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

Code generation in detail

ASTTranslator
IRBuilder

IRTranslator
BytecodeBuilder

code → Scanner / Parser → AST → Semantic Analysis → AST → Code Generation → Bytecode

AST → Build IR → IR → Bytecode Generation → Bytecode
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](image)

```plaintext
<decimalNumber>:  [0-9]+ \(, [0-9]+\)? ;
<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 \W literal arrays that handle #() -> #(\#;\#)
```

© Oscar Nierstrasz
Parser

```
%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

Implementors of `messageSend`:

```
SqueakParser messageSend: {reduction actions}

browse  senders  implementors  versions  inheritance  hierarchy  inst vars

messageSend: nodes

    + RBMessageNode new
      receiver: nodes first
      selectorParts: nodes second first
      arguments: nodes second last
```
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
RBReturnValue
RBBlockNode
RBCascadeNode
RBLiteralNode
RBMessageNode
RBOptimizedNode
RBVariableNode
```

© Oscar Nierstrasz
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);
  pushLiteral: 1;
  returnTop;
  ir.

"receiver and args declarations"

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

<table>
<thead>
<tr>
<th>Type</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 bits</td>
<td>4 bits</td>
</tr>
</tbody>
</table>
Example: Number>>asInteger

> Smalltalk code:

```smalltalk
Number>>asInteger
  "Answer an Integer nearest the receiver toward zero."
  ^self truncated
```

> Symbolic Bytecode

```smalltalk
9 <70> self
10 <D0> send: truncated
11 <7C> returnTop
```
Example: Step by Step

>  9  \texttt{self}  
    —  The receiver (self) is pushed on the stack
>  10  \texttt{send: truncated}  
    —  Bytecode 208: send literal 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  \texttt{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)
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
inject: thisValue into: binaryBlock
   | nextValue |
nextValue := thisValue.
self
   do: [:each | nextValue := binaryBlock value: nextValue value: each].
   ^nextValue
Contexts

``` Smalltalk
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

```smalltalk
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
t value: (indirectTempsCopy at: 1)
t 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 I 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: #()
```
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']`

```smalltalk
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);
pushLiteral: 2;
storeInstVar: 1;
pushTemp: #self;
returnTop;
ir.

"receiver and args"

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

```plaintext
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

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

> super send

```apl
... 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 analysis (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

> Parsers 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'
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 shouldn't:
            [scanner interpretNextInstructionFor: client]
         raise: Error.
      ].
   ].
```
Example: Printing Bytecode

> InstructionPrinter:
  — Print the bytecodes as human readable text

> Example:
  — print the bytecode of Number>>asInteger:

```smalltalk
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:

```smalltalk
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

```small
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:

```plaintext
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:

```plaintext
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:

```plaintext
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/

Attribution-ShareAlike 3.0 Unported

You are free:
  to Share — to copy, distribute and transmit the work
  to Remix — to adapt the work

Under the following conditions:
  Attribution. You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).
  Share Alike. If you alter, transform, or build upon this work, you may distribute the resulting work only under the same, similar or a compatible license.

For any reuse or distribution, you must make clear to others the license terms of this work. The best way to do this is with a link to this web page.

Any of the above conditions can be waived if you get permission from the copyright holder.

Nothing in this license impairs or restricts the author's moral rights.