Lab Terminal Report

Fa20-bcs-009

Maryam Amjad

BCS 7A

Compiler Construction

Submitted To :

Sir .Bilal Haider Bukhari

28/12/2023

Q1: Brief Introduction of the project

The Compiler project seamlessly transforms regular expressions into Nondeterministic Finite Automata (NFA) and optimizes performance through deterministic conversions to Finite Automata (DFA), with a focus on DFA minimization. The Lexical Analyzer Generator facilitates the creation of customized analyzers, ensuring smooth integration into projects. Additionally, the project supports a variety of parsers, including SLR (Simple LR), LALR (Look-Ahead LR) from LR(1), LALR from LR(0), and LR(1). The integration of Graphviz enhances visualization, providing a practical aid for developers to understand Regular Expression transitions and structures visually.

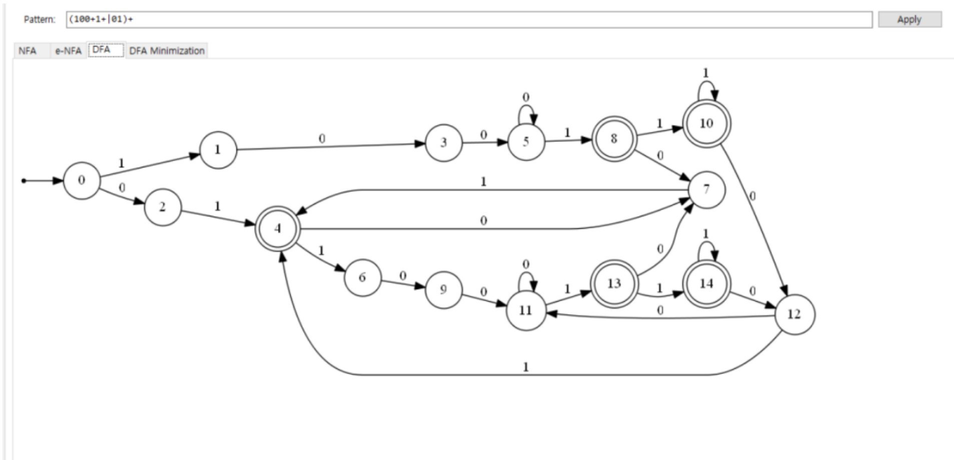
Key Features:

- **Regular Expression (RE) to Finite Automata Conversions:**
  - Conversion of regular expressions into Nondeterministic Finite Automata (NFA).
  - Subsequent transformation from NFA to Deterministic Finite Automata (DFA).
  - Minimization of DFA to optimize performance.
- **Lexical Analyzer Generator:**
  - Empower your compiler with a customized Lexical Analyzer using our user-friendly generator.
- **Parser Generators:**
  - Support for SLR (Simple LR), LALR (Look-Ahead LR) from LR(1), LALR from LR(0), and LR(1) parsers.
- **Graph Visualization with Graphviz:**
  - Visualize Regular Expression transitions and structures with Graphviz.
  - Ensure Graphviz is installed for optimal graph visualization:
  -

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];
```

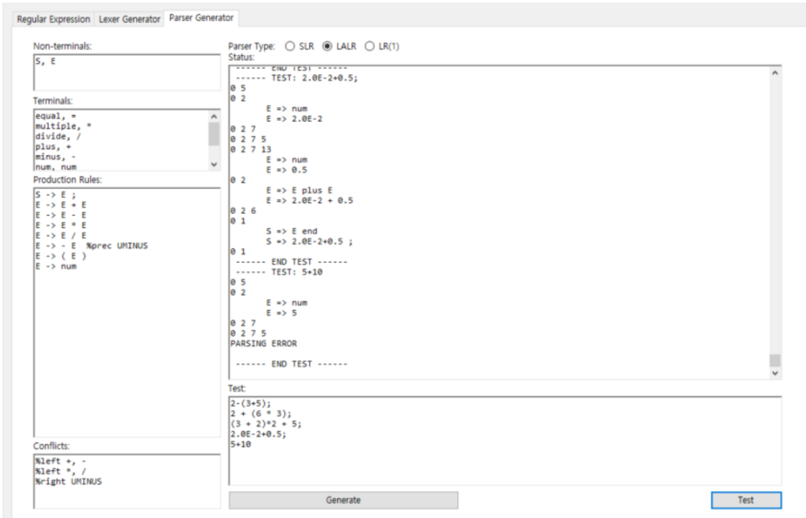
```
            if (pattern[i] == '\\')
            {
                i++;

                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, *, /
```

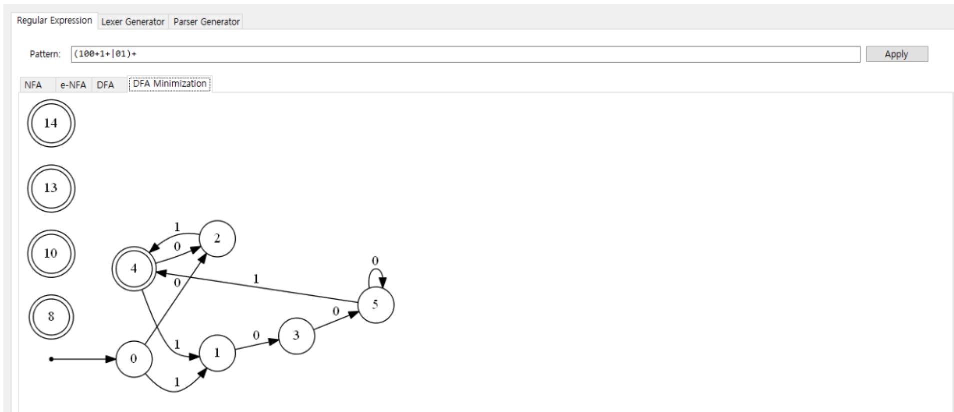
```
gen2.PushConflictSolver(true, mult, div);
// left associativity, +, -
gen2.PushConflictSolver(true, pp, mi);

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);
```

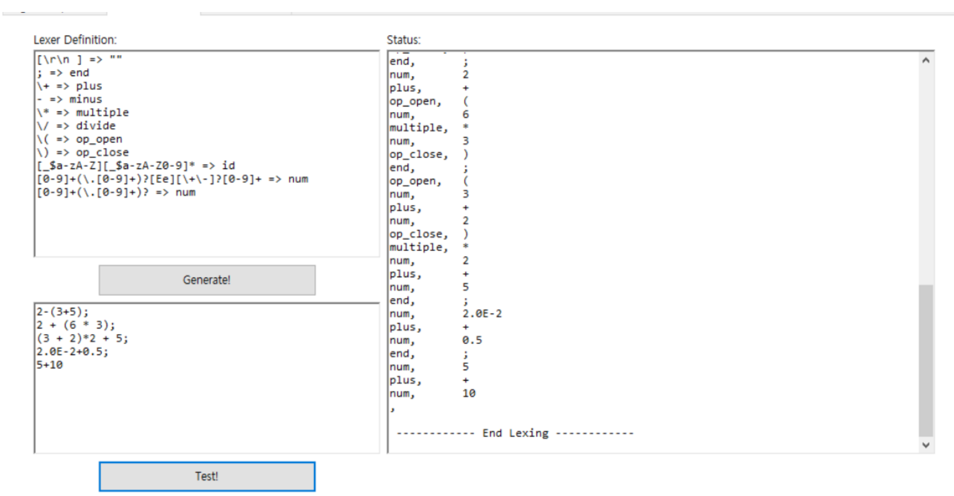
}insert("$", "$");

Q3:Two input along with output.

Re to DFA Minimization Input Output



Lexical Analyzer Generator:



Question 4: How functions work.

Regular Expression (RE) to DFA:

1. Lexical Analysis and Tokenization:

- - Use a lexer or tokenizer to convert the raw regular expression string into a sequence of tokens.
- - Tokens represent individual symbols, operators, and other elements.

2. Parse Tokens and Build NFA:

- - Parse the tokens generated in the previous step.
- - Build a Non-deterministic Finite Automaton (NFA) using algorithms like Thompson's Construction.

3. Convert NFA to DFA:

- - Implement the subset construction algorithm to convert the NFA to a Deterministic Finite Automaton (DFA).
- - DFA simplifies the representation and enhances efficiency.

4. Minimize DFA:

- - Apply a DFA minimization algorithm, such as Hopcroft's algorithm, to further optimize the DFA.
- - Identify and merge equivalent states to reduce the number of states while preserving language acceptance.

Putting it All Together:

- - Create a cohesive class or module that orchestrates the entire process.
- - Instantiate and use the necessary components: lexer, NFA builder, DFA converter, and DFA minimizer.
- - Pass the regular expression through each step to get the minimized DFA.

Lexical Generator:

1. Token Definition:

- - Define classes or structures to represent lexical tokens, including keywords, identifiers, operators, and literals.
- - Associate each token definition with a corresponding regular expression pattern.

2. Regular Expressions for Tokens:

- - Assign regular expressions to each token definition, specifying the patterns that correspond to each token type.

- - These regular expressions define the lexemes to be recognized during the lexical analysis.

3. RE to NFA:

- - Use the token-specific regular expressions to construct individual Non-deterministic Finite Automata (NFAs) for each token.
- - Employ algorithms, possibly Thompson's Construction, to build NFAs based on regular expressions.

4. Combine NFAs:

- - Merge the individual NFAs representing different tokens into a single NFA.
- - This combined NFA recognizes the entire lexical structure of the programming language.

5. Convert to DFA:

- - Utilize the subset construction algorithm to convert the combined NFA into a Deterministic Finite Automaton (DFA).
- - The DFA streamlines token recognition and improves efficiency.

6. Minimize DFA:

- - Apply a DFA minimization algorithm, such as Hopcroft's algorithm, to minimize the DFA.
- - Merge equivalent states to reduce the number of states while preserving language acceptance.

7. Lexical Analyzer Generation:

- - Generate code for a lexical analyzer based on the minimized DFA.
- - The lexical analyzer scans the input stream and recognizes tokens based on the minimized DFA.
- - Implement logic for token extraction and handling in the generated lexical analyzer.

Putting it All Together:

- - Develop a cohesive class or module that encapsulates the entire lexical generator process.
- - Instantiate and utilize the necessary components: token definitions, regular expressions, NFA builder, DFA converter, and DFA minimizer.
- - Sequentially pass through each step, from token definition to lexical analyzer generation, to achieve a fully functional lexical generator.

Parser Generator for SLR, LALR(from LR(1)), LALR(from LR(0)), LR(1):

1. Grammar Definition:

- Define the grammar for the language using appropriate notation (e.g., BNF).
- Clearly specify terminals, non-terminals, and production rules.

2. SLR Parser Generator:

- Implement an SLR (Simple LR) parser generator.
- Construct the LR(0) sets for the grammar.
- Build the SLR parsing table using LR(0) sets and follow sets.
- Generate parser code that utilizes the SLR parsing table for shift-reduce parsing.

3. LR(1) Parser Generator:

- Implement an LR(1) parser generator.
- Construct the LR(1) sets for the grammar.
- Build the LR(1) parsing table using LR(1) sets and lookaheads.
- Generate parser code that utilizes the LR(1) parsing table for shift-reduce parsing.

4. LALR(from LR(1)) Parser Generator:

- Implement an LALR (Look-Ahead LR) parser generator derived from LR(1).
- Construct the LR(1) sets for the grammar.
- Merge states with identical core LR(1) sets to form LALR sets.
- Build the LALR parsing table using the merged LALR sets and lookaheads.
- Generate parser code that utilizes the LALR parsing table for shift-reduce parsing.

5. LALR(from LR(0)) Parser Generator:

- Implement an LALR parser generator derived from LR(0).
- Construct the LR(0) sets for the grammar.
- Merge states with identical core LR(0) sets to form LALR sets.
- Build the LALR parsing table using the merged LALR sets and follow sets.
- Generate parser code that utilizes the LALR parsing table for shift-reduce parsing.

6. Code Generation:

- Develop logic for generating parser code in the target programming language (C#, in this case).
- Emit code for parsing actions, error handling, and constructing the parse tree.
- Ensure the generated code adheres to the parsing table entries for SLR, LR(1), LALR(from LR(1)), and LALR(from LR(0)) parsers.

7. Integration and Usage:

- Provide interfaces or methods for users to integrate the generated parser into their projects.
- Offer clear documentation on how to use the parser generator and the generated parser code.
- Demonstrate examples of parsing with the generated parser for various grammatical constructs.

Putting it All Together:

- Design a cohesive class or module that encompasses the entire parser generator process.
- Enable users to specify grammars, choose parsing algorithms (SLR, LR(1), LALR), and generate parser code accordingly.
- Empower developers to seamlessly integrate and utilize the generated parsers for language processing in their applications.

Question 5: What problem did you face

1. Parsing Ambiguities:

- Issue: Ambiguities in the grammar can lead to multiple interpretations of the same input, causing difficulties in constructing a deterministic parsing table.
- Challenge: Resolving ambiguities requires careful grammar design and, in some cases, may necessitate additional disambiguating rules or parser adjustments.

2. Handling LR(0) and LR(1) Sets:

- Issue: Constructing LR(0) and LR(1) sets involves dealing with a potentially large number of states, leading to increased complexity.
- Challenge:Efficiently managing and analyzing these sets, especially in the case of LR(1), can be resource-intensive and may require optimization strategies.

3.Conflict Resolution:

- Issue: Conflicts, such as shift-reduce or reduce-reduce conflicts, can arise during the construction of the parsing table.
- Challenge: Identifying the cause of conflicts and resolving them without altering the language's expressive power is a critical challenge. It may involve heuristic rules or manual intervention.

4. Error Handling:

- Issue: Creating a parser that robustly handles syntax errors and provides meaningful error messages is essential but challenging.
- Challenge:Balancing between graceful error recovery and accurate reporting without compromising the efficiency of the parser requires thoughtful design and implementation.

5. Grammar Complexity:

- Issue: A complex grammar with a high number of rules can make the parser generator less user-friendly and more prone to errors.
- -Challenge: Simplifying the grammar, when possible, and providing clear guidelines for users to create grammars that work well with the parser generator.

6. Code Generation and Integration:

- Issue: Generating clean and efficient parser code in the target language (C#) and integrating it seamlessly into user projects.
- Challenge:Ensuring the generated code is readable, performs well, and aligns with the intended use cases, which may vary across different applications.

7. User Documentation:

- Issue: Inadequate or unclear documentation can hinder users' ability to understand and effectively use the parser generator.
- Challenge: Providing comprehensive and easily understandable documentation, including examples, usage guidelines, and troubleshooting tips.

8. Testing and Validation:

- Issue: Ensuring the correctness and efficiency of the parser generator through extensive testing.
- Challenge: Developing a robust testing strategy, covering a wide range of grammars and input scenarios, to validate the correctness and performance of the parser generator.

9. Graph Visualization Dependencies:

- Issue: The dependency on external tools, such as Graphviz, for graph visualization features.
- Challenge: Managing dependencies and ensuring users have the required tools installed can be a potential source of complications, especially across different operating systems.

10. Usability and User Experience:

- Issue: Ensuring a positive user experience and usability of the parser generator.
- Challenge: Designing user interfaces, error messages, and feedback mechanisms that are intuitive and enhance the overall user experience.