CSE 216 – Homework I

Instructor: Dr. Ritwik Banerjee

This homework document consists of 3 pages. Carefully read the entire document before you start coding.

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 and Higher-order Functions (56 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 x^n. Also write a function `float_pow`, which does the same thing, but for `x` being a float. `n` is still a non-negative integer. (5)

2. Write a function `compress` to remove consecutive duplicates from a list. (5)
```
# compress ["a";"a";"b";"c";"c";"a";"a";"d";"e";"e";"e"];;
- : string list = ["a"; "b"; "c"; "a"; "d"; "e"]
```

3. Write a function `remove_if` of the type `'a list -> ('a -> bool) -> 'a list`, which takes a list and a predicate, and removes all the elements that satisfy the condition expressed in the predicate. (5)
```
# remove_if [1;2;3;4;5] (fun x -> x mod 2 = 1);;
- : int list = [2; 4]
```

4. Some programming languages (like Python) allow 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. (5)

   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 $\texttt{i}^{\text{th}}$ (inclusive) to the $\texttt{j}^{\text{th}}$ (not inclusive) positions in the original list.
```
# 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"];
```
   You do *not*, however, need to worry about handling negative indices.

5. Write a function `equivs` of the type `('a -> 'a -> bool) -> 'a list -> 'a list list`, which partitions a list into equivalence classes according to the equivalence function. (6)
```
# equivs (=) [1;2;3;4];;
- : int list list = [[1];[2];[3];[4]]
# equivs (fun x y -> (=) (x mod 2) (y mod 2)) [1; 2; 3; 4; 5; 6; 7; 8];;
- : int list list = [[1; 3; 5; 7]; [2; 4; 6; 8]]
```

6. Goldbach's conjecture states that every positive even number greater than 2 is the sum of two prime numbers. E.g., $18 = 5 + 13$, or $42 = 19 + 23$. It is one of the most famous conjectures in number theory. It is unproven, but verified for all integers up to 4×10^{18}. Write a function `goldbachpair : int -> int * int` to find two prime numbers that sum up to a given even integer. The returned pair must have a non-decreasing order. (6)

```
# goldbachpair 10;;  (* must return (3, 7) and not (7, 3) *)
- : int * int = (3, 7)
```

Note that the decomposition is not always unique. E.g., 10 can be written as 3+7 or as 5+5, so both `(3, 7)` and `(5, 5)` are correct answers.

7. 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`. (6)

```
# 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
```

8. 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". (6)

```
# 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"]
```

9. 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. (6)

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

10. The **power set** of a set S is the set of all subsets of S (including the empty set and the entire set). Write a function `powerset` of the type `'a list -> 'a list list`, which treats lists as unordered sets, and returns the powerset of its input list. You may assume that the input list has no duplicates. (6)

```
# powerset [3; 4; 10];;
- : int list list = [[]; [3]; [4]; [10]; [3; 4]; [3; 10]; [4; 10]; [3; 4; 10]];
```

2 Data Types (40 points)

1. Let us define a language for expressions in Boolean logic: (10)

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

using which we can write expressions in prefix notation. E.g., $(a \wedge b) \vee (\neg a)$ is `Or(And(Lit("a"), Lit("b")), Not(Lit("a")))`. Your task is to write a function `truth_table`, which takes as input a logical expression in two literals and returns its truth table as a list of triples, each 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)]
```

2. In this question you will use higher-order functions to implement an interpreter for a simple stack-based evaluation language. This language has a fixed set of commands: (30)

   - `start` → Initializes an empty stack. This is always the first command in a program and never appears again.
   - `(push n)` → Pushes the specified integer `n` on to the top of the stack. This command is always parenthesized.
   - `pop` → Removes the top element of the stack.
   - `add` → Pops the top two elements of the stack, computes their sum, and pushes the result back on to the stack.
   - `mult` → Pops the top two elements of the stack, computes their product, and pushes the result back on to the stack.
   - `clone` → Pushes a duplicate copy of the top element on to the stack.
   - `kpop` → Pops the top element `k` of the stack, and if `k` is a positive number, pops `k` more elements (or the stack becomes empty, whichever happens sooner).
   - `halt` → Terminates the stack evaluation program. This is always the last command.

   Your task is to define the stack data structure in OCaml (10 points), and then implement the above commands (20 points). Your stack must be implemented as a list. A complete running example would look something like the following:

   ```
   # start (push 2) (push 3) (push 4) mult add halt;;
   - : list int = [14]
   ```

   You may assume that only valid commands will be provided. As such, you do not have to worry about exception handling.

3 Version Control (4 points)

For this part, you must use a Bitbucket account with your `stonybrook.edu` email address. You should already have one, based on the first recitation task. You must create a repository with the name **CSE216HW1** on Bitbucket and give access to your grading TA (your grading TA will be allocated based on your last name, and will be announced on Blackboard by the end of this week). Access must be given to your grading TA *before* the homework deadline. Otherwise, this component of the homework will not be graded. We *strongly* encourage you to commit and push your code to this remote repository every time you finish solving any question, an even more frequently for the longer questions (e.g., 2.2). Your grading TA will check for the following:

- There must be at least 4 distinct code pushes to the remote repository, with each push safely committing the solution to at least one new solution to a homework question. [2 points]
- At least 2 code pushes to the remote repository must be made on separate days. [1 point]
- At least 2 code pushes must include meaningul commit messages describing what code has been contributed since the previous version. [1 point]

NOTES:

- **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 `.zip` file comprising three `.ml` files. The first file must be named `hw1.ml`, and should contain your code for the ten questions in section 1 of this assignment. The second file must be named `hw1bool.ml`, and this should contain your code for question 2.1 (Boolean logic). Finally, the third file must be named `hw1stack.ml`, with your code for question 2.2 (stack evaluation language). **This assignment will be graded by a script, so be absolutely sure that the submission follows this structure (including the name of your remote repository)**.

Submission Deadline: Feb 29, 2020, 11:59 pm