Application 1 : Rule Engine with 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.

Data Structure:

● Define a data structure to represent the AST.

● The data structure should allow rule changes

● E,g One data structure could be Node with following fields
○ type: String indicating the node type ("operator" for AND/OR, "operand" for conditions)
○ left: Reference to another Node (left child)
○ right: Reference to another Node (right child for operators)
○ value: Optional value for operand nodes (e.g., number for comparisons)

Data Storage

● Define the choice of database for storing the above rules and application metadata

● Define the schema with samples.

Sample Rules:

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

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

API Design: 1. create_rule(rule_string): This function takes a string representing a rule (as shown in the examples) and returns a Node object representing the corresponding AST.

2. combine_rules(rules): This function takes a list of rule strings and combines them into a single AST. It should consider efficiency and minimize redundant checks. You can explore different strategies (e.g., most frequent operator heuristic). The function should return the root node of the combined AST.

3. Evaluate_rule(JSON data): This function takes a JSON representing the combined rule's AST and a dictionary data containing attributes (e.g., data = {"age": 35, "department": "Sales", "salary": 60000, "experience": 3}). The function should evaluate the rule against the provided data and return True if the user is of that cohort based on the rule, False otherwise.

Test Cases: 1. Create individual rules from the examples using create_rule and verify their AST representation.

2. Combine the example rules using combine_rules and ensure the resulting AST reflects the combined logic.

3. Implement sample JSON data and test evaluate_rule for different scenarios.

4. Explore combining additional rules and test the functionality.

Bonus:

● Implement error handling for invalid rule strings or data formats (e.g., missing operators, invalid comparisons).

● Implement validations for attributes to be part of a catalog.

● Allow for modification of existing rules using additional functionalities within create_rule or separate functions.This could involve changing operators, operand values, or adding/removing sub-expressions within the AST.

● Consider extending the system to support user-defined functions within the rule language for advanced conditions (outside the scope of this exercise).

CODING

```
import re
import json
import sqlite3


class Node:
    def __init__(self, type, left=None, right=None, value=None):
        self.type = type  # 'operator' or 'operand'
        self.left = left
        self.right = right
        self.value = value  # For operands: ('age', '>', 30)

    def __repr__(self):
```

```
        if self.type == 'operator':
            return f"({self.left} {self.value} {self.right})"
        else:
            return f"{self.value[0]} {self.value[1]} {self.value[2]}"


def tokenize(rule_string):
    # Define the regex patterns for tokens
    token_specification = [
        ('NUMBER', r'\b\d+(\.\d*)?\b'),  # Integer or decimal number
        ('STRING', r"'[^']*'"),  # String enclosed in single quotes
        ('ID', r'\b\w+\b'),  # Identifiers
        ('OP', r'<=|>=|!=|=|<|>'),  # Operators
        ('AND', r'\bAND\b'),  # AND operator
        ('OR', r'\bOR\b'),  # OR operator
        ('LPAREN', r'\('),  # Left Parenthesis
        ('RPAREN', r'\)'),  # Right Parenthesis
        ('SKIP', r'[ \t]+'),  # Skip spaces and tabs
        ('MISMATCH', r'.'),  # Any other character
    ]
    token_regex = '|'.join('(?P<%s>%s)' % pair for pair in token_specification)
    get_token = re.compile(token_regex).match
    line = rule_string
    pos = 0
    tokens = []
    match = get_token(line)
    while match is not None:
        kind = match.lastgroup
        value = match.group(kind)
        if kind == 'NUMBER':
            tokens.append(('NUMBER', float(value) if '.' in value else int(value)))
        elif kind == 'STRING':
            tokens.append(('STRING', value.strip("'")))
        elif kind == 'ID':
            if value == 'AND' or value == 'OR':
                tokens.append((value, value))
            else:
                tokens.append(('ID', value))
        elif kind == 'OP':
            tokens.append(('OP', value))
        elif kind == 'LPAREN':
            tokens.append(('LPAREN', value))
        elif kind == 'RPAREN':
            tokens.append(('RPAREN', value))
```

```
            elif kind == 'SKIP':
                pass
            elif kind == 'MISMATCH':
                raise RuntimeError(f'Unexpected character {value!r} at position {pos}')
            pos = match.end()
            match = get_token(line, pos)
        return tokens


class Parser:
    def __init__(self, tokens):
        self.tokens = tokens
        self.pos = 0

    def parse(self):
        node = self.expression()
        if self.pos < len(self.tokens):
            raise RuntimeError('Unexpected token at the end')
        return node

    def match(self, expected_types):
        if self.pos < len(self.tokens):
            token_type, value = self.tokens[self.pos]
            if token_type in expected_types:
                self.pos += 1
                return token_type, value
        return None, None

    def expression(self):
        node = self.term()
        while True:
            token_type, value = self.match(['OR'])
            if token_type:
                right = self.term()
                node = Node('operator', left=node, right=right, value='OR')
            else:
                break
        return node

    def term(self):
        node = self.factor()
        while True:
            token_type, value = self.match(['AND'])
            if token_type:
```

```
            right = self.factor()
            node = Node('operator', left=node, right=right, value='AND')
        else:
            break
    return node

def factor(self):
    token_type, value = self.match(['LPAREN'])
    if token_type:
        node = self.expression()
        if not self.match(['RPAREN']):
            raise RuntimeError('Expected )')
        return node
    else:
        return self.comparison()

def comparison(self):
    token_type, left_value = self.match(['ID'])
    if not token_type:
        raise RuntimeError('Expected identifier')
    token_type, op_value = self.match(['OP'])
    if not token_type:
        raise RuntimeError('Expected operator')
    token_type, right_value = self.match(['NUMBER', 'STRING', 'ID'])
    if not token_type:
        raise RuntimeError('Expected number or string')
    return Node('operand', value=(left_value, op_value, right_value))


def create_rule(rule_string):
    tokens = tokenize(rule_string)
    parser = Parser(tokens)
    ast = parser.parse()
    return ast


def combine_rules(rule_strings):
    asts = [create_rule(rule_str) for rule_str in rule_strings]
    if not asts:
        return None
    combined_ast = asts[0]
    for ast in asts[1:]:
        combined_ast = Node('operator', left=combined_ast, right=ast, value='OR')
    return combined_ast
```

```
def evaluate_rule(ast, data):
    if ast.type == 'operator':
        left_val = evaluate_rule(ast.left, data)
        right_val = evaluate_rule(ast.right, data)
        if ast.value == 'AND':
            return left_val and right_val
        elif ast.value == 'OR':
            return left_val or right_val
    elif ast.type == 'operand':
        attr, op, value = ast.value
        if attr not in data:
            raise RuntimeError(f'Attribute {attr} not in data')
        data_value = data[attr]
        if op == '=':
            return data_value == value
        elif op == '!=':
            return data_value != value
        elif op == '>':
            return data_value > value
        elif op == '<':
            return data_value < value
        elif op == '>=':
            return data_value >= value
        elif op == '<=':
            return data_value <= value
        else:
            raise RuntimeError(f'Unknown operator {op}')
    else:
        raise RuntimeError(f'Unknown node type {ast.type}')


ATTRIBUTE_CATALOG = {'age', 'department', 'salary', 'experience'}


def modify_rule(ast, path, new_value):
    """
    Modify an existing rule AST.
    :param ast: The root node of the AST to modify
    :param path: A list of indices to navigate to the node to modify (e.g., [0, 1, 0])
    :param new_value: The new value to set for the node
    :return: The modified AST
    """
```

```
    current = ast
    for i in path[:-1]:
        if i == 0:
            current = current.left
        elif i == 1:
            current = current.right
        else:
            raise ValueError("Invalid path")

    if path[-1] == 0:
        current.left = new_value
    elif path[-1] == 1:
        current.right = new_value
    else:
        raise ValueError("Invalid path")

    return ast


def init_db():
    """Initialize the SQLite database and create the rules table."""
    conn = sqlite3.connect('rules.db')
    c = conn.cursor()
    c.execute('''CREATE TABLE IF NOT EXISTS rules
                 (id INTEGER PRIMARY KEY, name TEXT, rule_string TEXT)''')
    conn.commit()
    conn.close()


def save_rule(name, rule_string):
    """Save a rule to the database."""
    conn = sqlite3.connect('rules.db')
    c = conn.cursor()
    c.execute("INSERT INTO rules (name, rule_string) VALUES (?, ?)", (name, rule_string))
    conn.commit()
    conn.close()


def load_rule(name):
    """Load a rule from the database."""
    conn = sqlite3.connect('rules.db')
    c = conn.cursor()
    c.execute("SELECT rule_string FROM rules WHERE name = ?", (name,))
    result = c.fetchone()
```

```
    conn.close()
    if result:
        return result[0]
    return None


# Test Cases
if __name__ == '__main__':
    # Sample Rules
    rule1 = "((age > 30 AND department = 'Sales') OR (age < 25 AND department =
'Marketing')) AND (salary > 50000 OR experience > 5)"
    rule2 = "((age > 30 AND department = 'Marketing')) AND (salary > 20000 OR experience
> 5)"

    # Create individual rules
    ast1 = create_rule(rule1)
    print("AST for rule1:")
    print(ast1)

    ast2 = create_rule(rule2)
    print("\nAST for rule2:")
    print(ast2)

    # Combine rules
    combined_ast = combine_rules([rule1, rule2])
    print("\nCombined AST:")
    print(combined_ast)

    # Sample data
    data1 = {"age": 35, "department": "Sales", "salary": 60000, "experience": 3}
    data2 = {"age": 22, "department": "Marketing", "salary": 30000, "experience": 2}
    data3 = {"age": 40, "department": "Marketing", "salary": 25000, "experience": 6}
    data4 = {"age": 28, "department": "HR", "salary": 40000, "experience": 4}

    # Evaluate rules
    print("\nEvaluating data1:")
    result1 = evaluate_rule(combined_ast, data1)
    print(f"Result: {result1}")

    print("\nEvaluating data2:")
    result2 = evaluate_rule(combined_ast, data2)
    print(f"Result: {result2}")

    print("\nEvaluating data3:")
```

```
    result3 = evaluate_rule(combined_ast, data3)
    print(f"Result: {result3}")

    print("\nEvaluating data4:")
    result4 = evaluate_rule(combined_ast, data4)
    print(f"Result: {result4}")

    # Test error handling
    try:
        invalid_rule = "((invalid_attr > 30 AND department = 'Sales'))"
        create_rule(invalid_rule)
    except ValueError as e:
        print(f"\nCaught expected error: {e}")

    # Test rule modification
    modified_ast = modify_rule(ast1, [0, 0, 0], Node('operand', value=('age', '>', 35)))
    print("\nModified AST:")
    print(modified_ast)
```

OUTPUT

```
"C:\Users\HP\PycharmProjects\Rule Engine with AST\.venv\Scripts\python.exe"
"C:\Users\HP\PycharmProjects\Rule Engine with AST\main.py"
AST for rule1:
(((age > 30 AND department = Sales) OR (age < 25 AND department = Marketing)) AND
(salary > 50000 OR experience > 5))

AST for rule2:
((age > 30 AND department = Marketing) AND (salary > 20000 OR experience > 5))

Combined AST:
((((age > 30 AND department = Sales) OR (age < 25 AND department = Marketing)) AND
(salary > 50000 OR experience > 5)) OR ((age > 30 AND department = Marketing) AND
(salary > 20000 OR experience > 5)))

Evaluating data1:
Result: True

Evaluating data2:
Result: False

Evaluating data3:
Result: True
```

Evaluating data4:
Result: False

Modified AST:
(((age > 35 AND department = Sales) OR (age < 25 AND department = Marketing)) AND (salary > 50000 OR experience > 5))

Process finished with exit code 0