TD n° 4 : Récursivité terminale

EXERCICE 1 La fonction suivante n'est pas récursive terminale

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

Comment rendre cette fonction récursive terminale ?

EXERCICE 2 (*Concaténation récursive terminale*)

1. Rappeler comment écrire en CAML une fonction `concat` qui concatène deux listes (c'est-à-dire comment implémenter `@`). Quelle est sa complexité ? Cette fonction est-elle récursive terminale ?
2. Écrire une fonction `rev_append` de type `'a list -> 'a list -> 'a list` telle que `rev_append l1 l2` renvoie le renversement de `l1` concaténé avec `l2`. Cette fonction est-elle récursive terminale ? Quelle est sa complexité ?
3. En déduire la fonction `rev : 'a list -> 'a list`
4. En déduire une fonction `append : 'a list -> 'a list -> 'a list` qui concatène deux listes. Quelle est sa complexité ? Est-elle récursive terminale ?

EXERCICE 3 La fonction `sigma` vue en cours n'est pas récursive terminale :

OCAML
```
let rec sigma n =
  match n with
  | 0 -> 0
  | n -> n + sigma (n - 1)
```

En utilisant une fonction auxiliaire et un accumulateur, écrire une version récursive terminale de cette fonction somme.

EXERCICE 4 Réécrire la fonction `length : 'a list -> int` de façon à ce qu'elle soit récursive terminale [1].

EXERCICE 5 (*Fold left et fold right*) La majorité des fonctions agissant sur les listes suivent un schéma récursif similaire. Nous avons étudié en cours la fonctionnelle `List.map` qui généralise l'application d'une fonction à tous les éléments d'une liste. Dans cet exercice, nous allons étudier deux fonctionnelles prédéfinies en CAML :

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

1. Redéfinir la fonction `length` à l'aide de l'une puis l'autre de ces fonctions. Comment calculer la somme des éléments d'une liste ? Le produit ?
2. Implémenter ces deux fonctions (si possible de manière récursive terminale).
3. Déterminer le type et ce que réalisent les fonction suivantes :

   OCAML
   ```
   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 `map` à l'aide de `List.fold_right`.

EXERCICE 6 (*Fonction 91 de McCarthy*) Déterminer ce que calcule la fonction suivante et le démontrer :

OCAML
```
let rec f_91 n =
  if n > 100 then
    n - 10
  else
    f_91 (f_91 (n + 11))
```

Remarquons qu'il n'était pas évident *a priori* qu'une telle fonction renvoie bien un résultat.

1. C'est bien le cas de la fonction `List.length` fournie par CAML.