Université Abderrahmane Mira de Béjaïa

Faculté des Sciences Exactes
Département d'Informatique

RAPPORT DE PROJET TP MINI-COMPILATEUR PYTHON

Réalisé par :
Taguelmimt Badis

Module :
Compilation

Niveau : L3 Informatique
Groupe : B3

Année Universitaire :
2025 / 2026

8 décembre 2025

Table des matières

1 Introduction

Ce projet a pour but la réalisation d'un mini-compilateur pour un sous-ensemble du langage Python. L'objectif pédagogique est de comprendre et d'implémenter les phases d'analyse lexicale et syntaxique, ainsi que la gestion des erreurs, sans recourir à des outils de génération automatique.

Conformément au cahier des charges, l'analyseur syntaxique vérifie la correction de quatre éléments majeurs :

— Les **déclarations** (classes, méthodes, variables).

— Les **affectations** (variables, constantes, expressions arithmétiques et logiques).

— Les **comparaisons** (opérateurs relationnels).

— L'instruction principale imposée : la structure `try/except`.

Les autres structures de contrôle (comme `if`, `while` ou `for`) sont traitées de manière structurelle simplifiée ("ignorées"), c'est-à-dire que leur présence est validée syntaxiquement (mot-clé + deux-points + bloc indenté) sans analyse approfondie de leur contenu logique.

2 La Grammaire Choisie

Pour réaliser l'analyseur syntaxique, nous avons défini une grammaire de type **LL(1)** (lecture de gauche à droite, dérivation à gauche).

Afin de modéliser formellement l'ignorance du contenu de certaines structures (comme les conditions des `if`), nous utilisons des règles récursives droites qui consomment des tokens jusqu'à un délimiteur spécifique.

2.1 Structure Globale

```
Z                   -> Programme EOF
Programme           -> SautsLigne ListeInstructions
ListeInstructions   -> Instruction ListeInstructions | epsilon
```

2.2 Instructions Analysées (Focus)

Ces instructions sont vérifiées en détail (structure et sémantique).

```
Instruction         -> TryExcept | Declaration | Statement | StructureIgnoree

# 1. Instruction Principale (Sujet)
TryExcept           -> 'try' ':' Bloc Excepts FinallyOpt
Excepts             -> 'except' TypeExceptOpt ':' Bloc Excepts | epsilon
TypeExceptOpt       -> ID AsOpt | epsilon
AsOpt               -> 'as' ID | epsilon
FinallyOpt          -> 'finally' ':' Bloc | epsilon

# 2. Déclarations
DeclarationMethode  -> 'def' ID '(' ParamsIgnores ')' ':' Bloc
DeclarationClasse   -> 'class' ID ':' Bloc
```

```
# 3. Affectations et Comparaisons
Statement           -> Affectation | Expression
Affectation         -> ID OpAssign Expression
OpAssign            -> '=' | '+=' | '-='
Expression          -> Terme ExpPrime
```

2.3 Instructions Ignorées (Structurelles)

Pour ces instructions, on vérifie uniquement la présence du mot-clé, des deux-points : et d'un bloc indenté. Le symbole non-terminal `ContenuIgnoré` modélise la séquence de tokens "sautée" par le parser.

```
StructureIgnoree    -> MotCleStruct ContenuIgnoré ':' Bloc
MotCleStruct        -> 'if' | 'while' | 'for' | 'elif' | 'else'

InstructionSimple   -> MotCleSimple ResteLigne NEWLINE
MotCleSimple        -> 'print' | 'return' | 'break' | 'continue' | 'pass'
```

2.4 Règles de "Consommation" (Ignorance)

Ces règles formalisent la boucle `while` utilisée dans le code pour ignorer le contenu jusqu'à un délimiteur.

```
# Consomme tout token sauf ':' (utilisé pour les conditions if/while)
ContenuIgnoré       -> TokenSaufColon ContenuIgnoré | epsilon

# Consomme tout token sauf ')' (utilisé pour les paramètres de fonction)
ParamsIgnores       -> TokenSaufParFermante ParamsIgnores | epsilon

# Consomme tout jusqu'à la fin de la ligne (utilisé pour print/return)
ResteLigne          -> TokenSaufNewline ResteLigne | epsilon
```

3 L'Analyseur Lexical

L'analyseur lexical, implémenté dans la classe `Lexer.java`, transforme le code source brut en une suite de `Token`.

3.1 Approche Lexicale Manuelle

Contrairement à une approche reposant sur des générateurs automatiques (comme JFlex) ou des expressions régulières complexes, nous avons développé un **scanner manuel caractère par caractère**.

Cette méthode est **procédurale** : au lieu d'une table de transition d'états classique, nous utilisons des structures de contrôle algorithmiques (boucles et conditions) pour identifier les tokens. Cela permet :

— Un contrôle total sur la consommation du flux d'entrée (par exemple, avancer le curseur manuellement après un nombre).

— Une meilleure performance ($O(n)$) en évitant la surcharge des moteurs de regex.
— Une gestion fine des cas particuliers comme les commentaires ou les chaînes de caractères.

3.2 Gestion Avancée de l'Indentation

La particularité de Python réside dans l'utilisation de l'indentation sémantique. Notre Lexer implémente un mécanisme de "tokens virtuels" :

— Une pile `niveauxIndent` mémorise l'état des tabulations.
— L'algorithme calcule les deltas d'indentation à chaque début de ligne.
— Il injecte artificiellement des tokens `INDENT` ou `DEDENT` dans le flux, rendant la structure transparente pour l'analyseur syntaxique qui suit.

4 L'Analyseur Syntaxique

L'analyseur syntaxique (`Parser.java`) vérifie que la suite de tokens respecte la grammaire définie, selon une approche descendante récursive.

4.1 Conformité LL(1) et Efficacité

L'analyseur est conçu selon le modèle **LL(1)**, ce qui lui permet d'analyser le code de gauche à droite en anticipant un seul symbole.

— **Déterminisme :** L'analyseur identifie immédiatement la règle à appliquer grâce au token courant, sans aucune ambiguïté.
— **Performance :** Cette méthode évite tout retour en arrière (*no backtracking*). Le fichier est lu en une seule passe, garantissant une complexité linéaire $O(n)$.

4.2 Gestion des Ambiguïtés (Lookahead)

Pour distinguer une affectation (`x = 1`) d'une expression (`x + 1`) sans consommer les tokens prématurément, nous utilisons une technique de **Lookahead** (anticipation). Le parser consulte le token à l'index $i + 1$ sans déplacer le curseur de lecture, permettant de choisir la bonne branche syntaxique instantanément.

4.3 Robustesse et Tolérance

Pour garantir l'utilisabilité du compilateur, une boucle de nettoyage a été intégrée dans `ListeInstructions`. Elle filtre les tokens structurels parasites (sauts de ligne multiples, indentations vides), rendant l'analyseur robuste face aux variations de mise en forme du code source.

5 Gestion des Erreurs

La gestion des erreurs est centralisée dans le package `errors`.

— **Error.java** : Représente une erreur avec son type (LEXICAL ou SYNTAXIQUE), un message, la ligne et la colonne.

— **ErrorService.java** : Stocke la liste des erreurs rencontrées.

Le compilateur ne s'arrête pas immédiatement à la première erreur (sauf cas critique). Il tente de continuer l'analyse pour rapporter le maximum de problèmes à l'utilisateur.

6 Structure du Projet

L'arborescence respecte le modèle MVC standard :

— **compilateur**/

  — `Interface.java` : Fenêtre graphique (Swing) et point d'entrée principal.

— **lexer**/

  — `Lexer.java` : Logique de tokenisation.

  — `Token.java` : Objet Token (Type, Valeur, Ligne).

  — `TypeToken.java` : Enumération des types possibles.

— **parser**/

  — `Parser.java` : Cœur de l'analyse syntaxique.

— **errors**/

  — `Error.java` et `ErrorService.java`.

7 Jeux de Test

7.1 Test 1 : Instruction Try/Except (Succès)

Entrée :

```
try:
    x = 10
except Exception:
    print("Erreur")
```

Sortie : `COMPILATION REUSSIE`

7.2 Test 2 : Instruction If (Ignorée mais Validée)

Entrée :

```
if x > 0 + * / :  # Condition syntaxiquement fausse mais ignoree
    x = x + 1
```

Sortie : `COMPILATION REUSSIE`

Note : Ce test valide que le parser ignore bien le contenu de la condition mais vérifie la présence des deux-points et du bloc indenté.

7.3 Test 3 : Erreur Syntaxique

Entrée :

```
try
    print("Oubli des deux points")
```

Sortie : `[ERREUR SYNTAXIQUE] ligne 1, colonne 4 -> Token inattendu : print`

8 Conclusion

Ce projet a permis de mettre en pratique les concepts théoriques de la compilation. La contrainte d'ignorer certaines structures tout en en analysant d'autres a nécessité une conception flexible du parser. L'utilisation d'une interface graphique et la gestion manuelle du Lexer rendent le projet complet, fonctionnel et robuste.