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

A lexer, short for lexical analyzer, is a fundamental component of a compiler or interpreter. Its primary role is to break down the input source code into a sequence of tokens, which are the smallest meaningful units of the language. These tokens could be keywords, identifiers, operators, punctuation symbols, or literals like numbers and strings.

The lexer reads characters from the input source code and groups them into tokens based on predefined rules, called regular expressions or finite state machines. These rules define the syntax of the language being processed.

For example, in the C programming language, the lexer might identify tokens such as "int" (keyword), "main" (identifier), "(" (left parenthesis), ")" (right parenthesis), "{" (left brace), "}" (right brace), ";" (semicolon), etc.

Once the lexer has tokenized the input, the resulting stream of tokens is passed to the next stage of the compiler or interpreter, typically the parser, which then constructs a parse tree or abstract syntax tree to further analyze the structure and semantics of the code.

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.

The evaluation part of the ArithmeticLexer class code is implementing a simple arithmetic expression evaluator using the Shunting Yard algorithm with stacks. Let's break down each part of the code in detail:

Method `evaluate(List<Token> tokens)`

- This method takes a list of tokens representing an arithmetic expression.
- It initializes two stacks: `operandStack` to store operands (numbers) and `operatorStack` to store operators (`+`, `-`, `*`, `/`) and parentheses.
- It iterates through each token in the input list.
- Depending on the type of token:
    - If it's a `NUMBER`, it pushes the parsed integer value onto the `operandStack`.
    - If it's an `OPERATOR`, it compares the precedence of the operator with the operators on the top of the `operatorStack` using the `precedence()` method.
        - While the precedence of the current token is less than or equal to the precedence of the operator on the top of the stack, it evaluates the top of the stacks using the `evaluateTop()` method.
        - Then it pushes the current operator onto the `operatorStack`.
    - If it's a `LEFT_PAREN`, it simply pushes the left parenthesis onto the `operatorStack`.
    - If it's a `RIGHT_PAREN`, it evaluates the operators on the top of the `operatorStack` until it encounters a left parenthesis. It then discards the left parenthesis.
- After processing all tokens, it evaluates any remaining operators on the `operatorStack`.

- Finally, it returns the result, which is the only element left on the `operandStack`.

Method `evaluateTop(Stack<Integer> operandStack, Stack<Character> operatorStack)`

- This method evaluates the top of the stacks by performing the operation on the top two operands using the operator on the top of the operator stack.
- It pops the operator and two operands from their respective stacks.
- It performs the operation using the `performOperation()` method and pushes the result onto the operand stack.

Method `performOperation(int operand1, int operand2, char operator)`

- This method performs arithmetic operations (`+`, `-`, `*`, `/`) based on the operator provided.
- It handles division by zero by throwing an `ArithmeticException`.
- It's implemented using a switch statement to handle different operators.

Method `precedence(char operator)`

- This method returns the precedence of the operator.
- It returns 1 for addition and subtraction, and 2 for multiplication and division.

This code provides a robust and efficient way to evaluate arithmetic expressions by converting them into postfix notation using the Shunting Yard algorithm and then evaluating the postfix expression using stacks.

Code (main parts of it):

```
public List<Token> tokenize(String input) {
    List<Token> tokens = new ArrayList<>();
    StringBuilder currentToken = new StringBuilder();
    int currentPosition = 0;
    int leftParenCount = 0;
    int rightParenCount = 0;
    int lastDigitStartingPosition = 0;
    for (char c : input.toCharArray()) {
        if (Character.isWhitespace(c)) {
            if (ignoreWhitespace) {
                currentPosition++;
                continue;
            }
            addNumberIfNeeded(currentToken, tokens, lastDigitStartingPosition);
```

```
                tokens.add(new Token(TokenType.WHITESPACE, Character.toString(c),
currentPosition));

            } else if (isOperator(c)) {

                if (requiresNumberBeforeAndAfter(c)) {

                    if (!hasPreviousNumber(input, currentPosition) && !hasPreviousBracket(input,
currentPosition)) {

                        return invalidTokenError("Expected number or bracket before operator",
currentPosition);

                    }

                    if (!hasNextNumber(input, currentPosition) && !hasNextBracket(input,
currentPosition)) {

                        return invalidTokenError("Expected number or bracket after operator",
currentPosition);

                    }

                }

                addNumberIfNeeded(currentToken, tokens, lastDigitStartingPosition);

                tokens.add(new Token(TokenType.OPERATOR, Character.toString(c),
currentPosition));

            } else if (Character.isDigit(c)) {

                if(currentToken.isEmpty()) {

                    lastDigitStartingPosition = currentPosition;

                }

                currentToken.append(c);

            } else if (c == '(') {

                addNumberIfNeeded(currentToken, tokens, lastDigitStartingPosition);

                tokens.add(new Token(TokenType.LEFT_PAREN, Character.toString(c),
currentPosition));

                leftParenCount++;

            } else if (c == ')') {

                addNumberIfNeeded(currentToken, tokens, lastDigitStartingPosition);

                rightParenCount++;

                if (rightParenCount > leftParenCount) {

                    return invalidTokenError(Character.toString(c), currentPosition);

                }

                tokens.add(new Token(TokenType.RIGHT_PAREN, Character.toString(c),
currentPosition));

            } else {

                return invalidTokenError(Character.toString(c), currentPosition);
```

```
            }
            currentPosition++;
        }
        if (leftParenCount != rightParenCount) {
            return invalidTokenError("Mismatched parentheses", input.length());
        }
        addNumberIfNeeded(currentToken, tokens, lastDigitStartingPosition);
        return tokens;
    }
    public int evaluate(List<Token> tokens) {
        Stack<Integer> operandStack = new Stack<>();
        Stack<Character> operatorStack = new Stack<>();
        for (Token token : tokens) {
            switch (token.getType()) {
                case NUMBER:
                    operandStack.push(Integer.parseInt(token.getValue()));
                    break;
                case OPERATOR:
                    while (!operatorStack.isEmpty() && precedence(operatorStack.peek()) >=
precedence(token.getValue().charAt(0))) {
                        evaluateTop(operandStack, operatorStack);
                    }
                    operatorStack.push(token.getValue().charAt(0));
                    break;
                case LEFT_PAREN:
                    operatorStack.push('(');
                    break;
                case RIGHT_PAREN:
                    while (operatorStack.peek() != '(') {
                        evaluateTop(operandStack, operatorStack);
                    }
                    operatorStack.pop();
                    break;
                default:
                    break;
```

```
            }
        }
        while (!operatorStack.isEmpty()) {
            evaluateTop(operandStack, operatorStack);
        }
        return operandStack.pop();
    }
```

Conclusions and Results

The laboratory work has provided a comprehensive exploration of lexical analysis, tokenization, and expression evaluation, shedding light on fundamental concepts crucial in language processing and compiler design. Lexical analysis, the initial stage of language processing, involves extracting lexical tokens from a string of characters according to predefined language rules. Through the implementation of a sample lexer, we gained practical insights into how lexical tokens are generated, categorized, and utilized to represent elements such as keywords, identifiers, and operators.

Furthermore, the distinction between lexemes and tokens was highlighted, emphasizing that while lexemes are the raw units extracted from the input, tokens provide meaningful categorization and may include additional metadata. The process of tokenization, demonstrated through the lexer implementation, showcased its importance in facilitating subsequent stages of interpretation or compilation.

In addition to lexical analysis, the laboratory work explored the evaluation of arithmetic expressions, showcasing the practical application of tokenization in language processing. The implementation of an expression evaluator demonstrated how tokens extracted through lexical analysis can be leveraged to evaluate mathematical expressions efficiently. By employing stacks and precedence rules, the evaluator processed arithmetic expressions in a structured manner, handling operators and operands effectively.

Through this combined exploration of lexical analysis and expression evaluation, the laboratory work has not only provided insights into the foundational concepts of language processing but also underscored the interconnectedness between lexical analysis, tokenization, and subsequent stages of language interpretation or compilation. This holistic understanding fosters a deeper appreciation of compiler design principles and lays a solid groundwork for further study and exploration in this domain. Overall, the laboratory work has served as a valuable learning experience, offering practical insights into formal languages, finite automata, and their applications in compiler design and related fields.