Interrogation Écrite 4

Consignes et objectifs

Le but de cette interrogation est d'évaluer vos capacités à :

- utiliser **`Maybe`** en tirant avantage de sa monade et de son implémentation d'alternative pour ne pas avoir de filtrage de motif ;
- utiliser **`IO`** et ses fonctions types ;
- 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. Une **liste de fonctions utiles** est disponible à la fin du sujet. Vous pouvez à tout moment utiliser toute fonction de cette liste.

Exercice 1 : Foldify

Après un semestre plein de fun à faire de la programmation fonctionnelle, vous *savez* que vous allez vous ennuyer en vacances. Pour gonfler un peu votre CV, vous vous dites que monter votre start-up disruptive, c'est à la fois dans l'air du temps et utile.

Vous vous souvenez avoir parsé des playlists de musique récemment, et donc, à juste titre, vous vous dites que vous pouvez en tirer quelque chose.

Informations

On rappelle que le type de données concerné est :

Code

```haskell
data Duration = Duration {
    minutes :: Int,
    seconds :: Int
}
data Song = Song {
    artist   :: String,
    title    :: String,
    year     :: Int,
    duration :: Duration
}
type Playlist = [Song]
```

Q. 1 - Another File Bites the Dust

Écrire une fonction `loadPlaylist :: FilePath -> IO Playlist` qui lit le fichier au chemin donné en argument et parse le contenu pour en faire une liste de lecture. Laisser un message sur l'entrée standard pour indiquer à l'utilisateur le statut du chargement de la playlist.

✔ Éléments de correction

</> Code

```haskell
loadPlaylist :: FilePath -> IO Playlist
loadPlaylist path = do
    file <- readFile path
    let playlist = runParser parsePlaylist file
    case playlist of
        Just (p, "")  -> putStrLn "Playlist parsée avec succès" >> return
        ↪  p
        _             -> putStrLn "Erreur pendant le parsing" >> return
        ↪  []
```

Q. 2 - Play Me Maybe ♫

Écrire une fonction `play :: Song -> IO ()` qui prend une chanson en argument et qui affiche un message sur la sortie standard en lien avec cette chanson ("en train de jouer : ..." ou juste le titre et l'artiste, etc[1]).

✔ Éléments de correction

</> Code

```haskell
play :: Song -> IO ()
play song = do
    putStrLn $ "En train de jouer : " ++ show song
```

Q. 3 - Take Me To String ♫

Écrire une fonction `showPlaylist :: Playlist -> String` qui prend une playlist en argument et renvoie une chaîne de caractères représentant la liste des chansons numérotée. Encore une fois, c'est vous qui décidez quelles informations afficher (mais ça doit être utile). Utiliser `zipWith` et `unlines` ici peut être vraiment utile.

✎ Exemple

Par exemple, avec la liste du dernier sujet, on pourrait avoir :

M↓ Code

```
ghci> showPlaylist playlist
1. Fix You (Coldplay)
2. Time Is Running Out (Muse)
3. Radioactive (Imagine Dragons)
4. Mr. Brightside (The Killers)
5. Viva La Vida (Coldplay)
6. Starlight (Muse)
7. Believer (Imagine Dragons)
8. Ain't It Fun (Paramore)
9. Do I Wanna Know? (Arctic Monkeys)
10. Il suffira d'un fold - Lambda Remix (DJ TypeClass feat S. Salvat)
11. Counting Stars (OneRepublic)
```

1. C'est votre appli, c'est vous qui décidez !

✔ Éléments de correction

</> Code

```haskell
showPlaylist :: Playlist -> String
showPlaylist playlist = unlines (zipWith (\ i s -> show i ++ ". " ++ show
↪  s) [1..] playlist)
```

Q. 4 - Loop Me Baby One More Time 🎵

Compléter la fonction `loop :: Playlist -> IO ()` suivante et expliquer sans trop paraphraser ce qu'elle fait :

</> Code

```haskell
loop :: Playlist -> IO ()
loop playlist = do
    putStrLn "Choisir chanson : "
    -- récupérer l'entrée utilisateur (le numéro dans la liste)
    s <-
    if null s
        then return ()
        else do
            -- récupérer la chanson dans la liste
            let song =
            play song
            loop playlist
```

✔ Éléments de correction

</> Code

```haskell
loop :: Playlist -> IO ()
loop playlist = do
    putStr "Choisir chanson : "
    hFlush stdout
    s <- getLine
    if null s
        then return ()
        else do
            let song = playlist !! (read s - 1)
            play song
            loop playlist
```

Q. 5 - Take My Main Away 🎵

Écrire la fonction `main :: IO ()`, qui va :

1. charger la playlist contenue dans le fichier `"musiques.md"` ;
2. afficher les chansons de la playlist sous la forme d'une liste numérotée ;
3. et utiliser `loop`.

✔ Éléments de correction

</> Code

```haskell
main :: IO ()
main = do
    a <- loadPlaylist "musiques.md"
    putStrLn (showPlaylist a)
    loop a
```

Exercice 2 : All I Want For Christmas is (Maybe) You 🎁

Comme vous avez encore un peu de temps à perdre avant de passer de bonnes vacances, vous allez écrire un programme pour suivre votre budget cadeaux.

Vous avez établi une liste de souhaits, stockée comme ceci :

</> Code

```
liste = [
    ("Papa", ["Livre"]),
    ("Maman", ["Jeu", "Figurine"]),
    ("Prof", ["Copies"])
    ]
```

Q. 1 Quel est le type de cette liste ?

✔ Éléments de correction

</> Code

```
liste :: [(String, [String])]
```

Les cadeaux possibles sont stockés dans une liste associative. La clé est le nom du cadeau, la valeur son numéro de produit en magasin. En utilisant la fonction `lookup`, on peut écrire une fonction `associeCadeaux` qui prend en argument la liste de souhaits et la liste de cadeaux, et affecte à chaque souhait son numéro de produit s'il existe.

</> Code

```
associeCadeaux :: [(String, [String])] -- liste de souhaits initiale
               -> [(String, Int)]      -- cadeaux possibles
               -> [(String, [Maybe Int])] -- liste modifiée
associeCadeaux l cdx = map (
    \(nom, souhaits) -> (nom, map (\s -> lookup s cdx) souhaits)
    ) l
```

✎ Exemple

Avec la liste de cadeaux possibles :

</> Code

```
cadeaux = [("Peluche", 114), ("Livre", 882), ("Jeu", 374), ("Figurine",
↪  264)]
```

On aurait :

</> Code

```
ghci> associeCadeaux liste cadeaux
[("Papa",[Just 2]),("Maman",[Just 3,Just 4]),("Prof",[Nothing])]
```

Les prix en magasin des cadeaux sont également stockés dans une liste associative. Cette fois, la clé est le numéro de produit et la valeur le prix.

Q. 2 Écrire une fonction `associePrix :: [(String, [Maybe Int])] -> [(Int, Int)] -> [(String, [Maybe Int])]` qui prend la liste transformée par `associeCadeaux` et renvoie la liste des prix des cadeaux demandés par chaque personne. Le code de cette fonction est **très** similaire à celui d'`associeCadeaux`, mais utiliser le fait que **Maybe** soit une monade est ici **nécessaire**.

Exemple

Avec les prix suivants :

Code

```
magasin = [(114, 12), (882, 20), (374, 45), (617, 10)]
```

On aurait :

Code

```
ghci> associePrix (associeCadeaux liste cadeaux) magasin
[("Papa",[Just 20]),("Maman",[Just 45,Nothing]),("Prof",[Nothing])]
```

Éléments de correction

Code

```
associePrix :: [(String, [Maybe Int])] -> [(Int, Int)] -> [(String,
↪  [Maybe Int])]
associePrix l mag = map (\(name, cads) -> (name, map (\c -> c >>= \id ->
↪  lookup id mag) cads)) l
```

Q. 3 En utilisant la fonction `fromMaybe`, écrire une fonction `budget`, qui prend la liste transformée par `associePrix` et qui renvoie pour chaque personne, le budget total à consacrer.

Exemple

On aurait alors finalement :

Code

```
ghci> budget (associePrix (associeCadeaux liste cadeaux) magasin)
[("Papa",20),("Maman",45),("Prof",0)]
```

Éléments de correction

Code

```
budget :: [(String, [Maybe Int])] -> [(String, Int)]
budget l = map (\(name, prix) -> (name, sum (map fromMaybe prix))) l
```

Fonctions utiles

`parsePlaylist :: Parser Playlist`
- Parseur de Playlist.

`evalParser :: Parser a -> String -> Maybe a`
- Évalue le résultat d'un parseur sur une chaîne.

`putStrLn :: String -> IO ()`
- Imprime une chaîne sur la sortie standard.

`readFile :: FilePath -> IO String`
- Lit un fichier

`writeFile :: FilePath -> String -> IO ()`
- Écrit une chaîne dans un fichier.

`getLine :: IO String`
- Lit une chaîne depuis l'entrée standard.

`unlines :: [String] -> String`
- Joint une liste de chaînes par un `"\n"`.

`zip :: [a] -> [b] -> [(a,b)]`

`zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]`
- Joignent les éléments de deux listes deux à deux (`zipWith` prend une fonction de jointure).

`null :: [a] -> Bool`
- Renvoie **`True`** si la liste est vide.

`foldr :: (a -> b -> b) -> b -> [a] -> b`

`foldl :: (b -> a -> b) -> b -> [a] -> b`
- Replient des listes par la droite ou la gauche.

`map :: (a -> b) -> [a] -> [b]`
- Applique une fonction à tous les éléments d'une liste.

`sum :: Num a => [a] -> a`
- Somme les élements d'une liste.

`concat :: [[a]] -> [a]`
- Concatène les sous-listes.

`fromMaybe :: Maybe Int -> Int`
- Renvoie `0` si **`Nothing`** et `n` si **`Just`** n.

Idées cadeaux

Comme vous avez été sages, voici une petite liste d'idées cadeaux.

- Le mug "Keep calm and Map On"
- Le t-shirt "I fold under pressure"
- La peluche Monade Câline
- Les chaussettes "Functor" et "Applicative" (à ne pas mélanger dans la machine)
- Le parfum "Essence de Curry"
- Une figurine du "Professeur", le super-héros du lambda-calcul
- Le livre "Haskell pour les chats (et autres créatures paresseuses)"
- Une écharpe ornée de "map", "foldr" et "filter" en motif répété.

Q. Bonus Choisissez votre cadeau préféré dans la liste et expliquez pourquoi.