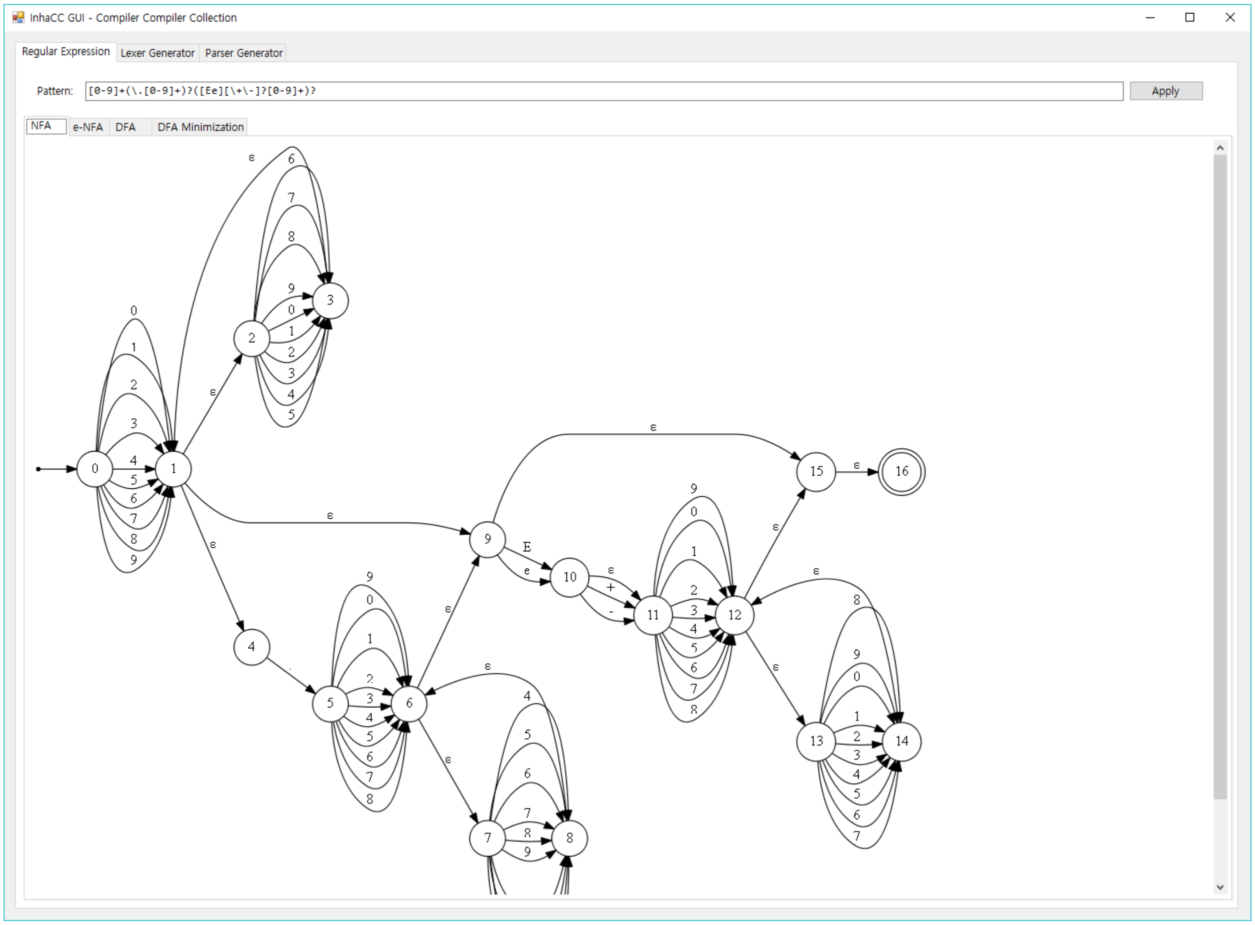
MUHAMMAD MUAAZ SHOAIB

FA20-BCS-074

Q2: TWO FUNCTIONALITIES ALONG WITH SCREENSHOTS

1. **Regular Expression (Regex)**



Pattern: `[0-9]+(\.[0-9]+)?([Ee][\+\-]?[0-9]+)?`

<u>Code:</u>

```
/// Try simple-regular-expression to NFA.
        /// </summary>
        /// <param name="pattern"></param>
        /// <returns></returns>
        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!");
```

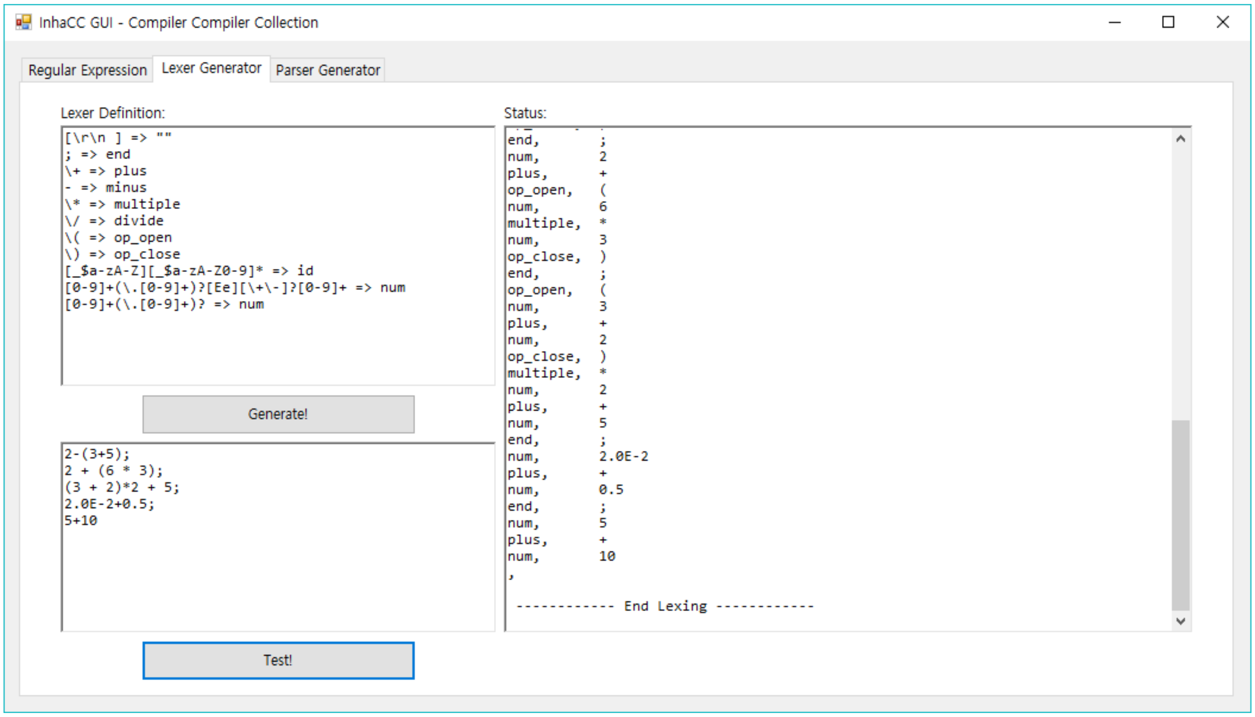
```
                            return null;
                        }
                        tail_nodes.Peek().Add(cur);
                        var ends_point = new transition_node { index = index_count++, transition =
new List<Tuple<char, transition_node>>() };
                        cur = ends_point;
                        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] == '\\' && i + 1 < pattern.Length)
              {
                if (@"+-?*|()[].=<>/\".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. Scanner Generator / Lexical Analyzer Generator



```
[\r\n ] => ""
; => end
\+ => plus
- => minus
\* => multiple
\/ => divide
\( => op_open
\) => op_close
[_$a-zA-Z][_$a-zA-Z0-9]* => id
[0-9]+(\.[0-9]+)?[Ee][\+\-]?[0-9]+ => num
[0-9]+(\.[0-9]+)? => num
```

```
2-(3+5);
2 + (6 * 3);
(3 + 2)*2 + 5;
2.0E-2+0.5;
5+10
```

```
end,       ;
num,       2
plus,      +
op_open,   (
num,       6
multiple,  *
num,       3
op_close,  )
end,       ;
op_open,   (
num,       3
plus,      +
num,       2
op_close,  )
multiple,  *
num,       2
plus,      +
num,       5
end,       ;
num,       2.0E-2
plus,      +
num,       0.5
end,       ;
num,       5
plus,      +
num,       10
,

 ------------ End Lexing ------------
```

CODE:

```
/// Lexical Analyzer Generator
  /// </summary>
  public class ScannerGenerator
  {
    bool freeze = false;
    List<Tuple<string, SimpleRegex.diagram>> tokens = new List<Tuple<string,
SimpleRegex.diagram>>();
    SimpleRegex.diagram diagram;

    public string PrintDiagram()
    {
      if (!freeze) throw new Exception("Retry after generate!");
      return SimpleRegex.PrintDiagram(diagram);
    }

    public void PushRule(string token_name, string rule)
    {
      if (freeze) throw new Exception("You cannot push rule after generate! Please
create new scanner-generator instance.");
      var sd = new SimpleRegex(rule);
      foreach (var node in sd.Diagram.nodes)
        if (node.is_acceptable)
          node.accept_token_name = token_name;
      tokens.Add(new Tuple<string, SimpleRegex.diagram>(token_name,
sd.Diagram));
    }

    /// <summary>
    /// Generate merged DFA using stack.
    /// </summary>
    public void Generate()
    {
      freeze = true;

      //                    * Warning *
      //
      // The merged_diagram index order is in the order of  DFA's
      // pattern mapping. Consider the PushRule function with this.

      var merged_diagram = get_merged_diagram();

      // Generated transition nodes for DFA based patttern matching.
      var diagram = new SimpleRegex.diagram();
      var nodes = new List<SimpleRegex.transition_node>();
      var states = new Dictionary<string, SimpleRegex.transition_node>();
      var index = new Dictionary<int, string>();
```

```
            var states_count = 0;

            // (diagram_indexes)
            var q = new Queue<List<int>>();
            q.Enqueue(populate(merged_diagram, new List<int> { 0 },
SimpleRegex.e_closure));

            var t = new SimpleRegex.transition_node { index = states_count++, transition =
new List<Tuple<char, SimpleRegex.transition_node>>() };
            states.Add(string.Join(",", q.Peek()), t);
            index.Add(t.index, string.Join(",", q.Peek()));
            nodes.Add(t);

            while (q.Count != 0)
            {
               var list = q.Dequeue();
               var list2str = string.Join(",", list);

               var tn = states[list2str];

               // Append accept tokens.
               foreach (var ix in list)
                  if (merged_diagram.nodes[ix].is_acceptable)
                  {
                     tn.is_acceptable = true;
                     if (tn.accept_token_names == null)
                        tn.accept_token_names = new List<string>();

tn.accept_token_names.Add(merged_diagram.nodes[ix].accept_token_name);
                  }

               var available = available_matches(merged_diagram, list);

               foreach (var pair in available)
               {
                  var populate = pair.Value.ToList();
                  var l2s = string.Join(",", populate);

                  if (!states.ContainsKey(l2s))
                  {
                     var tnt = new SimpleRegex.transition_node { index = states_count++,
transition = new List<Tuple<char, SimpleRegex.transition_node>>() };
                     states.Add(l2s, tnt);
```

```
                        index.Add(tnt.index, l2s);
                        nodes.Add(tnt);
                        q.Enqueue(populate);
                    }

                    var state = states[l2s];
                    tn.transition.Add(new Tuple<char, SimpleRegex.transition_node>(pair.Key,
state));
                }
            }

            diagram.nodes = nodes;
            diagram.start_node = nodes[0];
            diagram.count_of_vertex = nodes.Count;

            this.diagram = diagram;
        }
```