UNIVERSITÉ DE BEJAIA
FACULTÉ DES SCIENCES EXACTES
DÉPARTEMENT D'INFORMATIQUE

RAPPORT DE PROJET

Module : Compilation

Thème du projet : For each python

Réalisé par : SAIDANI Rabah

Année Académique : 2025/2026

Introduction Générale

Ce projet s'inscrit dans le cadre du module de **La Compilation**. L'objectif principal est la conception et l'implémentation d'un mini-compilateur pour un sous-ensemble du langage **Python**, en mettant l'accent sur la structure de contrôle itérative **« For »**. Contrairement aux compilateurs classiques pour des langages comme le C ou Java, le choix de Python impose une difficulté supplémentaire : la gestion de la structure par **indentation**, qui remplace les délimiteurs de blocs traditionnels.

Analyseur Lexical

Dans le cadre de la réalisation du mini-compilateur, la première étape fondamentale est le développement de l'**Analyseur Lexical** (ou *Scanner*). Ce module a pour responsabilité de lire le flux de caractères brut du code source et de le segmenter en unités logiques significatives appelées **Tokens**.

Ce chapitre détaille l'implémentation de mon analyseur lexical adapté à un sous-ensemble du langage **Python**. Il met l'accent sur trois spécificités : la reconnaissance de la structure for, la gestion de l'indentation significative (propre à Python) et l'utilisation d'automates finis pour la validation des lexèmes.

Structure des Données : Le Token

Le cœur de l'analyseur repose sur la classe Token. Chaque unité lexicale identifiée est instanciée sous forme d'objet comportant deux attributs principaux :

- **Type (`TokenType`)** : La catégorie grammaticale du mot (ex: FOR, IDENTIFIER, NUMBER, LTAB).
- **Valeur (`valeur`)** : La chaîne de caractères brute extraite du code source (le lexème).

La liste des types, définie via une énumération (Enum), couvre les catégories suivantes :

- **Mots-clés réservés** : for, in, range, if, else, class, def, print.
- **Opérateurs et Séparateurs** : `+`, `-`, `:`, `(`, `)`, `=`, etc.
- **Contrôle de blocs (Spécifique Python)** : LTAB (Début de bloc/Indentation) et RTAB (Fin de bloc/Désindentation).
- **Types dynamiques** : IDENTIFIER (pour les variables et noms de fonctions) et NUMBER (pour les constantes entières).
- **Mots-clés personnalisés** : Saidani, Rabah (Signature du compilateur).

Architecture de l'Analyseur

L'analyseur adopte une approche **hybride**, combinant une boucle de lecture linéaire pour les symboles simples et un **Automate Fini Déterministe (AFD)** pour la reconnaissance complexe des identificateurs et des nombres.

Algorithme Principal (`tokeniser`)

La méthode **tokeniser()** parcourt le fichier source caractère par caractère en appliquant une logique de priorité stricte pour résoudre les ambiguïtés :

1. **Gestion des Commentaires (#)** : Priorité absolue. Dès qu'un symbole `#` est détecté, l'analyseur ignore l'intégralité du reste de la ligne.
2. **Gestion de l'Indentation (`\n`)** : À chaque saut de ligne, l'analyseur déclenche la fonction verif_s_tab() pour analyser la structure des blocs (voir section 4).
3. **Symboles Fixes** : Les opérateurs et ponctuation (ex: `:`, `+`) sont reconnus directement via une structure conditionnelle (switch ou if/else).
4. **Alphanumériques** : Si le caractère lu est une lettre ou un chiffre, le contrôle est délégué à l'automate matriciel.

L'Automate Matriciel (DFA)

Pour distinguer efficacement les **Identificateurs** des **Nombres** et rejeter les lexèmes invalides, j'ai implémenté une matrice de transition M

```
private final int[][] M = {

   // LETTRES   CHIFFRES   AUTRE

   { 1,        2,        -1 }, // État 0 (Initial)

   { 1,        1,        -1 }, // État 1 (Identificateur)

   { -1,       2,        -1 }  // État 2 (Nombre)

};
```

Définition des États :

- **État 0 (Initial)** : Point d'entrée de l'automate.
- **État 1 (Identificateur)** : Atteint si le premier caractère est une lettre. Cet état boucle sur lui-même tant que les caractères suivants sont des lettres ou des chiffres.
- **État 2 (Nombre)** : Atteint si le premier caractère est un chiffre. Cet état n'accepte ensuite que d'autres chiffres.
- **État -1 (Erreur)** : Atteint en cas de transition interdite (exemple : un nombre suivi immédiatement d'une lettre, comme `2x`).

Expression Régulière équivalente :

$$(L(L+C)^*) + (C(C)^*)$$

Cette matrice assure une validation stricte : une variable ne peut jamais commencer par un chiffre, garantissant ainsi la conformité avec les normes standards de compilation.

Filtrage des Mots-Clés

Une fois qu'un lexème est validé par l'automate comme étant un "Identificateur" (État 1), il subit une vérification supplémentaire via la fonction verifierMotCle().

Cette étape permet de distinguer :

- Les mots réservés du langage (for, while, etc.).
- Les mots-clés de signature personnalisé (Saidani, Rabah).
- Les vrais identificateurs (noms de variables).

Si le lexème ne correspond à aucun mot réservé, il conserve son type `IDENTIFIER`.

Gestion de l'Indentation (Spécificité Python)

Contrairement aux langages comme le C ou Java qui délimitent les blocs par des accolades `{ }`, Python utilise l'indentation. C'est le défi technique majeur de ce module lexical.

J'ai implémenté la fonction verif_s_tab() qui opère à chaque début de ligne selon la logique suivante :

1. **Calcul de la profondeur** : Compte le nombre d'espaces ou de tabulations au début de la ligne actuelle (d).

1. **Comparaison avec le niveau précédent** (blud) :
    - **Si d > blud** : On entre dans un nouveau bloc. L'analyseur génère un token LTAB.
    - **Si d < blud** : On sort d'un ou plusieurs blocs. L'analyseur génère autant de tokens RTAB que nécessaire pour revenir au niveau d'indentation correspondant, fermant ainsi les blocs ouverts.
    - **Si d == blud** : On reste dans le même bloc, aucune action n'est requise.

Cette fonction intègre également une gestion robuste des lignes vides et des lignes de commentaires, afin de ne pas fausser la détection de la structure logique du programme.

. 4. Analyse Syntaxique (Parser)

Après l'analyse lexicale qui a transformé le code source en une suite linéaire de *tokens*, la seconde étape majeure du compilateur est l'**Analyse Syntaxique**. Le rôle de ce module est de vérifier que la suite de tokens respecte la structure grammaticale du langage. Il prend en entrée la liste fournie par le *Scanner* et valide si l'agencement des mots forme des phrases correctes selon les règles définies.

Dans ce projet, nous avons implémenter la méthode descendante récursive LL(1) pour le fonctionnement de l'analyseur syntaxique. Ce choix se justifie par plusieurs raisons :

1. **Correspondance directe** : Chaque règle de la grammaire (Non-Terminal) correspond directement à une fonction Java (ex: la règle <FOR> devient la méthode InstructionFor()).
2. **Efficacité** : Notre grammaire étant de type **LL(1)** (lecture de gauche à droite, dérivation à gauche, avec 1 token d'avance), l'analyseur n'a jamais besoin de revenir en arrière . Le token actuel suffit pour décider quelle fonction appeler.

Le Mécanisme de Consommation (`match`) : L'analyseur repose sur une primitive fondamentale : la fonction `match(Token attendu)`. Son rôle est double :

1. **Vérification** : Elle compare le token courant (pointé par le curseur `i`) avec le type attendu par la grammaire.
2. **Consommation** : Si le type correspond, elle incrémente le curseur (`i++`) pour passer au mot suivant. Sinon, elle signale une erreur sans avancer, permettant à l'analyseur de tenter une récupération.

Grammaire implémenter :

A. Structure Globale

Un programme est une suite d'instructions qui s'enchaînent récursivement.

```
<S> -> <INSTRUCTION> <S> | epsilon
```

B. Le Dispatcher (Sélecteur)

L'analyseur vérifie le type du token actuel pour orienter l'analyse vers la bonne fonction.

```
<INSTRUCTION>  -> <INSTR_FOR> | <DECL_CLASSE> | <DECL_METHODE>|
<AFFECTATION> | <INSTR_PRINT> | <SIGNATURE> | <INSTR_IGNORE>
```

C. Règles du Thème "Boucle For" (Strictes)

```
<INSTR_FOR>  -> "for" id "in" "range" "(" <EXPRESSION> ")" ":"
<BLOC>
```

D. Règles des Déclarations

```
<DECL_CLASSE>  -> "class" id ":" <BLOC>

<DECL_METHODE> -> "def" id "(" ")" ":" <BLOC>

<AFFECTATION>  -> id "=" <EXPRESSION>

<INSTR_PRINT>  -> "print" <EXPRESSION>

<SIGNATURE>    -> "NOM_PRENOM"
```

Nom et prénom qui corresponde aux tokens personnalisé

E. Gestion des Blocs (Indentation)

L'indentation est traitée comme une structure explicite grâce aux tokens générés par l'analyseur lexical.

```
<BLOC>   -> "LTAB" <S> "RTAB" | <INSTRUCTION>
```

F. Structures Ignorées (Hors-Thème)

les structures comme if ou while sont reconnues pour éviter les erreurs, mais leur contenu n'est pas vérifié.

```
<INSTR_IGNORE> -> ("if" | "while" | "else") ... ":" <BLOC>
```

E. Gestion des Expressions

Les calculs sont gérés par une grammaire récursive simple pour supporter les opérations de base.

```
<EXPRESSION>   -> <TERME> <SUITE_EXPR>
<SUITE_EXPR>   -> <OPERATEUR> <EXPRESSION> | epsilon
<TERME>        -> id | number | "(" <EXPRESSION> ")"
<OPERATEUR>    -> "+" | "-" | ">" | "<" | "=="
```

Gestion des Erreurs et Robustesse

Une fonctionnalité clé de cet analyseur est sa capacité de **gérer les erreurs lexicales et syntaxiques**. Au lieu d'arrêter brutalement le programme à la première faute , l'analyseur :

1. Signale l'erreur avec un message précis dans l'interface (ex: *"Ligne 5 : ')' manquante après range"*).
2. Marque le code comme "INVALIDE" via un booléen global.
3. Tente de consommer le token problématique pour se resynchroniser et continuer l'analyse du reste du fichier

Conclusion

La réalisation de ce mini-compilateur a été une expérience enrichissante qui nous a permis de concrétiser les concepts théoriques vus en cours, tels que **l'analyse lexical** avec matrice **et syntaxique** via les algorithmes d'analyse syntaxique **LL(1)**.

Au terme de ce projet, nous avons abouti à une application fonctionnelle capable de :

- Reconnaître et valider lexicalement un code source Python.
- Gérer la spécificité de l'indentation Python grâce à un mécanisme de tokens virtuels (LTAB, RTAB).

- Vérifier syntaxiquement la structure de la boucle for, ainsi que les déclarations de classes et de fonctions.
- Signaler les erreurs avec précision et proposer une récupération sur erreur pour ne pas bloquer l'analyse.