Programmation Fonctionnelle
Analyse syntaxique avec `Parsec`

Luigi Santocanale
LIF, Aix-Marseille Université
Marseille, FRANCE

14 octobre 2017

Plan

Une grammaire pour les expressions Booléennes

Grammaire :

```
bexpr -> ( bexpr && bexpr )
  | ( bexpr || bexpr )
  | constant | var
```

...et un type `Haskell` pour les expression Booléennes :

```
data BExpr = Const Bool | Var String
           | Conj BExpr BExpr
           | Disj BExpr BExpr deriving Show
```

Utilisation de la bibliothèque `Parsec`

```
import Text.Parsec
type Parser a = Parsec String () a
```

Des parseurs élémentaires :

```
lpar = char '('
rpar = char ')'
conj = string "&&"
disj = string "||"
```

Parser une BExpr

```
bexpr :: Parser BExpr
bexpr = try conjunction <|> disjunction
  <|> constant <|> var

var :: Parser BExpr
var = do
  string <- many letter
  return (Var string)

constant :: Parser BExpr
constant =
  (string "true" >> return (Const True))
  <|>
  (string "false" >> return (Const False))
```

Remarques :

- `Parser` est un instance de `Monad` : on utlise » et `return`
- on utilise `<|>` pour faire du « backtracking » et choisir une autre alternative.

```
conjunction  :: Parser BExpr
conjunction = do
  lpar
  e1 <- bexpr
  conj
  e2 <- bexpr
  rpar
  return (Conj e1 e2)

disjunction :: Parser BExpr
disjunction = do
  lpar
  e1 <- bexpr
  disj
  e2 <- bexpr
  rpar
  return (Disj e1 e2)
```

Pour terminer

```
import Data.Char

parseBExpr string = parse bexpr "" stream
  where
    stream = filter (not . isSpace) string
```

Une autre grammaire, sans `try`

Grammaire :

```
bexpr' -> ( bexpr' rest'
  | constant | var
rest' -> && bexpr' )
  |  || bexpr' )
```

```
bexpr' :: Parser BExpr
bexpr' = (lpar >> bexpr' >>= rest')
  <|> constant <|> var

rest' :: BExpr -> Parser BExpr
rest' e1  =
    (conj >> bexpr' >>= \e2 ->
      rpar >> return (Conj e1 e2))
    <|>
    (disj >> bexpr' >>=  \e2 ->
      rpar >> return (Disj e1 e2))

parseBExpr' =
  (parse bexpr' "") . (filter (not . isSpace))
```