Advanced Reflection: MetaLinks

Marcus Denker, Inria

http://marcusdenker.de

Lecture at VUB Brussels, March 26, 2026

What we know

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

Reflection in Smalltalk

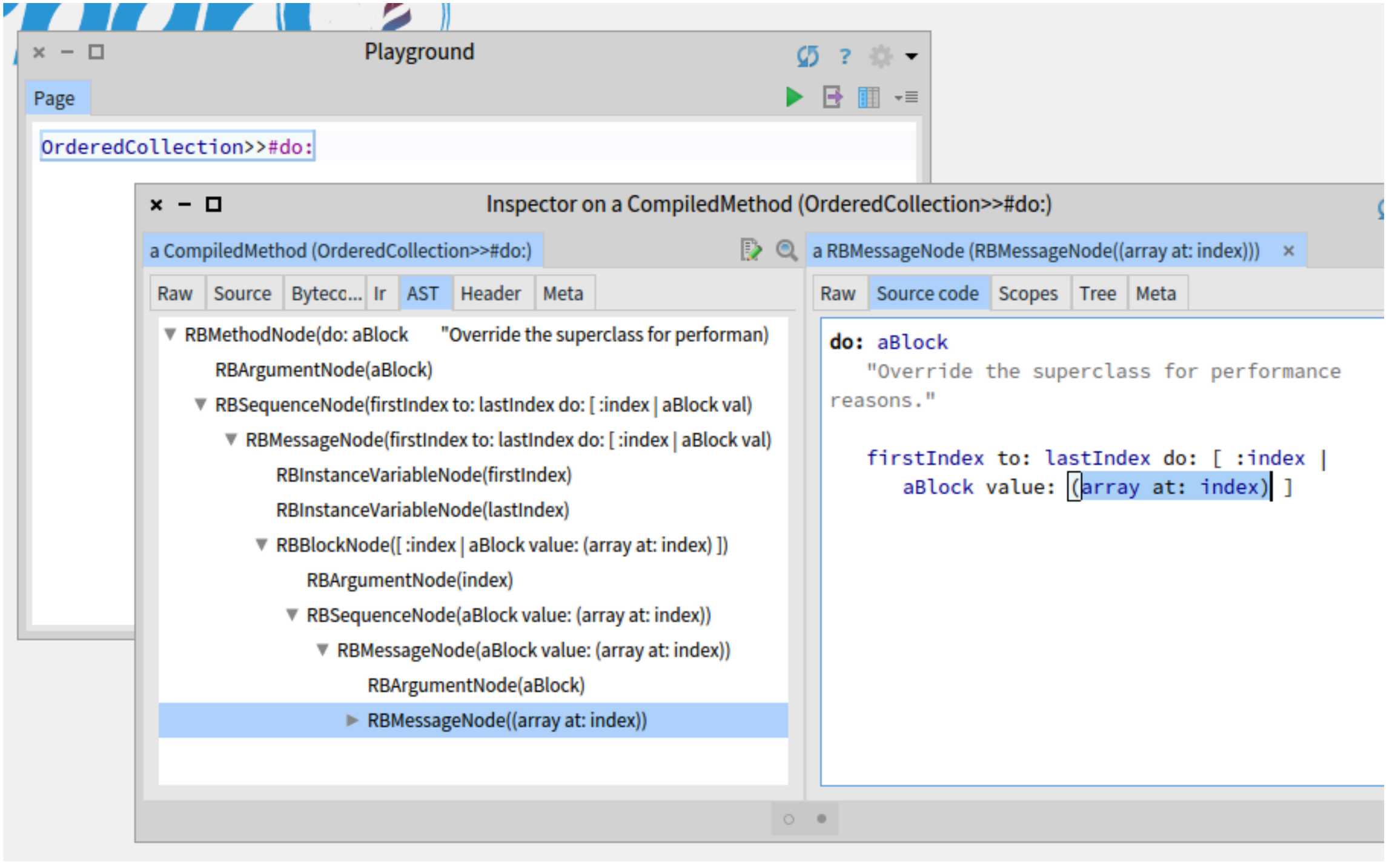
- Reflection is based on the Metaclass model, thus it is inherently structural
- Behavioral Reflection limited to:
  - Method lookup on failure (#doesNotUndersand:)
  - Reified stack (thisContext)

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



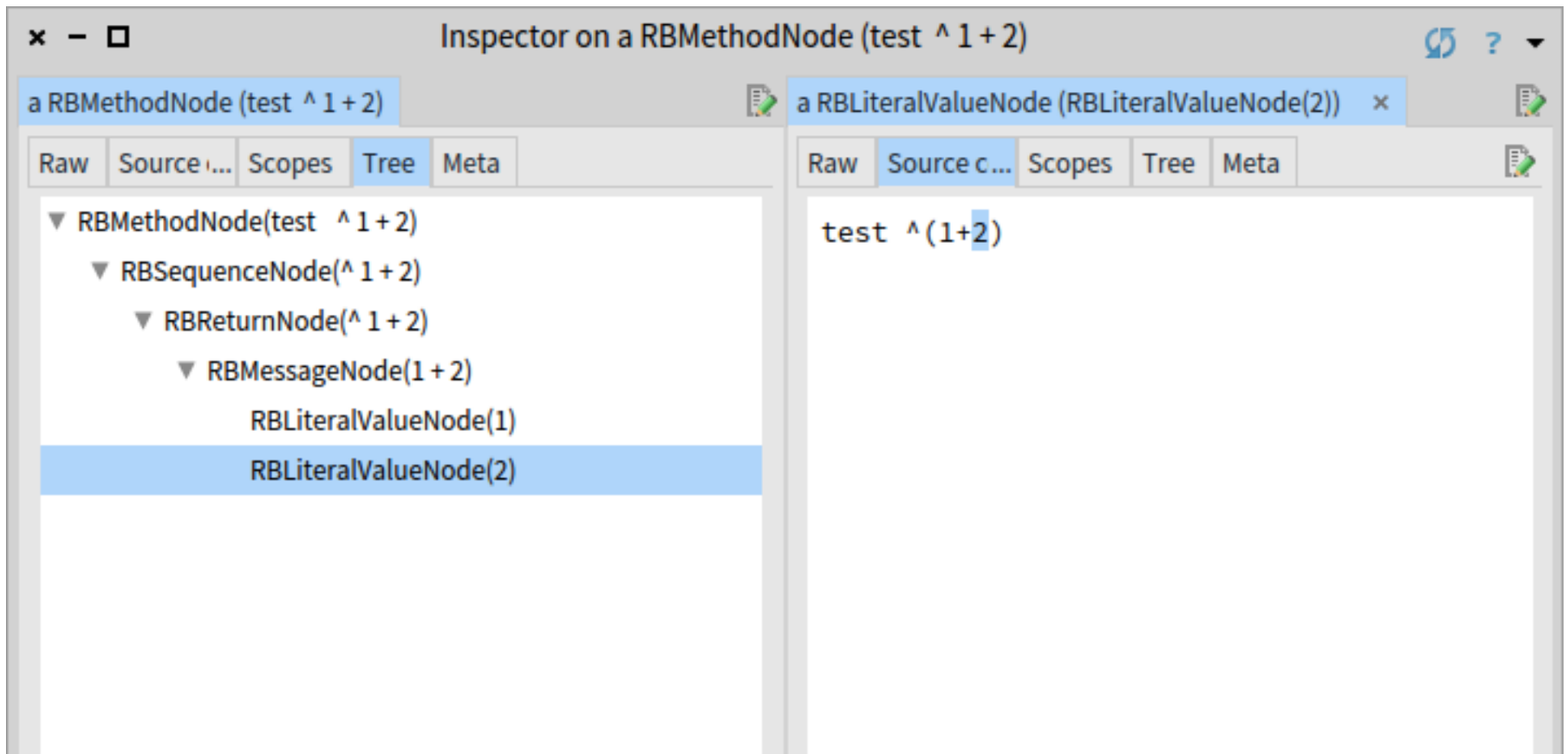
The AST

- 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,…)

Inspect a simple AST

- A very simple Example

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



AST

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

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

```
RBProgramNodeVisitor << #MyVisitor
    slots: { #counter };
    tag: 'Visitors';
    package: 'AST-Core'

visitMessageNode: aNode
    counter := counter + 1.
    super visitMessageNode: aNode.

MyVisitor new visit: (OrderedCollection>>#do:) ast
```

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,…)

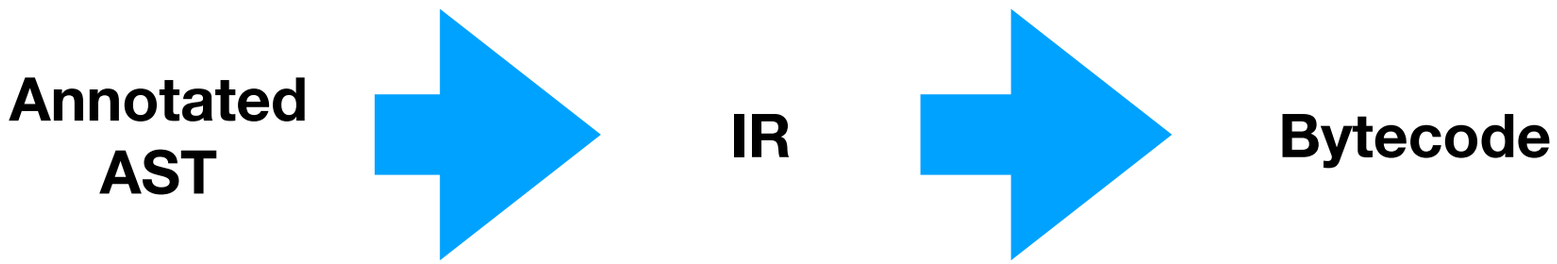
```
Smalltalk compiler parse: 'test ^(1+2)'
```

The Compiler

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

The Compiler

`RBParser` `OCSemanticAnalyzer`



`OCASTTranslator/ IRBuilder` `IRBytecodeGenerator`

AST Integration

- Originally just internal to the compiler
- Pharo:
  - used by all tools
  - send #ast to a method to get the AST
  - Cached for speed

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

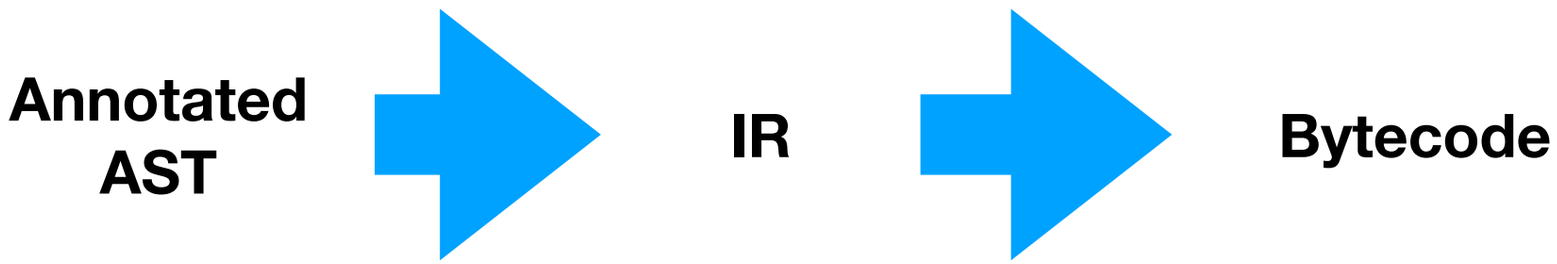
Compiler Plugins

- 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
```

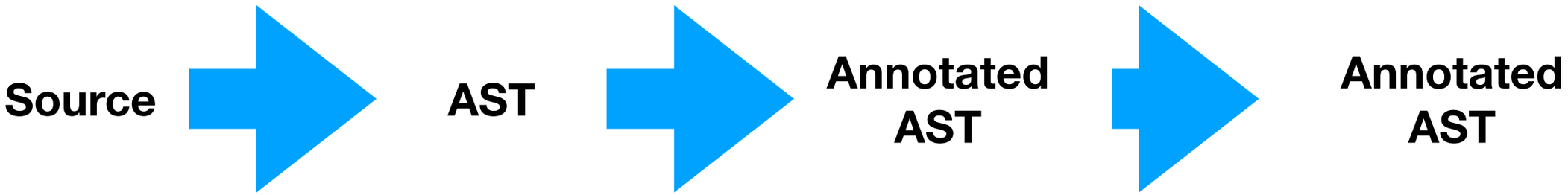
The Compiler

`RBParser` `OCSemanticAnalyzer`

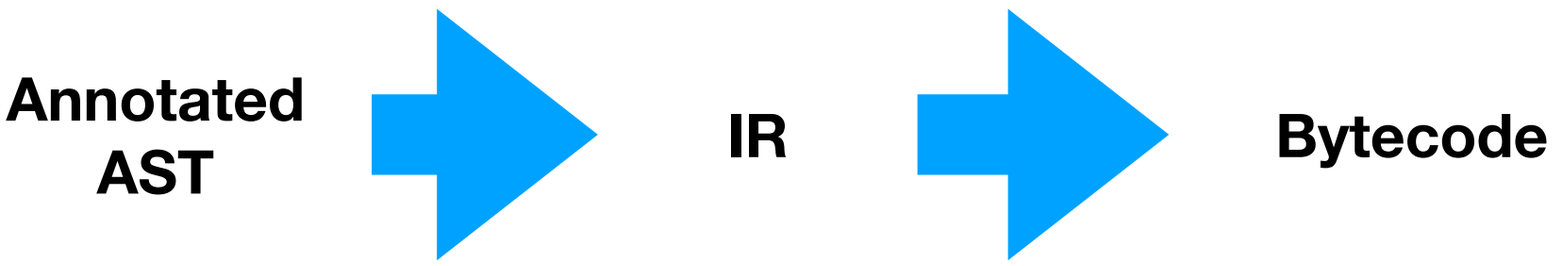


`OCASTTranslator/ IRBuilder` `IRBytecodeGenerator`

Plugin



`RBParser` `OCSemanticAnalyzer` `OCCompilerASTPlugin`



`OCASTTranslator/ IRBuilder`

`IRBytecodeGenerator`

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`

Plugins: Summary

- Compiler Plugins allow for simple experiments
- When doing bigger changes: Think about subclassing the compiler
    - override #compiler on the class side to enable for your code

Back to the topic…

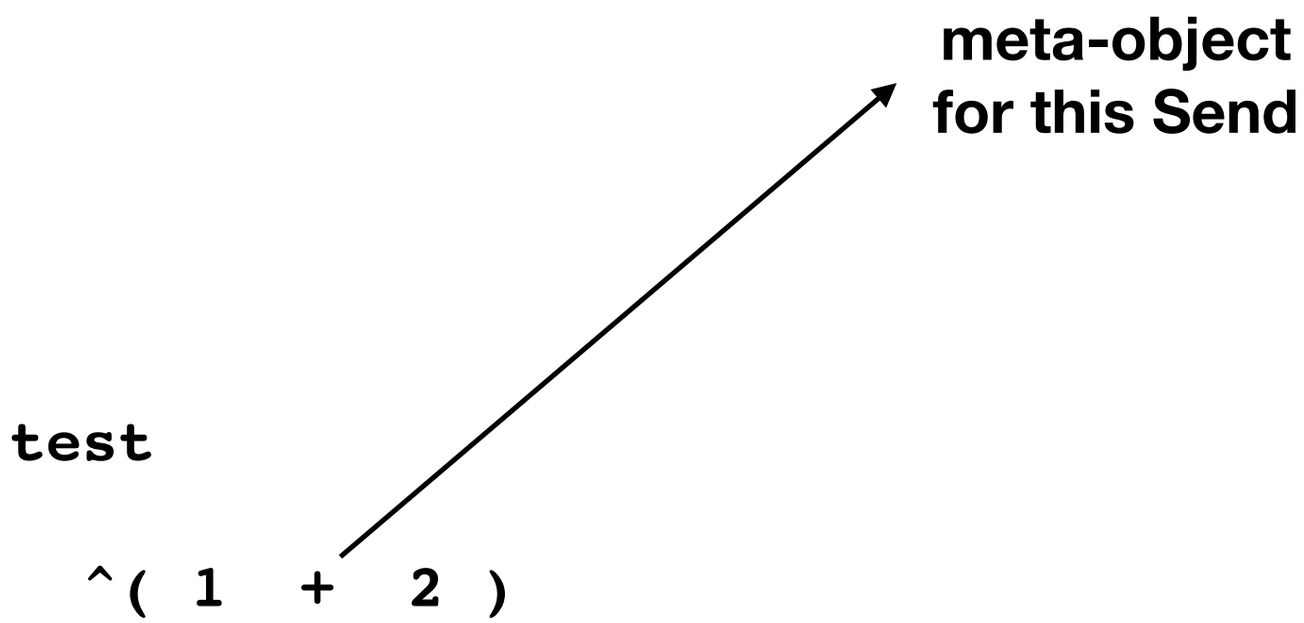
- A more fine-grained reflective mechanism seems to be missing

- 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 Behavioral Reflection?

- If we could somehow attach a “arrow to the code” that points to a meta-object

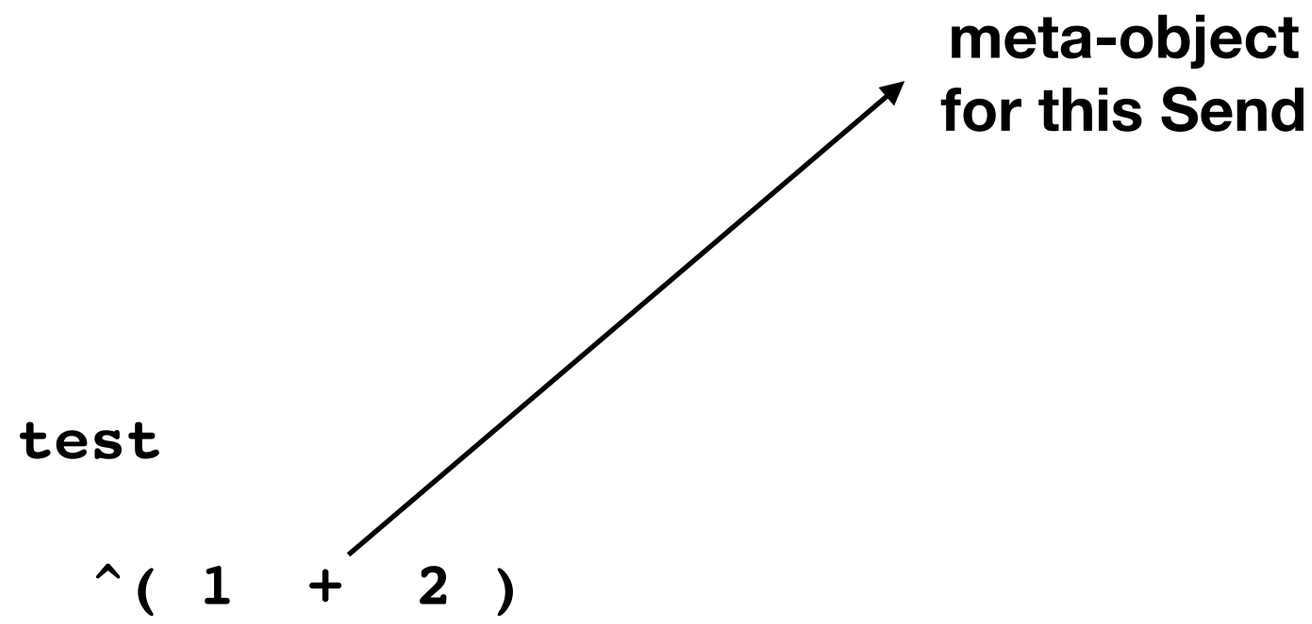


We have all pieces…

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

The arrow as an object?

- Can we turn the arrow into an object?



The MetaLink

```
link := MetaLink new
      metaObject: <someObjectHere>;
      selector: #aSelector;
      control: #before.
```

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

The MetaLink: Example

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

- Metaobject: Halt
- selector: #once
- Installed on a AST node:

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

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 —> Turn something into an object that is normally not one
- 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 RFReification

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

The MetaLink: Summary

```
link := MetaLink new
      metaObject: <someObjectHere>;
      selector: #aSelector;
      control: #before.
```

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

Example: Reify state access

- state (instance var) read is a bytecode
  - There is no way to “intercept” it easily
- Imagine the class would define the implementation
  - subclasses could re-define implementation
- “Metaobject Protocol” (CLOS has something similar)

Example: Reify Read

- What do we need?
  - name
  - object

```
link := MetaLink new
      metaObject: #class;
      selector: #read:in:;
      arguments: #(name object)
```

```
read: aVarName in: object
    ^object instVarNamed: aVarName
```

```
(Demo>>#a) ast variableNodes first link: link.
```

Can be done for write, send…

- What do we need?
  - receiver
  - selector
  - arguments

```
link := MetaLink new
        metaObject: #class;
selector: #send:to:args;
arguments: #(selector object arguments)
```

- But take care of recursion! (see later)

Very useful for Tools

- Meta-Object style examples are fun, but not practical
  - e.g. too slow
- Tools can use MetaLinks tailored to what they need
  - instead of reifying everything

Example: Log

- We want to just print something to the Transcript
- We could use the Transcript itself as a meta-object
- Ot just use the “block as metaobject” feature:

```
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: Code Coverage

- We can add the node itself as Metaobject
- Tag the node as being executed

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


tagExecuted
    ^self propertyAt: #tagExecuted put: true
```

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.
```

Example: Breakpoint

- We already had an example using Halt as the metaObject
- Here: halt once

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

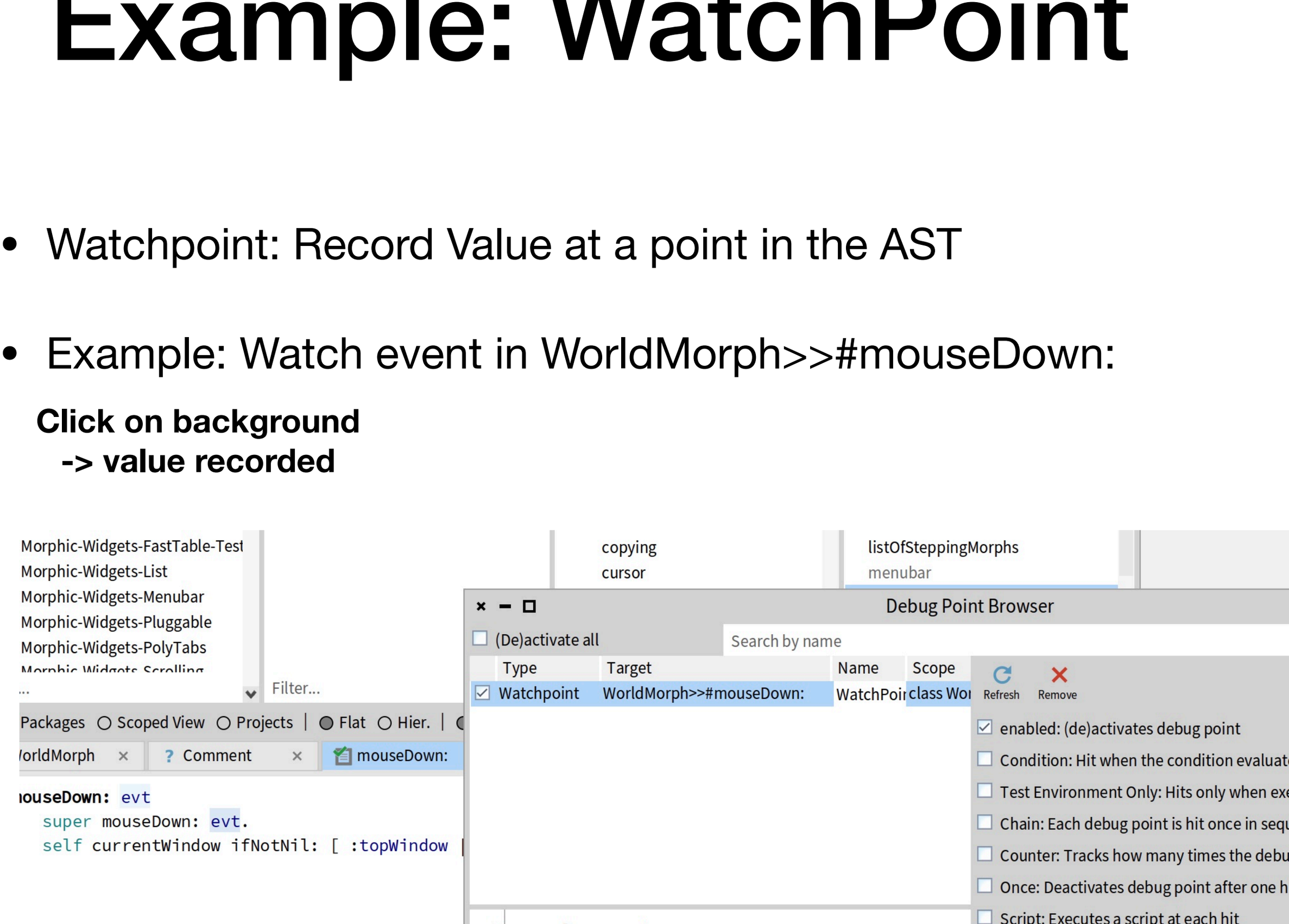
Pharo 12: DebugPoints

- DebugPoints are a generalized Breakpoints
- DebugPoints allow for composable behavior
  - Break
  - Watch
  - Conditions…

Example: WatchPoint

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

Click on background
-> value recorded



DebugPoint: MetaLink

- see DebugPoint>>#metaLink
- passes the execution context (the stack)
- Code:

```
metaLink
    ^(MetaLink new
        metaObject: self;
        options: #(+ optionCompileOnLinkInstallation);
        selector: #hitWithContext:;
        arguments: #(context) ).
```

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!

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!
- You have to manage installing links
    - Especially if code changes

Reflectivity NG

- It is time to step back
- What is good? What not?
- What would a “Future Reflectivity” Model and Framework look like?

Good Points

- High level, sub-method model
- Installation does not trigger immediate recompilation
    - Very fast to install lots of links
- Cross-Cutting
- Reifications
    - Partial: Where, When and what

Things to Improve

- AST: Not always persistent
  - But if, it takes memory
- Installation hard to control
  - Can we have Transaction semantics?
- Recursion Control is very slow
  - VM support?

Beyond AST

- Imagine Instance Variables
  - To “put a link” on a Variable: annotate all read/write AST nodes
  - We have helpers for that
- Idea: MetaLinks on structure outside of AST
  - First experiments: Metalinks on Variables

Reflectivity NG

- Experiment with real persistent AST
  - See Soil Project
- Transaction semantics for Link Install
- Better support to re-install links
- Beyond AST: MetaLinks on all structures
  - e.g. Variables

What did we see?

- ASTs and AST Visitors
- Compiler and Compiler Plugins
- MetaLinks
- Recursion Problem
- Examples: Log, Coverage, DebugPoint
- Some Future ideas

Questions?

- ASTs and AST Visitors
- Compiler and Compiler Plugins
- MetaLinks
- Recursion Problem
- Examples: Log, Coverage, DebugPoint
- Some Future ideas