Warning: Their is not much coding, but it is all very fussy stuff.

Part I: Extending the VM

Problem 1: (12 points).

(a) (6 points) Extend the VM (in the step function) to add two new instructions: Inc and Dec. They have the operational semantics:

\[
\begin{align*}
& \text{Inc: } \quad \text{obj } \vdash (pc, sp, stk, regs) \Rightarrow (pc + 1, sp, stk[(sp - 1) \mapsto v], regs) \quad (*) \\
& \text{Dec: } \quad \text{obj } \vdash (pc, sp, stk, regs) \Rightarrow (pc + 1, sp, stk[(sp - 1) \mapsto v], regs) \quad (\dagger)
\end{align*}
\]

(*): \( \text{obj}[pc] = \text{inc} \) and \( v = (\text{stk}[sp - 1] + 1) \mod 256 \)

(\dagger): \( \text{obj}[pc] = \text{inc} \) and \( v = (\text{stk}[sp - 1] - 1) \mod 256 \)

(b) (6 points) Use the incTest and decTest functions (in LCvm.hs) to test your implementation. Add your own tests.

Problem 2: (12 points).

(a) (6 points) Extend the VM (in the step function) to add the instruction Dup which duplicates the value at the top of the stack. (E.g., if the stack (from bottom to top) is [10,20,30], then a Dup changes it to [10,20,30,30].) Dup has the following (small-step) operational semantics.

\[
\begin{align*}
& \text{Dup: } \quad \text{obj } \vdash (pc, sp, stk, regs) \Rightarrow (pc + 1, sp', stk', regs) \quad (\S)
\end{align*}
\]

(\S): \( \text{obj}[pc] = \text{dup} \) and

\( \Rightarrow \) if \( sp = 0 \), then \( sp' = 0 \) and \( stk' = stk \); and

\( \Rightarrow \) if \( sp \neq 0 \) and \( top = stk[sp - 1] \), then \( sp' = sp + 1 \) and \( stk' = stk[sp \mapsto top] \).

(b) (6 points) Use the dupTest function (in LCvm.hs) to test your implementation. Add your own tests.

Problem 3: (14 points).

(a) (8 points) Extend the VM (in the step function) to add the Call and Ret instructions, which give us a very simple-minded procedure call mechanism:

Call \( addr \) does a simple subroutine call by pushing onto the stack the address of the next instruction after the Call and then jumping to the instruction at address addr.

Ret returns from a subroutine call by grabbing the top of the stack top, popping the stack, and jumping to the instruction with address top. 

IMPORTANT: Unlike Jmp, Jz, and Jnz, the addresses here are absolute, not relative. (Question: How might Ret cause security problems?)

Formally, they have the following (small-step) operational semantics.

\[
\begin{align*}
& \text{Call: } \quad \text{obj } \vdash (pc, sp, stk, regs) \Rightarrow (arg, sp + 1, stk[sp \mapsto (pc + 2)], regs) \quad (\I) \\
& \text{Ret: } \quad \text{obj } \vdash (pc, sp, stk, regs) \Rightarrow (top, sp - 1, stk, regs) \quad (\ast)
\end{align*}
\]

(\I): \( \text{obj}[pc] = \text{call} \) and \( \text{arg} = \text{obj}[pc + 1] \)

(\ast): \( \text{obj}[pc] = \text{ret} \) and \( \text{top} = stk[sp - 1] \)

(b) (3 points) Use the function callRetTest (in LCvm.hs) to test your implementation. The expected results are described in LCvm.hs.

(c) (3 points) For another test, run \( \text{stepRun fact4'} \) which is another assembly program (using Dup, Call, and Ret) that computes 4!.

The expected results are described in LCvm.hs.
Problem 4: (12 points)

**BACKGROUND.** There is a stack underflow in a step of the VM when the stack pointer starts at address 0 and changes to address 255. Similarly, there is a stack overflow in a step of the VM when the stack pointer starts at address 255 and changes to address 0. (One might argue with the later definition, but it will do for this problem.)

**YOUR PROBLEM.** Define a function

\[
\text{underOver} :: [\text{AsmCode}] \rightarrow ([\text{Int}],[\text{Int}])
\]

that given an assembly program \( \text{asm} \) returns a pair of lists (unders,overs), where unders is a list of the step numbers in which there is a stack underflow and overs is a list of the step numbers in which there is a stack overflow. For example,

\[
\text{underOver} [\text{Pop,Push 1,Pop,Push 1}] \sim ([1,3],[2,4])
\]

To test your function, use the example above and come up with your own examples. *Hint:* Look at showRun in Lcvm.hs and use some of its ideas (but not the I/O stuff). Also, as usual, list comprehension is your friend.

Part II: Extending the compiler

Problem 5: (20 points)

(a) (10 points) Implement the \( \text{Not}, \text{LEQ}, \text{and GEQ} \) cases of \( \text{transB} \) in the compiler. (\( \text{Ilt} \) is useful for each of these.)

(b) (10 points) Use the functions \( \text{notTest1}, \text{notTest2}, \text{leqTest}, \) and \( \text{geqTest} \) in \( \text{LVcompiler.hs} \) in testing your implementations. Add your own tests.

Problem 6: (30 points)

(a) (15 points) The \textit{for} command has the following big-step operational semantics.

\[
\text{For}_1:\quad \langle 0 \text{do } C, s_0 \rangle \downarrow \langle \text{skip}, s_0 \rangle
\]

\[
\text{For}_2:\quad \langle C, s_0 \rangle \downarrow \langle \text{skip}, s_1 \rangle\quad \langle \text{for } \ell_i \text{ do } C, s_2 \rangle \downarrow \langle \text{skip}, s_3 \rangle\quad \langle \text{for } \ell_i \text{ do } C, s_0 \rangle \downarrow \langle \text{skip}, s_1 \rangle
\]

\( \text{For}_1: \) \( s_0(\ell_i) = 0, \)

\( \text{For}_2: \) \( s_0(\ell_i) \neq 0, s_1(\ell_i) = v, \) and \( s_2 = s_1[\ell_i \mapsto v - 1] \)

In terms of the \textit{while-do} command:

\[
\text{for } \ell_i \text{ do } C \equiv \text{while } \ell_i \neq 0 \text{ do } \{ C; \ell_i := !\ell_i - 1 \}
\]

Extend the compiler (in the \textit{transC} function) to handle \textit{for} commands.

(b) (15 points) Among your tests you should run:

(i) \( \text{clg c9 } k \)

for various small integers \( k, \) e.g., \( \text{clg c9 2} \). The final configuration should have an empty stack and all registers containing 0.

(ii) \( \text{clg c10} \)

\( \text{c10} \) is yet another 4! computation. The final configuration should have an empty stack, \( x1=24, \) and all other registers containing 0.

(iii) Two original tests of your own. Explain why the final configurations are the right ones.

Notes:

- There are multiple ways to approach this problem.
- The \textbf{Dec} command from Problem 1 may be useful.
- Traces can be useful for correctly figuring jump lengths.
- But do \textbf{not} use \textbf{Call} or \textbf{Ret} here; that is the wrong track.
Challenge Problems

Writing Problem 1: (20 points). Write an assembly program for our VM that (i) takes a number, $n$, in register 0, (ii) computes $\sum_{j=1}^{n} j^2$, and (iii) leaves this number in register 1. Moreover, it should use a Call/Ret procedure that takes a number in register 2 and replaces that number with its square. Test your program with $n = 0, 1, \ldots , 8$. Why do you get a funny answer for $n = 9$?

Administrivia

This assignment is a solo effort: NO TEAMS!

- If you trade ideas with another student, document it in your source file.
- Turn in your assignment via Blackboard. Include (i) the source files, (ii) the transcripts of test runs, and (iii) the cover sheet.