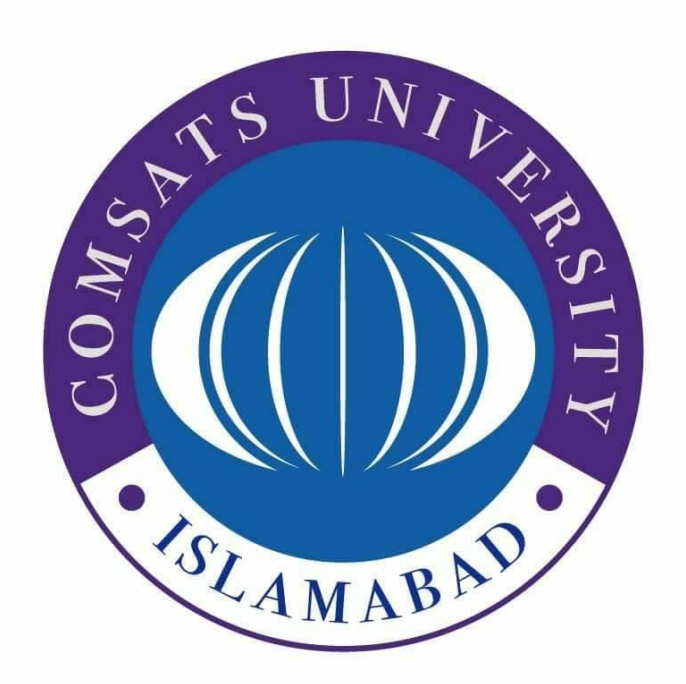


Group Members

Aiman Khatoon (Fa20-bcs-017)

Mahnoor (Fa20-bcs-045)

Submitted To: Sir Bilal Haider Bukhari

Date of Submission: 28-Dec-2023

Lab Terminal

1: Brief/Objective of the project:

The goal of this project is to design and implement a compiler that performs both lexical and semantic analysis for a programming language using the C# programming language. The compiler will be responsible for translating high-level source code into an intermediate representation, paving the way for subsequent stages of compilation.

Lexical Analysis:

Develop a lexical analyzer using regular expressions and finite automata.

Tokenize source code, handling keywords, identifiers, literals, operators, etc.

Generate a token stream for the next compiler stages.

Semantic Analysis:

Construct a symbol table for managing identifier information.

Implement type checking for operand and expression compatibility.

Detect and report semantic errors, handle scoping rules, and manage variable lifetimes.

Syntax Analysis:

If needed, implement a syntax analyzer using context-free grammars and parsing techniques.

Verify the syntactic correctness of the source code.

Error Handling:

Develop a robust error-handling mechanism for meaningful error messages.

Implement error recovery strategies to handle compilation errors gracefully.

Documentation and Testing:

Provide comprehensive documentation covering design decisions, algorithms, and user guidelines.

Create a suite of test cases for thorough validation of compiler correctness and performance.

User Interface (optional):

Develop a simple user interface or command-line interface for user interaction.

Allow users to input source code files and receive compiled output.

2: functionalities along with screenshots (function code +output)

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text.RegularExpressions;
using System.Windows.Forms;

namespace MathExpressionAnalyzer
{
    public partial class Form1 : Form
    {
        private string[] inputTokens;
        private int currentPosition;
```

```
public Form1()
{
    InitializeComponent();
}

private void analyzeButton_Click(object sender, EventArgs e)
{
    string expression = textBox1.Text;
    inputTokens = Tokenize(expression);

    try
    {
        currentPosition = 0;
        AnalyzeExpression();
        label1.Text = "Analysis successful: Valid math expression!";
    }
    catch (Exception ex)
    {
        label1.Text = $"Error: {ex.Message}";
    }
}

private string[] Tokenize(string expression)
{
    return Regex.Split(expression, @"(\d+|[-+*/()])", RegexOptions.IgnorePatternWhitespace)
                .Where(s => !string.IsNullOrWhiteSpace(s)).ToArray();
}

private string GetCurrentToken()
{
```

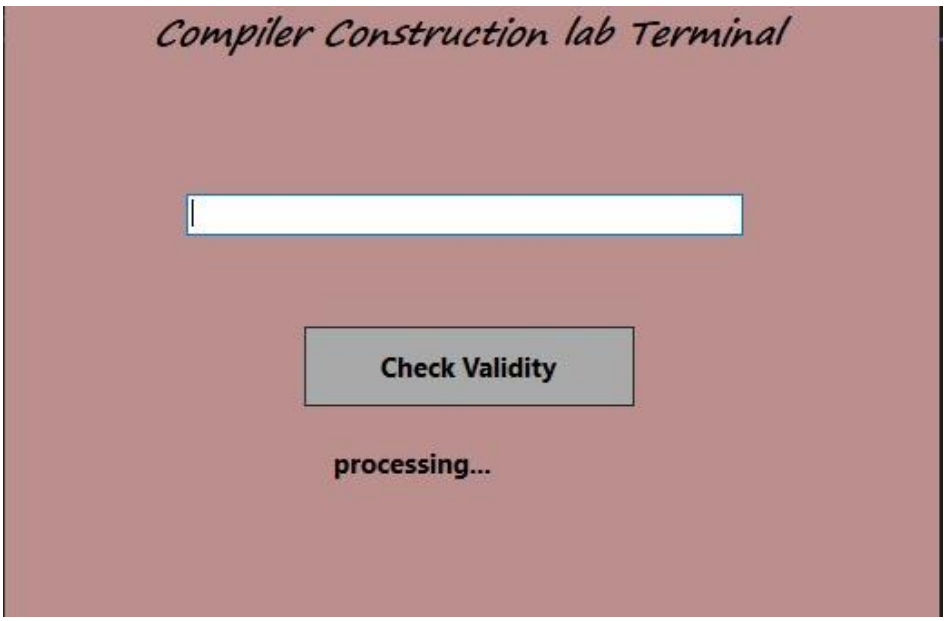
```
    if (currentPosition < inputTokens.Length)
        return inputTokens[currentPosition];
    return "$"; // End of input marker
}

private void ConsumeToken()
{
    currentPosition++;
}

private void AnalyzeExpression()
{
    AnalyzeT();
    AnalyzeEPrime();
}

private void AnalyzeEPrime()
{
    string token = GetCurrentToken();
    if (token == "+")
    {
        ConsumeToken();
        AnalyzeT();
        AnalyzeEPrime();
    }
    else if (token == "$" || token == ")")
    {
        // E' -> ?
        // Do nothing
    }
```

```
    else
    {
        throw new Exception($"Unexpected token: {token}");
    }
}

private void AnalyzeT()
{
    AnalyzeF();
    AnalyzeTPrime();
}

private void AnalyzeTPrime()
{
    string token = GetCurrentToken();
    if (token == "*")
    {
        ConsumeToken();
        AnalyzeF();
        AnalyzeTPrime();
    }
    else if (token == "+" || token == "$" || token == ")")
    {
        // T' -> ?
        // Do nothing
    }
    else
    {
        throw new Exception($"Unexpected token: {token}");
    }
```

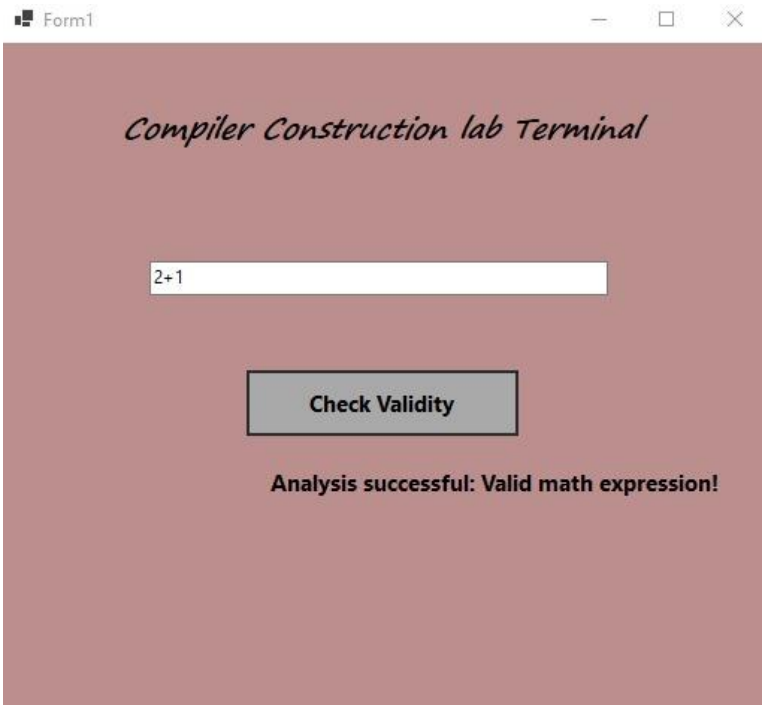
```
}

private void AnalyzeF()
{
    string token = GetCurrentToken();
    if (token == "(")
    {
        ConsumeToken();
        AnalyzeExpression();
        if (GetCurrentToken() == ")")
        {
            ConsumeToken();
        }
        else
        {
            throw new Exception("Expected closing parenthesis");
        }
    }
    else if (int.TryParse(token, out _))
    {
        ConsumeToken();
    }
    else
    {
        throw new Exception($"Unexpected token: {token}");
    }
}

private void label2_Click(object sender, EventArgs e)
{
```

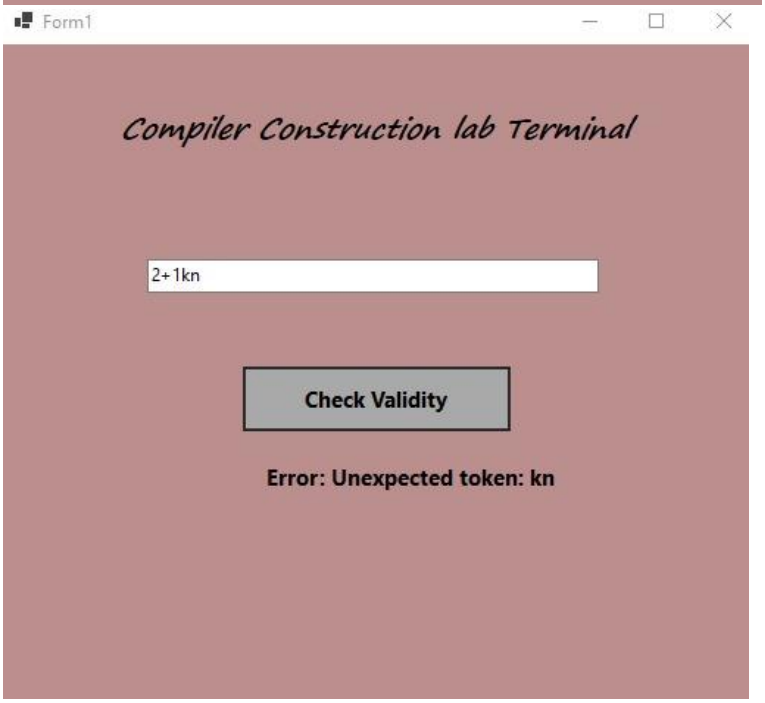
```
        }
    }
}
```

OUTPUT:


Compiler Construction lab Terminal
Check Validity
processing...

3: input and output of the project:


Form1
Compiler Construction lab Terminal
2+1
Check Validity
Analysis successful: Valid math expression!


Form1
Compiler Construction lab Terminal
2+1kn
Check Validity
Error: Unexpected token: kn

4: how functions works. Step by step:

Lexical Analysis:

Input: Source code in the form of a character stream.

Functionality: a. Tokenization: A function reads the input character stream and breaks it into tokens based on predefined lexical rules using regular expressions and finite automata. b. Categorization: Another function categorizes each token into types such as keywords, identifiers, literals, and operators. c. Token Stream Generation: A function generates a stream of tokens representing the input source code, and this token stream is passed to the next stage of the compiler.

Semantic Analysis:

Input: Token stream from the lexical analysis stage.

Functionality: a. Symbol Table Construction: A function constructs a symbol table to store information about identifiers, including their types and scopes. b. Type Checking: Functions are implemented to check the compatibility of operands and expressions, ensuring that the program adheres to semantic rules. c. Error Detection and Reporting: Functions detect and report semantic errors, such as undeclared variables or type mismatches, providing meaningful error messages to aid developers. d. Scoping and Lifetime Management: Functions manage scoping rules and handle variable lifetimes, ensuring correct variable access and lifetime management.

Syntax Analysis (if applicable):

Input: Token stream or parse tree from previous stages.

Functionality: a. Parse Tree Generation: If not covered by lexical and semantic analysis, a function generates a parse tree based on context-free grammars and parsing techniques such as LL or LR parsing. b. Syntactic Correctness Verification: Functions verify the syntactic correctness of the source code by analyzing the structure of the parse tree or token stream.

Error Handling:

Input: Detected errors during lexical, semantic, or syntax analysis.

Functionality: a. Error Reporting: Functions report errors with meaningful messages, aiding developers in identifying and fixing issues. b. Error Recovery Strategies: Functions implement strategies to gracefully recover from errors and continue the compilation process.

Documentation and Testing:

Input: Compiler components, algorithms, and test cases.

Functionality: a. Documentation Functions: Functions generate comprehensive documentation detailing design decisions, algorithms, and user guidelines. b. Testing Functions: Functions execute a suite of test cases to validate the correctness and performance of the compiler.

User Interface (optional):

Input: User commands or source code files.

Functionality: a. User Interface Functions: Functions handle user interactions, allowing input of source code files and providing compiled output through a simple user interface or command-line interface.

5: what challenges your faces during the project:

Implementing lexical and semantic analysis for a compiler in C# poses various challenges, including the intricate nature of language specifications, complexities in designing regular expressions and finite automata for token recognition, and the need for a robust symbol table considering scoping rules and variable lifetimes. Type checking and detection of semantic errors demand a deep understanding of

language semantics. The implementation of optimization strategies, error handling mechanisms, and recovery strategies requires careful consideration. Designing an intermediate code representation that balances optimization gains with target language suitability is challenging, as is the task of developing a user-friendly interface if part of the project. Comprehensive documentation and thorough test case development are time-consuming but crucial for ensuring the correctness and performance of the compiler. Effective project management and adaptability to unexpected challenges further contribute to the complexity of the compiler construction process.