INSA RENNES

Quatrième année - Informatique

Projet 4ème année INFO

Rapport de spécification

Réalisé par :
Nour Romdhane
Liantsoa Rasata
Mathilde Leparquier
Ibrahim Benali
Othmane Kabir

Sous la direction de :
Jean-Louis Pazat

Table des matières

1 Introduction

Dans le cadre de notre deuxième année d'étude au département Informatique à l'INSA de Rennes, nous réalisons un projet technique en équipe. Le sujet de notre projet porte sur les mécanismes de synchronisation avancée en Java. Ainsi nous développerons des solutions pour faciliter l'apprentissage des outils de synchronisation souvent fastidieux pour les programmeurs.

Après une première phase d'analyse de notre projet, nous détaillons dans ce rapport les spécifications fonctionnelles ainsi que son architecture générale.

Pour cela nous expliquons, dans ce rapport, la technique envisagée pour la réalisation de notre projet. Ainsi nous décrivons son fonctionnement, notre but étant de transformer un code simple avec des compteurs de synchronisation à un code utilisant les outils de synchronisation du langage Java.

Dans un premier temps, nous rappelons le contexte du projet. Puis dans un deuxième temps, nous expliquons les techniques étudiées pouvant résoudre notre problématique. Enfin dans un dernier temps, nous présenterons la technique choisie que nous implémenterons pour le second semestre.

2 Rappel du contexte du projet

L'objectif du projet est de faciliter l'apprentissage des outils de synchronisation.

Pour cela, nous allons mettre en oeuvre les compteurs de synchronisation en Java, ces derniers n'étant pas nativement implémentés.

Pour la mise en oeuvre de ce mécanisme de synchronisation il faut :

— comptabiliser les accès aux méthodes automatiquement.
— réaliser le contrôle d'accès aux méthodes (blocage et réveils) qui est basé sur des expressions booléennes.

2.1 Mécanisme des compteurs de synchronisations

Le processus d'exécution commence lors de l'invocation d'une méthode m, on commence par mettre à jour et réévaluer le compteur du nombres d 'invocations de requêtes depuis le début du programme (`m.req`) et ainsi que ceux qui sont en attente (`m.att`).

Ensuite, on évalue si la requête vérifie les conditions d'accès, à l'issue duquel il y a blocage éventuel. Quand une invocation vérifie les conditions d'accès, on met à jour le nombre d'invocations de m qui ont été autorisées (`m.aut`) et le nombre d'invocations en attente (`m.att`).

A la fin de l'exécution du corps de m, on met à jour le nombre d'invocations qui ont été réalisées (`m.term`) et ainsi que ceux qui sont en cours (`m.act`). Les compteurs de synchronisation peuvent être utilisés pour spécifier des conditions de manière concise avant d'utiliser d'autres mécanismes.

Tous les traitements de mises à jour et d'évaluations se font en exclusion mutuelle.

2.1.1 Exemple de traduction à générer

Le but de notre projet étant de traduire automatiquement un langage source Java avec des conditions et des compteurs vers un langage java utilisant la synchronisation ; nous avons fait deux exemples de traduction à générer afin d'en déduire des schémas types.

Ces deux exemples sont ceux du lecteur/rédacteur et du producteur/consommateur. Voici leurs codes et leurs conditions :

Lecteur/Rédacteur

Pour simplifier le code, nous avons choisi de faire un lecteur/rédacteur dans un buffer circulaire. La condition pour lire signifie qu'aucun écrivain n'est autorisé au même moment.

```
public Class LecteurRedacteur {
private final int TAILLE= 10;
    private String tab[] = new String [TAILLE];
```

```
    private int tab_lec=0;
    private int tab_red=0;

    condition lire =  (ecrire_act = = 0);
    condition ecrire = (ecrire_act = = 0 && lire_act = = 0);

    public String lire(){
     return tab[tab_lec];
        tab_lec=(tab_lec+1)%TAILLE;
    }

    public void ecrire(String s){
        tab[tab_red]=s;
        tab_red=(tab_red+1)%TAILLE;
    }
}

public Class ProducteurConsommateur {
    private final int TAILLE = 10;
    private object tab[] = new object [ TAILLE]
    private int tab_prod = 0;
    private int tab_cons = 0;

    condition produire = (produire_aut - consommer_term < N);
    condition consommer = (produire_term - consommer_aut >0);

    public void produire(object o){
        tab[tab_prod]=o;
        tab_prod=(tab_prod+1)%TAILLE;
    }

    public object consommer (){
        return tab[tab_cons];
     tab_cons=(tab_cons+1)%TAILLE;
}
```

Pour la traduction avec les compteurs de synchronisation, il faut mettre à jour les compteurs `att` et `act` ou `req`, `aut` et `term` à chaque fois que nécessaire. Il nous faut également faire un `notifyAll()` à chaque fois qu'un compteur est modifié afin que chaque processus bloqué teste de nouveau ses conditions. Finalement, nous utilisons `wait()` afin qu'un processus ne s'execute pas tant que les conditions ne sont pas remplies. Enfin, toutes modifications de compteur, les `wait()` et les `notify()` sont dans des blocs `synchronized` afin d'éviter que plusieurs threads n'éxécutent ces instructions en même temps.

Cependant, il faut veiller à ne pas tout mettre en exclusion mutuelle (dans le `synchronized`), car dans le cas du lecteur/rédacteur par exemple nous aboutirions à une situation bloquée.

Le reste du programme, lui, n'est pas modifié.

```
public Class LecteurRedacteur {
    private final int TAILLE = 10;
    private String tab[] = new String [TAILLE];
    private int tab_lec = 0;
    private int tab_red = 0;
```

```
    // Déclaration des compteurs à générer automatiquement

    private int lire_req = 0;
    private int lire_aut = 0;
    private int lire_term = 0;
    private int lire_att = 0;
    private int lire_act = 0;
    private int ecrire_req = 0;
    private int ecrire_aut = 0;
    private int ecrire_term = 0;
    private int ecrire_att = 0;
    private int ecrire_act = 0;

// Traduction automatique dans des méthodes des conditions

    public boolean cond_lire () {
         return (ecrire_act == 0 );
    }

    public boolean cond_ecrire () {
         return (ecrire_act==0 && lire_act==0 );
}


    public String lire(){
            synchronized (this) {
             lire_req++;
                this.notifyAll();

     lire_att++;
     this.notifyAll();

                while (  ! cond_lire() ){
                 this.wait();
             }

                lire_aut++;
                this.notifyAll();

                lire_att--;
                this.notifyAll();

                lire_act++;
                this.notifyAll();
            }

     return tab[tab_lec];

            tab_lec=(tab_lec+1)%TAILLE;

            synchronized (this){
             lire_term++;
                this.notifyAll();
```

```
                lire_act--;
                this.notifyAll();

            }
    }

    public void ecrire( String s){
            synchronized ( this ) {
             ecrire_req++;
                this.notifyAll();

                ecrire_att++;
                this.notifyAll();

                while ( ! cond_ecrire() ){
                    this.wait();
             }

                ecrire_aut++;
                this.notifyAll();

                ecrire_att--;
                this.notifyAll();

                ecrire_act++;
                this.notifyAll();
            }
            tab[tab_red]=s;

            tab_red=(tab_red+1)%TAILLE;

            synchronized(this){
             ecrire_term++;
                this.notifyAll();

                ecrire_act--;
                this.notifyAll();
            }
}
}
```

Mise en oeuvre :

Le reste du programme n'est pas modifié

```
public class Producteur extends Thread {

  private Buffer buf;
  private int identité;

  public Producteur(Buffer c, int n) {
      buf = c; this.identité = n;
  }
```

```
      public void run() {
         buf.ecrire(msg);
         System.out.println("Producteur #" + this.identité + " met : " + i);
      }
}

     public class Consommateur extends Thread {

      private Buffer buf;
      private int identité;

      public Producteur(Buffer c, int n) {
          buf = c; this.identité = n;
      }

      public void run() {
        System.out.println("Producteur #" + this.identité + " met : " + i);
         System.out.println(buf.lire());
      }
}

public class Main {
     public static void main(String[] args) {
          Buffer c = new Buffer();
          Producteur p1 = new Producteur(c, 1);
          Producteur p2 = new Producteur(c, 2);
          Producteur p3 = new Producteur(c, 3);

          Consommateur c1 = new Consommateur(c, 1);
          Consommateur c2 = new Consommateur(c, 2);
          Consommateur c3 = new Consommateur(c, 3);

          p1.start();p2.start();p3.start();
          c1.start();c2.start();c3.start();
          }
    }
```

Producteur/Consommateur

```
public Class ProducteurConsommateur {

    private final int TAILLE = 10;
    private object tab[] = new object [ TAILLE]
    private int tab_prod = 0;
    private int tab_cons = 0;
    private int produire_req = 0;
    private int produire_aut = 0;
    private int produire_term = 0;
    private int produire_att = 0;
    private int produire_act = 0;
    private int consommer_req = 0;
    private int consommer_aut  = 0;
    private int consommer_term = 0;
```

```
    private int consommer_att = 0;
    private int consommer_act = 0;

    public boolean cond_produire () {
            return (produire_aut - consommer_term < N);
    }

    public boolean cond_consommer () {
            return (produire_term - consommer_aut >0);
}

    public void produire(Object o){
        synchronized ( this ){
            produire_req++;
            this.notifyAll();

            produire_att++;
            this.notifyAll();

            while (! cond_produire()){
                this.wait();
            }
            produire_aut++;
            this.notifyAll();

            produire_att--;
            this.notifyAll();

            produire_act++;
            this.notifyAll();
        }
        tab[tab_prod]=o;
        tab_prod=(tab_prod+1)%TAILLE;
        synchronized (this){

            produire_term++;
            this.notifyAll();

            produire_act--;
            this.notifyAll();

        }
    }

    public Object consommer (){
        synchronized ( this ){
            consommer_req++;
            this.notifyAll();

            consommer_att++;
            this.notifyAll();

            while (!cond_consommer()){
                this.wait();
```

```
            }

            consommer_aut++;
            this.notifyAll();

            consommer_att--;
            this.notifyAll();

            consommer_act++;
            this.notifyAll();
        }
        return tab[tab_cons];

        tab_cons=(tab_cons+1)%TAILLE;

        synchronized (this){
            consommer_term++;
            this.notifyAll();

            consommer_act--;
        }


}
```

Dans une deuxième phase, nous pourrons optimiser cette traduction automatique. On peut par exemple enlever les compteurs (et les `notifyAll()` qui les suivent) qui ne sont pas utilsés dans les conditions. Dans la classe LecteurRédacteur seuls les compteurs ecrire_act et lire_act sont utilisés dans les conditions. Voici par exemple la méthode lire avec cette optimisation :

```
    public String lire(){
            synchronized (this) {
                while (  ! cond_lire() ){
                this.wait();
            }
            lire_act++;
            this.notifyAll();
            }
            return tab[tab_lec];
            tab_lec=(tab_lec+1)%TAILLE;
            synchronized (this){
            lire_act--;
            this.notifyAll();
            }
    }
```

Enfin, nous pourrons optimiser d'autant plus si nous décidons par exemple de ne tester les conditions que si le compteur a "évolué dans le bon sens". Par exemple si une méthode ne peut s'exécuter que si `lire_att == 0`, il est inutile de réévaluer sa condition aprés avoir éxécuté `lire_att++` (les compteurs étant tous positifs)

3 Techniques de réalisation

3.1 Introspection

Une première technique que nous avons étudié pour faire cette réalisation est l'introspection.

L'introspection consiste en la découverte, de façon dynamique, des informations propres à une classe Java. Ces informations décrivent de façon exhaustive les caractéristiques d'une classe Java (champs, méthodes, ...).

Ce mécanisme est utilisé par la machine virtuelle Java en cours d'exécution, mais également par les outils de développements. L'API Java qui permet ceci est l'API Reflection.

Au chargement d'une classe Java, la JVM crée automatiquement un objet. Celui-ci récupère toutes les caractéristiques de la classe. Il s'agit d'un objet Class.

Nous créons par exemple trois nouvelles classes Java. À l'exécution du programme, la JVM va créer un objet Class pour chacune d'elles. Cet objet possède une multitude de méthodes permettant d'obtenir tous les renseignements possibles et imaginables sur une classe parmi lesquelles on peut citer.

Méthodes	Rôle
static Class forName(String)	Instancier un objet de la classe dont le nom est fourni en paramètre et renvoie un objet Class la représentant
Class[] getClasses()	Renvoyer les classes et interfaces publiques qui sont membres de la classe
Constructor[] getConstructors()	Renvoyer les constructeurs publics de la classe
Class[] getDeclaredClasses()	Renvoyer un tableau des classes définies comme membres dans la classe
Constructor[] getDeclaredConstructors()	Renvoyer tous les constructeurs de la classe
Field[] getDeclaredFields()	Renvoyer un tableau de tous les attributs définis dans la classe

Ces méthodes à leurs tours possèdent leurs propres méthodes statiques qui permettent d'extraire ou de vérifier une information bien définie.Prenons par exemple la méthode getmodifier() qui retourne un entier représentant les modificateurs de la classe. Pour décoder cette valeur, la classe Modifier possède plusieurs méthodes qui attendent cet entier en paramètre et qui retourne un booléen selon leur fonction : isPublic(), isAbstract(), isFinal(), ...

La réflexivité n'est pas utilisée seulement pour connaître toutes les informations concernant une classe donnée, nous pourrons aussi à travers ce mécanisme créer des instances de classe de façon dynamique.

3.1.1 Paquetage Java.lang.reflect

Le paquetage `java.lang.reflect` possède plusieurs classes et interfaces permettant de mettre en oeuvre le mécanisme d'introspection.

Les trois classes `Field, Method et Constructor` possèdent un modificateur final. La machine Virtuelle Java peut seulement créer des instances de ces classes. Ces objets sont utilisés pour manipuler les objets sous-jacents : obtenir les informations d'introspection à propos des constructeurs et membres sous-jacents, obtenir et fixer les valeurs des champs, invoquer les méthodes sur les objets ou les classes, créer de nouvelles instances des classes.

La classe Array est également finale mais non-instanciable. Elle fournit des méthodes statiques qui permettent de créer de nouveaux tableaux et d'obtenir ou de fixer les éléments des tableaux.

Exemple

```
import java.lang.reflect.Constructor;
import java.lang.reflect.Field;
import java.lang.reflect.Method;

public class Introspection {
  public static void main(String[] args) {
    if (args.length > 0){
      try {
        Class classe = Class.forName(args[0]);
        Field[] champs = classe.getDeclaredFields();
        Method[] methodes = classe.getDeclaredMethods();
        Constructor[] constructeurs = classe.getDeclaredConstructors();
        System.out.println("Les champs de la classe " + classe.getName());
        System.out.println("Num :\tChamp\tType");
        for(int i = 0; i < champs.length; i++){
          System.out.println(i + " :\t" + champs[i].getName()
                                              + "\t" + champs[i].getType());
        }
        System.out.println("\nLes méthodes de la classe " + classe.getName());
        System.out.println("Num :\tType de retour\tMethode\t"
                                             + "Types des Parametres\tTypes des Exc
        for(int i = 0; i < methodes.length; i++){
          System.out.print(i + " :\t" + methodes[i].getReturnType()
                                               + "\t" + methodes[i].getName() + "\
          afficherInfos(methodes[i], "getParameterTypes");
          System.out.print("\t");
          afficherInfos(methodes[i], "getExceptionTypes");
          System.out.println("");
        }
        System.out.println("\nLes constructeurs de la classe " + classe.getName());
        System.out.println("Num :\tConstructeur\tTypes des Parametres"
                                              + "\tTypes des Exceptions");
        for(int i = 0; i < constructeurs.length; i++){
          System.out.print(i + " :\t" + constructeurs[i].getName() + "\t");
```

```
          afficherInfos(constructeurs[i], "getParameterTypes");
          System.out.print("\t");
          afficherInfos(constructeurs[i], "getExceptionTypes");
          System.out.println("");
        }
      }
      catch (ClassNotFoundException e) {
        e.printStackTrace();
        System.out.println("La classe n'a pu être déterminée dans la méthode main()!");
      }
    }
  }
```

La méthode `afficherInfos()` permet d'afficher des informations relatives à l'objet passé en argument. Les informations sont extraites par rapport au nom d'une méthode, laquelle doit être un membre de l'objet précité.

`@param obj` correspond à l'objet à partir duquel des informations doivent être extraite.

`@param methode` correspond à une méthode de l'objet et est destinée à extraire des informations.

```
  public static void afficherInfos(Object obj, String methode){
    try {
      Object types = obj.getClass().getMethod(methode, null).invoke(obj, null);
      Class[] tabTypes = (Class[])types;
      for(int i = 0; i < tabTypes.length; i++){
        System.out.print(i > 0 ? ", " : "\t");
        System.out.print(tabTypes[i].getName());
      }
    }
    catch (Exception e) {
      e.printStackTrace();
      System.out.println("Une exception a été lancée dans la méthode afficherInfos() !");
    }
  }
}
```

3.1.2 Limites de la réflexivité

Il existe deux grands inconvénients à la réflexivité.

Le premier est la détermination du comportement du programme lors de l'exécution : la plupart des erreurs ne peuvent être repérées à la compilation, donc le programme peut se bloquer d'un moment à l'autre. Ainsi des erreurs d'exécution difficiles à comprendre peuvent se produire lors de l'exécution.

Le second inconvénient de cette technique est le remplacement des traitements faits habituellement à la compilation par des actions réalisées à l'exécution comme dans le cas des langages interprétés. On perd donc énormément en vitesse d'exécution avec un code dont la vitesse d'exécution est multipliée 10 à 20 fois [1]. La réflexivité est donc à éviter si les performances sont cruciales pour l'application.

Finalement, nous avons décidé de ne pas nous tourner vers l'introspection pour réaliser le projet car en plus des inconvénients de cette technique, il nous aurait fallut maîtriser ce nouveau concept en plus de comprendre le parallélisme.

1. Selon un article publié en www.toshy.net "initiation à la réflexion"

Ainsi nous nous sommes tournés vers l'utilisation des annotations combinée avec des techniques de compilation pour pouvoir transformer un code simple avec des compteurs de synchronisation à un code utilisant les outils de synchronisation java. Ces deux outils font l'objet des deux parties suivantes.

3.2 Annotation

Une annotation permet de marquer certains éléments du langage Java afin de leur ajouter une propriété particulière. Ces annotations peuvent ensuite être utilisées à la compilation ou à l'exécution pour automatiser certaines tâches.

Les annotations utilisent leur propre syntaxe. Une annotation s'utilise avec le caractère `@` suivi du nom de l'annotation .

3.2.1 La définition d'une annotation

Sur la plate-forme Java, une annotation est une interface lors de sa déclaration et est une instance d'une classe qui implémente cette interface lors de son utilisation. Les annotations sont particulièrement adaptées à la génération de code source afin de faciliter le travail des développeurs notamment sur des tâches répétitives.

La définition d'une annotation nécessite une syntaxe particulière utilisant le mot clé interface. Une annotation se déclare donc de façon similaire à une interface.

3.2.2 Exemple

```
//Ajout des membres à l'annotation  en définissant une méthode
dont le nom correspond au nom de l'attribut en paramètre de l'annotation.

package com.test.annotations;

public @interface MonAnnotation {
String arg1();
String arg2();
}

package com.test.annotations;

@MonAnnotation(arg1="valeur1", arg2="valeur2")
public class MaCLasse {
}

Les annotations pour la réalisation de notre projet correspondent aux suivants:

 @condition lire(ecrire_act = = 0);
 @condition ecrire(ecrire_act = = 0 && lire_act = = 0);
```

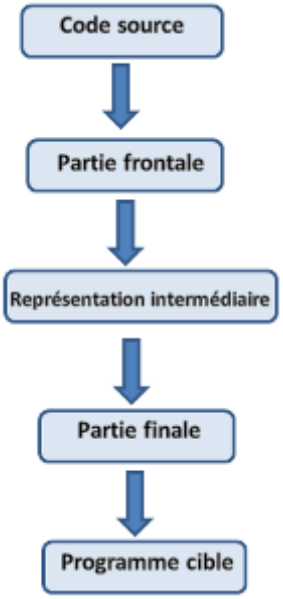
3.3 Compilation

En troisième année, nous avons écrit un compilateur dans le cadre d'un projet ce qui nous a permis de comprendre les techniques de compilation. Nous en rappelons donc ici les bases.

3.3.1 Les bases de la compilation

Un compilateur est un programme effectuant la traduction d'un code source lisible et manipulable par l'être humain vers un langage cible.

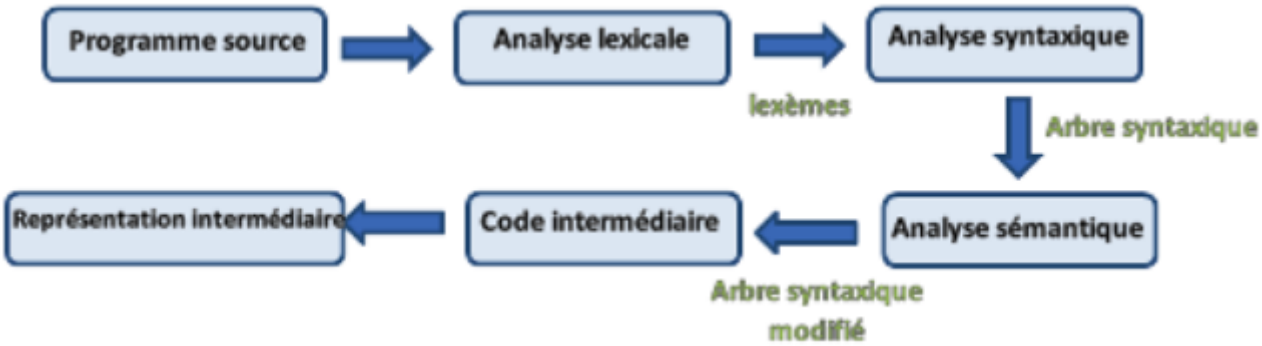
Le processus de compilation s'effectue comme suit :



La partie frontale (le frontend) du compilateur se charge de réaliser différentes analyses du code-source en même temps qu'elle le transforme en une représentation intermédiaire. Cette représentation intermédiaire est ensuite communiquée à la partie finale (le backend) du compilateur, dont le rôle est de générer le code du programme dans le langage cible, en opérant éventuellement une série d'optimisations propres à l'architecture visée. Cette séparation logique n'est pas obligatoirement visible dans tous les compilateurs, néanmoins elle facilite la conception de programmes portables, puisque leur partie frontale est totalement indépendante du langage cible, et leur partie finale indépendante du langage source.

Classiquement, le frontend d'un compilateur est chargé de réaliser trois phases d'analyses : lexicale, syntaxique et sémantique.

Son architecture canonique est la suivante :



Le rôle de l'analyse lexicale est de reconnaître les différents "mots" du programme et de leur associer une étiquette décrivant la nature de ces mots. La structure ainsi obtenue est appelée lexème. Grâce à cela nous pourrons définir des mots clés spécifiques aux compteurs de synchronisation pour les incrémenter et/ou décrémenter selon le contexte.

Le flux de lexèmes est ensuite envoyé en entrée d'un analyseur syntaxique, chargé d'assembler les mots pour reconnaître des phrases. La forme des phrases acceptées par le langage est

décrite par une grammaire constituée d'un certain nombre de règles que nous aurons défini. Plus précisemment,pour notre projet, nous utiliserons la grammaire du langage Java que nous modifierons pour permettre à l'utilisateur d'utiliser nos mots clés.

L'analyse syntaxique permet donc de détecter des constructions erronnées telles que "a = + * / ; 32", mais aussi de représenter celles-ci sous forme d'une structure arborescente, appelée arbre syntaxique (ou AST pour Abstract Syntax Tree), qui décrit plus précisément leur forme.

L'AST représentant le programme est ensuite soumis à une analyse sémantique qui peut éventuellement le modifier. Le rôle de celle-ci est de s'assurer que les phrases produites à l'étape précédente, qui sont syntaxiquement correctes, expriment quelque chose de sensé dans le langage. C'est durant l'analyse sémantique que sont vérifiées toutes les contraintes concernant le type des variables et des expressions.

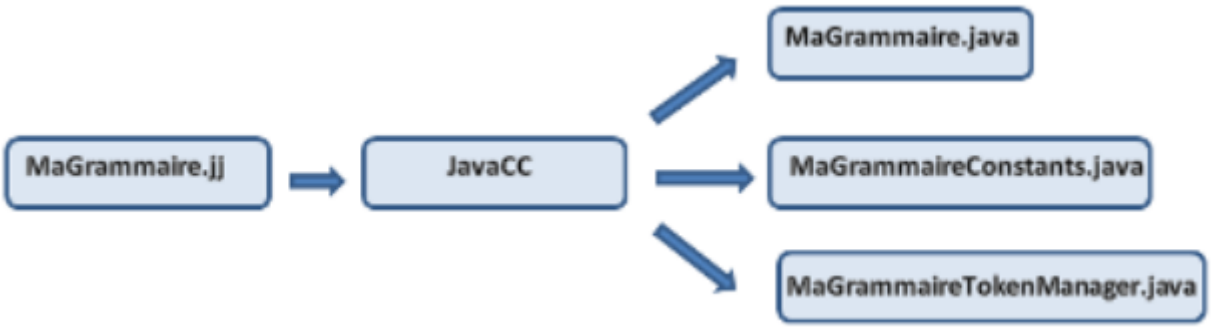
3.3.2 Générateurs d'analyse lexicale et syntaxique

Il existe aujourd'hui plusieurs générateurs d'analyse lexicale et syntaxique (parser generator) notamment pour notre projet, nous utiliserons JavaCC.

`Java Compiler Compiler (JavaCC)` est le générateur d'analyseur syntaxique le plus utilisé pour les applications Java. En plus du générateur d'analyseur lui-même, JavaCC fournit d'autres capacités standards liées à la génération de l'analyseur, comme la construction de l'arbre (via un outil appelé JJTree inclus avec JavaCC), les actions 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.

Ce mécanisme est expliqué par le schéma suivant :



Le fichier `MaGrammaire.jj` est dénifit comme suit :

```
PARSER_BEGIN(MonAnalyseur)
public class MonAnalyseur{
     public static void main (String args[]){
         MonAnalyseur parser = new MonAnalyseur(System.in);
            try{
             parser.racine();

            }catch(ParserException e){
             System.out.println("erreur : "+e.getMessage());

            }
```

```
        }
    }

PARSER_END(MonAnalyseur)
// les productions
...

void racine() : {}{ ... }
...
```

La définition des lexèmes (token) se fait comme la portion de code suivante où nous spéficions les litérales(eniters, décimaux, hexadécimaux, charactères) :

```
/* LITERALS */

TOKEN :
{
  < INTEGER_LITERAL:
        <DECIMAL_LITERAL> (["l","L"])?
      | <HEX_LITERAL> (["l","L"])?
      | <OCTAL_LITERAL> (["l","L"])?
  >
|
  < #DECIMAL_LITERAL: ["1"-"9"] (["0"-"9"])* >
|
  < #HEX_LITERAL: "0" ["x","X"] (["0"-"9","a"-"f","A"-"F"])+ >
|
  < #OCTAL_LITERAL: "0" (["0"-"7"])* >
|
  < FLOATING_POINT_LITERAL:
        (["0"-"9"])+ "." (["0"-"9"])* (<EXPONENT>)? (["f","F","d","D"])?
      | "." (["0"-"9"])+ (<EXPONENT>)? (["f","F","d","D"])?
      | (["0"-"9"])+ <EXPONENT> (["f","F","d","D"])?
      | (["0"-"9"])+ (<EXPONENT>)? ["f","F","d","D"]
  >
|
  < #EXPONENT: ["e","E"] (["+","-"])? (["0"-"9"])+ >
|
  < CHARACTER_LITERAL:
      "'"
      (   (~["'","\\","\n","\r"])
        | ("\\"
            ( ["n","t","b","r","f","\\","'","\""]
            | ["0"-"7"] ( ["0"-"7"] )?
            | ["0"-"3"] ["0"-"7"] ["0"-"7"]
            )
          )
      )
      "'"
  >
|
  < STRING_LITERAL:
      "\""
```

```
      (   (~["\"","\\","\n","\r"])
        | ("\\"
            ( ["n","t","b","r","f","\\","'","\""]
            | ["0"-"7"] ( ["0"-"7"] )?
            | ["0"-"3"] ["0"-"7"] ["0"-"7"]
            )
          )
      )*
      "\""
  >
}
```

L'ajout de code java permettant de transformer un langage source en un langage cible peut se faire à la lecture d'un lexème ou à l'exécution d'une règle comme le montre les exemples suivantes :

```
void declFonction() :{}
{
type() <FONCTION> <ident>

{
// code java : appel de fonctions après avoir lu le token <ident>
expression.declNomFonc=YakaTokenManager.identLu;
declaration.addFonction(expression.declNomFonc, expression.lastType, Token.beginLine);
yvmasm.nomFonctionASM(expression.declNomFonc);
}

paramForms()
bloc()
<FFONCTION>

{
// code java : appel de fonctions après l'exécution d'une règle
expression.flag = F;
yvmasm.fermeblocASM(tabIdent.getNbParam()*2);
tabIdent.affiche();
}
}
```

Il existe d'autres générateurs de parser, pour ne citer que :
— Lex
— Yacc
— JFlex
— ou encore RunCC
Ces outils donnent toutes les fonctionnalités standards d'un compilateur avec des quelques variantes en option plus avancés.

Pour notre projet, nous utiliserons JavaCC qui est un outil que l'on maîtrise déjà grâce au projet compilateur de troisième année pour ne garder que le parallélisme comme difficulté qui est déjà assez importante.

4 Conclusion

Cette phase de spécification nous a permis d'étudier les techniques possibles pour résoudre le problème qui nous a été posé et nous permettra de définir l'architecture logicielle de notre projet. Et à partir de cette étude, nous établirons une planification des tâches .

Nous avons estimé que la technique de l'introspection est assez complexe et ajoute beaucoup de difficultés au fait que nous ne maîtrisons pas encore parfaitement le parallélisme.

La technique de compilation combinée avec l'utilisation des annotations est un bon compromis car nous maîtrisons la compilation grâce au projet compilateur réalisé en troisième année. Nous utliserons donc ces deux méthodes pour réaliser notre projet.