

## TP n°3

### Listes

Les expressions suivantes sont des listes :

- liste vide : []
- liste d'entiers à 3 éléments : [42; 37; 15]
- liste de listes d'entiers : [[0; 1]; [2; 2; 1]; []]

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.   *)
```

La seconde ligne peut être interprétée par : *si l est de la forme : un certain élément a suivi d'une certaine liste l', alors, en appelant effectivement a le premier élément de l et l' la liste qui suit ce premier élément, passer à l'évaluation de l'expression associée.* Autre exemple :

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

*Remarques.* À l'évaluation, les différents cas d'un `match` seront examinés successivement. On peut associer à certains cas le déclenchement d'une exception, par exemple associer à [] l'expression `failwith "erreur, liste vide !"` si l'on souhaite restreindre l'usage de la fonction à des listes non vides. La rencontre d'une telle expression interrompt l'évaluation en cours, et affiche le message fourni au `failwith`.

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

1. `list_sigma` : renvoie la somme des éléments d'une liste d'entiers.

Exemple : `list_sigma [1;2;3] = 6.`

- 2. `mem` : détermine si une liste contient une valeur donnée.

Exemple : `mem 27 [12;27;1] = true, mem 3 [12;27;1] = false.`

- 3. `map` : la fonction telle que `map f [a1;...;an] = [(f a1);...;(f an)]`.  
Exemple : `let succ x = x + 1 in map succ [1;2;3] = [2;3;4]`.
- 4. `liste_min` : fonction calculant le minimum d'une liste d'entiers *non vide*. Si la liste donnée est vide, elle n'a pas de minimum : on renverra dans ce cas un message d'erreur au moyen de `failwith`.  
Exemple : `liste_min [-30;2;549] = -30`.
- 5. `is_sorted` : détermine si une liste est triée.  
Exemple : `is_sorted [1;3;5;6;4] = false`.
- 6. `append` : renvoie la concaténation de deux listes – sans utiliser l'opérateur `@`.  
Exemple : `append [1;2;3] [4;5] = [1;2;3;4;5]`.
- 7. `rev` : renvoie l'inverse d'une liste.  
Exemple : `rev [1;2;3] = [3;2;1]`.
- 8. `flatten` : renvoie la liste obtenue en aplanissant d'un niveau une liste de listes.  
Exemple : `flatten [[2];[];[3;4;5]] = [2;3;4;5]`.
- 9. `rotation_d` : place le dernier élément d'une liste en première position. Si la liste a moins de 2 éléments, elle est renvoyée telle quelle.  
Exemple : `rotation_d ['a';'b';'c';'d'] = ['d';'a';'b';'c']`  
Indications : on peut imbriquer les `match`, donc examiner la forme d'une liste n'importe où dans une fonction ; comment obtenir `['d';'a';'b';'c']` à partir de `['d';'b';'c']` et `'a'` ?
- 10. `moyenne` : la fonction renvoyant la moyenne des éléments d'une liste d'entiers *non vide*, et échouant au moyen de `failwith` si elle est vide. (?) Comment faire pour ne parcourir la liste qu'une seule fois ?  
Exemple : `moyenne [1;2;3] = 2`.

**Exercice 2.** Les fonctions suivantes attendent des listes dont les éléments sont de n'importe quel type permettant l'utilisation des opérateurs de comparaison génériques (`<`, `<=`, `=`, `>=`, `>`). Elles pourront donc manipuler indifféremment des listes d'`int`, de `char`, etc.

1. Ecrire une fonction `insert` qui étant donnée une liste supposée triée pour l'ordre strictement croissant et un élément `x`, renvoie la liste obtenue en insérant `x` à la bonne place. Si `x` est déjà dans la liste, celle-ci sera renvoyée telle quelle.  
Exemples :  
`insert 5 [1;3;8] = [1;3;5;8]`  
`insert 'e' ['a';'c';'g'] = ['a';'c';'e';'g']`
2. En utilisant la fonction `insert`, écrire une fonction `sort` permettant de trier une 'a list quelconque par ordre croissant, en fusionnant les doublons.  
Exemple : `sort [7;8;5;2;8] = [2;5;7;8]`  
Indication : comment obtenir `[2;5;7;8]` à partir de 7 et de `(sort [8;5;2;8])` ?
3. Ecrire une fonction `mem_sorted` se comportant comme la fonction `mem` (cf. l'exercice 1) sur une liste triée, et s'évaluant en un nombre minimum d'étapes.
4. Ecrire les opérations d'union et d'intersection (`union_sorted`, `inter_sorted`) de deux listes triées.  
Exemples :  
`union_sorted [1;3;5] [2;5;8] = [1;2;3;5;8]`  
`inter_sorted [1;3;5] [2;5;8] = [5]`