

**杭州电子科技大学**

**《编译原理课程实践》**

**实验报告**

题 目： 算法实现

学 院： 卓越学院

专 业： 智能计算与数据科学

班 级： 21184211

学 号： 21081407

姓 名： 操丰毅

完成日期： 2023.11.25

1. **实验目的**

实现理论课程中的各种算法

1. **实验内容与实验要求**

实现一下几个算法：

* + - 1. 从正规表达式转化为NFA
      2. 将NFA转化为DFA
      3. 求解生成式的First集
      4. 求解生成式的Fellow集
      5. 根据求解的First和Fellow集合来得到LL(1)预测分析表

1. **设计方案与算法描述**
2. 正规表达式转化为NFA

算法流程：

1. 将正规表达式特殊处理，转化为带连接符的字符串
2. 将处理过的字符串转化为后缀表达式
3. 将后缀表达式转化为NFA

算法细节：

* + - 1. 在转化为带连接符的字符串的时候需要判断是否需要加入+

bool isOperator(char c){  
 if(c == '\*' || c == '(' || c == ')' || c == '+' || c == '|'){  
 return true;  
 }  
 else  
 return false;

* + - 1. 然后我们将格式化后的表达式转化为后缀表达式，主要是通过栈来实现。主要有以下的思路：
      2. 如果是“（”就加入栈
      3. 如果是“）”就一直将栈顶的字符加入到最终的子符直到匹配上一个括号
      4. 如果是运算符号，判断与栈顶的运算符的优先级来判断谁加入最终的字符串
      5. 如果是字母，就直接加入到最终的字符串

下面是一个优先级的判断函数

//计算运算符的优先级  
int Priority(char c){  
 map<char,int> m;  
 m['\*'] = 3;  
 m['|'] = 1;  
 m['+'] = 2;  
 return m[c];  
}

* + - 1. 通过后缀表达式来构造NFA，首先我们考虑到NFA的三种运算方式，\*，｜和我们自己写的+连接运算符号，因此我们构建NFA的类之后，对这三个运算符号进行重载，进行运算的时候我们也是通过栈来构建一个NFA的栈，如果是一个字母就构建一个简单的字母NFA，对于运算符我们采取以下的思路来处理：

如果是｜或者+运算符，取出栈顶的两个NFA来做｜或者+运算得到一个新的NFA来加入栈中

如果是\*闭包运算符，那么就取栈顶的一个NFA做\*运算后加入到栈中

* + - 1. 输出NFA的形式，同时输出开始和结束状态

1. NFA转化为DFA

算法流程主要是根据以下的伪代码来实现：

// 初始时，e-closure(s0)作为Dstates中的第一个状态，即初始状态；  
将e-closure(s0)添加到Dstates集合中；  
while (Dstates中还有一个未标记的状态T) {  
 将T标记为已处理；  
 for (对于输入符号集合中的每个符号a) {  
 U = e-closure(move(T, a));  
 if (U不在Dstates中) {  
 将U添加到Dstates中，且未标记；  
 }  
 Dtran[T, a] = U;  
 }  
}

所以我们还需要构建两个辅助函数，move函数和e-closure函数

以下是move函数的实现

set<int> move(set<int> v, string s){  
 set<int> ans;  
 for(auto &i : v){  
 for(auto &edge : Graph[i]){  
 if(edge.c == s){  
 ans.insert(edge.to);  
 }  
 }  
 }  
 return ans;  
}

以下是e\_closure

set<int> e\_closure(set<int> states){  
 queue<int> e\_queue;  
 set<int> closure;  
 for(auto &state : states){  
 e\_queue.push(state);  
 closure.insert(state);  
 }  
 while (!e\_queue.empty()){  
 int node = e\_queue.front();  
 e\_queue.pop();  
 for(auto &edge :Graph[node]){  
 if(edge.c == "\u03B5"){  
 e\_queue.push(edge.to);  
 closure.insert(edge.to);  
 }  
 }  
 }  
 return closure;  
}

根据伪代码来实现从DFA到NFA的转化，简单来说，算法过程是从起始状态开始，然后通过每一个符号的进行状态转化，得到其他的状态，需要注意的是，包含NFA的最后状态的状态为DFA的接受状态，因为DFA可能有多个接受状态，所以需要标记。

1. 根据生成式得到First集

构建first集的方法按照书本上的方法遍历所有的产生式，直到得到一个不包含ε的first集，如果所有的都包含ε的话，就把ε加入到first集中。  
① 如果X属于终结符VT，则FIRST(X) = { X } 。  
② 如果X属于非终结符VN，且有产生式形如X → a…，则FIRST( X ) = { a }。  
③ 如果X属于非终结符VN，且有产生式形如X → ABCdEF…（A、B、C均属于非终结符且包含 ε，d为终结符），需要把d、FIRST( A )、FIRST( B )、FIRST( C )加入到FIRST( X)中。

④ 如果X经过一步或多步推导出空字符ε，将ε加入FIRST( X )。

不断的重复上面的步骤来得到first集

处理单个字母代码

//处理单个字符  
set<string> getFirst(char symbol) {  
 //小写字母终结符号加入  
 string s(1,symbol) ;//转化为string  
 if (!isupper(symbol)) {  
 set<string> c;  
 c.insert(s);  
 return c;  
 }  
  
 return first[s];  
}

处理字符串的代码

//处理字符串  
set<string> getFirstString(const string& str) {  
 set<string> result;  
 for (char symbol : str) {  
 set<string> symbolFirst = getFirst(symbol);  
 result.insert(symbolFirst.begin(), symbolFirst.end());  
 //如果找到了小写字母就退出  
 if(symbolFirst.size() == 1)break;  
 //特殊处理e,不包含e就退出  
 if (symbolFirst.find("ε") == symbolFirst.end()) {  
 break;  
 }  
 }  
 return result;  
}

得到First集

void calculateFirst() {  
 bool changed = true;  
 while (changed) {  
 changed = false;  
 for (const auto& prod : productions) {  
 for (const auto& str : prod.second) {  
 size\_t beforeSize = first[prod.first].size();  
 if(str == "ε"){  
 first[prod.first].insert(str);  
 }  
 else {  
 set<string> strFirst = getFirstString(str);  
 first[prod.first].insert(strFirst.begin(), strFirst.end());  
 }  
 if (first[prod.first].size() != beforeSize) {  
 changed = true;  
 }  
 }  
 }  
 }  
}

直到所有的first集不改变退出迭代

1. 根据生成式得到Fellow集

算法过程，主要有以下步骤：

1. 将#加入到FOLLOW(S)中，S为文法开始符号。
2. 遍寻所有产生式，找到所有类似A->…Ba…的产生式，将a加入到FOLLOW(B)中
3. 遍寻所有产生式，找到所有两个类似A->…BC…的产生式，BC都为非终结符
4. 然后我们将FOLLOW(A)加入到FOLLOW(B)中，然后我们将FOLLOW(A)加入到FOLLOW(B)中。
5. 遍寻所有产生式，找到所有类似A->…BC的产生式，如果FIRST(E)中含有空串，就将FOLLOW(A)加入到FOLLOW(D)中。步骤5其实是步骤4的特殊情况。

需要注意的是，当所有的产生式都能ε的时候才将FOLLOW（A）加入到FOLLOW（B）中。

下面是实现的代码：

void calculateFollow() {  
 follow["S"].insert("$"); //假设S是第一个推导的  
 bool changed = true;  
 while (changed){  
 changed = false;  
 for (const auto& prod : productions) {  
 for (const auto& str : prod.second) {  
 for (size\_t i = 0; i < str.size(); ++i) {//遍历字符串  
 if (isupper(str[i])) {//非终结符号  
 //如果后面还有其他符号，First集（去除e）加入  
 //如果是最后一个，就将左侧的Follow加入到右侧  
 string s(1,str[i]);//转化为string  
 size\_t beforeSize = follow[s].size();  
 set<string> nextFirst = (i + 1 < str.size()) ? getFirstString(str.substr(i + 1)) : follow[prod.first];  
 if (nextFirst.count("ε") == str.size() - 1 - i) {  
 nextFirst.erase("ε");  
 nextFirst.insert(follow[prod.first].begin(), follow[prod.first].end());  
 }  
 follow[s].insert(nextFirst.begin(), nextFirst.end());  
 //检查Follow是否更新  
 if (follow[s].size() != beforeSize) {  
 changed = true;  
 }  
 }  
 }  
 }  
 }  
 }  
}

1. 根据First集和Fellow集来得到LL(1)预测分析表

算法过程：

（1）对每个终结符a∈FIRST（a），将A->a加到M[A,a]中。

（2）如果ε∈FIRST（a），则对于任何b∈FOLLOW（A），将A->a加到M[A,b]中。

（3）表格填完后仍然空缺的位置表明没有对应的产生式，即：非终结符号U+输入符号a无法继续向下推导，所以将空缺的位置填写错误信息。

因此通过first集和follow集就可以得到ll分析表

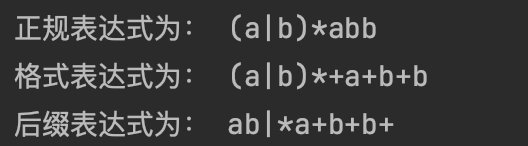
下面是具体的代码实现

void CreateLL1Table() {  
 map<pair<string , string>, string> ll1Table;  
 for (const auto& prod : productions) {  
 string nonTerminal = prod.first;//左侧非终结符号  
 for (const auto& alpha : prod.second) {  
 set<string> firstAlpha = first[nonTerminal];  
 //set<string> firstAlpha = getFirstString(alpha);//求first集  
 for (string terminal : firstAlpha) {//终结符号  
 if (terminal != "ε") {  
 auto table = make\_pair(nonTerminal, terminal);  
 ll1Table[table] = alpha;  
 } else {//如果是e，用Follow集  
 set<string> followNonTerminal = follow[nonTerminal];  
 for (string followTerminal : followNonTerminal) {  
 auto table = make\_pair(nonTerminal, followTerminal);  
 ll1Table[table] = alpha;  
 }  
 }  
 }  
 }  
 }

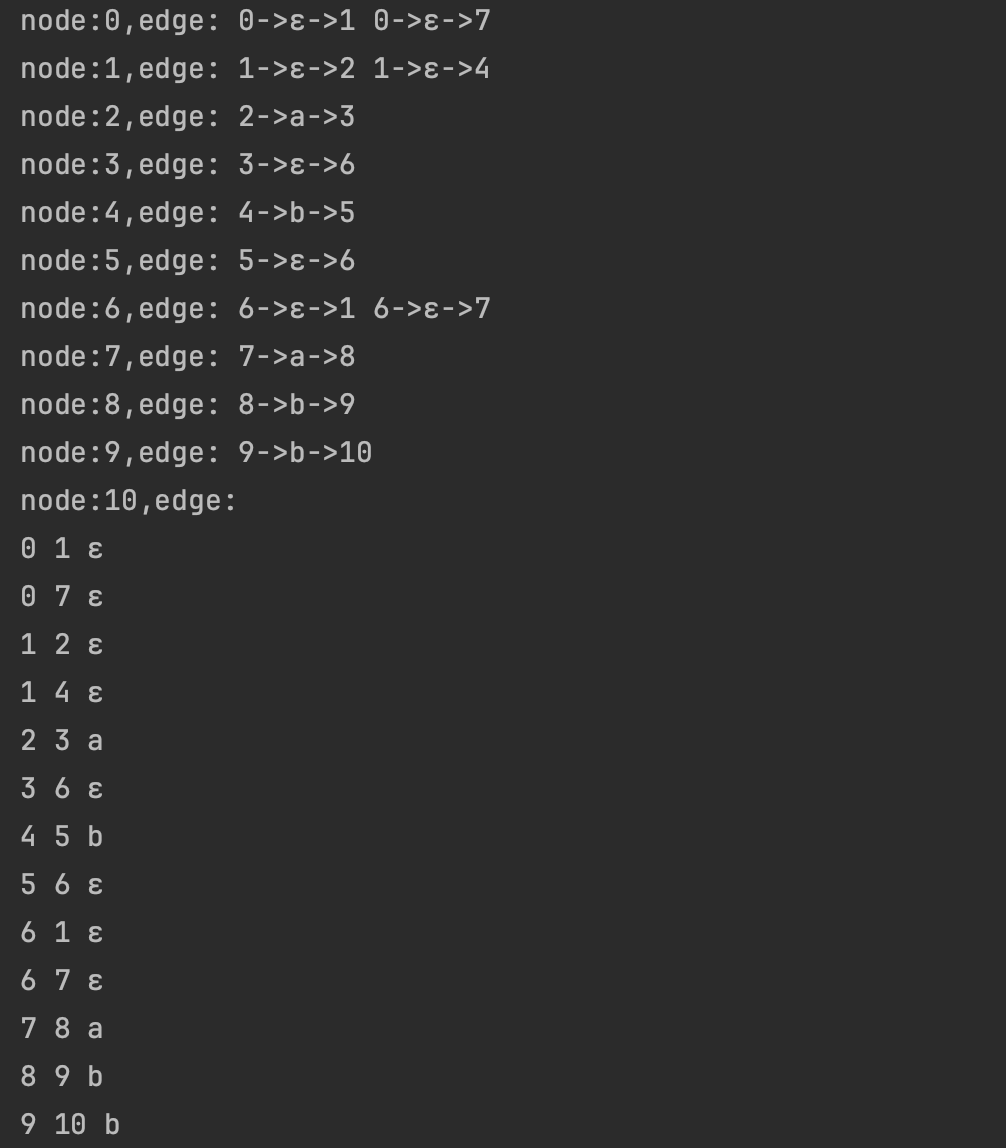
1. **测试结果**

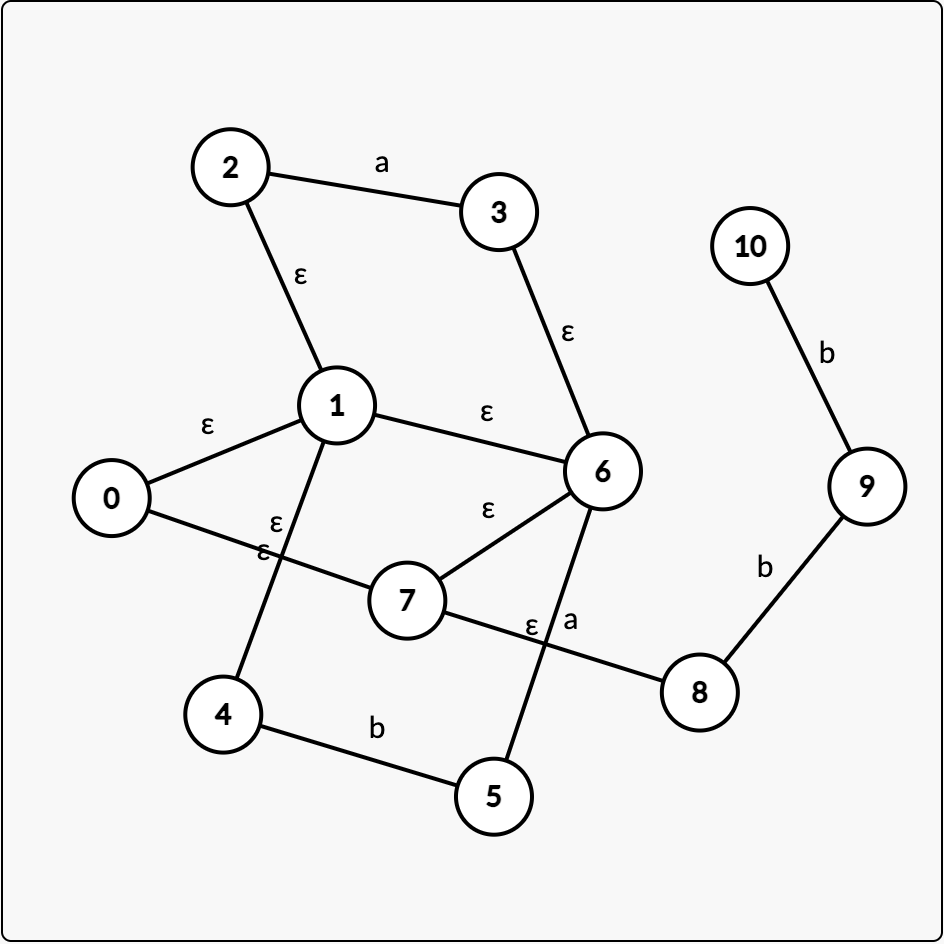
正规表达式 —> 格式表达式 —> 后缀表达式

下面是测试结果

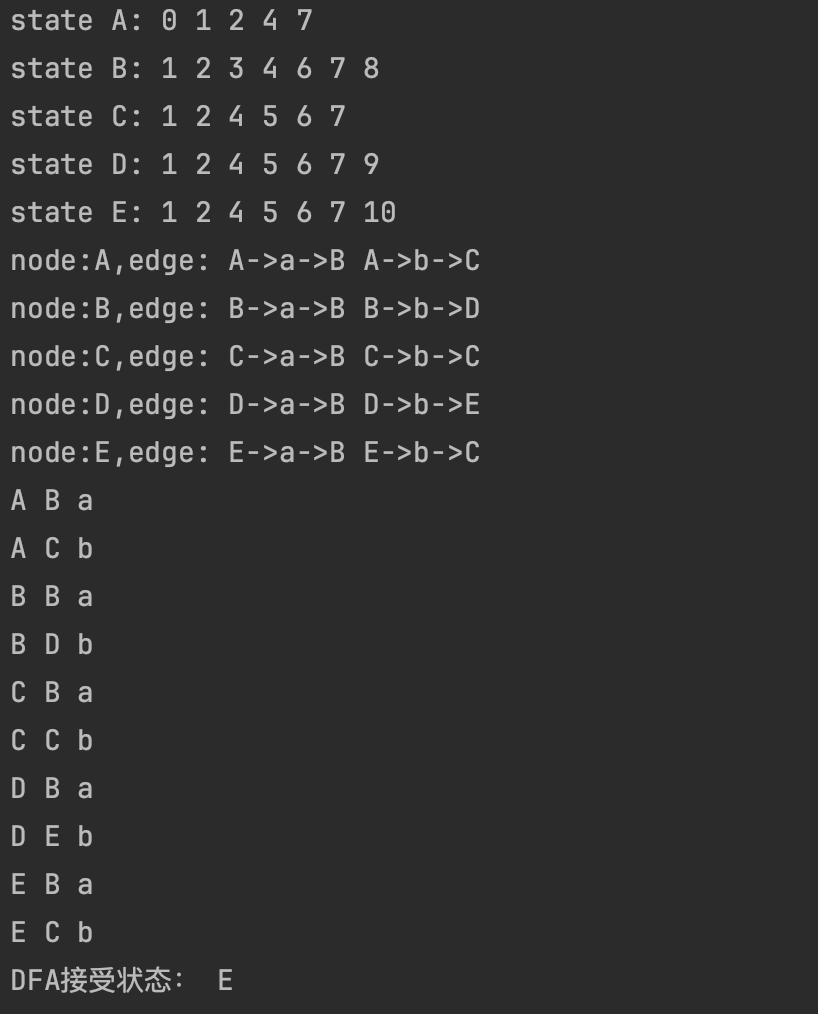


然后构建NFA，下图是输出

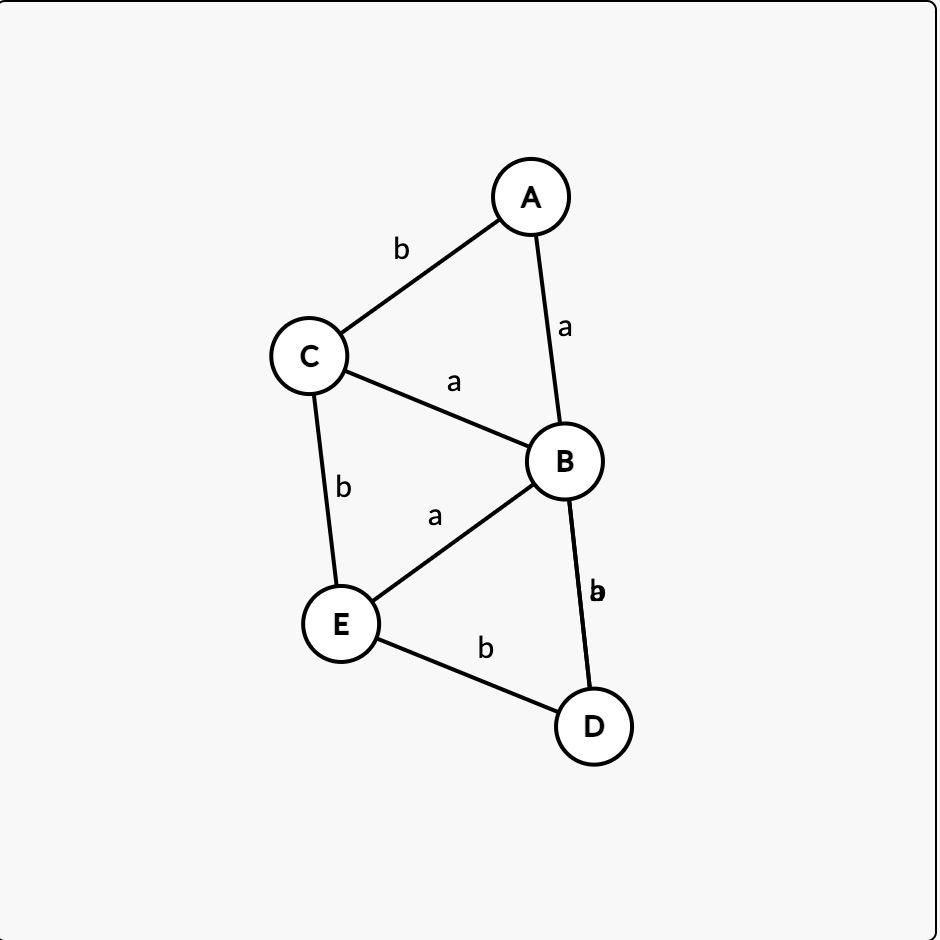


下图为NFA的可视化结果  


**下图为DFA的构建结果**

****

**下图为DFA的可视化结果**

****

1. **源代码**

词法分析相关算法代码

#include<iostream>  
#include <string>  
#include <stack>  
#include <vector>  
#include <map>  
#include <set>  
#include <queue>  
using namespace std;  
//从正规表达式构造NFA  
struct Edge{  
 int to;  
 string c;//需要用到ε这个字符，所以用string  
};  
struct Node{  
 int index;  
 vector<Edge>edge;  
};  
class NFA{  
public:  
 int begin, end;  
 map<int,vector<Edge> >Graph;//构建图  
 NFA(){  
 begin = 0;  
 end = 1;  
 Edge edge = {end,"\u03B5"};//"\u03B5"代表ε这个字符  
 Graph[begin].push\_back(edge);  
 Graph[end] = vector<Edge>();  
 }  
 NFA(string c){  
 begin = 0;  
 end = 1;  
 Edge edge = {1,c};  
 Graph[begin].push\_back(edge);  
 Graph[end] = vector<Edge>();  
 }  
 //重载三个操作方式  
 NFA operator \*(){  
 NFA newNFA;  
 int offset = 1;  
 for(auto &node : Graph){  
 if(node.second.size() != 0){  
 for(auto &edge:node.second){  
 edge.to += offset;  
 }  
 }  
 newNFA.Graph[node.first +offset] = node.second;  
 }  
 newNFA.begin = 0;  
 newNFA.end = Graph.size() + 1;  
 newNFA.Graph[newNFA.begin].clear();  
 newNFA.Graph[newNFA.end] = vector<Edge>();  
 Edge p = {newNFA.begin + 1,"\u03B5"};  
 newNFA.Graph[newNFA.end -1].push\_back(p);  
 Edge e1 = {newNFA.begin + 1,"\u03B5"};  
 newNFA.Graph[newNFA.begin].push\_back(e1);  
 Edge e2 = {newNFA.end,"\u03B5"};  
 newNFA.Graph[newNFA.end - 1].push\_back(e2);  
 Edge e3 = {newNFA.end,"\u03B5"};  
 newNFA.Graph[newNFA.begin].push\_back(e3);  
 return newNFA;  
 }  
 NFA operator |(NFA &t){  
 NFA newNFA;  
 int offset = 1;  
 for(auto &node : Graph){  
 if(node.second.size() != 0){  
 for(auto &edge : node.second){  
 edge.to += offset;//更新  
 }  
 }  
 newNFA.Graph[node.first + offset] = node.second;  
 }  
 offset = Graph.size() + 1;  
 for(auto &node :t.Graph){  
 if(node.second.size() != 0){  
 for(auto &edge : node.second){  
 edge.to += offset;//更新  
 }  
 }  
 newNFA.Graph[node.first + offset] = node.second;  
 }  
 newNFA.begin = 0;  
 newNFA.end = Graph.size() + t.Graph.size() + 1;  
 newNFA.Graph[newNFA.begin].clear();  
 newNFA.Graph[newNFA.end] = vector<Edge>();  
 Edge e1 = {1,"\u03B5"};  
 Edge e2 = {Graph.size() + 1 ,"\u03B5"};  
 newNFA.Graph[begin].push\_back(e1);  
 newNFA.Graph[begin].push\_back(e2);  
 Edge e3 = {newNFA.end,"\u03B5"};  
 newNFA.Graph[Graph.size()].push\_back(e3);  
 newNFA.Graph[Graph.size() + t.Graph.size()].push\_back(e3);  
 return newNFA;  
 }  
 NFA operator +(NFA &t){  
 int offset = Graph.size();  
 //加入边  
 for(auto &edge : t.Graph[t.begin]){  
 Edge p = {t.begin + offset ,edge.c};  
 Graph[end].push\_back(p);  
 }  
 // 重合了一个节点  
 offset --;  
 for(auto &node : t.Graph){  
 if(node.first != t.begin){  
 if(node.second.size() != 0){//第一个节点不加入  
 for(auto &edge : node.second){  
 edge.to += offset;//更新  
 }  
 }  
 Graph[node.first + offset] = node.second;  
 }  
  
 }  
 begin = 0;  
 end = Graph.size() - 1; //重合了一个节点  
 return \*this;  
 }  
 set<int> e\_closure(set<int> states){  
 queue<int> e\_queue;  
 set<int> closure;  
 for(auto &state : states){  
 e\_queue.push(state);  
 closure.insert(state);  
 }  
 while (!e\_queue.empty()){  
 int node = e\_queue.front();  
 e\_queue.pop();  
 for(auto &edge :Graph[node]){  
 if(edge.c == "\u03B5"){  
 e\_queue.push(edge.to);  
 closure.insert(edge.to);  
 }  
 }  
 }  
 return closure;  
 }  
  
 set<int> move(set<int> v, string s){  
 set<int> ans;  
 for(auto &i : v){  
 for(auto &edge : Graph[i]){  
 if(edge.c == s){  
 ans.insert(edge.to);  
 }  
 }  
 }  
 return ans;  
 }  
 void printNFA(){  
 for(auto &node :Graph){  
 cout << "node:"<<node.first << ",edge: ";  
 if(!node.second.empty()){  
 for (auto &edge: node.second)  
 cout <<node.first<<"->"<< edge.c<<"->"<<edge.to <<" ";  
 }  
 cout << endl;  
 }  
 for(auto &node :Graph){  
 if(!node.second.empty()){  
 for (auto &edge :node.second) {  
 cout << node.first << " "<< edge.to << " " << edge.c << endl;  
 }  
 }  
  
 }  
 cout << "NFA起始状态: " <<begin << ",NFA终止状态为: " << end <<endl;  
 }  
  
};  
class DFA{  
public:  
 map<int,vector<Edge> > Graph;  
 map<set<int>,char > states;  
 set<char> acceptStates;  
 void printDFA(){  
 for(auto &state :states){  
 cout <<"state " <<state.second << ": ";  
 for(auto &s : state.first){  
 cout << s <<' ';  
 }  
 cout << endl;  
 }  
 for(auto &node :Graph){  
 cout << "node:"<<char(node.first) << ",edge: ";  
 if(!node.second.empty()){  
 for (auto &edge: node.second)  
 cout <<char(node.first)<<"->"<< edge.c<<"->"<<char(edge.to) <<" ";  
 }  
 cout << endl;  
 }  
 for(auto &node :Graph){  
 if(!node.second.empty()){  
 for (auto &edge :node.second) {  
 cout << char(node.first) << " "<< char(edge.to) << " " << edge.c << endl;  
 }  
 }  
 }  
 cout << "DFA接受状态： ";  
 for(auto &state : acceptStates){  
 cout <<state << " ";  
 }  
 }  
};  
class Regex{  
public:  
 string regex;  
 string formatted;  
 string suffix;  
 NFA nfa;  
 DFA dfa;  
 set<char> chars;  
 void Build\_format();  
 void Build\_suffix();  
 void Build\_NFA();  
 void Build\_DFA();  
  
 Regex(string str){  
 regex = str;  
 Build\_format();//转化为带有连接符号的  
 Build\_suffix();//转化为后缀表达式  
 }  
 void print\_regex(){  
 cout << "正规表达式为： " << this->regex << endl;  
 }  
 void print\_formatted(){  
 cout << "格式表达式为： " << this->formatted << endl;  
 }  
 void print\_suffix(){  
 cout << "后缀表达式为： " <<this->suffix << endl;  
 }  
 void printGraph(map<int,vector<Edge> > Graph){  
 for(auto &node :Graph){  
 cout << "node:"<<node.first << ",edge: ";  
 if(!node.second.empty()){  
 for (auto &edge: node.second)  
 cout <<node.first<<"->"<< edge.c<<"->"<<edge.to <<" ";  
 }  
 cout << endl;  
 }  
 for(auto &node :Graph){  
 if(!node.second.empty()){  
 for (auto &edge :node.second) {  
 cout << node.first << " "<< edge.to << " " << edge.c << endl;  
 }  
 }  
  
 }  
 }  
};  
//判断是否要加连接符  
bool judge\_add\_point(char front,char back){  
 if(front != '(' && front != '|' && back != ')' && back != '|' && back != '\*' )  
 return true;  
 else  
 return false;  
}  
//构造格式化表达式  
void Regex::Build\_format() {  
 for(int i = 0; i < regex.size(); i++){  
 //找到一些添加连接符号的地方，最后不需要加  
 if(judge\_add\_point(regex[i],regex[i+1]) && i+1 < regex.size()){  
 //主要要插入的位置  
 formatted += regex[i];  
 formatted += '+';  
 }  
 else  
 formatted += regex[i];  
 }  
}  
//计算运算符的优先级  
int Priority(char c){  
 map<char,int> m;  
 m['\*'] = 3;  
 m['|'] = 1;  
 m['+'] = 2;  
 return m[c];  
}  
//用来表示显示的连接运算符  
bool isOperator(char c){  
 if(c == '\*' || c == '(' || c == ')' || c == '+' || c == '|'){  
 return true;  
 }  
 else  
 return false;  
}  
//构造后缀表达式  
void Regex::Build\_suffix() {  
 stack<char> s;  
 for(char &c : formatted){  
 //判断是不是字符  
 if(isOperator(c)){  
 switch (c) {  
 case '(':  
 s.push(c);  
 break;  
 case ')':  
 while (!s.empty() && s.top() != '('){  
 suffix += s.top();  
 s.pop();  
 }  
 s.pop();  
 break;  
 // c = '\*' && c == '|‘ && c == '+'的情况  
 default:  
 while (!s.empty() && Priority(c) <= Priority(s.top())){  
 suffix += s.top();  
 s.pop();  
 }  
 s.push(c);  
 }  
 }  
 //处理字母，直接加入  
 else{  
 chars.insert(c);  
 suffix += c ;  
 }  
 }  
 while (!s.empty()){  
 suffix += s.top();  
 s.pop();  
 }  
}  
//构造NFA  
void Regex::Build\_NFA() {  
 stack<NFA> NFAStack;  
 for(char &c : suffix){  
 switch (c) {  
 case '\*':  
 NFAStack.top() = \*NFAStack.top();  
 break;  
 case '|': {  
 NFA a = NFAStack.top();  
 NFAStack.pop();  
 NFA b = NFAStack.top();  
 NFAStack.pop();  
 NFAStack.push(b|a);  
 break;  
 }  
 case '+':{  
 NFA m, n;  
 m = NFAStack.top();  
 NFAStack.pop();  
 n = NFAStack.top();  
 NFAStack.pop();  
 NFAStack.push(n+m);  
 break;  
 }  
 default:  
 string s(1,c);//转化为string  
 NFAStack.push(NFA(s));  
 break;  
 }  
 }  
 nfa = NFAStack.top();  
 NFAStack.pop();  
}  
  
  
vector<int> judge\_empty\_state(map<vector<int>,int> states){  
 for(auto &state : states){  
 if(!state.second){  
 return state.first;  
 }  
 }  
 return vector<int>();  
}  
  
void Regex::Build\_DFA() {  
 // 初始化DFA  
 char index = 'A';  
 queue<set<int> > workQueue;  
 // 获取NFA起始状态的ε闭包，并将其设置为DFA的起始状态  
 set<int> begin\_state;  
 begin\_state.insert(0);  
 set<int> startState = nfa.e\_closure(begin\_state);  
 dfa.states[startState] = index++;  
 if (startState.find(nfa.end) != startState.end()) {  
 dfa.acceptStates.insert(dfa.states[startState]); // 将新的接受状态添加到集合中  
 }  
 workQueue.push(startState);  
 // 开始循环，处理所有的状态集合  
 while (!workQueue.empty()) {  
 set<int> currentState = workQueue.front();  
 workQueue.pop();  
 // 对于当前状态集合，按照字母表中的每个字符计算转移  
 for (char c : chars) {  
 string s(1, c); // 将字符转换为字符串  
 set<int> nextState = nfa.e\_closure(nfa.move(currentState, s)); // 计算转移后的ε闭包  
 if (nextState.empty()) {  
 continue; // 跳过空的ε闭包  
 }  
 // 如果新状态集合还没有在DFA中出现过，加入到DFA中  
 if (dfa.states.find(nextState) == dfa.states.end()) {  
 dfa.states[nextState] = index++;  
 workQueue.push(nextState);  
 }  
 if (nextState.find(nfa.end) != nextState.end()) {  
 dfa.acceptStates.insert(dfa.states[nextState]); // 将新的接受状态添加到集合中  
 }  
 // 在DFA中为当前状态集合到新状态集合添加边  
 Edge newEdge = {dfa.states[nextState], s};  
 dfa.Graph[dfa.states[currentState]].push\_back(newEdge);  
 }  
 }  
}  
  
  
  
  
int main(){  
 string str = "(a|b)\*abb";  
 //string str = "a|b|c|d";  
 //string str = "abc";  
 //string str = "(aa)\*";  
 Regex s(str);  
 s.print\_regex();  
 s.print\_formatted();  
 s.print\_suffix();  
 s.Build\_NFA();  
 s.nfa.printNFA();  
 s.Build\_DFA();  
 s.dfa.printDFA();  
 return 0;  
}

语法分析相关算法的代码

#include <iostream>  
#include <map>  
#include <set>  
#include <vector>  
#include <queue>  
#include <sstream>  
using namespace std;  
  
class Productions{  
private:  
 map<string ,set<string> > first, follow;  
 map<string ,vector<string> > productions;  
public:  
 void read\_productions(){  
 string line;  
 cout << "输入产生式：" << endl;  
 while (getline(cin,line) && line != "end"){  
 istringstream iss(line);//分割字符串  
 string non\_terminal;//非终结符号  
 string arrow;//箭头  
 string production;  
 iss >> non\_terminal >> arrow;  
 while (iss >> production) {  
 // 如果读取到了'|'，则跳过它  
 if (production == "|") {  
 continue;  
 }  
 productions[non\_terminal].push\_back(production);  
 }  
 }  
 }  
 //处理单个字符  
 set<string> getFirst(char symbol) {  
 //小写字母终结符号加入  
 string s(1,symbol) ;//转化为string  
 if (!isupper(symbol)) {  
 set<string> c;  
 c.insert(s);  
 return c;  
 }  
  
 return first[s];  
 }  
 //处理字符串  
 set<string> getFirstString(const string& str) {  
 set<string> result;  
 for (char symbol : str) {  
 set<string> symbolFirst = getFirst(symbol);  
 result.insert(symbolFirst.begin(), symbolFirst.end());  
 //如果找到了小写字母就退出  
 if(symbolFirst.size() == 1)break;  
 //特殊处理e,不包含e就退出  
 if (symbolFirst.find("ε") == symbolFirst.end()) {  
 break;  
 }  
 }  
 return result;  
 }  
 void calculateFirst() {  
 bool changed = true;  
 while (changed) {  
 changed = false;  
 for (const auto& prod : productions) {  
 for (const auto& str : prod.second) {  
 size\_t beforeSize = first[prod.first].size();  
 if(str == "ε"){  
 first[prod.first].insert(str);  
 }  
 else {  
 set<string> strFirst = getFirstString(str);  
 first[prod.first].insert(strFirst.begin(), strFirst.end());  
 }  
 if (first[prod.first].size() != beforeSize) {  
 changed = true;  
 }  
 }  
 }  
 }  
 }  
 void calculateFollow() {  
 follow["S"].insert("$"); //假设S是第一个推导的  
 bool changed = true;  
 while (changed){  
 changed = false;  
 for (const auto& prod : productions) {  
 for (const auto& str : prod.second) {  
 for (size\_t i = 0; i < str.size(); ++i) {//遍历字符串  
 if (isupper(str[i])) {//非终结符号  
 //如果后面还有其他符号，First集（去除e）加入  
 //如果是最后一个，就将左侧的Follow加入到右侧  
 string s(1,str[i]);//转化为string  
 size\_t beforeSize = follow[s].size();  
 set<string> nextFirst = (i + 1 < str.size()) ? getFirstString(str.substr(i + 1)) : follow[prod.first];  
 if (nextFirst.count("ε") == str.size() - 1 - i) {  
 nextFirst.erase("ε");  
 nextFirst.insert(follow[prod.first].begin(), follow[prod.first].end());  
 }  
 follow[s].insert(nextFirst.begin(), nextFirst.end());  
 //检查Follow是否更新  
 if (follow[s].size() != beforeSize) {  
 changed = true;  
 }  
 }  
 }  
 }  
 }  
 }  
 }  
 void PrintFirst(){  
 for (const auto& f : first) {  
 cout << "FIRST(" << f.first << ") = { ";  
 for (string c : f.second) {  
 cout << c << " ";  
 }  
 cout << "}" << endl;  
 }  
 }  
 void PrintFollow(){  
 for (const auto& f : follow) {  
 cout << "FOLLOW(" << f.first << ") = { ";  
 for (string c : f.second) {  
 cout << c << " ";  
 }  
 cout << "}" << endl;  
 }  
 }  
  
 void CreateLL1Table() {  
 map<pair<string , string>, string> ll1Table;  
 for (const auto& prod : productions) {  
 string nonTerminal = prod.first;//左侧非终结符号  
 for (const auto& alpha : prod.second) {  
 set<string> firstAlpha = first[nonTerminal];  
 //set<string> firstAlpha = getFirstString(alpha);//求first集  
 for (string terminal : firstAlpha) {//终结符号  
 if (terminal != "ε") {  
 auto table = make\_pair(nonTerminal, terminal);  
 ll1Table[table] = alpha;  
 } else {//如果是e，用Follow集  
 set<string> followNonTerminal = follow[nonTerminal];  
 for (string followTerminal : followNonTerminal) {  
 auto table = make\_pair(nonTerminal, followTerminal);  
 ll1Table[table] = alpha;  
 }  
 }  
 }  
 }  
 }  
 // 打印分析表  
 cout << "LL(1) 分析表:" << endl;  
 for (const auto& entry : ll1Table) {  
 cout << "M[" << entry.first.first << ", " << entry.first.second << "] = " << entry.second << endl;  
 }  
 }  
  
// void creatll1table(){  
// map<pair<char, string>, string> ll1Table;  
// for(auto &prod : productions){  
// for(auto &terminal : first[prod.first]){  
// auto table = make\_pair(prod.first, terminal);  
// ll1Table[table] = prod.second  
// }  
// }  
// }  
  
};  
  
  
  
  
int main() {  
 Productions s;  
 s.read\_productions();  
 s.calculateFirst();  
 s.calculateFollow();  
 s.PrintFirst();  
 s.PrintFollow();  
 s.CreateLL1Table();  
 return 0;  
}  
/\*  
  
 S -> a  
 B -> t | S  
  
E → TE'  
E' → +TE' | e  
T → FT'  
T' → \*FT' | e  
F → (E) | i  
  
  
S → A | a  
A → Bc | ε  
B → d | AS  
  
S → MH | a  
H → LSo | ε  
K → dML | ε  
L → eHf  
M → K | bLM  
  
  
E → TE’  
E’ → +TE’| ε  
T → FT’  
T’ → \*FT’| ε  
F → (E) | i  
  
 \*/