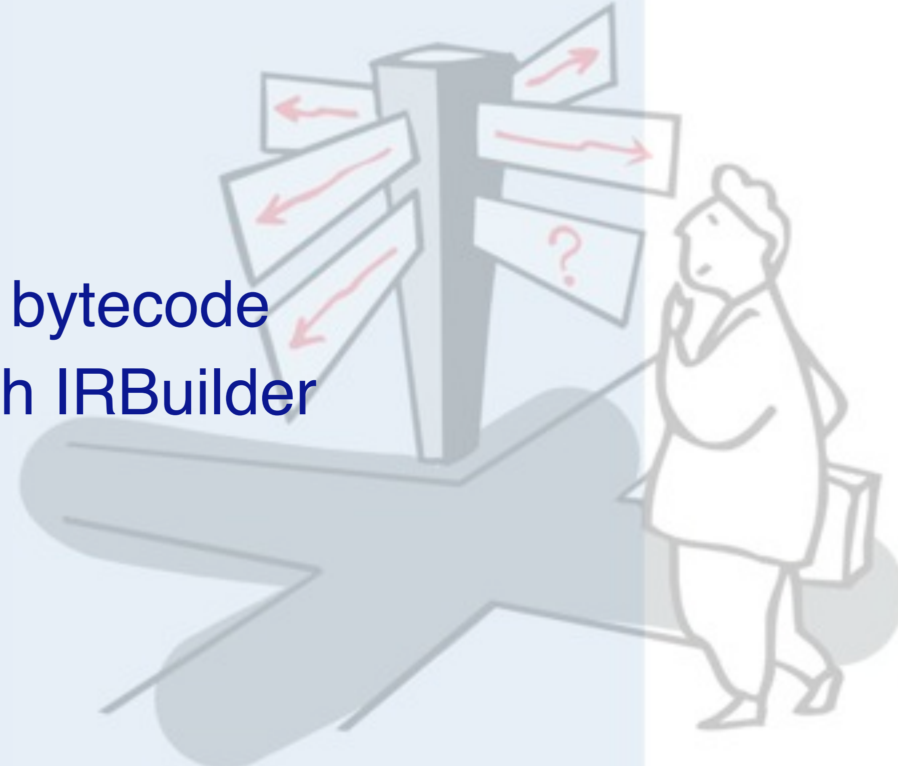
Opal Compiler

Jorge Ressia

Roadmap

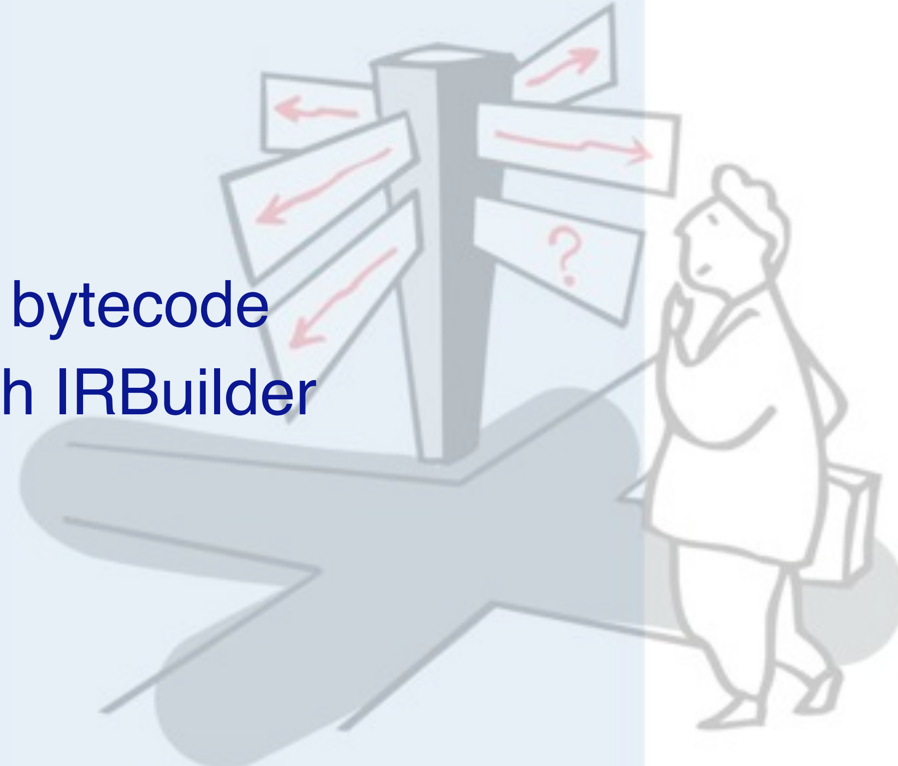
- The Pharo compiler
- Introduction to Smalltalk bytecode
- Generating bytecode with IRBuilder
- ByteSurgeon



Original material by Marcus Denker

Roadmap

- **The Pharo compiler**
- Introduction to Smalltalk bytecode
- Generating bytecode with IRBuilder
- ByteSurgeon



The Pharo Compiler

- Default compiler
  - very old design
  - quite hard to understand
  - hard to modify and extend

What qualities are important in a compiler?

> Correct code
> Output runs fast
> Compiler runs fast
> Compile time proportional to program size
> Support for separate compilation
> Good diagnostics for syntax errors
> Works well with the debugger
> Good diagnostics for flow anomalies
> Consistent, predictable optimization

Why do we care?

- ByteSurgeon — Runtime Bytecode Transformation for Smalltalk
- ChangeBoxes — Modeling Change as a first-class entity
- Reflectivity — Persephone, Geppetto and the rest
- Helvetia — Context Specific Languages with Homogeneous Tool Integration
- Albedo — A unified approach to reflection.

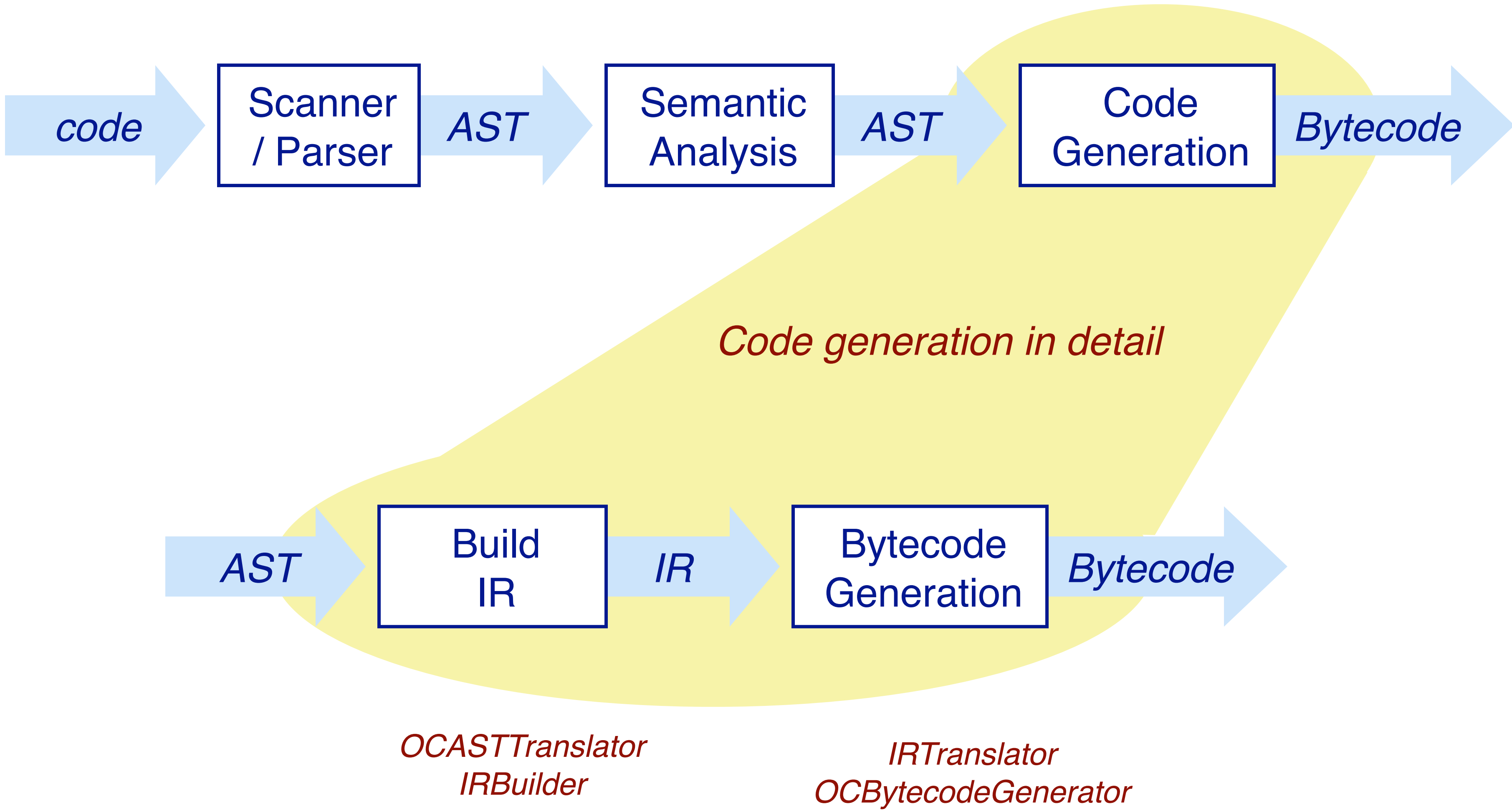
Opal Compiler

> Opal Compiler for Pharo
  - http://scg.unibe.ch/research/OpalCompiler

Opal Compiler

> Fully reified compilation process:
  - Scanner/Parser (RBParser)
    - *builds AST (from Refactoring Browser)*
  - Semantic Analysis: OCASTSemanticAnalyzer
    - *annotates the AST (e.g., var bindings)*
  - Translation to IR: OCASTTranslator
    - *uses IRBuilder to build IR (Intermediate Representation)*
  - Bytecode generation: IRTranslator
    - *uses OCBytecodeGenerator to emit bytecodes*

Compiler: Overview



Compiler: Design Decisions

- Every building block of the compiler is implemented as a visitor on the representation.
- The AST is never changed

Compiler: AST

- AST: Abstract Syntax Tree
  - Encodes the Syntax as a Tree
  - No semantics yet!
  - Uses the RB Tree:
    - *Visitors*
    - *Transformation (replace/add/delete)*
    - *Pattern-directed TreeRewriter*
    - *PrettyPrinter*

```
RBProgramNode
   RBDoItNode
   RBMethodNode
   RBReturnNode
   RBSequenceNode
   RBValueNode
      RBArrayNode
      RBAssignmentNode
      RBBlockNode
      RBCascadeNode
      RBLiteralNode
      RBMessageNode
      RBOptimizedNode
      RBVariableNode
```

Compiler: Syntax

> Before: SmaCC: Smalltalk Compiler Compiler
  - Similar to Lex/Yacc
  - SmaCC can build LARL(1) or LR(1) parser

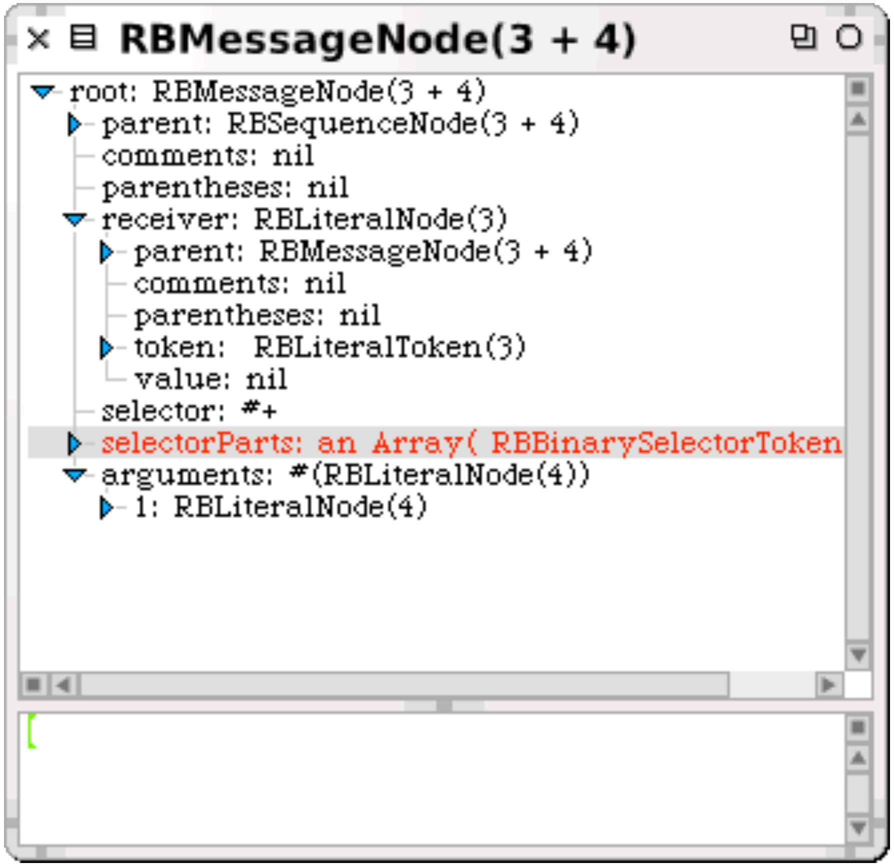
> Now: RBParser

> Future: PetitParser

A Simple Tree

```
RBParser parseExpression: '3+4'
```

NB: explore it



A Simple Visitor

```
RBProgramNodeVisitor new visitNode: tree
```

Does nothing except walk through the tree

TestVisitor

```
RBProgramNodeVisitor subclass: #TestVisitor
   instanceVariableNames: 'literals'
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Compiler-AST-Visitors'

TestVisitor>>acceptLiteralNode: aLiteralNode
   literals add: aLiteralNode value.

TestVisitor>>initialize
   literals := Set new.

TestVisitor>>literals
   ^literals
```

```
tree := RBParser parseExpression: '3 + 4'.
(TestVisitor new visitNode: tree) literals
```

```
a Set(3 4)
```

Compiler: Semantics

- We need to analyze the AST
  - Names need to be linked to the variables according to the scoping rules

- OCASTSemanticAnalyzer implemented as a Visitor
  - Subclass of RBProgramNodeVisitor
  - Visits the nodes
  - Grows and shrinks scope chain
  - Methods/Blocks are linked with the scope
  - Variable definitions and references are linked with objects describing the variables

Scope Analysis

```
testBlockTemp
  | block block1 block2 |
  block := [ :arg | [ arg ] ].
  block1 := block value: 1.
  block2 := block value: 2.
```

Scope Analysis

```
testBlockTemp
  | block block1 block2 |
  block := [ :arg | [ arg ] ].
  block1 := block value: 1.
  block2 := block value: 2.
```

```
OCClassScope
  OCInstanceScope
    OCMethodScope 2
      OCBlockScope 3
        OCBlockScope 4
```

Compiler: Semantics

- > OCASTClosureAnalyzer
  - — Eliot's Closure analysis: copying vs. tempvector

Closures

```
counterBlock
        | count |
        count := 0.
        ^[ count := count + 1].
```

Closures

- Break the dependency between the block activation and its enclosing contexts for accessing locals

Contexts

```
inject: thisValue into: binaryBlock
  | nextValue |
  nextValue := thisValue.
  self
    do: [:each |
          nextValue := binaryBlock
                   value: nextValue value: each].
  ^nextValue
```

Contexts

```
inject: thisValue into: binaryBlock
 | indirectTemps |
  indirectTemps := Array new: 1.
  indirectTemps at: 1 put: thisValue.
" was nextValue := thisValue."
  self do:
   [:each |
        indirectTemps
             at: 1
             put: (binaryBlock
          value: (indirectTemps at: 1)
                      value: each)].
 ^indirectTemps at: 1
```

Contexts

```
inject: thisValue into: binaryBlock
 | indirectTemps |
   indirectTemps := Array new: 1.
   indirectTemps at: 1 put: thisValue.
   self do: (thisContext
                 closureCopy:
                      [:each |
             binaryBlockCopy indirectTempsCopy |
                      indirectTempsCopy
                        at: 1
                        put: (binaryBlockCopy
                              value: (indirectTempsCopy at: 1)
                              value: each)]
                 copiedValues:
            (Array with: binaryBlock with: indirectTemps)).
 ^indirectTemps at: 1
```

Closures Analysis

```
| a |
a := 1.
[ a ]
```

Closures Analysis

```
| a |
a := 1.
[ a ]
```

a is copied

Closures Analysis

```
| index block collection |
index := 0.
block := [
  collection add: [ index ].
  index := index + 1 ].
[ index < 5 ] whileTrue: block.
```

Closures Analysis

```
| index block collection |
index := 0.
block := [
  collection add: [ index ].
  index := index + 1 ].
[ index < 5 ] whileTrue: block.
```

index is remote

Compiler: Intermediate Representation

> IR: Intermediate Representation
  - Semantic like Bytecode, but more abstract
  - Independent of the bytecode set
  - IR is a tree
  - IR nodes allow easy transformation
  - Decompilation to RB AST

> IR is built from AST using OCASTTranslator:
  - AST Visitor
  - Uses IRBuilder

Compiler: Intermediate Representation

```
IRBuilder new
    pushLiteral: 34;
    storeInstVar: 2;
    popTop;
    pushInstVar: 2;
    returnTop;
    ir.
```

```
17 <20> pushConstant: 34
18 <61> popIntoRcvr: 1
19 <01> pushRcvr: 1
20 <7C> returnTop
```

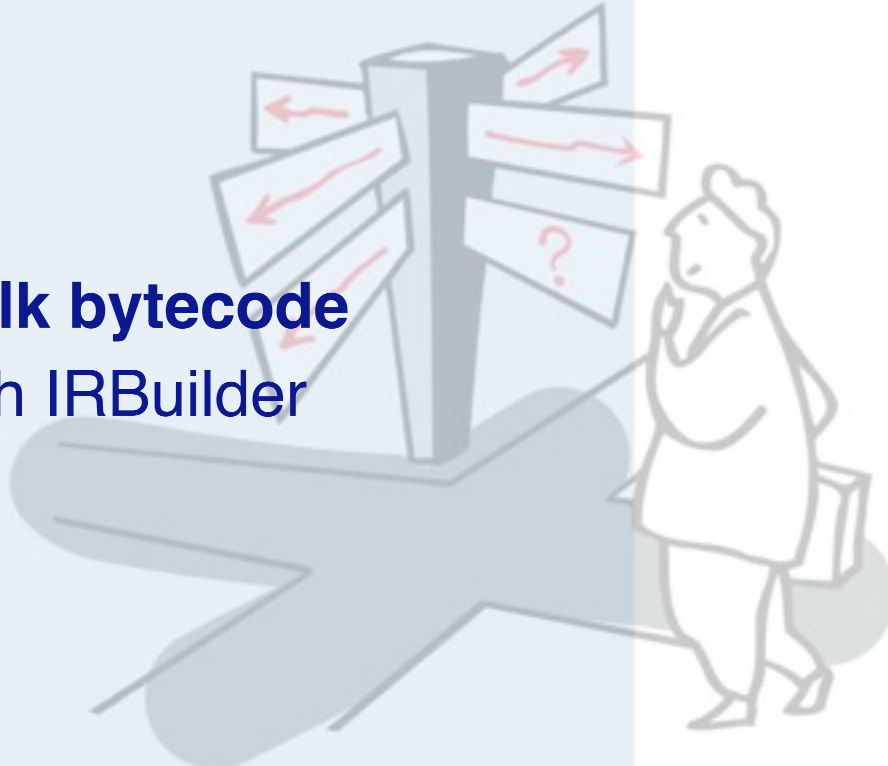
Compiler: Bytecode Generation

> IR needs to be converted to Bytecode
  - IRTranslator: Visitor for IR tree
  - Uses OCBytecodeGenerator to generate Bytecode
  - Builds a compiledMethod
  - Details to follow next section

```
testReturn1
   | iRMethod aCompiledMethod |
   iRMethod := IRBuilder new
      pushLiteral: 1;
      returnTop;
      ir.
```

```
aCompiledMethod := iRMethod compiledMethod.
self should:
   [(aCompiledMethod
      valueWithReceiver: nil
      arguments: #() ) = 1].
```

Roadmap



- The Pharo compiler
- **Introduction to Smalltalk bytecode**
- Generating bytecode with IRBuilder
- ByteSurgeon

Reasons for working with Bytecode

- Generating Bytecode
  - Implementing compilers for other languages
  - Experimentation with new language features
- Parsing and Interpretation:
  - Analysis (e.g., `self` and `super` sends)
  - Decompilation (for systems without source)
  - Printing of bytecode
  - Interpretation: `Debugger`, `Profiler`

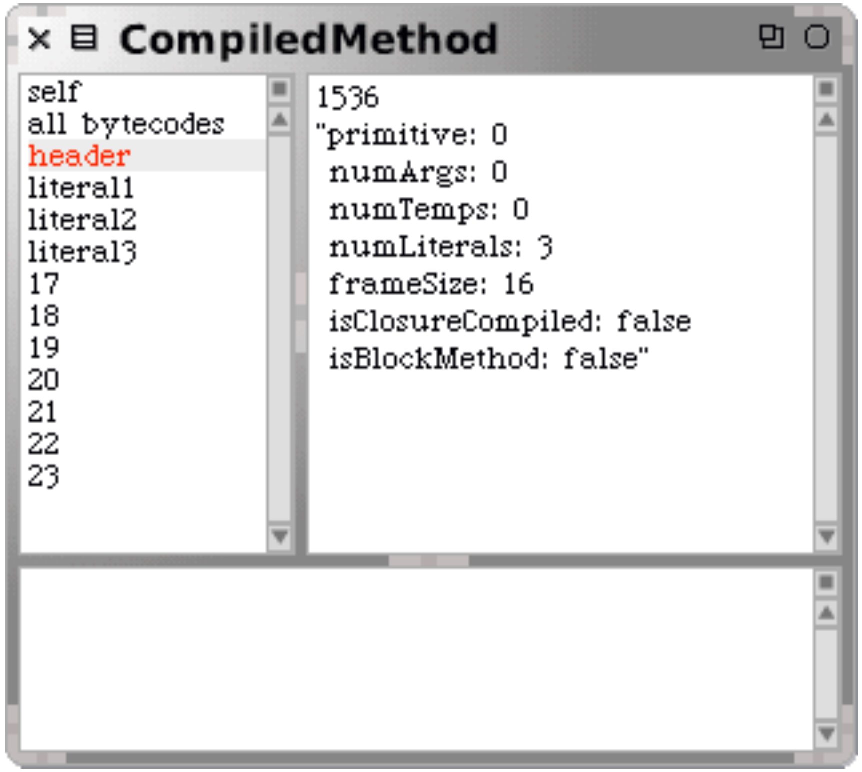
The Pharo Virtual Machine

- Virtual machine provides a virtual processor
  - Bytecode: The “machine-code” of the virtual machine
- Smalltalk (like Java): Stack machine
  - easy to implement interpreters for different processors
  - most hardware processors are register machines
- Squeak VM: Implemented in *Slang*
  - Slang: Subset of Smalltalk. (“C with Smalltalk Syntax”)
  - Translated to C

Bytecode in the CompiledMethod

> CompiledMethod format:

Header	Number of temps, literals...
Literals	Array of all Literal Objects
Bytecode	
Trailer	Pointer to Source




```
(Number>>#asInteger) inspect
```

```
(Number methodDict at: #asInteger) inspect
```

Bytecodes: Single or multibyte

> Different forms of bytecodes:

- Single bytecodes:
  - *Example: 120: push self*

- Groups of similar bytecodes
  - *16: push temp 1*
  - *17: push temp 2*
  - *up to 31*

Type	***Offset***
4 bits	4 bits

- Multibyte bytecodes
  - *Problem: 4 bit offset may be too small*
  - *Solution: Use the following byte as offset*
  - *Example: Jumps need to encode large jump offsets*

Example: Number>>asInteger

> Smalltalk code:

```
Number>>asInteger
   "Answer an Integer nearest
   the receiver toward zero."

   ^self truncated
```

> Symbolic Bytecode

```
9 <70> self
10 <D0> send: truncated
11 <7C> returnTop
```

Example: Step by Step

- `9 <70> self`
  - The receiver (self) is pushed on the stack
- `10 <D0> send: truncated`
  - Bytecode 208: send litereral selector 1
  - Get the selector from the first literal
  - start message lookup in the class of the object that is on top of the stack
  - result is pushed on the stack
- `11 <7C> returnTop`
  - return the object on top of the stack to the calling method

Pharo Bytecode

> 256 Bytecodes, four groups:

- Stack Bytecodes
  - *Stack manipulation: push / pop / dup*
- Send Bytecodes
  - *Invoke Methods*
- Return Bytecodes
  - *Return to caller*
- Jump Bytecodes
  - *Control flow inside a method*

Stack Bytecodes

- Push values on the stack
  - e.g., temps, instVars, literals
  - e.g: 16 - 31: push instance variable
- Push Constants
  - False/True/Nil/1/0/2/-1
- Push `self`, `thisContext`
- Duplicate top of stack
- Pop

Sends and Returns

> Sends: receiver is on top of stack
  - Normal send
  - Super Sends
  - Hard-coded sends for efficiency, e.g. `+`, `-`

> Returns
  - Return top of stack to the sender
  - Return from a block
  - Special bytecodes for return `self`, `nil`, `true`, `false` (for efficiency)

Jump Bytecodes

> Control Flow inside one method
  - Used to implement control-flow efficiently
  - Example:

```
^ 1<2 ifTrue: ['true']
```

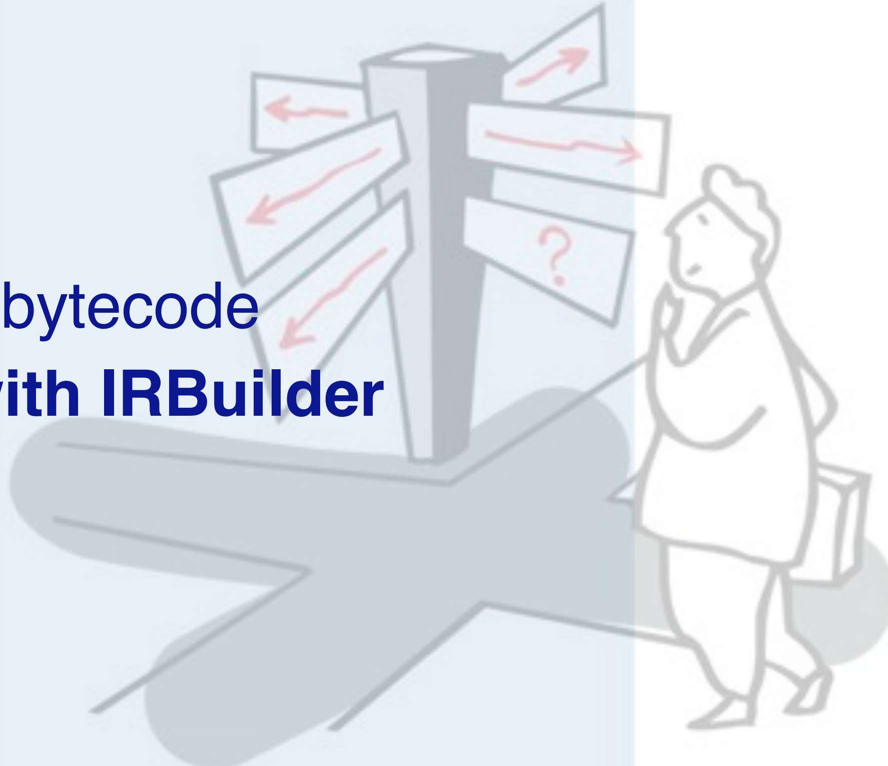
```
9 <76> pushConstant: 1
10 <77> pushConstant: 2
11 <B2> send: <
12 <99> jumpFalse: 15
13 <20> pushConstant: 'true'
14 <90> jumpTo: 16
15 <73> pushConstant: nil
16 <7C> returnTop
```

Closure Bytecode

- 138 Push (Array new: k)/Pop k into: (Array new: j)
- 140 Push Temp At k In Temp Vector At: j
- 141 Store Temp At k In Temp Vector At: j
- 142 Pop and Store Temp At k In Temp Vector At: j
- 143 Push Closure Num Copied l Num Args k BlockSize j

Roadmap

> The Pharo compiler
> Introduction to Smalltalk bytecode
> **Generating bytecode with IRBuilder**
> ByteSurgeon



Generating Bytecode

> IRBuilder: A tool for generating bytecode
  - Part of the OpalCompiler

> Like an Assembler for Pharo

IRBuilder: Simple Example

> *Number*>>asInteger

```
iRMethod := IRBuilder new
   pushReceiver;     "push self"
   send: #truncated;
   returnTop;
   ir.

aCompiledMethod := iRMethod compiledMethod.

aCompiledMethod valueWithReceiver:3.5
                 arguments: #()
```

3

IRBuilder: Stack Manipulation

> popTop
  - remove the top of stack

> pushDup
  - push top of stack on the stack

> pushLiteral:

> pushReceiver
  - push self

> pushThisContext

IRBuilder: Symbolic Jumps

- Jump targets are resolved:
- Example: `false ifTrue: ['true'] ifFalse: ['false']`

```
iRMethod := IRBuilder new
   pushLiteral: false;
   jumpAheadTo: #false if: false;
   pushLiteral: 'true';              "ifTrue: ['true']"
   jumpAheadTo: #end;
   jumpAheadTarget: #false;
   pushLiteral: 'false';             "ifFalse: ['false']"
   jumpAheadTarget: #end;
   returnTop;
   ir.
```

IRBuilder: Instance Variables

- Access by offset
- Read: pushInstVar:
  - receiver on top of stack
- Write: storeInstVar:
  - value on stack
- Example: set the first instance variable to 2

```
iRMethod := IRBuilder new
      pushLiteral: 2;
      storeInstVar: 1;
      pushReceiver;                    "self"
      returnTop;
      ir.

aCompiledMethod := iRMethod compiledMethod.
aCompiledMethod valueWithReceiver: 1@2 arguments: #()
```

```
2@2
```

IRBuilder: Temporary Variables

- Accessed by name
- Define with addTemp: / addTemps:
- Read with pushTemp:
- Write with storeTemp:
- Example:
  - set variables a and b, return value of a

```
iRMethod := IRBuilder new
       addTemps: #(a b);
       pushLiteral: 1;
       storeTemp: #a;
       pushLiteral: 2;
       storeTemp: #b;
       pushTemp: #a;
       returnTop;
       ir.
```

IRBuilder: Sends

- normal send

```
builder pushLiteral: 'hello'
builder send: #size;
```

- super send

```
…
builder send: #selector toSuperOf: aClass;
```

  - The second parameter specifies the class where the lookup starts.

IRBuilder: Example

```
OCInstanceVar>>emitStore: methodBuilder
   methodBuilder storeInstVar: index
```

IRBuilder: Example

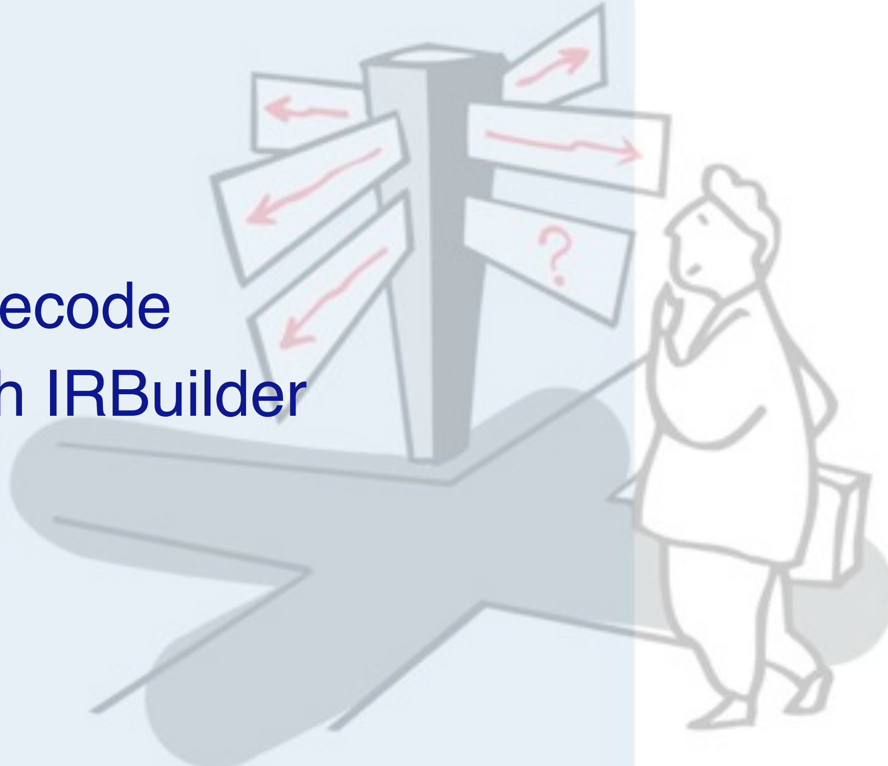
```
OCInstanceVar>>emitStore: methodBuilder
   methodBuilder
            pushReceiver;
            pushLiteral: index;
            send: #instVarAt
```

IRBuilder: Example

```
OCInstanceVar>>emitStore: methodBuilder
   methodBuilder
               pushReceiver;
               pushLiteral: index;
               send: #instVarAt:
```

This is global and we do not have much control

Roadmap



- The Pharo compiler
- Introduction to Pharo bytecode
- Generating bytecode with IRBuilder
- **ByteSurgeon**

ByteSurgeon

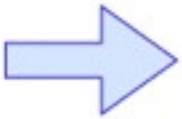
- Library for bytecode transformation in Smalltalk
- Full flexibility of Smalltalk Runtime
- Provides high-level API
- For Pharo, but portable

- Runtime transformation needed for
  - Adaptation of running systems
  - Tracing / debugging
  - New language features (MOP, AOP)

Example: Logging

- Goal: logging message send.
- First way: Just edit the text:

```
example
   self test.
```



```
example
   Transcript show: 'sending #test'.
   self test.
```

Logging with ByteSurgeon

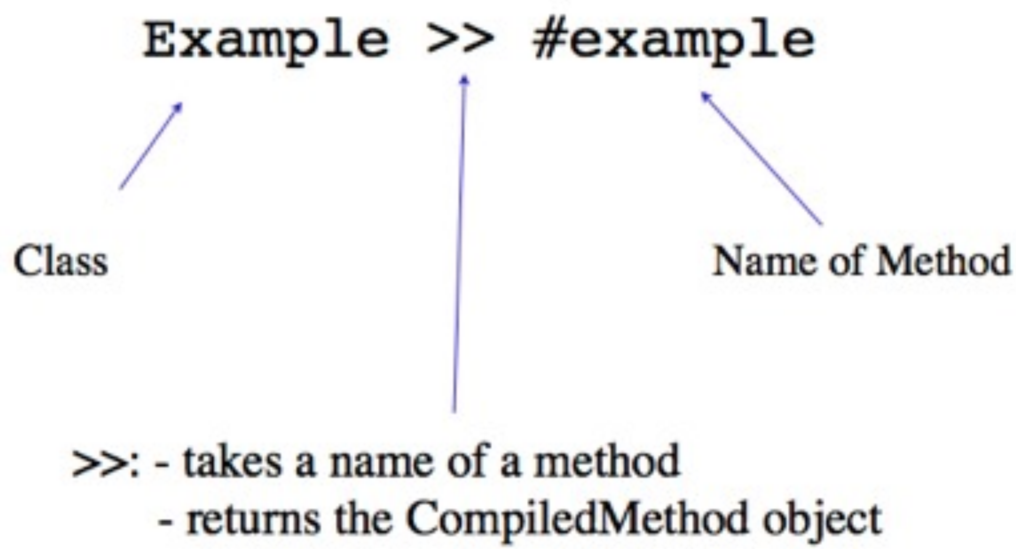
> Goal: Change the method without changing program text

> Example:

```
(Example>>#example)instrumentSend: [:send |
   send insertBefore:
      'Transcript show: ''sending #test'' '.
]
```

Logging: Step by Step

```
(Example>>#example)instrumentSend: [:send |
   send insertBefore:
      'Transcript show: ''sending #test'' '.
]
```



Logging: Step by Step

```
(Example>>#example)instrumentSend: [:send |
   send insertBefore:
      'Transcript show: ''sending #test'' '.
]
```

- instrumentSend:
  - takes a block as an argument
  - evaluates it for all send bytecodes

Logging: Step by Step

```
(Example>>#example)instrumentSend: [:send |
   send insertBefore:
      'Transcript show: ''sending #test'' '.
]
```

- The block has one parameter: send
- It is executed for each send bytecode in the method

Logging: Step by Step

```
(Example>>#example)instrumentSend: [:send |
   send insertBefore:
      'Transcript show: ''sending #test'' '.
]
```

> Objects describing bytecode understand how to insert code
  - insertBefor
  - insertAfter
  - replace

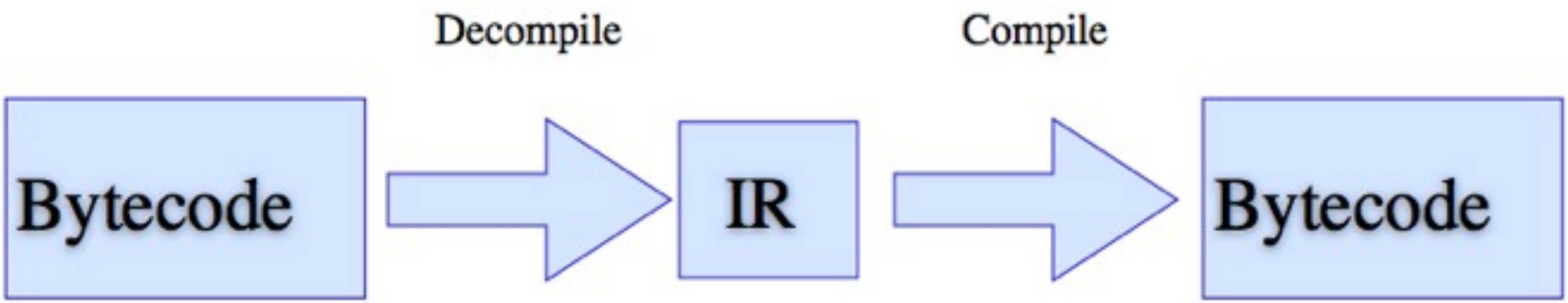
Logging: Step by Step

```
(Example>>#example)instrumentSend: [:send |
   send insertBefore:
      'Transcript show: ''sending #test'' '.
]
```

> The code to be inserted.
> Double quoting for string inside string
  - *Transcript show: 'sending #test'*

Inside ByteSurgeon

- > Uses IRBuilder internally



- > Transformation (Code inlining) done on IR

ByteSurgeon Usage

> On Methods or Classes:

```
MyClass instrument: [.... ].
(MyClass>>#myMethod) instrument: [.... ].
```

> Different instrument methods:

- instrument:
- instrumentSend:
- instrumentTempVarRead:
- instrumentTempVarStore:
- instrumentTempVarAccess:
- same for InstVar

Advanced ByteSurgeon

> Goal: extend a send with after logging

```
example
   self test.
```

```
example
      self test.
      Logger logSendTo: self.
```

Advanced ByteSurgeon

> With ByteSurgeon, something like:

```
(Example>>#example)instrumentSend: [:send |
   send insertAfter:
      'Logger logSendTo: ?' .
]
```

> How can we access the receiver of the send?
> Solution: Metavariable

Advanced ByteSurgeon

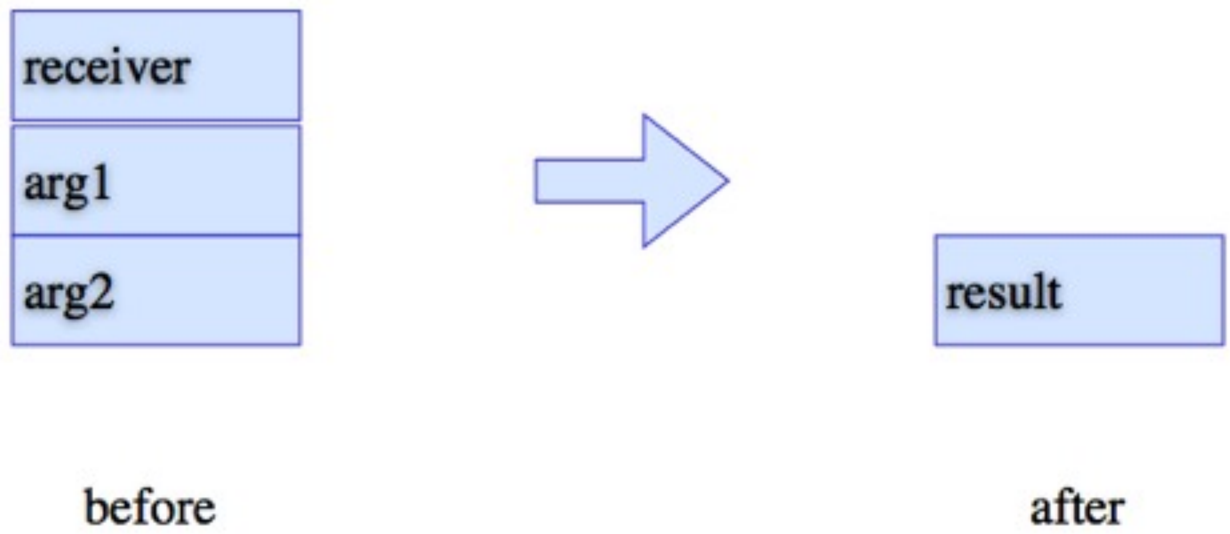
- With Bytesurgeon, something like:

```
(Example>>#example)instrumentSend: [:send |
   send insertAfter:
      'Logger logSendTo: <meta: #receiver>' .
]
```

- How can we access the receiver of the send?
- Solution: Metavariable

Implementation Metavariables

> Stack during send:



> Problem I: After send, receiver is not available
> Problem II: Before send, receiver is deep in the stack

Implementation Metavariables

> Solution: ByteSurgeon generates preamble
  - Pop the arguments into temps
  - Pop the receiver into temps
  - Rebuild the stack
  - Do the send
  - Now we can access the receiver even after the send

Implementation Metavariables

```
25 <70> self
26 <81 40> storeIntoTemp: 0
28 <D0> send: test
29 <41> pushLit: Transcript
30 <10> pushTemp: 0
31 <E2> send: show:
32 <87> pop
33 <87> pop
34 <78> returnSelf
```

Preamble

Inlined Code

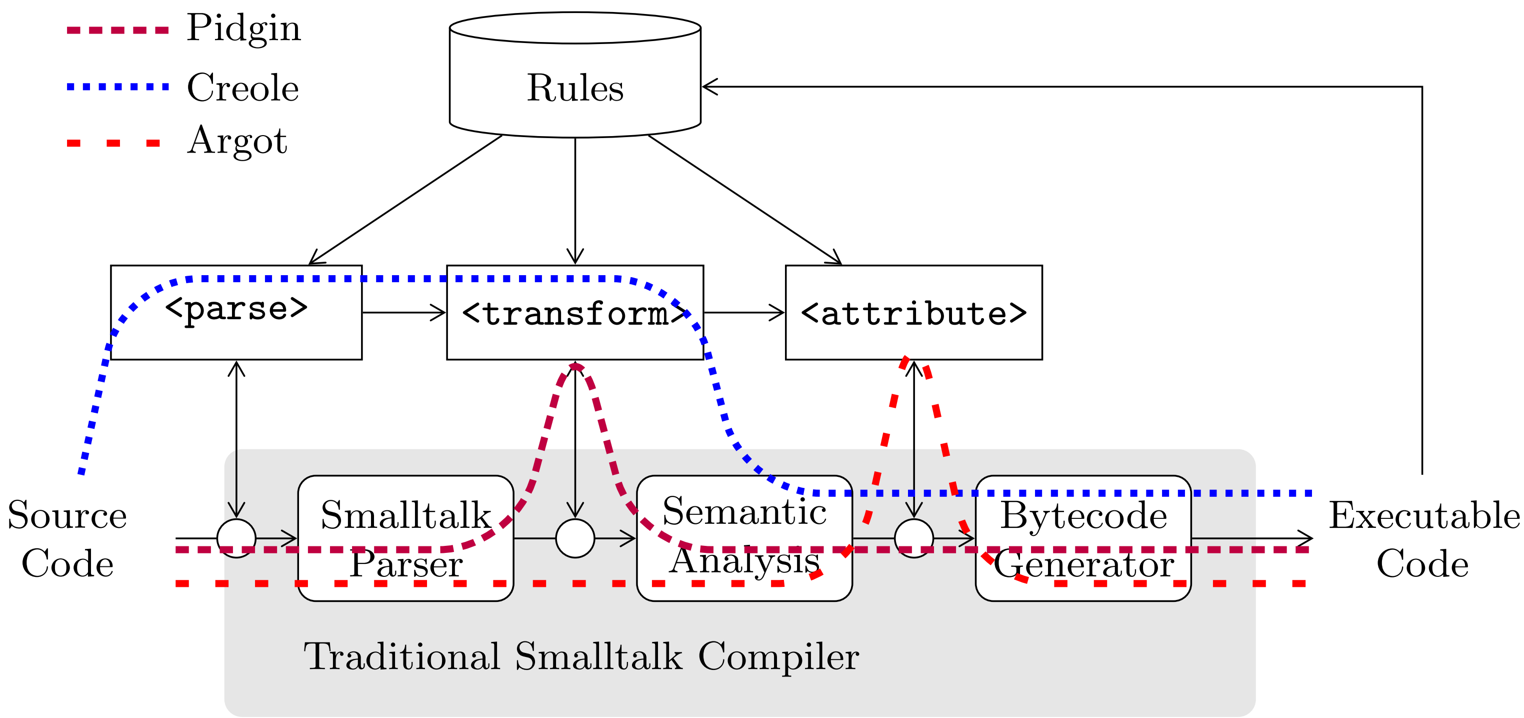
Why do we care?

> Helvetia — Context Specific Languages with Homogeneous Tool Integration

> Reflectivity — Unanticipated partial behavioral reflection.

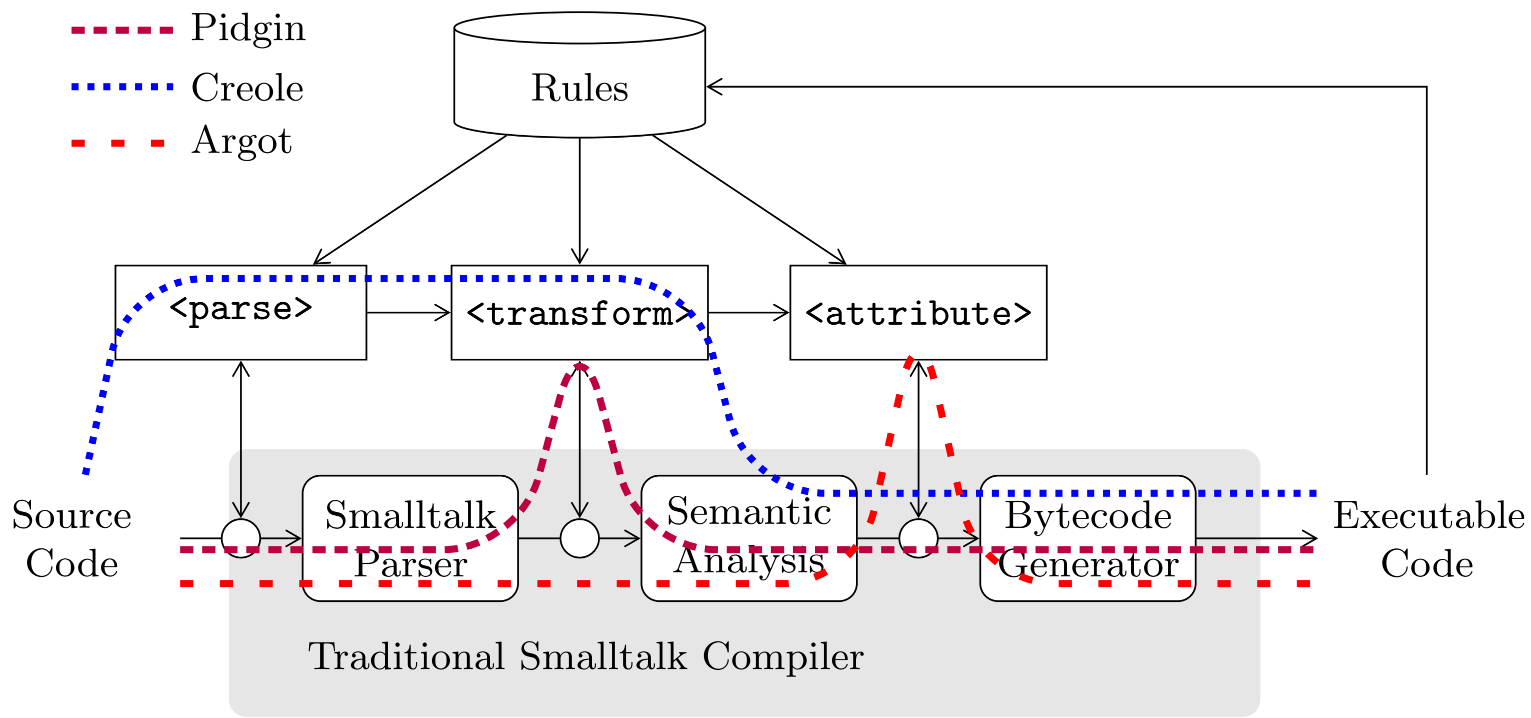
> Albedo — A unified approach to reflection.

Helvetia



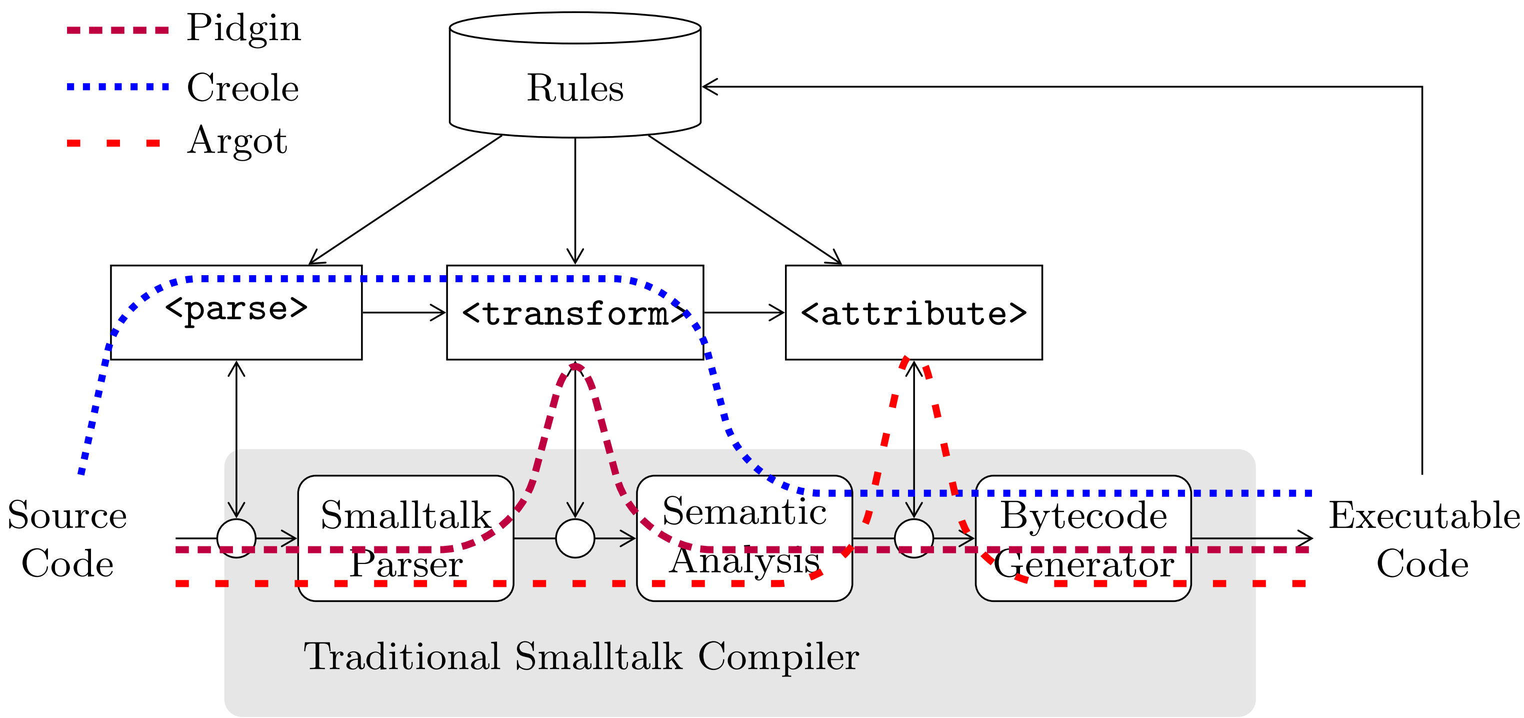
Pidgin
Creole
Argot
Rules
<parse>
<transform>
<attribute>
Source Code
Smalltalk Parser
Semantic Analysis
Bytecode Generator
Executable Code
Traditional Smalltalk Compiler

Helvetia



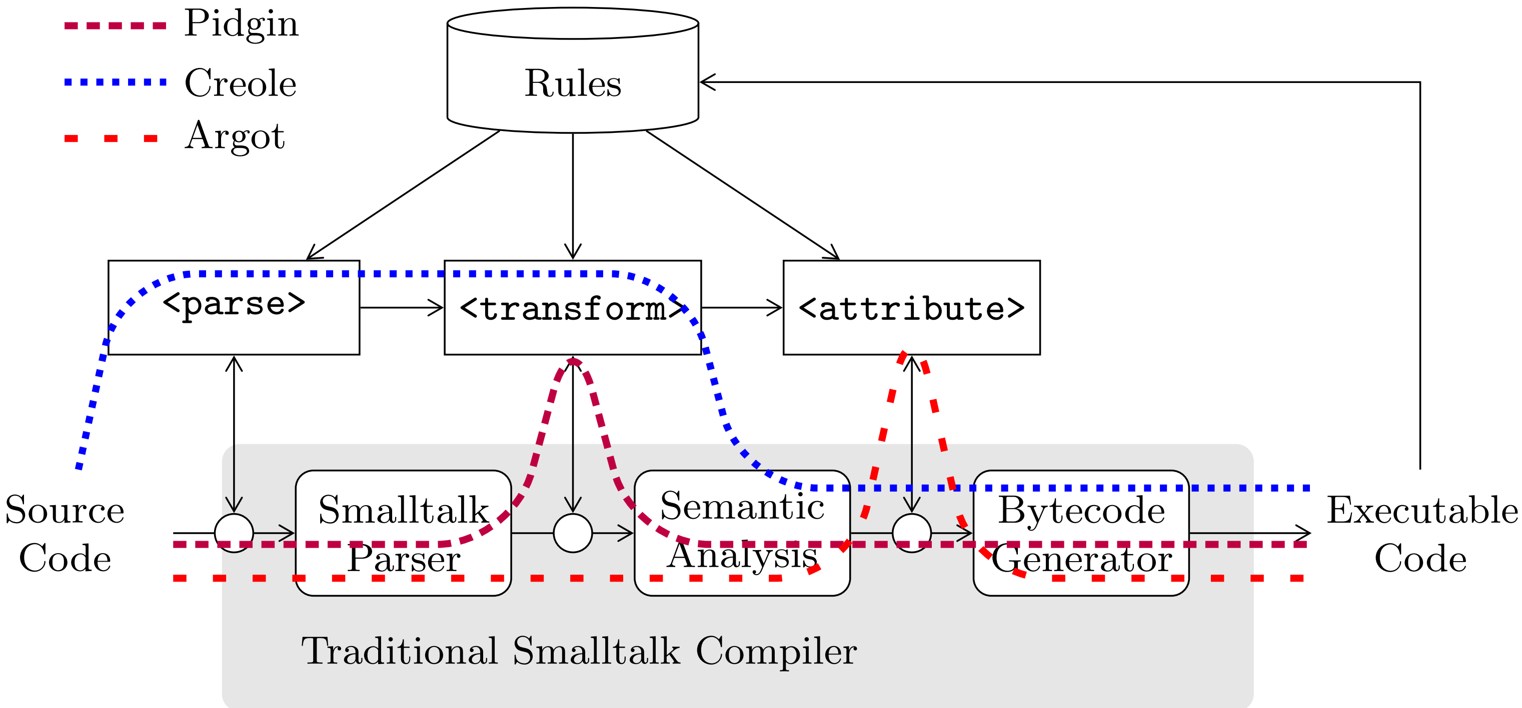
Pidgin
Creole
Argot
Rules
<parse>
<transform>
<attribute>
Source Code
Smalltalk Parser
Semantic Analysis
Bytecode Generator
Executable Code
Traditional Smalltalk Compiler

Helvetia



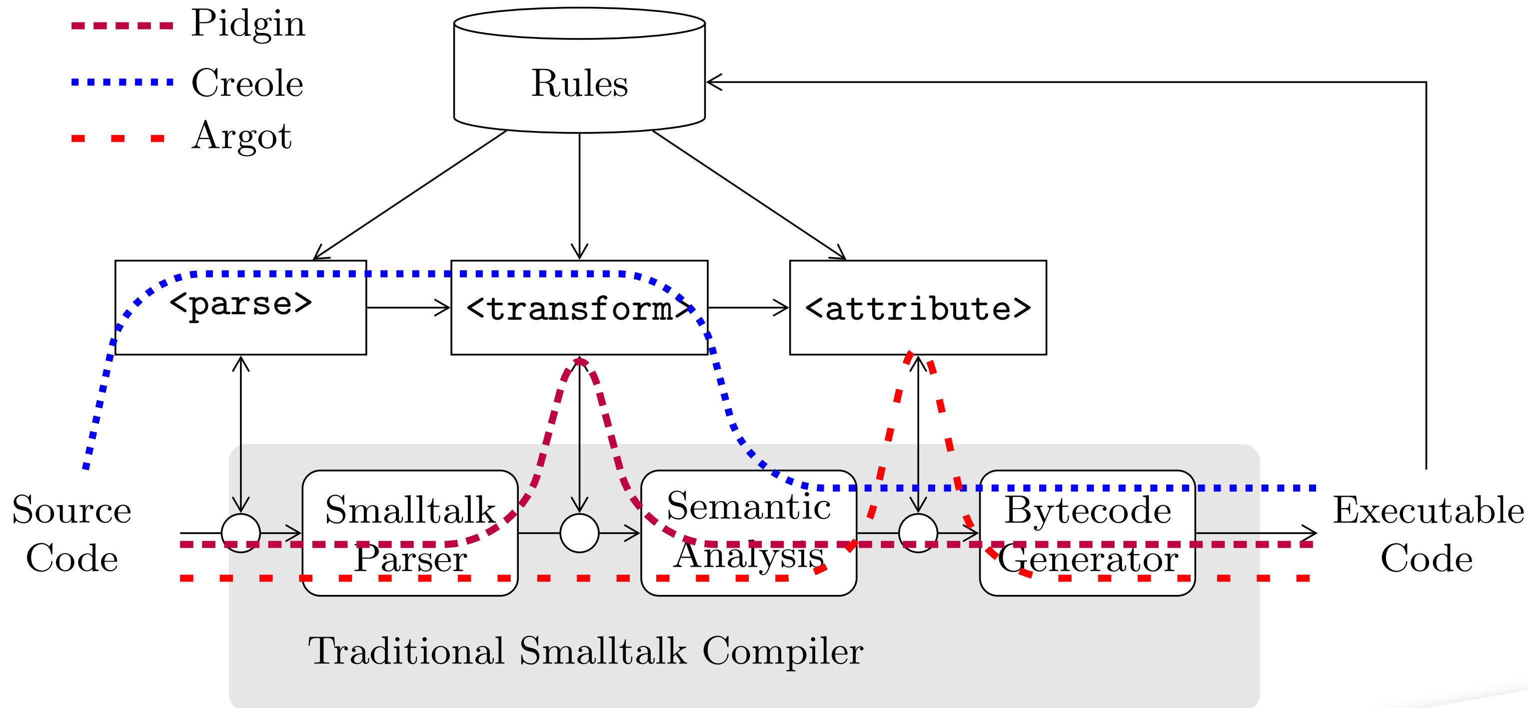
Pidgin
Creole
Argot
Rules
<parse>
<transform>
<attribute>
Source Code
Smalltalk Parser
Semantic Analysis
Bytecode Generator
Executable Code
Traditional Smalltalk Compiler

Helvetia



Pidgin
Creole
Argot
Rules
<parse>
<transform>
<attribute>
Source Code
Smalltalk Parser
Semantic Analysis
Bytecode Generator
Executable Code
Traditional Smalltalk Compiler

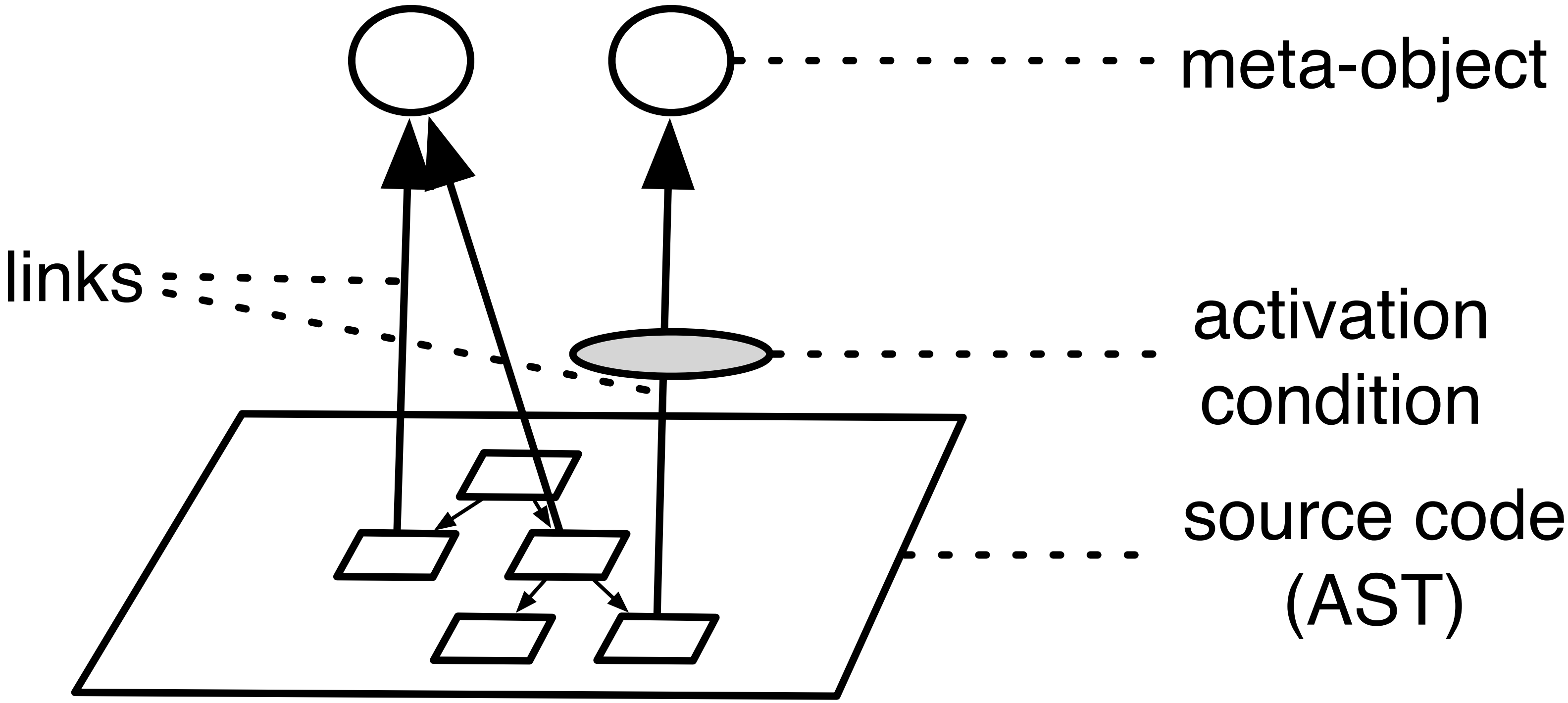
Helvetia



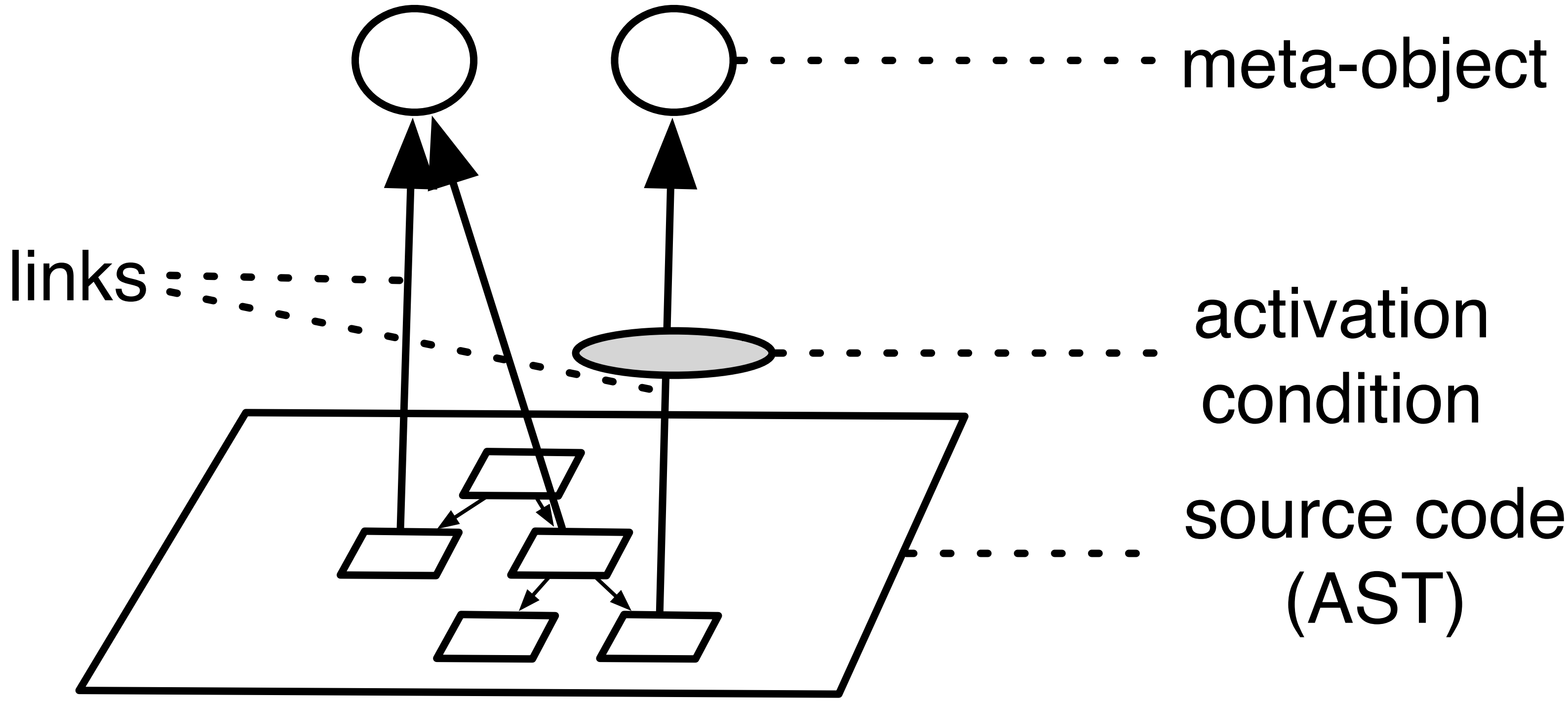
Pidgin
Creole
Argot
Rules
<parse>
<transform>
<attribute>
Source Code
Smalltalk Parser
Semantic Analysis
Bytecode Generator
Executable Code
Traditional Smalltalk Compiler

Helvetia
Renggli 2010

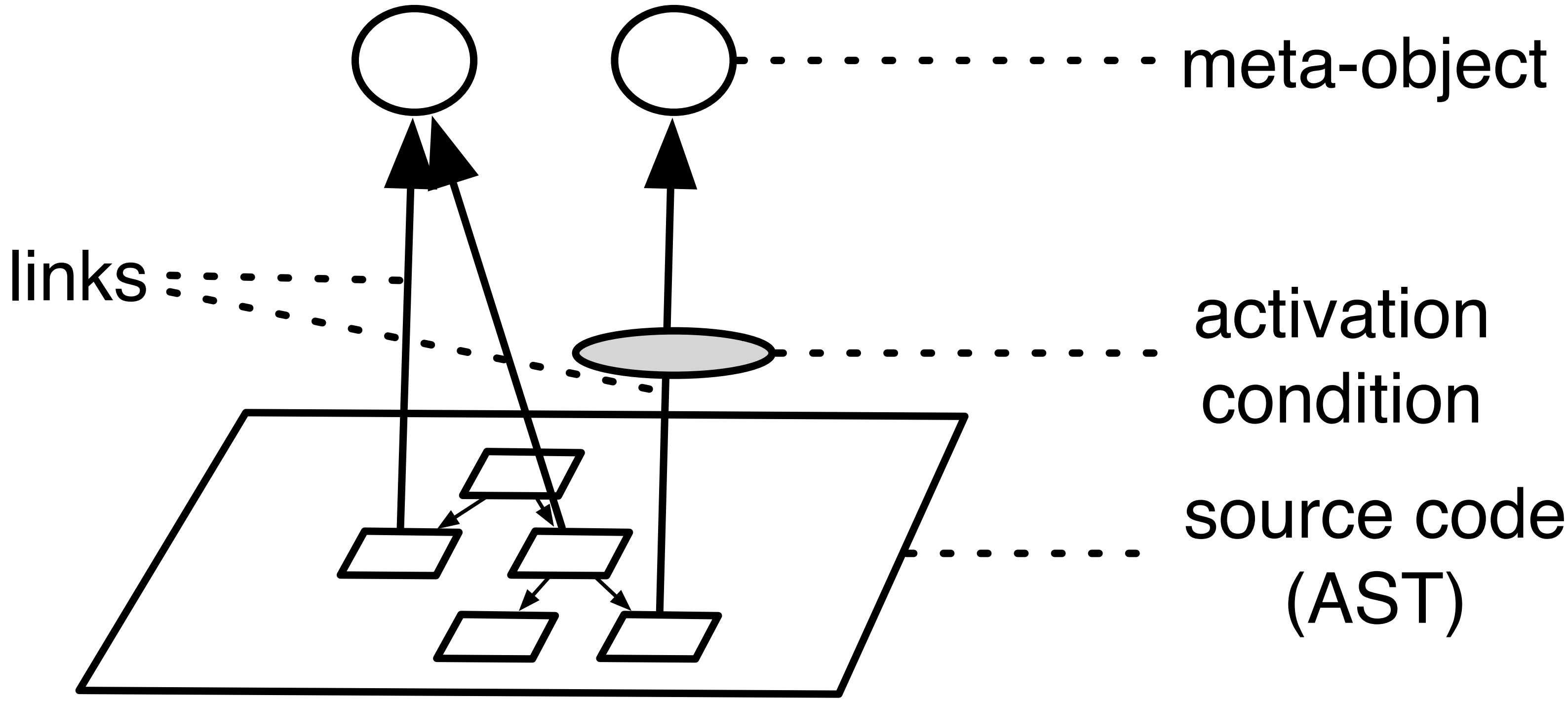
Reflectivity



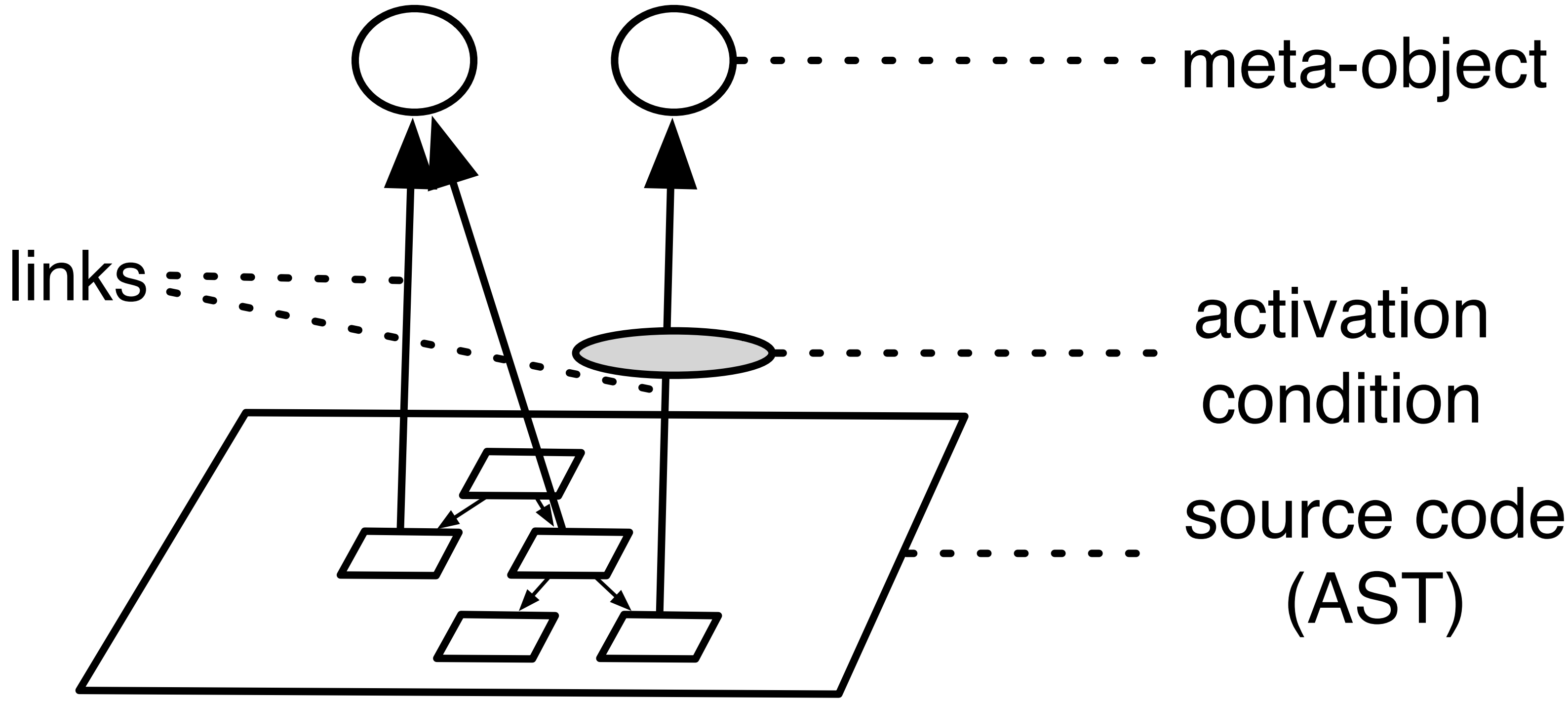
Reflectivity



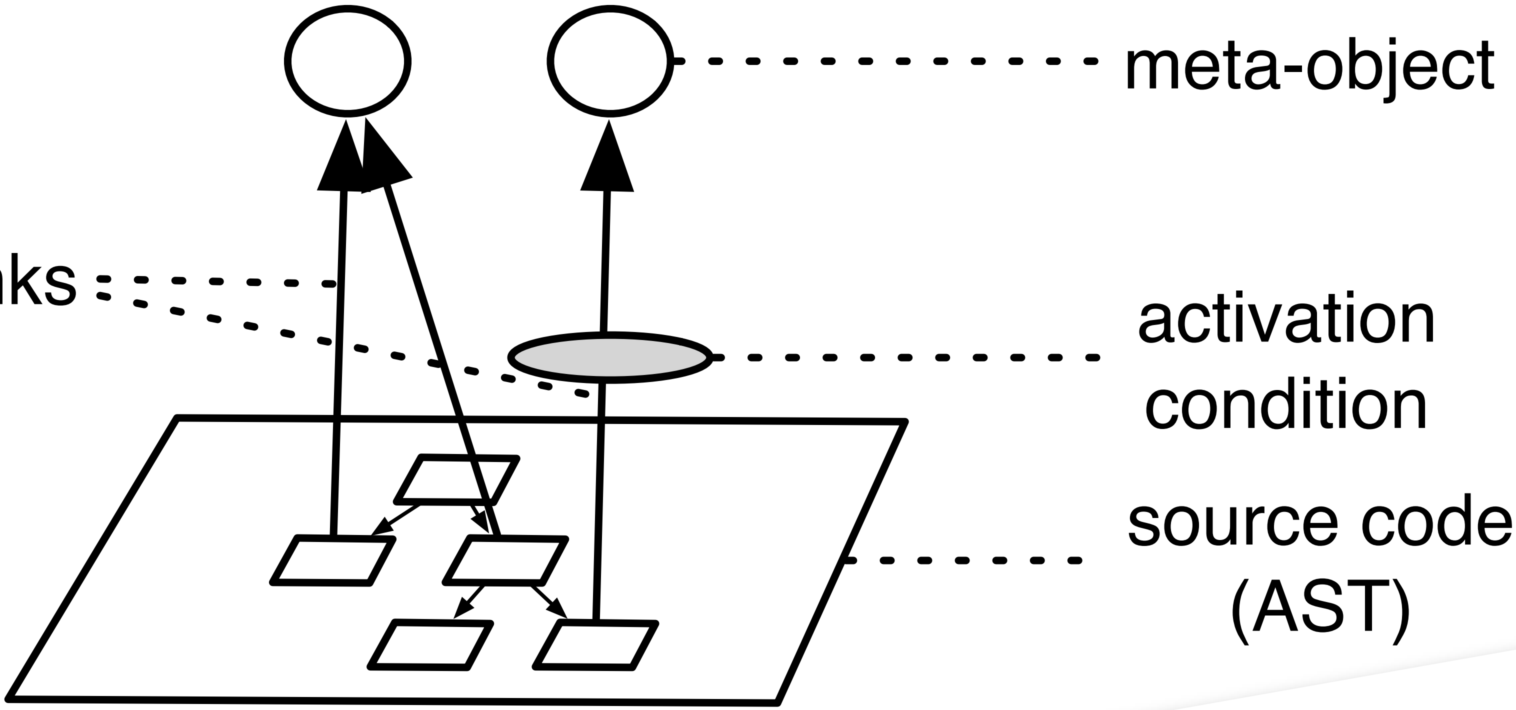
Reflectivity



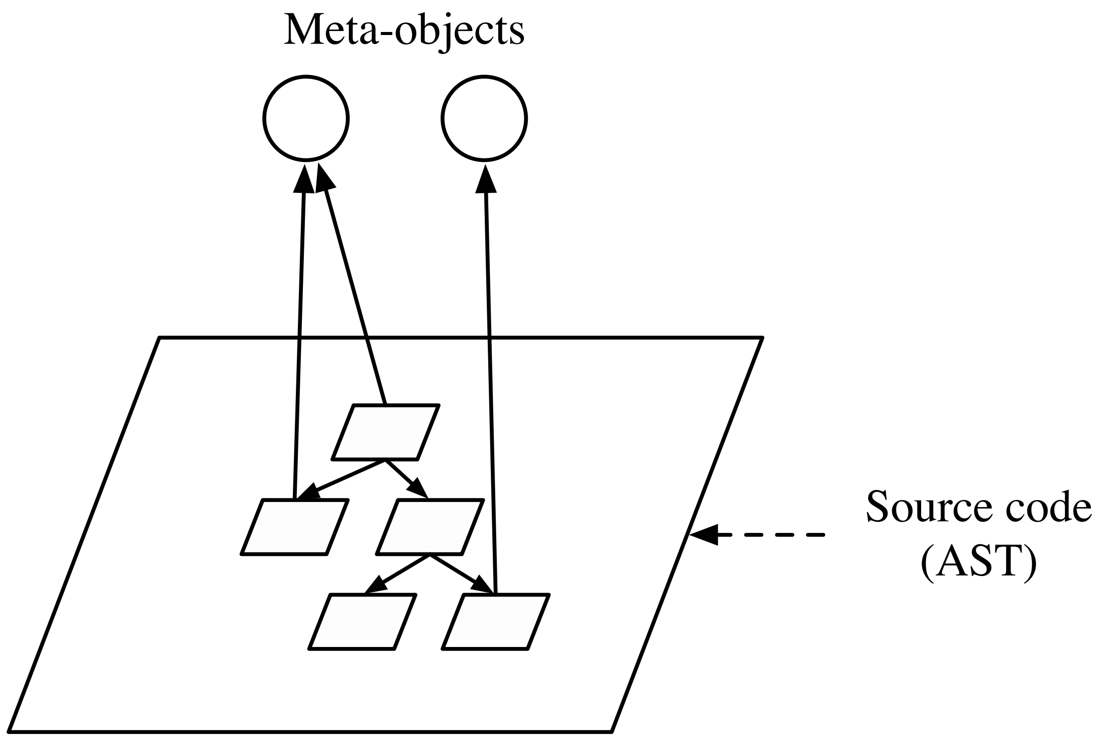
Reflectivity



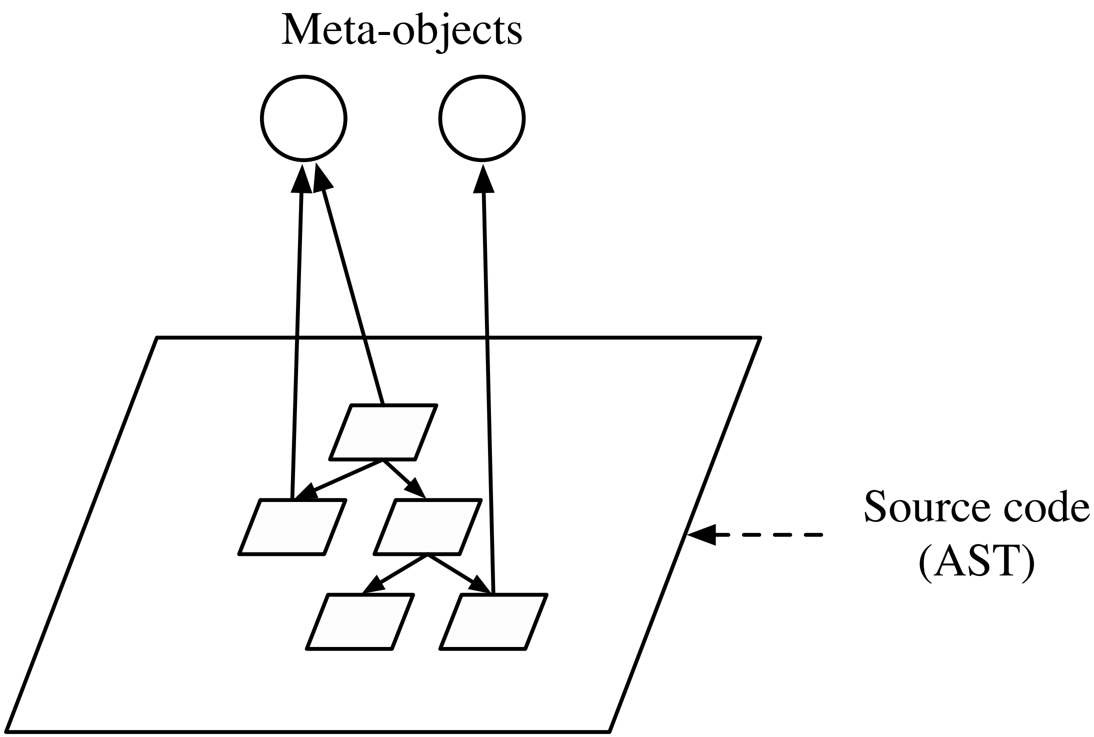
Reflectivity



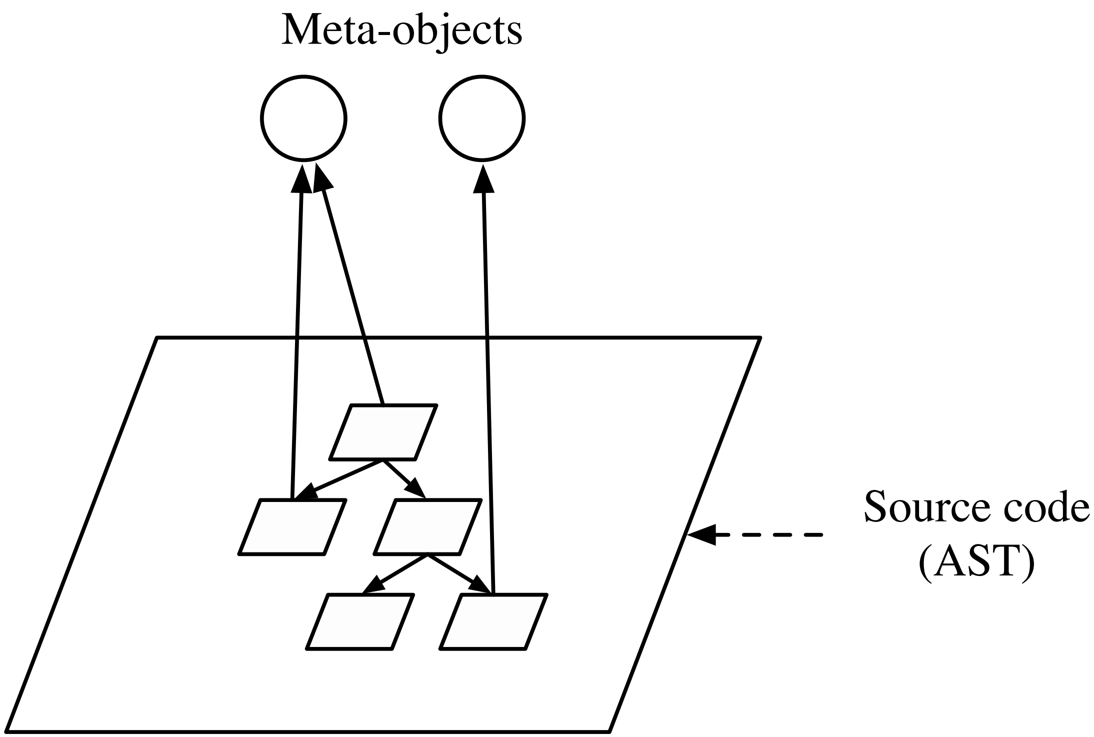
Albedo



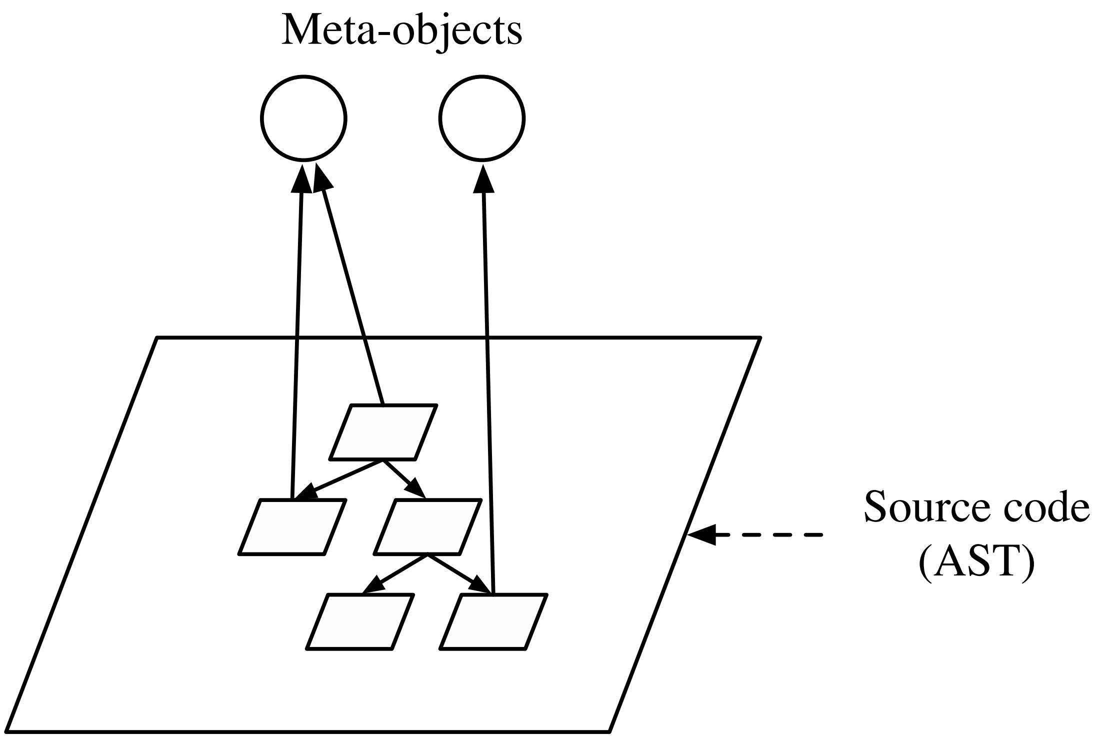
Albedo



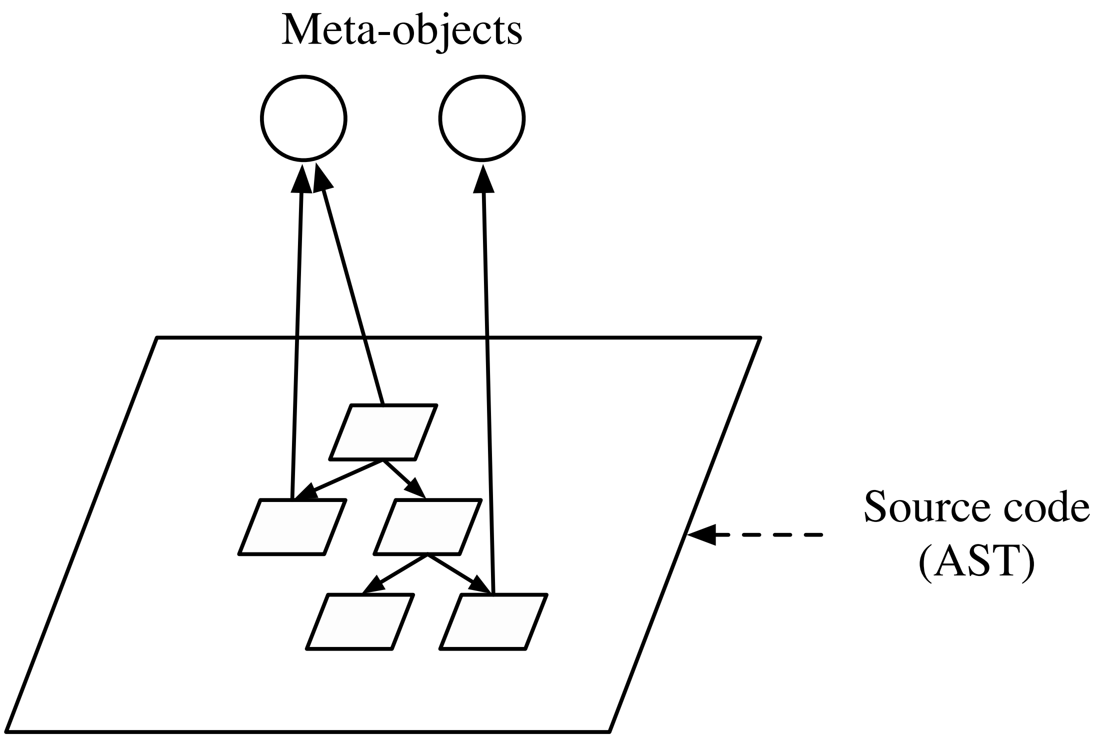
Albedo



Albedo



Albedo



Opal Compiler

http://scg.unibe.ch/research/OpalCompiler