Université A/Mira de Béjaia

Faculté des Sciences Exactes
Département d'Informatique

Projet TP

Mini-Compilateur en Java

Langage Cible : C

Module : Compilation
Niveau : 3ème année Licence Académique
Année Universitaire : 2024 – 2025

Réalisé par :
TEKERRABET ANIS

8 décembre 2025

Table des matières

1 Introduction

La compilation est une étape fondamentale en informatique permettant de traduire un programme écrit dans un langage de haut niveau vers une forme exploitable par la machine. Ce projet consiste à concevoir et implémenter un **mini-compilateur** pour un sous-ensemble du langage **C**, en utilisant le langage **Java**.

L'objectif principal est de mettre en œuvre les deux premières phases de la compilation :

— l'analyse lexicale ;

— l'analyse syntaxique par descente récursive.

Ce travail s'inscrit dans le cadre du module **Compilation** et permet d'appliquer les notions théoriques relatives aux automates, grammaires et analyseurs syntaxiques.

2 Grammaire du Mini-Compilateur

2.1 Grammaire Complète

La grammaire utilisée est définie comme suit :

```
(1) Programme -> #include < Directive > ListeFonctions MAIN
(2) ListeFonctions -> Fonction ListeFonctions | epsilon
(3) Fonction -> Type ID ( Parametres ) { Corps }
(4) Parametres -> Type ID AutresParam | epsilon
(5) AutresParam -> , Type ID AutresParam | epsilon
(6) MAIN -> Type main ( ) { Corps }
(7) Corps -> Instruction Corps | epsilon
(8) Instruction -> Declaration | Affectation | BoucleWhile |
   AppelFonction | Return
(9) Declaration -> Type ID SuiteDecl ;
(10) SuiteDecl -> , ID SuiteDecl | epsilon
(11) Affectation -> ID = Expression ;
(12) AppelFonction -> ID ( Arguments ) ;
(13) Arguments -> Expression AutresArgs | epsilon
(14) AutresArgs -> , Expression AutresArgs | epsilon
(15) BoucleWhile -> while ( Condition ) { Corps }
(16) Return -> return Expression ;
(17) Expression -> Terme Expression'
(18) Condition -> Expression OpComp Expression
(19) Type -> int | float | char | void
```

2.2 Propriétés de la Grammaire

Cette grammaire est conçue pour être compatible avec une analyse **LL(1)**, ce qui permet une implémentation simple à l'aide d'un analyseur syntaxique par descente récursive sans retour arrière.

3 Analyse Lexicale – Détails et Algorithme

3.1 Objectifs de l'analyse lexicale

L'analyse lexicale constitue la première phase du processus de compilation. Son rôle principal est de lire le code source caractère par caractère afin de le transformer en une suite de symboles abstraits appelés **tokens**. Chaque token représente une unité lexicale significative du langage.

Les objectifs principaux sont :

— Supprimer les caractères inutiles (espaces, retours à la ligne)

— Identifier les mots-clés du langage

— Reconnaître les identifiants et constantes numériques

— Détecter les erreurs lexicales le plus tôt possible

3.2 Algorithme général

L'algorithme de l'analyse lexicale suit les étapes suivantes :

1. Lecture du fichier source caractère par caractère
2. Regroupement des caractères en lexèmes
3. Vérification de chaque lexème à l'aide d'automates
4. Attribution d'un code token
5. Écriture de la séquence de tokens dans un fichier

3.3 Reconnaissance des identifiants

Un identifiant valide doit :

— Commencer par une lettre ou le caractère _

— Être suivi de lettres, chiffres ou _

— Ne pas être un mot-clé réservé

L'automate fini utilisé garantit le respect de ces règles.

3.4 Tokens Reconnu

Token	Code	Description
Type	T	int, float, char, void
Identifiant	I	Noms des variables et fonctions
Nombre	N	Constantes entières
main	M	Fonction principale
while	W	Boucle while
return	R	Instruction return
Opérateur	OP	+, -, *, /

3.5 Gestion des erreurs lexicales

Une erreur lexicale est détectée lorsque :

— Un caractère non autorisé est rencontré

— Un lexème ne correspond à aucun token connu

Dans ce cas, un message explicite est affiché indiquant le lexème fautif et sa position dans le fichier.

4 Analyse Syntaxique – Approfondissement

4.1 Rôle de l'analyse syntaxique

L'analyse syntaxique vérifie que la suite de tokens générée par l'analyse lexicale respecte la grammaire du langage. Elle permet de détecter les erreurs de structure telles que :

— Parenthèses manquantes

— Accolades non fermées

— Instructions incomplètes

4.2 Principe de la descente récursive

La descente récursive est une méthode d'analyse top-down adaptée aux grammaires LL(1). Chaque règle de grammaire est implémentée par une fonction Java.

Avantages :

— Simplicité d'implémentation

— Correspondance directe grammaire / code

— Facilité de débogage

4.3 Gestion du curseur de lecture

L'analyseur syntaxique utilise :

— Une liste de tokens

— Un index courant (`position`)

— Un token courant (`currentToken`)

La fonction `consommer()` assure la validation et l'avancement correct dans la séquence.

4.4 Stratégie de récupération d'erreurs

Afin de ne pas arrêter l'analyse à la première erreur, le compilateur :

— Signale l'erreur détectée

— Ignore certains tokens jusqu'à un séparateur

— Poursuit l'analyse globale

Cette approche permet de détecter plusieurs erreurs lors d'une seule exécution.

5 Exemples Supplémentaires

5.1 Expression arithmétique complexe

```
x = a + b * ( c - 2 ) / d ;
```

Cette expression permet de tester :

— La priorité des opérateurs

— Les parenthèses imbriquées

— Les affectations

5.2 Boucle `while` imbriquée

```
while ( x < 10 ) {
    while ( y > 0 ) {
        y = y - 1 ;
    }
    x = x + 1 ;
}
```

Ce test valide la gestion récursive des blocs d'instructions.

6 Conclusion Générale

La réalisation de ce mini-compilateur a permis de mettre en pratique les notions fondamentales étudiées dans le module de compilation. Le projet est fonctionnel, extensible et constitue une base claire pour des améliorations futures.

L'objectif pédagogique est pleinement atteint : comprendre avant d'optimiser.