Analyse syntaxique par combinateur

Martin Odersky

14 novembre 2006
version 1.1

Plan du cours

Deux approches de l'analyse syntaxique

Jusqu'à présent, nous avons vu deux méthodes pour construire un analyseur syntaxique :

manuelle en dérivant l'analyseur de la grammaire ;

automatique en utilisant un outil qui transforme la grammaire en un analyseur.

La seconde approche utilise un langage de description de grammaire pour décrire la grammaire d'entrée.

Langages spécifiques au domaine d'application

Le langage de description de grammaire est un exemple de langages spécifiques au domaine d'application (LSD, *domain-specific language*).

Le générateur d'analyseur syntaxique est un processeur (ou compilateur) pour ce langage — c'est pourquoi on l'appelle, un peu pompeusement, un «compilateur de compilateurs».

Exemple α : Un LSD de description des grammaires

```
Expr   ::= Term { '+' Term | '-' Term }.
Term   ::= Factor { '*' Factor | '/' Factor }.
Factor ::= Number | '(' Expr ')'.
```

LSDs accueillis

Une alternative au LSD indépendant est le LSD accueilli (*hosted language*) :

- il n'est plus un langage à part entière mais une librairie dans un langage hôte ;
- le langage hôte est en général un langage généraliste (*general purpose programming language*).

Nous allons maintenant développer cette approche pour un langage de description de grammaire.

Scala accueille un langage de description de grammaire

Nous souhaitons créer une librairie qui permet de décrire une grammaire de la façon suivante :

Exemple β : Un LSD accueilli de description des grammaires

```
object ExprParsers {
  def Expr: Parser =
    Term & rep{ token('+') & Term | token('-') & Term }
  def Term: Parser =
    Factor & rep{ token('*') & Factor | token('/') & Factor }
  def Factor: Parser =
    Number | token('(') & Expr & token(')')
}
```

Il est possible d'obtenir la grammaire de l'exemple β depuis celle de l'exemple α par des règles de remplacement systématique :

- Ajouter un `def` avant chaque production
- Remplacer le `::=` par un `: Parser =`
- La composition séquentielle est exprimée par `&`
- La répétition `{ ... }` est exprimée par `rep{ ... }`
- L'option `[ ... ]` est exprimée par `opt{ ... }`
- Les symboles terminaux sont encodés dans un `token( ... )`
- Les productions sont terminées par un point-virgule

Combinateurs d'analyseurs syntaxiques

Les différences entre la grammaire de l'exemple α et celle de l'exemple β sont minimes.

- En particulier si l'on considère qu'un vrai langage de description de grammaire ajoute des ornements syntaxiques (*syntactic sugar*) à la grammaire idéalisée de l'exemple α.

La vraie différence est que la grammaire de l'exemple β est un programme Scala valide...

- à condition qu'une librairie définisse les primitives nécessaires.

Ces primitives sont appelées combinateurs en général et plus spécifiquement ici combinateurs d'analyseurs syntaxiques (C.A.S.).

Combinateurs : idée de base

Définition : Combinateur

Pour chaque langage (défini par un symbole S de sa grammaire), on définit une fonction f_S qui, pour un flot d'entrée `in` :

- si un préfixe de `in` est dans S, retourne `Some(Pair(x, ins))` où `x` est un résultat pour S et `ins` est le reste de l'entrée ;
- sinon, retourne `None`.

On appelle le premier comportement succès et le second échec.

En pratique, l'*idée de base* peut être exprimée en Scala de la manière suivante :

```
type Parser = Input => Option[Pair[Any, Input]];
```

```
type Input = List[Token];
```

ou alternativement :

```
type Input = Stream[Token];
```

Un lexème étant de la forme :

```
class Token { val tclass: Int; val chars: String }
```

Combinateurs OO

Plus concrètement, nous devons exprimer les opérateurs `|` et `&` comme des méthodes de l'analyseur.

C'est pourquoi nous étendons le type fonctionnel de `Parser` de la façon suivante :

```
type Input = List[Token]
type Result = Option[Pair[Any, Input]]
abstract class Parser extends (Input => Result) {
  def apply(in: Input): Result
  def & ...
  def | ...
}
```

A partir de là, il ne reste plus qu'à définir des combinateurs concrets qui implantent cette classe.

Le combinateur «classe de lexèmes»

Le C.A.S. `token(tclass)` produit :

- un succès si le premier lexème de l'entrée est de la classe de lexème `tclass` ;
- une erreur autrement.

En cas de succès, la chaîne de caractères attachée au lexème est retournée.

```
def token(tclass: Int) = new Parser {
  def apply(in: Input): Result =
    if (!in.isEmpty && in.head.tclass == tclass)
      Some(Pair(in.head.chars, in.tail))
    else None
}
```

Le combinateur «séquence»

Le C.A.S. `P & Q` produit :

- un succès si `P`, puis `Q` réussissent ;
- une erreur autrement.

En cas de succès, une paire est retournée dont le premier élément est le résultat de `P` et le deuxième élément est le résultat de `Q`.
Ce combinateur est implanté comme une méthode de `Parser`.

```
abstract class Parser {
  def & (q: => Parser) = new Parser {
    def apply(in: Input): Result = Parser.this(in) match {
      case Some(Pair(x, in1)) => q(in1) match {
        case Some(Pair(xs, in2)) =>
          Some(Pair(compose(x, xs), in2))
        case None => None }
      case None => None
}}}
```


Le combinateur «alternative»

Le C.A.S. `P | Q` produit :

- un succès si un combinateur parmi `P` et `Q` réussit ;
- une erreur autrement.

En cas de succès, le résultat de `P` est retourné si `P` réussit, le résultat de `Q` autrement.
Le combinateur «alternative» est implanté comme une méthode de `Parser`.

```
abstract class Parser {
  def | (q: => Parser) = new Parser {
    def apply(in: Input): Result = Parser.this(in) match {
      case r @ Some(_) => r
      case None => q(in)
}}}
```

Le combinateur «répétition»

Le C.A.S. `rep{ P }` est appliqué zéro ou plusieurs fois jusqu'à ce que `P` retourne un échec.

Le résultat est une concaténation de tous les résultats successifs de `P` ou `null` si `P` est appliqué zéro fois.

```
def rep(p: Parser) = new Parser {
  def apply(in: Input): Result = p(in) match {
    case Some(Pair(x, in1)) =>
      apply(in1) match {
        case Some(Pair(xs, in2)) =>
          Some(Pair(compose(x, xs), in2))
        case None => Some(Pair(List(x), in1))
      }
    case None => Some(Pair(null, in))
  }
}
```

Notez que ce combinateur ne retourne jamais un échec.

Le combinateur «option»

Le C.A.S. `opt{ P }` essaye d'appliquer `P` une seule fois.

Le résultat est `null` ou le résultat de `P` si `P` a réussi.

Exercice

Implantez le combinateur «option» suivant :

```
def opt(p: Parser) = new Parser {
  def apply(in: Input): Result = ...
}
```

Combinateurs spéciaux

Le combinateur `failure` produit toujours un échec.

```
def failure = new Parser {
  def apply(in: Input): Result = None
}
```

Le combinateur `success(x)` réussit toujours, sans consommer l'entrée, en retournant `x`.

```
def success(x: Any) = new Parser {
  def apply(in: Input): Result = Some(Pair(x, in))
}
```

Le combinateur `empty` réussit toujours, sans consommer l'entrée, en retournant `null`.

```
def empty = success(null);
```


Si nécessaire, il est toujours possible d'implanter des C.A.S. supplémentaires.

Exercice

Implantez le C.A.S. `rep1{ P }` qui applique `P` une fois ou plus.

```
def rep1(p: Parser) = new Parser {
  def apply(in: Input): Result = ...
}
```

Exercice

Implantez les C.A.S. `opt{ P }`, `rep{ P }` et `rep1{ P }` en terme de `|`, `&` et `empty`.

Sortie de l'analyseur syntaxique

Les analyseurs syntaxiques par combinateur peuvent retourner des données variées.

Dans notre exemple, l'arbre de syntaxe retourné est une structure formée de chaînes de caractères et de paires. Un résultat est :

- soit une chaîne de caractères représentant un terminal ;
- soit une paire de résultats représentants une composition séquentielle ;
- soit `null` qui représente la chaîne vide.

La fonction `compose` permet de construire la composition séquentielle :

```
def compose(x: Any, y: Any) =
  if (x == null) y
  else if (y == null) x
  else Pair(x, y);
```


Exemple

Dans le cadre de notre grammaire d'exemple, l'entrée `12 * (13 - 7) - 40 + 10` serait représentée de la façon suivante (où les paires sont représentées par `[ , ]`) :

```
[ [ 12,
    [ *,
      [ [ (,
          [ 13 , [ -, 7 ] ] ],
        ) ] ] ],
  [ [ - , 40 ],
    [ + , 10 ]
  ]
]
```

Exécuter l'analyseur syntaxique

Si l'on dispose d'une méthode :

```
def tokenize(in: InputStream): Stream[Token];
```

on peut écrire une classe pilote pour l'analyseur syntaxique par combinateur de notre grammaire d'exemple de la façon suivante :

```
object Main {
  def main(args: Array[String]): Unit = {
    Console.print("> ")
    new ExprParsers.Expr(tokenize(System.in)) match {
      case Some(tree, _) =>
        Console.println("Parsed: " + tree)
      case None =>
        Console.println("Syntax error")
}}}
```


Analyseur ne retournant pas des arbres

Jusqu'à présent, nos analyseurs construisaient une forme d'arbre de syntaxe, l'idée étant que cet arbre serait utilisé plus tard. Mais cela peut devenir assez lent avec des programmes plus grands, les arbres pouvant atteindre des tailles importantes.

- Il est possible de paramétriser l'analyseur avec le résultat.

Exemple : Un analyseur paramétrisé

```
abstract class Parser[T] {
  type Input = Stream[Token]
  type Result = Option[Pair[T, Input]]
  def apply(in: Input): Result
  def & ...
  def | ...
}
```

Améliorer la composition séquentielle

Problème : si l'analyseur retourne des valeurs définies par l'utilisateur, comment les valeurs sont-elles composées ?

Considérons un analyseur qui lit et évalue des expressions en même temps :

- Une grande flexibilité de composition est requise.

Solution : on généralise l'opérateur `&` en une compréhension.

Un analyseur par combinateur qui retourne des résultats numériques pourrait être exprimé comme suit :

```
object ExprParsers {
  def Expr: Parser[Int] =
    for (val x <- Term
         val xs <- rep { token('+') & Term })
    yield (x /: xs) ((x, y) => x + y)
  def Term: Parser[Int] =
    for (val x <- Factor
         val xs <- rep { token('*') & Factor })
    yield (x /: xs) ((x, y) => x * y)
  def Factor: Parser[Int] =
    Number
  | (for (val _ <- token('('); val x <- Expr; val _ <- token(')'))
     yield x)
  def Number: Parser[Int] =
    for (val chars <- token(NUMBER)) yield Integer.parseInt(chars)
}
```

Implanter les compréhensions

Nous avons déjà vu (dans le cours programmation IV) les compréhensions sur les listes, les flots et les bases de données.

Exercice

Comment ces concepts peuvent-ils être définis pour un analyseur ?

Il est utile de se rappeler que les compréhensions peuvent être exprimées comme la composition des trois méthodes standards `map`, `flatMap` et `filter`.

Voici les règles de traduction des compréhensions dans Scala :

Pour `f`, un filtre et `s`, une suite (potentiellement vide) de générateurs et de filtres.

$$\frac{\texttt{for (val x <- e) yield } e_r}{\texttt{e.map(x => } e_r\texttt{)}}$$

$$\frac{\texttt{for (val x <- e; f; s) yield } e_r}{\texttt{for (val x <- e.filter(x => f); s) yield } e_r}$$

$$\frac{\texttt{for (val x <- } e_1\texttt{; val y <- } e_2\texttt{; s) yield } e_r}{e_1\texttt{.flatMap(x => for (val y <- } e_2\texttt{; s) yield } e_r\texttt{)}}$$

Les règles de traduction sont appliquées jusqu'à ce que toutes les compréhensions soient supprimées.

Ces règles ne s'appliquent pas qu'aux listes mais à tout type de générateur qui dispose des méthodes `map`, `flatMap` et `filter`.

L'analyseur par combinateur avec compréhensions

```
abstract class Parser[T] {
  type Input = Stream[Token]
  type Result = Option[Pair[T, Input]]
  def apply(in: Input): Result
  def filter(pred: T => boolean) = new Parser[T] {
    def apply(in: Input): Result = Parser.this.apply(in) match {
      case None => None
      case Some(Pair(x, in1)) =>
        if (pred(x)) Some(Pair(x, in1)) else None
    }}
  def map[S](f: T => S) = new Parser[S] {
    def apply(in: Input): Result = Parser.this.apply(in) match {
      case None => None
      case Some(Pair(x, in1)) => Some(Pair(f(x), in1))
    }}
  def flatMap[S](f: T => Parser[S]) = new Parser[S] {
    def apply(in: Input): Result = Parser.this.apply(in) match {
      case None => None
      case Some(Pair(x, in1)) => f(x).apply(in1)
}}}
```


- La méthode `|` est essentiellement définie comme avant.
- La méthode `&` est définie grâce à une compréhension :

```
def & [S](q: => Parser[S]): Parser[S] =
  for (val _ <- this; val x <- q) yield x;
```

- Les combinateurs `rep` et `opt` retournent eux aussi des valeurs :

```
def rep[T](p: Parser[T]): Parser[List[T]] =
  rep1(p) | success(List())
def rep1[T](p: Parser[T]): Parser[List[T]] =
  for (val x <- p; val xs <- rep(p)) yield x :: xs
def opt[T](p: Parser[T]): Parser[List[T]] =
  (for (val x <- p) yield
    List(x)) | success(List());
```

Conclusion : Avantages et désavantages

L'analyseur syntaxique par combinateur est un exemple intéressant d'un LSD accueilli.

- Il est visuellement très similaire à la grammaire, mais est en fait un programme exécutable.

D'autres variantes sont également possibles : la grammaire pourrait être transformée par un combinateur en un AST qui est ensuite transformé en un analyseur syntaxique LR(1).

- Il serait ainsi possible d'utiliser la même grammaire pour le prototype et la version de production, simplement en changeant les librairies qui traitent cette grammaire.

Avantages de l'analyseur syntaxique par combinateur :

- L'analyseur peut être défini dans une notation de haut niveau proche d'une grammaire indépendante du contexte.
- Nous n'avons pas défini un seul analyseur mais une multitude d'analyseurs qui peuvent être réutilisés séparément.
- Bien pour prototyper de petits langages avec des tailles d'entrée limitées.
- Pratique pour rapidement mettre en oeuvre un analyseur simple.

Désavantages de l'analyseur syntaxique par combinateur :

- Lorsqu'un analyseur par combinateur est exécuté tel quel, la performance est médiocre.
- Le traitement des erreurs (présentation et reprise) est mauvais. (ces deux problèmes peuvent toutefois être contournés lorsque les C.A.S. sont utilisés comme entrées pour des générateurs d'analyseur.)
- La concision est moins bonne lorsque l'on varie les résultats en utilisant les compréhensions.

Finalement, le langage hôte doit pouvoir exprimer l'analyseur par combinateur de manière concise :

- C'est le cas avec Scala ou Haskell.
- C'est plus compliqué avec d'autres langages qui ne supportent pas les compréhensions et l'évaluation paresseuse.
- Cependant, les langages modernes évoluent dans cette direction :
  - prenez pour exemple LINQ pour C# 3.0 qui définit des compréhensions tout à fait dans le style de Scala.