

Lab 5

Activity 1:

Implement symbol table using array data structure.

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;36
            //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] +
"");;37
    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 = 38
        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();
}39
// 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
            variable, 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|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
```

Graded task 1:

```
using System;
```

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

```
class SymbolTable
```

```
{
```

```
    private Dictionary<int, string> table;
```

```
    public SymbolTable()
```

```
    {
```

```
        table = new Dictionary<int, string>();
```

```
    }
```

```
    private int HashFunction(string key)
```

```
    {
```

```
        int hash = 0;
```

```
        foreach (char c in key)
```

```
        {
```

```
            hash = (hash * 31 + c) % 100; // Simple hash function
```

```
        }
```

```
        return hash;
```

```
    }
```

```
    public void Insert(string identifier)
```

```
    {
```

```
        int hash = HashFunction(identifier);
```

```
        if (!table.ContainsKey(hash))
```

```
        {
```

```
            table[hash] = identifier;
```

```
            Console.WriteLine($"Inserted: {identifier} at {hash}");
```

```
        }
```

```
        else
```

```
        {
```

```
            Console.WriteLine($"Collision occurred for {identifier} at {hash}");
```

```
        }
```

```
    }
```

```
    public bool Lookup(string identifier)
```

```
    {
```

```
        int hash = HashFunction(identifier);
```

```
        return table.ContainsKey(hash) && table[hash] == identifier;
```

```
    }
```

```
    public void Display()
```

```
    {
```

```
        Console.WriteLine("Symbol Table:");
```

```
        foreach (var entry in table)
```

```
        {
```

```
            Console.WriteLine($"{entry.Key}: {entry.Value}");
```

```
        }
```

```
    }  
}  
  
class Program  
{  
    static void Main()  
    {  
        SymbolTable symTable = new SymbolTable();  
        symTable.Insert("x");  
        symTable.Insert("y");  
        symTable.Insert("z");  
  
        Console.WriteLine("Lookup x: " + symTable.Lookup("x"));  
        Console.WriteLine("Lookup a: " + symTable.Lookup("a"));  
  
        symTable.Display();  
    }  
}
```

Output

```
Inserted: x at 20  
Inserted: y at 21  
Inserted: z at 22  
Lookup x: True  
Lookup a: False  
Symbol Table:  
20: x  
21: y  
22: z
```