Bref Rappel de JavaCC

Avant-Propos

Ce document présente les idées de base de JavaCC, qui seront utiles pour l'implémentation d'une analyse syntaxique LL vue dans le cours et les TDs. Pour une documentation plus complète, veuillez vois référer au site de JavaCC (https://javacc.github.io/javacc/).

JavaCC permet d'écrire des analyseurs syntaxique (ou parseurs) LL(k) par la méthode de la descente récursive. En outre, JavaCC permet également de définir des analyseurs lexicaux (tokenizer) pour les entités (terminaux) de la grammaire considérée. JavaCC génère des fichiers sources en langage Java, qui doivent alors être compilés et exécutées.

1. Installation de JavaCC

Sur Linux : sudo apt-get javacc

Sur Windows :

Versions antérieures à 6.0 : à utiliser directement

Version 6.0 : copier les fichiers .bat (jjtree.bat, javacc.bat, jjdoc.bat) des versions antérieures dans le répertoire javacc/bin, puis exécution de javacc dans le répertoire bin.

Pour exécuter Javacc, il suffit de taper : **javacc <nom_fichier>**

2. Structure d'un programme JavaCC

Voici un squelette d'un programme JavaCC (Pseudo Code1) :

```
<Options de javacc>
PARSER_BEGIN ( <Nom du programme/fichier> )
<code Java>
PARSER_END ( <Nom du programme/fichier>)
 (production )*
 (tokenizer)*
```

Pseudo Code 1

Si un fichier JavaCC est écrit correctement, le compilateur JavaCC produit un ensemble de fichiers sources java. Si le fichier est nommé a.jj, la méthode principale d'exécution est le main du fichier source a.java.

2.1.Les options JavaCC

Cette partie est facultative, elle se présente comme suit (pseudo code 2):

```
options {
<nom_option1>=valeur ;
<nom_option2>=valeur ;
…….
}
```

Pseudo Code 2

A titre d'indication, nous présentons ci-dessous quelques-unes (tableau 1):

Option	Signification
LOOKAHEAD	Un entier représentant le nombre de tokens que le parser doit examiner avant de choisir la règle de production àappliquer, si la grammaire est LL(1), LOOKHEAD=1 (valeur par défaut)
STATIC	Booléen qui indique si les attributs et méthodes générés dans le fichier source seront déclarés comme étant static, **true** est sa valeur par défaut
DEBUG_PARSER	Booléen qui indique si le parseur généré affiche les informations de débogage à l'exécution, sa valeur par défaut est **false**
DEBUG_LOOKAHEAD	Même rôle que l'action précédente, avec en plus de l'affichage des séquences d'actions effectuées durant les lookheads, sa valeur par défaut est **false**
DEBUG_TOKEN_MANAGER	Booléen qui indique si le token manager affiche les informations de débogage à l'exécution, sa valeur par défaut est **false**
IGNORE_CASE	Booléen qui indique si le token manager doit ignorer la casse, sa valeur par défaut est **false**
BUILD_PARSER	Booléen qui indique si le fichier parser.java doit être généré, sa valeur par défaut est **true**
BUILD_TOKEN_MANAGER	Booléen qui indique si le fichier tokenmanager.java doit être généré, sa valeur par défaut est **true**
SANITY_CHECK	Booléen qui indique à JavaCC de détecter s'il y a ambiguité, récursivité gauche dans la grammaire, sa valeur par défaut est **true**

Tableau 1

2.2.Le code java

Dans cette portion (code 1) d'un programme JavaCC, on insère une méthode principale dont le rôle est de récupérer l'input et de lancer le parseur pour vérifier si la syntaxe de l'input est correcte. Pour pouvoir lancer l'analyse syntaxique, une instruction permet de spécifier explicitement l'axiome.

```
public class Example {
  public static void main(String args[]) throws ParseException {
    Example parser = new Example(System.in);//Lecture de l'input
    parser.S(); // S est l'axiome de la grammaire, lancement de l'analyse
syntaxique
  }}
```

Code 1

2.3.Productions (Parser)

Dans cette portion du squelette, le programmeur écrit les méthodes qui seront utilisées dans la descente récursive. Le parseur peut contenir des productions javacode et/ou des productions BNF.

2.3.1. Les productions javacode

Permet au programmeur d'écrire une méthode dans la pure syntaxe java, les productions javacode sont précédées par le mot-clé JAVACODE. Ce mot-clé indique au JavaCC de reprendre la méthode telle quelle dans le fichier source résultant (voir pseudo code 3).

```
JAVACODE

Public int A() { …..}
```

Pseudo Code 3

L'inconvénient de ces productions est que le programmeur doit gérer lui-même l'analyse syntaxique. Si pour une grammaire donnée, le programmeur choisit d'écrire des productions BNF et javacode, et si dans une production BNF donnée, il y a un choix à faire entre plusieurs productions javacode, cela posera un problème d'ambiguïté. Le programmeur doit être alors prudent et éviter ce scénario.

2.3.2. Les productions BNF

Les productions BNF sont beaucoup plus fréquemment utilisées que les productions JAVACODE, car elles permettent de déléguer les choix et la gestion de l'ambiguïté au JavaCC. Elles sont structurées comme suit (pseudo code 4):

```
<mode accès> <type retour> <identifiant> ( <liste des paramètres>) :

             Code java

             { <choix d'expansion> }
```

Pseudo Code 4

Une production BNF est une réécriture des règles de productions.

Exemple :

Soit la règle de production : **A → a B C b | d** (a b d étant des terminaux, B C des non-terminaux).
La production BNF sera écrite comme suit :
void A() :
{}
{
"a" B () C() "b" | " d"
}
Dans la suite du programme, les méthodes B() et C() doivent exister.
| est appelé choix d'expansion, c'est-à-dire que l'analyseur syntaxique doit choisir entre tenter de dériver "a" B () C() "b" ou " d".

2.4.Le Tokenizer

Identique au lex, il permet de définir des expressions régulières pour reconnaitre des entités lexicales.

2.4.1. Notation des expressions régulières

La syntaxe du tokenizer de JavaCC diffère légèrement de celle utilisée dans le lex (voir Tableau 2).

Nom	Example	Description
Littéral	"hello"	Accepte le mot « hello »
Classe de caractère	["a", "b", "c"]	Accepte un « a » ou un « b » ou un « c »
Plages de caractère	["a"-"z"]	Accepte les caractères allant de « a » à « z »
Négation	~["a"]	Accepte n'importe quel caractère excepté le « a »
Répétition	("a"){4}	Accepte 4 fois le caractère « a »
Plages de répétition	("a"){1,4}	Accepte un nombre de « a » compris entre 1 et 4
Un ou plus	("a")+	Accepte au moins une occurrence de « a »
Zéro ou plus	("a") ? ou [a]	Accepte au plus une occurrence de « a »
Zéro ou plus	("a")*	Accepte toute occurrence de « a », y compris aucune

Tableau 2

2.4.2. Les actions lexicales

Nous distinguons 4 actions lexicales: TOKEN, SKIP, MORE, SPECIAL_TOKEN.

La syntaxe est la suivante (pseudo code 5):

```
<Action_Lexical> {

<identifiant1>:  <expr_reg1>  |<identifiant2>:  <expr_reg2>  |  …..<identifiantn>:
<expr_regn>

}
```

Pseudo Code 5

Un programme JavaCC peut fournir plusieurs actions lexicales à effectuer, c'est-à-dire des expressions à reconnaitre, à ignorer…etc. Les identifiants mentionnés ci-dessous sont des appelations que le programmeur assigne aux expressions régulières correspondantes. Dans la portion du parseur, on peut utiliser les identifiants au lieu des expressions régulières pour une meilleure lisibilité du code. Les identifiants permettent aussi d'organiser les entités lexicales telles que les mots réservés, les variables…etc.

Le TOKEN crée une entité lexicale (ou terminal) à partir de la chaine reconnue et la retourne au parseur. Le SKIP ignore l'entité lexicale associée à l'expression régulière reconnu. Le MORE permet de reconnaitre un préfixe (représenté par l'expression régulière formulée) qui sera ajouté au prochain token. Le MORE est équivalent au SKIP, sauf que le MORE retourne une chaine qui sera concaténée au prochain token reconnu (exemple d'utilisation : les commentaires multilignes). SPECIAL_TOKEN indique que les chaines reconnues peuvent apparaitre dans n'importe quel emplacement du programme, c'est-à-dire au milieu de n'importe quelles deux entités lexicales.

3. Les points de choix (choice points) dans JavaCC

L'analyseur syntaxique produit par un programme JavaCC doit effectuer un choix à un moment donné au cours de l'analyse d'une chaine. Il existe 4 points de choix en JavaCC :

- Cas de l'expansion : se produit quand une règle de production possède de multiples MDPs (par exemple A → Exp1|Expr2), l'analyseur doit choisir quel MDP il va tenter de dériver, Expr1 ou Expr2.
- Cas de (Expr) ?: le parseur choisit entre dériver Expr ou dériver ce qui vient après Expr.
- Cas de (Expr)* : même chose que (Expr) ?, seulement dans ce cas, le parseur réitère le choix.
- Cas de (Expr)+ : même chose que (Expr)*, seulement dans cas, le parseur dérive Expr une première fois, puis effectue des choix pour les fois suivantes.

3.1.Les choix par défaut du parseur

Par défaut, le parseur examine le prochain token, puis décide de la règle à appliquer.

Si plusieurs règles sont applicables pour un token, le parseur affiche un conflit lors du choix, et suggère la modification de l'option *LOOKHEAD*. Si vous testez *Example3.jj* du répertoire d'installation *JavaCC/Examples/Lookhead*, vous obtenez un message d'erreur indiquant le conflit entre les *<ID> "(" expr() ")"* et *<ID> "." <ID>.*

Parfois, un conflit peut surgir entre des règles formulées dans des méthodes différentes (on appelle cela expansion imbriquée). Examinez et testez le programme JavaCC *Example5.jj* : lors de l'appel de *identifier_list*, après avoir reconnu un <*id*>, et si le prochain token est une virgule, on peut considérer que cette virgule fait partie de l'expression *("," <ID>)**, et donc appliquer cette règle, ou alors considérer *("," <ID>)** comme un mot vide, et donc enchainer sur *"," <INT>* de la méthode *funny_list*.

3.2.Spécification du Lookhead

Nous avons vu précédemment que des conflits peuvent surgir lorsque l'option *Lookhead=1* (valeur par défaut), par ailleurs, en cas de conflit, le compilateur JavaCC suggère de modifier la valeur du *Lookhead*.

Si après compilation d'un programme JavaCC, aucun avertissement n'est affiché, alors la grammaire écrite est LL(1). Sinon, le programmeur peut selon les cas :

- Modifier la grammaire pour le rendre LL(1), *Example6.jj* est une transformation de la grammaire du fichier *Example3.jj*,
- Modifier localement ou globalement la valeur de LOOKHEAD.

La première option permet de conserver les performances d'une analyse LL(1), mais n'est pas toujours réalisable.

3.2.1. Spécification globale de Lookhead

Se fait en modifiant la valeur de LOOKHEAD dans les options d'un programme JavaCC ou dans la ligne de commande. Par exemple, si LOOKHEAD est égale à 2, alors le parseur sera de type LL(2).

Il est fortement déconseillé de modifier globalement la valeur de LOOKHEAD afin de ne pas dégrader les performances de l'analyseur.

3.2.2. Spécification locale de Lookhead

Elle est généralement insérée là où un avertissement est signalé, elle indique au parseur, lors d'un point de choix, d'utiliser un nombre spécifique de tokens. Le fichier *Example8.jj* est une réécriture de *Example3.jj* avec un LOOKHEAD modifié localement, au niveau de la méthode *basic_expr*.