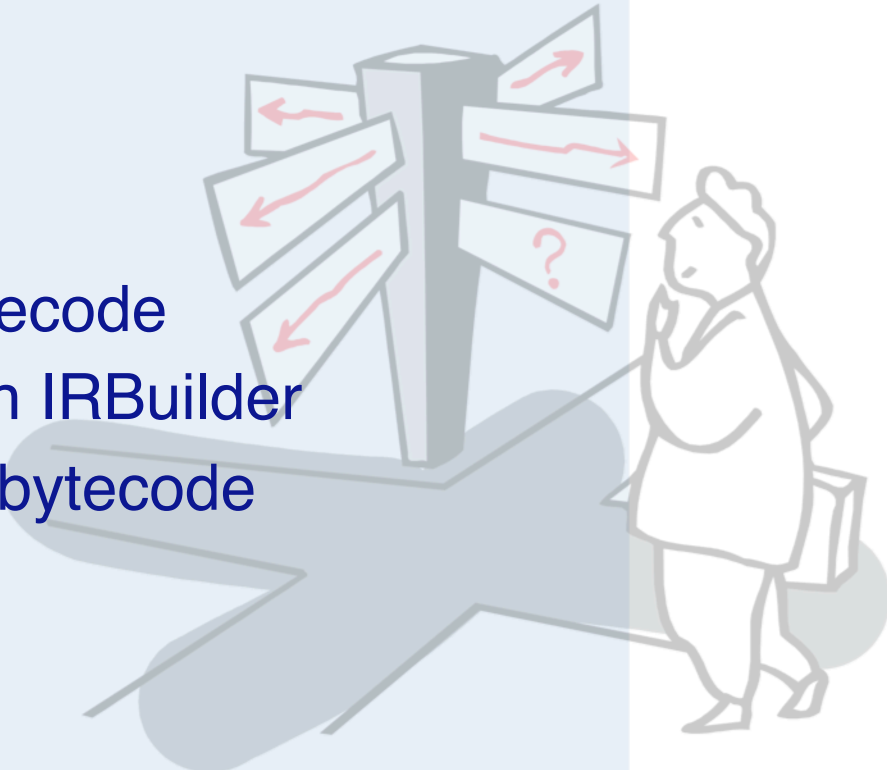
UNIVERSITÄT BERN

11. Working with Bytecode

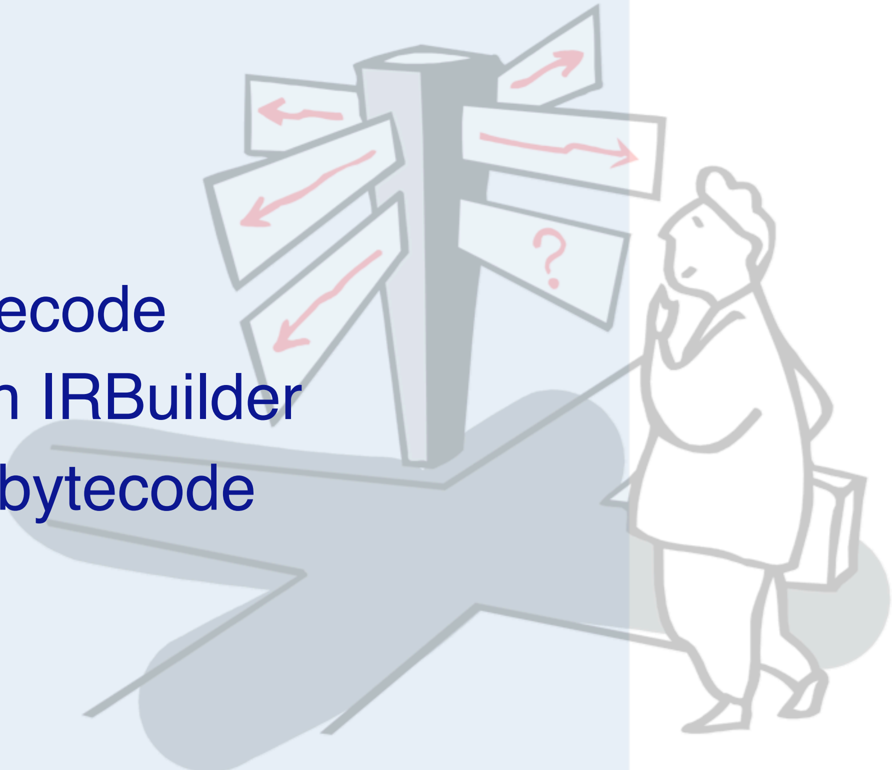
Roadmap



> The Pharo compiler
> Introduction to Pharo bytecode
> Generating bytecode with IRBuilder
> Parsing and Interpreting bytecode

Original material by Marcus Denker

Roadmap



> **The Pharo compiler**
> Introduction to Pharo bytecode
> Generating bytecode with IRBuilder
> Parsing and Interpreting bytecode

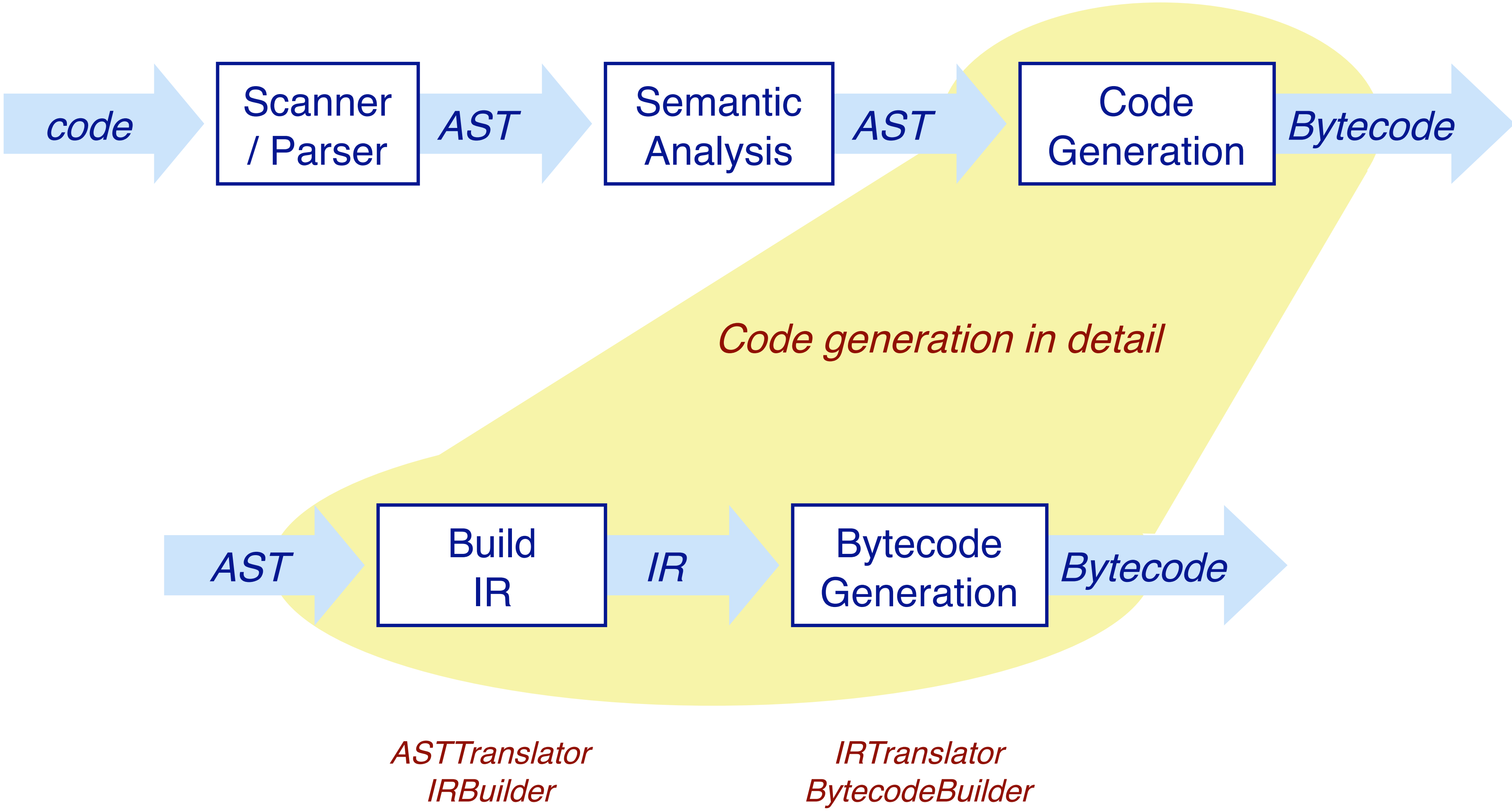
The Pharo Compiler

- Default compiler
  - very old design
  - quite hard to understand
  - impossible to modify and extend
- New compiler for Pharo
  - http://www.iam.unibe.ch/~scg/Research/NewCompiler/
  - adds support for true block closures (optional)

The Pharo Compiler

- Fully reified compilation process:
  - Scanner/Parser (built with SmaCC)
    - *builds AST (from Refactoring Browser)*
  - Semantic Analysis: ASTChecker
    - *annotates the AST (e.g., var bindings)*
  - Translation to IR: ASTTranslator
    - *uses IRBuilder to build IR (Intermediate Representation)*
  - Bytecode generation: IRTranslator
    - *uses BytecodeBuilder to emit bytecodes*

Compiler: Overview



Compiler: Syntax

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

> Input:
  — Scanner definition: regular expressions
  — Parser: BNF-like grammar
  — Code that builds AST as annotation

> Output:
  — class for Scanner (subclass SmaCCScanner)
  — class for Parser (subclass SmaCCParser)

Scanner

SmaCCParserGenerator: SqueakScanner/SqueakParser

Scanner Parser Compile Test Tutorial

```
<decimalNumber>:        [0-9]+ (\. [0-9]+)? ;
<radixNumber>:          [0-9]+ r [0-9A-Z]+ (\. [0-9A-Z]+)? ;
<scaledNumber>:         <decimalNumber> s [0-9]+ ;
<exponentNumber>:       (<decimalNumber> | <radixNumber>) e \-? [0-9]+ ;
<number>:               <decimalNumber> | <radixNumber> | <exponentNumber> | <scaledNumber> ;
<negativeNumber>:       \- <number> ;
<string>:               \'  [↑\']* \' (\' [↑\']* \')* ;
<name>:                 [a-zA-Z] [a-zA-Z0-9]* ;
<keyword>:              <name> \: ;
<multikeyword>:         <name> \: (<name> \: )+ ;
<binarySymbol>:         [\~\!\@\%\&\*\-\+\=\\\|\?\/\>\<\,] [\~\!\@\%\&\*\-\+\=\\\|\?\/\>\<\,]* ;
<assignment>:           \: \= | \_ ;
<alternateKeyword>: \: <name> \: (<name> \:)* ;
<whitespace>:           \s+ ;
<comment>:              \" [↑\"]* \" ;
<character>:            \$ . ;
<period>:               \. ;
<variableAssignment>:      <name> \: \= ;

<anyChar>:              . ; # For VW literal arrays that handle #(;) -> #(#';')
```

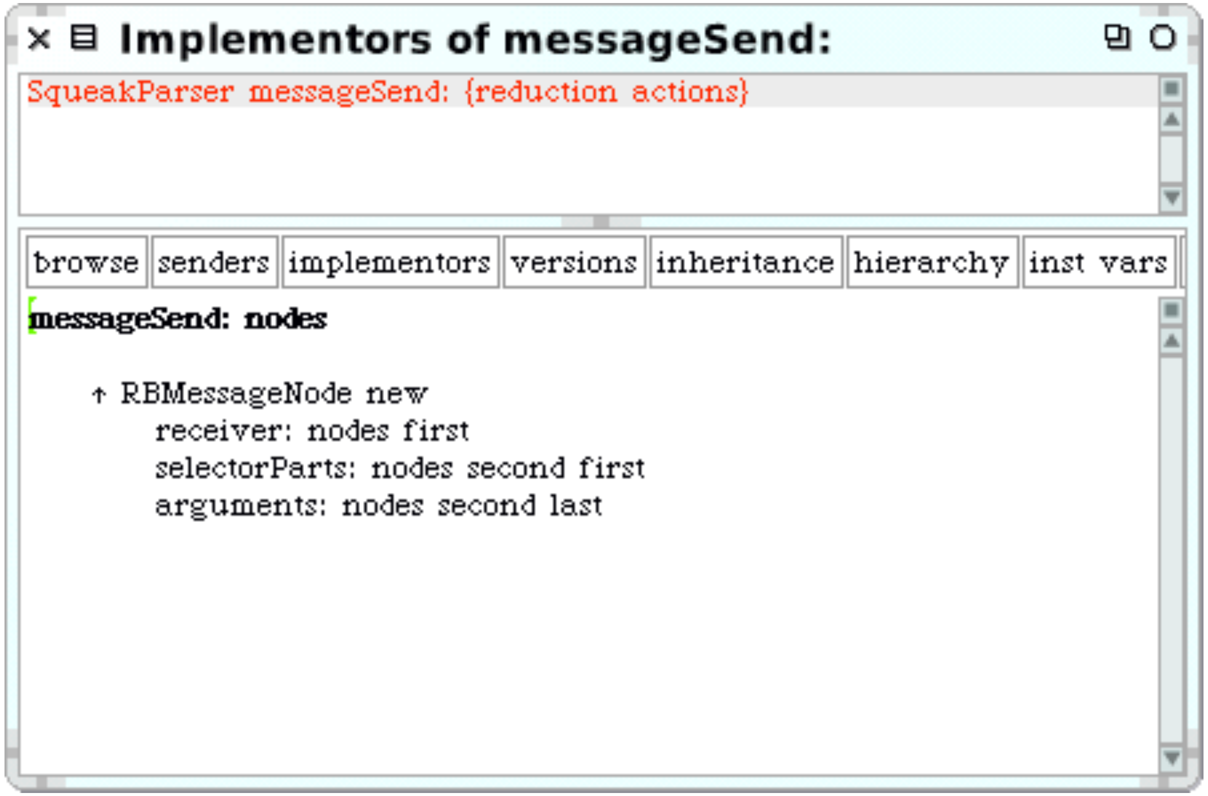
Parser

SmaCCParserGenerator: SqueakScanner/SqueakParser

Scanner Parser Compile Test Tutorial

```
%id <name> <number> <negativeNumber> <binarySymbol> <period>;
%start Sequence MethodPattern;

Method:
	MethodPattern Sequence                          {#method:}
|	MethodPattern Primitive Sequence                {#methodPrim:}
|	MethodPattern Temporaries Primitive Statements {#methodTempsPrim:};

MethodPattern:
	<name>                                          {#unaryMessage:}
|	<binarySymbol> Variable                         {#messagePart:}
|	KeywordMethodPattern                            {#first:};

KeywordMethodPattern:
	<keyword> Variable                              {#messagePart:}
|	KeywordMethodPattern <keyword> Variable         {#addMessagePart:};

Primitive:
	"<" PrimitiveMessage ">"                        {#primitiveMessage:};

Sequence:
	Statements                                      {#sequence:}
|	Temporaries Statements                          {#sequenceWithTemps:};
```

Calling Parser code



Compiler: AST

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

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

Compiler: Semantics

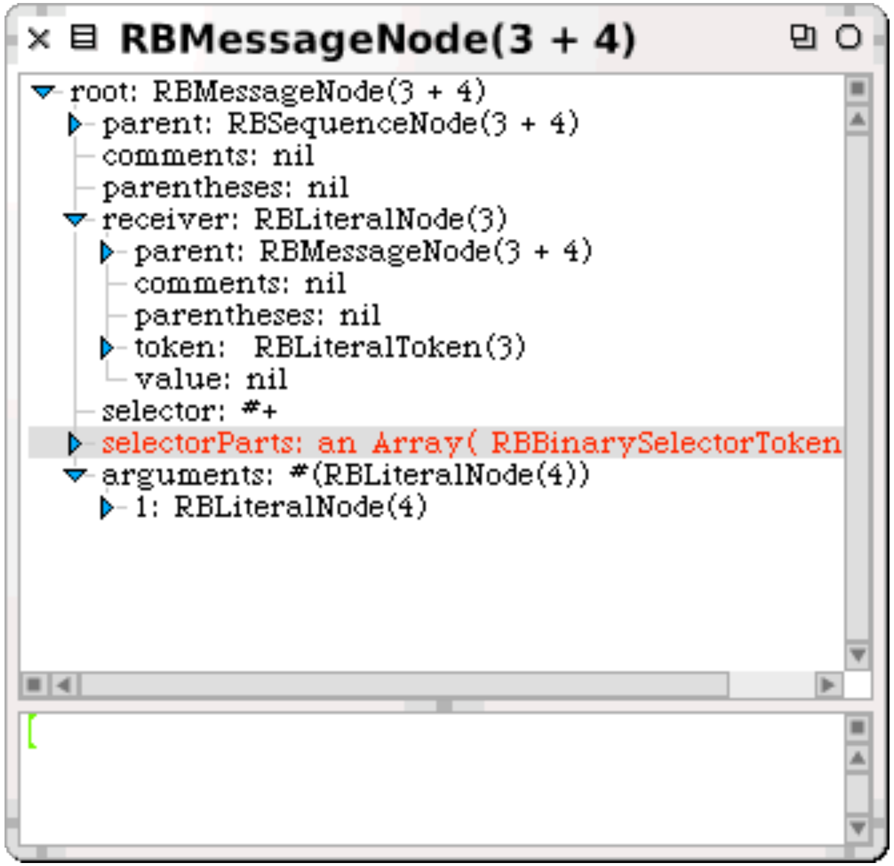
> We need to analyse the AST
- Names need to be linked to the variables according to the scoping rules

> ASTChecker 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

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: 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 ASTTranslator:
  — AST Visitor
  — Uses IRBuilder

Compiler: Bytecode Generation

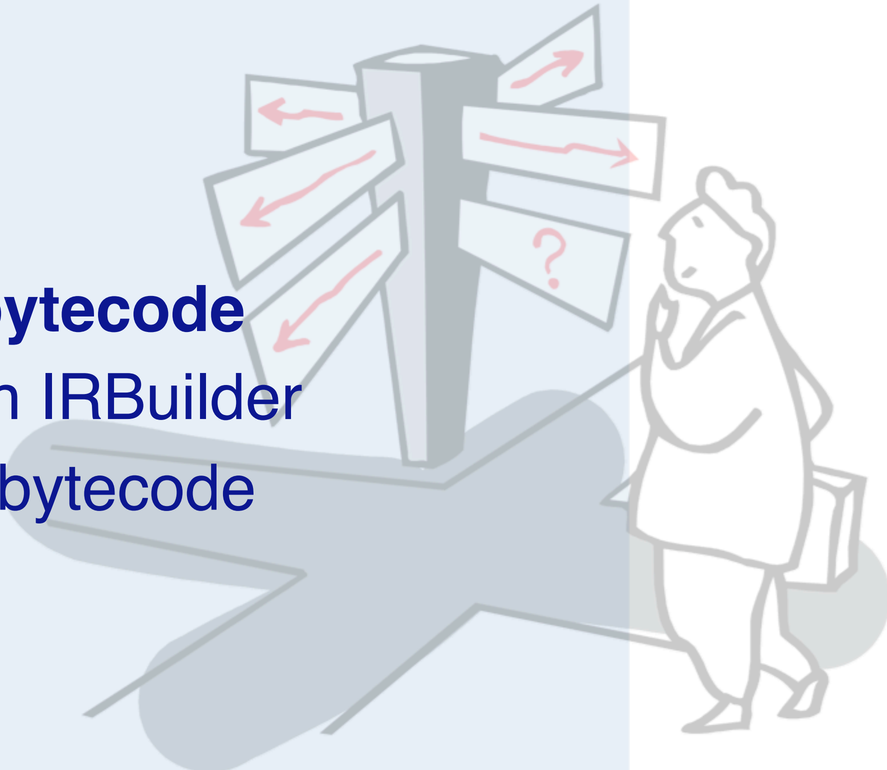
> IR needs to be converted to Bytecode
  — IRTranslator: Visitor for IR tree
  — Uses BytecodeBuilder to generate Bytecode
  — Builds a compiledMethod
  — Details to follow next section

```
testReturn1
   | iRMethod aCompiledMethod |
   iRMethod := IRBuilder new
      numRargs: 1;
      addTemps: #(self);    "receiver and args declarations"
      pushLiteral: 1;
      returnTop;
      ir.
```

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

Roadmap

> The Pharo compiler
> **Introduction to Pharo bytecode**
> Generating bytecode with IRBuilder
> Parsing and Interpreting bytecode



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

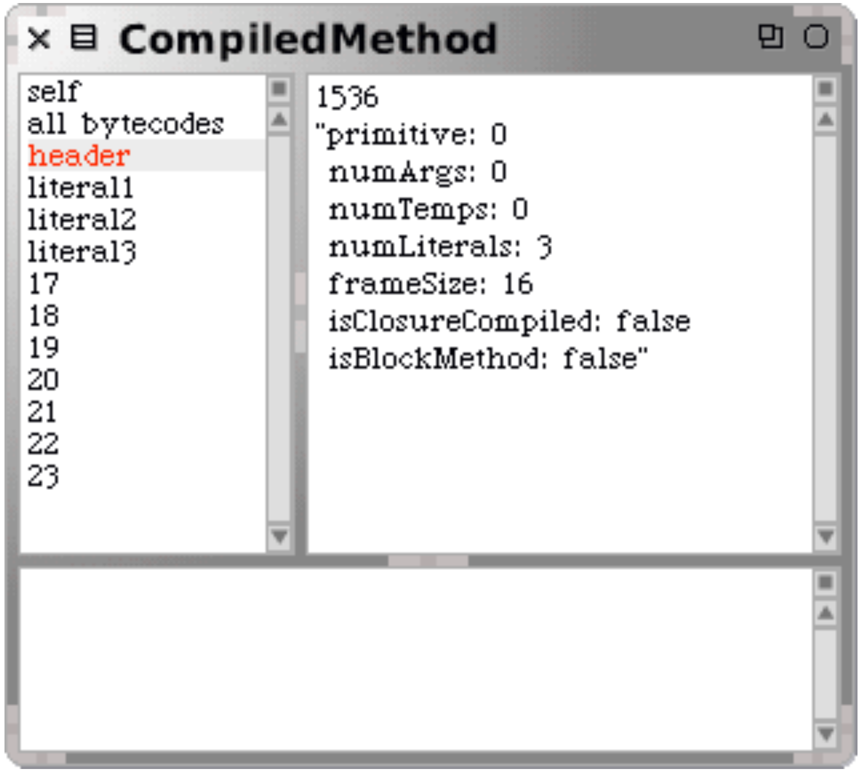
> Smalltalk (like Java): Stack machine
  - easy to implement interpreters for different processors
  - most hardware processors are register machines

> Pharo 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*
  - 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*

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

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
```

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
```

Closure Bytecode

> 138 Push (Array new: k)/Pop k into: (Array new: j)

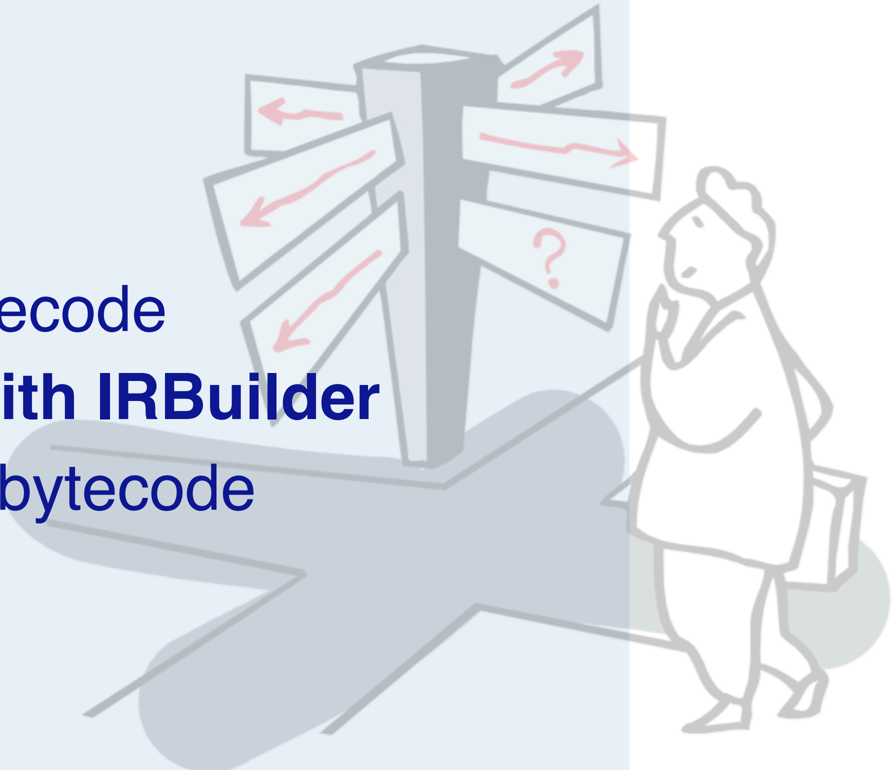
> 140 Push Temp At k In Temp Vector At: j

> 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 Pharo bytecode
- > **Generating bytecode with IRBuilder**
- > Parsing and Interpreting bytecode

Generating Bytecode

> IRBuilder: A tool for generating bytecode
  - Part of the NewCompiler
  - Pharo: Install packages AST, NewParser, NewCompiler

> Like an Assembler for Pharo

IRBuilder: Simple Example

> *Number*>>asInteger

```
iRMethod := IRBuilder new
   numRargs: 1;          "receiver"
   addTemps: #(self);    "receiver and args"
   pushTemp: #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
    numRargs: 1;
    addTemps: #(self);                  "receiver"
    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
      numRargs: 1;
      addTemps: #(self);          "receiver and args"
      pushLiteral: 2;
      storeInstVar: 1;
      pushTemp: #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
      numRargs: 1;
      addTemps: #(self);     "receiver"
      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;
```

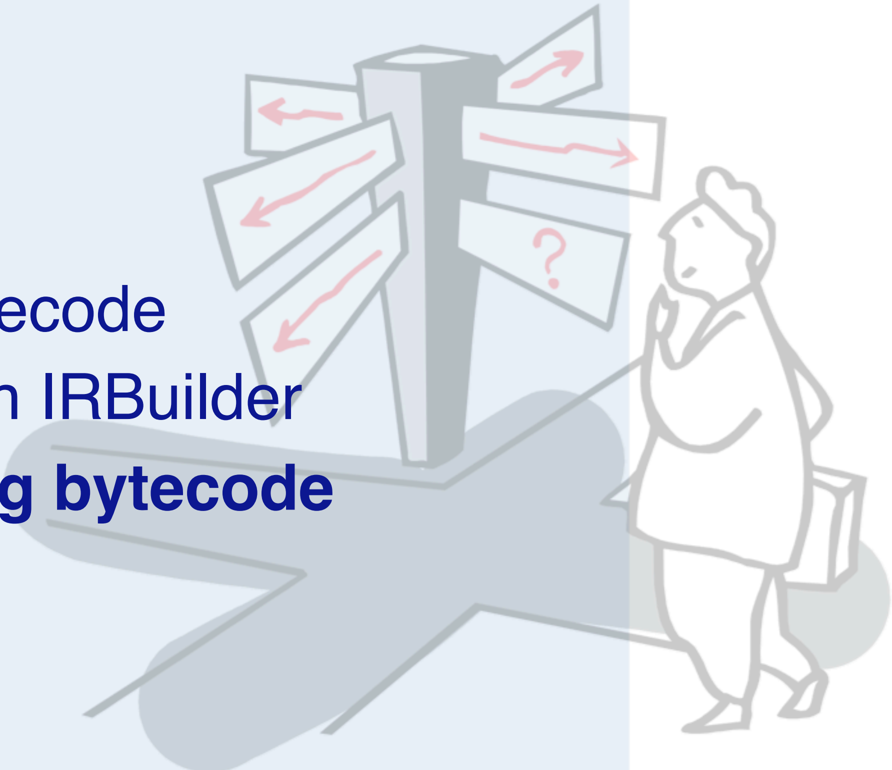
> super send

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

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

Roadmap

- The Pharo compiler
- Introduction to Pharo bytecode
- Generating bytecode with IRBuilder
- **Parsing and Interpreting bytecode**



Parsing and Interpretation

- First step: *Parse bytecode*
  - enough for easy analysis, pretty printing, decompilation
- Second step: *Interpretation*
  - needed for simulation, complex analyis (e.g., profiling)
- Pharo provides frameworks for both:
  - InstructionStream/InstructionClient (parsing)
  - ContextPart (Interpretation)

The InstructionStream Hierarchy

```
InstructionStream
  ContextPart
    BlockContext
    MethodContext
  Decompiler
  InstructionPrinter
  InstVarRefLocator
  BytecodeDecompiler
```

InstructionStream

- Parses the byte-encoded instructions
- State:
  - pc: program counter
  - sender: the method (bad name!)

```
Object subclass: #InstructionStream
   instanceVariableNames: 'sender pc'
   classVariableNames: 'SpecialConstants'
   poolDictionaries: ''
   category: 'Kernel-Methods'
```

Usage

> Generate an instance:

```
instrStream := InstructionStream on: aMethod
```

> Now we can step through the bytecode with:

```
instrStream interpretNextInstructionFor: client
```

> Calls methods on a client object for the type of bytecode, e.g.

— `pushReceiver`

— `pushConstant: value`

— `pushReceiverVariable: offset`

InstructionClient

- Abstract superclass
  - Defines empty methods for all methods that InstructionStream calls on a client
- For convenience:
  - Clients don’t need to inherit from this class

```
Object subclass: #InstructionClient
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Kernel-Methods'
```

Example: A test

```
InstructionClientTest>>testInstructions
   "just interpret all of methods of Object"
   | methods client scanner|

   methods := Object methodDict values.
   client := InstructionClient new.

   methods do: [:method |
      scanner := (InstructionStream on: method).
      [scanner pc <= method endPC] whileTrue: [
            self shouldnt:
                  [scanner interpretNextInstructionFor: client]
               raise: Error.
      ].
   ].
```

Example: Printing Bytecode

- InstructionPrinter:
  - Print the bytecodes as human readable text
- Example:
  - print the bytecode of *`Number`*`>>asInteger`:

```
String streamContents:
   [:str | (InstructionPrinter on: Number>>#asInteger)
            printInstructionsOn: str ]
```

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

InstructionPrinter

> Class Definition:

```
InstructionClient subclass: #InstructionPrinter
   instanceVariableNames: 'method scanner
                           stream indent'
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Kernel-Methods'
```

InstructionPrinter

> Main Loop:

```
InstructionPrinter>>printInstructionsOn: aStream
   "Append to the stream, aStream, a description
   of each bytecode in the instruction stream."
   | end |
   stream := aStream.
   scanner := InstructionStream on: method.
   end := method endPC.
   [scanner pc <= end]
      whileTrue: [scanner interpretNextInstructionFor: self]
```

InstructionPrinter

- Overwrites methods from `InstructionClient` to print the bytecodes as text
- e.g. the method for `pushReceiver`

```
InstructionPrinter>>pushReceiver
    "Print the Push Active Context's Receiver
    on Top Of Stack bytecode."

    self print: 'self'
```

Example: InstVarRefLocator

```
InstructionClient subclass: #InstVarRefLocator
    instanceVariableNames: 'bingo'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Kernel-Methods'

InstVarRefLocator>>interpretNextInstructionUsing: aScanner
    bingo := false.
    aScanner interpretNextInstructionFor: self.
    ^bingo

InstVarRefLocator>>popIntoReceiverVariable: offset
    bingo := true

InstVarRefLocator>>pushReceiverVariable: offset
    bingo := true

InstVarRefLocator>>storeIntoReceiverVariable: offset
    bingo := true
```

InstVarRefLocator

> Analyse a method, answer true if it references an instance variable

```
CompiledMethod>>hasInstVarRef
    "Answer whether the receiver references an instance variable."

    | scanner end printer |

    scanner := InstructionStream on: self.
    printer := InstVarRefLocator new.
    end := self endPC.

    [scanner pc <= end] whileTrue:
        [ (printer interpretNextInstructionUsing: scanner)
            ifTrue: [^true]. ].
    ^false
```

InstVarRefLocator

> Example for a simple bytecode analyzer
> Usage:

```
aMethod hasInstVarRef
```

> (has reference to variable testSelector)

```
(TestCase>>#debug) hasInstVarRef
```

```
true
```

> (has no reference to a variable)

```
(Integer>>#+) hasInstVarRef
```

```
false
```

ContextPart: Semantics for Execution

> Sometimes we need more than parsing
  - "stepping" in the debugger
  - system simulation for profiling

```
InstructionStream subclass: #ContextPart
   instanceVariableNames: 'stackp'
   classVariableNames: 'PrimitiveFailToken QuickStep'
   poolDictionaries: ''
   category: 'Kernel-Methods'
```

Simulation

- Provides a complete Bytecode interpreter
- Run a block with the simulator:

```
(ContextPart runSimulated: [3 factorial])
```

```
6
```

Profiling: MessageTally

> Usage:

```
MessageTally tallySends: [3 factorial]
```

```
This simulation took 0.0 seconds.
**Tree**
1 SmallInteger(Integer)>>factorial
  1 SmallInteger(Integer)>>factorial
    1 SmallInteger(Integer)>>factorial
      1 SmallInteger(Integer)>>factorial
```

> Other example:

```
MessageTally tallySends: ['3' + 1]
```

What you should know!

- ✎ *What are the problems of the old compiler?*
- ✎ *How is the new Pharo compiler organized?*
- ✎ *What does the Pharo semantic analyzer add to the parser-generated AST?*
- ✎ *What is the format of the intermediate representation?*
- ✎ *What kind of virtual machine does the Pharo bytecode address?*
- ✎ *How can you inspect the bytecode of a particular method?*

Can you answer these questions?

- ✎ *What different groups of bytecode are supported?*
- ✎ *Why is the SmaCC grammar only BNF-"like"?*
- ✎ *How can you find out what all the bytecodes are?*
- ✎ *What is the purpose of IRBuilder?*
- ✎ *Why do we not generate bytecode directly?*
- ✎ *What is the responsibility of class InstructionStream?*
- ✎ *How would you implement a statement coverage analyzer?*

License

http://creativecommons.org/licenses/by-sa/3.0/