**COMPILER MINI PROJECT**

**CODE:**

def removeLeftRecursion(rulesDiction):

store = {}

for lhs in rulesDiction:

alphaRules = []

betaRules = []

allrhs = rulesDiction[lhs]

for subrhs in allrhs:

if subrhs[0] == lhs:

alphaRules.append(subrhs[1:])

else:

betaRules.append(subrhs)

if len(alphaRules) != 0:

lhs\_ = lhs + "'"

while (lhs\_ in rulesDiction.keys()) \

or (lhs\_ in store.keys()):

lhs\_ += "'"

for b in range(0, len(betaRules)):

betaRules[b].append(lhs\_)

rulesDiction[lhs] = betaRules

for a in range(0, len(alphaRules)):

alphaRules[a].append(lhs\_)

alphaRules.append(['#'])

store[lhs\_] = alphaRules

for left in store:

rulesDiction[left] = store[left]

return rulesDiction

def LeftFactoring(rulesDiction):

newDict = {}

for lhs in rulesDiction:

# get rhs for given lhs

allrhs = rulesDiction[lhs]

# temp dictionary helps detect left factoring

temp = dict()

for subrhs in allrhs:

if subrhs[0] not in list(temp.keys()):

temp[subrhs[0]] = [subrhs]

else:

temp[subrhs[0]].append(subrhs)

new\_rule = []

# temp\_dict stores new subrules for left factoring

tempo\_dict = {}

for term\_key in temp:

# get value from temp for term\_key

allStartingWithTermKey = temp[term\_key]

if len(allStartingWithTermKey) > 1:

# left factoring required

# to generate new unique symbol

# - add ' till unique not generated

lhs\_ = lhs + "'"

while (lhs\_ in rulesDiction.keys()) \

or (lhs\_ in tempo\_dict.keys()):

lhs\_ += "'"

# append the left factored result

new\_rule.append([term\_key, lhs\_])

# add expanded rules to tempo\_dict

ex\_rules = []

for g in temp[term\_key]:

ex\_rules.append(g[1:])

tempo\_dict[lhs\_] = ex\_rules

else:

# no left factoring required

new\_rule.append(allStartingWithTermKey[0])

# add original rule

newDict[lhs] = new\_rule

# add newly generated rules after left factoring

for key in tempo\_dict:

newDict[key] = tempo\_dict[key]

return newDict

# calculation of first

# epsilon is denoted by '#' (semi-colon)

# pass rule in first function

def first(rule):

global rules, nonterm\_userdef, \

term\_userdef, diction, firsts

# recursion base condition

# (for terminal or epsilon)

if len(rule) != 0 and (rule is not None):

if rule[0] in term\_userdef:

return rule[0]

elif rule[0] == '#':

return '#'

# condition for Non-Terminals

if len(rule) != 0:

if rule[0] in list(diction.keys()):

# fres temporary list of result

fres = []

rhs\_rules = diction[rule[0]]

# call first on each rule of RHS

# fetched (& take union)

for itr in rhs\_rules:

indivRes = first(itr)

if type(indivRes) is list:

for i in indivRes:

fres.append(i)

else:

fres.append(indivRes)

# if no epsilon in result

# - received return fres

if '#' not in fres:

return fres

else:

# apply epsilon

# rule => f(ABC)=f(A)-{e} U f(BC)

newList = []

fres.remove('#')

if len(rule) > 1:

ansNew = first(rule[1:])

if ansNew != None:

if type(ansNew) is list:

newList = fres + ansNew

else:

newList = fres + [ansNew]

else:

newList = fres

return newList

# if result is not already returned

# - control reaches here

# lastly if eplison still persists

# - keep it in result of first

fres.append('#')

return fres

# calculation of follow

# use 'rules' list, and 'diction' dict from above

# follow function input is the split result on

# - Non-Terminal whose Follow we want to compute

def follow(nt):

global start\_symbol, rules, nonterm\_userdef, \

term\_userdef, diction, firsts, follows

# for start symbol return $ (recursion base case)

solset = set()

if nt == start\_symbol:

# return '$'

solset.add('$')

# check all occurrences

# solset - is result of computed 'follow' so far

# For input, check in all rules

for curNT in diction:

rhs = diction[curNT]

# go for all productions of NT

for subrule in rhs:

if nt in subrule:

# call for all occurrences on

# - non-terminal in subrule

while nt in subrule:

index\_nt = subrule.index(nt)

subrule = subrule[index\_nt + 1:]

# empty condition - call follow on LHS

if len(subrule) != 0:

# compute first if symbols on

# - RHS of target Non-Terminal exists

res = first(subrule)

if '#' in res:

newList = []

res.remove('#')

ansNew = follow(curNT)

if ansNew != None:

if type(ansNew) is list:

newList = res + ansNew

else:

newList = res + [ansNew]

else:

newList = res

res = newList

else:

# when nothing in RHS, go circular

# - and take follow of LHS

# only if (NT in LHS)!=curNT

if nt != curNT:

res = follow(curNT)

# add follow result in set form

if res is not None:

if type(res) is list:

for g in res:

solset.add(g)

else:

solset.add(res)

return list(solset)

def computeAllFirsts():

global rules, nonterm\_userdef, \

term\_userdef, diction, firsts

for rule in rules:

k = rule.split("->")

# remove un-necessary spaces

k[0] = k[0].strip()

k[1] = k[1].strip()

rhs = k[1]

multirhs = rhs.split('|')

# remove un-necessary spaces

for i in range(len(multirhs)):

multirhs[i] = multirhs[i].strip()

multirhs[i] = multirhs[i].split()

diction[k[0]] = multirhs

print(f"\nRules: \n")

for y in diction:

print(f"{y}->{diction[y]}")

print(f"\nAfter elimination of left recursion:\n")

diction = removeLeftRecursion(diction)

for y in diction:

print(f"{y}->{diction[y]}")

print("\nAfter left factoring:\n")

diction = LeftFactoring(diction)

for y in diction:

print(f"{y}->{diction[y]}")

# calculate first for each rule

# - (call first() on all RHS)

for y in list(diction.keys()):

t = set()

for sub in diction.get(y):

res = first(sub)

if res != None:

if type(res) is list:

for u in res:

t.add(u)

else:

t.add(res)

# save result in 'firsts' list

firsts[y] = t

print("\nCalculated firsts: ")

key\_list = list(firsts.keys())

index = 0

for gg in firsts:

print(f"first({key\_list[index]}) "

f"=> {firsts.get(gg)}")

index += 1

def computeAllFollows():

global start\_symbol, rules, nonterm\_userdef,\

term\_userdef, diction, firsts, follows

for NT in diction:

solset = set()

sol = follow(NT)

if sol is not None:

for g in sol:

solset.add(g)

follows[NT] = solset

print("\nCalculated follows: ")

key\_list = list(follows.keys())

index = 0

for gg in follows:

print(f"follow({key\_list[index]})"

f" => {follows[gg]}")

index += 1

# create parse table

def createParseTable():

import copy

global diction, firsts, follows, term\_userdef

print("\nFirsts and Follow Result table\n")

# find space size

mx\_len\_first = 0

mx\_len\_fol = 0

for u in diction:

k1 = len(str(firsts[u]))

k2 = len(str(follows[u]))

if k1 > mx\_len\_first:

mx\_len\_first = k1

if k2 > mx\_len\_fol:

mx\_len\_fol = k2

print(f"{{:<{10}}} "

f"{{:<{mx\_len\_first + 5}}} "

f"{{:<{mx\_len\_fol + 5}}}"

.format("Non-T", "FIRST", "FOLLOW"))

for u in diction:

print(f"{{:<{10}}} "

f"{{:<{mx\_len\_first + 5}}} "

f"{{:<{mx\_len\_fol + 5}}}"

.format(u, str(firsts[u]), str(follows[u])))

# create matrix of row(NT) x [col(T) + 1($)]

# create list of non-terminals

ntlist = list(diction.keys())

terminals = copy.deepcopy(term\_userdef)

terminals.append('$')

# create the initial empty state of ,matrix

mat = []

for x in diction:

row = []

for y in terminals:

row.append('')

# of $ append one more col

mat.append(row)

# Classifying grammar as LL(1) or not LL(1)

grammar\_is\_LL = True

# rules implementation

for lhs in diction:

rhs = diction[lhs]

for y in rhs:

res = first(y)

# epsilon is present,

# - take union with follow

if '#' in res:

if type(res) == str:

firstFollow = []

fol\_op = follows[lhs]

if fol\_op is str:

firstFollow.append(fol\_op)

else:

for u in fol\_op:

firstFollow.append(u)

res = firstFollow

else:

res.remove('#')

res = list(res) +\

list(follows[lhs])

# add rules to table

ttemp = []

if type(res) is str:

ttemp.append(res)

res = copy.deepcopy(ttemp)

for c in res:

xnt = ntlist.index(lhs)

yt = terminals.index(c)

if mat[xnt][yt] == '':

mat[xnt][yt] = mat[xnt][yt] \

+ f"{lhs}->{' '.join(y)}"

else:

# if rule already present

if f"{lhs}->{y}" in mat[xnt][yt]:

continue

else:

grammar\_is\_LL = False

mat[xnt][yt] = mat[xnt][yt] \

+ f",{lhs}->{' '.join(y)}"

# final state of parse table

print("\nGenerated parsing table:\n")

frmt = "{:>12}" \* len(terminals)

print(frmt.format(\*terminals))

j = 0

for y in mat:

frmt1 = "{:>12}" \* len(y)

print(f"{ntlist[j]} {frmt1.format(\*y)}")

j += 1

return (mat, grammar\_is\_LL, terminals)

def validateStringUsingStackBuffer(parsing\_table, grammarll1,

table\_term\_list, input\_string,

term\_userdef,start\_symbol):

print(f"\nValidate String => {input\_string}\n")

# for more than one entries

# - in one cell of parsing table

if grammarll1 == False:

return f"\nInput String = " \

f"\"{input\_string}\"\n" \

f"Grammar is not LL(1)"

# implementing stack buffer

stack = [start\_symbol, '$']

buffer = []

# reverse input string store in buffer

input\_string = input\_string.split()

input\_string.reverse()

buffer = ['$'] + input\_string

print("{:>20} {:>20} {:>20}".

format("Buffer", "Stack","Action"))

while True:

# end loop if all symbols matched

if stack == ['$'] and buffer == ['$']:

print("{:>20} {:>20} {:>20}"

.format(' '.join(buffer),

' '.join(stack),

"Valid"))

return "\nValid String!"

elif stack[0] not in term\_userdef:

# take font of buffer (y) and tos (x)

x = list(diction.keys()).index(stack[0])

y = table\_term\_list.index(buffer[-1])

if parsing\_table[x][y] != '':

# format table entry received

entry = parsing\_table[x][y]

print("{:>20} {:>20} {:>25}".

format(' '.join(buffer),

' '.join(stack),

f"T[{stack[0]}][{buffer[-1]}] = {entry}"))

lhs\_rhs = entry.split("->")

lhs\_rhs[1] = lhs\_rhs[1].replace('#', '').strip()

entryrhs = lhs\_rhs[1].split()

stack = entryrhs + stack[1:]

else:

return f"\nInvalid String! No rule at " \

f"Table[{stack[0]}][{buffer[-1]}]."

else:

# stack top is Terminal

if stack[0] == buffer[-1]:

print("{:>20} {:>20} {:>20}"

.format(' '.join(buffer),

' '.join(stack),

f"Matched:{stack[0]}"))

buffer = buffer[:-1]

stack = stack[1:]

else:

return "\nInvalid String! " \

"Unmatched terminal symbols"

sample\_input\_string = None

rules=["S -> A k O",

"A -> A d | a B | a C",

"C -> c",

"B -> b B C | r"]

nonterm\_userdef=['A','B','C']

term\_userdef=['k','O','d','a','c','b','r']

sample\_input\_string="a r k O"

diction = {}

firsts = {}

follows = {}

# computes all FIRSTs for all non terminals

computeAllFirsts()

# assuming first rule has start\_symbol

# start symbol can be modified in below line of code

start\_symbol = list(diction.keys())[0]

# computes all FOLLOWs for all occurrences

computeAllFollows()

# generate formatted first and follow table

# then generate parse table

(parsing\_table, result, tabTerm) = createParseTable()

# validate string input using stack-buffer concept

if sample\_input\_string != None:

validity = validateStringUsingStackBuffer(parsing\_table, result,

tabTerm, sample\_input\_string,

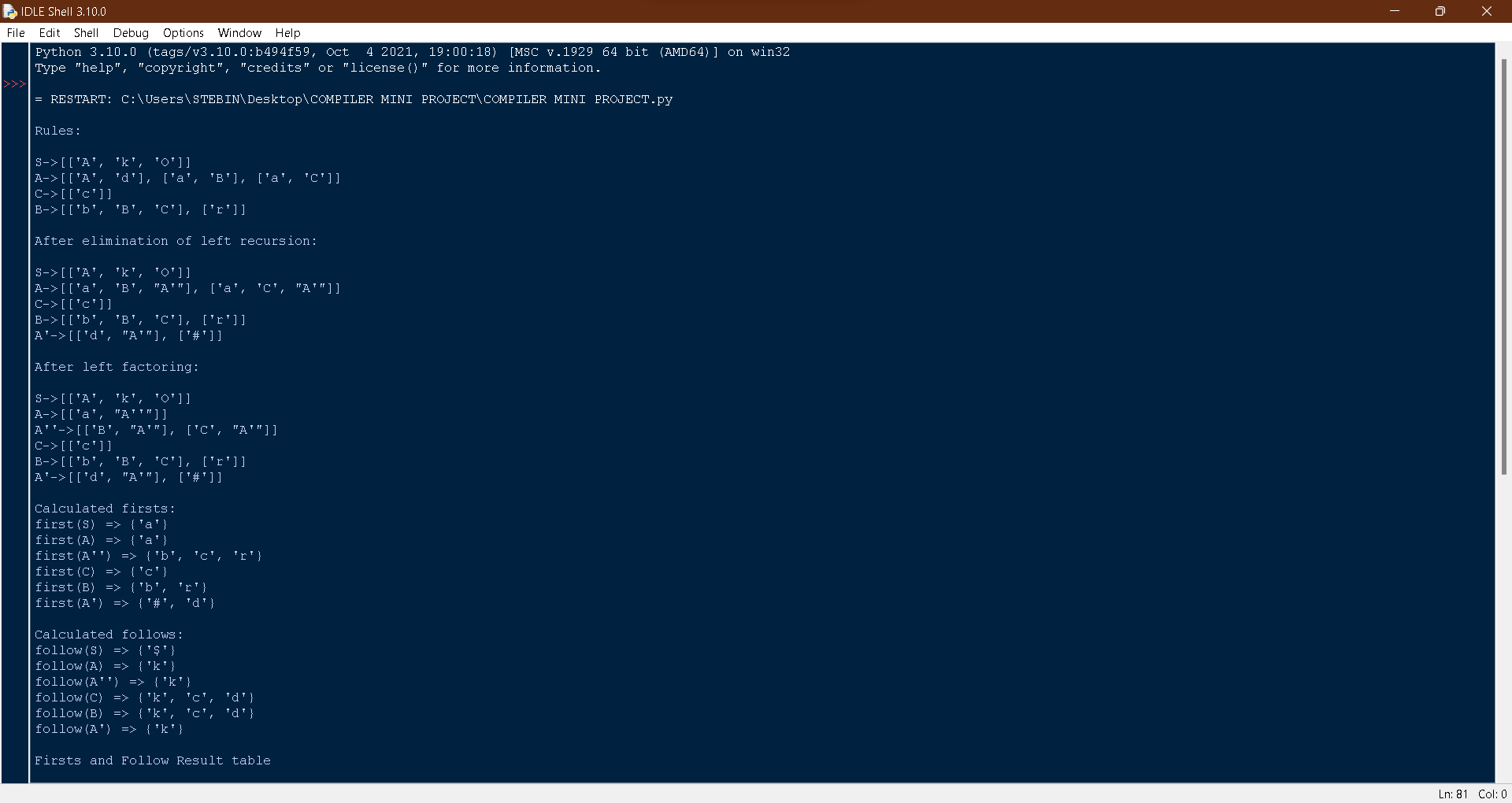
term\_userdef,start\_symbol)

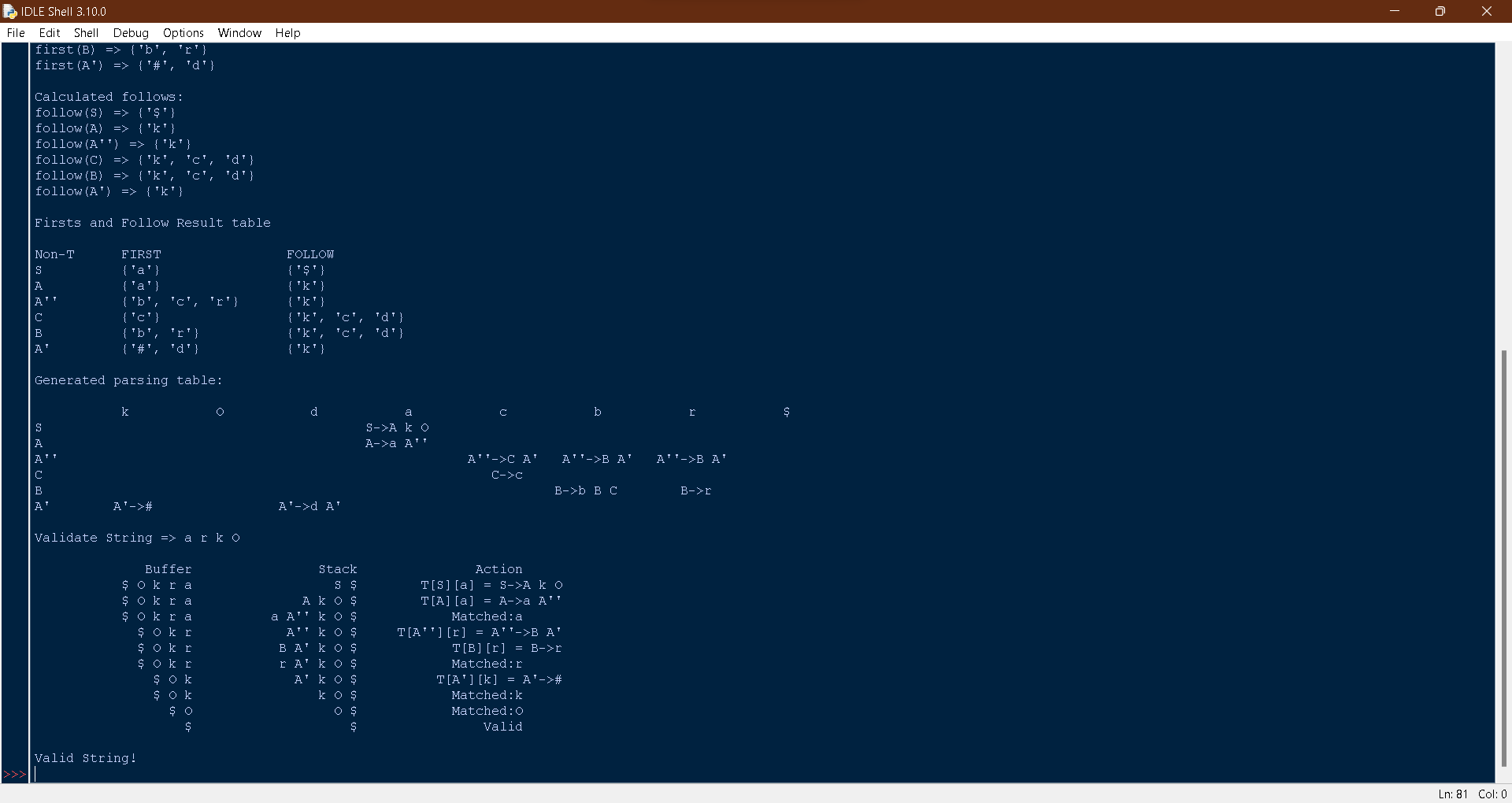
print(validity)

else:

print("\nNo input String detected")

**OUTPUTS:**

****

****