Interrogation Écrite 3

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 les parseurs ;
- 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 : L'AEI et les pingouins-manchots [~ 8 pts]

Les chercheurs de l'Association d'Élite des Informaticiens (ou AEI) ont reçu un message étrange de la part d'une tribu extraterrestre inconnue, qui se présentent comme des pingouins (ou des manchots)[1].

Il semblerait que le message soit encodé par des chiffres. Heureusement, vous êtes un membre éminent de cette association et vous avez suivi avec assiduité le cours de Programmation Fonctionnelle. Par conséquent, vous savez que vous pouvez *parser* le message afin de le rendre lisible par vos collègues.

Informations

On suppose dans cet exercice que vous avez accès à la librairie **Parser** vue en TD. Notamment, les parseurs `car`, `carQuand` et `chaine` sont considérés comme déjà définis.

Les messages sont des suites de nombres séparés par des espaces, et chaque mot semble être séparé par un tiret. Le message reçu est : `"1 8 - 2 15 14 - 10 15 21 18"`. On se donne donc les deux parseurs supplémentaires suivants :

Code

```haskell
parseSpaces :: Parser () -- Passe les espaces et ne renvoie rien
parseSpaces = many (carQuand isSpace) >> return ()

parseSep :: Parser () -- Passe le séparateur et les espaces suivantes
parseSep = car '-' >> parseSpaces >> return ()
```

Q. 1 Écrire un parseur `nombre` qui reconnaît un nombre (c'est à dire une chaîne de caractères où tous les caractères sont des chiffres). Vous pouvez utiliser la fonction `isDigit`.

Éléments de correction

Code

```haskell
nombre :: Parser String
nombre  = some (carQuand isDigit)
```

Q. 2 Écrire un parseur `parseInt :: Parser Int` qui reconnaît un nombre et le convertit en entier, en supprimant les espaces qui suivent le nombre.

1. oui, c'est bizarre, mais bon

✔ Éléments de correction

</> Code

```haskell
parseInt :: Parser Int
parseInt = nombre >>= \n -> parseSpaces >> return (read n)
```

Q. 3 On se donne le nouveau parseur `parseIntList :: Parser [Int]`, qui applique plusieurs fois (au moins une) le parseur `parseInt`.

On veut maintenant reconnaître la liste des mots, chacun représenté par une liste d'entiers. Les mots sont séparés par un séparateur que l'on sait reconnaître.

Écrire un parseur `parseWords :: Parser [[Int]]` qui reconnaît la liste des mots.

✎ Informations

Attention, le premier mot n'a pas de séparateur devant ! Il peut également y avoir un nombre arbitraire de mots (mais on suppose au moins un).

✔ Éléments de correction

</> Code

```haskell
parseWords :: Parser [[Int]]
parseWords = (:) <$> parseIntList <*> many (parseSep >> parseIntList)
```

Q. 4 On veut maintenant décoder (enfin) le message. Sachant que l'on met à disposition les fonctions suivantes :

</> Code

```haskell
decodeWord :: [Int] -> String -- change chaque entier dans le bon char
decodeWord = map (\i -> chr (i + 64))

decodeWords :: [[Int]] -> String -- décode les sous listes + joint avec ' '
decodeWords = unwords . map decodeWord
```

Écrire la fonction `decode :: String -> Maybe String` qui prend en paramètre le message reçu, le parse, le décode, et renvoie le résultat.

Vous pouvez utiliser la fonction `evalParser :: Parser a -> String ->` a, qui évalue un parseur sur une entrée donnée.

✔ Éléments de correction

</> Code

```haskell
decode :: String -> Maybe String
decode s = decodeWords <$> evalParser parseWords s
```

Exercice 2 : Le repos de l'étudiant [~ 8 pts]

Après un DS ou une interro de PF difficiles, vous aimez bien vous reposer en fabriquant des cocktails goûteux. Vous êtes fatigué, alors votre cocktail contient uniquement trois ingrédients. Vous voulez mesurer la pureté de votre cocktail, qui est la moyenne des proportions des ingrédients.

Q. 1 Écrire une fonction `moyenne` qui prend trois paramètres de type **`Double`** et qui renvoie un **`Double`** qui est la moyenne des trois paramètres.

✔ Éléments de correction

</> Code

```haskell
moyenne :: Double -> Double -> Double -> Double
moyenne x y z = (x + y + z) / 3
```

Q. 2 Bien sûr, vous êtes *vraiment* fatigué, alors vous n'êtes pas sûr de vos proportions. Vous vous rendez compte que vous pourriez avoir oublié un ingrédient ou quelque chose comme ça, et il ne faudrait pas que le calcul de la moyenne plante...

Écrire une fonction `pureteCocktail` qui prend trois paramètres de type **`Maybe Double`** et qui renvoie la moyenne sous la forme d'un **`Maybe Double`**. Pour cela, utiliser la fonction `moyenne` définie précédemment et le fait que **`Maybe`** soit un foncteur applicatif (pas de **`case of`**).

✔ Éléments de correction

</> Code

```haskell
pureteCoktail :: Maybe Double
              -> Maybe Double
              -> Maybe Double
              -> Maybe Double
pureteCoktail p1 p2 p3 = moyenne <$> p1 <*> p2 <*> p3
```

Q. 3 Une proportion doit être un nombre entre 1 et 100 (inclus).

Écrire une fonction `valide :: Maybe Double -> Maybe Double` qui renvoie la proportion passée en paramètre si elle est valide et échoue sinon.

✔ Éléments de correction

</> Code

```haskell
valide :: Maybe Double -> Maybe Double
valide (Just n) | n >= 1 && n <= 100 = Just n
valide _        = Nothing
```

Q. 4 Il faut maintenant utiliser ça dans notre méthode de calcul de pureté de cocktail.

En tirant profit du fait que Maybe soit une monade, réécrire la fonction `pureteCocktail` dans les styles monadique, **`do`** et applicatif.

Éléments de correction

Code

```haskell
pureteCoktail' :: Maybe Double
               -> Maybe Double
               -> Maybe Double
               -> Maybe Double
pureteCoktail' p1 p2 p3 = do
    x <- valide p1
    y <- valide p2
    z <- valide p3
    return $ moyenne x y z

pureteCoktail'' :: Maybe Double
                -> Maybe Double
                -> Maybe Double
                -> Maybe Double
pureteCoktail'' p1 p2 p3 = valide p1
                           >>= \x -> valide p2
                           >>= \y -> valide p3
                           >>= \z -> return $ moyenne x y z

pureteCoktail''' :: Maybe Double
                 -> Maybe Double
                 -> Maybe Double
                 -> Maybe Double
pureteCoktail''' p1 p2 p3 = moyenne <$> valide p1
                                    <*> valide p2
                                    <*> valide p3
```

Q. 5 (Bonus) Ajouter à la fonction `pureteCocktail` une vérification du fait que la somme des proportions doit également être comprise entre 1 et 100 (n'écrire qu'une seule version).

Informations

Vous pouvez utiliser la fonction `add :: Double -> Double -> Double -> Double` qui additionne trois éléments.

Éléments de correction

Code

```haskell
pureteCoktailBonus :: Maybe Double
                   -> Maybe Double
                   -> Maybe Double
                   -> Maybe Double
pureteCoktailBonus p1 p2 p3 = do
    x <- valide p1
    y <- valide p2
    z <- valide p3
    valide (add <$> p1 <*> p2 <*> p3)
    return $ moyenne x y z
```

Exercice 3 : Un vrai dialogue de sourds [~ 8 pts]

Vous avez (enfin) trouvé un stage ! Vous allez travailler avec un célèbre acteur de film d'auteur, qui a besoin d'un petit coup de main.

Comme il commence à vieillir, George Blooney [2] n'arrive plus à lire les scripts de ses films (et il est mauvais en improvisation). Il vous demande donc de lire le script pour lui et de ne lui donner que les lignes qui le concernent.

Globalement, un script, ça ressemble à ça [3] :

Exemple - Tiré du célèbre film "L'étrange monade de George Hack" [2]

George	:	Professeur, je tenais à vous dire, vos explications sur les monades...
George	:	C'est comme de la poésie en lambda-calcul !
Francis	:	Vous avez bien compris, au moins ?
George	:	Bien sûr, je la ressens même, comme une oeuvre abstraite !
Francis	:	Donc, vous ressentez la monade, mais... vous savez la coder ?
George	:	Disons que dans mon coeur, la monade est pure.
George	:	Dans mon code, elle a encore quelques... erreurs de syntaxe

Comme vous êtes un étudiant en informatique doué et par conséquent un peu paresseux, vous vous souvenez que vous avez vu les parseurs en PF.

On se donne le modèle de données suivant :

Code

```haskell
data Phrase = Phrase { -- Une phrase, c'est du texte dit par un acteur
    acteur :: String,
    texte  :: String
} deriving Show

type Script = [Phrase] -- Un script, c'est une liste de phrases
```

Vous avez récupéré de vos anciens TP les parseurs suivants :

Code

```haskell
parseSpaces :: Parser () -- Passe les espaces et ne renvoie rien
parseSpaces = many (carQuand isSpace) >> return ()

parseSep :: Parser () -- Passe le séparateur et les espaces suivantes
parseSep = car ':' >> parseSpaces >> return ()

parseNewLine :: Parser () -- Passe un '\n' et les espaces suivantes
parseNewLine = car '\n' >> parseSpaces >> return ()
```

Q. 1 Écrire un parseur `parseName` qui parse le nom de l'acteur qui parle et supprime les espaces suivantes. Vous pouvez utiliser la fonction `isAlpha`, qui reconnaît les lettres.

2. Toute ressemblance avec des personnes ou des productions existantes est fortuite
3. Bien sûr, © ChatGPT Productions 2024, tous droits réservés

✔ Éléments de correction

</> Code

```haskell
parseName :: Parser String
parseName = some (carQuand isAlpha) >>= \n -> parseSpaces >> return n
```

Q. 2 Écrire un parseur `parseText` qui parse le texte de l'acteur qui parle et passe le prochain retour à la ligne. Comme il peut y avoir de la ponctuation, vous pouvez utiliser la fonction `isPrint`, qui reconnaît les caractères imprimables (caractères, espaces, ponctuation, mais pas les retours à la ligne).

✔ Éléments de correction

</> Code

```haskell
parseText :: Parser String
parseText = some (carQuand isPrint) >>= \s -> parseNewLine >> return s
```

Q. 3 En utilisant les parseurs que vous venez de définir, écrire un parseur `parsePhrase` qui parse une phrase complète (donc le nom de l'acteur et le texte) et la renvoie sous la forme d'une **Phrase**.

✔ Éléments de correction

</> Code

```haskell
parsePhrase :: Parser Phrase
parsePhrase = do
    name <- parseName
    parseSep
    text <- parseText
    return $ Phrase name text
```

Q. 4 Écrire un parseur qui permet de reconnaître un **Script** et donc, une liste (qui pourrait être vide) de **Phrase**.

✔ Éléments de correction

</> Code

```haskell
parseScript :: Parser Script
parseScript = many parsePhrase
```

Q. 5 En utilisant `evalParser`, écrire une fonction `filtreLignes`, qui prend une chaîne représentant un script en paramètre, la parse en Script et filtre les phrases pour ne garder que celles de George.

✔ Éléments de correction

</> Code

```haskell
filtreLignes :: String -> Script
filtreLignes s = case (evalParser parseScript s) of
    Nothing -> []
    Just l  -> filter (\ph -> acteur ph == "George") l
```