REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE

Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

UNIVERSITÉ DE SCIENCE ET DE TECHNOLOGIE HOUARI BOUMEDIENE

Département d'informatique

Projet de compilation TinyLanguage_SII

Réalisé par:

DJEBROUNI Radhia

DJAOUADI Yamina

Encadré par:

BETIT Lilya

M1 SII: 2020-2021

Sommaire

Introduction

De nos jours, l'outil le plus puissant est ANTLR , qui est facile à utiliser et à manipuler . il est apparu après flex et bison qu'on les a déjà utilisés pour réaliser un tel projet.

Durant ce projet TinyLanguage, nous allons créer un compilateur pour le langage TinyLanguage qu'on va définir.

Ce projet est constitué de quatre chapitres, commençant par la création des règles lexicales et syntaxiques pour définir notre grammaire, ensuite générer les erreurs sémantiques , les quadruplets et enfin le code objet

Chapitre 1 : Généralités

1. La Compilation

La compilation informatique désigne le procédé de traduction d'un programme, écrit et lisible par un humain, en un programme exécutable par un ordinateur. De façon plus globale, il s'agit de la transformation d'un programme écrit en code source, en un programme transcrit en code cible, ou binaire. Habituellement, le code source est rédigé dans un langage de programmation (langage source), il est de haut niveau de conception et facilement accessible à un utilisateur. Le code cible, quant à lui, est transcrit en langage de plus bas niveau (langage cible), afin de générer un programme exécutable par une machine.

2. Le rôle d'un compilateur

Un compilateur est un sous-programme d'un langage de programmation. Il a pour rôle de rechercher toutes les erreurs possibles dans un programme source, telles que des fautes d'orthographe, les variables, les types, etc.

3. Les étapes de la création d'un compilateur

Un programme se compile selon une série d'étapes :

1. **le prétraitement** : il intervient avant toute analyse, pour savoir comment traiter les informations ;
2. **l'analyse lexicale** : elle divise le code source en petits morceaux : les tokens (les jetons). Chaque token a une unité lexicale unique de la langue (ou lexème), tel qu'un identifiant, un symbole ou un mot-clé par exemple. Cette étape est aussi nommée le lexing ou balayage, car le programme qui exécute cette analyse lexicale peut se nommer scanner ou analyseur lexical ;
3. **l'analyse syntaxique** : elle analyse la séquence des tokens pour reconnaître la structure syntaxique du programme. On modifie la syntaxe linéaire des tokens par une structure en arborescence, générée selon la grammaire formelle qui détermine la syntaxe du langage. L'arborescence est fréquemment modifiée et améliorée pendant la compilation ;
4. **l'analyse sémantique** : le compilateur, durant cette étape, complète l'arborescence par des informations sémantiques et érige la table des symboles. Cette phase teste

les erreurs de type, ou une tâche définie (telles que les variables locales), ou l'objet de liaison, et peut même dispenser des avertissements ou rejeter des programmes inexacts ;

5. **La génération de code intermédiaire** (quadruplets) ;
6. **la génération de code d'objet** : par la traduction du code intermédiaire en code assembleur .

4. Définitions de quelques termes utilisés

4.1. ANTLR

ANTLR, sigle de *ANother Tool for Language Recognition*, est un framework libre de construction de compilateurs utilisant une analyse LL(*).ANTLR utilise une notation identique pour définir les différents types d'analyseurs, qu'ils soient lexicaux, syntaxiques, ou d'arbre.

Les grammaires ANTLR sont des sous classes de Lexer, Parser, TreeParser

4.2. Listener

Un Listener est une interface qui pourra être utilisée pour implémenter les règles des routines sémantiques.

4.3. Parser

Une interface utilisée par Antlr pour construire l'arbre syntaxique de la grammaire.

4.4. Lexer

utilisé par Antlr pour traduire un flux de caractères en flux de Tokens.

4.5. Intellig IDEA

Également appelé « IntelliJ », « IDEA » ou « IDJ » est un environnement de développement intégré (en anglais Integrated Development Environment - IDE) de technologie Java destiné au développement de logiciels informatiques et développé par JetBrains.

4.6. JAVA

Java est un langage de programmation orienté objet créé par James Gosling

Chapitre 2 : L'environnement du travail et structure externe du projet

1. Environnement du travail

- Système d'exploitation: Windows 10
- Logiciel de programmation: Intellij IDEA
- Langage utilisé: JAVA
- Outil : ANTLR

2. L'utilisation du ANTLR:

L'utilisation de ANTLR nous a aider tout au long de la création du compilateur, il fournit des classes Listener , parser et Lexer contenant des méthodes prédéfinies qui vont être exécuté à la fin de chaque règle (ou bien token) détectée durant la génération de l'arbre syntaxique afin d'exécuter les routines sémantiques qu'on va les définir dans le chapitre suivant.

Donc notre projet va contenir un fichier.g4 qui va contenir les règles de notre grammaires, un fichier test.txt qui va contenir le programme a tester, une classe pour structurer la table des symboles de notre programme, une classe qui va hériter la classe Listener généré par defaut par ANTLR et qui va contenir les méthodes qui vont être déclenchés automatiquement après chaque règle détectée lors du parsing, et finalement notre classe main qui va déclancher les autres classes et méthodes créés.

Chapitre 3 : Étapes de la réalisation du projet

1. Introduction

Lexer et parser est un outil offert par ANTLR, utilisé pour générer une grammaire descendante. Pendant la génération de cette grammaire, on doit définir les règles d'abord après les entités.

2. Analyse lexicale

2.1. Les expressions régulières

```
NomProg:[A-Z][A-Z a-z]* ESPACE* Space*('ESPACE* Space*')';

ID :[a-zA-Z][a-zA-Z0-9]*;

EntierS:'('[+-][0-9]+')';

Entier:[0-9]+;

ReelS:'('[+-][0-9]+ '.'?[0-9]+')';

Reel:[0-9]+ '.'?[0-9]+;
```

*Espace: représente les sauts de ligne (\n) et les espaces longs (\t|\r)

*ID: c'est tout ce qui est nom de variable utilisé dans le langage

*EntierS: représente les entiers signés, de même pour ReelS qui représente les réels signés

***Problèmes d'Ambiguïté détectés:**

Vu que y'avait une ambiguïté entre le nom d'un programme et le ID de tel sorte que tout nom de programme appartient à l'expression régulière del' ID, alors on a pu régler ça par l'ajout des parenthèses et espaces facultatifs dans l'expression régulière

de NomProg, ce qui empêche le parser à mélanger entre les deux est générer des erreurs lexicales inutiles.

2.2. Lexique utilisé

Mots clés	Opérateurs

COMPIL:'compil ';
Start:'start';
Int:'intCompil';
String:'stringCompil';
Float:'floatCompil';
Then:'then';
Else:'else' ;
If:'if';
DoWhile:'while';
Do:'do';
Print:'printCompil';
Scan:'scanCompil';
	OPL :PLUS|MINUS; //lower priority
	OPH: DIV|MUL; //higher priority
	Comp:SUP|EQ|INF|DIF;
	AFF : '=';
	PLUS : '+';
	MINUS : '-';
	MUL : '*';
	DIV : '/';
	SUP : '>';
	INF : '<';
	EQ : '==';
	DIF : '!=';

2.3. To Skip

Les expressions toSkip sont les expressions à ignorer tel que les espaces et les commentaires.

```
COMM1L : '/''/'(~[\n])* -> skip;      //commentaire en une seule ligne
COMM2L : '/*' (.)*?'*/' -> skip;      //commentaire en plusieurs lignes
ESPACE : [ \n\t\r] -> skip;           // saut de ligne et espace double
Space: ' '->skip;                     //espace
```

3. Analyse syntaxique

3.1. Grammaire de la règle globale

```
programme:COMPIL NomProg '{' declar 'start' desc '}';
```

*déclare représente une suite de déclarations

*desc représente une suite d'instructions

3.2. Les déclarations

```
declar:declar dec|dec;
dec:type ids ';';
type: Int|Float|String;
```

3.3. Instructions

```
desc: inst|inst desc;
inst:comp|instif|aff|boucle|scan|print;
instif: 'if' '(' comp ')' Then '{' desc '}'(|el'{'desc'}') ;   //if et un else facultatif
comp: opr Comp opr ;                                          //comparaison
aff:idp AFF opr ';';                                          //affectation
el:Else;                                                      //else
// boucle do while
boucle: doo '{'desc '}'DoWhile '(' comp ')';
doo:Do;
idp:ID;
```

3.4. Opérations

```
opr:opr OPL t|t;

t:t OPH exp|exp;

exp: val|'(' opr ')';

val:Entier|Reel|ID|EntierS |ReelS ;
```

***Génération de priorité entre les opérateurs '/' '*' et '+' '-'**

Afin de traiter le problème de priorité lors de la génération de l'arbre syntaxique, on a utilisé la méthode suivante:

On a défini OPL pour les opérateurs PLUS et MOIN, OPH pour MUL et DIV. Selon la grammaire décrit ci dessus , notre programme va d'abord parser toute l'opération jusqu'à l'arrivé d'une OPL, il va générer l'addition ou bien la soustraction entre tout le chemin passé avec ce qu'il reste, de même , il parcourt tout le chemin passé et resté pour détecter les multiplication et les divisions, et ainsi de suite, le but de cette méthode et de mettre les opérations les moins prioritaire à la racine de l'arbre syntaxique pour pouvoir traiter les opérations les plus prioritaire seules.

3.5. Affichage et lecture

```
// L'instruction de l'ecture
scan:Scan '('id ')' ';';

// L'instruction d'écriture
print: Print '(' idp ')' ';';

ids:ID ',' ids|ID;
id: ID ',' id|ID;
idp:ID;
```

4. Table des symboles

La table de symboles est une table qui contient tous les symboles détectés dans la grammaire associés par leurs types et leurs valeurs afin de pouvoir gérer l'exécution et les erreurs sémantiques.

Afin de bien structurer notre table de symbole, on a utilisé un **Hashmap ,** c'est une structure qui permet d'avoir une liste d'objets, chaque objet contient une clé associé par son élément, On a définit le nom de chaque symbole comme une clé , vu qu'il doit être unique, son élément est un objet instancié d'une classe qui contient troi attributs:

*l'attribut type qui vaut 0 pour int, 1 pour un Float, et 2 pour un String.

*L'attribut valeur qui va contenir la valeur du symbole.

*Le boolean déclaré pour tester si le symbole est déjà déclaré ou pas.

Le but d'utiliser cette structure est de faciliter le travail en utilisant des méthodes prédéfinies faciles à gérer pour créer les méthodes de recherche, modification, suppression...etc dont on a besoin.

Illustration de la Table des symboles

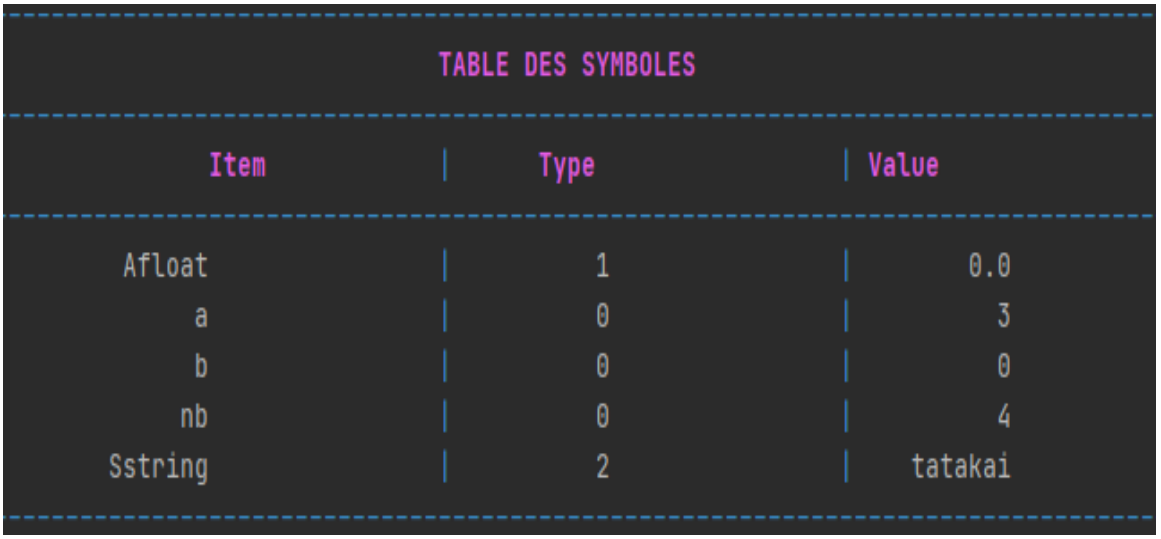


TABLE DES SYMBOLES

Item	Type	Value
Afloat	1	0.0
a	0	3
b	0	0
nb	0	4
Sstring	2	tatakai

5. Analyse sémantique

5.1. Mise à jour de la table de symbole

Notre table de symbole va être modifiée dans les cas suivants:

-Une déclaration d'une variable jamais déclarée: dans ce cas on va insérer une nouvelle ligne dans la TS contenant les nouvelles informations dans la méthode héritée "exitDec" qui va être exécutée après chaque instruction de déclaration.

-Une affectation d'une variable existante et sans opération, ex: a=0 ou bien a=c; dans la méthode "exitAff"

-Après chaque ScanCompil sans erreur dans la methode "exitScan"

5.2. Erreurs sémantiques

ANTLR génère une erreur si le code introduit par le programmeur ne correspond pas à notre grammaire. Par contre, c'est à notre compilateur de vérifier la sémantique du code, et de gérer les erreurs résultantes.

A. Double déclaration

Cette erreur va être générée dans la méthode “exitDec” lors d’un trouve d’une déclaration d’une variable déjà existante dans la table des symboles .

B. Une variable non déclarée

Cette erreur va être générée lors d’une utilisation d’une variable qui n’existe pas dans la table des symboles. Elle pourra être utilisée dans plusieurs cas : une affectation, une opération de lecture/ écriture, et aussi dans les expressions arithmétiques.

C. Initialisation

Dans notre langage Tiny LanguageSII, les variables vont être initialisées automatiquement après leurs déclarations, ce qui ne va générer aucune erreur d’initialisation;

Les entiers vont avoir la valeur 0, les Float vont avoir la valeur 0.0, et les String vont avoir la chaîne vide “” .

D. Division par 0

Après chaque opération qui contient un opérateur de division “/” on test la valeur du 2eme opérande (sauf si elle était sous forme d’une expression), si la valeur ou la variable vaut 0 alors on génère une erreur de division par 0

NB: la vérification se fait dans la méthode “exitT”

E. Incompatibilité de type

l’erreur de l’incompatibilité de type va être généré dans les cas suivants:

-Incompatibilité de type entre deux opérandes d’une même opération.

-Incompatibilité de type lors d’une affectation à un ID: s’il y avait des opérandes de type réel dans une opération dans l’ affectation, alors le ID devrait être de type réel , sinon le ID peut être un entier seulement si tous les opérandes de l'opération sont des entiers . Dans les autres cas, une erreur va être générée.

Illustration Voici le code suivant:

```
compil Arrt() {
intCompil a,a,b;
floatCompil Afloat;
stringCompil Sstring;
start
printCompil (a );
scanCompil(a);
scanCompil(Sstring);
a=3/b;
c=0;
Afloat=4.6;
a=0 +a/2 + Afloat ;
Afloat=a+2;
}
```

Lecture des variable a et Sstring

```
ENTRER UNE VALEUR
abcd
ENTRER UNE VALEUR
Sstringg
```

Table des symboles et erreurs détectés

```
************************************ PROGRAM FAILED****************************************

Variable a deja declarée
Incompatibilité de type dans le scan de a
Erreur de division par 0 dans l'operation 3/b
variable c non déclarée  dans l'affectation c=0;
Inconpatiblite de type dans l'affectation a=0+a/2+Afloat;


--------------------------------------------------------------------------------------------
                                        TABLE DES SYMBOLES
--------------------------------------------------------------------------------------------
                        Item            |       Type                |  Value
--------------------------------------------------------------------------------------------
                      Afloat            |          1                |         4.6
                           a            |          0                |           0
                           b            |          0                |           0
                     Sstring            |          2                |    Sstringg
--------------------------------------------------------------------------------------------
```

5.3. Les Quadruplets

Le but principal de cette phase est de transformer le code fourni par le programmeur en un autre code plus proche du langage machine. Ce qu'on appelle les quadruplets.

NB:La construction des quadruplets d'une certaine expression ne peut être effectuée que si l'expression est correcte sémantiquement.

Structure des données utilisées : Nous avons eu besoin d'une :

- LinkedList quad : qui a pour but de sauvegarder l'ensemble des quadruplets fournis par notre compilateur.
- stack : une pile pour sauvegarder les opérandes et les temporaires des expressions arithmétiques.

A. Les quadruplets des affectations

La méthode exitAff va générer le quadruplet finale de l' affectation , qui aura la forme suivante :

(“ := “, temporaire de l'expressions arithmétique, “ “,identificateur cible) ;

Le temporaire qui existe dans la forme générale est extrait à partir de la pile stack. Cette pile est remplie par les noms des opérandes lors de la rencontre des expressions arithmétiques.

B. Les quadruples des expressions arithmétiques

A chaque fois qu'on rencontre une valeur ou une variable, on l'empile. Une fois qu'une opération arithmétique est rencontrée, on dépile deux fois et on génère le quadruplet adéquat. Et à la fin, on rempile le temporaire jusqu'à obtenir tous les quadruplets d'une expression. Le dernier temporaire sera utilisé dans l'affectation.

La forme du quadruplé d'une opération:

(“ opérateur “, opérande 1, opérande 2, Temporaire) ;

C. Les quadruplets des I/O

Les quadruplets des entrées pour chaque ID s dans l'instruction ScanCompil:

("READ",s,"","")

Dans la méthode prédéfinie exitScan, on parcourt la liste des variables ,et pour chaque variable on génère un quadruplet READ. De même pour la méthode exitPrint en remplaçant le mot READ par WRITE

D. Les quadruplets des conditions et des boucles

Les Conditions:

Pour chaque condition en sauvegarde le qc de la fin de toute l'instruction if dans la méthode exitInstIf (le qc représente la taille actuelle de la pile Quad)

Juste après l'instruction de la condition, en génère le quadruplé qui contient l'opérateur inverse et qui va pointer vers la fin du if ou bien vers le else s'il existe, on laisse l'adresse cible vide, on sauvegarde le qc, et on accède une deuxième fois à cet quadruplet a la fin du if en modifiant l'adresse vide par le qc généré a la fin de l'instruction if.

ex: if(a<3) then{ ... }

1- qc1=1

2-On insère le quad(“BGE”,a,3,” “) dans la liste des quad

NB: BGE, BE, BNE, BLE sont les inverses de <, >,==, != respectivement

3-on est a la fin , le qc2 =6 par ex

4-On accede a Quad(qc1) , on modifie l'adresse vide par 6

Les quad de la boucles do{ }while(cond):

Pour chaque boucle en sauvegarde le qc de la fin de la règle do seulement.

Juste après l’instruction de la condition a la fin, en génère le quadruplé qui contient la condition en mettant la valeur de l'adresse cible le qc de do déjà sauvegardé

ex: do{ ... }while (a<3)

1- qc1=1

2-après le while On insère le quad(“BL”,a,3,” qc1“) dans la liste des quad

NB: dans la boucle on met l'opérateur adéquat à l'opération, contrairement à la condition if.

Illustration des quadruplets Exemple du code:

```
compil Arrt() {
intCompil a,b;
floatCompil Afloat;
stringCompil Sstring;
start
printCompil (tdhjfj );
printCompil (a );
scanCompil(b);
do{
if (a==3)
then
{a=3;
b=a*3 + 4;
}else
{a=5+5;}
}while(a<3)
}
```

Résultat:

```
-------------- QUADRUPLETS --------------

0-( WRITE ,tdhjfj , , )
1-( WRITE ,0 , , )
2-( READ ,b , , )
3-( BNE ,a ,3 ,9 )
4-( := ,3 , ,a )
5-( * ,a ,3 ,Temp1 )
6-( + ,Temp1 ,4 ,Temp2 )
7-( := ,Temp2 , ,b )
8-( BR , , ,11 )
9-( + ,5 ,5 ,Temp3 )
10-( := ,Temp3 , ,a )
11-( BL ,a ,3 ,3 )
12-( END , , , )

-----------------------------------------
```

6. Génération du code d'objet

Le principe est de parcourir la liste des quadruplet un par un, pour générer des lignes de code Assembleur pour chacun, et les insérer dans la liste codeAss qui contient la liste des lignes de notre code finale.

NB:La génération du code d'objet se fait au niveau de la méthode exitProgramme

Ex: Pour l'opération d'addition

```
if(quad.getQuad(i).getOperateur().equals("+"))
{
  codeAss.add("\tMOV  AX "+quad.getQuad(i).getOpgauche());
  codeAss.add("\tADD "+quad.getQuad(i).getOpdroit());
  codeAss.add("\tMOV "quad.getQuad(i).getContaineur()+" , "+"AX ");
}
```

Et de même pour les autres opérations en remplaçant par les opérateurs par les opérateurs qui convient ou bien par les instruction JE, JNE, JLE, JL, JGE, JG en cas de test ou bien l'instruction JUMP en cas de branchement inconditionnel (ex: juste avant le else on se branche vers la fin du else).

Illustration avec le code précédent

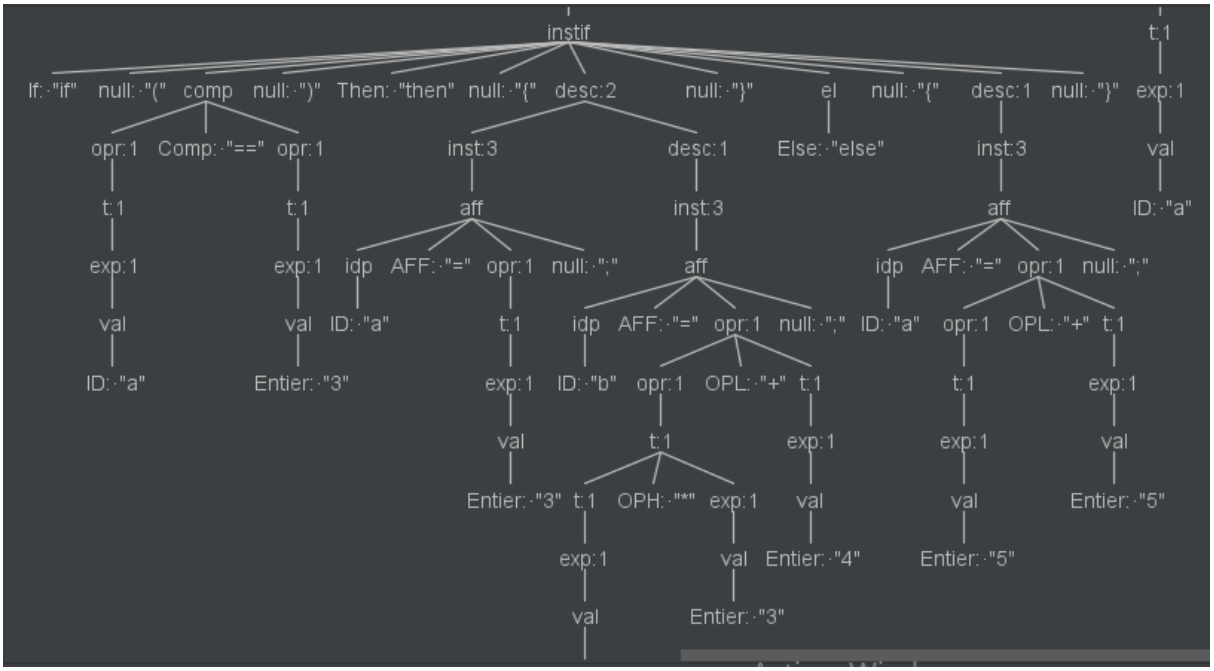
```
-------------- CODE D'OBJET ---

    BEGIN
Etiq0:
    OUTPUT  tdhjfj
Etiq1:
    OUTPUT  0
Etiq2:
    INPUT  b
Etiq3:
    MOV AX a
    COMP 3
    JNE ETIQ  9
Etiq4:
    MOV AX 3
    MOV a , AX
Etiq5:
    MOV AX ,3
    MUL a
    MOV Temp1 , AX
Etiq6:
    MOV AX 4
    ADD Temp1
    MOV Temp2 , AX
Etiq7:
    MOV AX Temp2
    MOV b , AX
Etiq8:
    MOV AX
    COMP
    JMP ETIQ  11
Etiq9:
    MOV AX 5
    ADD 5
    MOV Temp3 , AX
Etiq10:
    MOV AX Temp3
    MOV a , AX
Etiq11:
    MOV AX a
    COMP 3
    JL ETIQ  3
Etiq12:
    END

Process finished with exit
```

7. La classe main

Notre classe main qui s'appelle 'Launch' est la classe qui va lancer le lexer et le parser qui va parser la grammaire du fichier test.txt pour créer l'arbre syntaxique et en exécutant les méthodes du Listener qui vont générer la table de symboles, détecter les erreurs sémantiques, transformer la grammaire en code intermédiaire (Quadruplets), et enfin générer le code d'objet finale.

```
String source= "test.txt";
CharStream cs =fromFileName(source);
TinyLanguage_SIILexer lexer=new TinyLanguage_SIILexer(cs);
CommonTokenStream token =new CommonTokenStream(lexer);
TinyLanguage_SIIParser parser =new TinyLanguage_SIIParser(token);
ParseTree tree=parser.programme();
listner l=new listner();
ParseTreeWalker walker=new ParseTreeWalker();
walker.walk(l,tree);
```

Exemple d'un arbre syntaxique générée avec Antlr (l'instruction if de l'exemple précédent)



Conclusion

Tout au long de la réalisation de notre projet, nous avons rencontré beaucoup de problèmes et d'erreurs , mais à la fin on a pu réaliser toute les parties du projet, on a apri beaucoups d' informations, on a appris à utiliser de nouvelles technologies , allons de l'installation de l'outil Antler, apprendre à utiliser ses classes, l'utiliser pour générer la grammaire, l'arbre syntaxique, les routines sémantiques, le code intermédiaire, et finalement le code d'objet.

Notre Projet offre la possibilité de compiler un programme écrit en langage Tiny LanguageS II qu'on a défini, comme perspectives, nous voulons intégrer dans notre projet la possibilité

d’optimiser le code d’objet généré a la fin, afin de minimiser l’espace mémoire alloué pour l'exécution, et créer une solution optimale.

Ressources

https://www.journaldunet.fr/web-tech/dictionnaire-du-webmastering/1445284-compilation-informatique-definition-concrete-et-role/#:~:text=Un%20compilateur%20est%20un%20sous,variables%2C%20les%20types%2C%20etc.

http://dictionnaire.sensagent.leparisien.fr/ANTLR/fr-fr/

https://fr.wikipedia.org/wiki/Java_(langage)

http://dspace.univ-msila.dz:8080/xmlui/bitstream/handle/123456789/2905/SAOUDI%20Azzouz.pdf?sequence=1&isAllowed=y