Assignment 1: Rule Engine with AST.

Database schema choice:

A document-oriented NoSQL database like MongoDB would be suitable to store the AST structures because it allows flexible schema design and can easily store hierarchical data like trees.

JSON:

```
{
  "rule_id": "string",  // Unique identifier for the rule
  "name": "string",     // Human-readable name for the rule
  "ast": {
    "type": "operator",
    "value": "AND",
    "left": {
      "type": "operator",
      "value": "OR",
      "left": {
        "type": "operand",
        "value": "age > 30 AND department = 'Sales'"
      },
      "right": {
        "type": "operand",
        "value": "age < 25 AND department = 'Marketing'"
      }
    },
    "right": {
      "type": "operator",
      "value": "OR",
```

```
    "left": {
      "type": "operand",
      "value": "salary > 50000"
    },
    "right": {
      "type": "operand",
      "value": "experience > 5"
    }
   }
 }
}
```

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)"

Code for Assignmet 1:Rule Engine with AST.

```
import re
import json
from pymongo import MongoClient
from bson.json_util import dumps, loads

# Data Structure for the AST Node
class Node:
```

```
    def __init__(self, type, left=None, right=None, value=None):
        self.type = type  # "operator" or "operand"
        self.left = left  # Reference to the left child Node
        self.right = right  # Reference to the right child Node
        self.value = value  # Value for operand nodes (e.g., number for comparisons)
        self.validate()

    def validate(self):
        if self.type not in ["operator", "operand"]:
            raise ValueError(f"Invalid node type: {self.type}")
        if self.type == "operator" and (self.left is None or self.right is None):
            raise ValueError("Operator nodes must have both left and right children.")
        if self.type == "operand" and self.value is None:
            raise ValueError("Operand nodes must have a value.")

# Catalog for valid attributes
VALID_ATTRIBUTES = ["age", "department", "salary", "experience"]

def validate_attribute(attribute):
    if attribute not in VALID_ATTRIBUTES:
        raise ValueError(f"Invalid attribute: {attribute}")

# Parse condition and create operand node
def parse_condition(condition):
    match = re.match(r"(\w+)\s*([<>=]+)\s*(\S+)", condition)
    if not match:
        raise ValueError(f"Invalid condition format: {condition}")
    attribute, operator, value = match.groups()
    validate_attribute(attribute)
```

```
    return Node(type="operand", value=(attribute, operator, value))

# Create rule by parsing rule string
def create_rule(rule_string):
    try:
        rule_string = rule_string.strip()
        # Basic AND/OR splitting for simplicity
        if " AND " in rule_string:
            left, right = map(create_rule, rule_string.split(" AND ", 1))
            return Node(type="operator", left=left, right=right, value="AND")
        elif " OR " in rule_string:
            left, right = map(create_rule, rule_string.split(" OR ", 1))
            return Node(type="operator", left=left, right=right, value="OR")
        else:
            return parse_condition(rule_string)
    except Exception as e:
        print(f"Error creating rule: {e}")
        return None

# Combine multiple rules into one AST
def combine_rules(rules):
    try:
        combined_ast = None
        for rule_string in rules:
            rule_ast = create_rule(rule_string)
            if combined_ast is None:
                combined_ast = rule_ast
            else:
                # Example: Combine using AND by default
```

```
            combined_ast = Node(type="operator", left=combined_ast, right=rule_ast, value="AND")
        return combined_ast
    except Exception as e:
        print(f"Error combining rules: {e}")
        return None

# Evaluate the AST against provided user data
def evaluate_rule(ast, data):
    if ast.type == "operand":
        attribute, operator, value = ast.value
        if attribute not in data:
            raise ValueError(f"Missing attribute in data: {attribute}")
        user_value = data[attribute]
        if operator == ">":
            return user_value > float(value)
        elif operator == "<":
            return user_value < float(value)
        elif operator == "=":
            return str(user_value) == str(value)
        else:
            raise ValueError(f"Invalid operator: {operator}")
    elif ast.type == "operator":
        left_result = evaluate_rule(ast.left, data)
        right_result = evaluate_rule(ast.right, data)
        if ast.value == "AND":
            return left_result and right_result
        elif ast.value == "OR":
            return left_result or right_result
        else:
```

```
            raise ValueError(f"Invalid operator: {ast.value}")
    else:
        raise ValueError("Invalid AST Node")

# Functions to modify existing rules
def modify_node(node, new_type=None, new_value=None):
    if new_type:
        node.type = new_type
    if new_value:
        node.value = new_value
    node.validate()

def add_sub_expression(parent_node, sub_node, position="left"):
    if position == "left":
        parent_node.left = sub_node
    elif position == "right":
        parent_node.right = sub_node
    parent_node.validate()

# Support for user-defined functions
USER_DEFINED_FUNCTIONS = {}

def register_function(name, func):
    USER_DEFINED_FUNCTIONS[name] = func

def evaluate_function(name, *args):
    if name not in USER_DEFINED_FUNCTIONS:
        raise ValueError(f"Function {name} is not defined.")
    return USER_DEFINED_FUNCTIONS[name](*args)
```

```
def parse_condition_with_function(condition):
    match = re.match(r"(\w+)\(([^)]*)\)", condition)
    if match:
        func_name, args = match.groups()
        args = args.split(",")
        return Node(type="operand", value=(func_name, args))
    else:
        return parse_condition(condition)

# MongoDB Integration
MONGO_URI = "mongodb://localhost:27017/"
DB_NAME = "rule_engine_db"
COLLECTION_NAME = "rules"

client = MongoClient(MONGO_URI)
db = client[DB_NAME]
rules_collection = db[COLLECTION_NAME]

def store_rule(rule_name, rule_ast):
    rule_document = {
        "rule_name": rule_name,
        "rule_ast": dumps(rule_ast, default=lambda o: o.__dict__)
    }
    rules_collection.insert_one(rule_document)

def get_rule(rule_name):
    rule_document = rules_collection.find_one({"rule_name": rule_name})
    if rule_document:
```

```
    return loads(rule_document["rule_ast"])
  else:
    return None

def update_rule(rule_name, new_rule_ast):
  rules_collection.update_one(
    {"rule_name": rule_name},
    {"$set": {"rule_ast": dumps(new_rule_ast, default=lambda o: o.__dict__)}}
  )

def delete_rule(rule_name):
  rules_collection.delete_one({"rule_name": rule_name})

# Example usage
if __name__ == "__main__":
  # Create and store a rule
  rule_string = "age > 30 AND department = 'Sales'"
  rule = create_rule(rule_string)
  store_rule("example_rule", rule)

  # Retrieve and evaluate the rule
  stored_rule_ast = get_rule("example_rule")
  if stored_rule_ast:
    # Convert stored AST back to Node objects
    def dict_to_node(d):
      if d['type'] == 'operator':
        return Node(
          type=d['type'],
          left=dict_to_node(d['left']),
```

```
            right=dict_to_node(d['right']),
            value=d['value']
        )
    else:
        return Node(
            type=d['type'],
            value=d['value']
        )

rule = dict_to_node(stored_rule_ast)
user_data = {"age": 35, "department": "Sales", "salary": 60000}
result = evaluate_rule(rule, user_data)
print(f"Evaluation Result: {result}")

# Modify and update a rule
modify_node(rule, new_value=("age", "<", 40))
update_rule("example_rule", rule)

# Add a sub-expression
new_condition = create_rule("salary > 50000")
add_sub_expression(rule, new_condition, position="right")
update_rule("example_rule", rule)

# Register and use a user-defined function
def is_experienced(years):
    return years > 5

register_function("is_experienced", is_experienced)
```

Explanation

1. **Node Class**: Represents an AST node with validation for node type, operator, and operand values.
2. **Attribute Validation**: Ensures only valid attributes (e.g., age, department, etc.) are used.
3. **create_rule**: Parses a rule string and constructs an AST, handling errors like missing operators or invalid conditions.
4. **combine_rules**: Combines multiple ASTs into one, typically using "AND" logic, but this can be customized.
5. **evaluate_rule**: Evaluates the AST against a provided user data dictionary, supporting logical operations and comparisons.
6. **Modifying Rules**: Functions to modify existing rules (e.g., change operators or values) and to add sub-expressions.
7. **User-Defined Functions**: Register custom functions that can be used within rules, extending the flexibility of the rule engine.

This implementation is modular, allowing for the creation, combination, modification, and evaluation of rules with robust error handling and flexibility.

1. **MongoDB Integration**:
    - **store_rule(rule_name, rule_ast)**: Stores a rule with its AST in MongoDB.
    - **get_rule(rule_name)**: Retrieves a rule's AST from MongoDB.
    - **update_rule(rule_name, new_rule_ast)**: Updates an existing rule's AST in MongoDB.
    - **delete_rule(rule_name)**: Deletes a rule from MongoDB.
2. **Rule Handling**:
    - The code includes functions to create, combine, evaluate, modify, and store rules. It also handles conversion between AST node objects and JSON for MongoDB storage.
3. **Example Usage**:
    - Shows how to create, store, retrieve, modify, and evaluate rules using the integrated MongoDB database.

This complete code integrates MongoDB with the rule engine application, including MongoDB operations and example usage. Ensure MongoDB is running and accessible at mongodb://localhost:27017/ before running this code.