Université des Sciences et de la Technologie Houari Boumediene

Réalisation d'un mini compilateur pour le langage 'TinyLanguage_SII' Avec l'outil ANTLR

Binôme 14 :

-Walid BENCHABEKH
-Aimen Said MEZABIAT

Professeur :

-Dr Lylia BETIT

1 Analyse Lexicale

C'est la première phase de la compilation. Elle sert à extraire des entités lexicales que nous allons utiliser par la suite dans la prochaine phase de la compilation, l'analyse syntaxique, afin d'analyser la syntaxe d'un programme.

C'est entités lexicales peuvent être des identifiants (x, Var2, toto, ...), des mots-clefs (if, then, while, ...), des signes de ponctuation ({, },), ...), des opérateurs (>, !=, ==, ...), des littéraux (3.88, 7, « Salut », ...), des commentaires (//max des deux valeurs ...) …

On commence d'abord par définir l'expression régulière du PROGNAME (nom du programme) avant la définition ID (identifiants des variables) pour que le nom du programme ne soit pas considéré comme un identificateur d'une variable et ne pas l'insérer dans la Table des Symboles plus tard dans l'analyse sémantique.

Pour cela, nous avons créé un fichier .g4 pour qu'il soit reconnu par ANTLR comme suit :

```
PROGNAME : [A-Z]+[a-zA-Z0-9_]* ;
ID : [a-zA-Z]+[a-zA-Z0-9]* ;

INT :   '0' | [1-9] [0-9]* ;

FLOAT
    :   '-'? INT '.' INT
    |   '-'? INT
    ;


COMMENT : '/*' .*? '*/' -> channel(HIDDEN) ;

TEXT : '"' (~'"'|'\\"')* '"'  ;

WS : [ \t\n\r]+ -> channel(HIDDEN) ;
```

2 Analyse Syntaxique

Toujours dans le même fichier .g4, nous avons défini la structure générale d'un programme acceptable par notre langage, qui se présente comme suit :

```
start_rule : 'compil' PROGNAME '(' ')'
'{' declarations 'start' instructions '}';
```

Puis on a défini la structure pour chaque non terminale en respectant les règles LL(k) ainsi que les spécifications de l'énoncé.

Ci-dessous un exemple, l'instruction d'affectation :

```
instAff:    identifier '=' expression;
```

Avec :

instAff : la règle de définition d'une instruction d'affectation.

identifier : La règle qui défini un identifiant ID, comme suit :

```
identifier : ID ;
```

expression : la règle de définition de la composition d'une expression.

Elle est composée d'une succession de valeurs, identifiants et opérations arithmétiques, et prend en considération la priorité des opérateurs et leur ordre d'apparition.

```
expression : expression pm expression1 | expression1;
expression1 : expression1 md expression2 | expression2;
expression2 : identifier | '(' expression ')' | value ;
```

Avec la définition suivante des règles pm et md :

pm représente la première lettre des deux mots ”Plus” ”+” et ”Moins” ”-”.

md représente la première lettre des deux mots ”Multiplication” ”*” et ”Division” ”/”.

```
pm : (PLUS|MINUS) ;
md : (MULT|DIV) ;
```

ainsi de suite pour toutes les instructions demandées (if-then-else, do - while, printCompil , ...).

En plus de cela pour chaque terminal devant déclencher une action lors de sa rencontre par l'analyseur, il doit être définit comme règle unaire (qui génère un seul terminale ou mot clé de notre langage).

Puis remplacer toutes ses utilisations (références) par la nouvelle règle le générant ;

Ci-dessous quelques exemples :

```
WHILE : 'while';
DO : 'do';
PLUS : '+';
MINUS : '-';
MULT : '*';
```

3 Analyse Syntaxique

Lors de cette phase, nous avons donné une sémantique à nos entités, et vérifié la cohérence de notre programme à travers des routines (avec insertion dans la table des symboles) ,

Aussi nous avons généré la forme intermédiaire correspondante au langage développé sous forme de quadruplets.

Dans ce qui suit nous allons détailler les classes crées à cet effet :

Table des symboles :

Notre table des symboles (dans la classe TabSymbole) est une ArrayList de ligne, tel que :

Chaque ligne est composée de trois informations qui sont :

- Name : l'identifiant de la variable
- Type : le type de la variable (1 pour int , 2 pour float , 3 pour string).
- Declared : booleen permettant de vérifier si une variable utilisé dans le code a été déclaré au préalable.

Aussi nous avons les méthodes de manipulation des éléments de la Table des Symboles :

- getLigne : pour récupérer une ligne `a partir du nom d'une variable.
- containsLigne : pour tester si une variable existe dans la TS.
- addLigne : pour insérer une nouvelle ligne `a la Table des Symboles.
- getSize : pour connaitre le nombre de ligne dans la TS.
- toString et display : pour afficher l'intégralité de la Table des Symboles.

Routines :

Dans la classe « RoutinesTabSymbol » que nous avons fait hériter du baseListner nous avons pour chaque événement enter ou exit d'une règle implémenté les testes (routines) nécessaires, nous pouvons citer :

- Empêcher les doubles déclarations.
  En vérifiant le contenu de la table des symboles, si l'identifiant y est inséré, on vérifie s'il a été déclaré (champs DECLARED), sinon on génère une erreur, que l'on insère dans notre table errors.
- Tester la compatibilité des opérandes lors des affectations, calcule des expressions ou comparaisons.
  Pour cela nous avons utilisé un HashMap dans le quel nous avons sauvegardé le contexte comme clé puis le type de variable ou expression comme valeur, afin de tester la compatibilité avec le reste des éléments interagissant avec cette variable.
- Empêcher l'utilisation de variables non déclarées.
  A travers la recherche dans la table des symboles à chaque utilisation d'une variable.

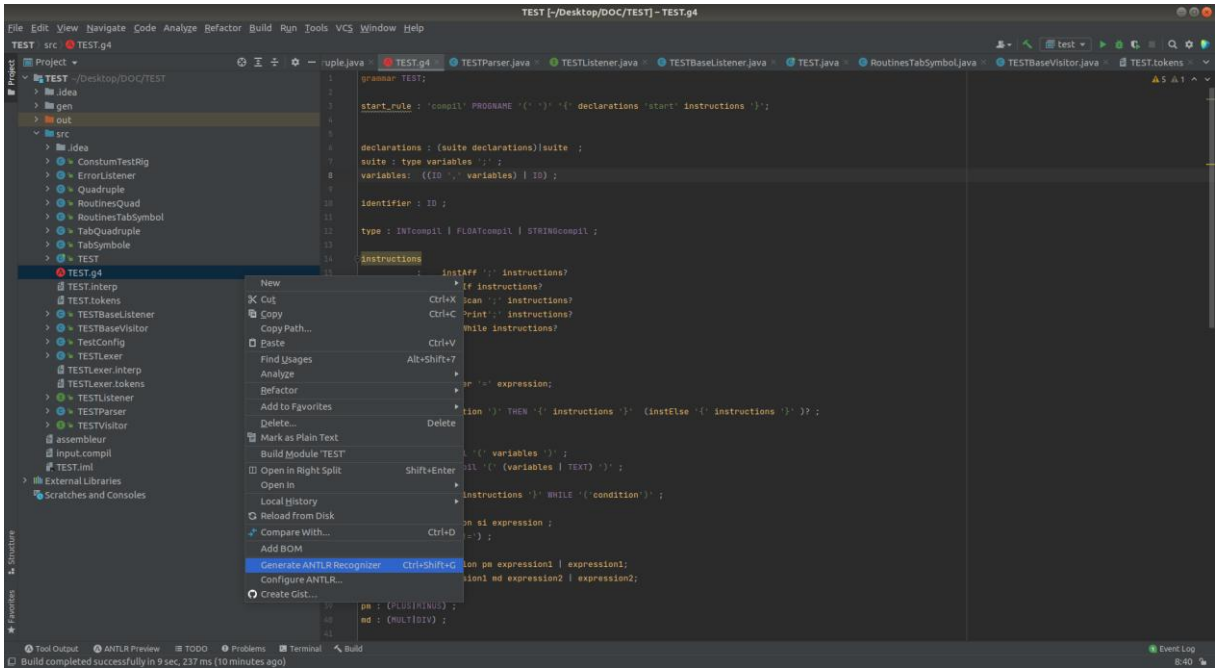
Mais aussi géré l'insertion des variables dans la TS lors des déclarations,

Intellij IDE :

- Génération des parser et listner :

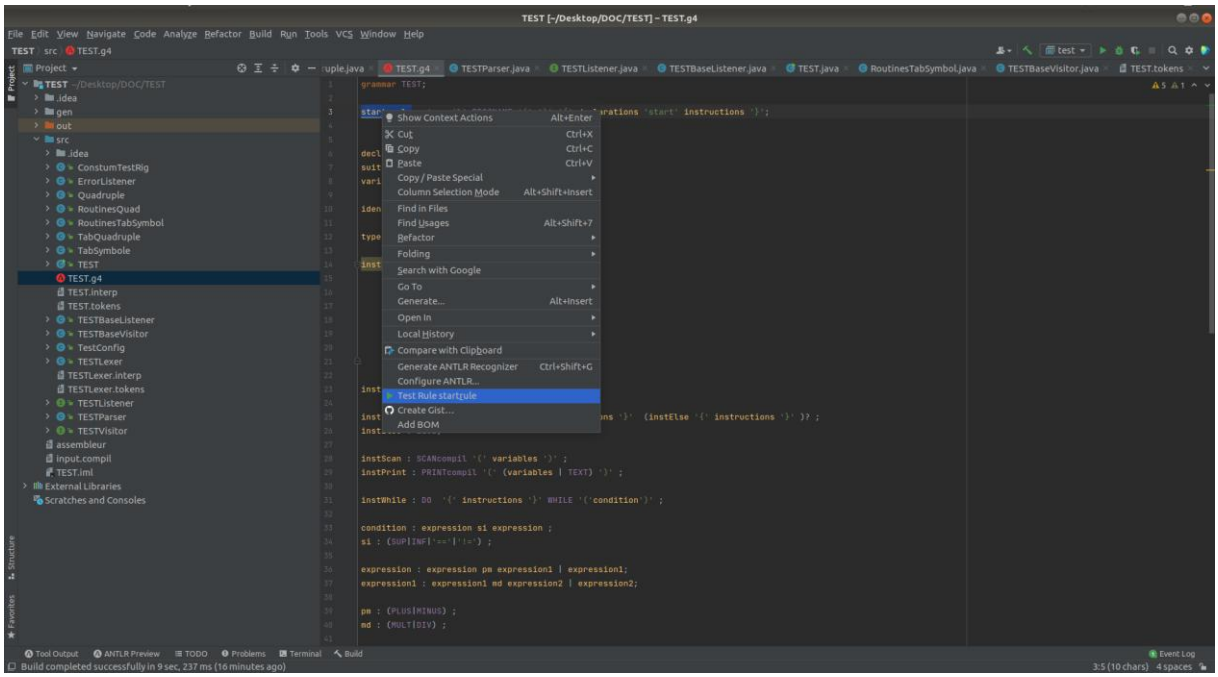
  Pour cela on doit faire un clic droit sur notre fichier .g4 puis, sélectionner l'option : Generate ANTLR recongnizer.

  Ce qui engendre la création des quatre classes :
  BaseListner , BaseVisitor , Lexer et Parser
  en plus des Interface et fichiers tokens ...

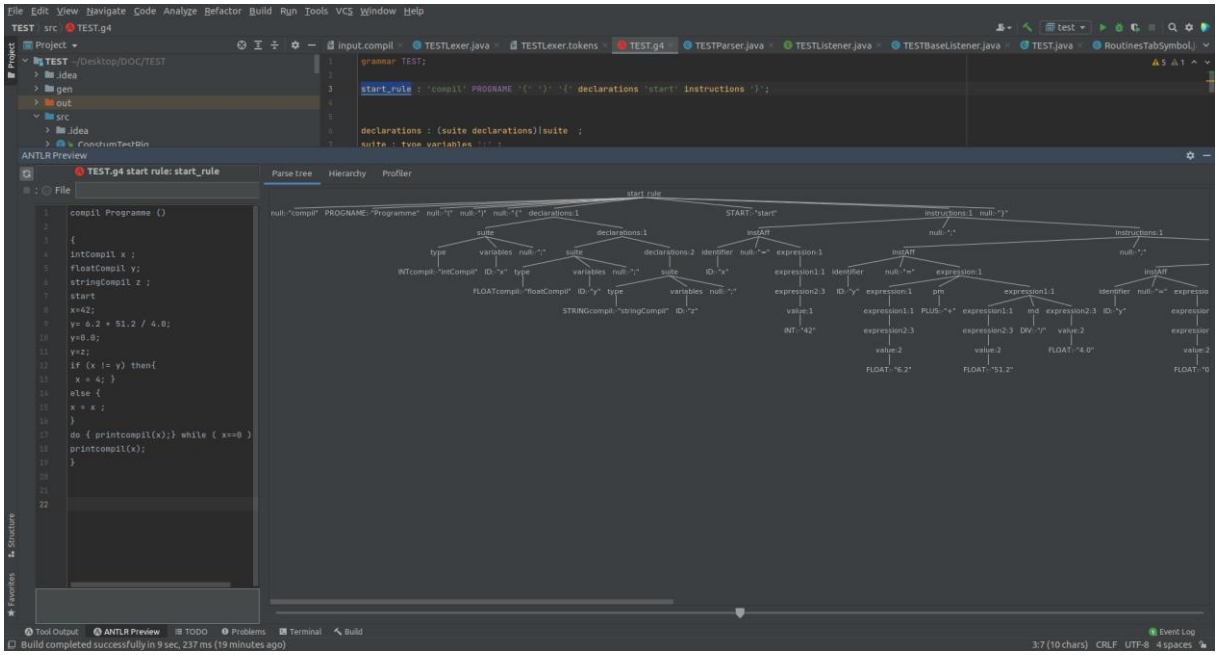


- Tester les règles (arbre)

Afin de générer l'arbre syntaxique et de tester nos règles produite lors de l'analyse syntaxique, nous devons faire un clic droit sur la règle à tester puis sélectionner l'option : Test rule start rule comme suit :



Ainsi si aucun conflit n'a été détecté l'arbre sera affiché comme ci-dessous :



Et si un ou plusieurs conflits ont été détectés, l'arbre affichera l'emplacement du conflit.

- Interface et organisation des codes

Voici comment les codes sont organisés dans notre projet :

- Dossier **gen** : qui contient les fichiers généré par ANTLR.

- Dossier **src** : qui contient les fichiers que nous avons créé pour les trois types d'analyse.

- Fichier **input.compil** : qui contient le code à tester, écrit dans le langage développé.

- Fichier **Assembleur** : vide à la base, mais destiné à être rempli suite à l'exécution du projet par le code objet correspondant au code testé.

- Création de la classe des routines

Dans notre projet afin de mieux concevoir et gérer nos routines, nous avons opté pour deux classe héritant de BaseListner (c'est à dire nous avons deux classes des routines).

1. Classe **RoutinesTabSymbol** : pour tout ce qui est en relation avec la Table des Symboles et les routines sémantique.

2. Classe **RoutinesQuad** : pour la génération des Quadruplets et du code Objet.

4 Génération du Code Intermédiaire

Afin de générer nos quadruplets nous avons créé plusieurs classes, tel que :

- Quadruple :

Représentant la structure d'un quadruplet, c'est-à-dire les quatre champs dont est composé un quadruplet (opération, opérande1, opérande2, temporaire) ;

Ainsi que les actions (méthodes) applicable sur ces derniers, nous citons : le constructeur, les getteurs et les setteurs.

- TabQuadruple :

Constitue la une table de quadruplet (LinkedList d'éléments de la classe Quadruple) en plus des opérations élémentaire dont :

-Ajouter un quadruplet a la liste

-Mise à jours d'un quadruplet (pour les branchements)

-Récupérer un quadruplet d'après son indice dans la liste

-Récupérer la taille de la liste

-Affichage de la table des quadruplets entièrement

- RoutinesQuad :

Cette classe hérite du « BaseListener », elle contient principalement les méthodes de création de quadruplets selon l'événement rencontré (enter ou exit d'une règle de la partie syntaxique).

Faisant usage d'une LinkedList géré comme une pile afin d'empiler les opérandes des expressions arithmétiques pour gérer les priorités.

Conclusion :

Ainsi lorsque nous serons en train de « parser » notre échantillon de code respectant la syntaxe de notre langage, à chaque rencontre d'une règle possédant un listner dans notre classe « RoutinesQuad » le quadruplet correspondant sera construit puis inséré dans la table des routines instanciées à partir de la classe « TabQuadruple » , puis nous pouvant les afficher ou/et les utiliser par la suite ; Dans notre cas nous avons affiché la table des quadruplets puis utilisé cette table pour la génération du code Objet qui constitue la prochaine section.

5 Génération du Code Objet

Nous avons fait quelques modifications sur la base du code obtenu lors de la phase précédente « génération des quadruplets », plus exactement sur les classes suivantes :

- **TabQuadruple** : dans laquelle nous avons ajouté deux méthodes :

-La première permet de remplir une ArrayList « assembly »en bouclant sur tous les quadruplets de la table et de les traduire en langage assembleur avec la méthode« toAssembler »que nous avons développé dans la classe Quadruple.

```
public ArrayList<String> toAssembly() {
    ArrayList<String> assembly = new ArrayList<>();
    int i =0;
    for (Quadruple q:quads){
        assembly.addAll(q.toAssembler(i));
        i++;
    }
    return assembly;
}
```

-La deuxième permet d'écrire dans un fichier texte dont le chemin est donné en paramètre à partir de la ArrayList de code assembler obtenu grâce à la méthode juste au-dessus.

```
public void saveAssembly(String filename) throws IOException {
    Files.write(Paths.get(filename), toAssembly());
}
```

- **Quadruple** : dans celle-ci nous avons ajouté des méthodes pour transformer un quadruplet en lignes de code assembleur sous forme d'une chaine de caractère ; Tel que nous avons :

-La première « toAssembler » permet principalement de vérifier le type d'opération que représente le quadruplet courant, tel que nous avons catégorisé cinq type d'opération sur les quadruplets, qui sont :

1/ opération arithmétique (+, -, *, /)

2/ affectation (=)

3/ branchement (BR, BGE, BLE, BNE, BE)

4/ fin des quadruplets (Finale)

Ainsi selon le type trouvé on fait appel à une méthode parmi :

-La deuxième « op » qui correspond au type (opération arithmétique) nous commençons par ajouter l'instruction assembleur :

assembly .add(mov (AX , op1)) ;

Qui nous donnera comme résultat une chaine de caractère (String) de la forme : « MOV AX, a» si op1 est une variable « a »

Puis selon l'opération en question une seconde instruction sera inséré, qui elle aussi aura la forme : « ADD AX, b» si op2 est une variable b et que l'opération est une addition.

Et enfin l'instruction qui met le résultat finale dans le temporaire, qui est comme suit : « MOV AX, Temp »; avec Temp le temporaire.

//idém pour les autres instructions de soustraction, multiplication et division.

-La troisième : « aff »permet de représenter l'affectation en assembleur, en deux instructions seulement, qui sont :

« MOV AX, op1 » puis «MOV Temp, AX »

-La quatrième : «jump» quant à celle-ci elle vérifie d'abord s'il s'agit d'un jump conditionnelle ou inconditionnelle,

Dans le premier cas de figure nous insérant seulement une instruction assembleur : « JMP etiq2» ;

Avec etiq2 l'étiquette donnée comme deuxième opérande dans le quadruplet.

Dans le deuxième cas, quatre instructions assembleurs sont générées :

«MOV AX, a» ;

Avec « a » le premier opérande à comparer

«MOV BX, b» ;

Avec « b » la deuxième opérande à comparer

«CMP a, b»

«JLE etiq10 » ou « JGE etiq10 »;

Avec « etiq10 » l'étiquette à rejoindre en cas ou « a » est inférieur (resp supérieur) à « b »

- En plus de vérifier à chaque instruction s'il s'agit d'une instruction vers laquelle il y a un JUMP (branchement) afin de la précéder par une étiquette.

Mais pour cela nous avons ajouté dans la classe « RoutinesQuad » un Vector contenant les numéros des quadruplets étiquettes (vers lesquels il y a des branchements), et ce afin de les exploiter dans cette classe Quadruple tel que le numéro de quadruplet est passé en paramètre dans la première méthode cité « toAssembler ».

6 Classe Main

Dans une classe du même nom que notre projet « TEST »nous avons notre main ; Le main prends un tableau de string en paramètre d'entrée, ces paramètres peuvent être : -gui ,-tree , -tokens , . . . mais un seul est obligatoire ! il s'agit du fichier texte contenant le code dans le langage développé à tester.

Pour tester nous avons d'abord instancié :

- Un objet de la classe TestConfig lui donnant les argument du main en paramètre.
- Une ArrayList errors pour accueillir les éventuelles erreurs que l'on rencontrera.
- Un objet de la classe RoutinesTabSymbol qui génère la TS
- Un objet de la classe RoutinesQuad qui génère la table des quadruplets
- Une Liste ArrayList<TESTBaseListener> routines afin d'y regrouper les routines de la TS ainsi que celles des Quadruplets comme suit :

routines.add(routinesQuad) ;

routines.add(routinesTabSymbol) ;

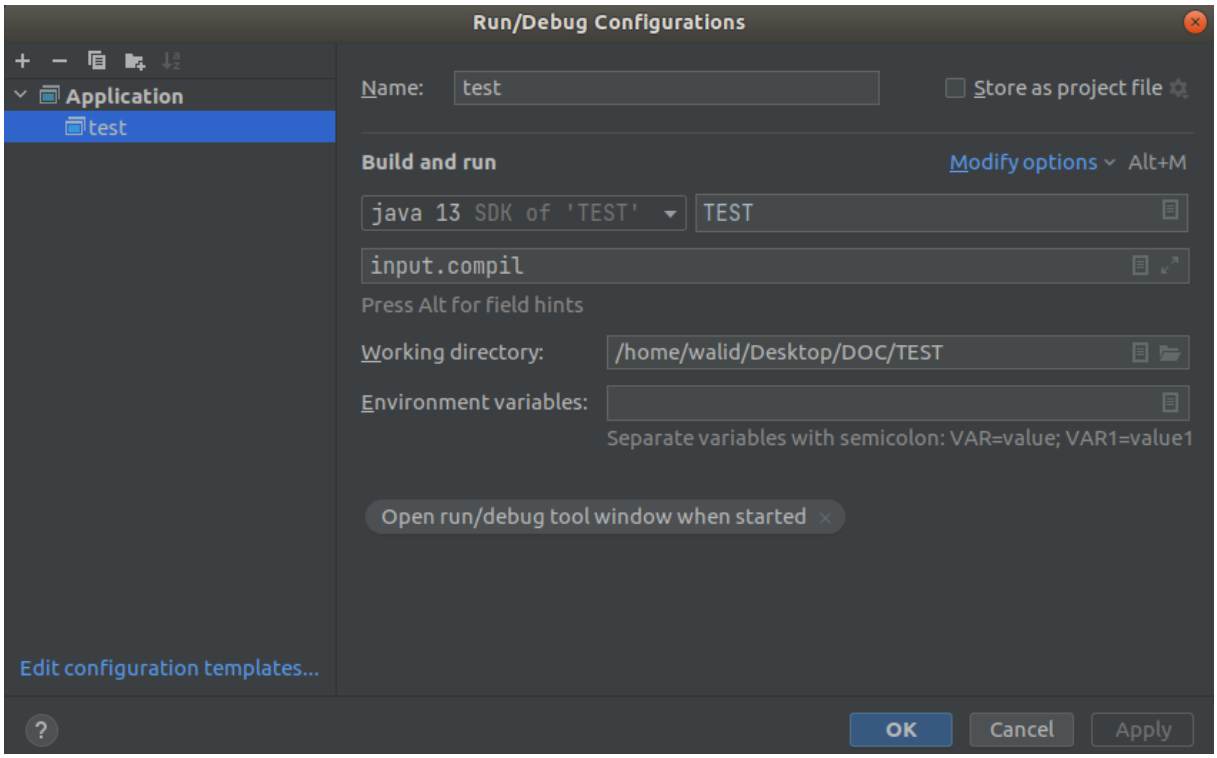
- Un objet ErrorListener errorListener pour y recueillir les erreurs rencontrées en pour les afficher par la suite ou même procéder a un traitement par la suite.
- Enfin on lance : ConstumTestRig.process(config, routines, errorListener) afin de générer le lexer et le parser
- Puis nous nous retrouvons devant deux possibilités :

1. Erreur rencontré : affichage de l'erreur rencontré.

2. Aucune erreur : affichage de la TS, affichage des quadruplets, génération du code objet.

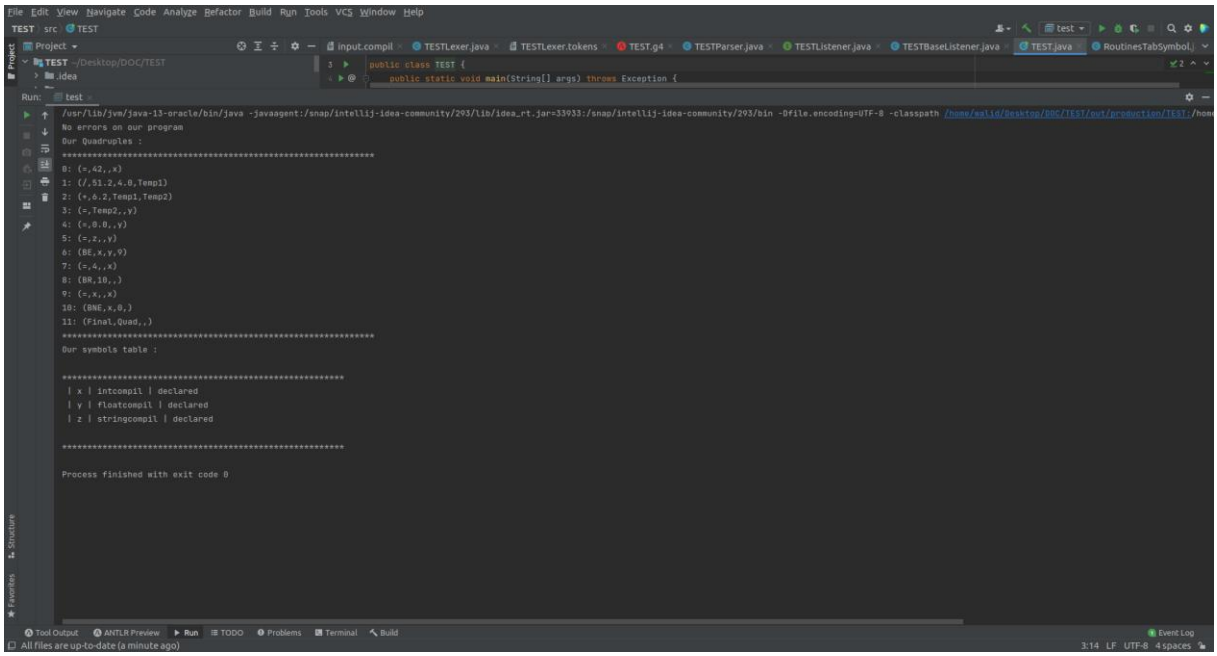
7 Compilation du projet

1. configurer la prise en charge du fichier contenant le code à tester : Run > Edit Configurations... > Tab ”Configuration” > Program arguments

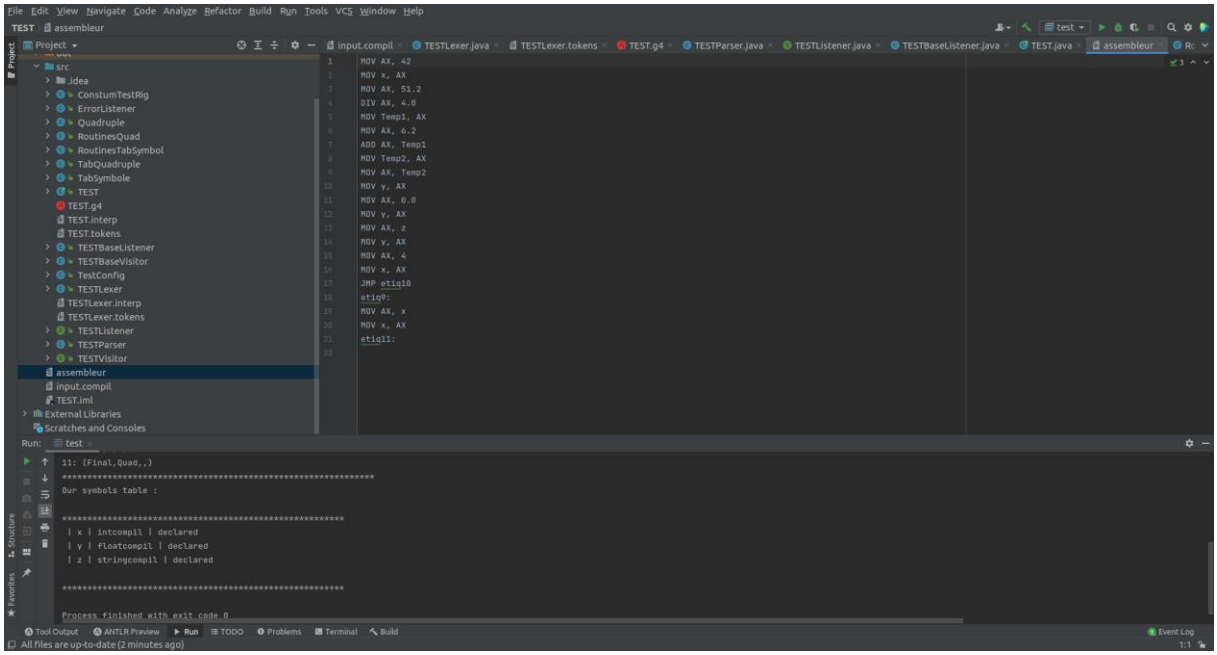


2. lancer l’exécution : Le lancement de l’exécution peut être réalisé à travers le bouton run de l’interface d’Intellij :

   2.1 Affichage de la table des symboles Ainsi que des quadruplets généré.



2.2 Génération du code objet dans un fichier ”Assembleur”



8 Section Optionnelle

Lors de la réalisation de ce projet, nous avons pu apporter quelques améliorations à notre code, nous citons :

1. Création de la classe CustomeTestRig :
La classe TestRig par defaut produit énormément d'efforts afin de chercher, trouver et utiliser des informations sur notre projet tel que : le nom du projet, le nom de notre fichier .g4 , notre parser ainsi que notre listner, ...
Afin de les instancier puis utiliser dans sa fonction process à laquelle nous faisons appel dans notre Main.
C'est pour cela que nous avons customisé notre classe TestRig lui donnant directement toutes ces informations nécessaire sur notre projet, et ainsi gagner en temps d'exécution.

2. Création de la classe ErrorListner : elle sert à récupérer les messages d'erreurs et de les afficher dans un format proche de celui des compilateur connu, en donnant le ligne et colonne ou se trouve l'erreur, mais aussi de sauvegarder ces erreurs en cas de besoin dans une utilisation ultérieure.