Homework 7: Extending the VM and Compiler

CIS 352: Programming Languages

1 March 2018, Version 1

Administrivia

- When you trade ideas with another student, *document it* in your cover sheet.
- For Part I problems, do your work in `LCvm.hs`.
- For Part II problems, do your work in `LCcompiler.hs`.
- Turn in your assignment via Blackboard. Include *(i)* the source files, *(ii)* the transcripts of test runs, and *(iii)* the cover sheet.

Grading Criteria

- The homework is out of 100 points.
- Unless otherwise stated, each problem is $\approx$ 70% correctness and $\approx$ 30% testing.
- Omitting your name(s) in the source code looses you 5 points.

> **Warning:** There is not much coding, but it is all very fussy.

Part I: Extending the VM

❖ ***Problem 1 (12 points)*** ❖

(a) ***(6 points)*** Extend the VM[1] to add two new instructions: **`Inc`** and **`Dec`**. They have the operational semantics:

[1] in the `step` function in `LCvm.hs`

(side-conditions)

$$\textit{Inc:}\quad \frac{}{\text{obj} \vdash (pc, sp, stk, regs) \Rightarrow (pc+1, sp, stk[(sp-1) \mapsto v], regs)} \quad \begin{pmatrix} obj[pc] = inc \text{ and} \\ v = (stk[sp-1]+1) \bmod 256 \end{pmatrix}$$

$$\textit{Dec:}\quad \frac{}{\text{obj} \vdash (pc, sp, stk, regs) \Rightarrow (pc+1, sp, stk[(sp-1) \mapsto v], regs)} \quad \begin{pmatrix} obj[pc] = dec \text{ and} \\ v = (stk[sp-1]-1) \bmod 256 \end{pmatrix}$$

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

Note: Arithmetic on numbers of type `Word8` is automatically mod256. E.g., try:

```
ghci> let a = 150 ::  Word8
ghci> let b = 150 ::  Word8
ghci> a + b
44
```

❖ ***Problem 2 (12 points)*** ❖

(a) ***(6 points)*** Extend the VM[2] 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.

[2] in the `step` function in `LCvm.hs`

$$\textit{Dup:}\quad \frac{}{\text{obj} \vdash (pc, sp, stk, regs) \Rightarrow (pc+1, sp', stk', regs)}$$

$\Big($ $obj[pc] = dup$ and
- if $sp = 0$, then $sp' = 0$ & $stk' = stk$; and
- if $sp \neq 0$ and $top = stk[sp-1]$, then $sp' = sp+1$ & $stk' = stk[sp \mapsto top]$. $\Big)$

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

❖ ***Problem 3 (10 points)*** ❖

(a) ***(5 points)*** Extend the VM[3] to add the instruction **`Ni`**. It has the operational semantics:

[3] in the `step` function in `LCvm.hs`

$$\textit{Ni::}\ \frac{}{\text{obj} \vdash (pc, sp, stk, regs) \Rightarrow (pc+1, sp, stk[(sp-1) \mapsto v], regs)}$$

$$\left(\begin{array}{l} obj[pc] = ni \quad \text{and} \\ v = \begin{cases} 1, & \text{if } stk[sp-1] = 0; \\ 0, & \text{if } stk[sp-1] \neq 0 \end{cases} \end{array} \right)$$

(b) ***(5 points)*** Use the `niTest` function (in `LCvm.hs`) to test your implementation. Add your own tests.

"Ni" is Welsh for "not", at least according to `http://www.geiriadur.net`. But also see: `http://en.wikipedia.org/wiki/Knights_who_say_Ni`.

❖ ***Problem 4 (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, poping the stack, and jumping to the instruction with address top.

IMPORTANT: Unlike **`Jmp`**, **`Jz`**, and **`Jnz`**, the addresses here are *absolute*, not relative.[4]

[4] See: `https://en.wikipedia.org/wiki/Addressing_mode#Simple_addressing_modes_for_code`

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

$$\textit{Call:}\ \frac{}{\text{obj} \vdash (pc, sp, stk, regs) \Rightarrow (arg, sp+1, stk[sp \mapsto (pc+2)], regs)}$$

$$\left(obj[pc] = call \text{ and } arg = obj[pc+1] \right)$$

$$\textit{Ret:}\ \frac{}{\text{obj} \vdash (pc, sp, stk, regs) \Rightarrow (top, sp-1, stk, regs)}$$

$$\left(obj[pc] = ret \text{ and } top = stk[sp-1] \right)$$

(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 `(stepRun fact4')` which is another assembly program (using **`Dup`**, **`Call`**, and **`Ret`**) that computes 4!. The expected results are described in `LCvm.hs`.

Side Question: How might the absolute addresses of **`Ret`** cause security problems?

Part II: Extending the compiler

❖ ***Problem 5 (20 points)*** ❖

(a) ***(10 points)*** Implement the `Not`, `LEQ`, and `GEQ` cases of `transB`[5]. (**`Ni`** can be useful for each of these.)

[5] In `LCcompiler.hs`

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

❖ ***Problem 6 (30 points)*** ❖

BACKGROUND: The `do-whilst` command has the following big-step operational semantics.

Recall that `tt` and `ff` are the LC constants for true and false.

$$\textit{DoWhilst}_1\text{:}\quad \frac{\langle C, s_0\rangle \Downarrow \langle \texttt{skip}, s_1\rangle \qquad \langle B, s_1\rangle \Downarrow \langle \texttt{ff}, s_2\rangle}{\langle \texttt{do}\ C\ \texttt{whilst}\ B, s_0\rangle \Downarrow \langle \texttt{skip}, s_2\rangle}$$

$$\textit{DoWhilst}_2\text{:}\quad \frac{\langle C, s_0\rangle \Downarrow \langle \texttt{skip}, s_1\rangle \qquad \langle B, s_1\rangle \Downarrow \langle \texttt{tt}, s_2\rangle \qquad \langle \texttt{do}\ C\ \texttt{whilst}\ B, s_2\rangle \Downarrow \langle \texttt{skip}, s_3\rangle}{\langle \texttt{do}\ C\ \texttt{whilst}\ B, s_0\rangle \Downarrow \langle \texttt{skip}, s_3\rangle}$$

Do-whilst is a version of the do-while construction from C.

YOUR PROBLEM:

(a) ***(15 points)*** Extend the `transC` function[6] to handle `repeat` commands.

[6] in `LCcompiler.hs`

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

(i) `(clg c9)`
The final configuration should have an empty stack and all the registers should be 0.

(ii) `(clg c10)`
`c10` is yet another 4! computation. The final configuration should have an empty stack, `x1`=24, and all other registers 0.

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

Notes:

- The problem has multiple approaches.
- Traces can be useful for correctly figuring jump lengths.
- The **`Ni`** instruction might be useful, but it is a really bad idea to try to use use **`Call`** or **`Ret`** here.

Challenge Problems

✪ ***Challenge 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 $n = 0, 1, 3, 8$, and 9. Why do you get a funny answer for $n = 9$?

Recall the formula $1^2 + 2^2 + 3^2 + \cdots + n^2 = \frac{n(n+1)(2n+1)}{6}$.