Université Abderrahmane Mira

Faculté des Sciences Exactes — Département d'Informatique

Rapport

Mini-Compilateur JavaScript

Étudiante :
Tidjet Mailiz

Groupe :
B4

Année universitaire : 2025/2026

Table des matières

1 Introduction

1.1 Contexte du Projet

Dans le cadre du module de compilation, nous avons développé un mini-compilateur capable d'analyser un sous-ensemble d'un langage inspiré de JavaScript. Ce compilateur comporte deux parties principales :

— **Analyse lexicale** : transformation du texte en tokens
— **Analyse syntaxique** : vérification de la conformité à une grammaire

Le projet est entièrement développé en Java.

1.2 Objectifs

Les objectifs principaux sont :

— détecter les unités lexicales : identifiants, mots-clés, chaînes, opérateurs...
— vérifier la syntaxe selon une grammaire récursive
— prendre en charge fonctions, blocs, conditions, déclarations, opérations, etc.
— gérer les erreurs lexicales et syntaxiques

1.3 Outils

— Langage : Java
— Analyse lexicale : automate + matrice de transitions
— Analyse syntaxique : descente récursive
— IDE : NetBeans / IntelliJ / Eclipse

2 Analyse Lexicale

2.1 Définition

L'analyse lexicale lit le code source caractère par caractère et le découpe en **tokens**. Chaque token contient :

— **un type** : ID, NOMBRE, OPERATEUR...
— **une valeur**
— **la ligne**

Exemple :

```
let x = 7;
x++;
```

Tokens produits :

— let → MOT_CLE
— x → ID
— = → AFFECTATION
— 7 → NOMBRE
— ; → SEMI
— ++ → INCREMENTATION

2.2 Structure d'un Token

```
public class Token {
    String type;
    String value;
    int line;

    public Token(String type, String value, int line) {
        this.type = type;
        this.value = value;
        this.line = line;
    }
}
```

2.3 Éléments Reconnaissables

- **Mots-clés** : function, let, var, if, else...
- **Opérateurs** : +, -, *, /, ==, >=, , ||, ++...
- **Séparateurs** : `() {} [] , ;`
- **Identificateurs** : commencent par lettre ou _
- **Nombres** : entiers, décimaux, hexadécimaux, binaires
- **Chaînes** : "texte", 'texte'

3 Analyse Syntaxique

3.1 Définition

L'analyse syntaxique vérifie que la suite de tokens forme un programme valide selon la grammaire. Chaque méthode du parseur correspond à une règle.

3.2 Grammaire du Projet

```
<Z> ::= <S> EOF
<S> ::= "function" ID "(" <entre> ")" <block>
      | <instructions>

<instructions> ::= <instruction> ";" <instructions>
                 | "if" "(" <valeur> ")" <block> <else_opt>
                 |

<else_opt> ::= "else" "if" "(" <valeur> ")" <block> <else_opt>
             | "else" <block>
             |

<block> ::= "{" <instructions> "}"

<instruction> ::= <declaration>
                | ID <CalcApl>

<declaration> ::= ("let"|"var"|"const") ID <D>
```

```
<D> ::= "=" <valeur> |

<CalcApl> ::= INCREMENTATION
            | A
            | "=" <valeur>

<A> ::= "." ID <A>
      | "(" <entre> ")"
      |

<entre> ::= <valeur> <e> |
<e> ::= "," <valeur> <e> |

<valeur> ::= NOMBRE <V>
           | ID <V>
           | STRING <V>
           | "(" <valeur> <V> ")"

<V> ::= OPERATEUR <valeur> <V>
      | COMPARATEUR <valeur> <V>
      |
```

4 Exemple de Programme Accepté

```
function calc(a, b) {
    let x = a + b;
    x++;
    if (x > 10) {
        y = "grand";
    } else {
        y = "petit";
    }
}
```

5 Conclusion

Ce projet nous a permis de :
— maîtriser l'analyse lexicale
— construire un parseur récursif
— comprendre la structure d'un compilateur
— gérer les erreurs syntaxiques
— implémenter un mini-langage proche de JavaScript

Le compilateur pourra être étendu dans le futur (boucles, retours de fonction, analyse sémantique...).