PRESENTATION JAVACC

L'outil JAVACC

- Signification : Java compiler compiler
- Utilité : générer des analyseurs lexicaux et/ou syntaxiques écrits en JAVA; à partir d'une description d'un langage

Le kit JAVACC

- 3 exécutables indépendants : JAVACC, JJTree et JJDoc

Le principal :

- **JAVACC : génère le code source de l'analyseur correspondant à la grammaire définit dans le fichier .jj**

Utilitaires :

- JJTree : agit en tant que préprocesseur des grammaires JAVACC et permet d'insérer du code pour agir sur la création de l'arbre syntaxique lors de l'exécution du parseur. Il génère un fichier .jj
- JJDoc: génère une description de la grammaire de l'analyseur sous forma BNF dans un fichier texte ou HTML.

Structure d'un fichier source JAVACC

Quatre (4) parties :

- Zone des options
- Zone qui définit la classe principale de l'analyseur à générer délimitée par : PARSER_BEGIN et PARSER_END.
- Zone de la définition de l'analyseur Lexical :
    - TOKEN : ER des unités lexicales
    - MORE : est équivalent à yymore(), ce qui est reconnu est reporter en début du prochain TOKEN.
    - SKIP: ER des unités à ne pas reconnaitre (à ignorer)
    - SPECIAL_TOKEN : sert à conserver le contenu normalement généré dans un attribut du prochain TOKEN.
- Zone de la définition de la descente récursive (ensemble de règles de production): une procédure par non-terminal:

Expression régulière en JAVACC

– Comme LEX voici le langage des ER de JAVACC

Nom	Exemple	Description
Litéral	`"hello"`	Accepte seulement « a »
Classe de caractères	`["a","b","c"]`	Accepte un « a » un « b » ou un « c »
Plage de caractères	`["a"-"z"]`	Accepte tous les caractères de « a » à « z »
Négation	`~["a"]`	Accepte un caractère excepté « a »
Répétition	`("a"){4}`	Accepte quatre caractères « a »
Plage de répétition	`("a"){2,4}`	Accepte de deux à quatre caractère « a »
Un ou plus	`("a")+`	Accepte au moins un caractère « a »
Zéro ou un	`("a")?`	Accepte zéro ou un caractère « a »
Zéro ou plus	`("a")*`	Accepte tout occurrence d'un caractère « a » y compris aucune

Exemple du code JAVACC pour un analyseur Lexical seul

```
options {
  BUILD_PARSER=false;
}

PARSER_BEGIN(Literals)
public class Literals {}
PARSER_END(Literals)

TOKEN_MGR_DECLS: {
  public static void main(String[] args) {
    java.io.StringReader sr = new java.io.StringReader(args[0]);
    SimpleCharStream scs = new SimpleCharStream(sr);
    LiteralsTokenManager mgr = new LiteralsTokenManager(scs);
    for (Token t = mgr.getNextToken(); t.kind != EOF;
      t = mgr.getNextToken()) {
       debugStream.println("Found token:" + t.image);
    }
  }
}

TOKEN : {
  <IDUnitLex : “ExpRegUnitLex">
}
```

Zone options: ici on désactive la génération d’analyseur syntaxique

Zone Définissant l’analyseur Syntaxique : ICI VIDE

Zone Définissant l’analyseur Lexical : ICI SA CLASSE PRINCIPALE

Zone Définissant des unités lexicales de l’analyseur

Exemple du code JAVACC pour un analyseur Syntaxique

```
options {
  STATIC = false;
}

PARSER_BEGIN(PhoneParser)
import java.io.*;
public class PhoneParser {
  public static void main(String[] args) {
    Reader sr = new StringReader(args[0]);
    PhoneParser p = new PhoneParser(sr);
    try {
      p.PhoneNumber();
    } catch (ParseException pe) {
      pe.printStackTrace();
    }
  }
}
PARSER_END(PhoneParser)
TOKEN : {
  <FOUR_DIGITS : (<DIGITS>){4}>
  | <THREE_DIGITS : (<DIGITS>){3}>
  | <#DIGITS : ["0"-"9"]>
}
void PhoneNumber() : { } {
  <THREE_DIGITS> "-" <THREE_DIGITS> "-" <FOUR_DIGITS> <EOF>
}
```

Zone options: ici on désactive l'option STATIC de l'analyseur

Zone Définissant l'analyseur SYNTAXIQUE

Zone Définissant des unités lexicales de l'analyseur Syntaxique

Déclarations pour la partie programme

Zone Définissant les procédures LL(1) de l'analyseur Syntaxique