Lab 04

Lexical Analyzer: Input Buffering scheme

Objective:

This lab is designed to demonstrate the implementation of lexical analyzer using buffering scheme.

Activity Outcomes:

On completion of this lab, students will be able to:

- implement lexical analyzer using input buffering scheme

Instructor Note:

As for this lab activity, read chapter 02 from the book "Compilers:Principles, Techniques and Tools" by Ullman Sethi.

1) Useful Concepts

In this lab, we implement the lexical analyzer using buffering scheme. Lexical analyzer reads source code character by character and produces tokens for each valid word. Specialized buffering techniques thus have been developed to reduce the amount of overhead required to process a single input character.
Two pointers to the input are maintained:

Pointer *Lexeme Begin*, marks the beginning of the current lexeme, whose extent we are attempting to determine

Pointer *Forward*, scans ahead until a pattern match is found.

Once the next lexeme is determined, *forward* is set to character at its right end.Then, after the lexeme is recorded as an attribute value of a token returned to the parser, *Lexeme Begin* is set to the character immediately after the lexeme just found.

If we use the scheme of Buffer pairs we must check, each time we advance forward, that we have not moved off one of the buffers; if we do, then we must also reload the other buffer. Thus, for each character read, we make two tests: one for the end of the buffer, and one to determine what character is read (the latter may be a multiway branch). We can combine the buffer-end test with the test for the current character if we extend each buffer to hold a sentinel character at the end. The sentinel is a special character that cannot be part of the source program, and a natural choice is the character **EOF.**

Note that **EOF** retains its use as a marker for the end of the entire input. Any **EOF** that appears other than at the end of a buffer means that the input is at an end.

2) Solved Lab Activities

Sr.No	Allocated Time	Level of Complexity	CLO Mapping
Activity 1	*45 mins*	*Medium*	*CLO-5*

Activity 1:
Implement lexical analyzer using input buffering scheme

Solution:

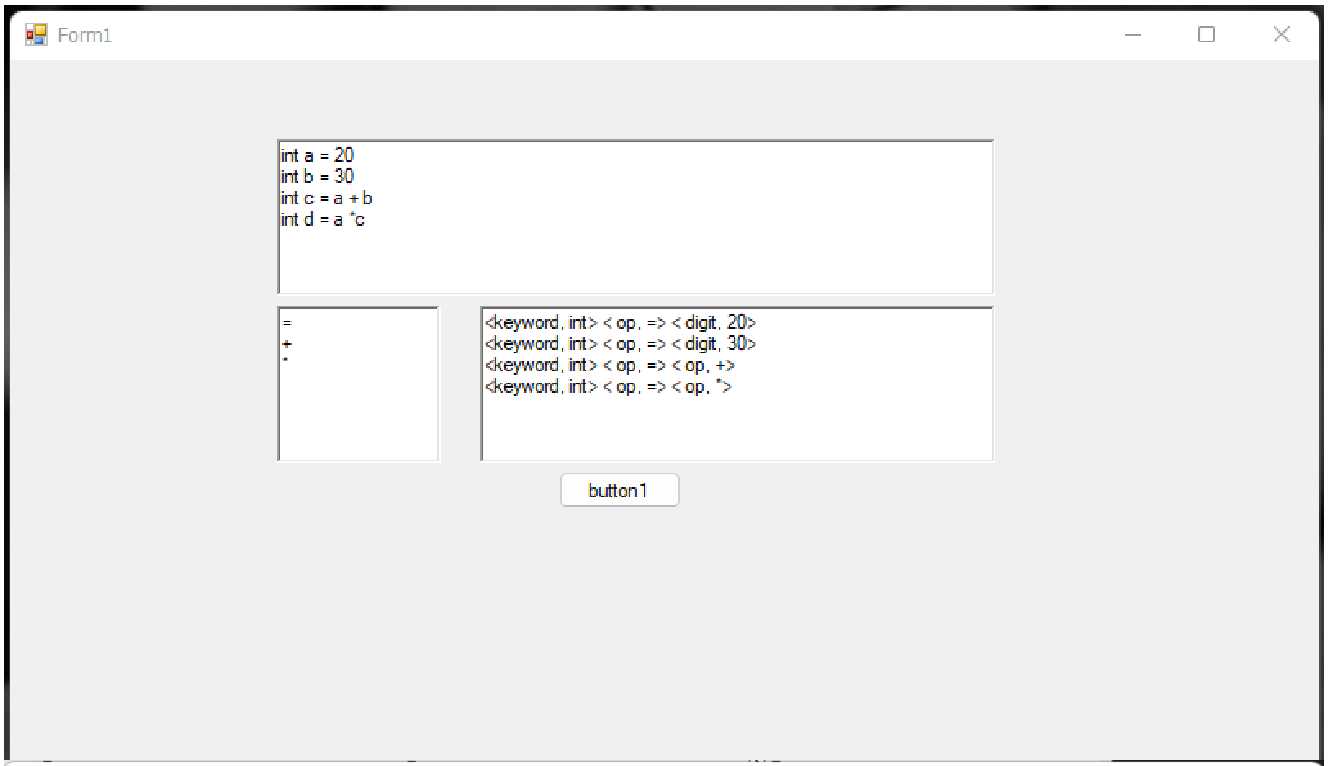
```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Text.RegularExpressions;
using System.Threading.Tasks;
using System.Windows.Forms;
using System.Collections;
namespace LexicalAnalyzerV1
{
    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;
            //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++)
                        {
                             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)
```

```
                        {
                            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());
```

```
                        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)
                                {
                                    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(); //type
```

```
                                        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 + "> ");
```

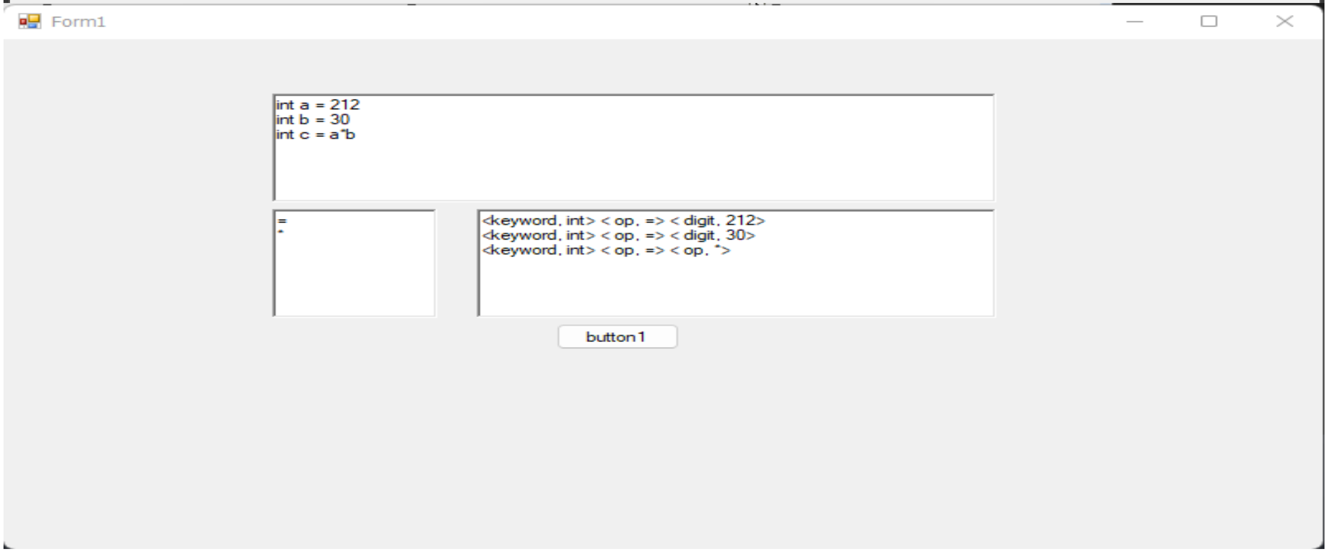
```
                                                break;
                                            }
                                        }
                                    }

                                }
                                // 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();
            }
        }
    }
 }
```



```
int a = 20
int b = 30
int c = a + b
int d = a *c
```

```
=
+
*
```

```
<keyword, int> < op, => < digit, 20>
<keyword, int> < op, => < digit, 30>
<keyword, int> < op, => < op, +>
<keyword, int> < op, => < op, *>
```



```
int a = 212
int b = 30
int c = a*b
```

```
=
*
```

```
<keyword, int> < op, => < digit, 212>
<keyword, int> < op, => < digit, 30>
<keyword, int> < op, => < op, *>
```

3) Graded Lab Tasks

Note: The instructor can design graded lab activities according to the level of difficult and complexity of the solved lab activities. The lab tasks assigned by the instructor should be evaluated in the same lab.

Lab Task 1:
Implement lexical analyzer using two buffers