Compte rendu du TP de compilation

Réalisé par
Kamil Ben
et
Hissine Adam

1. Introduction

Le but du TP est de réaliser un analyseur syntaxique pour une grammaire d'un mini langage C. L'analyseur prend en entrée une chaine (programme) et devra retourner l'arbre syntaxique. L'arbre syntaxique qu'il faut retourner est juste la liste des règles de grammaires correspondantes à ce programme.

L'analyseur à construire est un automate à pile fonctionnant à la base d'une méthode descendante de type LL(1). Pour ce TP nous avons utiliser le langage python.

2. Vérification du caractère LL(1) de la grammaire

- Grammaire du langage

<programme>::= main() { <liste_declarations> <liste_instructions> }

<liste_declarations>::= <une_declaration> <liste_declarations> | vide

<une_declaration>::= <type> id

<liste_instructions>::=<une_instruction> <liste_instructions> | vide

<une_instruction>::= <affectation> | <test>

<type>::= int | float

<affectation>::= id=nombre;

<test>::= if <condition> <une_instruction> else <une_instruction>;

<condition>::= id <operateur> nombre

<operateur>::= < | > | =

- Calcul de l'ensemble premier :

Premier(<programme>) = { 'main(){' }

Premier(<liste_declarations>) = premier(<une_declaration>)
= premier(<type>) = { 'int', 'float' }

Premier(<liste_instructions> = premier(<une_instruction>)
= premier(<affectation>) U premier(<test>)
= {'id' , 'if' }

Premier(<une_condition>) = {'id'}
Premier(<operateur>) = { '<' , '>' , '=' }

- Calcul de l'ensemble suivant :

Suivant(<liste_declarations>) = { '}', '$' } U premier(<liste_instructions>)
Suivant(<liste_instruction>) = { '}', '$' }

3. Table d'analyse

Dans le programme python la table d'analyse est représentée par une matrice dans laquelle si une entrée vaut -1 cela veut dire qu'il n'y'a pas de règles.

	main(){	Id	Int	Float	Nombre	If	else	<	>	=	}	;	$
<programme>	0												
<liste_declarations>													
<liste_declarations>		2				2					2		2
<une_declaration>			3	3									
<liste_instructions>		4				4							
<liste_instructions>											5		5
<une_instruction>		6											
<une_instruction>													
<type>			8										
<type>				9									
<affectation>		10											
<test>							11						
<condition>		12											
<operateur>									13				
<operateur>										14			
<operateur>											15		

4. Structures de données utilisées et méthodes

1. Pour les terminaux et non terminaux

Nous avons utilisé des dictionnaires pour stocker l'ensemble des terminaux et des non terminaux.

2. Classe Regle

- partie_gauche : qui représente la règle
- partie_droite : une liste qui représente la production de la règle.

3. Classe Grammaire

- terminaux : dictionnaire des terminaux
- non_terminaux : dictionnaire des non terminaux
- regles : la liste de toutes les instances des règles

4. Classe Analyseur

Un analyseur pour qu'il puisse fonctionner correctement, il a besoin d'une grammaire et d'une table d'analyse.

Voici la liste des propriétés de classe:
- grammaire : instance d'une classe Grammaire contenant toutes les règles.
- table_analyse : la matrice qui représente la table d'analyse.
- pile : structure permettant d'analyser la chaine
- err_s : une Boolean qui permet a chaque instant de savoir s'il y'a une erreur.
- ch_ok : une Boolean qui indique si la chaine est acceptée.

5. Les méthodes

1. La méthode gen_grammaire

Cette méthode permet de générer une grammaire mais pas de façon automatique. Dans cette dernière les instances des règles de la grammaire seront écrites en dur et ajoutées dans une instance de grammaire (g) qui sera alors retournée par la fonction.

2. La méthode lire_chaine

Cette fonction permet de lire une chaine à partir d'un fichier text.
La chaine sera stocker dans un tableau data.
Nous avions rencontré un problème qui était que le caractère de fin de chaine était aussi stocké dans la liste data.
L'opération data[-1] a remédié à ce problème.

5. Lancement du programme

```
dark-folt@MacBook-Pro-de-Ben TP_compilation % ./analyseur.py
chaine à analyser: main(){ifidnombreid=nombre;elseid=nombre;;}
<Programme> ::= ['main()', '{', '<liste_declarations>', '<liste_instructions>', '}']
<liste_declarations> ::= ['vide']
<liste_instructions> ::= ['<une_instruction>', '<liste_instructions>']
<une_instruction> ::= ['<test>']
<test> ::= ['if', '<condition>', '<une_instruction>', 'else', '<une_instruction>', ';']
<condition> ::= ['id', '<operateur>', 'nombre']
Erreur 8
dark-folt@MacBook-Pro-de-Ben TP_compilation % ./analyseur.py
chaine à analyser: main(){ifid=nombreid=nombre;elseid=nombre;;}
<Programme> ::= ['main()', '{', '<liste_declarations>', '<liste_instructions>', '}']
<liste_declarations> ::= ['vide']
<liste_instructions> ::= ['<une_instruction>', '<liste_instructions>']
<une_instruction> ::= ['<test>']
<test> ::= ['if', '<condition>', '<une_instruction>', 'else', '<une_instruction>', ';']
<condition> ::= ['id', '<operateur>', 'nombre']
<operateur> ::= ['=']
<une_instruction> ::= ['<affectation>']
<affectation> ::= ['id', '=', 'nombre', ';']
<une_instruction> ::= ['<affectation>']
<affectation> ::= ['id', '=', 'nombre', ';']
<liste_instructions> ::= ['vide']
la chaine acceptée
```

Comma indiqué dans l'introduction, notre programme est écrit en python.
Pour le lancer il faut utiliser la commande: python3 analyseur.py ou ./analyseur.py
Le programme peut lire une chaine écrite dans le fichier chaine.txt

Exemple de chaine acceptée:
- main(){ if id=nombre id=nombre ; else id=nombre ; ; }

Exemple de chaine refusée:
- main(){ if id nombre id=nombre ; else id=nombre ; ; }

Conclusion:

Dans l'ensemble la difficulté de ce TP est normal, mais la partie qui nous a prit un peu plus de temps était celle du tableau d'analyse car on voulez être sûre que les valeurs qui sera dans la matrice allée être correcte.
Parmi les problèmes rencontrés il y'avait celui de la double production, par exemple si un non terminal admet plusieurs productions.
Pour le résoudre nous avions fait appel à une seconde liste qui quant à elle au moment du parcoure de la table d'analyse stocker toutes les productions d'un même non terminal.