**TP 2 – Listes, fonctions sur les listes**

---

**Exercice 1. Une simple liste d'entiers**

On veut fabriquer une liste contenant les entiers de 0 à $n$.

**1.** Écrire une fonction `integers` qui fabrique une telle liste en utilisant l'opérateur `@`.

**2.** Écrire une nouvelle version en utilisant `List.rev` (doit-on utiliser une fonction interne ?).

**3.** Écrire une nouvelle version en n'utilisant que l'opérateur `::` (doit-on utiliser une fonction interne ?).

**4.** Mesurer les temps d'exécution pour chacune des versions en utilisant la fonction suivante :

```
let time f x =
  let t = Sys.time() in
    let resultat = f x in
      (Sys.time() -. t) *. 1000., resultat;;
```

Exemple d'inclusion du fichier de :

```
# #use "tp2.ml";;
val time : ('a -> 'b) -> 'a -> float * 'b = <fun>
val integers : int -> int list = <fun>
val integers1 : int -> int list = <fun>
...
```

**Exercice 2. Traiter des listes de nombres**

Écrire les fonctions suivantes et observer leur type. Vous pouvez réutiliser les fonctions que vous aurez définies pour en construire d'autres. **Dans la mesure du possible, vous ne devez parcourir la liste qu'une seule fois.**

**1.** `(size)` qui renvoie la taille d'une liste.

**2.** `(three_or_more l)` qui teste si une liste a au moins 3 éléments.

**3.** `(last l)` qui renvoie le dernier élément d'une liste non vide. Que ce passe-t-il avec une liste vide ?

**4.** `(sum l)` qui renvoie la somme des éléments d'une liste.

**5.** `(find e l)` qui teste si un élément est dans une liste.

**6.** `(nth n l)` qui renvoie le nième élément de $l$.

**7.** `(is_increasing l)` quqi teste si une liste est croissante.

**Ces prochaines questions sont facultatives, faites les si vous avez du temps et pour vous entrainer.**

**8.** `(even_odd l)` qui teste si une liste est telle que ses 1er, 3eme, 5eme, ... éléments sont impairs et les autres pairs.

**9.** `(max_list)l` qui renvoie le maximum d'une liste.

**10.** `(average l)` qui renvoie la moyenne des nombres (réels) de $l$.

**11.** `(size_in_range a b l)` qui teste si la longueur de $l$ est dans l'intervalle $[a, b]$ ou $[b, a]$.

Exemples :

```
# size [], size [3;1;4;5;2];;
- : int * int = (0, 5)
# three_or_more [], three_or_more [1;1;1;1;1];;
- : bool * bool = (false, true)
# last [1], last [3;1;4;5;2];;
- : int * int = (1, 2)
# sum [], sum [3;1;4;5;2];;
- : int * int = (0, 15)
# is_increasing [], is_increasing [3;1;4;5;2], is_increasing [1;3;5;5;7;];;
- : bool * bool * bool = (true, false, true)
# even_odd [], even_odd [1;4;3;6;9;2], even_odd [2;3;3];;
- : bool * bool * bool = (true, true, false)
# find 3 [], find 3 [1;2;3], find 3 [2;4;6];;
```

```
- : bool * bool * bool = (false, true, false)
# nth 3 [1;2;3;4;3;5], nth 3 [2;4;6];;
- : int * int = (3, 6)
# max_list [1;2;3;0;3;0], max_list [2;4;6];;
- : int * int = (3, 6)
# average [5.;8.5;11.5;15.];;
- : float = 10.
# size_in_range 0 0 [], size_in_range 1 3 [0;0], size_in_range 1 3 [0;0;0;0];;
- : bool * bool * bool = (true, true, false)
```

**Exercice 3. Créer des listes de nombres**

Écrire les fonctions suivantes (et observer leur type). Pensez aux "inner" fonctions lorsque c'est nécessaire !

**1.** (`list_copy l`) qui renvoie la copie de la liste.

**2.** (`reverse l`) qui retourne la liste en ordre inverse.

**3.** (`flatten_list l`) qui aplatit une liste de liste.

**4.** (`withouth_duplicates l`) qui supprime les doublons dans une liste triée.

**5.** (`records l`) qui calcule la liste des "records" d'une liste. Un records, dans une liste, est une valeur strictement plus grande que toutes les précédentes.

Exemples :

```
# list_copy [1;2;3];;
- : int list = [1; 2; 3]
# reverse l;;
- : int list = [0; 0; 0; 1; 1; 0; 1; 1; 0; 0]
# flatten_list [[1;2];[];[3;4;5];[6]];;
- : int list = [1; 2; 3; 4; 5; 6]
# without_duplicates [0;0;1;2;3;3;3;3;4;5;5;6;8;8];;
- : int list = [0; 1; 2; 3; 4; 5; 6; 8]
# records [0; 2; 3; 2; 6; 3; 2; 7; 4; 8; 4];;
- : int list = [0; 2; 3; 6; 7; 8]
```

**Exercice 4. Filtrer, transformer des listes**

Écrire les fonctions polymorphe suivantes :

**1.** (`filter f l`) qui retourne la liste dont les éléments respectent le prédicat $f$.

**2.** (`collect f l`) qui retourne le résultat de l'application de la fonction $f$ sur chacun de ses éléments.

**3.** (`reject f l`) qui retourne la liste des éléments sans ceux qui respectent le prédicat $f$.

**4.** (`includes e l`) qui vérifie qu'un élément appartient à la liste $l$.

**5.** (`including l1 l2`) qui verifie que tout les éléments de $l1$ sont compris dans $l2$.

**6.** (`excludes e l`) qui vérifie qu'un élément n'appartient pas à la liste.

**7.** (`excluding l1 l2`) qui vérifie que tout les éléments de $l1$ n'appartienent pas à $l2$.

**8.** (`zip l1 l2`) qui construit la liste des couples de chaque éléments deux à deux des listes $l1$ et $l2$. Que faire si les listes n'ont pas la même longueur ?

**Exercice 5. Combinaison de fonctions**
Nous savons que nous pouvons composer des fonctions en passant directement le résultat d'un appel comme paramètre d'une autre fonction. Il existe aussi en Caml (et en Haskell), un opérateur spécial `|>`, nous allons découvrir son utilisation ici.
**1.** Quel est le type de l'opérateur `|>` ?
**2.** Donner le résultat de l'appel suivant : `[1; 2; 3; 4] |> List.filter (fun x -> x mod 2 = 0);;`. Que c'est-il passé ? (ici, `List.filter` est à le même comportement que la fonction `filter` que vous avez déjà écrite).
**3.** Écrire une fonction retournant les nombres paires en ordre inverse et sans doublons d'une liste de nombre.
**4.** Écrire une fonction qui donne le nombre de tout les nombres impairs d'une liste.
**5.** Écrire une fonction qui transforme en chaine de caractère tout les entiers d'une liste.
**6.** Écrire une fonction qui retourne les carrés de tout les entiers impairs d'une liste.

**Exercice 6. Tri de listes (s'il vous reste du temps)**
On souhaite trier une liste d'entier en utilisant la methode du trie fusion.
**1.** Écrire une fonction `split` qui sépare une liste quelconque en deux listes de tailles à peu près égales.
**2.** Écrire une fonction `merge` qui permet de fusionner deux listes en les ordonnants.
**3.** Écrire une fonction `fusion_sort` qui effectue un tri fusion (le trie fusion est le `merge` du `fusion_sort` des sous listes obtenues par un `split`).