Projet de Compilation

Réalisation d'un Analyseur Lexical et Syntaxique
pour les boucles Do-While

Réalisé par :
ZENATI SABRINE

groupe :
B4

Année Universitaire 2024/2025

1 Introduction

Dans le cadre du module de compilation, nous avons réalisé un mini-compilateur capable d'analyser des programmes contenant des boucles do-while. Ce projet nous a permis de comprendre concrètement les différentes phases de la compilation, notamment l'analyse lexicale et l'analyse syntaxique.

L'objectif principal de ce travail est de développer un outil qui peut :

— Décomposer un code source en unités lexicales (tokens)
— Vérifier que la structure du programme respecte une grammaire définie
— Détecter et signaler les erreurs de manière claire
— Continuer l'analyse même en présence d'erreurs

Ce rapport présente l'architecture de notre compilateur, les choix techniques que nous avons faits, et les difficultés que nous avons rencontrées durant le développement.

2 Présentation du Projet

2.1 Contexte et Objectifs

Ce projet consiste à implémenter un compilateur simplifié en Java qui traite uniquement les structures do-while. Le compilateur doit effectuer deux phases principales :

1. **Analyse Lexicale** : Transformer le code source en une suite de tokens identifiables
2. **Analyse Syntaxique** : Vérifier que ces tokens respectent la grammaire définie

2.2 Grammaire Utilisée

Notre compilateur se base sur la grammaire formelle suivante :

```
 Z -> S #
 S -> do { Instructions } while ( E ) ;
Instructions -> Instruction Instructions | Instruction
 Instruction -> id = Expression ; | S
 E -> id O id
 O -> < | > | == | != | <= | >=
 Expression -> Terme OpArith Terme | Terme
 Terme -> id | nombre
 OpArith -> + | - | * | /
```

Cette grammaire décrit la structure d'un programme contenant des boucles do-while potentiellement imbriquées. Le symbole # marque la fin du programme.

2.3 Exemple de Code Valide

Voici un exemple de programme accepté par notre compilateur :

```
do {
    x = y;
    compteur = compteur + un;
    do {
        temp = x;
```

```
    } while ( temp < max ) ;
} while ( x < z ) ;
```

3 Architecture du Compilateur

Notre compilateur est composé de trois modules principaux :

3.1 Module d'Analyse Lexicale

Le module `AnalyseurLexical.java` est responsable de la première phase de compilation. Il utilise un automate à états finis représenté par une table de transition pour reconnaître les différents types de lexèmes.

3.1.1 Table de Transition

Nous avons implémenté une table de transition à 7 états :

État	Lettre	Chiffre	=	<	>	!	+/-/*//	Sép	Spé
0	1	2	4	4	4	4	4	5	6
1	1	1	-1	-1	-1	-1	-1	-1	-1
2	-1	2	4	-1	-1	-1	-1	-1	-1
3	-1	3	-1	-1	-1	-1	-1	-1	-1
4	-1	-1	-1	-1	-1	-1	-1	-1	-1
5	-1	-1	-1	-1	-1	-1	-1	-1	-1
6	-1	-1	-1	-1	-1	-1	-1	-1	-1

Chaque état correspond à un type de lexème :
— État 1 : Identificateurs
— État 2 : Nombres
— État 4 : Opérateurs
— État 5 : Séparateurs
— État 6 : Caractères spéciaux

3.1.2 Catégories de Lexèmes

L'analyseur lexical reconnaît les catégories suivantes :
— **Mots-clés** : if, else, while, do, begin, end, int, float,for,foreach
— **Mots-clés personnalisés** : Votre nom (configurable)
— **Opérateurs** : =, +, -, *, /, <, >, <=, >=, ==, !=
— **Séparateurs** : , ; ( ) { }
— **Caractères spéciaux** : # $ @ &
— **Identificateurs** : variables commençant par une lettre
— **Nombres** : entiers et flottants

3.1.3 Gestion des Erreurs Lexicales

Une des améliorations importantes que nous avons apportées est la gestion des erreurs. Contrairement à un compilateur classique qui s'arrête à la première erreur, notre analyseur :

— Détecte les caractères invalides
— Affiche un message d'erreur avec la position (ligne et colonne)
— Continue l'analyse pour détecter toutes les erreurs
— Affiche un résumé à la fin
Exemple de message d'erreur :

```
ERREUR LEXICALE [Ligne 2, Col 15]: Caractère invalide '°'
```

3.2 Module d'Analyse Syntaxique

Le module `AnalyseurSyntaxiqueDoWhile.java` vérifie que la suite de tokens respecte la grammaire définie. Nous avons utilisé la méthode de l'analyse descendante récursive.

3.2.1 Principe de l'Analyse Descendante

Chaque règle de la grammaire est implémentée par une méthode Java :
— `Z()` : Vérifie l'axiome de départ
— `S()` : Reconnaît une structure do-while
— `Instructions()` : Gère une suite d'instructions
— `Instruction()` : Traite une instruction simple ou imbriquée
— `E()` : Vérifie les conditions while
— `Expression()` : Analyse les expressions d'affectation
— `O()` : Reconnaît les opérateurs de comparaison
— `Terme` :operation id et nombre ou id et id ou nombre et nombre
— `OpArith` :les operateur a utiliser

3.2.2 Détection des Erreurs Syntaxiques

Notre analyseur syntaxique fournit des messages d'erreur précis qui indiquent :
— La position exacte de l'erreur
— Ce qui était attendu
— Ce qui a été trouvé
Exemple :

```
ERREUR SYNTAXIQUE [Position 5]: ')' attendu pour fermer
la condition, trouve ';'
```

3.3 Programme Principal

Le fichier `Main.java` coordonne l'exécution des deux phases d'analyse. Il :

1. Affiche une bannière avec les informations du compilateur
2. Lit le code source depuis l'entrée standard
3. Lance l'analyse lexicale
4. Si aucune erreur lexicale, lance l'analyse syntaxique
5. Affiche un résumé complet de la compilation

4 Implémentation

4.1 Choix Techniques

4.1.1 Langage de Programmation

Nous avons choisi Java pour plusieurs raisons :
— Langage orienté objet facilitant la modularisation
— Manipulation simple des chaînes de caractères
— Portabilité (code exécutable sur n'importe quel système avec JVM)
— Collections Java (ArrayList) pratiques pour gérer les listes de tokens

4.1.2 Structures de Données

— **Tableaux** : Pour stocker les mots-clés, opérateurs, etc.
— **Table de transition** : Matrice d'entiers pour l'automate
— **ArrayList** : Pour les listes dynamiques de tokens et d'erreurs
— **String[]** : Pour le flux de tokens dans l'analyseur syntaxique

4.2 Algorithmes Clés

4.2.1 Algorithme de Tokenisation

L'algorithme principal de l'analyseur lexical :

1. Parcourir le code source caractère par caractère
2. Déterminer la colonne dans la table de transition
3. Suivre les transitions d'états
4. Quand un état final est atteint, sauvegarder le token
5. Catégoriser et afficher immédiatement le token
6. Continuer jusqu'à la fin du code

4.2.2 Algorithme d'Analyse Syntaxique

1. Appeler la méthode correspondant à l'axiome
2. Pour chaque règle, vérifier les tokens attendus
3. En cas d'échec, essayer les alternatives (backtracking)
4. Enregistrer les erreurs sans interrompre l'analyse
5. Retourner vrai si la dérivation réussit

4.3 Gestion des Erreurs

Nous avons mis l'accent sur une gestion robuste des erreurs :

4.3.1 Erreurs Lexicales

— Caractères non reconnus par l'automate
— Tokens mal formés
— Position précise (ligne, colonne)

4.3.2 Erreurs Syntaxiques

- Structure ne correspondant pas à la grammaire
- Tokens manquants
- Tokens inattendus
- Position dans le flux de tokens

5 Tests et Validation

5.1 Cas de Test Valides

5.1.1 Test 1 : Do-While Simple

```
do {
    x = y;
} while ( x < z ) ;
```

Résultat : Analyse réussie

5.1.2 Test 2 : Do-While Imbriqués

```
do {
    a = b;
    do {
        c = d;
    } while ( c < e ) ;
} while ( a < f ) ;
```

Résultat : Analyse réussie

5.1.3 Test 3 : Instructions Multiples

```
do {
    x = y;
    a = b;
    c = d + e;
} while ( x != z ) ;
```

Résultat : Analyse réussie

5.2 Cas de Test Invalides

5.2.1 Test 4 : Erreur Lexicale

```
do {
    x = y@;
} while ( x < z ) ;
```

Résultat :

```
ERREUR LEXICALE [Ligne 2, Col 9]: Token invalide 'y@'
```

5.2.2 Test 5 : Parenthèse Manquante

```
do {
    x = y;
} while x < z ) ;
```

Résultat :

```
ERREUR SYNTAXIQUE [Position 6]: '(' attendu apres
'while', trouve 'x'
```

5.3 Tableau Récapitulatif des Tests

Test	Description	Résultat
1	Do-while simple	Valide
2	Do-while imbriqués	Valide
3	Instructions multiples	Valide
4	Caractère invalide	Erreur détectée
5	Parenthèse manquante	Erreur détectée

6 Conclusion

Ce projet m'a permis de comprendre concrètement le fonctionnement interne d'un compilateur. J'ai réalisé l'importance de chaque phase de compilation et la complexité de la gestion des erreurs.

Les points clés que j'ai retenus :

1. L'analyse lexicale transforme un texte brut en tokens exploitables
2. L'analyse syntaxique vérifie la structure selon une grammaire formelle
3. Une bonne gestion des erreurs améliore considérablement l'expérience utilisateur
4. Le backtracking est essentiel pour gérer les grammaires ambiguës
5. La modularité du code facilite les modifications et extensions

Ce travail m'a également appris à structurer un projet de programmation de manière professionnelle, avec une architecture claire et une documentation complète.

Annexes

A. Compilation et Exécution

Pour compiler le projet :

```
javac TestCompilateur.java AnalyseurLexical.java
      AnalyseurSyntaxiqueDoWhile.java
```

Pour exécuter :

```
java TestCompilateur
```

B. Création d'un Exécutable

Pour créer un fichier JAR exécutable :

```
# Creer le manifest
echo "Main-Class: TestCompilateur" > manifest.txt

# Creer le JAR
jar cvfm TestCompilateur.jar manifest.txt *.class

# Executer
java -jar TestCompilateur.jar
```

C. Code Source Complet

Le code source complet est disponible dans les fichiers :

— `AnalyseurLexical.java`
— `AnalyseurSyntaxiqueDoWhile.java`
— `TestCompilateur.java`