目录

[一、 前言 2](#_Toc106799953)

[二、 开发过程 2](#_Toc106799954)

[（一） 设计思路 2](#_Toc106799955)

[（二） 实现过程 2](#_Toc106799956)

[三、 设计方案 7](#_Toc106799957)

[（一） 功能组成 7](#_Toc106799958)

[（二） 界面组成 7](#_Toc106799959)

[（三） 例子结果展示(aa|bb)\*(a|b)(aa|bb)\* 8](#_Toc106799960)

[（四） 代码实现 10](#_Toc106799961)

# 前言

本次实验是实验一个词法生成器，可以通过该程序实现将正则表达式转化成NFA，将NFA转成DFA，将DFA化简成最小化的DFA，最后再将最小化的DFA转化成词法分析程序，用c++语言表示，其中NFA，DFA，最小化DFA用表格表示出来。

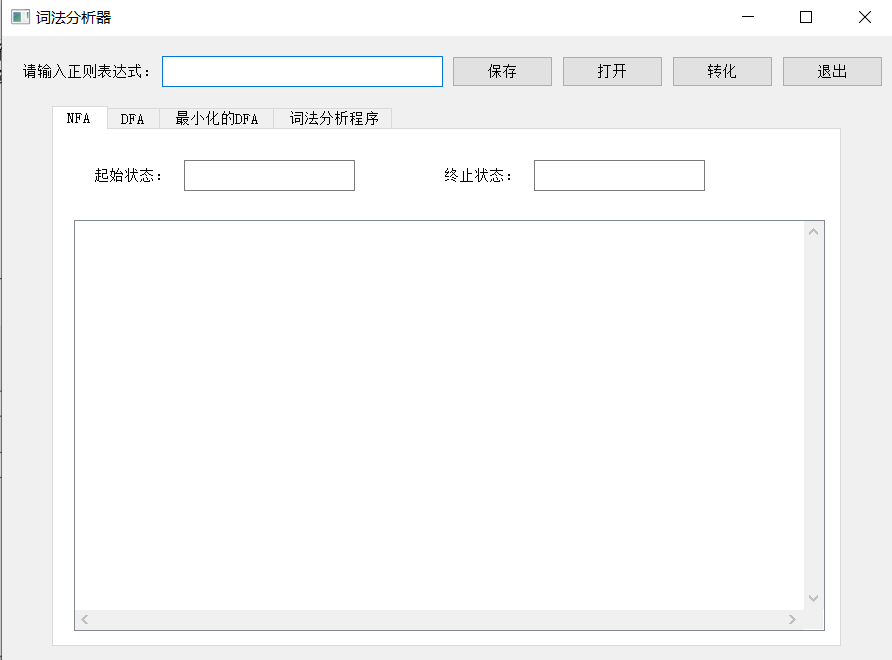
# 开发过程

## 设计思路

1. 首先完成页面布局
2. 保存，打开存有正则表达式的功能按钮的实现
3. 实现从正则表达式转化成NFA
4. 实现从NFA转化成DFA
5. 实现将DFA简化成最小化的DFA
6. 实现将最小化的DFA转成词法分析程序，用c++语言表示
7. 退出系统

## 实现过程

1. 界面设计如下：



1. 保存，打开存有正则表达式的功能按钮的实现

使用Qt的文件QfileDialog、QFile和QTextStream实现保存和打开功能。

fileDialog.getSaveFileName()函数实现获取保存文件的路径，再使用QFile和QTextStream将正则表达式写入文件。

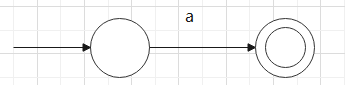
QFileDialog::getOpenFileName()函数实现获取打开文件的路径名，再使用

QFile和QTextStream将正则表达式读取出来并显示在编辑框上。

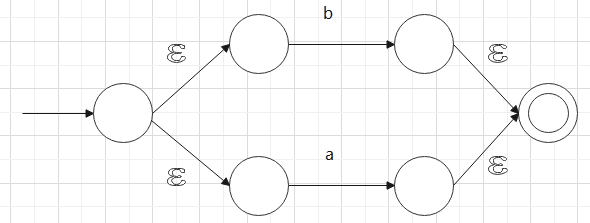
1. 实现从正则表达式转化成NFA

由于正则表达式的运算符有优先级，所以先将正则表达式转化成逆波兰表示法，其中正则表达式的连接运算符用&来表示。NFA图用链接矩阵来存储。首先定义一个栈，栈存放操作数的起始状态和终止状态。对逆波兰表示的正则表达式依次扫描，当扫描到操作数则把基本操作数的NFA图建立起来，当扫描到 | 则从栈中弹出两个操作数的起始和终止状态进行 | 运算符NFA图的选择，将完成 | 运算的新NFA图的起始状态和终止状态重新压入栈，当扫描到 & 则从栈中弹出两个操作数的起始和终止状态进行 & 运算符NFA图的连接，将完成 & 运算的新NFA图的起始状态和终止状态重新压入栈，当扫描到 \* 则从栈中弹出一个操作数的起始和终止状态进行\*运算，将完成 \* 运算的新NFA图的起始状态和终止状态重新压入栈，依次完成直到扫描完整个表达式，如果正则表达式有误的话，系统将异常退出。

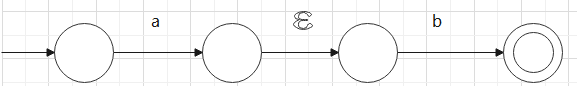
基本操作数的NFA图表示如下：a



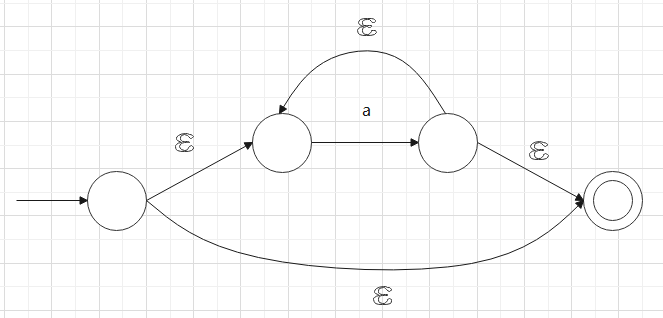
a|b的NFA图表示如下：



ab的NFA图表示如下：



a\*的NFA图表示如下：



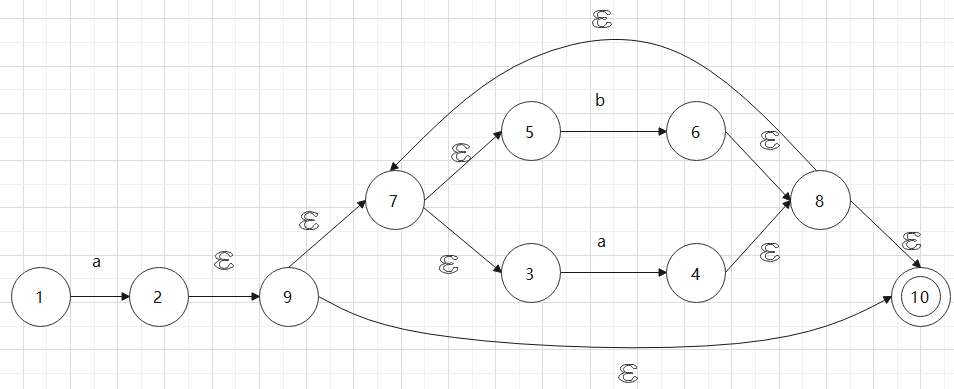
1. 实现从NFA转化成DFA

注意：NFA图的空操作数用@来表示

首先先求出各个状态经过空操作数（‘@’）到达的状态的集合，用深度优先搜索获得其状态集合。由于@是空串，所以经过@相连的状态都一样，消除@转换即@-闭包的构造。接着消除多重转换，把状态集合通过某个操作数得到的状态，求其得到状态的@-闭包的并集，整个算法叫做子集构造。DFA的初态只有一个，将NFA的初态进行等价合并即@-闭包的构建。先将DFA的初态保存起来作为DFA的第一个状态，然后根据这个状态去找经过出现在正则表达式里面非空串的操作数所到达的状态的集合，得到该集合若已存在则在DFA的状态里不建立新的状态，如果不存在，则将这个集合作为新的状态加入DFA中，以此类推直到DFA没有新的状态产生，则得到了存放DFA图的邻接矩阵。

例子：a(a|b)\* （该例子是用上述描述的算法来走一遍的）

NFA图如下：



状态1的@-闭包：｛1｝

状态2的@-闭包：｛2，9，7，5，3，10｝

状态3的@-闭包：｛3｝

状态4的@-闭包：｛4，8，7，3，5，10｝

状态5的@-闭包：｛5｝

状态6的@-闭包：｛6，8，7，3，5，10｝

状态7的@-闭包：｛7，3，5｝

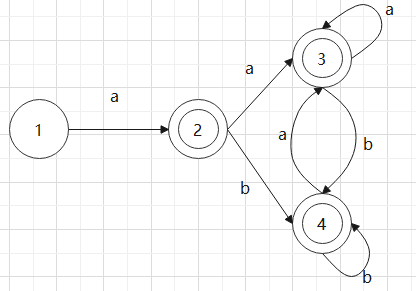
状态8的@-闭包：｛8，7，3，5，10｝

状态9的@-闭包：｛9，7，3，5，10｝

状态10的@-闭包：｛10｝

起始状态集合｛1｝作为DFA的起始状态。

起始状态｛1｝经过操作数a到达的状态有2，所以得到的状态集合为｛2，9，7，5，3，10｝，该集合在DFA中未出现为新状态即加入DFA作为第二个状态；起始状态｛1｝经过操作数b到达的状态为空，所以不产生新的状态。第二个状态是｛2，9，7，5，3，10｝经过a操作数到达的状态有4，所以得到的状态集合为｛4，8，7，3，5，10｝该集合在DFA中未出现为新状态即加入DFA作为第三个状态；第二个状态｛2，9，7，5，3，10｝经过操作数b到达的状态有6，所以得到的状态集合为｛6，8，7，3，5，10｝，该集合在DFA中未出现为新状态即加入DFA作为第四个状态。第三个状态｛4，8，7，3，5，10｝经过操作数a得到的是DFA的第三个状态，经过操作数b得到的状态是DFA的第四个状态，没有产生新的状态。第四个状态｛6，8，7，3，5，10｝经过操作数a得到的是DFA的第三个状态，经过操作数b得到的状态是DFA的第四个状态，没有产生新的状态，循环结束，DFA构造完成，如下图：



1. 实现将DFA简化成最小化的DFA

首先先将未化简的DFA的状态划分为终态集合和非终态集合，这是先假设最小化的DFA只有这两个状态集合构成，然后再逐步细化去划分。若划分集合只有一个状态则不再进行划分作为一个最小化DFA的一个状态。对已有划分状态进行循环，判断该划分状态能否再进行划分，当该划分状态存在两个状态经过操作数到达的状态不属于统一划分集合时则将这两个状态划分为两个集合，一次类推直到没有新的划分集合产生即结束，最后有多少个划分集合最小化DFA就要多少个状态。

例子：a(a|b)\*

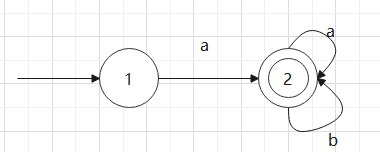
有刚刚得到的DFA图将其转化成状态表如下：

|  |  |  |
| --- | --- | --- |
|  | a | b |
| 1 | 2 |  |
| 2 | 3 | 4 |
| 3 | 3 | 4 |
| 4 | 3 | 4 |

非终态集合是：｛1｝

终态集合是｛2，3，4｝

由于非终态集合只有一个状态不在进行划分作为最小化DFA的第一个状态，现到终态集合｛2，3，4｝，由于状态2，3，4经过操作数a，b到达的状态都属于终态集合｛2，3，4｝，所以他们属于同一个状态。即最小化DFA有两个状态一个是｛1｝，一个是｛2，3，4｝。其最小化的DFA图如下：



1. 实现将最小化的DFA转成词法分析程序，用c++语言表示

根据最小化DFA状态表的走向写出对应程序即可。

例子：a(a|b)\*

int status = 1;

while(status == 1 || status == 2)

{ switch(status){

case 1 :

if input() == 'a':

status = 2;

else: status = -1; //进入错误状态或其他状态跳出循环

break;

case 2 :

if input() == 'a':

status = 2;

if input() == 'b':

status = 2;

else: status = -1; //进入错误状态或其他状态跳出循环

break;

default: break;

}

}

1. 退出按钮功能

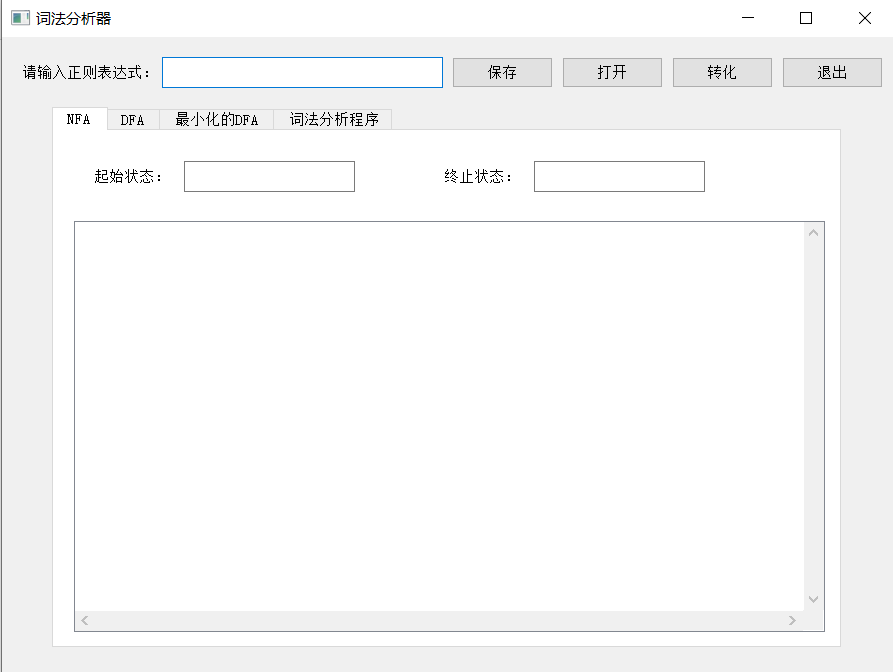
exit(0)

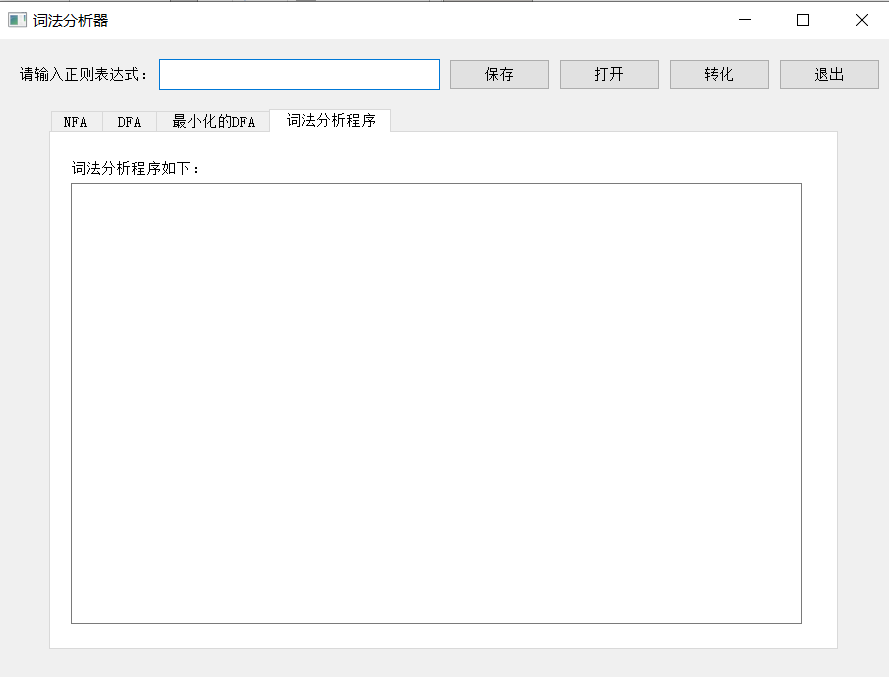
# 设计方案

## 功能组成

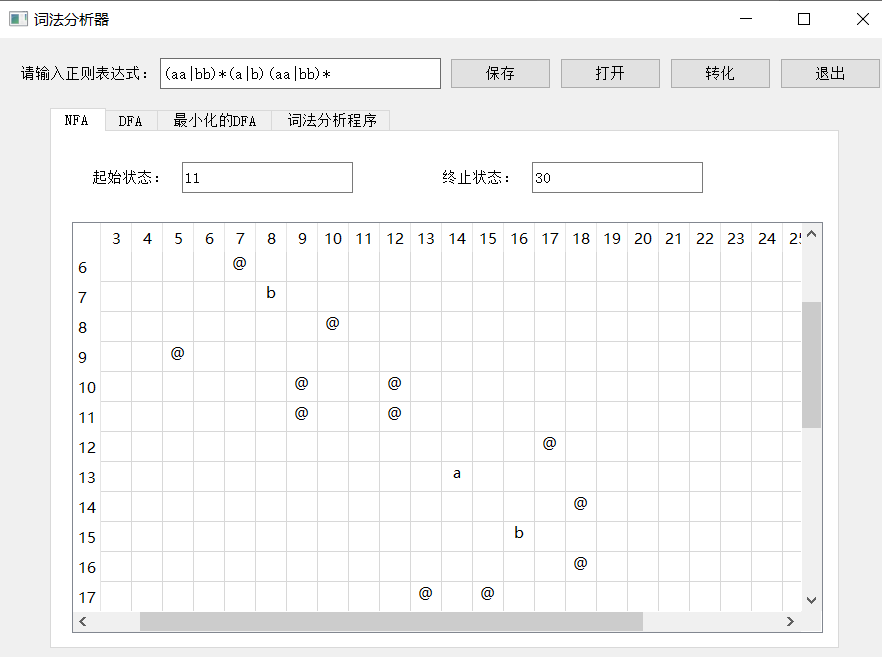
1. 保存正则表达式
2. 打开正则表达式
3. 将正则表达式转成NFA
4. 将NFA转成DFA
5. 将DFA化简成最小化的DFA
6. 将最小化的DFA转成词法分析程序
7. 退出系统

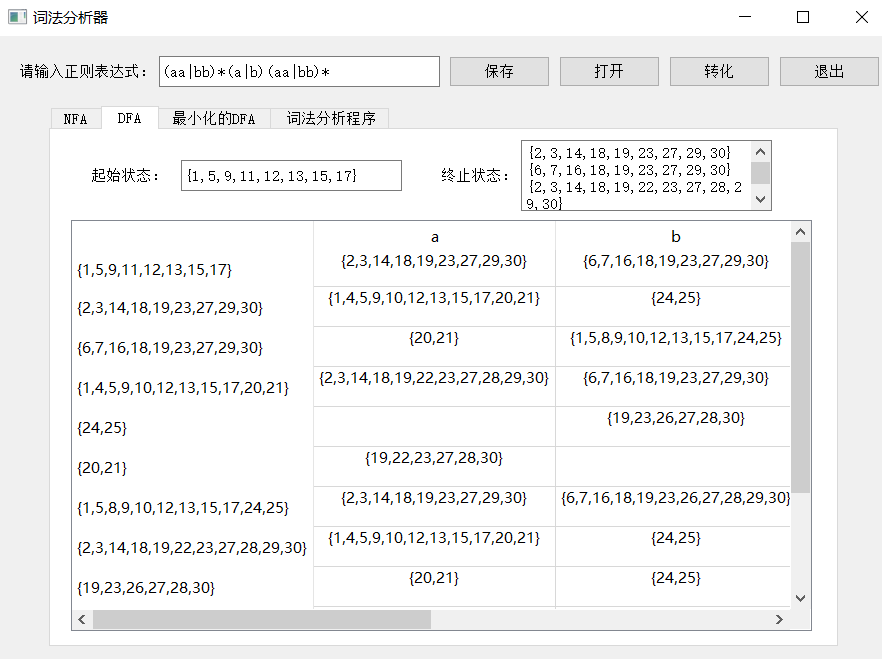
## 界面组成





## 例子结果展示(aa|bb)\*(a|b)(aa|bb)\*









## 代码实现

1. 保存正则表达式

//保存正则表示式

connect(ui->pB\_save,&QPushButton::clicked,this,[=](){

QString exp = ui->lE\_exp->text();

if(exp == "")

{

QMessageBox::warning(this, tr("提示") ,tr("正则表达式为空，无法保存！"));

}

else{

QFileDialog fileDialog;

QString file\_name = fileDialog.getSaveFileName(this,tr("Open File"),"/text",tr("Text File(\*.txt)"));

QFile file(file\_name);

if(!file.*open*(QIODevice::WriteOnly|QIODevice::Text))

return;

QTextStream out(&file);

QString exp = ui->lE\_exp->text();

out<<exp;

}

});

1. 打开正则表达式

//打开写有正则表达式的文本文件到编辑框

connect(ui->pB\_open,&QPushButton::clicked,this,[=](){

QString file\_name = QFileDialog::getOpenFileName(this,"获取正则表达式文本文件",".","\*.txt");

QFile file(file\_name);

if(!file.*open*(QIODevice::ReadOnly | QIODevice::Text))

return;

QTextStream in(&file);

QString exp = in.readLine();

ui->lE\_exp->setText(exp);

});

1. 将正则表达式转成NFA

QString MainWindow::**addSign**(QString exp)//由于正则表达式没有连接的元符号，了方便根据优先级转成RPN，加入&作为连接

{

QString exp2 = ""; //存放更改后的表达式

for(int i=1;i<exp.length();i++)

{

exp2 = exp2 + exp[i-1];

if((exp[i-1] != '|' && exp[i-1] != '(' && exp[i] == '(') || (exp[i-1]=='\*' && exp[i]!='|' && exp[i]!=')') ||

(exp[i-1] == ')' && (exp[i] != '|' && exp[i] !='\*' && exp[i]!=')')) ||

(exp[i-1]!=')' && exp[i-1]!='\*' && exp[i-1]!='|' && exp[i-1]!='(' && exp[i]!=')' && exp[i]!='\*' && exp[i]!='|' && exp[i]!='(')){

exp2 = exp2 + '&';

}

}

exp2 = exp2 + exp[exp.length()-1];

return exp2;

}

QVector<QChar> MainWindow::**changeToRPN**(QString exp) //转为逆波兰表示法

{

/\*

优先级从大到小排序为：(),\*,&,|

定义两个容器，一个放操作数，一个放运算符

遇到操作数就直接压入num容器，遇到运算符则要分优先级处理，处理如下：

遇到左括号，直接压入op容器；

遇到右括号，把左括号前面的操作数弹出压入num容器；

遇到\*，直接压入容易，优先级较高，先运算

遇到&，把优先级比&高或相等的运算符弹出到num容器里（注意左括号），再将自己压入num容器

遇到|，把op容器里的运算符全压入num容器（若有左括号先将左括号后面的运算符压入num容器），再将自己压入num容器

\*/

QVector<QChar> num;

QVector<QChar> op;

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

if(exp[i]=='('){

op.push\_back(exp[i]);

}

else if(!op.empty() && exp[i]==')'){

while(op.back()!='('){

num.push\_back(op.back());

op.pop\_back();

}

op.pop\_back();

}

else if(exp[i]=='\*'){

op.push\_back(exp[i]);

}

else if(exp[i]=='&'){

while(!op.empty()){

if(op.back()=='|' || op.back()=='(')

break;

num.push\_back(op.back());

op.pop\_back();

}

op.push\_back(exp[i]) ;

}

else if(exp[i]=='|'){

while(!op.empty()){

if(op.back()=='(')

break;

num.push\_back(op.back());

op.pop\_back();

}

op.push\_back(exp[i]) ;

}

else{

num.push\_back(exp[i]);

}

}

while(!op.empty())

{

num.push\_back(op.back());

op.pop\_back();

}

return num;

}

//用于查找某个元素是都在vector里面

template<typename T> bool MainWindow::**find**(QVector<T> vt, T value)

{

for(int i=0;i<vt.size();i++)

{

if(vt[i] == value)

return true;

}

return false;

}

void MainWindow::**changeToNFA**(QVector<QChar> exp) //将正则表达式转成NFA

{

QStack<int> s; //存放表达式操作数的初始状态和结束状态

start = endless = 0;

for(int i=0;i<exp.size();i++){

if(exp[i] == '|'){

int start1, start2, end1, end2;

end2 = s.top();

s.pop();

start2 = s.top();

s.pop();

end1 = s.top();

s.pop();

start1 = s.top();

s.pop();

start = ++count1;

endless = ++count1;

e[start][start1] = '@'; //该字符用来表示空

e[start][start2] = '@';

e[end1][endless] = '@';

e[end2][endless] = '@';

s.push(start);

s.push(endless);

}

else if(exp[i] == '&'){

int start1, start2, end1, end2;

end2 = s.top();

s.pop();

start2 = s.top();

s.pop();

end1 = s.top();

s.pop();

start1 = s.top();

s.pop();

start = start1;

endless= end2;

e[end1][start2] = '@';

s.push(start);

s.push(endless);

}

else if(exp[i] == '\*'){

int start1,end1;

end1 = s.top();

s.pop();

start1 = s.top();

s.pop();

start = ++count1;

endless = ++count1;

e[start][start1] = '@';

e[end1][endless] = '@';

e[start][endless]= '@';

e[end1][start1] = '@';

s.push(start);

s.push(endless);

}

else

{

if(!find<QChar>(num,exp[i]) && exp[i] != '@') //将还没出现的操作数放入num

num.push\_back(exp[i]);

int tempStart,tempEnd; //存放操作数的起始状态和结束状态压入栈

tempStart = ++count1;

tempEnd = ++count1;

e[tempStart][tempEnd] = exp[i];

start = tempStart;

endless = tempEnd;

s.push(tempStart);

s.push(tempEnd);

}

}

}

1. 将NFA转成DFA

//求图的第一个邻接顶点

int MainWindow::**getFirstNeighbor**(int vertex)

{

for(int i=1;i<=count1;i++)

{

if(e[vertex][i]!='\0')

return i;

}

return -1; //没有邻接顶点返回-1

}

//求图的其他邻接顶点

int MainWindow::**getNextNeighbor**(int vertex,int w)

{

for(int i=w+1; i <= count1;i++)

{

if(e[vertex][i]!='\0')

return i;

}

return -1; //没有在w之后的邻接顶点了

}

//判断经过空操作数（‘@’）到达的顶点有哪些

void MainWindow::**getThroughNumV**(int vertex, char ch, bool visit[])

{

visit[vertex] = true;

temp.push\_back(vertex);

int w = getFirstNeighbor(vertex);

while(w!=-1)

{

if(e[vertex][w] == ch && !visit[w])

{

getThroughNumV(w, ch, visit);

}

w = getNextNeighbor(vertex, w);

}

}

//判断某个顶点有没有经过某个操作数到达另一个状态数，将其经过某个操作数到达的状态集合保存起来

QVector<int> MainWindow::**getSetThroughNum**(int vertex, QChar ch)

{

QVector<int> vt;

for(int i=1;i<=count1;i++)

{

if(e[vertex][i]==ch){

vt.push\_back(i);

}

}

return vt;

}

//两个vector求并集

QVector<int> MainWindow::**vectors\_set\_union**(QVector<int> vt1,QVector<int> vt2){

QVector<int> v;

sort(vt1.begin(),vt1.end());

sort(vt2.begin(),vt2.end());

set\_union(vt1.begin(),vt1.end(),vt2.begin(),vt2.end(),back\_inserter(v)); //求交集

return v;

}

//NFA转DFA

void MainWindow::**changeToDFA**()

{

bool \*visit = new bool[count1+1];

for(int j=1;j<=count1;j++){ //求各个状态经过‘@’达到的集合,并放入tempV二维QVector容器中

for(int i=1;i<=count1;i++)

visit[i] = false;

getThroughNumV(j, '@', visit);

sort(temp.begin(),temp.end());

tempV.push\_back(temp);

temp.clear();

}

dfa.push\_back(tempV[start-1]); //将DFA初始状态加入dfa容器中

endS.push\_back(find<int>(tempV[start-1], endless)); //判断该终态是不是初始状态

count2++;

int k =0; //用于记录当前处理的状态集合

while(true) //直到没有新的状态产生即退出循环

{

for(int i=0;i<num.size();i++)

{

QVector<int> temp2;

for(int j=0;j<dfa[k].size();j++)

{

temp = getSetThroughNum(dfa[k][j],num[i]);

if (!temp.empty()){

temp2 = vectors\_set\_union(temp,temp2);

temp.clear();

}

}

if(!temp2.empty())

{

bool flag1 = false; //判断经过某个操作数达到的集合是否已经存在

QVector<int> temp3;

for(int j = 0;j<temp2.size();j++){

temp = tempV[temp2[j]-1];

temp3 = vectors\_set\_union(temp,temp3);

temp.clear();

}

sort(temp3.begin(),temp3.end());

for(int j = 0;j<dfa.size();j++)

{

if(dfa[j] == temp3){ //若状态集合已经存在即自己经过num[i]到达自己

e2[k+1][j+1] = num[i];

flag1 = true;

break;

}

}

if(!flag1){ //否则产生新的状态

count2++;

e2[k+1][count2] = num[i];

dfa.push\_back(temp3);

endS.push\_back(find<int>(temp3, endless));

}

}

}

k++;

if(k==count2)

break;

}

}

1. 将DFA化简成最小化的DFA

//求经过某个操作数后到达的状态是哪个(DFA边数组）

int MainWindow::**getStatus**(int vertex, QChar ch)

{

for(int i=1;i<=count2;i++)

{

if(e2[vertex][i] == ch)

{

return i;

}

}

return -1; //-1代表为空

}

//判断某个状态属于哪个划分集合

int MainWindow::**judgeVec**(int vec)

{

for(int i=0;i<statusVec.size();i++)

{

if(find<int>(statusVec[i], vec))

return i;

}

return -1;

}

//合并状态将到达s2状态的改为s1的函数

void MainWindow::**mergeStatus**(int s1, int s2)

{

for(int i=1;i<=count3;i++)

{

if(i==s2)

{

for(int j=0;j<num.size();j++)

DFASmall[i][j] = 0; //0表示该状态是被合并了

continue;

}

for(int j=0;j<num.size();j++)

{

if(DFASmall[i][j]==s2)

{

DFASmall[i][j] = s1;

}

}

}

}

//合并状态，将s状态删除掉

void MainWindow::**deleteStatus**(int s)

{

for(int i=s;i<=count3;i++)

{

for(int j=0;j<num.size();j++)

{

DFASmall[i][j] = DFASmall[i+1][j];

}

}

for(int i=1;i<=count3;i++)

{

for(int j=0;j<num.size();j++)

{

if(DFASmall[i][j]>s)

DFASmall[i][j] -= 1; //将某一行删除时还得将指向后面行的状态数减1，s状态后面的整体向前移动了一行

}

}

count3--;

}

//从DFA转成最小化的DFA

void MainWindow::**changeToSmallDFA**()

{

int length = num.size();//存放操作数的个数

count3 = count2; //未简化之前最小化DFA的状态数为count2

//初始划分，划分为终态和非终态两个集合

QVector<int> fs; //存放终态的顶点

QVector<int> nfs; //存放非终态的顶点

for(int i=0;i<endS.size();i++)

{

if(endS[i]==0)

nfs.push\_back(i+1);

else

fs.push\_back(i+1);

}

for(int i=1;i<=count3;i++) //构建未简化的DFA的状态图

{

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

{

DFASmall[i][j] = getStatus(i,num[j]);

}

}

if(!nfs.empty())

statusVec.push\_back(nfs);

statusVec.push\_back(fs);

QVector<QVector<int>> tempVec; //用于临时存放每次划分集合的结果

bool \*dealFlag = new bool[count2+1]; //记录该顶点是否已被合并了

while(true)

{

int num1 = statusVec.size(); //临时存放划分的个数

for(int i=0;i<=count2;i++)

{

dealFlag[i] = false; //每次判断前先赋值为全部未合并

}

for(int i=0;i<num1;i++) //对现存状态进行划分直到不能再划分则退出循环

{

if(statusVec[i].size()==1)

{

tempVec.push\_back(statusVec[i]);

continue;

}

for(int j=0;j<statusVec[i].size()-1;j++)

{

if(dealFlag[statusVec[i][j]])

continue; //如果已被合并过则直接下一个循环

QVector<int> temp1; //存放属于同一划分的集合

temp1.push\_back(statusVec[i][j]);

for(int k=j+1;k<statusVec[i].size();k++)

{

if(dealFlag[statusVec[i][k]])

continue; //如果已被合并过则直接下一个循环

int t;

for(t = 0;t<num.size();t++) //判断是否相同，相同则划分到同一个集合

{

if(judgeVec(DFASmall[statusVec[i][j]][t]) != judgeVec(DFASmall[statusVec[i][k]][t]))

break;

}

if(t==num.size())

{

dealFlag[statusVec[i][k]] = true;

temp1.push\_back(statusVec[i][k]);

}

}

tempVec.push\_back(temp1);

}

}

statusVec.clear();

while(!tempVec.empty())

{

statusVec.push\_back(tempVec.back());

tempVec.pop\_back();

}

if(num1 == statusVec.size())

break;

}

delete []dealFlag; //删除开辟的空间

for(int i = 0; i < statusVec.size(); i++) //合并相同的状态

{

if(statusVec[i].size() == 1)

continue;

for(int j=1;j<statusVec[i].size();j++)

{

mergeStatus(statusVec[i][0], statusVec[i][j]);

}

}

//将标记为0即被合并的行数移动掉

for(int i=1;i<=count3;i++)

{

for(int j=0;j<num.size();j++)

{

if(DFASmall[i][j]==0)

{

deleteStatus(i);

i=i-1;

break;

}

}

}

}

1. 将最小化的DFA转成词法分析程序

//将最小化的DFA转成词法分析程序

void MainWindow::**changeToCode**()

{

QString code = ""; //存放词法分析程序

code += code + "int status = 1;\nwhile(";

for(int i=1;i<=count3;i++) //每个状态进行选择，不在这些状态内则退出循环

{

if(i == count3)

code = code + "status == " + QString::number(i) + ")\n{";

else

code = code + "status == " + QString::number(i) + " || ";

}

code = code + " switch(status){\n";

for(int i=1;i<=count3;i++) //进行每个状态

{

int k=0; //判断该状态是否没有经过任何操作数

code = code + " case " + QString::number(i) + " :\n";

for(int j = 0; j<num.size(); j++) //判断经过某个状态后到达哪个转台

{

if(DFASmall[i][j] != -1){

code = code + " if input() == " + "'" + num[j] + "'" + ":\n";

code = code + " status = " + QString::number(DFASmall[i][j]) + ";\n";

}

else

k++;

}

if(k!=num.size())

code = code + " else: status = -1; //进入错误状态或其他状态跳出循环\n";

else

code = code + " status = -1; //进入错误状态或其他状态跳出循环\n";

code = code + " break;\n";

}

code = code + " default: break;\n }\n}";

ui->textEdit->setText(code);

}

1. 退出系统

//退出程序

connect(ui->pB\_exit,&QPushButton::clicked,this,[=](){

exit(0);

});