Université des Sciences et de la Technologie Houarri Boumediene

Faculté d'électronique et informatique

Département d'informatique

MASTER I Système Informatiques Intelligents

PROJET COMPILATION

Réalisation d'un mini compilateur pour le langage 'Small Java'

Avec l'outil ANTLR

METALLAOUI Nassim 161636006074

HAMIDET Hakim 201500010434

Groupe 01

2020/2019

Introduction

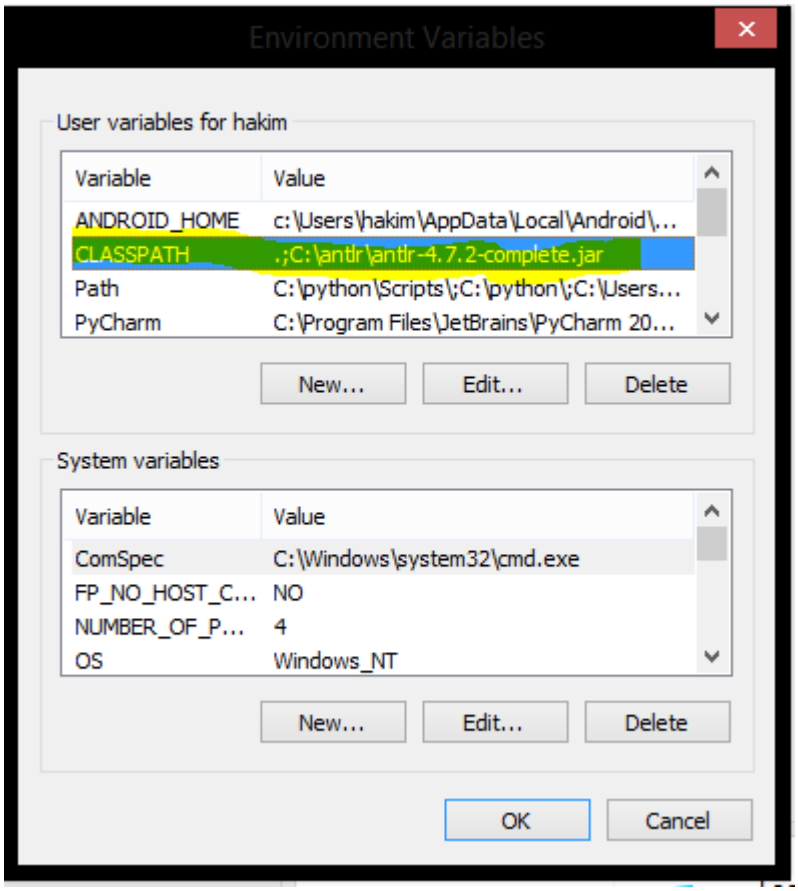
Dans ce projet, nous avons pour objectif de créer un compilateur capable d'interpréter notre nouveau langage de programmation qu'on appellera 'Small Java'. Mais avant c'est quoi un compilateur. Un compilateur est un traducteur automatique qui transforme un langage de programmation donnée, qui a sa propre alphabet (*Analyse lexicale*), son propre vocabulaire (*Analyse syntaxique*) et ses règles de grammaire (*Analyse sémantique*), en un code cible et traduit pour que la machine puisse l'exécuter. D'où, le compilateur aura à faire plusieurs opérations, on citera : l'analyse lexicale, l'analyse syntaxique et l'analyse sémantique.

Pour la réalisation de notre projet, on se dotera de **ANTLR** (ANother Toul For Language Recognition) un framework libre de construction de compilateur utilisant une analyse LL(*). ANTLR permet de générer des analyseurs lexicaux, syntaxiques ou des analyseurs lexicaux et syntaxiques combinés. Pour la suite du projet, on utilisera ANTLR avec le langage JAVA et comme IDE on a choisi d'utiliser **IntelliJ**.

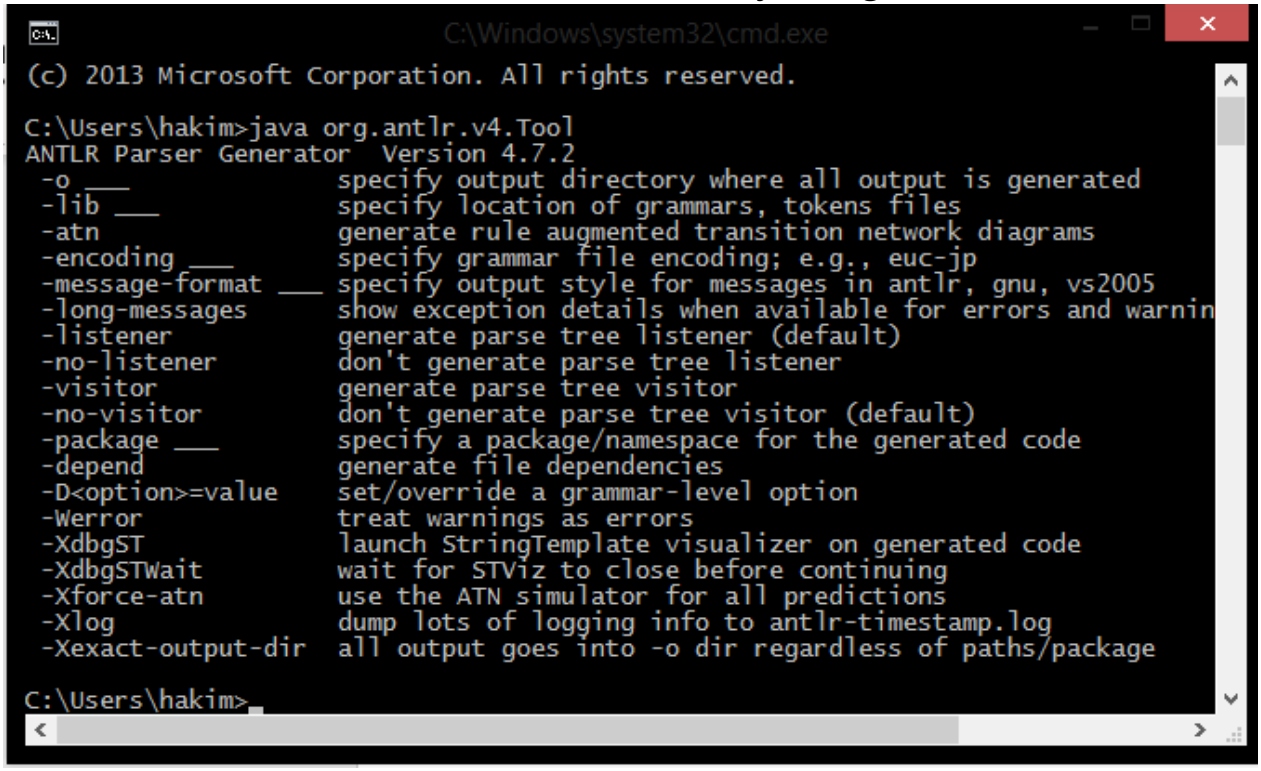
❖ Utilisation d'ANTLR :

– L'installation :

1. Ajouter le chemin du fichier « antlr-4.7.2-complete.jar » obtenu par le biais de la responsable TP dans la variable d'environnement CLASSPATH.



2. Tester le bon fonctionnement avec la commande : java org.antlr.v4.Tool



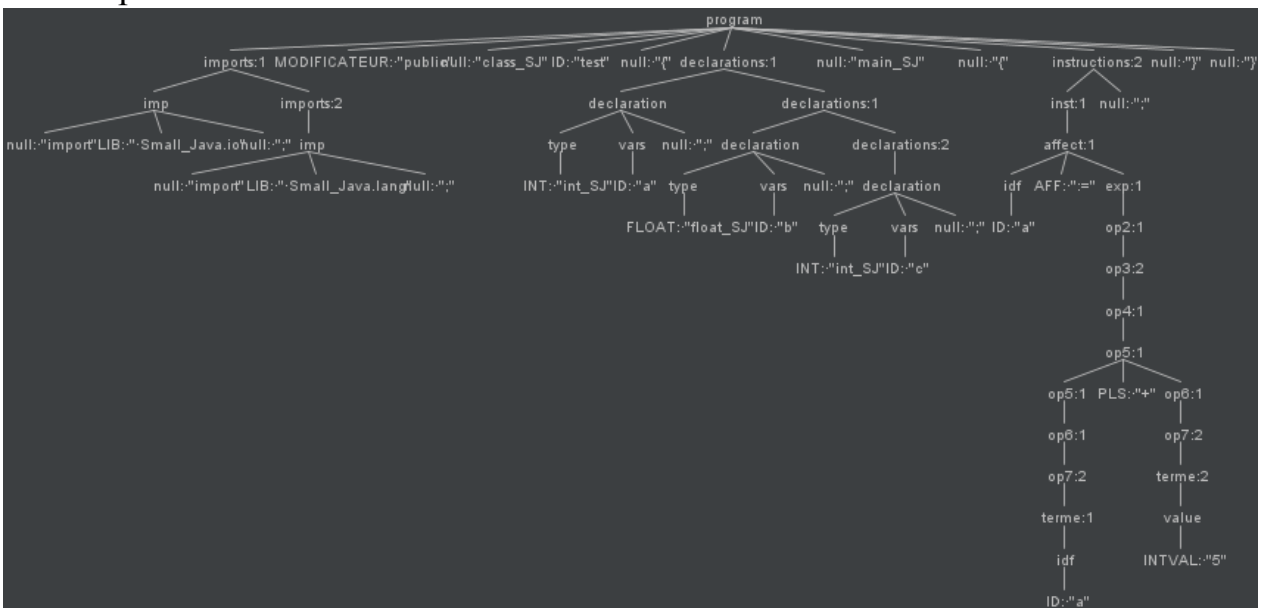
- **IntelliJ :**

Au début, l'IDE IntelliJ ne reconnait pas notre grammaire. Pour remédier à ce problème on a dû télécharger le plugin 'ANTLR v4 grammar plugin'. Ce dernier est pour les grammaires ANTLR v4 et inclut ANTLR 4.7.2

Après l'avoir téléchargé, IntelliJ reconnait notre langage, il s'est affiché comme suit :

```
grammar Small_Java;

program : (imports)? MODIFICATEUR 'class_SJ' ID  '{' declarations 'main_SJ' '{' instructions '}' '}' ;
imp: 'import' LIB ';';
imports: imp imports|imp;
declarations : (declaration declarations)|declaration  ;
declaration : type vars ';' ;
type : INT | FLOAT | STRING;
vars : ((ID ',' vars) | ID) ;
idf : ID;
value : INTVAL | FLOATVAL |STRVAL;
instructions : (inst ';' instructions) | inst ';' ;
inst : affect | ifinst| read | write;
affect : idf ':=' exp|value ;
exp: exp OU op2 | op2;
op2: op2 ET op3 | op3;
op3: NEG op4 | op4;
op4: op4 comp op5 | op5;
op5: op5 PLS op6 | op5 MNS op6|op6;
op6: op6 MUL op7 | op6 DIV op7 | op7;
op7: '('exp')'| terme;
terme:idf | value;
read:'In_SJ' '('signe',' idf ')';
```

Et on a pu afficher l'arbre :



Ce rapport sera décomposé en 3 parties. La 1ère dans laquelle on abordera l'analyse lexicale et syntaxique suivi par l'analyse sémantique. On finira par la génération du code objet de notre compilateur.

I. Analyse lexicale et syntaxique

L'analyse lexicale est la première phase de la chaîne de compilation. Elle consiste à convertir une chaîne de caractères en une liste de symboles. Ces symboles sont ensuite consommés lors de l'analyse syntaxique.

L'analyse syntaxique consiste à mettre en évidence la structure d'un texte, généralement une phrase écrite dans une langue naturelle.

Pour l'analyse lexicale de notre langage on a utilisé les langages formels et les expressions régulières pour générer par la suite l'arbre syntaxique. L'arbre syntaxique est important, car il est le support de la sémantique du langage.

La capture ci-dessous présente notre langage avec toutes ses propriétés. Il est contenu dans un fichier **.g4** propre aux fichier de grammaire de ANTLR.

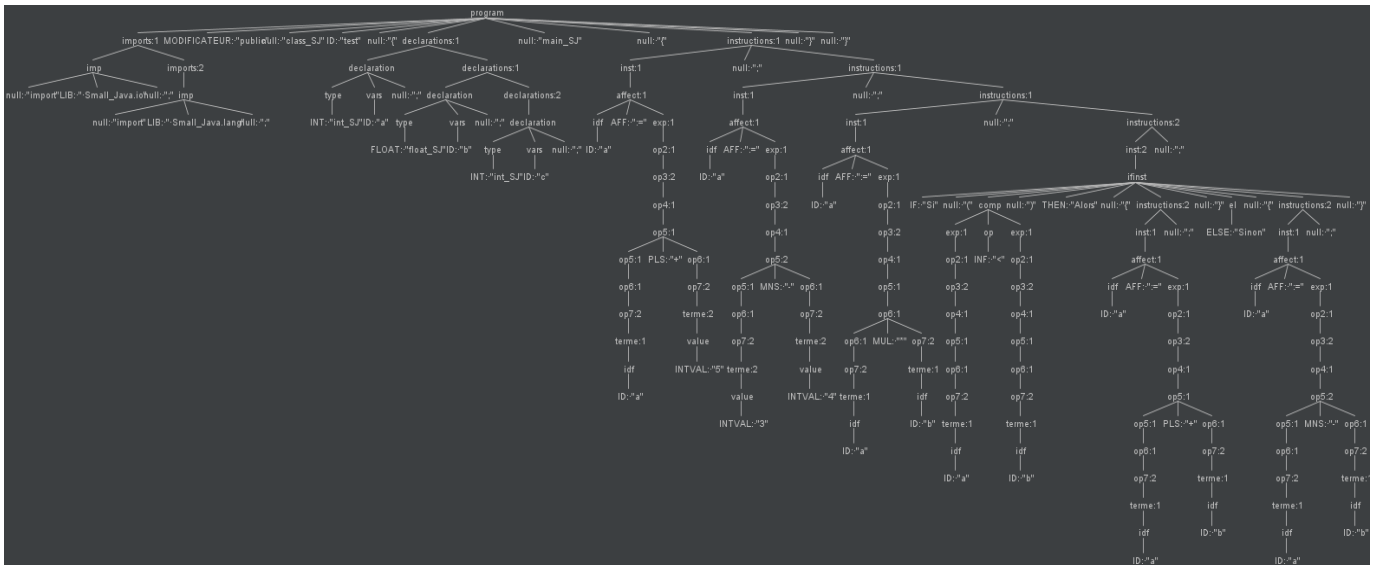
```
grammar Small_Java;

program : (imports)? MODIFICATEUR 'class_SJ' ID  '{' declarations 'main_SJ' '{' i
nstructions '}' '}' ;
imp: 'import' LIB ';';
imports: imp imports|imp;
declarations : (declaration declarations)|declaration  ;
declaration : type vars ';' ;
type : INT | FLOAT | STRING;
vars : ((ID ',' vars) | ID) ;
idf : ID;
value : INTVAL | FLOATVAL |STRVAL;
instructions : (inst ';' instructions) | inst ';' ;
inst : affect | ifinst| read | write;
affect : idf ':=' exp|value ;
exp: exp OU op2 | op2;
op2: op2 ET op3 | op3;
op3: NEG op4 | op4;
op4: op4 comp op5 | op5;
op5: op5 PLS op6 | op5 MNS op6|op6;
op6: op6 MUL op7 | op6 DIV op7 | op7;
op7: '('exp')'| terme;
```

```
terme:idf | value;
read:'In_SJ' '('signe',' idf ')';
write:'Out_SJ' '(' STRVAL','idf')';
signe:INTS|FLOATS|STRINGS;
ifinst : IF '(' comp ')' THEN '{' instructions '}' ( |el '{' instructions '}');
el : ELSE;
comp : exp op exp ;
op : SUP | INF | SUPEGAL | INFEGAL | DEFF | EGAL ;


listID : idf ',' listID | idf ;
INTS:'"%d"';
FLOATS:'"%f"';
STRINGS:'"%s"';
IF : 'Si';
THEN : 'Alors';
ELSE : 'Sinon';
MODIFICATEUR:'public'|'protected';
ID : [a-zA-Z][a-zA-Z0-9]*;
INT : 'int_SJ';
FLOAT : 'float_SJ';
STRING : 'string_SJ';

ET:'&';
OU:'|';
NEG:'!';
AFF: ':=';
PLS:'+';
MNS:'-';
DIV:'/';
MUL:'*';
EGAL: '=';
DEFF: '!=';
SUP : '>';
INF : '<';
SUPEGAL:'>=';
INFEGAL:'<=';
INTVAL : '0'|[1-9][0-9]* ;
FLOATVAL : '0'|[1-9][0-9]*('.'[0-9]*) ;
LIB:' Small_Java.lang'|' Small_Java.io';
WHITESPACE : [ \n\t\r] -> skip;
STRVAL : '"'(~["]|'\\"')*'"';
```

Pour tester notre grammaire, on a utilisé cette portion de code :

```
import Small_Java.io;

import Small_Java.lang;


public class_SJ test
{
    int_SJ a;
    float_SJ b;
    int_SJ c;
    main_SJ
    {
        a:=a+5;
        a:=3-4;
        a:=a*b;
        Si(a<b) Alors
        {
            a:=a+b;
        }Sinon
        {
            a:=a-b;
        };

    }
}

```

Et voici l’arbre syntaxique correspondant :



II. Analyse sémantique

L'analyse sémantique est l'étape où on établit la signification d'une instruction. Elle intervient après l'analyse syntaxique et avant la génération de code. Elle effectue les vérifications nécessaires à la sémantique du langage de programmation considéré, ajoute des informations à l'arbre syntaxique et construit la table des symboles.

Pour la réalisation de l'analyseur sémantique, on a procédé à la création des classes suivantes :

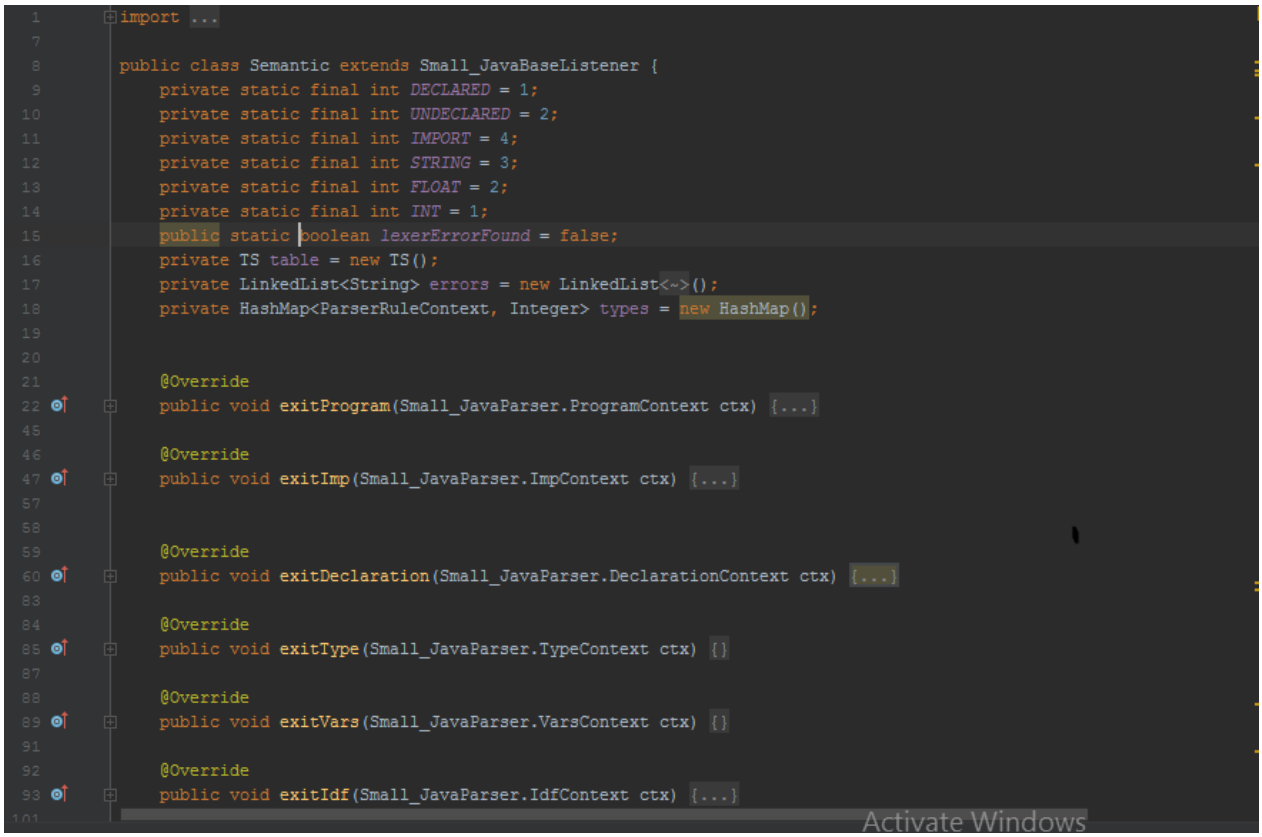
- ***Element :*** Ici on définit un élément de la table des symboles. Un Element a pour attribut : nom, type, s'il est déclaré ou non, s'il est défini ou non, avec tous les getters et setters sans oublier le constructeur.

```
static public class Element {

    public Element(String name, int declared, int type, int value ,int defined) {...}

    String name;
    int declared; // 2:undeclared 1:declared
    int type; // 1:int 2:float 3:string
    int value;
    int defined;

    @Override
    public String toString()
    {...}

    public void setDefined(int defined) { this.defined = defined; }
}


public ArrayList<Element> L = new ArrayList<~>();


public Element getElement(String name)
{
    for (int i = 0; i < L.size(); i++) {
        if(L.get(i).name.equals(name))
            return L.get(i);
    }
    return null;
}
```

- *TS :* Ici on définit notre table des symboles qui représente une lise de Element. Cette class java a pour méthode : afficher la table, recherche d'un élément et ajout d'un élément.

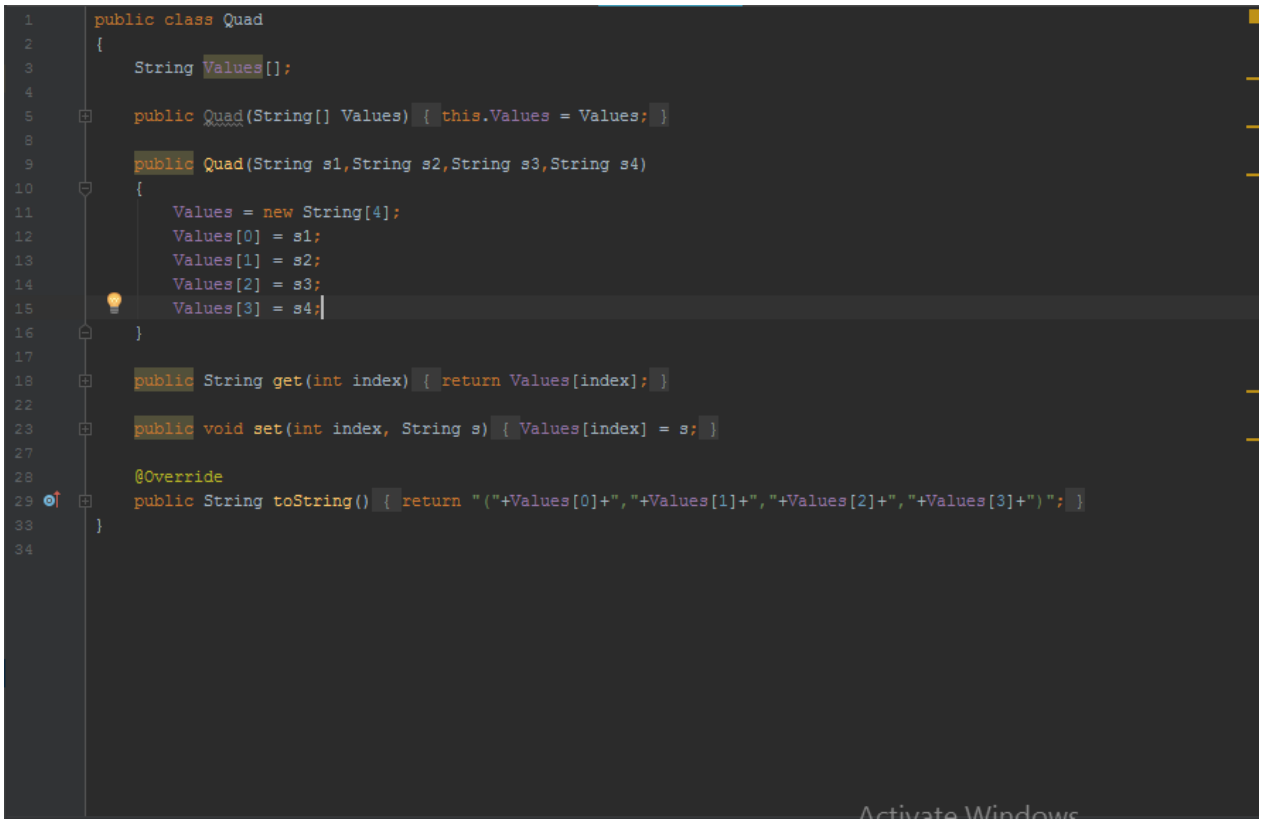
```java
import java.util.ArrayList;

public class TS
{
    static public class Element {...}


    public ArrayList<Element> L = new ArrayList<~>();


    public Element getElement(String name)
    {
        for (int i = 0; i < L.size(); i++) {
            if(L.get(i).name.equals(name))
                return L.get(i);
        }
        return null;
    }

    public boolean containsElement(String name) { return getElement(name) != null; }

    public void addElement(Element e) { L.add(e); }

    public void deleteElement(String name)
    {...}

    public void deleteElement(Element e) { L.remove(e); }

    public int getSize() { return L.size(); }

    public Element getElement(int i) { return L.get(i); }
```

- *La classe Semantic :* Toutes les classes créées précédemment vont être utilisé dans cette classe, qui redéfini les méthodes **enter__()**, **exit__()**.



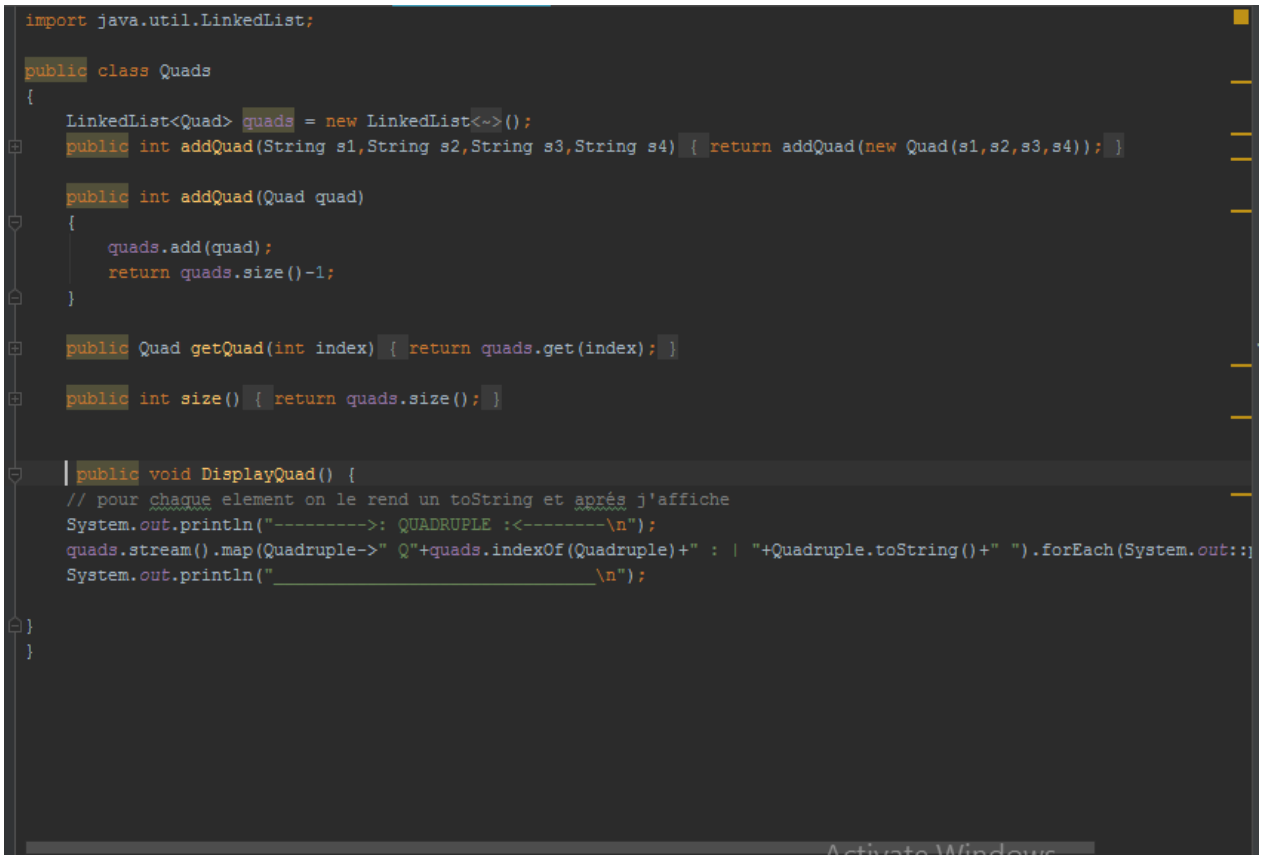
```java
import ...

public class Semantic extends Small_JavaBaseListener {
    private static final int DECLARED = 1;
    private static final int UNDECLARED = 2;
    private static final int IMPORT = 4;
    private static final int STRING = 3;
    private static final int FLOAT = 2;
    private static final int INT = 1;
    public static boolean lexerErrorFound = false;
    private TS table = new TS();
    private LinkedList<String> errors = new LinkedList<~>();
    private HashMap<ParserRuleContext, Integer> types = new HashMap();


    @Override
    public void exitProgram(Small_JavaParser.ProgramContext ctx) {...}

    @Override
    public void exitImp(Small_JavaParser.ImpContext ctx) {...}


    @Override
    public void exitDeclaration(Small_JavaParser.DeclarationContext ctx) {...}

    @Override
    public void exitType(Small_JavaParser.TypeContext ctx) {}

    @Override
    public void exitVars(Small_JavaParser.VarsContext ctx) {}

    @Override
    public void exitIdf(Small_JavaParser.IdfContext ctx) {...}
```


- *Quad:* on représente un quadruplé par un tableau à 4 éléments. En plus d'une méthode pour l'afficher.



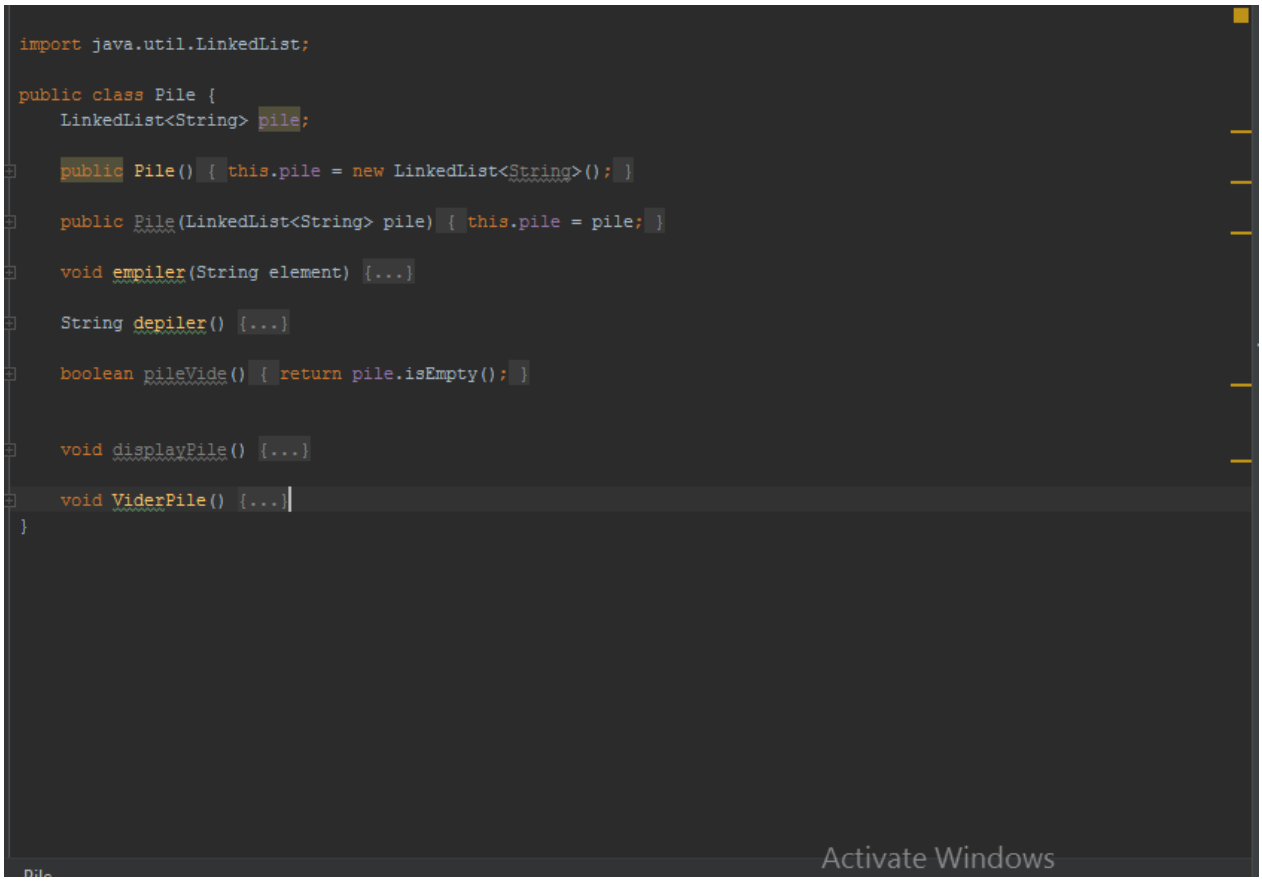
```java
public class Quad
{
    String Values[];

    public Quad(String[] Values) { this.Values = Values; }

    public Quad(String s1,String s2,String s3,String s4)
    {
        Values = new String[4];
        Values[0] = s1;
        Values[1] = s2;
        Values[2] = s3;
        Values[3] = s4;
    }

    public String get(int index) { return Values[index]; }

    public void set(int index, String s) { Values[index] = s; }

    @Override
    public String toString() { return "("+Values[0]+","+Values[1]+","+Values[2]+","+Values[3]+")"; }
}
```


- ***Quads :*** Ici on y présente la liste des quadruplés. On dispose des méthodes : affichage des quadruplé, ajout d'un quadruplé et le nombre de quadruplé existant.

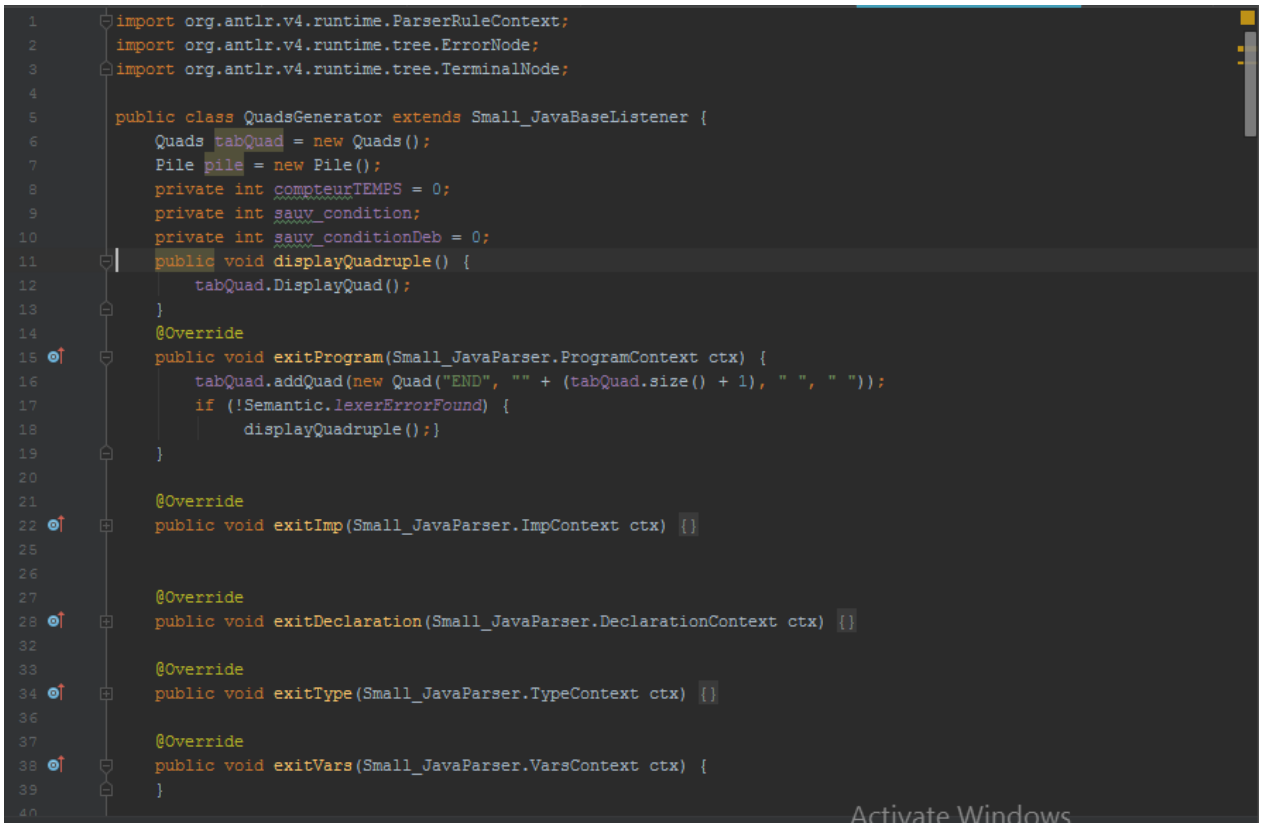


```java
import java.util.LinkedList;

public class Quads
{
    LinkedList<Quad> quads = new LinkedList<>();
    public int addQuad(String s1,String s2,String s3,String s4) { return addQuad(new Quad(s1,s2,s3,s4)); }

    public int addQuad(Quad quad)
    {
        quads.add(quad);
        return quads.size()-1;
    }

    public Quad getQuad(int index) { return quads.get(index); }

    public int size() { return quads.size(); }


    public void DisplayQuad() {
    // pour chaque element on le rend un toString et aprés j'affiche
    System.out.println("---------->: QUADRUPLE :<---------\n");
    quads.stream().map(Quadruple->" Q"+quads.indexOf(Quadruple)+" : | "+Quadruple.toString()+" ").forEach(System.out::
    System.out.println("_____\n");

}
}
```

- ***Pile :*** Cette classe a été créé pour pouvoir manipuler les temporaires lors de la compilation. Pour les méthodes on citera : empiler, dépiler, pileVide, ...etc.

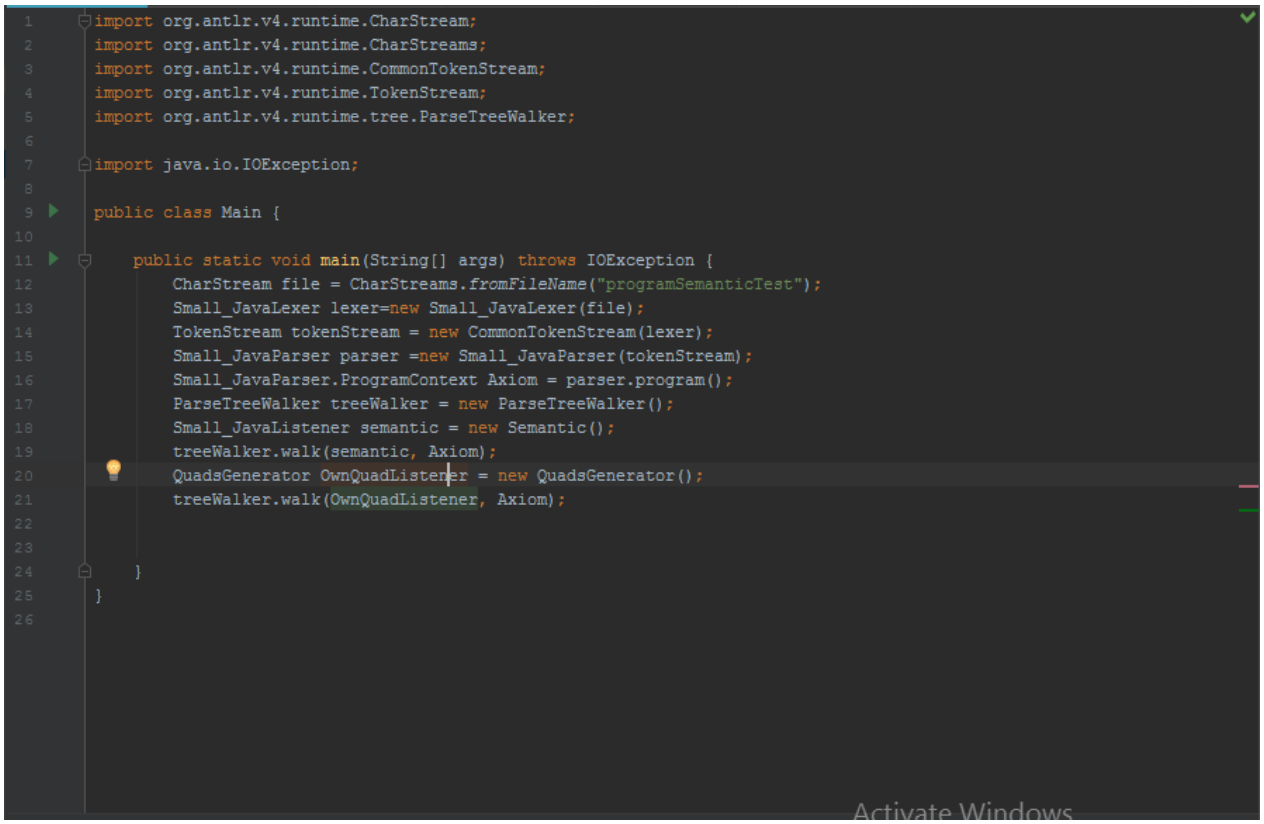


- Les classes Quad, Quads et Pile seront aussi utilisés dans la classe **QuandGenerator** pour implémenter les deux méthodes **enter__()** et **exit__()**.



```java
import org.antlr.v4.runtime.ParserRuleContext;
import org.antlr.v4.runtime.tree.ErrorNode;
import org.antlr.v4.runtime.tree.TerminalNode;

public class QuadsGenerator extends Small_JavaBaseListener {
    Quads tabQuad = new Quads();
    Pile pile = new Pile();
    private int compteurTEMPS = 0;
    private int sauv_condition;
    private int sauv_conditionDeb = 0;
    public void displayQuadruple() {
        tabQuad.DisplayQuad();
    }
    @Override
    public void exitProgram(Small_JavaParser.ProgramContext ctx) {
        tabQuad.addQuad(new Quad("END", "" + (tabQuad.size() + 1), " ", " "));
        if (!Semantic.lexerErrorFound) {
            displayQuadruple();}
    }

    @Override
    public void exitImp(Small_JavaParser.ImpContext ctx) {}


    @Override
    public void exitDeclaration(Small_JavaParser.DeclarationContext ctx) {}

    @Override
    public void exitType(Small_JavaParser.TypeContext ctx) {}

    @Override
    public void exitVars(Small_JavaParser.VarsContext ctx) {
    }
```


➢ Ensuite nous avons implémenté la classe Main :



```java
import org.antlr.v4.runtime.CharStream;
import org.antlr.v4.runtime.CharStreams;
import org.antlr.v4.runtime.CommonTokenStream;
import org.antlr.v4.runtime.TokenStream;
import org.antlr.v4.runtime.tree.ParseTreeWalker;

import java.io.IOException;

public class Main {

    public static void main(String[] args) throws IOException {
        CharStream file = CharStreams.fromFileName("programSemanticTest");
        Small_JavaLexer lexer=new Small_JavaLexer(file);
        TokenStream tokenStream = new CommonTokenStream(lexer);
        Small_JavaParser parser =new Small_JavaParser(tokenStream);
        Small_JavaParser.ProgramContext Axiom = parser.program();
        ParseTreeWalker treeWalker = new ParseTreeWalker();
        Small_JavaListener semantic = new Semantic();
        treeWalker.walk(semantic, Axiom);
        QuadsGenerator OwnQuadListener = new QuadsGenerator();
        treeWalker.walk(OwnQuadListener, Axiom);


    }
}
```


Résultat d’exécution :

```
"C:\Program Files\Java\jdk1.8.0_221\bin\java.exe" ...
program compiled without errors!
symbols table:
-------------------------------------------------------------------------
|      var          |     Type      |     Declared     |   Defined    |
-------------------------------------------------------------------------
|  Small_Java.io    | Library       | Declared         | Defined      |
-------------------------------------------------------------------------
|  Small_Java.lang  | Library       | Declared         | Defined      |
-------------------------------------------------------------------------
| a                 | int_SJ        | Declared         | Defined      |
-------------------------------------------------------------------------
| b                 | float_SJ      | Declared         | Defined      |
-------------------------------------------------------------------------
| c                 | int_SJ        | Declared         | Undefined    |
-------------------------------------------------------------------------
--------->: QUADRUPLE :<--------

 Q0 : | (+,5,a,Temp1)
 Q1 : | (:=,,Temp1,a)
 Q2 : | (-,4,3,Temp2)
 Q3 : | (:=,,Temp2,a)
 Q4 : | (*,b,a,Temp3)
 Q5 : | (:=,,Temp3,a)
 Q6 : | (BGE,10,a,b)
 Q7 : | (+,b,a,Temp4)
 Q8 : | (:=,,Temp4,a)
 Q9 : | (BR,12,,)
 Q10 : | (-,b,a,Temp5)
 Q11 : | (:=,,Temp5,a)
 Q12 : | (END,13, , )
```

III. Génération du code objet

Nous avons créé une classe Code_obj qui consiste a convertir les Quadruplets en code Assembleur :

```java
import java.util.ArrayList;

public class Code_obj {
    private Quadruple quad;
    private String inst="";

    private final static String MOV="MOV";
    private final static String ADD="ADD";
    private final static String SUB="SUB";
    private final static String MUL="MUL";
    private final static String DIV="DIV";
    private final static String CMP="CMP";
    private final static String JMP="JMP";
    private final static String JB="JB";
    private final static String JBE="JBE";
    private final static String JG="JG";
    private final static String JGE="JGE";
    private final static String JZ="JZ";
    private final static String JNZ="JNZ";
    private final static String AX="AX";
    private final static String BX="BX";
    private ArrayList<Integer> branchements=new ArrayList<>();

    public Code_obj(QuadrupleUses tabQuad) {

        for (int j=0;j<tabQuad.size();j++) {
            quad=tabQuad.getQuad(j);
            if (quad.get(0).startsWith("B")) {
                branchements.add(Integer.parseInt(quad.get(1)));
            }
```


Résultat d'exécution :

```
--------------------:> [OBJECT CODE] <:--------------------

MOV AX,5
ADD AX,a
MOV Temp1,AX
MOV AX,4
SUB AX,3
MOV Temp2,AX
MOV AX,b
MUL AX,a
MOVTemp3,AX
MOV AX,b
ADD AX,a
MOV Temp4,AX
JMP ETIQ 12
ETIQ10:
MOV AX,b
SUB AX,a
MOV Temp5,AX
ETIQ12:


_____________________________________________________________
```

Conclusion

Pour conclure, on peut dire que durant la réalisation de ce mini projet qui avait comme but la création d'un compilateur, on a pu découvrir un nouvel outil qui est ANTLR. Au fur et à mesure de l'avancement du projet, nous avons pu définir notre grammaire et générer l'analyseur lexicale et aussi syntaxique pour finir avec l'analyseur sémantique. Une fois ces étapes faites, on a pu générer les quadruplés et le code objet en langage assembleur.