import json

Abstract Syntax Tree (AST)

import re

class Node:

def \_\_init\_\_(self, node\_type, left=None, right=None, value=None):

self.node\_type = node\_type # 'operator' or 'operand'

self.left = left # Reference to left Node

self.right = right # Reference to right Node

self.value = value # Optional value for operand nodes

def parse\_expression(expression):

tokens = re.findall(r'\w+|[<>!=]=|AND|OR|\(|\)', expression)

def parse(tokens):

stack = []

current = []

for token in tokens:

if token == '(':

stack.append(current)

current = []

elif token == ')':

node = build\_ast(current)

current = stack.pop()

current.append(node)

else:

current.append(token)

return build\_ast(current)

def build\_ast(parsed\_tokens):

if not parsed\_tokens:

return None

if len(parsed\_tokens) == 1:

token = parsed\_tokens[0]

if re.match(r'\w+', token):

return Node('operand', value=token)

else:

raise ValueError("Invalid token")

for op in ('OR', 'AND'):

if op in parsed\_tokens:

index = parsed\_tokens.index(op)

left = build\_ast(parsed\_tokens[:index])

right = build\_ast(parsed\_tokens[index + 1:])

return Node('operator', left=left, right=right, value=op)

# Handle comparisons

for comparison in ['>', '<', '=', '>=', '<=', '!=']:

if comparison in parsed\_tokens:

index = parsed\_tokens.index(comparison)

left = parsed\_tokens[index - 1]

right = parsed\_tokens[index + 1]

return Node('operator',

left=Node('operand', value=left),

right=Node('operand', value=right),

value=comparison)

raise ValueError("Invalid expression")

return parse(tokens)

def create\_rule(rule\_string):

return parse\_expression(rule\_string)

def combine\_ast\_nodes(ast1, ast2):

# Assuming ast1 and ast2 are both root nodes of their respective ASTs

return Node('operator', left=ast1, right=ast2, value='OR') # Combine with OR for simplicity

def combine\_rules(rules):

combined\_ast = None

for rule in rules:

ast = create\_rule(rule)

if combined\_ast is None:

combined\_ast = ast

else:

combined\_ast = combine\_ast\_nodes(combined\_ast, ast)

return combined\_ast

def evaluate\_node(node, data):

if node.node\_type == 'operand':

if isinstance(node.value, str):

return data.get(node.value)

else:

return node.value

elif node.node\_type == 'operator':

left\_result = evaluate\_node(node.left, data)

right\_result = evaluate\_node(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.value in ['>', '<', '=', '>=', '<=', '!=']:

left\_value = evaluate\_node(node.left, data)

right\_value = evaluate\_node(node.right, data)

if node.value == '>':

return left\_value > right\_value

elif node.value == '<':

return left\_value < right\_value

elif node.value == '=':

return left\_value == right\_value

elif node.value == '>=':

return left\_value >= right\_value

elif node.value == '<=':

return left\_value <= right\_value

elif node.value == '!=':

return left\_value != right\_value

def evaluate\_rule(ast, data):

return evaluate\_node(ast, data)

# Test Cases

def test\_create\_rule():

rule\_string = "((age > 30 AND department = 'Sales') OR (age < 25 AND department = 'Marketing'))"

ast = create\_rule(rule\_string)

assert ast is not None # Further checks on the AST structure

def test\_combine\_rules():

rules = [

"((age > 30 AND department = 'Sales'))",

"((age < 25 AND department = 'Marketing'))"

]

combined\_ast = combine\_rules(rules)

assert combined\_ast is not None # Further checks on the combined AST

def test\_evaluate\_rule():

data = {"age": 35, "department": "Sales", "salary": 60000, "experience": 3}

ast = create\_rule("((age > 30 AND department = 'Sales'))")

result = evaluate\_rule(ast, data)

assert result is True # Check the expected output

# Run tests

test\_create\_rule()

test\_combine\_rules()

test\_evaluate\_rule()

print("All tests passed!")

**Overview of the Code**

* **Node Class**: Represents nodes of the AST.
* **parse\_expression**: Parses a rule string and constructs the AST.
* **create\_rule**: Creates an AST from a rule string.
* **combine\_ast\_nodes**: Combines two ASTs.
* **combine\_rules**: Combines multiple rules into a single AST.
* **evaluate\_node**: Evaluates an AST node based on provided data.
* **evaluate\_rule**: Evaluates the entire AST against the given data.
* **Test Cases**: Verifies the functionality of rule creation, combination, and evaluation