|  |
| --- |
| **Experiment-6: Implement SLR:**  **Code:**  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 findClosure(input\_state, dotSymbol):  global start\_symbol, separatedRulesList, statesDict  closureSet = []  if dotSymbol == start\_symbol:  for rule in separatedRulesList:  if rule[0] == dotSymbol:  closureSet.append(rule)  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:  tempClosureSet.append(in\_rule)  for rule in tempClosureSet:  if rule not in closureSet:  closureSet.append(rule)  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.append(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][indexOfDot] = shiftedRule[1][indexOfDot + 1]  shiftedRule[1][indexOfDot + 1] = '.'  newState.append(shiftedRule)  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.append(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]  elif rule[0] == '#':  return '#'  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(indivRes)  if '#' not in fres:  return fres  else:  fres.remove('#')  if len(rule) > 1:  ansNew = first(rule[1:])  if ansNew != None:  if type(ansNew) is list:  return fres + ansNew  else:  return fres + [ansNew]  else:  return fres  fres.append('#')  return fres  def follow(nt):  global start\_symbol, rules, nonterm\_userdef, term\_userdef, diction, firsts, follows  solset = set()  if nt == start\_symbol:  solset.add('$')  for curNT in diction:  rhs = diction[curNT]  for subrule in rhs:  if nt in subrule:  while nt in subrule:  index\_nt = subrule.index(nt)  subrule = subrule[index\_nt + 1:]  if len(subrule) != 0:  res = first(subrule)  if '#' in res:  res.remove('#')  ansNew = follow(curNT)  if ansNew != None:  if type(ansNew) is list:  res += ansNew  else:  res += [ansNew]  else:  res = res  else:  if nt != curNT:  res = follow(curNT)  if res is not None:  if type(res) is list:  for g in res:  solset.add(g)  else:  solset.add(res)  return list(solset)  def createParseTable(statesDict, stateMap, T, NT):  rows = list(statesDict.keys())  cols = T + ['$'] + 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] += f"{stateMap[entry]} "  elif symbol in T:  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]} -> {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] += 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]} -> {' '.join(rule[1])}")  def printAllGOTO(diction):  for itr in diction:  print(f"GOTO ( I{itr[0]} , {itr[1]} ) = I{stateMap[itr]}")  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]  print("\nOriginal grammar input:\n")  for y in rules:  print(y)  print("\nGrammar after Augmentation: \n")  separatedRulesList = grammarAugmentation(rules, nonterm\_userdef, 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 =  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:** |