Compilation en Haskell avec Parsec

Olivier Hermant

septembre 2020

1 Introduction

Le but de ce TP est de définir un interpréteur en ligne d'expressions arithmétiques, dans l'esprit du TP correspondant sur la calculatrice fait précédemment.

L'outil utilisé sera évidemment Haskell, plus spécifiquement la librairie *parsec* de combinateurs de parseurs monadiques.

2 Expressions Arithmétiques

2.1 Type de données

Proposez un type de données `AExpr` pour les expressions arithmétiques (avec opérateurs binaires) en nombres entiers. On utilisera de préférence le type `Int` (entiers machine) plutôt que `Integer` (entiers relatifs) pour les constantes. Pas de variable pour l'instant.

2.2 Évaluation d'une expression arithmétique

Programmez une fonction `eval :: AExpr -> Int`. Que se passe-t-il en cas de division par zéro ? Vous pouvez vous occuper de traiter ce comportement (en restant simple).

3 Interpréteur d'expresssions arithmétiques

3.1 Grammaire pour les expressions simples

Proposez une grammaire pour les expressions arithmétiques avec *opérateurs binaires* et *sans gestion de priorité entre opérateurs*. Pas de variables pour l'instant. On pourra utiliser des parenthèses, qui seront donc obligatoires.

Donnez-en une forme BNF.

3.2 Parsers

Commençons par importer le module

```
import Text.Parsec
```

Le type le plus général d'un parser est alors

```
ParsecT s u m a
```

Ce type est polymorphe en `s, u, m` et `a` :

- `s` est le type du flux d'entrée (typiquement : des caractères, mais on pourrait penser à des tokens)
- `u` est le type d'état utilisateur, si nécéssaire
- `m` est la monade interne, si nécéssaire

- a est le type de valeurs produites par l'exécution du parser

`ParsecT s u m a` est un ~~censuré~~[1]. Vous allez donc très rapidement avoir besoin de plonger dans ce monde, sauf pour les fonctions pures bien entendu, mais la majorité d'entre elles font partie de la Section 2 précédente.

On utilisera un type de parser simplifié

```
type Parser a = Parsec String () a
```

Ce qui correspond au type `ParsecT String () Identity a` (`Identity` étant le foncteur identité, après import de la librairie adéquate).

Nous n'aurons donc besoin ni d'état, ni d'une monade interne informative. Notons qu'un élément de type `Parser a` n'est pas une fonction qui produit quelque chose en soi, il faut l'exécuter une fois qu'il a été construit.

3.3 Parseurs élémentaires

Munis de ce type, nous pouvons définir un parser pour la parenthèse ouvrante

```
lpar :: Parser Char
lpar = char '('
```

`char` étant une fonction de la librairie parsec qui prend un caractère en entrée et renvoie un parseur produisant un caractère... on pourrait plutôt parler d'un lexeur, à ce niveau.

De la même manière, construisez un parser pour les autres symboles de votre grammaire (y compris les chiffres et les espaces).

3.4 Lancer un parseur

Cela se fait avec la fonction `parse`, qui prend trois arguments : le parseur, un nom de fichier (que l'on peut laisser vide) et le flux d'entrée, pour nous, une chaîne de caractères, c'est à dire une liste de caractètres. Par exemple :

```
parse lpar "" "("
```

La valeur de retour est empaquetée dans la monade `Either` car soit le parsing échoue, soit il réussit. Dans notre exemple, c'est réussi et nous obtienons le résultat

```
Right '('
```

`lpar` est censé produire des `Char` (empaquetés). En cas d'échec, nous aurons quelque chose du style `Left erreur`, et du type `ParseError`. Testez vos fonctions, et les cas d'erreur, avant de continuer.

3.5 Combiner les parseurs

Tout l'intérêt de `Parsec` est la possibilité de combiner des parseurs, presque exactement comme une syntaxe BNF. Ceci est possible car le type de données `Parser` est une instance de ~~censuré~~.[2]

1. Choix multiples. On peut former un "parseur de parenthèses" comme ceci :

   ```
   paren :: Parser Char
   paren = lpar <|> rpar
   ```

   L'effet de l'opérateur `<|>`, lorsqu'il sera exécuté, est de tenter tout d'abord `lpar` et, en cas d'échec, de tenter ensuite `rpar`.

2. backtracking : le flux d'entrée étant un flux, chaque parseur le consomme quand il le lit, y compris lorsqu'il échoue après le premier caractère lu. Pour remettre en place le flux d'entrée originel après un échec, il faut utiliser le combinateur `try`. Par exemple :

[1] transformateur de monade en la monade m, et lui-même une monade en a (entre autres)

[2] la typeclass `Alternative`.

```
eqSign = (try $ string "==") <|> string "="
```

Quel est le type de `eqSign` ? Parsez la chaîne "==" avec, puis sans `try`. Et d'autres chaînes "intéressantes".

3. Répétition de 0 à *n* fois un même parseur :

   ```
   many space
   ```

   où `space` est un parseur prédéfini pour les caractères représentant un espace.

4. Quels sont les types de `many`, de `try` et de `<|>` ? Et dans notre version simplifiée avec le type `Parser a` ?

5. etc. Voir la documentation en ligne de la librairie Parsec.

Pour vous exercer, construisez (et testez) un parseur pour :

- les chiffres (sans utiliser `digit`)
- les suites *non vides* de chiffres.

3.6 Analyseur syntaxique

On souhaite maintenant construire des parseurs qui ne produisent pas des caractères ou des chaînes de caractères, mais des expressions arithmétiques.

Voici un exemple de syntaxe pour un tel parseur :

```
constant = chiffre >>= \x -> return (Const 42)
```

Ceci suppose que votre type d'expressions arithmétiques possède un constructeur `Const` pour les constantes. Le parseur pour un chiffre (censé produire un caractère) est ensuite lié à la production de l'expression arithmétique `42` (empaqueté dans un Parser), le caractère reconnu étant ensuite lié à la variable `x`.

Une manière plus idiomatique de faire ce parseur serait d'utiliser le bloc `do` :

```
constant = do
  x <- chiffre
  return (Const 42)
```

et, comme on ne se sert par du caractère extrait par le parseur `chiffre`, on peut encore le simplifier en

```
constant = do
  chiffre
  return (Const 42)
```

ce qui équivaut à

```
constant = chiffre >> return (Const 42)
```

Construisez un parseur pour les expressions arithmétiques sans variables. Autorisez les espaces.

3.7 Interpréteur interactif

Il s'agit maintenant d'écrire toutes les fonctions qui font des entrées/sorties.

1. commencez par récupérer une ligne (`getLine`), puis la parser et en afficher le résultat.

2. répétez cette action indéfiniment. Pour être plus propre, trouvez un moyen de dire à votre calculatrice de s'arrêter.

4 Fabriquer des modules

Il est temps de faire plusieurs fichiers séparés. Pour créer un module nommé `Expression`, qui expose votre type de données d'expressions arithmétiques, ses constructeurs (courant et futurs) et la fonction d'évaluation on peut utiliser la syntaxe suivante

```
module Expression(AExpr(..),eval) where
```

Dans les autres fichiers on importe ensuite ce nouveau module avec `import Expression`.
Découpez votre programme en différents modules.

5 Une calculatrice avec variables

On souhaite maintenant ajouter à la calculatrice des variables.

5.1 Ajout de variables aux expressions

1. étendre le type de données `AExpr`
2. modifier `eval` de manière à pouvoir évaluer correctement une variable : quelle structure de données pour stocker les valeurs ? Quel impact sur le type de `eval` ? Que faire en cas d'échec ? On pourra en profiter pour rendre plus propre le traitement de la division par 0.

Note. En Haskell on peut introduire des synonymes de type comme ceci :

```
type ListeDeListes a = [[a]]
```

5.2 Modification du parser/interpréteur

On ajoute aussi à la calculatrice une instruction d'affectation (syntaxe de votre choix).

Il pourra être utile de trouver un moyen de demander à Haskell de parser une ligne dans son intégralité (on pourrait aussi dire : consommer la totalité du flux d'entrée), afin d'échouer sur des entrées de la forme

```
"(3+4)égalneuf"
```

6 Moins de parenthèses

Il s'agit dans cette section d'introduire de l'associativité et des précédences entre les opérateurs, afin de rendre les parenthèses optionnelles.

Au lieu de faire cela à la main en introduisant une grammaire non ambigüe, servez-vous de la librairie `Text.Parsec.Expr`.

Attention : l'exemple fourni dans la documentation de cette librairie ne fonctionne pas tel quel, il présuppose que certaines fonctions, comme `parens`, `reservedOp`, etc. ont été extraites d'un lexeur généré au préalable (cf. `Text.Parsec.Token`).

À vous de décider si vous voulez suivre cette voie – il existe des exemples très complets d'utilisation que vous pouvez reprendre. Pour une compréhension plus fine, vous pouvez adapter l'exemple de la librairie en programmant vous-même les fonctions manquantes. Par exemple, `reservedOp` peut être remplacé `string`, au prix d'une spécialisation du type des fonctions `binary, prefix, postfix`. Il faut aussi garder en tête que `reservedOp`, quand il échoue, ne consomme pas le flux, contrairement à `string`, et qu'il faudra donc simuler ce comportement.

Dans les deux cas, prenez soin de faire générer à votre parseur des expressions de type `AExpr`: l'exemple donné dans la documentation de `Text.Parsec.Expr`, lui, les évalue directement.

7 Aller encore plus loin

On souhaite maintenant avoir des expressions booléennes et des nombres flottants. On pourra prendre exemple sur la syntaxe de OCaml, pour éviter d'avoir des opérateurs et des constantes ambigües (`+.` est l'opérateur d'addition sur les flottants, par exemple).

Vous pourrez ensuite commencer une analyse de typage sur l'arbre d'expressions, notamment au niveau des variables. Par exemple

```
x = true
3*x
```

devrait produire une erreur de typage autre que celle générée par Haskell lorsque l'on tente d'exécuter l'opération. À vous de décider si vous souhaitez que le code suivant

```
x= true
x = 3.14
```

fasse une erreur de typage ou non.

Pour avoir une calculatrice opérationnelle, rajoutez des commandes pour vider la mémoire (syntaxe ?), des constantes textuelles (mots réservés) telles que `pi`, `e`, un opérateur de puissance, des fonctions arithmétiques...