

# Advanced Reflection: MetaLinks

Marcus Denker, Inria

<http://marcusdenker.de>

**Lecture at VUB Brussels, March 30, 2023**

# What we know (I)

- Smalltalk is reflective
- Classes, Methods, Stack-Frames... are Objects
- Reflective API on all Objects

# Reflection in Smalltalk

- Reflection is based on the Metaclass model, thus it is inherently structural
- Behavioral Reflection limited to:
  - Method lookup on failure (`#doesNotUnderstand:`)
  - Reified stack (`thisContext`)

# Can we do better?

- A more fine-grained reflective mechanism seems to be missing
- Let's look again at a Method in the Inspector

# Inspector on a Method

The image shows a screenshot of a Ruby Playground interface. At the top, a text input field contains the code `OrderedCollection>>#do:`. Below this, an "Inspector on a CompiledMethod (OrderedCollection>>#do:)" window is open. The inspector has two panes. The left pane shows the Abstract Syntax Tree (AST) for the method, with the following structure:

- ▼ RBMethodNode(do: aBlock "Override the superclass for performan)
  - RBArgumentNode(aBlock)
  - ▼ RBSequenceNode(firstIndex to: lastIndex do: [ :index | aBlock val])
    - ▼ RBMessageNode(firstIndex to: lastIndex do: [ :index | aBlock val])
      - RBInstanceVariableNode(firstIndex)
      - RBInstanceVariableNode(lastIndex)
      - ▼ RBBlockNode([ :index | aBlock value: (array at: index) ])
        - RBArgumentNode(index)
        - ▼ RBSequenceNode(aBlock value: (array at: index))
          - ▼ RBMessageNode(aBlock value: (array at: index))
            - RBArgumentNode(aBlock)
            - ▶ RBMessageNode((array at: index))

The right pane shows the source code for the selected `RBMessageNode` object:

```
do: aBlock
  "Override the superclass for performance reasons."

  firstIndex to: lastIndex do: [ :index |
    aBlock value: (array at: index) ]
```

# Excursion: Pragma

```
- aNumber
  "Primitive. Subtract the argument from the receiver and answer with the
  result if it is a SmallInteger. Fail if the argument or the result is not a
  SmallInteger. Essential. No Lookup. See Object documentation
  whatIsAPrimitive."

  <primitive: 2>
  ^super - aNumber
```

- <someSelector: #hereJustLiterals arg: #yes>
- A Pragma is a selector + arguments (which are literals)
- They are annotations on methods

# Pragmas: API

```
(SmallInteger>>#-) pragmas.  
"ask a method for its pragmas"
```

```
Pragma pragmaCache.  
"cache of all pragmas and using method, inspect it"
```

```
Pragma allNamed: #primitive:  
"returns all primitive methods, fast due to cache"
```

# Pragmas: Usage

- Compiler Options
  - `<compilerOptions:  
#(+optionCleanBlockClosure)>`
- Primitives
- Menus
- Inspector
- more...



# What we know (II)

- There is an AST (Abstract Syntax Tree)
- The Pharo Smalltalk->Bytecode Compiler
- We have Compiler Plugins

# The AST

- AST = **A**bstract **S**yntax **T**ree
- Tree Representation of the Method
- Produced by the Parser (part of the Compiler)
- Used by all tools (refactoring, syntax-highlighting,...)

**Smalltalk compiler parse: 'test ^ (1+2) '**

# AST

- RBMethodNode                      Root
- RBVariableNode                    Variable (read and write)
- RBAssignmentNode                Assignment
- RBMessageNode                    A Message (most of them)
- RBReturnNode                     Return

# Inspect a simple AST

- A very simple Example

**Smalltalk compiler parse: 'test ^ (1+2)'**

The screenshot displays the Smalltalk Inspector interface. The title bar reads "Inspector on a RBMethodNode (test ^ 1 + 2)". There are two panes:

- Left Pane:** Shows a tree view of the AST. The root is `RBMethodNode(test ^ 1 + 2)`. It contains a `RBSequenceNode(^ 1 + 2)`, which contains an `RBReturnNode(^ 1 + 2)`. This node contains an `RBMessageNode(1 + 2)`, which contains two `RBLiteralValueNode` objects: `RBLiteralValueNode(1)` and `RBLiteralValueNode(2)`. The `RBLiteralValueNode(2)` node is currently selected and highlighted in blue.
- Right Pane:** Shows the source code `test ^ (1+2)`. The `2` in `(1+2)` is highlighted in blue, corresponding to the selected node in the left pane.

# AST: Navigation

- To make it easy to find and enumerate nodes, there are some helper methods
- CompiledMethod has: `#sendNodes`,  
`#variableNodes`, `#assignmentNodes`
- Every AST node has `#nodesDo:` and `#allChildren`

# AST: Visitor

- `RBProgramNodeVisitor`: Visitor Pattern for the AST
- Make subclass, override `visit...` methods
- Let's see it in action: Count Message sends

# Demo: Visitor

# Repeat: The AST

- AST = **A**bstract **S**yntax **T**ree
- Tree Representation of the Method
- Produced by the Parser (part of the Compiler)
- Used by all tools (refactoring, syntax-highlighting,...)

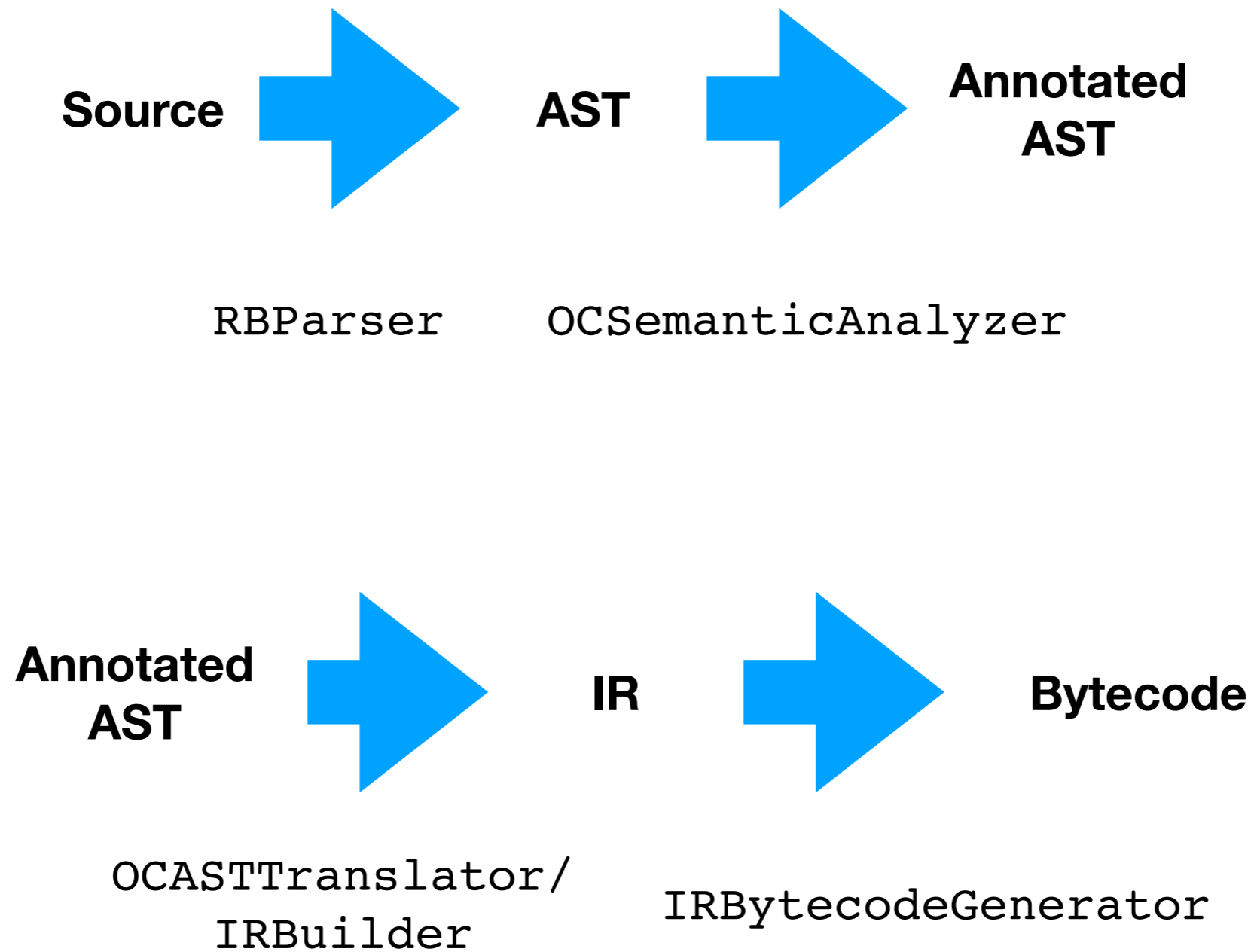
**Smalltalk compiler parse: 'test ^ (1+2)'**



# The Compiler

- `Smalltalk compiler -> Compiler Facade`
- Classes define the compiler to use
  - You can override method `#compiler`
- Behind: Compiler Chain

# The Compiler



# AST Integration

- Originally just internal to the compiler
- Pharo:
  - send `#ast` to a method to get the AST
  - Cached for persistency.

```
(Point>>#x) ast == (Point>>#x) ast  
→ true
```

# AST Integration

- We can navigate from execution to AST
- Example:

```
[ 1 + 2 ] sourceNode ==
```

```
  thisContext method sourceNode blockNodes first
```

# Compiler: Extensible

- All parts can be subclassed
- Compiler instance can be setup to use the subclass for any part (parser, name analysis, translator...)
- enable for a class only by implementing #compiler on the class side

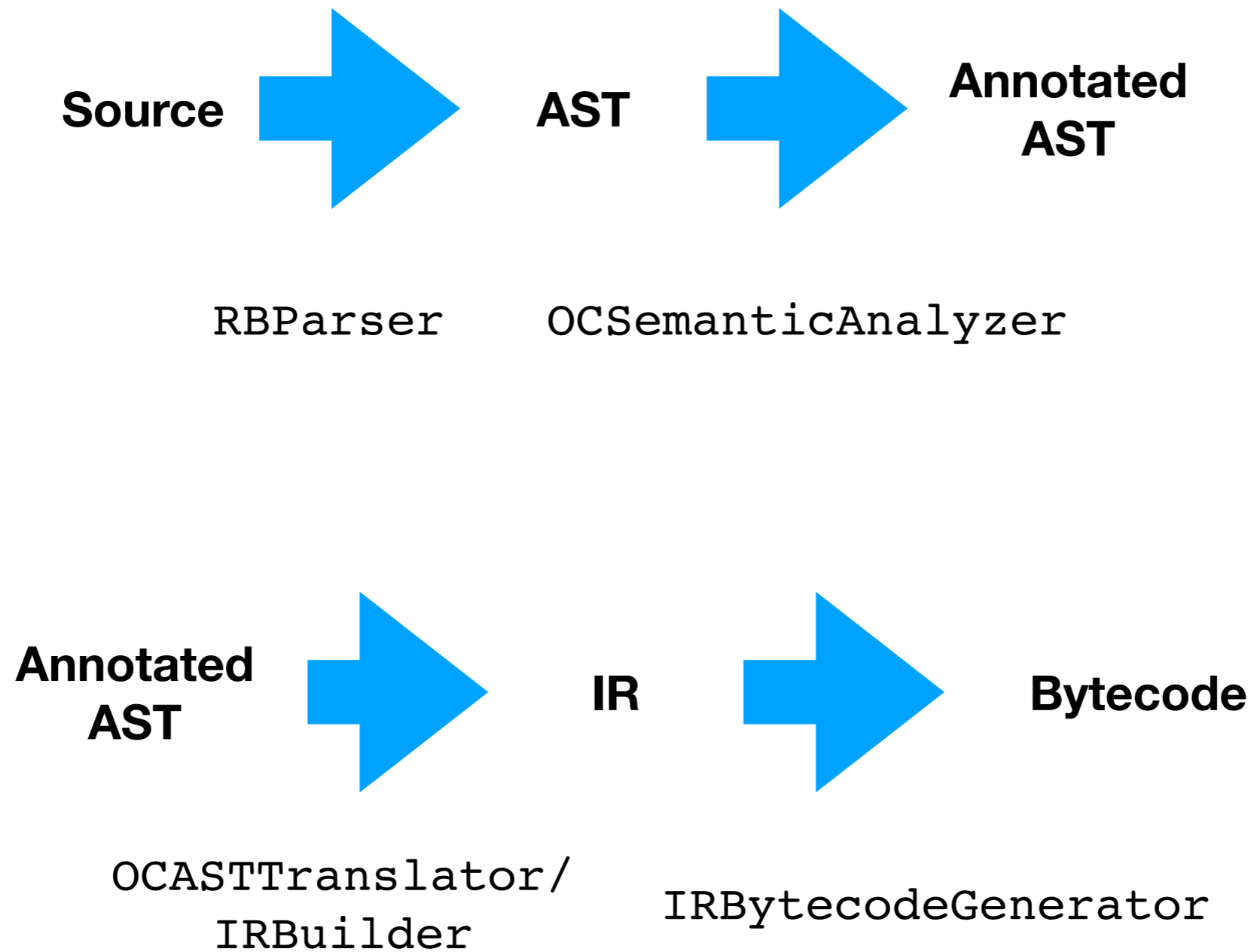
# Compiler Plugins

- The AST can be easily transformed
- We added a Plugin architecture to the Compiler
- enable for a class only by implementing:

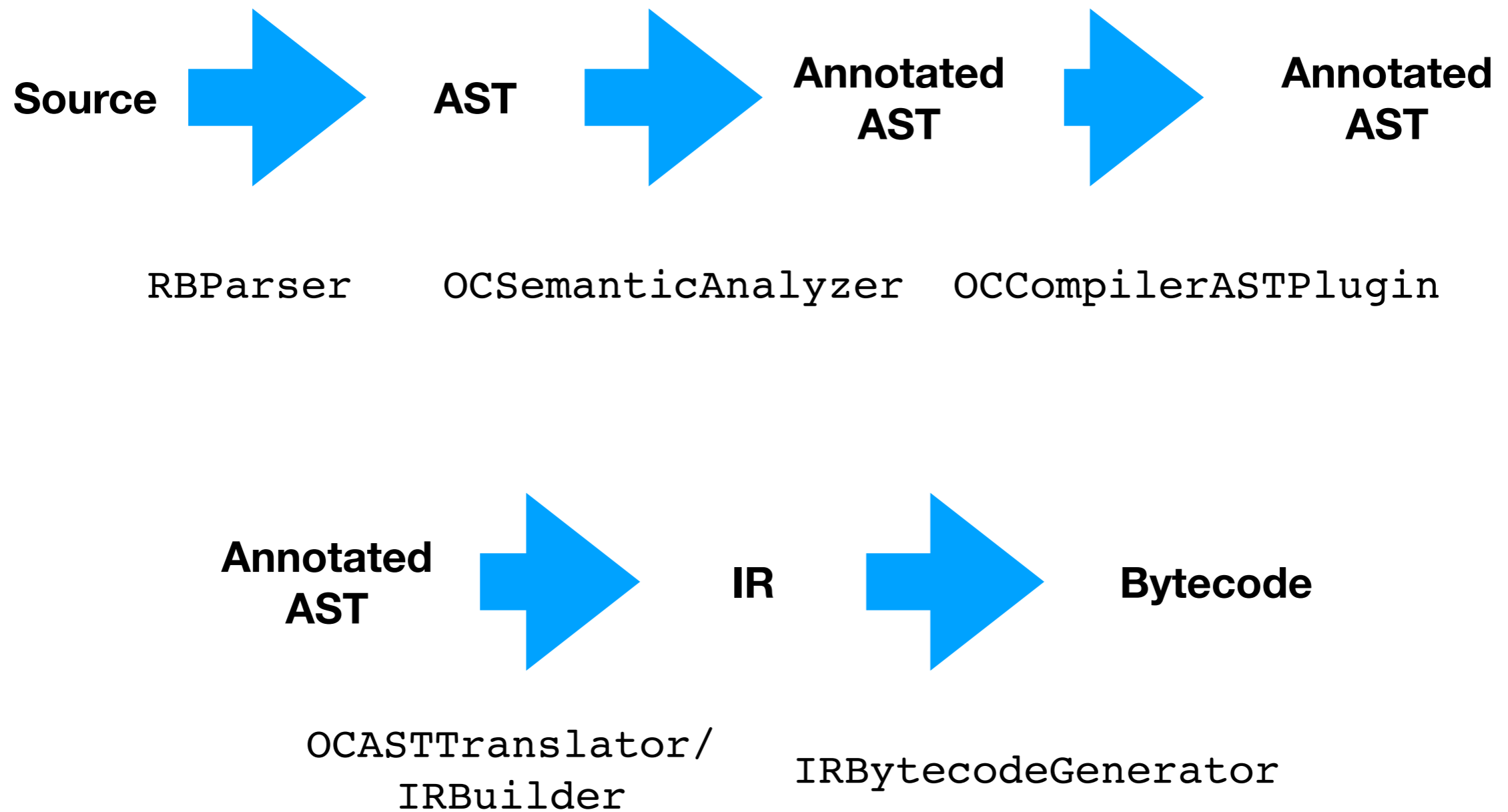
compiler

^super compiler addPlugin: MyPlugin

# The Compiler



# Plugin





# Plugin: Example

```
DemoPlugin>>transform
transform
  | sends |
sends := ast sendNodes.
sends := sends select: [ :each | each selector = #ifTrue: ].
sends do: [:each | each replaceWith:
  (RBLiteralNode value: true)].
^ast
```

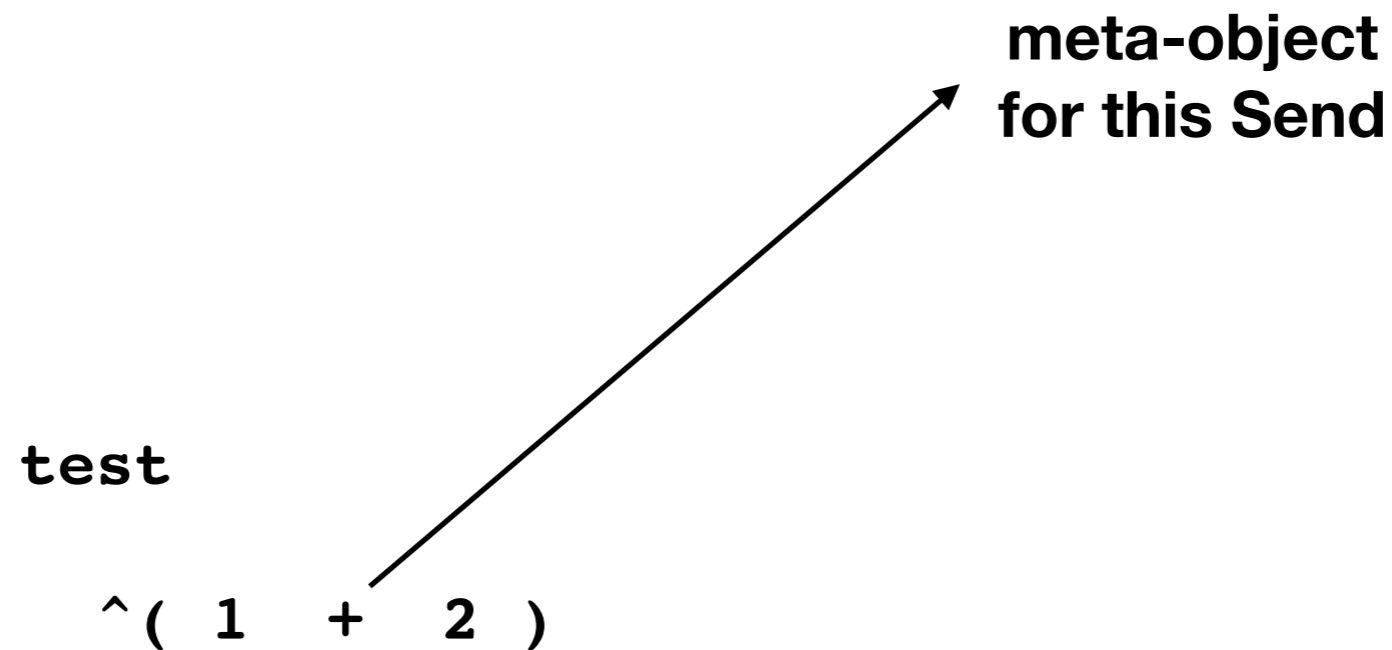
- We get all ifTrue: sends
- replace them with true

# Back to the topic...

- A more fine-grained reflective mechanism seems to be missing
- Can't we do something with the AST?

# Wouldn't it be nice..

- With the AST, wouldn't it be nice if we could use this structure for Behavioural Reflection?
- If we could somehow attach a “arrow to the code” that points to a meta-object



# We have all pieces...

- We have the AST for each method
- It is quite simple
- We have a compiler in the system
- So this should be possible...

# The MetaLink

```
link := MetaLink new
      metaObject: Halt;
      selector: #once;
      control: #before.
```

- MetaLink points to metaObject
- Defines a selector to call
- And a control attribute: #before, #after, #instead
- Installed on a AST node:

```
(Number>>#sin) ast link: link
```

# The MetaLink

- Can be installed on any AST Node
- Methods will be re-compiled on the fly just before next execution
  - Link installation is very fast
- Changing a method removes all links from this method
  - Managing link re-installation has to be done by the user

# MetaLink: MetaObject

- MetaObject can be any object
- Even a Block: `[Transcript show 'hello']`
- Install on any Node with `#link:`
- de-install a link with `#uninstall`

# MetaLink: Selector

- MetaLink defines a message send to the MetaObject
- #selector defines which one
- Default is #value
- Yes, a selector with arguments is supported
  - We can pass information to the meta-object



# MetaLink: Argument

- The arguments define which arguments to pass
- We support a number of **reifications**

# Reifications

- Reifications define data to be passed as arguments
- Reify —> Make something into an object that is not one normally
- Example: “All arguments of this message”

# Reifications: examples

- All nodes: `#object #context #class #node #link`
- Sends: `#arguments #receiver #selector`
- Method: `#arguments #selector`
- Variable: `#value`

**They are defined as subclasses of class RReification**

# Reifications as MetaObject

- We support some special metaObjects:
  - `#node`      The AST Node we are installed on
  - `#object`      `self` at runtime
  - `#class`      The class the links is installed in

# MetaLink: Condition

- We can specify a condition for the MetaLink
- Link is active if the condition evaluates to true
- We can pass reifications as arguments

```
link := MetaLink new
  metaObject: Halt;
  selector: #once;
  condition: [:object | object == 5] arguments: #(object).

(Number>>#sin) ast link: link.
```

# MetaLink: control

- We can specify when to call the meta-object
- We support `#before`, `#after` and `#instead`
- The `instead` is very simple: last one wins

# Example: Log

- We want to just print something to the Transcript

```
link := MetaLink new
      metaObject: [Transcript show: 'Reached Here'].

(Number>>#sin) ast link: link
```

# Recursion Problem

- Before we see more examples: There is a problem
- Imagine we put a MetaLink on some method deep in the System (e.g `new`, `+`, `do:` ).
- Our Meta-Object might use exactly that method, too



**Endless Loop!!**



# Recursion Problem

- Solution: Meta-Level
- We encode the a level in the execution of the system
- Every Link Activation increases the level
- A meta-link is just active for one level. (e.g. 0)

```
link := MetaLink new
      metaObject: [ Object new ];
      level: 0.
```

```
(Behavior>>#new) ast link: link.
```

# Example: Log

- Better use #level: 0
- Nevertheless: be careful! If you add this to method called often it can be very slow.

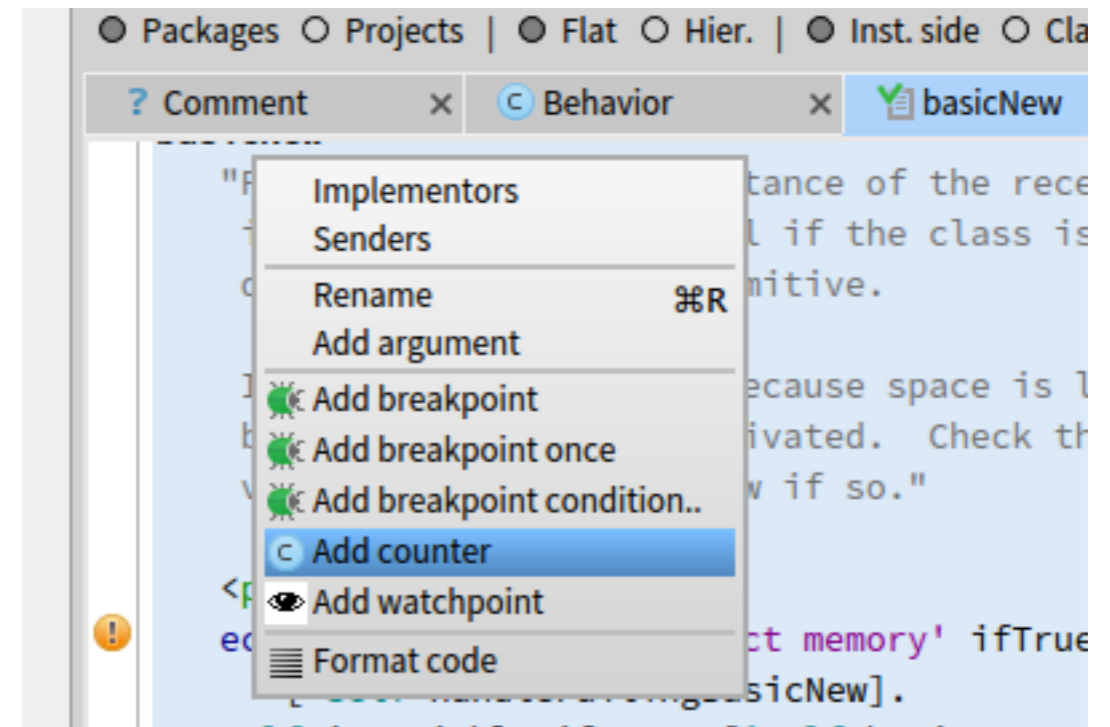
```
link := MetaLink new
  metaObject: [Transcript show: 'Reached Here'];
  level: 0.
```

# Example: Counter

- In the Browser you can add a “counter” to the AST
- See class `ExecutionCounter`

**install**

```
link := MetaLink new  
    metaObject: self;  
    selector: #increase.  
node link: link.
```



# Example: Breakpoint

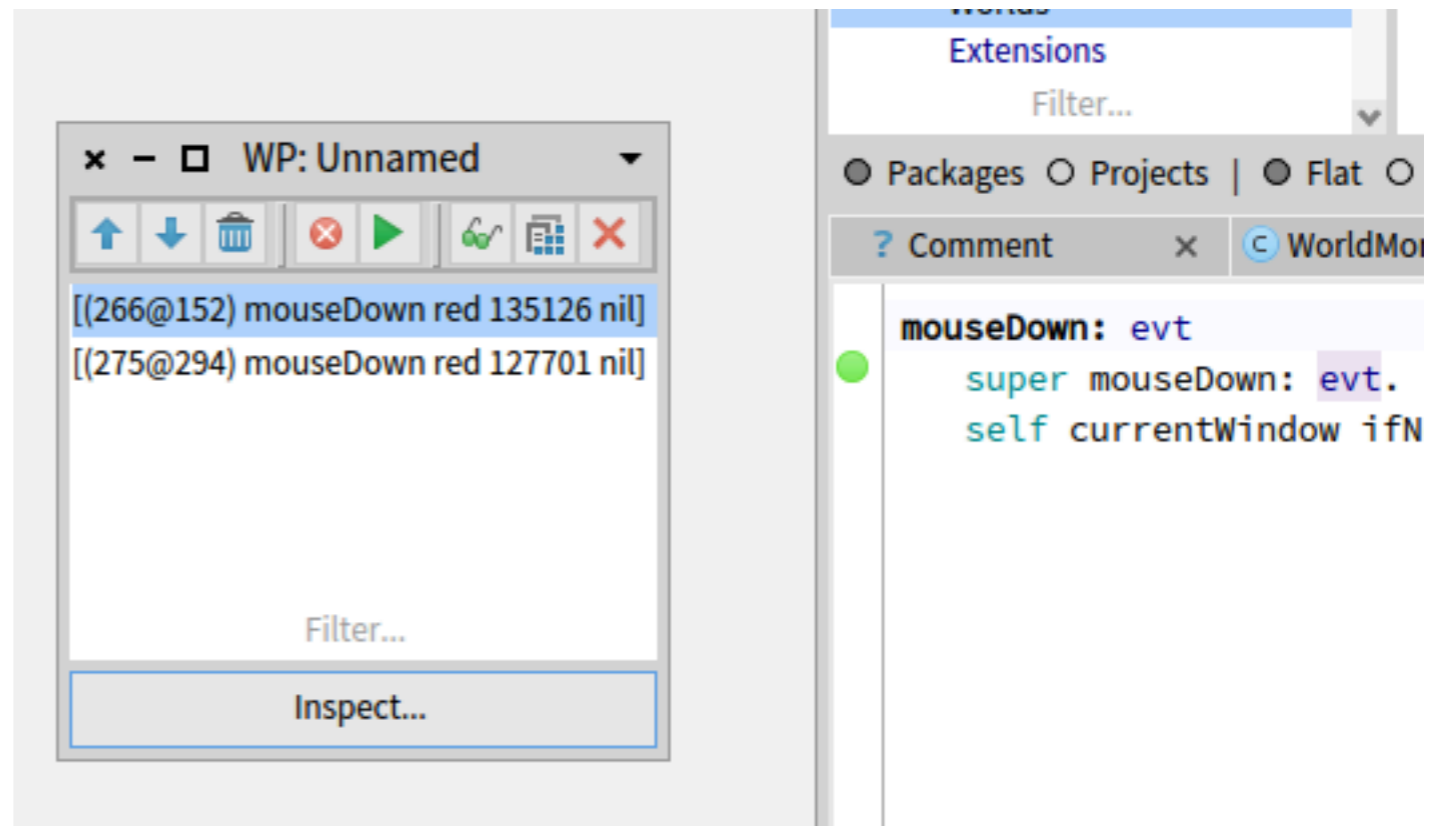
- “*Add Breakpoint*” in AST (Suggestions) Menu
- See class Breakpoint
- Break Once
- Conditional Break

```
breakLink
  ^ MetaLink new
    metaObject: Break;
    selector: #break;
    options: options
```

# Example: WatchPoint

- Watchpoint: Record Value at a point in the AST
- Example: Watch event in WorldMorph>>#mouseDown:

**Click on background  
-> value recorded**



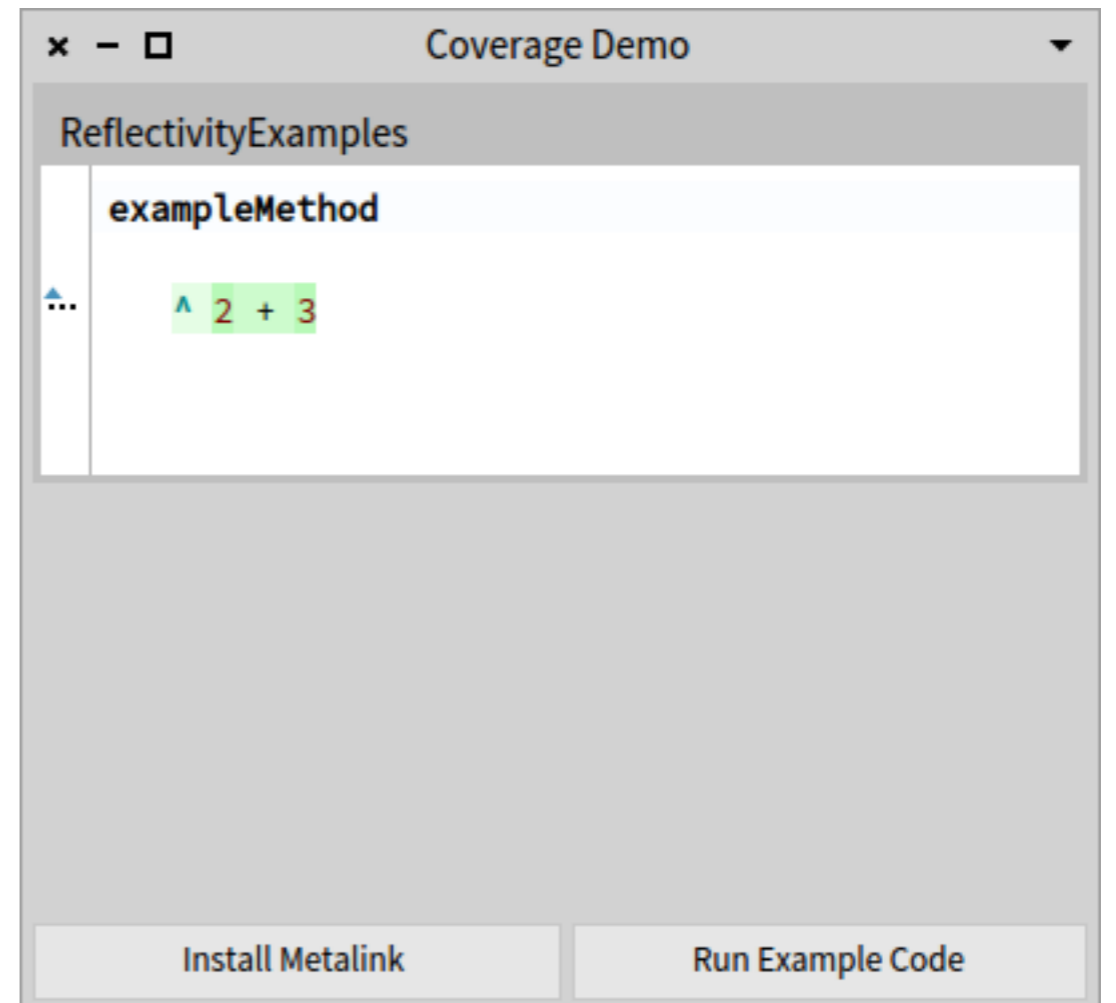
# Example: WatchPoint

- Implementation: class `Watchpoint`, method `install`
- example of a `#after` link with a condition

```
link := MetaLink new
      metaObject: self;
      selector: #addValue;;
      arguments: #(value);
      control: #after;
      condition: [ recording ].
```

# Example: Code Coverage

- Small Demo.
- Start with CoverageDemo new open



# Example: Code Coverage

- Example of a MetaLink with a #node MetaObject
- Meta-Object is the node that the link is installed on

```
link := MetaLink new  
    metaObject: #node;  
    selector: #tagExecuted.
```



# Interesting Properties

- Cross Cutting
  - One Link can be installed multiple times
  - Over multiple methods and even Classes
  - And across operations (e.g., Send and Assignment) as long as all reifications requested are compatible
- Fully Dynamic: Links can be added and removed at runtime
- Even by the meta-object of another meta-link!

# Example: Accept for Test

- Imagine we want to edit a method that is called often by the System.
- How do we test it?
- It would be nice if we could “Accept for Test”

# Example: Accept for Test

- Menu in the browser: AST menu shows for all nodes.

```
SycSourceCodeCommand subclass: #SycAcceptForTest
  instanceVariableNames: 'source'
  classVariableNames: ''
  package: 'SystemCommands-SourceCodeCommands'
```

```
defaultMenuItemName
  ^'Accept for Test'
```

```
readParametersFromContext: aSourceCodeContext
  super readParametersFromContext: aSourceCodeContext.
  source := aSourceCodeContext tool pendingText
```

- We implement our code in the #execute method

# Example: Accept for Test

- How we know that we are in a test?

```
CurrentExecutionEnvironment value isTest
```

- We can compile the current text buffer

```
newMethod := method methodClass compiler  
  source: source;  
  options: #(+ optionParseErrors);  
  compile.
```

# Example: Accept for Test

- Add this code to the beginning of the method:

```
[ :aContext :args |  
    CurrentExecutionEnvironment value isTest ifTrue: [  
  
        aContext return: (newMethod  
            valueWithReceiver: aContext  
            receiver  
            arguments: args) ] ]
```

- Let's do that with a MetaLink!

# Example: Accept for Test

```
execute
```

```
| newMethod metaLink |
```

```
newMethod := method methodClass compiler  
    source: source;  
    options: #( + optionParseErrors);  
    compile.
```

```
"the link executes the method we just created and returns"
```

```
metaLink := MetaLink new  
    metaObject: [ :aContext :args |  
        CurrentExecutionEnvironment value isTest  
            ifTrue: [ aContext return: (newMethod  
                valueWithReceiver: aContext receiver  
                arguments: args) ] ];  
    selector: #value:value;;  
    arguments: #(context arguments).
```

```
self method ast link: metaLink
```

# What did we see?

- ASTs and AST Visitors
- Compiler and Compiler Plugins
- MetaLinks
- Recursion Problem
- Examples: Counter, Breakpoint, Coverage
- Accept for Test

# Limitations

- #instead needs more work (e.g to support conditions)
- Keep in mind: next metaLink taken into account for next method activation
  - Take care with long running loops!



# Help Wanted

- We are always interested in improvements!
- Pharo 12 development started, with lots of work on the Compiler
- Pull Requests Welcome!

**Questions?**