UNIVERSITÉ / ÉCOLE SUPÉRIEURE

Faculté des Sciences et Technologies

Département Informatique

Analyseur Lexical et Syntaxique pour le Langage PHP IF ELSE

Compilateur spécialisé IF/ELSE

Projet de Compilation

Module : Théorie des Langages et Compilation

Année Universitaire : 2024-2025

Réalisé par :

MISSILVA
BOUMOULA
Groupe : A3

Encadré par :

Mme Nadia TASSOULT
Département Informatique

Année universitaire :2025-2026

7 décembre 2025

Résumé

Ce rapport présente l'implémentation d'un compilateur PHP spécialisé dans l'analyse des structures conditionnelles if/else. Le projet se compose de trois modules principaux : un analyseur lexical (Lexer), un analyseur syntaxique (Parser) et un programme principal interactif. L'objectif est de valider la syntaxe PHP en se concentrant particulièrement sur les instructions conditionnelles, les déclarations de fonctions et de classes.

Table des matières

1 Introduction

1.1 Contexte

L'analyse de code est une étape fondamentale dans le processus de compilation. Ce projet implémente les deux premières phases d'un compilateur : l'analyse lexicale et l'analyse syntaxique pour le langage PHP.

1.2 Objectifs

— Tokeniser le code PHP en unités lexicales
— Valider la syntaxe des structures if/else/elseif
— Analyser les déclarations de fonctions et classes
— Fournir une interface interactive pour tester du code
— Gérer les erreurs sans bloquer l'analyse

1.3 Architecture du Système

Le compilateur est composé de trois classes Java principales :

1. **LexerPHP.java** : Analyseur lexical
2. **ParserPHP.java** : Analyseur syntaxique
3. **MainLexerParser.java** : Interface utilisateur

2 Analyse Lexicale (LexerPHP)

2.1 Principe de Fonctionnement

L'analyseur lexical transforme une chaîne de caractères (code source) en une séquence de tokens. Chaque token représente une unité lexicale significative du langage PHP.

2.2 Automates à États Finis (DFA)

Le Lexer utilise trois automates à états finis déterministes :

2.2.1 DFA pour Identificateurs et Variables

— **États** : S0 (initial), S1 (identificateur), S2 (symbole $), S3 (variable)
— **Colonnes** : lettre/underscore, chiffre, $, autre
— **États finaux** : S1 (identificateur), S3 (variable PHP)

Listing 1 – Matrice de transition pour identificateurs

```
private static final int[][] MIdent = {
    {1, -1, 2, -1}, // S0
    {1, 1, -1, -1}, // S1 (identificateur)
    {3, -1, -1, -1}, // S2 (apres $)
    {3, 3, -1, -1}  // S3 (variable)
};
```

2.2.2 DFA pour Nombres

Reconnaît trois types de nombres :
— Entiers : `42`, `1000`
— Flottants : `3.14`, `0.5`
— Notation scientifique : `1.5e-10`, `3E8`

2.2.3 DFA pour Opérateurs

Gère les opérateurs PHP :
— Arithmétiques : `+`, `-`, `*`, `/`, `%`
— Comparaison : `==`, `!=`, `<`, `>`, `<=`, `>=`
— Logiques : `&&`, `||`
— Affectation : `=`, `+=`, `-=`, `*=`, `/=`
— Incrémentation : `++`, `--`

2.3 Types de Tokens Reconnus

Type	Description	Exemple
MOT_CLE	Mot-clé PHP	if, else, function
VARIABLE	Variable PHP	$age, $nom
IDENT	Identificateur	verifierCategorie
NOMBRE (INT)	Nombre entier	42, 1000
NOMBRE (FLOAT)	Nombre flottant	3.14, 0.5
STRING	Chaîne de caractères	”Hello”, ’World’
OPERATEUR	Opérateur	+, -, *
COMPARATEUR	Comparateur	==, !=, ¡
DELIMITEUR	Délimiteur	;, {, }, (,)
TAG_PHP	Balise PHP	¡?php, ?¿

Table 1 – Types de tokens reconnus par le Lexer

2.4 Gestion des Commentaires

Le Lexer ignore trois types de commentaires PHP :

Listing 2 – Types de commentaires supportés

```
// Commentaire sur une ligne

# Commentaire style shell

/* Commentaire
   multi-lignes */
```

2.5 Exemple de Tokenisation

Listing 3 – Code PHP exemple

```
<?php
$x = 10;
```

```php
if ($x > 5) {
    echo "Grand";
}
?>
```

Tokens générés :

```
[<?php] => type: TAG_PHP, ligne: 1
[$x] => type: VARIABLE, ligne: 2
[=] => type: OPERATEUR_ASSIGN, ligne: 2
[10] => type: NOMBRE (INT), ligne: 2
[;] => type: DELIMITEUR, ligne: 2
[if] => type: MOT_CLE, ligne: 3
[($x] => type: VARIABLE, ligne: 3
[>] => type: COMPARATEUR, ligne: 3
[5] => type: NOMBRE (INT), ligne: 3
...
```

3 Analyse Syntaxique (ParserPHP)

3.1 Principe de l'Analyse Descendante Récursive

Le Parser utilise une approche d'analyse descendante récursive où chaque règle de grammaire correspond à une méthode Java.

3.2 Grammaire Complète Utilisée dans le Parser

3.2.1 Règles de Production Principales

```
Programme ::= TAG_PHP Statement* TAG_PHP?

Statement ::= ClassDecl
            | FunctionDecl
            | IfStmt
            | LoopStmt
            | Assignment
            | FunctionCall ';'
            | Echo
            | Return
            | ';'

ClassDecl ::= 'class' IDENT ('extends' IDENT)?
              ('implements' IDENT (',' IDENT)*)?
              '{' ClassMember* '}'

ClassMember ::= Visibility? Modifier?
                (FunctionDecl | PropertyDecl)

Visibility ::= 'public' | 'private' | 'protected'

Modifier ::= 'static' | 'final'
```

```
PropertyDecl ::= VARIABLE ('=' Expression)? ';'

FunctionDecl ::= 'function' IDENT '(' ParamList? ')'
                 ReturnType? '{' Statement* '}'

ParamList ::= Param (',' Param)*

Param ::= Type? VARIABLE ('=' Expression)?

Type ::= 'int' | 'float' | 'string' | 'bool'
       | 'array' | 'mixed' | 'void'

ReturnType ::= ':' Type

IfStmt ::= 'if' '(' Expression ')' Block
           ElseIfClause*
           ElseClause?

ElseIfClause ::= 'elseif' '(' Expression ')' Block
               | 'else' 'if' '(' Expression ')' Block

ElseClause ::= 'else' Block

Block ::= '{' Statement* '}'
        | Statement

LoopStmt ::= 'while' '(' Expression ')' Block
           | 'for' '(' Expression? ';' Expression? ';'
                     Expression? ')' Block
           | 'foreach' '(' Expression 'as' ... ')' Block
           | 'switch' '(' Expression ')' '{' ... '}'
           | 'do' Block 'while' '(' Expression ')' ';'

Assignment ::= (VARIABLE | IDENT) AssignmentTail ';'

AssignmentTail ::= '=' Expression
                 | '+=' Expression
                 | '-=' Expression
                 | '*=' Expression
                 | '/=' Expression
                 | '%=' Expression
                 | '++'
                 | '--'
                 | '(' ArgList? ')'  // Appel fonction
                 | '->' IDENT ('(' ArgList? ')')?  // Méthode
                 | '::' IDENT ('(' ArgList? ')')?  // Statique
                 | '[' Expression ']'  // Tableau

Echo ::= ('echo' | 'print') Expression (',' Expression)* ';'
```

```
Return ::= 'return' Expression? ';'

ArgList ::= Expression (',' Expression)*
```

3.2.2 Règles pour les Expressions (Analyse Récursive)

```
Expression ::= Comparison

Comparison ::= Term (CompOp Term)*

CompOp ::= '==' | '!=' | '<' | '>' | '<=' | '>='
         | '&&' | '||'

Term ::= Factor (TermOp Factor)*

TermOp ::= '+' | '-' | '.'  // Concat string

Factor ::= Primary (FactorOp Primary)*

FactorOp ::= '*' | '/' | '%'

Primary ::= '(' Expression ')'
          | UnaryOp Primary
          | VARIABLE ('[' Expression ']')?
          | NUMBER
          | STRING
          | BOOLEAN
          | Array
          | FunctionCall

UnaryOp ::= '!' | '-'

Array ::= '[' (Expression (',' Expression)*)? ']'

FunctionCall ::= IDENT '(' ArgList? ')'
```

3.2.3 Hiérarchie de Priorité des Opérateurs

Niveau	Opérateurs	Associativité
1 (plus haut)	(), [], $->$, ::	Gauche
2	!, $-$ (unaire)	Droite
3	$*$, /, %	Gauche
4	$+$, $-$, . (concat)	Gauche
5	$<$, $>$, $<=$, $>=$	Gauche
6	$==$, $!=$	Gauche
7 (plus bas)	&&, $\|\|$	Gauche

TABLE 2 – Priorité des opérateurs dans l'analyse

3.2.4 Tokens Terminaux

```
TAG_PHP      ::= '<?php' | '<?'  | '?>'
MOT_CLE      ::= 'if' | 'else' | 'elseif' | 'function'
               | 'class' | 'return' | 'echo' | ...
VARIABLE     ::= '
```

```
\subsection{Analyse des Structures IF/ELSE}
C'est le cœur du compilateur. La méthode \texttt{parseIf()} gère :

\begin{itemize}
    \item IF simple avec condition
    \item ELSEIF multiples
    \item ELSE IF (deux mots séparés)
    \item ELSE final
    \item Blocs avec ou sans accolades
\end{itemize}

\begin{lstlisting}[style=javastyle, caption=Structure de parseIf()]
private void parseIf() {
    advance(); // 'if'
    expectValue("(", "'(' attendu");
    parseExpression(); // Condition
    expectValue(")", "')' attendu");

    // Bloc THEN
    if (matchValue("{")) {
        // Bloc avec accolades
    } else {
        // Instruction unique
    }

    // ELSEIF / ELSE IF
    while (matchValue("elseif") ||
           (matchValue("else") && peek("if"))) {
        // Traiter elseif
    }

    // ELSE
    if (matchValue("else")) {
        // Traiter else
    }
}
\end{lstlisting}

\subsection{Gestion des Erreurs}
Le Parser implémente un mécanisme de récupération d'erreur :

\begin{enumerate}
    \item Détection de l'erreur
    \item Enregistrement du message d'erreur
```

```
    \item Recherche du prochain point d'ancrage (;, \}, mot-clé)
    \item Continuation de l'analyse
\end{enumerate}

\begin{lstlisting}[style=javastyle, caption=Méthode de récupération]
private void recoverFromError() {
    while (currentPos < tokens.size()) {
        if (match(";") || match("}") ||
            isKeyword(current())) {
            advance();
            return;
        }
        advance();
    }
}
\end{lstlisting}

\subsection{Support des Types PHP}
Le Parser reconnaît :
\begin{itemize}
    \item Types de paramètres : \texttt{int}, \texttt{string}, \texttt{float}, \texttt{b
    \item Types de retour : \texttt{: string}, \texttt{: int}, \texttt{: void}
    \item Types nullable : \texttt{?string}
\end{itemize}

\begin{lstlisting}[style=phpstyle, caption=Fonction avec types]
function verifierCategorie(int $age): string {
    if ($age < 0) {
        return "Invalide";
    }
    return "Valide";
}
\end{lstlisting}

% ========================================
\section{Interface Utilisateur (MainLexerParser)}
% ========================================

\subsection{Boucle Interactive}
Le programme principal fonctionne en boucle infinie permettant de tester plusieurs codes

\subsection{Détection Automatique de Fin}
Deux méthodes pour terminer la saisie :
\begin{enumerate}
    \item Taper \texttt{###} sur une ligne
    \item Le tag de fermeture \texttt{?>} est détecté automatiquement
\end{enumerate}

\subsection{Flux d'Exécution}
\begin{enumerate}
```

```
    \item Lecture du code multi-lignes
    \item Ajout automatique de \texttt{<?php} si absent
    \item \textbf{Phase 1 :} Analyse lexicale
    \begin{itemize}
        \item Génération des tokens
        \item Affichage de la liste complète
        \item Statistiques par type
    \end{itemize}
    \item \textbf{Phase 2 :} Analyse syntaxique
    \begin{itemize}
        \item Validation de la structure
        \item Détection des erreurs
        \item Rapport final
    \end{itemize}
    \item Retour à la saisie (ou exit)
\end{enumerate}

\subsection{Exemple d'Exécution}
\begin{verbatim}

   ANALYSEUR LEXICAL ET SYNTAXIQUE PHP - IF/ELSE


============================================================
Entrez votre code PHP
- Terminez avec '###' OU simplement avec '?>'
- Tapez 'exit' pour quitter
============================================================

<?php
$x = 10;
if ($x > 5) {
    echo "Grand";
}
?>

============================================================
 PHASE 1 : ANALYSE LEXICALE
============================================================

 Tokens générés : 15

--- LISTE DES TOKENS ---
[<?php] => type: TAG_PHP, ligne: 1
[$x] => type: VARIABLE, ligne: 2
[=] => type: OPERATEUR_ASSIGN, ligne: 2
...

--- STATISTIQUES ---
TAG_PHP : 2
```

```
VARIABLE : 1
MOT_CLE : 2
...

============================================================
 PHASE 2 : ANALYSE SYNTAXIQUE
============================================================

=== DEBUT DE L'ANALYSE SYNTAXIQUE ===

Analyse: Affectation/Appel (ligne 2)
Analyse: Instruction IF (ligne 3)
Analyse: Echo/Print (ligne 4)

=== RESULTATS ===
Analyse REUSSIE : Programme syntaxiquement correct !

 VALIDATION COMPLÈTE RÉUSSIE !
Le code est lexicalement et syntaxiquement correct.

============================================================
```

4 Cas d'Utilisation

4.1 Exemple 1 : IF/ELSE Simple

Listing 4 – Structure conditionnelle simple

```php
<?php
$age = 15;
if ($age < 18) {
    echo "Mineur";
} else {
    echo "Majeur";
}
?>
```

Résultat : Programme syntaxiquement correct

4.2 Exemple 2 : ELSEIF Multiple

Listing 5 – Structure avec plusieurs elseif

```php
<?php
function note($score) {
    if ($score >= 90) {
        return "A";
    } elseif ($score >= 80) {
        return "B";
    } elseif ($score >= 70) {
        return "C";
    } else {
```

```
10          return "F";
11      }
12  }
13  ?>
```

Résultat : Programme syntaxiquement correct

4.3 Exemple 3 : ELSE IF (deux mots)

Listing 6 – Support de else if séparé

```
1  <?php
2  $x = 10;
3  if ($x > 20) {
4      echo "Grand";
5  } else if ($x > 10) {  // Deux mots
6      echo "Moyen";
7  } else {
8      echo "Petit";
9  }
10 ?>
```

Résultat : Programme syntaxiquement correct (grâce à la détection spéciale)

4.4 Exemple 4 : Code avec Erreur

Listing 7 – Code syntaxiquement incorrect

```
1  <?php
2  if ($x > 5 {  // Parenthese fermante manquante
3      echo "Test";
4  }
5  ?>
```

Résultat :

```
Analyse ECHOUEE : 1 erreur(s) detectee(s)

Erreur ligne 2: ')' attendu apres la condition (trouve: {)
```

5 Limitations et Améliorations Futures

5.1 Limitations Actuelles

- Pas d'analyse sémantique (types, portée des variables)
- Structures while, for, foreach sont ignorées
- Pas de génération de code intermédiaire
- Classes et méthodes analysées mais pas complètement validées

5.2 Améliorations Possibles

1. **Analyse sémantique**
   - Vérification des types

   - Table des symboles
   - Portée des variables
2. **Support complet PHP**
   - Boucles (while, for, foreach)
   - Switch/case
   - Try/catch
   - Namespaces
3. **Génération de code**
   - Arbre syntaxique abstrait (AST)
   - Code intermédiaire
   - Optimisations
4. **Interface graphique**
   - Coloration syntaxique
   - Affichage visuel des erreurs
   - Arbre de dérivation

6 Conclusion

Ce projet démontre l'implémentation d'un compilateur front-end pour PHP, spécialisé dans l'analyse des structures conditionnelles. Les deux phases principales (analyse lexicale et syntaxique) sont fonctionnelles et robustes, avec une gestion d'erreur permettant de continuer l'analyse même en cas de problèmes syntaxiques.

L'utilisation d'automates à états finis pour le Lexer et de l'analyse descendante récursive pour le Parser représente une approche classique et efficace en compilation. Le système est extensible et peut servir de base pour un compilateur PHP plus complet.

6.1 Compétences Développées

- Conception et implémentation d'automates à états finis
- Analyse syntaxique descendante récursive
- Gestion d'erreurs en compilation
- Architecture modulaire en Java
- Manipulation de structures de données complexes

Références

- Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2006). *Compilers : Principles, Techniques, and Tools* (2nd ed.). Pearson.
- PHP Documentation Officielle : `https://www.php.net/manual/fr/`
- Grune, D., & Jacobs, C. J. (2008). *Parsing Techniques : A Practical Guide* (2nd ed.). Springer.