CS322 Winter'14 Lab 5 Handout
Stack IRs and Stack Code Generation

Learning Objectives

Upon successful completion, students will be able to:

- understand and write simple stack IR programs;
- implement an stack code generator for simple expressions and statements.

1. Stack IR Code

Stack machine code is a form of intermediate code. It assumes the presence of an operand stack. Most operations take their operands from the stack and push their results back onto the stack. For example, an integer subtract operation would remove the top two elements from the stack and push their difference onto the stack. Neither the operands nor the result need be referenced explicitly in the subtract instruction. In fact, only a few instructions (e.g. `push` and `pop`) need to reference a *single* operand explicitly. Comparing to register-machine based IR code (*e.g.* three-address code), stack IR code has the advantage of being simple, compact, and easy to interpret.

In this lab, we'll use Java bytecode and a simpler stack code called SC0 as examples to study this form of IR, and practice code-generation for it.

2. A Brief Look at Java Bytecode

Go to the `prog` sub-directory. Take a look inside the two Java programs, `Length.java` and `Max.java`. You'll see familiar code — they implement the array-length and array-max-element functions that you programmed before (in x86-64 assembly and in our IR1 language).

Now compile them with `javac`, then disassemble the `.class` programs with `javap`:

```
linux> javac Length.java Max.java
linux> javap -c Length.class > Length.bcode
linux> javap -c Max.class > Max.bcode
```

The two resulting files, `Length.bcode` and `Max.bcode`, contain instruction listings of Java bytecode for these two programs. Your first exercise is to study these instruction listings and try to break them into groups such that each group corresponds to one statement in the original Java program. The Lab instructor will help you with a brief introduction on the Java bytecode instructions. But you could also pull off a Java bytecode instruction listing yourself from the Internet to use.

3. Stack Code Programming

Although we could edit and program Java bytecode directly (there are tools available), we choose to program in a much simpler stack code. Our stack-based IR language is called SC0.

SC0 is based on a standard stack machine model. It operates with an implicit operand stack and a storage array for variables. About half of SC0 instructions do not have explicit operands; the other half take a single integer operand. SC0's instructions are listed in the table below. The first column lists the instructions; the second column explains them; and the third column shows the stack content differences before and after instructions.

```
Stack Code (SC0):

Instruction  Sematics                          Stack (bottom<-->top)
-------------------------------------------------------------------
CONST n      load constant n to stack          -> n
LOAD  n      load var[n] to stack              -> val
STORE n      store val to var[n]               val ->
ALOAD        load array element                arrayref, idx -> val
ASTORE       store val to array element        arrayref, idx, val ->
NEWARRAY     allocate new array                count -> arrayref
PRINT        print val                         val ->
NEG          - val                             val -> result
ADD          val1 + val2                       val1, val2 -> result
SUB          val1 - val2                       val1, val2 -> result
MUL          val1 * val2                       val1, val2 -> result
DIV          val1 / val2                       val1, val2 -> result
AND          val1 & val2                       val1, val2 -> result
OR           val1 | val2                       val1, val2 -> result
GOTO n       pc = pc + n
IFZ  n       if (val==0)     pc = pc + n       val ->
IFEQ n       if (val1==val2) pc = pc + n       val1, val2 ->
IFNE n       if (val1==val2) pc = pc + n       val1, val2 ->
IFLT n       if (val1==val2) pc = pc + n       val1, val2 ->
IFLE n       if (val1==val2) pc = pc + n       val1, val2 ->
IFGT n       if (val1==val2) pc = pc + n       val1, val2 ->
IFGE n       if (val1==val2) pc = pc + n       val1, val2 ->
-------------------------------------------------------------------
```

Note: For the jump instructions, the operand `n` represends the *relative* displacement from the the current instruction position. `n` can be either positive or negative.

Your task is to write two programs in this language, `Length.sc` and `Max.sc`, corresponding to the Java version. (*Hint:* It maybe easier to convert the Java bytecode version to SC0 code than to write from scratch.) You can verify your `SC0` programs with the provided interpreter:

```
linux> java -jar sc0int.jar Length.sc
```

4. Stack Code Generation

Now we are switching to IR codegen. We'd like to see how to write a SC0 code generator for our AST0 language. Go to the `scgen` sub-directory. You'll see a program file, `SC0Gen.java`. This is the file you'll be working on. Inside you'll see a setup similar to the IR0 code generator that you worked on in Lab 4. The `main` method reads in an AST0 program and invokes the `gen` method on the top-level AST0 node. The rest of the program is a collection of overloaded `gen` methods, one for each type of AST0 nodes.

There are two major differences comparing to `IR0Gen.java`. First, since SC0 instructions are very simple, there is no need to define a family of classes to represent them. We therefore represent SC0 instructions simply by Strings. A SC0 program hence is represented by a list of Strings.

Second, since stack code uses a stack to keep all operands, there is no need to keep track where sub-expression's value is kept at. Therefore there is no need to use `ValPack` to handle expression methods' return value. In `SC0Gen`, all `gen` methods return a list of Strings.

Your task is to complete the `gen` method for each AST0 node. After finishing coding, you can compile and test your `SC0Gen` program with some test `AST0` programs in the `tst` sub-directory. You may also run the generated programs with the provided SC0 interpreter.