一、实验题目

归结原理

二、实验内容

2.1 算法原理

基本概念

归结原理是一种自动推理技术，用于判断一个逻辑公式集（知识库）是否蕴含某个特定的公式。这种方法特别适用于命题逻辑和一阶谓词逻辑。

在逻辑领域中，一个**子句**是一系列文字的析取（即"或"关系的连接），例如 $A \vee \neg B$。一个**文字**是一个原子或其否定，其中**原子**是最基本的不可分割的公式单位。

命题逻辑归结算法

命题逻辑归结算法的核心思想是通过反复应用归结规则来简化公式，直到得到一个空子句或无法进一步归结为止。归结规则如下所示：

如果有两个子句 C_1：$A \vee P$ 和 C_2：$\neg A \vee Q$，其中 A 是一个文字，P 和 Q 是文字的析取（可能为空），那么我们可以得到一个新的子句 C：$P \vee Q$。

这个过程称为**归结**，新产生的子句 C 称为 C_1 和 C_2 的**归结子**。

MGU算法

在一阶逻辑中，变量的引入使得我们需要通过合一算法来确定是否存在一个变量的替换，使得两个谓词能够匹配。**最一般合一（MGU）**是这样的变量替换，它能使得两个表达式相等，且在所有能使这两个表达式相等的替换中是最一般的。

如果有两个表达式 E_1 和 E_2，MGU是一个替换集合 σ，使得 $E_1\sigma = E_2\sigma$，且对于任何其他使 E_1 和 E_2 相等的替换 σ'，都存在替换 θ 使得 $\sigma'\theta = \sigma$。

一阶逻辑归结算法

一阶逻辑归结算法是基于归结原理的自动推理技术，用于判断一组一阶逻辑公式（知识库 KB）是否蕴含某个特定公式 α。算法主要包括以下步骤：

1. **转换为Skolem标准形**：

   将所有公式转换为**前束范式**（所有量词出现在公式前部）。

   应用**Skolem化**消除存在量词（$\exists$）。通过引入Skolem函数或常量，将量词全部转化为全称量词（$\forall$）的形式。

2. **子句化**：

   将Skolem标准形的公式转换为一组子句的合取形式，其中每个子句是一组文字的析取。

   此步骤包括消除全称量词、将公式转换为合取范式（CNF），然后提取子句。

3. **归结推理过程**：

   输入：知识库 KB 和查询 α。

   输出：判断 $KB \models \alpha$ 是否成立。

步骤：

1. 将 $KB \cup \{\neg\alpha\}$ 转换为一组子句 S。
2. 对于 S 中的任意两个子句 C_1 和 C_2，如果它们可以通过MGU σ 归结得到子句 C，则 $C\sigma$ 加入到 S 中。
3. 重复步骤2，直到生成空子句或无法进一步归结为止。
4. 如果生成了空子句，则 $KB \models \alpha$；否则，$KB \not\models \alpha$。

2.2 关键代码展示

在本次实验中，为了规范起见，我们将归结求解器封装为一个类`ResolutionFOL`。

接下来，我们对`ResolutionFOL`类的不同部分逐个分析。

辅助函数

`is_variable`函数

`is_variable`函数用于判断一个字符串是否为变量。

- **输入**：`term`，一个字符串，代表待检测的项。
- **输出**：布尔值（`True`或`False`）。如果`term`符合变量的命名规则，则返回`True`，否则返回`False`。

```
def is_variable(self, term):
    return re.match(r'^[t-z]{1,2}$', term) is not None
```

首先，我们定义变量的规则为：必须由小写字母`t`到`z`之间的字符组成，且长度为1到2个字符。这显然是符合实际的。为了根据这个定义判断，我们引入正则表达式`'^[t-z]{1,2}$'`，检测`term`是否符合变量的命名规则。

然后，我们使用`re.match`函数检查`term`是否完全匹配这个正则表达式。`re.match`函数从字符串的开始位置开始匹配正则表达式，如果整个字符串匹配，则返回一个匹配对象；否则返回`None`。

`parse_predicate`函数

`parse_predicate`函数用于解析逻辑谓词字符串，提取出谓词的名称（包括是否为否定谓词）和它的参数列表。

- **输入**：`predicate`，一个字符串，代表逻辑谓词，可能包含否定符号`~`、谓词名称以及圆括号内的参数列表。
- **输出**：一个元组`(name, args)`，其中`name`是谓词名称（包括可能的否定符号），`args`是一个列表，包含谓词的所有参数。

```python
def parse_predicate(self, predicate):
    negation = False

    if predicate.startswith('~'):
        negation = True
        predicate = predicate[1:]
    name_end = predicate.find('(')
    name = predicate[:name_end]
    if negation:
        name = '~' + name
    args = predicate[name_end + 1:-1].split(',')

    return (name, args)
```

首先，初始设定`negation`标志为`False`。如果谓词以否定符号`~`开头，将`negation`设为`True`并从谓词中移除该否定符号，后续再进一步处理。

然后，通过查找第一个左括号`(`来确定谓词名称的结束位置（`name_end`），然后从谓词字符串中提取谓词名称`name`。如果原谓词被否定，将否定符号`~`添加到提取的谓词名称前。

最后，我们通过左括号的位置找到参数列表部分，然后去除末尾的右括号`)`，最后通过分割逗号`,`将参数字符串分割为参数列表`args`。

`apply_mgu`函数

`apply_mgu`函数用于尝试对两个谓词应用最一般合一（MGU）算法，并且可能的话，生成变量替换的映射。

- **输入**：`predicate1`和`predicate2`，两个字符串，代表要进行合一的两个逻辑谓词。
- **输出**：`substitutions`，一个字典，表示变量替换的映射。如果合一不可能，则返回`None`。

```python
def apply_mgu(self, predicate1, predicate2):
    _, args1 = self.parse_predicate(predicate1)
    _, args2 = self.parse_predicate(predicate2)

    substitutions = {}
    for arg1, arg2 in zip(args1, args2):
        if arg1 == arg2:
            continue
        if '(' in arg1 and '(' in arg2:
            inner_substitutions = self.apply_mgu(arg1, arg2)
            if inner_substitutions is None:
                return None
            substitutions.update(inner_substitutions)
        elif self.is_variable(arg1) and self.is_variable(arg2):
            return None
        elif self.is_variable(arg1):
            substitutions[arg1] = arg2
        elif self.is_variable(arg2):
            substitutions[arg2] = arg1
        else:
            return None
    return substitutions
```

在这个函数中，首先解析两个谓词，获取它们的参数列表。然后，对这两个列表中的每一对参数进行比较和尝试合一：

1. 如果两个参数完全相同，就跳过这对参数。
2. 如果两个参数都是复杂表达式（即含有括号），递归地对这两个表达式应用MGU算法。如果递归合一失败，则整体合一失败。
3. 如果一个参数是变量而另一个不是，尝试将变量替换为另一个参数。如果两个参数都是变量，不进行合一。
4. 如果上述条件都不适用，且两个参数不相等，合一失败。

注意，由于我们在上面第二种情况中使用递归解决，我们的MGU算法可以处理多层函数作为谓词元素的情况，如$f(g(h))$。

`can_resolve`函数

`can_resolve`函数用于判断两个子句是否可以通过归结原理进行解析，即是否存在至少一对互补的谓词，使得这两个子句可以合一并产生新的子句。

- **输入**：`clause1`和`clause2`，两个子句，每个子句都是一系列谓词的集合。
- **输出**：布尔值和元组。如果找到至少一对可以合一的互补谓词，返回`True`和这对谓词；否则，返回`False`和空元组。

```
def can_resolve(self, clause1, clause2):
    for pred1 in clause1:
        for pred2 in clause2:
            name1, args1 = self.parse_predicate(pred1)
            name2, args2 = self.parse_predicate(pred2)

            if (name1.startswith('~') != name2.startswith('~')) and
(name1.lstrip('~') == name2.lstrip('~')):
                substitutions = self.apply_mgu(pred1, pred2)
                if substitutions is not None:
                    return True, (pred1, pred2)
    return False, ()
```

首先遍历`clause1`和`clause2`中的每对谓词。检查它们是否形成互补对，即一个谓词是另一个谓词的否定形式（例如，`A`和`~A`）。这是通过检查谓词名称的开始是否有`~`来实现的，并且在去除可能的否定符号后，比较两个谓词的名称是否相等。

如果找到一对互补谓词，使用`apply_mgu`尝试对它们进行合一。如果合一成功（即`apply_mgu`返回非`None`的替换映射），则函数返回`True`和该对谓词，表示这两个子句可以通过归结原理解析。如果遍历完所有谓词对都没有找到可以合一的互补对，函数返回`False`和空元组，表示这两个子句无法解析。

子句操作和格式化

`apply_substitutions_to_clause`函数

`apply_substitutions_to_clause`函数用于在一个子句中应用一组变量替换，生成一个新的子句，其中包含了所有替换后的谓词。

- **输入**：

  `clause`：一个子句，表示为谓词的列表；

  `substitutions`：一个字典，表示变量到它们替换值的映射。

- **输出**：新的子句，包含了应用了所有变量替换的谓词。

```
def apply_substitutions_to_clause(self, clause, substitutions):
    new_clause = []
    for predicate in clause:
        name, args = self.parse_predicate(predicate)
        new_args = []
        for arg in args:
            for var, sub in substitutions.items():
                arg = re.sub(r'\b' + var + r'\b', sub, arg)
            new_args.append(arg)
        new_predicate = f"{name[1:] if name.startswith('~') else name}({','.join(new_args)})"
        if name.startswith('~'):
            new_predicate = f"~{new_predicate}"
        new_clause.append(new_predicate)
    return tuple(new_clause)
```

具体来讲，对子句中的每个谓词执行以下步骤：

1. 使用`parse_predicate`解析谓词名称和参数。
2. 对每个参数，遍历`substitutions`字典中的所有变量替换规则。使用正则表达式`re.sub(r'\b' + var + r'\b', sub, arg)`来确保只替换完整的变量名，而不是变量名的一部分。这是通过单词边界`\b`来实现的。
3. 将替换后的参数重新组合成新的谓词字符串，如果原谓词为否定形式则加上`~`。
4. 将所有处理后的新谓词收集到新子句列表中，并最终将其转换为元组形式返回。

`format_clause_index`函数

`format_clause_index`函数用于格式化子句索引，用于生成步骤字符串中的子句引用。当子句包含多个谓词时，添加适当的后缀（如`a,b`）以区分不同的谓词。

- **输入**：

  `clause`：当前处理的子句；

  `predicate_index`：子句中谓词的索引；

  `total_predicates`：子句中谓词的总数。

- **输出**：格式化后的子句索引字符串。如果子句中只有一个谓词，直接返回该子句的基本索引；如果有多个谓词，则在基本索引后附加一个小写字母，以区分不同的谓词。

```
def format_clause_index(self, clause, predicate_index, total_predicates):
    base_index = self.clause_to_step[clause]
    if total_predicates > 1:
        return f"{base_index}{chr(97 + predicate_index)}"
    return base_index
```

首先，通过`clause`从`clause_to_step`字典中获取子句的基本索引（`base_index`）。这个索引表示了子句在推理过程中得到的顺序。

接下来，考虑子句中谓词的总数（`total_predicates`）：

- 如果子句中只包含一个谓词（`total_predicates`为1），则直接返回子句的`base_index`。

- 如果子句中包含多个谓词，需要对每个谓词分别标记以区分它们。这里通过在`base_index`后附加一个小写字母来实现，该字母根据谓词在子句中的索引（`predicate_index`）计算得出（`chr(97 + predicate_index)`）。这里使用ASCII码中的小写字母进行编码，索引0对应字母`a`，索引1对应`b`，依此类推。

主推理逻辑

这部分封装到`__call__`方法中，是归结推理算法的核心。它通过反复查找可归结的子句对并应用归结规则，直到推理过程结束（发现空子句或无法进一步归结），并返回推理的步骤。

- **输入**：`KB`，一个知识库，表示为一组子句。
- **输出**：推理步骤列表，每个步骤是一个字符串，记录了归结的步骤。

```
def __call__(self, KB):
    # 初始化
    self.steps = []
    self.new_clauses = set()
    self.resolved_pairs = set()
    self.clause_to_step = {}
    self.clauses = list(KB)
    self.clause_to_step = {clause: str(i + 1) for i, clause in
enumerate(self.clauses)}

    # 记录初始子句
    for clause in self.clauses:
        self.steps.append(f"{self.clause_to_step[clause]} {clause}")

    # 归结过程
    while True:
        possible_resolutions = []
        # 遍历子句对，尝试归结
        for i, clause1 in enumerate(self.clauses):
            for j, clause2 in enumerate(self.clauses[i + 1:], start=i + 1):
                if (i, j) not in self.resolved_pairs and
self.can_resolve(clause1, clause2)[0]:
                    self.resolved_pairs.add((i, j))
                    _, predicates_to_resolve = self.can_resolve(clause1,
clause2)
                    substitutions = self.apply_mgu(predicates_to_resolve[0],
predicates_to_resolve[1])
                    unified = '{' + ', '.join([f"{k}={v}" for k, v in
substitutions.items()]) + '}' if substitutions else ''
                    clause1_substituted =
self.apply_substitutions_to_clause(clause1, substitutions)
                    clause2_substituted =
self.apply_substitutions_to_clause(clause2, substitutions)
                    resolvents = set(clause1_substituted +
clause2_substituted) -
set(self.apply_substitutions_to_clause(predicates_to_resolve, substitutions))
                    resolvents = tuple(sorted(resolvents, key=lambda x:
x.replace('~', '')))

                    # 记录归结步骤
```

```
                    i_formatted = self.format_clause_index(clause1,
clause1.index(predicates_to_resolve[0]), len(clause1))
                    j_formatted = self.format_clause_index(clause2,
clause2.index(predicates_to_resolve[1]), len(clause2))
                    step_str = f"R[{i_formatted},{j_formatted}]{unified} =
{resolvents}"
                    # 如果推出NULL则结束
                    if not resolvents:
                        self.steps.append(f"{len(self.steps) + 1}
{step_str}")
                        return self.steps
                    possible_resolutions.append((step_str, resolvents))
        # 如果没有新的子句则结束
        if not possible_resolutions:
            break

        # 更新子句集和步骤记录
        for step_str, resolvents in possible_resolutions:
            if resolvents not in set(self.clauses) and resolvents not in
self.new_clauses:
                self.new_clauses.add(resolvents)
                self.steps.append(f"{len(self.steps) + 1} {step_str}")
                self.clause_to_step[resolvents] = str(len(self.steps))

        self.clauses.extend(self.new_clauses)
        self.new_clauses.clear()

    return self.steps
```

首先，我们进行一些初始化工作，包括定义归结过程用于记录推理步骤的列表、新产生的子句集合、已解析的子句对集合以及子句到步骤编号的映射。然后，对知识库中的每个子句进行编号并记录初始步骤。

接下来，进入推理主循环，不断尝试对子句进行归结操作。对每一对子句，检查它们是否可以进行归结（即是否存在互补的谓词对），并应用最一般合一算法来计算变量的替换。如果能够成功归结，计算得到的新子句（resolvents）会被加入到可能的解决方案中，并记录相应的归结步骤。

如果在某次迭代中没有新的归结结果产生，或者某次迭代中归结产生空子句，说明已经无法进一步推进推理过程，循环终止。

最后，方法返回一个记录了所有归结步骤的列表。

2.3 创新点

封装`ResolutionFOL`类

面向对象是一种编程范式，它通过将数据和处理数据的方法组织成对象，以此来模拟现实世界中的事物和行为，从而提高代码的重用性、灵活性和可维护性。

在编码过程中，我们遵循面向对象的思想，把`ResolutionFOL`封装为一个类，使得代码结构更为清晰、模块化，还为使用和扩展提供了极大的便利。

- **面向对象**

通过面向对象的设计，我们能够将归结推理过程中涉及的数据和操作封装在一个独立的实体中。这样不仅有助于隔离不同功能的实现细节，还使得代码的可读性和可维护性大大提高。每个`ResolutionFOL`实例都可以看作是一个独立的推理求解器，拥有自己的知识库、推理历史。

- **易于实例化和复用**

通过封装`ResolutionFOL`类，我们在实例化一个求解器后，能够针对不同的知识库执行归结推理过程，如下例所示。

特别地，我们把主推理逻辑封装到`__call__`方法中，使得`ResolutionFOL`实例可以被直接调用，接收一个知识库（KB）作为参数，并执行归结推理过程，返回推理的步骤。这种实例化的能力极大地增强了代码的复用性，避免了重新实现或复制推理逻辑。

```
# 类的实例化
solver = ResolutionFOL()

# 调用实例化进行推理
KB0 = {...}
resolution_steps_kb0 = solver(KB0)

# 代码复用
KB1 = {...}
resolution_steps_kb1 = solver(KB1)
```

正则表达式

正则表达式是一种强大的文本处理工具，它允许我们定义一套规则来匹配、查找、替换或分割字符串，从而实现复杂的文本分析和处理任务。

在实现`ResolutionFOL`类的过程中，我们在`is_variable`方法和`apply_substitutions_to_clause`方法中引入正则表达式，使得归结推理算法能够更精确地处理变量识别、替换这两个关键操作。

`is_variable`方法

```
def is_variable(self, term):
    return re.match(r'^[t-z]{1,2}$', term) is not None
```

在`is_variable`方法中，正则表达式`r'^[t-z]{1,2}$'`被用于判断一个字符串是否符合变量的命名规则。

- `^`表示字符串的开始。
- `[t-z]`表示匹配任何小写字母`t`到`z`之间的字符。
- `{1,2}`表示前面的字符类（即`t`到`z`之间的任何字符）出现1到2次。
- `$`表示字符串的结束。

通过引入正则表达式，`is_variable`方法能够快速准确地区分变量和其他类型的符号，而无需采取更加复杂的编码方式。

`apply_substitutions_to_clause`方法

```
def apply_substitutions_to_clause(self, clause, substitutions):
    new_clause = []
    for predicate in clause:
        name, args = self.parse_predicate(predicate)
```

```
        new_args = []
        for arg in args:
            for var, sub in substitutions.items():
                arg = re.sub(r'\b' + var + r'\b', sub, arg)
            new_args.append(arg)
        new_predicate = f"{name[1:] if name.startswith('~') else name}
({','.join(new_args)})"
        if name.startswith('~'):
            new_predicate = f"~{new_predicate}"
        new_clause.append(new_predicate)
    return tuple(new_clause)
```

在`apply_substitutions_to_clause`方法中，正则表达式`r'\b' + var + r'\b'`被用于实现变量的精确替换。

- `re.sub(pattern, repl, string)`是Python中的正则表达式替换函数，用于在字符串中查找与模式`pattern`匹配的所有子串，将它们替换为`repl`，并返回替换后的字符串。
- `r'\b' + var + r'\b'`构成了这里的模式`pattern`。其中`r'\b'`表示单词边界，确保只匹配完整的单词。
- `var`是要被替换的变量名，它被插入到两个单词边界符`\b`之间，确保只有完整匹配`var`的单词才会被替换。
- `sub`是替换`var`的新值。
- `arg`是原始的参数字符串，在这个字符串中进行查找和替换操作。

通过引入正则表达式，我们确保了`apply_substitutions_to_clause`方法精确替换变量，而不会错误地替换变量名的一部分。

递归使用最一般合一

递归是一种编程思想，通过在函数内调用自身来解决问题，允许代码以简洁的方式处理复杂的任务，尤其是那些可以分解为更小、相似问题的任务。

在`apply_mgu`方法中，我们实现了一个递归操作来处理嵌套谓词的最一般合一（MGU）算法。这样，当遇到形如`f(x)`甚至`f(g(y))`这种多层函数作为谓词元素时，我们的MGU算法依然适用，保证了算法的通用性。

```
def apply_mgu(self, predicate1, predicate2):
    _, args1 = self.parse_predicate(predicate1)
    _, args2 = self.parse_predicate(predicate2)

    substitutions = {}
    for arg1, arg2 in zip(args1, args2):
        if arg1 == arg2:
            continue

        # 递归处理嵌套谓词
        if '(' in arg1 and '(' in arg2:
            inner_substitutions = self.apply_mgu(arg1, arg2)
            if inner_substitutions is None:
                return None
            substitutions.update(inner_substitutions)
        elif self.is_variable(arg1) and self.is_variable(arg2):
            return None
```

```
        elif self.is_variable(arg1):
            substitutions[arg1] = arg2
        elif self.is_variable(arg2):
            substitutions[arg2] = arg1
        else:
            return None

    return substitutions
```

- 首先检查两个谓词参数`arg1`和`arg2`中是否都含有嵌套结构，即是否包含左括号`(`。如果存在左括号，说明参数可能是一个复杂的表达式，而非简单的变量或常量。
- 如果两个参数都是复杂表达式，那么尝试对这两个嵌套表达式递归地应用`apply_mgu`函数，以尝试找到它们之间可能的合一替换（`inner_substitutions`）。
- 如果在尝试合一这两个嵌套表达式时发现它们无法合一（即`apply_mgu`返回`None`），则整个合一过程失败，函数返回`None`。
- 如果合一成功，即找到了有效的`inner_substitutions`，则将这些内部替换更新到当前的替换映射`substitutions`中。

通过这里的递归操作，我们的`apply_mgu`函数能够有效处理多层函数作为谓词参数的情况，这是其他代码难以做得到的。我们后面会准备一些这样的测例进行测试。

三、实验结果及分析

我们先初始化一个求解器`ResolutionFOL`。

```
solver = ResolutionFOL()
```

接下来，简单准备一些测例，对`ResolutionFOL`类进行测试。我们一共准备了五个测例，我会逐个分析这些测例的特点、难点，并展示每个测例对应的输出。

测例1

```
KB1 = {('GradStudent(sue)',),
       ('~GradStudent(x)', 'Student(x)'),
       ('~Student(x)', 'HardWorker(x)'),
       ('~HardWorker(sue)',)}

print('Test case # 1')
resolution_steps1 = solver(KB1)
for step in resolution_steps1:
    print(step)
```

这个测试样例的难点在于，`ResolutionFOL`类需要分辨出`sue`和`x`哪个是常量，哪个是变量。

在上面创新点部分已经详细讲过，我们通过引入正则表达式，可以完美解决这个问题。

测例输出如下，符合我们的预期：

```
Test case # 1
1 ('~Student(x)', 'HardWorker(x)')
2 ('~GradStudent(x)', 'Student(x)')
3 ('GradStudent(sue)',)
4 ('~HardWorker(sue)',)
5 R[1a,2b] = ('~GradStudent(x)', 'HardWorker(x)')
6 R[1b,4]{x=sue} = ('~Student(sue)',)
7 R[2a,3]{x=sue} = ('Student(sue)',)
8 R[6,7] = ()
```

测例2

```python
KB2 = {('A(tony)',),
       ('A(mike)',),
       ('A(john)',),
       ('L(tony,rain)',),
       ('L(tony,snow)',),
       ('~A(x)', 'S(x)', 'C(x)'),
       ('~C(y)', '~L(y,rain)'),
       ('L(z,snow)', '~S(z)'),
       ('~L(tony,u)', '~L(mike,u)'),
       ('L(tony,v)', 'L(mike,v)'),
       ('~A(w)', '~C(w)', 'S(w)')}

print('Test case # 2')
resolution_steps2 = solver(KB2)
for step in resolution_steps2:
    print(step)
```

这个测例的难点在于，对于`L(y,rain)`这样多于一个元素的表达式，`ResolutionFOL`类需要正确处理并且替换。

在上面关键代码展示已经讲过，我们对于一条表达式，首先会通过`parse_predicate`函数提取谓词和里面的元素。

所以别说两个元素，多少个元素我们都可以正确提取并处理。

测例的输出如下。这里的输出比较多，是因为计算机并不像人一样可以引入那么多先验知识，每次循环的时候直接挑选特定的语句做归结。但是事实上，每一条归结的语句都是正确的。我已经用红色把得到结论的关键部分标出。

```
Test case # 2
1 ('A(mike)',)
2 ('~L(tony,u)', '~L(mike,u)')
3 ('L(tony,rain)',)
4 ('A(john)',)
5 ('L(tony,v)', 'L(mike,v)')
6 ('A(tony)',)
7 ('~A(w)', '~C(w)', 'S(w)')
8 ('~A(x)', 'S(x)', 'C(x)')
9 ('L(z,snow)', '~S(z)')
10 ('~C(y)', '~L(y,rain)')
11 ('L(tony,snow)',)
12 R[1,7a]{w=mike} = ('~C(mike)', 'S(mike)')
```

```
13 R[1,8a]{x=mike} = ('C(mike)', 'S(mike)')
14 R[2a,3]{u=rain} = ('~L(mike,rain)',)
15 R[2a,9a]{z=tony, u=snow} = ('~L(mike,snow)', '~S(tony)')
16 R[2a,11]{u=snow} = ('~L(mike,snow)',)
17 R[3,10b]{y=tony} = ('~C(tony)',)
18 R[4,7a]{w=john} = ('~C(john)', 'S(john)')
19 R[4,8a]{x=john} = ('C(john)', 'S(john)')
20 R[5a,10b]{y=tony, v=rain} = ('~C(tony)', 'L(mike,rain)')
21 R[6,7a]{w=tony} = ('~C(tony)', 'S(tony)')
22 R[6,8a]{x=tony} = ('C(tony)', 'S(tony)')
23 R[2b,20b]{u=rain} = ('~C(tony)', '~L(tony,rain)')
24 R[5b,15a]{v=snow} = ('L(tony,snow)', '~S(tony)')
25 R[7b,22a]{w=tony} = ('~A(tony)', 'S(tony)')
26 R[7b,13a]{w=mike} = ('~A(mike)', 'S(mike)')
27 R[7c,15b]{w=tony} = ('~A(tony)', '~C(tony)', '~L(mike,snow)')
28 R[7b,19a]{w=john} = ('~A(john)', 'S(john)')
29 R[8c,20a]{x=tony} = ('~A(tony)', 'L(mike,rain)', 'S(tony)')
30 R[8b,15b]{x=tony} = ('~A(tony)', 'C(tony)', '~L(mike,snow)')
31 R[9b,22b]{z=tony} = ('C(tony)', 'L(tony,snow)')
32 R[9b,13b]{z=mike} = ('C(mike)', 'L(mike,snow)')
33 R[9a,16]{z=mike} = ('~S(mike)',)
34 R[9b,21b]{z=tony} = ('~C(tony)', 'L(tony,snow)')
35 R[9b,18b]{z=john} = ('~C(john)', 'L(john,snow)')
36 R[9a,15a]{z=mike} = ('~S(mike)', '~S(tony)')
37 R[9b,12b]{z=mike} = ('~C(mike)', 'L(mike,snow)')
38 R[9b,19b]{z=john} = ('C(john)', 'L(john,snow)')
39 R[10a,22a]{y=tony} = ('~L(tony,rain)', 'S(tony)')
40 R[10a,13a]{y=mike} = ('~L(mike,rain)', 'S(mike)')
41 R[10b,20b]{y=mike} = ('~C(mike)', '~C(tony)')
42 R[10a,19a]{y=john} = ('~L(john,rain)', 'S(john)')
43 R[22a,17] = ('S(tony)',)
44 R[22a,20a] = ('L(mike,rain)', 'S(tony)')
45 R[22b,15b] = ('C(tony)', '~L(mike,snow)')
46 R[13a,12a] = ('S(mike)',)
47 R[21b,15b] = ('~C(tony)', '~L(mike,snow)')
48 R[18a,19a] = ('S(john)',)
49 R[33,46] = ()
```

测例3

```
KB3 = {('On(tony,mike)',),
       ('On(mike,john)',),
       ('Green(tony)',),
       ('~Green(john)',),
       ('~On(xx,yy)', '~Green(xx)', 'Green(yy)')}

print('Test case # 3')
resolution_steps3 = solver(KB3)
for step in resolution_steps3:
    print(step)
```

可以看成是测例1和测例2的结合版。

- 需要 `ResolutionFOL` 类正确判断 `xx,yy,mike,tony,john` 哪个是变量，哪个是常量。

我们通过正则表达式处理了这一点。

- 需要`ResolutionFOL`类正确处理多元素的谓词。

  我们借助`parse_predicate`等辅助函数处理了这一点。

此外，`ResolutionFOL`类还需要保证不会出现`xx`和`yy`相互替换的情况，否则就会死循环了。

测例输出如下：

```
Test case # 3
1 ('~Green(john)',)
2 ('~On(xx,yy)', '~Green(xx)', 'Green(yy)')
3 ('On(mike,john)',)
4 ('Green(tony)',)
5 ('On(tony,mike)',)
6 R[1,2c]{yy=john} = ('~Green(xx)', '~On(xx,john)')
7 R[2a,3]{xx=mike, yy=john} = ('Green(john)', '~Green(mike)')
8 R[2b,4]{xx=tony} = ('Green(yy)', '~On(tony,yy)')
9 R[2a,5]{xx=tony, yy=mike} = ('Green(mike)', '~Green(tony)')
10 R[1,7a] = ('~Green(mike)',)
11 R[1,8a]{yy=john} = ('~On(tony,john)',)
12 R[2b,7a]{xx=john} = ('~Green(mike)', 'Green(yy)', '~On(john,yy)')
13 R[2b,9a]{xx=mike} = ('~Green(tony)', 'Green(yy)', '~On(mike,yy)')
14 R[4,9b] = ('Green(mike)',)
15 R[7a,6a]{xx=john} = ('~Green(mike)', '~On(john,john)')
16 R[7b,8a]{yy=mike} = ('Green(john)', '~On(tony,mike)')
17 R[7b,9a] = ('Green(john)', '~Green(tony)')
18 R[6a,9a]{xx=mike} = ('~Green(tony)', '~On(mike,john)')
19 R[8a,9b]{yy=tony} = ('Green(mike)', '~On(tony,tony)')
20 R[14,10] = ()
```

测例4（附加）

以上都是课件上比较简单的的测例。但是事实上，课件的测例并不包含`F(g(x))`这种谓词中出现函数作为元素的情况，导致测例不够全面，不能充分展示我们代码的通用性。我们因此额外补充测例4、测例5。

```python
KB4 = {('I(bb)',),
       ('U(aa,bb)',),
       ('~F(u)',),
       ('~I(y)', '~U(x,y)', 'F(f(z))'),
       ('~I(v)', '~U(w,v)', 'E(w,f(w))')}

print('Test case # 4')
resolution_steps4 = solver(KB4)
for step in resolution_steps4:
    print(step)
```

这个测例主要测试多元谓词中，出现简单函数（一层函数）作为元素的情况。具体包括：

- 一元谓词：`F(f(z))`
- 二元谓词：`E(w,f(w))`

在上面的创新点部分已经详细讲过，我们通过递归调用MGU函数，可以完美解决这个问题。

测例的输出如下。我已经用红色把关键部分标出。可以看到，会出现`{u=f(z)}`这样的替换，符合我们的预期。

```
Test case # 4
1 ('I(bb)',)
2 ('~I(y)', '~U(x,y)', 'F(f(z))')
3 ('U(aa,bb)',)
4 ('~I(v)', '~U(w,v)', 'E(w,f(w))')
5 ('~F(u)',)
6 R[1,2a]{y=bb} = ('F(f(z))', '~U(x,bb)')
7 R[1,4a]{v=bb} = ('E(w,f(w))', '~U(w,bb)')
8 R[2b,3]{x=aa, y=bb} = ('F(f(z))', '~I(bb)')
9 R[2c,5]{u=f(z)} = ('~I(y)', '~U(x,y)')
10 R[3,4b]{w=aa, v=bb} = ('E(aa,f(aa))', '~I(bb)')
11 R[1,8b] = ('F(f(z))',)
12 R[1,10b] = ('E(aa,f(aa))',)
13 R[1,9a]{y=bb} = ('~U(x,bb)',)
14 R[3,9b]{x=aa, y=bb} = ('~I(bb)',)
15 R[1,14] = ()
```

测例5（附加）

```python
KB5 = {('~P(aa)',),
       ('P(z)', '~Q(f(z),f(u))'),
       ('Q(x,f(g(y)))', 'R(s)'),
       ('~R(t)',)}

print('Test case # 5')
resolution_steps5 = solver(KB5)
for step in resolution_steps5:
    print(step)
```

这个测例主要测试多元谓词中，出现复合函数（这里以两层函数为例）`f(g(y))`作为元素的情况。

同样地，我们通过递归调用MGU函数，可以完美解决这个问题。别说处理`f(g(y))`，再复合多几层都没问题。

测例的输出如下。关键部分用红色标出。可以看到，程序可以正确处理这种情况。

```
Test case # 5
1 ('P(z)', '~Q(f(z),f(u))')
2 ('~R(t)',)
3 ('Q(x,f(g(y)))', 'R(s)')
4 ('~P(aa)',)
5 R[1b,3a]{x=f(z), u=g(y)} = ('P(z)', 'R(s)')
6 R[1a,4]{z=aa} = ('~Q(f(aa),f(u))',)
7 R[2,3b]{t=s} = ('Q(x,f(g(y)))',)
8 R[7,6]{x=f(aa), u=g(y)} = ()
```

四、参考资料

- Python3 正则表达式 | 菜鸟教程 (runoob.com)
- Python3 面向对象 | 菜鸟教程 (runoob.com)