CS 316 Fall 2009
All projects in this course must be completed **individually and independently**. All programs must be written in Java or C++.

PROJECT 4: Top-Down Parser with Instruction Emission
Due: 12/13/09, Sunday, Midnight

Late submissions will be accepted until **12/18/09** with penalties of one letter grade reduction per day. For example, a submission on 12/16/09 would get a reduction of three letter grades (A would be reduced to B, etc.).

This project is a continuation of Project 3. You will modify the Project 3 top-down parser to emit stack-based virtual machine instructions.

You've been working on a small pure functional language with the data types *int*, *float*, and *boolean*. It has no assignment statement. Function definitions take the form:

funName ($p_1 \cdots p_n$) { Exp } $\quad n \geq 0$

The p_i are the formal parameters, and Exp is the function body expression. If n = 0, the function has no parameters.

The conditional expression takes the form:

if (expr) Exp_1 else Exp_2

Its structured operational semantics is

if Eval(expr) = true **then** Eval(Exp_1) **else** Eval(Exp_2)

Here the program state is implicitly given by the values of the parameters and never changes throughout evaluation of the function body expression. So we can omit it in the definition of Eval. This is a characteristic of purely functional languages without assignment statements. Note that the conditional expression itself is an expression that has a value. In pure functional languages all computation is done by expression evaluations and nothing else.

The following table summarizes the operators, the number of arguments, and the corresponding virtual machine instructions.

operator	# of arguments	VM instruction
+	2	add
−	2	sub
*	2	mul
/	2	div
−	1	neg
||	2	or
&&	2	and
!	1	inv
<	2	lt
<=	2	le
>	2	gt
>=	2	ge
==	2	eq
!=	2	neq

The following VM instructions will also be used.

instruction	semantics
push *const*	push constant literal *const* onto operand stack
push #*i*	push value of *i*-th function formal parameter onto operand stack
if_f goto *label*	pop the top of operand stack; if its value is false, goto *label*
goto *label*	unconditionally goto *label*
call *L*, *i*	call the function starting at label *L* with # of parameters given in *i*
return	return from function call

The *i*-th formal parameter variable of each function must be translated into "#*i*" to be used as operand of the *push* instruction.

The *pop* instruction is absent due to the absence of assignment statement.

The *push* instruction and the instructions for the 14 operators will be emitted using the methods described in class. The following intermediate-code operational

semantics should be used to compile the conditional expression and the function call expression:

if (expr) Exp_1 else Exp_2

```
        code to evaluate expr  // puts boolean value of expr at top of operand stack
        if_f goto Else
        code to evaluate Exp1
        goto Out
Else: code to evaluate Exp2
Out:
```

funName($expr_1$ ⋯ $expr_n$) // funName is user-defined function

```
code to evaluate expr1 // puts value of expr1 in operand stack
...
code to evaluate exprn // puts value of exprn in operand stack
call L, n  // L is the start label of code for funName
```

The VM uses a single operand stack shared by all function calls. The *call L, n* instruction assumes that the *n* actual parameter values are placed in the top *n* elements of the operand stack, and automatically handles the creation of a new activation record. The *return* instruction pops the *n* actual parameter values from the operand stack, pushes the function return value, and jumps to the return address.

Your program will need to use some type of mapping or association table to remember the start label and the sequential numbers of formal parameters of each function. (I've used java.util.HashMap for this purpose.) The program will also need to count the number of actual parameters of each function call to determine the value of *n* in "*call L, n*" instruction.

For simplicity we assume that all user-defined function calls are only made to function definitions that textually precede them. (Real-world compilers use a 2nd pass or some kind of function call table to resolve the unknown start labels of forward function calls.)

Positive integers must be used for labels. Integer labels may appear in any order, but must each appear at a unique point in the compiled instruction stream. A global integer variable, initially set to 0, can be used to generate labels; simply increment this variable by 1 to generate a new label. In this project there is no need to remove redundant uses of labels, which would be the task of the optimizer.

The expression to be evaluated is prefixed by "/" and appended after all the function definitions. The syntax for <fun defs> is slightly modified to this effect:

<fun defs> → {<fun def>}$^+$ "/" <expr>

The entire instruction stream should start with a goto instruction jumping to the start of this expression – see the sample files.

You may modify the sample Project 3 program that will be posted on 11/25/09.

Here's a sample set of test input files:

in1 | out1
in2 | out2
in3 | out3
in4 | out4
in5 | out5

You should make your own additional input files to test the program.

Submission

Your source program, including the lexical analyzer, must be emailed to yukawa@cs.qc.cuny.edu with the subject header:

CS 316, Project 4, your full name

The due date is 12/13/09, Sunday, Midnight. Please include the program as attachment **without compression into .zip, .rar, etc**.