11. Working with Bytecode
Roadmap

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

Original material by Marcus Denker
Roadmap

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

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

> New compiler for Squeak 3.9
  - http://www.iam.unibe.ch/~scg/Research/NewCompiler/
  - adds support for true block closures (optional)
The Squeak 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

- Scanner / Parser
- Semantic Analysis
- Code Generation

Code generation in detail

- Build IR
- Bytecode Generation

ASTTranslator
IRBuilder

IRTranslator
BytecodeBuilder

© Oscar Nierstrasz
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

```
<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
Calling Parser code
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
RMethodNode
RReturnNode
RSequenceNode
RValueNode
RBArrayNode
RAssignmentNode
RBlockNode
RCascadeNode
RLiteralNode
RMessageNode
ROptimizedNode
RVariableNode
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'.
(tree literals (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
> 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 |
|  | iRMetho... | aCompiledMethod |
| iRMetho... := IRBuilder new |
| numRargs: 1; |
| addTemps: #(self); |
| pushLiteral: 1; |
| returnTop; |
| ir. |
```

```
"receiver and args declarations"
```

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

> The Squeak compiler
> Introduction to Squeak 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 Squeak 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:

<table>
<thead>
<tr>
<th>Header</th>
<th>Number of temps, literals...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Literals</td>
<td>Array of all Literal Objects</td>
</tr>
<tr>
<td>Bytecode</td>
<td></td>
</tr>
<tr>
<td>Trailer</td>
<td>Pointer to Source</td>
</tr>
</tbody>
</table>

```
(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:

```
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 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 <7C> returnTop
  — return the object on top of the stack to the calling method
Squeak 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:

```plaintext
^ 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
```
Roadmap

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

> IRBuilder: A tool for generating bytecode
  > Part of the NewCompiler
  > Squeak 3.9: Install packages AST, NewParser, NewCompiler

> Like an Assembler for Squeak
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']`

```
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

```plaintext
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: #()
```
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
iRMethocl := 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

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

> super send

```ruby
...
builder send: #selector toSuperOf: aClass;
```

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

- The Squeak compiler
- Introduction to Squeak bytecode
- Generating bytecode with IRBuilder
- Parsing and Interpreting bytecode
Parsign and Interpretation

> **First step:** *Parse bytecode*
  > enough for easy analysis, pretty printing, decompilation

> **Second step:** *Interpretation*
  > needed for simulation, complex analysis (e.g., profiling)

> **Squeak 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

```smalltalk
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:

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]
```

"Append to the stream, aStream, a description of each bytecode in the instruction stream."
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

``` Smalltalk
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:

\[(\text{ContextPart runSimulated: } [3 \text{ factorial}])\]
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 Squeak compiler organized?
- What does the Squeak semantic analyzer add to the parser-generated AST?
- What is the format of the intermediate representation?
- What kind of virtual machine does the Squeak 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.