|  |
| --- |
| **Experiment-8: Implement LALR:**  **Code:**  from grammar import \*  #Semantic specific classes  class CHILD:  def \_init\_(self, ind):  self.index = ind  class Token:  def \_init\_(self, type, \*value):  self.type = type  self.value = None  if len(value) == 1:  self.value = value[0]  def isTerminal(self):  return True  def \_repr\_(self):  return 'Token(' + self.type + ')'  class InternalTree(Token):  def \_init\_(self, type, data):  self.type = type  self.data = data  def isTerminal(self):  return False  def \_repr\_(self):  return "Node(" + self.type + ')'  nodeCounter = 0  def printNode(gNode):  nodeId = gNode.id  node = gNode.data  print("Node #" + str(nodeId))    for prod in node:  l = " ".join([i if i != None else 'Eps' for i in prod[1]])  r = " ".join([i if i != None else 'Eps' for i in prod[2]])  la = ' | '.join([i if i != None else 'Eps' for i in prod[3]])  print(prod[0], '->', l + ' . ' + r, ',', la)  print()  class GraphNode:  def \_init\_(self):  self.transitions = {}  self.data = None  self.id = None  def addNeigh(self, symbol, gNode):  self.transitions[symbol] = gNode  def addData(self, data):  self.data = data  class LALRParser:  def \_init\_(self, grammar, start, evalSemantic=False):  if not start:  self.start = "S"  else:  self.start = start  self.grammar = grammar  self.graphNodes = {}  self.nodeCount = 0  self.rowCount = 0  self.evalSemantic = evalSemantic  # dictionaries to map rules with an index and vice-versa. used to handle reduce ids  self.ruleDict = {}  self.invertedRuleDict = {}  ind = 1  for symbol, rules in self.grammar.productions.items():  for rule in rules:  t = self.hashRule(symbol, rule)  self.ruleDict[t] = ind  ind += 1  self.mergedNodes = {} # map merged nodes (same LR(0) items) to ids  self.table = None # parser table  self.hasConflicts = False  self.createParser() # generate parser, graph + table + compression  def createParser(self):  node1 = []  for right in self.grammar.productions[self.start]:  node1.append((self.start, [], right, {"$"}) )    graph = self.expand(node1)  self.findNodesToMerge(graph, [])  nodeList = [ v for k, v in self.mergedNodes.items()]  nodeList = sorted(nodeList, key=lambda node: node.id)  self.table = [{i:[] for i in self.grammar.terminals + self.grammar.nonterminals} for j in range( len(nodeList) )]  self.createTable(graph, [])  self.invertedRuleDict = {v:k for k, v in self.ruleDict.items()}    self.hasConflicts = self.fixConflicts()  self.printTable()  def fixConflicts(self):  conflict = False  for i in range(len(self.mergedNodes)):  for j in self.grammar.terminals + self.grammar.nonterminals:  if len(self.table[i][j]) > 1:  l = self.table[i][j]  if len(l) == 2:  if l[0][0] == 'shift' or l[1][0] == 'shift': # shift-reduce, can be solved  r = None  s = None  if l[0][1] == 'reduce':  r = l[0]  s = l[1]  else:  r = l[1]  s = l[0]  #print(self.invertedRuleDict[ r[1] ])  if len(self.invertedRuleDict[ r[1] ][1] ) > 1:  opReduce = self.invertedRuleDict[ r[1] ] [1][-2] # get r[1] (reduce index), then get second element (right part of the production), then get the operator  opShift = j  #print('reduce op', opReduce)  if self.grammar.assoc != None and opReduce == opShift:  if opReduce in self.grammar.assoc:  if self.grammar.assoc[opReduce] == 'left':  self.table[i][j] = r  else:  self.table[i][j] = s  continue  if self.grammar.precedence != None:  if (opReduce, opShift) in self.grammar.precedence:  self.table[i][j] = r  continue  elif (opShift, opReduce) in self.grammar.precedence:  self.table[i][j] = s  continue  # TODO: Automatic shift, fix this  self.table[i][j] = s  continue  conflict = True  print('conflict not solved', self.table[i][j])  elif len(self.table[i][j]) == 1:  self.table[i][j] = self.table[i][j][0]  else:  self.table[i][j] = None  return conflict  def getNextNodeId(self):  a = self.nodeCount  self.nodeCount += 1  return a  def hashRule(self, symbol, right):  return (symbol, tuple(right) )  def hashLR0item(self, node): # create hashable element with all rules of a LR(0) item (ignoring lookaheads), useful to merge nodes  s = set()  for rule in node:  s.add( (rule[0], tuple(rule[1]), tuple(rule[2])) )  return frozenset(s)  def hashLR1item(self, node): # create hashable element with all rules of a LR(0) item (ignoring lookaheads), useful to merge nodes  s = set()  for rule in node:  s.add( (rule[0], tuple(rule[1]), tuple(rule[2]), tuple(rule[3])) )  return frozenset(s)  def findNodesToMerge(self, gNode, vis):  if gNode in vis:  return  vis.append(gNode)  m = self.hashLR0item(gNode.data)  if m not in self.mergedNodes:  self.mergedNodes[m] = gNode  gNode.rowId = self.rowCount  print('row:', self.rowCount, 'nodeId:', gNode.id)  self.rowCount += 1  else:  gNode.rowId = self.mergedNodes[m].rowId  for tr in gNode.transitions:  self.findNodesToMerge(gNode.transitions[tr], vis)  def createTable(self, gNode, vis):  if gNode in vis:  return  vis.append(gNode)  for tr in gNode.transitions:  nNode = gNode.transitions[tr]  nId = nNode.rowId  cell = self.table[gNode.rowId][tr]  if tr in self.grammar.terminals:  if ('shift', nId) not in cell:  cell.append(('shift', nId))  else:  if ('goto', nId) not in cell:  cell.append(('goto', nId))  self.createTable(nNode, vis)  for rule in gNode.data:  if len(rule[2]) == 0: # finished production, reduce it  t = self.hashRule(rule[0], rule[1])  ind = self.ruleDict[t]  for la in rule[3] - {None}:  cell = self.table[gNode.rowId][la]  if ('reduce', ind) not in cell:  cell.append(('reduce', ind))  def printTable(self):  print('\t' + '\t'.join([i[0:6] for i in self.grammar.terminals + self.grammar.nonterminals]))  id = 0  for row in self.table:  l = []  for k in self.grammar.terminals + self.grammar.nonterminals:  item = []  if row[k] != None:  if not isinstance(row[k], list): # if row is not a list, iterate over a list with only one element  r = [row[k]]  else:  r = row[k]  for i in r:  if len(i) == 0:  item.append('')  elif i[0] == 'shift':  item.append('S' + str(i[1]))  elif i[0] == 'reduce':  item.append('R' + str(i[1]))  else:  item.append(str(i[1]))  item = '/'.join(item)  else:  item = ''  l.append(item)  print(str(id) + ':\t' + '\t'.join(l))  id += 1  def advance(self, prod):  # generate next node (move dot forward in all productions)  right = []  if len(prod[2]) == 1:  right = []  else:  right = prod[2][1:]  return [prod[0], prod[1] + [prod[2][0]], right, prod[3] | set()]  def closure(self, rules, remainingSymbols, symbol, lookahead):  for right in self.grammar.productions[symbol]:  s = (symbol, tuple(right))    #does this complete rule exist? if not, added it with lookahead  if s in rules and len(lookahead - rules[s]) == 0: # rule exists, and lookahead is included in the existing one  continue  else: # rule doesn't exist, or existing lookahead doesn't include the new one  if s not in rules:  rules[s] = set()  rules[s] |= lookahead  # apply closure to left most symbol of production?  if len(right) > 0 and right[0] not in self.grammar.terminals:  newLA = self.grammar.FIRST(right[1:], set())  if len(newLA) == 0 or None in newLA:  newLA |= lookahead  nSymbol = right[0]  if nSymbol not in remainingSymbols:  remainingSymbols[nSymbol] = set()  remainingSymbols[nSymbol] |= newLA  def getLookahead(self, right, lookahead):  la = self.grammar.FIRST(right[1:], set())  if len(la) == 0 or None in la:  la |= lookahead  return la  def expand(self, node):  graphNode = GraphNode()  remainingSymbols = {} # dict that maps closure symbol to look-ahead symbols set  expanded = set() # (symbol, right, lookahead)  # for current rules, get all symbols to apply closure and respective look-ahead  for rule in node:  symbol, left, right, lookahead = rule  if len(right) > 0 and right[0] not in self.grammar.terminals:  if right[0] not in remainingSymbols:  remainingSymbols[right[0]] = set()  remainingSymbols[right[0]] |= self.getLookahead(right, lookahead)  # apply closure to all symbols, keep adding to the queue the left most symbols that belong to the new rules (or have new lookaheads)  newRules = {}  while len(remainingSymbols) > 0:  symbol, lookahead = pop(remainingSymbols)  self.closure(newRules, remainingSymbols, symbol, lookahead)  for k in newRules:  node.append([k[0], [], k[1], newRules[k]])  # generate transitions for each production, joining together those for the same input symbol  transitions = {}  for prod in node:  if len(prod[2]) > 0:  if prod[2][0] not in transitions:  transitions[prod[2][0]] = []  transitions[prod[2][0]].append(self.advance(prod))  # update graph node, making sure that there are no repeated nodes (same rules with same lookaheads)  nodeHash = self.hashLR1item(node)  if nodeHash in self.graphNodes:  return self.graphNodes[nodeHash]  graphNode.id = self.getNextNodeId()  graphNode.addData(node)  self.graphNodes[nodeHash] = graphNode  printNode(graphNode)  # recursively generate nodes connected to this one  for tr in transitions:  print('on', tr)  gNode = self.expand(transitions[tr]) # will return an existing node if the generated node matches one  graphNode.addNeigh(tr, gNode)  return graphNode  def getNodeData(self, node):  if isinstance(node, InternalTree):  return node.data  else:  return node  def parse(self, inp):  if self.hasConflicts:  raise ValueError('Conflicts must be solved first!')  queue = []  queue.append(0)    inputSize = len(inp)  t = inp[0]  inp.pop(0)  solved = False  while True:  op = self.table[queue[-1]][t.type]  #print(queue, t.type, op)    if op == None:  l = inputSize - len(inp)  raise ValueError('Cannot parse input! Shifted ' + str(l) + ' tokens')  if op[0] == 'shift':  queue.append(t)  queue.append(op[1])  t = inp[0]  inp.pop(0)  if op[0] == 'reduce':  ruleInd = op[1]  rule = self.invertedRuleDict[ruleInd]  symbol, prod = rule  n = len(prod) \* 2    if n > 0:  popped = queue[-n:]  queue = queue[:-n]  else:  popped = []  children = [popped[i] for i in range(len(popped)) if i%2 == 0]    data = None  v = None  if rule in self.grammar.annotatedRules:  s = self.grammar.annotatedRules[rule]  if self.evalSemantic:  def childToken(n):  return self.getNodeData(children[n])    l = locals()  r = exec(s, {}, l)  data = l['v']  else:  if type(s) is list and len(s) > 0 and callable(s[0]):  args = [self.getNodeData(children[i.index]) if isinstance(i, CHILD) else i for i in s[1:]]  data = s[0](\*args)  elif not (type(s) is list):  if isinstance(s, CHILD):  data = self.getNodeData(children[s.index])  else:  data = s  else:  data = [self.getNodeData(children[i.index]) if isinstance(i, CHILD) else i for i in s]    node = InternalTree(symbol, data)    queue.append(node)  r = queue[-2]  c = queue[-1].type  if len(queue) >= 2 and queue[-2] == 0 and queue[-1].type == 'S': # TODO: Fix this  solved = True  break  queue.append(self.table[r][c][1])  tree = queue[-1]  return tree.data  def pop(d):  for i in d:  s = (i, d[i])  del d[i]  return s  def isBlank(s):  for i in s:  if not (i == '\n' or i == '\r' or i == ' '):  return False  return True  class Grammar:  def \_init\_(self, terminals, rules, semantic=None):  self.terminals = terminals  self.productions = {}  self.nonterminals = []  self.assoc = None  self.precedence = None  self.annotatedRules = {}  i = 0  if semantic != None:  for rule in rules:  self.annotatedRules[rule] = semantic[i]  i+=1  self.rules = rules  for rule in rules:  if rule[0] not in self.productions:  self.productions[rule[0]] = []  self.productions[rule[0]].append(rule[1])  for i in self.productions:  if i not in self.nonterminals:  self.nonterminals.append(i)  self.\_first = {}  for s in self.productions:  self.\_first[s] = None  for s in self.productions:  if self.\_first[s] == None:  self.FIRST([s], set())  def FIRST(self, rule, stack):  if len(rule) == 0:  return {None}  if rule[0] in stack:  return set()  total = set()  if rule[0] in self.terminals:  return {rule[0]}    total = set()  s2 = stack | {rule[0]}  #print(self.productions, rule[0])  for p in self.productions[rule[0]]:  total |= self.FIRST(p, s2)  self.\_first[rule[0]] = total | set()  if None in total and len(rule) > 1:  res = self.FIRST(rule[1:], stack)  total |= res  if None not in res:  total.remove(None)  return total  def setAssoc(self, assoc):  self.assoc = assoc  def setPrecedence(self, precedence):  self.precedence = precedence  def readGrammar(f, semantic=None):  f = open(f)  lines = f.readlines()  terminals = lines[0].split()  terminals.append('$')  lines = lines[1:]  operatorPrecedence = [] # pair of tuples, the first element has higher precedence  operatorAssoc = {} # maps each operator to a string, with left or right  rules = []  for line in lines:  if isBlank(line):  continue  special = line.split()  if special[0][0] == '%':  if special[0] == '%left' or special[0] == '%right':  r = special[0][1:]  for op in special[1:]:  operatorAssoc[op] = r  if special[0] == '%priority':  special.pop(0)  for op1 in range(len(special)):  for op2 in range(op1+1, len(special)):  operatorPrecedence.append((special[op1], special[op2]))  else:  left, right = line.split('->')  left = left.strip()  r = right.split('|')  for prod in r:  l = prod.split()  l = [i for i in l if i != 'eps']  rules.append( (left, tuple(l)) )  #print(operatorPrecedence, operatorAssoc)  #print(terminals, rules)  g = Grammar(terminals, rules, semantic=semantic)  g.setPrecedence(operatorPrecedence)  g.setAssoc(operatorAssoc)  return g  **OUTPUT:**  Parsing Table:  -----------------------------------  | State | id | + | \* | ( | ) | $ |  -----------------------------------  | 0 | shift 4| shift 5| | s1 | | |  | 1 | | shift 3| | | | |  | 2 | | | | | |accept |  | 3 | reduce E -> T | reduce E -> T | shift 8 | reduce E -> T | reduce E -> T | reduce E -> T |  | 4 | shift 4 | | shift 8 | s1 | reduce F -> id | reduce F -> id |  | 5 | shift 4 | | | s1 | | |  | 6 | shift 7 | | shift 8 | | | |  | 7 | reduce F -> id | | shift 8 | | | |  | 8 | shift 9 | | | | reduce T -> F | reduce T -> F |  | 9 | reduce F -> id | | | | reduce T -> F | reduce T -> F |  -----------------------------------  grammar = {  (0, 'id'): 'shift 4',  (0, '('): 'shift 5',  (4, '+'): 'shift 6',  (5, 'id'): 'shift 4',  (6, 'id'): 'shift 7',  (7, ')'): 'reduce F -> id',  (3, 'id'): 'reduce E -> T',  (1, '+'): 'shift 3',  (2, '\*'): 'shift 8',  (8, 'id'): 'shift 9',  (9, '$'): 'reduce F -> id',  (8, ')'): 'reduce T -> F',  (4, '\*'): 'shift 8',  (6, '\*'): 'shift 8',  (3, '+'): 'reduce E -> T',  (3, '\*'): 'reduce T -> F',  (0, '$'): 'accept',  }  input\_string = "id + id \* id" |