Université de Béjaïa
Faculté des Sciences Exactes
Département d'Informatique

Rapport de Mini Projet Compilateur While JS

Étudiante : Nissette Slimani

Groupe : B3

Année universitaire 2025–2026

Introduction

Ce mini projet a pour objectif de développer un **analyseur lexical et syntaxique** pour un langage simplifié inspiré de JavaScript, nommé **WhileJS**. L'objectif principal est de vérifier la syntaxe des programmes et de produire des messages d'erreur précis, permettant à l'utilisateur de corriger efficacement son code.

Le projet se compose de deux modules principaux :

— **Analyseur lexical (Lexer)** : il divise le code source en unités lexicales appelées **tokens**. Chaque token représente un élément fondamental du langage, comme un mot-clé, un identifiant, un opérateur ou un nombre.

— **Analyseur syntaxique (Parser)** : il analyse la séquence de tokens fournie par le Lexer afin de vérifier la conformité des instructions à la grammaire de WhileJS et d'identifier les erreurs syntaxiques.

Ces deux composants collaborent pour assurer une détection fiable des erreurs et la validation correcte des programmes. Le Lexer prépare le code pour le Parser, qui s'assure que chaque instruction respecte les règles définies par la grammaire.

1 Analyseur Lexical

1.1 Rôle du Lexer

L'analyseur lexical, ou *Lexer*, a pour mission de transformer le code source en une suite de **tokens**. Chaque token représente une unité lexicale du langage WhileJS et contient les informations suivantes :

— La **ligne** où il apparaît dans le code source.
— La **valeur** exacte du token.
— Le **type** du token, tel que défini dans la classe `TypeToken`.

Cette étape est essentielle pour simplifier le travail du Parser, qui n'a plus à gérer directement le code brut.

1.2 Tokens définis

Le Lexer identifie différents types de tokens, dont voici les principaux :

— **Mots-clés :** WHILE, LET, CONST, RETURN, BREAK, IF, ELSE, FOR, CASE, TRY, CATCH
— **Identifiants et littéraux :** VARIABLE, IDENTIFIANT, NOMBRE, STRING
— **Opérateurs :** EQUAL, PLUS, MOINS, FOIS, DIVISE, MODULO, EGAL_EGAL, DIFFERENT
— **Symboles de structure :** ACCOLADE_OUVRANTE, ACCOLADE_FERMANTE, PARENTHESE_OUVRANTE, PARENTHESE_FERMANTE, POINT_VIRGULE, POINT
— **Autres :** Console_log, ET, OU, SLIMANI, NISSETTE, EOF, ERREUR

1.3 Exemple d'instruction et tokens générés

Code source :

```
[language=JavaScript]
let x = 0;
while (x < 5) {
    x = x + 1;
}
```

Tokens générés :

Ligne	Valeur	Type
1	let	LET
1	x	VARIABLE
1	=	EQUAL
1	0	NOMBRE
1	;	POINT_VIRGULE
2	while	WHILE
2	(	PARENTHESE_OUVRANTE
2	x	VARIABLE
2	¡	INFERIEUR
2	5	NOMBRE
2	)	PARENTHESE_FERMANTE
2	{	ACCOLADE_OUVRANTE
3	x	VARIABLE
3	=	EQUAL
3	x	VARIABLE
3	+	PLUS
3	1	NOMBRE
3	;	POINT_VIRGULE
4	}	ACCOLADE_FERMANTE

1.4 Fonctionnement du Lexer

Le Lexer parcourt le code source caractère par caractère et applique les règles suivantes :

— Les mots-clés et identifiants sont détectés à partir de lettres et underscore.

— Les nombres sont reconnus comme une suite de chiffres.

— Les opérateurs et symboles sont identifiés par des motifs fixes.

— Tout caractère ou séquence non reconnue est marqué comme un `ERREUR`.

Cette structuration permet au Parser de travailler uniquement sur des tokens cohérents, sans se préoccuper des détails syntaxiques de bas niveau.

2 Analyse Syntaxique

2.1 Grammaire utilisée

Le Parser se base sur la grammaire suivante pour analyser les programmes WhileJS. Les flèches sont représentées par → pour une meilleure lisibilité dans le rapport.

```
1
2  program          → instruction program |
3  instruction      → while_statement | declaration | assignation |
       returnStatement
4                     | breakStatement | slimani | nissette |
                        IGNORABLE_KEYWORD
5  while_statement → while '(' condition ')' '{' program '}'
6  declaration      → (let|const) VARIABLE '=' expression ';'
7  assignation      → VARIABLE '=' expression ';'
8  returnStatement → return expression ';'
9  breakStatement   → break ';'
10 slimani          → Slimani ';'
11 nissette         → Nissette ';'
12 condition        → expression operation expression
13 operation        → > | < | >= | <= | == | !=
14 expression       → term { (+|-) term }
15 term             → facteur { (*|/|%) facteur }
16 facteur          → NOMBRE | VARIABLE | '(' expression ')'
17 IGNORABLE_KEYWORD → if | else | for | case | try | catch |
       Console_log
```

2.2 Déroulement de l'analyse syntaxique

Le Parser suit les étapes suivantes pour analyser un programme :

1. Lecture de la séquence de tokens générée par le Lexer.
2. Tentative de reconnaissance de chaque instruction selon la grammaire.
3. Signalement d'une erreur si un token ne correspond à aucune règle.
4. Saut des mots-clés ignorables (`if`, `else`, `for`, `case`, `try`, `catch`, `Console_log`) avec leur bloc associé.
5. Poursuite de l'analyse jusqu'à la fin du programme pour détecter plusieurs erreurs simultanément.

2.3 Types d'erreurs détectées

Le Parser est capable d'identifier différents types d'erreurs :

— **Erreur de syntaxe** : variable attendue, symbole ';' manquant, parenthèse ou accolade manquante.
— **Facteur invalide** : un nombre, une variable ou une expression entre parenthèses est attendu.
— **Opérateur relationnel attendu** : ¿, ¡, ¿=, ¡=, ==, !=.
— **Instruction invalide** : début d'instruction non reconnu par la grammaire.

2.4 Tokens ignorés

Les mots-clés non encore implémentés sont ignorés par le Parser :
— `if`, `else`, `for`, `case`, `try`, `catch`, `Console_log`

2.5 Exemples de déroulement et d'erreurs

Exemple 1 : erreurs syntaxiques

```
let x 0;
while(x < 5 {
    x = x + 1;
}
```

Sortie du Parser :

```
Erreur syntaxique: = attendu apr s variable (jeton: 0) ligne 1
Erreur syntaxique: ) attendue apr s condition (jeton: {) ligne 2
```

Exemple 2 : motclé ignoré

```
if(x > 3) {
    x = x + 1;
}
while(x < 5) { x = x + 1; }
```

Sortie du Parser :

```
Chaine acceptee
```

Le Parser ignore le bloc `if` et continue l'analyse des instructions valides.

2.6 Conclusion sur l'analyse syntaxique

L'analyseur syntaxique a permis de :
— Vérifier la conformité des programmes à la grammaire de WhileJS.
— Détecter de manière précise les erreurs syntaxiques et relationnelles.
— Gérer les mots-clés ignorables pour des structures non implémentées.
— Fournir une base solide pour le développement futur d'un interpréteur ou compilateur complet.

Cette approche assure une analyse complète et robuste des programmes, tout en permettant à l'utilisateur de corriger efficacement les erreurs détectées.

3 Conclusion

Ce mini projet a permis de concevoir et de réaliser un analyseur lexical et syntaxique pour le langage simplifié WhileJS.

L'analyseur lexical (Lexer) a été capable de transformer un programme en une suite de tokens précis et structurés, facilitant l'analyse ultérieure. L'analyseur syntaxique (Parser) a permis de vérifier la conformité des programmes à la grammaire, de détecter diverses erreurs syntaxiques et relationnelles, et de gérer les mots-clés ignorables pour les structures non encore implémentées.

Grâce à cette approche modulaire et progressive, nous avons obtenu un système capable de :

— Identifier correctement les unités lexicales et les erreurs lexicales.
— Vérifier la structure syntaxique des programmes tout en signalant des erreurs claires et compréhensibles.
— Maintenir la robustesse du système même en présence de plusieurs erreurs simultanées.
— Fournir une base solide pour un futur interpréteur ou compilateur complet pour WhileJS.

En conclusion, ce projet a permis d'acquérir une expérience pratique dans la conception d'outils d'analyse de langage, en consolidant à la fois les connaissances théoriques et les compétences en programmation. Il constitue un point de départ solide pour l'extension future du langage et pour le développement de fonctionnalités avancées, telles que l'interprétation ou la compilation.