TD : Listes

1 Fonctions sur les listes

Vous pouvez construire une liste en Caml par la donnée de ses éléments de la façon suivante : `[1; 2; 3; 4]` est une liste à 4 éléments. Les fonctions `List.hd` et `List.tl` renvoient respectivement la tête et la queue de la liste, mais en général on fonctionne plutôt par filtrage à l'aide du constructeur « conse » :

```
let rec parcours l=match l with
 | [] -> ()
 | x::q -> parcours q
;;
```

On peut filtrer sur des motifs plus compliqués, comme `x::y::q` par exemple, qui filtre toute liste possédant au moins deux éléments, ou encore `[x;y]` pour une liste à deux éléments.

Exercice 1. *Quelques fonctions sur les listes.* Écrire des fonctions prenant en entrée une liste et effectuant les actions suivantes. On parcourera la liste une seule fois si possible (ça l'est !). On pourra introduire une fonction auxiliaire interne.

1. `trois_ou_plus l` qui teste si la liste `l` possède au moins trois éléments (inutile de parcourir toute la liste !).

```
#trois_ou_plus [1;2] ;;
- : bool = false
#trois_ou_plus [1;2;3;4] ;;
- : bool = true
```

2. `avant_dernier l` donnant l'avant dernier élément de la liste (si il existe, sinon renvoie une erreur).

```
#avant_dernier [1] ;;
Uncaught exception: Failure "pas assez d'elements"
#avant_dernier [1;2;3;4] ;;
- : int = 3
```

3. `maxi l` qui donne le maximum d'une liste (supposée non vide). La fonction `max` peut-être utilisée pour obtenir le maximum de deux éléments.

```
#maxi [1;2;3;4;3;2] ;;
- : int = 4
```

4. `nb_occ l x` qui donne le nombre d'occurences d'un certain élément `x` dans une liste.

```
#nb_occ [1;2;3;2;3;1;2;3;2] 3 ;;
- : int = 3
```

5. `nb_maxi l` qui donne le nombre d'occurences du maximum d'une liste (on calculera le maximum en même temps !)

```
#nb_maxi [1;2;3;4;3;2] ;;
- : int = 1
#nb_maxi [1;2;3;2;3;1;2;3;2] ;;
- : int = 3
```

6. `nb_records l` qui calcule le nombre de *records* d'une liste, c'est à dire le nombre d'éléments strictement plus grands que les précédents (le premier compte)

```
#nb_records [1;2;3;2;3;4;2;3;5] ;;
- : int = 5
```

Exercice 2. La fonction `List.map` de signature `('a -> 'b) -> 'a list -> 'b list` de Caml prend en entrée une fonction f de signature `'a -> 'b` et une liste ℓ de type `'a list` et crée la liste des $f(a)$ pour a dans ℓ. La réécrire sous le nom `map`. Faire ensuite une variante utilisant une fonction auxiliaire récursive (on pourra utiliser `List.rev`, qui renvoie une copie miroir de la liste passée en paramètre).

```
#let f x=2*x in map f [0;2;5;4] ;;
- : int list = [0; 4; 10; 8]
```

Exercice 3. Écrire une fonction `combine l1 l2` combinant deux listes $[a_0; \dots; a_{n-1}]$ et $[b_0; \dots; b_{n-1}]$ pour former la liste des couples (a_i, b_i) (dans le même ordre) et renvoyant une erreur si les listes n'ont pas même taille. La fonction a pour signature `'a list -> 'b list -> ('a * 'b) list`.

```
#combine [0;1;2] [3;4;5] ;;
- : (int * int) list = [0, 3; 1, 4; 2, 5]
#combine [0;1;2] [3;4] ;;
Uncaught exception: Failure "pas le meme nombre d'elements"
```

Exercice 4. La fonction `List.iter` en Caml, de signature `('a -> unit) -> 'a list -> unit`, prend en entrée une fonction f de signature `'a -> unit` (appelée un traitement) et une liste $\ell = [a_0; \dots; a_{n-1}]$ de type `'a list`, et effectue les traitements $f(a_0), \dots, f(a_{n-1})$ (dans cet ordre). Écrire une fonction `iter_rev` effectuant les traitements $f(a_{n-1}), \dots, f(a_0)$ dans cet ordre. Par exemple :

```
#iter_rev print_int [4;5;6] ;;
654- : unit = ()
```

Exercice 5. Écrire une fonction `prefixes` de type `'a list -> 'a list list` prenant en entrée une liste et renvoyant la liste de ses préfixes comme ci-dessous. On pourra utiliser les fonctions précédentes. (L'ordre dans la liste des préfixes n'a pas d'importance). Estimer la complexité de votre fonction.

```
#prefixes [0;1;2;5;6;7;8] ;;
- : int list list =
 [[]; [0]; [0; 1]; [0; 1; 2]; [0; 1; 2; 5]; [0; 1; 2; 5; 6]; [0; 1; 2; 5; 6; 7]; [0; 1; 2; 5; 6; 7; 8]]
```

Exercice 6. Même exercice avec la liste des suffixes. On essaiera d'avoir une complexité linéaire en la taille de la liste passée en paramètre.

2 Des algorithmes de tri

Exercice 7. *Tri par sélection sur les listes.*

1. Écrire une fonction `min_reste q` prenant en entrée une liste `q` supposée non vide et renvoyant un couple formé du minimum de la liste et de la liste des autres éléments dans un ordre quelconque.

   ```
   #min_reste [5; 3; 2; 4; 9] ;;
   - : int * int list = 2, [9; 4; 3; 5]
   ```

2. En déduire une fonction `tri_selection q` triant la liste passée en entrée dans l'ordre croissant.

   ```
   #tri_selection [5; 3; 2; 4; 9] ;;
   - : int list = [2; 3; 4; 5; 9]
   ```

3. Quelle est la complexité du tri ?

Exercice 8. *Tri par insertion sur les listes.*

1. Écrire une fonction `insertion q x` prenant en entrée une liste `q` supposée triée dans l'ordre croissant, et un élément `x`, et renvoyant une liste constituée des éléments de `q` et de `x`, dans l'ordre croissant.

   ```
   #insertion [2; 4; 6; 12; 18] 7  ;;
   - : int list = [2; 4; 6; 7; 12; 18]
   ```

2. En déduire une fonction `tri_insertion q` triant la liste passée en entrée dans l'ordre croissant.

   ```
   #tri_insertion [5; 3; 2; 4; 9] ;;
   - : int list = [2; 3; 4; 5; 9]
   ```

3. Quelle est la complexité du tri ?