CS322 Winter'14 Assignment 4: IR Code Generation (II)

(Due Tuesday 2/25/14 @ 12pm)

This assignment is a follow-up to last week's lab (Lab 5). Your task is to complete the stack IR code generator, `SC0Gen.java`, for the input language Ast0 and output language SC0. Both languages' definitions are on the next page.

Overview

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 generate and implement (by an interpreter).

Strategy for doing this assignment is simple. First, you want to get familiar with the stack code SC0. Lab 5 has started you on it, but you may want to study it a little more. Next, you should prepare a template for each type of the Ast0 nodes, outlining the structure of the target SC0 code. Once you have the templates, the actual programming should be straightforward, since there is very little code to generate comparing to three-address-style IR code.

One area you may encounter some difficulty is debugging SC0 code. It is harder to trace a program when operands are implicit. The provided SC0 interpreter has a program-tracing feature turned on. When you run the interpreter on a SC0 program, you'll see every step of the interpreter's actions. You should take advantage of the tracing information in your debugging. (*Note:* When you use the script `runsc` on a SC0 program, the tracing information is suppressed.)

Code Organization

Download a copy of `hw4.zip` from the D2L website. After unzipping, you should see an `hw4` directory with the following items:

- `ast0` — a directory containing AST node definitions
- `SC0Gen.java0` — a starting version of SC0Gen
- `sc0int.jar` — a SC0 interpreter
- `Makefile` — for compiling your program
- `tst` — a directory containing sample Ast0 programs and their expected output
- `gensc` — a script for invoking your SC0Gen on Ast0 test files
- `runsc` — a script for running SC0 programs with an interpreter

Note that except for `test1` and `test2`, there is no reference SC0 programs. This is intentional. It encourages you to come up with the right templates yourself, and to use the interpreter to debug your program.

Requirements and Grading

Your SC0 programs should run successfully with the provided interpreter and generate matching output to those in `.out.ref` files. This assignment will be graded mostly on your `SC0Gen` program's correctness. We may use additional programs to test. We'll also check the internals of your program. The minimum requirement for receiving a non-F grade is that your `SC0Gen.java` program compiles without error, and it generates validate SC0 code for at least one simple Ast0 program.

What to Turn in

Submit a single file, `SC0Gen.java`, through the "Dropbox" on the D2L class website.

The SC0 Stack Code Definition

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 have no 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.

```
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` represents the *relative* displacement from the the current instruction position. `n` can be either positive or negative.

The Ast0 Language

This version of Ast0 is almost identical to the one used in Lab 4, except that we added to `Stmt` the `Block` node to allow statement blocks.

```
Program -> {Stmt}

Stmt -> "{" {Stmt} "}"
     |  "Assign" Exp Exp
     |  "If" Exp Stmt ["Else" Stmt]
     |  "While" Exp Stmt
     |  "Print" Exp

Exp  -> "(" "Binop" BOP Exp Exp ")"
     |  "(" "Unop" UOP Exp ")"
     |  "(" "NewArray" <IntLit> ")"
     |  "(" "ArrayElm" Exp Exp ")"
     |  <Id>
     |  <IntLit>
     |  <BoolLit>

BOP  -> "+" | "-" | "*" | "/" | "&&" | "||" | "==" | "!=" | "<" | "<=" | ">" | ">="
UOP  -> "-" | "!"
```