République Algérienne Démocratique et Populaire

Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

Université Abderrahmane Mira – Béjaïa

Faculté des Sciences Exactes

Département d'Informatique

Projet Compilation

Sujet : Mini compilateur en JAVA sur 'match-case' de python

Réalisé par :

-Nassim Manadi

Année universitaire : 2025–2026

Table des matières

Chapitre 1

Introduction

Ce projet a pour objectif de concevoir et implémenter en Java un analyseur lexical et syntaxique pour un sous-ensemble du langage Python. L'originalité réside dans la prise en charge de la nouvelle construction `match/case` introduite en Python 3.10.

Le projet est structuré en quatre classes principales :

— **TOKEN** : définit les types de tokens.

— **Analyseur_lexical** : implémente le lexer.

— **Analyseur_syntaxique** : implémente le parser.

— **Main** : exécute le code source par un chemin vers un ficher

Cette organisation permet de séparer clairement les responsabilités et de mettre en évidence le rôle de chaque étape du processus de compilation, depuis la reconnaissance des unités lexicales jusqu'à la validation syntaxique.

Chapitre 2

Analyseur_lexical

2.1 Méthode utilisée

Le lexer a été implémenté à la main, sans recourir à un générateur automatique. La méthode consiste à parcourir le code source caractère par caractère et à construire des tokens spécifiques à la structure `match/case`. Les principaux éléments reconnus sont : le mot-clé `match`, le mot-clé `case`, les identifiants, les constantes, ainsi que les séparateurs (deux-points, parenthèses, etc.).

2.2 Fonctionnement général

— Ignorer les espaces et tabulations grâce à une fonction `ignorerEspaces()`.

— Identifier les mots-clés `match` et `case`.

— Reconnaître les identifiants et constantes numériques ou chaînes de caractères.

— Détecter les séparateurs et opérateurs nécessaires (`:`, `,`, `=`, etc.).

— Gérer les retours à la ligne et l'indentation (tokens `INDENT`, `DEDENT`, `NEWLINE`).

Remarque : Dans cette implémentation, les tokens `INDENT` et `DEDENT` correspondent à un décalage fixe de quatre espace, utilisé pour signaler respectivement l'entrée et la sortie d'un bloc de code.

2.3 Exemple de tokenisation

Le lexer transforme le code source Python suivant :

```
x = 2

match x:
    case 1:
        print("Un")
    case 2:
        print("Deux")
    case _:
        print("Autre")
```

en une liste de tokens :

```
IDENTIFICATEUR(x), EGAL(=), NOMBRE(2), MATCH,
IDENTIFICATEUR(x), DEUX_POINT, NEWLINE, INDENT,
CASE, NOMBRE(1), DEUX_POINT, NEWLINE, INDENT, PRINT,
PARENT_OUV, CHAINE("Un"), PARENT_FER, NEWLINE, DEDENT,
CASE, NOMBRE(2), DEUX_POINT, NEWLINE, INDENT, PRINT,
PARENT_OUV, CHAINE("Deux"), PARENT_FER, NEWLINE, DEDENT,
CASE, UNDERSCORE, DEUX_POINT, NEWLINE, INDENT, PRINT,
PARENT_OUV, CHAINE("Autre"), PARENT_FER, NEWLINE, DEDENT,
EOF
```

Chapitre 3

Analyseur_syntaxique

3.1 Méthode utilisée

Le parser a été conçu selon une approche récursive descendante. Chaque règle de la grammaire est traduite en une méthode Java, ce qui permet de vérifier la conformité des tokens générés par le lexer avec la syntaxe attendue pour la structure `match/case`. Cette approche facilite la lecture et la maintenance du code, tout en restant proche des principes théoriques de la compilation.

3.2 Grammaire utilisée

La grammaire factorisée et adaptée est la suivante :

```
PROGRAM → STATEMENT_LIST EOF

STATEMENT_LIST → STATEMENT STATEMENT_LIST1
STATEMENT_LIST1 → STATEMENT STATEMENT_LIST1 | (epsilon)

STATEMENT  -> ASSIGN_STMT | PRINT_STMT | IF_STMT | MATCH_STMT

ASSIGN_STMT → IDENTIFICATEUR EGAL EXPR
PRINT_STMT → PRINT PARENT_OUV ARG_LIST PARENT_FER
ARG_LIST → EXPR ARG_LIST1
ARG_LIST1 → VERGULE EXPR ARG_LIST1 | (epsilon)


IF_STMT → IF EXPR DEUX_POINT BLOCK IF_TAIL
IF_TAIL → ELIF EXPR DEUX_POINT BLOCK IF_TAIL | ELSE DEUX_POINT BLOCK | (epsilon)
```

```
MATCH_STMT → MATCH EXPR DEUX_POINT NEWLINE INDENT CASE_LIST DEDENT
CASE_LIST → CASE_BLOCK CASE_LIST1
CASE_LIST1 → CASE_BLOCK CASE_LIST1 | (epsilon)
CASE_BLOCK → CASE PATTERN DEUX_POINT NEWLINE INDENT STATEMENT_LIST DEDENT
PATTERN → EXPR | UNDERSCORE

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 → ADD_EXPR COMP_EXPR1
COMP_EXPR1 → COMP_OP ADD_EXPR COMP_EXPR1 | (epsilon)
COMP_OP → DOUBLE_EGAL | NO_EGAL | INF | INF_OU_EGAL | SUP | SUP_OU_EGAL
ADD_EXPR → TERM ADD_EXPR1
ADD_EXPR1 → PLUS TERM ADD_EXPR1 | MOIN TERM ADD_EXPR1 | (epsilon)
TERM → FACTOR TERM1
TERM1 → MULT FACTOR TERM1 | DIV FACTOR TERM1 | MOD FACTOR TERM1 | (epsilon)
FACTOR→ NOMBRE | TRUE | FALSE | IDENTIFICATEUR | CHAINE | PARENT_OUV EXPR PARENT_FER
```

3.3 Fonctionnement général

Le parser lit la liste de tokens générée par le lexer et :

— Vérifie la cohérence des instructions simples (`assignations`, `print`).

— Reconnaît et traite la construction `match/case`, en validant chaque motif (`pattern`) et son bloc associé.

— Gère correctement l'indentation et la déduction des blocs (`INDENT`, `DEDENT`).

— Vérifie la validité des expressions arithmétiques et logiques selon la grammaire définie.

3.4 Exemple d'exécution

Le fichier source est lu depuis :

`/Users/manadinassim/Desktop/exemple_code.txt`

3.4.1 Code source

```
x = 2
match x:
    case 1:
        print("Un")
    case 2:
        print("Deux")
    case 3:
        print("Trois")
    case _:
        print("Autre nombre")
```

3.4.2 Résultat

```
Programme accepté
```

Chapitre 4

Conclusion

La réalisation de ce mini-compilateur en Java pour la construction match/case de Python 3.10 a permis de mettre en pratique les principes essentiels de l'analyse lexicale et syntaxique. Ce projet illustre la manière dont une grammaire peut être traduite en un parseur fonctionnel, tout en offrant une compréhension plus profonde des mécanismes internes des langages modernes. Au-delà de l'aspect technique, il constitue une expérience formatrice qui renforce les compétences en conception de compilateurs et en programmation système.