TP : révisions sur les listes

1 Pour s'échauffer...

Toutes les solutions seront écrites dans le style fonctionnel.

Question 1

Écrire une fonction `double : int list -> int list` qui double la valeur de tous les éléments d'une liste.

Question 2

Écrire une fonction `intervalle : int -> int -> int list` telle que `intervalle a b` construise la liste ordonnée de tous les entiers entre a et b inclus.

Question 3

Écrire une fonction `miroir : 'a list -> 'a list` inversant l'ordre des éléments d'une liste. Sa complexité devra être linéaire $O(n)$.

2 Tris élémentaires

Ces tris sont à connaître. Les tris insertion et sélection sont quadratiques $O(n^2)$. Le tri fusion est de complexité $O(n \log n)$. Toutes les solutions seront écrites dans le style fonctionnel.

2.1 Tri par insertion

Le **tri par insertion** est le plus simple à programmer en OCaml : c'est donc un indispensable !

Question 4

Écrire une fonction

```
insere : 'a -> 'a list -> 'a list
```

insérant un élément en bonne position dans une liste déjà triée par ordre croissant.

Question 5

En déduire une fonction

`tri_insertion : 'a list -> 'a list`

triant une liste d'éléments par ordre croissant.

2.2 Tri par sélection

Question 6

Écrire une fonction

`extraire_min : 'a list -> 'a * 'a list`

prenant en entrée une liste non vide et retournant un couple `(min, reste)` où `min` est le plus petit élément de la liste et `reste` est la liste privée de cet élément.

Question 7

En déduire une fonction `tri_selection : 'a list -> 'a list` triant une liste par ordre croissant.

2.3 Tri fusion

Question 8

Écrire une fonction `partition : 'a list -> 'a list * 'a list` prenant en entrée une liste d'éléments et la séparant en deux listes de longueurs égales ou quasi-égales. L'ordre des éléments dans le résultat n'a pas d'importance.

Exemple :

```
partition [1; 2; 3; 4; 5] = ([1; 3; 5], [2; 4])
```

Question 9

Écrire une fonction `fusion : 'a list -> 'a list -> 'a list` fusionnant deux listes **déjà triées** par ordre croissant en une liste triée par ordre croissant.

Question 10

En déduire une fonction `tri_fusion : 'a list -> 'a list` réalisant le tri fusion.

3 Fonctions d'ordre supérieur

Une possibilité très puissante de la programmation fonctionnelle est l'utilisation de fonctions prenant en entrée des fonctions. Voici un exemple simple :

```
let rec map f l = match l with
    | [] -> []
    | t::q -> (f t)::(map f q)
;;
```

Cette fonction existe déjà dans la bibliothèque OCaml sous le nom `List.map`

Question 11

Écrire à l'aide de `map` des fonctions réalisant les tâches suivantes :
1. `plusun` qui ajoute la valeur 1 à toutes les valeurs d'une liste d'entiers
2. `double` qui double toutes les valeurs d'une liste

3.1 Utilisation de prédicats

Question 12

Écrire une fonction d'ordre supérieur

`filter : ('a -> bool) -> 'a list -> 'a list`

prenant en entrée une fonction `predicat` et une liste `l`. Cette fonction filtre les éléments de la liste `l` et ne garde que ceux qui prennent la valeur `true` sur la fonction `predicat`.

Question 13

Écrire à l'aide de `filter` des fonctions réalisant les tâches suivantes :
1. `impair` qui ne conserve que les valeurs impaires d'une liste.
2. `positifs` qui supprime toutes les valeurs strictement négatives d'une liste.

Question 14

Dans le même esprit que `filter` nous avons également les fonctions :

```
exists : ('a -> bool) -> 'a list -> bool
```

```
forall : ('a -> bool) -> 'a list -> bool
```

testant si un élément (resp. tous les éléments) d'une liste vérifient un prédicat donné. Proposer une implémentation de ces deux fonctions en OCaml. On pourra remarquer que l'une peut s'obtenir avec l'autre...

Question 15

Écrire à l'aide des fonctions `exists` et `forall`, des fonctions réalisant les tâches suivantes :
1. `negatif` qui teste si une liste contient une valeur strictement négative.
2. `mem : 'a -> 'a list -> bool` qui teste si un élément apparaît dans une liste.
3. `moitie : int list -> int list` prenant une liste d'entiers et si elle ne contient que des entiers pairs construit la liste des entiers divisés par deux. Sinon la liste est inchangée.

3.2 Réductions

Voyons maintenant une fonction d'ordre supérieur un peu plus compliquée :

```
let rec fold_left f accu l =
    | [] -> accu
    | t::q -> fold_left f (f accu t) q
;;
```

Question 16

Comprendre le fonctionnement de `fold_left` en détaillant l'exécution de

```
List.fold_left (fun x y -> x + y) 0 [1; 2; 3; 4]
```

Question 17

En utilisant la fonction `fold_left`, programmer une fonction calculant le produit de toutes les valeurs d'une liste.

Question 18

En utilisant la fonction `fold_left`, programmer la fonction factorielle `maxliste` retournant la plus grande valeur présente dans une liste non vide d'entiers positifs.

Question 19

En utilisant la fonction `fold_left`, programmer la fonction factorielle `fact`

Question 20

Écrire une fonction `flat : 'a list list -> 'a list` qui applatit une liste de listes. Par exemple :

```
flat [[1; 2; 3]; [4; 5]; []; [6]; [7; 8; 9]] = [1; 2; 3; ...; 9]
```

Question 21

Écrire la fonction `fold_right` qui réalise la même chose que `fold_left` mais en traitant les éléments de la liste de droite à gauche.

4 Exercices plus difficiles

Question 22

Écrire une fonction

```
produit : 'a list -> 'b list -> ('a * 'b) list
```

qui calcule le produit cartésien de deux listes l_1 et l_2. La complexité devra être $O(|l_1| \cdot |l_2|)$.

Exemple :

```
produit ['a'; 'b'] [1; 2]= [('a', 1); ('a', 2); ('b', 1); ('b', 2)]
```

Question 23

Écrire une fonction

```
combinaison : 'a list -> int -> 'a list list
```

telle que `combinaison l k` retourne la liste de toutes les combinaisons de k élements de la liste l.

Question 24

On considère ici qu'on peut comparer deux listes à l'aide de l'ordre lexicographique. Écrire une fonction

```
tri_listes : 'a list list -> 'a list list
```

réalisant le tri par ordre croissant d'une liste de listes. On programmera la comparaison lexicographique.