https://www.laurentbloch.net/BlogLB/Programmer-des-grammaires-avec

Programmer des grammaires avec Bigloo

- Zinformatiques - Cours de bioinformatique au CNAM -

Date de mise en ligne : mercredi 16 mars 2005

L'analyse de textes par ordinateur pour y reconnaître des motifs est très fastidieuse si l'on emploie des algorithmes naïfs. L'usage des grammaires régulières facilite ce travail. Bigloo dispose d'outils adaptés à cette fin.

L'analyse de textes, au sens large, pour y reconnaître des formes ou des motifs, est un problème de programmation récurrent, spécialement en bioinformatique. Le recours à des algorithmes naïfs débouche sur une programmation particulièrement fastidieuse, de surcroît sujette à des erreurs particulièrement sournoises.

Or il existe une technique éprouvée pour aborder de tels problèmes, ce sont les grammaires régulières, utilisées au premier chef pour la construction de traducteurs de langages informatiques, compilateurs ou interpréteurs. Le compilateur Bigloo pour le langage Scheme met à la disposition du programmeur son propre moteur grammatical, dont vous trouverez la syntaxe en suivant le lien.

Cette page ne propose pas un cours sur les grammaires régulières. Mais je suis tombé (un peu tardivement je le confesse) sur un excellent ouvrage de Jeffrey E. F. Friedl, *Mastering Regular Expressions* (en français Maîtrise des expressions régulières). L'analyse des expressions régulières est un des problèmes que l'on peut résoudre avec les grammaires régulières, parmi bien d'autres. Les expressions régulières passent auprès de beaucoup d'informaticiens (y compris moi avant d'avoir lu Friedl) pour un outil technique de bas niveau, alors qu'il s'agit d'un formalisme puissant, introduit par Warren McCulloch et Walter Pitts dans leur article fondateur du *Bulletin of Math. Biophysics 5 (1943)*, « *A logical calculus of the ideas immanent in nervous activity* », qui a joué un rôle décisif, pour le meilleur et pour le pire, dans la naissance de l'informatique moderne et de l'intelligence artificielle. C'est Ken Thompson qui a publié en 1968 le premier article consacré à l'utilisation concrète des expressions régulières dans une perspective de programmation informatique, en l'occurrence un compilateur d'expressions régulières, précurseur de l'éditeur de texte *ed* sous Unix.

Le livre de Friedl révèle toute la portée intellectuelle des expressions régulières, en même temps qu'il propose aux lecteur des exercices subtils et amusants, qui vont de la conversion de degrés Farenheit en degrés Celsius à la mise en forme de pages de cours de la bourse. Je me suis dit qu'il serait amusant de comparer les langages, et pour cela de résoudre certains exercices du Friedl avec Bigloo.

Il y a deux familles de moteurs de grammaires régulières, ceux qui reposent sur des automates à états finis déterministes (DFA dans la suite), et ceux qui utilisent les automates à états finis non-déterministes (NFA dans la suite). Il n'est pas dans mon propos d'écrire un texte sur la théorie des automates et des langages, mais les liens présents ici vous donneront quelques repères pour savoir de quoi il s'agit.

Le moteur grammatical de Bigloo est de type DFA : les moteurs DFA ont l'avantage de donner des grammaires rapides et surtout qui analysent en temps constant. Leur défaut est que l'analyse ne peut jamais « revenir en arrière », et qu'il est de ce fait impossible de capturer dans une variable un résultat partiel de l'analyse. Mais dans le cas de Bigloo cet inconvénient peut être contourné grâce au fait que nous disposons, pour effectuer des actions en fonction des étapes de l'analyse, d'un langage de programmation *Turing-équivalent*, en l'occurrence Bigloo.

En fait, une grammaire régulière de Bigloo ressemble à un `cond` Scheme, où pour chaque clause la condition du `cond` serait remplacée par une règle de la grammaire, suivie comme dans un `cond` d'une séquence d'expressions Scheme, qui seront évaluées si la règle de la grammaire a reconnu une chaîne de caractères dans le texte analysé. L'action `ignore` permet de continuer l'analyse après une reconnaissance partielle. L'exemple à la fin de cette page devrait

aider à la compréhension. Cette ressemblance n'est que partielle, il y a entre la sémantique du `cond` et celle des grammaires de Bigloo une différence expliquée ci-dessous.

L'exemple qui sert de fil conducteur à toute la première partie du livre de Friedl consiste à résoudre le problème suivant : il s'agit de relire un texte électronique, ensemble de pages WWW ou manuscrit en LaTeX, réparti en un nombre arbitraire de fichiers, pour y détecter les mots redoublés par erreur, comme si j'écrivais « comme comme » par exemple. Tout auteur de textes sait que ce type d'erreur est aussi fréquent que difficile à voir à l'œil nu. Voici l'énoncé du problème de Friedl :

– analyser des fichiers en nombre quelconque ; afficher, pour chaque fichier, chaque ligne qui comporte des mots redoublés en mettant en valeur le mot en question dans la ligne ; il faut aussi afficher le nom du fichier concerné et le rang de la ligne dans le fichier ;

– les mots redoublés devront être détectés même s'ils diffèrent par la casse (comme dans « Ainsi ainsi ») ou s'ils sont séparés par un nombre arbitraire d'espaces, de caractères de tabulation, de sauts de ligne, etc.

– la détection devra également se jouer des balises HTML, comme ici : « `très` très ».

La question est loin d'être simple ! Friedl donne une solution en Perl, puis en Java. Vous trouverez ci-dessous une solution partielle avec Bigloo, que je vous invite à compléter (en vous aidant de la documentation du moteur grammatical). Voici le programme complet, le commentaire des passages grammaticaux est en-dessous :

```
 (module detect-the-doubles
   (main start))

(define *THE-PORT* #unspecified)
(define *LINE-NUMBER* 1)

(define (start args)
   (let loop ((file-list (cdr args)))
      (if (not (null? file-list))
   (let ((the-file (car file-list)))
      (set! *LINE-NUMBER* 1)
      (print the-file " : ")
      (set! *THE-PORT*
     (open-input-file the-file))
      (the-reader)
      (close-input-port *THE-PORT*)
      (loop (cdr file-list))))))

(define (the-reader)
   (read/rp the-grammar *THE-PORT*))

(define the-grammar
   (let ((previous-word "")
  (last-word ""))
      (regular-grammar ()
  ((: #\newline)
   (set! *LINE-NUMBER* (+ *LINE-NUMBER* 1))
   (ignore))
  ((+ (in #\space #\tab #\ #\" #\' #\/ #\* #`
   "&:.,;!?+=-_|%$@^#()<>[]{}«»" digit
   #a160))
   (ignore))
  ((+ (in alpha "éèàâùêîïôüûëçÉÈÀÂÙÊÎÏÔÜÛËÇ"))
   (set! previous-word last-word)
   (set! last-word (the-string))
   (if (string-ci=? previous-word last-word)
       (print "  line n° " *LINE-NUMBER*
       " : " previous-word " "
       last-word ))
   (ignore))
  (else
   (let ((this-char (the-failure)))
      (if (not (eof-object? this-char))
   (begin
      (print #\newline "----")
      (display* "Ligne " *LINE-NUMBER*)
      (print " the wrong character : "
      this-char " its ASCII code : "
      (char->integer this-char))))))
  )))
```

- La grammaire régulière proprement-dite commence à la ligne :

```
(regular-grammar ()
```

Comme je l'ai indiqué plus haut, le traitement d'une telle expression est analogue à celui d'un `cond`, avec une différence notable :

-
  - pour le `cond` l'évaluateur examine dans l'ordre les conditions qui figurent en tête de toutes les clauses, et il s'arrête à la première clause dont la condition s'évalue en donnant comme valeur *vrai* (`#t`) ; il évalue alors, dans l'ordre, les autres expressions contenues dans cette clause, et n'examine pas les clauses suivantes ;

-
  - pour une grammaire régulière, l'évaluateur examine toutes les règles, sélectionne celle qui permet de reconnaître la chaîne de caractères la plus longue, et évalue alors les expressions associées à cette règle.

- La première règle de notre grammaire est celle-ci :

```
((: #\newline)
       (set! *LINE-NUMBER* (+ *LINE-NUMBER* 1))
       (ignore))
```

La forme `(:` introduit une séquence d'expressions régulières ; le langage reconnu ici est constitué de l'unique caractère `#\newline` ; si `#\newline` est reconnu, l'action effectuée consiste à incrémenter la variable globale `*LINE-NUMBER*`, puis, c'est le sens de `(ignore)`, à continuer l'analyse.

- Voici la seconde règle :

```
((+ (in #\space #\tab #\ #\" #\' #\/ #\* #`
  "&:.,;!?+=-_|%$@^#()<>[]{}«»" digit
  #a160))
  (ignore))
```

Le langage reconnu est constitué d'au moins un caractère non-alphabétique, suivi d'un nombre quelconque de ses semblables. L'action consiste simplement à continuer l'analyse.

- Troisième règle :

```
((+ (in alpha "éèàâùêîïôüûëçÉÈÀÂÙÊÎÏÔÜÛËÇ"))
  (set! previous-word last-word)
  (set! last-word (the-string))
  (if (string-ci=? previous-word last-word)
      (print "  line n° " *LINE-NUMBER*
      " : " previous-word " "
      last-word ))
  (ignore))
```

L'expression régulière `(+ (in alpha "éèàâùêîïôüûëçÉÈÀÂÙÊÎÏÔÜÛËÇ"))` reconnaît la plus longue séquence ininterrompue de caractères alphabétiques, c'est-à-dire quelque-chose que nous pouvons considérer comme un *mot*, séparé du précédent et du suivant par un espace ou un autre caractère analogue. À noter le cas délicat des mots précédés d'un article lié au mot par une apostrophe, qui n'est pas traité parfaitement ici. Les actions effectuées dans ce cas visent à comparer les deux derniers mots rencontrés, à émettre un diagnostic s'ils sont égaux, puis à continuer l'analyse.

- Quatrième règle, cas non prévus :

```
(else
  (let ((this-char (the-failure)))
     (if (not (eof-object? this-char))
  (begin
     (print #\newline "----")
     (display* "Ligne " *LINE-NUMBER*)
     (print " the wrong character : "
     this-char " its ASCII code : "
     (char->integer this-char))))))
```

Si un caractère n'est reconnu par aucune des règles précédentes, il va déclencher l'arrêt de l'analyse : c'est ce cas que traite notre quatrième et dernière règle. L'accès au caractère en question est donné par la forme Bigloo `(the-failure)`. Il y a un cas évident et normal, qui se produit lorsque l'analyse arrive à la fin du fichier : ce cas est détecté par la condition `(eof-object? (the-failure))`. Dans les autres cas, il peut être commode d'afficher le caractère concerné et son code ASCII, à des fins de dépannage.