

Marmara University - Faculty of Engineering  
Department of Computer Engineering

CSE2260 Principles of Programming Languages (Spring 2023)

Submit Date: 16/04/2023

Scanner Implementation (Lexical Analyzer)

Student Number (ID)	Name	Surname
150120998	Abdelrahman	Zahran
150121538	Muhammed	Enes Gökdeniz
150120022	Tolga	Fehmioğlu

Sections Of the Report: -

- Section (1): Problem Definition.
- Section (2): Implementation (Code).
- Section (3): Explanation.
- Section (4): Test Cases.

**Section (1): Definition: -**

A lexical analyzer, also known as a lexer or a scanner, is a component of a compiler that reads the source code of a program and converts it into a stream of tokens that can be processed by the compiler's parser.

The lexical analyzer scans the source code character by character, grouping them into meaningful units called tokens. It ignores whitespace and comments and identifies keywords, identifiers, operators, literals, and other elements of the programming language syntax.

The tokens produced by the lexical analyzer are usually represented by an integer code that identifies the type of the token, along with any associated data such as the value of a literal or the name of an identifier. The tokens are passed on to the parser, which uses them to construct a parse tree that represents the structure of the program's syntax.

The lexical analyzer is an essential component of the compiler, as it provides the foundation for subsequent phases of the compilation process. It must be able to handle a wide range of input, including edge cases and invalid syntax, and produce meaningful error messages when it encounters errors in the input.

**Section (2): Implementation**

In the Implementation of this project, we used **Java 20** programming language and its imported libraries provided and supported by oracle.

▪ default package: -

- Project Class



Project.java

■ Imported Libraries: -

- Import java.io.\*;
- Import java.util.\*;

■ Input Files: -



Input\_1.txt



Input\_2.txt



Input\_3.txt

■ Output File: -

- Depends on the input file!

### Section (3): Explanation

#### Main Overview:

To implement a lexical analyzer for the PPLL programming language, we need to perform the following steps:

- Read the input file character by character.
- Ignore whitespaces and comments.
- Recognize and tokenize Brackets, Number literals, Boolean literals, Character literals, String literals, Keywords, and Identifiers.
- Output each token with its position in the input.

To implement the lexical analyzer, we can use a finite-state machine that reads characters from the input file and transitions to different states based on the input. Each state represents a possible token type, and transitions between states correspond to recognizing characters that match a regular expression for the current state.

We can implement the lexical analyzer in Java or C. In either case, we can use a class or struct to represent each token, with fields for the token type, token value (if applicable), and the position of the token in the input file. We can use an array or list to store the sequence of tokens and output each token with its position after scanning the entire input file.

To handle errors, we can throw an exception or print an error message with the position of the first incorrect token and terminate the scanning process.

A breakdown of the steps involved in implementing a lexical analyzer for the given programming language, PPLL:

- Understand the problem statement: The first step is to carefully read and understand the problem statement. This involves identifying the tokens that need to be recognized, the regular expressions for each token, and any special rules or constraints that apply.
- Design a lexer: The next step is to design a lexer, which is a program that reads in the input file and produces a stream of tokens. The lexer should be designed to implement the rules and constraints specified in the problem statement.
- Implement the lexer: Once the lexer has been designed, the next step is to implement it in code. The lexer should be written in Java or C, as specified in the problem statement. The lexer should be designed to handle both correct and incorrect input, and should produce appropriate error messages for any lexically incorrect input.

- Test the lexer: After the lexer has been implemented, it should be thoroughly tested to ensure that it produces the correct output for a variety of inputs. This includes testing the lexer on both correct and incorrect input, as well as testing edge cases.
- Output the token stream: Finally, the lexer should be modified to output the sequence of tokens in the input, one token per line. Each token should be annotated with the position of the token in the input, as specified in the problem statement.

In summary, implementing a lexical analyzer for the PPL programming language involves understanding the problem statement, designing and implementing a lexer, testing the lexer, and outputting the token stream.

```
//-----
//
// A HashMap containing mappings of brackets to their respective names
public static HashMap<Character, String> brackets = new HashMap<>().putAll(
    Map.of(
        '(', "LEFTPAR",
        ')', "RIGHTPAR",
        '[', "LEFTSQUAREB",
        ']', "RIGHTSQUAREB",
        '{', "LEFTCURLYB",
        '}', "RIGHTCURLYB"));

// An ArrayList containing keywords used in the language
public static ArrayList<String> keywords = new ArrayList<>().addAll(
    Arrays.asList(
        "define",
        "let",
        "cond",
        "if",
        "begin"));

// An ArrayList containing boolean values used in the language
private static ArrayList<String> booleans = new ArrayList<>().addAll(
    Arrays.asList(
        "true",
        "false"));

// An ArrayList containing various signs used in the language
public static ArrayList<Character> signs = new ArrayList<>().addAll(
    Arrays.asList(
        '!',
        '*',
        '/',
        ':',
        '<',
        '=',
        '>',
        '?'));

// An ArrayList containing arithmetic signs used in the language
public static ArrayList<Character> arithmeticSign = new ArrayList<>().addAll(
    Arrays.asList(
        '+',
        '-',
        '.',
        '*'));

//-----
//
```

This code snippet defines several data structures used in a programming language implementation.

The first data structure is a HashMap called brackets. It maps specific characters, such as (, ), [, {, and }, to corresponding string names, such as "LEFTPAR", "RIGHTPAR", "LEFTSQUAREB", "LEFTCURLYB", and "RIGHTCURLYB". These names are likely used in parsing and interpreting code written in the programming language.

The second data structure is an ArrayList called keywords. It contains several strings that represent keywords in the programming language, including "define", "let", "cond", "if", and "begin". Keywords are reserved words in a programming language that have special meanings and cannot be used as variable or function names.

The third data structure is another ArrayList called booleans. It contains two strings, "true" and "false", which represent the boolean values in the programming language. Boolean values are typically used to represent true/false or yes/no conditions in programming.

The fourth data structure is an ArrayList called signs. It contains several characters that represent various signs used in the programming language, including "!", "\*", "/", ":", "<", "=", ">", and "?". These signs are likely used for comparisons, logical operations, or other purposes in the programming language.

Finally, the fifth data structure is an ArrayList called arithmeticSign. It contains three characters that represent arithmetic signs used in the programming language, including "+", "-", and ".". These arithmetic signs are likely used for mathematical calculations in the programming language.

```
//-----//
// This is a Java program that reads input from a file specified by the user
// and writes the output to a file named "Output.txt".
public static void main(String[] args) throws IOException {
    // Initialize variables-----//
    // Initialize ArrayList to store lines
    ArrayList<String> lines = new ArrayList<>();
    // Initialize ArrayList to store tokens
    ArrayList<String> tokens = new ArrayList<>();
    // Initialize ArrayList to store errors
    ArrayList<String> errors = new ArrayList<>();
    // token position
    int[] rowColumn = {1, 1};

    Scanner input = new Scanner(System.in);
    try (input) {
        // Prompt the user to enter the file name
        System.out.print("Enter Input File Name: ");
        String inputName = input.next();

        // Create a File object for input and output
        File inputFile = new File(inputName);
        File outputFile = new File("Output.txt");

        Scanner scanner = new Scanner(inputFile);
        FileWriter writer = new FileWriter(outputFile);

        try (scanner) {
            while (scanner.hasNextLine()) {
                lines.add(scanner.nextLine());
            }
            addTokens(lines, tokens, errors, rowColumn);
            int last = tokens.size() - 1;
            try (writer) {
                // Print tokens list
                for (String token : tokens) {
                    writer.write(token);
                    System.out.print(token);
                    if (tokens.indexOf(token) != last) {
                        writer.write("\n");
                        System.out.println();
                    }
                }

                // Print errors list
                for (String error : errors) {
                    System.out.println(error);
                }
            }
            // Catch any IO exceptions thrown while writing to the file
        } catch (IOException Ex) {
            // Print the error message
            System.err.println("Error writing to file: " + Ex.getMessage());
            // Return to exit the program
            return;
        }

    }

    // Catch the exception thrown when the file is not found
} catch (FileNotFoundException Ex) {
    // Print the stack trace for the exception
}
```

```

        System.err.println("Invalid Input File!");
        Ex.printStackTrace();
    }

    //-----//
    -----//

```

This is a Java program that reads input from a file specified by the user and writes the output to a file named "Output.txt". The program initializes several variables, including ArrayLists to store lines, tokens, and errors, as well as an array to store the position of tokens.

The program prompts the user to enter the name of the input file, creates File objects for both input and output, and then uses a Scanner to read in the lines of the input file. It then calls a method named "addTokens" to tokenize the input and add the tokens to the tokens ArrayList.

The program then creates a FileWriter object to write the tokens to the output file and writes each token to the file and prints it to the console. Finally, the program catches any IOExceptions that may occur while writing to the file, prints an error message, and exits.

If the input file is not found, the program catches the FileNotFoundException, prints an error message, and exits.

```

//-----//
-----//
// This method identifies tokens from the lines of the input file and store them
private static void addTokens(ArrayList<String> lines, ArrayList<String> tokens,
ArrayList<String> errors,int [] rowColumn) {
    String str = "";
    int size;
    for (String currentLine : lines) {

        for (int j = 0; j < currentLine.length(); j++) {

            char ch = currentLine.charAt(j);
            // check if the character is a bracket
            if (isBracket(j, currentLine)) {
                // get the corresponding closing bracket and add the position information
                str = brackets.get(ch) + String.format(" %d:%d", rowColumn[0], rowColumn[1]);
                // add the current token to the list of tokens
                tokens.add(str);
                rowColumn[1]++;

                // check if the character is the start of a keyword
            } else if (isHasKeyword(j, currentLine)) {
                // get the keyword and add the position information
                size = getSizeOf(j, currentLine);
                String newStr = currentLine.substring(j, j + size);
                str = newStr.toUpperCase(Locale.ROOT) + String.format(" %d:%d", rowColumn[0],
rowColumn[1]);

                // add the current token to the list of tokens
                tokens.add(str);
                j += (size - 1);
                rowColumn[1] += size;

                // check if the character is a whitespace
            } else if (ch == ' ') {
                rowColumn[1]++;

                // check if the character is the start of a char literal
            } else if (isChar(j, currentLine)) {
                // add the char literal and position information
                str = "CHAR" + String.format(" %d:%d", rowColumn[0], rowColumn[1]);
                // add the current token to the list of tokens
                tokens.add(str);
                rowColumn[1] += 3;
                j += 2;

                // check if the character is the start of a boolean literal

```

```

    } else if (isBoolean(j, currentLine)) {
        // get the boolean literal and add the position information
        size = getSizeOf(j, currentLine);
        str = currentLine.substring(j, j + size);
        str = str.toUpperCase(Locale.ROOT) + String.format(" %d:%d", rowColumn[0],
rowColumn[1]);
        // add the current token to the list of tokens
        tokens.add(str);
        j += (size - 1);
        rowColumn[1] += size;

    // check if the character is a tilde
    } else if (ch == '~') {
        // add the tilde token and position information
        str = "TILDE" + String.format("%d:%d", rowColumn[0], rowColumn[1]);
        // add the current token to the list of tokens
        tokens.add(str);
        break;

    // check if the character is the start of a string literal
    } else if (isStringLiteral(j, currentLine)) {
        // get the string literal and add the position information
        int lastIndex = getQuotesIndex(j, currentLine);
        str = "STRING " + String.format("%d:%d", rowColumn[0], rowColumn[1]);
        // add the current token to the list of tokens
        tokens.add(str);
        size = lastIndex - j + 1;
        j += size - 1;
        rowColumn[1] += size;

    // check if the character is the start of an identifier
    } else if (isIdentifier(j, currentLine)) {
        // add the identifier and position information
        str = "IDENTIFIER " + String.format("%d:%d", rowColumn[0], rowColumn[1]);
        // add the current token to the list of tokens
        tokens.add(str);
        size = getSizeOf(j, currentLine);
        j += size - 1;
        rowColumn[1] += size;

    // check if the character is the start of a number
    } else if (isNumber(j, currentLine)) {
        // add the number and position information
        str = "NUMBER " + String.format("%d:%d", rowColumn[0], rowColumn[1]);
        // add the current token to the list of tokens
        tokens.add(str);
        size = getSizeOf(j, currentLine);
        j += size - 1;
        rowColumn[1] += size;
    } else {
        int error_size = getSizeOf(j, currentLine);
        String errorPart = currentLine.substring(j, j + error_size);
        String err_mes = "LEXICAL ERROR" + String.format("[%d:%d]: Invalid token
's'", rowColumn[0], rowColumn[1], errorPart);
        // add the Error Message to the list of tokens
        tokens.add(err_mes);
        // add the error to the list of errors
        errors.add(str);
        return;
    }
}
// Increase the row value(move to next row)
rowColumn[0]++;
// Set Column value to 1
rowColumn[1] = 1;

    }
}
//-----//
-----//

```

This is a Java method that takes in three ArrayLists of Strings: lines, tokens, and errors, as well as an integer array rowColumn. The purpose of this method is to tokenize each line in the lines ArrayList and add the resulting tokens to the tokens ArrayList. If there is a lexical error, an error message is added to the tokens ArrayList and the error itself is added to the errors ArrayList.

The method uses a loop to iterate through each line in the lines ArrayList. For each line, the loop uses another loop to iterate through each character in the line. Depending on the character, the method identifies the token and adds it to the tokens ArrayList.

The isBracket method is called to check if the character is a bracket. If it is, the method gets the corresponding closing bracket from a HashMap called brackets and adds it to the token along with the row and column information. The resulting token is then added to the tokens ArrayList.

The isHasKeyword method is called to check if the character is the start of a keyword. If it is, the method determines the length of the keyword and adds it to the token in uppercase form, along with the row and column information. The resulting token is then added to the tokens ArrayList.

If the character is a whitespace, the method simply increments the column value in the rowColumn array.

The isChar method is called to check if the character is the start of a character literal. If it is, the method adds "CHAR" to the token along with the row and column information. The resulting token is then added to the tokens ArrayList, and the column value is incremented by 3 to account for the additional characters in the token.

The isBoolean method is called to check if the character is the start of a boolean literal. If it is, the method determines the length of the literal and adds it to the token in uppercase form, along with the row and column information. The resulting token is then added to the tokens ArrayList.

If the character is a tilde, the method adds "TILDE" to the token along with the row and column information. The resulting token is then added to the tokens ArrayList.

The isStringLiteral method is called to check if the character is the start of a string literal. If it is, the method determines the length of the string and adds "STRING" to the token along with the row and column information. The resulting token is then added to the tokens ArrayList, and the column value is incremented by the length of the string.

The isIdentifier method is called to check if the character is the start of an identifier. If it is, the method adds "IDENTIFIER" to the token along with the row and column information. The resulting token is then added to the tokens ArrayList.

The isNumber method is called to check if the character is the start of a number. If it is, the method adds "NUMBER" to the token along with the row and column information. The resulting token is then added to the tokens ArrayList.

If none of the above conditions are met, the method assumes that the character is an invalid token and generates an error message. The message is added to the tokens ArrayList, and the error itself is added to the errors ArrayList.

Finally, after processing each line, the row value in the rowColumn array is incremented, and the column value is reset to 1 to prepare for the next line.

In summary, the method addTokens takes in three ArrayLists (lines, tokens, and errors) and an array rowColumn as input parameters. It is responsible for analyzing each character in the input lines of code, identifying the corresponding tokens, and adding them to the tokens ArrayList. If it encounters an invalid token, it adds an error message to the errors ArrayList.

The method loops through each line of the code and each character in the line. For each character, it checks if it is a bracket, keyword, whitespace, char literal, boolean literal, tilde, string literal, identifier, or number. If it matches any of these criteria, it identifies the corresponding token and adds it to the tokens ArrayList, along with its position information.

If the character does not match any of these criteria, it identifies the invalid token and adds an error message to the errors ArrayList. It then returns, stopping the tokenization process.

Finally, the method increases the row value and sets the column value to 1 to move to the next row. This process continues until all characters in all lines are analyzed.

```
//-----//
// This method checks if the input string is a number.
private static boolean isNumber(int index, String CurrentLine) {

    return isDecimalSignedInteger(index, CurrentLine) || isHexadecimalUnsignedInteger(index,
CurrentLine) || isBinaryUnsignedInteger(index, CurrentLine) || isFloatingPointNumber(index,
CurrentLine);
}
//-----//
//-----//
```

This is a Java method that checks whether a given string is a number. The method takes two parameters: an integer index and a string CurrentLine. The index parameter represents the position of the character in CurrentLine that the method should start checking from.

The method returns a boolean value indicating whether the input string is a number or not. The method checks the input string against four different criteria to determine if it is a number. These four criteria are:

isDecimalSignedInteger: This method checks whether the input string represents a signed decimal integer. A signed decimal integer is a number that can have a positive or negative sign and can be expressed in base 10.

isHexadecimalUnsignedInteger: This method checks whether the input string represents an unsigned hexadecimal integer. A hexadecimal integer is a number that is expressed in base 16 and uses the digits 0-9 and the letters A-F to represent values from 0 to 15.

isBinaryUnsignedInteger: This method checks whether the input string represents an unsigned binary integer. A binary integer is a number that is expressed in base 2 and uses the digits 0 and 1 to represent values of 0 and 1.

isFloatingPointNumber: This method checks whether the input string represents a floating-point number. A floating-point number is a number that contains a decimal point and can be expressed in scientific notation, such as 1.23E-4.

The isNumber method returns true if the input string matches any of the four criteria above, and false otherwise.

```
//-----//
// This method checks if a given character is a valid hexadecimal digit.
private static boolean isHexadecimalDigit(Character ch) {
    // The method uses a switch statement to match the input character against a list of
valid hexadecimal digits,
// which includes both upper-case and lower-case letters from 'A' to 'F', as well as
digits from '0' to '9'.
    return switch (ch) {
        case '0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'a', 'b', 'c', 'd', 'e', 'f',
'A', 'B', 'C', 'D', 'E', 'F' ->
            true;
        default -> false;
    };
}
//-----//
//-----//
```

This method is used to determine whether a given character is a valid hexadecimal digit.

The method takes a single argument, which is a Character object representing the character to be checked. It then uses a switch statement to match the input character against a list of valid hexadecimal digits.



The list of valid hexadecimal digits includes both uppercase and lowercase letters from 'A' to 'F', as well as digits from '0' to '9'.

If the input character matches one of the valid hexadecimal digits, the method returns true. If the input character does not match any of the valid hexadecimal digits, the method returns false.

Overall, this method is used as a helper method for other methods that deal with hexadecimal numbers, such as the `isHexadecimalUnsignedInteger()` method used in the `isNumber()` method described earlier.

---

```
//-----//
// This method get size of (distance) between 2 characters / lexemes
private static int getSizeOf(int startIndex, String CurrentLine) {
    // Get the character at the specified index.
    char ch = CurrentLine.charAt(startIndex);
    // Get the index of the last character in the string.
    int lastIndex = CurrentLine.length() - 1;
    // Initialize a counter to keep track of the size.
    int count = 0;
    // Continue looping until a bracket or whitespace is encountered.
    while (!isBracket(startIndex, CurrentLine) && !Character.isSpaceChar(ch)) {
        // If we've reached the end of the line, increment the counter and exit the loop.
        if (lastIndex == startIndex) {
            count++;
            break;
        }
        // Increment the counter and move to the next character.
        count++;
        startIndex++;
        ch = CurrentLine.charAt(startIndex);
    }
    // Return the size of the string.
    return count;
}
//-----//
```

This is a method that returns the size of a lexeme, which is the distance between two characters in a string. It takes in two arguments, an integer `startIndex` and a `String CurrentLine`. The `startIndex` specifies the starting position of the lexeme, and the `CurrentLine` is the string in which the lexeme is located.

The method starts by getting the character at the specified index (`startIndex`) using the `charAt()` method. It then gets the index of the last character in the string (`lastIndex`) using the `length()` method.

Next, it initializes a counter variable (`count`) to keep track of the size of the lexeme. It then enters a loop that continues until a bracket or whitespace character is encountered. Inside the loop, it checks if the current character is the last character in the string. If so, it increments the counter and exits the loop.

If the current character is not the last character in the string, the method increments the counter, moves to the next character, and retrieves the character at the new index using the `charAt()` method. This process continues until a bracket or whitespace character is encountered.

Finally, the method returns the size of the lexeme, which is stored in the `count` variable.

---

```
//-----//
// Check if the number is a single digit first
private static boolean isDecimalSignedInteger(int startIndex, String CurrentLine) {

    char ch = CurrentLine.charAt(startIndex);
    int lastIndex = CurrentLine.length() - 1;

    // If the number has a sign
    if (ch == '-' || ch == '+') {
        // If the sign is at the end of the line, it cannot be a number
    }
}
```

```

        if (startIndex == lastIndex) return false;
        startIndex++;
        ch = CurrentLine.charAt(startIndex);
        // If a bracket or a space comes right after the sign, it cannot be a number
        if (isBracket(startIndex, CurrentLine) || Character.isSpaceChar(ch)) {
            return false;
        }
    }

    // Check the rest of the digits
    while (!isBracket(startIndex, CurrentLine) && !Character.isSpaceChar(ch)) {
        // If the character is not a digit, it cannot be a number
        if (!Character.isDigit(ch)) {
            return false;
        }
        // If the index is at the end of the line, break the loop
        if (lastIndex == startIndex) {
            break;
        }
        startIndex++;
        ch = CurrentLine.charAt(startIndex);
    }

    // If none of the above conditions apply, it must be a valid signed decimal integer
    return true;
}
//-----//
-----//

```

This is a method that checks whether a given string represents a signed decimal integer or not. The method takes two parameters, the starting index of the string and the string itself.

The first step is to check if the number is a single digit. If it is, the method returns true. If not, it moves on to the next step.

The method then checks whether the number has a sign (+ or -) at the beginning. If it does, it checks if the sign is followed by a bracket or a space. If it is, the method returns false because it cannot be a valid signed decimal integer. Otherwise, it moves on to the next step.

The method then checks the rest of the string character by character until it reaches a bracket or a space. If any of the characters is not a digit, the method returns false. If it reaches the end of the string without encountering a bracket or a space, the method returns true, indicating that the string represents a valid signed decimal integer.

In summary, this method checks whether a given string represents a signed decimal integer or not by checking whether it starts with a sign (+ or -) followed by a digit, and whether the rest of the string contains only digits.

```

//-----//
-----//
// This method searches for the index of the closing quotation mark of a string literal
// starting from the given 'start' index in the 'CurrentLine' string
private static int getQuotesIndex(int start, String CurrentLine) {
    char ch;
    int index = start;
    for (int i = start + 1; i < CurrentLine.length(); i++) {
        ch = CurrentLine.charAt(i);
        // If a closing quotation mark is found, update the 'index' variable to the current
index
        if (ch == '"') {
            index = i;
        }
    }
    return index;
}
//-----//
-----//

```

This is a method that searches for the index of the closing quotation mark of a string literal starting from the given 'start' index in the 'CurrentLine' string.

The method starts by declaring a character variable named 'ch' and an integer variable named 'index' which is initialized with the start index. A for loop is then used to iterate through the characters in the string starting from the index next to the 'start' index.

Within the loop, each character is checked to see if it matches a closing quotation mark. If a closing quotation mark is found, the 'index' variable is updated to the current index where the closing quotation mark was found.

Once the loop has finished iterating through all the characters in the string, the 'index' variable is returned, which will hold the index of the closing quotation mark for the string literal starting from the given 'start' index.

In summary, this method helps in finding the end of a string literal by searching for the index of the closing quotation mark, which can be useful in parsing and analyzing code that contains string literals.

```
//-----//
// This method check if the token is a string literal or not
private static boolean isStringLiteral(int start, String CurrentLine) {

    char ch = CurrentLine.charAt(start);

    if (ch != '"') {
        return false;
    }

    // String literals cannot be of length 1
    if (CurrentLine.length() - start <= 2) return false;

    // String literals cannot start with double quotes
    if (CurrentLine.charAt(start + 1) == '"') return false;

    // Check if there is a closing quote
    if (!remainContainsQuote(start, CurrentLine)) {
        return false;
    }

    // Get the index of the closing quote
    int quoteIndex = getQuotesIndex(start, CurrentLine);

    // Loop through the characters in the string literal
    for (int i = start + 1; i < quoteIndex; i++) {
        ch = CurrentLine.charAt(i);
        if (ch == '\\') {
            // Handle escape sequences
            int next = i + 1;
            char nextChar = CurrentLine.charAt(next);
            if (nextChar != '\\' && nextChar != '"') {
                return false;
            }
            if (nextChar == '"') {
                if (next == quoteIndex) {
                    return false;
                }
            }
        } else if (ch == '"') {
            // Double quotes must be escaped
            int prev = i - 1;
            char prevChar = CurrentLine.charAt(prev);
            if (prevChar != '\\') {
                return false;
            }
        }
    }

    return true;
}
```

```
//-----  
-----//
```

This code snippet contains a method called `isStringLiteral()` that takes two arguments: `start`, which is the starting index of a potential string literal in a string called `CurrentLine`. The method returns a boolean value indicating whether the string starting at the `start` index is a valid string literal or not.

The method first checks whether the character at the `start` index is a double quote (`"`). If it is not, then it is not a string literal, and the method returns false.

Next, it checks if the length of the string is greater than two. This is because a string literal should have at least one character inside it, in addition to the opening and closing double quotes.

The method then checks if the second character is also a double quote. If it is, then it is not a string literal and the method returns false.

After that, the method checks if the given string contains a closing double quote. If it does not, then the string is not a valid string literal and the method returns false.

If there is a closing quote, the method gets its index using the `getQuotesIndex()` method.

Then, the method loops through each character in the potential string literal starting from the index of the opening quote to the index of the closing quote. If it encounters a backslash (`\`), it checks whether the next character is either another backslash or a double quote. If it is not, then it is not a valid escape sequence and the method returns false.

If the next character is a double quote, it also checks whether it is the closing quote. If it is, then it is not a valid string literal and the method returns false.

If the character is not a backslash, the method checks whether it is a double quote. If it is, it checks whether the previous character is a backslash. If it is not, then it is not a valid string literal and the method returns false.

If the method passes all these checks, then the potential string literal is a valid string literal, and the method returns true.

```
//-----  
-----//  
// Method to check if the rest of the line contains a double quote character  
private static boolean remainContainsQuote(int start, String CurrentLine) {  
    char ch = CurrentLine.charAt(start);  
    // Iterate over the rest of the string, starting from the given index  
    for (int i = start + 1; i < CurrentLine.length(); i++) {  
        // Check if a double quote is found  
        if (CurrentLine.charAt(i) == '"') {  
            return true;  
        }  
    }  
    // Return false if no double quote is found  
    return false;  
}  
//-----  
-----//
```

This is a method that checks whether the remaining portion of a given string starting from a specific index contains a double quote character or not. The method takes two parameters: the starting index and the string to be searched.

The method starts by getting the character at the specified `start` index in the given string. It then iterates over the remaining characters of the string, starting from the next index, and checks whether any of them is a double quote character.

If a double quote character is found, the method returns true, indicating that the rest of the line contains a double quote character. If no double quote character is found in the remaining portion of the string, the method returns false.

Overall, this method is used as a helper method in the larger task of checking whether a given token in a string is a valid string literal or not.

```
//-----//
// This method check if the token is a character token or not
private static boolean isChar(int start, String CurrentLine) {
    char ch = CurrentLine.charAt(start);
    if (ch != '\\') {
        return false;
    } // 'a'
    if (CurrentLine.length() - start < 3) return false; // minimum length should be 3, i.e.,
'a'
    int last = getSizeOf(start, CurrentLine) - 1; // find the last index before space or
bracket
    if (CurrentLine.charAt(start + 2) == '\\') {
        if (CurrentLine.charAt(start + 1) == '\\')
            return false; // empty character is not allowed
        if (CurrentLine.charAt(last) != '\\') return false; // ignore last if it is not '
        if ((last - start == 2) && CurrentLine.charAt(last) == '\\') { // 'a' 'b' ....
            return CurrentLine.charAt(start + 1) != '\\'; // ignore '\\'
        }
        // '\\ '
        return CurrentLine.charAt(start + 1) == '\\ ' && last - start == 3; // '\\ '
    }
    return false;
}
//-----//
```

This code defines a private static method called `isChar` that checks if a given token in a string is a character literal or not. The method takes in two parameters: `start`, which represents the starting index of the token, and `CurrentLine`, which is the string that contains the token.

The method starts by checking if the first character of the token is a single quote (`'`), which is the starting character for a character literal. If it is not a single quote, the method returns `false` indicating that the token is not a character literal.

The next check ensures that the length of the token is at least 3 characters long, as a character literal should consist of a starting single quote, a character, and an ending single quote.

The method then finds the last index of the token before a space or bracket. This index will be used later to check if the token ends with a single quote.

The next check verifies that the second character of the token is not a single quote, as empty character literals are not allowed.

The following check ensures that the token ends with a single quote character by checking if the last character of the token is a single quote. If the last character is not a single quote, the method returns `false`.

The method then checks if the token contains only one character between the starting and ending single quotes. If the length of the token is equal to 3, i.e., the token is `'a'`, the method returns `true` if the character is not preceded by a backslash (`\`) or `false` otherwise.

Finally, the method checks if the token contains an escape sequence by checking if the second character of the token is a backslash (`\`) and the length of the token is equal to 4, i.e., the token is `'\x'`. If the token contains an escape sequence, the method returns `true`. Otherwise, it returns `false`, indicating that the token is not a character literal.

```
//-----//
// The method check if the number is an unsigned hexadecimal integer
private static boolean isHexadecimalUnsignedInteger(int startIndex, String CurrentLine) {
    char ch = CurrentLine.charAt(startIndex);
    // Check the first character. If it's not 0, it's not a hexadecimal number
    if (ch != '0') {
```

```

        return false;
    }
    int remain_len = CurrentLine.length() - startIndex;
    // If the remaining length is less than 3 and the second character is not 'x', return
false;
    if (remain_len < 3 && CurrentLine.charAt(startIndex + 1) != 'x') {
        return false;
    }
    int lastIndex = CurrentLine.length() - 1;
    int digit_index = startIndex + 2;
    ch = CurrentLine.charAt(digit_index);
    // If two characters are satisfied, then check the hexadecimals
    while (!(brackets.containsKey(ch) || Character.isSpaceChar(ch))) {
        if (!isHexadecimalDigit(ch)) {
            return false;
        }
        if (lastIndex == digit_index) {
            break;
        }
        digit_index++;
        ch = CurrentLine.charAt(digit_index);
    }
    return true;
}
//-----//
-----//

```

This code is a Java method that checks if a given string represents a hexadecimal unsigned integer.

The method takes two parameters: the starting index of the string to be checked (startIndex), and the string itself (CurrentLine).

The first step is to check if the first character of the string is '0', as all hexadecimal numbers start with '0'. If it's not '0', the method returns false.

Then the method checks if the remaining length of the string is at least three characters, and if the second character is 'x'. If not, the method returns false.

Next, the method loops through the string, starting from the third character (startIndex + 2) until it finds a whitespace or a bracket character. Within the loop, it checks if each character is a hexadecimal digit (i.e., a character between '0' and '9', or between 'a' and 'f' or 'A' and 'F'). If a non-hexadecimal character is encountered, the method returns false.

Finally, if the loop completes without returning false, the method returns true, indicating that the string represents a valid hexadecimal unsigned integer.

```

//-----//
-----//
// This is a method called isBinaryUnsignedInteger, which takes a starting index and a string
as input and
// returns a boolean indicating whether or not the string starting at the given index
represents a binary unsigned integer.
private static boolean isBinaryUnsignedInteger(int startIndex, String CurrentLine) {
    char ch = CurrentLine.charAt(startIndex);
    // check the first character. If it's not 0 it's not a binary integer
    if (ch != '0') {
        return false;
    }
    int remain_len = CurrentLine.length() - startIndex;
    // if the remaining length is less than 3 and second character is not b return false;
    if (remain_len < 3 && CurrentLine.charAt(startIndex + 1) != 'b') {
        return false;
    }
    int lastIndex = CurrentLine.length() - 1;
    int digit_index = startIndex + 2;
    ch = CurrentLine.charAt(digit_index);
    // if two characters are satisfied then check the binary digits;
    while (!(brackets.containsKey(ch) || Character.isSpaceChar(ch))) {

```

```

        if (!isBinary(ch)) {
            return false;
        }
        if (lastIndex == digit_index) {
            break;
        }
        digit_index++;
        ch = CurrentLine.charAt(digit_index);
    }
    return true;
}
//-----//
-----//

```

The method `isBinaryUnsignedInteger` checks if the substring of a given string starting from a specific index represents a binary unsigned integer or not. The method first checks the character at the starting index. If it is not '0', then the string is not a binary integer, and it returns false. Then the method checks the length of the remaining substring. If it is less than 3 and the second character is not 'b', then the substring cannot represent a binary integer, and it returns false.

The method then initializes some variables and starts iterating over the substring from the third character onwards. It checks if each character is a binary digit or not. If any character is not a binary digit, then the substring cannot represent a binary integer, and it returns false. If the method reaches the end of the substring or encounters a bracket or a space character, it breaks out of the loop.

Finally, if all the characters in the substring are binary digits and it has passed all the checks, the method returns true.

```

//-----//
-----//
// This function checks if a character is a binary digit (0 or 1)
private static boolean isBinary(Character ch) {
    return ch == '0' || ch == '1';
}
//-----//
-----//

```

This is a method named "isBinary" that takes a character as input and returns a boolean value. The method checks if the given character is a binary digit, which can only be either 0 or 1. If the character is 0 or 1, the method returns true; otherwise, it returns false. This method is likely used in other methods or functions to validate if a given character represents a valid binary digit.

```

//-----//
-----//
// This is a method that checks whether a given substring is a floating point number or not.
// The method takes in two parameters: the starting index of the substring and the string
// itself.
private static boolean isFloatingPointNumber(int startIndex, String CurrentLine) {
    // check the first character;
    char ch = CurrentLine.charAt(startIndex);
    // if first character is not a digit, plus sign, minus sign, or decimal point, return
false
    if (!(ch == '+' || ch == '-' || Character.isDigit(ch) || ch == '.')) {
        return false;
    }
    int length = CurrentLine.length();
    int curr_index = startIndex;
    // if the current index is the last character in the string, return false
    if (curr_index == length - 1) return false;
    // if the first character is a plus or minus sign, move to the next character
    if (ch == '-' || ch == '+') {
        curr_index++;
    }
    char current_char = CurrentLine.charAt(curr_index);
    // while the current character is not a decimal point or an 'e' or 'E' character
    while (current_char != '.' && !(current_char == 'e' || current_char == 'E')) {
        // if the current index is the last character in the string, return false

```

```

        if (curr_index == length - 1) {
            return false;
        }
        // if the current character is not a digit, return false
        if (!Character.isDigit(current_char)) {
            return false;
        }
        curr_index++;
        current_char = CurrentLine.charAt(curr_index);
    }

    // if current char is .
    if (current_char == '.') {
        // find the index of the decimal point
        curr_index = CurrentLine.indexOf(current_char);
        curr_index++;
        // if the index of the decimal point is the last character in the string, return
false
        if (curr_index == length) return false;
        current_char = CurrentLine.charAt(curr_index);
        // if the character following the decimal point is not a digit, return false
        if (!Character.isDigit(current_char)) return false; // .dan sonra e veya decimal
gelmesi lazım

        // while the current character is not an 'e' or 'E' character
        while (!(current_char == 'e' || current_char == 'E')) {
            // if the index of the current character plus one is the last character in the
string
            if (curr_index + 1 == length) {
                // if the current character is a bracket or a space character or a digit,
return true
                if (brackets.containsKey(current_char) ||
Character.isSpaceChar(current_char)) {
                    return true;
                }
                if (Character.isDigit(current_char)) {
                    return true;
                }
            }
            // if the current character is not a digit, return false
            if (!Character.isDigit(current_char)) {
                return false;
            }
            curr_index++;
            current_char = CurrentLine.charAt(curr_index);
        }
        // if the index of the current character plus one is the last character in the
string, return false
        if (curr_index + 1 == length) return false;
        // check if the exponential part of the floating point number is satisfied
        return IsExponentialPartSatisfied(CurrentLine, length, curr_index, current_char);
    }
    // if the index of the current character plus one is the last character in the string,
return false
    if (curr_index + 1 == length) return false;
    // check if the exponential part of the floating point number is satisfied
    return IsExponentialPartSatisfied(CurrentLine, length, curr_index, current_char);
}
//-----
-----//

```

This is a method called `isFloatingPointNumber` that checks whether a given substring is a floating-point number or not. It takes in two parameters: the starting index of the substring and the string itself.

The method first checks the first character of the substring to see if it is a valid starting character for a floating-point number. If it is not a digit, plus sign, minus sign, or decimal point, then the method returns false.

If the first character is a plus or minus sign, the method moves to the next character.



The method then loops through the rest of the substring, checking each character to see if it is a digit or a decimal point. If it encounters any other character, it returns false. If it reaches the end of the substring without encountering an 'e' or 'E' character, then the method returns true.

If it encounters a decimal point, the method finds the index of the decimal point and checks the character immediately following it to see if it is a digit. If it is not a digit, the method returns false. It then loops through the rest of the substring, checking each character to see if it is a digit or an 'e' or 'E' character. If it encounters any other character, it returns false. If it reaches the end of the substring without encountering an 'e' or 'E' character, then the method returns true.

If it encounters an 'e' or 'E' character, the method checks the character immediately following it to see if it is a plus or minus sign. If it is, the method moves to the next character. The method then loops through the rest of the substring, checking each character to see if it is a digit. If it encounters any other character, it returns false. If it reaches the end of the substring without encountering a non-digit character, then the method returns true.

```
//-----//
// This method checks if the exponential part of a number is valid
private static boolean IsExponentialPartSatisfied(String CurrentLine, int remain_len, int
curr_index, char current_char) {
    int len = CurrentLine.length();

    // Get the next character
    current_char = CurrentLine.charAt(++curr_index);

    // Check if there's a sign (+/-) in the exponential part
    if (current_char == '+' || current_char == '-') {
        // Check if there are enough characters left to form a valid exponential part
        if (curr_index + 1 == len) return false;

        // Get the next character after the sign
        current_char = CurrentLine.charAt(++curr_index);

        // Loop through the remaining characters in the exponential part
        while (!brackets.containsKey(current_char) && !Character.isSpaceChar(current_char)) {
            // Check if the character is a digit
            if (!Character.isDigit(current_char)) {
                return false;
            }

            // Check if we have reached the end of the exponential part
            if (curr_index == len - 1) {
                break;
            }

            // Get the next character
            curr_index++;
            current_char = CurrentLine.charAt(curr_index);
        }

    } else {
        // If there's no sign, the first character in the exponential part must be a digit
        if (!Character.isDigit(current_char)) {
            return false;
        }

    }

    // If we reach here, the exponential part is valid
    return true;
}
//-----//
```

This is a method that checks whether the exponential part of a floating-point number is valid or not. It takes in four parameters: the string containing the floating-point number, the remaining length of the string, the current index of the character being processed, and the current character being processed.

The method first gets the next character after the current character, which should be the first character of the exponential part. It then checks if there is a sign (+/-) in the exponential part. If there is a sign, it checks if there are enough characters left to form a valid exponential part. If there are not enough characters, the method returns false. If there are enough characters, it gets the next character after the sign.

The method then loops through the remaining characters in the exponential part, checking if each character is a digit. If a non-digit character is encountered, the method returns false. If the end of the exponential part is reached before encountering a non-digit character, the loop breaks.

If there is no sign in the exponential part, the method checks if the first character is a digit. If it is not a digit, the method returns false.

Finally, if none of the above conditions return false, the method returns true, indicating that the exponential part is valid.

```
//-----  
-----//  
// This method checks if a given startIndex in the CurrentLine string is a valid identifier  
private static boolean isIdentifier(int startIndex, String CurrentLine) {  
  
    char ch = CurrentLine.charAt(startIndex);  
  
    // If the character is an arithmetic sign, check if it is followed by a space or a  
bracket  
    if (arithmeticSign.contains(ch)) {  
        if (startIndex == CurrentLine.length() - 1) {  
            return true;  
        }  
  
        char nextChar = CurrentLine.charAt(startIndex + 1);  
        if (Character.isSpaceChar(nextChar) || isBracket(startIndex, CurrentLine)) {  
            return true;  
        }  
    }  
  
    // If the character is a sign or alphabetic, check the rest of the characters in the  
identifier  
    if (signs.contains(ch) || Character.isAlphabetic(ch)) {  
  
        int lastIndex = CurrentLine.length() - 1;  
        if (lastIndex == startIndex) {  
            return true;  
        }  
  
        int iter = 1;  
        char nextChar = CurrentLine.charAt(startIndex + iter);  
  
        // Loop through the remaining characters in the identifier  
        while ((!Character.isSpaceChar(nextChar) && !isBracket(startIndex + iter,  
CurrentLine))) {  
            // Check if the character is valid (alphabetic, numeric, or arithmetic sign)  
            if ((!Character.isAlphabetic(nextChar) && !Character.isDigit(nextChar)  
                && !arithmeticSign.contains(nextChar)) || signs.contains(nextChar)) {  
                return false;  
            }  
            // example : if there are a sign elements except in first element, return false.  
ex: =aa?++  
            //if(signs.contains(nextChar)) return false;  
  
            // Check if we have reached the end of the identifier  
            if (lastIndex == startIndex + iter) {  
                break;  
            }  
        }  
    }  
}
```

```

    }

    iter++;
    nextChar = CurrentLine.charAt(startIndex + iter);

    }

    return true;
}
return false;
}
//-----//
-----//

```

This is a Java method that checks if a given `startIndex` in the `CurrentLine` string is a valid identifier. An identifier is a name given to a variable, function, or class in programming.

The method first checks the character at the given `startIndex`. If it is an arithmetic sign, it checks if it is followed by a space or a bracket. If it is not followed by a space or a bracket, the method returns false as it is not a valid identifier.

If the character is a sign or an alphabetic character, the method continues to check the rest of the characters in the identifier. It initializes the `lastIndex` variable to be the last index in the `CurrentLine` string. If the given `startIndex` is the same as `lastIndex`, then the method returns true as it is a valid identifier.

If the given `startIndex` is not the same as `lastIndex`, the method loops through the remaining characters in the identifier. For each character, it checks if it is valid (alphabetic, numeric, or an arithmetic sign) and if it is not a sign. If the character is not valid, the method returns false. If the method reaches the end of the identifier without encountering any invalid characters, it returns true.

The method also calls another method, `isBracket()`, which checks if a given index in the `CurrentLine` string is a bracket. The method returns true if the given index is a bracket and false otherwise.

Overall, this method checks if a given `startIndex` in the `CurrentLine` string is a valid identifier by checking each character in the identifier and returning false if any character is invalid.

---

```

//-----//
-----//
// This method checks if a given startIndex in the CurrentLine string is a valid boolean
value
private static boolean isBoolean(int startIndex, String CurrentLine) {

    char ch = CurrentLine.charAt(startIndex);
    int lastIndex = CurrentLine.length() - 1;
    int iter = 0;
    StringBuilder str = new StringBuilder();

    // Loop through the characters starting at the startIndex until a bracket or space is
    encountered
    while (!isBracket(startIndex, CurrentLine) && !Character.isSpaceChar(ch)) {
        // Check if we have reached the end of the string
        if (lastIndex == startIndex + iter) {
            str.append(ch);
            break;
        }
        str.append(ch);
        iter++;
        ch = CurrentLine.charAt(startIndex + iter);
    }

    // Check if the characters between startIndex and the bracket/space form a valid boolean
    return booleans.contains(str.toString());
}
//-----//
-----//

```

This code snippet contains a method named "isBoolean" that checks if a given startIndex in the "CurrentLine" string is a valid boolean value.

The method first gets the character at the startIndex in the CurrentLine string and initializes some variables like lastIndex, iter, and a StringBuilder object named "str".

Then, the method loops through the characters starting at the startIndex until a bracket or space is encountered. During the loop, the characters are appended to the StringBuilder object "str". If the method reaches the end of the string, the loop is broken.

Finally, the method checks if the characters between startIndex and the bracket/space form a valid boolean value. The boolean values are stored in the "booleans" variable which is a HashSet of strings. If the value in the StringBuilder object "str" is found in the "booleans" set, the method returns true. Otherwise, it returns false.

```
//-----  
//  
// This method checks if a given startIndex in the CurrentLine string contains a valid  
keyword  
private static boolean isHasKeyword(int startIndex, String CurrentLine) {  
    char ch = CurrentLine.charAt(startIndex);  
    int lastIndex = CurrentLine.length() - 1;  
    int iter = 0;  
    StringBuilder str = new StringBuilder();  
  
    // Loop through the characters starting at the startIndex until a bracket or space is  
    encountered  
    while (!isBracket(startIndex, CurrentLine) && !Character.isSpaceChar(ch)) {  
        if (lastIndex == startIndex + iter) {  
            str.append(ch);  
            break;  
        }  
        str.append(ch);  
        iter++;  
        ch = CurrentLine.charAt(startIndex + iter);  
    }  
  
    // Check if the characters between startIndex and the bracket/space form a valid keyword  
    return keywords.contains(str.toString());  
}  
//-----  
//
```

This is a method in a Java program that checks if a given startIndex in the CurrentLine string contains a valid keyword.

The method takes two arguments, an integer startIndex and a String CurrentLine. The integer startIndex represents the index in the CurrentLine string to start checking for a keyword. The String CurrentLine contains the text that needs to be checked for the presence of a keyword.

The method first initializes a char variable ch to the character at the startIndex in the CurrentLine string. It then initializes two integer variables lastIndex and iter to the last index of the CurrentLine string and 0 respectively. Finally, it initializes a StringBuilder object named str to an empty string.

The method then enters a loop that continues until a bracket or space is encountered, using the isBracket() method to check if a bracket is encountered. Inside the loop, the character ch is appended to the StringBuilder object str, the iter variable is incremented by 1, and the ch variable is set to the character at the startIndex plus iter position in the CurrentLine string.

When the loop finishes, the method checks if the characters between startIndex and the bracket/space form a valid keyword by checking if the string representation of str is contained in the Set of keywords using the contains() method. If the string is a valid keyword, the method returns true, otherwise it returns false.

Overall, this method allows a Java program to identify keywords in a text string by checking if the characters starting from a given startIndex match any of the predefined keywords.

```

//-----//
// This method checks if the character at a given index in the CurrentLine string is a
bracket
private static boolean isBracket(int index, String CurrentLine) {
    char ch = CurrentLine.charAt(index);
    // Check if the character is a key in the brackets map, indicating that it is a bracket
    return brackets.containsKey(ch);
}
//-----//

```

This is a method that checks whether the character at a specified index in the CurrentLine string is a bracket or not. The method takes two parameters: index which is the position of the character in the string to be checked, and CurrentLine which is the string to be checked.

The method first retrieves the character at the given index in the CurrentLine string and then checks if this character is a key in the brackets map. The brackets map contains keys for all the valid bracket characters, such as opening and closing parentheses, braces, and brackets.

If the character at the given index is a bracket and is found in the brackets map, the method returns true. Otherwise, it returns false.

Overall, this method is used to identify if a given character in a string is a bracket or not, which is useful for various other methods in the code that need to check for bracket usage.

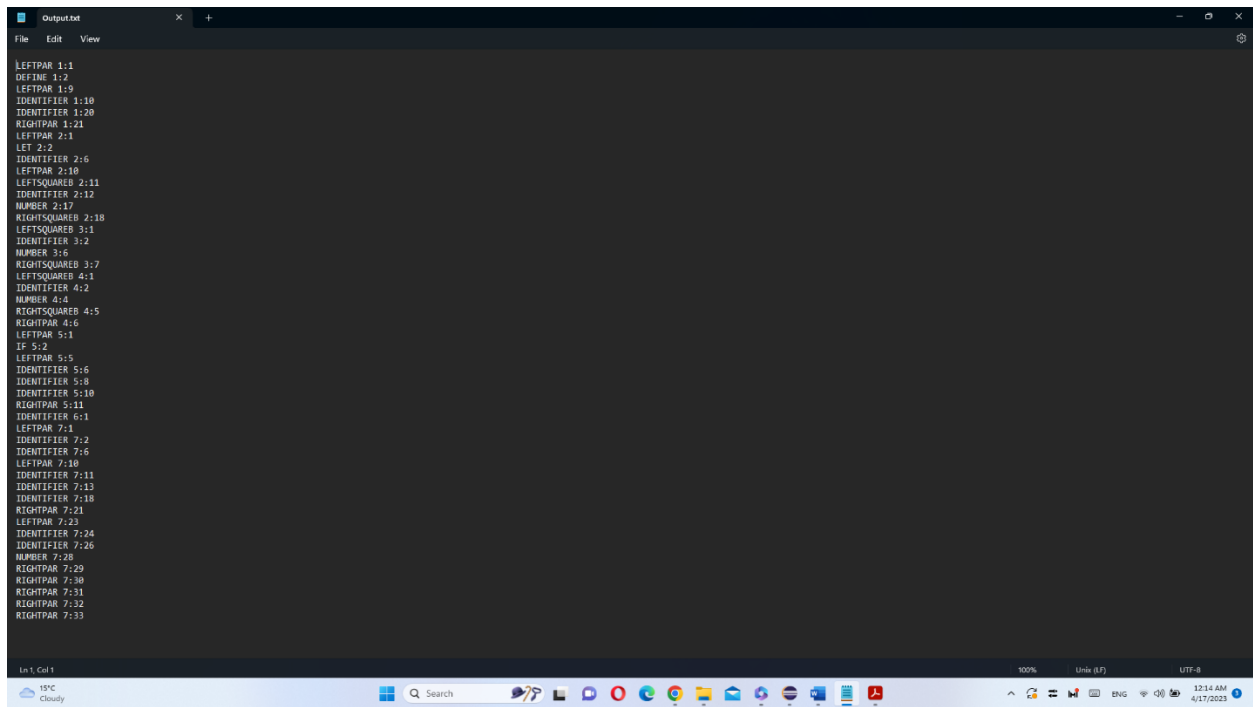
## Section (4): Test Cases

### Input File (1): Input\_1.txt

```

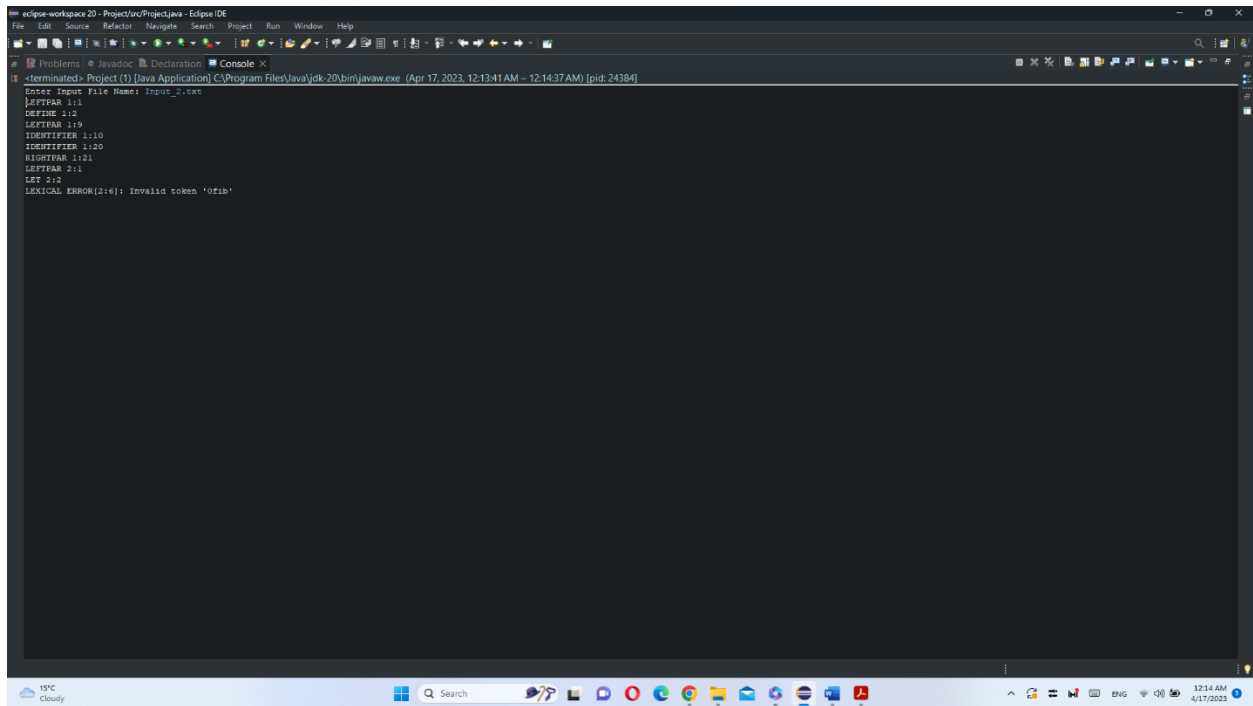
Enter Input File Name: Input_1.txt
LEFTPAR 1:1
IDENTIFIER 1:12
RIGHTPAR 1:21
LEFTPAR 1:19
IDENTIFIER 1:110
IDENTIFIER 1:120
RIGHTPAR 1:21
LEFTPAR 2:1
LET 2:2
IDENTIFIER 2:6
LEFTPAR 2:10
LEFTSQUARE 2:11
IDENTIFIER 2:12
NUMBER 2:17
RIGHTSQUARE 2:18
LEFTSQUARE 3:1
IDENTIFIER 3:2
NUMBER 3:6
RIGHTSQUARE 3:7
LEFTSQUARE 4:1
IDENTIFIER 4:2
NUMBER 4:4
RIGHTSQUARE 4:5
RIGHTPAR 4:6
LEFTPAR 5:1
IF 5:2
LEFTPAR 5:5
IDENTIFIER 5:6
IDENTIFIER 5:8
IDENTIFIER 5:10
RIGHTPAR 5:11
IDENTIFIER 6:1
LEFTPAR 7:1
IDENTIFIER 7:2
IDENTIFIER 7:4
LEFTPAR 7:10
IDENTIFIER 7:11
IDENTIFIER 7:13
IDENTIFIER 7:15
RIGHTPAR 7:21
LEFTPAR 7:23
IDENTIFIER 7:24
IDENTIFIER 7:26
NUMBER 7:28
RIGHTPAR 7:29
RIGHTPAR 7:30
RIGHTPAR 7:31
RIGHTPAR 7:32
RIGHTPAR 7:33

```

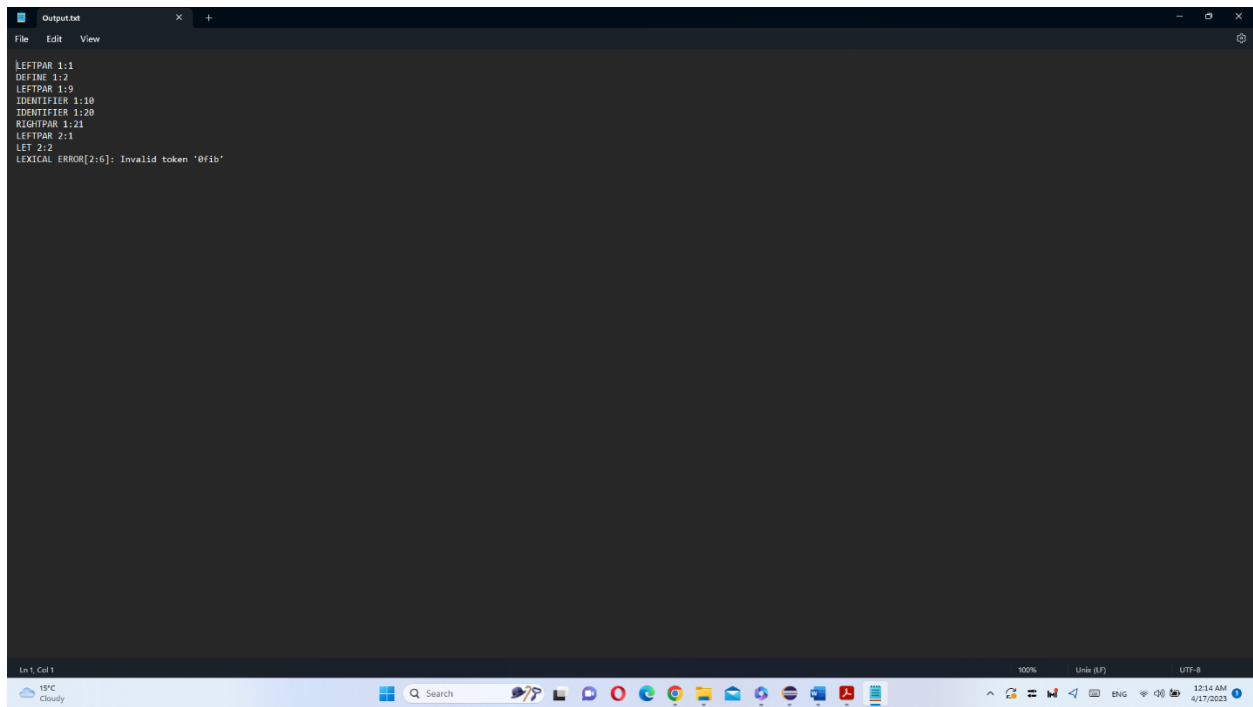


```
LEFTPAR 1:1
DEFINE 1:2
LEFTPAR 1:9
IDENTIFIER 1:10
IDENTIFIER 1:20
RIGHTPAR 1:21
LEFTPAR 2:1
LET 2:2
IDENTIFIER 2:6
LEFTPAR 2:10
LEFTSQUARE 2:11
IDENTIFIER 2:12
NUMBER 2:17
RIGHTSQUARE 2:18
LEFTSQUARE 3:1
IDENTIFIER 3:2
NUMBER 3:6
RIGHTSQUARE 3:7
LEFTSQUARE 4:1
IDENTIFIER 4:2
NUMBER 4:4
RIGHTSQUARE 4:5
RIGHTPAR 4:6
LEFTPAR 5:1
IF 5:2
LEFTPAR 5:5
IDENTIFIER 5:6
IDENTIFIER 5:8
IDENTIFIER 5:10
RIGHTPAR 5:11
IDENTIFIER 6:1
LEFTPAR 7:1
IDENTIFIER 7:2
IDENTIFIER 7:6
LEFTPAR 7:10
IDENTIFIER 7:11
IDENTIFIER 7:13
IDENTIFIER 7:18
RIGHTPAR 7:21
LEFTPAR 7:23
IDENTIFIER 7:24
IDENTIFIER 7:26
NUMBER 7:28
RIGHTPAR 7:29
RIGHTPAR 7:30
RIGHTPAR 7:31
RIGHTPAR 7:32
RIGHTPAR 7:33
```

## Input File (2): Input\_2.txt



```
<terminated> Project (1) [Java Application] C:\Program Files\Java\jdk-20\bin\javaw.exe (Apr 17, 2023, 12:13:41 AM - 12:14:37 AM) [pid: 24384]
Enter Input File Name: Input_2.txt
LEFTPAR 1:1
DEFINE 1:2
LEFTPAR 1:9
IDENTIFIER 1:10
IDENTIFIER 1:20
RIGHTPAR 1:21
LEFTPAR 2:1
LET 2:2
LEXICAL ERROR(2:6): Invalid token '0Z1b'
```

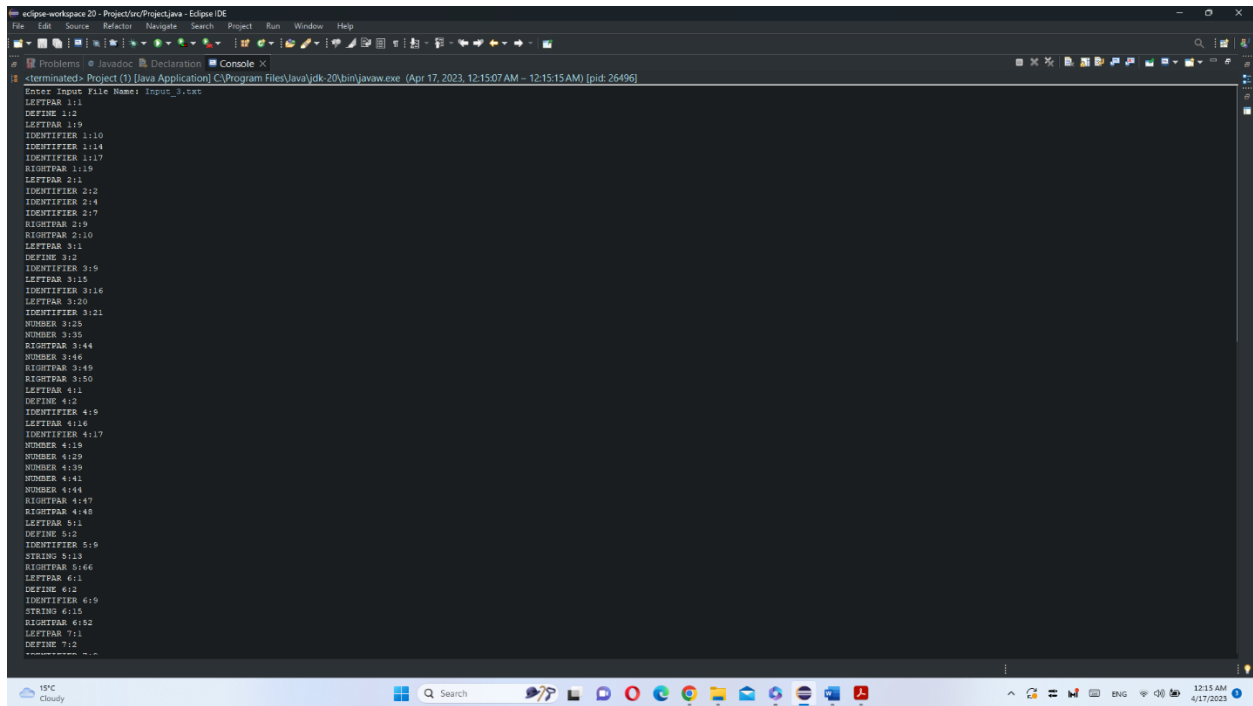


The screenshot shows the Eclipse IDE's console window. The title bar reads "Output.txt". The menu bar includes "File", "Edit", and "View". The console text is as follows:

```
LEFTPAR 1:1  
DEFINE 1:2  
LEFTPAR 1:5  
IDENTIFIER 1:10  
IDENTIFIER 1:20  
RIGHTPAR 1:21  
LEFTPAR 2:1  
LET 2:2  
LEXICAL ERROR[2:6]: Invalid token '0fib'
```

The status bar at the bottom indicates "Ln 1, Col 1", "100%", "Unicode (UTF)", and "UTF-8". The system tray shows "15°C Cloudy", a search bar, and the date/time "12:14 AM 4/17/2023".

### Input File (3): Input\_3.txt



The screenshot shows the Eclipse IDE's console window. The title bar reads "eclipse-workspace 20 - Project\src\Project.java - Eclipse IDE". The menu bar includes "File", "Edit", "Source", "Refactor", "Navigate", "Search", "Project", "Run", "Window", and "Help". The console text is as follows:

```
<terminated> Project (1) [Java Application] C:\Program Files\Java\jdk-20\bin\javaw.exe (Apr 17, 2023, 12:15:07 AM - 12:15:15 AM) [pid: 26496]  
Enter Input File Name: Input_3.txt  
LEFTPAR 1:1  
DEFINE 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  
DEFINE 3:2  
IDENTIFIER 3:5  
LEFTPAR 3:15  
IDENTIFIER 3:16  
LEFTPAR 3:20  
IDENTIFIER 3:21  
NUMBER 3:25  
NUMBER 3:35  
RIGHTPAR 3:44  
NUMBER 3:46  
RIGHTPAR 3:49  
RIGHTPAR 3:50  
LEFTPAR 4:1  
DEFINE 4:2  
IDENTIFIER 4:5  
LEFTPAR 4:14  
IDENTIFIER 4:17  
NUMBER 4:19  
NUMBER 4:29  
NUMBER 4:39  
NUMBER 4:41  
NUMBER 4:44  
RIGHTPAR 4:47  
RIGHTPAR 4:48  
LEFTPAR 5:1  
DEFINE 5:2  
IDENTIFIER 5:5  
STRING 5:13  
RIGHTPAR 5:66  
LEFTPAR 6:1  
DEFINE 6:2  
IDENTIFIER 6:5  
STRING 6:15  
RIGHTPAR 6:52  
LEFTPAR 7:1  
DEFINE 7:2  
.....
```

The status bar at the bottom indicates "15°C Cloudy", a search bar, and the date/time "12:15 AM 4/17/2023".

```
eclipse-workspace 20 - Project/nc/ProjectJava - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help
<terminated> Project (1) [Java Application] C:\Program Files\Java\jdk-20\bin\javaw.exe (Apr 17, 2023, 12:15:07 AM - 12:15:15 AM) [pid: 26496]
RIGHTPAR 6:12
LEFTPAR 7:1
DEFINE 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
LEFTPAR 8:13
IDENTIFIER 8:14
IDENTIFIER 8:16
IDENTIFIER 8:18
RIGHTPAR 8:19
LEFTPAR 8:21
IDENTIFIER 8:22
IDENTIFIER 8:24
IDENTIFIER 8:24
RIGHTPAR 8:27
RIGHTPAR 8:28
RIGHTPAR 8:29
NUMBER 9:1
NUMBER 9:3
RIGHTPAR 9:5
RIGHTPAR 9:6
LEFTPAR 10:1
DEFINE 10:2
LEFTPAR 10:9
IDENTIFIER 10:10
IDENTIFIER 10:24
IDENTIFIER 10:24
RIGHTPAR 10:27
LEFTPAR 11:3
COND 11:4
LEFTSQUARES 11:9
LEFTPAR 11:10
IDENTIFIER 11:11
IDENTIFIER 11:13
IDENTIFIER 11:15
RIGHTPAR 11:16
NUMBER 11:18
RIGHTSQUARES 11:20
LEFTSQUARES 12:9
IDENTIFIER 12:10
NUMBER 12:15
RIGHTSQUARES 12:17
RIGHTPAR 12:18
RIGHTPAR 12:19
```

```
eclipse-workspace 20 - Project/nc/ProjectJava - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help
<terminated> Project (1) [Java Application] C:\Program Files\Java\jdk-20\bin\javaw.exe (Apr 17, 2023, 12:15:07 AM - 12:15:15 AM) [pid: 26496]
IDENTIFIER 8:14
IDENTIFIER 8:16
IDENTIFIER 8:18
RIGHTPAR 8:19
LEFTPAR 8:21
IDENTIFIER 8:22
IDENTIFIER 8:24
IDENTIFIER 8:24
RIGHTPAR 8:27
RIGHTPAR 8:28
RIGHTPAR 8:29
NUMBER 9:1
NUMBER 9:3
RIGHTPAR 9:5
RIGHTPAR 9:6
LEFTPAR 10:1
DEFINE 10:2
LEFTPAR 10:9
IDENTIFIER 10:10
IDENTIFIER 10:24
IDENTIFIER 10:24
RIGHTPAR 10:27
LEFTPAR 11:3
COND 11:4
LEFTSQUARES 11:9
LEFTPAR 11:10
IDENTIFIER 11:11
IDENTIFIER 11:13
IDENTIFIER 11:15
RIGHTPAR 11:16
NUMBER 11:18
RIGHTSQUARES 11:20
LEFTSQUARES 12:9
IDENTIFIER 12:10
NUMBER 12:15
RIGHTSQUARES 12:17
RIGHTPAR 12:18
RIGHTPAR 12:19
LEFTPAR 15:1
DEFINE 15:2
IDENTIFIER 15:9
LEFTPAR 15:16
IDENTIFIER 15:17
NUMBER 15:19
NUMBER 15:29
NUMBER 15:39
NUMBER 15:41
NUMBER 15:44
NUMBER 15:48
LEXICAL ERROR[15:51]: Invalid token '9E'
```



```
Output.txt
File Edit View

LEFTPAR 1:1
DEFINE 1:2
LEFTPAR 1:5
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
DEFINE 3:2
IDENTIFIER 3:9
LEFTPAR 3:15
IDENTIFIER 3:16
LEFTPAR 3:20
IDENTIFIER 3:21
NUMBER 3:25
NUMBER 3:35
RIGHTPAR 3:44
NUMBER 3:46
RIGHTPAR 3:49
RIGHTPAR 3:50
LEFTPAR 4:1
DEFINE 4:2
IDENTIFIER 4:9
LEFTPAR 4:16
IDENTIFIER 4:17
NUMBER 4:19
NUMBER 4:20
NUMBER 4:30
NUMBER 4:41
NUMBER 4:44
RIGHTPAR 4:47
RIGHTPAR 4:48
LEFTPAR 5:1
DEFINE 5:2
IDENTIFIER 5:9
STRING 5:13
RIGHTPAR 5:66
LEFTPAR 6:1
DEFINE 6:2
IDENTIFIER 6:9
STRING 6:15
RIGHTPAR 6:52
LEFTPAR 7:1
DEFINE 7:2
IDENTIFIER 7:9
Ln 1, Col 1
```

```
Output.txt
File Edit View

IDENTIFIER 6:9
STRING 6:15
RIGHTPAR 6:52
LEFTPAR 7:1
DEFINE 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
LEFTPAR 8:13
IDENTIFIER 8:14
IDENTIFIER 8:16
IDENTIFIER 8:18
RIGHTPAR 8:19
LEFTPAR 8:21
IDENTIFIER 8:22
IDENTIFIER 8:24
IDENTIFIER 8:26
RIGHTPAR 8:27
RIGHTPAR 8:28
RIGHTPAR 8:29
NUMBER 9:1
NUMBER 9:3
RIGHTPAR 9:5
RIGHTPAR 9:6
LEFTPAR 10:1
DEFINE 10:2
LEFTPAR 10:9
IDENTIFIER 10:10
IDENTIFIER 10:24
IDENTIFIER 10:26
RIGHTPAR 10:27
LEFTPAR 11:3
COND 11:4
LEFTSQUAREB 11:9
LEFTPAR 11:10
IDENTIFIER 11:11
IDENTIFIER 11:13
IDENTIFIER 11:15
RIGHTPAR 11:16
NUMBER 11:18
RIGHTSQUAREB 11:20
LEFTSQUAREB 12:9
IDENTIFIER 12:10
NUMBER 12:15
RIGHTSQUAREB 12:17
Ln 1, Col 1
```

```
Output.txt
File Edit View
LEFTPAR 8:13
IDENTIFIER 8:14
IDENTIFIER 8:16
IDENTIFIER 8:18
RIGHTPAR 8:19
LEFTPAR 8:21
IDENTIFIER 8:22
IDENTIFIER 8:24
IDENTIFIER 8:26
RIGHTPAR 8:27
RIGHTPAR 8:28
RIGHTPAR 8:29
NUMBER 9:1
NUMBER 9:3
RIGHTPAR 9:5
RIGHTPAR 9:6
LEFTPAR 10:1
DEFINE 10:2
LEFTPAR 10:9
IDENTIFIER 10:10
IDENTIFIER 10:24
IDENTIFIER 10:26
RIGHTPAR 10:27
LEFTPAR 11:3
COND 11:4
LEFTSQUAREB 11:9
LEFTPAR 11:10
IDENTIFIER 11:11
IDENTIFIER 11:13
IDENTIFIER 11:15
RIGHTPAR 11:16
NUMBER 11:18
RIGHTSQUAREB 11:20
LEFTSQUAREB 12:9
IDENTIFIER 12:10
NUMBER 12:15
RIGHTSQUAREB 12:17
RIGHTPAR 12:18
RIGHTPAR 12:19
LEFTPAR 15:1
DEFINE 15:2
IDENTIFIER 15:9
LEFTPAR 15:16
IDENTIFIER 15:17
NUMBER 15:19
NUMBER 15:29
NUMBER 15:39
NUMBER 15:41
NUMBER 15:44
NUMBER 15:48
LEXICAL ERROR[15:51]: Invalid token '8E'
```

Ln 1, Col 1 100% Unix (LF) UTF-8

15°C Cloudy Search 12:16 AM 4/17/2021