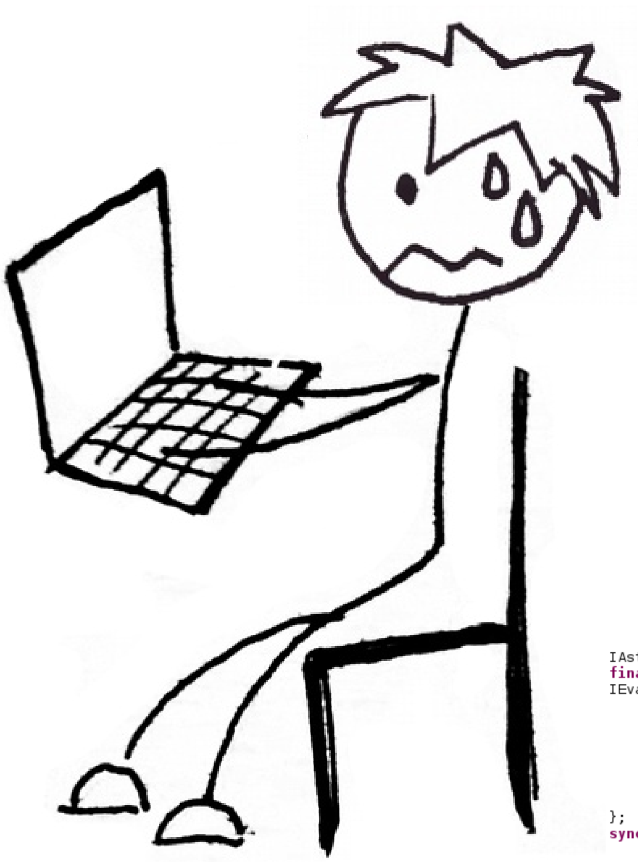
CodeHint: Dynamic and Interactive Synthesis of Code Snippets

Joel Galenson, Ras Bodik, Koushik Sen
UC Berkeley

Motivation

Specification



Test cases

```
IAstEvaluationEngine engine = getASTEvaluationEngine(stack);
final IEvaluationResult[] results = new IEvaluationResult[1];
IEvaluationListener listener = new IEvaluationListener() {
    @Override
    public void evaluationComplete(IEvaluationResult result) {
        synchronized (stack) {
            results[0] = result;
            stack.notifyAll();
        }
    }
};
synchronized(stack) {
    if (stack.isTerminated())
        return null;
    engine.evaluate(stringValue, stack, listener, DebugEvent.EVALUATION_IMPLICIT, false);
    try {
        stack.wait(TimeoutChecker.TIMEOUT_TIME_MS);  // Timeout the execution.
    } catch (InterruptedException e) {
        if (results[0] == null)
            throw new RuntimeException(e);
    }
}
IEvaluationResult result = results[0];
```

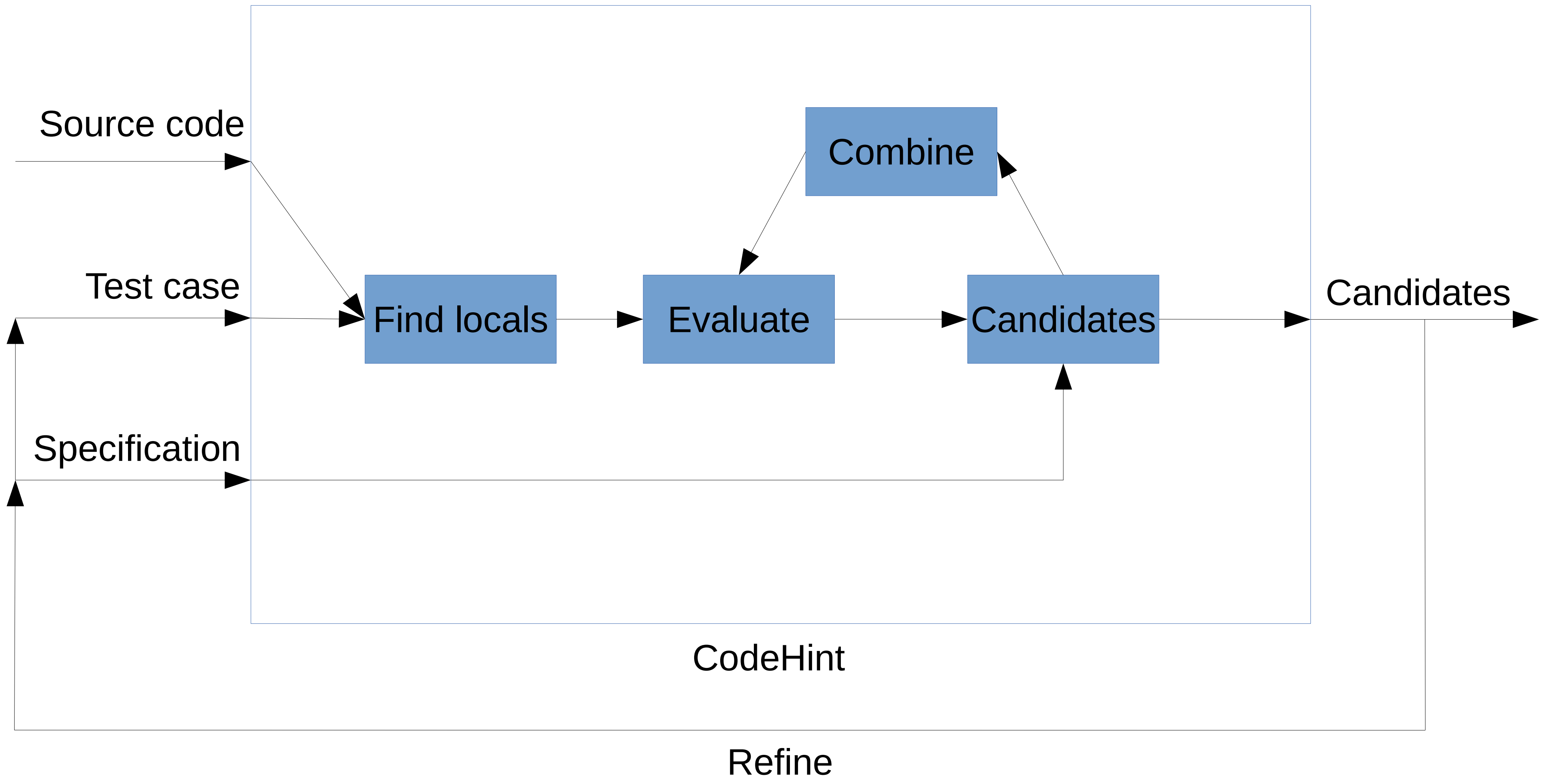
Desired type

Pictures by Mangpo Phitchaya Phothilimthana

CodeHint: Autocomplete for the modern age

- Autocomplete is useful but very limited.
- Our improvements:
  - Being dynamic
  - General specifications
  - Synthesis

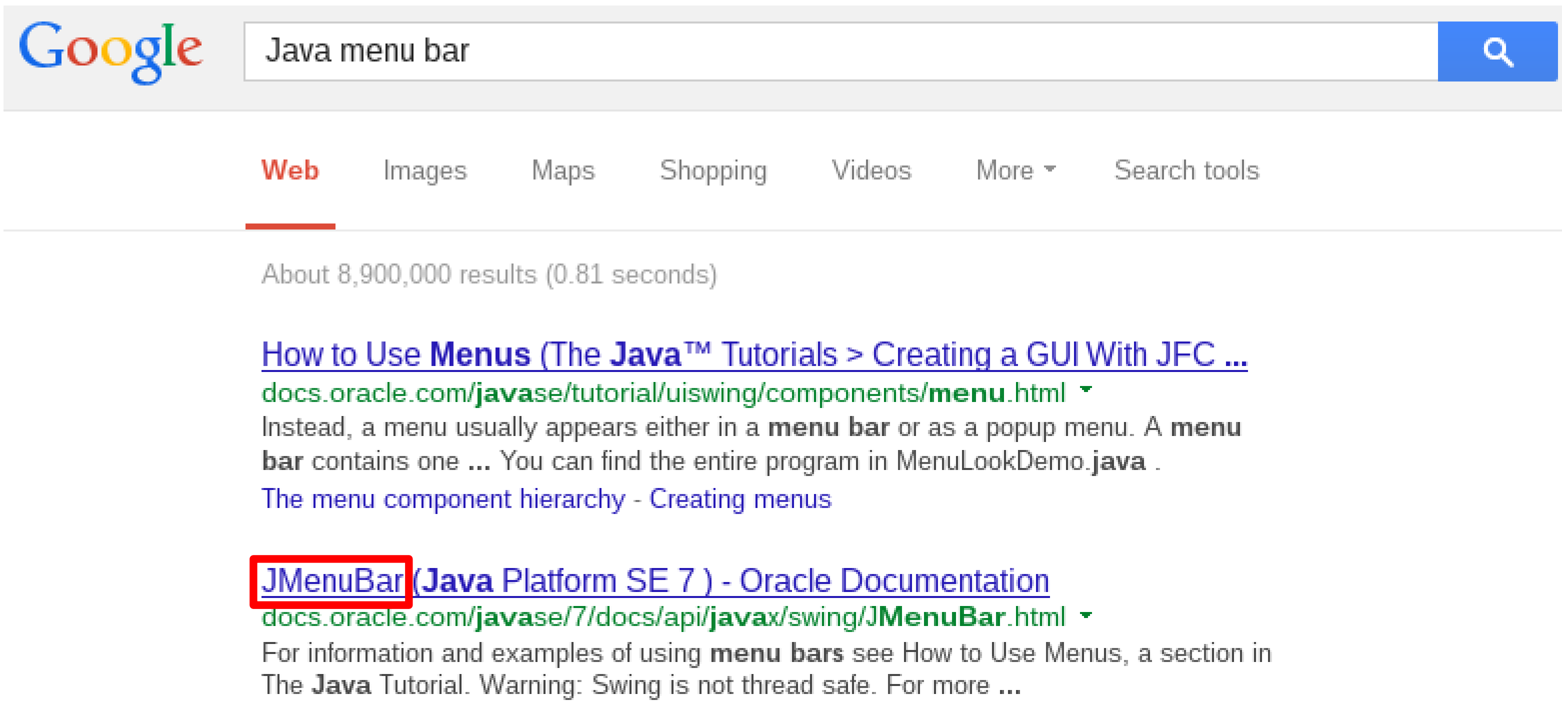
Overview


Source code
Test case
Specification
Combine
Find locals
Evaluate
Candidates
Candidates
CodeHint
Refine

Example: The problem

```java
final JComponent tree = makeTree();
tree.addMouseListener(new MouseAdapter() {
  public void mousePressed(MouseEvent e) {
    int x = e.getX(), y = e.getY();
    // Get the menu bar.
  }
});
```

Example: First step

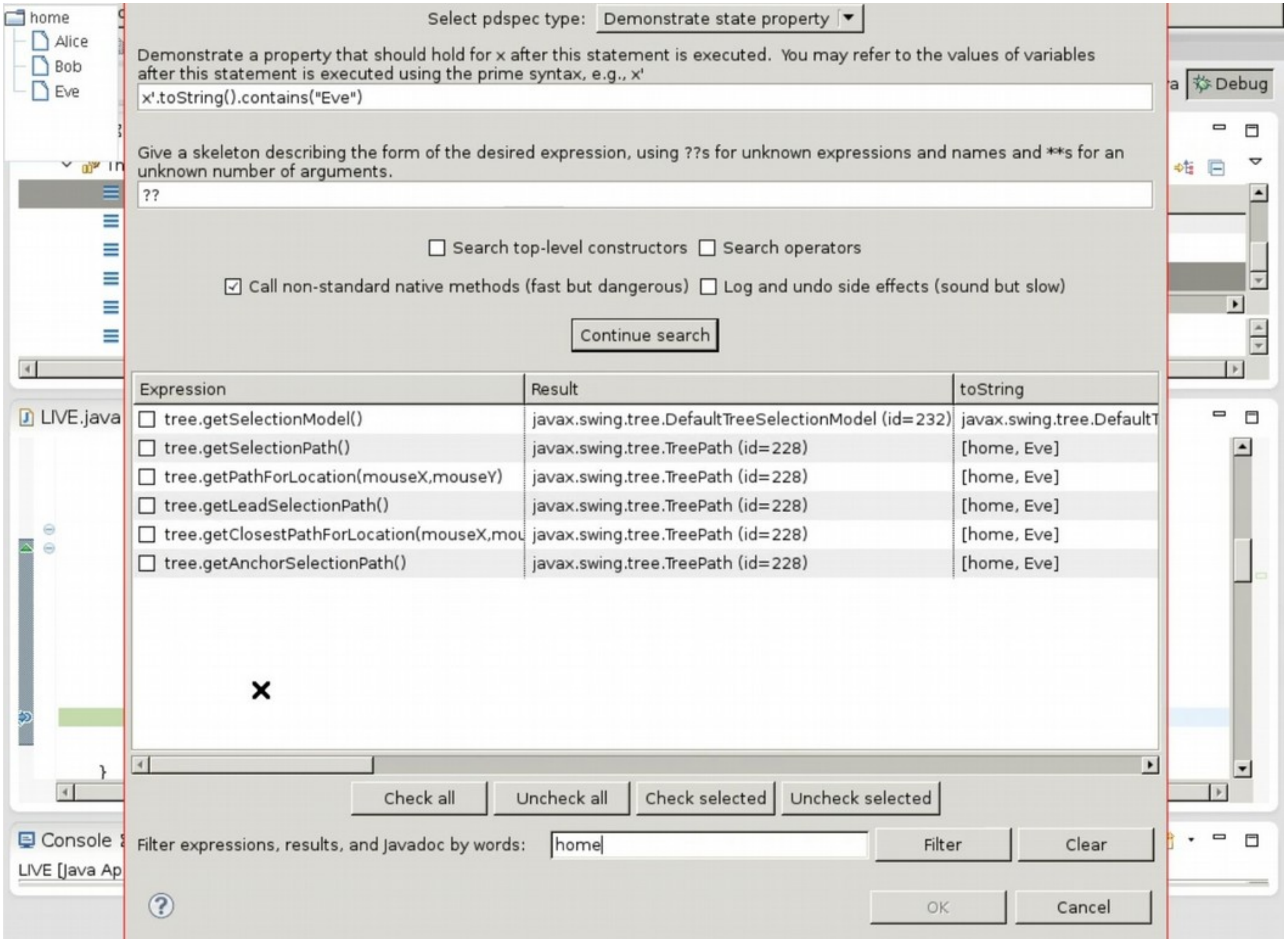



```java
final JComponent tree = makeTree();
tree.addMouseListener(new MouseAdapter() {
  public void mousePressed(MouseEvent e) {
    int x = e.getX(), y = e.getY();
    // Get the menu bar.
  }
});
```

"Find a JMenuBar"

```java
rv instanceof JMenuBar
```

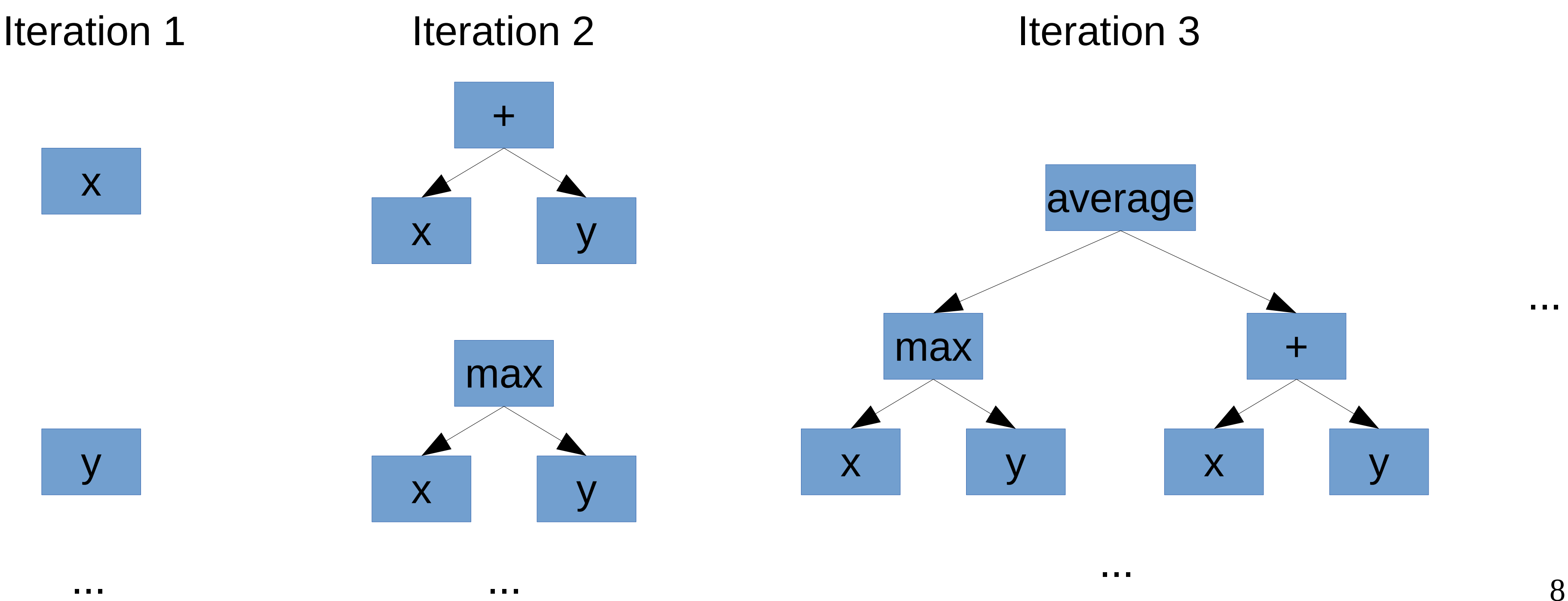
Demo



home
Alice
Bob
Eve
LIVE.java
Console
LIVE [Java Ap
Debug
Select pdspec type: Demonstrate state property
Demonstrate a property that should hold for x after this statement is executed. You may refer to the values of variables after this statement is executed using the prime syntax, e.g., x'
x'.toString().contains("Eve")
Give a skeleton describing the form of the desired expression, using ??s for unknown expressions and names and **s for an unknown number of arguments.
??
Search top-level constructors
Search operators
Call non-standard native methods (fast but dangerous)
Log and undo side effects (sound but slow)
Continue search
Expression
Result
toString
tree.getSelectionModel()
javax.swing.tree.DefaultTreeSelectionModel (id=232)
javax.swing.tree.DefaultT
tree.getSelectionPath()
javax.swing.tree.TreePath (id=228)
[home, Eve]
tree.getPathForLocation(mouseX,mouseY)
javax.swing.tree.TreePath (id=228)
[home, Eve]
tree.getLeadSelectionPath()
javax.swing.tree.TreePath (id=228)
[home, Eve]
tree.getClosestPathForLocation(mouseX,mou
javax.swing.tree.TreePath (id=228)
[home, Eve]
tree.getAnchorSelectionPath()
javax.swing.tree.TreePath (id=228)
[home, Eve]
Check all
Uncheck all
Check selected
Uncheck selected
Filter expressions, results, and Javadoc by words:
home
Filter
Clear
OK
Cancel

Basic algorithm

- Find local variables, do BFS over Java statements, show user those that pass spec.
  - Actually evaluate these statements, including file I/O, reflection, etc.



`rv instanceof JMenuBar` `rv instanceof JMenuBar`

First iteration

- Find local variables.

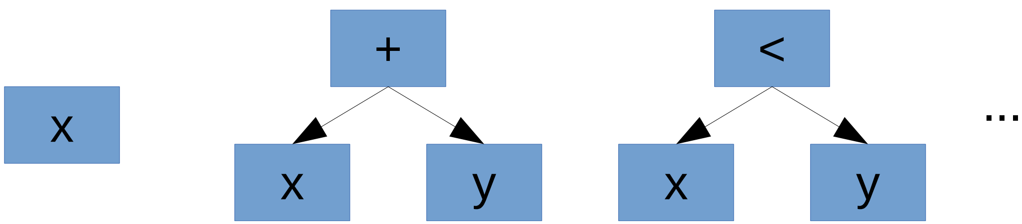
```
final JComponent tree = makeTree();
tree.addMouseListener(new MouseAdapter() {
  public void mousePressed(MouseEvent e) {
    int x = e.getX(), y = e.getY();
    // Get the menu bar.
  }
});
```

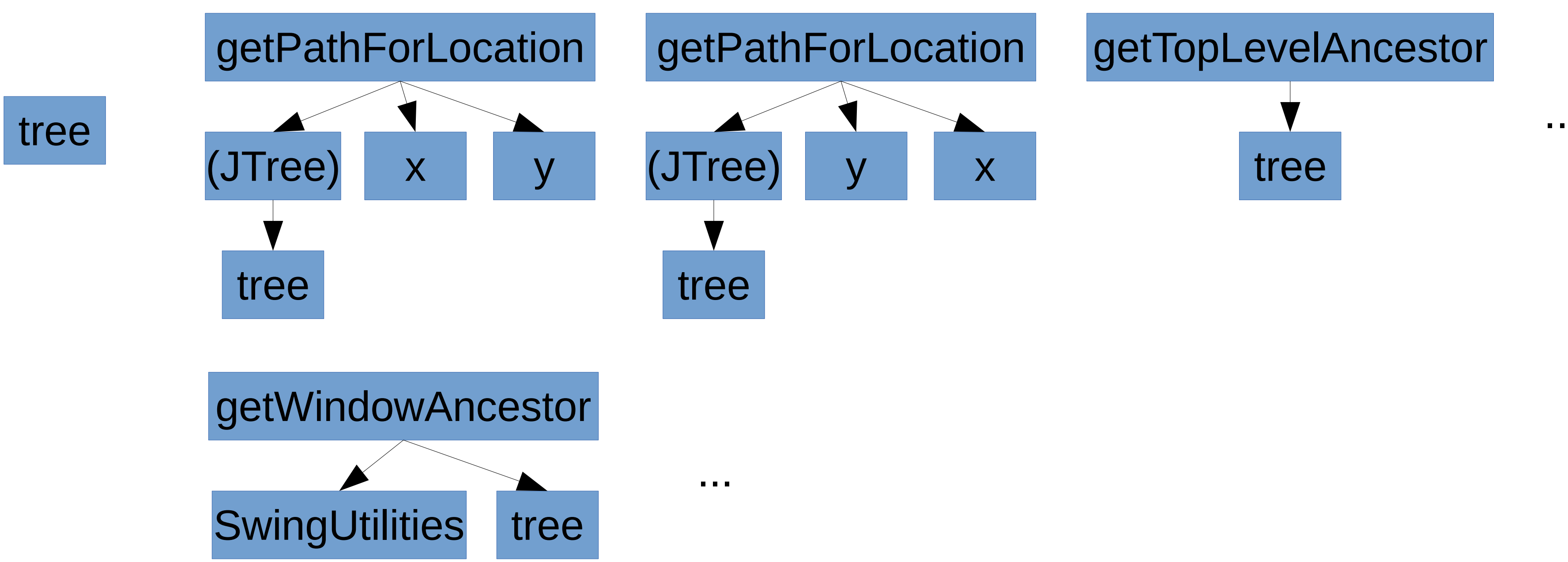
x | y | tree | this | null | ...

Second iteration

- Get each expression's type and combine it with others in type-correct ways.

```java
final JComponent tree = makeTree();
tree.addMouseListener(new MouseAdapter() {
  public void mousePressed(MouseEvent e) {
    int x = e.getX(), y = e.getY();
    // Get the menu bar.
  }
});
```

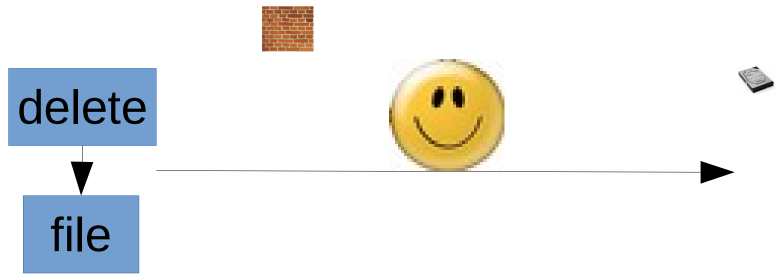




Side effects

- Handle in-memory and external effects.

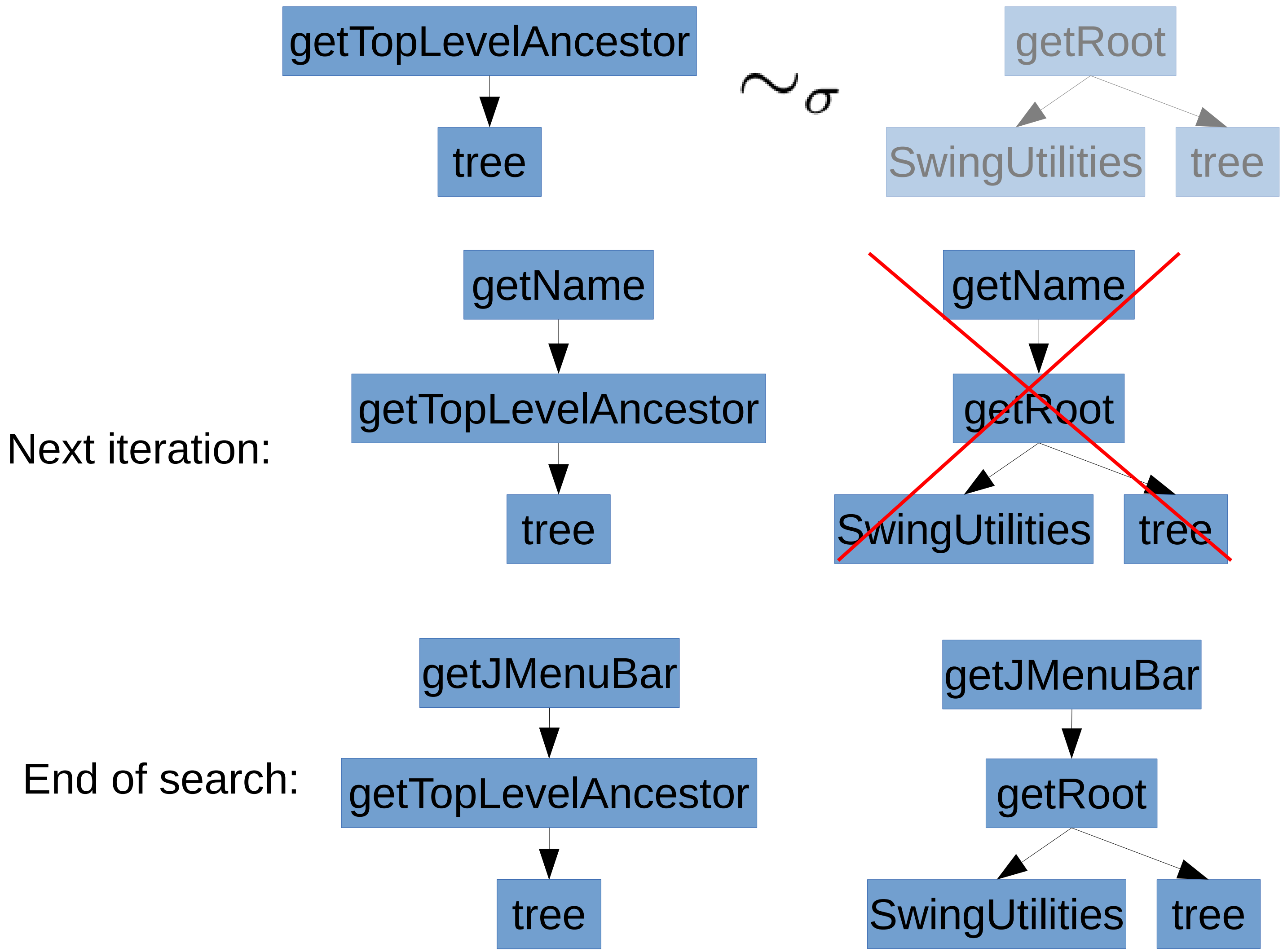
External:



SecurityManager

Equivalence classes

- Group equivalent code to avoid unneeded work.



Probabilistic model

- Mined 10MLOC to build model of likely code.

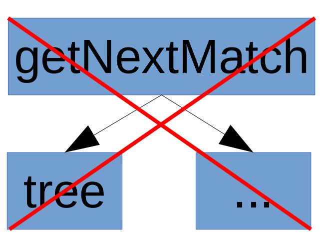
Probability of accessing method/field m on type T:

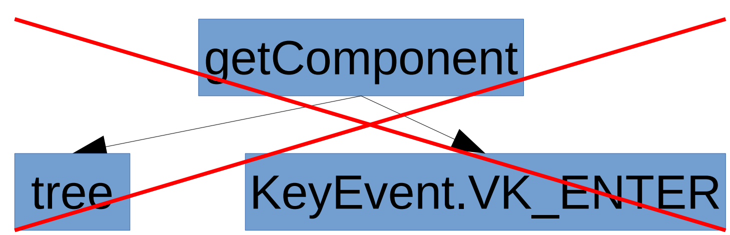
$$P(m) = P(m|T)P(T) = \frac{\text{\# accesses of } m \text{ on } T}{\text{\# of accesses on } T} \times \frac{\text{\# of accesses on } T}{\text{\# of accesses}} = \frac{\text{\# accesses of } m \text{ on } T}{\text{\# of accesses}}$$

Probability of using constant c as argument i to method m:

$$P(c,m,i) = P(c,m,i|c)P(c) = \frac{\text{\# uses of } c \text{ on method } m \text{ at index } i}{\text{\# of uses of } c} \times \frac{\text{\# of uses of } c}{\text{\# of uses}} = \frac{\text{\# uses of } c \text{ on method } m \text{ at index } i}{\text{\# of uses}}$$

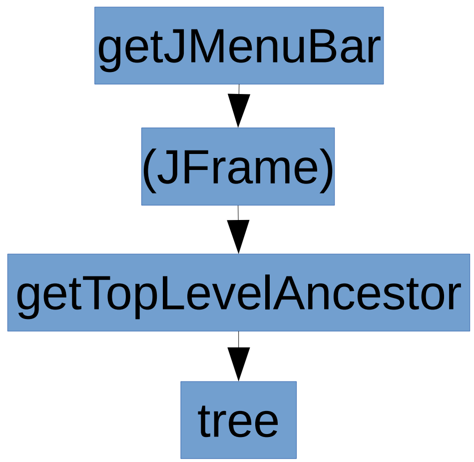
- Use to avoid unlikely expressions.

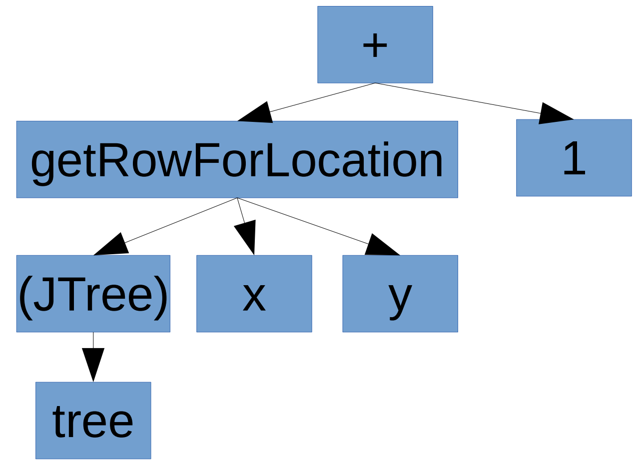




Third iteration and result

Third iteration:

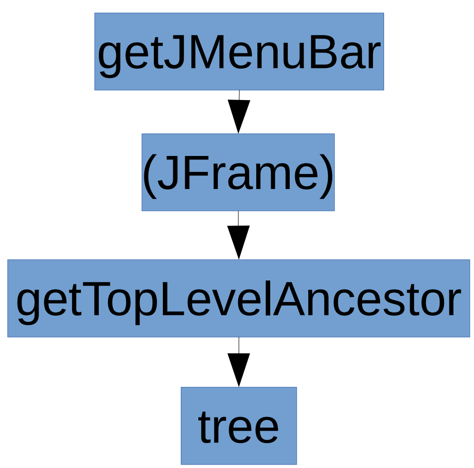


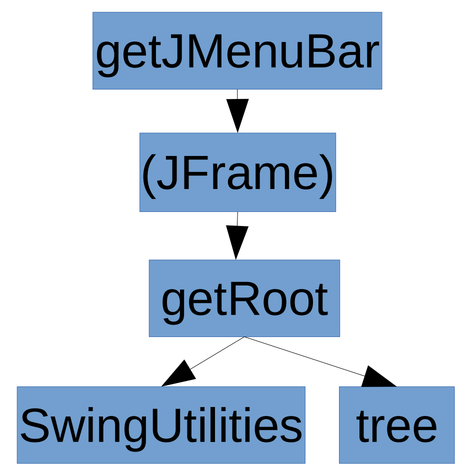


...

`rv instanceof JMenuBar`   `rv instanceof JMenuBar`

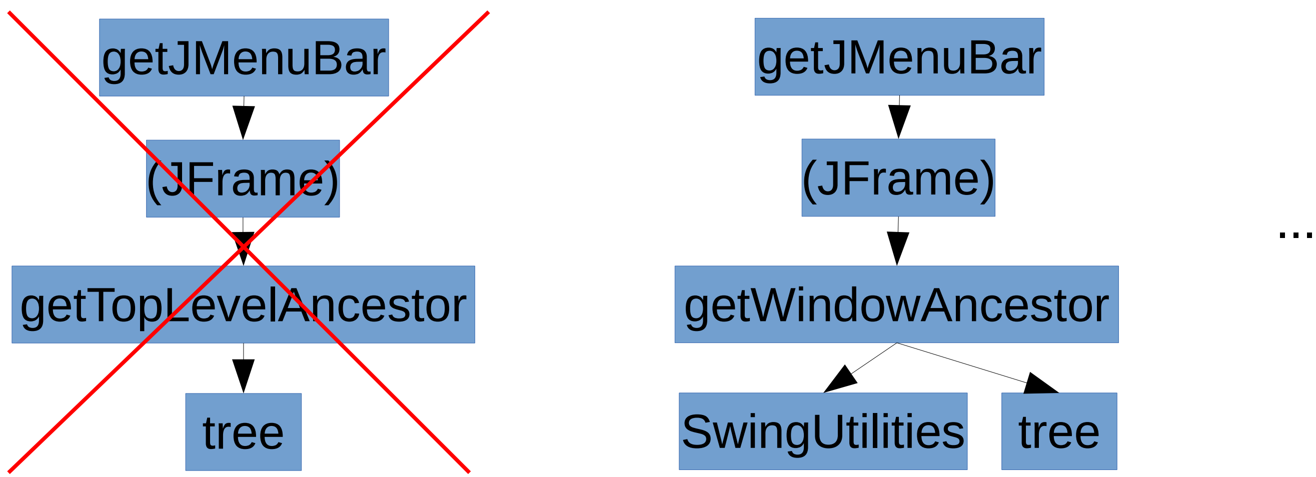
Result:



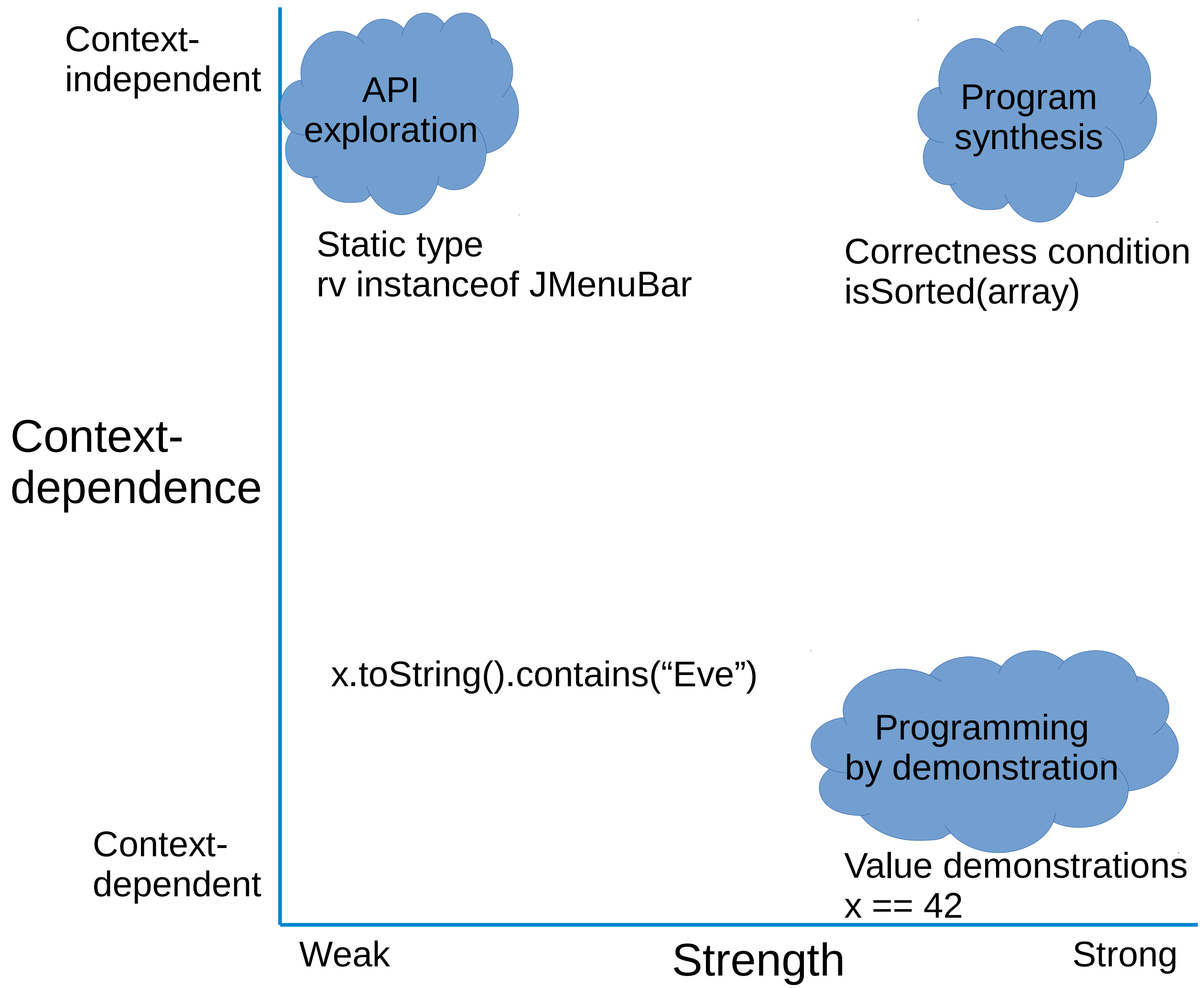


Refinement

- Users can give another demonstration in a different state to refine the results.



Synthesis from specifications



Empirical results

Normal algorithm

	Iteration 2		Iteration 3		Iteration 4	
	#	Time	#	Time	#	Time
P 1	34	0.1	611	0.6	19644	6.1
P 2	52	0.1	727	0.7	[illegible]	[illegible]
P 3	53	0.1	1091	[illegible]	[illegible]	[illegible]
P 4	7	0.1	[illegible]	[illegible]	[illegible]	[illegible]
P 5	22	0.1	[illegible]	[illegible]	[illegible]	[illegible]
S 1	8	0.2	[illegible]	[illegible]	[illegible]	[illegible]
S 2	12	0.1	[illegible]	[illegible]	[illegible]	[illegible]
S 3	70	1.0	[illegible]	[illegible]	[illegible]	[illegible]
S 4	103	0.3	[illegible]	[illegible]	[illegible]	[illegible]
S 5	32	0.2	[illegible]	[illegible]	[illegible]	[illegible]
R 1	20	0.1	[illegible]	[illegible]	[illegible]	[illegible]
R 2	12	0.0	137	[illegible]	[illegible]	[illegible]
R 3	8	0.2	20	0.2	[illegible]	[illegible]
R 4	6	0.0	19	0.1	68	0.3
R 5	24	0.2	226	0.4	1998	2.3
Avg	30.9	0.2	397.9	0.7	17029.9	4.8
Med	22	0.1	239	0.6	2044	2.3

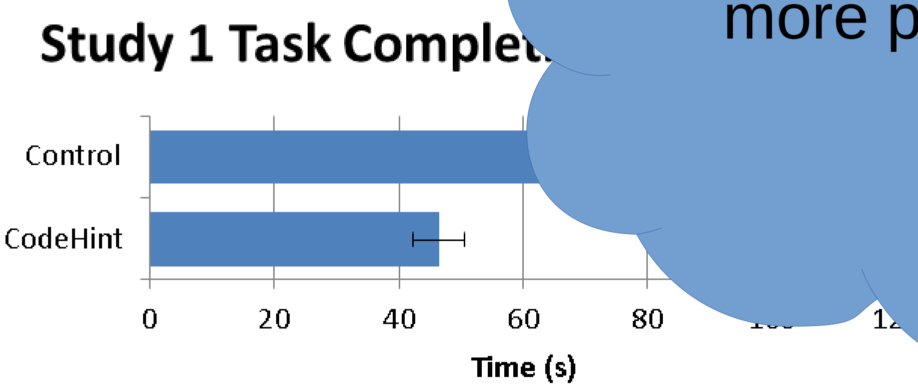
Our algorithms
well in practic

In real-world code, ~95% of expressions need ≤3 iterations and ~99% need ≤4.

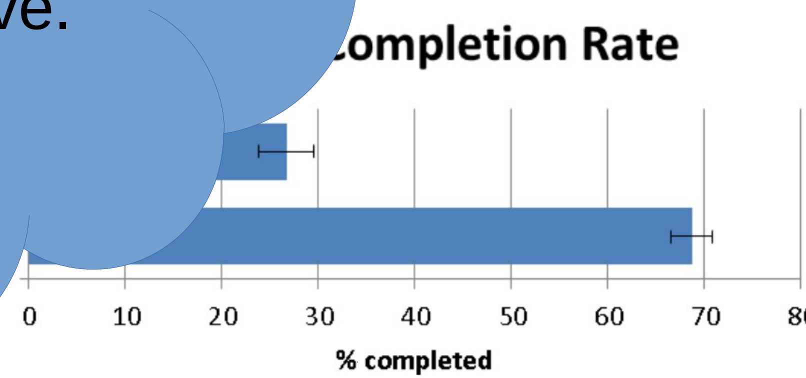
User studies

- Completed two user studies with 28 subjects.
- Found statistically-signifi… productivity improvement…
  - Fewer bug… …ed in less time.

CodeHint makes programmers more productive.



Lower is better

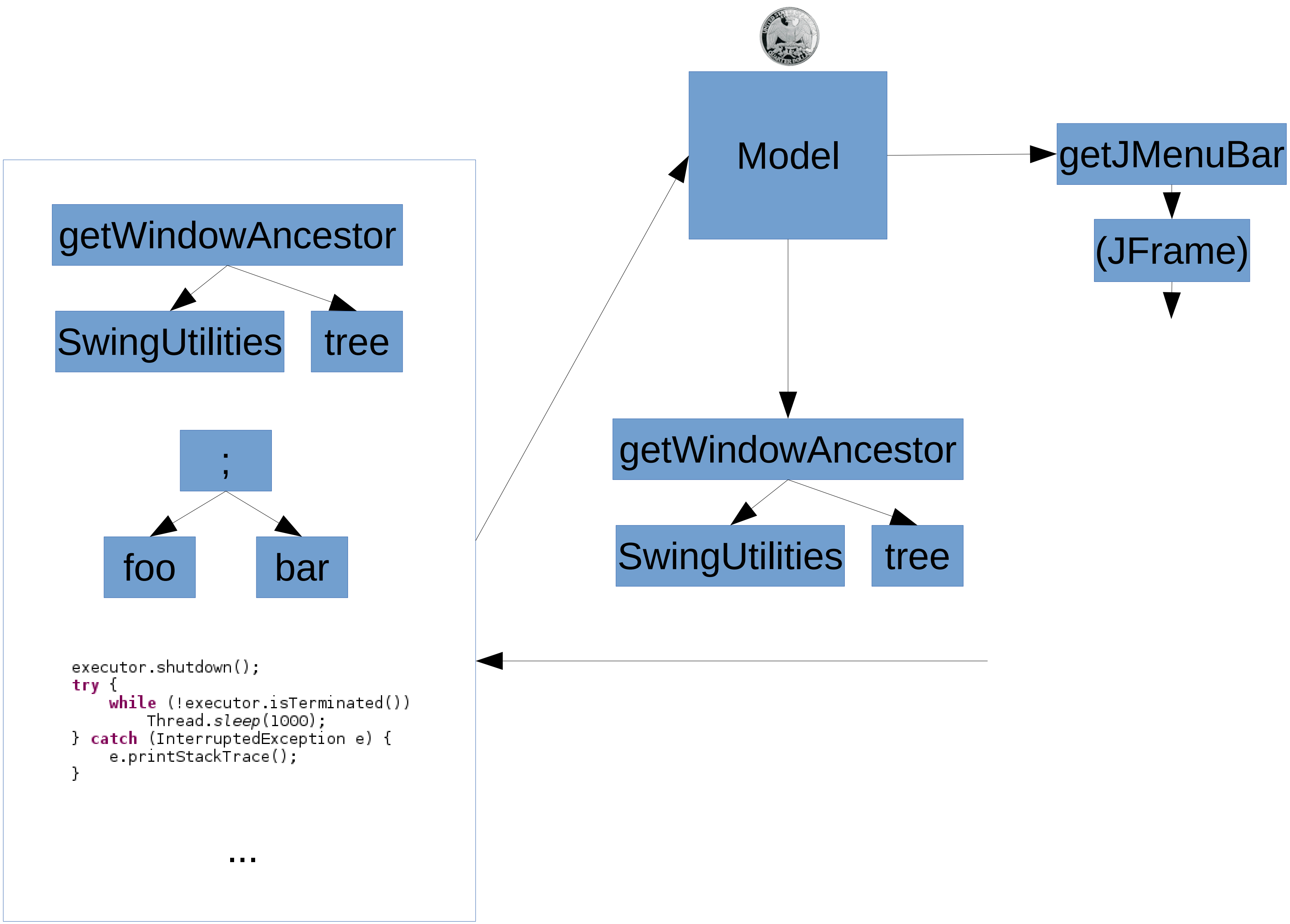


Higher is better

Future work

- Improving the probabilistic model
- CodeHint for JavaScript
  - https://github.com/jgalenson/codehint.js
  -
- Integrating symbolic techniques

Probabilistic search



Candidates

Summary

- Dynamic and interactive synthesis
- Autocomplete for the modern age
- User studies showed productivity improvements

Thanks!

https://jgalenson.github.io/codehint/