

---

---

# Lexer Parser Compiler

---

---

Siddharth SAHAY  
Jordi UGARTE

Prof. GERAERTS Gilles  
BALACHANDER Mrudula  
SASSOLAS Mathieu

## Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Background . . . . .	1
	GILLES Grammar Rules . . . . .	1
	Highlighting Key Rules . . . . .	2
	Project Structure . . . . .	3
<b>2</b>	<b>Objective</b>	<b>4</b>
<b>3</b>	<b>Part 1 - The Lexer</b>	<b>5</b>
3.1	Part 1 . . . . .	5
3.1.1	Lexical Analyzer . . . . .	5
3.2	To Be, Or Not To Be... Stateful . . . . .	5
3.2.1	Regular Expressions used . . . . .	6
3.2.2	Lexer Rules . . . . .	7
3.2.3	Symbol Table Management . . . . .	9
3.2.4	Error Handling . . . . .	9
<b>4</b>	<b>Part 2 - The Parser</b>	<b>10</b>
4.1	Transforming the GILLES grammar . . . . .	10
4.2	First and Follow Sets . . . . .	12
4.2.1	First <sup>1</sup> sets . . . . .	12
4.2.2	Follow <sup>1</sup> sets . . . . .	13
4.3	LL(1) Action Table . . . . .	14
4.4	Parser Implementation . . . . .	14
4.4.1	Overview of the Parser . . . . .	14
4.4.2	Initialization and Setup . . . . .	15
4.4.3	Parsing Entry Point . . . . .	16
4.4.4	Some Parsing Methods . . . . .	16
4.4.5	ifStatement() . . . . .	18

<b>5</b>	<b>Testing</b>	<b>20</b>
5.1	Test Files . . . . .	20
5.2	Running the Tests . . . . .	20
5.3	Output of the Parser . . . . .	21
5.4	Generated ParseTree . . . . .	22

## List of Tables

1.1	Production Rules, The GILLES grammar . . . . .	1
1.2	GILLES Parser Project Structure . . . . .	3
3.1	Regular expressions used . . . . .	6
3.2	Whitespaces and comments example . . . . .	9
3.3	Exception in case of error in the lexer . . . . .	9
4.1	Priority and associativity of the GILLES operators . . . . .	10
4.2	Modified Grammar for <ExprArith> . . . . .	11
4.3	Non-Left-Recursive Grammar for <Cond> . . . . .	11
4.4	Modified grammar rules for GILLES . . . . .	13
4.5	LL(1) Parsing table (first part) . . . . .	15
4.6	LL(1) Parsing table (second part) . . . . .	15
5.1	Lexer output for Part 1 . . . . .	21
5.2	Parser output for Part 2 . . . . .	21

## **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

Rule No.	Production Rule
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	$\rightarrow \epsilon$
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	$\langle Assign \rangle \rightarrow [\text{VarName}] = \langle ExprArith \rangle$
11	$\langle ExprArith \rangle \rightarrow [\text{VarName}]$
12	$\rightarrow [\text{Number}]$
13	$\rightarrow (\langle ExprArith \rangle)$
14	$\rightarrow -(\langle ExprArith \rangle)$
15	$\rightarrow \langle ExprArith \rangle \langle Op \rangle \langle ExprArith \rangle$
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	$\langle Cond \rangle \rightarrow \langle Cond \rangle$
23	$\rightarrow   \langle Cond \rangle  $
24	$\rightarrow \langle ExprArith \rangle \langle Comp \rangle \langle ExprArith \rangle$
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: Production Rules, The GILLES grammar

## Highlighting 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 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

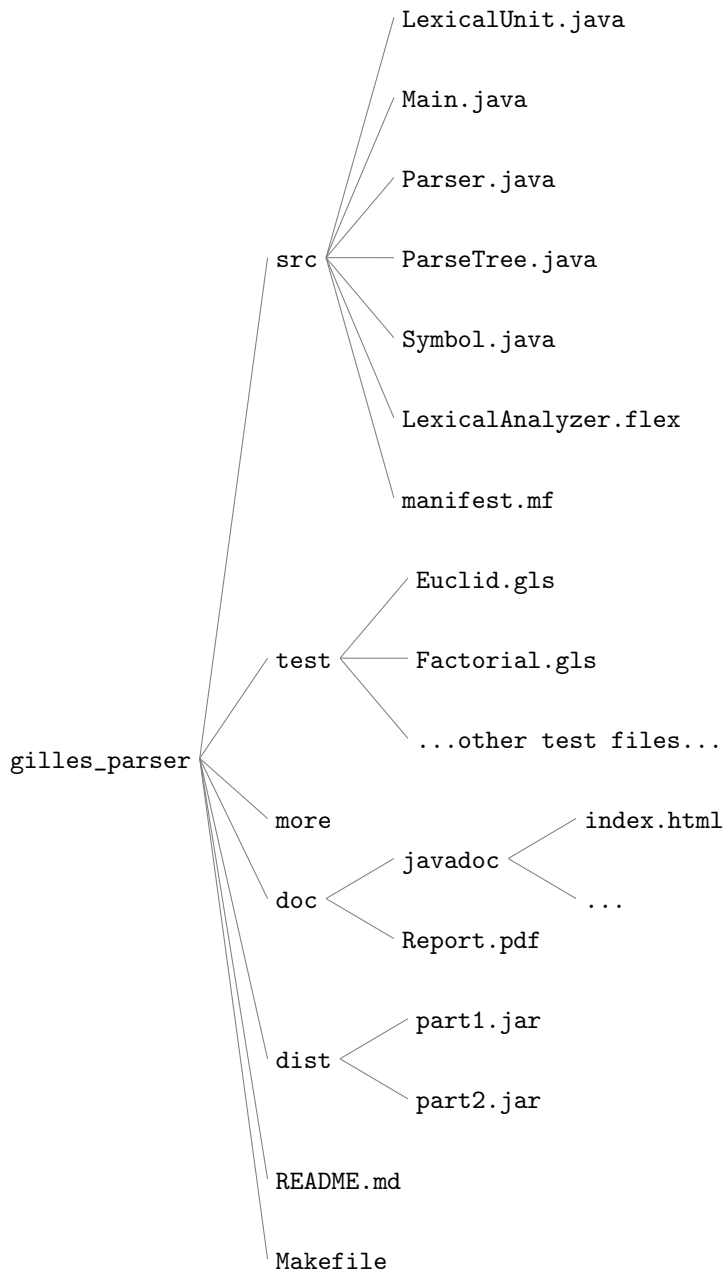


Table 1.2: GILLES Parser Project Structure



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.
2. Develop and design a recursive descent LL(1) parser for the Gilles compiler.

*\*The last part of the project was not yet specified by the date this was submitted (24/11/2024). They will be referenced and worked on by the time these other two are revealed.\**

## 3.1 Part 1

### 3.1.1 Lexical Analyzer

A 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 for this project will be 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. It 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, it throws 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.

## 3.2 To Be, Or Not To Be... Stateful

While developing this first part of the parser, the lexer, we came across a debate between stateful vs. stateless lexers. The decision to implement a stateful lexer arose from the need to handle context-sensitive tokens, manage nested constructs such as conditionals and loops, and simplify the parser's workload by ensuring correct tokenization based on context.

In the first part of the project, we made the deliberate decision to maintain state: a stateful lexer could manage context-sensitive tokens, like `ELSE` and `END`, ensuring they were recognized only within specific blocks to prevent misinterpretation. This approach also allowed us to handle complex structures, such as nested conditionals, by switching modes to differentiate normal code from special cases. By resolving some context-specific tokenization at the lexical stage, we aimed to reduce parser complexity, enabling it to focus more on syntax. Additionally, a stateful lexer provided early error detection, catching issues upfront and enforcing partial rule compliance for better performance.

We later recognised that there was a need to simplify these states once work shifted to the parser - we revisited this

decision in the second part of the project to address concerns over what the role of the lexer should really be. The conclusion was that the lexer should really only concern itself with the production of tokens, rather than implementing grammar rules. There are a few reasons why: simplifying the lexer to produce tokens only—without enforcing grammar rules—clarified the division of responsibilities: the lexer handles tokenization, and the parser enforces structure. This approach reduces redundancy, maintains modularity, and streamlines error handling, allowing the parser to manage syntax more effectively without extra state management in the lexer.

### 3.2.1 Regular Expressions used

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

```

AlphaUpperCase = [A – Z]
AlphaLowerCase = [a – z]
Alpha = [A – Z a – z]
Numeric = [0 – 9]
AlphaNumeric = [A – Z a – z 0 – 9]
BadInteger = (0[0 – 9]+)
Integer = ([1 – 9][0 – 9]*)|0
ProgName = [A – Z]([A – Z a – z]|"_")*
VarName = [a – z]([A – Z a – z 0 – 9])*
LineFeed = "\n"
CarriageReturn = "\r"
EndLine = (LineFeedCarriageReturn?)(CarriageReturnLineFeed?)
Space = ("t"|"f"|"")
Spaces = (Space)+
Separator = (Spaces)|(EndLine)
Any = ("n"|"r")*
UpToEnd = (AnyEndLine)|(EndLine)

```

Table 3.1: Regular expressions used

These regular expressions identify different token types in the GILLES language, matching the specified grammar rules. We will see their application in the JFlex file in the next section.

### 3.2.2 Lexer Rules

The following section outlines the primary rules used by the lexer to generate tokens in the GILLES language. In this implementation, the lexer does not transition between states; instead, it processes each token individually, generating symbols as defined by the grammar.

#### 3.2.2.1 Token Identification

The lexer processes tokens in a specific order to match keywords, operators, and identifiers based on the grammar rules. This ensures each token type is accurately captured before proceeding to the next. Below is an overview of the key tokens identified:

- **Keywords:** Keywords such as LET, BE, END, IF, THEN, ELSE, WHILE, REPEAT, IN, and OUT are recognized directly by the lexer and generate corresponding symbols.
- **Identifiers:** Program names and variable names are identified using regular expressions for ProgName and VarName. Both follow the conventions defined by the grammar.
- **Numbers:** Integer literals are captured, with warnings generated for numbers with leading zeros.
- **Operators and Delimiters:** Arithmetic operators (+, -, \*, /), logical operators (==, <=, <, ->), and code delimiters (such as parentheses, brackets, and colons) are matched and processed in a specific order.
- **Comments:** Comments are handled using two types: long comments, initiated by !! and closed by another !!, and short comments, initiated by \$ and extending to the end of the line.

#### 3.2.2.2 Order of Token Processing

The order of token processing follows the code structure to ensure that keywords and symbols are recognized first, followed by identifiers and operators. Any unmatched tokens produce an error, ensuring robust error handling within the lexer.

---

```
1 <LONGCOMMENTS> {
2 // End of comment
3     "!!"                {yybegin(YYINITIAL);} // go back to analysis
4     <<EOF>>              {throw new PatternSyntaxException("A comment is never closed.", yytext(), yyline);}
5     [^]                  {} //ignore any character
6 }
7
8 <YYINITIAL> {
9 // Comments
10    "!!"                {yybegin(LONGCOMMENTS);} // go to ignore mode
11    "$"{UpToEnd}        {} // go to ignore mode
12 // Code delimiters
13    "LET"                {return new Symbol(LexicalUnit.LET, yyline, yycolumn, yytext());}
14    "BE"                 {return new Symbol(LexicalUnit.BE, yyline, yycolumn, yytext());}
15    "END"                {return new Symbol(LexicalUnit.END, yyline, yycolumn, yytext());}
16    ":"                 {return new Symbol(LexicalUnit.COLUMN, yyline, yycolumn, yytext());}
17 // Assignment
```

```

18     "="                {return new Symbol(LexicalUnit.ASSIGN, yyline, yycolumn, yytext());}
19 // Parenthesis
20     "("                {return new Symbol(LexicalUnit.LPAREN, yyline, yycolumn, yytext());}
21     ")"                {return new Symbol(LexicalUnit.RPAREN, yyline, yycolumn, yytext());}
22 // Brackets
23     "{"                {return new Symbol(LexicalUnit.LBRACK, yyline, yycolumn, yytext());}
24     "}"                {return new Symbol(LexicalUnit.RBRACK, yyline, yycolumn, yytext());}
25     "|"                {return new Symbol(LexicalUnit.PIPE, yyline, yycolumn, yytext());}
26 // Arithmetic signs
27     "+"                {return new Symbol(LexicalUnit.PLUS, yyline, yycolumn, yytext());}
28     "-"                {return new Symbol(LexicalUnit.MINUS, yyline, yycolumn, yytext());}
29     "*"                {return new Symbol(LexicalUnit.TIMES, yyline, yycolumn, yytext());}
30     "/"                {return new Symbol(LexicalUnit.DIVIDE, yyline, yycolumn, yytext());}
31 // Logical operators
32     "->"               {return new Symbol(LexicalUnit.IMPLIES, yyline, yycolumn, yytext());}
33 // Conditional keywords
34     "IF"               {return new Symbol(LexicalUnit.IF, yyline, yycolumn, yytext());}
35     "THEN"             {return new Symbol(LexicalUnit.THEN, yyline, yycolumn, yytext());}
36     "ELSE"             {return new Symbol(LexicalUnit.ELSE, yyline, yycolumn, yytext());}
37 // Loop keywords
38     "WHILE"            {return new Symbol(LexicalUnit.WHILE, yyline, yycolumn, yytext());}
39     "REPEAT"           {return new Symbol(LexicalUnit.REPEAT, yyline, yycolumn, yytext());}
40 // Comparison operators
41     "=="               {return new Symbol(LexicalUnit.EQUAL, yyline, yycolumn, yytext());}
42     "<="               {return new Symbol(LexicalUnit.SMALEQ, yyline, yycolumn, yytext());}
43     "<"               {return new Symbol(LexicalUnit.SMALLER, yyline, yycolumn, yytext());}
44 // IO keywords
45     "OUT"              {return new Symbol(LexicalUnit.OUTPUT, yyline, yycolumn, yytext());}
46     "IN"               {return new Symbol(LexicalUnit.INPUT, yyline, yycolumn, yytext());}
47 // Numbers
48     {BadInteger}       {System.err.println("Warning! Numbers with leading zeros are deprecated: " + yytext()); return new Symbol
49     {Integer}          {return new Symbol(LexicalUnit.NUMBER, yyline, yycolumn, Integer.valueOf(yytext()));}
50     {ProgName}         {return new Symbol(LexicalUnit.PROGNAME,yyline, yycolumn,yytext());}
51     {VarName}          {return new Symbol(LexicalUnit.VARNAME,yyline, yycolumn,yytext());}
52     {Separator}        {}// ignore spaces
53     [^]                {throw new PatternSyntaxException("Unmatched token, out of symbols", yytext(), yyline);} // unmatched tok
54 }

```

---

This simplified implementation allows the lexer to efficiently scan and identify tokens without managing complex state transitions, resulting in a straightforward and streamlined lexical analysis process.

### 3.2.2.3 Nested Multi-Line 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 3.2: 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.

### 3.2.3 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.

### 3.2.4 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  java.util.regex.PatternSyntaxException: Unmatched token, out of symbols near index 5
2  >
3      at LexicalAnalyzer.nextToken(LexicalAnalyzer.java:744)
4      at Main.main(Main.java:13)
```

---

Table 3.3: Exception in case of error in the lexer

## 4.1 Transforming the GILLES grammar

The goal of this transformation was to remove any unproductive rules and unreachable variables and to make the grammar non-ambiguous, taking into account the priority and associativity of the operators. We will also remove the left recursion and apply factorisation.

This is done because this transformation ensures that the grammar is ultimately well-suited for parsing actual GILLES code! Removing unproductive rules and unreachable variables simplifies the grammar by eliminating elements that do not contribute to valid parse trees, making the parsing process more efficient. Factoring and enforcing operator precedence and associativity prevent ambiguity, ensuring that expressions are consistently parsed according to intended evaluation rules. These transformations ultimately make the grammar clearer and more efficient for building the parser, not to mention easier to code!

Operators	Associativity
– (unary)	right
*, /	left
+, – (binary)	left
=, <=, <	left
– >	right

Table 4.1: Priority and associativity of the GILLES operators

### 4.1.0.1 Removing unproductive rules

Upon checking the original grammar, we find that the non-terminal `<Call>` is unproductive because it has no production rules defined. It is also unreachable because it cannot derive any terminal strings. Therefore, we will remove `<Call>` and any references to it.

#### Modified Productions:

Remove production [7] `<Instruction> → <Call>`

### 4.1.0.2 Making the grammar non-ambiguous

We need to adjust the grammar to reflect the operator precedence and associativity from the table earlier. This involves restructuring the arithmetic expressions (`<ExprArith>`) and conditions (`<Cond>`) to enforce the correct parsing order.

#### Modified Grammar for `<ExprArith>`:

Non-Terminal	Production Rule
<ExprArith>	<Expr>
<Expr>	<Term> <Expr'>
<Expr'>	<PlusMinus> <Term> <Expr'> $\epsilon$
<PlusMinus>	+ -
<Term>	<Unit> <Term'>
<Term'>	<MulDiv> <Unit> <Term'> $\epsilon$
<MulDiv>	* /
<Unit>	- <Unit> [VarName] [Number] ( <ExprArith> )

Table 4.2: Modified Grammar for <ExprArith>

**Explanation of the modified grammar for <ExprArith>:**

- <Expr> handles addition and subtraction (left-associative).
- <Term> handles multiplication and division (left-associative).
- <Unit> handles unary minus and parentheses.

#### 4.1.0.3 Removing left-recursion

Now we remove the left recursion and apply factorisation where necessary.

**Eliminated Left Recursion and Applied Left Factoring:**

Non-Terminal	Production Rule
<Cond>	<ExprArith> <Comp> <ExprArith> <Cond'>   <Cond>
<Cond'>	-> <Cond> $\epsilon$

Table 4.3: Non-Left-Recursive Grammar for <Cond>

**Explanation of the modified grammar for <Cond>:**

- <CondImp> handles the implication operator -> (right-associative).
- <CondComp> handles comparisons and parentheses in conditions.



- For  $\langle \text{ExprArith} \rangle$  and  $\langle \text{Cond} \rangle$ , left recursion has been removed as shown above.
- **Left Factoring in  $\langle \text{If} \rangle$  Statements:**

*Original Productions:*

[20]  $\langle \text{If} \rangle \rightarrow \text{IF } \{ \langle \text{Cond} \rangle \} \text{ THEN } \langle \text{Code} \rangle \text{ END}$   
 [21]  $\rightarrow \text{IF } \{ \langle \text{Cond} \rangle \} \text{ THEN } \langle \text{Code} \rangle \text{ ELSE } \langle \text{Code} \rangle \text{ END}$

*Modified Productions:*

$\langle \text{If} \rangle \rightarrow \text{IF } \{ \langle \text{Cond} \rangle \} \text{ THEN } \langle \text{Code} \rangle \langle \text{IfTail} \rangle$   
 $\langle \text{IfTail} \rangle \rightarrow \text{END} \mid \text{ELSE } \langle \text{Code} \rangle \text{ END}$

As shown in Table 4.4, the grammar has been transformed to be suitable for top-down parsing.

## 4.2 First and Follow Sets

### 4.2.1 First<sup>1</sup> sets

Below are the First<sup>1</sup> sets of each of the non-terminals:

- $\text{First}^1(\langle \text{Program} \rangle) = \{\text{LET}\}$
- $\text{First}^1(\langle \text{Code} \rangle) = \text{First}^1(\langle \text{Instruction} \rangle) \cup \{\epsilon\}$
- $\text{First}^1(\langle \text{Instruction} \rangle) = \{[\text{VarName}], \text{IF}, \text{WHILE}, \text{OUT}, \text{IN}\}$
- $\text{First}^1(\langle \text{Assign} \rangle) = \{[\text{VarName}]\}$
- $\text{First}^1(\langle \text{If} \rangle) = \{\text{IF}\}$
- $\text{First}^1(\langle \text{IfTail} \rangle) = \{\text{END}, \text{ELSE}\}$
- $\text{First}^1(\langle \text{While} \rangle) = \{\text{WHILE}\}$
- $\text{First}^1(\langle \text{Output} \rangle) = \{\text{OUT}\}$
- $\text{First}^1(\langle \text{Input} \rangle) = \{\text{IN}\}$
- $\text{First}^1(\langle \text{ExprArith} \rangle) = \text{First}^1(\langle \text{Expr} \rangle)$
- $\text{First}^1(\langle \text{Expr} \rangle) = \text{First}^1(\langle \text{Unit} \rangle)$
- $\text{First}^1(\langle \text{Expr}' \rangle) = \{+, -, \epsilon\}$
- $\text{First}^1(\langle \text{PlusMinus} \rangle) = \{+, -\}$
- $\text{First}^1(\langle \text{Term} \rangle) = \text{First}^1(\langle \text{Unit} \rangle)$

Rule	Production
[1]	$\langle \text{Program} \rangle \rightarrow \text{LET } [\text{ProgName}] \text{ BE } \langle \text{Code} \rangle \text{ END}$
[2]	$\langle \text{Code} \rangle \rightarrow \langle \text{Instruction} \rangle : \langle \text{Code} \rangle$
[3]	$\rightarrow \epsilon$
[4]	$\langle \text{Instruction} \rangle \rightarrow \langle \text{Assign} \rangle$
[5]	$\rightarrow \langle \text{If} \rangle$
[6]	$\rightarrow \langle \text{While} \rangle$
[7]	$\rightarrow \langle \text{Output} \rangle$
[8]	$\rightarrow \langle \text{Input} \rangle$
[9]	$\langle \text{Assign} \rangle \rightarrow [\text{VarName}] = \langle \text{ExprArith} \rangle$
[10]	$\langle \text{If} \rangle \rightarrow \text{IF } \{ \langle \text{Cond} \rangle \} \text{ THEN } \langle \text{Code} \rangle \langle \text{IfTail} \rangle$
[11]	$\langle \text{IfTail} \rangle \rightarrow \text{END}$
[12]	$\rightarrow \text{ELSE } \langle \text{Code} \rangle \text{ END}$
[13]	$\langle \text{While} \rangle \rightarrow \text{WHILE } \{ \langle \text{Cond} \rangle \} \text{ REPEAT } \langle \text{Code} \rangle \text{ END}$
[14]	$\langle \text{Output} \rangle \rightarrow \text{OUT } ([\text{VarName}])$
[15]	$\langle \text{Input} \rangle \rightarrow \text{In } ([\text{VarName}])$
[16]	$\langle \text{ExprArith} \rangle \rightarrow \langle \text{Expr} \rangle$
[17]	$\langle \text{Expr} \rangle \rightarrow \langle \text{Term} \rangle \langle \text{Expr}' \rangle$
[18]	$\langle \text{Expr}' \rangle \rightarrow \langle \text{PlusMinus} \rangle \langle \text{Term} \rangle \langle \text{Expr}' \rangle$
[19]	$\rightarrow \epsilon$
[20]	$\langle \text{PlusMinus} \rangle \rightarrow +$
[21]	$\rightarrow -$
[22]	$\langle \text{Term} \rangle \rightarrow \langle \text{Unit} \rangle \langle \text{Term}' \rangle$
[23]	$\langle \text{Term}' \rangle \rightarrow \langle \text{MulDiv} \rangle \langle \text{Unit} \rangle \langle \text{Term}' \rangle$
[24]	$\rightarrow \epsilon$
[25]	$\langle \text{MulDiv} \rangle \rightarrow *$
[26]	$\rightarrow /$
[27]	$\langle \text{Unit} \rangle \rightarrow - \langle \text{Unit} \rangle$
[28]	$\rightarrow [\text{VarName}]$
[29]	$\rightarrow [\text{Number}]$
[30]	$\rightarrow ( \langle \text{ExprArith} \rangle )$
[31]	$\langle \text{Cond} \rangle \rightarrow \langle \text{ExprArith} \rangle \langle \text{Comp} \rangle \langle \text{ExprArith} \rangle \langle \text{Cond}' \rangle$
[32]	$\rightarrow   \langle \text{Cond} \rangle  $
[33]	$\langle \text{Cond}' \rangle \rightarrow -> \langle \text{Cond} \rangle$
[34]	$\rightarrow \epsilon$
[35]	$\langle \text{Comp} \rangle \rightarrow ==$
[36]	$\rightarrow <=$
[37]	$\rightarrow <$

Table 4.4: Modified grammar rules for GILLES

- $\text{First}^1(\langle \text{Term}' \rangle) = \text{First}^1(\langle \text{MulDiv} \rangle)$
- $\text{First}^1(\langle \text{MulDiv} \rangle) = \{*, /\}$
- $\text{First}^1(\langle \text{Unit} \rangle) = \{-, [\text{VarName}], [\text{Number}], ()\}$
- $\text{First}^1(\langle \text{Cond} \rangle) = \{|, -, [\text{VarName}], [\text{Number}], ()\}$
- $\text{First}^1(\langle \text{Cond}' \rangle) = \{->, \epsilon\}$
- $\text{First}^1(\langle \text{Comp} \rangle) = \{==, <=, <\}$

#### 4.2.2 Follow<sup>1</sup> sets

Below are the Follow<sup>1</sup> sets of each of the non-terminals:

- $\text{Follow}^1(\langle \text{Program} \rangle) = \{\$ \}$

- $\text{Follow}^1(\langle \text{Code} \rangle) = \{\text{END}\}$
- $\text{Follow}^1(\langle \text{Instruction} \rangle) = \{:, \text{END}\}$
- $\text{Follow}^1(\langle \text{Assign} \rangle) = \{:, \text{END}\}$
- $\text{Follow}^1(\langle \text{If} \rangle) = \{:, \text{END}\}$
- $\text{Follow}^1(\langle \text{IfTail} \rangle) = \{:, \text{END}\}$
- $\text{Follow}^1(\langle \text{While} \rangle) = \{:, \text{END}\}$
- $\text{Follow}^1(\langle \text{Output} \rangle) = \{:, \text{END}\}$
- $\text{Follow}^1(\langle \text{Input} \rangle) = \{:, \text{END}\}$
- $\text{Follow}^1(\langle \text{ExprArith} \rangle) = \{==, <=, <, ), ->\}$
- $\text{Follow}^1(\langle \text{Expr} \rangle) = \text{Follow}^1(\langle \text{ExprArith} \rangle)$
- $\text{Follow}^1(\langle \text{Expr}' \rangle) = \text{Follow}^1(\langle \text{Expr} \rangle)$
- $\text{Follow}^1(\langle \text{PlusMinus} \rangle) = \{-, [\text{VarName}], [\text{Number}], \{\}$
- $\text{Follow}^1(\langle \text{Term} \rangle) = \{+, -\}$
- $\text{Follow}^1(\langle \text{Term}' \rangle) = \text{Follow}^1(\langle \text{Term} \rangle)$
- $\text{Follow}^1(\langle \text{MulDiv} \rangle) = \{[\text{VarName}], [\text{Number}], (, -\}$
- $\text{Follow}^1(\langle \text{Unit} \rangle) = \{*, /\}$
- $\text{Follow}^1(\langle \text{Cond} \rangle) = \{ \}, \{ \}$
- $\text{Follow}^1(\langle \text{Cond}' \rangle) = \text{Follow}^1(\langle \text{Cond} \rangle)$
- $\text{Follow}^1(\langle \text{Comp} \rangle) = \{-, [\text{VarName}], [\text{Number}], \{\}$

## 4.3 LL(1) Action Table

The corresponding LL(1) parser table is presented on Table 4.5 and Table 4.6. The numbers in the cells correspond to the rule number of the grammar rules in Table 4.4.

*It is divided into two parts so it could fit on the report. The '\$' symbol is replaced by the word "dollar" and the '/' symbol is replaced by the word "div" as well due to render problems in the .csv table.*

## 4.4 Parser Implementation

### 4.4.1 Overview of the Parser

Our `Parser.java` implements a recursive descent parser that processes tokens generated by a lexical analyzer. The main goals are to analyze the syntactic structure of a program, construct a parse tree, and validate input based on specific grammar rules. The key components include initialization, error handling, recursive parsing methods, and building the parse tree.

0	LET	[ProgName]	BE	END	:	[VarName]	=	IF	openKey	closedkey	THEN	ELSE
<Program>	1											
<Code>					2							
<Instruction>						4		5				
<Assign>						9						
<If>								10				
<IfTail>				11								12
<While>												
<Output>												
<Input>												
<ExprArith>						16						
<Expr>						17						
<Expr'>												
<PlusMinus>												
<Term>						22						
<Term'>												
<MulDiv>												
<Unit>						27						
<Cond>						30						
<Cond'>												
<Comp>												

Table 4.5: LL(1) Parsing table (first part)

0	WHILE	REPEAT	OUT	IN	+	-	*	div	==	<=	<	[Number]	(	)	->	dollar
<Program>																
<Code>																
<Instruction>	6		7	8												
<Assign>																
<If>																
<IfTail>																
<While>	13															
<Output>			14													
<Input>				15												
<ExprArith>							16					16		16		
<Expr>							17					17		17		
<Expr'>						18	18									
<PlusMinus>					20	21										
<Term>							22					22		22		
<Term'>								23	23							
<MulDiv>								25	26							
<Unit>							27					28		29		
<Cond>							30					30		30		
<Cond'>															31	
<Comp>									33	34	35					

Table 4.6: LL(1) Parsing table (second part)

#### 4.4.2 Initialization and Setup

The parser initializes with a `LexicalAnalyzer` instance, which tokenizes the input. The following fields are critical:

- `lexicalAnalyzer`: an instance of `LexicalAnalyzer`, responsible for tokenizing the input. This is essentially the result of Part 1.
- `derivation`: a list storing rule numbers used during parsing, tracking the applied grammar rules for printing to `stdout` later.
- `currentToken`: the current token from the lexer, which the parser processes to construct the parse tree. It's

updated every time we read a new token.

### 4.4.3 Parsing Entry Point

The main entry point is the `parse` method, which initiates parsing by calling the `program()` method, the starting rule of our grammar. If parsing completes without errors and reaches the end of input (denoted by EOS), it returns the parse tree; otherwise, it raises an error since it implies there was an error somewhere or we're reading an invalid program!

### 4.4.4 Some Parsing Methods

Our parser implements the grammar rules as separate recursive methods. Each function corresponds to a specific rule in the grammar and constructs a `ParseTree` node. This is how we've implemented the parser "by hand"! Below are explanations of some of the main methods in our parser, the rest follow a similar pattern going by the grammar.

#### 4.4.4.1 `program()`

The `program()` method represents the top-level rule for a program. Its role is to match specific tokens that define the start and end of the program and to recursively call other methods that build the code structure.

---

```
1 private ParseTree program() throws IOException {
2     if (currentToken.getType() == LexicalUnit.LET) {
3         derivation.add(1); // Rule number [1]
4         match(LexicalUnit.LET); // Match LET
5         match(LexicalUnit.PROGNAME); // Match ProgName
6         match(LexicalUnit.BE); // Match BE
7         ParseTree codeTree = code(); // Parse <Code>
8         match(LexicalUnit.END); // Match END
9         List<ParseTree> children = Arrays.asList(
10             new ParseTree(new Symbol(LexicalUnit.LET)),
11             new ParseTree(new Symbol(LexicalUnit.PROGNAME)),
12             new ParseTree(new Symbol(LexicalUnit.BE)),
13             codeTree,
14             new ParseTree(new Symbol(LexicalUnit.END))
15         );
16         return new ParseTree(new Symbol(null, NonTerminal.PROGRAM), children);
17     } else {
18         unexpectedToken();
19         return null;
20     }
21 }
```

---

**Explanation:** The `program()` method begins by checking if the `currentToken` is a `LET` token. It then matches tokens in sequence: `LET`, `PROGNAME`, and `BE`, before parsing a sequence of instructions with `code()`. Finally, it matches the `END` token. This method constructs a parse tree node for the program, combining child nodes for each component.

#### 4.4.4.2 `code()`

The `code()` method represents a block of instructions and manages the chaining of multiple instructions.

---

```

1 private ParseTree code() throws IOException {
2     if (currentToken.getType() == LexicalUnit.VARNAME || currentToken.getType() == LexicalUnit.IF || currentToken.getType() == LexicalUnit.WHILE) {
3         derivation.add(2); // Rule number [2]
4         ParseTree instructionTree = instruction(); // Parse <Instruction>
5         match(LexicalUnit.COLON); // Match ;
6         ParseTree codeTree = code(); // Parse <Code>
7         List<ParseTree> children = Arrays.asList(
8             instructionTree,
9             new ParseTree(new Symbol(LexicalUnit.COLON)),
10            codeTree
11        );
12        return new ParseTree(new Symbol(null, NonTerminal.CODE), children);
13    } else {
14        derivation.add(3); // Rule number [3]
15        return new ParseTree(new Symbol(null, NonTerminal.EPSILON));
16    }
17 }

```

---

**Explanation:** The `code()` method handles sequences of instructions, enabling recursive chaining of instructions until it reaches a terminal point. If the current token represents an instruction (e.g., `VARNAME`, `IF`, `WHILE`, etc.), it parses an instruction using `instruction()` and then expects a semicolon (`COLON`) to separate instructions. Finally, it calls `code()` recursively to parse any additional instructions. If no valid instruction token is found, it returns an epsilon node to mark the end of the instruction sequence.

#### 4.4.4.3 instruction()

The `instruction()` method determines the type of instruction (assignment, conditional, loop, output, or input) based on the `currentToken` and calls the corresponding method.

---

```

1 private ParseTree instruction() throws IOException {
2     if (currentToken.getType() == LexicalUnit.VARNAME) {
3         derivation.add(4); // Rule number [4]
4         return assign();
5     } else if (currentToken.getType() == LexicalUnit.IF) {
6         derivation.add(5); // Rule number [5]
7         return ifStatement();
8     } else if (currentToken.getType() == LexicalUnit.WHILE) {
9         derivation.add(6); // Rule number [6]
10        return whileStatement();
11    } else if (currentToken.getType() == LexicalUnit.OUTPUT) {
12        derivation.add(7); // Rule number [7]
13        return output();
14    } else if (currentToken.getType() == LexicalUnit.INPUT) {
15        derivation.add(8); // Rule number [8]
16        return input();
17    } else {
18        unexpectedToken();
19        return null;
20    }
21 }

```

---

```
20     }
21 }
```

---

**Explanation:** The `instruction()` method differentiates between possible instructions by checking the type of the `currentToken`. Based on this, it calls the appropriate parsing method, such as `assign()` for assignments or `ifStatement()` for conditional statements. Each type of instruction generates a corresponding parse tree node, which is then integrated into the overall syntax tree.

#### 4.4.5 `ifStatement()`

The `ifStatement()` method handles the parsing of conditional IF statements. This method verifies that the `currentToken` matches the expected IF token, then proceeds through the components of an IF statement, including a conditional expression, a THEN clause, and optionally an ELSE clause. This method returns a parse tree node representing the complete IF structure.

---

```
1 private ParseTree ifStatement() throws IOException {
2     if (currentToken.getType() == LexicalUnit.IF) {
3         derivation.add(10); // Rule number [10]
4         match(LexicalUnit.IF); // Match IF
5         match(LexicalUnit.LBRACK); // Match {
6         ParseTree condTree = cond(); // Parse <Cond>
7         match(LexicalUnit.RBRACK); // Match }
8         match(LexicalUnit.THEN); // Match THEN
9         ParseTree codeTree = code(); // Parse <Code>
10        ParseTree ifTailTree = ifTail(); // Parse <IfTail>
11        // Build parse tree node
12        List<ParseTree> children = Arrays.asList(
13            new ParseTree(new Symbol(LexicalUnit.IF)),
14            new ParseTree(new Symbol(LexicalUnit.LBRACK)),
15            condTree,
16            new ParseTree(new Symbol(LexicalUnit.RBRACK)),
17            new ParseTree(new Symbol(LexicalUnit.THEN)),
18            codeTree,
19            ifTailTree
20        );
21        return new ParseTree(new Symbol(null, NonTerminal.IFSTATEMENT), children);
22    } else {
23        unexpectedToken();
24        return null;
25    }
26 }
```

---

**Explanation:** The `ifStatement()` method begins by verifying that the `currentToken` is an IF token and then proceeds to match the IF keyword and opening bracket. It parses the conditional expression using `cond()`, matches the closing bracket and THEN keyword, and parses the statement body with `code()`. Finally, it handles any optional ELSE clause by invoking `ifTail()`, constructing a complete parse tree node for the IF structure.

The `ifStatement()` method constructs a `ParseTree` node for the IF statement, combining child nodes for each

part of the conditional structure. This modular approach supports nested conditionals, ensuring that complex conditional structures can be represented in the parse tree.

The rest of the functions follow a similar pattern, encoding the rules we finalised in Table 4.4.



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.

## 5.1 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.
- `UnclosedComment.gls` - Unclosed multi-line comment, which should throw an error.

## 5.2 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 5.1: Lexer output for Part 1

## 5.3 Output of the Parser

For Part 2, the parser outputs the following for the same input file, `Sum.gls`.

---

1	1	2	8	15	2	8	15	2	4	9	16	17	22	28	24	18	20	22	28	24	19	2	7	14	3
---	---	---	---	----	---	---	----	---	---	---	----	----	----	----	----	----	----	----	----	----	----	---	---	----	---

---

Table 5.2: Parser output for Part 2

We have also enabled the creation of the parse tree in LaTeX when the variable `OUTPUT_TEX_FILE` is set while running *make test*. For example, if the command:

```
make test TEST_FILE=test/Sum.gls OUTPUT_TEX_FILE=sum_output.tex
```

is run, the Makefile internally runs the command:

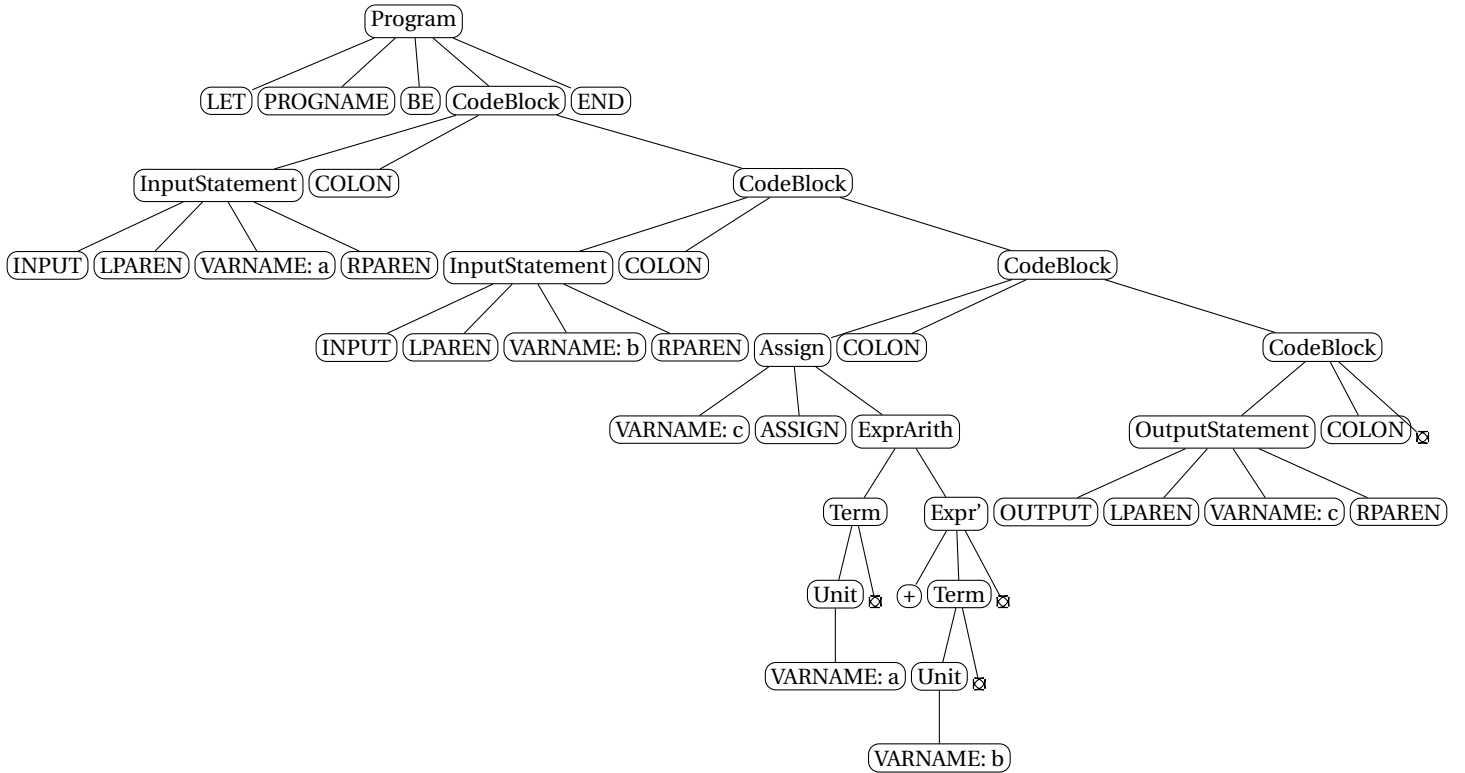
```
java -jar part2.jar -wt ${OUTPUT_TEX_FILE} ${TEST_FILE}
```

This additionally also creates a PDF from the LaTeX that is generated inside the `doc` directory.

If the option `-wt` is not passed, we just print the left-most derivation on *stdout* without creating the `.tex` file or the `.pdf`.

## 5.4 Generated ParseTree

For the file we presented the output for Part 1, `Sum.gls`, we also have the below parse tree produced in LaTeX:



The parse tree starts with the root node `Program`, with children in the structure represented through the `LET`, `PROGNAME`, `BE`, and `END` nodes. Inside, the `CodeBlock` node contain statements and expressions.

The first `CodeBlock` begins by handling two `InputStatement` nodes, where the variables `a` and `b` are read from input. Following these inputs, an `Assign` node assigns an arithmetic expression to the variable `c`. This arithmetic expression is represented by the `ExprArith` node, which further expands into `Term` and `Expr'` components to illustrate the addition of `a` and `b`. Here, `Term` captures the value of `a`, followed by `Expr'` with the `+` operator and `Term` for `b`, completing the summation expression. This structure mirrors the leftmost derivation of the expression, systematically breaking down the addition operation as the parser processes each part in sequence.

The summation expression demonstrates a leftmost derivation, where `ExprArith` is expanded to the leftmost components first, starting with `Term` for `a`, followed by the `+` operator and then `Term` for `b`. The entire tree was generated through recursive descent parsing, with each node corresponding to a leftmost expansion. Producing this parse tree in LaTeX allows us to visualize the program's syntactic structure quite well!