Université Genoble Alpes **UFR IM²AG**
DU ISN-C2.3 **Année 2016-17**

Langages et Compilation : travail pratique

`JavaCC` est un générateur d'analyseur syntaxique qui produit des analyseurs descendants, écrits en Java. On lui fournit en entrée une spécification des lexèmes du langage (sous forme d'expressions régulières), une spécification de la syntaxe du langage (sous forme d'une grammaire LL et/ou d'expressions régulières), et `javaCC` produit alors un ensemble de classes Java (code source) qui implémentent l'analyseur syntaxique correspondant. Plus d'informations sur le site officiel de l'outil `https://javacc.java.net/` ...

1 Analyse syntaxique

Dans un premier temps, on ne s'intéresse qu'à l'analyse syntaxique "pure", c'est-à-dire uniquement décider si une phrase donnée est conforme ou non à une grammaire hors-contexte.

1.1 Première expérience

On considère le langage défini sur l'ensemble de lexèmes "b", "c", "d" et constitué de l'ensemble de phrases "bc", "bd". Une (mauvaise ?) manière pour décrire ce langage est d'utiliser la grammaire suivante :

A : := B C | B D
B : := b
C : := c
D : := d

La spécification correspondante `Exemple1.jj` (Cf. 3.1) comporte :

— le corps du programme principal, entre les mot-clés `PARSER_BEGIN` et `PARSER_END`. Son rôle est de construire l'analyseur (variable parser), et d'appeler la méthode `Exemple1` de ce parser pour reconnaitre une phrase générée par le non-terminal Exemple1 (qui est l'axiome de notre grammaire).

— la spécification de la grammaire : à chaque non-terminal est associé une méthode écrite en `JavaCC`. Le corps de cette méthode décrit la partie droite des règles associées à ce non-terminal.

Pour exécuter `JavaCC` sur cet exemple tapez la commande : `javacc Exemple1.jj` Comme on pouvait s'y attendre, `javaCC` nous indique que notre grammaire n'est pas LL(1) :

```
Warning: Choice conflict involving two expansions at
         line 36, column 3 and line 37, column 5 respectively.
         A common prefix is: "b"
         Consider using a lookahead of 2 for earlier expansion.
```

Nous avons deux façon de corriger le problème :

1. Une première manière de corriger ce problème est de modifier la grammaire pour la rendre LL(1), par exemple en factorisant les deux règles qui définissent le non-terminal A. Faites-le, et vérifiez que cette fois-ci JavaCC génère correctement un analyseur. L'exécution de la commande `javacc Exemple1.jj` donne `Parser generated successfully.` et un certain nombre de fichiers `.java` sont produits.

   Pour exécuter l'analyseur il faut d'abord le compiler avec la commande : `javac *.java`. Puis on peut exécuter avec la commande : `java Exemple1`. Le programme attend une chaîne de caractères terminée par deux "Ctrl-D" (sans retour a la ligne) . . .Exécutez-le pour tester son fonctionnement sur des chaînes d'entrées correctes ("bc", "bd") ou incorrectes ("bb", "a", ...).

2. Une seconde possibilité offerte par JavaCC est d'indiquer que la définition du non-terminal A nécessite localement une analyse LL(2) pour lever le conflit. On utilise alors le mot-clé LOOKAHEAD, comme indiqué ci-dessous :

   ```
   void A() :
   {}
   {
     LOOKAHEAD(2)
     B() C()
   | B() D()
   }
   ```

   Vérifiez qu'avec cette modification la première grammaire proposée permet bien d'obtenir un analyseur, et que celui-ci fonctionne . . .

1.2 Extension de la lexicographie : prise en compte de séparateurs

On peut étendre l'exemple précédent (fichier `Exemple2.jj`) en modifiant la définition des lexèmes de manière à autoriser la présence de séparateurs (espace, fin-de-ligne, tabulation) entre deux lexèmes. On introduit une section de définition des lexemes dans laquelle une section définit le mot-clé SKIP pour décrire des séparateurs.

```
/* ================================================================== */
/*                      definition des lexemes                        */
/* ================================================================== */

SKIP : // les separateurs
{
  " "
| "\t"
| "\n"
}
```

Générez l'analyseur correspondant et vérifiez qu'il fonctionne correctement.

Remarque : au lieu de la donner au clavier, on peut mettre la chaine à analyser dans un fichier `ex.txt` par exemple et exécuter : `java Exemple1 <ex.txt`.

1.3 Un exemple plus sérieux : expressions arithmétiques

On étudie maintenant un exemple un peu plus élaboré, le langage des expressions arithmétiques construites sur les opérateurs "+" et "*" et des opérandes de type "entier" ou "identificateur", en supposant que ces deux opérateurs sont associatifs à gauche et que la multiplication est plus prioritaire que l'addition.

La grammaire initiale est décrite dans le fichier `Expressions.jj` (Cf. 3.2). Remarquez la section introduite par le mot-clé `TOKEN` pour spécifier les lexèmes par des expressions régulières. Un entier est une suite non vide de chiffre ne commençant pas par 0. Un identificateur est une suite non vide de chiffres ou lettres commençant par une lettre

Comme attendu, la commande `javacc Expressions.jj` nous apprend que l'analyseur ne peut être contruit car la grammaire est récursive à gauche. Ici l'introduction d'un LOOKAHEAD ne résoudrait pas le problème dans le cas général (une grammaire récursive à gauche est non-LL(k) quelque soit k).

Ré-écrivez la spécification `Expressions.jj` en éliminant la récursivité à gauche.

Une règle de la forme "X ->X Y | Z" peut être remplacée par les deux règles :
"X -> Z U" et "U -> Y U | ε".

Dans le formalisme de `JavaCC` une partie droite égale à ε est repréntée par {}. Ainsi la règle "U -> Y U | ε" s'écrit :

```
void U() :
{}
{
   Y() U()
  |
   {}
}
```

On peut aussi exprimer une règle de la forme "X -> X Y | ε" en utilisant une expression régulière en partie droite, comme ceci :

```
void X()
{}
{
   (Y())*
}
```

Après avoir corrigé la grammaire, vérifiez que l'analyseur peut être généré. On peut alors le tester sur les expressions suivantes : "toto", "toto + 4", "toto + 4 * 238", "(toto + 4) * 238", "25 + to#to", "2toto + 1", "x ++", "x + * (4 -2)", en remarquant la pertinence des messages d'erreurs !

2 Programmation d'un interpréteur

Nous allons écrire un interpréteur d'expressions arithmétiques comportant des opérateurs et des entiers. Pour une première version on peut se baser sur la grammaire suivante dans laquelle n désigne une constante entière :

E : : E + T | T
T : : T * F | F
F : : n | (E)

Par la suite vous pourrez introduire d'autres opérateurs binaires ($-, /$, élévation à la puissance) ou unaires ($-$ unaire) ou des constantes réelles ...

Techniquement, la description de la grammaire peut être complétée de façon à calculer la valeur de l'expression analysée.

On peut ainsi modifier le main :

```
public class Expressions {
  public static void main(String args[]) throws ParseException {
    Expressions parser = new Expressions(System.in);
    int val = parser.Expressions();
    System.out.println() ;
    System.out.println("syntaxe correcte !") ;
    System.out.println() ;
    System.out.println("la valeur de l'expresssion est : " + val) ;
  }
}
```

La fonction associée au non-terminal `Expressions` doit alors rendre une valeur entière.

On peut compléter chaque règle par des déclarations de variables (entre les accolades du début de la règle) et du code `Java` (entre les non-terminaux ou les terminaux). Par exemple la description associée à la règle `Expressions -> E` devient :

```
int Expressions1() :
{
int val;
}
{
  val = E()
  {return val;}
  <EOF>
}
```

Il est possible aussi de passer des valeurs en paramètre de la description d'une règle comme par exemple :

```
int X(int i) :
{}
{
   ..................
   {return (i);}
}
```

Enfin pour récuperer la valeur associée au "token" `<entier>` utilisé par exemple dans une règle `X -> <entier>` on utilise la méthode `parseInt` de la façon suivante :

```
int X() :
{
Token t;
}
{
  t=<entier>
  {return Integer.parseInt(t.image);}
}
```

Vous avez maintenant tous les éléments pour compléter le fichier `Expression.jj` et produire un interpréteur d'expressions arithmétiques.

3 Annexes

3.1 Exemple1

```
/* ================================================================================ */
/*                        programme principal                                       */
/* ================================================================================ */
PARSER_BEGIN(Exemple1)
public class Exemple1 {
  public static void main(String args[]) throws ParseException {
    Exemple1 parser = new Exemple1(System.in);
    parser.Exemple1();
    System.out.println() ;
    System.out.println("syntaxe correcte !") ;
  }
}
PARSER_END(Exemple1)

/* ================================================================================ */
/*                        definition de la grammaire                                */
/* ================================================================================ */
void Exemple1() :
// Exemple1 -> A
{}
{
  A() <EOF>
}

void A() :
// A -> B C | B D
{}
{
  B() C()
  | B() D()
}

void B() :
// B -> b
{}
{
  "b"
}

void C() :
// C -> c
{}
{
   "c"
}

void D() :
// D -> d
{}
{
   "d"
}
```

3.2 Le fichier Expressions.jj

```
/* ================================================================================ */
/*       programme principal                  */
/* ================================================================================ */
PARSER_BEGIN(Expressions)
public class Expressions {
  public static void main(String args[]) throws ParseException {
    Expressions parser = new Expressions(System.in);
    parser.Expressions();
    System.out.println() ;
    System.out.println("syntaxe correcte !") ;
  }
}
PARSER_END(Expressions)

/* ================================================================================ */
/*       definition des lexemes                  */
/* ================================================================================ */
SKIP :   // les separateurs de lexeme
{
  " "
| "\t"
| "\n"
}

TOKEN : // les lexemes
{
  < plus: "+" >
| < mult: "*" >
| < parg: "(" >
| < pard: ")" >
| < idf: <lettre> (["0"-"9"] | <lettre>)* >
| < entier: ["1"-"9"] (["0"-"9"])* >
| < lettre: ["A"-"Z"] | ["a"-"z"] >
}
```

```
/* ========================================================================== */
/*       definition de la grammaire              */
/* ========================================================================== */
// Expressions -> E
void Expressions() :
{}
{
  E() <EOF>
}

// E -> E + T | T
void E() :
{}
{
  E() <plus> T()
| T()
}

// T -> T * F | F
void T() :
{}
{
  T() <mult> F()
| F()
}

// F -> idf | entier | (E)
void F() :
{}
{
  <idf>
| <entier>
| <parg> E() <pard>
}
```