1. **Etapes d’Installation d’ANTLR**

/\* Quelles sont les étapes d’installation d’ANTLR sur votre machine et quelles sont les commandes utilisées, est-ce-que vous avez trouvé des problèmes? Quelle est la solution utilisée pour régler ce problème. \*/

Etapes d’installation d’ANTLR sous Linux (UBUNTU)

* commandes utilisées
* problème rencontré :

Une fois le terminal fermé toutes les configurations faites précédemment sont oublié, donc elle ne reste que le temps d’ouverture du terminale

* solution aux problèmes :

Rendre les configurations globales en exécutant les commandes suivantes

1. **Présentation des Différentes Phases de mon Compilateur**
   1. **Analyse Lexicale**

Lors de cette étape nous avons étudié le langage du mini compilateur de l’énoncé,

Puis défini les expressions régulières, tel que :

* Int, float, string, comment, progname, les identifiants …
  1. **Analyse Syntaxique**

Puis nous avons définit la structure générale d’un programme acceptable par notre .langage, qui est comme suit :

\begin{sql}

start\_rule : 'compil' PROGNAME '(' ')'

'{'

declarations

'start'

instructions

'}';

\end{sql}

Puis chaque nonTérminale est détaillé en respectant les règles LL(1) ainsi que les spécifications de l’énoncé.

Tel que toutes les instructions demandées (IF, SCANcompil, PRINTcompil, Affectation, les opérations sur les expressions) mais aussi d’autres instructions supplémentaires (FOR, WHILE, SWITCH…CASE)

* 1. **Analyse Sémantique**

/\* commentez votre table de symbole. Quelles sont les routines utilisées?. Si vous avez utilisé un IDE, quelle sont les étapes de configuration d’ANTLR et quelles sont les étapes de compilation de votre projet (imprime écrans). Comment vous avez créé votre classe MyListner ? \*/

* Table des symboles :

Notre table des symboles (dans la classe « TabSymbole ») est une ArrayList de ligne, tel que chaque ligne est composée de trois (3) informations qui sont :

* Name : l’identifiant de la variable
* Type : le type de la variable (qui peut être soit un « int » représenté par la valeur 1, soit un « float » représenté par la valeur 2)
* 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 TS :

getLigne : pour récupérer une ligne à partir du nom d’une variable.

containsLigne : pour tester si une variable existe dans la TS

addLigne : pour insérer une nouvelle ligne à 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.
* Tester la compatibilité des opérandes lors des affectations, calcule des expressions ou comparaisons.
* Empêcher l’utilisation de variables non déclarées.

Mais aussi géré l’insertion des variables dans la TS,

* Intellij IDE :
* Etapes de configuration d’ANTLR dans intellij
* Interface et organisation des codes
* Génération des parser et listner
* Tester les règles (arbre)
* Interface et organisation des codes
* Etapes de compilation du projet
* Création de la classe MyListner
  1. **Génération du Code Intermédiaire**

/\* comment vous avez généré vos quadruplets \*/

Pour généré nos quadruplets nous avons crée plusieurs classes, tel que :

**Quadruple**: représentant la structure d’un quadruplet, c'est-à-dire les quatre (4) 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

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 « parsser » notre échantillon de code respectant la syntaxe de notre langage, a 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.

* 1. **Génération du Code Objet**

/\*comment vous avez généré votre 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 la quelle nous avons ajouter deux (2) 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.

\begin{sql}

public ArrayList<String> toAssembly() {

ArrayList<String> assembly = new ArrayList<>();

int i =0;

for (Quadruple q:quads){

assembly.addAll(q.toAssembler(i));

i++;

}

return assembly;

}

\end{sql}

* 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 a la méthode juste au dessus.

\begin{sql}

public void saveAssembly(String filename) throws IOException {

Files.write(Paths.get(filename), toAssembly());

}

\end{sql}

* **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)

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 :

\begin{sql}

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

\end{sql}

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 Temps 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 a 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 (1) instruction assembleur :

«  JMP etiq2 » ; avec etiq2 l’étiquette donnée comme deuxième opérande dans le quadruplet.

Dans le deuxième cas, quatre (4) instructions assembleurs sont générées :

«MOV AX, a» ; avec « a » la première 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 a chaque instruction s’il s’agit d’une instruction vers la quelle 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 les quels 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».

* 1. **Classe Main**

/\* présenter votre classe Main \*/

Dans une classe du meme nom que notre projet « TinyLanguage » 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’agis du fichier texte contenant le code dans le langage développé a 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<TP2BaseListener> 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.
3. **Différence entre Flex/Bison et ANTLR**

/\* répondre à la question suivante, quelle est la différence entre Flex/Bison et ANTLR ?\*/

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

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

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 contenue 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.

Les analyseurs syntaxiques générés par ANTLR sont généralement à proximité de 10.000 lignes de code et plus.

Les parseurs de descente récursifs manuscrits sont souvent dans le même ordre de grandeur.

Le compilateur Wirth's Oberon est peut-être le plus compact avec environ 4000 lignes de code incluant la génération de code,

mais Oberon est un langage très compact avec seulement environ 40 règles de production.

Un grand avantage pour ANTLR est l'outil IDE graphique, appelé ANTLRworks.

C'est un laboratoire complet de grammaire et de langage. Il visualise vos règles de grammaire au fur et à mesure que vous les tapez et s'il trouve des conflits (s'il y en a),

il vous montrera graphiquement ce qu'est le conflit et ce qui le provoque.

Il peut même automatiquement refactoriser et résoudre des conflits tels que la récursivité à gauche. Une fois que vous avez une grammaire sans conflit,

vous pouvez laisser ANTLRworks analyser un fichier d'entrée de votre langage et construire une arborescence d'analyse

et AST pour vous et afficher graphiquement l'arbre dans l'EDI.

C'est un très gros avantage car cela peut vous faire économiser beaucoup d'heures de travail: vous trouverez des erreurs conceptuelles dans votre langage

avant de commencer à coder! Je n'ai pas trouvé un tel outil pour les grammaires LALR, il semble qu'il n'y en ait pas.

Peut produire des analyseurs syntaxiques dans différentes langues.

Java n'est pas requis pour exécuter l'analyseur généré.

1. **Section Optionnelle**

/\* si vous avez quelques points à ajouter \*/

Ajout d’un end if a la fin de l’instruction IF afin de délimiter la fin de la condition

CustomeTestRig :

ErrorListner : elle sert a récupérer les messages d’erreurs en cas de besoin dans une utilisation ultérieure