

---

---

# Université des Sciences et de la Technologie Houari Boumediene

*U.S.T.H.B*

---

---

By

KADRI ADLANE	201400007288
MOUSSAOUI AMINA	201400007447
SAOULI HADJER	201400007202
CHENNIT NAILA	201400007519



Faculté d'Electronique et  
d'Informatique Département  
d'Informatique

**Projet de réalisation d'un mini-compilateur pour le langage  
TinyLanguage\_SII  
En utilisant ANTLR (ANother Tool for Language Recognition) & de génération  
du code objet**

Novembre 2017

Assistante de TP : Dr. F. MEKAHLIA

Spécialité : S.I.I

Module: compilation

## ● *Introduction générale*

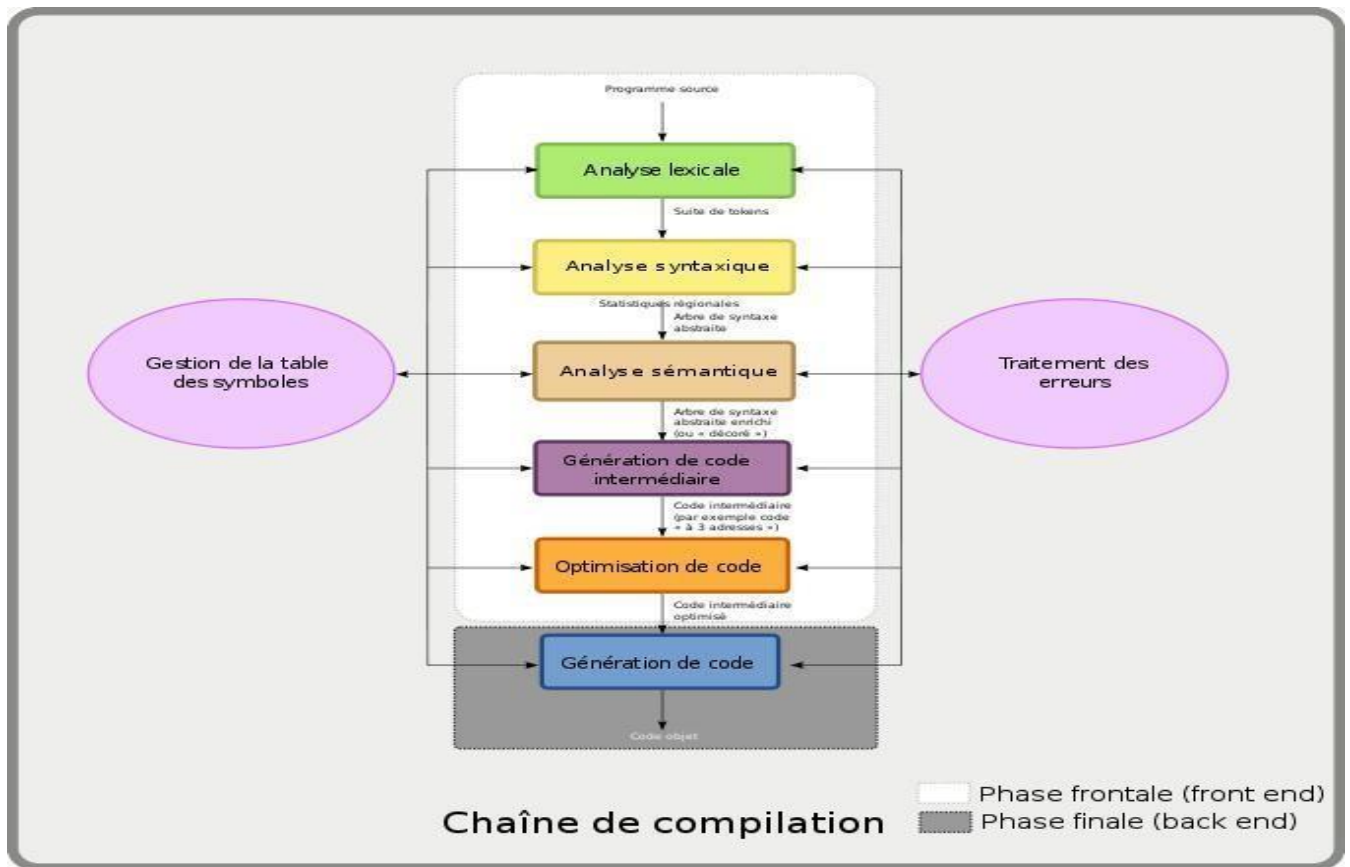
En informatique, un langage de programmation est une notation conventionnelle destinée à formuler des algorithmes 'un code' et produire des programmes informatiques qui les appliquent 'un besoin'. D'une manière similaire à une langue naturelle, un langage de programmation est composé d'un alphabet (***analyse lexical***), d'un vocabulaire (***analyse syntaxique***), de règles de grammaire et de significations (***analyse sémantique***).

Un langage est mis en œuvre par un traducteur automatique : **Compilateur** ou interprète. *Qu'est-ce qu'un compilateur?*

*Un compilateur* est un programme informatique qui transforme dans un premier temps un code source écrit dans un langage de programmation donné en un code cible qui pourra être directement exécuté par un ordinateur, à savoir un programme en langage machine ou en code intermédiaire, tandis que l'interprète réalise cette traduction, c'est à dire quelque chose que la machine peut exécuter directement, disons une suite d'entiers que le processeur de votre machine comprend comme des instructions. Et Alors d'après tout ça on peut bien comprendre que le compilateur effectue les opérations suivantes : ***analyse lexicale***, prétraitement (préprocesseur), ***analyse syntaxique*** (parsing), ***analyse sémantique***, et ***génération de code optimisé*** (améliorer la vitesse d'exécution). Quand le programme compilé (code objet) peut être exécuté sur un ordinateur dont le processeur ou le système d'exploitation est différent de celui du compilateur, on parle de compilation croisée. La compilation est souvent suivie d'une étape d'édition des liens, pour générer un fichier exécutable, et elle se décompose en deux phases:

1. Une phase d'analyse, qui va reconnaître les variables, les instructions, les opérateurs et élaborer la structure syntaxique du programme ainsi que certaines propriétés sémantiques.
2. Une phase de synthèse et de production qui devra produire le code cible.

La figure suivante montre la chaîne de compilation



### *Pourquoi réaliser un compilateur ?*

La compilation n'est pas limitée à la traduction d'un programme informatique écrit dans un langage de haut niveau en un programme directement exécutable par une machine, cela peut aussi être :

1. La traduction d'un langage de programmation de haut niveau vers un autre langage de programmation de haut niveau.
2. La traduction d'un langage de programmation de bas niveau vers un autre langage de programmation de haut niveau. Par exemple pour retrouver le code C à partir d'un code compilé (piratage, récupération de vieux logiciels, etc.).
3. La traduction d'un langage quelconque vers un autre langage quelconque (i.e. pas forcément de programmation)  
*Exemple* : Word vers html, PDF vers Ps, etc.

Pour réaliser un tel copulateur on peut utiliser différents Framework de travail, et parmi ces Framework on cite :

- **Lex/Yacc**,
- **Flex/Bison**,
- **Java Compiler**,
- **Free Compiler Construction Tools**
- **ANTLR, ANother Tool for Language Recognition**,

Dans ce travail nous avons choisi ANTLR comme un environnement et par la suite on va découvrir ce Framework



## *C'est quoi ANTLR ?*

*ANTLR*, sigle de ANother Tool for Language Recognition, est un Framework libre de construction de compilateurs utilisant une analyse LL(\*), ANTLR prend en entrée une grammaire définissant un langage et produit le code reconnaissant ce langage. La dernière version d'*ANTLR* permet de générer du code pour les langages Java, C#, Python2, Python3, JavaScript, C++, Go, Swift.

Dans sa dernière version, *ANTLR* peut supporter des grammaires utilisant de la récursivité gauche directe, mais pas indirecte.

*ANTLR* permet de générer des analyseurs lexicaux, syntaxiques ou des analyseurs lexicaux et syntaxiques combinés. Un analyseur syntaxique peut créer automatiquement un arbre syntaxique abstrait qui peut alors à son tour être traité par un analyseur d'arbre. *ANTLR* utilise une notation identique pour définir les différents types d'analyseurs, qu'ils soient lexicaux, syntaxiques, ou d'arbre. Des actions peuvent être assignées aux branches de l'arbre syntaxique abstrait ainsi obtenu. Ces actions peuvent être directement insérées dans la spécification de la grammaire utilisée, ou utilisés de façon découplée à travers un système de traversée d'arbres fourni par *ANTLR*.

Il existe un environnement d'utilisation graphique appelé AntlrWorks. ANTLR et AntlrWorks sont libres et open source. Plusieurs outils de développement peuvent être utilisés tel que : plug-ins for IntelliJ, NetBeans et Eclipse.

## *Comment utiliser ANTLR ?*

Pour utiliser ce Framework on va passer par la phase d'installation et la phase de la réalisation de notre Mini-Compilateur qui s'appelle *TinyLanguage\_SII(1)*, et pour cela on va expliquer tout d'abord *comment installer ANTLR, et comment l'utiliser*

### *Installation:*

Dans cette section on va expliquer comment installer ANTLR pour l'utiliser soit sur la l'invite de commande 'CMD', soit en utilisant l'IDE **IntelliJ**.

#### Remarque :

On a choisi à utiliser **IntelliJ** car il nous a facilité l'utilisation de ANTLR, mais au début on a fait les traitements sur l'invite de commande (sous-Windows), alors par la suit on va expliquer ce qu'on a fait comme traitement pour utiliser ANTLR sous-Windows et puis on vous donne comment l'utiliser sous-Linux (en cas de besoin), et on finit avec l'installation du plugin ANTLR sur **IntelliJ**.

# 1. L'INVITE DE COMMANDE 'CMD'

## Sous Windows :

1. On a Téléchargé « antlr-4.7-complete.jar » on l'a trouvé disponible sur ce site <http://www.antlr.org/download.html>
2. Sur l'invite de commande 'CMD' on a tapé la commande suivante `java -jar org.antlr.v4.Tool`

on a trouvé un *problème*, il nous a affiché le message

suivant **Error: Unable to access jarfile org.antlr.v4.Tool**

Et pour traiter ce problème on a du copier le chemin du **JAR** « antlr-4.7-complete.jar » dans la variable d'environnement « CLASSPATH » car le jar n'est pas reconnu pour le moment, et pour cela en suivant ces étapes :

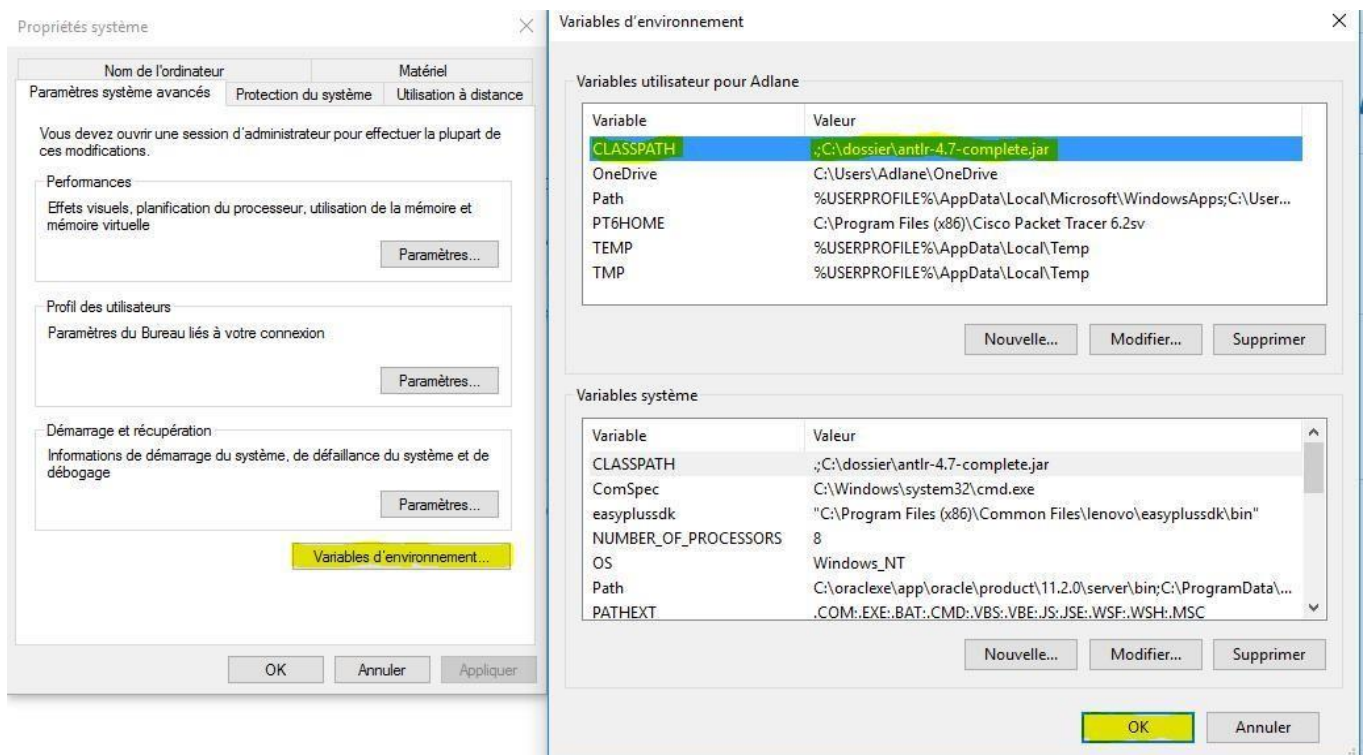
Panneau de configuration > Système et sécurité > Système > Paramètres système avancés > Variables d'environnement > Variables utilisateur > Nouvelle CLASSPATH>OK

Remarque: Si vous avez déjà la variable d'environnement «CLASSPATH » > Double clic sur 'CLASSPATH'



Valeur de la variable (on a ajouté '.;C:\dossier\antlr-4.7-complete.jar → .;Chemin\_De\_JAR )

Ces traitements sont montrés sur la figure suivante :










3. Après on a testé si ANTLR marche bien, on a tapé la même commande sur le CMD : `java -jar org.antlr.v4.Tool`

Et là on l'a trouvé marche bien

4. La phase de compilation (supposons qu'on a déjà notre langage)  
Pour traiter cette phase on a tapé sur le CMD la commande suivante  
`java -jar chemin_de_jar chemin_de_fichier_de_notre_langage`



Après l'exécution de cette commande, elle nous a généré quelque classe JAVA dans le répertoire de notre langage

	TinyLanguage_SII	19/11/2017 15:00	Fichier G4	1 Ko
	TinyLanguage_SII.tokens	29/12/2017 02:44	Fichier TOKENS	1 Ko
	TinyLanguage_SIIBaseListener	29/12/2017 02:44	Java source file	3 Ko
	TinyLanguage_SIILexer	29/12/2017 02:44	Java source file	5 Ko
	TinyLanguage_SIILexer.tokens	29/12/2017 02:44	Fichier TOKENS	1 Ko
	TinyLanguage_SIIListener	29/12/2017 02:44	Java source file	3 Ko
	TinyLanguage_SIIParser	29/12/2017 02:44	Java source file	12 Ko

5. Après cette dernière phase on a testé 'javac' en tapant ces commandes sur le CMD :

1/s'acheminer vers le dossier 'bin' :

```
cd C:\Program Files\Java\jdk1.8.0_71\bin
```

2/ taper:

```
javac -cp <path Of Compiler><path Of grammar* /"sans .g4">
```

Exemple:

```
javac -cp C:\antlr-4.7-complete.jar D:\X-Master1-sii\Compilation\Adlane\Tp\projet\TinyLanguage_SII*.java
```

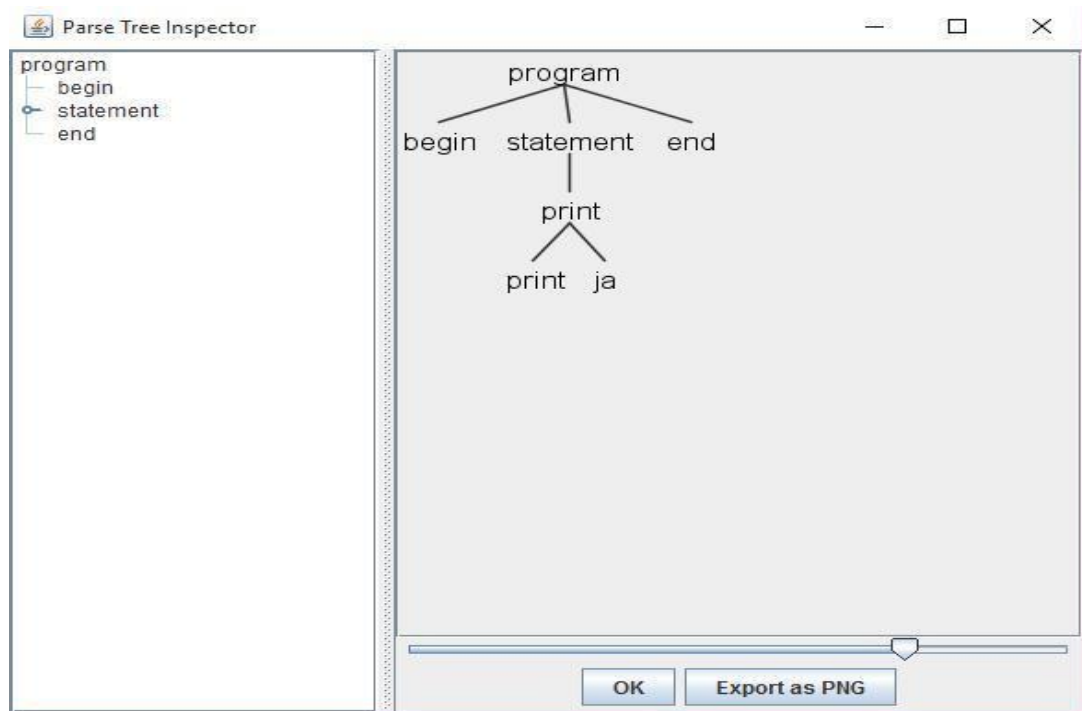
6. Après avoir testé javac, on est arrivé à l'étape pour tester notre grammaire et afficher l'arbre syntaxique, et pour faire les traitements de cette phase on a créé un fichier **.bat** qui a comme contenu '*java org.antlr.v4.runtime.misc.TestRig%\**' On l'a sauvegardé sous nom '**grun**'

Si on n'a pas créé ce fichier **.bat** qui a ce contenu et sous-nom 'grun', elle ne va pas marcher la commande qui nous affiche l'arbre

Après la création de fichier grun, on a lancé cette commande :

```
grun <notre grammaire><Axiom> -gui <chemin_Du_Program_A_Tester.txt>
```

Après l'exécution de cette dernière on a eu un affichage qui ressemble à ça :



## Sous Linux :

- [1] `cd /usr/local/lib`
- [2] `wget http://wwwantlr.org/download/antlr-4.7-complete.jar`
- [3] `export CLASSPATHChemin_De_JAR:$CLASSPATH"`
- [4] `java -jar Chemin_De_JAR`

## *INTELLIJ*

Afin d'utiliser IntelliJ, on l'a téléchargé, tous les IDE de JetBrains sont disponibles gratuitement sur le site officiel <https://www.jetbrains.com/> pour une durée d'une année pour une licence 'étudiant'.

Après avoir téléchargé l'IDE IntelliJ sur notre PC, on a entamé directement notre travail (*développement de notre compilateur*), mais après avoir implémenté notre langage, l'IDE IntelliJ ne l'a pas reconnu, dans un premier temps, notre langage était affiché comme suit:

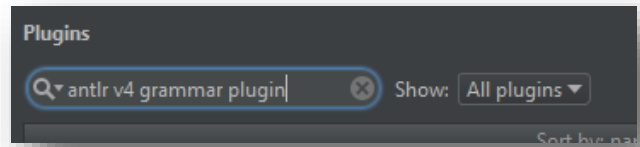
```
grammar TinyLanguage_SII;
|
program : 'Compil' NOM_PROGME '(' ')' '{' declaration+ 'start' instruction+ '}'
;
;
declaration : type variables ';'
;
;
type : 'intcompil' | 'floatcompil'
;
;
variables : NOM_VARIABLE ',' variables | NOM_VARIABLE
;
;
instruction : affectation | lecture | ecrisure | instruction_conditionnelle
;
;
affectation : NOM_VARIABLE '=' expression ';'
;
;
expression : expression op1 expression1 | expression1
;
;
op1 : '+' | '-'
;
;
expression1: expression1 op2 expression2 | expression2
;
;
op2 : '*' | '/'
;
;
expression2: '(' expression ')' | terme
;
;
terme : NOM_VARIABLE | NOMBRE_ENTIER | NOMBRE_REEL
;
;
```

Afin de résoudre 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. Cela fonctionne avec IntelliJ et devrait fonctionner dans d'autres IDE JetBrains.

Pour faire cette configuration sur IntelliJ on a suivi ces étapes :

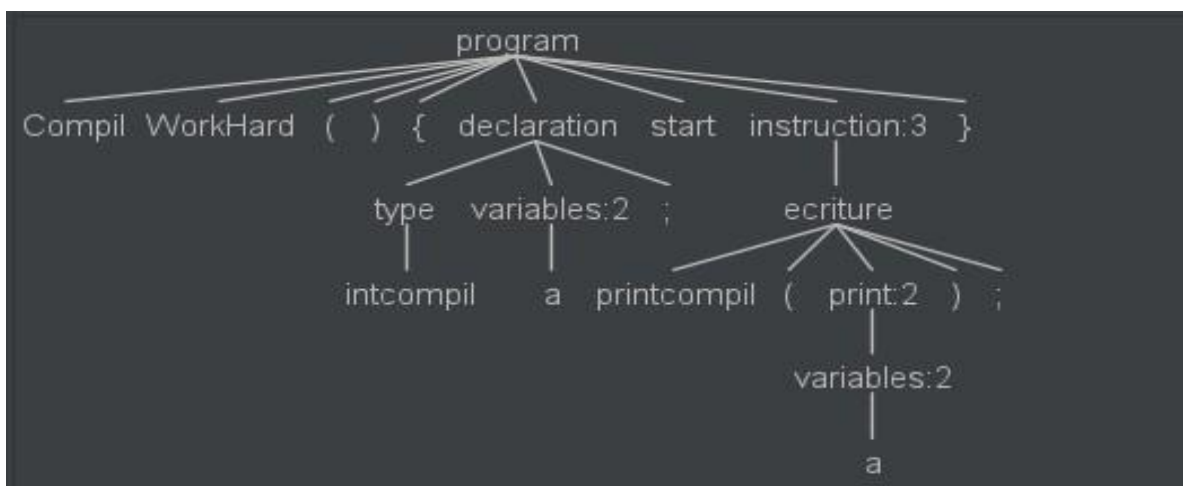
1. ► File ► Settings  
'ctrl+alt+s'
2. ► Plugins
3. Après on a l'a téléchargé



Après avoir téléchargé le plugin l'IDE IntelliJ a reconnu notre langage (*partie lexical-syntaxique*), il s'est affiché comme suit,

```
1 grammar TinyLanguage_SII;  
2  
3 program : 'Compil' NOM_PROGRAME '(' ')' '{' declaration+ 'start' instruction+ '}'  
4 ;  
5  
6 declaration : type variables ';' ;  
7  
8  
9 type : 'intcompil' | 'floatcompil'  
10 ;  
11  
12 variables : NOM_VARIABLE ',' variables | NOM_VARIABLE  
13 ;  
14  
15 instruction : affectation | lecture | ecriture | instruction_conditionnelle  
16 ;  
17  
18 affectation : NOM_VARIABLE '=' expression ';' ;  
19  
20  
21  
22 expression : expression op1 expression1 | expression1  
23 ;  
24  
25 op1 : '+' | '-'  
26 ;  
27  
28 expression1: expression1 op2 expression2 | expression2  
29 ;  
30  
31 op2 : '*' | '/'  
32 ;  
33 expression2: '(' expression ')' | terme  
34 ;  
35  
36 terme : NOM_VARIABLE | NOMBRE_ENTIER | NOMBRE_REEL  
37 ;
```

Maintenant on a pu afficher l'arbre (*la figure suivante montre une petite partie de l'arbre généré à travers notre langage*) :





## ● *Présentation des Différentes Phases de notre Compilateur*

Dans cette partie nous allons aborder l'analyse lexicale, puis détailler les étapes de l'analyse syntaxique, sémantique et enfin finir avec la génération du code objet de notre compilateur.

### **I. Analyse Lexical & Syntaxique:**

#### ***C'est quoi l'analyse lexical et Syntaxique ?***

Les deux analyses (lexicales et syntaxiques) utilisent de façon essentielle les automates, mais on retrouve aussi les automates dans de nombreux domaines de l'informatique. L'analyse lexicale s'explique dans le cadre restreint des automates finis et des expressions régulières (le terme français « autorisé » est expression rationnelle, mais je préfère adapter la terminologie anglaise). Les expressions régulières sont utilisées dans de nombreux outils Unix (éditeur de textes, commande `grep` etc.), et fournies en bibliothèque dans la plupart des langages de programmation. Pour l'analyse lexicale de notre langage on a utilisé les **langages formels** (1) et les **expressions régulières** (2) (identificateur, mots clés, nombres...etc.).

#### **Définition d'un Langage Formel :**

On se donne un ensemble  $\Sigma$  appelé alphabet, dont les éléments sont appelés caractères. Un mot (sur  $\Sigma$ ) est une séquence de caractères (de  $\Sigma$ ). On note  $\epsilon$  le mot vide,  $uv$  la concaténation des mots  $u$  et  $v$  (la concaténation est associative avec  $\epsilon$  pour élément neutre). On note  $\Sigma^*$  l'ensemble des mots sur  $\Sigma$ .

#### **Définition d'une Expression Régulière :**

une *expression régulière* ou *expression normale*<sup>1</sup> ou *expression rationnelle* ou *motif*, est une chaîne de caractères, qui décrit, selon une syntaxe précise, un ensemble de chaînes de caractères possibles

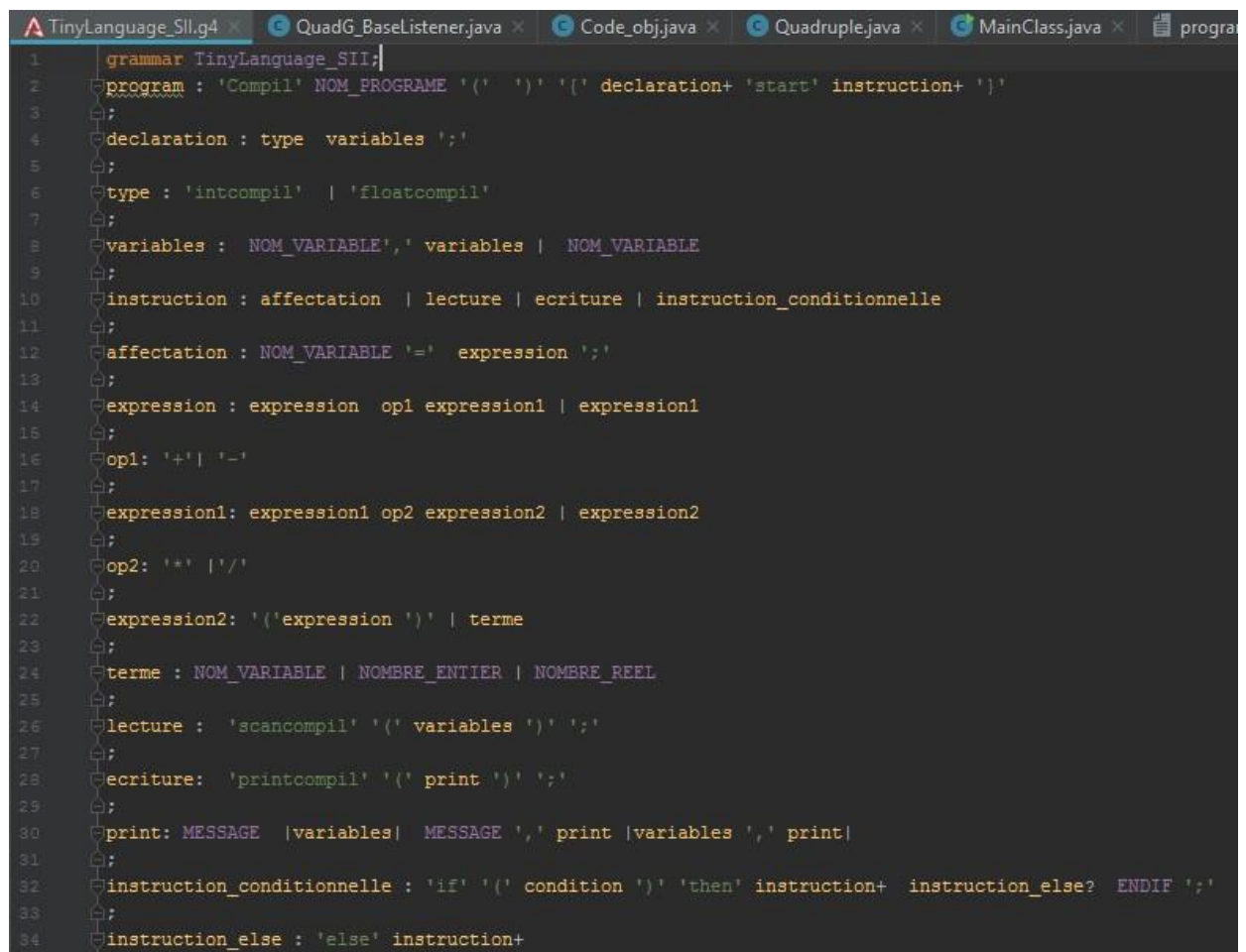
Alors l'analyse lexicale est la première phase de la chaîne de compilation. Elle consiste à convertir une chaîne de caractères en une liste de symboles. Ces symboles sont ensuite consommés lors de l'analyse syntaxique.

L'analyse syntaxique consiste à mettre en évidence la structure d'un texte, généralement une phrase écrite dans une langue naturelle, mais on utilise également cette terminologie pour l'analyse d'un programme informatique.

Comme on a dit, l'*analyse syntaxique* fabrique l'*arbre de syntaxe* abstraite à partir des lexèmes (mots-clés, variable, symbole...etc.) produits par l'analyse lexicale. L'arbre de syntaxe abstraite est important, car il est le support de la sémantique du langage. Il importe donc, pour comprendre ce que fait exactement un programme, de bien comprendre d'abord comment son source s'explique en termes de syntaxe abstraite.

La capture se dessous représente *la grammaire de notre TINYLANGUAGE\_SII*. Nous avons pris en compte les *mots clés*, les différentes instructions de bases et complexes ainsi que les priorités entre les opérateurs. Enfin nous avons testé notre grammaire avec une portion de code et avons généré l'arbre syntaxique.

Pour cela nous lui avons donné *l'extension g4*.



```
1 grammar TinyLanguage_SII;
2 program : 'Compil' NOM_PROGRAME '(' ')' '[' declaration+ 'start' instruction+ ']'
3 ;
4 declaration : type variables ';'
5 ;
6 type : 'intcompil' | 'floatcompil'
7 ;
8 variables : NOM_VARIABLE ',' variables | NOM_VARIABLE
9 ;
10 instruction : affectation | lecture | ecriture | instruction_conditionnelle
11 ;
12 affectation : NOM_VARIABLE '=' expression ';'
13 ;
14 expression : expression op1 expression1 | expression1
15 ;
16 op1 : '+' | '-'
17 ;
18 expression1 : expression1 op2 expression2 | expression2
19 ;
20 op2 : '*' | '/'
21 ;
22 expression2 : '(' expression ')' | terme
23 ;
24 terme : NOM_VARIABLE | NOMBRE_ENTIER | NOMBRE_REEL
25 ;
26 lecture : 'scancompil' '(' variables ')' ';'
27 ;
28 ecriture : 'printcompil' '(' print ')' ';'
29 ;
30 print : MESSAGE | variables | MESSAGE ',' print | variables ',' print
31 ;
32 instruction_conditionnelle : 'if' '(' condition ')' 'then' instruction+ instruction_else? 'ENDIF' ';'
33 ;
34 instruction_else : 'else' instruction+
```

```

33  ⌞;
34  ⌞instruction_else : 'else' instruction+
35  ⌞;
36  ⌞condition:  expression compareteur expression
37  ⌞;
38  ⌞compareteur : '<' | '>' | '==' | '<=' | '>=' | '!='
39  ⌞;
40  ⌞NOM_PROGRAMME : [A-Z][a-zA-Z0-9]*
41  ⌞;
42  ⌞ENDIF:'endif'
43  ⌞;
44  ⌞NOM_VARIABLE : [a-zA-Z][a-zA-Z0-9]*
45  ⌞;
46  ⌞NOMBRE_ENTIER : [0]|[1-9][0-9]*
47  ⌞;
48  ⌞NOMBRE_REEL:([0]|[1-9][0-9]*)'.'[0-9]*[1-9] | [1-9][0-9]*
49  ⌞;
50  ⌞MESSAGE : '''(~["|'\\''')*'''
51  ⌞;
52  ⌞COMMENT : '/*' .*? '*/' -> channel(HIDDEN)
53  ⌞;
54  ⌞LINE_COMMENT : '//'' ~'\n'* '\n' -> channel(HIDDEN)
55  ⌞;
56  ⌞WS : [ \t\n\r]+ -> channel(HIDDEN)
57  ⌞;
58

```

Pour tester nous avons écrit cette petite portion de code :

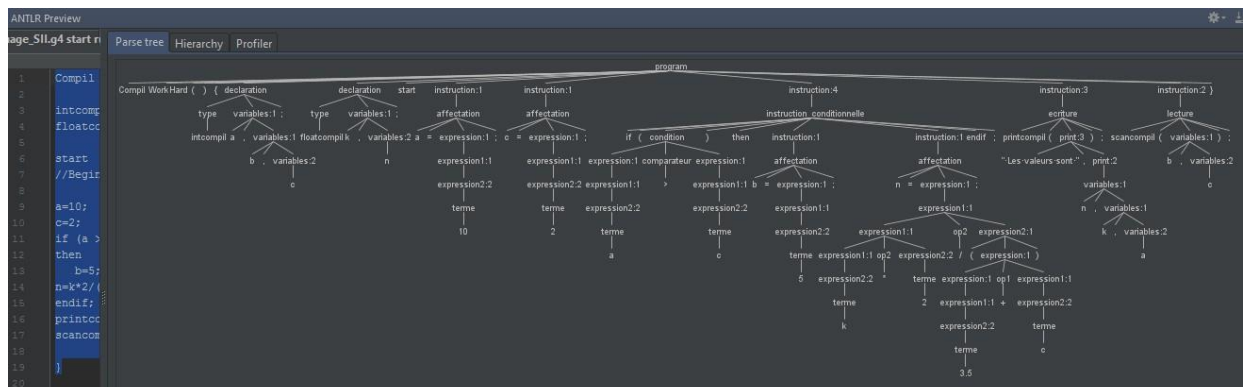
```

1  ⌞ Compil WorkHard() {
2
3      intcompil a,b,c;
4      floatcompil k,n;
5
6      start
7      //Begin Of my main()
8
9      a=10;
10     c=2;
11     if (a > c)
12     then
13         b=5;
14         n=k*2/(3.5+c);
15     endif;
16     printcompil(" Les valeurs sont ", n,k,a);
17     scancompil(b,c);
18
19 }

```

Figure 1 Portion de code test

Arbre syntaxique :



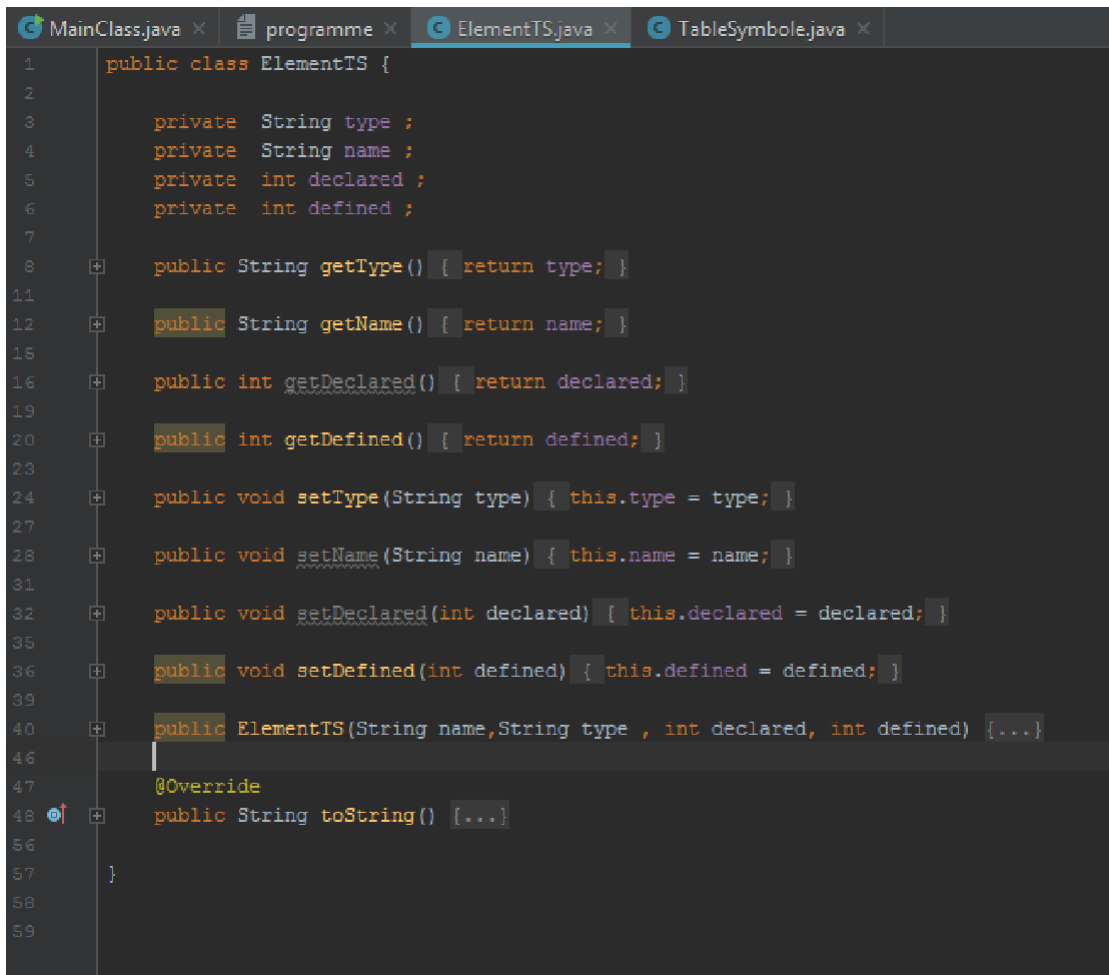
## II. Analyse Sémantique :

L'**analyse sémantique** d'un message est la phase de son analyse qui en établit la signification en utilisant le sens des éléments (mots) du texte, par opposition aux analyses lexicales ou syntaxiques qui décomposent le message à l'aide d'un lexique ou d'une grammaire.

En compilation, l'analyse sémantique est la phase intervenant 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 abstrait et construit la table des symboles, tous ces traitements ils sont expliqué par la suite,

Pour réaliser l'analyse sémantique nous avons défini des classes java telles que :

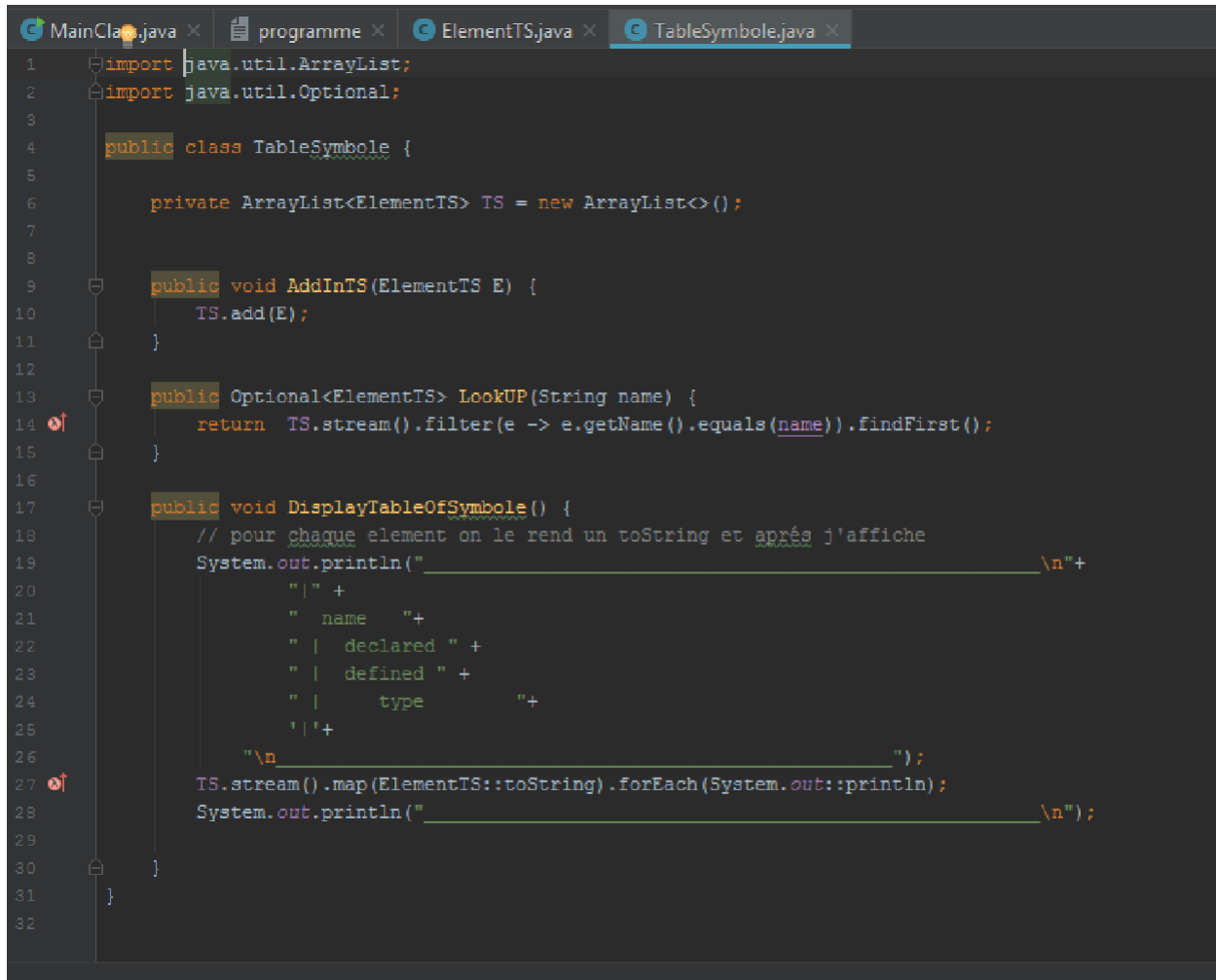
- **ElementTS** : Elle représente un élément de la table de symbole avec les attributs type , nom , s'il est déclaré ou non ,s'il est défini ou non . Sans oublier les getters ,setters et constructeur.



```
1 public class ElementTS {
2
3     private String type ;
4     private String name ;
5     private int declared ;
6     private int defined ;
7
8     public String getType() { return type; }
9
10
11     public String getName() { return name; }
12
13
14     public int getDeclared() { return declared; }
15
16     public int getDefined() { return defined; }
17
18     public void setType(String type) { this.type = type; }
19
20     public void setName(String name) { this.name = name; }
21
22     public void setDeclared(int declared) { this.declared = declared; }
23
24     public void setDefined(int defined) { this.defined = defined; }
25
26     public ElementTS(String name,String type , int declared, int defined) {...}
27
28     @Override
29     public String toString() {...}
30
31 }
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
```

- **TableSymbole** : Notre table symbole qui est une liste d'élément de type ElementTS. Nous avons défini des méthodes pour

ajouter un element dans la table, afficher la table. Pour rechercher un element dans la table de symbole nous avons utilisé la classe qui utilise *des concepts de programmation fonctionnelle* « *Optional* » .....⊗.



```
1  import java.util.ArrayList;
2  import java.util.Optional;
3
4  public class TableSymbole {
5
6      private ArrayList<ElementTS> TS = new ArrayList<>();
7
8
9      public void AddInTS(ElementTS E) {
10         TS.add(E);
11     }
12
13     public Optional<ElementTS> LookUP(String name) {
14         return TS.stream().filter(e -> e.getName().equals(name)).findFirst();
15     }
16
17     public void DisplayTableOfSymbole() {
18         // pour chaque element on le rend un toString et après j'affiche
19         System.out.println("_____ \n"+
20             " | " +
21             "  name  "+
22             " |  declared " +
23             " |  defined " +
24             " |    type    "+
25             " |'+
26             " \n_____");
27         TS.stream().map(ElementTS::toString).forEach(System.out::println);
28         System.out.println("_____ \n");
29     }
30 }
31
32
```

- **Erreur :** Nous avons défini toutes les erreurs possibles. Chaque fois que notre compilateur en détecte une erreur il l'ajoute à la liste d'erreur. Sans oublier la méthode d'affichage de toutes les erreurs du programme.

```

MainClass.java x Erreur.java x TinyLanguage_SII.g4 x programme x ElementTS.java x TableSymbole.java x
1 import java.util.ArrayList;
2
3 public class Erreur {
4     private final static String DOUBLE_DEF = "DOUBLE DECLARATION DE LA VARIABLE : ";
5     private final static String INCOMPATIBILITE_TYPE = "INCOMPATIBILITE DE TYPE DANS L'EXPRESSION : ";
6     private final static String UNDECLARED_ID = "VARIABLE NON DECLAREE : ";
7     private final static String SYNTAX_ERROR = "ERREUR SYNTAXIQUE : ";
8     private final static String UNDEFINED = "VARIABLE N'A PAS DE VALEUR : ";
9     private final static String INCOMPATIBILITE_TYPE_OF_CONDITION = "INCOMPATIBILITE DE TYPE DANS LA CONDITION : ";
10    private ArrayList<String> listeErreur;
11
12
13    public Erreur() { listeErreur = new ArrayList<>(); }
14
15
16
17    void Double_def(String var) {...}
18    void unDefined(String var) {...}
19    void syntax_error(String var) {...}
20    void Type_incompatible(String exp) {...}
21    void typeConditionincompatible(String exp) {...}
22    void notDeclared(String var) {...}
23
24    public boolean No_Errors() {
25        return listeErreur.isEmpty();
26    }
27
28    public void Display_Errors() {
29        listeErreur.
30            stream().
31            map(error -> "Error N°" + listeErreur.indexOf(error) + " : " + error).
32            forEach(System.out::println);
33    }
34
35 }

```

Les classes vues précédemment ont été utilisées dans notre Listener qui redéfinit les méthodes `exitInstruction()` , `enterInstruction()`..

```

MainClass.java x programme x Pile.java x QuadG_BaseListener.java x Quadruple.java x QuadrupleUses.java x SII_Listener.java x
1 import org.antlr.v4.runtime.ParserRuleContext;
2 import org.antlr.v4.runtime.tree.ErrorNode;
3 import org.antlr.v4.runtime.tree.TerminalNode;
4
5 import java.util.HashMap;
6 import java.util.Optional;
7
8 import static java.lang.System.out;
9
10 class SII_Listener extends TinyLanguage_SIIBaseListener {
11     // private static int undeclared=0;
12     private static int defined = 1;
13     private TableSymbole ts = new TableSymbole();
14     private Erreur erreur = new Erreur();
15     private HashMap<ParserRuleContext, String> types = new HashMap<>();
16     private Printer printer = new Printer();
17
18
19     @Override
20     public void exitProgram(TinyLanguage_SIIParser.ProgramContext ctx) {...}
21
22
23     @Override
24     public void exitDeclaration(TinyLanguage_SIIParser.DeclarationContext ctx) {...}
25
26
27     @Override
28     public void enterInstruction(TinyLanguage_SIIParser.InstructionContext ctx) {}
29
30
31     @Override
32     public void exitAffectation(TinyLanguage_SIIParser.AffectationContext ctx) {...}
33
34
35     @Override
36     public void exitExpression(TinyLanguage_SIIParser.ExpressionContext ctx) {...}
37
38
39 }

```



```
286
287     @Override
288     public void exitEveryRule(ParserRuleContext ctx) {
289     }
290
291     @Override
292     public void visitTerminal(TerminalNode node) {
293     }
294
295     @Override
296     public void visitErrorNode(ErrorNode node) { erreur.syntax_error(node.getParent().getText()); }
297
298
299
300
301     /***** USING THIS METHODS *****/
302     void ajouterType(ParserRuleContext ctx, String type) { types.put(ctx, type); }
303
304     String getType(ParserRuleContext ctx) {
305         return types.get(ctx);
306     }
307
308     Optional<String> typeCompatible(String type_id, String type_exp) {...}
309
310     Optional<String> typePriority(String type_exp1, String type_exp2) {...}
311
312     public boolean anySemanticErrors() { return erreur.No_Errors(); }
313
314     void displayErrors() {...}
315
316     void displaySymbolTable() {...}
317
318     void clearTypeHashMap() { types.clear(); }
319 }
320
321
```

Pour tester la qualité de notre travail nous avons utilisé la portion de code suivante :

```
1  Compil Program_Sem() {
2
3  intcompil a,b,b,c,l,p;
4  floatcompil k,d;
5
6  start
7
8  a=2; k=3.2;
9  l=5.5;
10 if (a==k)
11 then
12
13  b=b+1;
14
15 else
16  a=r;
17 endif;
18
19
20 if (a> 1.5) then
21  printcompil (d,a);
22
23 endif;
24
25
26 }
27
28
```

Ensuite nous avons implémenté la classe main comme on le voit ci-dessous :

```

1  import org.antlr.v4.runtime.*;
2  import org.antlr.v4.runtime.tree.ParseTreeWalker;
3
4  import java.io.IOException;
5  import javax.print.PrintException;
6  import java.lang.reflect.InvocationTargetException;
7  import java.lang.reflect.Method;
8  import java.util.Arrays;
9  import java.util.Iterator;
10
11 public class MainClass {
12     private static int channel;
13     public static void main(String[] args) throws IOException {
14         CharStream file = CharStreams.fromFileName("programme");
15         TinyLanguage_SIILexer lexer = new TinyLanguage_SIILexer(file);
16         TokenStream tokenStream = new CommonTokenStream(lexer);
17         TinyLanguage_SIIParser parser = new TinyLanguage_SIIParser(tokenStream);
18         TinyLanguage_SIIParser.ProgramContext Axiom = parser.program();
19         ParseTreeWalker treeWalker = new ParseTreeWalker();
20         //TinyLanguage_SIIBaseListener listener = new TinyLanguage_SIIBaseListener();
21         //treeWalker.walk(listener, Axiom);
22
23         SII_Listener OwnSemanticListenr = new SII_Listener();
24         treeWalker.walk(OwnSemanticListenr, Axiom);
25
26         // QuadG_BaseListener OwnQuadListener = new QuadG_BaseListener();
27         //treeWalker.walk(OwnQuadListener, Axiom);
28
29         //*****
30     }
31 }

```

### Remarque :

Dans notre projet ‘code’ on a écrit des commentaires partout, même dans la classe main (pour cette dernière capture d’écran, on n’a pas affiché le commentaire pour une meilleure visibilité)

On a utilisé trois versions de programme à tester, la figure suivante montre les trois versions :

```

//pour recuperer le programme a tester

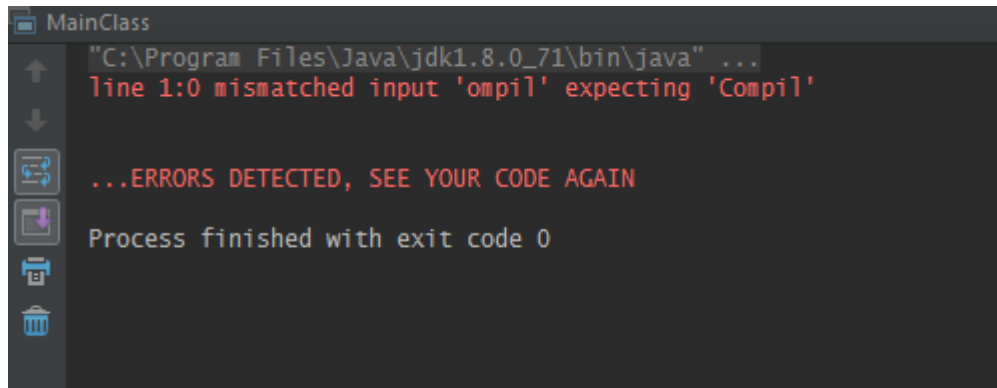
/** ***** */
/**          //Programme with errors lexer,          //
/**          //CharStream file = CharStreams.fromFileName("programLexerTest");          //
/**          //          //
/**          //          //
/**          // Programme With Semantic errors          //
/**          //CharStream file = CharStreams.fromFileName("programSemanticTest");          //
/**          //          //
/**          //          //
/**          // a programme --> no errors          //
/**          // CharStream file = CharStreams.fromFileName("programNoErrors");          //
/**          //          //
/** ***** */

```

## Résultat d'exécution :

Avec un programme qui a des erreurs lexicales ou syntaxiques «*programLexerTest*», on n'affiche que la première erreur :

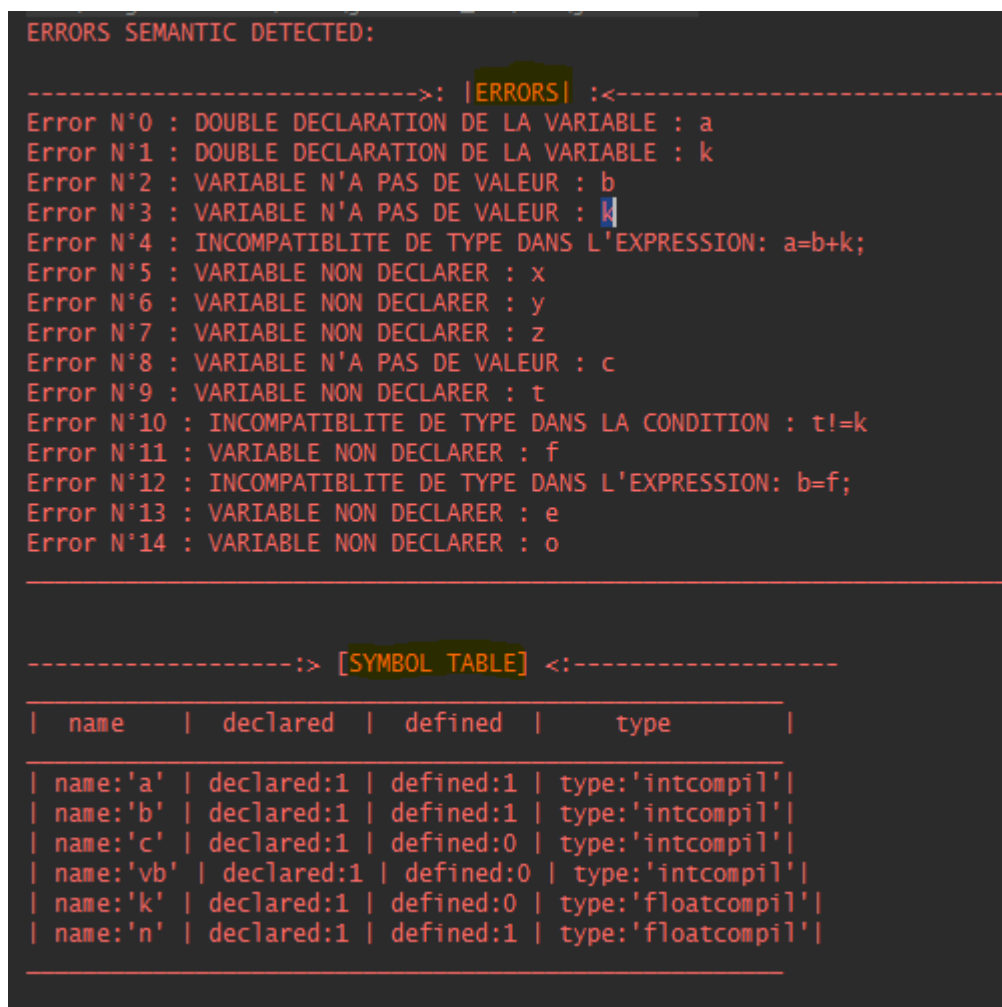
L'erreur : *program* commence avec 'ompil' mais cette entité lexicale n'existe pas de notre langage, en plus syntaxiquement faux car notre langage commence avec 'Compil'



The screenshot shows an IDE window titled 'MainClass'. The console output is as follows:

```
"C:\Program Files\Java\jdk1.8.0_71\bin\java" ...  
line 1:0 mismatched input 'ompil' expecting 'Compil'  
  
...ERRORS DETECTED, SEE YOUR CODE AGAIN  
  
Process finished with exit code 0
```

Avec un programme qui a des erreurs sémantique «*programSemanticTest*», on n'affiche que la première erreur :



The screenshot shows an IDE window displaying semantic errors and a symbol table. The output is as follows:

```
ERRORS SEMANTIC DETECTED:  
  
----->: |ERRORS| :<-----  
Error N°0 : DOUBLE DECLARATION DE LA VARIABLE : a  
Error N°1 : DOUBLE DECLARATION DE LA VARIABLE : k  
Error N°2 : VARIABLE N'A PAS DE VALEUR : b  
Error N°3 : VARIABLE N'A PAS DE VALEUR : k  
Error N°4 : INCOMPATIBILITE DE TYPE DANS L'EXPRESSION: a=b+k;  
Error N°5 : VARIABLE NON DECLARER : x  
Error N°6 : VARIABLE NON DECLARER : y  
Error N°7 : VARIABLE NON DECLARER : z  
Error N°8 : VARIABLE N'A PAS DE VALEUR : c  
Error N°9 : VARIABLE NON DECLARER : t  
Error N°10 : INCOMPATIBILITE DE TYPE DANS LA CONDITION : t!=k  
Error N°11 : VARIABLE NON DECLARER : f  
Error N°12 : INCOMPATIBILITE DE TYPE DANS L'EXPRESSION: b=f;  
Error N°13 : VARIABLE NON DECLARER : e  
Error N°14 : VARIABLE NON DECLARER : o  
  
----->: [SYMBOL TABLE] <:-----  
  
| name | declared | defined | type |  
| name:'a' | declared:1 | defined:1 | type:'intcompil'|  
| name:'b' | declared:1 | defined:1 | type:'intcompil'|  
| name:'c' | declared:1 | defined:0 | type:'intcompil'|  
| name:'vb' | declared:1 | defined:0 | type:'intcompil'|  
| name:'k' | declared:1 | defined:0 | type:'floatcompil'|  
| name:'n' | declared:1 | defined:1 | type:'floatcompil'|
```

Avec un programme qui n'a aucune erreur «*programNoErrors*», on n'affiche que la première erreur :

```

BRAVO/ NO ERROR DETECTED!
-----> [SYMBOL TABLE] <:-----

| name | declared | defined | type |
|-----|-----|-----|-----|
| name:'a' | declared:1 | defined:1 | type:'intcompil'|
| name:'b' | declared:1 | defined:1 | type:'intcompil'|
| name:'c' | declared:1 | defined:1 | type:'intcompil'|
| name:'x' | declared:1 | defined:1 | type:'floatcompil'|
| name:'y' | declared:1 | defined:1 | type:'floatcompil'|
| name:'z' | declared:1 | defined:1 | type:'floatcompil'|
|-----|-----|-----|-----|

----->: QUADRUPLE :<-----

Q0 : | ( = , , 10 , a )
Q1 : | ( + , a , 8 , Temp1 )
Q2 : | ( = , , Temp1 , b )
Q3 : | ( * , 8 , a , Temp2 )
Q4 : | ( + , Temp2 , b , Temp3 )
Q5 : | ( = , , Temp3 , c )
Q6 : | ( * , b , a , Temp4 )
Q7 : | ( / , 4 , Temp4 , Temp5 )
Q8 : | ( + , Temp5 , c , Temp6 )
Q9 : | ( = , , Temp6 , x )
Q10 : | ( * , a , x , Temp7 )
Q11 : | ( + , Temp7 , c , Temp8 )
Q12 : | ( + , 3 , Temp8 , Temp9 )
Q13 : | ( = , , Temp9 , y )
Q14 : | ( BGE , 18 , x , a )
Q15 : | ( * , 2 , x , Temp10 )
Q16 : | ( = , , Temp10 , z )
Q17 : | ( BR , 20 , , )
Q18 : | ( * , 2 , a , Temp11 )
Q19 : | ( = , , Temp11 , z )
Q20 : | ( END , 21 , , )

```

Pour la génération des quadruples nous avons procédé comme suit.  
D'abord avec en définissant les classes suivantes :

- *Quadruple* : Elle représente un quadruple par un tableau de chaîne de caractères de 4 colonnes.

```
MainClass.java x programme x Pile.java x QuadG_BaseListener.java x Quadruple.java x QuadrupleUses.java x
1 import java.util.ArrayList;
2 import java.util.Arrays;
3
4 public class Quadruple {
5
6
7     //private ArrayList<String> quad = new ArrayList<>();
8
9     private String[] quad;
10    public Quadruple(String q1, String q2, String q3, String q4) {...}
11
12
13
14
15
16
17
18
19
20
21
22
23    public Quadruple(String[] quad) {...}
24
25
26
27
28    public String get(int index) { return quad[index]; }
29
30
31
32
33    public void set(int index, String q) { quad[index] = q; }
34
35
36
37
38
39
40
41    @Override
42    public String toString() { return " (" +quad[0]+" , "+quad[1]+" , "+quad[2]+" , "+quad[3]+" )"; }
43
44
45    public void displayQuadruple() { System.out.println(); }
46
47
48
49
50    public int size() { return this.size(); }
51
52
53 }
54
```

- **QuadrupleUses** :Elle représente la liste des quadruples , et dispose de méthodes telles que l'ajout d'un quad , l'affichage de tous les quads , et le nombre de quad déjà existant.

```

1  import java.util.ArrayList;
2
3  public class QuadrupleUses {
4
5      ArrayList<Quadruple> quadruples = new ArrayList<>();
6
7
8      public QuadrupleUses() {}
9
10     public int addQuadruple(String q1, String q2, String q3, String q4)
11     {...}
12
13
14     public int addQuadruple(Quadruple quadruple)
15     {...}
16
17
18
19
20
21
22
23
24     public Quadruple getQuad(int qc) { return quadruples.get(qc); }
25
26
27
28
29     public int size() { return quadruples.size(); }
30
31
32
33
34
35     public void DisplayQuad() {...}
36
37
38
39
40
41
42
43

```

□ **Pile** : Pour le traitement des temporaires lors de la compilation.

```

1  import ...
2
3
4
5  public class Pile {
6      LinkedList <String> pile ;
7
8      public Pile() { this.pile= new LinkedList<String>(); }
9
10     public Pile(LinkedList<String> pile) { this.pile = pile; }
11
12
13
14
15
16     void empiler (String element) {
17         //if(!pile.contains(element))
18         pile.add(element);
19     }
20
21     String depiler() {
22         return pile.removeLast();
23     }
24
25
26     boolean pileVide() { return pile.isEmpty(); }
27
28
29
30
31     void displayPile() {...}
32
33
34     void ViderPile() {...}
35
36
37
38
39
40
41

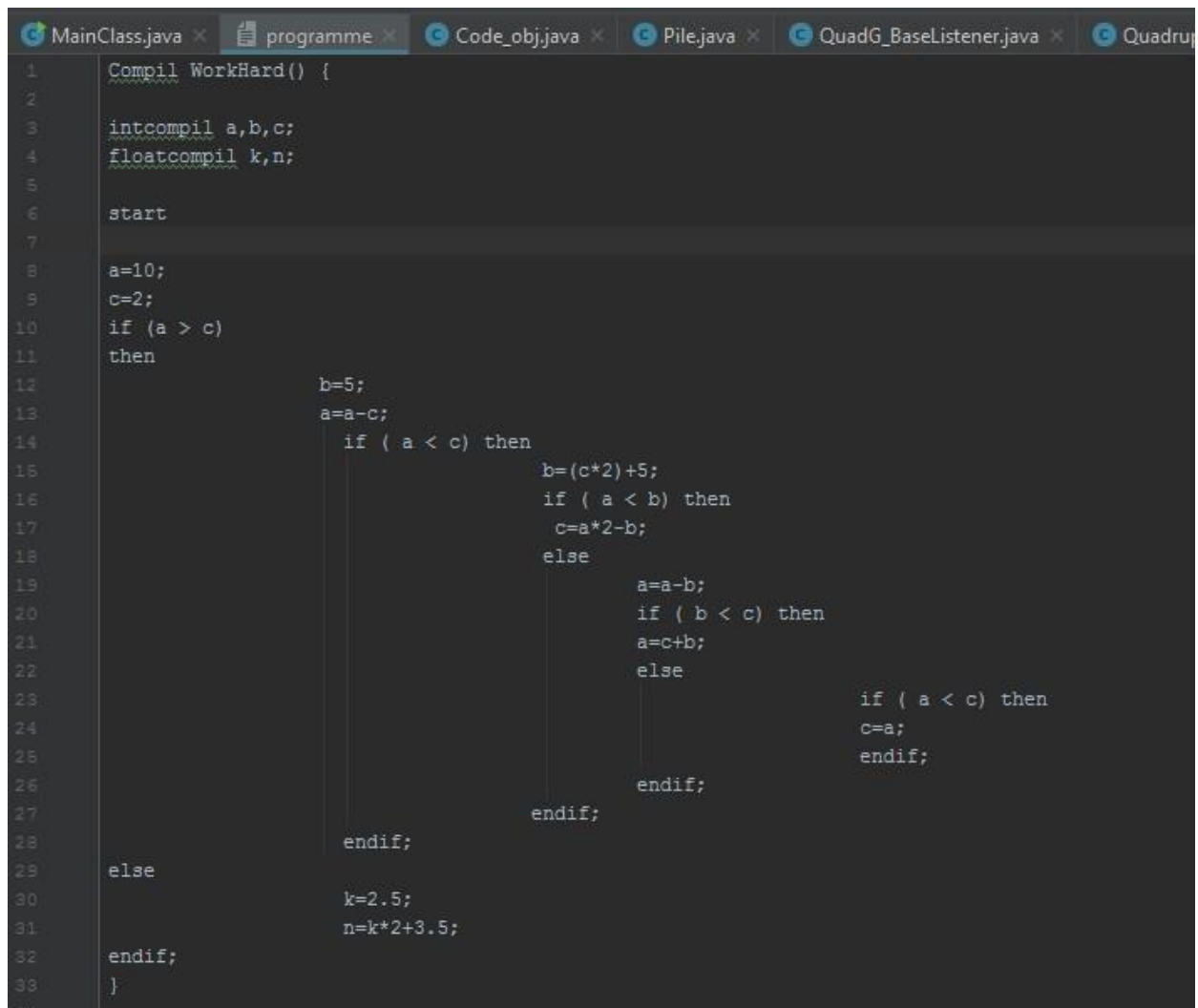
```



Les classes vues précédemment ont été utilisées dans notre *QuadListener* qui redéfinit les méthodes *exitInstruction()*, *enterInstruction()*..

```
1  MainClass.java x programme x Pile.java x QuadG_BaseListener.java x QuadrupleUses.java x
2  import org.antlr.v4.runtime.ParserRuleContext;
3  import org.antlr.v4.runtime.tree.ErrorNode;
4  import org.antlr.v4.runtime.tree.TerminalNode;
5
6  import java.util.ArrayList;
7  import java.util.HashMap;
8
9  public class QuadG_BaseListener extends TinyLanguage_SIIBaseListener{
10
11      QuadrupleUses tabQuad = new QuadrupleUses();
12      Quadruple quad;
13      Pile pile = new Pile();
14      private int compteurTEMPS = 0; // NUM DU TEMPORAIRE UTILISE DANS LES QUADS
15      private Code_obj code_obj;
16
17      /* CES DEUX ETAPES SONT EN PLUS : */
18      ArrayList<TempMeans> quadTable = new ArrayList<>(); // TempMeans : ( + , temp1 , temp2 , temp3 ) -----> ( + , valeur(temp
19      private HashMap<String, String> valeur = new HashMap<>(); // POUR RECUPERER LE CONTENU DU temp'i'
20      /* ---- */
21
22      private int sau_v_condition ;
23      private int sau_v_conditionDeb=0;
24
25      /* <-----: START CODING :-----> */
26
27      @Override
28      public void exitProgram(TinyLanguage_SIIParser.ProgramContext ctx) {...}
29
30      @Override
31      public void exitAffectation(TinyLanguage_SIIParser.AffectationContext ctx) {...}
32
33      private int sau_v_conditionDeb=0;
34
35      /* <-----: START CODING :-----> */
36
37      @Override
38      public void exitProgram(TinyLanguage_SIIParser.ProgramContext ctx) {...}
39
40      @Override
41      public void exitAffectation(TinyLanguage_SIIParser.AffectationContext ctx) {...}
42
43      @Override
44      public void exitExpression(TinyLanguage_SIIParser.ExpressionContext ctx) {...}
45
46      @Override
47      public void exitExpression1(TinyLanguage_SIIParser.Expression1Context ctx) {...}
48
49      @Override
50      public void exitExpression2(TinyLanguage_SIIParser.Expression2Context ctx) {...}
51
52      @Override
53      public void exitCondition(TinyLanguage_SIIParser.ConditionContext ctx) {...}
54
55      @Override
56      public void exitInstruction_conditionnelle(TinyLanguage_SIIParser.Instruction_conditionnelleContext ctx) {...}
57
58      @Override
59      public void enterInstruction_else(TinyLanguage_SIIParser.Instruction_elseContext ctx) {...}
60
61      @Override
62      public void exitInstruction_conditionnelle(TinyLanguage_SIIParser.Instruction_conditionnelleContext ctx) {...}
63
64      @Override
65      public void enterInstruction_else(TinyLanguage_SIIParser.Instruction_elseContext ctx) {...}
66
67      /***** SOME METHOD WE USED *****/
68
69      public void displayQuadruple() { tabQuad.DisplayQuad(); }
70
71      public void displayQuadrupleWithShowingTempMeaning() {
72          System.out.println("-----: QUADRUPE WITH TEMP MEANS :<-----\n");
73          quadTable.stream().map(elem->" "+elem.toString()+" ").forEach(System.out::println);
74          System.out.println("-----\n");
75      }
76
77      // Pour le traitement sup qui visualise le contenu des temporaires
78      void addVal(String expression, String val) {
79          valeur.put(expression, val);
80      }
81
82      String getVal(String expression) {
83          return valeur.get(expression);
84      }
85  }
```

Pour tester la qualité de notre travail nous avons utilisé la portion de code suivante :



```
1  Compil WorkHard() {
2
3  intcompil a,b,c;
4  floatcompil k,n;
5
6  start
7
8  a=10;
9  c=2;
10 if (a > c)
11 then
12     b=5;
13     a=a-c;
14     if ( a < c) then
15         b=(c*2)+5;
16         if ( a < b) then
17             c=a*2-b;
18         else
19             a=a-b;
20             if ( b < c) then
21                 a=c+b;
22             else
23                 if ( a < c) then
24                     c=a;
25                 endif;
26             endif;
27         endif;
28     endif;
29 else
30     k=2.5;
31     n=k*2+3.5;
32 endif;
33 }
```

Tout en modifiant notre classe main :

```
QuadG_BaseListener OwnQuadListener = new QuadG_BaseListener();  
treeWalker.walk(OwnQuadListener, Axiom);
```

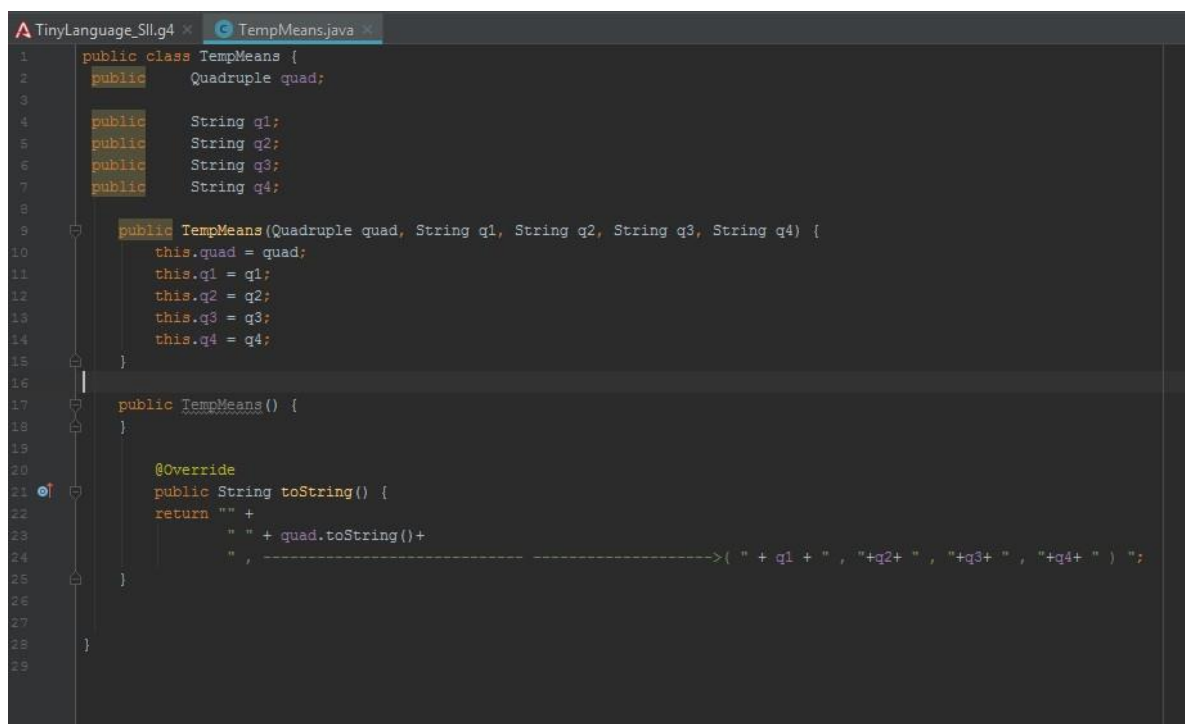
## Résultat ‘Affichage des quadruplés généré’:

```
----->: QUADRUPLE :<-----  
  
Q0 : | ( + , k , b , Temp1 )  
Q1 : | ( = , , Temp1 , a )  
Q2 : | ( BNE , 15 , a , c )  
Q3 : | ( + , b , 5 , Temp2 )  
Q4 : | ( = , , Temp2 , a )  
Q5 : | ( BE , 8 , t , k )  
Q6 : | ( + , c , b , Temp3 )  
Q7 : | ( = , , Temp3 , a )  
Q8 : | ( BL , 14 , y , b )  
Q9 : | ( = , , f , b )  
Q10 : | ( / , f , c , Temp4 )  
Q11 : | ( + , Temp4 , b , Temp5 )  
Q12 : | ( + , e , Temp5 , Temp6 )  
Q13 : | ( = , , Temp6 , b )  
Q14 : | ( BR , 19 , , )  
Q15 : | ( / , o , z , Temp7 )  
Q16 : | ( = , , Temp7 , x )  
Q17 : | ( + , 6 , b , Temp8 )  
Q18 : | ( = , , Temp8 , n )  
Q19 : | ( + , a , c , Temp9 )  
Q20 : | ( = , , Temp9 , n )  
Q21 : | ( END , 22 , , ) |
```

Pour avoir plus de détails sur les temporaires (*Voir figure précédente*) et être précis de nos résultats (*y'en a des variables et des expressions qu'ont été remplacé par des temporelles « temps(i) »*), nous avons ajouté un traitement qui permet à chaque fois de visualiser le contenu des temporaires.

Pour cela nous avons eu besoin de créer une nouvelle classe java ‘**TempMeans**’ :

Elle possède la même structure que la classe **Quadruple** mais avec une sémantique différente.



```
TinyLanguage_Sll.g4 x TempMeans.java x  
1 public class TempMeans {  
2     public Quadruple quad;  
3  
4     public String q1;  
5     public String q2;  
6     public String q3;  
7     public String q4;  
8  
9     public TempMeans(Quadruple quad, String q1, String q2, String q3, String q4) {  
10        this.quad = quad;  
11        this.q1 = q1;  
12        this.q2 = q2;  
13        this.q3 = q3;  
14        this.q4 = q4;  
15    }  
16  
17    public TempMeans() {  
18    }  
19  
20    @Override  
21    public String toString() {  
22        return "" +  
23            " " + quad.toString() +  
24            " , -----> ( " + q1 + " , " + q2 + " , " + q3 + " , " + q4 + " ) ";  
25    }  
26  
27 }  
28  
29
```

Avec ce programme nous avons effectué notre test

```
TinyLanguage_Sll.g4 x MainClass.java x programme x
1
2  compil WorkHard() {
3
4  intcompil a,b,c,a,a,a,vb;
5  floatcompil k,n,k;
6
7  //Begin Of my main()
8  start
9
10 a=b+k;
11
12 scancompil(x,y,z);
13 printcompil("Hello World!",x);
14
15 if (a == c)
16 then
17     a=5+b ;
18     if (t != k)
19     then
20         a=b+c;
21     endif;
22
23     if (y >= b)
24     then
25         b=f;
26         b=b+c/f+e;
27     endif;
28
29
30 else
31     x=z/o;
32     n=b+6;
33 endif;
34
35 n=c+a;
36
37 //return0;
38 }
39
```

## Résultat :

On peut clairement observer et comprendre le contenu des temporaires.

```
----->: QUADRUPLE WITH TEMP MEANS :<-----
| ( + , k , b , Temp1 ) , -----> ( + , b , k , b+k )
| ( = , , Temp1 , a ) , -----> ( = , , b+k , a )
| ( + , b , 5 , Temp2 ) , -----> ( + , 5 , b , 5+b )
| ( = , , Temp2 , a ) , -----> ( = , , 5+b , a )
| ( + , c , b , Temp3 ) , -----> ( + , b , c , b+c )
| ( = , , Temp3 , a ) , -----> ( = , , b+c , a )
| ( = , , f , b ) , -----> ( = , , f , b )
| ( / , f , c , Temp4 ) , -----> ( / , c , f , c/f )
| ( + , Temp4 , b , Temp5 ) , -----> ( + , b , c/f , b+c/f )
| ( + , e , Temp5 , Temp6 ) , -----> ( + , b+c/f , e , b+c/f+e )
| ( = , , Temp6 , b ) , -----> ( = , , b+c/f+e , b )
| ( / , o , z , Temp7 ) , -----> ( / , z , o , z/o )
| ( = , , Temp7 , x ) , -----> ( = , , z/o , x )
| ( + , 6 , b , Temp8 ) , -----> ( + , b , 6 , b+6 )
| ( = , , Temp8 , n ) , -----> ( = , , b+6 , n )
| ( + , a , c , Temp9 ) , -----> ( + , c , a , c+a )
| ( = , , Temp9 , n ) , -----> ( = , , c+a , n )
| ----->
```

### III. Génération du code objet

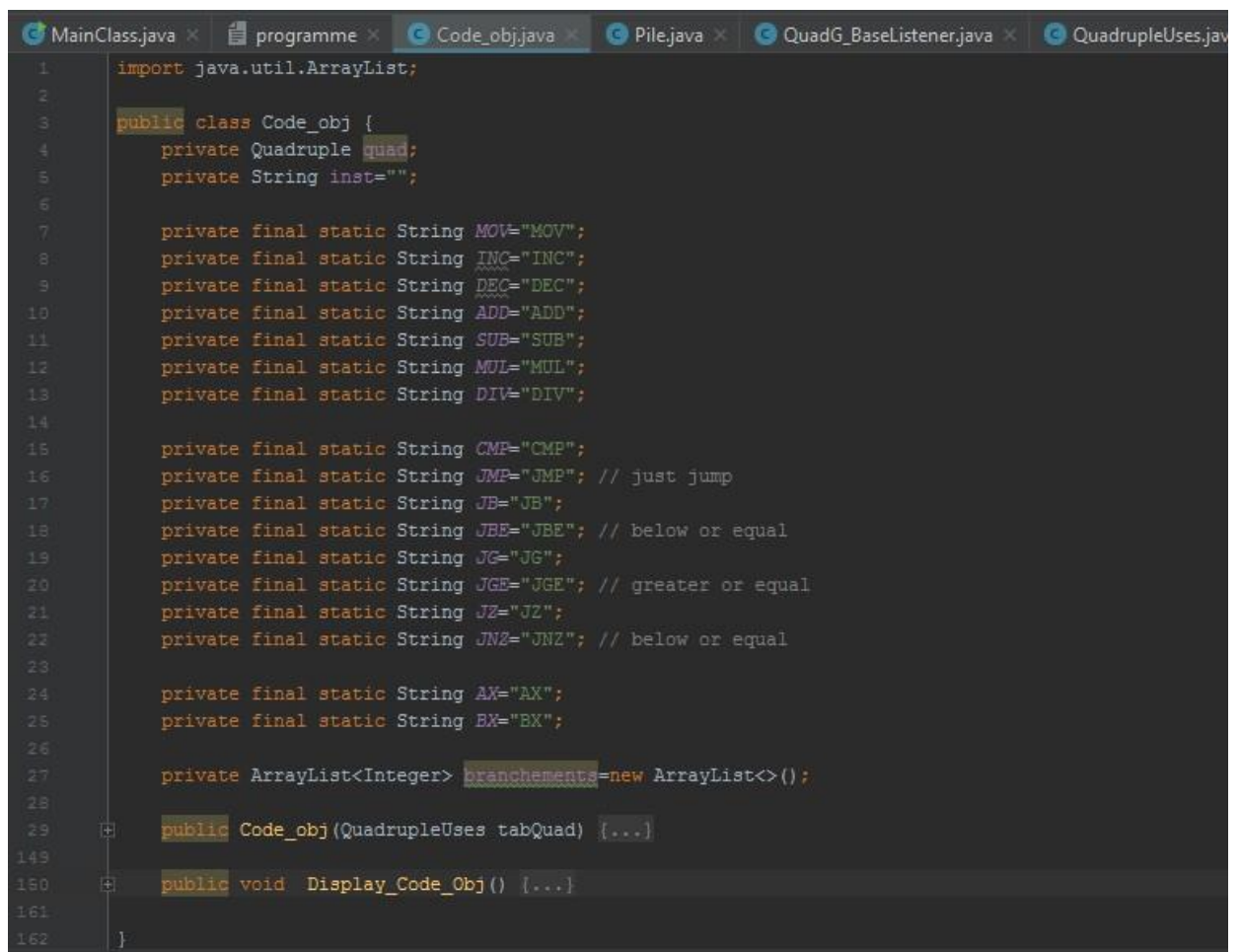
Celle-ci est la dernière phase de notre compilateur.

Elle consiste en la traduction des quadruples générés précédemment en code objet.

Donc nous avons ajouté une nouvelle classe java 'Code\_obj' qui dispose de deux principales méthodes

- Display\_Code\_Obj : qui va simplement afficher toutes les instructions traduites en code assembleur

Code\_obj : le constructeur qui fait tout le travail. Il reçoit en paramètre la liste des quadruples et avec un switch case et d'éventuels traitements pour le comptage (spécialement des adresses de branchement) il fait la traduction en instructions assembleurs (INTEL 8086 )



```
1  import java.util.ArrayList;
2
3  public class Code_obj {
4      private Quadruple quad;
5      private String inst="";
6
7      private final static String MOV="MOV";
8      private final static String INC="INC";
9      private final static String DEC="DEC";
10     private final static String ADD="ADD";
11     private final static String SUB="SUB";
12     private final static String MUL="MUL";
13     private final static String DIV="DIV";
14
15     private final static String CMP="CMP";
16     private final static String JMP="JMP"; // just jump
17     private final static String JB="JB";
18     private final static String JBE="JBE"; // below or equal
19     private final static String JG="JG";
20     private final static String JGE="JGE"; // greater or equal
21     private final static String JZ="JZ";
22     private final static String JNZ="JNZ"; // below or equal
23
24     private final static String AX="AX";
25     private final static String BX="BX";
26
27     private ArrayList<Integer> branchements=new ArrayList<>();
28
29     public Code_obj(QuadrupleUses tabQuad) {...}
30
31     public void Display_Code_Obj() {...}
32 }
```

Pour tester nous avons utilisé le bout de code précédant et nous avons obtenu les résultats suivants :

```
*****

Code Objet

MOV AX,k
ADD AX,b
MOV Temp1,AX
MOV AX,Temp1
MOV a,AX
MOV AX,c
MOV BX,a
CMP AX,BX
JZ ETIQ 10
MOV AX,b
MOV BX,y
CMP AX,BX
JBE ETIQ 9
MOV AX,f
MOV b,AX
MOV AX,f
DIV AX,c
MOVTemp2,AX
MOV AX,Temp2
ADD AX,b
MOV Temp3,AX
MOV AX,e
ADD AX,Temp3
MOV Temp4,AX
MOV AX,Temp4
MOV b,AX
ETIQ9:
JMP ETIQ 12
JMP ETIQ 12
ETIQ10:
MOV AX,b
ADD AX,b
MOV Temp5,AX
MOV AX,Temp5
MOV n,AX
ETIQ12:
MOV AX,c
MOV BX,a
CMP AX,BX
JGE ETIQ 15
MOV AX,a
ADD AX,c
MOV Temp6,AX
MOV AX,Temp6
MOV n,AX
ETIQ15:
JMP ETIQ 18
MOV AX,b
ADD AX,b
MOV Temp7,AX
MOV AX,Temp7
MOV n,AX
ETIQ18:

*****
```



## ANTLR vs Flex/Bison :

La différence la plus significative entre *Flex/YACC/Bison* et *ANTLR* est le type de grammaire que ces outils peuvent traiter.

*Flex/YACC/Bison* gère les grammaires LALR, *ANTLR* gère les grammaires LL.

Souvent, les personnes qui ont travaillé avec des grammaires LALR pendant une longue période, trouveront plus difficile de travailler avec les grammaires LL et vice versa. Cela ne signifie pas que les grammaires ou les outils sont intrinsèquement plus difficiles à utiliser. L'outil que vous trouvez le plus facile à utiliser dépend surtout du type de grammaire.

En ce qui concerne les avantages, il y a des aspects où les grammaires LALR ont des avantages sur les grammaires LL et il y a d'autres aspects où les grammaires LL ont des avantages sur les grammaires LALR.

*Flex/YACC/Bison* génère des analyseurs pilotés par table, ce qui signifie que la "logique de traitement" est contenue dans les données du programme analyseur, pas tellement dans le code de l'analyseur. L'avantage est que même un parseur pour un langage très complexe a une empreinte de code relativement faible. C'était plus important dans les années 1960 et 1970, lorsque le matériel était très limité. Les générateurs d'analyseurs pilotés par table remontent à cette époque et l'empreinte du petit code était une exigence principale à l'époque.

*ANTLR* génère des analyseurs de descente récursifs, ce qui signifie que la "logique de traitement" est contenu dans le code de l'analyseur, car chaque règle de production de la grammaire est représentée par une fonction dans le code de l'analyseur. L'avantage est qu'il est plus facile de comprendre ce que fait l'analyseur en lisant son code. En outre, les analyseurs de descente récursifs sont généralement plus rapides que les analyseurs pilotés par table. Cependant, pour les langues très complexes, l'empreinte du code sera plus grande. C'était un problème dans les années 1960 et 1970. À l'époque, seuls des langages relativement petits comme **Pascal** par exemple étaient implémentés de cette façon en raison de limitations matérielles.

Et finalement on va citer quelques avantages pour ANTLR:

- peut produire des analyseurs syntaxiques dans plusieurs langues - Java n'est pas nécessaire pour exécuter l'analyseur généré.
- L'interface graphique géniale facilite le débogage de la grammaire (par exemple, vous pouvez voir les AST générés dans l'interface graphique, sans outils supplémentaires requis).
- Le code généré est réellement lisible par l'homme (c'est l'un des objectifs d'ANTLR) et le fait qu'il génère des parseurs LL aide certainement à cet égard.
- la définition des terminaux est également sans contexte (par opposition à *regex* dans *Flex*) - permettant ainsi, par exemple, la définition de terminaux contenant des parenthèses correctement fermées.

## Conclusion:

La réalisation de ce projet, qui avait pour but le développement d'un compilateur, nous a permis de nous familiariser avec un outil qui nous était jusqu' alors inconnu qui est l'ANTLR.

Nous avons donc pu définir notre grammaire et générer un analyseur lexical ainsi qu'un analyseur syntaxique, complétés par l'analyse sémantique de notre grammaire. Une fois l'analyseur conçu la prochaine étape fut la génération du code intermédiaire sous forme de quadruplés, pour finir avec la génération du code objet en langage assembleur.