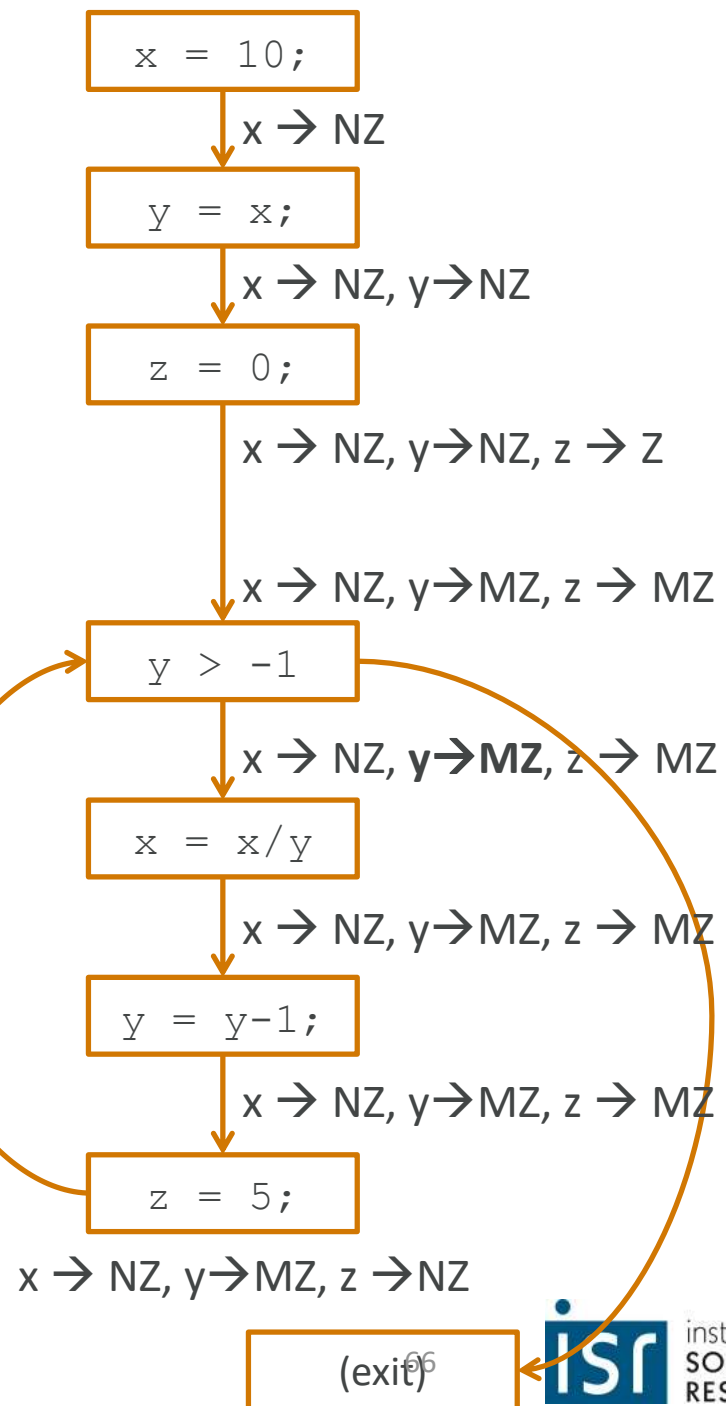




(end of iteration 2)



Warning! Possible division by zero error!



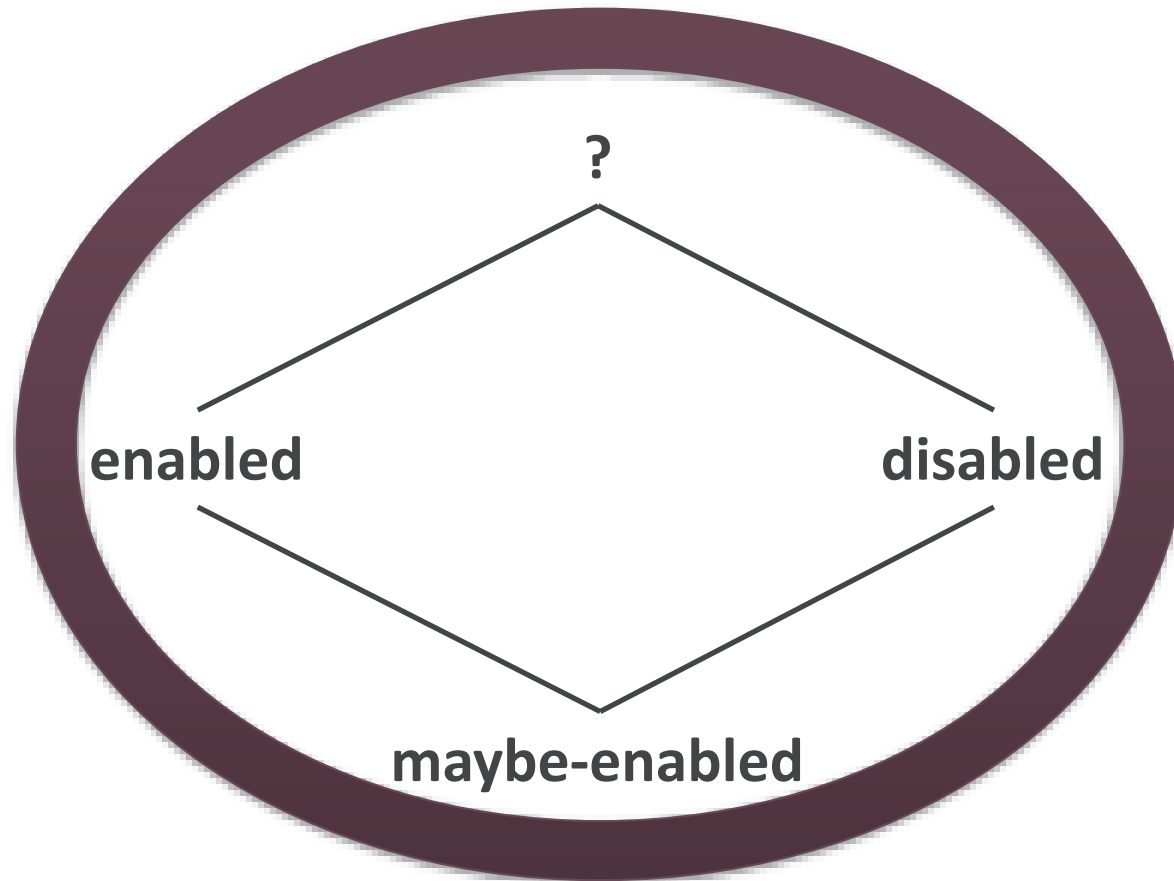
Abstraction at work

- Number of possible states gigantic
 - n 32 bit variables results in 2^{32*n} states
 - $2^{(32*3)} = 2^{96}$
 - With loops, states can change indefinitely
- Zero Analysis narrows the state space
 - Zero or not zero
 - $2^{(2*3)} = 2^6$
 - When this limited space is explored, then we are done
 - Extrapolate over all loop iterations

Order doesn't actually matter

- Can process instructions in whatever order we want, until the information doesn't change over the whole program.
- Use bottom of the lattice (?) as initial value of all uncomputed states

Example: interrupt checker



Termination intuition

- A **fixed point** of a function is a data value v that a function maps to itself:
 - $f(v) = v$
- The flow function is the mathematical function.
- The dataflow analysis state at each fix point is the data values.

Simple algorithm

```
1. for all node indexes i do
2.   input[i] = ?
3. input[ firstInstruction ] = initialA
4. while not at fixed point
5.   pick an instruction i
6.   output = flow(i, input[i])
7.   for j in succs ( i )
8.     input[j] = input[j] join output
```

Example of Worklist

1. `[a := 0]`
2. `[b := 0]`
3. `while [a < 2] do`
4. `[b := a];`
5. `[a := a + 1];`
6. `[a := 0]`

1. `for all node indexes i do`
2. `input[i] = ?`
3. `input[firstInstruction] =`
 `initialA`
4. `while not at fixed point`
5. `pick an instruction i`
6. `output = flow(i, input[i])`
7. `for j in succs (i)`
8. `input[j] = input[j] t output`

Kildall's Worklist Algorithm

```
1. worklist = new Set();
2. for all node indexes i do
3.   input[i] = ?A;
4. input[entry] = initialA;
5. worklist.add(all nodes);
6. while (!worklist.isEmpty()) do
7.   i = worklist.pop();
8.   output = flow(input[i], i);
9.   for j ∈ succ(i) do
10.    if ! (output v input[j])
11.      input = input[j] join output
12.      worklist.add(j)
```

Note on line 5: it's OK to just add entry to worklist if the flow functions cannot return bottom, which is true for our example but not generally.

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)

Why? Infinite loops.

- I have a program, and it takes input.
- That program is written in a reasonable programming language, so it has loops.
- One way a program with loops can go horribly awry is that it can loop infinitely.
- It's often hard to tell the difference between a program that just takes a long time to execute, and a program that's stuck in an infinite loop.

Computability theory says...

- **Halting problem:** the problem of determining whether a given program will halt/terminate on a given input.
- A *general* algorithm that solves this problem is impossible.
 - More specifically: it's undecidable (it's possible to get a *yes* answer, but not a *no* answer).
 - (sometimes you can use heuristics, but solving it generally for all programs is still out.)
- The proof here is very elegant. But trust me: this problem is extremely impossible.

OK, so?

- If you could always statically tell if any program had a non-trivial property (never dereferences null, always releases all file handles, etc, etc), you could also generally solve the halting problem.
- ...but the halting problem is *definitely* impossible.
- So: no static analysis is perfect. They will always have false positives or false negatives (or both).
- *All tools make tradeoffs.*

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

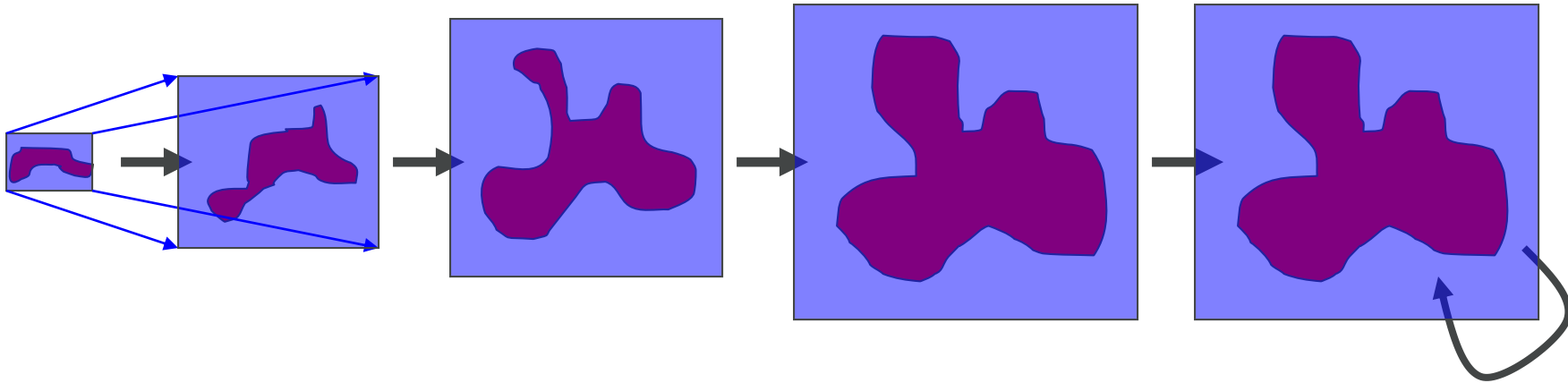
Sound Analysis:

- reports all defects
- > no false negatives
- typically overapproximated

Complete Analysis:

- every reported defect is an actual defect
- > no false positives
- typically underapproximated

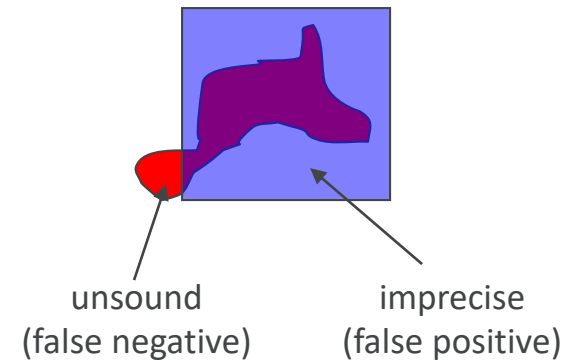
Soundness and precision



Program state covered in actual execution



Program state covered by abstract execution with analysis



Sound vs. Heuristic Analysis

- Heuristic Analysis
 - FindBugs, coverity, ...
 - Follow rules, approximate, avoid some checks to reduce false positives
 - May report false positives and false negatives
- Sound Static Analysis
 - Type checking, Not-Null, ... (specific fault classes)
 - Sound abstraction, precise analysis to reduce false positives

Exercise:

Null pointers

```
1.int foo() {  
2.    Integer x = new Integer(6);  
3.    Integer y = bar();  
4.    int z;  
5.    if (y != null)  
6.        z = x.intVal() + y.intVal();  
7.    } else {  
8.        z = x.intVal();  
9.        y = x;  
10.       x = null;  
11.    }  
12.    return z + x.intVal();  
13.}
```

```
Integer x = new Integer(6);
```

```
Integer y = bar();
```

```
int z;  
if (y != null)
```

```
z = x.intVal() +  
y.intVal();
```

```
z = x.intVal();  
y = x;  
x = null;
```

```
return z + x.intVal();
```

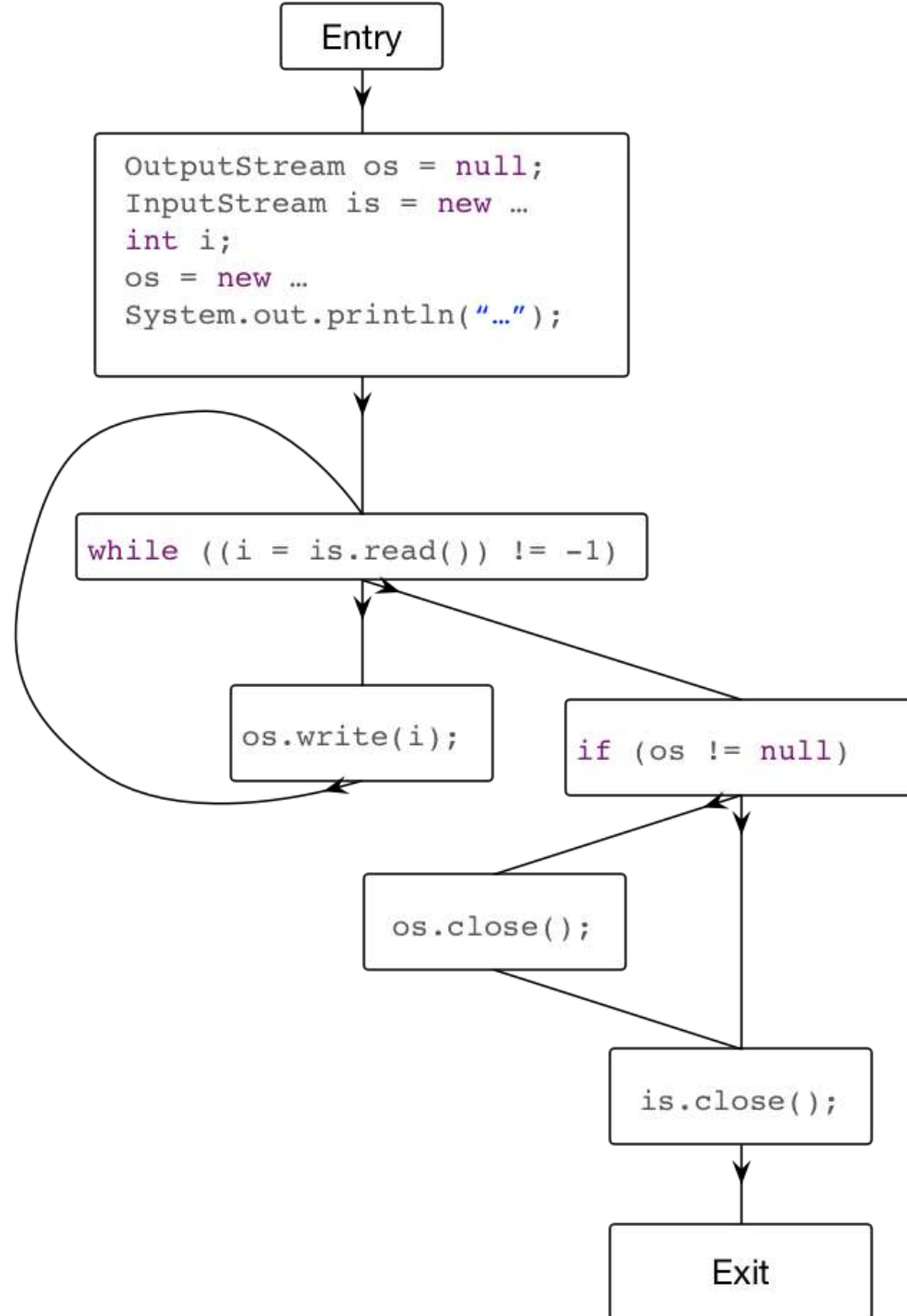
What about that function call?

1. If you're worried about totally wacky control flow (exceptions, longjumps), they can be modeled in wackier/more complicated control flow graphs.
2. Ignore it by assuming that all functions return and tempering your claim:
“assuming the program terminates, the analysis soundly computes...”
 - Most people don't bother; this is basically assumed.

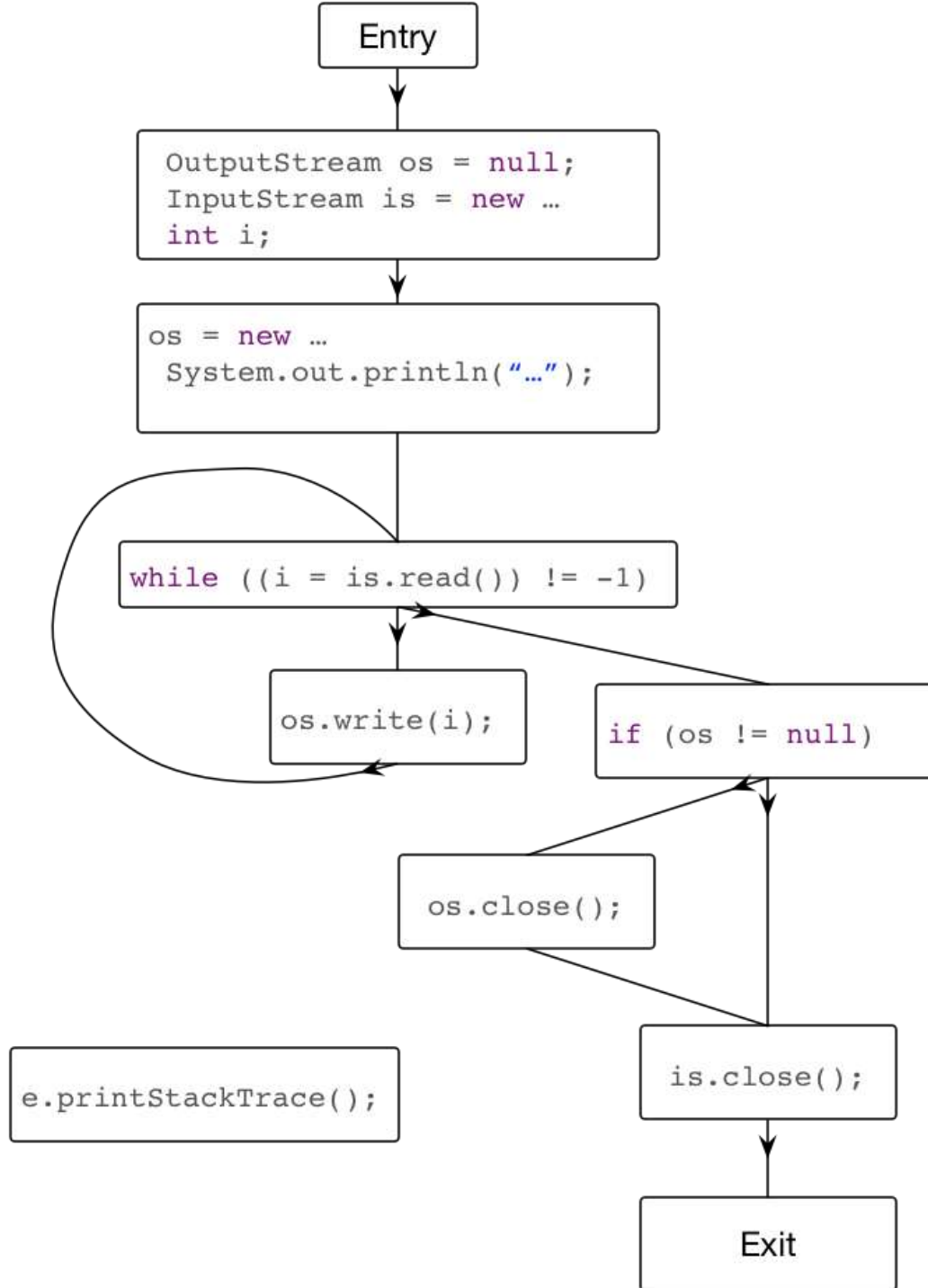
Exercise: File open/close

```
1. public class StreamDemo {
2.     public static void main(String[] args) throws Exception {
3.         OutputStream os = null;
4.         InputStream is = new FileInputStream("in.txt");
5.         int i;
6.         try {
7.             os = new FileOutputStream("out.txt");
8.             System.out.println("Copying in progress...");
9.             while ((i = is.read()) != -1) {
10.                 os.write(i);
11.             }
12.             if (os != null) {
13.                 os.close();
14.             }
15.         } catch (IOException e) {
16.             e.printStackTrace();
17.         }
18.         is.close();
19.     }
20. }
```

Try- Catch?



Try-Catch?



Design choices: representation and abstract domain

- What if we don't model the try/catch?
- If we do...how should we include it?
- ...what about non-IOExceptions?
- Broader question: How precisely should we model semantics?
 - E.g., Of instructions, of conditional checks, etc.

Upshot: analysis as approximation

- Analysis must approximate in practice
 - False positives: may report errors where there are really none
 - False negatives: may not report errors that really exist
 - All analysis tools have either false negatives or false positives
- Approximation strategy
 - Find a pattern P for correct code
 - which is feasible to check (analysis terminates quickly),
 - covers most correct code in practice (low false positives),
 - which implies no errors (no false negatives)
- Analysis can be pretty good in practice
 - Many tools have low false positive/negative rates
 - A sound tool has no false negatives
 - Never misses an error in a category that it checks

Tools

- Most commercial “static analysis tools”, bug detectors, incl. FindBugs
- Examples: Nullness, atomicity, information flow, ...
- Many compiler optimizations...

Beyond a single method

- Interprocedural analyses challenging to scale
- Build single big graph or abstract at method level; often manual annotations to help

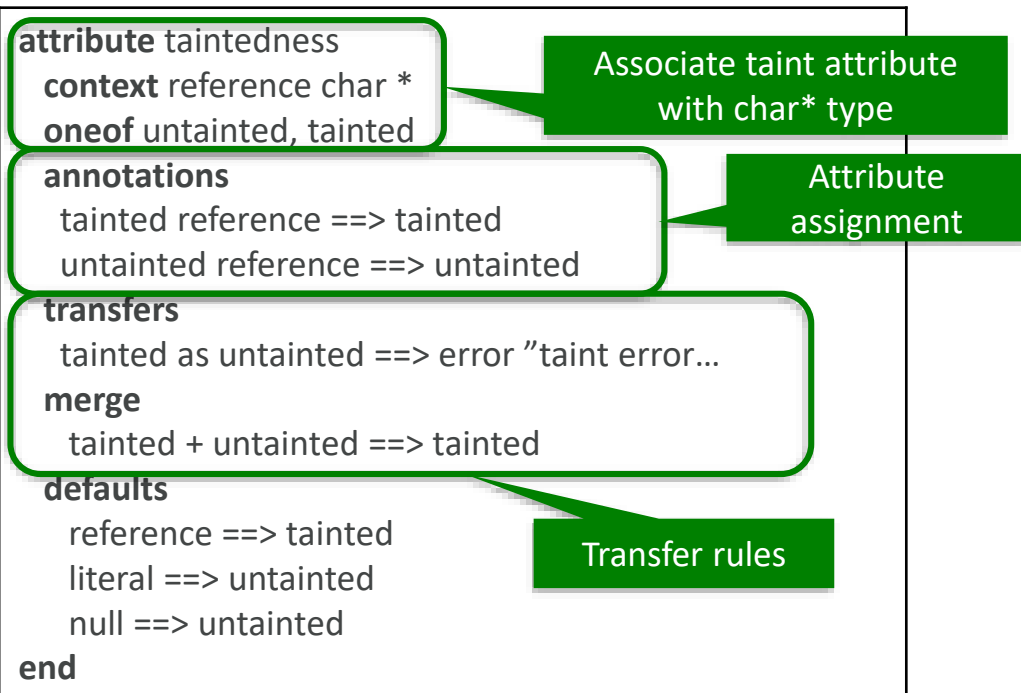
Splint Example

Code (ex.c)	Splint output
<pre>int main() { char c; while (c != 'x'); { c = getchar(); if (c = 'x') return 1; } return 0; }</pre>	<p>\$> splint ex.c</p> <p>Splint 3.1.1 --- 19 Jul 2006</p> <p>ex.c:3:10: Variable c used before definition. An rvalue is used that may not be initialized to a value on some execution path. (Use -usedef to inhibit warning)</p> <p>ex.c:3:10: Suspected infinite loop. No value used in loop test (c) is modified by test or loop body. This appears to be an infinite loop. Nothing in the body of the loop or the loop test modifies the value of the loop test. Perhaps the specification of a function called in the loop body is missing a modification. (Use -infloops to inhibit warning)</p> <p>ex.c:5:5: Assignment of int to char: c = getchar() To make char and int types equivalent, use +charint.</p> <p>...</p>

Extending Splint to Analyze Taintedness

- Tainting marks data as untrusted
 - Tainted data originates from the user/external environment
 - Mark (taint) data as untrusted and analyze program to determine how/where it is used
- We can extend splint to analyze taintedness at compile time

Tainted character pointers



Using the new definition in annotations

```
int printf (/*@untainted@*/ char *fmt,
...);
```

Summary

- Static analysis: systematic automated analysis of the program source without executing the program
- Structural analyses look for patterns in the code
- Control-flow analyses analyze all possible paths (global property)
- Data-flow analyses analyze possible (abstract) values of variables on all paths
 - Abstraction, transfer function, join
 - Fix point computation; termination
- Analyses unsound or incomplete or both

BONUS SLIDES: SYMBOLIC EXECUTION

Symbolic Execution

- Execute program with symbolic inputs.

```
y = read()  
y = 2 * y  
if (y == 12)  
    fail()  
print("OK")
```

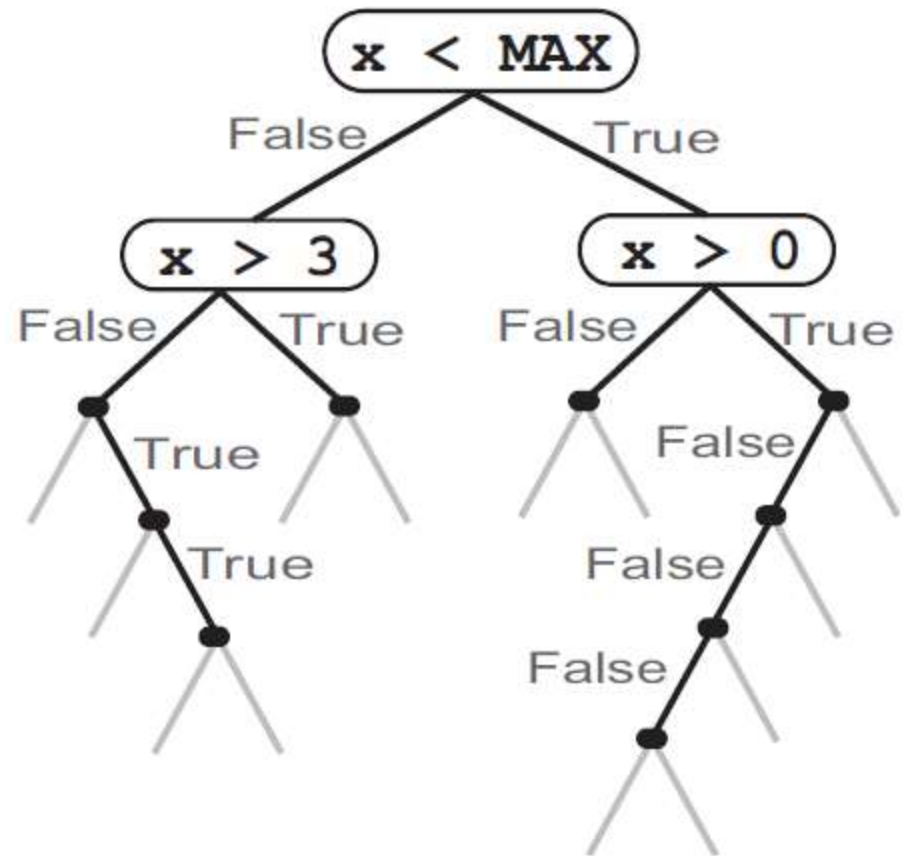
```
y =  $\alpha$   
y = 2 *  $\alpha$   
Successful path  
condition:  
    y = 2 *  $\alpha$ 
```

- Used for verification, test generation.

Symbolic Execution

- Exploring all paths

```
if (x < MAX) {  
    if (x > 0)  
        ...  
    else  
        ...  
} else {  
    if (x > 3)  
        ...  
}
```



Symbolic Execution: Limitations

- Path explosion
- Undecidable Path Constraints ($\alpha * \beta < 10$)
- Nontermination with unlimited loop bounds (while ($x < y$))

Practical scalability today: ~10,000 lines of code

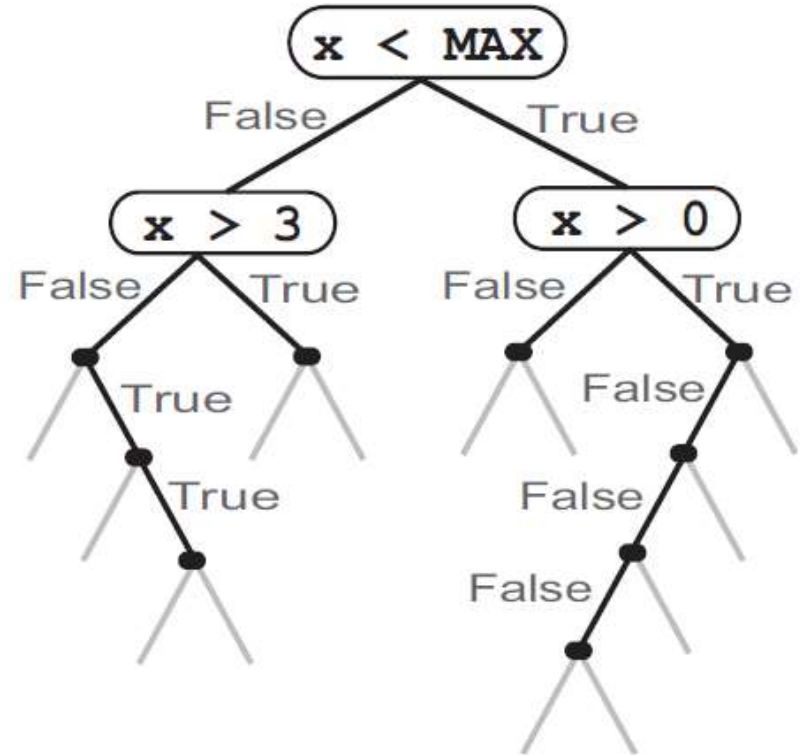
Dynamic Symbolic Execution

- Mixing Concrete and Symbolic Values
- Unsound -> Test Case Generation
- Given Unsolvability Constraint or Loop Bound: just guess one variable and continue

$$\begin{array}{l} \alpha * \beta < 10 \\ \alpha * 2 < 10 \end{array}$$

Automatic white-box test generation

- Dynamic Symbolic Execution to guide Fuzz Testing
- Microsoft SAGE
 - In production on Office, Windows
 - 200+ machines
 - 3 B+ constraints



The general procedure

- Start with random inputs.
- Execute the program.
 - Identify the paths/decisions/statements covered by the test case.
 - Collect **path constraints** corresponding to the execution.
- Flip one of the constraints, ask a **constraint solver** to give new inputs to force the execution down a different path.

- Execute with random input values (a = 0, b = 0).

– PC: a ≤ 0

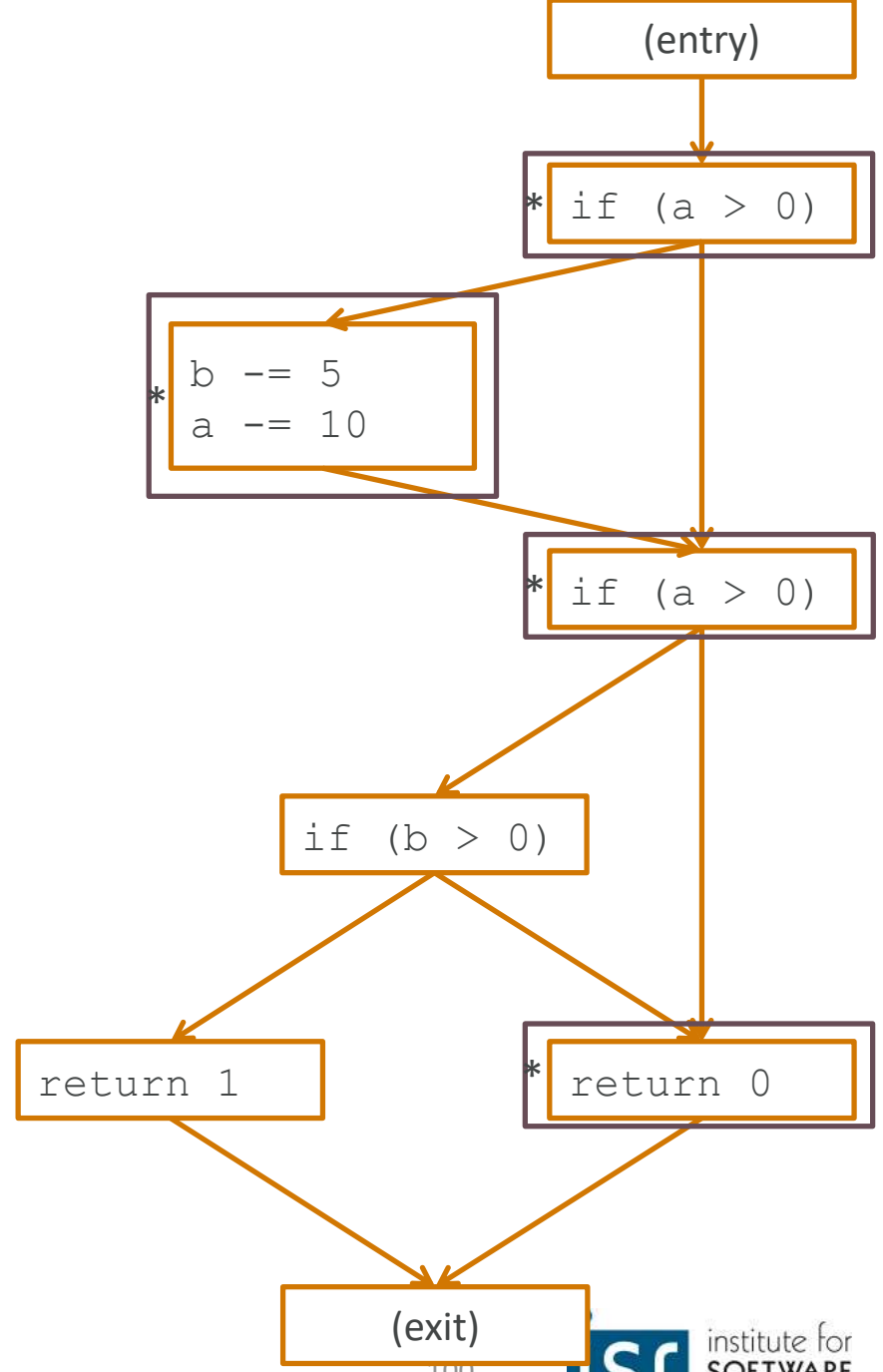
- Flip a ≤ 0, ask for a new input (a = 1, b = 0).

4. PC: a = 10; a - 10 ≤ 0

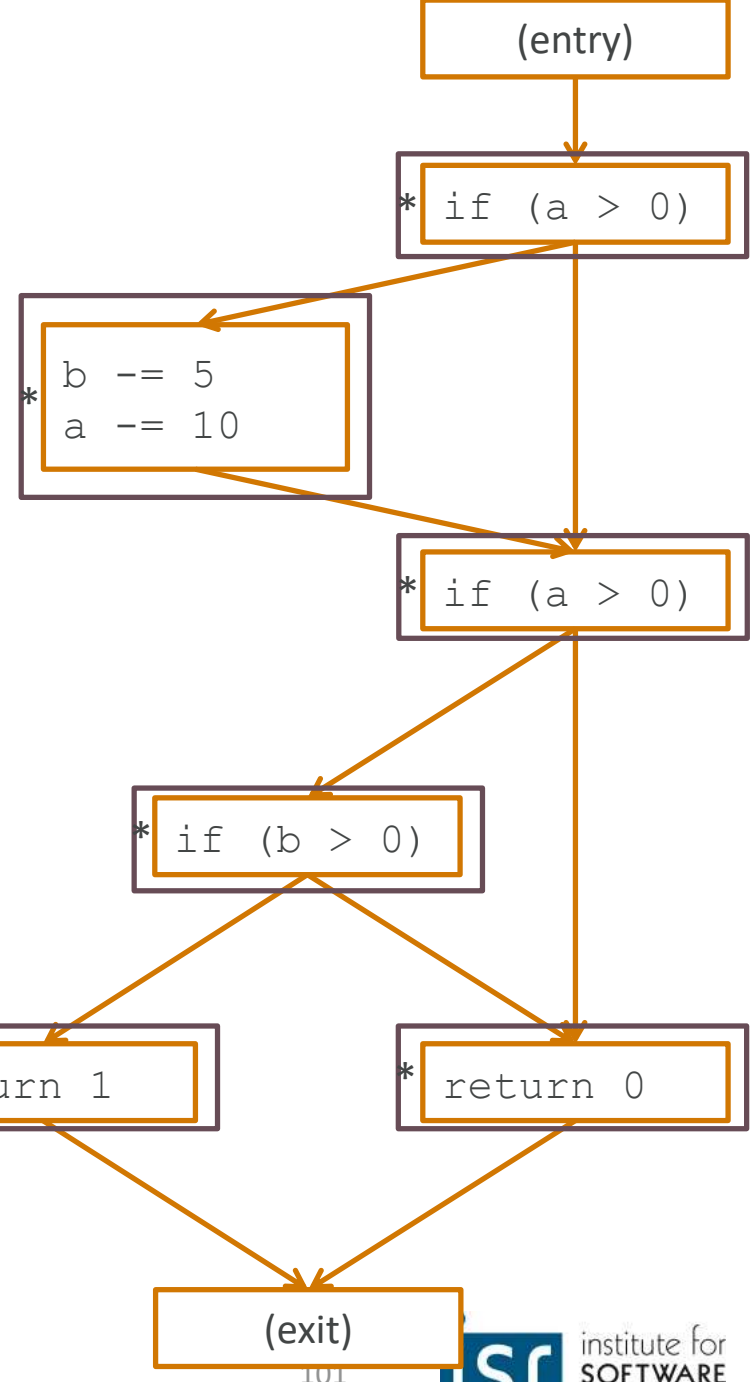
```

1. int foobar(a, b) {
2.   if (a > 0) {
3.     b -= 5;
4.     a -= 10;
5.   }
6.   if (a > 0) {
7.     if (b > 0)
8.       return 1;
9.   }
10.  return 0;
11. }

```



- Execute with random input values ($a = 0, b = 0$).
 - PC: $a \leq 0$
- Flip $a \leq 0$, ask for a new input ($a = 1, b = 0$).
 - PC: $a > 0; a - 10 \leq 0$
- Flip $a - 10 \leq 0$, ask for new input: ($a = 11, b = 0$).
 - PC: $a > 0; a - 10 > 10; b - 5 < 0$
- Flip $b - 5 < 0$, ask for a new input ($a = 11, b = 6$).
- Test cases: (0,0), (1,0), (11,0), (11,6)



Making things better: termination

- Secret weapon: define your abstraction such that it is finite.
- If you come to a statement and you've already explored a given state for that statement, stop.
 - The analysis depends on the code and the current state; continuing the analysis from this program point and state would yield the same results.
- If the number of possible states isn't finite, you're stuck.
 - Your analysis may not terminate.
- Common solution: cap the number of paths/loop iterations to 0, 1, or 2.

Check out...

- PEX: Automated White Box Testing for .NET
 - Technique out of Microsoft Research
 - Extension to Visual Studio
- Pex4Fun: educational programming web game based on PEX.

