Université Paris Diderot — Programmation Fonctionnelle
Licence — Année 2010-2011

TP n 2 - Correction

Listes

Les expressions suivantes sont des listes :
– liste vide : `[]`
– liste composée de 3 éléments : `[42;37;15]`

Les éléments d'une liste doivent être de même type. L'opérateur `::` permet d'ajouter un élément en tête d'une liste. L'opérateur `@` permet de concaténer deux listes. Les listes suivantes sont par exemple égales :

```
[42;37;15]                  42::[37;15]              42::(37::[15])
42::(37::(15::[]))          42::37::15::[]           42::37::[15]
                            [42;37]@[15]
```

La construction suivante permet de définir une fonction en raisonnant par cas sur la forme d'une liste :

```
let rec f l = match l with
| []    -> ...   (* expression associée si l est vide *)
| a::l' -> ...   (* expression associée si l est non vide.  *)
;;               (* a désigne le 1er élément de l, et l'    *)
                 (* la liste l privée de son 1er élément.    *)
```

Autre exemple :

```
let rec f l = match l with
| []      -> ...
| [a]     -> ...   (* cas où l contient un seul élément        *)
| a::b::l'-> ...   (* cas où l contient au moins deux éléments *)
;;
```

Remarque. À l'évaluation, les différents cas d'un `match` seront examinés successivement. La suite des cas peut ne pas couvrir toutes les formes possibles : dans ce cas, la fonction définie est dite *partielle* : elle déclenche une exception si son argument n'est d'aucune des formes mentionnées.

1 Listes

Exercice 1 Écrire les fonctions suivantes sur les listes. Les fonctions précédées de sont prédéfinies en Caml (elles s'écrivent `List.length`, `List.rev`, etc) et le but est de retrouver leur implémentation.

– `list_sigma` : calcule la somme des éléments d'une liste d'entiers.
Exemple : `list_sigma [1;2;3] = 6`.
Correction :

```
let rec list_sigma l = match l with
| [] -> 0
| h::l -> h + list_sigma l;;

let list_sigma = List.fold_left (+) 0;;
```

`map` : la fonction telle que `map f [a1 ;... ;an] = [(f a1) ;... ;(f an)]`.
Correction :

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

`mem` : déterminer si une liste contient une valeur donnée.
Exemple : `mem 27 [12 ;27 ;1] = true`, `mem 3 [12 ;27 ;1] = false`.
Correction :

```
let rec mem a l = match l with
| [] -> false
| h::l -> h = a || mem a l;;
```

– `liste_min` : calcule le minimum d'une liste d'entiers non vide.
Exemple : `liste_min [-30 ;2 ;549] = -30`.
Correction :

```
let rec liste_min l = match l with
| [] -> failwith "Pas de minimum !"
| [a] -> a
| h::l -> min h (liste_min l);;
```

`append` : concatène deux listes – cette fonction ne doit pas utiliser l'opérateur `@`.
Exemple : `append [1 ;2 ;3] [4 ;5] = [1 ;2 ;3 ;4 ;5]`.
Correction :

```
let rec append l1 l2 = match l1 with
| [] -> l2
| h::l -> h :: (append l l2);;
```

`rev` : inverse l'ordre des éléments d'une liste.
Exemple : `rev [1 ;2 ;3] = [3 ;2 ;1]`.
Correction :

```
let rec rev l = match l with
| [] -> []
| h::l -> (rev l) @ [h];;

let rev l =
  let rec aux l acc = match l with
  | [] -> acc
  | h :: l -> aux l (h :: acc)
  in aux l [];;
```

`flatten` : aplanir d'un niveau une liste de listes.
Exemple : `flatten [[2] ;[] ;[3 ;4 ;5]] = [2 ;3 ;4 ;5]`.
Correction :

```
let rec flatten l = match l with
| [] -> []
```

```
| h::l -> append h (flatten l);;
```

– `is_sorted` : détermine si une liste est triée.
Exemple : `is_sorted [1 ;3 ;5 ;6 ;4] = false`.
Correction :

```
let rec is_sorted l = match l with
| [] | _::[] -> true
| a::b::l -> a <= b && is_sorted (b::l);;
```

– Question subsidiaire : `moyenne`, qui calcule la moyenne des éléments d'une liste (potentiellement vide) d'entiersnon vide. Exemple : `moyenne [1 ;2 ;3] = 2`.
Correction :

```
let moyenne l = list_sigma l / (List.length l)

let moyenne l =
  let rec aux l = match l with
  | [] -> 0, 0
  | h::l -> let somme, taille = aux l in
            somme+h, taille+1
  in
  let somme, taille = aux l in
  somme / taille
```

Exercice 2 On supposera que les fonctions suivantes attendent une liste dont les éléments sont d'un type muni d'un ordre total.

– Écrire une fonction `insert` qui insère un élément `x` à la bonne place dans une `'a list` supposée triée par ordre strictement croissant. Si `x` est déjà dans la liste, laissera la liste telle quelle.
Correction :

```
let rec insert a l = match l with
| [] -> [a]
| h::_ when h = a -> l
| h::l when h > a -> a::h::l
| h::l -> h :: (insert a l);;
```

– Écrire une fonction `sort` qui trie une `'a list` quelconque par ordre croissant, en fusionnant les doublons.
Correction :

```
let rec sort l = match l with
| [] -> []
| h::l -> insert h (sort l);;
```

– Écrire une fonction `mem_sorted` se comportant comme `mem` sur une liste triée, mais s'évaluant en un nombre minimum d'étapes.
Correction :

```
let rec mem_sorted a l = match l with
| [] -> false
| h::_ when h > a -> false
| h::_ when h = a -> true
| _::l -> mem_sorted a l;;
```

– Question subsidiaire : Écrire les opérations d'union et d'intersection (`union_sorted`, `inter_sorted`) de deux listes triées.
Correction :

```
let rec union_sorted l1 l2 = match l1,l2 with
| _,[] -> l2
| [],_ -> l1
| a1::l1',a2::l2' ->
    if a1 < a2 then a1::(union_sorted l1' l2 ) else
    if a2 < a1 then a2::(union_sorted l1  l2') else
    a1::(union_sorted l1' l2');;

let rec inter_sorted l1 l2 = match l1,l2 with
|  _,[] -> []
| [],_ -> []
| a1::l1',a2::l2' ->
    if a1 < a2 then inter_sorted l1' l2  else
    if a2 < a1 then inter_sorted l1  l2' else
    a1::(inter_sorted l1' l2');;
```

2 Vecteurs

On considère les vecteurs à coordonnées entières, représentés en Caml par des listes d'entiers : `type vecteur = int list ; ;`

Exercice 3 [Norme] Écrire une fonction `norme` qui calcule la norme d'un vecteur, c'est-à-dire la racine carrée de la somme de ses coordonnées.

Exemple : `norme [1 ;2 ;3] =` $\sqrt{1^2+2^2+3^2}$.

On se servira des fonctions prédéfinies `sqrt` (racine carrée sur les `float`), et `float_of_int` (conversion d'un entier en flottant).
Correction :

```
let norme v =
  let rec aux v = match v with
  | [] -> [0]
  | h::l -> (h*h) :: (aux l)
  in let somme_carree = list_sigma (aux v)
  in sqrt (float_of_int somme_carree);;

let norme v =
  let somme_carree = List.fold_left (fun acc x ->  acc + x*x) 0 v
  in sqrt (float_of_int somme_carree);;

let norme v =
  let somme_carree = list_sigma (List.map2 ( * ) v v)
  in sqrt (float_of_int some_carree);;
```

Exercice 4 [Produit scalaire] Écrire une fonction `produit_scalaire` prenant en argument deux vecteurs et renvoyant leur produit scalaire. Rappelons que le produit scalaire de `[x_1 ; . . . ; x_n]` et `[y_1 ; . . . ; y_n]` vaut `x_1 * y_1 +...+ x_n * y_n`, et n'est calculable que si les deux

vecteurs sont de même longueur. Si les deux vecteurs fournis ne sont pas de même longueur, la fonction devra interrompre son exécution en renvoyant un message d'erreur, à l'aide d'une expression de la forme `(failwith "erreur !")`.
Correction :

```
let produit_sclaire v1 v2 =
  if List.length v1 = List.length v2 then
    List.fold_left (+) 0 (List.map2 ( * ) v1 v2)
  else failwith "erreur!";;
```

3 Matrices

On considère à présent les matrices d'entiers, représentées à l'aide du type suivant :
`type matrice = vecteur list ; ;`

Exercice 5 [Validation] Une matrice est dite bien formée si tous les vecteurs qui la composent sont de même taille. Écrire une fonction `matrice_valide` qui renvoie `true` si la matrice donnée en argument est bien formée, et `false` sinon.

Correction :

```
let rec matrice_valide m = match m with
| [] | _::[] -> true
| a::b::m -> (List.length a = List.length b) && matrice_valide (b::m);;
```

Exercice 6 [Matrice carrée] Une matrice est dite carrée si le nombre de ses lignes est égal au nombre de ses colonnes. Écrire une fonction `matrice_carrée` qui renvoie `true` si la matrice est carrée et `false` sinon.

Correction :

```
let matrice_carree m = match m with
| [] -> true
| h::_ -> matrice_valide m && List.length m = List.length h;;
```

Exercice 7 [Somme] La somme de deux matrices m_1 et m_2 de même taille est définie comme la matrice ayant pour éléments la somme des éléments de m_1 et m_2, case par case. Écrire une fonction `somme_matrice` prenant en arguments deux matrices de même taille et renvoyant leur somme.

Correction :

```
let somme_matrice m1 m2 = List.map2 (List.map2 (+)) m1 m2
```

Exercice 8 [Transposée] Soit m une matrice à n lignes et p colonnes. On appelle transposée de m la matrice à p lignes et n colonnes, dont la ligne n^o i est égale à la colonne n^o i de m. Écrire la fonction `tranposée` renvoyant la transposée d'une matrice.

Exemple : `transposée [[1 ; 3 ; 5] ; [2 ; 4 ; 6]] = [[1 ; 2] ; [3 ; 4] ; [5 ; 6]]`

Correction :

```
(* Il faut d'abord couper la matrice en deux parties: les premiers
éléments des vecteurs et le reste *)
let rec split = function
  | [] -> [], []
  | []::l -> split l
  | (h::[])::l ->
    let hs, ls = split l
    in (h :: hs), ls
  | (h::k)::l ->
    let hs, ls = split l
    in (h :: hs), k :: ls;;

(* Une fois la matrice coupée, il suffit d'itérer jusqu'à la fin *)
let rec transposee m = match m with
  | [] -> []
  | h::l ->
    let hs, ls = split m
    in hs :: transposee ls;;

(* Une deuxième version plus courte et (2 fois) plus efficace *)
let rec transposee m = match m with
  | [l] -> List.map (fun e -> [e]) l
  | l :: s ->
     let s = transposee s
     in List.map2 (fun e l -> e :: l) l s
  | [] -> [];;
```

Exercice 9 [Produit] Soit deux matrices m_1 (à n_1 lignes et p_1 colonnes) et m_2 (à n_2 lignes et p_2 colonnes). Le produit de m_1 par m_2 n'est possible que si $p_1 = n_2$ (le nombre de colonnes de m_1 est égal au nombre de lignes de m_2, *i.e.* le produit scalaire d'une ligne de m_1 et d'une colonne de m_2 est toujours défini). La matrice résultante aura n_1 lignes et p_2 colonnes. Le produit se calcule en mettant dans chaque cellule (i, j) le résultat du produit scalaire du vecteur de la ligne n$^{\text{o}}$ i de la première matrice par le vecteur de la colonne n$^{\text{o}}$ j de la deuxième matrice.

Écrire la fonction `produit_matrice` calculant le produit de deux matrices données en arguments lorsque ce produit est calculable.

Correction :

```
let produit_matrice m1 m2 =
  let m2 = transposee m2 in
  List.map
    (fun v1 -> List.map (produit_scalaire v1) m2)
    m1;;
```