

CB310 Event report

For

Converting Regular Expression to -NFA

Prepared by

Akshata P Choukimath

01JST19CB002

Harsha Vishweshwara Bhat

01JST19CB018

Insha Suroor

01JST19CB020

Organization

JSS Science And Technology University, Mysuru

INDEX

**1. Problem Statement**

Convert a regular Expression to NFA

**2. Introduction and solution the problem**

**(Intro)**

class Type:

    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):          #create a tree for the regular expression

    stack = []

    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):                      #Order of regularExpression

    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):             #Set precedence of operators

    p = ["+", ".", "\*"]

    return p.index(a) > p.index(b)

def postfix(regexp):                          #do postfix of the regular expression

    # 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:                                    #new Class for Finite Automata

    def \_\_init\_\_(self):

        self.next\_state = {}

def evalRegex(et):                                            #evaluating the regex

    # 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):                            #print the transition 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: ")

print("\nstate (symbol)  --->  next state")

pr = postfix(r)                                 #postfix

et = constructTree(pr)                          #tree construction

fa = evalRegex(et)                              #evaluate regular expression

printTransitionTable(fa)                        #print transition table

print("\n'q0' is the starting state")

print("\nThe last Next state is the final state(s) ")

**The complete source code (assignment) for this problem statement is “** [**https://colab.research.google.com/drive/1ZKoaBpjbbW\_tu7vKUWVNv4WhOKtCnUTQ#scrollTo=mw5aX5o5t8vL**](https://colab.research.google.com/drive/1ZKoaBpjbbW_tu7vKUWVNv4WhOKtCnUTQ#scrollTo=mw5aX5o5t8vL) **“**

**3. Steps or algorithm**

1. Creating class Type for the operators and class Expression tree for regular expression.

i.e

. class Type, class ExpressionTree

1. Constructing a regular expression Tree

i.e def constructTree(regexp)

1. Analyzing the operator Union, Kleene, Concatenation

def inorder(et)

1. Setting the precedence of operators.

def higherPrecedence(a, b)

1. Creating another class for converting to a Finite Automata State

class FiniteAutomataState

1. Evaluating the regular expression : This has a set of functions to evaluate regex symbols
2. Functions to print the Transition table.

def printStateTransitions(state, states\_done, symbol\_table) and def printTransitionTable(finite\_automata)