ULB UNIVERSITÉ LIBRE DE BRUXELLES

Introduction to Language Theory and Compiling

Gilles Compiler

Siddharth SAHAY
Jordi UGARTE

Prof. GERAERTS Gilles

BALACHANDER Mrudula

SASSOLAS Mathieu

2024

Contents

List of Tables

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 creation of a compiler for a new Programming Language using Jflex and other tools. This execution is divided into three parts: The creation of a lexical analyzer which is responsible for scanning GILLES source code, the recursive-descent LL(1) parser development for it based on its grammar rules and the compiling process into an assembly language. In this project, the assembly code that is going to be used is LLVM.

1 Introduction

1.1 Background

The primary objective of this project is to design and implement a compiler for GILLES. This report explains the three parts for the compiler to be developed, specifically, the creation of a lexical analyzer using the JFlex tool, the generation of a corresponding Parse Tree of any input of Gilles source code, and the parsing into an assembly language like LLVM.

GILLES Grammar Rules

In Table 1.1, the Gilles programming language grammar rules can be appreciated. It is important to know that the grammar rules where changed during the execution of the parts of the project. These grammar rules were provided by the last correction of the second project and are implemented along the execution of the LLVM parser. There is another table of grammar rules prior to the development of the first part of the project that can be shown on Table 1.2. There is also another Gilles Grammar table for part 2 shown in that corresponding section.

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), output (Output), or input (Input).
- Arithmetic Expressions ([10]-[21]): These rules define how arithmetic expressions are constructed using variables, numbers, parentheses, negation, and binary operators (+, -, *, /).
- Conditionals ([22]-[33]): The <If> and <While> constructs enable conditional execution and looping based on specified conditions.
- Input/Output ([34]-[35]): 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

[1]	$\langle Program \rangle \rightarrow$ **LET** [**ProgName**] **BE** $\langle Code \rangle$ **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]	$\rightarrow \langle Output \rangle$
[8]	$\rightarrow \langle Input \rangle$
[9]	$\langle Assign \rangle \rightarrow$ [**VarName**] $= \langle ExprArith \rangle$
[10]	$\langle ExprArith \rangle \rightarrow \langle Prod \rangle \langle ExprArith' \rangle$
[11]	$\langle ExprArith' \rangle \rightarrow + \langle Prod \rangle \langle ExprArith' \rangle$
[12]	$\rightarrow - \langle Prod \rangle \langle ExprArith' \rangle$
[13]	$\rightarrow \varepsilon$
[14]	$\langle Prod \rangle \rightarrow \langle Atom \rangle \langle Prod' \rangle$
[15]	$\langle Prod' \rangle \rightarrow * \langle Atom \rangle \langle Prod' \rangle$
[16]	$\rightarrow / \langle Atom \rangle \langle Prod' \rangle$
[17]	$\rightarrow \varepsilon$
[18]	$\langle Atom \rangle \rightarrow$ [**VarName**]
[19]	$\rightarrow$ [**Number**]
[20]	$\rightarrow (\langle ExprArith \rangle)$
[21]	$\rightarrow - \langle Atom \rangle$
[22]	$\langle If \rangle \rightarrow$ **IF** $\{\langle Cond \rangle\}$ **THEN** $\langle Code \rangle \langle IfTail \rangle$
[23]	$\langle IfTail \rangle \rightarrow$ **ELSE** $\langle Code \rangle$ **END**
[24]	$\rightarrow$ **END**
[25]	$\langle Cond \rangle \rightarrow \langle SimpleCond \rangle \langle Cond' \rangle$
[26]	$\langle Cond' \rangle \rightarrow -> \langle Cond \rangle$
[27]	$\rightarrow \varepsilon$
[28]	$\langle SimpleCond \rangle \rightarrow \lvert \langle Cond \rangle \rvert$
[29]	$\rightarrow \langle ExprArith \rangle \langle Comp \rangle \langle ExprArith \rangle$
[30]	$\langle Comp \rangle \rightarrow ==$
[31]	$\rightarrow <=$
[32]	$\rightarrow <$
[33]	$\langle While \rangle \rightarrow$ **WHILE** $\{\langle Cond \rangle\}$ **REPEAT** $\langle Code \rangle$ **END**
[34]	$\langle Output \rangle \rightarrow$ **OUT**([**VarName**])
[35]	$\langle Input \rangle \rightarrow$ **IN**([**VarName**])

Table 1.1: The GILLES updated grammar.

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 *part3.jar* to be runnable inside the folder. This *.jar* file will run any input *.gls* file from the test folder and return an output parsed into LLVM code. This can be run with the command:

```
$ make test
```

Rule No.	Production Rule
1	$\langle Program \rangle \rightarrow$ LET [ProgName] BE $\langle Code \rangle$ 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	$\langle Assign \rangle \rightarrow$ [VarName] = $\langle ExprArith \rangle$
11	$\langle ExprArith \rangle \rightarrow$ [VarName]
12	$\rightarrow$ [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	$\langle If \rangle \rightarrow$ IF $\{\langle Cond \rangle\}$ THEN $\langle Code \rangle$ END
21	$\rightarrow$ IF $\{\langle Cond \rangle\}$ THEN $\langle Code \rangle$ ELSE $\langle Code \rangle$ END
22	$\langle Cond \rangle \rightarrow \langle Cond \rangle$
23	$\rightarrow \lvert \langle Cond \rangle \rvert$
24	$\rightarrow \langle ExprArith \rangle \langle Comp \rangle \langle ExprArith \rangle$
25	$\langle Comp \rangle \rightarrow ==$
26	$\rightarrow <=$
27	$\rightarrow <$
28	$\langle While \rangle \rightarrow$ WHILE $\{\langle Cond \rangle\}$ REPEAT $\langle Code \rangle$ END
29	$\langle Output \rangle \rightarrow$ OUT([VarName])
30	$\langle Input \rangle \rightarrow$ IN([VarName])

Table 1.2: Previous GILLES grammar execution for Part 2.

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

```
$ make run-llvm TEST_FILE=test/FILENAME.gls
```

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

Project Structure

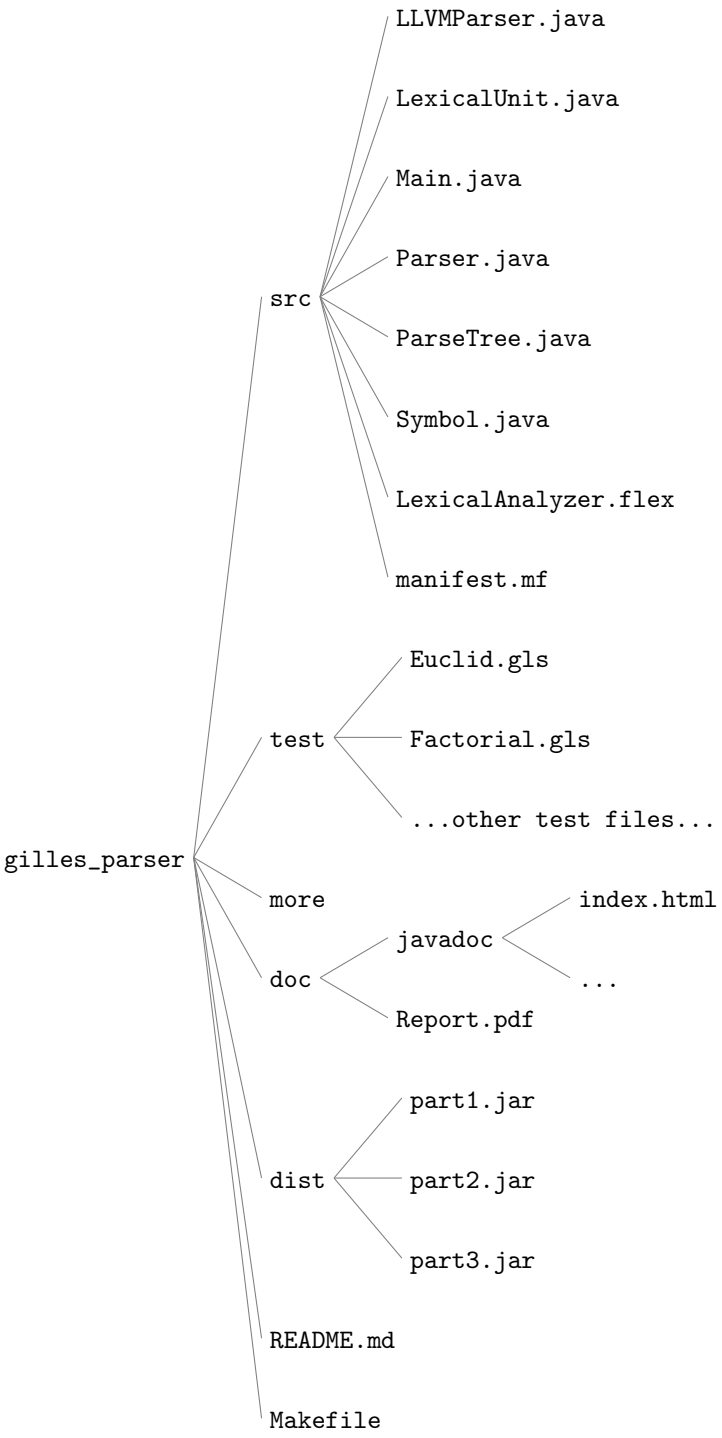


Table 1.3: GILLES Parser Project Structure

2 Objective

The objective of the project is to develop a compiler for the Genial Imperative Language for Learning and the Enlightenment of Students (GILLES) 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.
3. Augment the recursive-descent LL(1) parser to let it generate code that corresponds to the semantics of the gilles program that is being compiled into LLVM IR code.

3 Part 1 - The Lexer

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 = $[A-Za-z]$

Numeric = $[0-9]$

AlphaNumeric = $[A-Za-z0-9]$

BadInteger = $(0[0-9]+)$

Integer = $([1-9][0-9]*)|0$

ProgName = $[A-Z]([A-Za-z]|"\_")*$

VarName = $[a-z]([A-Za-z0-9])*$

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.

```
<LONGCOMMENTS> {
// End of comment
  "!!"                          {yybegin(YYINITIAL);} // go back to analysis
  <<EOF>>          {throw new PatternSyntaxException("A comment is never closed.", yytext(), yyline);}
  [^]                                              {} //ignore any character
}

<YYINITIAL> {
// Comments
  "!!"               {yybegin(LONGCOMMENTS);} // go to ignore mode
  "$"{UpToEnd}      {} // go to ignore mode
// Code delimiters
  "LET"              {return new Symbol(LexicalUnit.LET, yyline, yycolumn, yytext());}
  "BE"               {return new Symbol(LexicalUnit.BE, yyline, yycolumn, yytext());}
  "END"              {return new Symbol(LexicalUnit.END, yyline, yycolumn, yytext());}
  ":"                {return new Symbol(LexicalUnit.COLUMN, yyline, yycolumn, yytext());}
// Assignation
```

```
  "="                {return new Symbol(LexicalUnit.ASSIGN, yyline, yycolumn, yytext());}
// Parenthesis
  "("                 {return new Symbol(LexicalUnit.LPAREN, yyline, yycolumn, yytext());}
  ")"                 {return new Symbol(LexicalUnit.RPAREN, yyline, yycolumn, yytext());}
// Brackets
  "{"                 {return new Symbol(LexicalUnit.LBRACK, yyline, yycolumn, yytext());}
  "}"                 {return new Symbol(LexicalUnit.RBRACK, yyline, yycolumn, yytext());}
  "|"                 {return new Symbol(LexicalUnit.PIPE, yyline, yycolumn, yytext());}
// Arithmetic signs
  "+"                 {return new Symbol(LexicalUnit.PLUS, yyline, yycolumn, yytext());}
  "-"                 {return new Symbol(LexicalUnit.MINUS, yyline, yycolumn, yytext());}
  "*"                 {return new Symbol(LexicalUnit.TIMES, yyline, yycolumn, yytext());}
  "/"                 {return new Symbol(LexicalUnit.DIVIDE, yyline, yycolumn, yytext());}
// Logical operators
  "->"               {return new Symbol(LexicalUnit.IMPLIES, yyline, yycolumn, yytext());}
// Conditional keywords
  "IF"                {return new Symbol(LexicalUnit.IF, yyline, yycolumn, yytext());}
  "THEN"              {return new Symbol(LexicalUnit.THEN, yyline, yycolumn, yytext());}
  "ELSE"              {return new Symbol(LexicalUnit.ELSE, yyline, yycolumn, yytext());}
// Loop keywords
  "WHILE"             {return new Symbol(LexicalUnit.WHILE, yyline, yycolumn, yytext());}
  "REPEAT"            {return new Symbol(LexicalUnit.REPEAT, yyline, yycolumn, yytext());}
// Comparison operators
  "=="                {return new Symbol(LexicalUnit.EQUAL, yyline, yycolumn, yytext());}
  "<="                {return new Symbol(LexicalUnit.SMALEQ, yyline, yycolumn, yytext());}
  "<"                 {return new Symbol(LexicalUnit.SMALLER, yyline, yycolumn, yytext());}
// IO keywords
  "OUT"              {return new Symbol(LexicalUnit.OUTPUT, yyline, yycolumn, yytext());}
  "IN"               {return new Symbol(LexicalUnit.INPUT, yyline, yycolumn, yytext());}
// Numbers
  {BadInteger}        {System.err.println("Warning! Numbers with leading zeros are deprecated: " + yytext()); return new Symbol
  {Integer}           {return new Symbol(LexicalUnit.NUMBER, yyline, yycolumn, Integer.valueOf(yytext()));}
  {ProgName}           {return new Symbol(LexicalUnit.PROGNAME,yyline, yycolumn,yytext());}
  {VarName}           {return new Symbol(LexicalUnit.VARNAME,yyline, yycolumn,yytext());}
  {Separator}         {}// ignore spaces
  [^]                 {throw new PatternSyntaxException("Unmatched token, out of symbols", yytext(), yyline);} // unmatched tok
}
```

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 becasue 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:

```
!! This is a 1st level comment
```

```
!! Nested 2nd level comment !!
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:

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

Table 3.3: Exception in case of error in the lexer

4

Part 2 - The Parser

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'> ϵ
<PlusMinus>	+ -
<Term>	<Unit> <Term'>
<Term'>	<MulDiv> <Unit> <Term'> ϵ
<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> ϵ

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 <ExprArith> and <Cond>, left recursion has been removed as shown above.**
- **Left Factoring in <If> Statements:**

  Original Productions:

  ```
  [20] <If> → IF { <Cond> } THEN <Code> END
  [21] → IF { <Cond> } THEN <Code> ELSE <Code> END
  ```

  Modified Productions:

  ```
  <If> → IF { <Cond> } THEN <Code> <IfTail>

  <IfTail> → END | ELSE <Code> 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[1] sets

Below are the First[1] sets of each of the non-terminals:

- First[1](<Program>) = {LET}
- First[1](<Code>) = First[1](<Instruction>) ∪ {ϵ}
- First[1](<Instruction>) = {[VarName], IF, WHILE, OUT, IN}
- First[1](<Assign>) = {[VarName]}
- First[1](<If>) = {IF}
- First[1](<IfTail>) = {END, ELSE}
- First[1](<While>) = {WHILE}
- First[1](<Output>) = {OUT}
- First[1](<Input>) = {IN}
- First[1](<ExprArith>) = First[1](<Expr>)
- First[1](<Expr>) = First[1](<Unit>)
- First[1](<Expr'>) = {+, -, ϵ}
- First[1](<PlusMinus>) = {+, -}
- First[1](<Term>) = First[1](<Unit>)

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

Table 4.4: Modified grammar rules for GILLES

- First^1(<Term'>) = First^1(<MulDiv>)
- First^1(<MulDiv>) = {*, /}
- First^1(<Unit>) = {-, [VarName], [Number], (}
- First^1(<Cond>) = {|, -, [VarName], [Number], (}
- First^1(<Cond'>) = {->, ϵ}
- First^1(<Comp>) = {==, <=, <}

4.2.2 Follow^1 sets

Below are the Follow^1 sets of each of the non-terminals:

- Follow^1(<Program>) = {$}

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

```
private ParseTree code() throws IOException {
    if (currentToken.getType() == LexicalUnit.VARNAME || currentToken.getType() == LexicalUnit.IF || currentToken.getType() == L
        derivation.add(2); // Rule number [2]
        ParseTree instructionTree = instruction(); // Parse <Instruction>
        match(LexicalUnit.COLON); // Match ;
        ParseTree codeTree = code(); // Parse <Code>
        List<ParseTree> children = Arrays.asList(
            instructionTree,
            new ParseTree(new Symbol(LexicalUnit.COLON)),
            codeTree
        );
        return new ParseTree(new Symbol(null, NonTerminal.CODE), children);
    } else {
        derivation.add(3); // Rule number [3]
        return new ParseTree(new Symbol(null, NonTerminal.EPSILON));
    }
}
```

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.

```
private ParseTree instruction() throws IOException {
    if (currentToken.getType() == LexicalUnit.VARNAME) {
        derivation.add(4); // Rule number [4]
        return assign();
    } else if (currentToken.getType() == LexicalUnit.IF) {
        derivation.add(5); // Rule number [5]
        return ifStatement();
    } else if (currentToken.getType() == LexicalUnit.WHILE) {
        derivation.add(6); // Rule number [6]
        return whileStatement();
    } else if (currentToken.getType() == LexicalUnit.OUTPUT) {
        derivation.add(7); // Rule number [7]
        return output();
    } else if (currentToken.getType() == LexicalUnit.INPUT) {
        derivation.add(8); // Rule number [8]
        return input();
    } else {
        unexpectedToken();
        return null;
```

```
    }
}
```

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.

```
private ParseTree ifStatement() throws IOException {
    if (currentToken.getType() == LexicalUnit.IF) {
        derivation.add(10); // Rule number [10]
        match(LexicalUnit.IF); // Match IF
        match(LexicalUnit.LBRACK); // Match {
        ParseTree condTree = cond(); // Parse <Cond>
        match(LexicalUnit.RBRACK); // Match }
        match(LexicalUnit.THEN); // Match THEN
        ParseTree codeTree = code(); // Parse <Code>
        ParseTree ifTailTree = ifTail(); // Parse <IfTail>
        // Build parse tree node
        List<ParseTree> children = Arrays.asList(
                new ParseTree(new Symbol(LexicalUnit.IF)),
                new ParseTree(new Symbol(LexicalUnit.LBRACK)),
                condTree,
                new ParseTree(new Symbol(LexicalUnit.RBRACK)),
                new ParseTree(new Symbol(LexicalUnit.THEN)),
                codeTree,
                ifTailTree
        );
        return new ParseTree(new Symbol(null, NonTerminal.IFSTATEMENT), children);
    } else {
        unexpectedToken();
        return null;
    }
}
```

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.

5 Part 3 - Assembly code generation (LLVM IR)

5.1 LLVM Parser

As an augmentation of the LL(1) parser class, another class was developed called *LLVMParser.java* which contains all the necessary methods and procedures to parse GILLES code into assembly code (LLVM IR). The class receives as input a *ParseTree*, which its execution and how it is obtained can be contemplated in Part 2. The LLVMParser class receives the parse tree and recursively analyzes its terminal and non terminal values and generates their corresponding LLVM IR implementations. Several procedures to generate this assembly code were not only based on "pure parsing". It was necessary to handle memory towards variables, labels, atomic values, and counters to make the code compilable in by the LLVM IR tool.

5.1.1 Memory allocations

- `programName`: The program name obtained by the terminal [PROGNAME]. It is used just to generate the name of the file.
- `variables`: A HashMap of type <*String, Integer*> which contains all the variables registered in the code with their corresponding indexes that represent how many times it was called to load their value. This is implemented in order to get updated values of a variable whenever it is necessary to create a pointer that gets their current value in run-time (*load i32, i32 %n, align 4*).
- `variableSet`: A LinkedHashSet that contains all the names of the variables used in the whole program. This is implemented first for getting all the variables at the beggining of the program (*entry:* label) in order to allocate their memories and be accessible throughout the whole code. This method was the best way of dealing with all variables and calling their updated pointers as some variables declarations cannot be always accessed throughout labels when declaring them.
- `Counters`: Integers that handle how many times the following operations are called with their own codes or distinct executions. This is implemented in particular to nest *IF* operations or *WHILE* loops. The counters are mainly used in this case to reference these operations with distinction from one another. In the case of arithemtic, conditions and productions, the use of counters is implemented to call multiple pointers that reference dual operations that is one of the basis of assembly (i.e. *%arith0 add i32 0, 1*). In the case of *ifCounter* and *whileCounter* their counters are used in their corresponding labels (i.e. *while_cond1:*, *while_block1:*, *while_end1:*).

- `ifCounter`: Counter corresponding to the "if" conditions quantity declared in the code.
- `whileCounter`: Counter corresponding to the "while" conditions quantity declared in the code.
- `arithCounter`: Counter corresponding to the arithmetic values declared.
- `condCounter`: Counter corresponding to the conditions values declared.
- `prodCounter`: Counter corresponding to the production values declared.

- *Queues*: When dealing with arithmetic expressions, Atoms are always present in sequences followed by arithmetic operators (+, -, *, /). Contemplating the priority between productions and arithmetic expressions ('*' and '/' have more priority than '+' and '-'), the parser first processes the operations of the productions, and later on, the arithmetic operations.

  The arithmetic and production queues work in order to only contain two values. The parser reads an arithmetic operation and takes pairs of numbers and add them to their corresponding queue (arithmetic or production queues). If a queue is empty, it searches for an atom to add. Therefore, a pointer is created with that number summing it with 0. On the other hand, if it has two values inside the queue, the values are retrieved and popped from the queue leaving it empty to be added to a new operation pair. This is an example of how queues work:

  Take the following GILLES code that prints the result of the arithmetic operation *x* transformed into LLVM IR in Table 5.1:

```
LET Main BE
    x = 5 * 3 + 4 / 2 - 1 + 0 + 7:
    OUT(x):
END
```

```
define i32 @main() {
entry:
    %x = alloca i32, align 4

    %prod1 = mul i32 5, 3
    %prod2 = add i32 0, %prod1
    %prod3 = sdiv i32 4, 2
    %prod4 = add i32 0, %prod3
    %arith1 = add i32 %prod2, %prod4
    %prod5 = add i32 0, 1
    %arith2 = sub i32 %arith1, %prod5
    %prod6 = add i32 0, 0
    %arith3 = add i32 %arith2, %prod6
    %prod7 = add i32 0, 7
    %arith4 = add i32 %arith3, %prod7
    %arith5 = add i32 0, %arith4
    store i32 %arith5, i32* %x, align 4
    %x_val1 = load i32, i32* %x, align 4
    call void @println(i32 %x_val1)

    ret i32 0
}
```

Table 5.1: GILLES code example of arithmetic operations in LLVM IR.

In the case of multiple conditions followed by the implication operator (->), a queue is also used to reference pairs of the condition pointers and to parse them with the implication operator if it is needed. The only queue that is not limited to store pairs is the end queue. The end queue stores references to all the *end labels*. An *end label* is a label in the LLVM IR code that references the code continuation whenever a while loop ends or an if condition ends. This example can be seen in Table 5.2.

```
LET Main BE
  IN(x):
  IF {x <= 10 -> 0 < x} THEN
    OUT(x):
  END:
END
```

```
define i32 @main() {
entry:
    %x = alloca i32, align 4
    %x_input = call i32 @readInt()
    store i32 %x_input, i32* %x, align 4
    %x_val1 = load i32, i32* %x, align 4
    %prod2 = add i32 0, %x_val1
    %arith2 = add i32 0, %prod2
    %prod3 = add i32 0, 10
    %arith3 = add i32 0, %prod3
    %cond1 = icmp sle i32 %arith2, %arith3
    %prod5 = add i32 0, 0
    %arith5 = add i32 0, %prod5
    %prod6 = add i32 0, %x_val6
    %arith6 = add i32 0, %prod6
    %cond2 = icmp slt i32 %arith5, %arith6
    %cond3 = call i1 @logical_implication(i1%cond1, i1%cond2)
    br i1 %cond3, label %if_block1, label %else_block1
if_block1:
    %x_val8 = load i32, i32* %x, align 4
    call void @println(i32 %x_val8)
    br label %end1
else_block1:
    br label %end1
end1:
    ret i32 0
}
```

Table 5.2: GILLES code example of sequences of conditions in LLVM IR.

- `atomicQueue`: Queue corresponding to atomic values contemplated in an arithmetic expression.
- `prodQueue`: Queue corresponding to production values contemplated in an arithmetic expression.
- `conditionQueue`: Queue corresponding to boolean conditions contemplated in an condition with multiple conditions with the operator ->.
- `endQueue`: Queue corresponding to the end labels of the code.

5.1.2 Functions by default

In order to implement some functions from GILLES Code that are repeated multiple times but have complex implementations in LLVM IR, functions by default are implemented on any generated LLVM IR code to be available for any possible or future use to avoid writing them multiple times it is needed, and instead, call its corresponding function. These complex functions are referred to: IN(n), OUT(n), and the operator implication, which receives two conditionals and returns its value. These codes can be shown on Table 5.3.

```
define i1 @logical_implication(i1 %p, i1 %q) {
entry:
    %not_p = xor i1 %p, true
    %result = or i1 %not_p, %q
    ret i1 %result
}

@.strR = private unnamed_addr constant [3 x i8] c"%d\00", align 1
define i32 @readInt() #0 {
  %1 = alloca i32, align 4
  %2 = call i32 (i8*, ...) @scanf(i8* getelementptr inbounds ([3 x i8], [3 x i8]* @.strR, i32 0, i32 0), i32* %1)
  %3 = load i32, i32* %1, align 4
  ret i32 %3
}

declare i32 @scanf(i8*, ...) #1

@.strP = private unnamed_addr constant [4 x i8] c"%d\0A\00", align 1
define void @println(i32 %x) #0 {
  %1 = alloca i32, align 4
  store i32 %x, i32* %1, align 4
  %2 = load i32, i32* %1, align 4
  %3 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x i8], [4 x i8]* @.strP, i32 0, i32 0), i32 %2)
  ret void
}
```

Table 5.3: Functions implemented by default in each code generated.

5.1.3 Parsing functions

The LLVM Parser works in a recursive way, taking all nodes in the Parse Tree and taking different actions and parsings depending on the expression that the node references. A node can be terminal or non-terminal. In order to achieve this, the method *look(ParseTree node)* takes a node and generates LLVM code depending on the terminal or non-terminal symbols.

The whole parser works fundamentally by this method in a recursive way. This method can be seen in 5.4.

```java
public String look(ParseTree node) {
    String expression = node.getLabel().toString();
    if (expression.startsWith("Non-terminal symbol: ")) {
        String nonTerminal = expression.split("Non-terminal symbol: ")[1];
        return switch (nonTerminal) {
            case "Program" -> program(node);
            case "Code" -> code(node);
            case "Instruction" -> instruction(node);
            case "Assign" -> assign(node);
            case "ExprArith" -> exprArith(node);
            case "ExprArith'" -> exprArithPrime(node);
            case "Prod" -> prod(node);
            case "Prod'" -> prodPrime(node);
            case "Atom" -> atom(node);
            case "Input" -> input(node);
            case "Output" -> output(node);
            case "If" -> iF(node);
            case "Cond" -> cond(node);
            case "Cond'" -> condPrime(node);
            case "Comp" -> comp(node);
            case "SimpleCond" -> simpleCond(node);
            case "IfTail" -> ifTail(node);
            case "While" -> whilE(node);
            default -> throw new RuntimeException("Unknown Non-terminal Expression: " + nonTerminal);
        };
    } else {
    // Terminal symbols
        String terminal = expression.split("lexical unit: ")[1];
        String token = expression.split("lexical unit: ")[0].split("token: ")[1].replaceAll("\\s", "");
        return switch (terminal) {
            case "[ProgName]" -> setProgramName(token);
            case "[VarName]" -> "%".concat(token);
            case "[Number]" -> token;
            case "LET" -> "define i32 ";
            case "BE" -> line(" {");
            case "END" -> getProgramEnd();
            case "COLUMN" -> line();
            case "OUT" -> "ret ";
            case "+" -> "add";
            case "-" -> "sub";
            case "*" -> "mul";
            case "/" -> "sdiv";
            case "=" -> " = ";
            case "==" -> "icmp eq";
            case "<=" -> "icmp sle";
            case "<" -> "icmp slt";
            case "->" -> "and";
            case "|" -> "|";
            case "ELSE" -> "ELSE";
            default -> "";
        };
    }
}
```

Table 5.4: *look()* function in Java.

6 Testing

6.1 Testing Lexical Analyzer

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.

6.1.0.1 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 unncessarily 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.

6.1.0.2 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
```

```
token: a                lexical unit: VARNAME
token: )                lexical unit: RPAREN
token: :                lexical unit: COLUMN
token: IN               lexical unit: INPUT
token: (                lexical unit: LPAREN
token: b                lexical unit: VARNAME
token: )                lexical unit: RPAREN
token: :                lexical unit: COLUMN
token: c                lexical unit: VARNAME
token: =                lexical unit: ASSIGN
token: a                lexical unit: VARNAME
token: +                lexical unit: PLUS
token: b                lexical unit: VARNAME
token: :                lexical unit: COLUMN
token: END              lexical unit: END
token: :                lexical unit: COLUMN
token: OUT              lexical unit: OUTPUT
token: (                lexical unit: LPAREN
token: c                lexical unit: VARNAME
token: )                lexical unit: RPAREN
token: :                lexical unit: COLUMN
token: END              lexical unit: END

Variables
a       4
b       5
c       6
```

Table 6.1: Lexer output for Part 1

6.2 Output of the Parser for Part 2

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

```
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 6.2: 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:

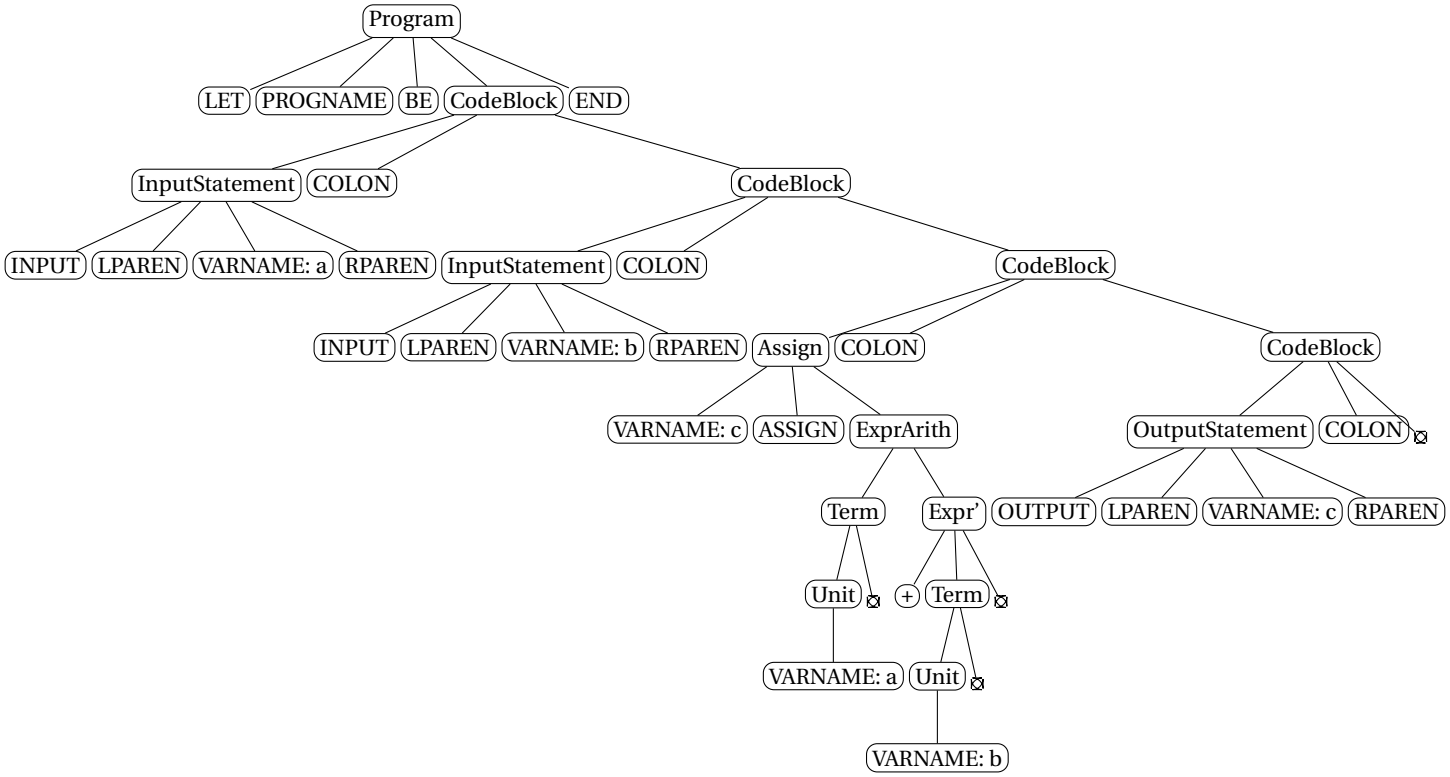
```
java -jar part2.jar -wt ${OUTPUT_TEX_FILE} ${TEST_FILE}
```

Running the `make test` command with the additional parameter `OUTPUT_TEX_FILE` additionally also creates a PDF from the LaTeX that is generated in the same location as the output `.tex` file. This is not done when just the `jar` is run directly.

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

6.2.0.1 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!

6.3 Testing the LLVM IR Parser for Part 3

The generated output when running Part 3, which consists of creating the relevant LLVM IR code for the given file works as follows.

The command used to run Part 3:

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

Or:

```
$  java -jar dist/part3.jar test/Sum.gls
```

These are equivalent commands to run the test. The output is the generated LLVM IR (for *Sum.gls* as in the above sections):

```
define i1 @logical_implication(i1 %p, i1 %q) {
entry:
    %not_p = xor i1 %p, true
    %result = or i1 %not_p, %q
    ret i1 %result
}

@.strR = private unnamed_addr constant [3 x i8] c"%d\00", align 1
define i32 @readInt() #0 {
  %1 = alloca i32, align 4
  %2 = call i32 (i8*, ...) @scanf(i8* getelementptr inbounds ([3 x i8], [3 x i8]* @.strR, i32 0, i32 0), i32* %1)
  %3 = load i32, i32* %1, align 4
  ret i32 %3
}

declare i32 @scanf(i8*, ...) #1

@.strP = private unnamed_addr constant [4 x i8] c"%d\0A\00", align 1
define void @println(i32 %x) #0 {
  %1 = alloca i32, align 4
  store i32 %x, i32* %1, align 4
  %2 = load i32, i32* %1, align 4
  %3 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x i8], [4 x i8]* @.strP, i32 0, i32 0), i32 %2)
  ret void
}

declare i32 @printf(i8*, ...) #1

define i32 @main() {
entry:
%a = alloca i32, align 4
%b = alloca i32, align 4
%c = alloca i32, align 4
; Updated variables references
%a_val2 = load i32, i32* %a, align 4
%b_val2 = load i32, i32* %b, align 4
```

```
%c_val2 = load i32, i32* %c, align 4

%a_input = call i32 @readInt()
store i32 %a_input, i32* %a, align 4
%a_val3 = load i32, i32* %a, align 4

; Updated variables references
%a_val4 = load i32, i32* %a, align 4
%b_val3 = load i32, i32* %b, align 4
%c_val3 = load i32, i32* %c, align 4

%b_input = call i32 @readInt()
store i32 %b_input, i32* %b, align 4
%b_val4 = load i32, i32* %b, align 4

; Updated variables references
%a_val5 = load i32, i32* %a, align 4
%b_val5 = load i32, i32* %b, align 4
%c_val4 = load i32, i32* %c, align 4

%prod1 = add i32 0, %a_val5
%prod2 = add i32 0, %b_val5
%arith1 = add i32 %prod1, %prod2
%arith2 = add i32 0, %arith1
store i32 %arith2, i32* %c, align 4
%c_val5 = load i32, i32* %c, align 4

; Updated variables references
%a_val6 = load i32, i32* %a, align 4
%b_val6 = load i32, i32* %b, align 4
%c_val6 = load i32, i32* %c, align 4

; Updated variables references
%a_val7 = load i32, i32* %a, align 4
%b_val7 = load i32, i32* %b, align 4
%c_val7 = load i32, i32* %c, align 4

call void @println(i32 %c_val7)

ret i32 0
}
```

6.3.1 Running with *lli*

To validate the functionality of the LLVM IR code generator, we followed a systematic testing approach that also included our other test *.gls* files which include some edge cases.

Following this, we ran our test cases with *lli* to verify whether the generated code actually functions as a valid executable.

We automated this testing for our own ease (via the *Makefile*, notice the *run-llvm* command):

```
$ make run-llvm TEST_FILE=test/Sum.gls
```

This command runs *make test* along with an additional section defined in the Makefile to run *lli* on the generated output file, which is sent to the *dist/llvm_generated/* directory.

As an example, if we run this command with *Euclid.gls*, this is the output (we would need to enter 2 numbers as that is what the program expects):

```
<Generated LLVM IR code>


Running LLVM assembly for Euclid.ll...
36
24
12
```

The last line above was output by the program, correctly printing the GCD of 36 and 24, which is 12! This was a definitive test as it covers a large portion of the rules we have in our grammar and so we can conclude that our GILLES parser compiler functions as we expect it to!

7 Conclusion

The GILLES compiler project successfully demonstrates the design and implementation of a complete end-to-end compiler pipeline. We started with building a lexer, then transitioned into developing an LL(1) recursive-descent parser by hand and finally developed an LLVM IR code-generating compiler that seamlessly compiles into an executable!

Through this process, we transformed an input GILLES source file *(.gls)* into a functional binary.

Our project achieved its main goal of creating a fully operational compiler that adheres to the provided GILLES grammar and supports core programming constructs such as assignments, conditionals, while loops, and input/output operations. Integrating testing at each stage further ensured the reliability of the implementation.

Future work might include the extensibility of the GILLES language. Features such as support for for-loops, recursive functions, and additional data structures could expand the expressiveness of the language.

Overall, the GILLES compiler project has allowed us to dive deep into the world of language theory and its complexities. This project was a very valuable learning experience! It gave us insight into how compilers really work and the challenges involved in building one from scratch. It's been quite a satisfying process to see everything come to life!