Langages et Compilation : travail pratique

1 Un outil pour déterminer si une grammaire est LL(1)

L'outil **Grammophone** permet d'entrer une grammaire, de l'analyser pour savoir si elle est LL(1). Si elle ne l'est pas, on a un diagnostic des règles qui entrent en conflit. Si elle l'est, on peut voir la table qui donne, pour chaque symbole terminal, les règles pour lesquelles ce symbole est directeur.

1.1 Tester grammophone sur une première grammaire

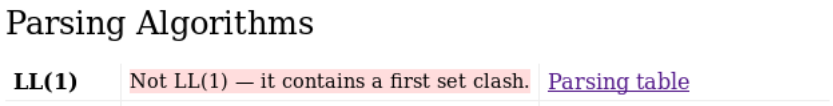
Pour utiliser Grammophone, lancer l'URL `http://mdaines.github.io/grammophone/#` depuis votre navigateur préféré.

1. Saisir, dans la partie gauche de **Grammophone**, les règles de la grammaire du langage $\mathbf{a}^n\mathbf{b}^n\mathbf{c}$**, pour n ≥1** (vue en cours, grammaire G1 diapo 23). Chaque règle doit être de la forme `X -> ... .`, toujours terminée par un `.`

   Grammaire (dans la syntaxe **Grammophone**) :

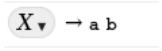
   ```
   Z   ->   X Y .
   X   ->   a b .
   X   ->   a X b .
   Y   ->   c .
   ```

2. Cliquer sur `Analyze` (en haut à gauche).
3. Observer que l'outil mentionne que la grammaire n'est pas LL(1) (voir copie écran ci-dessous) et propose une table (cliquer sur `Parsing Table`) indiquant les calculs de directeurs et, en rose, les règles en conflit.

4. Cliquer sur `Transform` (en haut à gauche) pour voir les possibilités de transformation de la grammaire pour tenter la rendre LL(1).

   La proposition de transformation intéressante est celle obtenue en cliquant sur le X surligné avec une flèche vers le bas (voir copie écran ci-dessus).

5. La proposition est de factoriser le facteur de gauche **a**. Cliquer sur cette proposition.

   On obtient alors une grammaire modifiée, qui est LL(1), et la table des directeurs correspondants. La comparer avec celle vue en cours (diapo 25).

1.2 Essayer avec les autres grammaires vues en cours (et en TD si envie)

Jouer avec l'outil avec les autres grammaires vues en cours (diapos 26, 28, 30) et éventuellement aussi avec celles des deux premiers exercices du TD précédent.

2 Analyse syntaxique

`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/` ...

Avant de commencer ...

1. Démarrer le poste de travail sous Linux (et non Windows ...)
2. Créer un répertoire JavacCC et récupérer les fichiers `Exemple1.jj`, `Exemple2.jj` et `Expressions.jj` depuis la page `http://www-verimag.imag.fr/~mounier/Enseignement/ISN-LT/`

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.

2.1 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. 4.1) 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.

Générer l'analyseur sur cet exemple avec 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 façons 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.
   (a) Modifier la grammaire dans ce sens, et vérifier que cette fois-ci JavaCC génère correctement un analyseur (il est aussi possible d'utiliser **Grammophone** pour faire ce travail de transformation). L'exécution de la commande `javacc Exemple1.jj` donne `Parser generated successfully.` Et un certain nombre de fichiers `.java` sont produits.
   (b) Compiler l'analyseur avec la commande `javac *.java`.
   (c) Exécuter l'analyseur avec la commande `java Exemple1`.
   A chaque modification du fichier `.jj`, il faudra refaire l'enchainement : `javacc Exemple1.jj` puis `javac *.java`, pour générer et compiler l'analyseur.

Pour chaque exécution de l'analyseur : `java Exemple1`

Le programme attend une chaîne de caractères terminée par deux "Ctrl-D" (sans retour à la ligne) ...

(d) Exécuter l'analyseur, 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()
}
```

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

Nous en resterons pour autant dans la suite à des analyses LL(1) et non pas LL(2).

2.2 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 lexèmes dans laquelle une section définit le mot-clé SKIP pour décrire des séparateurs.

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

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

Générer l'analyseur correspondant et vérifier 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 Exemple2 < ex.txt`.

2.3 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"`, en supposant que ces deux opérateurs sont associatifs à gauche et que la multiplication est plus prioritaire que l'addition. La grammaire initiale est la suivante, où n désigne une constante entière :

$$\begin{array}{lll} \mathrm{E} & :: & \mathrm{E} + \mathrm{T} \mid \mathrm{T} \\ \mathrm{T} & :: & \mathrm{T} * \mathrm{F} \mid \mathrm{F} \\ \mathrm{F} & :: & \mathrm{n} \mid (\mathrm{E}) \end{array}$$

Cette grammaire est fournie dans le fichier `Expressions.jj` (Cf. 4.2). Remarquer la section introduite par le mot-clé `TOKEN` pour spécifier les lexèmes par des expressions régulières. Un entier est une suite non vide de chiffres ne commençant pas par 0.

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-écriver la grammaire `Expressions.jj` en éliminant la récursivité à gauche en vous rappelant qu'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 représentée par {}. Ainsi la règle "U -> Y U | ε" s'écrit :

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

Attention : il faut bien respecter cet ordre dans le fichier `.jj`, ε (noté {}) doit apparaitre en dernier ...

Vous pouvez aussi utiliser **Grammophone** pour transformer votre grammaire jusqu'à obtenir une grammaire LL(1).

Une fois la grammaire mise sous forme LL(1), vérifier que l'analyseur peut être généré. On peut alors le tester sur les expressions suivantes : `"42"` , `"42 + 4"`, `"42 + 4 * 238"`, `"(42 + 4) * 238"`, `"25 + 4#2"`, `"4 2 + 1"`, `"12 ++"`, etc., en remarquant la pertinence des messages d'erreurs !

3 Programmation d'un interpréteur

Nous allons maintenant écrire un interpréteur d'expressions arithmétiques comportant des opérateurs et des entiers. Pour une première version, on peut se baser sur la grammaire LL(1) des expressions arithmètiques obtenue dans la section 2.3.

Une grammaire solution LL(1) et le calcul des directeurs correspondants est disponible à l'adresse : `http://www-verimag.imag.fr/~mounier/Enseignement/ISN-LT/SUJETS_COMPIL/exercices_langage_expressions.pdf`.

La partie 3 de ce document explique aussi comment ajouter de l'information pour faire le calcul sémantique de la valeur de l'expression reconnue. Prendre le temps de comprendre cette partie 3 avant de continuer.

Techniquement, la description de la grammaire peut être complétée de façon à calculer la valeur de l'expression analysée. On peut ainsi modifier le programme principal :

```
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 A(int i) :
{
  int j, res ;
}
{
   j = B()
   res = C(i+j)
   {return (res);}
}
```

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

Ceci donne tous les éléments pour compléter le fichier `Expression.jj` et produire un interpréteur d'expressions arithmétiques ...

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

4 Annexes

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

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

4.2 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: ")" >
| < entier: ["1"-"9"] (["0"-"9"])* >
}

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