

谓词逻辑编程实验

Howling Hounds Example

1. Howling Hounds 符号抽象

1. 所有猎犬在夜晚都会嚎叫：

$$\circ (\forall x (\text{hound}(x) \rightarrow \text{howls}(x, \text{night})))$$

2. 任何人如果有猫就不会有老鼠：

$$\circ (\forall x (\text{has}(x, \text{cat}) \rightarrow \neg \text{has}(x, \text{mouse})))$$

3. 轻睡者没有会在夜晚嚎叫的东西：

$$\circ (\forall x (\text{light_sleeper}(x) \rightarrow \forall y (\text{howls}(y, \text{night}) \rightarrow \neg \text{has}(x, y)))$$

4. 约翰有猫或者猎犬：

$$\circ (\text{has}(\text{John}, \text{cat}) \vee \text{has}(\text{John}, \text{hound}))$$

需要证明：

- 如果约翰是轻睡者，那么他没有老鼠：
- $(\text{light_sleeper}(\text{John}) \rightarrow \neg \text{has}(\text{John}, \text{mouse}))$

2. Howling Hounds 逻辑实现

抽取谓词

- $\text{Hound}(x)$ ：x 是一只猎犬。
- $\text{HowlsAtNight}(x)$ ：x 在夜晚嚎叫。
- $\text{Has}(x, y)$ ：x 拥有 y。
- $\text{Cat}(y)$ ：y 是一只猫。
- $\text{Mouse}(y)$ ：y 是一只老鼠。
- $\text{LightSleeper}(x)$ ：x 是一个轻睡者。

范式转换

消除蕴含和双重否定

1. $[\neg \forall x (\neg \text{Hound}(x) \vee \text{HowlsAtNight}(x))]$
2. $[\neg \forall x (\neg (\exists y (\text{Cat}(y) \wedge \text{Has}(x, y))) \vee \neg \exists z (\text{Mouse}(z) \wedge \text{Has}(x, z)))]$
3. $[\neg \forall x (\neg \text{LightSleeper}(x) \vee \forall y (\neg \text{HowlsAtNight}(y) \vee \neg \text{Has}(x, y)))]$
4. 前提4不含蕴含，不需转换。

5. 结论:

$$[\neg \text{LightSleeper}(\text{John}) \vee \forall z (\neg \text{Mouse}(z) \vee \neg \text{Has}(\text{John}, z))]$$

####将公式转换为前束范式 (Prenex Normal Form, PNF)

德摩根律

$$[\forall x (\neg \exists y (\text{Cat}(y) \wedge \text{Has}(x, y)) \vee \neg \exists z (\text{Mouse}(z) \wedge \text{Has}(x, z)))]$$

$$[\forall x (\forall y (\neg (\text{Cat}(y) \wedge \text{Has}(x, y))) \vee \forall z (\neg (\text{Mouse}(z) \wedge \text{Has}(x, z))))]$$

$$[\forall x (\neg \text{LightSleeper}(x) \vee \forall y (\neg \text{HowlsAtNight}(y) \vee \neg \text{Has}(x, y)))]$$

消除量词嵌套，标准化变量

为了避免变量冲突，我们需要标准化变量名。

标准化前提

$$1. [\forall x (\neg \text{Hound}(x) \vee \text{HowlsAtNight}(x))]$$

$$2. [\forall x (\forall y (\neg \text{Cat}(y) \vee \neg \text{Has}(x, y)) \vee \forall z (\neg \text{Mouse}(z) \vee \neg \text{Has}(x, z)))]$$

$$3. [\forall x (\neg \text{LightSleeper}(x) \vee \forall y (\neg \text{HowlsAtNight}(y) \vee \neg \text{Has}(x, y)))]$$

$$4. [\exists y ((\text{Cat}(y) \wedge \text{Has}(\text{John}, y)) \vee (\text{Hound}(y) \wedge \text{Has}(\text{John}, y)))]$$

5. 结论:

$$[\neg \text{LightSleeper}(\text{John}) \vee \forall z (\neg \text{Mouse}(z) \vee \neg \text{Has}(\text{John}, z))]$$

将公式转换为合取范式 (CNF)

$$[\forall x (\forall y (\neg \text{Cat}(y) \vee \neg \text{Has}(x, y)) \vee \forall z (\neg \text{Mouse}(z) \vee \neg \text{Has}(x, z)))]$$

$$[\forall x \forall y \forall z ((\neg \text{Cat}(y) \vee \neg \text{Has}(x, y)) \vee (\neg \text{Mouse}(z) \vee \neg \text{Has}(x, z)))]$$

最终的CNF表示

为了简化，我们可以将所有前提和结论转换为子句的集合。

前提子句

$$1. [\neg \text{Hound}(x) \vee \text{HowlsAtNight}(x)]$$

$$2. [\neg \text{Cat}(y) \vee \neg \text{Has}(x, y) \vee \neg \text{Mouse}(z) \vee \neg \text{Has}(x, z)]$$

由于存在所有组合，我们可以将其拆分为两个子句：

$$\circ [\neg \text{Cat}(y) \vee \neg \text{Has}(x, y)]$$

$$\circ [\neg \text{Mouse}(z) \vee \neg \text{Has}(x, z)]$$

3. $[\neg \text{LightSleeper}(x) \vee \neg \text{HowlsAtNight}(y) \vee \neg \text{Has}(x, y)]$

4. $[(\text{Cat}(y) \wedge \text{Has}(\text{John}, y)) \vee (\text{Hound}(y) \wedge \text{Has}(\text{John}, y))]$

这可以拆分为两个子句：

- $[\text{Cat}(y) \vee \text{Hound}(y)]$
- $[\text{Has}(\text{John}, y)]$

3. Howling Hounds 代码实现

定义符号

```
from collections import defaultdict
import copy

class Term:
    def __init__(self, name):
        self.name = name

    def is_variable(self):
        # 变量以小写字母开头
        return self.name[0].islower()

    def __repr__(self):
        return self.name

    def __eq__(self, other):
        return isinstance(other, Term) and self.name == other.name

    def __hash__(self):
        return hash(self.name)

class Predicate:
    def __init__(self, name, args, is_negative=False):
        self.name = name
        self.args = args # list of Term
        self.is_negative = is_negative

    def negate(self):
        return Predicate(self.name, self.args, not self.is_negative)

    def __repr__(self):
        sign = "¬" if self.is_negative else ""
        args_str = ", ".join([str(arg) for arg in self.args])
        return f"{sign}{self.name}({args_str})"

    def __eq__(self, other):
        return (self.name == other.name and
                self.args == other.args and
                self.is_negative == other.is_negative)
```

```

def __hash__(self):
    return hash((self.name, tuple(self.args), self.is_negative))

class Clause:
    def __init__(self, predicates):
        self.predicates = predicates # list of Predicate

    def __repr__(self):
        return " ∨ ".join([str(p) for p in self.predicates])

    def is_empty(self):
        return len(self.predicates) == 0

```

定义规则

```

# 定义常量和变量
John = Term("John")
a = Term("a") # 用于存在变量

# 定义变量
x = Term("x")
y = Term("y")
z = Term("z")

# 定义谓词
def Hound(term):
    return Predicate("Hound", [term])

def HowlsAtNight(term):
    return Predicate("HowlsAtNight", [term])

def Has(term1, term2):
    return Predicate("Has", [term1, term2])

def Cat(term):
    return Predicate("Cat", [term])

def Mouse(term):
    return Predicate("Mouse", [term])

def LightSleeper(term):
    return Predicate("LightSleeper", [term])

# 定义子句 (规则)

# 1.  $\neg \text{Hound}(x) \vee \text{HowlsAtNight}(x)$ 
clause1 = Clause([Hound(x).negate(), HowlsAtNight(x)])

# 2a.  $\neg \text{Cat}(y) \vee \neg \text{Has}(x, y)$ 
clause2a = Clause([Cat(y).negate(), Has(x, y).negate()])

```

```

# 2b.  $\neg \text{Mouse}(z) \vee \neg \text{Has}(x, z)$ 
clause2b = Clause([Mouse(z).negate(), Has(x, z).negate()])

# 3.  $\neg \text{LightSleeper}(x) \vee \neg \text{HowlsAtNight}(y) \vee \neg \text{Has}(x, y)$ 
clause3 = Clause([LightSleeper(x).negate(), HowlsAtNight(y).negate(), Has(x,
y).negate()])

# 4a.  $\text{Cat}(y) \vee \text{Hound}(y)$ 
clause4a = Clause([Cat(y), Hound(y)])

# 4b.  $\text{Has}(\text{John}, y)$ 
clause4b = Clause([Has(John, y)])

# 结论的否定
# 5a.  $\text{LightSleeper}(\text{John})$ 
clause5a = Clause([LightSleeper(John)])

# 5b.  $\text{Mouse}(a)$ 
clause5b = Clause([Mouse(a)])

# 5c.  $\text{Has}(\text{John}, a)$ 
clause5c = Clause([Has(John, a)])

# 收集所有子句
clauses = [clause1, clause2a, clause2b, clause3, clause4a, clause4b, clause5a,
clause5b, clause5c]

```

推理

```

def unify(x, y, subst):
    """
    **description**
    合一算法，尝试合一两个项或谓词，并返回替换列表。

    **params**
    x: 第一个项或谓词。
    y: 第二个项或谓词。
    subst: 当前的替换列表。

    **returns**
    更新的替换列表，如果无法合一则返回 None。
    """
    if subst is None:
        return None;
    elif x == y:
        return subst;
    elif isinstance(x, Term) and x.is_variable():
        return unify_var(x, y, subst);
    elif isinstance(y, Term) and y.is_variable():
        return unify_var(y, x, subst);

```

```
elif isinstance(x, Predicate) and isinstance(y, Predicate):
    if x.name != y.name or len(x.args) != len(y.args):
        return None;
    for a, b in zip(x.args, y.args):
        subst = unify(a, b, subst);
        if subst is None:
            return None;
    return subst;
else:
    return None;

def unify_var(var, x, subst):
    '''
    **description**
    合一变量。

    **params**
    var: 变量项。
    x: 另一个项。
    subst: 当前的替换列表。

    **returns**
    更新的替换列表, 如果无法合一则返回 None。
    '''
    if var in subst:
        return unify(subst[var], x, subst);
    elif x in subst:
        return unify(var, subst[x], subst);
    else:
        if occurs_check(var, x, subst):
            return None;
        subst[var] = x;
        return subst;

def occurs_check(var, x, subst):
    '''
    **description**
    检查变量是否出现在替换中, 防止无限递归。

    **params**
    var: 变量项。
    x: 另一个项。
    subst: 当前的替换列表。

    **returns**
    如果出现返回 True, 否则返回 False。
    '''
    if var == x:
        return True;
    elif isinstance(x, Term) and x.is_variable() and x in subst:
        return occurs_check(var, subst[x], subst);
    return False;

def substitute(predicate, subst):
```

```
'''
**description**
在谓词中应用替换。

**params**
predicate: 需要替换的谓词。
subst: 替换列表。

**returns**
应用替换后的新谓词。
'''

new_args = [];
for arg in predicate.args:
    if arg in subst:
        new_arg = subst[arg];
        while new_arg in subst:
            new_arg = subst[new_arg];
        new_args.append(new_arg);
    else:
        new_args.append(arg);
return Predicate(predicate.name, new_args, predicate.is_negative);

def substitute_clause(clause, subst):
    '''
    **description**
    在子句中应用替换。

    **params**
    clause: 需要替换的子句。
    subst: 替换列表。

    **returns**
    应用替换后的新子句。
    '''

    new_predicates = [];
    for pred in clause.predicates:
        new_pred = substitute(pred, subst);
        new_predicates.append(new_pred);
    return Clause(new_predicates);

def resolution(clauses):
    '''
    **description**
    主归结算法，尝试在子句集合中证明矛盾（导出空子句）。

    **params**
    clauses: 初始子句集合。

    **returns**
    布尔值，表明是否成功证明矛盾。
    '''

    new = set();
    steps = 0; # 初始化推理步数计数器
    while True:
```

```

n = len(clauses);
pairs = [(clauses[i], clauses[j]) for i in range(n) for j in range(i + 1,
n)];
for (ci, cj) in pairs:
    # 输出当前正在处理的子句对
    print(f"\n正在尝试归结子句对: {ci} 和 {cj}");
    resolvents = resolve(ci, cj);
    steps += 1; # 每次归结操作计为一步
    for resolvent in resolvents:
        # 明确标记涉及的子句、操作及结果
        print(f"步骤 {steps}: 将子句 {ci} 和 {cj} 归结得出新子句:
{resolvent}");
        if resolvent.is_empty():
            print("导出空子句, 证明成功! ");
            print(f"推理结束, 总步数: {steps}");
            return True;
        new.add(resolvent);
    new_clauses = new - set(clauses);
    if not new_clauses:
        print("无法继续归结, 证明失败。");
        print(f"推理结束, 总步数: {steps}");
        return False;
    clauses.extend(new_clauses);

def resolve(clause1, clause2):
    """
    **description**
    尝试对两个子句进行归结, 返回新的子句列表。

    **params**
    clause1: 第一个子句。
    clause2: 第二个子句。

    **returns**
    归结后的新子句列表。
    """
    resolvents = [];
    for pred1 in clause1.predicates:
        for pred2 in clause2.predicates:
            if pred1.name == pred2.name and pred1.is_negative !=
pred2.is_negative:
                subst = {};
                pred1_subst = copy.deepcopy(pred1);
                pred2_subst = copy.deepcopy(pred2);
                # 明确指出使用unify处理的两个谓词
                print(f"尝试合一谓词 {pred1_subst} 和 {pred2_subst} 使用替换列表
{subst}");
                subst = unify(pred1_subst, pred2_subst, subst);
                if subst is not None:
                    # 输出替换的具体操作
                    print(f"替换列表更新为 {subst}, 应用到子句 {clause1} 和
{clause2}");
                    new_predicates = [substitute(p, subst) for p in
clause1.predicates + clause2.predicates

```



```

        if p != pred1 and p != pred2];
        new_clause = Clause(new_predicates);
        resolvents.append(new_clause);

    return resolvents;

# 开始推理
print("初始子句集合: ");
for clause in clauses:
    print(clause);

print("\n开始归结推理...");

result = resolution(clauses);

```

Drug Dealer and Customs Official Example

1. Drug dealer and customs official 符号抽象

1. 海关官员搜查所有进入该国但不是VIP的人：

- $(\forall x (\text{entered}(x) \wedge \neg \text{VIP}(x) \rightarrow \text{searched_by_customs}(x)))$

2. 一些毒贩进入了国家，他们只被毒贩搜查：

- $(\exists x (\text{drug_dealer}(x) \wedge \text{entered}(x) \wedge \text{searched_by_drug_dealer}(x)))$

3. 没有毒贩是VIP：

- $(\forall x (\text{drug_dealer}(x) \rightarrow \neg \text{VIP}(x)))$

需要证明：

- 一些海关官员是毒贩：
- $(\exists x (\text{customs_official}(x) \wedge \text{drug_dealer}(x)))$

2. Drug Dealer and Customs Official 逻辑实现

抽取谓词

- $\text{entered}(x)$ ：x 进入该国。
- $\text{VIP}(x)$ ：x 是VIP。
- $\text{searched_by_customs}(x)$ ：x 被海关官员搜查。
- $\text{drug_dealer}(x)$ ：x 是毒贩。
- $\text{searched_by_drug_dealer}(x)$ ：x 被毒贩搜查。
- $\text{customs_official}(x)$ ：x 是海关官员。

范式转换

消除蕴含和双重否定

1. $[\forall x (\text{entered}(x) \wedge \neg \text{VIP}(x) \rightarrow \text{sought_by_customs}(x))]$ 转换为: $[\forall x (\neg \text{entered}(x) \vee \text{VIP}(x) \vee \text{sought_by_customs}(x))]$
2. $[\exists x (\text{drug_dealer}(x) \wedge \text{entered}(x) \wedge \text{sought_by_drug_dealer}(x))]$
3. $[\forall x (\text{drug_dealer}(x) \rightarrow \neg \text{VIP}(x))]$ 转换为: $[\forall x (\neg \text{drug_dealer}(x) \vee \neg \text{VIP}(x))]$
4. $[\exists x (\text{customs_official}(x) \wedge \text{drug_dealer}(x))]$

将公式转换为前束范式 (Prenex Normal Form, PNF)

1. $[\forall x (\neg \text{entered}(x) \vee \text{VIP}(x) \vee \text{sought_by_customs}(x))]$
2. $[\exists x (\text{drug_dealer}(x) \wedge \text{entered}(x) \wedge \text{sought_by_drug_dealer}(x))]$
3. $[\forall x (\neg \text{drug_dealer}(x) \vee \neg \text{VIP}(x))]$
4. $[\exists x (\text{customs_official}(x) \wedge \text{drug_dealer}(x))]$

将公式转换为合取范式 (CNF)

1. $[\neg \text{entered}(x) \vee \text{VIP}(x) \vee \text{sought_by_customs}(x)]$
2. $[\text{drug_dealer}(x) \wedge \text{entered}(x) \wedge \text{sought_by_drug_dealer}(x)]$
3. $[\neg \text{drug_dealer}(x) \vee \neg \text{VIP}(x)]$
4. $[\text{customs_official}(x) \wedge \text{drug_dealer}(x)]$

3. Drug Dealer and Customs Official 代码实现

定义符号

```
from collections import defaultdict
import copy

class Term:
    def __init__(self, name):
        """
        **description**
        初始化一个项（变量或常量）。

        **params**
        name: 项的名称。

        **returns**
        None
        """
        self.name = name;

    def isVariable(self):
```

```
'''
    **description**
    判断项是否为变量（名称以小写字母开头）。

    **params**
    None

    **returns**
    如果是变量返回 True, 否则返回 False。
'''
    return self.name[0].islower();

def __repr__(self):
    return self.name;

def __eq__(self, other):
    return isinstance(other, Term) and self.name == other.name;

def __hash__(self):
    return hash(self.name);

class Predicate:
    def __init__(self, name, args, isNegative=False):
        '''
            **description**
            初始化一个谓词。

            **params**
            name: 谓词名称。
            args: 参数列表 (Term 对象列表)。
            isNegative: 是否为否定谓词。

            **returns**
            None
        '''
        self.name = name;
        self.args = args; # list of Term
        self.isNegative = isNegative;

    def negate(self):
        '''
            **description**
            返回该谓词的否定形式。

            **params**
            None

            **returns**
            否定的 Predicate 对象。
        '''
        return Predicate(self.name, self.args, not self.isNegative);

    def __repr__(self):
        sign = "¬" if self.isNegative else "";
```

```

        argsStr = ", ".join([str(arg) for arg in self.args]);
        return f"{sign}{self.name}({argsStr})";

    def __eq__(self, other):
        return (self.name == other.name and
                self.args == other.args and
                self.isNegative == other.isNegative);

    def __hash__(self):
        return hash((self.name, tuple(self.args), self.isNegative));

class Clause:
    def __init__(self, predicates):
        """
        **description**
        初始化一个子句（由谓词组成的析取）。

        **params**
        predicates: 谓词列表。

        **returns**
        None
        """
        self.predicates = predicates; # list of Predicate

    def __repr__(self):
        return " ∨ ".join([str(p) for p in self.predicates]);

    def isEmpty(self):
        """
        **description**
        判断子句是否为空（没有任何谓词）。

        **params**
        None

        **returns**
        如果为空返回 True, 否则返回 False。
        """
        return len(self.predicates) == 0;

```

定义规则

```

# 定义常量和变量
x = Term("x");
a = Term("a"); # 用于存在量词的实例
b = Term("b");

# 定义谓词
def entered(term):
    return Predicate("entered", [term]);

```

```

def VIP(term):
    return Predicate("VIP", [term]);

def searchedByCustoms(term):
    return Predicate("searchedByCustoms", [term]);

def drugDealer(term):
    return Predicate("drugDealer", [term]);

def searchedByDrugDealer(term):
    return Predicate("searchedByDrugDealer", [term]);

def customsOfficial(term):
    return Predicate("customsOfficial", [term]);

# 定义子句 (规则)

# 1.  $\neg \text{entered}(x) \vee \text{VIP}(x) \vee \text{searchedByCustoms}(x)$ 
clause1 = Clause([entered(x).negate(), VIP(x), searchedByCustoms(x)]);

# 2.  $\text{drugDealer}(a) \wedge \text{entered}(a) \wedge \text{searchedByDrugDealer}(a)$ 
clause2a = Clause([drugDealer(a)]);
clause2b = Clause([entered(a)]);
clause2c = Clause([searchedByDrugDealer(a)]);

# 2. 他们只被毒贩搜查  $\Rightarrow \neg \text{searchedByCustoms}(a)$ 
clause2d = Clause([searchedByCustoms(a).negate()]);

# 3.  $\neg \text{drugDealer}(x) \vee \neg \text{VIP}(x)$ 
clause3 = Clause([drugDealer(x).negate(), VIP(x).negate()]);

# 4.  $\text{customsOfficial}(b) \wedge \text{drugDealer}(b)$ 
clause4a = Clause([customsOfficial(b)]);
clause4b = Clause([drugDealer(b)]);

# 收集所有子句
clauses = [clause1, clause2a, clause2b, clause2c, clause2d, clause3, clause4a,
            clause4b];

```

推理

```

def unify(x, y, subst):
    ...
    **description**
    合一算法, 尝试合一两个项或谓词, 并返回替换列表。

    **params**
    x: 第一个项或谓词。
    y: 第二个项或谓词。
    subst: 当前的替换列表。

```

```
    **returns**
    更新的替换列表，如果无法合一则返回 None。
    '''
    if subst is None:
        return None;
    elif x == y:
        return subst;
    elif isinstance(x, Term) and x.isVariable():
        return unifyVar(x, y, subst);
    elif isinstance(y, Term) and y.isVariable():
        return unifyVar(y, x, subst);
    elif isinstance(x, Predicate) and isinstance(y, Predicate):
        if x.name != y.name or len(x.args) != len(y.args):
            return None;
        for a, b in zip(x.args, y.args):
            subst = unify(a, b, subst);
            if subst is None:
                return None;
        return subst;
    else:
        return None;

def unifyVar(var, x, subst):
    '''
    **description**
    合一变量。

    **params**
    var: 变量项。
    x: 另一个项。
    subst: 当前的替换列表。

    **returns**
    更新的替换列表，如果无法合一则返回 None。
    '''
    if var in subst:
        return unify(subst[var], x, subst);
    elif x in subst:
        return unify(var, subst[x], subst);
    else:
        if occursCheck(var, x, subst):
            return None;
        subst[var] = x;
        return subst;

def occursCheck(var, x, subst):
    '''
    **description**
    检查变量是否出现在替换中，防止无限递归。

    **params**
    var: 变量项。
    x: 另一个项。
```

```
    subst: 当前的替换列表。

    **returns**
    如果出现返回 True, 否则返回 False.
    '''
    if var == x:
        return True;
    elif isinstance(x, Term) and x.isVariable() and x in subst:
        return occursCheck(var, subst[x], subst);
    return False;

def substitute(predicate, subst):
    '''
    **description**
    在谓词中应用替换。

    **params**
    predicate: 需要替换的谓词。
    subst: 替换列表。

    **returns**
    应用替换后的新谓词。
    '''
    newArgs = [];
    for arg in predicate.args:
        if arg in subst:
            newArg = subst[arg];
            while newArg in subst:
                newArg = subst[newArg];
            newArgs.append(newArg);
        else:
            newArgs.append(arg);
    return Predicate(predicate.name, newArgs, predicate.isNegative);

def substituteClause(clause, subst):
    '''
    **description**
    在子句中应用替换。

    **params**
    clause: 需要替换的子句。
    subst: 替换列表。

    **returns**
    应用替换后的新子句。
    '''
    newPredicates = [];
    for pred in clause.predicates:
        newPred = substitute(pred, subst);
        newPredicates.append(newPred);
    return Clause(newPredicates);

def resolution(clauses):
    '''
```

```

    **description**
    主归结算法，尝试在子句集合中证明矛盾（导出空子句）。

    **params**
    clauses: 初始子句集合。

    **returns**

    布尔值，表明是否成功证明矛盾。
    ...

    new = set();
    steps = 0; # 初始化推理步数计数器
    while True:
        n = len(clauses);
        pairs = [(clauses[i], clauses[j]) for i in range(n) for j in range(i + 1,
n)];
        for (ci, cj) in pairs:
            # 输出当前正在处理的子句对
            print(f"\n正在尝试归结子句对: {ci} 和 {cj}");
            resolvents = resolve(ci, cj);
            steps += 1; # 每次归结操作计为一步
            for resolvent in resolvents:
                # 明确标记涉及的子句、操作及结果
                print(f"步骤 {steps}: 将子句 {ci} 和 {cj} 归结得出新子句:
{resolvent}");
                if resolvent.isEmpty():
                    print("导出空子句，证明成功！");
                    print(f"推理结束，总步数: {steps}");
                    return True;
                new.add(resolvent);
            newClauses = new - set(clauses);
            if not newClauses:
                print("无法继续归结，证明失败。");
                print(f"推理结束，总步数: {steps}");
                return False;
            clauses.extend(newClauses);

def resolve(clause1, clause2):
    ...
    **description**
    尝试对两个子句进行归结，返回新的子句列表。

    **params**
    clause1: 第一个子句。
    clause2: 第二个子句。

    **returns**
    归结后的新子句列表。
    ...

    resolvents = [];
    for pred1 in clause1.predicates:
        for pred2 in clause2.predicates:
            if pred1.name == pred2.name and pred1.isNegative != pred2.isNegative:

```



```
        subst = {};
        pred1Subst = copy.deepcopy(pred1);
        pred2Subst = copy.deepcopy(pred2);
        # 明确指出使用unify处理的两个谓词
        print(f"尝试合一谓词 {pred1Subst} 和 {pred2Subst} 使用替换列表
{subst}");
        subst = unify(pred1Subst, pred2Subst, subst);
        if subst is not None:
            # 输出替换的具体操作
            print(f"替换列表更新为 {subst}, 应用到子句 {clause1} 和
{clause2}");
            newPredicates = [substitute(p, subst) for p in
clause1.predicates + clause2.predicates
                            if p != pred1 and p != pred2];
            newClause = Clause(newPredicates);
            resolvents.append(newClause);
        return resolvents;

# 开始推理
print("初始子句集合: ");
for clause in clauses:
    print(clause);

print("\n开始归结推理...");

result = resolution(clauses);
```