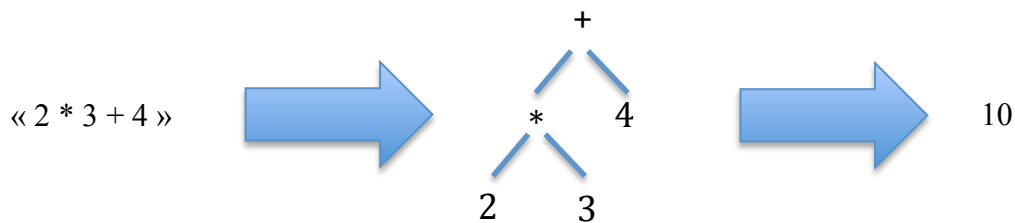


TP n°7 – Analyseur syntaxique

Ce TP porte sur la construction d'un analyseur syntaxique du type monadique.

Un analyseur syntaxique (« *syntactic parser* » en anglais) vise à tester la validité d'une expression donnée conformément à une grammaire tout en construisant comme résultat un arbre syntaxique par exemple. Un arbre syntaxique est facilement évaluable car il suffit de le parcourir et d'évaluer chaque nœud de l'arbre. Le résultat peut cependant prendre une autre forme de structure qu'un arbre.



Le type Parser

Un analyseur syntaxique « *parse* » une chaîne et construit un résultat. Néanmoins, l'analyse peut être partielle (toute la chaîne en entrée n'est pas analysée) ou peut complètement échouer. D'autre part, comme il est indiqué précédemment, le résultat peut prendre différentes formes de structure. Enfin, comme vous allez instancier différentes classes, le type doit être déclarée comme un nouveau type et non comme un type synonyme.

Pour toutes ces raisons, vous utiliserez le type suivant :

```
data Parser a = P (String -> [(a, String)])
```

Un type *Parser* est donc un type encapsulant une fonction. Cette fonction prend une chaîne et retourne une liste de couples. Le couple est constitué de la structure résultat de type *a* et de la chaîne restant à analyser. Si la liste est vide, l'analyse a complètement échouée sinon une liste singleton (un seul couple) est retournée comprenant une analyse complète ou incomplète.

L'analyse et le premier analyseur

- a) Ecrivez une fonction nommée `item :: Parser Char` qui retourne un parser. La fonction (à un argument) encapsulée échoue si la chaîne est vide et retourne comme résultat le 1^{er} caractère de la chaîne dans le cas contraire (regardez l'exemple de l'exercice suivant).

Complétez le code suivant :

```
item :: Parser Char
item = P (\s -> case ... of
    [] -> []
    x:xs -> ... )
```

- b) Ecrivez la fonction `parse :: Parser a -> String -> [(a, String)]`. Il s'agit d'une fonction qui prend comme 1^{er} arg. un parser et comme 2^{ième} arg. la chaîne à traiter. La fonction retourne le résultat du *parser* appliqué à la chaîne.

```
Exemples : parse item "abc" == [('a', "bc")]
           parse item ""   == []
```

Appliquer des analyseurs en séquence

L'objectif est maintenant de pouvoir enchaîner l'application de plusieurs analyseurs en utilisant l'expression *do*. Pour cela, nous allons successivement définir le type *Parser* comme instance de la classe *Functor* puis de la classe *Applicative* et enfin de la classe *Monad* (importez le module *Control.Applicative*).

- c) Complétez le code suivant pour instancier la classe de type *Functor*.

Le but est de pouvoir appliquer une fonction unaire sur le contenu d'un premier *Parser*. On veut donc obtenir un nouveau *Parser* dont le résultat de la fonction encapsulée sera une transformation par la fonction unaire du résultat du premier *Parser*.

```
instance Functor Parser where
    -- fmap :: (a -> b) -> Parser a -> Parser b
    fmap f p = P (\s -> case ... of
        [] -> ...
        [(r, s')] -> [(f r, s')])
```

```
Exemples : parse (fmap toUpper item) "abc" == [('A', "bc")]
           parse (fmap toUpper item) ""   == []
```

Rappel : *toUpper* est une fonction du module *Data.Char*.

- d) Complétez le code suivant pour instancier la classe de type *Applicative*.

Avec la classe *Applicative*, on s'intéresse à des types qui encapsulent une fonction. Cette fois-ci le but est de pouvoir appliquer cette fonction qui est le résultat de l'application d'un *Parser* et d'obtenir un nouveau *Parser*. Il s'agit d'instancier la fonction `<*>`.

Il faut aussi instancier la fonction *pure* qui transforme une valeur de type quelconque en valeur de type *Parser* c'est-à-dire en fonction de type *Parser*.

```
instance Applicative Parser where
  -- pure :: a -> Parser a
  pure a = P ...

  -- <*> :: Parser (a -> b) -> Parser a -> Parser b
  pf <*> p = P (\s -> case ... of
    [] -> ...
    [(f, s')] -> parse (fmap ...) s')
```

Exemples :

```
> :type pure toUpper
pure toUpper :: Applicative f => f (Char -> Char)

parse (pure 1) "abc" == [(1, "abc")]
parse (pure toUpper <*> item) "abc" == [('A', "bc")]
```

- e) La dernière étape consiste à instancier la classe *Monad* afin de pouvoir appliquer en séquence des parsers avec l'expression *do*. Pour instancier la classe *Monad*, il faut instancier la fonction *return* et la fonction *>>=*.

Complétez le code suivant :

```
instance Monad Parser where
  -- return :: a -> Parser a
  return = ...

  -- (>>=) :: Parser a -> (a -> Parser b) -> Parser b
  p >>= f = P (\s -> case ... of
    [] -> ...
    [(r, s')] -> parse ... s')
```

Exemples :

```
p1 = item >>= (\c -> fmap toUpper (return c))
parse p1 "abc" == [('A', "bc")]

p2 = item >>= (\c -> return toUpper <*> return c)
parse p2 "abc" == [('A', "bc")]
```

- f) Ecrivez maintenant un parser monadique nommé *three* qui analyse les trois premiers caractères d'une chaîne mais ne retourne dans un tuple que le 1^{er} et le 3^{ème} caractère.

Utilisez pour cela la notation *do* en réécrivant le parser suivant :

```
three :: Parser (Char,Char)
three = item >>= (\c -> item >>= (\_ -> item >>= (\e -> pure (c,e))))
```

Rappel : dans l'opérateur `>>=` la fonction (2ème paramètre) n'est pas appliquée si la monade (1er paramètre) n'est pas « valide ». En conséquence, dans une expression *do*, l'exécution des traitements s'arrête dès qu'un des traitements échoue !

Exemples : `parse three "" == []` car le 1^{er} parser échoue
`parse three "a" == []` car le 2^{ième} parser échoue
`parse three "ab" == []` car le 3^{ième} parser échoue
`parse three "abcdef" == [(('a','c'), "def")]` car les 3 parsers réussissent

Appliquer alternativement des analyseurs

Dans la partie précédente, nous avons instancié la classe *Monad* pour pouvoir appliquer des parsers en séquence les uns après les autres en utilisant l'expression *do* réservée aux types monadiques, la sortie des uns étant l'entrée des autres.

Il existe cependant une autre manière d'appliquer deux traitements : c'est d'appliquer le second traitement sur la même entrée que le premier traitement si ce premier traitement échoue. Le second traitement est donc une alternative au premier traitement.

Il existe pour cela la classe de type *Alternative* définie dans le module *Control.Applicative* de la manière suivante :

```
class Applicative f => Alternative f where
  empty :: f a
  (<|>) :: f a -> f a -> f a
  many :: f a -> f [a]
  some :: f a -> f [a]

  many x = some x <|> pure []           -- implémentation par défaut
  some x = pure (:) <*> x <*> many x    -- implémentation par défaut
```

empty définit une alternative qui échoue tout le temps, tandis que *<|>* définit une alternative entre deux traitements de même type. Ainsi, symboliquement on peut dire que :

$$\begin{aligned} \text{empty} <|> x &= x \\ x <|> \text{empty} &= x \\ x <|> (y <|> z) &= (x <|> y) <|> z \end{aligned}$$

Les fonctions *many* et *some* sont expliquées plus loin dans la partie « *Applications répétitives de parsers* » du sujet de TP.

Un exemple d'instanciation de la classe *Alternative* est le type *Maybe* défini de la manière suivante :

```
instance Alternative Maybe where
  -- empty :: Maybe a
  empty = Nothing

  -- (<|>) :: Maybe a -> Maybe a -> Maybe a
  Nothing <|> x = x
  (Just x) <|> _ = Just x
```

Vous allez maintenant écrire l'instance de la classe *Alternative* pour un type *Parser* afin de pouvoir définir des analyseurs alternatifs.

g) Complétez le code suivant :

```
instance Alternative Parser where
  -- empty :: Parser a
  empty = ...

  -- (<|>) :: Parser a -> Parser a -> Parser a
  p1 <|> p2 = P (\s -> case ... of
    [] -> ...
    [(r, s')] -> ...
```

Exemples :

```
parse empty "abc" == []
parse (item <|> return 'd') "abc" == [('a', "bc")]
parse (empty <|> return 'd') "abc" == [('d', "abc")]
parse (item <|> item) "abc" == [('a', "bc")]
```

Parsers primitifs

Vous avez maintenant à votre disposition plusieurs *parsers* simples que vous allez pouvoir combiner ensemble afin d'obtenir des *parsers* plus évolués. Vous avez à votre disposition :

- le parser *item* qui traite le premier caractère de la chaîne ;
- le parser *empty* qui échoue tout le temps ;
- et le parser *return* qui marche tout le temps.

h) Ecrivez avec l'expression *do* un parser de type `sat :: (Char -> Bool) -> Parser Char` qui marche si le premier caractère vérifie un prédicat donné en paramètre et qui échoue sinon.

Exemples :

```
parse (sat isDigit) "abc" == []
parse (sat isDigit) "1abc" == [('1', "abc")]
```

i) A partir du parser *sat* et des fonctions du module *Data.Char* écrivez les parsers suivant :

```
digit :: Parser Char qui parse les chiffres
lower :: Parser Char qui parse les lettres minuscules
```

```
upper :: Parser Char qui parse les lettres majuscules
letter :: Parser Char qui parse toutes les lettres
alphanum :: Parser Char qui parse les lettres et les chiffres
char :: Char -> Parser Char qui parse le caractère donné en argument
```

```
Exemples : parse digit "1abc" == [('1', "abc")]
           parse lower "Abc" == []
           parse upper "Abc" == [('A', "bc")]
           parse letter "1abc" == []
           parse alphanum "1abc" == [('1', "abc")]
           parse (char 'a') "abc" == [('a', "bc")]
```

- j) Ecrivez avec une expression *do* et avec le parser *char* un parser récursif de type `string :: String -> Parser String` qui marche si le motif donné en paramètre correspond au début de la chaîne à analyser.

Complétez le code suivant :

```
string :: String -> Parser String
string [] = return []
string (x:xs) = do char x
                  ...
                  return ...
```

```
Exemple : parse (string "abc") "abcdef" == [("abc", "def")]
          parse (string "abc") "ab123" == []
```

Applications répétitives de parsers

Dans la classe de type *Applicative* vue précédemment, les fonctions *many* et *some* sont deux fonctions de répétitions d'un foncteur applicatif donné en paramètre. La fonction *many* permet d'appliquer zéro ou plusieurs fois le foncteur applicatif, tandis que la fonction *some* applique au moins une fois le foncteur applicatif.

```
Par exemples : parse (many digit) "123abc" == [("123", "abc")]
               parse (many digit) "abc" == [("", "abc")]
               parse (some digit) "abc" == []
               parse (some digit) "123abc" == [("123", "abc")]
```

- k) En utilisant une expression *do* et la fonction *many*, écrivez la fonction de type `ident :: Parser String` qui essaie de parser un identifiant c'est-à-dire une chaîne commençant par une minuscule puis pouvant comporter des lettres ou des chiffres.

```
Exemples : parse ident "abc def" == [("abc", " def")]
           parse ident "Abc def" == []
           parse ident "1Abc def" == []
           parse ident "a" == [("a", "")]
           parse ident "a123 bcd" == [("a123", " bcd")]
```

- l) Ecrivez le parser de type `nat :: Parser Int` qui essaie de parser un nombre naturel (entier positif ou nul) comportant un ou plusieurs chiffres. Pensez à utiliser la fonction *read* pour la conversion.

Exemples : `read "123" :: Int == 123`
`parse nat "123 456" == [(123, " 456")]`
`parse nat "0 abc" == [(0, " abc")]`

- m) Ecrivez le parser de type `space :: Parser ()` qui essaie de parser les caractères d'espaces (un ou plusieurs espaces, tabulations ou retour à la ligne). Utilisez le parser *sat* et le prédicat *isSpace* de *Data.Char*.

Exemples : `parse space " abc" == [((), "abc")]`
`parse space " abc " == [((), "abc ")]`

- n) Enfin écrivez un parser de type `int :: Parser Int` qui essaie de parser un nombre entier. Un entier est un nombre naturel pouvant éventuellement être négatif. Utilisez la fonction `<|>`.

Complétez le code suivant :

```
int :: Parser Int
int = do char '-'
      ...
      return ...
<|> ...
```

Exemples : `parse int "-123 abc" == [(-123, " abc")]`
`parse int "123 abc" == [(123, " abc")]`

Analyse syntaxique d'expressions arithmétiques

Dans cette dernière partie, vous allez définir un évaluateur d'expression arithmétique simple basée sur la grammaire suivante :

$$\begin{aligned} \text{expr} &::= \text{term } '+' \text{ expr} \mid \text{term} \\ \text{term} &::= \text{factor } '*' \text{ term} \mid \text{factor} \\ \text{factor} &::= '(' \text{ expr } ')' \mid \text{nat} \\ \text{nat} &::= 0 \mid 1 \mid 2 \mid 3 \mid \dots \end{aligned}$$

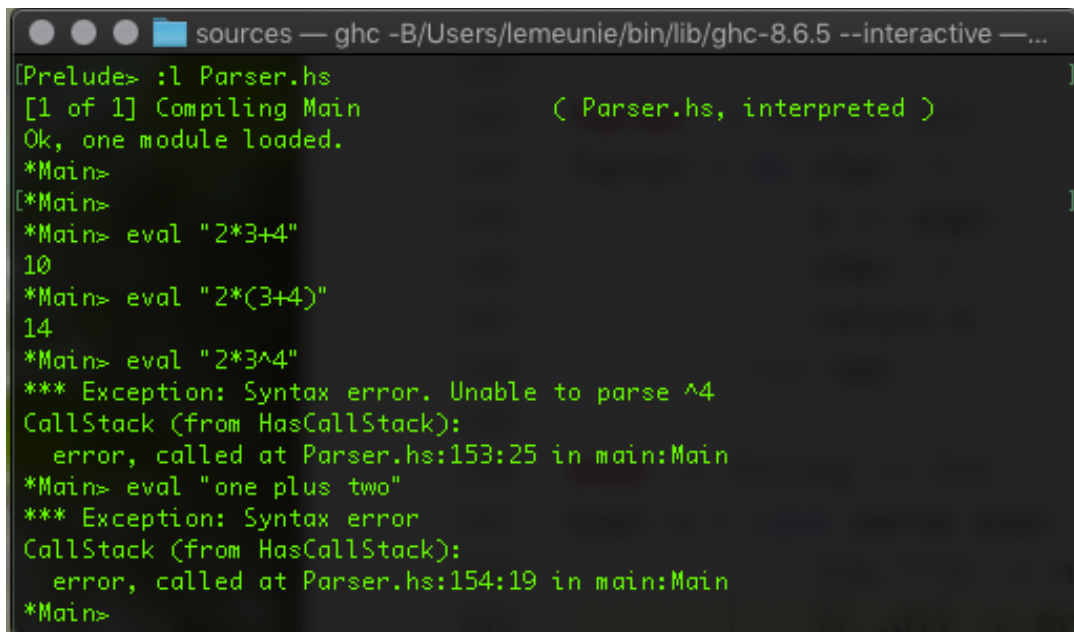
La grammaire permet d'écrire des expressions telles que :
`2+3+4` ou `(2*3)+4` ou `(2+3)+4 ...`

- o) Ecrivez un parser `expr :: Parser Int` pour parser une expression.
- p) Ecrivez un parser `term :: Parser Int` pour parser un terme.
- q) Ecrivez un parser `factor :: Parser Int` pour parser un facteur.

- r) Ecrivez une fonction d'évaluation `eval :: String -> Int` qui parse une expression et qui retourne un résultat ou lève une exception.

Complétez le code suivant :

```
eval :: String -> Int
eval s = case ... of
    ... -> n
    [(_,o)] -> ...
    [] -> error "Syntax error"
```



```
sources — ghc -B/Users/lemeunie/bin/lib/ghc-8.6.5 --interactive —...
[Prelude> :l Parser.hs
[1 of 1] Compiling Main           ( Parser.hs, interpreted )
Ok, one module loaded.
*Main>
[*Main>
*Main> eval "2*3+4"
10
*Main> eval "2*(3+4)"
14
*Main> eval "2*3^4"
*** Exception: Syntax error. Unable to parse ^4
CallStack (from HasCallStack):
  error, called at Parser.hs:153:25 in main:Main
*Main> eval "one plus two"
*** Exception: Syntax error
CallStack (from HasCallStack):
  error, called at Parser.hs:154:19 in main:Main
*Main>
```

Références web

Site officiel Haskell

<https://www.haskell.org>

Site des modules officiels Haskell

<https://downloads.haskell.org/~ghc/latest/docs/html/libraries/>