Marmara University - Faulty of Engineering

Department of Computer Engineering

CSE2260 Principles of Programming Languages (Spring 2023)

Submit Date: 25/05/2023.

Parser 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.

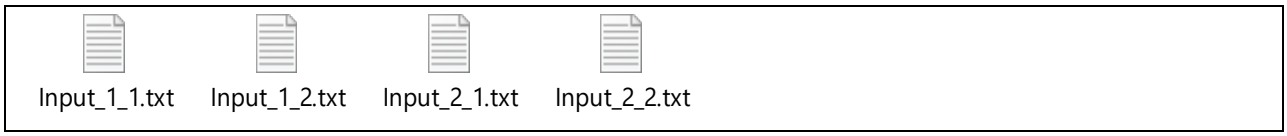
Section (2): Implementation

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

- default package: -
  - Parser Class
  - Token Class
  - SyntaxErrorException Class
  - LexicalChecker Class

- TokenType Class
- KEYWORDS Class
- GrammerRules Class

- Imported Libraries: -
  - Import java.io.*;
    - Io.file
    - Io.FileNotFoundException
    - Io.FileWriter
    - Io.IoException
  - Import java.util.*;
    - Util.ArrayList
    - Util.Scanner
- Input Files: -



- Output File: -
  - Depends on the input file!

Section (3): Explanation

Main Overview: -

A syntax analyzer, also known as a parser, is an essential component of a compiler or interpreter. Its primary task is to analyze the structure of the source code written in a programming language and ensure that it adheres to the language's grammar rules. The parser performs this analysis by breaking down the source code into a hierarchical structure known as a parse tree or an abstract syntax tree (AST).

Here is a detailed overview of the syntax analyzer (parser):

1. Input: The parser takes the source code as input. The source code is usually a sequence of characters or tokens obtained from the lexical analyzer (lexer).
2. Grammar: The parser works based on formal grammar that defines the syntax rules of the programming language. The grammar is typically specified using notations such as Backus-Naur Form (BNF) or Extended Backus-Naur Form (EBNF).
3. Tokenization: Before the parsing process begins, the parser receives a stream of tokens from the lexer. Tokens are the smallest meaningful units of the programming language, such as keywords, identifiers, operators, and literals. The parser uses these tokens to build the parse tree.
4. Parsing Techniques: There are different parsing techniques used by parsers, such as:
   - LL Parsing: LL (left-to-right, leftmost derivation) parsers read the input from left to right and construct a leftmost derivation of the parse tree. LL parsing is commonly used for programming languages with deterministic grammars, such as Pascal and Java.

- LR Parsing: LR (left-to-right, rightmost derivation) parsers read the input from left to right and construct a rightmost derivation of the parse tree. LR parsing is more powerful and can handle a larger class of grammars, including those with ambiguity. Common LR parsing algorithms include LR(0), SLR(1), LALR(1), and LR(1).
- Recursive Descent Parsing: This is a top-down parsing technique where each non-terminal in the grammar corresponds to a recursive procedure. The parser recursively calls these procedures to match the input against the grammar rules. Recursive descent parsing is intuitive and easy to implement, especially for smaller grammars.
- Parser Generators: Parser generators such as Yacc (Yet Another Compiler Compiler) or Bison can automatically generate a parser from a formal grammar specification. These tools typically use LR parsing techniques.

5. Parse Tree Construction: As the parser analyzes the source code, it builds a parse tree or an abstract syntax tree (AST). A parse tree represents the complete derivation of the source code, including all the grammar rules and intermediate steps. An AST, on the other hand, simplifies the parse tree by removing unnecessary details and focuses on the essential structure of the code.

6. Error Handling: During parsing, if the parser encounters syntax errors, it reports these errors to the user. The error handling mechanism may vary depending on the parser implementation. Some parsers attempt error recovery by synchronizing to a known point in the grammar to continue parsing, while others may stop at the first error.

7. Semantic Actions: In addition to constructing the parse tree, the parser may perform semantic actions during parsing. Semantic actions can include type checking, symbol table management, generating intermediate code, or building a more detailed representation of the code for subsequent compilation or interpretation phases.

8. Output: Once the parsing process completes successfully, the parser produces the parse tree or AST as an output. This tree-like structure is then passed to the next phase of the compiler or interpreter for further processing, such as semantic analysis, optimization, or code generation.

In summary, a syntax analyzer or parser is responsible for analyzing the structure of the source code and ensuring its adherence to the grammar rules of the programming language. It constructs a parse tree or AST, which serves as the foundation for subsequent compilation or interpretation phases. Various parsing techniques and tools exist to implement a parser, depending on the complexity and characteristics of the language's grammar.

Recursive-Descent Parser: -

A recursive-descent parser is a top-down parsing technique used to analyze the syntax of a programming language. It is based on the concept of recursive procedures, where each non-terminal symbol in the grammar corresponds to a procedure that is recursively called to match the input against the grammar rules. Here is a detailed overview of a recursive-descent parser:

- Grammar Specification: A recursive-descent parser operates based on a formal grammar that defines the syntax rules of the programming language. The grammar is typically specified using a notation such as Backus-Naur Form (BNF) or Extended Backus-Naur Form (EBNF). The grammar consists of a set of production rules that describe how different language constructs can be formed.

- Non-terminal Symbols and Procedures: The non-terminal symbols in the grammar represent language constructs such as expressions, statements, or declarations. For each non-terminal symbol, the parser defines a corresponding procedure that is responsible for recognizing and parsing that construct.

- Terminal Symbols and Lexical Analyzer: Terminal symbols in the grammar represent the basic elements of the language, such as keywords, identifiers, operators, and literals. The parser relies on a lexical analyzer (lexer) to tokenize the input source code and provide a stream of tokens to the parser.
- Recursive Procedure Calls: The parser begins parsing by calling the procedure corresponding to the start symbol of the grammar. From there, the procedure recursively calls other procedures to match the input against the grammar rules. Each procedure corresponds to a non-terminal symbol and is responsible for recognizing and parsing the constructs defined by its production rule(s).
- Predictive Parsing: In a recursive-descent parser, the choice of the next production rule to apply is determined by the current input token. This is known as predictive parsing because the parser predicts the next production rule based on the current token. It uses a look-ahead token or a set of tokens to make this prediction.
- Matching and Advancing Tokens: When a procedure is called, it checks if the current token matches the expected terminal symbol(s) defined by its production rule(s). If there is a match, the parser advances to the next token and continues parsing. If there is no match, it reports a syntax error.
- Handling Ambiguity: Recursive-descent parsers work best for unambiguous grammars, where there is a unique leftmost derivation for every input. If the grammar is ambiguous, the parser may encounter issues with multiple valid choices at a given point. In such cases, the grammar needs to be modified or disambiguation rules need to be introduced to resolve the ambiguity.
- Error Handling: When a syntax error is encountered, the parser may attempt error recovery. This can involve skipping tokens until a synchronization point is reached in the grammar, where parsing can resume. Alternatively, the parser may stop at the first error or continue parsing while reporting multiple errors.
- Semantic Actions: During parsing, the recursive-descent parser can also perform semantic actions. These actions may include type checking, building symbol tables, generating intermediate representations, or performing other tasks related to semantic analysis.
- Parse Tree or AST Construction: As the recursive-descent parser successfully matches the input against the grammar rules, it constructs a parse tree or an abstract syntax tree (AST). The parse tree represents the complete derivation of the input, including all the grammar rules and intermediate steps. The AST simplifies the parse tree by removing unnecessary details and focuses on the essential structure of the code.
- Output: The output of the recursive-descent parser is typically the parse tree or AST. This structure serves as the basis for subsequent phases of the compiler or interpreter, such as semantic analysis, optimization, or code generation.

In summary, a recursive-descent parser is a top-down parsing technique that uses recursive procedures to match the input against the grammar rules of a programming language. It operates based on a formal grammar specification and constructs a parse tree or AST as the output. The parser's recursive structure allows it to closely mirror the grammar rules, making it intuitive and easy to implement for smaller grammars.

A recursive-descent parser for a programming language called PPLL(!):

The given grammar (in the .pdf file) represents the syntax rules for a programming language. Here's a detailed overview of the grammar:

- **<Program>**: Represents a program and can be either empty (**ε**) or consists of a **<TopLevelForm>** followed by another **<Program>**.
- **<TopLevelForm>**: Represents a top-level form and is enclosed in parentheses. It can either be a **<SecondLevelForm>** or a function call.

- **<SecondLevelForm>**: Represents a second-level form, which can be a **<Definition>** or a function call enclosed in parentheses.
- **<Definition>**: Represents a definition statement and starts with the keyword **DEFINE**, followed by **<DefinitionRight>**.
- **<DefinitionRight>**: Represents the right-hand side of a definition statement. It can either be an **<IDENTIFIER>** followed by an **<Expression>**, or it can be a function definition with an **<IDENTIFIER>**, **<ArgList>**, and **<Statements>**.
- **<ArgList>**: Represents a list of function arguments. It can be empty (**ε**) or consists of one or more **<IDENTIFIER>**s.
- **<Statements>**: Represents a sequence of statements. It can be an **<Expression>** or a **<Definition>** followed by more **<Statements>**.
- **<Expressions>**: Represents a sequence of expressions. It can be empty (**ε**) or consists of an **<Expression>** followed by more **<Expressions>**.
- **<Expression>**: Represents an expression. It can be an **<IDENTIFIER>**, **<NUMBER>**, **<CHAR>**, **<BOOLEAN>**, **<STRING>**, or an expression enclosed in parentheses **<Expr>**.
- **<Expr>**: Represents a complex expression. It can be a **<LetExpression>**, **<CondExpression>**, **<IfExpression>**, **<BeginExpression>**, or a function call **<FunCall>**.
- **<FunCall>**: Represents a function call. It consists of an **<IDENTIFIER>** followed by a sequence of **<Expressions>**.
- **<LetExpression>**: Represents a let expression and starts with the keyword **LET**, followed by **<LetExpr>**.
- **<LetExpr>**: Represents the body of a let expression. It can either be a set of variable definitions **<VarDefs>** enclosed in parentheses, or it can be a function call with an **<IDENTIFIER>**, **<VarDefs>**, and **<Statements>**.
- **<VarDefs>**: Represents a list of variable definitions. It starts with an **<IDENTIFIER>** followed by an **<Expression>**, and can be followed by more **<VarDefs>** recursively.
- **<CondExpression>**: Represents a conditional expression and starts with the keyword **COND**, followed by **<CondBranches>**.
- **<CondBranches>**: Represents a sequence of conditional branches enclosed in parentheses. It starts with an **<Expression>**, followed by **<Statements>**, and can be followed by more **<CondBranches>** recursively.
- **<CondBranch>**: Represents an individual conditional branch. It can be empty (**ε**) or consists of an **<Expression>** followed by **<Statements>**.
- **<IfExpression>**: Represents an if expression and starts with the keyword **IF**, followed by two **<Expression>**s, and an optional **<EndExpression>**.
- **<EndExpression>**: Represents the optional else part of an if expression and can be empty (**ε**) or consist of an **<Expression>**.
- **<BeginExpression>**: Represents a begin expression and starts with the keyword **BEGIN**, followed by **<Statements>**.

This grammar provides the rules for the syntax of the programming language, defining how different elements can be combined to form valid programs. A recursive-descent parser can be implemented based on this grammar to analyze and parse the source code written in this language.

Parser Class: -

```
//-------------------------------------------------------------------------------------------------//
    public void parse() { // Method for parsing
        if (tokenList.size() == 0) { // Checking if tokenList is empty
            return; // Return if empty
        }
        currentToken = tokenList.get(currentIndex); // Assigning the current token from tokenList
        int level = 0; // Initializing level to 0

        program(level); // Calling the program method with level parameter
    }
//-------------------------------------------------------------------------------------------------//
```

Parse() Method: -

Here's a breakdown of the provided code:

- **parse()** Method: This method serves as the entry point for the parsing process. It first checks if the **tokenList** (presumably a list of tokens) is empty. If it is empty, the method returns and the parsing process ends. If the **tokenList** is not empty, the method proceeds with parsing.
- **currentToken**: This variable is used to keep track of the current token being processed during parsing. It is initially assigned the token at index **currentIndex** in the **tokenList**.
- **level**: This variable is used to track the current level of parsing. It is initialized to 0.
- **program(level)**: This line calls the **program** method, passing the **level** variable as a parameter. This initiates the parsing process by starting with the **program** non-terminal.
- **program(level)**: This method represents the non-terminal **program** in the grammar. It likely contains the logic to parse the structure and content of a program according to the defined grammar rules.

The provided code snippet sets up a basic framework for parsing by defining the **parse()** method as the entry point and using the **program()** method to parse the program.

```
//-------------------------------------------------------------------------------------------------//
    private void program(int level) { // Method for program
        printRule(level, GrammerRules.program); // Calling the printRule method with level and program as
parameters
        if (currentIndex <= tokenList.size() - 1 && currentToken.getType().equals(TokenType.LEFTPAR)) { //
Checking conditions
            TopLevelForm(level + 1); // Calling the TopLevelForm method with level + 1 parameter
            program(level + 1); // Recursive call to program method with level + 1 parameter
        } else {
            printEpsilon(level + 1); // Calling the printEpsilon method with level + 1 parameter
        }
    }
//-------------------------------------------------------------------------------------------------//
```

Program() Method: -

Here's a detailed breakdown of the provided code:

1. **program(int level)**: This method represents the non-terminal **program** in the grammar. It takes an integer parameter **level**, which is used for indentation and tracking the parsing depth.

2. **printRule(level, GrammerRules.program)**: This line calls the **printRule()** method, which presumably prints the current parsing rule being applied. It takes the **level** parameter for indentation and the **GrammerRules.program** as a parameter, which likely represents the rule for the **program** non-terminal.

3. **currentIndex <= tokenList.size() - 1**: This condition checks if the **currentIndex** is within the valid range of tokens in the **tokenList**.

4. **currentToken.getType().equals(TokenType.LEFTPAR)**: This condition checks if the type of the current token is equal to the **LEFTPAR** type, indicating an opening parenthesis.

5. **TopLevelForm(level + 1)**: This line calls the **TopLevelForm()** method with **level + 1** as a parameter. It represents the non-terminal **TopLevelForm** in the grammar and is called when the current token satisfies the condition in step 4.

6. **program(level + 1)**: This line represents the recursive call to the **program()** method with **level + 1** as a parameter. It allows for parsing multiple **TopLevelForm** structures in succession.

7. **printEpsilon(level + 1)**: This line calls the **printEpsilon()** method with **level + 1** as a parameter. It represents the epsilon or empty production of the **program** non-terminal. It is called when the current token does not satisfy the condition in step 4, indicating the end of parsing.

The provided code segment demonstrates the parsing logic for the **program** non-terminal. It applies the corresponding grammar rules based on the current token and makes recursive calls to handle nested structures.

```
//-------------------------------------------------------------------------------------------------//
  private void printRule(int level, String rule) { // Method for printing a rule
      for (int i = 0; i < level; i++) { // Loop for indentation
          System.out.print("     "); // Printing indentation
          output.append("     "); // Appending indentation to output
      }
      System.out.print("<"); // Printing opening angle bracket
      output.append("<"); // Appending opening angle bracket to output
      System.out.print(rule); // Printing rule
      System.out.println(">"); // Printing closing angle bracket and newline
      output.append(rule); // Appending rule to output
      output.append(">"); // Appending closing angle bracket to output
      output.append("\n"); // Appending newline to output
  }
//-------------------------------------------------------------------------------------------------//
```

printRule(): -

Here's an explanation of the provided code:

- **printRule(int level, String rule)**: This method is responsible for printing a rule during the parsing process. It takes two parameters: **level**, representing the current parsing level for indentation, and **rule**, representing the rule to be printed.

- **for (int i = 0; i < level; i++)**: This line initiates a loop that iterates **level** number of times. It is used for indentation during printing.

- **System.out.print(" ")** and **output.append(" ")**: These lines print and append four spaces for indentation. The number of spaces printed is determined by the current value of **level**.
- **System.out.print("<")** and **output.append("<")**: These lines print and append an opening angle bracket before the rule.
- **System.out.print(rule)**: This line prints the actual rule.
- **System.out.println(">")**: This line prints a closing angle bracket and a newline character, indicating the end of the rule.
- **output.append(rule)**: This line appends the rule to an **output** variable or buffer. The **output** variable is likely used to accumulate the printed rules for further processing or display.
- **output.append(">")**: This line appends a closing angle bracket to the **output** variable.
- **output.append("\n")**: This line appends a newline character to the **output** variable, indicating the end of the current rule.

The provided code segment demonstrates a simple method for printing a rule with proper indentation and formatting. It prints the rule enclosed in angle brackets and appends it to an **output** variable. The indentation level is controlled by the **level** parameter, and the resulting output can be used for various purposes, such as debugging, logging, or generating a parse tree representation.

```
  //-----------------------------------------------------------------------------------------------//
    private void TopLevelForm(int level) { // Method for TopLevelForm

        printRule(level, GrammerRules.TopLevelForm); // Calling the printRule method with level and
TopLevelForm as parameters
        consume(TokenType.LEFTPAR, level + 1); // Calling the consume method with LEFTPAR and level + 1 as
parameters
        SecondLevelForm(level + 1); // Calling the SecondLevelForm method with level + 1 parameter
        consume(TokenType.RIGHTPAR, level + 1); // Calling the consume method with RIGHTPAR and level + 1
as parameters
    }
  //-----------------------------------------------------------------------------------------------//
    private void printTerminal(String terminal, int level) { // Method for printing a terminal
        for (int i = 0; i < level; i++) { // Loop for indentation
            System.out.print("     "); // Printing indentation
            output.append("     "); // Appending indentation to output
        }
        String message = terminal + " (" + currentToken.getValue() + ")"; // Creating the terminal message
        output.append(message); // Appending message to output
        System.out.println(message); // Printing the message
        output.append("\n"); // Appending newline to output
    }
  //-----------------------------------------------------------------------------------------------//
 // Moves to the next token in the token list
    private void nextMove() {
        currentIndex++;
        if (currentIndex < tokenList.size())
```

```
            currentToken = tokenList.get(currentIndex);
    }
 //-------------------------------------------------------------------------------------------//
```

TopLevelForm(), printTerminal() & nextMove(): -

Here's a detailed explanation of the provided code segments:

- **TopLevelForm(int level)**: This method represents the non-terminal **TopLevelForm** in the grammar. It takes an integer parameter **level**, which is used for indentation and tracking the parsing depth.
- **printRule(level, GrammerRules.TopLevelForm)**: This line calls the **printRule()** method, which is responsible for printing the current rule being applied. It takes the **level** parameter for indentation and the **GrammerRules.TopLevelForm** as a parameter, which likely represents the rule for the **TopLevelForm** non-terminal.
- **consume(TokenType.LEFTPAR, level + 1)**: This line calls the **consume()** method, which is responsible for consuming the next token in the token list. It expects the token to be of type **LEFTPAR** (left parenthesis) and takes **level + 1** as a parameter for indentation.
- **SecondLevelForm(level + 1)**: This line calls the **SecondLevelForm()** method, which represents the non-terminal **SecondLevelForm** in the grammar. It is called within the **TopLevelForm** method and takes **level + 1** as a parameter for indentation.
- **consume(TokenType.RIGHTPAR, level + 1)**: This line calls the **consume()** method again, but this time it expects the token to be of type **RIGHTPAR** (right parenthesis). It also takes **level + 1** as a parameter for indentation.
- **printTerminal(String terminal, int level)**: This method is responsible for printing a terminal token. It takes two parameters: **terminal**, which represents the terminal type or name, and **level**, which is used for indentation.
- **for (int i = 0; i < level; i++)**: This line initiates a loop that iterates **level** number of times. It is used for indentation during printing.
- **System.out.print(" ")** and **output.append(" ")**: These lines print and append four spaces for indentation. The number of spaces printed is determined by the current value of **level**.
- **String message = terminal + " (" + currentToken.getValue() + ")";**: This line creates a message string by concatenating the **terminal** name and the value of the current token. It represents the terminal token being printed.
- **output.append(message)**: This line appends the message to an **output** variable or buffer. The **output** variable is likely used to accumulate the printed tokens for further processing or display.
- **System.out.println(message)**: This line prints the message, representing the terminal token, to the console.
- **output.append("\n")**: This line appends a newline character to the **output** variable, indicating the end of the current terminal token.
- **nextMove()**: This method moves to the next token in the token list. It increments the **currentIndex** and assigns the token at the updated index to the **currentToken** variable, if there are more tokens available.

The provided code segments demonstrate methods for printing rules and terminal tokens, as well as consuming tokens and moving to the next token in the token list. These methods are likely used in the parsing process to handle specific grammar rules and terminal tokens according to the defined grammar.

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

```
    // Handles the second level form in the grammar
    private void SecondLevelForm(int Level) {
        printRule(level, GrammerRules.SecondLevelForm);

        // Checks if the current token is of type "DEFINE"
        if (currentToken.getType().equals(TokenType.DEFINE)) {
            Definition(level + 1); // Calls the Definition method
        } else {
            consume(TokenType.LEFTPAR, level + 1); // Consumes a left parenthesis token
            FunCall(level + 1); // Calls the FunCall method
            consume(TokenType.RIGHTPAR, level + 1); // Consumes a right parenthesis token
        }
    }

    // Consumes the expected token type
    private void consume(String expectedType, int level) {
        if (currentToken.getType().equals(expectedType)) {
            printTerminal(expectedType, level); // Prints the terminal token
            nextMove(); // Moves to the next token
        } else {
            expectedType = getExpectedType(expectedType);
            String message = "SYNTAX ERROR[" + currentToken.getRow() + ":" + currentToken.getColumn() +
"]: "
                    + "'" + expectedType + "'" + " is expected.";
            output.append(message);
            throw new SyntaxErrorException(message); // Throws a syntax error exception
        }
    }
    //-----------------------------------------------------------------------------------------------//
 // Returns the expected type as a string for error messages
    private String getExpectedType(String expectedType) {
        switch (expectedType) {
            case TokenType.IDENTIFIER -> {
                return "identifier";
            }
            case TokenType.LEFTPAR -> {
                return "(";
            }
            case TokenType.RIGHTPAR -> {
                return ")";
            }
            case TokenType.BOOLEAN -> {
                return "boolean";
            }
            case TokenType.CHAR -> {
```

```
                return "char";
            }
            case TokenType.DEFINE -> {
                return "define";
            }
            case TokenType.NUMBER -> {
                return "number";
            }
            case TokenType.STRING -> {
                return "string";
            }
            default -> {
                return expectedType;
            }
        }
    }
    //-------------------------------------------------------------------------------------------//
```

SecondLevelForm(), Consume() & getExpectedType(): -

Here's a detailed explanation of the provided code segments:

- **SecondLevelForm(int level)**: This method handles the second-level form in the grammar. It takes an integer parameter **level**, which is used for indentation and tracking the parsing depth.
- **printRule(level, GrammerRules.SecondLevelForm)**: This line calls the **printRule()** method, which is responsible for printing the current rule being applied. It takes the **level** parameter for indentation and the **GrammerRules.SecondLevelForm** as a parameter, representing the rule for the **SecondLevelForm** non-terminal.
- **currentToken.getType().equals(TokenType.DEFINE)**: This condition checks if the type of the current token is equal to **DEFINE**. It determines whether the parsing should proceed with the **Definition** rule or the **FunCall** rule.
- **Definition(level + 1)**: This line calls the **Definition()** method when the current token is of type **DEFINE**. It handles the definition rule in the grammar.
- **consume(TokenType.LEFTPAR, level + 1)**: This line calls the **consume()** method when the current token is not of type **DEFINE**. It consumes a left parenthesis token from the token list.
- **FunCall(level + 1)**: This line calls the **FunCall()** method, which handles the function call rule in the grammar. It is called when the current token is not of type **DEFINE**.
- **consume(TokenType.RIGHTPAR, level + 1)**: This line calls the **consume()** method again, which consumes a right parenthesis token from the token list. It is called after the **FunCall** rule.
- **consume(String expectedType, int level)**: This method consumes the expected token type from the token list. It takes two parameters: **expectedType**, representing the type of the token to be consumed, and **level**, representing the current parsing level for indentation.
- **printTerminal(expectedType, level)**: This line calls the **printTerminal()** method, responsible for printing the terminal token being consumed. It takes the **expectedType** and **level** as parameters.

- **nextMove()**: This method moves to the next token in the token list. It increments the **currentIndex** and assigns the token at the updated index to the **currentToken** variable.
- **getExpectedType(String expectedType)**: This method returns the expected type as a string for error messages. It takes the **expectedType** as a parameter and maps it to a human-readable string representation.
- The **switch** statement in the **getExpectedType()** method maps different token types to their corresponding expected string representations. For example, it maps **TokenType.IDENTIFIER** to "identifier", **TokenType.LEFTPAR** to "(", **TokenType.RIGHTPAR** to ")", and so on.

The provided code segments demonstrate methods for handling the **SecondLevelForm** rule, consuming tokens, printing terminal tokens, and obtaining the expected type as a string. These methods are likely used in the parsing process to enforce the grammar rules, handle syntax errors, and provide informative error messages.

```
    //-----------------------------------------------------------------------------------------------//
        // Handles the "Definition" rule in the grammar
    private void Definition(int level) {
        printRule(level, GrammerRules.Definition);
        consume(TokenType.DEFINE, level + 1); // Consumes a "DEFINE" token
        DefinitionRight(level + 1); // Calls the DefinitionRight method
    }

    // Handles the "DefinitionRight" rule in the grammar
    private void DefinitionRight(int level) {
        printRule(level, GrammerRules.DefinitionRight);

        // Checks if the current token is of type "IDENTIFIER"
        if (currentToken.getType().equals(TokenType.IDENTIFIER)) {
            nextMove(); // Moves to the next token
            expression(level + 1); // Calls the expression method
        } else {
            consume(TokenType.LEFTPAR, level + 1); // Consumes a left parenthesis token
            consume(TokenType.IDENTIFIER, level + 1); // Consumes an "IDENTIFIER" token
            ArgList(level + 1); // Calls the ArgList method
            consume(TokenType.RIGHTPAR, level + 1); // Consumes a right parenthesis token
            Statements(level + 1); // Calls the Statements method
        }
    }
    //-----------------------------------------------------------------------------------------------//
// Handles the "Statements" rule in the grammar
    private void Statements(int level) {
        printRule(level, GrammerRules.Statements);

        // Checks if the current token is one of the valid types for an expression
        if (currentToken.getType().equals(TokenType.IDENTIFIER) ||
            currentToken.getType().equals(TokenType.NUMBER) ||
            currentToken.getType().equals(TokenType.CHAR) ||
            currentToken.getType().equals(TokenType.BOOLEAN) ||
```

```
            currentToken.getType().equals(TokenType.STRING) ||
            currentToken.getType().equals(TokenType.LEFTPAR)) {

            expression(level + 1); // Calls the expression method
        } else {
            Definition(level + 1); // Calls the Definition method
            Statements(level + 1); // Calls the Statements method recursively
        }
    }
    //-------------------------------------------------------------------------------------------------//
    // Handles the "Expressions" rule in the grammar
    private void expressions(int level) {
        printRule(level, GrammerRules.Expressions);

        // Checks if the current token is one of the valid types for an expression
        if (currentToken.getType().equals(TokenType.IDENTIFIER) ||
            currentToken.getType().equals(TokenType.NUMBER) ||
            currentToken.getType().equals(TokenType.CHAR) ||
            currentToken.getType().equals(TokenType.BOOLEAN) ||
            currentToken.getType().equals(TokenType.STRING) ||
            currentToken.getType().equals(TokenType.LEFTPAR)) {

            expression(level + 1); // Calls the expression method
            expressions(level + 1); // Calls the expressions method recursively
        } else {
            printEpsilon(level + 1); // Prints epsilon (empty) production
        }
    }
    //-------------------------------------------------------------------------------------------------//
```

Definition(), DefinitionRight(), Statements() & expressions(): -

Here's a detailed explanation of the provided code segments:

- **Definition(int level)**: This method handles the "Definition" rule in the grammar. It takes an integer parameter **level**, which is used for indentation and tracking the parsing depth.
- **printRule(level, GrammerRules.Definition)**: This line calls the **printRule()** method, which is responsible for printing the current rule being applied. It takes the **level** parameter for indentation and the **GrammerRules.Definition** as a parameter, representing the rule for the "Definition" non-terminal.
- **consume(TokenType.DEFINE, level + 1)**: This line calls the **consume()** method, which consumes a "DEFINE" token from the token list. It ensures that the next token is of type "DEFINE".
- **DefinitionRight(level + 1)**: This line calls the **DefinitionRight()** method, which handles the "DefinitionRight" rule in the grammar.
- **DefinitionRight(int level)**: This method handles the "DefinitionRight" rule in the grammar. It takes an integer parameter **level**, which is used for indentation and tracking the parsing depth.

- **printRule(level, GrammerRules.DefinitionRight)**: This line calls the **printRule()** method to print the current rule being applied. It takes the **level** parameter for indentation and the **GrammerRules.DefinitionRight** as a parameter, representing the rule for the "DefinitionRight" non-terminal.
- **currentToken.getType().equals(TokenType.IDENTIFIER)**: This condition checks if the type of the current token is equal to "IDENTIFIER". It determines whether the parsing should proceed with the alternative production for the "DefinitionRight" rule.
- **nextMove()**: This method moves to the next token in the token list. It increments the **currentIndex** and assigns the token at the updated index to the **currentToken** variable.
- **expression(level + 1)**: This line calls the **expression()** method when the current token is of type "IDENTIFIER". It handles the expression rule in the grammar.
- **consume(TokenType.LEFTPAR, level + 1)**: This line consumes a left parenthesis token from the token list when the current token is not of type "IDENTIFIER".
- **consume(TokenType.IDENTIFIER, level + 1)**: This line consumes an "IDENTIFIER" token from the token list.
- **ArgList(level + 1)**: This line calls the **ArgList()** method, which handles the "ArgList" rule in the grammar.
- **consume(TokenType.RIGHTPAR, level + 1)**: This line consumes a right parenthesis token from the token list.
- **Statements(level + 1)**: This line calls the **Statements()** method, which handles the "Statements" rule in the grammar.
- **Statements(int level)**: This method handles the "Statements" rule in the grammar. It takes an integer parameter **level**, which is used for indentation and tracking the parsing depth.
- **printRule(level, GrammerRules.Statements)**: This line calls the **printRule()** method to print the current rule being applied. It takes the **level** parameter for indentation and the **GrammerRules.Statements** as a parameter, representing the rule for the "Statements" non-terminal.
- The subsequent **if** condition checks if the current token is one of the valid types for an expression (IDENTIFIER, NUMBER, CHAR, BOOLEAN, STRING, or LEFTPAR). If true, it calls the **expression()** method.
- If the current token is not one of the valid types for an expression, it calls the **Definition()** method, followed by recursively calling the **Statements()** method.
- **expressions(int level)**: This method handles the "Expressions" rule in the grammar. It takes an integer parameter **level**, which is used for indentation and tracking the parsing depth.
- **printRule(level, GrammerRules.Expressions)**: This line calls the **printRule()** method to print the current rule being applied. It takes the **level** parameter for indentation and the **GrammerRules.Expressions** as a parameter, representing the rule for the "Expressions" non-terminal.
- The subsequent **if** condition checks if the current token is one of the valid types for an expression (IDENTIFIER, NUMBER, CHAR, BOOLEAN, STRING, or LEFTPAR). If true, it calls the **expression()** method and recursively calls the **expressions()** method.
- If the current token is not one of the valid types for an expression, it calls the **printEpsilon()** method, which prints epsilon (empty) production.

The provided code segments demonstrate methods for handling the "Definition," "DefinitionRight," "Statements," and "Expressions" rules in the grammar. They involve consuming tokens, calling other rule-specific methods, printing the applied rules, and handling alternative productions based on the current token type.

```
//-------------------------------------------------------------------------------------------------------//
    // Handles the "Expression" rule in the grammar
    private void expression(int Level) {
        printRule(level, GrammerRules.Expression);

        // Checks if the current token is one of the valid types for a terminal
        if (currentToken.getType().equals(TokenType.IDENTIFIER) ||
            currentToken.getType().equals(TokenType.NUMBER) ||
            currentToken.getType().equals(TokenType.CHAR) ||
            currentToken.getType().equals(TokenType.BOOLEAN) ||
            currentToken.getType().equals(TokenType.STRING)) {

            printTerminal(currentToken.getType(), level + 1); // Prints the terminal token
            nextMove(); // Moves to the next token
        } else {
            consume(TokenType.LEFTPAR, level + 1); // Consumes a left parenthesis token
            Expr(level + 1); // Calls the Expr method
            consume(TokenType.RIGHTPAR, level + 1); // Consumes a right parenthesis token
        }
    }
    //-------------------------------------------------------------------------------------------------------//
// Handles the "Expr" rule in the grammar
    private void Expr(int Level) {
        printRule(level, GrammerRules.Expr);

        // Switch statement based on the type of the current token
        switch (currentToken.getType()) {
            case KEYWORDS.LET -> LetExpression(level + 1); // Calls the LetExpression method
            case KEYWORDS.COND -> CondExpression(level + 1); // Calls the CondExpression method
            case KEYWORDS.IF -> IfExpression(level + 1); // Calls the IfExpression method
            case KEYWORDS.BEGIN -> BeginExpression(level + 1); // Calls the BeginExpression method
            default -> FunCall(level + 1); // Calls the FunCall method
        }
    }
    //-------------------------------------------------------------------------------------------------------//
    // Handles the "IfExpression" rule in the grammar
    private void IfExpression(int Level) {
        printRule(level, GrammerRules.IfExpression);
        consume(KEYWORDS.IF, level + 1); // Consumes the "IF" keyword
        expression(level + 1); // Calls the expression method
        expression(level + 1); // Calls the expression method
        EndExpression(level + 1); // Calls the EndExpression method
```

```
    }
    //-------------------------------------------------------------------------------------------------//
    // Handles the "EndExpression" rule in the grammar
    private void EndExpression(int level) {
        printRule(level, GrammerRules.EndExpression);

        // Checks if the current token is one of the valid types for an expression
        if (currentToken.getType().equals(TokenType.IDENTIFIER) ||
            currentToken.getType().equals(TokenType.NUMBER) ||
            currentToken.getType().equals(TokenType.CHAR) ||
            currentToken.getType().equals(TokenType.BOOLEAN) ||
            currentToken.getType().equals(TokenType.STRING) ||
            currentToken.getType().equals(TokenType.LEFTPAR)) {

            expression(level + 1); // Calls the expression method
        } else {
            printEpsilon(level + 1); // Prints epsilon (empty) production
        }
    }
    //-------------------------------------------------------------------------------------------------//
```

expression(), Expr(), IfExpression() & EndExpression(): -

Here's a detailed explanation of the provided code segments:

- **expression(int level)**: This method handles the "Expression" rule in the grammar. It takes an integer parameter **level**, which is used for indentation and tracking the parsing depth.
- **printRule(level, GrammerRules.Expression)**: This line calls the **printRule()** method to print the current rule being applied. It takes the **level** parameter for indentation and the **GrammerRules.Expression** as a parameter, representing the rule for the "Expression" non-terminal.
- The subsequent **if** condition checks if the current token is one of the valid types for a terminal (IDENTIFIER, NUMBER, CHAR, BOOLEAN, or STRING). If true, it means that the current token represents a terminal, and it prints the terminal token using the **printTerminal()** method.
- **printTerminal(currentToken.getType(), level + 1)**: This line calls the **printTerminal()** method to print the current terminal token. It takes the type of the current token and the **level + 1** parameter for indentation.
- **nextMove()**: This method moves to the next token in the token list. It increments the **currentIndex** and assigns the token at the updated index to the **currentToken** variable.
- If the current token is not one of the valid types for a terminal, it means it is an opening parenthesis, and the parsing should proceed with the alternative production for the "Expression" rule.
- **consume(TokenType.LEFTPAR, level + 1)**: This line consumes a left parenthesis token from the token list.
- **Expr(level + 1)**: This line calls the **Expr()** method, which handles the "Expr" rule in the grammar.
- **Expr(int level)**: This method handles the "Expr" rule in the grammar. It takes an integer parameter **level**, which is used for indentation and tracking the parsing depth.

- **printRule(level, GrammerRules.Expr)**: This line calls the **printRule()** method to print the current rule being applied. It takes the **level** parameter for indentation and the **GrammerRules.Expr** as a parameter, representing the rule for the "Expr" non-terminal.
- The subsequent **switch** statement checks the type of the current token and selects the appropriate case based on the token's type.
- If the current token is of type **KEYWORDS.LET**, it calls the **LetExpression()** method.
- If the current token is of type **KEYWORDS.COND**, it calls the **CondExpression()** method.
- If the current token is of type **KEYWORDS.IF**, it calls the **IfExpression()** method.
- If the current token is of type **KEYWORDS.BEGIN**, it calls the **BeginExpression()** method.
- If none of the above cases match, it means the current token is not a keyword, and it calls the **FunCall()** method.
- **IfExpression(int level)**: This method handles the "IfExpression" rule in the grammar. It takes an integer parameter **level**, which is used for indentation and tracking the parsing depth.
- **printRule(level, GrammerRules.IfExpression)**: This line calls the **printRule()** method to print the current rule being applied. It takes the **level** parameter for indentation and the **GrammerRules.IfExpression** as a parameter, representing the rule for the "IfExpression" non-terminal.
- **consume(KEYWORDS.IF, level + 1)**: This line consumes the "IF" keyword token from the token list.
- The subsequent two calls to the **expression()** method handle the expressions that appear after the "IF" keyword in the IfExpression production.
- **EndExpression(int level)**: This method handles the "EndExpression" rule in the grammar. It takes an integer parameter **level**, which is used for indentation and tracking the parsing depth.
- **printRule(level, GrammerRules.EndExpression)**: This line calls the **printRule()** method to print the current rule being applied. It takes the **level** parameter for indentation and the **GrammerRules.EndExpression** as a parameter, representing the rule for the "EndExpression" non-terminal.
- The subsequent **if** condition checks if the current token is one of the valid types for an expression (IDENTIFIER, NUMBER, CHAR, BOOLEAN, STRING, or LEFTPAR). If true, it calls the **expression()** method.
- If the current token is not one of the valid types for an expression, it means the production should be empty (epsilon), and it calls the **printEpsilon()** method, which prints epsilon (empty) production.

The provided code segments demonstrate methods for handling the "Expression," "Expr," "IfExpression," and "EndExpression" rules in the grammar. They involve checking token types, consuming tokens, calling other rule-specific methods, printing the applied rules, and handling alternative productions based on the current token type.

```
//-----------------------------------------------------------------------------------------------//
 // Handles the "BeginExpression" rule in the grammar
 private void BeginExpression(int level) {
     printRule(level, GrammerRules.BeginExpression);
     consume(KEYWORDS.BEGIN, level + 1); // Consumes the "BEGIN" keyword
     printTerminal(KEYWORDS.BEGIN, level + 1); // Prints the "BEGIN" keyword
     Statements(level + 1); // Calls the Statements method
```

```
    }
    //-------------------------------------------------------------------------------------------------//
// Handles the "CondExpression" rule in the grammar
    private void CondExpression(int level) {
        printRule(level, GrammerRules.CondExpression);
        consume(KEYWORDS.COND, level + 1); // Consumes the "COND" keyword
        CondBranches(level + 1); // Calls the CondBranches method
    }
    //-------------------------------------------------------------------------------------------------//
    // Handles the "CondBranches" rule in the grammar
    private void CondBranches(int level) {
        printRule(level, GrammerRules.CondBranches);
        consume(TokenType.LEFTPAR, level + 1); // Consumes a left parenthesis token
        expression(level + 1); // Calls the expression method
        Statements(level + 1); // Calls the Statements method
        consume(TokenType.RIGHTPAR, level + 1); // Consumes a right parenthesis token
        CondBranch(level + 1); // Calls the CondBranch method
    }
    //-------------------------------------------------------------------------------------------------//
    // Handles the "CondBranch" rule in the grammar
    private void CondBranch(int level) {
        printRule(level, GrammerRules.CondBranch);

        // Checks if the current token is a left parenthesis token
        if (currentToken.getType().equals(TokenType.LEFTPAR)) {
            printTerminal(TokenType.LEFTPAR, level + 1); // Prints the left parenthesis token
            nextMove(); // Moves to the next token
            expression(level + 1); // Calls the expression method
            Statements(level + 1); // Calls the Statements method
            consume(TokenType.RIGHTPAR, level + 1); // Consumes a right parenthesis token
        }
    }
    //-------------------------------------------------------------------------------------------------//
```

BeginExpression(), CondExpression(), CondBranches() & CondBranch(): -

The given code snippet appears to be a part of a program that handles parsing and processing of expressions based on a specific grammar. Here's each method in detail:

1. **BeginExpression(int level)**:

   - This method corresponds to the "BeginExpression" rule in the grammar.
   - It takes an integer parameter **level** which represents the indentation level for printing.
   - It first prints the rule being processed using the **printRule** function.
   - The **consume** function is called with the argument **KEYWORDS.BEGIN** to consume the "BEGIN" keyword token.

- The **printTerminal** function is called to print the consumed "BEGIN" keyword.
- The **Statements** method is called, which likely handles parsing and processing of statements within the expression.

2. **CondExpression(int level)**:

- This method corresponds to the "CondExpression" rule in the grammar.
- It takes an integer parameter **level** for printing indentation.
- It prints the rule being processed using **printRule**.
- The **consume** function is called with the argument **KEYWORDS.COND** to consume the "COND" keyword token.
- The **CondBranches** method is called, which is responsible for handling conditional branches within the expression.

3. **CondBranches(int level)**:

- This method corresponds to the "CondBranches" rule in the grammar.
- It takes an integer parameter **level** for indentation.
- It prints the rule being processed using **printRule**.
- The **consume** function is called with the argument **TokenType.LEFTPAR** to consume a left parenthesis token.
- The **expression** method is called, which likely handles parsing and processing of an expression.
- The **Statements** method is called, responsible for processing statements within the conditional branch.
- The **consume** function is called with the argument **TokenType.RIGHTPAR** to consume a right parenthesis token.
- The **CondBranch** method is called, which handles a single conditional branch.

4. **CondBranch(int level)**:

- This method corresponds to the "CondBranch" rule in the grammar.
- It takes an integer parameter **level** for indentation.
- It prints the rule being processed using **printRule**.
- It checks if the current token is a left parenthesis token using the **currentToken.getType()** function.
- If the current token is a left parenthesis, it prints the left parenthesis token using **printTerminal**.
- It then moves to the next token using **nextMove**.
- The **expression** method is called, handling parsing and processing of an expression.
- The **Statements** method is called, responsible for processing statements within the conditional branch.
- The **consume** function is called with the argument **TokenType.RIGHTPAR** to consume a right parenthesis token.

The provided methods handle specific rules within the grammar and likely work together with other methods to perform the overall parsing and processing of expressions.

```
    //-------------------------------------------------------------------------------------------------------//
    // Handles the "LetExpression" rule in the grammar
    private void LetExpression(int level) {
        printRule(level, GrammerRules.LetExpression);
        consume(KEYWORDS.LET, level + 1); // Consumes the "LET" keyword
        LetExpr(level + 1); // Calls the LetExpr method
    }
    //-------------------------------------------------------------------------------------------------------//
    // Handles the "LetExpr" rule in the grammar
    private void LetExpr(int level) {
        printRule(level, GrammerRules.LetExpr);

        // Checks if the current token is a left parenthesis token
        if (currentToken.getType().equals(TokenType.LEFTPAR)) {
            nextMove(); // Moves to the next token
            VarDefs(level + 1); // Calls the VarDefs method
            consume(TokenType.RIGHTPAR, level + 1); // Consumes a right parenthesis token
        } else {
            consume(TokenType.IDENTIFIER, level + 1); // Consumes an "IDENTIFIER" token
            consume(TokenType.LEFTPAR, level + 1); // Consumes a left parenthesis token
            VarDefs(level + 1); // Calls the VarDefs method
            consume(TokenType.RIGHTPAR, level + 1); // Consumes a right parenthesis token
            Statements(level + 1); // Calls the Statements method
        }
    }
    //-------------------------------------------------------------------------------------------------------//
// Handles the "VarDefs" rule in the grammar
    private void VarDefs(int level) {
        printRule(level, GrammerRules.VarDefs);
        consume(TokenType.LEFTPAR, level + 1); // Consumes a left parenthesis token
        consume(TokenType.IDENTIFIER, level + 1); // Consumes an "IDENTIFIER" token
        expression(level + 1); // Calls the expression method
        consume(TokenType.RIGHTPAR, level + 1); // Consumes a right parenthesis token
        VarDef(level + 1); // Calls the VarDef method
    }
    //-------------------------------------------------------------------------------------------------------//
    // Handles the "VarDef" rule in the grammar
    private void VarDef(int level) {
        printRule(level, GrammerRules.VarDef);

        // Checks if the current token is a left parenthesis token
        if (currentToken.getType().equals(TokenType.LEFTPAR)) {
            VarDefs(level + 1); // Calls the VarDefs method
```

```
        } else {
            printEpsilon(level + 1); // Prints epsilon (empty) production
        }
    }
    //---------------------------------------------------------------------------------------------------//
    // Handles the "ArgList" rule in the grammar
    private void ArgList(int level) {
        printRule(level, GrammerRules.ArgList);

        // Checks if the current token is an "IDENTIFIER" token
        if (currentToken.getType().equals(TokenType.IDENTIFIER)) {
            printTerminal(TokenType.IDENTIFIER, level + 1); // Prints the "IDENTIFIER" token
            nextMove(); // Moves to the next token
            ArgList(level + 1); // Calls the ArgList method recursively
        } else {
            printEpsilon(level + 1); // Prints epsilon (empty) production
        }
    }
    //---------------------------------------------------------------------------------------------------//
    // Prints epsilon (empty) production
    private void printEpsilon(int level) {
        for (int j = 0; j < level; j++) {
            System.out.print("     ");
            output.append("     ");
        }
        System.out.println('_');
        output.append("_");
        output.append("\n");
    }
    //---------------------------------------------------------------------------------------------------//
    // Handles the "FunCall" rule in the grammar
    private void FunCall(int level) {
        printRule(level, GrammerRules.FunCall);
        consume(TokenType.IDENTIFIER, level + 1); // Consumes an "IDENTIFIER" token
        expressions(level + 1); // Calls the expressions method
    }
    //---------------------------------------------------------------------------------------------------//
```

LetExpression(), LetExpr(), VarDefs(),VarDef(), ArgList(), printEpslion() & FunCall(): -

Each method in detail:

1. **LetExpression(int level)**:

    - This method corresponds to the "LetExpression" rule in the grammar.
    - It takes an integer parameter **level** which represents the indentation level for printing.
    - It prints the rule being processed using the **printRule** function.

- The **consume** function is called with the argument **KEYWORDS.LET** to consume the "LET" keyword token.
- The **LetExpr** method is called, which handles the LetExpr rule.

2. **LetExpr(int level)**:

- This method corresponds to the "LetExpr" rule in the grammar.
- It takes an integer parameter **level** for indentation.
- It prints the rule being processed using **printRule**.
- It checks if the current token is a left parenthesis token using the **currentToken.getType()** function.
- If the current token is a left parenthesis, it moves to the next token using **nextMove**.
- The **VarDefs** method is called, which handles variable definitions within the LetExpr.
- The **consume** function is called with the argument **TokenType.RIGHTPAR** to consume a right parenthesis token.
- If the current token is not a left parenthesis, it assumes that an identifier token is present.
- It consumes the identifier token using **consume(TokenType.IDENTIFIER, level + 1)**.
- It then consumes a left parenthesis token using **consume(TokenType.LEFTPAR, level + 1)**.
- The **VarDefs** method is called, which handles variable definitions.
- It consumes a right parenthesis token using **consume(TokenType.RIGHTPAR, level + 1)**.
- The **Statements** method is called, responsible for processing statements within the LetExpr.

3. **VarDefs(int level)**:

- This method corresponds to the "VarDefs" rule in the grammar.
- It takes an integer parameter **level** for indentation.
- It prints the rule being processed using **printRule**.
- It consumes a left parenthesis token using **consume(TokenType.LEFTPAR, level + 1)**.
- It consumes an identifier token using **consume(TokenType.IDENTIFIER, level + 1)**.
- The **expression** method is called, handling parsing and processing of an expression.
- It consumes a right parenthesis token using **consume(TokenType.RIGHTPAR, level + 1)**.
- The **VarDef** method is called, which handles a single variable definition.

4. **VarDef(int level)**:

- This method corresponds to the "VarDef" rule in the grammar.
- It takes an integer parameter **level** for indentation.
- It prints the rule being processed using **printRule**.
- It checks if the current token is a left parenthesis token using the **currentToken.getType()** function.

- If the current token is a left parenthesis, it calls the **VarDefs** method to handle multiple variable definitions recursively.
- If the current token is not a left parenthesis, it assumes an empty production and prints epsilon (empty) production using **printEpsilon**.

5. **ArgList(int level)**:

- This method corresponds to the "ArgList" rule in the grammar.
- It takes an integer parameter **level** for indentation.
- It prints the rule being processed using **printRule**.
- It checks if the current token is an "IDENTIFIER" token using the **currentToken.getType()** function.
- If the current token is an "IDENTIFIER" token, it prints the token using **printTerminal**.
- It moves to the next token using **nextMove**.
- It calls the **ArgList** method recursively to handle multiple arguments.
- If the current token is not an "IDENTIFIER" token, it assumes an empty production and prints epsilon (empty) production using **printEpsilon**.

6. **printEpsilon(int level)**:

- This method prints an epsilon (empty) production.
- It takes an integer parameter **level** representing the indentation level for printing.
- It prints underscores (_) to represent an empty production at the specified indentation level.

7. **FunCall(int level)**:

- This method corresponds to the "FunCall" rule in the grammar.
- It takes an integer parameter **level** for indentation.
- It prints the rule being processed using **printRule**.
- It consumes an "IDENTIFIER" token using **consume(TokenType.IDENTIFIER, level + 1)**.
- The **expressions** method is called, which likely handles parsing and processing of function call arguments.

These methods are part of a larger program that handles parsing and processing of expressions based on a specific grammar. They work together to recursively process different rules of the grammar and perform the necessary actions based on the tokens encountered.

```
    //-----------------------------------------------------------------------------------------//
// The main method for the program
    public static void main(String[] args) {
        ArrayList<String> inputList = getInputList(); // Retrieves the input list
        File outputFileParser = new File("Output.txt"); // Output file for parser results
        File outputFileTokenList = new File("OutputTokenList.txt"); // Output file for token list
```

```
        ArrayList<String> tokenMessages = new ArrayList<>(); // Stores token messages
        ArrayList<Token> tokenList = new ArrayList<>(); // Stores tokens

        LexicalChecker lexicalChecker = new LexicalChecker(); // Creates a lexical checker object
        lexicalChecker.addTokens(inputList, tokenMessages, tokenList); // Performs lexical analysis

        if (lexicalChecker.isLexicalError(tokenMessages)) {
            lexicalChecker.printResults(tokenMessages, outputFileParser); // Prints lexical error messages
            return;
        }

        Parser parser;
        parser = new Parser(tokenList); // Creates a parser object with the token list

        try {
            parser.parse(); // Parses the token list
        } catch (SyntaxErrorException e){
            System.out.println(e.getMessage()); // Prints syntax error message
        }

        try {
            FileWriter writer = new FileWriter(outputFileParser); // Creates a file writer for the parser
output
            writer.write(String.valueOf(parser.output)); // Writes the parser output to the file
            writer.close();
        } catch (IOException e) {
            throw new RuntimeException(e.getMessage());
        }

        try {
            FileWriter writer_tokenList = new FileWriter(outputFileTokenList); // Creates a file writer
for the token list output

            // Print tokens list
            for (Token token : tokenList) {
                writer_tokenList.write(token + "\n"); // Writes each token to the file
            }

            writer_tokenList.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
    //-----------------------------------------------------------------------------------------------//
```

Main() Method: -

The **main** method step by step:

- **ArrayList<String> inputList = getInputList();**:
  - This line retrieves the input list. The exact implementation of **getInputList()** is not provided, but it is assumed to return an **ArrayList** of strings containing the input for the program.
- **File outputFileParser = new File("Output.txt");**:
  - This line creates a **File** object named **outputFileParser** with the filename "Output.txt". It is used to store the output of the parser.
- **File outputFileTokenList = new File("OutputTokenList.txt");**:
  - This line creates a **File** object named **outputFileTokenList** with the filename "OutputTokenList.txt". It is used to store the list of tokens.
- **ArrayList<String> tokenMessages = new ArrayList<>();**:
  - This line creates an empty **ArrayList** named **tokenMessages** to store token messages.
- **ArrayList<Token> tokenList = new ArrayList<>();**:
  - This line creates an empty **ArrayList** named **tokenList** to store tokens.
- **LexicalChecker lexicalChecker = new LexicalChecker();**:
  - This line creates a **LexicalChecker** object named **lexicalChecker**.
- **lexicalChecker.addTokens(inputList, tokenMessages, tokenList);**:
  - This line invokes the **addTokens** method of the **lexicalChecker** object to perform lexical analysis on the **inputList**.
  - The **tokenMessages** and **tokenList** parameters are passed to store the resulting token messages and tokens, respectively.
- **if (lexicalChecker.isLexicalError(tokenMessages)) { ... }**:
  - This **if** statement checks if there were any lexical errors detected during the lexical analysis.
  - The **isLexicalError** method of **lexicalChecker** is invoked with **tokenMessages** as the parameter to determine if there are any errors.
  - If there are lexical errors, the block of code inside the **if** statement is executed.
- **lexicalChecker.printResults(tokenMessages, outputFileParser);**:
  - This line invokes the **printResults** method of **lexicalChecker** to print the lexical error messages.
  - The **tokenMessages** and **outputFileParser** parameters are passed to specify the messages to be printed and the output file where the messages will be written.
- **return;**:
  - This line terminates the execution of the **main** method and exits the program.
- **Parser parser; parser = new Parser(tokenList);**:
  - These lines declare a **Parser** object named **parser** and initialize it with the **tokenList**.

- The **tokenList** contains the tokens generated during the lexical analysis.

- **try { parser.parse(); } catch (SyntaxErrorException e) { ... }**:
  - This **try-catch** block is used to handle any syntax errors that occur during parsing.
  - The **parse** method of the **parser** object is invoked to perform the parsing process.
  - If a **SyntaxErrorException** is thrown during parsing, the corresponding **catch** block is executed.
- **System.out.println(e.getMessage());**:
  - This line prints the syntax error message to the console.
- **try { FileWriter writer = new FileWriter(outputFileParser); ... }**:
  - This **try-catch** block is used to handle any errors that occur while writing the parser output to the file.
  - A **FileWriter** named **writer** is created with **outputFileParser** as the file to be written.
  - The parser output, stored in **parser.output**, is written to the file using **writer.write**.
- **try { FileWriter writer_tokenList = new FileWriter(outputFileTokenList); ... }**:
  - This **try-catch** block is used to handle any errors that occur while writing the token list to the file.
  - A **FileWriter** named **writer_tokenList** is created with **outputFileTokenList** as the file to be written.
  - Each token in the **tokenList** is written to the file using **writer_tokenList.write(token + "\n")**.

Overall, the **main** method performs the following tasks:

- Retrieves the input list.
- Performs lexical analysis on the input list using a **LexicalChecker** object.
- Checks for lexical errors and prints the error messages if any.
- Creates a **Parser** object and performs parsing on the generated tokens.
- Writes the parser output and the token list to separate output files.

Note: Some specific details and methods related to the **LexicalChecker** and **Parser** classes are not provided, so the exact behavior of those classes may vary based on their implementation.

Token Class: -

```
//-----------------------------------------------------------------------------------------------//
/** Token Class **/
//-----------------------------------------------------------------------------------------------//
class Token {
    private final String type; // Type of the token
    private final String value; // Value of the token
    private final int row; // Row position of the token
    private final int column; // Column position of the token
```

```
    public Token(String type, String value, int row, int column) { // Constructor for creating a Token
object
        this.type = type; // Assigning the type of the token
        this.value = value; // Assigning the value of the token
        this.column = column; // Assigning the column position of the token
        this.row = row; // Assigning the row position of the token
    }

    public String getType() { // Getter for retrieving the type of the token
        return type;
    }

    public String getValue() { // Getter for retrieving the value of the token
        return value;
    }

    public int getRow() { // Getter for retrieving the row position of the token
        return row;
    }

    public int getColumn() { // Getter for retrieving the column position of the token
        return column;
    }

    @Override
    public String toString() { // Method for converting the Token object to a string representation
        return "TYPE: " + type + "\n" + "VALUE: " + value + "\n" + "ROW: " + row + " COLUMN: " + column +
"\n" + "=========" + "\n";
    }
}
//-----------------------------------------------------------------------------------------------------//
```

The provided code defines a **Token** class, which represents a token in a programming language. Let's go through the details of the class:

1. Class Declaration:

- The class is declared with the name **Token**.
- It is assumed to be in its own file, separate from the code you provided.

2. Class Members:

- **private final String type**: This member variable represents the type of the token, such as "identifier," "keyword," "operator," etc.
- **private final String value**: This member variable represents the value of the token, which is the actual content or lexeme of the token.

- **private final int row**: This member variable represents the row position or line number where the token is found in the source code.
- **private final int column**: This member variable represents the column position where the token starts in the source code.

3. Constructor:

- The class defines a constructor with four parameters: **String type**, **String value**, **int row**, and **int column**.
- The constructor is used to create a **Token** object by initializing its member variables with the provided values.

4. Getter Methods:

- The class provides getter methods to access the values of the member variables.
- **public String getType()**: This method returns the type of the token.
- **public String getValue()**: This method returns the value of the token.
- **public int getRow()**: This method returns the row position of the token.
- **public int getColumn()**: This method returns the column position of the token.

5. **toString()** Method:

- The class overrides the **toString()** method inherited from the **Object** class.
- This method converts the **Token** object to a string representation.
- It returns a string that includes the type, value, row, column, and a separator line for readability.

The **Token** class is used to represent individual tokens generated during lexical analysis. Each token object contains information about its type, value, and position in the source code. It also provides methods to retrieve the token's properties and a string representation of the token.

SyntaxErrorException Class

```
//-------------------------------------------------------------------------------------------------//
/** SyntaxErrorException **/
public class SyntaxErrorException extends RuntimeException {
    public SyntaxErrorException(String message){
        super(message);
    }
}
//-------------------------------------------------------------------------------------------------//
```

The provided code defines a custom exception class named **SyntaxErrorException**. Let's go through the details of the class:

1. Class Declaration:

- The class is declared with the name **SyntaxErrorException**.

- It is assumed to be in its own file, separate from the code you provided.

2. Inheritance:

- The class extends the **RuntimeException** class, which makes it an unchecked exception.
- By extending **RuntimeException**, instances of **SyntaxErrorException** do not need to be declared in method signatures or explicitly caught, allowing for more flexible exception handling.

3. Constructor:

- The class defines a constructor with one parameter: **String message**.
- The constructor is used to create a **SyntaxErrorException** object by passing an error message as the parameter.
- The error message is passed to the superclass constructor using the **super** keyword, which initializes the exception with the provided message.

4. Usage:

- This custom exception class can be used to indicate a syntax error in a program during parsing or any other relevant operations.
- When a syntax error is encountered, a **SyntaxErrorException** object can be instantiated with an appropriate error message and thrown using the **throw** statement.
- The exception can be caught and handled in a try-catch block to provide specific error handling logic or display error messages to the user.

The **SyntaxErrorException** class serves as a way to handle and communicate syntax errors in the program. By extending **RuntimeException**, it allows for more flexible error handling without the need for explicit catch clauses. Developers can instantiate and throw this exception when a syntax error is encountered, providing relevant error information to aid in debugging or user feedback.

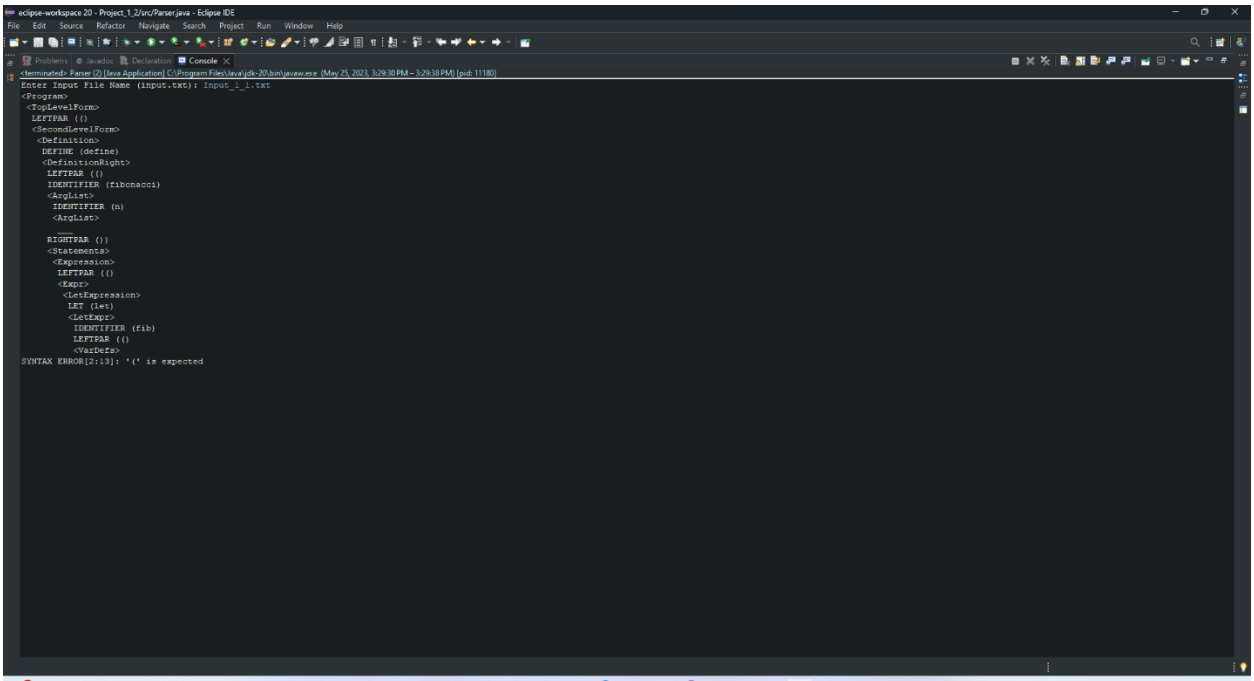
Classes: TokenType, KEYWORDS, GrammerRules

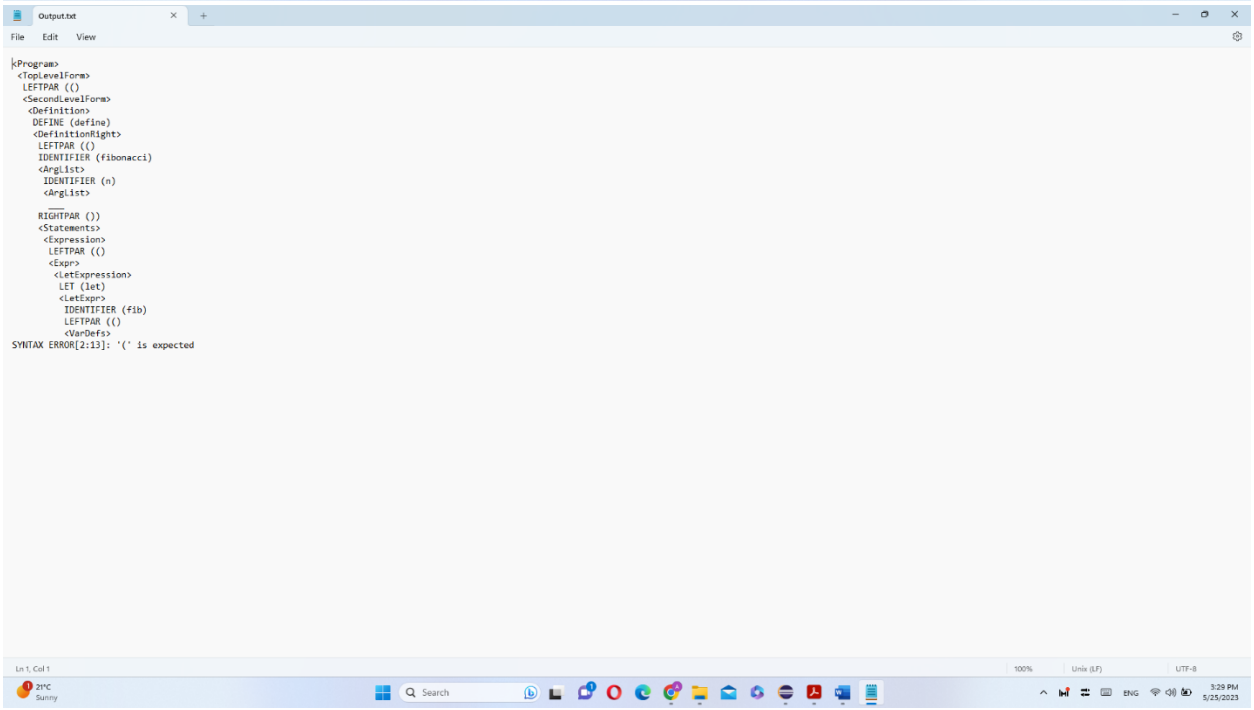
```
//-------------------------------------------------------------------------------------------------//
 // Class defining token types used in the program
    class TokenType {
        public static final String LEFTPAR = "LEFTPAR";       // Represents the left parenthesis token
        public static final String RIGHTPAR = "RIGHTPAR";     // Represents the right parenthesis token
        public static final String DEFINE = "DEFINE";         // Represents the define token
        public static final String IDENTIFIER = "IDENTIFIER"; // Represents an identifier token
        public static final String NUMBER = "NUMBER";         // Represents a number token
        public static final String CHAR = "CHAR";             // Represents a character token
        public static final String BOOLEAN = "BOOLEAN";       // Represents a boolean token
        public static final String STRING = "STRING";         // Represents a string token
    }
    //-------------------------------------------------------------------------------------------------//
    // Class defining keywords used in the program
    class KEYWORDS {
```

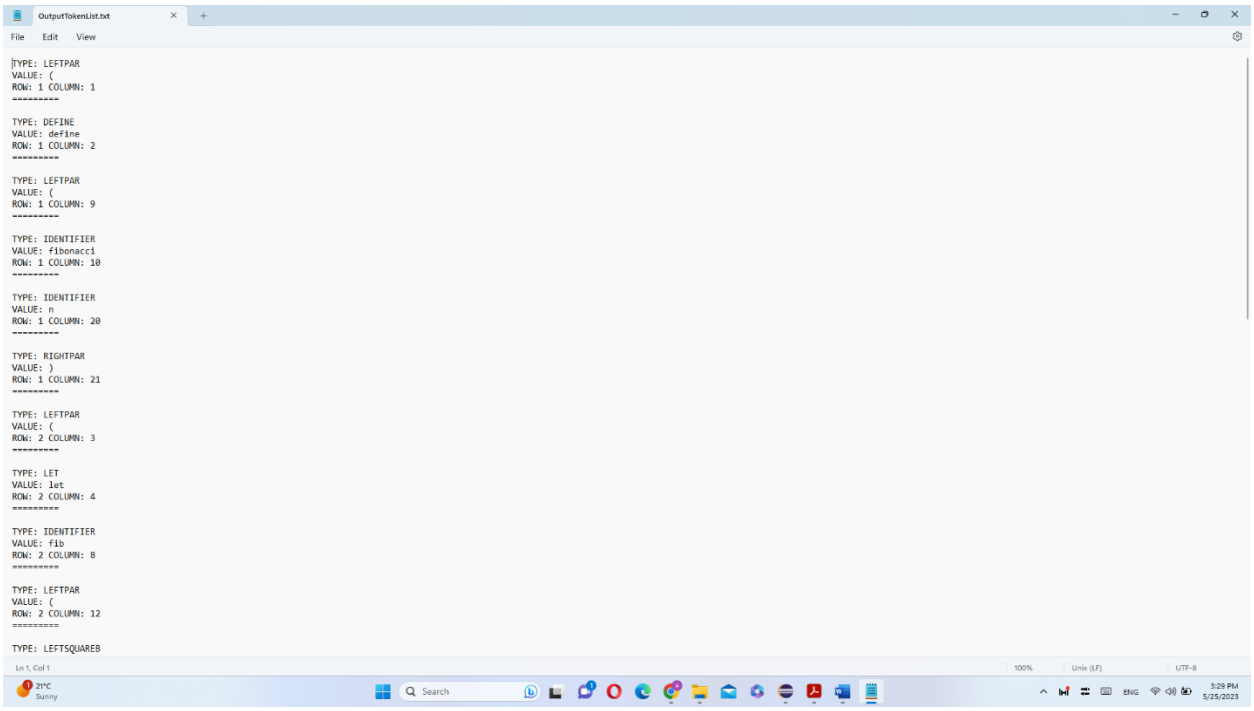
```
        public static final String IF = "IF";         // Represents the if keyword
        public static final String LET = "LET";       // Represents the let keyword
        public static final String BEGIN = "BEGIN";   // Represents the begin keyword
        public static final String COND = "COND";     // Represents the cond keyword
    }
    //-------------------------------------------------------------------------------------------//
    // Class defining grammar rules used in the program
    class GrammerRules {
        public static String program = "Program";                 // Represents the program rule
        public static String TopLevelForm = "TopLevelForm";       // Represents the top-level form rule
        public static String SecondLevelForm = "SecondLevelForm"; // Represents the second-level form rule
        public static String Definition = "Definition";           // Represents the definition rule
        public static String DefinitionRight = "DefinitionRight"; // Represents the right-hand side of a
definition rule
        public static String ArgList = "ArgList";                 // Represents the argument list rule
        public static String Statements = "Statements";           // Represents the statements rule
        public static String Expressions = "Expressions";         // Represents the expressions rule
        public static String Expression = "Expression";           // Represents the expression rule
        public static String Expr = "Expr";                       // Represents the expr rule
        public static String FunCall = "FunCall";                 // Represents the function call rule
        public static String LetExpression = "LetExpression";     // Represents the let expression rule
        public static String LetExpr = "LetExpr";                 // Represents the let expression inside
a let rule
        public static String VarDefs = "VarDefs";                 // Represents the variable definitions
rule
        public static String VarDef = "VarDef";                   // Represents a single variable
definition rule
        public static String CondExpression = "CondExpression";   // Represents the cond expression rule
        public static String CondBranches = "CondBranches";       // Represents the branches of a cond
expression rule
        public static String CondBranch = "CondBranch";           // Represents a single branch in a cond
expression rule
        public static String IfExpression = "IfExpression";       // Represents the if expression rule
        public static String EndExpression = "EndExpression";     // Represents the end expression rule
        public static String BeginExpression = "BeginExpression"; // Represents the begin expression rule
    }
    //-------------------------------------------------------------------------------------------//
```

Section (4): Test Cases

Input File (1): Input_1_1.txt


eclipse-workspace 20 - Project_1_2/src/Parser.java - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help
Problems Javadoc Declaration Console
<terminated> Parser (2) [Java Application] C:\Program Files\Java\jdk-20\bin\javaw.exe (May 25, 2023, 3:29:30 PM – 3:29:38 PM) [pid: 11180]
Enter Input File Name (input.txt): Input_1_1.txt
<Program>
<TopLevelForm>
LEFTPAR (()
<SecondLevelForm>
<Definition>
DEFINE (define)
<DefinitionRight>
LEFTPAR (()
IDENTIFIER (fibonacci)
<ArgList>
IDENTIFIER (n)
<ArgList>
RIGHTPAR ())
<Statements>
<Expression>
LEFTPAR (()
<Expr>
<LetExpression>
LET (let)
<LetExpr>
IDENTIFIER (fib)
LEFTPAR (()
<VarDefs>
SYNTAX ERROR[2:13]: '(' is expected
21°C Sunny
Search
ENG
3:29 PM 5/25/2023


Output.txt
File Edit View
<Program>
<TopLevelForm>
LEFTPAR (()
<SecondLevelForm>
<Definition>
DEFINE (define)
<DefinitionRight>
LEFTPAR (()
IDENTIFIER (fibonacci)
<ArgList>
IDENTIFIER (n)
<ArgList>
RIGHTPAR ())
<Statements>
<Expression>
LEFTPAR (()
<Expr>
<LetExpression>
LET (let)
<LetExpr>
IDENTIFIER (fib)
LEFTPAR (()
<VarDefs>
SYNTAX ERROR[2:13]: '(' is expected
Ln 1, Col 1
100%
Unix (LF)
UTF-8
21°C Sunny
Search
ENG
3:29 PM 5/25/2023


OutputTokenList.txt
File Edit View
TYPE: LEFTPAR
VALUE: (
ROW: 1 COLUMN: 1

TYPE: DEFINE
VALUE: define
ROW: 1 COLUMN: 2

TYPE: LEFTPAR
VALUE: (
ROW: 1 COLUMN: 9

TYPE: IDENTIFIER
VALUE: fibonacci
ROW: 1 COLUMN: 10

TYPE: IDENTIFIER
VALUE: n
ROW: 1 COLUMN: 20

TYPE: RIGHTPAR
VALUE:)
ROW: 1 COLUMN: 21

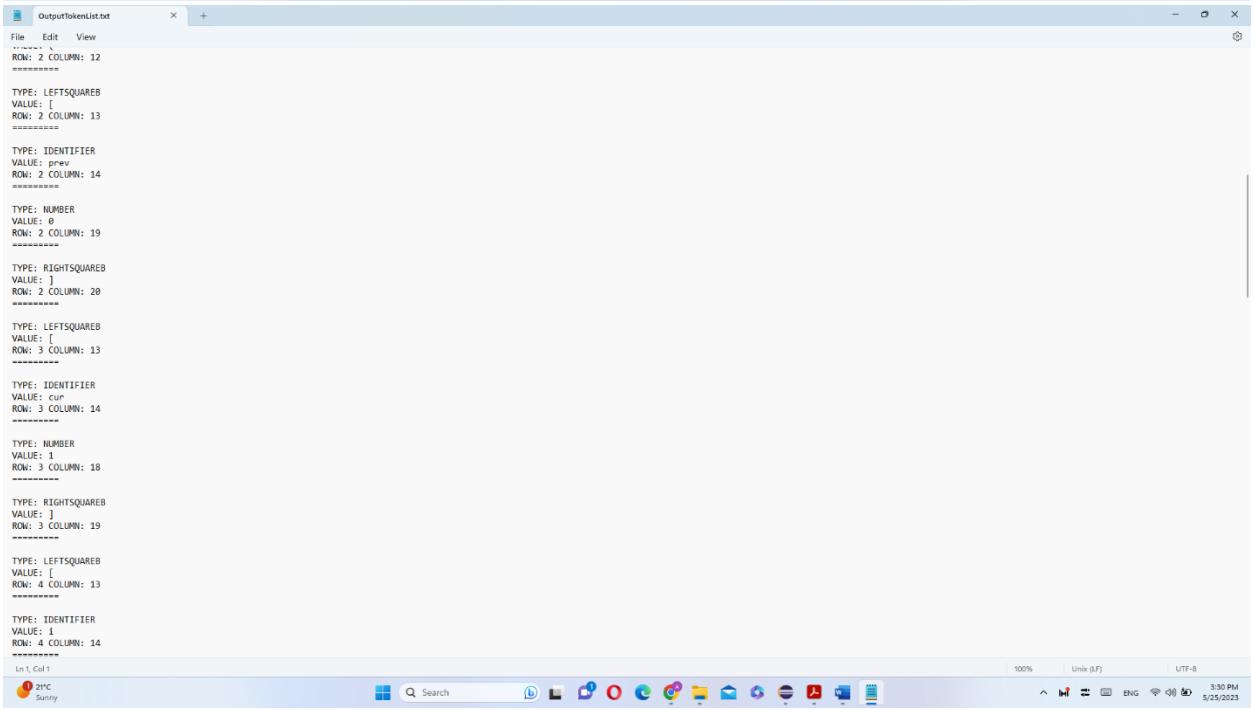
TYPE: LEFTPAR
VALUE: (
ROW: 2 COLUMN: 3

TYPE: LET
VALUE: let
ROW: 2 COLUMN: 4

TYPE: IDENTIFIER
VALUE: fib
ROW: 2 COLUMN: 8

TYPE: LEFTPAR
VALUE: (
ROW: 2 COLUMN: 12

TYPE: LEFTSQUAREB
Ln 1, Col 1
100%
Unix (LF)
UTF-8
21°C
Sunny
Search
ENG
3:29 PM
5/25/2023


OutputTokenList.txt
File Edit View
ROW: 2 COLUMN: 12

TYPE: LEFTSQUAREB
VALUE: [
ROW: 2 COLUMN: 13

TYPE: IDENTIFIER
VALUE: prev
ROW: 2 COLUMN: 14

TYPE: NUMBER
VALUE: 0
ROW: 2 COLUMN: 19

TYPE: RIGHTSQUAREB
VALUE:]
ROW: 2 COLUMN: 20

TYPE: LEFTSQUAREB
VALUE: [
ROW: 3 COLUMN: 13

TYPE: IDENTIFIER
VALUE: cur
ROW: 3 COLUMN: 14

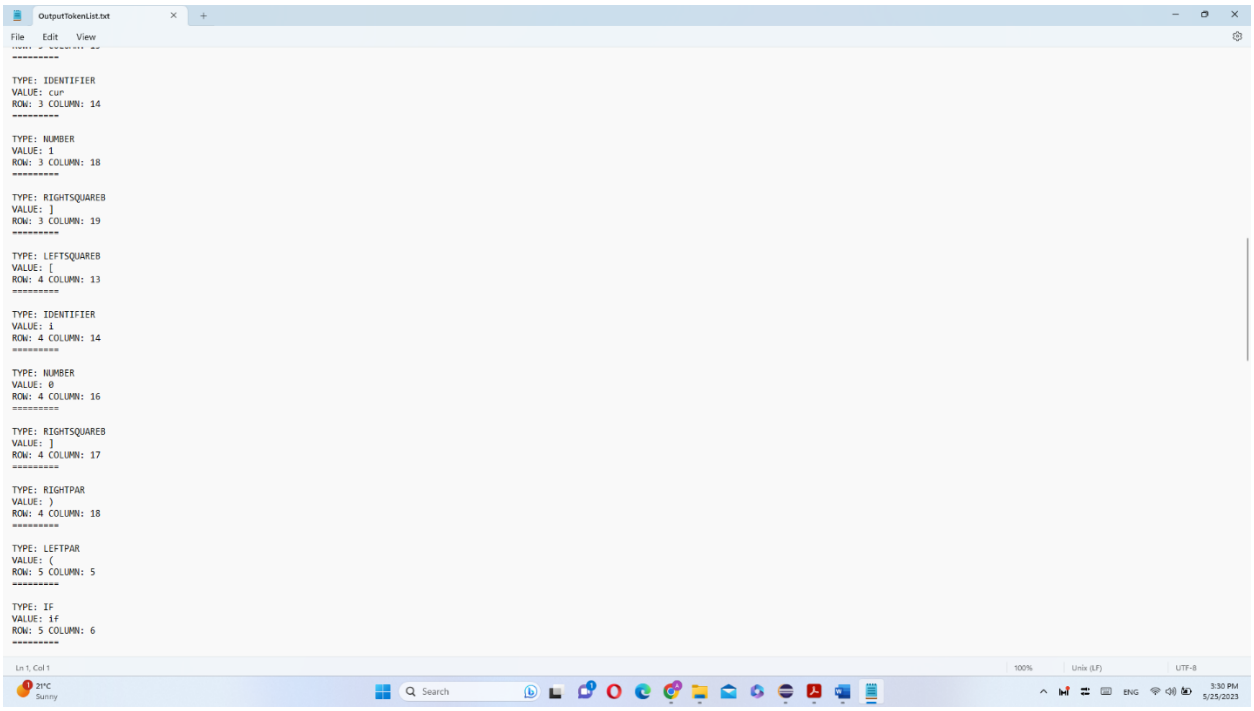
TYPE: NUMBER
VALUE: 1
ROW: 3 COLUMN: 18

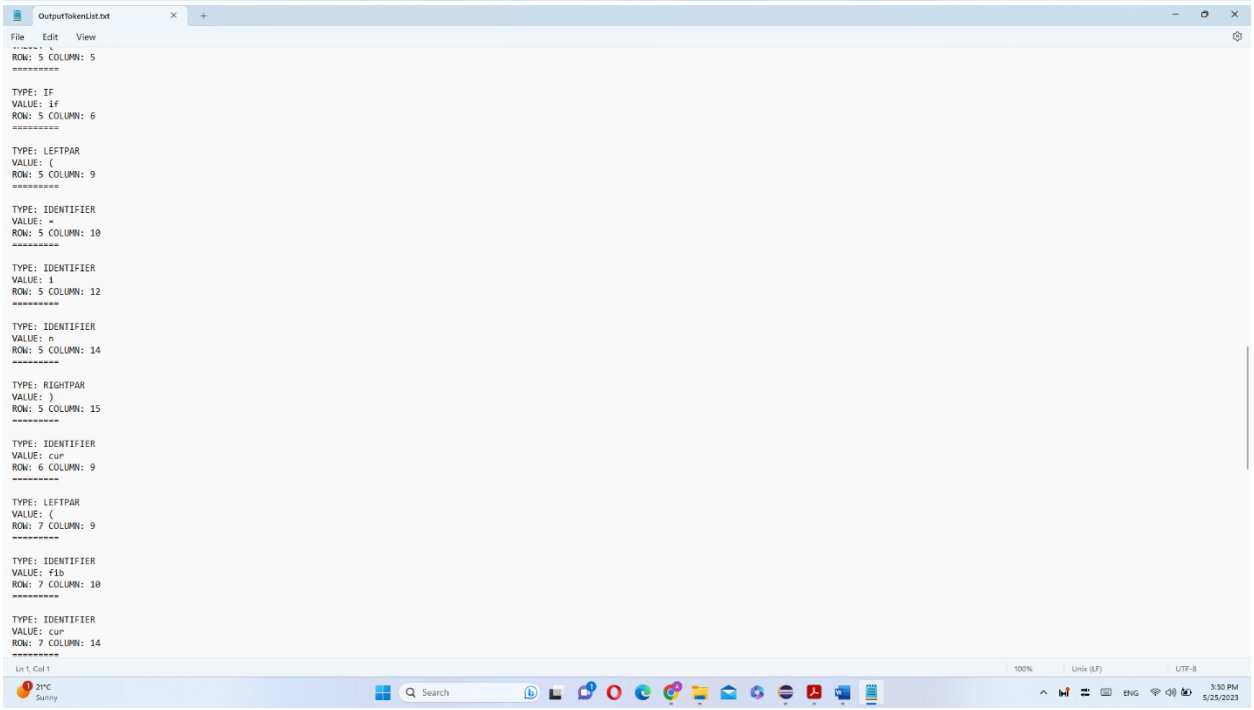
TYPE: RIGHTSQUAREB
VALUE:]
ROW: 3 COLUMN: 19

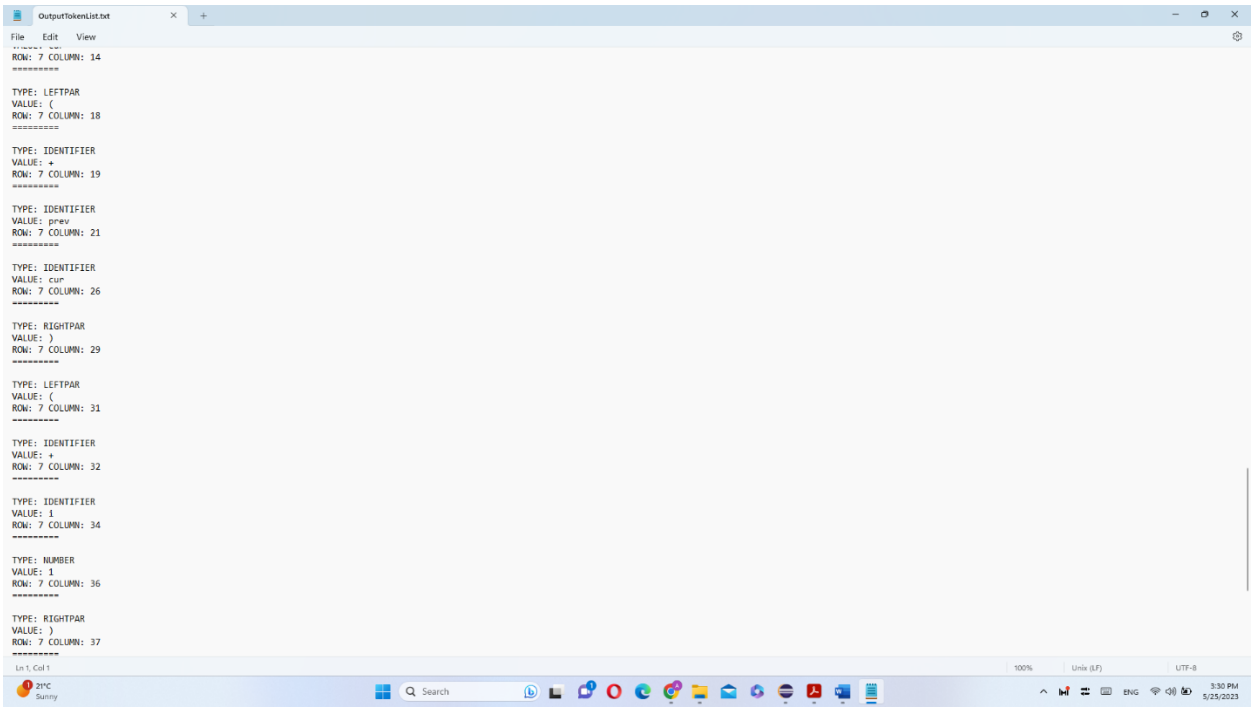
TYPE: LEFTSQUAREB
VALUE: [
ROW: 4 COLUMN: 13

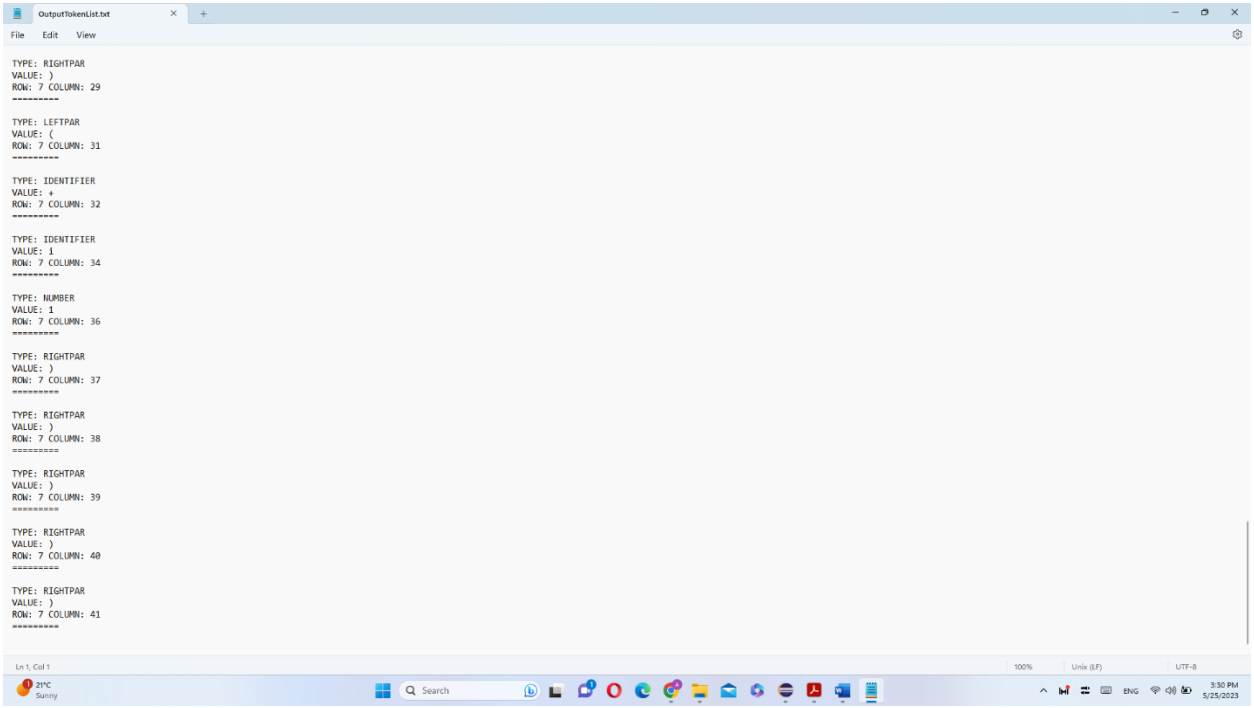
TYPE: IDENTIFIER
VALUE: i
ROW: 4 COLUMN: 14

Ln 1, Col 1
100%
Unix (LF)
UTF-8
21°C
Sunny
Search
ENG
3:30 PM
5/25/2023

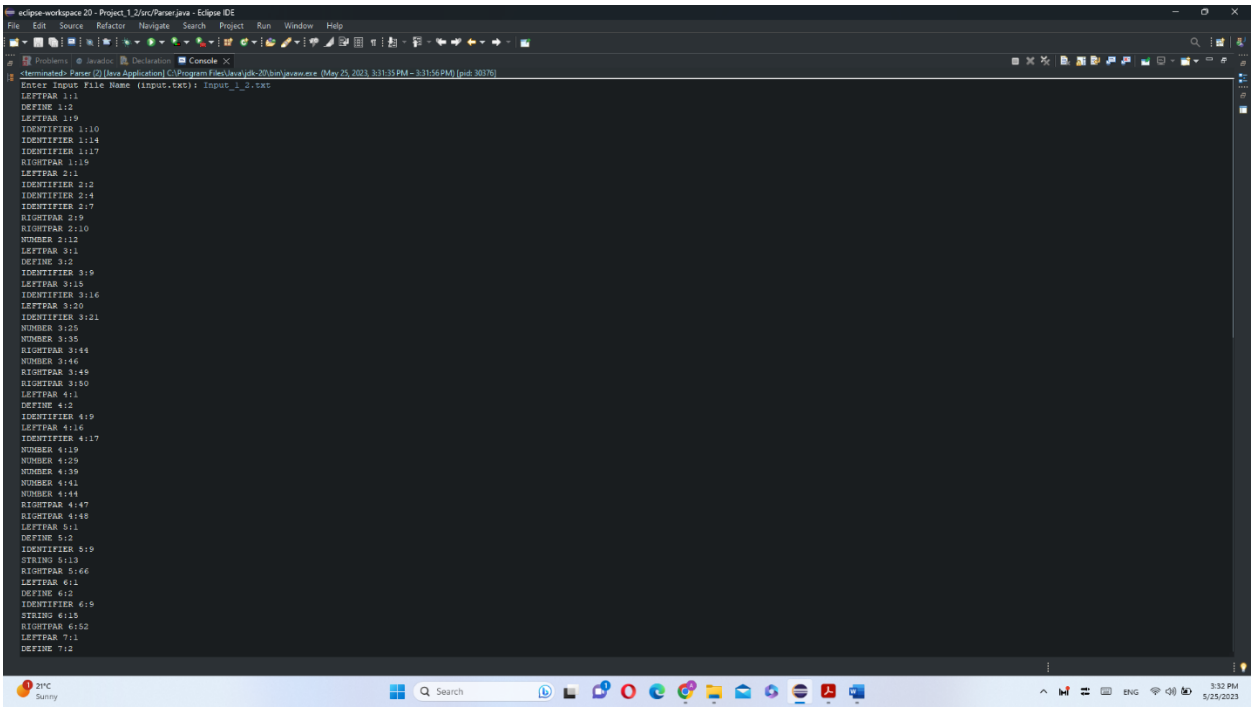

OutputTokenList.txt
File Edit View
TYPE: IDENTIFIER
VALUE: cur
ROW: 3 COLUMN: 14
TYPE: NUMBER
VALUE: 1
ROW: 3 COLUMN: 18
TYPE: RIGHTSQUAREB
VALUE:]
ROW: 3 COLUMN: 19
TYPE: LEFTSQUAREB
VALUE: [
ROW: 4 COLUMN: 13
TYPE: IDENTIFIER
VALUE: i
ROW: 4 COLUMN: 14
TYPE: NUMBER
VALUE: 0
ROW: 4 COLUMN: 16
TYPE: RIGHTSQUAREB
VALUE:]
ROW: 4 COLUMN: 17
TYPE: RIGHTPAR
VALUE:)
ROW: 4 COLUMN: 18
TYPE: LEFTPAR
VALUE: (
ROW: 5 COLUMN: 5
TYPE: IF
VALUE: if
ROW: 5 COLUMN: 6
Ln 1, Col 1
100%
Unix (LF)
UTF-8
21°C
Sunny
Search
ENG
3:30 PM
5/25/2023

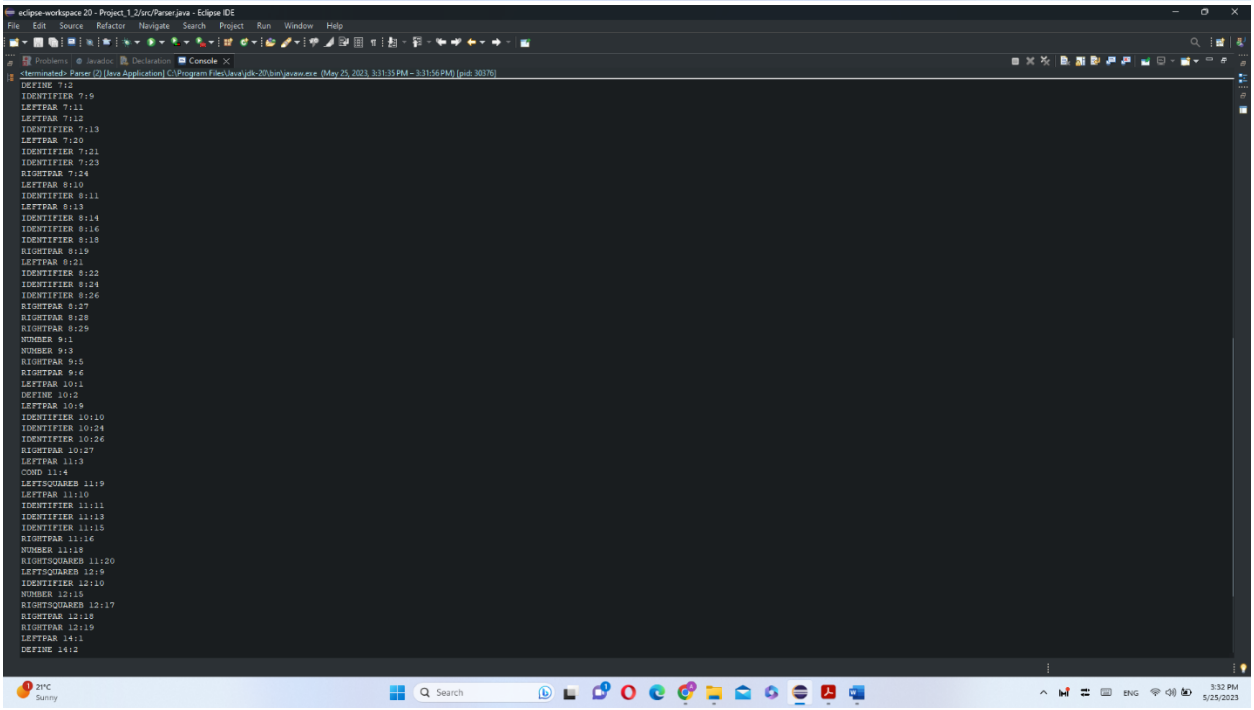

OutputTokenList.txt
File Edit View
ROW: 5 COLUMN: 5
TYPE: IF
VALUE: if
ROW: 5 COLUMN: 6
TYPE: LEFTPAR
VALUE: (
ROW: 5 COLUMN: 9
TYPE: IDENTIFIER
VALUE: =
ROW: 5 COLUMN: 10
TYPE: IDENTIFIER
VALUE: i
ROW: 5 COLUMN: 12
TYPE: IDENTIFIER
VALUE: n
ROW: 5 COLUMN: 14
TYPE: RIGHTPAR
VALUE:)
ROW: 5 COLUMN: 15
TYPE: IDENTIFIER
VALUE: cur
ROW: 6 COLUMN: 9
TYPE: LEFTPAR
VALUE: (
ROW: 7 COLUMN: 9
TYPE: IDENTIFIER
VALUE: fib
ROW: 7 COLUMN: 10
TYPE: IDENTIFIER
VALUE: cur
ROW: 7 COLUMN: 14
Ln 1, Col 1
100%
Unix (LF)
UTF-8
21°C
Sunny
Search
ENG
3:30 PM
5/25/2023

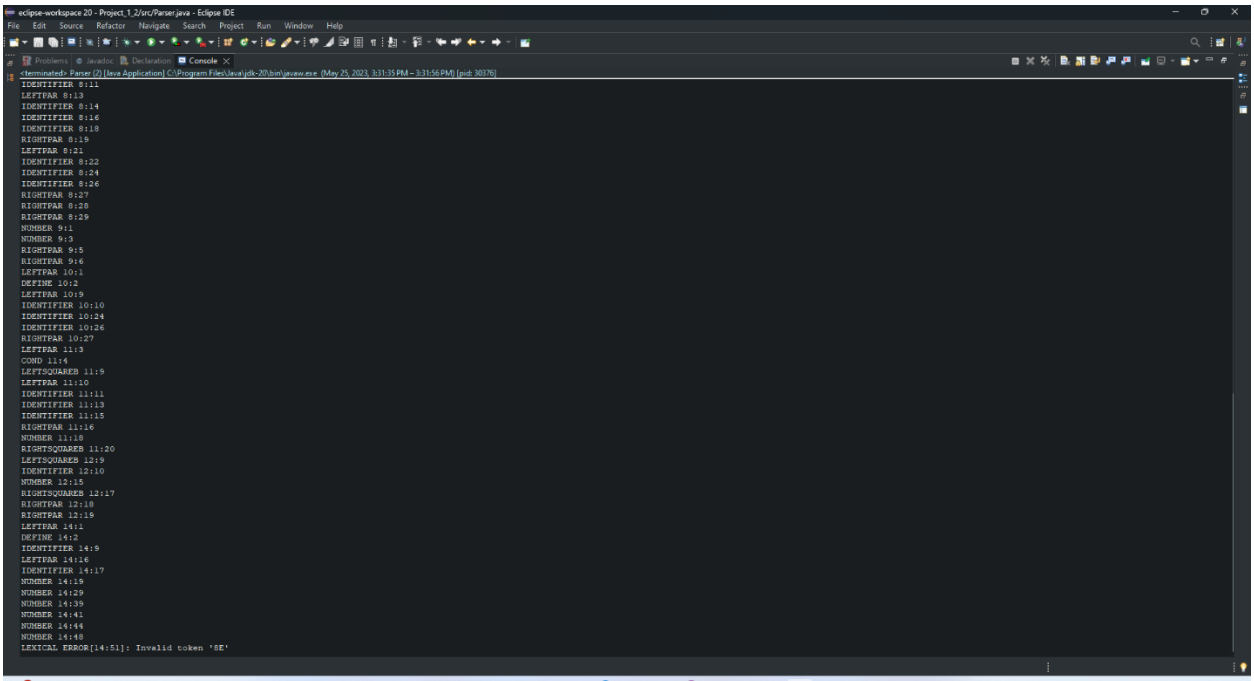


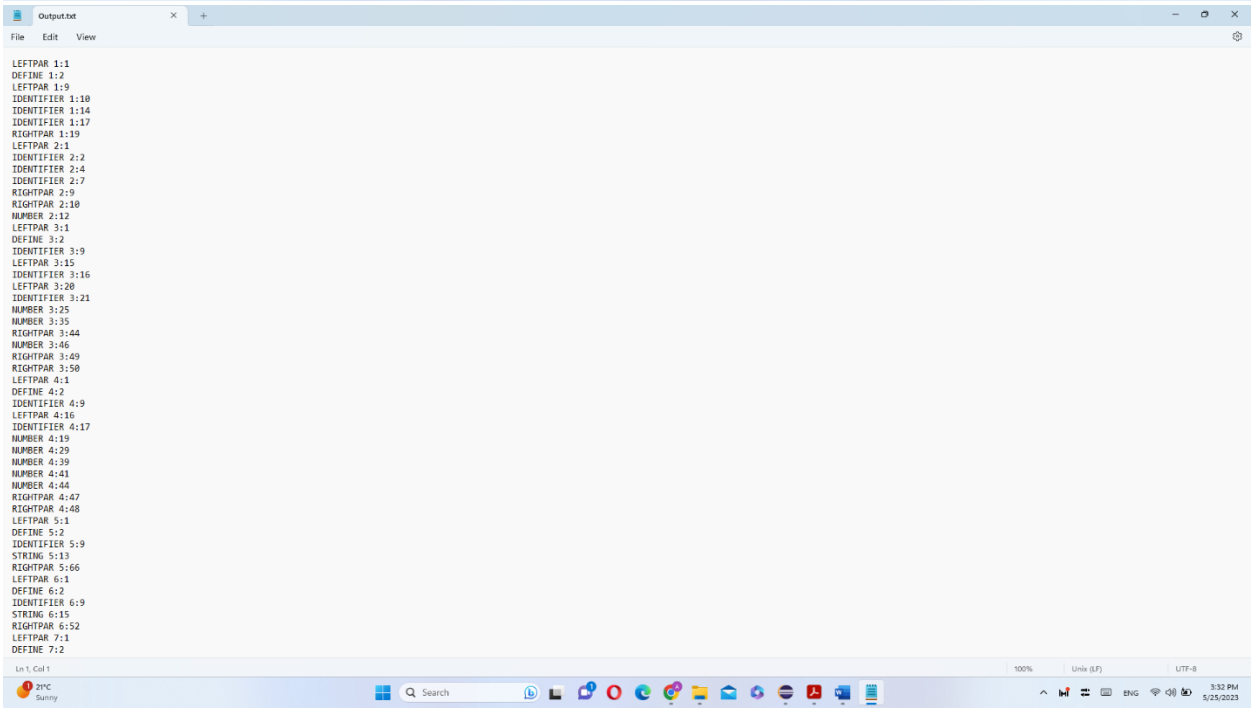


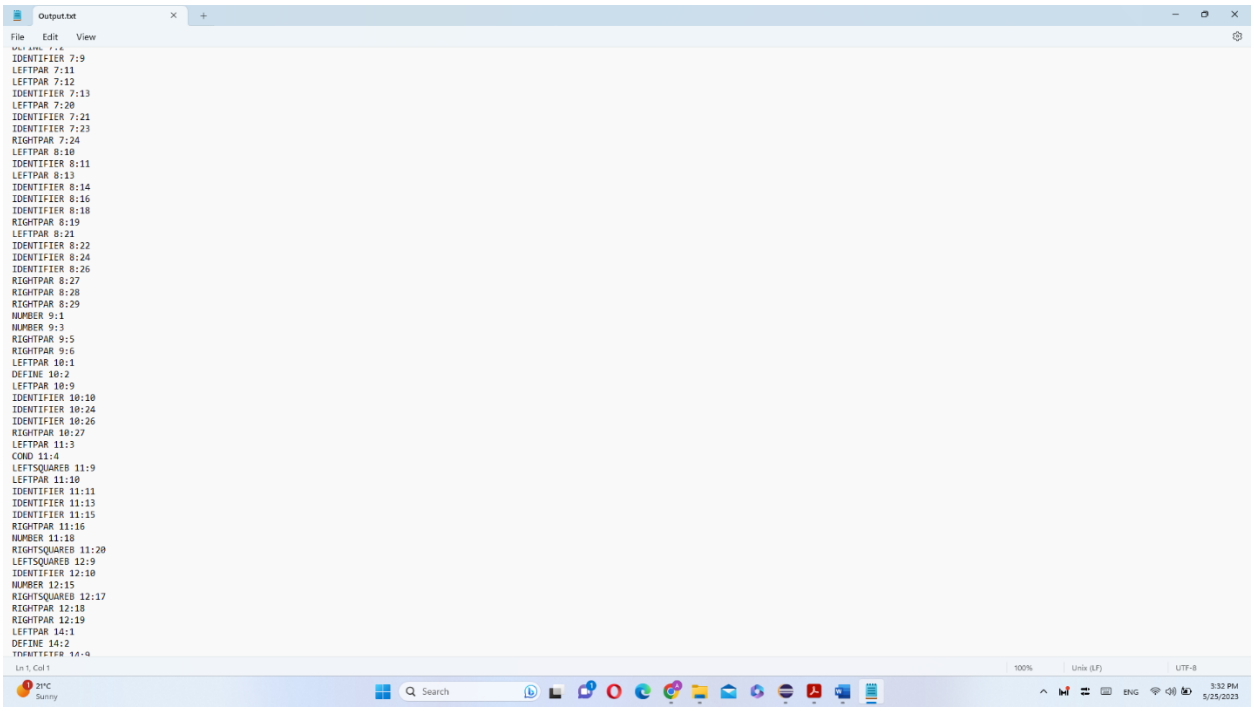
Input File (2): Input_1_2.txt

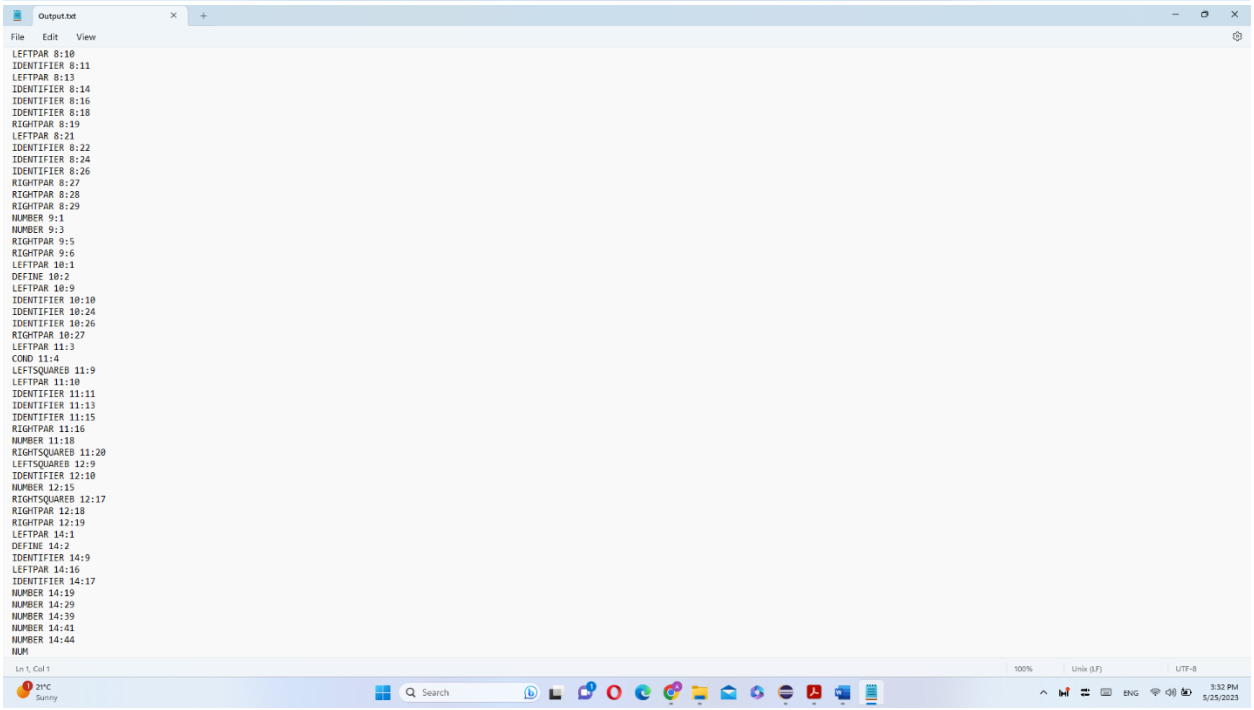

eclipse-workspace 20 - Project_1_2/src/Parser.java - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help
Problems Javadoc Declaration Console
<terminated> Parser (2) [Java Application] C:\Program Files\Java\jdk-20\bin\javaw.exe (May 25, 2023, 3:31:35 PM – 3:31:56 PM) [pid: 30376]
Enter Input File Name (input.txt): Input_1_2.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
NUMBER 2:12
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 4:29
NUMBER 4:39
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
21°C Sunny
Search
ENG
3:32 PM 5/25/2023

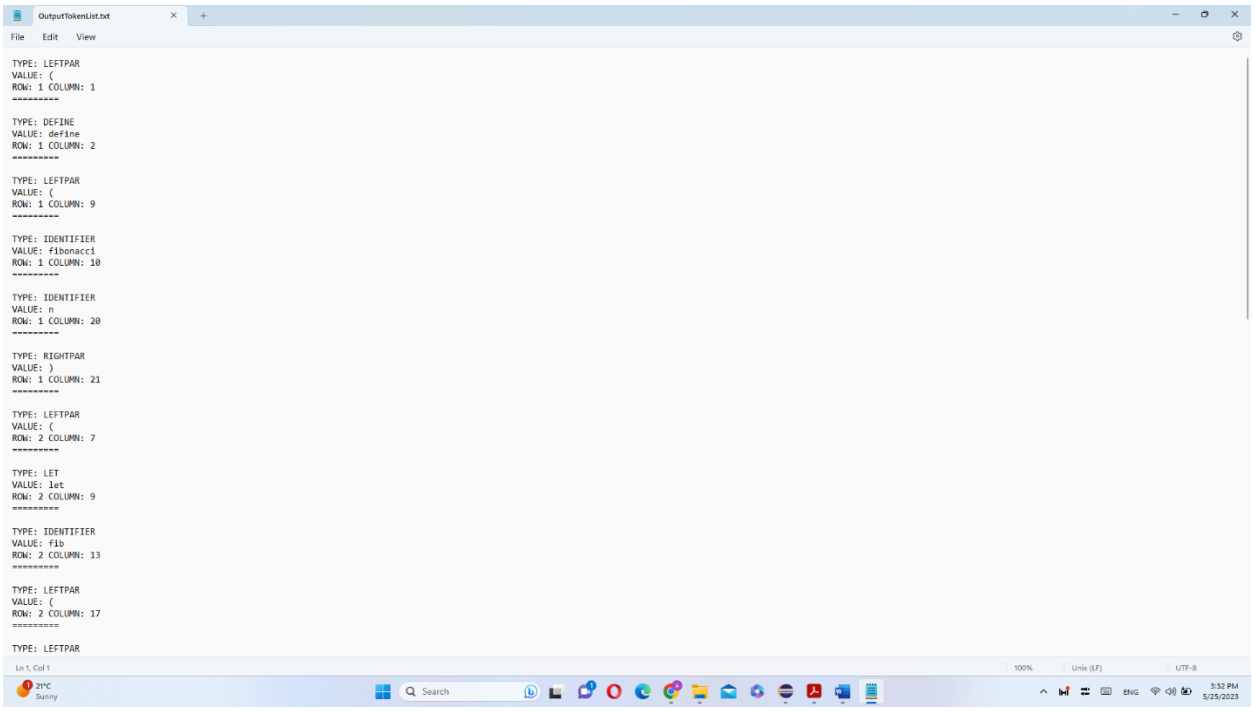

eclipse-workspace 20 - Project_1_2/src/Parser.java - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help
Problems Javadoc Declaration Console
<terminated> Parser (2) [Java Application] C:\Program Files\Java\jdk-20\bin\javaw.exe (May 25, 2023, 3:31:35 PM – 3:31:56 PM) [pid: 30376]
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
RIGHTPAR 12:18
RIGHTPAR 12:19
LEFTPAR 14:1
DEFINE 14:2
21°C Sunny
Search
ENG
3:32 PM 5/25/2023

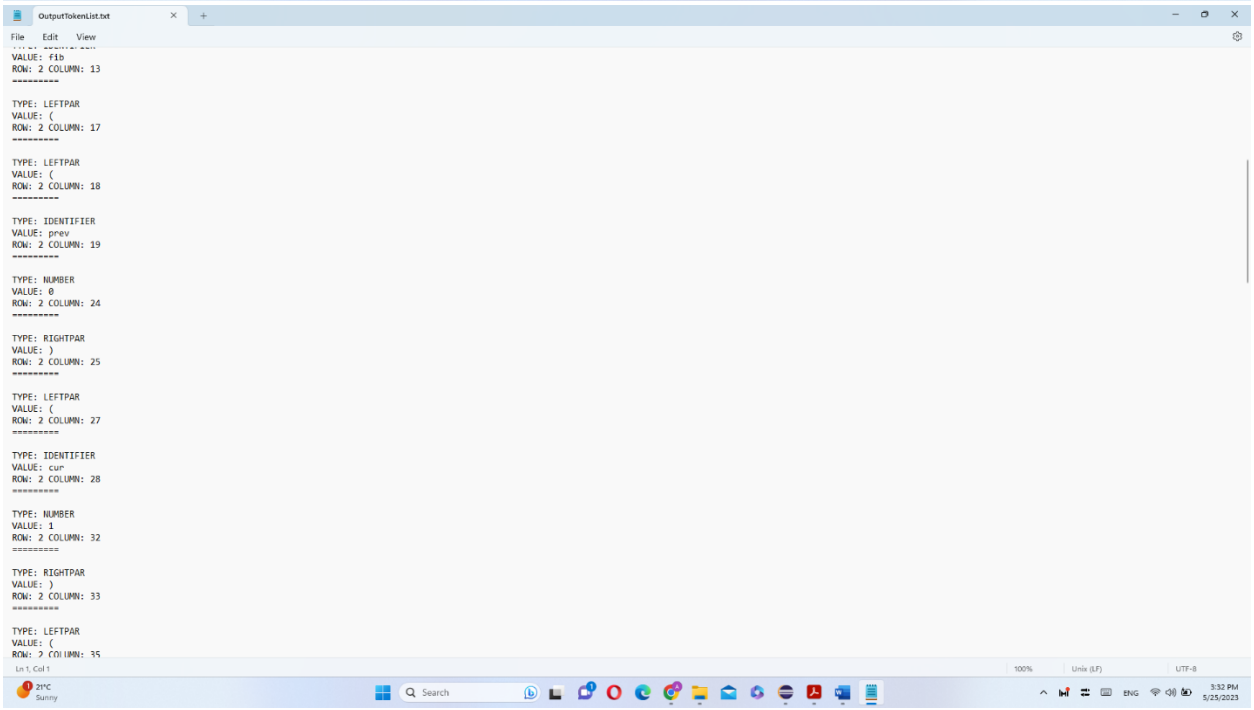

eclipse-workspace 20 - Project_1_2/src/Parser.java - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help
Problems Javadoc Declaration Console
<terminated> Parser (2) [Java Application] C:\Program Files\Java\jdk-20\bin\javaw.exe (May 25, 2023, 3:31:35 PM – 3:31:56 PM) [pid: 30376]
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
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 14:1
DEFINE 14:2
IDENTIFIER 14:9
LEFTPAR 14:16
IDENTIFIER 14:17
NUMBER 14:19
NUMBER 14:29
NUMBER 14:39
NUMBER 14:41
NUMBER 14:44
NUMBER 14:48
LEXICAL ERROR[14:51]: Invalid token '8E'
21°C Sunny
Search
ENG
3:32 PM 5/25/2023

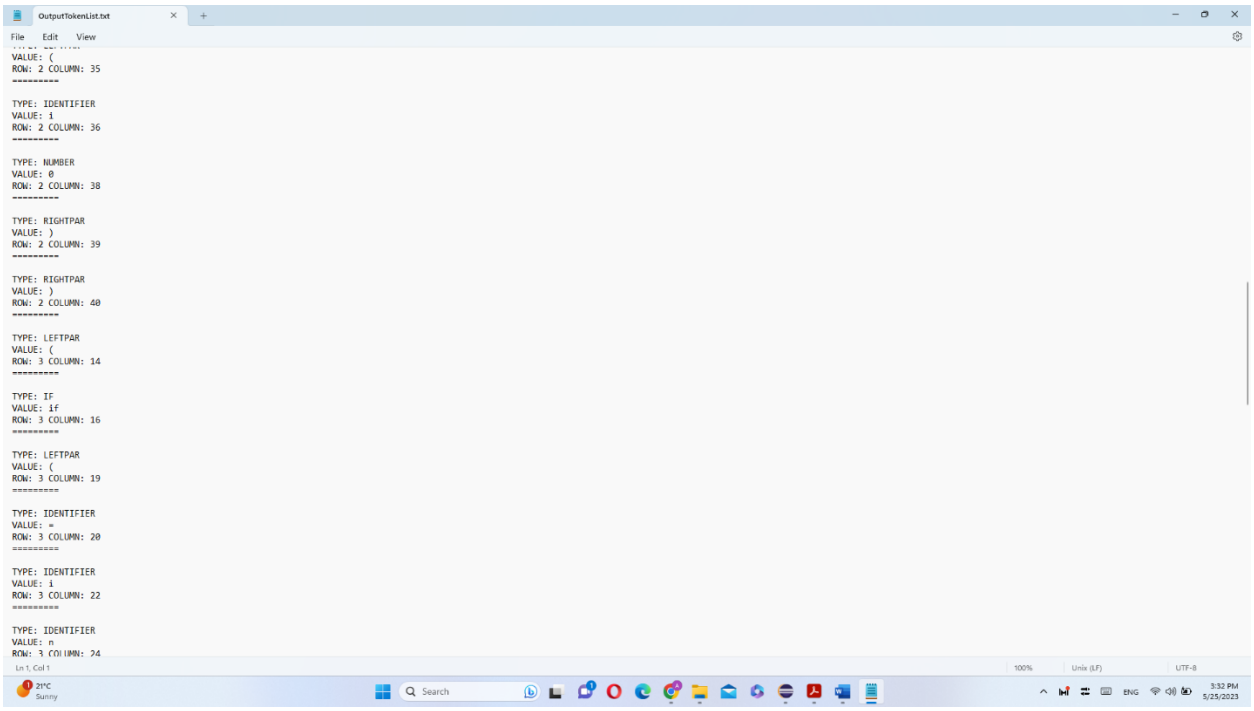

Output.txt
File Edit View
LEFTPAR 1:1
DEFINE 1:2
LEFTPAR 1:9
IDENTIFIER 1:10
IDENTIFIER 1:14
IDENTIFIER 1:17
RIGHTPAR 1:19
LEFTPAR 2:1
IDENTIFIER 2:2
IDENTIFIER 2:4
IDENTIFIER 2:7
RIGHTPAR 2:9
RIGHTPAR 2:10
NUMBER 2:12
LEFTPAR 3:1
DEFINE 3:2
IDENTIFIER 3:9
LEFTPAR 3:15
IDENTIFIER 3:16
LEFTPAR 3:20
IDENTIFIER 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 4:29
NUMBER 4:39
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
Ln 1, Col 1
100%
Unix (LF)
UTF-8
21°C Sunny
Search
ENG
3:32 PM 5/25/2023

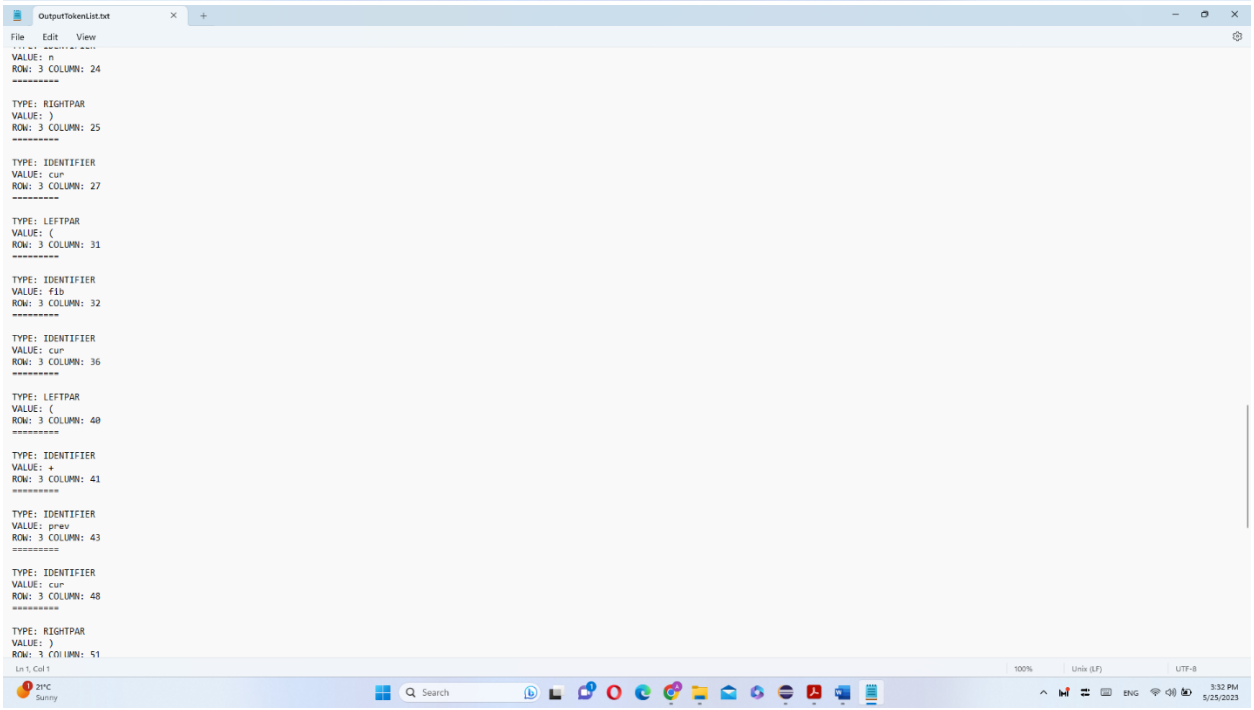

Output.txt
File Edit View
IDENTIFIER 7:9
LEFTPAR 7:11
LEFTPAR 7:12
IDENTIFIER 7:13
LEFTPAR 7:20
IDENTIFIER 7:21
IDENTIFIER 7:23
RIGHTPAR 7:24
LEFTPAR 8:10
IDENTIFIER 8:11
LEFTPAR 8:13
IDENTIFIER 8:14
IDENTIFIER 8:16
IDENTIFIER 8:18
RIGHTPAR 8:19
LEFTPAR 8:21
IDENTIFIER 8:22
IDENTIFIER 8:24
IDENTIFIER 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
RIGHTPAR 12:18
RIGHTPAR 12:19
LEFTPAR 14:1
DEFINE 14:2
Ln 1, Col 1
100%
Unix (LF)
UTF-8
21°C
Sunny
Search
ENG
3:32 PM
5/25/2023

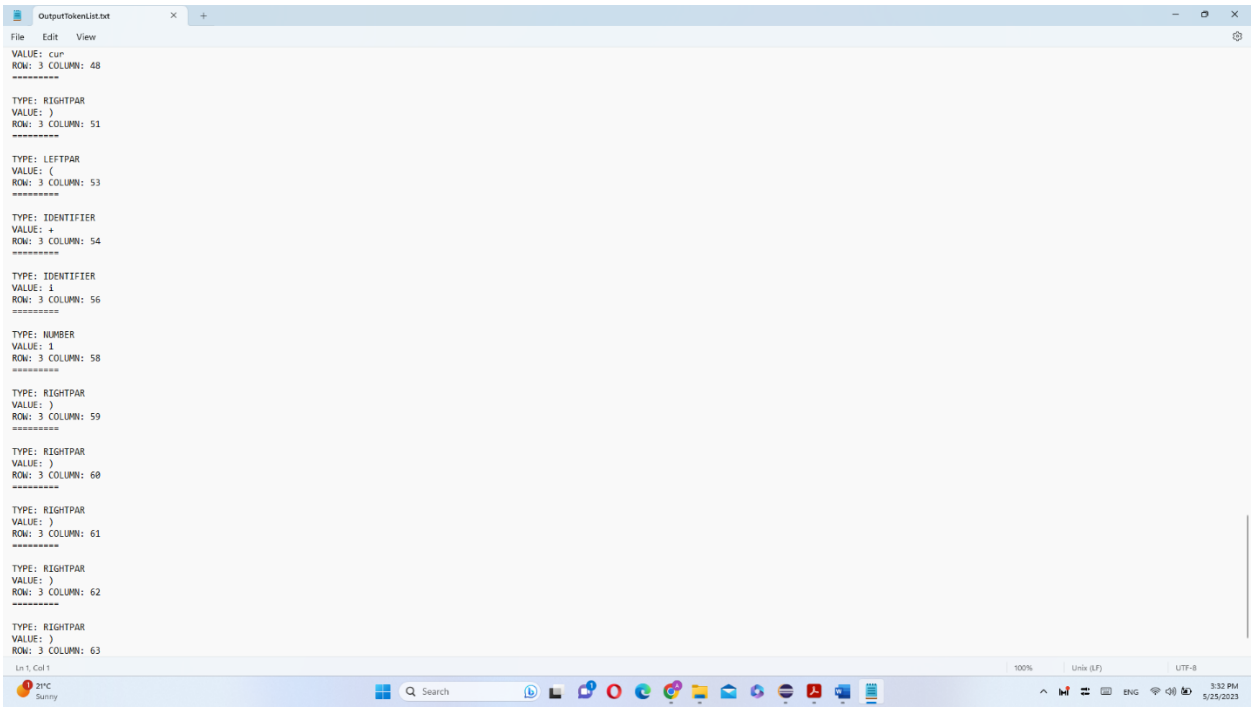

Output.txt
File Edit View
LEFTPAR 8:10
IDENTIFIER 8:11
LEFTPAR 8:13
IDENTIFIER 8:14
IDENTIFIER 8:16
IDENTIFIER 8:18
RIGHTPAR 8:19
LEFTPAR 8:21
IDENTIFIER 8:22
IDENTIFIER 8:24
IDENTIFIER 8:26
RIGHTPAR 8:27
RIGHTPAR 8:28
RIGHTPAR 8:29
NUMBER 9:1
NUMBER 9:3
RIGHTPAR 9:5
RIGHTPAR 9:6
LEFTPAR 10:1
DEFINE 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
RIGHTPAR 12:18
RIGHTPAR 12:19
LEFTPAR 14:1
DEFINE 14:2
IDENTIFIER 14:9
LEFTPAR 14:16
IDENTIFIER 14:17
NUMBER 14:19
NUMBER 14:29
NUMBER 14:39
NUMBER 14:41
NUMBER 14:44
NUM
Ln 1, Col 1
100%
Unix (LF)
UTF-8
21°C
Sunny
Search
ENG
3:32 PM
5/25/2023

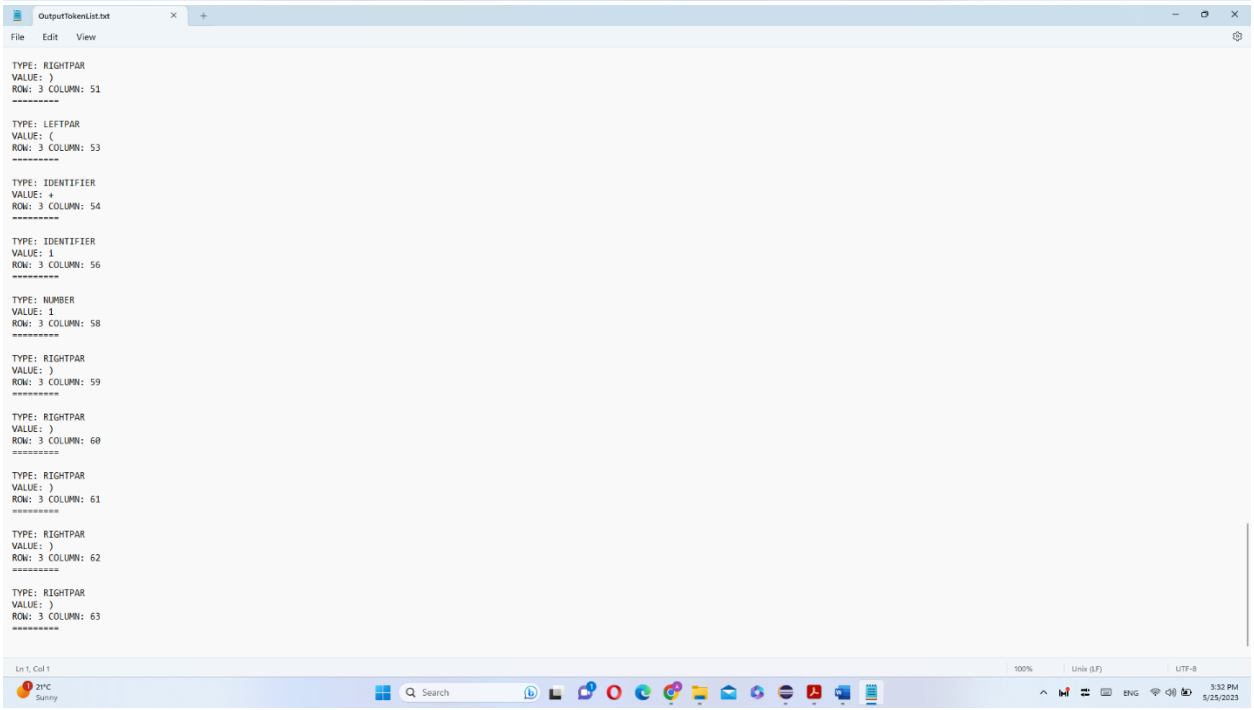

OutputTokenList.txt
File Edit View
TYPE: LEFTPAR
VALUE: (
ROW: 1 COLUMN: 1
=========
TYPE: DEFINE
VALUE: define
ROW: 1 COLUMN: 2
=========
TYPE: LEFTPAR
VALUE: (
ROW: 1 COLUMN: 9
=========
TYPE: IDENTIFIER
VALUE: fibonacci
ROW: 1 COLUMN: 10
=========
TYPE: IDENTIFIER
VALUE: n
ROW: 1 COLUMN: 20
=========
TYPE: RIGHTPAR
VALUE:)
ROW: 1 COLUMN: 21
=========
TYPE: LEFTPAR
VALUE: (
ROW: 2 COLUMN: 7
=========
TYPE: LET
VALUE: let
ROW: 2 COLUMN: 9
=========
TYPE: IDENTIFIER
VALUE: fib
ROW: 2 COLUMN: 13
=========
TYPE: LEFTPAR
VALUE: (
ROW: 2 COLUMN: 17
=========
TYPE: LEFTPAR
Ln 1, Col 1
100%
Unix (LF)
UTF-8
21°C
Sunny
Search
ENG
3:32 PM
5/25/2023


OutputTokenList.txt
File Edit View
VALUE: fib
ROW: 2 COLUMN: 13
=========
TYPE: LEFTPAR
VALUE: (
ROW: 2 COLUMN: 17
=========
TYPE: LEFTPAR
VALUE: (
ROW: 2 COLUMN: 18
=========
TYPE: IDENTIFIER
VALUE: prev
ROW: 2 COLUMN: 19
=========
TYPE: NUMBER
VALUE: 0
ROW: 2 COLUMN: 24
=========
TYPE: RIGHTPAR
VALUE:)
ROW: 2 COLUMN: 25
=========
TYPE: LEFTPAR
VALUE: (
ROW: 2 COLUMN: 27
=========
TYPE: IDENTIFIER
VALUE: cur
ROW: 2 COLUMN: 28
=========
TYPE: NUMBER
VALUE: 1
ROW: 2 COLUMN: 32
=========
TYPE: RIGHTPAR
VALUE:)
ROW: 2 COLUMN: 33
=========
TYPE: LEFTPAR
VALUE: (
Ln 1, Col 1
100%
Unix (LF)
UTF-8
21°C
Sunny
Search
ENG
3:32 PM
5/25/2023

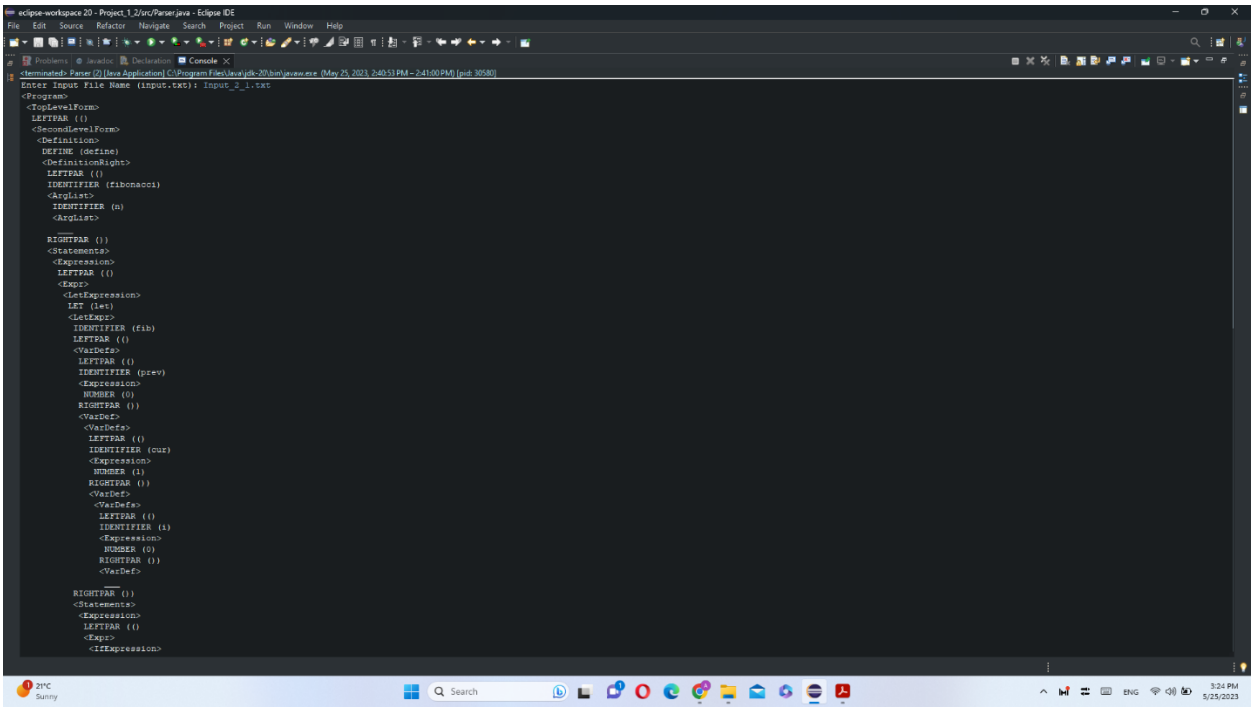

OutputTokenList.txt
File Edit View
VALUE: (
ROW: 2 COLUMN: 35
=========
TYPE: IDENTIFIER
VALUE: i
ROW: 2 COLUMN: 36
=========
TYPE: NUMBER
VALUE: 0
ROW: 2 COLUMN: 38
=========
TYPE: RIGHTPAR
VALUE:)
ROW: 2 COLUMN: 39
=========
TYPE: RIGHTPAR
VALUE:)
ROW: 2 COLUMN: 40
=========
TYPE: LEFTPAR
VALUE: (
ROW: 3 COLUMN: 14
=========
TYPE: IF
VALUE: if
ROW: 3 COLUMN: 16
=========
TYPE: LEFTPAR
VALUE: (
ROW: 3 COLUMN: 19
=========
TYPE: IDENTIFIER
VALUE: n
ROW: 3 COLUMN: 20
=========
TYPE: IDENTIFIER
VALUE: i
ROW: 3 COLUMN: 22
=========
TYPE: IDENTIFIER
VALUE: n
ROW: 3 COLUMN: 24
Ln 1, Col 1
100%
Unix (LF)
UTF-8
21°C
Sunny
Search
ENG
3:32 PM
5/25/2023

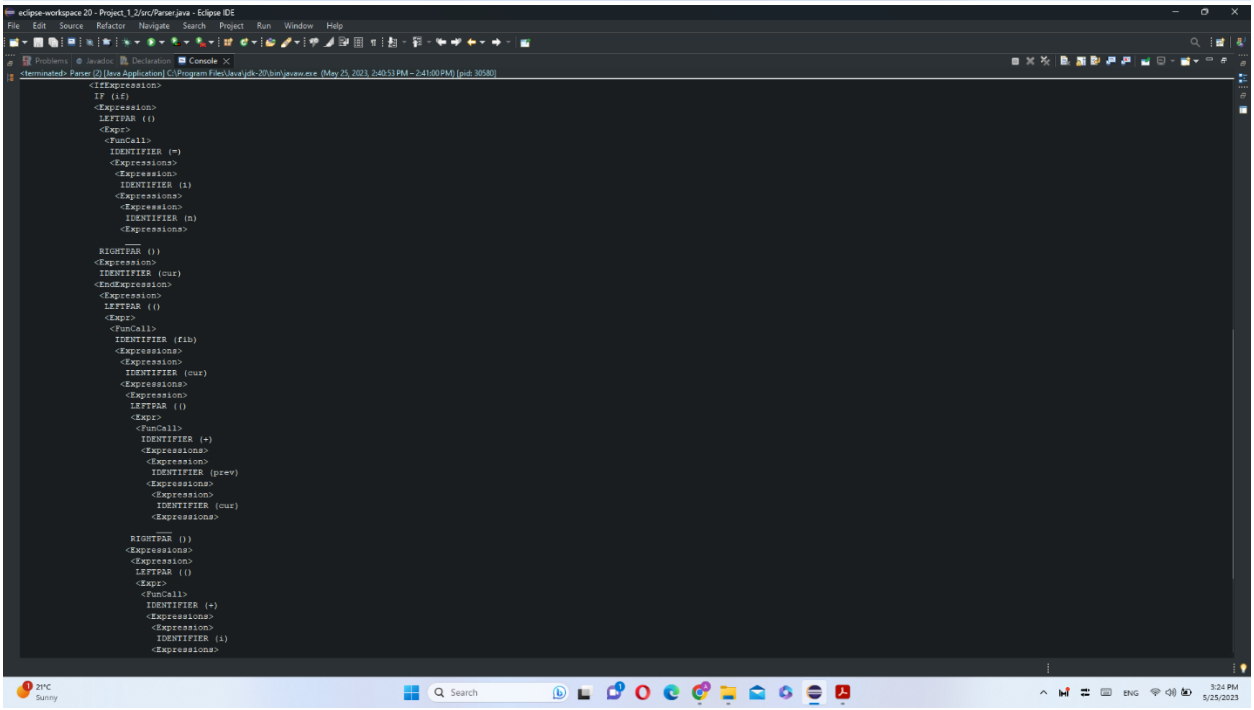

OutputTokenList.txt
File Edit View
VALUE: n
ROW: 3 COLUMN: 24
=========
TYPE: RIGHTPAR
VALUE:)
ROW: 3 COLUMN: 25
=========
TYPE: IDENTIFIER
VALUE: cur
ROW: 3 COLUMN: 27
=========
TYPE: LEFTPAR
VALUE: (
ROW: 3 COLUMN: 31
=========
TYPE: IDENTIFIER
VALUE: fib
ROW: 3 COLUMN: 32
=========
TYPE: IDENTIFIER
VALUE: cur
ROW: 3 COLUMN: 36
=========
TYPE: LEFTPAR
VALUE: (
ROW: 3 COLUMN: 40
=========
TYPE: IDENTIFIER
VALUE: +
ROW: 3 COLUMN: 41
=========
TYPE: IDENTIFIER
VALUE: prev
ROW: 3 COLUMN: 43
=========
TYPE: IDENTIFIER
VALUE: cur
ROW: 3 COLUMN: 48
=========
TYPE: RIGHTPAR
VALUE:)
ROW: 3 COLUMN: 51
Ln 1, Col 1
100%
Unix (LF)
UTF-8
21°C
Sunny
Search
ENG
3:32 PM
5/25/2023

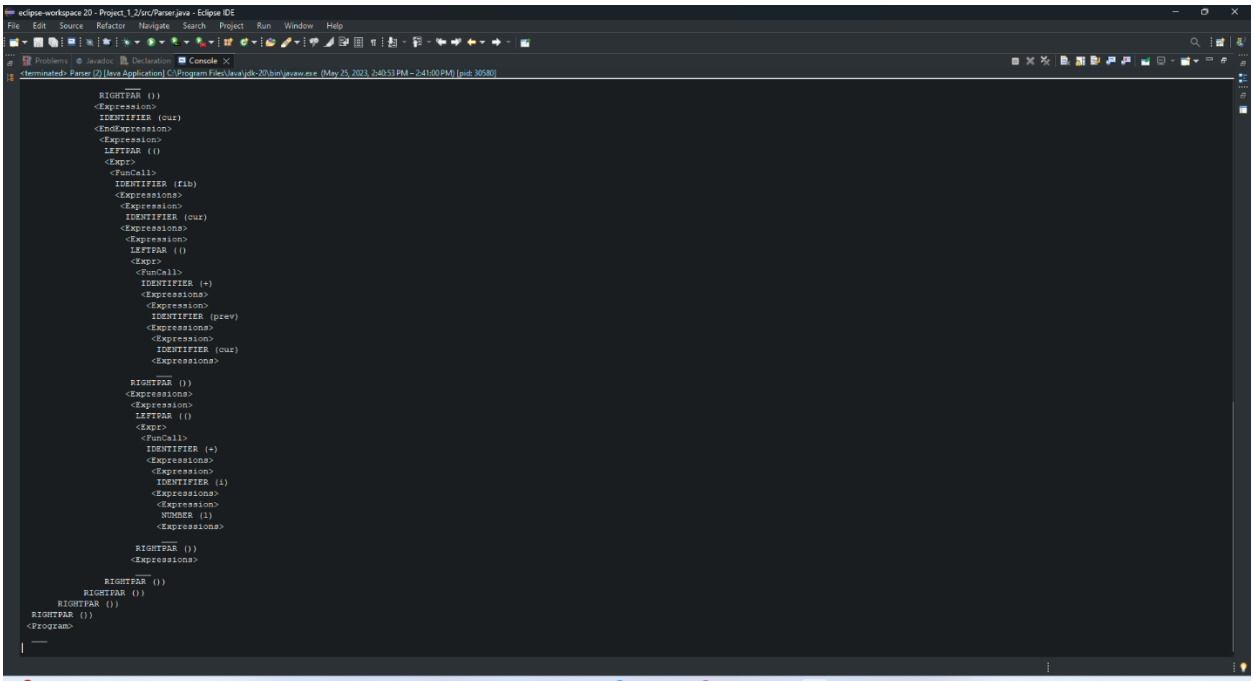


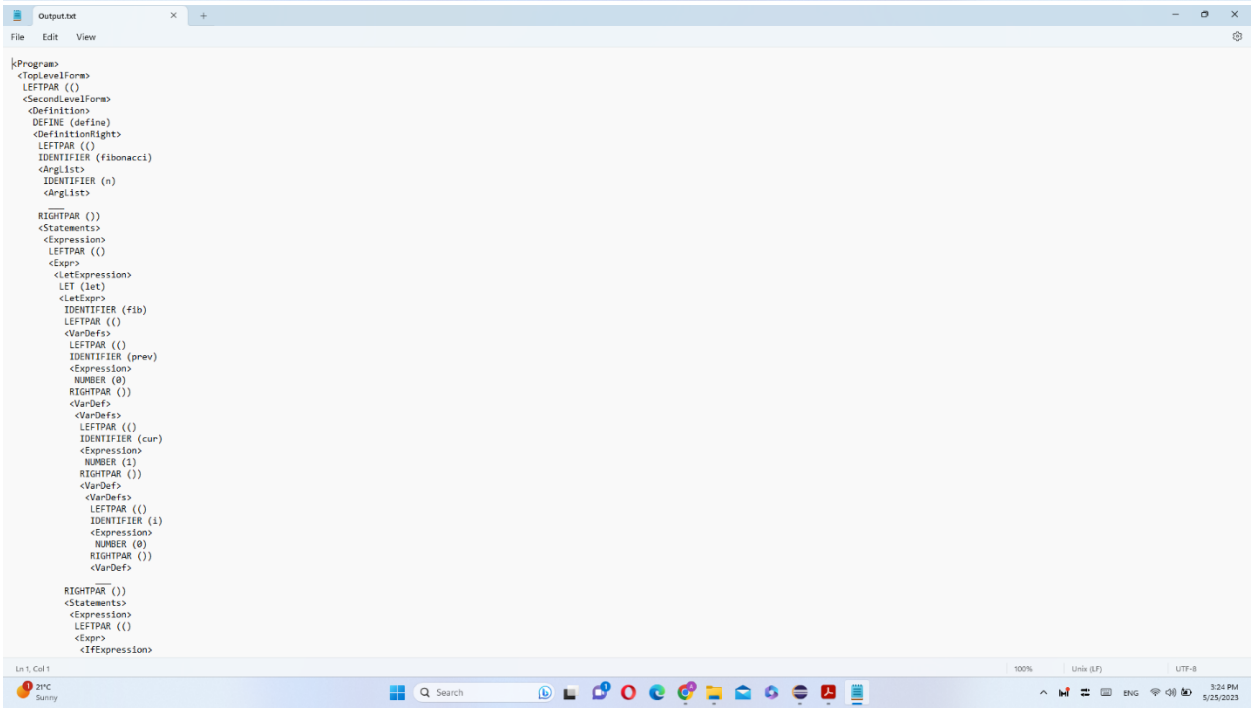


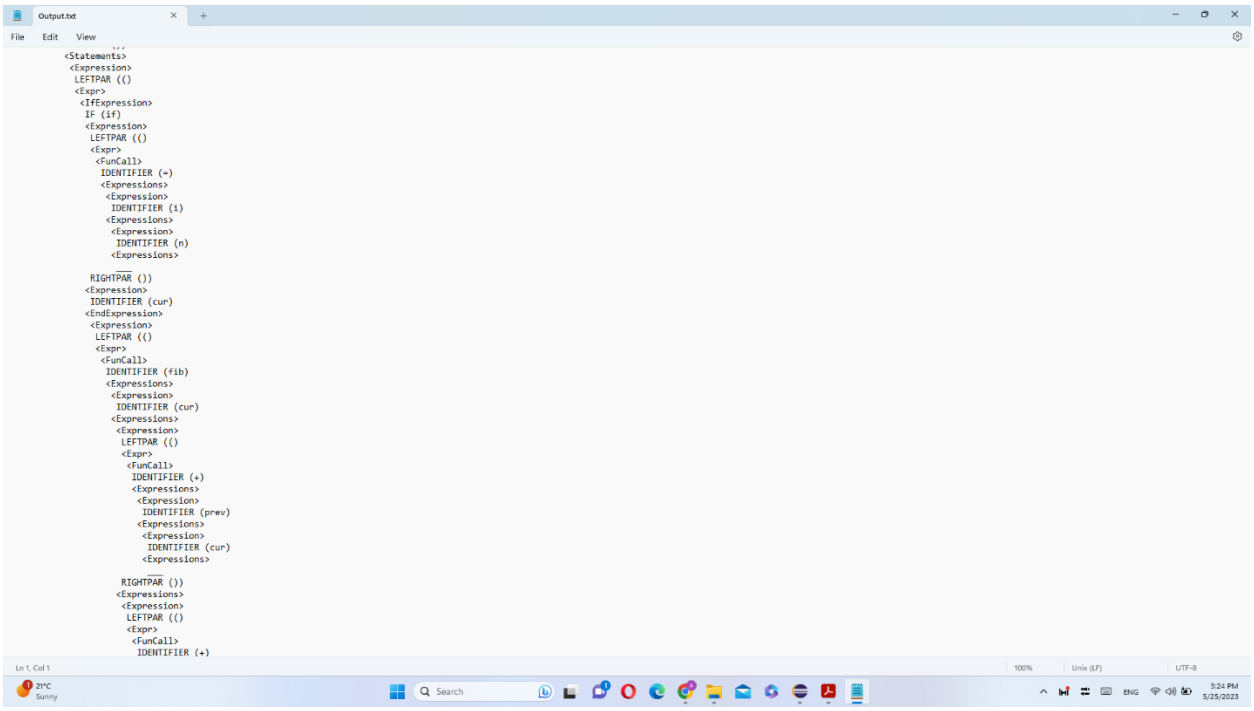
Input File (3): Input_2_1.txt

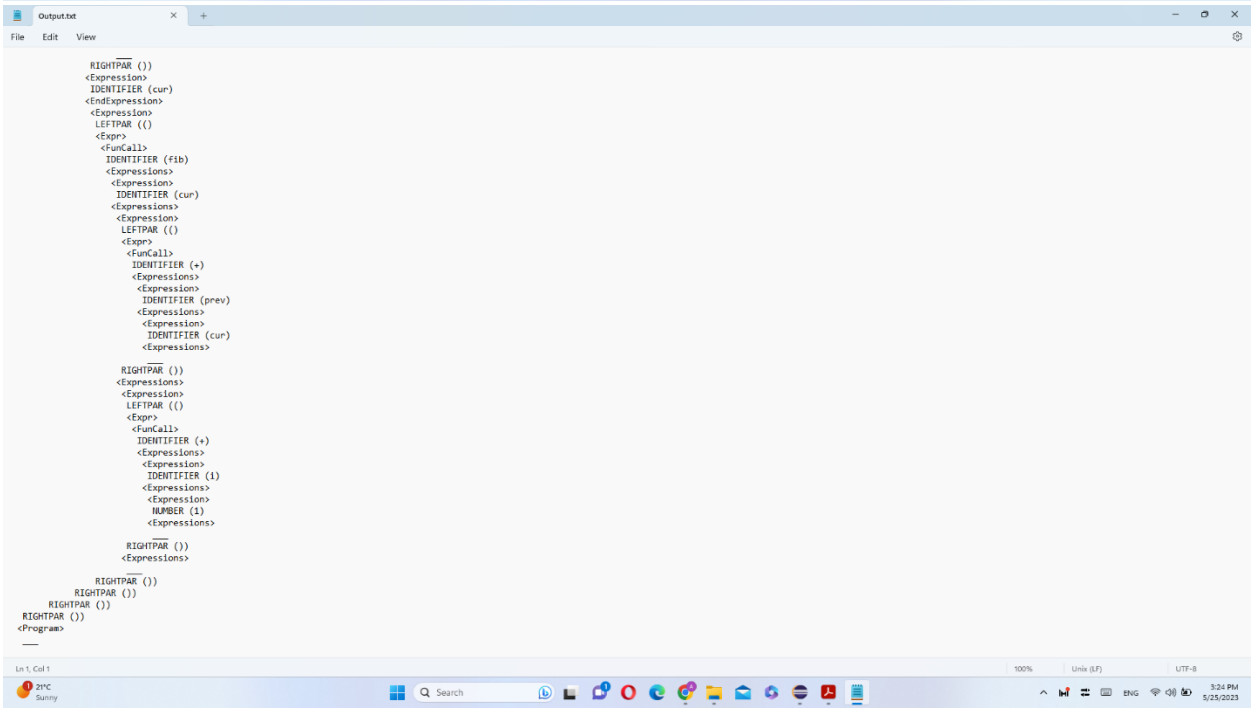

eclipse-workspace 20 - Project_1_2/src/Parser.java - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help
Problems Javadoc Declaration Console
<terminated> Parser (2) [Java Application] C:\Program Files\Java\jdk-20\bin\javaw.exe (May 25, 2023, 2:40:53 PM – 2:41:00 PM) [pid: 30580]
Enter Input File Name (input.txt): Input_2_1.txt
<Program>
<TopLevelForm>
LEFTPAR (()
<SecondLevelForm>
<Definition>
DEFINE (define)
<DefinitionRight>
LEFTPAR (()
IDENTIFIER (fibonacci)
<ArgList>
IDENTIFIER (n)
<ArgList>
RIGHTPAR ())
<Statements>
<Expression>
LEFTPAR (()
<Expr>
<LetExpression>
LET (let)
<LetExpr>
IDENTIFIER (fib)
LEFTPAR (()
<VarDefs>
LEFTPAR (()
IDENTIFIER (prev)
<Expression>
NUMBER (0)
RIGHTPAR ())
<VarDef>
<VarDefs>
LEFTPAR (()
IDENTIFIER (cur)
<Expression>
NUMBER (1)
RIGHTPAR ())
<VarDef>
<VarDefs>
LEFTPAR (()
IDENTIFIER (i)
<Expression>
NUMBER (0)
RIGHTPAR ())
<VarDef>
RIGHTPAR ())
<Statements>
<Expression>
LEFTPAR (()
<Expr>
<IfExpression>
21°C Sunny
Search
ENG
3:24 PM 5/25/2023

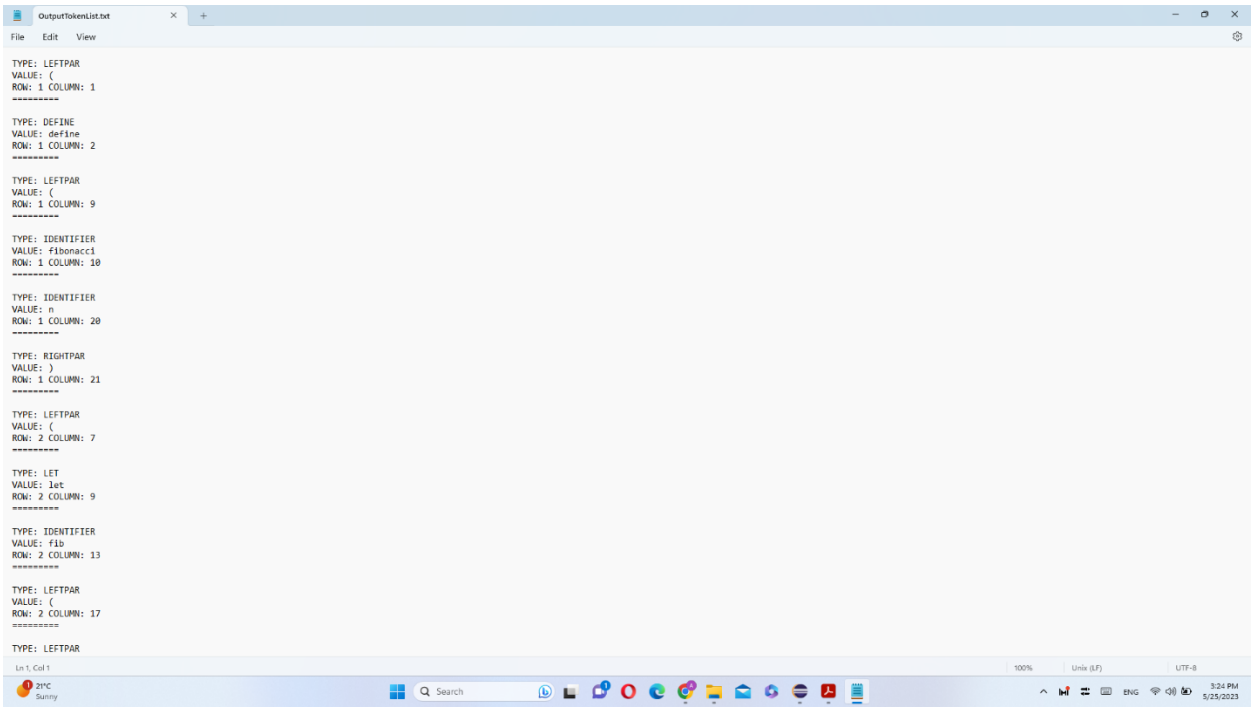

eclipse-workspace 20 - Project_1_2/src/Parser.java - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help
Problems Javadoc Declaration Console
<terminated> Parser (2) [Java Application] C:\Program Files\Java\jdk-20\bin\javaw.exe (May 25, 2023, 2:40:53 PM – 2:41:00 PM) [pid: 30580]
<IfExpression>
IF (if)
<Expression>
LEFTPAR (()
<Expr>
<FunCall>
IDENTIFIER (=)
<Expressions>
<Expression>
IDENTIFIER (i)
<Expressions>
<Expression>
IDENTIFIER (n)
<Expressions>
RIGHTPAR ())
<Expression>
IDENTIFIER (cur)
<EndExpression>
<Expression>
LEFTPAR (()
<Expr>
<FunCall>
IDENTIFIER (fib)
<Expressions>
<Expression>
IDENTIFIER (cur)
<Expressions>
<Expression>
LEFTPAR (()
<Expr>
<FunCall>
IDENTIFIER (+)
<Expressions>
<Expression>
IDENTIFIER (prev)
<Expressions>
<Expression>
IDENTIFIER (cur)
<Expressions>
RIGHTPAR ())
<Expressions>
<Expression>
LEFTPAR (()
<Expr>
<FunCall>
IDENTIFIER (+)
<Expressions>
<Expression>
IDENTIFIER (i)
<Expressions>
21°C Sunny
Search
ENG
3:24 PM 5/25/2023

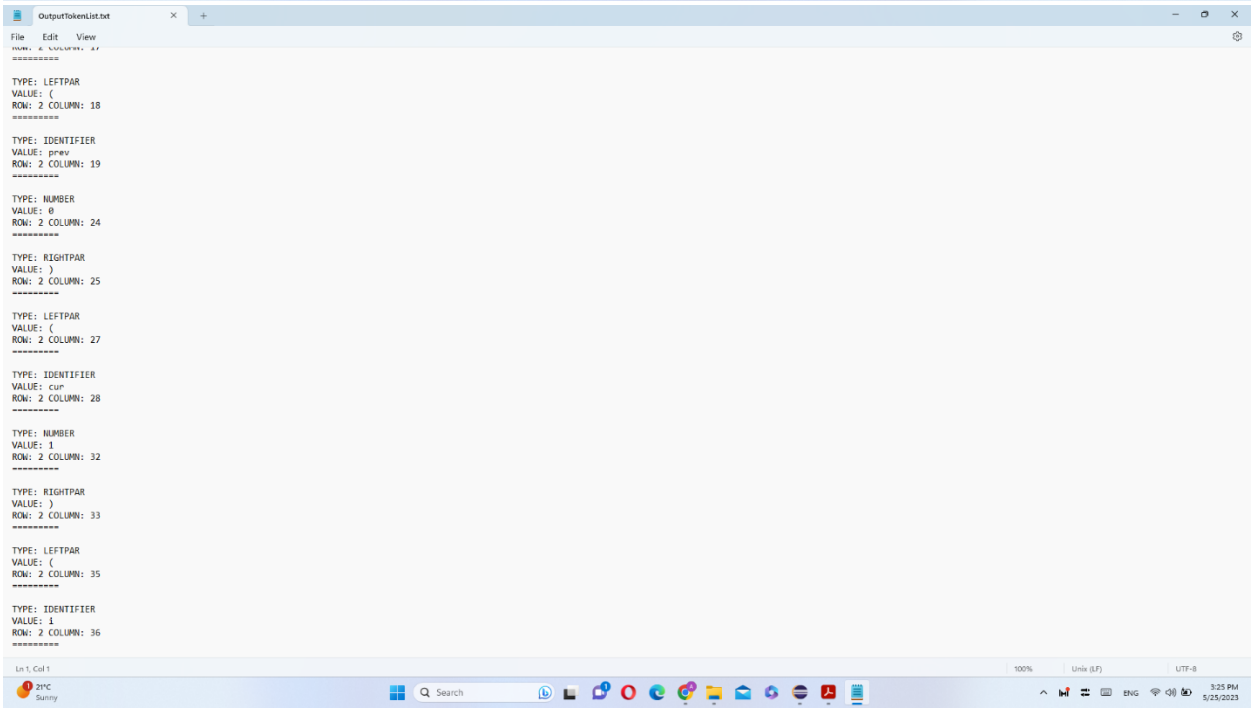

eclipse-workspace 20 - Project_1_2/src/Parser.java - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help
Problems Javadoc Declaration Console
<terminated> Parser (2) [Java Application] C:\Program Files\Java\jdk-20\bin\javaw.exe (May 25, 2023, 2:40:53 PM – 2:41:00 PM) [pid: 30580]
RIGHTPAR ())
<Expression>
IDENTIFIER (cur)
<EndExpression>
<Expression>
LEFTPAR (()
<Expr>
<FunCall>
IDENTIFIER (fib)
<Expressions>
<Expression>
IDENTIFIER (cur)
<Expressions>
<Expression>
LEFTPAR (()
<Expr>
<FunCall>
IDENTIFIER (+)
<Expressions>
<Expression>
IDENTIFIER (prev)
<Expressions>
<Expression>
IDENTIFIER (cur)
<Expressions>
RIGHTPAR ())
<Expressions>
<Expression>
LEFTPAR (()
<Expr>
<FunCall>
IDENTIFIER (+)
<Expressions>
<Expression>
IDENTIFIER (i)
<Expressions>
<Expression>
NUMBER (1)
<Expressions>
RIGHTPAR ())
<Expressions>
RIGHTPAR ())
RIGHTPAR ())
RIGHTPAR ())
<Program>
21°C Sunny
Search
ENG
3:24 PM 5/25/2023

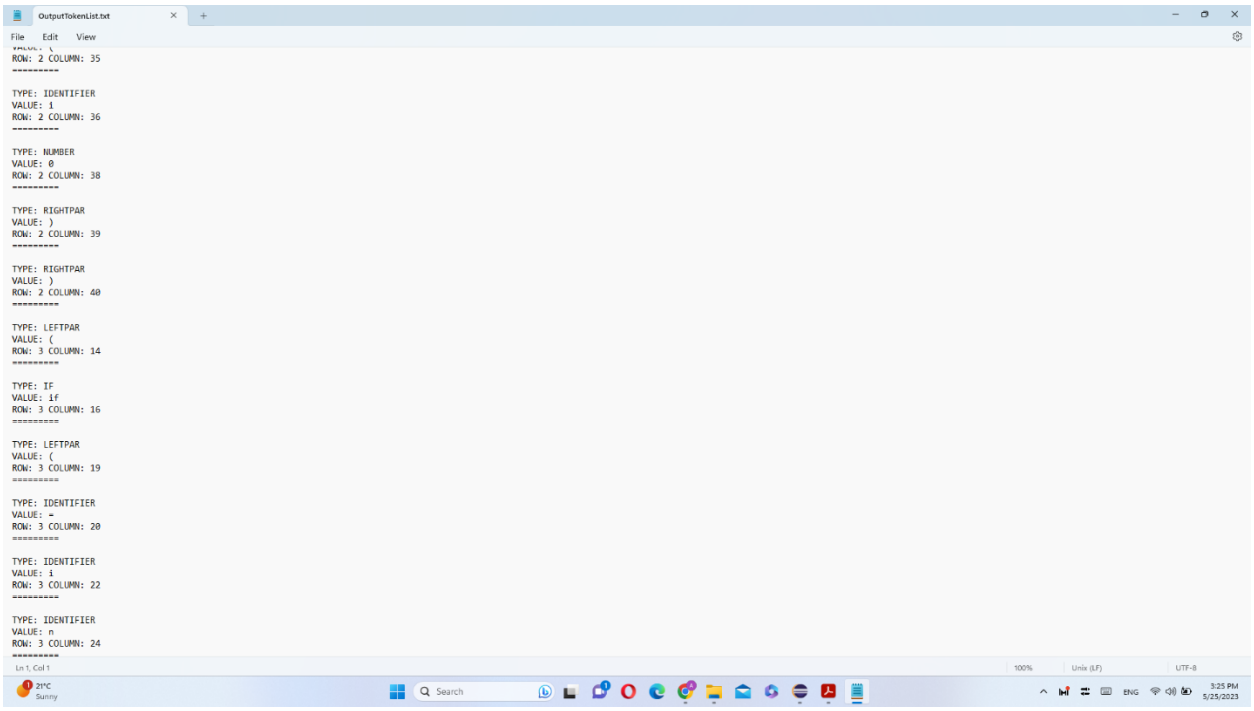

Output.txt
File Edit View
<Program>
<TopLevelForm>
LEFTPAR (()
<SecondLevelForm>
<Definition>
DEFINE (define)
<DefinitionRight>
LEFTPAR (()
IDENTIFIER (fibonacci)
<ArgList>
IDENTIFIER (n)
<ArgList>
RIGHTPAR ())
<Statements>
<Expression>
LEFTPAR (()
<Expr>
<LetExpression>
LET (let)
<LetExpr>
IDENTIFIER (fib)
LEFTPAR (()
<VarDefs>
LEFTPAR (()
IDENTIFIER (prev)
<Expression>
NUMBER (0)
RIGHTPAR ())
<VarDef>
<VarDefs>
LEFTPAR (()
IDENTIFIER (cur)
<Expression>
NUMBER (1)
RIGHTPAR ())
<VarDef>
<VarDefs>
LEFTPAR (()
IDENTIFIER (i)
<Expression>
NUMBER (0)
RIGHTPAR ())
<VarDef>
RIGHTPAR ())
<Statements>
<Expression>
LEFTPAR (()
<Expr>
<IfExpression>
Ln 1, Col 1
100%
Unix (LF)
UTF-8
21°C Sunny
Search
ENG
3:24 PM 5/25/2023

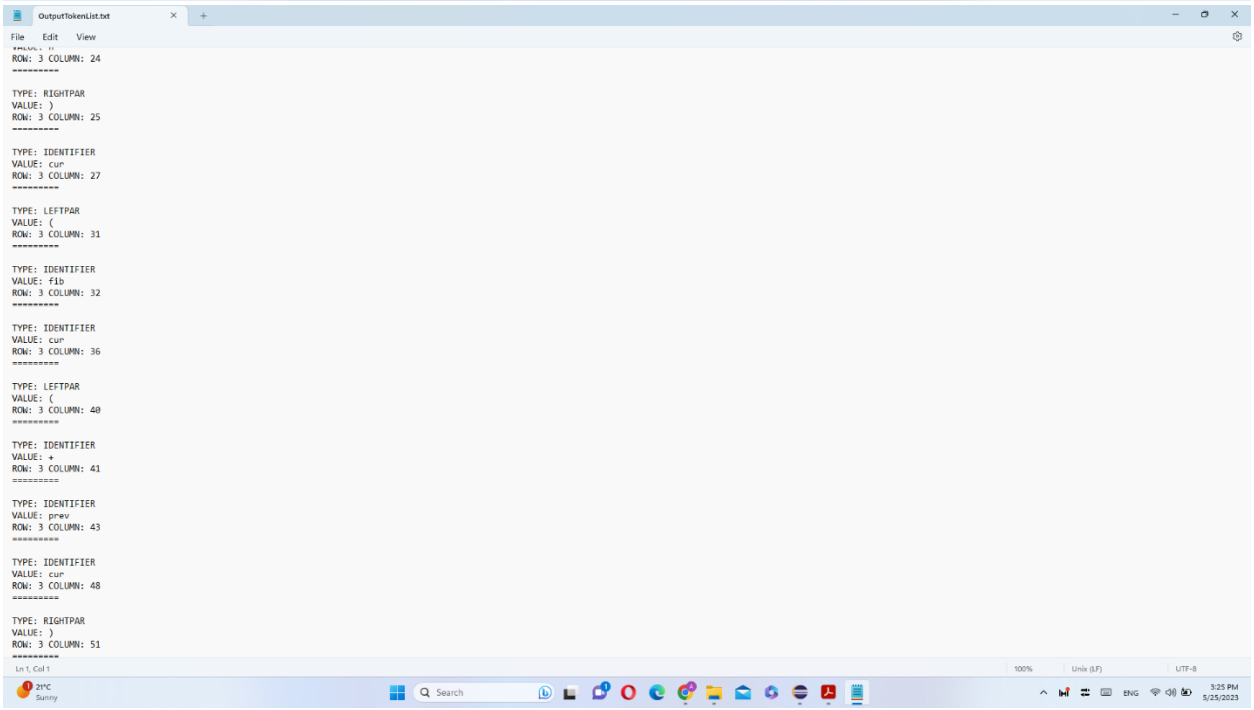

Output.txt
File Edit View
<Statements>
<Expression>
LEFTPAR (()
<Expr>
<IfExpression>
IF (if)
<Expression>
LEFTPAR (()
<Expr>
<FunCall>
IDENTIFIER (=)
<Expressions>
<Expression>
IDENTIFIER (i)
<Expressions>
<Expression>
IDENTIFIER (n)
<Expressions>
RIGHTPAR ())
<Expression>
IDENTIFIER (cur)
<EndExpression>
<Expression>
LEFTPAR (()
<Expr>
<FunCall>
IDENTIFIER (fib)
<Expressions>
<Expression>
IDENTIFIER (cur)
<Expressions>
<Expression>
LEFTPAR (()
<Expr>
<FunCall>
IDENTIFIER (+)
<Expressions>
<Expression>
IDENTIFIER (prev)
<Expressions>
<Expression>
IDENTIFIER (cur)
<Expressions>
RIGHTPAR ())
<Expressions>
<Expression>
LEFTPAR (()
<Expr>
<FunCall>
IDENTIFIER (+)
Ln 1, Col 1
100%
Unix (LF)
UTF-8
21°C
Sunny
Search
ENG
3:24 PM
5/25/2023

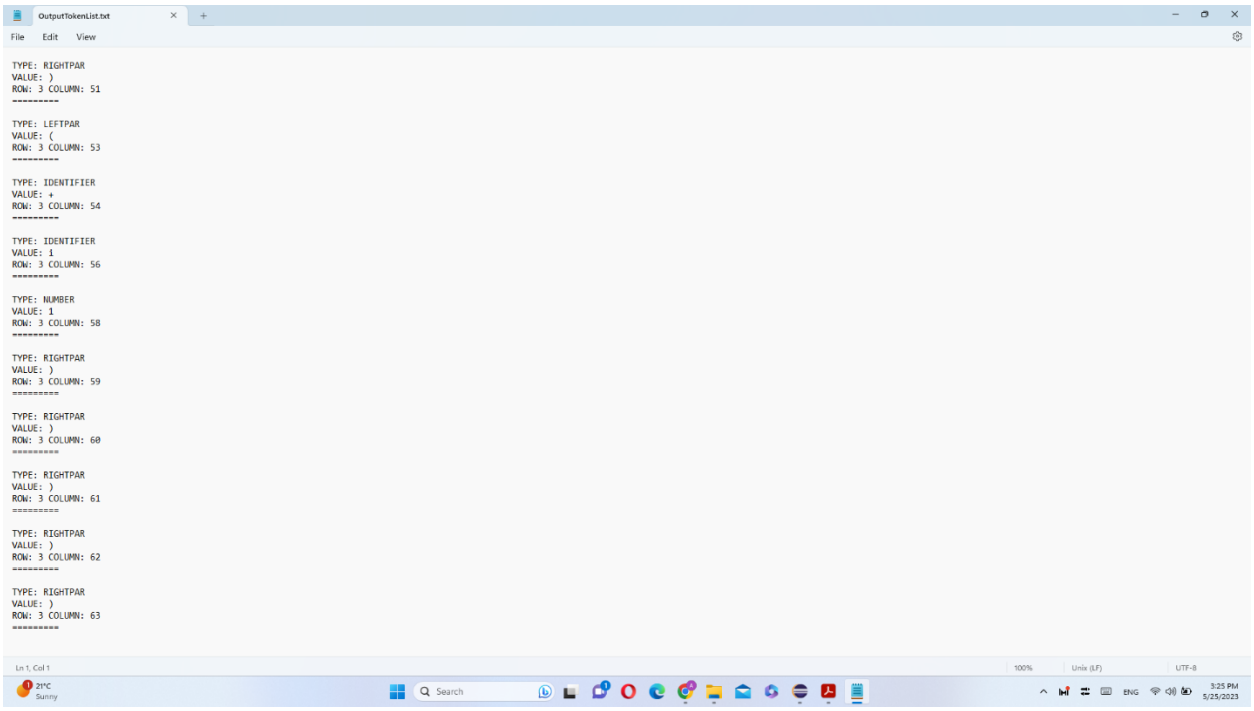

Output.txt
File Edit View
RIGHTPAR ())
<Expression>
IDENTIFIER (cur)
<EndExpression>
<Expression>
LEFTPAR (()
<Expr>
<FunCall>
IDENTIFIER (fib)
<Expressions>
<Expression>
IDENTIFIER (cur)
<Expressions>
<Expression>
LEFTPAR (()
<Expr>
<FunCall>
IDENTIFIER (+)
<Expressions>
<Expression>
IDENTIFIER (prev)
<Expressions>
<Expression>
IDENTIFIER (cur)
<Expressions>
RIGHTPAR ())
<Expressions>
<Expression>
LEFTPAR (()
<Expr>
<FunCall>
IDENTIFIER (+)
<Expressions>
<Expression>
IDENTIFIER (i)
<Expressions>
<Expression>
NUMBER (1)
<Expressions>
RIGHTPAR ())
<Expressions>
RIGHTPAR ())
RIGHTPAR ())
RIGHTPAR ())
<Program>
Ln 1, Col 1
100%
Unix (LF)
UTF-8
21°C
Sunny
Search
ENG
3:24 PM
5/25/2023


OutputTokenList.txt
File Edit View
TYPE: LEFTPAR
VALUE: (
ROW: 1 COLUMN: 1
=========
TYPE: DEFINE
VALUE: define
ROW: 1 COLUMN: 2
=========
TYPE: LEFTPAR
VALUE: (
ROW: 1 COLUMN: 9
=========
TYPE: IDENTIFIER
VALUE: fibonacci
ROW: 1 COLUMN: 10
=========
TYPE: IDENTIFIER
VALUE: n
ROW: 1 COLUMN: 20
=========
TYPE: RIGHTPAR
VALUE:)
ROW: 1 COLUMN: 21
=========
TYPE: LEFTPAR
VALUE: (
ROW: 2 COLUMN: 7
=========
TYPE: LET
VALUE: let
ROW: 2 COLUMN: 9
=========
TYPE: IDENTIFIER
VALUE: fib
ROW: 2 COLUMN: 13
=========
TYPE: LEFTPAR
VALUE: (
ROW: 2 COLUMN: 17
=========
TYPE: LEFTPAR
Ln 1, Col 1
100%
Unix (LF)
UTF-8
21°C
Sunny
Search
ENG
3:24 PM
5/25/2023

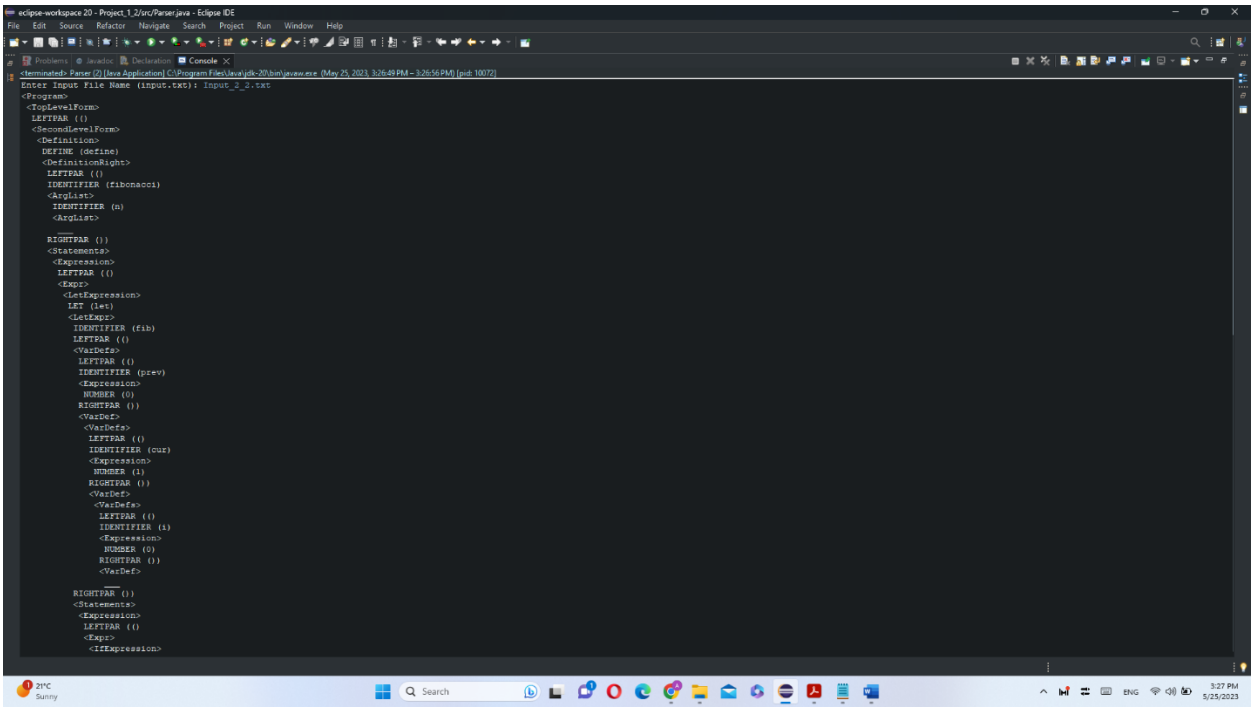

OutputTokenList.txt
File Edit View
=========
TYPE: LEFTPAR
VALUE: (
ROW: 2 COLUMN: 18
=========
TYPE: IDENTIFIER
VALUE: prev
ROW: 2 COLUMN: 19
=========
TYPE: NUMBER
VALUE: 0
ROW: 2 COLUMN: 24
=========
TYPE: RIGHTPAR
VALUE:)
ROW: 2 COLUMN: 25
=========
TYPE: LEFTPAR
VALUE: (
ROW: 2 COLUMN: 27
=========
TYPE: IDENTIFIER
VALUE: cur
ROW: 2 COLUMN: 28
=========
TYPE: NUMBER
VALUE: 1
ROW: 2 COLUMN: 32
=========
TYPE: RIGHTPAR
VALUE:)
ROW: 2 COLUMN: 33
=========
TYPE: LEFTPAR
VALUE: (
ROW: 2 COLUMN: 35
=========
TYPE: IDENTIFIER
VALUE: i
ROW: 2 COLUMN: 36
=========
Ln 1, Col 1
100%
Unix (LF)
UTF-8
21°C
Sunny
Search
ENG
3:25 PM
5/25/2023

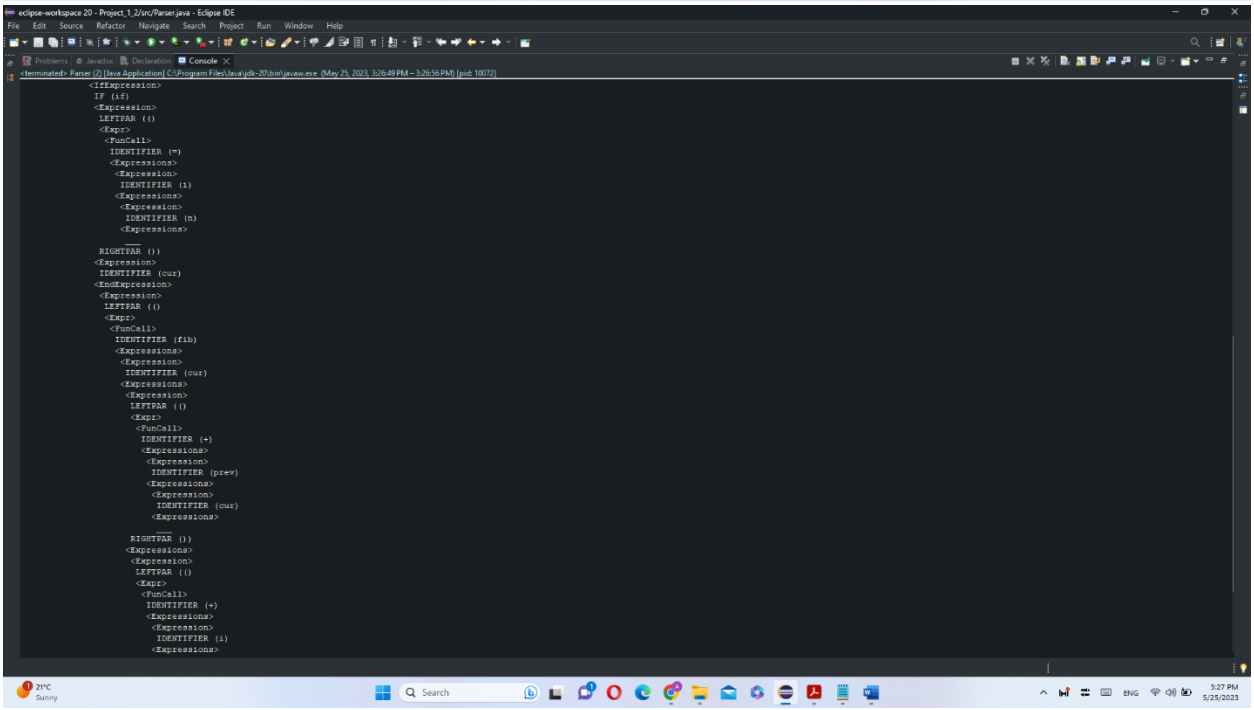

OutputTokenList.txt
File Edit View
ROW: 2 COLUMN: 35
TYPE: IDENTIFIER
VALUE: i
ROW: 2 COLUMN: 36
TYPE: NUMBER
VALUE: 0
ROW: 2 COLUMN: 38
TYPE: RIGHTPAR
VALUE:)
ROW: 2 COLUMN: 39
TYPE: RIGHTPAR
VALUE:)
ROW: 2 COLUMN: 40
TYPE: LEFTPAR
VALUE: (
ROW: 3 COLUMN: 14
TYPE: IF
VALUE: if
ROW: 3 COLUMN: 16
TYPE: LEFTPAR
VALUE: (
ROW: 3 COLUMN: 19
TYPE: IDENTIFIER
VALUE: -
ROW: 3 COLUMN: 20
TYPE: IDENTIFIER
VALUE: i
ROW: 3 COLUMN: 22
TYPE: IDENTIFIER
VALUE: n
ROW: 3 COLUMN: 24
Ln 1, Col 1
100%
Unix (LF)
UTF-8
21°C
Sunny
Search
ENG
3:25 PM
5/25/2023

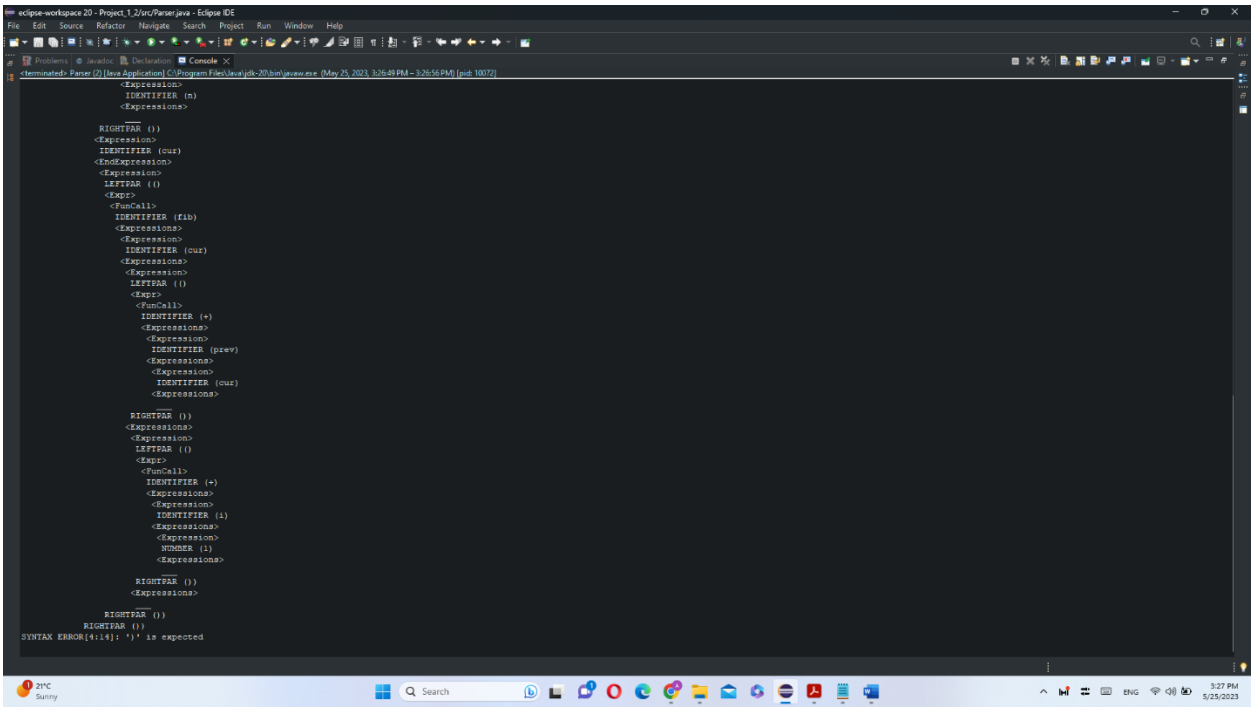

OutputTokenList.txt
File Edit View
ROW: 3 COLUMN: 24
TYPE: RIGHTPAR
VALUE:)
ROW: 3 COLUMN: 25
TYPE: IDENTIFIER
VALUE: cur
ROW: 3 COLUMN: 27
TYPE: LEFTPAR
VALUE: (
ROW: 3 COLUMN: 31
TYPE: IDENTIFIER
VALUE: fib
ROW: 3 COLUMN: 32
TYPE: IDENTIFIER
VALUE: cur
ROW: 3 COLUMN: 36
TYPE: LEFTPAR
VALUE: (
ROW: 3 COLUMN: 40
TYPE: IDENTIFIER
VALUE: +
ROW: 3 COLUMN: 41
TYPE: IDENTIFIER
VALUE: prev
ROW: 3 COLUMN: 43
TYPE: IDENTIFIER
VALUE: cur
ROW: 3 COLUMN: 48
TYPE: RIGHTPAR
VALUE:)
ROW: 3 COLUMN: 51
Ln 1, Col 1
100%
Unix (LF)
UTF-8
21°C
Sunny
Search
ENG
3:25 PM
5/25/2023

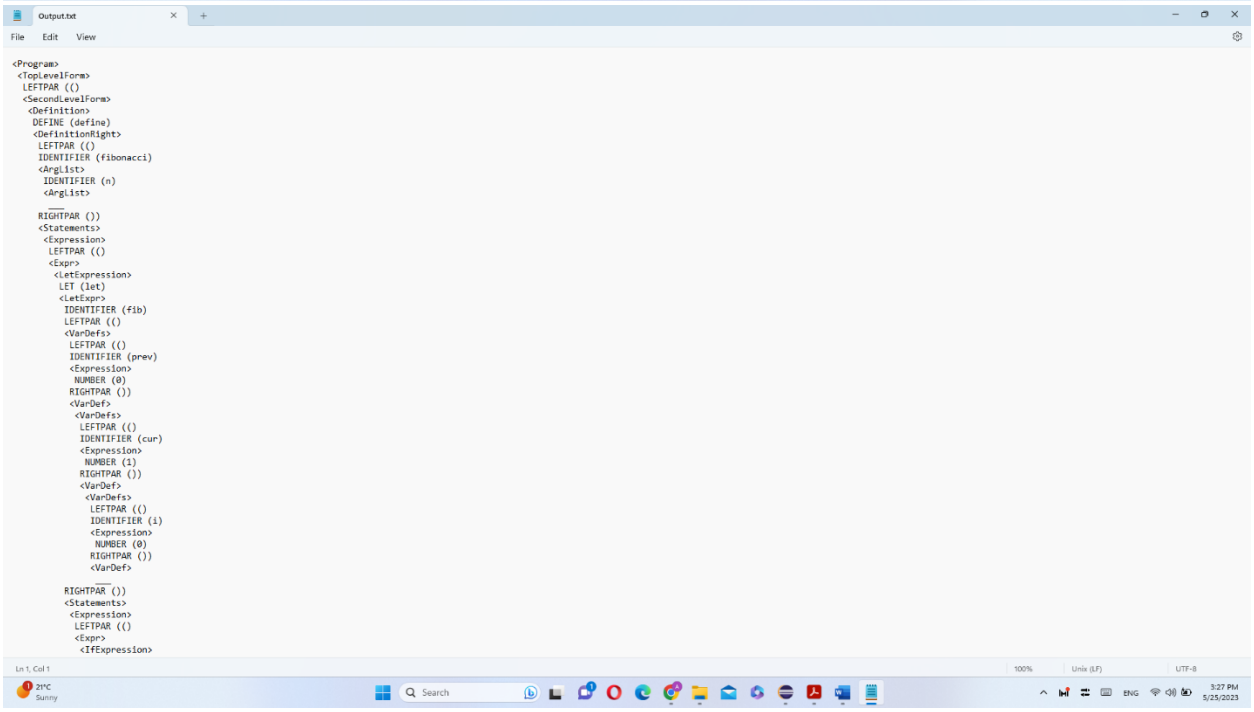


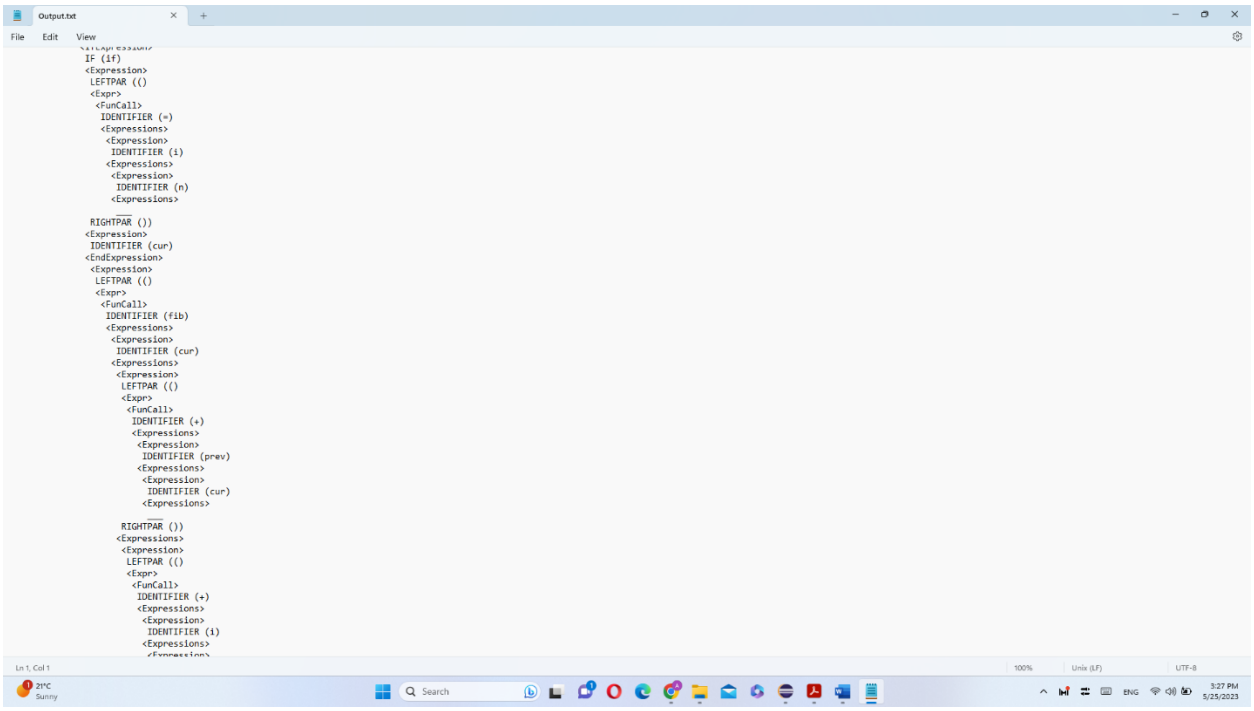
Input File (4): Input_2_2.txt

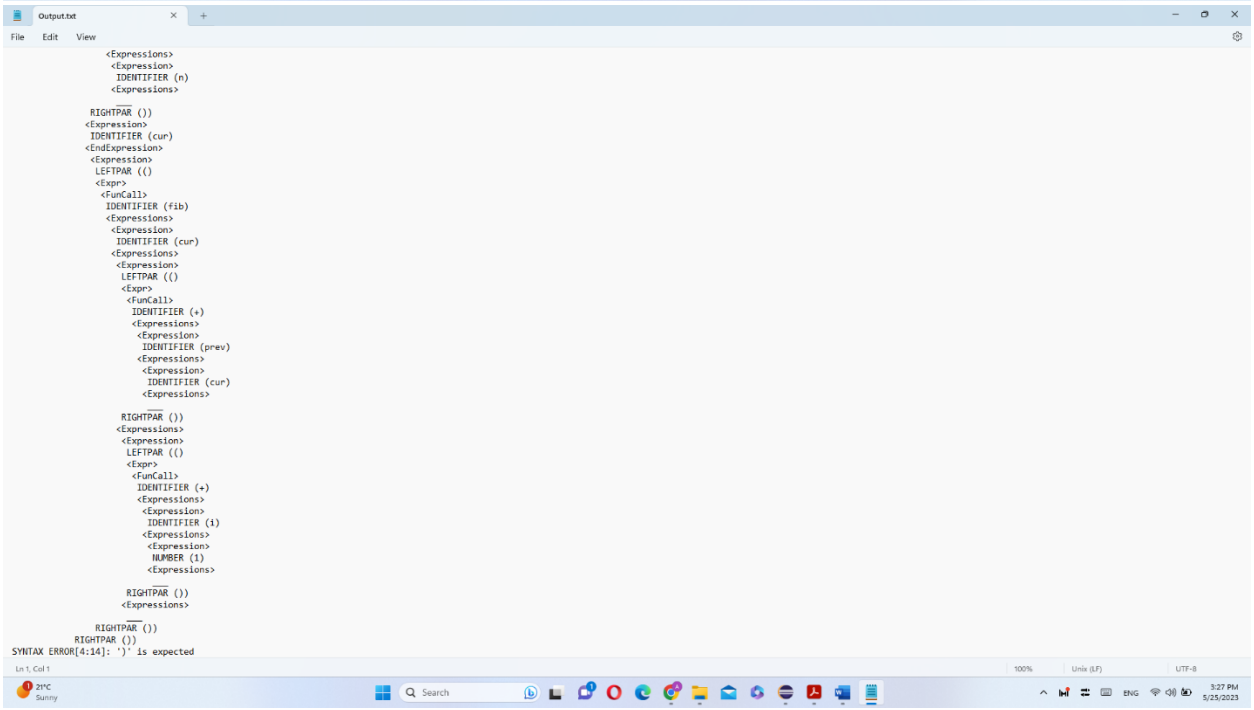

eclipse-workspace 20 - Project_1_2/src/Parser.java - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help
Problems Javadoc Declaration Console
<terminated> Parser (2) [Java Application] C:\Program Files\Java\jdk-20\bin\javaw.exe (May 25, 2023, 3:26:49 PM – 3:26:56 PM) [pid: 10072]
Enter Input File Name (input.txt): Input_2_2.txt
<Program>
<TopLevelForm>
LEFTPAR (()
<SecondLevelForm>
<Definition>
DEFINE (define)
<DefinitionRight>
LEFTPAR (()
IDENTIFIER (fibonacci)
<ArgList>
IDENTIFIER (n)
<ArgList>
RIGHTPAR ())
<Statements>
<Expression>
LEFTPAR (()
<Expr>
<LetExpression>
LET (let)
<LetExpr>
IDENTIFIER (fib)
LEFTPAR (()
<VarDefs>
LEFTPAR (()
IDENTIFIER (prev)
<Expression>
NUMBER (0)
RIGHTPAR ())
<VarDef>
<VarDefs>
LEFTPAR (()
IDENTIFIER (cur)
<Expression>
NUMBER (1)
RIGHTPAR ())
<VarDef>
<VarDefs>
LEFTPAR (()
IDENTIFIER (i)
<Expression>
NUMBER (0)
RIGHTPAR ())
<VarDef>
RIGHTPAR ())
<Statements>
<Expression>
LEFTPAR (()
<Expr>
<IfExpression>
21°C Sunny
Search
ENG
3:27 PM 5/25/2023

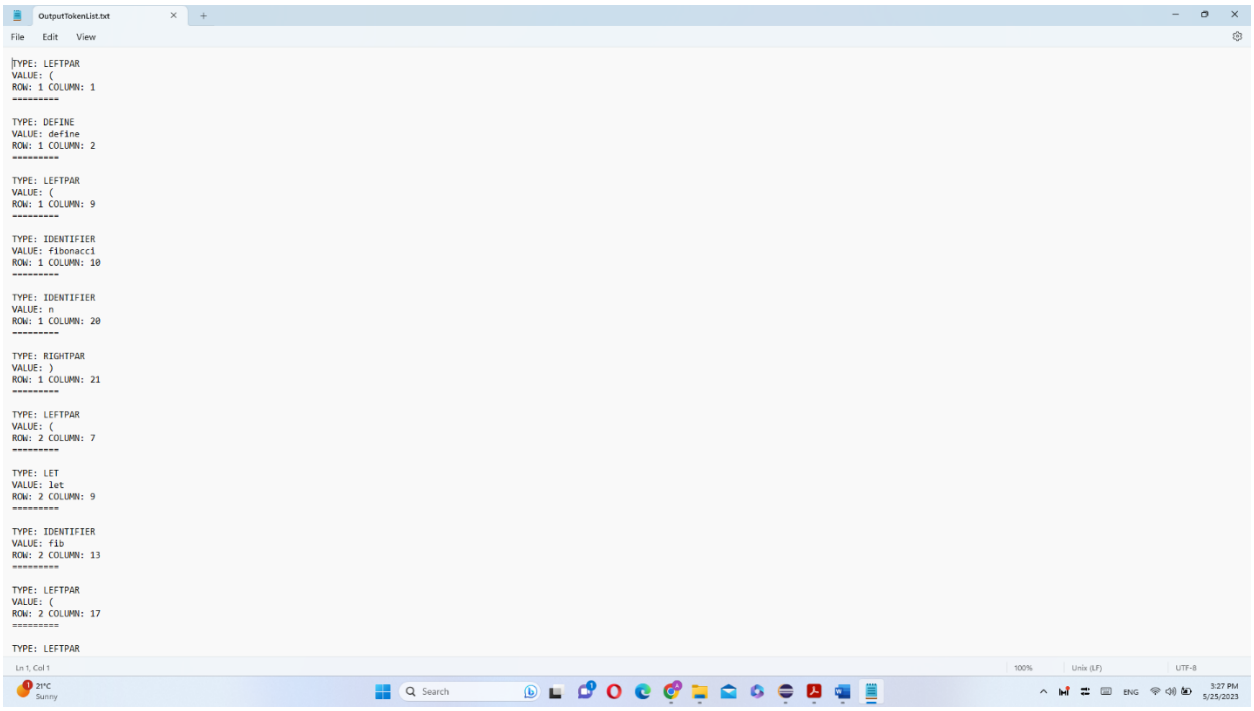

eclipse-workspace 20 - Project_1_2/src/Parser.java - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help
Problems Javadoc Declaration Console
<terminated> Parser (2) [Java Application] C:\Program Files\Java\jdk-20\bin\javaw.exe (May 25, 2023, 3:26:49 PM – 3:26:56 PM) [pid: 10072]
<IfExpression>
IF (if)
<Expression>
LEFTPAR (()
<Expr>
<FunCall>
IDENTIFIER (=)
<Expressions>
<Expression>
IDENTIFIER (i)
<Expressions>
<Expression>
IDENTIFIER (n)
<Expressions>
RIGHTPAR ())
<Expression>
IDENTIFIER (cur)
<EndExpression>
<Expression>
LEFTPAR (()
<Expr>
<FunCall>
IDENTIFIER (fib)
<Expressions>
<Expression>
IDENTIFIER (cur)
<Expressions>
<Expression>
LEFTPAR (()
<Expr>
<FunCall>
IDENTIFIER (+)
<Expressions>
<Expression>
IDENTIFIER (prev)
<Expressions>
<Expression>
IDENTIFIER (cur)
<Expressions>
RIGHTPAR ())
<Expressions>
<Expression>
LEFTPAR (()
<Expr>
<FunCall>
IDENTIFIER (+)
<Expressions>
<Expression>
IDENTIFIER (i)
<Expressions>
21°C Sunny
Search
ENG
3:27 PM 5/25/2023

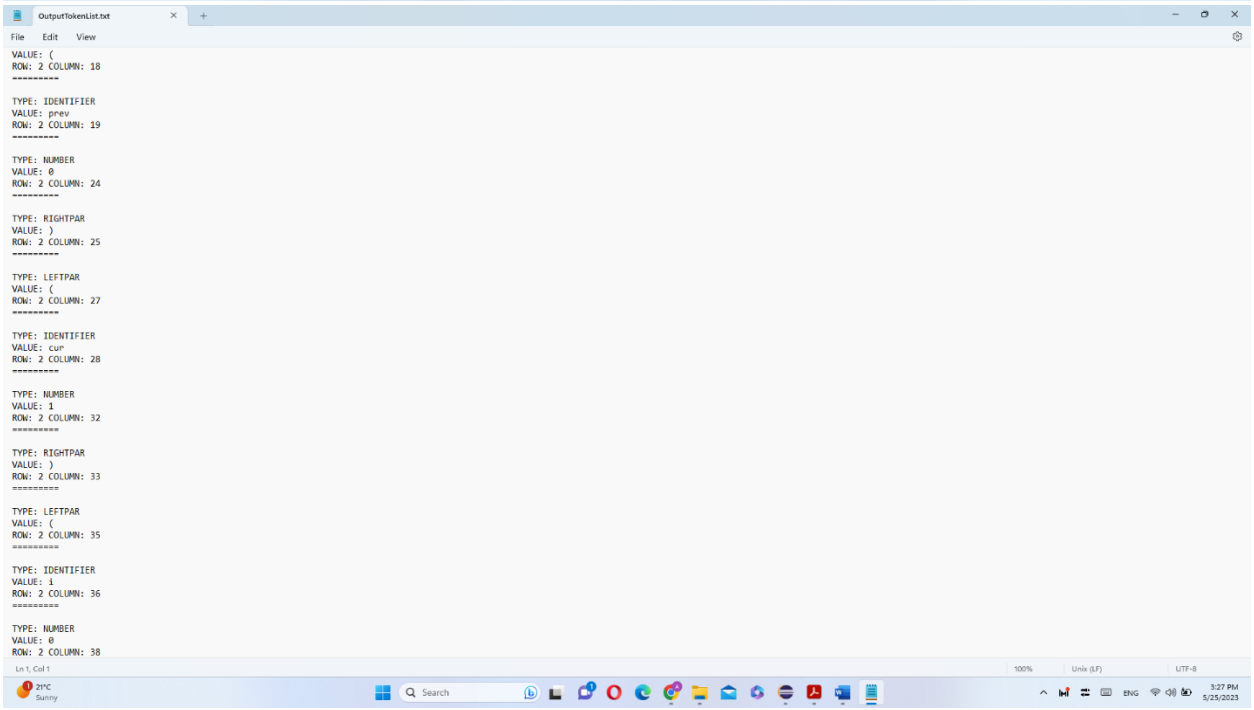

eclipse-workspace 20 - Project_1_2/src/Parser.java - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help
Problems Javadoc Declaration Console
<terminated> Parser (2) [Java Application] C:\Program Files\Java\jdk-20\bin\javaw.exe (May 25, 2023, 3:26:49 PM – 3:26:56 PM) [pid: 10072]
<Expression>
IDENTIFIER (n)
<Expressions>
RIGHTPAR ())
<Expression>
IDENTIFIER (cur)
<EndExpression>
<Expression>
LEFTPAR (()
<Expr>
<FunCall>
IDENTIFIER (fib)
<Expressions>
<Expression>
IDENTIFIER (cur)
<Expressions>
<Expression>
LEFTPAR (()
<Expr>
<FunCall>
IDENTIFIER (+)
<Expressions>
<Expression>
IDENTIFIER (prev)
<Expressions>
<Expression>
IDENTIFIER (cur)
<Expressions>
RIGHTPAR ())
<Expressions>
<Expression>
LEFTPAR (()
<Expr>
<FunCall>
IDENTIFIER (+)
<Expressions>
<Expression>
IDENTIFIER (i)
<Expressions>
<Expression>
NUMBER (1)
<Expressions>
RIGHTPAR ())
<Expressions>
RIGHTPAR ())
RIGHTPAR ())
SYNTAX ERROR[4:14]: ')' is expected
21°C Sunny
Search
ENG
3:27 PM 5/25/2023

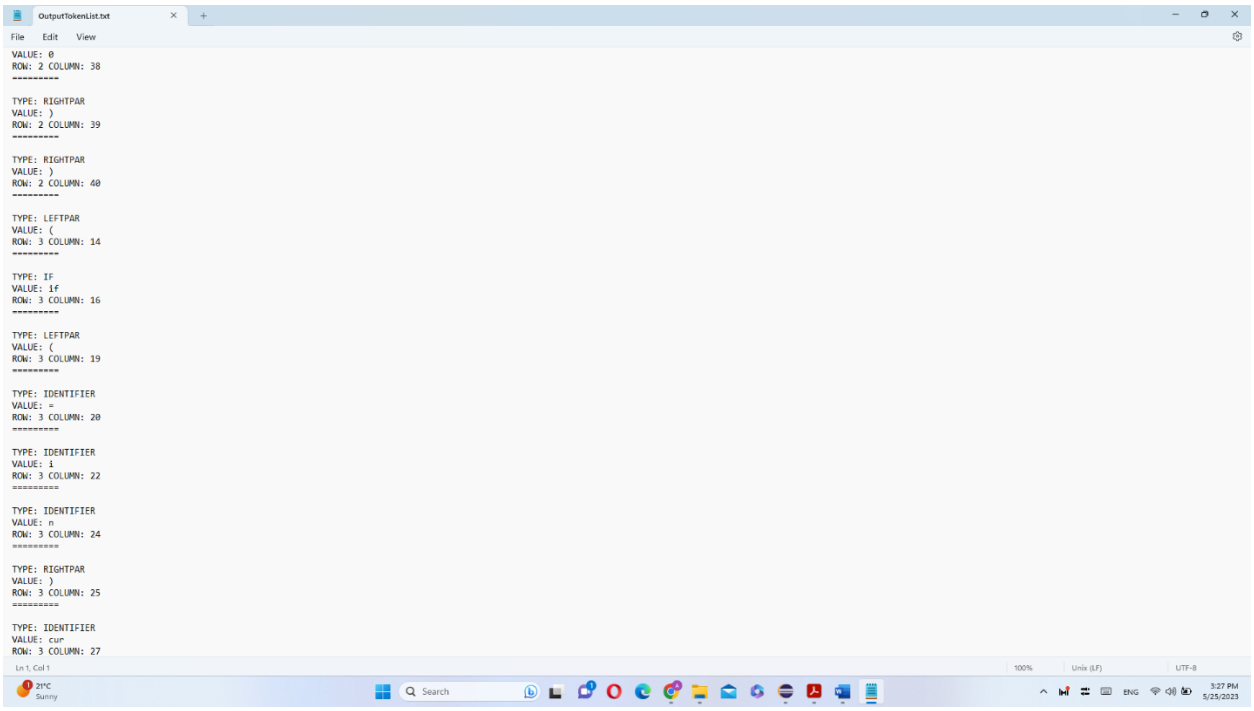

Output.txt
File Edit View
<Program>
<TopLevelForm>
LEFTPAR (()
<SecondLevelForm>
<Definition>
DEFINE (define)
<DefinitionRight>
LEFTPAR (()
IDENTIFIER (fibonacci)
<ArgList>
IDENTIFIER (n)
<ArgList>
RIGHTPAR ())
<Statements>
<Expression>
LEFTPAR (()
<Expr>
<LetExpression>
LET (let)
<LetExpr>
IDENTIFIER (fib)
LEFTPAR (()
<VarDefs>
LEFTPAR (()
IDENTIFIER (prev)
<Expression>
NUMBER (0)
RIGHTPAR ())
<VarDef>
<VarDefs>
LEFTPAR (()
IDENTIFIER (cur)
<Expression>
NUMBER (1)
RIGHTPAR ())
<VarDef>
<VarDefs>
LEFTPAR (()
IDENTIFIER (i)
<Expression>
NUMBER (0)
RIGHTPAR ())
<VarDef>
RIGHTPAR ())
<Statements>
<Expression>
LEFTPAR (()
<Expr>
<IfExpression>
Ln 1, Col 1
100%
Unix (LF)
UTF-8
21°C Sunny
Search
ENG
3:27 PM 5/25/2023


Output.txt
File Edit View
IF (if)
<Expression>
LEFTPAR (()
<Expr>
<FunCall>
IDENTIFIER (=)
<Expressions>
<Expression>
IDENTIFIER (i)
<Expressions>
<Expression>
IDENTIFIER (n)
<Expressions>
RIGHTPAR ())
<Expression>
IDENTIFIER (cur)
<EndExpression>
<Expression>
LEFTPAR (()
<Expr>
<FunCall>
IDENTIFIER (fib)
<Expressions>
<Expression>
IDENTIFIER (cur)
<Expressions>
<Expression>
LEFTPAR (()
<Expr>
<FunCall>
IDENTIFIER (+)
<Expressions>
<Expression>
IDENTIFIER (prev)
<Expressions>
<Expression>
IDENTIFIER (cur)
<Expressions>
RIGHTPAR ())
<Expressions>
<Expression>
LEFTPAR (()
<Expr>
<FunCall>
IDENTIFIER (+)
<Expressions>
<Expression>
IDENTIFIER (i)
<Expressions>
Ln 1, Col 1
100%
Unix (LF)
UTF-8
21°C
Sunny
Search
ENG
3:27 PM
5/25/2023


Output.txt
File Edit View
<Expressions>
<Expression>
IDENTIFIER (n)
<Expressions>
RIGHTPAR ())
<Expression>
IDENTIFIER (cur)
<EndExpression>
<Expression>
LEFTPAR (()
<Expr>
<FunCall>
IDENTIFIER (fib)
<Expressions>
<Expression>
IDENTIFIER (cur)
<Expressions>
<Expression>
LEFTPAR (()
<Expr>
<FunCall>
IDENTIFIER (+)
<Expressions>
<Expression>
IDENTIFIER (prev)
<Expressions>
<Expression>
IDENTIFIER (cur)
<Expressions>
RIGHTPAR ())
<Expressions>
<Expression>
LEFTPAR (()
<Expr>
<FunCall>
IDENTIFIER (+)
<Expressions>
<Expression>
IDENTIFIER (i)
<Expressions>
<Expression>
NUMBER (1)
<Expressions>
RIGHTPAR ())
<Expressions>
RIGHTPAR ())
RIGHTPAR ())
SYNTAX ERROR[4:14]: ')' is expected
Ln 1, Col 1
100%
Unix (LF)
UTF-8
21°C
Sunny
Search
ENG
3:27 PM
5/25/2023


OutputTokenList.txt
File Edit View
TYPE: LEFTPAR
VALUE: (
ROW: 1 COLUMN: 1
=========
TYPE: DEFINE
VALUE: define
ROW: 1 COLUMN: 2
=========
TYPE: LEFTPAR
VALUE: (
ROW: 1 COLUMN: 9
=========
TYPE: IDENTIFIER
VALUE: fibonacci
ROW: 1 COLUMN: 10
=========
TYPE: IDENTIFIER
VALUE: n
ROW: 1 COLUMN: 20
=========
TYPE: RIGHTPAR
VALUE:)
ROW: 1 COLUMN: 21
=========
TYPE: LEFTPAR
VALUE: (
ROW: 2 COLUMN: 7
=========
TYPE: LET
VALUE: let
ROW: 2 COLUMN: 9
=========
TYPE: IDENTIFIER
VALUE: fib
ROW: 2 COLUMN: 13
=========
TYPE: LEFTPAR
VALUE: (
ROW: 2 COLUMN: 17
=========
TYPE: LEFTPAR
Ln 1, Col 1
100%
Unix (LF)
UTF-8
21°C
Sunny
Search
ENG
3:27 PM
5/25/2023


OutputTokenList.txt
File Edit View
VALUE: (
ROW: 2 COLUMN: 18
=========
TYPE: IDENTIFIER
VALUE: prev
ROW: 2 COLUMN: 19
=========
TYPE: NUMBER
VALUE: 0
ROW: 2 COLUMN: 24
=========
TYPE: RIGHTPAR
VALUE:)
ROW: 2 COLUMN: 25
=========
TYPE: LEFTPAR
VALUE: (
ROW: 2 COLUMN: 27
=========
TYPE: IDENTIFIER
VALUE: cur
ROW: 2 COLUMN: 28
=========
TYPE: NUMBER
VALUE: 1
ROW: 2 COLUMN: 32
=========
TYPE: RIGHTPAR
VALUE:)
ROW: 2 COLUMN: 33
=========
TYPE: LEFTPAR
VALUE: (
ROW: 2 COLUMN: 35
=========
TYPE: IDENTIFIER
VALUE: i
ROW: 2 COLUMN: 36
=========
TYPE: NUMBER
VALUE: 0
ROW: 2 COLUMN: 38
Ln 1, Col 1
100%
Unix (LF)
UTF-8
21°C
Sunny
Search
ENG
3:27 PM
5/25/2023



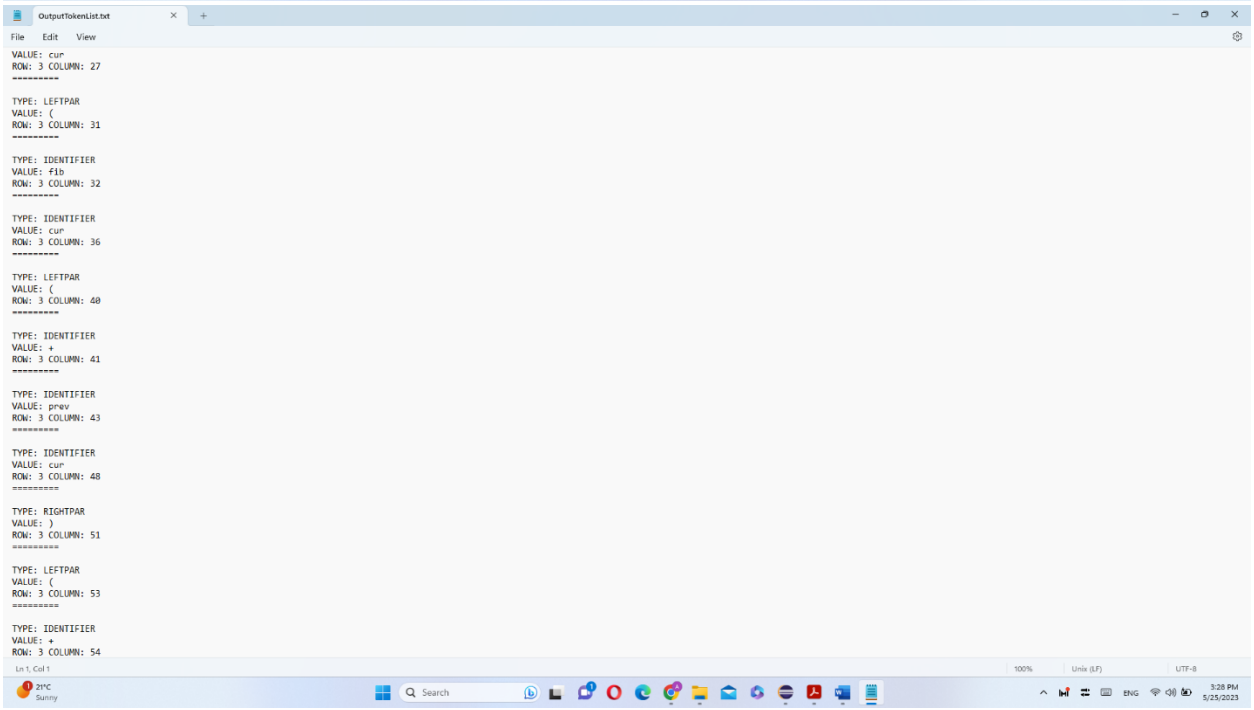
OutputTokenList.txt

File Edit View

```
VALUE: 0
ROW: 2 COLUMN: 38
=========

TYPE: RIGHTPAR
VALUE: )
ROW: 2 COLUMN: 39
=========

TYPE: RIGHTPAR
VALUE: )
ROW: 2 COLUMN: 40
=========

TYPE: LEFTPAR
VALUE: (
ROW: 3 COLUMN: 14
=========

TYPE: IF
VALUE: if
ROW: 3 COLUMN: 16
=========

TYPE: LEFTPAR
VALUE: (
ROW: 3 COLUMN: 19
=========

TYPE: IDENTIFIER
VALUE: =
ROW: 3 COLUMN: 20
=========

TYPE: IDENTIFIER
VALUE: i
ROW: 3 COLUMN: 22
=========

TYPE: IDENTIFIER
VALUE: n
ROW: 3 COLUMN: 24
=========

TYPE: RIGHTPAR
VALUE: )
ROW: 3 COLUMN: 25
=========

TYPE: IDENTIFIER
VALUE: cur
ROW: 3 COLUMN: 27
```

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

21°C Sunny Search ENG 3:27 PM 5/25/2023



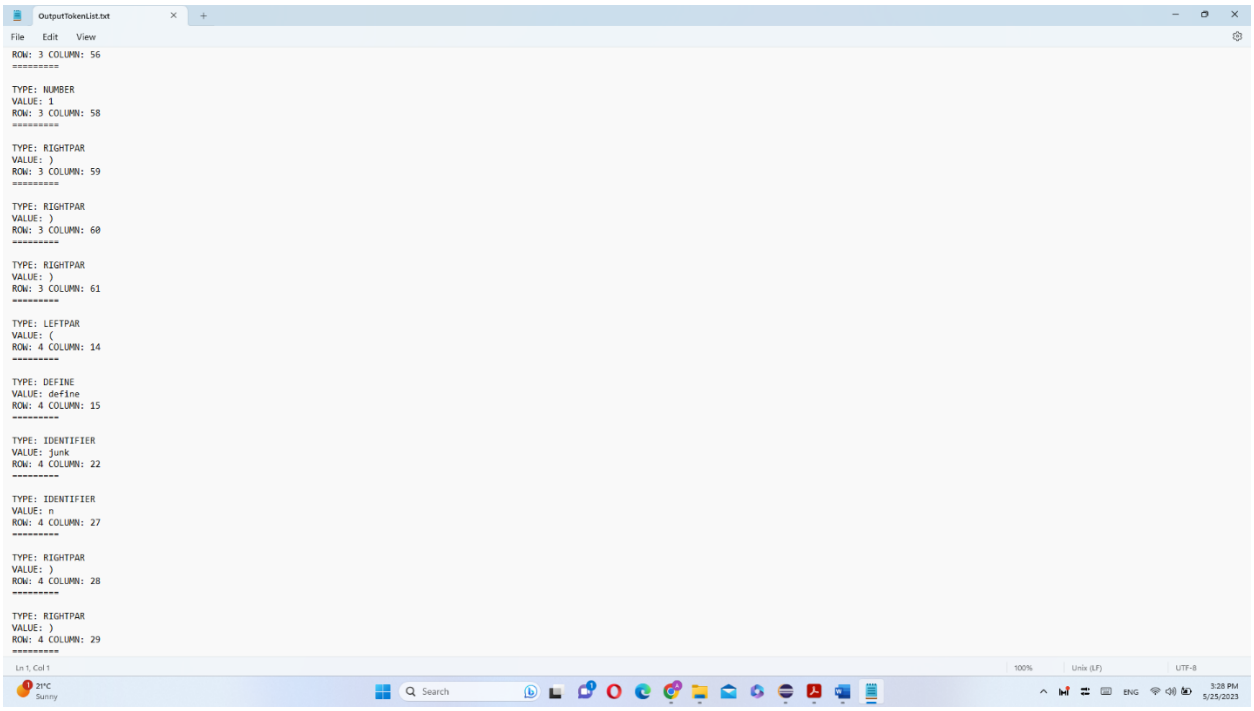
OutputTokenList.txt

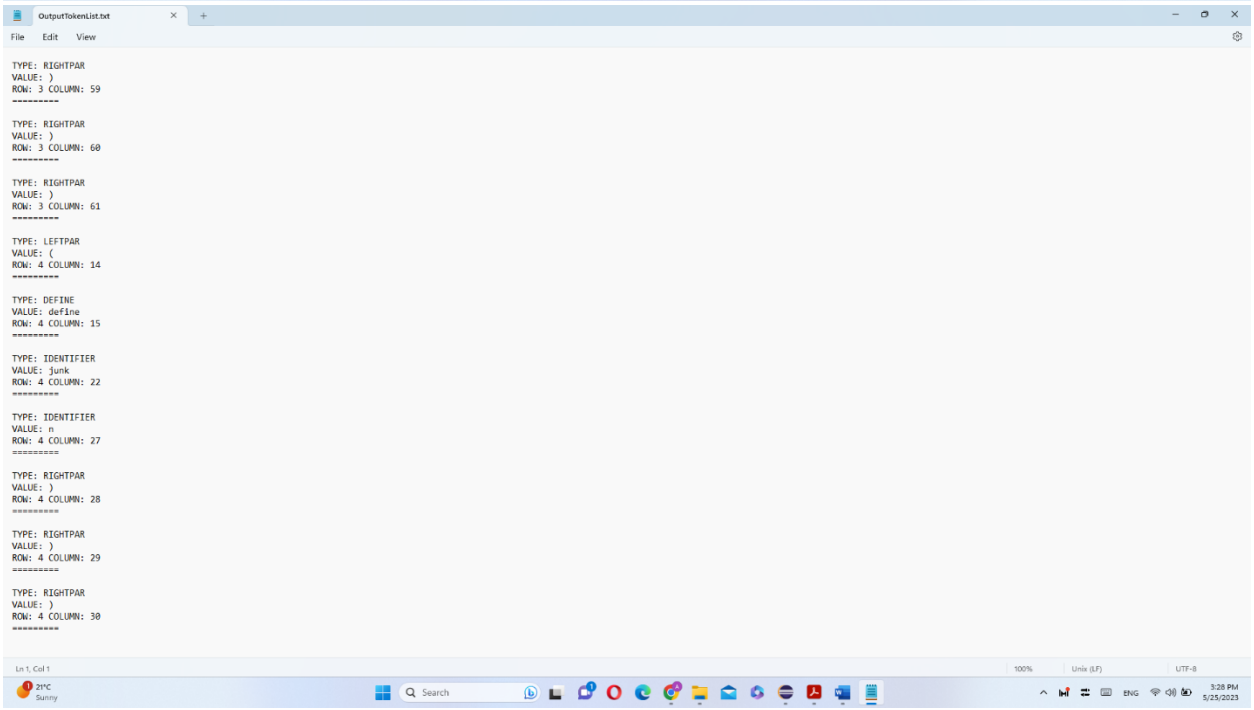
File Edit View

```
VALUE: cur
ROW: 3 COLUMN: 27
=========

TYPE: LEFTPAR
VALUE: (
ROW: 3 COLUMN: 31
=========

TYPE: IDENTIFIER
VALUE: fib
ROW: 3 COLUMN: 32
=========

TYPE: IDENTIFIER
VALUE: cur
ROW: 3 COLUMN: 36
=========

TYPE: LEFTPAR
VALUE: (
ROW: 3 COLUMN: 40
=========

TYPE: IDENTIFIER
VALUE: +
ROW: 3 COLUMN: 41
=========

TYPE: IDENTIFIER
VALUE: prev
ROW: 3 COLUMN: 43
=========

TYPE: IDENTIFIER
VALUE: cur
ROW: 3 COLUMN: 48
=========

TYPE: RIGHTPAR
VALUE: )
ROW: 3 COLUMN: 51
=========

TYPE: LEFTPAR
VALUE: (
ROW: 3 COLUMN: 53
=========

TYPE: IDENTIFIER
VALUE: +
ROW: 3 COLUMN: 54
```

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

21°C Sunny Search ENG 3:28 PM 5/25/2023


OutputTokenList.txt
File Edit View
ROW: 3 COLUMN: 56
=========
TYPE: NUMBER
VALUE: 1
ROW: 3 COLUMN: 58
=========
TYPE: RIGHTPAR
VALUE:)
ROW: 3 COLUMN: 59
=========
TYPE: RIGHTPAR
VALUE:)
ROW: 3 COLUMN: 60
=========
TYPE: RIGHTPAR
VALUE:)
ROW: 3 COLUMN: 61
=========
TYPE: LEFTPAR
VALUE: (
ROW: 4 COLUMN: 14
=========
TYPE: DEFINE
VALUE: define
ROW: 4 COLUMN: 15
=========
TYPE: IDENTIFIER
VALUE: junk
ROW: 4 COLUMN: 22
=========
TYPE: IDENTIFIER
VALUE: n
ROW: 4 COLUMN: 27
=========
TYPE: RIGHTPAR
VALUE:)
ROW: 4 COLUMN: 28
=========
TYPE: RIGHTPAR
VALUE:)
ROW: 4 COLUMN: 29
=========
Ln 1, Col 1
100%
Unix (LF)
UTF-8
21°C
Sunny
Search
ENG
3:28 PM
5/25/2023


OutputTokenList.txt
File Edit View
TYPE: RIGHTPAR
VALUE:)
ROW: 3 COLUMN: 59
=========
TYPE: RIGHTPAR
VALUE:)
ROW: 3 COLUMN: 60
=========
TYPE: RIGHTPAR
VALUE:)
ROW: 3 COLUMN: 61
=========
TYPE: LEFTPAR
VALUE: (
ROW: 4 COLUMN: 14
=========
TYPE: DEFINE
VALUE: define
ROW: 4 COLUMN: 15
=========
TYPE: IDENTIFIER
VALUE: junk
ROW: 4 COLUMN: 22
=========
TYPE: IDENTIFIER
VALUE: n
ROW: 4 COLUMN: 27
=========
TYPE: RIGHTPAR
VALUE:)
ROW: 4 COLUMN: 28
=========
TYPE: RIGHTPAR
VALUE:)
ROW: 4 COLUMN: 29
=========
TYPE: RIGHTPAR
VALUE:)
ROW: 4 COLUMN: 30
=========
Ln 1, Col 1
100%
Unix (LF)
UTF-8
21°C
Sunny
Search
ENG
3:28 PM
5/25/2023