# 中山大学计算机学院人工智能本科生实验报告

课程：Artificial Intelligence

姓名： 学号：

## 一、实验题目

编写程序，实现一阶逻辑归结算法，并用于求解给出的三个逻辑推理问题。

## 二、实验内容

### 1.算法原理

在已经给定$KB$的基础上，在子句集中寻找存在互补文字的子句，并尝试将这两个子句归结。

为了将这两个子句归结，必须使互补文字的谓词的项不是变量、并且两者完全相同，为此需要寻找一个置换使得其能够合一。为此使用最一般合一算法，给出两个互补文字的合法的置换。

给出置换之后，就将这两个子句合一，产生一个新的归结子句，并添加到子句集中。然后开始下一对子句的归结。

### 2.伪代码

在归结算法中，归结原理被拆分为几个子问题，在子问题中又有子问题。各子问题都各有自己的函数解决，各函数之间属于层层调用的关系。以下给出函数的层级：

$$ResolutionFOL\begin{cases}Collision\\SingleStepResolutin\begin{cases}Unify\begin{cases}isContradict\begin{cases}TermExtract\\isVariable->isLower\end{cases}\\MostGeneralUnify\begin{cases}TermExtract\\isVariable->isLower\\Replace\end{cases}\end{cases}\\SingleStepReport\end{cases}\end{cases}$$

为了说明设计各函数(或功能)的必要性，现从顶层向底层说明各函数(或功能)的作用。首先，在这次实验中，我将这个算法封装为一个类，类中的数据成员是 `set[tuple[str]]` ，即以 `str` 表示一个原子公式， `tuple[str]` 是几个原子公式构成的子句， `set[tuple[str]]` 是已有子句构成的子句集。而 `ResolutionFOL` 函数是这个类对用户唯一开放的可调用函数。

归结的第一步是找到存在互补文字的子句，这是 `Collision` 的作用。它接受两个子句 `tuple[str]` ，在其中寻找互补文字，以二元 `tuple[int]` 的形式返回互补文字在两个子句中各自的位置。如果不存在互补文字则返回空元组。

如果存在互补文字，那么这对文字进入单步归结 SingleStepResolution 。这对文字被合一 Unify 之后，按格式生成这次单步归结的报告，即 SingleStepReport 。

Unify 接受两个子句 tuple[str] ，返回一个 list ， list 中第 0 项是 tuple[str] ，即归结出的新子句，第 1 项是 dict[str : str] ，即变量置换规则。

但是文字中存在互补文字不代表它是可合一的。进入 Unify 之后，首先需要判定这对互补文字是否可以直接消去，这是 isContradict 的作用——如果文字中的项存在变量，那么不可消去；若除否定符 ~ 之外完全相同，那么可以直接消去。在可以直接消去的情况下 isContradict 返回 True ，那么 Unify 不会进入 MostGeneralUnify 以寻找变量置换规则、而直接进行合一。否则，在不能直接消去的情况下，就会进入 MGU 获取变量置换规则。

MostGeneralUnify 接受两个原子公式，以字典 dict[str : str] 返回能使这对原子公式合一的变量置换规则。规定只有第一个原子公式的变量能置换为第二原子公式的常量。如果不能合一，则返回空字典。

在 MGU 中，如果找不到变量置换规则即返回空字典，那么 Unify 合一失败，返回空列表——于是单步归结 SingleStepResolution 判定失败，不生成单步归结报告、而返回一个单字符字符串 "F" —— ResolutionFOL 开始下一对子句的单步归结。

如果在 MGU 中找到了变量置换规则，那么单步归结 SingleStepResolution 接收到归结子句与置换规则，将归结子句添加到子句集，并据此产生 str 型的单步归结报告 SingleStepReport 。 ResolutionFOL 接收到报告后，将其编号并添加到自身的日志中。并检测新加入的归结子句是否是空子句，如果是那么结束，否则开始下一对子句的单步归结。

其中，文字互补判定 isContradict 和最小一般合一 MGU 都需要对提取原子命题中的每一项，这就是 TermExtract 的功能；提取出项之后，需要判定项是变量或是常量，这是 isVariable 的功能。 isVariable 通过检测是否存在大写字母、以及小写字母的数量判定项的类型。在 MGU 中，通过 Replace 将第一原子公式的变量置换为常量，从而判定原子公式是否已经等价。

接下来先展示类定义，再从底层向顶层说明各函数的伪代码：

## 1.类定义

```
class Resolution
    构造函数：
    __init__(初始KB:str) -> 无返回值
    归结原理：
    ResolutionFOL() -> list[str]

    以下为静态函数：

    变量判定：
    __isVariable(项:str) -> bool

    原子公式项拆分：
    __TermExtract(原子公式:str) -> list[str]

    原子公式互补判定：
    __isContradict(原子公式1:str, 原子公式2:str) -> bool

    子句互补文字位：
    __Collision(子句1:tuple[str], 子句2:tuple[str]) -> tuple[int]

    互补谓词变量置换：
    __MGU(原子公式1:str, 原子公式2:str) -> dict

    子句合一：
    __Unify(子句1:tuple[str], 子句2:tuple[str], 互补文字位:tuple[int]) -> list

    单步归结：
    __SingleStepResolution(子句集:list[tuple[str]], 索引1:int, 索引2:int, 互补文字位:tuple[int]) -> str
```

## 2.各函数伪代码实现

```
变量判定：
__isVariable(项:str) -> bool:
    if (项的字数小于等于2 and 都是小写字母): return True
    else return False
```

```
原子公式项拆分：
__TermExtract(原子公式:str) -> list[str]:
    截取原子公式括号中内容
    按照','拆分
```

原子公式互补判定：

```
__isContradict(原子公式1:str, 原子公式2:str) -> bool:
    原子1各项, 原子2各项 = __TermExtract(原子公式1), __TermExtract(原子公式2)
    if (__isVariable(原子1各项, 原子2各项)): return False
    if (原子公式1, 原子公式2只相差'~'): return True
```

子句互补文字位：

```
    __Collision(子句1:tuple[str], 子句2:tuple[str]) -> tuple[int]:
    for 子句1中每一原子:
        for 子句2中每一原子:
            if (两原子的谓词相同并且相差'~'): return (两原子在子句中索引)
    return (空元组)
```

互补谓词变量置换：

```
__MGU(原子公式1:str, 原子公式2:str) -> dict:
    置换表 初始置空
    原子1各项, 原子2各项 = __TermExtract(原子公式1), __TermExtract(原子公式2)
    for 谓词中各位置的项:
        if (原子1此项 != 原子2此项):
            if (__isVariable(原子1此项) and not __isVariable(原子2此项)):
                置换表添加置换对{原子1此项 : 原子2此项}
                原子1中此变量 置换为 原子2中此常量
                原子1各项 中 此项 置换为 原子2各项 中 此项
        #其余情况直接跳过
    return 置换表
```

子句合一：

```
__Unify(子句1:tuple[str], 子句2:tuple[str], 互补文字位:tuple[int]) -> list:
    if (__isContradict(子句1[互补文字位[0]], 子句2[互补文字位[1]])):
        #此时子句中存在可以直接消去的量
        归结子句 中加入子句1子句2除互补文字外其余原子
        return [归结子句, {}]
    置换表 = __MGU(子句1[互补文字位[0]], 子句2[互补文字位[1]])
    if (置换表为空): return []
    子句1 按置换表 置换变量
    归结子句 中加入子句1子句2除互补文字外其余原子
        return [归结子句, 置换表]
```

单步归结：

```
__SingleStepResolution(子句集:list[tuple[str]], 索引1:int, 索引2:int, 互补文字位:tuple[int]) -> str:
    子句1, 子句2 = 子句集[索引1], 子句集[索引2]
    [归结子句, 置换表] = __Unify(子句1, 子句2, 互补文字位)
    if (置换表为空 or 归结子句在子句集中已存在): return "F"
    向 子句集 中添加 归结子句
    生成 单步归结报告
    return 单步归结报告
```

归结原理：

```
ResolutionFOL() -> list[str]:
    将原有子句集编号后添加到日志
    while(未归结出空子句):
        for 广度优先搜索:
            互补文字位 = __Collision(子句1, 子句2)
            if (互补文字位非空):
                单步报告 = __SingleStepResolution(子句集, 索引1, 索引2, 互补文字位)
                if (单步报告 == "F"): continue
                将单步报告编号后添加到日志
                if (归结子句为空子句): return 日志
    return 日志
```

构造函数：

```
__init__(初始KB:str) -> None:
    剔除初始KB无效字符
    for 字符串按'),('拆分:
        子字符串按'),'拆分
        此组字符串生成元组
        元组添加到数据成员
```

# 3.关键代码展示

以下代码中省略删除了一些比较简单的函数：

```
class Resolution:
    def ResolutionFOL(self) -> list[str]:
        #对外公开的归结方法
        clause_list = list(self.__clause_set)
        #clause_list.sort()
        count, last = len(clause_list), 0
        output = []
        for i in range(0, count): output.append(str(i + 1) + ' ' + str(clause_list[i]))
        while (clause_list[-1] != ()):
            now = len(clause_list)
            for i in range(last, len(clause_list)):
                for j in range(0, len(clause_list)):
                    collision = Resolution.__Collision(clause_list[i], clause_list[j]) #在子句中是否存在矛盾谓
                    if (collision != ()): #若存在矛盾谓词
                        string = Resolution.__SingleStepResolution(clause_list, i, j, collision) #对这对子句单
                        if (string == "F"): continue #若子句不能归结，开始下一对子句归结
                        count = count + 1 #若能归结
                        output.append(str(count) + ' ' + string) #在归结过程中添加此子句生成报告
                        if (clause_list[-1] == ()): return output #如果已生成空子句，结束归结
            last = now
        return output

    @staticmethod
    def __Collision(clause1 : tuple[str], clause2 : tuple[str]) -> tuple[int]:
        #输入两个子句，返回冲突谓词的位置。无冲突则返回空元组
        for i in range(0, len(clause1)):
            for j in range(0, len(clause2)):
                if (clause1[i][0] != '~' and clause2[j][0] == '~'):
                    if (clause1[i][0] == clause2[j][1]): return (i, j)
                if (clause1[i][0] == '~' and clause2[j][0] != '~'):
                    if (clause1[i][1] == clause2[j][0]): return (i, j)
        return ()

    @staticmethod
    def __MGU(atom1 : str, atom2 : str) -> dict:
        #输入两个谓词，返回对第一个谓词的变量的替换
        substitution = {}
        terms1, terms2 = Resolution.__TermExtract(atom1), Resolution.__TermExtract(atom2)
        for i in range(0, len(terms1)): #找到第一个不匹配项
            if (terms1[i] != terms2[i]):
                if (Resolution.__isVariable(terms1[i]) and not Resolution.__isVariable(terms2[i])): #terms1[i
                    substitution[terms1[i]] = terms2[i] #在代换集中添加代换terms1[i]->terms2[i]
                    atom1 = atom1.replace(terms1[i], terms2[i]) #将atom1中的变量都换为terms2[i]
                    terms1[i] = terms2[i]
                #如果都是常量或是变量则跳过
        return substitution #如果没有合法的替换，返回空字典

    @staticmethod
    def __Unify(clause1 : tuple[str], clause2 : tuple[str], collision : tuple[int]) -> list:
```

```
        #输入两个子句，返回两个子句的归结子句与谓词变量替换
        if (Resolution.__isContradict(clause1[collision[0]], clause2[collision[1]])):
            returner = set()
            for i in range(0, len(clause1)):
                if (i != collision[0]): returner.add(clause1[i]) #向returner中添加子句1的无冲突谓词
            for i in range(0, len(clause2)):
                if (i != collision[1]): returner.add(clause2[i]) #向returner中添加子句2的无冲突谓词
            return [tuple(returner), {}]
        substitution = Resolution.__MGU(clause1[collision[0]], clause2[collision[1]]) #寻找子句冲突谓词的变量
        if (substitution == {}): return [] #如果没有合法替换，返回空列表
        atoms1 = list(clause1)
        for i in range(0, len(atoms1)):
            for var, const in substitution.items(): atoms1[i] = atoms1[i].replace(var, const) #将子句1的可换变
        returner = set()
        for i in range(0, len(atoms1)):
            if (i != collision[0]): returner.add(atoms1[i]) #向returner中添加子句1的无冲突谓词
        for i in range(0, len(clause2)):
            if (i != collision[1]): returner.add(clause2[i]) #向returner中添加子句2的无冲突谓词
        return [tuple(returner), substitution]

    @staticmethod
    def __SingleStepResolution(clause_list : list[tuple[str]], index1 : int, index2 : int, collision : tuple[
        #单步归结，对可归结子句对输出生成报告，归结失败输出"F"
        clause1, clause2 = clause_list[index1], clause_list[index2] #存在矛盾谓词的一对子句
        if (len(clause2) > 1): return "F"
        unified_clause = Resolution.__Unify(clause1, clause2, collision) #由这对子句导出的归结子句及其变量代换
        if (unified_clause == [] or unified_clause[0] in clause_list): return "F" #寻找替换失败或归结子句已产生
        clause_list.append(unified_clause[0]) #向子句集添加新的归结子句
        string = "R[" + str(index1 + 1) #生成归结子句生成报告
        if (len(clause1) != 1): string = string + chr(collision[0] + 97)
        string = string + ',' + str(index2 + 1)
        if (len(clause2) != 1): string = string + chr(collision[1] + 97)
        string = string + ']'
        if (unified_clause[1] != {}): #如果有变量替换
            string = string + '{'
            for var, const in unified_clause[1].items(): string = string + str(var) + '=' + str(const) + ','
            string = string[:-1] + '}'
        string = string + str(unified_clause[0]) #结束生成归结子句生成报告
        return string
```

# 4.创新优化

本次实验中，我采取的是尽量将问题分解的策略。通过逐步分解问题，使得问题更模块化，不会出现因为各种小问题缠绕耦合的问题。

在应用广搜的同时，为了剪枝，使用了单文字策略。在不使用的情况下，例题二的归结会产生百行之多。使用之后可以控制在几十行内。

但是使用单文字策略确实会带来一些问题，例如单文字策略本身的不完备性。这导致例题二会出现死循环的现象。为此，我将其改为“半”单文字策略，即将“至少一子句为单文字子句”改为“只要求第二子句为单文字子句”，并将集合$set$列表化。这样，可以在剪枝的同时，减少出现死循环的概率。

# 三、实验结果分析

## 1.实验结果展示

以下是在代码末尾添加的测试用程序段：

```
string1 = "KB = {(GradStudent(sue),),(~GradStudent(x),Student(x)),(~Student(x),HardWorker(x)),(~HardWorker(su
string2 = "KB = {(A(tony),),(A(mike),),(A(john),),(L(tony,rain),),(L(tony,snow),),(~A(x),S(x),C(x)),(~C(y),~l
string3 = "KB = {(On(tony,mike),),(On(mike,john),),(Green(tony),),(~Green(john),),(~On(xx,yy),~Green(xx),Gree
test = Resolution(string2)
output = test.ResolutionFOL()
for step in output: print(step)
```

以下是各例题的输出结果：

例题一：

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

例题二：

```
('~A(w)', '~C(w)', 'S(w)')
('L(z,snow)', '~S(z)')
('~C(y)', '~L(y,rain)')
('~L(tony,u)', '~L(mike,u)')
('L(tony,rain)',)
('~A(x)', 'S(x)', 'C(x)')
('A(mike)',)
('L(tony,v)', 'L(mike,v)')
('A(tony)',)
('A(john)',)
('L(tony,snow)',)
R[1a,7]{w=mike}('~C(mike)', 'S(mike)')
R[1a,9]{w=tony}('~C(tony)', 'S(tony)')
R[1a,10]{w=john}('S(john)', '~C(john)')
R[3b,5]{y=tony}('~C(tony)',)
R[4a,5]{u=rain}('~L(mike,rain)',)
R[4a,11]{u=snow}('~L(mike,snow)',)
R[6a,7]{x=mike}('C(mike)', 'S(mike)')
R[6a,9]{x=tony}('S(tony)', 'C(tony)')
R[6a,10]{x=john}('S(john)', 'C(john)')
R[6c,15]{x=tony}('S(tony)', '~A(tony)')
R[8a,16]{v=rain}('L(mike,rain)',)
R[8a,17]{v=snow}('L(mike,snow)',)
R[16,22]()
```

例题三:

```
('~On(xx,yy)', '~Green(xx)', 'Green(yy)')
('~Green(john)',)
('Green(tony)',)
('On(mike,john)',)
('On(tony,mike)',)
R[1c,2]{yy=john}('~On(xx,john)', '~Green(xx)')
R[1b,3]{xx=tony}('Green(yy)', '~On(tony,yy)')
R[1a,4]{xx=mike,yy=john}('Green(john)', '~Green(mike)')
R[1a,5]{xx=tony,yy=mike}('~Green(tony)', 'Green(mike)')
R[6b,3]{xx=tony}('~On(tony,john)',)
R[6a,4]{xx=mike}('~Green(mike)',)
R[6a,5]{xx=tony}('~Green(tony)',)
R[7b,4]{yy=john}('Green(john)',)
R[7b,5]{yy=mike}('Green(mike)',)
R[7a,11]{yy=mike}('~On(tony,mike)',)
R[7a,12]{yy=tony}('~On(tony,tony)',)
R[11,14]()
```

```
PS F:\资料及作业\作业> python -u "f:\资料及作业\作业\人工智能\Python Project\第三次实验.py"
1 ('Green(tony)',)
2 ('On(tony,mike)',)
3 ('~On(xx,yy)', '~Green(xx)', 'Green(yy)')
4 ('~Green(john)',)
5 ('On(mike,john)',)
6 R[3b,1]{xx=tony}('Green(yy)', '~On(tony,yy)')
7 R[3a,2]{xx=tony,yy=mike}('Green(mike)', '~Green(tony)')
8 R[3c,4]{yy=john}('~On(xx,john)', '~Green(xx)')
9 R[3a,5]{xx=mike,yy=john}('~Green(mike)', 'Green(john)')
10 R[6b,2]{yy=mike}('Green(mike)',)
11 R[6a,4]{yy=john}('~On(tony,john)',)
12 R[6b,5]{yy=john}('Green(john)',)
13 R[8a,2]{xx=tony}('~Green(tony)',)
14 R[8a,5]{xx=mike}('~Green(mike)',)
15 R[8b,10]{xx=mike}('~On(mike,john)',)
16 R[8b,12]{xx=john}('~On(john,john)',)
17 R[10,14]()
```

## 2.评测指标展示分析

由于使用的是广度优先搜索策略，对先前生成的子句、和后来生成的子句都必须遍历一次，这会对时间产生极大浪费。但通过判定生成的子句是否已在子句集中，这通过增加时间减少了空间的浪费。对于例题这样小型的问题，时间复杂度的影响暂时不深刻，但当问题规模更为宏大的时候，可能会产生不良的结果。

# 四、思考题

无

# 五、参考资料

无