Correction de la feuille de TD n°2 : structure de liste

MPSI Lycée Clemenceau : option informatique

Mars 2025

Exercice 1 : Ecrire une fonction `make_list` de signature `int -> 'a -> 'a list` renvoyant une liste de n éléments identiques.

Correction :

```
let rec make_list n a = match n with
        | 0 -> []
        | n -> a::(make_list (n-1) a);;
```

Autre méthode : terminale

```
let make_list2 n a = let rec aux n lst = match n with
                        | 0 -> lst
                        | n -> aux (n-1) (a::lst)
                     in aux n [];;
```

Exercice 2 : Ecrire une fonction qui retourne l'avant-dernier élément d'une liste, s'il existe.

Correction :

```
let rec avantder lst = match lst with
            | [] -> failwith "liste vide"
            | [_] -> failwith "liste trop courte"
            | [a;_] -> a
            | a::q -> avantder q;;
```

Exercice 3 : Ecrire une fonction calculant la somme des éléments (entiers) d'une liste.

Correction : Première version non terminale

```
let rec somme1 lst = match lst with
             | [] -> failwith "liste vide"
             | [a] -> a
             | t::q -> t + somme1 q;;
```

Version récursive terminale

```
let somme lst = match lst with
            | [] -> failwith "liste vide"
            | t::q -> let rec aux l s = match l with
                        | [] -> s
                        | t::q -> aux q (s+t)
             in aux q t;;
```

Exercice 4 : Ecrire une fonction calculant la liste de tous les préfixes d'une liste donnée.

Par exemple :

```
prefixes [1;2;3;4] ;;
- : int list list = [[1];[1;2];[1;2;3];[1;2;3;4]]
```

Correction : Une solution consiste à créer une fonction qui ajoute un élément en tête de toutes les listes contenues dans une liste de liste (cette fonction existe, ou presque, dans les fonctions prédéfinies mais on va s'en passer). Ensuite on construit la liste de listes en partant de la fin.

```
let prefixes lst = match lst with
    | [] -> failwith "list vide"
    | t::q -> let rec ajout a lstlst = match lstlst with
                    | [] -> []
                    | tt::qq -> (a::tt)::(ajout a qq)
              and aux lst = match lst with
                    | [] -> [[]]
                    | [a] -> [[a]]
                    | t::q -> ajout t ([]::(aux q))
            in aux (t::q);;
```

Exercice 5 : Ecrire une fonction `ocsmin : 'a list -> 'a * int = <fun>` qui renvoie le couple formé par l'élément minimum d'une liste et le nombre de fois où il apparaît.

Correction :

la fonction qui suit ne parcourt qu'une seule fois la liste

```
let ocsmin lst = match lst with
        | [] -> failwith "liste vide"
        | a::q -> let rec aux lst e n = match lst with
                        | [] -> (e,n)
                        | a::q ->
                        | a::q when a < e -> aux q a 1
                        | a::q when a = e -> aux q a (n+1)
                        | a::q -> aux q e n
                in aux q a 1;;
```

Exercice 6 : Ecrire une fonction `flatten : 'a list list -> 'a list` aplatissant une liste de listes.

Par exemple, `flatten [[3; 4]; [5;7;9]; [0]; [6;8]]` doit renvoyer `[3; 4; 5; 7; 9; 0; 6; 8]`.

Correction : on met le premier élément de la liste qui est en tête de la liste de listes, en tête de la liste finale dont la queue est le résultat de `flatten` sur la liste de listes dont la tête est la queue de la première liste.

```
let rec flatten lst =  match lst with
                    | [] -> []
                    | []::qlst -> flatten qlst
                    | (a::q)::qlst -> a::(flatten (q::qlst));;
```

Exercice 7 : *Listes et ordre*

1) Ecrire un fonction `est_constante : 'a list -> bool`, qui indique si une liste est constante, c'est à dire tous ses membres sont égaux.

Correction : on choisit de dire que la liste vide est constante mais ce n'est pas une obligation, cela peut être considérée comme une erreur.

```
let rec est_constante lst = match lst with
            | [] -> true
            | [a] -> true
            | a::q -> if a = List.hd q then est_constante q else false;;
```

Autre écriture : lorsque le premier booléen est faux Caml n'évalue pas le second et donc ne fait pas l'appel récursif

```
let rec est_constante lst = match lst with
            | [] -> true
            | [a] -> true
            | a::q ->(a =List.hd q) && (est_constante q);;
```

2) Ecrire une fonction `est_croissante` et une fonction `est_decroissante` qui indiquent si la liste transmise en argument est respectivement croissante ou décroissante au sens large.

Correction :

```
let rec est_croissante lst = match lst with
            | [] -> true
            | [a] -> true
            | a::q -> if a <= List.hd q then est_croissante q else false;;

let rec est_decroissante lst = match lst with
            | [] -> true
            | [a] -> true
            | a::q -> if a >= List.hd q then est_decroissante q else false;;
```

On peut aussi faire comme pour le cas constant :

```
let rec est_croissante lst = match lst with
            | [] -> true
            | [a] -> true
            | a::q -> (a <= List.hd q)&& (est_croissante q);;

let rec est_decroissante lst = match lst with
            | [] -> true
            | [a] -> true
            | a::q -> (a >= List.hd q)&& (then est_decroissante q);;
```

3) Utiliser la question précédente pour écrire une fonction `est_monotone` qui spécifie si la liste est monotone au sens large.

Correction : immédiat

```
let rec est_monotone lst = match lst with
            | [] -> true
            | [a] -> true
            | a::b::q when a<b -> est_croissante(b::q)
            | a::b::q when a>b -> est_decroissante(b::q)
            | _::q -> est_monotone q ;;
```

4) Ecrire une fonction `est_monotone2` qui n'utilise pas la question 2 et qui spécifie si la liste est monotone au sens large en ne parcourant qu'au maximum qu'une fois la liste.

Correction : on utilise deux booléens qui permettent de juger la monotonie

```
let est_monotone2 lst =
        let rec aux up dn lst = match lst with
               | [] -> true
               | [_] -> up or dn
               | a::b::q when a<b -> aux up false  (b::q)
               | a::b::q when a>b -> aux  false dn (b::q)
               | _::q -> aux up dn q
          in aux true true lst ;;
```

Autre méthode sans parcours complet de la liste : on regarde l'état des booléens au fur et à mesure

```
let est_monotone2 lst =
        let rec aux up dn lst = match (lst,up,dn) with
               | ([],_,_) -> true
               | ([_],_,_) -> up or dn
               | (a::b::q,false,_) when a<b -> false
               | (a::b::q,true,_) when a<b -> aux up false  (b::q)
               | (a::b::q,_,false) when a>b -> false
               | (a::b::q,_,true) when a>b -> aux  false dn (b::q)
               | (_::q,_,_) -> aux up dn q
        in aux true true lst ;;
```

Une autre méthode utilisant une fonction auxiliaire : elle utilise un test donc le signe ne doit pas changer

```
let est_monotone3 lst = match lst with
           | [] -> true
           | [a] -> true
           | a::b::q -> let rec aux test lst = match lst with
                        | [] -> true
                        | [c] -> true
                        | c::d::q when (d-c)*test>=0 -> aux ((d-c)+test) (d::q)
                        | _ -> false
                      in aux (b-a) (b::q);;
```

Exercice 8 : Ecrire une fonction `lfactors: int -> int list` renvoyant la liste des facteurs premiers d'un entier positif n (chaque facteur étant éventuellement répété autant que nécessaire).

Correction : on utilise une fonction auxiliaire qui donne le premier diviseur premier, c'est à dire le plus diviseur différent de 1, d'un nombre. Une fois trouver on le met en tête de la liste qui contient les diviseurs premiers du quotient de n avec ce nombre.

Dans la fonction auxiliaire on utilise un booléen qui compare le carré du diviseur potentiel avec n car il est nécessairement plus petit que la racine carrée de n.

```
let lfactor n = let rec prem n k bo = if bo then match n mod k with
                          | 0 -> k
                          | _ -> prem n (k+1) ((k+1)*(k+1)<n) else n in
                  let rec aux n k lst = match n with
                          | 1 -> lst
                          | n -> let a = prem n k  true in  a :: aux (n/a) a lst
                in aux n 2 [];;
```

Exercice 9 : On souhaite écrire une fonction `purge` qui, appliquée à une liste, retourne une liste dans laquelle les doublons ont été éliminés (on pourra utiliser la fonction prédéfinie `List.mem` qui détermine si un élément appartient ou pas à une liste, on fera ensuite une version qui redéfinit cette fonction).

1) Écrire une première version de purge dans laquelle seule la dernière occurrence de chaque doublon sera conservée. Par exemple, `purge [1; 2; 3; 1; 4; 3; 1]` renverra comme résultat `[2; 4; 3; 1]`.

Correction : Sachant que le coût de la fonction mem est linéaire, le coût de la première fonction est quadratique dans le pire des cas (lorsque la liste ne contient aucun doublon).

```
let rec purge lst = match lst with
| [] -> []
| t::q when List.mem t q -> purge q
| t::q  -> t::(purge q) ;;
```

2) Écrire une seconde version de purge dans laquelle seule la première occurrence de chaque doublon sera conservée. Cette fois, `purge [1; 2; 3; 1; 4; 3; 1]` renverra le résultat `[1; 2; 3; 4]`.

Correction : La seconde version utilise une fonction auxiliaire pour éliminer les occurrences d'un élément dans une liste.

```
let  purge2 lst =
       let rec elimine a lst = match lst with
                    | [] -> []
                    | t::q when t = a -> elimine a q
                    | t::q   -> t::(elimine a q)
       in
       let rec aux lst =
                match lst with
                     | [] -> []
                     | t::q -> t::(elimine t (aux q))
       in aux lst ;;
```

Exercice 10 : On peut définir une fonction qui teste des éléments devant vérifier une propriété si celle-ci est de signature `'a -> bool`.

1) Ecrire une fonction de signature `('a -> bool) -> int -> 'a list -> 'a`, qui donne le $n$ième élément d'une liste vérifiant une certaine propriété (s'il existe).

Correction : il suffit de parcourir la liste en diminuant la valeur de n

```
let rec testeliste f n lst = match (n,lst) with
            | _ , [] -> failwith "n'existe pas"
            | 1 , a::q when f a -> a
            | n , a::q when f a -> testeliste f (n-1) q
            | n , a::q -> testeliste f n q;;
```

2) Ecrire une fonction de signature `('a -> bool) -> 'a list -> 'a` qui donne le dernier élément d'une liste vérifiant une certaine propriété (s'il existe).

Correction : on parcourt la liste et on met dans l'accumulateur la valeur qui vérifie la propriété (une liste d'un seul élément s'il existe).

```
let dernier f lst = let rec aux lst acc = match lst with
                                | [] -> acc
                                | a::q when f a -> aux q [a]
                                | a::q -> aux q acc
                            in
                            match (aux lst []) with
                                | [] -> failwith "pas d'éléments"
                                | r -> List.hd r;;
```

Exercice 11 : On représente un ensemble par une liste, chaque élément de l'ensemble ne devant apparaître qu'une seule fois dans la liste, à un emplacement arbitraire.

1) Définir une fonction intersection qui calcule l'intersection de deux ensembles. Évaluer son coût en fonction des cardinaux de ces ensembles.

Correction : on utilise la fonction prédéfinie `List.mem` qu'on pourrait redéfinir. Sa complexité étant linéaire.

```
let rec inter lst1 lst2 = match lst1 with
            | [] -> []
            | a::q when List.mem a lst2 -> a::(inter q lst2)
            | a::q -> inter q lst2;;
```

2) Définir de même l'union et la différence symétrique de deux ensembles.

Correction :

```
let rec union lst1 lst2 = match lst1 with
            | [] -> lst2
            | a::q when List.mem a lst2 -> union q lst2
            | a::q -> a::(union q lst2);;
```

3) Rédiger enfin une fonction égal qui détermine si deux ensembles sont égaux.

Correction :

```
let  diff lst1 lst2 = let rec suppr a lst = match lst with
| [] -> []
| b::q when b = a -> q (* a n'est qu'une fois dans la liste*)
| b::q -> b::(suppr a q)
```

```
in
let rec aux lst1 lst2 = match lst1 with
| [] -> lst2
| a::q when List.mem a lst2 -> aux q (suppr a lst2)
| a::q -> a::(aux q lst2)
in aux lst1 lst2;;
```

Exercice 12 : On représente un polynôme par une liste de type `( float * int ) list`. Chaque élément de la liste représente un monôme, ceux-ci sont rangés dans la liste par ordre de degré croissant.
Ainsi le polynôme $\frac{3}{2}X^7 - 4X^3 + 5X^2$ est représenté par la liste `[ (5.,2) ; (4.,3) ; (1.5,7) ]`.

1) Quel est l'intérêt d'une telle représentation, par rapport aux tableaux vus lors du TD 1 ?
Correction : cela prend moins de place en mémoire lorsqu'il y a peu de coefficients non nuls.

2) Définir les fonctions somme et produit pour cette représentation.
Correction : On utilise la déclaration de types suivante : on définit les monômes qui seront utiles pour les fonctions

```
type 'a monome = int * 'a;;
type 'a polynome = ('a monome) list;;
```

On commence par l'addition. On commence par le cas de polynômes nuls. L'avantage d'avoir le monôme de plus haut degré en tête permet d'étudier le cas de la simplification lorsqu'on somme deux polynômes de même degré et de coefficients dominants opposés.

```
let rec add (p1 : int polynome) p2 = match (p1,p2) with
        | [], _ -> p2
        | _,[] -> p1
        | ((d1,a1)::r1),((d2,a2)::r2) ->
             if d1 < d2 then (d1,a1)::(add r1 p2)
             else if d2 < d1 then (d2,a2)::(add p1 r2)
                  else if (a1+a2) <> 0 then (d1,a1+a2)::(add r1 r2)
                       else add r1 r2;;
```

Ensuite on définit la multiplication d'un monôme par un polynôme. Ce qui permet de définir enfin la multiplication de deux polynômes.
Remarque : dans une sujet de concours il faudrait mettre la première fonction comme fonction auxiliaire de la multiplication générale.

```
let rec mulmono m (p : int polynome) = match p with
        | [] -> []
        | (d,a)::q -> (d + fst m, snd m * a)::(mulmono m q);;


let rec mult p1 p2 = match p1 with
        | [] -> []
        | m::q -> let p = mulmono m p2 and r = mult q p2 in add p r;;
```