PFA Laboratoire - Parseur

Christoph Rouff Soit Rueff, Tiago Povoa Quinteiro

Objectifs

- Créer un parseur capable de construire un arbre syntaxique abstrait à partir de la syntaxe suivante:
  - terme = facteur + terme | facteur
  - facteur = expression simple * facteur | expression simple
  - expression simple = entier | terme parenthésé
  - entier = [0..9]+
  - terme parenthésé = (terme)
- Ajouter la notion de variable à la syntaxe (définition et utilisation)
- Ajouter une fonction d'évaluation des expressions dans le style applicatif

Fichiers Modules

- `parser.hs` (module Parser)

Exemples d'exécution

Voici une série d'exemples d'exécution:

Note: Le parser gère un nombre arbitraire d'espaces.

```
*Parser> parse term "1   *3"
[(Operator (Value 1) '*' (Value 3),"")]

*Parser> interpret "1    *3"
3
```

```
*Parser> parse term "(2 + 2) * (3 + 2)"
[(Operator (Operator (Value 2) '+' (Value 2)) '*' (Operator (Value 3) '+' (Value
2)),"")]

*Parser> interpret "(2 + 2) * (3 + 2)"
20
```

```
*Parser> parse term "let a = (2 + 2) in a * (let x = 3 * 2 in x * a )"
[(Definition "let" (Variable "a") '=' (Operator (Value 2) '+' (Value 2)) "in"
(Operator (Variable "a") '*' (Definition "let" (Variable "x") '=' (Operator
(Value 3) '*' (Value 2)) "in" (Operator (Variable "x") '*' (Variable
"a")))),"")]

*Parser> interpret "let a = (2 + 2) in a * (let x = 3 * 2 in x * a )"
96
```

Implémentation

Expressions

Une expression est décrite ainsi:

```
data Expression =
  Value Int |
  Operator Expression Char Expression |
  Variable String |
  Definition String Expression Char Expression String Expression deriving Show
```

`Definition` sert à définir le `"let" Expression '=' Expression "in" Expression`.

Parser (Analyseur Syntaxique)

Point de départ: on a un **Parser** de type `a` qui tient une fonction définissant un "parsing" sur une string.

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

On va commencer par dériver les `Functor` qui vont nous être utile pour créer le **Parser** de manière applicative, donc `Functor`, `Applicative` et `Alternative`.

`Functor` permet de décrire comment une fonction doit être appliquée sur **Parser** en implémentant fmap qui va agir comme un adaptateur.

```
instance Functor Parser where
  fmap f (Parser p) = Parser (\cs -> [(f x, ys) | (x, ys) <- p cs])
```

`Applicative` permet de décrire comment une fonction dans un **Parser** est appliquée à un autre **Parser**, dans notre cas cela décrit comment s'effectue le chaînage de paramètres pour construire une expression englobante. (Un `Operator` est construit à partir de plusieurs `Expression` en paramètre)

```
instance Applicative Parser where
  pure x = Parser (\cs -> [(x, cs)])
  p <*> q = Parser (\cs -> [y | (x, ys) <- parse p cs, y <- parse (fmap x q)
ys])
```

`Alternative` permet de définir une alternative entre p et q. Si le "parsing" de p échoue, il essaie alors de parser q. Concrètement, cela fait office de "OU" dans notre syntaxe.

Exemple: une expression est un entier OU un groupe (terme parenthésé) OU une variable

```haskell
instance Alternative Parser where
  empty = Parser (\_ -> [])
  p <|> q = Parser (\cs -> case parse p cs of
    [] -> parse q cs
    ps -> ps)
```

Fonctions de parsing simples

Avant d'aborder les fonctions plus complexes de parsing de grammaire, on commence par définir des fonctions de parsing élémentaires.

```haskell
char c = Parser (split' (==c))
digit = Parser (split' isDigit)
letter = Parser (split' isLetter)
...
```

Note: ici `split'` prend un prédicat et une chaîne. Si le premier caractère de la chaîne est accepté par le prédicat, alors il renvoie celui-ci et la suite de la chaîne. Dans le cas contraire, renvoie une chaîne vide.

Fonctions de parsing de grammaire

En utilisant les parsers simples, on peut alors créer les fonctions qui vont parser la grammaire voulue.

```haskell
...
facteur = Operator <$> expression <* space <*> char '*' <*> facteur <|>
expression
...
```

Ici on définit ce qu'est un facteur tel que `facteur = expression simple * facteur | expression simple`, pour prendre un exemple parmi la grammaire donnée en consigne.

Note: `<*` et `*>` sont des opérateurs applicatifs qui permettent d'ignorer l'un des deux paramètres (gauche ou droite).

```haskell
(*>) :: Applicative f => f a -> f b -> f b
(<*) :: Applicative f => f a -> f b -> f a
```

Cela nous permet notamment de parser les espaces sans les inclures dans les paramètres des expressions.

Fonction parse

Grâce à tout ce qu'on a définit dans les sections précédentes, on peut alors simplement appeler la fonction `parse` qui prend en argument un **Parser** et une chaîne de caractères à parser ainsi:

```haskell
parse (Parser p) cs = p cs
```

Évaluation

Pour effectuer l'évaluation on définit une fonction `eval'` qui prend un environnement de manière implicite.

```
eval' :: Expression -> Map [Char] Int -> Int
```

Gestion de l'environnement

L'environnement est une **Map** de chaîne de caractère vers un entier. `Map [Char] Int`

Pour chercher le contenu d'une variable on a définit la fonction `lookup'`. Si la variable n'est pas définit, cela lève une erreur.

```
eval' (Variable v) = lookup' v
```

Pour ajouter une nouvelle variable on a définit `addvar`. Elle prend une variable (nom et contenu), l'expression dans laquelle elle doit être utilisée et un environnement. Elle opère de la manière suivante:

```
eval' (Definition _ (Variable v) _ e _ e1) = addvar v e e1
```

Elle crée un nouveau environnement où elle ajoute la variable qui a été évaluée avec l'environnement précédent, puis évalue l'expression où cette variable est utilisée, en utilisant ce nouvel environnement. De cette manière, même si on crée une nouvelle variable avec le même nom, elle sera définie correctement dans sa portée.

Exemple:

```
*Parser> interpret "let a = 2 in a * (let a = 3 in 1 + a)"
8
```

Interpréteur

Pour test facilement le chaînage de parsing et d'évaluation, on a ajouté la fonction `interpret` qui va d'abord parser puis évaluer une chaîne de caractères donnée, ou le cas échéant lever une erreur.

Conclusion

Ce qui est très élégant avec cette approche, c'est qu'on peut définir les fonctions de parsing presque comme on définirait une grammaire bnf.

Ce travail nous à aidé à comprendre le fonctionnement des Functor applicatifs.