Université Paris Diderot — Programmation Fonctionnelle
L2 Math-Info/L3 Informatique/M1 Linguistique — Année 2011-2012

TP n°3

Vecteurs, matrices et itérateurs

Appelons *vecteur de dimension n* tout n-uplet de la forme $(x_1, \ldots, x_n)$. Les $x_1, \ldots, x_n$ seront appelées *composantes* de ce vecteur. On s'intéresse ici au cas où les composantes des vecteurs sont des entiers.

On définit les opérations suivantes sur les vecteurs de même dimension :
- La *somme* de $(x_1, \ldots, x_n)$ et $(y_1, \ldots, y_n)$ est le vecteur $(x_1 + y_1, \ldots, x_n + y_n)$.
- La *différence* de $(x_1, \ldots, x_n)$ et de $(y_1, \ldots, y_n)$ est le vecteur $(x_1 - y_1, \ldots, x_n - y_n)$.
- La *produit scalaire* de $(x_1, \ldots, x_n)$ et de $(y_1, \ldots, y_n)$ est l'entier $(x_1 \times y_1) + \ldots + (x_n \times y_n)$.

D'autre part, pour tout vecteur $(x_1, \ldots, x_n)$,
- Le *produit externe* de l'entier (signé) k et de $(x_1, \ldots, x_n)$ est le vecteur $(k \times x_1, \ldots, k \times x_n)$,
- La *norme* de $(x_1, \ldots, x_n)$ est le réel $\sqrt{(x_1 \times x_1) + \ldots + (x_n \times x_n)}$. Noter qu'elle est égale à la racine carrée du produit scalaire de $(x_1, \ldots, x_n)$ par lui-même.

Le but des exercices ci-dessus est de représenter ces vecteurs en Caml, et d'implémenter en particulier toutes les opérations précédentes. La première partie considère le cas simplifié de vecteurs de dimension 3, représentés par des enregistrements. La seconde considère des vecteurs de dimensions arbitraires, représentés par des listes d'entiers.

1 Vecteurs de dimension 3

Les vecteurs de dimension 3 peuvent être représentes à l'aide du type d'enregistrement

```
type vect3D = {
   vx : int; (* premiere composante *)
   vy : int; (* seconde   *)
   vz : int  (* troisieme *)
}
```

Exercice 1 Ecrire les fonctions suivantes, implémentant pour les vecteurs de dimension 3 les opérations présentées ci-dessus :

- `fois3D : int -> vect3D -> vect3D` (produit externe)
- `plus3D : vect3D -> vect3D -> vect3D` (somme)
- `diff3D : vect3D -> vect3D -> vect3D` (différence)
- `scal3D : vect3D -> vect3D -> int` (produit scalaire)
- `norm3D : vect3D -> float` (norme)

Noter que la différence peut être calculée à l'aide de la somme et du produit externe. Pour la norme (qui peut se resservir de `scal3D`), il vous faudra la fonction de conversion `float_of_int`, et la fonction (prédéfinie) racine carrée `sqrt : float -> float`.

2 Vecteurs de dimension arbitraire

Les vecteurs de dimensions quelconques peuvent être représentés par des listes d'entiers :

```
type vect = int list
```

2.1 Implémentations de base des opérations sur les vecteurs

Exercice 2 Ecrire les fonctions suivantes implémentant les opérations présentées ci-dessus. Chaque fonction devra être indépendante des autres, et ne pas se servir de fonctions prédéfines. Pour la norme, définissez une fonction auxiliaire locale.

- `fois : int  -> vect -> vect` (produit externe)
- `plus : vect -> vect -> vect` (somme)
- `diff : vect -> vect -> vect` (différence)
- `scal : vect -> vect -> int` (produit scalaire)
- `norm : vect -> float` (norme)

Les opérations somme, différence et produit scalaire de deux vecteurs supposerons que leurs deux arguments sont des listes de mêmes longueurs.

2.2 Optimisations du code à l'aide des itérateurs `map` et `map2`

Rappelons que la fonction prédéfinie `List.map` permet d'appliquer une fonction à chacun des éléments d'une liste :

`List.map f [`x_1`;...;`x_n`] = [(f `x_1`);...;(f `x_n`)]`

La fonction prédéfinie `List.map2` considère quant à elle *deux* listes, impérativement de même longueur, et applique une fonction `f` aux éléments des deux listes de même rangs :

`List.map2 f [`x_1`;...;`x_n`] [`y_1`;...;`y_n`] = [(f `x_1 y_1`);...;(f `x_n y_n `)]`

D'autre part, les opérateurs arithmétiques de Caml peuvent être utilisés comme des fonctions ordinaires, avec la notation préfixe habituelle des fonctions : il suffit de les entourer de parenthèses (en ajoutant des espaces autour de `*` – pourquoi ?).

```
(+) x x' = x + x'
(-) x x' = x - x'
( * ) x x' = x * x'
...
```

En particulier, `(+)`, `(-)`, `( * )` ...peuvent être passés à `List.map2` comme des fonctions ordinaires.

Exercice 3 Ecrire une nouvelle implémentation de la fonction `fois` utilisant la fonction `List.map`, et déléguant entièrement la récurrence à cette fonction.

Ecrire de même deux nouvelles implémentations des fonctions `plus` et `diff` utilisant la fonction prédéfinie `List.map2`, et les versions préfixes des opérateurs `+` et `-` .

2.3 Matrices

Appelons *matrice à n lignes et p colonnes* tout n-uplet de p-uplets d'entiers $((x_{11}, \ldots, x_{1p}), \ldots, (x_{n1}, \ldots, x_{np}))$. Une telle matrice se représente généralement de la manière suivante :

$$\begin{pmatrix} x_{11} & \ldots & x_{1p} \\ \vdots & & \vdots \\ x_{n1} & \ldots & x_{np} \end{pmatrix}$$

Le vecteur $(x_{i1}, \ldots, x_{ip})$ est appelée *i-ième ligne* de cette matrice. Le vecteur $(x_{1j}, \ldots, x_{pj})$ est appelée sa *j-ième colonne*. On souhaite représenter les matrices en Caml comme des listes de vecteurs. Chaque matrice sera représentée par la liste de ses lignes successives :

```
type matrice = vecteur list;;
```

Exercice 4 Les listes de vecteurs ne représentent pas toutes des matrices : pour qu'une liste réprésente effectivement une matrice, il faut que tous ses éléments soient de même longueur. Écrire une fonction `matrice_valide : matrice -> bool` renvoyant `true` lorsque son argument représente bien une matrice, et `false` sinon. Cette fonction peut se servir de `List.length`.

Exercice 5 [⋆] Trouver une seconde implémentation de `matrice_valide` à l'aide de `List.length`, de `List.hd` et `List.tl` (ou d'un `match`), et de la fonction `List.for_all` :

`List.for_all : ('a -> bool) -> 'a list -> bool`

`List.for_all p [x1;...;xn] = (p x1) && ... && (p xn)`

Toutes les fonctions qui suivent supposeront que leurs arguments de type `matrice` représentent effectivement des matrices.

Exercice 6 Une matrice est dite carrée si le nombre de ses lignes est égal au nombre de ses colonnes. Écrire une p fonction `matrice_carree : matrice -> bool` renvoyant `true` si son argument est carrée et `false` sinon.

Exercice 7 La somme de deux matrices de même taille est définie comme la matrice ayant pour éléments la somme des éléments des matrices, position par position :

$$\begin{pmatrix} x_{11} & \ldots & x_{1p} \\ \vdots & & \vdots \\ x_{n1} & \ldots & x_{np} \end{pmatrix} + \begin{pmatrix} y_{11} & \ldots & y_{1p} \\ \vdots & & \vdots \\ y_{n1} & \ldots & y_{np} \end{pmatrix} = \begin{pmatrix} x_{11}+y_{11} & \ldots & x_{1p}+y_{1p} \\ \vdots & & \vdots \\ x_{n1}+y_{n1} & \ldots & x_{np}+y_{np} \end{pmatrix}$$

A l'aide de `List.map2` et de la fonction `plus` de l'exercice 2, ou même uniquement à l'aide de `List.map2` (utilisée deux fois) et de `(+)`, écrire une fonction `somme_matrices` prenant en arguments deux matrices et renvoyant leur somme. Cette fonction supposera que ses deux arguments sont de même taille.

Exercice 8 [⋆] Etant donnée une matrice M à n lignes et p colonnes, la *transposée* de M est la matrice à p lignes et n colonnes, dont la ligne n^o i est égale à la colonne n^o i de M.

$$\begin{pmatrix} x_{11} & \ldots & x_{1p} \\ \vdots & & \vdots \\ x_{n1} & \ldots & x_{np} \end{pmatrix} \text{ se transpose en } \begin{pmatrix} x_{11} \ldots x_{n1} \\ \vdots \quad\quad \vdots \\ x_{1p} \ldots x_{np} \end{pmatrix}$$

Écrire la fonction `tranposee` renvoyant la transposée d'une matrice.

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

Cette fonction peut librement utiliser `List.map` et `List.map2`

3 Pliages

3.1 Pliage gauche simple

La fonction prédéfinie `List.fold_left` considère une fonction `f`, une valeur `a`, et une liste $[x_1;\ldots;x_n]$. Pour calculer `List.fold_left f a` $[x_1;\ldots;x_n]$, la fonction part de a_0 = `a`, puis calcule successivement pour chaque x_i un nouveau a_i = `(f` a_{i-1} x_i`)` :

(0) a_0 = `a`

(1) a_1 = `(f` a_0 x_1`)`

$\vdots$

(n) a_n = `(f` a_{n-1} x_n`)`

La valeur renvoyée est alors a_n. Le code de cette fonction est donc :

```
let rec fold_left f a l = match l with
  [] -> a
| x::l' -> f (fold_left f a l') x;;
```

Exercice 9 Montrer comment utiliser la fonction `List.fold_left` et la version préfixe de `+` (voir ci-dessus) pour écrire une fonction `sigma : int list -> int` renvoyant la somme des éléments d'une liste d'entiers - la récurrence devra bien sûr être déléguée à `List.fold_left`.

Exercice 10 En vous inspirant de la fonction précédente, et en combinant `List.fold_left`, `(+)`, `List.map2` et `(*)`, donner une nouvelle implémentation de la fonction `scal` de l'exercice 2.

3.2 Pliage gauche double

La fonction prédéfinie `List.fold_left2` considère une fonction `f`, une valeur `a`, et *deux* listes de même longueur $[x_1;\ldots;x_n]$ et $[y_1;\ldots;y_n]$.

Pour calculer `List.fold_left2 f a` $[x_1;\ldots;x_n]$ $[y_1;\ldots;y_n]$, la fonction part de a_0 = `a`, puis calcule successivement pour chaque paire x_i, y_i un nouveau a_i = `(f` a_{i-1} x_i y_i `)` :

(0) a_0 = `a`

(1) a_1 = `(f` a_0 x_1 y_1`)`

$\vdots$

(n) a_n = `(f` a_{n-1} x_n y_n`)`

La valeur renvoyée est alors a_n. Le code de cette fonction (avec, à la troisième ligne, un *lancement d'exception* atteint lorsque les listes ne sont pas de même longueur) est donc :

```
let rec fold_left2 f a l1 l2 = match l1,l2 with
  [],[] -> a
| x1::l1', x2::l2' -> f (fold_left2 f a l1' l2') x1 x2
| _ -> raise Invalid_argument;;
```

Exercice 11 Trouver une nouvelle implémentation de la fonction `scal` de l'exercice 2 déléguant entièrement la récurrence à la seule fonction `List.fold_left2`.