Ministerul Educaţiei și Cercetării al Republicii Moldova Universitatea Tehnică a Moldovei

Facultatea Calculatoare, Informatică și Microelectronică

Laboratory work 3:
Lexer & Scanner.

Elaborated:

st. gr. FAF-223 Bujilov Dmitrii

Verified:

asist. univ. Dumitru Cretu

Chișinău - 2024

Objectives:

Understand what lexical analysis [1] is.

Get familiar with the inner workings of a lexer/scanner/tokenizer.

Implement a sample lexer and show how it works.

Implementation description

The ArithmeticLexer is a Java class tailored for parsing arithmetic expressions. It disassembles expressions into tokens, representing distinct elements like numbers, operators, and parentheses. This lexer ensures precise tokenization while handling errors effectively.

Key Points:

Token Types Enumeration: Employs TokenType enum to classify tokens into categories such as NUMBER, OPERATOR, LEFT_PAREN, RIGHT_PAREN, WHITESPACE, and ERROR.

Token Structure: Utilizes the Token class to encapsulate token properties like type, value, and position within the input string.

Tokenization Process: Implements the tokenize method to parse expressions character by character, assigning them appropriate token types.

Error Management: Includes error handling mechanisms to generate ERROR tokens with detailed messages when encountering invalid characters or expressions.

Whitespace Handling: Offers flexibility by allowing users to choose whether to ignore whitespace characters during tokenization.

Bracket Matching: Ensures balanced parentheses within expressions, maintaining syntactic correctness.

Implementation Insights:

Parsing Algorithm: Implements a robust parsing algorithm to accurately categorize characters into token types based on predefined rules.

Syntax Validation: Validates the presence of required elements like numbers or brackets before and after operators, ensuring syntactic validity.

Informative Error Messages: Provides descriptive error messages indicating the nature and position of encountered errors, aiding in debugging.

Customization Options: Allows users to tailor whitespace handling through a configurable flag, catering to specific parsing needs.

Usage Scenario:

Developers can leverage the ArithmeticLexer class and its tokenize method to dissect arithmetic expressions, yielding a list of tokens for further processing. This lexer serves as a foundation for constructing arithmetic expression evaluation or parsing systems.

Code (main parts of it):

```
    // Enum defining different types of tokens.
    public enum TokenType {
        NUMBER,         // Represents numeric values.
        OPERATOR,       // Represents arithmetic operators (+, -, *, /).
        LEFT_PAREN,     // Represents left parenthesis '('.
        RIGHT_PAREN,    // Represents right parenthesis ')'.
        WHITESPACE,     // Represents whitespace characters.
        ERROR           // Represents an error token.
    }

    // Class representing a token with its type, value, and position in the
input string.
    public static class Token {
        private TokenType type;     // Type of the token.
        private String value;       // Value of the token.
        private int position;       // Position of the token in the input
string.
        // Constructor to initialize the token.
        public Token(TokenType type, String value, int position) {
            this.type = type;
            this.value = value;
            this.position = position;
        }
        // String representation of the token.
        @Override
        public String toString() {
```

```
            return "[" + type + ": " + value + ", position: " + position + "]";
        }
    }
    private boolean ignoreWhitespace = false;    // Flag to determine if whitespace should be ignored.
    // Method to tokenize the input string and return a list of tokens.
    public List<Token> tokenize(String input) {
        List<Token> tokens = new ArrayList<>();  // List to store tokens.
        StringBuilder currentToken = new StringBuilder(); // StringBuilder to build the current token.
        int currentPosition = 0;    // Current position in the input string.
        int leftParenCount = 0;     // Count of left parentheses.
        int rightParenCount = 0;    // Count of right parentheses.
        int lastDigitStartingPosition = 0; // Starting position of the last digit encountered.
        // Iterate through each character in the input string.
        for (char c : input.toCharArray()) {
            // Check if the character is a whitespace.
            if (Character.isWhitespace(c)) {
                // If whitespace should be ignored, skip to the next character.
                if (ignoreWhitespace) {
                    currentPosition++;
                    continue;
                }
                // Add the current number token if any, and add the whitespace token.
                addNumberIfNeeded(currentToken, tokens, lastDigitStartingPosition);
                tokens.add(new Token(TokenType.WHITESPACE, Character.toString(c), currentPosition));
            }
            // Check if the character is an operator.
            else if (isOperator(c)) {
```

```
                // Check if the operator requires a number before and after it.
                if (requiresNumberBeforeAndAfter(c)) {
                    // Check if there is a number or a bracket before the
operator.
                    if (!hasPreviousNumber(input, currentPosition) &&
!hasPreviousBracket(input, currentPosition)) {
                        return invalidTokenError("Expected number or bracket
before operator", currentPosition);
                    }
                    // Check if there is a number or a bracket after the
operator.
                    if (!hasNextNumber(input, currentPosition) &&
!hasNextBracket(input, currentPosition)) {
                        return invalidTokenError("Expected number or bracket
after operator", currentPosition);
                    }
                }
                // Add the current number token if any, and add the operator
token.
                addNumberIfNeeded(currentToken, tokens,
lastDigitStartingPosition);
                tokens.add(new Token(TokenType.OPERATOR, Character.toString(c),
currentPosition));
            }
            // Check if the character is a digit.
            else if (Character.isDigit(c)) {
                // If the current token is empty, update the starting position
of the last digit encountered.
                if(currentToken.isEmpty()) {
                    lastDigitStartingPosition = currentPosition;
                }
                // Append the digit to the current token.
                currentToken.append(c);
            }
            // Check if the character is a left parenthesis.
```

```
            else if (c == '(') {
                // Add the current number token if any, and add the left
parenthesis token.
                addNumberIfNeeded(currentToken, tokens,
lastDigitStartingPosition);
                tokens.add(new Token(TokenType.LEFT_PAREN,
Character.toString(c), currentPosition));
                leftParenCount++;
            }
            // Check if the character is a right parenthesis.
            else if (c == ')') {
                // Add the current number token if any, and add the right
parenthesis token.
                addNumberIfNeeded(currentToken, tokens,
lastDigitStartingPosition);
                rightParenCount++;
                // Check if there are more right parentheses than left
parentheses.
                if (rightParenCount > leftParenCount) {
                    return invalidTokenError(Character.toString(c),
currentPosition);
                }
                tokens.add(new Token(TokenType.RIGHT_PAREN,
Character.toString(c), currentPosition));
            }
            // If none of the above conditions are met, return an error token.
            else {
                return invalidTokenError(Character.toString(c),
currentPosition);
            }
            currentPosition++; // Move to the next position in the input string.
        }
        // Check if the number of left parentheses matches the number of right
parentheses.
        if (leftParenCount != rightParenCount) {
```

```
            return invalidTokenError("Mismatched parentheses", input.length());
        }
        // Add the current number token if any.
        addNumberIfNeeded(currentToken, tokens, lastDigitStartingPosition);
        return tokens; // Return the list of tokens.
    }
    // Method to create an error token with the provided message and position.
    private List<Token> invalidTokenError(String token, int position) {
        List<Token> errorTokens = new ArrayList<>();
        // Add an error token indicating the invalid expression and its
position.
        errorTokens.add(new Token(TokenType.ERROR, "Invalid expression: " +
token + " at position " + position + ".", position));
        return errorTokens; // Return the list containing the error token.
    }
    // Method to create a token from the provided string and position.
    private Token createToken(String tokenString, int position) {
        // Check if the token string represents a numeric value.
        if (tokenString.matches("\\d+")) {
            // If yes, create and return a token with NUMBER type.
            return new Token(TokenType.NUMBER, tokenString, position);
        } else {
            // If not, create and return an error token.
            return new Token(TokenType.ERROR, "Invalid expression: " +
tokenString + " at position " + position + ".", position);
        }
    }
```

Conclusions and Results

In conclusion, the laboratory work focused on understanding lexical analysis and delved into the inner workings of a lexer, scanner, or tokenizer. Lexical analysis, as discussed, involves extracting lexical tokens from a string of characters based on predefined rules of the language. These tokens, representing various elements such as keywords, identifiers, operators, etc., are crucial for subsequent stages in a compiler or interpreter.

Through the implementation of a sample lexer, the process of tokenizing a string was demonstrated. The lexer effectively split the input based on delimiters, creating lexemes, and assigned tokens to these lexemes based on their types or categories. This distinction between lexemes and tokens was emphasized, highlighting that while lexemes represent the raw units extracted from the input, tokens provide meaningful categorization and may include additional metadata.

By completing this laboratory work, a deeper understanding of the fundamental concepts underlying lexical analysis and the role of lexers in language processing was achieved. Furthermore, the practical implementation of a lexer helped solidify the theoretical knowledge by demonstrating its application in real-world scenarios. Overall, this laboratory work served as a valuable learning experience in formal languages and finite automata, laying a strong foundation for further exploration in compiler design and related fields.