Principes de Programmation
Partiel 2017

24 mai 2017

Le partiel dure 2 heures. Aucun document n'est autorisé.

Les exercices sont indépendants. Il est autorisé de sauter des questions, d'en changer l'ordre, ou même de répondre à plusieurs questions à la fois, mais il faut chaque fois **indiquer à quelle(s) question(s) vous répondez**.

Certaines questions peuvent être difficile, mais :

- aucune réponse ne nécessite plus d'une demie page, [1]
- répondre partiellement à une question difficile peut être suffisant,
- les questions difficiles offrent des points bonus si bien répondues (15 points bonus disponibles !).

Chaques exercice est sur 5 points

Exercice 1 (Cours).

1. Quel est le type de `max` donné en annexe ?
2. Quel est le type de `map` donné en annexe ?
3. Qu'est-ce qu'une *type-class* ? Donner un exemple.
4. Qu'est ce qu'une *monade*, moralement ?
5. Donnez trois exemples de monades (définition formelle non requise).

Exercice 2 (Programmation).
On définit les arbres de préfixes ainsi :

```
data PTree a = Node (Char -> PTree a)  |  Leaf a  |  Vide
```

Donnez le type et implémentez (en haskell) les fonctions suivantes ; essayez d'utiliser les outils vus en cours lorsque c'est pertinent : [2]

1. `rechClef :: PTree a -> String -> a`
   qui cherche un élément dans l'arbre grâce à sa clef ; la clef d'un élément de l'arbre est le `String` dont les lettres définissent le chemin d'accès,

1. Sauf peut-être dans les exercices 3 et 4 si vous écrivez gros.
2. Le type donné est indicatif, vous êtes invités à le changer si vous avez une idée plus pertinente.

2. `parsser :: PTree a -> String -> [a]`
   qui utilise un arbre préfixes pour lire une phrase, rendant une liste des éléments (de types `a`) ; en effet, puisque les clefs sont des préfixes, une phrase ne peut être découpée qu'en une seule suite de clefs (plus éventuellement quelques lettres à la fin).
3. `recherche :: (Eq a) => PTree a -> a -> Bool`
   qui cherche si un élément est dans l'arbre,
4. `toList :: PTree a -> [a]`
   qui donne la liste des éléments apparaissant dans l'arbre,
5. Implementez une type class de votre choix.

Exercice 3 (Programmation).
On considère l'implémentation suivante des AVL :

```
type Hauteur = Int
data AVL a = F | N Hauteur a (AVL a) (AVL a)

h F = 0
h (N n _ _ _) = n

left  (N _ _ f _) = f
right (N _ _ _ g) = g

n x f g = N (1 + max (h f) (h g))) f g

ajt F x = N 1 x F F
ajt (N k x f g) y
   | x==y = (N k y f g)
   | (x>y) && h (ajt f y) > (h g +1)  =
       (if (h (left (ajt f y)) >= h(right (ajt f y)) )
         then rotg else rotgd)
       (N (h (ajt f y)) x (ajt f y) g)
   | (x<y) && h (ajt g y) > (h f +1)  =
       (if (h (left (ajt g y)) <= h(right (ajt g y)) )
         then rotd else rotdg)
       (N (h (ajt g y)) x f (ajt g y))


rotg (N h x (N i y f gg) g) =  (N' y f (N' x gg g))
rotgd (N h x (N y f (N z ff gg)) g) = (N' z (N' y f ff) (N' x gg g))
rotd (N x g (N y ff g)) =  (N' y (N' x f ff) g)
rotdg (N x f (N y (N z ff gg) g)) = (N' z (N' x f ff) (N' y gg g))
```

Ce programme est en fait plein de défauts.

1. Listez ceux que vous voyez.

2. Au choix :
   - expliquer comment corriger chacun de ces défauts
   - écrivez une version corrigée du code.

Exercice 4 (Egalités). (Questions 1,2,3,4 non nécéssaires pour 5,6,7,8)
On définit la classe des monades mathématiques comme un foncteur spécialisé :

```
Class Functor m => MathMonad m where:
   unit :: a -> m a
   mult :: m (m a) -> m a
```

En plus des règles du foncteur, une monade mathématique doit respecter les règles suivantes (pour tout `l::m a` et `v::m(m(m a))`) :

```
mult (unit l)     = l                 :: m a
mult (map unit l) = l                 :: m a
mult (mult v)     = mult (map mult v) :: m a
```

1. Rappelez les équations que doit satisfaire un foncteur (et donc une monade mathématique).
2. Rappelez la définition de la classe `Monad`.
3. Montrez que toute monade est une monade mathématique.
4. Inversement, montrez que toute monade mathématique est une monade.

Comme dans l'exercice 2, on définit les arbres de préfixes ainsi :

```
data PTree a = Node (Char -> PTree a)  |  Leaf a
```

5. Donnez une implémentation de `Functor PTree`.
6. Montrez que les arbres de préfixe forment bien un foncteur.
7. Donnez une implémentation de `MathMonad PTree`.
8. Montrez que les arbres de préfixe forment bien une monade.

Annexes

On rappel qu'une barre vertical "|" après un patern-maching est une garde, c'est à dire une condition pour effectuer le code qui suit.

Le code de `!!` est (moralement) le suivant :

```
!! :: [a] -> Int -> a
[]    !! _ = undefined
(x:_) !! 0 = x
(_:l) !! n = l !! (n-1)
```

Le code de `map` est (moralement) le suivant :

```
!! :: (a -> b) -> [a] -> [b]
map _ []    = []
map f (x:l) = (f x) : (map f l)
```

Le code de `[n..]` est (moralement) le suivant :

```
[n..] = n : [(n+1)..]
```

Le code de `max` est (moralement) le suivant :

```
max x y | x >= y    = x
        | otherwise = y
```

Le code de `&&` est (moralement) le suivant :

```
x && False = False
x && y     = y
```

Le code de `foldl` est (moralement) le suivant :

```
foldl _ a []  = a
map f a (x:l) = map f (f a x) l
```

Le code de la classe `Bonded` est (moralement) le suivant : [3]

```
class Bounded a  where
    minBound :: a
    maxBound :: a
```

Le code de la classe `Enum` est (moralement) le suivant : [4]

```
class Enum a  where
    succ       :: a -> a
    prec       :: a -> a
    enumFrom   :: a -> [a]
    enumTo     :: a -> [a]
    enumFromTo :: a -> a -> [a]
```

veut dire que les valeurs de `a` peuvent s'énumérer. En particulier, `[x..]` et `[x..y]` sont des macro (du sucre syntaxique) pour `(enumFrom x)` et `(enumFromTo x y)`.

On rappel aussi qu'un opérateur comme `!!` peut être mis en notation préfixe en mettant des parenthèse. Par exemple `((!!) x y) = (x !! y)`.

3. Bounded ne peut être utilisé pour répondre à la question 1.3
4. Enum ne peut être utilisé pour répondre à la question 1.3