# 《人工智能》实验一报告

|  |  |  |  |
| --- | --- | --- | --- |
| 姓名 | 孙潇桐 | 学号 | 2021117405 |
| 实验地点 | 院楼321机房 | 实验日期 | 2023-10-16 |

1. **实验内容**

Implement unification algorithms in Prolog or Lisp or Java or C++.

**实验要求：**

**1.内容完备：方法简介、实验设置、实验结果及分析、代码；**

**2.格式规范、排版美观**

1. **实验原理**

根据书上的伪代码进行编码。

为了方便，我假定输入是列表语法的谓词表达式。类似于：

1. (foo x a (goo y))  
    (foo w a (goo jack))
2. (parents X (father X)(mother bill))  
    (parents bill (father bill) Y)

在读取字符串之后将字符串转换为表的形式，为了达到这个目的我设计了这样一个数据结构：

struct list {  
 string elem = {};  
 vector<list> sublist = {};  
 enum {  
 *ELEM*, *LIST* } type = {};

*//省略了一些成员函数*  
}

转换之后就能够按照书上给出的算法进行递归计算，关键点在于**unify**函数，该函数用于判断两个逻辑表达式是否可以统一，并返回替换规则列表。下面是大概的过程：

首先检查两个表达式的类型和内容，如果两个表达式都是空列表或相同的元素，则可以统一，返回空的替换规则列表。

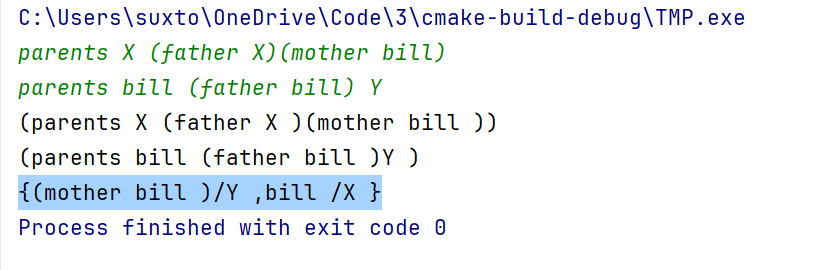
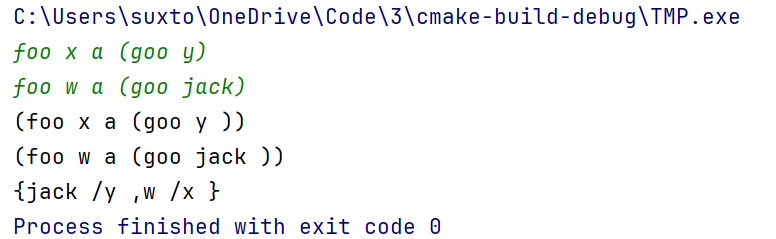
如果一个表达式是空列表，而另一个表达式是元素，说明无法统一，返回一个包含失败标志的替换规则。

如果一个表达式是元素，而另一个表达式是列表，检查元素是否在列表中出现，如果出现则无法统一，返回一个包含失败标志的替换规则，否则返回一个将元素替换为列表的替换规则。

如果两个表达式都是列表，则递归处理列表的第一个子表达式，并将得到的替换规则应用到剩余的表达式上，然后将两部分的替换规则合并返回。

完整代码如下：

#include<iostream>  
#include<vector>  
#include<algorithm>  
  
#define **FAIL** {{"", "", true}}  
using namespace std;  
  
struct rep {  
 string to, from;  
 bool fail = false;  
  
 bool operator==(rep &that) const {  
 return this->to == that.to && this->from == that.from;  
 }  
  
 bool operator<(const rep &that) const {  
 return this->to < that.to;  
 }  
};  
  
ostream &operator<<(ostream &o, rep &r) {  
 o << r.to << '/' << r.from;  
 return o;  
}  
  
struct list {  
 string elem = {};  
 vector<list> sublist = {};  
 enum {  
 *ELEM*, *LIST* } type = {};  
  
  
 bool contains(string &str) const {  
 if (type == *ELEM*) {  
 return elem == str;  
 } else {  
 bool res = false;  
 for (auto &e: sublist) {  
 if (e.type == *ELEM*)  
 res = res | (str == e.elem);  
 }  
 return res;  
 }  
 }  
  
 string to\_string() {  
 string s;  
 if (this->type == list::*LIST*) {  
 s.push\_back('(');  
 for (auto &ll: this->sublist) s.append(ll.to\_string());  
 s.push\_back(')');  
 } else {  
 s.append(this->elem + ' ');  
 }  
 return s;  
 }  
  
 void apply(vector<rep> const &r) {  
 if (type == *ELEM*) {  
 for (auto &rep: r) {  
 if (elem == rep.from) {  
 elem = rep.to;  
 break;  
 }  
 }  
 } else {  
 for (auto &l: sublist) {  
 l.apply(r);  
 }  
 }  
 }  
  
};  
  
ostream &operator<<(ostream &o, list const &l) {  
 if (l.type == list::*LIST*) {  
 o << '(';  
 for (auto &ll: l.sublist) o << ll;  
 o << ')';  
 } else {  
 o << l.elem << ' ';  
 }  
 return o;  
}  
  
ostream &operator<<(ostream &o, vector<rep> &vec) {  
 o << '{';  
 if (!vec.empty()) {  
 for (int i = 0; i < vec.size() - 1; i++) {  
 o << vec[i] << ',';  
 }  
 o << vec.back() << '}';  
 } else o << '}';  
 return o;  
}  
  
list conv(const string &input, int &pos) {  
 list result = {"", {}, list::*LIST*};  
 while (pos < input.size()) {  
 if (input[pos] == '(') {  
 ++pos;  
 result.sublist.push\_back(conv(input, pos));  
 } else if (input[pos] == ')') {  
 ++pos;  
 break;  
 } else if (input[pos] == ' ') {  
 ++pos;  
 } else {  
 string elem;  
 while (pos < input.size() && input[pos] != '(' && input[pos] != ')' && input[pos] != ' ') {  
 elem += input[pos];  
 ++pos;  
 }  
 result.sublist.emplace\_back(list{elem, {}, list::*ELEM*});  
 }  
 }  
 return result;  
}  
  
vector<rep> unify(list e1, list e2) {  
 if (e1.type == list::*LIST* && e2.type == list::*LIST* && e1.sublist.empty() && e2.sublist.empty()) {  
 return {};  
 }  
 if (e1.type == list::*ELEM* && e2.type == list::*ELEM* && e1.elem == e2.elem) return {};  
 if (e1.type == list::*ELEM*) {  
 if (e2.contains(e1.elem)) {  
 return **FAIL**;  
 } else {  
 return **{**{e2.to\_string(), e1.to\_string(), false}**}**;  
 }  
 }  
 if (e2.type == list::*ELEM*) {  
 if (e1.contains(e2.elem)) {  
 return **FAIL**;  
 } else {  
 return **{**{e1.to\_string(), e2.to\_string(), false}**}**;  
 }  
 }  
 *//both list* if (e1.sublist.empty() || e2.sublist.empty()) return **FAIL**;  
 list he1 = e1.sublist.front();  
 list he2 = e2.sublist.front();  
 auto subs1 = unify(he1, he2);  
 if (subs1.size() == 1 && subs1.front().fail) return **FAIL**;  
*// list te1 = e1, te2 = e2;  
// cout << e1 << '\n' << e2 << '\n'<<"-\n";* e1.sublist.erase(e1.sublist.begin());  
 e2.sublist.erase(e2.sublist.begin());  
 e1.apply(subs1);  
 e2.apply(subs1);  
*// cout << e1 << '\n' << e2 << '\n'<<"--\n";* auto subs2 = unify(e1, e2);  
 if (subs2.size() == 1 && subs2.front().fail) return **FAIL**;  
*// cout << subs1 << '\n' << subs2 << "\n--------------\n";* subs1.insert(subs1.end(), subs2.begin(), subs2.end());  
 return subs1;  
}  
  
  
int main() {  
 string s1, s2;  
 getline(cin, s1);  
 getline(cin, s2);  
 int pos = 0;  
 list e1 = conv(s1, pos);  
 pos = 0;  
 list e2 = conv(s2, pos);  
 cout << e1 << '\n' << e2 << '\n';  
 auto ans = unify(e1, e2);  
 sort(ans.begin(), ans.end());  
 ans.erase(unique(ans.begin(), ans.end()), ans.end());  
 cout << ans;  
}  
*/\*  
foo x a (goo y)  
foo w a (goo jack)  
  
parents X (father X)(mother bill)  
parents bill (father bill) Y  
\*/*

1. **实验过程以及结果分析**
2. **首先对代码进行编写，代码在上面贴出来了**
3. **对结果进行测试：**
   1. **测试一**
   2. ****
   3. **测试二**
   4. ****
4. **实验总结**

本次实验旨在实现归一化算法，我选择了在C++中进行编码。实验过程中，根据书上提供的伪代码，我成功地编写了最关键的unify函数和相关数据结构的程序代码。

在测试阶段，我使用了两组不同的逻辑表达式进行验证。根据实验结果分析，经过测试，实验代码能够正确地判断输出逻辑表达式的合一式，达到了预期的目标。

通过本次实验，我巩固了对统一算法的理解，在C++中实现算法的能力。这次实验对我进一步掌握人工智能领域中的统一算法起到了积极的促进作用。