Automates et lexeurs

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

IFIPS – TP 2
4 et 5 mars 2008

Il est vivement recommandé de faire tourner « en vrai » les exemples indiqués dans cette fiche ainsi que dans les fiches de TP.

Prenez votre temps. La machine est rapide, ne vous adaptez pas à sa vitesse. N'hésitez pas à me poser des questions.

À l'inverse, vous pouvez même aller encore plus loin que ne le suggère ce TP.

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.

1 `grep` et `sed`

Posez toutes les questions sur ce que vous n'avez pas compris hier.

2 `jflex` : analyse lexicale

2.1 Présentation

`http://jflex.de/manual.html`

jflex est utilisé par les programmeurs pour générer des parseurs pour des langages de programmation ou autres. jflex est la première étape qui permet de détecter les *tokens*, ou lexèmes, élémentaires du langage. Ces lexèmes serviront ensuite pour le parseur proprement dit que l'on écrira en BYacc/J ou CUP.

Usage :

`jflex` *fichier_source.jl*

Le programme lit le *fichier source* décrivant les *règles* de formation des lexèmes, et produit en sortie une *classe* Java destinée à être compilée et intégrée au projet.

Une règle de formation est un couple qui à une expression régulière associe une action. Alors, la classe Java produite par jflex traitera l'entrée standard, vérifiera si une partie du texte d'entrée correspond à l'expression régulière d'une des règles et exécutera l'action associée.

Un fichier source flex se présente en trois parties sous la forme suivante :

préface
%%
préambule
%%
règles

2.1.1 Préface

La préface est un morceau de code Java à inclure au tout début de la classe (typiquement les divers `import`).

2.1.2 Préambule

Le préambule est une suite de déclarations. Elles sont optionnelles.

Options de la classe.

- `%class` *nom*
  Nom de la classe à générer (défaut Ylex)
- `%implements` *nom1 nom2...*
- `%extends` *nom*
- `%public`
- `%final`
- `%abstract`
- `%init{`
  code
  `%init}`
  Code Java à écrire dans le constructeur de la classe
- `%initthrow{`
  exc1 exc2...
  `%initthrow}`
  Exceptions qui pourraient être levées par le constructeur
- `%standalone`
  Crée une classe « principale » avec un point d'entrée main.

Options de lexage :

- `%function` *nom*
  Nom de la fonction de lexage (défaut yylex)
- `%type` *nom*
  Type de sa valeur de retour
- `%eof{`
  code Java
  `%eof}`
  Code exécuté une fois que la fin de fichier est atteinte
- `%line`
  Active le comptage des lignes, qui permet aux actions de savoir à quel numéro de ligne on se trouve
- `%column`
  Comptage des colonnes
- `%{`
  code Java
  `%}`
  Définit un morceau de code Java à ajouter avant le code correspondant au lexeur proprement dit (par exemple, une primitive de calcul)

Alias de motifs :

- *nom* = *motif*
  Définit une sorte d'alias pour le motif, destiné à être réutilisé dans les règles.

2.1.3 Règles

L'ensemble des règles est présenté sous la forme suivante :

règle1
règle2
...

Une règle a pour forme :

motif { *code_Java* }

où le code Java correspond à l'action à effectuer. L'accolade ouvrante doit se situer sur la même ligne que le motif. Le code Java peut inclure les variables suivantes :

- `String yytext()` : chaîne de caractères trouvée respectant le *motif*
- `int yylength()` : sa longueur
- `int yyline` : numéro de ligne (si %line déclaré dans le préambule)
- `int yycolumn` : numéro de ligne (si %column déclaré dans le préambule)
- `void yypushback (int n)` : permet de remettre les n caractères lus dans la file de lecture, pour le choix et l'exécution de la prochaine règle
- `int yylength()` : sa longueur

Il faut une règle pour <<EOF>> : la fin de fichier.

2.1.4 Syntaxe des motifs

Dans l'ordre décroissant de la priorité opératoire

Motif `jflex`	Expression rationnelle
`[abc]`	a+b+c
`[^abc]`	A*\(a+b+c)
`[a-c0-2]`	(a+b+c+0+1+2)
`(`e`)`	e
e`*`	e^*
e`+`	$e(e^*)$
f `?`	$\varepsilon + f$
ef	ef
e`|`f	e+f

Motif `jflex`	Signification
`^`	Début de ligne (impossible dans un alias)
`$`	Fin de ligne (impossible dans un alias)
"*chaine*"	Chaîne de caractères verbatim
{*nom*}	Expression définie par l'alias dans le préambule

La construction de chaînes de caractères remplace l'échappement \ de `grep` et `sed`.

Petit exemple complet : `http://www.eleves.ens.fr/~ramanana/enseigne/ifips/exemple.jl`

2.2 Une calculatrice en notation polonaise inversée (ou postfixe)

Certaines calculatrices utilisaient (et utilisent encore de nos jours) une notation appelée *notation polonaise inversée*. Cette notation permettait d'écrire a+b sous la forme a b +

Exemple : (1 + (2 * 3)) - 4 s'écrivait 1 2 3 * + 4 -

Formellement, elle correspond à la grammaire suivante :

$$operande ::= nombre \mid operande\, operande\, operateur$$

Montrer que cette grammaire est non ambiguë : pour toute expression respectant cette grammaire, il existe une unique façon d'appliquer les règles. (On pourra dire qu'un mot est formé par une liste de nombres et d'opérateurs, et on pourra raisonner par récurrence sur la longueur de cette liste.)

Comment une machine pourrait-elle effectuer simplement un calcul en notation polonaise inversée ?

En s'inspirant de l'exemple fourni, écrire un source jflex implémentant une calculatrice en notation polonaise inversée gérant l'addition, la soustraction, la multiplication. Commencer avec les entiers seulement. Puis, observer ce qu'il se passe si on rajoute les flottants. Enfin, on pourra rajouter des fonctions telles que le cosinus, etc. (cos x s'écrira alors x cos) (on admettra que la grammaire reste non ambiguë tant que les opérateurs binaires ont tous des notations distinctes des opérateurs unaires).

Que se serait-il passé si on avait choisi la notation polonaise « non inversée » (ou préfixe) ?

2.3 Plusieurs contextes

Il est possible d'avoir plusieurs règles pour un même motif, règles que l'on voudrait choisir selon un certain contexte.

Pour cela, il faut déclarer un *contexte* supplémentaire dans le préambule, en y ajoutant la déclaration :

%state *contexte*

où *contexte* est un identificateur quelconque.

Alors, parmi les règles, on bénéficiera d'une construction supplémentaire :

<*contexte*> {
règle1
règle2
...
}

qui permet d'isoler les règles dans le *contexte*.

Pour passer d'un contexte à l'autre, il faut le préciser dans l'action d'une règle, avec l'instruction :

yybegin (*contexte*) ;

Pour revenir au contexte initial (règles non isolées), il faut revenir au contexte spécial appelé INITIAL.

Exemple :

```
%state CONTEXTE

%%

"a" { yybegin (CONTEXTE); }
"b" { System.out.println ("1"); }

<CONTEXTE> {
 "b" { System.out.println ("2");  yybegin (INITIAL); }
}
```

Ce lexeur accepte tous les mots du langage $(b^\star a)^\star$, et affiche un 1 dès qu'il rencontre un b, sauf s'il est suivi d'un a auquel cas il affiche un 2.

Exercice : étendre l'exemple fourni en acceptant les chaînes de caractères entre guillemets ". Ces chaînes devront elles-mêmes accepter les guillemets, échappés par un \. On pourra s'aider d'une variable globale.

3 Automates

Cette section servira pour les TP suivants (avril et mai). On va définir le concept d'automate pour pouvoir le réutiliser avec le compilateur que nous allons écrire dans les dernières séances. Dans la mesure du possible, il faudrait terminer cette partie avant le début de la prochaine séance en avril.

3.1 Définition « en dur »

Un automate n'est pas la donnée de tous ses états, mais seulement de ses états initiaux, chaque état étant livré avec ses transitions vers les états suivants. Pour éviter des problèmes de définition « bouclante », on rend public l'accès à la liste des transitions de chaque état, ce qui permet de créer des états « vides » puis de définir leurs transitions *a posteriori*.

Squelette à compléter :

```
public class Couple<A,B> extends java.lang.Object {
  Couple<A,B> (A, B);
  A premier ();
  B second ();
}

public class Etat extends java.lang.Object {
  public Vector<Couple<Char,Etat>> transitions;
  public Vector<Etat> suivants (char);
  public boolean estFinal;
  Etat ();
}
```

Exemple : quel est le langage reconnu par monAutomate défini ci-dessous ? (dessiner l'automate, puis donner une expression rationnelle)

```
Etat Q0 = new Etat ();
Etat Q1 = new Etat ();
Etat Q2 = new Etat ();
Q2.estFinal = true;
Q0.transitions.addElement (new Couple<char,Etat> ('c', Q2));
Q0.transitions.addElement (new Couple<char,Etat> ('b', Q0));
Q0.transitions.addElement (new Couple<char,Etat> ('a', Q1));
Q1.transitions.addElement (new Couple<char,Etat> ('b', Q0));
Q1.transitions.addElement (new Couple<char,Etat> ('b', Q2));
Vector<Etat> monAutomate = new Vector<Etat> ();
monAutomate.addElement (Q0);
```

On veut lancer un automate (donné par la liste de ses états initiaux) sur une chaîne en partant de son i-ème caractère. Pour cela, définir une fonction :

```
public Couple<Etat,int> positionsFinales (String, Vector<Etat>, int);
```

telle qu'un couple (q, j) soit dans la liste `positionsFinales (chaine, automate, i)` si, et seulement si, les deux conditions suivantes sont vérifiées :

– à partir de la position i et d'un état initial de l'automate, on atteigne l'état q
– le prochain caractère à lire est en position j (on n'est pas nécessairement arrivé en fin de la chaîne)
– q est final

Remarque : quelle optimisation d'exécution peut-on utiliser à ce stade ?

Sur l'automate de l'exemple ci-dessus, on voudrait que positionsFinales ("baba", monAutomate, 1) (exécuter l'automate sur la chaîne "baba" à partir du 2ème caractère) retourne la liste [(Q2,3)].

Filtrer alors les résultats de la fonction positionsFinales pour définir la reconnaissance d'une chaîne par un automate (donné par ses états initiaux).

```
public boolean reconnait (Vector<Etat>, String);
```

Remarque : on aurait pu écrire la fonction de reconnaissance directement. Cependant, positionsFinales nous resservira dans un prochain TP.

3.2 Génération à partir d'une expression rationnelle

Une expression rationnelle est soit un caractère, soit une concaténation, soit une somme, soit une étoile. En termes de grammaire, on a :

$$exprat ::= caractere \mid exprat\, exprat \mid exprat + exprat \mid exprat^\star$$

On peut donc définir les classes suivantes (squelette à compléter), notamment les automates correspondant à ces expressions rationnelles (à partir d'un algorithme vu en cours).

```
public abstract class ExpReg extends java.lang.Object {
  String toString ();
  Vector<Etat> automate ();
}
public class ExpRegCaractere extends ExpReg {
  char caractere;
  ExpRegCaractere (char);
}
public class ExprRegConcat extends ExpReg {
  ExpReg E1, E2;
  ExpRegConcat (ExpReg, ExpReg);
}
public class ExpRegSomme extends ExpReg {
...
}
public class ExpRegEtoile extends ExpReg {
...
}
```

Exemple : définir (en utilisant les classes ci-dessus) l'expression rationnelle $c + (b^\star a)^\star\, b$, et observer le comportement de son automate.

On peut optimiser le code en calculant l'automate correspondant dès la construction de l'expression rationnelle. Cependant, avec la définition ci-dessus de Etat, cela ne marchera pas. Pourquoi ? Modifier la définition de Etat ci-dessus pour ce faire.