RAPPORT DU MINI-PROJET : MINI-COMPILATEUR EN JAVA

1. Informations générales

Nom et Prénom : Dairi Lilia

Groupe : A3

Module : Compilation

Enseignante : Mme TASSOULT

Année universitaire : 2024 / 2025

Sujet : Réalisation d'un mini-compilateur en Java (analyse lexicale + syntaxique)

2. Introduction

Ce mini-projet a pour objectif de développer un mini-compilateur pour un langage simplifié inspiré de Python.

Il comprend deux parties essentielles :

1. Analyse lexicale : découper le programme source en unités lexicales (tokens).

2.Analyse syntaxique : vérifier que la structure du programme respecte les règles de la grammaire simplifiée.

Ce travail permet de comprendre le fonctionnement interne d'un compilateur et d'utiliser des concepts comme les automates, les grammaires, et la descente récursive.

3. Choix du langage et grammaire simplifiée

J'ai choisi de traiter un langage proche de Python, en supportant uniquement :

Instructions prises en charge :

Déclarations implicites de variables.

Affectation.

Expressions arithmétiques : + - * /

Comparaisons simples.

Incrémentation : +=, -=

Structure de contrôle principale (exigée) : foreach (for … in).

Appel print().

Listes : [1,2,3].

Instructions ignorées volontairement (comme demandé):

while

do/while

switch

boucles for classiques

autres structures Python (def, class, return…)

Cette simplification permet de respecter la consigne : un seul type de structure de controle doit etre analyse syntaxiquement.

3.1 Grammaire utilisée

Z → PROGRAM

PROGRAM → INSTRS

INSTRS → INSTR INSTRS | ε

INSTR → AFFECT | FOREACH | PRINT

AFFECT → IDENT '=' EXPR

| IDENT '+=' EXPR

| IDENT '-=' EXPR

FOREACH → 'for' IDENT 'in' IDENT ':' INSTRS

PRINT → 'print' '(' EXPR (',' EXPR)* ')'

EXPR → TERME RESTEXPR

RESTEXPR → ('+'|'-') TERME RESTEXPR | ε

TERME → FACT RESTTERME

RESTTERME → ('*'|'/') FACT RESTTERME | ε

FACT → IDENT | NOMBRE | '(' EXPR ')' | CHAINE | LIST

LIST → '[' (EXPR (',' EXPR)*)? ']'

4. Analyseur lexical

L’analyseur lexical repose sur :

Un automate déterministe (AFD) représenté par une matrice.

Chaque caractère est classé dans une colonne (lettre, chiffre, symbole, etc.).

Si la transition est -1, une erreur lexicale est signalée.

Les mots-clés sont reconnus :

for, in, print, if, else, elif, while …

Exemple de tokens générés :

IDENTIFIANT : nombres

ASSIGN : =

SYMBOLE : [

NOMBRE : 1

SYMBOLE : ,

...

PRINT : print

SYMBOLE : (

SYMBOLE : "

IDENTIFIANT : message

5. Analyseur syntaxique

L’analyseur syntaxique utilise :

Une descente récursive pour parser la grammaire

Un pointeur pos pour parcourir la liste de tokens

Un système d’erreurs claires :

Erreur syntaxique

Token attendu mais absent

Parenthèse ou guillemet non fermé

Mauvaise structure de for

Exemple d’erreur affichée:

Erreur: ')' attendu dans print

Code refusé !

Exemple de réussite

Code accepte!

6. Jeux de tests

6.1 Test valide (liste + foreach + print)

```
nombres = [1,2,3,4]
somme = 0
for n in nombres:
    somme += n
print("somme:", somme)
```

→ Code accepté

6.2 Affectation complexe

```
a = 5
b = 10
c = a + b - 3
print("c:", c)
```

→ Code accepté

6.3 Erreur volontaire : guillemet manquant

```
print("hello)
```

→ Erreur détectée

6.4 while doit être ignoré mais pas accepté

```
while x < 5:
    x += 1
```

→ Ignoré — Non analysé syntaxiquement

→ Si contaminant la structure : Code refusé (normal)

7. Structure du projet

```
analyselexprojet/
  ├── Analyselexprojet.java      (analyseur lexical + main)
  ├── Analysesyntaxique.java     (analyseur syntaxique)
  ├── Token.java                 (structure des lexèmes)
  └── rapport.docx               (rapport final)
```

8. Gestion des erreurs

Le mini-compilateur :

Continue même après une erreur (ne s'arrête pas brutalement)

Affiche des messages explicites :

Erreur lexicale

Erreur syntaxique

Parenthèse manquante

Deux-points manquant

Mauvaise structure foreach

9. Conclusion

Ce mini-projet m’a permis de comprendre :

La construction d’un analyseur lexical (AFD, classification des caractères)

La création d’un analyseur syntaxique en descente récursive

La gestion d’erreurs dans un mini-compilateur

L’importance d’une grammaire claire et simple

Il constitue une base solide pour un compilateur plus complet (ajout des fonctions, classes, types, etc.).

10. Références

Cours de compilation

Documentation Java

Théorie des automates (AFD / AFD)

Grammaires LL(1)