Université libre de Bruxelles

École Polytechnique de Bruxelles

M-IRIFS

INFO-F403 - Introduction to language theory and compiling - 2023/2024

PASCALmaispresque

Project – Part 1

Membres :

Younes El Mokhtari

Ismail Jeq

Professeur :

Gilles GEERAERTS

23 octobre 2023

Table des matières

1 Description of the project

1.1 Introduction

The LexicalAnalyzer is a critical component of a larger language processing system designed to tokenize and process input text written in a specific programming language.

1.2 Features

— Recognizes keywords, operators, variables, and numbers.

— Handles comments, including both long comments (denoted by ”’) and short comments (denoted by ’**’).

— Reports and handles unrecognized symbols.

1.3 Implementation Details

— Generated as a Java class named `LexicalAnalyzer.java`.

— Utilizes Java to define the lexer rules and actions.

— Supports Unicode input, character counting, line counting, and column counting.

— Returns tokens of type `Symbol`, which include information such as token type, line, and column.

1.4 Lexer States

— Four lexer states : `YYINITIAL`, `LONG_COMMENT_STATE`, `SHORT_COMMENT_STATE` and `EXIT_STATE`.

1.5 Token Recognition

— Recognizes keywords like "begin," "end," "if," "while," and others.

— Identifies operators such as "+," "-", "*", "/", and more.

— Captures variables and numbers.

— Skips spaces and end-of-line characters.

— Reports and handles unrecognized symbols.

1.6 Custom Actions

— Includes custom Java code to track variables and output token information.

— Handles the end-of-file situation, unclosed comments, unrecognized tokens and returns an end-of-stream token.

1.7 Usage

— Typically integrated into a compiler or interpreter for a specific programming language to analyze the source code.

2 Regular expressions

`"begin"`, `"end"` `"..."`, `":="`, `"("`, `")"`, `"-"`, `"+"`, `"*"`, `"/"`, `"if"`, `"then"`, `"else"`, `"AND"`, `"OR"`, `"["`, `"]"`, `"="`, `"<"`, `"while"`, `"do"`, `"print"`, `"read"`, `{Variable}`, `{Number}`, `{Space}`, `{EndOfLine}`, `.`

Where the Macros are :

```
AlphaUpperCase = [A-Z]
AlphaLowerCase = [a-z]
Alpha = {AlphaUpperCase}|{AlphaLowerCase}
Numeric = [0-9]
AlphaNumeric = {Alpha}|{Numeric}
Variable = {AlphaLowerCase}{AlphaNumeric}*
Number = {Numeric}+
Space = "\t" | " "
EndOfLine = "\r"?"\n"
```

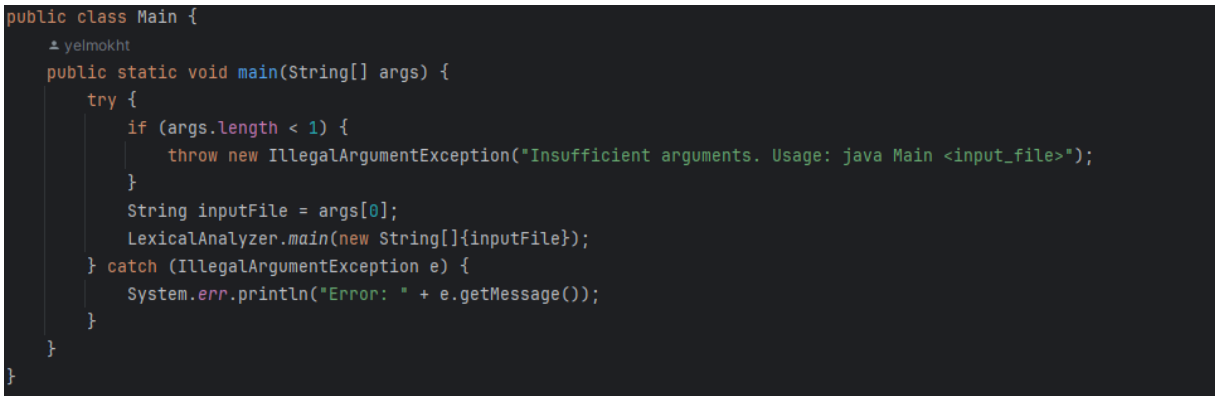
3 Implementation choices

Regarding the implementation choices, the program functions as follows :

In the `LexicalAnalyzer.flex` file, we define various regular expressions that allow us to scan and process input files. The `LexicalAnalyzer.java` file is then generated by the JFlex tool.

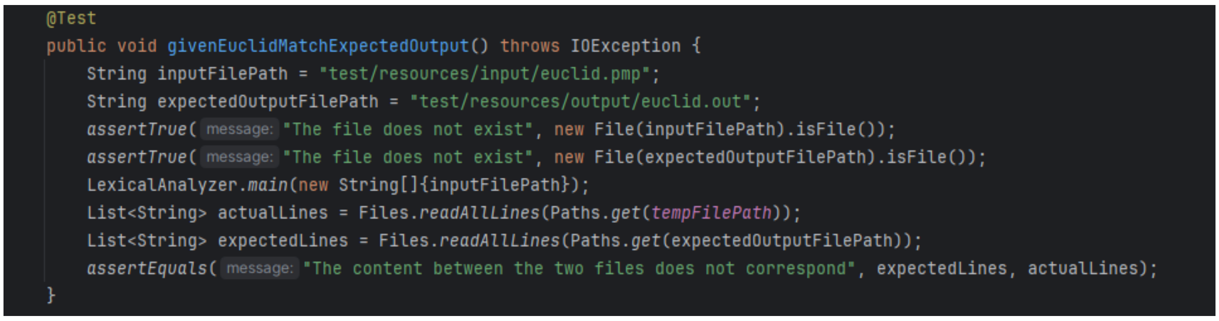
The `Main` file serves to receive the given argument file and pass it to the scanner by calling the `LexicalAnalyzer` class. The result of the scanner is finally displayed on the terminal.

To test our scanner, we created different test files and submitted them to it. To automate these tests, we created a `TestLexicalAnalyzer` class that verifies that the scanner's output matches the expected output.



```java
public class Main {
    ± yelmokht
    public static void main(String[] args) {
        try {
            if (args.length < 1) {
                throw new IllegalArgumentException("Insufficient arguments. Usage: java Main <input_file>");
            }
            String inputFile = args[0];
            LexicalAnalyzer.main(new String[]{inputFile});
        } catch (IllegalArgumentException e) {
            System.err.println("Error: " + e.getMessage());
        }
    }
}
```

FIGURE 1 – Main



```java
    @Test
    public void givenEuclidMatchExpectedOutput() throws IOException {
        String inputFilePath = "test/resources/input/euclid.pmp";
        String expectedOutputFilePath = "test/resources/output/euclid.out";
        assertTrue( message: "The file does not exist", new File(inputFilePath).isFile());
        assertTrue( message: "The file does not exist", new File(expectedOutputFilePath).isFile());
        LexicalAnalyzer.main(new String[]{inputFilePath});
        List<String> actualLines = Files.readAllLines(Paths.get(tempFilePath));
        List<String> expectedLines = Files.readAllLines(Paths.get(expectedOutputFilePath));
        assertEquals( message: "The content between the two files does not correspond", expectedLines, actualLines);
    }
```

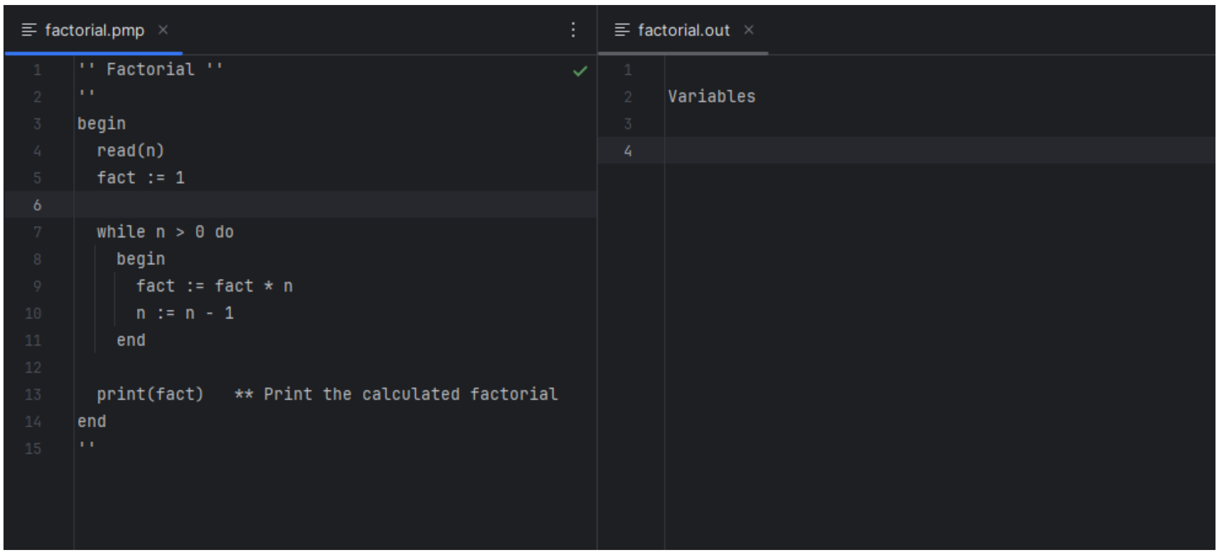
FIGURE 2 – TestLexicalAnalyzer

4 Example files

For this project, it was required to test and provide our own example files, ensuring their relevance. To do so, we identified three scenarios in which our scanner implementation can be tested :

4.1 Ignoring multi-line comments

Long comments are inherently ignored by the scanner as it is a required functionality for the project. To further validate this, we tested if multi-line comments are also ignored using the file `factorial.pmp` :



factorial.pmp

```
'' Factorial ''
''
begin
  read(n)
  fact := 1

  while n > 0 do
    begin
      fact := fact * n
      n := n - 1
    end

  print(fact)   ** Print the calculated factorial
end
''
```

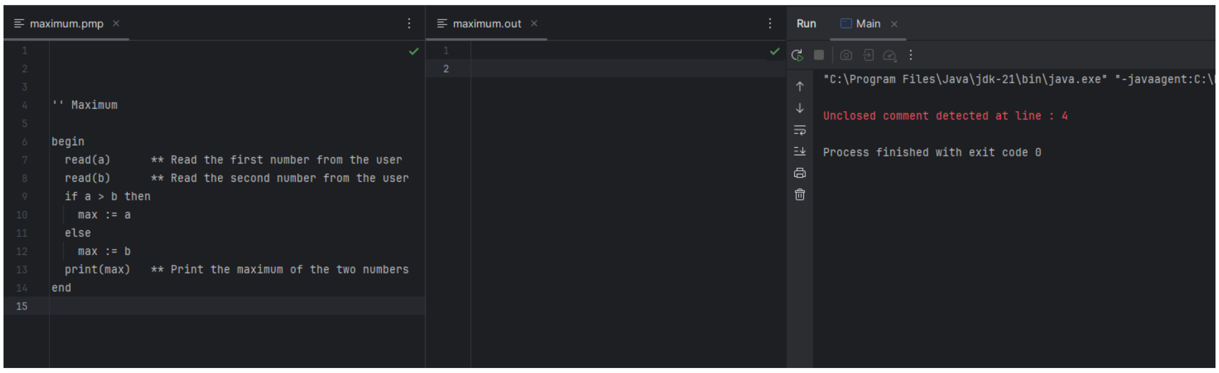
factorial.out

```

Variables


```

FIGURE 3 – Factorial

4.2 Proper closure of comments

Another aspect to test is the proper closure of comments. For short comments, closure occurs with a newline character. However, for long comments, the '' symbol must be at the end. Since the project instructions do not explicitly state that comments must be closed, we tested this functionality with the file `maximum.pmp`. If the scanner reaches the end of a file while still in the LONG COMMENT state, it displays an error message indicating the line where the comment is not closed :



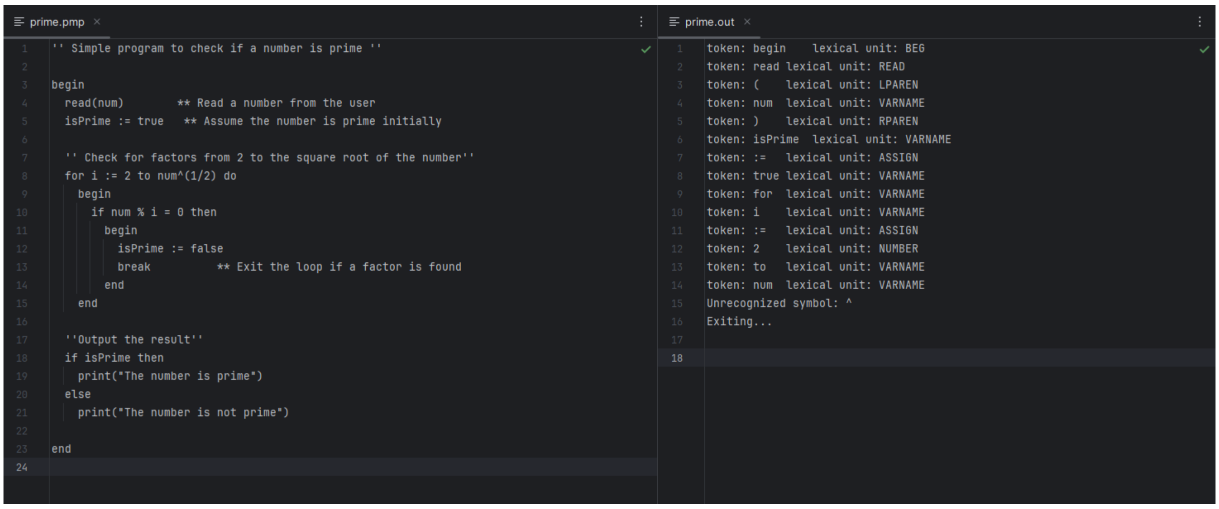
```
'' Maximum

begin
  read(a)      ** Read the first number from the user
  read(b)      ** Read the second number from the user
  if a > b then
    max := a
  else
    max := b
  print(max)   ** Print the maximum of the two numbers
end
```

```
"C:\Program Files\Java\jdk-21\bin\java.exe" "-javaagent:C:\

Unclosed comment detected at line : 4

Process finished with exit code 0
```

FIGURE 4 – Maximum

4.3 Identifying unrecognized symbols

For the project, a symbol table was provided to scan input files. However, some symbols are not included in the table and need to be distinguished. We tested with the file `prime.pmp` to ensure that unrecognized symbols are indicated by the scanner with the message : "Unrecognized symbol." After detecting an unrecognized symbol, the scanner doesn't read the rest of the file and exits.



```
'' Simple program to check if a number is prime ''

begin
  read(num)        ** Read a number from the user
  isPrime := true   ** Assume the number is prime initially

  '' Check for factors from 2 to the square root of the number''
  for i := 2 to num^(1/2) do
    begin
      if num % i = 0 then
        begin
          isPrime := false
          break          ** Exit the loop if a factor is found
        end
    end

  ''Output the result''
  if isPrime then
    print("The number is prime")
  else
    print("The number is not prime")

end
```

```
token: begin    lexical unit: BEG
token: read lexical unit: READ
token: (    lexical unit: LPAREN
token: num  lexical unit: VARNAME
token: )    lexical unit: RPAREN
token: isPrime  lexical unit: VARNAME
token: :=   lexical unit: ASSIGN
token: true lexical unit: VARNAME
token: for  lexical unit: VARNAME
token: i    lexical unit: VARNAME
token: :=   lexical unit: ASSIGN
token: 2    lexical unit: NUMBER
token: to   lexical unit: VARNAME
token: num  lexical unit: VARNAME
Unrecognized symbol: ^
Exiting...
```

FIGURE 5 – Prime