Devoir de programmation fonctionnelle et logique

Produits scalaires de vecteurs

Travail demandé

Ce travail est à effectuer seul ou en binôme, à l'exclusion de tout regroupement de taille supérieure. Pour le mercredi 30 mai vous devrez m'envoyer par mail (adresse : vivien@icps.u-strasbg.fr) un fichier contenant le code de toutes vos fonctions. Vous n'avez pas à me rendre de rapport mais, si besoin est, vous pouvez insérer des commentaires dans votre fichier (syntaxe `(* *)`), en plus de vos noms et prénoms...

La date du mercredi 30 mai ne pourra en aucun cas être dépassée : à partir du lendemain, vous pourrez retirer au secrétariat une annale contenant les corrigés de tous les TDs et TPs, de ce devoir, et de certains des examens précédents.

Vous tâcherez de proposer des solutions les plus simples et les plus « fonctionnelles » possibles. À toutes fins utiles, je vous rappelle que les corrigés des TDs et TPs sont disponibles à l'url : `http://icps.u-strasbg.fr/~vivien/Enseignement/PFEL-2000-2001/`.

Vecteurs d'entiers

Dans tout ce devoir, nous allons représenter les vecteurs par des listes. Ainsi, le vecteur à trois dimensions de coordonnées 1, 3 et 5, sera représenté par la liste : `[1;3;5]`.

Pour l'instant nous ne manipulons que des vecteurs d'entiers.

1. Écrivez une fonction prenant en entrée deux vecteurs et retournant le produit scalaire de ces vecteurs.

```
# (* test de commentaires *)
 let rec produit_scalaire x y =
   match x,y with
     a::b,c::d -> a*c + (produit_scalaire b d)
   |  [] , []  -> 0
   |     _     -> failwith "Vecteurs de tailles différentes";;
val produit_scalaire : int list -> int list -> int = <fun>
```

   Type et exemple d'utilisation :

```
# produit_scalaire;;
- : int list -> int list -> int = <fun>

# produit_scalaire [1;3;5] [2;-6;1];;
- : int = -11
```

2. Écrivez une fonction prenant en entrée un vecteur `v` et une liste `l` de vecteurs, et renvoyant la liste des produits scalaires de `v` avec les éléments de `l`.

   (a) Écrivez une version récursive de cette fonction.

```
# let rec produit_vec_liste v = function
      []  -> []
    | t::r -> (produit_scalaire t v)::(produit_vec_liste v r);;
val produit_vec_liste : int list -> int list list -> int list = <fun>
```

   (b) Écrivez une version de cette fonction utilisant une ou des fonctionnelles prédéfinies, mais pas la récursivité.

```
# let produit_vec_liste v l =
      List.map (function x -> produit_scalaire x v) l;;
```

```
val produit_vec_liste : int list -> int list list -> int list = <fun>
```

```
# let produit_vec_liste v l = List.map (produit_scalaire v) l;;
val produit_vec_liste : int list -> int list list -> int list = <fun>
```

Type et exemple d'utilisation :

```
# produit_vec_liste;;
- : int list -> int list list -> int list = <fun>
```

```
# produit_vec_liste [1;3;5] [[2;-6;1]; [7;1;-2]; [-4;2;4]];;
- : int list = [-11; 0; 22]
```

3. Écrivez une fonction prenant en entrée deux listes `l` et `ll` de vecteurs, et renvoyant la liste des produits scalaires des éléments de `l` et de `ll`.
   (a) Écrivez une version récursive de cette fonction.

```
# let rec produit_vec_vec l = function
      []  -> []
    | t::r -> (produit_vec_liste t l)@(produit_vec_vec l r);;
val produit_vec_vec : int list list -> int list list -> int list = <fun>
```

```
# produit_vec_vec [[1;3;5];[1;0;1]] [[2;-6;1]; [7;1;-2]; [-4;2;4]];;
- : int list = [-11; 3; 0; 5; 22; 0]
```

   (b) Écrivez une version de cette fonction utilisant une ou des fonctionnelles prédéfinies, mais pas la récursivité.

```
# let produit_vec_vec l ll =
    List.fold_left
       (function x -> function y -> x @ (produit_vec_liste y l))
       []
       ll;;
val produit_vec_vec : int list list -> int list list -> int list = <fun
>
```

```
# produit_vec_vec [[1;3;5];[1;0;1]] [[2;-6;1]; [7;1;-2]; [-4;2;4]];;
- : int list = [-11; 3; 0; 5; 22; 0]
```

```
# let produit_vec_vec l ll =
    List.fold_right
       (function x -> function y -> (produit_vec_liste x l) @ y)
       ll
       [];;
val produit_vec_vec : int list list -> int list list -> int list = <fun
>
```

```
# produit_vec_vec [[1;3;5];[1;0;1]] [[2;-6;1]; [7;1;-2]; [-4;2;4]];;
- : int list = [-11; 3; 0; 5; 22; 0]
```

```
# let produit_vec_vec l ll =
    let liste = List.map (function x -> (produit_vec_liste x l)) ll
    in  List.fold_left
           (function x -> function y -> x@y)
           []
           liste;;
val produit_vec_vec : int list list -> int list list -> int list = <f
un>
```

```
# produit_vec_vec [[1;3;5];[1;0;1]] [[2;-6;1]; [7;1;-2]; [-4;2;4]];;
- : int list = [-11; 3; 0; 5; 22; 0]
```

Type et exemple d'utilisation :

```
# produit_vec_vec;;
- : int list list -> int list list -> int list = <fun>
```

```
# produit_vec_vec [[1;3;5];[1;0;1]] [[2;-6;1]; [7;1;-2]; [-4;2;4]];;
- : int list = [-11; 3; 0; 5; 22; 0]
```

Des vecteurs d'un type quelconque

Écrivez une fonction calculant le produit scalaire de deux vecteurs d'un même type quelconque.

```
# let rec produit_scalaire_polym x y add neutre_add mul =
   match x,y with
     a::b,c::d -> (add (mul a c) (produit_scalaire_polym b d add neutre_add mul))
   |  [] , []  -> neutre_add
   |     _     -> failwith "Vecteurs de tailles différentes";;
val produit_scalaire_polym :
'a list -> 'b list -> ('c -> 'd -> 'd) -> 'd -> ('a -> 'b -> 'c) -> 'd =
<fun>
```

Dans toute la suite de ce devoir, nous manipulerons des vecteurs d'entiers.

Recherche d'un produit scalaire nul

Nous nous focalisons ici sur les couples de vecteurs de produit scalaire nul.

1. Écrivez une fonction `produit_scalaire_exc` qui prend en entrée deux vecteurs et qui renvoie leur produit scalaire, sauf si celui-ci est nul, auquel cas la fonction devra déclencher une exception `Nul`.

   ```
   # exception Nul;;
   exception Nul
   ```

   ```
   # let produit_scalaire_exc x y =
       let ps = produit_scalaire x y in
       if  ps = 0 then raise Nul else ps;;
   val produit_scalaire_exc : int list -> int list -> int = <fun>
   ```

   Type et exemple d'utilisation :

   ```
   # produit_scalaire_exc;;
   - : int list -> int list -> int = <fun>
   ```

   ```
   # produit_scalaire_exc [1;3;5] [2;-6;1];;
   - : int = -11
   ```

   ```
   # produit_scalaire_exc [1;3;5] [7;1;-2];;
   Exception: Nul.
   ```

2. Écrivez une fonction `produit_vec_liste_exc v l` prenant en entrée un vecteur `v` et une liste `l` et qui renvoie la liste des produits scalaires du vecteur `v` avec les éléments de `l` sauf s'il existe un élément `u` de `l` dont le produit scalaire avec `v` est nul, auquel cas la fonction devra déclencher une exception `Vecteur` transportant le vecteur `u`.

   La fonction `produit_vec_liste_exc` devra utiliser la fonction `produit_scalaire_exc` et non la fonction `produit_scalaire`.

   ```
   # exception Vecteur of int list;;
   exception Vecteur of int list
   ```

   ```
   # let rec produit_vec_liste_exc v = function
        []  -> []
     | t::r -> (try produit_scalaire_exc t v with Nul -> raise (Vecteur t))
               ::(produit_vec_liste_exc v r);;
   val produit_vec_liste_exc : int list -> int list list -> int list = <fun>
   ```

   ```
   # produit_vec_liste_exc [1;3;5] [[2;-6;1]; [-4;2;4]];;
   - : int list = [-11; 22]
   ```

   ```
   # produit_vec_liste_exc [1;3;5] [[2;-6;1]; [7;1;-2]; [-4;2;4]];;
   Exception: Vecteur [7; 1; -2].
   ```

   ```
   # let produit_vec_liste_exc v l =
      List.fold_left
        (function x -> function y ->
   ```

```
        x@[(try produit_scalaire_exc y v with Nul -> raise (Vecteur y))])
      []
      l;;
val produit_vec_liste_exc : int list -> int list list -> int list = <
fun>
```

```
# produit_vec_liste_exc [1;3;5] [[2;-6;1]; [-4;2;4]];;
- : int list = [-11; 22]
```

```
# produit_vec_liste_exc [1;3;5] [[2;-6;1]; [7;1;-2]; [-4;2;4]];;
Exception: Vecteur [7; 1; -2].
```

Type et exemple d'utilisation :

```
# produit_vec_liste_exc;;
- : int list -> int list list -> int list = <fun>
```

```
# produit_vec_liste_exc [1;3;5] [[2;-6;1]; [-4;2;4]];;
- : int list = [-11; 22]
```

```
# produit_vec_liste_exc [1;3;5] [[2;-6;1]; [7;1;-2]; [-4;2;4]];;
Exception: Vecteur [7; 1; -2].
```

3. Écrivez une fonction `produit_vec_vec_exc` prenant en entrée deux listes de vecteurs, `l` et `ll`, et renvoyant la liste des produits scalaires des éléments de `l` avec ceux de `ll` sauf s'il existe un élément `u` de `l` et un élément `v` de `ll` dont le produit scalaire est nul, auquel cas la fonction devra déclencher l'exception `Paire` transportant la paire `(u,v)`.

   La fonction `produit_vec_vec_exc` devra utiliser la fonction `produit_vec_liste_exc` et non la fonction `produit_vec_liste`.

```
# exception Paire of int list * int list;;
exception Paire of int list * int list
```

```
# let rec produit_vec_vec_exc l = function
      []  -> []
   | t::r -> try (produit_vec_liste_exc t l)@(produit_vec_vec_exc l r)
             with Vecteur u -> raise (Paire (u,t));;
val produit_vec_vec_exc : int list list -> int list list -> int list = <f
un>
```

```
# produit_vec_vec_exc [[1;3;5];[1;0;1]] [[2;-6;1]; [7;2;-2]; [-4;2;5]];;
- : int list = [-11; 3; 3; 5; 27; 1]
```

```
# produit_vec_vec_exc [[1;3;5];[1;0;1]] [[2;-6;1]; [7;1;-2]; [-4;2;4]];;
Exception: Paire ([1; 0; 1], [-4; 2; 4]).
```

```
# let produit_vec_vec_exc l ll =
    List.fold_left
       (function x -> function y ->
          x@(try (produit_vec_liste_exc y l)
             with Vecteur u -> raise (Paire (u,y))))
       []
       ll;;
val produit_vec_vec_exc : int list list -> int list list -> int lis
t = <fun>
```

```
# produit_vec_vec_exc [[1;3;5];[1;0;1]] [[2;-6;1]; [7;2;-2]; [-4;2;5]];;
- : int list = [-11; 3; 3; 5; 27; 1]
```

```
# produit_vec_vec_exc [[1;3;5];[1;0;1]] [[2;-6;1]; [7;1;-2]; [-4;2;4]];;
Exception: Paire ([1; 3; 5], [7; 1; -2]).
```

Type et exemple d'utilisation :

```
# produit_vec_vec_exc;;
- : int list list -> int list list -> int list = <fun>
```

```
# produit_vec_vec_exc [[1;3;5];[1;0;1]] [[2;-6;1]; [7;2;-2]; [-4;2;5]];;
```

```
- : int list = [-11; 3; 3; 5; 27; 1]
```

```
# produit_vec_vec_exc [[1;3;5];[1;0;1]] [[2;-6;1]; [7;1;-2]; [-4;2;4]];;
Exception: Paire ([1; 3; 5], [7; 1; -2]).
```

4. Finalement, écrivez une fonction `vecteurs_orthogonaux` prenant en entrée deux listes `l` et `ll` et renvoyant `([],[])`, sauf s'il existe un élément `u` de `l` et un élément `v` de `ll` dont le produit scalaire est nul, auquel cas la fonction renverra la paire `(u,v)`.

```
# let vecteurs_orthogonaux l ll =
    try let res = produit_vec_vec_exc l ll in ([],[])
    with Paire (u,v) -> (u,v);;
val vecteurs_orthogonaux :
int list list -> int list list -> int list * int list = <fun>
```

```
# let vecteurs_orthogonaux l ll =
    try produit_vec_vec_exc l ll ; ([],[]) with Paire (u,v) -> (u,v);;
# let vecteurs_orthogonaux l ll =
try produit_vec_vec_exc l ll ; ([],[]) with Paire (u,v) -> (u,v);;

Warning: this expression should have type unit.
val vecteurs_orthogonaux :
int list list -> int list list -> int list * int list = <fun>
```

Type et exemple d'utilisation :

```
# vecteurs_orthogonaux;;
- : int list list -> int list list -> int list * int list = <fun>
```

```
# vecteurs_orthogonaux [[1;3;5];[1;0;1]] [[2;-6;1];[7;2;-2];[-4;2;5]];;
- : int list * int list = [], []
```

```
# vecteurs_orthogonaux [[1;3;5];[1;0;1]] [[2;-6;1];[7;1;-2];[-4;2;4]];;
- : int list * int list = [1; 3; 5], [7; 1; -2]
```

De nouvelles fonctionnelles prédéfinies

Nous aimerions pouvoir écrire la fonction calculant le produit scalaire de deux vecteurs au moyen d'une des fonctionnelles prédéfinies vues en cours. Ce n'est pas possible, puisque nous avons besoin de décomposer simultanément deux listes. Voici d'autres fonctionnelles prédéfinies en `ocaml` :

1. `# List.iter2;;`
   `- : ('a -> 'b -> unit) -> 'a list -> 'b list -> unit = <fun>`
   `List.iter2 f [a1; ...; an] [b1; ...; bn]` est équivalent à
   `f a1 b1; ...; f an bn.`
2. `# List.map2;;`
   `- : ('a -> 'b -> 'c) -> 'a list -> 'b list -> 'c list = <fun>`
   `List.map2 f [a1; ...; an] [b1; ...; bn]` est équivalent à
   `[f a1 b1; ...; f an bn].`
3. `# List.fold_left2;;`
   `- : ('a -> 'b -> 'c -> 'a) -> 'a -> 'b list -> 'c list -> 'a = <fun>`
   `List.fold_left2 f a [b1; ...; bn] [c1; ...; cn]` est équivalent à
   `f (... (f (f a b1 c1) b2 c2) ...) bn cn.`
4. `# List.fold_right2;;`
   `- : ('a -> 'b -> 'c -> 'c) -> 'a list -> 'b list -> 'c -> 'c = <fun>`
   `List.fold_right2 f [a1; ...; an] [b1; ...; bn] c` est équivalent à
   `f a1 b1 (f a2 b2 (... (f an bn c) ...)).`

Toutes ces fonctionnelles déclenchent l'exception `Invalid_argument` si leur deux listes arguments ne sont pas de même taille.

1. Écrivez une fonction qui calcule le produit scalaire de deux vecteurs en utilisant une ou plusieurs des nouvelles fonctionnelles prédéfinies, mais sans utiliser la récursion.

```
# let produit_scalairef u v =
    let liste = List.map2 (function x -> function y -> x*y) u v
    in List.fold_left (function a -> function b -> a + b) 0 liste;;
val produit_scalairef : int list -> int list -> int = <fun>

# produit_scalairef [1;3;5] [2;-6;1];;
- : int = -11

# let produit_scalairef u v =
    List.fold_left2
       (function x -> function y -> function z -> x + y*z)
       0
       u
       v;;
val produit_scalairef : int list -> int list -> int = <fun>

# produit_scalairef [1;3;5] [2;-6;1];;
- : int = -11

# let produit_scalairef u v =
    List.fold_right2
       (function x -> function y -> function z -> x*y  + z)
       u
       v
       0;;
val produit_scalairef : int list -> int list -> int = <fun>

# produit_scalairef [1;3;5] [2;-6;1];;
- : int = -11
```

2. *Question facultative :* Écrivez une fonctionnelle `fold_left2` qui réalise (fait la même chose que) la fonctionnelle prédéfinie `List.fold_left2`.

```
# let rec fold_left2 f a b c = match (b,c) with
     [] , []  -> a
  | t::r,u::s -> fold_left2 f (f a t u) r s
  |     _     -> raise (Invalid_argument "fold_left2");;
val fold_left2 : ('a -> 'b -> 'c -> 'a) -> 'a -> 'b list -> 'c list -> 'a
=
<fun>

# fold_left2;;
- : ('a -> 'b -> 'c -> 'a) -> 'a -> 'b list -> 'c list -> 'a = <fun>
```