Automates et lexeurs

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

IFIPS – TP 3 et 4
8 et 10 avril 2008

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.

Préparation

On téléchargera sur mon site web l'archive calc.tar, que l'on extraira via la commande :

`tar xf calc.tar`

Cette archive contient JFlex, CUP ainsi que tous les fichiers exemples de ce TP.

1 `cup` : analyse syntaxique

1.1 Présentation

`http://www2.cs.tum.edu/projects/cup`

CUP est utilisé par les programmeurs pour générer des parseurs pour des langages de programmation ou autres. CUP est la seconde étape, le parsing proprement dit, une fois que les *tokens*, ou lexèmes, ont été détectés par JFlex.

Ceci est l'enchaînement logique. Mais concrètement, lexing et parsing se font en même temps : le parseur demande les tokens au lexeur un par un au fur et à mesure du parsing. Le parsing est donc incrémental.

Usage :

`java -jar java-cup-11a.jar -parser` *Classe_parseur* `-symbols` *Classe_symboles fichier_source.cup*

Le programme lit le *fichier source* décrivant la grammaire, et produit en sortie deux classes Java destinées à être compilées et intégrées au projet :

– *Classe_symboles* : déclaration des symboles terminaux
– *Classe_parseur* : le parseur proprement dit, implémentant la grammaire

Un fichier source CUP se présente sous la forme suivante :

`import` *importation* ;
...
`action code {:` *code Java* `:}`
`init with {:` *code Java* `:}`
liste de symboles
précédences
règles

1.1.1 import

Les divers `import` comme pour n'importe quelle classe. Peut être répété.

1.1.2 action code

(optionnel) Le code Java à ajouter avant le parseur (typiquement, déclaration de variables et méthodes annexes)

1.1.3 init with

(optionnel) Le code exécuté avant que le parsing ne commence.

1.1.4 Liste de symboles

La liste de symboles se présente sous la forme d'une succession de lignes ayant l'une des formes suivantes :

`terminal` *classe identificateur1, identificateur2,... ;*
`non terminal` *classe identificateur1, identificateur2,... ;*

définissant une liste de symboles terminaux ou non. L'indication de *classe* donne le « type de retour » de l'action à effectuer par le symbole non terminal, ou bien le type de la « valeur contenue » dans le symbole terminal.

1.1.5 Précédences

Lorsqu'on écrit une grammaire, elle est le plus souvent ambiguë. Par exemple, si on définit la grammaire :

expr : := entier | *expr* - *expr* | *expr* / *expr*

alors, on ne sait pas comment parser :

– 1 - 2 - 3 se parse-t-il en (1 - 2) - 3 ou bien en 1 - (2 - 3) ? Problème d'associativité
– 1 - 2 / 3 se parse-t-il en (1 - 2) / 3 ou bien en 1 - (2 / 3) ? Problème de priorité opératoire

La déclaration des précédences permet de résoudre ces deux problèmes en même temps.

On déclare l'associativité de chaque opérateur sous la forme d'une ligne :

`precedence left` *identificateur1 identificateur2...*

ou bien

`precedence right` *identificateur1 identificateur2...*

selon que l'on veuille des opérateurs associatifs à gauche ((1 - 2) - 3) ou bien à droite (1 - (2 - 3)).

On peut aussi exiger qu'un symbole ne soit jamais « enchaîné » (i.e. que jamais l'utilisateur ne puisse écrire directement 1 / 2 / 3).

`precedence nonassoc` *identificateur1 identificateur2...*

L'ordre de ces lignes définit la priorité opératoire : les précédences sont déclarées par ordre *croissant* des priorités opératoires.

1.1.6 Règles

La grammaire est définie par une succession de règles, une par symbole non terminal.

Une règle a pour forme :

non_terminal : := *clause1* | *clause2* | ... ;

Une clause a pour forme :

symbole1 symbole2 ... `{:` *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 :

– `RESULT` : le résultat que doit « renvoyer » l'action (ne pas utiliser return !). Souvent utilisé en affectation...

Chaque symbole doit être un non-terminal ou un terminal déclaré dans le source, mais il peut être utilisé ici sous la forme *symbole :variable* afin que la *variable* soit utilisée dans l'action comme variable « calculée » par l'action associée au symbole non-terminal, ou bien « portée » par le symbole terminal. Cette variable a alors le type qui a été déclaré avec le symbole.

C'est la première règle qui est considérée comme point d'entrée du parseur.

Clause et précédence Une clause peut être affectée de la précédence d'un autre symbole terminal que celui qui est effectivement employé par l'utilisateur. Par exemple, si on veut utiliser le même symbole terminal MINUS pour la soustraction et pour la « négation » (moins binaire et moins unaire), on peut :

1. Déclarer un symbole terminal supplémentaire UMINUS
2. Déclarer sa précédence différemment de celle de MINUS
3. Utiliser %prec UMINUS dans la clause voulue (avant le bloc de code Java)

1.1.7 Interfacer JFlex et CUP

Il faut indiquer **%cup** dans le préambule du source JFlex.

Dans le source JFlex, les actions doivent renvoyer (via `return`) un objet de type `java_cup.runtime.Symbol`. Cette classe offre un constructeur à quatre paramètres :

`Symbol`(*terminal, ligne, colonne, valeur*)

où :

– le *terminal* est un entier représentant un symbole terminal. Les symboles terminaux sont déclarés dans la classe Classe_sym générée par CUP sous la forme de constantes entières qui sont destinées à être utilisées ici. En plus de EOF, qui désigne la fin de fichier.
– les *ligne* et *colonne* où le token est lu (typiquement yyline et yycolumn), afin de permettre au parseur de signaler une erreur de syntaxe en affichant la ligne et la colonne où ils sont apparus
– la *valeur*, optionnelle, est celle qui doit être « portée » par le symbole terminal, lorsqu'il est déclaré avec un certain type dans la définition du parseur. Ce type doit correspondre à celui de la valeur.

Mais on peut aussi forcer le lexeur à passer au jeton suivant (par exemple pour ignorer des espaces), en utilisant `return next_token`

1.1.8 Utilisation du parseur obtenu

Il faut définir une classe Java supplémentaire pour relier le tout.

La classe *Classe_parseur* créée par CUP est munie d'un constructeur admettant un paramètre. Ce paramètre est un objet de type `java_cup.runtime.Scanner`. Mais grâce à l'option `%cup` spécifiée dans la définition du lexeur, la classe du lexeur créée par JFlex implémente cette interface. Un tel objet lexeur peut donc être utilisé comme paramètre lors de la création.

On rappelle que la classe du lexeur créée par JFlex est munie d'un constructeur admettant un paramètre de type `java.io.InputStrea` qui peut être par exemple l'entrée standard (`System.in`), ou bien un fichier (`FileInputStream`(*nomfich*)).

La classe du parseur propose un membre utile : `parse()` qui effectue le parsing et renvoie un objet correspondant au type du non-terminal point d'entrée de la grammaire (dont la règle a été définie en premier).

1.2 Une calculatrice infixe

L'exemple complet (calc) permet l'utilisation d'une calculatrice infixe totalement parenthésée (i.e. toutes les expressions doivent être parenthésées). Observer son comportement.

Puis, enlever les parenthèses (LPAREN, RPAREN). Que se passe-t-il ?

Définir les bonnes priorités opératoires.

Rajouter ensuite un opérateur unaire postfixe ! pour la factorielle, en laissant une certaine liberté à l'utilisateur de ne pas employer de parenthèses (par exemple, 5 !). Dans quels cas est-ce envisageable ? Il faudra peut-être ajouter une seconde règle (et donc, un second symbole non terminal).

Enfin, ajouter l'opérateur unaire préfixe **fib**, pour la suite de Fibonacci, également en laissant à l'utilisateur une certaine liberté de ne pas employer de parenthèses.

2 Automates

2.1 Modélisation et simulation

Complétez la classe Automates.java. (Il y a environ 9 bouts de code à ajouter en tout, plus ou moins faciles).

2.2 De l'écriture d'expressions régulières à l'automate

Écrire un parseur qui permette à l'utilisateur d'entrer une expression régulière selon la grammaire suivante, et qui construise l'automate correspondant.

$expr ::= @ \mid caractere \mid expr + expr \mid expr\ expr \mid expr*$

où le caractère @ est utilisé pour l'expression régulière désignant le mot vide. On rappelle l'ordre croissant des priorités opératoires : étoile, concaténation, somme.