



编译原理期末大作业

正则表达式等价性

数据科学与计算机学院 17大数据与人工智能

17341015 陈鸿峥

目录

1 实验目的	2
2 算法原理	2
2.1 字符串预处理	2
2.1.1 加法符号替代	2
2.1.2 连接符号插入	3
2.2 中缀转后缀表示	3
2.3 正则表达式转NFA	4
2.4 NFA转DFA	5
2.5 DFA最小化	6
2.6 DFA等价性判断	7
3 实验结果	8
A 完整源码	9
B 测试数据	24

一、实验目的

对于两个正则表达式 r 和 s ，判断这两个正则表达式的关系。正则表达式 r 和 s 的关系有4种：

1. r 和 s 等价，即 r 描述的语言和 s 描述的语言相等；
2. r 描述的语言是 s 描述的语言的真子集；
3. s 描述的语言是 r 描述的语言的真子集；
4. 非上述情况。

正则表达式的字符集为小写字母a-z，|号表示或者，*号表示闭包，?表示出现0或1次，+表示至少出现一次，大写字母E表示epsilon（空串）。

编写一个C++程序，实现上述功能。

输入格式：

第一行是测试数的组数 T 。接下来的 T 行，每行是两个正则表达式 r 和 s ，每个正则表达式只含a-z，|，*，?，+，（，），E。两个正则表达式之间用空格分开。

输出格式：

输出有 T 行。对于每组数据，如果 r 和 s 等价，输出=；如果 r 是 s 的真子集，输出<；如果 s 是 r 的真子集，输出>；非上述情况，输出!。

提交内容：

1. 能在Linux下或在Windows的Dev C++下编译运行的C++源程序；
2. 实验报告，包括算法描述，和你的测试用例，及测试结果。

二、算法原理

本次实验主要分为以下几个部分（如图1所示），正则表达式读入、字符串预处理、中缀转后缀表示、正则表达式转NFA、NFA转DFA、DFA最小化、DFA等价性判断，下文会依次阐述算法细节。



图 1: 算法核心流程

1. 字符串预处理

最开始读入的正则表达式并不方便后续的操作，因此需要对其进行一些预处理。这里包括加法符号替代和连接符号插入两项预处理。

(i) 加法符号替代

加法符号+用来表示前文的符号出现至少一次，这并非最简正则所支持的语法，因此我们

需要对其进行替代。注意到

$$R^+ \iff RR^*$$

因此可以将 R^+ 替换成用星号表示的形式。

需要注意 R 也是一个正则表达式，故需要确定其包含的内容，然后才可以做拷贝替换。这里我直接在输入的正则字符串上进行操作：

- 若 R 就是单一字母（ $a-z$ ），那么直接在其后面添加 R^* 。
- 若 R 为括号包围的正则表达式，则需要将括号包围的内容整份进行拷贝。具体实现可用栈维护括号的位置，遇到左括号则将其下标进栈，遇到右括号弹出栈顶的下标；如果右括号的下一符号为 $+$ ，则将当前下标与栈顶下标这一区间的字符串进行拷贝，附加在当前字符的右侧，并添加 $*$ 作结。

一个例子如下

$$((x|y)^+)^+ \text{ 展开后为 } ((x|y)(x|y)^*)((x|y)(x|y)^*)^*$$

只有正确使用栈操作才能够正确处理上述**嵌套括号**的情况。

完整实现见源码的`substitute_plus`函数。

(ii) 连接符号插入

接下来一步则是对连接符进行插入。由于原有的正则表示式都是默认省略连接符，这会给后续的分析转换带来麻烦，因此这一步则是将省略的这些连接符进行插入，这里用`.`代表连接符。

由于只需判断当前字符与下一字符的关系，这里只给出一个例子以示说明，完整实现请见`insert_concat`函数。

$$b^*a^*b^*a^* \text{ 变为 } b^*.a^*.b^*.a^*$$

2. 中缀转后缀表示

处理完加号和连接符后，即可将中缀的正则表达式转换为后缀形式表达，这将方便后续NFA转换的处理。

算法流程如下，利用一个算子栈进行状态维护：

1. 从左到右遍历中缀表达式
2. 如果当前字符是
 - 字母（ $a-z$ 或 E ），则将其添加入输出中
 - 左括号 $($ ，将其推入栈中

- 右括号)，将栈顶的符号依次弹出，直至遇见左括号（左括号也要弹出）
 - 算子，先将栈顶优先级比该算子高的符号依次弹出，再将当前算子推入栈中
3. 循环以上过程，直至字符串遍历完
 4. 将栈顶剩余的符号全部弹出

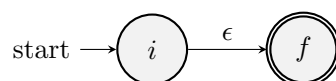
完整代码请见infix2postfix函数。

3. 正则表达式转NFA

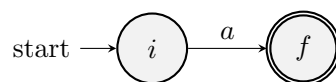
正则表达式转NFA即课本3.7.4节所述的Thompson算法，按照以下方式可以进行构造。

1. 奠基

- 对于表达式 ϵ （即输入为E），构建NFA

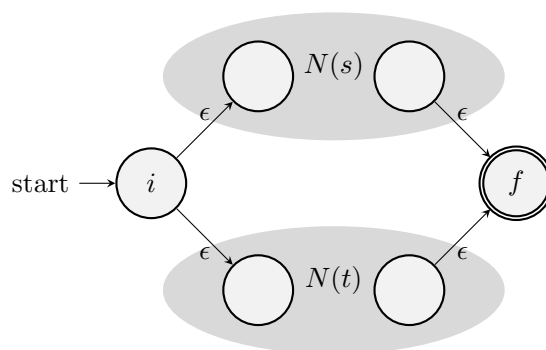


- 对于任意子表达式 $a \in \Sigma$ ，构建NFA

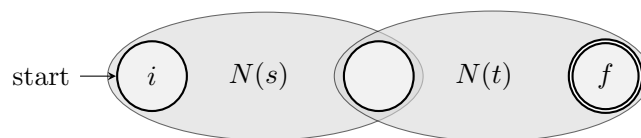


2. 推论

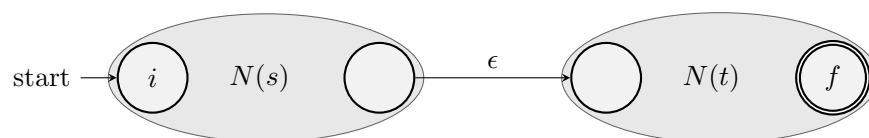
- $r = s|t$ ，取并集



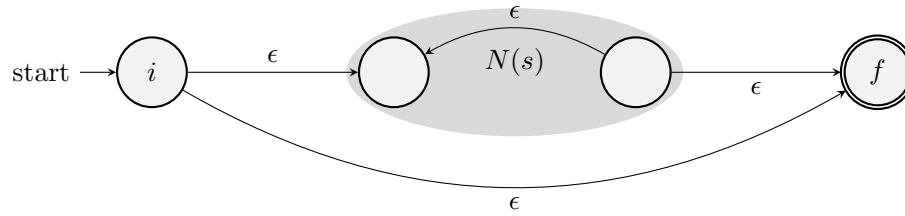
- $r = st$ ，取连接



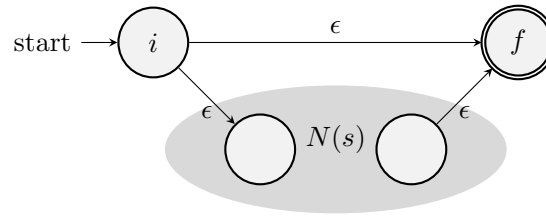
这里为方便程序实现，采用了如下的构造方式。



- $r = s^*$, Kleene闭包



- $r = s?$, 这是本问题新加的语法, 代表0个或1个输入, 相当于 $r = \epsilon|s$, 故只需对并集自动机的构造进行适当修改即可。



具体实现上需要创建一个NFA结点类, 存储其结点编号、状态、出边, 定义如下。

```

1 class NFA_Node{
2 public:
3     NFA_Node() : id(cnt), accepting(false) {
4         cnt++;
5     }
6     int id;
7     static int cnt;
8     bool accepting;
9     map<char,int> out;
10    vector<int> e;
11 };
  
```

并用一个栈对NFA结点进行维护, 每次从后缀表达式中读取一个符号, 从栈顶弹出对应 $N(s)$ 和 $N(t)$ 的入口结点及出口结点, 并按照以上构造方式, 创建新的 i 和 f 结点, 连接好对应边后, 将 i 和 f 推入栈顶。

完整代码见`regex2nfa`函数。

4. NFA转DFA

NFA转DFA的算法参见课本第3.7.1节, 即子集构造算法, 这里需要以下几个操作。关于集合的操作均采用C++的`<set>`标准库进行实现, 而计算闭包则是通过递归深度优先搜索(DFS)实现。

操作	描述
$\epsilon\text{-closure}(s)$	从状态 s 能够通过 ϵ 边转换的集合（包括自己的状态）
$\epsilon\text{-closure}(T)$	$\bigcup_{s \in T} \epsilon\text{-closure}(s)$
$\text{move}(T, a)$	从 $s \in T$ 中的状态通过输入符号 a 进行转换

同时，构造DFA结点类，存储其结点编号、状态及出边，定义如下。

```

1 class DFA_Node{
2 public:
3     DFA_Node() : id(cnt), start(false), accepting(false), group(1) {
4         cnt++;
5     }
6     int id;
7     static int cnt;
8     bool start;
9     bool accepting;
10    map<char,int> out; // be careful of non-existed keys
11    int group;
12 };

```

注意到DFA中就没有 ϵ 边了。

具体算法参见图2，其中状态的访问用队列`queue`维护，状态标记用集合`set`维护，状态转移用映射`map`维护。

```

initially,  $\epsilon\text{-closure}(s_0)$  is the only state in  $Dstates$ , and it is unmarked;
while ( there is an unmarked state  $T$  in  $Dstates$  ) {
    mark  $T$ ;
    for ( each input symbol  $a$  ) {
         $U = \epsilon\text{-closure}(\text{move}(T, a))$ ;
        if (  $U$  is not in  $Dstates$  )
            add  $U$  as an unmarked state to  $Dstates$ ;
         $Dtran[T, a] = U$ ;
    }
}

```

图 2: 子集构造算法

完整代码见`nfa2dfa`函数，需要注意在计算闭包构造新结点的同时也要记录开始状态和接受状态。

5. DFA最小化

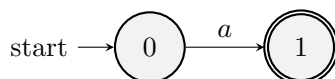
由于每个正则表达式对应的最小化DFA是唯一的，因此可以通过最小化DFA的方式来判断

两个正则表达式之间的关系，这里采用课本3.9.6节的Hopcroft划分算法。这一部分算是本次实验所有算法中最难实现的一个，需要考虑到非常多的细节。

算法流程如下：

1. 开始初始划分为 Π ，所有接受状态 F 为一个组，该集合的补为另一个组 $S - F$ （注意有可能所有状态均为接受状态，那么 $S - F = \emptyset$ ），将这两个组推入队列中（在具体实现上我并没有将所有划分用统一的数据结构进行存储，而是在原有每个DFA结点下添加group成员进行标记）
2. 若组队列非空，则弹出头部的组
 - 若该组只有一个状态，则不需划分
 - 若改组有多个状态，并且存在两个状态对于相同符号的出边落在不同的组，那么这两个状态需要被划分为不同的组（具体实现上采用`map<int, vector<int>>`对状态进行维护，考察每个DFA结点对相同符号的出边，目标结点的组标号为`map`的key，而value则是组内的状态编号），将划分后的新组重新加入队列
3. 对组编号进行重新分配及映射，为每个组挑选旧DFA中的标志结点，并创建新的DFA结点，将新的结点按照旧的连边关系建图，生成最终最小化DFA

完整代码见`minimize_dfa`函数，这里在划分组时需要小心一些出边并不完全的DFA。一个简单的例子即a，生成的DFA如下图所示。



该DFA的接受结点即没有任何的出边，这在访问时需要特别小心。比如在分组时就会将这些访问不到出符号的结点的出结点的组别设为-1。

在具体实现上`map`不能直接通过`[]`进行读取，哪怕没有该元素，也会读出值造成错误；故应该用`.count(c)`判断c元素是否存在映射中，并用`.at(c)`来读出c的内容。

6. DFA等价性判断

最后一步则是进行DFA等价性的判断。注意到每个正则表达式都可以对应一个DFA，而两个DFA R 和 S 之间的关系只会有四种情况，可以考察原DFA与补DFA的交关系来判断原始两个DFA之间的包含关系，如图3所示。

- R 等价于 S : $R \cap \bar{S} = \emptyset \wedge \bar{R} \cap S = \emptyset$
- R 含于 S : $R \cap \bar{S} = \emptyset \wedge \bar{R} \cap S \neq \emptyset$
- S 含于 R : $R \cap \bar{S} \neq \emptyset \wedge \bar{R} \cap S = \emptyset$
- 其他关系: $R \cap \bar{S} \neq \emptyset \wedge \bar{R} \cap S \neq \emptyset$

其中这里 \bar{R} 代表DFA的补，即将所有接受状态变为非接受状态，非接受状态变为接受状态。



图 3: DFA关系的四种情况

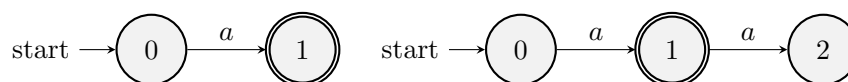
所以只需求两个DFA是否**相交**即可，具体方法是对两个DFA同时从开始状态进行模拟。如果模拟到两个状态 $s1$ 和 $s2$ 都为接受态，意味着有一个字符串可以同时被两个DFA/正则表达式所匹配，即两个DFA相交；如果对于所有字符串，都没有办法使两个DFA同时到达接受态，则这两个DFA不相交。

具体实现采用递归的DFS。为防止陷入死循环，开设一个visited二维数组记录两个DFA访问过的状态。算法流程如下，函数为`intersection(s1,s2,dfa1,dfa2,visited)`。

1. 设置状态 $(s1, s2)$ 为已访问
2. 若 $(s1, s2)$ 均为接受态，返回真
3. 对于每一个输入符号
 - 求两个状态机在 $(s1, s2)$ 下的状态转移，新的状态为 $(o1, o2)$
 - 若 $(o1, o2)$ 未访问且`intersection(o1,o2,dfa1,dfa2,visited)`为真，返回真
4. 若所有输入符号都没返回真，则返回假

有了求交的算法就可以通过执行两次交判断语句来得到两个正则表达式的关系，即 R 与 \bar{S} 求交， S 与 \bar{R} 求交。因此还需要有一个辅助函数来对DFA进行求反。由于之前已经创建了DFA_Node的类，故求反只需将每个DFA的结点由接受态改为非接受态，非接受态改为接受态。

这里还需注意对于DFA的某些状态，可能没有对所有输入符号都有出边，这在模拟DFA时可能会出现错误。对于这种情况，可以假设有一个冗余结点来接收这些非法输入。如对于a的DFA来说，右侧结点已经无法再接收其他符号，那么为保持每个结点输入符号都存在，可以添加多一个非接受态结点用以作为输出（即黑洞结点，只进不出），如下右图所示。

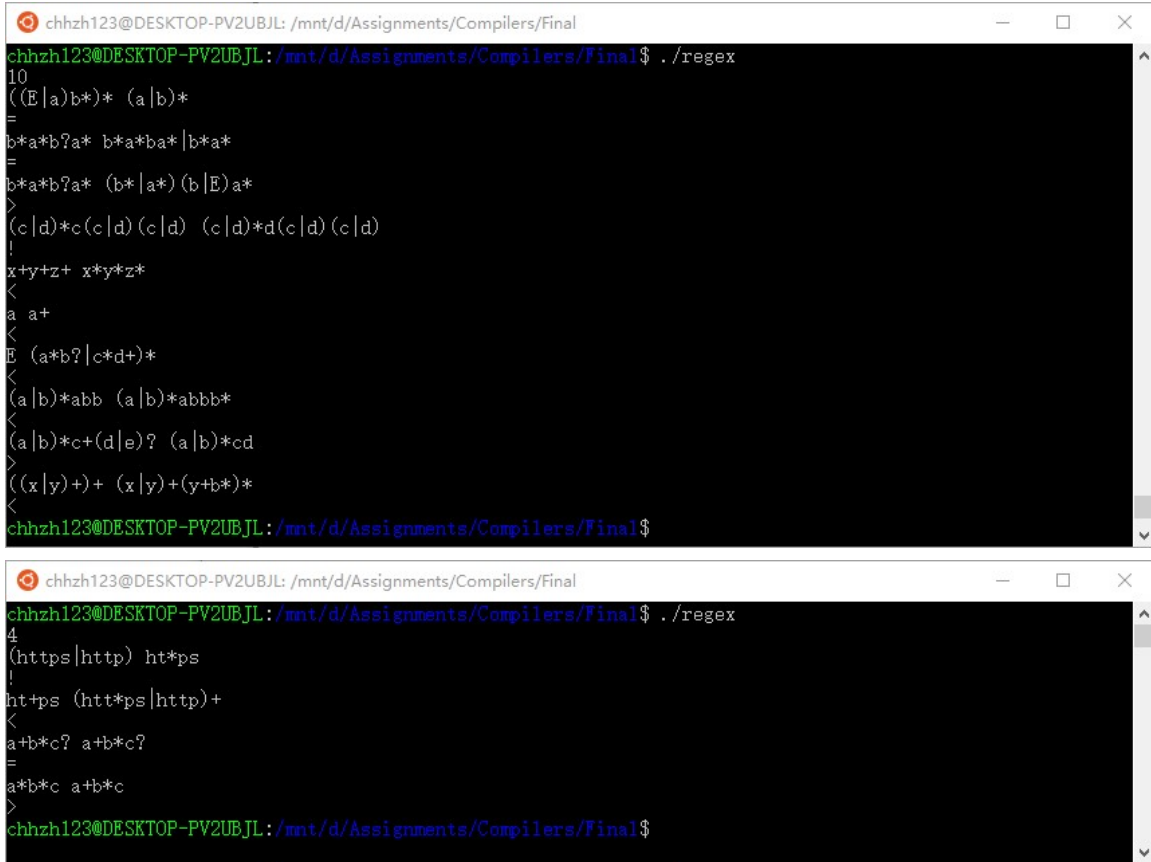


完整代码可见judge函数。

三、实验结果

本实验代码已提交至Sicily，并测试通过全部样例。

所有测试样例请见附录B，具体实验结果如图4所示。



```

chhzh123@DESKTOP-PV2UBJL: /mnt/d/Assignments/Compilers/Final$ ./regex
10
((E|a)b*)* (a|b)*
=
b*a*b?a* b*a*ba*|b*a*
=
b*a*b?a* (b*|a*)(b|E)a*
>
(c|d)*c(c|d)(c|d) (c|d)*d(c|d)(c|d)
!
x+y+z+ x*y*z*
<
a a+
<
E (a*b*|c*d+)*
<
(a|b)*abb (a|b)*abbb*
<
(a|b)*c+(d|e)? (a|b)*cd
>
((x|y)+) (x|y)+(y+b)*
<
chhzh123@DESKTOP-PV2UBJL: /mnt/d/Assignments/Compilers/Final$

chhzh123@DESKTOP-PV2UBJL: /mnt/d/Assignments/Compilers/Final$ ./regex
4
(https|http) ht*ps
!
ht*ps (htt*ps|http)+
<
a+b*c? a+b*c?
=
a*b*c a+b*c
>
chhzh123@DESKTOP-PV2UBJL: /mnt/d/Assignments/Compilers/Final$

```

图 4: 实验结果

附录 A. 完整源码

源码分为三个文件，main.cpp为核心算法文件，utils.h内含调试输出设施，Makefile则提供了自动化编译的脚本。

main.cpp

```

1  #include <iostream>
2  #include <string>
3  #include <cstring>
4  #include <stack>
5  #include <map>
6  #include <vector>
7  #include <queue>
8  #include <set>
9  #include <utility>
10 // #include "utils.h"
11 using namespace std;
12
13 class NFA_Node{
14 public:
15     NFA_Node() : id(cnt), accepting(false) {

```

```

16         cnt++;
17     }
18     int id;
19     static int cnt;
20     bool accepting;
21     map<char,int> out;
22     vector<int> e;
23 };
24
25 class DFA_Node{
26 public:
27     DFA_Node() : id(cnt), start(false), accepting(false), group(1) {
28         cnt++;
29     }
30     int id;
31     static int cnt;
32     bool start;
33     bool accepting;
34     map<char,int> out; // be careful of non-existed keys
35     int group;
36 };
37
38 void print_nfa(const vector<NFA_Node*>& nfa) {
39     for (auto state : nfa) {
40         cout << state->id;
41         if (state->accepting)
42             cout << "(A)";
43         cout << ": ";
44         for (auto edge : state->e)
45             cout << edge << " ";
46         for (auto& c : state->out)
47             cout << c.first << "(" << c.second << ")";
48         cout << endl;
49     }
50 }
51
52 void print_dfa(const vector<DFA_Node*>& dfa, const set<char>& input_symbol) {
53     for (auto c : input_symbol)
54         cout << "\t" << c;
55     cout << endl;
56     for (auto node : dfa) {
57         // cout << (char)(node->id+'A');
58         cout << node->id;
59         if (node->start)
60             cout << "S";
61         if (node->accepting)
62             cout << "*";
63         cout << "\t";
64         for (auto c : input_symbol)

```

```

65         // cout << (char)(node->out[c]+'A') << "\t";
66         if (node->out.count(c) == 0)
67             cout << (-1) << "\t";
68         else
69             cout << node->out.at(c) << "\t";
70         cout << endl;
71     }
72 }
73
74 template<typename T>
75 void free_node(vector<T*> v) {
76     for (auto it = v.begin(); it != v.end(); ++it)
77         delete *it;
78     v.clear();
79 }
80
81 int prec(char c) {
82     switch (c) {
83         case '(':
84             case ')': return 3;
85         case '*':
86             case '?':
87             case '+': return 2;
88             case '.': return 1;
89             case '|': return 0;
90         default: return -1;
91     }
92 }
93
94 /*
95  * R+ is equivalent to RR* (one or more)
96  * Since NFA engine cannot recognize +,
97  * substitute + using * first.
98  */
99 string substitute_plus(const string str) {
100     if (str.find("+") == string::npos)
101         return str;
102     string res = "";
103     stack<int> out_stk;
104     int len = str.size();
105     // do parentheses matching & make duplication
106     for (int i = 0; i < len; ++i) {
107         if (str[i] == '(') {
108             res += "(";
109             out_stk.push(res.size() - 1);
110         } else if (str[i] == ')') {
111             if (i + 1 < len && str[i + 1] == '+') {
112                 int front = out_stk.top();
113                 res += "(";

```

```

114         string new_str = res.substr(front, res.size() - front);
115         res += new_str + "*";
116     } else {
117         res += ")";
118     }
119     out_stk.pop();
120 } else if (str[i] == '+' && str[i-1] != ')') {
121     char c = str[i-1];
122     res += c;
123     res += "*";
124 } else if (str[i] != '+')
125     res += str[i];
126 }
127 return res;
128 }
129
130 /*
131  * To easier parse the regex and change it
132  * to postfix format, need to insert concatenation
133  * sign first. Here use "." to represent.
134  */
135 string insert_concat(const string str) {
136     string res = "";
137     int i = 0;
138     int len = str.size();
139     for (auto c1 : str) {
140         res += c1;
141         if (i + 1 < len) {
142             char c2 = str[i + 1];
143             if (c1 != '(' && c1 != '|' &&
144                 c2 != '*' && c2 != '?' && c2 != '+' &&
145                 c2 != '|' && c2 != ')')
146                 res += '.';
147         }
148         i++;
149     }
150     return res;
151 }
152
153 /*
154  * Change the regex to postfix format
155  * Use a stack to maintain operators
156  */
157 string infix2postfix(const string str) {
158     string res = "";
159     stack<char> op_stk;
160     for (auto c : str) {
161         if (isalpha(c)) // operand
162             res += c;

```

```

163     else if (c == '(')
164         op_stk.push('(');
165     else if (c == ')') {
166         char top = op_stk.top();
167         while (top != '(') {
168             res += top;
169             op_stk.pop();
170             top = op_stk.top();
171         }
172         op_stk.pop(); // discard '('
173     } else { // operator
174         while (!op_stk.empty() && op_stk.top() != '('
175             && prec(c) <= prec(op_stk.top())) {
176             char top = op_stk.top();
177             res += top;
178             op_stk.pop();
179         }
180         op_stk.push(c);
181     }
182 }
183 // pop all remaining ops in the stack
184 while (!op_stk.empty()) {
185     char top = op_stk.top();
186     res += top;
187     op_stk.pop();
188 }
189 return res;
190 }
191
192 string get_postfix(const string str) {
193     string res;
194     // cout << str << endl;
195     res = substitute_plus(str);
196     // cout << res << endl;
197     res = insert_concat(res);
198     // cout << res << endl;
199     res = infix2postfix(res);
200     // cout << res << endl;
201     return res;
202 }
203
204 set<char> get_input_symbol(const string str) {
205     set<char> input_symbol(str.begin(), str.end());
206     input_symbol.erase('E');
207     input_symbol.erase('.');
208     input_symbol.erase('*');
209     input_symbol.erase('?');
210     input_symbol.erase('+');
211     input_symbol.erase('|');

```

```

212     // print_set<char>(input_symbol);
213     return input_symbol;
214 }
215
216 /*
217  * 3.7.4 Construction of an NFA from a Regular Expression
218  * McNaughton-Yamada-Thompson algorithm
219  *
220  * Return:
221  * pair<NFA_Node*,NFA_Node*>: start & accepting state of nfa
222  * vector<NFA_Node*>: The built NFA
223  *
224  */
225 pair<int,int> regex2nfa(const string postfix_str,
226                        vector<NFA_Node*>& nfa) {
227     stack<NFA_Node*> nfa_stk;
228     for (auto c : postfix_str) {
229         if (isalpha(c)) { // a-z E
230             NFA_Node* begin = new NFA_Node();
231             NFA_Node* end = new NFA_Node();
232             nfa.push_back(begin);
233             nfa.push_back(end);
234             // cout << c << " " << begin->id << " " << end->id << endl;
235             if (c != 'E')
236                 begin->out[c] = end->id;
237             else
238                 begin->e.push_back(end->id);
239             end->accepting = true;
240             nfa_stk.push(begin);
241             nfa_stk.push(end);
242         } else if (c == '|' ) { // union
243             NFA_Node* begin = new NFA_Node();
244             NFA_Node* end = new NFA_Node();
245             nfa.push_back(begin);
246             nfa.push_back(end);
247             // N(t)
248             NFA_Node* d = nfa_stk.top(); nfa_stk.pop();
249             NFA_Node* c = nfa_stk.top(); nfa_stk.pop();
250             // N(s)
251             NFA_Node* b = nfa_stk.top(); nfa_stk.pop();
252             NFA_Node* a = nfa_stk.top(); nfa_stk.pop();
253             begin->e.push_back(a->id);
254             begin->e.push_back(c->id);
255             b->e.push_back(end->id);
256             d->e.push_back(end->id);
257             b->accepting = false;
258             d->accepting = false;
259             end->accepting = true;
260             nfa_stk.push(begin);

```

```

261     nfa_stk.push(end);
262 } else if (c == '.') { // concatenation
263     // N(t)
264     NFA_Node* d = nfa_stk.top(); nfa_stk.pop();
265     NFA_Node* c = nfa_stk.top(); nfa_stk.pop();
266     // N(s)
267     NFA_Node* b = nfa_stk.top(); nfa_stk.pop();
268     NFA_Node* a = nfa_stk.top(); nfa_stk.pop();
269     b->e.push_back(c->id);
270     b->accepting = false;
271     d->accepting = true;
272     nfa_stk.push(a);
273     nfa_stk.push(d);
274 } else if (c == '*') { // Kleen closure
275     NFA_Node* begin = new NFA_Node();
276     NFA_Node* end = new NFA_Node();
277     nfa.push_back(begin);
278     nfa.push_back(end);
279     NFA_Node* b = nfa_stk.top(); nfa_stk.pop();
280     NFA_Node* a = nfa_stk.top(); nfa_stk.pop();
281     b->e.push_back(a->id);
282     begin->e.push_back(end->id);
283     begin->e.push_back(a->id);
284     b->e.push_back(end->id);
285     b->accepting = false;
286     end->accepting = true;
287     nfa_stk.push(begin);
288     nfa_stk.push(end);
289 } else if (c == '?') { // zero or one, E|N(t)
290     NFA_Node* begin = new NFA_Node();
291     NFA_Node* end = new NFA_Node();
292     nfa.push_back(begin);
293     nfa.push_back(end);
294     NFA_Node* b = nfa_stk.top(); nfa_stk.pop();
295     NFA_Node* a = nfa_stk.top(); nfa_stk.pop();
296     begin->e.push_back(a->id);
297     begin->e.push_back(end->id);
298     b->e.push_back(end->id);
299     b->accepting = false;
300     end->accepting = true;
301     nfa_stk.push(begin);
302     nfa_stk.push(end);
303 } else if (c == '+') {
304     // preprocess in input string
305 }
306 }
307 // print_nfa(nfa);
308 NFA_Node* end = nfa_stk.top(); nfa_stk.pop();
309 NFA_Node* begin = nfa_stk.top(); nfa_stk.pop();

```

```

310     pair<int,int> res(begin->id,end->id);
311     return res;
312 }
313
314 // helper function for calculating e-closure of a state
315 void traverse_e(const NFA_Node* s, const vector<NFA_Node*>& nfa, set<int>& res) {
316     for (auto neigh : s->e) {
317         if (res.find(neigh) != res.end())
318             break;
319         res.insert(neigh);
320         traverse_e(nfa[neigh], nfa, res);
321     }
322 }
323
324 // e-closure of a state
325 set<int> epsilon_closure(const NFA_Node* s, const vector<NFA_Node*>& nfa) {
326     set<int> res;
327     res.insert(s->id);
328     traverse_e(s, nfa, res);
329     return res;
330 }
331
332 // e-closure of a set
333 set<int> epsilon_closure(const set<int>& T, const vector<NFA_Node*>& nfa) {
334     set<int> res;
335     for (auto s : T) {
336         set<int> tmp = epsilon_closure(nfa[s], nfa);
337         res.insert(tmp.begin(), tmp.end());
338     }
339     return res;
340 }
341
342 // helper function for calculating transition
343 void traverse(NFA_Node* s, const vector<NFA_Node*>& nfa,
344             const char a, set<int>& res) {
345     if (s->out.count(a) != 0)
346         res.insert(s->out[a]);
347 }
348
349 // transition of a set
350 set<int> move_to(const set<int>& T, const vector<NFA_Node*>& nfa, const char a) {
351     set<int> res;
352     for (auto s : T)
353         traverse(nfa[s], nfa, a, res);
354     return res;
355 }
356
357 /*
358 * 3.7.1 Conversion of an NFA to a DFA

```



```

359  *
360  * Return:
361  * int: start state id of the DFA (only one entrance)
362  * vector<DFA_Node*>: The built DFA
363  *
364  */
365  int nfa2dfa(const pair<int,int>& p,
366             const vector<NFA_Node*>& nfa,
367             const set<char>& input_symbol,
368             vector<DFA_Node*>& dfa) {
369      int start = p.first ;
370      int end = p.second;
371      queue<set<int>> q;
372      set<int> start_closure = epsilon_closure(nfa[start], nfa);
373      q.push(start_closure);
374      DFA_Node* node = new DFA_Node();
375      dfa.push_back(node);
376      set<set<int>> marked; // used to record visited states
377      // loop up table
378      // used to record mapping from NFA states to DFA
379      // set of NFA states -> DFA id
380      map<set<int>,int> lut;
381      lut[q.front()] = 0;
382      int start_id ;
383      if ( start_closure.count(end) != 0 ) {
384          node->accepting = true;
385          node->group = 0;
386      }
387      if ( start_closure.count(start) != 0 ) {
388          node->start = true;
389          start_id = dfa.size() - 1;
390      }
391      while (!q.empty()) {
392          set<int> s = q.front();
393          int idx = lut[s];
394          for (auto sym : input_symbol) { // only alphas
395              set<int> move = move_to(s,nfa,sym);
396              if (move.empty())
397                  continue;
398              set<int> U = epsilon_closure(move,nfa);
399              if (marked.find(U) == marked.end()) { // U not in Dstates
400                  q.push(U);
401                  marked.insert(U);
402                  DFA_Node* node = new DFA_Node();
403                  dfa.push_back(node);
404                  lut[U] = dfa.size() - 1;
405                  // record the start & accepting states
406                  if (U.count(end) != 0) {
407                      node->accepting = true;

```

```

408         node->group = 0;
409     }
410     if (U.count(start) != 0) {
411         node->start = true;
412         start_id = dfa.size() - 1;
413     }
414 }
415 dfa[idx]->out[sym] = lut[U];
416 }
417 q.pop();
418 }
419 return start_id;
420 }
421
422 /*
423  * 3.9.6 Minimizing the Number of States of a DFA
424  * Hopcroft's algorithm
425  *
426  * Return:
427  * int: start state id
428  * vector<DFA_Node*>: Minimum DFA
429  */
430 int minimize_dfa(vector<DFA_Node*>& dfa,
431                 const set<char>& input_symbol,
432                 vector<DFA_Node*>& min_dfa) {
433     queue<vector<int>>> partition;
434     vector<int> s1, s2;
435     set<int> group_id;
436     for (auto state : dfa) {
437         if (state->accepting)
438             s1.push_back(state->id);
439         else
440             s2.push_back(state->id);
441         group_id.insert(state->group);
442     }
443     int n_group = 2;
444     partition.push(s2);
445     partition.push(s1);
446     map<int,int> state_map;
447     // do partition
448     while (!partition.empty()) {
449         vector<int> p = partition.front();
450         partition.pop();
451         int size = p.size();
452         // only one state, need not to be partitioned
453         if (size < 2)
454             continue;
455         // more than 2 states
456         int origin_group = dfa[p[0]]->group;

```

```

457     for (auto c : input_symbol) {
458         map<int,vector<int>> groups; // gid, idx in group
459         for (int i = 0; i < size; ++i) {
460             if (dfa[p[i]]->out.count(c) != 0) {
461                 int out = dfa[p[i]]->out.at(c);
462                 groups[dfa[out]->group].push_back(p[i]);
463             } else {
464                 // be careful of this case!
465                 // no available transition in DFA!
466                 groups[-1].push_back(p[i]);
467             }
468         }
469         // if jump to different groups
470         // can be partitioned
471         int g_size = groups.size();
472         if (g_size >= 2) {
473             group_id.erase(origin_group);
474             for (auto& item : groups) {
475                 n_group++;
476                 group_id.insert(n_group);
477                 for (auto idx : item.second)
478                     dfa[idx]->group = n_group;
479                 partition.push(item.second);
480             }
481             break;
482         }
483     }
484 }
485 // be careful that start and end may overlap
486 vector<bool> start_flag(n_group,false);
487 vector<bool> end_flag(n_group,false);
488 map<int,int> group_map;
489 // reallocate group id
490 int cnt = 0;
491 for (auto id : group_id) {
492     group_map[id] = cnt;
493     cnt++;
494 }
495 // remap group id
496 int start_id;
497 int len = dfa.size();
498 for (int i = 0; i < len; ++i) {
499     dfa[i]->group = group_map[dfa[i]->group];
500     int group = dfa[i]->group;
501     state_map[group] = i;
502     if (dfa[i]->start) {
503         start_flag[group] = true;
504         start_id = group;
505     }

```

```

506     if (dfa[i]->accepting) {
507         end_flag[group] = true;
508     }
509 }
510 // generate new DFA
511 n_group = group_id.size();
512 for (int i = 0; i < n_group; ++i) {
513     DFA_Node* node = new DFA_Node();
514     for (auto c : input_symbol) {
515         if (dfa[state_map[i]]->out.count(c) != 0) {
516             int out = dfa[state_map[i]]->out.at(c);
517             node->out[c] = dfa[out]->group;
518         }
519     }
520     if (start_flag[i])
521         node->start = true;
522     if (end_flag[i])
523         node->accepting = true;
524     min_dfa.push_back(node);
525 }
526 return start_id;
527 }
528
529 int build_dfa(const string postfix_str ,
530              const set<char>& input_symbol,
531              vector<DFA_Node*> min_dfa) {
532     NFA_Node::cnt = 0;
533     vector<NFA_Node*> nfa;
534     pair<int,int> p = regex2nfa(postfix_str,nfa);
535
536     DFA_Node::cnt = 0;
537     vector<DFA_Node*> dfa;
538     int start_dfa = nfa2dfa(p,nfa,input_symbol,dfa);
539     free_node<NFA_Node*>(nfa);
540     // print_dfa(dfa,input_symbol);
541     // min_dfa = dfa;
542     // return start_dfa;
543
544     DFA_Node::cnt = 0;
545     int start_mindfa = minimize_dfa(dfa,input_symbol,min_dfa);
546     free_node<DFA_Node*>(dfa);
547     // print_dfa(min_dfa,input_symbol);
548     return start_mindfa;
549 }
550
551 bool intersection(const int s1, const int s2,
552                  const vector<DFA_Node*>& dfa1,
553                  const vector<DFA_Node*>& dfa2,
554                  const int len1, const int len2,

```

```

555         const set<char>& input_symbol,
556         vector<vector<bool>>& visited) {
557     visited[s1][s2] = true;
558     bool acc1 = (s1 == len1) ? false : dfa1[s1]->accepting;
559     bool acc2 = (s2 == len2) ? true : dfa2[s2]->accepting;
560     if (acc1 && acc2)
561         return true;
562     for (auto c : input_symbol) {
563         // be careful of unavailable states of DFA
564         int o1 = (s1 == len1 || dfa1[s1]->out.count(c) == 0) ?
565             len1 : dfa1[s1]->out.at(c);
566         int o2 = (s2 == len2 || dfa2[s2]->out.count(c) == 0) ?
567             len2 : dfa2[s2]->out.at(c);
568         if (!visited[o1][o2])
569             if (intersection(o1,o2,dfa1,dfa2,len1,len2,input_symbol,visited))
570                 return true;
571     }
572     return false;
573 }
574
575 void complement(vector<DFA_Node*>& dfa) {
576     for (auto node : dfa) {
577         node->accepting = !node->accepting;
578     }
579 }
580
581 int judge(vector<DFA_Node*>& dfa1,
582           vector<DFA_Node*>& dfa2,
583           const set<char>& symbol1,
584           const set<char>& symbol2,
585           const int s1, const int s2) {
586     // add a dummy state
587     int len1 = dfa1.size();
588     int len2 = dfa2.size();
589
590     vector<vector<bool>> visited;
591     for (int i = 0; i < len1+1; ++i) {
592         vector<bool> tmp(len2+1,false);
593         visited.push_back(tmp);
594     }
595     complement(dfa2);
596     bool a_in_cb = intersection(s1,s2,dfa1,dfa2,len1,len2,symbol1,visited);
597
598     visited.clear();
599     for (int i = 0; i < len2+1; ++i) {
600         vector<bool> tmp(len1+1,false);
601         visited.push_back(tmp);
602     }
603     complement(dfa2);

```

```

604 complement(dfa1);
605 bool b_in_ca = intersection(s2,s1,dfa2,dfa1,len2,len1,symbol2,visited);
606
607 if (!a_in_cb && !b_in_ca)
608     return 0; // dfa1 = dfa2
609 else if (!a_in_cb && b_in_ca)
610     return 1; // dfa1 in dfa2
611 else if (a_in_cb && !b_in_ca)
612     return 2; // dfa2 in dfa1
613 else
614     return 3; // none
615 }
616
617 int NFA_Node::cnt = 0;
618 int DFA_Node::cnt = 0;
619
620 #ifdef NO_STDIN
621 const vector<vector<string>> input_str = {
622     {"(E|a)b*", "(a|b)*"}, // =
623     {"b*a*b?a*", "b*a*ba*|b*a*"}, // =
624     {"b*a*b?a*", "(b*|a*)(b|E)a*"}, // >
625     {"(c|d)*c(c|d)(c|d)", "(c|d)*d(c|d)(c|d)"}, // !
626     {"x+y+z+", "x*y*z*"}, // <
627     {"a", "a+"}, // <
628     {"(a|b)*abb", "(a|b)*abbb*"}, // <
629     {"(a|b)*c+(d|e)?", "(a|b)*cd"}, // >
630     {"(x|y)+", "(x|y)+(y+b*)*"}, // <
631     {"(https|http)", "ht*ps"}, // !
632     {"ht+ps", "(htt*ps|http)+"}, // <
633     {"a+b*c?", "a+b*c?"}, // =
634     {"a*b*c", "a+b*c"}, // >
635 };
636 #endif
637
638 int main() {
639     int n_case;
640     #ifdef NO_STDIN
641         n_case = input_str.size();
642     #else
643         cin >> n_case;
644     #endif
645     for (int case_id = 0; case_id < n_case; ++case_id) {
646         string str1, str2;
647         #ifdef NO_STDIN
648             str1 = input_str[case_id][0];
649             str2 = input_str[case_id][1];
650         #else
651             cin >> str1 >> str2;
652         #endif

```

```

653     string postfix_str1 = get_postfix(str1);
654     string postfix_str2 = get_postfix(str2);
655     set<char> symbol1 = get_input_symbol(str1);
656     set<char> symbol2 = get_input_symbol(str2);
657     vector<DFA_Node*> dfa1;
658     int s1 = build_dfa(postfix_str1, symbol1, dfa1);
659     vector<DFA_Node*> dfa2;
660     int s2 = build_dfa(postfix_str2, symbol2, dfa2);
661     int res = judge(dfa1, dfa2, symbol1, symbol2, s1, s2);
662     if (res == 0)
663         cout << "=" << endl;
664     else if (res == 1)
665         cout << "<" << endl;
666     else if (res == 2)
667         cout << ">" << endl;
668     else
669         cout << "!" << endl;
670     free_node<DFA_Node>(dfa1);
671     free_node<DFA_Node>(dfa2);
672 }
673 return 0;
674 }

```

utils.h

```

1  #ifndef UTILS_H
2  #define UTILS_H
3
4  #include <set>
5  #include <vector>
6  using namespace std;
7
8  template<typename T>
9  void print_set(const set<T>& s, bool newline=true) {
10     cout << "Set: ";
11     for (auto x : s)
12         cout << x << " ";
13     if (newline)
14         cout << endl;
15 }
16
17 template<typename T>
18 void print_vector(const vector<T> v) {
19     cout << "Vector: ";
20     for (auto x : v)
21         cout << x << " ";
22     cout << endl;
23 }
24
25 #endif // UTILS_H

```

Makefile

```

1  ifdef NO_STDIN
2  FLAGS = -DNO_STDIN
3  endif
4
5  all: regex
6
7  regex: main.cpp
8      g++ $< -g $(FLAGS) -o $@
9
10 .PHONY: clean
11 clean:
12     rm regex

```

在Linux环境下输入make可自动编译，make NO_STDIN=1则编译出直接从源码常量数组中读取输入数据的程序。

附录 B. 测试数据

regex1	regex2	output	描述
((E a)b*)*	(a b)*	=	标准测试1
b*a*b?a*	b*a*ba* b*a*	=	标准测试2
b*a*b?a*	(b* a*)(b E)a*	>	标准测试3
(c d)*c(c d)(c d)	(c d)*d(c d)(c d)	!	标准测试4
x+y+z+	x*y*z*	<	标准测试5
a	a+	<	最简测试
(a b)*abb	(a b)*abbb*	<	课本样例
(a b)*c+(d e)?	(a b)*cd	>	混杂所有符号
((x y)+)+	(x y)+(y+b*)*	<	嵌套括号
(https http)	ht*ps	!	彼此不含
ht+ps	(htt*ps http)+	<	多种符号
a+b*c?	a+b*c?	=	完全等同
a*b*c	a+b*c	>	星号与加号比较