4A IR - CS444

TP Langages et Compilation
2016 / 2017

Passe 1

Analyse lexicale et syntaxique, construction de l'arbre abstrait

- Ioannis Parissis
- Catherine Oriat

Plan

- **Analyse lexicale**
  - JFlex
  - Mise en oeuvre de l'analyse lexicale
  - Travail à effectuer
- **Analyse syntaxique**
  - Les arbres
  - Cup
  - Principe de l'analyse syntaxique
  - Travail à effectuer

Plan

- **Analyse lexicale**
  - JFlex
  - Mise en oeuvre de l'analyse lexicale
  - Travail à effectuer
- **Analyse syntaxique**
  - Les arbres
  - Cup
  - Principe de l'analyse syntaxique
  - Travail à effectuer

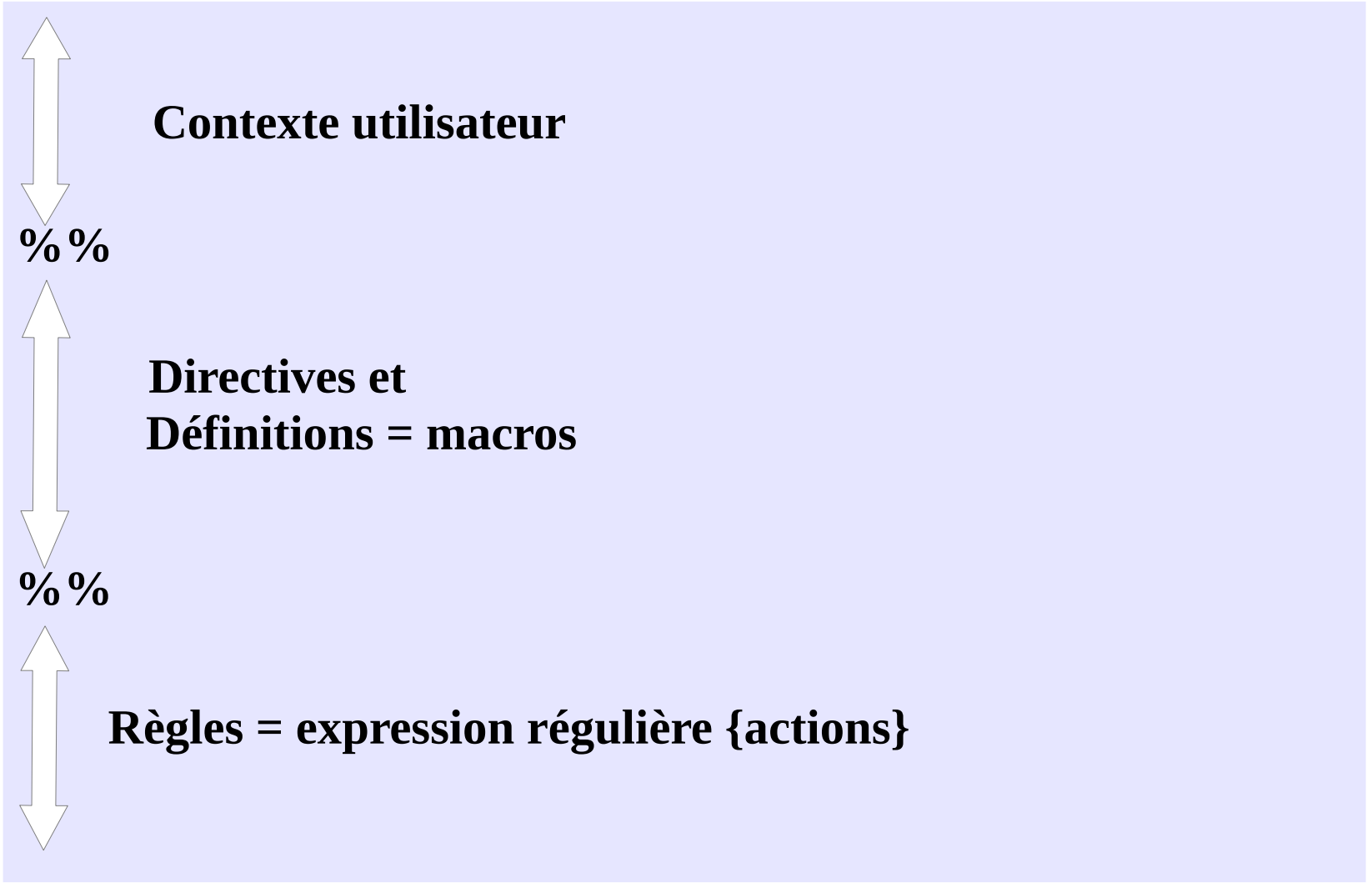
Analyse lexicale
– JFlex

- JFlex : générateur d'analyseur lexical pour Java
  - dans la lignée de Lex, générateur d'analyseur lexical pour C.
- JFlex permet, à partir d'un fichier d'entrée qui décrit la forme d'un ensemble de lexèmes, de générer un analyseur lexical.

```
java -jar ../../Global/Bin/JFlex.jar lexical.flex
```

- Cette commande produit le fichier Lexical.java.
  - On trouve dans ce fichier :
    - La méthode Symbol next_token(), qui permet de lire le lexème suivant ;
    - La méthode String yytext(), qui permet de récupérer la chaîne de caractères qui correspond au lexème qui vient d'être reconnu.

Analyse lexicale
– Forme du fichier lexical.flex



Analyse lexicale
– Forme du fichier lexical.flex

- **Expression régulière**

x	Le caractère ‘x’
"**(**"	Le caractère ‘(’
\(	Le caractère ‘(’
"**:=**"	Le caractère ‘:’ suivi du caractère ‘=’

Ensemble des caractères spéciaux (appelés « opérateurs » dans JFlex)

" \ { } [ ] ^ $ < > ? . * + | ( ) /

{IDF}	La définition IDF
x*	x répété 0, 1, ... n fois
x+	x répété 1, 2, ... n fois
x?	x optionnel (0 ou 1 fois)
x|y	x ou y
[abc]	‘a’, ‘b’ ou ‘c’
[A-Za-z]	caractères entre ‘A’ et ‘Z’ et ‘a’ et ‘z’
[^abc]	tout caractère sauf ‘a’, ‘b’ et ‘c’
\n	retour à la ligne
\t	tabulation
\040	espace
\041	‘!’
\042	‘“’
\134	‘\’
\176	‘~’
.	Tout caractère sauf \n

Analyse lexicale
– Forme du fichier lexical.flex

- **Définitions**
  - Dans la partie “Définition”, on associe à des noms des expressions régulières.
  - Exemple:

```
CHIFFRE [0-9]

LETTRE [A-Za-z]

IDF {LETTRE}({LETTRE}|{CHIFFRE}|"_")*
```

  - Remarques:
    - Les expressions régulières ne doivent pas contenir d'espaces.
    - Dans les noms, les minuscules/majuscules sont significatives: `CHIFFRE` et `chiffre` sont deux noms différents.

Analyse lexicale
– Forme du fichier lexical.flex

- **Règles**
  - Une règle est une expression régulière suivie d'une suite d'actions :

```
Expression régulière { actions }
```

  - Une règle associe à une expression régulière des actions à effectuer.
  - Les actions sont des instructions Java.
    - Exemple:

```
":=" { return symbol(sym.AFFECT); }
"/=" { return symbol(sym.DIFF); }
```

Analyse lexicale
– Forme du fichier lexical.flex

- **Reconnaissance du préfixe le plus long possible**
  - Lorsqu'il y a une ambiguïté sur la règle à appliquer, comme

```
toto     { action1 ; }
tototi   { action2 ; }
```

    on reconnaît le préfixe le plus long possible. Si la chaîne d'entrée est `tototi`, on applique `action2`, si la chaîne d'entrée est `totota`, on applique `action1`.

  - Lorsque les deux chaînes sont de la même longueur, on applique la première règle :

```
toto     { action1 ; }
tot.     { action2 ; }
```

  - Si la chaîne d'entrée est `toto`, on applique `action1`

Analyse lexicale

- **Exemple**
  - Affichage des commentaires d'un programme JCas.

Remarque

```
write("-- Ceci n’est pas un commentaire") ;
```

Analyse lexicale

```
//-----------------------------------------------
//   Fichier d'entrée pour jflex : comment.flex
//-----------------------------------------------
import java.io.*;
%%
// Début de la partie "directives JFLex"

// Nom de la classe qui contient l'analyseur lexical.
// Par défaut : Yylex.
%class Commentaire
%public
// Nom de la fonction donnant la prochaine unité lexicale
%function next_token
//Type renvoyé par next_token
%type String
//Indications lorsqu'on atteint la fin du fichier
%eofval{
   return null;
%eofval}
%eofclose
```

Analyse lexicale

```
// %cup permet de configurer les 4 paramètres ci-dessus pour la compatibilité avec Cup

// Définition des macros
CHAINE_CAR      = " "|"!"|[\043-\176]
CHAINE          = \"({CHAINE_CAR}|(\"\"))*\"
COMM_CAR        = \t|[\040-\176]
COMM            = "--"{COMM_CAR}*

%%
// -------------------------
// Début de la partie "règles"
// -------------------------

{COMM}   { System.out.println(yytext());
           return yytext(); }
{CHAINE} { return yytext(); }
.|\n     { return yytext(); }
```

Analyse lexicale

```
/** La classe TestComment permet de tester l'analyseur
 * lexical défini dans la classe Comment. */
class TestComment {
  public static void main(String args[]){
    try {
      Commentaire com = new Commentaire(System.in);
      String s ;
      // Lecture de la première unité lexicale
      s = com.next_token() ;
      // Boucle principale
      while (s != null) {
          s = com.next_token() ;
      }
      System.out.println() ;
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Analyse lexicale
– Mise en oeuvre de l'analyse lexicale

- L'analyse lexicale est effectuée par la méthode Symbol_next_token()de la classe Lexical (fichier Lexical.java).

- Le type Symbol est défini par Cup (générateur d'analyse syntaxique), celui-ci utilise le fichier sym.java qui définit les différentes unités lexicales.

  Ce type permet de stocker des informations supplémentaires associées unités lexicales : valeur d'un entier ou d'un flottant, chaîne associée à un identificateur, numéro de ligne.

- Le rôle de l'analyse lexicale est donc de renvoyer le Symbol correspondant au lexème reconnu. De plus, l'exception ErreurLexicale doit être levée en cas de lexème non reconnu.

Analyse lexicale
– Mise en oeuvre de l'analyse lexicale

- **Exemples de règles**

```
[ \t]              { }
\n                 { }
{COMMENT}          { }
":="               { return symbol(sym.AFFECT}; }
".."               { return symbol(sym.DOUBLE_POINT); }
{ENTIER}           { try {
                         return symbol(sym.CONST_ENT,
                                       new Integer(yytext());
                     } catch (NumberFormatException e) {
                           // Ici : Afficher message d'erreur
                          throw new ErreurLexicale();
                     }
                   }
```

Analyse lexicale
– Mise en oeuvre de l'analyse lexicale

- **Traitement des mots réservés dans l'analyse lexicale**
  - Une solution: On écrit une règle par mot réservé.

```
IF      { return symbol(sym.IF); }
WHILE   { return symbol(sym.while); }
```

JFlex crée alors un (gros) automate.

  - Autre solution:

On utilise un dictionnaire, qui stocke tous les mots réservés avec leur lexème correspondant.

Au départ : on initialise le dictionnaire avec tous les mots réservés.

On reconnaît alors les mots réservés comme des identificateurs et on consulte ensuite le dictionnaire pour savoir s'il s'agit d'un mot réservé.

```
private final Hashtable<String, Integer> dictionnaire =  initialiserDictionnaire();
```

Analyse lexicale

- **Travail à effectuer pour l'analyse lexicale**
  - Dans le répertoire ProjetCompil/Syntaxe/Src, compléter le fichier lexical.flex
  - Pour tester, utiliser le script lexico dans le répertoire ProjetCompil/Syntaxe/Test.

Plan

- **Analyse lexicale**
  - JFlex
  - Mise en oeuvre de l'analyse lexicale
  - Travail à effectuer
- **Analyse syntaxique**
  - Les arbres
  - Cup
  - Principe de l'analyse syntaxique
  - Travail à effectuer

Analyse syntaxique – Les arbres

a) Syntaxte abstraite (ArbreAbstrait.txt page 11)

- Syntaxe abstraite du langage JCas

  Définit la représentation intermédiaire utilisée par les compilateurs (ou interprètes) du langage. Cette syntaxe abstraite est définie par une grammaire d'arbres.

Analyse syntaxique – Les arbres

- Exercice

  Donner les arbres abstraits correspondant aux programmes JCas suivants :

```
program
   a  : integer  ;
   b, c : boolean ;
begin
   a := 1 ;
   b := a + 1 ;
end.
```

```
program
   t : array[1..10] of array[1..5] of integer ;
begin
   t[10][5] := 0 ;
end.
```

Analyse syntaxique – Les arbres

b) Implémentation des arbres

- Type énuméré Noeud.Constantes : Noeud.Affect, Noeud.Ident, Noeud.Entier, Noeud.Plus …
- Classe Arbre
- Constructeurs, préfixés par “creation ” :

```
static Arbre creation0(Noeud noeud, int numLigne)  ;
static Arbre creation1
 (Noeud noeud, Arbre fils1, int numLigne)  ;
static Arbre creation2
 (Noeud noeud, Arbre fils1, Arbre fils2, int numLigne);
static Arbre creationEntier(int valEntier, int numLigne);
```

- Exemples :

```
Arbre a1 = Arbre.creationIdent(C, numLigne)  ;
Arbre a2 = Arbre.creationEntier(4, numLigne)  ;
Arbre a3 = Arbre.creation2(Noeud.Plus, a1, a2, numLigne)  ;
```

Analyse syntaxique – Les arbres

b) Implémentation des arbres (suite)

- Sélecteurs, qui permettent de décomposer un arbre. Les sélecteurs sont préfixés par `get`.
  - Exemple

```
a2.getEntier() → 4
a3.getFils1()  → a1
a3.getFils2()  → a2
```

- Des mutateurs, qui permettent de modifier un arbre. Les mutateurs sont préfixés par `set`.
  - Exemple

```
a3.setFils1(Arbre.creationEntier(3, numLigne)  ;
```

- Décors, qui serviront lors de la passe 2.

Analyse syntaxique – Les arbres

b) Implémentation des arbres (suite)

- Méthodes d'affichage.

```
void afficher(int niveau);
```

- permet d'afficher un arbre, avec un certain niveau de détail pour les décors. Si niveau = 0, les décors ne sont pas affichés.

```
void decompiler(int niveau);
```

- Permet de “décompiler” un arbre abstrait, c'est-à-dire d'afficher le programme JCas correspondant.

Analyse syntaxique – Les arbres

c) Implémentation des arbres, “sémantique de partage”

- Un arbre est un pointeur, ce qui implique qu'on a une “sémantique de partage”.
- Exemple:

```
Arbre a = Arbre.creation2(Noeud.Plus,
                  Arbre.creationEntier(1, n),
                  Arbre.creationEntier(2, n), n);
Arbre b = a;
b.setFils1(Arbre.creationEntier(4, n));
a.getFils1() → Noeud.Entier(4)
```

Donc la modification de l'arbre b a modifié l'arbre a.

- **L'affectation entre deux arbres est une affectation entre <u>pointeurs</u>.**

Analyse syntaxique – Les arbres

```
Arbre c = Arbre.creation2(Noeud.Plus,
                Arbre.creationEntier(4, n),
                Arbre.creationEntier(2, n), n)  ;

a == b → true
a == c → false
```

- **L'égalité entre deux arbres est une égalité entre <u>pointeurs</u> (et non une égalité structurelle).**
- La plupart des types abstraits utilisés dans le projet sont implémentés à l'aide de pointeurs. Pour tous ces types, on a donc une “sémantique de partage”.

Analyse syntaxique
– Cup

a) Introduction

- **Cup**: générateur d'analyseurs syntaxiques pour Java, dans la lignée de Yacc, générateur d'analyseurs syntaxiques pour le langage C.
- Fichier `syntaxe.cup`, qui contient une grammaire hors-contexte, accompagnée d'un ensemble d'actions Java.
- Cup produit deux fichiers :
  - `sym.java` , qui contient les terminaux de la grammaires ;
  - `parser.java` , qui contient la classe `parser` et la méthode principale (héritée) `parse()` .

`parse()` réalise l'analyse syntaxique et effectue les actions associées aux règle de la grammaire.

Analyse syntaxique

a) Structure du fichier syntaxe.cup

Correspond à Syntaxe.txt avec en plus :

- Code à inclure dans la classe parser.
- Définition du type de l'attribut retourné par certains terminaux. (String pour IDF, Integer pour CONST_ENT...)
- Définition des non terminaux, avec le type de l'attribut retourné par chaque non terminal. Dans le cadre du projet : Arbre.
- Règles de la grammaire et actions Java associées (qui permettent de construire l'arbre abstrait du programme).

Analyse syntaxique
– Principe de l'analyse syntaxique

- **Analyse ascendante**
  - On reconnaît des parties droites de règles et on essaie de remonter vers l'axiome de la grammaire.
  - Exemple 1.

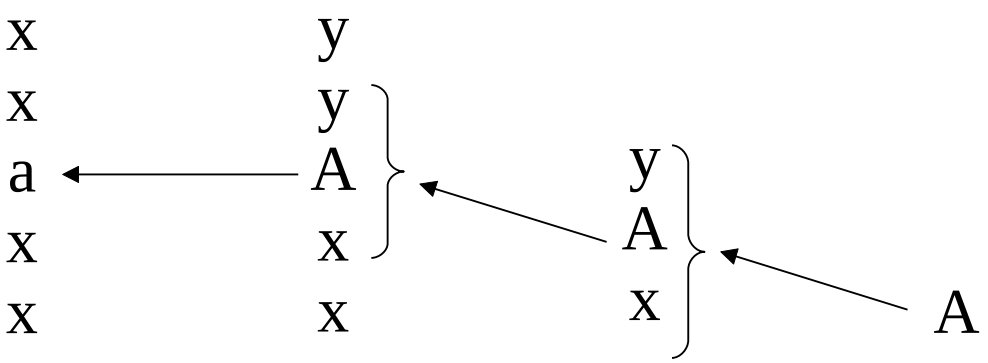
    Grammaire pour le langage

```
A ::= x A y      {: afficher("A -> x A y");  :}
  | a            {: afficher("A -> a") ;     :}
```

    On considère la chaîne d'entrée : **x x a y y**

    On a des “empilements” et des “réductions”.

Analyse syntaxique



Les flèches correspondent à des “réductions”.

Lorsqu'on réduit, l'action correspondante est effectuée.

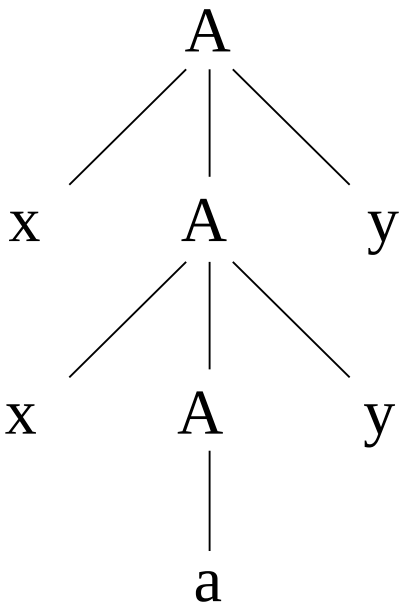
On affiche donc:

```
A -> a
A -> x A y
A -> x A y
```

La chaîne est reconnue si, après avoir empilé tous les caractères et effectué toutes les réductions, la pile ne contient que l'axiome de la grammaire.

Analyse syntaxique

Arbre de dérivation



Analyse ascendante : on effectue d'abord les règle associées aux feuilles de l'arbre de dérivation, puis on remonte vers l'axiome.

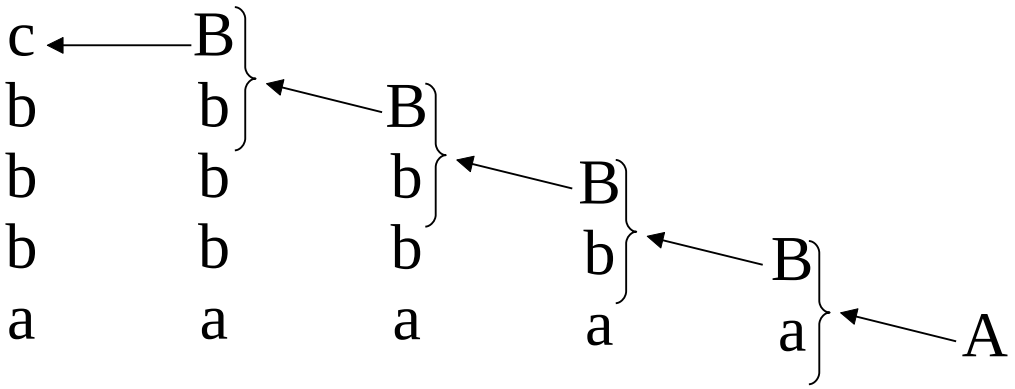
Analyse syntaxique

- Exemple 2.

  Soit le langage a b* c, engendré par la grammaire :

```
A ::= a B
B ::= b B | c
```

  On considère la chaîne d'entrée : **a b b b c**



  On doit empiler toute la chaîne d'entrée avant de commencer à réduire.

Analyse syntaxique

Problème:

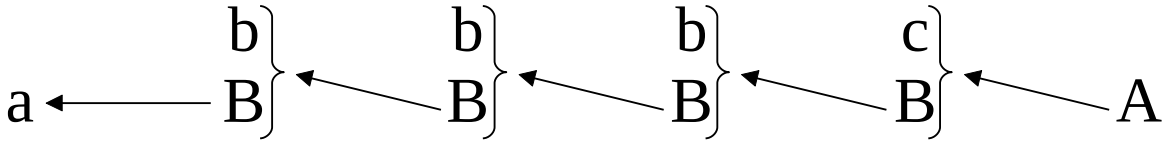
La pile utilisée par Cup a une taille limitée; donc si la chaîne d'entrée est très longue, la pile déborde.

La grammaire est récursive à droite.

On peut reconnaître le même langage avec une grammaire récursive à gauche :

```
A ::= B c
B ::= B b | a
```

On reprend la même chaîne d'entrée : **a b b b c**



Les réductions ont lieu au fur et à mesure des empilements.

Analyse syntaxique

- Conclusion

  En analyse ascendante, on utilise de préférence des règles récursives à gauche (pour des raisons de place mémoire).

  Par exemple, dans le projet, on a la règle :

```
liste_inst  ::= liste_inst inst POINT_VIRGULE
              | inst POINT_VIRGULE
```

  Remarque : en analyse descendante, on doit avoir des règles récursives à droite et non à gauche.

- Attributs
  - A chaque terminal et non terminal de la grammaire est associé un attribut synthétisé (c'est-à-dire un attribut transmis du fils vers le père dans l'arbre de dérivation).
  - Pour les non-terminaux, l'attribut est un arbre.
  - Ces valeurs permettent de construire l'arbre abstrait associé au programme.

Analyse syntaxique

- **Exemple de règles**

```
program  ::=
   PROGRAM:p liste_decl:a1 BEGIN liste_inst:a2 END POINT
{: RESULT =
      Arbre.creation2(Noeud.Programme, a1, a2, pleft);
:}
// pleft : Numéro de ligne du mot réservé program.
;

liste_decl  ::= liste_decl  :a1 decl  :a2 POINT_VIRGULE
{: RESULT =
      Arbre.creation2(Noeud.ListeDecl, a1, a2,
                        a2.getNumLigne());
:}
| // epsilon
{: RESULT =
      Arbre.creation0(Noeud.Vide, parser.numLigne());
:}
;
```

Analyse syntaxique

- **Méthode principale de l'analyseur syntaxique**

```
public static Arbre analyseSyntaxique(String[] args)
        throws Exception, ErreurLexicale, ErreurSyntaxe {

    // On récupère le fichier à analyser
    InputStream fichierCas = ArgsFichier.ouvrir(args);

    // Création de l'analyseur lexical
    Lexical analyseurLex = new Lexical(fichierCas);

    // Création de l'analyseur syntaxique
    parser analyseurSynt = new parser(analyseurLex);

    // Appel de l'analyseur syntaxique
    Object result = analyseurSynt.parse().value;
    Arbre arbre = (Arbre) result;

    // On retourne l'arbre abstrait construit
    return arbre;
}
```

Analyse syntaxique

- **Travail à effectuer pour l'analyse syntaxique**
  - Compléter le fichier syntaxe.cup
  - Ecrire des programmes JCas de test dans ProjetCompil/Syntaxe/Test.
  - Effectuer des tests avec le script syntaxe dans le répertoire ProjetCompil/Syntaxe/Test

    Ce script utilise le programme principal présent dans la classe TestSynt.