Exercise 1

Generating Bytecode

Introduction

Generating bytecode can be done with *IRBuilder*. The following exercise uses the IRBuilder of Squeak 3.7.

Here is the the example from the lecture again:

```
irMethod:= IRBuilder new
  rargs: #(self);      "receiver and args"
  pushTemp: #self;
  send: #truncated;
  returnTop;
  ir.

aCompiledMethod  := irMethod compiledMethod.
```

The variable *aCompiledMethod* now contains the generated compiled method. This method can be executed:

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

1.1 Expressions

With the help of IRBuilder, generate a method that calculates the expression `(3 + 4) factorial` and returns the result.

1.2 Parameters

Change the code that you wrote for the first exercise to use parameters instead of hard coded numbers. So in the end you will have a method that requires two arguments. If executed with

```
aCompiledMethod valueWithReceiver: nil arguments: #(3 4)
```

the result should be 7.

1.3 Loops

The Squeak bytecode has support for jumps. Jumps are used to implement conditionals and loops in an efficient way.

Generate a compiledMethod with `IRBuilder` that outputs the numbers 1 to 10 on the Transcript window.

1.4 Instance Variables

Generate a method that adds two instance variables and returns the result. Test the code by running it on a Point, e.g., 3@4.

1.5 Installing a Method in a Class

Find a way to add the method from Exercise 4 to the class Point with the name `returnSum`. After that, the following test should be green:

```
testReturnAdd
   self assert: (1@2) returnSum = 3.
   self assert: (3@4) returnSum = 7.
```

Exercise 2

Bytecode Analysis

2.1 Counting Number of Executed Bytecodes

Look at the method tallyInstructions: in the class ContextPart (class-side):

```
"This method uses the simulator to count the number of occurrences of
each of the Smalltalk instructions executed during evaluation of aBlock.
Results appear in order of the byteCode set."
| tallies |
tallies := Bag new.
thisContext sender
runSimulated: aBlock
contextAtEachStep:
[:current | tallies add: current nextByte].
^tallies sortedElements

"ContextPart tallyInstructions: [3.14159 printString]"
^anArray at: 2
```

The method runSimulated: aBlock contextAtEachStep: [:current— ...] execute aBlock and for each bytecode executed in this block or in called methods, the second argument is evaluated with an instance of one of the subclasses of ContextPart as the argument.

Write a similar method named numberOfBytecodeExecuted: aBlock that returns the number of bytecode executed when evaluating the provided block. For instance:

```
ContextPart numberOfBytecodeExecuted: [3.14159 printString]
==> 1029
```

In total, the expression 3.14159 printString is evaluated by executing 1029 bytecodes.

2.2 Methods Coverage Analysis

Getting information about methods that are currently needed to perform a computation is often difficult to obtain with languages like Java. However this information can easily retrieved in Smalltalk.

Number of Methods Invoked

Create a method numberOfInvokedMethods: aBlock that return the number of all the methods invoked when aBlock is evaluated.

```
ContextPart numberOfInvokedMethods: [3.14159 printString]
==> 38
```

Set of Methods Covered

We are now interested in the methods name.

```
ContextPart methodCovered: [3.14159 printString]
==> #('ContextPart class>>DoIt' 'Object>>printString'
 'Object>>printStringLimitedTo:' ... )
```

Bytecode Covered

Let's focus on bytecode. When a method is invoked, not all the bytecode contained in this method are executed. For instance, when executing 3.14159 printString the method on: defined in the class WriteStream is executed, but only 90% of its bytecode are executed.

```
ContextPart bytecodeCovered: [3.14159 printString]
==>  #(#('WriteStream>>on:' 90) #('LimitedWriteStream>>nextPut:' 69)
#('Object>>species' 100) ...)
```