UNIVERSITÉ LIBRE DE BRUXELLES

Introduction to Language Theory and Compiling

Lexer Compiler

Siddharth SAHAY
Jordi UGARTE

Prof. Gilles GERAERTS

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 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 Introduction

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	$\langle Program \rangle \rightarrow$ 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 1.1: 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 me compiled into classes by running the following command:

```
$ make
```

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

```
$ make test
```

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

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

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

Project Structure

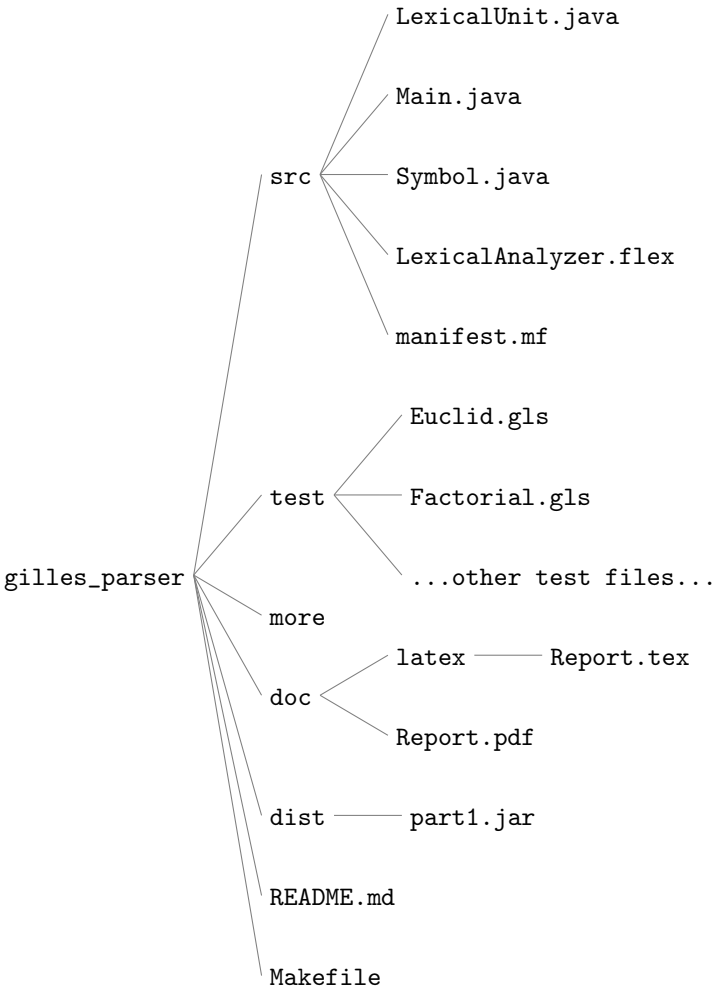

Table 1.2: GILLES Parser project structure

2 Objective

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

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

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

3 Part 1

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 this project, we made the deliberate decision to maintain state:

- **Context-Sensitive Tokens:** Tokens such as `ELSE` and `END` have specific meanings only within certain blocks, such as `IF-THEN-ELSE` or loops. A stateful lexer ensures these tokens are recognized and handled correctly based on their context, preventing misinterpretation.
- **Managing Complex Lexical Structures:** Nested or multi-line constructs, such as conditionals, require the lexer to switch modes. By maintaining states, the lexer can differentiate between normal code and special cases, such as conditional expressions (which may themselves be complex).

- **Simplifying Parser Responsibilities:** By resolving some context-specific tokenization issues at the lexical stage, the lexer reduces complexity for the parser. This allows the parser to focus on building the syntactic structure without handling low-level tokenization issues.
- **Performance Optimisation:** Identifying errors early on in the lexical analysis stage helps point out the issues to users as soon as possible. Since we have a grammar to adhere to already provided, we thought it cogent to have some compliance to the rules there.

We also recognise that there could be a need to simplify these states once work is shifted to the parser - we can revisit this discussion at a later stage to address concerns, or if the lexer begins to have more responsibility than it should.

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:

Table 3.1: Regular expressions used

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

3.2.2 Lexer Rules

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

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

3.2.2.1 The initial `LET` Statement

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

```
1        <Program> → LET [ProgName] BE <Code> END
```

Table 3.2: Initialization of YYINITIAL state

In the lexer, this is represented as:

```
        "LET" { yybegin(PROGRAM);
                        System.out.println(new Symbol(LexicalUnit.LET, yyline, yycolumn, yytext())); }
```

Table 3.3: Initialization of YYINITIAL state in the lexer

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

3.2.2.2 ProgName and transition to the CODE block

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

```
<Code> → <Instruction> : <Code>
<Instruction> → <Assign>
    → <If>
    → <While>
    → <Call>
    → <Output>
    → <Input>
```

Table 3.4: ProgName and transition to the CODE block

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

Table 3.5: ProgName and transition to the CODE block

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

3.2.2.3 Code Block

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

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

```
            new Symbol(LexicalUnit.END, yyline, yycolumn, yytext()));
        }
        ":" { System.out.println(new Symbol(LexicalUnit.COLUMN, yyline, yycolumn, yytext())); }
        {VarName}"="
        { yybegin(ARITHMETIC); System.out.println(new Symbol(LexicalUnit.ASSIGN, yyline, yycolumn, yytext())); }
        "="
        { yybegin(ARITHMETIC); System.out.println(new Symbol(LexicalUnit.ASSIGN, yyline, yycolumn, yytext())); }
        "IN"
        { yybegin(INPUT_OUTPUT); System.out.println(new Symbol(LexicalUnit.INPUT, yyline, yycolumn, yytext())); }
        "OUT"
        { yybegin(INPUT_OUTPUT); System.out.println(new Symbol(LexicalUnit.OUTPUT, yyline, yycolumn, yytext())); }
        "IF"
        { yybegin(CONDITION); System.out.println(new Symbol(LexicalUnit.IF, yyline, yycolumn, yytext())); }
        "WHILE"
        { yybegin(CONDITION); System.out.println(new Symbol(LexicalUnit.WHILE, yyline, yycolumn, yytext())); }
        "ELSE"
        { System.out.println(new Symbol(LexicalUnit.ELSE, yyline, yycolumn, yytext())); }
        {VarName} {
            if (!variables.containsKey(yytext())) {
                variables.put(yytext(), yyline+1);
            }
            System.out.println(new Symbol(LexicalUnit.VARNAME, yyline, yycolumn, yytext()));
        }
        "(" { System.out.println(new Symbol(LexicalUnit.LPAREN, yyline, yycolumn, yytext())); }
        ")" { System.out.println(new Symbol(LexicalUnit.RPAREN, yyline, yycolumn, yytext())); }
}
```

Table 3.6: Implementation of Code Block in the lexer

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

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

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

3.2.2.4 Condition Block

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

```
<If> → IF { <Cond> } THEN <Code> END
    → IF { <Cond> } THEN <Code> ELSE <Code> END
<While> → WHILE {<Cond>} REPEAT <Code> END
```

```
/* Conditionals and loops */
<CONDITION> {
```

```
    "THEN" { yybegin(CODE); System.out.println(new Symbol(LexicalUnit.THEN, yyline, yycolumn, yytext())); }
    "ELSE" { yybegin(CODE); System.out.println(new Symbol(LexicalUnit.ELSE, yyline, yycolumn, yytext())); }
    "REPEAT" { yybegin(CODE); System.out.println(new Symbol(LexicalUnit.REPEAT, yyline, yycolumn, yytext())); }
    "END" { yybegin(CODE); System.out.println(new Symbol(LexicalUnit.END, yyline, yycolumn, yytext())); }
    {Whitespace} { /* Ignore whitespace */ }
    {Number} { System.out.println(new Symbol(LexicalUnit.NUMBER, yyline, yycolumn, yytext())); }
    {VarName} {
        if (!variables.containsKey(yytext())) {
            variables.put(yytext(), yyline+1);
        }
        System.out.println(new Symbol(LexicalUnit.VARNAME, yyline, yycolumn, yytext()));
    }
    "==" { System.out.println(new Symbol(LexicalUnit.EQUAL, yyline, yycolumn, yytext())); }
    "<=" { System.out.println(new Symbol(LexicalUnit.SMALEQ, yyline, yycolumn, yytext())); }
    "<" { System.out.println(new Symbol(LexicalUnit.SMALLER, yyline, yycolumn, yytext())); }
    "|" { System.out.println(new Symbol(LexicalUnit.PIPE, yyline, yycolumn, yytext())); }
    "->" { System.out.println(new Symbol(LexicalUnit.IMPLIES, yyline, yycolumn, yytext())); }
    "(" { System.out.println(new Symbol(LexicalUnit.LPAREN, yyline, yycolumn, yytext())); }
    ")" { System.out.println(new Symbol(LexicalUnit.RPAREN, yyline, yycolumn, yytext())); }
    ":" { System.out.println(new Symbol(LexicalUnit.COLUMN, yyline, yycolumn, yytext())); }
    "{" { System.out.println(new Symbol(LexicalUnit.LBRACK, yyline, yycolumn, yytext())); }
    "}" { System.out.println(new Symbol(LexicalUnit.RBRACK, yyline, yycolumn, yytext())); }
}
```

Table 3.7: Condition block implementation

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

3.2.2.5 Assignment Statements

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

```
<Assign> → [VarName] = <ExprArith>
```

Table 3.8: Assignment of variables

This rule is implemented in the lexer as:

```
{VarName}"=" { yybegin(ARITHMETIC);
               System.out.println(new Symbol(LexicalUnit.ASSIGN, yyline, yycolumn, yytext())); }
```

Table 3.9: Implementation of assignment of variables

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

3.2.2.6 Arithmetic Expressions

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

```
<ExprArith> → [VarName]
    → [Number]
    → ( <ExprArith> )
    → - <ExprArith>
    → <ExprArith> <Op> <ExprArith>
```

Table 3.10: Arithmetic expressions

In the lexer, this is implemented as follows:

```
/* Arithmetic expressions */
    <ARITHMETIC> {
        {Whitespace} { /* Ignore whitespace */ }
          {Number} { System.out.println(new Symbol(LexicalUnit.NUMBER, yyline, yycolumn, yytext())); }
          {VarName} {
            if (!variables.containsKey(yytext())) {
                variables.put(yytext(), yyline+1);
            }
            System.out.println(new Symbol(LexicalUnit.VARNAME, yyline, yycolumn, yytext()));
        }
          "+" { System.out.println(new Symbol(LexicalUnit.PLUS, yyline, yycolumn, yytext())); }
          "-" { System.out.println(new Symbol(LexicalUnit.MINUS, yyline, yycolumn, yytext())); }
          "*" { System.out.println(new Symbol(LexicalUnit.TIMES, yyline, yycolumn, yytext())); }
          "/" { System.out.println(new Symbol(LexicalUnit.DIVIDE, yyline, yycolumn, yytext())); }
          "(" { System.out.println(new Symbol(LexicalUnit.LPAREN, yyline, yycolumn, yytext())); }
          ")" { yybegin(CODE); System.out.println(new Symbol(LexicalUnit.RPAREN, yyline, yycolumn,
   yytext())); }
          ":" { yybegin(CODE); System.out.println(new Symbol(LexicalUnit.COLUMN, yyline, yycolumn,
   yytext())); }
        }

```

Table 3.11: Implementation of arithmetic expressions

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

3.2.2.7 Input and Output

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

```
/* Input/Output instructions */
<INPUT_OUTPUT> {
    {Whitespace} { /* Ignore whitespace */ }
    "(" { System.out.println(new Symbol(LexicalUnit.LPAREN, yyline, yycolumn, yytext())); }
=   ")" { yybegin(CODE); System.out.println(new Symbol(LexicalUnit.RPAREN, yyline, yycolumn, yytext())); }
    {VarName} {
        if (!variables.containsKey(yytext())) {
            variables.put(yytext(), yyline+1);
        }
        System.out.println(new Symbol(LexicalUnit.VARNAME, yyline, yycolumn, yytext()));
    }
    ":" { System.out.println(new Symbol(LexicalUnit.COLUMN, yyline, yycolumn, yytext())); }
}
```

Table 3.12: Input and output operations

3.2.2.8 Conditionals and Loops

Conditionals and loops are defined in the grammar as follows:

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

Table 3.13: Conditionals and loops

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

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

Table 3.14: Conditionals and loops implementation

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

3.2.2.9 Input and Output

Input and output operations are defined as:

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

In the lexer, these are handled in the `INPUT_OUTPUT` state:

Table 3.15: Input and output operations

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

Table 3.16: Input and output operations implementation

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

3.2.2.10 Whitespace and Comments

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

```
/* Ignore whitespace and comments */
{Whitespace} { /* Ignore whitespace */ }
{Comment} { /* Ignore comments */ }
{ShortComment} { /* Ignore short comments */ }
```

Table 3.17: Whitespaces and comments

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

3.2.2.11 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!
```

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.

Table 3.18: Whitespaces and comments example

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:

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

Table 3.19: Exception in case of error in the lexer

4 Testing

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

4.1 Test Files

- `Euclid.gls` - A simple program to calculate the greatest common divisor of two numbers.
- `InvalidSymbolEuclid.gls` - Same as above but with an invalid symbol, .
- `Sum.gls` - A simple program to calculate the sum of two numbers.
- `ThreeLoopGibberish.gls` - An 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.

4.2 Running the Tests

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

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

```
1                token: LET              lexical unit: LET
2                token: sum              lexical unit: PROGNAME
3                token: BE               lexical unit: BE
4                token: IN               lexical unit: INPUT
5                token: (                lexical unit: LPAREN
```

```
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 4.1: Test datasheet