Kathmandu University

Department of Computer Science and Engineering

Dhulikhel, Kavre

Lab Report #1

Compiler Design

[Course Code: COMP 409]

[For the partial fulfillment of 4th year/1st Semester in Computer Engineering]

Submitted by:

Sabin Thapa

Roll no. 54

CE 4th Year

Submitted to:

Mr. Sushil Nepal

Assistant Professor

Department of Computer Science and Engineering

Submission date: Sept. 19, 2022

Tasks

a. Read a file consisting of several keywords, identifiers, relational operators listed in each line. Write a piece of code and try to find those keywords, identifiers, relational operators etc. If they are matched, then make a symbol table entry.

Source Code

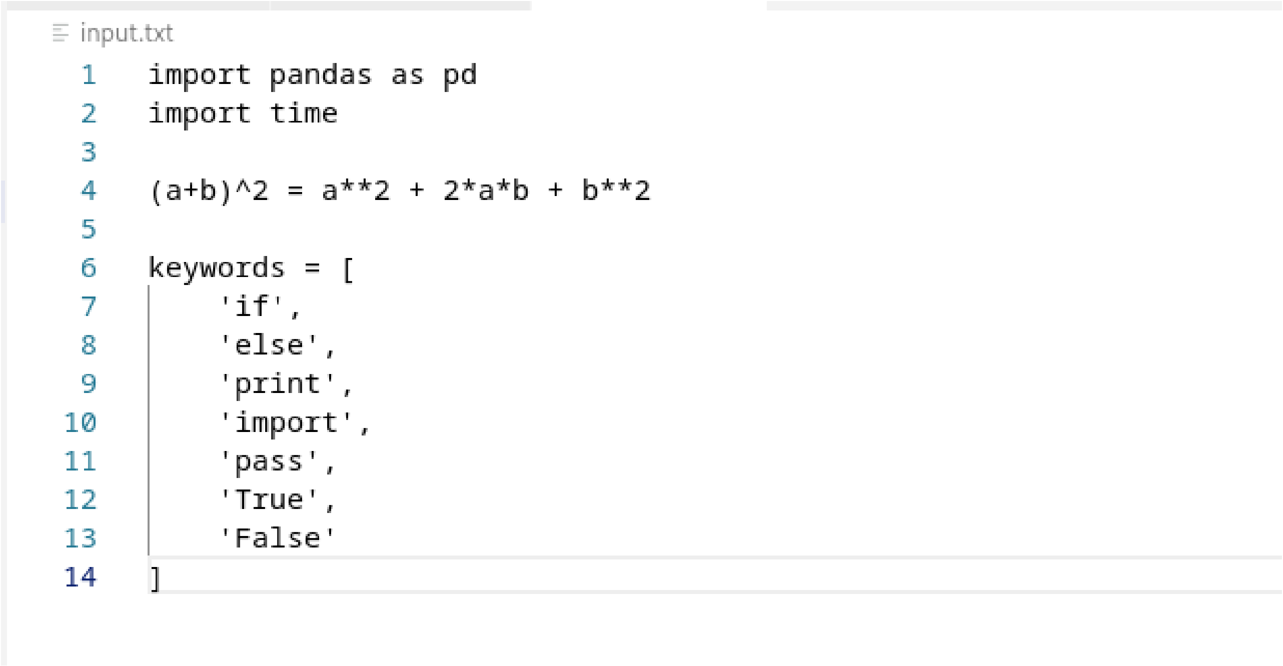
```
from curses.ascii import isalpha, isdigit

line_number = 0
current = ' '
file_path = ' '

reserved_words = set((
       'import',
       'global',
       'print',
       'while',
       'for',
       'if',
       'elif',
       'else',
       'True',
       'False',
       'None',
       'as',
       'from',
       'try',
       'except',
       'and',
       'break',
       'class',
       'continue',
       'def',
       'lambda',
       'pass',
       'with',
       'del',
       'in',
       'range',
       'return',
   ))

operators_mappings = {
   '/': "DIV_OP",
   '*': "MULT_OP",
   '+': "PLUS_OP",
   '-': "MINUS_OP",
   '%': "MOD_OP",
   '**': "POW_OP",
   '^': "POW_OP",
   '=': "ASSIGN_OP",
   ',': "COMMA_OP"
}

special_characters = [
   '"',
   "'",
   '#',
   '(',
   ')',
   '[',
   ']',
   ':',]
```

```python
special_char_mappings = {
    '#': "HASH",
    "'": "SINGLE_QUOTE",
    '"': "DOUBLE_QUOTE",
    '(': 'PARAM_OPEN',
    ')': 'PARAM_CLOSE',
    '[': 'BRACKET_OPEN',
    ']': 'BRACKET_CLOSE',
    ':': 'DELIMITER_COLON'
}

data_type = {
    'int': 'INTEGER',
    'float': 'FLOATING_POINT',
    'bool': 'BOOLEAN',
    'str': 'STRING',
    'dict': 'DICTIONARY',
    'list': 'LIST',
}

def scan_input(input):
    global current
    if current.isdigit():
        val = 0
        while True:
            val = 10*val + int(current)
            current = input.read(1)
            if not current.isdigit():
                if current == '.':
                    i = 1
                    while True:
                        current = input.read(1)
                        if not current.isdigit():
                            break
                        val = val+float(int(current))/10**i
                        i += 1
                break
        return data_type[str(type(val))[8:-2]],val

    if current in special_characters:
        if current == '"' or current == "'":
            quotation = current
            buffer = []
            while True:
                buffer.append(current)
                current = input.read(1)
                if current == quotation:
                    current = input.read(1)
                    break
            w = ''.join(buffer)
            return data_type['str'],w
        temp = current
        current = input.read(1)
        return special_char_mappings[temp],temp


    if current.isalpha():
        buffer = []
        while True:
            buffer.append(current)
            current = input.read(1)
            if not current.isalnum(): break
        word = ''.join(buffer)
        if word in reserved_words:
            return 'reserved_words',word
        return 'IDENTIFIER',word

    t = (operators_mappings[current],current)
    current = ' '
    return t

def generate_tokens(input):
    global current,line_number
    tokens = []
    current = input.read(1)
    while True:
        if (current == ' ' or current == '\t'):
            current = input.read(1)
            continue
        if (current == '\n' or current == ';'):
            current = input.read(1)
            line_number += 1
            continue
        if current:
            tokens.append(scan_input(input))
        else:
            break
    print('Symbol Table:')
    for token in tokens:
        print(token)

if __name__=='__main__':
    file_path = input("Enter the path of the file (eg: input.txt): ")
    inputs = open(repr(file_path)[1:-1],'r')
    generate_tokens(inputs)
```

```
inputs.close()
```

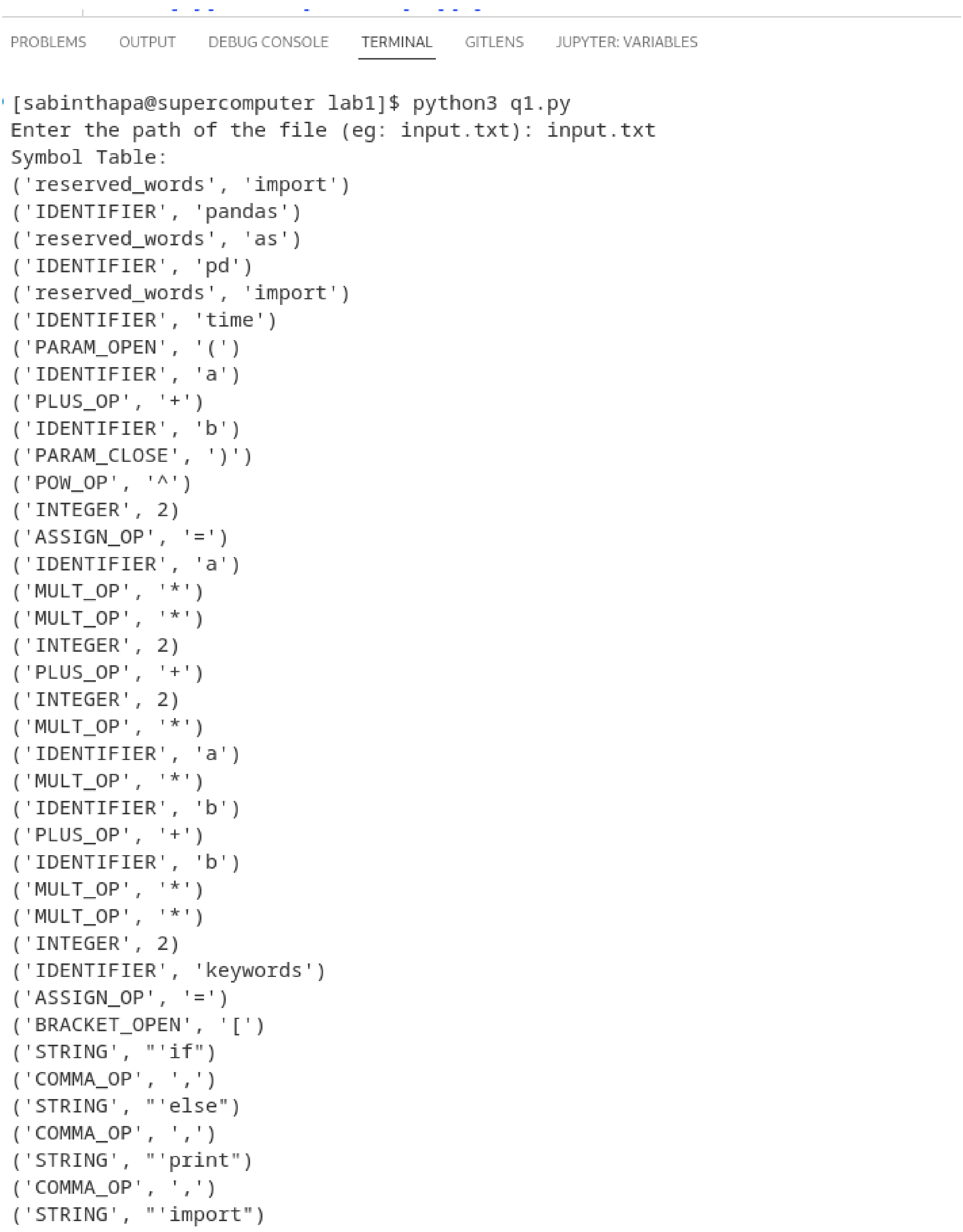
Output

Input file

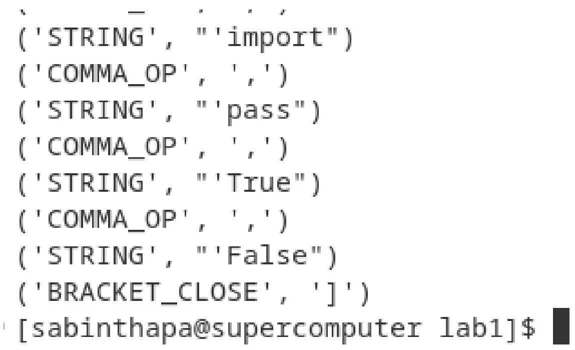


```
import pandas as pd
import time

(a+b)^2 = a**2 + 2*a*b + b**2

keywords = [
    'if',
    'else',
    'print',
    'import',
    'pass',
    'True',
    'False'
]
```

Terminal Output



```
[sabinthapa@supercomputer lab1]$ python3 q1.py
Enter the path of the file (eg: input.txt): input.txt
Symbol Table:
('reserved_words', 'import')
('IDENTIFIER', 'pandas')
('reserved_words', 'as')
('IDENTIFIER', 'pd')
('reserved_words', 'import')
('IDENTIFIER', 'time')
('PARAM_OPEN', '(')
('IDENTIFIER', 'a')
('PLUS_OP', '+')
('IDENTIFIER', 'b')
('PARAM_CLOSE', ')')
('POW_OP', '^')
('INTEGER', 2)
('ASSIGN_OP', '=')
('IDENTIFIER', 'a')
('MULT_OP', '*')
('MULT_OP', '*')
('INTEGER', 2)
('PLUS_OP', '+')
('INTEGER', 2)
('MULT_OP', '*')
('IDENTIFIER', 'a')
('MULT_OP', '*')
('IDENTIFIER', 'b')
('PLUS_OP', '+')
('IDENTIFIER', 'b')
('MULT_OP', '*')
('MULT_OP', '*')
('INTEGER', 2)
('IDENTIFIER', 'keywords')
('ASSIGN_OP', '=')
('BRACKET_OPEN', '[')
('STRING', "'if")
('COMMA_OP', ',')
('STRING', "'else")
('COMMA_OP', ',')
('STRING', "'print")
('COMMA_OP', ',')
('STRING', "'import")
```



```
('STRING', "'import")
('COMMA_OP', ',')
('STRING', "'pass")
('COMMA_OP', ',')
('STRING', "'True")
('COMMA_OP', ',')
('STRING', "'False")
('BRACKET_CLOSE', ']')
[sabinthapa@supercomputer lab1]$
```

b. Write a program to implement a Regular Expression. The program must verify a string given as input into the program and check whether the given input is valid or invalid. You're not supposed to use the library.

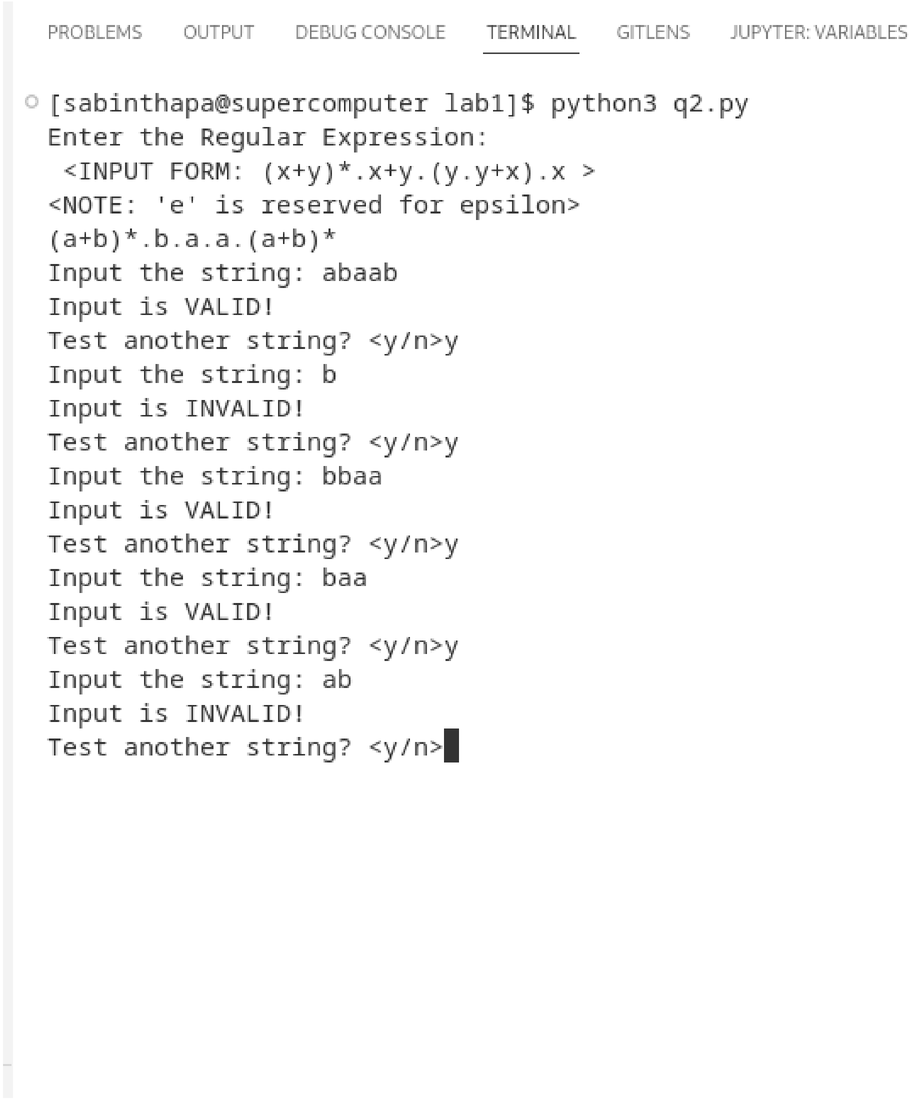
Source Code

```
DFA = {}
node_dict = {}
alphabet = []
data = []

def evaluate_postfix(str):
   stack = []
   for i in str:
       if i == ' ':
           continue
       elif i=='+' or i=='|':
           if stack:
               if stack[len(stack)-1] == '(':
                   stack.append(i)
               else:
                   data.append([stack.pop()])
                   stack.append(i)
           else:
               stack.append(i)
       elif i=='.':
           if stack:
                if stack[len(stack)-1] == '(' or stack[len(stack)-1] ==
'+' or stack[len(stack)-1] == '|':
                   stack.append(i)
               else:
                   data.append([stack.pop()])
                   stack.append(i)
           else:
               stack.append(i)
       elif i=='(':
           stack.append(i)
       elif i==')':
           j= stack.pop()
           while j != '(':
               data.append([j])
               j = stack.pop()
       else:
           data.append([i])
   while stack:
       data.append([stack.pop()])

def alpha():
   for d in data:
         if not(d[0] == '*' or d[0]=='.' or d[0]=='+' or d[0]=='|' or
d[0]=='#'):
```

```
            if d[0] not in alphabet:
                alphabet.append(d[0])

def evaluate_firstpos():
    stack = []
    for d in data:
        if d[0] == '+' or d[0] == '|':
            val2 = stack.pop()
            val1 = stack.pop()
            for e in val2:
                if e not in val1:
                    val1.append(e)
            stack.append(val1)
            d[3] = val1.copy()
        elif d[0] == '.':
            val2 = stack.pop()
            val1 = stack.pop()
            if d[3]:
                for e in val2:
                    if e not in val1:
                        val1.append(e)
            stack.append(val1)
            d[3] = val1.copy()
        elif d[0] == 'e':
            stack.append([0])
            d[3] = [0]
        elif d[0] == '*':
            val = stack.pop()
            stack.append(val)
            d[3] = val.copy()
        else:
            stack.append([d[1]])
            d[3] = [d[1]]

def evaluate_lastpos():
    stack = []
    for d in data:
        if d[0] == '+' or d[0] == '|':
            val2 = stack.pop()
            val1 = stack.pop()
            for e in val2:
                if e not in val1:
                    val1.append(e)
            stack.append(val1)
            d[4] = val1.copy()
        elif d[0] == '.':
            val2 = stack.pop()
            val1 = stack.pop()
            if d[4]:
                for e in val1:
                    if e not in val2:
                        val2.append(e)
            stack.append(val2)
```

```
           d[4] = val2.copy()
       elif d[0] == 'e':
           stack.append([0])
           d[4] = [0]
       elif d[0] == '*':
           val = stack.pop()
           stack.append(val)
           d[4] = val.copy()
       else:
           stack.append([d[1]])
           d[4] = [d[1]]


def evaluate_nullable():
   stack = []
   for d in data:
       if d[0] == '+' or d[0] == '|':
           val2 = stack.pop()
           val1 = stack.pop()
           val = val1 or val2
           stack.append(val)
           d.append(val)
           d.append(val1)
           d.append(val2)
       elif d[0] == '.':
           val2 = stack.pop()
           val1 = stack.pop()
           val = val1 and val2
           stack.append(val)
           d.append(val)
           d.append(val1)
           d.append(val2)
       elif d[0] == 'e':
           stack.append(True)
           d.append(True)
           d.append(False)
           d.append(False)
       elif d[0] == '*':
           stack.pop()
           stack.append(True)
           d.append(True)
           d.append(False)
           d.append(False)
       else:
           stack.append(False)
           d.append(False)
           d.append(False)
           d.append(False)

def label():
   i = 1
   j = 1
   for d in data:
        if not(d[0] =='e' or d[0] =='|' or d[0] =='+' or d[0] =='.' or
d[0] =='*'):
           d.append(i)
           i += 1
       else:
           d.append(100-j-i)
           j += 1

def make_dict():
   for d in data:
       d.append([])
       a = f'{d[1]}'
       node_dict.update({a:d})

def evaluate_followpos():
   stack = []
   for d in data:
       if d[0] == '+' or d[0] == '|':
           stack.pop()
           stack.pop()
           stack.append(d[1])
       elif d[0] == '.':
           val2 = stack.pop()
           val1 = stack.pop()
           for i in node_dict[f'{val1}'][4]:
               follow = node_dict[f'{i}'][5]
               first = node_dict[f'{val2}'][3]
               for k in first:
                   if k not in follow:
                       follow.append(k)
               node_dict[f'{i}'][5] = follow.copy()
           stack.append(d[1])
       elif d[0] == 'e':
           stack.append(d[1])
       elif d[0] == '*':
           val = stack.pop()
           for i in d[4]:
               follow = node_dict[f'{i}'][5]
               first = d[3]
               for k in first:
                   if k not in follow:
                       follow.append(k)
               node_dict[f'{i}'][5] = follow.copy()
           stack.append(d[1])
       else:
           stack.append(d[1])

def update_data():
   for d in data:
       d = node_dict[f'{d[1]}']

def create_DFA():
   n = f'{data[-1][3]}'
   DFA[n] = {}
   for a in alphabet:
       DFA[n][a] = None
   DFA[n]['value'] = data[-1][3]
   if data[-2][1] in DFA[n]['value']:
       DFA[n]['valid'] = True
   else:
       DFA[n]['valid'] = False
   nodes = [n]
   while nodes:
       node = nodes.pop()
       for a in alphabet:
           follow = []
           for d in DFA[node]['value']:
               if node_dict[f'{d}'][0] == a:
                   for i in node_dict[f'{d}'][5]:
                       if i not in follow:
                           follow.append(i)
           follow = sorted(follow)
           if follow:
               DFA[node][a] = f'{follow}'
               try:
                   a = DFA[f'{follow}']
               except:
                   nodes.append(f'{follow}')
                   DFA[f'{follow}'] = {}
                   for a in alphabet:
                       DFA[f'{follow}'][a] = None
                   DFA[f'{follow}']['value'] = follow
                   if data[-2][1] in DFA[f'{follow}']['value']:
                       DFA[f'{follow}']['valid'] = True
                   else:
                       DFA[f'{follow}']['valid'] = False

def validate_string(string):
   valid = False
   n = next(iter(DFA))
   if string:
       for s in string:
           if s not in alphabet:
               return False
           try:
               n = DFA[n][s]
           except:
               return False
   try:
       valid = DFA[n]['valid']
   except:
       return False
   return valid

def printer():
   for val in data:
       print(val)

if __name__=="__main__":
   reg_ex = input("Enter the Regular Expression: \n <INPUT FORM:
(x+y)*.x+y.(y.y+x).x >\n<NOTE: 'e' is reserved for epsilon>\n")
   reg_ex = '('+reg_ex+').#'
   evaluate_postfix(reg_ex)
   alpha()
   label()
   evaluate_nullable()
   evaluate_firstpos()
   evaluate_lastpos()
   make_dict()
   evaluate_followpos()
   update_data()
   create_DFA()
   while True:
       inp = input("Input the string: ")
       if validate_string(inp):
           print('Input is VALID!')
       else:
           print('Input is INVALID!')

       if input("Test another string? <y/n>") == 'n':
           break
```

Output



```
○ [sabinthapa@supercomputer lab1]$ python3 q2.py
  Enter the Regular Expression:
   <INPUT FORM: (x+y)*.x+y.(y.y+x).x >
  <NOTE: 'e' is reserved for epsilon>
  (a+b)*.b.a.a.(a+b)*
  Input the string: abaab
  Input is VALID!
  Test another string? <y/n>y
  Input the string: b
  Input is INVALID!
  Test another string? <y/n>y
  Input the string: bbaa
  Input is VALID!
  Test another string? <y/n>y
  Input the string: baa
  Input is VALID!
  Test another string? <y/n>y
  Input the string: ab
  Input is INVALID!
  Test another string? <y/n>
```

Conclusion

In the first question, a text file containing several keywords, identifiers and operators was read and each of them was identified and matched and finally a symbol table entry was made. In the second question, a program was written to implement regular expression. As we can see, only the strings that are valid for the given regular expression are accepted and those that are not valid are not accepted. This is my attempt to evaluate the regular expression without using any library by using the concepts of DFA, firstpos, followpos, nullable as learned in the class.