Université A. MIRA – BEJAIA

Département d'Informatique

RAPPORT Sur Application Java de Gestion Superette

Année universitaire 2025–2026

Réalisé par :

SAHRAOUI Samy

GROUPE :3 SECTION:B

Date : December 8, 2025

Introduction

Ce mini-projet s'inscrit dans le module de Compilation et consiste à réaliser un mini-compilateur en Java comprenant deux phases principales : l'analyse lexicale et l'analyse syntaxique. L'objectif pédagogique est de comprendre :

- la construction d'un analyseur lexical basé sur des automates finis déterministes (AFD),
- la conception d'un analyseur syntaxique par descente récursive,
- l'utilisation d'une grammaire simplifiée inspirée du langage C.

Le compilateur développé reconnaît trois formes d'instructions : les déclarations, les affectations et l'opérateur ternaire. Il analyse de plus des expressions arithmétiques, des identificateurs, des constantes, ainsi que des opérateurs logiques et arithmétiques.

Grammaire du Langage

La grammaire retenue est une version simplifiée du langage C. Elle permet de traiter les déclarations de variables, les affectations et l'opérateur ternaire.

$$
\begin{aligned}
Z &\rightarrow I\ \# \\
I &\rightarrow Declaration \mid Affectation \mid Ternaire \\
Declaration &\rightarrow Type\ Ident\ ; \\
Type &\rightarrow int \mid float \mid char \mid double \\
Affectation &\rightarrow Ident\ =\ Expr\ ; \\
Ternaire &\rightarrow Expr\ ?\ Expr\ :\ Expr\ ; \\
Expr &\rightarrow Expr\ Op\ Term \mid Term \\
Term &\rightarrow (Expr) \mid Ident \mid Const \\
Op &\rightarrow + \mid - \mid * \mid / \mid < \mid > \mid <= \mid >= \mid == \mid !=
\end{aligned}
$$

Explication

- **Z** : symbole de départ représentant le programme entier. Il se termine toujours par le symbole indiquant la fin du code.
- **I** : instruction, peut être soit une déclaration, une affectation ou une expression ternaire.
- **Declaration** : commence par un *Type* suivi d'un *Ident* (nom de variable) et se termine par un point-virgule.
- **Type** : mots-clés représentant les types primitifs (int, float, char, double).
- **Affectation** : identificateur suivi de '=' et d'une expression, se terminant par ';'.
- **Ternaire** : expression conditionnelle de la forme *Expr ? Expr : Expr ;*.
- **Expr** : expression arithmétique ou logique pouvant contenir des opérateurs binaires et des termes.
- **Term** : composant d'une expression : identificateur, constante ou expression parenthésée.

Analyseur lexical

L'analyse lexicale transforme le texte brut en tokens compréhensibles pour le parser. Elle est essentielle car elle simplifie l'analyse syntaxique et permet d'identifier des catégories lexicales comme : nombres, identificateurs, mots-clés, opérateurs, séparateurs et chaînes.

Automates utilisés

- **Automate des nombres (autnmbr)** :
  - État 0 : départ
  - État 1 : lecture d'un ou plusieurs chiffres
  - État 2 : lecture d'un point pour nombre flottant
  - État 3 : lecture de chiffres après le point
  - État 4 : état final accepteur
  - -1 : état d'erreur
- **Automate des identificateurs (autident)** :
  - État 0 : départ
  - État 1 : lecture de lettres ou chiffres
  - État 2 : état accepteur
  - -1 : état d'erreur

  Les identificateurs commencent par une lettre ou tiret et peuvent contenir lettres, chiffres et tiret.
- **Automate des opérateurs (autoper)** :
  - État 0 : départ
  - État 1 : lecture d'un opérateur simple
  - État 2 : lecture possible d'un opérateur double (ex : ==)
  - État 3 : état accepteur
  - -1 : état d'erreur

Fonctionnement de l'analyseur lexical

1. Ignore les espaces et commentaires.
2. Identifie le type de caractère pour déterminer l'automate à utiliser.
3. Passe par les états successifs de l'automate jusqu'à atteindre un état accepteur ou une erreur.
4. Retourne le token correspondant.

Analyseur syntaxique

L'analyse syntaxique vérifie que les tokens suivent la grammaire. Le parser descend récursif implémente chaque non-terminal comme une méthode :

- **Z()** : point d'entrée, vérifie la fin avec le symbole .
- **I()** : détecte la nature de l'instruction (déclaration, affectation ou ternaire).
- **D()** : gère les déclarations, vérifie que le type est suivi d'un identificateur et du point-virgule.
- **A()** : gère les affectations avec une expression après '='.
- **T()** : gère les expressions ternaires avec '?' et ':'.
- **E()** et **OpExpr()** : gèrent les expressions et la priorité des opérateurs.
- **Parens()** : gère les parenthèses pour les expressions imbriquées.

[language=Java, caption=Exemple d'utilisation de l'analyseur syntaxique] AnalyseurSyntaxique parser = new AnalyseurSyntaxique("int x = 5;"); parser.Z(); // Affiche "chaine acceptee"

Structure du projet

```
MiniCompil/
 src/
   minicompil/
     AnalyseurLexical.java
     AnalyseurSyntaxique.java
    Main.java
```

Cas de test

- `int x = 5;` (déclaration + affectation)
- `float y = 3.14;` (déclaration + affectation flottante)
- `x = y + 2;` (affectation avec expression)
- `z = (a<k) ?  b :  c;` (expression ternaire)
- `int a;` (déclaration simple)

Conclusion

Ce mini-compilateur en Java met en œuvre les étapes essentielles d'un processus de compilation : la décomposition du texte source en unités lexicales, puis l'analyse syntaxique basée sur une grammaire formelle et une stratégie de descente récursive. Le projet offre une base solide pour une extension future vers des phases plus avancées comme l'analyse sémantique, la table des symboles ou encore la génération de code intermédiaire.