COMSATS Institute of
Information Technology

Group Members:

M. Abdullah Arshad (SP20-BCS-033)

Hasnain Ahmed (FA20-BCS-005)

Class/Section: BCS-7 (A)

Subject: CC-Lab (Compiler Construction)

Submission To: Sir Bilal Haider

Date: 28-Dec-2023

CC-Lab Terminal:

Question 1: Write brief of the project.

Answer: We have develop a mini compiler that performs Lexical Analysis and LR parsing of strings.

- **Lexical analyzer:** It breaks down the source code into a sequence of tokens. Tokens are the smallest units of meaning in a programming language, such as keywords, identifiers, literals, and operators.

  The main tasks of a lexical analyzer include:

- **Tokenization:** Breaking the source code into tokens based on the language's syntax rules.
- **Removing Whitespace and Comments:** Discarding elements like spaces, tabs, and comments that do not contribute to the meaning of the program.

  The specific actions performed by a lexical analyzer can vary depending on the programming language. Here's a general overview of what a lexical analyzer does in C#:
  - **Scanning:** Reads the source code character by character.
  - **Lexical Error Detection:** Identifies and reports lexical errors, such as misspelled keywords or undefined symbols.
  - **Token Generation:** Recognizes and generates tokens for keywords, identifiers, literals, operators, and other language constructs.

❖ **LR Parsing:** It based on a shift-reduce approach, where the parser shifts input symbols onto a stack until it identifies a sequence that can be reduced to a grammar production.

LR parsers main tasks include:

- **Shift Operation**: The parser shifts (moves) input symbols onto a stack until it identifies a valid right-hand side of a grammar production.
- **Reduce Operation**: Once a valid right-hand side is on top of the stack, the parser replaces that sequence with the corresponding non-terminal symbol of the grammar production. This is known as a reduce operation.
- **Acceptance**: The process continues until the parser accepts the entire input, indicating that the input adheres to the grammar rules.

Question 2: 2 functionalities along with Screen Shorts (Function Code + Output)

Answer:

❖ Lexical Analyzer (Implemented in C# Visual Studio):

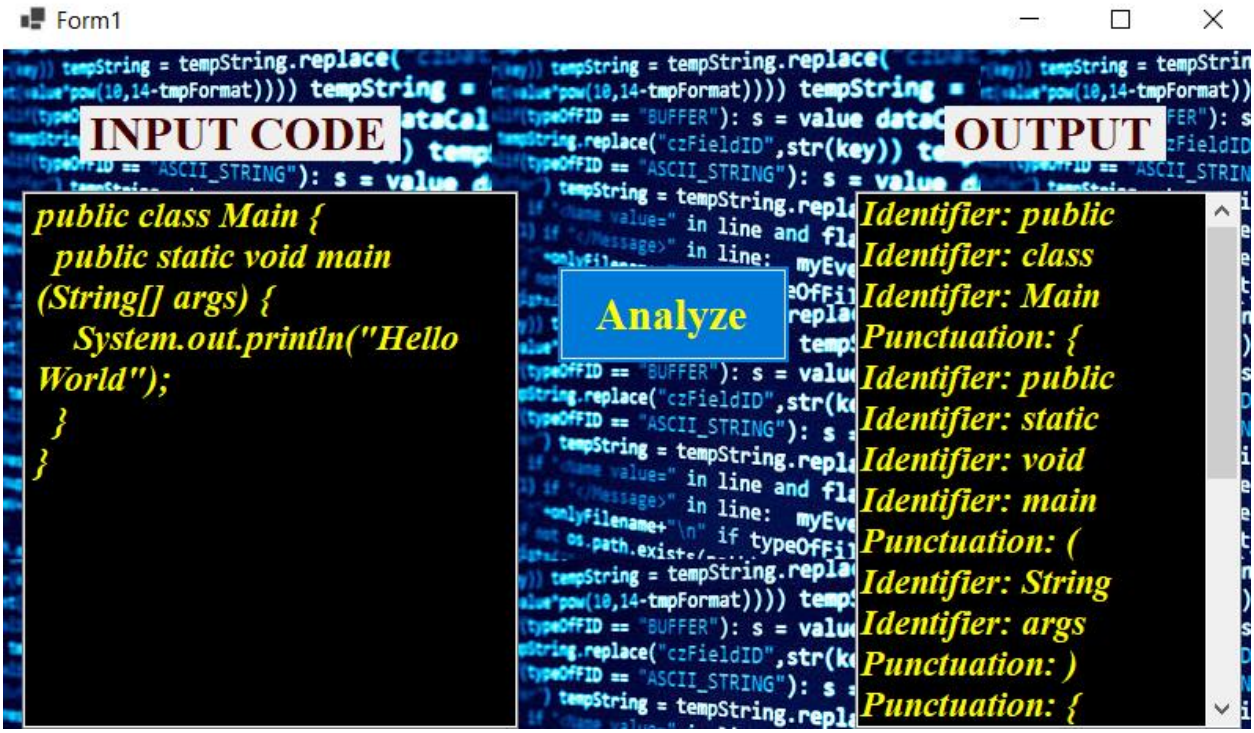
- **CODE:**

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Windows.Forms;
using System.Text.RegularExpressions;
namespace CC_TERMINAL_005_LEXICAL_ANALYZER_
{
   public partial class Form1 : Form
   {
      public Form1()
      {
         InitializeComponent();
      }

      private void btnAnalyze_Click(object sender, EventArgs e)
      {
         string input = txtInput.Text;
         List<Token> tokens = Analyze(input);

         // Display the tokens in the list box
         lstTokens.Items.Clear();
         foreach (Token token in tokens)
         {
            lstTokens.Items.Add(token);
         }

      }
      private List<Token> Analyze(string input)
      {
         List<Token> tokens = new List<Token>();

         // Define regular expressions for various token types
         var keywordRegex = new Regex(@"\b(if|else|while)\b");
         var identifierRegex = new Regex(@"\b[a-zA-Z_]\w*\b");
         var numberRegex = new Regex(@"\b\d+\b");
         var stringLiteralRegex = new Regex(@"""([^""\\]|\\.)*""");
         var operatorRegex = new Regex(@"[\+\-\*/=]");
         var punctuationRegex = new Regex(@"[{};,()]");
         var commentRegex = new Regex(@"\/\/.*|\/\*[\s\S]*?\*\/");
         var whitespaceRegex = new Regex(@"\s+");

         // Remove comments and whitespaces
         input = commentRegex.Replace(input, string.Empty);
         input = whitespaceRegex.Replace(input, " ");
```

```
        // Tokenize the input and keep track of positions
        var matches = keywordRegex.Matches(input);
        foreach (Match match in matches)
        {
            tokens.Add(new Token(match.Value, TokenType.Keyword, match.Index));
        }

        matches = identifierRegex.Matches(input);
        foreach (Match match in matches)
        {
            tokens.Add(new Token(match.Value, TokenType.Identifier, match.Index));
        }

        matches = numberRegex.Matches(input);
        foreach (Match match in matches)
        {
            tokens.Add(new Token(match.Value, TokenType.Number, match.Index));
        }

        matches = stringLiteralRegex.Matches(input);
        foreach (Match match in matches)
        {
            tokens.Add(new Token(match.Value, TokenType.StringLiteral, match.Index));
        }

        matches = operatorRegex.Matches(input);
        foreach (Match match in matches)
        {
            tokens.Add(new Token(match.Value, TokenType.Operator, match.Index));
        }

        matches = punctuationRegex.Matches(input);
        foreach (Match match in matches)
        {
            tokens.Add(new Token(match.Value, TokenType.Punctuation, match.Index));
        }

        // Sort tokens based on their positions
        tokens = tokens.OrderBy(t => t.Position).ToList();

        return tokens;
    }


}
public class Token
{
    public string Lexeme { get; }
    public TokenType Type { get; }
    public int Position { get; }  // Added position

    public Token(string lexeme, TokenType type, int position)
    {
        Lexeme = lexeme;
        Type = type;
        Position = position;
    }
```

```
        public override string ToString()
        {
            return $"{Type}: {Lexeme}";
        }
    }

    public enum TokenType
    {
        Keyword,
        Identifier,
        Number,
        StringLiteral,
        Operator,
        Punctuation,
        Comment
    }
}
```

- **OUPUT:**



❖ LR Parsing(Implemented in Python):

- **Code:**

```
from pprint import pprint

def import_grammar(fileHandle):
    G, T, Nt = [], [], []
    for lines in fileHandle:
        production = lines.split(' -> ')
        if production[0] not in Nt:
            Nt.append(production[0])
        listStr = list(production[1])
        del listStr[-1]
        production[1] = ''.join(i for i in listStr)
        for char in production[1]:
            if 65 <= ord(char) <= 90:
                if char not in Nt:
                    Nt.append(char)
            else:
                if char not in T:
                    T.append(char)
        if production not in G:
            G.append((production[0], production[1]))
    T.append('#')
    return G, T, Nt

def closure(I, G, Nt):
    J = [p for p in I]
    while True:
        J1 = [x for x in J]
        for x in J1:
            handle = list(x[1])
            k = handle.index('.')
            if k + 1 != len(handle):
                if handle[k + 1] in Nt:
                    for p in G:
                        if p[0] == handle[k + 1]:
                            new_p = (p[0], '.' + p[1])
                            if new_p not in J1:
                                J1.append(new_p)
        flag = True
        for x in J1:
            if x not in J:
                flag = False
                J.append(x)
        if flag:
            break
    return J

def goto(I, X, Nt):
    W = []
    for x in I:
        handle = list(x[1])
        k = handle.index('.')
        if k != len(handle) - 1:
            if handle[k + 1] == X:
                S1 = ''.join([handle[i] for i in range(k)])
                S2 = ''.join([handle[i] for i in range(k + 2, len(handle))])
                W.append((x[0], S1 + X + '.' + S2))
    return closure(W, G, Nt)
```

```
def items(G, T, Nt):
    C = [closure([(G[0][0], '.' + G[0][1])], G, Nt)]
    action = {}
    goto_k = {}
    reduction_states = {}
    while True:
        C1 = [I for I in C]
        for I in C1:
            for X in T:
                goto_list = goto(I, X, Nt)
                if len(goto_list) != 0 and goto_list not in C1:
                    C1.append(goto_list)
                    if C1.index(I) not in action:
                        action[C1.index(I)] = {}
                    if X not in action[C1.index(I)]:
                        action[C1.index(I)][X] = C1.index(goto_list)
                elif goto_list in C1:
                    if C1.index(I) not in action:
                        action[C1.index(I)] = {}
                    if X not in action[C1.index(I)]:
                        action[C1.index(I)][X] = C1.index(goto_list)
        for I in C1:
            for X in Nt:
                goto_list = goto(I, X, Nt)
                if len(goto_list) != 0 and goto_list not in C1:
                    C1.append(goto_list)
                    if C1.index(I) not in goto_k:
                        goto_k[C1.index(I)] = {}
                    if X not in goto_k[C1.index(I)]:
                        goto_k[C1.index(I)][X] = C1.index(goto_list)
                elif goto_list in C1:
                    if C1.index(I) not in goto_k:
                        goto_k[C1.index(I)] = {}
                    if X not in goto_k[C1.index(I)]:
                        goto_k[C1.index(I)][X] = C1.index(goto_list)
        flag = True
        for x in C1:
            if x not in C:
                flag = False
                C.append(x)
        if flag:
            break
    for P in G:
        Pp = (P[0], P[1] + '.')
        for state in range(len(C)):
            if Pp in C[state]:
                reduction_states[state] = P
    accept_state = 0
    for x in reduction_states:
        if reduction_states[x] == G[0]:
            accept_state = x
    return C, action, goto_k, reduction_states, accept_state

def parse_input_string(G, T, Nt, action_list, goto_list, reduction_states,
accept_state, input_str):
    stack = [0]
```

```
    i, top = 0, 0

    while True:
        s = stack[top]
        try:
            print(s, stack, input_str[i] if i != len(input_str) else
'Finish', end=' ')
            if s == accept_state:
                print('accept')
                break
            elif s in reduction_states:
                A, beta = reduction_states[s]
                print('reduce', A, '->', beta)
                for j in range(len(beta)):
                    del stack[top]
                t = stack[top]
                stack.insert(top, goto_list[t][A])
            else:
                a = input_str[i]
                stack.insert(top, action_list[s][a])
                print('shift', action_list[s][a])
                i = i + 1
        except Exception as e:
            print('Syntax error')
            break

if __name__ == "__main__":
    txt = input('Enter the name of the file: ') + '.txt'
    fileHandle = open(txt)
    G, T, Nt = import_grammar(fileHandle)
    print('Terminals:', T)
    print('Non-terminals:', Nt)
    C, action_list, goto_list, reduction_states, accept_state = items(G, T,
Nt)
    print('Action list')
    pprint(action_list)
    print('Goto list')
    pprint(goto_list)
    print('Reduction states')
    pprint(reduction_states)
    print('Accept state', accept_state)

    input_str = input('Enter some string: ') + '#'
    parse_input_string(G, T, Nt, action_list, goto_list, reduction_states,
accept_state, input_str)
```

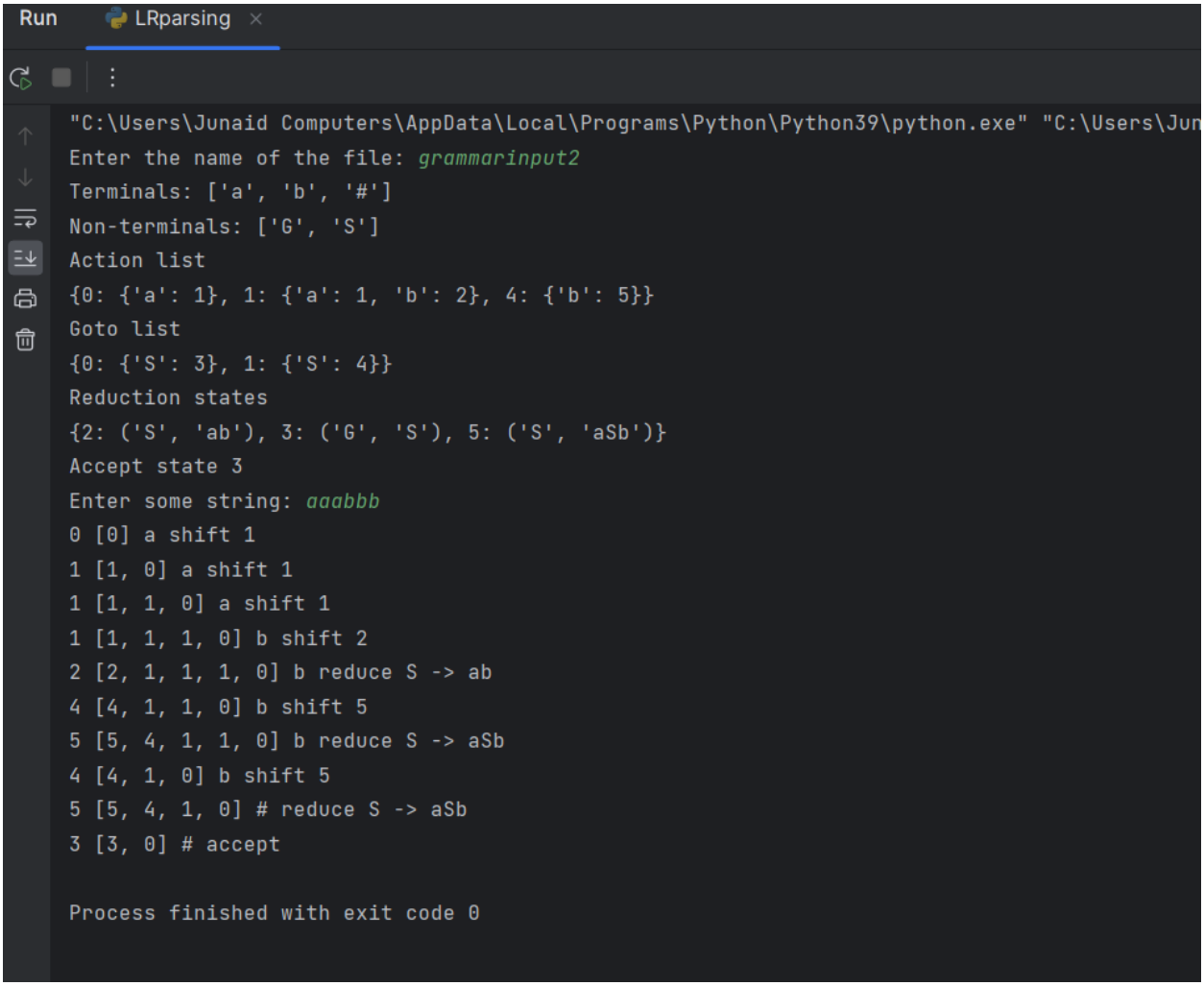
- **Output**
  - **Given Grammar:**

G -> S

S -> aSb

S -> ab

- **Testing String : aaabbb**

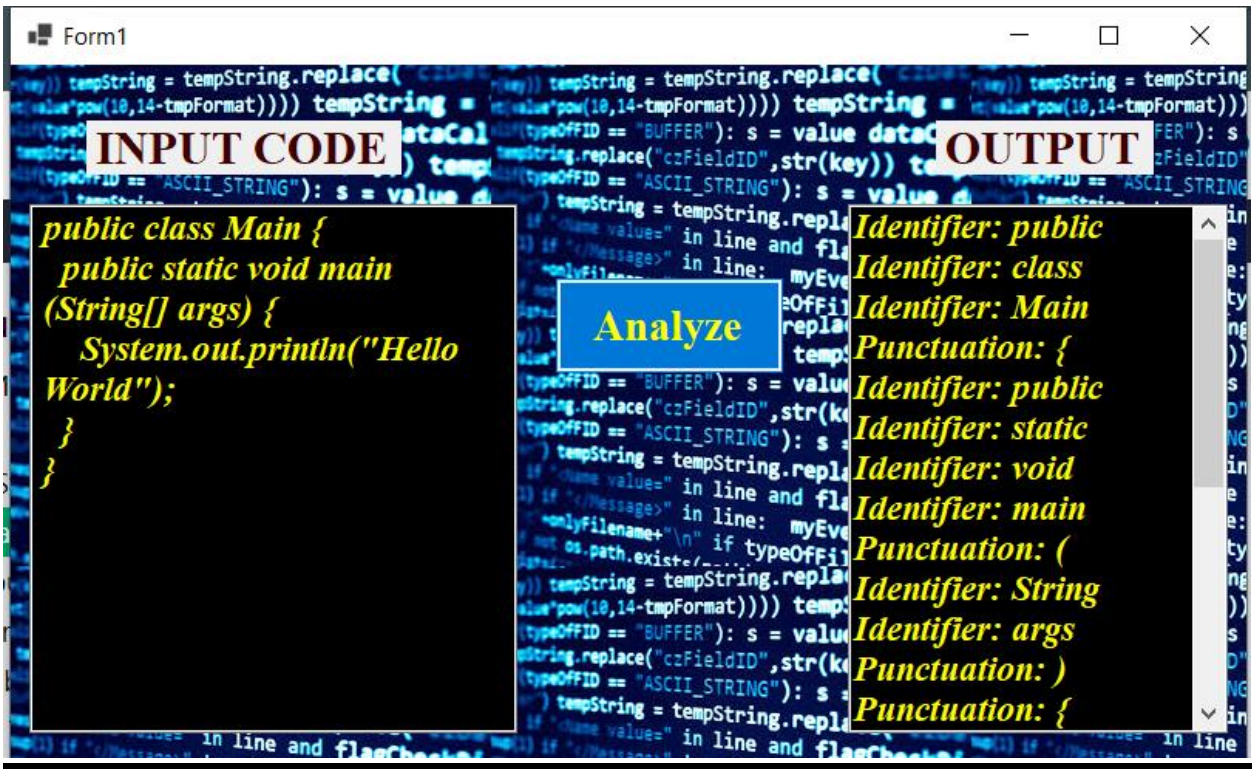


Question 3: Give Input and then show its Output.

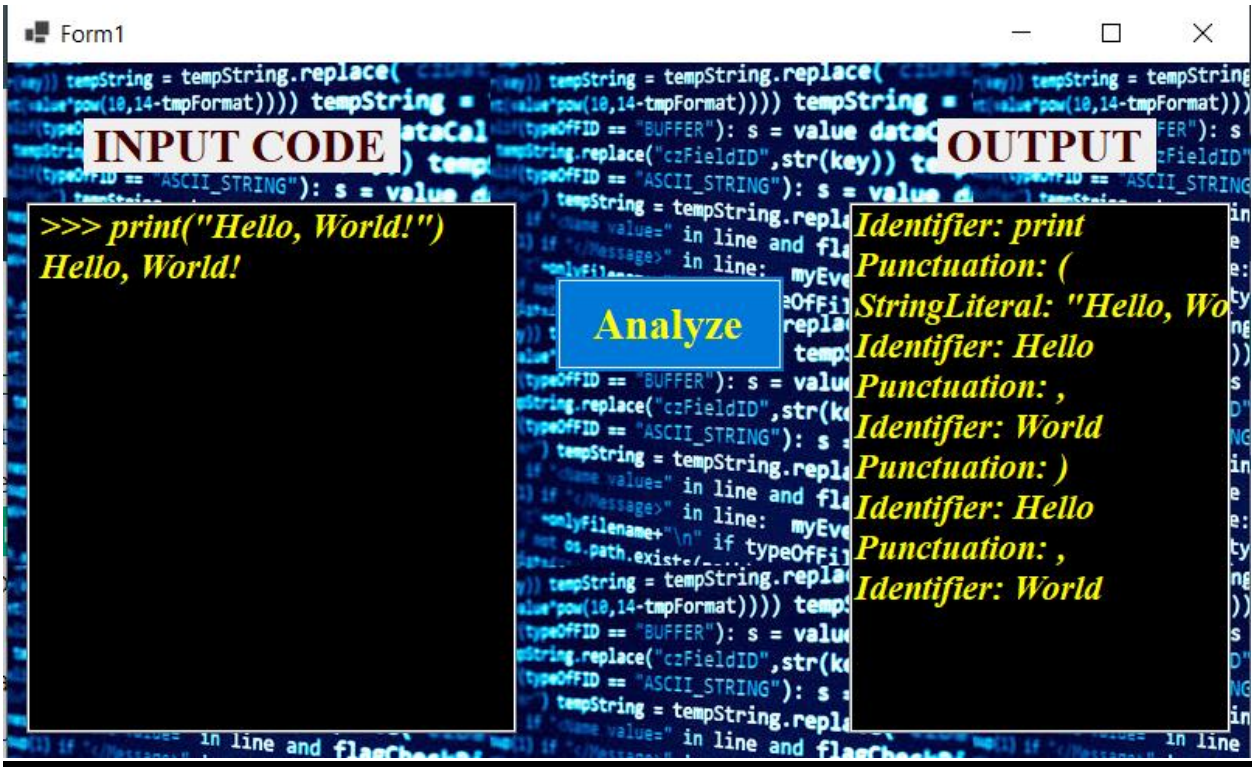
Answer:

❖Lexical analyzer:

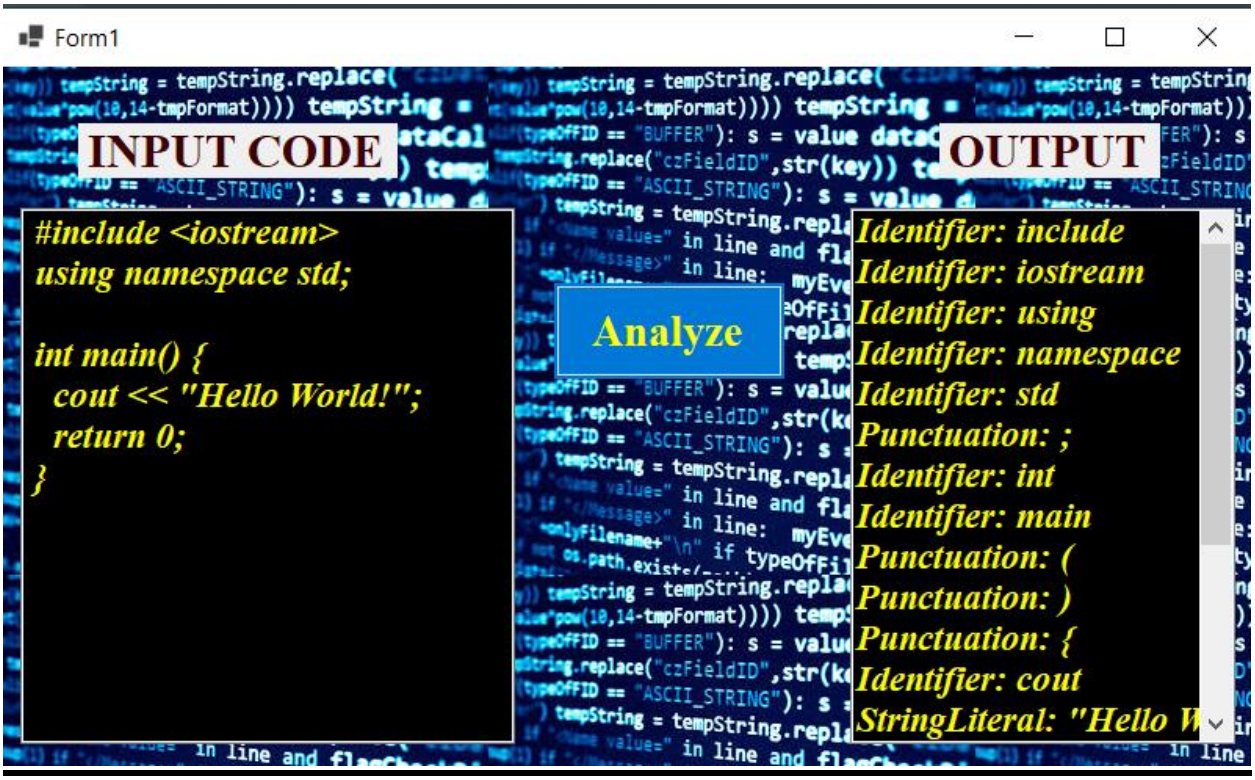
- **Input 1 & its Output:**



- **Input 2 & its Output:**



- **Input 3 & its Output:**



❖LR Parsing:

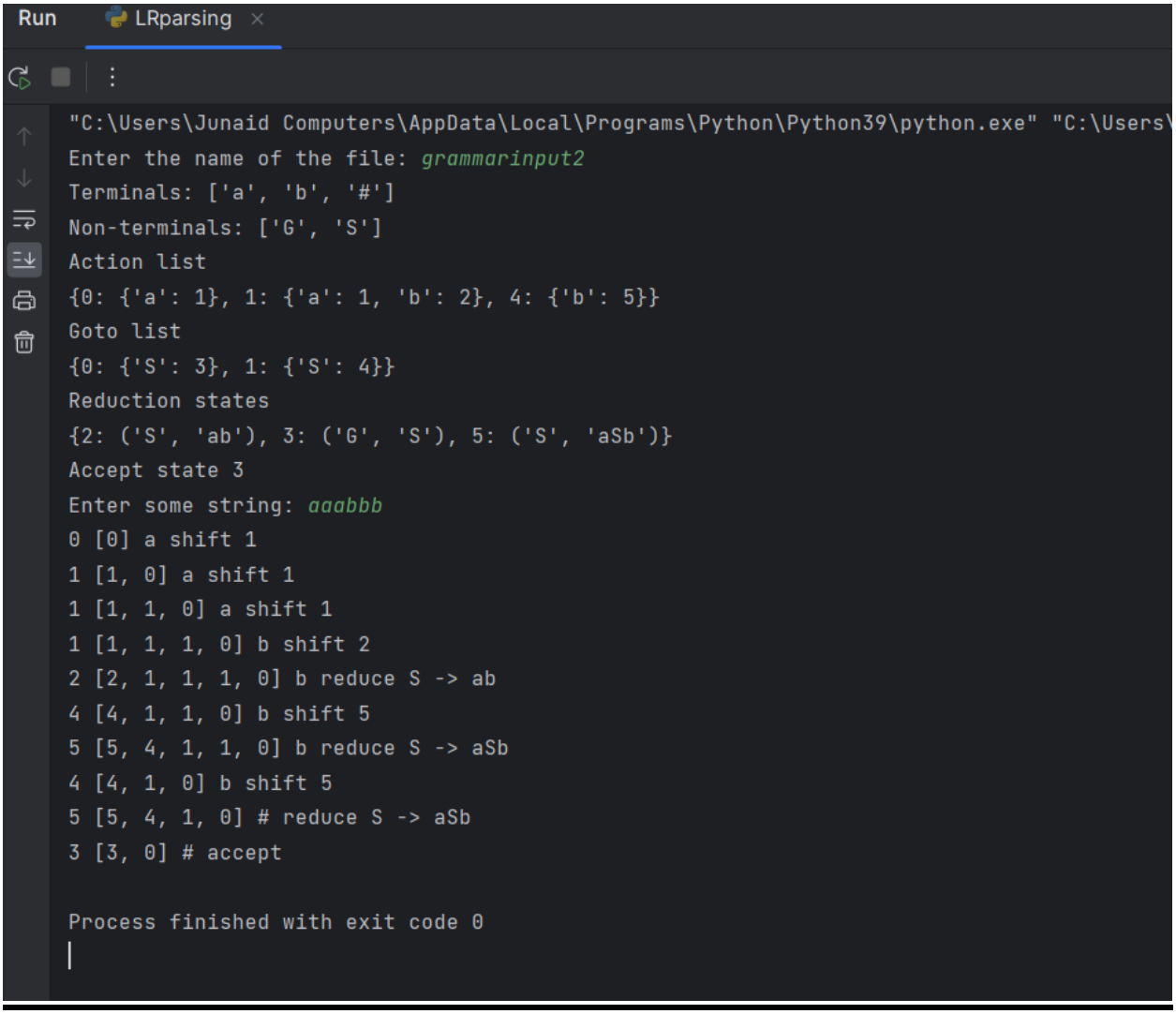
- **Grammar 1:**

G -> S

S -> aSb

S -> ab

  - **Test String**: aaabbb



- **Grammar 2:**

G -> S

S -> aB

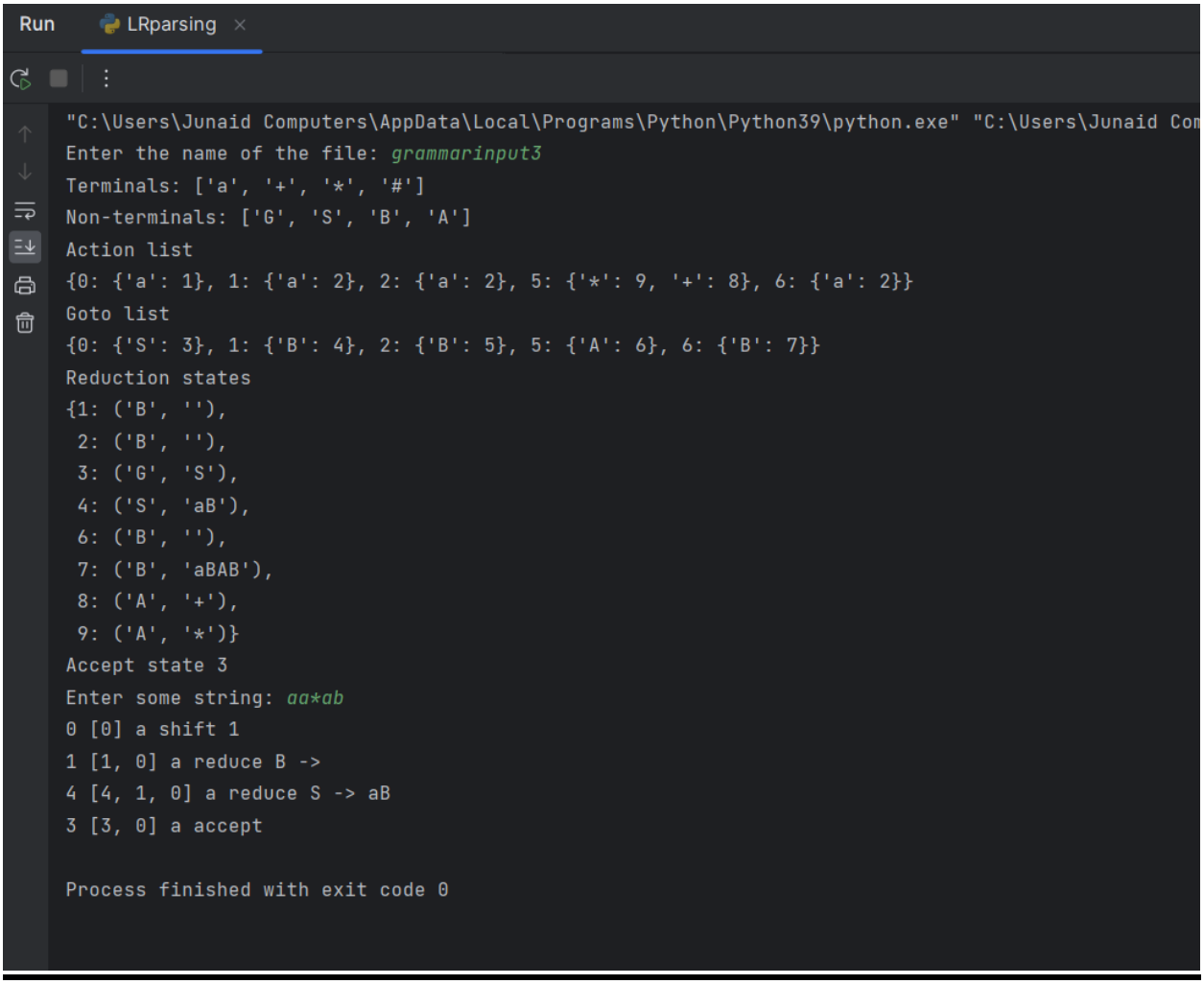
B -> aBAB

B ->

A -> +

A -> *

- **Test string** : aa*ab



- **Grammar 3:**

G -> E

E -> E+T

E -> E-T

E -> T

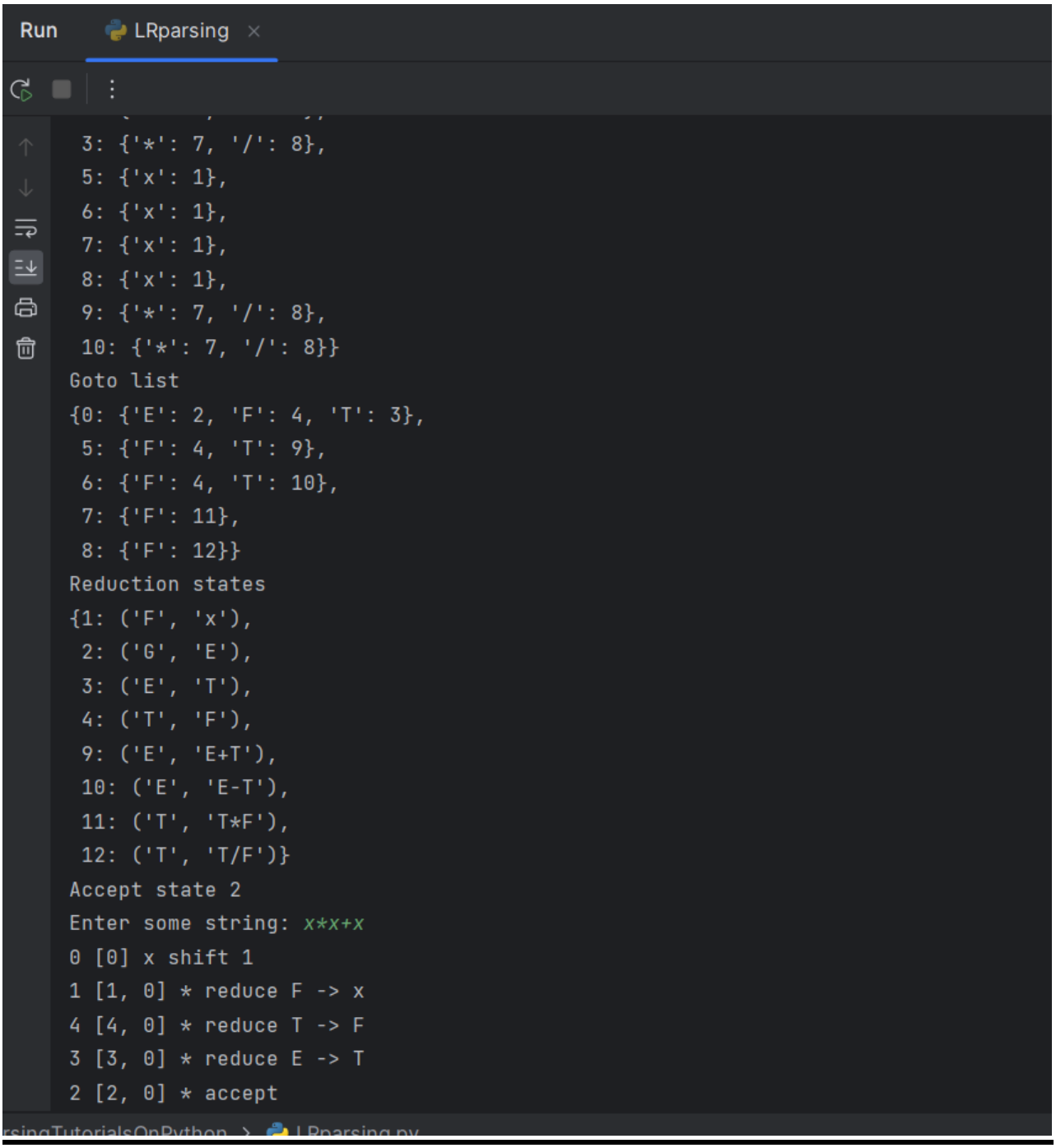
T -> T*F

T -> T/F

T -> F

F -> x

- **Test string :** x*x+x


Run
LRparsing
3: {'*': 7, '/': 8},
5: {'x': 1},
6: {'x': 1},
7: {'x': 1},
8: {'x': 1},
9: {'*': 7, '/': 8},
10: {'*': 7, '/': 8}}
Goto list
{0: {'E': 2, 'F': 4, 'T': 3},
5: {'F': 4, 'T': 9},
6: {'F': 4, 'T': 10},
7: {'F': 11},
8: {'F': 12}}
Reduction states
{1: ('F', 'x'),
2: ('G', 'E'),
3: ('E', 'T'),
4: ('T', 'F'),
9: ('E', 'E+T'),
10: ('E', 'E-T'),
11: ('T', 'T*F'),
12: ('T', 'T/F')}
Accept state 2
Enter some string: x*x+x
0 [0] x shift 1
1 [1, 0] * reduce F -> x
4 [4, 0] * reduce T -> F
3 [3, 0] * reduce E -> T
2 [2, 0] * accept

Question 4: Explain how functions works step by step?

Answer:

❖Lexical Analyzer:

- **Form1 Class:** Form1 is a class that represents a Windows Forms application.

It contains an event handler btnAnalyze_Click associated with the button named btnAnalyze. When this button is clicked, it triggers the analysis of the input text.

- **btnAnalyze_Click Event Handler:** This function is called when the "Analyze" button is clicked.

It retrieves the input text from a TextBox named txtInput.

Calls the Analyze function with the input text and obtains a list of Token objects.

Clears the items in a ListBox named lstTokens.

Iterates through the list of tokens and adds each token to the ListBox for display.

- **Analyze Function:** Analyze function takes a string input and returns a list of Token objects.

Initializes a list called tokens to store the identified tokens.

Defines regular expressions for various token types, such as keywords, identifiers, numbers, string literals, operators, punctuation, comments, and whitespaces.

Removes comments and whitespaces from the input string using regular expressions.

Tokenizes the input string based on the defined regular expressions and creates Token objects for each match.

The created tokens are added to the tokens list.

Finally, the tokens list is sorted based on the token positions in the input string and returned.

- **Token Class:** Token is a class representing a lexical token.

It has properties for Lexeme (the actual text of the token), Type (the type of the token), and Position (the position of the token in the input string).

A constructor is defined to initialize these properties when creating a new token.

The ToString method is overridden to provide a custom string representation of a token.

- **TokenType Enumeration:** An enumeration defining different types of tokens like Keyword, Identifier, Number, etc.

❖LR Parsing:

- **import_grammar(fileHandle) Function:** Reads a context-free grammar from a file and returns the grammar, terminals, and non-terminals.

G: List of productions.

T: List of terminals.

Nt: List of non-terminals.

It parses each line of the file to extract productions and builds the lists.

- **closure(I, G, Nt) Function:** Takes a set of LR(0) items I, the grammar G, and non-terminals Nt.

Computes the closure of the given set of LR(0) items by repeatedly expanding items until no more items can be added.

- **goto(I, X, Nt) Function:** Computes the GOTO set for LR(0) items.

Given a set of LR(0) items I, a symbol X, and non-terminals Nt, it calculates the set of items that can be obtained by shifting the dot over X.

- **items(G, T, Nt) Function:** Computes the LR(0) items for the grammar.

Returns the set of LR(0) items, the action table, goto table, reduction states, and the accept state.

It uses closure and goto to compute LR(0) items and transitions.

- **parse_input_string(G, T, Nt, action_list, goto_list, reduction_states, accept_state, input_str) Function:** Implements the LR(0) parsing algorithm for the given grammar and input string.

It initializes a stack and processes the input string based on the action and goto tables.

Outputs the parsing actions (shift, reduce, accept) and the production rules applied.

- **Main Execution Section (__main__):** Reads a grammar from a file specified by the user.

Calls import_grammar to get the grammar, terminals, and non-terminals.

Calls items to compute LR(0) items and related information.

Prints the terminals, non-terminals, action table, goto table, reduction states, and the accept state.

Asks the user to input a string for parsing.

Calls parse_input_string to parse the input string using the generated LR(0) parser.

Question 5: What challenges did you faced during this project?

Answer:

❖ Lexical analyzer:

- **Regular Expression Complexity:**

Designing accurate regular expressions for various token types can be challenging. Balancing precision and efficiency is crucial. It may take several iterations to fine-tune regular expressions to correctly identify tokens without introducing false positives or negatives.

- **Tokenization Logic:**

Developing a robust tokenization logic requires careful consideration of the order in which different token types are identified. For example, identifying keywords before identifiers ensures correct tokenization.

- **Handling Comments and Strings:**

Dealing with comments and string literals can be complex due to their multiline nature and escape characters. Ensuring that the regular expressions correctly capture these scenarios can be challenging.

- **Position Tracking:**

Adding the position information to tokens may require additional logic to accurately track the position of tokens in the input string. This is important for providing meaningful error messages or for later stages in the compiler where the position information is needed.

- **User Interface Integration:**

Integrating the lexical analyzer with a Windows Forms application involves understanding event-driven programming and UI components. Connecting the analyze button to the lexical analysis logic and displaying the results in the ListBox may require careful handling of events and user interface updates.

- **Testing and Debugging:**

Testing the lexical analyzer with various input scenarios is essential. Debugging issues related to incorrect tokenization or unexpected behavior can be time-consuming. Ensuring that the analyzer works correctly for edge cases and a variety of input types is crucial.

❖ LR Parsing:

- **Grammar Interpretation:**

Understanding and interpreting the grammar from a text file can be challenging. The structure and syntax might vary, and ensuring accurate parsing of each production rule requires careful attention.

- **Tokenization Logic:**

Designing a mechanism to tokenize the grammar rules correctly into terminal and non-terminal symbols can be intricate. Ensuring proper separation and identification of symbols in each production rule could be challenging.

- **Closure and Goto Functions:**

Implementing closure and goto functions involves handling the LR(0) items and transitions, which can be complex. Determining the closure of a set of items or computing the goto sets accurately requires a deep understanding of parsing algorithms like LR parsing.

- **State Generation:**

Generating states and constructing the LR(0) automaton involves managing sets of items and transitions. Arriving at the correct states and transitions while handling diverse grammars might pose challenges.

- **Action and Goto Tables:**

Populating the action and goto tables accurately for the LR(0) parser requires meticulous tracking of states and symbols. Any error in table construction can lead to incorrect parsing decisions.

- **Error Handling:**

Dealing with syntax errors or unexpected input can be complex. Ensuring that the parser provides informative error messages when encountering invalid input is essential for usability.

- **Testing and Debugging:**

Thoroughly testing the parser with various grammars and input strings is crucial. Debugging issues related to incorrect parsing, unexpected behavior, or erroneous table entries might be time-consuming.

-End