République Algérienne Démocratique et Populaire

Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

Université de Béjaïa

Faculté des Sciences Exactes – Département d'Informatique

Projet COMPILATION

(Mini compilateur)

Sujet :

Compilateur mini en java pour langage python (if/else)

Réalisé par : SALIM MANCER

Table des matières

Chapitre 1

Introduction

L'objectif de ce projet est de concevoir un compilateur simple pour le langage Python en Java, comprenant un analyseur lexical (**lexer**) et un analyseur syntaxique (**parser**). Le projet illustre les concepts fondamentaux de la compilation, notamment la reconnaissance lexicale et l'analyse syntaxique basée sur une grammaire.

Chapitre 2

Présentation du Lexer

2.1 Méthode utilisée

Le lexer a été implémenté **à la main**, sans utiliser de générateur de lexer comme JFlex. La méthode consiste à parcourir le code source caractère par caractère et à construire des **tokens** en fonction du type rencontré : mots-clés, identifiants, nombres, opérateurs, chaînes de caractères, etc.

2.2 Fonctionnement général

- Ignorer les espaces et tabulations grâce à la méthode `ignorerespaces()`.
- Identifier les mots-clés et identifiants avec `avancer_caractere()` et `chiffre_nombre()`.
- Détecter les nombres avec `avancernember()`.
- Reconnaître les opérateurs (`=`, `==`, `+`, `+=`, `-`, `-=`, `*`, `*=`, `/`, `/=`, `%`) via `avanceroperateur()`.
- Gérer les chaînes de caractères et les retours à la ligne pour l'indentation (tokens `INDENT`, `DEDENT`, `NEWLINE`).

2.3 Exemple de tokenisation

Le lexer transforme le code source Python suivant :
[language=Python] if a == 10 : print("Bonjour") else : a = a + 1
en une liste de tokens :
IF, IDENTIFIER(a), EQEQ, NUMBER(10), COLON, NEWLINE, INDENT, PRINT, LPAREN, CHAINE("Bonjour"), RPAREN, NEWLINE, DEDENT, ELSE, COLON, NEWLINE, INDENT, IDENTIFIER(a), EQUALS, IDENTIFIER(a), PLUS, NUMBER(1), NEWLINE, DEDENT, EOF

Chapitre 3

Présentation du Parser

3.1 Méthode utilisée

Le parser a été conçu selon une approche **récursive descendante**, ce qui signifie que chaque règle de la grammaire est traduite en méthode Java récursive.

3.2 Grammaire utilisée

La grammaire factorisée et sans récursion à gauche est la suivante :
— PROGRAM → STATEMENT_LIST EOF
— STATEMENT_LIST → STATEMENT STATEMENT_LIST1
— STATEMENT_LIST1 → STATEMENT STATEMENT_LIST1 $|\epsilon$
— STATEMENT → SIMPLE_STMT | COMPOUND_STMT
— SIMPLE_STMT → ASSIGN | PRINT_CALL
— COMPOUND_STMT → IF_STMT
— IF_STMT → IF EXPR COLON BLOCK IF_TAIL
— IF_TAIL → ELIF EXPR COLON BLOCK IF_TAIL | ELSE COLON BLOCK $|\epsilon$
— BLOCK → NEWLINE INDENT STATEMENT_LIST DEDENT
— EXPR → OR_EXPR
— OR_EXPR → AND_EXPR OR_EXPR1
— OR_EXPR1 → OR AND_EXPR OR_EXPR1 $|\epsilon$
— AND_EXPR → NOT_EXPR AND_EXPR1
— AND_EXPR1 → AND NOT_EXPR AND_EXPR1 $|\epsilon$
— NOT_EXPR → NOT NOT_EXPR | COMP_EXPR
— COMP_EXPR → FACTOR (COMP_OP FACTOR)*
— FACTOR → NUMBER | TRUE | FALSE | IDENTIFIER | CHAINE | LPAREN EXPR RPAREN

3.3 Fonctionnement général

Le parser lit la liste de tokens générée par le lexer et :
— Vérifie que les instructions simples (`ASSIGN`, `PRINT`) sont syntaxiquement correctes.
— Gère les blocs d'instructions indentés avec `INDENT` et `DEDENT`.
— Analyse les expressions logiques et arithmétiques récursivement.
— Vérifie la cohérence des structures conditionnelles (`if`, `elif`, `else`).

Chapitre 4

Conclusion et Perspectives

Le projet a permis de mettre en pratique les concepts fondamentaux de la compilation :
— Conception d'un lexer à la main pour Python.
— Implémentation d'un parser récursif descendant.
— Gestion des indentations et des structures conditionnelles.
Des perspectives d'amélioration comprennent :
— Support complet des boucles (`for`, `while`) et des exceptions (`try/except`).
— Génération d'un arbre syntaxique abstrait (AST) pour des optimisations ultérieures.
— Extension pour d'autres structures Python (fonctions, classes, imports...).