SLR1.py

```
import copy

def grammarAugmentation(rules,
nonterm_userdef,
                        start_symbol)
:

    newRules = []

    newChar = start_symbol + "'"
    while (newChar in
nonterm_userdef):
        newChar += "'"

    newRules.append([newChar,
                     ['.',
start_symbol]])

    for rule in rules:

        k = rule.split("->")
        lhs = k[0].strip()
        rhs = k[1].strip()

        multirhs = rhs.split('|')
        for rhs1 in multirhs:
            rhs1 =
rhs1.strip().split()

            rhs1.insert(0, '.')
            newRules.append([lhs,
rhs1])
    return newRules



def findFirst(non_terminal):
    grammar = {
        'S': [['E']],
        'E': [['T', '+', 'E',],
['T']],
        'T': [['F', '*', 'T'],
['F']],
        'F': [['id']],
    }
    first_set = set()

    if non_terminal not in grammar:
        first_set.add(non_terminal)
        return first_set

    for production in
grammar[non_terminal]:

        if not production:
            first_set.add('ε')
        else:

            if production[0] not in
grammar.keys():
                first_set.add(product
ion[0])
            else:

                first_set.update(find
First(production[0]))

    return first_set




def findFollow(non_terminal,
visited=None):
    grammar = {
        'S': [['E']],
        'E': [['T', '+', 'E',],
['T']],
        'T': [['F', '*', 'T'],
['F']],
        'F': [['id']],
    }
```

```
    if visited is None:
        visited = set()

    follow_set = set()


    follow_set.add('$')

    if non_terminal != 'S':
        follow_set.add('+')

    if non_terminal not in ['S',
'E']:
        follow_set.add('*')

    return follow_set

    if non_terminal in visited:
        return follow_set

    visited.add(non_terminal)


    for symbol, productions in
grammar.items():
        for production in
productions:

            if non_terminal in
production:
                index =
production.index(non_terminal)


            if index ==
len(production) - 1:
                if symbol !=
non_terminal:
                    follow_set.update
(findFollow(symbol, visited))


            elif production[index +
1] not in grammar:
                follow_set.add(produc
tion[index + 1])


            else:

                first_set =
findFollow(production[index + 1])


                if 'ε' in first_set:
                    follow_set.update
(findFollow(symbol, visited))
                    first_set.remove(
'ε')


                follow_set.update(fir
st_set)

    return follow_set


print("------------First sets--------
-------")
print("S:", findFirst('S'))
print("E:", findFirst('E'))
print("T:", findFirst('T'))
print('F:', findFirst('F'))
print("------------------------------
-------")


print("------------Follow sets-------
--------")
print("S:", findFollow('S'))
print("E:", findFollow('E'))
print("T:", findFollow('T'))
print('F:', findFollow('F'))
print("------------------------------
--------")
```

```
def findClosure(input_state,
dotSymbol):
    global start_symbol, \
        separatedRulesList, \
        statesDict

    closureSet = []

    if dotSymbol == start_symbol:
        for rule in
separatedRulesList:
            if rule[0] == dotSymbol:
                closureSet.append(rul
e)
    else:
        closureSet = input_state

    prevLen = -1
    while prevLen != len(closureSet):
        prevLen = len(closureSet)

        tempClosureSet = []

        for rule in closureSet:
            indexOfDot =
rule[1].index('.')
            if rule[1][-1] != '.':
                dotPointsHere =
rule[1][indexOfDot + 1]
                for in_rule in
separatedRulesList:
                    if dotPointsHere
== in_rule[0] and \
                            in_rule
not in tempClosureSet:
                        tempClosureSe
t.append(in_rule)

        for rule in tempClosureSet:
            if rule not in
closureSet:
                closureSet.append(rul
e)
    return closureSet
```

```
def compute_GOTO(state):
    global statesDict, stateCount

    generateStatesFor = []
    for rule in statesDict[state]:
        if rule[1][-1] != '.':
            indexOfDot =
rule[1].index('.')
            dotPointsHere =
rule[1][indexOfDot + 1]
            if dotPointsHere not in
generateStatesFor:
                generateStatesFor.app
end(dotPointsHere)

    if len(generateStatesFor) != 0:
        for symbol in
generateStatesFor:
            GOTO(state, symbol)
    return


def GOTO(state, charNextToDot):
    global statesDict, stateCount,
stateMap

    newState = []
    for rule in statesDict[state]:
        indexOfDot =
rule[1].index('.')
        if rule[1][-1] != '.':
            if rule[1][indexOfDot +
1] == \
                    charNextToDot:
                shiftedRule =
copy.deepcopy(rule)
                shiftedRule[1][indexO
fDot] = \
                    shiftedRule[1][in
dexOfDot + 1]
                shiftedRule[1][indexO
fDot + 1] = '.'
                newState.append(shift
edRule)
```

```
    addClosureRules = []
    for rule in newState:
        indexDot = rule[1].index('.')
        if rule[1][-1] != '.':
            closureRes = \
                findClosure(newState,
rule[1][indexDot + 1])
            for rule in closureRes:
                if rule not in
addClosureRules \
                        and rule not
in newState:
                    addClosureRules.a
ppend(rule)

    for rule in addClosureRules:
        newState.append(rule)

    stateExists = -1
    for state_num in statesDict:
        if statesDict[state_num] ==
newState:
            stateExists = state_num
            break

    if stateExists == -1:

        stateCount += 1
        statesDict[stateCount] =
newState
        stateMap[(state,
charNextToDot)] = stateCount
    else:

        stateMap[(state,
charNextToDot)] = stateExists
    return


def generateStates(statesDict):
    prev_len = -1
    called_GOTO_on = []

    while (len(statesDict) !=
prev_len):
```

```
        prev_len = len(statesDict)
        keys =
list(statesDict.keys())

        for key in keys:
            if key not in
called_GOTO_on:
                called_GOTO_on.append
(key)
                compute_GOTO(key)
    return


def first(rule):
    global rules, nonterm_userdef, \
        term_userdef, diction, firsts

    if len(rule) != 0 and (rule is
not None):
        if rule[0] in term_userdef:
            return rule[0]

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

            fres = []
            rhs_rules =
diction[rule[0]]

            for itr in rhs_rules:
                indivRes = first(itr)
                if type(indivRes) is
list:
                    for i in
indivRes:
                        fres.append(i
)
                else:
                    fres.append(indiv
Res)

                return fres
            else:
```

```
                newList = []
                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

                return fres


def follow(nt):
    global start_symbol, rules,
nonterm_userdef, \
        term_userdef, diction,
firsts, follows

    solset = set()
    if nt == start_symbol:
        solset.add('$')


    for curNT in diction:
        rhs = diction[curNT]

        for subrule in rhs:
            if nt in subrule:

                while nt in subrule:
                    index_nt =
subrule.index(nt)
                    subrule =
subrule[index_nt + 1:]
```

```
                    if len(subrule)
!= 0:

                        res =
first(subrule)

                            newList =
[]
                            ansNew =
follow(curNT)
                            if ansNew
!= None:
                                if
type(ansNew) is list:
                                    n
ewList = res + ansNew
                                else:
                                    n
ewList = res + [ansNew]
                            else:
                                newLi
st = res
                            res =
newList
                    else:

                        if nt !=
curNT:
                            res =
follow(curNT)

                    if res is not
None:
                        if type(res)
is list:
                            for g in
res:
                                solse
t.add(g)
                        else:
                            solset.ad
d(res)
    return list(solset)
```

```
def createParseTable(statesDict, stateMap, T, NT):
    global separatedRulesList, diction

    rows = list(statesDict.keys())
    cols = T+['$']+NT

    Table = []
    tempRow = []
    for y in range(len(cols)):
        tempRow.append('')
    for x in range(len(rows)):
        Table.append(copy.deepcopy(tempRow))

    for entry in stateMap:
        state = entry[0]
        symbol = entry[1]
        a = rows.index(state)
        b = cols.index(symbol)
        if symbol in NT:
            Table[a][b] = Table[a][b]\
                + f"{stateMap[entry]} "
        elif symbol in T:
            Table[a][b] = Table[a][b]\
                + f"S{stateMap[entry]} "


    numbered = {}
    key_count = 0
    for rule in separatedRulesList:
        tempRule = copy.deepcopy(rule)
        tempRule[1].remove('.')
        numbered[key_count] = tempRule
        key_count += 1

    addedR = f"{separatedRulesList[0][0]} -> " \
        f"{separatedRulesList[0][1][1]}"
    rules.insert(0, addedR)
    for rule in rules:
        k = rule.split("->")

        k[0] = k[0].strip()
        k[1] = k[1].strip()
        rhs = k[1]
        multirhs = rhs.split('|')

        for i in range(len(multirhs)):
            multirhs[i] = multirhs[i].strip()
            multirhs[i] = multirhs[i].split()
        diction[k[0]] = multirhs

    for stateno in statesDict:
        for rule in statesDict[stateno]:
            if rule[1][-1] == '.':

                temp2 = copy.deepcopy(rule)
                temp2[1].remove('.')
                for key in numbered:
                    if numbered[key] == temp2:

                        follow_result = follow(rule[0])
                        for col in follow_result:
                            index = cols.index(col)
                            if key == 0:
                                Table[stateno][index] = "Accept"
                            else:
                                Table[stateno][index] =\
```

```
                                    T
able[stateno][index]+f"R{key} "

    print("\nSLR(1) parsing
table:\n")
    frmt = "{:>8}" * len(cols)
    print(" ", frmt.format(*cols),
"\n")
    ptr = 0
    j = 0
    for y in Table:
        frmt1 = "{:>8}" * len(y)
        print(f"{{:>3}}
{frmt1.format(*y)}"
            .format('I'+str(j)))
        j += 1

def printResult(rules):
    for rule in rules:
        print(f"{rule[0]} ->"
            f" {' '.join(rule[1])}")

def printAllGOTO(diction):
    for itr in diction:
        print(f"GOTO ( I{itr[0]} ,"
            f" {itr[1]} ) =
I{stateMap[itr]}")



rules = ["S -> E",
        "E -> T + E | T",
        "T -> T * F | F",
        "F -> id"
        ]
nonterm_userdef = ['E', 'T', 'F',
'S']
term_userdef = ['id', '+', '*']
start_symbol = nonterm_userdef[0]


print("\nOriginal grammar input:\n")
for y in rules:
    print(y)

print("\nGrammar after Augmentation:
\n")
separatedRulesList = \
    grammarAugmentation(rules,
                        nonterm_userd
ef,
                        start_symbol)
printResult(separatedRulesList)

start_symbol =
separatedRulesList[0][0]
print("\nCalculated closure: I0\n")
I0 = findClosure(0, start_symbol)
printResult(I0)

statesDict = {}
stateMap = {}

statesDict[0] = I0
stateCount = 0

generateStates(statesDict)

print("\nStates Generated: \n")
for st in statesDict:
    print(f"State = I{st}")
    printResult(statesDict[st])
    print()

print("Result of GOTO
computation:\n")
printAllGOTO(stateMap)

diction = {}

createParseTable(statesDict,
stateMap,
                 term_userdef,
                 nonterm_userdef)
```

Output

```
PS C:\Users\PMLS> python -u "c:\Users\PMLS\Downloads\slr.py"
------------First sets---------------
S: {'id'}
E: {'id'}
T: {'id'}
F: {'id'}
-------------------------------------
------------Follow sets---------------
S: {'$'}
E: {'$', '+'}
T: {'*', '$', '+'}
F: {'*', '$', '+'}
-------------------------------------

Original grammar input:

S -> E
E -> T + E | T
T -> T * F | F
F -> id
```

```
SLR(1) parsing table:

        id       +       *       $       E       T       F       S

 I0      S4                              1       2       3
 I1                              Accept
 I2              S5      S6      R3
 I3              R5      R5      R5
 I4              R6      R6      R6
 I5      S4                              7       2       3
 I6      S4                                              8
 I7                              R2
 I8              R4      R4      R4
PS C:\Users\PMLS>
```