Cahiers GUTenberg n ° 9 — Juillet 91

Rgc: un générateur d'analyseurs lexicaux efficaces en Scheme

Manuel Serrano[1,2]

1: Laboratoire d'Informatique de l'École Polytechnique,
91128 Palaiseau CEDEX, France
2: INRIA–Rocquencourt,
BP 105, 78153 Le Chesnay CEDEX, France
`serrano@margaux.inria.fr`

Abstract

Cet article présente Rgc, un générateur d'analyseurs lexicaux rapides, développé pour Scheme. Nous ne décrivons pas ici une maquette mais un produit final efficace. Par ses performances, il est en concurrence directe avec le logiciel Flex. Après mesures, il apparaît que Rgc est entre 5 et 10 % plus rapide que Flex et entre 250 et 260 % plus rapide que Lex. Pour obtenir ce niveau de performance, nous avons réalisé un compilateur spécialisé restreint Scheme→C. De plus, puisque Scheme ne possède pas de primitives rapides de lecture il s'est avéré indispensable de programmer les requêtes systèmes et la gestion des tampons en C. Le code est donc composé de 90% de Scheme et 10% de C.

1. Introduction

Les langages de programmation traditionnels ne disposent que de primitives de lecture peu évoluées. Les informations lexicales qu'il est possible de spécifier sont directement liées aux types des données que les langages manipulent. En C, par exemple, il est facile de préciser qu'on souhaite lire des entiers, des flottants ou des chaînes de caractères, mais on ne peut pas exprimer le souhait de lire des nombres donnés sous forme de fraction car la fraction ne correspond à aucun type de C. Pour prendre un autre exemple, il est difficile de lire des textes C avec un programme écrit en Lisp. Pour pallier ce manque, il existe des outils qui génèrent des analyseurs lexicaux. Lex [Lesk 1975] et Flex [Paxon 1990] en font partie. Ils produisent du code source qui doit être compilé avec le reste du programme. Nous adoptons une approche différente en proposant d'intégrer, dans la définition même des langages, des primitives d'analyse lexicale. Nous définissons des instructions de lectures qui prennent en paramètres toutes les informations standard (nom du fichier, taille des tampons de lecture, etc.), mais aussi une grammaire qui indique comment se comporter à la lecture d'un texte. Nous avons réalisé ces extensions pour Scheme et c'est le résultat de cette recherche que nous présentons ici.

Bien que le principe de fonctionnement soit calqué sur celui de Lex, nous rappellerons comment l'analyseur lexical procède. Nous donnerons un exposé non-exhaustif des nouvelles primitives que nous ajoutons à Scheme. Puisque le point crucial est la vitesse d'analyse, nous expliquerons assez en détail comment sont codés les analyseurs. Pour finir, nous conclurons sur les performances du système. Mais avant toutes choses, précisons quelques points de vocabulaire.

2. Vocabulaire

Il n'est donné ici que quelques rapides définitions. En particulier, nous ne redéfinissons pas tous les termes qui seraient indispensables à un discours complètement rigoureux sur les langages rationnels. Nous invitons le lecteur désireux de trouver ces informations à consulter [Autebert 1987]. Définissons donc juste ce dont nous nous servons.

Langages et expressions rationnels : Sans donner ces deux définitions, précisons que nous employons indifféremment ces deux termes. Une expression rationnelle définissant un et un seul langage, il n'y a pas d'ambiguïté et donc on pourrait substituer dans tout le texte "expression rationnelle" par "langage rationnel".

Mot : Un mot est une suite finie de lettres sur un alphabet quelconque. Pour les analyseurs, puisque les caractères séparateurs (blanc, tabulations ...) appartiennent à l'alphabet, un texte (dans le sens français usuel) est un mot.

Facteur gauche : Un facteur gauche d'un mot *m1* est un mot *m2* constitué des *n* premières lettres de *m1*. Souvent facteur gauche sera abrégé en facteur.

Grammaire rationnelle : Suite de couples *(expression rationnelle, expression Scheme)* ou triplet *(contexte, expression rationnelle, expression Scheme)*. Souvent dans la suite de ce texte nous abrégerons grammaire rationnelle en grammaire. L'ordre dans lequel sont donnés les couples ou triplet a de l'importance. On parlera du langage défini "le plus tôt" ou "en premier" dans la grammaire pour désigner le langage appartenant au couple ou triplet le plus près de la tête de liste.

Reconnaître : Reconnaître un mot m parmi une liste de langages L signifie déterminer à quels langages de L m appartient. Puisqu'à chaque expression rationnelle correspond un langage, reconnaître un mot parmi une liste d'expressions a la même signification que ci-dessus. On dira également qu'un langage reconnaît, admet ou identifie un mot s'il lui appartient.

Analyser : Analyser un texte t avec une grammaire G signifie reconnaître le plus long facteur possible de t parmi tous les langages contenus dans G.

Analyseur : Nous appelons analyseur un programme Scheme qui est le résultat d'une compilation de grammaire.

Lecture rationnelle : Lecture effectuée en utilisant une grammaire.

3. Le principe d'analyse

Nous avons choisi de séparer totalement la lecture de l'analyse. Les lecteurs réalisent toutes les opérations d'entrées. Ils adressent au système les requêtes d'ouverture et de consultation de fichiers. Ils gèrent les tampons dans lesquels sont stockés les textes à analyser. Les analyseurs consultent ces tampons, ils les traitent (en reconnaissent certains éléments) et demandent aux lecteurs de poursuivre leurs travaux de lectures. Les lecteurs sont, en quelque sorte, dirigés par les analyseurs.

3.1. La séparation des tâches

Une des principales différences avec Lex est cette séparation entre lecteur et analyseur. Lex construit une procédure (*yylex*), où tous les travaux de lectures et d'analyses sont effectués. On ne peut analyser simultanément plusieurs textes. Les analyseurs de Lex ne sont pas réentrants. Ici, puisque nous séparons lecture et analyse, ce défaut est corrigé. L'analyseur n'a pas besoin de savoir qui lit quoi. Il lui suffit juste de savoir à quel lecteur il doit s'adresser. Un analyseur peut changer de lecteur à sa guise. Cette possibilité est également offerte par Flex.

3.2. Le choix du langage

Puisqu'une grammaire contient plusieurs expressions rationnelles, le problème se pose de savoir laquelle utiliser pour reconnaître les mots. Le principe est le suivant: on cherche, parmi tous les langages de la grammaire ceux qui admettent le plus long facteur gauche. Si on n'a trouvé qu'un seul langage, le choix est fait. Si plusieurs langages reconnaissent le facteur, on ne garde que celui qui est défini en premier dans la grammaire. Une fois que l'analyseur a choisi le langage, il exécute l'expression Scheme correspondante (celle donnée dans le couple ou le triplet où est défini le langage). Si aucun langage n'admet de facteur, on exécute une action par défaut. La reconnaissance des mots se fait au moyen d'automates finis déterministes.

4. Les nouvelles primitives

Nous donnons ici seulement un résumé des primitives que nous ajoutons à Scheme. Chacune d'entre elles se décline en plusieurs variantes. Nous ne donnons qu'un aperçu pour que le lecteur puisse se faire une idée du fonctionnement "en pratique" de notre système.

`(open-input-stream/rp` *[type]*`)`

Cette fonction est la seule qui permet de créer des lecteurs. Le type du lecteur à créer est déterminé par l'argument qui lui est passé. Il existe plusieurs types de lecteurs: des lecteurs qui lisent soit des fichiers soit des flux, des lecteurs qui lisent lignes/lignes ou caractères/caractères, d'autres qui commencent par charger le texte entier en mémoire, etc.

`(close-input-stream/rp` *lecteur*`)`

Si le fichier ouvert par lecteur n'était pas la console, cette macro le ferme, le tampon est détruit. L'objet lecteur est recyclé (il n'a plus le type lecteur).

`(compile-rg` *name env . body*`)`

L'argument *name* est le nom du futur lecteur. *Env* est une liste de liaisons. *Body* est le corps de la grammaire.

`(read/rp` *analyseur lecteur*`)`

Ceci est la primitive de lecture rationnelle, elle réalise une lecture puis une analyse.

4.1. La syntaxe des grammaires

Nous donnons sous forme BNF la syntaxe des grammaires. Les symboles terminaux sont marqués en majuscules. Les crochets désignent un opérateur “0 ou 1 occurrence”. Ainsi <A> ::= [<B>] est équivalent à <A> ::= <B> | <vide>. La syntaxe des grammaires se trouve en la figure 1. Examinons en détail les formes spéciales présentes dans la définition des règles.

Les terminaux : Les symboles terminaux sont des objets Scheme.

Les opérateurs : Les opérateurs permettent la construction des expressions rationnelles. Dans le tableau décrivant la syntaxe de la grammaire ils apparaissent dans la définition de <reg+> (ce sont les termes soulignés placés en première position derrière une parenthèse ouvrante). Parmi eux on distingue deux catégories: les primitives, qui sont les opérateurs essentiels, et les macros qui sont des opérateurs qu'on peut décrire au moyen des primitives. Les primitives sont données dans la figure 2, les macros dans la figure 3.

Les exceptions : Nous appelons exceptions toutes les parties d'une expression qui ne sont pas définies au moyen des opérateurs vus ci-dessus. Intuitivement, les exceptions ne sont pas des conditions “rationnelles” mais plutôt contextuelles qui font qu'une expression d'une grammaire va accepter des mots. Nous avons choisi le nom exception car il fait référence à un mécanisme de détournement du comportement normal. Dans les quatre définitions que nous donnons à présent, nous employons des phrases comme “*reg* si et seulement si la condition C est vérifiée”. Le lecteur doit comprendre que *reg* peut-être utilisée pour reconnaître des mots si et seulement si la condition C est vérifiée. La liste des exceptions est donnée dans la figure 4.

Les SEXPs : Les SEXPs sont des expressions Scheme. L'utilisateur peut naturellement y utiliser ses propres fonctions (définies au *top-level*) et toutes les primitives du Scheme utilisé. On propose, en supplément, un certain nombre de fonctions qui permettent des traitements sur les

```
<env>        ::=  '(' [ <bindings> ] ')'
<bindings>   ::=  <bindings> <bindings> | '(' <var> <reg+> ')'
<var>        ::=  SYMBOL
<body>       ::=  <rexp+> [ <else-exp> ]
<rexp+>      ::=  <rexp> [ <rexp+> ]
<else-exp>   ::=  '(' ELSE SEXP ')'
<rexp>       ::=  '(' [ <context> ] reg+ SEXP ')'
<context>    ::=  INTEGER
<reg+>       ::=  CHAR
              |   STRING
              |   (ALL)
              |   <var>
              |   '(' uncase STRING ')'
              |   '(' ... INTEGER STRING ')'
              |   '(' ... INTEGER '(' uncase STRING ')' ')'
              |   '(' ! <reg+> ')'
              |   '(' * <reg> ')'
              |   '(' * INTEGER <reg> ')'
              |   '(' + <reg> ')'
              |   '(' + INTEGER <reg> ')'
              |   '(' ? <reg> ')'
              |   '(' >-< <interval> ')'
              |   '(' <-> <interval> ')'
              |   '(' <-> CHAR ')'
              |   '(' in CHAR * ')'
              |   '(' out CHAR * ')'
              |   '(' BOL <reg> ')'
              |   '(' EOL <reg> ')'
              |   '(' EOF <reg> ')'
              |   '(' <reg> ')'
<interval>   ::=  CHAR CHAR [ <interval> ]
```

Fig. 1 - . La définition des grammaires

*	Opérateur de Kleene.
?	Opérateur "0 ou 1 occurrence".
!	Opérateur union.
	Opérateur concaténation (opérateur implicite)

FIG. 2 - . Les primitives

parties du texte qui sont reconnues.

`(context [new-context])`

Si aucun argument n'est passé à cette fonction, l'analyseur repasse dans un état sans contexte (i.e. toutes les règles préfixées par un contexte ne peuvent plus reconnaître de texte). Si un argument est passé, l'analyseur passe dans le contexte correspondant à l'argument. Toutes les règles préfixées par le même contexte que l'argument ainsi que celles qui n'en possèdent pas peuvent servir à la reconnaissance de mot.

`(the-length)`

Retourne la taille de la dernière chaîne reconnue.

`(the-string)`

Retourne une copie de la dernière chaîne reconnue.

`(unget-char)`

Permet au lecteur de revenir un caractère en arrière.

`(first-unmatched-char)`

Retourne le prochain caractère du tampon. En particulier si l'analyseur échoue sur un texte, cette fonction permet d'en connaître la première lettre.

`(ignore)`

Grâce à cette fonction on peut demander à poursuivre l'analyse juste après avoir reconnu une chaîne.

+	(+ e) → (e (* e))
	Au moins une occurrence de e.
+ num	(+ num e) → (e e .. e) num fois
	Num occurrrence de e.
* num	(* num e) → (! (? e) (e e) (e e e) ... (e e ... e))
	0 ou 1 ou num occurrence de e.
in	(in CHAR *) → (! char1 char2 ... charN)
	Tous les caractères compris dans CHAR *.
out	(out CHAR *) → (! char1 char2 ... charN)
	Tous les caractères sauf ceux compris dans CHAR *.
>-<	(>-< intervalle) → (! char1 char2 ... charN)
	Tous les caractères compris dans l'intervalle.
<->	(<-> intervalle) → (! char1 char2 ... charN)
	Tous sauf ceux compris dans l'intervalle.
	(<-> char) → (! char1 char2 ... charN)
	Tous les caractères sauf char.
uncase	(uncase "string")
	→
	((! s S)
	(! t T)
	(! r R)
	(! i I)
	(! n N)
	(! g G))
	Une chaîne indiférement majuscule/minuscule.
...	(... 3 "string") → (! "str" "stri" "strin" "string")
	(... 3 (uncase "string"))
	→
	(! (uncase "str")
	(uncase "stri")
	(uncase "strin")
	(uncase "string"))
	Les n ou (n+1) ou (n+2) ou .. m premières lettres d'une chaîne.
all	(out #Newline)

FIG. 3 - . Les macros

contexte	Un contexte est un nombre. Les règles précédées d'un contexte ne peuvent être utilisées pour connaître un texte que si l'analyseur a été placé préalablement dans ce contexte. Les contextes sont d'une utilisation assez complexe. Il est donné en annexe un exemple de grammaire qui les utilisent.
eol	(eol reg) "end of line". Reg si et seulement si elle est placée en fin de ligne.
eof	(eof reg) "end of file". Reg si et seulement si elle est en fin de fichier.
bol	(bol reg) "beginning of line". Reg si et seulement si elle est placée en début de ligne.

Fig. 4 - . Les exceptions

Les actions sémantiques par défaut : Tout analyseur possède deux actions sémantiques qui correspondent aux cas echec et fin de fichier. Par défaut la première de ces fonctions est

```
(lambda () (first-unmatched-char))
```

la seconde,

```
(lambda () 'eof)
```

L'utilisateur peut changer ces fonctions. Pour la première, il lui suffit de spécifier une *else-exp* (voir grammaire ci-dessus). Pour la seconde, il lui faut placer des *exceptions eof* dans sa grammaire.

Pour clarifier cet exposé donnons en exemple une grammaire *compteuse de mots*. Elle accepte tous les caractères d'un texte. Elle compte le nombre de mots, de lignes et de caractères.

```
(compile-rg wc ()
   ((+ #\Newline)
    (set! char (+ char (the-length)))
```

```
     (set! line (+ line (the-length)))
     (ignore))
    ((+ (in #\space #\tab #\- #\.))
     (set! char (+ char (the-length)))
     (ignore))
    ((+ (out #\Newline #\space #\tab #\. #\-))
     (set! char (+ char (the-length)))
     (set! word (+ 1 word))
     (ignore)))
```

5. La compilation des grammaires rationnelles

Partant d'une grammaire rationnelle, on calcule un automate fini déterministe. Ce dernier est ensuite traduit en Scheme. Nous ne parlons pas du premier calcul car l'algorithme utilisé est décrit dans [Aho, Sethi & Ullman 1986]. La traduction *grammaire →procédure Scheme* est effectuée lors de l'évaluation d'une forme *(compile-rg ...)*. Pour simplifier ce texte nous nommerons le compilateur Rgc.

5.1. Notes techniques

L'analyseur est divisé en deux procédures Scheme. La première est une fermeture où sont closes les actions sémantiques et plusieurs variables utiles à l'interfaçage avec les lecteurs. La deuxième est juste le codage de l'automate. Cette *lambda* ne prend aucun argument et retourne un numéro qui correspond à une *sexp* à évaluer. Le choix de ne pas inclure l'automate dans la procédure globale d'analyse peut sembler curieux et même coûteux. Il a été motivé par le fait que comme nous n'étions pas satisfait de la rapidité obtenue avec Scheme→C [Bartlett 1988], nous souhaitions pouvoir compiler l'automate indépendamment du reste (et surtout des actions sémantiques). Ainsi isolé, son code est extrêmement simple (les variables ne peuvent contenir que des entiers, les procédures ont toutes un seul argument et sont toutes récursives terminales). Nous avons alors pu réaliser un traducteur spécifique très facilement. L'automate global associé à une grammaire est codé par une fonction Scheme. A l'intérieur de cette *lambda* chaque état de l'automate est lui aussi codé par une procédure. Un état peut-être compacté ou non.

5.2. Une illustration

Pour l'exemple, compilons une grammaire très simple.

```
(compile-rg ex ((lettre  (>-< #\a #\z))
                (chiffre (>-< #\0 #\9)))
      ((lettre (* (! lettre chiffre))) 'action-numero-1)
      ((#\; (* (all))) 'action-numero-2))
```

Cette grammaire reconnaît les identificateurs ainsi que les lignes de commentaire Scheme. Elle n'a pas d'intérêt en soit mais elle est suffisamment complexe pour être représentative de ce qu'est capable de générer le compilateur. Nous donnons juste le code de la deuxième *lambda* (le codage de l'automate).

```
(define ex-states
   (lambda ()
      (define (state0 mr)
         (let ( (c (next-ref)) )
            (cond
               ((= c 59)
                (begin
                   (accept-buffer-ref)
                   (state1 2)))
               ((and (>= c 97) (<= c 122))
                (begin
                   (accept-buffer-ref)
                   (state2 1)))
               ((= c 0)
                ;; 0 est un marqueur. Il repère une fin de
                ;; tampon. A ce moment, on réinvoque le lecteur.
                ;; Si rien ne peut plus être lu (!= NOK.OR.MR 0)
                ;; on stop le calcul.
                (let ( (nok.or.mr (eofb mr)) )
                   (if (= nok.or.mr 0)
                       (state0 mr)
                       nok.or.mr)))
               (else
                mr))))
      (define (state1 mr)
         (let ( (c (next-ref)) )
            (cond
               ((= c 10)
                mr)
               ((= c 0)
                (let ( (nok.or.mr (eofb mr)) )
                   (if (= nok.or.mr 0)
                       (state1 mr)
                       nok.or.mr)))
```

```
              (else
               (begin
                  (accept-buffer-ref)
                  (state1 2))))))
      (define (state2 mr)
         (let ( (c (next-ref)) )
            (cond
               ((or (and (>= c 48) (<= c 57))
                    (and (>= c 97) (<= c 122)))
                (begin
                   (accept-buffer-ref)
                   (state2 1)))
               ((= c 0)
                (let ( (nok.or.mr (eofb mr)) )
                   (if (= nok.or.mr 0)
                       (state2 mr)
                       nok.or.mr)))
               (else
                mr))))
      (state0 0)))
```

5.3. Le compactage

Rgc compacte les états de l'automate. Il est possible d'influer sur le degré de compactage en modifiant une variable globale. L'algorithme de traduction des grammaires nous donne comme résultat une liste d'états. Chaque état est lui-même une liste de couples *(lettre, état destination)*. On regroupe ensemble toutes les lettres qui partent vers un même état. On minimise les tests à faire. On associe à chaque test un coût. Le calcul du coût est donnée dans la figure 5.

test	coût
=,<,>	1
or(t1,t2)	1 + coût(t1) + coût(t2)
and(t1, t2)	1 + coût(t1) + coût(t2)
not(t1)	1 + coût(t1)

FIG. 5 - . Le cout des tests

Les tests sont alors classés du plus faible au plus grand coût. Si la somme excède une valeur seuil l'état sera non compacté. Il sera alors codé au moyen de l'instruction Scheme *case*.

On peut remarquer sur l'exemple de la grammaire ex que l'organisation des tests à conduit à deux solutions différentes pour l'état 1 (qui correspond à la reconnaissance des mots appartenant à *L=(* (not #Newline))* et l'état 2. Les *else* ne testent pas la même chose. Dans le deuxième cas seulement il s'agit d'un échec de l'analyseur. Les différentes étapes du calcul pour le premier état sont:

```
;; Tous les états sont calculés normalement.
(cond
   ((or (= c 1) .. (= c 9) (= c 11) ...)
    (state1 2))
   ((= c 0) ...)
   (else  ;; Bien qu'on utilise le mot clé else, on a calculé à
          ;; quel ensemble de lettres correspond cette dernière
          ;; clause (quelles sont les lettres qui y conduisent).
          ;; Dans le cas présent le else correspond à (= c 10).
    mr))

;; On change le or en not.
(cond
   ((not (= c 10))
    (state1 2))
   ....
   (else
    mr) )

;; Puisque le test not est plus cher que le test du else,
;; on les inverse.
(cond
   ((= c 10)
    mr)
    ...
   (else
    (state1 2))))
```

5.4. Les deux compilateurs

Actuellement Rgc est capable de générer du code pour deux compilateurs, Scheme→C et Z→C. Z→C est le compilateur très simple qui traduit le sous-ensemble de Scheme utilisé pour coder les automates en C. Nous l'avons développé car nous n'étions pas toujours satisfait des performances obtenues en utilisant Scheme→C. Z→C est totalement interfaçable avec Scheme→C. Ainsi, dans la version la plus performante, les automates sont compilés avec Z→C et le reste avec Scheme→C.

6. Les performances

Voici quelques temps d'analyse. La première série correspond à une grammaire qui reconnaît tout Scheme (les mots clé, les différents types de nombre, etc.). La deuxième correspond à la grammaire *wc* donnée précédemment en exemple. Les temps de Lex sont pris comme temps références. Les temps *s2c* sont obtenus en générant du code pour Scheme→C. Les temps *z2c* ont été mesurés en compilant les automates avec Z→C. Les temps de Lex et Flex ont été relevés en utilisant ces derniers sans aucune option. Nous avons utilisé la version 2.3 de Flex. Tous ces temps ont été obtenus sur un SUN3/280 avec 8 Mo de mémoire.

Scheme's parser			
lex	2.65 s	32590 c/s	100 %
flex	1.15fs	75100 c/s	230 %
s2c	3 s	28788 c/s	89 %
z2c	1.01 s	85000 c/s	260 %

Word counter			
lex	3.01 s	28630 c/s	100 %
flex	1.36 s	63190 c/s	220 %
s2c	3.46 s	24910 c/s	87 %
z2c	1.21 s	70985 c/s	250 %

Les temps de la grammaire *wc* sont plus faibles que ceux de la grammaire *scheme* alors que les analyses ont été faites sur le même fichier (un fichier de 86000 caractères contenant du Scheme). Ceci s'explique principalement par le fait que les actions sémantiques y sont plus grosses (plusieurs *set!* par action). Il est intéressant de noter que les performances sont inférieures pour tous les analyseurs mais que les écarts entre les analyseurs restent à peu près identiques. Les légères variations sont probablement dues aux meilleurs codages de C pour les incrémentations (un ++ au lieu d'un lourd *set!*).

7. Conclusion

L'objectif initial était d'atteindre des vitesses d'analyses proches de celles de Lex. Ceci ne semblait pas forcément évident à réaliser en Scheme. Comme le montrent les performances, ce but a été en fait largement dépassé. Il reste cependant des améliorations à apporter:

- Rgc n'est pas parfaitement intégré à Scheme. La primitive de création des grammaires n'est pas totalement satisfaisante. Dans une version future la forme

  (`compile-rg` *name env . body*)

  qui affecte à la variable de nom *name* un analyseur sera remplacée par

  (`regular-grammar` *env . body*)

  qui retournera un analyseur. On pourra, dès lors écrire et compiler des expressions comme

  ```
  (read/rp (regular-grammar ....) lecteur)
  ```

- Dans la version que nous considérons aujourd'hui, les grammaires sont des objets rigides. Elles sont créées par une forme unique et ne peuvent plus être modifiées. On ne peut ni définir une grammaire à partir d'autre grammaires ni les composer. Ce ne sont pas des objets de première classe. Il semble pourtant évident que des opérations comme la composition apporte une plus grande souplesse d'emploi. On pourrait souhaiter, par exemple, définir une grammaire qui reconnaîtrait les nombres, une autre qui reconnaîtrait les identificateurs puis les réunir pour former une troisième grammaire. Les définitions des grammaires seraient alors formées d'opérateurs manipulant non seulement des expressions rationnelles, mais aussi des grammaires. Dans l'état actuel de notre travail, la seule façon d'y parvenir serait de recompiler totalement les grammaires à composer. Cette ope'ration n'est pas irréalisable car la compilation est relativement rapide, mais ce n'est pas entièrement satisfaisant car cela oblige en fait à dupliquer le code des automates résultats. Nous voudrions une solution plus évoluée ou se seraient vraiment les résultats de compilation et non les sources qui seraient composés. Cette solution semble toutefois difficile à réaliser si l'on veut conserver des vitesses d'analyse élevées.

Nous voulons toutefois croire que Rgc présente, dès aujourd'hui, une alternative valable à Lex. Pour le prouver, nous avons écrit une boucle fondamentale d'interaction avec notre lecteur. Cette boucle offre quelques particularités amusantes, telles que l'expansion des mots clés, un *message d'invite* qui affiche le nombre de parenthèses ouvertes mais non closes, et d'autres petits gadgets du même ordre. Une partie de la grammaire ainsi qu'une petite session d'utilisation sont données en annnexe.

Références bibliographiques

[Aho, Sethi & Ullman 1986] A. V. Aho, R. Sethi and J. D. Ullman. *Compiler: Principles, Techniques and Tools*, Addison Wesley 1986.

[Autebert 1987] J-M. Autebert. *Les Langages Algébriques*, Masson 1987.

[Bartlett 1988] J. Bartlett. "The Scheme To C compiler", Dec Laboratories 1988.

[Kernighan & Ritchie 1998] B. Kernighan and D. Ritchie. *The C Programming Language*. Software series. Prentice Hall, 1988.

[Lesk 1975] M. E. Lesk. "Lex - a lexical analyzer generator", *Computing Science Technical Report 39*, AT&T Bell Laboratories, Murray Hill, N.J.

[Mauny 1989] M. Mauny. "Parser and printers as stream destructors embedded in functional languages". *FPCA 89, Functional Programming and Computer Architecture*, Londres 1989.

[Paxon 1990] V. Paxon. "Flex - fast lexical analyzer generator" Computer Science Departement 4126 Upson Hall, Cornell University. Ithaca, NY 14853-7501 1990

[Rees & Clinger 1986] J. Rees and W.Clinger. "Revised3 report on the algorithmic language scheme". *SIGPLAN Notices, 21(12):37-79*, Dec. 1986.

A. Un exemple d'utilisation

Nous avons programmé une *boucle fondamentale d'interaction* avec une grammaire rationnelle. Nous ne donnons pas la grammaire complète mais juste les fragments qui nous semblent importants. Cet exemple a pour but de montrer qu'il devient très facile de programmer des choses qui, auparavant, étaient laborieuses et complexes. Il illustre aussi la nécessité des contextes pour résoudre certains problèmes d'analyses.

```
(compile-rg p ( - - - )
       ((+ #\tab)
        (ignore))
```

```
((#\Newline)
 (if (= *par-open* 0)
    (let ( (p (open-input-string *cur-exp*)) )
        (print (eval (read p) '()))
        (set! *cur-exp* "")
        (close-input-port p)
        (set! *prompt/rp* ":=> ")
        (display *prompt/rp*) )
    (begin
       (set! *prompt/rp*
           (string-append (number->string *par-open*) "=> "))
       (add-to-exp " ")
       (display *prompt/rp*) ) )
 (ignore))
((#\()
 (set! *par-open* (++ *par-open*))
 (add-to-exp "(")
 (ignore))
- - -
(((... 2 (uncase "define")) #\tab)
 (display (substring "define" (- (the-length) 1) 6))
 (upper-add-to-exp "DEFINE ")
 (context 1)
 (ignore))
- - -
(((... 4 (uncase "defmacro")) #\tab)
- - -
(1 ("fibo" #\tab)
   (display " x) (if (< x 2) 1 (+ (fibo (- x 1))
                                 (fibo (- x 2)))))")
   (add-to-exp
        "fibo x) (if (< x 2) 1 (+ (fibo (- x 1))
                                 (fibo (- x 2)))))")
   (context)
   (set! *par-open* (- *par-open* 2))
   (ignore))
(2 ("fibo" #\tab)
   (display "nacci x) (if (number? x) (fibo x) `(fibo ,x)))")
- - -

(define (repl)
   (print "Welcome in the repl+ (type "end" to end session)")
   (display *prompt/rp*)
   (let ((s (open-input-stream/rp 'snap-shot)))
   (read/rp p s)
   (newline)
   (close-input-stream/rp s)))
```

Les fonctions *add-to-exp* et *upper-add-to-exp* sont définies au *top-level*, **cur-exp** est une variable globale.

Voici alors un exemple de session. Ce que tape l'utilisateur est imprimé en caractère gras. Ce que répond l'ordinateur est imprimé en italique.

$ **repl**
welcome in the repl+ (type "end" to end session)
=> **(def[tab]**
(define **(toto x)**
1=> **x)))))**
TOTO
:=> **(def[tab]**
(define **(fibo[tab]**
(define (fibo x) (if (< x 2) 1 (+ (fibo (- x 1)) (fibo (- x 2)))))
FIBO
:=> **(defm[tab]**
(defmacro **(fibo[tab]**
(defmacro (fibonacci x) (if (number? x) (fibo x) '(fibo ,x)))
FIBONACCI
:=> ...