Rapport de Mini-Projet

Réalisation d'un Analyseur Lexical (Lexer)
et d'un Analyseur Syntaxique (Parser) pour
l'instruction Try/Catch en JS

Auteurs :

Samy Abdelkader ACHOUCHE

Module de Compilation / TP

8 décembre 2025

Table des matières

Chapitre 1

Introduction

Ce document décrit le fonctionnement d'un compilateur partiel écrit en Java, composé d'un **Lexer** (Analyseur Lexical) et d'un **Parser** (Analyseur Syntaxique).

L'objectif de ce module est double :

1. Lire un code source brut et le transformer en une suite de **Tokens** (unités lexicales).
2. Analyser cette suite de tokens pour vérifier la conformité syntaxique, en se concentrant particulièrement sur la structure de gestion d'erreurs `try/catch`.

Le compilateur reconnaît un sous-ensemble du langage JavaScript et intègre des mots-clés personnalisés.

Chapitre 2

Architecture du Projet

L'application est structurée de manière modulaire, séparant l'interface utilisateur, l'analyse lexicale et l'analyse syntaxique.

2.1 Flux de Données

Le schéma suivant illustre le cheminement des données depuis la saisie de l'utilisateur jusqu'au résultat de l'analyse.

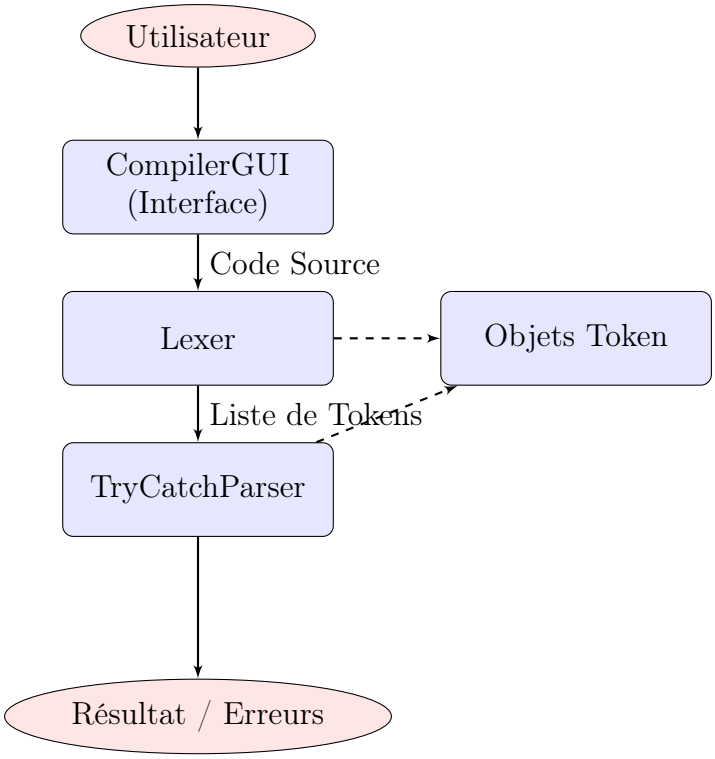

FIGURE 2.1 – Diagramme de flux de données du compilateur

2.2 Organisation des Fichiers

L'architecture du projet est organisée en packages distincts, chacun ayant une responsabilité claire :

src/Main.java Point d'entrée de l'application.

src/GUI/
Contient `CompilerGUI.java`, l'interface graphique (Swing) permettant à l'utilisateur de saisir le code.

src/Lexer/ (Analyse Lexicale)
- `Lexer.java` : Le moteur qui découpe le texte.
- `Token.java` : Structure de données représentant un mot.
- `TokenType.java` : Énumération des types (Mots-clés, Opérateurs...).

src/Parser/ (Analyse Syntaxique)
- `TryCatchParser.java` : Analyseur descendant récursif.
- `ParseException.java` : Exception personnalisée avec localisation.
- `ErrorHandler.java` : Gestionnaire de diagnostic intelligent.

Chapitre 3

Analyse Syntaxique (Parser)

L'analyseur syntaxique (`TryCatchParser`) a été conçu selon le modèle **LL(1)**, c'est-à-dire qu'il lit le code de gauche à droite et prend des décisions en ne regardant qu'un seul token à l'avance.

3.1 Méthodologie de Parsing

Le parser utilise deux méthodes fondamentales pour naviguer dans le flux de tokens :

- `parcourire()` : Avance au token suivant en ignorant automatiquement les commentaires (`SINGLE_LINE` et `MULTI_LINE`).
- `expect(TokenType type)` : Vérifie que le token courant correspond au type attendu. Si c'est le cas, elle avance ; sinon, elle lève une `ParseException`.

3.2 Gestion de la structure Try/Catch

Le parser applique strictement la grammaire suivante pour les blocs de gestion d'erreurs, implémentée dans la méthode `parseTryStatement` :

```
try_statement -> "try" block catch finally?
catch  -> "catch" "(" ID ")" block
finally -> "finally" block
```

Points d'attention implémentés dans le code :

- **Catch Obligatoire :** Le parser impose la présence d'un bloc `catch` après un `try`.
- **Paramètre de Catch :** Le parser vérifie spécifiquement que le `catch` possède un paramètre (identifiant).

Chapitre 4

Gestion Avancée des Erreurs

Une des forces de ce compilateur réside dans sa classe `ErrorHandler`. Elle ne se contente pas de signaler une erreur, elle tente de l'expliquer.

4.1 Diagnostics Intelligents

La méthode `diagnose(ParseException pe)` analyse le message d'erreur brut pour fournir des explications contextuelles :

Symptôme détecté	**Explication fournie à l'utilisateur**
Message contient "catch" et "attendu"	"Explication : Après un bloc 'try', un 'catch' est attendu. Vérifiez la présence de 'catch (e) { ... }'."
"paramètre manquant" ou "identifiant"	"Explication : Le catch doit recevoir un identifiant entre parenthèses."
"bloc non fermé"	"Explication : Un '{' n'a pas été fermé par un '}'. Vérifiez les accolades imbriquées."
"bloc attendu"	"Explication : Un bloc (accolades) était attendu ici."

Chapitre 5

Spécifications Lexicales

(Cette section détaille les définitions des Tokens)

5.1 Liste des Tokens et Motifs

— **Identifiant** : $(A-Za-z\_\$)(A-Za-z0-9\_\$)*$
— **Nombres** : Entiers ou décimaux $([0-9]+(.[0-9]+)?)$.
— **Chaînes** : Délimitées par ' ou ", supportent l'échappement.
— **Commentaires** : `//` (ligne) et `/* ... */` (bloc).

5.2 Mots-clés Spéciaux

Le parser rejette explicitement l'utilisation des mots-clés réservés aux auteurs (`SAMY`, `ACHOUCHE`, `ABDELKADER`) s'ils sont utilisés comme instructions simples.

Chapitre 6

Grammaire

La grammaire supportée par `TryCatchParser.java` est la suivante :

Listing 6.1 – Grammaire

```
Program -> Statement*

Statement -> TryStatement
           | VariableDeclaration
           | Block
           | ExpressionStatement
           | ";"

TryStatement -> "try" Block Catch Finally?
Catch -> "catch" "(" ID ")" Block
Finally -> "finally" Block

Block -> "{" Statement* "}"

VariableDeclaration -> ("let" | "var" | "const") ID ("=" Expression
   )? ";"

ExpressionStatement -> Expression ";"

Expression -> Assignment
Assignment -> Comparison ("=" Assignment)?
Comparison -> Primary (OperatorComp Primary)*
Primary -> ID | NUMBER | STRING | BOOLEAN | "(" Expression ")"

OperatorComp -> "==" | "!=" | "<" | "<=" | ">" | ">="
```

Chapitre 7

Utilisation

L'utilisation est simplifier au maximum inspirer par la forme generale des IDE j'ai realiser une interface simple ou l'utilisateur a le choix entre introduire sans code dans la zone du code ou bien il choisis le fichier qui contient le code a tester ,se dernier s'execute directement au moment de la selection , pour la premier option il ya un boutton pour activer l'analyseur . Pour plus de details consulter le repository sur mon compt github Compt Github