Experiment no – 04

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

Algorithm:

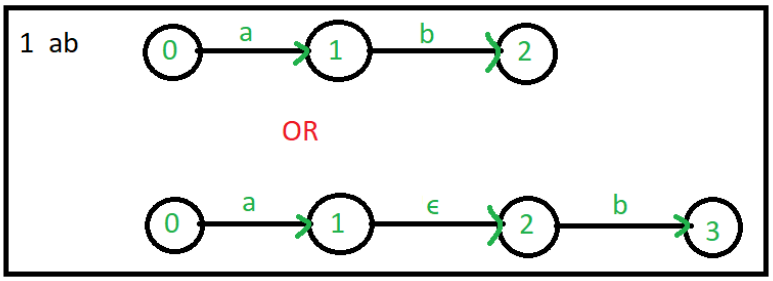
1. Create a menu for getting four regular expressions input as choice.
2. To draw NFA for a, a/b ,ab ,a* create a routine for each regular expression.
3. For converting from regular expression to NFA, certain transition had been made based on choice of input at the rumtime.
4. Each of the NFA will be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

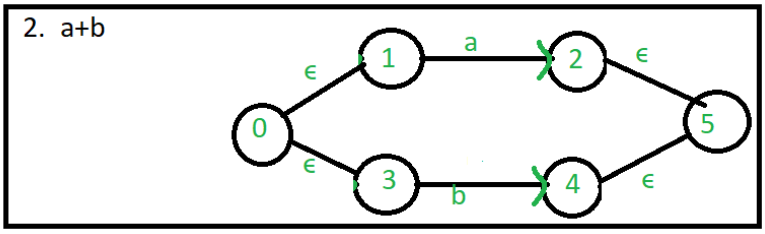
Theory Explanation :

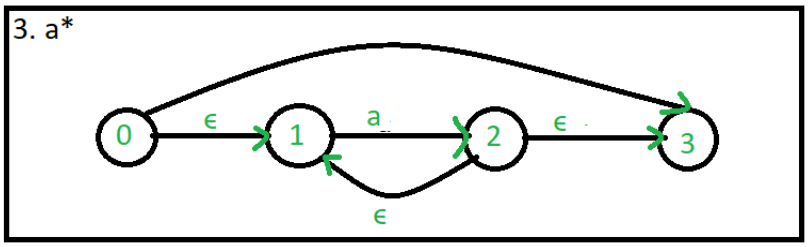
∈-NFA is similar to the NFA but have minor difference by epsilon move. This automaton replaces the transition function with the one that allows the empty string ∈ as a possible input. The transitions without consuming an input symbol are called ∈-transitions.

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:

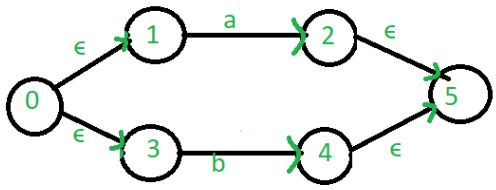




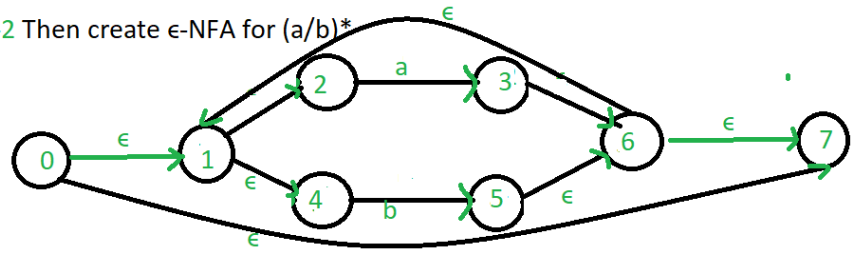


Example: Create a ∈-NFA for regular expression: (a/b)*a

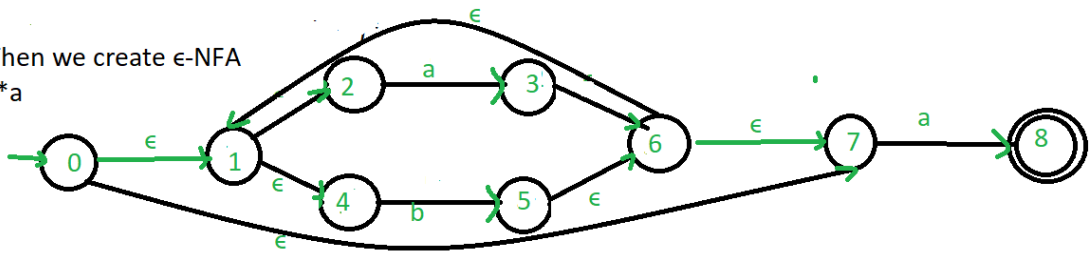
Step-1 First we create ϵ-NFA for (a/b)



Step-2 Then create ϵ-NFA for (a/b)*



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



Program:

1. **Main.Py**

```
#Program to Construct NFA using REGEX


import sys
import nfa_utils
import time

# print the intro text block
with open("intro.dat") as intro_file:
    print(intro_file.read())

# 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 = ""

# continuously parse and process user input
while True:
    # read in line of user input
    line_read = input("> ")
    # make a lowercase copy of the input for case insensitive comparisons
    line_read_lower = line_read.lower()

    if line_read_lower == "exit":
        # exit the program
        print("\nExiting...")
        sys.exit()

    if line_read_lower.startswith("regex="):
        # user wants to set the regex to a string they've provided
        regex = line_read[6:]
        print("New regex pattern:", regex, "\n")
        start_time = time.time()
        # turn regular expression string into an NFA object
        regex_nfa = nfa_utils.get_regex_nfa(regex)
        regex_nfa.reset()
        finish_time = time.time()
        ms_taken = (finish_time - start_time) * 1000

        print("\nBuilt NFA in {:.3f} ms.\n".format(ms_taken))
        print(regex_nfa)
    else:
        # assume the user intends to test this entered string against the
regex
        if regex_nfa is None:
            # regex has not yet been set
            print("Please supply a regular expression string first, with
regex=(regex here)")
        else:
            start_time = time.time()
            # feed input string into NFA
            regex_nfa.feed_symbols(line_read, return_if_dies=True)
            accepts = regex_nfa.is_accepting()
            finish_time = time.time()
            ms_taken = (finish_time - start_time) * 1000
```

```
            print("String was {} by NFA"
                  .format("ACCEPTED" if accepts else "REJECTED"))

            print("Calculated in {:.3f} ms.".format(ms_taken))

            # print(regex_nfa)
            regex_nfa.reset()

    # print a new line for aesthetics
    print()
```

2. NFA.py

```
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 feed_symbol(self, symbol):
        """
        Feeds a symbol into the NFA, calculating which states the
        NFA is now in, based on which states it used to be in
        """

        # a dead NFA will not have any transitions after a symbol is fed
in
        if self.is_dead():
            return

        new_states = set()

        # process each old state in turn
        for state in self.in_states:
            pair = (state, symbol)

            # check for a legal transition from the old state to a
```

```
            # new state, based on what symbol was fed in
            if pair in self.transition_function:
                # add the corresponding new state to the updated states
list
                new_states |= self.transition_function[pair]

        self.in_states = new_states

        # feed the empty string through the nfa
        self.feed_empty()

    def feed_symbols(self, symbols, return_if_dies=False):
        """
        Feeds an iterable into the NFAs feed_symbol method

        :param symbols: Iterable of symbols to feed through the NFA
        :param return_if_dies: If true, ignore any further symbols after
the NFA dies (for efficiency),
        since a dead NFA will never accept, regardless of any further
input.
        """

        for symbol in symbols:
            self.feed_symbol(symbol)

            if return_if_dies and self.is_dead():
                # NFA is dead; feeding further symbols will not change
the NFA's state
                return

    def feed_empty(self):
        """
        Continuously feeds empty strings into the NFA until they fail
        to cause any further state transitions
        """

        # a dead NFA will not have any empty string transitions
        if self.is_dead():
            return

        old_states_len = None
        # set of states that will be fed the empty string on the next
pass
        unproc_states = self.in_states
        first_run = True

        # keep feeding the empty string until no more new states are
transitioned into
        while first_run or len(self.in_states) > old_states_len:
            old_states_len = len(self.in_states)
            # set of new states transitioned into after the empty string
was fed
            new_states = set()

            # process each state in turn
            for state in unproc_states:
                pair = (state, "")

                # check if this state has a transition using the empty
string
                # to another state
```

```
                if pair in self.transition_function:
                    # add the new state to a set to be added to
self.in_states later
                    new_states |= self.transition_function[pair]

            # merge new states back into "in" states
            self.in_states |= new_states
            # all new states discovered will be fed the empty string on
the next pass
            unproc_states = new_states
            first_run = False

    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 is_dead(self):
        """
        Returns true if the NFA is not in ANY states.
        A "dead" NFA can never be in any states again.
        """
        return len(self.in_states) == 0

    def reset(self):
        """
        Resets the NFA by putting it back to it's initial state,
        and feeding the empty string through it
        """
        self.in_states = {0}
        self.feed_empty()

    def __str__(self):
        """
        String representation of this NFA.
        Useful for debugging.
        """
        return "NFA:\n" \
               "Alphabet: {}\n" \
               "States: {}\n" \
               "Transition Function: {}\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")

    def __eq__(self, other):
        """
        Checks if two NFAs are equal. Used for testing.

        Tests if they are structurally the same; does NOT check if they
are in the same states.

        Also ignores alphabets.
        """
        return self.states == other.states \
```

```
            and self.transition_function == other.transition_function \
            and self.accept_states == other.accept_states
```

3. NFA.UTILS.py

```
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
```

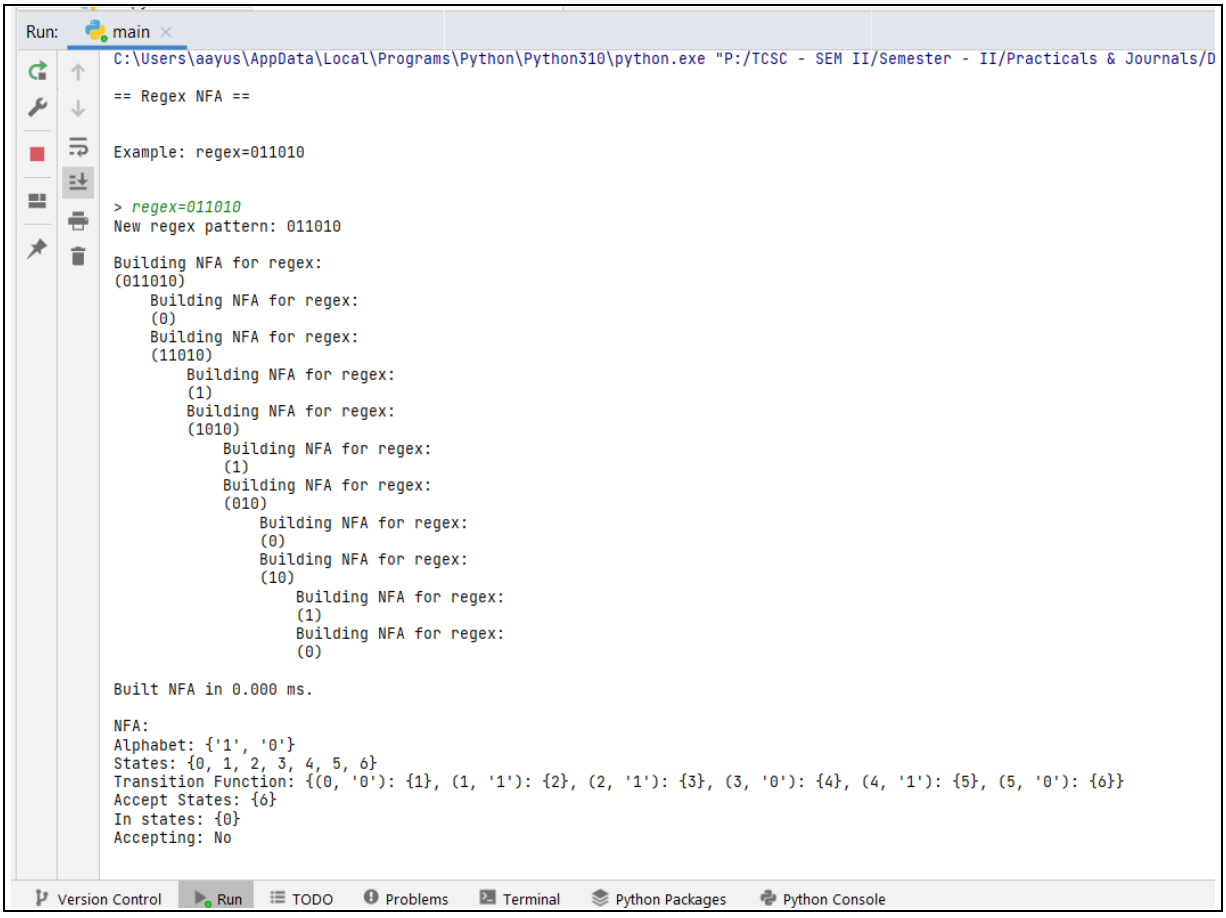
```
    a.transition_function.update(b.transition_function)
    a.alphabet |= b.alphabet


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)

def get_regex_nfa(regex, indent=""):
    """Recursively builds an NFA based on the given regex string"""

    print("{0}Building NFA for regex:\n{0}({1})".format(indent, regex))
    indent += " " * 4

    # 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:



Conclusion : Successfully construct NFA using given regular expression.

Reference:

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

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