COMPILER DESIGN LAB

SUBMITTED BY:

SAMAR MITTAL
MIS: 112315159

FACULTY:

DR MAYANK LOVANSHI

Lab Task 9: Implementing an LR Bottom-Up Parser

Objective:

To understand and implement an LR Bottom-Up Parser for a given grammar in Python. The students will develop a parser that can analyze and validate strings based on a given set of production rules.

Prerequisites:

- Understanding of Context-Free Grammars (CFGs).
- Basics of Lexical Analysis and Tokenization.
- Knowledge of Shift-Reduce Parsing and LR Parsing Tables.
- Familiarity with Stacks and State Transitions.

Task 1: Define the Grammar

1. Choose a simple grammar such as:

   S -> CC

   C -> cC | d

2. Represent the grammar rules in Python using dictionaries or lists.

Task 2: Construct the Parsing Table

- Manually construct the Action and Goto tables for the given grammar.

Task 3: Implement the Parsing Algorithm

- Use a stack to simulate the parsing process.
- Implement the Shift and Reduce operations.
- Stop when the input is successfully parsed or an error is encountered.

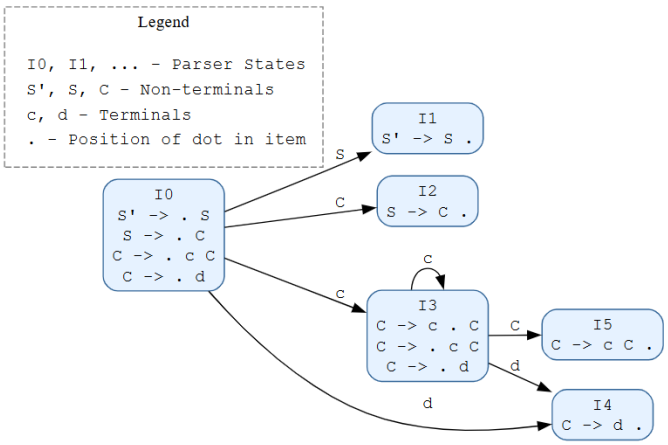
Algorithm:

1. Initialize a stack with the start state.
2. Read input symbols one by one.
3. Based on the parsing table, perform:
   - Shift: Push the symbol and transition state onto the stack.
   - Reduce: Replace symbols on the stack based on a production rule.
4. If the stack contains the start symbol and reaches an accept state, the input is accepted.

Task 4: Run the Parser on Test Inputs

- Test with different strings such as:
  - Valid: ccd
  - Invalid: ccc

SLR Parser State Diagram for Grammar: S → C, C → cC | d



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 :

```
PS C:\Users\Samar Mittal\Desktop\Compiler LAb\lab10> & "C:/Users/Samar Mittal
/Python/Python312/python.exe" "c:/Users/Samar Mittal/Desktop/Compiler LAb/lab

Original grammar input:

S -> C C
C -> c C | d

Grammar after Augmentation:

S' -> . S
S -> . C C
C -> . c C
C -> . d

Calculated closure: I0

S' -> . S
S -> . C C
C -> . c C
C -> . d

States Generated:

State = I0
S' -> . S
S -> . C C
C -> . c C
C -> . d

State = I1
C -> c . C
C -> . c C
C -> . d

State = I2
S' -> S .

State = I3
C -> d .

State = I4
S -> C . C
C -> . c C
C -> . d

State = I5
C -> c C .

State = I6
S -> C C .
```

```
SLR(1) parsing table:

          c       d       $       S       C

  I0      S1      S3                2       4
  I1      S1      S3                        5
  I2                      Accept
  I3      R3      R3      R3
  I4      S1      S3                        6
  I5      R2      R2      R2
  I6                      R1

Testing 1st Input :

Parsing Input: cdd

-------------------------------------------------------------------------------
|       Step       |       Stack       |       Input       |      Action      |
-------------------------------------------------------------------------------
|1                 |0                  |cdd$               |S1                |
-------------------------------------------------------------------------------
|2                 |0 c 1              |dd$                |S3                |
-------------------------------------------------------------------------------
|3                 |0 c 1 d 3          |d$                 |R3                |
-------------------------------------------------------------------------------
|4                 |0 c 1 C 5          |d$                 |R2                |
-------------------------------------------------------------------------------
|5                 |0 C 4              |d$                 |S3                |
-------------------------------------------------------------------------------
|6                 |0 C 4 d 3          |$                  |R3                |
-------------------------------------------------------------------------------
|7                 |0 C 4 C 6          |$                  |R1                |
-------------------------------------------------------------------------------
|8                 |0 S 2              |$                  |Accept            |
-------------------------------------------------------------------------------

Input string 'cdd' accepted!

================================================
Testing 2nd Input :

Parsing Input: ccc

-------------------------------------------------------------------------------
|       Step       |       Stack       |       Input       |      Action      |
-------------------------------------------------------------------------------
|1                 |0                  |ccc$               |S1                |
-------------------------------------------------------------------------------
|2                 |0 c 1              |cc$                |S1                |
-------------------------------------------------------------------------------
|3                 |0 c 1 c 1          |c$                 |S1                |
-------------------------------------------------------------------------------
|4                 |0 c 1 c 1 c 1      |$                  |                  |
-------------------------------------------------------------------------------
Error: No action defined for state 1 and symbol '$'
Input string 'ccc' is not valid according to the grammar
PS C:\Users\Samar Mittal\Desktop\Compiler LAb\lab10>
```