Paradigmes et Langages de Programmation

Haute École d'Ingénierie et de Gestion du Canton de Vaud

4. Haskell / Types utilisateurs

2022

Exercice 1

Dans les transparents, la fonction suivante a été donnée à titre d'exemple :

```
initiales prenom nom
  | null prenom = "X." ++ initiale nom
  | null nom    = initiale prenom ++ " X."
  | otherwise   = initiale prenom ++ ' ':initiale nom
  where initiale (c:_) = toUpper c:['.']
```

Récrire cette fonction sur le type *Personne* telle que définie dans le cours :

```
data Personne = Personne {
      prenom :: String,
      nom    :: String,
      age    :: Int,
      taille :: Float
} deriving (Show)
```

Compléments :

- Effectuez une recherche dans une liste de personnes à l'aide de *filter*.
- Vieillissez les personnes d'une liste d'une année avec *map*.
- Calculez la moyenne d'âge des personnes dans une liste avec rien d'autres qu'un pliage

Définissez finalement un module où vous exposerez toutes ces fonctionnalités.

Exercice 2

Soit la déclaration de type suivante :

```
data Expr
    = Const Int
    | Sum Expr Expr
    | Prod Expr Expr
```

Ce type décrit des expressions arithmétiques sommaires. Il ne dérive pas de *Show*, donc une expression de type *Sum (Const 2) (Const 3)* ne peut pas être affichée. Pour rendre cet affichage possible, il faut explicitement dériver le type *Expression* de *Show* de la manière suivante :

```
instance Show Expr where
  show ... = ...
```

Définissez cette fonction *show* de manière à reproduire l'affichage d'une calculatrice. Définissez ensuite une fonction *eval* qui permet de calculer le résultat d'une expression :

```
eval :: Expr -> Int
```

Finalement, définissez un module où vous exposerez toutes ces fonctionnalités.

Exercice 3

Définissez un type *Liste* que vous exposerez via un module avec les fonctionnalités suivantes :

- Définissez les opérations d'insertion et suppression sur une liste triée.
- Construisez une liste à l'aide d'un pliage et une liste standard.
- Dérivez le type *Liste* de *Show* en implémentant une fonction *show* plus lisible que la version automatique de Haskell.
- Implémentez une fonction de recherche qui retourne la liste (de type *Liste*) des éléments satisfaisant un prédicat.

Indication : Il ne faut pas dériver votre type Liste de Show dans la déclaration data sinon il y a un conflit. Ne cherchez pas nécessairement à obtenir une représentation identique à celle des listes standard de Haskell ([1,2,3]).

Exercice 4

Définissez un type *Arbre* que vous exposerez via un module et créez une fonction d'insertion, ce qui vous permettra de créer aisément un arbre à partir d'une liste d'éléments.

Ajoutez ensuite une opération de test d'appartenance :

```
contient :: ... => Arbre t -> t -> Bool
```

Quelle est la contrainte de classe sur le type des éléments ?

Créez une fonction qui aplatit un arbre et rend la liste des éléments :

```
aplatir :: Arbre t -> [t]
```

Dérivez votre type *Arbre* de la classe de type *Functor*. Testez le fonctionnement de votre dérivation en appliquant une fonction aux éléments de l'arbre.

Vérifiez que les propriétés attendues soient respectées :

```
fmap id = id
fmap f . fmap g = fmap f . g
```

Exercice 5

Définissez une fonction qui permet de déterminer si un arbre est symétrique, c.-à-d. si ses deux sous-arbres gauche et droit sont en miroir (on ne se préoccupe pas de la valeur des noeuds, uniquement de la structure).

Exercice 6

Le Run Length Encoding, ou RLE, est un algorithme de compression où toute suite de n symboles s identiques est représentée par le couple (n,s). Cet algorithme est utile dans certaines applications et peut être efficace par exemple sur des images en noir/blanc.

Écrivez une fonction qui applique l'algorithme de RLE :

```
rle :: ... => [t] -> [(Int,t)]
```

Bon travail !