Name: Adwait Hegde
Roll No: 2019130019
TE Comp (Batch-A)

EXPERIMENT 1

Aim:

To convert a Regular Expression to optimized DFA.

Theory:

Convert Regular Expression to DFA

- Uses augmented regular expression *r#*.
- Important states of NFA correspond to positions in regular expression that hold symbols of the alphabet.
- Regular expression is represented as syntax tree where interior nodes correspond to operators representing union, concatenation and closure operations.
- Leaf nodes corresponds to the input symbols
- Construct DFA directly from a regular expression by computing the functions *nullable(n), firstpos(n), lastpos(n) andfollowpos(i)* from the syntax tree.
    - **nullable** *(n):* Is true for * node and node labeled with ε. For other nodes it is false.
    - **firstpos** *(n):* Set of positions at node *ti* that corresponds to the first symbol of the sub-expression rooted at *n*.
    - **lastpos** *(n):* Set of positions at node *ti* that corresponds to the last symbol of the sub-expression rooted at *n*.
    - **followpos** (i): Set of positions that follows given position by matching the first or last symbol of a string generated by sub-expression of the given regular expression.

Rules for computing nullable, firstpos and lastpos

Node n	nullable *(n)*	firstpos *(n)*	lastpos *(n)*
A leaf labeled ε	True	Ø	Ø
A leaf with position i	False	{i}	{i}
An *or* node *n* = c_1 | c_2	Nullable (c_1) or	firstpos (c_1) U	lastpos (c_1) U

	Nullable (c_2)	firstpos (c_2)	lastpos (c_2)

A cat node n = c_1c_2	Nullable (c_1) and
	
	Nullable (c_2)
		If (Nullable (c_1))
		
		firstpos (c_1) U
		
		firstpos (c_2)
		
		else
		
		firstpos (c_1)
			If (Nullable (c_2))
			
			lastpos (c_1) U
			
			lastpos (c_2)
			
			else
			
			lastpos (c_1)

A star node n = c_{1*}	True	firstpos (c_1)	lastpos (c_1)

Computation of followpos

The position of regular expression can follow another in the following ways:

- If n is a cat node with left child c_1 and right child c_2, then for every position i in *lastpos*(c_1), all positions in *firstpos*(c_2) are in *followpos(i)*.
- For cat node, for each position i in *lastpos* of its *left child*, the *firstpos* of its *right child* will be in *followpos(i)*.
- If n is a star node and i is a position in *lastpos(n)*, then all positions in *firstpos(n)* are in *followpos(i)*.
- For star node, the *firstpos* of that node is in *f ollowpos* of all positions in *lastpos* of that node.

Implementation

Code:

```
# Adwait Hegde
# 2019130019
# TE Comps

leaf_no = 0
leaf_array = []
follow_pos = []
print()

# input a regular expression
re = input(" [+] Enter the regular expression: ")

# convert to augmented regular expression
```

```
are = '('
for e in re:
    if are[-1] in [')'] and e not in [')','|','*']:
        are = are + '.' + e
    elif are[-1] not in ['(',')','|'] and e not in [')','|','*']:
        are = are + '.' + e
    else:
        are = are + e
are = are[1:]+".#"
print('\n - Augmented regular expression: ' + are + '\n')

# Class to construct a syntax tree for the ARE
class SyntaxTree():
    content = '.'
    nullable = False
    first_pos = set()
    last_pos = set()
    leaf_number = int()
    left = None
    right = None

    def __init__(self, content, leaf_number, left, right):
        self.content = content
        self.leaf_number = leaf_number
        self.left = left
        self.right = right
        if content in ['*']:
            self.nullable = True

    def __str__(self) -> str:
        return self.content + ' ' + str(self.leaf_number) + ' ' +
str(self.nullable) + ' ' + str(self.first_pos) + ' ' + str(self.last_pos)

    def update_nullable(self):
        if self.content == '|':
            self.nullable = bool(self.right.nullable) or bool(self.left.nullable)
        elif self.content == '.':
            self.nullable = bool(self.right.nullable) and
bool(self.left.nullable)

    def update_first_pos(self):
        if self.content == '*':
            self.first_pos = self.left.first_pos
        elif self.content == '|':
            lfp = self.left.first_pos
```

```
            rfp = self.right.first_pos
            self.first_pos = lfp | rfp
        elif self.content == '.':
            ln = self.left.nullable
            lfp = self.left.first_pos
            rfp = self.right.first_pos
            if ln:
                self.first_pos = lfp | rfp
            else:
                self.first_pos = lfp
        else:
            self.first_pos = {self.leaf_number}

    def update_last_pos(self):
        if self.content == '*':
            self.last_pos = self.left.last_pos
        elif self.content == '|':
            llp = self.left.last_pos
            rlp = self.right.last_pos
            self.last_pos = llp | rlp
        elif self.content == '.':
            rn = self.right.nullable
            llp = self.left.last_pos
            rlp = self.right.last_pos
            if rn:
                self.last_pos = llp | rlp
            else:
                self.last_pos = rlp

        else:
            self.last_pos = {self.leaf_number}

    def update_nfl(self):
        if self.left:
            self.left.update_nfl()
        if self.right:
            self.right.update_nfl()
        self.update_nullable()
        self.update_first_pos()
        self.update_last_pos()

    def print_tree(self):
        if self.left:
            self.left.print_tree()
        print(self)
```

```
        if self.right:
            self.right.print_tree()


# nullable, firstpos, lastpos, followpos
# re -> dfa

# function to create a syntax tree and return its root node
def create_syntax_tree(are):
    print(are)
    global leaf_no, leaf_array, follow_pos
    if len(are) == 1:
        leaf_no+=1
        head = SyntaxTree(are,leaf_no,None,None)
        leaf_array.append(head)
        follow_pos.append(set())
        return head

    stack = 0
    flag = True
    for e in are:
        if e == '(':
            stack += 1
        if e ==')':
            stack -= 1

        if (e == '.' or e == '|') and stack == 0:
            flag = False

    if flag:
        re = are
        if re[-1]=='*':
            if re[0] ==='(':
                left = create_syntax_tree(re[1:-2])
            else:
                left = create_syntax_tree(re[:-1])

            head = SyntaxTree('*',-1,left,None)
            return head
        if re[0]=='(':
            return create_syntax_tree(re[1:-1])

    stack = 0
    temp = ''
    left = None
```

```
    right = None
    prev = None
    root = None
    for e in are+'.':
        if e == '(' :
            stack += 1
        if e ==')':
            stack -= 1

        if (e == '.' or e == '|') and stack == 0:
            if left == None:
                left = create_syntax_tree(temp)
            elif right == None:
                right = create_syntax_tree(temp)
                root = SyntaxTree(prev,-1,left,right)
            else:
                left = root
                right = create_syntax_tree(temp)
                root = SyntaxTree(prev,-1,left,right)
            prev = e
            temp = ''
        else:
            temp = temp + e
    return root


#  calculate follow pos of the syntax tree
def caluculate_follow_pos(head):
    if head:
        global follow_pos
        caluculate_follow_pos(head.left)

        if head.content == '*':
            for i in head.last_pos:
                follow_pos[i-1] = follow_pos[i-1] | head.first_pos

        if head.content == '.':
            for i in head.left.last_pos:
                follow_pos[i-1] = follow_pos[i-1] | head.right.first_pos

        caluculate_follow_pos(head.right)



head = create_syntax_tree(are)
```

```
head.update_nfl()
print("The tree is:")
head.print_tree()
print("-------")
caluculate_follow_pos(head)
print("  FOLLOW-POS TABLE  ")
for i, leaf in enumerate(leaf_array):
    print(leaf.content , '\t' , leaf.leaf_number , '\t' , follow_pos[i])
print()

# to get the unique terminals
terminals = []
for i in leaf_array:
    terminal = i.content
    if terminal == '#':
        continue
    if terminal not in terminals:
        terminals.append(terminal)


# Making of the DFA table
states = [head.first_pos]
table = []
ptr = 0

while ptr<len(states):
    sub_table = []
    for terminal in terminals:
        cur_state = set()
        for i in states[ptr]:
            if leaf_array[i-1].content==terminal:
                cur_state = cur_state.union(follow_pos[i-1])
        if cur_state not in states:
            states.append(cur_state)
        sub_table.append(states.index(cur_state))
    table.append(sub_table)
    ptr+=1


# Printing the final DFA table
A = ord('A')

print("\n  Minimized DFA TABLE  ")
for i in [''] + terminals:
    print(i, end='\t')
```

```
print("\n---------"+"------"*len(terminals))

for id, row in enumerate(table):
    print(chr(A+id), end='\t')
    for column in row:
        print(chr(A+column), end='\t')
    print()
print()
```

Output:

```
 [+] Enter the regular expression: (a|b)*ab

 - Augmented regular expression: (a|b)*.a.b.#

  FOLLOW-POS TABLE
a        1       {1, 2, 3}
b        2       {1, 2, 3}
a        3       {4}
b        4       {5}
#        5       set()


  Minimized DFA TABLE
        a       b
---------------------
A       B       A
B       B       C
C       B       A
```

```
 [+] Enter the regular expression: (a|b)ab(a|b)*

 - Augmented regular expression: (a|b).a.b.(a|b)*.#

  FOLLOW-POS TABLE
 a       1       {3}
 b       2       {3}
 a       3       {4}
 b       4       {5, 6, 7}
 a       5       {5, 6, 7}
 b       6       {5, 6, 7}
 #       7       set()


  Minimized DFA TABLE
        a       b
--------------------
A       B       B
B       C       D
C       D       E
D       D       D
E       E       E
```

Conclusion:

From the above experiment, I was able to implement code and programatically convert a Regular Expression to optimized DFA

Ref.:
https://ecomputernotes.com/compiler-design/convert-regular-expression-to-dfa
https://www.youtube.com/watch?v=rGRSiPSmhwE