Rapport du Projet - Mini-Compilateur en Java

Réalisée par: Ouaret Abdel Amine
Établissement: Université A/Mira de Béjaia
Module: Compilation - 3ème année Licence Informatique
Date: Décembre 2025

Table des matières

---

1. Introduction

Ce projet consiste en l'implémentation d'un mini-compilateur pour un sous-ensemble du langage C. Le compilateur effectue une analyse lexicale et syntaxique du code source, avec un focus particulier sur l'instruction **switch/case**.

Objectifs du projet

- Implémenter un analyseur lexical sans utiliser de bibliothèques externes (pas de regex)
- Implémenter un analyseur syntaxique par descente récursive
- Gérer les erreurs lexicales et syntaxiques sans arrêter l'analyse
- Créer un exécutable prêt à l'emploi

Langage cible

Langage C avec les constructions suivantes :

- Déclarations de variables (int, float, char, double, void)
- Affectations simples et composées (=, +=, -=, *=, /=)
- Expressions arithmétiques et logiques
- Comparaisons (==, !=, <, >, <=, >=)

- Incrémentation et décrémentation (++, --)
- **Structure de contrôle principale : switch/case**

---

2. Grammaire Choisie

Grammaire BNF simplifiée

bnf

```
Programme ::= {Declaration | Instruction}

Declaration ::= Type Identificateur [Affectation] ';'

Type ::= 'int' | 'float' | 'char' | 'double' | 'void'

Instruction ::= InstructionSimple
              | InstructionSwitch
              | Bloc

InstructionSimple ::= Affectation
                    | Incrementation
                    | Decrementation
                    | 'break' ';'
                    | 'continue' ';'
                    | 'return' [Expression] ';'

Affectation ::= Identificateur OperateurAffectation Expression ';'

OperateurAffectation ::= '=' | '+=' | '-=' | '*=' | '/='

Incrementation ::= Identificateur '++' ';' | '++' Identificateur ';'
```

Decrementation ::= Identificateur '--' ';' | '--' Identificateur ';'

InstructionSwitch ::= 'switch' '(' Expression ')' '{' {Case} [Default] '}'

Case ::= 'case' Constante ':' {Instruction}

Default ::= 'default' ':' {Instruction}

Bloc ::= '{' {Declaration | Instruction} '}'

Expression ::= ExpressionLogique

ExpressionLogique ::= ExpressionComparaison {('&&' | '||') ExpressionComparaison}

ExpressionComparaison ::= ExpressionArithmetique {('==' | '!=' | '<' | '>' | '<=' | '>=') ExpressionArithmetique}

ExpressionArithmetique ::= Terme {('+' | '-') Terme}

Terme ::= Facteur {('*' | '/' | '%') Facteur}

Facteur ::= Nombre
| Identificateur
| Caractere
| '(' Expression ')'
| '!' Facteur
| '-' Facteur

Constante ::= Nombre | Caractere | Identificateur

Particularités

- **Mots-clés personnalisés** : ouaret et abdelamine sont reconnus comme mots-clés
- **Commentaires** : Support des commentaires ligne (//) et bloc (/* */)
- **Reconnaissance complète** : L'analyseur lexical reconnaît tous les mots-clés C (if, else, while, for, etc.) même si seul switch/case est analysé syntaxiquement

---

3. Analyseur Lexical

Fonctionnement

L'analyseur lexical lit le code source caractère par caractère et produit une liste de tokens.

Types de tokens reconnus

1. **Mots-clés du langage C** :
    - Types : int, float, char, double, void
    - Structures : if, else, while, do, for, switch, case, default, break, continue, return
2. **Mots-clés personnalisés** :
    - ouaret / OUARET
    - abdelamine / ABDELAMINE
3. **Identificateurs** : Noms de variables et fonctions
4. **Littéraux** :
    - Nombres entiers : 42, 100
    - Nombres réels : 3.14, 19.99
    - Caractères : 'A', 'x'
    - Chaînes : "Hello"
5. **Opérateurs** :
    - Arithmétiques : +, -, *, /, %, ++, --
    - Comparaison : ==, !=, <, >, <=, >=
    - Logiques : &&, ||, !
    - Affectation : =, +=, -=, *=, /=

6. **Délimiteurs** : (,), {, }, [,], ;, ,, :

Gestion des commentaires

- **Commentaires ligne** : // ... jusqu'à la fin de la ligne
- **Commentaires bloc** : /* ... */ sur plusieurs lignes

Les commentaires sont ignorés et ne génèrent pas de tokens.

Gestion des erreurs lexicales

L'analyseur détecte et signale :

- Caractères non reconnus
- Caractères non terminés ('A au lieu de 'A')
- Chaînes non terminées ("Hello sans guillemet fermant)

Important : L'analyse continue après une erreur pour détecter toutes les erreurs du fichier.

Exemple de tokenisation

Code source :

c

```
int x = 10;
x++;
```

Tokens produits :

```
INT            'int'          [L1:C1]
IDENTIFICATEUR  'x'            [L1:C5]
AFFECTATION     '='            [L1:C7]
NOMBRE_ENTIER   '10'           [L1:C9]
POINT_VIRGULE   ';'            [L1:C11]
IDENTIFICATEUR  'x'            [L2:C1]
INCREMENT       '++'           [L2:C2]
POINT_VIRGULE   ';'            [L2:C4]
```

---

4. Analyseur Syntaxique

Méthode d'analyse

Descente récursive : Chaque règle de grammaire correspond à une méthode qui analyse cette construction.

Principe de fonctionnement

1. **Parcours des tokens** : L'analyseur parcourt la liste de tokens produite par l'analyseur lexical
2. **Vérification de la structure** : Chaque règle de grammaire est vérifiée
3. **Gestion des erreurs** : Les erreurs sont collectées mais l'analyse continue

Méthodes principales

- programme() : Point d'entrée, analyse l'ensemble du programme
- declaration() : Vérifie les déclarations de variables
- instruction() : Analyse les instructions
- instructionSwitch() : **Analyse principale** de switch/case
- instructionCase() : Analyse d'un case
- instructionDefault() : Analyse du default
- expression() : Analyse des expressions
- expressionLogique() : Gère les opérateurs && et ||
- expressionComparaison() : Gère les comparaisons
- expressionArithmetique() : Gère + et -
- terme() : Gère *, /, %
- facteur() : Éléments de base (nombres, identificateurs, parenthèses)

Gestion des erreurs syntaxiques

Lorsqu'une erreur est détectée :

1. **Message d'erreur** : Affichage de l'erreur avec ligne et colonne
2. **Récupération** : L'analyseur avance jusqu'au prochain point-virgule ou accolade
3. **Continuation** : L'analyse continue pour détecter d'autres erreurs

Exemple d'analyse

Code avec erreur :

c

int x

x = 10;

Erreur détectée :

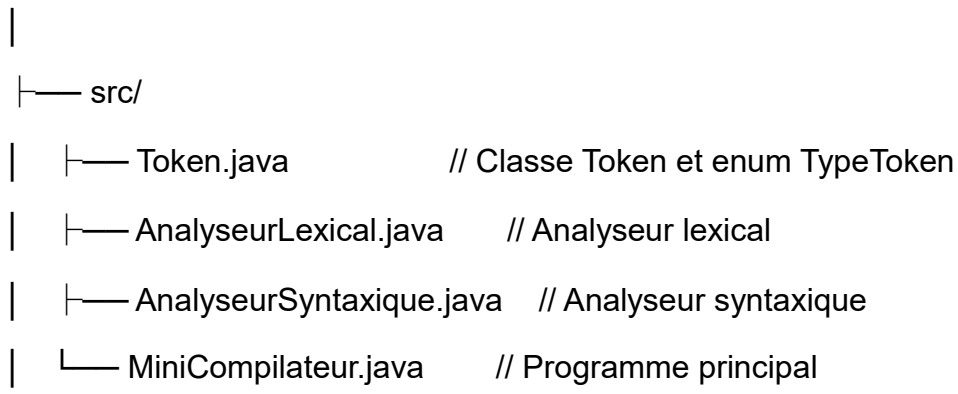
Erreur syntaxique ligne 1, col 6: Point-virgule attendu après la déclaration (token actuel: x)

---

5. Structure du Projet

Arborescence des fichiers

MiniCompilateur/



```
├── src/
│   ├── Token.java                // Classe Token et enum TypeToken
│   ├── AnalyseurLexical.java     // Analyseur lexical
│   ├── AnalyseurSyntaxique.java  // Analyseur syntaxique
│   └── MiniCompilateur.java      // Programme principal
```

Description des classes

Token.java

- **Classe Token** : Représente un token avec son type, sa valeur, sa ligne et sa colonne
- **Enum TypeToken** : Énumération de tous les types de tokens possibles

AnalyseurLexical.java

- Lit le code source caractère par caractère
- Produit une liste de tokens
- Gère les commentaires et les espaces
- Détecte les erreurs lexicales

AnalyseurSyntaxique.java

- Parcourt la liste de tokens
- Vérifie la conformité syntaxique
- Implémente les règles de grammaire par descente récursive

- Gère les erreurs syntaxiques avec récupération

MiniCompilateur.java

- Programme principal
- Lecture du fichier source
- Orchestration des analyses lexicale et syntaxique
- Affichage des résultats et erreurs

---

6. Cas de Test

Test 1 : Programme correct (test1.c)

Description : Programme utilisant switch/case avec toutes les fonctionnalités

Contenu :

- Déclarations de variables (int, float, char)
- Switch/case avec 3 cases et un default
- Expressions arithmétiques
- Incrémentation/décrémentation
- Expressions logiques
- Break dans chaque case
- Utilisation des mots-clés personnalisés (ouaret, abdelamine)

Résultat attendu : □ Aucune erreur

Test 2 : Erreurs lexicales (test2.c)

Description : Programme contenant des erreurs lexicales intentionnelles

Erreurs :

- Caractère non terminé : 'A;
- Chaîne non terminée : "Bonjour ouaret;
- Caractère invalide : @

Résultat attendu : 3 erreurs lexicales détectées

Test 3 : Erreurs syntaxiques (test3.c)

Description : Programme contenant des erreurs syntaxiques

Erreurs :

- Point-virgule manquant après déclaration
- Switch sans parenthèses autour de l'expression
- Deux-points manquant après case
- Accolade fermante manquante

Résultat attendu : 4 erreurs syntaxiques détectées

Test 4 : Test complet (test4.c)

Description : Programme démontrant toutes les fonctionnalités

Fonctionnalités testées :

- Fonction avec paramètres
- Switch/case imbriqué dans une fonction
- Affectations composées (+=, -=, *=)
- Expressions complexes
- Blocs imbriqués dans les cases
- Mots-clés personnalisés comme identificateurs

Résultat attendu : Aucune erreur

---

7. Conclusion

Objectifs atteints

 Analyseur lexical : Reconnaissance complète de tous les tokens sans utiliser de fonctions prédéfinies
 Analyseur syntaxique : Analyse par descente récursive avec focus sur switch/case
 Gestion des erreurs : Détection et affichage de multiples erreurs sans arrêt
 Mots-clés personnalisés : Reconnaissance de "ouaret" et "abdelamine"
 Exécutable : Programme prêt à l'emploi avec interface claire

Points forts

- **Aucune bibliothèque externe** : Tout le code est écrit manuellement
- **Gestion robuste des erreurs** : Continuation de l'analyse après chaque erreur
- **Interface utilisateur claire** : Affichage structuré et coloré des résultats
- **Code modulaire** : Séparation claire des responsabilités

- **Tests complets** : Couverture de tous les cas (succès et erreurs)

Améliorations possibles

- Ajout d'une table des symboles pour vérifier les déclarations
- Analyse sémantique (types, portée des variables)
- Génération de code intermédiaire
- Optimisation des performances pour les gros fichiers
- Support de structures de données (tableaux, structures)

Difficultés rencontrées

1. **Gestion des commentaires** : Nécessité de gérer les compteurs de lignes et colonnes correctement
2. **Récupération d'erreurs** : Trouver le bon point de reprise après une erreur syntaxique
3. **Tests exhaustifs** : Création de cas de test couvrant toutes les situations

---

Annexes

Références

- Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools* (Dragon Book)
- Niklaus Wirth. *Compiler Construction*
- Documentation officielle du langage C

Auteur

Nom : OUARET
Prénom : Abdelamine
Établissement : Université A/Mira de Béjaia
Département : Informatique
Année : 3ème année Licence académique
Module : Compilation

Fin du rapport