编译原理实验报告

**词法分析程序**



|  |  |  |
| --- | --- | --- |
| 姓名 | : | 牟鑫一 |
| 学号 | : | 20161001764 |
| 班级 | : | 191174 |
| 指导老师 | : | 刘远兴 |

目 录

[一、 题目 1](#_Toc25019277)

[二、 问题描述 1](#_Toc25019278)

[三、 基本要求 1](#_Toc25019279)

[四、 小组分工 1](#_Toc25019280)

[五、 整体设计 2](#_Toc25019281)

[1、 正则规则和正则表达式 2](#_Toc25019282)

[2、 NFA的存储结构设计 2](#_Toc25019283)

[3、 正规式转化NFA 3](#_Toc25019284)

[4、 数据结构 4](#_Toc25019285)

[5、 文件结构 7](#_Toc25019286)

[6、 基本思想 7](#_Toc25019287)

[六、 自己负责的模块设计 8](#_Toc25019288)

[1、 函数调用关系图 8](#_Toc25019289)

[2、 模块结构说明 8](#_Toc25019290)

[3、 函数的功能实现 8](#_Toc25019291)

[七、 算法设计 9](#_Toc25019292)

[八、 调试分析 12](#_Toc25019293)

[1、 优点分析 12](#_Toc25019294)

[2、 缺点分析 12](#_Toc25019295)

[3、 改进方法 12](#_Toc25019296)

[九、 使用手册 12](#_Toc25019297)

[十、 测试结果 13](#_Toc25019298)

[十一、 总结 13](#_Toc25019299)

# 题目

读入一个正则表达式，实现正则表达式转NFA，NFA转DFA，以及DFA的最小化

# 问题描述

传入一个正则表达式，构造一个与之等价的NFA，实现词法分析程序

# 基本要求

* 使用正规式等技术实现一个词法分析程序；
* 使用算符优先分析方法实现其语法分析程序；
* 在语法分析过程中同时完成常量表达式的计算。

# 小组分工

* 牟鑫一：正则式转NFA，输入一个正规表达式，输出与该文法等价的有限自动机；
* 江佳盛、刘栋阳：有限自动机的确定化，将一个非确定有限自动机转换为确定有限自动机；
* 郭兴：确定有限自动机的最小化，根据一个正规表达式构造最少状态的确定有限自动机；
* 李泽栋：模拟DFA，模拟最少状态的确定有限自动机，判断输入串能否识别；
* 梅涵：程序框架和界面设计，编写整个程序的框架、界面设计、数据结构设计等。

# 整体设计

* 1. 正则规则和正则表达式
* 数据结构：正则规则使用RegularRule类来存储，正则表达式为一个string。
* 正则规则RegularRule：
  + alphabet:map<char, set<char>>，存储主字符表，key为某类字符的代表符号，value是这个代表符号代表的所有字符的集合。这个属性提供给用户自定义的方法，用户可以随意更改正则规则。
  + assistAlpha:set<char>，存储辅助字符，实际上只有‘\*’，‘|’，‘（’，‘）’四个符号。这个属性用户无法修改，因为所有的正则文法都可以用这四个辅助字符来表示。
* bool check(string )：用于检查一个字符串是否是符合这种正则文法的正则表达式。大体算法如下：首先判断表达式中的各个辅助符号是否符合限制要求，然后遍历各字符是否是设定好的集合中的字符，一条不满足就返回false，全部满足返回true。
  1. NFA的存储结构设计

1. 原则：使用节点序号来唯一标识一个节点，使用类似邻接表但做了一定修改的形式来存储。（详见UML静态类图）
2. Node类：在主表中存储NFA的各个节点，包含节点序号和邻接表地址两个属性。
3. AdjoinNode类：邻接表节点类，含有该节点的序号，由主节点转换为该节点的条件，以及下一个邻接节点的地址三个属性组成。
4. nodeMap:map<int,Node\*>，NFA节点主表，使用哈希表来无序存储各个节点，key为节点序号，value为该节点对应的Node对象。
5. sIndex:int，存储初态节点序号，初始化为0。
6. zIndex:int，存储终态节点序号，初始化为1。
7. regularRule:RegularRule，存储该NFA遵守的自定义的正则规则。
8. regularExpr:string，存储该NFA对应的正则表达式。
   1. 正规式转化NFA
9. 核心：把正则表达式分成一个一个的子模块，然后依据各个辅助符号进行模块间的连接，最终形成一个完整的NFA。
10. 数据结构：moduleStack：这是一个栈，每一个节点有两个属性，为该模块的初态节点和终态节点。assistStack：这是储存辅助符号的栈，但只负责存储’(’，’|’，’.’三个符号，‘\*’会单独处理，‘)’是出栈的信号。
11. 具体流程：
    1. 检验正则表达式是否符合正则规则，否则返回false。
    2. 把0和1的初始初态终态压入moduleStack。
    3. 在已知正则表达式的首尾添加一对括号。
    4. 把已知正则表达式中省略的连接符号添加上。
    5. 开始进行输入：
    6. If 检测到输入为(‘(’ || ‘|’ || ‘.’)，就把这些符号压入assistStack。
    7. If 检测到输入为主字符，就给这个字符构造初态终态，把新添加的节点和跃迁关系添加进nodeMap，并把这个字符的初态终态作为一个子模块压入moduleStack。
    8. If 检测到‘\*’，就把moduleStack的栈顶的子模块添加两个节点，进行闭包操作，把新添加的节点和关系存储进nodeMap。之后把原栈顶的模块替换成新生成的模块。
    9. If 检测到‘）’，就在assistStack中进行出栈操作。如果出栈元素为‘|’，就把module栈中的栈顶的两个模块取出来，按照‘|’的结合方式合成一个新的子模块，把添加的节点和关系加入nodeMap，并把这个新的子模块再存入moduleStack中，然后继续出栈；出栈元素是‘.’，那么操作和‘|’类似，只不过更改了两个模块的结合方式；如果出栈元素是‘（’，就停止出栈。
    10. 一直到输入结束，moduleStack中只有一个模块了，那么它存储的就是最终NFA的初态和终态序号，把最终的终态序号存入zIndex中，NFA构造完毕。
    11. 数据结构
12. RegularRule类：

class RegularRule

{

private:

//字母表（key是表中包含的字母，value是该字母对应的所有匹配字符），可用户自定义

map<char, set<char>> \*\_alphabet;

//辅助字母表，包含所有的辅助字符，不可用户自定义

set<char> assistAlpha;

public:

RegularRule(map<char, set<char>>\*);//根据一个字母表构建正则表达式

~RegularRule();

map<char, set<char>>\* getAlphabet();//获取字母表

void setAlphabet(map<char, set<char>>\*);//设置字母表

bool check(string);//检查一个字符串是否为一个正则表达式

};

1. Node类

//NFA节点类

class Node

{

private:

//邻接表首节点地址

AdjoinNode \*adjoinTable;

public:

//节点序号

int index;

Node(int);

//设置邻接表首节点

void setAdjoinTable(AdjoinNode\*);

//获取邻接表首节点

AdjoinNode\* getAdjoinTable();

//添加邻接点

void addAdjoinNode(AdjoinNode\*);

};

1. AdjoinNode类

//邻接表节点类

class AdjoinNode

{

private:

AdjoinNode \*\_next;

public:

//节点序号

int index;

//节点跃迁的条件

char condition;

AdjoinNode(int, char);

//获取下一个节点

AdjoinNode\* next();

//设置下一个节点

void setNext(AdjoinNode\*);

};

1. AssistStackNode类

//正则表达式转化NFA辅助符号存储栈节点

class AssistStackNode

{

public:

AssistStackNode(char, AssistStackNode\*);

char symbol;

AssistStackNode \*next;

};

1. AssistStack类

//正则表达式转化NFA辅助符号存储栈

class AssistStack

{

public:

AssistStackNode \*top;

AssistStack();

void push(AssistStackNode\*);

AssistStackNode\* pop();

};

1. NFA类

//NFA类

class NFA

{

private:

//节点表

//空串使用'c'表示

map<int, Node\*> nodeMap;

//初态节点序号

int sIndex;

//终态节点序号

int zIndex;

//遵守的正则规则

RegularRule \*regularRule;

//对应的正规式

string regularExpr;

public:

NFA(RegularRule \*regularRule, string regularExpr);

~NFA();

//已有正则规则和正则表达式的情况下，构造NFA图

bool createNFA();

RegularRule\* getRegularRule();

void setRegularRule(RegularRule\*);

string getRegularExpr();

void setRegularExpr(string);

map<int, Node\*> getNodeMap();

int getSIndex();

int getZIndex();

//测试方法：遍历整个节点表并打印

void testNFA();

};

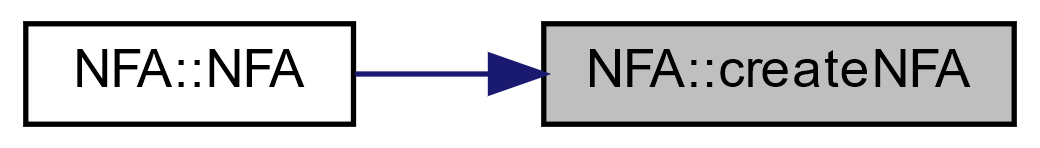
* 1. 文件结构
* RegularRule.h //正则表达式规则类
* RegularRule.cpp
* Node.h //NFA节点类
* Node.cpp
* AdjoinNode.h //邻接表节点类
* AdjoinNode.cpp
* AssistStackNode.h //正则表达式转化NFA辅助符号存储栈节点
* AssistStackNode.cpp
* AssistStack.h //正则表达式转化NFA辅助符号存储栈
* AssistStack.cpp
* NFA.h //NFA类
* NFA.cpp
  1. 基本思想

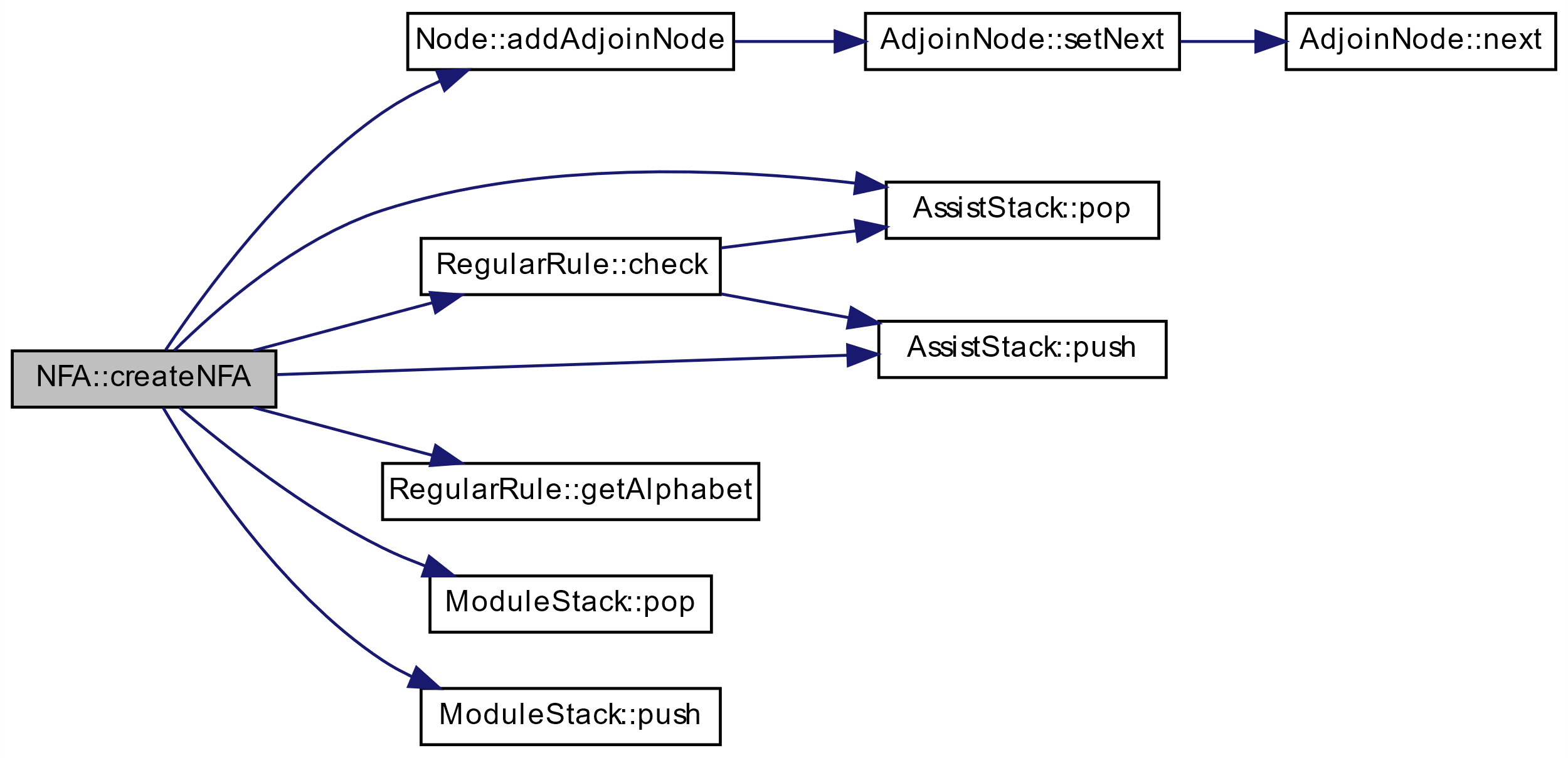
初始的初态终态压入栈moduleStack，给正则式首尾加上括号并添加连接符号，例如输入正则式为：a(a|b)\*a，经过这一步操作，正则式变为：(a.(a|b)\*.a)。然后循环遍历正则式字符串，如果是字符 '(' 、 '|' 或 '.' ，则将其压入辅助符号存储栈；如果是符号 '\*' ，模块状态出栈，加入新的节点并插入节点表，为新的节点构建邻接表，最后将新模块的初态和终态入栈；如果是 ')' ，将符号栈内元素储栈，如果出战符号是 '.' 则进行连接操作，如果出栈符号是'|'就进行或操作，直至遇到左括号 '(' 跳出内循环；如果是一般符号，则将新元素加入节点表，连接并入栈。最后连接moduleStack里得最后两个元素。

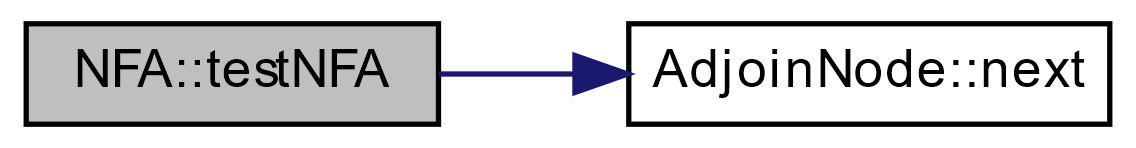
# 自己负责的模块设计

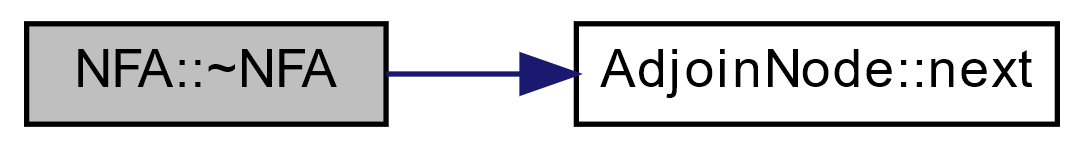
我负责的模块是：正则式到NFA的转换

* 1. 函数调用关系图









* 1. 模块结构说明

//构造函数

NFA(RegularRule \*regularRule, string regularExpr);

//析构函数

~NFA();

//已有正则规则和正则表达式的情况下，构造NFA图

bool createNFA();

* 1. 函数的功能实现

//获取正则规则

RegularRule\* getRegularRule();

//设置正则规则

void setRegularRule(RegularRule\*);

//获取正则表达式

string getRegularExpr();

//设置正则表达式

void setRegularExpr(string);

//获取图结点

map<int, Node\*> getNodeMap();

//获取初态结点序号

int getSIndex();

//获取终态结点序号

int getZIndex();

//测试方法：遍历整个节点表并打印

void testNFA();

# 算法设计

正则式转NFA的关键函数：

bool NFA::createNFA()

{

if (!regularRule->check(regularExpr)) return false;

AssistStack assistStack;

ModuleStack moduleStack;

//初始的初态终态压入moduleStack

moduleStack.push(new ModuleStackNode(sIndex, zIndex, moduleStack.top));

//正则式首尾加括号

regularExpr.insert(0,1,'(');

regularExpr.append(")");

//正则式添加连接符号

for (int i = 0; i < regularExpr.length() - 1; i++) {

if ((regularRule->getAlphabet()->count(regularExpr[i]) != 0 || regularExpr[i] == '\*' || regularExpr[i] == ')')

&& (regularRule->getAlphabet()->count(regularExpr[i + 1]) != 0 || regularExpr[i + 1] == '(')) {

regularExpr.insert(i + 1, 1, '.');

}

}

//开始输入

char x;//当前字符

int n = 0;//当前的最高节点序号

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

x = regularExpr[i];

if (x == '(' || x == '|' || x == '.') assistStack.push(new AssistStackNode(x, assistStack.top));

else if (x == '\*') {

ModuleStackNode \*node = moduleStack.pop();//先出栈

//加入新的节点并插入节点表

Node \*newS = new Node(++n);

Node \*newZ = new Node(++n);

nodeMap.insert(pair<int, Node\*>(newS->index, newS));

nodeMap.insert(pair<int, Node\*>(newZ->index, newZ));

//为新的节点构建邻接表

newS->addAdjoinNode(new AdjoinNode(node->sIndex, 'c'));

newS->addAdjoinNode(new AdjoinNode(newZ->index, 'c'));

nodeMap[node->zIndex]->addAdjoinNode(new AdjoinNode(newZ->index, 'c'));

nodeMap[node->zIndex]->addAdjoinNode(new AdjoinNode(newS->index, 'c'));

//把新模块的初态和终态入栈

delete(node);

moduleStack.push(new ModuleStackNode(newS->index, newZ->index, moduleStack.top));

}

else if (x == ')') {

AssistStackNode \*node = NULL;//从符号栈pop出来的node

while (true) {

node = assistStack.pop();

//出栈元素为'('就停止出栈

if (node->symbol == '(') break;

//出栈元素为'.'就进行连接操作

if (node->symbol == '.') {

//先从ModuleStack里把顶两个元素出栈

ModuleStackNode \*node1 = moduleStack.pop();

ModuleStackNode \*node2 = moduleStack.pop();

//连接操作

nodeMap[node2->zIndex]->addAdjoinNode(new AdjoinNode(node1->sIndex, 'c'));

//把新模块的初态和终态入栈

moduleStack.push(new ModuleStackNode(node2->sIndex, node1->zIndex, moduleStack.top));

delete(node1);

delete(node2);

}

//出栈元素为'|'就进行或操作

if (node->symbol == '|') {

//先从ModuleStack里把顶两个元素出栈

ModuleStackNode \*node1 = moduleStack.pop();

ModuleStackNode \*node2 = moduleStack.pop();

//加入新的节点并插入节点表

Node \*newS = new Node(++n);

Node \*newZ = new Node(++n);

nodeMap.insert(pair<int, Node\*>(newS->index, newS));

nodeMap.insert(pair<int, Node\*>(newZ->index, newZ));

//或操作

newS->addAdjoinNode(new AdjoinNode(node1->sIndex, 'c'));

newS->addAdjoinNode(new AdjoinNode(node2->sIndex, 'c'));

nodeMap[node1->zIndex]->addAdjoinNode(new AdjoinNode(newZ->index, 'c'));

nodeMap[node2->zIndex]->addAdjoinNode(new AdjoinNode(newZ->index, 'c'));

//把新模块的初态和终态入栈

moduleStack.push(new ModuleStackNode(newS->index, newZ->index, moduleStack.top));

delete(node1);

delete(node2);

}

}

}

else {

//加入新的节点并插入节点表

Node \*newS = new Node(++n);

Node \*newZ = new Node(++n);

nodeMap.insert(pair<int, Node\*>(newS->index, newS));

nodeMap.insert(pair<int, Node\*>(newZ->index, newZ));

//连接

newS->addAdjoinNode(new AdjoinNode(newZ->index, x));

//入栈

moduleStack.push(new ModuleStackNode(newS->index, newZ->index, moduleStack.top));

}

}

//连接moduleStack里最后两个元素

//先从ModuleStack里把顶两个元素出栈

ModuleStackNode \*node1 = moduleStack.pop();

ModuleStackNode \*node2 = moduleStack.pop();

//连接操作

nodeMap[node2->zIndex]->addAdjoinNode(new AdjoinNode(node1->sIndex, 'c'));

//得到最终的NFA终态赋值

zIndex = node1->zIndex;

delete(node1);

delete(node2);

return true;

}

# 调试分析

* 1. 优点分析
* 模块化设计，便于功能得扩展以及函数得复用；
* 分别处理正则表达式得普通字符和特殊字符，用栈存储便于左右括号得匹配。
  1. 缺点分析
* 对正则式分析的情况不够全面，例如不能识别转义符号，存在普通字符 ’\*’ 和克林闭包 ’\*’ 无法区分的问题；
* 正则式转NFA的主函数 NFA::createNFA() 还是过于复杂，情况多样，导致函数内部多层嵌套循环，代码可读性较差。
  1. 改进方法
* 综合考虑更广泛全面的正则式情况，以编写出适用于所有形式正则表达式的NFA转换函数；
* 可以将部分较为复杂的函数继续细分为更小更轻量的多个函数，通过函数调用实现同样的功能，这样可以提高代码的可读性，也便于代码调试。

# 使用手册

该模快实现正则表达式到NFA的转换，可通过函数 NFA::testNFA() 进行测试。

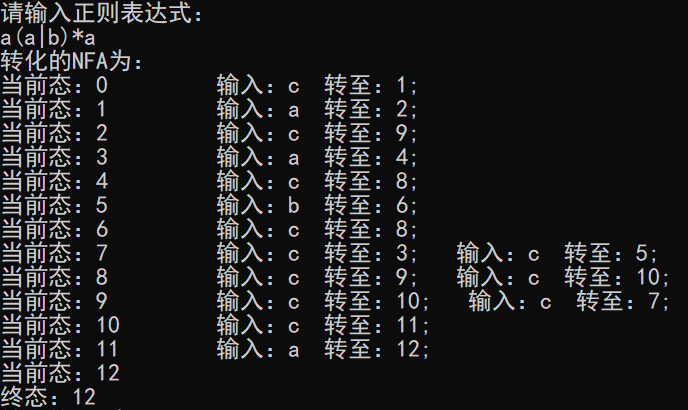
模块通过NFA构造函数传入正则规则和正则表达式，初始化空NFA，然后调用函数 NFA::createNFA() 构造NFA，转换得到的NFA存入NFA类nodeMap，可通过迭代NFA对象读取NFA状态。

调用函数 NFA::testNFA() 可读取并输出NFA的状态以及对应于某一个输入而转移到的下一状态。

# 测试结果

输入正则表达式：a(a|b)\*a

执行结果如下：



# 总结

学习编程以来一直都是自己写代码，没有过团队合作的机会，这一次小组合作完成课程设计还是遇到不少问题，缺乏沟通是关键，另一方面沟通也有些困难，毕竟大家水平参差不齐，编程风格也不尽相同，实在难以统一，也就难免出现大家写的模块不能适用的情况。这也体现了我们的编程学习和真正工作的不同之处，所以要尽量多找机会参与到项目中去锻炼自己的编程能力，学会和他人合作。

通过这个题目也大致的了解到了编译程序的工作原理，不仅仅是尝试了从正则表达式到NFA的转换，也对其它编译过程如词法分析、语法分析、语义分析有了一些简单的思路，对编译器的工作原理有了更加深入的了解，也有利于往后对编程中一些错误或警告的理解分析。