Rapport de mini Projet compilation

Analyseur Lexical pour le langage **C**
et Syntaxique pour l'instruction **Switch - - Case**

Réalisé par :LAHLOU Katia

Année universitaire : 2024-2025

7 décembre 2025

1 Introduction

1.1 Contexte et objectif du projet

Ce projet s'inscrit dans le cadre du cours de compilation et vise à développer un analyseur lexical et syntaxique pour un sous-ensemble du langage C. L'objectif principal est de comprendre et d'implémenter les deux premières phases d'un compilateur : l'analyse lexicale (ou tokenisation) et l'analyse syntaxique.

1.2 Fonctionnalités principales

Le système développé offre les fonctionnalités suivantes :

— Analyse lexicale complète avec reconnaissance de tous les tokens du langage C

— Analyse syntaxique spécialisée pour les structures `switch-case`

— Gestion robuste des erreurs lexicales et syntaxiques

— Support des switch imbriqués et des structures de contrôle complexes

— Rapport détaillé des erreurs avec localisation précise (ligne et colonne)

1.3 Technologies utilisées

— **Langage** : Java

— **Paradigme** : Programmation orientée objet

— **Techniques** : Automates à états finis,algorithmes à la main, Descente récursive

2 Architecture du Projet

2.1 Structure générale

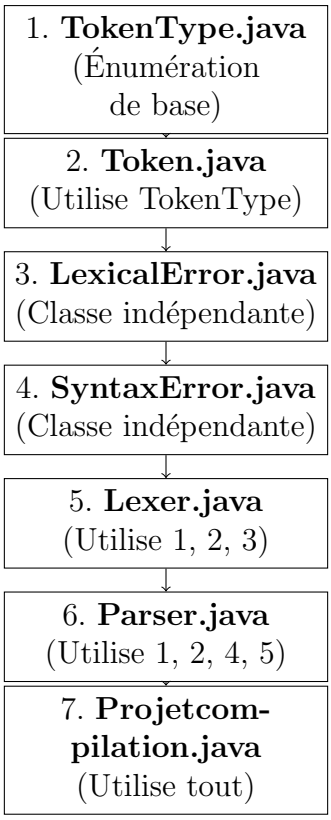
Le projet est organisé en sept classes Java interconnectées, chacune ayant une responsabilité spécifique :

```
projetcompilation/
|
|-- TokenType.java          -> Enumeration des types de tokens
|-- Token.java              -> Representation d'un token
|-- LexicalError.java       -> Gestion des erreurs lexicales
|-- SyntaxError.java        -> Gestion des erreurs syntaxiques
|-- Lexer.java              -> Analyseur lexical
|-- Parser.java             -> Analyseur syntaxique
`-- Projetcompilation.java  -> Point d'entree principal (main)
```

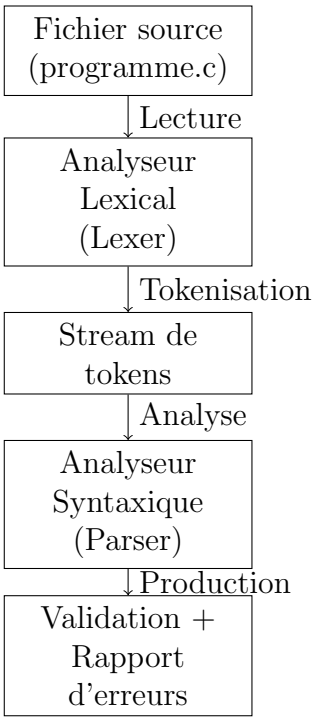
Listing 1 – Structure du package

2.2 Hiérarchie des dépendances

L'ordre de compilation respecte la hiérarchie suivante :



2.3 Flux de traitement



2.4 Principe de séparation des préoccupations

— **Lexer** : Transforme le texte brut en séquence de tokens

— **Parser** : Vérifie la structure syntaxique selon la grammaire définie

— **Classes d'erreur** : Encapsulent et formatent les erreurs détectées

3 Analyse Lexicale

3.1 Principe général

L'analyseur lexical (Lexer) lit le fichier source caractère par caractère et produit une séquence de tokens. Il utilise la technique de **sentinelle EOF** (`\0`) pour éviter les vérifications répétitives de la longueur de la chaîne d'entrée.

3.2 Automate pour les identificateurs

3.2.1 Description formelle

L'automate reconnaît les identificateurs C (variables, fonctions, mots-clés) :

— **Alphabet** : $\Sigma = \{a\text{-}z, A\text{-}Z, 0\text{-}9, \_\}$

— **États** : $Q = \{q_0, q_1\}$

— **État initial** : q_0

— **États finaux** : $F = \{q_1\}$

— **Règle** : Commence par une lettre ou _, suivi de lettres, chiffres ou _

3.2.2 Matrice de transition

État	Lettre	Chiffre	Underscore	Autre
q_0 (initial)	q_1	$\emptyset$	q_1	$\emptyset$
q_1 (final)	q_1	q_1	q_1	$\emptyset$

TABLE 1 – Matrice de transition pour les identificateurs

3.2.3 Diagramme d'états

$$q_0 \xrightarrow{\text{L, \_}} q_1 \circlearrowright \text{L, C, \_}$$

Légende : L = Lettre, C = Chiffre, _ = Underscore

3.2.4 Exemples

✓ Acceptés : `variable`, `_counter`, `temp123`, `myVar_2`

× Rejetés : `123abc` (commence par un chiffre), `my-var` (caractère invalide)

3.3 Automate pour les nombres

3.3.1 Description formelle

Reconnaît les entiers et les nombres flottants :

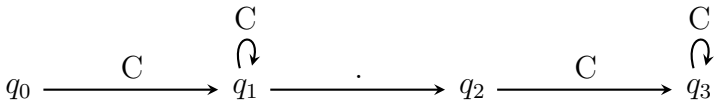
— **Alphabet** : $\Sigma = \{0\text{-}9, .\}$
— **États** : $Q = \{q_0, q_1, q_2, q_3\}$
— **État initial** : q_0
— **États finaux** : $F = \{q_1, q_3\}$ (q_1 pour entier, q_3 pour flottant)

3.3.2 Matrice de transition

État	Chiffre	Point	Autre
q_0 (initial)	q_1	$\emptyset$	$\emptyset$
q_1 (entier final)	q_1	q_2	$\emptyset$
q_2 (après point)	q_3	$\emptyset$	$\emptyset$
q_3 (flottant final)	q_3	$\emptyset$	$\emptyset$

TABLE 2 – Matrice de transition pour les nombres

3.3.3 Diagramme d'états



Légende : C = Chiffre

3.3.4 Exemples

✓ Entiers : `123`, `0`, `999`
✓ Flottants : `3.14`, `0.5`, `123.456`
× Invalides : `.123` (commence par un point), `12.` (se termine par un point)

3.4 Gestion des chaînes et caractères

3.4.1 Chaînes de caractères (`"..."`)

— Support des séquences d'échappement : `\n`, `\t`, `\\`, `\"`, `\0`
— Détection des chaînes non fermées (erreur si EOF avant `"`)

3.4.2 Caractères (`'...'`)

— Un seul caractère entre apostrophes
— Support des séquences d'échappement
— Détection des caractères non fermés

3.5 Gestion des commentaires

— **Commentaires de ligne** : `//` jusqu'à la fin de ligne
— **Commentaires de bloc** : `/* ... */` avec support du multiligne

3.6 Reconnaissance des opérateurs

Le lexer reconnaît tous les opérateurs C, incluant :
— Opérateurs composés : `++`, `--`, `+=`, `-=`, `*=`, etc.
— Opérateurs de décalage : `<<`, `>>`, `<<=`, `>>=`
— Opérateurs logiques : `&&`, `||`, `!`
— Opérateurs de comparaison : `==`, `!=`, `<=`, `>=`

4 Analyse Syntaxique

4.1 Grammaire formelle

Le parser implémente une grammaire LL(1) pour les instructions `switch-case` :

$$
\begin{aligned}
\text{SwitchStmt} &\to \texttt{switch (} \text{Expr} \texttt{) \{ } \text{CaseList} \texttt{ \}} \\
\text{CaseList} &\to \text{Case CaseList} \mid \varepsilon \\
\text{Case} &\to \texttt{case } \text{IntLiteral} \texttt{ :} \text{ StmtList Break} \\
&\quad \mid \texttt{default :} \text{ StmtList Break} \\
\text{Break} &\to \texttt{break ;} \mid \varepsilon \\
\text{StmtList} &\to \text{Stmt StmtList} \mid \varepsilon \\
\text{Stmt} &\to \text{SwitchStmt} \mid \text{IfStmt} \mid \text{WhileStmt} \\
&\quad \mid \text{ForStmt} \mid \text{ReturnStmt} \mid \text{AssignStmt} \mid \text{Block} \\
\text{Block} &\to \{ \text{ StmtList } \} \\
\text{Expr} &\to \text{(expression simplifiée)}
\end{aligned}
$$

4.2 Ensembles FIRST et FOLLOW

4.2.1 Ensembles FIRST

$$
\begin{aligned}
\text{FIRST(SwitchStmt)} &= \{\texttt{switch}\} \\
\text{FIRST(CaseList)} &= \{\texttt{case}, \texttt{default}, \varepsilon\} \\
\text{FIRST(Case)} &= \{\texttt{case}, \texttt{default}\} \\
\text{FIRST(Break)} &= \{\texttt{break}, \varepsilon\} \\
\text{FIRST(StmtList)} &= \{\texttt{switch}, \texttt{if}, \texttt{while}, \texttt{for}, \\
&\qquad \texttt{return}, \{, \text{IDENTIFIER}, \varepsilon\}
\end{aligned}
$$

4.2.2 Ensembles FOLLOW

$$\begin{aligned} \text{FOLLOW(CaseList)} &= \{\} \\ \text{FOLLOW(Case)} &= \{\texttt{case}, \texttt{default}, \}\} \\ \text{FOLLOW(StmtList)} &= \{\texttt{case}, \texttt{default}, \}, \texttt{break}\} \end{aligned}$$

4.3 Méthode de parsing : Descente récursive

Le parser utilise la **descente récursive**, où chaque règle de grammaire correspond à une méthode Java :

Règle de grammaire	Méthode Java
SwitchStmt	`switchStatement()`
CaseList	`caseList()`
Case	`caseClause()`
Break	`breakStatement()`
StmtList	`statementList()`

TABLE 3 – Correspondance grammaire-méthodes

4.4 Récupération d'erreurs

Le parser implémente une stratégie de **récupération par synchronisation** :

1. **Détection d'erreur** : Lorsqu'un token inattendu est rencontré
2. **Synchronisation** : Avancer jusqu'à un "point de reprise" sûr
3. **Continuation** : Reprendre l'analyse normalement

4.4.1 Points de synchronisation

— Pour les erreurs dans `switch` : chercher {

— Pour les erreurs dans `case` : chercher `case`, `default`, ou }

— Pour les erreurs dans les instructions : chercher `;`

4.4.2 Exemple de récupération

```
switch (x) {
    case 1  // Erreur : ':' manquant
        printf("un");
    case 2:  // Parser recupere ici
        printf("deux");
}
```

Listing 2 – Exemple de récupération d'erreur

4.5 Support des structures imbriquées

Le parser gère correctement :

— **Switch imbriqués** : Un `case` peut contenir un autre `switch`

— **Blocs d'instructions** : { ... } avec récursion

— **Structures de contrôle** : `if`, `while`, `for` dans les `case`

5 Gestion des Erreurs

5.1 Erreurs lexicales

5.1.1 Types d'erreurs détectées

— Identificateurs invalides

— Nombres mal formés (ex : `12.`)

— Chaînes non fermées

— Caractères non fermés

— Caractères non reconnus

5.1.2 Format de rapport

```
ERREUR LEXICALE [Ligne 5, Colonne 12] : Chaine de caracteres non fermee
  Contexte : '"hello world'
```

Listing 3 – Exemple de rapport d'erreur lexicale

5.2 Erreurs syntaxiques

5.2.1 Types d'erreurs détectées

— Parenthèses manquantes : `switch x) { ... }`

— Accolades manquantes : `switch (x) case 1: ...`

— Deux-points manquants : `case 1 printf("un");`

— Point-virgule manquant après `break`

— Valeur non entière dans `case`

5.2.2 Format de rapport

```
ERREUR SYNTAXIQUE a la ligne 8, colonne 15 :
   Message : Attendu ':' apres la valeur du case
   Contexte : '1'
```

Listing 4 – Exemple de rapport d'erreur syntaxique

6 Résultats et Performance

6.1 Capacités du lexer

— Reconnaît tous les tokens du langage C (mots-clés, opérateurs, littéraux)
— Gère correctement les commentaires et les espaces
— Détecte et rapporte toutes les erreurs lexicales avec localisation précise

6.2 Capacités du parser

— Analyse complète des structures `switch-case`
— Support des switch imbriqués sans limite de profondeur
— Récupération d'erreur robuste permettant de continuer l'analyse
— Rapports d'erreur détaillés avec position exacte

6.3 Exemple de sortie

```
========== ANALYSE SYNTAXIQUE (Switch-Case uniquement) ==========

================================================
Switch #1 detecte a la ligne 5
================================================
-> Debut de l'instruction switch
  -> Case clause detectee (ligne 6)
    Valeur: 1
    -> Break detecte
  -> Default clause detectee (ligne 9)
    -> Break detecte
-> Fin de l'instruction switch

================================================
RESUME DE L'ANALYSE SYNTAXIQUE
================================================
Nombre de switch analyses : 1
Tous les switch sont syntaxiquement corrects !
```

Listing 5 – Sortie du programme

7 Conclusion

Ce projet a permis de développer un analyseur lexical et syntaxique fonctionnel pour un sous-ensemble du langage C. Les automates à états finis utilisés pour la reconnaissance des tokens garantissent une analyse rapide et fiable. La grammaire LL(1) implémentée pour les structures `switch-case` permet une analyse syntaxique claire avec une bonne gestion des erreurs.