كلية علوم الحاسب والمعلومات
College of Computer and Information Sciences

Al-Imam Muhammad Bin Saud Islamic University

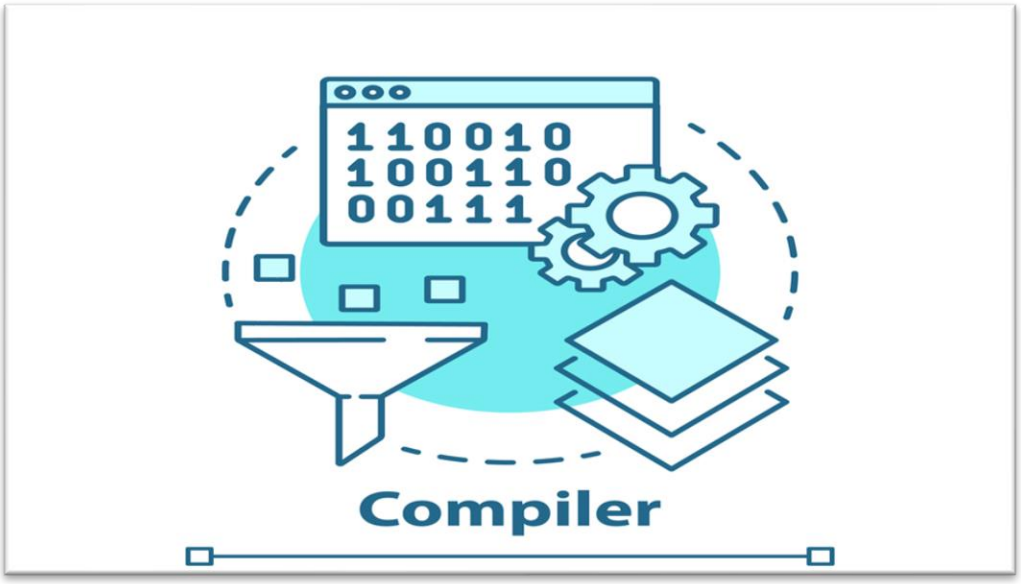
College of Computer and Information Sciences

Department of Computer Science

CS 445 Compiler

Course project

Nov 5th, 2022



Section: 373

Name	Email	ID
Wea'am Alghaith	wksalghait@sm.imamu.edu.sa	440023306
Khloud Alnufaie	khaalnufaie@sm.imamu.edu.sa	440020617
Raghad Albosais	rkaalbosais@sm.imamu.edu.sa	440020209
Alhanouf Almansour	aalmansour83@sm.imamu.edu.sa	440019183
Ameerah Alsuhaibani	aalsuhaibani29@sm.imamu.edu.sa	440019129

Index of content:

Index of figure:

List of tables:

1.OVERIEW OF THE PROJECT

1.1 Description

In this project we utilized what we learn in lectures to implement two programs divided into two parts. Part one is program deal with regular expression to construct finite automata or recognizer that take string 'x' and answer 'accept' if x is a sentence of the language, and 'not accept' otherwise. Finite automata can be deterministic (DFA) or non-deterministic (NFA). Part two is program deal with grammar to implement LL(1) parser. LL(1) parser is non-recursive predictive parsing where the parse tree is created top to down for creates the syntactic structure of the given token. It is used to check for syntax error during the syntax analysis phase.

2. STEP OF IMMPLEMENTION

2.1 Part one

2.1.1 Builds an NFA from a given regular expression.

- For build NFA from given regular expression first we define class *automata* where the finite automata attribute is initialized the start state and final state and language and transitions of automata.

```
4   class Automata:
5       """class to represent an Automata"""
6       #constructer for init
7       def __init__(self, language = set(['0', '1'])):
8           self.states = set() #init state by set
9           self.startstate = None
10          self.finalstates = []
11          self.transitions = dict()
12          self.language = language #insert language
13
```

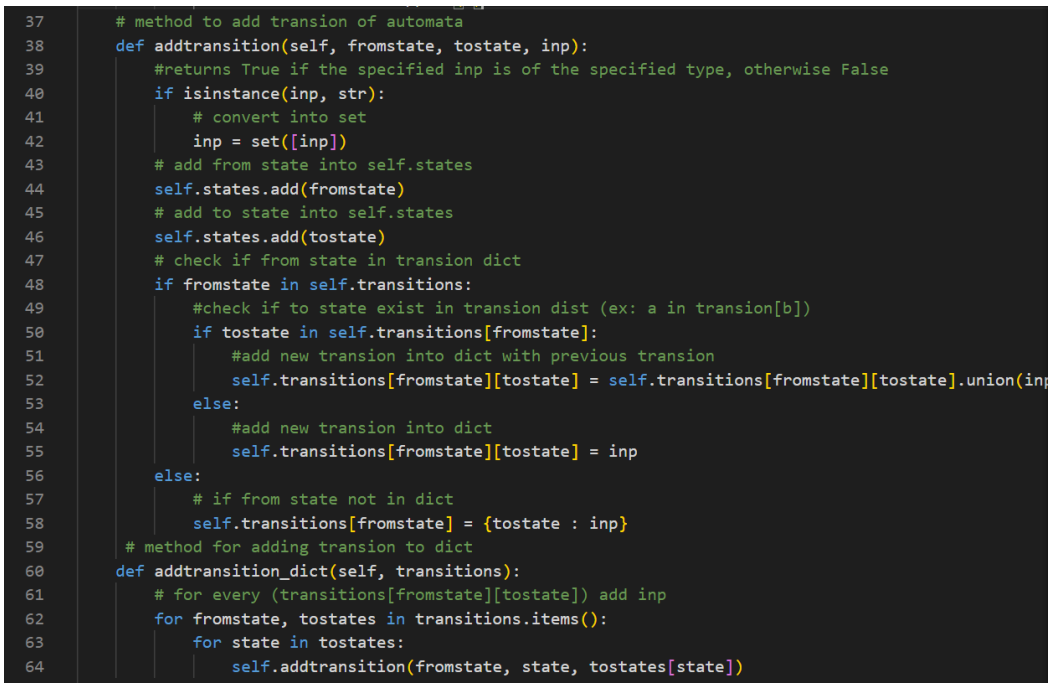
Figure 1. class automata

- In class automata we define help method for build NFA . First epsilon() definition for get epsilon sign . Second addfinalstates() definition for adding the final state into the list. Third startstate() to adding start state to startstate attribute.

```
14      #method for return epsilon
15      @staticmethod
16      def epsilon():
17          return ":e:"
18
19      # method to establish start state
20      def setstartstate(self, state):
21          self.startstate = state
22          # add to state set
23          self.states.add(state)
24
25
26      # method to insert final states
27      def addfinalstates(self, state):
28          #returns True if the specified state is of the specified type, otherwise False
29          if isinstance(state, int):
30              # add state to array state
31              state = [state]
32          # to add every final state in array to self.finalstates
33          for s in state:
34              if s not in self.finalstates:
35                  # add final to array
36                  self.finalstates.append(s)
```

Figure 2. class automata helper definitions

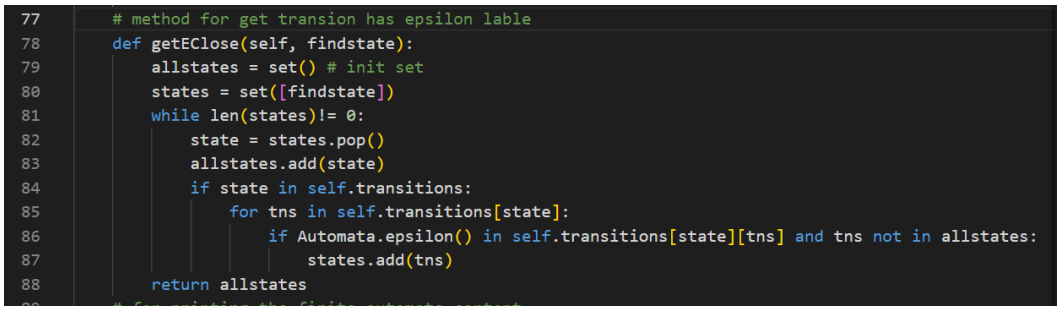
- Another two-helper definition in class automata responsible on add transitions of NFA to dictionary.



```
    # method to add transion of automata
    def addtransition(self, fromstate, tostate, inp):
        #returns True if the specified inp is of the specified type, otherwise False
        if isinstance(inp, str):
            # convert into set
            inp = set([inp])
        # add from state into self.states
        self.states.add(fromstate)
        # add to state into self.states
        self.states.add(tostate)
        # check if from state in transion dict
        if fromstate in self.transitions:
            #check if to state exist in transion dist (ex: a in transion[b])
            if tostate in self.transitions[fromstate]:
                #add new transion into dict with previous transion
                self.transitions[fromstate][tostate] = self.transitions[fromstate][tostate].union(inp
            else:
                #add new transion into dict
                self.transitions[fromstate][tostate] = inp
        else:
            # if from state not in dict
            self.transitions[fromstate] = {tostate : inp}
     # method for adding transion to dict
    def addtransition_dict(self, transitions):
        # for every (transitions[fromstate][tostate]) add inp
        for fromstate, tostates in transitions.items():
            for state in tostates:
                self.addtransition(fromstate, state, tostates[state])
```

Figure 3. addtransition() and addtransition_dict() definitions

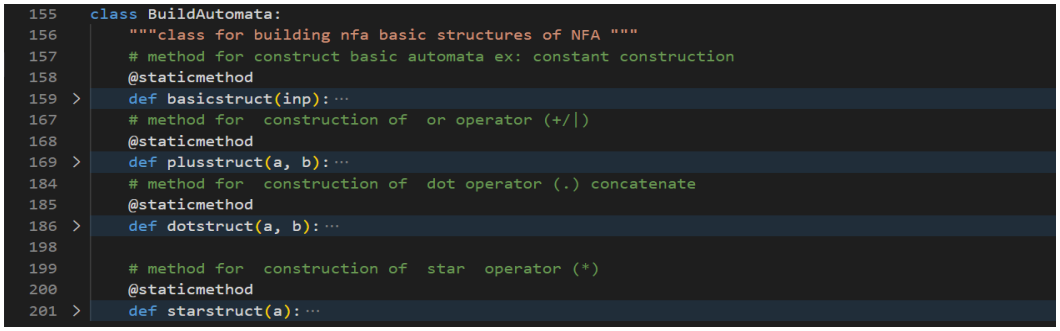
- An definition called getEClose() for find the transitions have epsilon label to find e-closure of state in NFA.



```
    # method for get transion has epsilon lable
    def getEClose(self, findstate):
        allstates = set() # init set
        states = set([findstate])
        while len(states)!= 0:
            state = states.pop()
            allstates.add(state)
            if state in self.transitions:
                for tns in self.transitions[state]:
                    if Automata.epsilon() in self.transitions[state][tns] and tns not in allstates:
                        states.add(tns)
        return allstates
```

Figure 4. getEClose() definition

- In class BuildAutomata there are four definitions for construct the basic of NFA such as constant, plus, dot and star. By call method from automata class called newBuildFromNumber() for construct the NFA components .



```
class BuildAutomata:
    """class for building nfa basic structures of NFA """
    # method for construct basic automata ex: constant construction
    @staticmethod
    def basicstruct(inp):···
    # method for  construction of  or operator (+/|)
    @staticmethod
    def plusstruct(a, b):···
    # method for  construction of  dot operator (.) concatenate
    @staticmethod
    def dotstruct(a, b):···

    # method for  construction of  star  operator (*)
    @staticmethod
    def starstruct(a):···
```

Figure 5. class BuildAutomata

```
114     # method for construction
115     def newBuildFromNumber(self, startnum):
116         translations = {} # init dict
117         for i in list(self.states): # for evry state
118             translations[i] = startnum # add state with number
119             startnum += 1
120         rebuild = Automata(self.language) # creat another automata
121         rebuild.setstartstate(translations[self.startstate]) # set start state
122         rebuild.addfinalstates(translations[self.finalstates[0]]) # set final states
123         for fromstate, tostates in self.transitions.items(): # add tranisions
124             for state in tostates:
125                 rebuild.addtransition(translations[fromstate], translations[state], tostates[state])
126         return [rebuild, startnum]
```

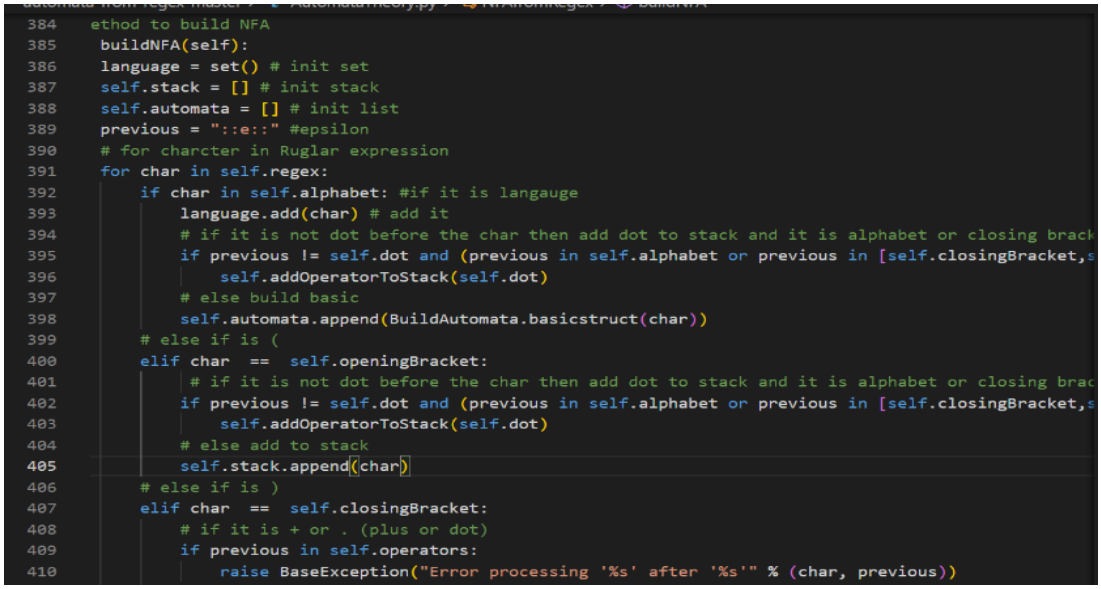
Figure 6. newBuildFromNumber() definition

- we initialize in constructor the possible input symbol appears in regular expression .

```
360 class NFAfromRegex:
361     """class for building nfa from regular expressions"""
362     # constructor
363     def __init__(self, regex):
364         self.star = '*'
365         self.plus = '+'
366         self.dot = '.'
367         self.openingBracket = '('
368         self.closingBracket = ')'
369         self.operators = [self.plus, self.dot]
370         self.regex = regex
371         self.alphabet = [chr(i) for i in range(65,91)] # from A to Z
372         self.alphabet.extend([chr(i) for i in range(97,123)]) # from a to z
373         self.alphabet.extend([chr(i) for i in range(48,58)]) #symbol
374         self.buildNFA()
375 
376     #method for get NFA object
377     def getNFA(self):
378         return self.nfa
379 
```

Figure 7. NFAfromRegex class

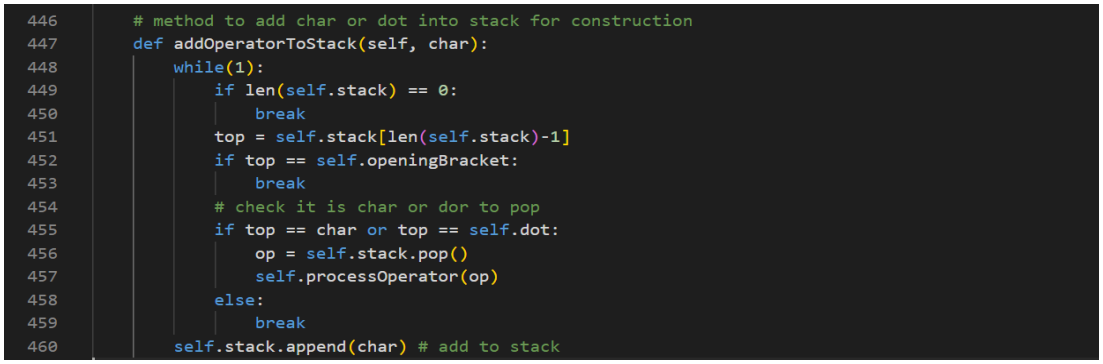
- The buildNFA() definition have a stack from list and automata from list and attribute previous set up with epsilon. The method check every character in regular expression if it is alphabet add it to language list then check if the previous character not dot and it is alphabet or closing bracket or star and then call addOperatorToStack() with dot . the method continuo check every character with previous to append to stack and build NFA.



```
384   ethod to build NFA
385    buildNFA(self):
386    language = set() # init set
387    self.stack = [] # init stack
388    self.automata = [] # init list
389    previous = "::e::" #epsilon
390    # for charcter in Ruglar expression
391    for char in self.regex:
392        if char in self.alphabet: #if it is langauge
393            language.add(char) # add it
394            # if it is not dot before the char then add dot to stack and it is alphabet or closing brack
395            if previous != self.dot and (previous in self.alphabet or previous in [self.closingBracket,s
396                self.addOperatorToStack(self.dot)
397            # else build basic
398            self.automata.append(BuildAutomata.basicstruct(char))
399        # else if is (
400        elif char  ==  self.openingBracket:
401            # if it is not dot before the char then add dot to stack and it is alphabet or closing brac
402            if previous != self.dot and (previous in self.alphabet or previous in [self.closingBracket,s
403                self.addOperatorToStack(self.dot)
404            # else add to stack
405            self.stack.append(char)
406        # else if is )
407        elif char  ==  self.closingBracket:
408            # if it is + or . (plus or dot)
409            if previous in self.operators:
410                raise BaseException("Error processing '%s' after '%s'" % (char, previous))
```

Figure 8. buildNFA() definition

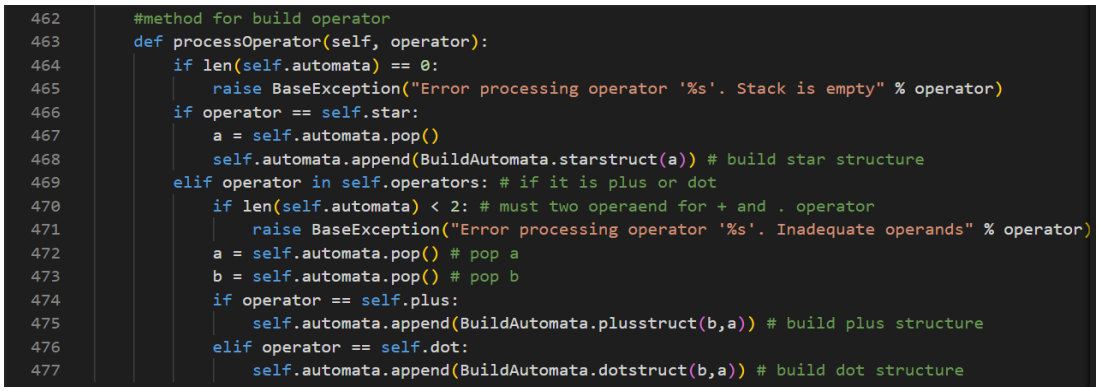
- The definition we mentioned above addOperatorToStack() take dot or plus to construct them in NFA.



```
446     # method to add char or dot into stack for construction
447     def addOperatorToStack(self, char):
448         while(1):
449             if len(self.stack) == 0:
450                 break
451             top = self.stack[len(self.stack)-1]
452             if top == self.openingBracket:
453                 break
454             # check it is char or dor to pop
455             if top == char or top == self.dot:
456                 op = self.stack.pop()
457                 self.processOperator(op)
458             else:
459                 break
460         self.stack.append(char) # add to stack
```

Figure 9. addOperatorToStack() defintion

- To process the build of NFA there are processOperator() for call the helping method in BuildAutomata class.



```
462     #method for build operator
463     def processOperator(self, operator):
464         if len(self.automata) == 0:
465             raise BaseException("Error processing operator '%s'. Stack is empty" % operator)
466         if operator == self.star:
467             a = self.automata.pop()
468             self.automata.append(BuildAutomata.starstruct(a)) # build star structure
469         elif operator in self.operators: # if it is plus or dot
470             if len(self.automata) < 2: # must two operaend for + and . operator
471                 raise BaseException("Error processing operator '%s'. Inadequate operands" % operator)
472             a = self.automata.pop() # pop a
473             b = self.automata.pop() # pop b
474             if operator == self.plus:
475                 self.automata.append(BuildAutomata.plusstruct(b,a)) # build plus structure
476             elif operator == self.dot:
477                 self.automata.append(BuildAutomata.dotstruct(b,a)) # build dot structure
```

Figure 10. processOperator()

- Also in NFAfromregex class there are two definition responsible for draw the graph and install it .

```
479  # method for drawing graph
480  def drawGrap                    ):
481      """From https://github.com/max99x/automata-editor/blob/master/util.py"""
482      f = popen(r"dot -Tpng -o graph%s.png" % file, 'w')
483      try:
484          f.write(automata.getDotFile()) # get graph and write in f file
485      except:
486          raise BaseException("Error creating graph") # error
487      finally:
488          f.close() # close f file
489
490  #method for install image of graph
491  def isInstalled(program):
492      """From http://stackoverflow.com/questions/377017/test-if-executable-exists-in-python"""
493      import os
494      def is_exe(fpath):
495          return os.path.isfile(fpath) and os.access(fpath, os.X_OK)
496      fpath, fname = os.path.split(program)
497      if fpath:
498          if is_exe(program) or is_exe(program+".exe"):
499              return True
500      else:
501          for path in os.environ["PATH"].split(os.pathsep):
502              exe_file = os.path.join(path, program)
503              if is_exe(exe_file) or is_exe(exe_file+".exe"):
504                  return True
505      return False
```

Follow link (ctrl + click)

Figure11 .drawGraph() and isInstalled() defintion

2.1.2 Converts a giving NFA into a DFA.

- By use previous classes to construct DFA we define DFAfromNFA class take the NFA as input and convert to DFA .

```
217  class DFAfromNFA:
218      """class for building dfa from nfa and minimise it"""
219      # constructor
220      def __init__(self, nfa):
221          self.buildDFA(nfa)
222
223
224      #method for get DFA object
225      def getDFA(self):
226          return self.dfa
```

Figure 12. DFAfromNFA class

- We define buildDFA() definition by initialize dictionary called allstate to and counter for key . and another dictionary is initialized for closure of state. Then we follow the steps of convert NFA to DFA by use this dictionaries . as shown in figure 13 the rest of the code in source code section

```
238     # method to build DFA from NFA
239     def buildDFA(self, nfa):
240         allstates = dict() # init dict for states
241         eclose = dict() #init dict for closure
242         count = 1
243         state1 = nfa.getEClose(nfa.startstate) # first step : find e-closure of inital state
244         eclose[nfa.startstate] = state1 # let's e-closure of inital state is state 1
245         dfa = Automata(nfa.language) # get posible input
246         dfa.setstartstate(count) # init or set start state
247         states = [[state1, count]] # insert into list
248         allstates[count] = state1 # insert int dict of all state in grapgh ex:(1,{0,1,3,4})
249         count +=  1 # for next state
250         while len(states) != 0:
251             [state, fromindex] = states.pop()
252             for char in dfa.language:
253                 # to find transion
254                 trstates = nfa.gettransitions(state, char)
255                 # loop for check move with e to get next state set
256                 for s in list(trstates)[:]:
257                     if s not in eclose:
258                         eclose[s] = nfa.getEClose(s)
259                     trstates = trstates.union(eclose[s])
260                 if len(trstates) != 0:
261                     if trstates not in allstates.values():
262                         states.append([trstates, count]) # insert into list
263                         allstates[count] = trstates
264                         toindex = count
265                         count +=  1 #for next state
```

Figure 13. buildDFA() defintion

- In the last we add acceptsString() definition to check the string is 'accept' if it is a sentence of the language, and 'not accept' otherwise.

```
273     # method for check if string is accepted or not
274     def acceptsString(self, string):
275         currentstate = self.dfa.startstate  # start from start state
276         for ch in string:
277             if ch==":e:": # if it is epsilon
278                 continue
279             st = list(self.dfa.gettransitions(currentstate, ch)) # else move depend on inp
280             if len(st) == 0: # not belong to set of strings
281                 return False
282             currentstate = st[0] # continue
283         if currentstate in self.dfa.finalstates: # if its reach to accept state or final
284             return True # belong to set of string
285         return False # in case dose not reach to accept state or final
```

Figure 14. acceptsString() definition

2.1.3 Builds a DFA from a given regular expression directly.

- Our first step in implementation is to import the library we need to build our code.

```
1   #ply.lex is an implementation of lex parsing tools for Python.
2   import ply.lex as lex
3   # networkx package for the Python used to create, manipulate graph networks used to draw tree
4   import networkx as nx
5   import matplotlib.pyplot as plt
6   # import to draw graph of DFA
7   from AutomataTheory import *
```

Figure 15. import library

- The second step we defined data type depend on the token that possible to appear in regular expression. Then write the possible entities of these type.

```
# Tokens
tokens = (
    'OPEN_P',
    'CLOSED_P',
    'OR',
    'STAR',
    'CONCAT',
    'LETTER',
    'LAMBDA'
)

# Rules that define the tokens
t_OPEN_P = r'\('
t_CLOSED_P = r'\)'
t_OR = r'\+'
t_STAR = r'\*'
t_CONCAT = r'\.'
t_LETTER = r'[a-zA-Z]'
t_LAMBDA = r'\#'
```

Figure 16.data type initialize

- Class RegexLexer is class build for construct syntax tree get regular expression concatenate it with #. Then divided into token and add it to list.

```
class RegexLexer:
    # A lexer to tokenize the content
    lexer = lex.lex()
    # A container containing the tokens
    parts = []

    # constructer
    def __init__(self, inp):
            self.content = '(' + inp + ')#'
            self.tokenize()


   # method of add tokens into list
    def tokenize(self):
        self.lexer.input(self.content)
        while True:
            token = self.lexer.token()
            if not token: # No more input
                break
            self.parts.append(token)

    # for printing tokens
    def printTokens(self):
        for part in self.parts:
            print(part)
```

Figure 17. class RegexLexer

- In the class RegexLexer we build _addConcats() definition to add dot between the concatenate token.

```
61      # method add concantion (dot) step 2 in coversion
62      def _addConcats(self):
63          possibleConcat = False
64          concatMessage = ''
65          index = 0
66
67          for part in self.parts:
68              if part.type in ['LETTER', 'LAMBDA']: # operand or #
69                  if possibleConcat is True:
70                      concatMessage = concatMessage + '.' # add dot
71                  else:
72                      possibleConcat = True
73              elif part.type in ['STAR', 'CLOSED_P']:# * or )
74                  possibleConcat = True
75              elif (part.type == 'OPEN_P') & (self.parts[index - 1].type in ['LETTER', 'STAR']):
76                  concatMessage = concatMessage + '.'
77              else:
78                  possibleConcat = False
79              concatMessage = concatMessage + part.value
80              index = index + 1
81
82          self.content = concatMessage # after add dot
83          self.parts.clear()
84          self.tokenize()
```

Figure 18. _addConcats() definition

- For build syntax tree we need to convert the regular expression into postfix notation or reverse polish notation where that its operators come after the corresponding operands as shown in Figure19 and the rest in source code section.
- We build _makeAST() definition to draw the syntax tree as shown in Figure 20 and the rest of code in source code section .

```
86      # to write in postfix notation by use reverse polish notation
87      #where every operator follows all of its operands
88      def writeAsRPN(self, permanent=False):
89          if self.content[len(self.content) - 1] == '.':
90              print("Already in rpn notation")
91              return
92          # First we need to have the regex written in reverse polish notation ot postfix
93          operator_stack = [] # init list
94          last_index = -1 # pointer of stack
95          rpn_regex = ''
96          for part in self.parts:
97              if part.type in ['LETTER', 'STAR', 'LAMBDA']:
98                  rpn_regex = rpn_regex + part.value # add to RE
99                  continue
100             if part.type == 'OPEN_P':
101                 operator_stack.append(part) # put in stack
102                 last_index += 1
103                 continue
104             if part.type == 'CLOSED_P':
105                 while operator_stack[last_index].type != 'OPEN_P': # to pop until (
106                     rpn_regex += operator_stack[last_index].value # add to RE
107                     operator_stack.pop() # pop from stack
108                     last_index -= 1
109                 # Doing one more pop to  extract the open parenthesis
110                 operator_stack.pop()
111                 last_index -= 1
112                 continue
```

Figure 19. writeAsRPN() definition

```
140     def makeAST(self):
141         self._addConcats()
142         print("\nRegex after adding concatenations: " + self.content + '\n')
143         self.printTokens()
144 
145         print("\nRegex written in reverse polish notation: " + self.writeAsRPN(permanent=True) + '\n')
146 
147         AST = nx.DiGraph() # init tree
148         node_stack = []
149         last_index = -1
150 
151         for part in self.parts:
152             if part.type in ['LETTER', 'LAMB  value: Any  operand or #
153                 AST.add_node(part, val=part.value) # add node
154                 node_stack.append(part) # add to stak
155                 last_index += 1
156             elif part.type in ['OR', 'CONCAT']: # add + or .
157                 AST.add_node(part, val=part.value)
158                 for i in range(0, 2): # two operand
159                     AST.add_edge(part, node_stack[last_index]) # add edage
160                     node_stack.pop() # pop operand from stack
161                     last_index -= 1
162                 node_stack.append(part) # add inner node
163                 last_index += 1
164             elif part.type == 'STAR':
165                 AST.add_node(part, val=part.value) # add node
166                 AST.add_edge(part, node_stack[last_index]) # add edage
167                 node_stack.pop() # pop from stake
```

Figure 20. _makeAST() definition

- Class converter build to take the regular expression after postfix notation and syntax tree to extracts the function that help in DFA construction.

```
184 # class convert RE to DFA
185 class Converter:
186     FirstPos = []
187     LastPos = []
188     FollowPos = []
189 
190     DFA = nx.MultiDiGraph()
191 
192     #constructor
193     def __init__(self, rpn_tokens: list):
194 
195         self.rpn_tokens = rpn_tokens
196         self._makeFirstPos()
197         self._makeLastPos()
198         self._makeFollowPos()
```

Figure 21. class conventer

- In class converter we build _makeFirstPos() definition to extract the First pos of node in syntax tree and add it to list as shown in figure 22 and rest of code in source code section .
- In class converter we build _makeLasttPos() definition to extract the Last pos of node in syntax tree and add it to list as shown in figure 23 and rest of code in source code section .
- In class converter we build _makeFollowPos() definition to extract the Follow pos of node in syntax tree and add it to list. as shown in figure 24 and rest of code in source code section.

```
200     def _makeFirstPos(self):
201         """
202         This method is used to compute the first_pos list for each token.
203 
204         There are special cases that we need to address: when a branch could generate both a sequence from a wor
205         or nothing at all. In this case the "sub-word" that is generated by the sub-tree could start on both bra
206         Since here we are building only the first-pos collection we are interested in left side sub-trees only.
207 
208         :return: nothing
209         """
210         index = 0  # Used to copy positions from previous nodes
211         letter_no = 1  # Each letter in the regex is associated a number in the order of appearance
212         nullable = []
213 
214         for token in self.rpn_tokens:
215             if token.type in ['LETTER', 'LAMBDA']: #if it is operand or #
216                 newpos = [letter_no]
217                 letter_no += 1
218                 index += 1
219                 self.FirstPos.append(newpos)
220                 if token.type == 'LAMBDA':
221                     if token.lexpos == len(self.rpn_tokens) - 2:
222                         nullable.append(False)
223                     else:
224                         nullable.append(True)
225                 else:
226                     nullable.append(False)
```

Figure 22. _makeFirstPos() definition

```
277     def _makeLastPos(self):
278         index = 0  # Used to copy positions from previous nodes
279         letter_no = 1  # Each letter in the regex is associated a number in the order of appearance
280         nullable = []
281 
282         for token in self.rpn_tokens:
283             if token.type in ['LETTER', 'LAMBDA']: #if ot is operand or #
284                 newpos = [letter_no]
285                 letter_no += 1
286                 index += 1
287                 self.LastPos.append(newpos)
288                 if token.type == 'LAMBDA': # if it is #
289                     if token.lexpos == len(self.rpn_tokens) - 2:
290                         nullable.append(False)
291                     else:
292                         nullable.append(True)
293                 else:
294                     nullable.append(False)
295 
296             elif token.type == 'OR':
297                 newpos = self.LastPos[index - 2].copy() + self.LastPos[index - 1].copy()
298                 self.LastPos.append(newpos)
299                 index += 1
300                 if nullable[index - 3] | nullable[index - 2]:
301                     nullable.append(True)
302                 else:
303                     nullable.append(False)
304 
305             elif token.type == 'STAR':
```

Figure 23. _makeLastPos() definition

```
342     def _makeFollowPos(self):
343         index = 0
344         number_nop = 0
345 
346         # Counting the number of operand and #
347         for token in self.rpn_tokens:
348             if token.type in ['LETTER','LAMBDA']:
349                 number_nop += 1
350 
351         self.FollowPos = [[] for _ in range(number_nop)]
352 
353         for token in self.rpn_tokens:
354             if token.type == 'STAR':
355                 for pos in self.LastPos[index]:
356                     newpos = self.FirstPos[index].copy()
357                     self.FollowPos[pos - 1] += newpos
358                 index += 1
359             elif token.type == 'CONCAT':
360                 if self.rpn_tokens[index - 1].type == 'STAR':
361                     nb = index - 2
362                     while True:
363                         if self.rpn_tokens[nb].type in ['CONCAT', 'OR']:
364                             nb -= 2
365                         else:
366                             nb -= 1
367                             break
368                     for pos in self.LastPos[nb]:
```

Figure24 . _makeFollowPos() definition

2.2 Part two

In part two, we create LL (1) parser using the following grammar:

1) PROGRAM → STMTS
2) STMTS → STMT| STMT ; STMTS
3) STMT → id = EXPR
4) EXPR → EXPR + TERM | EXPR - TERM | TERM
5) TERM → TERM * FACTOR | TERM / FACTOR | FACTOR
6) FACTOR → (EXPR) | id | integer

In section 2.2.1 we build the parser manually. The code implementation of all steps to create the LL (1) parser are illustrated in section 2.2.2.

2.2.1 Build LL (1) parser

2.2.1.1 Add the power and unary operators

1) PROGRAM → STMTS
2) STMTS → STMT| STMT ; STMTS
3) STMT → id = EXPR
4) EXPR → EXPR + TERM | EXPR - TERM | TERM
5) TERM → TERM * POWER | TERM / POWER | POWER
6) POWER → FACTOR ^ POWER | FACTOR
7) FACTOR → (EXPR) | id | integer | @ FACTOR | ! FACTOR

Where, @: represent + unary and !: represent – unary.

2.2.1.2 Re-write grammar to be LL (1) grammar

Criteria	Check
Operator precedence	Done
Operator associativity Left associativity (+, -, *, /, (), =) Right associativity (^, @,!)	Done
Left recursion	Rule 4 & 5
Left factoring	Rule 2 & 6

Table 1 Criteria to convert the grammar to LL (1)

1) PROGRAM → STMTS
2) STMTS → STMT STMTS'
3) STMTS' → # | ; STMTS
4) STMT → id = EXPR
5) EXPR → TERM EXPT'
6) EXPR' → + TERM EXPR' | - TERM EXPR' | #
7) TERM → POWER TERM'
8) TERM' → * POWER TERM' | / POWER TERM' | #
9) POWER → FACTOR POWER'
10) POWER' → ^ POWER | #
11) FACTOR → (EXPR) | id | integer | @ FACTOR | ! FACTOR

2.2.1.3 Implement LL (1) parser

- Encode words into letters

  P: PROGRAM

  S: STMTS

  L: STMTS'

  M: STMT

  E: EXPR

  R: EXPR'

  T: TERM

  N: TERM'

  O: POWER

  W: POWER'

  F: FACTOR

- Four-Tuples (Start symbol, non-terminal, terminal, production rules)

Start Symbol	P
Non-terminals	{P, S, L, M, E, R, T, N, O, W, F}
Terminals	{; , id, =, +, -, *, /, ^, (,), integer, @, !}

Production Rules	{
	1) P → S
	2) S → M L
	3) L → # | ; S
	4) M → id = E
	5) E → T R
	6) R → + T R | - T R | #
	7) T → O N
	8) N → * O N | / O N | #
	9) O → F W
	10) W → ^ O | #
	11) F → (E) | id | integer | @ F | ! F
	}

Table 2 Four-Tuples of the grammar

- First and Follow functions

Production Rules	First	Follow
1) **P → S**	id	$
2) **S → M L**	id	$
3) **L → # | ; S**	#, ;	$
4) **M → id = E**	id	;, $
5) **E → T R**	(, id, integer, @, !	;, $,)
6) **R → + T R | - T R | #**	+, -, #	;, $,)
7) **T → O N**	(, id, integer, @, !	+, -, ;, $,)
8) **N → * O N | / O N | #**	*, /, #	+, -, ;, $,)
9) **O → F W**	(, id, integer, @, !	*, /, +, -, ;, $,)
10) **W → ^ O | #**	^, #	*, /, +, -, ;, $,)
11) **F → (E) | id | integer | @ F | ! F**	(, id, integer, @, !	^, *, /, +, -, ;, $,)

Table 3 First and Follow functions result

- Parsing table

	;	id	=	+	-	*	/	^	(	)	integer	@	!	$
P		P->S												
S		S->ML												
L	L->;S													L->#
M		M->id=E												
E		E->TR							E->TR		E->TR	E->TR	E->TR	
R	R->#			R->+TR	R->-TR					R->#				R->#
T		T->ON							T->ON		T->ON	T->ON		
N	N->#			N->#	N->#	N->*ON	N->/ON			N->#				N->#
O		O->FW							O->FW		O->FW	O->FW	O->FW	
W	W->#			W->#	W->#	W->#	W->#	W->^O		W->#				W->#
F		F->id							F->(E)		F->integer	F->@F	F->!F	

Table 4 Parsing table result

2.2.2 Code implementation

Firstly, the grammar will be stored in rules variable along with the non-terminals in nonterm_userdef and terminals in term_userdef. We assume that the start symbol is the first rule, and we store it in start_symbol variable. Then, the grammar will be processed to be in format suitable for the code and store it in diction dictionary. In order to implement the LL (1) parser, we need to compute first and follow functions, and store them in firsts and follows dictionaries. Then use them to build the parser table. By doing so, it can check the validity of any input string using stack data structure. The First and follow functions are illustrated in section 2.2.2.1 and 2.2.2.2, respectively. Section 2.2.2.3 illustrate the building of parser table and stack implementation in section 2.2.2.4. test cases of the code is provided in section 2.2.2.5.

2.2.2.1 Calculate first: computeAllFirst()

- The computeAllFirsts() fuction executes computing the all the firsts of given rules by calling helper method first().

```
168 def computeAllFirsts():
169     global rules, nonterm_userdef, \
170         term_userdef, diction, firsts
```

*Figure parameters of **computeAllFirsts()** method*

- The global variables initiated are the same as the ones initiated for first().
-

```
172     # clean all rules (strip and split rules, eliminate left recursion, eliminate left factoring)
173     preprocesssing(rules, diction)
```

*Figure preprocessing in **computeAllFirsts()** method*

- The preprocessing takes rules and diction.
- First step is to apply preprocessing to eliminate left recursion, left factoring strip and split rules.

```
175     # calculate first for each rule
176     # - (call first() on all RHS)
177     for y in list(diction.keys()):
178         t = set()
179         for sub in diction.get(y):
180             res = first(sub)
181             if res != None:
182                 if type(res) is list:
183                     for u in res:
184                         t.add(u)
185                 else:
186                     t.add(res)
```

Figure calling first() on RHS

- Second step is to call the first() helper method for each rule and that is done nesting two loops the outer in line 177 goes through every diction key and the second one goes through each element in that diction key in line 179. Then the first method is called for each element in the outer element y. After checking if fist exists, If the first is a list then add each element using the loop in line 183 if not a list then add directly using .add().

```
188         # save result in 'firsts' list
189         firsts[y] = t
190
```

Figure saving in firsts

- Saving all in firsts

```
191     print("\nCalculated firsts: ")
192     key_list = list(firsts.keys())
193     index = 0
194     for gg in firsts:
195         print(f"first({key_list[index]}) "
196             f"=> {firsts.get(gg)}")
197         index += 1
```

*Figure printing of **computeAllFirsts()** method*

- Finally, we'll have to get through each element in firsts to print it.

2.2.2.2 Calculate follow: computeAllFollows()

```
200 def computeAllFollows():
201     global start_symbol, rules, nonterm_userdef,\
202         term_userdef, diction, firsts, follows
```

Figure 25 computeAllFollows() method

The computeAllFollows() define the basic variables and dictionary as global to share the results among all methods. It perform the process of calculate follow function with one helper method, follow(), and store the results in follow dictionary. For every non-terminal, we loop through them and store the intermediate results in the solset, sol and List.

```
204     for NT in diction:
205         solset = set()
206         sol = None
207         List = []
```

Figure 26 Step1 to calculate follow in computeAllFollows() method

- In this block of if and else statements, we compute follow function. The first and if statement handle a special case when there is a recursive during computing the follow. For example, follow(A) = follow(B) and follow(B) = follow(A). This case only happens in rules number 2&3, for the first if statement, and 9&10, for the second if statement. Whereas in else statement, it compute the follow in natural case (no recursive during compute the follow) with helping method: follow() , it will be illustrated with more details below. In each case, we store the results in sol variable.

```
209         if NT == 'S' or NT == "L":
210             for i in follows[start_symbol]:
211                 List.append(i)
212             sol = List
213         elif NT == 'O' or NT == "W":
214             first_N_ = firsts["N"]
215             for i in first_N_:
216                 if i == '#':
217                     continue
218                 List.append(i)
219             for j in follows['T']:
220                 List.append(j)
221             sol = List
222         else:
223             sol = follow(NT)
```

Figure 27 Step2 to calculate follow in computeAllFollows() method

- After computing the follow for the current non-terminal and store the results in sol list. We loop through the sol list and add it in solset to remove any duplications. Then, we assign the solset to the follows dictionary, where the key is the cuurent non-terminal.

```
225     if sol is not None:
226         for g in sol:
227             solset.add(g)
228     follows[NT] = solset
```

Figure 28 Step3 to calculate follow in computeAllFollows() method

follow() helper method

```
88  def follow(nt):
89      global start_symbol, rules, nonterm_userdef, \
90          term_userdef, diction, firsts, follows
91
92      solset = set()
```

Figure 29 Parameters of follow() helper method

The follow() define the basic variables and dictionary as global to share the results among all methods. It take one parameter: nt, which represent the non-terminal to be computed inside the first for loop in computerAllFollow(). In Line 92, it defines the solset to store all element of the follow result.

- When the given nt is the start symbol, we all $ to its follow's elements.

```
94      # for start symbol return $ (recursion base case)
95      if nt == start_symbol:
96          # return '$'
97          solset.add('$')
```

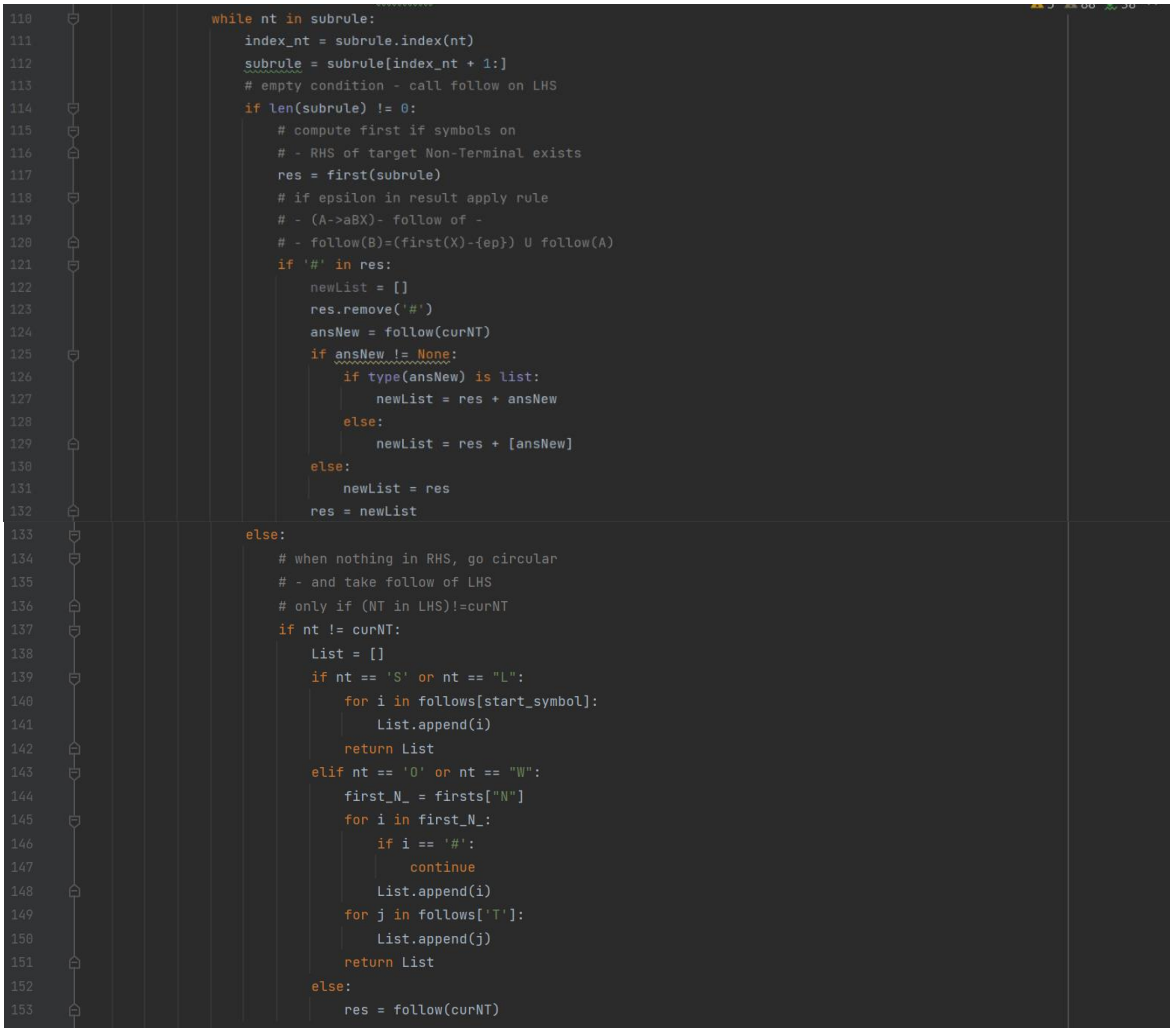
Figure 30 Calculate follow for start symbol in follow() helper method

- We loop through all rules in Line 103, to see if they have the given nt in their RHS or net. The inner loop in Line 106 pass through every options or sub rules in RHS for the current non-terminal curNT in the outer loop. We enter the inner loop only if the given nt appear in the RHS of the curNT. The if

statement in Line 107 consists of while loop. It is responsible to continues computing the follow of nt if it is appeared many times in the RHS of the curNT.

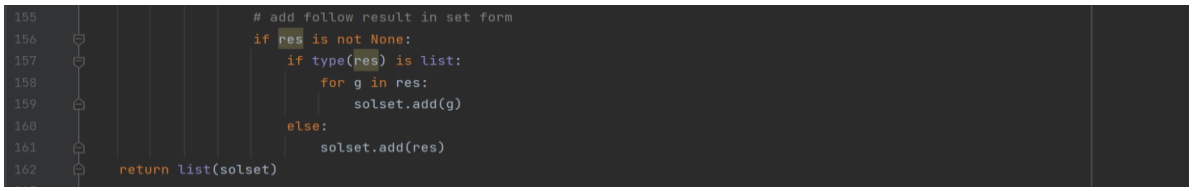
Figure 31 Initial settings to calculate follow in follow() helper method

- Inside the while statement in Line 110 as illustrated in Figure 32, we find the index of the next position of the nt. To check if there is any terminal or non-terminal after the nt. Line 114 to Line 132 handle the case when there is a terminal or non-terminal. And Line 133 to Line 153 handle the case when there is nothing after the nt. This process is continued while there is terminal or non-terminal after the nt until no more terminal or non-terminal exist. We store the result in res variable, then the added to solset from Line 156 to Line 161 to return the solset in Line 162 to computeAllFollows() method as illustrated in Figure 33.



```
110             while nt in subrule:
111                 index_nt = subrule.index(nt)
112                 subrule = subrule[index_nt + 1:]
113                 # empty condition - call follow on LHS
114                 if len(subrule) != 0:
115                     # compute first if symbols on
116                     # - RHS of target Non-Terminal exists
117                     res = first(subrule)
118                     # if epsilon in result apply rule
119                     # - (A->aBX)- follow of -
120                     # - follow(B)=(first(X)-{ep}) U follow(A)
121                     if '#' in res:
122                         newList = []
123                         res.remove('#')
124                         ansNew = follow(curNT)
125                         if ansNew != None:
126                             if type(ansNew) is list:
127                                 newList = res + ansNew
128                             else:
129                                 newList = res + [ansNew]
130                         else:
131                             newList = res
132                         res = newList
133                 else:
134                     # when nothing in RHS, go circular
135                     # - and take follow of LHS
136                     # only if (NT in LHS)!=curNT
137                     if nt != curNT:
138                         List = []
139                         if nt == 'S' or nt == "L":
140                             for i in follows[start_symbol]:
141                                 List.append(i)
142                             return List
143                         elif nt == 'O' or nt == "W":
144                             first_N_ = firsts["N"]
145                             for i in first_N_:
146                                 if i == '#':
147                                     continue
148                                 List.append(i)
149                             for j in follows['T']:
150                                 List.append(j)
151                             return List
152                         else:
153                             res = follow(curNT)
```

Figure 32. Overview of the calculate follow process in follow() helper method



```
155                 # add follow result in set form
156                 if res is not None:
157                     if type(res) is list:
158                         for g in res:
159                             solset.add(g)
160                     else:
161                         solset.add(res)
162     return list(solset)
```

Figure 33. Store and return the results in follow() helper method

- For the case when there is a terminal or non-terminal after nt, such as A → aBb (nt is B), the follow of B will be the first(b), as shown in Line 117. But if there is epsilon ϵ in the first(b), we remove the epsilon ϵ and add the follow(A), as shown from Line 121 to .

```
114                 if len(subrule) != 0:
115                     # compute first if symbols on
116                     # - RHS of target Non-Terminal exists
117                     res = first(subrule)
118                     # if epsilon in result apply rule
119                     # - (A->aBX)- follow of -
120                     # - follow(B)=(first(X)-{ep}) U follow(A)
121                     if '#' in res:
122                         newList = []
123                         res.remove('#')
124                         ansNew = follow(curNT)
125                         if ansNew != None:
126                             if type(ansNew) is list:
127                                 newList = res + ansNew
128                             else:
129                                 newList = res + [ansNew]
130                         else:
131                             newList = res
132                         res = newList
```

Figure 34 Hnadling the first case in follow() helper method

- For the case when there is nothing after nt, such as A → aB (nt is B), we go circular to find the follow of the LHS, in this case, A, only if the LHS not equal the curNT. We compute follow function for the LHS from Line 139 to Line 153. The first if statement handle a special case when there is a recursive during computing the follow. For example, follow(A) = follow(B) and follow(B) = follow(A). This case only happens in rules number 2&3, for the first if statement, and 9&10, for the second if statement. Whereas in else statement, it compute the follow in natural case (no recursive during compute the follow) by recursively the follow() method call itself.

```
133             else:
134                 # when nothing in RHS, go circular
135                 # - and take follow of LHS
136                 # only if (NT in LHS)!=curNT
137                 if nt != curNT:
138                     List = []
139                     if nt == 'S' or nt == "L":
140                         for i in follows[start_symbol]:
141                             List.append(i)
142                         return List
143                     elif nt == 'O' or nt == "W":
144                         first_N_ = firsts["N"]
145                         for i in first_N_:
146                             if i == '#':
147                                 continue
148                             List.append(i)
149                         for j in follows['T']:
150                             List.append(j)
151                         return List
152                     else:
153                         res = follow(curNT)
```

Figure 35 Handling the second case in follow() helper method

2.2.2.3 Build parse table: createParseTable()

```
236  # create parse table
237  def createParseTable():
238      import copy
239      global diction, firsts, follows, term_userdef
```

Figure 36 createParseTable() method

The createParseTable() define the basic variables and dictionary as global to share the results among all methods. It consists of two steps. The first step is to perform the process of build the parse table using matrix. The second step is converting the matrix to string to be used in GUI. After that, the method will return the results.

1-Create the parse table.

- Establish the initial setting before build the parse table. First from Line 267 to Line 269, we define two lists, one for the non-terminals and the other for the terminals with $. Second from Line 272 to Line 278, we define a matrix of size [non-terminals x (terminals + 1 for $)] with empty entry.

```
265      # create matrix of row(NT) x [col(T) + 1($)]
266      # create list of non-terminals
267      ntlist = list(diction.keys())
268      terminals = copy.deepcopy(term_userdef)
269      terminals.append('$')
270
271      # create the initial empty state of ,matrix
272      mat = []
273      for x in diction:
274          row = []
275          for y in terminals:
276              row.append('')
277          # of $ append one more col
278          mat.append(row)
```

Figure 37 Initial settings to create the parse table in createParseTable() method

- The outer loop Line 284 pass through all rules, each rule has unique non-terminal in its LHS that represent one of the rows in the matrix. In each rule, we store its RHS (one subrule or set of subrules) in rhs. The inner loop in Line 286 pass through the rhs list to full the column entries table for the corresponding non-terminal. For the current subrule, y, we add this subrule to all entries where the columns match all terminals in first(y). The first(y) terminals are stored in res from Line 305 to Line 308. We add additional terminals if the first(y) contain epsilon ϵ. The if statement block from Line 290 to Line 303 132 handle the case when there is epsilon ϵ in the first(y). After that, from Line 309 to Line 322 we full the subrule, y, in the matrix for the corresponding terminals in res. This process is continued for every terminal's rule to full the rows.

```
283     # rules implementation
284     for lhs in diction:
285         rhs = diction[lhs]
286         for y in rhs:
287             res = first(y)
288             # epsilon is present,
289             # - take union with follow
290             if '#' in res:
291                 if type(res) == str:
292                     firstFollow = []
293                     fol_op = follows[lhs]
294                     if fol_op is str:
295                         firstFollow.append(fol_op)
296                     else:
297                         for u in fol_op:
298                             firstFollow.append(u)
299                     res = firstFollow
300                 else:
301                     res.remove('#')
302                     res = list(res) +\
303                         list(follows[lhs])
304             # add rules to table
305             ttemp = []
306             if type(res) is str:
307                 ttemp.append(res)
308                 res = copy.deepcopy(ttemp)
309             for c in res:
310                 xnt = ntlist.index(lhs)
311                 yt = terminals.index(c)
312                 if mat[xnt][yt] == '':
313                     mat[xnt][yt] = mat[xnt][yt] \
314                                 + f"{lhs}->{' '.join(y)}"
315                 else:
316                     # if rule already present
317                     if f"{lhs}->{y}" in mat[xnt][yt]:
318                         continue
319                     else:
320                         grammar_is_LL = False
321                         mat[xnt][yt] = mat[xnt][yt] \
322                                     + f",{lhs}->{' '.join(y)}"
```

Figure 38 Overview of create parse table process in createParseTable() method

- For the case when there is epsilon ϵ in the first(y). We add additional terminals to res. We check if there is only one terminal, epsilon ϵ, in the first(y), so we take the follow(lhs) as columns (terminals), as shown from Line 291 to Line 299. Whereas when there are another terminals in first(y), we take the first(y-ϵ) U follow(lhs),as columns (terminals), as shown from Line 301 to Line 303.

```
290             if '#' in res:
291                 if type(res) == str:
292                     firstFollow = []
293                     fol_op = follows[lhs]
294                     if fol_op is str:
295                         firstFollow.append(fol_op)
296                     else:
297                         for u in fol_op:
298                             firstFollow.append(u)
299                     res = firstFollow
300                 else:
301                     res.remove('#')
302                     res = list(res) +\
303                         list(follows[lhs])
```

Figure 39 Handling the case of epsilon in createParseTable() method

2-Convert matrix to string.

- It is simply step, which we only need to loop through the matrix and convert it to string.

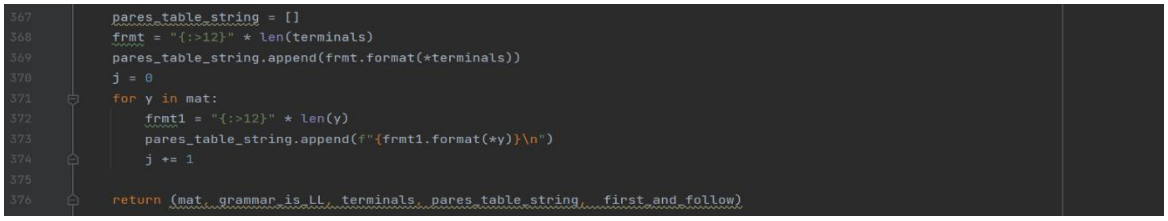


Figure 40 Convert matrix to string in createParseTable() method

2.2.2.4 Build stack to check input validity: validateStringUsingBuffer()

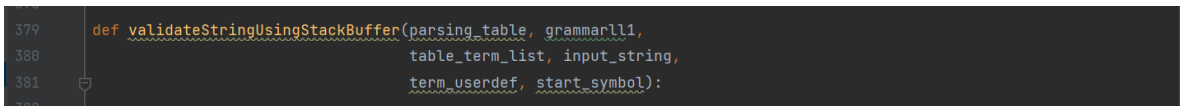


Figure 41 Parameters of validateStringUsingBuffer() method

The validateStringUsingBuffer() method take six parameters: parsing_table, grammarll1, table_term_list, input_string, term_userdef, start_symbol. Which help to parse the input and full the stack.

- Firstly, we establish the initial settings before parsing the input. From Line 389 to Line 400, we define a stack and store the start symbol with $. So, we read the stack from left to right, $ represent the end of the stack. Then we create a buffer to store the input string with $. We reverse the input so we can read it from right to left and the $ represent the end of the input string. In Line 400, we define a string of the stack so it can be returned latter to the GUI.

Figure 42 Initial settings for validateStringUsingBuffer() method

- A while loop start fulling the stack until either we reach to the accepted state or we find an error. It consists of three cases. Form Line 404 to 409 handle the case of accepted action, where the stack and buffer both contain the $. And Line 410 to Line 433 handle the case when there is a non-terminal in the stack, where a predicted action or error will occur. Finally, Line 434 to Line 448 handle the case when there are terminals in both stack and buffer, whether to be a match action or error.

Figure 43 Overview of build stack process in validateStringUsingBuffer() method

- For the case when the stack and buffer both contain the $. We reach to the accepted action. We print the action and ending the loop by return the result and stack string to the GUI.

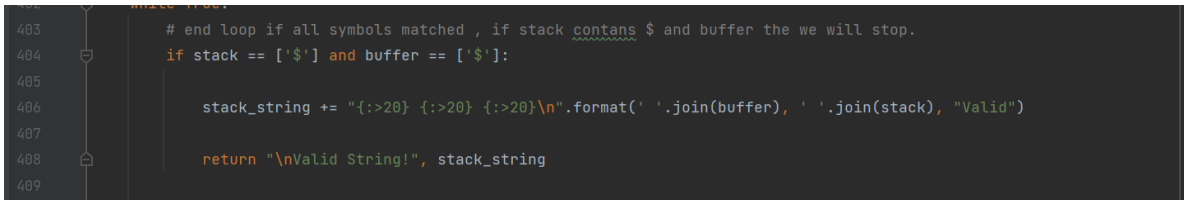


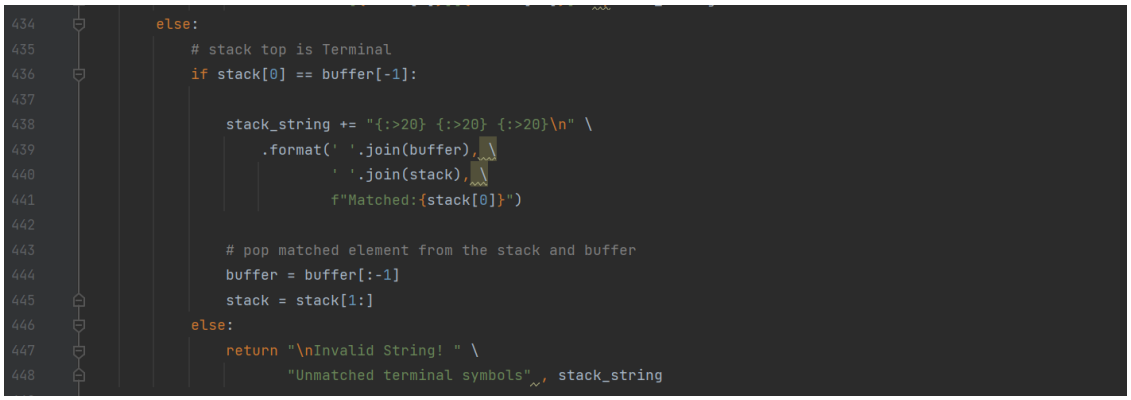
Figure 44 Handling the firsr case in validateStringUsingBuffer() method

- For the case when there is a non-terminal in the stack, we take the non-terminal in the stack and the terminal in the buffer to see If there is a rule exist in the table at row=non-terminal, columns= terminal. If yes, we reach to the predict action. So, we pop the non-terminal from the stack, add the RHS of the rule in the stack without epsilon ϵ, and print the action.

```
elif stack[0] not in term_userdef:
    # take font of buffer (y) and top of stack (x) indexes
    x = list(diction.keys()).index(stack[0])
    y = table_term_list.index(buffer[-1])

    if parsing_table[x][y] != '':
        # format table entry received
        entry = parsing_table[x][y]

        # prepare content of stack to display it on GUI
        stack_string += "{:>20} {:>20} {:>25}\n". \
            format(' '.join(buffer), \
                   ' '.join(stack), \
                   f"T[{stack[0]}][{buffer[-1]}] = {entry}")

        lhs_rhs = entry.split("->")
        lhs_rhs[1] = lhs_rhs[1].replace('#', '').strip()
        entryrhs = lhs_rhs[1].split()
        stack = entryrhs + stack[1:]
    else:
        return f"\nInvalid String! No rule at " \
               f"Table[{stack[0]}][{buffer[-1]}]." stack_string
```

Figure 45 Handling the second case in validateStringUsingBuffer() method

- For the case when there are terminals in both stack and buffer. We check whether the terminals are matched or not. If yes, we reach to match action and we pop the terminal from the stack and buffer. Whereas when they are not match, it indicates an error, we stop the loop and return a message.



```
else:
    # stack top is Terminal
    if stack[0] == buffer[-1]:

        stack_string += "{:>20} {:>20} {:>20}\n" \
            .format(' '.join(buffer), \
                    ' '.join(stack), \
                    f"Matched:{stack[0]}")

        # pop matched element from the stack and buffer
        buffer = buffer[:-1]
        stack = stack[1:]
    else:
        return "\nInvalid String! " \
               "Unmatched terminal symbols", stack_string
```

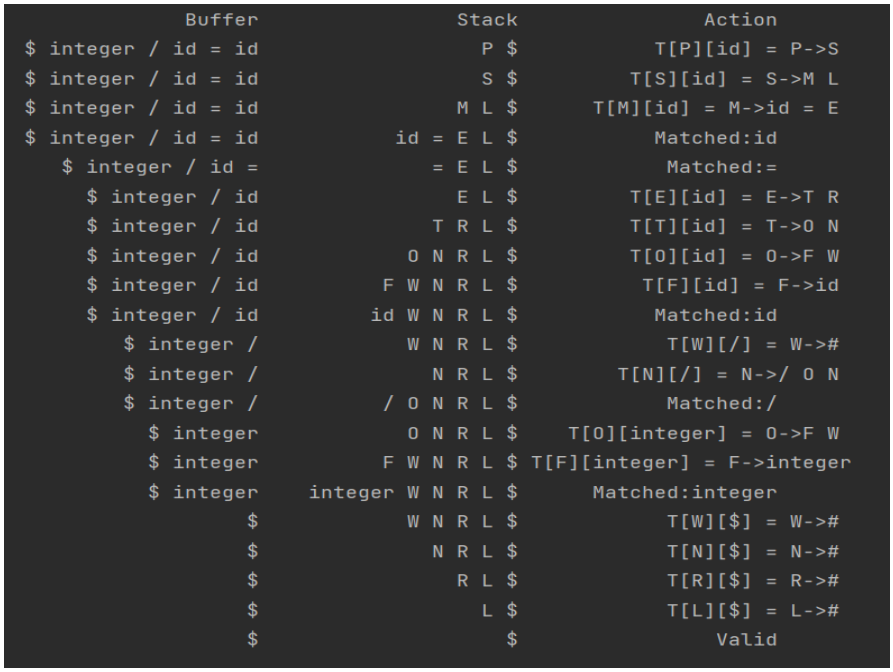
Figure 46 Handling the third case in validateStringUsingBuffer() method

2.2.2.5 Test the code

Correct Strings	Incorrect Strings
id = id	id
id = integer	integer
id = @ integer	id = (id) (id)
id = ! integer	id = integer id
id = (integer * integer) + integer	
id = id / integer	
id = (id) - (id)	

Table 5 Samples to test the code

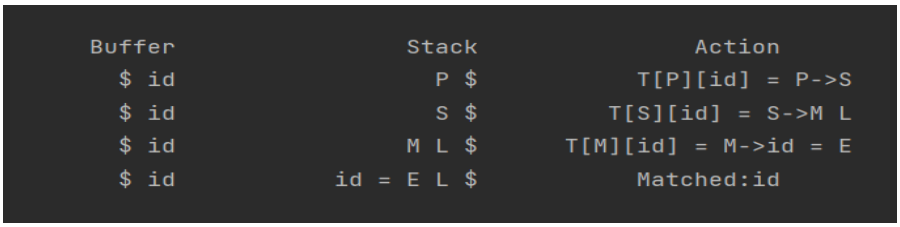
- STACK implementation for “id = id / integer” as correct string.



Buffer	Stack	Action
$ integer / id = id	P $	T[P][id] = P->S
$ integer / id = id	S $	T[S][id] = S->M L
$ integer / id = id	M L $	T[M][id] = M->id = E
$ integer / id = id	id = E L $	Matched:id
$ integer / id =	= E L $	Matched:=
$ integer / id	E L $	T[E][id] = E->T R
$ integer / id	T R L $	T[T][id] = T->O N
$ integer / id	O N R L $	T[O][id] = O->F W
$ integer / id	F W N R L $	T[F][id] = F->id
$ integer / id	id W N R L $	Matched:id
$ integer /	W N R L $	T[W][/] = W->#
$ integer /	N R L $	T[N][/] = N->/ O N
$ integer /	/ O N R L $	Matched:/
$ integer	O N R L $	T[O][integer] = O->F W
$ integer	F W N R L $	T[F][integer] = F->integer
$ integer	integer W N R L $	Matched:integer
$	W N R L $	T[W][$] = W->#
$	N R L $	T[N][$] = N->#
$	R L $	T[R][$] = R->#
$	L $	T[L][$] = L->#
$	$	Valid

Figure 47 Test the code with correct input string

- STACK implementation for “id” as incorrect string.



Buffer	Stack	Action
$ id	P $	T[P][id] = P->S
$ id	S $	T[S][id] = S->M L
$ id	M L $	T[M][id] = M->id = E
$ id	id = E L $	Matched:id

Figure 48 Test the code with incorrect input string

3. GRAGHICAL USER INTERDACE (GUI)

3.1 Part one

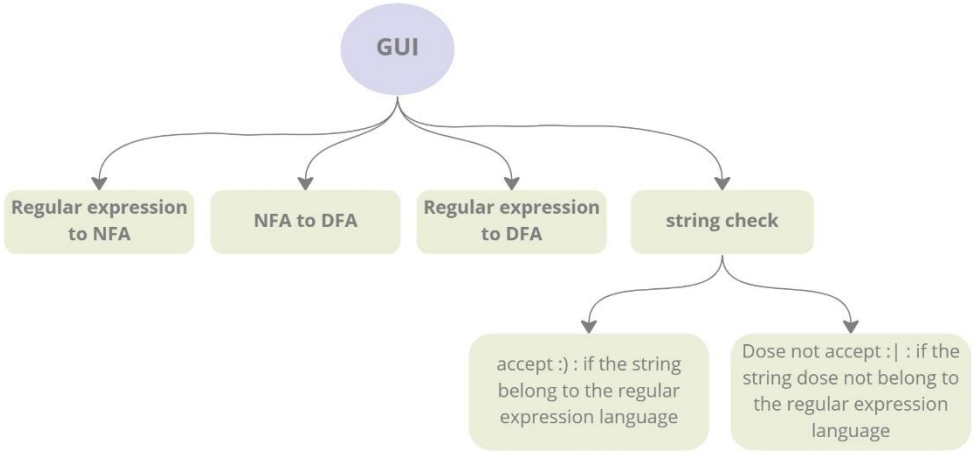


Figure 49. GUI cases

1- Regular expression to NFA

The input is regular expression as shown in Figure 50:

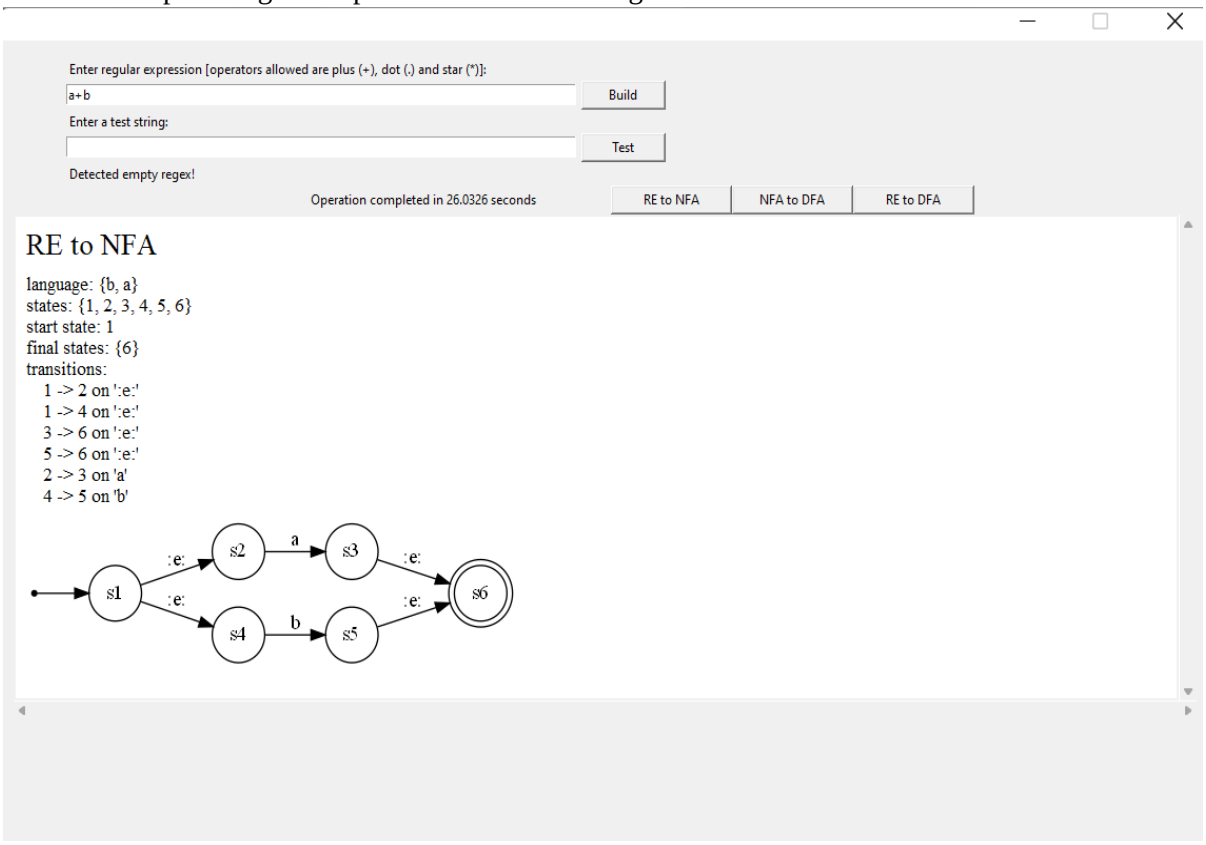


Figure 50. case: regular expression to NFA

2- NFA to DFA

In this case the previous NFA graph converted into DFA as shown in Figure 51:

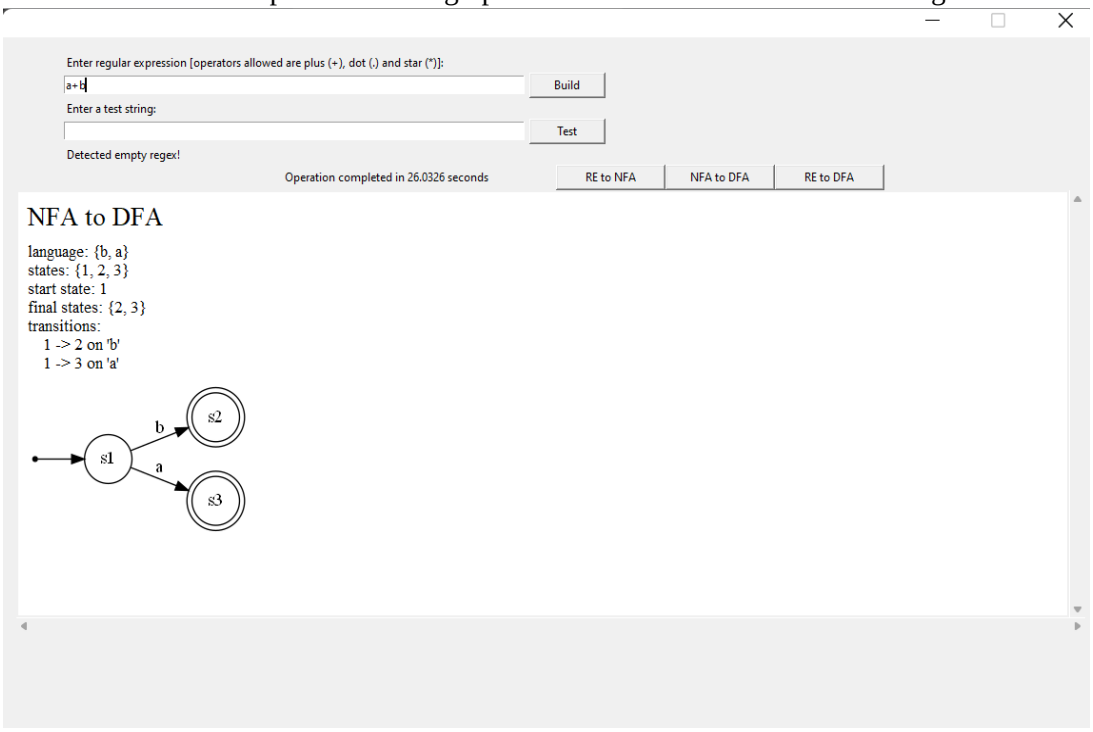


Figure 51. case: NFA to DFA

3- Regular expression to DFA

The input is regular expression as shown in Figure 52. in addition, in this case the syntax tree is generated as shown in Figure 53.

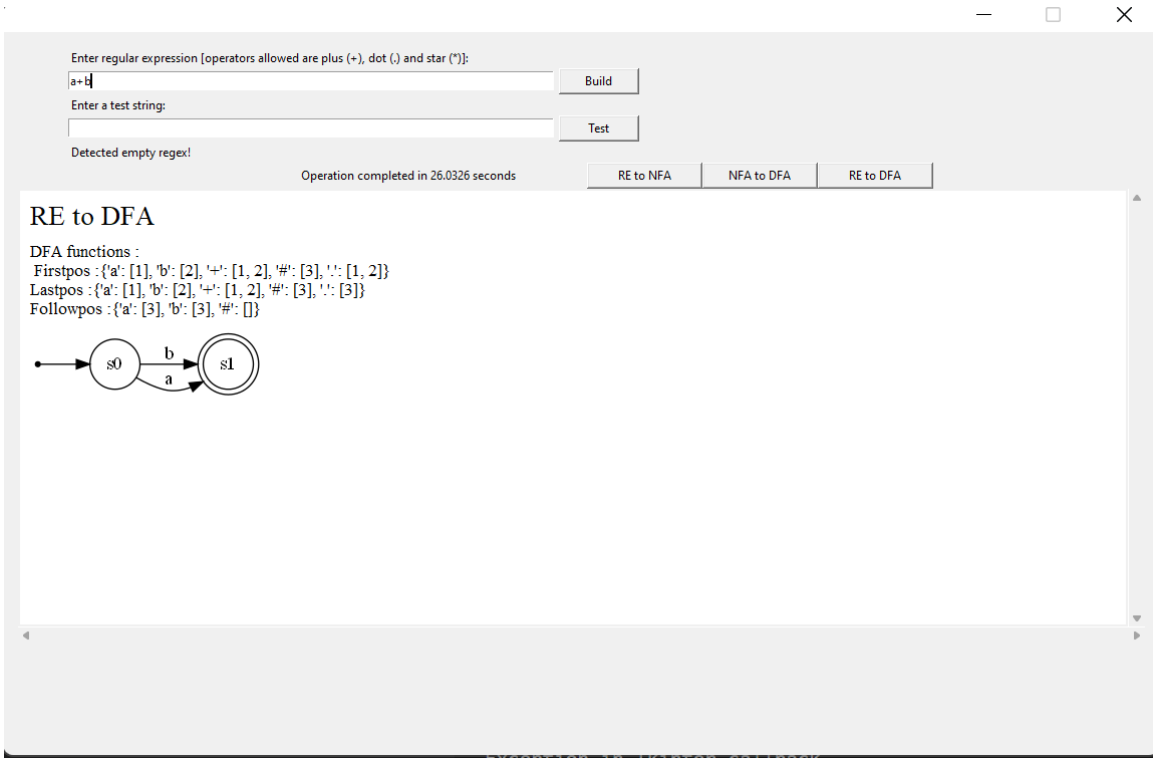


Figure 52. case: regular expression to DFA

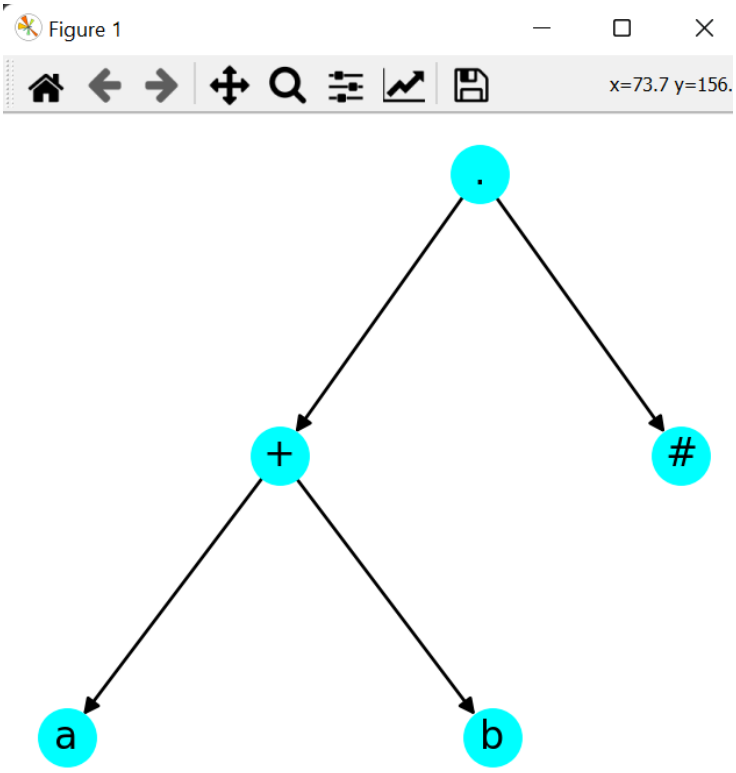


Figure 53. the GUI of syntax tree

4- String check

In this case the input is string there two cases first accept case as shown in Figure 54 and does not accept case as shown in Figure 55.

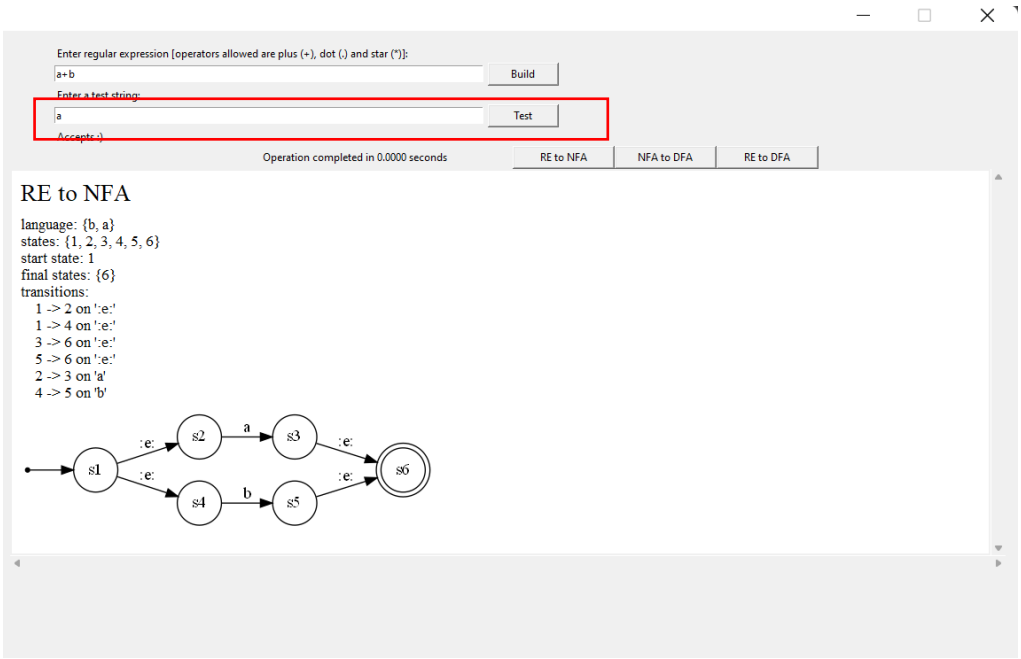


Figure54 . case accept string

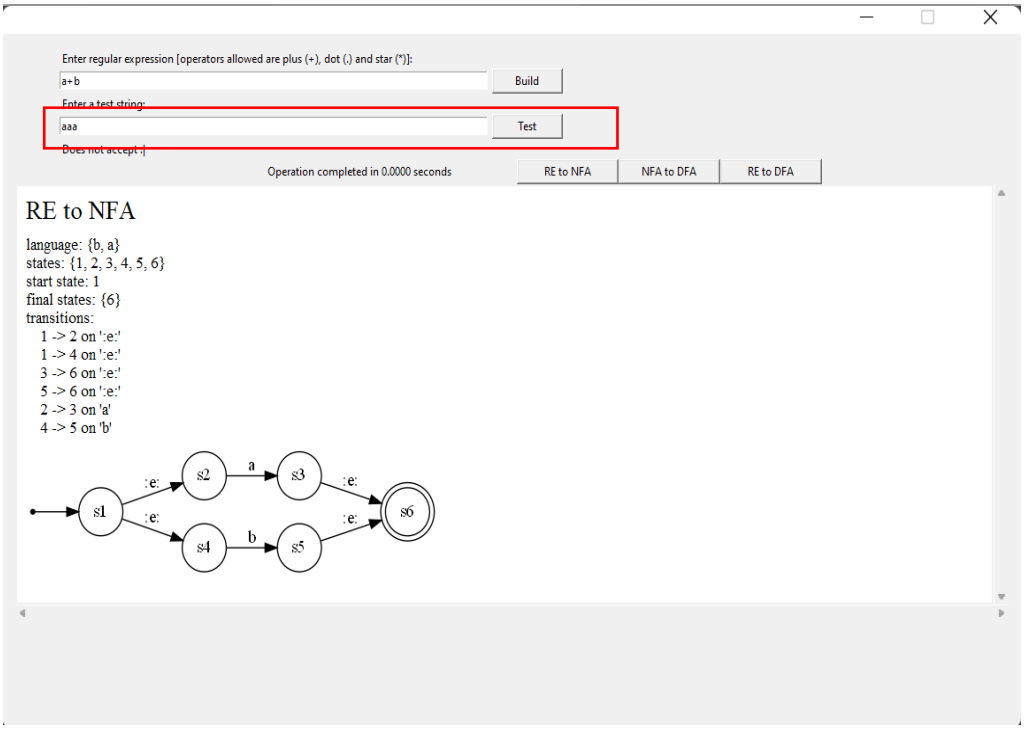


Figure 55.case : does not accept

3.2 Part two

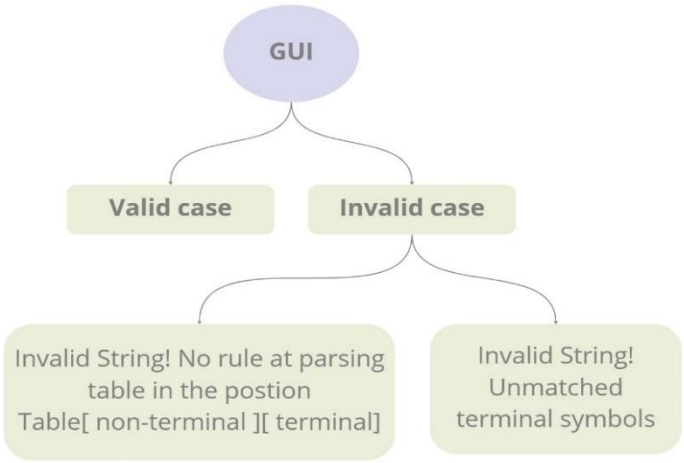


Figure 56 . GUI cases

There are two cases in our program, valid and invalid cases:

1- Valid case

The input expression matches our rules, as shown in Figure 58.

- No rule at the parsing table in position Table [Non-terminal] [Terminal], as shown in Figure 59.
- Unmatched terminal symbols. The element at the top of the stack is unmatched to the element at the top of buffer, as shown in Figure 60.

The continent of the main window is:

- (1) Rules after modifications.
- (2) Rules before modifications.
- (3) Input filed to insert expression.
- (4) First and follow table.
- (5) The content of the stack.
- (6) parsing table.

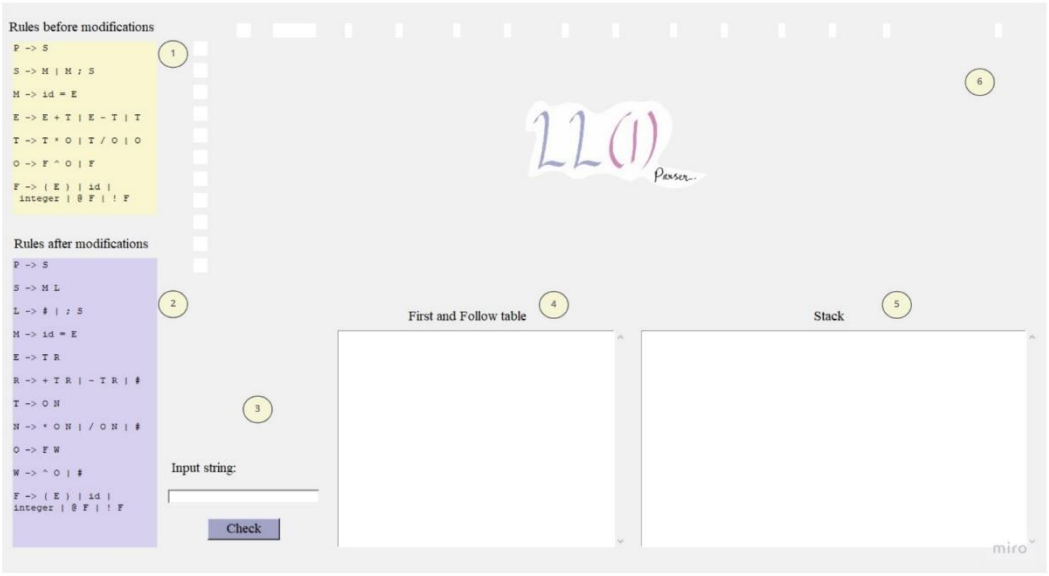


Figure 57 . Main window

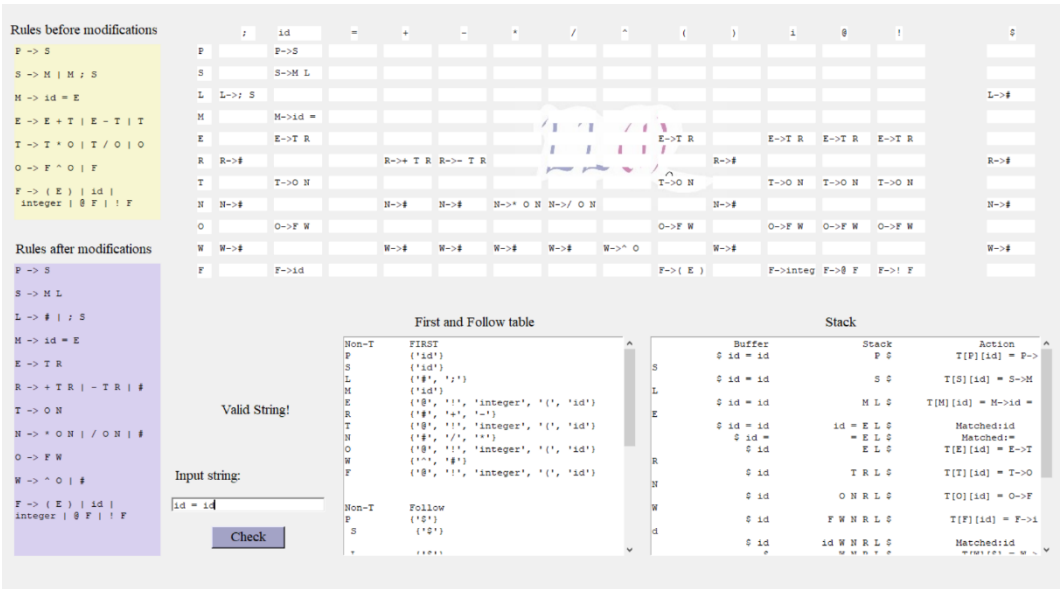


Figure 58 . valid case

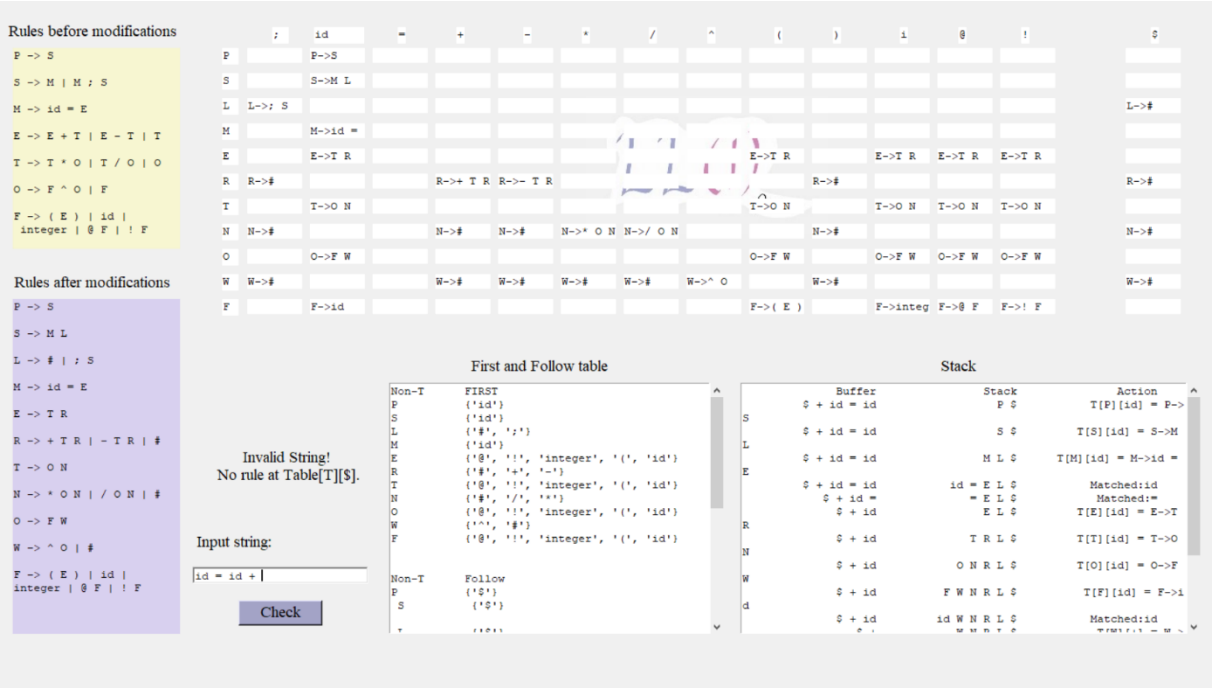


Figure59 . Invalid case no rule at the parsing table.

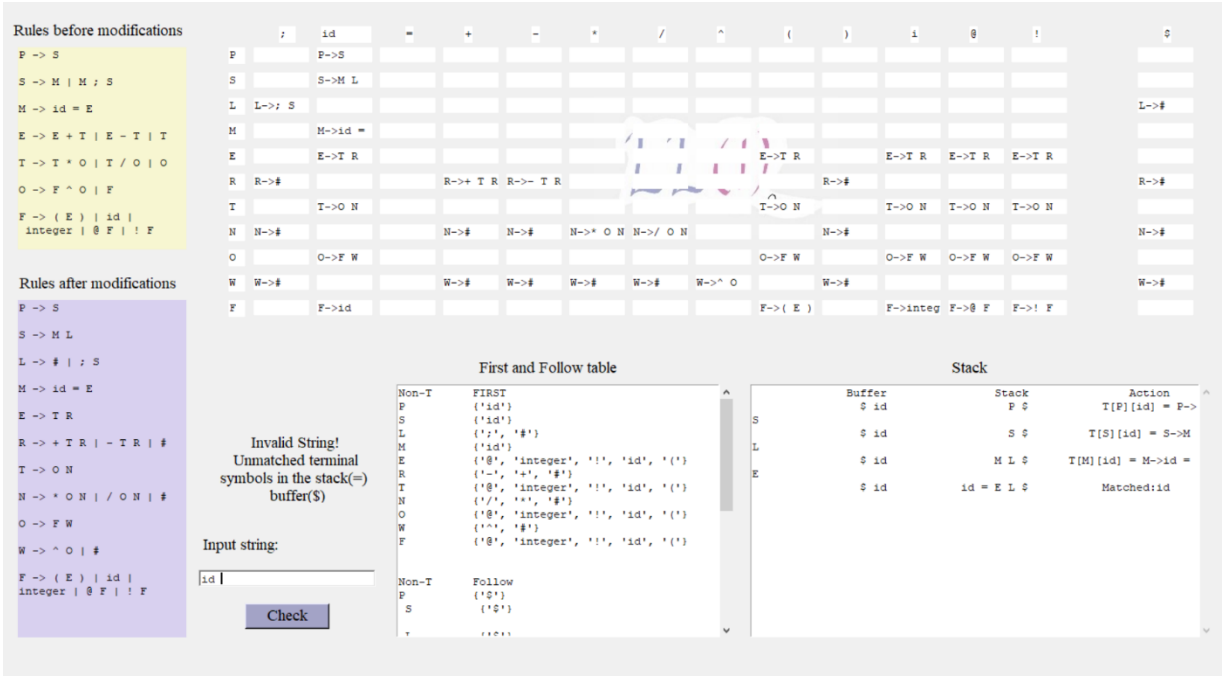


Figure60 . Invalid case unmatched terminal symbols

4. SOURCE CODE

4.1 Part one

```python
from os import popen
import time

class Automata:
    """class to represent an Automata"""
    #constructer for init
    def __init__(self, language = set(['0', '1'])):
        self.states = set() #init state by set
        self.startstate = None
        self.finalstates = []
        self.transitions = dict()
        self.language = language #insert language

    #method for return epsilon
    @staticmethod
    def epsilon():
        return ":e:"

    # method to establish start state
    def setstartstate(self, state):
        self.startstate = state
        # add to state set
        self.states.add(state)


    # method to insert final states
    def addfinalstates(self, state):
        #returns True if the specified state is of the specified type, otherwise False
        if isinstance(state, int):
            # add state to array state
            state = [state]
        # to add every final state in array to self.finalstates
        for s in state:
            if s not in self.finalstates:
                # add final to array
                self.finalstates.append(s)
    # method to add transion of automata
    def addtransition(self, fromstate, tostate, inp):
        #returns True if the specified inp is of the specified type, otherwise False
        if isinstance(inp, str):
            # convert into set
            inp = set([inp])
        # add from state into self.states
        self.states.add(fromstate)
        # add to state into self.states
        self.states.add(tostate)
        # check if from state in transion dict
        if fromstate in self.transitions:
            #check if to state exist in transion dist (ex: a in transion[b])
            if tostate in self.transitions[fromstate]:
                #add new transion into dict with previous transion
                self.transitions[fromstate][tostate] = self.transitions[fromstate][tostate].union(inp)
            else:
                #add new transion into dict
                self.transitions[fromstate][tostate] = inp
        else:
            # if from state not in dict
            self.transitions[fromstate] = {tostate : inp}
     # method for adding transion to dict
    def addtransition_dict(self, transitions):
        # for every (transitions[fromstate][tostate]) add inp
        for fromstate, tostates in transitions.items():
            for state in tostates:
                self.addtransition(fromstate, state, tostates[state])
    # method to get transion of inp
    def gettransitions(self, state, key):
        #returns True if the specified state is of the specified type, otherwise False
        if isinstance(state, int):
            state = [state]
        trstates = set() #init set
        for st in state: # to search of transion
            if st in self.transitions:
                for tns in self.transitions[st]:
                    if key in self.transitions[st][tns]:
                        trstates.add(tns)
        return trstates  #return transion

    def getEClose(self, findstate):
        allstates = set() # init set
        states = set([findstate])
        while len(states)!= 0:
            state = states.pop()
            allstates.add(state)
            if state in self.transitions:
                for tns in self.transitions[state]:
                    if Automata.epsilon() in self.transitions[state][tns] and tns not in allstates:
                        states.add(tns)
        return allstates

```

```python
    # for printing the finite automata content
    def display(self):
        print ("states:", self.states) # print states
        print ("start state: ", self.startstate) # print start state
        print ("final states:", self.finalstates) # print final states or accept
        print ("transitions:") # print transions or move
        for fromstate, tostates in self.transitions.items():
            for state in tostates:
                for char in tostates[state]:
                    print ("  ",fromstate, "->", state, "on '"+char+"'",)
            print
    # to get the finite automata content in text manner for GUI
    def getPrintText(self):
        text = "language: {" + ", ".join(self.language) + "}\n" #  language
        text += "states: {" + ", ".join(map(str,self.states)) + "}\n" # states
        text += "start state: " + str(self.startstate) + "\n" # start state
        text += "final states: {" + ", ".join(map(str,self.finalstates)) + "}\n" # final states or accept
        text += "transitions:\n" # transions or move
        linecount = 5
        for fromstate, tostates in self.transitions.items():
            for state in tostates:
                for char in tostates[state]:
                    text += "     " + str(fromstate) + " -> " + str(state) + " on '" + char + "'\n"
                    linecount +=1
        return [text, linecount]
    # method to get new number state for construction
    def newBuildFromNumber(self, startnum):
        translations = {}
        for i in list(self.states):
            translations[i] = startnum
            startnum += 1
        rebuild = Automata(self.language)
        rebuild.setstartstate(translations[self.startstate])
        rebuild.addfinalstates(translations[self.finalstates[0]])
        for fromstate, tostates in self.transitions.items():
            for state in tostates:
                rebuild.addtransition(translations[fromstate], translations[state], tostates[state])
        return [rebuild, startnum]

    def newBuildFromEquivalentStates(self, equivalent, pos):
        rebuild = Automata(self.language)
        for fromstate, tostates in self.transitions.items():
            for state in tostates:
                rebuild.addtransition(pos[fromstate], pos[state], tostates[state])
        rebuild.setstartstate(pos[self.startstate])
        for s in self.finalstates:
            rebuild.addfinalstates(pos[s])
        return rebuild
    # DOT file represents a graph which consists of nodes and edges
    # Method for get dot file for the graph
    def getDotFile(self):
        dotFile = "digraph DFA {\nrankdir=LR\n"
        if len(self.states) != 0:
            dotFile += "root=s1\nstart [shape=point]\nstart->s%d\n" % self.startstate
            for state in self.states:
                if state in self.finalstates:
                    dotFile += "s%d [shape=doublecircle]\n" % state
                else:
                    dotFile += "s%d [shape=circle]\n" % state
            for fromstate, tostates in self.transitions.items():
                for state in tostates:
                    for char in tostates[state]:
                        dotFile += 's%d->s%d [label="%s"]\n' % (fromstate, state, char)
        dotFile += "}"
        return dotFile

class BuildAutomata:
    """class for building nfa basic structures"""
    # method for construct basic automata ex: constant construction
    @staticmethod
    def basicstruct(inp):
        state1 = 1
        state2 = 2
        basic = Automata()
        basic.setstartstate(state1)
        basic.addfinalstates(state2)
        basic.addtransition(1, 2, inp)
        return basic
    # method for  construction of  or operator (+/|)
    @staticmethod
    def plusstruct(a, b):
        [a, m1] = a.newBuildFromNumber(2)  # to get new number state
        [b, m2] = b.newBuildFromNumber(m1) # to get new number state
        state1 = 1
        state2 = m2
        plus = Automata()
        plus.setstartstate(state1)
        plus.addfinalstates(state2)
        plus.addtransition(plus.startstate, a.startstate, Automata.epsilon())
        plus.addtransition(plus.startstate, b.startstate, Automata.epsilon())
        plus.addtransition(a.finalstates[0], plus.finalstates[0], Automata.epsilon())
        plus.addtransition(b.finalstates[0], plus.finalstates[0], Automata.epsilon())
        plus.addtransition_dict(a.transitions)
        plus.addtransition_dict(b.transitions)
        return plus
```

```
    # method for  construction of  dot operator (.) concatenate
    @staticmethod
    def dotstruct(a, b):
        [a, m1] = a.newBuildFromNumber(1)
        [b, m2] = b.newBuildFromNumber(m1)
        state1 = 1
        state2 = m2-1
        dot = Automata()
        dot.setstartstate(state1)
        dot.addfinalstates(state2)
        dot.addtransition(a.finalstates[0], b.startstate, Automata.epsilon())
        dot.addtransition_dict(a.transitions)
        dot.addtransition_dict(b.transitions)
        return dot

    # method for  construction of  star  operator (*)
    @staticmethod
    def starstruct(a):
        [a, m1] = a.newBuildFromNumber(2)
        state1 = 1
        state2 = m1
        star = Automata()
        star.setstartstate(state1)
        star.addfinalstates(state2)
        star.addtransition(star.startstate, a.startstate, Automata.epsilon())
        star.addtransition(star.startstate, star.finalstates[0], Automata.epsilon())
        star.addtransition(a.finalstates[0], star.finalstates[0], Automata.epsilon())
        star.addtransition(a.finalstates[0], a.startstate, Automata.epsilon())
        star.addtransition_dict(a.transitions)
        return star


class DFAfromNFA:
    """class for building dfa from nfa and minimise it"""
    # constructor
    def __init__(self, nfa):
        self.buildDFA(nfa)
        self.minimise()

    #method for get DFA object
    def getDFA(self):
        return self.dfa


    def getMinimisedDFA(self):
        return self.minDFA

    #method for get DFA content
    def displayDFA(self):
        self.dfa.display()

    def displayMinimisedDFA(self):
        self.minDFA.display()

    # method to build DFA from NFA
    def buildDFA(self, nfa):
        allstates = dict() # init dict for states
        eclose = dict() #init dict for closure
        count = 1
        state1 = nfa.getEClose(nfa.startstate) # first step : find e-closure of inital state
        eclose[nfa.startstate] = state1 # let's e-closure of inital state is state 1
        dfa = Automata(nfa.language) # get posible input
        dfa.setstartstate(count) # init or set start state
        states = [[state1, count]] # insert into list
        allstates[count] = state1 # insert int dict of all state in grapgh ex:{1, {0,1,3,4}}
        count +=  1 # for next state
        while len(states) != 0:
            [state, fromindex] = states.pop()
            for char in dfa.language:
                # to find transion
                trstates = nfa.gettransitions(state, char)
                # loop for check move with e to get next state set
                for s in list(trstates)[:]:
                    if s not in eclose:
                        eclose[s] = nfa.getEClose(s)
                    trstates = trstates.union(eclose[s])
                if len(trstates) != 0:
                    if trstates not in allstates.values():
                        states.append([trstates, count]) # insert into list
                        allstates[count] = trstates
                        toindex = count
                        count +=  1 #for next state
                    else:
                        toindex = [k for k, v in allstates.items() if v  ==  trstates][0]
                    dfa.addtransition(fromindex, toindex, char) # add transion of DFA
        for value, state in allstates.items():
            if nfa.finalstates[0] in state: # check if it is final state
                dfa.addfinalstates(value) # add final state of DFA
        self.dfa = dfa
    # method for check if string is accepted or not
    def acceptsString(self, string):
        currentstate = self.dfa.startstate  # start from start state
        for ch in string:
            if ch==":e:": # if it is epsilon
                continue
            st = list(self.dfa.gettransitions(currentstate, ch)) # else move depend on inp
            if len(st) == 0: # not belong to set of strings
                return False
            currentstate = st[0] # continue
        if currentstate in self.dfa.finalstates: # if its reach to accept state or final
            return True # belong to set of string
        return False # in case dose not reach to accept state or final
```

```
def minimise(self):
        states = list(self.dfa.states)
        n = len(states)
        unchecked = dict()
        count = 1
        distinguished = []
        equivalent = dict(zip(range(len(states)), [{s} for s in states]))
        pos = dict(zip(states,range(len(states))))
        for i in range(n-1):
            for j in range(i+1, n):
                if not ([states[i], states[j]] in distinguished or [states[j], states[i]] in distinguished):
                    eq = 1
                    toappend = []
                    for char in self.dfa.language:
                        s1 = self.dfa.gettransitions(states[i], char)
                        s2 = self.dfa.gettransitions(states[j], char)
                        if len(s1) != len(s2):
                            eq = 0
                            break
                        if len(s1) > 1:
                            raise BaseException("Multiple transitions detected in DFA")
                        elif len(s1) == 0:
                            continue
                        s1 = s1.pop()
                        s2 = s2.pop()
                        if s1 != s2:
                            if [s1, s2] in distinguished or [s2, s1] in distinguished:
                                eq = 0
                                break
                            else:
                                toappend.append([s1, s2, char])
                                eq = -1
                    if eq == 0:
                        distinguished.append([states[i], states[j]])
                    elif eq == -1:
                        s = [states[i], states[j]]
                        s.extend(toappend)
                        unchecked[count] = s
                        count += 1
                    else:
                        p1 = pos[states[i]]
                        p2 = pos[states[j]]
                        if p1 != p2:
                            st = equivalent.pop(p2)
                            for s in st:
                                pos[s] = p1
                            equivalent[p1] = equivalent[p1].union(st)
        newFound = True
        while newFound and len(unchecked) > 0:
            newFound = False
            toremove = set()
            for p, pair in unchecked.items():
                for tr in pair[2:]:
                    if [tr[0], tr[1]] in distinguished or [tr[1], tr[0]] in distinguished:
                        unchecked.pop(p)
                        distinguished.append([pair[0], pair[1]])
                        newFound = True
                        break
        for pair in unchecked.values():
            p1 = pos[pair[0]]
            p2 = pos[pair[1]]
            if p1 != p2:
                st = equivalent.pop(p2)
                for s in st:
                    pos[s] = p1
                equivalent[p1] = equivalent[p1].union(st)
        if len(equivalent) == len(states):
            self.minDFA = self.dfa
        else:
            self.minDFA = self.dfa.newBuildFromEquivalentStates(equivalent, pos)


class NFAfromRegex:
    """class for building nfa from regular expressions"""
    # constructor
    def __init__(self, regex):
        self.star = '*'
        self.plus = '+'
        self.dot = '.'
        self.openingBracket = '('
        self.closingBracket = ')'
        self.operators = [self.plus, self.dot]
        self.regex = regex
        self.alphabet = [chr(i) for i in range(65,91)] # from A to Z
        self.alphabet.extend([chr(i) for i in range(97,123)]) # from a to z
        self.alphabet.extend([chr(i) for i in range(48,58)]) #symbol
        self.buildNFA()

    #method for get NFA object
    def getNFA(self):
        return self.nfa

    #method for get NFA content
    def displayNFA(self):
        self.nfa.display()
```

```python
# method to build NFA
    def buildNFA(self):
        language = set() # init set
        self.stack = [] # init stack
        self.automata = [] # init list
        previous = "::e::" #epsilon
        # for charcter in Ruglar expression
        for char in self.regex:
            if char in self.alphabet: #if it is langauge
                language.add(char) # add it
                # if it is not epslion add dot to stack
                if previous != self.dot and (previous in self.alphabet or previous in [self.closingBracket,self.star]):
                    self.addOperatorToStack(self.dot)
                # else build basic
                self.automata.append(BuildAutomata.basicstruct(char))
            # else if is (
            elif char  ==  self.openingBracket:
                # if it is not epslion add dot to stack
                if previous != self.dot and (previous in self.alphabet or previous in [self.closingBracket,self.star]):
                    self.addOperatorToStack(self.dot)
                # else add to stack
                self.stack.append(char)
            # else if is )
            elif char  ==  self.closingBracket:
                # if it is + or . (plus or dot)
                if previous in self.operators:
                    raise BaseException("Error processing '%s' after '%s'" % (char, previous))
                while(1):
                    if len(self.stack) == 0: # error
                        raise BaseException("Error processing '%s'. Empty stack" % char)
                    # pop of stack
                    o = self.stack.pop()
                    if o == self.openingBracket:
                        break
                    elif o in self.operators: # plus or dot
                        self.processOperator(o)
            # else if is * star
            elif char == self.star:
                if previous in self.operators or previous  == self.openingBracket or previous == self.star:
                    raise BaseException("Error processing '%s' after '%s'" % (char, previous))
                self.processOperator(char)
            # else if is + or
            elif char in self.operators:
                if previous in self.operators or previous  == self.openingBracket:
                    raise BaseException("Error processing '%s' after '%s'" % (char, previous))
                else:
                    self.addOperatorToStack(char) #add to stack
            # else any other case error massage
            else:
                raise BaseException("Symbol '%s' is not allowed" % char)
            previous = char # update previous by char
        # pop for all operator in stake
        while len(self.stack) != 0:
            op = self.stack.pop()
            self.processOperator(op)
        # if more than one NFA in the end must be one
        if len(self.automata) > 1:
            print (self.automata)
            raise BaseException("Regex could not be parsed successfully")
        self.nfa = self.automata.pop()
        self.nfa.language = language # put language

    # method to add char or dot into stack for construction
    def addOperatorToStack(self, char):
        while(1):
            if len(self.stack) == 0:
                break
            top = self.stack[len(self.stack)-1]
            if top == self.openingBracket:
                break
            # check it is char or dor to pop
            if top == char or top == self.dot:
                op = self.stack.pop()
                self.processOperator(op)
            else:
                break
        self.stack.append(char) # add to stack

    #method for build operator
    def processOperator(self, operator):
        if len(self.automata) == 0:
            raise BaseException("Error processing operator '%s'. Stack is empty" % operator)
        if operator == self.star:
            a = self.automata.pop()
            self.automata.append(BuildAutomata.starstruct(a)) # build star structure
        elif operator in self.operators: # if it is plus or dot
            if len(self.automata) < 2: # must two operaend for + and . operator
                raise BaseException("Error processing operator '%s'. Inadequate operands" % operator)
            a = self.automata.pop() # pop a
            b = self.automata.pop() # pop b
            if operator == self.plus:
                self.automata.append(BuildAutomata.plusstruct(b,a)) # build plus structure
            elif operator == self.dot:
                self.automata.append(BuildAutomata.dotstruct(b,a)) # build dot structure

# method for drawing graph
def drawGraph(automata, file = ""):
    """From https://github.com/max99x/automata-editor/blob/master/util.py"""
    f = popen(r"dot -Tpng -o graph%s.png" % file, 'w')
    try:
        f.write(automata.getDotFile()) # get graph and write in f file
    except:
        raise BaseException("Error creating graph") # error
    finally:
        f.close() # close f file

#method for install image of graph
def isInstalled(program):
    """From http://stackoverflow.com/questions/377017/test-if-executable-exists-in-python"""
    import os
    def is_exe(fpath):
        return os.path.isfile(fpath) and os.access(fpath, os.X_OK)
    fpath, fname = os.path.split(program)
    if fpath:
        if is_exe(program) or is_exe(program+".exe"):
            return True
    else:
        for path in os.environ["PATH"].split(os.pathsep):
            exe_file = os.path.join(path, program)
            if is_exe(exe_file) or is_exe(exe_file+".exe"):
                return True
    return False
```

```python
#ply.lex is an implementation of lex parsing tools for Python.
import ply.lex as lex
# networkx package for the Python used to create, manipulate graph networks used to draw tree
import networkx as nx
import matplotlib.pyplot as plt
# import to draw graph of DFA
from AutomataTheory import *

# Tokens
tokens = (
    'OPEN_P',
    'CLOSED_P',
    'OR',
    'STAR',
    'CONCAT',
    'LETTER',
    'LAMBDA'
)

# Rules that define the tokens
t_OPEN_P = r'\('
t_CLOSED_P = r'\)'
t_OR = r'\+'
t_STAR = r'\*'
t_CONCAT = r'\.'
t_LETTER = r'[a-zA-Z]'
t_LAMBDA = r'\#'


# Error rule
def t_error(t):
    print("Illegal character '%s'" % t.value[0])
    t.lexer.skip(1)

class RegexLexer:
    # A lexer to tokenize the content
    lexer = lex.lex()
    # A container containing the tokens
    parts = []

    # constructer
    def __init__(self, inp):
            self.content = '(' + inp + ')#'
            self.tokenize()


   # method of add tokens into list
    def tokenize(self):
        self.lexer.input(self.content)
        while True:
            token = self.lexer.token()
            if not token: # No more input
                break
            self.parts.append(token)

    # for printing tokens
    def printTokens(self):
        for part in self.parts:
            print(part)

    # method add concantion (dot) step 2 in coversion
    def _addConcats(self):
        possibleConcat = False
        concatMessage = ''
        index = 0

        for part in self.parts:
            if part.type in ['LETTER', 'LAMBDA']: # operand or #
                if possibleConcat is True:
                    concatMessage = concatMessage + '.' # add dot
                else:
                    possibleConcat = True
            elif part.type in ['STAR', 'CLOSED_P']:# * or )
                possibleConcat = True
            elif (part.type == 'OPEN_P') & (self.parts[index - 1].type in ['LETTER', 'STAR']):
                concatMessage = concatMessage + '.'
            else:
                possibleConcat = False
            concatMessage = concatMessage + part.value
            index = index + 1

        self.content = concatMessage # after add dot
        self.parts.clear()
        self.tokenize()
```

```python
    # to write in postfix notation by use reverse polish notation
    #where every operator follows all of its operands
    def writeAsRPN(self, permanent=False):
        if self.content[len(self.content) - 1] == '.':
            print("Already in rpn notation")
            return
        # First we need to have the regex written in reverse polish notation ot postfix
        operator_stack = [] # init list
        last_index = -1 # pointer of stack
        rpn_regex = ''
        for part in self.parts:
            if part.type in ['LETTER', 'STAR', 'LAMBDA']:
                rpn_regex = rpn_regex + part.value # add to RE
                continue
            if part.type == 'OPEN_P':
                operator_stack.append(part) # put in stack
                last_index += 1
                continue
            if part.type == 'CLOSED_P':
                while operator_stack[last_index].type != 'OPEN_P': # to pop until (
                    rpn_regex += operator_stack[last_index].value # add to RE
                    operator_stack.pop() # pop from stack
                    last_index -= 1
                # Doing one more pop to  extract the open parenthesis
                operator_stack.pop()
                last_index -= 1
                continue
            if part.type in ['OR', 'CONCAT']:
                if last_index >= 0:
                    if operator_stack[last_index].type not in ['OPEN_P', 'CLOSED_P']:
                        rpn_regex += operator_stack[last_index].value # add to RE
                        operator_stack.pop()  # pop from stack
                        operator_stack.append(part) # then add to stack
                    else:
                        operator_stack.append(part)  # add to stack
                        last_index += 1
                else:
                    operator_stack.append(part) # add to stack
                    last_index += 1
                continue

        # Clearing the stack of any operators that might have remained
        while len(operator_stack) != 0:
            rpn_regex += operator_stack[last_index].value
            operator_stack.pop() # pop from stack

        if permanent is True:
            self.parts.clear()
            self.content = rpn_regex # after postfix update content
            self.tokenize()

        return rpn_regex

    # method to creating syntax tree
    def makeAST(self):
        self._addConcats()
        print("\nRegex after adding concatenations: " + self.content + '\n')
        self.printTokens()

        print("\nRegex written in reverse polish notation: " +
self.writeAsRPN(permanent=True) + '\n')

        AST = nx.DiGraph() # init tree
        node_stack = []
        last_index = -1

        for part in self.parts:
            if part.type in ['LETTER', 'LAMBDA']: # add operand or #
                AST.add_node(part, val=part.value) # add node
                node_stack.append(part) # add to stak
                last_index += 1
            elif part.type in ['OR', 'CONCAT']: # add + or .
                AST.add_node(part, val=part.value)
                for i in range(0, 2): # two operand
                    AST.add_edge(part, node_stack[last_index]) # add edage
                    node_stack.pop() # pop operand from stack
                    last_index -= 1
                node_stack.append(part) # add inner node
                last_index += 1
            elif part.type == 'STAR':
                AST.add_node(part, val=part.value) # add node
                AST.add_edge(part, node_stack[last_index]) # add edage
                node_stack.pop() # pop from stake
                node_stack.append(part)  # add inner node

        return AST
    # method to print syntax tree
    def printAST(self):
        AST = self.makeAST()
        nx.nx_agraph.write_dot(AST, 'test.dot')
        pos = nx.nx_agraph.graphviz_layout(AST, prog='dot')
        nx.draw(AST, pos, node_color='cyan')
        nx.draw_networkx_labels(AST, pos, nx.get_node_attributes(AST, 'val'))

        plt.savefig('ast.png')
        plt.show()
```

```python
   # class convert RE to DFA
class Converter:
    FirstPos = []
    LastPos = []
    FollowPos = []

    DFA = nx.MultiDiGraph()

    #constructor
    def __init__(self, rpn_tokens: list):

        self.rpn_tokens = rpn_tokens
        self._makeFirstPos()
        self._makeLastPos()
        self._makeFollowPos()

    def _makeFirstPos(self):
        """
        This method is used to compute the first_pos list for each token.

        There are special cases that we need to address: when a branch could generate
both a sequence from a word
        or nothing at all. In this case the "sub-word" that is generated by the sub-tree
could start on both branches.
        Since here we are building only the first-pos collection we are interested in
left side sub-trees only.

        :return: nothing
        """
        index = 0  # Used to copy positions from previous nodes
        letter_no = 1  # Each letter in the regex is associated a number in the order of
appearance
        nullable = []

        for token in self.rpn_tokens:
            if token.type in ['LETTER', 'LAMBDA']: #if it is operand or #
                newpos = [letter_no]
                letter_no += 1
                index += 1
                self.FirstPos.append(newpos)
                if token.type == 'LAMBDA':
                    if token.lexpos == len(self.rpn_tokens) - 2:
                        nullable.append(False)
                    else:
                        nullable.append(True)
                else:
                    nullable.append(False)

            elif token.type == 'OR':
                newpos = self.FirstPos[index - 2].copy() + self.FirstPos[index -
1].copy() # c1 U c2
                self.FirstPos.append(newpos)
                index += 1
                if nullable[index - 3] | nullable[index - 2]:
                    nullable.append(True)
                else:
                    nullable.append(False)

            elif token.type == 'STAR':
                newpos = self.FirstPos[index - 1].copy()
                self.FirstPos.append(newpos)
                nullable.append(True)
                index += 1

            elif token.type == 'CONCAT':

                if self.rpn_tokens[index - 1].type == 'STAR':
                    nb = index - 2
                    while True:
                        if self.rpn_tokens[nb].type in ['CONCAT', 'OR']:
                            nb -= 2
                        else:
                            nb -= 1
                            break
                    newpos = self.FirstPos[nb].copy()
                    self.FirstPos.append(newpos)
                    index += 1

                    if nullable[nb]:
                        newpos = self.FirstPos[index - 2].copy()
                        self.FirstPos[index - 1] += newpos
                    if nullable[nb] & nullable[index - 2]:
                        nullable.append(True)
                    else:
                        nullable.append(False)
                else:
                    newpos = self.FirstPos[index - 2].copy()
                    self.FirstPos.append(newpos)
                    index += 1

                    if nullable[index - 3]:
                        newpos = self.FirstPos[index - 2].copy()
                        self.FirstPos[index - 1] += newpos
                    if nullable[index - 3] & nullable[index - 2]:
                        nullable.append(True)
                    else:
                        nullable.append(False)
```

```
    def _makeLastPos(self):
        index = 0  # Used to copy positions from previous nodes
        letter_no = 1  # Each letter in the regex is associated a number in the order of appearance
        nullable = []

        for token in self.rpn_tokens:
            if token.type in ['LETTER', 'LAMBDA']: #if ot is operand or #
                newpos = [letter_no]
                letter_no += 1
                index += 1
                self.LastPos.append(newpos)
                if token.type == 'LAMBDA': # if it is #
                    if token.lexpos == len(self.rpn_tokens) - 2:
                        nullable.append(False)
                    else:
                        nullable.append(True)
                else:
                    nullable.append(False)

            elif token.type == 'OR':
                newpos = self.LastPos[index - 2].copy() + self.LastPos[index - 1].copy()
                self.LastPos.append(newpos)
                index += 1
                if nullable[index - 3] | nullable[index - 2]:
                    nullable.append(True)
                else:
                    nullable.append(False)

            elif token.type == 'STAR':
                newpos = self.LastPos[index - 1].copy()
                self.LastPos.append(newpos)
                nullable.append(True)
                index += 1

            elif token.type == 'CONCAT':
                newpos = self.LastPos[index - 1].copy()
                self.LastPos.append(newpos)
                index += 1

                if self.rpn_tokens[index - 2].type == 'STAR':
                    nb = index - 3
                    while True:
                        if self.rpn_tokens[nb].type in ['CONCAT', 'OR']:
                            nb -= 2
                        else:
                            nb -= 1
                            break

                    if nullable[index - 2]:
                        newpos = self.LastPos[nb].copy()
                        self.LastPos[index - 1] += newpos

                    if nullable[nb] & nullable[index - 2]:
                        nullable.append(True)
                    else:
                        nullable.append(False)
                else:
                    if nullable[index - 2]:
                        newpos = self.LastPos[index - 3].copy()
                        self.LastPos[index - 1] += newpos
                    if nullable[index - 3] & nullable[index - 2]:
                        nullable.append(True)
                    else:
                        nullable.append(False)

    def _makeFollowPos(self):
        index = 0
        number_nop = 0

        # Counting the number of operand and #
        for token in self.rpn_tokens:
            if token.type in ['LETTER','LAMBDA']:
                number_nop += 1

        self.FollowPos = [[] for _ in range(number_nop)]

        for token in self.rpn_tokens:
            if token.type == 'STAR':
                for pos in self.LastPos[index]:
                    newpos = self.FirstPos[index].copy()
                    self.FollowPos[pos - 1] += newpos
                index += 1
            elif token.type == 'CONCAT':
                if self.rpn_tokens[index - 1].type == 'STAR':
                    nb = index - 2
                    while True:
                        if self.rpn_tokens[nb].type in ['CONCAT', 'OR']:
                            nb -= 2
                        else:
                            nb -= 1
                            break
                    for pos in self.LastPos[nb]:
                        self.FollowPos[pos - 1] += self.FirstPos[index - 1].copy()
                else:
                    for pos in self.LastPos[index - 2]:
                        self.FollowPos[pos - 1] += self.FirstPos[index - 1].copy()
                index += 1
            else:
                index += 1

        # Stripping duplicate values
        for collection in self.FollowPos:
            if collection is not None:
                collection = set(collection)


    # method to get the content of Firstpos,Lastpos, for GUI
    def getPrintText(self):
        text = "DFA functions :\n " # print DFA function
        fres = {}
        for part in self.rpn_tokens:
          for pos in self.FirstPos:
           fres[part.value] = pos
           self.FirstPos.remove(pos)
           break
        text += "Firstpos :" + str(fres) + "\n"
        lres = {}
        for part in self.rpn_tokens:
          for pos in self.LastPos:
           lres[part.value] = pos
           self.LastPos.remove(pos)
           break
        text += "Lastpos :" + str(lres) + "\n"
        ores = {}
        for part in self.rpn_tokens:
         if part.type in ['LETTER','LAMBDA']:
            for pos in self.FollowPos:
                 ores[part.value] = pos
                 self.FollowPos.remove(pos)
                 break
        text += "Followpos :" + str(ores) + "\n"
        linecount = 4
        return [text, linecount]


try:
    from tkinter import * # import tkinter library for GUI
    import tkinter.font as tkFont # import tkFont class for GUI
except ImportError as err: # for error in instalision of library
    print ("error: %s. Tkinter library is required for using the GUI.") % err.message
    sys.exit(1)

# import AutomataTheory for construct RE to NFA and NFA to DFA
from AutomataTheory import *
# import DFAfromRE for construct RE to DFA
from DFAfromRE import *

dotFound = isInstalled("dot") # install image
if dotFound:
    try:
        #import Python Imaging Library for image processing and display graph
        from PIL import Image, ImageTk
    except ImportError as err:
        print ("Notice: %s. The PIL library is required for displaying the graphs.") % err.message
        dotFound = False
else:
    print ("Notice: The GraphViz software is required for displaying the graphs.")

# class of creating GUI
class AutomataGUI:

    def __init__(self, root, dotFound):
        self.root = root
        self.initUI() # call method to init GUI
        self.selectedButton = 0 # for button selection
        self.dotFound = dotFound #for graph
        startRegex = "" # string of reuglar expression
        # regexVar for manage the value of a widget such as an Entry widget or a Label widget
        self.regexVar.set(startRegex)
        # for handle button selection
        self.handleBuildRegexButton()
```

```
    # method creat GUI
    def initUI(self):
        self.root.title("Finite Automata") # title
        # screen setting
        ScreenSizeX = self.root.winfo_screenwidth()
        ScreenSizeY = self.root.winfo_screenheight()
        ScreenRatioX = 0.9
        ScreenRatioY = 1.0
        self.FrameSizeX  = int(ScreenSizeX * ScreenRatioX)
        self.FrameSizeY  = int(ScreenSizeY * ScreenRatioY)
        FramePosX   = (ScreenSizeX - self.FrameSizeX)/2
        FramePosY   = (ScreenSizeY - self.FrameSizeY)/2
        padX = 15
        padY = 15
        # creating main window
        self.root.geometry("%dx%d+%d+%d" %
(self.FrameSizeX,self.FrameSizeY,FramePosX,FramePosY))
        self.root.resizable(width=False, height=False)
        # creatin frame
        parentFrame = Frame(self.root, width = int(self.FrameSizeX - 2*padX), height =
int(self.FrameSizeY - 2*padY))
        parentFrame.grid(padx=padX, pady=padY, stick=E+W+N+S)

        regexFrame = Frame(parentFrame) # frame of regular expression
        # creating label
        enterRegexLabel = Label(regexFrame, text="Enter regular expression [operators
allowed are plus (+), dot (.) and star (*)]:")
        self.regexVar = StringVar()
        self.regexField = Entry(regexFrame, width=80, textvariable=self.regexVar) #
entry regular expression
        buildRegexButton = Button(regexFrame, text="Build", width=10,
command=self.handleBuildRegexButton)
        # setting :
        enterRegexLabel.grid(row=0, column=0, sticky=W)
        self.regexField.grid(row=1, column=0, sticky=W)
        buildRegexButton.grid(row=1, column=1, padx=5)
        # creatin frame
        testStringFrame = Frame(parentFrame)
        testStringLabel = Label(testStringFrame, text="Enter a test string: ") #
creating label
        self.testVar = StringVar()
        self.testStringField = Entry(testStringFrame, width=80,
textvariable=self.testVar) # entry string
        testStringButton = Button(testStringFrame, text="Test", width=10,
command=self.handleTestStringButton)
        # setting :
        testStringLabel.grid(row=0, column=0, sticky=W)
        self.testStringField.grid(row=1, column=0, sticky=W)
        testStringButton.grid(row=1, column=1, padx=5)

        self.statusLabel = Label(parentFrame) # creating label

        buttonGroup = Frame(parentFrame)
        self.timingLabel = Label(buttonGroup, text="Tasks : ", width=50, justify=RIGHT)
        # creating button
        nfaButton = Button(buttonGroup, text="RE to NFA", width=15,
command=self.handlenfaButton)
        dfaButton = Button(buttonGroup, text="NFA to DFA", width=15,
command=self.handledfaButton)
        minDFAButton = Button(buttonGroup, text="RE to DFA", width=15,
command=self.handleminDFAButton)
        # setting :
        self.timingLabel.grid(row=0, column=0, sticky=W)
        nfaButton.grid(row=0, column=1)
        dfaButton.grid(row=0, column=2)
        minDFAButton.grid(row=0, column=3)

        # creat frame of implemention taska
        automataCanvasFrame = Frame(parentFrame, height=100, width=100)
        self.cwidth = int(self.FrameSizeX - (2*padX + 20))
        self.cheight = int(self.FrameSizeY * 0.6)
        self.automataCanvas = Canvas(automataCanvasFrame, bg='#FFFFFF', width=
self.cwidth, height = self.cheight,scrollregion=(0,0,self.cwidth,self.cheight))
        hbar=Scrollbar(automataCanvasFrame,orient=HORIZONTAL)
        hbar.pack(side=BOTTOM,fill=X)
        hbar.config(command=self.automataCanvas.xview)
        vbar=Scrollbar(automataCanvasFrame,orient=VERTICAL)
        vbar.pack(side=RIGHT,fill=Y)
        vbar.config(command=self.automataCanvas.yview)
        self.automataCanvas.config(xscrollcommand=hbar.set, yscrollcommand=vbar.set)
        self.canvasitems = []
        self.automataCanvas.pack()

        # setting :
        regexFrame.grid(row=0, column=0, sticky=W, padx=(50,0))
        testStringFrame.grid(row=1, column=0, sticky=W, padx=(50,0))
        self.statusLabel.grid(row=2, column=0, sticky=W, padx=(50,0))
        buttonGroup.grid(row=3, column=0)
        automataCanvasFrame.grid(row=4, column=0, sticky=E+W+N+S)

    # get action of button to do tasks
    def handleBuildRegexButton(self):
        t = time.time()
        try:
            inp = self.regexVar.get().replace(' ','') # get reuglar expression
            if inp == '':
                self.statusLabel.config(text="Detected empty regex!")
                return
            self.createAutomata(inp) # start creat NFA and NFA to DFA
        except BaseException as e:
            self.statusLabel.config(text="Failure: %s" % e)
        # for time
        self.timingLabel.configure(text="Operation completed in " + "%.4f" %
(time.time() - t) + " seconds")
        # for display graph
        self.displayAutomata()
```

```
    # get action of button to do string check
    def handleTestStringButton(self):
        t = time.time()
        inp = self.testVar.get().replace(' ','') # get string
        if inp == '':
            inp = [':e:']
        if self.dfaObj.acceptsString(inp): # check
            self.statusLabel.config(text="Accepts :)")
        else:
            self.statusLabel.config(text="Does not accept :|")
        self.timingLabel.configure(text="Operation completed in " + "%.4f" %
(time.time() - t) + " seconds")
    # method to set selected button
    def handlenfaButton(self):
        self.selectedButton = 0
        self.displayAutomata()
    # method to set selected button
    def handledfaButton(self):
        self.selectedButton = 1
        self.displayAutomata()
    # method to set selected button
    def handleminDFAButton(self):
        self.selectedButton = 2
        self.displayAutomata()

    def createAutomata(self, inp):
        print ("Regex: ", inp)
        nfaObj = NFAfromRegex(inp) # from reuglar expression to NFA
        self.nfa = nfaObj.getNFA() # get NFA
        self.dfaObj = DFAfromNFA(self.nfa) # from NFA to DFA
        self.dfa = self.dfaObj.getDFA() #get DFA
        self.minDFA = self.dfaObj.getMinimisedDFA() # to get the graph of RE to DFA
        self.DFA = RegexLexer(inp) # RE to DFA
        self.DFA.printAST() # to construct syntax tree
        self.mdfa = Converter(self.DFA.parts) # take RE with syntax tree to find
first,last,follow
        if self.dotFound: # creating graph
            drawGraph(self.dfa, "dfa")
            drawGraph(self.nfa, "nfa")
            drawGraph(self.minDFA, "mdfa")
            dfafile = "graphdfa.png"
            nfafile = "graphnfa.png"
            mindfafile = "graphmdfa.png"
            self.nfaimagefile = Image.open(nfafile)
            self.dfaimagefile = Image.open(dfafile)
            self.mindfaimagefile = Image.open(mindfafile)
            self.nfaimg = ImageTk.PhotoImage(self.nfaimagefile)
            self.dfaimg = ImageTk.PhotoImage(self.dfaimagefile)
            self.mindfaimg = ImageTk.PhotoImage(self.mindfaimagefile)
    # to display int GUI
    def displayAutomata(self):
        for item in self.canvasitems:
            self.automataCanvas.delete(item)
        if self.selectedButton == 0:
            header = "RE to NFA"
            automata = self.nfa
            if self.dotFound:
                image = self.nfaimg
                imagefile = self.nfaimagefile
        elif self.selectedButton == 1:
            header = "NFA to DFA"
            automata = self.dfa
            if self.dotFound:
                image = self.dfaimg
                imagefile = self.dfaimagefile
        elif self.selectedButton == 2:
            header = "RE to DFA"
            automata = self.minDFA
            if self.dotFound:
                image = self.mindfaimg
                imagefile = self.mindfaimagefile
            automata = self.mdfa
        # setting :
        font = tkFont.Font(family="times", size=20)
        (w,h) = (font.measure(header),font.metrics("linespace"))
        headerheight = h + 10
        itd = self.automataCanvas.create_text(10,10,text=header, font=font, anchor=NW)
        self.canvasitems.append(itd)
        [text, linecount] = automata.getPrintText()
        font = tkFont.Font(family="times", size=13)
        (w,h) = (font.measure(text),font.metrics("linespace"))
        textheight = headerheight + linecount * h + 20
        itd = self.automataCanvas.create_text(10, headerheight + 10,text=text,
font=font, anchor=NW)
        self.canvasitems.append(itd)
        if self.dotFound:
            itd = self.automataCanvas.create_image(10, textheight, image=image,
anchor=NW)
            self.canvasitems.append(itd)
            totalwidth = imagefile.size[0] + 10
            totalheight = imagefile.size[1] + textheight + 10
        else:
            totalwidth = self.cwidth + 10
            totalheight = textheight + 10
        if totalheight < self.cheight:
            totalheight = self.cheight
        if totalwidth < self.cwidth:
            totalwidth = self.cwidth
        self.automataCanvas.config(scrollregion=(0,0,totalwidth,totalheight))

# main
def main():
    global dotFound
    root = Tk()
    app = AutomataGUI(root, dotFound)
    root.mainloop()

if __name__ == '__main__':
    main()
```

4.2 Part two

```

# preprocess rules
def preprocessing(rules, diction):
    # loop on each rule to remove unnecessary spaces
    for rule in rules:
        k = rule.split("->")

        # remove spaces from LHS and RHS of the rule
        k[0] = k[0].strip()
        k[1] = k[1].strip()

        # store k[1] which is the RHS of the rule in variable RHS
        rhs = k[1]

        # split RHS and then store it in multirhs list
        multirhs = rhs.split('|')

        """
        remove spaces from each element in the list multirhs (multirhs[i])
        and then split each element if we find a space
        """
        for i in range(len(multirhs)):
            multirhs[i] = multirhs[i].strip()
            multirhs[i] = multirhs[i].split()

        # store the RHS for each rule after preprocessing in dictionary Key: {LHS}, value : {RHS}
        diction[k[0]] = multirhs


# Calculation of first. Note epsilon is denoted by '#' (semi-colon)
def first(rule, non_term_user_def, term_user_def, diction, firsts):
    # This is a (recursion base case) (for terminal or epsilon)
    if len(rule) != 0 and (rule is not None):
        # if rule[0] is terminal
        if rule[0] in term_userdef:
            return rule[0]
        # if rule[0] is epsilon
        elif rule[0] == '#':
            return '#'

    # condition for Non-Terminals
    if len(rule) != 0:
        if rule[0] in list(diction.keys()):

            # fres temporary list of result
            fres = []

            # bring rule[0] RHS
            rhs_rules = diction[rule[0]]

            # call first on each rule of RHS
            for itr in rhs_rules:
                indivRes = first(itr, non_term_user_def, term_user_def, diction, firsts)

                # if first of (its) variable is a list, then we will append all elements in the list to fres list.
                if type(indivRes) is list:
                    for i in indivRes:
                        fres.append(i)
                # if first of (its) variable is an element, the add it directly
                else:
                    fres.append(indivRes)

            # if no epsilon in result received return fres
            if '#' not in fres:
                return fres

            # if epsilon in result received return fres
            else:
                # apply epsilon, rule => f(ABC)=f(A)-{e} U f(BC)
                newList = []

                # remove epsilon from fres
                fres.remove('#')

                # if rule still have more than one element, then we will call first again
                if len(rule) > 1:

                    # find first for all rules after rule[0]
                    ansNew = first(rule[1:], non_term_user_def, term_user_def, diction, firsts)

                    # if ansNew is not empty
                    if ansNew != None:

                        # if ansNew is a list
                        if type(ansNew) is list:
                            newList = fres + ansNew

                        # if ansNew is an element
                        else:
                            newList = fres + [ansNew]

                    # if ansNew is empty, then store fres list on newList
                    else:
                        newList = fres
                    return newList

                """
                 if result is not already returned - control reaches her.
                 Lastly if eplison still persists - keep it in result of first
                """

                # add epsilon to fres list
                fres.append('#')

                return fres


```

```

"""
calculation of follow, follow function input is the split
result on Non-Terminal whose Follow we want to compute
"""
def follow(nt, start_symbol, rules, nonterm_userdef, term_userdef, diction, firsts, follows):

    # for start symbol return $ (recursion base case)
    solset = set()
    if nt == start_symbol:
        # return '$'
        solset.add('$')

    # solset - is result of computed 'follow' so far
    # check all occurrences for input, check in all rules
    for curNT in diction:

        # RHS for curNT
        rhs = diction[curNT]

        # go for all productions of NT
        for subrule in rhs:
            if nt in subrule:

                # call for all occurrences on - non-terminal in subrule
                while nt in subrule:

                    # find index of nt
                    index_nt = subrule.index(nt)

                    # bring all element after nt
                    subrule = subrule[index_nt + 1:]

                    # empty condition - call follow on LHS
                    if len(subrule) != 0:
                        # compute first if symbols on RHS of target Non-Terminal exists
                        res = first(subrule, nonterm_userdef, term_userdef, diction, firsts)

                        """
                        if epsilon in result apply rule - (A->aBX)- follow of -
                        follow(B)=(first(X)-{ep}) U follow(A)
                        """
                        if '#' in res:
                            newList = []

                            # remove epsilon from res
                            res.remove('#')

                            # find follow of curNT
                            ansNew = follow(curNT, start_symbol, rules, nonterm_userdef, term_userdef, diction, firsts, follows)

                            # if ansNew is not empty
                            if ansNew != None:
                                if type(ansNew) is list:
                                    newList = res + ansNew
                                # if ansNew is empty
                                else:
                                    newList = res + [ansNew]
                            else:
                                newList = res
                            res = newList
                    else:
                        # when nothing in RHS, go circular and take follow of LHS only if (NT in LHS)!=curNT
                        # if we have a spicial cases in our grammer
                        if nt != curNT:
                            List = []
                            if nt == 'S' or nt == "L":
                                for i in follows[start_symbol]:
                                    List.append(i)
                                return List
                            elif nt == 'O' or nt == "W":
                                first_N_ = firsts["N"]
                                for i in first_N_:
                                    if i == '#':
                                        continue
                                    List.append(i)
                                for j in follows['T']:
                                    List.append(j)
                                return List

                            else:
                                res = follow(curNT,  start_symbol, rules, nonterm_userdef, term_userdef, diction, firsts, follows)

                    # add follow result in set form
                    if res is not None:
                        if type(res) is list:
                            for g in res:
                                solset.add(g)
                        else:
                            solset.add(res)
    return list(solset)
```

```

def computeAllFirsts(rules, nonterm_userdef, term_userdef, diction, firsts):

    # clean all rules (strip and split rules, eliminate left recursion, eliminate left factoring)
    preprocessing(rules, diction)

    # calculate first for each rule (call first() on all RHS)
    # diction.keys() --> bring all non-terminals
    for y in list(diction.keys()):
        t = set()

        # get RHS for y which is on of non-terminals
        for sub in diction.get(y):

            # compute the first for one rule in RHS for y
            res = first(sub, nonterm_userdef, term_userdef, diction, firsts)

            # if res is not empty, res contain the first for one rule in RHS for y
            if res != None:

                # if res is a list add element one by one to set t
                if type(res) is list:
                    for u in res:
                        t.add(u)

                # if res is an element add to set t directly
                else:
                    t.add(res)

        # save result in 'firsts' list
        firsts[y] = t



def computeAllFollows(start_symbol, rules, nonterm_userdef, term_userdef, diction, firsts, follows):

    # for all non-terminals in diction
    for NT in diction:
        solset = set()

        sol = None

        # compute follow for special cases
        List = []
        if NT == 'S' or NT == 'L':
            for i in follows[start_symbol]:
                List.append(i)
            sol = List
        elif NT == 'O' or NT == "W":

            first_N_ = firsts['N']
            for i in first_N_:
                if i == '#':
                    continue
                List.append(i)
            for j in follows['T']:
                List.append(j)
            sol = List

        else:
            # calculate follow for other non-terminal
            sol = follow(NT, start_symbol, rules, nonterm_userdef, term_userdef, diction, firsts, follows)

        # if sol is not none which means if contain elements
        if sol is not None:
            for g in sol:
                solset.add(g)

        # add follow for non-termnal NT
        follows[NT] = solset


```

```

# create parse table
def createParseTable():
    import copy
    global diction, firsts, follows, term_userdef

    # find space size
    mx_len_first = 0
    mx_len_fol = 0

    # for each non-terminal look at the max length of first and follow as a string
    for u in diction:
        k1 = len(str(firsts[u]))
        k2 = len(str(follows[u]))
        if k1 > mx_len_first:
            mx_len_first = k1
        if k2 > mx_len_fol:
            mx_len_fol = k2


    # # prepare the head of the first table
    first_and_follow = f"{{:<{10}}} " \
                       f"{{:<{mx_len_first + 5}}}\n" \
                       .format("Non-T", "FIRST")

    # prepare the first for each non-terminal
    for u in diction:
        first_and_follow += f"{{:<{10}}} " \
                            f"{{:<{mx_len_first + 5}}}\n" \
                            .format(u, str(firsts[u]))


    # create matrix of row(NT) x [col(T) + 1($)]

    # create list of non-terminals
    ntlist = list(diction.keys())

    # create list of terminals
    terminals = copy.deepcopy(term_userdef)

    # add '$' to the list of terminals
    terminals.append('$')

    # create the initial empty state of ,matrix
    mat = []
    for x in diction:
        row = []
        for y in terminals:
            row.append('')
        # of $ append one more col
        mat.append(row)

    # Classifying grammar as LL(1) or not LL(1)
    grammar_is_LL = True

    # rules implementation
    for lhs in diction:
        rhs = diction[lhs]
        for y in rhs:
            res = first(y, nonterm_userdef, term_userdef, diction, firsts)
            # epsilon is present,
            # - take union with follow
            if '#' in res:
                if type(res) == str:
                    firstFollow = []
                    fol_op = follows[lhs]
                    if fol_op is str:
                        firstFollow.append(fol_op)
                    else:
                        for u in fol_op:
                            firstFollow.append(u)
                    res = firstFollow
                else:
                    res.remove('#')
                    res = list(res) + \
                          list(follows[lhs])
            # add rules to table
            ttemp = []
            if type(res) is str:
                ttemp.append(res)
                res = copy.deepcopy(ttemp)
            for c in res:
                xnt = ntlist.index(lhs)
                yt = terminals.index(c)
                if mat[xnt][yt] == '':
                    mat[xnt][yt] = mat[xnt][yt] \
                                   + f"{lhs}->{' '.join(y)}"
                else:
                    # if rule already present
                    if f"{lhs}->{y}" in mat[xnt][yt]:
                        continue
                    else:
                        grammar_is_LL = False
                        mat[xnt][yt] = mat[xnt][yt] \
                                       + f",{lhs}->{' '.join(y)}"

    # final state of parse table

    pares_table_string = []
    frmt = "{:>12}" * len(terminals)
    pares_table_string.append(frmt.format(*terminals))
    j = 0
    for y in mat:
        frmt1 = "{:>12}" * len(y)
        pares_table_string.append(f"{frmt1.format(*y)}\n")
        j += 1

    return (mat, grammar_is_LL, terminals, pares_table_string,  first_and_follow)


```

```

def validateStringUsingStackBuffer(parsing_table, grammarll1,
                                   table_term_list, input_string,
                                   term_userdef, start_symbol):

    # for more than one entries - in one cell of parsing table
    if grammarll1 == False:
        return f"\nInput String = " \
               f"\"{input_string}\"\n" \
               f"Grammar is not LL(1)"

    # implementing stack buffer
    stack = [start_symbol, '$']
    buffer = []

    # reverse input string store in buffer
    input_string = input_string.split()
    input_string.reverse()
    buffer = ['$'] + input_string


    # # prepare content of stack to display it on GUI
    stack_string = "{:>20} {:>20} {:>20}\n".format("Buffer", "Stack", "Action")

    while True:
        # end loop if all symbols matched , if stack contans $ and buffer the we will stop.
        if stack == ['$'] and buffer == ['$']:

            stack_string += "{:>20} {:>20} {:>20}\n".format(' '.join(buffer), ' '.join(stack), "Valid")

            return "\nValid String!", stack_string

        elif stack[0] not in term_userdef:

            # take font of buffer (y) and top of stack (x) indexes
            x = list(diction.keys()).index(stack[0])
            y = table_term_list.index(buffer[-1])

            if parsing_table[x][y] != '':
                # format table entry received
                entry = parsing_table[x][y]

                # prepare content of stack to display it on GUI
                stack_string += "{:>20} {:>20} {:>25}\n". \
                    format(' '.join(buffer), \
                           ' '.join(stack), \
                          f"T[{stack[0]}][{buffer[-1]}] = {entry}")


                lhs_rhs = entry.split("->")
                lhs_rhs[1] = lhs_rhs[1].replace('#', '').strip()
                entryrhs = lhs_rhs[1].split()
                stack = entryrhs + stack[1:]
            else:
                return f"\nInvalid String! No rule at " \
                       f"Table[{stack[0]}][{buffer[-1]}]." ,stack_string
        else:
            # stack top is Terminal
            if stack[0] == buffer[-1]:

                stack_string += "{:>20} {:>20} {:>20}\n" \
                    .format(' '.join(buffer), \
                            ' '.join(stack), \
                            f"Matched:{stack[0]}")

                # pop matched element from the stack and buffer
                buffer = buffer[:-1]
                stack = stack[1:]
            else:
                return "\nInvalid String! " \
                       "Unmatched terminal symbols" , stack_string


```

```

from tkinter import *
from PIL import ImageTk, Image
from tkinter.scrolledtext import ScrolledText
import main

root = Tk()
root.title("LL(1) parsar")
# root.minsize(1200, 800)

width = root.winfo_screenwidth()
height = root.winfo_screenheight()

# setting tkinter window size
root.geometry("%dx%d" % (width, height))


def click_button():

    """
    P: PROGRAM
    S: STMTS
    L: STMTS'
    M: STMT
    E: EXPR
    R: EXPR'
    T: TERM
    N: TERM'
    O: POWER
    W: POWER'
    F: FACTOR
    """

    rules = [
        "P -> S",
        "S -> M L",
        "L -> # | ; S",
        "M -> id = E",
        "E -> T R",
        "R -> + T R | - T R | #",
        "T -> O N",
        "N -> * O N | / O N | #",
        "O -> F W",
        "W -> ^ O | #",
        "F -> ( E ) | id | integer | @ F | ! F"
    ]
    non_term_user_def = ['P', 'S', 'L', 'M', 'E', 'R', 'T', 'N', 'O', 'W', 'F']
    term_user_def = [';', 'id', '=', '+', '-', '*', '/', '^', '(', ')', 'integer', '@', '!']

    Text_5.delete('1.0', END)
    Text_6.delete('1.0', END)
    # set vertical non terminals (start)
    Text_8.delete('1.0', END)
    Text_9.delete('1.0', END)
    Text_10.delete('1.0', END)
    Text_11.delete('1.0', END)
    Text_12.delete('1.0', END)
    Text_13.delete('1.0', END)
    Text_14.delete('1.0', END)
    Text_15.delete('1.0', END)
    Text_16.delete('1.0', END)
    Text_17.delete('1.0', END)
    Text_18.delete('1.0', END)
    # set vertical non terminals (start)

    # set horizontal terminals (start)
    Text_19.delete('1.0', END)
    Text_20.delete('1.0', END)
    Text_21.delete('1.0', END)
    Text_22.delete('1.0', END)
    Text_23.delete('1.0', END)
    Text_24.delete('1.0', END)
    Text_25.delete('1.0', END)
    Text_26.delete('1.0', END)
    Text_27.delete('1.0', END)
    Text_28.delete('1.0', END)
    Text_29.delete('1.0', END)
    Text_30.delete('1.0', END)
    Text_31.delete('1.0', END)
    Text_32.delete('1.0', END)
    # set horizontal terminals (start)


```

```
    sample_input_string = Text_1.get("1.0", 'end-1c')


    # diction - store rules inputted
    diction = {}
    # firsts - store computed firsts
    firsts = {}
    # follows - store computed follows
    follows = {}

    # computes all FIRSTs for all non-terminals
    firsts = main.computeAllFirsts(rules, non_term_user_def, term_user_def, diction,
firsts)
    print(firsts)
    # assuming first rule has start_symbol
    start_symbol = list(diction.keys())[0]

    # computes all follows for all non-terminals
    follows = main.computeAllFollows(start_symbol, rules, non_term_user_def,
term_user_def, diction, firsts, follows)

    # then generate parse table, after we compute first and follow
    (parsing_table, result, tabTerm, first_and_follow, pares_table_string) =
main.createParseTable()

    Text_6.insert("1.0", first_and_follow)
    # Text_4.insert("1.0", pares_table_string)

    # validate string input using stack-buffer concept
    if sample_input_string is not None:
        validity, stack_string = main.validateStringUsingStackBuffer(parsing_table,
result,
                                                                     tabTerm,
sample_input_string,
                                                                     term_user_def,
start_symbol)

        Text_5.insert("1.0", stack_string)
        Label_7.config(text=validity)
        print(validity)
    else:
        print("\nNo input String detected")

    # set vertical non terminals (start)
    Text_8.insert('1.0', non_term_user_def[0])
    Text_9.insert('1.0', non_term_user_def[1])
    Text_10.insert('1.0', non_term_user_def[2])
    Text_11.insert('1.0', non_term_user_def[3])
    Text_12.insert('1.0', non_term_user_def[4])
    Text_13.insert('1.0', non_term_user_def[5])
    Text_14.insert('1.0', non_term_user_def[6])
    Text_15.insert('1.0', non_term_user_def[7])
    Text_16.insert('1.0', non_term_user_def[8])
    Text_17.insert('1.0', non_term_user_def[9])
    Text_18.insert('1.0', non_term_user_def[10])
    # set vertical non terminals (start)

    # set horizontal terminals (start)
    Text_19.insert('1.0', term_user_def[0])
    Text_20.insert('1.0', term_user_def[1])
    Text_21.insert('1.0', term_user_def[2])
    Text_22.insert('1.0', term_user_def[3])
    Text_23.insert('1.0', term_user_def[4])
    Text_24.insert('1.0', term_user_def[5])
    Text_25.insert('1.0', term_user_def[6])
    Text_26.insert('1.0', term_user_def[7])
    Text_27.insert('1.0', term_user_def[8])
    Text_28.insert('1.0', term_user_def[9])
    Text_29.insert('1.0', term_user_def[10])
    Text_30.insert('1.0', term_user_def[11])
    Text_31.insert('1.0', term_user_def[12])
    Text_32.insert('1.0', '$')
    # set horizontal terminals (start)

    X_position = 300
    Y_postion = 60
    for x in range(len(parsing_table)):
        for y in range(len(parsing_table[x])):
            Text_33 = Text(root, width=8, height=1, borderwidth=0)
            Text_33.grid(row=x, column=y)
            Text_33.insert( END,parsing_table[x][y])
            Text_33.place(x=X_position, y=Y_postion)
            X_position += 75
        Text_33.place(x=X_position, y= Y_postion)
        Y_postion += 30
        X_position = 300

# ----------- Image logo (start)

frame = Frame(root, width=30, height=40)
frame.pack()
frame.place(anchor='center', x=850, y=200)

# Create an object of tkinter ImageTk
img = ImageTk.PhotoImage(Image.open("LL(1) parser logo.jpg"))

# Create a Label Widget to display the text or Image
label = Label(frame, image = img)
label.pack()

# ----------- Image logo (end)


```

```
# --------------------Input string (start)
Label_1 = Label(root, text='Input string:', font='Times 14')
Label_1.place(height=20, width=400, x=85, y=640)

Label_7 = Label(root, text='', font='Times 14')
Label_7.place(height=100, width=400, x=150, y=500)

Text_1 = Text(root, borderwidth=2)
Text_1.place(height=20, width=210, x=235, y=680)

Button_1 = Button(root, text="Check", background='#A3A3C6', font='Times 14',
command=click_button)
Button_1.place(height=30, width=100, x=290, y=720)

# --------------------Input string (end)

# ---------------------Rule after modifications (start)
Label_2 = Label(root, text='Rules after modifications', font='Times 14')
Label_2.place(height=20, width=200, x=15, y=330)

Text_2 = Text(root, width=50, height=1, borderwidth=0, background='#D8D0EF')
Text_2.place(height=400, width=200, x=20, y=360)

rules_a_m = "P -> S\n\nS -> M L\n\nL -> # | ; S\n\nM -> id = E\n\nE -> T R\n\nR -> + T R |
- T R | #" \
            "\n\nT -> O N\n\nN -> * O N | / O N | #\n\nO -> F W\n\nW -> ^ O | #\n\n" \
            "F -> ( E ) | id |\ninteger | @ F | ! F"


Text_2.insert("1.0", rules_a_m)

# ---------------------Rule after modifications (end)

# ---------------------Rule before modifications (start)
Label_3 = Label(root, text='Rules before modifications', font='Times 14')
Label_3.place(height=20, width=200, x=15, y=30)

Text_3 = Text(root, width=50, height=1, borderwidth=0, bg="#F7F6D1")
Text_3.place(height=240, width=200, x=20, y=60)

rules_b_m = "P -> S\n\nS -> M | M ; S\n\nM -> id = E\n\nE -> E + T | E - T | T" \
            "\n\nT -> T * O | T / O | O\n\nO -> F ^ O | F\n\nF -> ( E ) | id |\n integer |
@ F | ! F "

Text_3.insert("1.0", rules_b_m)
# ---------------------Rule before modifications (end)

# --------------------Parsing table (start)

# ---------------------- horizontal text or Non-terminals (start)
Text_8 = Text(root, width=50, height=1, borderwidth=0)
Text_8.place(height=20, width=20, x=270, y=60)

Text_9 = Text(root, width=50, height=1, borderwidth=0)
Text_9.place(height=20, width=20, x=270, y=90)

Text_10 = Text(root, width=50, height=1, borderwidth=0)
Text_10.place(height=20, width=20, x=270, y=120)

Text_11 = Text(root, width=50, height=1, borderwidth=0)
Text_11.place(height=20, width=20, x=270, y=150)

Text_12 = Text(root, width=50, height=1, borderwidth=0)
Text_12.place(height=20, width=20, x=270, y=180)

Text_13 = Text(root, width=50, height=1, borderwidth=0)
Text_13.place(height=20, width=20, x=270, y=210)

Text_14 = Text(root, width=50, height=1, borderwidth=0)
Text_14.place(height=20, width=20, x=270, y=240)

Text_15 = Text(root, width=50, height=1, borderwidth=0)
Text_15.place(height=20, width=20, x=270, y=270)

Text_16 = Text(root, width=50, height=1, borderwidth=0)
Text_16.place(height=20, width=20, x=270, y=300)

Text_17 = Text(root, width=50, height=1, borderwidth=0)
Text_17.place(height=20, width=20, x=270, y=330)

Text_18 = Text(root, width=50, height=1, borderwidth=0)
Text_18.place(height=20, width=20, x=270, y=360)

# ---------------------- horizontal text for Non-terminals (end)

```

```
# ---------------------- Vertical text for terminals (start)
Text_19 = Text(root, borderwidth=0)
Text_19.place(height=20, width=20, x=330, y=35)

Text_20 = Text(root, borderwidth=0)
Text_20.place(height=20, width=60, x=380, y=35)

Text_21 = Text(root, borderwidth=0)
Text_21.place(height=20, width=10, x=480, y=35)

Text_22 = Text(root, width=50, height=1, borderwidth=0)
Text_22.place(height=20, width=10, x=550, y=35)

Text_23 = Text(root, width=50, height=1, borderwidth=0)
Text_23.place(height=20, width=10, x=630, y=35)

Text_24 = Text(root, width=50, height=1, borderwidth=0)
Text_24.place(height=20, width=10, x=700, y=35)

Text_25 = Text(root, width=50, height=1, borderwidth=0)
Text_25.place(height=20, width=10, x=780, y=35)

Text_26 = Text(root, width=50, height=1, borderwidth=0)
Text_26.place(height=20, width=10, x=850, y=35)

Text_27 = Text(root, width=50, height=1, borderwidth=0)
Text_27.place(height=20, width=10, x=930, y=35)

Text_28 = Text(root, width=50, height=1, borderwidth=0)
Text_28.place(height=20, width=10, x=1000, y=35)

Text_29 = Text(root, width=50, height=1, borderwidth=0)
Text_29.place(height=20, width=10, x=1080, y=35)

Text_30 = Text(root, width=50, height=1, borderwidth=0)
Text_30.place(height=20, width=10, x=1150, y=35)

Text_31 = Text(root, width=50, height=1, borderwidth=0)
Text_31.place(height=20, width=10, x=1225, y=35)

Text_32 = Text(root, width=50, height=1, borderwidth=0)
Text_32.place(height=20, width=10, x=1380, y=35)

# ---------------------- Vertical text for Non-terminals (end)

# ---------------------Parsing table  (end)

# ---------------------Stack (start)
Label_5 = Label(root, text='Stack', font='Times 14')
Label_5.place(height=20, width=200, x=1050, y=430)

Text_5 = ScrolledText(root)
# Text(root, width=50, height=1, borderwidth=2)
Text_5.place(height=300, width=550, x=890, y=460)
# ---------------------Stack (end)

# ---------------------First && Follow table (start)
Label_6 = Label(root, text='First and Follow table', font='Times 14')
Label_6.place(height=20, width=200, x=550, y=430)

Text_6 = ScrolledText(root, width=50, height=10)
# Text(root, width=50, height=1, borderwidth=2)
Text_6.place(height=300, width=400, x=470, y=460)

# ---------------------First && Follow table (end)

root.mainloop()
```

5.Challanges

- The most challenging part of creating the GUI for the second part was creating the parsing table, since each rule must be in a precise position (Table [non-terminal] [terminal]).
- Solve an error we faced in most of the source codes for calculating Follow function. After analyzing all cases to calculate the Follow function, we found that there is a special case not handling in existed code. Which is when a recursive between two non-terminals. Such as, Follow(A) = Follow(B) and Follow(B) = Follow(A). we handle this case by adding if-else statements inside the follow method.

- Not knowing how to convert a concept to a code.
- Difficulty in knowing some Methods due to lack of knowledge of Python language.

6. References

- Spivak, Ruslan. “Let’s Build A Simple Interpreter. Part 8. - Ruslan’s Blog.” Ruslan’s Blog, 18 Jan. 2016, https://ruslanspivak.com/lsbasi-part8/.
- “CFG to LL(k).” Tool, https://cyberzhg.github.io/toolbox/cfg2ll. Accessed 2 Nov. 2022.
- “Compiler Design LL(1) Parser in Python - GeeksforGeeks.” GeeksforGeeks, 18 Feb. 2022, https://www.geeksforgeeks.org/compiler-design-ll1-parser-in-python/.
- “Context-Free Grammars.” UW Computer Sciences User Pages, https://pages.cs.wisc.edu/~fischer/cs536.s08/course.hold/html/NOTES/3.CFG.html. Accessed 2 Nov. 2022.
- Tasbeer, Tasbeer. “Parsing - Grammar for Arithmetic Expressions - Stack Overflow.” Stack Overflow, https://stackoverflow.com/questions/645762/grammar-for-arithmetic-expressions. Accessed 2 Nov. 2022.
- S. Sahu, “Automata From Regular Expressions,” GitHub, Oct. 31, 2022. https://github.com/sdht0/automata-from-regex (accessed Nov. 05, 2022).
- G. of geeks, “Regular Expression to DFA,” GeeksforGeeks, Feb. 19, 2022. https://www.geeksforgeeks.org/regular-expression-to-dfa/ (accessed Nov. 05, 2022).
- Balan, “Converter from REGEX to DFA NFA,” GitHub, May 06, 2020. https://github.com/alexandru-balan/regex_to_dfa