université de BORDEAUX	**Année Universitaire 2017/2018, Session 1 d'automne** Licence Informatique & MI L2, UE 4TINA01U, Programmation fonctionnelle. Jeudi 11 janvier 2018. 11h30–13h Durée: 1h30. **Documents interdits.**	**Collège Sciences et Technologies**

Expliquez brièvement en français le code de chaque fonction écrite.
Seule la syntaxe vue en cours intégré et en TD est autorisée.
Le barème est indicatif.

Vous pouvez utiliser les fonctions du module `List`. On rappelle en particulier la fonction `map`, qui prend en argument une fonction f et une liste `[`a_1`; ...; `a_n`]` et renvoie la liste `[`fa_1`; ...; `fa_n`]`.

Exercice 1 — 11 points. On considère le type suivant, servant à représenter des arbres dont seules les feuilles sont étiquetées par des entiers strictement positifs.

```
type mobile = Leaf of int | Bin of mobile * mobile
```

Le nom de « mobile » est suggéré par les œuvres de même nom du sculpteur Alexander Calder. En voici quelques exemples :

```
let m1 = Bin(Bin(Leaf 1, Leaf 1), Leaf 2)
let m2 = Bin(Bin(m1,m1),Leaf 8)
let m3 = Bin(Bin(m1,m1),Leaf 42)
```

Le *poids* d'un mobile m est défini comme la somme des étiquettes de toutes ses feuilles. Par exemple, le poids de `m1` vaut 4, celui de `m2` vaut 16 et celui de `m3` vaut 50.

1) Écrire la fonction `poids` de type `mobile -> int`.

On dit qu'un mobile est *équilibré* si :

- soit c'est une feuille étiquetée par un entier strictement positif,
- soit c'est un nœud binaire dont les fils sont deux mobiles équilibrés et de même poids.

Par exemple, `m1` et `m2` sont équilibrés, mais pas `m3`, car les poids de `Bin(m1,m1)` et de `Leaf 42` sont différents.

2) Écrire une fonction `equilibre_et_poids`, de type `mobile -> bool * int` de telle sorte que `equilibre_et_poids` m renvoie :

   - `(true,` p`)` si m est un mobile équilibré de poids p,
   - `(false,` p`)` sinon, où p est le poids de m.

   On demande que le nombre d'appels récursifs de la fonction `equilibre_et_poids` soit au maximum le nombre de nœuds de son argument, en justifiant ce fait.

3) Utilisez `equilibre_et_poids` pour écrire une fonction `equilibre` de type `mobile -> bool` qui teste si un mobile est équilibré.

4) Écrire une fonction `combine_one` de type `mobile -> mobile list -> mobile list` telle que `combine_one` m `[`m_1`; ...; `m_k`]` renvoie la liste de mobiles `[Bin(`m`,`m_1`); ...; Bin(`m`,`m_k`)]`.

5) Écrire une fonction `combine` de type `mobile list -> mobile list -> mobile list` telle que `combine` ℓ ℓ' renvoie une liste contenant tous les mobiles de la forme `Bin(`m`,`m'`)` où m est un mobile de ℓ et m' est un mobile de ℓ'.

6) On veut écrire une fonction `all_mobiles` de type `int -> mobile list` qui renvoie la liste de tous les mobiles équilibrés de poids donné. Par exemple

```
# all_mobiles 6;;
- : mobile list = [Leaf 6; Bin (Leaf 3,Leaf 3)]

# all_mobiles 4;;
- : mobile list = [Leaf 4; Bin (Leaf 2,Leaf 2); Bin (Leaf 2,Bin (Leaf 1,Leaf 1));
 Bin (Bin (Leaf 1,Leaf 1),Leaf 2); Bin (Bin (Leaf 1,Leaf 1),Bin (Leaf 1,Leaf 1))]
```

Pour écrire `all_mobiles`, on note que si p est un entier strictement positif :

- Si p est impair, le seul mobile équilibré de poids p est `Leaf` p.
- Si p est pair, un mobile équilibré de poids p est
  - soit `Leaf` p,
  - soit `Bin(`m_1`,`m_2`)`, où m_1 et m_2 sont deux mobiles équilibrés de poids $p/2$.

En utilisant la fonction `combine` de la question 5), écrire la fonction `all_mobiles`.

Exercice 2 — 9 points. On s'intéresse à des listes représentant une suite de déplacements :

```
type move = Up | Down
type trek = move list
```

À toute liste ℓ de type `trek`, on associe une liste d'entiers `altitudes` ℓ de même longueur que ℓ. Informellement, `altitudes` ℓ représente la liste d'altitudes atteintes en faisant les mouvements de ℓ de gauche à droite : un `Up` correspond à incrémenter l'altitude et un `Down` correspond à la décrémenter. Par exemple, la liste associée à `[Up;Up;Up;Down]` est `[1;2;3;2]`. Celle associée à `[Up;Up;Down;Up;Down;Down]` est `[1;2;1;2;1;0]`. Celle associée à `[Down;Up;Up]` est `[-1;0;1]`.

1) Écrire la fonction `altitudes` de type `trek -> int list`.

On dit qu'une liste ℓ de type `trek` est *réussie* si elle est vide, ou si elle a les 2 propriétés suivantes :

(a) le dernier élément de `altitudes` ℓ est `0`, et

(b) `altitudes` ℓ ne contient aucune valeur strictement négative.

Par exemple, les listes `[Up; Down; Up; Down]` et `[Up; Up; Down; Down]` sont réussies, contrairement à `[Up; Down; Up]` et `[Down; Up]`.

2) Écrire une fonction `success` de type `trek -> bool` qui teste si une liste de type `trek` est réussie.

3) Écrire une fonction `reduce` de type `trek -> trek` telle que `reduce` ℓ renvoie la liste obtenue à partie de ℓ en effaçant tous les mouvements `Up; Down` consécutifs. Par exemple, `reduce [Up; Down; Up; Down]` est la liste vide, et `reduce [Up; Up; Down; Down]` vaut `[Up; Down]`.

4) Montrer que si ℓ `= reduce` ℓ alors `success` ℓ renvoie `true` si ℓ `= []` et `false` sinon.

5) Donner un argument montrant que la fonction suivante calcule le même résultat que `success`.

```
let rec success' l =
  match l with
  | [] -> true
  | _ -> let l' = reduce l in
    (List.length l' < List.length l) && success' l'
```

On rappelle que la fonction `List.length` renvoie la longueur d'une liste.