

**院 系：计 算 机 学 院**

**实验课程：编译原理**

**实验项目：XLEX生成器**

**指导老师：黄煜廉**

**专 业：计算机科学与技术**

**班 级：2018级 3 班**

**学 生：**

**学 号：**

**华南师范大学教务处**

**XLEX生成器**

**一．实验目的**

设计一个应用软件，以实现将正则表达式-->NFA--->DFA-->DFA最小化-->词法分析程序。

1. **实验内容**

1.要提供一个源程序编辑界面，让用户输入正则表达式（可保存、打开源程序）

2.需要提供窗口以便用户可以查看转换得到的NFA（用状态转换表呈现即可）

3.需要提供窗口以便用户可以查看转换得到的DFA（用状态转换表呈现即可）

4.需要提供窗口以便用户可以查看转换得到的最小化DFA（用状态转换表呈现即可）

5.需要提供窗口以便用户可以查看转换得到的词法分析程序（该分析程序需要用C语言描述）

**三．设计思路**

**1.正则表达式到NFA**

(1). 先让用户输入或选择文件导入正则表达式

(2). 获取正则表达式后先判断是否含有禁止符号(即：~，本实验让~作为空)

(3). 将正则表达式中的连接运算显示使用“$”来表示

(4). 将正则表达式转为后缀表达式

(5). 使用汤姆森构造法(Thompson Construction)，将正则表达式构造为NFA

**2.NFA到DFA**

(1). 将获取到的NFA使用子集构造法(Subset Construction)转为DFA

(2). 使用二维数组记录下闭包运算的各个子集(每个元素都是NFA对应的标号)，以便输入状态转移表。其次记录下各个节点的状态

**3.DFA到最小DFA**

(1). 使用 上一步记录下来的各个子集的二位数组和记录下来的状态表，对逐个状态进行比较，判断是否可以进行合并，从而进行最小化DFA。

**4.最小DFA生成词法分析程序**

(1).使用固定的格式，最终建立起由扫描DFA来进行判断的词法分析程序。

**四．程序源代码**

程序的主要源代码如下：（具体请看文件中的头文件和 源文件）

1. **用Thompson构造法构造NFA**

// 用Thompson构造法构造NFA

char\*\* toNFA::toNfa\_main(string express){

//建立并初始化NFA图存储

nfaGraph = new char\*[ 2\*express.length() ];

for(int i=0; i<2\*express.length(); i++){

nfaGraph[i]=new char[2\*express.length()];

}

for(int i=0; i<2\*express.length(); i++){

for(int j=0; j<2\*express.length(); j++){

nfaGraph[i][j]=NULL;

}

}

//开始建图

stack<nfa> NFA\_stack;

nfa temp\_nfa,temp1,temp2;

int statu\_num=0;

for(int i=0; i<express.length(); i++){

//为标识符情况

if( (express[i]>='a' && express[i]<='z') || (express[i]>='A' && express[i]<='Z') || (express[i]>='0' && express[i]<='9') ){

temp\_nfa.start=statu\_num; temp\_nfa.end=statu\_num+1; statu\_num += 2;

nfaGraph[temp\_nfa.start][temp\_nfa.end]=express[i];

NFA\_stack.push(temp\_nfa);

}

else{//不为标识符情况

switch(express[i]){

case '$':{//新加statu在两个nfa中间

//从temp1到temp2

temp2=NFA\_stack.top(); NFA\_stack.pop();

temp1=NFA\_stack.top(); NFA\_stack.pop();

nfaGraph[temp1.end][statu\_num]='~';

nfaGraph[statu\_num][temp2.start]='~';

statu\_num++;

//新建的nfa图

temp\_nfa.start=temp1.start;

temp\_nfa.end=temp2.end;

NFA\_stack.push(temp\_nfa);

break;

}

case '|':{//新加statu在两个nfa开头

temp2=NFA\_stack.top(); NFA\_stack.pop();

temp1=NFA\_stack.top(); NFA\_stack.pop();

nfaGraph[statu\_num][temp1.start]='~';

nfaGraph[statu\_num][temp2.start]='~';

temp\_nfa.start=statu\_num;

statu\_num++;

nfaGraph[temp2.end][statu\_num]='~';

nfaGraph[temp1.end][statu\_num]='~';

temp\_nfa.end=statu\_num;

statu\_num++;

NFA\_stack.push(temp\_nfa);

break;

}

case '\*':{//新加的两个statu在一个nfa开头

temp1=NFA\_stack.top(); NFA\_stack.pop();

nfaGraph[statu\_num][temp1.end]='~';

nfaGraph[statu\_num][temp1.start]='~';

temp\_nfa.start=statu\_num;

statu\_num++;

nfaGraph[statu\_num][temp1.start]='~';

nfaGraph[temp1.end][statu\_num]='~';

temp\_nfa.end=statu\_num;

statu\_num++;

NFA\_stack.push(temp\_nfa);

break;

}

}

}

}

//得到NFA的初态和终态

temp\_nfa=NFA\_stack.top();

Start=temp\_nfa.start;

End=temp\_nfa.end;

status\_num=statu\_num;

return nfaGraph;

1. }**用子集构造法构造DFA**

// 用子集构造法构造DFA

char\*\* toDFA::toDFA\_main(){

int statu\_set\_num=0;

//获得初态集合

set<int> Start\_statu; //初始状态集合

stack<int> status; //DFS用

int temp\_start;

status.push(nfaStart);

Start\_statu.insert(nfaStart);

while(!status.empty()){

temp\_start=status.top();

for(int i=0; i<Graphsize; i++){

if(nfaGraph[temp\_start][i]=='~' && (Start\_statu.find(i)==Start\_statu.end() )){

status.push(i);

temp\_start=i; i=0;

Start\_statu.insert(temp\_start);

}

}

status.pop();

}

Status[Start\_statu]=statu\_set\_num;

statu\_set\_num++;

//开始建图并初始化

stack< set<int> > Statu\_stack;

set<int> goal\_statu,temp;

dfaGraph=new char\*[Graphsize];

for(int i=0; i<Graphsize; i++){

dfaGraph[i]=new char[Graphsize];

}

for(int i=0; i<Graphsize; i++){

for(int j=0; j<Graphsize; j++)

dfaGraph[i][j]=NULL;

}

Statu\_stack.push(Start\_statu);

while(!Statu\_stack.empty()){

goal\_statu=Statu\_stack.top();

for(int i=0; i<symbol.length(); i++){

temp=findStatuSet(goal\_statu,symbol[i]);

if(temp.size()!=0){

if(Status.find(temp)==Status.end()){

Status[temp]=statu\_set\_num;

statu\_set\_num++;

Statu\_stack.push(temp);

}

dfaGraph[Status[goal\_statu]][Status[temp]]=symbol[i];

}

}

Statu\_stack.pop();

}

Start=0;

dfaStatus=Status;

return dfaGraph;

1. }**用子集构造法构造DFA**

//最小化DFA

int\*\* miniDFA::miniDFA\_main(){

int Graphsize=initial\_Status.size();

map<char,int> Symbol\_no;

set<int> end\_set,nend\_set; //终态与非终态集合

map<int,set<int> > temp\_map; //暂用map

map<int,set<int> >::iterator temp\_it; //暂用map的指针

map<set<int>,int>::iterator initialmap\_it; //初始DFA的map指针

stack<set<int> > goal\_stack; //目标集合栈

stack<set<int> > temp\_stack; //暂用集合栈

set<int> temp; //暂用集合（用于获取从stack中取出的集合）

set<int>::iterator tempset\_it; //暂用集合指针

set<int> maptemp; //暂用集合(用于存储将要放入temp\_map的数据的集合)

int status\_no;

for(int i=0; i<Symbol.length(); i++){

Symbol\_no[Symbol[i]]=i;

}

//先绘制转换表

//建立并初始化

int\*\* Switch=new int\*[Graphsize];

for(int i=0; i<Graphsize; i++){

Switch[i]=new int[Symbol.length()];

}

for(int i=0; i<Graphsize; i++){

for(int j=0; j<Symbol.length(); j++){

Switch[i][j]=-1;

}

}

for(int i=0; i<Graphsize; i++){

for(int j=0; j<Graphsize; j++){

if(initial\_DFA[i][j]!=NULL){

Switch[i][Symbol\_no[initial\_DFA[i][j]] ]=j;

}

}

}

//开始通过扫描转换表得到最小化DFA

//划分状态集合

//划分终态集和非终态集

for(initialmap\_it=initial\_Status.begin(); initialmap\_it!=initial\_Status.end(); initialmap\_it++){

if( (initialmap\_it->first).find(NFAend)==(initialmap\_it->first).end() ){ //为非终态集情况

nend\_set.insert( (initialmap\_it->second) );

}

else{ //为终态集情况

end\_set.insert( (initialmap\_it->second) );

}

}

goal\_stack.push(nend\_set); goal\_stack.push(end\_set);

//划分具体集合

for(int i=0; i<Symbol.length(); i++){

temp\_stack=goal\_stack;

while(!goal\_stack.empty()) goal\_stack.pop();

while(!temp\_stack.empty()){

//开始进行stack中一个集合的划分

temp=temp\_stack.top(); temp\_stack.pop();

for(tempset\_it=temp.begin(); tempset\_it!=temp.end(); tempset\_it++){

status\_no=Switch[\*tempset\_it][Symbol\_no[Symbol[i]]];

if(temp\_map.find(status\_no)==temp\_map.end()){ //不在temp\_map内部

maptemp.insert(\*tempset\_it);

temp\_map[status\_no]= maptemp;

maptemp.clear();

}

else{ //在temp\_map内部

temp\_it=temp\_map.find(status\_no);

(temp\_it->second).insert(\*tempset\_it);

}

}

//完成一个集合的划分

for(temp\_it=temp\_map.begin(); temp\_it!=temp\_map.end(); temp\_it++){

goal\_stack.push((temp\_it->second));

}

temp\_map.clear();

}

}

//转换表建图

//初始化表格

DFA=new int\* [goal\_stack.size()];

for(int i=0; i<goal\_stack.size(); i++){

DFA[i]=new int[Symbol.length()];

}

for(int i=0; i<goal\_stack.size(); i++){

for(int j=0; j<Symbol.length(); j++){

DFA[i][j]=-1;

}

}

//将goal\_stack内各个集合编号

int status\_number=0;

map<int,set<int> > goal\_map;

while(!goal\_stack.empty()){

temp=goal\_stack.top(); goal\_stack.pop();

goal\_map[status\_number]=temp;

status\_number++;

if(temp.find(0)!=temp.end()) Start=status\_number-1;

}

//开始正式建立表格

int now\_no,switch\_now;

for(temp\_it=goal\_map.begin(); temp\_it!=goal\_map.end(); temp\_it++){

map<int,set<int> >::iterator map\_it;

now\_no=\*((temp\_it->second).begin());

for(int i=0; i<Symbol.length(); i++){

switch\_now=Switch[now\_no][Symbol\_no[Symbol[i]]];

for(map\_it=goal\_map.begin(); map\_it!=goal\_map.end(); map\_it++){

if((map\_it->second).find(switch\_now)!=(map\_it->second).end()){

DFA[(temp\_it->first)][Symbol\_no[Symbol[i]]]=map\_it->first;

}

}

}

}

Col=status\_number;

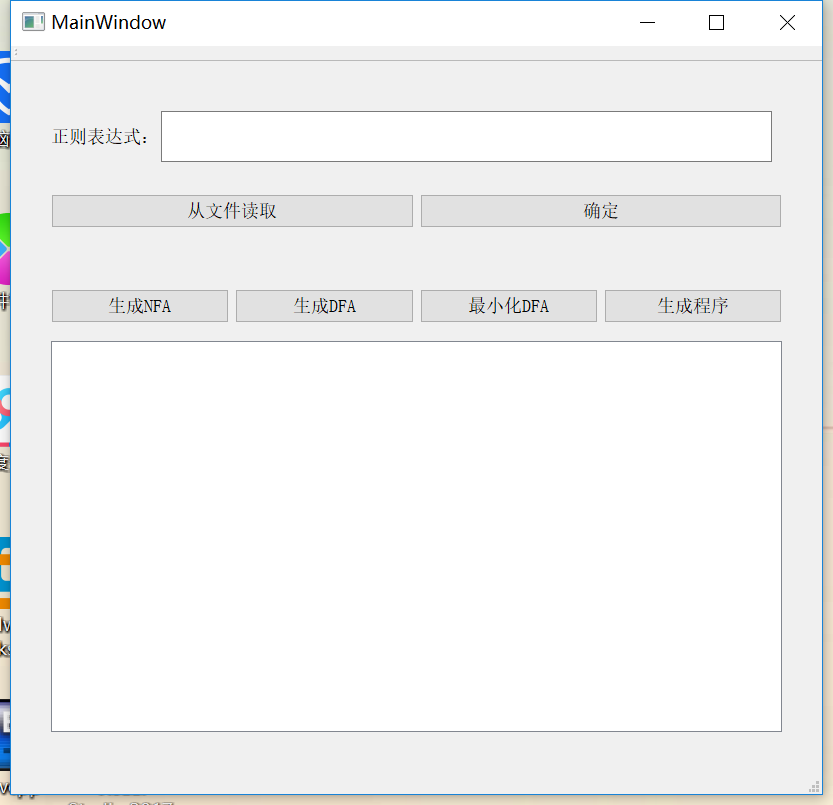
return DFA;

}

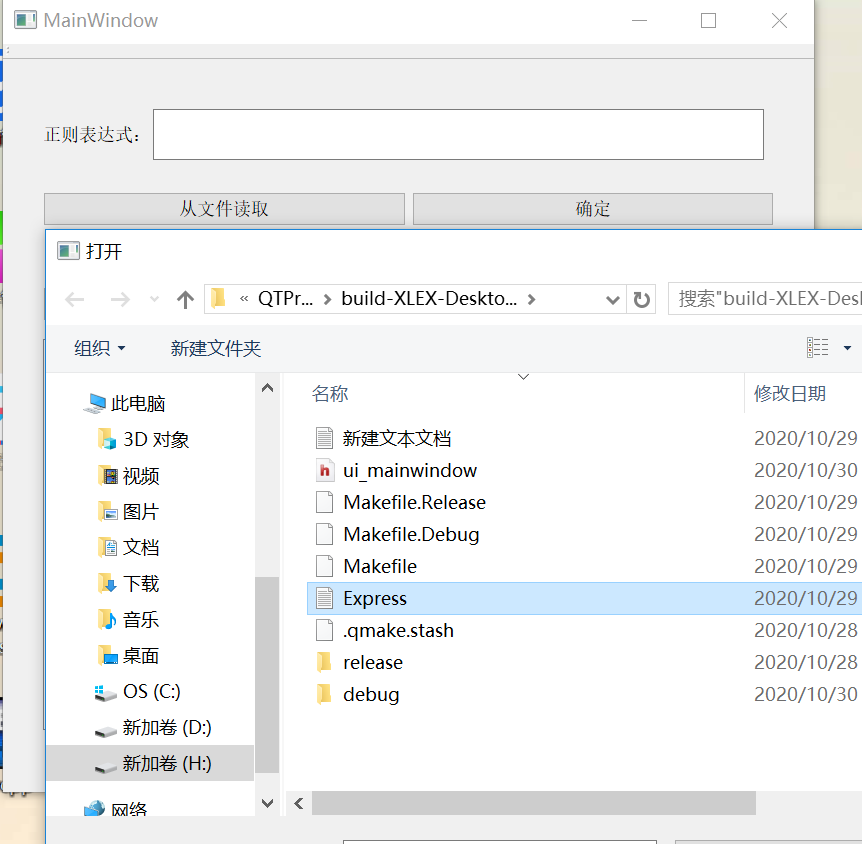
**五．实验结果**

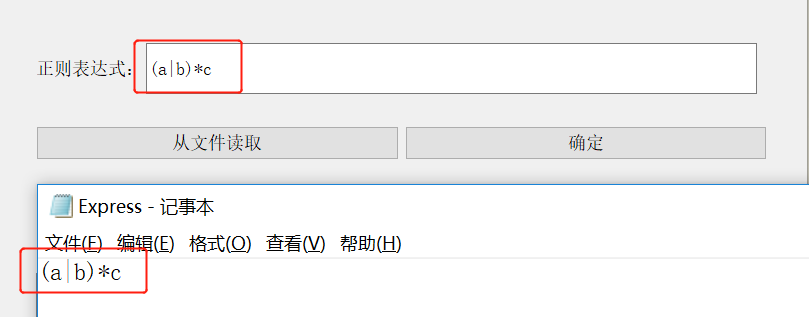
运行程序，截图如下：

1. 选择从文件输入正则表达式或手动输入

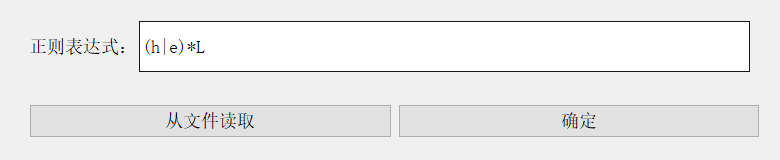


由文件扫描：

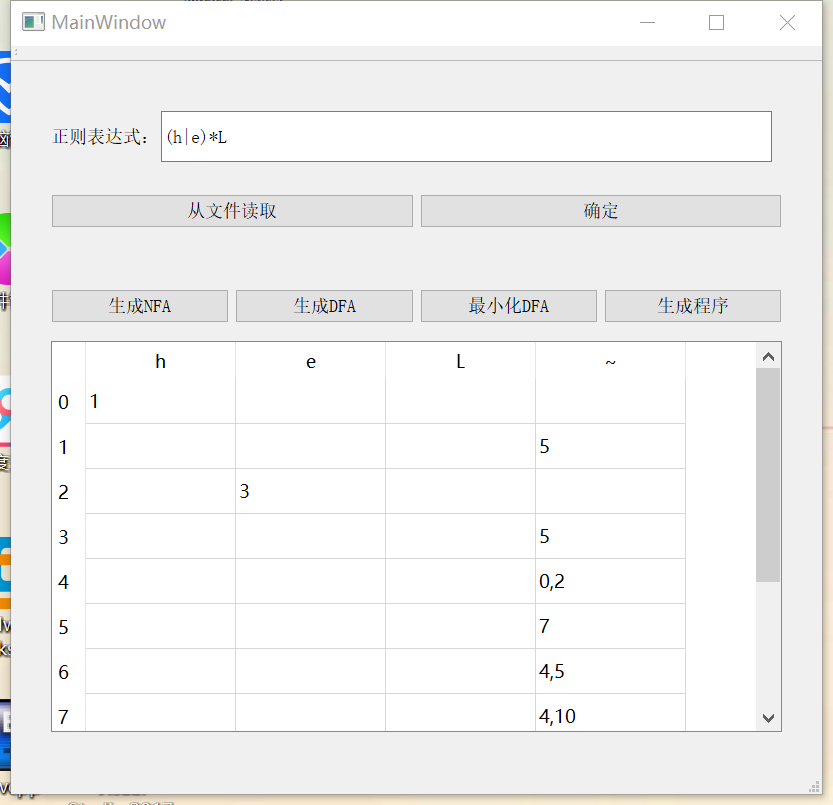




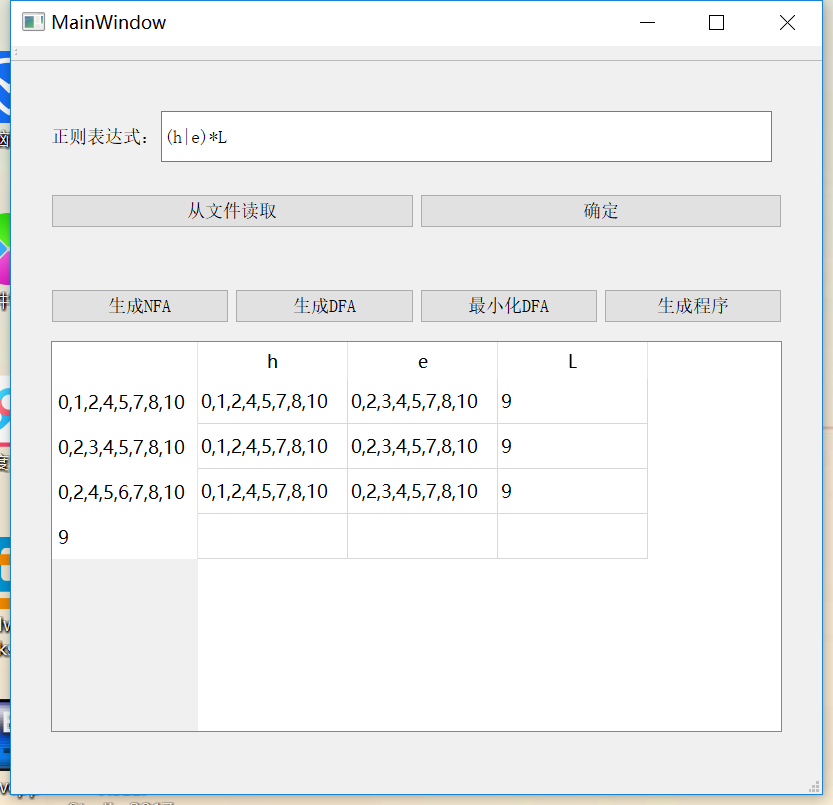
手动输入：



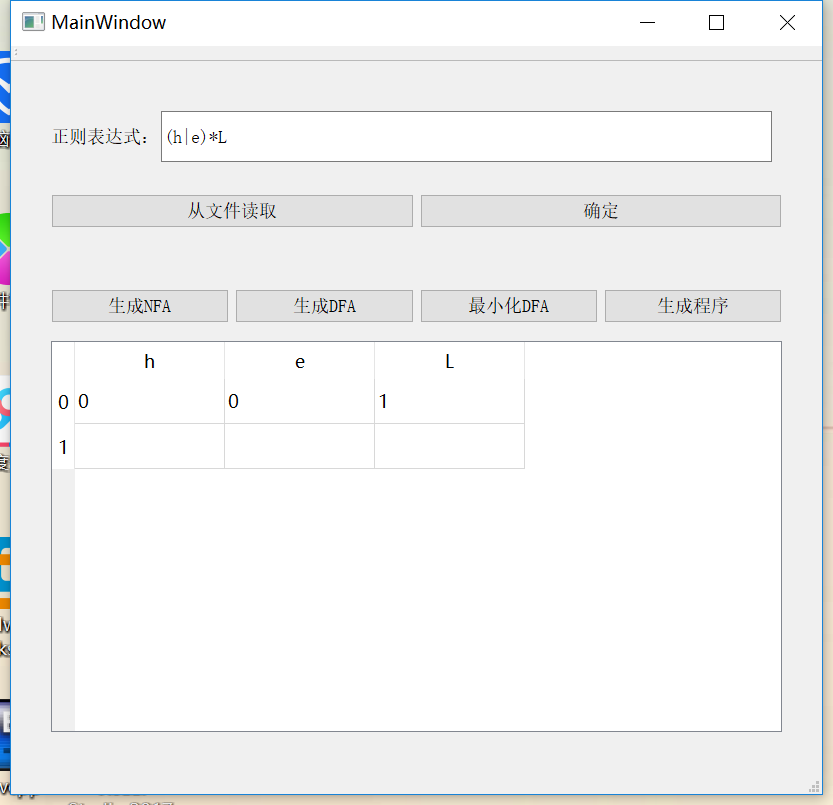
1. 生成NFA



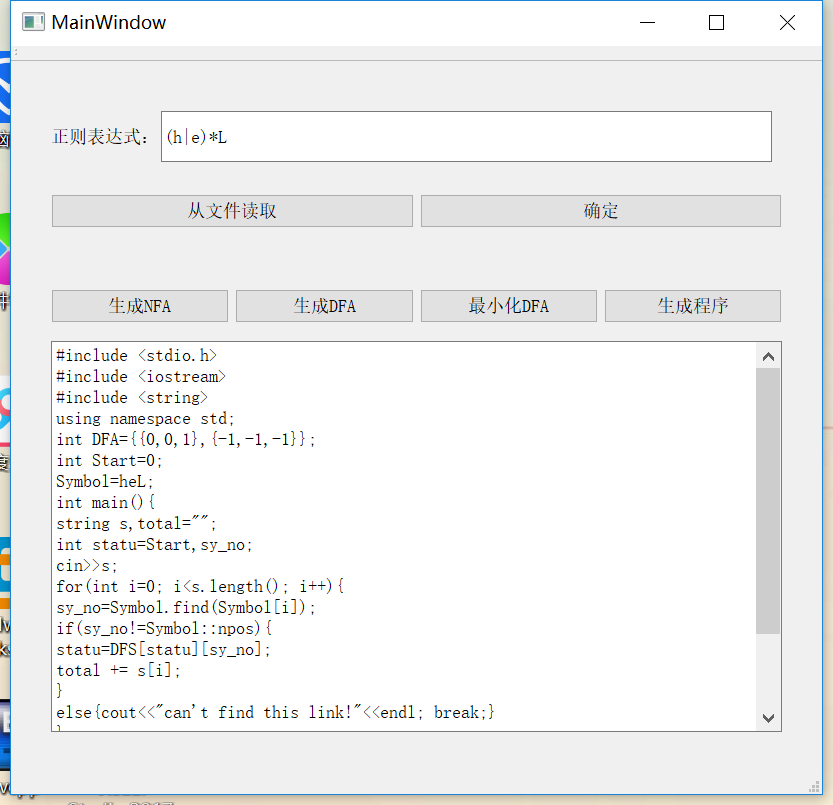
1. 生成DFA



1. 生成最小DFA



1. 生成词法分析程序



**六．实验小结**

本次实验通过对编译原理中词法分析器的实现，更加深刻地了解了词法分析的相关知识，对于正则表达式的理解更进一步。

经过测试，对于正则表达式可以很好的生成NFA，DFA，最小DFA，最后生成可以直接运行的词法分析程序的cpp文件。

虽然在理论在知道正则表达式，NFA，DFA之间的转换关系，但在编程时却有不少阻碍，本实验中相当一部分的编程技巧和思路都来源于博客，而源码均为本人手动编写。耗时两周左右。

1. **实验小结**

参考资料：

1. :黄煜廉老师的课件
2. :博客:https://blog.csdn.net/johnf\_nash/article/details/22086149，正则表达式转NFA
3. :博客:https://blog.csdn.net/Hungryof/article/details/46822841，NFA到DFA的转换