Manipulations de listes

I. Échauffement

Question 1 Écrire une fonction

```
avant_dernier: 'a list -> 'a
```

qui fait ce que vous pensez.

Question 2 Écrire une fonction `appli: ('a -> 'b) -> 'a list -> 'b list` qui prends en entrée une fonction f et une liste l, et retourne la liste obtenue en appliquant f à chaque élément de l.

Par exemple, on veut :

```
appli String.length ["les"; "ordinateurs"; "sont"; "vos"; "amis"] ;;
- : int list = [3; 11; 4; 3; 4]
```

Remarque Cette fonction existe déjà dans la bibliothèque standard d'*OCaml*, il s'agit de `List.map`. Bien sûr, le but de l'exercice précédent n'est pas d'utiliser cette fonction mais de la réécrire entièrement.

II. Quelques fonctions supplémentaires

Question 3 Écrire une fonction `longueur: 'a list -> int` qui calcule la longueur d'une liste.

Question 4 Écrire une fonction `somme: int list -> int` qui renvoie la somme des entiers contenus dans la liste passée en argument.

Question 5 Écrire une fonction `suffixes: 'a list -> 'a list list` qui renvoie la liste de tous les suffixes de la liste.

Par exemple,

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

On parle ensuite de `map`, `iter` et `fold_(left|right)`.

Question 6 Écrire des fonctions `pour_tous` et `il_existe`, toutes les deux de type `('a -> bool) -> 'a list -> bool` telles qu'étant donné un prédicat P

et une liste l, les fonctions renvoient `true` ou `false` suivant que tous les éléments de l (resp. au moins un élément de l) vérifient P.

```
# pour_tout (fun x -> x mod 3 = 0) [7; 8; 9; 10] ;;
- : bool = false
# il_existe (fun x -> x mod 3 = 0) [7; 8; 9; 10] ;;
- : bool = true
```

Toutes les fonctions de cette section ont, normalement, la même structure :

```
let rec ma_fonction l =
    match l with
    | [] -> ... (* valeur pour la liste vide *)
    | a :: l' ->
        let u = ma_fonction l' in
        ... (* combine a et u *)
;;
```

En remplaçant les points de suspension par respectivement `u0` et `f`, on a une fonction générique qui ne dépend que de `u0` et `f`, que l'on peut écrire :

```
let rec ma_fonction l =
    match l with
    | [] -> u0
    | a :: l' -> f a (ma_fonction l')
;;
```

C'est exactement ce que fait la fonction générique

```
List.fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
```

la fonction précédente correspondant à `List.fold_right f l u0`. L'expression `List.fold_right` f `[`a_0`;`a_1`;`$\cdots$`;`a_{n-1}`]` u_0 correspond à :

$$f(a_0, f(a_1, f(\cdots, f(a_{n-1}, u_0))))$$

Ainsi, la fonction `somme` précédente peut se réécrire[1] :

```
let somme l = List.fold_right (fun x y -> x + y) l 0 ;;
```

Question 7 Réécrire les fonctions `longueur`, `suffixes`, `pour_tous` et `il_existe` en utilisant `fold_right`.

Remarque C'est une chose de connaître et savoir utiliser `List.fold_right`, c'est encore mieux de savoir reconnaître et maîtriser les fonctions de cette forme.

[1] Elle peut même s'écrire `let somme l = List.fold_right (+) l 0 ;;`, en entourant l'opérateur infixe de parenthèses.

III. Combinaison à droite, combinaison à gauche

Avec les fonctions précédentes, on part de la droite avec la valeur d'initialisation u_0 et on applique la fonction f vers la gauche (en associant à droite, d'où le nom). Un problème est que la fonction n'est pas récursive terminale, ce qui peut-être gênant si on l'applique à de très longues listes.

Il en existe une autre version, nommée `List.fold_left`, qui va de la gauche vers la droite. L'avantage étant que l'on a alors une récursion terminale. Son type est le suivant (attention, il y a de petits différences avec l'autre) :

```
List.fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
```

Question 8 Définissez les fonctions suivantes :

```
let myst1 u v = List.fold_left (fun a b -> b :: a) u v ;;
let myst2 u v = List.fold_right (fun a b -> a :: b) u v ;;
```

1. Essayez ces fonctions avec des exemples.
2. En déduire une fonction `miroir : 'a list -> 'a list` qui renvoit l'image miroir (de droite à gauche) de la liste pasée en argument. Pouvez-vous en faire une version récursive terminale ?
3. Écrire une fonction de concaténation

   ```
   concat : 'a list -> 'a list -> 'a list
   ```

   Pouvez-vous en faire une version récursive terminale ?

IV. Listes représentant des ensembles

On s'intéresse aux opérations ensemblistes sur des ensembles représentés par des listes. Par exemple, la liste `[2; 4; 1]` représente l'ensemble $\{1,2,4\}$.

Question 9 Écrire une fonction `appartient : 'a -> 'a list -> bool` qui indique si un élément appartient ou non à la liste.

Question 10 Écrire des fonctions `intersection` et `union` de type

```
'a list -> 'a list -> 'a list
```

et retourne la liste représentant l'intersection (resp. l'union) des ensembles représentés par les listes passées en argument.

Question 11 Quel est la complexité de ces fonctions ?

On suppose maintenant que les listes représentant les ensembles sont triées par ordre croissant.

Question 12 Réécrire les trois fonctions précédentes avec cette nouvelle information. On essayera d'avoir une complexité linéaire.

V. Quelques exercices supplémentaires

Question 13 Étant donné une liste de booléens, déterminer où couper la liste en deux de façon à maximiser le nombre de `true` dans la partie gauche plus le nombre de `false` dans la partie droite.

Question 14 Écrire une fonction

```
suffixes : 'a list -> 'a list list
```

qui, étant donné une liste, retourne la liste de ses suffixes.

On veut par exemple

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

Question 15 *(Plus dur)* Écrire une fonction

```
sous_listes : 'a list -> 'a list list
```

qui, étant donné une liste, retourne la liste de ses sous-listes.

On veut par exemple

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