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

Overview

1. Introduction to Squeak Bytecodes
2. Generating Bytecode with IRBuilder
3. Decoding Bytecodes
4. Bytecode Execution

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

- CompiledMethods format:
  - Header
  - Literals
  - Bytecode
  - Trailer

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: 4bit 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:
  ```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 <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 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 Bytecodes**

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

```smalltalk
9 <70> pushConstant: 1
10 <77> pushConstant: 2
11 <82> send: <
12 <99> jumpFalse: 15
13 <20> pushConstant: 'true'
14 <90> jumpTo: 16
15 <73> pushConstant: nil
16 <7C> returnTop
```
What you should have learned...

- ... dealing with bytecodes directly is possible, but very boring.
- We want reusable abstractions that hide the details (e.g. the different send bytecodes)
- We would like to have frameworks for
  - Generating bytecode easily
  - Parsing bytecode
  - Evaluating bytecode

Generating Bytecodes

- IRBuilder: A tool for generating bytecode
- Squeak 3.7: IRBuilder has no support for blocks
- New version: Part of the new compiler (on SqueakMap)
- Following slides will use 3.7 IRBuilder
- Like an Assembler for Squeak

IRBuilder: Simple Example

- Number>>asInteger
  
iRMethod := IRBuilder new
  rargs: #(self); "receiver and args"
  pushTemp: #self;
  send: #truncated;
  returnTop;
  ir.
  aCompiledMethod := iRMethod compiledMethod.
  aCompiledMethod valueWithReceiver: 3.5
  arguments: #()
IRBuilder: Temporary Variables

- Accessed by name
- Define with addTemp:/addTemps:
- Read with pushTemp:
- Write with storeTemp:

```
IRMethod := IRBuilder new
  rargs: #(self); "receiver and args"
  addTemps: #(a b);
  pushLiteral: 1;
  storeTemp: #a;
  pushLiteral: 2;
  storeTemp: #b;
  pushTemp: #a;
  returnTop;
```

IRBuilder: Sends

- normal send
  ```
  builder pushLiteral: 'hello'
  builder send: #size;
  ```
- super send
  ```
  ....
  builder send: #selector toSuperOf: aClass;
  ```

IRBuilder: Lessons learned

- IRBuilder: Easy bytecode generation
- Next: Decoding 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)
- 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

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: [
  ].
].

Example: Printing Bytecodes

• InstructionPrinter: Print the bytecodes as human readable text
• Example: print the bytecode of Number>>#asInteger:

String streamContents: [:str |
  (InstructionPrinter on: Number>>#asInteger) printInstructionsOn: str
]
result:
'9 <70> self
10 <D0> send: truncated
11 <7C> returnTop
'

InstructionPrinter

• Class Definition:

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

• 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.
  oldPC := scanner pc.
  [scanner pc <= end] whileTrue: [scanner interpretNextInstructionFor: self]
**InstructionPrinter**

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

```smalltalk
InstructionPrinter>>pushReceiver
'Print the Push Active Context’s Receiver on Top Of Stack bytecode.'
self print: 'self'
```

**Example: InstVarRefLocator**

- Analyse a method, answer true if it references an instance variable
- We implement `CompiledMethod>>#hasInstVarRef`

```smalltalk
hasInstVarRef
| scanner end printer |
scanner := InstructionStream on: self.
printer := InstVarRefLocator new.
end := self endPC.
[scanner pc <= end] whileTrue: [
   (printer interpretNextInstructionUsing: scanner) ifTrue: [^true].
].
^false
```

**Example: Decompiling Bytecode**

- BytecodeDecompiler
  - Uses IRBuilder for building IR (Intermediate Representation)
  - Code for the bytecode `pushReceiver`:

```smalltalk
BytecodeDecompiler>>pushReceiver
irBuilder pushReceiver
```
**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])
  ```

**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]
  ```

**End**

- Short overview of Squeak bytecode
- Introduction to bytecode generation with IRBuilder
- Parsing bytecode with InstructionStream
- Example for interpretation
- Questions?