Rule Engine with Abstract Syntax Tree (AST)

Objective:

Develop a simple 3-tier rule engine application (Simple UI, API, and Backend, Data) to determine user eligibility based on attributes like age, department, income, spend etc. The system can use Abstract Syntax Tree (AST) to represent conditional rules and allow for dynamic creation, combination, and modification of these rules.

1. Data Structure:

Define the Node class to represent the AST structure. Each node can either be an operator (AND/OR) or an operand (conditions).

```
class Node:
    def __init__(self, node_type, left=None, right=None, value=None):
        self.type = node_type  # "operator" (AND/OR) or "operand" (conditions)
        self.left = left       # Reference to left child Node
        self.right = right     # Reference to right child Node
        self.value = value     # Optional value for operand (e.g., 30, "Sales")
```

2. API Design:

2.1 create_rule(rule_string) Function:

This function parses a rule string and builds the AST.

```
import re


def create_rule(rule_string):
    tokens = re.findall(r"\(|\)|\w+|'\w+'|[><=]+", rule_string)
    def parse_expression(tokens):
```

```
        stack = []
        for token in tokens:
            if token in ('AND', 'OR'):
                right = stack.pop()
                left = stack.pop()
                stack.append(Node(node_type='operator', left=left, right=right, value=token))
            elif re.match(r'[><=]+', token):
                op = token
                operand = stack.pop()
                value = stack.pop()
                condition = f"{value} {op} {operand}"
                stack.append(Node(node_type='operand', value=condition))
            else:
                stack.append(token)
        return stack[0]
    return parse_expression(tokens)
```

2.2 combine_rules(rules) Function:

This function combines multiple rules into a single AST using a common operator (AND/OR).

```
def combine_rules(rules, operator="AND"):
    if not rules:
        return None
    root = rules[0]
    for rule in rules[1:]:
        root = Node(node_type='operator', left=root, right=rule, value=operator)
    return root
```

2.3 evaluate_rule(AST, data) Function:

This function evaluates the rule represented by the AST against the provided user data.

```
def evaluate_rule(node, data):
    if node.type == 'operator':
        left_result = evaluate_rule(node.left, data)
        right_result = evaluate_rule(node.right, data)
        if node.value == 'AND':
            return left_result and right_result
        elif node.value == 'OR':
            return left_result or right_result
    elif node.type == 'operand':
        condition = node.value
        field, operator, threshold = re.split(r'([><=]+)', condition)
        field_value = data.get(field.strip())
        if operator == '>':
            return field_value > int(threshold)
        elif operator == '<':
            return field_value < int(threshold)
        elif operator == '=':
            return field_value == threshold.strip("'")
    return False
```

3. Test Cases:

Test Case 1: Create Rule and Verify AST Representation

rule1 = create_rule('((age > 30 AND department = Sales) OR (age < 25 AND department = Marketing)) AND (salary > 50000 OR experience > 5)')

Test Case 2: Combine Rules

rule2 = create_rule('((age > 30 AND department = Marketing)) AND (salary > 20000 OR experience > 5)')

combined_rule = combine_rules([rule1, rule2])

Test Case 3: Evaluate Rule

result = evaluate_rule(rule1, data={'age': 35, 'department': 'Sales', 'salary': 60000, 'experience': 3})