Programmation fonctionnelle et logique
TD4 : listes et récursivité

Fonctions sur les listes

Écrire une fonction :

1. qui rend `true` si et seulement si une liste n'a qu'un seul élément.

```
let un_seul_élément = function
   []      -> false
 | t::r::l -> false
 | _       -> true;;
val un_seul_élément : 'a list -> bool = <fun>
```

2. qui renvoie le second élément d'une liste d'au moins deux éléments ;

```
let second_élément = function
   []      -> failwith "second_élément"
 | t::r::l -> r
 | _       -> failwith "second_élément";;
val second_élément : 'a list -> 'a = <fun>
```

3. qui calcule la longueur d'une liste ;

```
let rec longueur = function
    []   -> 0
  | t::r -> 1+ longueur r;;
val longueur : 'a list -> int = <fun>
```

4. qui calcule la somme des éléments d'une liste de nombres ;

```
let rec somme = function
    []   -> 0
  | t::r -> t + somme r;;
val somme : int list -> int = <fun>
```

5. qui concatène deux listes ;

```
let rec concatene l1 l2 =
  match l1 with
    []   -> l2
  | t::r -> t :: concatene r l2 ;;
val concatene : 'a list -> 'a list -> 'a list = <fun>
```

6. qui renverse une liste (écrire une fonction avec et une fonction sans accumulateur) ;

```
let rec renverse = function
    []   -> []
  | t::r -> concatene (renverse r)  [t] ;;
val renverse : 'a list -> 'a list = <fun>
```

```
let renverse l =
  let rec renverse_accu accu = function
      []   -> accu
```

```
    | t::r -> renverse_accu (t::accu) r
  in renverse_accu [] l;;
val renverse : 'a list -> 'a list = <fun>
```

7. qui rend le dernier élément d'une liste non vide ;

```
let rec dernier = function
    []    -> failwith "dernier"
  | t::[] -> t
  | _::r  -> dernier r;;
val dernier : 'a list -> 'a = <fun>
```

8. qui prend une liste et un élément en arguments et renvoie `true` si et seulement si l'élément appartient à la liste ;

```
let rec appartient_a x = function
    []   -> false
  | t::r -> t=x or appartient_a x r;;
val appartient_a : 'a -> 'a list -> bool = <fun>
```

9. qui teste l'égalité de deux listes.

```
let rec egales l1 l2 = match (l1,l2) with
        [],[]     -> true
  | t1::r1,t2::r2 -> t1=t2 & egales r1 r2
  |      _,_      -> false;;
val egales : 'a list -> 'a list -> bool = <fun>
```

Tri par insertion générique

Le tri par insertion est un tri simple : à l'étape i il prend le i^{e} élément de la liste à trier et l'insère à la place idoine dans la liste des $i-1$ élément déjà triés.

1. Écrire une fonction réalisant le tri par insertion.

```
let rec tri_insertion =
let rec insère x = function
    []        -> [x]
  | t::r as l -> if x <= t
                 then x::l
                 else t::(insère x r)
in function
      []  -> []
   | t::r -> insère t (tri_insertion r);;
val tri_insertion : 'a list -> 'a list = <fun>
```

2. Réécrire cette fonction avec des **match** ... **with** et sans **function**.

```
let rec tri_insertion l =
let rec insère l x = match l with
    []        -> [x]
  | t::r as l -> if x <= t
                 then x::l
                 else t::(insère r x)
in match l with
      []  -> []
   | t::r -> insère (tri_insertion r) t;;
val tri_insertion : 'a list -> 'a list = <fun>
```

3. Passer le prédicat de test en paramètre.

```
let rec tri_insertion l comp =
let rec insère l x = match l with
    []        -> [x]
  | t::r as l -> if comp x t
                 then x::l
                 else t::(insère r x)
in match l with
      []  -> []
   | t::r -> insère (tri_insertion r comp) t;;
val tri_insertion : 'a list -> ('a -> 'a -> bool) -> 'a list = <fun>
```

4. Écrire le plus simplement possible une fonction de tri par ordre croissant et une fonction de tri dans ordre décroissant.

```
let tri_croissant l = tri_insertion l (function x -> function y -> x <= y);;
val tri_croissant : 'a list -> 'a list = <fun>
```

```
let tri_décroissant l = tri_insertion l (function x -> function y -> x >= y);;
val tri_décroissant : 'a list -> 'a list = <fun>
```