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

Partie 1 : Analyses lexicale et syntaxique

Analyse Lexicale

Analyse syntaxique

Partie 2 : Analyse sémantique

Table des symboles

structure table & classe ligne

structure valeur et évaluation des expressions (+Implémentation les méthodes)

Routines

Insertion dans la TS.

Empêcher les doubles déclarations.

Tester la compatibilité des types.

Empêcher l'utilisation des variables non déclarées.

Tester la compatibilité des types.

Empêcher l'utilisation des variables non déclarées.

Compatibilité des types d'expression .

Erreurs sémantiques

incompatibilité des types

division par 0

Double déclaration

idf non déclaré

spécifier le type de l'erreur

Instanciation d'une variable

Ajouter des Errors Listeners

Partie 3 : Code intermédiaire et transformation en code objet

Génération de la forme intermédiaire

Classe TableQuad

Classe Quadruplet

Classe RoutinesQuad.

Instructions arithmétique.

Instructions de branchement.

Branchement conditionnel (Instruction IF ELSE)

Instruction LIRE

Instruction ECRIRE

Génération de code objet

Quad to Assembler

ADD/SUB/DIV/MUL

Read

Write

Branchements conditionnels

Branchements inconditionnels

Affectation

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.

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 entier si tous ses opérandes sont entiers, réel sinon
* 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 :

A faire !!!!!!!!!!!!!!!!!!!!!!!!

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.

On déplace ce problème à 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. Empêcher les doubles déclarations

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

Dans le cas où l’un des côtés d’une expression à calculer est un identifiant auquel on n’a rien affecté encore.

ERR\_INSTANTIOATION =" variable non instatiated : ";

* 1. Incompatibilité des types

ERR\_INCOMPATIBILITY =" type mismatch : ";

* 1. Division par 0

ERR\_DIV\_Z =" numeric overflow : ";

* 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 compilation à chaque erreur.

Problème rencontré :

A

Solution trouvée :

B

1. La table des symboles
2. Le code intermédiaire

Problème rencontré :

A

Solution trouvée :

B

Problème rencontré :

A

Solution trouvée :

B