Prof. Jingke Li (FAB120-06, lij@pdx.edu), Class: TR 12:00-13:15 @ KMC 185; Office Hour: TR 11:00-11:55.

Assignment 4: More on Imperative Languages

(Due Thursday 10/25/18)

This assignment is a follow up on Assignment 3.

1. EL1 Compiler

Recall the simple imperative expression language EL1:

```
Expr -> Int
     |  String
     |  (+ Expr Expr)
     |  (- Expr Expr)
     |  (* Expr Expr)
     |  (/ Expr Expr)
     |  (% Expr Expr)
     |  (<= Expr Expr)
     |  (:= String Expr)
     |  (while Expr Expr)
     |  (if Expr Expr Expr)
     |  (write Expr)
     |  (seq Expr Expr)
     |  (skip)
     |  (for String Expr Expr Expr)
```

This time, you are going to implement a compiler for it.

Target Machine The target machine, Machine1, is a stack machine with the following instructions:

Instruction	Semantics
Const n	load constant n to stack
Load x	load `vars[`x`]` to stack
Store x	store val to `vars[`x`]`
Plus	$val_1 + val_2$, push result to stack
Times	$val_1 * val_2$, push result to stack
Divrem	$val_1 \,/\, val_2$, push div and rem results to stack
Lessequ	$val_1 \leq val_2$, push 1 or 0 to stack
Pop	pop off val
Dup	replicate val
Swap	swap val_1 and val_2
Print	print val
Label i	nop, marking a label position
Branch i	branch to Label i
Branchz i	if $val = 0$ branch to Label i

Note: `vars[]` is an auxiliary array mapping variables to values; val and val_2 represent the top element of the stack, while val_1 represents the second-to-the-top element; i is an instruction attribute, not an operand.

This machine is implemented in the program file, `Machine1.scala`.

Your Tasks

1. For each of EL1's expressions*, provide an informal semantics in the form of a Machine1 code sequence. The sequence may contain actual Machine1 instructions, as well as symbolic names for representing components' sub-sequences. For example, for the expression `(+ e1 e2)`, the code sequence should be:

   ```
   e1.code + e2.code + "Plus"
   ```

   where `e1.code` and `e2.code` are two symbolic names representing the code sequences for `e1` and `e2`, respectively; the `+` operator is for connecting the pieces. Note that the order of the components are important, since `e1` and `e2` may have side effects.

   If a result or partial result of an expression is not needed, it should be popped off from the operand stack. At the end of program execution, the operand stack should be empty. (*Hint:* Think about the `seq` and `divrem` expressions.)

   Save your results in a file, `EL1semantics.txt` (include your name in it).

2. The file `Comp1.scala` contains a skeleton of an EL1 compiler. Complete the implementation.

   Hints:

   - The informal semantics from the previous part can be used as guiding templates for the implementation.
   - The provided `newLabel()` function can be useful in compiling `while` and `if` expressions; it returns a unique label number each time it is called.

A test program, `TestComp1.scala`, is provided. It contains the same set of tests as in `TestInterp1.scala` of the last assignment. Run your compiler with these tests and make sure they all pass.

***Extra Credit** A correct code sequence for the expression `(for x e1 e2 e3)` is not easy to define. You'll earn a small extra credit for getting the sequence right, and for implementing it correctly in the compiler.

2. Tail Recursion

The program file `recursion.c` contains two versions of a factorial function, as well as two other recursive functions, `g()` and `h()`, all from this week's lectures. The program is written in GNU C, which supports nested functions.

Your Tasks

1. Following the factorial function's example, create a tail-recursive version for both `g` and `h()`. Make and keep your changes in the same program file, `recursion.c`. Compile and test the program to make sure the two tail-recursive functions work correctly and produce the same results as their regular-version counterparts.

2. As mentioned in class, advanced compilers can perform optimizations on tail-recursive functions. Run `gcc` with two different optimization settings, `-O1` and `-O2`, and save the assembly code in two separate files, `recursion.s1` and `recursion.s2`. You can achieve this by executing the following commands (on the CS linuxlab system):

   ```
   linux> gcc -O1 -S recursion.c -o recursion.s1
   linux> gcc -O2 -S recursion.c -o recursion.s2
   ```

   Inspect the two assembly programs, and answer the following questions:

   (a) Are all six functions still in recursive form in `recursion.s1`?

   (b) Are all six functions still in recursive form in `recursion.s2`?

Note that each of the tail-recursive functions uses a `helper()` function; you need to carefully link them to their parent function.

Provide a short discussion (one or two paragraphs) based on your observations. Assembly all your answers in a file, `recursion.txt` (include your name in it).

3. Submission

Zip the following four files, `EL1semantics.txt`, `Comp1.scala`, modified `recursion.c`, and `recursion.txt`, into a single zip file, and submit it through the "Assignment 4" submission folder on the D2L class website (under the "Activities/Assignments" tab). Keep your original files untouched in case there is a need to show their timestamp.