Lab 4

Activity 1:

Implement lexical analyzer using input buffering scheme

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Text.RegularExpressions;
using System.Threading.Tasks;
using System.Windows.Forms;
using System.Collections;
namespace LexicalAnalyzerV1
{
public partial class Form1 : Form
{
public Form1()
{
InitializeComponent();
}
private void btn_Input_Click(object sender, EventArgs e)
{
//taking user input from rich textbox
String userInput = tfInput.Text;
//List of keywords which will be used to seperate keywords
from variables
List<String> keywordList = new List<String>();
keywordList.Add("int");
keywordList.Add("float");
keywordList.Add("while");
keywordList.Add("main");
keywordList.Add("if");
keywordList.Add("else");
keywordList.Add("new");
//row is an index counter for symbol table
int row = 1;
//count is a variable to incremenet variable id in tokens
int count = 1;26
//line_num is a counter for lines in user input
int line_num = 0;
//SymbolTable is a 2D array that has the following
structure
//[Index][Variable Name][type][value][line#]
//rows are incremented with each variable information entry
String[,] SymbolTable = new String[20, 6];
List<String> varListinSymbolTable = new List<String>();
//Input Buffering
ArrayList finalArray = new ArrayList();
ArrayList finalArrayc = new ArrayList();
ArrayList tempArray = new ArrayList();
char[] charinput = userInput.ToCharArray();
```

```
//Regular Expression for Variables
Regex variable_Reg = new Regex(@"^[A-Za-z|_][A-Za-z|0-
9]*$");
//Regular Expression for Constants
Regex constants_Reg = new Regex(@"^[0-9]+([.][0-
9]+)?([e]([+|-])?[0-9]+)?$");
//Regular Expression for Operators
Regex operators_Reg = new Regex(@"^[-*+/><&&||=]$");
//Regular Expression for Special_Characters
Regex Special_Reg = new Regex(@"^[.,'\[\]{}();:?]$");
for (int itr = 0; itr < charinput.Length; itr++)
{
Match Match_Variable =
variable_Reg.Match(charinput[itr] + "");
Match Match_Constant =
constants_Reg.Match(charinput[itr] + "");
Match Match_Operator =
operators_Reg.Match(charinput[itr] + "");
Match Match_Special = Special_Reg.Match(charinput[itr]
+ "");
if (Match_Variable.Success || Match_Constant.Success ||
Match_Operator.Success || Match_Special.Success ||
charinput[itr].Equals(' '))
{
tempArray.Add(charinput[itr]);
}
if (charinput[itr].Equals('\n'))
{
if (tempArray.Count != 0)
{
int j = 0;
String fin = "";
for (; j < tempArray.Count; j++)
{27
fin += tempArray[j];
}
finalArray.Add(fin);
tempArray.Clear();
}
}
}
if (tempArray.Count != 0)
{
int j = 0;
String fin = "";
for (; j < tempArray.Count; j++)
{
fin += tempArray[j];
}
finalArray.Add(fin);
tempArray.Clear();
}
// Final Array SO far correct
tfTokens.Clear();
symbolTable.Clear();
```

```
//looping on all lines in user input
for (int i = 0; i < finalArray.Count; i++)
{
String line = finalArray[i].ToString();
//tfTokens.AppendText(line + "\n");
char[] lineChar = line.ToCharArray();
line_num++;
//taking current line and splitting it into lexemes by
space
for (int itr = 0; itr < lineChar.Length; itr++)
{
Match Match_Variable =
variable_Reg.Match(lineChar[itr] + "");
Match Match_Constant =
constants_Reg.Match(lineChar[itr] + "");
Match Match_Operator =
operators_Reg.Match(lineChar[itr] + "");
Match Match_Special =
Special_Reg.Match(lineChar[itr] + "");
if (Match_Variable.Success ||
Match_Constant.Success)
{
tempArray.Add(lineChar[itr]);
}
if (lineChar[itr].Equals(' '))
{
if (tempArray.Count != 0)28
{
int j = 0;
String fin = "";
for (; j < tempArray.Count; j++)
{
fin += tempArray[j];
}
finalArrayc.Add(fin);
tempArray.Clear();
}
}
if (Match_Operator.Success ||
Match_Special.Success)
{
if (tempArray.Count != 0)
{
int j = 0;
String fin = "";
for (; j < tempArray.Count; j++)
{
fin += tempArray[j];
}
finalArrayc.Add(fin);
tempArray.Clear();
}
finalArrayc.Add(lineChar[itr]);
}
}
```

```
if (tempArray.Count != 0)
{
String fina = "";
for (int k = 0; k < tempArray.Count; k++)
{
fina += tempArray[k];
}
finalArrayc.Add(fina);
tempArray.Clear();
}
// we have asplitted line here
for (int x = 0; x < finalArrayc.Count; x++)
{
Match operators =
operators_Reg.Match(finalArrayc[x].ToString());
Match variables =
variable_Reg.Match(finalArrayc[x].ToString());
Match digits =
constants_Reg.Match(finalArrayc[x].ToString());
Match punctuations =
Special_Reg.Match(finalArrayc[x].ToString());29
if (operators.Success)
{
// if a current lexeme is an operator then
make a token e.g. < op, = >
tfTokens.AppendText("< op, " +
finalArrayc[x].ToString() + "> ");
}
else if (digits.Success)
{
// if a current lexeme is a digit then make
a token e.g. < digit, 12.33 >
tfTokens.AppendText("< digit, " +
finalArrayc[x].ToString() + "> ");
}
else if (punctuations.Success)
{
// if a current lexeme is a punctuation
then make a token e.g. < punc, ; >
tfTokens.AppendText("< punc, " +
finalArrayc[x].ToString() + "> ");
}
else if (variables.Success)
{
// if a current lexeme is a variable and
not a keyword
if
(!keywordList.Contains(finalArrayc[x].ToString())) // if it is
not a
keyword
{
// check what is the category of
varaible, handling only two cases here
//Category1- Variable initialization of
type digit e.g. int count = 10 ;
```

```
//Category2- Variable initialization of
type String e.g. String var = ' Hello ' ;
Regex reg1 = new
Regex(@"^(int|String|float|double)\s([A-Za-z|_][A-Za-z|0-
9]{0,10})\s(=)\s([0-9]+([.][0-9]+)?([e][+|-]?[0-9]+)?)\s(;)$");
// line
of type int alpha = 2 ;
Match category1 = reg1.Match(line);
Regex reg2 = new
Regex(@"^(String|char)\s([A-Za-z|_][A-Za-z|0-
9]{0,10})\s(=)\s[']\s([A
Za-z|_][A-Za-z|0-9]{0,30})\s[']\s(;)$"); // line of type
String alpha =
' Hello ' ;
Match category2 = reg2.Match(line);
//if it is a category 1 then add a row
in symbol table containing the information related to that
variable
if (category1.Success)30
{
SymbolTable[row, 1] =
row.ToString(); //index
SymbolTable[row, 2] =
finalArrayc[x].ToString(); //variable name
SymbolTable[row, 3] = finalArrayc[x
- 1].ToString(); //type
SymbolTable[row, 4] =
finalArrayc[x+2].ToString(); //value
SymbolTable[row, 5] =
line_num.ToString(); // line number
tfTokens.AppendText("<var" + count
+ ", " + row + "> ");
symbolTable.AppendText(SymbolTable[row, 1].ToString() + " \t
");
symbolTable.AppendText(SymbolTable[row, 2].ToString() + " \t
");
symbolTable.AppendText(SymbolTable[row, 3].ToString() + " \t
");
symbolTable.AppendText(SymbolTable[row, 4].ToString() + " \t
");
symbolTable.AppendText(SymbolTable[row, 5].ToString() + " \n
");
row++;
count++;
}
//if it is a category 2 then add a row
in symbol table containing the information related to that
variable
else if (category2.Success)
{
// if a line such as String var =
' Hello ' ; comes and the loop moves to index of array
containing Hello
,
//then this if condition prevents
```

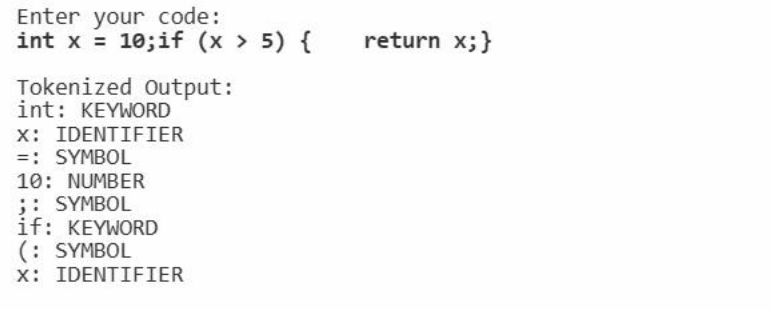
```
addition of Hello in symbol Table because it is not a variable
it is
just a string
if (!(finalArrayc[x-
1].ToString().Equals("'") &&
finalArrayc[x+1].ToString().Equals("'")))
{
SymbolTable[row, 1] =
row.ToString(); // index
SymbolTable[row, 2] =
finalArrayc[x].ToString(); //varname
SymbolTable[row, 3] =
finalArrayc[x-1].ToString(); //type31
SymbolTable[row, 4] =
finalArrayc[x+3].ToString(); //value
SymbolTable[row, 5] =
line_num.ToString(); // line number
tfTokens.AppendText("<var" +
count + ", " + row + "> ");
symbolTable.AppendText(SymbolTable[row, 1].ToString() + " \t
");
symbolTable.AppendText(SymbolTable[row, 2].ToString() + " \t
");
symbolTable.AppendText(SymbolTable[row, 3].ToString() + " \t
");
symbolTable.AppendText(SymbolTable[row, 4].ToString() + " \t
");
symbolTable.AppendText(SymbolTable[row, 5].ToString() + " \n
");
row++;
count++;
}
else
{
tfTokens.AppendText("<String" +
count + ", " + finalArrayc[x].ToString() + "> ");
}
}
else
{
// if any other category line comes
in we check if we have initializes that varaible before,
// if we have initiazed it before
then we put the index of that variable in symbol table, in its
token
String ind = "Default";
String ty = "Default";
String val = "Default";
String lin = "Default";
for (int r = 1; r <=
SymbolTable.GetLength(0); r++)
{
//search in the symbol table if
variable entry already exists
if (SymbolTable[r,
2].Equals(finalArrayc[x].ToString()))
{
ind = SymbolTable[r, 1];
ty = SymbolTable[r, 3];
val = SymbolTable[r, 4];
lin = SymbolTable[r, 5];
tfTokens.AppendText("<var"
+ ind + ", " + ind + "> ");32
break;
}
}
}
}
// if a current lexeme is not a variable
but a keyword then make a token such as: <keyword, int>
else
{
tfTokens.AppendText("<keyword, " +
finalArrayc[x].ToString() + "> ");
}
}
}
tfTokens.AppendText("\n");
finalArrayc.Clear();
}
}
}
}
```

Graded lab task 1

```
using System;
using System.Collections.Generic;

class LexicalAnalyzer
{
   static readonly HashSet<string> Keywords = new HashSet<string> { "if", "else", "while", "return",
"int", "float" };
   static readonly HashSet<char> Operators = new HashSet<char> { '+', '-', '*', '/', '=', '<', '>', '!' };
   static readonly HashSet<char> Separators = new HashSet<char> { ';', ',', '(', ')', '{', '}' };

   private const int BufferSize = 16;
   private char[] buffer1 = new char[BufferSize];
   private char[] buffer2 = new char[BufferSize];
   private bool useFirstBuffer = true;
   private int bufferIndex = 0;
   private int bufferLength = 0;
   private string inputCode;

   public LexicalAnalyzer(string code)
   {
      inputCode = code;
      FillBuffer();
   }

   private void FillBuffer()
   {
      int length = Math.Min(BufferSize, inputCode.Length);
      for (int i = 0; i < length; i++)
         buffer1[i] = inputCode[i];

      bufferLength = length;
      bufferIndex = 0;
   }

   private char? GetNextChar()
   {
      if (bufferIndex >= bufferLength)
         return null;

      char ch = useFirstBuffer ? buffer1[bufferIndex] : buffer2[bufferIndex];
      bufferIndex++;

      return ch;
   }

   public List<(string, string)> Tokenize()
   {
      List<(string, string)> tokens = new List<(string, string)>();
      string currentToken = "";
      char? ch = GetNextChar();

      while (ch != null)
      {
         if (char.IsWhiteSpace(ch.Value))
         {
            if (currentToken.Length > 0)
            {
               tokens.Add(ClassifyToken(currentToken));
               currentToken = "";
            }
         }
         else if (Operators.Contains(ch.Value) || Separators.Contains(ch.Value))
         {
            if (currentToken.Length > 0)
            {
               tokens.Add(ClassifyToken(currentToken));
               currentToken = "";
            }
            tokens.Add((ch.Value.ToString(), "SYMBOL"));
         }
         else
         {
            currentToken += ch.Value;
```

```
            }
            ch = GetNextChar();
        }

        if (currentToken.Length > 0)
            tokens.Add(ClassifyToken(currentToken));

        return tokens;
    }

    private (string, string) ClassifyToken(string token)
    {
        if (Keywords.Contains(token)) return (token, "KEYWORD");
        if (char.IsDigit(token[0])) return (token, "NUMBER");
        return (token, "IDENTIFIER");
    }
}

class Program
{
    static void Main()
    {
        Console.WriteLine("Enter your code:");
        string inputCode = Console.ReadLine();

        LexicalAnalyzer lexer = new LexicalAnalyzer(inputCode);
        var tokens = lexer.Tokenize();

        Console.WriteLine("\nTokenized Output:");
        foreach (var (token, type) in tokens)
            Console.WriteLine($"{token}: {type}");
    }
```

Output:



```
Enter your code:
int x = 10;if (x > 5) {    return x;}

Tokenized Output:
int: KEYWORD
x: IDENTIFIER
=: SYMBOL
10: NUMBER
;: SYMBOL
if: KEYWORD
(: SYMBOL
x: IDENTIFIER
```