Foundation of Software Engineering

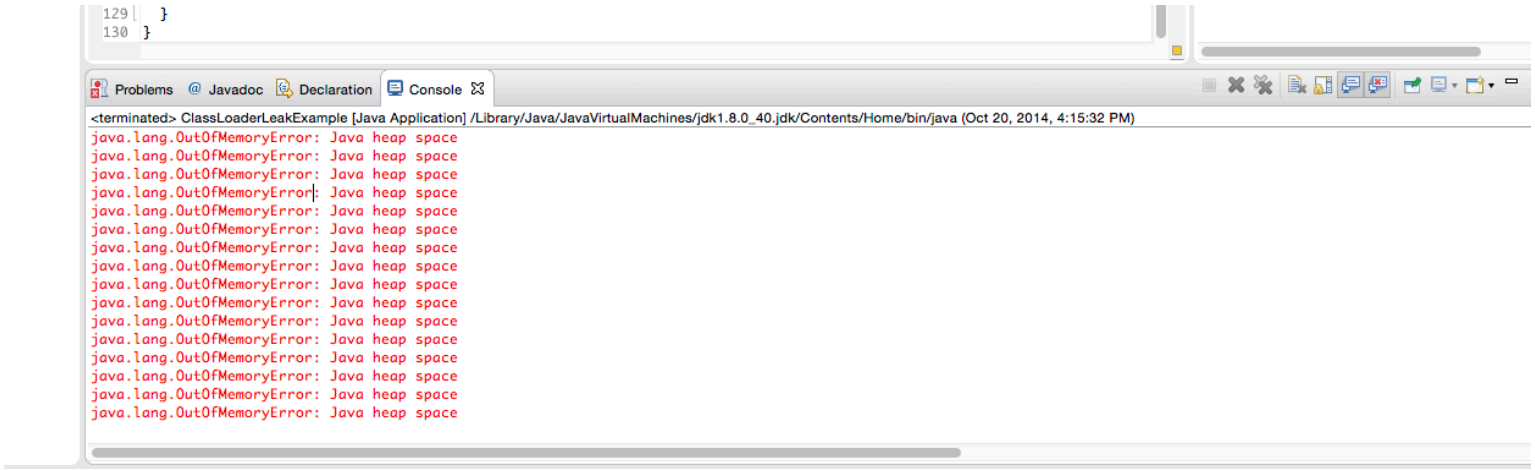
Lecture 14a – Dynamic Analysis

Claire Le Goues

institute for SOFTWARE RESEARCH

Learning goals

- Define dynamic analysis, including the high-level components of such analyses, and understand how abstraction applies.
  - Identify defect classes and circumstances for which dynamic analyses might apply.
  - Give examples of common dynamic analyses.
- Explain two high-level ways, with tools or examples, for collecting information for dynamic analysis.
- Enumerate the challenges posed by and limitations of dynamic analysis, such that you can choose between them or determine if it might be suitable in a given situation.
- (If we have time) Describe and give tradeoffs for mutation testing and capture/recapture as adequacy criteria.



WHAT'S A MEMORY LEAK?

Definition: Memory leak

- Memory is allocated, but not released properly.
- In C: malloc()'d memory that is not eventually free()'d:
- In OO/Java: objects created/rooted in memory that cannot be accessed but will not be freed.
  - *Is this actually possible?*
  - Memory usually automagically managed by the garbage collector, but…

How can we tackle this problem?

- Testing:
- Static analysis:
- Inspection:

Wouldn't it be nice if we could learn about the program's memory usage as it was running?

Dynamic analysis: learn about a program's properties by executing it

- How can we learn about properties that are more interesting than "did this test pass" (e.g., memory use)?
- Short answer: examine program state throughout/after execution by gathering additional information.

Common dynamic analyses

- Coverage
- Performance
- Memory usage
- Security properties
- Concurrency errors
- Invariant detection

Collecting execution info

- Instrument at compile time
  - e.g., Aspects, logging
- Run on a specialized VM
  - e.g., valgrind
- Instrument or monitor at runtime
  - also requires a special VM
  - e.g., hooking into the JVM using debugging symbols to profile/monitor (VisualVM)

Collecting execution info

- In
  - e.g., A
- Run on a specialized VM
  - e.g., valgrind
- Instrum
  - als
  - e.g., hooki ugging symbols to prof nonit (VisualVM)

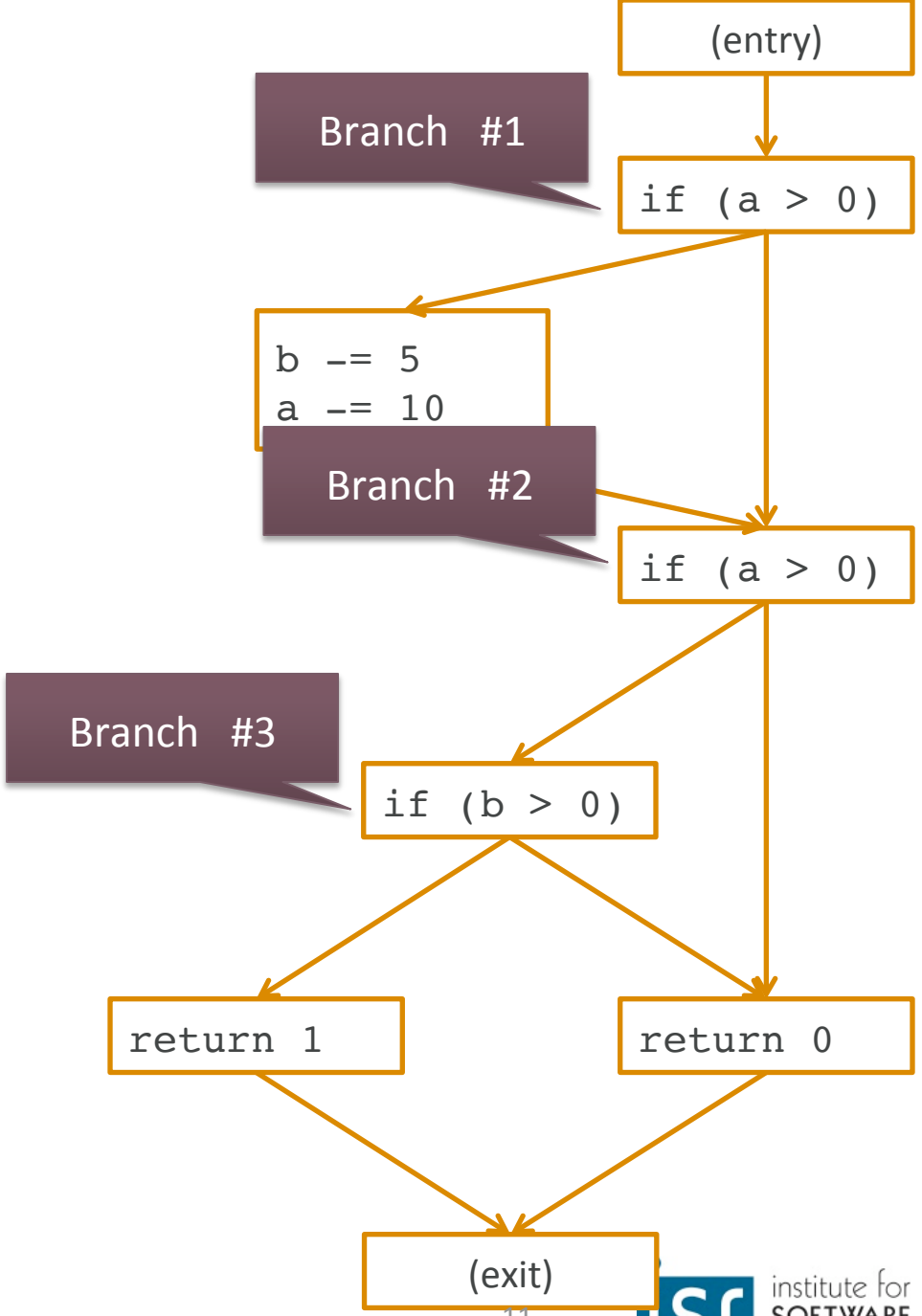
Note: some of these methods require a *static* pre processing step!

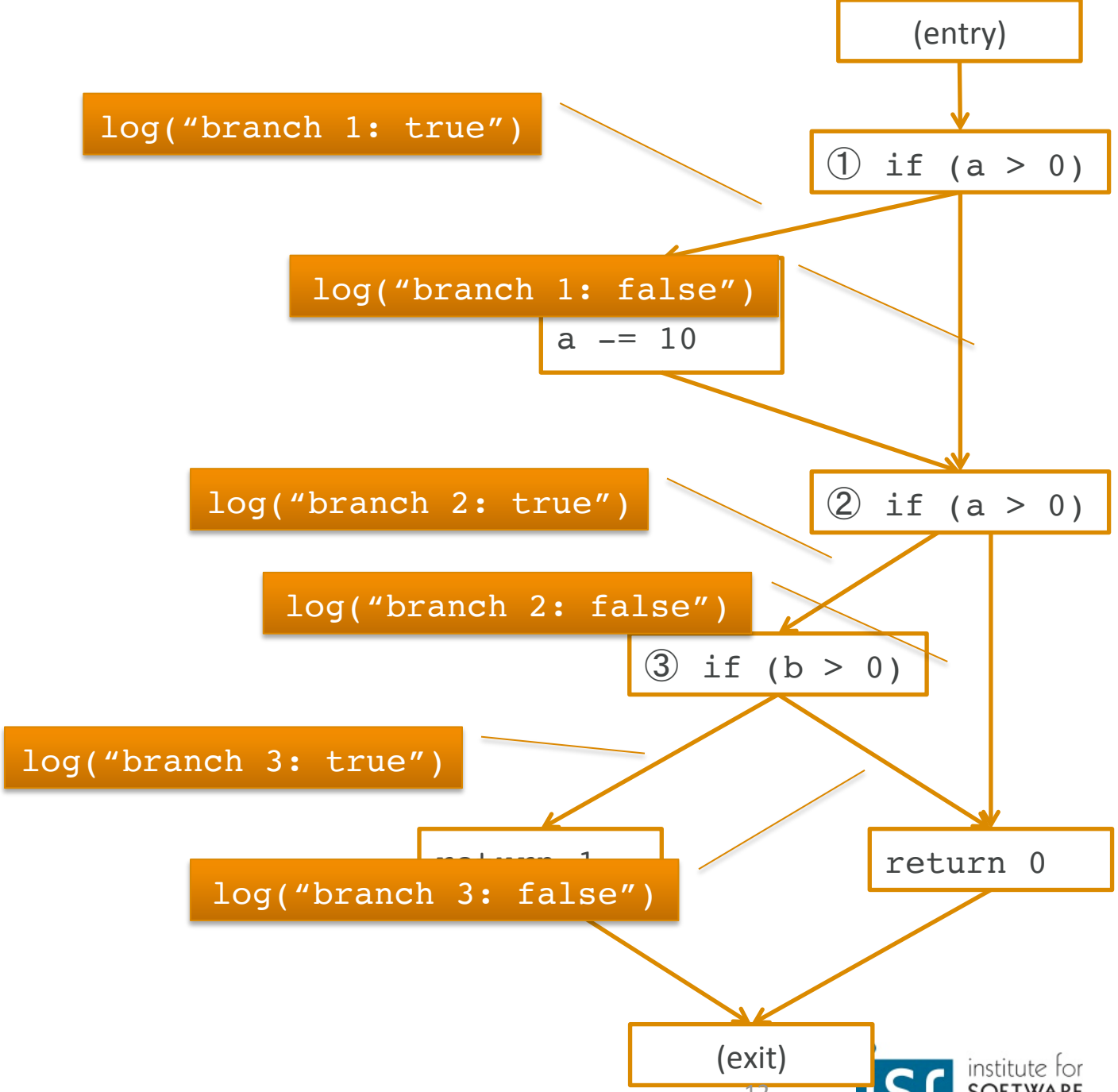
Avoid mixing up static things done to collect info and the dynamic analyses that use the info.

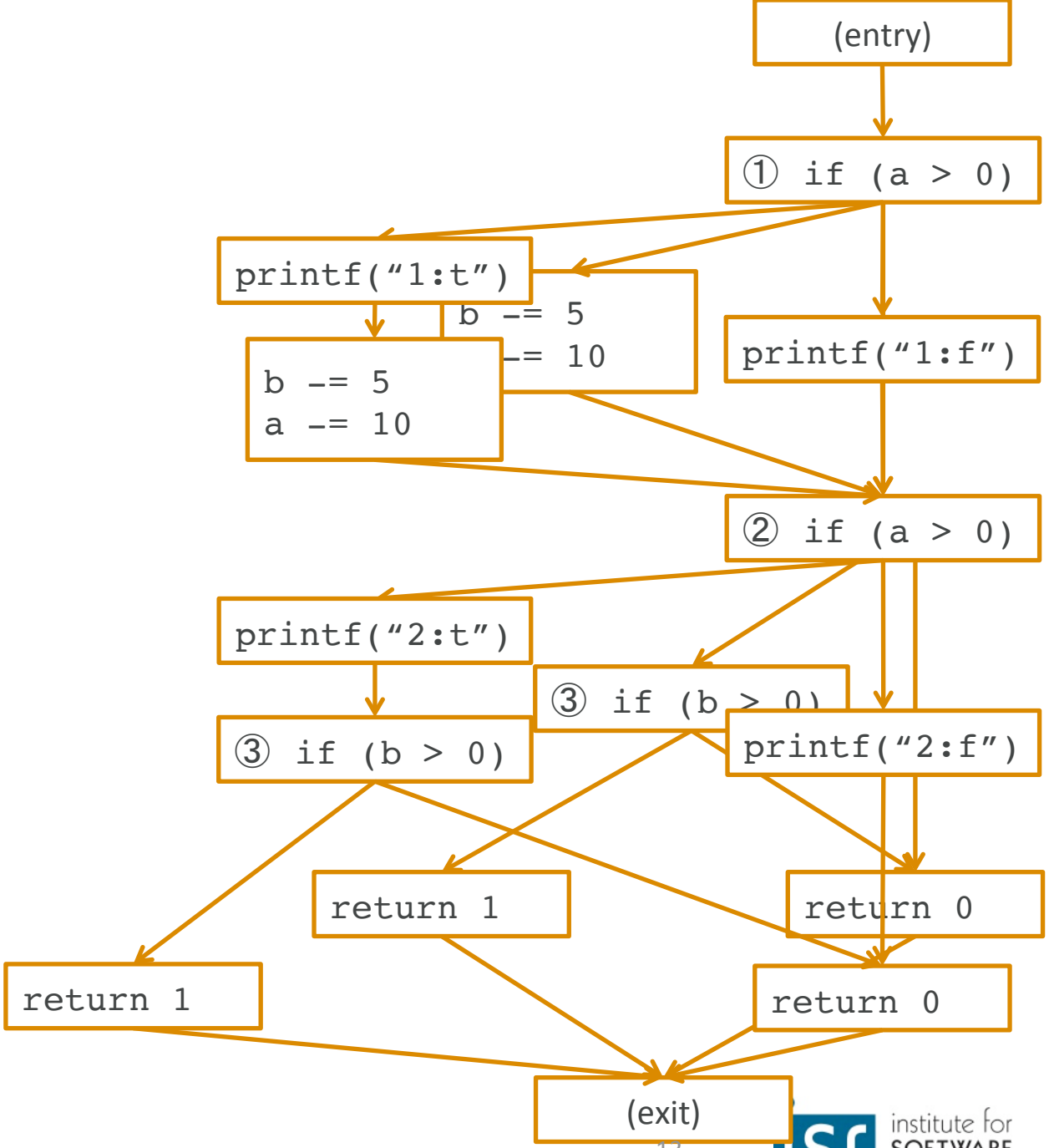
Instrumentation: a simple example

- How might tools that compute test suite coverage work?
- One option: *instrument* the code to track a certain type of data as the program executes.
  - **Instrument:** add of special code to track a certain type of information as a program executes.
  - Rephrase: insert logging statements (e.g., at compile time).
- What do we want to log/track for branch coverage computation?

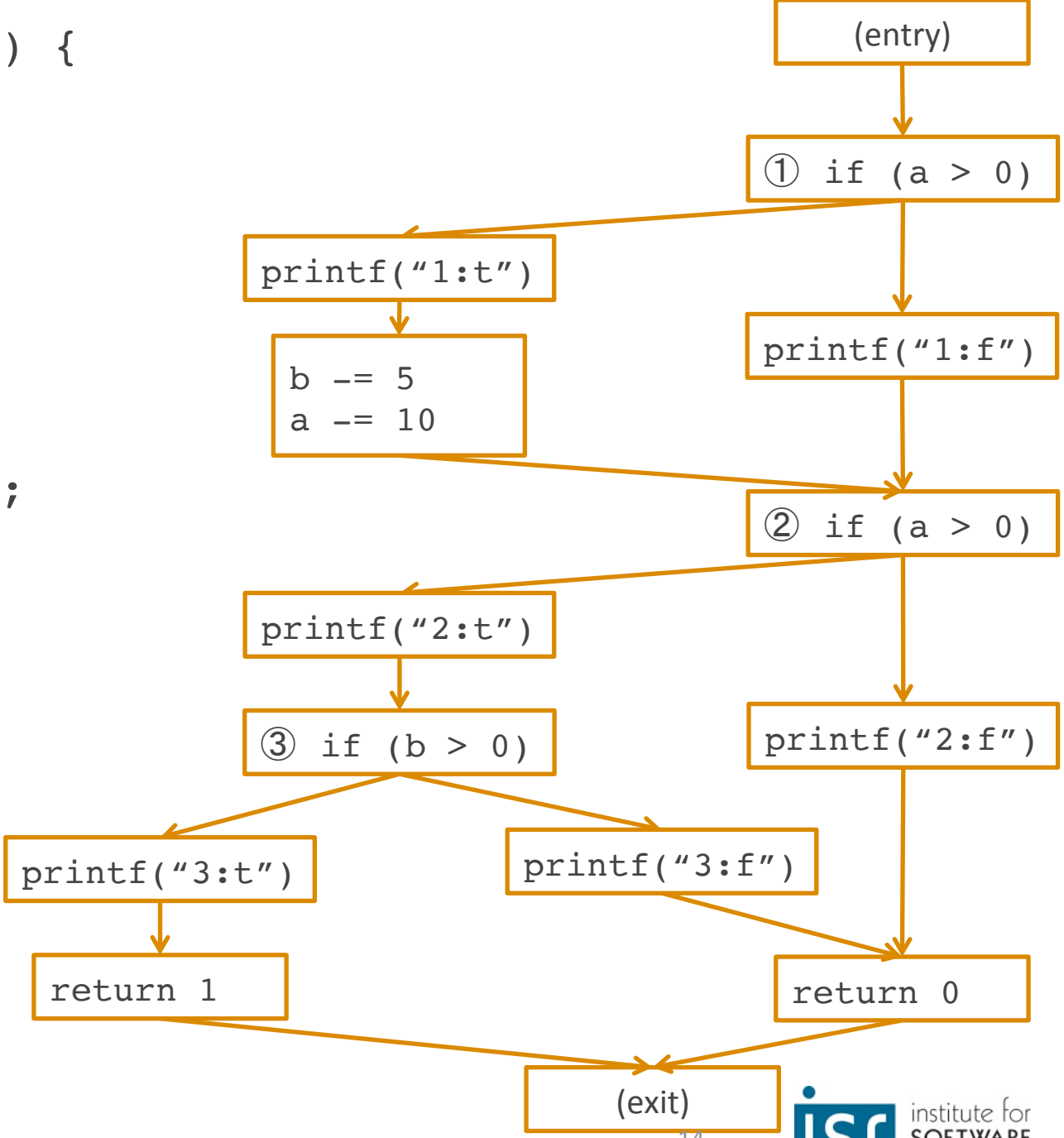
```
1.  int foobar(a,b) {
2.    if (a > 0) {
3.      b -= 5;
4.      a -= 10;
5.    }
6.    if(a > 0) {
7.      if (b > 0)
8.        return 1;
9.    }
10.   return 0;
11. }
```



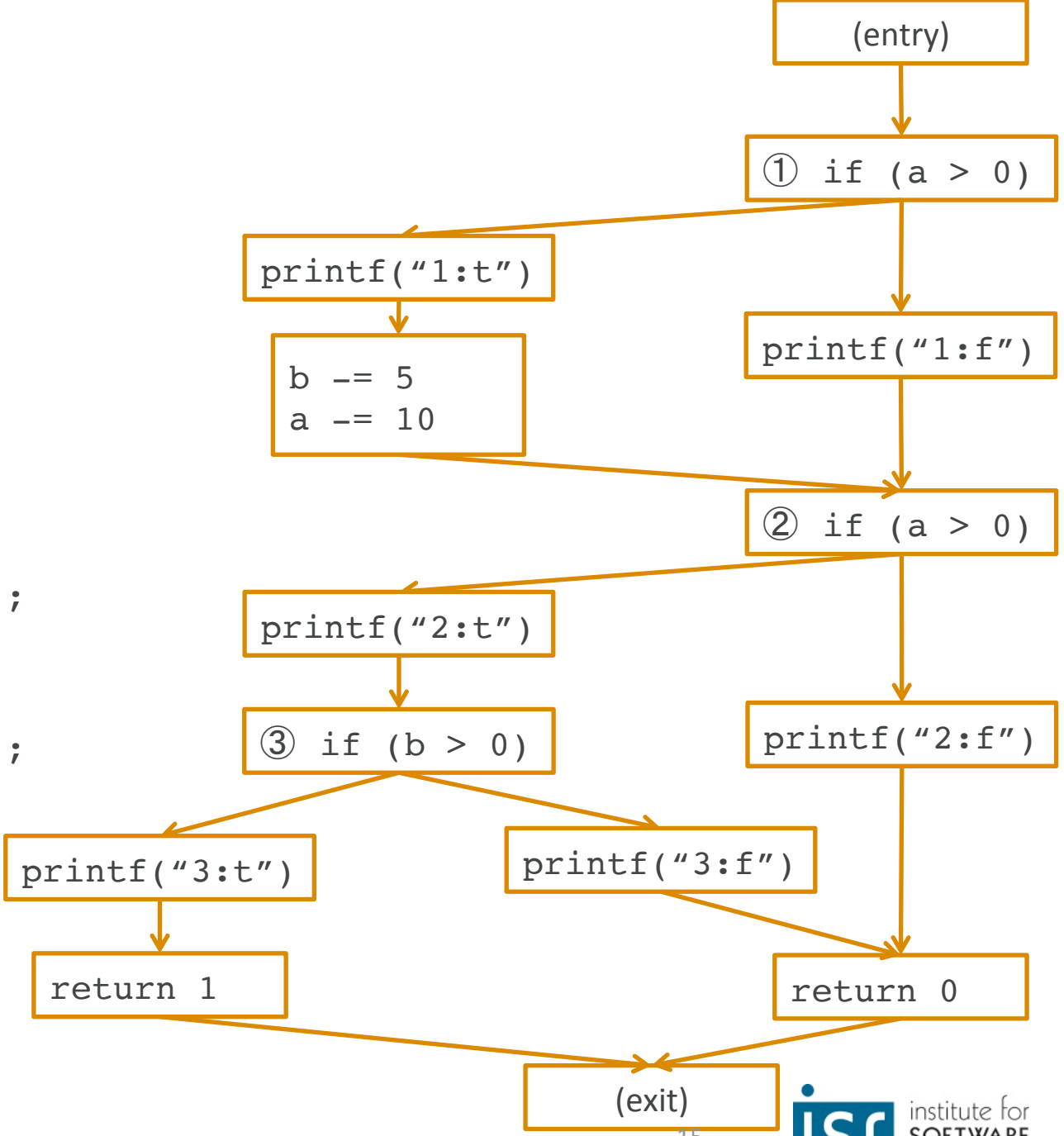




```
1.  int foobar(a,b) {
2.    if (a > 0) {
3.      b -= 5;
4.      a -= 10;
5.    }
6.    if(a > 0) {
7.      if (b > 0)
8.        return 1;
9.    }
10.   return 0;
11. }
```



```
1.int foobar(a,b) {
2.  if (a > 0) {
3.    printf("1:t");
4.    b -= 5;
5.    a -= 10;
6.  } else {
7.    printf("1:f");
8.  }
9.  if(a > 0) {
10.    printf("2:t");
11.    if (b > 0) {
12.      printf("3:t");
13.      return 1;
14.    } else {
15.      printf("3:f");
16.    }
17.  } else {
18.    printf("2:f");
19.  }
20.  return 0;
21.}
```



```
1.int foobar(a,b) {
2.  if (a > 0) {
3.    printf("1:t ");
4.    b -= 5;
5.    a -= 10;
6.  } else {
7.    printf("1:f ");
8.  }
9.  if(a > 0) {
10.    printf("2:t ");
11.    if (b > 0) {
12.      printf("3:t ");
13.      return 1;
14.    } else {
15.      printf("3:f ");
16.    }
17.  } else {
18.    printf("2:f ");
19.  }
20.  return 0;
21.}
```

- Test cases: (0,0), (1,0), (11,0), (11,6)
  - foobar(0,0): "1:f 2:f "
  - foobar(1,0): "1:t 2:f "
  - foobar(11,0): "1:t 2:t 3:f "
  - foobar(11,6): "1:t 2:t 3:t "

Assuming we saved how many branches were in this method when we instrumented it, we could now process these logs to compute branch coverage.

Abstraction

- Why was abstraction relevant to static analysis, again?
- *Dynamic analysis also requires abstraction.*
- You're still focusing on a particular program property or type of information.
  - Abstracting parts of a trace or execution rather than the entire state space.
- How does abstraction apply in the coverage example?

Parts of a dynamic analysis

- Property of interest.
- Information related to property of interest.
- Mechanism for collecting that information from a program execution.
- Test input data.
- Mechanism for learning about the property of interest from the information you collected.

What are you trying to learn about? Why?

How are you learning about that property?

Instrumentation, etc.

What are you running the program on to collect the information?

For example: how do you get from the logs to branch coverage?

Coverage example, redux:

1. Property of interest.
2. Information related to property of interest.
3. Mechanism for collecting that information from a program execution.
4. Test input data.
5. Mechanism for learning about the property of interest from the information you collected.

1. *Branch coverage of the test suite!*
2. *Which branch was executed when!*
3. *Logging statements!*
4. *The test cases we generated for that example last Thursday!*
5. *Postprocessing step to go from logs to coverage info!*

Tool support can help with this.

- No tool support → “printf debugging”
- Other options:
  - Aspects
  - Binary/source level rewriting
  - Logging frameworks

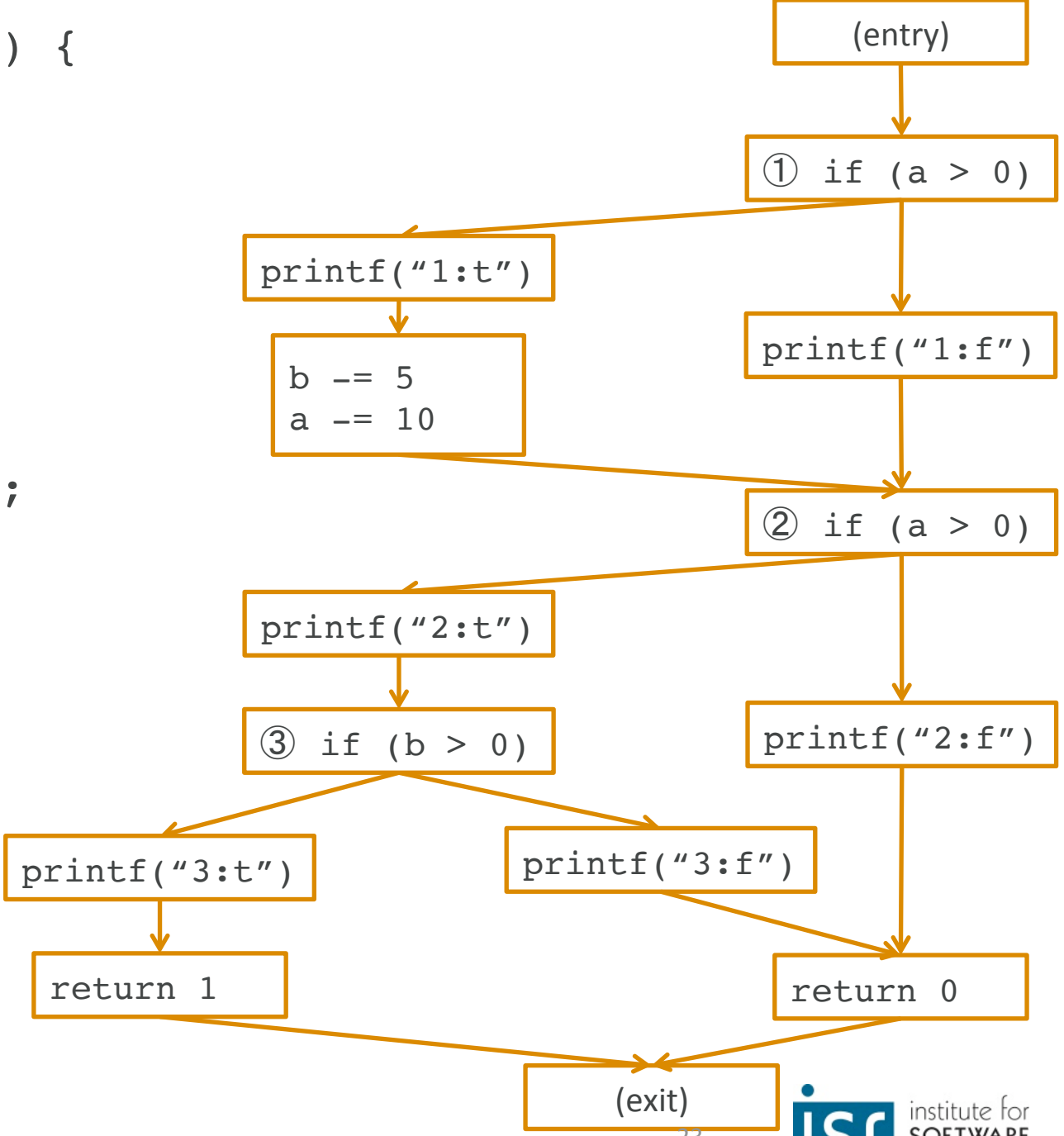
More on info collection

- Instrumentation: many complex profilers work this way, in the spirit of our example.
- Aspects can also used for compile-time instrumentation.

Aspect-oriented programming

- Cross-cutting concern: a problem that "cuts across" multiple abstractions in a program.
  - Classic example: logging.
- Aspect: additional code that you would like to be executed at particular points in your programming to address a cross-cutting concern.
- Compilers can support this, allowing you to define the thing you'd like executed and the conditions under which you'd like it to be executed, and automatically inserting the code.

```
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. }
```



Example instrumentation uses

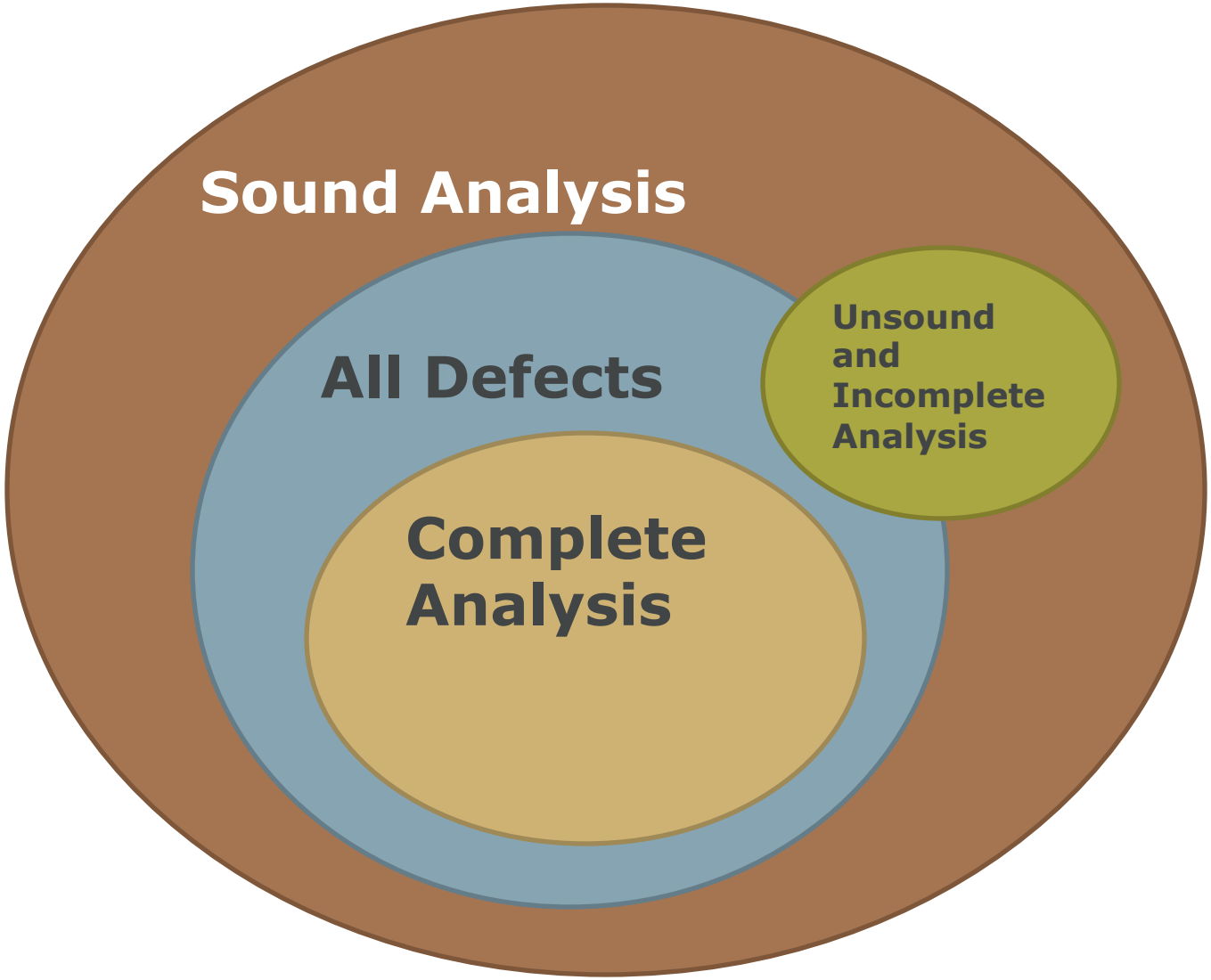
- Coverage computation
- Performance profiling and analysis: where does your program spend all of it's time, energy, etc?
  - Classic example: gprof
- Concurrency error debugging? Hmm….

More on info collection

- Instrumentation: many complex profilers work this way, in the spirit of our example.
- Aspects can also used for compile-time instrumentation.
- Virtual machines/emulators: especially convenient for languages that already use VMs, like Java.
  - (but not limited to, see valgrind or gdb)
- VMs can intercept all executed instructions, allowing:
  1. Selectively rewrite running code, or runtime instrumentation. (e.g., software breakpoints in the gdb debugger)
  2. profile or otherwise do behavioral sampling.

(Alternative section title(s): What could possibly go wrong?, or, Things to think about when the used-dynamic analysis tool salesperson shows up at your door)

LIMITATIONS AND CHALLENGES



	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

Very input dependent

- Good if you have lots of tests!
  - (system tests are often best)
- Are those tests indicative of normal use
  - And is that what you want?
- Can also use logs from live software runs that include actual user interactions (sometimes, see next slides).
- Or: specific inputs that replicate specific defect scenarios (like memory leaks).

Heisenbuggy behavior

- Instrumentation and monitoring can change the behavior of a program.
  - e.g., slowdown, memory overhead.
- **Important question 1:** can/should you deploy it live?
  - Or possibly just deploy for debugging something specific?
- **Important question 2:** *Will the monitoring meaningfully change the program behavior with respect to the property you care about?*

Too much data

- Logging events in large and/or long-running programs (even for just one property!) can result in HUGE amounts of data.
- How do you process it?
  - Common strategy: sampling

Benefits over static analysis

- Precise data (for a specific run)
  - No false positives or false negatives on a given run.
- Always possible to determine where the error occurred.
  - No confusion about which path was taken.
- Can (often) use on live code
  - Very common for security and data gathering

Static Analysis

- Analyzing the code, without executing it
- Systematically following an abstraction
- Find bugs / enforce stronger specifications / proof correctness
- Often correctness / security related

Dynamic Analysis

- Analyzing specific executions
- Instrumenting program or special interpreter
- Abstracting parts of the trace
- Find bugs / enforce stronger specifications / debugging / profiling
- Often memory / performance / concurrency / security related

Lifecycle

- During QA
  - Instrument code for tests
  - Let it run on all regression tests
  - Store output as part of the regression
- During Production
  - Only works for web apps
  - Instrument a few of the servers
    - Use them to gather data
    - Statistical analysis, similar to seeding defects in code reviews
  - Instrument all of the servers
    - Use them to protect data

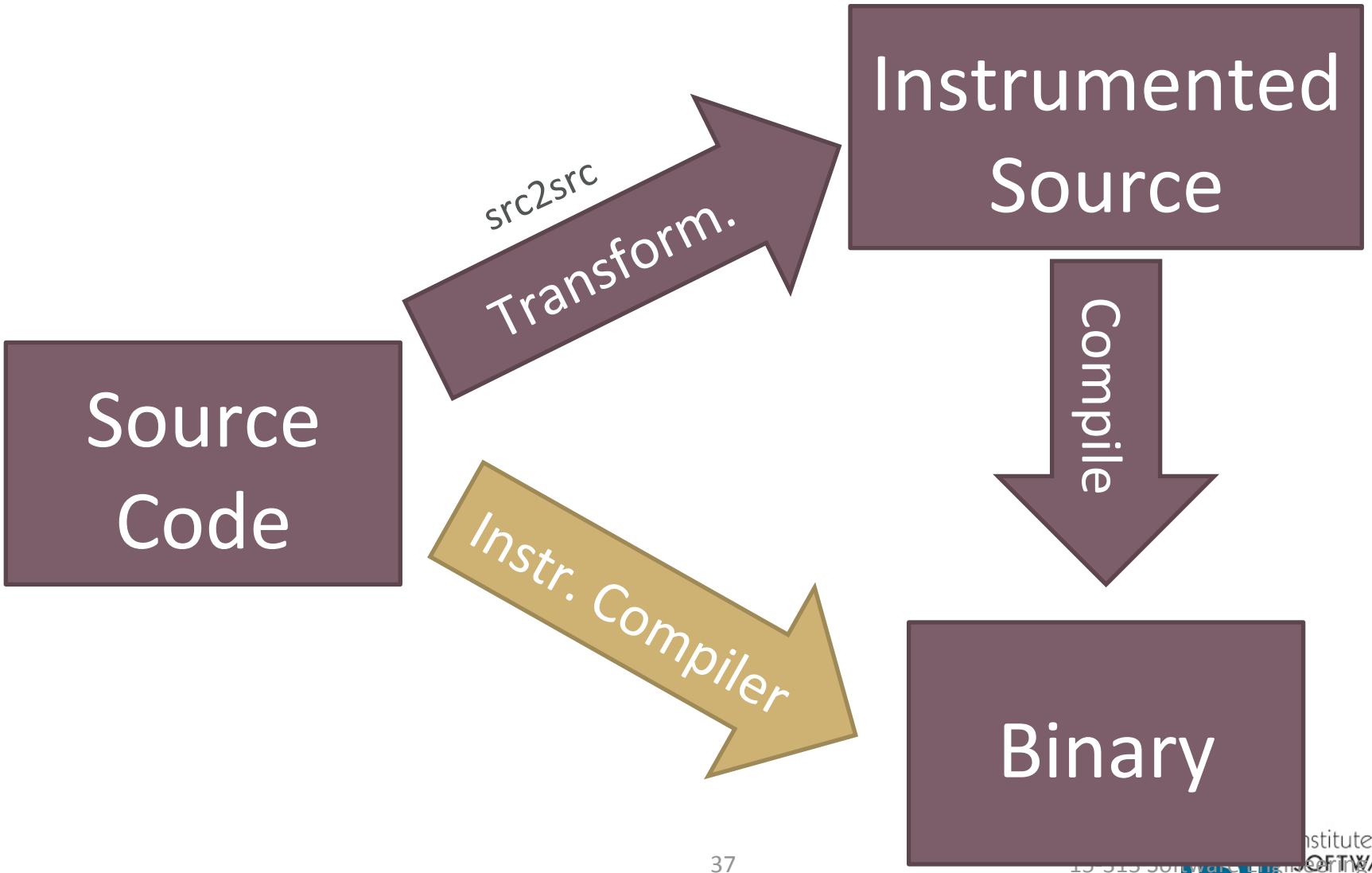
Foundations of Software Engineering

Lecture 14b: Code Instrumentation

Christian Kästner

Learning Goals

Code Transformation



Examples

- Check every parameter of every method is non-null
- Write the duration of the method execution of every method into a file
- Report warning on Integer overflow
- Use a connection pool instead of creating every database connection from scratch

How to Transform Source Code?

Text manipulation

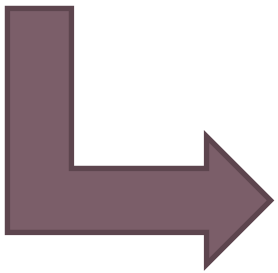
- Regular expressions

- s/(\w+\(.*\);)/int t=time();\
  $1 print(time()-t);/g

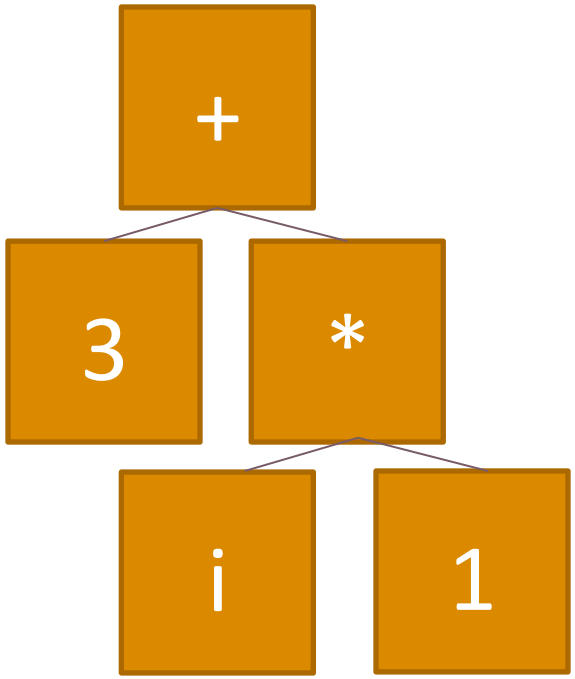
- Benefits?
- Drawbacks?

Parsing + Pretty Printing

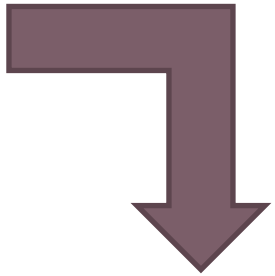
"3+(i*1)"



parsing



pretty printing

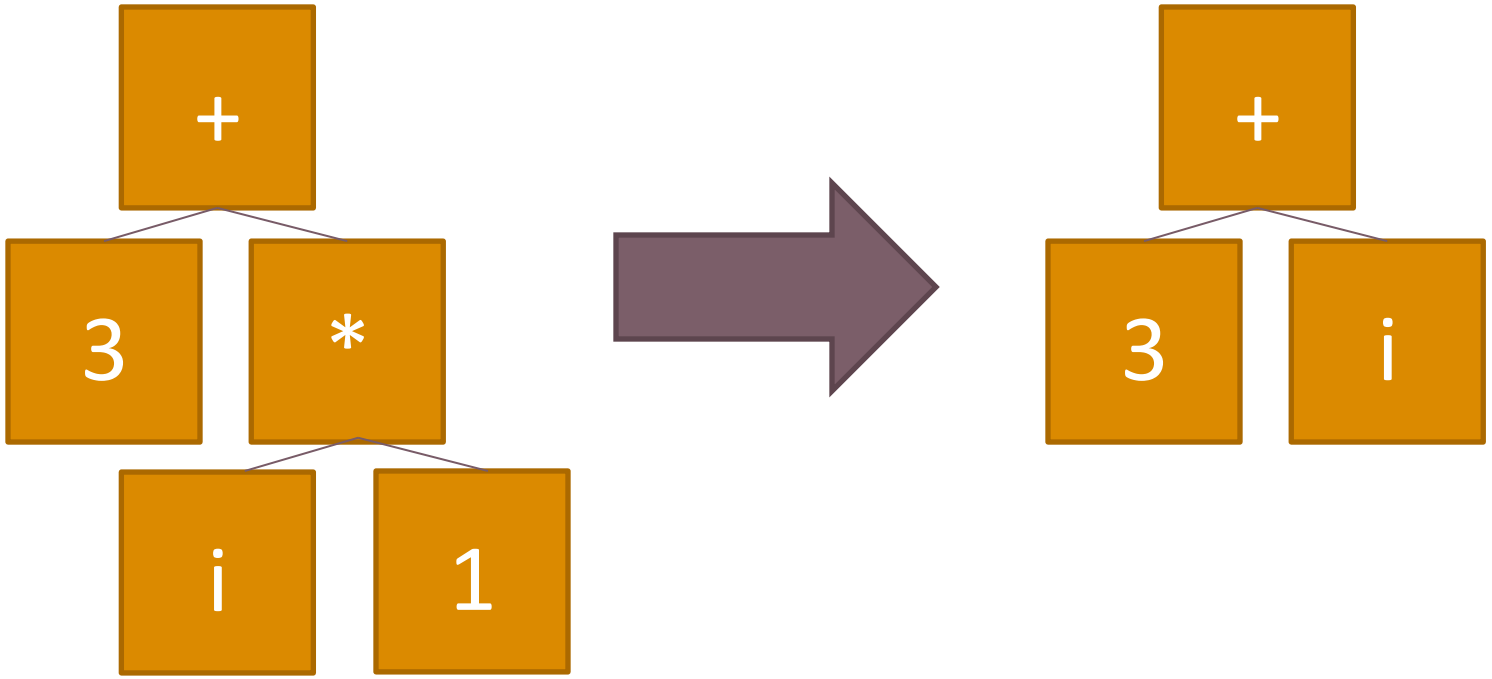


"3+i*1"

Parsing technology

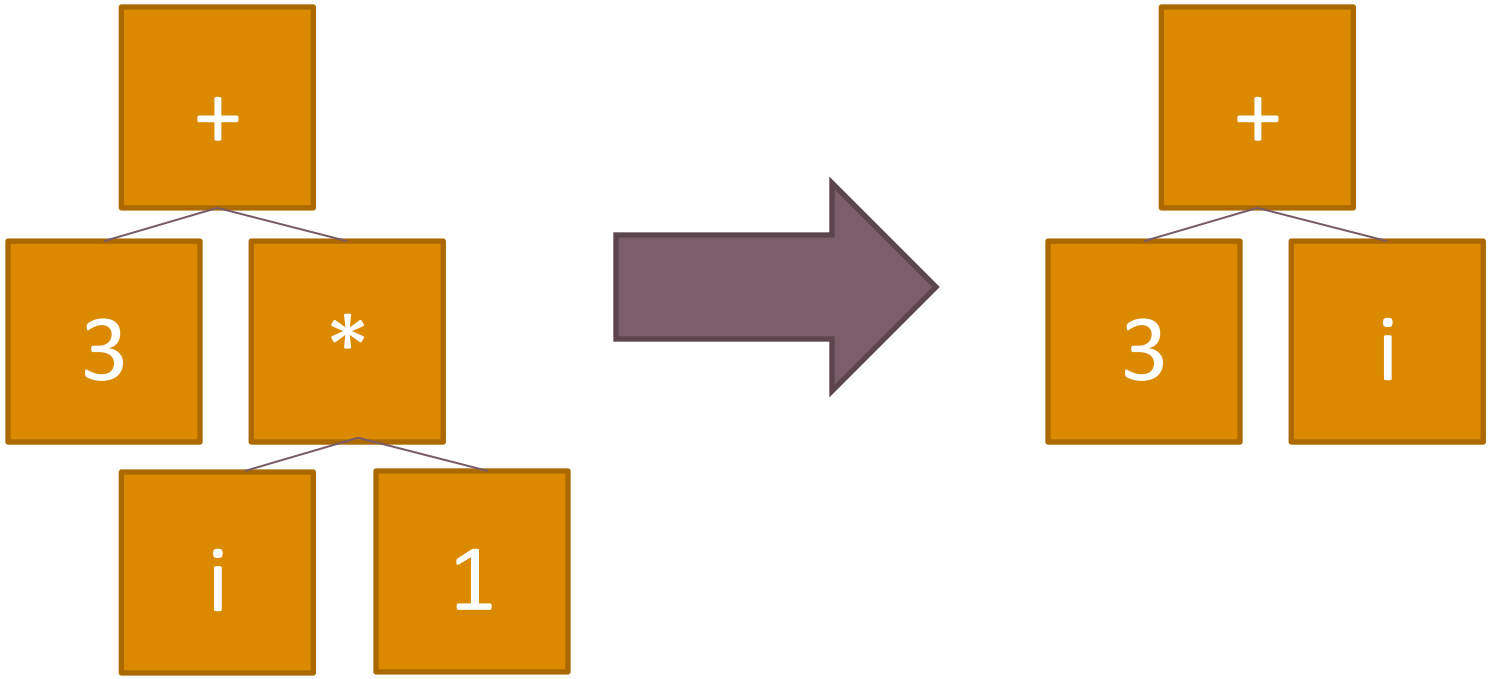
- Standard technology
  - Handwritten parsers
  - Parser generators LR, LL, GLR, …
  - Parser combinators
  - …
- Pretty printer often written separately

AST Rewriting



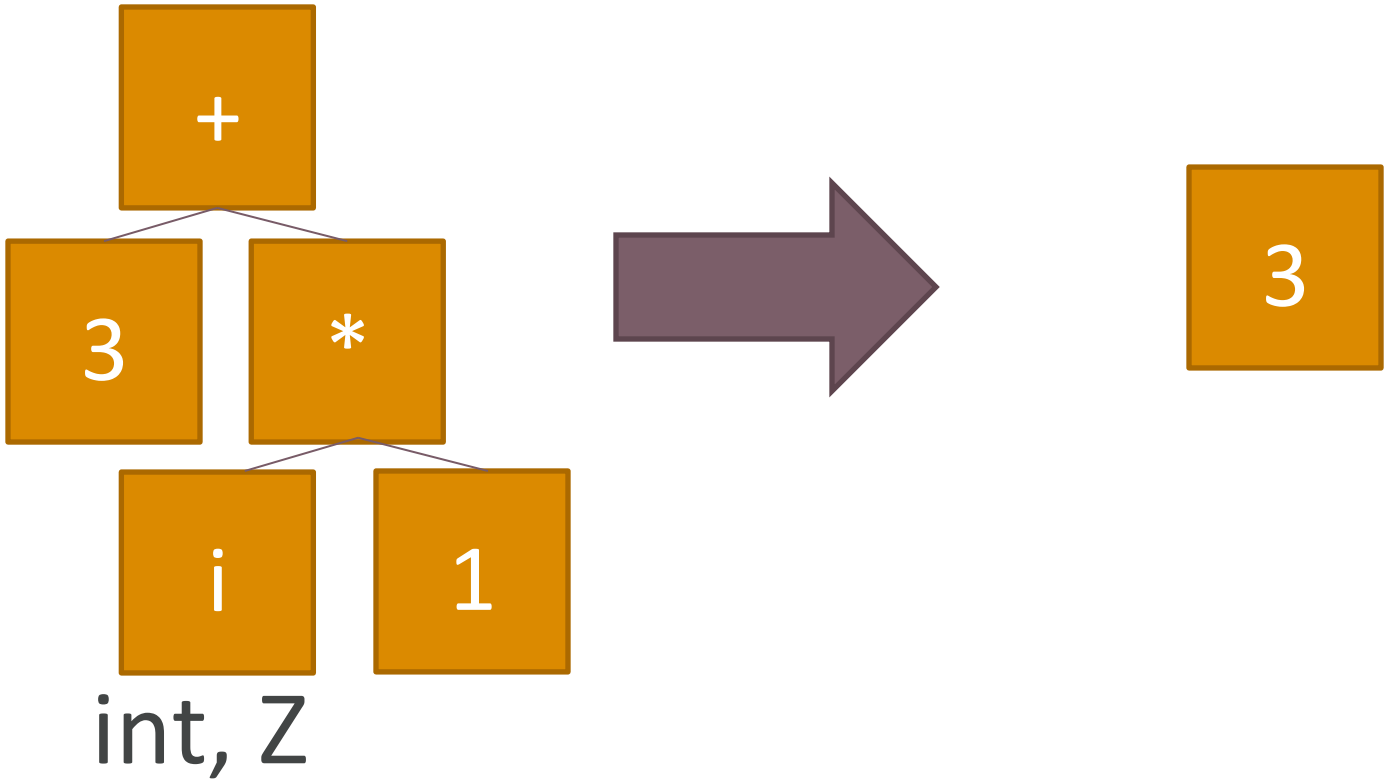
- Benefits/Drawbacks?
- Commercial rewrite systems exist
- Visitors, pattern matcher, …

AST Rewriting

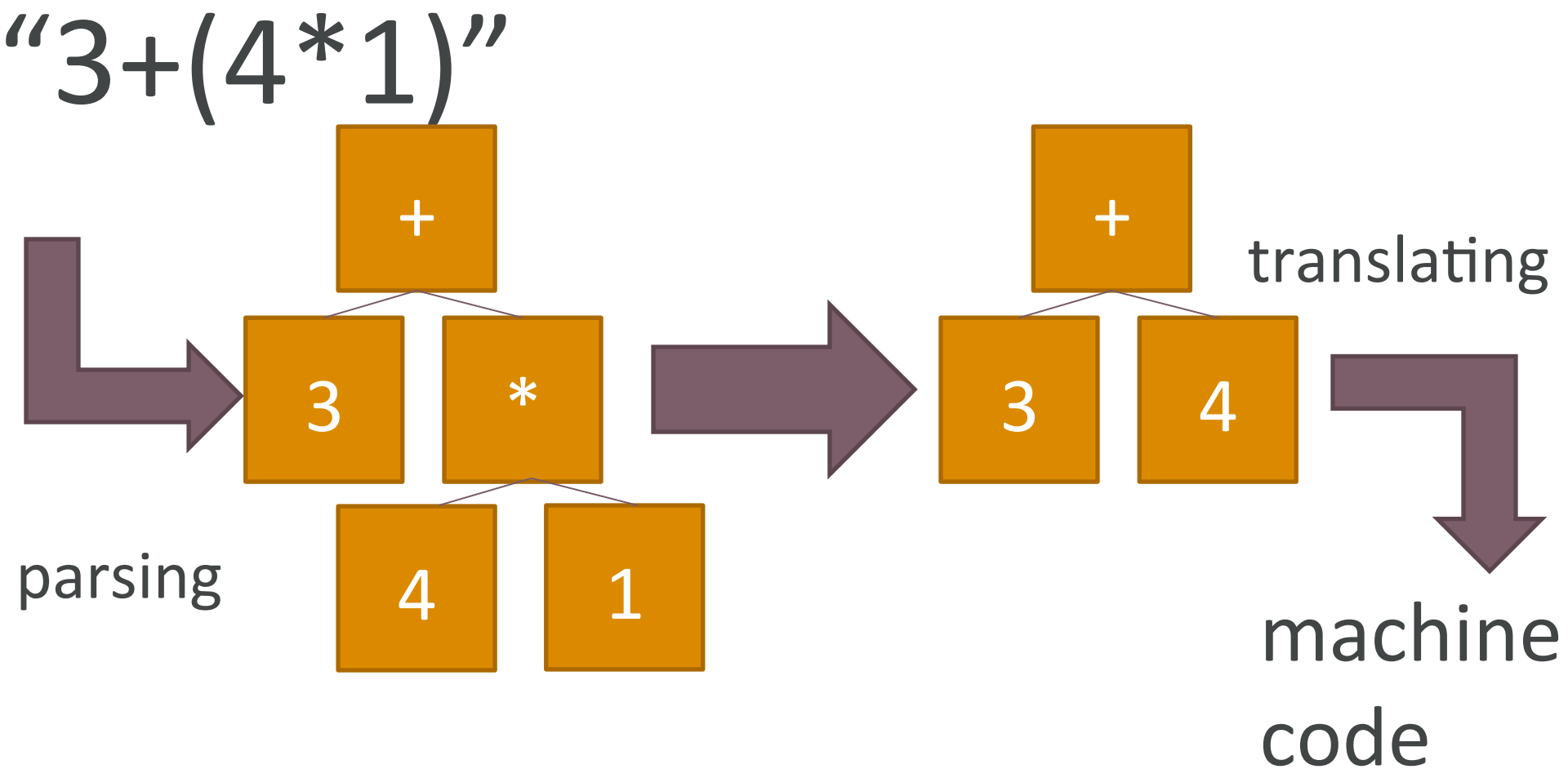


- Often useful to have type/context information

Static Analysis + Rewriting



Rewriting as a Compiler Pass



Rewriting tools

- Rewrite patterns over trees, typically with parser/pretty printer systems
  - Stratego/XT
  - DSM
  - …
- Within language rewriting
  - Aspect-oriented programming

AspectJ

```
Object around() :
        execution(public * com.company..*.* (..)) {
    long start = System.currentTimeMillis();
    try {
        return proceed();
    } finally {
        long end = System.currentTimeMillis();
        recordTime(start, end,
            thisJoinPointStaticPart.getSignature());
    }
}
```

Byte Code Rewriting

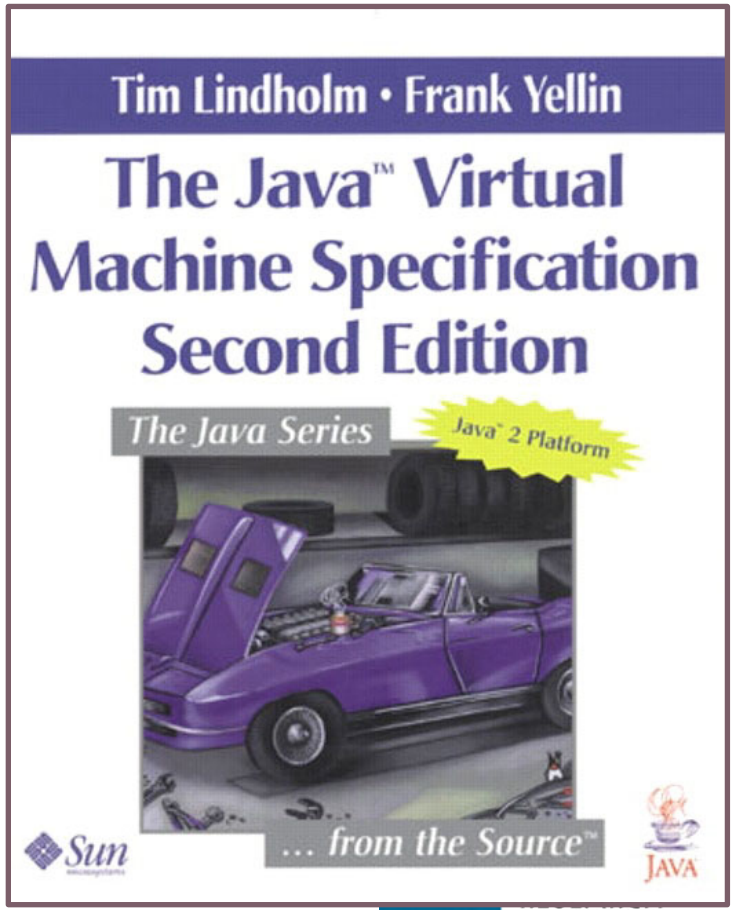
- Java AST vs Byte Code
- Byte Code is JVM input (binary equivalent)
  - Stack machine
  - Load/push/pop values from variables to stack
  - Stack operations, e.g. addition
  - Call methods, …

Byte Code example
(of a method with a single int parameter)

- ALOAD 0
- ILOAD 1
- ICONST 1
- IADD
- INVOKEVIRTUAL “my/Demo” “foo”
  “(I)Ljava/lang/Integer;”
- ARETURN

JVM Specification

- https://docs.oracle.com/javase/specs/
- See byte code of Java classes with *javap* or ASM Eclipse plugin
- Several analysis/rewrite frameworks as ASM or BECL (internally also used by AspectJ, …)



Examples

- Check every parameter of every method is non-null
- Write the duration of the method execution of every method into a file
- Report warning on Integer overflow
- Use a connection pool instead of creating every database connection from scratch