```
import re

class Node:
    def __init__(self, node_type, left=None, right=None, value=None):
        """
        Represents a node in the AST. Can be an operator (AND/OR) or an operand (condition).
        :param node_type: Type of node ('operator' or 'operand')
        :param left: Left child (for operators)
        :param right: Right child (for operators)
        :param value: Condition (for operand nodes) or the operator type (for operator nodes)
        """
        self.type = node_type  # 'operator' (AND/OR) or 'operand' (condition)
        self.left = left
        self.right = right
        self.value = value

    def evaluate(self, data):
        """
        Recursively evaluate the AST against input data.
        :param data: Dictionary containing values like age, department, etc.
        :return: Boolean result of rule evaluation
        """
        if self.type == 'operand':
            # Use the format() method to replace placeholders with actual values from data
            return eval(self.value.format(**data))
        elif self.type == 'operator':
            left_val = self.left.evaluate(data)
            right_val = self.right.evaluate(data)
            if self.value == 'AND':
                return left_val and right_val
            elif self.value == 'OR':
```

```
            return left_val or right_val
        return False


def create_rule(rule_string):
    """
    Parses a rule string into an Abstract Syntax Tree (AST).
    :param rule_string: A string representing the rule, e.g., "(age > 30 AND department == 'Sales')"
    :return: Root node of the AST
    """
    tokens = re.split(r'(\(|\)|AND|OR)', rule_string)
    stack = []

    for token in tokens:
        token = token.strip()
        if not token:
            continue
        if token == '(':
            stack.append(token)
        elif token in ('AND', 'OR'):
            stack.append(token)
        elif token == ')':
            right = stack.pop()
            operator = stack.pop()
            left = stack.pop()
            stack.pop()  # Remove '('
            node = Node('operator', left=left, right=right, value=operator)
            stack.append(node)
        else:
            # Operand node, store the condition (e.g., "age > 30")
            node = Node('operand', value=token)
            stack.append(node)
```

```
    return stack.pop()  # Root node of the AST

def combine_rules(rules, operator='AND'):
    """
    Combines multiple rules (ASTs) using the specified operator (AND/OR).
    :param rules: List of rule ASTs
    :param operator: Operator to combine the rules ('AND' or 'OR')
    :return: Combined AST node
    """
    if len(rules) == 1:
        return rules[0]

    combined_rule = rules[0]
    for rule in rules[1:]:
        combined_rule = Node('operator', left=combined_rule, right=rule, value=operator)

    return combined_rule

def evaluate_rule(ast, data):
    """
    Evaluates a rule AST against the provided data.
    :param ast: Root node of the AST
    :param data: Dictionary of user attributes (e.g., age, department)
    :return: Boolean result of rule evaluation
    """
    return ast.evaluate(data)

# Example usage:

# Define rule strings
```

```
rule1_str = "(age > 30 AND department == 'Sales')"
rule2_str = "(salary > 50000 OR experience > 5)"

# Create ASTs for the rules
rule1 = create_rule(rule1_str)
rule2 = create_rule(rule2_str)

# Combine the two rules using 'AND'
combined_rule = combine_rules([rule1, rule2], operator='AND')

# Example user data
user_data = {
    "age": 35,
    "department": "Sales",
    "salary": 60000,
    "experience": 3
}

# Evaluate the combined rule
result = evaluate_rule(combined_rule, user_data)
print(f"Rule evaluation result: {result}")  # Expected output: True

# Another example user data
user_data2 = {
    "age": 28,
    "department": "Marketing",
    "salary": 40000,
    "experience": 2
}

# Evaluate with different user data
```

```
result2 = evaluate_rule(combined_rule, user_data2)
print(f"Rule evaluation result: {result2}")  # Expected output: False
```