



MARMARA UNIVERSITY

FACULTY OF ENGINEERING

CSE 2260

Principles of Programming Languages

Project 1

Scanner

Submitted to: Serap Korkmaz

Due Date: 14.04.2023

	Dept	Student Id	Name Surname	Participation
1	CSE	150121822	Barış Giray AKMAN	Code
2	CSE	150121058	Musa ÖZKAN	Code, Report
3	CSE	150121076	Abdullah Kan	-

Table of Contents

1. ABSTRACT	3
2. HOW TO RUN THE CODE.....	4
3. FUNCTIONS.....	6
A. isBracketExist() Function	6
B. printBracket() Function	6
C. isHexorBin() Function	6
D. booleanLiterals() Function	6
E. charLiterals() Function	7
F. identifierElementCheck() Function	7
G. checkIdentifierFirstElement () Function	7
H. identifierLiteral() Function	8
I. keywordLiteral() Function	8
J. numberLiteral() Function	8
H. main() Function	9
4. INPUTS & OUTPUTS.....	9
A. Given Inputs	9
B. Custom Inputs	14
5. CONCLUSION & WHERE TO IMPROVE	19

1. ABSTRACT

In this project for **Principles of Programming Languages Course**, we are tasked with implementing a **lexical analyzer** or **scanner** for a programming language called **PPLL**. This language includes various token types such as brackets, number literals, boolean literals, character literals, string literals, keywords, and identifiers. The scanner must accurately recognize these tokens and output them with their positions in the input file.

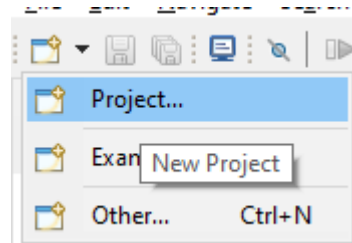
To implement this scanner, we will use **Java Programming Language** without any regular expression libraries. It is crucial to consider each token's construction and differentiate between them accurately. For example, number literals can be either integer or floating-point, and there are three different types of integer numbers - decimal signed integers, hexadecimal unsigned integers, and binary unsigned integers. We will also need to handle errors appropriately, such as when an incorrect token is found, halt the scanner, and report the error message along with its position in the input file.

Overall, this project challenges our understanding of **lexical analysis**, **programming skills** and **teamwork capabilities**. It requires careful attention to detail and precise coding skills to produce an efficient and accurate scanner. Successfully completing this project will demonstrate our ability to apply our knowledge practically and produce high-quality code.

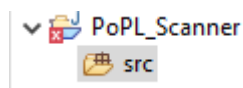
2. HOW TO RUN THE CODE

In this part of the report, the detailed explanation on how to run our implementation of scanner in Java program in Eclipse is expressed. The purpose of this part is to explain how to run the program with ease and to ensure that anyone can accurately verify its performance.

Firstly, please open Eclipse and create a new Java project by selecting "File" > "New" > "Java Project". This will create a new project folder within your workspace and allow you to store all of your Java files in one place.

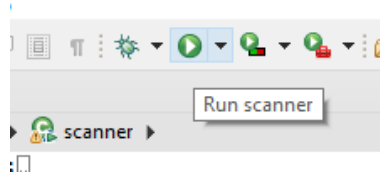


Next, add the Java class that we sent within this project by right-clicking on the "src" folder and selecting "New" > "Class".

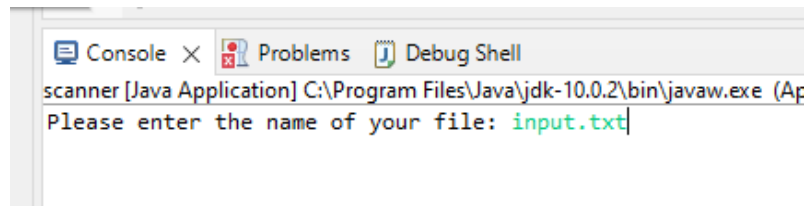


After adding the Java file, add the input.txt file in the same directory as the Java code.

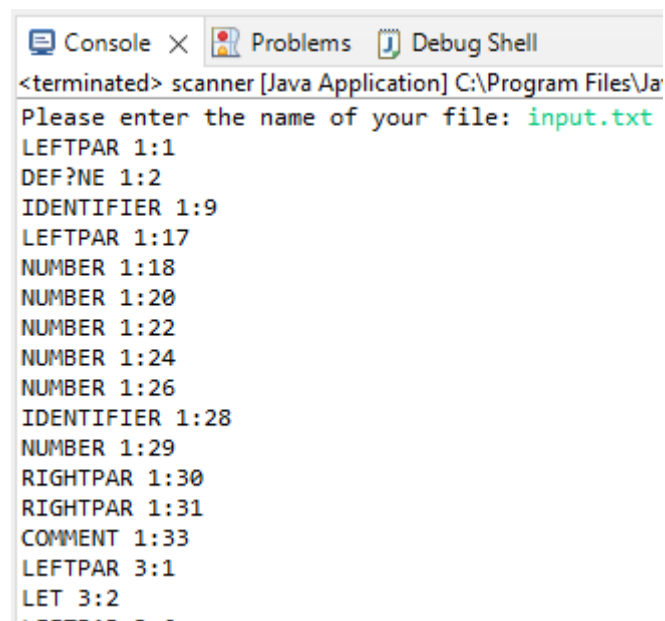
Then, run the program by right-clicking on the Java file and selecting "Run As" > "Java Application". This should execute the code and generate the output file.



After running the code, you have to write the input files name which is “input.txt” and then the code will give the output both from the console and the “output.txt”.



Finally, check the output file to verify whether the program has worked as intended. If there are any errors or issues with the program, Eclipse will usually display them in the Console window.



3. FUNCTIONS

A. isBracketExist() Function

This Function checks if there is a bracket on the index or not.

B. printBracket() Function

This Function checks and prints which bracket type is on the index.

C. isHexorBin() Function

This function checks if the number given is hexadecimal or binary by checking the digits 'b' or 'x'.

D. booleanLiterals() Function

This Function searches for Boolean literals (i.e., 'true' or 'false') within the given line, and if it finds one, it adds a message to an outputs list indicating the position of the Boolean literal. The method loops through each character of the line, starting from the current index, until it reaches either the end of the line or a space character. If the current characters make up the word "true" or "false", the method adds the message to the outputs list and returns true. Otherwise, it continues looping until the end of the line is reached and returns false if no Boolean literal is found.

E. charLiterals() Function

This Function searches for character literals (i.e., a single character surrounded by single quotes) within the given line, and if it finds one, it adds a message to an outputs list indicating the position of the character literal. The method checks whether the character at the “current_index” of the line equals the ASCII value for a single quote (39). If so, it enters the loop, then it checks whether the length of the line is less than current_index+2 or whether the character at the position current_index+2 is not a single quote and prints an error message and exits the program if either condition is true. Otherwise, it adds a string to the outputs list indicating that a char has been found on the current line and at the current index position and increments the “current_index” value accordingly. Finally, if the length of the line is less than current_index+3, it prints an error message and exits the program. If the method finds a char literal, it returns true, otherwise, it returns false.

F. identifierElementCheck() Function

This Function checks if the current index is suitable to be an identifier.

G. checkIdentifierFirstElement () Function

This Function checks if the first character is suitable to be an identifier.

H. identifierLiteral() Function

This Function searches for identifier literals (i.e., names of variables, functions, or classes) within the given line, and if it finds one, it adds a message to an outputs list indicating the position of the identifier literal. The method loops through each character of the line, starting from the current index value, until it reaches either the end of the line or a space character. If the first character is a valid element of an identifier, the method enters the loop and checks whether each subsequent character is also a valid element. If not, the method stores the position of the error and exits the program with an error message. Otherwise, it adds a message to the outputs list and increments the “current_index” value accordingly. If the method finds an identifier literal, it returns true, otherwise, it returns false.

I. keywordLiteral() Function

The method searches for keyword literals (i.e., special words with reserved meaning) the position of the keyword literal. The method loops through each character of the line, starting from the current index value, until it reaches either the end of the line or a space character. If the current character is not a bracket, the character is appended to an empty string identifier. After the loop completes, the method checks whether the identifier equals one of the keywords "define", "let", "cond", "if", or "begin". If so, it adds a string to the outputs list indicating that a keyword has been found on the current line and at the current index position. It increments the “current_index” value accordingly. If the method finds a keyword literal, it returns true, otherwise, it returns false.

J. numberLiteral() Function

This Function tokenize and validate number literals in a given input string. The function returns a Boolean value - true if a valid number literal is found and false otherwise. The function starts by checking if the current index (initialized as 0 outside the function) exceeds or equals the length of the input string. If so, it indicates that there are no more tokens to process, and the function returns true. Next, the code checks for hexadecimal notation and binary notation using a separate function

called `isHexorBin()`. If either is found, the input is further processed and validated according to these notations. If the input is not a hexadecimal or binary number, the code checks if the first character of the token is a dot sign, a plus sign, a minus sign, or a digit between 0 and 9. If so, it suggests that it is a number literal or a floating-point literal. The input is then further processed and validated accordingly. During the processing of the token, several error-checking conditions are implemented. For example, it checks for invalid characters, more than one exponential symbol, multiple dot signs, and other issues that may indicate an invalid token. If any of these errors are detected, the function prints out an error message with the line and position of the invalid token and exits the program. Otherwise, the function returns true, indicating that a valid number literal was found.

H. `main()` Function

This Function starts by prompting the user for the name of the file to open and then creates a new File object with that name. It then creates a Scanner object to read from the file. The code then iterates through each line of the file using a while loop that checks whether there are any more lines left to read using the `hasNextLine()` method of the Scanner. Within the loop, the code again iterates through each character in the current line, checking for various types of literals using the functions previously defined (such as `numberLiterals()`, `keywordLiteral()`, etc.). The code also checks for string literals with opening and closing quotes and sets a boolean flag `string_literal` accordingly. If an invalid string literal is found, the program prints out an error message and exits. After processing each token in the line, the program increments the current index and continues the loop until all tokens have been processed in the current line. Once all lines have been processed, the program closes the input scanner.

4. INPUTS & OUTPUTS

A. Given Inputs

While implementing the code, we tried 3 example input files that given to us. Those were the inputs and the outputs of our code.

Inputs:

1)

```
*input.txt - Not Defteri
Dosya Düzen Biçim Görünüm Yardım
(define (fibonacci n)
  (let fib ([prev 0]
            [cur 1]
            [i 0])
    (if (= i n)
        cur
        (define fibonacci
          (fib cur (+ prev cur) (+ i 1))))))
```

2)

```
*input.txt - Not Defteri
Dosya Düzen Biçim Görünüm Yardım
(define (fibonacci n)
  (let fib ([prev 0]
            [cur 1]
            [i 0])
    (if (= i n)
        cur
        (fib cur (+ prev cur) (+ i 1))))))
```

3)

```
input.txt - Not Defteri
Dosya Düzen Biçim Görünüm Yardım
(define (sum a1 b1)
  (+ a1 b1))
(define myval (sum (sum -1.234E+9 +7.001e-2) .12))
(define myval2 (+ +7.001e-2 -1.234E+9 2 -5 .12))
(define str "I am A String With Digits17=8+9, 3:6/2 \"abc\" ****")
(define !-abc "again an identifier called as !-abc")
(define m ((lambda (y z)
              (+ (- z y) (- z y)))
 3 -5))
(define (<d-is-smaller d e)
  (cond [(< d e) 1 ]
        [else 0 ]))
(define myval3 (+ +7.001e-2 -1.234E+9 2 -5 .12 +4 8E))
```

Outputs:

1)


```
scanner.java  input.txt X
1 (define (sum a1 b1)
<
<terminated> scanner [Java Application] C:\Program Files\Java
Please enter the name of your file: input.txt
LEFTPAR 1:1
DEF?NE 1:2
LEFTPAR 1:9
IDENTIFIER 1:10
IDENTIFIER 1:14
IDENTIFIER 1:17
RIGHTPAR 1:19
LEFTPAR 2:1
IDENTIFIER 2:2
IDENTIFIER 2:4
IDENTIFIER 2:7
RIGHTPAR 2:9
RIGHTPAR 2:10
LEFTPAR 3:1
DEF?NE 3:2
IDENTIFIER 3:9
LEFTPAR 3:15
IDENTIFIER 3:16
LEFTPAR 3:20
IDENTIFIER 3:21
IDENTIFIER 3:25
NUMBER 3:26
NUMBER 3:35
RIGHTPAR 3:44
IDENTIFIER 3:46
NUMBER 3:47
RIGHTPAR 3:49
RIGHTPAR 3:50
LEFTPAR 4:1
DEF?NE 4:2
IDENTIFIER 4:9
LEFTPAR 4:16
IDENTIFIER 4:17
```

```
scanner.java  input.txt X
1 (define (sum a1 b1)
<
<terminated> scanner [Java Application] C:\Program Fi
LEFTPAR 4:16
IDENTIFIER 4:17
NUMBER 4:19
IDENTIFIER 4:29
NUMBER 4:30
NUMBER 4:39
IDENTIFIER 4:41
NUMBER 4:42
IDENTIFIER 4:44
NUMBER 4:45
RIGHTPAR 4:47
RIGHTPAR 4:48
LEFTPAR 5:1
DEF?NE 5:2
IDENTIFIER 5:9
STRING 5:13
RIGHTPAR 5:66
LEFTPAR 6:1
DEF?NE 6:2
IDENTIFIER 6:9
STRING 6:15
RIGHTPAR 6:52
LEFTPAR 7:1
DEF?NE 7:2
IDENTIFIER 7:9
LEFTPAR 7:11
LEFTPAR 7:12
IDENTIFIER 7:13
LEFTPAR 7:20
IDENTIFIER 7:21
IDENTIFIER 7:23
RIGHTPAR 7:24
LEFTPAR 8:10
IDENTIFIER 8:11
```

Console × Problems Debug Shell

<terminated> scanner [Java Application] C:\Program

LEFTPAT 8:10
IDENTIFIER 8:11
LEFTPAT 8:13
IDENTIFIER 8:14
IDENTIFIER 8:16
IDENTIFIER 8:18
RIGHTPAT 8:19
LEFTPAT 8:21
IDENTIFIER 8:22
IDENTIFIER 8:24
IDENTIFIER 8:26
RIGHTPAT 8:27
RIGHTPAT 8:28
RIGHTPAT 8:29
NUMBER 9:1
IDENTIFIER 9:3
NUMBER 9:4
RIGHTPAT 9:5
RIGHTPAT 9:6
LEFTPAT 10:1
DEF?NE 10:2
LEFTPAT 10:9
IDENTIFIER 10:10
IDENTIFIER 10:24
IDENTIFIER 10:26
RIGHTPAT 10:27
LEFTPAT 11:3
COND 11:4
LEFTSQUAREB 11:9
LEFTPAT 11:10
IDENTIFIER 11:11
IDENTIFIER 11:13
IDENTIFIER 11:15
RIGHTPAT 11:16

Console × Problems Debug Shell

<terminated> scanner [Java Application] C:\Program Fi

IDENTIFIER 10:26
RIGHTPAT 10:27
LEFTPAT 11:3
COND 11:4
LEFTSQUAREB 11:9
LEFTPAT 11:10
IDENTIFIER 11:11
IDENTIFIER 11:13
IDENTIFIER 11:15
RIGHTPAT 11:16
NUMBER 11:18
RIGHTSQUAREB 11:20
LEFTSQUAREB 12:9
IDENTIFIER 12:10
NUMBER 12:15
RIGHTSQUAREB 12:17
RIGHTPAT 12:18
RIGHTPAT 12:19
LEFTPAT 14:1
DEF?NE 14:2
IDENTIFIER 14:9
LEFTPAT 14:16
IDENTIFIER 14:17
NUMBER 14:19
IDENTIFIER 14:29
NUMBER 14:30
NUMBER 14:39
IDENTIFIER 14:41
NUMBER 14:42
IDENTIFIER 14:44
NUMBER 14:45
NUMBER 14:48
LEXICAL ERROR [14:51]: Invalid token '8E'

B. Custom Inputs

After implementing our code, we created example inputs to be able to detect any errors as we are figuring out if there is any other **edge case**.

Inputs:

1)

```
input.txt - Not Defteri
Dosya Düzen Biçim Görünüm Yardım
(define 0x1ff_2b #b101010 ~ This is a comment
(let ((a 12) (b -0.5) (c 1e+2) (d #\a) (e "hello \"world\"")) ~ Another comment
  (cond ((< a b) (display (+ c 3.14159))) ((= a b) (display d) (newline)) (else (display e))))
(if true "yes" "no")
(begin (define x 42) (+ x 8))
)
```

2)

```
*input.txt - Not Defteri
Dosya Düzen Biçim Görünüm Yardım
(define 0x1ff_2b #b1010E10 ~ Define a hexadecimal number and binary number with underscores and a comment
)

(let ((a 12) (b -0.5) (c 1e+2) (d #\a) (e "hello \"world\"")) ~ Define variables a, b, c, d, and e using let binding
  (cond ((< a b) (display (+ c 3.14159))) ((= a b) (display d) (newline)) (else (display e))))
~ Use a conditional statement to display different values depending on the values of a and b

(if true 'yes 'no) ~ Use an if statement to return 'yes' if true, 'no' otherwise

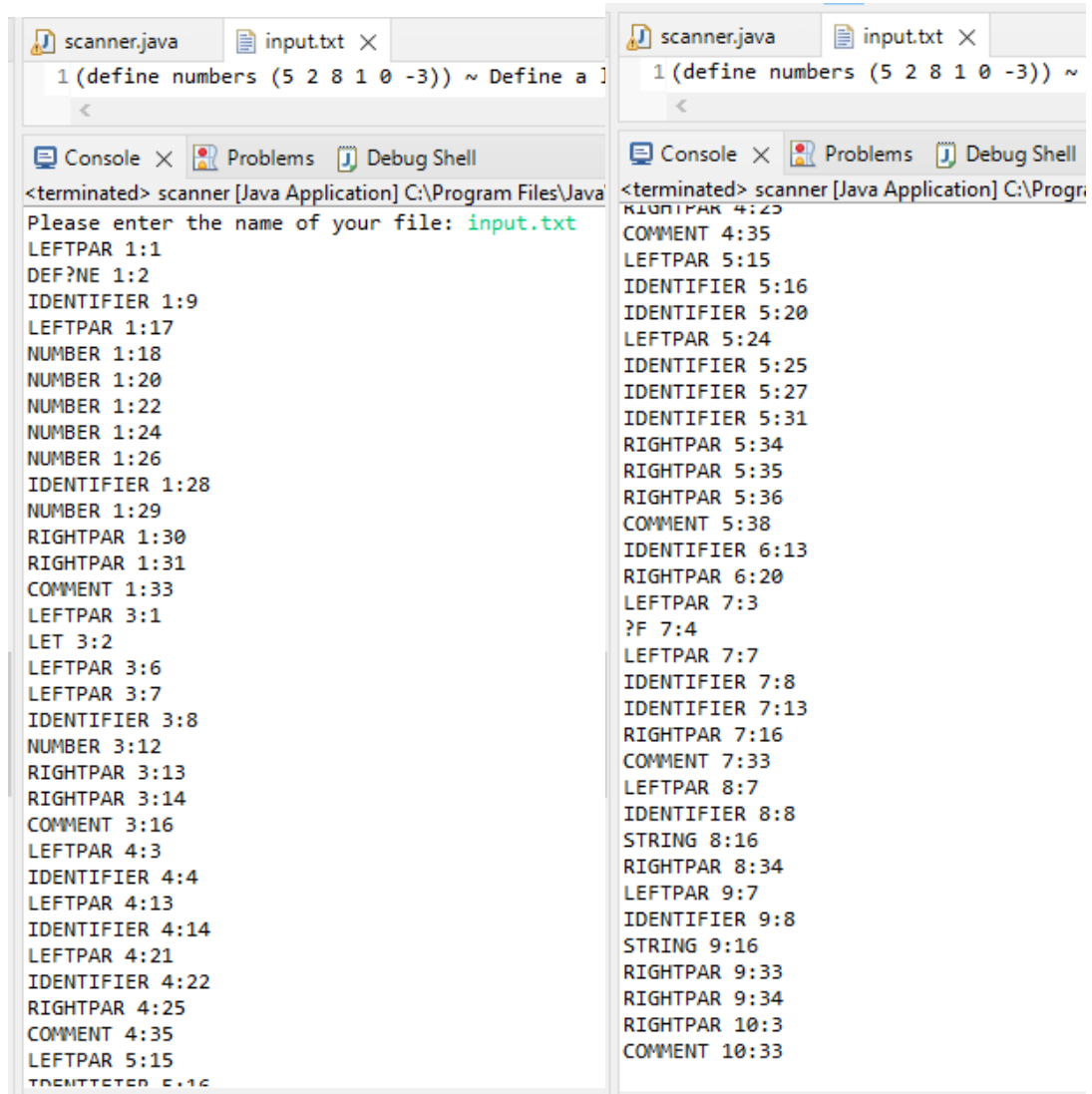
(begin ~ Use a begin expression to define a variable and add it to another number
  (define x 42)
  (+ x 8)))
```

3)

```
*input.txt - Not Defteri
Dosya Düzen Biçim Görünüm Yardım
(let ((sum 0)) ~ Define a variable to store the sum of the numbers
  (for-each (lambda (num) ~ Use a for loop to iterate over each number in the list
    (set! sum (+ sum num))) ; Add the current number to the sum
    numbers)
  (if (even? sum) ~ Check if the sum is even using the even? function
    (display "The sum is even.")
    (display "The sum is odd. "))
  ) ~ Close the let expression
```

Outputs:

1)



```
scanner.java  input.txt X
1 (define numbers (5 2 8 1 0 -3)) ~ Define a

<terminated> scanner [Java Application] C:\Program Files\Java
Please enter the name of your file: input.txt
LEFTPAR 1:1
DEF?NE 1:2
IDENTIFIER 1:9
LEFTPAR 1:17
NUMBER 1:18
NUMBER 1:20
NUMBER 1:22
NUMBER 1:24
NUMBER 1:26
IDENTIFIER 1:28
NUMBER 1:29
RIGHTPAR 1:30
RIGHTPAR 1:31
COMMENT 1:33
LEFTPAR 3:1
LET 3:2
LEFTPAR 3:6
LEFTPAR 3:7
IDENTIFIER 3:8
NUMBER 3:12
RIGHTPAR 3:13
RIGHTPAR 3:14
COMMENT 3:16
LEFTPAR 4:3
IDENTIFIER 4:4
LEFTPAR 4:13
IDENTIFIER 4:14
LEFTPAR 4:21
IDENTIFIER 4:22
RIGHTPAR 4:25
COMMENT 4:35
LEFTPAR 5:15
IDENTIFIER 5:16

scanner.java  input.txt X
1 (define numbers (5 2 8 1 0 -3)) ~

<terminated> scanner [Java Application] C:\Program Files\Java
RIGHTPAR 4:25
COMMENT 4:35
LEFTPAR 5:15
IDENTIFIER 5:16
IDENTIFIER 5:20
LEFTPAR 5:24
IDENTIFIER 5:25
IDENTIFIER 5:27
IDENTIFIER 5:31
RIGHTPAR 5:34
RIGHTPAR 5:35
RIGHTPAR 5:36
COMMENT 5:38
IDENTIFIER 6:13
RIGHTPAR 6:20
LEFTPAR 7:3
?F 7:4
LEFTPAR 7:7
IDENTIFIER 7:8
IDENTIFIER 7:13
RIGHTPAR 7:16
COMMENT 7:33
LEFTPAR 8:7
IDENTIFIER 8:8
STRING 8:16
RIGHTPAR 8:34
LEFTPAR 9:7
IDENTIFIER 9:8
STRING 9:16
RIGHTPAR 9:33
RIGHTPAR 9:34
RIGHTPAR 10:3
COMMENT 10:33
```

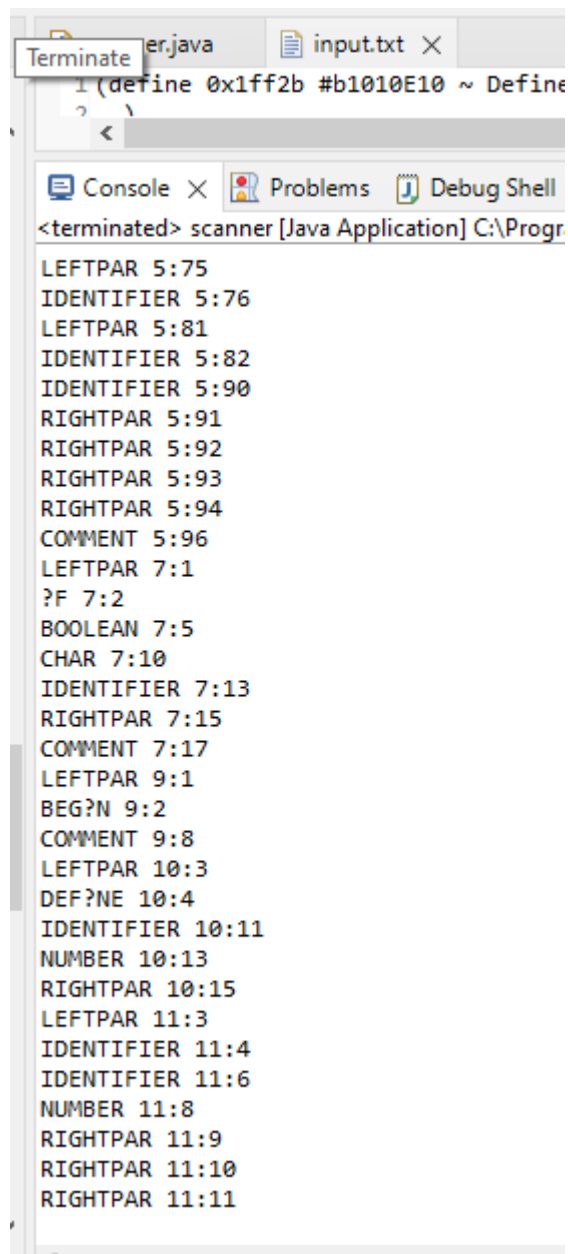
2)

```
scanner.java  input.txt X
1 (define 0x1ff2b #b1010E10 ~ Define a hexade
? \
<

Console X Problems Debug Shell
<terminated> scanner [Java Application] C:\Program Files\Java
Please enter the name of your file: input.txt
LEFTPAR 1:1
DEF?NE 1:2
NUMBER 1:9
IDENTIFIER 1:18
COMMENT 1:27
RIGHTPAR 2:3
LEFTPAR 4:1
LET 4:2
LEFTPAR 4:6
LEFTPAR 4:7
IDENTIFIER 4:8
NUMBER 4:10
RIGHTPAR 4:12
LEFTPAR 4:14
IDENTIFIER 4:15
IDENTIFIER 4:17
NUMBER 4:18
RIGHTPAR 4:21
LEFTPAR 4:23
IDENTIFIER 4:24
NUMBER 4:26
RIGHTPAR 4:30
LEFTPAR 4:32
IDENTIFIER 4:33
IDENTIFIER 4:35
RIGHTPAR 4:36
LEFTPAR 4:38
IDENTIFIER 4:39
STRING 4:41
RIGHTPAR 4:58
RIGHTPAR 4:59
COMMENT 4:61

scanner.java  input.txt X
1 (define 0x1ff2b #b1010E10 ~ Define a hexi
? \
<

Console X Problems Debug Shell
<terminated> scanner [Java Application] C:\Program Files\
LEFTPAR 5:3
COND 5:4
LEFTPAR 5:9
LEFTPAR 5:10
IDENTIFIER 5:11
IDENTIFIER 5:13
IDENTIFIER 5:15
RIGHTPAR 5:16
LEFTPAR 5:18
IDENTIFIER 5:19
LEFTPAR 5:27
IDENTIFIER 5:28
IDENTIFIER 5:30
NUMBER 5:32
RIGHTPAR 5:39
RIGHTPAR 5:40
RIGHTPAR 5:41
LEFTPAR 5:43
LEFTPAR 5:44
IDENTIFIER 5:45
IDENTIFIER 5:47
IDENTIFIER 5:49
RIGHTPAR 5:50
LEFTPAR 5:52
IDENTIFIER 5:53
IDENTIFIER 5:61
RIGHTPAR 5:62
LEFTPAR 5:64
IDENTIFIER 5:65
RIGHTPAR 5:72
RIGHTPAR 5:73
LEFTPAR 5:75
IDENTIFIER 5:76
```

The screenshot shows a Java IDE window with two tabs: 'er.java' and 'input.txt'. The 'er.java' tab is active, displaying a single line of code: `1 (define 0x1ff2b #b1010E10 ~ Define`. A 'Terminate' button is visible over the first tab. Below the editor, the 'Console' tab is selected, showing the output of a Java application named 'scanner'. The output consists of a series of tokens and their positions, indicating the scanner's progress through the input file.

```
<terminated> scanner [Java Application] C:\Progr  
LEFTPAR 5:75  
IDENTIFIER 5:76  
LEFTPAR 5:81  
IDENTIFIER 5:82  
IDENTIFIER 5:90  
RIGHTPAR 5:91  
RIGHTPAR 5:92  
RIGHTPAR 5:93  
RIGHTPAR 5:94  
COMMENT 5:96  
LEFTPAR 7:1  
?F 7:2  
BOOLEAN 7:5  
CHAR 7:10  
IDENTIFIER 7:13  
RIGHTPAR 7:15  
COMMENT 7:17  
LEFTPAR 9:1  
BEG?N 9:2  
COMMENT 9:8  
LEFTPAR 10:3  
DEF?NE 10:4  
IDENTIFIER 10:11  
NUMBER 10:13  
RIGHTPAR 10:15  
LEFTPAR 11:3  
IDENTIFIER 11:4  
IDENTIFIER 11:6  
NUMBER 11:8  
RIGHTPAR 11:9  
RIGHTPAR 11:10  
RIGHTPAR 11:11
```

3)

The screenshot shows an IDE with two main panes. The left pane contains a code editor with a file named `scanner.java` and a file explorer showing `input.txt`. The code in `scanner.java` is a simple scanner that reads a file and prints its contents. The right pane shows the debug console output, which includes the program's execution flow and the contents of the input file.

```
1 (define numbers (5 2 8 1 0 -3)) ~ Define a
2
<terminated> scanner [Java Application] C:\Program Files\Java
COMMENT 4:35
LEFTPAR 5:15
IDENTIFIER 5:16
IDENTIFIER 5:20
LEFTPAR 5:24
IDENTIFIER 5:25
IDENTIFIER 5:27
IDENTIFIER 5:31
RIGHTPAR 5:34
RIGHTPAR 5:35
RIGHTPAR 5:36
COMMENT 5:38
IDENTIFIER 6:13
RIGHTPAR 6:20
LEFTPAR 7:3
?F 7:4
LEFTPAR 7:7
IDENTIFIER 7:8
IDENTIFIER 7:13
RIGHTPAR 7:16
COMMENT 7:33
LEFTPAR 8:7
IDENTIFIER 8:8
STRING 8:16
RIGHTPAR 8:34
LEFTPAR 9:7
IDENTIFIER 9:8
STRING 9:16
RIGHTPAR 9:33
RIGHTPAR 9:34
RIGHTPAR 10:3
COMMENT 10:33
```

5. CONCLUSION & WHERE TO IMPROVE

In this project, we built a lexical analyzer (scanner) for the PPLL programming language. This scanner was designed to recognize specific tokens in the input source code, such as Brackets, Number literals, Boolean literals, Character literals, String literals, Keywords, and Identifiers. We defined regular expressions for each of these token types, allowing our scanner to identify them in the input stream.

The scanner was implemented in either Java, and it was required to output the sequence of tokens present in the input source file, one token per line. Additionally, each token needed to be annotated with its position in the input file. This information is useful for debugging purposes and for identifying specific errors in the source code.

Finally, we also had to handle any lexically incorrect inputs by producing a single error message that would indicate the nature of the error and the position in the input where it occurred. Once an error was detected, the scanner would stop analyzing the input stream and report the error message. This feature ensured that the user would receive immediate feedback on any errors present in their input code.

We detected that when the program writes the Lexical Error part to the file, it writes “LE” part and then it stops. The problem occurs only when the program writes the output to an output.txt file and the problem isn’t happening when it prints the output to the Terminal.