COMPILER CONSTRUCTION (UCS802)
LAB ASSIGNMENT - II

Name: Satyam Sharma
Rollno: 102117172
Group: 4CS7

Problem: Create an SLR parser for any given grammar

Code:

```
# SLR(1)

import copy

# perform grammar augmentation
def grammarAugmentation(rules, nonterm_userdef,
                                                start_symbol):

        # newRules stores processed output rules
        newRules = []

        # create unique 'symbol' to
        # - represent new start symbol
        newChar = start_symbol + "'"
        while (newChar in nonterm_userdef):
                newChar += "'"

        # adding rule to bring start symbol to RHS
        newRules.append([newChar,
                                        ['.', start_symbol]])

        # new format => [LHS,[.RHS]],
        # can't use dictionary since
        # - duplicate keys can be there
        for rule in rules:

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

                # split all rule at '|'
```

```
                # keep single derivation in one rule
                multirhs = rhs.split('|')
                for rhs1 in multirhs:
                        rhs1 = rhs1.strip().split()

                        # ADD dot pointer at start of RHS
                        rhs1.insert(0, '.')
                        newRules.append([lhs, rhs1])
        return newRules


# find closure
def findClosure(input_state, dotSymbol):
        global start_symbol, \
                separatedRulesList, \
                statesDict

        # closureSet stores processed output
        closureSet = []

        # if findClosure is called for
        # - 1st time i.e. for I0,
        # then LHS is received in "dotSymbol",
        # add all rules starting with
        # - LHS symbol to closureSet
        if dotSymbol == start_symbol:
                for rule in separatedRulesList:
                        if rule[0] == dotSymbol:
                                closureSet.append(rule)
        else:
                # for any higher state than I0,
                # set initial state as
                # - received input_state
                closureSet = input_state

        # iterate till new states are
        # - getting added in closureSet
        prevLen = -1
        while prevLen != len(closureSet):
                prevLen = len(closureSet)

                # "tempClosureSet" - used to eliminate
                # concurrent modification error
                tempClosureSet = []
```

```
                # if dot pointing at new symbol,
                # add corresponding rules to tempClosure
                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:
                                                tempClosureSet.append(in_rule)

                # add new closure rules to closureSet
                for rule in tempClosureSet:
                        if rule not in closureSet:
                                closureSet.append(rule)
        return closureSet


def compute_GOTO(state):
        global statesDict, stateCount

        # find all symbols on which we need to
        # make function call - GOTO
        generateStatesFor = []
        for rule in statesDict[state]:
                # if rule is not "Handle"
                if rule[1][-1] != '.':
                        indexOfDot = rule[1].index('.')
                        dotPointsHere = rule[1][indexOfDot + 1]
                        if dotPointsHere not in generateStatesFor:
                                generateStatesFor.append(dotPointsHere)

        # call GOTO iteratively on all symbols pointed by dot
        if len(generateStatesFor) != 0:
                for symbol in generateStatesFor:
                        GOTO(state, symbol)
        return


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

        # newState - stores processed new state
```

```
newState = []
for rule in statesDict[state]:
        indexOfDot = rule[1].index('.')
        if rule[1][-1] != '.':
                if rule[1][indexOfDot + 1] == \
                                charNextToDot:
                        # swapping element with dot,
                        # to perform shift operation
                        shiftedRule = copy.deepcopy(rule)
                        shiftedRule[1][indexOfDot] = \
                                shiftedRule[1][indexOfDot + 1]
                        shiftedRule[1][indexOfDot + 1] = '.'
                        newState.append(shiftedRule)

# add closure rules for newState
# call findClosure function iteratively
# - on all existing rules in newState

# addClosureRules - is used to store
# new rules temporarily,
# to prevent concurrent modification error
addClosureRules = []
for rule in newState:
        indexDot = rule[1].index('.')
        # check that rule is not "Handle"
        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.append(rule)

# add closure result to newState
for rule in addClosureRules:
        newState.append(rule)

# find if newState already present
# in Dictionary
stateExists = -1
for state_num in statesDict:
        if statesDict[state_num] == newState:
                stateExists = state_num
                break
```

```
        # stateMap is a mapping of GOTO with
        # its output states
        if stateExists == -1:

                # if newState is not in dictionary,
                # then create new state
                stateCount += 1
                statesDict[stateCount] = newState
                stateMap[(state, charNextToDot)] = stateCount
        else:

                # if state repetition found,
                # assign that previous state number
                stateMap[(state, charNextToDot)] = stateExists
        return


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

        # run loop till new states are getting added
        while (len(statesDict) != prev_len):
                prev_len = len(statesDict)
                keys = list(statesDict.keys())

                # make compute_GOTO function call
                # on all states in dictionary
                for key in keys:
                        if key not in called_GOTO_on:
                                called_GOTO_on.append(key)
                                compute_GOTO(key)
        return

# 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
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 epsilon in result apply rule
                                        # - (A->aBX)- follow of -
                                        # - follow(B)=(first(X)-{ep}) U follow(A)
                                        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 createParseTable(statesDict, stateMap, T, NT):
        global separatedRulesList, diction

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

        # create empty table
        Table = []
        tempRow = []
```

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

# make shift and GOTO entries in table
for entry in stateMap:
        state = entry[0]
        symbol = entry[1]
        # get index
        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]} "

# start REDUCE procedure

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

# start REDUCE procedure
# format for follow computation
addedR = f"{separatedRulesList[0][0]} -> " \
        f"{separatedRulesList[0][1][1]}"
rules.insert(0, addedR)
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

        # find 'handle' items and calculate follow.
        for stateno in statesDict:
                for rule in statesDict[stateno]:
                        if rule[1][-1] == '.':

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

                                                # put Rn in those ACTION symbol columns,
                                                # who are in the follow of
                                                # LHS of current Item.
                                                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] =\

Table[stateno][index]+f"R{key} "

        # printing table
        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]}")

# *** MAIN *** - Driver Code

# uncomment any rules set to test code
# follow given format to add -
# user defined grammar rule set
# rules section - *START*

# example sample set 01
rules = ["E -> E + T | T",
                "T -> T * F | F",
                "F -> ( E ) | id"
                ]
nonterm_userdef = ['E', 'T', 'F']
term_userdef = ['id', '+', '*', '(', ')']
start_symbol = nonterm_userdef[0]

# example sample set 02
# rules = ["S -> a X d | b Y d | a Y e | b X e",
#                  "X -> c",
#                  "Y -> c"
#                  ]
# nonterm_userdef = ['S','X','Y']
# term_userdef = ['a','b','c','d','e']
# start_symbol = nonterm_userdef[0]

# rules section - *END*
print("\nOriginal grammar input:\n")
for y in rules:
        print(y)

# print processed rules
print("\nGrammar after Augmentation: \n")
separatedRulesList = \
        grammarAugmentation(rules,
                                                nonterm_userdef,
```

```
                                                start_symbol)
printResult(separatedRulesList)

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

# use statesDict to store the states
# use stateMap to store GOTOs
statesDict = {}
stateMap = {}

# add first state to statesDict
# and maintain stateCount
# - for newState generation
statesDict[0] = I0
stateCount = 0

# computing states by GOTO
generateStates(statesDict)

# print goto states
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)

# "follow computation" for making REDUCE entries
diction = {}

# call createParseTable function
createParseTable(statesDict, stateMap,
                                term_userdef,
                                nonterm_userdef)
```

Output:

```
Original grammar input:

E -> E + T | T
T -> T * F | F
F -> ( E ) | id
```

```
Result of GOTO computation:

GOTO ( I0 , E ) = I1
GOTO ( I0 , T ) = I2
GOTO ( I0 , F ) = I3
GOTO ( I0 , ( ) = I4
GOTO ( I0 , id ) = I5
GOTO ( I1 , + ) = I6
GOTO ( I2 , * ) = I7
GOTO ( I4 , E ) = I8
GOTO ( I4 , T ) = I2
GOTO ( I4 , F ) = I3
GOTO ( I4 , ( ) = I4
GOTO ( I4 , id ) = I5
GOTO ( I6 , T ) = I9
GOTO ( I6 , F ) = I3
GOTO ( I6 , ( ) = I4
GOTO ( I6 , id ) = I5
GOTO ( I7 , F ) = I10
GOTO ( I7 , ( ) = I4
GOTO ( I7 , id ) = I5
GOTO ( I8 , ) ) = I11
GOTO ( I8 , + ) = I6
GOTO ( I9 , * ) = I7
```

```
SLR(1) parsing table:

    |   id      +       *       (       )       $       E       T       F

 I0     S5                      S4                      1       2       3
 I1             S6                              Accept
 I2             R2      S7              R2      R2
 I3             R4      R4              R4      R4
 I4     S5                      S4                      8       2       3
 I5             R6      R6              R6      R6
 I6     S5                      S4                              9       3
 I7     S5                      S4                              10
 I8             S6                      S11
 I9             R1      S7              R1      R1
I10             R3      R3              R3      R3
I11             R5      R5              R5      R5

[Done] exited with code=0 in 0.09 seconds
```