CSE 216 – Homework IV

Instructor: Dr. Ritwik Banerjee

This homework document consists of 3 pages. Carefully read the entire document before you start coding. This assignment is entirely about functional programming in OCaml. This is also a much shorter homework (especially since some of the questions are things you have already solved in Java and Python earlier this semester), hence the shorter time until the submission is due.

Note: All functions, unless otherwise specified, should be polymorphic (i.e., they should work with any data type). For example, if you are writing a method that should work for lists, the type must be `'a list`, and not `int list`.

1 Recursion (40 points)

In this section, you may not use any functions available in the OCaml library that already solves all or most of the question. For example, OCaml provides a `List.rev` function, but you may not use that in this section.

1. Write a recursive function `pow`, which takes two integer parameters `x` and `n`, and returns $\mathtt{x}^{\mathtt{n}}$. Also write a function `float_pow`, which does the same thing, but for `x` being a float. (8)

2. Write a function `reverse : 'a list -> 'a list` to reverse a list. (8)

```
# reverse ["a" ; "b" ; "c"];;
- : string list = ["c"; "b"; "a"]
```

3. Write a function `compress` to remove consecutive duplicates from a list. (8)

```
# compress ["a";"a";"b";"c";"c";"a";"a";"d";"e";"e";"e"];;
- : string list = ["a"; "b"; "c"; "a"; "d"; "e"]
```

4. Write a function `cluster` to cluster consecutive duplicate into nested lists. That is, the type of this function must be `'a list - >'a list list`. (8)

```
# cluster ["a";"a";"a";"b";"c";"c";"a";"a"];;
- : string list list =[["a"; "a"; "a"]; ["b"]; ["c"; "c"]; ["a"; "a"]]
```

5. Python allows us to quickly *slice* a list based on two integers `i` and `j`, to return the sublist from index `i` (inclusive) and `j` (not inclusive). We want such a slicing function in OCaml as well. (8)

   Write a function `slice` as follows: given a list and two indices, `i` and `j`, extract the slice of the list containing the elements from the `i`th (inclusive) to the `j`th (not inclusive) positions in the original list. Lists are 0-indexed in OCaml.

```
# slice ["a";"b";"c";"d";"e";"f";"g";"h"] 2 6;;
- : string list = ["c"; "d"; "e"; "f"]
```

   Invalid index arguments should be handled *gracefully*. For example,

```
# slice ["a";"b";"c";"d";"e";"f";"g";"h"] 3 2;;
- : string list = []
# slice ["a";"b";"c";"d";"e";"f";"g";"h"] 3 20;
- : string list = ["d";"e";"f";"g";"h"];
```

2 Higher-order Functions (30 points)

1. Write a function called `composition`, which takes two functions as its input, and returns their composition as the output. (5)

```
# let square_of_increment = composition square increment;;
val square_of_increment : int -> int = <fun>
# square_of_increment 4;;
- : int = 25
```

2. Write a function called `equiv_on`, which takes three inputs: two functions `f` and `g`, and a list `lst`. It returns `true` if and only if the functions `f` and `g` have identical behavior on every element of `lst`. (5)

```
# let f i = i * i;;
val f : int -> int = <fun>
# let g i = 3 * i;;
val g : int -> int = <fun>
# equiv_on f g [3];;
- : bool = true
# equiv_on f g [1;2;3];;
- : bool = false
```

3. Write a functions called `pairwisefilter` with two parameters: (i) a function `cmp` that compares two elements of a specific `T` and returns one of them, and (ii) a list `lst` of elements of that same type `T`. It returns a list that applies `cmp` while taking two items at a time from `lst`. If `lst` has odd size, the last element is returned "as is". (10)

```
# pairwisefilter min [14; 11; 20; 25; 10; 11];;
- : int list = [11; 20; 10]

# (* assuming that shorter : string * string -> string = <fun> already exists *)
# pairwisefilter shorter ["and"; "this"; "makes"; "shorter"; "strings"; "always"; "win"];;
- : string list = ["and"; "makes"; "always"; "win"]
```

4. Write the `polynomial` function, which takes a list of tuples and returns the polynomial function corresponding to that list. Each tuple in the input list consists of (i) the coefficient, and (ii) the exponent. (10)

```
# let f = polynomial [3, 3;; -2, 1; 5, 0];;
val f : int -> int = <fun>
# f 2;; (* f is the polynomial function f(x) = 3x^3 - 2x + 5 *)
- : int = 25
```

3 Data Types (30 points)

1. Let us define a language for expressions in Boolean logic: (15)

```
type bool_expr =
    | Lit of string
    | Not of bool_expr
    | And of bool_expr * bool_expr
    | Or of bool_expr * bool_expr
```

Using this language, we can write logical expressions in prefix notation. For example, $(a \wedge b) \vee (\neg a)$ can be written as `Or(And(Var("a"), Var("b")), Not(Var("a")))`.

Write a function called `truth_table`, which takes as input a logical expression in two literals, and returns its truth table as a list of triples. Each triple being a tuple of the form:

```
(truth-value-of-first-literal, truth-value-of-second-literal, truth-value-of-expression).
```

For example,

```
# (* the outermost parentheses are needed for OCaml to parse the third argument
     correctly as a bool_expr *)
# truth_table "a" "b" (And(Lit("a"), Lit("b")));;
- : (bool * bool * bool) list = [(true,  true,  true); (true,  false, false);
                                 (false, true, false); (false, false, false)]
```

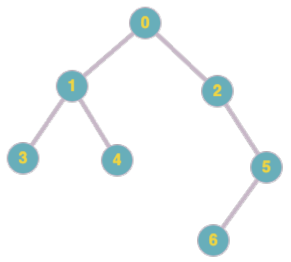

Figure 1: This binary tree should have the string representation `"0(1(3,4),2(,5(6,)))"`.

2. Recall that a binary tree can be defined recursively as follows: (5)

   A binary tree is either empty or it is composed of a root element and two successors, which are binary trees themselves.

   Just like the type `bool_expr` was defined in the previous question, define a polymorphic type `binary_tree` using the above definition. Each element of the tree must be a `Node`, or be `Empty`. For example, a complete binary tree with three levels, and incremental integer values in the nodes, would be defined as

```
let a_tree = Node(1, Node(2, Node(4, Empty, Empty), Node(5, Empty, Empty)),
                     Node(3, Node(6, Empty, Empty), Node(7, Empty, Empty)));;
```

3. Write a function called `tree2str` to obtain the string representation of a binary tree as shown in Fig. 1. For this problem, you may assume that the data in each node is `int`. This restriction is for the `tree2str` method only. The tree implementation (i.e., question 3.2) must be polymorphic. (10)

NOTES:

- As always, **late submissions** or **uncompilable code** will not be graded.
- Please remember to verify what you are submitting. Make sure you are, indeed, submitting what you think you are submitting!
- What to submit? A single `.ml` file containing all your code. Since the method/function definitions are precise, you do NOT have to write your own test cases for grading (or course, you should always use test cases yourself to make sure your code is free of bugs). **This assignment may be graded by a script, so be absolutely sure that the submission follows this structure**.

Submission Deadline: May 3, 2019, 11:59 pm