EXP:2: REGULAR EXPRESSION TO NFA

AIM: To Design a converter to convert Regular expression to NFA.

LANGUAGE USED: Python 3

ALGORITHM/PROCEDURE: -

1. Write the given code in Python complier
2. We used the input given grammar and run by the given functions.
3. We use the transition functions and get the input from the class.
4. We print the transition table for the required grammar taken as input.
5. We print the transition table and check the grammar tree diagram consecutively.
6. We get the output of regular expression to NFA for a given code.

SOURCE CODE: -

```
class Type:
    NONE   = 0
    SYMBOL = 1
    CONCAT = 2
    UNION  = 3
    KLEENE = 4


class ExpressionTree:


    def __init__(self, _type, value=None):
        self._type = _type
        self.value = value
        self.left = None
        self.right = None
```

```
def constructTree(regexp):
    stack = []
    z=ExpressionTree(Type.NONE)
    for c in regexp:
        if c.isalpha():
            stack.append(ExpressionTree(Type.SYMBOL, c))
        else:
            if c == "+":
                z = ExpressionTree(Type.UNION)
                z.right = stack.pop()
                z.left = stack.pop()
            elif c == ".":
                z = ExpressionTree(Type.CONCAT)
                z.right = stack.pop()
                z.left = stack.pop()
            elif c == "*":
                z = ExpressionTree(Type.KLEENE)
                z.left = stack.pop()
            stack.append(z)

    return stack[0]

def inorder(et):
    if et._type == Type.SYMBOL:
        print(et.value)
    elif et._type == Type.CONCAT:
        inorder(et.left)
        print(".")
        inorder(et.right)
    elif et._type == Type.UNION:
        inorder(et.left)
        print("+")
        inorder(et.right)
    elif et._type == Type.KLEENE:
```

```
        inorder(et.left)
        print("*")

def higherPrecedence(a, b):
    p = ["+", ".", "*"]
    return p.index(a) > p.index(b)

def postfix(regexp):
    # adding dot "." between consecutive symbols
    temp = []
    for i in range(len(regexp)):
        if i != 0\
            and (regexp[i-1].isalpha() or regexp[i-1] == ")" or regexp[i-1] == "*")\
            and (regexp[i].isalpha() or regexp[i] == "("):
            temp.append(".")
        temp.append(regexp[i])
    regexp = temp

    stack = []
    output = ""

    for c in regexp:
        if c.isalpha():
            output = output + c
            continue

        if c == ")":
            while len(stack) != 0 and stack[-1] != "(":
                output = output + stack.pop()
            stack.pop()
        elif c == "(":
            stack.append(c)
        elif c == "*":
            output = output + c
```

```
        elif len(stack) == 0 or stack[-1] == "(" or higherPrecedence(c, stack[-1]):
            stack.append(c)
        else:
            while len(stack) != 0 and stack[-1] != "(" and not higherPrecedence(c, stack[-1]):
                output = output + stack.pop()
            stack.append(c)

    while len(stack) != 0:
        output = output + stack.pop()

    return output

class FiniteAutomataState:
    def __init__(self):
        self.next_state = {}

def evalRegex(et):
    # returns equivalent E-NFA for given expression tree (representing a Regular
    # Expression)
    if et._type == Type.SYMBOL:
        return evalRegexSymbol(et)
    elif et._type == Type.CONCAT:
        return evalRegexConcat(et)
    elif et._type == Type.UNION:
        return evalRegexUnion(et)
    elif et._type == Type.KLEENE:
        return evalRegexKleene(et)

def evalRegexSymbol(et):
    start_state = FiniteAutomataState()
    end_state   = FiniteAutomataState()

    start_state.next_state[et.value] = [end_state]
    return start_state, end_state
```

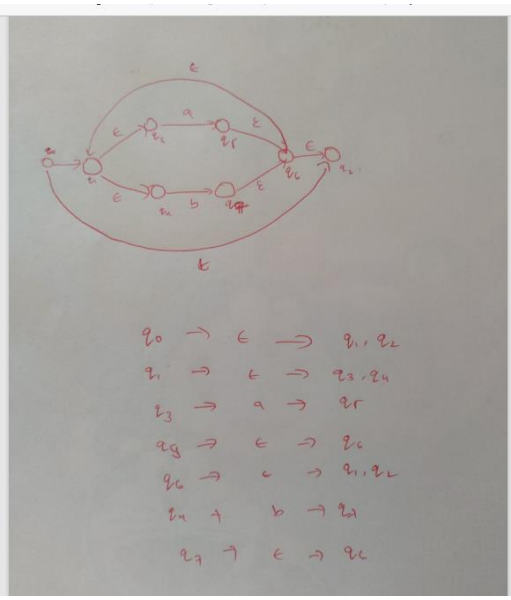
```
def evalRegexConcat(et):
    left_nfa  = evalRegex(et.left)
    right_nfa = evalRegex(et.right)

    left_nfa[1].next_state['epsilon'] = [right_nfa[0]]
    return left_nfa[0], right_nfa[1]

def evalRegexUnion(et):
    start_state = FiniteAutomataState()
    end_state   = FiniteAutomataState()

    up_nfa   = evalRegex(et.left)
    down_nfa = evalRegex(et.right)

    start_state.next_state['epsilon'] = [up_nfa[0], down_nfa[0]]
    up_nfa[1].next_state['epsilon'] = [end_state]
    down_nfa[1].next_state['epsilon'] = [end_state]

    return start_state, end_state

def evalRegexKleene(et):
    start_state = FiniteAutomataState()
    end_state   = FiniteAutomataState()

    sub_nfa = evalRegex(et.left)

    start_state.next_state['epsilon'] = [sub_nfa[0], end_state]
    sub_nfa[1].next_state['epsilon'] = [sub_nfa[0], end_state]

    return start_state, end_state

def printStateTransitions(state, states_done, symbol_table):
    if state in states_done:
```

```
        return

    states_done.append(state)

    for symbol in list(state.next_state):
        line_output = "q" + str(symbol_table[state]) + "\t\t" + symbol + "\t\t\t"
        for ns in state.next_state[symbol]:
            if ns not in symbol_table:
                symbol_table[ns] = 1 + sorted(symbol_table.values())[-1]
            line_output = line_output + "q" + str(symbol_table[ns]) + " "

        print(line_output)

        for ns in state.next_state[symbol]:
            printStateTransitions(ns, states_done, symbol_table)

def printTransitionTable(finite_automata):
    print("State\t\tSymbol\t\t\tNext state")
    printStateTransitions(finite_automata[0], [], {finite_automata[0]:0})



r = input("Enter regex: ")
pr = postfix(r)
et = constructTree(pr)

#inorder(et)

fa = evalRegex(et)
printTransitionTable(fa)
```

INPUT: -

(a+b)*

Space Tree Diagram:



OUTPUT: -

Enter regex: (a+b)*

State	Symbol	Next state
q0	epsilon	q1 q2
q1	epsilon	q3 q4
q3	a	q5
q5	epsilon	q6
q6	epsilon	q1 q2
q4	b	q7
q7	epsilon	q6

RESULT: Regular Expression to NFA converter has been successfully implemented.