Lab Terminal

Project: Mini C Compiler using Lex and Yacc

Group Members:

Sana Saeed sp21-bcs-032
Fatimah Ijaz sp21-bcs-006

Course: Compiler construction
Date: 31/05/2024
Submitted To: Mr. Syed Bilal Haider

Question No.01:

Write an introduction of your compiler construction project.

INTRODUCTION

A compiler is a special program that processes statements written in a particular programming language and turns them into machine language or code that a computer's processors use. The file used for writing a C-language contains what are called the source statements. The programmer then runs the appropriate language compiler, specifying the name of the file that contains the source statements. When executing, the compiler first parses all of the language statements syntactically one after the other and then, in one or more successive stages, builds the output code, making sure that statements that refer to other statements are referred to correctly in the final code. The output of the compilation is called object code or sometimes an object module.

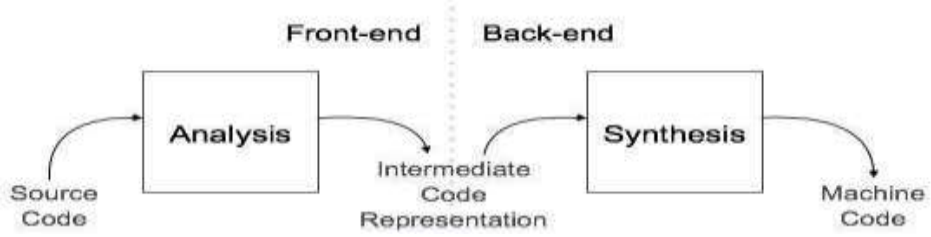
Lexical analysis is the first phase of a compiler. It takes the modified source code from language preprocessors that are written in the form of sentences. The lexical analyzer breaks these syntaxes into a series of tokens, by removing any whitespace or comments in the source code. Symbol table is an important data structure created and maintained by compilers in order to store information about the occurrence of various entities such as variable names, function names, etc.

Symbol table is used by both the analysis and the synthesis parts of a compiler. We have designed a lexical analyzer for the C language using lex. It takes as input a C code and outputs a stream of tokens. The tokens displayed as part of the output include keywords, identifiers, signed/unsigned integer/floating point constants, operators, special characters, headers, data-type specifiers, array, single-line comment, multi-line comment, preprocessor directive, pre-defined functions (printf and scanf), user-defined functions and the main function. The token, the type of token and the line number of the token in the C code are being displayed. The line number is displayed so that it is easier to debug the code for errors. Errors in single-line comments, multi-line comments are displayed along with line numbers. The output also contains the symbol table which contains tokens and their type. The symbol table is generated using the hash organization.

ARCHITECTURE OF LANGUAGE

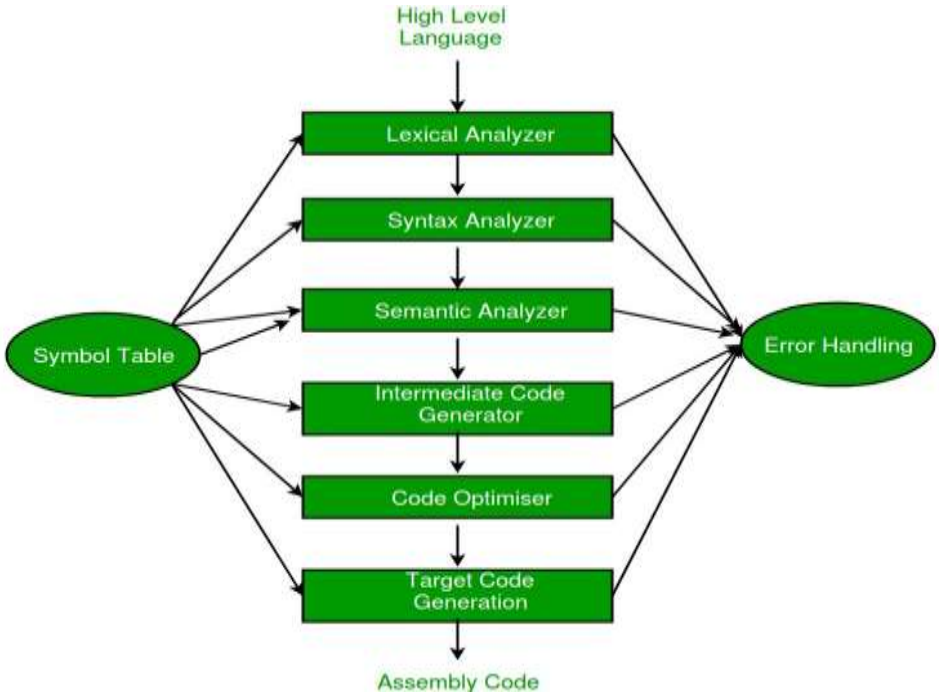
1. *Analysis phase:* Known as the front-end of the compiler, the analysis phase of the compiler reads the source program, divides it into core parts and then checks for lexical, grammar and syntax errors. The analysis phase generates an intermediate representation of the source program and symbol table.
This phase consists of:

- ➢ Lexical Analysis
- ➢ Syntax Analysis
- ➢ Semantic Analysis



2. Synthesis phase: Known as the back-end of the compiler, the synthesis phase generates the target program with the help of intermediate source code representation and symbol table. This phase consists of: ➢ Code Optimization

➢ Intermediate Code Generation



Lexical Analysis

Lexical analysis is the first phase of a compiler. It takes the modified source code from language preprocessors that are written in the form of sentences. The lexical analyzer breaks these syntaxes into a series of tokens, by removing any whitespace or comments in the source code. If the lexical analyzer finds a token invalid, it generates an error. The lexical analyzer works closely with the syntax analyzer. It reads character streams from the source code, checks for legal tokens, and passes the data to the syntax analyzer when it demands.

Syntax Analysis

Syntax analysis or parsing is the second phase of a compiler. It takes the token produced by lexical analysis as input and generates a parse tree (or syntax tree).

Semantic Analysis

Semantic analysis is the third phase of a compiler. Semantic analyzer checks whether the parse tree constructed by the syntax analyzer follows the rules of language.

Intermediate Code Generator

It generates intermediate code, that is a form which can be readily executed by machine. We have many popular intermediate codes. Example – Three address code etc. Intermediate code is converted to machine language using the last two phases which are platform dependent.

Till intermediate code, it is the same for every compiler out there, but after that, it depends on the platform. To build a new compiler we don't need to build it from scratch. We can take the intermediate code from the already existing compiler and build the last two parts.

Code Optimizer

It transforms the code so that it consumes fewer resources and produces more speed. The meaning of the code being transformed is not altered. Optimization can be categorized into two types: machine dependent and machine independent.

In our project, we have handled the following constructs:

- Looping construct: while, for, do-while
- Data types: (signed/unsigned) int, float
- Arithmetic and Relational Operators
- Data structure: Arrays
- User defined functions
- Keywords of C language
- Single and multi-line comments
- Identifiers and Constant errors
- Selection statement: (nested) if-else, while

Question No :02

Give a sample input and output for your compiler construction project

7. Input and Output of Mini Compiler

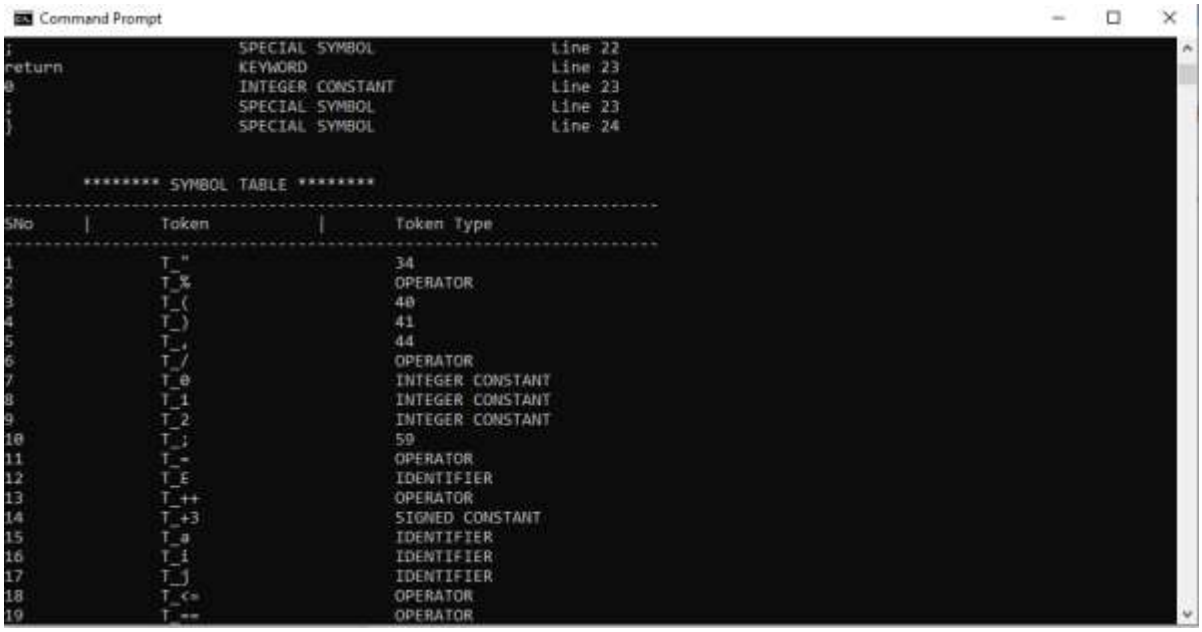
7.1 Lexical Analysis

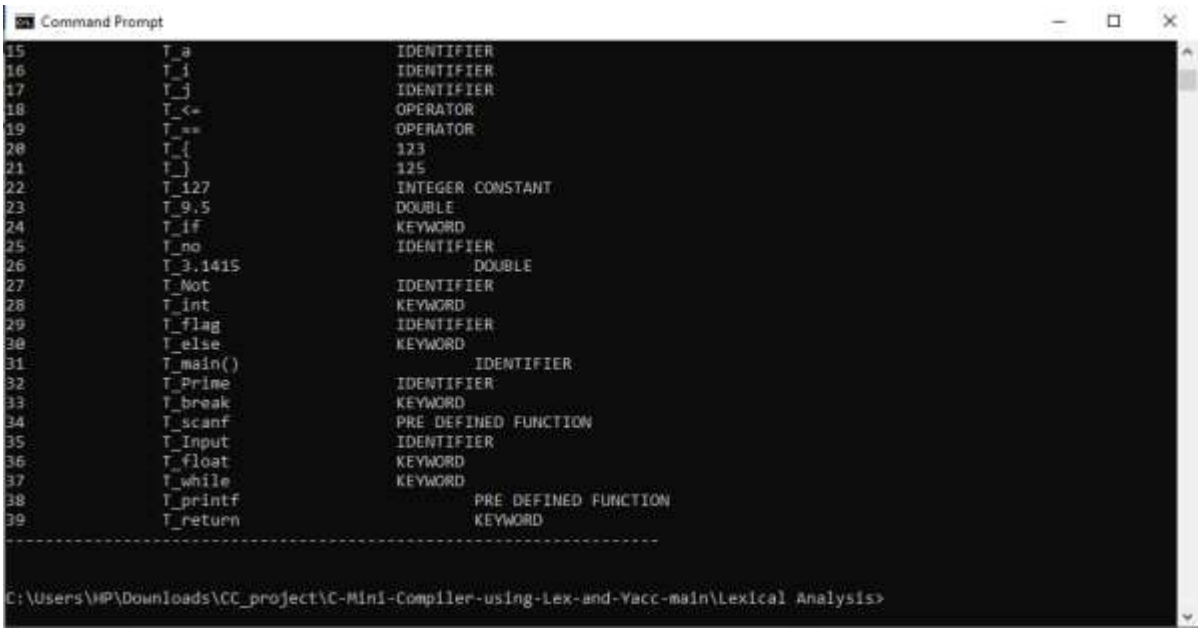
Test case : **isPrime.c**

isPrime.c

```c
#include<stdio.h>
int main()
{
    int a,i,j,flag=0;
    printf("Input no"); //Input
    scanf("%d",&a);
    i=3.1415E+3;
    j=127;
    float 3b = 9.5;
    while(i <= a/2)
    {
        if(a%i == 0)
        {
            flag=1;
            break;
        }
        i++;
    }
    if(flag==0)
        printf("Prime"); // It's a prime number.
    else
        printf("Not Prime");
    return 0;
}
```

```
Command Prompt
Microsoft Windows [Version 10.0.19045.2251]
(c) Microsoft Corporation. All rights reserved.

C:\Users\HP>cd C:\Users\HP\Downloads\CC_project\C-Mini-Compiler-using-Lex-and-Yacc-main\Lexical Analysis

C:\Users\HP\Downloads\CC_project\C-Mini-Compiler-using-Lex-and-Yacc-main\Lexical Analysis>flex lexer.l

C:\Users\HP\Downloads\CC_project\C-Mini-Compiler-using-Lex-and-Yacc-main\Lexical Analysis>gcc lex.yy.c

C:\Users\HP\Downloads\CC_project\C-Mini-Compiler-using-Lex-and-Yacc-main\Lexical Analysis>a.exe


#include<stdio.h>       HEADER                  Line 1
int                     KEYWORD                 Line 2
main()                  MAIN FUNCTION           Line 2
{                       SPECIAL SYMBOL          Line 3
int                     KEYWORD                 Line 4
a                       IDENTIFIER              Line 4
,                       SPECIAL SYMBOL          Line 4
i                       IDENTIFIER              Line 4
,                       SPECIAL SYMBOL          Line 4
j                       IDENTIFIER              Line 4
;                       SPECIAL SYMBOL          Line 4
```


Command Prompt
; SPECIAL SYMBOL Line 22
return KEYWORD Line 23
0 INTEGER CONSTANT Line 23
; SPECIAL SYMBOL Line 23
} SPECIAL SYMBOL Line 24
******** SYMBOL TABLE ********
SNo | Token | Token Type
1 T_" 34
2 T_% OPERATOR
3 T_(40
4 T_) 41
5 T_, 44
6 T_/ OPERATOR
7 T_0 INTEGER CONSTANT
8 T_1 INTEGER CONSTANT
9 T_2 INTEGER CONSTANT
10 T_; 59
11 T_= OPERATOR
12 T_E IDENTIFIER
13 T_++ OPERATOR
14 T_+3 SIGNED CONSTANT
15 T_a IDENTIFIER
16 T_i IDENTIFIER
17 T_j IDENTIFIER
18 T_<= OPERATOR
19 T_== OPERATOR


Command Prompt
15 T_a IDENTIFIER
16 T_i IDENTIFIER
17 T_j IDENTIFIER
18 T_<= OPERATOR
19 T_== OPERATOR
20 T_{ 123
21 T_} 125
22 T_127 INTEGER CONSTANT
23 T_9.5 DOUBLE
24 T_if KEYWORD
25 T_no IDENTIFIER
26 T_3.1415 DOUBLE
27 T_Not IDENTIFIER
28 T_int KEYWORD
29 T_flag IDENTIFIER
30 T_else KEYWORD
31 T_main() IDENTIFIER
32 T_Prime IDENTIFIER
33 T_break KEYWORD
34 T_scanf PRE DEFINED FUNCTION
35 T_Input IDENTIFIER
36 T_float KEYWORD
37 T_while KEYWORD
38 T_printf PRE DEFINED FUNCTION
39 T_return KEYWORD
C:\Users\HP\Downloads\CC_project\C-Mini-Compiler-using-Lex-and-Yacc-main\Lexical Analysis>

7.2 Syntax Analysis

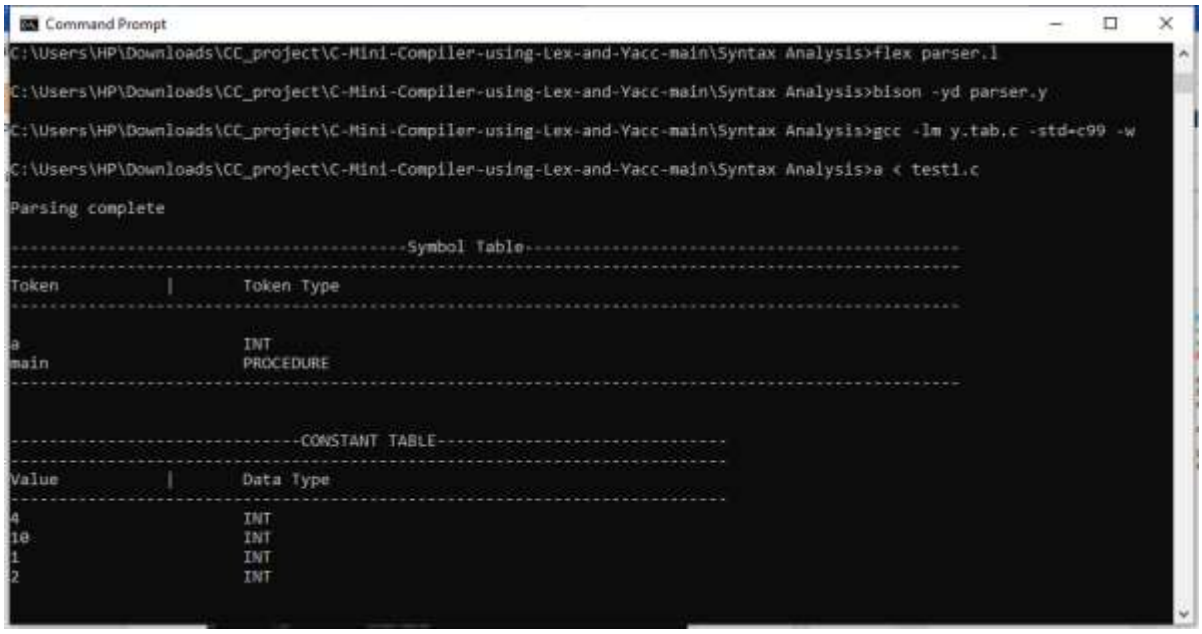
isPrime.c | test1.c

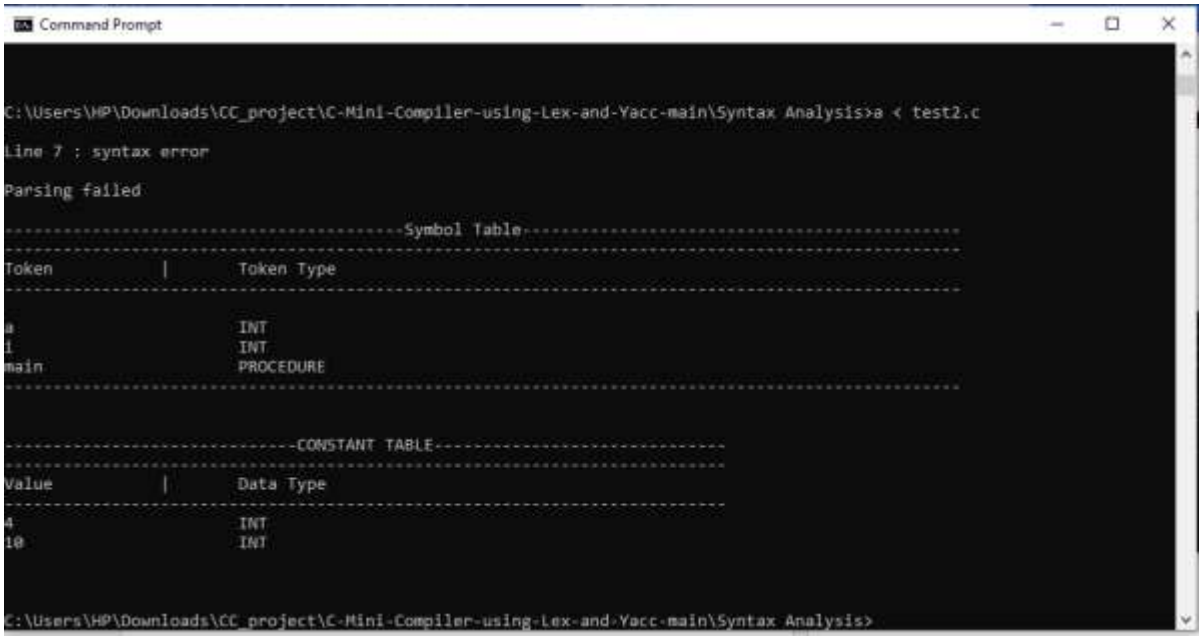
```c
//without error - nested if-else
#include<stdio.h>
#define x 3
int main()
{
    int a=4;
    if(a<10)
    {
        a = a + 1;
    }
    else
    {
        a = a + 2;
    }
    return;
}
```

isPrime.c | test1.c | test2.c

```c
//without error - while and for loop
#include<stdio.h>
#define x 3
int main()
{
    int a=4;
    int i;
    whi(a<10)
    {
        a++;
    }
    for(i=1;i<5;i++)
        a--;
}
```

Output:





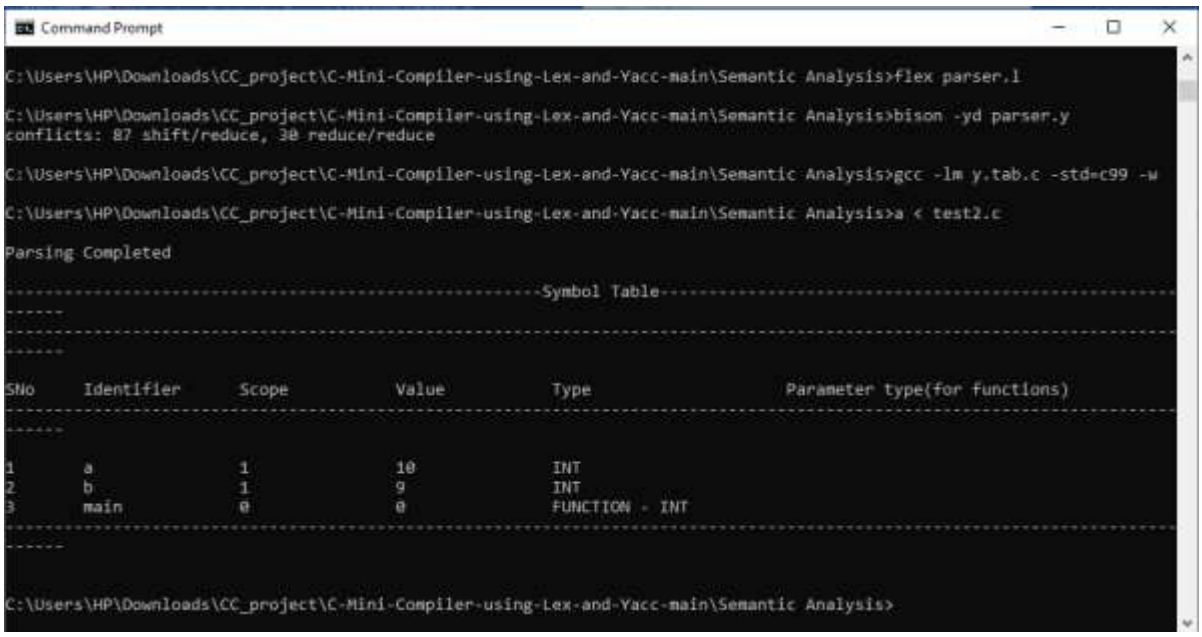
7.3 Semantic Analysis

Input:

Test case:

```c
#include <stdio.h>
int main()
{
    int a=4;
    int b=9;          //undeclared variable
    a=10;
    return 0;
}
```

Output:



7.4 Abstract Syntax Tree

Input:

Test Case: input.c

```c
#include <stdio.h>

int main()
{
    int n = 37; //check if n is prime
    int i;
    int factors = 0;
    for(i = 1; i <= n; i++)
        factors = (n % i == 0) ? factors + 1 : factors;
    int isprime = (factors == 2) ? 1 : 0;
    // if n is prime, isprime == 1, else isprime == 0
}
```

Output :

7.5 Intermediate Code Generation:

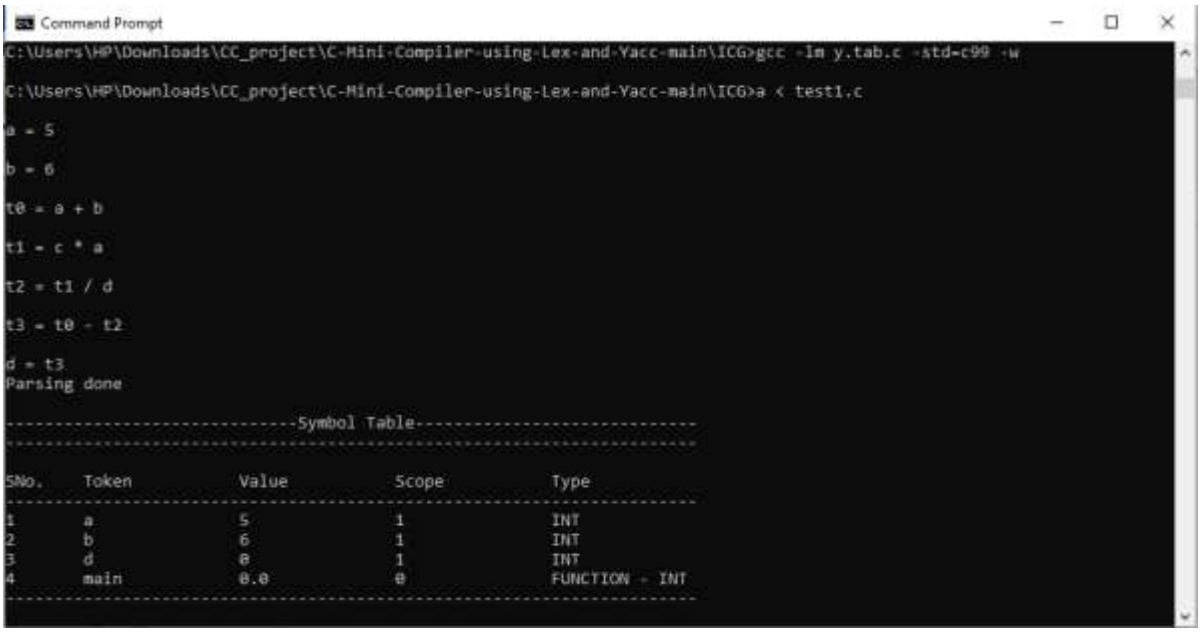
Test Cases:

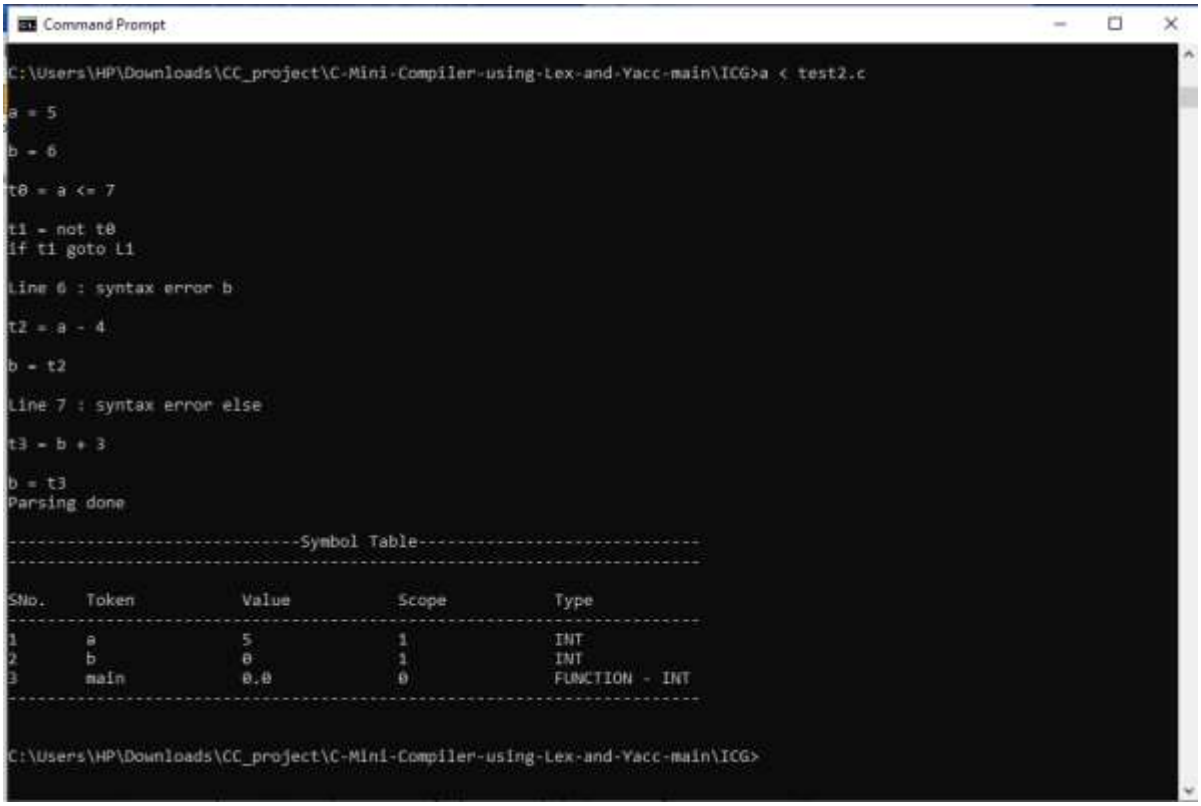
test1.c :

```c
#include <stdio.h>
int main()
{
    int a=5;
    int b=6;
    int d=a+b-c*a/d;
    return 0;
}
```

test2.c :

```c
#include <stdio.h>
int main()
{
    int a=5;
    int b=6;
    if(a<=7)
        b=b-4;
    else
        b=b+3;
    return 0;
}
```


Command Prompt
C:\Users\HP\Downloads\CC_project\C-Mini-Compiler-using-Lex-and-Yacc-main\ICG>gcc -lm y.tab.c -std=c99 -w
C:\Users\HP\Downloads\CC_project\C-Mini-Compiler-using-Lex-and-Yacc-main\ICG>a < test1.c
a = 5
b = 6
t0 = a + b
t1 = c * a
t2 = t1 / d
t3 = t0 - t2
d = t3
Parsing done
----------Symbol Table----------
SNo. Token Value Scope Type
1 a 5 1 INT
2 b 6 1 INT
3 d 0 1 INT
4 main 0.0 0 FUNCTION - INT


Command Prompt
C:\Users\HP\Downloads\CC_project\C-Mini-Compiler-using-Lex-and-Yacc-main\ICG>a < test2.c
a = 5
b = 6
t0 = a <= 7
t1 = not t0
if t1 goto L1
Line 6 : syntax error b
t2 = a - 4
b = t2
Line 7 : syntax error else
t3 = b + 3
b = t3
Parsing done
----------Symbol Table----------
SNo. Token Value Scope Type
1 a 5 1 INT
2 b 0 1 INT
3 main 0.0 0 FUNCTION - INT
C:\Users\HP\Downloads\CC_project\C-Mini-Compiler-using-Lex-and-Yacc-main\ICG>

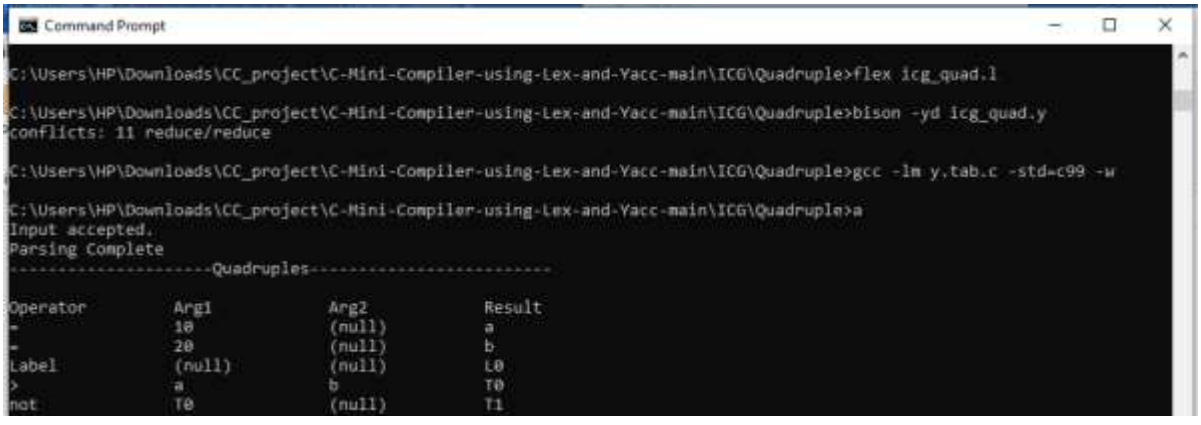
7.6 ICG in Quadruple format

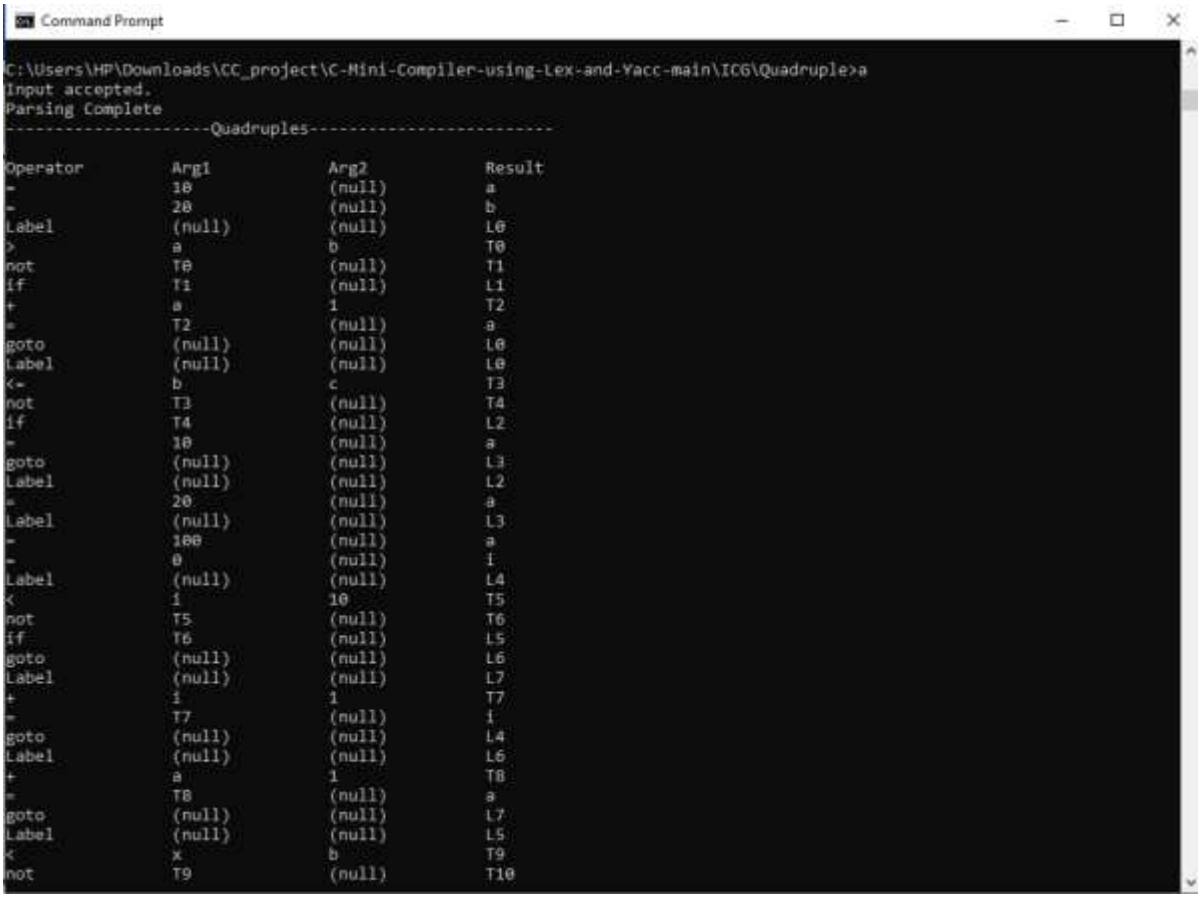
Input:

Test Case: input.c

```
#include<stdio.h>
void main()
{
        int a=10;
        int b=20;
        while( a > b ){
                a = a+1;
        }
        if( b < = c ){
                a = 10;
        }
        else{
                a = 20;
        }
        a = 100;
    for(i=0;i<10;i = i+1){
            a = a+1;
    }
    (x < b) ? x = 10 : x=11;

}
```

Output:




Command Prompt
C:\Users\HP\Downloads\CC_project\C-Mini-Compiler-using-Lex-and-Yacc-main\ICG\Quadruple>a
Input accepted.
Parsing Complete
----------Quadruples----------
Operator Arg1 Arg2 Result
= 10 (null) a
= 20 (null) b
Label (null) (null) L0
> a b T0
not T0 (null) T1
if T1 (null) L1
+ a 1 T2
= T2 (null) a
goto (null) (null) L0
Label (null) (null) L0
<= b c T3
not T3 (null) T4
if T4 (null) L2
= 10 (null) a
goto (null) (null) L3
Label (null) (null) L2
= 20 (null) a
Label (null) (null) L3
= 100 (null) a
= 0 (null) i
Label (null) (null) L4
< i 10 T5
not T5 (null) T6
if T6 (null) L5
goto (null) (null) L6
Label (null) (null) L7
+ i 1 T7
= T7 (null) i
goto (null) (null) L4
Label (null) (null) L6
+ a 1 T8
= T8 (null) a
goto (null) (null) L7
Label (null) (null) L5
< x b T9
not T9 (null) T10

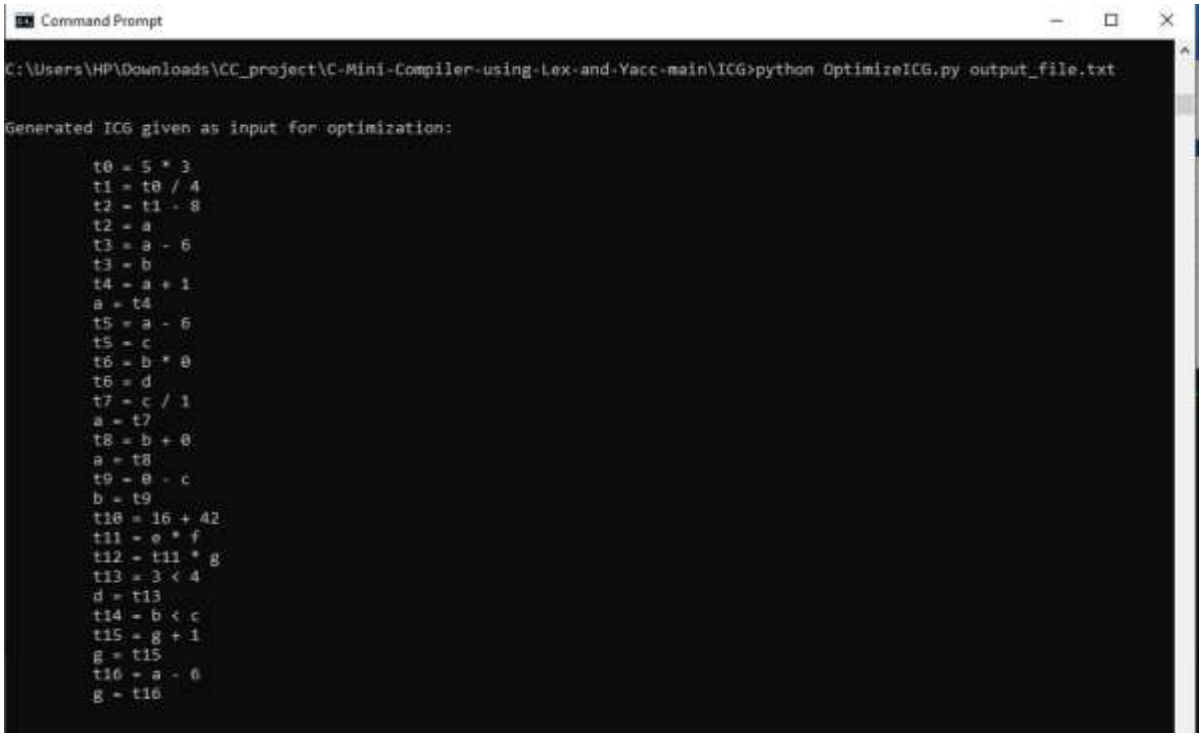
7.7 Code Optimization

Input:

Test case : output_file.txt

```
t0 = 5 * 3
t1 = t0 / 4
t2 = t1 - 8
t2 = a
t3 = a - 6
t3 = b
t4 = a + 1
a = t4
t5 = a - 6
t5 = c
t6 = b * 0
t6 = d
t7 = c / 1
a = t7
t8 = b + 0
a = t8
t9 = 0 - c
b = t9
t10 = 16 + 42
t11 = e * f
t12 = t11 * g
t13 = 3 < 4
d = t13
t14 = b < c
t15 = g + 1
g = t15
t16 = a - 6
g = t16
```

Output:


Command Prompt
C:\Users\HP\Downloads\CC_project\C-Mini-Compiler-using-Lex-and-Yacc-main\ICG>python OptimizeICG.py output_file.txt
Generated ICG given as input for optimization:
t0 = 5 * 3
t1 = t0 / 4
t2 = t1 - 8
t2 = a
t3 = a - 6
t3 = b
t4 = a + 1
a = t4
t5 = a - 6
t5 = c
t6 = b * 0
t6 = d
t7 = c / 1
a = t7
t8 = b + 0
a = t8
t9 = 0 - c
b = t9
t10 = 16 + 42
t11 = e * f
t12 = t11 * g
t13 = 3 < 4
d = t13
t14 = b < c
t15 = g + 1
g = t15
t16 = a - 6
g = t16

Command Prompt

```
ICG after eliminating common subexpressions:

        t0 = 5 * 3
        t1 = t0 / 4
        t2 = t1 - 8
        t2 = a
        t3 = a - 6
        t3 = b
        t4 = a + 1
        a = t4
        t5 = t3
        t5 = c
        t6 = b * 0
        t6 = d
        t7 = c / 1
        a = t7
        t8 = b + 0
        a = t8
        t9 = 0 - c
        b = t9
        t10 = 16 + 42
        t11 = e * f
        t12 = t11 * g
        t13 = 3 < 4
        d = t13
        t14 = b < c
        t15 = g + 1
        g = t15
        t16 = t3
        g = t16
```

```
ICG after constant folding:

        t0 = 15
        t1 = t0 / 4
        t2 = t1 - 8
        t2 = a
        t3 = a - 6
        t3 = b
        t4 = a + 1
        a = t4
        t5 = t3
        t5 = c
        t6 = 0
        t6 = d
        t7 = c
        a = t7
        t8 = b
        a = t8
        t9 = -c
        b = t9
        t10 = 58
        t11 = e * f
        t12 = t11 * g
        t13 = True
        d = t13
        t14 = b < c
        t15 = g + 1
        g = t15
        t16 = t3
        g = t16
```

```
Optimized ICG after dead code elimination:

        t3 = a - 6
        t3 = b
        t4 = a + 1
        a = t4
        t7 = c
        a = t7
        t8 = b
        a = t8
        t9 = -c
        b = t9
        t13 = True
        d = t13
        t15 = g + 1
        g = t15
        t16 = t3
        g = t16


Optimization done by eliminating 12 lines.


C:\Users\HP\Downloads\CC_project\C-Mini-Compiler-using-Lex-and-Yacc-main\ICG>
```

CONCLUSION:

The lexical analyzer, syntax analyzer and the semantic analyzer for a subset of C language, which include selection statements, compound statements, iteration statements (for, while and do-while) and user defined functions are generated. It is important to define unambiguous grammar in the syntax analysis phase.

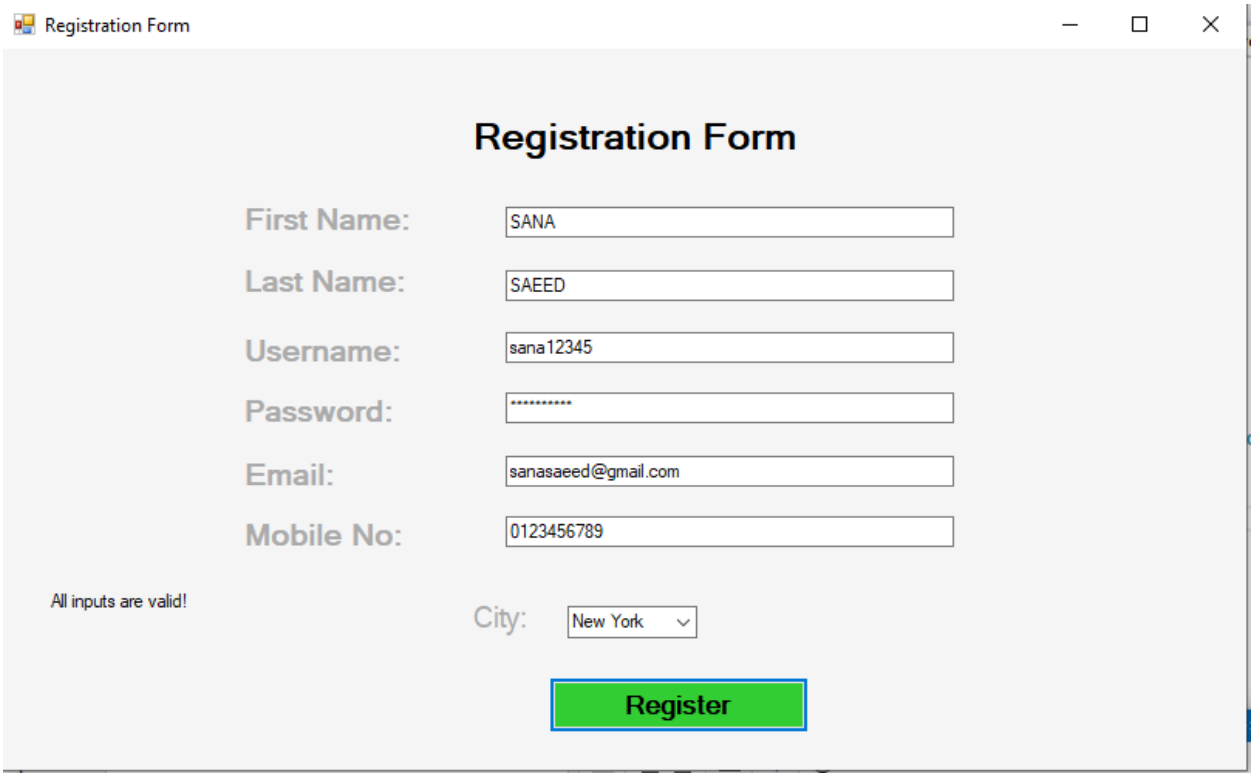
The semantic analyzer performs type checking, reports various errors such as undeclared variable, type mismatch, errors in function call (number and datatypes of parameters mismatch) and errors in array indexing.

The syntax analyzer for the C language by writing two scripts, one that acts as a lexical analyzer (lexer) and outputs a stream of tokens, and the other one that acts as a parser. The Syntax analyzer generates the various statements and expressions required based on the context free grammar defined. Parse trees for various statements and expressions are generated by the syntax analyzer.

The Intermediate Code Generation phase involves the execution of lex and yacc codes. Once the parsing is complete, if errors are encountered then the errors are displayed along with the line numbers. Along with this, the updated symbol table is displayed.

Question 3

Create and implement RE and DFAs for the form below



RE and NFA to DFA:

The regular expressions, construct their NFAs (with lambda transitions), and then convert them to DFAs.

q0 is the starting state.

First Name and Last Name

Regular Expression: ^[A-Za-z]{1,50}$

DFA:

q0 --[A-Za-z]--> q1 --[A-Za-z]--> q2 --[A-Za-z]--> ... --[A-Za-z]--> q50

Username

Regular Expression: ^[A-Za-z0-9_.]{5,15}$

DFA:

q0 --[A-Za-z0-9_.]--> q1 --[A-Za-z0-9_.]--> q2 --[A-Za-z0-9_.]--> ... --[A-Za-z0-9_.]--> q15

Password

Regular Expression:

^(?=.*[A-Za-z])(?=.*\d)(?=.*[@$!%*?&])[A-Za-z\d@$!%*?&]{8,}$

DFA:

q0--[A-Za-z\d@$!%*?&]--> q1 --[A-Za-z\d@$!%*?&]--> ... --[A-Za-z\d@$!%*?&]--> q8

Email

Regular Expression: ^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$

DFA:

q0 --[a-zA-Z0-9._%+-]+--> q1 --[@]--> q2 --[a-zA-Z0-9.-]+--> q3 --[.]--> q4 --[a-zA-Z]{2,}--> q5

Mobile No

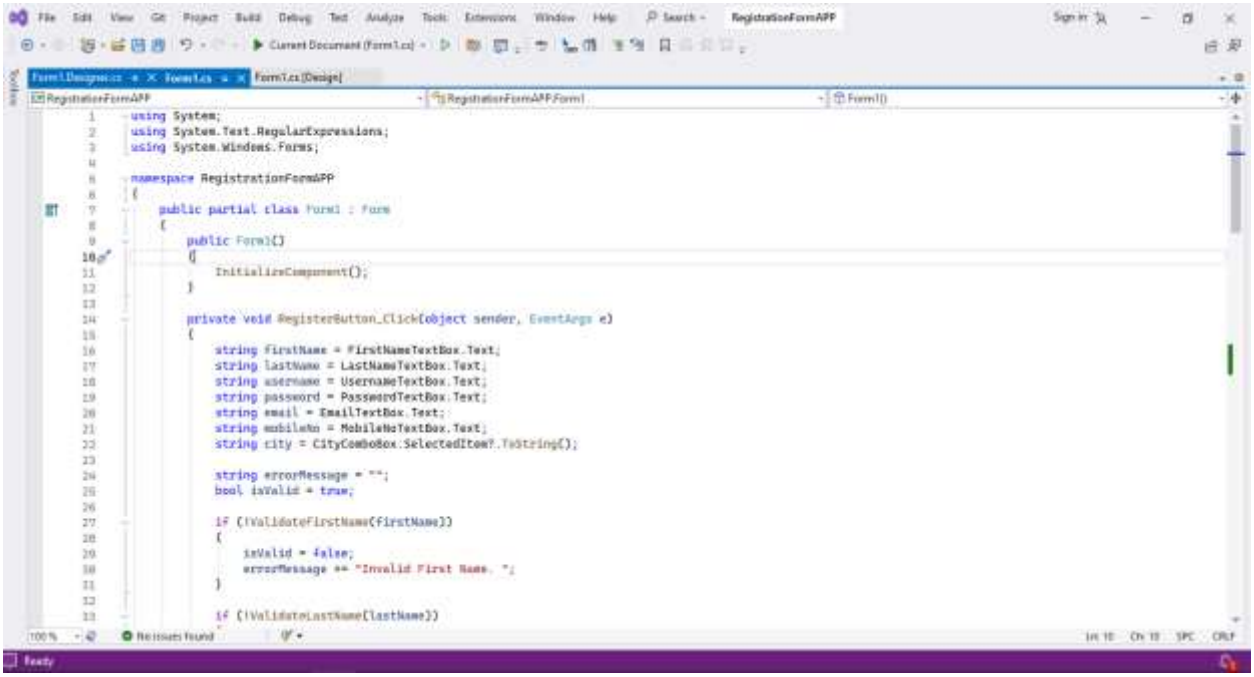
Regular Expression: ^\d{10}$

DFA:

q0 --[\d]--> q1 --[\d]--> q2 --[\d]--> ... --[\d]--> q10

Creating the NFAs from Res and converting them to DFAs manually for each regular expression is feasible but complex and tedious. Tools like regex to DFA converters can simplify this process significantly.

Code:



Written code:

```
using System;
using System.Text.RegularExpressions;
using System.Windows.Forms;

namespace RegistrationFormAPP
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void RegisterButton_Click(object sender, EventArgs e)
        {
            string firstName = FirstNameTextBox.Text;
            string lastName = LastNameTextBox.Text;
            string username = UsernameTextBox.Text;
            string password = PasswordTextBox.Text;
            string email = EmailTextBox.Text;
            string mobileNo = MobileNoTextBox.Text;
            string city = CityComboBox.SelectedItem?.ToString();

            string errorMessage = "";
```

```
    bool isValid = true;

    if (!ValidateFirstName(firstName))
    {
        isValid = false;
        errorMessage += "Invalid First Name. ";
    }

    if (!ValidateLastName(lastName))
    {
        isValid = false;
        errorMessage += "Invalid Last Name. ";
    }

    if (!ValidateUsername(username))
    {
        isValid = false;
        errorMessage += "Invalid Username. ";
    }

    if (!ValidatePassword(password))
    {
        isValid = false;
        errorMessage += "Invalid Password. ";
    }

    if (!ValidateEmail(email))
    {
        isValid = false;
        errorMessage += "Invalid Email. ";
    }

    if (!ValidateMobileNo(mobileNo))
    {
        isValid = false;
        errorMessage += "Invalid Mobile No. ";
    }

    if (!ValidateCity(city))
    {
        isValid = false;
        errorMessage += "Invalid City. ";
    }

    ResultLabel.Text = isValid ? "All inputs are valid!" : errorMessage;
}

private bool ValidateFirstName(string firstName)
{
    return Regex.IsMatch(firstName, @"^[A-Za-z]{1,50}$");
}

private bool ValidateLastName(string lastName)
{
    return Regex.IsMatch(lastName, @"^[A-Za-z]{1,50}$");
}

private bool ValidateUsername(string username)
```

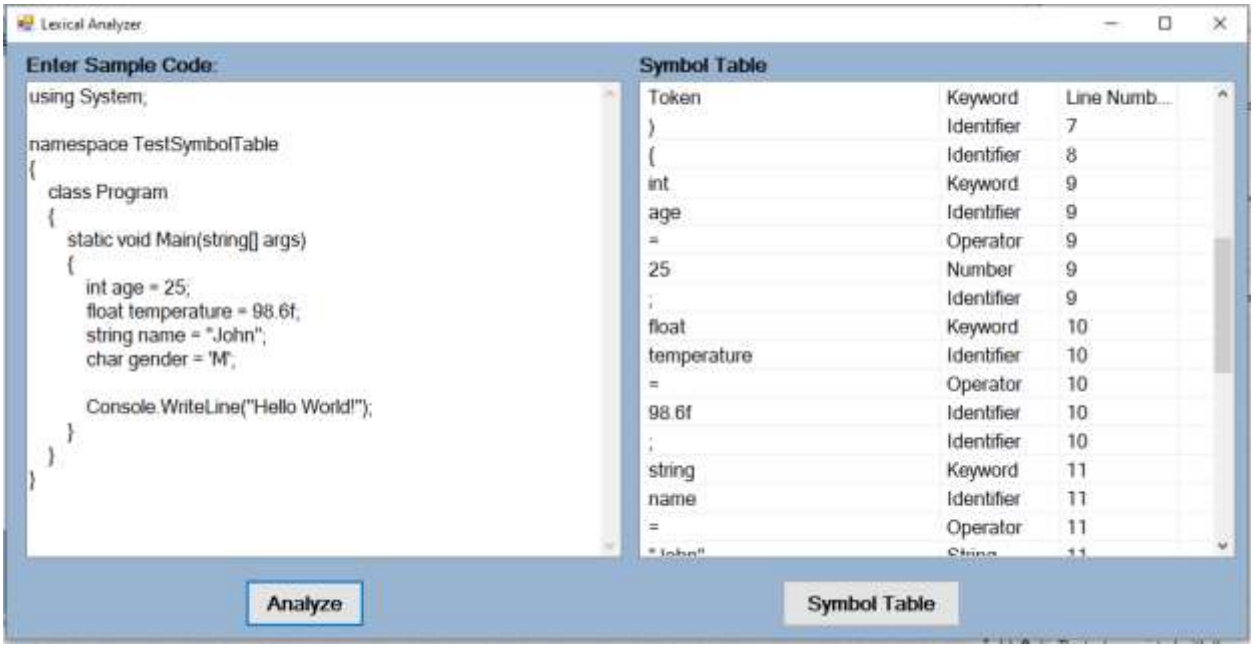
```
        {
            return Regex.IsMatch(username, @"^[A-Za-z0-9_.]{5,15}$");
        }

        private bool ValidatePassword(string password)
        {
            return Regex.IsMatch(password, @"^(?=.*[A-Za-
z])(?=.*\d)(?=.*[@$!%*?&])[A-Za-z\d@$!%*?&]{8,}$");
        }

        private bool ValidateEmail(string email)
        {
            return Regex.IsMatch(email, @"^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-
Z]{2,}$");
        }


        private bool ValidateMobileNo(string mobileNo)
        {
            return Regex.IsMatch(mobileNo, @"^\d{10}$");
        }

        private bool ValidateCity(string city)
        {
            return city != null && city != "Select";
        }
    }
}
```

Question 4:

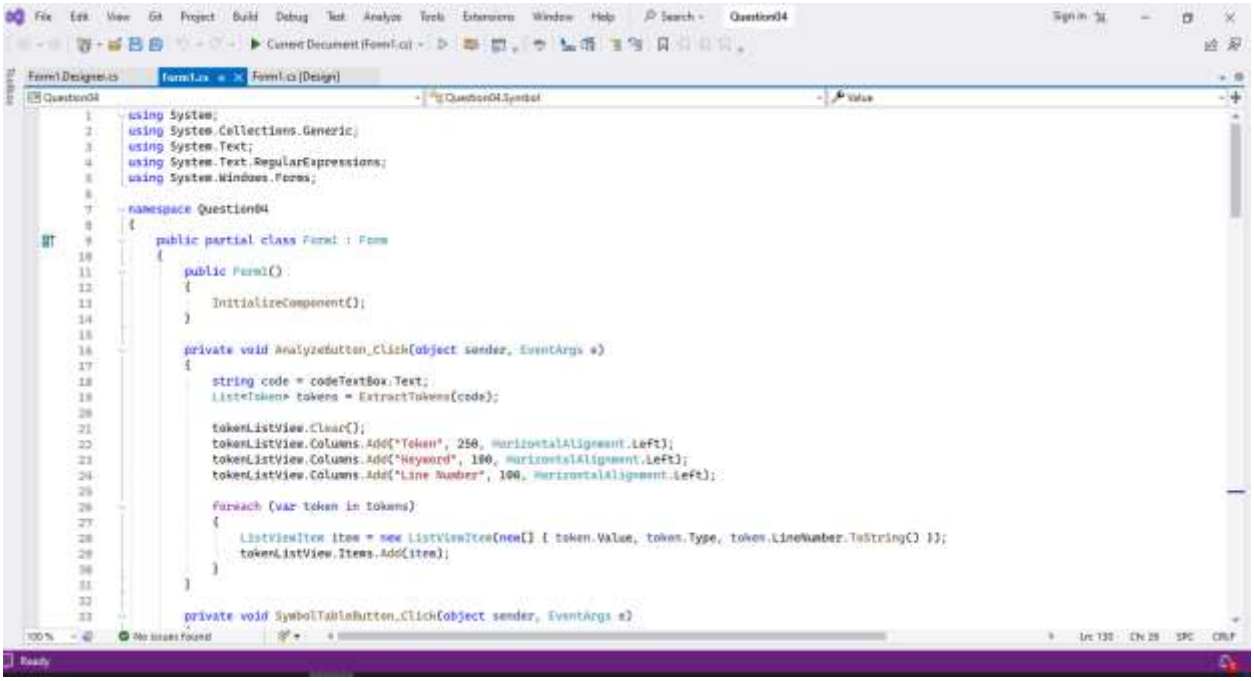
Write a program which generates symbol table for the code you submitted in question 3

Screen:

We can add any code as input and it will generate symbol table of it.



Code:



Written Code:

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

namespace Question04
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void AnalyzeButton_Click(object sender, EventArgs e)
        {
            string code = codeTextBox.Text;
            List<Token> tokens = ExtractTokens(code);

            tokenListView.Clear();
            tokenListView.Columns.Add("Token", 250, HorizontalAlignment.Left);
            tokenListView.Columns.Add("Keyword", 100, HorizontalAlignment.Left);
            tokenListView.Columns.Add("Line Number", 100, HorizontalAlignment.Left);

            foreach (var token in tokens)
            {
```

```
                ListViewItem item = new ListViewItem(new[] { token.Value,
token.Type, token.LineNumber.ToString() });
                tokenListView.Items.Add(item);
            }
        }

        private void SymbolTableButton_Click(object sender, EventArgs e)
        {
            string code = codeTextBox.Text;
            List<Token> tokens = ExtractTokens(code);

            Dictionary<string, Symbol> symbolTable = CreateSymbolTable(tokens);

            StringBuilder tableDisplay = new StringBuilder();
            tableDisplay.AppendLine($"+{"",-30} + {"",-25} + {"",-15}+");
            tableDisplay.AppendLine($"| {"Identifier",-30} | {"Data Type",-25} |
{"Value",-15}|");
            tableDisplay.AppendLine($"+{"",-30} + {"",-25} + {"",-15}+");
            tableDisplay.AppendLine(new string('-', 76));

            foreach (var entry in symbolTable)
            {
                tableDisplay.AppendLine($"| {entry.Key,-30} |
{entry.Value.DataType,-25} | {entry.Value.Value,-15}|");
            }

            tableDisplay.AppendLine($"+{"",-30} + {"",-25} + {"",-15}+");
            MessageBox.Show(tableDisplay.ToString(), "Symbol Table");
        }

        private List<Token> ExtractTokens(string code)
        {
            var tokens = new List<Token>();
            var lines = code.Split('\n');
            int lineNumber = 1;

            foreach (var line in lines)
            {
                var words = Regex.Split(line, @"(\(|\)|\s+|\t|{|}|;|,|\+|\-
|\*|\/|%|=|<|>|!|&|\||\^|~)");

                foreach (var word in words)
                {
                    if (!string.IsNullOrWhiteSpace(word))
                    {
                        string type = "Identifier";
                        if (Regex.IsMatch(word, @"^\d+$")) type = "Number";
                        else if (Regex.IsMatch(word, @"^[\+\-
\*\/\%\=\<\>\!\&\|\^\~]+$")) type = "Operator";
                        else if (Regex.IsMatch(word,
@"^(if|else|return|int|for|switch|case|while|do|float|double|string|char)$")) type =
"Keyword";
                        else if (Regex.IsMatch(word, @"^"".*""$") ||
Regex.IsMatch(word, @"^'.*'$")) type = "String";

                        tokens.Add(new Token(word, type, lineNumber));
                    }
                }
```

```
                lineNumber++;
            }

            return tokens;
        }


        private Dictionary<string, Symbol> CreateSymbolTable(List<Token> tokens)
        {
            var symbolTable = new Dictionary<string, Symbol>();
            string currentType = null;

            foreach (var token in tokens)
            {
                if (token.Type == "Keyword" && (token.Value == "int" || token.Value
== "float" || token.Value == "double" || token.Value == "string" || token.Value ==
"char"))
                {
                    currentType = token.Value;
                }
                else if (currentType != null && token.Type == "Identifier")
                {
                    symbolTable[token.Value] = new Symbol(currentType, "N/A");
                    currentType = null;
                }
            }

            return symbolTable;
        }

        private void label2_Click(object sender, EventArgs e)
        {

        }
    }

    public class Token
    {
        public string Value { get; }
        public string Type { get; }
        public int LineNumber { get; }

        public Token(string value, string type, int lineNumber)
        {
            Value = value;
            Type = type;
            LineNumber = lineNumber;
        }
    }

    public class Symbol
    {
        public string DataType { get; }
        public string Value { get; }

        public Symbol(string dataType, string value)
        {
```

```
            DataType = dataType;
            Value = value;

        }
    }
}
```