CS:3820 Programming Language Concepts
Spring 2019

Homework 3

Due: Friday, Mar 1 by 11:59pm

This is a programming assignment in F#, with some additional non-programming questions. You can do it alone or in a team two people. In the latter case, you are responsible for contacting other students and form a team.

Download the accompanying F# source file `hw3.fsx` and enter your solutions in it where indicated. When you are done, submit it as instructed on Piazza.

Each of your answers *must be free of static errors.* You may receive no credit for problems whose code contains syntax or type errors.

Pay close attention to the specification of each problem and the restrictions imposed on its solution. Solutions ignoring the restrictions may receive only partial credit or no credit at all.

In the following problems *do not* use any mutable variables (i.e., variables declared with `let mutable`) or any mutable predefined types such as references, mutable records, arrays, and so on. Define your functions recursively as needed and take advantage of the `match` operator.

You do not need, and *should not use*, any F# library functions except for the usual integer and Boolean operators, the list operators `::` , `[ ]` , `@`, `List.length` and the conversion function `string`. However, you can use your own auxiliary functions if you prefer.

Do not worry about the efficiency of your solution. Strive instead for cleanness and simplicity. *Unnecessarily complicated code may not receive full credit.*

Do the assigned readings before starting attempting this assignment. If you do not, you might find the assignment exceedingly hard.

Be sure to review the syllabus for details about this course's cheating policy.

1 Processing arithmetic expressions

For this assignment we will consider a language of expressions over integers similar to those seen in class and the previous homework. This language contains C-style Boolean operators that treat 0 as false and any non-zero number as true. The abstract syntax of the language is provided by the following F# types:

```
type operUna = Neg | Not
```

```
type operBin = Add | Mul | Sub | Gt | Eq | And

type expr =
  | Cst of int
  | OpUna of operUna * expr
  | OpBin of operBin * expr * expr
  | IfElse of expr * expr * expr
```

Types operUna and operBin define the operators used in the expression language, where `Neg` is integer negation, `Not` is Boolean negation, `Add` is integer addition, `Mul` is integer multiplication, `Sub` is integer subtraction, `Gt` is integer $>$ comparison, `Eq` is integer equality, and `And` is Boolean conjunction. `Neg` and `Not` are unary operators, all the others are binary.

In type `expr`, constructor `Cst` constructs expressions consisting of an integer constant, `OpUna` constructs the expression obtained by applying a unary operator to another expression, and `OpBin` constructs the expression obtained by applying a binary operator to two other expressions. Finally, `IfElse` constructs conditional expressions as in Homework 1: an expression `IfElse(`e`,` e_1`,` e_2`)` evaluates to the value of e_2 if e evaluates to 0, and evaluates to the value of e_1 otherwise.

We start with a set of warm up problem involving the processing of expressions.

1. Using pattern matching, define a recursive function `drop : int -> 'a list -> 'a list` that takes a non-negative integer n and a list l and drops the first n elements from l, that is, returns a list containing, in the same order, all the elements of l but the first n. The function should raise an error, with `failwith`, if n is greater than the length of the l.

2. Using pattern matching, define a recursive function `size : expr -> int` that returns the *size* of its input expression defined as the number of constructors from type `expr` in the expression.

   For example,
   `size (Cst 4)` is `1`,
   `size (IfElse (Cst 4, OpBin (Add, Cst 1, Cst 2), Cst 9))` is `6`.

3. Using pattern matching, define a recursive function `subexpressions : expr -> expr list` that takes an expression e and returns a list consisting of all the subexpressions of e, excluding e itself. The subexpressions can appear in the list in any order you choose. However, if some subexpression occurs multiple times in e it should occur the same number of times in the output list.

   For example,
   `subexpressions (Cst 4)` is `[]`,
   `subexpressions (IfElse (Cst 4, OpBin (Add, Cst 1, Cst 2), Cst 4))` is
   `[Cst 4; OpBin (Add, Cst 1, Cst 2); Cst 4; Cst 1; Cst 2]`.

2 Compiling expressions to stack machine code

We will consider a stack machine like the one seen in class to evaluate expressions in the language of Part 1. This time, the stack machine has operations encoded by the following F# type:

```
type sInstr = SCst of int | SAdd | SSub | SMul | SNeg
            | SGt | SIfze of int | SJump of int
```

The machine instructions `SCst`, `SAdd`, `SSub`, `SMul`, and `SNeg` have the same behavior as the corresponding instructions discussed in class. The new instructions have the following behavior:

- `SGt` replaces the two topmost elements in the stack with `1` if the element below the top element is strictly greater than the top element; and with `0` otherwise.
- `SIfze` n, where n is a non-negative integer, is a goto instruction that causes the machine to skip the next n instructions if the top of the stack is `0`, and to skip no instructions otherwise. In both cases, it pops the top element from the stack.
- `SJump` n, where n is a non-negative integer, is a goto instruction that causes the machine to skip the next n instructions unconditionally, leaving the stack unchanged.

Write the following functions for compilation to and execution on the stack machine.

1. Using pattern matching, define a recursive function `scomp : expr -> sInstr list` that compiles expressions to stack machine programs similarly to the function `rcomp` seen in class.

   The source language of expressions has operators that do not map directly to machine operations. Part of your job is to figure out, on paper first, how to encode each source language operator in terms of the available machine operations.

   For this problem, you may find it convenient to use the library function `List.length` that takes a list and returns its length (i.e., the number of elements in it).

2. Using pattern matching, define a recursive function `seval : sInstr list -> int list -> int` that executes machine programs similarly to the function `reval` seen in class. The first argument is the list of instructions to execute and the second argument is the stack.

   For this problem, you may find it convenient to use the function `drop` defined in a previous problem.

   Note: For testing purposes, a machine interpreter is already provided in `hw3.fsx` by the function `run` that takes a list of machine instructions and executes them with `seval`, starting with the empty stack.

3. Using pattern matching, define a recursive function `byteCode : sInstr list -> string` that takes a list of stack machine instructions and compiles it into a *byte code program*, a string consisting of numbers separated by a *single* space. Use byte codes `0` to `7`, respectively, for `SCst`, `SAdd`, `SSub`, `SMul`, `SNeg`, `SGt`, `SIfze`, and `SJump`, in that order.

   Make sure not to have any spurious spaces in the output string—for instance, at the beginning or at the end.

   For example,
   `byteCode []` is `""`,
   `byteCode [SCst 10; SCst 2; SAdd; SCst 32; SCst 4; SCst 5; SAdd; SMul; SAdd]` is `"0 10 0 2 1 0 32 0 4 0 5 1 3 1"`.

4. Using pattern matching, define a recursive function `beval : int list -> int list -> int` that takes as input a byte code program (as a list of integers) and a stack (also as a list of integers), and executes that program on the given stack. The function should raise suitable errors if the input program is ill formed in one way or another.

   Your implementation should be such that, for all expressions e,
   `(beval (parse (byteCode (scomp` e`))) [])` returns the same result as
   `(seval (scomp` e`) [])`, where `parse : string -> int list` is a function (provided in `hw3.fsx`) that converts an input byte code program to the list of numbers in it.

   For example,
   `parse "0 10 0 2 1 0 32 0 4 0 5 1 3 1"` is
   `[0; 10; 0; 2; 1; 0; 32; 0; 4; 0; 5; 1; 3; 1]`.

3 Regular Expressions

For this part, write your answers in an F# comment. For compactness, if you need to use a certain subexpression more than once, you are allowed to give it a name and use that name instead. For instance, instead of writing

$$(0+1)11(0+1)^*$$

you are allowed to write something like

$$B11B^* \text{ where } B = 0+1$$

You must use exclusively the regular expression operators ε, *, $+$ and juxtaposition for sequential concatenation.

1. Write a regular expression that generates all and only the words over the alphabet 0, 1 that contain at most two occurrences of 0.

2. Many programming languages use the *exponential notation* for large floating point constants. For instance, `6.022E23` or `6.022e23` which both stand for 6.022×10^{23}, and `1.6e-35` which stands for 1.6×10^{-35}. The notation consists of a numeral[1] followed by a period (.), followed by one or more digits, followed by `E` or `e`, followed possibly by $-$, followed by a numeral.

   Write a regular expression that captures all numbers in exponential notation.[2]

3. Consider the following set of requirements for passwords:

   (a) They can contain only letters and digits.

   (b) They must contain at least one letter and one digit.

   Write a regular expression denoting the set of all passwords that satisfy these requirements. For brevity, use L and D, to denote, respectively, an arbitrary letter and an arbitrary digit.

[1] Recall that a *numeral* is either 0 or a non-empty sequence of decimal digits that does not start with 0.

[2] The actual notation also imposes limits on the number of digits in each part. We are going to ignore this restriction, for simplicity.