CS 496: Homework Assignment 1
Due: January 27th, 11:55pm

1 Assignment Policies

Collaboration Policy. Homework will be done individually: each student must hand in their own answers. It is acceptable for students to collaborate in understanding the material but not in solving the problems or programming. Use of the Internet is allowed, but should not include searching for existing solutions.

Under absolutely no circumstances code can be exchanged between students. Excerpts of code presented in class can be used.

Assignments from previous offerings of the course must not be re-used. Violations will be penalized appropriately.

2 Assignment

Exercise 1

Give the type annotations and implement the functions listed below. Type annotations should be presented as a comment, just before the definition of the function itself. For example, here is the annotation and code for the add function:

```
(* add: int -> int -> int *)
let add x y = x+y
```

1. `seven`: that given any value returns 7.
2. `sign`: that given an integer returns 1 if it is positive, -1 if it is negative and 0 if it is zero.
3. `absolute`: the absolute value function.
4. `andp,orp,notp,xorp`: the standard, two argument (except for `notp` which is unary), boolean connectives (you must resort to if-then-else or a `match` statement).
5. `dividesBy`: that given two numbers determines if the first is divisible by the second (use remainder).

6. `is_singleton`: a predicate that, given a list, returns a boolean indicating whether it has exactly one element. Provide your solution using the `match` construct for pattern matching. Your solution should NOT depend on computing the length of the list.

7. `swap`: a function that, given a pair, returns the same pair except that its first and second components are interchanged.

8. `app`: a function that, given two arguments, applies the first argument to the second one. Eg. if `succ` is the successor function then

```
# app succ 2;;
- : int = 3
```

9. `twice`: a function that, given two arguments, applies the first one to the second argument and then again to the result. Eg.

```
# twice succ 2;;
- : int = 4
```

10. `compose`: a function that, given three arguments, applies the second to the third and then the first to its result. Eg.

```
# compose succ succ 3;;
- : int = 5
```

Exercise 2

Give the type annotations and implement the following functions:

1. Define the binary operations of union and intersection and the `belongsTo` predicate, for sets of type `a` when such sets are represented as:

   (a) Extensionally (i.e. as a list of elements of type a).

   (b) Characteristic function (i.e. as a predicate of type `'a -> bool`).

   This should yield six functions: `union_ext, union_char, intersection_ext, intersection_char, belongsTo_ext, belongsTo_char`.

2. `remAdjDups` that removes adjacent duplicates from a list. Eg. `remAdjDups [1;1;2;3;3;1;4;4;4]` should return `[1;2;3;1;4]`.

3. `sublists` that computes all the sublists of a list, in any order. Eg. `sublists [1;2;3]` should produce `[[]; [1];[2];[1;2];[3];[1;3];[2;3];[1;2;3]]`. Note that, a sublist results from a list by dropping any number of its elements (without rearranging them).

Exercise 3

Assume that we have the type `calcExp` of *calculator expressions*, defined as follows:

```
type calcExp =
  | Const of int
  | Add of (calcExp*calcExp)
  | Sub of (calcExp*calcExp)
  | Mult of (calcExp*calcExp)
  | Div of (calcExp*calcExp)
```

Here are two example calculator expressions:

```
let e1 = Const(2)

let e2 = Add(Sub(Const(2),Const(3)), Const(4))
```

1. Define `mapC`, a map for calculator expressions. Eg. `mapC (fun x -> x + 1) e2` should return `Add(Sub(Const(3), Const(4)), Const(5))`.
2. Define `foldC`, a fold for calculator expressions.
3. Using `foldC`, define a function `numAdd` that returns how many occurrences of `Add` there are in a given calculator expression.
4. Using `foldC`, define a function `replaceAddWithMult` that replaces all `Add` by `Mult` in a calculator expression.
5. Define `evalC`, an evaluator for calculator expressions without using `foldC`. Eg. `evalC e2` should return `3`.
6. Define `evalCf`, an evaluator for calculator expressions using `foldC`.

Exercise 4

Give the type annotations and implement the following functions:

1. Explain what the function `f` below does. Justify your answer by running it on some examples.

```
let f xs =
  let g = fun x r -> if x mod 2 = 0 then (+) r 1 else r
  in List.fold_right g xs 0
```

2. (Optional if you do 3) Complete the following definition of `concat` that flattens a list of lists. In order to complete it you have to fill in the dots. Careful, this is a use of fold at a higher-order type.

```
let concat xss =
  let g = fun xs h -> ...
  in List.fold_right g xss []
```

3. (Optional if you do 2) Complete the following definition of `append` that appends two lists. In order to complete it you have to fill in the dots.

```
let append xs =
  let g = fun x h -> ...
  in List.fold_right g xs
```

Note that `append [1;2]` should produce a function that when applied to another list, concatenates the two. For example,

```
# append [1;2;3] [4;5;6];;
- : int list = [1;2;3;4;5;6]
```

3 Submission instructions

Submit a single file named `hw1.ml` through Canvas. No report is required. Your grade will be determined as follows:

- You will get 0 points if your code does not compile.
- Partial credit may be given for style, comments and readability.