MINISTERE DE L'ENSEIGNEMENT SUPERIEUR
ET DE LA RECHERCHE SCIENTIFIQUE
BADJI MOKHTAR-ANNABA UNIVERSITY
UNIVERSITE BADJI MOKHTAR-ANNABA

وزارة التعليم العالي و البحث العلمي
جامعة باجي مختار – عنابـــــــــة

FACULTE DES SCIENCES DE L'INGENIORAT
DEPARTEMENT D'INFORMATIQUE

كلية علـــــــوم الهندســـــة
قسم الإعـــــلام الألـــــــي

# COMPILATION

3ème Année Licence en Informatique

## BROCHURE DE TRAVAUX PRATIQUES

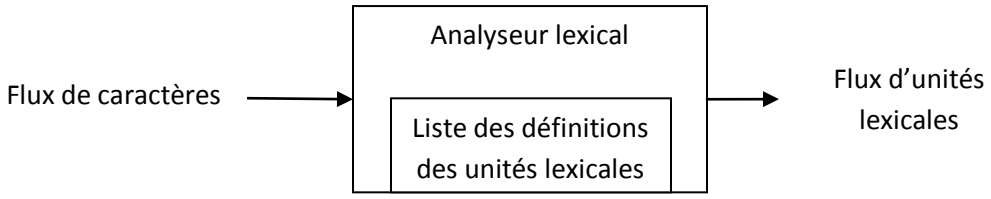
REALISEE PAR
DR BENOUHIBA TOUFIK

2013-2014

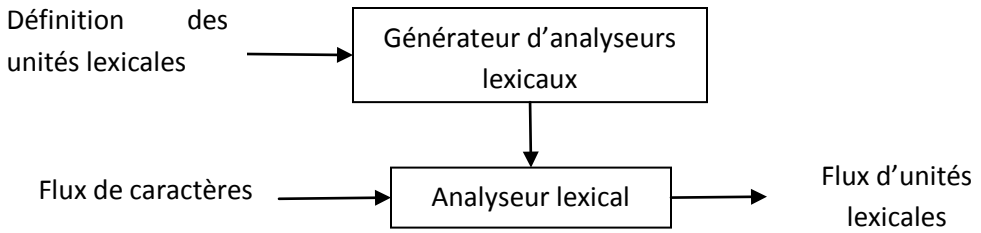
# 1. Introduction à l'outil JFlex

## 1.1. Principe des générateurs d'analyseurs lexicaux

D'une manière générale, le fonctionnement des analyseurs lexicaux des langages est, presque toujours, le même. Il s'agit d'analyser un flux de caractères (simples ou en *Unicode*) afin d'y détecter les unités lexicales définies par le langage. La sortie est alors un flux d'unités lexicales.




Afin de simplifier la construction des analyseurs lexicaux, et compte tenu de la généricité de leur fonctionnement, plusieurs outils ont été mis en œuvre pour les générer automatiquement. En effet, il suffit juste de fournir la définition des différentes unités à ces outils pour obtenir le code source d'un programme qui peut analyser un flux de caractères et y détecter les unités en question.




L'avantage d'une telle approche est de faciliter la construction des analyseurs lexicaux. Il suffit alors de fournir les définitions exactes des unités lexicales du langage et le générateur fera le reste en générant le code source de l'analyseur.

## 1.2. L'outil JFlex

### 1.2.1. Introduction

Plusieurs générateurs d'analyseurs lexicaux ont été conçus et largement diffusés. Historiquement, l'outil « lex » était le premier outil à être conçu, il est de ce fait, la base de la plupart des autres générateurs (lex est une abréviation du mot *lexer* : analyseur lexical).

L'outil lex a été conçu pour les systèmes UNIX et permet de générer du code source en langage C. Il est également propriétaire, ce qui signifie que les développeurs n'ont pas accès à son code source et ne peuvent pas le modifier. Une alternative *open source* a été proposée, il s'agit de Flex (Fast lex), un outil qui génère également du code en C (et même en C++).

Avec l'avènement du langage Java, plusieurs développeurs se sont intéressés à écrire des générateurs permettant de générer des analyseurs écrits en ce langage. Ainsi, on a vu naître des outils comme Javacc, Jlex et JFlex. Cette brochure s'intéresse à ce dernier (JFlex est une réécriture de JLex). L'outil ainsi que toute la documentation nécessaire sont à télécharger depuis le site : http://jflex.de

### 1.2.2. Utilisation

L'outil doit d'abord être téléchargé à l'adresse indiquée ci-haut. Après installation, il est prêt à être utilisé. L'outil JFlex se présente sous la forme d'un fichier JAR exécutable. On peut le lancer à partir de la ligne de commande en utilisant la commande :

```
java -jar JFlex.jar
```

On peut également le lancer en allant dans le répertoire bin de l'installation avec la commande suivante :

```
jflex(.bat) fichier_specfication
```

fichier_specification représente le nom d'un fichier contenant la définition des unités lexicales ainsi que quelques petites instructions en Java. Il peut également être rajouté à un projet Eclipse et être lancé depuis. Enfin, si on double-clique dur le fichier JFlex.jar, situé dans le dossier *lib* de l'installation, une interface graphique s'affiche permettant ainsi d'utiliser JFlex aisément.

Lorsque la génération de code se fait avec succès, un fichier d'extension .java est produit. Il s'agit du code source de l'analyseur lexical. Il peut alors être compilé et utilisé comme bon il nous semble. Des statistiques, concernant les automates utilisés, sont également fournies.

### 1.2.3. Structure du fichier de spécification

Une spécification JFlex est divisée en trois parties séparées par les caractères %%. Sa structure générale est définie par :

```
Code utilisateur
%%
Options et déclarations
%%
Règles lexicales
```

Les lignes séparant les parties doivent commencer par %%. Dans toutes, les parties, il est possible d'utiliser des commentaires comme ceux utilisés en Java (/* et */ ainsi que //), ceci est particulièrement utile pour documenter le fichier de spécification (surtout lorsqu'il s'agit d'un fichier complexe).

#### *Le code utilisateur*

Cette partie contient un code Java qui va être copié tel quel dans le code de l'analyseur avant toute autre chose. En d'autres termes, tout ce qu'écrirait l'utilisateur dans cette partie serait copié au début du fichier contenant le code source de l'analyseur.

Bien que l'on peut y placer n'importe quel code Java, il est d'usage de mettre dans cette partie les instructions relatives au package et aux importations de bibliothèques (les mots clés Java package et import). Par exemple, si la classe de l'analyseur doit être définie dans la package « langage.lexical » et qu'elle a besoin du package « java.util.* » alors cette section comportera :

```
package langage.lexical ;

import java.util.* ;
```

#### *Options et déclarations*

Cette partie contient des déclarations d'options permettant de personnaliser l'analyseur généré. On peut également y déclarer des états lexicaux (voir plus bas) ainsi que la définition de macros pouvant être utilisées dans la troisième partie (voir plus bas).

Chaque option doit commencer au début d'une nouvelle ligne, son premier caractère est obligatoirement %. Parmi les options les plus utilisées, on trouve l'option %class permettant de définir le nom de la classe représentant l'analyseur lexical. Par exemple :

```
%class Lexer
```

permet d'avoir une classe appelée Lexer comme analyseur lexical.

Les états lexicaux permettent d'analyser plusieurs langages en une seule spécification. L'idée est de d'associer un état à chaque langage et de définir un ensemble de règles lexicales pour chaque état. Chaque fois que l'on veut passer d'un langage à un autre, il suffit de passer à l'état associé (voir plus

bas). Par défaut, il y a un seul état qui s'appelle YYINITIAL, on peut déclarer d'autres états grâce à la syntaxe : « %state nom_état ».

On peut également utiliser l'option spéciale qui commence par %{ et se termine par %}. Entre ces deux délimiteurs, on peut mettre n'importe quel code Java, il sera copié tel quel à l'intérieur de la classe générée (la classe de l'analyseur lexical). Exemple :

```
%{
private int count ;
private Vector<String>list=new Vector<String>() ;
%}
```

Cet exemple copie les deux déclarations dans la classe générée.

Une autre possibilité très utile est de définir des macros. Une macro n'est autre que la définition d'une expression régulière et lui affecter un nom. La forme générale d'une macro est « Nom=Expression ». Nous verrons plus tard la syntaxe des expressions régulières. Exemple :

```
SuiteA=a*
```

Cette déclaration définit une macro de nom « SuiteA » qui est égale à « a* ».

D'autres options seront expliquées à la fin de cette partie.

### *Règles lexicales*

Les règles lexicales représentent le moyen permettant de définir les unités lexicales du langage. Leur forme est simple :

<ETAT> Expression_regulière { code_java }

Tel que <ETAT> représente l'état l'analyseur dans lequel on peut utiliser la règle (**il peut être omis si on s'intéresse à l'état YYINITIAL**), Expression_régulière représente une expression régulière décrivant la forme générale de l'unité lexicale (par exemple : les entiers sont représentés par [0-9]+), code_java représente des instructions Java à appliquer lorsque l'analyseur détecte l'unité lexicale en question.

### *Comment fonctionne l'analyseur ?*

L'analyseur généré examine le texte en entrée ainsi que l'ensemble des règles correspondant à son état. Il essaie d'établir une correspondance entre les règles et le texte de longueur maximale. Par exemple, si l'analyseur a comme entrée le texte : « aaab » et qu'il dispose de trois règles dont les expressions régulières sont « aa », « a+ » et « a+b », alors il choisira la dernière vu **qu'elle permet de récupérer le maximum de caractères** (la première règle récupère aa, la deuxième récupère aaa et la troisième récupère aaab).

Dans le cas où l'analyseur trouve deux règles permettant de récupérer le même texte, alors il choisira celle qui est la première déclarée.

Si, en revanche, aucune règle ne correspond au texte en cours d'analyse, une erreur est signalée et l'analyse est tout simplement arrêtée à moins d'utiliser l'option %standalone (attention, cette option ne fait pas cela uniquement).

### *Exemple complet*

```
%%
%class Lexer
%int
entier = [0-9]+
binaire = [01]+
%{
  int countE=0,countB=0 ;
%}
%%
{binaire} {countB++ ;}
{entier} {countE++ ;}
.|\n {}
```

Le code généré sera utilisé par la classe suivante :

```
public class Test{
      public static void main(String...args) throws Exception{
            Lexer lex=new Lexer(System.in);
            lex.yylex();
            System.out.println(lex.countE);
            System.out.println(lex.countB);
      }
}
```

**Explication :**

L'option %int permet d'indiquer que la fonction de lancement de l'analyse (yylex()) est de type entier (int). Deux macros ont été définies, l'une pour les entiers décimaux, l'autre pour les binaires. Le code définit deux variables permettant de calculer les nombres des entiers et des binaires. La règle « .|\n » correspond à n'importe quel texte qui ne soit pas entier ni binaire. L'action correspondante consiste à ne rien faire (ce qui signifie que l'on ignore tout simplement ce texte).

Il est à noter que le constructeur de la classe Lexer prend un paramètre de type InputStream ou Reader (par exemple, on peut utiliser la classe FileReader). Ici, on lui passe l'entrée standard du système (le clavier généralement).

Il est à noter que l'ordre des règles dans cet exemple est important. Qu'est ce qui se passe si on intervertit les deux premières règles ?

### 1.2.4. Les expressions régulières

Une expression régulière est un moyen pratique pour décrire les langages réguliers. En JFlex, elles permettent de décrire les unités lexicales du langage. Si on s'intéresse, par exemple, au mot clé « begin » en Pascal, alors il suffit de l'écrire tel quel ou entre deux guillemets "". Il existe plusieurs façons pour les construire, le tableau suivant les explique sommairement :

| **Constructeur** | **Signification** |
|---|---|
| [abc] | Correspond à l'un des caractères a, b ou c |
| [a-z] | Correspond à l'un des caractères compris entre a et z (selon le jeu ASCII) |
| [^abc] | Aucun caractère dans l'ensemble {a,b,c} |
| "TEXTE" | Correspond à TEXTE même s'il contient des caractères spéciaux |
| {macro} | Reprend la définition de la macro dont le nom est placé entre les deux accolades |
| \n, \t | Saut de ligne, tabulation |
| \c | Correspond au caractère c (lorsqu'il est spécial comme +,*, ., etc.) |
| . | Correspond à n'importe quel caractère sauf \n |
| r \| s | Correspond à r ou s |
| rs | Correspond à la concaténation de r et s |
| r* | Correspond à la répétition de r zéro ou plusieurs fois |
| r+ | Correspond à la répétition de r une ou plusieurs fois |
| r? | Correspond à la répétition de r zéro ou une fois (r est optionnel) |
| ^r | Correspond à une ligne commençant par r |
| r$ | Correspond à une ligne se terminant par r |
| ~r | Correspond à tout ce qui précède r |
| r{n} | r répété n fois |
| r{n,m} | r répété entre n et m fois |
| (r) | Identique à r |
| <<EOF>> | Correspond à la fin du fichier |

### 1.2.5. Code java des actions associées aux règles lexicales

Ce code peut être n'importe quelles instructions Java valides. Attention, JFlex n'est pas un compilateur du langage Java, si la spécification contient des erreurs Java, elles ne seront pas détectées lors de la génération.

A tout moment, le code Java inséré a le droit d'utiliser entre autres les méthodes et variables suivantes :

- yylex() : c'est la méthodes qui permet de lancer l'analyse lexicale. D'habitude elle n'est pas appelée à partir des règles lexicales.

- yytext() : cette chaîne de caractère est de type String. Lorsque l'analyseur établit la correspondance entre le texte en entrée et une des règles lexicales, cette fonction renvoie le texte en question.

- yylength() : cette fonction renvoie la longueur du texte récupéré par yytext(). Une autre possibilité est d'utiliser yytext().length()

- yyclose() : cette fonction ferme le fichier en cours d'analyse.

- yyreset(java.io.Reader reader) : cette fonction ferme le fichier en cours et réinitialise l'analyse de sorte à utiliser le fichier passé comme paramètre comme source.

- yystate() : retourne l'état en cours de l'analyseur

- yybegin(int *state*) : change l'état de l'analyseur à *state*.

- yyline : représente la ligne actuelle dans le fichier analysé (disponible avec l'option %line).

- yycolumn : représente la colonne actuelle dans le fichier analysé (disponible avec l'option %column).

### 1.2.6. Options d'analyse

Plusieurs options sont disponibles. Elles permettent de personnaliser le fonctionnement de l'analyseur généré. Nous donnons ici la liste des options les plus utiles :

- %class nom_class : définit le nom de la classe de l'analyseur lexical

- %public : la classe de l'analyseur sera publique

- %function nom_fonction : définit le nom de la méthode permettant de lancer l'analyse à nom_fonction. Par défaut, ce nom est yylex().

- %int : indique que les méthodes d'analyse générées sont de type entier.

- %type nom_type : indique que les méthodes d'analyse sont de type nom_type.

- %eofval{ et %eofval} permettent de définir un code Java à exécuter chaque fois que l'on arrive à la fin du fichier analysé (on peut arriver à la même fin avec <<EOF>> ).

- %cup : indique que l'analyseur généré est compatible avec le générateur d'analyseurs syntaxiques CUP.

- %byacc : indique que l'analyseur généré est compatible avec le générateur d'analyseurs syntaxiques BYacc/J.

- %line : la variable yyline est disponible.

- %column : la variable yycolumn est disponible.

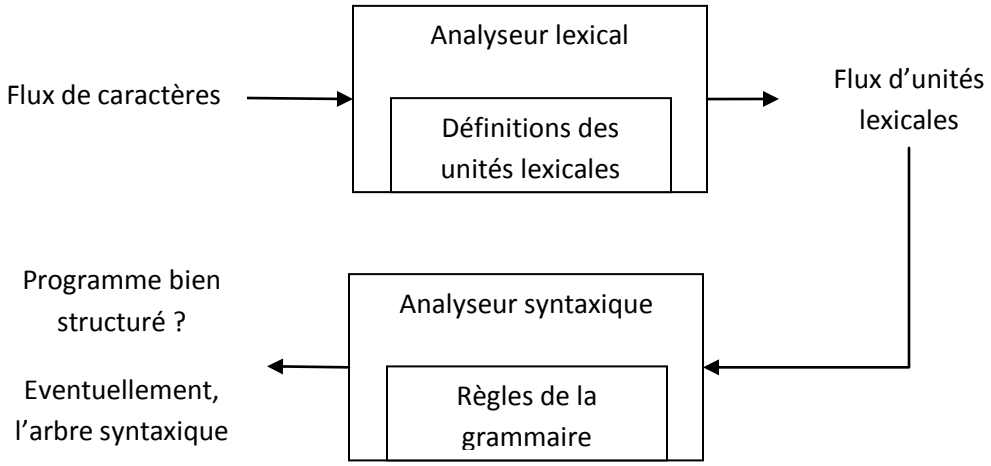
## 1.3. Conclusion

Cette brochure n'est qu'un aperçu sur l'outil JFlex, il n'est en aucun cas un substitut du manuel officiel de JFlex. Les étudiants sont fortement recommandés de le consulter à l'adresse http://jflex.de/manual.pdf pour des informations plus détaillées concernant la syntaxe, les options et le fonctionnement de JFlex.

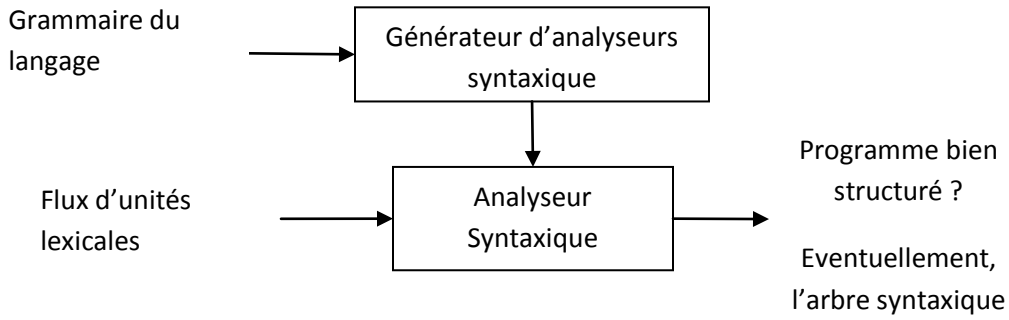
# 2. Introduction à l'outil BYacc/J

## 2.1. Principe des générateurs d'analyseurs syntaxiques

Tout comme les analyseurs lexicaux, le fonctionnement des analyseurs syntaxiques présente une certaine généricité. En effet, un analyseur syntaxique prend en entrée un flux d'unités lexicales (générées par l'analyseur lexical) et vérifie si elles sont bien structurées selon la grammaire du langage.




La construction des analyseurs syntaxique suit des méthodes et algorithmes bien définis. Parmi ces méthodes, on trouve la méthode LALR(1) qualifiée de méthode ascendante (construction de l'arbre syntaxique à partir des feuilles jusqu'à la racine). Une fois la grammaire du langage définie, la méthode LALR(1) construit une table d'analyse que l'on peut utiliser chaque fois que l'on veut analyser un programme écrit dans ce langage. Afin, de simplifier la construction des analyseurs syntaxiques, plusieurs outils ont été conçus afin d'automatiser cette opération. De tels outils ont juste besoin de la grammaire du langage.




Selon cette figure, l'analyseur syntaxique a besoin de coopérer avec l'analyseur lexical afin de mener à bien sa tâche. Il est à noter que la sortie principale de l'analyse syntaxique (sortie par défaut) est une réponse à la question « est-ce que le programme est bien structuré ? » On doit rajouter des actions pour pouvoir construire un arbre syntaxique et effectuer une analyse sémantique du programme.

## 2.2. L'outil BYacc/J

L'un des premiers générateurs d'analyseurs syntaxiques ayant été mis en œuvre est l'outil « Yacc » (Yet another compiler compiler). Il est généralement utilisé conjointement avec le générateur d'analyseurs lexicaux « lex ». Tout comme ce dernier, Yacc a été conçu pour les systèmes UNIX et permet de générer du code source en langage C. Il est également propriétaire. Des alternatives open source ont donc été proposées dont le fameux outil « Bison ».

L'outil BYacc/J est une implémentation open source de l'outil Yacc capable de générer des analyseurs écrits en Java. Il n'est pas le seul à pouvoir faire cela, il existe également un autre outil appelé CUP. Nous préférons ici utiliser l'outil BYacc/J vu qu'il est assez proche des outils Yacc/Bison. Il est toutefois à noter que l'outil JFlex supporte bien les deux outils BYacc/J et CUP. Enfin, BYacc/J peut également générer des analyseurs écrits en langage C.

L'outil BYacc/J est téléchargeable sur le site http://BYacc/J.sourceforge.net/. La documentation présentée sur ce site est malheureusement très peu abondante, on peut alors utiliser n'importe quelle documentation de Yacc ou de Bison.

### 2.2.1. Utilisation

L'outil BYacc/J se présente sous la forme d'un seul fichier exécutable (yacc.exe). Pour le lancer, il suffit de lancer la commande :

```
yacc -J fichier_specification
```

fichier_specification représente le nom d'un fichier contenant la définition de la grammaire du langage ainsi que du code Java dans le cas où l'on veut faire de l'analyse sémantique ou même une traduction dirigée par la syntaxe. L'option –J indique à Byacc/J de générer du code Java.

Lorsque la génération de code se fait avec succès, un fichier appelé **Parser.java** est généré (entre autres). Il s'agit du code source de l'analyseur syntaxique qu'il faudra compiler ensuite. Il faut également préciser qu'en cas de conflits (voir plus bas), des indications sont fournies en sortie. On peut également demander à BYacc/J de fournir une description d'un tas de détails concernant la construction de la table d'analyse. Ceci est particulièrement utile pour résoudre les problèmes de conflit.

**Remarque :** on peut changer le nom de la classe de l'analyseur syntaxique, il faut utiliser l'option :

```
-Jclass=nom_de_la_classe
```

On supposera, néanmoins, dans la suite que la classe de l'analyseur s'appelle Parser.

### 2.2.2. Modifications à apporter à la spécification JFlex

Pour pouvoir utiliser conjointement JFlex et BYacc/J, il faut modifier légèrement la spécification de JFlex comme suit :

- Rajouter l'option %byaccj dans la première section pour déclarer que l'analyseur lexical est compatible avec BYacc/J
- Rajouter un code permettant de spécifier ce que l'on fait lorsqu'on arrive à la fin du fichier. Par exemple, si on veut juste arrêter l'analyse et retourner 0, on écrira :

```
%eofval{
  return 0;
%eofval}
```

- Rajouter un attribut de type Parser ainsi qu'un constructeur de l'analyseur lexical afin qu'il puisse prendre un deuxième argument. On a alors à injecter le code suivant dans la première section

```
%{
      private Parser yyparser;

      public Lexer(java.io.Reader r, Parser yyparser) {
            this(r);
            this.yyparser = yyparser;
      }
%}
```

- Dans les règles lexicales, il faut rajouter des instructions pour renvoyer l'unité lexicale. Si on s'intéresse à une unité lexicale représentant un entier, alors la règle sera :

```
[0-9]+ {yyparser.yylval=new ParserVal(new Integer(yytext()));
        return Parser.INT;}
```

Tel que : l'attribut yylval représente la valeur de l'unité lexicale (on peut y stocker n'importe quel objet Java), INT est un token, en d'autres termes, c'est le nom d'une unité lexicale qui est déclaré dans le fichier de la spécification de l'analyseur syntaxique (voir plus bas). La classe ParserVal est généré par BYacc/J et permet d'encapsuler n'importe quelle unité lexicale.

- La fonction d'analyse doit être de type entier et renvoyer :
  - 0 si elle rencontre la fin du fichier
  - Une valeur négative en cas d'erreur
  - Un *token* dans les autres cas

### 2.2.3. Structure du fichier de spécification

Une spécification BYacc/J est devisée en trois parties séparées par les caractères %%. Sa structure générale est la suivante :

```
Déclarations
%%
Actions
%%
Code
```

#### *Les déclarations*

Cette partie permet d'inclure un code Java qui sera copié tel quel au début du fichier de l'analyseur. Il faudra le placer entre %{ et %}. On y place généralement les commandes de package et d'importation de bibliothèques. On peut également y déclarer des classes Java mais ceci n'est pas une bonne idée.

On peut déclarer, dans cette partie, les tokens : les noms des unités lexicales qui seront retournées par l'analyseur lexical. Ceci se fait grâce à l'option **%token**. Par exemple :

```
%token INT IDENT EQ
```

déclare que l'analyseur lexical peut renvoyer trois types d'unités lexicales, les constantes (de type entier) INT, IDENT et EQ sont alors accessibles par l'analyseur lexical.

On peut également déclarer le nom de l'axiome de la grammaire grâce à l'option **%start**. Par exemple :

```
%start Prog
```

déclare que l'axiome s'appelle Prog. Cette option n'est pas obligatoire, en son absence, le nom de l'axiome est, par défaut, **input**.

### *Les actions*

Cette partie comporte la définition de la grammaire du langage. Ceci est fait en définissant les règles de production comme suit :

```
Non_terminal :
       corps_1 {action sémantique 1}
      |corps_2 {action sémantique 2}
      |…
      |corps_n {action sémantique n}
      ;
```

Tel que corps_i représente une expression comportant des terminaux (tokens) et des non-terminaux, elle peut également être vide (le cas de ε). Les actions sémantiques représentent du code Java à exécuter lorsque la règle est activée (elles ne sont pas obligatoires). Par exemple :

```
input :
       IDENT EQ INT
      |IDENT EQ input
      ;
```

On peut bien évidemment définir autant de règles et de non-terminaux que l'on veut. Il faut juste que la grammaire soit LALR(1).

### *Le code*

La troisième partie de la spécification permet d'écrire du code Java qui sera copié tel quel à l'intérieur de la classe Parser. On y trouve généralement la déclaration d'un attribut correspondant à l'analyseur lexical ainsi qu'une fonction yylex (à ne pas confondre avec celle déclarée par l'analyseur lexical). Typiquement, on trouve au moins le code suivant :

```
private Lexer lex ;
private int yylex () {
      try{
```

```
            return lex.analyser();
      }
      catch(IOException e){
            System.err.println("Erreur E/S");
            return -1;
      }
}
public Parser(Reader r) {
    lexer = new Lexer(r, this);
}
```

La fonction yylex de Parser appelle la méthode « analyser » de l'analyseur lexical (dont la classe est supposée être Lexer ici). N'oublions pas que la méthode « analyser » sauvegarde dans l'attribut yylval l'unité lexicale rencontrée. Le constructeur de Parser crée un analyseur lexical en invoquant le constructeur de Lexer.

Il est également d'usage de rajouter une méthode spéciale appelée yyerror(String msg) qui sera appelée lorsqu'une erreur d'analyse est rencontrée. Typiquement, son code est :

```
public void yyerror (String error) {
    System.err.println (error);
}
```

### 2.2.4. Exemple complet

Commençons d'abord par la spécification de l'analyseur lexical :

```
%%
%class Lexer
%line
%column
%byaccj
%unicode
%function analyser
%eofval{
      return 0;
%eofval}

%{
      private Parser yyparser;
      public Lexer(java.io.Reader r, Parser yyparser) {
            this(r);
            this.yyparser = yyparser;
      }
%}
%%
```

```
[0-9]+ {yyparser.yylval=new ParserVal(new Integer(yytext())); return
Parser.INT;}
[a-z]+ {yyparser.yylval=new ParserVal(yytext()); return Parser.IDENT;}
= {return Parser.EQ ;}
.|\n {return -1 ;}
```

La spécification de l'analyseur syntaxique :

```
%{
import java.io.*;
%}
%token INT IDENT EQ
%%
input :
       IDENT EQ INT {}
      |IDENT EQ input {}
      ;
%%
private Lexer lex ;
private int yylex () {
      try{
            return lex.analyser();
      }
      catch(IOException e){
            System.err.println("Erreur E/S");
            return -1;
      }
}
public Parser(Reader r) {
    lex = new Lexer(r, this);
}
public void yyerror (String error) {
    System.err.println (error);
}
```

Enfin, le code source de classe du programme principal est :

```
public class Test{
      public static void main(String...args) throws Exception{
            Parser p=new Parser(new java.io.FileReader("text.txt"));
            p.yyparse();
            //yyparse est le nom de la méthode qui lance l'analyseur
            //syntaxique
      }
}
```

### 2.2.5. Gestion des erreurs

Par défaut, lorsque l'analyseur syntaxique rencontre une erreur, il s'arrête et affiche un message « Syntax error ». En plus du fait que ce message n'est pas très informatif, ce mode de fonctionnement ne permet pas de récupérer beaucoup d'erreurs (on est obligé de lancer la compilation autant de fois qu'il y a d'erreurs).

BYacc/J permet une gestion plus élaborée des erreurs grâce au token spécial **error**. L'idée est de mettre ce token dans les endroits où on attend à des erreurs de syntaxe. Par exemple, l'utilisateur peut oublier un signe =, ou identificateur ou entier. On utilise alors le code suivant :

```
 IDENT EQ INT
|IDENT EQ input
|IDENT error {yyerror("= est attendu");} input
|IDENT EQ error {yyerror("une constante ou un identificateur est attendu");}
|error {yyerror("Identificateur attendu");} EQ input
```

Le code Java inséré juste après le token error sert à afficher le message d'erreur, on peut y rajouter aussi l'incrémentation de la variable **yynerrs** qui indique le nombre d'erreurs rencontrées lors de la compilation.

### 2.2.6. Gestion des conflits

Les grammaires traitées par BYacc/J doivent être de type LALR(1). Ainsi, une grammaire ambiguë, par exemple, génère des conflits lors de la construction de la table d'analyse. Les grammaires LR(1) génèrent également des conflits.

Les conflits sont de deux types : shift/reduce ou reduce/reduce. BYacc/J résout ces conflits comme suit :

- Pour les conflits shifft/reduce : BYacc/J favorise l'opération shift (il préfère avancer dans l'analyse).
- Pour les conflits reduce/reduce : BYacc/J favorise la règle qui figure en premier dans le fichier de spécification.

En cas de conflits rencontrés, BYacc/J les et permet à l'utilisateur d'afficher une description de tous les itemsets construits afin qu'il puisse retrouver la source de conflit (l'option -v).

Malheureusement, les mécanismes de résolutions de conflits de BYacc/J peuvent ne pas correspondre à ce que souhaite faire l'utilisateur. Il faut alors modifier légèrement la spécification pour *imposer* ses règles de gestion de conflits.

Pour commencer, il faut dire que les conflits reduce/reduce sont généralement dus à une mauvaise conception de la grammaire. On peut les éviter en favorisant, par exemple, la récursivité à gauche (mais la solution dépend généralement de la grammaire).

Pour illustrer comment peut-on résoudre le problème des conflits shift/reduce, prenons cette nouvelle grammaire générant le même langage que les exemples précédents.

```
input : suite fin ;
suite :   IDENT
        | suite EQ suite ;
fin : EQ INT ;
```

Il s'agit ici d'une grammaire ambiguë car le programme « a=b=c=3 » possède deux arbres de dérivation. En effet, l'opérateur = peut être interprété comme a=((b=c)=3) ou encore comme (a=b)=(c=3). Le problème provient du fait que l'on ne connaît pas quel opérateur doit d'abord être considéré : le plus à gauche ou le plus à droite. Notez que lors de la génération du code source de l'analyseur, BYacc/J indique la présence d'un conflit shift/reduce et que la règle « suite : suite EQ suite ; » n'est jamais utilisée (vu que c'est le shift qui est privilégié »).

Pour éliminer ce problème, il suffit d'indiquer l'associativité de l'opérateur =. Par exemple, si on veut une associativité soit à gauche on écrira :

%left EQ

Pour une association à droite, on écrira :

%right EQ

Ce choix n'est pas possible pour cette grammaire vu qu'il mène à ignorer la règle « suite : suite EQ suite ; ».

Il faut aussi préciser que l'option %left ne sert pas seulement à préciser l'associativité à gauche de l'opérateur mais également sa priorité. En effet, lorsqu'on écrit « %left op1 op2 », alors on signifie que op1 et op2 ont la même priorité. Plus un opérateur figure loin lors de l'utilisation de l'option %left, plus il est prioritaire. Par exemple, si on écrit :

%left op1

%left op2

ceci signifie que op2 est plus prioritaire que op1.

Enfin, supposons qu'un opérateur op est unaire (il a besoin d'un seul opérande) et qu'il est le plus prioritaire. Il faut, alors, déclarer son associativité à la fin de la première section (par exemple, %left op) et ensuite placer **%prec op** à la fin de la production de l'opérateur.

### 2.2.7. Les actions sémantiques

Nous avons déjà vu des actions sémantiques lors de la gestion des erreurs. Une action sémantique revient donc à insérer du code Java quelque part dans la grammaire pour spécifier un traitement à effectuer lorsqu'une règle est utilisée.

Par code Java, on entend n'importe quelles instructions pouvant accéder librement à tous les attributs et les méthodes de la classe Parser. On a également le droit d'utiliser quelques variables spéciales relatives aux symboles utilisés dans la règle. Prenons l'exemple de la règle suivante :

E : F G H I

Le symbole à gauche, c'est-à-dire E, est référence par le symbole $$. Le premier symbole à droite, F, est référencé par $1, le symbole G par $2, etc. Toutes ces variables sont de type **ParseVal**, ainsi les attributs **ival, dval, sval et obj** est, entre autres, accessible. Avec les règles :

```
input : suite fin ;
suite :  IDENT {System.out.println($1.sval);}
          | suite EQ suite ;
fin : EQ INT {System.out.println($2.obj);} ;
```

on génère un analyseur qui affiche les noms de variables rencontrées ainsi que le dernier entier.

## 2.3. Conclusion

L'outil Byacc/J est un outil riche en options et en possibilités. Il permet d'écrire des compilateurs efficaces assez rapidement et de se concentrer sur les aspects grammaticaux et sémantiques du langage. Il est recommandé de consulter la documentation officielle de l'outil Yacc (ou celle de Bison) pour plus d'explications. Quelques exemples sont également disponibles sur le site http://byaccj.sourceforge.net.

# Série de TP 1 : Le langage Java

## Exercice 1 :

Ecrire un programme Java permettant de vérifier si une chaîne de caractères saisie correspond au format suivant : **Adresse_IP:port** où **Adresse_IP** représente une adresse IP valide et **port** représente un entier. Le programme doit ensuite extraire chaque partie et l'afficher à part.

## Exercice 2 :

Ecrire un programme Java permettant de vérifier la forme puis calculer la somme représentée par une chaîne saisie, par exemple : 23+45+12+2.

Afin d'éviter les opérations inutiles, il est demandé d'écrire un programme qui simplifie une somme contenant des zéros, par exemple 3+0+2+00+1+0 devient 3+2+1.

## Exercice 3 :

Un fichier contient les noms, les prénoms et les années de naissances des étudiants. Il est organisé comme suit : chaque ligne contient le nom, le prénom et l'année de naissance d'un étudiant, le tout séparé par des virgules (exemple : **benahmed,ahmed,1990**). Ecrire un programme qui permet de répondre à des questions du genre :

- Quels sont les étudiants qui sont nés avant une telle année ?

- Quels sont les étudiants dont le nom commence par **ben** (par exemple) ?

Le programme ne doit pas différencier les minuscules et les majuscules. Les lignes non conformes sont ignorées.

# Série de TP 2 : L'outil JFlex

## Exercice 1 :

Ecrire une spécification JFlex permettant de calculer le nombre de consonnes, de voyelles et de caractères de ponctuation dans un fichier.

## Exercice 2 :

Ecrire une spécification JFlex permettant de calculer la moyenne des entiers rencontrés dans un fichier.

## Exercice 3 :

Ecrire une spécification JFlex permettant de vérifier le bon parenthésage d'un fichier. On supposera que l'on n'utilise que la paire ().

## Exercice 4 :

Ecrire une spécification JFlex permettant de vérifier le bon parenthésage d'un fichier. On supposera que l'on utilise les paires (), [] et {}.

## Exercice 5 :

Ecrire une spécification JFlex permettant de calculer le résultat d'une expression arithmétique. On supposera que les opérateurs sont associatifs à gauche et qu'ils ont la même priorité.

Exemple : 34+12*2=(34+12)*2=46*2=92.

## Exercice 6 :

Ecrire une spécification JFlex permettant d'afficher un texte en utilisant les césures. Il s'agit d'afficher le texte sur un nombre de colonnes fixe (donné comme paramètre). Lorsqu'un mot ne peut être écrit entièrement sur la ligne, il est alors découpé en deux parties séparées par "-". La coupure doit néanmoins observer la règle suivante : les deux parties doivent avoir une longueur strictement supérieure à 1. On considère qu'un mot est composé de symboles différents de l'espace. Les suites d'espaces doivent être réduites à un seul. Enfin, on suppose que la longueur des mots est faible par rapport au nombre de colonnes.

Exemple : « Le langage Java est un langage orienté objet» sera écrit (avec un nombre de colonne égale à 15)

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| L | e | | l | a | n | g | a | g | e | | |
| J | a | v | a | | e | s | t | | u | n | |
| l | a | n | g | a | g | e | | o | r | i | - |
| e | n | t | e | | o | b | j | e | t | | |

## Exercice 7 :

Ecrire la spécification d'un analyseur lexical d'une partie simple de Pascal (program, déclarations de variables entières, mots clés principaux, affectation, test, boucle, expressions arithmétiques usuelles, opérateurs de comparaison).

# Série de TP 3 : L'outil BYacc/J

## Exercice 1 :

Ecrire une spécification Yacc pour le langage des expressions arithmétiques portant sur les nombres entiers et les opérateurs +, *, / et - ainsi que les parenthèses.

En donnant le sens usuel aux entiers et les opérateurs, rajouter les instructions permettant de calculer la valeur de l'expression.

## Exercice 2 :

Ecrire une spécification Yacc permettant d'analyser un programme écrit en Pascal simplifié. Rajouter les instructions permettant de construire la table des symboles et de tester si les variables utilisées sont déclarées.