Projet : un sed contextuel

Tahina Ramananandro
ramanana@clipper.ens.fr
http://www.eleves.ens.fr/~ramanana/enseigne/ifips

IFIPS – TP 4 et suivants
10 avril – 13 mai 2008

Travaillez et coopérez : montrez votre code à vos voisins, demandez-leur de le tester, de trouver des cas qui feraient bugguer le programme obtenu, etc.

Je requiers votre indulgence quant à la qualité du code Java présenté ici : optimisez-le ! En effet, Java n'est pas mon langage de programmation de base.

Présentation

On souhaite réaliser une version *contextuelle* de *sed*.

Étant données trois expressions régulières G, M et D, et un mot r, on veut construire une machine qui, lorsqu'elle reçoit un mot, remplace par r un sous-mot matché par l'expression régulière M seulement lorsque ce sous-mot est encadré par des mots matchés, à gauche par l'expression régulière G, et à droite par D.

Formellement :

- le mot d'entrée lu par la machine s'écrit $g'gmdd'$ avec g, m, d correspondant respectivement aux expressions régulières G, M et D
- la machine produit en sortie le mot $g'grdd'$

En gros, la machine vérifie si le mot d'entrée admet un sous-mot *u'* correspondant à l'expression régulière GMD. Si c'est le cas, alors *u'* est décomposé en trois parties suivant les expressions régulières G, M et D, et la machine remplace la partie du milieu par r. Les parties de gauche et de droite sont en quelque sorte des *contextes*. conditionnant le remplacement de la partie du milieu.

Exemple : si $G = a$, $M = b$, $D = a + c$, et $r = \varepsilon$, alors sur le mot $baa\mathbf{b}cc$, la machine trouve un b encadré à gauche par un a et à droite par un c (qui matche bien l'expression régulière $a + c$), alors elle remplace ce b par le mot vide, on obtient alors en sortie $baacc$.

Visuellement :

$$\begin{array}{cccccc} & a & b & a+c & & \\ & \uparrow & \uparrow & \uparrow & & \\ ba & a & b & c & c & \\ & & \Downarrow & & & \\ ba & a & \varepsilon & c & c & \end{array}$$

La machine $G = a$, $M = b$, $D = a + c$, $r = \varepsilon$ se notera alors :

$$\begin{array}{ccc} [a] & b & [a+c] \\ & \Downarrow & \\ & \varepsilon & \end{array}$$

Une telle machine est appelée *transducteur*.

1 La machine principale

1.1 Ambiguïtés

1.1.1 Selon la position du sous-mot matché par G

Toujours dans le cas $G = a$, $M = b$, $D = a + c$, et $r = \varepsilon$. Considérons le mot $ababcc$. Montrer que la machine a deux possibilités de s'exécuter selon la position du sous-mot matché par l'expression régulière G.

1.1.2 Selon la longueur du sous-mot matché par G

Plaçons-nous maintenant dans le cas où $G = a\,(b^*)$, $M = b$, $D = b$, et $r = \varepsilon$:

$$\begin{array}{ccc} [a\,(b^*)] & b & [b] \\ & \Downarrow & \\ & \varepsilon & \end{array}$$

Considérons le mot *abbb* . Montrer que la machine a deux possibilités de s'exécuter selon la longueur du sous-mot matché par l'expression régulière G.

1.1.3 Selon la longueur du sous-mot matché par M

Exhiber des machines et des mots tels que la machine ait plusieurs possibilités de s'exécuter selon la longueur du sous-mot matché par l'expression régulière M.

Une ambiguïté similaire peut se produire pour D.

1.2 Implémentation

Écrire une procédure `transducteur` telle que si on a :

```
Couple<String, Integer> resultat = transducteur (
  String chaine,
  int debut,
  ExprRat G, ExprRat M, ExprRat D, String r,
  bool plusLongG, bool plusLongM, bool plusLongD
)
```

alors il existe des mots *g'*,*g*,*m*,*d*,*d'* tels qu'on ait :

$$\begin{array}{ccccc} & G & M & D & \\ & \uparrow & \uparrow & \uparrow & \\ g' & g & m & d & d' \\ & & \Downarrow & & \\ g' & g & r & d & d' \end{array}$$

avec :

- `chaine`=*g'gmdd*
- *g'* de longueur `debut` (i.e. la chaîne *g* matchée par G commence à la position `debut` dans la chaîne de départ)
- `resultat.premier()`=*g'grdd*
- *g'gr* de longueur `resultat.second()` (i.e. le mot *dd'*, représentant toute la partie droite inchangée, commence à la position `resultat.second()` dans la chaîne produite)

On se servira :

- de la fonction `passagesFinals` définie dans la classe `Automates`
- des classes `ExprRat` définies dans la classe `Automates`
- de la méthode `substring` (définie dans la classe standard `String`) pour extraire d'une chaîne un sous-mot étant données sa position de départ et éventuellement sa position de fin (la position du caractère qui *suit* le sous-mot).

Les booléens `plusLongG`, `plusLongM` et `plusLongD` permettent de régler la question des ambiguïtés de longueur des sous-mots matchés par G, M, D. Ils sont vrais (resp. faux) si on veut matcher le plus long (resp. le plus court) sous-mot.

Remarque : si par exemple `plusLongG` et `plusLongM` sont vrais, alors on considèrera que c'est la longueur du sous-mot matché par G qui prime.

Enfin, on écrira une fonction qui exécutera une machine sur un mot en minimisant la longueur du mot *g'*.

2 Langage utilisateur : parseur et lexeur

On voudrait que l'utilisateur puisse écrire un script pour utiliser cette machine. Par exemple, l'utilisateur pourrait taper :

`regle [a] b => @ [a+c] mots baabcc, ababcc`

pour demander à exécuter la machine :

$$\begin{array}{ccc} [a] & b & [a+c] \\ & \Downarrow & \\ & \varepsilon & \end{array}$$

d'abord sur le mot *baabcc*, puis sur le mot *ababcc*.

La machine afficherait alors les mots transformés, ou inchangés si aucune correspondance n'était trouvée.

La grammaire de ce langage est donc :

$$\begin{array}{rcl} programme & ::= & REGLE\ [exprRat]\, exprRat \Rightarrow mot\, [exprRat]\ MOTS\, listeMots \\ mot & ::= & | \\ & & | \quad caractere\, mot \\ caractere & ::= & | \quad @ \\ & & | \quad LETTRE \\ listeMots & ::= & | \quad mot \\ & & | \quad mot, listeMots \end{array}$$

où *LETTRE* représente une lettre alphabétique (c'est en fait un symbole terminal portant une valeur caractère), et REGLE et MOTS sont des symboles terminaux représentant respectivement les mots-clés `regle` et `mots`).

Écrire un parseur pour cette grammaire, et un lexeur pour associer les caractères @, [,], la virgule, les mots-clés `regle` et `mots`, et les lettres, à des symboles terminaux.

Pour le point d'entrée de l'application, on s'inspirera de Calc.java.

3 Pour aller plus loin

Tous les raffinements ci-dessous pourront donner lieu à l'extension du parseur-lexeur.

3.1 Paramètres

Étendre la grammaire (et donc le lexeur aussi) pour permettre à l'utilisateur de personnaliser les critères de désambiguïfication.

3.2 Itération

Écrire une fonction qui permette d'itérer l'exécution d'une machine. C'est-à-dire que si on a :

$$\begin{array}{ccccc} & G & M & D & \\ & \uparrow & \uparrow & \uparrow & \\ g' & g & m & d & d' \\ & & \Downarrow & & \\ g' & g & r & d & d' \end{array}$$

alors la machine s'exécute sur le mot restant *dd'* juste après le remplacement (et non seulement *d'* : le contexte droit est aussi repris en compte par la nouvelle itération) tant que cela est possible.

En clair, la machine effectue tous les remplacements possibles de gauche à droite sur le mot d'entrée, en réutilisant le contexte droit de l'itération précédente pour la nouvelle itération.

3.3 Plusieurs machines en compétition

Supposons qu'on ait maintenant plusieurs transducteurs qu'on veuille exécuter sur une même chaîne.

À cause du remplacement, une seule machine peut être exécutée à la fois sur un mot.

Il faut donc lever l'ambiguïté de savoir quelle machine exécuter si plusieurs à la fois en sont capables. C'est le principe de compétition. On dispose de différents critères :

– quelle machine s'exécute « le plus tôt » (i.e. minimisant la longueur de g')
– quelle machine s'exécute « le plus longtemps » (i.e. minimisant la longueur de d')
– quelle machine remplace la plus longue chaîne
– ...

Choisir des critères de désambiguïfication (ils ne seront probablement pas suffisants), et étendre le programme (parseur et lexeur compris) pour permettre à l'utilisateur d'exécuter plusieurs machines en compétition sur un même mot.

Puis on itérera : à chaque nouvelle itération, la question de savoir quelle machine utiliser se posera à nouveau.

3.4 Pour aller encore plus loin

Réitération (i.e. réexécuter les machines sur le mot transformé une fois qu'on est arrivé à la fin du mot de départ), ou que sais-je encore !

Si on a le temps !