Analyseur Lexical et Syntaxique

Réalisé par: Yasmine Yahia

Date : 7 decembre 2025

Groupe: B4

Sous la direction de : MadameTASSOULT

Année académique: 2025/2026

Sommaire:

1. Introduction

La compilation est un processus fondamental en informatique, permettant de transformer du code source écrit par un développeur en instructions compréhensibles par une machine. L'objectif de ce mini-projet est de concevoir un mini-compilateur en Java, capable de lire, analyser et vérifier des programmes écrits dans un langage simplifié inspiré de Java. Ce projet pédagogique permet de comprendre les étapes clés d'un compilateur, à savoir l'analyse lexicale et l'analyse syntaxique, ainsi que la gestion des erreurs.

L'analyse lexicale consiste à découper le programme en unités élémentaires appelées tokens, représentant des identifiants, des nombres, des opérateurs, des séparateurs et des mots-clés. Dans ce projet, nous avons ajouté un mot-clé personnalisé yasmineyahia pour illustrer l'extensibilité de notre langage. L'analyse syntaxique, quant à elle, vérifie que ces tokens respectent les règles de grammaire définies, détectant ainsi les erreurs comme des parenthèses non fermées, des opérateurs isolés, des points-virgules manquants ou des expressions incomplètes.

La gestion des erreurs est un aspect essentiel du mini-compilateur. En utilisant des blocs try-catch, nous garantissons que le programme ne s'arrête pas à la première erreur rencontrée, mais continue à analyser l'ensemble du code pour détecter toutes les anomalies présentes. Cette approche permet à l'utilisateur d'obtenir un retour complet sur la validité de son programme.

Enfin, ce mini-compilateur est interactif : l'utilisateur peut choisir indépendamment de tester l'analyse lexicale ou l'analyse syntaxique, ligne par ligne. Le projet combine l'utilisation d'automates finis déterministes (AFD) pour l'identification des lexèmes, un Lexer et un Parser robustes, ainsi qu'une organisation modulaire en Java, permettant de comprendre de manière concrète le fonctionnement interne d'un compilateur et les mécanismes de détection et de traitement des erreurs dans un langage simplifié.

1. Chapitre 1 : Objectif du mini-projet:

1.1 L'objectif du projet:

Le mini-projet de compilation a pour but de nous initier aux étapes fondamentales de construction d'un compilateur.

Bien que les compilateurs professionnels soient très complexes, ce projet se concentre uniquement sur l'analyse lexicale et l'analyse syntaxique, ce qui permet de comprendre progressivement le fonctionnement interne d'un compilateur.

Les objectifs principaux de ce mini-projet sont les suivants :

✓ **Objectif 1 :** Comprendre le rôle de l'analyse lexicale

L'analyse lexicale est la première étape de compilation. Elle consiste à parcourir le code source caractère par caractère pour détecter les lexèmes, puis les transformer en tokens.

Cet objectif nous apprend :

comment identifier les mots-clés, identifiants, opérateurs, séparateurs, etc. ;

comment structurer ces informations dans une table des symboles ;

comment gérer les erreurs lexicales de manière propre.

✓ **Objectif 2 :** Apprendre à analyser la structure d'une expression avec un Parser

Une fois les tokens générés, l'analyse syntaxique permet de vérifier si la suite de tokens respecte une petite grammaire.

L'objectif est donc de :

comprendre comment un compilateur vérifie la structure d'une expression,

détecter les erreurs syntaxiques (parenthèses, opérateurs isolés, point-virgule, etc.),

appliquer les règles d'une grammaire simple pour valider une instruction.

✓ **Objectif 3** : Manipuler des structures essentielles d'un compilateur

Le projet nous permet de manipuler plusieurs structures importantes :

liste de tokens produite par le Lexer,
table des symboles contenant les mots-clés et opérateurs reconnus,
pile (stack) utilisée pour vérifier les parenthèses,
gestion des erreurs via try-catch.
Cela donne une vision concrète de la façon dont les compilateurs organisent les données durant le traitement du code.

✓ **Objectif 4 :** Construire un mini-compilateur fonctionnel et interactif en Java
Le projet vise aussi à intégrer les différentes étapes dans une application complète, avec :
un package lexer contenant le Token, la table des symboles et le Lexer,
un package parse contenant le Parser et la classe Main,
un menu interactif permettant à l'utilisateur de choisir l'analyse lexicale ou syntaxique.
L'objectif final est d'obtenir un mini-compilateur simple mais entièrement fonctionnel.

Chapitre 2 : Analyseur lexical

2.1 Définition et rôle

L'analyseur lexical (ou Lexer) est la première étape de tout compilateur.
Son rôle est de transformer le texte source en une suite de lexèmes (tokens), chaque lexème représentant une unité significative du langage, comme :
_un mot-clé (int, while, etc.),
_un identifiant (x, yassmina, etc.),
_un nombre (5, 100, etc.),
_un opérateur (+, ==, etc.),
_un séparateur (;, (,), etc.),
_un lexème invalide s'il ne correspond à aucune catégorie connue.
Le Lexer permet donc de préparer le code pour l'analyse syntaxique en fournissant une représentation structurée sous forme de tokens.

2.2 Organisation et fonctionnement du Lexer

Le Lexer est constitué de trois classes principales :
Token : représente chaque lexème par un type et sa valeur.

SymboleTable : contient tous les mots-clés, opérateurs et séparateurs connus, et permet de déterminer le type d'un lexème.
Lexer : parcourt le texte caractère par caractère, détecte le type de chaque lexème et crée la liste de tokens.
Le Lexer fonctionne en suivant cette logique :
1.Ignorer les espaces et sauts de ligne.
2.Identifier les lettres → identifiants ou mots-clés.
3.Identifier les chiffres → nombres.
4.Identifier les opérateurs simples ou doubles.
5.Identifier les séparateurs.
Tout ce qui ne correspond à aucune catégorie devient un token invalide.

2.3 Détection des lexèmes et mots-clés

Pour détecter correctement les lexèmes :
Le Lexer utilise la SymboleTable pour vérifier si une chaîne correspond à un mot-clé ou à un opérateur connu.
Si elle n'existe pas dans la table, elle est considérée comme un identifiant.
Les opérateurs peuvent être simples (+) ou doubles (==).
Les mots-clés spécifiques du projet, comme yasmineyahia, sont aussi reconnus.

2.4 Gestion des erreurs lexicales avec try-catch

Lorsqu'un caractère inconnu ou invalide est rencontré, le Lexer :
crée un token de type INVALIDE,
continue l'analyse du reste du code,
utilise des blocs try-catch pour éviter que le programme ne plante.
Ainsi, toutes les erreurs lexicales sont détectées et affichées sans interrompre l'exécution.

2.5 AFD utilisé pour le Lexer

L'AFD (Automate Fini Déterministe) utilisé décrit comment le Lexer lit le texte et identifie les tokens :
q0 : état initial → lettre → q1, chiffre → q2, opérateur → q3, séparateur → q4, autre → q5
q1 : identifiant / mot-clé → lettres/chiffres continuent, sinon retour à q0
q2 : nombre → chiffres continuent, sinon retour à q0
q3 : opérateur → opérateur simple ou double, puis retour à q0
q4 : séparateur → fin du token, retour à q0
q5 : invalide → fin du token, retour à q0
qf : fin du texte → token FIN #
L'AFD est déterministe : chaque caractère conduit à un état précis, permettant de reconnaître tous les lexèmes valides et de détecter les erreurs lexicales.

2.6 Grammaire correspondant au Lexer

La grammaire du Lexer décrit les règles permettant de reconnaître tous les tokens du code source. Elle est simplifiée et correspond directement à la logique implémentée dans la classe **Lexer.**

<token> ::= <mot_cle> | <identifiant> | <nombre> | <operateur> | <separateur> | <invalide>

<mot_cle> ::= "int" | "double" | "float" | "String" | "char" | "boolean"
| "if" | "else" | "while" | "for" | "do" | "switch" | "case"
| "break" | "continue" | "return" | "public" | "private" | "protected"
| "static" | "final" | "class" | "void" | "new" | "try" | "catch"
| "yasmineyahia"

<identifiant> ::= lettre (lettre | chiffre)*

<nombre> ::= chiffre+

<operateur> ::= "+" | "-" | "*" | "/" | "=" | "==" | "!=" | "<" | ">" | "<=" | ">="
| "&&" | "||" | "%" | "++" | "--" | "+=" | "-=" | "*=" | "/="

<separateur> ::= ";" | "," | "(" | ")" | "{" | "}" | "[" | "]" | "."

<invalide> ::= tout caractère non reconnu

Explication :
Chaque lexème du code source correspond à un token selon cette grammaire.
Les mots-clés et opérateurs sont définis dans la SymboleTable.
Les identifiants et nombres sont reconnus grâce à la lecture des lettres et chiffres.
Les caractères invalides sont signalés comme tokens INVALIDE.
Cette grammaire reflète directement le fonctionnement de l'AFD utilisé dans le Lexer et garantit la reconnaissance correcte de tous les lexèmes du programme

2.2.6.La Matrice:

Matrice de transition :

État / Caractère	L (lettre/_)	D (chiffre)	O (opérateur)	S (séparateur)	A (autre)	FIN
q0	q1	q2	q3	q4	q5	qf
q1	q1	q1	q0	q0	q0	qf
q2	q0	q2	q0	q0	q0	qf
q3	q0	q0	q3 (double?)	q0	q0	qf
q4	q0	q0	q0	q0	q0	qf
q5	q0	q0	q0	q0	q0	qf
qf	-	-	-	-	-	-

Chapitre 3 : Analyseur syntaxique

3.1 Définition et rôle:

L'analyseur syntaxique (Parser) est la deuxième étape du compilateur.
Son rôle est de vérifier que la suite de tokens respecte une grammaire définie.
Dans notre projet, la grammaire simplifiée permet de vérifier :
si une expression commence par un identifiant, un nombre ou un mot-clé personnalisé,
si chaque opérateur est suivi d'une valeur valide,
si les points-virgules terminent correctement les expressions,
si les parenthèses sont équilibrées.

3.2 Organisation et fonctionnement du Parser

Le Parser fonctionne en lisant la liste des tokens générés par le Lexer.
Il effectue plusieurs vérifications :
Parcours séquentiel de tous les tokens.
Détection des expressions correctes ou incorrectes.
Suivi des parenthèses avec une pile (stack) pour s'assurer de leur équilibre.
Signalement des erreurs syntaxiques rencontrées.

3.3 Vérification de la grammaire et détection des erreurs

Le Parser identifie les erreurs suivantes :
Expression mal formée au départ.
Opérateur isolé ou mal suivi.
Point-virgule manquant à la fin d'une expression.
Parenthèses ouvrantes non fermées ou fermantes sans ouvrante.
Caractères invalides provenant du Lexer.
Chaque erreur est affichée à l'écran pour informer l'utilisateur de la nature du problème.

3.4 Gestion des erreurs syntaxiques avec try-catch

Le Parser utilise également des blocs try-catch pour :
éviter l'arrêt brutal du programme lors d'erreurs inattendues,
continuer l'analyse malgré la présence d'erreurs,
garantir un retour clair sur l'état du code analysé.
Ainsi, le Parser est robuste et fiable pour un mini-compilateur.

Chapitre 4 : Classe Main

4.1 Présentation et rôle

La classe Main est l'interface utilisateur du mini-compilateur.
Elle permet :
de choisir entre **analyse lexicale** et **analyse syntaxique**,
de saisir du code ligne par ligne,
d'afficher les tokens générés,
de lancer le Parser et d'afficher les résultats.
Elle centralise toutes les interactions entre l'utilisateur, le Lexer et le Parser.

4.2 Choix interactifs pour l'utilisateur (lexical ou syntaxique)

Le programme propose un menu interactif :
Tester l'analyse lexicale → affiche la liste des tokens.
Tester l'analyse syntaxique → affiche les tokens et vérifie la syntaxe.
Quitter → termine l'exécution.
L'utilisateur peut entrer autant de lignes de code qu'il le souhaite.
Une ligne vide déclenche le traitement des lignes saisies.

4.3 Gestion des erreurs avec try-catch

Dans la classe **Main**, les blocs try-catch jouent un rôle essentiel pour assurer **la robustesse** du programme :

Prévenir les plantages : toute exception inattendue (comme un choix invalide ou une erreur lors de la lecture du code) est capturée, évitant que le programme se termine brutalement.
Informer l'utilisateur : les messages d'erreur clairs sont affichés pour indiquer la nature du problème. Par exemple, si l'utilisateur entre un choix de menu invalide, le programme signale "Erreur : Choix invalide !".
Permettre la continuité : même en présence d'erreurs, le programme continue de fonctionner, permettant à l'utilisateur de corriger ses saisies et de poursuivre l'analyse lexicale ou syntaxique.
Ainsi, les try-catch dans Main garantissent que l'interface reste fiable et conviviale, même si l'utilisateur fait des erreurs ou si des situations inattendues apparaissent.

```
 try {
            if (choix.equals("1")) {
               System.out.println("Entrez votre code ligne par ligne. Entrez une ligne vide pour
terminer :");
               StringBuilder code = new StringBuilder();
               while (true) {
                  String ligne = sc.nextLine();
                  if (ligne.isEmpty()) break;
                  code.append(ligne).append("\n");
               }
               Lexer lexer = new Lexer(code.toString());
               List<Token> tokens = lexer.tokenize();
               System.out.println("\n--- Lexèmes générés ---");
               lexer.printTokens();

            } else if (choix.equals("2")) {
               System.out.println("Entrez votre code ligne par ligne. Entrez une ligne vide pour
terminer :");
               StringBuilder code = new StringBuilder();
               while (true) {
                  String ligne = sc.nextLine();
                  if (ligne.isEmpty()) break;
                  code.append(ligne).append("\n");
               }
               Lexer lexer = new Lexer(code.toString());
               List<Token> tokens = lexer.tokenize();
               System.out.println("\n--- Lexèmes générés ---");
               lexer.printTokens();

               Parser parser = new Parser(tokens);
               parser.parse();

            } else if (choix.equalsIgnoreCase("q")) {
               continuer = false;
               System.out.println("Fin du programme !");
            } else {
               throw new Exception("Choix invalide !");
            }
         } catch (Exception e) {
            System.out.println("Erreur : " + e.getMessage());
         }
        System.out.println("\n----------------------------------------------\n");}sc.close();}}
```

4.4 Exécution et tests ligne par ligne

Exemple d'exécution :

```ini
x = yasmineyahia + 5;
```

Le Lexer produit les tokens correspondants.
Le Parser vérifie la syntaxe.
Les erreurs éventuelles sont affichées grâce à la gestion des exceptions dans Main et dans les classes Lexer/Parser.
Si aucun problème n'est détecté, le programme indique que l'analyse est terminée avec succès

Conclusion

Le développement de ce mini-projet de compilation a offert une vision pratique et complète des étapes fondamentales d'un compilateur. En réalisant un analyseur lexical et un analyseur syntaxique, nous avons pu comprendre comment un code source est transformé en une représentation structurée et vérifiée.
L'analyse lexicale, mise en œuvre dans la classe Lexer, permet de reconnaître automatiquement les mots-clés, identifiants, nombres, opérateurs et séparateurs. L'utilisation d'un AFD déterministe et d'une table de symboles garantit que chaque lexème est correctement identifié, tout en gérant les caractères invalides grâce à des tokens spécifiques. Les blocs try-catch intégrés assurent une robustesse et une stabilité du programme face aux erreurs lexicales, permettant au processus de continuer sans interruption.

L'analyse syntaxique, implémentée dans la classe Parser, vérifie la structure des expressions et la cohérence des opérations, y compris la gestion des parenthèses et des opérateurs. Cette étape est cruciale pour détecter les erreurs logiques et syntaxiques avant toute exécution réelle du code. Elle offre une rétroaction immédiate à l'utilisateur, favorisant la correction rapide des erreurs.
La classe Main complète le projet en fournissant une interface interactive et conviviale, où l'utilisateur peut choisir entre analyse lexicale et syntaxique, entrer son code ligne par ligne, et visualiser les résultats directement. L'emploi de try-catch dans Main assure également la gestion des erreurs liées aux choix de l'utilisateur ou à des entrées incorrectes, renforçant la fiabilité globale de l'application.

En somme, ce mini-compilateur a permis de mettre en pratique les concepts théoriques de la compilation, de la reconnaissance lexicale à l'analyse syntaxique, tout en intégrant des mécanismes de gestion d'erreurs et une interface interactive. Ce projet constitue une excellente introduction aux principes de conception d'un compilateur et offre une base solide pour des développements plus avancés dans le domaine de l'analyse de code et des langages de programmation.