TD n°8 - Corrigé

Exercice 1

```
let length l =
 let rec aux l acc = match l with
   |[] -> acc
   |t::q -> aux q (acc+1)
 in aux l 0;;
```

```
let sum l =
 let rec aux l acc = match l with
   |[] -> acc
   |t::q -> aux q (acc+t)
 in aux l 0;;
```

Exercice 2

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

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

```
let f3 l =
  let rec aux l lacc = match l with
    |[] -> lacc
    |t::q -> aux q (t::t::lacc)
  in List.reverse (aux l []) (*La liste lacc est construite vdans l'envers*);;
```

```
let f4 n x =
  let rec aux n acc =
    if n=0 then acc
    else aux (n-1) (acc*x)
  in aux n 1;;
```

Exercice 3

La fonction calcule $f^n(x)$, c'est à dire f composée n fois avec elle-même, appliquée en x. La relation de récurrence de la fonction récursive du TD est $f^n(x) = f(f^{n-1}(x))$. On a également $f^n(x) = f^{n-1}(f(x))$, qui permet l'écriture d'un fonction récursive terminale.

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

Exercice 4

1. ```
   let rec concat l1 l2 =
     match l1 with
     | [] -> l2
     | h::q -> h::(concat q l2);;
   ```

   Cette fonction n'est pas récursive terminale, la dernière opération effectuée est la construction d'une liste avec h comme premier élément et le résultat de `concat q l2` comme queue.

2. ```
   let rec rev_concat l1 l2 =
     match l1 with
     | [] -> l2
     | h::q -> rev_concat q h::l2;;
   ```

   Cette fonction est récursive terminale

3. ```
   let rec rev l = rev_concat l [];;
   ```

   Cette fonction est récursive terminale car la fonction à laquelle elle fait appel est récursive terminale.

4. ```
   let rec concat2 l1 l2 = rev_concat (rev l1) l2;;
   ```

   Cette fonction est bien récursive terminale car toutes les fonctions auxquelles elle fait appel le sont.

Exercice 5

On rappelle :

- **`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. Questions pour vérifier la compréhension de la définition de **`fold_left`**.

   ```
   let length l = fold_left (function a x -> a+1) 0 l;;

   let length_bis l = fold_right (function x a -> a+1) l 0;;

   let somme l = fold_left (function a x -> a+x) 0 l;;

   let prod l = fold_left (function a x -> a*x) 0 l;;
   ```

2. La formule de récurrence est donnée dans l'énoncé, c'est la définition de **`fold_left`** (resp. **`fold_right`**). Pour le cas de base on peut considérer la liste vide, dans ce cas on renvoie juste **`a`** (on peut juger que ça en respecte pas la définition). On peut sinon considérer le cas de base à un élément **`[e]`** pour lequel on renvoie **`f a e`** (resp. **`f e a`**).

   ```
   let rec fold_left f a l = match l with
     |[] -> a
     |h::q -> fold_left f (f a h) q;;

   let rec fold_right f l b = match l with
     |[] -> b
     |h::q -> f h (fold_right f q b);;
   ```

   Pour rendre **`fold_right`** récursive terminale, il faut renverser la liste. (On admet que **`List.rev`** est récursive terminale, ou on reprend un exo précédent)

   ```
   let rec fold_right f l b =
     let rec aux f l b = match l with
       |[] -> b
       |h::q -> aux f q (f b h)
     in aux f (List.rev) b;;
   ```

3. Le type de **`myst1`** est **`'a -> 'a list -> 'a list`**. **`myst1`** renvoie la liste **`lst`** concaténée avec **`[elt]`**
   Le type de **`myst2`** est **`'a list -> 'a list`**. Il s'agit de l'application partielle de **`fold_right`** au cas où **`f`** est **`fun a b -> a :: b`**.
   Le type de **`myst3`** est **`'a list -> a`**. Cette fonction calcule le minimum de **`lst`**.

4. **`List.map`** prend en entrée une fonction f à un argument et une liste $l = [e_1, ..., e_n]$ et renvoie la liste $[f(e_1), ...f(e_n)]$.
   `fold_right` prend en entrée une fonction de deux arguments. On doit donc construire une fonction **`f'`** qui prend deux arguments en entrée. Ensuite comme **`fold_right`** calculera $f'(e_1, f'(e_2, ...f'(e_n, b)...))$, on sait que **`f'`** doit renvoyer une liste. $f'ab = f(a) :: b$ convient.

   ```
   let map f l =
     let f' a b = (f a)::b in
     fold_right f' l [];;
   ```