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. *Comprendre comment on instancie un type à une classe.*
3. *Prendre vision du projet.*

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

Exercice 1. Considérez la définition suivante du type des expressions arithmétiques pouvant contenir des variables :

```
type Variable = String
data Expr = Var Variable
          | Val Int
          | Add Expr Expr
          | Mul Expr Expr
```

1. Un « store » (ou contexte) est une fonction (possiblement partielle) associant des valeurs aux variables. On peut représenter les stores par des listes associatives :

   ```
   type Store = [(String,Int)]
   ```

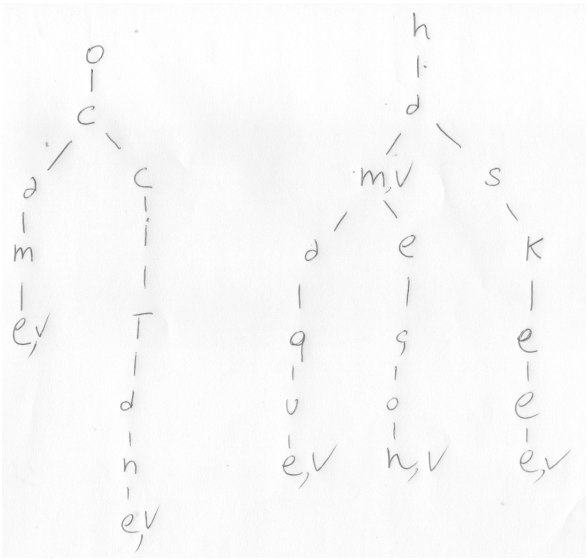
   Écrivez la définition de la fonction

   ```
   evaluate :: Store -> Expr -> Int
   ```

   qui évalue une expressions par rapport à la valeur des variables définie dans un store.
2. Modifiez la définition du type `Expr` de façon que l'on puisse utiliser l'exponentiation et la division entière.
3. Ajoutez le type `Int` (ou un type isomorphe) à la classe `Fractional`. Remplacez, dans la définition du type `Expr`, le type `Int` par une variable de type `a` (et donc paramétriez le type avec la variable `a`). Modifiez la définition de la fonction `evaluate` (et ce qui est nécessaire dans votre code) de façon à ce que l'on ait

   ```
   type Store a = [(String,a)]
   evaluate :: Fractional a => Store a -> Expr a -> a
   ```

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,hamaque,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
   ---aque
   ---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/projet2016.pdf

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