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 x → (
  match x with
  (a,b) →
    match a with
      None →
      some a →.
```

   1.2. `y: (string)option)option)list`

```
match y with
[] →
a::_ →
match a with
  None →
  Some a →
  match a with
    None →
    Some a →.
```

1.3. `z : (int option) list) option`

```
match z with
  None ->
  Some z ->
    match z with
      [] ->
      a::_ ->
      match a with
        None ->
        Some a ->
```

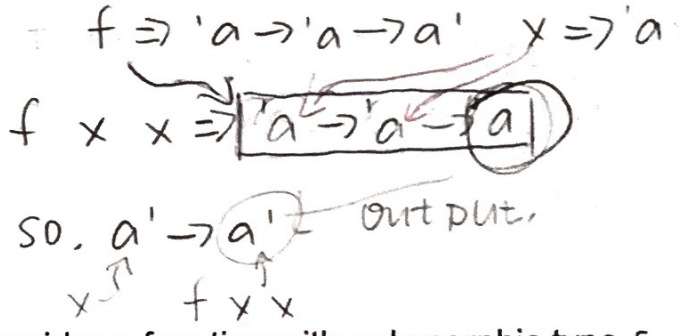
Type Inference

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.

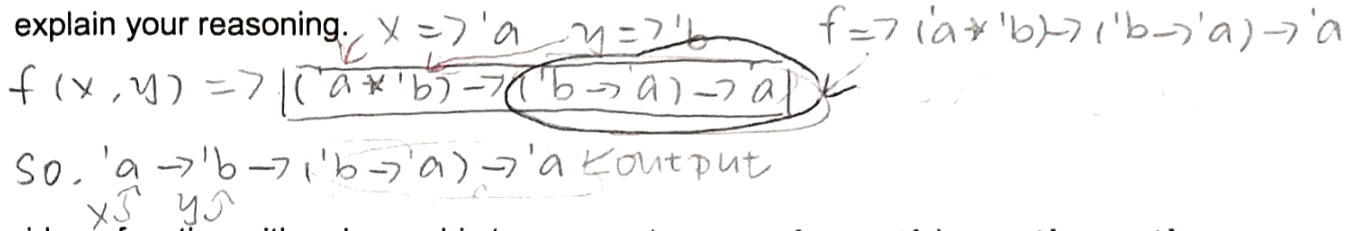


3. Consider a function with polymorphic type `f : ('a * 'b) -> ('b -> 'a) -> 'a`

   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.

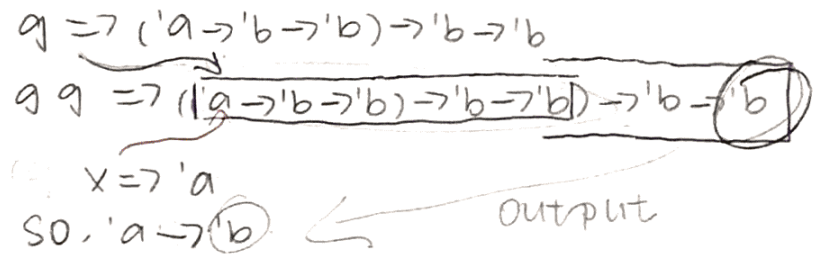


4. Consider a function with polymorphic type `g : ('a -> 'b -> 'b) -> 'b -> 'b.`

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

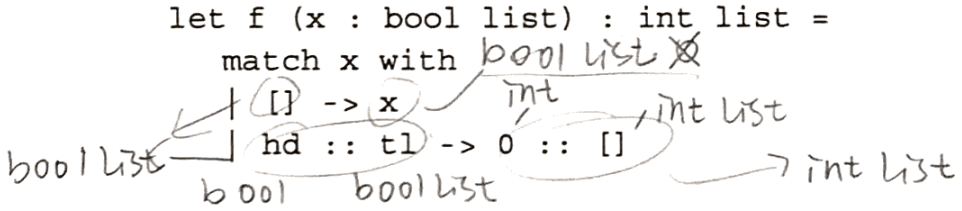
string -> string

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



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.

No -> bool list -> Error (should be int list)

[] -> x

hd :: tl -> 0 :: []

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.)

Please see screenshot on next page :>

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`

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

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

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

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

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 rev l =
  List.fold_left (fun a x -> x::a) [] l

let append l ll =
  List.fold_left (fun a x -> x::a) l (rev ll)

let map f l =
  List.fold_left (fun a x -> (f x)::a) [] (rev l)

let filter f l =
  List.fold_left (fun a x -> if f x then x::a else a) [] (rev l)

let fold_right f l accu =
  List.fold_left (fun a x -> f x a) accu (rev l)

let combinations l ll =
  let aux =
    List.fold_left (fun accu x -> List.fold_left (fun accu y -> (x, y)::accu) accu ll) [] l
  in (rev aux)
```