TD de programmation fonctionnelle et logique
Corrigé du TP 1 : listes et récursivité

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

Fonctions classiques sur les ensembles triés

On considère des ensembles triés représentés par des listes. Écrire des procédures réalisant les fonctions suivantes :

1. Intersection ;

```
# let rec intersection l1 l2 =
   match (l1,l2) with
     t1::r1,t2::r2 -> if t1=t2
                      then t1::(intersection r1 r2)
                      else if t1<t2
                             then (intersection r1 l2)
                             else (intersection l1 r2)
    | _ -> [];;
              val intersection : 'a list -> 'a list -> 'a list = <fun>
```

2. Union ;

```
# let rec union l1 l2 =
   match (l1,l2) with
     t1::r1,t2::r2 -> if t1=t2
                      then t1::(union r1 r2)
                      else if t1<t2
                             then t1::(union r1 l2)
                             else t2::(union l1 r2)
    | [],l2 -> l2
    | l1,[] -> l1;;
                 val union : 'a list -> 'a list -> 'a list = <fun>
```

3. Différence ;

```
# let rec différence l1 l2 =
    match (l1,l2) with
      t1::r1,t2::r2 -> if t1=t2
                       then (différence r1 r2)
                       else if t1<t2
                              then t1::(différence r1 l2)
                              else (différence l1 r2)
    | l1,[] -> l1
    |   _   -> [];;
                 val différence : 'a list -> 'a list -> 'a list = <fun>
```

4. Différence symétrique ;

```
# let différence_symétrique l1 l2 =
    différence (union l1 l2) (intersection l1 l2);;
    val différence_symétrique : 'a list -> 'a list -> 'a list = <fun>
```

5. Produit cartésien (résultat donné dans l'ordre lexicographique).

```
# let rec produit_cartésien l1 l2 =
    match (l1,l2) with
        [t1],t2::r2 -> (t1,t2) :: produit_cartésien [t1] r2
    | t1::r1,t2::r2 -> concatene (produit_cartésien [t1] l2)
                                 (produit_cartésien  r1  l2)
    |   _   -> [];;
          val produit_cartésien : 'a list -> 'b list -> ('a * 'b) list
= <fun>
```