



University of Mumbai

JOURNAL

PSDS101: PROGRAMMING PARADIGM

(MSc. Computer Science with specialization in Data Science 2021)

Submitted by

Sumon Singh

Roll No. 16

1. Write a Program in python to Regular Expression to NFA

Source Code :

```
inp = "((e+a).b*)*"#input("")
"""

Give your input in the above variable
a and b are the only terminals accepted by this script
e denotes epsilon
. is used for "and" operation Eg. ab = a.b
+ is used for "or" operation Eg. a|b = a+b
* is the Kleene's Closure operator. You can give star operator after any closing brackets and
terminals
"""

_="-"

start = 1 # denotes start of e-nfa table
end = 1 # denotes end of our table which is initially same as start
cur = 1 # denotes current position of our pointer
# this is initial e-nfa table with only one state which is start and end both
table = [["state","epsilon","a","b"],
[1,_,_,_]]

def print_t(table):
"""

This function prints the e-nfa table
"""

i = table[0]
print(f'{i[0]: <10}'+f'| {i[1]: <10}'+f'| {i[2]: <10}'+f'| {i[3]: <10}')
print("-"*46)
for i in table[1:]:
    try:
        x = " ".join([str(j) for j in i[1]])
    except:
        x = ""
    try:
        y = " ".join([str(j) for j in i[2]])
    except:
        y = ""
    try:
        z = " ".join([str(j) for j in i[3]])
    except:
        z = ""
    print(f'{i[0]: <10}'+f'| {x: <10}'+f'| {y: <10}'+f'| {z: <10}')
```

```

def e_(cur,ed=end):
    """
    this fuction adds epsilon to the table
    """
    temp = table[cur]
    try:
        table[cur] = [cur,temp[1].append(cur+1),temp[2],temp[3]]
    except:
        table[cur] = [cur,[cur+1],temp[2],temp[3]]
    try:
        nv = table([cur+1])
    except:
        table.append([ed+1,_,_,_])
    ed+=1
    return ed

def a_(cur,ed=end):
    temp = table[cur]
    try:
        table[cur] = [cur,temp[1],temp[2].append(cur+1),temp[3]]
    except:
        table[cur] = [cur,temp[1],[cur+1],temp[3]]
    try:
        nv = table([cur+1])
    except:
        table.append([ed+1,_,_,_])
    ed+=1
    return ed

def b_(cur,ed=end):
    temp = table[cur]
    try:
        table[cur] = [cur,temp[1],temp[2],temp[3].append(cur+1)]
    except:
        table[cur] = [cur,temp[1],temp[2],[cur+1]]
    try:
        nv = table([cur+1])
    except:
        table.append([ed+1,_,_,_])
    ed+=1
    return ed

def or_b(cur,ed=end):
    temp = table[cur]
    try:

```

```

table[cur] = [cur,temp[1],temp[2],temp[3].append(cur+1)]
except:
table[cur] = [cur,temp[1],temp[2],[cur+1]]

def or_a(cur,ed=end):
temp = table[cur]
try:
table[cur] = [cur,temp[1],temp[2].append(cur+1),temp[3]]
except:
table[cur] = [cur,temp[1],[cur+1],temp[3]]

def and_a(cur,ed=end):
cur+=1
temp = table[cur]
try:
table[cur] = [cur,temp[1],temp[2].append(cur+1),temp[3]]
except:
table[cur] = [cur,temp[1],[cur+1],temp[3]]
try:
nv = table([cur+1])
except:
table.append([cur+1,_,_,_])
ed+=1
return cur,ed

def and_b(cur,ed=end):
cur+=1
temp = table[cur]
try:
table[cur] = [cur,temp[1],temp[2],temp[3].append(cur+1)]
except:
table[cur] = [cur,temp[1],temp[2],[cur+1]]
try:
nv = table([cur+1])
except:
table.append([cur+1,_,_,_])
ed+=1
return cur,ed

def star(cur,ed=end):
table.append([ed+1,_,_,_])
table.append([ed+2,_,_,_])
ed+=2
for i in range(cur,ed):
temp = [table[ed-i+cur][0]]+table[ed-i+cur-1][1:4]
for j in [1,2,3]:

```

```

try:
temp[j] = [x+1 for x in table[ed-i+cur-1][j]]
except:
pass
table[ed-i+cur] = temp
table[cur]=[cur,_,_,_]

temp = table[cur]
try:
table[cur] = [temp[0],temp[1]+[cur+1,ed],temp[2],temp[3]]
except:
table[cur] = [temp[0],[cur+1,ed],temp[2],temp[3]]

temp = table[ed-1]
try:
table[ed-1] = [temp[0],temp[1]+[cur+1,ed],temp[2],temp[3]]
except:
table[ed-1] = [temp[0],[cur+1,ed],temp[2],temp[3]]

return ed-1,ed

```

```

def mod_table(inp,start,cur,end,table):
#print(inp)
k = 0
while k<len(inp):
#print(start,cur,end,k,inp[k:],len(table)-1)
if inp[k]=="a":
end = a_(cur,end)
#print("in a_")
elif inp[k]=="b":
end = b_(cur,end)
#print("in b_")
elif inp[k]=="e":
end = e_(cur,end)
elif inp[k]==".":
k+=1
if inp[k]=="a":
#k-=1
cur,end = and_a(cur,end)
elif inp[k]=="b":
cur,end = and_b(cur,end)
#k-=1
elif inp[k]=="(":
li = ["("]
l = k
for i in inp[k+1:]:

```

```

if i == "(":
li.append("(")
if i == ")":
try:
del li[-1]
except:
break
if len(li)==0:
break
l+=1
m = k
k=l+1
cur+=1
start,cur,end,table = mod_table(inp[m+1:l+1],start,cur,end,table)

elif inp[k]=="+":
k+=1
if inp[k]=="a":
or_a(cur,end)
#print("in or_a")
elif inp[k]=="b":
or_b(cur,end)
#print("in or_b")
else:
print(f"ERROR at{k }Done:{inp[:k+1]}Rem{inp[k+1:]}")

elif inp[k]=="*":
#print("in star")
cur,end = star(cur,end)
elif inp[k]=="(":
li = ["("]
l = k
for i in inp[k+1:]:
if i == "(":
li.append("(")
if i == ")":
try:
del li[-1]
except:
break
if len(li)==0:
break
l+=1
m = k

```

```

k=l+1
try:
if inp[k+1]=="*":
cur_ = cur
except:
pass
#print(inp[m+1:l+1])
start,cur,end,table = mod_table(inp[m+1:l+1],start,cur,end,table)
try:
if inp[k+1]=="*":
cur = cur_
except:
pass
else:
print(f'error{k}{inp[k]}')
k+=1
return start,cur,end,table

start,cur,end,table = mod_table(inp,start,cur,end,table)
print_t(table)

```

Output :

The screenshot shows a Visual Studio Code editor with a Python script named `test.py` open. The script defines a function `mod_table` and a function `print_t` to print an NFA table. The terminal output shows the execution of the script, which prints the NFA table for the input string `1234567`.

```

sumon@Lenovo:~/Desktop/Assignment-paper1$ python3 test.py
state | epsilon | a | b
-----|-----|---|---
1      | 2,7     | -  | -
2      | 3        | -  | -
3      | 4,6     | -  | -
4      | -        | -  | 5
5      | 4,6     | -  | -
6      | 2,7     | -  | -
7      | -        | -  | -

```

```
sumon@Lenovo:~/Desktop/Assignment-paper1$ python3 test.py
state      | epsilon   | a          | b
-----
1          | 2 7       | -          | -
2          | 3         | 3          | -
3          | 4 6       | -          | -
4          | -         | -          | 5
5          | 4 6       | -          | -
6          | 2 7       | -          | -
7          | -         | -          | -
```

2. Write a program in python for Elimination of Left Recursion

Source Code :

```
gram = {}

def add(str): #to rules together
    x = str.split("->")
    y = x[1]
    x.pop()
    z = y.split("|")
    x.append(z)
    gram[x[0]]=x[1]

def removeDirectLR(gramA, A):
    """gramA is dictionary"""
    temp = gramA[A]
    tempCr = []
    tempInCr = []
    for i in temp:
        if i[0] == A:
            #tempInCr.append(i[1:])
            tempInCr.append(i[1:]+[A+""])
        else:
            #tempCr.append(i)
            tempCr.append(i+[A+""])
            tempInCr.append(["e"])
    gramA[A] = tempCr
    gramA[A+""] = tempInCr
    return gramA

def checkForIndirect(gramA, a, ai):
    if ai not in gramA:
```



```

return False
if a == ai:
    return True
for i in gramA[ai]:
    if i[0] == ai:
        return False
    if i[0] in gramA:
        return checkForIndirect(gramA, a, i[0])
    return False

def rep(gramA, A):
    temp = gramA[A]
    newTemp = []
    for i in temp:
        if checkForIndirect(gramA, A, i[0]):
            t = []
            for k in gramA[i[0]]:
                t=[]
                t+=k
                t+=i[1:]
            newTemp.append(t)
        else:
            newTemp.append(i)
    gramA[A] = newTemp
    return gramA

def rem(gram):
    c = 1
    conv = {}
    gramA = {}
    revconv = {}
    for j in gram:
        conv[j] = "A"+str(c)
        gramA["A"+str(c)] = []
        c+=1

    for i in gram:
        for j in gram[i]:
            temp = []
            for k in j:
                if k in conv:
                    temp.append(conv[k])
                else:
                    temp.append(k)
            gramA[conv[i]].append(temp)

```

```

#print(gramA)
for i in range(c-1,0,-1):
    ai = "A"+str(i)
    for j in range(0,i):
        aj = gramA[ai][0][0]
        if ai!=aj :
            if aj in gramA and checkForIndirect(gramA,ai,aj):
                gramA = rep(gramA, ai)

    for i in range(1,c):
        ai = "A"+str(i)
        for j in gramA[ai]:
            if ai==j[0]:
                gramA = removeDirectLR(gramA, ai)
        break

    op = {}
    for i in gramA:
        a = str(i)
        for j in conv:
            a = a.replace(conv[j],j)
        revconv[i] = a

    for i in gramA:
        l = []
        for j in gramA[i]:
            k = []
            for m in j:
                if m in revconv:
                    k.append(m.replace(m,revconv[m]))
                else:
                    k.append(m)
            l.append(k)
        op[revconv[i]] = l

    return op

n = int(input("Enter No of Production: "))
for i in range(n):
    txt=input()
    add(txt)

result = rem(gram)

for x,y in result.items():
    print(f'{x} -> {y}')

```

Output :

```
1 gram = {}
2
3 def add(str):
4     x = str.split("->")
5     y = x[1]
6     x.pop()
7     z = y.split("|")
8     x.append(z)
9     gram[x[0]] = x[1]
10
11 def removeDirectLR(gramA, A):
12     """gramA is dictionary"""
13     temp = gramA[A]
14     tempCr = []
15     tempInCr = []
16     for i in temp:
17         if i[0] == A:
18             #tempInCr.append(i[1:])
19             tempInCr.append(i[1:] + [A + ""])
20         else:
21             #tempCr.append(i)
22             tempCr.append(i + [A + ""])
23     tempInCr.append(["ε"])
24     gramA[A] = tempCr
25     gramA[A + ""] = tempInCr
26     return gramA
27
28 def checkForIndirect(gramA, a, ai):
29     if ai not in gramA:
30         return False
31     if a == ai:
32         return True
33     for i in gramA[ai]:
34         if i[0] == a:
```

```
sumon@Lenovo:~/Desktop/Assignment-paper1$ python3 left_recursion.py
Enter No of Production: 4
A->aB
B->D
B->A
D->bZ
A-> [['a', 'B']]
B-> [['A']]
D-> [['b', 'Z']]
sumon@Lenovo:~/Desktop/Assignment-paper1$
```

```
sumon@Lenovo:~/Desktop/Assignment-paper1$ python3 left_recursion.py
Enter No of Production: 4
A->aB
B->D
B->A
D->bZ
A-> [['a', 'B']]
B-> [['A']]
D-> [['b', 'Z']]
```

3. Write a program in python for Computation of First and Follow sets

Source Code :

```
gram = {
    "E":["E+T","T"],
    "T":["T*F","F"],
    "F":["(E)","i"]
}

def removeDirectLR(gramA, A):
    """gramA is dictionary"""
    temp = gramA[A]
    tempCr = []
    tempInCr = []
    for i in temp:
        if i[0] == A:
            #tempInCr.append(i[1:])
            tempInCr.append(i[1:]+[A+""])
        else:
            #tempCr.append(i)
            tempCr.append(i+[A+""])
            tempInCr.append(['e'])
    gramA[A] = tempCr
    gramA[A+""] = tempInCr
    return gramA

def checkForIndirect(gramA, a, ai):
    if ai not in gramA:
        return False
    if a == ai:
        return True
    for i in gramA[ai]:
        if i[0] == ai:
            return False
        if i[0] in gramA:
            return checkForIndirect(gramA, a, i[0])
    return False

def rep(gramA, A):
    temp = gramA[A]
    newTemp = []
    for i in temp:
        if checkForIndirect(gramA, A, i[0]):
            t = []
            for k in gramA[i[0]]:
```

```

t=[]
t+=k
t+=i[1:]
newTemp.append(t)

else:
newTemp.append(i)
gramA[A] = newTemp
return gramA

def rem(gram):
c = 1
conv = {}
gramA = {}
revconv = {}
for j in gram:
conv[j] = "A"+str(c)
gramA["A"+str(c)] = []
c+=1

for i in gram:
for j in gram[i]:
temp = []
for k in j:
if k in conv:
temp.append(conv[k])
else:
temp.append(k)
gramA[conv[i]].append(temp)

#print(gramA)
for i in range(c-1,0,-1):
ai = "A"+str(i)
for j in range(0,i):
aj = gramA[ai][0][0]
if ai!=aj :
if aj in gramA and checkForIndirect(gramA,ai,aj):
gramA = rep(gramA, ai)

for i in range(1,c):
ai = "A"+str(i)
for j in gramA[ai]:
if ai==j[0]:
gramA = removeDirectLR(gramA, ai)
break

op = {}

```

```

for i in gramA:
    a = str(i)
    for j in conv:
        a = a.replace(conv[j],j)
    revconv[i] = a

for i in gramA:
    l = []
    for j in gramA[i]:
        k = []
        for m in j:
            if m in revconv:
                k.append(m.replace(m,revconv[m]))
            else:
                k.append(m)
        l.append(k)
    op[revconv[i]] = l

return op

result = rem(gram)

def first(gram, term):
    a = []
    if term not in gram:
        return [term]
    for i in gram[term]:
        if i[0] not in gram:
            a.append(i[0])
        elif i[0] in gram:
            a += first(gram, i[0])
    return a

firsts = {}
for i in result:
    firsts[i] = first(result,i)
print(f'First({i}):',firsts[i])
# temp = follow(result,i,i)
# temp = list(set(temp))
# temp = [x if x != "e" else "$" for x in temp]
# print(f'Follow({i}):',temp)

def follow(gram, term):
    a = []
    for rule in gram:
        for i in gram[rule]:
            if term in i:

```

```

temp = i
indx = i.index(term)
if indx+1!=len(i):
if i[-1] in firsts:
a+=firsts[i[-1]]
else:
a+=["e"]
else:
a+=["e"]
if rule != term and "e" in a:
a+= follow(grammar,rule)
return a

follows = {}
for i in result:
follows[i] = list(set(follow(result,i)))
if "e" in follows[i]:
follows[i].pop(follows[i].index("e"))
follows[i]+=["$"]
print(f'Follow({i}):',follows[i])

```

Output :

The screenshot shows a Visual Studio Code window with a Python file named 'First_fellow.py'. The code defines two functions: 'removeDirectLR' and 'checkForIndirect'. The 'removeDirectLR' function takes a grammar and a set of LR items, and returns a new set of LR items with direct items removed. The 'checkForIndirect' function checks if a set of LR items is indirect. The terminal output shows the results of these functions for a given grammar.

```

sumon@Lenovo:~/Desktop/Assignment-paper1$ python3 First_fellow.py
First(E): ['(', 'i']
First(T): ['+', '-']
First(F): ['+', '-']
First(E): ['+', 'e']
First(T): ['+', '-']
Follow(E): ['$', ')', 's']
Follow(T): ['+', '-']
Follow(F): ['+', '-']
Follow(E): ['$', ')', 's']
Follow(T): ['+', '-']
Follow(F): ['+', '-']

```

```
sumon@Lenovo:~/Desktop/Assignment-paper1$ python3 First_fellow.py
First(E): ['(', 'i']
First(T): ['(', 'i']
First(F): ['(', 'i']
First(E'): ['+', 'e']
First(T'): ['*', 'e']
Follow(E): [')', '$']
Follow(T): ['+', ')', '$']
Follow(F): ['+', '*', ')', '$']
Follow(E'): [')', '$']
Follow(T'): [')', '+', '$']
```

4. Write a program in python for the Implementation of Shift Reduce Parsing

Source Code :

```
gram = {
    "S":["S+S","S*S","i"]
}
starting_terminal = "S"
inp = "i+i*i"
"""
stack = "$"
print(f{"Stack": <15}'+|'+f{"Input Buffer": <15}'+|'+f"Parsing Action")
print(f{"-":-<50}')
while True:
    action = True
    i = 0
    while i<len(gram[starting_terminal]):
        if gram[starting_terminal][i] in stack:
            stack = stack.replace(gram[starting_terminal][i],starting_terminal)
            print(f{"stack: <15}'+|'+f{"inp: <15}'+|'+f"Reduce S->{gram[starting_terminal][i]}')
            i=-1
            action = False
            i+=1
        if len(inp)>1:
            stack+=inp[0]
            inp=inp[1:]
            print(f{"stack: <15}'+|'+f{"inp: <15}'+|'+f"Shift')
            action = False
```



```

if inp == "$" and stack == ("S"+starting_terminal):
print(f'{stack: <15}'+ "|" +f'{inp: <15}'+ "|" +f'Accepted')
break

if action:
print(f'{stack: <15}'+ "|" +f'{inp: <15}'+ "|" +f'Rejected')
break

```

Output :

The screenshot shows the Visual Studio Code editor with the file `shift_reduce.py` open. The code implements a shift-reduce parser. The terminal output shows the execution of the script, displaying the stack, input buffer, and parsing actions for the input string `i*i*i`.

```

sumon@Lenovo:~/Desktop/Assignment-paper1$ python3 shift_reduce.py
Stack      |Input Buffer  |Parsing Action
-----
$i         |+i*i         |Shift
$$         |+i*i         |Reduce S->i
$$+        |i*i          |Shift
$$+i       |*i           |Shift
$$+S       |+i           |Reduce S->i
$$         |+i           |Reduce S->S+S
$$*        |i            |Shift
$$*        |i            |Rejected

```

```

sumon@Lenovo:~/Desktop/Assignment-paper1$ python3 shift_reduce.py
Stack      |Input Buffer  |Parsing Action
-----
$i         |+i*i         |Shift
$$         |+i*i         |Reduce S->i
$$+        |i*i          |Shift
$$+i       |*i           |Shift
$$+S       |+i           |Reduce S->i
$$         |+i           |Reduce S->S+S
$$*        |i            |Shift
$$*        |i            |Rejected

```

5. Write a program in python for intermediate Code Generation: Three Address Code, Postfix, Prefix

Source Code :

```
OPERATORS = set(['+', '-', '*', '/', '(', ')'])
PRI = {'+':1, '-':1, '*':2, '/':2}

### INFIX ==> POSTFIX ###
def infix_to_postfix(formula):
    stack = [] # only pop when the coming op has priority
    output = ""
    for ch in formula:
        if ch not in OPERATORS:
            output += ch
        elif ch == '(':
            stack.append('(')
        elif ch == ')':
            while stack and stack[-1] != '(':
                output += stack.pop()
            stack.pop() # pop '('
        else:
            while stack and stack[-1] != '(' and PRI[ch] <= PRI[stack[-1]]:
                output += stack.pop()
            stack.append(ch)
    # leftover
    while stack:
        output += stack.pop()
    print(f'POSTFIX: {output}')
    return output

### INFIX ==> PREFIX ###
def infix_to_prefix(formula):
    op_stack = []
    exp_stack = []
    for ch in formula:
        if not ch in OPERATORS:
            exp_stack.append(ch)
        elif ch == '(':
            op_stack.append(ch)
```

```

elif ch == ')':
while op_stack[-1] != '(':
op = op_stack.pop()
a = exp_stack.pop()
b = exp_stack.pop()
exp_stack.append( op+b+a )
op_stack.pop() # pop '('
else:
while op_stack and op_stack[-1] != '(' and PRI[ch] <= PRI[op_stack[-1]]:
op = op_stack.pop()
a = exp_stack.pop()
b = exp_stack.pop()
exp_stack.append( op+b+a )
op_stack.append(ch)

# leftover
while op_stack:
op = op_stack.pop()
a = exp_stack.pop()
b = exp_stack.pop()
exp_stack.append( op+b+a )
print(f'PREFIX: {exp_stack[-1]}')
return exp_stack[-1]

### THREE ADDRESS CODE GENERATION ###
def generate3AC(pos):
print("### THREE ADDRESS CODE GENERATION ###")
exp_stack = []
t = 1

for i in pos:
if i not in OPERATORS:
exp_stack.append(i)
else:
print(f't{t} := {exp_stack[-2]} {i} {exp_stack[-1]}')
exp_stack=exp_stack[:-2]
exp_stack.append(f't{t}')
t+=1

expres = input("INPUT THE EXPRESSION: ")
pre = infix_to_prefix(expres)
pos = infix_to_postfix(expres)
generate3AC(pos)

```

Output :

The screenshot shows the Visual Studio Code interface with a Python file named `inf_pos_thr.py` open. The code implements functions for converting infix expressions to postfix and prefix, and generating three-address code. The terminal output shows the execution of the script with the input expression `a+b*c`, resulting in the postfix `abc*+` and the three-address code generation steps.

```
1 OPERATORS = set(['+', '-', '*', '/', '(', ')'])
2 PRI = {'+':1, '-':1, '*':2, '/':2}
3
4 ### INFIX ==> POSTFIX ###
5 def infix_to_postfix(formula):
6     stack = [] # only pop when the coming op has priority
7     output = ''
8     for ch in formula:
9         if ch not in OPERATORS:
10             output += ch
11         elif ch == '(':
12             stack.append('(')
13         elif ch == ')':
14             while stack and stack[-1] != '(':
15                 output += stack.pop()
16             stack.pop() # pop '('
17         else:
18             while stack and stack[-1] != '(' and PRI[ch] <= PRI[stack[-1]]:
19                 output += stack.pop()
20             stack.append(ch)
21     # leftover
22     while stack:
23         output += stack.pop()
24     print(f'POSTFIX: {output}')
25     return output
26
27 ### INFIX ==> PREFIX ###
28 def infix_to_prefix(formula):
29     op_stack = []
30     exp_stack = []
31     for ch in formula:
32         if not ch in OPERATORS:
33             exp_stack.append(ch)
34         elif ch == '(':
35             op_stack.append(ch)
36         elif ch == ')':
37             op = op_stack.pop()
38             while op != '(':
39                 output += op_stack.pop()
40             op_stack.pop()
41     # leftover
42     while op_stack:
43         output += op_stack.pop()
44     print(f'PREFIX: {output}')
45     return output
46
47 ### THREE ADDRESS CODE GENERATION ###
48 t1 := b * c
49 t2 := a + t1
```

Terminal Output:

```
sumon@Lenovo: ~/Desktop/Assignment-paper1$ python3 inf_pos_thr.py
INPUT THE EXPRESSION: a+b*c
PREFIX: +a*bc
POSTFIX: abc*+
### THREE ADDRESS CODE GENERATION ###
t1 := b * c
t2 := a + t1
```

```
sumon@Lenovo:~/Desktop/Assignment-paper1$ python3 inf_pos_thr.py
INPUT THE EXPRESSION: a+b*c
PREFIX: +a*bc
POSTFIX: abc*+
### THREE ADDRESS CODE GENERATION ###
t1 := b * c
t2 := a + t1
```