1 ——— [2+2 points]

Fill in boxes so that the following equalities hold for all lists l and all functions f.

(a) `List.map` f l = `List.fold_right` [ ] l [ ]

(b) `List.rev_map` f l = `List.fold_left` [ ] [ ] l

Note that `List.rev_map` f $[x_1; \cdots ; x_n] = [f\ x_n; \cdots ; f\ x_1]$.

---

2 ——— [2+8 points]

(a) Define a tail recursive version `length'` of `length`:

```
let rec length = function
  | [] -> 0
  | x :: xs -> 1 + length xs
```

(b) Prove that `length` l = `length'` l holds for all lists l.

---

3 [8 POINTS]

Consider the following type of binary trees:

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

We say that a tree t is *balanced* if all paths from the root to any leaf have the same length. The function `balanced` t returns `true` if t is balanced, `false` otherwise. Implement `balanced`.

```
# balanced Leaf;;
- : bool = true
# balanced (Node (Leaf, Leaf));;
- : bool = true
# balanced (Node (Leaf, Node (Leaf, Leaf)));;
- : bool = false
# balanced (Node (Node (Leaf, Leaf), Leaf));;
- : bool = false
# balanced (Node (Node (Leaf, Leaf), Node (Leaf, Leaf)));;
- : bool = true
```

4 [10 POINTS]

Suppose that we use adjacency lists to represent finite directed graphs.

```
type 'a graph = ('a * 'a list) list
```

We say that in a graph g a node y is *reachable from* a node x if there is a path from x to y in g. Implement the function `reachable_from : 'a graph -> 'a -> 'a list`. Here `reachable_from` g x returns a list containing all reachable nodes from x in g. For example,

```
# let g = [(1,[3]);(2,[3]);(3,[4]);(4,[3;5;6]);(5,[]);(6,[])];;
# reachable_from g 3;;
- : int list = [3; 4; 5; 6]
```

(You do not need to eliminate duplication from resulting lists.)

Name:

5 [8 POINTS]

Consider the following expressions of the type `expr`:

```
type expr =
  | Int of int
  | Add of expr * expr
  | Sub of expr * expr
  | Let of string * expr * expr
type env = (string * int) list
exception Unbound of string
```

For example, the expression "`let x = 3 - 1 in x + x`" is represented by

```
Let ("x", Sub (Int 3, Int 1), Add (Var "x", Var "x")).
```

Implement an evaluator for `expr`.

```
# eval [] (Let ("x", Sub (Int 3, Int 1), Add (Var "x", Var "x")));;
- : int = 4
# eval [("x", 1)] (Add (Int 3, Var "x"));;
- : int = 4
# eval [] (Add (Int 3, Var "x"));;
Exception: Unbound "x".
```

6 [10 POINTS]

Recall the following simplification rules for type inference problems.

$$
\begin{array}{lll}
A \rhd z : \tau & \rightsquigarrow \bot & \text{if } z \notin \text{dom}(A) \\
A \rhd z : \tau & \rightsquigarrow A(z) \doteq \tau & \text{if } z \in \text{dom}(A) \\
A \rhd (\texttt{fun}\ x \to e) : \tau & \rightsquigarrow \exists \alpha_1, \alpha_2.(A, x : \alpha_1 \rhd e : \alpha_2 \ \wedge \ \tau \doteq \alpha_1 \to \alpha_2) & \\
A \rhd e_1 e_2 : \tau & \rightsquigarrow \exists \alpha.(A \rhd e_1 : \alpha \to \tau \ \wedge \ A \rhd e_2 : \alpha) &
\end{array}
$$

where z is a constant or a variable, and α, α_1, and α_2 are fresh type variables. Solve the type inference problem:

$$A \rhd (\texttt{fun}\ x \to x = 0) : \alpha$$

Here A is the type environment defined by $A(0) = \texttt{int}$ and $A(=) = \texttt{int} \to \texttt{int} \to \texttt{bool}$.