Université de Lille — FACULTÉ DES SCIENCES ET TECHNOLOGIES — Département Informatique

Informatique

L3S5 Miage

Analyse Syntaxique – TP4

Analyse récursive descendante

Un exemple

Vous trouverez ci-dessous un analyseur syntaxique qui reconnaît les commandes dont la syntaxe est celle d'un exercice vu en TD. Pour rappel :

- Une commande est composée de 4 parties dont seule la première est obligatoire.
- la première partie est constituée du nom de la commande (un identificateur de commande).
- la deuxième partie est composée du nom de la variable (un identificateur) sur laquelle va s'appliquer la commande.
- la troisième partie correspond à la liste des paramètres : c'est une suite de paramètres (qui sont ici des caractères) séparés par des virgules, encadrée par des accolades.
- la dernière partie correspond à la partie option. Elle est constituée par une liste d'options séparées par des virgules, encadrée par des crochets. Une option est représentée soit par un caractère, soit par un ensemble ayant la syntaxe de l'ensemble des paramètres.

Un analyseur récursif descendant LL se construit directement à partir des règles d'une grammaire (en supposant que celle-ci ait certaines bonnes propriétés que l'on précisera plus tard). Un analyseur prédictif LL(1) connaît de plus le premier token non encore reconnu : il peut ainsi déterminer la règle à appliquer en testant ce token et éviter les rebroussements.

Du point de vue de la programmation, il suffit de définir pour chaque variable X de la grammaire une procédure qui sera capable de reconnaître les mots correspondants à chacune des règles associées à la variable X. Ainsi, dans cet exemple, pour la partie paramètres, nous avons la règle de grammaire :

$$\texttt{parametre} \rightarrow \{ \quad \texttt{car} \quad \texttt{suiteparametre} \quad \}$$
$$\texttt{parametre} \rightarrow \quad \epsilon$$

La variable *parametre* possède donc deux règles : la procédure `parametre` associée va prédire grâce au token lu par avance quelle règle appliquer. Il suffit de tester si le token lu est le token correspondant à l'accolade ouvrante, auquel cas on applique la première règle. Appliquer une règle revient à consommer les terminaux (tokens) et/ou à appeler récursivement les non-terminaux (variables) qui la composent. Si le token n'est pas une accolade ouvrante , on applique la seconde règle (en fait, on ne fait rien quand le membre droit d'une règle est ϵ).

D'où le code en Java pour la méthode `parametre` :

```
private void parametre() throws SyntaxException, java.io.IOException{
/* parametre -> TOKEN_ACC_OUV TOKEN_CAR suiteparametre TOKEN_ACC_FER
                | epsilon */
    if (leToken==Lexico.TOKEN_ACC_OUV) {
      lireLeToken();
      verifierToken(Lexico.TOKEN_CAR);
```

```
        lireLeToken();
        suiteparametre();
        verifierToken(Lexico.TOKEN_ACC_FER);
        lireLeToken();
    }
}
```

Pour compiler l'analyseur de commandes, récupérez les fichiers `Lexico.java`, `SyntaxException.java` et `Commande.java` et compilez l'ensemble avec la commande

```
[levaire@zywiec TP1]$ javac Lexico.java SyntaxException.java Commande.java
```

Exécuter l'analyseur avec la commande

```
[levaire@zywiec TP1]$ java Commande
Entrez une commande: id1 id2 {a,b,d} [e, {c, c,c}, r,{r }   ]
Entrez une commande: ,
Exception in thread "main" SyntaxException: TOKEN_IDENTIF token attendu : ligne 1
        at Commande.erreur(Commande.java:26)
        at Commande.verifierToken(Commande.java:32)
        at Commande.nom(Commande.java:52)
        at Commande.commande(Commande.java:116)
        at Commande.analyse(Commande.java:127)
        at Commande.main(Commande.java:137)
[levaire@zywiec TP1]\$
```

qui vous demande de rentrer au clavier des commandes et ceci jusqu'à ce qu'une commande n'ait pas la syntaxe correcte (auquel cas il y a une exception).

Manipulation 1. Pour comprendre le principe de l'analyse récursive descendante, ajoutez des affichages de messages à chaque début de procédure. Vous constaterez ainsi que les appels récurifs aux procédures correspondent à la construction d'un arbre de dérivation.

Expressions arithmétiques

Vous allez construire un analyseur récursif descendant pour des expressions arithmétiques simples. La grammaire G de ces expressions est la suivante

$$
\begin{array}{lll}
G = (& \{\texttt{identif}, \texttt{entier}, \texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}, \texttt{(}, \texttt{)}\}, & \textit{(* les terminaux *)} \\
& \{\text{expression}, \text{suiteExpression}, \text{terme}, \text{suiteTerme}, \text{facteur}, \text{atome}\}, & \textit{(* les non terminaux *)} \\
& \text{expression}, & \textit{(* l'axiome *)} \\
& P) & \textit{(* les règles de production *)}
\end{array}
$$

$$
\text{avec } P = \left\{
\begin{array}{lcl}
\text{expression} & \rightarrow & \text{terme} \quad \text{suiteExpression} \\
\text{suiteExpression} & \rightarrow & \texttt{+} \quad \text{terme} \quad \text{suiteExpression} \mid \epsilon \\
\text{suiteExpression} & \rightarrow & \texttt{-} \quad \text{terme} \quad \text{suiteExpression} \mid \epsilon \\
\text{terme} & \rightarrow & \text{facteur} \quad \text{suiteTerme} \\
\text{suiteTerme} & \rightarrow & \texttt{*} \quad \text{facteur} \quad \text{suiteTerme} \mid \epsilon \\
\text{suiteTerme} & \rightarrow & \texttt{/} \quad \text{facteur} \quad \text{suiteTerme} \mid \epsilon \\
\text{facteur} & \rightarrow & \texttt{(} \quad \text{expression} \quad \texttt{)} \\
\text{facteur} & \rightarrow & \text{atome} \\
\text{atome} & \rightarrow & \texttt{identif} \mid \texttt{entier} \mid \texttt{-} \quad \texttt{identif} \mid \texttt{-} \quad \texttt{entier}
\end{array}
\right\}
$$

Question 1. *Construire un analyseur récursif descendant pour cette grammaire*[1] *en vous basant sur l'exemple précédent. Utilisez les fichiers* `Lexico.java`, `SyntaxException.java` *et* `Syntax.java`. *Le dernier fichier contient le squelette d'un analyseur récursif descendant qu'il vous reste à compléter pour reconnaître les expressions arithmétiques.*

Javacc

`javacc` est un outil permettant de construire un analyseur descendant pour une grammaire LL(1). Il prend en entrée un fichier de description de la grammaire et produit un ensemble de fichiers source Java qu'il suffit de compiler. Sans entrer dans les détails, un fichier de description est composée de :

1. une liste facultative d'options : nous n'en utiliserons pas ici ;
2. une unité de compilation Java placée entre les déclarations `PARSER_BEGIN(parser_name)` et `PARSER_END(parser_name)` : c'est du code source Java qui doit contenir une classe de nom `parser_name`. `javacc` rajoutera le code Java nécessaire à la construction d'un analyseur syntaxique, en particulier le constructeur. Celui pourra être invoqué depuis votre code Java pour procéder à l'analyse syntaxique d'un texte : pour les expressions arithmétiques, nous invoquons ainsi le constructeur `Expr` avec comme argument l'entrée standard.

```
PARSER_BEGIN(Expr)

/** Expressions arithmetiques. */
public class Expr {

  /** Main entry point. */
  public static void main(String args[]) throws ParseException {
    Expr parser = new Expr(System.in);
    parser.listeExpression();
  }

}

PARSER_END(Expr)
```

3. l'ensemble des règles de production de la grammaire : en plus des règles classiques, `javacc` y inclut les productions d'expressions régulières liées à la phase d'analyse lexicale. Nous déclarons ainsi les caractères non significatifs (`SKIP`) et les unités lexicales (tokens ou lexèmes : `TOKEN`) :

```
SKIP :
{
  " "
| "\t"
}
TOKEN :
{
  <TOKEN_ENTIER: ["0"-"9"](["0"-"9"])* >
| <TOKEN_IDENTIF: ["A"-"Z","a"-"z"](["A"-"Z","a"-"z"])*>
| <TOKEN_EOL: ["\n","\r"]>
```

1. Elle n'est pas complétement factorisée à gauche, `atome` comporte deux règles commencant par `TOKEN_MOINS`. Je vous laisse le soin de la factoriser.

```
| <TOKEN_PAR_OUV: "(">
| <TOKEN_PAR_FER: ")">
| <TOKEN_PLUS: "+">
| <TOKEN_MOINS: "-">
| <TOKEN_MULT: "*">
| <TOKEN_DIV: "/">
| <TOKEN_CRO_OUV: "[">
| <TOKEN_CRO_FER: "]">
| <TOKEN_PT_VIRGULE: ";">
| <TOKEN_2_PTS: ":">
| <TOKEN_CAR: "[A-Za-z]">
| <TOKEN_ACC_OUV: "{">
| <TOKEN_ACC_FER: "}">
| <TOKEN_VIRGULE: ",">
}
```

La grammaire elle-même se décrit par l'intermédiaire de méthodes Java (un peu comme vous avez fait dans l'analyse récursive descendante). On déclare donc une méthode par variable de la grammaire en utilisant une syntaxe particulière :

```
/** Root production. */

void listeExpression():
{}
{
 ( expression() <TOKEN_EOL> )* <EOF>
}

void expression() :
{}
{
   terme()    suiteExpression()
  { System.out.println("Expression Ok."); }
}

void terme() :
{}
{
  facteur() suiteTerme()
}

void suiteExpression() :
{}
{
  <TOKEN_PLUS> terme() suiteExpression()
| <TOKEN_MOINS> terme() suiteExpression()
| {}
}

void suiteTerme() :
{}
{
  <TOKEN_MULT> facteur() suiteTerme()
| <TOKEN_DIV> facteur() suiteTerme()
```

```
| {}
}

void facteur() :
{}
{
  <TOKEN_PAR_OUV> expression() <TOKEN_PAR_FER>
|  atome()
}

void atome() :
{}
{
  <TOKEN_MOINS> valeur()
| valeur()
}

void valeur():
{}
{
  <TOKEN_IDENTIF>
| <TOKEN_ENTIER>
}
```

Pour compiler cet exemple, récupérer le fichier `expr.jj` et exécuter les commandes suivantes :

```
levaire@lxo26:~/TPJavacc/expr$ javacc-5.0/bin/javacc expr.jj
...
Parser generated successfully.
levaire@lxo26:~/TPJavacc/expr$ javac *.java
```

Un exemple d'exécution de l'analyseur construit :

```
levaire@lxo26:~/TPJavacc/expr$ java Expr
3+(aa-4) * 16
Expression Ok.
Expression Ok.
5-6
Expression Ok.
^D
levaire@lxo26:~/TPJavacc/expr$
```

Il est possible de voir l'arbre de syntaxe abstraite pendant l'analyse en utilisant le mode DEBUG de `javacc` :

```
levaire@lxo26:~/TPJavacc/expr$ javacc-5.0/bin/javacc -DEBUG_PARSER=true expr.jj
...
Parser generated successfully.
levaire@lxo26:~/TPJavacc/expr$ javac *.java
levaire@lxo26:~/TPJavacc/expr$ java Expr
Call:   listeExpression
3+(aa-4)
  Call:   expression
    Call:   terme
      Call:   facteur
```

```
        Call:   atome
          Call:   valeur
            Consumed token: <<TOKEN_ENTIER>: "3" at line 1 column 1>
          Return: valeur
        Return: atome
      Return: facteur
      Call:   suiteTerme
      Return: suiteTerme
    Return: terme
    Call:   suiteExpression
      Consumed token: <"+" at line 1 column 2>
      Call:   terme
        Call:   facteur
          Consumed token: <"(" at line 1 column 3>
          Call:   expression
            Call:   terme
              Call:   facteur
                Call:   atome
                  Call:   valeur
                    Consumed token: <<TOKEN_IDENTIF>: "aa" at line 1 column 4>
                  Return: valeur
                Return: atome
              Return: facteur
              Call:   suiteTerme
              Return: suiteTerme
            Return: terme
            Call:   suiteExpression
              Consumed token: <"-" at line 1 column 6>
              Call:   terme
                Call:   facteur
                  Call:   atome
                    Call:   valeur
                      Consumed token: <<TOKEN_ENTIER>: "4" at line 1 column 7>
                    Return: valeur
                  Return: atome
                Return: facteur
                Call:   suiteTerme
                Return: suiteTerme
              Return: terme
              Call:   suiteExpression
              Return: suiteExpression
            Return: suiteExpression
Expression Ok.
          Return: expression
          Consumed token: <")" at line 1 column 8>
        Return: facteur
        Call:   suiteTerme
        Return: suiteTerme
      Return: terme
      Call:   suiteExpression
      Return: suiteExpression
    Return: suiteExpression
Expression Ok.
  Return: expression
```

```
  Consumed token: <<TOKEN_EOL>: "
" at line 1 column 9>
^D
```

Question 2. *Modifiez le fichier* `expr.jj` *pour ajouter un token* `let`*, un token* `=` *et les règles suivantes pour les listes d'expressions/affectations :*

$$
\begin{array}{lcl}
\textit{listeAffExpr} & \rightarrow & \textit{AffExprlisteAffExpr} \\
\textit{listeAffExpr} & \rightarrow & \epsilon \\
\textit{AffExpr} & \rightarrow & \textit{expression} \mid \textit{affectation} \\
\textit{affectation} & \rightarrow & \texttt{let identif =} \ \textit{expression} \\
\textit{expression} & \rightarrow & \textit{terme} \quad \textit{suiteExpression}
\end{array}
$$

Vous trouverez la documentation sur le site de `javacc`*.*