Analyse syntaxique

Le but de ce TD est de créer un début de bibliothèque pour faire de l'analyse syntaxique qui nous servira pour le prochain TP.

Exercice 1 : Mise en place et parseur pour un caractère

Q1. Définissez un type `Resultat a` qui doit être un type **sûr** ayant comme paramètre un tuple dont le premier élément est le résultat de l'analyseur syntaxique, et le second le reste de la chaîne qui n'a pas été lu.

Q2. Ecrivez une structure de donnée `Parseur a` qui représente un analyseur syntaxique. Un analyseur syntaxique est une fonction qui prend une chaîne de caractère, et retourne un résultat.

Q3. Ecrivez une fonction `parser :: Parseur a -> String -> Resultat a` qui exécute un analyseur syntaxique sur une chaîne de caractères.

Q4. Ecrivez une fonction `lireCaractere :: Parseur Char` qui doit lire un caractère si c'est possible.

```
ghci> parser lireCaractere "un Message"
Just ('u','n Message')
ghci> parser lireCaractere ""
Nothing
```

Note: `lireCaractere` doit construire une fonction. Pour cela, il faut utiliser la syntaxe avec le `where`, pour vous aider, voici le début du code à compléter.

```
lireCaractere :: Parseur Char
lireCaractere = Cons f
    where f [] = ....
          f (h:t) = ....
```

Exercice 2 : Lire un caractère si <condition>

Pour réaliser un parseur qui lit un caractère sous certaines conditions, nous avons besoin d'implémenter d'autres outils.

Il nous faut d'autres briques de bases.

Q1. Ecrivez une fonction `echouer :: Parseur a` qui échoue toujours peut-importe la chaîne de caractères.

```
ghci> parser echouer "truc"
Nothing
ghci> parser echouer ""
Nothing
ghci> parser echouer "123"
Nothing
```

Q2. Ecrivez une fonction `retourner :: a -> Parseur a` qui retourne son paramètre peut-importe la chaîne de caractères de l'analyseur syntaxique.

```
ghci> parser (retourner 3) "truc"
Just (3,"truc")
ghci> parser (retourner "aba") ""
Just ("aba","")
ghci> parser (retourner 45) "123"
Just (45,"123")
```

On souhaite maintenant combiner nos analyseurs syntaxiques pour créer des analyseurs plus complexes. Un premier combinateur à implémenter sera celui de la composition, qui permet d'enchaîner les analyseurs les uns après les autres.

Voici le code de ce combinateur :

```haskell
(<.>) :: Parseur a -> (a -> Parseur b) -> Parseur b
(<.>) parseur1 parseur2 = Cons (\chaine -> case parser parseur1 chaine of
                                    Nothing      -> Nothing
                                    Just (resultat, reste) -> parser (parseur2 resultat) reste)
```

Le combinateur effectue l'analyse syntaxique `parseur1` sur la chaîne passée en paramètre. Si le `parseur1` échoue, alors on retourne Nothing. Si l'on récupère un résultat, dans ce cas, on utilise le résultat précédent dans l'analyse syntaxique `parseur2`.

Q3. Ecrivez une fonction `lireCaractereSi :: (Char -> Bool) -> Parseur Char` qui analyse une chaîne en fonction d'une condition.

```
ghci> parser (lireCaractereSi (\s -> s == 'a')) "aba"
Just ('a',"ba")
ghci> parser (lireCaractereSi (\s -> s == 'a')) "cba"
Nothing
```

Q4. Ecrivez une fonction `isDigit :: Char -> Bool` qui teste si un caractère est un chiffre ou non.

Q5. Ecrivez une fonction `lireChiffre :: Parseur Char` qui lit un caractère si c'est un chiffre, échoue sinon.

```
ghci> parser lireChiffre "123"
Just ('1',"23")
ghci> parser lireChiffre "1abc"
Just ('1',"abc")
ghci> parser lireChiffre "abc"
Nothing
```

Exercice 3 : L'étoile de Kleene

En théorie de langage, l'étoile de Kleene consiste à répéter 0 ou plusieurs fois un élément. On peut également appliquer l'étoile de Kleene sur nos analyseurs syntaxiques. Par exemple si l'on souhaite lire "zéro ou plusieurs chiffres" ou encore "un ou plusieurs chiffres".

Pour réaliser cela, on va avoir besoin du combinateur "ou", qui permet de réaliser une analyse syntaxique, et si cette dernière échoue, tente une autre analyse syntaxique.

Q1. En vous inspirant du code du combinateur `(<.>)`, écrivez une fonction `(<|>) :: Parseur a -> Parseur a -> Parseur a` correspondant au combinateur "ou".

Q2. Ecrivez une fonction `unOuPlus :: Parser a -> Parser [a]` qui applique un analyseur syntaxique une ou plusieurs fois. L'analyseur est appelé récursivement jusqu'à ce qu'il échoue.

```
ghci> parser (unOuPlus lireChiffre) "123"
Just ("123","")
ghci> parser (unOuPlus lireChiffre) "123abc"
Just ("123","abc")
ghci> parser (unOuPlus lireChiffre) "abc123abc"
Nothing
```

Q3. Ecrivez une fonction `zeroOuPlus :: Parser a -> Parser [a]` qui applique un analyseur syntaxique zéro ou plusieurs fois. L'analyseur est appelé récursivement jusqu'à ce qu'il échoue.

```
ghci> parser (zeroOuPlus lireChiffre) "123"
Just ("123","")
ghci> parser (zeroOuPlus lireChiffre) "abc123abc"
Just ("","abc123abc")
ghci> parser (zeroOuPlus lireChiffre) "123abc"
Just ("123","abc")
```

Q4. Ecrivez une fonction `lireNombre :: Parser Int` qui est un analyseur syntaxique permettant de lire des nombres.

Pour vous aider, voici une fonction qui convertit un caractère en un entier :

```haskell
charToInt :: Char -> Int
charToInt c = read [c]
```