# 華南師範大學

# 本科学生实验（实践）报告

院　　系：

实验课程：

实验项目：

指导老师：

开课时间：　　~　　年度第　学期

专　　业：

班　　级：

华南师范大学教务处

# 华南师范大学实验报告

学生姓名_____学　　号_____

专　　业_____年级、班级_____

课程名称_____实验项目_____

实验类型　　□验证 □设计 □综合　实验时间_____年____月____日

实验指导老师_____实验评分_____

# 一、实验内容

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。

必做实验要求：

（1）正则表达式应该支持单个字符，运算符号有： 连接、选择（|）、闭包（*）、括号（）、可选（?）

（2）要提供一个源程序编辑界面，让用户输入一行（一个）或多行（多个）正则表达式（可保存、打开正则表达式文件）

（3）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）

（4）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）

（5）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）

（6）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/C++语言描述）

（7）用户界面应该是 windows 界面

（9）应该书写完善的软件文档

选做实验要求：

扩充正则表达式的运算符号，如 [ ]、正闭包（+） 等。

# 二、实验目的

（1）掌握正则表达式语法及计算规则；

（2）掌握 NFA 的 Thompson 结构以及由正则表达式转换 NFA 的方法；

（3）掌握如何使用子集构造将 NFA 转换为 DFA；

（4）掌握将 DFA 状态数最小化的方法；

（5）掌握如何用代码实现有穷自动机。

# 三、实验文档

## 1. 总体要求

### 1.1 总体功能要求

程序应具备以下基本功能：

（1）将用户所给的正则表达式（所支持运算有连接、选择、闭包、括号和可选）转换为 NFA；

（2）将 NFA 转换为 DFA；

（3）将得到的 DFA 最小化；

（4）根据最小化 DFA，生成 c++分析程序；

（5）对于上述前三个功能，要求软件能够以状态转换表或图片形式展示转换的结果；软件应可以直接显示分析程序的内容；

（6）系统能够分析一条或多条正则表达式并展示结果，此外，还可保存、打开正则表达式文件。

### 1.2 软件开发要求

（1）开发环境：Windows 系统

（2）开发语言：c++11

## 2. 软件设计

## 2.1 UI 界面设计

根据总体功能要求，该软件的 UI 界面设计如下：

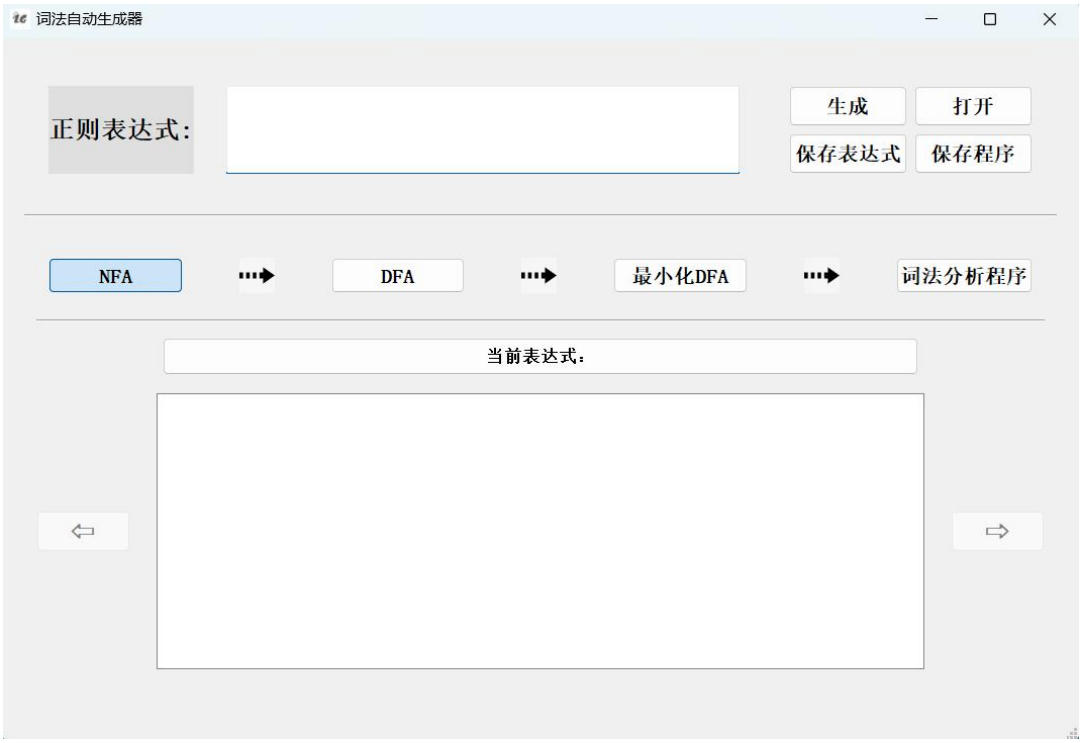



图 1　UI 界面展示

## 2.2 软件功能设计

界面总体可以按垂直方向分成三个部分。

最上方的模块是软件的输入及导出部分。在编辑框中可以直接输入一行或多行正则表达式，也可以点击“打开”按钮导入保存了正则表达式的文本文件。在输入内容后，可以点击“保存表达式”将编辑框内容以文本形式保存。程序分析完成后，点击“保存程序”，可以将 c++词法分析程序保存到本地。点击“生成”即可对表达式进行分析。

中部的模块是显示模式的控制部分。四个按钮分别对应各个转换的分析结果，在点击“生成”后，点击四个按钮可切换转换结果显示。

位于下方的是分析结果显示部分。NFA、DFA 和最小化 DFA 以状态转换表形式在显示框中显示，其中接受状态带有“(+)”后缀，初始状态带有“(-)”

后缀。词法分析程序则是将生成的 c++程序直接显示在编辑框。显示框上方控件对当前展示结果对应的正则表达式进行指示，显示框左右的按钮可以切换查看每行正则表达式的分析结果。

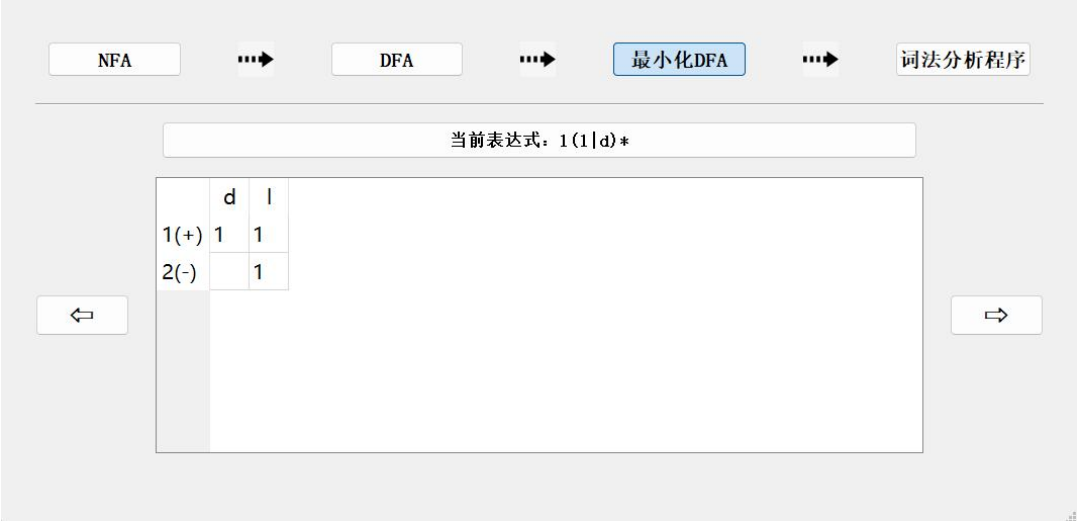



图 2　分析内容展示

除此之外，软件还带有对输入错误正则表达式的提示、打开文件失败的提示、忽略空白行等功能。

# 3. 功能实现及分析

## 3.1 处理正则表达式

对于正则表达式的运算，这里使用了转换后缀表达式再运算的方法。由于正则表达式中连接符号为隐式显示，不利于转换后的计算，这里以“&”表示连接，我们需要先为表达式显式添加“&”符号，如 ab 转换为 a&b。添加方法只需对连接运算可能出现的情况进行枚举即可。

代码：

```
string REGEX::addOP(string regex)
{
```

```
    string res=delEmpBrk(regex);
    int len=res.length();
    for(int i=0;i<len-1;)//主要是情况枚举
    {
        char c1=res[i],c2=res[i+1];
        if(isTrans(c1)||c1==')'||c1=='?'||c1=='*')
        {
            if(isTrans(c2)||c2=='(')
            {
                res=(res.substr(0,i+1)+'&')+res.substr(i+1);
                len=res.length();
                i+=2;
            }
            else
                i++;
        }
        else
            i++;
    }
    return res;
}
```

参考数值计算（+、-、*、/）中缀转后缀的算法，需要借助数据结构栈实现。接下来的关键问题便是如何设计各运算符的优先级。设置栈内优先数 isp 和栈外优先数 icp，在数值计算转换后缀表达式中，优先数设计有以下特点：

①乘、除、取余等在中缀表达式具有更高优先级的符号，其栈内外后缀优先数都大于加、减符号；

②优先数相等的只有这几种情况：

栈内“(”遇到栈外“)”，“)”是不入栈的，并且将使相匹配的“(”出栈，因为 icp[‘)’]的优先级极小，在“(”出栈前这两个半括号间的运算符将全部出栈，体现优先运算括号内容的规则；

栈内“)”遇到栈外“(”，这种情况并不会发生，因为 icp[‘)’]的优先级极小，只有“#”的更小，但“#”是栈中第一个元素或表达式终止符，“)”入栈说明表达式有问题。

③“(”入栈后优先数变极低，方便后续运算符入栈。

④除了括号外，对于同个运算符，都有 isp>icp。

在正则表达式运算中，优先级为 *=? > & > |，根据上述特点，可以推知它们的优先数应有如下关系：

$$(^{(外)},)^{(内)} > *,?^{(内)} > *,?^{(外)} > \&^{(内)} > \&^{(外)} > |^{(内)} > |^{(外)} > (^{(内)},)^{(外)}$$

易知可将优先数设计如下：

| 运算符 | 栈内（isp） | 栈外（icp） |
|---|---|---|
| ( | 0 | 7 |
| ) | 7 | 0 |
| *,? | 6 | 5 |
| & | 4 | 3 |
| \| | 2 | 1 |

后缀表达式转换的步骤如下：

创建空字符串 str，对中缀表达式从左到右检索。遇到字符直接并入 str，若遇到操作符：

①栈空时运到运算符直接入栈；

②当前运算符栈外优先数大于栈顶运算符栈内优先数，当前操作符入栈；

③若小于，栈顶运算符退栈且并入 str；

④若等于，栈顶运算符退栈但不并入，如果退‘(’，读入下一字符；

⑤扫码结束 str 即为后缀表达式。

对此设计一个类，负责为一个中缀正则表达式检错、添加连接符号并转换为后缀表达式。后缀表达式的转换实现见“regex.cpp”文件。

REGEX 类定义：

```
class REGEX
{
public:
    REGEX();
```

```
    static bool isTrans(const char c);//是否为转换

    static bool isOperator(const char c);//是否是合法运算符

    static bool isLegal(const string regex);//简单检测正则表达式是否合法，主要关注字符上的错误，忽略部分语法出现的错误

    static string addOP(string regex);//添加'&'连接符号

    bool postfix(string regex,string &res);//将中缀正则表达式转后缀，res 传参接收结果，返回表达式是否合法的布尔值

private:
    unordered_map<char,int> isp,icp;

    static string delEmpBrk(string regex);//删除空括号

};
```

### 3.2 正则表达式转换 NFA

现在来考虑如何由得到的后缀表达式转换 NFA。首先我们需要有能够保存 NFA 结构的方法。考虑到使用 Thompson 结构构建 NFA，可以按存储图的方法，使用邻接表保存：

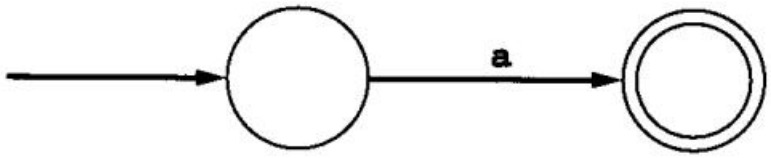
```
struct edge//转换边
{
    char CvType;//转换类型
    int vj;//转换的下一状态
    edge(){CvType='0',vj=0;}
    edge(char c,int v):CvType(c),vj(v){}
    bool operator<(const edge &t)const {if(CvType==t.CvType)return vj<t.vj;  else return CvType<t.CvType;}
    bool operator==(const edge &t)const {return vj==t.vj&&CvType==t.CvType;}
    bool operator!=(const edge &t)const {return vj!=t.vj||CvType!=t.CvType;}
};

vector<set<edge>> graph;//邻接表，从 1 开始
```

其中 vector 的下标表示了一个状态，每个邻接表中存储的是结构体 edge，

表示转换类型和转换后的状态，如 vector[1]里存储了一个 edge(‘a’,2)表示状态 1 能通过 a 转换到状态 2。

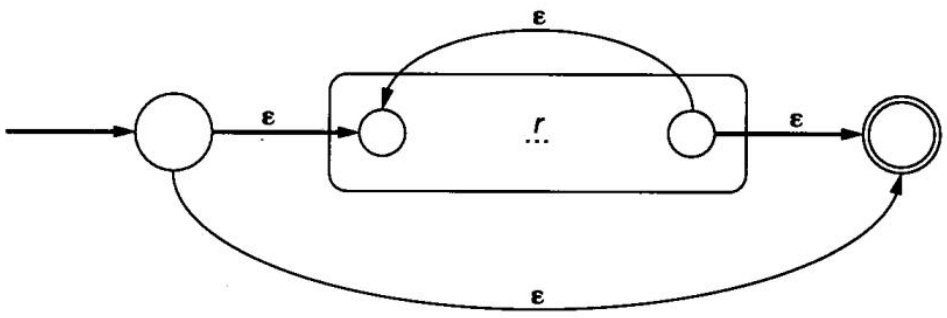
可以存储 NFA 结构后，就是考虑如何转换。NFA 类创建一个初始化方法，接受一个后缀正则表达式，准备一个存储 NFA 结构的<初始状态,接受状态>的栈，然后从左向右扫描，根据遇到的字符或运算符按对应的 Thompson 结构，从栈中取出相应数量 NFA，建立新 NFA，随后压入栈。Thompson 中有如下几种结构：

基本转换：




即遇到单个字符时，我们需要创建两个新状态 start 和 end，在 start 的邻接表中记录转换 edge(‘a’,end)，随后压入栈。

闭包*：




闭包是单目运算符，需要从栈中取一个 NFA，然后建立两个新状态 start，end，按照 Thomson 分别在邻接表中记录新增的 ε -转换，其中 start 是初始状态，end 是接受状态。

对于连接&、选择 | 等双目运算符，我们需要从栈中取出两个 NFA 建立新的 ε -转换，将运算后的<初始状态,接受状态>入栈，这里不再赘述。

扫描完成后，栈中会剩下一个 NFA 结构即最终结果。我们知道，NFA 有

初始状态和接受状态，此时我们如何知道是哪个状态？根据这几个运算符的 Thompson 结构可知，每次运算得到的 NFA 都只有一个初始状态和一个接受状态，也就是说最终的初始状态和接受状态，正好分别对应最后的 NFA 在栈中的存储的值。最后，在计算过程中，可以为类添加一个 set<char>类型成员变量存储出现过的转换类型。

### 3.3 NFA 转换 DFA

首先考虑下 DFA 类如何存储。DFA 和 NFA 结构相似，只是没有 ε -转换，所以可以用同样类型的邻接表和字符表存储。需要注意的是 DFA 可能存在多个接受状态，所以我们需要使用 vector 容器去标注各状态是否是接受状态了。此外，由 NFA 转换的 DFA，其每个状态可能包含 NFA 中的多个状态，所以还需要建立 vector<set<int>>类型由状态编号到元素集合的映射。

使用子集构造的方法来进行 NFA 到 DFA 的转换。简要地说步骤有以下几点：

（1）列出 NFA 各状态的 ε -闭包；

（2）找出 NFA 中的所有非 ε -转换；

（3）以 NFA 初始状态的 ε -闭包作为 DFA 的初始状态，以（2）中找到的规则进行转换，若产生新状态，对新状态继续转换，还要用之前的搜索过的转换对新状态进行转换，直到不再产生新的状态。

（4）若状态存在中有元素是 NFA 中的接受状态，则该状态也是 DFA 的接受状态。

这个方法需要我们获取 NFA 中各状态的 ε -闭包。于是为 NFA 添加成员函数 getClosure 获取 ε -闭包。由于 ε -闭包是某一状态 i 经零个、一个或多个 ε -转换能得到的集合，容易想到使用深度搜索的方法在 NFA 结构中沿着 ε -转换搜索：

```
void NFAGraph::DFS(set<int> &s,vector<bool> &f,int fa)//搜索到的闭包集合，是否访问，父节点编号
```

```
{
    s.insert(fa);
    f[fa]=true;
    for(auto node:graph[fa])
    {
        if(node.CvType=='#')//找到ε转换
            DFS(s, f, node.vj);
    }
}
```

现在我们有ε-闭包了，需要设计一个算法使得 DFA 产生的每个新状态都进行所有转换。可以考虑双重循环：外层是 DFA 的状态集 vector<set<int>>类型成员变量 states，内层是字符集，这样，每次产生的新状态会插入 states 尾部，使得循环继续进行；进入内循环则可以对当前状态遍历所有转换。在这之前，先创建 map<char,map<int,int>>类型字符转换表，方便搜索在某个转换中哪些状态转换到了自身或其他状态。在第 i 次循环中，首先遍历 states[i]中元素，将元素在字符转换表中搜索，看看是否有转换，有则保留转换后对应的ε-闭包集合。将所有这样的闭包集合并运算得到状态，在 states 中搜索该状态是否已存在，若不存在则尾插建立新状态。在这过程中产生了 DFA 中状态的转换，可以顺便建立 DFA 的邻接表。

代码：

```
void DFAGraph::init(NFAGraph &nfa)
{
    isMinimized=false;
    vector<set<int>> closure=nfa.getClosure();//ε-闭包
    map<char,map<int,int>> chtrans;//字符转换表
    set<char> chs=nfa.getCharSet();//获取字符集
    for(auto c:chs)
    {
        if(c!='#')
        {
            cs.insert(c);
            chtrans[c]=nfa.getCharTrans(c);
        }
    }

    if(closure.size()<=1)
        return;
```

```
    states.push_back(set<int>());
    isAccepted.push_back(false);

    //加入初始状态
    int initst=nfa.getInitState();
    states.push_back(closure[initst]);
    graph.push_back(set<edge>());//下标 0 不用
    graph.push_back(set<edge>());//为初始状态创建邻接表
    InitState=states.size()-1;

closure[initst].find(nfa.getAcceptState())!=closure[initst].end()?isAccepted
.push_back(true):isAccepted.push_back(false);
    //开始转换
    for(int i=1;i<states.size();i++)
    {
        for(mcmi mi=chtrans.begin();mi!=chtrans.end();mi++)
        {
            char ch=mi->first;//当前转换字符
            set<int> res;
            for(auto j:states[i])//遍历 i 的状态集合
            {
                map<int,int>::iterator it=chtrans[ch].find(j);
                if(it!=chtrans[ch].end())//当前字符 ch，当前状态 j，若找到转换
的状态，进行集合并

set_union(res.begin(),res.end(),closure[it->second].begin(),closure[it->seco
nd].end(),inserter(res,res.begin()));
            }
            if(res.empty())
                continue;

            int sz=states.size();
            int w=addState(nfa,res);
            if(sz<states.size())//若 size 发生变化，说明产生新状态
                graph.push_back(set<edge> ());
            graph[i].insert(edge(ch,w));
        }
    }
    NumOfStates=states.size()-1;
}
```

### 3.4 最小化 DFA

DFA 的最小化算法虽然容易理解和手算，但是论实现应该是本次实验最难的一个部分，难点主要在于将状态集合分割并管理分割的集合，由于时间紧迫，这里并不打算设计一个效率较高的算法处理，而是直接采取模拟方式。最小化 DFA 结构和 DFA 相同，所以可以用同个类表示，同时 DFA 类中应该添加一个最小化自身的成员函数。

先来研究如何为模拟过程做准备。首先我们需要知道集合最后被如何分割，创建 vector<int>型集合号映射表，它为每个状态赋予一个编号，同个编号的状态表示划分到同个集合中。然后考虑如何持续进行分割知道算法结束。设计一个双端队列存储待分割的集合序列，每次弹出一个集合进行分割，最终需要判断当前集合是否被分割了，若进行了分割，则将多于一个元素的集合继续加在双端队列末端，仅一个元素的集合直接存储在我们的最终结果中；若未被分割，说明集合内各状态是不可分割的，可加入最终结果而无需加入队列。这使得我们的算法能够自动停止（队列为空）并得到正确答案。初始时，将状态按接受状态和非接受状态划分，加入队列。

接下来考虑如何分割。在分割时，一个集合可能被分割为多个集合，用 vector<set<int>>收集分割的集合。由于我们需要对比集合内各状态的转换是否是可区分的，可以用 vector<set<edge>>创建转移集合表。先从邻接表和集合号映射表中获取 edge(转换名,集合号)内容，然后比对与集合内其他状态的转移集合是否相同，若不同，则可区分，将该状态分割到一个新集合中。将具有相同转移集合表的状态收集为各个集合。若一个集合被分割了，我们需要创建新的集合号给集合映射表中被分割出去的状态。

循环结束之后，我们已经可以从集合号映射表中知道哪些状态被划分到同一集合，可以进行合并了。为最小化的 DFA 变量创建集合数量大小的邻接表。首先需要知道哪些合并后哪些集合是接受状态，这只需要从 DFA 的接受状态表中映射到相应集合中，将接受状态所在的集合修改为接受状态即可。其次便是构建最小化 DFA 的邻接表，同样将状态到状态的转换映射为集合到集合的转换即可。以各个集合便为作为最小化 DFA 的各个状态，如此便完成了最小

化 DFA 最困难的部分。

集合的分割实现如下：

```
 vector<int> setNum;//集合号表
setNum.resize(NumOfStates+1);
set<int> k1,k2;
for(int i=1;i<isAccepted.size();i++)
{
    if(isAccepted[i])
    {
        setNum[i]=1;
        k2.insert(i);
    }
    else
    {
        setNum[i]=2;
        k1.insert(i);
    }
}
int mNumStates;//MDFA 状态数
if(k1.size()==0)
    mNumStates=1;
else
    mNumStates=2;

deque<set<int>> dq;//状态集合
if(k1.size()>1)
    dq.push_back(k1);
if(k2.size()>1)
    dq.push_back(k2);

while(!dq.empty())//开始最小化
{
    set<int> s=dq.front();
    dq.pop_front();
    vector<set<edge>> ve;//转移集合表
    vector<set<int>> vs;//分割的集合
    for(auto i:s)//对集合 s 进行分割
    {
        //获取 i 的能转移到的集合号的集合
        set<edge> ste;
        for(int vi=1;vi<graph.size();vi++)
        {
```

```
                for(auto cm:graph[i])//遍历 i 的邻接表
                    ste.insert(edge(cm.CvType,setNum[cm.vj]));//在集合号表中
找到转换后状态所在的集合
            }

            if(vs.size()==0)
            {
                ve.push_back(ste);
                vs.push_back(set<int>());
                vs[0].insert(i);
            }
            else
            {
                bool fequal=false;//转换相同标志
                for(int vi=0;vi<ve.size();vi++)
                {
                    if(transEqual(ve[vi],ste))
                    {
                        vs[vi].insert(i);//转换相同
                        fequal=true;
                        break;
                    }
                }
                if(!fequal)//不同，分割
                {
                    ve.push_back(ste);
                    vs.push_back(set<int>());
                    vs[vs.size()-1].insert(i);
                }
            }
        }

        //清算分割集合 vs
        if(vs.size()>1)
        {
            if(vs[0].size()>1)//集合多于一个元素，需要继续测试是否分割
                dq.push_back(vs[0]);
        }

        for(int j=1;j<vs.size();j++)//为分割出去的集合编号
        {
            if(vs[j].size()>1)
                dq.push_back(vs[j]);
            mNumStates++;
```

```
            for(auto vk:vs[j])//遍历集合内状态
            {
                setNum[vk]=mNumStates;
            }
        }
    }
```

最小化 DFA 成员变量的构建如下：

```
    mdfa.InitState=setNum[InitState];//mdfa 的初始状态是 dfa 初始状态被划分到的集合
    mdfa.NumOfStates=mNumStates;
    mdfa.states.resize(mNumStates+1);
    mdfa.isAccepted.resize(mNumStates+1);
    mdfa.isMinimized=true;
    mdfa.cs=set<char> (cs);
    mdfa.graph.resize(mNumStates+1);
    for(int t=1;t<mdfa.states.size();t++)
        mdfa.states[t].insert(t);

    //合并状态集合
    //接受状态判断
    for(int i=1;i<setNum.size();i++)
    {
        if(isAccepted[i])
            mdfa.isAccepted[setNum[i]]=true;
    }

    //构建最小化 DFA 的邻接表
    for(int vi=1;vi<graph.size();vi++)
    {
        set<edge> ts;//存储状态合并后的集合转换
        for(auto ls:graph[vi])
        {
            int w=ls.vj;
            char ttype=ls.CvType;
            //映射到合并后集合
            w=setNum[w];
            ts.insert({ttype,w});
        }

        //处理 mdfa 的状态转换表
        for(auto ti:ts)
            mdfa.graph[setNum[vi]].insert(ti);
```

```
}
```

### 3.5 DFA 转换 c++分析程序

现在我们的 DFA 采用邻接表形式存储，使用双层 case 方法转换起来较为简单。首先为 c++程序定义一个 state 变量，其初始值是 DFA 的初始状态。最外层是一个 while 循环，根据 DFA，我们应该想到若一个状态能够进行转换则循环应可以继续进行下去，所以需要搜索邻接表中哪些状态存在转换，将这些状态作为循环条件。在循环内部，外层 switch-case 应指向 state 的值，为各个存在转换的状态创建一个 case，default 为 error 或 other 状态，需使 state=-1，使得其能够跳出循环。对于 state=i 的 case，内部设置 switch-case，指向 i 状态的各个转换。可以通过邻接表直接找到 states[i]的转换。以下是生成的一个实例：

```
int state=2;
char trans;
while((state==2)&&cin>>trans)
{
        switch(state)
        {
        case 2:
                switch(trans)
                {
                case 'a':
                        state=2;
                        break;
                case 'b':
                        state=1;
                        break;
                case 'c':
                        state=2;
                        break;
                default:
                        state=-1;
                        break;
                }
                break;
        default:
                state=-1;
                break;
}
if(state==1)
        cout<<"Accept"<<endl;
else
        cout<<"Error"<<endl;
```

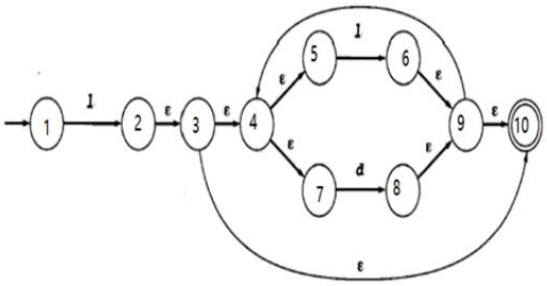
图 3　(a|c)*b 的分析程序

## 4. 软件测试

样例的测试如下。样例正则表达式：l(l|d)*。更多的测试结果见测试报告。样例测试结果：正确。（由于状态编号不同，转换表可能有所差异，但所画结构图是一样的）

## 2.正则表达式→NFA

**输出：如下图的NFA或该图对应的状态转换表**




| | | l | d | # |
|---|---|---|---|---|
| − | 1 | 2 | | |
| | 2 | | | 3 |
| | 3 | | | 4, 10 |
| | 4 | | | 5, 7 |
| | 5 | 6 | | |
| | 6 | | | 9 |
| | 7 | | 8 | |
| | 8 | | | 9 |
| | 9 | | | 4, 10 |
| + | 10 | | | |

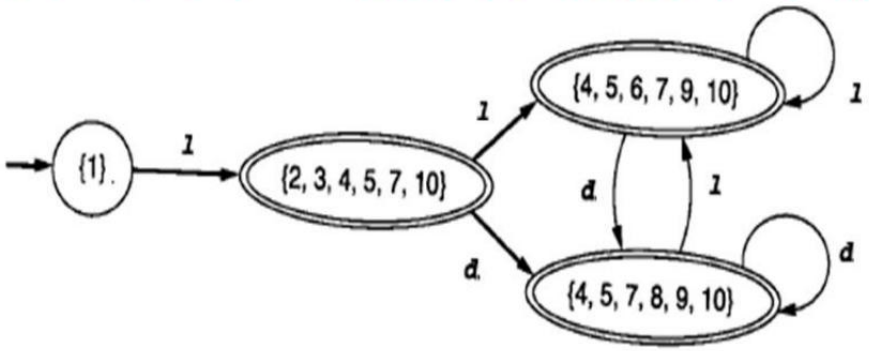
当前表达式：1(1|d)*

| | d | l | ε |
|---|---|---|---|
| 1(-) | | 2 | |
| 2 | | | 9 |
| 3 | | 4 | |
| 4 | | | 8 |
| 5 | 6 | | |
| 6 | | | 8 |
| 7 | | | 3,5 |
| 8 | | | 7,10 |
| 9 | | | 7,10 |
| 10(+) | | | |

图 4、5　NFA 分析结果

## 3.NFA→DFA

**输出：如下图的DFA或对应的状态转换表(如下表)**




| 状态集合 \ 字符 | 1 | d |
|---|---|---|
| { 1 } | { 2, 3, 4, 5, 7, 10 } | |
| { 2, 3, 4, 5, 7, 10 } | { 6, 9, 4, 7, 10, 5 } | { 8, 9, 4, 7, 5, 10 } |
| { 6, 9, 4, 7, 10, 5 } | { 6, 9, 4, 7, 10, 5 } | { 8, 9, 4, 7, 5, 10 } |
| { 8, 9, 4, 7, 5, 10 } | { 6, 9, 4, 7, 10, 5 } | { 8, 9, 4, 7, 5, 10 } |

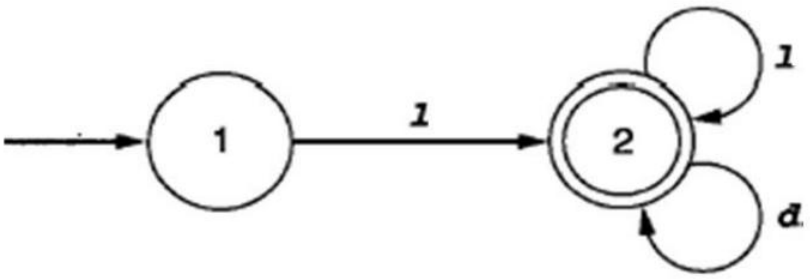
当前表达式：1(1|d)*

| | d | l |
|---|---|---|
| {1}(-) | | {2,3,5,7,9,10} |
| {2,3,5,7,9,10}(+) | {3,5,6,7,8,10} | {3,4,5,7,8,10} |
| {3,5,6,7,8,10}(+) | {3,5,6,7,8,10} | {3,4,5,7,8,10} |
| {3,4,5,7,8,10}(+) | {3,5,6,7,8,10} | {3,4,5,7,8,10} |

图 6、7　DFA 分析结果

## 4.DFA最小化

**输出：如下图的最小化DFA或对应的状态转换表**




| | | l | d |
|---|---|---|---|
| − | 1 | 2 | |
| + | 2 | 2 | 2 |

当前表达式：1(1|d)*

| | d | l |
|---|---|---|
| 1(+) | 1 | 1 |
| 2(-) | | 1 |

图 8、9　最小化 DFA 分析结果

```
int state=2;
char trans;
while((state==1||state==2)&&cin>>trans)
{
        switch(state)
        {
        case 1:
                switch(trans)
                {
                case 'd':
                        state=1;
                        break;
                case 'l':
                        state=1;
                        break;
                default:
                        state=-1;
                        break;
                }
                break;
        case 2:
                switch(trans)
                {
                case 'l':
                        state=1;
                        break;
                default:
                        state=-1;
                        break;
                }
                break;
        default:
                state=-1;
                break;
}
if(state==1)
        cout<<"Accept"<<endl;
else
        cout<<"Error"<<endl;
```

图 10　词法分析程序结果

## 5. 改进方向及可行性分析

### 5.1 更多运算符

由于时间原因，本次我并未完成选做内容。其中有两个运算符[]和+。但是这是可以实现的。首先，两个运算符有对应的 Thompson 结构，只需先处理这两个运算符和其他运算符的优先级关系，其中[]可以转换为选择符号|，如[A-C]转换为 A|B|C，根据优先级关系制定优先数，再在 NFA 类初始化中加入相应的 Thompson 结构构造方法即可。

### 5.2 图像可视化

本人曾经做个一个使用 c++对 dot 语言进行封装的软件，而 dot 语言的使用也十分简单：只需要输入始点名、终点名、边的注解和点的样式即可绘制出 NFA 和 DFA 的结构图。这些信息在设计的类中可以通过邻接表直接获取，只需要调用封装后的接口，即可实现对 NFA 和 DFA 图的显示。

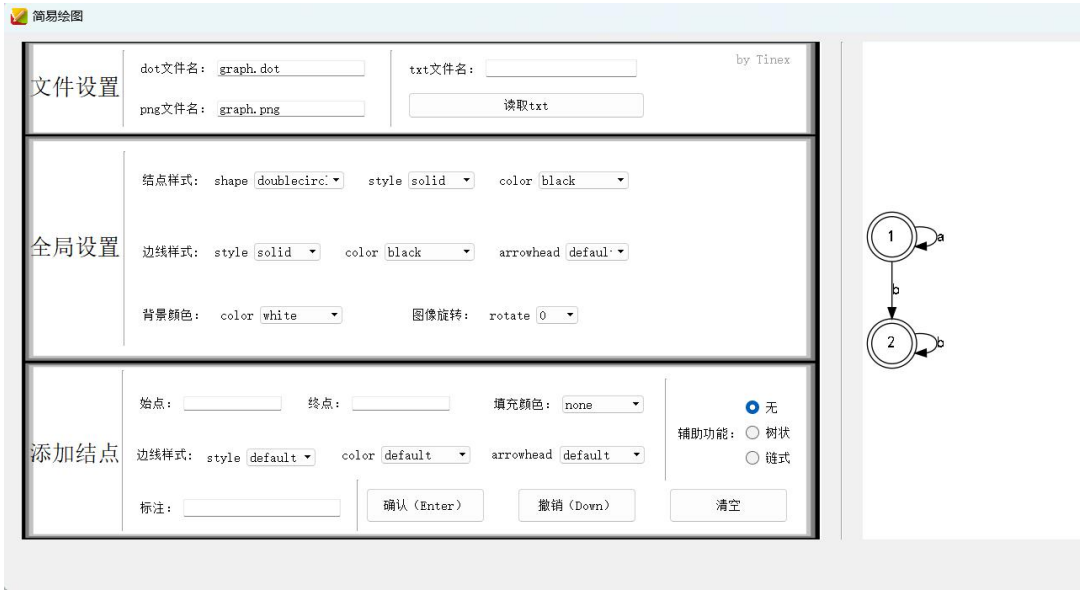



图 11　dot 封装软件绘制正则表达式 a*b*

## 四、实验总结（心得体会）

这次实验是一个难度和工作量都较大的实验，其中难点主要集中在 NFA

转 DFA 和最小化 DFA，其中出现过各种各样的 bug，如今都已基本解决了。在解决问题的过程中，对各种转换方法的理解更加深刻。

## 五、参考文献

Kenneth C.Louden．编译原理及实践[M]．机械工业出版社 ．冯博琴译 ．2004 年 2 月 1 日