编译原理实验报告

姓　　名：　　牟鑫一

学　　号：　　20161001764

班　　级：　　191174

指 导 老 师：　　刘远兴

目　　录

题目一：词法分析程序

一、 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

二、 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

三、 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。

四、 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。

五、 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 checkstring):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。

2、 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。

3、 正规式转化 NFA

1) 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。
2) 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。

3) 具体流程：

a) 检验正则表达式是否符合正则规则，否则返回 false。

b) 把 0 和 1 的初始初态终态压入 moduleStack。

c) 在已知正则表达式的首尾添加一对括号。

d) 把已知正则表达式中省略的连接符号添加上。

e) 开始进行输入：

f) If 检测到输入为 ‘’ || ‘|’ || ‘.’)，就把这些符号压入 assistStack。

g) 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。

h) 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。

i) 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。

j) 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。

4、 数据结构

1) 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
};
```

2) 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*);
};
```

3) 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*);
};
```

4) 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;
};
```

5) 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();
};
```

6) 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();
};
```

5、 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

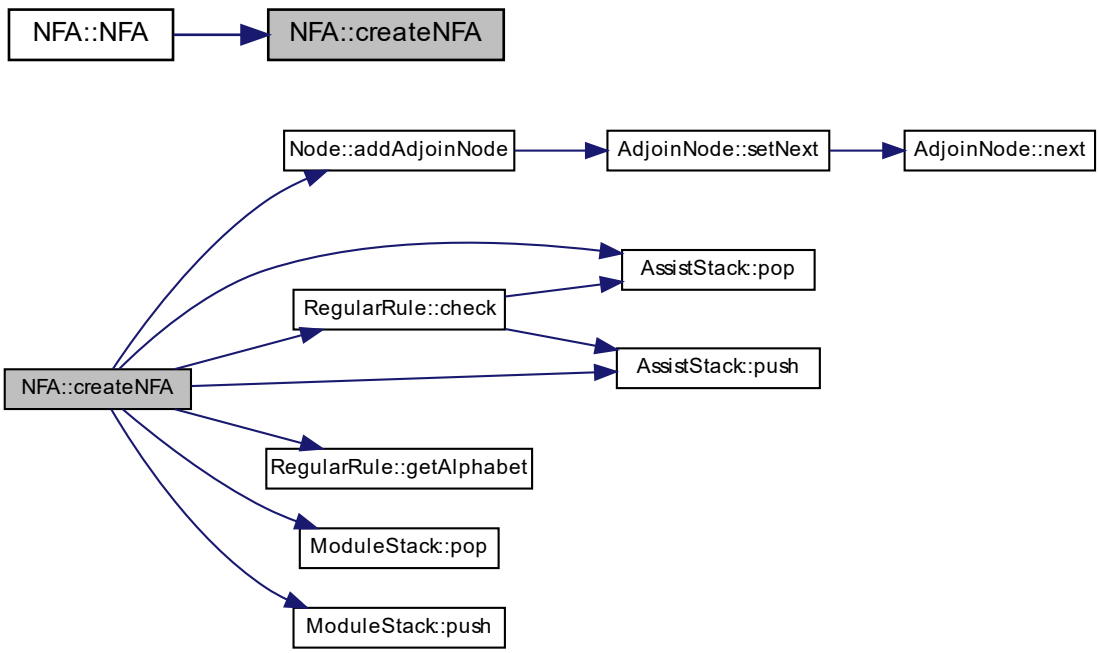
6、 基本思想

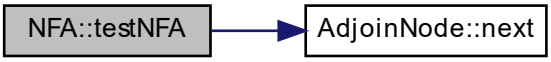
初始的初态终态压入栈 moduleStack，给正则式首尾加上括号并添加连接符号，例如输入正则式为：aa|b)*a，经过这一步操作，正则式变为：a.a|b)*.a) 。然后循环遍历正则式字符串，如果是字符 '' 、 '|' 或 '.' ，则将其压入辅助符号存储栈；如果是符号 '*' ，模块状态出栈，加入新的节点并插入节点表，为新的节点构建邻接表，最后将新模块的初态和终态入栈；如果是 ')' ，将符号栈内元素储栈，如果出战符号是 '.' 则进行连接操作，如果出栈符号是'|'就进行或操作，直至遇到左括号 '' 跳出内循环；如果是一般符号，则将新元素加入节点表，连接并入栈。最后连接 moduleStack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

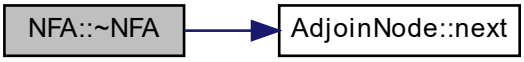
六、 自己负责的模块设计

我负责的模块是：正则式到 NFA 的转换

1、 函数调用关系图







2、 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();
```

3、 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();
```

七、 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;
}
```

八、 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。

2、 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。

3、 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。

九、 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。

十、 测试结果

输入正则表达式：aa|b)*a

执行结果如下：

```
请输入正则表达式：
a(a|b)*a
转化的NFA为：
当前态：0        输入：c  转至：1;
当前态：1        输入：a  转至：2;
当前态：2        输入：c  转至：9;
当前态：3        输入：a  转至：4;
当前态：4        输入：c  转至：8;
当前态：5        输入：b  转至：6;
当前态：6        输入：c  转至：8;
当前态：7        输入：c  转至：3;   输入：c  转至：5;
当前态：8        输入：c  转至：9;   输入：c  转至：10;
当前态：9        输入：c  转至：10;   输入：c  转至：7;
当前态：10       输入：c  转至：11;
当前态：11       输入：a  转至：12;
当前态：12
终态：12
```

十一、 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。

题目二：算符优先文法分析

一、 题目

算符优先文法分析

二、 问题描述

- 根据给定文法，先求出 FIRSTVT 和 LASTVT 集合，构造算符优先关系表（要求算符优先关系表输出到屏幕或者输出到文件）；
- 根据算法和优先关系表分析给定表达式是否是该文法识别的正确的算术表达式（要求输出归约过程）；
- 给定表达式文法为：

```
GE '): E'→#E#
E->E+T|T
T->T*F|F
F->E)|i
```

- 分析的句子为：

  i+i)*i 和 i+i)*i

三、 基本要求

- 选择最有代表性的语法分析方法算符优先法；
- 选择对各种常见程序语言都用的语法结构，如赋值语句（尤指表达式）作为分析对象，并且与所选语法分析方法要比较贴切；
- 实习时间为 6 学时。

四、 小组分工

- 牟鑫一：根据给定文法，求出 FIRSTVT 集；
- 江佳盛：根据给定文法，求出 LASTVT 集；
- 刘栋阳：构造算符优先关系表；
- 郭兴：根据算法和优先关系表分析给定表达式是否该文法识别的正确的算术表达式，

输出归约过程；

- 李泽栋、梅涵：综合，编写整个程序的框架、界面设计、数据结构设计等。

五、 整体设计

1、 正则文法

- 用户需输入正则文法的规则个数，以便于对正则规则的遍历；
- 由用户输入正则文法，可识别“或”关系（即“|”），输入的正则规则存储到二维数组 str[maxn][Maxn]中；
- 用一个双层循环获取输入的正则规则里包含的终结符，存储到数组 terminator[maxn]中备用。

2、 获取 FIRSTVT 集

- 遍历正则规则的左部，依次查找每个非终结符的 FIRSTVT 集；
- 对应某一个正则规则的左部非终结符，遍历这条正则规则的每个字符，对正则规则的右部做判断，找出该正则规则对应左部非终结符的 FIRSTVT 集。

3、 输出 FIRSTVT 集

- 遍历数组 firstvt[maxn][maxn]输出各个非终结符的 FIRSTVT 集

4、 数据结构

由于是但文件结构，所以可直接使用全局变量：

```
const int Maxn = 110;        //单条规则最多 110 个字符
const int maxn = 20;         //最多 20 条文法规则
char str[maxn][Maxn];        //储存输入的正则文法
char terminator[maxn];       //存储终结符
char firstvt[maxn][maxn];    //存储 FIRSTVT 集
int firstflag[maxn];         //记录非终结符的 FIRSTVT 集是否已求出
```

```
int fcnt[maxn];                    //非终结符 FIRSTVT 集的元素个数
```

5、 文件结构

- Operator-precedence Parsing.cpp

6、 基本思想

对应某一个正则规则的左部非终结符，遍历这条正则规则的每个字符，判断右部的第一个字符是否终结符，若是则将其加入该非终结符的 FIRSTVT 集；若不是则判断第二个元素是否终结符，若是则将其加入该非终结符的 FIRSTVT 集；若右部第一个字符是自身，则继续往后判断；若右部第一个字符是不是自身的其它非终结符，则递归获取这个非终结符的 FIRSTVT 集并加入该非终结符的 FIRSTVT 集；继续往后遍历这一正则规则的后续字符，查找是否有符号“|”，若有则作上述相同的判断。

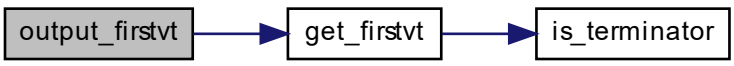
如上直至遍历完该字符串，表明该非终结符的 FIRSTVT 集已找全，已找全的非终结符在数组 firstflag[maxn]中标记为 1，当再次遇到该非终结符时则不用重复求 FIRSTVT 集。

六、 自己负责的模块设计

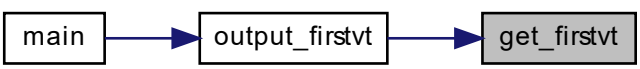
我负责的模块是：根据给定文法，求出 FIRSTVT 集

1、 函数调用关系图

output_firstvt） 函数调用 get_firstvt） 函数获取某一正则规则的左部非终结符的 FIRSTVT 集，get_firstvt） 函数调用 is_terminator） 函数判断某字符是否终结符：



Main）函数调用 output_firstvt） 函数输出所有非终结符的 FIRSTVT 集：



2、 模块接口说明

1、 void output_firstvt(int T);

传入正则规则的条数 T，输出 FIRSTVT 集

2、 void get_firstvt(char s, int T);

传入非终结符 s，正则规则条数 T，获取非终结符 s 的 FIRSTVT 集

3、 int is_terminator(char c);

传入字符 c，判断其是否为终结符，是返回 1，不是返回 0

3、 函数的功能实现

```
//判断字符 c 是否终结符
int is_terminator(char c);
//输出 firstvt 集
void output_firstvt(int T);
//求 s 非终结符的 FIRSTVT 集
//参数 s 为正则文法的左部非终结符，T 为正则规则的个数
void get_firstvt(char s, int T);
```

七、 算法设计

1、 判断字符 c 是否终结符

```
int is_terminator(char c) { //判断字符 c 是否终结符
    for (int i = 0; terminator[i] != '\0'; i++) {
        if (terminator[i] == c)
            return 1;
    }
    return 0;
}
```

2、 求 s 非终结符的 FIRSTVT 集

```
//参数 s 为正则文法的左部非终结符，T 为正则规则的个数
void get_firstvt(char s, int T) {
    int i, j, tt;
    //该 for 循环找出非终结符 s 所在正在规则的序号 i
    for (i = 0; i < T; i++) {
        if (str[i][0] == s)
            break;
```

```
    }
    //如果 str[i][0]的 FIRSTVT 集未求出，
    if (!firstflag[i]) {
        int k = fcnt[i]; //str[i][0]的 FIRSTVT 集元素个数，初值为 0
        //遍历正则规则 str[i]的每个字符
        for (j = 0; str[i][j] != '\0'; j++) {
            if (str[i][j] == '>' | str[i][j] == '|') {
                //判断后一个字符是否终结符，若是则加入终结符集
                if (is_terminator(str[i][j + 1])) {
                    firstvt[i][k++] = str[i][j + 1];
                }
                else {
                    //否则判断其后第二个字符是否终结符，若是则加入终结符集
                    if (is_terminator(str[i][j + 2])) {
                        firstvt[i][k++] = str[i][j + 2];
                    }
                    //如果后一个非终结符不是自身，则需要递归获取该非终结符的 FIRSTVT 集加
入到 str[i][0]的非终结符集中
                    if (str[i][j + 1] != s) {
                        int ii, jj;
                        //递归获取非终结符 str[i][j + 1]的 FIRSTVT 集
                        get_firstvt(str[i][j + 1], T);
                        //该 for 循环找出非终结符 str[i][j+1]所在正在规则的序号 ii
                        for (ii = 0; ii < T; ii++) {
                            if (str[ii][0] == str[i][j + 1])
                                break;
                        }
                       //将非终结符 str[i][j + 1]的非终结符集中的元素加入到 str[i][0]
的 FIRSTVT 集中
                        for (jj = 0; jj < fcnt[ii]; jj++) {
                            for (tt = 0; tt < k; tt++) {
                                if (firstvt[i][tt] == firstvt[ii][jj])
                                    break;
                            }
                            //tt == k 表明终结符 firstvt[ii][jj]应加入 FIRSTVT 集
                            if (tt == k) {
                                firstvt[i][k++] = firstvt[ii][jj];
                            }
                        }
                    }
                }
            }
        }
        firstvt[i][k] = '\0';//str[i][0]的 FIRSTVT 集已查找完毕，添加空串作为串结束符
```

```
            fcnt[i] = k;//记录 str[i][0]的 FIRSTVT 集元素个数
            firstflag[i] = 1;//标记已获取到 str[i][0]的 FIRSTVT 集
        }
    }
```

3、 输出 FIRSTVT 的函数

```
void output_firstvt(int T) { //输出 firstvt 集
    for (int i = 0; i < T; i++) {
        get_firstvt(str[i][0], T);//获取 str[i][0]的 FIRSTVT 集
    }
    for (int i = 0; i < T; i++) {
        printf("FIRSTVT[%c]:", str[i][0]);
        for (int j = 0; j < fcnt[i]; j++) {//fcnt[i]为 FIRSTVT 集元素个数
            printf("%c ", firstvt[i][j]);
        }
        puts("");//将空字符串写入到标准输出 stdout，并追加一个换行符，等效于输出一个换行
    }
}
```

八、 调试分析

1、 优点分析

获取 FIRSTVT 集的函数考虑比较全面，综合考虑了各种情况，使用多个 if 语句进行判断，可以处理带有特殊符号或“|”的正则规则，采用递归处理了开头为非终结符的问题，一个函数就可以应付多层推导的问题

2、 缺点分析

- 数据存储采用的是全局普通变量，对于我们在做的程序是没有什么问题的，但是全局变量局限太多，不适用于大型项目，不利于程序的扩展；
- 获取 FIRSTVT 集的函数 get_firstvt) 有些复杂，嵌套了多层选择结构和循环结构，应该是可以再精简些或者将一些代码独立出来写成单独的函数来调用。

3、 改进方法

- 可以将一些数据和函数抽象到类中，增强可读性和扩展性；

- 较为复杂的函数可以模块化处理，拆分为多个小函数，通过函数调用实现同样的功能，是程序更容易理解。

九、 使用手册

从全局变量中获取正则文法数据，外部只需调用 output_firstvt) 即可输出所有非终结符的 FIRSTVT 集，output_firstvt) 函数调用 get_firstvt) 函数获取某一非终结符的 FIRSTVT 集，将各非终结符的 FIRSTVT 集中的元素的个数存储到 fcnt[]数组中，用 firstflag[maxn]数组标记某一非终结符的 FIRSTVT 集是否已求出，将求出的 FIRSTVT 集存储到数组 firstvt[maxn][maxn] 中。

遍历二维数组 firstvt[maxn][maxn] 即可获取各非终结符的 FIRSTVT 集元素。

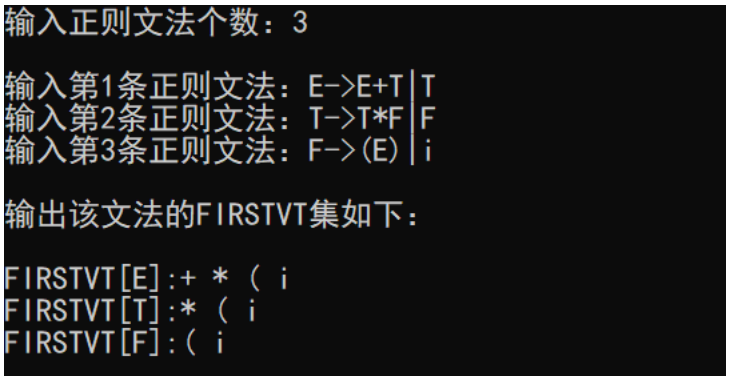
十、 测试结果

输入正则规则个数：3

输入正则文法：

```
E->E+T|T
T->T*F|F
F->(E)|i
```

执行结果如下：



十一、 总结

首先通过这个题目，我对算符优先分析有了更加深刻的理解，对 FIRSTVT 集的定义理解得更加透彻，搞清楚了求解 FIRSTVT 集以及 LASTVT 集得详细步骤。

其次，通过这两个题目的联系，我对编译原理学过的内容都有了更多自己的理解，对我们学习编程以来一直在用的各种编译器也有了一些自己的理解，收获颇多。