TD de programmation fonctionnelle et logique
Corrigé du TD 3 : récursivité

Suite de Fibonacci

La suite de Fibonacci est définie comme suit :

$$\text{Fib}(n) = \begin{cases} 1 \text{ si } n = 0 \\ 1 \text{ si } n = 1 \\ \text{Fib}(n-1) + \text{Fib}(n-2) \text{ sinon.} \end{cases}$$

1. Écrire une fonction récursive qui calcule $\text{Fib}(n)$.

```
# let rec fib_filtre =
  function
     0 -> 1
   | 1 -> 1
   | n -> (fib_filtre (n-1)) + (fib_filtre (n-2));;
        val fib_filtre : int -> int = <fun>
```

2. Écrire une fonction récursive qui calcule, pour $n > 0$, $(\text{Fib}(n), \text{Fib}(n-1))$. L'utiliser pour écrire une fonction calculant $\text{Fib}(n)$.

```
# let rec fib_paire =
  function
     0 -> failwith "L'argument est nul."
   | 1 -> (1,1)
   | n -> let (x,y)= fib_paire (n-1) in(x+y,x);;
        val fib_paire : int -> int * int = <fun>
```

```
# let fib n = function
     0 -> 1
   | n -> let (x,y)= fib_paire n in x;;
     val fib : 'a -> int -> int = <fun>
```

Calcul de (cos(nx), sin(nx))

Écrire une fonction qui prend en entrée un entier `n` et une paire de valeurs réelles qui sont en fait les valeurs du cosinus et du sinus d'un certain angle `x`, et qui renvoie la paire `(cos(nx), sin(nx))`. Autrement dit, le deuxième argument de la fonction est une paire `(a,b)` telle que `a = cos x` et `b = sin x` . Le schéma de calcul doit bien évidemment être récursif.

On pourra se servir des formules de trigonométrie suivantes :

```
cos(nx) = cos((n-1)x) cos(x) - sin((n-1)x) sin(x)
sin(nx) = sin((n-1)x) cos(x) - cos((n-1)x) sin(x)
```

```
# let trigo (a, b) n =
   let rec trigo_aux = function
       0 -> (1.,0.)
```

```
  | 1 -> (a, b)
  | m -> let (c,d) = trigo_aux (m-1)
          in (a *. c -. b *. d, a *. d +. b *. c)
in trigo_aux n;;
         val trigo : float * float -> int -> float * float = <fun>
```

Fonctions classiques sur les ensembles triés

On considère des ensembles triés représentés par des listes. Écrire des fonctions réalisant les opérations 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 -> List.append (produit_cartésien [t1] l2)
```

```
                                    (produit_cartésien  r1  l2)
    |   _   -> [];;
          val produit_cartésien : 'a list -> 'b list -> ('a * 'b) list
= <fun>
```