**目录**

**一、实验内容及需求分析**

1.1 实验内容

1.2 程序功能

1.3 程序输入

1.4 输出形式

1.5 测试数据

**二、概要设计**

2.1 任务的分解

2.2 数据类型定义

2.2.1 词法分析器类型定义

2.2.2 语法分析器类型定义

2.3 主程序流程图

2.4 模块间的调用关系

**三、详细设计**

3.1 词法分析器设计方法

3.1.1 词法分析器的任务

3.1.2 词法分析方法

3.1.3 词法分析器的输出

3.2 语法分析器设计方法

3.2.1 LR(1)分析方法介绍

3.2.2 LR(1)分析表构造方法

3.2.3. 求ACTION和GOTO表

3.2.4 调用词法分析器进行分析

**四、调试分析**

4.1 测试数据与测试输出结果

4.2 时间复杂度分析

4.3 调试时出现问题

**五、总结与收获**

5.1 项目收获

5.2 对于解决问题过程的思考

5.3 对课程的认识

**六、参考文献**

**一、实验内容及需求分析**

**1.1. 实验内容**

本实验要求根据 LR(1)分析方法，编写一个类C语言的 LR(1)语法分析程序：

(1) 根据LR(1)分析方法，编写一个类C语言的语法分析程序，可以选择以下两项之一作为分析算法的输入：

a. 直接输入根据已知文法人工构造的ACTION表和GOTO表

b. 输入已知文法，由程序自动生成该文法的ACTION表和GOTO表

(2) 语法分析程序要能够调用词法分析程序，并为后续调用语义分析模块做考虑。

(3) 对输入的一个文法和一个单词串，程序能正确判断此单词串是否为该文法的句子，并要求输出分析过程。

**1.2. 程序功能**

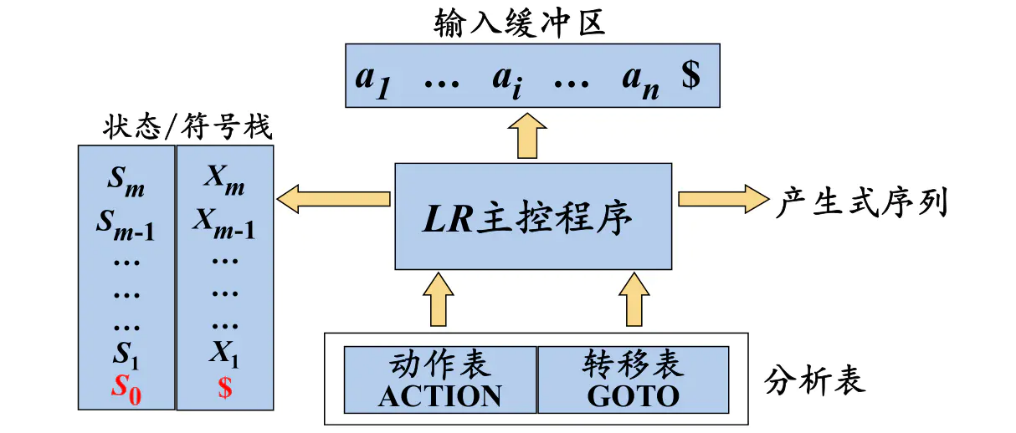
本程序使用 LR(1)分析方法编写，在指定的文本文件中放入文法，可以自动生成对应的语法分析程序。

向本程序输入已知文法的产生式以及源程序，本程序负责进行词法分析和语法分析，并输出语法分析结果，包括：

1. 是否通过语法分析

2. 若通过语法分析，则输出源程序的分析过程和对应的ACTION\GOTO表

3. 若无法通过语法分析，输出报错信息



**1.3. 程序输入**

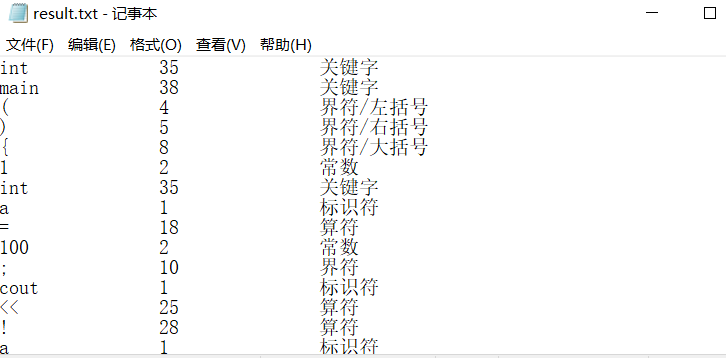
本程序的输入分为两部分，分别是类C语言文法产生式和源程序。该两部分均以文件形式输入。

类C语言文法的输入格式为终结符/产生式左部 → 产生式右部。

源程序则为类C语言文法。

**1.4. 输出信息**

词法分析器的输出为分析类C程序后的字符集，包括原始数据、表内次序和对应类型。



语法分析器的输出则为源程序的分析过程和对应的ACTION\GOTO表。

**1.5. 测试数据**

在 源代码目录下给出了test.txt(文本) 和 text.txt(文法)，作为测试文件。具体的测试细节参考调试分析与结果展示部分。

**二、概要设计**

**2.1. 任务分解**

任务采用 LR(1)分析方法进行设计，分为词法分析和语法分析。

在词法分析部分，程序读入源程序文件，并按照词法规则对源程序进行整理，

使得语法分析器可以更方便地处理输入串。

语法分析部分，程序首先读入文本中的文法产生式，由产生式构造出对应的 ACTION表和 GOTO 表；每当语法分析器需要读输入串时就调用词法分析器，获得一个输入符号，并根据 LR(1)分析表进行相应移进、归约、接受、报错动作，直到源程序分析完毕或者出现错误为止。词法分析器是语法分析器的子模块，每当语法分析器需要下一个符号时，便去词法分析器处取得。

文法文件

输入

语法分析器

返回

调用

输入

源程序文件

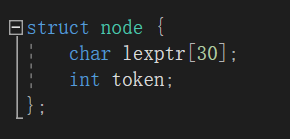
词法分析器

**2.2. 数据类型定义**

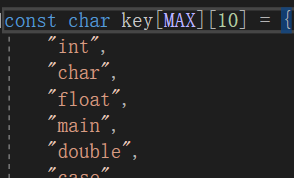
**2.2.1. 词法分析器类型定义**

使用两个文件指针控制源程序的读入和分析结果的写入。

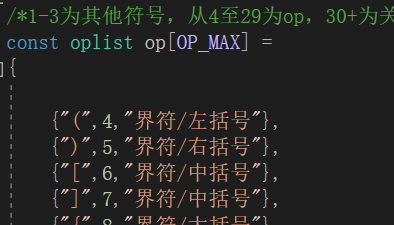
在头文件中给定了分析表的节点定义，为“类型 键值”的组合：



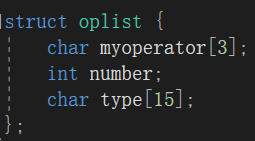
给定了关键字的类型：



界符定义：



符号表定义



**2.2.2. 语法分析器类型定义**

文法非终结符和终结符定义：

CSS products[MAX\_NUM]; //产生式

set<string> VT; //终结符

set<string> VN; //非终结符

map<string, int> mark\_Follow; //用于标记Follow 防止套娃

int p\_num = 0; //产生式数量

int I\_count = 0; //记录LR1项目数

vector<CSS\_LR1> I[MAX\_Count]; //项目集

内部结构定义：

class LR\_1

{

CSS products[MAX\_NUM]; //产生式

set<string> VT; //终结符

set<string> VN; //非终结符

map<string, int> mark\_Follow; //用于标记Follow 防止套娃

int p\_num = 0; //产生式数量

int I\_count = 0; //记录LR1项目数

vector<CSS\_LR1> I[MAX\_Count]; //项目集

map<string, int> GOTO[MAX\_Count];

map<string, string> ACTION[MAX\_Count];

fstream output; //输出文件

public:

LR\_1();

void init();

bool read\_wenfa(string filename); //读入文法

//工具函数

void print\_date(); //测试用

//过程函数

set<string> get\_first(string x); //获取非终结符的FIRST集合

set<string> get\_follow(string x); //获取非终结符的FPLLOW集合

vector<CSS\_LR1> CLOSURE(CSS\_LR1 I); //生成closure集

void LR1\_Analyse();

//IO函数

void showI(vector<CSS\_LR1> I); //展示项目集

void print\_ACTION\_GOTO(); //打印goto表

void Input\_Analyse(); //输入字符串并分析

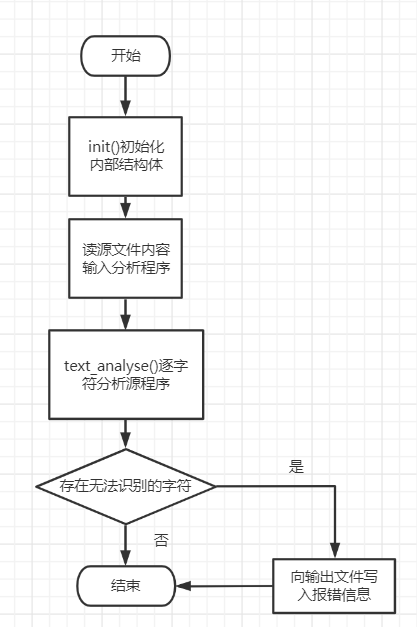
void print\_line(); //输出分隔符号

};

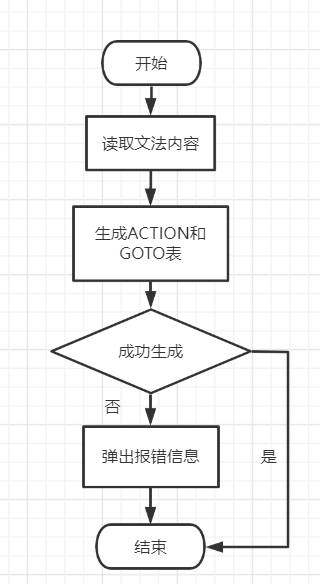
**2.3. 主程序流程图**

主程序流程图如下。

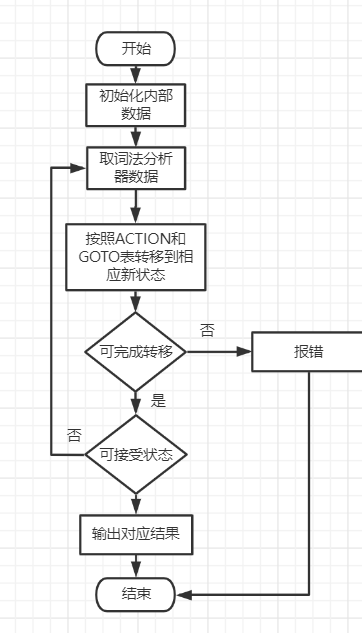
首先是对源文件词法的分析：



接着是LR(1) ACTION表和GOTO表的生成：



最后是词法分析器调用语法分析器进行分析：



**三、详细设计**

**3. 词法分析器设计方法**

**3.1. 词法分析器任务**

词法分析（英语：lexical analysis）是计算机科学中将字符序列转换为记号（token）序列的过程。进行词法分析的程序或者函数叫作词法分析器（lexical analyzer，简称lexer），也叫扫描器（scanner）。词法分析器一般以函数的形式存在，供语法分析器调用。

这里的记号是一个字串，是构成源代码的最小单位。从输入字符流中生成记号的过程叫作记号化（tokenization），在这个过程中，词法分析器还会对记号进行分类。词法分析器通常不会关心记号之间的关系（属于语法分析的范畴），举例来说：词法分析器能够将括号识别为记号，但并不保证括号是否匹配。语法分析器读取输入字符流、从中识别出语素、最后生成不同类型的记号。其间一旦发现无效记号，便会报错。

词法分析的第一阶段即词法分析器（扫描器），通常基于有限状态自动机。扫描器能够识别其所能处理的记号中可能包含的所有字符序列（单个这样的字符序列即前面所说的“语素”）。例如“整数”记号可以包含所有数字字符序列。很多情况下，根据第一个非空白字符便可以推导出该记号的类型，于是便可逐个处理之后的字符，直到出现不属于该类型记号字符集中的字符（即最长一致原则）。

**3.2. 词法分析方法**

语法分析过程模拟DFA进行。

本实验中类C 语言的词法规则如下：

标识符： 字母(字母|数字)\*

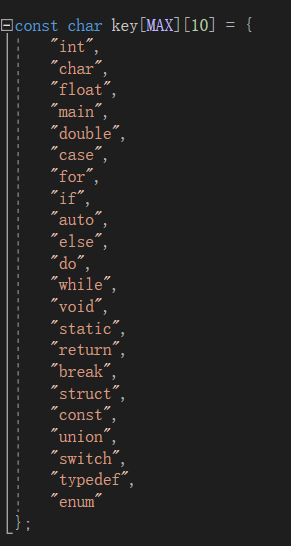
数值：数字\*，包含浮点数

字母：a |....| z | A |....| Z

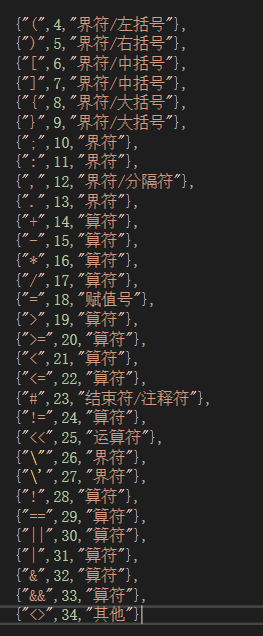
数字：0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

结束符：#

关键字：



界符和算符等其他字符：

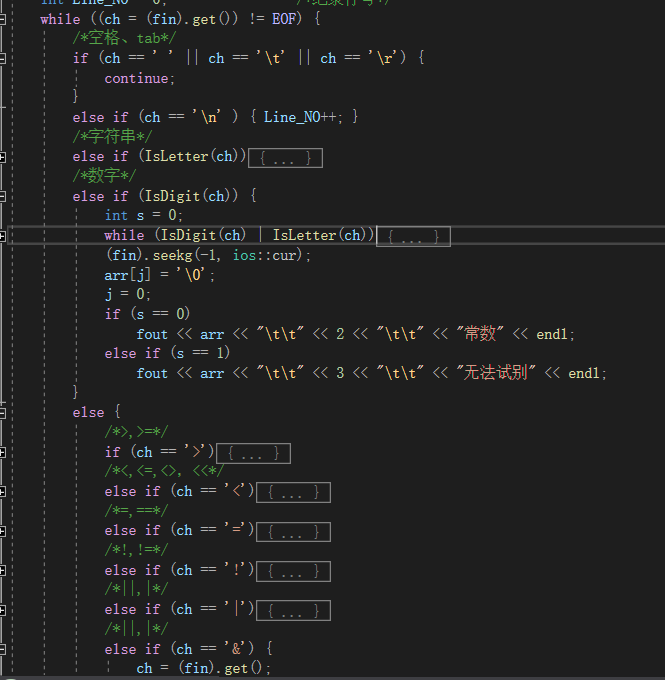


由于本实验中类 C 语言的词法规则较为简单，故直接使用 if-else 语句模拟 DFA 分析过程。

程序从fin.get()输入流中不断获取字符，并根据输入的字符判断当前为何种类型或是继续读取下一个字符加以判断。判断过程需要调用相应的判断函数，如Isletter()/IsDigit()/IsOp()。

对于含有两个单字符的符号，如‘<=’需要进行预处理，看下一位是否为对应的字符，若是则输出对应信息，不是则回退一个字符。对于输入数字为实数的情况，在调用完IsDigit()判断前串为数字的前提下判断是否为小数点；对于//和/\*\*/的注释接用上述方法预先判断即可除去对应字符。

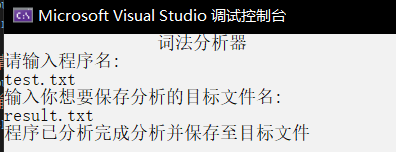
由于篇幅问题这里只贴部分函数代码。

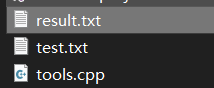


代码细节可参考github上的repo: <https://github.com/vo-lar/lexical-analyzer.git>

**3.1.3. 词法分析器的输出**

输出的分析结果暂时存放在指定的文件中，只执行词法分析器可以在同目录下看到结果。





**3.2. 语法分析器设计方法**

在计算机科学和语言学中，语法分析（英语：syntactic analysis，也叫 parsing）是根据某种给定的形式文法对由单词序列（如英语单词序列）构成的输入文本进行分析并确定其语法结构的一种过程。

语法分析器（parser）通常是作为编译器或解释器的组件出现的，它的作用是进行语法检查、并构建由输入的单词组成的数据结构（一般是语法分析树、抽象语法树等层次化的数据结构）。语法分析器通常使用一个独立的词法分析器从输入字符流中分离出一个个的“单词”，并将单词流作为其输入。实际开发中，语法分析器可以手工编写，也可以使用工具（半）自动生成。

**3.2.1. LR(1)分析法**

为克服SLR(1)分析法的不足，引入LR(1)对文法进行分析。

* 重新定义项目，使得每个项目都附带有k个终结符。每个项目的一般形式是[A→α·β, a1a2…ak] ，这样的一个项目称为一个LR(k)项目。项目中的 a1a2…ak 称为它的向前搜索符串(或展望串)。
* 向前搜索符串仅对归约项目[A→α·，a1a2…ak]有意义。对于任何移进或待约项目[A→α·β, a1a2…ak], β≠ε，搜索符串 a1a2…ak 没有作用。

归约项目[A→α·, a1a2…ak]意味着：当它所属的状态呈现在栈顶且后续的k个输入符号为 a1a2…ak 时，才可以把栈顶上的α归约为A, 其重点时需要向前搜索(展望)一个符号以确定“移进”或“归约”。

简单来说，LR(1)分析法作为一种自下而上的语法分析方法从输入串开始利用栈，当前输入字符和下一个字符寻找句柄进行移进-规约操作完成相应的语法分析，分析器参考分析表上的动作进行状态转移并最后进行判断，接受该字符串或是报错提醒，其难点在于如何构造出对应的文法分析表。

**3.2.2 LR(1)分析表构造方法**

LR(1)的分析核心时构造DFA构造出分析表的转移状态，

为构造LR(1)，需要两个函数CLOSURE和GO

其中CLOSURE(l)的理论构造方法为：

1. I的任何项目都属于CLOSURE(I)。

2. 若项目[A→α·Bβ, *a*]属于CLOSURE(I)，B→ξ 是一个产生式，那么，对于FIRST(β*a*) 中的每个终结符*b*，如果[B→·ξ, *b*]原来不在CLOSURE(I)中，则把它加进去。

3. 重复执行步骤2，直至CLOSURE(I)不再增大为止。

对于GO，

令I是一个项目集，X是一个文法符号，函数GO(I，X)定义为：

GO(I，X)＝CLOSURE(J)

其中

J＝{任何形如[A→αX·β, *a*]的项目| [A→α·Xβ, *a*]∈I}

其构造算法为：

BEGIN

C:={CLOSURE({[S′→·S，#]})};

REPEAT

FOR C中每个项目集I和G′的每个符号X DO

IF GO(I，X)非空且不属于C，THEN

把GO(I，X)加入C中

UNTIL C不再增大

END

得到上述函数之后，可得到LR(1)分析表算法，即令每个Ik的下标k为分析表的状态，令含有[S′→·S, #]的Ik的k为分析器的初态，此后便能使用ACTION表和GOTO表得到相应的分析表，具体步骤参考3.2.3 。

**3.2.3. 求ACTION和GOTO表**

构造方法：

动作ACTION和状态转换GOTO构造如下：

1. 若项目[A→α·aβ, b]属于Ik且GO(Ik, a)＝Ij， a为终结符，则置ACTION[k, a]为“ “sj”

2. 若项目[A→α·，a]属于Ik，则置ACTION[k, a]为 “rj”；其中假定A→α为文法G′的第j个产生式。

3. 若项目[S′→S·, #]属于Ik，则置ACTION[k, #]为 “acc”。

4. 若GO(Ik，A)＝Ij，则置GOTO[k, A]=j。

5. 分析表中凡不能用规则1至4填入信息的空白栏均填上“出错标志”。

构造代码如下：

void LR\_1::LR1\_Analyse() {

CSS\_LR1 p;

//初始项目 S’->.S ,#

p.start = products[0].start + "^";

p.num = 0;//点在最前面

p.tail.push\_back("#");

p.next.push\_back(products[0].start);

I[0] = CLOSURE(p);//求闭包后的I[0]

I[0].insert(I[0].begin(), p);

I\_count = 1;

//计算项目集

for (int i = 0; i < I\_count; i++) {//每个项目集 项目集I(i)

output << "===================" << endl;

output << "现在在计算项目集I" << i << endl;

showI(I[i]);//展示项目集

output << "===================" << endl;

//---------求ACTION的r部分--------------

vector<CSS\_LR1>::iterator t;

for (t = I[i].begin(); t != I[i].end(); t++) {

CSS\_LR1 t2 = \*t;

if (t2.num == t2.next.size()) {

int num = 0;

for (int xp = 0; xp < p\_num; xp++) {

if (products[xp].start == t2.start && products[xp].next == t2.next) {

num = xp;

break;

}

}

std::stringstream ss;

ss << num;

string s = ss.str();

for (int q = 0; q < t2.tail.size(); q++) {

ACTION[i][t2.tail[q]] = "r" + s;

}

if (t2.num == 1 && t2.next[0] == products[0].start) {

ACTION[i]["#"] = "acc";

}

}

}

set<string>::iterator it;

for (it = VN.begin(); it != VN.end(); it++) { //每个非终结符

vector<CSS\_LR1> temp;

for (int j = 0; j < I[i].size(); j++) {

CSS\_LR1 lr = I[i][j];

if (lr.num < lr.next.size() && lr.next[lr.num] == \*it) {

//cout<<\*it<<endl;

vector<CSS\_LR1> t2;

lr.num++;

t2 = CLOSURE(lr);

t2.push\_back(lr);

temp = temp + t2;

}

}

//cout<<"temp.size"<< temp.size()<<endl;

if (temp.size() > 0) {

int k;

for (k = 0; k < I\_count; k++) {//找一找项目集是否已经存在

if (cmp\_vector(I[k], temp)) {

break;

}

}

if (k == I\_count) {

//产生了新的项目集

I[I\_count] = temp;

output << " I" << i << " -- " << \*it << "->" << "I" << I\_count << endl << endl;

GOTO[i][\*it] = I\_count;//更新goto表

I\_count++;

}

else {

//项目集已经存在，需要自己指向自己

output << " I" << i << " -- " << \*it << "->" << "I" << k << endl << endl;

GOTO[i][\*it] = k;

}

}

}

for (it = VT.begin(); it != VT.end(); it++) { //每个终结符

vector<CSS\_LR1> temp;

for (int j = 0; j < I[i].size(); j++) {

CSS\_LR1 lr = I[i][j];

if (lr.num < lr.next.size() && lr.next[lr.num] == \*it) {

vector<CSS\_LR1> t2;

lr.num++;

t2 = CLOSURE(lr);//闭包求出的结果不包含本身

t2.insert(t2.begin(), lr);

//showI(t2);

temp = temp + t2;

}

}

if (temp.size() > 0) {

int k;

for (k = 0; k < I\_count; k++) {//找一找项目集是否已经存在

if (cmp\_vector(I[k], temp)) {

break;

}

}

if (k == I\_count) {

//产生了新的项目集

I[I\_count] = temp;

output << " I" << i << " -- " << \*it << "->" << "I" << I\_count << endl << endl;

std::stringstream ss;

ss << I\_count;

string s = ss.str();

ACTION[i][\*it] = "S" + s;//更新AVTION表

I\_count++;

}

else {

//项目集已经存在，需要自己指向自己

output << " I" << i << " -- " << \*it << "->" << "I" << k << endl << endl;

std::stringstream ss;

ss << k;

string s = ss.str();

ACTION[i][\*it] = "S" + s;

}

}

}

}

}

**3.2.4 调用词法分析程序进行分析**

算法：

Step1: 读入源程序，设置状态栈和符号栈，初始栈底为#。

Step2: 使用词法分析器分析一个词，对于该词，查 对应函数 得到与其对应的行动（action），若为移进，则将该词法符号压入符号栈，并将 goto 的新状态压入状态栈；若为归约，则根据 goto 找到归约的产生式，根据产生式右部的长度将固定数量的符号从符号栈出栈，同时状态栈也出栈相同数量的状态，最后把产生式左部压入符号栈，并将新状态栈栈顶的状态对应的 goto 状态压入状态栈。

Step3: 重复执行 Step2，直至扫描到文件末尾并达到 accept 状态，或遇到错误提前退出。

代码：

void LR\_1::Input\_Analyse() {//输入句子，开始分析

vector<int> status;//定义状态栈

vector<string> sign;//定义符号栈

int step = 1; //步骤

string input;

cout << "请输入分析的字符串(请以#结尾)：";

cin >> input;//输入待分析的句子

input = input + "#";

status.push\_back(0);//把状态0入栈

//把#加入符号栈

sign.push\_back("#");

//输出初始栈状态

cout << setw(10) << "步骤" << setw(10) << "状态栈" << setw(10) << "符号栈" << setw(10) << "输入串" << setw(25) << "动作说明" << endl;

int s = 0;//初始状态

int oldStatus;//保存之前的状态

string input\_s; //获取初始符号

input\_s = input.substr(0, 1);

while (ACTION[s][input\_s] != "acc") {//如果action[s][input\_s] =="acc" ，则分析成功

//获取字符串

string str = ACTION[s][input\_s];

//如果str为空，报错并返回

if (str.size() == 0) {

cout << "出错";

return;

}

//获取S或r后面的数字

stringstream ss;

ss << str.substr(1);

ss >> s;//新的状态号

//如果是移进

if (str.substr(0, 1) == "S") {

cout << setw(10) << step << setw(10) << vectTrancStr(0, status, sign) << setw(10) << vectTrancStr(1, status, sign) << setw(10) << input << setw(10) << "A" << "CTION[" << status.back() << "," << input\_s << "]=S" << s << "," << "状态" << s << "入栈" << endl;

sign.push\_back(input\_s); //输入符号入栈

input.erase(0, 1);

status.push\_back(s);//将状态数字入栈

}

//如果是规约

else if (str.substr(0, 1) == "r") {

string kaitou;//产生式的头部

kaitou = products[s].start;

int pop\_num = products[s].next.size();//获取符号栈的出栈次数

string r;

stringstream ss;

ss << s;

ss >> r;

int oldStatus;//保存之前的状态

int status\_size = status.size();

oldStatus = status[status\_size - 1 - pop\_num];

s = GOTO[oldStatus][kaitou];

cout << setw(10) << step << setw(10) << vectTrancStr(0, status, sign) << setw(10) << vectTrancStr(1, status, sign) << setw(10) << input << setw(10) << (string)":产生式" + r + (string)"归约,GOTO(" << oldStatus << "," << kaitou << ")=" << s << "入栈" << endl;

//对符号栈进行出栈和状态栈进行出栈

while (pop\_num--) {

sign.pop\_back();

status.pop\_back();

}

sign.push\_back(kaitou);//再对产生式的开始符号入栈

status.push\_back(s);//再把新的状态入栈

}

else {

//nothing

}

step++; //步骤数加1

s = status.back();//获取栈顶状态

input\_s = input.substr(0, 1);//获取输入的字符

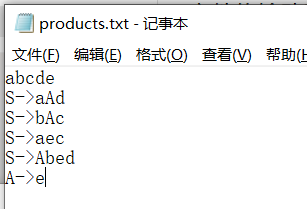
}

cout << setw(10) << step << setw(10) << vectTrancStr(0, status, sign) << setw(10) << vectTrancStr(1, status, sign) << setw(10) << input << setw(10) << "A" << "cc:分析成功" << endl;

}

**四、调试分析与结果展示**

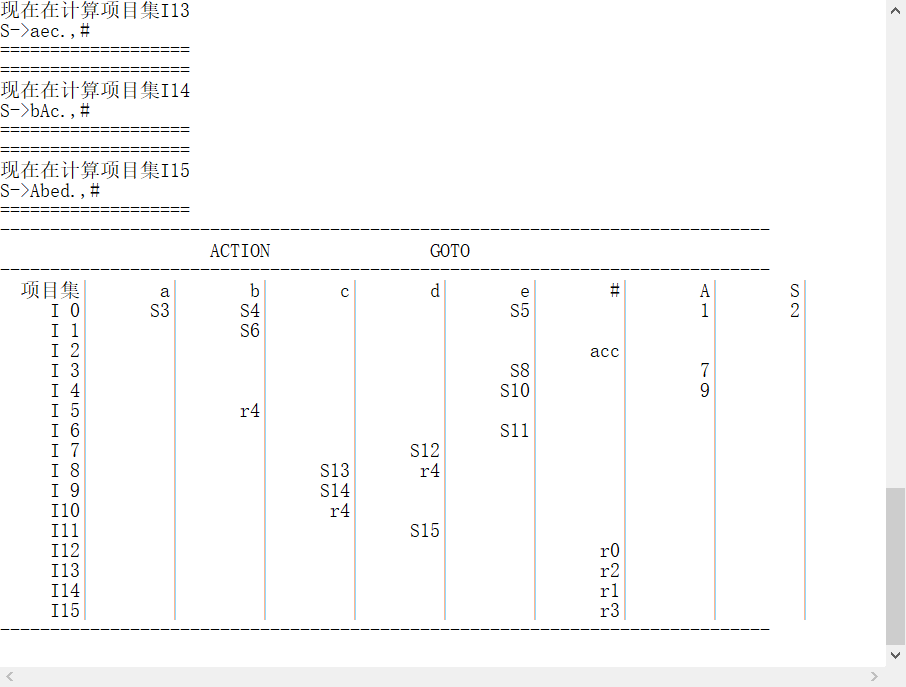
输入文法放入products.txt文件中：

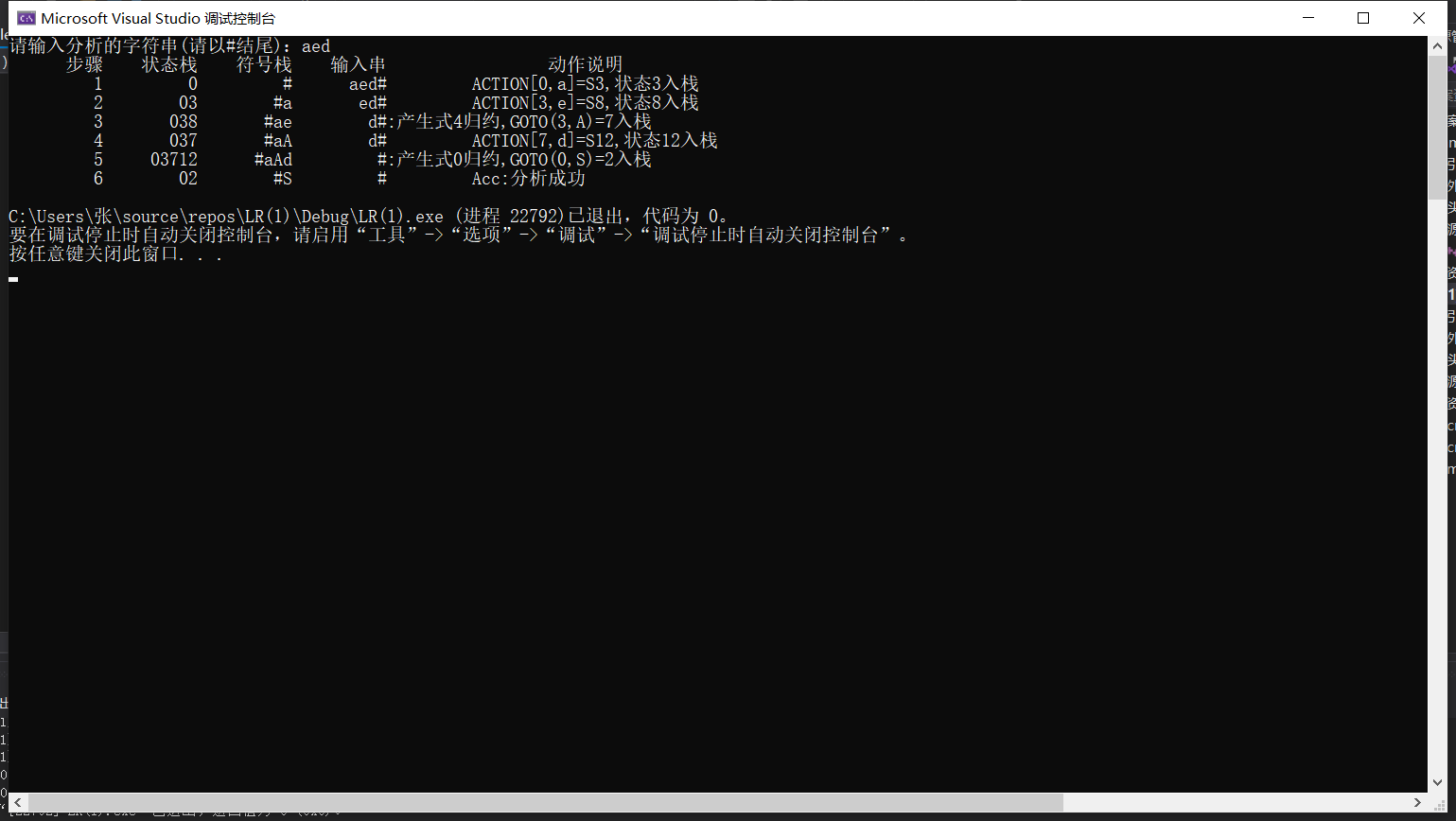


输出内容（文法，规范族集，action\_goto表）放入result.txt文件中。

输入与输出验证字符串在控制台进行：







**五、总结与收获**

**5.1 项目收获**

本次项目中，我们小组实现了DFA词法分析器和基于 LR(1)的语法分析器，可以用于对类 C 程序进行词法和语法分析，并输出分析过程结果。

词法分析方面，在基础上增加了对于常数的识别可以识别C语言下带有小数点的实数，增加了增加单词（如保留字、运算符、分隔符等）的数量，并在出错时会给出错误信息写入文件中，对于注释行也能进行识别并除去//和/\*\*/的影响。

对于语法分析，对于输入的文法能够得到对应的ACTION表和GOTO表并基于此生成对应的LR(1)语法分析器，能够打印出对应的分析步骤得到相应结果。

**5.2 对于解决问题过程的思考**

对于解决算法问题，我们注意到一些正确的封装接口和结构体的调用将会使得函数变得更为简洁，数据结构的设计将会决定函数的复杂程度，引入FILE指针或是fstream这样的结构体能极大减少代码量。除此之外，在头文件中设定相应的参数（如关键字）可以方便后续增加或是删除，而函数中调用这些函数时设置相应的动态偏移量能够使得编写更为灵活。

在项目开发方面，做好项目规划是很重要的。本次实验时间不足，有些细节部分并未很好完成，编写语法分析器时任务分工初期出错和疫情导致的干扰导致后期压力过大，存在较大失误。

**5.3 对课程的认识**

通过完成这个大作业，我们对编译原理词法分析器和词法分析器这两个部分有了更深的理解和认识。通过编写这两个部分的代码，我们对计算机编译程序环节的前两节的内部代码有了清晰的理解，对于DFA词法分析、FIRST集和FOLLOW集、LR（1）语法分析这几种最为常见的方式有了初步的编写经验，我们逐步认识到编译器一步步在做些什么，将这些知识点有机串联在一起，从而跳出课程算法的范围，在实践中学习。

本次项目开发不仅帮助我们复习、强化了编译原理的相关知识，并且锻炼了我们的设计和思考能力，也锻炼了编程综合方面的能力，跳出课堂之外进行学习训练，可谓是大有脾益。

**六、参考文献**

[1]陈火旺,刘春林.程序设计语言编译原理[M].北京:国防工业出版社,2000.1

[2] [维基百科：词法分析器](https://zh.wikipedia.org/wiki/%E8%AF%8D%E6%B3%95%E5%88%86%E6%9E%90)