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

جامعة بجاية
Tasdawit n Bgayet
Université de Béjaïa

Université A/Mira de Béjaïa
Faculté des Sciences Exactes
Département d'Informatique

Rapport de Projet TP

Module : Compilation

Thème : Mini-Compilateur en Java

Contrôle : WHILE en Java

Présenté par :
Touati Jugurta

Groupe : B4

Encadré par :
Mme Nadia TASSOULT

Année universitaire : 2025 / 2026

Table des matières

1 Introduction générale **3**
- 1.1 Introduction . . . 4
- 1.2 Objectif de la grammaire . . . 4

2 Analyse Lexical : **5**
- 2.1 Étapes de l'analyse lexicale . . . 5
  - 2.1.1 Lecture caractère par caractère . . . 5
  - 2.1.2 Ignorer les espaces et les commentaires . . . 5
  - 2.1.3 Identifier les tokens . . . 5
  - 2.1.4 Construction du token . . . 5
  - 2.1.5 Avancer caractère par caractère . . . 5
  - 2.1.6 Fin de l'analyse lexicale . . . 6
- 2.2 Grammaire générale du langage . . . 6
  - 2.2.1 Notation simplifiée (EBNF) . . . 6
- 2.3 Grammaire simplifiée (BNF) . . . 7
- 2.4 Exemple d'application de la grammaire . . . 7
- 2.5 Tokens générés par le lexer . . . 9

3 Analyseur syntaxique **10**
- 3.1 Rôle de l'analyse syntaxique . . . 10
- 3.2 Déroulement général de l'analyse syntaxique . . . 10
- 3.3 PROCESSUS DU PARSING ÉTAPE PAR ÉTAPE . . . 10
  - 3.3.1 Exemple : Déroulement concret pour une déclaration . . . 11
- 3.4 Gestion des erreurs (Mode Panique) . . . 11
- 3.5 Automate à Pile (PDA) pour l'Analyse Syntaxique . . . 11
  - 3.5.1 Définition de l'automate . . . 11
  - 3.5.2 Transitions . . . 12
- 3.6 Grammaire Générale du Langage . . . 13
- 3.7 Grammaire LL(1) Factorisée . . . 14
- 3.8 Tables FIRST . . . 15
- 3.9 Diagramme de Transition d'États du Parser . . . 15
- 3.10 Schéma syntaxique . . . 16
- 3.11 Dérivation syntaxique complète . . . 17
  - 3.11.1 Arbre syntaxique simplifié . . . 17

4 La structure de votre projet **19**
- 4.1 Structure du projet . . . 19
  - 4.1.1 Explication de l'architecture . . . 20

4.2 Cas de test 20
4.2.1 Étape 1 : Analyse lexicale 20
4.2.2 Étape 2 : Analyse syntaxique 21
4.2.3 Conclusion du cas de test 22
4.3 Conclusion générale 22

Chapitre 1

Introduction générale

La compilation constitue une étape essentielle dans le développement d'un programme informatique. Elle permet de traduire un code source, écrit dans un langage de programmation de haut niveau, en un code exécutable compréhensible par la machine. Comprendre le fonctionnement interne d'un compilateur est donc une étape importante dans l'apprentissage de la conception des langages et des outils de développement. Dans le cadre de ce projet, nous avons été amenés à concevoir et à implémenter un mini-compilateur pour un sous-ensemble du langage Java. L'objectif principal est d'acquérir une compréhension pratique des différentes phases de la compilation, notamment l'analyse lexicale, l'analyse syntaxique et la gestion des erreurs. Le mini-compilateur développé a pour particularité de se concentrer sur une structure de contrôle spécifique : **la boucle while**, afin d'en simplifier la conception et de se focaliser sur les principes fondamentaux de la compilation. Pour la réalisation de ce projet, nous avons utilisé le langage **Java** à travers un environnement de développement intégré (**VS code**). Le code source a été organisé de manière modulaire afin de séparer les composants lexicaux, syntaxiques et logiques. L'outil **GitHub** a également été utilisé pour assurer le suivi du développement et la gestion des versions.

Dans ce projet, nous avons implémenté un mini-compilateur pour un sous-ensemble du langage **Java**, centré sur la boucle **while**.

Ce projet met en pratique :

— **l'analyse lexicale,**

— **l'analyse syntaxique,**

— **la gestion des erreurs.**

Le développement a été réalisé en **Java** sous **VS Code**, avec **GitHub** pour la gestion des versions.

Grammaire choisie

1.1 Introduction

La construction d'un compilateur repose sur plusieurs étapes essentielles permettant la transformation d'un programme source en une représentation interne compréhensible par la machine.

Dans ce projet, nous avons implémenté un mini-compilateur simplifié pour un sous-ensemble du langage **Java**, dont la principale structure de contrôle prise en charge est la boucle **while**.

Le compilateur que nous avons développé est composé de trois phases principales :

1. **L'analyse lexicale** : cette étape consiste à découper le code source en unités lexicales appelées *tokens* (mots-clés, identificateurs, opérateurs, symboles, etc.). Elle permet de filtrer les éléments du programme et d'éliminer les espaces et commentaires inutiles.
2. **L'analyse syntaxique** : elle vérifie que la suite de tokens respecte la grammaire définie du langage. C'est à cette étape que le compilateur s'assure que les instructions `while`, les déclarations et les affectations suivent une structure correcte.
3. **La gestion des erreurs** : le compilateur doit être capable de détecter et d'afficher clairement les erreurs lexicales ou syntaxiques rencontrées, sans interrompre brutalement le processus de compilation.

Ce projet a pour finalité de comprendre ces différentes phases et de les mettre en œuvre concrètement sur un exemple restreint mais représentatif du fonctionnement d'un véritable compilateur.

Ainsi, à travers ce mini-projet, nous avons pu mettre en pratique les concepts théoriques de compilation étudiés dans le module, tout en développant des compétences en programmation orientée objet et en analyse de langages formels.

1.2 Objectif de la grammaire

La grammaire choisie doit permettre à notre compilateur :

1. • d'identifier la syntaxe d'une boucle while valide,
2. • de vérifier la cohérence des parenthèses, accolades et expressions,
3. • de reconnaître les instructions internes comme les affectations et les incrémentations.

Chapitre 2

Analyse Lexical :

L'objectif de l'analyse lexicale est de transformer le texte brut en tokens. Chaque token est un élément significatif du langage : mot-clé, identifiant, opérateur, nombre, chaîne, etc

2.1 Étapes de l'analyse lexicale

2.1.1 Lecture caractère par caractère

— Le lexer lit le programme de gauche à droite, un caractère à la fois.

2.1.2 Ignorer les espaces et les commentaires

— Les espaces, tabulations et retours à la ligne ne produisent pas de token mais servent à compter les lignes et colonnes pour les erreurs
— Les commentaires (// ou /* ... */) sont également ignorés.

2.1.3 Identifier les tokens

— Mots-clés : int, while, if, class, etc.
— Identifiants : noms de variables ou fonctions (x, myVar).
— Nombres : 123, 3.14.
— Chaînes : "hello".
— Opérateurs : +, -, *, /, ==, !=.
— Délimiteurs : (,), , , [,], ;, ,.

2.1.4 Construction du token

— Pour chaque lexème reconnu, on crée un objet Token :
  — TokenType = type de token (IDENTIFIER, NUMBER, etc.)
  — value = valeur du token
  — line et column = position dans le fichier

2.1.5 Avancer caractère par caractère

— Le lexer maintient une position courante et une fonction peek() pour regarder le caractère suivant.

2.1.6 Fin de l'analyse lexicale

— Lorsque tous les caractères sont analysés, un token spécial EOF (End of File) est ajouté pour signaler la fin du code.

2.2 Grammaire générale du langage

La grammaire définit comment les tokens peuvent être combinés pour former des programmes valides.

2.2.1 Notation simplifiée (EBNF)

```
    <program>        ::= { <statement> }

<statement>      ::= <declaration> ";"
                   | <assignment> ";"
                   | <if_statement>
                   | <while_statement>
                   | <method_call> ";"
                   | <block>

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

<declaration>    ::= <type> <identifier> [ "=" <expression> ]

<assignment>     ::= <identifier> "=" <expression>

<if_statement>   ::= "if" "(" <condition> ")" <statement> [ "else" <statement> ]

<while_statement>::= "while" "(" <condition> ")" <statement>

<method_call>    ::= <identifier> "(" [ <expression> { "," <expression> } ] ")"

<condition>      ::= <expression> [ <comparison_op> <expression> ]

<expression>     ::= <term> { ("+" | "-") <term> }

<term>           ::= <factor> { ("*" | "/" | "%") <factor> }

<factor>         ::= <number> | <identifier> | <string> | "(" <expression> ")"

<type>           ::= "int" | "double" | "boolean" | "String" | "char" | "void" | "jugu

<comparison_op>  ::= "==" | "!=" | "<" | ">" | "<=" | ">="
```

2.3 Grammaire simplifiée (BNF)

```
programme      = declaration bloc_principal
declaration    = "int" identifiant "=" nombre ";"
bloc_principal = while_stmt
while_stmt     = "while" "(" condition ")" "{" instructions "}"
condition      = expression operateur expression

instruction    = affectation | incrementation | decrementation
instructions   = instruction | instruction instructions

affectation    = identifiant "=" expression ";"
incrementation = identifiant "++" ";"
decrementation = identifiant "--" ";"

operateur      = "==" | "!=" | "<" | ">" | "<="

identifiant    = [a-zA-Z_][a-zA-Z0-9_]*
nombre         = [0-9]+
```

2.4 Exemple d'application de la grammaire

```java
public class WhileTest {
    public static void main (String[] args) {
        int i = 0;
        int x;
        x = 0;
        double l = 0;

        while (i < 5) {
            l++;
            System.out.println("i = ");
            i++;

            while (l < 2) {
                l++;
            }
        }
    }
}
```

Listing 2.1 – Code source WhileTest.java

Lexème	Type de Token	Ligne :Col	Lexème	Type de Token	Ligne :Col
'public'	PUBLIC	2 :1	'class'	CLASS	2 :8
'WhileTest'	IDENTIFIER	2 :14	'{'	LBRACE	2 :24
'jugurta'	JUGURTA	5 :4	'public'	PUBLIC	7 :5
'static'	STATIC	7 :12	'void'	VOID	7 :19
'main'	MAIN	7 :24	'('	LPAREN	7 :29
'String'	STRING	7 :30	'['	LBRACKET	7 :36
']'	RBRACKET	7 :37	'args'	ARGS	7 :39
')'	RPAREN	7 :43	'{'	LBRACE	7 :45
'int'	INT	8 :9	'i'	IDENTIFIER	8 :13
'='	EQUAL	8 :15	'0'	NUMBER	8 :16
';'	SEMICOLON	8 :17	'int'	INT	9 :9
'x'	IDENTIFIER	9 :13	';'	SEMICOLON	9 :14
'x'	IDENTIFIER	10 :9	'='	EQUAL	10 :10
'0'	NUMBER	10 :11	';'	SEMICOLON	10 :12
'double'	DOUBLE	11 :9	'l'	IDENTIFIER	11 :16
'='	EQUAL	11 :17	'0'	NUMBER	11 :18
';'	SEMICOLON	11 :19	'while'	WHILE	12 :9
'('	LPAREN	12 :15	'i'	IDENTIFIER	12 :16
'<'	LESS	12 :18	'5'	NUMBER	12 :20
')'	RPAREN	12 :21	'{'	LBRACE	12 :23
'l'	IDENTIFIER	13 :13	'++'	PLUS_PLUS	13 :14
';'	SEMICOLON	13 :16	'System'	SYSTEM	14 :13
'.'	DOT	14 :19	'out'	OUT	14 :20
'.'	DOT	14 :23	'println'	PRINTLN	14 :24
'('	LPAREN	14 :31	'i = '	STRING_LITERAL	14 :32
')'	RPAREN	14 :39	';'	SEMICOLON	14 :40
'i'	IDENTIFIER	15 :13	'++'	PLUS_PLUS	15 :14
';'	SEMICOLON	15 :16	'while'	WHILE	18 :13
'('	LPAREN	18 :19	'l'	IDENTIFIER	18 :20
'<'	LESS	18 :21	'2'	NUMBER	18 :22
')'	RPAREN	18 :23	'{'	LBRACE	18 :25
'l'	IDENTIFIER	19 :17	'++'	PLUS_PLUS	19 :18
';'	SEMICOLON	19 :20	'}'	RBRACE	20 :13
'}'	RBRACE	21 :9	'}'	RBRACE	22 :5
'}'	RBRACE	23 :1			

TABLE 2.1 – Tableau des tokens générés par l'analyseur lexical

2.5 Tokens générés par le lexer

Lexème	**Token**	**Lexème**	**Token**
public	KEYWORD_PUBLIC	class	KEYWORD_CLASS
WhileTest	IDENTIFIER	{	LBRACE
public	KEYWORD_PUBLIC	static	KEYWORD_STATIC
void	KEYWORD_VOID	main	IDENTIFIER
(	LPAREN	String	TYPE_ID
_[	LBRACKET	_]	RBRACKET
args	IDENTIFIER	)	RPAREN
{	LBRACE	int	TYPE_INT
i	IDENTIFIER	=	ASSIGN
0	NUMBER	;	SEMICOLON
int	TYPE_INT	x	IDENTIFIER
;	SEMICOLON	x	IDENTIFIER
=	ASSIGN	0	NUMBER
;	SEMICOLON	double	TYPE_DOUBLE
l	IDENTIFIER	=	ASSIGN
0	NUMBER	;	SEMICOLON
while	WHILE	(	LPAREN
i	IDENTIFIER	<	LT
5	NUMBER	)	RPAREN
{	LBRACE	l	IDENTIFIER
++	INCREMENT	;	SEMICOLON
System	IDENTIFIER	.	DOT
out	IDENTIFIER	.	DOT
println	IDENTIFIER	(	LPAREN
"i = "	STRING	)	RPAREN
;	SEMICOLON	i	IDENTIFIER
++	INCREMENT	;	SEMICOLON
while	WHILE	(	LPAREN
l	IDENTIFIER	<	LT
2	NUMBER	)	RPAREN
{	LBRACE	l	IDENTIFIER
++	INCREMENT	;	SEMICOLON
}	RBRACE	}	RBRACE
}	RBRACE	}	RBRACE

TABLE 2.2 – Table des tokens générés par l'analyseur lexical

Chapitre 3

Analyseur syntaxique

3.1 Rôle de l'analyse syntaxique

Après l'analyse lexicale, qui transforme le programme source en une suite de tokens, l'analyse syntaxique (parser) vérifie si ces tokens respectent la structure grammaticale du langage.

Elle construit :

— un arbre syntaxique abstrait (AST),
— ou signale des erreurs de syntaxe.

3.2 Déroulement général de l'analyse syntaxique

L'analyse syntaxique du compilateur utilise la technique du descendant récursif (Recursive Descent Parser), ce qui signifie :

— chaque règle de la grammaire est implémentée par une méthode Java ;
— le parser lit les tokens un par un avec un lookahead ;
— selon le token rencontré, il décide quelle règle appliquer ;
— en cas d'erreur, il applique un mode de récupération (panic mode) pour continuer l'analyse.

Exemple général de déroulement

— Le Lexer renvoie une liste de tokens
— Le parser lit le premier token (lookahead).
— Selon le token, il appelle une méthode
— Chaque méthode consomme les tokens attendus dans sa règle grammaticale.

3.3 PROCESSUS DU PARSING ÉTAPE PAR ÉTAPE

— Récupération des tokens :Le parser reçoit une liste similaire
— Le parser lit les tokens un par un :Selon des règles de grammaire
— Chaque règle correcte crée un nœud dans l'AST
— Si une séquence ne correspond pas à une règle → ERREUR

3.3.1 Exemple : Déroulement concret pour une déclaration

Code :

```
int x = 5 + 3;
```

Tokens :

```
INT IDENTIFIER ASSIGN NUMBER PLUS NUMBER SEMICOLON
```

Le déroulement :

- parseStatement() voit INT → appelle parseIntDeclaration().
- parseIntDeclaration() vérifie :
  - INT → OK → consomme
  - IDENTIFIER → OK → consomme
  - ASSIGN → OK → consomme
  - appelle parseExpression()
  - doit trouver ;
- parseExpression() :
  - lit 5
  - lit +
  - lit 3
  - construit un nœud BinaryExpression(+, 5, 3)
- Résultat dans l'AST :
  - VarDecl(x, BinaryExpr(+, 5, 3))

3.4 Gestion des erreurs (Mode Panique)

Si un élément attendu est manquant : **Exemple :**

```
int = 5;
```

Il manque l'identifiant.

Le parser :

- affiche une erreur :
  - "Erreur syntaxique : identifiant attendu après int"
- ignore les tokens jusqu'au prochain ;
- continue l'analyse du reste du code.

Ce mécanisme permet d'obtenir plusieurs erreurs en une seule analyse.

3.5 Automate à Pile (PDA) pour l'Analyse Syntaxique

3.5.1 Définition de l'automate

- **États** : $\{q_0, q_1, q_f\}$
- **Alphabet d'entrée** : Tokens
- **Alphabet de pile** : Terminaux + Non-terminaux + $
- **État initial** : q_0
- **État final** : q_f
- **Symbole initial de pile** : Program $

3.5.2 Transitions

$$(q_0, \varepsilon, \varepsilon) \rightarrow (q_1, \text{Program } \$)$$

$$(q_1, token, X) \rightarrow (q_1, \varepsilon) \quad \text{si } X = token$$

$$(q_1, \varepsilon, A) \rightarrow (q_1, \alpha) \quad \text{si } A \rightarrow \alpha \text{ dans la grammaire}$$

$$(q_1, \$, \$) \rightarrow (q_f, \varepsilon) \quad \text{(acceptation)}$$

3.6 Grammaire Générale du Langage

$$
\begin{array}{rcl}
\langle Program \rangle & \rightarrow & (\langle ClassDeclaration \rangle \mid \langle Statement \rangle)^* \texttt{ EOF} \\
\langle ClassDeclaration \rangle & \rightarrow & \langle Modifier \rangle^* \texttt{ class } \langle Identifier \rangle \texttt{ \{ } \langle ClassBody \rangle \texttt{ \}} \\
\langle ClassBody \rangle & \rightarrow & (\langle FieldDeclaration \rangle \mid \langle MethodDeclaration \rangle)^* \\
\langle MethodDeclaration \rangle & \rightarrow & \langle Modifier \rangle^* \langle Type \rangle \langle Identifier \rangle \texttt{ (} \langle Parameters \rangle^? \texttt{)} \langle Block \rangle \\
\langle FieldDeclaration \rangle & \rightarrow & \langle Modifier \rangle^* \langle Type \rangle \langle Identifier \rangle \texttt{ (=} \langle Expression \rangle)^? \texttt{ ;} \\
\langle Parameters \rangle & \rightarrow & \langle Parameter \rangle \texttt{ (,} \langle Parameter \rangle)^* \\
\langle Parameter \rangle & \rightarrow & \langle Type \rangle \langle Identifier \rangle \\
\langle Modifier \rangle & \rightarrow & \texttt{public} \mid \texttt{private} \mid \texttt{protected} \\
 & & \mid \texttt{static} \mid \texttt{final} \\
\langle Type \rangle & \rightarrow & \texttt{int} \mid \texttt{double} \mid \texttt{boolean} \\
 & & \mid \texttt{char} \mid \texttt{String} \\
 & & \mid \texttt{void} \mid \texttt{jugurta} \mid \texttt{touati} \\
\langle Statement \rangle & \rightarrow & \langle IfStatement \rangle \mid \langle WhileStatement \rangle \mid \langle Block \rangle \\
 & & \mid \langle ExpressionStatement \rangle \mid \langle VariableDeclaration \rangle \\
 & & \mid \langle ReturnStatement \rangle \mid \langle MethodCallStatement \rangle \\
\langle IfStatement \rangle & \rightarrow & \texttt{if (} \langle Expression \rangle \texttt{)} \langle Statement \rangle \\
 & & (\texttt{else} \langle Statement \rangle)^? \\
\langle WhileStatement \rangle & \rightarrow & \texttt{while (} \langle Expression \rangle \texttt{)} \langle Statement \rangle \\
\langle Block \rangle & \rightarrow & \texttt{\{} \langle Statement \rangle^* \texttt{\}} \\
\langle ExpressionStatement \rangle & \rightarrow & \langle Expression \rangle \texttt{ ;} \\
\langle VariableDeclaration \rangle & \rightarrow & \langle Type \rangle \langle Identifier \rangle \texttt{ (=} \langle Expression \rangle)^? \texttt{ ;} \\
\langle ReturnStatement \rangle & \rightarrow & \texttt{return } \langle Expression \rangle^? \texttt{ ;} \\
\langle MethodCallStatement \rangle & \rightarrow & \langle MethodCall \rangle \texttt{ ;} \\
\langle Expression \rangle & \rightarrow & \langle Assignment \rangle \\
\langle Assignment \rangle & \rightarrow & \langle Identifier \rangle \texttt{ =} \langle Expression \rangle \\
 & & \mid \langle LogicalOr \rangle \\
\langle LogicalOr \rangle & \rightarrow & \langle LogicalAnd \rangle \texttt{ (||} \langle LogicalAnd \rangle)^* \\
\langle LogicalAnd \rangle & \rightarrow & \langle Equality \rangle \texttt{ (\&\&} \langle Equality \rangle)^* \\
\langle Equality \rangle & \rightarrow & \langle Comparison \rangle \texttt{ ((== | !=)} \langle Comparison \rangle)^* \\
\langle Comparison \rangle & \rightarrow & \langle Term \rangle \texttt{ ((< | > | <= | >=)} \langle Term \rangle)^* \\
\langle Term \rangle & \rightarrow & \langle Factor \rangle \texttt{ ((+ | -)} \langle Factor \rangle)^* \\
\langle Factor \rangle & \rightarrow & \langle Unary \rangle \texttt{ ((* | / | \%)} \langle Unary \rangle)^* \\
\langle Unary \rangle & \rightarrow & \texttt{(+ | - | ! | ++ | -)} \langle Unary \rangle \\
 & & \mid \langle Postfix \rangle \\
\langle Postfix \rangle & \rightarrow & \langle Primary \rangle \texttt{ (++ | -)}^? \\
\langle Primary \rangle & \rightarrow & \texttt{(} \langle Expression \rangle \texttt{)} \\
 & & \mid \langle Identifier \rangle \\
 & & \mid \langle Number \rangle \\
 & & \mid \langle StringLiteral \rangle \\
 & & \mid \langle MethodCall \rangle \\
\langle MethodCall \rangle & \rightarrow & \langle Identifier \rangle \texttt{ (.} \langle Identifier \rangle)^* \texttt{ (} \langle Arguments \rangle^? \texttt{)} \\
\langle Arguments \rangle & \rightarrow & \langle Expression \rangle \texttt{ (,} \langle Expression \rangle)^* \\
\langle Identifier \rangle & \rightarrow & [a-zA-Z\_][a-zA-Z0-9\_]^* \\
\langle Number \rangle & \rightarrow & [0-9]^+ \\
\langle StringLiteral \rangle & \rightarrow & \texttt{"} ([^{\wedge}\backslash n\backslash] \mid \backslash.)^* \texttt{"}
\end{array}
$$

3.7 Grammaire LL(1) Factorisée

$$
\begin{aligned}
\langle Program \rangle &\rightarrow \langle StatementList \rangle \\
\langle StatementList \rangle &\rightarrow \langle Statement \rangle \ \langle StatementList \rangle \mid \varepsilon \\
\langle Statement \rangle &\rightarrow \langle IfStmt \rangle \mid \langle WhileStmt \rangle \mid \langle Block \rangle \mid \langle ExprStmt \rangle \\
&\quad \mid \langle VarDecl \rangle \mid \langle ReturnStmt \rangle \mid \langle MethodCallStmt \rangle \\
\langle IfStmt \rangle &\rightarrow \texttt{if (} \langle Expression \rangle \texttt{)} \ \langle Statement \rangle \ \langle ElsePart \rangle \\
\langle ElsePart \rangle &\rightarrow \texttt{else} \ \langle Statement \rangle \mid \varepsilon \\
\langle WhileStmt \rangle &\rightarrow \texttt{while (} \langle Expression \rangle \texttt{)} \ \langle Statement \rangle \\
\langle Block \rangle &\rightarrow \texttt{\{} \ \langle StatementList \rangle \ \texttt{\}} \\
\langle ExprStmt \rangle &\rightarrow \langle Expression \rangle \ \texttt{;} \\
\langle VarDecl \rangle &\rightarrow \langle Type \rangle \ \langle Identifier \rangle \ \langle Init \rangle \ \texttt{;} \\
\langle Init \rangle &\rightarrow \texttt{=} \ \langle Expression \rangle \mid \varepsilon \\
\langle ReturnStmt \rangle &\rightarrow \texttt{return} \ \langle Expression \rangle \ \texttt{;} \\
\langle MethodCallStmt \rangle &\rightarrow \langle MethodCall \rangle \ \texttt{;} \\
\langle Expression \rangle &\rightarrow \langle Assignment \rangle \\
\langle Assignment \rangle &\rightarrow \langle LogicalOr \rangle \ \langle AssignOp \rangle \\
\langle AssignOp \rangle &\rightarrow \texttt{=} \ \langle Expression \rangle \mid \varepsilon \\
\langle LogicalOr \rangle &\rightarrow \langle LogicalAnd \rangle \ \langle LogicalOr' \rangle \\
\langle LogicalOr' \rangle &\rightarrow \texttt{||} \ \langle LogicalAnd \rangle \ \langle LogicalOr' \rangle \mid \varepsilon \\
\langle LogicalAnd \rangle &\rightarrow \langle Equality \rangle \ \langle LogicalAnd' \rangle \\
\langle LogicalAnd' \rangle &\rightarrow \texttt{\&\&} \ \langle Equality \rangle \ \langle LogicalAnd' \rangle \mid \varepsilon \\
\langle Equality \rangle &\rightarrow \langle Comparison \rangle \ \langle Equality' \rangle \\
\langle Equality' \rangle &\rightarrow (\texttt{==} \mid \texttt{!=}) \ \langle Comparison \rangle \ \langle Equality' \rangle \mid \varepsilon \\
\langle Comparison \rangle &\rightarrow \langle Term \rangle \ \langle Comparison' \rangle \\
\langle Comparison' \rangle &\rightarrow (\texttt{<} \mid \texttt{>} \mid \texttt{<=} \mid \texttt{>=}) \ \langle Term \rangle \ \langle Comparison' \rangle \mid \varepsilon \\
\langle Term \rangle &\rightarrow \langle Factor \rangle \ \langle Term' \rangle \\
\langle Term' \rangle &\rightarrow (\texttt{+} \mid \texttt{-}) \ \langle Factor \rangle \ \langle Term' \rangle \mid \varepsilon \\
\langle Factor \rangle &\rightarrow \langle Unary \rangle \ \langle Factor' \rangle \\
\langle Factor' \rangle &\rightarrow (\texttt{*} \mid \texttt{/} \mid \texttt{\%}) \ \langle Unary \rangle \ \langle Factor' \rangle \mid \varepsilon \\
\langle Unary \rangle &\rightarrow (\texttt{+} \mid \texttt{-} \mid \texttt{!} \mid \texttt{++} \mid \texttt{-}) \ \langle Unary \rangle \\
&\quad \mid \langle Postfix \rangle \\
\langle Postfix \rangle &\rightarrow \langle Primary \rangle \ \langle Postfix' \rangle \\
\langle Postfix' \rangle &\rightarrow \texttt{++} \mid \texttt{-} \mid \varepsilon \\
\langle Primary \rangle &\rightarrow \texttt{(} \ \langle Expression \rangle \ \texttt{)} \\
&\quad \mid \langle Identifier \rangle \ \langle PrimaryTail \rangle \\
&\quad \mid \langle Number \rangle \\
&\quad \mid \langle StringLiteral \rangle \\
&\quad \mid \langle MethodCall \rangle \\
\langle PrimaryTail \rangle &\rightarrow \texttt{(} \ \langle Arguments \rangle^{?} \ \texttt{)} \mid \varepsilon
\end{aligned}
$$

3.8 Tables FIRST

Non-terminal	Ensemble FIRST
⟨*Program*⟩	{ `if`, `while`, `{`, IDENTIFIER, NUMBER, STRING_LITERAL, `int`, `double`, `boolean`, `char`, `String`, `void`, `return`, EOF }
⟨*Statement*⟩	{ `if`, `while`, `{`, IDENTIFIER, NUMBER, STRING_LITERAL, `int`, `double`, `boolean`, `char`, `String`, `void`, `return` }
⟨*Expression*⟩	{ `(`, IDENTIFIER, NUMBER, STRING_LITERAL, `+`, `-`, `!`, `++`, `-` }
⟨*Term*⟩	FIRST(⟨*Expression*⟩)
⟨*Factor*⟩	FIRST(⟨*Expression*⟩)
⟨*Unary*⟩	{ `+`, `-`, `!`, `++`, `-`, `(`, IDENTIFIER, NUMBER, STRING_LITERAL }
⟨*Primary*⟩	{ `(`, IDENTIFIER, NUMBER, STRING_LITERAL }

Table 3.1 – Tables FIRST pour la grammaire LL(1)

3.9 Diagramme de Transition d'États du Parser

- **État START_PROGRAM**
  - token FIRST(ClassDeclaration) → PARSE_CLASS
  - token FIRST(Statement) → PARSE_STATEMENT
  - EOF → ACCEPT
- **État PARSE_STATEMENT**
  - if → PARSE_IF
  - while → PARSE_WHILE
  - { → PARSE_BLOCK
  - Type → PARSE_DECLARATION
  - IDENTIFIER → PARSE_IDENTIFIER_STMT
  - other → ERROR

3.10 Schéma syntaxique

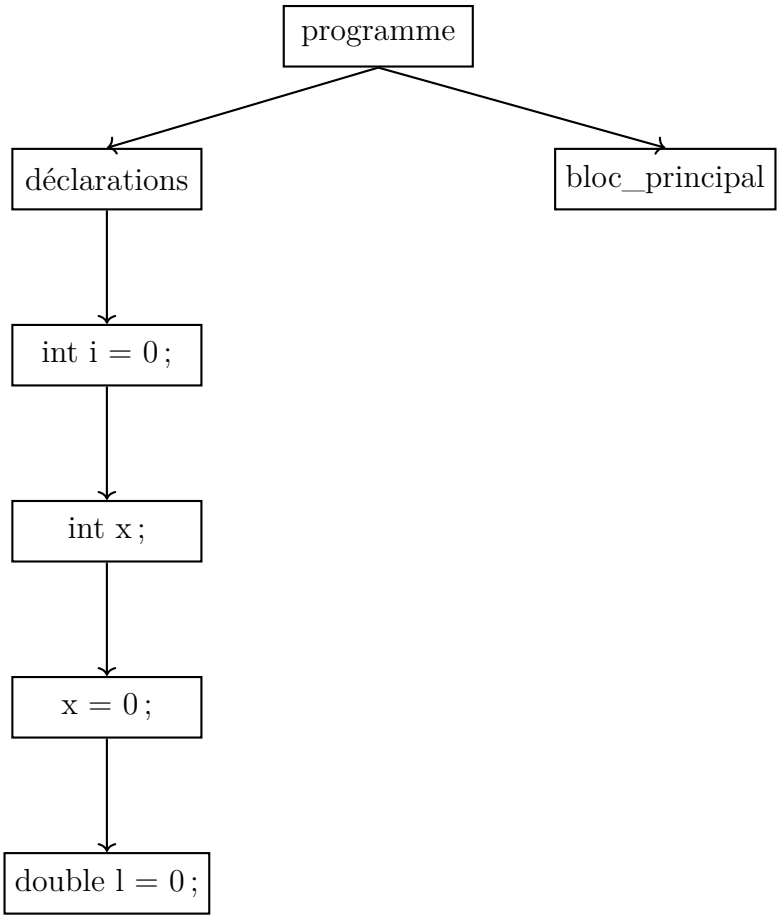


FIGURE 3.1 – Structure générale du programme

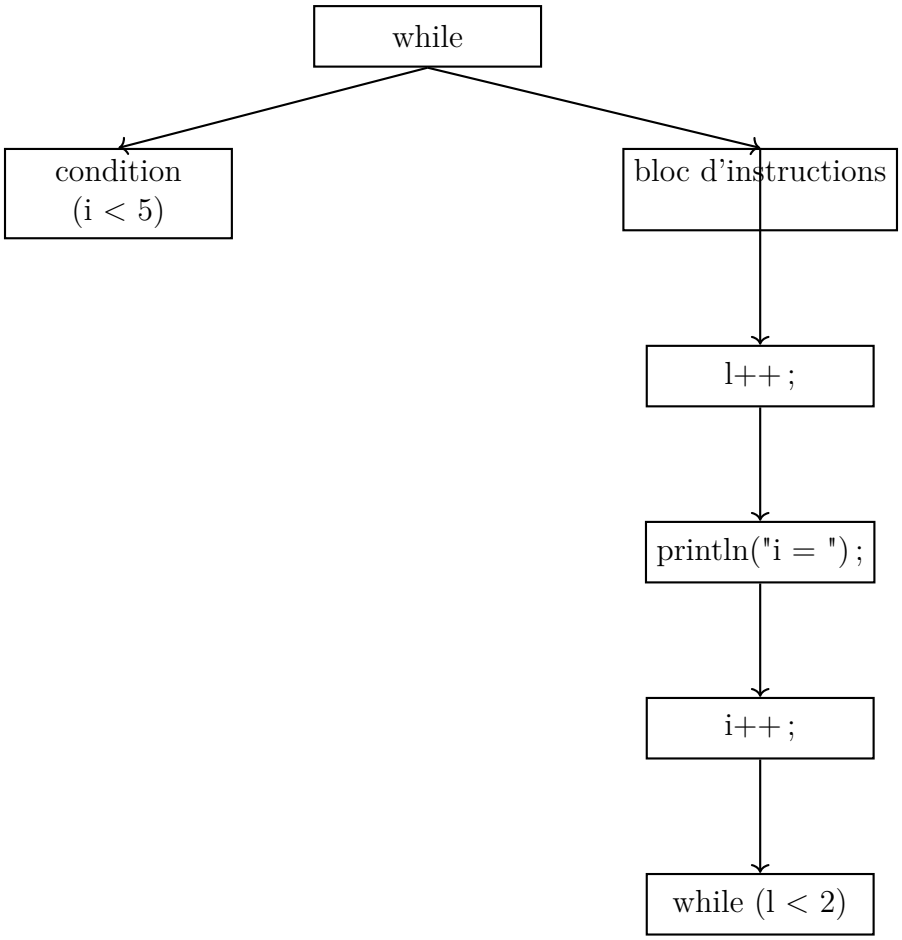
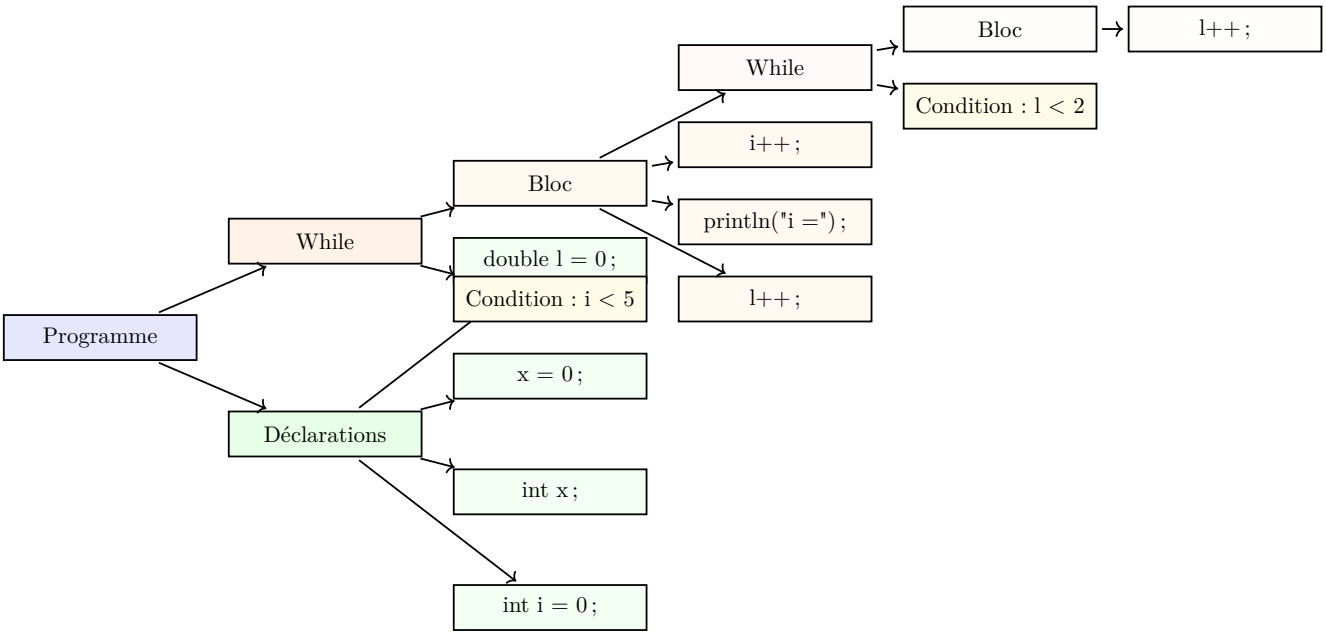


FIGURE 3.2 – Structure de la boucle while principale

3.11 Dérivation syntaxique complète

3.11.1 Arbre syntaxique simplifié



Programme	Racine de l'arbre syntaxique
Déclarations	Section des déclarations de variables
While	Structure de boucle principale
Condition	Tests booléens dans les boucles
While imbriqué	Boucle à l'intérieur d'une autre

Chapitre 4

La structure de votre projet

4.1 Structure du projet

La structure du projet `mini-compilateur-java` est organisée de manière à séparer le code source, les tests et la documentation. Voici l'arborescence des fichiers et dossiers :

```
mini-compilateur-java/

 .github/
    Contient la configuration GitHub (workflows, actions CI/CD, etc.)

 bin/
    Contient les fichiers compilés (.class) générés après la compilation du code Java

 doc/
    Documentation du projet (manuel, schémas, notes)

 mini-compilateur-java/
    Sous-projet ou ancien code (à préciser selon votre usage)

 scripts/
    Scripts utiles pour le projet (par exemple, scripts de compilation ou de test)

 src/
    lexical/
       Lexer.java      # Classe pour l'analyse lexicale du code source
       Token.java      # Classe représentant les tokens générés par le Lexer

    models/
       ASTNode.java    # Classe pour représenter les nœuds de l'arbre syntaxique abst

    syntax/
       Parser.java     # Classe pour l'analyse syntaxique (Parser) du code source

    Main.java           # Classe principale pour exécuter le compilateur
```

```
tests/
   WhileTest.java      # Programme de test pour vérifier le fonctionnement du compila

README.md
    Présentation générale du projet
```

4.1.1 Explication de l'architecture

- **.github/** : contient les configurations pour l'intégration continue ou la gestion du projet sur GitHub.
- **bin/** : dossier de sortie de compilation Java, où se trouvent les fichiers `.class`.
- **doc/** : documentation détaillée du projet, comme des diagrammes UML, manuels ou rapports.
- **scripts/** : scripts pour automatiser des tâches, comme la compilation ou l'exécution des tests.
- **src/** : dossier principal du code source Java.
  - **lexical/** : contient les classes du Lexer et des tokens pour l'analyse lexicale.
  - **models/** : contient les modèles de données, principalement l'AST (Abstract Syntax Tree).
  - **syntax/** : contient le Parser qui réalise l'analyse syntaxique.
  - **Main.java** : point d'entrée du programme pour lancer le compilateur.
- **tests/** : fichiers de test pour vérifier les fonctionnalités du compilateur (ex : `WhileTest.java`).
- **README.md** : fichier de présentation et d'explication du projet.

4.2 Cas de test

Pour vérifier le fonctionnement de notre mini-compilateur, nous avons utilisé le programme `WhileTest.java` situé dans le dossier `tests/`. Le code source est le suivant :

```
// Exemple de test dans tests/WhileTest.java
```

4.2.1 Étape 1 : Analyse lexicale

Le compilateur a identifié les tokens suivants :

```
[PUBLIC: 'public' @2:1]
[CLASS: 'class' @2:8]
[IDENTIFIER: 'WhileTest' @2:14]
[LBRACE: '{' @2:24]
...
[WHILE: 'while' @17:13]
[LPAREN: '(' @17:19]
[IDENTIFIER: 'l' @17:20]
[LESS: '<' @17:21]
[NUMBER: '2' @17:22]
[RPAREN: ')' @17:23]
```

```
[LBRACE: '{' @17:25]
[IDENTIFIER: 'l' @18:17]
[PLUS_PLUS: '++' @18:18]
[SEMICOLON: ';' @18:20]
[RBRACE: '}' @19:13]
[RBRACE: '}' @20:9]
[RBRACE: '}' @21:5]
[RBRACE: '}' @22:1]
```

Chaque mot-clé, identificateur, opérateur ou symbole a été reconnu et classé en token avec sa position dans le code.

4.2.2 Étape 2 : Analyse syntaxique

Lors de l'analyse syntaxique, les erreurs suivantes ont été détectées :
— Expected method name mais trouvé 'main' à la ligne 6
— Instruction non reconnue : `System`, `.`, `out`, `println`, `(`, `i = `, `)`, `;` à la ligne 13
Malgré ces erreurs, l'arbre syntaxique abstrait (AST) a pu être généré partiellement :

```
PROGRAM
 CLASS [WhileTest] (@2)
   MODIFIER [public] (@2)
   METHOD [main] (@6)
     RETURN_TYPE [void] (@6)
     MODIFIER [public] (@6)
     MODIFIER [static] (@6)
     DECLARATION [int i] (@7)
       NUMBER [0]
     DECLARATION [int x] (@8)
     ASSIGNMENT [x] (@9)
       NUMBER [0]
     DECLARATION [double l] (@10)
       NUMBER [0]
     WHILE (@11)
       CONDITION
         COMPARISON [<]
           IDENTIFIER [i]
           NUMBER [5]
       BODY
         BLOCK (@11)
           INCREMENT [l]
           INCREMENT [i]
           WHILE (@17)
             CONDITION
               COMPARISON [<]
                 IDENTIFIER [l]
                 NUMBER [2]
             BODY
               BLOCK (@17)
                 INCREMENT [l]
```

4.2.3 Conclusion du cas de test

Ce test montre que :

— Le **Lexer** fonctionne correctement et reconnaît tous les tokens du langage cible.

— Le **Parser** rencontre des difficultés avec certaines instructions comme `System.out.println` car elles ne sont pas encore prises en charge dans le mini-compilateur.

— L'AST généré montre que les structures de contrôle de base (`while`, déclarations, affectations) sont correctement reconnues.

Ce cas de test nous permet de vérifier les fonctionnalités de base et de détecter les limitations du compilateur, ce qui guidera les prochaines améliorations.

4.3 Conclusion générale

Le projet de mini-compilateur réalisé a permis de mettre en pratique les concepts fondamentaux de la compilation, notamment l'analyse lexicale, l'analyse syntaxique et la construction de l'arbre syntaxique abstrait (AST).

— L'**analyse lexicale** a été correctement implémentée et permet de reconnaître avec précision les mots-clés, identificateurs, opérateurs et symboles du langage cible.

— L'**analyse syntaxique** permet de détecter les erreurs dans la structure des programmes et de générer partiellement l'AST, ce qui constitue la base pour la génération de code ou l'interprétation future.

— Les tests réalisés, comme `WhileTest.java`, ont mis en évidence les fonctionnalités supportées (déclarations, affectations, boucles `while`) et les limitations actuelles du compilateur (instructions complexes non encore implémentées comme `System.out.println`).

En conclusion, ce projet a permis de comprendre et d'appliquer les étapes principales de la compilation d'un langage simplifié, et constitue une base solide pour de futures améliorations telles que :

— L'ajout de nouvelles structures de contrôle (`if`, `for`, etc.).

— La prise en charge des instructions d'entrée/sortie.

— La génération de code exécutable ou interprété.

Ainsi, ce mini-compilateur constitue un bon point de départ pour l'étude approfondie des compilateurs et des langages de programmation.