Practical 4:

Aim: - Write a program to construct NFA using given regular expression.

Theory: -

Regular Expression

- The language accepted by finite automata can be easily described by simple expressions called Regular Expressions. It is the most effective way to represent any language.
- The languages accepted by some regular expression are referred to as Regular languages.
- A regular expression can also be described as a sequence of pattern that defines a string.
- Regular expressions are used to match character combinations in strings. String searching algorithm used this pattern to find the operations on a string.

For instance:

In a regular expression, x* means zero or more occurrence of x.

It can generate {e, x, xx, xxx, xxxx, ...}

In a regular expression, x^+ means one or more occurrence of x.

It can generate {x, xx, xxx, xxxx, ...}

Operations on Regular Language

The various operations on regular language are:

- **Union:** If L and M are two regular languages then their union L U M is also a union.
  - L U M = {s | s is in L or s is in M}
- **Intersection:** If L and M are two regular languages then their intersection is also an intersection.
  - L ⋂ M = {st | s is in L and t is in M}
- **Kleen closure:** If L is a regular language then its Kleen closure L1* will also be a regular language.
  - L* = Zero or more occurrence of language L.

Example 1:

Write the regular expression for the language accepting all combinations of a's, over the set ∑ = {a}

Solution:

All combinations of a's means a may be zero, single, double and so on. If a is appearing zero times, that means a null string. That is we expect the set of {ε, a, aa, aaa, ...}. So we give a regular expression for this as:

1. R = a*

That is Kleen closure of a.

Example 2:

Write the regular expression for the language accepting all combinations of a's except the null string, over the set ∑ = {a}

Solution:

The regular expression has to be built for the language

1. L = {a, aa, aaa, ...}

This set indicates that there is no null string. So we can denote regular expression as:

$R = a^+$

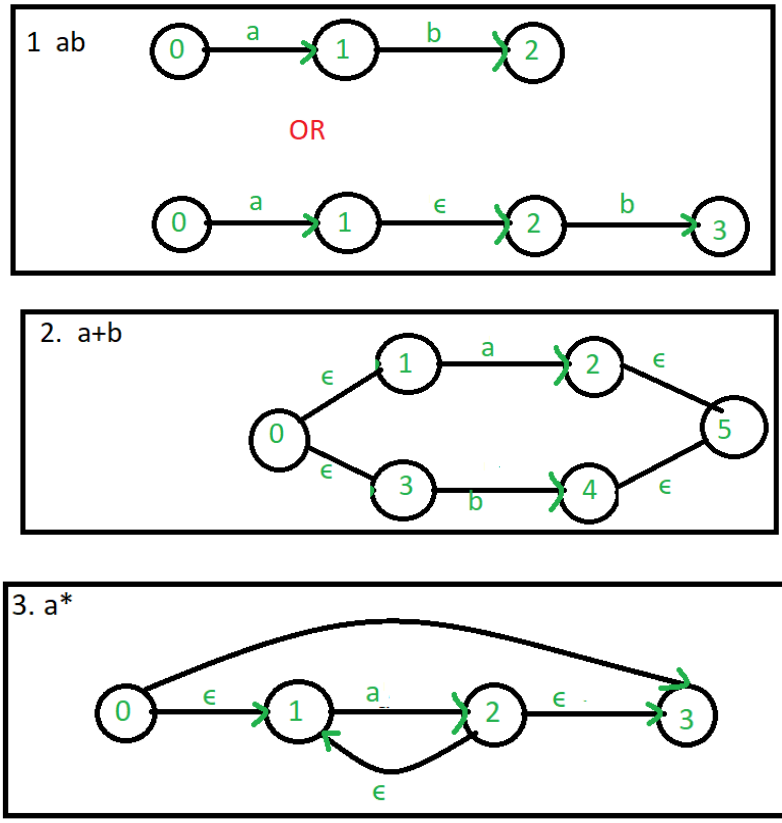
Theory Explanation :

∈-NFA is similar to the NFA but have minor difference by epsilon move. This automaton replaces the transition function with the one that allows the empty string ∈ as a possible input. The transitions without consuming an input symbol are called ∈-transitions.

In the state diagrams, they are usually labeled with the Greek letter ∈. ∈-transitions provide a convenient way of modeling the systems whose current states are not precisely known: i.e., if we are modeling a system and it is not clear whether the current state (after processing some input string) should be q

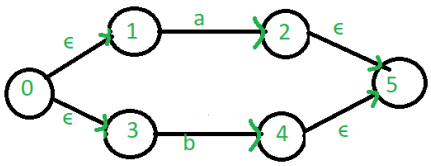
or q', then we can add an ∈-transition between these two states, thus putting the automaton in both states simultaneously.

Common regular expression used in make ∈-NFA:

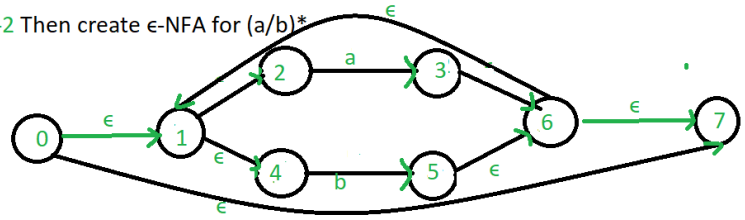


Example: Create a ∈-NFA for regular expression: (a/b)*a

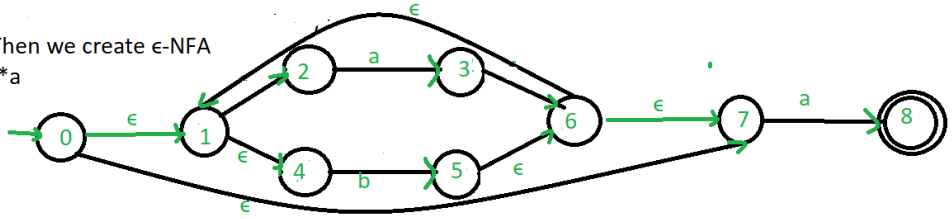
Step-1 First we create ϵ-NFA for (a/b)



Step-2 Then create ϵ-NFA for (a/b)*



Step-3 Then we create ϵ-NFA for(a/b)*a



Algorithm:

1. To draw NFA for a, a/b ,ab ,a* create a routine for each regular expression.
2. For converting from regular expression to NFA, certain transition had been made based on choice of input at the rumtime.
3. Each of the NFA will be displayed is sequential order.

Program:-

nfa.py – consist of main module and nfa class structure.

```
from nfa_utils import *
import sys
import time

class NFA:
    """Class representing a non-deterministic finite automaton"""

    def __init__(self):
        """Creates a blank NFA"""

        # all NFAs have a single initial state by default
        self.alphabet = set()
        self.states = {0}
        self.transition_function = {}
        self.accept_states = set()

        # set of states that the NFA is currently in
        self.in_states = {0}

    def add_state(self, state, accepts=False):
        self.states.add(state)

        if accepts:
            self.accept_states.add(state)

    def add_transition(self, from_state, symbol, to_states):
        self.transition_function[(from_state, symbol)] = to_states

        if symbol != "":
            self.alphabet.add(symbol)


    def is_accepting(self):
        # accepts if we are in ANY accept states
        # ie. if in_states and accept_states share any states in common
        return len(self.in_states & self.accept_states) > 0


    def __str__(self):
        return "NFA:\n" \
               "Alphabet: {}\n" \
               "States: {}\n" \
               "Transition Function: \n {} \n" \
               "Accept States: {}\n" \
               "In states: {}\n" \
               "Accepting: {}\n"\
```

```
            .format(self.alphabet,
                    self.states,
                    self.transition_function,
                    self.accept_states,
                    self.in_states,
                    "Yes" if self.is_accepting() else "No")

if __name__ == "__main__":

    # regular expression string to compare against provided input
    regex = None
    regex_nfa = None
    # last line of user input read from the command line
    line_read = ""

    # read in line of user input
    line_read = input("Enter the regex pattern \n>")
    # make a lowercase copy of the input for case insensitive comparisons
    regex = line_read.lower()
    print("Given Regex pattern:", regex, "\n")
    print("Building NFA ")
    start_time = time.time()

    # turn regular expression string into an NFA object
    regex_nfa = get_regex_nfa(regex)

    finish_time = time.time()
    ms_taken = (finish_time - start_time) * 1000

    print("\nBuilt NFA in {:.3f} ms.\n".format(ms_taken))
    print(regex_nfa)
```

nfa_utils.py - consist of code for handling regular expression

```
from nfa import NFA
import copy

def get_single_symbol_regex(symbol):
    """ Returns an NFA that recognizes a single symbol """
    nfa = NFA()
    nfa.add_state(1, True)
    nfa.add_transition(0, symbol, {1})
    return nfa

def shift(nfa, inc):
    """Increases the value of all states (including accept states and transition function etc)
    of a given NFA bya given value.
    This is useful for merging NFAs, to prevent overlapping states
    """
    # update NFA states
    new_states = set()
    for state in nfa.states:
        new_states.add(state + inc)
    nfa.states = new_states

    # update NFA accept states
    new_accept_states = set()
    for state in nfa.accept_states:
        new_accept_states.add(state + inc)
    nfa.accept_states = new_accept_states

    # update NFA transition function
    new_transition_function = {}
    for pair in nfa.transition_function:
        to_set = nfa.transition_function[pair]
        new_to_set = set()
```

```
        for state in to_set:
            new_to_set.add(state + inc)

        new_key = (pair[0] + inc, pair[1])
        new_transition_function[new_key] = new_to_set

    nfa.transition_function = new_transition_function

def merge(a, b):
    """Merges two NFAs into one by combining their states and transition function"""
    a.accept_states = b.accept_states
    a.states |= b.states
    a.transition_function.update(b.transition_function)
    a.alphabet |= b.alphabet

def get_concat(a, b):
    """ Concatenates two NFAs, ie. the dot operator """
    # number to add to each b state number
    # this is to ensure each NFA has separate number ranges for their states
    # one state overlaps; this is the state that connects a and b
    add = max(a.states)
    # shift b's state/accept states/transition function, etc.
    shift(b, add)
    # merge b into a
    merge(a, b)
    return a

def get_union(a, b):
    """Returns the resulting union of two NFAs (the '|' operator)"""
    # create a base NFA for the union
    nfa = NFA()

    # clear a and b's accept states
    a.accept_states = set()
    b.accept_states = set()

    # merge a into the overall NFA
    shift(a, 1)
    merge(nfa, a)

    # merge b into the overall NFA
    shift(b, max(nfa.states) + 1)
    merge(nfa, b)

    # add an empty string transition from the initial state to the start of a and b
    # (so that the NFA starts in the start of a and b at the same time)
    nfa.add_transition(0, "", {1, min(b.states)})

    # add an accept state at the end so if either a or b runs through,
    # this NFA accepts
    new_accept = max(nfa.states) + 1
    nfa.add_state(new_accept, True)
    nfa.add_transition(max(a.states), "", {new_accept})
    nfa.add_transition(max(b.states), "", {new_accept})

    return nfa


def get_kleene_star_nfa(nfa):
    """
    Wraps an NFA inside a kleene star expression
    (NFA passed in recognizes 0, 1 or many of the strings it originally recognized)
    """
    # clear old accept state
    nfa.accept_states = {}

    # shift NFA by 1 and insert new initial state
    shift(nfa, 1)
    nfa.add_state(0)
```

```
    # add new ending accept state
    last_state = max(nfa.states)
    new_accept = last_state + 1
    nfa.add_state(new_accept, True)
    nfa.add_transition(last_state, "", {new_accept})

    # add remaining empty string transitions
    nfa.add_transition(0, "", {1, new_accept})
    nfa.add_transition(last_state, "", {0})

    return nfa

def get_one_or_more_of_nfa(nfa):
    """
    Wraps an NFA inside the "one or more of" operator (plus symbol)
    Simply combines the concatenation operator and the kleene star operator.
    """

    # must make a copy of the nfa,
    # these functions operate on the nfa passed in, they do not make a copy
    return get_concat(copy.deepcopy(nfa), get_kleene_star_nfa(nfa))

def get_zero_or_one_of_nfa(nfa):
    """
    Wraps an NFA inside the "zero or one of" operator (question mark symbol)
    Simply uses the union operator, with one path for the empty string, and the other path
    for the NFA being wrapped.
    """

    return get_union(get_single_symbol_regex(""), nfa)

#first call
def get_regex_nfa(regex, indent=""):
    """Recursively builds an NFA based on the given regex string"""
    print("{0}{0}({1})".format(indent, regex))
    indent += " " * 2

    # special symbols: +*.| (in order of precedence highest to lowest, symbols coming before
that
    # union operator
    bar_pos = regex.find("|")
    if bar_pos != -1:
        # there is a bar in the string; union both sides
        # (uses the leftmost bar if there are more than 1)
        return get_union(
            get_regex_nfa(regex[:bar_pos], indent),
            get_regex_nfa(regex[bar_pos + 1:], indent)
        )

    # concatenation operator
    dot_pos = regex.find(".")
    if dot_pos != -1:
        # there is a dot in the string; concatenate both sides
        # (uses the leftmost dot if there are more than 1)
        return get_concat(
            get_regex_nfa(regex[:dot_pos], indent),
            get_regex_nfa(regex[dot_pos + 1:], indent)
        )

    # kleene star operator
    star_pos = regex.find("*")
    if star_pos != -1:
        # there is an asterisk in the string; wrap everything before it in a kleene star
expression
        # (uses the leftmost dot if there are more than 1)
        star_part = regex[:star_pos]
        trailing_part = regex[star_pos + 1:]
        kleene_nfa = get_kleene_star_nfa(get_regex_nfa(star_part, indent))

        if len(trailing_part) > 0:
```

```
            return get_concat(
                kleene_nfa,
                get_regex_nfa(trailing_part, indent)
            )
        else:
            return kleene_nfa

    # "one or more of" operator ('+' symbol)
    plus_pos = regex.find("+")
    if plus_pos != -1:
        # there is a plus in the string; wrap everything before it in the "one or more of"
expression
        # (uses the leftmost plus if there are more than 1)

        plus_part = regex[:plus_pos]
        trailing_part = regex[plus_pos + 1:]
        plus_nfa = get_one_or_more_of_nfa(get_regex_nfa(plus_part, indent))

        if len(trailing_part) > 0:
            return get_concat(
                plus_nfa,
                get_regex_nfa(trailing_part, indent)
            )
        else:
            return plus_nfa

    # "zero or one of" operator ('?' symbol)
    qmark_pos = regex.find("?")
    if qmark_pos != -1:
        # there is a question mark in the string; wrap everything before it in the "zero or one
of" expression
        # (uses the leftmost question mark if there are more than 1)

        leading_part = regex[:qmark_pos]
        trailing_part = regex[qmark_pos + 1:]
        zero_or_one_of_nfa = get_zero_or_one_of_nfa(get_regex_nfa(leading_part, indent))

        if len(trailing_part) > 0:
            return get_concat(
                zero_or_one_of_nfa,
                get_regex_nfa(trailing_part, indent)
            )
        else:
            return zero_or_one_of_nfa

    # no special symbols left at this point

    if len(regex) == 0:
        # base case: empty nfa for empty regex
        return NFA()
    elif len(regex) == 1:
        # base case: single symbol is directly turned into an NFA
        return get_single_symbol_regex(regex)
    else:
        # multiple characters left; apply implicit concatenation between the first character
        # and the remaining characters
        return get_concat(
            get_regex_nfa(regex[0], indent),
            get_regex_nfa(regex[1:], indent)
        )
```

Output:-

```
Enter the regex pattern
>a.b.c.e.d.f
Given Regex pattern: a.b.c.e.d.f

Building NFA
(a.b.c.e.d.f)
    (a)
    (b.c.e.d.f)
        (b)
        (c.e.d.f)
            (c)
            (e.d.f)
                (e)
                (d.f)
                    (d)
                    (f)

Built NFA in 139.255 ms.

NFA:
Alphabet: {'f', 'e', 'd', 'c', 'a', 'b'}
States: {0, 1, 2, 3, 4, 5, 6}
Transition Function:
 {(0, 'a'): {1}, (1, 'b'): {2}, (2, 'c'): {3}, (3, 'e'): {4}, (4, 'd'): {5}, (5, 'f'): {6}}
Accept States: {6}
In states: {0}
Accepting: No
```

```
Enter the regex pattern
>p|THON
Given Regex pattern: p|thon

Building NFA
(p|thon)
    (p)
    (thon)
        (t)
        (hon)
            (h)
            (on)
                (o)
                (n)

Built NFA in 109.411 ms.

NFA:
Alphabet: {'n', 'p', 'o', 't', 'h'}
States: {0, 1, 2, 3, 4, 5, 6, 7, 8}
Transition Function:
 {(1, 'p'): {2}, (3, 't'): {4}, (4, 'h'): {5}, (5, 'o'): {6}, (6, 'n'): {7}, (0, ''): {1, 3}, (2, ''): {8}, (7, ''): {8}}
Accept States: {8}
In states: {0}
Accepting: No
```

```
Enter the regex pattern
>p?ab.c|thon
Given Regex pattern: p?ab.c|thon

Building NFA
(p?ab.c|thon)
    (p?ab.c)
        (p?ab)
            (p)
            (ab)
                (a)
                (b)
        (c)
    (thon)
        (t)
        (hon)
            (h)
            (on)
                (o)
                (n)

Built NFA in 187.555 ms.

NFA:
Alphabet: {'t', 'o', 'c', 'p', 'b', 'h', 'n', 'a'}
States: {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15}
Transition Function:
 {(2, ''): {3}, (4, 'p'): {5}, (1, ''): {2, 4}, (3, ''): {6}, (5, ''): {6}, (6,
'a'): {7}, (7, 'b'): {8}, (8, 'c'): {9}, (10, 't'): {11}, (11, 'h'): {12}, (12,
'o'): {13}, (13, 'n'): {14}, (0, ''): {1, 10}, (9, ''): {15}, (14, ''): {15}}
Accept States: {15}
In states: {0}
Accepting: No
```

Conclusion:-

The given program Successfully makes a NFA state table from given regular expression.

References:-

https://www.javatpoint.com/automata-regular-expression

https://www.geeksforgeeks.org/regular-expression-to-nfa/

https://userpages.umbc.edu/~squire/cs451_l7.html