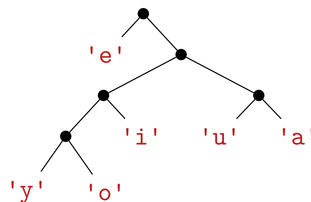
Programmation fonctionnelle

PF : TD 7

Dans cette feuille de TD, pour chaque fonction que vous implémenterez, vous en indiquerez le type le plus général.

Exercice 1 : Codage de Huffman [6pt]

Le codage de Huffman peut être utilise pour compresser une suite de valeurs (par exemple un texte, qui est une suite de caracteres) en remplacant chaque valeur par son code, qui est une sequence de valeurs booleennes construite à partir d'un arbre binaire portant les valeurs à coder dans ses feuilles. Par exemple



est un tel arbre contenant les voyelles de l'alphabet.

Toute feuille d'un arbre binaire peut être identifiée par le chemin qui va de la racine à cette feuille, et plus précisément ici par la liste des choix de la direction à prendre à chaque nœud interne : `G` pour continuer dans le sous-arbre gauche ou `D` pour continuer dans le sous-arbre droit. Dans un arbre de Huffman, où toutes les valeurs sont différentes, le code d'une valeur est la liste de directions menant à la feuille qui porte cette valeur. Pour l'arbre ci-dessus, on a donc pour la lettre `'e'` le code `[G]`et pour la lettre `'i'` le code `[D,G,D]`. Une liste de directions menant à une feuille sera appelée *code valide* pour l'arbre considéré. Dans notre exemple ni `[G,D]` ni `[D,D]` ne sont des codes valides.

Dans ce contexte, on définit les types de données suivant :

```haskell
data ArbreH a = Feuille a | Noeud (ArbreH a) (ArbreH a)
  deriving Show
data Direction = G | D
  deriving Show
type Codes = [Direction]
```

L'arbre de notre exemple peut donc être défini comme :

```haskell
voyellesH :: ArbreH Char
voyellesH = Noeud (Feuille 'e')
                  (Noeud(Noeud (Noeud (Feuille 'y') (Feuille 'o'))
                   (Feuille 'i'))
                   (Noeud (Feuille 'u') (Feuille 'a')))
```

Q 1.1 Écrire une fonction qui, étant donnés un arbre et une liste de directions, retourne le couple formé de la valeur codée et du reste de la liste de directions qui n'a pas été utilisée. Comme cette fonction peut ne pas renvoyer de valeur, utiliser `Maybe` afin d'éviter qu'elle ne renvoie une erreur.

```
ghci> decodeH voyellesH [D,G,D]
Just ('i',[])
ghci> decodeH voyellesH [D,G,D,G,D,D]
Just ('i',[G,D,D])
ghci> decodeH voyellesH [D,D]
Nothing
```

Q 1.2 Ecrire une fonction `decodeToutH` qui, etant donnes un arbre et une liste de directions, retourne la liste des elements de type `a` qu'elle code. Comme précédemment, cette fonction est partielle ; utiliser `Maybe` afin d'éviter qu'elle ne renvoie une erreur.

```
ghci> decodeToutH voyellesH [D,D,D,G,D,G,D]
Just "aei"
ghci> decodeToutH voyellesH []
Just ""
ghci> decodeToutH voyellesH [G,D,D]
Nothing
```

La construction d'un arbre de Huffman s'appuie sur les fréquences d'apparition respectives des valeurs dans les séquences à coder. Les valeurs les plus fréquentes doivent être le plus près possible de la racine si on veut un code efficace. C'est le cas dans l'arbre `voyellesH` de l'exemple basé sur la fréquences des différentes voyelles apparaissant dans un texte français.

On définit le type suivant associant un pourcentage d'apparition à une valeur de type `a` qu'on appelle le *poids* de la valeur :

```
data Freq a = F a Float
```

Ce qui permet de définir par exemple (la fréquence est en pourcentage) :

```
voyellesF :: [Freq Char]
voyellesF =
  [F 'a' 17.5,F 'e' 42.2,F 'i' 14.3,F 'o' 10.4,F 'u' 14.7,F 'y' 0.8]
```

Afin de construire un arbre de Huffman à partir d'une telle liste, il va nous falloir comparer les éléments de type `Freq a`. Cette comparaison est uniquement basée sur les poids.

Q 1.3 Ecrire deux fonctions :
- `sortF` qui permet de trier une liste d'élément de type `Freq a` en les classant par fréquence **croissante**,
- `insertF` qui permet d'insérer dans une liste triée par fréquence croissante un élément de type `Freq a` pour obtenir une liste triée.

Pour cela vous pouvez utiliser dans votre définition les fonctions suivantes :
- `sortBy :: (a -> a -> Ordering) -> [a] -> [a]` ; `sortBy f l` renvoie une permutation de `l` triée dans l'ordre croissant donné par `f`.
- `insertBy :: (a -> a -> Ordering) -> a -> [a] -> [a]` ; lorsque `l` est une liste triée dans l'ordre croissant donné par `f`, `insertBy f a l` renvoie la liste triée construite en ajoutant `a` dans `l`.
- `compare :: Ord a => a -> a -> Ordering` ; `compare a1 a2` indique si `a1` est plus petit, égal ou plus grand que `a2`.

Pour rappel, le type `Ordering` est défini par :

```
data  Ordering    =  LT  -- plus petit
                   | EQ  -- égal
                   | GT  -- plus grand
```

Exemple :

```
ghci> sortF voyellesF
[F 'y' 0.8,F 'o' 10.4,F 'i' 14.3,F 'u' 14.7,F 'a' 17.5,F 'e' 42.2]
```

La première étape de construction d'un arbre *optimal* de Huffman consiste à transformer une liste de poids de valeurs de type `a` en liste de poids de valeurs de type `ArbreH a`.

Q 1.4 [**0,5pt**] Ecrire une fonction `initH` qui prend une liste d'éléments de type `Freq a` et retourne la liste d'éléments de type `Freq (ArbreH a)` ou chaque element `F v f` de la liste en entrée est remplace par `F (Feuille v) f`.

Vous pouvez utiliser la fonction `map :: (a -> b) -> [a] -> [b]` dans votre définition.

```
ghci> initH []
[]
ghci> initH [F 'a' 17.5]
[F (Feuille 'a') 17.5]
ghci> initH [F 'a' 17.5, F 'e' 42.2, F 'i' 14.3]
[F (Feuille 'a') 17.5,F (Feuille 'e') 42.2,F (Feuille 'i') 14.3]
```

L'algorithme de construction de l'arbre de Huffman à partir d'une liste de type `[Freq (ArbreH a)]` triée (suivant les fréquences) est le suivant :

- si la liste ne contient qu'un seul élément alors cet élément est l'arbre de Huffman résultat
- sinon on choisit dans la liste deux arbres t_1 et t_2 de poids p_1 et p_2 minimaux ; on considere alors la liste de depart privée des éléments correspondants à t_1 et t_2 et augmentée de l'élement correspondant à un arbre t de fils gauche t_1, de fils droit t_2 et de poids $p_1 + p_2$. Cette nouvelle liste contient un élément de moins que la liste d'origine. Elle doit être triée.

Q 1.5 Ecrire une fonction `etapeH` qui prend en argument deux éléments `t1 t2` de type `Freq (ArbreH a)` et une liste de `Freq (ArbreH a)` triée dans l'ordre croissant des fréquences et renvoie cette liste augmentée de l'élément obtenu à partir des deux `t1` et `t2` comme expliqué dans la construction plus haut. Pour rappel, la liste renvoyée doit être triée.

```
ghci> etapeH (F (Feuille 'a') 17.5) (F (Feuille 'i') 14.3) [F (Feuille 'e') 42.2]
[F (Noeud (Feuille 'a') (Feuille 'i')) 31.8,F (Feuille 'e') 42.2]
```

Q 1.6 Ecrire une fonction `arbreH` qui construit un arbre de Huffman à partir d'une liste d'éléments de type `Freq a` (non nécessairement triée) donnée en entrée. Vous utiliserez `Maybe` pour éviter que la fonction ne déclenche une erreur dans le cas où la liste est vide.

```
ghci> arbreH []
Nothing
ghci> arbreH voyellesF
Just (Noeud (Feuille 'e') (Noeud (Noeud (Noeud (Feuille 'y')
       (Feuille 'o')) (Feuille 'i')) (Noeud (Feuille 'u') (Feuille 'a'))))
```

Q 1.7 On souhaite maintenant écrire une fonction `codeH` qui prend un arbre de Huffman en argument contenant des valeurs de type `a`, et qui retourne la fonction de codage associee (c'est-a-dire la fonction qui prend un argument de type `a` et renvoie la liste de directions qui mene a cet element dans l'arbre donne si la valeur est présente et rien si la valeur est absente de l'arbre). Vous utiliserez `Maybe` afin que la fonction `codeH` ne déclenche pas une erreur.

Écrire la fonction `codeH`.

Q 1.8 Pour terminer on souhaite écrire une fonction `codeListeH` qui prend un arbre de Huffman contenant des valeurs de type `a`, et qui retourne une fonction qui permet de coder une liste de `a`. Vous utiliserez `Maybe` afin que la fonction `codeListeH` ne déclenche pas une erreur.