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

Chişinău - 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

Firstly, I added the ArithmeticLexer class. It has many commentaries, which explain the code and the concept itself:

```
import java.util.ArrayList;

import java.util.List;


// This class is responsible for lexing arithmetic expressions, breaking them
down into tokens.

public class ArithmeticLexer {


    // 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;
    }

    // Getters and setters for token properties.
    public TokenType getType() {
        return type;
    }

    public void setType(TokenType type) {
        this.type = type;
    }

    public String getValue() {
        return value;
    }

    public void setValue(String value) {
        this.value = value;
    }

    public int getPosition() {
        return position;
    }

    public void setPosition(int position) {
        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.

    // Default constructor.
    public ArithmeticLexer() {}

    // Constructor with an option to set whether to ignore whitespace.
    public ArithmeticLexer(boolean ignoreWhitespace) {
        this.ignoreWhitespace = ignoreWhitespace;
    }

    // Setter for ignoreWhitespace flag.
    public void setIgnoreWhitespace(boolean ignoreWhitespace) {
        this.ignoreWhitespace = ignoreWhitespace;
    }

    // Getter for ignoreWhitespace flag.
    public boolean getIgnoreWhitespace() {
        return ignoreWhitespace;
    }

    // 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);
        }
    }


    // Method to check if a character is an operator.
    private static boolean isOperator(char c) {
        return c == '+' || c == '-' || c == '*' || c == '/';
```

```
}

// Method to check if an operator requires a number before and after it.
private static boolean requiresNumberBeforeAndAfter(char c) {
    return isOperator(c);
}

// Method to check if there is a number after the current position.
private boolean hasNextNumber(String input, int currentPosition) {
    for (int i = currentPosition + 1; i < input.length(); i++) {
        if (Character.isDigit(input.charAt(i))) {
            return true;
        } else if (!Character.isWhitespace(input.charAt(i))) {
            return false;
        }
    }
    return false;
}

// Method to check if there is a bracket after the current position.
private boolean hasNextBracket(String input, int currentPosition) {
    for (int i = currentPosition + 1; i < input.length(); i++) {
        if (input.charAt(i) == '(') {
            return true;
        } else if (!Character.isWhitespace(input.charAt(i))) {
            return false;
        }
    }
    return false;
}
```

```
    // Method to check if there is a number before the current position.
    private boolean hasPreviousNumber(String input, int currentPosition) {
        for (int i = currentPosition - 1; i >= 0; i--) {
            if (Character.isDigit(input.charAt(i))) {
                return true;
            } else if (!Character.isWhitespace(input.charAt(i))) {
                return false;
            }
        }
        return false;
    }

    // Method to check if there is a bracket before the current position.
    private boolean hasPreviousBracket(String input, int currentPosition) {
        for (int i = currentPosition - 1; i >= 0; i--) {
            if (input.charAt(i) == ')') {
                return true;
            } else if (!Character.isWhitespace(input.charAt(i))) {
                return false;
            }
        }
        return false;
    }

    // Method to add the current number token to the list if it's not empty.
    private void addNumberIfNeeded(StringBuilder currentToken, List<Token>
tokens, int lastDigitStartingPosition) {
        if (!currentToken.isEmpty()) {
            tokens.add(createToken(currentToken.toString(),
lastDigitStartingPosition));
            currentToken.setLength(0); // Clear the current token StringBuilder.
        }
    }
}
```

And I test my lexer in the following way:

```
    private static void testThirdLab(Scanner scanner) {
        ArithmeticLexer arithmeticLexer = new ArithmeticLexer();


        System.out.print("Enter an arithmetic expression: ");
        String input = scanner.nextLine();
        System.out.print("Should whitespaces be ignored? (y/n): ");
        String ignoreWhitespace = scanner.nextLine();


        while (!ignoreWhitespace.equalsIgnoreCase("y") &&
!ignoreWhitespace.equalsIgnoreCase("n")) {
            System.err.print("Invalid input. Please enter 'y' or 'n': ");
            ignoreWhitespace = scanner.nextLine();
        }



arithmeticLexer.setIgnoreWhitespace(ignoreWhitespace.equalsIgnoreCase("y"
));


        List<ArithmeticLexer.Token> tokens =
arithmeticLexer.tokenize(input);


        List<String> answer = new ArrayList<>();
        boolean containsError = false;


        for (ArithmeticLexer.Token token : tokens) {
            if (token.getType() == ArithmeticLexer.TokenType.ERROR) {
                containsError = true;
                System.err.println(token.getValue());
                return;
            }
            answer.add(token.toString());
        }

        if(answer.isEmpty())
            System.out.println("No tokens were found.");

        if(!containsError) {
            System.out.println("The tokens are: ");
            for (String token : answer)
                System.out.println(token);
        }
    }
```

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.