

Les analyseurs syntaxiques fondés sur la combinaison d'analyseurs

Stéphane Bortzmeyer
<stephane+blog@bortzmeyer.org>

Première rédaction de cet article le 31 Août 2006. Dernière mise à jour le 14 Novembre 2006

<http://www.bortzmeyer.org/combinatorial-parser.html>

Il est fréquent qu'un programme informatique aie besoin de procéder à une analyse syntaxique d'un document. Cette tâche est en général effectuée par des programmes générés à partir d'une description de la grammaire, description effectuée dans un langage spécialisé. Yacc est le plus courant de ces générateurs d'analyseurs syntaxiques. Mais une autre approche est la combinaison d'analyseurs.

Par exemple, si je veux extraire du registre IANA des langues <<http://www.iana.org/assignments/language-subtag-registry>> (décrit dans le RFC 4646¹) toutes les langues qui s'écrivent couramment avec l'alphabet arabe (ce sont les langages pour lesquels le champ `Suppress-Script` vaut `Arab`), je dois **analyser** ce fichier, écrit avec la syntaxe dite "*record-jar*". Je peux écrire sa grammaire (qui est triviale) en Yacc mais Yacc, comme la plupart de ses concurrents (comme Yappy <<http://www.ncc.up.pt/fado/Yappy/>> pour Python) oblige à apprendre un deuxième langage, distinct du langage hôte (C pour Yacc) et ne permet pas d'utiliser les constructions du langage hôte dans la grammaire elle-même, ce qui limite les possibilités de cette grammaire (notamment pour les langages sensibles au contexte).

Une autre solution est d'utiliser le langage hôte pour décrire la grammaire, en décrivant de nombreux petit analyseurs syntaxiques, combinés avec une bibliothèque de fonctions de combinaison, qui mettent en œuvre les opérations de combinaison classiques (comme la répétition, l'optionnel, etc). C'est ce que permet par exemple la bibliothèque de combinateurs Parsec <<http://www.cs.uu.nl/~daan/parsec.html>> pour le langage Haskell.

Comme premier exemple, essayons d'analyser le fichier `/etc/passwd` d'une machine Unix (pour cet exemple particulier, où le fichier a une structure très simple, le module `Text:Regex` serait une autre solution, mais Parsec devient nécessaire si la structure du fichier est plus complexe). La grammaire est :

1. Pour voir le RFC de numéro NNN, <http://www.ietf.org/rfc/rfcNNN.txt>, par exemple <http://www.ietf.org/rfc/rfc4646.txt>

```
field = many (noneOf ":")
line = field `sepBy` char ':'
```

Ce code est du Haskell pur, utilisant les fonctions de la bibliothèque Parsec (comme `many`) ou bien ses définitions déjà faites (comme `newline`).

Une fois la grammaire écrite, on peut la tester facilement avec la fonction `parseTest` fournie par Parsec. Puis l'utiliser dans un vrai programme avec la fonction `parse` de Parsec. Voici un test avec `ghci` (partie de `ghc`) :

```
*Main> parseTest line "a:b:ct:e"
["a","b","ct","e"]
*Main> parseTest line "az"
["az"]
*Main> parseTest line "az:"
["az",""]
```

Le programme complet pourrait ressembler à :

```
field = many (noneOf ":")
line = field `sepBy` char ':'
alllines = line `sepBy` newline
display (Right contents) = concat (map unwords contents)
main = do
  contents <- readFile "/etc/passwd"
  let result = parse alllines "/etc/passwd" contents
      putStr (display result)
```

Voici maintenant une partie de la grammaire qui permet d'analyser le registre des langues :

```
doublepercent = do {string "%%"; newline} <?> "double percent line"
record = do
  fields <- many1 field <?> "record"
  let therecord = fromList (map extract fields)
      return (recordFactory therecord)
field = do
  name <- fieldname
  spaces
  colon
  spaces
  value <- fieldvalue
  newline
  return (Field name value)
<?> "field"
registry = do
  allrecords <- record `sepBy` doublepercent
  eof
  return allrecords
```

La fonction `fromList` vient du module `Data.Map` et sert à transformer la liste renvoyée par `parse` en un tableau indexé.

Parsec est **prédicatif** par défaut, ce qui veut dire qu'il ne garde pas assez d'état pour revenir en arrière s'il se trompe. Avec certaines grammaires (par exemple, si "x" est défini comme "a c" ou "a b"), Parsec aura consommé l'entrée (ici "a") avant de détecter l'erreur et ne pourra pas revenir en arrière. C'est la source de la plupart des bogues Parsec et c'est un grand sujet d'énerverment pour le programmeur, qui doit penser à insérer des `try` (qui indique à Parsec de garder assez d'état pour pouvoir reculer) judicieusement.

Plusieurs de mes programmes sont écrits avec Parsec (un exemple ultra-simple <http://www.bortzmeyer.org/simple-compiler-in-haskell.html>, conçu à des fins pédagogiques) est aussi disponible) :

- Shadok <http://www.cosmogol.fr/shadok.html>, un programme de traitement des automates à états finis écrits dans le langage Cosmogol <http://www.cosmogol.fr/>.
- GaBuZoMeu <http://www.bortzmeyer.org/gabuzomeu-parsing-language-tags.html>, un programme d'analyse des "language tags" du RFC 4646.
- Eustathius <http://www.bortzmeyer.org/eustathius-test-grammars.html>, un programme de génération de jeux de tests à partir de grammaires ABNF du RFC 5234.

Un très bon article d'introduction sur ces analyseurs à combinateurs est "*Monadic parser combinators*" <http://www.cs.nott.ac.uk/~gmh/bib.html#monparsing> qui décrit très en détail ces analyseurs. Les exemples sont dans un langage très proche d'Haskell, Gofer. (La partie sur les monades n'est pas indispensable.)

Des analyseurs bâtis sur le même principe existent pour d'autres langages de programmation comme `rparsec` <http://rubyforge.org/projects/rparsec/> pour Ruby, `Jparsec` <http://jparsec.codehaus.org/> pour Java ou bien `Combo` <http://www.math.chalmers.se/~koen/ParserCombo/parser-combo-c.html> en C. En Python, il semble n'y avoir que des expérimentations <http://codepoetics.com/poetix/index.php?p=94>.