UNIVERSITÉ D'ORLÉANS **DEUG MIAS**
IF 3.01 - Informatique **Année 2001-2002**

Corrigé de l'examen 2000

Exercice 1

```
# let test1 p l = (l <> []) & (p = List.hd l);;
val test1 : 'a -> 'a list -> bool = <fun>
# let test2 p l = (p = List.hd l) & (l <> []);;
val test2 : 'a -> 'a list -> bool = <fun>
# let et_alors b1 b2 = b1 & b2;;
val et_alors : bool -> bool -> bool = <fun>
# let test3 p l = et_alors (l <> []) (p = List.hd l);;
val test3 : 'a -> 'a list -> bool = <fun>
```

Les trois fonctions `test` font, *globalement*, la même chose. Mais dans un ordre différent, et ceci a une importance particulière : le `&` est séquentiel, c'est à dire que la partie gauche est évaluée en premier, et, si elle est fausse, la partie droite n'est même pas évaluée. Ainsi :

```
# test1 1 [];;
- : bool = false
# test2 1 [];;
Uncaught exception: Failure "hd".
# test3 1 [];;
Uncaught exception: Failure "hd".
```

Pour la fonction `test3`, avant de faire l'appel à la fonction `et_alors`, CAML évalue d'abord les deux arguments. C'est à ce moment-là que l'erreur survient, lors de l'évaluation de `p = List.hd l`.

Exercice 2

1. Nous devons écrire une fonction qui extrait le n-ième élément d'une liste :

```
# let rec element_d_indice n liste = match (n,liste) with
          (_,[])           -> failwith "Liste trop courte"
        | (a,_) when (a<1) -> failwith "Indice invalide"
        | (1,e::m)         -> e
        | (a,e::m)         -> element_d_indice (a-1) m;;
val element_d_indice : int -> 'a list -> 'a = <fun>
```

2. Extraire la n-ième ligne du tableau revient à extraire le n-ième élément de la liste des lignes. Donc :

```
# let ligne_numero = element_d_indice;;
val ligne_numero : int -> 'a list -> 'a = <fun>
# let t = [[ 1; 2; 3; 4];
           [ 5; 6; 7; 8];
           [ 9;10;11;12]];;
val t : int list list = [[1; 2; 3; 4]; [5; 6; 7; 8]; [9; 10; 11; 12]]
# ligne_numero 2 t;;
- : int list = [5; 6; 7; 8]
```

3. pour extraire une colonne, il suffit d'extraire le n-ième élément de chaque ligne...

```
# let colonne_numero n liste = List.map (element_d_indice n) liste;;
val colonne_numero : int -> 'a list list -> 'a list = <fun>
# colonne_numero 2 t;;
- : int list = [2; 6; 10]
```

4. Pour extraire l'élément de coordonnées (`ligne`, `colonne`), il suffit d'extraire l'élement d'indice `colonne` de la ligne numéro `ligne`. Donc :

```
# let element_d_indices ligne colonne liste = element_d_indice colonne
                (ligne_numero ligne liste);;
val element_d_indices : int -> int -> 'a list list -> 'a = <fun>
```

5. Pour transposer, la méthode la plus simple est sans doute de procéder par colonnes. Il nous faut donc une fonctin qui supprime la première colonne d'un tableau (pour pouvoir procéder récursivement...)

```
# let supprime_element liste = match liste with
          []   -> failwith "liste deja vide"
        | a::m -> m;;
val supprime_element : 'a list -> 'a list = <fun>
# let supprime_colonne t = List.map supprime_element t;;
val supprime_colonne : 'a list list -> 'a list list = <fun>
# let rec transpose t =
        try (colonne_numero 1 t)::(transpose (supprime_colonne t)) with
                _ -> [];;
# transpose t;;
- : int list list = [[1; 5; 9]; [2; 6; 10]; [3; 7; 11]; [4; 8; 12]]
```

Il existe d'autres méthodes pour faire ce genre d'opérations : par exemple, on pourrait extraire les colonnes sans les supprimer du tableau, mais plutôt en comptant :

```
# let transpose t = let rec transposeaux t i =
        let l = List.length (ligne_numero 1 t)+1 in match i with
          i when i>l -> failwith "erreur"
        | i when i<1 -> failwith "erreur"
        | i when i=l -> []
        | n          -> (colonne_numero n t)::(transposeaux t (i+1)) in
                transposeaux t 1;;
val transpose : 'a list list -> int -> 'a list list = <fun>
# transpose t;;
- : int list list = [[1; 5; 9]; [2; 6; 10]; [3; 7; 11]; [4; 8; 12]]
```

6. On l'a déjà fait ci-dessus.
7. La fonction `diagonale` est assez simple à écrire au premier abord :

```
# let rec diagonale t = match t with
          []   -> []
        | a::m -> (element_d_indice 1 a)::(diagonale (supprime_colonne m));;
val diagonale : 'a list list -> 'a list = <fun>
# diagonale t;;
- : int list = [1; 6; 11]
# diagonale (transpose t);;
Uncaught exception: Failure "Liste trop courte".
```

On s'apreçoit ici qu'on ne teste pas si la matrice est carrée ou pas. Le r'esultat sera évidemment correct si elle l'est. Si elle est rectangulaire, et qu'elle a plus de colonne que de lignes, la fonction va renvoyer la « diagonale » de la matrice carrée incluse à gauche dans cette matrice. On pourrait définir cette diagonale sur toutes les matrices de la façon suivante :

```
# let diagonale_generale t = try diagonale t with
        Failure "Liste trop courte" -> diagonale (transpose t);;
val diagonale_generale : 'a list list -> 'a list = <fun>
# diagonale_generale t;;
- : int list = [1; 6; 11]
# diagonale_generale (transpose t);;
- : int list = [1; 6; 11]
```

8. Voici comment faire la somme de deux matrices ; nous commençons par additionner deux listes...

```
# let rec somme_listes l1 l2 = match (l1,l2) with
          ([],[])     -> []
        | ([],_)      -> failwith "il faut des listes de meme longueur"
        | (_,[])      -> failwith "il faut des listes de meme longueur"
        | (a::m,b::n) -> (a+b)::(somme_listes m n);;
val somme_listes : int list -> int list -> int list = <fun>
# let rec somme_tab t1 t2 = match (t1,t2) with
          ([],[])     -> []
        | ([],_)      -> failwith "il faut des tableaux de meme taille"
        | (_,[])      -> failwith "il faut des tableaux de meme taille"
        | (a::m,b::n) -> (somme_listes a b)::(somme_tab m n);;
val somme_tab : int list list -> int list list -> int list list = <fun>
# somme_tab t t;;
- : int list list = [[2; 4; 6; 8]; [10; 12; 14; 16]; [18; 20; 22; 24]]
# somme_tab [[1;2];[3;4]] [[10;20];[30;40]];;
- : int list list = [[11; 22]; [33; 44]]
```

9. Appliquer une fonction à tous les éléments d'un tableau revient à construire une fonction `map` un peu plus évoluée ; cela revient en fait à emboîter deux `map`s.

```
# let appliquer f t = List.map (List.map f) t;;
val appliquer : ('a -> 'b) -> 'a list list -> 'b list list = <fun>
# appliquer (function x -> x+1) t;;
- : int list list = [[2; 3; 4; 5]; [6; 7; 8; 9]; [10; 11; 12; 13]]
#  appliquer (function x -> 2*x) t;;
- : int list list = [[2; 4; 6; 8]; [10; 12; 14; 16]; [18; 20; 22; 24]]
```