MINISTERUL EDUCAȚIEI, CULTURII ȘI CERCETĂRII AL REPUBLICII MOLDOVA
Universitatea Tehnică a Moldovei

Facultatea Calculatoare, Informatică şi Microelectronică
Departamentul Inginerie Software și Automatică

Zagorodniuc Anastasia FAF-223

Report

Intro to formal languages. Regular grammar. Finite Automata.

of Formal Languages & Finite Automata

Checked by:

Cretu Dumitru, *university assistant*

Chișinău – 2023

Objectives

1. Understand what lexical analysis [1] is.
2. Get familiar with the inner workings of a lexer/scanner/tokenizer.
3. Implement a sample lexer and show how it works.

Theory block

A lexer, also known as a scanner or tokenizer, is a fundamental component of a compiler or interpreter. Its primary function is to perform lexical analysis by breaking down the input source code into meaningful units called tokens. In the context of programming languages, lexical analysis is often the initial step before parsing, where the source code is analyzed for syntactical structure. Traditionally, the process of parsing programming languages involves two distinct phases: lexical analysis (lexing) and syntax analysis (parsing). Although both phases are essentially parsers, they serve different roles. The lexer converts the input source code into a stream of tokens representing fundamental language elements such as identifiers, literals, and operators. These tokens serve as input for the parser, which constructs a hierarchical representation of the code's syntactic structure. The separation of lexing and parsing is advantageous because the lexer's task is typically simpler and more straightforward than that of the parser. The lexer focuses on recognizing individual tokens and handling lexical elements like reserved keywords and whitespace elimination.

Code

This is the function that runs the laboratory work. Inside it is an input that is read from the console and after that a new Lexer is created with this input. After that from this lever program gets every next token.

```
private static void RunLab3()
{
    try
    {
        Console.WriteLine("Enter an arithmetic expression:");
        string input = Console.ReadLine();
        Lexer lexer = new Lexer(input);

        Token token;
        do
        {
            token = lexer.GetNextToken();
            Console.WriteLine(token);
        } while (token.Type != Token.TokenType.EOF);
    }
    catch (Exception ex)
    {
        Console.WriteLine($"Error in lab2: {ex.Message}");
    }
}
```

This is how the program gets every next token. It checks the position of each symbol and while its smaller than the input length, it parses the symbols. If the program sees the white space it skips it.

```
public Token GetNextToken()
{
    while (_position < _input.Length)
    {
        if (char.IsWhiteSpace(CurrentChar))
        {
            SkipWhitespace();
            continue;
        }

        if (char.IsDigit(CurrentChar))
        {
            return ParseInteger();
        }

        return ParseSymbol();
    }
```

This how I parse all the symbols

```
private Token ParseSymbol()
{
    switch (CurrentChar)
    {
        case '+':
            Advance();
            return new Token(Token.TokenType.Plus);
        case '-':
            Advance();
            return new Token(Token.TokenType.Minus);
        case '*':
            Advance();
            return new Token(Token.TokenType.Multiply);
        case '/':
            Advance();
            return new Token(Token.TokenType.Divide);
        case '=':
            Advance();
            return new Token(Token.TokenType.Equals);
        case '(':
            Advance();
            return new Token(Token.TokenType.LeftParenthesis);
        case ')':
            Advance();
            return new Token(Token.TokenType.RightParenthesis);
        default:
            throw new ArgumentException($"Invalid character: '{CurrentChar}'");
    }
}
```

Output:

The output of the lexer implementation would be a sequence of tokens generated from the input source code. Each token comprises a type (e.g., Integer, Plus, Minus) and, optionally, a corresponding value.

```
Enter lab:
lab3
Enter an arithmetic expression:
3+2=5
Token(Integer, '3')
Token(Plus)
Token(Integer, '2')
Token(Equal)
Token(Integer, '5')
Token(EOF)
```

Conclusion:

In conclusion, this laboratory work has provided valuable insights into the concept of lexical analysis and the role of a lexer in the compilation process. By implementing a basic lexer, I have gained practical experience in tokenizing source code and understanding its structure at a lexical level. This knowledge forms a foundational understanding for further studies in compiler construction and language processing.