**《编译原理与设计》**

# LR语法分析程序的设计与实现

实验报告

班级 2018211308

学号 2018211393

姓名 王一丰

日期 2020.11.18

目录

[LR语法分析程序的设计与实现 1](#_Toc56626982)

[一．实验内容 3](#_Toc56626983)

[二．实验环境 4](#_Toc56626984)

[三．实验设计 4](#_Toc56626985)

[1.总体思路 4](#_Toc56626986)

[2.数据结构与函数说明 5](#_Toc56626987)

[四．实验原理 9](#_Toc56626988)

[1.项目集闭包运算： 9](#_Toc56626989)

[算法原理: 9](#_Toc56626990)

[算法实现： 10](#_Toc56626991)

[2.DFA构建 10](#_Toc56626992)

[算法原理： 10](#_Toc56626993)

[算法实现： 11](#_Toc56626994)

[3.推空集 12](#_Toc56626995)

[算法原理： 12](#_Toc56626996)

[算法实现： 13](#_Toc56626997)

[4.first集 13](#_Toc56626998)

[算法原理： 13](#_Toc56626999)

[算法实现： 14](#_Toc56627000)

[5.follow集 14](#_Toc56627001)

[算法原理： 14](#_Toc56627002)

[算法实现： 15](#_Toc56627003)

[6.预测分析表 16](#_Toc56627004)

[算法原理： 16](#_Toc56627005)

[算法实现： 17](#_Toc56627006)

[7.预测分析程序 17](#_Toc56627007)

[算法原理： 17](#_Toc56627008)

[算法实现： 19](#_Toc56627009)

[五．测试样例 20](#_Toc56627010)

[1.文法输入 20](#_Toc56627011)

[2.文法分析结果输出 21](#_Toc56627012)

[3.字符流输入与结果输出 23](#_Toc56627013)

[六．实验总结 26](#_Toc56627014)

[1.难点攻克： 26](#_Toc56627015)

[a.含左递归的first集 26](#_Toc56627016)

[b.类内部的比较 26](#_Toc56627017)

[c.迭代器失效问题 27](#_Toc56627018)

[d.数据结构过于复杂问题 27](#_Toc56627019)

[e.后继状态问题 27](#_Toc56627020)

[2.实验心得： 27](#_Toc56627021)

# 实验内容

编写语法分析程序，实现对算术表达式的语法分析。要求所分析的算术表达式 由如下的文法产生：

E → E+T | E–T | T

T → T\*F | T/F | F

F → (E) | num

在对输入的算术表达式进行分析的过程中，依次输出所采用的产生式。编写语 法分析程序实现自底向上的分析，要求如下：

(1) 构造识别该文法所有活前缀的 DFA；

(2) 构造该文法的 LR 分析表；

(3) 编程实现算法 4.3，构造 LR 分析程序。

# 二．实验环境

硬件环境：搭载了微软Windows10操作系统的笔记本电脑；

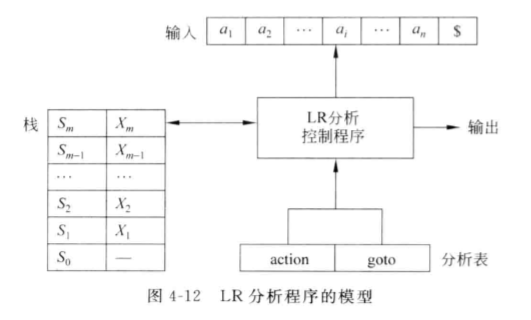
软件环境：Visual Studio 2017 15.8.9集成开发环境。

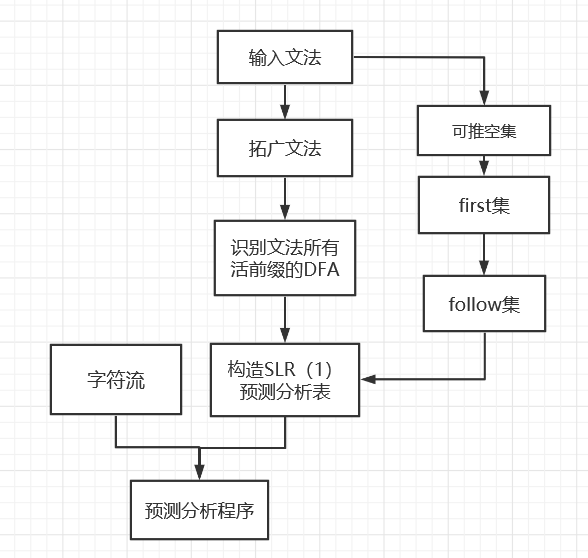
# 三．实验设计

## 1.总体思路

本语法分析程序首先利用上次实验实现的词法分析程序将输入串转化为token流，再在此基础上采用 SLR(1)分析方法对输入算术表达式进行分析，并输出析过程采用的产生式。

SLR(1)分析的关键在于构造预测分析表，然后使用分析表与一个分析栈进行联合控制，实现对输入符号串的自底向上分析。





本分析器使用了SLR（1）分析表，比较LR（0）分析表有着更好的性能。

## 2.数据结构与函数说明

vector<string> split(const string &str,char ch=' ');//字符串分隔函数

typedef string token;//符号类型

struct production { //产生式数据结构

token left; //左侧非终结符

deque<token> right; //右侧产生式

};

class grammar { //文法类

public:

vector<token> non\_terminators, terminators; //非终结符集，终结符集

token start\_token; //文法起始符

vector<production> productions; //文法所有产生式

map<token, bool> first\_over, follow\_over,epsilon\_over;//记录非终结符first、follow集是否已求

map<token, set<token>> first,follow; //非终结符的first、follow集

map<token, bool> epsilon;

map<token, set<int>> productions\_class; //按产生式左侧非终结符为产生式分类

vector<set<token>> first\_of\_production; //产生式的first集

map<token, map<token, bool>> follow\_include; //follow集的互相包含关系

public:

grammar(); //默认构造函数，通过文件读入文法

void infer\_epsilon(const token&);//推导可推空符号

void output\_epsilon();//计算可推空非终结符

void first\_derivation(const token&);//对应非终结符first推导

void follow\_derivation(const token&);//对应非终结符follow堆推导

void output\_productions();//输出所有产生式

void output\_first();//计算first集

void output\_follow();//计算follow集

void output\_first\_follow();//打印first、follow表

void extend(); //拓广文法

void ini\_productions\_class();//初始化productions\_class

};

extern grammar g;//全局文法

struct Item//项目类

{

int production\_num;//产生式编号

int point\_position;//点的位置

//运算符的重载

bool operator <(const Item& rhs) const {

return (production\_num < rhs.production\_num)

|| (production\_num == rhs.production\_num && point\_position < rhs.point\_position);

};

bool operator ==(const Item& rhs) const {

return (production\_num == rhs.production\_num

&& point\_position == rhs.point\_position);

};

bool operator !=(const Item& rhs) const {

return !(\*this == rhs);

};

};

class Item\_set//项目集类

{

public:

set<Item> items;//项目集合

map<token, set<Item>> trans;//后继项目集合

set<Item> reduce\_items;//规约项目集合

public:

Item\_set(set<Item>);//通过初始项目集构造新项目集

void item\_set\_closure();//闭包运算函数

bool in\_item\_set(Item);//判断项目是否存在于项目集

void output\_item\_set();//输出项目集

bool operator ==(const Item\_set& rhs) const {//运算符重载

if (items.size() == rhs.items.size()) {

for (auto it(items.begin()), r\_it(rhs.items.begin());it != items.end();++it, ++r\_it) {

if (\*it != \*r\_it) {

return false;

}

}

return true;

}

return false;

}

};

class DFA//识别文法活前缀的项目集规范族类

{

public:

vector<Item\_set> item\_sets;//项目集规范族

struct state//状态结构

{

map<token, int> go\_to;//后继状态

};

vector<state> states;//状态集合（与项目一一对应）

int state\_num;//状态数

void ini();//初始化函数

};

enum action\_type//动作枚举类型

{

shift,

reduce,

acc

};

struct action//动作表项

{

action\_type type;

int value;

};

class parser //LR(1)语法分析器类

{

public:

DFA dfa;//该分析器依据DFA

vector<map<token, action>> action\_table;//action表格

vector<map<token, int>> goto\_table;//goto表格

vector<token> token\_stream;//字符流

string action\_name[3] = { "shift","reduce","acc" };//动作名称数组

public:

void input\_token();//字符流导入

void table\_derivation();//预测分析表创建

void output\_parser\_table();//打印分析表

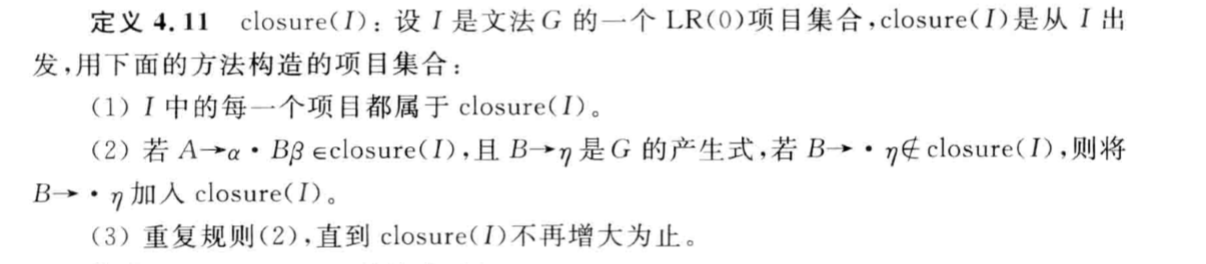
void predict();//预测分析程序

};

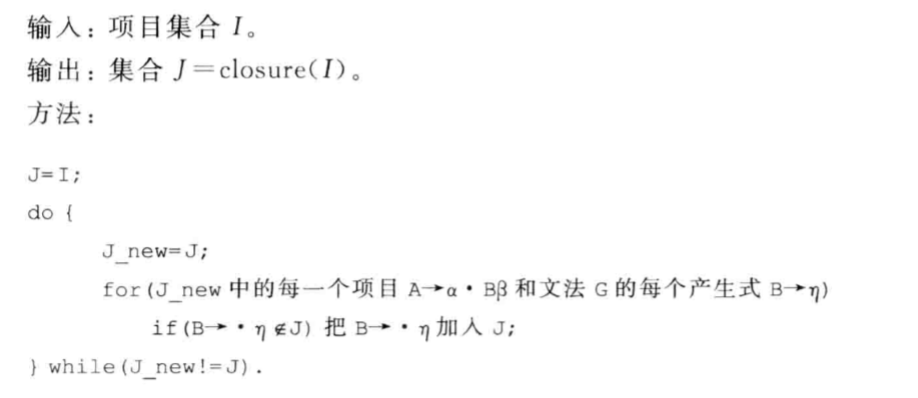
# 四．实验原理

## 1.项目集闭包运算：

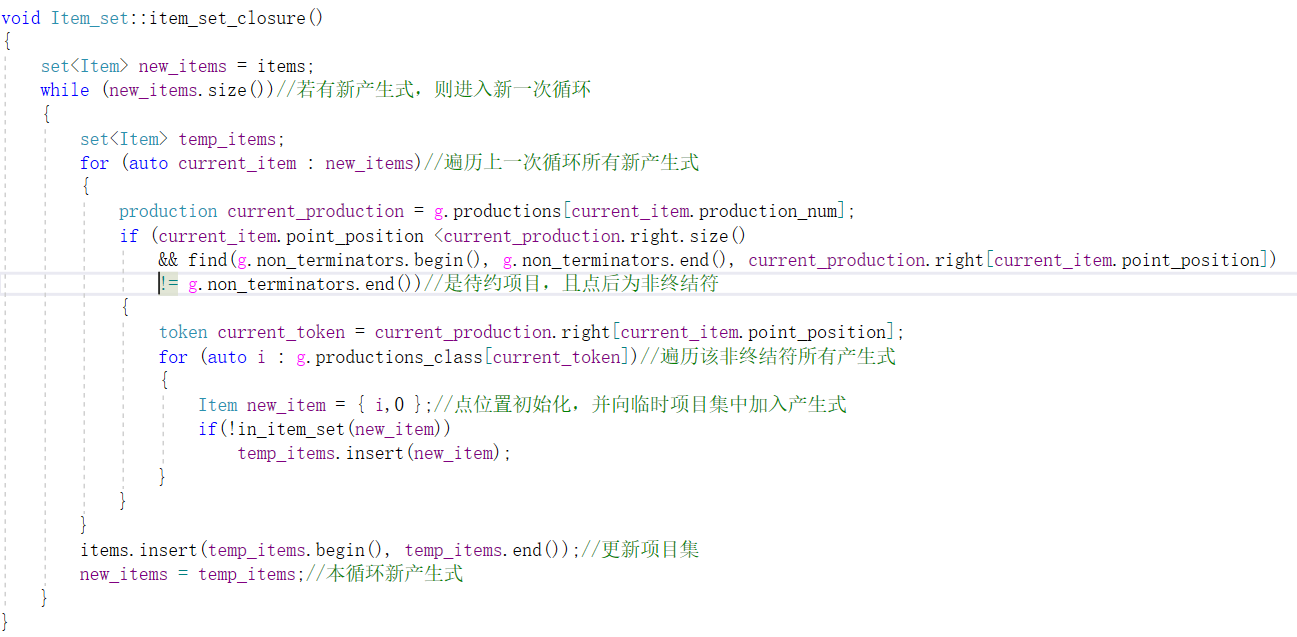
### 算法原理:



伪代码如下：

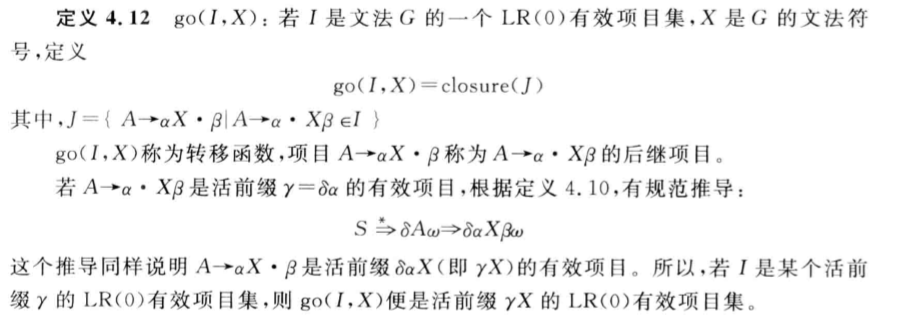


### 算法实现：

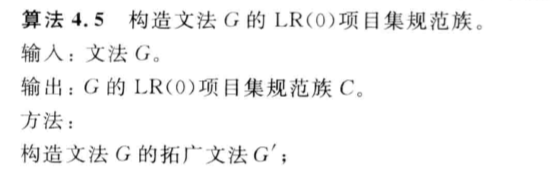


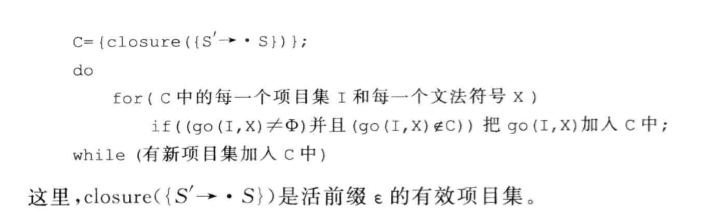
## 2.DFA构建

### 算法原理：



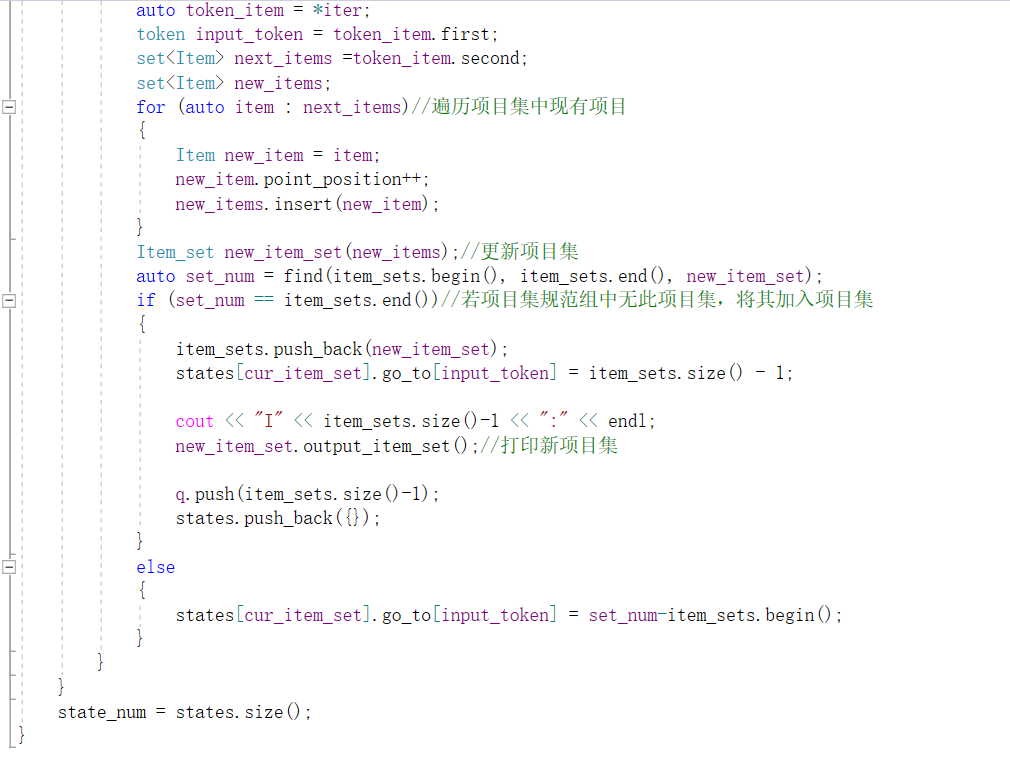
伪代码如下：





### 算法实现：





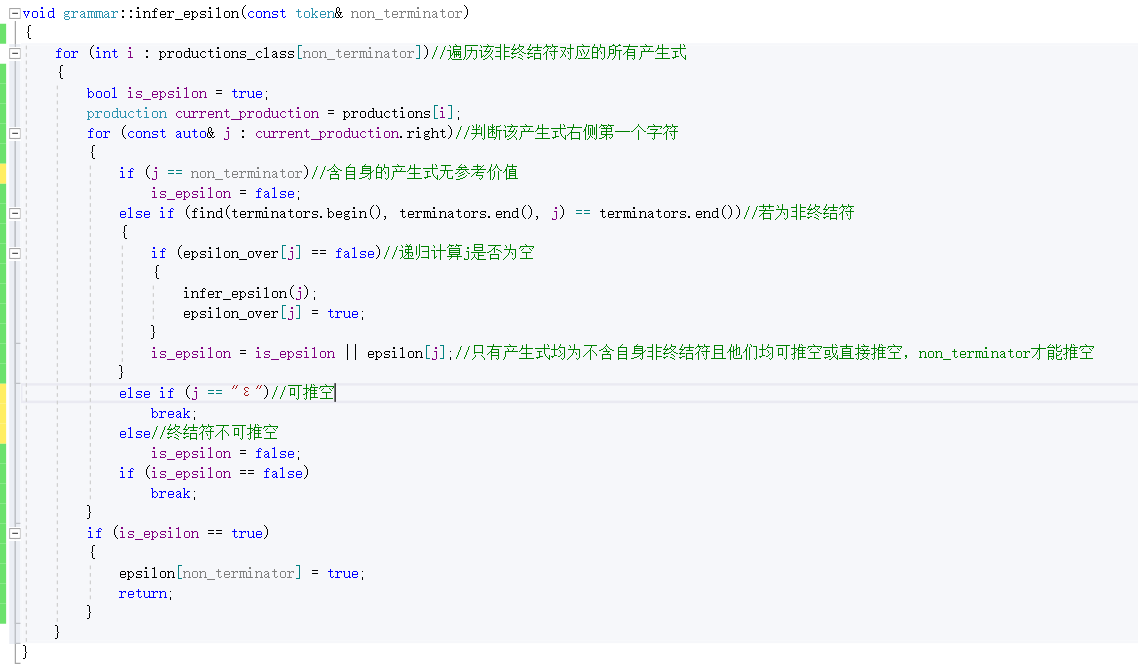
## 3.推空集

### 算法原理：

遍历该非终结符对应的所有产生式，遍历该产生式右侧字符，

1. 含自身或终结符的产生式无参考价值，直接跳过
2. 可直接推空的，直接写入可推空
3. 产生式中均为非终结符，递归求他们能否推空，均能推空，该非终结符才能推空

### 算法实现：



## 4.first集

### 算法原理：

对任意文法符号串，其 FIRST集为其可推导出的开头终结符号集合。

构造任意文法符号𝑋的 FIRST集FIRST(𝑋)可遍历𝑋的产生式并利用如下

规则：

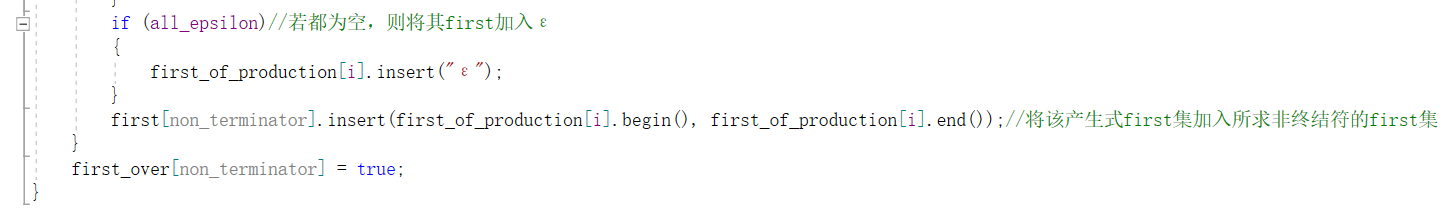
(1) 若𝑋 ∈ 𝑉T，则FIRST(𝑋) = {𝑋}；

(2) 若𝑋 ∈ 𝑉N，且有𝑋 → 𝑎…(𝑎 ∈ 𝑉T ⋁𝑎 = 𝜀)，则FIRST(𝑋) = FIRST(X) ∪ {𝑎}；

(3) 若有𝑋 → 𝑌1𝑌2 …𝑌𝑘𝑌𝑘+1 …，且有ε ∈ FIRST(𝑌 𝑖)(𝑖 = 1,…,𝑘)，则 FIRST(𝑋) = FIRST(X) ∪ {𝑎 ∈ FIRST(𝑌 𝑖) | 𝑎 ≠ 𝜀}； 若∀𝑌 𝑖,ε ∈ FIRST(𝑌 𝑖)，FIRST(𝑋) = FIRST(X) ∪ {𝜀}。

### 算法实现：





## 5.follow集

### 算法原理：

对任意非终结符，其 FOLLOW集是该文法所有句型中紧跟在其后的终 结符或结尾符号 end 的集合。

构造任意文法符号𝑋的 FOLLOW集FOLLOW(𝑋)可遍历文法所有产生式 并利用如下规则：

(1) 若𝑋是起始符号，则FOLLOW(𝑋) = FOLLOW(𝑋) ∪ {end}；

(2) 若有𝐴 → ⋯𝑋𝛽，则FOLLOW(𝑋) = FOLLOW(𝑋) ∪ {𝑎 ∈ FIRST(𝛽) | 𝑎 ≠ 𝜀}；

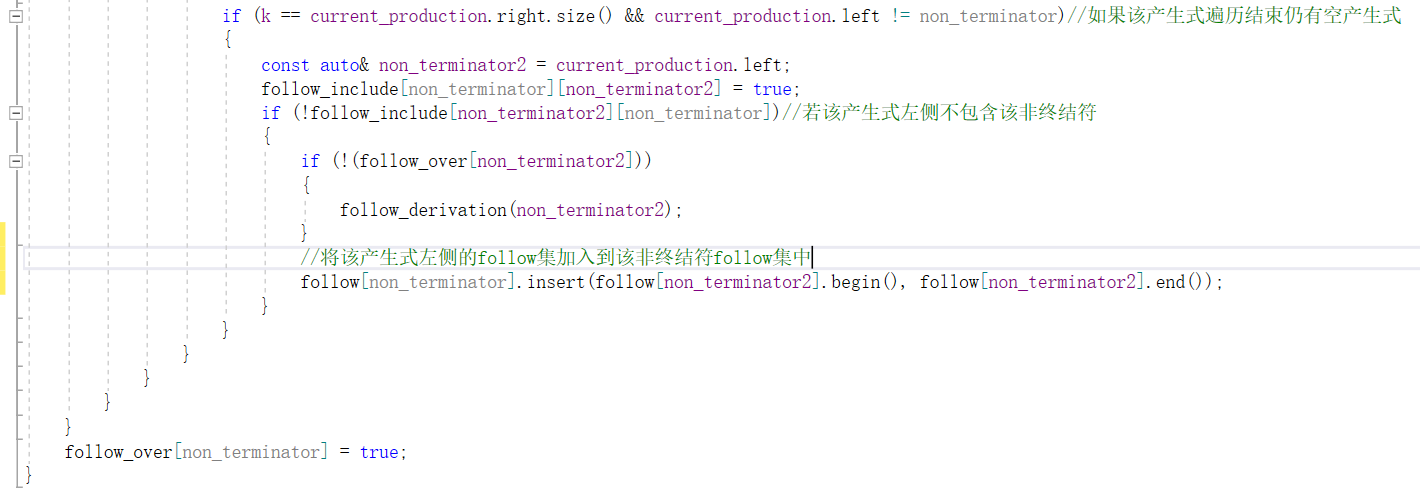
(3) 若有𝐴 → ⋯𝑋，或有𝐴 → ⋯𝑋𝛽且𝜀 ∈ FIRST(𝛽)，则FOLLOW(𝑋) = FOLLOW(𝑋) ∪ FOLLOW(𝐴)。

由于两个非终结符的 FOLLOW集完全有可能互相包含，此时若在求其 中一个的 FOLLOW直接递归求另一个的 FOLLOW集，则会出现无穷 递归的死循环。引入

follow\_include用于存储非终结符之间FOLLOW集的包含关系，在自己包含对方情况下，先判断对方若也包含自己，则停止递归，在所有推导结束后，互相包含的非终结符FOLLOW集取并集

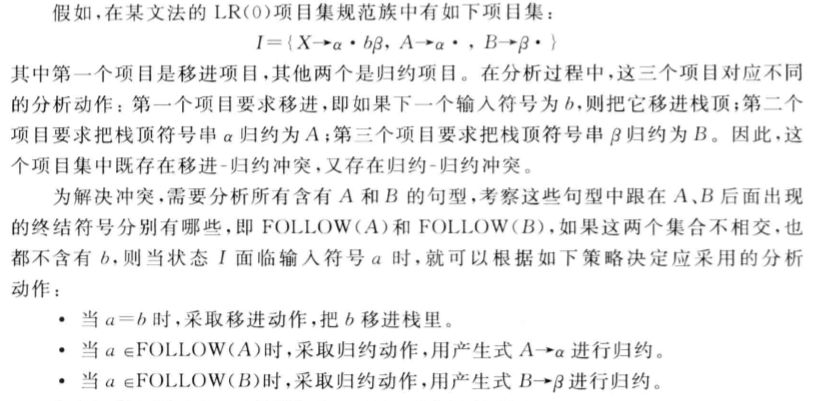
### 算法实现：



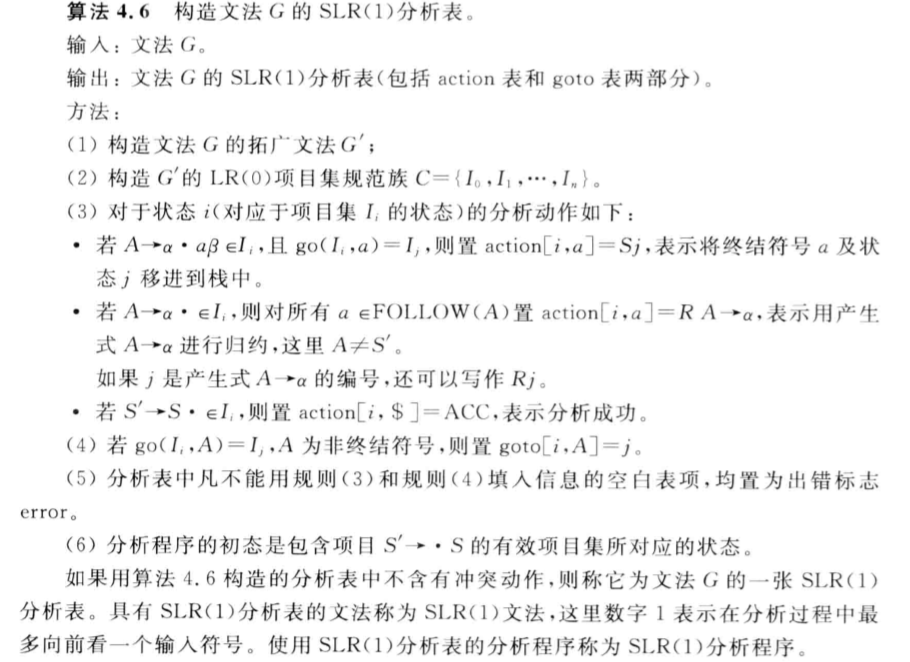


## 6.预测分析表

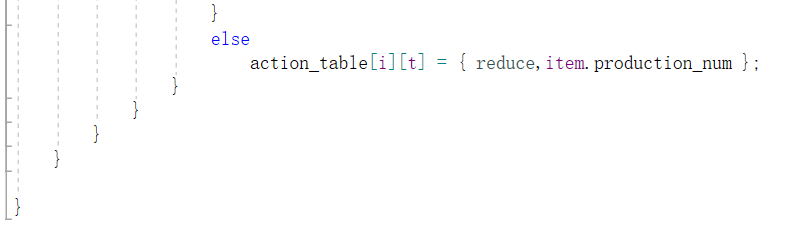
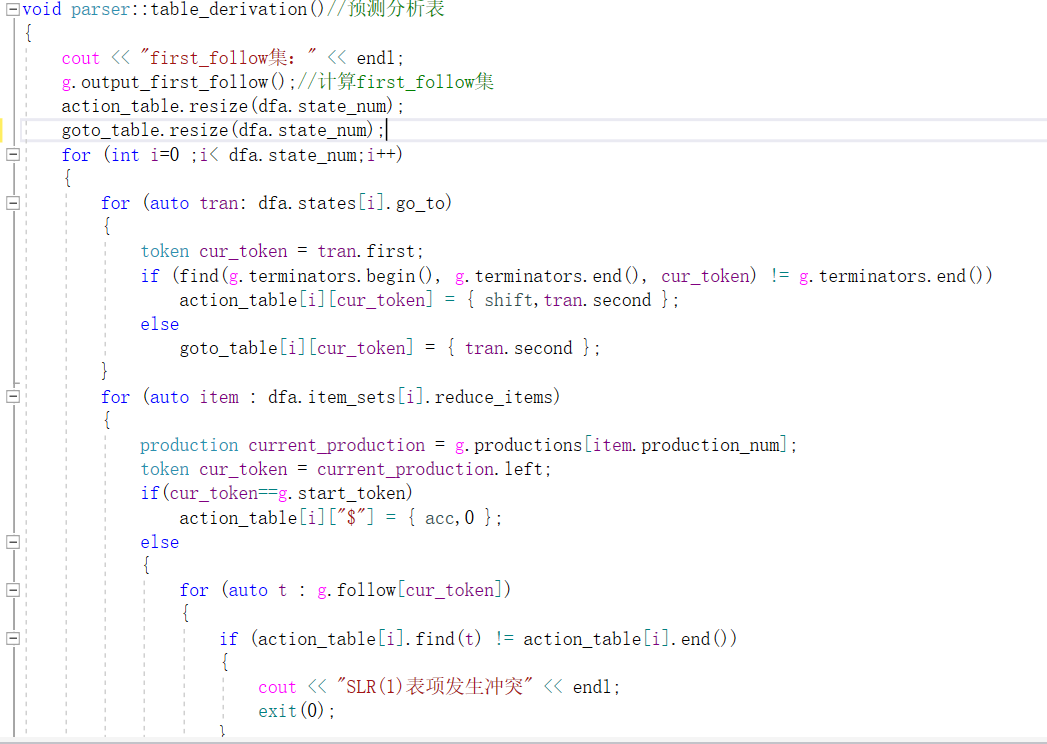
### 算法原理：



伪代码如下：

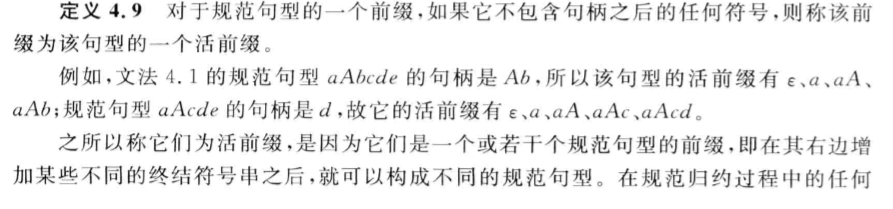


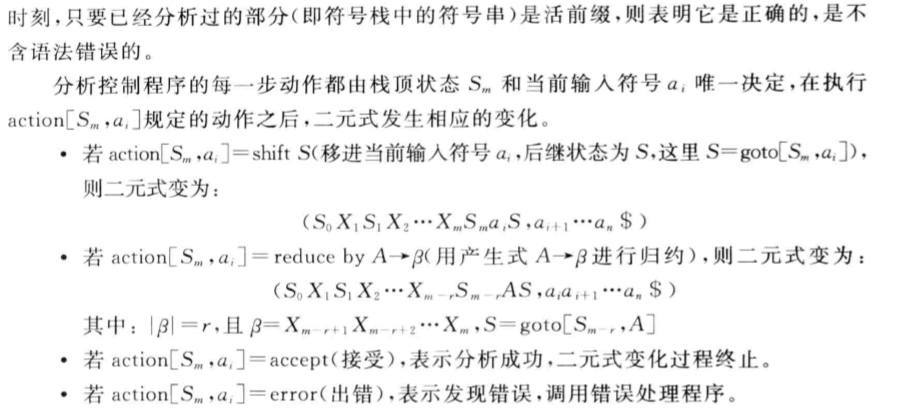
### 算法实现：



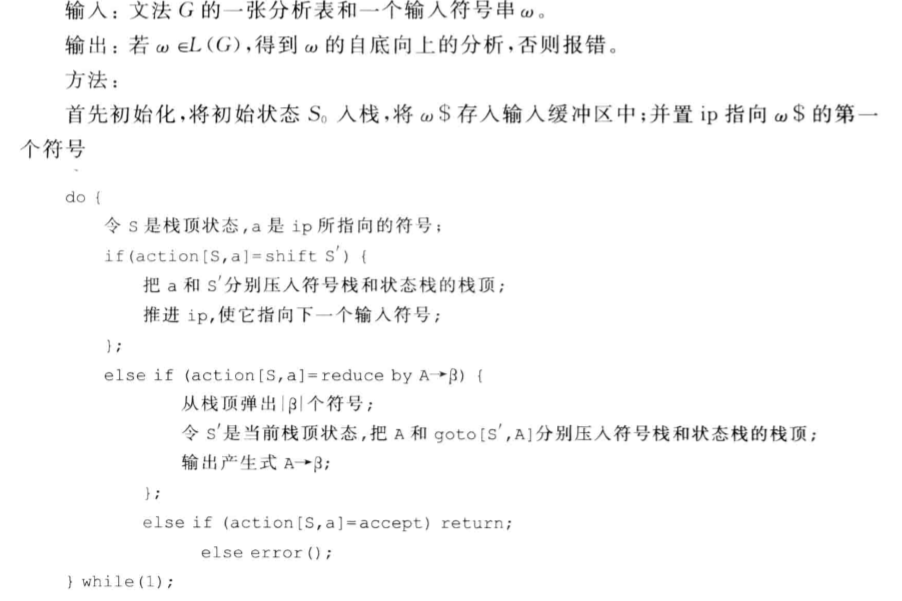
## 7.预测分析程序

### 算法原理：



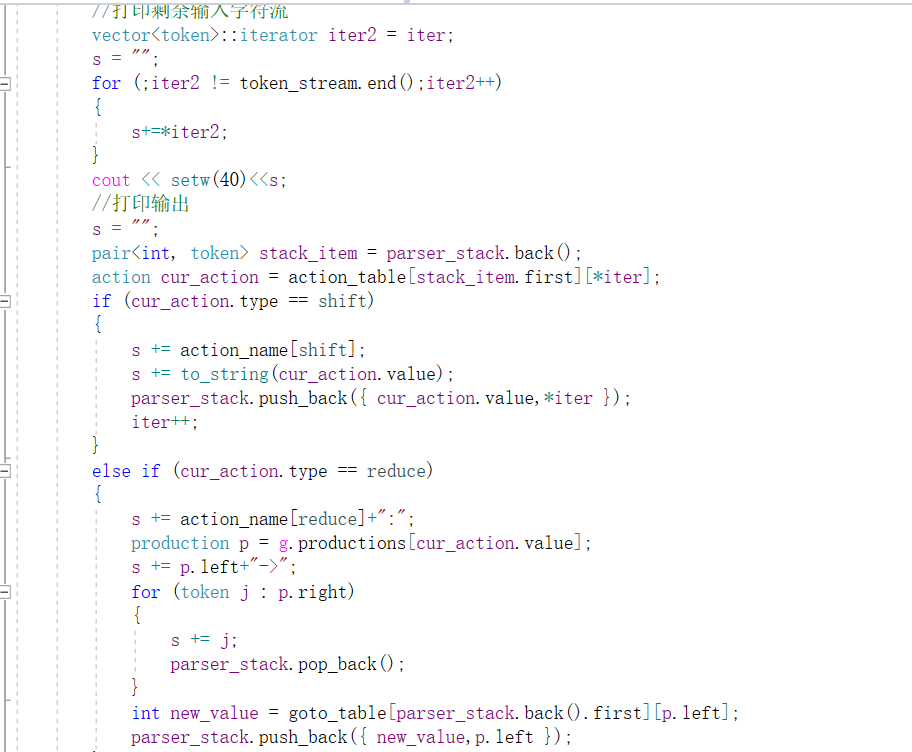


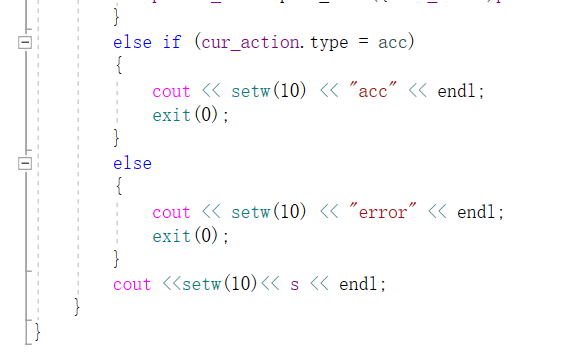
伪代码如下：



### 算法实现：



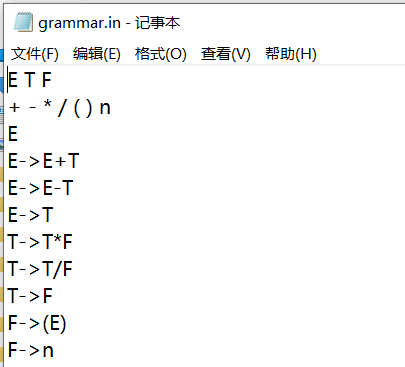




# 五．测试样例

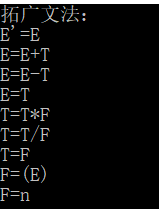
## 1.文法输入

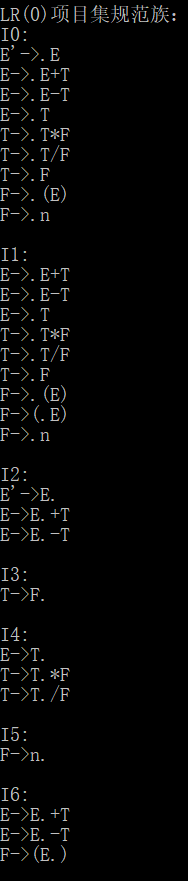
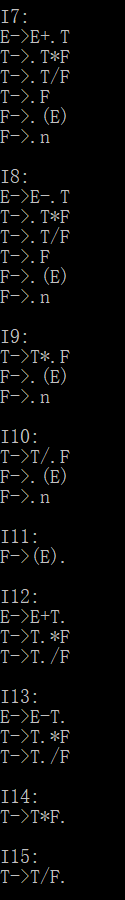
通过grammar.in文件读入所需识别的文法

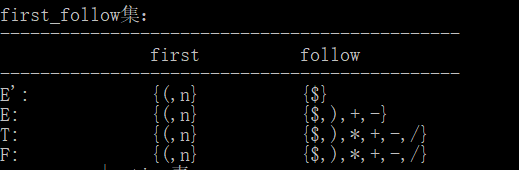


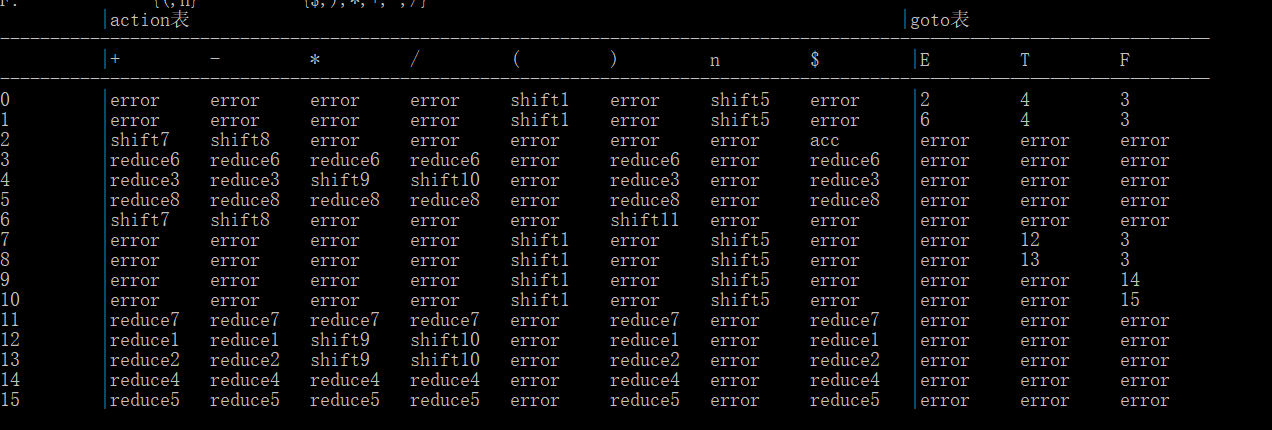
输入一行非终结符，一行终结符，一行文法开始符，接下来若干行文法的产生式。

## 2.文法分析结果输出



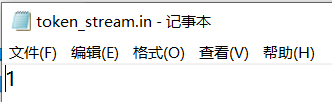
 

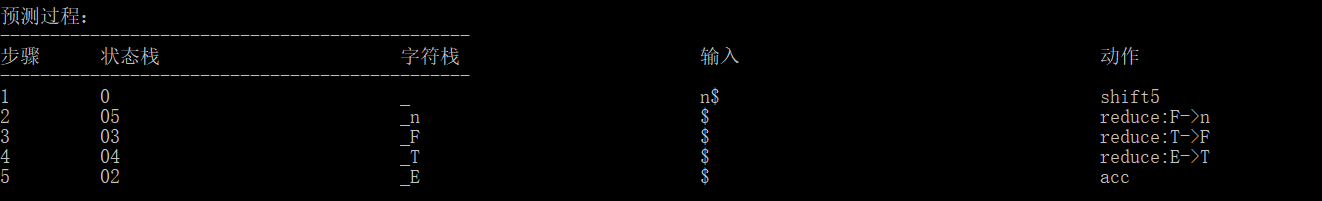


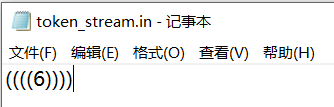


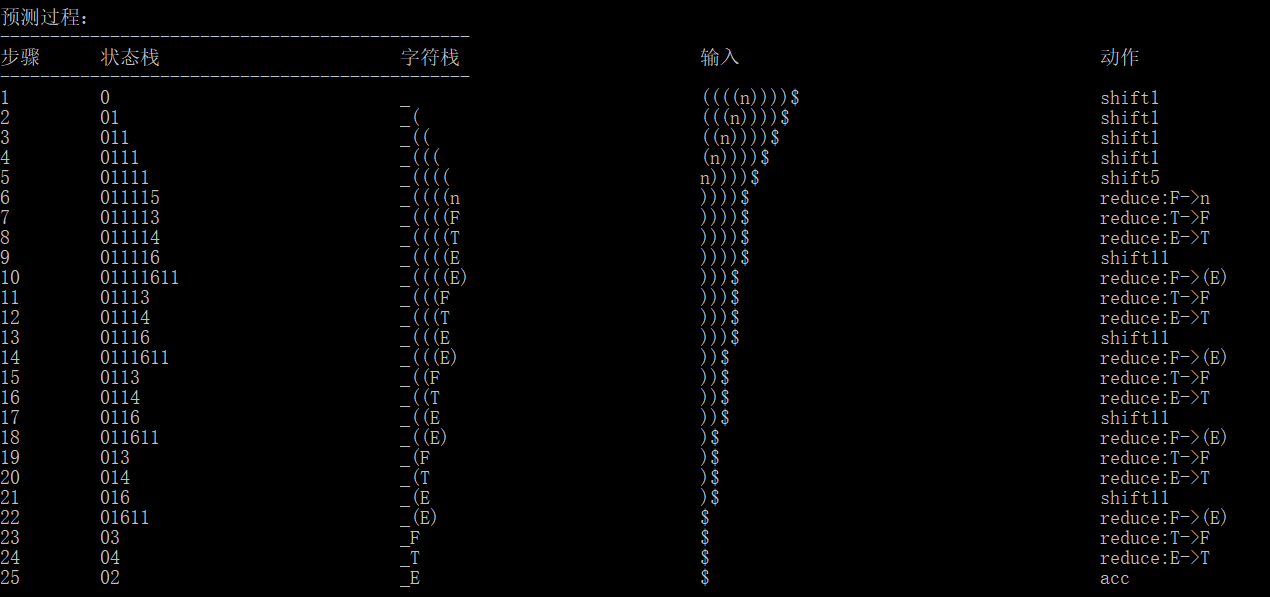
## 3.字符流输入与结果输出

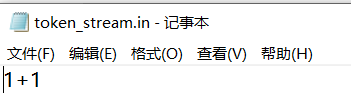
通过token\_stream.in文件读入所要识别的字符流（由于本次实验未链接词法分析器，所以为了不必要的处理，将num更改为n）

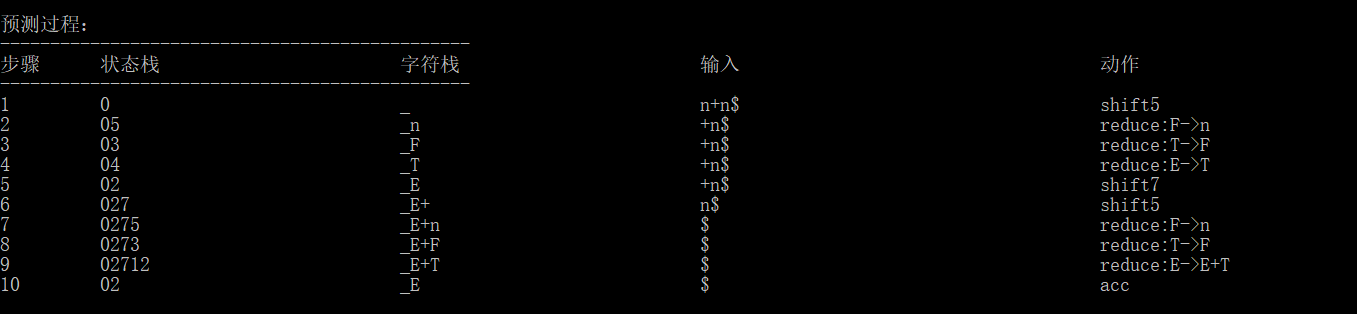


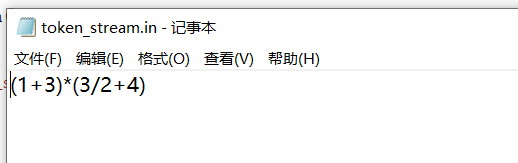


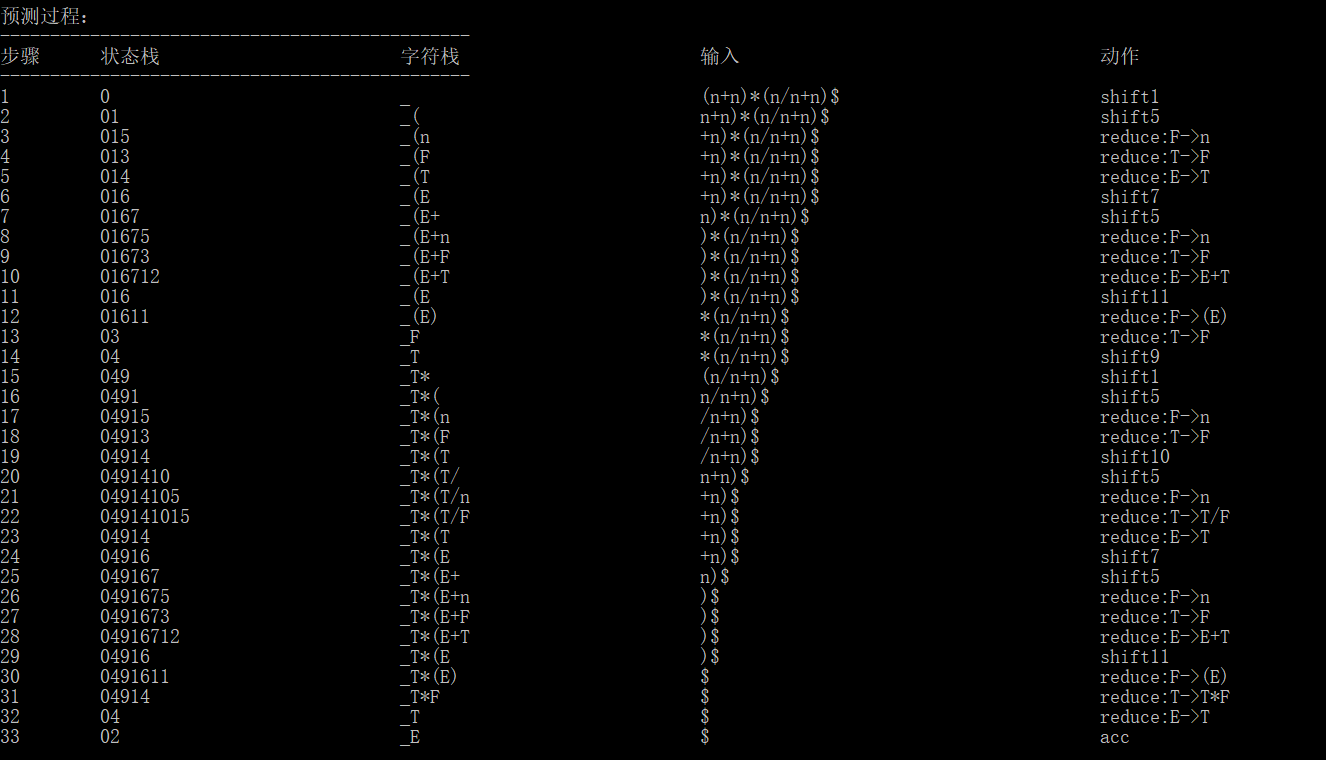




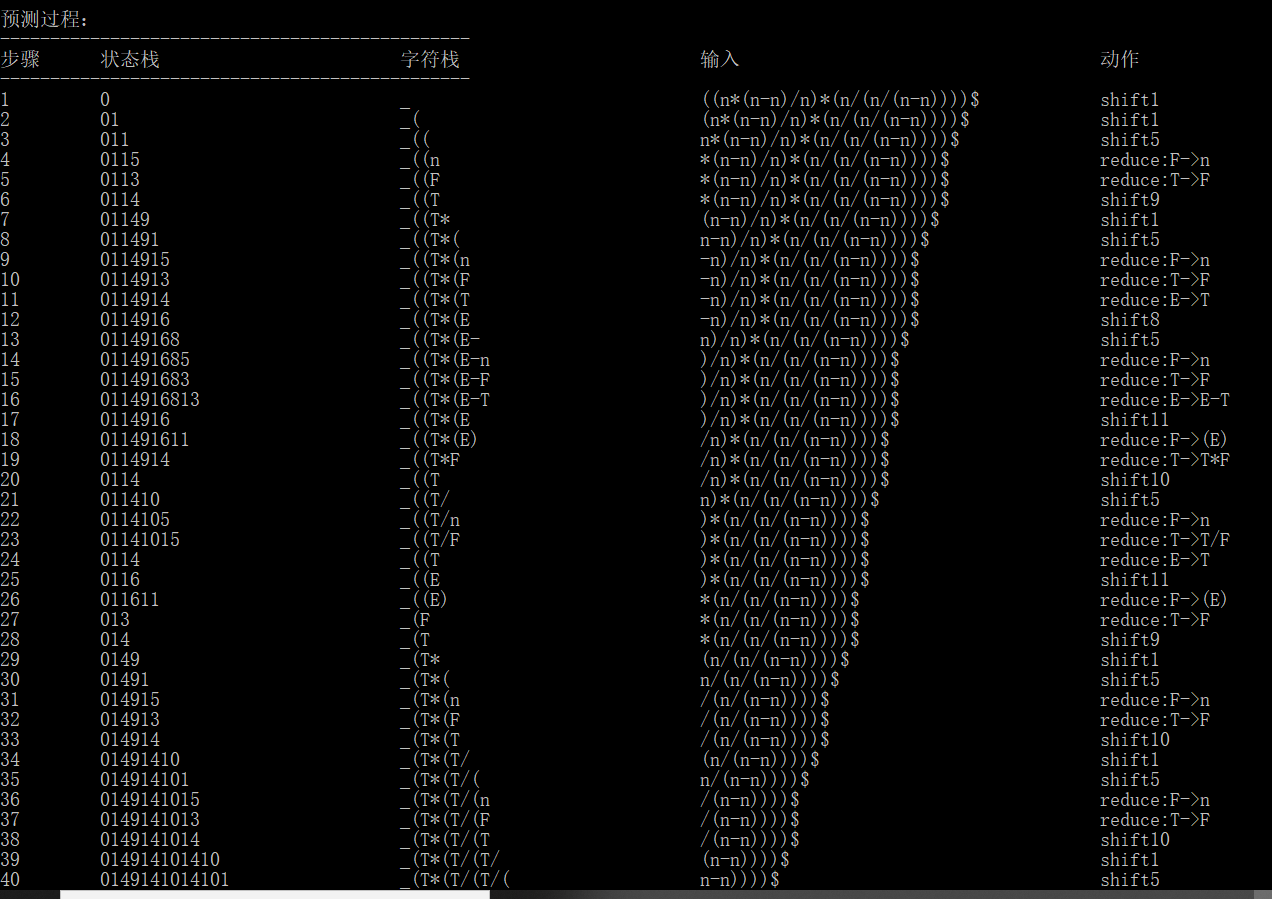


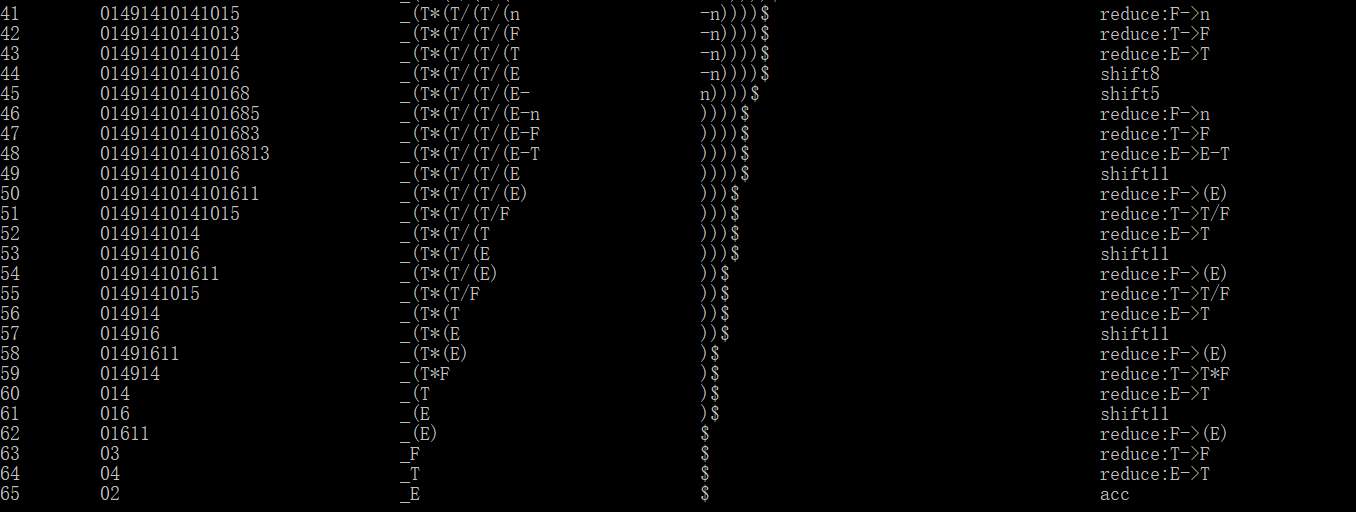












# 六．实验总结

## 1.难点攻克：

### a.含左递归的first集

与LL分析中first集求法不同的就是，可能会因为左递归的存在，无法判定其能否推空而考虑其后继符号，因此引入推空集的算法。

而在求解符号能否推空时也遇到了困难，但抓住一些特征，如只要含自身的产生式能否推空与非终结符能否推空布尔值一样，而能间接推空的产生式也一定全为能推空的非终结符组成，因此此问题迎刃而解。

### b.类内部的比较

由于整体分析器所需设计的结构较为复杂，将大部分数据结构都存储在STL容器中，虽然使模型在逻辑上得到了简化，但却为实际编程过程造成了不少的麻烦，其中一大问题就是，设计的数据结构包含大量属性，但由于需要自动排序的必要，需要自己重载比较运算符，而达到容器可顺利使用的问题。

### c.迭代器失效问题

在对于一个容器中元素进行遍历时，经常会改动容器中内容，进而使迭代器失效，在多次尝试后，发现即使使用“=”创建临时变量也只是浅拷贝，无法实际解决问题，最后通过自己实现深拷贝完成对迭代器的替换。

### d.数据结构过于复杂问题

在起初对数据结构进行设计使，原本将项目集元素与器后继状态定义在一起，但是数据结构过于复杂，繁琐了编程过程，因此将他们分开存入vector顺序容器中，并通过下标将他们练习在一起。

### e.后继状态问题

起初的是先直接算出项目集，再对后继可能产生状态逐一推导，经过改良，为Item\_set增加trans属性，在推导项目集过程中就算出后继可能项目集的自己，再通过闭包运算可直接获得后继状态，大大提高了运算效率

## 2.实验心得：

从测试结果可看出，DFA状态均正确，预测分析表正确，对于正确的表达式，语法分析输出结果也是正确的，。

在编译流程中，LR语法分析与LL语法分析的有所异同，其分析的基础是识别活前缀DFA，FOLLOW集为辅，通过输入“被动“预测，自底向上归约分析树，而LL语法分析核心是FIRST集，FOLLOW集为辅，通过文法开始符号“主动”预测，根据输入自顶向下展开分析树。

编写语法分析程序的过程中遇到了许多问题，如如何设计数据结构，运用了大量的容器知识，深入了解了LR语法分析的各个步骤如何实现与其中的细节问题。本次实验无论是对理论课知识，还是编程能力，都有着不小的提升。