

Introduction to Language Theory and Compiling

---

# Lexer Compiler

---

Siddharth SAHAY

Jordi UGARTE

**Prof. Gilles GERAERTS**

2024



## Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Background . . . . .	1
	GILLES Grammar Rules . . . . .	2
	Explanation of Key Rules . . . . .	3
	Project Structure . . . . .	4
<b>2</b>	<b>Objective</b>	<b>5</b>
<b>3</b>	<b>Part 1</b>	<b>6</b>
3.1	Part 1 . . . . .	6
3.1.1	Lexical Analyzer . . . . .	6
3.1.2	Regular Expressions used . . . . .	7
3.1.3	Lexer Rules . . . . .	7
3.1.4	Symbol Table Management . . . . .	14
3.1.5	Error Handling . . . . .	15
<b>4</b>	<b>Testing</b>	<b>16</b>
4.1	Test Files . . . . .	16
4.2	Running the Tests . . . . .	16

## List of Tables

1.1	The GILLES grammar . . . . .	2
1.2	GILLES Parser project structure . . . . .	4
3.1	Regular expressions used . . . . .	7
3.2	Initialization of YYINITIAL state . . . . .	7
3.3	Initialization of YYINITIAL state in the lexer . . . . .	8
3.4	ProgName and transition to the CODE block . . . . .	8
3.5	ProgName and transition to the CODE block . . . . .	8
3.6	Implementation of Code Block in the lexer . . . . .	9
3.7	Condition block implementation . . . . .	10
3.8	Assignment of variables . . . . .	11
3.9	Implementation of assignment of variables . . . . .	11
3.10	Arithmetic expressions . . . . .	11
3.11	Implementation of arithmetic expressions . . . . .	12
3.12	Input and output operations . . . . .	12
3.13	Conditionals and loops . . . . .	13
3.14	Conditionals and loops implementation . . . . .	13
3.15	Input and output operations . . . . .	13
3.16	Input and output operations implementation . . . . .	14
3.17	Whitespaces and comments . . . . .	14
3.18	Whitespaces and comments example . . . . .	14
3.19	Exception in case of error in the lexer . . . . .	15
4.1	Test datasheet . . . . .	17

## **Abstract**

This project involves designing and developing a compiler for the language called *Genial Imperative Language for Learning and the Enlightenment of Students* (GILLES). Its grammar is stated under the Background section.

This report focuses on the initial phase of compiler development—the creation of a lexical analyzer using JFlex. The lexical analyzer is responsible for scanning GILLES source code, identifying lexical units, managing a symbol table, and handling errors. Through comprehensive testing with various GILLES programs, the lexical analyzer demonstrates robust tokenization and effective state transitions, laying a solid foundation for subsequent parts of the project.

## 1.1 Background

The primary objective of this project is to design and implement a compiler for GILLES. This report focuses on the initial phase of the compiler development, specifically the creation of a lexical analyzer using the JFlex tool. The lexical analyzer is responsible for scanning the source code, identifying lexical units, and managing a symbol table essential for further compilation stages.

## GILLES Grammar Rules

- [1]  $\langle Program \rangle \rightarrow \text{LET } [\text{ProgName}] \text{ BE } \langle Code \rangle \text{ END}$
- [2]  $\langle Code \rangle \rightarrow \langle Instruction \rangle : \langle Code \rangle$
- [3]  $\rightarrow \varepsilon$
- [4]  $\langle Instruction \rangle \rightarrow \langle Assign \rangle$
- [5]  $\rightarrow \langle If \rangle$
- [6]  $\rightarrow \langle While \rangle$
- [7]  $\rightarrow \langle Call \rangle$
- [8]  $\rightarrow \langle Output \rangle$
- [9]  $\rightarrow \langle Input \rangle$
- [10]  $\langle Assign \rangle \rightarrow [\text{VarName}] = \langle Expr Arith \rangle$
- [11]  $\langle Expr Arith \rangle \rightarrow [\text{VarName}]$
- [12]  $\rightarrow [\text{Number}]$
- [13]  $\rightarrow (\langle Expr Arith \rangle)$
- [14]  $\rightarrow -\langle Expr Arith \rangle$
- [15]  $\rightarrow \langle Expr Arith \rangle \langle Op \rangle \langle Expr Arith \rangle$
- [16]  $\langle Op \rangle \rightarrow +$
- [17]  $\rightarrow -$
- [18]  $\rightarrow *$
- [19]  $\rightarrow /$
- [20]  $\langle If \rangle \rightarrow \text{IF } \{ \langle Cond \rangle \} \text{ THEN } \langle Code \rangle \text{ END}$
- [21]  $\rightarrow \text{IF } \{ \langle Cond \rangle \} \text{ THEN } \langle Code \rangle \text{ ELSE } \langle Code \rangle \text{ END}$
- [22]  $\langle Cond \rangle \rightarrow \langle Cond \rangle \rightarrow \langle Cond \rangle$
- [23]  $\rightarrow | \langle Cond \rangle |$
- [24]  $\rightarrow \langle Expr Arith \rangle \langle Comp \rangle \langle Expr Arith \rangle$
- [25]  $\langle Comp \rangle \rightarrow ==$
- [26]  $\rightarrow <=$
- [27]  $\rightarrow <$
- [28]  $\langle While \rangle \rightarrow \text{WHILE } \{ \langle Cond \rangle \} \text{ REPEAT } \langle Code \rangle \text{ END}$
- [29]  $\langle Output \rangle \rightarrow \text{OUT}([\text{VarName}])$
- [30]  $\langle Input \rangle \rightarrow \text{IN}([\text{VarName}])$

Table 1.1: The GILLES grammar

## Explanation of Key Rules

- Program Declaration ([1]): Every GILLES program starts with the LET keyword, followed by a program name, the BE keyword, the code block, and concludes with END.
- Code Block ([2], [3]): The <Code> non-terminal allows for a sequence of instructions separated by colons (:). It is recursively defined to accommodate multiple instructions or an empty sequence.
- Instructions ([4]-[9]): The <Instruction> non-terminal can be an assignment, conditional (If), loop (While), function call (Call), output (Output), or input (Input).
- Arithmetic Expressions ([10]-[19]): These rules define how arithmetic expressions are constructed using variables, numbers, parentheses, negation, and binary operators (+, -, \*, /).
- Conditionals ([20]-[27]): The <If> and <While> constructs enable conditional execution and looping based on specified conditions.
- Input/Output ([29]-[30]): The language supports input (IN) and output (OUT) operations involving variables.

The following java files: `LexicalUnit.java` and `Symbol.java` were provided from the beginning as support for the assignment. The file `Main.java` file was developed later to run the lexical analyzer class to perform the tests. The project structure can be best described in the Project Structure section.

The source code is located in the `src` folder, where the java files will be compiled into classes by running the following command:

```
$ make
```

The previous command will also generate a `.jar` file called `part1.jar` to be runnable inside the `dist` folder. This `jar` file will run a `test.gls` file from the `test` folder. This can be run with the command:

```
$ make test
```

We can also run the below to run a specific test file:

```
$ make test TEST_FILE=test/FILENAME.gls
```

There are other commands that can be run to clean the project and generate the documentation.

## Project Structure

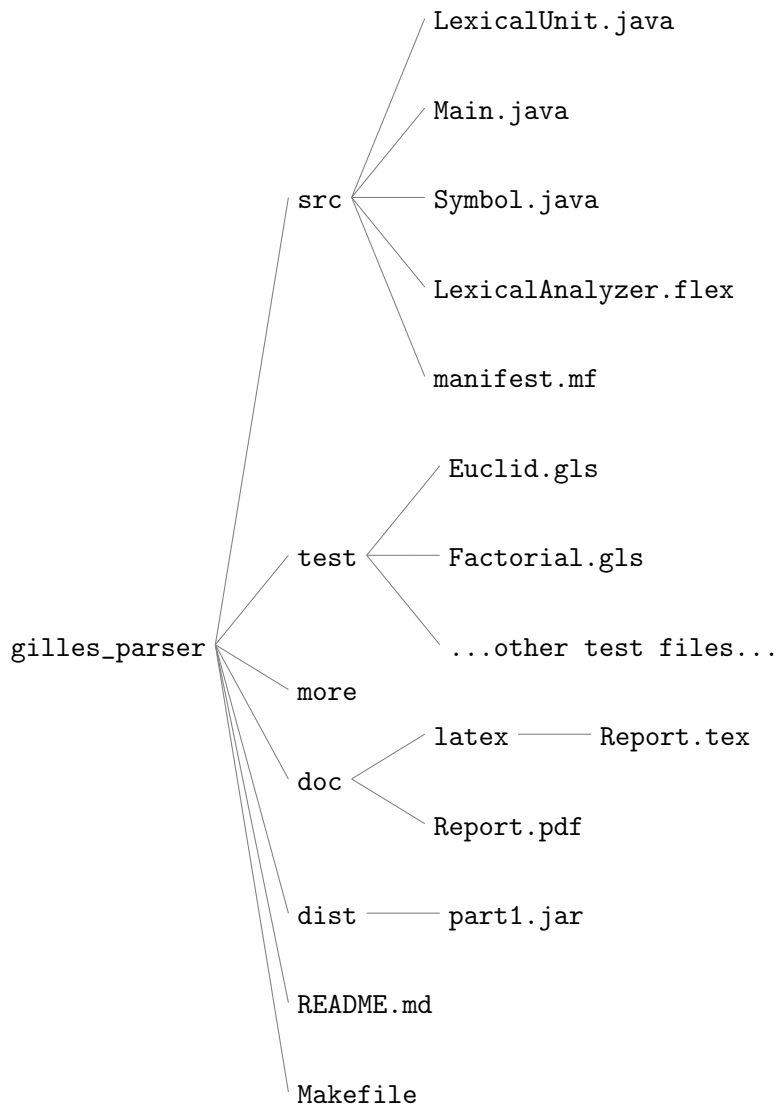


Table 1.2: GILLES Parser project structure



The objective of the project is to develop a compiler for the Genial Imperative Language for Learning and the Enlightenment of Students mentioned in the introduction. The project is separated in three parts:

1. Develop and design a lexical analyzer of the compiler by using JFlex.

*\*The other parts of the project were not yet specified by the date this was submitted (23/10/2024). They will be referenced and worked on by the time these other two are revealed.\**

## 3.1 Part 1

### 3.1.1 Lexical Analyzer

The lexical analyzer is responsible for scanning the source code, identifying lexical units, and managing a symbol table essential for further compilation stages. The lexical analyzer is implemented using JFlex, a lexical analyzer generator for Java. The JFlex file `LexicalAnalyzer.flex` contains the regular expressions that define the lexical units of the GILLES language. The JFlex tool is used to generate the Java source code for the lexical analyzer. The generated Java source code is then compiled and executed to analyze the source code of GILLES programs. The lexical analyzer reads the source code character by character, identifies the lexical units, and stores them in the symbol table. The symbol table is a data structure that stores information about the lexical units, such as their type, value, and position in the source code. The lexical analyzer also handles errors in the source code, such as invalid characters or tokens, and reports them to the user, if an unrecognised symbol is recognised we throw an "Unknown symbol detected" error. The lexical analyzer is of course the first essential component of the compiler, as it provides the input for the parser, which analyzes the syntactic structure of the source code.

### 3.1.2 Regular Expressions used

Following the grammar rules, the following regular expressions were used to define the lexical units of the GILLES language:

ProgName =  $[a-zA-Z\_][a-zA-Z0-9\_]*$

VarName =  $[a-zA-Z\_][a-zA-Z0-9\_]*$

Number =  $[0-9]^+$

Whitespace =  $[\t\r\n]^+$

Comment =  $"!!"[\n]^*$

ShortComment =  $"\$"[\n]^*$

Table 3.1: Regular expressions used

These are relatively straightforward RegExps that identify the different types of tokens in the GILLES language. We will see how they're used in the JFlex file in the next section.

### 3.1.3 Lexer Rules

This section describes the main rules of the analyzer and how we transition from one block to another.

Here we can see the main rules of the lexical analyzer:

#### 3.1.3.1 The initial LET Statement

The lexer begins in the YYINITIAL state, which captures the declaration of a program using the LET keyword. Upon encountering LET, the lexer transitions to the PROGRAM state, where the program name is expected, as per the grammar rule:

1	<hr/>
	<code>&lt;Program&gt; → LET [ProgName] BE &lt;Code&gt; END</code>
	<hr/>

Table 3.2: Initialization of YYINITIAL state

In the lexer, this is represented as:

1	<hr/>
	<code>"LET" { yybegin(PROGRAM);</code>
2	<code>System.out.println(new Symbol(LexicalUnit.LET, yyline, yycolumn, yytext())); }</code>
	<hr/>

Table 3.3: Initialization of YYINITIAL state in the lexer

Once LET is matched, the lexer moves to the PROGRAM state where it expects the program name ([ProgName]), ensuring adherence to the grammar rule that requires a program name after LET.

### 3.1.3.2 ProgName and transition to the CODE block

After matching the program name in the PROGRAM state, the lexer transitions to the CODE state upon encountering the BE keyword, as required by the grammar:

---

1	<Code>	→	<Instruction>	:	<Code>
2	<Instruction>	→	<Assign>		
3		→	<If>		
4		→	<While>		
5		→	<Call>		
6		→	<Output>		
7		→	<Input>		

---

Table 3.4: ProgName and transition to the CODE block

---

```

1 {ProgName} { yybegin(CODE);
2 System.out.println(new Symbol(LexicalUnit.PROGNAME, yyline, yycolumn, yytext())); }

```

---

Table 3.5: ProgName and transition to the CODE block

The instruction block has been broken down into the different types of blocks directly rather than one by one for simplicity.

### 3.1.3.3 Code Block

The following is the implementation of the CODE block in the lexer:

---

```

1  /* Code block state */
2  <CODE> {
3      {Whitespace}+ { /* Ignore whitespace */ }
4      {Comment} { /* Ignore comments */ }
5      {ShortComment} { /* Ignore short comments */ }
6      "BE" { System.out.println(new Symbol(LexicalUnit.BE, yyline, yycolumn, yytext())); }
7      "END" {
8          yybegin(CODE);

```

---

```

9      System.out.println(
10          new Symbol(LexicalUnit.END, yyline, yycolumn, yytext()));
11  }
12  ":" { System.out.println(new Symbol(LexicalUnit.COLUMN, yyline, yycolumn, yytext())); }
13  {VarName}"="
14  { yybegin(ARITHMETIC); System.out.println(new Symbol(LexicalUnit.ASSIGN, yyline, yycolumn, yytext())); }
15  "=="
16  { yybegin(ARITHMETIC); System.out.println(new Symbol(LexicalUnit.ASSIGN, yyline, yycolumn, yytext())); }
17  "IN"
18  { yybegin(INPUT_OUTPUT); System.out.println(new Symbol(LexicalUnit.INPUT, yyline, yycolumn, yytext())); }
19  "OUT"
20  { yybegin(INPUT_OUTPUT); System.out.println(new Symbol(LexicalUnit.OUTPUT, yyline, yycolumn, yytext())); }
21  "IF"
22  { yybegin(CONDITION); System.out.println(new Symbol(LexicalUnit.IF, yyline, yycolumn, yytext())); }
23  "WHILE"
24  { yybegin(CONDITION); System.out.println(new Symbol(LexicalUnit.WHILE, yyline, yycolumn, yytext())); }
25  "ELSE"
26  { System.out.println(new Symbol(LexicalUnit.ELSE, yyline, yycolumn, yytext())); }
27  {VarName} {
28      if (!variables.containsKey(yytext())) {
29          variables.put(yytext(), yyline+1);
30      }
31      System.out.println(new Symbol(LexicalUnit.VARNAME, yyline, yycolumn, yytext()));
32  }
33  "(" { System.out.println(new Symbol(LexicalUnit.LPAREN, yyline, yycolumn, yytext())); }
34  ")" { System.out.println(new Symbol(LexicalUnit.RPAREN, yyline, yycolumn, yytext())); }
35  }

```

---

Table 3.6: Implementation of Code Block in the lexer

Since there is a recursive definition of the Code block, the lexer transitions back to the CODE state after processing each instruction. This allows the lexer to handle multiple instructions in a program, as defined by the grammar.

We have also left out the Call block as it is not used in the grammar.

The "ELSE" keyword is also handled in the CODE block since it is not a separate block in the grammar. Once this has been parsed, the lexer should stay in the CODE block.

### 3.1.3.4 Condition Block

The lexer transitions to the CONDITION state upon encountering the IF or WHILE keywords, as defined in the grammar:

---

```

1  <If> → IF { <Cond> } THEN <Code> END
2      → IF { <Cond> } THEN <Code> ELSE <Code> END
3  <While> → WHILE {<Cond>} REPEAT <Code> END

```

---

```

1  /* Conditionals and loops */
2  <CONDITION> {
3      "THEN" { yybegin(CODE); System.out.println(new Symbol(LexicalUnit.THEN, yyline, yycolumn, yytext())); }
4      "ELSE" { yybegin(CODE); System.out.println(new Symbol(LexicalUnit.ELSE, yyline, yycolumn, yytext())); }
5      "REPEAT" { yybegin(CODE); System.out.println(new Symbol(LexicalUnit.REPEAT, yyline, yycolumn, yytext())); }
6      "END" { yybegin(CODE); System.out.println(new Symbol(LexicalUnit.END, yyline, yycolumn, yytext())); }
7      {Whitespace} { /* Ignore whitespace */ }
8      {Number} { System.out.println(new Symbol(LexicalUnit.NUMBER, yyline, yycolumn, yytext())); }
9      {VarName} {
10         if (!variables.containsKey(yytext())) {
11             variables.put(yytext(), yyline+1);
12         }
13         System.out.println(new Symbol(LexicalUnit.VARNAME, yyline, yycolumn, yytext()));
14     }
15     "==" { System.out.println(new Symbol(LexicalUnit.EQUAL, yyline, yycolumn, yytext())); }
16     "<=" { System.out.println(new Symbol(LexicalUnit.SMALEQ, yyline, yycolumn, yytext())); }
17     "<" { System.out.println(new Symbol(LexicalUnit.SMALLER, yyline, yycolumn, yytext())); }
18     "|" { System.out.println(new Symbol(LexicalUnit.PIPE, yyline, yycolumn, yytext())); }
19     "->" { System.out.println(new Symbol(LexicalUnit.IMPLIES, yyline, yycolumn, yytext())); }
20     "(" { System.out.println(new Symbol(LexicalUnit.LPAREN, yyline, yycolumn, yytext())); }
21     ")" { System.out.println(new Symbol(LexicalUnit.RPAREN, yyline, yycolumn, yytext())); }
22     ":" { System.out.println(new Symbol(LexicalUnit.COLUMN, yyline, yycolumn, yytext())); }
23     "{" { System.out.println(new Symbol(LexicalUnit.LBRACK, yyline, yycolumn, yytext())); }
24     "}" { System.out.println(new Symbol(LexicalUnit.RBRACK, yyline, yycolumn, yytext())); }
25 }

```

---

Table 3.7: Condition block implementation

Notice also here that we do not have `>` (GREATER\_THAN) as a token, this is because it is not used in the grammar (possibly left out intentionally?).

### 3.1.3.5 Assignment Statements

The assignment of values to variables is handled by recognizing both `VarName = ExprArith` as defined in the grammar:

---

1	<code>&lt;Assign&gt; → [VarName] = &lt;ExprArith&gt;</code>
---	---

---

Table 3.8: Assignment of variables

This rule is implemented in the lexer as:

---

1	<code>{VarName}"=" { yybegin(ARITHMETIC);</code>
2	<code>System.out.println(new Symbol(LexicalUnit.ASSIGN, yyline, yycolumn, yytext())); }</code>

---

Table 3.9: Implementation of assignment of variables

Upon matching an assignment, the lexer transitions to the ARITHMETIC state to evaluate arithmetic expressions.

### 3.1.3.6 Arithmetic Expressions

Arithmetic expressions (`ExprArith`) are handled in the ARITHMETIC state. The lexer recognizes variable names, numbers, and operators (+, -, \*, /), as well as parentheses for grouping, as defined by the grammar:

---

1	<code>&lt;ExprArith&gt; → [VarName]</code>
2	<code>→ [Number]</code>
3	<code>→ ( &lt;ExprArith&gt; )</code>
4	<code>→ - &lt;ExprArith&gt;</code>
5	<code>→ &lt;ExprArith&gt; &lt;Op&gt; &lt;ExprArith&gt;</code>

---

Table 3.10: Arithmetic expressions

In the lexer, this is implemented as follows:

---

1	<code><i>/* Arithmetic expressions */</i></code>
2	<code>&lt;ARITHMETIC&gt; {</code>
3	<code>  {Whitespace} { <i>/* Ignore whitespace */</i> }</code>
4	<code>  {Number} { System.out.println(new Symbol(LexicalUnit.NUMBER, yyline, yycolumn, yytext())); }</code>
5	<code>  {VarName} {</code>
6	<code>    if (!variables.containsKey(yytext())) {</code>
7	<code>      variables.put(yytext(), yyline+1);</code>

---

```

8         }
9         System.out.println(new Symbol(LexicalUnit.VARNAME, yyline, yycolumn, yytext()));
10    }
11    "+" { System.out.println(new Symbol(LexicalUnit.PLUS, yyline, yycolumn, yytext())); }
12    "-" { System.out.println(new Symbol(LexicalUnit.MINUS, yyline, yycolumn, yytext())); }
13    "*" { System.out.println(new Symbol(LexicalUnit.TIMES, yyline, yycolumn, yytext())); }
14    "/" { System.out.println(new Symbol(LexicalUnit.DIVIDE, yyline, yycolumn, yytext())); }
15    "(" { System.out.println(new Symbol(LexicalUnit.LPAREN, yyline, yycolumn, yytext())); }
16    ")" { yybegin(CODE); System.out.println(new Symbol(LexicalUnit.RPAREN, yyline, yycolumn,
17 yytext())); }
18    ":" { yybegin(CODE); System.out.println(new Symbol(LexicalUnit.COLUMN, yyline, yycolumn,
19 yytext())); }
20    }
21

```

---

Table 3.11: Implementation of arithmetic expressions

The arithmetic expressions are evaluated based on the sequence of operators and operands, this might be useful later when we need to evaluate the expressions.

### 3.1.3.7 Input and Output

Input and output operations are defined in the grammar as follows. The ) is used to move back to the CODE block since this rule has been collapsed into the CODE block instead of having it's own "Output" block.

---

```

1  /* Input/Output instructions */
2  <INPUT_OUTPUT> {
3      {Whitespace} { /* Ignore whitespace */ }
4      "(" { System.out.println(new Symbol(LexicalUnit.LPAREN, yyline, yycolumn, yytext())); }
5  =  ")" { yybegin(CODE); System.out.println(new Symbol(LexicalUnit.RPAREN, yyline, yycolumn, yytext())); }
6      {VarName} {
7          if (!variables.containsKey(yytext())) {
8              variables.put(yytext(), yyline+1);
9          }
10         System.out.println(new Symbol(LexicalUnit.VARNAME, yyline, yycolumn, yytext()));
11     }
12     ":" { System.out.println(new Symbol(LexicalUnit.COLUMN, yyline, yycolumn, yytext())); }
13 }

```

---

Table 3.12: Input and output operations



### 3.1.3.8 Conditionals and Loops

Conditionals and loops are defined in the grammar as follows:

---

1	<If> → IF { <Cond> } THEN <Code> END
2	<While> → WHILE {<Cond>} REPEAT <Code> END

---

Table 3.13: Conditionals and loops

In the lexer, the IF and WHILE keywords trigger a transition to the CONDITION state where the conditional expressions are evaluated:

---

1	"IF" { yybegin(CONDITION);
2	System.out.println(new Symbol(LexicalUnit.IF, yyline, yycolumn, yytext())); }
3	"WHILE" { yybegin(CONDITION);
4	System.out.println(new Symbol(LexicalUnit.WHILE, yyline, yycolumn, yytext())); }

---

Table 3.14: Conditionals and loops implementation

The CONDITION state then handles comparison operators (==, <=, <) and conditional expressions (VarName, Number, ExprArith), adhering to the grammar's definitions for conditions.

### 3.1.3.9 Input and Output

Input and output operations are defined as:

---

1	<Output> → OUT([VarName])
2	<Input> → IN([VarName])

---

Table 3.15: Input and output operations

In the lexer, these are handled in the INPUT\_OUTPUT state:

---

1	"IN" { yybegin(INPUT_OUTPUT);
2	System.out.println(new Symbol(LexicalUnit.INPUT, yyline, yycolumn, yytext())); }
3	"OUT" { yybegin(INPUT_OUTPUT);
4	System.out.println(new Symbol(LexicalUnit.OUTPUT, yyline, yycolumn, yytext())); }

---

This implementation captures the IN and OUT keywords, ensuring that the correct symbols are generated in accordance with the grammar.

Table 3.16: Input and output operations implementation

### 3.1.3.10 Whitespace and Comments

The lexer is designed to ignore unnecessary whitespace and comments. Both multi-line (`{Comment}`) and single-line (`{ShortComment}`) comments are handled across different states to prevent irrelevant tokens from being processed. This aligns with the GILLES grammar, which does not consider whitespace and comments as significant parts of the language.

---

```

1  /* Ignore whitespace and comments */
2  {Whitespace} { /* Ignore whitespace */ }
3  {Comment} { /* Ignore comments */ }
4  {ShortComment} { /* Ignore short comments */ }

```

---

Table 3.17: Whitespaces and comments

This ensures that the parsing of valid tokens, such as program names or arithmetic expressions, is unaffected by whitespace or comments.

The reason that nested comments are not supported is that it is explicitly not part of the grammar, but also because nested comments are challenging to track, i.e., it is difficult to accurately track when a comment ends.

For example, if your comment structure looks like `!!` to start and `!!` to end, the difficulty arises when we encounter something like this:

---

```

1  !! This is a 1st level comment
2  !! Nested 2nd level comment !!
3  1st level comment again !! $ Not a valid comment since it already ended above!

```

---

Table 3.18: Whitespaces and comments example

In the above example, the lexer would need to keep track of the nested comments and ensure that it doesn't stop at the first `!!` it encounters. The solution would possibly be to introduce a stack to keep track of the nested comments but that seems like an overkill for the current simplistic grammar.

### 3.1.4 Symbol Table Management

The symbol table is implemented as a `HashMap` in Java, where each variable name (`VarName`) is mapped to its corresponding line number in the source code where it first appears. When a variable is encountered

in the source code, the lexer checks if it is already present in the symbol table. If not, the variable name and its line number are added to the symbol table.

### 3.1.5 Error Handling

The lexer is designed to detect and report errors gracefully. When an unrecognized symbol is encountered or a symbol is encountered in the incorrect rule, the lexer throws an error with the message "Unknown symbol detected".

For example, if the lexer encounters the > symbol, which is not defined in the current grammar, it will output:

---

```
1 Exception in thread "main" java.lang.Error: Unknown symbol detected 6, column 12: '>'
2   at LexicalAnalyzer.yylex(LexicalAnalyzer.java:765)
3   at Main.main(Main.java:11)
4 make: *** [Makefile:45: test] Error 1
```

---

Table 3.19: Exception in case of error in the lexer

To test the lexical analyzer, we have provided a set of test files in the `test` folder. These test files contain GILLES programs that cover various aspects of the language. We've tried to incorporate different kinds of edge cases and code structures to ensure that the lexical analyzer can handle a wide range of inputs.

## 4.1 Test Files

- `Euclid.gls` - A simple program to calculate the greatest common divisor of two numbers.
- `InvalidSymbolEuclid.gls` - Same as above but with an invalid symbol, `.`
- `Sum.gls` - A simple program to calculate the sum of two numbers.
- `ThreeLoopGibberish.gls` - An unnecessarily complex program to test the lexer.
- `InvalidAssignment.gls` - A program with invalid syntax to test error handling.
- `ComplexAssignment.gls` - A program with complex arithmetic expression.
- `Fibonacci.gls` - A program to calculate the factorial of a number.
- `Whitespace.gls` - Random whitespace to test whitespace handling.

## 4.2 Running the Tests

To run the tests, we can use the following command, output will be displayed on the console:

```
$ make test TEST_FILE=test/Sum.gls
```

---

1	token: LET	lexical unit: LET
2	token: sum	lexical unit: PROGNAME
3	token: BE	lexical unit: BE

4	token: IN	lexical unit: INPUT
5	token: (	lexical unit: LPAREN
6	token: a	lexical unit: VARNAME
7	token: )	lexical unit: RPAREN
8	token: :	lexical unit: COLUMN
9	token: IN	lexical unit: INPUT
10	token: (	lexical unit: LPAREN
11	token: b	lexical unit: VARNAME
12	token: )	lexical unit: RPAREN
13	token: :	lexical unit: COLUMN
14	token: c	lexical unit: VARNAME
15	token: =	lexical unit: ASSIGN
16	token: a	lexical unit: VARNAME
17	token: +	lexical unit: PLUS
18	token: b	lexical unit: VARNAME
19	token: :	lexical unit: COLUMN
20	token: END	lexical unit: END
21	token: :	lexical unit: COLUMN
22	token: OUT	lexical unit: OUTPUT
23	token: (	lexical unit: LPAREN
24	token: c	lexical unit: VARNAME
25	token: )	lexical unit: RPAREN
26	token: :	lexical unit: COLUMN
27	token: END	lexical unit: END
28		
29	Variables	
30	a	4
31	b	5
32	c	6

---

Table 4.1: Test datasheet

[thy] thyssenkrupp (n.d.). "Bagger 288 – a giant among bucket wheel excavators". In: (). URL: <https://www.thyssenkrupp-industrial-solutions.com/en/products-and-services/mining-systems/bucket-wheel-excavators/bagger-288>.