***COMPILER CONSTRUCTION***

**UCS802**

**LAB ASSIGNEMENT 2**

**Design a SLR Parser**

Submitted By:

|  |  |
| --- | --- |
| **Akshita Gupta** | **102103741** |

**BE Fourth Year- Group: 4CO’26**

**Date:- 10-11-2024**

Submitted To **Dr. Shashank Singh** Assistant Professor



**Computer Science and Engineering Department Thapar Institute of Engineering and Technology, Patiala**

**July-December 2024**

**Q1. Design a SLR parser for the grammar given below:**

**E→E+T/T**

**T→T\*F/F F→(E)/id**

**This will involve three steps:**

* **Generate the Set of Items (5 Marks) (3 Lab of 2 Hrs.)**
* **Generate the Action and GOTO table (5 marks) (3 Labs of 2 Hrs. each)**

# CODE



import copy

def augment\_grammar(rules, nonterminals, start\_symbol):new\_rules = [] new\_char = start\_symbol + "'"

while new\_char in nonterminals:new\_char

+= "'"

new\_rules.append([new\_char, ['.', start\_symbol]]) for rule in rules:

lhs, rhs = map(str.strip, rule.split("->"))

multirhs = [r.strip().split() for r in rhs.split('|')]

for rhs1 in multirhs: rhs1.insert(0, '.') new\_rules.append([lhs, rhs1])

return new\_rules

def find\_closure(input\_state, dot\_symbol):

global start\_symbol, separated\_rules\_list, states\_dict closure\_set = []

if dot\_symbol == start\_symbol:

for rule in separated\_rules\_list:if rule[0] == dot\_symbol:

closure\_set.append(rule)

else:

closure\_set = input\_state prev\_len = -1

while prev\_len != len(closure\_set):prev\_len = len(closure\_set) temp\_closure\_set = []

for rule in closure\_set: index\_of\_dot = rule[1].index('.')

if rule[1][-1] != '.':

dot\_points\_here = rule[1][index\_of\_dot + 1]

temp\_closure\_set:

for in\_rule in separated\_rules\_list:

if dot\_points\_here == in\_rule[0] and in\_rule not in temp\_closure\_set.append(in\_rule)

for rule in temp\_closure\_set: if rule not in closure\_set:



closure\_set.append(rule) return closure\_set

def compute\_goto(state):

global states\_dict, state\_count generate\_states\_for = []

for rule in states\_dict[state]:if rule[1][-1]

!= '.':

index\_of\_dot = rule[1].index('.') dot\_points\_here = rule[1][index\_of\_dot + 1]

if dot\_points\_here not in generate\_states\_for: generate\_states\_for.append(dot\_points\_here) if len(generate\_states\_for) != 0:

for symbol in generate\_states\_for:goto(state,

symbol)

def goto(state, char\_next\_to\_dot):

global states\_dict, state\_count, state\_map new\_state = []

for rule in states\_dict[state]: index\_of\_dot = rule[1].index('.')

if rule[1][-1] != '.':

if rule[1][index\_of\_dot + 1] == char\_next\_to\_dot:shifted\_rule = copy.deepcopy(rule)

shifted\_rule[1][index\_of\_dot] = shifted\_rule[1][index\_of\_dot +

1]

shifted\_rule[1][index\_of\_dot + 1] = '.' new\_state.append(shifted\_rule)



add\_closure\_rules = []

for rule in new\_state:

index\_dot = rule[1].index('.')

if rule[1][-1] != '.':

closure\_res = find\_closure(new\_state, rule[1][index\_dot + 1])

for rule in closure\_res:

if rule not in add\_closure\_rules and rule not in new\_state: add\_closure\_rules.append(rule)

for rule in add\_closure\_rules: new\_state.append(rule)

state\_exists = -1

for state\_num in states\_dict:

if states\_dict[state\_num] == new\_state:state\_exists = state\_num

break

if state\_exists == -1: state\_count += 1

states\_dict[state\_count] = new\_state state\_map[(state, char\_next\_to\_dot)] = state\_count

else:

state\_map[(state, char\_next\_to\_dot)] = state\_exists

def generate\_states(states\_dict):prev\_len = -1 called\_goto\_on = []

while len(states\_dict) != prev\_len:prev\_len = len(states\_dict) keys = list(states\_dict.keys())

for key in keys:

if key not in called\_goto\_on: called\_goto\_on.append(key) compute\_goto(key)

def first(rule):

global rules, nonterminals, terminals, dictionary, firsts if len(rule) != 0 and (rule is not None):

if rule[0] in terminals:return rule[0]

elif rule[0] == '#':return '#'

if len(rule) != 0:

if rule[0] in list(dictionary.keys()):fres = [] rhs\_rules = dictionary[rule[0]]

for itr in rhs\_rules: indiv\_res = first(itr)

if type(indiv\_res) is list: fres.extend(indiv\_res)

else:

fres.append(indiv\_res)

if '#' not in fres:return fres else:

fres.remove('#') if len(rule) > 1:

ans\_new = first(rule[1:])

if ans\_new is not None:

if type(ans\_new) is list:return fres

+ ans\_new

else:

else:

return fres + [ans\_new]

return fres



fres.append('#')return fres

def follow(nt):

global start\_symbol, rules, nonterminals, terminals, dictionary, firsts,follows sol\_set = set()

if nt == start\_symbol: sol\_set.add('$')

for cur\_nt in dictionary: rhs = dictionary[cur\_nt]

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: new\_list = [] res.remove('#')

ans\_new = follow(cur\_nt)

if ans\_new is not None:

if type(ans\_new) is list: new\_list = res

+ ans\_new

else:

else:

new\_list = res + [ans\_new]

new\_list = res

else:

res = new\_list

if nt != cur\_nt:

res = follow(cur\_nt)

if res is not None:

if type(res) is list:

sol\_set.update(res)

return list(sol\_set)

else:

sol\_set.add(res)



def create\_parse\_table(states\_dict, state\_map, terminals, nonterminals):global separated\_rules\_list, dictionary

rows = list(states\_dict.keys())

cols = terminals + ['$'] + nonterminals

table = [['' for \_ in range(len(cols))] for \_ in range(len(rows))]

for entry in state\_map:state = entry[0] symbol = entry[1]

a = rows.index(state) b = cols.index(symbol)

if symbol in nonterminals:



table[a][b] += f"{state\_map[entry]} "elif symbol in terminals:

table[a][b] += f"S{state\_map[entry]} "

numbered = {} key\_count = 0

for rule in separated\_rules\_list: temp\_rule = copy.deepcopy(rule) temp\_rule[1].remove('.') numbered[key\_count] = temp\_rule key\_count += 1

added\_r = f"{separated\_rules\_list[0][0]} ->

{separated\_rules\_list[0][1][1]}" rules.insert(0, added\_r)

for rule in rules:

lhs, rhs = map(str.strip, rule.split("->")) rhs\_rules = [r.strip().split() for r in rhs.split('|')]

for i in range(len(rhs\_rules)):

rhs\_rules[i] = [r.strip() for r in rhs\_rules[i]] dictionary[lhs] = rhs\_rules

for stateno in states\_dict:

for rule in states\_dict[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")



for i, row in enumerate(table):frmt1 = "{:>8}" \* len(row)

print(f"{{:>3}} {frmt1.format(\*row)}".format('I'+str(i)))

def print\_result(rules):for rule in rules:

print(f"{rule[0]} -> {' '.join(rule[1])}")

def print\_all\_goto(diction):for itr in diction:

print(f"GOTO ( I{itr[0]} , {itr[1]} ) = I{state\_map[itr]}") # \*\*\* MAIN \*\*\* - Driver Code

rules\_input = [

"E -> E + T | T",

"T -> T \* F | F",

"F -> ( E ) | id"

]

nonterminals\_input = ['E', 'T', 'F'] terminals\_input = ['id', '+', '\*', '(', ')']start\_symbol\_input = nonterminals\_input[0]

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

print(rule\_input)

print("\nGrammar after Augmentation: \n")

separated\_rules\_list = augment\_grammar(rules\_input, nonterminals\_input,start\_symbol\_input) print\_result(separated\_rules\_list)

start\_symbol\_input = separated\_rules\_list[0][0] print("\nCalculated closure: I0\n")

I0 = find\_closure(0, start\_symbol\_input)

print\_result(I0)

states\_dict = {} state\_map = {} states\_dict[0] = I0 state\_count = 0

generate\_states(states\_dict) print("\nStates Generated: \n")

for state\_num, state\_rules in states\_dict.items(): print(f"State = I{state\_num}") print\_result(state\_rules)

print()

print("Result of GOTO computation:\n") print\_all\_goto(state\_map)

dictionary = {}

create\_parse\_table(states\_dict, state\_map, terminals\_input, nonterminals\_input)

# OUTPUT





