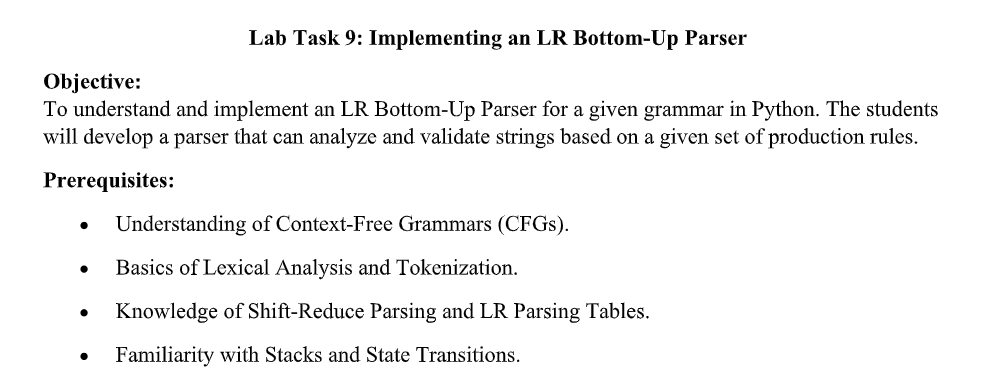
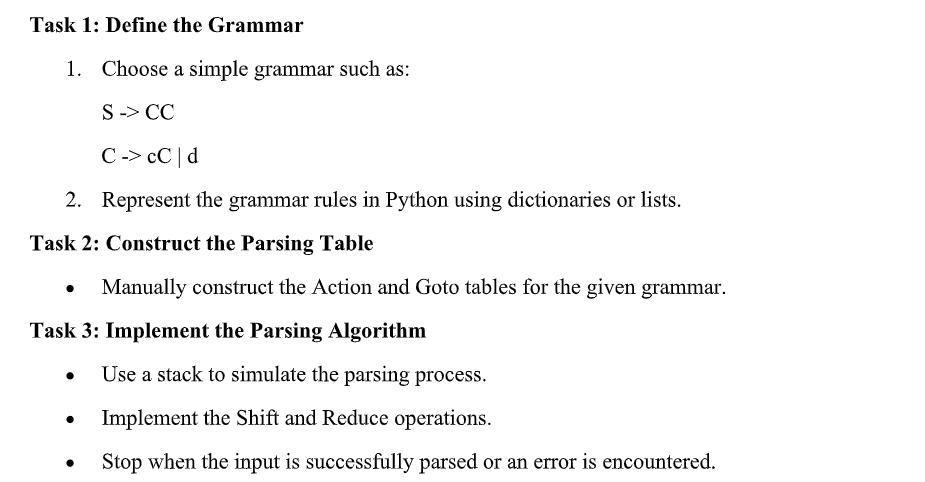
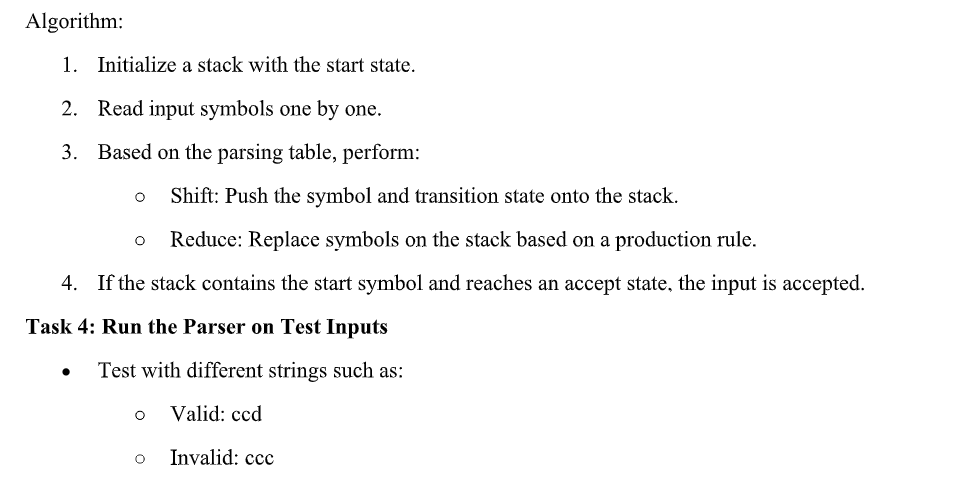
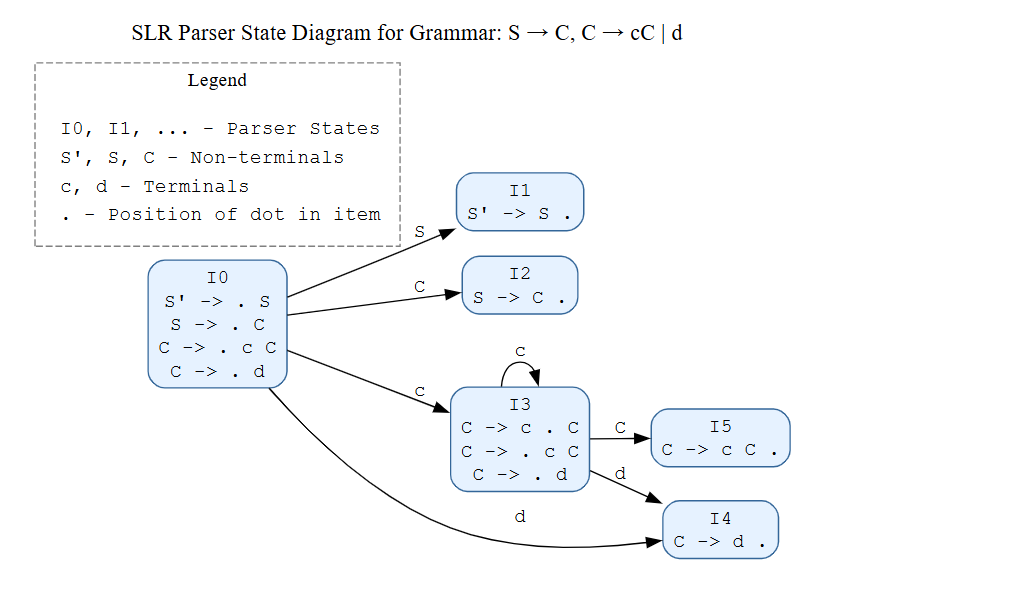


|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | | | | | | | | |
|  | | COMPILER DESIGN LAB | | | | |  | |
|  |  | | | | | | |  |
|  | | | |  |  | | | |
|  | | | | SUBMITTED BY: |  | | | |
|  | | | | SAMAR MITTALMIS: 112315159FACULTY:DR MAYANK LOVANSHI |  | | | |
|  | | |  | | |  | | |









## PYTHON CODE :

import copy

from collections import defaultdict

class SLRParser:

    def \_\_init\_\_(self, grammar\_rules, non\_terminals, terminals, start\_symbol):

        self.rules = grammar\_rules

        self.non\_terminals = non\_terminals

        self.terminals = terminals

        self.start\_symbol = start\_symbol

        self.augmented\_rules = []

        self.states\_dict = {}

        self.state\_map = {}

        self.state\_count = 0

        self.diction = defaultdict(list)

        self.firsts\_cache = {}

        self.follows\_cache = {}

        self.parse\_table = None

        self.numbered\_rules = {}

    def parse(self):

        print("\nOriginal grammar input:\n")

        for rule in self.rules:

            print(rule)

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

        self.augmented\_rules = self.augment\_grammar()

        self.print\_result(self.augmented\_rules)

        self.start\_symbol = self.augmented\_rules[0][0]

        print("\nCalculated closure: I0\n")

        I0 = self.find\_closure([], self.start\_symbol)

        self.print\_result(I0)

        self.states\_dict[0] = I0

        self.generate\_states()

        print("\nStates Generated: \n")

        for st in self.states\_dict:

            print(f"State = I{st}")

            self.print\_result(self.states\_dict[st])

            print()

        self.create\_parse\_table()

    def augment\_grammar(self):

        new\_rules = []

        new\_start = f"{self.start\_symbol}'"

        while new\_start in self.non\_terminals:

            new\_start += "'"

        new\_rules.append([new\_start, ['.', self.start\_symbol]])

        for rule in self.rules:

            lhs, rhs = [x.strip() for x in rule.split("->")]

            for subrule in rhs.split('|'):

                rhs\_items = subrule.strip().split()

                # Add dot at beginning

                rhs\_items.insert(0, '.')

                new\_rules.append([lhs, rhs\_items])

        return new\_rules

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

        if dot\_symbol == self.start\_symbol:

            closure\_set = [rule for rule in self.augmented\_rules if rule[0] == dot\_symbol]

        else:

            closure\_set = input\_state

        prev\_len = -1

        while prev\_len != len(closure\_set):

            prev\_len = len(closure\_set)

            temp\_closure = []

            for rule in closure\_set:

                dot\_index = rule[1].index('.')

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

                    next\_symbol = rule[1][dot\_index + 1]

                    if next\_symbol in self.non\_terminals:

                        for new\_rule in self.augmented\_rules:

                            if new\_rule[0] == next\_symbol and new\_rule not in temp\_closure:

                                temp\_closure.append(new\_rule)

            closure\_set.extend([rule for rule in temp\_closure if rule not in closure\_set])

        return closure\_set

    def compute\_goto(self, state):

        symbols\_after\_dot = set()

        for rule in self.states\_dict[state]:

            if rule[1][-1] != '.':  # If not at end

                dot\_index = rule[1].index('.')

                symbol = rule[1][dot\_index + 1]

                symbols\_after\_dot.add(symbol)

        for symbol in symbols\_after\_dot:

            self.goto(state, symbol)

    def goto(self, state, symbol):

        new\_state = []

        for rule in self.states\_dict[state]:

            dot\_index = rule[1].index('.')

            if rule[1][-1] != '.' and rule[1][dot\_index + 1] == symbol:

                shifted\_rule = copy.deepcopy(rule)

                shifted\_rule[1][dot\_index], shifted\_rule[1][dot\_index + 1] = \

                    shifted\_rule[1][dot\_index + 1], '.'

                new\_state.append(shifted\_rule)

        closure\_additions = []

        for rule in new\_state:

            dot\_index = rule[1].index('.')

            if rule[1][-1] != '.':  # If not at end

                next\_symbol = rule[1][dot\_index + 1]

                if next\_symbol in self.non\_terminals:

                    closure\_result = self.find\_closure(new\_state, next\_symbol)

                    for new\_rule in closure\_result:

                        if new\_rule not in closure\_additions and new\_rule not in new\_state:

                            closure\_additions.append(new\_rule)

        new\_state.extend(closure\_additions)

        state\_exists = -1

        for state\_num, state\_rules in self.states\_dict.items():

            if self.\_compare\_states(state\_rules, new\_state):

                state\_exists = state\_num

                break

        if state\_exists == -1:

            self.state\_count += 1

            self.states\_dict[self.state\_count] = new\_state

            self.state\_map[(state, symbol)] = self.state\_count

        else:

            self.state\_map[(state, symbol)] = state\_exists

    def \_compare\_states(self, state1, state2):

        if len(state1) != len(state2):

            return False

        for rule in state1:

            if rule not in state2:

                return False

        return True

    def generate\_states(self):

        prev\_len = -1

        processed\_states = set()

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

            prev\_len = len(self.states\_dict)

            current\_states = set(self.states\_dict.keys())

            for state in current\_states - processed\_states:

                processed\_states.add(state)

                self.compute\_goto(state)

    def first(self, rule):

        rule\_key = tuple(rule) if isinstance(rule, list) else rule

        if rule\_key in self.firsts\_cache:

            return self.firsts\_cache[rule\_key]

        if not rule:

            return []

        if rule[0] in self.terminals:

            result = [rule[0]]

            self.firsts\_cache[rule\_key] = result

            return result

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

            result = ['#']

            self.firsts\_cache[rule\_key] = result

            return result

        if rule[0] in self.diction:

            result = []

            for subrule in self.diction[rule[0]]:

                first\_set = self.first(subrule)

                if isinstance(first\_set, list):

                    result.extend([x for x in first\_set if x not in result])

                else:

                    if first\_set not in result:

                        result.append(first\_set)

            if '#' in result and len(rule) > 1:

                result.remove('#')

                rest\_first = self.first(rule[1:])

                if rest\_first:

                    if isinstance(rest\_first, list):

                        result.extend([x for x in rest\_first if x not in result])

                    else:

                        if rest\_first not in result:

                            result.append(rest\_first)

                if isinstance(rest\_first, list) and '#' in rest\_first:

                    result.append('#')

            self.firsts\_cache[rule\_key] = result

            return result

        return []

    def follow(self, nt):

        if nt in self.follows\_cache:

            return self.follows\_cache[nt]

        result = set()

        if nt == self.start\_symbol:

            result.add('$')

        for lhs, rhs\_list in self.diction.items():

            for rhs in rhs\_list:

                if nt in rhs:

                    i = 0

                    while i < len(rhs):

                        if rhs[i] == nt:

                            if i < len(rhs) - 1:

                                first\_set = self.first(rhs[i+1:])

                                if isinstance(first\_set, list):

                                    for symbol in first\_set:

                                        if symbol != '#':

                                            result.add(symbol)

                                    if '#' in first\_set and nt != lhs:

                                        follow\_set = self.follow(lhs)

                                        if follow\_set:

                                            if isinstance(follow\_set, list):

                                                result.update(follow\_set)

                                            else:

                                                result.add(follow\_set)

                                else:

                                    if first\_set != '#':

                                        result.add(first\_set)

                            elif nt != lhs:  # Avoid infinite recursion

                                follow\_set = self.follow(lhs)

                                if follow\_set:

                                    if isinstance(follow\_set, list):

                                        result.update(follow\_set)

                                    else:

                                        result.add(follow\_set)

                        i += 1

        self.follows\_cache[nt] = list(result)

        return list(result)

    def create\_parse\_table(self):

        self.\_prepare\_rules\_dict()

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

        cols = self.terminals + ['$'] + self.non\_terminals

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

        for (state, symbol), next\_state in self.state\_map.items():

            row = rows.index(state)

            col = cols.index(symbol)

            if symbol in self.non\_terminals:

                table[row][col] += f"{next\_state} "

            elif symbol in self.terminals:

                table[row][col] += f"S{next\_state} "

        for i, rule in enumerate(self.augmented\_rules):

            rule\_copy = copy.deepcopy(rule)

            if '.' in rule\_copy[1]:  # Remove dot if present

                rule\_copy[1].remove('.')

            self.numbered\_rules[i] = rule\_copy

        for state in self.states\_dict:

            for rule in self.states\_dict[state]:

                if rule[1][-1] == '.':  # If dot at end, it's a reduce item

                    rule\_copy = copy.deepcopy(rule)

                    rule\_copy[1].remove('.')

                    rule\_num = -1

                    for num, r in self.numbered\_rules.items():

                        if r == rule\_copy:

                            rule\_num = num

                            break

                    if rule\_num != -1:

                        follow\_set = self.follow(rule[0])

                        for symbol in follow\_set:

                            col = cols.index(symbol)

                            row = rows.index(state)

                            if rule\_num == 0:  # Accept for augmented start rule

                                table[row][col] = "Accept"

                            else:

                                table[row][col] += f"R{rule\_num} "

        self.parse\_table = {

            'rows': rows,

            'cols': cols,

            'table': table

        }

        print("\nSLR(1) parsing table:\n")

        col\_format = "{:>8}" \* len(cols)

        print(" ", col\_format.format(\*cols), "\n")

        for i, row in enumerate(table):

            row\_format = "{:>8}" \* len(row)

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

    def \_prepare\_rules\_dict(self):

        augmented = f"{self.augmented\_rules[0][0]} -> {self.augmented\_rules[0][1][1]}"

        if augmented not in self.rules:

            self.rules.insert(0, augmented)

        for rule in self.rules:

            lhs, rhs = [x.strip() for x in rule.split("->")]

            for subrule in rhs.split('|'):

                self.diction[lhs].append(subrule.strip().split())

    def print\_result(self, rules):

        for rule in rules:

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

    def parse\_input(self, input\_string):

        if not self.parse\_table:

            print("Parse table not created. Run create\_parse\_table() first.")

            return False

        input\_string = input\_string + '$'

        input\_tokens = list(input\_string)

        stack = [0]

        index = 0  # Current position in input string

        print("\nParsing Input:", input\_string[:-1])

        print()

        print("-" \* 80)

        print("|{0:^18}|{1:^19}|{2:^19}|{3:^18}|".format("Step" ,"Stack","Input","Action"))

        print("-" \* 80)

        step = 1

        while True:

            current\_state = stack[-1]

            current\_symbol = input\_tokens[index]

            try:

                col\_index = self.parse\_table['cols'].index(current\_symbol)

            except ValueError:

                print(f"Error: Symbol '{current\_symbol}' not in grammar")

                return False

            action = self.parse\_table['table'][current\_state][col\_index]

            stack\_str = ' '.join(map(str, stack))

            input\_str = ''.join(input\_tokens[index:])

            print(f"|{step:<18}|{stack\_str:<19}|{input\_str:<19}|{action:<18}|")

            print("-" \* 80)

            if not action:

                print(f"Error: No action defined for state {current\_state} and symbol '{current\_symbol}'")

                print(f"Input string '{input\_string[:-1]}' is not valid according to the grammar")

                return False

            # Process action

            if action == "Accept":

                print(f"\nInput string '{input\_string[:-1]}' accepted!")

                return True

            elif action[0] == 'S':  # Shift

                next\_state = int(action[1:])

                stack.append(current\_symbol)

                stack.append(next\_state)

                index += 1

            elif action[0] == 'R':  # Reduce

                rule\_num = int(action[1:])

                lhs, rhs = self.numbered\_rules[rule\_num]

                if not hasattr(self, 'reductions'):

                    self.reductions = []

                rhs\_str = ' '.join(rhs) if rhs else "ε"

                self.reductions.append(f"{lhs} -> {rhs\_str}")

                for \_ in range(2 \* len(rhs)):

                    stack.pop()

                current\_state = stack[-1]

                stack.append(lhs)

                goto\_col = self.parse\_table['cols'].index(lhs)

                goto\_state = int(self.parse\_table['table'][current\_state][goto\_col])

                stack.append(goto\_state)

            elif ' ' in action:  # Handle multiple actions (conflict)

                print(f"Error: Conflict in parse table: {action}")

                print(f"Input string '{input\_string[:-1]}' cannot be parsed unambiguously")

                return False

            step += 1

if \_\_name\_\_ == "\_\_main\_\_":

    rules = [

        "S -> C C",

        "C -> c C | d"

    ]

    non\_terminals = ['S', 'C']

    terminals = ['c', 'd']

    start\_symbol = 'S'

    parser = SLRParser(rules, non\_terminals, terminals, start\_symbol)

    parser.parse()

    valid\_input = "cdd"

    invalid\_input = "ccc"

    print("\n" + "="\*50)

    print("Testing 1st Input : ")

    parser.parse\_input(valid\_input)

    print("\n" + "="\*50)

    print("Testing 2nd Input : ")

    parser.parse\_input(invalid\_input)

### OUTPUT :



