Fiche de TD no. 3

Input-output

Exercice 1. Considérez la fonction suivante :

```
afficheEtIncrementeComplexe (x,y) =
    putStr (show x ++" + i"++show y)
    >>=
    \() -> putStr "\n"
    >>=
    \() -> return (x+1,y+1)
```

1. Après avoir rappelé le type de `putStr`, de `return`, et de l'opérateur binaire `>>=`, typez cette fonction.
2. Expliquez ce qui est accompli par cette fonction.
3. Combien de patterns (motifs) pouvez vous reconnaître dans ce code ? Justifiez votre réponse.
4. Que fait, dans le code ci-dessus, la fonction `show` ? Quel est son type ?
5. Expliquez pourquoi le code similare

   ```
   afficheEtIncrementeComplexe (x,y) =
       putStr (show x ++" + i"++show y)
       >>=
       putStr "\n"
       >>=
       return (x+1,y+1)
   ```

   produit un erreur de type.

Exercice 2. Le programme suivant utilise la syntaxe `do` pour enchaîner les actions :

```
main = do
  _ <- putStr "Quel est votre nom ? \n"
  nom <- getLine
  _ <- putStr "Quel est votre age ? \n"
  age <- getLine
  putStrLn (show (nom,age))
```

Réécrivez ce programme en utilsant l'opérateur `>>=` à la place de `do`.

Définitions de types

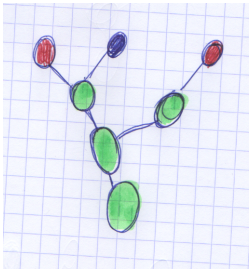
Exercice 3. Considérez les définitions de types suivantes :

```
data Arbre = Int | (Arbre,Arbre)
type Tree = (Int,Tree,Tree)
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. Soulignez tous les constructeurs que vous voyez, et ecrivez leurs types.
3. Quels sont les types polymorphes définis ?

4. Quelles sont les types recursifs définis ?

Exercice 4. Considérez le cactus de la figure suivante :



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

Exercice 5.

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`.