Functional Programming – Series 5

In this series of exercises we will work on instruction sets and code generation.

Exercise 1. Code Generation.
Download from Blackboard the file

```
Core.hs
```

This file contains a very elementary processor (*core*) which may simulate the execution of a program written as a list of instructions. The instructions, as well as the "opcodes" for the arithmetical unit (*alu*), are defined as *algebraic types* (i.e., as "embedded languages").

Furthermore, the file `Core.hs` contains:

– the function *alu* that executes the arithmetical operations,

– the function *core* that executes at every clock cycle one of the instructions from the list *instrs*, and updates the following elements:

  − the stack (*stack*),
  − the program counter (*pc*),
  − the stack pointer (*sp*),

  Note that *pc* refers to that instruction that is executed next, and *sp* points to the first free position "above" the top of the stack (even though on that position there still may be an "old" number).

In addition, the file `Core.hs` contains the definition of the (recursive) algebraic type *Expr* which expresses the tree structure of arithmetical expressions (we don't use the constructors "*Leaf*" and "*Node*" that we saw in practical series 3 and 4 for reasons of readability, and because the type *Expr* has to be extended later on). Note that *Expr* also may be seen as an embedded language, for arithmetical expressions in this case.

To start, evaluate the expression *test* from the file `Core.hs` to see its effect (*PutStr* is a standard *IO*-function in Haskell).

Exercise 2. Write a function

$$codeGen :: Expr \to [Instr]$$

which generates the list of instructions of type *Instr*, that, when processed by the function *core*, yield the result of the expression as the top element of the stack, i.e., as that element of the stack that is indicated by $sp - 1$.

Exercise 3. Import `FPPrac.Trees` and transform the expressions of type *Expr* to trees of type *RoseTree* to show expressions graphically.

Adapt your definition for the extensions that you have to define below, both for the type *Expr*, and for the type *Stmnt* (see below).

Exercise 4. The stack represents a piece of memory that (by means of the instructions) only is accessible at the top. Add a second memory

$$heap :: [Int]$$

that is accessible at any address directly. Thus, the second argument of the function *core* now becomes:

$$(pc, sp, heap, stack)$$

Rename the existing *Push*-instruction to *PushConst*, and extend the instruction set *Instr* and the function *core* with the following instructions:

```
PushAddr Int  -- for pushing the value at the indicated address
              -- in the heap to the stack,
Store Int     -- to store the value at the top of the stack
              -- in the heap at the indicated address, at the same
              -- time removing the value from the stack.
```

Adapt and extend the definition of *core* accordingly.

Exercise 5. Add an embedded language for *statements* with initially only one statement for *assignment*:

data *Stmnt* = **Assign** *Variable Expr*

It is upto you whether you define the type *Variable* as an *Int* for the address in the heap to which the variable refers, or as a *String* for the name of the variable. In the latter case you will need to use a lookup table *lut* (say x, y are variables, and a, b are addresses in the heap)

$$lut = [(x, a), (y, b), ...]$$

to bind a variable to the address to which it refers. Clearly, then you will also need a function to determine the address of a variable from this lookup table.

Exercises:

(i) Extend the type *Expr* with a clause for variables,

(ii) Write a function

$$codeGen' :: Stmnt \to [Instr]$$

that generates code to execute the *assign*-statement, such that the value of the expression is stored at the correct address in the heap.

Exercise 6. Define a *type class CodeGen* that contains a single function *codeGen*, such that the types *Expr* and *Stmnt* can be defined as *instances* of this class. Then the same function name *codeGen* will work for both *Expr* and *Stmnt*.

Define also a type class for your functions to transform expressions of types *Expr* and *Stmnt* to rosetrees.

Exercise 7. In this exercise you have to add a statement for *repetitions* to the type *Stmnt*:

Repeat *Expr* [*Stmnt*]	-- the list of statements is executed a number -- of times as indicated by the expression.

To make that possible, extend the instruction set with the following instructions:

PushPC	-- pushes the current program counter on the stack,
EndRep	-- a limited jump-instruction at the end of the body -- of the Repeat-statement: it decreases the value -- of the expression by 1 (on the correct position -- in the stack), and changes the value of the program -- counter to the pc-value that was pushed onto the stack.

Extend the definition of *core* for these instructions, and extend the function *codeGen* for the **Repeat** statement.