CS 320 Theory Homework #2

Due: TBD

Pattern Match

Exhaustively match all patterns of the following expressions using only the `match` keyword. You should match until there are only base types (`int, float, bool, string`). A wildcard `(_)` should be used to match the right-hand side of the cons `(_::_)` pattern.

Below are three examples that attempt to illustrate what satisfies our requirements.

WRONG EXAMPLE 1.1:

```
e: (int * bool) list

match e with
| [] -> …
| a :: _ -> …
```

Wrong, because `a` is a tuple which is not a base type so it must be matched further.

CORRECT EXAMPLE 1.2:

```
e: (int * bool) list

match e with
| [] -> …
| (a, b) :: _ -> …
```

Correct, because the pattern `(a, b)` are of type int and bool respectively.

CORRECT EXAMPLE 1.3:

```
x: (int * bool) list option

match x with
| None -> …
| Some l -> (
  match l with
  | []->…
  | (a, b) :: _ -> …)
```

Correct, because both constructors of `x` have been matched. In the case of the `(_::_)` pattern, the head element of type (int * bool) has been matched against its left and right components. The components `a` has type `int` and `b` has type bool, which are base types.

1. Match the following expressions

1.1. `x: (int option * float) option`

```
match x with

|None -> …
|Some -> 1 -> match l with
     |(v1,v2) -> match v1 with
          |None -> …
          |Some d-> …
```

1.2. `y: string option option list`

```
match y with
|None -> …
|Some x -> match x with
     |None -> …
     |Some l -> match l with
          |[] -> …
          |list::_ -> …
```

1.3. `z: int option list option`

```
match z with
|None ->…
|Some a -> match a with...
     |[] -> …
     |mdx:: -> match mdx with
          |None -> …
          |Some b -> …
```

Type Inference

2. Consider a function with polymorphic type `f : 'a -> 'a -> 'a.`

2.1. What is the type of `f 0` ?

`int -> int`

2.2. What is the type of `fun x -> f x x` ? Briefly explain your reasoning.

('a^2->'a^2->'a^2)

f first takes a function type due to input, then we get 'a->'a->'a, x isn't identified, so we take the input and square the x's in the function. x*x, because of f x x. so it would be this data type with the squared a's due to the squared input.

3. Consider a function with polymorphic type `f : ('a * 'b) -> ('b -> 'a) -> 'a`

3.1. What is the type of `f (0, true)` ?

(bool -> int) -> int

3.2. What is the type of `fun x y -> f (x, y)` ? Does it have the same type as `f` ? Briefly explain your reasoning.

‘a -> ‘b -> (‘b -> ‘a) -> ‘a

No, f is a different type, it takes (x,y) which is type ‘a and ‘b and not of int and bool.

4. Consider a function with polymorphic type `g : ('a -> 'b -> 'b) -> 'b -> 'b.`

4.1. What is the type of `g (^)`?

int-> int

4.2. What is the type of `fun x -> (g g) x` ? Briefly explain your reasoning.

‘a->’a->’a->’a

x is type of ‘a, as the input, and it outputs the input, so two outputs of g would end up as ‘a->’a->’a->’a, when the ocaml phrase x-> comes into play.

5. Consider the following function.

```
let f (x : bool list) : int list =
   match x with
   | [] -> x
   | hd :: tl -> 0 :: []
```

Is it well typed? Briefly explain your reasoning.

`It is not well typed because x is returned in the first match case, when x is empty bool list, but x itself is a bool list and not a int list.`

Higher Order Functions

In the following section, you have access to the `fold_left` standard library function with the following signature.

```
fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
```

You may **not** use pattern matching or calls to other library functions. If a function has been declared in a previous problem, you may call it in future problems even if you have not worked out its exact solution. (Example: you may use 6.1 rev inside of 6.2 append, but you may not use 6.4 filter inside of 6.3 map.)

6. Implement the following standard list functions. When given the same input, they should have the same output as their standard library counterparts.

6.1. `rev : 'a list -> 'a list`

```
let rev y = List.fold_left (fun accum x-> x::accum) [] y;;
```

6.2. `append : 'a list -> 'a list -> 'a list`

```
let append y x = List.fold_left (fun accum a -> a:: accum) (rev y) x
```

6.3. `map : ('a -> 'b) -> 'a list -> 'b list`

```
let map x y = List.fold_left(fun accum b -> (x b) :: accum) [] (rev y)
```

6.4. `filter : ('a -> bool) -> 'a list -> 'a list`

```
let filter y l = List.fold_left(fun accum x-> if y x then accum@x else
accum) [] (rev l)
```

6.5. `fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b`

```
let fold_right y l = List.fold_left (fun accum x-> y x accum ) 0 (rev l)
```

7. Construct a function with the follow signature, using only functions declared above (6.1 - 6.5).

```
combinations : 'a list -> 'b list -> ('a * 'b) list list
```

Such that when given two lists of lengths `m` and `n` respectively,

```
[ a1; a2; a3; … ; am ] : 'a list
[ b1; b2; b3; … ; bn ] : 'b list
```

It computes a nested list of all combinations of their elements pair together. They should be ordered as shown below. (Hint: In python we can do this with a nested for-loop. What corresponds to a for-loop in ocaml?)

```
[ [ (a1, b1); (a1, b2); (a1, b3); … ; (a1, bn) ];
  [ (a2, b1); (a2, b2); (a2, b3); … ; (a2, bn) ];
  [ (a3, b1); (a3, b2); (a3, b3); … ; (a3, bn) ];
  …
  [ (am, b1); (am, b2); (am, b3); … ; (am, bn) ] ]
```

```
let combinations i j = List.map(fun x -> List.map(fun y -> (x,y))j) i
```