EDAN65 Guest Lecture

CodeProber, Testing, ExtendJ

Anton Risberg Alaküla
22/9 2025, MA:6

Outline & Goal

Today I will present:

- My research area, focusing on “CodeProber”
- How to test compiler semantics

Goal:

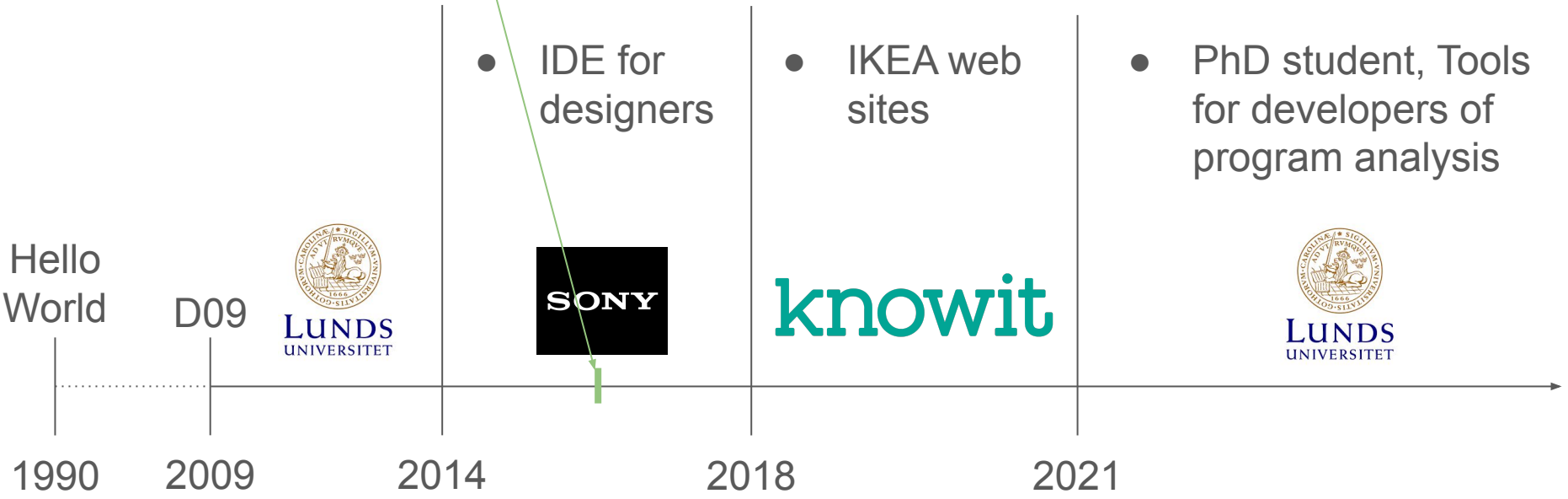
- Help you get started on Lab 4 (*perhaps the most challenging lab in the course)*
- Show that the tools & techniques you learn scales to “real” languages

Who am I?

Antons Timeline

Promotion!
A good thing, right?

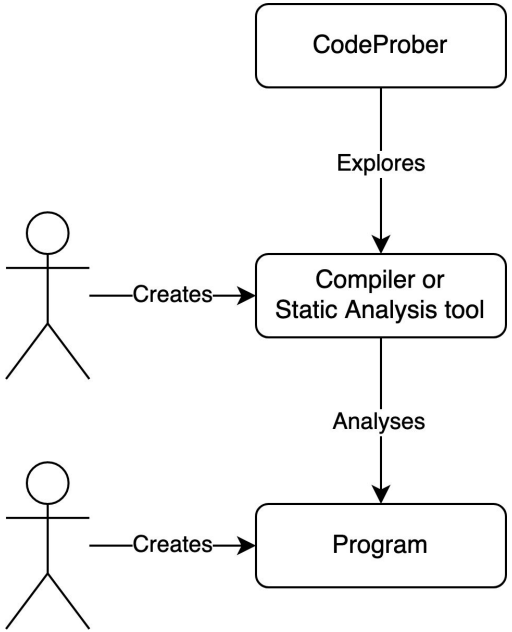
Not entirely: I now had to do more code review.
“Would a better compiler or program analyzer reduce the need for code review?”

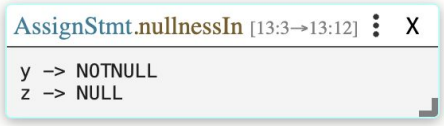


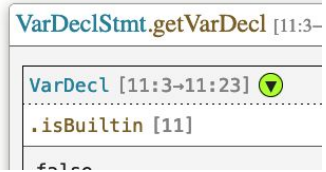
- IDE for designers
- IKEA web sites
- PhD student, Tools for developers of program analysis

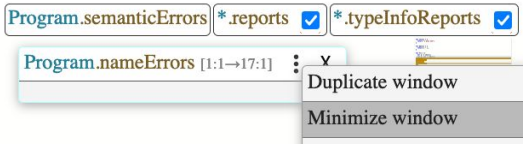
CodeProber

- Live exploration tool for compilers and program analyzers
  - Doesn’t actually compute anything, merely provides a UI for exploring your own computations
- Used during labs in
  - Compilers (EDAN65)
  - Program Analysis (EDAP15)
- Developed since 2022



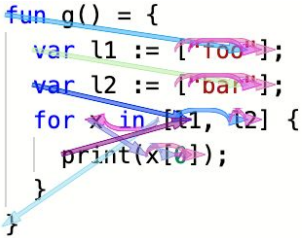

AssignStmt.nullnessIn [13:3→13:12]
y -> NOTNULL
z -> NULL


VarDeclStmt.getVarDecl [11:3–
VarDecl [11:3→11:23]
.isBuiltin [11]

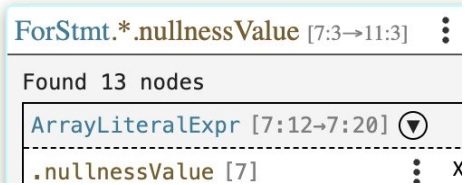

Program.semanticErrors
*.reports
*.typeInfoReports
Program.nameErrors [1:1→17:1]
Duplicate window
Minimize window

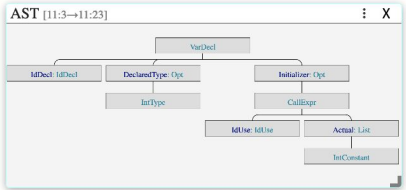

IndexExpr.getBase [5:11→5:14]
Access [5:11→5:11]
Access.? [5:11→5:11]
Filter
getIdUse()


x ↦ NULL Program.nullReport
View Problem (⌥F8)
No quick fix
:(x[0]);

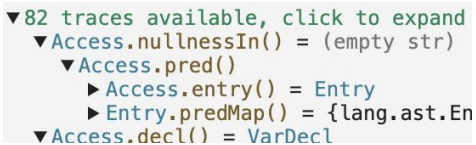

fun g() = {
var l1 := ["foo"];
var l2 := ["bar"];
for x in [l1, l2] {
print(x[0]);
}
}

CodeProber


ForStmt.*.nullnessValue [7:3→11:3]
Found 13 nodes
ArrayLiteralExpr [7:12→7:20]
.nullnessValue [7]

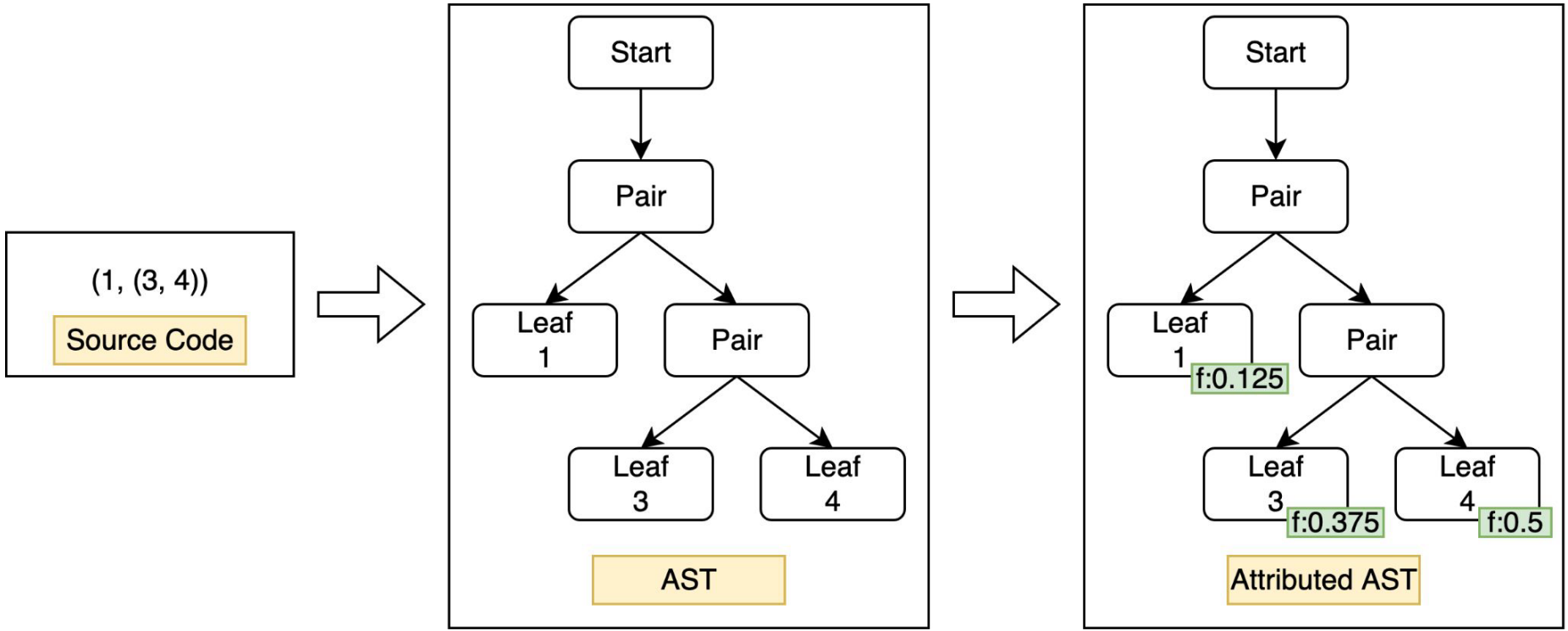

AST [11:3→11:23]
VarDecl
IdDecl: IdDecl
DeclaredType: Opt
Initializer: Opt
IntType
CallExpr
IdUse: IdUse
Actual: List
IntConstant


ArrayLiteralExpr.slowTask [4:12→4:19]
Stop
Property evaluation is taking a while, status below:


▼82 traces available, click to expand
▼Access.nullnessIn() = (empty str)
▼Access.pred()
▶Access.entry() = Entry
▶Entry.predMap() = {lang.ast.En
▼Access.decl() = VarDecl

Fractions (Again!)

Goal: Compute `f` for each `Leaf`, where `f` is the `Leaf`'s fraction of all values.



Implementation Session: Fractions

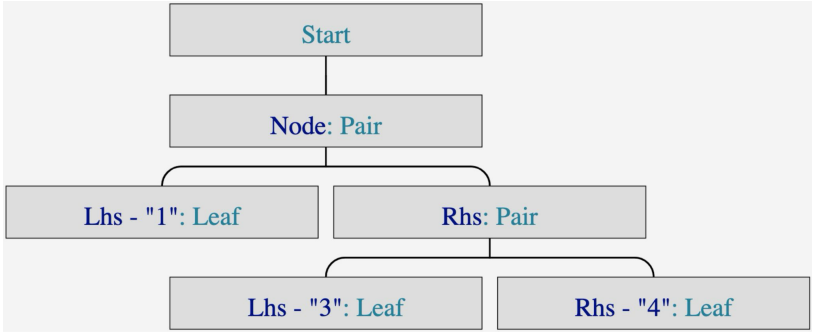
On-Demand Evaluation

```
Start ::= Node;
abstract Node;
Pair : Node ::= Lhs:Node Rhs:Node;
Leaf : Node ::= <Val:Integer>;
```

Our implementation

```
syn float Leaf.fraction() = (float)getVal() / sum();
syn int Start.sum() = getNode().partsum();
inh int Node.sum();
eq Start.getNode().sum() = sum();
eq Pair.getLhs().sum() = sum();
eq Pair.getRhs().sum() = sum();
syn int Node.partsum();
eq Leaf.partsum() = getVal();
eq Pair.partsum() = getLhs().partsum() + getRhs().partsum();
```

Input AST



Main function:

```
Start s = ..;
Leaf l1 = ..; Leaf l3 = ..;
System.out.println(l1.fraction());
System.out.println(l3.fraction());
```

Challenge: In what order are things evaluated?

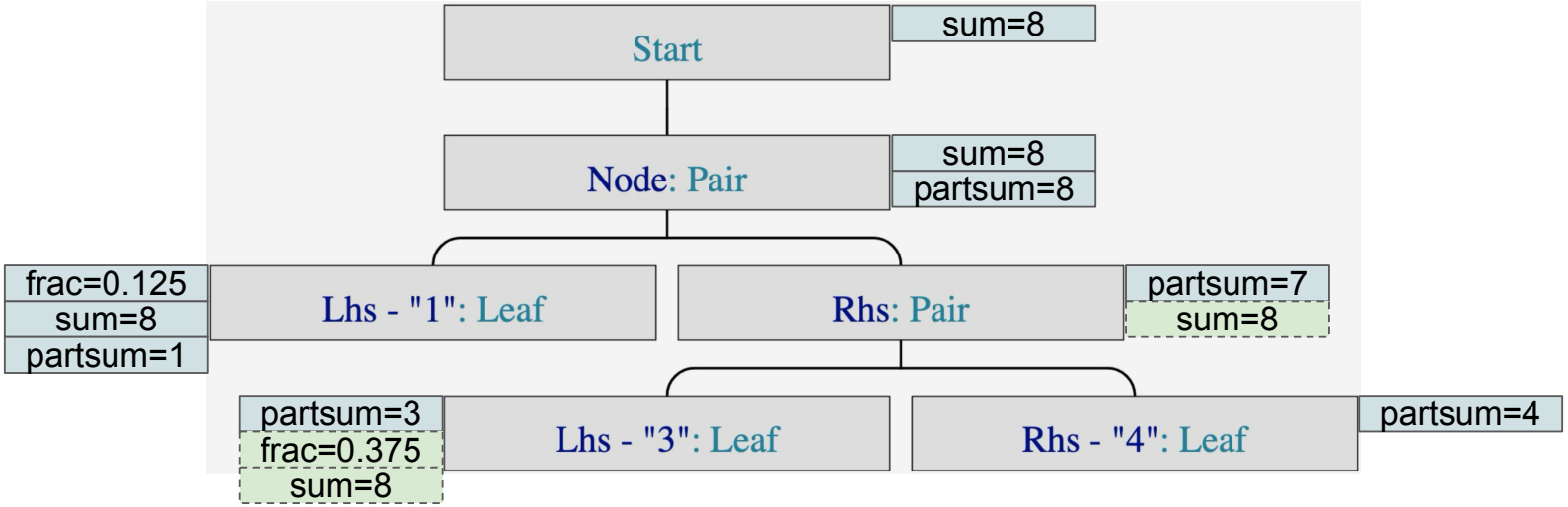
Recursive evaluation algorithm with memoization

If not cached
  find the equation
  compute its right-hand side
  cache the value
fi
Return the cached value

Evaluation result

solid blue = evaluated during “leaf1.fraction()”

dashed green = evaluated during “leaf3.fraction()”



Things to note:

- When calling an attribute like “leaf1.fraction()”, JastAdd will compute necessary dependencies automatically. This means you can access attributes in any order you want, no setup/preparation needed.
- During “leaf3.fraction()”, only 3 new values were computed. Most dependencies were already cached.
- Very little was computed for “Leaf 4”. Unless explicitly requested, JastAdd/on-demand evaluation will not compute anything. This is good for performance!

After break:

How does Rust test their compiler?
Does CodeProber work with “real” compilers?
For all this, and more, stay tuned!

Compiler Testing

Hypothesis & Goal (active research)

- Compilers and program analyzers are mostly tested using end-to-end tests
  - Example: the “.in” and “.expected” files you work with
  - This is popular in part because unit tests are inconvenient to create/maintain
- Unit tests are used & seen as useful in nearly all other parts of software engineering
- Can we improve unit testing for compilers/program analyzers?

How are programming languages* tested?

Only looked at Rust so far

Rust repo

- 2.8m lines of rust code
  - 570k compiler, 403k LSP implementation

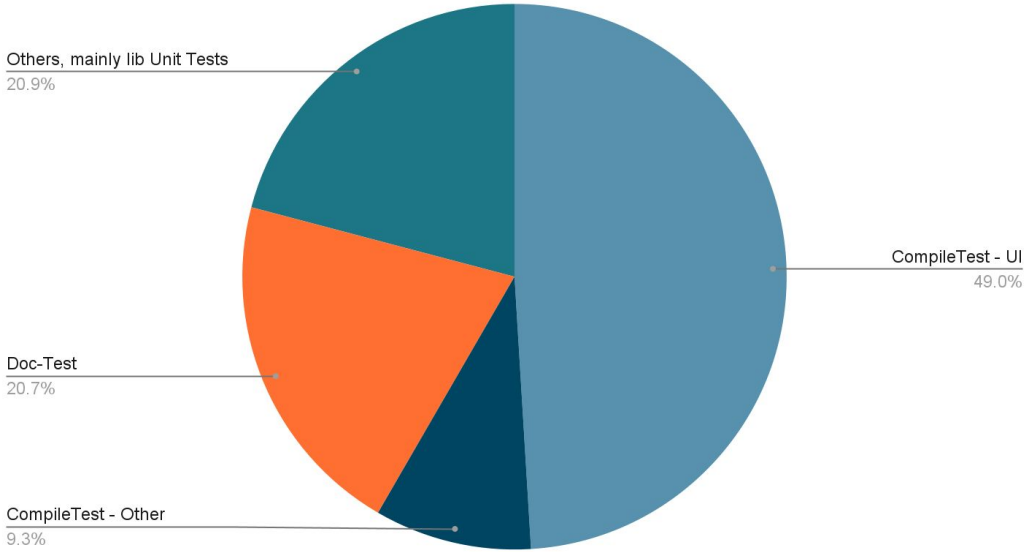
```
⚡ repo/rust: cloc . | grep Rust
Rust                                 34062         393450         614583        2852329

⚡ repo/rust: cloc compiler | grep Rust
Rust                                  1908          69070         114112         571961

⚡ repo/rust: cloc src/tools/rust-analyzer | grep Rust
Rust                                  1329          43846          34436         403434
```

Rust Test Breakdown

- I ran all tests on my laptop
  - 37709 tests in 43 minutes
- 22007 compiler tests
  - 18490, or 84% are “ui”



Compiler “UI” ~= Terminal

- A “UI” test contains two files, for example:

super-at-top-level.rs

tests > ui > super-at-top-level.rs > ...

```
1  use super::f; //~ ERROR there are too many
   leading `super` keywords
2
3  fn main() {
4  }
```

super-at-top-level.stderr

tests > ui > super-at-top-level.stderr

```
1  error[E0433]: failed to resolve: there are
   too many leading `super` keywords
2    --> $DIR/super-at-top-level.rs:1:5
3     |
4  LL | use super::f;
5     |     ^^^^^ there are too many leading
      `super` keywords
6
7  error: aborting due to 1 previous error
8
9  For more information about this error, try
   `rustc --explain E0433`.
```

- Basically an end-to-end test
- Similar to “.in” and “.expected” files

UI Testing Pro/Con

Pro

- Easy
  - Easy to write
  - Easy to understand the purpose of the test
- Shows that the system works end-to-end

Con

- Involves large part of the system
  - Harder to understand what code is involved
  - Regressions harder to fix
- Cannot be used during development
  - Can’t do end-to-end testing if the two “ends” don’t exist yet!
- Mainly used for error messages
  - What about testing non-failure functionality?

Problem with Unit-Testing Compilers

ExtendJ type inference unit test

Arrange

```
String code = "import java.util.*;"
    + "public class Test {"
    + "  void m1(List<? extends String> names) {"
    + "    names.stream().mapToInt(name -> name.length());"
    + "  }"
    + "}";
CompilationUnit cu = parseCompilationUnit(code);
```

Traversal

```
MethodDecl m1 = (MethodDecl) cu.getTypeDecl(0).getBodyDecl(0);
Dot dot1 = (Dot) ((ExprStmt) m1.getBlock().getStmt(0)).getExpr();
Dot dot2 = (Dot) dot1.getRight();
MethodAccess mapToInt = (MethodAccess) dot2.getRight();
LambdaExpr lambda = (LambdaExpr) mapToInt.getArg(0);
TypeDecl anonType = lambda.toClass().type();
for (BodyDecl decl : anonType.getBodyDeclList()) {
  if (decl instanceof MethodDecl) {
```

Act

```
    MethodDecl method = (MethodDecl) decl;
```

Assert

```
    String typeSignature = method.methodTypeSignature();
    System.out.format("Type signature of %s: %s%n", method.name(),
    typeSignature);
  }
}
```

This part is annoying and boring to write. Also a maintenance problem.

What would you prefer?

```
import java.util.*;
public class Test {
  void m1(List<? extends String> names) {
    names.stream().mapToInt(name -> name.length()); // [[$list:=LambdaExpr.toClass.type.getBodyDeclList]]
  }
}
// [[$list.getNumChild=1]]
// [[$list.firstChild.methodTypeSignature=(Ljava/lang/String;)I]]
```

```
String code = "import java.util.*;"
    + "public class Test {"
    + "  void m1(List<? extends String> names) {"
    + "    names.stream().mapToInt(name -> name.length());"
    + "  }"
    + "}";
CompilationUnit cu = parseCompilationUnit(code);
MethodDecl m1 = (MethodDecl) cu.getTypeDecl(0).getBodyDecl(0);
Dot dot1 = (Dot) ((ExprStmt) m1.getBlock().getStmt(0)).getExpr();
Dot dot2 = (Dot) dot1.getRight();
MethodAccess mapToInt = (MethodAccess) dot2.getRight();
LambdaExpr lambda = (LambdaExpr) mapToInt.getArg(0);
TypeDecl anonType = lambda.toClass().type();
for (BodyDecl decl : anonType.getBodyDeclList()) {
  if (decl instanceof MethodDecl) {
    MethodDecl method = (MethodDecl) decl;
    String typeSignature = method.methodTypeSignature();
    System.out.format("Type signature of %s: %s%n", method.name(),
    typeSignature);
  }
}
```

…+20 lines outside of screen

Testing Style Recommendations: Do Both!

- During development, create many smaller unit tests in CodeProber for each piece of functionality

```
// [[Leaf.fraction=0.5]]
```

- Once a larger feature works, create one or more E2E tests too
  - ..either in CodeProber

```
// [[Program.errors~=symbol 'b' is not declared!]]
```

  - ..or using .in/.expected

Demo: Rust(-Analyzer) + CodeProber

Coding Session: Extending ExtendJ

In Conclusion

- CodeProber usage during labs is optional, but recommended
  - It is most effective in Lab 4, but can be used in 5 & 6 too
- Please write tests
- Want to try CodeProber with ExtendJ to solve some riddles involving Java?
  - https://github.com/Kevlanche/codeprober-playground