Fiche de TD no. 4

Exercice 1. Avant aborder cet exercice, veuillez reviser la syntaxe des définitions de type et assurez vous d'avoir compris la signification du mot clef *constructeur*. (Par exemple, faite la liste de tous le constructeurs vus en cours).

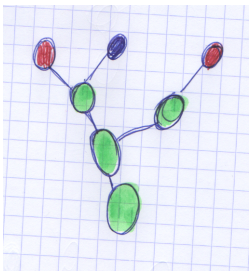
Considérez les définitions de type suivantes :

```
type Tree = (Int,Tree,Tree)
data Arbre = Int | (Arbre,Arbre)
data Couleur = Rgb (Float,Float,Float)
data ListeNonVide a = Fin a | Cons a (ListeNonVide a)
data Cactus = Feuille Couleur | Noeud (ListeNonVide Cactus)
```

1. Quelles sont les définitions correctes ? Justifiez votre réponse. Si une définition de type n'est pas correcte, proposez une correction.
2. Listez tous les noms de types qui appairassent dans les définitions ci-dessus.
3. Listez tous les constructeurs que vous voyez et écrivez leurs types.
4. Quels sont les types polymorphes définis ?
5. Quels sont les types récursifs définis ?

Exercice 2.

Considérez le cactus de la figure à droite. Dans cette figure (originairement en couleurs, mais imprimée en noir et blanc) la première et la troisième feuilles sont rouges, la deuxième feuille est bleu.
Vues les définitions des types `Couleur` et `Cactus` (dans l'exercice précédent), représentez ce cactus en Haskell comme un valeur (c'est-à-dire, expression non ultérieurement évaluable) de type `Cactus`.



Exercice 3.

1. En utilisant le filtrage (avec les *motifs fondamentaux* déterminés par le *constructeurs* du type `ListeNonVide a`) et la récursion, définissez des fonctions

   ```
   longueur :: ListeNonVide a -> Int
   maximum :: Ord a => ListeNonVide a -> a
   ```

   qui calculent ce que est leur nom.
2. En utilisant encore le filtrage et la récursion, définissez des fonctions

   ```
   taille :: Cactus -> Int
   hauteur :: Cactus -> Int
   ```

   qui calculent ce que est leur nom. Est il possible d'utiliser seulement les motifs fondamentaux du type `Cactus` ?
3. Définissez, maintenant, une version de la fonction `map`, adaptée au type `ListeNonVide a` :

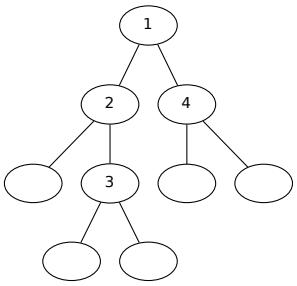
   ```
   autreMap :: (a -> b) -> ListeNonVide a -> ListeNonVide b
   ```

4. En utilisant la fonction `autreMap` définissez une deuxième version de la fonction `hauteur`, qui utilise seulement les motifs fondamentaux du type `Cactus`.

Exercice 4. Considérez la variante du type récursif des arbres binaires définie par :

```
data ArbreBin = Feuille | Noeud ArbreBin Int ArbreBin
```

1. Écrivez une expression Haskell, de type `ArbreBin`, qui représente l'arbre



2. Dessinez l'arbre binaire représenté par `a1` dans la définition Haskell

```
a1 = Noeud (Noeud Feuille 4 Feuile)
     3
     (Noeud (Noeud Feuille 5 Feuile) 6 Feuile)
```

3. Proposez trois autres expressions Haskell de type `ArbreBin` et dessinez les arbres qu'elles dénotent.
4. Définissez les fonctions suivantes :

`taille : ArbreBin -> Int`	compte le nombre des noeuds dans un arbre binaire,
`maxA : ArbreBin -> Int`	l'étiquette maximum d'un arbre binaire,
`minA : ArbreBin -> Int`	l'étiquette minimum d'un arbre binaire.

   Comment modifier les fonctions `maxA` et `minA` si on veut que `maxA Feuille` $= -\infty$ et `minA Feuille` $= +\infty$?
5. Un arbre binaire est un *arbre de recherche* si l'étiquette de chaque noeud est plus grande des toutes les étiquettes dans le sous-arbre à gauche, et plus petite des toutes les étiquettes dans le sous-arbre à droite. Définissez un prédicat Haskell `aRecherche :: ArbreBin -> Bool` qui répond vrai à un arbre donné ssi il s'agit d'un arbre de recherche.
6. Un arbre binaire est équilibré si, pour chaque noeud, la taille du sous-arbre à gauche et celle du sous-arbre à droite différent au plus de 1. Définissez un prédicat Haskell `aEquilibre :: ArbreBin -> Bool` qui répond vrai à un arbre donné ssi il s'agit d'un arbre équilibré.
7. Est ce que les fonctions `aRecherche` et `aEquilibre` sont performantes ? Comment modifier la définition du type `arbreBin` pour améliorer la performance de ces fonctions ?
8. La plus part des algorithmes sur les arbres binaires peuvent se spécifier comme suit :
   — dans le cas d'une feuille, on retourne un valeur donné `v` ;
   — pour le cas d'un noeud, on calcule la valeur de cet arbre à laide d'une fonction `f` de trois arguments : un entier est les deux valeurs calculés de façon récursive avec le sous-arbre à gauche et le sous-arbre à droite.
   Montrez comment
   — les fonctions `taille`, `maxA` et `minA`,
   — le calcul de la somme de toutes les étiquettes d'un arbre,
   — le calcul de l'hauteur d'un arbre (longueur de la plus longue branche)
   peuvent se faire/définir selon ce schéma de calcul : qui sont `v` et `f` ?
9. Définissez une fonction

```
foldArbre :: a -> (a -> Int -> a -> a) -> ArbreBin -> a
```

   qui capture le schéma de récursion qu'on vient de voir.