République Algérienne Démocratique et Populaire
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

Université A/ Mira de Béjaia

Faculté des Sciences Exactes
Département d'Informatique

Module : Compilation (L3)

Mini-Compilateur JavaScript Instruction : forEach

Réalisé par :
OUARET RAYEL
Groupe : B3

Encadré par :
Mme TASSOULT NADIA

Année Universitaire 2024 – 2025

Table des matières

Introduction

Ce projet a été réalisé dans le cadre du module de Compilation de la 3ème année Licence. L'objectif est de concevoir et d'implémenter un mini-compilateur pour un sous-ensemble du langage JavaScript.

La tâche principale consiste à réaliser les phases d'analyse lexicale et syntaxique. Le sujet spécifique qui m'a été attribué concerne l'analyse de l'instruction `forEach`, en intégrant une contrainte : la reconnaissance de mon nom **RAYEL** comme mot-clé.

Ce rapport détaille la grammaire formelle définie, l'architecture de l'automate lexical, la méthode de descente récursive pour l'analyse syntaxique, ainsi que les tests de validation.

Chapitre 1

Grammaire du Langage

La définition d'une grammaire formelle est l'étape fondamentale avant le développement du parser. Elle définit les règles exactes que le code source doit respecter.

J'ai défini une grammaire **Non-Contextuelle (Context-Free Grammar)**, adaptée à une analyse syntaxique descendante ($LL(1)$).

1.1 Définition des Symboles

- **Symboles Terminaux** (Les mots du code) :
  `var`, `let`, `function`, `RAYEL`, `forEach`, `=`, `;`, `.`, `(`, `)`, `{`, `}`, **identifiant**, **valeur**.
- **Symboles Non-Terminaux** (Les règles) :
  Z (Axiome), S (Programme), $Instruction$, $DeclVar$, $DeclFonction$, $ForEach$, $Affectation$, $BlocIgnore$.

1.2 Règles de Production (BNF)

Voici l'ensemble des règles de production implémentées dans l'analyseur syntaxique :

1. L'Axiome de départ

$$Z \rightarrow S \ \#$$

Explication : L'analyse commence par le symbole de départ S et doit obligatoirement se terminer par la fin du fichier (représentée par la sentinelle $\#$).

2. Structure du Programme

$$S \rightarrow Instruction \ S \mid \epsilon$$

Explication : Un programme est une suite récursive d'instructions. La récursivité s'arrête (cas ϵ) à la fin du fichier ou à la fin d'un bloc.

3. Les Instructions

$$Instruction \rightarrow DeclVar \mid DeclFonction \mid ForEach \mid Affectation \mid BlocIgnore$$

4. Règles Détaillées

Déclaration de Variable :

$$DeclVar \rightarrow (\texttt{var} \mid \texttt{let})\ \textbf{id}\ [\texttt{=}\ \textbf{valeur}]\ \texttt{;}$$

Déclaration de Fonction :

$$DeclFonction \rightarrow \texttt{function}\ \textbf{id}\ \texttt{( )}\ \texttt{\{}\ S\ \texttt{\}}$$

Note : Le symbole S est rappelé à l'intérieur des accolades, ce qui permet d'imbriquer des instructions dans la fonction.

Instruction Cible (ForEach) :

$$ForEach \rightarrow \textbf{RAYEL}\ \texttt{.}\ \texttt{forEach}\ \texttt{(}\ \textbf{id}\ \texttt{=>}\ \texttt{\{}\ S\ \texttt{\}}\ \texttt{)}\ \texttt{;}$$

Explication : Cette règle impose la structure exacte demandée : le mot-clé **RAYEL**, suivi de la méthode `forEach`, d'une variable temporaire, d'une fonction fléchée, et d'un bloc de code.

Affectation :

$$Affectation \rightarrow \textbf{id}\ \texttt{=}\ \textbf{valeur}\ \texttt{;}$$

Structures Ignorées :

$$BlocIgnore \rightarrow (\texttt{if} \mid \texttt{while} \mid \texttt{for})\ \ldots\ \texttt{\{}\ S\ \texttt{\}}$$

Chapitre 2

Analyse Lexicale (Scanner)

L'analyseur lexical a pour but de découper le code source en *tokens*. J'ai implémenté un automate à états finis déterministe (DFA) utilisant des **matrices de transition**, comme vu en cours.

2.1 Les Matrices de Transition

Pour reconnaître la *forme* des lexèmes, j'utilise trois matrices :

1. **MAT_ID** : Reconnaît les chaînes alphanumériques (commençant par une lettre).
2. **MAT_NUM** : Reconnaît les suites de chiffres.
3. **MAT_SEP** : Reconnaît les séparateurs simples.

Chaque matrice possède une colonne d'erreur (souvent notée -1) qui permet d'arrêter la lecture du mot dès qu'un caractère invalide est rencontré.

2.2 Les Tableaux de Vérification

Une fois la forme reconnue, le scanner vérifie le *contenu* grâce à des tableaux statiques :
— **TAB_MOTS_CLES** : Liste des mots réservés (`var`, `if`, etc.).
— **TAB_SPECIAL** : Contient uniquement **RAYEL**.
— **TAB_OPERATEURS** : Contient les opérateurs simples et doubles (`+`, `++`, `=>`).

Chapitre 3

Analyse Syntaxique (Parser)

Le Parser vérifie la conformité de la suite de tokens par rapport à la grammaire définie au Chapitre 1.

3.1 Méthode : Descente Récursive

J'ai utilisé la méthode de la descente récursive. Cela signifie que pour chaque règle Non-Terminale de la grammaire (S, $ForEach$, etc.), j'ai écrit une méthode Java correspondante.

- La méthode `Z()` lance l'analyse.
- La méthode `S()` agit comme un aiguilleur : elle regarde le token actuel et appelle la fonction appropriée (`DeclVar()`, `InstructionForEach()`...).

3.2 Gestion des Erreurs

Le parser est conçu pour ne pas s'arrêter à la première erreur mineure. Si un token attendu est manquant (ex : un point-virgule), il affiche un message d'erreur explicite indiquant la ligne, marque le programme comme "INCORRECT", mais tente de continuer l'analyse pour trouver d'autres erreurs potentielles.

Chapitre 4

Jeux de Tests

Voici 5 scénarios de tests effectués pour valider la robustesse du compilateur.

4.1 Test 1 : Code Valide

Ce test vérifie une déclaration, une fonction et l'instruction cible.

```
var x = 10;
function calcul() { var y = 2; }
RAYEL.forEach(elt => { x = x + 1; });
```

Résultat : `Programme Syntaxiquement CORRECT !`

4.2 Test 2 : Erreur Lexicale

Insertion d'un caractère interdit (`@`) non reconnu par les matrices.

```
var mail = user@gmail.com;
```

Résultat : Le Scanner affiche `ERREUR LEXICALE : Caractère invalide '@'` mais continue l'analyse vers le Parser.

4.3 Test 3 : Erreur Syntaxique (Instruction Cible)

Syntaxe incorrecte du `forEach` (oubli de la flèche `=>`).

```
RAYEL.forEach(elt { x = 0; });
```

Résultat : `ERREUR SYNTAXIQUE : Attendu '=>'`. Le programme est déclaré INCORRECT.

4.4 Test 4 : Erreur Syntaxique (Déclaration)

Oubli du point-virgule final.

```
var note = 20
```

Résultat : `ERREUR SYNTAXIQUE : Manque le point-virgule`.

4.5 Test 5 : Structures Ignorées

Vérification que le compilateur ignore correctement un `while` imbriqué.

```
while(true) {
    if(x < 0) { x = 0; }
}
RAYEL.forEach(e => { x = 1; });
```

Résultat : Le parser ignore le bloc `while` et valide ensuite le `RAYEL.forEach`. `Programme Syntaxiquement CORRECT !`.

Conclusion

Ce projet m'a permis de mettre en pratique les concepts théoriques de la compilation. J'ai pu implémenter un scanner robuste basé sur des matrices de transition et un parser récursif capable de valider une grammaire définie. L'utilisation de GitHub pour le versioning et la génération d'un exécutable autonome ont complété cette expérience de développement.