Introduction à JavaCC

Générateur d'analyseur syntaxique

Définition d'un langage

Introduction

- Un langage est un ensemble de phrases constituées à partir d'un alphabet. Tout langage se définit par sa syntaxe et sa sémantique.

Syntaxe : ensemble de règles d'organisation

- lexicographie
  - assemblage des signes de l'alphabet en unités lexicales (lexème ou mot)
- grammaticale
  - agencement des lexèmes en structures syntaxiques cohérentes (phrases)
- exemple du français:
  - alphabet: lettres
  - lexèmes: mots du dictionnaire, symboles de ponctuation
  - grammaire: Phrase = Sujet Verbe Complément

Sémantique : sens du langage (interprétation des phrases)

- problème de l'ambiguïté

  « le pilote ferme la porte », « il a rapporté un vase de chine »

Notion de grammaire (http://fr.wikipedia.org/wiki/Grammaire_formelle)

Définition d'une grammaire

- Une grammaire est un moyen fini de définir la syntaxe des énoncés possibles d'un langage au moyen de:
  - symboles terminaux : symboles du langage étudié
  - symboles non terminaux : symboles du méta-langage de description
  - axiome : symbole non terminal à partir duquel tout énoncé du langage est dérivé
  - règles de grammaire : définissant l'ensemble des non-terminaux de la grammaire
- Exemple du français
  - symboles terminaux : • , le , chien, soupe , mange ...
  - symbole non terminaux : phrase, groupeVerbal, groupeNominal ...
  - axiome: phrase
  - règles: une phrase est définie comme un groupeNominal suivi d'un groupeVerbal suivi d'un point

Notion de grammaire

Notations

– Il existe plusieurs manières de décrire les règles de grammaire

- BNF (Backus-Naur Form : http://fr.wikipedia.org/wiki/Forme_de_Backus-Naur)
- *Diagrammes syntaxiques*

EBNF (attention nombreux dialectes)

Règle 1 **phrase ::= GN GV "." .**
Règle 2 **exponent_part ::= "e" ["+" | "-"] decimal_digits .**
Règle 3 **decimal_digits ::= "0..9" "0..9" * .**
::= symbole de définition, se lit "est défini comme"
| alternative, se lit "ou"
[ ]partie facultative, se lit "optionnel"
*** répétition, se lit "répéter 0,1 ou n fois le terme précédent"**
+ répétition, se lit "répéter 1 ou n fois le terme précédent"
() groupement

– la juxtaposition des symboles se lit "suivi de"
– les symboles entre guillemets sont terminaux, les autres sont les non-terminaux
– le point termine une règle
– rq: "a..b" signifie "tout symbole entre les bornes a et b"

Analyse syntaxique

Utilisation d'une grammaire

- Génération de textes: on peut construire les énoncés possibles du langage en appliquant successivement toutes les règles à partir de l'axiome. Attention la grammaire peut décrire un ensemble non fini.
- Analyse d'un texte: la grammaire permet de vérifier qu'une séquence de symboles terminaux est un énoncé valide du langage du point de vue syntaxique, et sémantique si l'on décore celle-ci des règles appropriées.

Analyse syntaxique (http://en.wikipedia.org/wiki/Parsing)

- Analyseur descendant
  - LL (**L**eft-to-right, **L**eftmost derivation)
    - LL(1) *ou LL(k) qui fixe le nombre de lookahead nécessaires à la désambiguïsation.*
- Analyseur ascendant (**L**eft-to-right, **R**ightmost derivation)
  - *Simple LR (SLR)*
  - *LALR*
  - *Canonical LR (LR(1))*

Analyse syntaxique

Exemple: un petit sous-ensemble du français

Vocabulaire terminal:

{ "chat", "souris", "voit", "mange", "le", "la", "petit", "petite", "." }

Vocabulaire non terminal:

{ phrase, groupe verbal (GV), groupe nominal (GN), nom, adjectif, article , verbe }

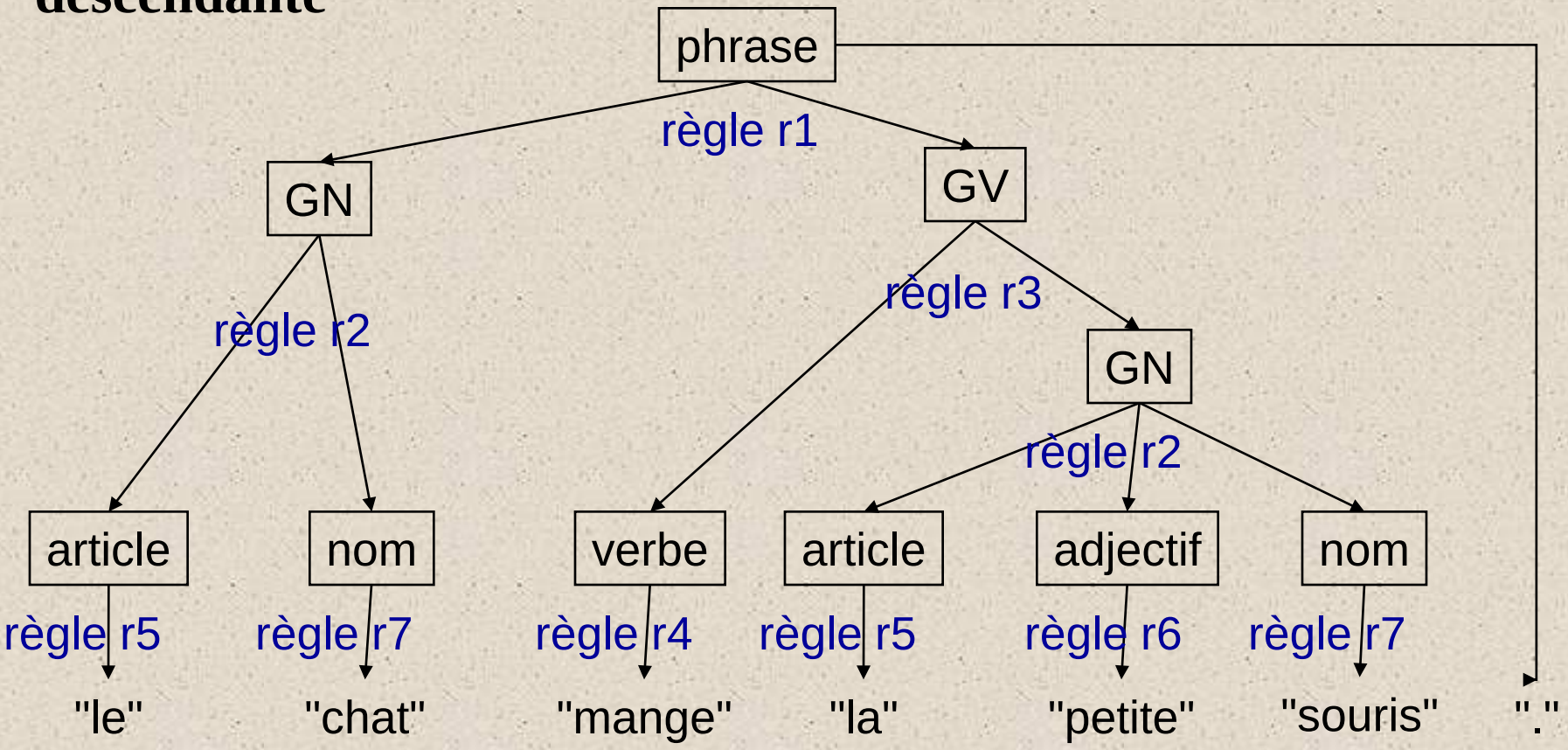
Axiome: **phrase**

règles:

r1	**phrase**	**::=**	**GN GV "." .**
r2	**GN**	**::=**	**article adjectif nom | article nom .**
r3	**GV**	**::=**	**verbe GN .**
r4	**verbe**	**::=**	**"voit" | "mange" .**
r5	**article**	**::=**	**"le" | "la" .**
r6	**adjectif**	**::=**	**"petit" | "petite" .**
r7	**nom**	**::=**	**"chat" | "souris" .**

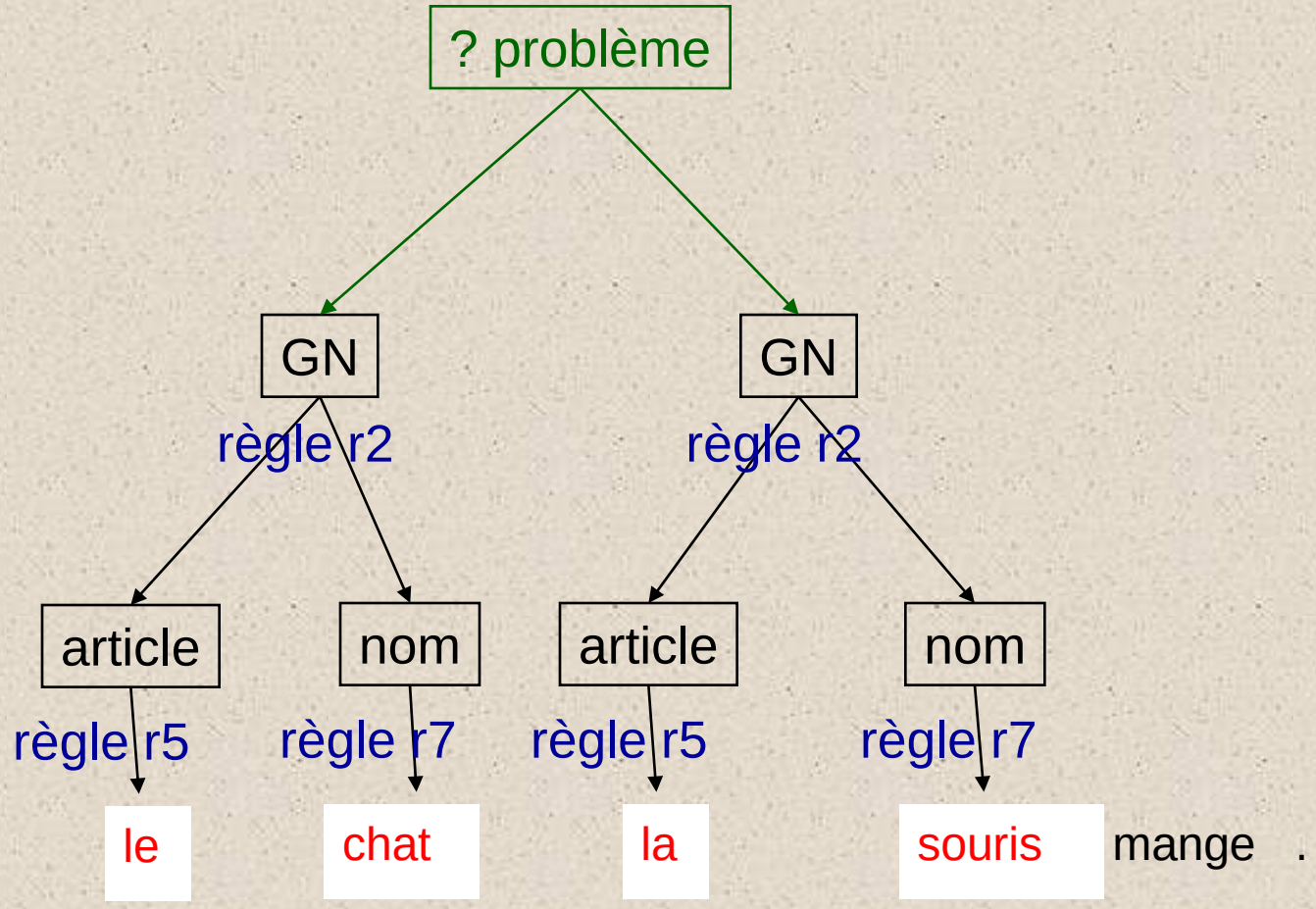
Analyse syntaxique

"le chat mange la petite souris .": arbre syntaxique avec analyse descendante



Analyse syntaxique

"le chat la souris mange . ": erreur syntaxique avec analyse ascendante



JavaCC (https://javacc.dev.java.net/)

Java Compiler Compiler (JavaCC) est un logiciel destiné à faciliter la réalisation de programmes informatiques en langage Java. En plus d'être un générateur d'analyseur (ou parser) - un outil qui lit les spécifications d'une grammaire et qui la convertit en programme Java- , JavaCC fournit d'autres possibilités relatives à la génération de parser comme la construction d'arbre et le débogage.

JavaCC prend comme entrée un fichier **MaGrammaire.jj** qui contient entre autres les descriptions des règles de la grammaire et produit un parser descendant (dans le fichier MaGrammaire.java). Une classe **MaGrammaire** est définie dans le fichier java. Elle implémente l'interface MaGrammaireConstants, définie dans MaGrammaireConstants.java et qui contient les définitions des mots clés de la grammaire.

Dans la suite de cette présentation nous utiliserons un langage très simple pour illustrer l'utilisation de JavaCC. C'est est un langage (**kkmel**) d'expression réduit aux opérateurs :

- let : définit une variable, évalue l'expression associée dans le contexte augmenté de cette variable
- * : effectue une multiplication de réels (float dans notre cas)
- + et – unaires

Exemple de phrase en kkmel

```
let rayon = 5.0 in
  let PI=3.14 in
    2.0 * PI * rayon ;; // calcul du périmètre d'un cercle
```


JavaCC

Structure générale d'une description JavaCC contenu dans un fichier <X>.jj

<javacc option>

PARSER_BEGIN (<identificateur>)

<unité de compilation de nom identificateur >

PARSER_END (< identificateur >)

<productions grammaticales>

Nota : les productions sont

- soit des expressions régulières
  => pour définir les unités lexicales (ou lexèmes ou token)
- soit des productions EBNF
  => pour définir les règles de grammaire (syntaxiques)

Options (https://javacc.dev.java.net/doc/docindex.html)

Permet de fixer les options de Javacc

- Ce sont de couples : *attribut* = *valeur*;
- Ceux-ci peuvent surcharger les options de la commandes
- The integer valued options are:
  - LOOKAHEAD (default 1) = nbre maximum de symboles à examiner pour prendre une décision (LL(1)...LL(k))
  - …
- The boolean valued options are:
  - STATIC (default true) = l'analyseur produit est une classe avec méthodes static (un seul analyseur)
  - UNICODE_INPUT (default false) = permet de travailler en unicode plutôt qu'ASCII …
- The string valued options are:
  - OUTPUT_DIRECTORY (default Current Directory)
  - JDK_VERSION (1.6)
  - …

Gestionnaire des lexèmes

Expressions régulières décrivant l'analyseur lexicographique

– Format d'une règle lexicographique :

Expression Régulière

- ["<" sous-analyseur ">"] operateur "{" ER"}"
- L'analyseur lexicographique peut être décomposé en sous-analyseurs (ensemble d'automates)
  – Il existe un sous-automate standard nommé DEFAULT.
  – Voir exemple avec MORE et SPECIAL_TOKEN

Opérateurs

– SKIP

- Elimine les ER stipulées de la séquence d'entrée. Souvent réduit aux caractères non pris en compte.

```
SKIP : {" " | "\t" | "\"" | "\\" }
```

– MORE

- Continue la construction d'un lexème avec l'ER stipulée.
  – Utile pour factoriser un préfixe d'autres ER

```
MORE : { "//" : IN_SL_COMMENT}
```

Passage dans le sous-automate IN_SL_COMMENT

Token Manager

– TOKEN

- Construit un lexème avec l'ER stipulée, le renvoie à l'analyseur.

```
TOKEN [IGNORE_CASE]:{
        <LPAR:   "("> |
        <RPAR:   ")" > |
        <ID:    <LETTER> ( <DIGIT> | <LETTER> )* > |
        <#LETTER: ["a"-"z","A"-"Z"] >
}
TOKEN:{
   <EOL:"\n"  | "\r"  | "\r\n">
}
```

case "insensitive" pour ce token

Meta-terminal

ER interne utilisée uniquement dans une autre ER

– SPECIAL_TOKEN

- Tokens qui peuvent apparaître partout et qui peuvent servir de balise sans faire partie de la syntaxe.

```
<IN_SL_COMMENT> SPECIAL_TOKEN :{<SL_COMMENT: "\n" | "\r" | "\r\n"> :
   DEFAULT}
<IN_SL_COMMENT> SKIP :{< ~[] >}
```

Règle appartenant au sous-automate IN_SL_COMMENT, transfert dans le sous-automate DEFAULT

Retour dans DEFAULT

Négation du vide = tout

Compléter la grammaire

Donner les règles lexicographiques pour compléter la grammaire de notre langage.

```
SKIP : { " " | "\t" | "\n" | "\r" }                    // chaîne à "sauter ": espace, tab, saut ligne, RC
TOKEN [IGNORE_CASE] : {                                // mot-clés du langage
      < LET: "let">
   |  < IN: "in">
}
TOKEN : {
      < CSTE: (["0"-"9"])+ "." (["0"-"9"])+>           // constante réelle
   |  < ID: (["A"-"Z", "a"-"z"])+ >                    // identificateur
}
TOKEN : {                                              // opérateurs
      < ASSIGN: "=" >
   |  < MUL: "*" >
   |  < PLUS: "+" >
   |  < MOINS: "-" >
}
MORE :{                                                // traitement des commentaires fin de ligne
   "//" : IN_SL_COMMENT>
}
<IN_SL_COMMENT> SPECIAL_TOKEN :{ "\n" | "\r" | "\r\n" : DEFAULT} // retour à DEFAULT
<IN_SL_COMMENT> SKIP :{< ~[] >}                        // ER qui reconnait n'importe quel caractère
```


Règle de production

Chaque règle syntaxique est représentée par une méthode

- Ayant le nom de la règle (partie gauche)
- Complétée par la production (partie droite)

```
Production  ::= java_modifier  java_type  java_identifier "(" java_parameter_list ")"  ":"
                java_block
                "{" expansion_choices "}"
expansion_choices ::= expansion ( "|" expansion )*
expansion ::= ( expansion_unit )*
expansion_unit ::=
                java_block |
                "(" expansion_choices ")" [ "+" | "*" ] |
                "[" expansion_choices "]" |
                [ java_assignment_lhs "=" ] regular_expression |
                [ java_assignment_lhs "=" ] java_identifier "(" java_expression_list ")"
regular_expression ::=      "<" java_identifier ">" | "<" "EOF" ">"
```


Règle de production

Règle

float Constante() :

{}

{

<CSTE>

| ***<PLUS>*** ***<CSTE>***

| ***<MOINS>*** ***<CSTE>***

}

Primitive Java liée à la syntaxe

```
/** réalignement en cas d'erreur de syntaxe sur une fin de ligne */
private void skiptoEOL() {
   while(token.kind != EOL){getNextToken();}
}
```

Le lexème courant

La nature du lexème courant

Acquisition du prochain lexème

Compléter la grammaire

Donner les règles JavaCC pour compléter la grammaire de notre langage.

Expr ::= Factor [MulExpr] .

Factor :: = Let | Constante | Variable .

Let ::= "let" Decl "in" Expr "; " .

Decl ::= ID "=" Expr .

MulExpr ::= "*" Expr .

Constante ::= [["+","-"]] CSTE .

Variable ::= ID .

– ID est un méta-terminal constitué uniquement de lettres.

– CSTE est un méta-terminal (réel) constitué de chiffres, d'un point et d'au moins un chiffre.

```
void Expr() :{}{
   Factor() [ MulExpr() ]
}
void Factor() : {}{
       Let()
   |   Constante()
   |   Variable()
}
void Let() : {}{
   <LET> Decl() <IN> Expr() ";"
}
void Decl(){}{
   <ID> <ASSIGN> Expr()
}
void MulExpr() : {}{
   "*" Expr()
}
void Constante() : {}{
   <CSTE>
|  <PLUS>  <CSTE>
|  <MOINS> <CSTE>
}
void Variable() : {}{
    <ID>
}
```

Grammaire attribuée

Pour traiter les règles sémantiques, on introduit des actions sémantiques pour :

- Vérifier de propriétés.
- Produire un résultat

– Nécessite de disposer des informations utiles à cette opération:

- Introduction d'attributs (au sens des grammaires) hérités et synthétisés :
  - Paramètres en entrée et/ou sortie des règles qui permettent de faire descendre ou remonter dans l'arbre d'analyse les informations désirées
  - Généralement réduit à 1 en sortie dans la plupart des outils.
- Utilisation d'attributs
  - NT0↑v1↓p1 ::= NT1 ↑v1↓f(p1)

Lieu de la définition des attributs hérités

Type de l'attribut synthétisé

Récupération de l'attribut synthétisé

Production de l'attribut synthétisé

```
float Constante() : // Constante ::= [["+","-"]] CSTE
{Token valeur;}
{
   valeur=<CSTE> {return Float.parseFloat(valeur.image);}
|  <PLUS>  valeur=<CSTE> {return Float.parseFloat(valeur.image);}
|  <MOINS> valeur=<CSTE> {return -Float.parseFloat(valeur.image);}
}
```


La grammaire de KKmel

Réalisation des méthodes d'utilisation du parser.

```
PARSER_BEGIN(KKmel)
package jus.aoo.kkmel;
import java.io.*;
import java.util.*;
public class KKmel {
   private static Map<String,Float>  variables = new HashMap<String,Float> ();
   public static void  parse(){
      JFileChooser fc = new JFileChooser(new File(System.getProperty("user.dir")+"/exemples"));
      fc.setDialogTitle("Open file");
      fc.setApproveButtonText("Parse");
      fc.showOpenDialog(null);
      File f = fc.getSelectedFile();
      if (f == null || !f.exists()) return;
      try{ KKmel parser = new KKmel(new FileInputStream(f));
            parser.Expr (); System.out.println("analyse terminée");
      }catch(Exception e){e.printStackTrace();}
   }
PARSER_END(KKmel)
```

Gestion élémentaire des variables avec une Map. Dans un premier temps on considère que toutes les variables ont des noms différents.

Contenu du fichier expr.kkmel

```
let rayon = 5.0 in
   let PI=3.14 in
      2.0 * PI * rayon ;; // calcul du périmètre d'un cercle d'un rayon donné
```


La grammaire de KKmel

```
/* Skip whitespace */
SKIP : { " " | "\t" | "\n" | "\r" }
TOKEN [IGNORE_CASE]: {
  < LET: "let">
| < IN: "in">
}
TOKEN: {
  < CSTE: (["0"-"9"])+ "." (["0"-"9"])+ >    // constante réelle
| < ID: (["A"-"Z", "a"-"z"])+ > // identificateur
}
TOKEN: {
  < ASSIGN: "=" >
| < MUL: "*" >
| < PLUS: "+" >
| < MOINS: "-" >
}
MORE : {
  "//" : IN_SL_COMMENT
}
<IN_SL_COMMENT> SPECIAL_TOKEN :{<SL_COMMENT: "\n" | "\r" | "\r\n"> : DEFAULT}
<IN_SL_COMMENT> SKIP :{< ~[] >}
```


La grammaire de KKmel

```
void Expr() : {} {
  Factor() [ MulExpr() ]
}

void Factor() : {} {
  Let()
| Constante()
| Variable()
}

void Let() : {} {
  <LET> Decls() <IN> Expr() ";"
}

void Decls() : {} {
  Decl() ["," Decls()]
}

void Decl() : {Token t;} {
  t=<ID> <ASSIGN> Expr()
}
```

La définition de ce token (",") n'est pas essentielle, il peut être dénoté par sa valeur

```
void MulExpr() : {} {
  "*" Expr()
}

void Constante() : {} {
    <CSTE>
|   <PLUS>  <CSTE>
|   <MOINS> <CSTE>
}

void Variable() : {}{
  <ID>
}
```


La grammaire de KKmel

```
float Expr() : {float v1,v2;}{
   v1=Factor() [ v2=MulExpr(v1) ] {return v2;}
}
float Factor() : {float v;}{
       v=Let() {return v;}
   |   v=Constante() {return v;}
   |   v=Variable() {return v;}
}
float Let() : {float v;}{
   <LET> Decl() <IN> v=Expr() ";" {return v;}
}
void Decl() : {Token var; float v;}{     // pas de valeur retournée mais mis à jour du contexte
    var=<ID> <ASSIGN> v=Expr() {variables.put(var.image, v);}
}
float MulExpr(float v1) : {float v2;}{
  "*" v2=Expr() {return v1*v2;}
}
float Constante() : {Token valeur;}{
   valeur=<CSTE> {return Float.parseFloat(valeur.image);} // construction du réel à partir du token (String)
|  <PLUS>  valeur=<CSTE> {return Float.parseFloat(valeur.image);}
|  <MOINS> valeur=<CSTE> {return -Float.parseFloat(valeur.image);}
}
float Variable() : {Token valeur;}{
  valeur=<ID> {return variables.get(valeur.image);}
}
```

Décorer les règles de la grammaire de notre langage pour produire la β-réduction de l'expression.

Réalisation des méthodes d'utilisation du parser avec évaluation de l'expression KKmel

```
PARSER_BEGIN(KKmel)
...
public class KKmel {
  private static Map<String,Float>  variables = new HashMap<String,Float> ();
  public static void  parse(){
    ...
    float result ;
    try{
        KKmel parser = new KKmel(new FileInputStream(f));
        result = parser.Expr ();
        System.out.println("analyse terminée – valeur de l'expr = " + result);
    } catch(Exception e) {e.printStackTrace();}
  }
PARSER_END(KKmel)
```