1 Repeats

Without using any functions from the standard library, implement the function

```
val repeats : ('a * int) list → 'a option list
```

so that `repeats l` is the result of replacing each tuple `(x, n)` with `n` copies of `Some x` in the case that `n` is nonnegative and `-n` copies of `None` otherwise. **Your implementation must be tail recursive.**

```
let repeats l =
  let rec f l acc =
    match l with
    | [] → acc
    | (x,n)::l →
      if n = 0 then f l acc
      else if n > 0
      then f ((x,n-1)::l) (Some x::acc)
      else f ((x,n+1)::l) (None::acc)
  in let rec rev l acc =
    match l with
    | [] → acc
    | a::l → rev l (a::acc)
  in rev (f l []) []
```

2 Merge Sort

Consider the following partial implementation of merge sort using a specialized ADT called `merge list`.

```
let rec append (x : 'a) (l : 'a merge_list) :
'a merge_list = match l with
  | Nil -> Single x
  | Single y -> Merge {left=Single x;right=Single y}
  | Merge {left;right} ->
     Merge {left=right;right=append x left}
let rec of_list l = match l with
  | [] -> Nil
  | x :: xs -> append x (of_list xs)

let rec merge l r = assert false

let rec merge_sort (l : 'a list) : 'a list =
  let rec go l = match l with
    | Nil -> []
    | Single x -> [x]
    | Merge ls -> merge (go ls.left) (go ls.right)
  in go (of_list l)
```

A. Based on the above code, give the definition of the `merge list` type.

B. Implement the function

```
val merge :  'a list -> 'a list -> 'a list
```

so that `merge l r` is the sorted list with the same elements as `l @ r`, **assuming** `l` **and** `r` **are already sorted.** You may not use any functions from the standard library except for comparison functions like (<).

```
type 'a merge_list =
   | Nil
   | Single of 'a
   | Merge of
     { left: 'a merge_list;
       right:'a merge_list }
```

```
let rec merge l r =
  match l,r with
  | [],r → r
  | l,[] → l
  | a::l,b::r →
    if a < b then a :: merge l (b::r)
    else b :: merge (a::l) r
```

3 Typing Derivations

A. Write down an expression of type `('a * 'b) -> ('b * 'a)`.

B. Let e denote the expression you wrote down in the previous part. Write a derivation of the judgment

$$\cdot \vdash e : (\tau_1 \; * \; \tau_2) \; \rightarrow \; (\tau_2 \; * \; \tau_1)$$

where your derivation should be written in terms of τ_1 and τ_2.[1]

```
fun v → match v with (x,y) → (y,x)
```

$$\dfrac{\dfrac{\dfrac{}{v : \tau_1 \times \tau_2 \vdash v : \tau_1 \times \tau_2}\;\text{var} \qquad \dfrac{\dfrac{}{v : \tau_1 \times \tau_2, x : \tau_1, y : \tau_2 \vdash y : \tau_2}\;\text{var} \quad \dfrac{}{v : \tau_1 \times \tau_2, x : \tau_1, y : \tau_2 \vdash x : \tau_1}\;\text{var}}{v : \tau_1 \times \tau_2, x : \tau_1, y : \tau_2 \vdash (y,x) : \tau_2 \times \tau_1}\;\text{tuple}}{v : \tau_1 \times \tau_2 \vdash \texttt{match v with (x,y)} \longrightarrow \texttt{(y,x)} : \tau_2 \times \tau_1}\;\text{tuple-match}}{\emptyset \vdash \texttt{fun v} \longrightarrow \texttt{match v with (x,y)} \longrightarrow \texttt{(y,x)} : \tau_1 \times \tau_2 \rightarrow \tau_2 \times \tau_1}\;\text{fun}$$

4 Alternating Paths

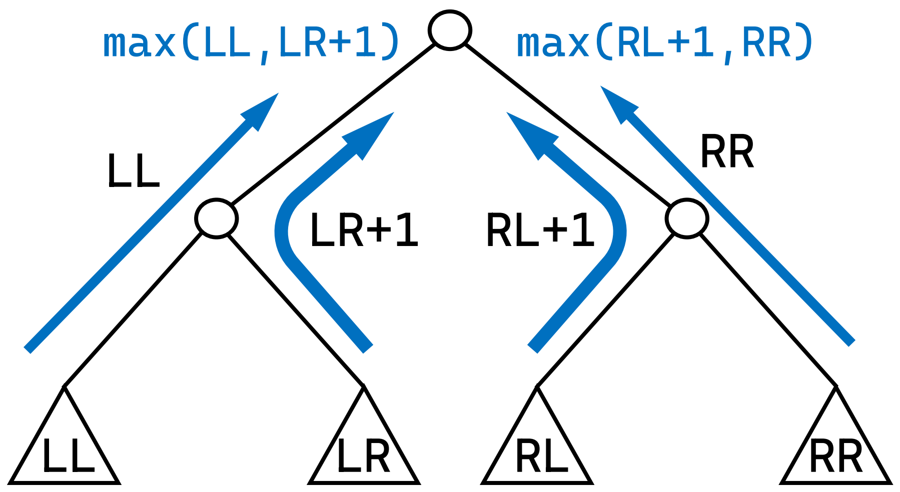
Consider the following ADT for a binary tree.

```
type 'a tree =
  | Leaf
  | Node of 'a * 'a tree * 'a tree
```

We can think of a path in a binary tree from the root of the tree to a leaf as a sequence of "lefts" and "rights", i.e., whether the path goes down a left subtree or a right subtree. The *alternation number* of a path is the number of times the path went "left" after going "right" or vice versa. The alternation number of a tree is the maximum alternation number over all paths from the root to a leaf in the tree. Implement the function

```
val alt num : 'a tree -> int
```

so that `alt num t` is the alternation number of the tree `t`. You may not use any function in the standard library except `max`. *Hint:* Write a helper function that returns *two* values instead of one.




```
let rec alt_num_aux (t:'a tree) : int * int =
  match t with
  | Leaf → (-1,-1)
  | Node (_,tl,tr) →
    let ll,lr = alt_num_aux tl in
    let rl,rr = alt_num_aux tr in
    max ll (lr+1), max (rl+1) rr

let alt_num t =
  match t with
  | Leaf → 0
  | _ →
    let l,r = alt_num_aux t
    in max l r
```

5 Options, Formally

We've seen option types in OCaml, but we did not include the typing rules in our `320Caml` specification.

A. In analogy with lists, provide the typing rules for option types. Recall that options are defined by the following ADT

```
type 'a option =
  | None
  | Some of 'a
```

B. Give the typing rule for shallow pattern matching on options. That is, write down the rules for determining how to type an evaluate an expression of the following form:

```
match o with | None -> none_case | Some n -> some_case
```

$$\frac{}{\Gamma \vdash \texttt{None} : \tau\ \texttt{option}}\ \text{option-none} \qquad \frac{\Gamma \vdash \texttt{e} : \tau}{\Gamma \vdash \texttt{Some e} : \tau\ \texttt{option}}\ \text{option-some}$$

$$\frac{\Gamma \vdash \texttt{o} : \tau\ \texttt{option} \qquad \Gamma \vdash \texttt{e1} : \tau' \qquad \Gamma, \texttt{v} : \tau \vdash \texttt{e2} : \tau'}{\Gamma \vdash \texttt{match o with | None} \rightarrow \texttt{e1 | Some v} \rightarrow \texttt{e2} : \tau'}\ \text{option-match}$$

6 Semantic Derivation

Give a derivation of the following semantic judgment.

$$\texttt{let x = 2 in let z = x + x in (x * z, z)} \Downarrow (8, 4)$$

$$
\begin{prooftree}
\AxiomC{}
\RightLabel{int-lit}
\UnaryInfC{$2 \Downarrow 2$}
\AxiomC{}
\RightLabel{int-lit}
\UnaryInfC{$2 \Downarrow 2$}
\AxiomC{}
\RightLabel{int-lit}
\UnaryInfC{$2 \Downarrow 2$}
\RightLabel{add-int}
\BinaryInfC{$\texttt{2+2} \Downarrow 4$}
\AxiomC{}
\RightLabel{int-lit}
\UnaryInfC{$2 \Downarrow 2$}
\AxiomC{}
\RightLabel{int-lit}
\UnaryInfC{$4 \Downarrow 4$}
\RightLabel{mul-int}
\BinaryInfC{$\texttt{2*4} \Downarrow 8$}
\AxiomC{}
\RightLabel{int-lit}
\UnaryInfC{$4 \Downarrow 4$}
\RightLabel{tuple}
\BinaryInfC{$\texttt{(2*4, 4)} \Downarrow \texttt{(8,4)}$}
\RightLabel{let}
\BinaryInfC{$\texttt{let z = 2 + 2 in (2*z, z)} \Downarrow \texttt{(8,4)}$}
\RightLabel{let}
\BinaryInfC{$\texttt{let x = 2 in let z = x + x in (x*z, z)} \Downarrow \texttt{(8,4)}$}
\end{prooftree}
$$