CSE 595 – Homework I

Instructor: Dr. Ritwik Banerjee

This homework document consists of **??** pages. Carefully read the entire document before you start coding.
Note: In all λ-calculus problems, the steps should be clearly stated, along with substitutions. All OCaml functions, unless otherwise specified, should be polymorphic. For example, if you are writing a method that should work for lists, the type must be `'a list`, and not `int list`.

1 λ Calculus

1. In the basic untyped lambda calculus, the boolean `true` is encoded as $\lambda x.\lambda y.x$ (i.e., it takes in two arguments and returns the first), and `false` is encoded as $\lambda x.\lambda y.y$ (i.e., takes in two arguments and returns the second). These definitions of the boolean constants may seem strange, but they are designed to work with the if-then-else expression. The if-then-else expression is defined as $\lambda x.\lambda y.\lambda z.((xy)z)$. Verify that these definitions do, indeed, make sense, by evaluating the following: (3)
   (a) $(((\lambda x.\lambda y.\lambda z.((xy)z)(\lambda u.\lambda v.u))A)B)$
   (b) $(((\lambda x.\lambda y.\lambda z.((xy)z)(\lambda u.\lambda v.v))A)B)$
2. Reduce the following λ-expressions to their fullest possible extent: (3)
   (a) $(\lambda x.xx)(\lambda y.yx)z$
   (b) $(((\lambda x.\lambda y.(xy))(\lambda y.y))w)$
3. Given the definitions of `true` and `false` as in the very question, and `not` defined as $\lambda x.((x$ `false`$)$ `true`$)$, and `or` defined as $\lambda x.\lambda y.((x$ `true`$)\ y)$, prove the following: (4)
   (a) `not (not true) = true`
   (b) `or false true = true`

2 Recursion and Higher-order Functions

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. `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 `i`$^{\text{th}}$ (inclusive) to the `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. (7)

   ```
   # 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. (7)

   ```
   # 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`. (7)

   ```
   # 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". (7)

   ```
   # 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. (7)

   ```
   # (* 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. The output must be in increasing order of the set sizes (but the order of the elements within each set is irrelevant). (7)

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

3 Data Types

1. Let us define a language for expressions in Boolean logic: (8)

```
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. Some data types are naturally recursive. Trees (or tree-like data structures) come to mind, since a subtree rooted under some node is a tree by itself. Here, you are being asked to define the abstract syntax tree of a simple arithmetic language. The specifications are as follows: (10)

   - An arithmetic expression (`expr`) will be either a numeric constant called `Const`, or a variable called `Var`, or addition (`Plus`), multiplication (`Mult`), difference (`Minus`), or division (`Div`) of two arithmetic expressions.
   - Since we are dealing with binary arithmetic operators, the arguments are defined with the names `Arg1` and `Arg2`.

   Using this type definition, we can represent simple arithmetic expressions in OCaml. For example, $2x + 3(y - 1)$ can be represented as

```
# Plus {Mult {Arg1 = Const 2; Arg1 = Var "x"}; Mult {Arg1 = Const 3; Arg2 =
  Minus {Arg1 = Var "y"; Const 1}}};;
- : expr = Plus { Mult {Arg1 = Const 2; Arg1 = Var "x"}; Mult {Arg1 = Const 3;
    Arg2 = Minus {Arg1 = Var "y"; Const 1}}}
```

3. Finally, write a function called `evaluate`, which takes as input a single arithmetic expression you defined in above, and gives as output its final value. You may assume that the input arithmetic expression to this function will only non-negative integer numeric values (i.e., no variables and no floats). (10)

```
# let expr = Plus {Mult {Arg1 = Const 2; Arg1 = Const 3}; Mult {Arg1 = Const 3;
 Arg2 = Minus {Arg1 = Const 4; Const 1}}} in evaluate(expr);;
- : int = 15
```

NOTES:

- 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 one PDF file with the λ-calculus solutions, and 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 `hw1math.ml`, with your code for question 2.2 and 2.3 (the arithmetic expression definition and evaluation).
  This assignment will be graded by a script, so be absolutely sure that you follow this structure.
- **Uncompilable code** will not be graded. So make sure that your `.ml` files actually compile and run. Do NOT simply check individual methods and submit the final code without doing this!
  You don't want to be in a situation where `hw1.ml` doesn't compile because of that one function you copy-pasted from your terminal REPL and made a careless mistake (for example, forgot to include the "`;;`" at the end of it)!
- Please remember that after the submission deadline, there is a penalty of 10% for every 1-day delay in the submission. For example, if a submission receives 75/100 and submitted 10 minutes after the deadline, it will receive 67.5/100.

Submission Deadline: March 19, 2021, 11:59 pm