Rapport de Projet : Analyseur Lexical et Syntaxique pour PHP Simplifié

Votre Nom

25 novembre 2025

Résumé

Ce projet consiste en la mise en œuvre d'un analyseur lexical et syntaxique pour un sous-ensemble du langage PHP, utilisant la technique de **descente récursive**. L'objectif est de comprendre les principes fondamentaux de la construction des compilateurs. L'analyseur supporte un sous-ensemble significatif de PHP incluant les structures de contrôle avancées comme `foreach`.

Table des matières

Chapitre 1

Introduction

1.1 Contexte et Objectifs

Ce projet consiste en la mise en œuvre d'un analyseur lexical et syntaxique pour un sous-ensemble du langage PHP, utilisant la technique de **descente récursive**. L'objectif est de comprendre les principes fondamentaux de la construction des compilateurs.

Chapitre 2

Architecture du Projet

2.1 Fichiers Implémentés

2.1.1 TokenType.java

Rôle : Définition des types de tokens reconnus par l'analyseur lexical.

- **Mots-clés** : `VAR`, `IF`, `ELSE`, `WHILE`, `FOREACH`, `AS`
- **Identifiants** : `IDENTIFIER` (avec ou sans `$`)
- **Littéraux** : `NUMBER`, `STRING`
- **Opérateurs** :
  - Arithmétiques : `PLUS`, `MINUS`, `MUL`, `DIV`
  - Comparaison : `EQ`, `NEQ`, `LT`, `GT`, `LE`, `GE`
  - Logiques : `AND`, `OR`
  - Incrémentation : `INC`, `DEC`
  - Assignation : `ASSIGN`
- **Délimiteurs** :
  - Parenthèses : `LPAREN`, `RPAREN`
  - Accolades : `LBRACE`, `RBRACE`
  - Crochets : `LBRACKET`, `RBRACKET`
  - Ponctuation : `COMMA`, `SEMICOLON`, `DOT`, `QUESTION`, `COLON`

2.1.2 Token.java

Rôle : Représentation d'un token avec son type, valeur et ligne.

```
1 public class Token {
2     public TokenType type;
3     public String value;
4     public int line;
5 }
```

Listing 2.1 – Classe Token

2.1.3 Lexer.java

Rôle : Analyseur lexical convertissant le code source en flux de tokens.
Méthodes principales :
— `tokenize()` : Conversion caractères → tokens
— `readIdentifierOrKeyword()` : Lecture identifiants/mots-clés
— `readString()` : Lecture chaînes de caractères
— `readNumber()` : Lecture nombres

2.1.4 Parser.java

Rôle : Analyseur syntaxique vérifiant la structure grammaticale.

2.1.5 SyntaxException.java

Rôle : Gestion des erreurs syntaxiques avec numéros de ligne.

2.1.6 Main.java

Rôle : Point d'entrée principal .

Chapitre 3

Méthodologie : Descente Récursive

3.1 Principe de la Descente Récursive

La **descente récursive** est une méthode d'analyse syntaxique descendante où chaque production de la grammaire est implémentée comme une fonction récursive.

3.2 Application dans le Projet

3.2.1 Fonctions Principales du Parser

`parseProgram()` - Point d'entrée

```
public void parseProgram() {
    while (peek().type != TokenType.EOF) {
        parseStatement();
    }
}
```

Listing 3.1 – Fonction parseProgram

`parseStatement()` - Distribution aux différentes structures

```
private void parseStatement() {
    switch (peek().type) {
        case VAR: parseVarDecl(); break;
        case IF: parseIf(); break;
        case WHILE: parseWhile(); break;
        // ...
    }
```

```
8 }
```

Listing 3.2 – Fonction parseStatement

Fonctions Spécialisées

— `parseVarDecl()` : Déclarations de variables
— `parseIf()`, `parseWhile()`, `parseForeach()` : Structures de contrôle
— `parseExpression()` : Expressions arithmétiques et logiques

3.3 Avantages de la Descente Récursive

— **Simplicité** : Code proche de la grammaire
— **Maintenabilité** : Facile à comprendre et modifier
— **Détection d'erreurs précise** : Localisation exacte des problèmes
— **Adaptabilité** : Extension aisée de la grammaire

Chapitre 4

Analyse Lexicale

4.1 Processus d'Analyse Lexicale

1. **Segmentation** : Découpage du code en tokens
2. **Classification** : Identification du type de chaque token
3. **Validation** : Vérification de la conformité lexicale

4.2 Tokens Supportés

— **Mots-clés** : `var`, `if`, `else`, `while`, `foreach`, `as`
— **Identifiants** : Variables avec `$` ou sans
— **Littéraux** : Nombres, chaînes de caractères
— **Opérateurs** : Arithmétiques (`+`, `-`, `*`, `/`), de comparaison (`==`, `!=`, `<`, `>`), logiques (`&&`, `||`)
— **Délimiteurs** : Parenthèses, accolades, crochets, points-virgules

Chapitre 5

Analyse Syntaxique

5.1 Grammaire Implémentée

```
Prog        → StmtList
StmtList    → Stmt StmtList |

Stmt        → VarDecl ';'
            | Assignment ';'
            | IfStmt
            | WhileStmt
            | ForEachStmt
            | IncDecStmt ';'
            | Comparison ';'

VarDecl     → 'var' Identifier [ '=' Expr ]
Assignment  → Identifier '=' Expr
IncDecStmt  → Identifier ( '++' | '--' )
Comparison  → Expr ComparisonOp Expr

IfStmt      → 'if' '(' Expr ')' '{' StmtList '}' [ 'else' '{' StmtList '}' ]
WhileStmt   → 'while' '(' Expr ')' '{' StmtList '}'

ForEachStmt → 'foreach' '(' Identifier 'as' Identifier ')' '{' StmtList '}'

Expr        → Expr ( '+' | '-' | '*' | '/' | '&&' | '||' ) Expr
            | '(' Expr ')'
            | Identifier
            | Number
```

```
            | String

ComparisonOp → '==' | '!=' | '<' | '>' | '<=' | '>='

Identifier  → [a-zA-Z_][a-zA-Z0-9_]*
Number      → [0-9]+
String      → '"' .* '"'
```

5.2 Vérifications Syntaxiques

— Structure des blocs (`{}`, `()`, `[]`)
— Point-virgules terminaux
— Expressions bien formées
— Correspondance des structures de contrôle

Chapitre 6

Tests et Validation

6.1 Cas de Test Validés

1. **Déclarations simples** : `var x = 10;`
2. **Structures de contrôle** : `if`, `while`, `foreach`
3. **Expressions complexes** : `$result = ($a + $b) * $c;`
4. **Tableaux** : `$numbers = [1, 2, 3];`
5. **Programmes complets** avec structures imbriquées

6.2 Détection d'Erreurs

L'analyseur détecte :
— Chaînes non fermées
— Points-virgules manquants
— Parenthèses non correspondantes
— Expressions incomplètes
— Structures mal formées

6.3 Stratégie de Récupération

Approche "Fail-Fast" :
— Détection immédiate à la première erreur
— Arrêt du parsing avec message descriptif
— Localisation précise (numéro de ligne)

Chapitre 7

Résultats

7.1 Succès du Projet

— **Analyseur lexical fonctionnel** : Tokenisation correcte
— **Analyseur syntaxique robuste** : Vérification structurelle
— **Gestion d'erreurs efficace** : Messages informatifs
— **Couverture grammaticale** : Support du PHP simplifié

Chapitre 8

Conclusion

Ce projet a démontré la mise en œuvre réussie d'un analyseur lexical et syntaxique utilisant la méthode de descente récursive. L'analyseur supporte un sous-ensemble significatif de PHP incluant les structures de contrôle avancées comme `foreach`.