REGEX

Parseur d'expressions régulières

L'objectif de cet exercice est de mettre en oeuvre un parseur d'expressions régulières simples et d'implémenter la fonction `pregMatch` qui permet de tester une chaîne de caractère par rapport à une expression régulière.

Déclaration des types pour les expressions régulières

Dans ce TP, une Regex peut être :

- `Mot` : Une chaîne de caractère composée uniquement de lettres.
- `UnCarQuelconque` : Correspond au `.` - C'est un joker, tous les caractères sont acceptés.
- `ZeroOuUn` : Correspond au `?` - Indique 0 ou 1 occurrence d'un caractère ou d'une expression régulière.
- `UnOuPlus` : Correspond au `+` - Indique au moins 1 occurrence d'un caractère ou d'une expression régulière.
- `ZeroOuPlus` : Correspond au `*` - Indique 0 ou plusieurs occurrences d'un caractère ou d'une expression régulière.
- `Ou` : Correspond au `|` - Autoriser deux valeurs possibles (Comme le combinateur `<|>`)
- `Et` : permets de chainer les expressions régulières (comme le combinateur `<.>`)

Prenons quelques exemples de représentation pour clarifier ces types. Voici une liste d'expressions régulières et leur représentation (de type `Regex`) :

REGEX	Représentation en Haskell
`UnMot`	`Mot "UnMot"`
`.`	`UnCarQuelconque`
`(mot)?`	`ZeroOuUn (Mot "mot")`
`(mot)*`	`ZeroOuPlus (Mot "mot")`
`(mot)+`	`UnOuPlus (Mot "mot")`
`(un)|(deux)`	`Ou (Mot "un") (Mot "deux")`
`UnM.t`	`Et (Et (Mot "UnM") UnCarQuelconque)) (Mot "t")`
`mot?`	`Et (Mot "mo") (ZeroOuUn (Mot "t"))`

Q1. Déclarer le type `Regex`.

Parseur d'expressions régulières

Des expressions simples

Commençons par parser les expressions régulières les plus simples possible : les mots et le caractère quelconque.

Q2. Définissez un analyseur syntaxique `mot :: Parseur Regex` qui analyse le premier mot apparaissant dans la chaîne.

Exemple :

```
Main> parser mot ""
Nothing
Main> parser mot "abc"
Just (Mot "abc","")
Main> parser mot "abc def"
Just (Mot "abc"," def")
```

Note : Un mot est une chaîne de caractères composée des caractères `[a-zA-z]`

Aide : Dans le TD nous avons fait un parseur de nombre en faisant les fonctions `isDigit`, `lireChiffre` et `lireNombre`. Sur le même principe, faites des fonctions intermédiaires `lireLettre` et `lireMot`, cela vous sera utilise

pour faire le parseur `mot`

Q3. Définissez un analyseur syntaxique `unCarQuelconque :: Parseur Regex` qui permet d'obtenir l'expression régulière correspondant à un caractère quelconque `.`

Exemple :

```
Main> parser unCarQuelconque "abc"
Nothing
Main> parser unCarQuelconque "."
Just (UnCarQuelconque,"")
Main> parser unCarQuelconque ".abc"
Just (UnCarQuelconque, "abc")
```

Q4. Définissez un analyseur syntaxique `regexSimple :: Parseur Regex` qui analyse un seul élément d'une expression régulière (actuellement: `unCarQuelconque` **OU** `mot`)

Q5. Définissez un analyseur syntaxique `regex :: Parseur Regex` qui analyse une expression régulière complète.

Exemple:

```
Main> parser regex "unMot"
Just (Mot "unMot", "")
Main> parser regex "."
Just (UnCarQuelconque,"")
Main> parser regex ".abc"
Just (Et UnCarQuelconque (Mot "abc"), "")
Main> parser regex ".abc ah ah"
Just (Et UnCarQuelconque (Mot "abc"), " ah ah")
```

Aide : La fonction s'écrit très bien avec une fonction de pliage. Cependant, ici, il ne faut pas initialiser l'accumulateur, l'accumulateur devant en fait être initialisé avec le premier élément de la liste, pour cela il existe les fonctions `foldl1 :: (a -> a -> a) -> [a] -> a` et `foldr1 :: (a -> a -> a) -> [a] -> a` qui n'ont pas besoin d'accumulateur.

Des expressions parenthésées

Nous pouvons donc maintenant écrire nos premières expressions régulières. Avant de passer aux opérateurs, il faut savoir parser une expression parenthésée.

Q6. Définissez un analyseur syntaxique `exprParenthesee :: Parseur Regex` qui reconnaisse une expression parenthésée contenant une regex.

Q7. Ajoutez `exprParenthesee` à la liste des regex possibles pour le parseur `regexSimple`.

Exemple:

```
Main> parser regex "(unMot)"
Just (Mot "unMot","")
Main> parser regex "(.)"
Just (UnCarQuelconque,"")
Main> parser regex "(unM.t)"
Just (Et (Et (Mot "unM") UnCarQuelconque) (Mot "t"),"")
Main> parser regex "(unM.t)(unAutreMot)"
Just (Et (Et (Et (Mot "unM") UnCarQuelconque) (Mot "t")) (Mot "unAutreMot"),"")
```

Des répétitions

Q8. Définissez les trois analyseurs syntaxiques suivants :

- `zeroOuUnRegex :: Parseur Regex` (équivalent au `?`)
- `zeroOuPlusRegex :: Parseur Regex` (équivalent au `*`)
- `UnOuPlusRegex :: Parseur Regex` (équivalent au `+`)

Attention : Les opérateurs sont toujours précédés par une expression parenthésée. En pratique ce n'est pas le cas, mais ici c'est pour simplifier la tâche.

Exemple :

```
Main> parser zeroOuUnRegex "(a)?"
Just (ZeroOuUn (Mot "a"), "")
Main> parser zeroOuPlusRegex "(unMot)*"
Just (ZeroOuPlus (Mot "unMot"), "")
Main> parser unOuPlusRegex "(unMot)+"
Just (UnOuPlus (Mot "unMot"), "")
```

Vous remarquerez que vos trois parseurs se ressemblent très fortement, il peut-être intéressant de factoriser votre code pour n'avoir qu'une seule fonction au lieu de trois. Cela permet d'éviter de Parser plusieurs fois une même expression parenthésée, juste parce que le caractère suivant n'est pas l'un des trois opérateurs.

Q7. Proposez une fonction `repetition :: Parser Regex` qui définit le bon nombre d'occurrences autorisées (`ZeroOuUn`, `UnOuPluset` `ZeroOuPlus`) d'une expression régulière en fonction des opérateurs `?`, `+` et `*`. Vous veillerez à ne parser l'expression parenthésée qu'une seule fois.

Q8. Ajoutez `repetition` à la liste des regex possibles pour le parseur `regexSimple`.

Le ou

Q9. Définissez un analyseur syntaxique `ou :: Parseur Regex` qui définit le "ou" d'une expression régulière.

Exemple

```
Main> parser ou "(chien)|(chat)"
Just (Ou (Mot "chien") (Mot "chat"), "")
```

Q10. Ajoutez `repetition` à la liste des regex possibles pour le parseur `regexSimple`.

```
Main> parser regex "tr(i(s)?)|(op)"
Just (Et (Mot "tr") (Ou (Et (Mot "i") (ZeroOuUn (Mot "s"))) (Mot "op")),"")
```

Evaluation d'une expression régulière

Maintenant que nous savons parser une expression régulière, nous devons construire un parseur qui permet de vérifier une chaîne de caractères par rapport à l'expression régulière construite.

Voici une fonction qui vous sera utile pour cette partie :

```
lireMotExact :: [Char] -> Parseur [Char]
lireMotExact ""    = retourner ""
lireMotExact (h:t) = lireCaractereSi (\x -> x == h) <.> \_ ->
                     lireMotExact t <.> \_ ->
                    retourner (h:t)
```

Cette fonction permet de parser un mot exactement et de consommer toutes les lettres de ce mot.

Exemple

```
Main> parser (lireMotExact "test") "teste"
Just ("test","e")
Main> parser (lireMotExact "test") "test"
Just ("test","")
Main> parser (lireMotExact "test") "texte"
Nothing
```

Q11. Définissez un analyseur syntaxique `match :: Regex -> Parseur ()` qui analyse si une chaîne est valide par rapport à une expression régulière. Si elle l'est, elle consomme les caractères de la chaîne, sinon elle renvoie `Nothing`

Exemple :

```
Main> parser (match (Ou (Mot "chien") (Mot "chat")) "chien"
Just ((), "")
Main> parser (match (Ou (Mot "chien") (Mot "chat")) "chat"
Just ((), "")
Main> parser (match (Ou (Mot "chien") (Mot "chat")) "chatton"
Just ((), "ton")
Main> parser (match (Ou (Mot "chien") (Mot "chat")) "cheval"
Nothing
```

Aide : Il s'agit ici d'indiquer pour chaque élément du type Regex quel est le parseur à utiliser.

Nous avons maintenant tous les éléments pour écrire la fonction `pregMatch :: String -> String -> Bool` qui prend en paramètre une expression régulière, puis la chaîne à analyse, et enfin le résultat l'évaluation. La fonction doit retourner True, si et seulement si l'expression régulière est syntaxiquement correcte et si tous les caractères de la chaîne à analyser ont été consommés.

Q12. Définissez la fonction `pregMatch :: String -> String -> Bool`

```
Main> pregMatch "tr(i(s)?)|(op)" "triste"
False
Main> pregMatch "tr(i(s)?)|(op)" "tris"
True
Main> pregMatch "tr(i(s)?)|(op)" "tri"
True
Main> pregMatch "tr(i(s)?)|(op)" "trop"
True
Main> pregMatch "tr(i(s)?)|(op)" "triss"
False
Main> pregMatch "tr(i(s)?)|(op)" "tros"
False
```