Master 1 année 2017 -2018
TP n°1 CUP

Introduction

Le but de cet TP est de vous donner les bases nécessaires afin de pouvoir écrire votre analyseur syntaxique en Java. Pour Java, nous utiliserons le couple jflex (http://jflex.de/) et Cup (http://www2.cs.tum.edu/projects/cup/).

Analyseur Syntaxique Cup

`Cup` est programmé en Java. La dernière version est (en 2018) la version 0.11b. L'utilisation de cette version se fait de la façon suivante :

```
java -jar java-cup-11b.jar moncup.cup
```

`moncup.cup` étant le fichier contenant la spécification Cup. A partir d'un fichier d'entrée respectant le format spécifié ci-dessous, Cup génère deux fichiers : un fichier parser.java contenant la classe permettant de parser, et un fichier sym.java contenant les types des objets utilisés comme terminaux (symboles).

Format d'une spécification Cup

Ce fichier de spécification se compose de cinq parties :

La première partie est la spécification optionnelle du package auquel appartiendront les fichiers parser.java et sym.java générés ainsi que les importations éventuelles d'autres packages. Elle contient également des déclarations de code Java qui se retrouveront dans le fichier parser.java généré.

En général par défaut, vous pouvez mettre cette importation **import java_cup.runtime.***

La deuxième partie consiste en la déclaration des terminaux et des non terminaux de la grammaire avec la syntaxe suivante :

```
terminal nom1, nom2, ...;
terminal nom_class nom1, nom2, ...;
non terminal nom1, nom2, ...;
non terminal nom_class nom1, nom2, ...;
```

`nom_class` est un nom de classe JAVA qui correspond aux attributs associés aux symboles de la grammaire :

dans le cas des terminaux, c'est le type des objets associés aux unités lexicales et retournés par l'analyseur lexical ;

dans le cas des non terminaux, c'est le type des objets retournés par les actions sémantiques d'une règle dont le membre gauche est un des non terminaux qui suit nom_class dans la déclaration.

Par exemple, pour compter les occurrences de a dans un mot

```
terminal a,b;
non-terminal Integer S;
```

La trosième partie est la déclaration optionnelle des différentes priorités des opérateurs ainsi que leur associativité ;

La quatrième partie est la déclaration des règles de la grammaire ainsi que des actions sémantiques associées à ces règles.

Elle débute par une déclaration optionnelle de l'axiome

```
start with nonterminal_name;
```

Si cette déclaration est omise, alors le membre gauche de la première règle de production rencontrée sera considéré comme l'axiome.

Viennent ensuite les règles de production qui sont de la forme suivante :

```
nonterminal_name :: = name1 name2 ... namek {: action
semantique :};
```

où name est un symbole de terminal ou de non terminal déclaré dans la troisième partie.

Dans le cas de plusieurs règles ayant le même membre gauche, on utilise | pour séparer les différentes alternatives.

```
nonterminal_name :: = name1 name2 ... namek {: action
semantique :}
                      |
                      name1' name2' ... namek' {: action
semantique :};
```

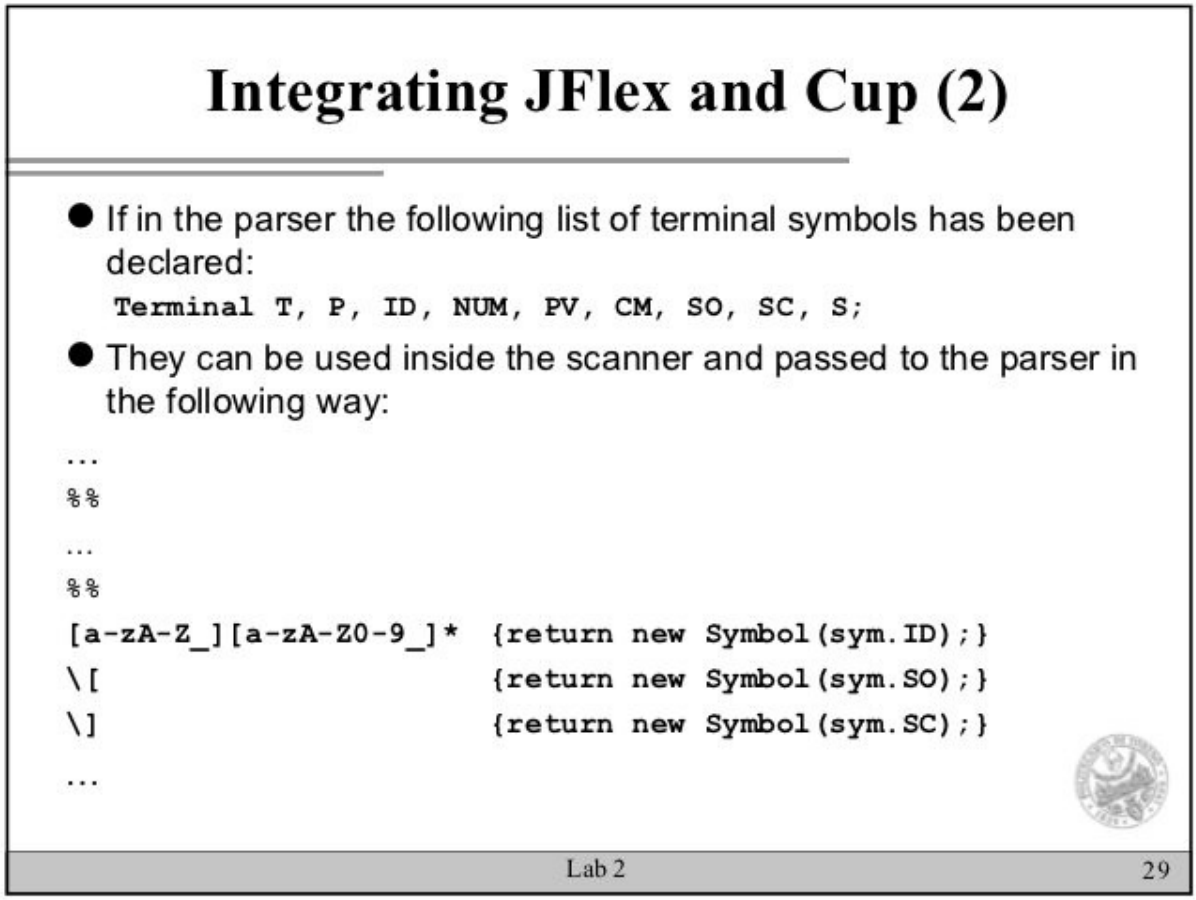
La partie action sémantique est simplement du code JAVA.

Les analyseurs syntaxique et lexical doivent être d'accord sur les valeurs associées à chaque élément lexical (terminal).

Quand l'analyseur lexical reconnaît un token, il doit communiquer cet élément à l'analyseur syntaxique. Cette communication est réalisée par l'intermédiaire de la classe \texttt{Symbol} dont voici quelques uns des constructeurs :

```
public Symbol(int sym_id)
  public Symbol (int sym_id, int value)
  public Symbol(int sym_id, Object o)
```

Les valeurs communiquées à l'analyseur syntaxique sont contenues dans le fichier `sym.java` généré par Cup.



1 Exercice

Nous allons dans cet exercice vérifier syntaxiquement qu'une expression arithmétique est correctement écrite. Nous allons procéder en deux étapes : dans une première partie, nous allons utiliser jflex seul afin de découper le flot de données en éléments lexicaux, puis dans une deuxième étape, nous allons utiliser le couple Cup et jflex afin de vérifier au niveau syntaxique que l'entrée est correcte.

La grammaire que nous allons utiliser est la suivante :

E ::= E + T ;
E ::= E - T ;

E ::= T ;
T ::= T * F;
T ::= F ;
F ::= (E) ;
F ::= NUM;

Cette grammaire doit reconnaître par exemple des expressions arithmétiques comme :

```
3 + 2
3*(8-4)
```

2 Analyse lexicale

En utilisant jflex seul, décrire les spécifications des différents éléments lexicaux. Vérifier que le résultat attendu pour une expression arithmétique de la forme 3*(8-4) soit :

NUM
FOIS
PO
NUM
MOINS
NUM
PF

NUM correspondant à l'élément lexical nombre entier, FOIS au symbole
de multiplication, PO à la parenthèse ouvrante, etc...

3 Analyse lexicale et syntaxique

Nous allons maintenant utiliser de concert le couple Cup et jflex. Le fonctionnement est un peu plus lourd, vous devez respecter la séquence suivante :

- commencer par créer le fichier de grammaire Cup ;
- compiler le fichier Cup : `java -jar java-cup-11b.jar grammaire.cup` ou

cup grammaire.cup

- deux fichiers sont produits : sym.java représentant les différents terminaux à utiliser avec jflex et parser.java qui est l'analyseur syntaxique ;
- reprendre les différentes valeurs fournies par sym.java pour les incorporer dans le fichier jflex ;
- compiler sym.java
- compiler l'analyseur lexical jflex monfichier.jflex. Pour permettre l'interfaçage de jflex et cup vous rajouterez dans la partie importation de votre fichier jflex **import java_cup.runtime.***
- compiler l'analyseur lexical `javac -cp .:java-cup-11b.jar Yylex.java` si son nom est Yylex.java
- compiler l'analyseur syntaxique `javac -cp .:java-cup-11b.jar parser.java`
- compiler le Main qui appelle l'analyseur syntaxique `javac -cp .:java-cup-11b.jar Main.java` et enfin exécuter votre analyseur syntaxique `java -cp .:java-cup-11b.jar Main fichier_a_analyser`

Votre programme Main devra être semblable à celui ci-dessous :

```
import java.io.FileReader;

import java_cup.runtime.*;
public class Main {
    public static  void main(String argv[]) {
        try {
            /* Instantiate the scanner and open input file argv[0] */
            Yylex l = new Yylex(new FileReader(argv[0]));
            /* Instantiate the parser */
            parser p = new parser(l);
            /* Start the parser */
            Object result = p.parse();
            System.out.println("\nfile ok");
        } catch (Exception e) {
            System.out.println("\nSyntax error");
            e.printStackTrace();
        }
    }
}
```

Integrating JFlex and Cup