Rapport de Projet : Mini-Compilateur do-while en C

Lebsir Mohamed Ali - B1

Université A. Mira de Béjaïa

1 Introduction

Ce projet implémente un mini-compilateur en Java qui analyse la structure `do-while` dans des fichiers C. Il effectue deux phases principales :

- **Analyse lexicale** : Lecture caractère par caractère du fichier source et transformation en tokens
- **Analyse syntaxique** : Vérification de la syntaxe des structures do-while selon une grammaire définie

2 Lecture du fichier source

L'analyse commence par la lecture caractère par caractère du fichier C source :

```
public static List<Token> analyser(String chemin) {
    List<Token> tokens = new ArrayList<>();

    try (BufferedReader br = new BufferedReader(new FileReader(chemin))
       ) {
        int car;
        String lexeme = "";
        int ligne = 1;
        int colonne = 1;

        // Lecture caractere par caractere
        while ((car = br.read()) != -1) {
            char c = (char) car;

            // Gestion des lignes/colonnes
            if (c == '\n') {
                ligne++;
                colonne = 1;
            } else {
                colonne++;
            }

            // Construction des lexemes caractere par caractere
            // ... logique de regroupement des caracteres en lexemes

            // Quand un lexeme est complet, on le transforme en token
            if (condition_de_fin_lexeme) {
                tokens.add(creerToken(lexeme, colonne, ligne));
                lexeme = "";  // Reinitialisation pour le prochain
                   lexeme
```

```
            } else {
                lexeme = lexeme + c;  // Ajout du caractere au lexeme
                    courant
            }
        }
    } catch (IOException e) {
        System.out.println("Erreur lecture fichier : " + e.getMessage()
            );
    }

    return tokens;
}
```

3 Grammaire utilisée

La grammaire formelle pour l'analyse syntaxique est définie comme suit :

$$\begin{aligned} \text{DO\_WHILE} &\rightarrow \text{"do" BLOC "while" "(" CONDITION ")" ";"} \\ \text{BLOC} &\rightarrow \text{"" INSTRUCTIONS ""} \\ \text{INSTRUCTIONS} &\rightarrow (\text{instruction})^* \\ \text{CONDITION} &\rightarrow (\text{expression})^* \end{aligned}$$

4 Analyseur Lexical

4.1 Processus de lecture et regroupement

Le processus lexical suit ces étapes :

1. **Lecture caractère par caractère** du fichier source
2. **Regroupement** des caractères en lexèmes selon les règles :
   — Les espaces/tabulations séparent les lexèmes
   — Les chaînes (`"..."`) et caractères (`'x'`) sont traités comme unités complètes
   — Les commentaires sont ignorés
3. **Classification** de chaque lexème selon son type

4.2 Méthodes de reconnaissance

1. Tableaux statiques pour les mots-clés, opérateurs et séparateurs :

```
// Tableau des mots-cles
private static final String[] MOTS_CLES = {
    "main", "do", "while", "if", "int", "char", "return"
};

// Tableau des operateurs
private static final String[] OPERATEURS = {
    "+", "-", "*", "/", "=", "==", "!=", ">", "<", ">=", "<="
};
```

```
10
11 // Tableau des separateurs
12 private static final String[] SEPARATEURS = {
13     "(", ")", "{", "}", ";", ",", ":", ".", "#"
14 };
```

2. Automate à états finis pour les identificateurs :

```
1  public static boolean estIdentificateur(String lex) {
2      int[][] matrice = {
3          {1, 1, -1, -1},  // Etat 0
4          {1, 1, 1, -1}    // Etat 1 (etat final)
5      };
6
7      int etat = 0;
8      for (int i = 0; i < lex.length(); i++) {
9          int colonne = col(lex.charAt(i));
10         if (colonne == -1 || matrice[etat][colonne] == -1)
11             return false;
12         etat = matrice[etat][colonne];
13     }
14     return etat == 1;  // Accepte si dans l'etat final
15 }
```

Fonction de transition :

$$\text{col}(c) = \begin{cases} 0 & \text{si } c = \_ \\ 1 & \text{si } c \text{ est une lettre} \\ 2 & \text{si } c \text{ est un chiffre} \\ 3 & \text{sinon (erreur)} \end{cases}$$

4.3 Fonctionnalités

- Lecture caractère par caractère avec `BufferedReader`
- Ignore les commentaires (`//` et `/* */`)
- Ignore les directives `#include`
- Détecte les chaînes (`"..."`) et caractères (`'x'`)
- Classifie les tokens selon leur type
- Signale les erreurs lexicales (identificateurs commençant par un chiffre, etc.)

5 Analyseur Syntaxique

5.1 Approche par descente récursive

L'analyseur syntaxique suit la grammaire définie précédemment. Chaque règle de la grammaire est implémentée comme une méthode récursive :

```
1 // Methode pour la regle DO_WHILE
2 private void S() {
3     if (verifierToken("do", "MOT_CLE")) {
```

```
        avancer();
        if (verifierToken("{", "SEPARATEUR")) {
            avancer();
            inst();  // Ignorer instructions
            if (verifierToken("}", "SEPARATEUR")) {
                avancer();
                if (verifierToken("while", "MOT_CLE")) {
                    avancer();
                    if (verifierToken("(", "SEPARATEUR")) {
                        avancer();
                        cond();  // Ignorer condition
                        if (verifierToken(")", "SEPARATEUR")) {
                            avancer();
                            if (verifierToken(";", "SEPARATEUR")) {
                                avancer();
                                System.out.println("Correct");
                            }
                        }
                    }
                }
            }
        }
    }
}
```

5.2 Erreurs syntaxiques détectées

1. `do` manquant
2. `{` manquant après `do`
3. `}` manquant
4. `while` manquant
5. `(` ou `)` manquant
6. `;` manquant à la fin

6 Structure du Projet

6.1 Arborescence des fichiers

```
src/
 LexicalUtils.java      # Tableaux + automate pour identificateurs
 Token.java             # Structure d'un token
 AnalyseurLexical.java # Lecture caractere par caractere + analyse
 AnalyseurSyntaxiqueDoWhile.java # Analyse syntaxique (descente recursive)
 MainCompilateurComplet.java     # Programme principal
```

6.2 Description des composants

- **LexicalUtils.java** : Contient les tableaux statiques (mots-clés, opérateurs, séparateurs) et l'automate pour la reconnaissance des identificateurs
- **Token.java** : Représente un token avec son lexème, type, ligne et colonne
- **AnalyseurLexical.java** : Lit le fichier caractère par caractère, regroupe en lexèmes, utilise les tableaux et l'automate pour transformer le code en tokens
- **AnalyseurSyntaxiqueDoWhile.java** : Implémente la descente récursive selon la grammaire
- **MainCompilateurComplet.java** : Orchestre l'ensemble du processus

7 Cas de Test

7.1 Test 1 : Structure correcte

```
do {
    printf("Test");
} while (x > 0);
```

Résultat : Compilation réussie

7.2 Test 2 : Erreur syntaxique

```
do {
    x = 5;
} while (x > 0)    // ERREUR: ';' manquant
```

Résultat : ";" manquant à la fin

7.3 Test 3 : Erreur lexicale

```
int 123var = 10;  // ERREUR: identificateur commence par chiffre
```

Résultat : Erreur lexicale (l'automate rejette)

7.4 Test 4 : Do-while imbriqué

```
do {
    do {
        printf("Imbriqu ");
    } while (false);
} while (true);
```

Résultat : 2 structures do-while correctes

8 Résumé des techniques utilisées

8.1 Analyseur Lexical

— **Lecture caractère par caractère** : Avec `BufferedReader` et `FileReader`
— **Tableaux statiques** : Pour les mots-clés, opérateurs et séparateurs
— **Automate à états finis** : Pour la reconnaissance des identificateurs
— **Regroupement séquentiel** : Des caractères en lexèmes

8.2 Analyseur Syntaxique

— **Grammaire** : Pour définir la structure syntaxique
— **Descente récursive** : Pour implémenter la grammaire
— **Vérification séquentielle** : Des éléments de la structure
— **Ignorance intelligente** : Du contenu des blocs et conditions

9 Conclusion

9.1 Réalisations

— Lecture caractère par caractère du fichier source
— Analyse lexicale avec automate pour identificateurs
— Analyse syntaxique par descente récursive selon grammaire BNF
— Gestion claire des erreurs lexicales et syntaxiques
— Tests complets pour valider le fonctionnement

9.2 Points techniques clés

— Utilisation de `BufferedReader` pour la lecture caractère par caractère
— Automate pour la reconnaissance des identificateurs
— Tableaux statiques pour mots-clés, opérateurs, séparateurs
— Grammaire pour la définition syntaxique
— Implémentation par descente récursive

9.3 Perspectives

— Étendre l'automate pour d'autres types de tokens
— Ajouter plus de structures syntaxiques
— Implémenter une table des symboles