Kathmandu University

Department of Computer Science and Engineering

Dhulikhel, Kavre

Lab Report #2

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. 20, 2022

Tasks

Write a program to simulate a Deterministic Finite Automata. Your program should clearly demonstrate the number of states. The program takes several valid as well as invalid data to test whether the string is valid or not.

Source Code

```
import graphviz # for drawing the graph

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]=='#' or d[0]=='e'):
            if d[0] not in alphabet:
                alphabet.append(d[0])


def label():
    i = 1
    j = 1
    for d in data:
```

```
        #e = epsilon
         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 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 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([])
            d[3] = []
        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([])
            d[4] = []
        elif d[0] == '*':
            val = stack.pop()
            stack.append(val)
            d[4] = val.copy()
        else:
            stack.append([d[1]])
            d[4] = [d[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 graph_printer(regular_expression):
    graph = graphviz.Digraph(f'DFA:{regular_expression}')
    graph.attr(layout='dot')
    keys = DFA.keys()
    current = 'START'
    graph.node(current, shape = 'circle', style='filled', color='white')
    graph.node(str(0), shape = 'circle', style='filled',color='yellow')
    graph.edge(current,str(0))
    for i,k in enumerate(keys):
        current = str(i)
        for a in alphabet:
            try:
                edge = DFA[k][a]
                VALID = DFA[edge]['VALID']
                for i,key in enumerate(keys):
                    if edge == key:
                                graph.node(str(i), shape = 'doublecircle' if
VALID  else  'circle',  style='filled',  color='green'  if  VALID  else
'yellow')
                        graph.edge(current, str(i),label=a)
            except:
                                    graph.node('REJECT',  shape='circle',
style='filled',color='red')
                graph.edge(current,'REJECT', label = a)
```

```
    graph.format = 'png'
    graph.render(directory='graph/').replace('\\', '/')



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


if __name__=="__main__":
    regular_expression = input("Enter the Regular Expression: \n <INPUT FORM: (x+y)*.x+y.(y.y+x).x >\n<NOTE: 'e' is reserved for epsilon>\n")
    #Augmenting the given grammar
    regular_expression = '('+regular_expression+').#'
    evaluate_postfix(regular_expression)
    alpha()
    label()
    evaluate_nullable()
    evaluate_firstpos()
    evaluate_lastpos()
    make_dict()
    evaluate_followpos()
    update_data()
    create_DFA()
    print(DFA,'\n')
    printer()
    graph_printer(regular_expression)

    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
```

Explanation

In DFA, for each input symbol, there can only be one state transition. As it has a finite number of states, the machine is called Deterministic Finite Machine or Deterministic Finite Automaton. In order to convert the Regular Expression directly into DFA, we first augment the given regular expression by concatenating it with the special symbol **#.** We then create the syntax tree for the augmented regular expression and number each alphabet symbol including the #. We then traverse the tree to construct the functions **nullable, firstpos, lastpos** and **followpos.** Finally, DFA is drawn from the followpos.

The python implementation of the above explanation is shown above and the output is shown below.

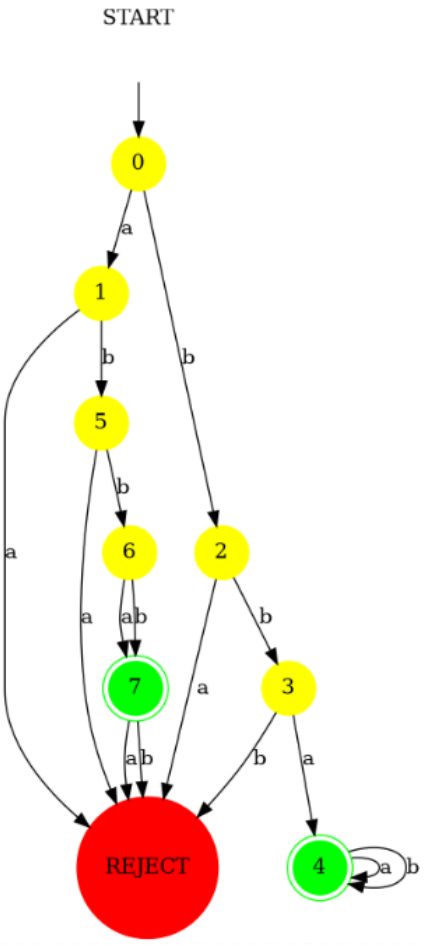
Output 1

```
(venv) [sabinthapa@supercomputer lab2]$ python3 q1.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|b)|b.b.a.(a|b)*
{'[1, 6]': {'a': '[2]', 'b': '[7]', 'value': [1, 6], 'VALID': False}, '[2]': {'a': None, 'b': '[
3]', 'value': [2], 'VALID': False}, '[7]': {'a': None, 'b': '[8]', 'value': [7], 'VALID': False}
, '[8]': {'a': '[9, 10, 11]', 'b': None, 'value': [8], 'VALID': False}, '[9, 10, 11]': {'a': '[9
, 10, 11]', 'b': '[9, 10, 11]', 'value': [9, 10, 11], 'VALID': True}, '[3]': {'a': None, 'b': '[
4, 5]', 'value': [3],
(venv) [sabinthapa@supercomputer lab2]$ python3 q1.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|b)|b.b.a.(a|b)*
{'[1, 6]': {'a': '[2]', 'b': '[7]', 'value': [1, 6], 'VALID': False}, '[2]': {'a': None, 'b': '[
3]', 'value': [2], 'VALID': False}, '[7]': {'a': None, 'b': '[8]', 'value': [7], 'VALID': False}
, '[8]': {'a': '[9, 10, 11]', 'b': None, 'value': [8], 'VALID': False}, '[9, 10, 11]': {'a': '[9
, 10, 11]', 'b': '[9, 10, 11]', 'value': [9, 10, 11], 'VALID': True}, '[3]': {'a': None, 'b': '[
4, 5]', 'value': [3], 'VALID': False}, '[4, 5]': {'a': '[11]', 'b': '[11]', 'value': [4, 5], 'VA
LID': False}, '[11]': {'a': None, 'b': None, 'value': [11], 'VALID': True}}


['a', 1, False, [1], [1], [2]]
['b', 2, False, [2], [2], [3]]
['.', 96, False, [1], [2], []]
['b', 3, False, [3], [3], [4, 5]]
['.', 94, False, [1], [3], []]
['a', 4, False, [4], [4], [11]]
['b', 5, False, [5], [5], [11]]
['|', 91, False, [4, 5], [4, 5], []]
['.', 90, False, [1], [4, 5], []]
['b', 6, False, [6], [6], [7]]
['b', 7, False, [7], [7], [8]]
['.', 87, False, [6], [7], []]
['a', 8, False, [8], [8], [9, 10, 11]]
['.', 85, False, [6], [8], []]
['a', 9, False, [9], [9], [9, 10, 11]]
['b', 10, False, [10], [10], [9, 10, 11]]
['|', 82, False, [9, 10], [9, 10], []]
['*', 81, True, [9, 10], [9, 10], []]
['.', 80, False, [6], [9, 10, 8], []]
['|', 79, False, [1, 6], [4, 5, 9, 10, 8], []]
['#', 11, False, [11], [11], []]
['.', 77, False, [1, 6], [11], []]
Input the string:
```

```
Input the string: abba
Input is VALID!
Test another string? <y/n>y
Input the string: bbaa
Input is VALID!
Test another string? <y/n>y
Input the string: bbab
Input is VALID!
Test another string? <y/n>y
Input the string: ab
Input is INVALID!
Test another string? <y/n>y
Input the string: abbabb
Input is INVALID!
Test another string? <y/n>y
Input the string: bbaa
Input is VALID!
Test another string? <y/n>n
(venv) [sabinthapa@supercomputer lab2]$
```

DFA:



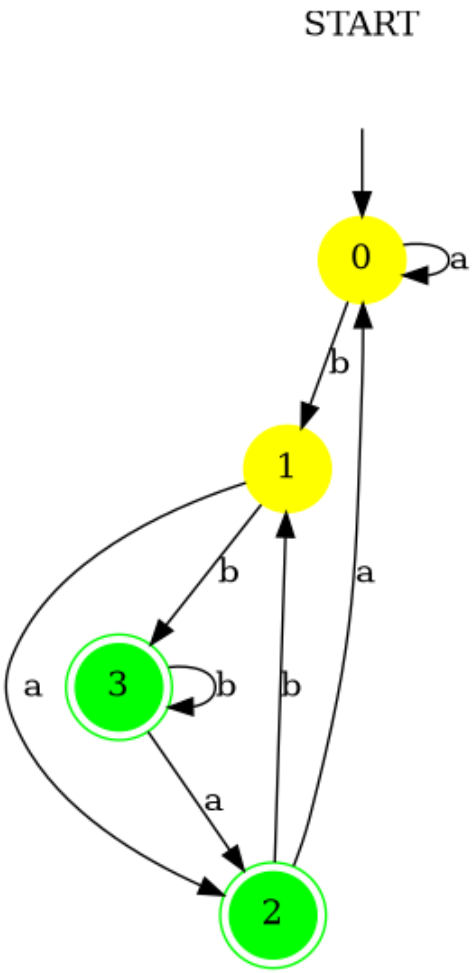
Output 2

```
(venv) [sabinthapa@supercomputer lab2]$ python3 q1.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|b)
{'[1, 2, 3]': {'a': '[1, 2, 3]', 'b': '[1, 2, 3, 4, 5]', 'value': [1, 2, 3], 'VALID': False}, '[
1, 2, 3, 4, 5]': {'a': '[1, 2, 3, 6]', 'b': '[1, 2, 3, 4, 5, 6]', 'value': [1, 2, 3, 4, 5], 'VAL
ID': False}, '[1, 2, 3, 6]': {'a': '[1, 2, 3]', 'b': '[1, 2, 3, 4, 5]', 'value': [1, 2, 3, 6], '
VALID': True}, '[1, 2, 3, 4, 5, 6]': {'a': '[1, 2, 3, 6]', 'b': '[1, 2, 3, 4, 5, 6]', 'value': [
1, 2, 3, 4, 5, 6], 'VALID': True}}

['a', 1, False, [1], [1], [1, 2, 3]]
['b', 2, False, [2], [2], [1, 2, 3]]
['|', 96, False, [1, 2], [1, 2], []]
['*', 95, True, [1, 2], [1, 2], []]
['b', 3, False, [3], [3], [4, 5]]
['.', 93, False, [1, 2, 3], [3], []]
['a', 4, False, [4], [4], [6]]
['b', 5, False, [5], [5], [6]]
['|', 90, False, [4, 5], [4, 5], []]
['.', 89, False, [1, 2, 3], [4, 5], []]
['#', 6, False, [6], [6], []]
['.', 87, False, [1, 2, 3], [6], []]


Input the string: aba
Input is VALID!
Test another string? <y/n>y
Input the string: bbb
Input is VALID!
Test another string? <y/n>y
Input the string: aaaaaaaaba
Input is VALID!
Test another string? <y/n>y
Input the string: bbbbbbbb
Input is VALID!
Test another string? <y/n>y
Input the string: bab
Input is INVALID!
Test another string? <y/n>y
Input the string: aaa
Input is INVALID!
Test another string? <y/n>y
Input the string: bababa
Input is VALID!
Test another string? <y/n>n
(venv) [sabinthapa@supercomputer lab2]$
```

DFA:



Conclusion

In this way, the program to simulate DFA was implemented in Python. The number of states are clearly demonstrated in the graph and the program can take several valid as well as invalid data to test whether the string is valid or not.