Université des Sciences et de la Technologie Houari Boumediene



Faculté d’Électronique et d’Informatique

Département d’Informatique

Rapport de projet de complexité

Réalisation d’un mini compilateur avec l’outil ANTLR

Réalisé par :

* ABDELALI Asma Nihad (Groupe: 1)
* KEMOUM Meroua (Groupe: 1)

Responsable du module : Mlle Lylia BETIT

Année : 2020/2021

Introduction

Au long de ce projet nous allons réaliser un mini compilateur pour un langage donné. Le compilateur doit être capable d’analyser un fichier, reconnaitre les entités lexicales, s’assurer de la cohérence syntaxique et vérifier leurs erreurs sémantiques. En passant par un code intermédiaire notre langage devra à la fin être transformé en code assembleur

Vous trouverez ici le lien Github du projet : <https://github.com/AANihad/TP_Compile>

Langage à reconnaitre :

* Le nom du programme commence par une lettre majuscule
* Les identificateurs commencent par une lettre éventuellement suivie de lettres ou de chiffres
* On distingue les majuscules et les minuscules : a et A ne sont pas le même identificateur
* Les variables doivent être déclarées
* Les variables doivent être initialisées
* Les instructions sont terminées par des points virgules ;
* Les opérateurs des expressions numériques sont : +, -, \*, /
* Leurs opérandes sont des nombres entiers ou réels
* Le type de l’expression est casté au type de l’id auquel on l’affecte
* On peut comparer des nombres entre eux par : >, =>, =<, <, ==, !=
* La priorité des opérateurs est comme suit, par ordre croissant : /, \*, +, -, >, <, ==, ! = sauf le cas des parenthèses.
* L’affectation (=) de variables avec des valeurs entières ou réelles

Instructions reconnues et implémentées :

* Une instruction conditionnelle avec un block sinon
* Une boucle do – while
* Une instruction d’écriture
* Une instruction de Lecture

Partie 1 : Analyses lexicale et syntaxique

1. Analyse Lexicale

Le but de l’analyse lexicale est de s’assurer que les caractères sont dans le bon ordre et que les mots reconnus font partie de notre langage.

Constituant la première étape dans la compilation, l’analyse lexicale consiste d’abord à analyser le mini langage, puis de définir les entités lexicales qui le constituent. Ces entités seront utilisées dans la partie syntaxique plus tard.

Dans cette partie nous avons défini les mots clés, réservés par le langage ainsi que défini les expressions régulières qui identifient le nom du programme, les identificateurs, les entiers, les nombres flottants, les commentaires et les chaines de caractères.

//Expressions régulières

NOM\_PROGRAMME : [A-Z]([a-zA-Z0-9]|'\_')\* .\*? '('.\*?')';

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

INTEGER : [0-9]+ ;

FLOAT : [0-9]+','[0-9]+;

STRING : '"'(~["])\*'"'; // Tout caractère excepté " entre deux "

// Commentaires

UNELIGNE : '//'~[\n]\* -> skip;

MULTILIGNE : '/\*' (.)\*? '\*/' -> skip; // le ? est pour avoir une expression non avide

WS : [ \n\t\r] -> skip;

Problème rencontré :

Ambiguïté entre le nom du programme et les identificateurs de variables. Comme le nom du programme et les identificateurs sont des chaines de caractères qui, pour la part des identificateurs acceptent majuscules, minuscules et chiffres alors que le nom du programme accepte les mêmes caractères en plus du tiret ‘\_’ et qu’il doit absolument commencer par une majuscule. Cette petite différence n’empêche pas que le nom du programme soit reconnu en tant qu’identifiant ou inversement (dans certains cas).

Solution trouvée :

Inclure les parenthèses qui suivent le nom du programme dans l’expression régulière en permettant les espaces.

1. Analyse syntaxique

Dans cette partie nous avons défini la structure du langage en écrivant une grammaire syntaxique qui permet de reconnaitre des terminaux et des non terminaux. Nous avons donc défini notre règle de début ‘programme’. Puis nous avons développé chaque non terminal et Terminal afin d’obtenir le bon arabe syntaxique.

Voici un exemple des règles que nous avons définies, (en sachant que les terminaux en ANTLR commencent par des majuscules et les non terminaux doivent commencer par une minuscule).

programme : COMPIL NOM\_PROGRAMME'{' declarations START instructions '}';

declarations : declarations type ids ';'

| ;

instructions : instruction instructions? ;

boucle : DO '{' instructions '}' WHILE cdtDO;

condition : IF cdtIF THEN '{' instructions '}' (instElse '{' instructions '}')\*;

ecrire : PRINTFCOMPIL '(' (STRING | ids) ')';

lire : SCANCOMPIL '(' ids ')';

Toutes nos règles lexicales ont été enregistrés dans un fichier. g4, nous y ajoutons les règles syntaxiques, ce fichier va être utilisé par le générateur ANTLR pour définir les interfaces java qui permettent de continuer d’analyser le code présenté au compilateur.

Définition de l’ordre de priorité des opérateurs :

Définition de l’ordre de priorité des opérateurs :

Lorsqu'il y a plus d'un opérateur arithmétique dans une expression, l'ordre dans lequel les opérations doivent être effectuées dépend de l'emplacement des règles, tel que :

* Les règles des opérateurs prioritaire doivent être placé avant les règles opérateurs moins prioritaire.

exp : MOINS exp  
 | parG exp parD  
 | exp opMD exp  
 | exp opPM exp  
 | INTEGER  
 | FLOAT  
 | ID  
 ;  
parG: '(';  
parD: ')';

* Les opérateurs de même force doivent se retrouver dans les mêmes règles.

opMD : MUL | DIV;  
opPM : PLUS | MOINS;

Problèmes rencontrés :

Le problème rencontré dans cette partie c’était la reconnaissance des nombres négatifs, effectivement dans la parie lexicale vous remarquez que nous ne les reconnaissons pas, par contre nous reconnaissons ‘MOINS expression’, ce qui nous permet de les traiter plus tard dans la partie sémantique.

Nous n’avons pas vraiment rencontré d’autres problèmes à ce niveau, mais une fois arrivés à la partie sémantique nous nous sommes retrouvés obligées à changer quelques règles de notre grammaire.

Voici un exemple de la première version du langage :

Nous avons rencontré des difficultés à reconnaitre le signe ‘-‘ d’une parenthèse, ainsi que les différents opérateurs.

boucle : DO '{' instructions '}' WHILE cdt;

condition : IF cdt THEN '{' instructions '}' (ELSE '{' instructions '}')\*;

cdt : '('exp oplog exp')';

exp : '-' exp

| '(' exp ')'

| exp ('\*'|'/') exp

| exp ('+'|'-') exp

| INTEGER

| FLOAT

| ID

;

La génération du code objet nécessite d’enregistrer la position courante, ce qu’on ne peut pas effectuer à la fin de terminaux, et c’est pourquoi nous y sommes revenus plus tard pour transformer quelques terminaux en non terminaux afin de permettre à ANTLR de surveiller ces entités et nous permettre d’opérer leur de leur rencontre.

Solution trouvée :

Le code ci-dessous à été transformé de la façon suivante :

boucle : DO '{' instructions '}' WHILE cdtDO;

condition : IF cdtIF THEN '{' instructions '}' (instElse '{' instructions '}')\*;

instElse : ELSE; // pour pouvoir génerer les quadruplets

cdtIF : '('exp oplog exp')';

cdtDO : '('exp oplog exp')';

exp : MOINS exp

| parG exp parD

| exp opMD exp

| exp opPM exp

| INTEGER

| FLOAT

| ID

;

1. Gestion des erreurs

Pour récupérer les erreurs lexicales et syntaxiques rencontrés nous avons utilisé la classe **‘BaseErrorListener’** propre à ANTLR, nous insérons ces erreurs dans une structure qu’on passe aux parties suivantes, on affiche le contenu de cette entité à la fin de la compilation afin de permettre au compilateur de détecter toutes les erreurs possibles au lieu d’opérer une erreur à la fois.

Partie 2 : L’analyse sémantique et le code intermédiaire

1. Analyse sémantique

Dans cette partie nous avons donné une sémantique aux entités précédemment reconnues, nous les avons aussi insérés dans la table des symboles et générés le code intermédiaire.

Nous avons utilisé les classes java générées par l’outil ANTLR, nous avons choisi de travailler avec la classe des ‘Listener’ dans notre projet, par choix nous avons séparé notre travail en plusieurs classes :

Routines de la table des symboles, ou nous avons insérée nos items dans la table après les vérifications suivantes :

1. Table des symboles

La table de symboles est créée lors de la phase de l’analyse lexicale. Elle regroupe l’ensemble des variables reconnues avec leurs type, valeur et si elles étaient déclarées ou pas.

Cette table sera mise à jour au fur et à mesure de l’avancement de la compilation :

lors de la déclaration d'une variable**,** son utilisation (pour vérifier si elle a été déclaré) **et** lors de la mise à jour de sa valeur.

Pour l’implémentation de la TS nous avons utilisé une ArrayList de ligne, tel que chaque ligne est composée des (5) champs suivants :

* Name : une chaine de caractère, représentant le nom de la variable.
* Declared: un entier, permettant de vérifier si une variable utilisée dans le code a été déclaré auparavant ou pas. (0: var non déclarée / 1: var déclarée)
* Type : un entier, représentant le type de la variable (1:intcompil, 2:floatcompil, 3:stringcompil)
* Init : un booléen, permettant de vérifier si une variable a été déjà initialisé ou pas.
* Value : une chaine de caractère, contenant la valeur d’une variable.

Pour la manipulation des éléments de la TS, nous avons utilisé les méthodes suivantes:

* toString et display : pour afficher la Table des Symboles.
* getRowByName : pour récupérer une ligne de la TS à partir du nom d’une variable.
* add : pour insérer une nouvelle ligne dans la TS.
* contains: pour vérifier l’existence d’une variable dans la TS.
* Declare : pour mettre à jour la valeur de declared.
* getSize : pour connaitre le nombre de ligne dans la TS.

Tests : pour le code suivant :

compil Nom\_du\_programme ()

{

intCompil a;

floatCompil b;

stringCompil c;

start

a = 10 \* 3;

b = 8/3 + 1,9;

c = "Chaine simple";

}

Sortie du programme :

SYMBOLS TABLE

%%%%%%%%%%%%%%%%%%%%.%%%%%%%%%%%%%%%%%%%%.%%%%%%%%%%%%%%%%%%%%%%%.%%%%%%%%%%%%%%%%%%%%

Identifier | Type | Declared | Value |

%%%%%%%%%%%%%%%%%%%%.%%%%%%%%%%%%%%%%%%%%.%%%%%%%%%%%%%%%%%%%%%%%.%%%%%%%%%%%%%%%%%%%%

| a | intCompil | DECLARED | 30

| b | floatCompil | DECLARED | 4.5666666

| c | stringCompil | DECLARED | "Chaine simple"

======================================================================================

1. Routines sémantiques

Après initialisation de notre Listener dans la classe Main du programme les routines de notre classe ‘routinesTS extends TinyLanguageSIIBaseListener’ vont être déclenchés à chaque entrée ou sortie d’un non terminal. Voici une brève description de chaque méthode implémentée :

* 1. Structures utilisées

La classe ‘elementValue’ nous permet de garder une valeur sous forme de chaine de caractère et son type. Nous utilisons lors des routines une ‘HashMap typeValeur’ qui garde le contexte d’une entité ainsi que son ‘elementValue’, qu’on vide et libère à la fin de la partie sémantique. Nous avons aussi implémenté toutes les méthodes qui nous permettent de manipuler cette structure. Une structure Table des symboles a aussi été déclarée.

* 1. Sortie de l’entité ‘Déclaration’

A chaque déclaration on vérifie le type déclaré et on l’affecte à chaque identificateur se trouvant à sa droite. Nous insérons ce type dans la table des symboles ainsi que dans la HashMap typeValeur temporaire. Lors de cette étape une double déclaration peut être détectée

.

* 1. Sortie de l’entité ‘exp’ (expression)

C’est une entitée un peu compliquée ou on a plusieurs cas qui peuvent générer une erreur, voici les vérifications effectuées.

* Si c’est un ‘INTEGER’ ou un ‘FLOAT’ on garde la valeur dans la HashMap, si c’est un ID on met à jour la table des symboles, s’il a été déclaré sinon c’est une erreur de déclaration.
* Si une ‘exp’ a été détecté, si c’est un ‘ID’ alors vérifier la déclaration, après vérifier son instanciation.
* Si la branche détectée est un ‘exp opMD exp’ vérifie si l’opérateur est une division, alors on vérifie que le coté droit est différent de ‘0’
* Si c’est une opération on vérifie la compatibilité puis on effectue les calculs et on garde le résultat dans la HashMap et dans la TS dans le cas d’un ID ;
  1. Sortie d’une ‘affectation’

Dans cette partie on vérifie si l’identificateur a été déclaré, puis on vérifie la compatibilité des types, et on récupère la valeur de la HashMap et on met à jour la TS.

Dans le cas de l’affectation d’un nombre flottant à un entier on effectue un cast.

* 1. Sortie des opérations ‘Ecrire’ et ‘Lire’

Ici on n’a qu’une seule vérification à faire, c’est que tous les identificateurs ont été déclarés.

1. Gestion des erreurs

De la même façon qu’on a opéré dans la partie précédente, nous avons passé à notre classe une structure d’erreur à laquelle on y ajoute nos erreurs sémantiques, on n’arrête pas la partie sémantique à chaque erreur, par conte nous n’avançons pas jusqu’au code intermédiaire ni au code objet, vu que cela ne servira qu’à générer de faux résultats.

Nous affichons aussi la ligne et la colonne de l’erreur ainsi que son type. (Voir l’exemple dans la page suivante)

* 1. Erreur de double déclaration

A chaque déclaration, nous avons vérifié si l’item a déjà été inséré dans la table des symboles, dans le cas affirmatif nous avons généré l’erreur suivante :

1. ERR\_DOUBLE\_DEC =" double declaration : ";
   1. Déclaration d’une variable

Dans le cas ou on est dans la partie instructions, nous devons vérifier pour chaque identificateur qu’on rencontre qu’il a été initialisé auparavant. Dans le cas de la rencontre d’une variable non déclarée l’erreur suivante est générée :

1. ERR\_NO\_DEC =" variable non declared : ";
   1. Instanciation d’une variable

On vérifie le cas où l’un des côtés d’une expression à calculer est un identifiant auquel on n’a rien affecté encore, on génère alors l’erreur.

1. ERR\_INSTANTIOATION =" variable non instatiated : ";
   1. Division par 0

Cette erreur survient lors d’une division par 0 ou un ID ayant la valeur de 0, ou null.

1. ERR\_DIV\_Z =" numeric overflow : ";
   1. Incompatibilité des types

Cette erreur survient lors d’une affectation de deux types incompatibles, par exemple affecter une chaine de caractère à un float ou un float à un entier. Ou bien lors d’une opération arithmétique entre deux identifiants, ou l’un est une chaine de caractères et l’autre un entier ou un nombre flottant.

1. ERR\_INCOMPATIBILITY =" type mismatch : ";

Ce qui est de la compatibilité entre les nombres entiers et flottants, nous avons permis à un entier de recevoir un flottant casté.

Code exemple :

compil Nom\_du\_programme ()

{

intCompil a,c;

floatCompil a;

start

a = 10 \* 3;

b = 5;

c = c/0;

c = "Chaine simple";

}

Sortie du programme et contenu de la table des symboles :

Program compiled with the following errors :

Semantic error at line 3:4 double declaration : a

Semantic error at line 7:4 variable non declared : b

Semantic error at line 8:8 variable non instatiated : c

Semantic error at line 8:8 numeric overflow : c/0

Semantic error at line 9:4 type mismatch : c

SYMBOLS TABLE

%%%%%%%%%%%%%%%%%%%%.%%%%%%%%%%%%%%%%%%%%.%%%%%%%%%%%%%%%%%%%%%%%.%%%%%%%%%%%%%%%%%%%%

Identifier | Type | Declared | Value |

%%%%%%%%%%%%%%%%%%%%.%%%%%%%%%%%%%%%%%%%%.%%%%%%%%%%%%%%%%%%%%%%%.%%%%%%%%%%%%%%%%%%%%

| a | intCompil | DECLARED | 30

| c | intCompil | DECLARED | null

======================================================================================

1. Génération du Le code intermédiaire

Pour de générer notre code nous avons utilisé une structure similaire à la table des symboles dans la quelle on garde nos quadruplets. Une ligne est une classe quadruplet ayant quatre champs visibles (opération, opérande1, opérande2, temporaire), nous avons aussi implémenté les méthodes qui servent à manipuler les quadruplets et leur table.

1. Routines Quadruplets

Là encore nous avons crée une nouvelle classe pour gérer la création et la mise à jour de nos quadruplets, nous aurions bien pu mettre nos routines dans le même fichier que les routines de la table des symboles mais nous avons préféré les séparer. La classe ‘routinesQuad extends TinyLanguageSIIBaseListener’ elle aussi. Voici une brève description de chaque méthode implémentée :

* 1. Structures utilisées :

La même structure d’erreurs utilisée dans les parties précédentes passée en paramètres, Une pile pour garder les temporaires, une HashMap de branchement pour faciliter la traduction des opérateurs logiques en leur branchement équivalents ainsi qu’un compteur de temporaires.

compil Nom\_du\_programme ()

{

intCompil i, k;

floatCompil a ;

stringCompil d;

start

printCompil ("valeur");

d = "chaine";

i=6; k=55;

if (i<k) then {

k= k/2;

do {

i = i +1;

}while (i<k)

}

}

* 1. Sortie d’une condition :

On génère un quadruplet avec un le branchement équivalent à l’opérateur utilisé dans l’opération ainsi que les deux dernières valeur mises dans la pile (si on sort d’une condition, les deux dernières expressions rencontrées sont celles qui lui correspondent), on laisse l’adresse du branchement vide.

* 1. Sortie d’une instruction else :

La raison pour laquelle on avait transformé le token ‘else ‘ en non terminal dans la partie syntaxique ‘est pour pouvoir avoir accès à sa méthode de sortie.

Ici on met à jour le quadruplet généré lors de la rencontre de la condition qui suit une instruction ‘if’. On insère aussi un nouvel quadruplet de branchement en laissant l’adresse vide.

* 1. Sortie d’une instruction ‘if’

On met à jour l’adresse du branchement de la condition rencontrée vers le début de l’instruction ‘else’.

* 1. Sortie d’une boucle :

On met à jour le quadruplet généré lors de la sortie de la condition du ‘Do’ rencontré.

* 1. Entrée de boucle :

On sauvegarde l’index de l’instruction suivante pour pouvoir lui revenir et boucler.

* 1. Sortie des instructions Entrée et Sortie et programme :

Dans ce cas on génère un quadruplet READ, WRITE ou END. Sans besoin d’aucun branchement

Exemple d’exécution :

Code en entrée :

compil Nom\_du\_programme ()

{

intCompil i, k;

floatCompil a ;

stringCompil d;

start

printCompil ("valeur");

d = "chaine";

i=6; k=55;

if (i<k) then {

k= k/2;

do {

i = i +1;

}while (i<k)

}

}

Sortie su programme :

No semantic errors found !

SYMBOLS TABLE

%%%%%%%%%%%%%%%%%%%%.%%%%%%%%%%%%%%%%%%%%.%%%%%%%%%%%%%%%%%%%%%%%.%%%%%%%%%%%%%%%%%%%%

Identifier | Type | Declared | Value |

%%%%%%%%%%%%%%%%%%%%.%%%%%%%%%%%%%%%%%%%%.%%%%%%%%%%%%%%%%%%%%%%%.%%%%%%%%%%%%%%%%%%%%

| i | intCompil | DECLARED | 7

| k | intCompil | DECLARED | 27

| a | floatCompil | DECLARED | null

| d | stringCompil | DECLARED | "chaine"

======================================================================================

%%%%%%%%%%%%%%%%%%%%%%%%%%

QUADRETS

%%%%%%%%%%%%%%%%%%%%%%%%%%

0- (WRITE,"valeur", , )

1- (=, ,"chaine",d)

2- (=, ,6,i)

3- (=, ,55,k)

4- (BL,k,i,10)

5- (/,k,2,T1)

6- (=, ,T1,k)

7- (+,i,1,T2)

8- (=, ,T2,i)

9- (BL,k,i,7)

10- (END, , , )

Partie 3 : Génération du code objet.

Dans cette partie nous transformons nos quadruplets en code assembleur. Pour cela nous avons crée une classe ‘ocGenerator’ ou bien ‘Générateur de Code Objet’. Cette classe inclut un attribut de classe ‘Assembler Instruction’ et une méthode ‘Assemble’ qui prends en entrée une table de quadruplets et retourne une ArrayList de ‘AssemblerInstruction’.

D’abord nous avons commencé par définir une hashMap qui nous a permis à éviter les répétitions et à transformer rapidement les opérateurs en instructions assembleur. La structure est initialisée en même temps que notre ‘ocGenerator’

private static HashMap<String,String> quadToCo = new HashMap<>();

public ocGenerator() {

quadToCo.put("+","ADD");

quadToCo.put("-","SUB");

quadToCo.put("\*","MUL");

quadToCo.put("/","DIV");

quadToCo.put("READ", "INPUT");

quadToCo.put("WRITE", "OUTPUT");

quadToCo.put("BG", "JG");

quadToCo.put("BL", "JL");

quadToCo.put("BGE", "JGE");

quadToCo.put("BLE", "JLE");

quadToCo.put("BE", "JE");

quadToCo.put("BNE", "JNE");

quadToCo.put("BR", "JMP");

quadToCo.put("END", "END");

}

Dans la méthode ‘Assemble’ on parcourt chaque quadruplet en le transformant en code objet de la façon suivante :

|  |  |
| --- | --- |
| Quadruplet | Instructions en Assembleur |
| (Opérateur, op1, op2, resultat) | Mov AX, op1  Branchements.get(opérateur), AX, op2  MOV resultat, AX |
| (=, ,op\_source, op\_cible) | MOV AX, op\_source  MOV op\_cible, AX |
| (Branchement\_cond, @, op1, op2) | MOV AX, op1  CMP AX, op2  Branchements.get(Branchement\_cond) etiq@ |
| (Branchement, @, , ) | JUMP etiq@ |
| (READ, var, , ) | INPUT var |
| (WRITE, var, ) | OUTPUT var |

Le plus grand obstacle rencontré dans cette étape a été de bien placer les étiquettes afin de bien transformer les branchements. Pour cela nous avons défini une méthode ‘needsEtiq’ qui retourne une instruction assembleur ainsi qu’un simple tableau de la longeur de nos quadruplets, avec chaque valeur initialisée à 0 ;

Sans utiliser un moyen de garder trace des quadruplés sur lesquels on pointe, la numérotation des étiquettes ne sera pas cohérente vu qu’un quadruplet peut être traduit en plus d’une instruction assembleur.

Exemple du fonctionnement de cette méthode :

compil Nom\_du\_programme ()

{

intCompil i, k;

start

k = 8;

i=6;

if (i<k) then {

k=55;

} else {

i=6;

}

}

%%%%%%%%%%%%%%%%%%%%%%%%%%

QUADRETS

%%%%%%%%%%%%%%%%%%%%%%%%%%

0- (=, ,8,k)

1- (=, ,6,i)

2- (BL,k,i,5)

3- (=, ,55,k)

4- (BR, , ,6)

5- (=, ,6,i)

6- (END, , , )

%%%%%%%%%%%%%%%%%%%%%%%%%%

OBJECT CODE

%%%%%%%%%%%%%%%%%%%%%%%%%%

start:

MOV AX, 8

MOV k , AX

MOV AX, 6

MOV i , AX

MOV AX, k

CMP AX, i

JL etiq5

MOV AX, 55

MOV k , AX

JMP

JMP etiq6

etiq5 MOV AX, 6

MOV i , AX

etiq6 END

code ends

end start

%%%%%%%%%%%%%%%%%%%%%%%%%%

///////////////////---- COMPILING COMPLETED ----///////////////////

D’abord on parcourt nos quadruplés en les transformant en code assembleur, arrivé au quadruplet 2 avec un branchement, on va mettre JL « etiq » + Quadruplet[5] et on va mettre la valeur de etiquette[5] à 5.

Pour chaque quadruplet, à la première instruction on appelle ‘needsEtiq’ avec l’index du quadruplet, dans le cas du Quadruplet 5 la méthode va trouver une valeur positive dans le tableau etiquette et lui donner une etiquette ayant la valeur « etiq » + etiquette[5].

Partie 4 Test du compilateur crée :

Classe Main de notre programme :

1. public class Main {
2. public static void main(String[] args) {
3. try {
4. ArrayList<String> errors = new ArrayList<>();
6. // create a CharStream that reads from standard input
7. CharStream input = CharStreams.fromFileName("src/test.txt");
9. // create a lexer that feeds off of input CharStream
10. TinyLanguageSIILexer lexer = new TinyLanguageSIILexer(input);
12. // create a buffer of tokens pulled from the lexer
13. CommonTokenStream tokens = new CommonTokenStream(lexer);
15. //Handeling errors
16. errorListener eListener = new errorListener(errors);
18. // create a parser that feeds off the tokens buffer
19. TinyLanguageSIIParser parser = new TinyLanguageSIIParser(tokens);
20. parser.removeErrorListeners();
21. parser.addErrorListener(eListener);
22. parser.getInterpreter().setPredictionMode(PredictionMode.LL\_EXACT\_AMBIG\_DETECTION);
24. // begin parsing at chosen rule
25. ParseTree tree = parser.programme();
27. // create the Symbols Table and initiate it with an errors array
28. routinesTS routinesTabSymbol = new routinesTS(errors);
29. routinesQuad routinesQuadruplets = new routinesQuad(errors);
31. //A tree walker that can trigger callbacks
32. ParseTreeWalker walker = new ParseTreeWalker();
34. // Walk the tree created during the parse
35. walker.walk(routinesTabSymbol, tree);
36. ocGenerator oc = new ocGenerator();
38. routinesTabSymbol.getTable().display();
39. if (errors.size()==0) {
40. walker.walk(routinesQuadruplets, tree);
41. routinesQuadruplets.getTable().display();
42. oc.Assemble(routinesQuadruplets.getTable());
43. oc.display();
44. }
45. } catch (Exception e){
46. if (e.getMessage() != null){
47. System.out.println("Exception caught at line "+e.getStackTrace()[0].getLineNumber()+": "+e.getMessage());
48. } else {
49. e.printStackTrace();
50. }
51. }
52. }
53. }

Contenu du fichier test.txt

compil Nom\_du\_programme ()

{

intCompil a, t1;

floatCompil b, t2;

stringCompil c;

start

t1 =8;

a = 10 \* 3;

if (t1 > 0) then {

b = a + 1 ;

}else{

do {

t2 = a + 1 ;

t1 = t1+1 ;

b = t1 / t2 ;

} while (t1 != 10)

}

printCompil ("test du projet") ;

printCompil (c) ;

}

Sortie du programme :

No semantic errors found !

SYMBOLS TABLE

%%%%%%%%%%%%%%%%%%%%.%%%%%%%%%%%%%%%%%%%%.%%%%%%%%%%%%%%%%%%%%%%%.%%%%%%%%%%%%%%%%%%%%

Identifier | Type | Declared | Value |

%%%%%%%%%%%%%%%%%%%%.%%%%%%%%%%%%%%%%%%%%.%%%%%%%%%%%%%%%%%%%%%%%.%%%%%%%%%%%%%%%%%%%%

| a | intCompil | DECLARED | 32.0

| t1 | intCompil | DECLARED | 0.28125

| b | floatCompil | DECLARED | 0.28125

| t2 | floatCompil | DECLARED | 32.0

| c | stringCompil | DECLARED | null

======================================================================================

QUADRETS

%%%%%%%%%%%%%%%%%%%%%%%%%%

0- (=, ,8,t1)

1- (\*,10,3,T1)

2- (=, ,T1,a)

3- (BG,0,t1,7)

4- (+,a,1,T2)

5- (=, ,T2,b)

6- (BR, , ,14)

7- (+,a,1,T3)

8- (=, ,T3,t2)

9- (+,t1,1,T4)

10- (=, ,T4,t1)

11- (/,t1,t2,T5)

12- (=, ,T5,b)

13- (BNE,10,t1,7)

14- (WRITE,"test du projet", , )

15- (WRITE,c, , )

16- (END, , , )

%%%%%%%%%%%%%%%%%%%%%%%%%%

OBJECT CODE

%%%%%%%%%%%%%%%%%%%%%%%%%%

start:

MOV AX, 8

MOV t1 , AX

MOV AX, 10

MUL AX, 3

MOV T1 , AX

MOV AX, T1

MOV a , AX

MOV AX, 0

CMP AX, t1

JG etiq7

MOV AX, a

ADD AX, 1

MOV T2 , AX

MOV AX, T2

MOV b , AX

JMP

JMP etiq14

etiq7 MOV AX, a

ADD AX, 1

MOV T3 , AX

MOV AX, T3

MOV t2 , AX

MOV AX, t1

ADD AX, 1

MOV T4 , AX

MOV AX, T4

MOV t1 , AX

MOV AX, t1

DIV AX, t2

MOV T5 , AX

MOV AX, T5

MOV b , AX

etiq13 MOV AX, 10

CMP AX, t1

JNE etiq7

etiq14 OUTPUT "test du projet"

OUTPUT c

END

code ends

end start

%%%%%%%%%%%%%%%%%%%%%%%%%%

///////////////////---- COMPILING COMPLETED ----///////////////////