TD n°8 - Récursivité terminale

Exercice 1

Écrire une fonction récursive terminale `length` qui calcule la longueur d'une liste.

Écrire une fonction récursive terminale `sum` qui calcule la somme des éléments d'une liste.

Exercice 2

Transformer les fonctions suivantes en des fonctions récursives terminales :

```
let rec f1 n =
 if n=0 then 1
 else 1+n*3+ (f1 (n-1));;
```

```
let rec f2 l el = match l with
   |[] -> 0
   |t::q when t=el -> 1+(f2 q el)
   |t::q -> f2 q el;;
```

```
let rec f3 l = match l with
   |[] -> []
   |t::q -> t::t::(f3 q);;
```

```
let rec f4 n x = match n with
 |0 -> 1
 |_ -> x*(f4 (n-1) x);;
```

Exercice 3

On donne la fonction suivante qui est de type `int -> ('a->'a) -> 'a -> 'a` et qui prend en entrée un nombre entier n, une fonction f et un élément x de l'espace de départ de f.

```
let rec itere n f x = match n with
  | 0 -> x
  | _ -> f (itere (n - 1) f x);;
```

1. Que fait cette fonction ?
2. Écrire une version récursive terminale de cette fonction.

Exercice 4

1. Compléter la fonction `concat` suivante qui concatène deux listes (c'est-à-dire qui impélmente `@`). Cette version n'est pas récursive terminale.

   ```
   let rec concat l1 l2 = match l1 with
    |[] -> ...
    |t::q -> ... :: (concat ... ...);;
   ```

2. Compléter la fonction `rev_concat` suivante telle que `rev_concat l1 l2` renvoie le renversement de `l1` concaténé avec `l2`. Cette fonction est-elle récursive terminale ?

   ```
   let rec rev_concat l1 l2 = match l1 with
    |[] -> ...
    |t::q -> rev_concat ... (...::...);;
   ```

3. En déduire une fonction récursive terminale `rev : 'a list -> 'a list` qui renverse une liste. Par exemple elle transforme `[1;2;3;4]` en `[4;3;2;1]`
4. Utiliser les deux fonctions précédentes pour écrire une fonction `concat2 : 'a list -> 'a list -> 'a list` qui concatène deux listes et qui soit récursive terminale.

Exercice 5

Dans cet exercice nous allons étudier deux fonctions qui sont prédéfinies en Ocaml :

- `List.map : ('a -> 'b) -> 'a list -> 'b list` qui a une fonction $f : A \mapsto B$ et une liste $[a_0; a_1; ...; a_{n-1}]$, associe la liste $[f(a_0); f(a_1); ...; f(a_{n-1})]$,
- `List.fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a` qui a une fonction $f : A \times B \mapsto A$, un élément $a \in A$ et une liste $[b_0; b_1; ...; b_{n-1}]$ d'éléments de B associe l'élément $f(...f(f(f(a, b_0), b_1), b_2)..., b_{n-1})$ de A.
- `List.fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b` qui, à une fonction $f : A \times B \mapsto B$, une liste $[a_0; a_1; ...a_{n-1}]$ d'éléments de A et un élément $b \in B$, associe l'élément $f(a_0, f(a_1, f(a_2, ..., f(a_{n-1}, b)))$ de B.

1. Redéfinir la fonction `List.length` à l'aide de `List.fold_left` puis à l'aide de `List.fold_left`. Comment calculer la somme des éléments d'une liste en utilisant ces deux fonctions ? Et pour le produit ?
2. Implémenter ces deux fonctions (si possible de manière récursive terminale).

3. Déterminer le type et ce que réalisent les fonction suivantes :

```
let myst1 elt lst = List.fold_right (fun a b -> a :: b) lst [elt];;
let myst2 = List.fold_right (fun a b -> a :: b);;
let myst3 lst = List.fold_left min (List.hd lst) (List.tl lst);;
```

4. Réaliser la fonction **`List.map`** à l'aide de **`List.fold_right`**.