Lab Terminal

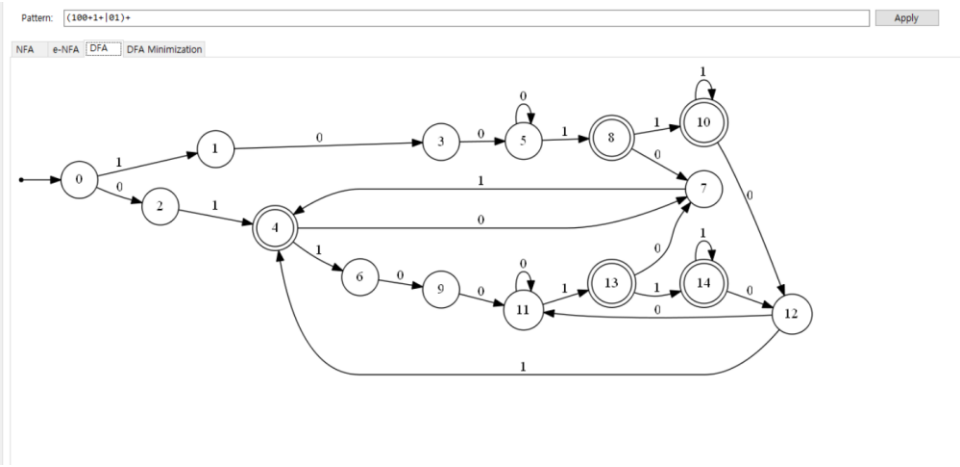
Fa20-bcs-009

Maryam Amjad

Question 2: Two functionalities along with screenshots (function code +output).

1. RE to DFA :

Output:



Code:

```
private diagram make_nfa(string pattern)
{
    var first_valid_stack = new Stack<transition_node>();
    var second_valid_stack = new Stack<transition_node>();
    var first_valid_stack_stack = new List<Stack<transition_node>>();
    var second_valid_stack_stack = new List<Stack<transition_node>>();
    var tail_nodes = new Stack<List<transition_node>>();
    var opstack = new Stack<char>();
    var diagram = new diagram();
```

```
   var index_count = 0;
   var cur = new transition_node();
   var nodes = new List<transition_node>();
   var depth = 0;

   cur.index = index_count++;
   cur.transition = new List<Tuple<char, transition_node>>();
   diagram.start_node = cur;
   first_valid_stack.Push(cur);
   nodes.Add(cur);

   for (int i = 0; i < pattern.Length; i++)
   {
      switch (pattern[i])
      {
         case '(':
            opstack.Push('(');
            depth++;
            // Copy stack and push to stack stack
            first_valid_stack_stack.Add(new Stack<transition_node>(new
Stack<transition_node>(first_valid_stack)));
            second_valid_stack_stack.Add(new Stack<transition_node>(new
Stack<transition_node>(second_valid_stack)));
            second_valid_stack.Push(first_valid_stack.Peek());
            first_valid_stack.Push(cur);
            tail_nodes.Push(new List<transition_node>());
            break;
```

```
        case ')':
            if (opstack.Count == 0 || opstack.Peek() != '(')
            {
                build_errors.Add($"[regex] {i} no opener!");
                return null;
            }

            tail_nodes.Peek().Add(cur);
            var ends_point = new transition_node { index = index_count++, transition = new
List<Tuple<char, transition_node>>() };
            cur = ends_point;
            nodes.Add(cur);

            // Connect tail nodes
            foreach (var tail_node in tail_nodes.Peek())
                tail_node.transition.Add(new Tuple<char, transition_node>(e_closure, cur));

            tail_nodes.Pop();

            // Pop from stack stack
            first_valid_stack = first_valid_stack_stack.Last();
            first_valid_stack_stack.RemoveAt(first_valid_stack_stack.Count - 1);
            second_valid_stack = second_valid_stack_stack.Last();
            second_valid_stack_stack.RemoveAt(second_valid_stack_stack.Count - 1);

            second_valid_stack.Push(first_valid_stack.Peek());
            first_valid_stack.Push(cur);
```

```
            depth--;
            break;

        case '|':
            tail_nodes.Peek().Add(cur);
            cur = first_valid_stack_stack[first_valid_stack_stack.Count - 1].Peek();
            break;

        case '?':
            second_valid_stack.Peek().transition.Add(new Tuple<char,
transition_node>(e_closure, cur));
            break;

        case '+':
            var ttc = copy_nodes(ref nodes, second_valid_stack.Peek().index, cur.index);
            cur.transition.Add(new Tuple<char, transition_node>(e_closure, ttc.Item1));
            ttc.Item2.transition.Add(new Tuple<char, transition_node>(e_closure, cur));
            index_count += ttc.Item3;
            break;

        case '*':
            second_valid_stack.Peek().transition.Add(new Tuple<char,
transition_node>(e_closure, cur));
            cur.transition.Add(new Tuple<char, transition_node>(e_closure,
second_valid_stack.Peek()));
            break;

        case '[':
```

```
var ch_list = new List<char>();
i++;
bool inverse = false;

if (i < pattern.Length && pattern[i] == '^')
{
   inverse = true;
   i++;
}

for (; i < pattern.Length && pattern[i] != ']'; i++)
{
   if (pattern[i] == '\\')
   {
      if (i + 1 < pattern.Length && @"+-?*|()[].=<>/\".Contains(pattern[i + 1]))
      {
         ch_list.Add(pattern[++i]);
      }
      else
      {
         switch (pattern[++i])
         {
            case 'n':
               ch_list.Add('\n');
               break;
            case 't':
               ch_list.Add('\t');
```

```
                        break;
                    case 'r':
                        ch_list.Add('\r');
                        break;
                    case 'x':
                        char ch2;
                        ch2 = (char)(pattern[i + 1] >= 'A' ? (pattern[i + 1] - 'A' + 10) : pattern[i + 1] -
'0');
                        ch2 <<= 4;
                        ch2 |= (char)(pattern[i + 2] >= 'A' ? (pattern[i + 2] - 'A' + 10) : pattern[i + 2] -
'0');
                        i += 2;
                        ch_list.Add(ch2);
                        break;
                    default:
                        build_errors.Add($"{pattern[i]} escape character not found!");
                        ch_list.Add(pattern[i]);
                        break;
                }
            }
        }
        else if (i + 2 < pattern.Length && pattern[i + 1] == '-')
        {
            for (int j = pattern[i]; j <= pattern[i + 2]; j++)
                ch_list.Add((char)j);

            i += 2;
        }
```

```
            else
            {
                ch_list.Add(pattern[i]);
            }
        }

        var ends_point2 = new transition_node { index = index_count++, transition = new
List<Tuple<char, transition_node>>() };
        if (inverse)
        {
            var set = new bool[byte_size];
            var nch_list = new List<char>();
            foreach (var ch2 in ch_list)
                set[ch2] = true;
            for (int j = 0; j < byte_size; j++)
            {
                if (!set[j])
                    nch_list.Add((char)j);
            }

            ch_list.Clear();
            ch_list = nch_list;
        }

        foreach (var ch2 in ch_list)
        {
            cur.transition.Add(new Tuple<char, transition_node>(ch2, ends_point2));
```

```
            }
cur = ends_point2;
            nodes.Add(cur);

            if (first_valid_stack.Count != 0)
            {
               second_valid_stack.Push(first_valid_stack.Peek());
            }

            first_valid_stack.Push(cur);
            break;

         case '.':
            var ends_point3 = new transition_node { index = index_count++, transition = new
List<Tuple<char, transition_node>>() };

            for (int i2 = 0; i2 < byte_size; i2++)
            {
               cur.transition.Add(new Tuple<char, transition_node>((char)i2, ends_point3));
            }

            cur = ends_point3;
            nodes.Add(cur);

            if (first_valid_stack.Count != 0)
            {
               second_valid_stack.Push(first_valid_stack.Peek());
```

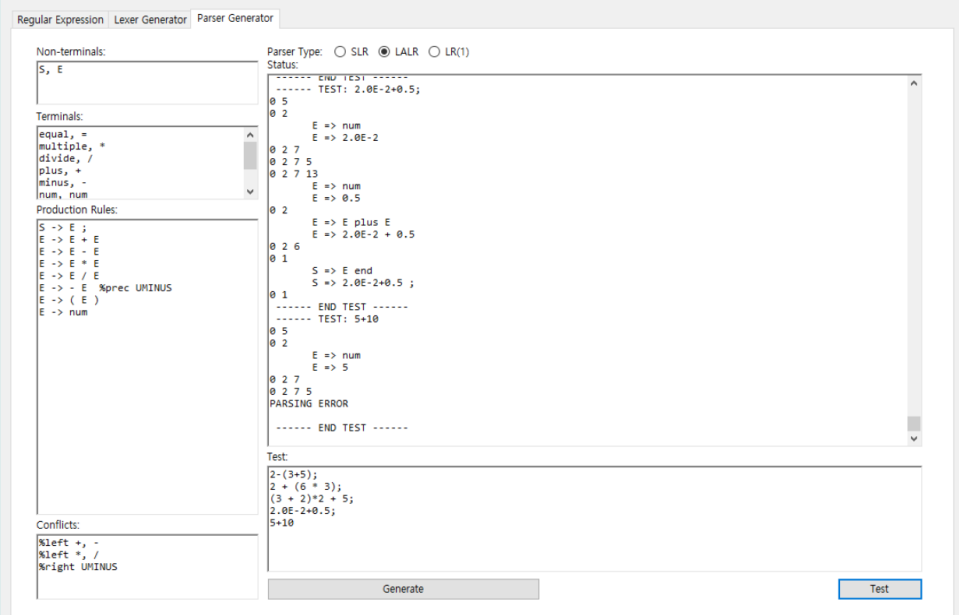
```
    }
    first_valid_stack.Push(cur);
    break;

case '\\':
default:
    char ch = pattern[i];

    if (pattern[i] == '\\')
    {
        i++;

        if (@"+-?*|()[].=<>/".Contains(pattern[i]))
        {
            ch = pattern[i];
        }
        else
        {
            switch (pattern[i])
            {
                case 'n':
                    ch = '\n';
                    break;
                case 't':
                    ch = '\t';
                    break;
                case 'r':
```

```
                    ch = '\r';
                    break;
                case 'x':
                    ch = (char)(pattern[i + 1] >= 'A' ? (pattern[i + 1] - 'A' + 10) : pattern[i + 1] - '0');
                    ch <<= 4;
                    ch |= (char)(pattern[i + 2] >= 'A' ? (pattern[i + 2] - 'A' + 10) : pattern[i + 2] -
'0');
                    i += 2;
                    break;
                default:
                    build_errors.Add($"{pattern[i]} escape character not found!");
                    ch = pattern[i];
                    break;
            }
        }
    }

    var etn = new transition_node { index = index_count++, transition = new
List<Tuple<char, transition_node>>() };
    cur.transition.Add(new Tuple<char, transition_node>(e_closure, etn));
    cur = etn;
    nodes.Add(cur);

    if (first_valid_stack.Count != 0)
    {
        second_valid_stack.Push(first_valid_stack.Peek());
    }
```

```
            first_valid_stack.Push(cur);

            var tn = new transition_node { index = index_count++, transition = new
List<Tuple<char, transition_node>>() };
            cur.transition.Add(new Tuple<char, transition_node>(ch, tn));
            cur = tn;
            nodes.Add(cur);

            if (first_valid_stack.Count != 0)
            {
               second_valid_stack.Push(first_valid_stack.Peek());
            }

            first_valid_stack.Push(cur);
            break;
        }
    }

    diagram.count_of_vertex = index_count;
    diagram.nodes = nodes;

    nodes.Where(x => x.transition.Count == 0).ToList().ForEach(y => y.is_acceptable = true);

    return diagram;
}
```

2. LALR Generator :

Output:



Code:

```
var gen2 = new ParserGenerator();


// Non-Terminals
var S = gen2.CreateNewProduction("S", false);
var L = gen2.CreateNewProduction("L", false);
var R = gen2.CreateNewProduction("R", false);


// Terminals
var equal = gen2.CreateNewProduction("=");
var mult = gen2.CreateNewProduction("*");
var div = gen2.CreateNewProduction("/");
var pp = gen2.CreateNewProduction("+");
```

```
var mi = gen2.CreateNewProduction("-");
var num = gen2.CreateNewProduction("num");

// right associativity, -
gen2.PushConflictSolver(false, new Tuple<ParserProduction, int>(S, 4));
// left associativity, *, /
gen2.PushConflictSolver(true, mult, div);
// left associativity, +, -
gen2.PushConflictSolver(true, pp, mi);

S |= S + pp + S;
S |= S + mi + S;
S |= S + mult + S;
S |= S + div + S;
S |= mi + S;
S |= num;

gen2.PushStarts(S);
gen2.Generate();
gen2.PrintStates();
gen2.GenerateLALR();
gen2.PrintStates();
var slr = gen2.CreateShiftReduceParserInstance();

// 2*4+5$

Action<string, string> insert = (string x, string y) =>
```

```
{
    slr.Insert(x, y);
    while (slr.Reduce())
    {
        Console.Instance.WriteLine(slr.Stack());
        var l = slr.LatestReduce();
        Console.Instance.Write(l.Produnction.PadLeft(8) + " => ");
        Console.Instance.WriteLine(string.Join(" ", l.Childs.Select(z => z.Produnction)));
        Console.Instance.Write(l.Produnction.PadLeft(8) + " => ");
        Console.Instance.WriteLine(string.Join(" ", l.Childs.Select(z => z.Contents)));
        slr.Insert(x, y);
    }
    Console.Instance.WriteLine(slr.Stack());
};
var sg2 = new ScannerGenerator();
sg2.PushRule("", "[\\r\\n ]");
sg2.PushRule("+", "\\+");
sg2.PushRule("-", "-");
sg2.PushRule("*", "\\*");
sg2.PushRule("/", "\\/");
sg2.PushRule("(", "\\(");
sg2.PushRule(")", "\\)");
sg2.PushRule(",", ",");
sg2.PushRule("id", "[a-z][a-z0-9]*");
sg2.PushRule("num", "[0-9]+");
sg2.Generate();
sg2.CreateScannerInstance();
```

```
var scanner2 = sg2.CreateScannerInstance();
//scanner2.AllocateTarget("2+6*(6+4*7-2)+sin(a,b,c,d)+cos()*pi");
scanner2.AllocateTarget("2+--6*4");
while (scanner2.Valid())
{var ss = scanner2.Next();
   insert(ss.Item1, ss.Item2);
}
insert("$", "$");
```