Université A/ Mira de Béjaia

Département d'informatique

3ème année Licence académique

Projet TP
Mini-Compilateur en Java

Module : Compilation

Étudiant : Malek Leiticia

Langage cible : Java

Instruction analysée : WHILE

Année universitaire 2024-2025

Table des matières

1 La grammaire choisie

La grammaire définie pour ce mini-compilateur est **non récursive à gauche** (LL(1)) et adaptée à l'analyse par **descente récursive**.

1.1 Règles de production

1. Programme → Classe
2. Classe → `public class` IDENTIFICATEUR `{` Methode `}`
3. Methode → `public static void main ( String [ ] args )` Bloc
4. Bloc → `{` Instructions `}`
5. Instructions → Instruction Instructions | ε
6. Instruction → Declaration | Affectation | While | If
7. Declaration → Type IDENTIFICATEUR `=` Expression `;`
8. Type → `int` | `double` | `float` | `boolean` | `String`
9. Affectation → IDENTIFICATEUR OpAffect `;`
10. OpAffect → `=` Expression | `++` | `-`
11. While → `while (` Condition `)` Bloc **[ANALYSE DÉTAILLÉE]**
12. If → `if (` Condition `)` Bloc **[Reconnue mais ignorée]**
13. Condition → Expression OpComp Expression | Expression
14. OpComp → `==` | `!=` | `<` | `>` | `<=` | `>=` | `&&` | `||`
15. Expression → Terme Suite_Expression
16. Suite_Expression → `+` Terme Suite_Expression | `-` Terme Suite_Expression | ε
17. Terme → Facteur Suite_Terme
18. Suite_Terme → `*` Facteur Suite_Terme | `/` Facteur Suite_Terme | `%` Facteur Suite_Terme | ε
19. Facteur → NOMBRE_ENTIER | NOMBRE_DECIMAL | IDENTIFICATEUR | IDENTIFICATEUR `++` | IDENTIFICATEUR `-` | `(` Expression `)`

1.2 Symboles terminaux et non-terminaux

Symboles terminaux :

— Mots-clés : `public`, `class`, `static`, `void`, `main`, `String`, `int`, `double`, `float`, `boolean`, `while`, `if`, `Malek`, `Leiticia`

— Opérateurs : `+`, `-`, `*`, `/`, `%`, `=`, `==`, `!=`, `<`, `>`, `<=`, `>=`, `++`, `-`, `&&`, `||`

— Séparateurs : `(`, `)`, `{`, `}`, `[`, `]`, `;`, `,`

— IDENTIFICATEUR, NOMBRE_ENTIER, NOMBRE_DECIMAL

Symbole de départ : Programme

1.3 Propriétés de la grammaire

— **Non récursive à gauche :** Toutes les récursions sont à droite
— **LL(1) compatible :** Chaque règle peut être déterminée avec un lookahead de 1
— **Priorité des opérateurs :** Respectée par la structure de la grammaire

2 L'analyseur lexical

2.1 Principe de fonctionnement

L'analyseur lexical (ou *lexer*) est la première phase du compilateur. Son rôle est de :

1. Lire le code source caractère par caractère
2. Regrouper les caractères en **tokens** (unités lexicales)
3. Identifier le type de chaque token
4. Ignorer les espaces et commentaires
5. Détecter les erreurs lexicales

Chaque token contient :

— Un **type** (MOT_CLE, IDENTIFICATEUR, NOMBRE_ENTIER, etc.)
— Une **valeur** (le texte du token)
— Une **position** (ligne et colonne dans le fichier source)

2.2 Matrice de transition pour les identificateurs

Pour reconnaître les identificateurs, nous utilisons un **automate à états finis** représenté par une matrice de transition :

État	Lettre	Chiffre	Underscore	Autres
0 (initial)	1	-1	1	-1
1 (identificateur)	1	1	1	-1

TABLE 1 – Matrice de transition pour les identificateurs

Règle : Un identificateur doit commencer par une lettre ou un underscore (_), puis peut contenir des lettres, des chiffres ou des underscores.

Exemple de reconnaissance :
Pour le mot `"compteur"` :

1. Position 0 : 'c' (lettre) → État 0 → État 1
2. Position 1 : 'o' (lettre) → État 1 → État 1
3. Position 2 : 'm' (lettre) → État 1 → État 1
4. ... (même processus pour p, t, e, u, r)
5. Position 8 : ' ' (espace) → État 1 → État -1 (arrêt)

Résultat : Identificateur `"compteur"` reconnu.

2.3 Types de tokens reconnus

Type	Description	Exemples
MOT_CLE	Mots-clés Java + Malek, Leiticia	`public`, `while`, `Malek`
IDENTIFICATEUR	Noms de variables, méthodes	`compteur`, `main`
NOMBRE_ENTIER	Nombres entiers	`0`, `10`, `123`
NOMBRE_DECIMAL	Nombres décimaux	`3.14`, `0.5`
PLUS, MOINS, FOIS, etc.	Opérateurs arithmétiques	`+`, `-`, `*`
AFFECTATION	Opérateur d'affectation	`=`
INCREMENT, DECREMENT	Incrémentation/décrémentation	`++`, `-`
EGAL, DIFFERENT, etc.	Opérateurs de comparaison	`==`, `!=`, `<`
PAREN_OUVRANT, etc.	Séparateurs	`(`, `)`, `{`
POINT_VIRGULE	Ponctuation	`;`
CHAINE	Chaînes de caractères	`"Hello"`
EOF	Fin de fichier	`EOF`

Table 2 – Types de tokens reconnus

2.4 Gestion des erreurs lexicales

L'analyseur détecte les erreurs suivantes **sans arrêter l'analyse** :

— **Caractères invalides :** Caractères non reconnus dans le langage

— **Chaînes non fermées :** Guillemets ouvrants sans fermeture

— **Commentaires non fermés :** Commentaire bloc `/*` sans `*/`

Format des erreurs :

```
Erreur ligne X, colonne Y : [description de l'erreur]
```

2.5 Lecture du fichier source

L'analyseur lit le code source depuis un fichier `.java` grâce à la méthode `lireFichier()` qui utilise `BufferedReader` pour une lecture ligne par ligne efficace.

3 L'analyseur syntaxique

3.1 Méthode utilisée : Descente récursive

L'analyseur syntaxique vérifie que la structure du programme respecte la grammaire définie. Il utilise la méthode de **descente récursive** :

— **Une méthode Java par règle de grammaire**

— Chaque méthode reconnaît un non-terminal de la grammaire
— Navigation dans la liste de tokens avec un pointeur `i`
— Appels récursifs pour les dérivations

3.2 Structure principale

```java
private int i;                          // Position dans les tokens
private ArrayList<Token> tokens;        // Liste des tokens
private boolean error;                  // Indicateur d'erreur

// Methode principale
public void Z() {
    i = 0;
    error = false;
    Programme();  // Axiome de la grammaire

    // Verification finale
    if (tokenActuel().type.equals("EOF") && !error) {
        System.out.println("PROGRAMME ACCEPTE");
    }
}
```

3.3 Correspondance règles-méthodes

Règle de grammaire	**Méthode**	**Description**
Programme → Classe	`Programme()`	Point d'entrée
Classe → ...	`Classe()`	Structure de classe
Methode → ...	`Methode()`	Méthode main
Bloc → ...	`Bloc()`	Bloc d'instructions
Instructions → ...	`Instructions()`	Séquence d'instructions
Instruction → ...	`Instruction()`	Dispatch vers le type
Declaration → ...	`Declaration()`	Déclarations
Affectation → ...	`Affectation()`	Affectations
While → ...	`While()`	**ANALYSE DÉTAILLÉE**
If → ...	`IgnorerIf()`	Reconnu mais ignoré
Expression → ...	`Expression()`	Expressions arithmétiques
Suite_Expression → ...	`Suite_Expression()`	Récursion à droite
Terme → ...	`Terme()`	Termes multiplicatifs
Facteur → ...	`Facteur()`	Facteurs de base

TABLE 3 – Correspondance entre règles et méthodes

3.4 Analyse détaillée de l'instruction WHILE

L'instruction **WHILE** est la seule analysée en profondeur. La méthode `While()` vérifie successivement :

1. Le mot-clé `while`
2. La parenthèse ouvrante `(`
3. La condition (expression avec comparaison)
4. La parenthèse fermante `)`
5. Le bloc d'instructions `{ ... }`

Messages affichés lors de l'analyse :

```
=====================================
[WHILE] *** ANALYSE DETAILLEE DE WHILE ***
=====================================
[WHILE] Mot-cle 'while' reconnu
[WHILE] Parenthese ouvrante '(' trouvee
[WHILE] Analyse de la condition...
[WHILE] Condition analysee
[WHILE] Parenthese fermante ')' trouvee
[WHILE] Analyse du bloc d'instructions...
[WHILE] Bloc analyse avec succes
[WHILE] *** FIN DE L'ANALYSE DE WHILE ***
=====================================
```

3.5 Instructions reconnues mais ignorées

Les instructions **IF** et **FOR** sont reconnues par leur mot-clé mais leur structure interne n'est pas analysée en détail. Elles sont simplement "sautées" pendant l'analyse.

3.6 Gestion des erreurs syntaxiques

L'analyseur détecte les erreurs suivantes **sans arrêter l'analyse** :

— **Tokens manquants :** Parenthèses, accolades, points-virgules

— **Tokens inattendus :** Symboles non conformes à la grammaire

— **Structure incorrecte :** Ordre des éléments non respecté

Format des erreurs :

```
Erreur syntaxique ligne X, colonne Y : [description]
```

4 La structure du projet

4.1 Arborescence des fichiers

```
MiniCompilateur/
|
+-- src/
|   +-- minicompilateur/
|       +-- AnalyseurLexical.java
```

```
|       +-- AnalyseurSyntaxique.java
|
+-- test.java
|
+-- README.md
```

4.2 Organisation du code

4.2.1 AnalyseurLexical.java

- **Classe Token :** Représente un token avec type, valeur, ligne, colonne
- **Classe AnalyseurLexical :**
  - Attributs : `code`, `position`, `ligne`, `colonne`, `tokens`, `erreurs`
  - Matrice de transition pour identificateurs
  - Méthodes de reconnaissance : `reconnaitreIdentificateur()`, `reconnaitreNombre()`, etc.
  - Méthode principale : `analyser()`
  - Méthode de lecture : `lireFichier()`

4.2.2 AnalyseurSyntaxique.java

- **Classe AnalyseurSyntaxique :**
  - Attributs : `i`, `tokens`, `error`, `erreurs`
  - Méthode principale : `Z()`
  - Une méthode par règle de grammaire
  - Méthode `While()` pour analyse détaillée
  - Méthodes utilitaires : `verifierType()`, `verifierValeur()`, `consommer()`

4.3 Flux de compilation

5 Les cas de test

5.1 Test 1 : Programme correct

Fichier test.java :

```java
public class Test {
    public static void main(String[] args) {
        int compteur = 0;
        int x = 10;

        while (compteur < 5) {
            compteur++;
        }
    }
}
```

```
test.java (fichier source)
    |
    v
AnalyseurLexical.lireFichier()
    |
    v
AnalyseurLexical.analyser()
    |
    v
Liste de tokens
    |
    v
AnalyseurSyntaxique(tokens)
    |
    v
AnalyseurSyntaxique.Z()
    |
    v
Programme accepte ou rejete
```

FIGURE 1 – Flux de compilation

Résultat :

```
 Aucune erreur lexicale
 PROGRAMME ACCEPTÉ
Le programme est syntaxiquement correct.
```

5.2 Test 2 : Erreur lexicale - Caractère invalide

Fichier test.java :

```java
public class Test {
    public static void main(String[] args) {
        int x = 5;
        int @erreur = 10;
    }
}
```

Résultat :

```
Erreur ligne 4, colonne 13 : Caractere invalide '@'
 Compilation terminee avec erreurs
```

5.3 Test 3 : Erreur syntaxique - Parenthèse manquante

Fichier test.java :

```
public class Test {
    public static void main(String[] args) {
        int x = 0;

        while x < 5 {
            x++;
        }
    }
}
```

Résultat :

```
 Aucune erreur lexicale

Erreur syntaxique ligne 5, colonne 15 :
    Parenthese ouvrante '(' attendue
 PROGRAMME REJETÉ
```

5.4 Test 4 : Instruction IF (reconnue mais ignorée)

Fichier test.java :

```
public class Test {
    public static void main(String[] args) {
        int x = 10;

        if (x > 5) {
            x = 0;
        }

        while (x < 20) {
            x++;
        }
    }
}
```

Résultat :

```
[IF] Instruction IF reconnue (ignoree)

========================================
[WHILE] *** ANALYSE DETAILLEE DE WHILE ***
========================================
...
 PROGRAMME ACCEPTÉ
```

5.5 Test 5 : Mots-clés personnalisés

Fichier test.java :

```java
public class Test {
    public static void main(String[] args) {
        int Malek = 100;
        int Leiticia = 200;

        while (Malek < Leiticia) {
            Malek++;
        }
    }
}
```

Résultat :

```
MOT_CLE              | Malek           | L:3    | C:13
MOT_CLE              | Leiticia        | L:4    | C:13
...
 PROGRAMME ACCEPTÉ
```

Note : Malek et Leiticia sont reconnus comme mots-clés mais utilisés ici comme noms de variables.

6 Conclusion

Ce projet de mini-compilateur a permis de mettre en pratique les concepts fondamentaux de la compilation :

- Définition d'une **grammaire LL(1)** non récursive à gauche
- Implémentation d'un **analyseur lexical** avec matrice de transition
- Implémentation d'un **analyseur syntaxique** par descente récursive
- Analyse détaillée de l'instruction **WHILE**
- Gestion des **erreurs** lexicales et syntaxiques sans arrêt
- Reconnaissance des mots-clés personnalisés **Malek** et **Leiticia**

Le compilateur fonctionne correctement pour les programmes Java simplifiés respectant la grammaire définie. Les tests réalisés montrent que l'analyse lexicale et syntaxique sont efficaces et détectent correctement les erreurs.

Compétences acquises :

- Compréhension des phases de compilation
- Maîtrise des automates à états finis
- Pratique de la descente récursive
- Conception de grammaires formelles
- Programmation orientée objet en Java