Interrogation Écrite 2

Consignes et objectifs

Le but de cette interrogation est d'évaluer vos capacités à :
- faire un parcours récursif d'une structure arborescente pour faire un calcul ;
- définir et utiliser une fonction partielle avec `Maybe` ;
- utiliser des pliages ;
- typer correctement (en utilisant les types les plus génériques et les classes de type).

Pour chaque implémentation, vous devez préciser **le type le plus général**. En cas de doute, écrivez ce que vous avez essayé.

Exercice 1 : Structure de documents [~ 4 pts]

Quand on y réfléchit bien, un document (même ce sujet !) peut être vu comme une structure de données récursive. Un document a toujours un titre et un contenu. Le contenu peut être simplement du texte ou bien une liste de documents.

Q. 1 Décrivez la structure de données qui peut représenter ce type de document. Vous pouvez utiliser le type `data Either a b = Left a | Right b` qui permet de gérer le cas où un élément peut être soit du type `a` soit du type `b`.

Indication

N'hésitez pas à utiliser la syntaxe d'enregistrement (*record syntax*) pour simplifier votre modèle de données.

Éléments de correction

Code

```haskell
data Document = Document {
  title :: String, -- le titre, toujours un string
  contents :: Either [Document] String
}
```

Q. 2 Écrivez une fonction `titres` qui prend un `Document` en paramètre et qui renvoie la liste des titres contenus à l'intérieur.

Vous pouvez utiliser la fonction `concatMap :: (a -> [b]) -> [a] -> [b]`

✔ Éléments de correction

</> Code

```haskell
titres :: Document -> [String]
titres (Document t (Left ld)) = t : concatMap (titres) ld
titres d                      = [titre d]
```

Exercice 2 : Élément minimum d'une liste (version sûre) [~ 5 pts]

Q. 1 En utilisant un filtrage de motif, écrire une fonction qui calcule le minimum des éléments d'une liste, sans générer d'erreur quand la liste est vide.

✔ Éléments de correction

</> Code

```haskell
maxListe :: Ord a => [a] -> Maybe a
maxListe []     = Nothing
maxListe (x:xs) = case maxListe xs of
                    Nothing  -> Just x
                    (Just y) -> Just (max x y)
```

</> Code

```haskell
minListe :: Ord a => [a] -> Maybe a
minListe []     = Nothing
minListe (x:xs) = case minListe xs of
                    Nothing  -> Just x
                    (Just y) -> Just (min x y)
```

Q. 2 Donner une version de cette fonction utilisant un pliage.

✔ Éléments de correction

</> Code

```haskell
minListe' :: Ord a => [a] -> Maybe a
minListe' = foldr (minM) Nothing
    where
      minM el acc = case acc of
        Nothing -> Just el
        Just x  -> Just (min x el)
```

</> Code

```haskell
maxListe' :: Ord a => [a] -> Maybe a
maxListe' = foldr (maxM) Nothing
    where
      maxM el acc = case acc of
        Nothing -> Just el
        Just x  -> Just (max x el)
```

Exercice 3 : Dernier élément, sous condition [~ 3 pts]

Écrire une fonction qui renvoie le dernier élément d'une liste qui satisfait une propriété. Cette fonction doit être sûre, si aucune valeur n'est trouvée, elle renverra `Nothing`.

✔ Éléments de correction

</> Code

```haskell
findLast :: (a -> Bool) -> [a] -> Maybe a
findLast pred = foldl (\acc e -> if p e then Just e else acc) Nothing
```

Exercice 4 : Approximation de la fonction cosinus [~ 10 pts]

La fonction **cosinus** est égale à la série entière $\sum_{k=0}^{\infty}(-1)^k \frac{x^{2k}}{(2k)!}$

On veut écrire une fonction `cosApprox` qui prend une valeur `x` et une valeur `n` et qui approxime cos(x) au rang n.

Pour que le calcul soit efficace, on veut éviter d'effectuer intégralement les calculs de x^{2k} et $(2k)!$ à chaque étape. Pour cela, on utilise le fait que :

— $x^{2(k+1)} = x^{2k} \times x^2$

— $(2(k+1))! = (2k)! \times (2k+1) \times (2k+2)$

Pour rappel, $(-1)^k \frac{x^{2k}}{(2k)!}$ vaut 1 lorsque $k = 0$. Vous pouvez vous servir de cette information pour éviter un cas particulier.

Indication

À quelle liste appliquer un pliage pour réaliser ce calcul ?
La fonction puissance en Haskell peut ici s'écrire `(^^)`

Éléments de correction

Code

```
cosApprox :: Fractional a => a -> a -> a
cosApprox x n = somme
  where (_, _, _, somme) = foldl
        (\(pow, fac, sq, s) e -> let fac' = (2*e+1)*(2*e+2)*fac
                                     pow' = pow * sq
                                     in (pow', fac', sq, s + (-1)^^e *
                                      ↪  pow'/fac'))
                           (1, 1, x*x, 1)
                           [1..n]
```