CONSTRUCTION COMPILER LAB TERMINAL (Q2 & 3)

SUBMITTED BY : FAJAR AAMIR SHEIKH

REGISTRATION NO: SP22-BCS-031

SUBMITTED TO : SIR BILAL BUKHARI

SUBMISSION DATE: 18TH JUNE2025

QUESTION NO 2:

Explain any 2 analysis functionalities along with screenshots (function code +output)

Functionality 1: Lexical Analysis (Token Generation)

Purpose:

Breaks input source code into valid tokens: keywords, identifiers, literals, operators, etc.

CODE:

```
using System;
using System.Text.RegularExpressions;

0 references
class SyntaxAnalyzer
{
    1 reference
    public static void ParseIfStatement(string code)
    {
        string pattern = @"^if\s*\(\s*[a-zA-Z_]\w*\s*[><=]\s*\d+\s*\)\s*\{\s*[a-zA-Z_]\w*\s*=\s*[a-zA-
        if (Regex.IsMatch(code, pattern))
        {
            Console.WriteLine(" Syntax is valid: IF statement is correct.");
        }
        else
        {
            Console.WriteLine(" Syntax Error: Invalid IF statement.");
        }
    }

    0 references
    static void Main()
    {
```

```
        else
        {
            Console.WriteLine(" Syntax Error: Invalid IF statement.");
        }
    }

    0 references
    static void Main()
    {
        Console.WriteLine("Enter an if-statement (e.g., if (x > 5) { x = x + 1; }):");
        string input = Console.ReadLine();

        ParseIfStatement(input);
    }
}
```

OUTPUT:

```
Microsoft Visual Studio Debu

=== Tokens ===
[1] => int
[Keyword] => int
[Identifier] => x
[Operator] => =
[Number] => 10
[Punctuation] => ;
[1] => if
[Keyword] => if
[Punctuation] => (
[Identifier] => x
[Operator] => >
[Number] => 5
[Punctuation] => )
[Punctuation] => {
[Identifier] => x
[Operator] => =
[Identifier] => x
[Operator] => +
[Number] => 1
[Punctuation] => ;
[Punctuation] => }

C:\Users\HP\source\repos\ConsoleApp3\Q2\Q2\bin\Debug\net8.0\Q2.exe (process 9732) exited with code 0 (0x0).
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically cl
ging stops.
Press any key to close this window . . .
```

<u>Functionality 2: Syntax Analysis</u> (Simple Parser - IF Statement Validation)

Purpose:

Validate structure of simple if statements using regular expression parsing logic.

CODE:

```
using System;
using System.Text.RegularExpressions;

0 references
class SyntaxAnalyzer
{
    1 reference
    public static void ParseIfStatement(string code)
    {
        string pattern = @"^if\s*\(\s*[a-zA-Z_]\w*\s*[><=]\s*\d+\s*\)\s*\{\s*[a-zA-Z_]\w*\s*=\s*[a-zA-

        if (Regex.IsMatch(code, pattern))
        {
            Console.WriteLine("☑ Syntax is valid: IF statement is correct.");
        }
        else
        {
            Console.WriteLine("❌ Syntax Error: Invalid IF statement.");
        }
    }

    0 references
    static void Main()
    {
```

```
        else
        {
            Console.WriteLine("❌ Syntax Error: Invalid IF statement.");
        }
    }

    0 references
    static void Main()
    {
        Console.WriteLine("Enter an if-statement (e.g., if (x > 5) { x = x + 1; }):");
        string input = Console.ReadLine();

        ParseIfStatement(input);
    }
}
```

OUTPUT:

```
Microsoft Visual Studio Debu

Enter an if-statement (e.g., if (x > 5) { x = x + 1; }):
if (x > 5) { x = x + 1; }
? Syntax is valid: IF statement is correct.
```

QUESTION NO 3:

For any given input give detail of how you arrive at the output.(attach relevant code segements and give screenshot of input and output)

1. Lexical Analysis (Tokenization)

The Lexer class reads the characters and converts them into meaningful **tokens**

```
minicompiler2 | MiniCompiler.Program | Main()

public Token NextToken()
{
    while (char.IsWhiteSpace(Current)) _pos++;

    int start = _pos;

    if (char.IsLetter(Current))
    {
        while (char.IsLetterOrDigit(Current)) _pos++;
        string word = _input.Substring(start, _pos - start);
        return new Token(word == "int" ? TokenType.Keyword : TokenType.Identifier, word, star
    }

    if (char.IsDigit(Current))
    {
        while (char.IsDigit(Current)) _pos++;
        return new Token(TokenType.Number, _input.Substring(start, _pos - start), start);
    }

    if ("=+-*/;".Contains(Current))
    {
        return new Token(TokenType.Operator, _input[_pos++].ToString(), start);
    }
```

```
            if ("=+-*/;".Contains(Current))
            {
                return new Token(TokenType.Operator, _input[_pos++].ToString(), start);
            }

            if (Current == '\0')
                return new Token(TokenType.EOF, "", _pos);

            throw new Exception($"Lexical Error at position {_pos}: Invalid character '{Current}'");
        }
    }
```

OUTPUT:

```
--- Compilation Start ---

########## Line 1 ##########
[Input] int x = 5;

+------------------+
| Lexical Analysis |
+------------------+
| Keyword: int     |
| Identifier: x    |
| Operator: =      |
| Number: 5        |
| Operator: ;      |
+------------------+
```

These are the tokens extracted from the input line.

2. Syntax Analysis

The Parser.ParseAssignment() method expects a strict format:

```
    1 reference
    public void ParseAssignment()
    {
        Eat(TokenType.Keyword);                // int
        VariableName = _current.Value;
        Eat(TokenType.Identifier);             // x
        Eat(TokenType.Operator);               // =
        VariableValue = _current.Value;
        Eat(TokenType.Number);                 // 5
        Eat(TokenType.Operator);               // ;
    }
}

    1 reference
```

OUTPUT:

```
+------------------------------+
| Syntax Analysis              |
+------------------------------+
| Parsed assignment: int x = 5; |
+------------------------------+
```

The parser confirms that the syntax is correct and extracts:

- VariableName = x
- VariableValue = 5

3. Semantic Analysis & Symbol Table

The SymbolTable class:

- Declares the variable x as int
- Checks that x is declared before usage
- Ensures correct type usage

```
// Phase 3 & 7: Semantic Analysis + Symbol Table
symbolTable.Declare(parser.VariableName, "int");
symbolTable.Check(parser.VariableName, "int");
PrintBox("Semantic Analysis & Symbol Table", new List<string> {
    $"Variable '{parser.VariableName}' declared as 'int'",
    $"Type check passed for '{parser.VariableName}'"
});
```

OUTPUT:

```
+-----------------------------------+
| Semantic Analysis & Symbol Table |
+-----------------------------------+
| Variable 'x' declared as 'int'    |
| Type check passed for 'x'         |
+-----------------------------------+
```

4. Optimization (Constant Folding)

The Optimizer does basic constant folding:

```
1 reference
public class Optimizer
{
    1 reference
    public string ConstantFold(string expr)
    {
        if (expr == "2 + 3") return "5"; // Example folding
        return expr;
    }
}
```

OUTPUT:

```
+-------------------------+
| Optimization            |
+-------------------------+
| No optimization applied |
+-------------------------+
```

5. Intermediate Code Generation (IR)

The IRGenerator generates two lines of intermediate code:

```
1 reference
public class IRGenerator
{
    1 reference
    public List<string> Generate(string id, string value)
    {
        return new List<string> {
            $"t1 = {value}",
            $"{id} = t1"
        };
    }
}
```

OUTPUT:

```
+-----------------------------+
| Intermediate Code Generation |
+-----------------------------+
| t1 = 5                      |
| x = t1                      |
+-----------------------------+
```

Intermediate Representation (IR) helps abstract away hardware for later stages.

6. Target Code Generation

The TargetCodeGenerator translates it into target-level pseudo code:

```
1 reference
public class TargetCodeGenerator
{
    1 reference
    public List<string> Generate(string id, string value)
    {
        return new List<string> {
            $"LOAD {value}",
            $"STORE {id}"
        };
    }
}
```

OUTPUT:

```
+-------------------------+
| Target Code Generation |
+-------------------------+
| LOAD 5                  |
| STORE x                 |
+-------------------------+

Line compiled successfully ?
```

Line Compiled successfully.