Fiche de TP no. 4

Caveats : *le temps passe vite ! Si, après vingt minutes, vous êtes encore au premier exercice, cela est un bon indice que vous ne vous donnez pas les moyens d'apprendre.*
Objectifs : *Nous avons aujourd'hui ces objectifs :*

1. *Comprendre comment on définit des types et comment on utilise les types récursifs (notamment, la récursion avec les types récursifs).*
2. *Aborder le projet.*

Distribuez vos énergies et temps de façon égalitaire entre ces objectifs.

Exercice 1. Le type des arbres binaires dont les feuilles sont étiquetés par des objets de type `a` peut se définir via le code suivant :

```
data ABin a = Feuille a | Noeud (ABin a) (ABin a) deriving Show
```

Ces arbres binaire servent à coder des mots parenthésés, comme par exemple

`((((12)3)4)5)`, `(((12)3)(45))`, `((12)(3(45)))`, `(1(2(3(45))))` (**)

1. Ecrivez une fonction

   ```
   toABin :: [a] -> ABin a
   ```

   qui prend une liste, et parenthese cette liste tout à gauche. Par exemple, `toABin [1,2,3,4,5]` donnera `((((12)3)4)5)`.
2. Écrivez une fonction

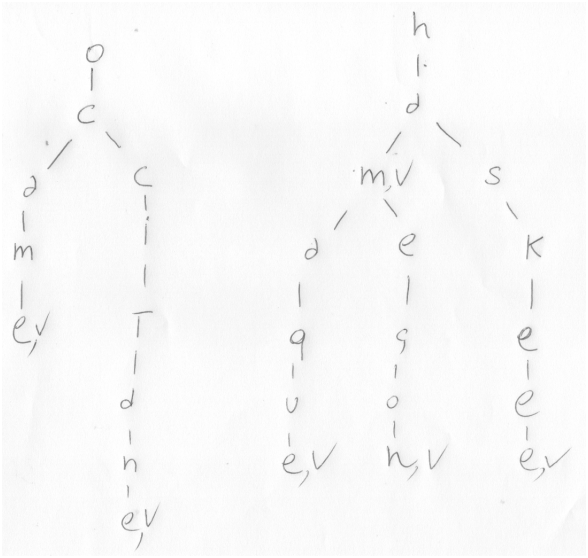
   ```
   showABin :: Show a => ABin a -> String
   ```

   qui prend un arbre binaire et écrit la chaîne de caractères parenthesée.
   Par exemple, `showABin (Noeud (Noeud (Feuille 1) (Feuille 2)) (Feuille 3))` donnera `((12)3)`.
3. Écrivez une fonction

   ```
   moveRight :: ABin a -> Maybe (ABin a)
   ```

   qui re-paranthèse un mot (parenthésé) de la gauche vers la droite. Par exemple, les mots dans (**) ont été obtenues par itération successive de cette fonction.
   On retourne `Nothing` quand un mot est déjà parenthésé tout à la droite.

Exercice 2. Un *arbre de préfixes* est une structures de données arborescente adaptée à maintenir un ensemble de chaînes de caractères (non vides) en minimisant au même temps l'espace de stockage et le temps de recherche. La minimisation a lieu en partageant l'espace des préfixes communs à deux ou plusieurs mots. Chaque noeud d'un arbre de l'arbre de préfixes est étiqueté par un caractère et par un Booléen. Si on parcourt un arbre en lisant un après l'autre les caractères d'un mot et on arrive ainsi à un noeud étiqueté par vrai, alors le mot appartient à l'arbre de préfixes (sinon, non). Par exemple, l'arbre de préfixes suivant :



contient exactement les mots : *ocaml,occitane,ham,hamac,hamecon,haskell.* La définition de type suivante est adaptée à coder cette strcture de données :

```
data APref = APref [(Char,Bool,APref)] deriving (Show,Eq)
```

1. Après avoir défini `empty` comme l'arbre de préfixes vide, définissez la fonction

   ```
   ajouter :: String -> APref -> APref
   ```

   qui ajoute un mot à l'arbre de préfixes. Donc, si on définit

   ```
   mots = ["ocaml","occitane","ham","hamaque","hamecon","haskell"]
   dict = foldr ajouter empty mots
   ```

   alors `dict` sera la représentation Haskell du arbre de préfixes ci-dessus. (N'hésitez pas à découper le problème en morceaux et à définir toute fonction intermédiaire).
2. Écrivez une fonction `prettyPrint` pour pour pouvoir afficher de façon sympathique un tel arbre de préfixes avec sa structure. Par exemple, depuis `ghci`, on aura

   ```
   *Main> putStrLn $ prettyPrint dict
   ham
   ---ac
   ---econ
   --skell
   ocaml
   --citane
   *Main>
   ```

   Remarques. Que veut dire le symbole `$`? Quel est le type de la fonction `prettyPrint`? Vous pouvez résoudre aisément cette deuxième partie de l'exercice en utilisant les fonctions `lines` et `intercalate` depuis le module `Data.List`.

Appendice : le projet

Pendant les derniers 20 minutes, prenez vision du texte provisoire du projet :

http://pageperso.lif.univ-mrs.fr/~luigi.santocanale/teaching/PF/projet2017.pdf

Si nécessaire, posez des questions sur le projet à votre encadrant de TP.