COMSATS University, Islamabad

***Title**: Lab Midterm*

***Course**: Compiler Construction*

Name: Aaiza Irfan (SP20-BCS-001)

Aliza Tanweer (SP20-BCS-013)

Kulsoom Khurshid (SP20-BCS-044)

Submitted to: Sir Salman Aslam

Class: BCS-7A

```
using Parser.Lexical;
using Parser.Models;
using Parser.Parse;
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.IO;
using System.Linq;
using System.Threading.Tasks;
using System.Windows.Forms;
using Microsoft.Msagl.Core.Geometry.Curves;
using Microsoft.Msagl.Drawing;
using Microsoft.Msagl.GraphViewerGdi;
using Parser.LLTable;
using Action = Parser.State.Action;
using Color = System.Drawing.Color;
using TreeNode = Parser.Parse.TreeNode;

namespace Parser
{
    public partial class FrmMain : Form
    {
        private readonly Stopwatch _stopwatch = new Stopwatch();
        private GrammarRules _grammarRules;
        private Preprocessor _preprocessor;
        private LeftToRight_RightMost_Zero _lrZero;
        private LeftToRight_LookAhead_One _leftToRightLookAhead1;

        public FrmMain()
        {
            InitializeComponent();
        }

        private void btnChooseFile_Click(object sender, EventArgs e)
        {
            ChooseFile(txtgrammarFile);
            btnParseGrammar_Click(null, null);
        }

        private void ChooseFile(TextBox textbox)
        {
            OpenFileDialog openFile = new OpenFileDialog()
            {
                CheckFileExists = true,
                AddExtension = true,
                Multiselect = false,
                CheckPathExists = true,
                DefaultExt = "txt",
                InitialDirectory = AppDomain.CurrentDomain.BaseDirectory,
            };

            openFile.ShowDialog();
            textbox.Text = openFile.FileName;
        }

        private void btnParseGrammar_Click(object sender, EventArgs e)
        {
            listBoxGrammar.Items.Clear();
            listBoxFirst.Items.Clear();
            if (string.IsNullOrEmpty(txtgrammarFile.Text))
            {
                MessageBox.Show("Grammar file is empty!");
                return;
            }
            var text = File.ReadAllText(txtgrammarFile.Text);
            LexicalAnalyzer lex = new LexicalAnalyzer(text);
            RestartStopWatch();
            _grammarRules = lex.TokenizeGrammar();
            _stopwatch.Stop();
            lblTime.Text = $"Tokenizing process took
{_stopwatch.ElapsedMilliseconds} ms.";
            foreach (ISymbol symbol in _grammarRules.SymbolList)
            {
                if (symbol.SymbolType == SymbolType.Variable)
                    listBoxGrammar.Items.Add(((Variable)symbol).ShowRules());
            }
        }

        private void FrmMain_Load(object sender, EventArgs e)
        {
            cmbGrammarType.SelectedIndexChanged -=
cmbGrammarType_SelectedIndexChanged;
            cmbGrammarType.SelectedIndex = 0;
            cmbGrammarType.SelectedIndexChanged +=
cmbGrammarType_SelectedIndexChanged;


        }

        private void TabPreprocess_Enter(object sender, EventArgs e)
        {
            listBoxFirst.Items.Clear();
            listBoxFollow.Items.Clear();
            if (_grammarRules == null)
            {
                MessageBox.Show("Grammar File is empty");
                return;
            }
            _preprocessor = new Preprocessor(_grammarRules);
            RestartStopWatch();
            _preprocessor.CalculateAllFirsts();
            _preprocessor.CalculateAllFollows();
            _stopwatch.Stop();
            lblTime.Text = $"First and follow calculation took
{_stopwatch.ElapsedMilliseconds} ms.";
            foreach (ISymbol symbol in _grammarRules.SymbolList)
            {
                if (symbol is Variable variable)
                {
                    listBoxFirst.Items.Add(variable.ShowFirsts());
                    listBoxFollow.Items.Add(variable.ShowFollows());
                }
            }
        }

        private async void ll_1_Tab_Enter(object sender, EventArgs e)
        {
            dataGridViewReport.Rows.Clear();
            Progress<ParseReportModel> progress = new Progress<ParseReportModel>();
            progress.ProgressChanged += Progress_ProgressChanged;
            if (_grammarRules == null)
            {
                MessageBox.Show("Grammer file is empty");
                return;
            }

            _leftToRightLookAhead1 = new LeftToRight_LookAhead_One(_grammarRules,
progress);

            _leftToRightLookAhead1.Init();
            RestartStopWatch();

            var data = await Task.Run(() => _leftToRightLookAhead1.ProcessTable());
            _stopwatch.Stop();
            var creatingTableTime = _stopwatch.ElapsedMilliseconds;

            dataGridViewLL_1.Columns.Clear();
            dataGridViewLL_1.Rows.Clear();
            foreach (KeyValuePair<string, int> keyValuePair in
_leftToRightLookAhead1.MapperToNumber.MapTerminalToNumber)
            {
                dataGridViewLL_1.Columns.Add(keyValuePair.Key, keyValuePair.Key);
            }

            foreach (var keyValue in
_leftToRightLookAhead1.MapperToNumber.MapVariableToNumber)
            {
                dataGridViewLL_1.Rows.Add(new DataGridViewRow()
                {
                    HeaderCell = { Value = keyValue.Key },
                });
            }

            bool isValid = true;
            for (var i = 0; i < _leftToRightLookAhead1.MapperToNumber.VariableCount;
i++)
            {
                for (var j = 0; j <
_leftToRightLookAhead1.MapperToNumber.TerminalCount; j++)
                {
                    if (data[i, j] == null)
                    {
                        continue;
                    }
                    dataGridViewLL_1.Rows[i].Cells[j].Value = string.Join("",
data[i, j]);
                    if (data[i, j].Contains(Terminal.Error))
                    {
                        dataGridViewLL_1.Rows[i].Cells[j].Style.BackColor =
Color.Orange;
                        isValid = false;
                    }
                    else
                    {
                        dataGridViewLL_1.Rows[i].Cells[j].Style.BackColor = 
Color.LightGreen;
                    }
                }
            }

            long calculatingString = 0;
            if (isValid)
            {
                var terminals = GetTerminals();
                if (terminals == null)
                {
                    MessageBox.Show("test file is empty!");
                    return;
                }
                RestartStopWatch();
                _leftToRightLookAhead1.Parse(terminals);
                _stopwatch.Stop();
                calculatingString = _stopwatch.ElapsedMilliseconds;
            }
            lblTime.Text = $"Creating LookAhead Table took {creatingTableTime} ms. 
Stack Calculation took {calculatingString} ms.";
        }

        private List<Terminal> GetTerminals()
        {
            if (string.IsNullOrWhiteSpace(txtTestFile.Text)) return null;
            return new LexicalAnalyzer(
                File.ReadAllText(txtTestFile.Text)).TokenizeInputText();
        }


        private void Progress_ProgressChanged(object sender, ParseReportModel e)
        {
            dataGridViewReport.Rows.Add(e.Stack, e.InputString, e.Output);
        }

        private void RestartStopWatch()
        {
            _stopwatch.Stop();
            _stopwatch.Reset();
            _stopwatch.Start();
        }

        private void btnChooseTestFile_Click(object sender, EventArgs e)
        {
            ChooseFile(txtTestFile);
        }

        private void tabItem_Enter(object sender, EventArgs e)
        {

        }

        private void tabLR_0_Enter(object sender, EventArgs e)
        {
            dataGridReportLR.Rows.Clear();
            dgvLR_0.Rows.Clear();
            dgvLR_0.Columns.Clear();

            if (_preprocessor == null)
            {
                TabPreprocess_Enter(null,null);
                if (_preprocessor == null)
                {
                    return;
                }
            }

            LRType lrType = (LRType) cmbGrammarType.SelectedIndex;
            _lrZero = new
LeftToRight_RightMost_Zero(_grammarRules,lrType,_preprocessor);
            RestartStopWatch();
            txtLRStates.Text=_lrZero.CalculateStateMachine();
            var grammarTable = _lrZero.FillTable();
            _stopwatch.Stop();
            var tableAndStateMachine = _stopwatch.ElapsedMilliseconds;
            foreach(var keyValuePair in _lrZero.MapperToNumber.MapTerminalToNumber)
            {
                dgvLR_0.Columns.Add(keyValuePair.Key, keyValuePair.Key);
            }
            foreach(var keyValuePair in
_lrZero.MapperToNumber.MapVariableToNumber.Skip(1))
            {
                dgvLR_0.Columns.Add(keyValuePair.Key, keyValuePair.Key);
            }
            foreach (var keyValue in _lrZero.FiniteStateMachine.States)
            {
                dgvLR_0.Rows.Add(new DataGridViewRow()
                {
                    HeaderCell = { Value = keyValue.StateId },
                });
            }

            bool valid = true;
            foreach (var state in _lrZero.FiniteStateMachine.States)
            {
                for (int j = 0; j < _lrZero.MapperToNumber.TerminalCount; j++)
                {
                    var parserAction = grammarTable.ActionTable[state.StateId, j];
                    if(parserAction==null) continue;
                    dgvLR_0.Rows[state.StateId].Cells[j].Value = parserAction;
                    dgvLR_0.Rows[state.StateId].Cells[j].Style.BackColor =
!parserAction.HasError? Color.LightGreen: Color.Orange;
                    if (parserAction.HasError) valid = false;
                }

                int terminalCount = _lrZero.MapperToNumber.TerminalCount;
                for (int j = 0; j < _lrZero.MapperToNumber.VariableCount; j++)
                {
                    if(grammarTable.GoToTable[state.StateId, j]==null) continue;
                    dgvLR_0.Rows[state.StateId].Cells[j+terminalCount-1].Value =
grammarTable.GoToTable[state.StateId, j].StateId;
                    dgvLR_0.Rows[state.StateId].Cells[j+terminalCount-
1].Style.BackColor = Color.LightGreen;
                }
            }
            Progress<ParseReportModel> progress = new Progress<ParseReportModel>();
            progress.ProgressChanged += (o, m) =>
            {
                dataGridReportLR.Rows.Add(m.Stack, m.InputString, m.Output);
            };

            long stackTime = 0;
            if(valid)
            {
                var terminals = GetTerminals();
                if (terminals == null)
                {
                    MessageBox.Show("Terminal is empty!");
                    return;
                }
                RestartStopWatch();
                _lrZero.Parse(terminals,progress);
                _stopwatch.Stop();
                stackTime = _stopwatch.ElapsedMilliseconds;
            }
            lblTime.Text = $"Creating LR Table took {tableAndStateMachine} ms. Stack
Calculation took {stackTime} ms.";
        }

        private void tabLR_0_Click(object sender, EventArgs e)
        {

        }

        private void cmbGrammarType_SelectedIndexChanged(object sender, EventArgs e)
        {
            tabLR_0_Enter(null,null);
        }

        private void tabPage1_Leave(object sender, EventArgs e)
        {

        }

        private void tabItem_Selecting(object sender, TabControlCancelEventArgs e)
        {
//            if (e.TabPageIndex == 0)
//            {
//                if (string.IsNullOrWhiteSpace(txtgrammarFile.Text.Trim()))
//                {
//                    e.Cancel = true;
//                }
//            }
        }

        private void btnFSM_Click(object sender, EventArgs e)
        {
            System.Windows.Forms.Form form = new System.Windows.Forms.Form();
            form.WindowState = FormWindowState.Maximized;
            //create a viewer object
            Microsoft.Msagl.GraphViewerGdi.GViewer viewer = new
Microsoft.Msagl.GraphViewerGdi.GViewer();
```

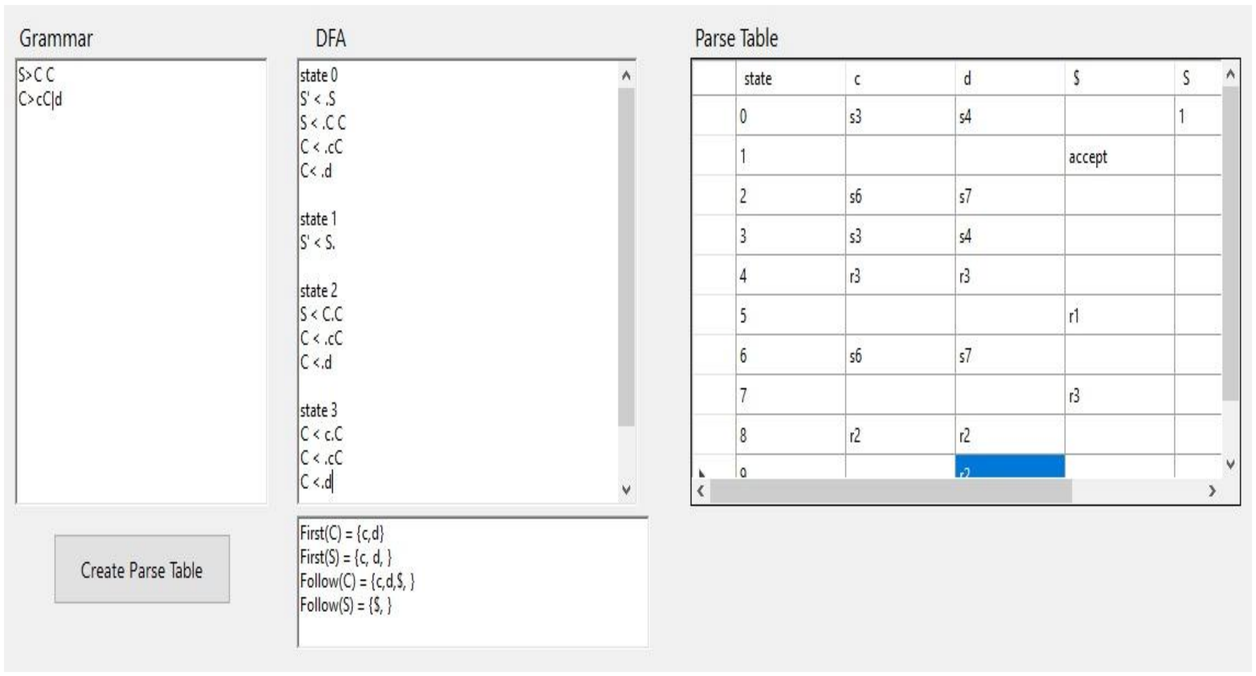
```
            //create a graph object
            var graph = new Graph("Finite State Machine");
            //create the graph content

            Dictionary<States.State,Node> dictionary = new Dictionary<States.State,
Node>();
            foreach (States.State state in _lrZero.FiniteStateMachine.States)
            {
                Node node = new Node(state.ToStringCompact());
                node.Attr.FillColor = state.ReduceOnly ?
Microsoft.Msagl.Drawing.Color.SeaGreen :
                                        (state.ShiftOnly ?
Microsoft.Msagl.Drawing.Color.LightGreen : Microsoft.Msagl.Drawing.Color.Orange);

                dictionary.Add(state,node);
                graph.AddNode(node);
            }

            foreach (States.State state in _lrZero.FiniteStateMachine.States)
            {
                foreach (KeyValuePair<ISymbol, States.State> stateNextState in
state.NextStates)
                {
                    var edge = new
Edge(dictionary[state],dictionary[stateNextState.Value],ConnectionToGraph.Connected)
;
                    edge.LabelText = stateNextState.Key.ToString();
                    graph.AddPrecalculatedEdge(edge);
                }
            }

            viewer.Graph = graph;
            //associate the viewer with the form
            form.SuspendLayout();
            viewer.Dock = System.Windows.Forms.DockStyle.Fill;
            form.Controls.Add(viewer);
            form.ResumeLayout();
            //show the form
            form.ShowDialog();
        }

        private void btnShowParseTree_Click(object sender, EventArgs e)
        {

            Queue<TreeNode> nodes = new Queue<TreeNode>();
            foreach (TreeNode lrZeroNode in _lrZero.Nodes)
            {
                nodes.Enqueue(lrZeroNode);
            }


        }

        private void btnLLParseTree_Click(object sender, EventArgs e)
        {
            Queue<TreeNode> nodes = new Queue<TreeNode>();
            foreach (TreeNode lrZeroNode in _leftToRightLookAhead1.BaseNode.Nodes)
```

```
                {
                    nodes.Enqueue(lrZeroNode);
                }

            }

            private void txtLRStates_TextChanged(object sender, EventArgs e)
            {

            }

            private void dgvLR_0_CellContentClick(object sender,
    DataGridViewCellEventArgs e)
            {

            }

            private void dataGridReportLR_CellContentClick(object sender,
    DataGridViewCellEventArgs e)
            {

            }
        }
    }

    using System;
    using System.CodeDom;
    using System.Collections.Generic;
    using System.Linq;
    using System.Text;
    using System.Threading.Tasks;
    using Parser.Models;
    using Parser.State;

    namespace Parser.Lexical
    {
        public class GrammarRules
        {
            /// <summary>
            /// Accessing all symbols with string
            /// </summary>
            private Dictionary<string,ISymbol> Symbols { get; set; }

            public IEnumerable<ISymbol> SymbolList => Symbols.Values;

            public Variable HeadVariable { get; set; }



            public GrammarRules()
            {
                Symbols = new Dictionary<string, ISymbol>();
            }
            /// <summary>
            /// access a symbol with string
            /// </summary>
            /// <param name="value"></param>
            /// <param name="symbolType"></param>
            /// <returns></returns>
            public ISymbol GetOrCreateSymbol(string value,SymbolType symbolType)
            {
                if (symbolType == SymbolType.Terminal)
                {
                    if (value == Terminal.EndOfFile.Value)
                        return Terminal.EndOfFile;
                    if (value == "")
                        return Terminal.Epsilon;
                    if (!Symbols.ContainsKey(value))
                        Symbols.Add(value, new Terminal(value));

                    return Symbols[value];
                }
                // else
                if (!Symbols.ContainsKey(value))
                    Symbols.Add(value,new Variable(value));
                return Symbols[value];

            }
        }
    }
    using System.Collections.Generic;
    using System.Linq;
    using Parser.Lexical;
    using Parser.Models;

    namespace Parser.Parse
    {
        /// <summary>
        /// Maps Variable and states to number
        /// for example S -> 2
        ///
        /// </summary>
        public class MapperToNumber
        {
            private readonly GrammarRules _grammarRules;

            /// <summary>
            /// variable mapping
            /// </summary>
            public Dictionary<string, int> MapVariableToNumber { get; }

            /// <summary>
            /// terminal mapping
            /// </summary>
            public Dictionary<string, int> MapTerminalToNumber { get; }

            /// <summary>
            /// count of Variables
            /// </summary>
            public int VariableCount { get; private set; }

            /// <summary>
            /// count of terminals
            /// </summary>
            public int TerminalCount { get; private set; }
            public MapperToNumber(GrammarRules grammarRules)
            {
                _grammarRules = grammarRules;
                MapTerminalToNumber = new Dictionary<string, int>();
                MapVariableToNumber = new Dictionary<string, int>();
            }

            /// <summary>
            /// calculating terminal and variables count
            /// note that I put $ in the terminals
            /// </summary>
            public void Initialize()
            {
                VariableCount = 0;
                TerminalCount = 0;
                foreach (var symbol in _grammarRules.SymbolList
                    .Where(symbol => !symbol.Equals(Terminal.Epsilon) &&
                                     !symbol.Equals(Terminal.EndOfFile)))
                {
                    if (symbol.SymbolType == SymbolType.Variable)
                    {
                        MapVariableToNumber.Add(symbol.Value, VariableCount);
                        VariableCount++;
                    }
                    else
                    {
                        MapTerminalToNumber.Add(symbol.Value, TerminalCount);
                        TerminalCount++;
                    }
                }
                MapTerminalToNumber.Add(Terminal.EndOfFile.Value, TerminalCount++);
            }

            public int Map(Terminal terminal)
            {
                return MapTerminalToNumber[terminal.Value];
            }

            public int Map(Variable variable)
            {
                return MapVariableToNumber[variable.Value];
            }
        }
    }

    using System;
    using System.Collections.Generic;
    using System.Linq;
    using System.Text;
    using System.Text.RegularExpressions;
    using System.Threading.Tasks;
    using Parser.Lexical;
    using Parser.Models;

    namespace Parser
    {

        public class LexicalAnalyzer
        {
            // Variable -> ProducedRule
            // Produced Rule is a list of variable or terminals
            private const string Head = "Head";
            private readonly GrammarRules _grammarRules;

            private string Data { get; set; }

            public LexicalAnalyzer(string data)
            {
                Data = data;
                _grammarRules = new GrammarRules();
            }

            public GrammarRules TokenizeGrammar()
            {
                //My Wife insisted that first rule should be the head!
                _grammarRules.GetOrCreateSymbol(Head, SymbolType.Variable);

                var lines = Data.Split('\n');
                foreach (var line in lines)
                {
                    if(!string.IsNullOrWhiteSpace(line))
                        LineTokenExtractor(line);
                }

                return _grammarRules;
            }

            public List<Terminal> TokenizeInputText()
            {
                var lines = Data.Split('\n');

                return AddEndSymbol((from line in lines
                    .Where(s => !string.IsNullOrEmpty(s))
                    from item in line.Split(' ')
                    select new Terminal(item)).ToList());
            }

            public List<Terminal> AddEndSymbol(List<Terminal> terminals)
            {
                terminals.Add(Terminal.EndOfFile);
                return terminals;
            }

            private void LineTokenExtractor(string line)
            {
                Regex text = new Regex(@"<(?<variable>[\w-
    ]+)>|""(?<terminal>[^""<>]+)?""",RegexOptions.Compiled);

                var matches=text.Matches(line);
                var firstVariable = matches[0].Groups["variable"];

                if (!firstVariable.Success) return;
                var headVariable=_grammarRules.GetOrCreateSymbol(firstVariable.Value,
    SymbolType.Variable);

                var symbols = new List<ISymbol>();
                for (var index = 1; index < matches.Count; index++)
                {
                    Match match = matches[index];
                    var variable = match.Groups["variable"];
                    if (variable.Success)
                    {

    symbols.Add(_grammarRules.GetOrCreateSymbol(variable.Value,SymbolType.Variable));
                        continue;
                    }

                    var terminal = match.Groups["terminal"];
                    if (terminal.Success)
                    {

    symbols.Add(_grammarRules.GetOrCreateSymbol(terminal.Value,SymbolType.Terminal));
                        continue;
                    }

                    //if it comes here then it's epsilon

    symbols.Add(_grammarRules.GetOrCreateSymbol("",SymbolType.Terminal));
                }


                if (_grammarRules.HeadVariable == null)
                {
                    Variable addedVariable=(Variable)
    _grammarRules.GetOrCreateSymbol(Head, SymbolType.Variable);
                    addedVariable.RuleSet.Definitions.Add(new
    List<ISymbol>(){headVariable});
                    _grammarRules.HeadVariable = addedVariable;
                    //_grammarRules.HeadVariable = (Variable)headVariable;
                }
                ((Variable)headVariable).RuleSet.Definitions.Add(symbols);
            }

        }
    }

    using Parser.Lexical;
    using Parser.Models;
    using System.Collections.Generic;
    using System.Linq;
    using Parser.Parse;

    namespace Parser.States
    {
        public class FiniteStateMachine
        {
            private readonly GrammarRules _grammarRules;
            private readonly bool _isClr;
            public HashSet<State> States { get; }
            private Preprocessor _preprocessor;
            public FiniteStateMachine(GrammarRules grammarRules,Preprocessor
    preprocessor, bool isClr)
            {
                _grammarRules = grammarRules;
                _isClr = isClr;
                States = new HashSet<State>();
                _preprocessor = preprocessor;
            }
            public void InitializeAllStates()
            {
                Queue<State> queue = new Queue<State>();

                State firstState = new State(_preprocessor,_isClr);
                foreach (var rule in _grammarRules.HeadVariable.RuleSet.Definitions)
                {
                    var rowState = new RowState(_grammarRules.HeadVariable, rule);
                    if(_isClr) rowState.LookAhead = new
    List<Terminal>(){Terminal.EndOfFile};
                    firstState.AddRowState(rowState);
                }
                firstState.AddClosures();
                queue.Enqueue(firstState);
                States.Add(firstState);
                int stateNo = 1;
                while (queue.Count > 0)
                {
                    var state = queue.Dequeue();

                    //producing new items
                    var extractFirstSymbol = state.ExtractFirstSymbol().Distinct();
                    foreach (ISymbol symbol in extractFirstSymbol)
                    {
                        var nextState = state.CreateNextState(symbol);
                        nextState.AddClosures();
                        if (nextState.RowStates.Count > 0)
                        {
                            nextState.PreviousState = state;
                            nextState.TransferredSymbol = symbol;

                            //It's right not to add the state
                            if (!States.Contains(nextState))
                            {
                                queue.Enqueue(nextState);
                                nextState.StateId = stateNo;
                                stateNo++;
                                States.Add(nextState);

                                //if it's not the first State
                                if (!state.NextStates.ContainsKey(symbol))
                                {
                                    state.NextStates.Add(symbol, nextState);
                                }

                            }
                            else
                            {//but we should add next state to know where to go
                                state.NextStates.Add(symbol, States.First(f =>
    f.Equals(nextState)));
                            }
                        }
                    }
                }
            }

            public override string ToString()
            {
                return string.Join("\n-------------\n", States);
            }
        }

    }
```

LR Table code

```
using Parser.Models;
using Parser.Parse;
using Parser.States;
using System.Collections.Generic;
using System.Linq;
using Parser.LLTable;

namespace Parser.State
{
    /// <summary>
    /// Containing All LR Table Information
    /// GoTo
    /// Action
    /// </summary>
    public class LRGrammarTable
    {
        private readonly FiniteStateMachine _fsm;
        private readonly MapperToNumber _mapperToNumber;
        private readonly LRType _lrType;
        public ParserAction[,] ActionTable { get; set; }
        public GoTo[,] GoToTable { get; set; }

        public LRGrammarTable(FiniteStateMachine fsm, MapperToNumber mapperToNumber,
LRType lrType)
        {
            _fsm = fsm;
            _mapperToNumber = mapperToNumber;
            _lrType = lrType;
        }

        public void Init()
        {
            ActionTable = new ParserAction[_fsm.States.Count,
_mapperToNumber.TerminalCount];
            GoToTable = new GoTo[_fsm.States.Count, _mapperToNumber.VariableCount];
        }

        public ParserAction GetParserAction(int state, Terminal terminal)
        {
            return ActionTable[state, _mapperToNumber.Map(terminal)];
        }

        public GoTo GetGoTo(int state,Variable variable)
        {
            return GoToTable[state, _mapperToNumber.Map(variable)];
        }

        public void AddParseActionToTable(int row,int cell,ParserAction
parserAction)
        {
            if (ActionTable[row, cell] == null)
                ActionTable[row, cell] = parserAction;
            else
            {
                if(!ActionTable[row,cell].Equals(parserAction))
                    ActionTable[row, cell].ErrorAction = parserAction;
            }
        }
        public void FillTable(Variable head)
        {

            foreach (States.State currentState in _fsm.States)
            {
                AddState(head, currentState);
            }
        }

        private void AddState(Variable head, States.State currentState)
        {
            AddReduceAccept(head, currentState);
            AddShiftGo(currentState);
        }

        private void AddShiftGo(States.State currentState)
        {
            foreach (KeyValuePair<ISymbol, States.State> fsmStateNextState in
currentState.NextStates)
            {
                //shift
                if (fsmStateNextState.Key is Terminal terminal)
                {
                    ParserAction action = new ParserAction
                    {
                        ShiftState = fsmStateNextState.Value.StateId,
                        Action = Action.Shift
                    };
                    AddParseActionToTable(currentState.StateId,
_mapperToNumber.Map(terminal), action);
                }
                //goto
                else if (fsmStateNextState.Key is Variable variable)
                {
                    GoToTable[currentState.StateId, _mapperToNumber.Map(variable)] =
                        new GoTo(fsmStateNextState.Value.StateId);
                }
            }
        }

        private void AddReduceAccept(Variable head, States.State currentState)
        {
            foreach (RowState currentStateRowState in currentState.RowStates)
            {
                ParserAction parser = new ParserAction();
                //if some rule is finished it means reduce or accept
                if (!currentStateRowState.Finished) continue;

                if (currentStateRowState.Variable.Equals(head))
                {
                    parser.Action = Action.Accept;
                    AddParseActionToTable(currentState.StateId,
_mapperToNumber.Map(Terminal.EndOfFile), parser);
                }
                else
                {
                    parser.Action = Action.Reduce;
                    parser.Variable = currentStateRowState.Variable;
                    parser.Handle = currentStateRowState.Rule;

                    if (_lrType == LRType.Zero)
                    {
                        for (int i = 0; i < _mapperToNumber.TerminalCount; i++)
                        {
                            AddParseActionToTable(currentState.StateId, i, parser);
                        }
                    }
                    else if (_lrType == LRType.SLR_One)
                    {
                        foreach (Terminal terminal in
currentStateRowState.Variable.Follows)
                        {
                            AddParseActionToTable(currentState.StateId,
_mapperToNumber.Map(terminal), parser);
                        }
                    }
                    else if (_lrType == LRType.ClR_One)
                    {
                        foreach (Terminal terminal in
currentStateRowState.LookAhead)
                        {

AddParseActionToTable(currentState.StateId,_mapperToNumber.Map(terminal),parser);
                        }
                    }
                }
            }
        }
    }
}
```

Output:



Grammar

```
S>C C
C>cC|d
```

DFA

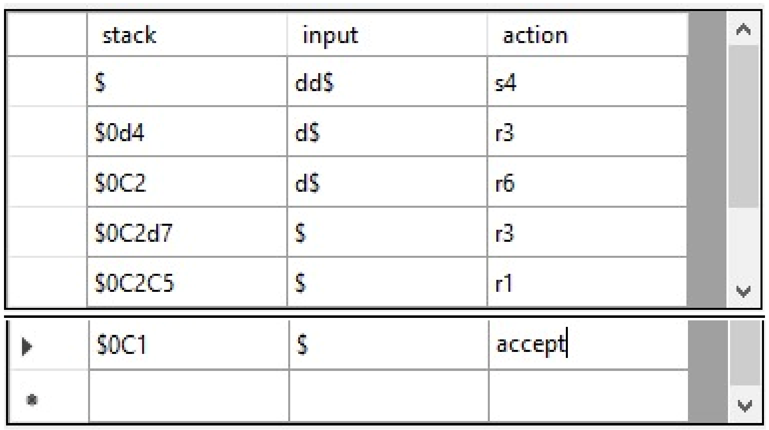
```
state 0
S' < .S
S < .C C
C < .cC
C< .d

state 1
S' < S.

state 2
S < C.C
C < .cC
C <.d

state 3
C < c.C
C < .cC
C <.d
```

```
First(C) = {c,d}
First(S) = {c, d, }
Follow(C) = {c,d,$, }
Follow(S) = {$, }
```

Parse Table

	state	c	d	$	S
	0	s3	s4		1
	1			accept	
	2	s6	s7		
	3	s3	s4		
	4	r3	r3		
	5			r1	
	6	s6	s7		
	7			r3	
	8	r2	r2		
	9		r2		

Input:



	stack	input	action
	$	dd$	s4
	$0d4	d$	r3
	$0C2	d$	r6
	$0C2d7	$	r3
	$0C2C5	$	r1
	$0C1	$	accept