**北 京 邮 电 大 学**

**实 验 报 告**

**课程名称：编译原理**

**实验名称：语法分析**

**学院：计算机学院**

**班级：2018211304**

**学号：2018211208**

**姓名：谢睿**

**教师：李文生**

**2020 年 11 月 21 日**

目录

[一．实验题目及要求 3](#_Toc2918)

[1. 实验题目：语法分析程序的设计与实现 3](#_Toc918)

[2. 实验要求： 3](#_Toc1686)

[3. 实现方法： 3](#_Toc22102)

[二．LL(1)文法程序设计与实现 3](#_Toc19080)

[1. 消除原文法左递归 ： 3](#_Toc14992)

[2. 求非终结符的FIRST集和FOLLOW集： 4](#_Toc24528)

[3. 算法4.2——自动构造预测分析表： 5](#_Toc15990)

[4. 算法4.1——LL(1)预测分析程序： 5](#_Toc27651)

[5. 程序中使用的主要变量以及结构体说明： 6](#_Toc10447)

[6. 程序中功能模块设计： 7](#_Toc31816)

[7. 程序逻辑流程： 7](#_Toc5556)

[8. 程序输出含义： 7](#_Toc14745)

[9. 程序测试： 8](#_Toc14940)

[三．LR(0)文法程序设计与实现： 11](#_Toc31728)

[1. 修改文法： 11](#_Toc4104)

[2. 求FIRST集和FOLLOW集： 11](#_Toc7155)

[3. 构造识别该文法所有活前缀的DFA： 11](#_Toc26511)

[4. 构造该文法的LR分析表： 12](#_Toc7893)

[5. 程序中使用的主要变量以及结构体说明： 13](#_Toc12833)

[6. 程序中功能模块设计： 14](#_Toc30355)

[7. 程序逻辑流程： 14](#_Toc28976)

[8. 程序输出含义： 15](#_Toc8366)

[9. 程序测试： 16](#_Toc3752)

[四．实验心得与总结 19](#_Toc11)

[1.实验心得： 19](#_Toc10204)

[2.实验总结： 19](#_Toc29360)

[五．附录：C++实现-代码 19](#_Toc26825)

[1. LL分析： 19](#_Toc27244)

[2. LR分析： 27](#_Toc17066)

# 一．实验题目及要求

## 1. 实验题目：语法分析程序的设计与实现

## 2. 实验要求：

编写语法分析程序，实现对算术表达式的语法分析。要求所分析算数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

实验要求：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。

## 3. 实现方法：

1） 方法2：编写LL(1)语法分析程序，要求如下。

(1) 编程实现算法4.2，为给定文法自动构造预测分析表。

(2) 编程实现算法4.1，构造LL(1)预测分析程序 。

1. 方法3：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。

1. 方法4：利用YACC自动生成语法分析程序，调用LEX自动生成的词法分

析程序。

# 二．LL(1)文法程序设计与实现

## 1. 消除原文法左递归 ：

原文法产生式含有左递归，故需要改写文法，使其满足LL(1)文法的定义：

|  |
| --- |
| E→TA  A→+TA  A→-TA  A→ε  T→FB  B→\*FB  B→/FB  B→ε  F→(E)  F→num |

值得注意的是，其中生成式右边的”ε”符号在程序中将使用不常用的符号”@”表示；num将使用单个字符”d”表示。以简化程序设计并防止系统字符集不兼容的问题。

## 2. 求非终结符的FIRST集和FOLLOW集：

构造分析表首先需要给定输入文法的FIRST集与FOLLOW集。

使用课本所写FIRST，FOLLOW集生成算法进行计算，

计算FIRST集程序的算法如下：

Step1. 如果，则。

Step2. 如果，且有产生式，，则把加入到中。若存在，则把也加入到中。

Step3.若存在产生式，，且，则把中除的一切符号加入到中。

Step4. 若，则把加入到中。

Step5. 如果有新加入的符号，则重复步骤1，2，3，4，直到没有新元素加入到集合中为止。

计算FOLLOW集程序的算法如下：

Step1. 把程序尾部符号加入到中，其中是文法开始符号。

Step2. 如果存在产生式，，则把中一切除的符号都加入到中。

Step3. 如果存在产生式，，则把的所有符号都加入到中。

Step4. 如果存在产生式，，且，则把的所有符号都加入到中。

Step5. 如果有新加入的符号，则重复步骤2，3，4，直到没有新元素加入到集合中为止。

最终得到FIRST，FOLLOW集如下：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | E | A | T | B | F |
| FIRST | ( , num | + , - ,ε | ( , num | \* , / ,ε | ( , num |
| FOLLOW | $ , ) | $ , ) | + , - , $ , ) | + , - , $ , ) | + , - , \* , / , $ , ) |

## 3. 算法4.2——自动构造预测分析表：

算法4.2实现预测分析表的自动构建

输入：文法G

输出：文法G的预测分析表M

for(文法G的每一个产生式){

for(每个终结符号)

把放入表项M[A,a]中;

if(ε∈FIRST(α))

for(每个b∈FOLLOW(A))

把放入表项M[A,b]中;

}；

for(所有无定义的表项M[A,a])

标上错误标志;

## **4. 算法4.1——LL(1)预测分析程序：**

利用算法4.1构造预测分析程序

输入: 输入符号串ω,文法G的一张预测分析表

输出: 若ω∈L(G)，则输出ω的最左推倒，否则报告错误

方法：

首先，初始化，即将$压入栈底，将文法开始符号S压入栈底；ω$放入输入缓冲区中，并置向前指针ip，指向ω$的第一个符号。

然后，预测分析控制程序根据分析表M对输入符号串ω做出自顶向下的分析，过程如下：

do{

令X是栈顶文法符号，a是ip所指向的输入符号；

if(X是终结符号或$)

{

if(X==a)

{

从栈顶弹出X;

Ip前移一个位置；

}

else

error();

}

else

{

if()

{

从栈顶弹出X;

依次把Yk,Yk-1,…,Y2,Y1压入栈；

输出产生式

}

else

error();

}

}while(X!=$)

## 5. 程序中使用的主要变量以及结构体说明：

5.1 宏定义：

1. #define TERM\_NUM 8              // 终结符数量
2. #define UNTERM\_NUM 5            // 非终结符数量
3. #define G\_NUM 10                // 产生式数量

5.2 结构体定义：

本次语法分析使用一个与上次词法分析相同的结构”WPRDS”，用于记录记号流信息。

1. // 储存记号流中每个词以及相关信息
2. **typedef** **struct** words
3. {
4. **char** sign;                  // 记号
5. string attribute;           // 属性
6. }WORDS;

5.3 全局变量说明：

1. vector<string> G\_LEFT;            // 储存产生式左边
2. vector<string> G\_RIGHT;           // 储存产生式右边
3. vector<string> TERM;           // 储存终结符
4. vector<string> UNTERM;            // 储存非终结符
5. vector<string> FIRST[TERM\_NUM];   // 所有非终结符的FIRST集
6. vector<string> FOLLOW[TERM\_NUM]; // 所有非终结符的FOLLOW集
7. vector<vector<string>> table;    // LL的预测分析表
8. vector<WORDS> input;           // 输入流(来自词法分析)

## 6. 程序中功能模块设计：

|  |  |
| --- | --- |
| 名称 | 功能 |
| int isTerm(string str) | 判断字符(串)str是否为终结符 |
| int isUnTerm(string str) | 判断字符(串)str是否为非终结符 |
| int findIndex(string ch) | 寻找非终结符所在索引号(即在vector UNTERM的下标) |
| int findIndexT(string ch) | 寻找终结符所在索引号(即在vector TERM的下标) |
| void initProgram() | 初始化函数，设定部分变量初值 |
| void getInput() | 从词法分析程序的输出获取记号流 |
| void outPutStack(vector<char> stack) | 输出栈内数据 |
| void outPutIn(int ip) | 输出当前输入串 |
| void genTable() | 算法4.2——构造分析表 |
| int LLanalyse() | 算法4.1——预测分析程序 |

## 7. 程序逻辑流程：

首先，进行程序初始化，设定FIRST集，FOLLOW集。为全局变量申请空间；

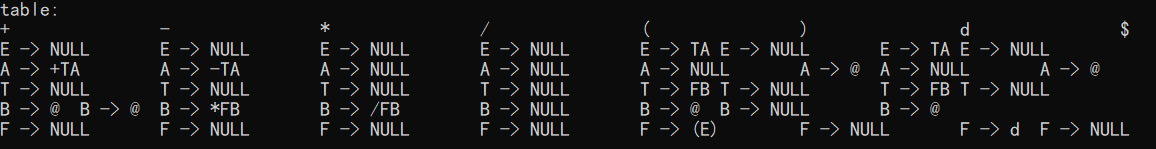
接着通过函数genTable()生成分析表，并将其输出便于检查；

下一步通过函数getInput()读取词法分析的输出文件，获取记号流；

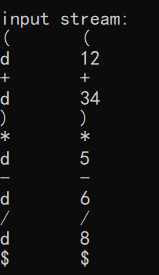
最后进行LL语法分析，并输出过程与结果。

## 8. 程序输出含义：

8.1 首先输出生成的分析表

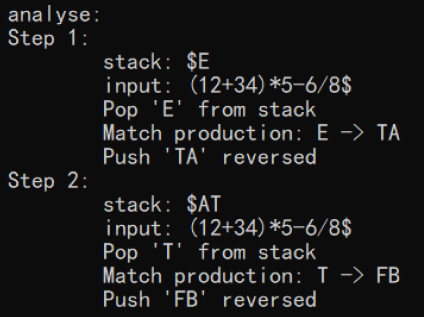


8.2 接着输出处理过的记号流

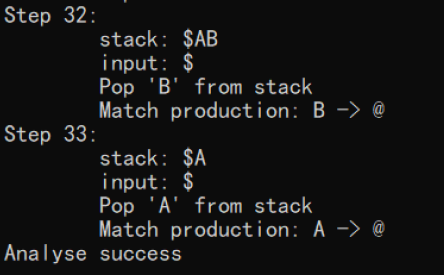


两列分别表示记号类型和记号属性，增加可拓展性，便于以后可能增加的功能。

8.3 最后输出分析过程：



...



## 9. 程序测试：

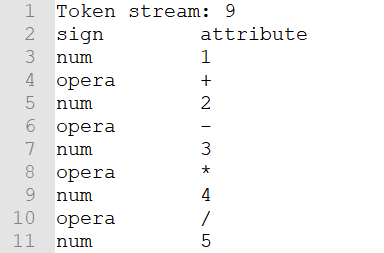
9.1 简单测试1：

包含四种运算符，不含括号，同时演示与词法分析的联动工作。之后的测试将直接展示语法分析过程。

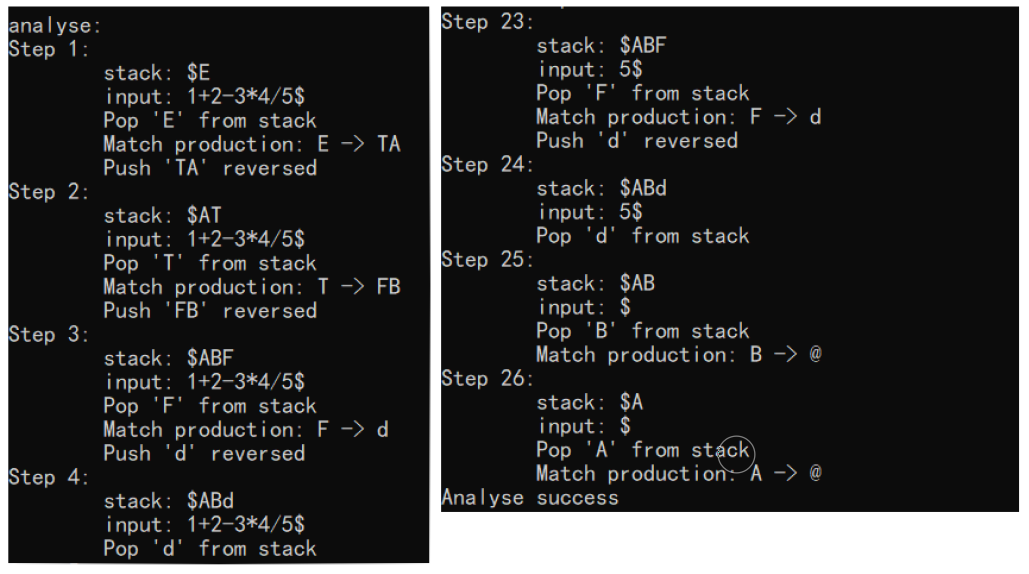
词法分析程序输入为：



输出记号流文件：



语法分析输出：



经过26步的分析，成功分析输入串，说明输入合法。

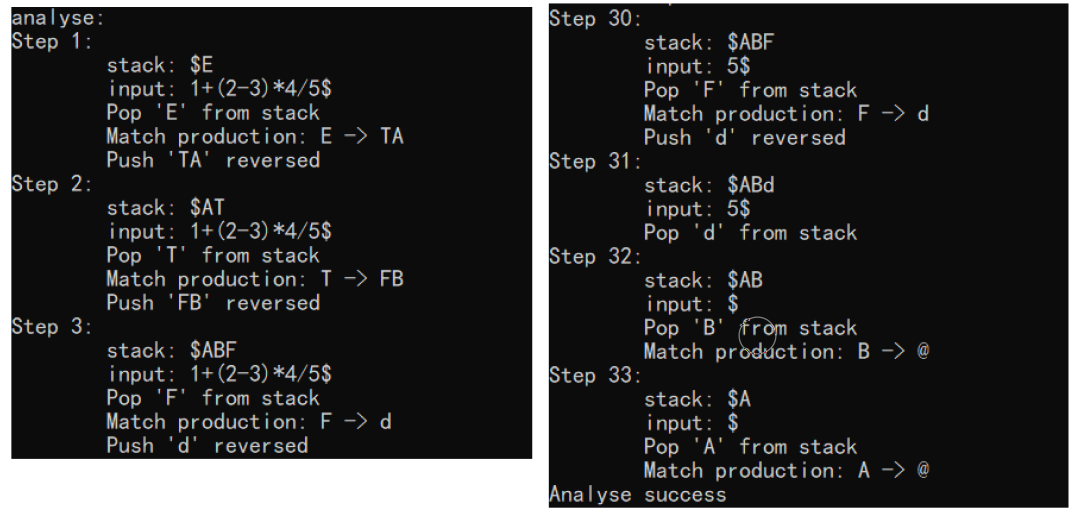
9.2 综合测试1：

包含全元素(四种运算符与括号)

输入：



输出：



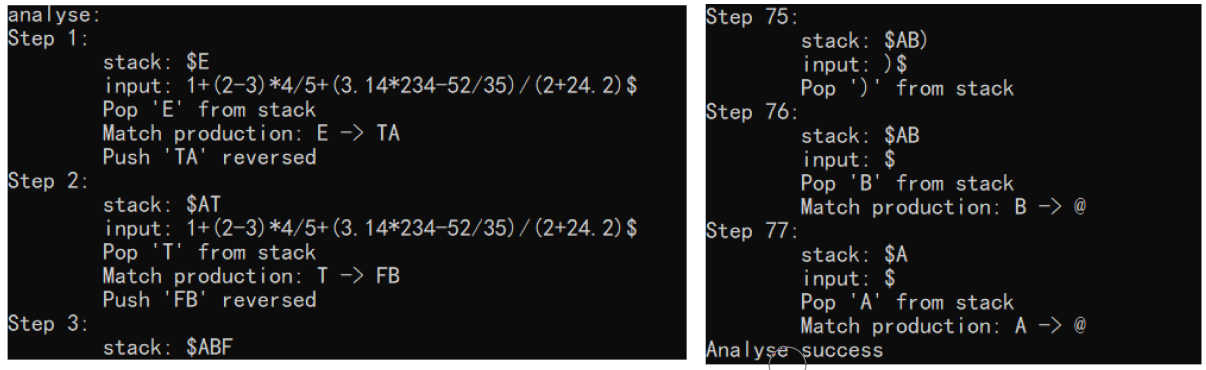
9.3 复杂测试1：

一个较长并且含有小数的包含全元素的式子

输入：



输出：



可以看到，经过较长过程(77步)的分析，最终正确分析成功。

9.4 错误测试：

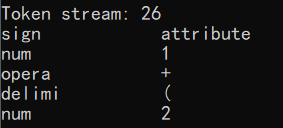
输入串不符合文法规则，考验程序检错能力：

输入：



即去掉测试9.3中的左后一个除号，导致两个括号连在一起出现。

首先运行词法分析，没有报错，正确地输出了记号流：

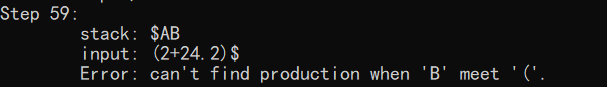


...





输出：



查看语法分析程序输出，第59步告诉我们，”(”出现在了不该出现的地方，因此输入不符合文法，分析结束。

# 三．LR(0)文法程序设计与实现：

## 1. 修改文法：

原始文法如下：

|  |
| --- |
| E→E+T|E-T|T  T→T\*F|T/F|F  F→(E)|num |

改为增广文法：

|  |
| --- |
| E’→E  E→E+T|E-T|T  T→T\*F|T/F|F  F→(E)|num |

写为分开的9个产生式：

|  |
| --- |
| 0: S→E  1: E→E+T  2: E→E-T  3: E→T  4: T→T\*F  5: T→T/F  6: T→F  7 :F→(E)  8: F→num |

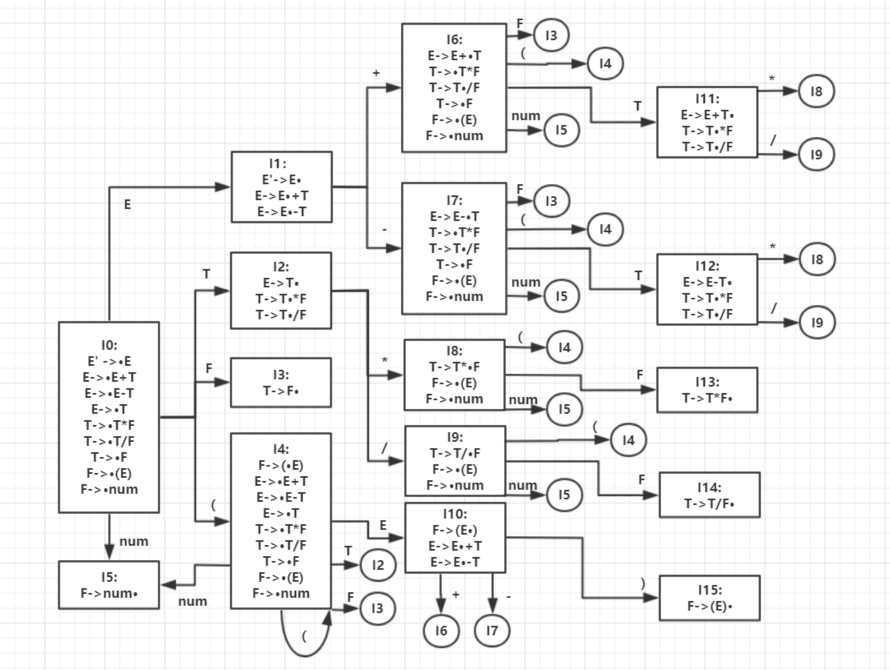
## 2. 求FIRST集和FOLLOW集：

求法同LL，所得结果如下：

|  |  |  |  |
| --- | --- | --- | --- |
|  | E | T | F |
| FIRST | (, num | (, num | (, num |
| FOLLOW | $, ), +, - | $, ), +, -, \*, / | $, ), +, -, \*, / |

## 3. 构造识别该文法所有活前缀的DFA：

此文法使用LR(0)即可分析，DFA如下图：



## 4. 构造该文法的LR分析表：

由DFA可以写出如下的分析表：

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | GOTO | | | | | | | | ACTION | | |
| STATE | + | - | \* | / | ( | ) | num | $ | E | T | F |
| 0 |  |  |  |  | S4 |  | S5 |  | 1 | 2 | 3 |
| 1 | S6 | S7 |  |  |  |  |  | ACC |  |  |  |
| 2 | R3 | R3 | S8 | S9 |  | R3 |  | R3 |  |  |  |
| 3 | R6 | R6 | R6 | R6 |  | R6 |  | R6 |  |  |  |
| 4 |  |  |  |  | S4 |  | S5 |  | 10 | 2 | 3 |
| 5 | R8 | R8 | R8 | R8 |  | R8 |  | R8 |  |  |  |
| 6 |  |  |  |  | S4 |  | S5 |  |  | 11 | 3 |
| 7 |  |  |  |  | S4 |  | S5 |  |  | 12 | 3 |
| 8 |  |  |  |  | S4 |  | S5 |  |  |  | 13 |
| 9 |  |  |  |  | S4 |  | S5 |  |  |  | 14 |
| 10 | S6 | S7 |  |  |  | S15 |  |  |  |  |  |
| 11 | R1 | R1 | S8 | S9 |  | R1 |  | R1 |  |  |  |
| 12 | R2 | R2 | S8 | S9 |  | R2 |  | R2 |  |  |  |
| 13 | R4 | R4 | R4 | R4 |  | R4 |  | R4 |  |  |  |
| 14 | R5 | R5 | R5 | R5 |  | R5 |  | R5 |  |  |  |
| 15 | R7 | R7 | R7 | R7 |  | R7 |  | R7 |  |  |  |

## 5. 程序中使用的主要变量以及结构体说明：

5.1 宏定义：

1. #define STATE\_NUM 16        // 状态数
2. #define TERM\_NUM 8          // 终结符数量，即action列数
3. #define UNTERM\_NUM 3        // 非终结符数量，即goto列数
4. #define G\_NUM 9             // 产生式数
5. #define file\_path "lexical.txt"  // 文件路径

5.2 结构体定义：

本次语法分析使用一个与上次词法分析相同的结构”WPRDS”，用于记录记号流信息。

此外，由于分析表不是单一元素组成的结构，所以设计结构体item记录分析表中每个单元格的数据。

1. // 储存记号流中每个词以及相关信息
2. **typedef** **struct** words
3. {
4. **char** sign;              // 记号
5. string attribute;       // 属性
6. }WORDS;
8. // 储存分析表的表项
9. **typedef** **struct** item
10. {
11. string action;          // action动作(归约R/移入S/接受ACC)或goto标记(G)
12. **int** state;              // 属性
13. }ITEM;

5.3 全局变量说明：

1. // 储存终结符
2. vector<string> TERM;
3. // 储存非终结符
4. vector<string> UNTERM;
5. // 左部生成式
6. string LEFT[] = { "E'", "E", "E", "E", "T", "T", "T", "F", "F"};
7. // 右部生成式
8. string RIGHT[] = { "E", "E+T", "E-T", "T", "T\*F", "T/F", "F", "(E)", "d" };
9. // 右部生成式长度
10. **int** LEN[] = { 1, 3, 3, 1, 3, 3, 1, 3, 1 };
11. // 分析表
12. ITEM table[STATE\_NUM][TERM\_NUM + UNTERM\_NUM];
13. // 输入流(来自词法分析)
14. vector<WORDS> input;

其中，分析表的action部分和goto部分采用相同设计结构，仅使用下标区分。

## 6. 程序中功能模块设计：

|  |  |
| --- | --- |
| 名称 | 功能 |
| int isTerm(string str) | 判断字符(串)str是否为终结符 |
| int isUnTerm(string str) | 判断字符(串)str是否为非终结符 |
| int findIndex(string ch) | 寻找非终结符所在索引号(即在vector UNTERM的下标) |
| int findIndexT(string ch) | 寻找终结符所在索引号(即在vector TERM的下标) |
| void initProgram() | 初始化函数，设定部分变量初值 |
| void getInput() | 从词法分析程序的输出获取记号流 |
| void outPutStack(vector<char> stack) | 输出栈内数据(char格式) |
| void outPutStack(vector<string> stack) | 函数重载：输出栈内数据(string格式) |
| void outPutIn(int ip) | 输出当前输入串 |
| int LRanalyse() | 算法4.3——预测分析程序 |

## 7. 程序逻辑流程：

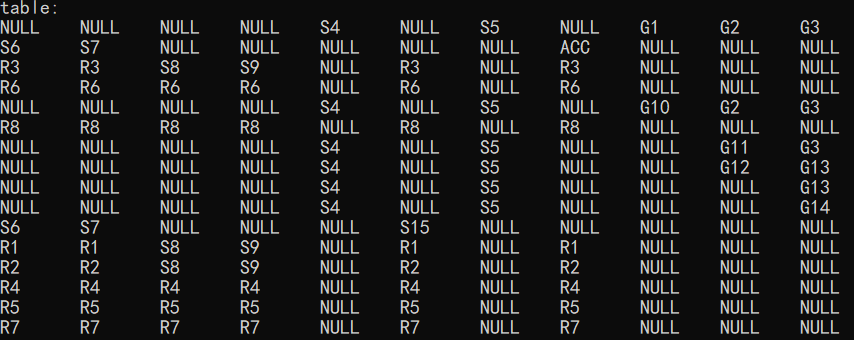
首先，进行程序初始化，设定分析表。为全局变量申请空间；

下一步通过函数getInput()读取词法分析的输出文件，获取记号流；

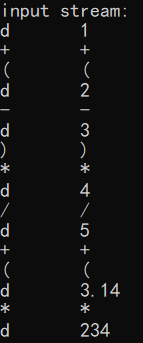
最后进行LR语法分析，并输出过程与结果。

## 8. 程序输出含义：

8.1 首先输出输入的分析表，用于检错

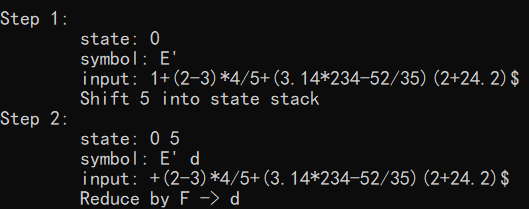


8.2 接着输出处理过的记号流

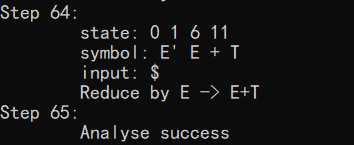


两列分别表示记号类型和记号属性，增加可拓展性，便于以后可能增加的功能。

8.3 最后输出分析过程：



...



## 9. 程序测试：

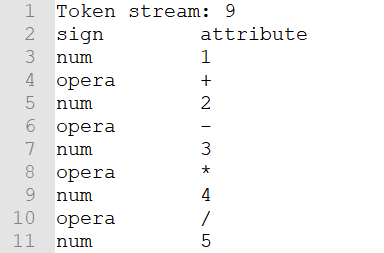
9.1 简单测试1：

包含四种运算符，不含括号，同时演示与词法分析的联动工作。之后的测试将直接展示语法分析过程。

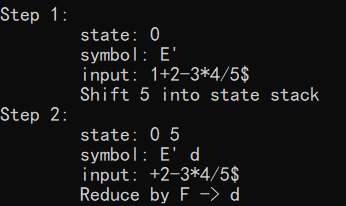
词法分析程序输入为：



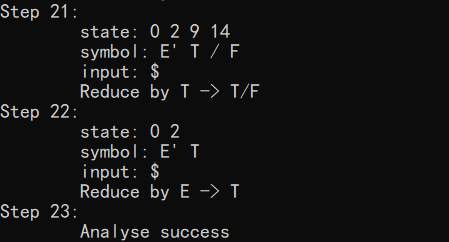
输出记号流文件：



语法分析输出：



...



经过23步的分析，成功分析输入串，说明输入合法。

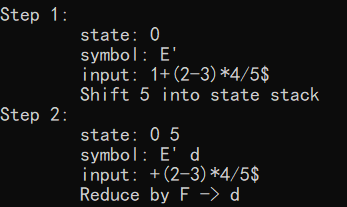
9.2 综合测试1：

包含全元素(四种运算符与括号)

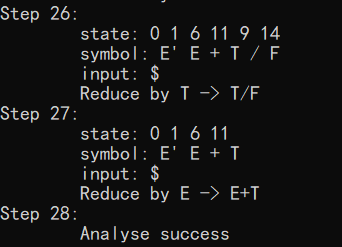
输入：



输出：



...



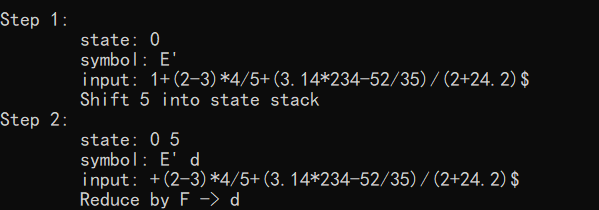
9.3 复杂测试1：

一个较长并且含有小数的包含全元素的式子

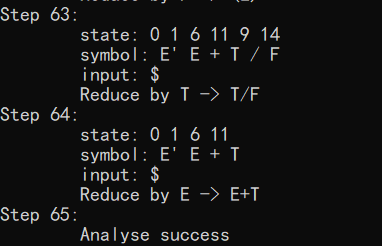
输入：



输出：



...



可以看到，经过较长过程(65步)的分析，最终正确分析成功。

9.4 错误测试：

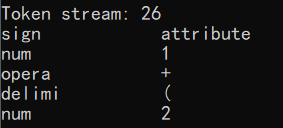
输入串不符合文法规则，考验程序检错能力：

输入：



即去掉测试9.3中的左后一个除号，导致两个括号连在一起出现。

首先运行词法分析，没有报错，正确地输出了记号流：

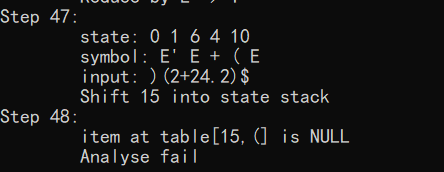


...





输出：



查看语法分析程序输出，第48步告诉我们，”(”出现在了不该出现的地方，因此输入不符合文法，分析结束。

# 四．实验心得与总结

## 1.实验心得：

通过本次实验，复习了语法分析的相关知识，实际上手编写菜吗实现，增强了对自顶向下和自底向上两种语法分析方法的理解。

## 2.实验总结：

实现过程中，发现部分数据结构设计还是有不合理之处，可能对可拓展性产生影响。

对于产生式较多较复杂的文法会带来复杂的实现。

此外，初始化过程占据程序大量代码长度，不便于修改与查错。

# 五．附录：C++实现-代码

## 1. LL分析：

1. #include<iostream>
2. #include<fstream>
3. #include<vector>
4. #include<string>
5. #include<stack>
7. #define TERM\_NUM 8              // 终结符数量
8. #define UNTERM\_NUM 5            // 非终结符数量
9. #define G\_NUM 10                // 产生式数量
10. #define file\_path "lexical.txt"
12. **using** **namespace** std;
14. // 储存记号流中每个词以及相关信息
15. **typedef** **struct** words
16. {
17. **char** sign;                  // 记号
18. string attribute;           // 属性
19. }WORDS;
21. // 全局变量定义区
22. vector<string> G\_LEFT;            // 储存产生式左边
23. vector<string> G\_RIGHT;           // 储存产生式右边
24. vector<string> TERM;          // 储存终结符
25. vector<string> UNTERM;            // 储存非终结符
26. vector<string> FIRST[TERM\_NUM];   // 所有非终结符的FIRST集
27. vector<string> FOLLOW[TERM\_NUM];// 所有非终结符的FOLLOW集
28. vector<vector<string>> table;   // LL的预测分析表
29. vector<WORDS> input;          // 输入流(来自词法分析)
31. // 判断是否为终结符
32. **int** isTerm(string str)
33. {
34. **for** (**int** i = 0; i < TERM.size(); i++)
35. **if** (str == TERM[i])
36. **return** 1;
37. **return** 0;
38. }
40. // 判断是否为非终结符
41. **int** isUnTerm(string str)
42. {
43. **for** (**int** i = 0; i < UNTERM.size(); i++)
44. **if** (str == UNTERM[i])
45. **return** 1;
46. **return** 0;
47. }
49. // 寻找非终结符所在索引号
50. **int** findIndex(string ch)
51. {
52. **for** (**int** i = 0; i < UNTERM\_NUM; i++)
53. **if** (ch == UNTERM[i])
54. **return** i;
55. cout << "ERROR: can't find '" << ch << "' in UnTermin Set" << endl;
56. **return** -1;
57. }
59. // 寻找终结符所在索引号
60. **int** findIndexT(string ch)
61. {
62. **for** (**int** i = 0; i < TERM\_NUM; i++)
63. **if** (ch == TERM[i])
64. **return** i;
65. cout << "ERROR: can't find '" << ch << "' in Termin Set" << endl;
66. **return** -1;
67. }
69. // 初始化
70. **void** initProgram()
71. {
72. vector<string> tmp;
73. string F[] = { "E", "A", "T", "B", "F" };                                           // 非终结符
74. string T[] = { "+", "-", "\*", "/", "(", ")", "d", "$" };                            // 终结符
75. string LEFT[] = { "E", "A", "A", "A", "T", "B", "B", "B", "F", "F" };               // 左部生成式
76. string RIGHT[] = { "TA", "+TA", "-TA", "@", "FB", "\*FB", "/FB", "@", "(E)", "d" };  // 右部生成式
78. // 初始化table
79. **for** (**int** i = 0; i < UNTERM\_NUM; i++)
80. {
81. **for** (**int** j = 0; j < TERM\_NUM; j++)
82. {
83. tmp.push\_back("NULL");
84. }
85. table.push\_back(tmp);
86. tmp.clear();
87. }
89. // 初始化产生式，终结符与非终结符
90. **for** (**int** i = 0; i < 10; i++)
91. {
92. G\_LEFT.push\_back(LEFT[i]);
93. G\_RIGHT.push\_back(RIGHT[i]);
94. }
95. **for** (**int** i = 0; i < TERM\_NUM; i++)
96. {
97. TERM.push\_back(T[i]);
98. }
99. **for** (**int** i = 0; i < UNTERM\_NUM; i++)
100. {
101. UNTERM.push\_back(F[i]);
102. }
104. // 初始化FIRST集，FOLLOW集
105. FIRST[0].push\_back("(");
106. FIRST[0].push\_back("d");
107. FIRST[1].push\_back("+");
108. FIRST[1].push\_back("-");
109. FIRST[1].push\_back("@");
110. FIRST[2].push\_back("(");
111. FIRST[2].push\_back("d");
112. FIRST[3].push\_back("\*");
113. FIRST[3].push\_back("/");
114. FIRST[3].push\_back("@");
115. FIRST[4].push\_back("(");
116. FIRST[4].push\_back("d");
118. FOLLOW[0].push\_back("$");
119. FOLLOW[0].push\_back(")");
120. FOLLOW[1].push\_back("$");
121. FOLLOW[1].push\_back(")");
122. FOLLOW[2].push\_back("+");
123. FOLLOW[2].push\_back("-");
124. FOLLOW[2].push\_back("$");
125. FOLLOW[2].push\_back(")");
126. FOLLOW[3].push\_back("+");
127. FOLLOW[3].push\_back("-");
128. FOLLOW[3].push\_back("$");
129. FOLLOW[3].push\_back(")");
130. FOLLOW[4].push\_back("\*");
131. FOLLOW[4].push\_back("/");
132. FOLLOW[4].push\_back("+");
133. FOLLOW[4].push\_back("-");
134. FOLLOW[4].push\_back("$");
135. FOLLOW[4].push\_back(")");
137. **return**;
138. }
140. // 从词法分析程序的输出获取记号流
141. **void** getInput()
142. {
143. string str;
144. **int** count = 0;
145. WORDS tmpWord;
146. ifstream infile;
148. // 打开由词法分析输出的文件
149. infile.open(file\_path);
151. // 去除无用符号
152. infile >> str;
153. infile >> str;
155. // 获取记号流数量
156. infile >> count;
158. // 去除无用符号
159. infile >> str;
160. infile >> str;
162. // 获取记号详细属性与信息
163. **for** (**int** i = 0; i < count; i++)
164. {
165. infile >> str;
166. **if** (str == "num")
167. {
168. tmpWord.sign = 'd';
169. infile >> str;
170. tmpWord.attribute = str;
171. }
172. **else**
173. {
174. infile >> str;
175. tmpWord.sign = str[0];
176. tmpWord.attribute = str;
177. }
178. input.push\_back(tmpWord);
179. }
181. tmpWord.attribute = "$";
182. tmpWord.sign = '$';
183. input.push\_back(tmpWord);
185. // 输出得到的记号流(测试用)
186. **for** (**int** i = 0; i < input.size(); i++)
187. {
188. cout << input[i].sign << "\t" << input[i].attribute << endl;
189. }
190. }
192. // 输出栈内数据
193. **void** outPutStack(vector<**char**> stack)
194. {
195. cout << "\tstack: ";
196. **for** (**int** i = 0; i < stack.size(); i++)
197. cout << stack[i];
198. cout << endl;
199. }
201. // 输出当前输入串
202. **void** outPutIn(**int** ip)
203. {
204. cout << "\tinput: ";
205. **for** (**int** i = ip; i < input.size(); i++)
206. cout << input[i].attribute;
207. cout << endl;
208. }
210. // 算法4.2——构造分析表
211. **void** genTable()
212. {
213. string tmp;
214. **for** (**int** i = 0; i < G\_NUM; i++)
215. {
216. tmp = G\_RIGHT[i][0];
217. **if** (isUnTerm(tmp))  // 如果右部第一个符号为非终结符
218. {
219. **for** (**int** j = 0; j < FIRST[findIndex(tmp)].size(); j++)
220. {
221. **if** (FIRST[findIndex(tmp)][j] != "@")
222. {
223. table[findIndex(G\_LEFT[i])][findIndexT(FIRST[findIndex(tmp)][j])] = G\_RIGHT[i];
224. }
225. }
226. }
227. **else** **if** (tmp == "@")  // 如果可致空
228. {
229. **for** (**int** j = 0; j < FOLLOW[findIndex(G\_LEFT[i])].size(); j++)
230. {
231. table[findIndex(G\_LEFT[i])][findIndexT(FOLLOW[findIndex(G\_LEFT[i])][j])] = G\_RIGHT[i];
232. }
233. }
234. **else** **if** (isTerm(tmp))  // 如果是终结符
235. {
236. table[findIndex(G\_LEFT[i])][findIndexT(tmp)] = G\_RIGHT[i];
237. }
238. }
239. }
241. // 算法4.1——预测分析程序
242. **int** LLanalyse()
243. {
244. vector<**char**> LLstack;
245. LLstack.push\_back('$');
246. LLstack.push\_back('E');
247. **int** step = 1;           // 记录分析步骤数
248. **int** ip = 0;             // 记录输入串读到的位置
249. **int** i, j;
250. string tmpX;
251. string tmpa;
253. **char** X = LLstack.back();
254. tmpX.clear();
255. tmpX.push\_back(X);
256. **char** a = input[ip].sign;
257. tmpa.clear();
258. tmpa.push\_back(a);
260. **while** (X != '$')
261. {
262. cout << "Step " << step << ": " << endl;
263. outPutStack(LLstack);
264. outPutIn(ip);
265. **if** (isTerm(tmpX))  // 如果是终结符
266. {
267. **if** (X == a)  // 匹配成功，弹栈并继续分析
268. {
269. cout << "\tPop '" << X << "' from stack" << endl;
270. LLstack.pop\_back();
271. ip++;
272. }
273. **else**  //匹配失败，报错并结束分析
274. {
275. cout << "\tError: can't match '" << X << "' and '" << a << "'." << endl;
276. **return** -1;
277. }
278. }
279. **else** **if** (isUnTerm(tmpX))  // 如果是非终结符
280. {
281. i = findIndex(tmpX);
282. j = findIndexT(tmpa);
283. **if** (table[i][j] == "NULL")  // 分析表对应项为空，报错并结束分析
284. {
285. cout << "\tError: can't find production when '" << X << "' meet '" << a << "'." << endl;
286. **return** -1;
287. }
288. **else**
289. {
290. cout << "\tPop '" << X << "' from stack" << endl;
291. cout << "\tMatch production: " << X << " -> " << table[i][j] << endl;
292. LLstack.pop\_back();
293. **if** (table[i][j] != "@")  // 如果产生式不是空串
294. {
295. cout << "\tPush '" << table[i][j] << "' reversed" << endl;
296. **for** (**int** k = table[i][j].size() - 1; k >= 0; k--)
297. LLstack.push\_back(table[i][j][k]);
298. }
299. }
300. }
302. step++;
304. // 读取栈顶与输入串第一个字符
305. X = LLstack.back();
306. tmpX.clear();
307. tmpX.push\_back(X);
308. **if** (ip < input.size())
309. {
310. a = input[ip].sign;
311. tmpa.clear();
312. tmpa.push\_back(a);
313. }
314. }
315. **if** (X == '$' && ip == input.size() - 1)
316. {
317. cout << "Analyse success" << endl;
318. }
319. **else**
320. {
321. cout << "Analyse fail: stack is empty but input haven't been exhausted" << endl;
322. }
323. }



328. **int** main()
329. {
330. initProgram();
331. genTable();
333. cout << "table: " << endl;
334. cout << "+\t\t-\t\t\*\t\t/\t\t(\t\t)\t\td\t\t$" << endl;
335. **for** (**int** i = 0; i < UNTERM\_NUM; i++)
336. {
337. **for** (**int** j = 0; j < TERM\_NUM; j++)
338. cout << UNTERM[i] << " -> " << table[i][j] << "\t";
339. cout << endl;
340. }
341. cout << endl;
343. cout << "input stream: " << endl;
344. getInput();
345. cout << endl;
347. cout << "analyse: " << endl;
348. LLanalyse();
350. **return** 0;
351. }

## 2. LR分析：

1. #include<iostream>
2. #include<fstream>
3. #include<vector>
4. #include<string>
5. #include<stack>
7. #define STATE\_NUM 16        // 状态数
8. #define TERM\_NUM 8          // 终结符数量，即action列数
9. #define UNTERM\_NUM 3        // 非终结符数量，即goto列数
10. #define G\_NUM 9             // 产生式数
11. #define file\_path "C:/Users/87290/Desktop/PROGRAM/编译原理-词法分析/我的代码/lexical\_analysis/lexical.txt"
13. **using** **namespace** std;
15. // 储存记号流中每个词以及相关信息
16. **typedef** **struct** words
17. {
18. **char** sign;              // 记号
19. string attribute;       // 属性
20. }WORDS;
22. // 储存分析表的表项
23. **typedef** **struct** item
24. {
25. string action;          // action动作(归约R/移入S/接受ACC)或goto标记(G)
26. **int** state;              // 属性
27. }ITEM;

30. // 储存终结符
31. vector<string> TERM;
32. // 储存非终结符
33. vector<string> UNTERM;
34. // 左部生成式
35. string LEFT[] = { "E'", "E", "E", "E", "T", "T", "T", "F", "F"};
36. // 右部生成式
37. string RIGHT[] = { "E", "E+T", "E-T", "T", "T\*F", "T/F", "F", "(E)", "d" };
38. // 右部生成式长度
39. **int** LEN[] = { 1, 3, 3, 1, 3, 3, 1, 3, 1 };
40. // 分析表
41. ITEM table[STATE\_NUM][TERM\_NUM + UNTERM\_NUM];
42. // 输入流(来自词法分析)
43. vector<WORDS> input;
45. // 判断是否为终结符
46. **int** isTerm(string str)
47. {
48. **for** (**int** i = 0; i < TERM.size(); i++)
49. **if** (str == TERM[i])
50. **return** 1;
51. **return** 0;
52. }
54. // 判断是否为非终结符
55. **int** isUnTerm(string str)
56. {
57. **for** (**int** i = 0; i < UNTERM.size(); i++)
58. **if** (str == UNTERM[i])
59. **return** 1;
60. **return** 0;
61. }
63. // 寻找非终结符所在索引号
64. **int** findIndex(string ch)
65. {
66. **for** (**int** i = 0; i < UNTERM\_NUM; i++)
67. **if** (ch == UNTERM[i])
68. **return** i;
69. cout << "ERROR: can't find '" << ch << "' in UnTermin Set" << endl;
70. **return** -1;
71. }
73. // 寻找终结符所在索引号
74. **int** findIndexT(string ch)
75. {
76. **for** (**int** i = 0; i < TERM\_NUM; i++)
77. **if** (ch == TERM[i])
78. **return** i;
79. cout << "ERROR: can't find '" << ch << "' in Termin Set" << endl;
80. **return** -1;
81. }
83. // 输出栈内数据
84. **void** outPutStack(vector<string> stack)
85. {
86. cout << "\tsymbol: ";
87. **for** (**int** i = 0; i < stack.size(); i++)
88. cout << stack[i] << " ";
89. cout << endl;
90. }
91. **void** outPutStack(vector<**int**> stack)
92. {
93. cout << "\tstate: ";
94. **for** (**int** i = 0; i < stack.size(); i++)
95. cout << stack[i] << " ";
96. cout << endl;
97. }
99. // 输出当前输入串
100. **void** outPutIn(**int** ip)
101. {
102. cout << "\tinput: ";
103. **for** (**int** i = ip; i < input.size(); i++)
104. cout << input[i].attribute;
105. cout << endl;
106. }
108. // 从词法分析程序的输出获取记号流
109. **void** getInput()
110. {
111. string str;
112. **int** count = 0;
113. WORDS tmpWord;
114. ifstream infile;
116. // 打开由词法分析输出的文件
117. infile.open(file\_path);
119. // 去除无用符号
120. infile >> str;
121. infile >> str;
123. // 获取记号流数量
124. infile >> count;
126. // 去除无用符号
127. infile >> str;
128. infile >> str;
130. // 获取记号详细属性与信息
131. **for** (**int** i = 0; i < count; i++)
132. {
133. infile >> str;
134. **if** (str == "num")
135. {
136. tmpWord.sign = 'd';
137. infile >> str;
138. tmpWord.attribute = str;
139. }
140. **else**
141. {
142. infile >> str;
143. tmpWord.sign = str[0];
144. tmpWord.attribute = str;
145. }
146. input.push\_back(tmpWord);
147. }
149. tmpWord.attribute = "$";
150. tmpWord.sign = '$';
151. input.push\_back(tmpWord);
153. // 输出得到的记号流(测试用)
154. **for** (**int** i = 0; i < input.size(); i++)
155. {
156. cout << input[i].sign << "\t" << input[i].attribute << endl;
157. }
158. }
160. // 算法4.3——LR分析
161. **int** LRanalyse()
162. {
163. vector<**int**> State;        // 状态栈
164. vector<string> Symbol;    // 符号栈
165. **int** step = 1;           // 记录分析步骤数
166. **int** ip = 0;             // 记录输入串读到的位置
167. **int** i, j;
168. **int** S = 0;              // 栈顶状态
169. **char** a;                 // ip所指符号
170. string tmpa;            // a的string形式的副本，便于契合库函数
172. State.push\_back(0);
173. Symbol.push\_back("E'");
175. a = input[ip].sign;
176. tmpa.clear();
177. tmpa.push\_back(a);
179. **while** (1)
180. {
181. cout << "Step " << step << ": " << endl;
183. **if** (isTerm(tmpa))
184. j = findIndexT(tmpa);
185. **else**
186. {
187. cout << "ERROR: can't find '" << a << "' in Termin Set" << endl;
188. **return** -1;
189. }
191. **if** (table[S][j].action == "S")  // 移进动作
192. {
193. outPutStack(State);
194. outPutStack(Symbol);
195. outPutIn(ip);
196. State.push\_back(table[S][j].state);
197. Symbol.push\_back(tmpa);
198. ip++;
199. cout << "\tShift " << table[S][j].state << " into state stack" << endl;
200. }
201. **else** **if** (table[S][j].action == "R")  // 归约动作
202. {
203. outPutStack(State);
204. outPutStack(Symbol);
205. outPutIn(ip);
206. **for** (**int** k = 0; k < LEN[table[S][j].state]; k++)
207. {
208. State.pop\_back();
209. Symbol.pop\_back();
210. }
211. Symbol.push\_back(LEFT[table[S][j].state]);
212. cout << "\tReduce by " << LEFT[table[S][j].state] << " -> " << RIGHT[table[S][j].state] << endl;
213. **if** (isUnTerm(LEFT[table[S][j].state]))
214. j = findIndex(LEFT[table[S][j].state]) + TERM\_NUM;
215. **else**
216. {
217. cout << "ERROR: can't find '" << a << "' in UnTermin Set" << endl;
218. **return** -1;
219. }
220. S = State.back();
221. State.push\_back(table[S][j].state);
222. }
223. **else** **if** (table[S][j].action == "ACC")
224. {
225. cout << "\tAnalyse success" << endl;
226. **break**;
227. }
228. **else**
229. {
230. cout << "\titem at table[" << S << "," << TERM[j] << "] is " << table[S][j].action << endl;
231. cout << "\tAnalyse fail" << endl;
232. **return** -1;
233. }
235. step++;
237. S = State.back();
238. **if** (ip < input.size())
239. {
240. a = input[ip].sign;
241. tmpa.clear();
242. tmpa.push\_back(a);
243. }
244. }
246. **return** 0;
247. }
249. // 初始化
250. **void** initProgram()
251. {
252. // 非终结符
253. string F[] = { "E", "T", "F" };
254. // 终结符
255. string T[] = { "+", "-", "\*", "/", "(", ")", "d", "$" };
257. // 初始化终结符，非终结符
258. **for** (**int** i = 0; i < TERM\_NUM; i++)
259. {
260. TERM.push\_back(T[i]);
261. }
262. **for** (**int** i = 0; i < UNTERM\_NUM; i++)
263. {
264. UNTERM.push\_back(F[i]);
265. }
267. // 初始化分析表
268. // 先全部初始化为空(NULL, -1)
269. **for** (**int** i = 0; i < STATE\_NUM; i++)
270. **for** (**int** j = 0; j < TERM\_NUM + UNTERM\_NUM; j++)
271. table[i][j].action = "NULL", table[i][j].state = -1;
273. table[0][4] = { "S", 4 };
274. table[0][6] = { "S", 5 };
275. table[0][8] = { "G", 1 };
276. table[0][9] = { "G", 2 };
277. table[0][10] = { "G", 3 };
279. table[1][0] = { "S", 6 };
280. table[1][1] = { "S", 7 };
281. table[1][7] = { "ACC", -1 };
283. table[2][0] = { "R", 3 };
284. table[2][1] = { "R", 3 };
285. table[2][2] = { "S", 8 };
286. table[2][3] = { "S", 9 };
287. table[2][5] = { "R", 3 };
288. table[2][7] = { "R", 3 };
290. table[3][0] = { "R", 6 };
291. table[3][1] = { "R", 6 };
292. table[3][2] = { "R", 6 };
293. table[3][3] = { "R", 6 };
294. table[3][5] = { "R", 6 };
295. table[3][7] = { "R", 6 };
297. table[4][4] = { "S", 4 };
298. table[4][6] = { "S", 5 };
299. table[4][8] = { "G", 10 };
300. table[4][9] = { "G", 2 };
301. table[4][10] = { "G", 3 };
303. table[5][0] = { "R", 8 };
304. table[5][1] = { "R", 8 };
305. table[5][2] = { "R", 8 };
306. table[5][3] = { "R", 8 };
307. table[5][5] = { "R", 8 };
308. table[5][7] = { "R", 8 };
310. table[6][4] = { "S", 4 };
311. table[6][6] = { "S", 5 };
312. table[6][9] = { "G", 11 };
313. table[6][10] = { "G", 3 };
315. table[7][4] = { "S", 4 };
316. table[7][6] = { "S", 5 };
317. table[7][9] = { "G", 12 };
318. table[7][10] = { "G", 13 };
320. table[8][4] = { "S", 4 };
321. table[8][6] = { "S", 5 };
322. table[8][10] = { "G", 13 };
324. table[9][4] = { "S", 4 };
325. table[9][6] = { "S", 5 };
326. table[9][10] = { "G", 14 };
328. table[10][0] = { "S", 6 };
329. table[10][1] = { "S", 7 };
330. table[10][5] = { "S", 15 };
332. table[11][0] = { "R", 1 };
333. table[11][1] = { "R", 1 };
334. table[11][2] = { "S", 8 };
335. table[11][3] = { "S", 9 };
336. table[11][5] = { "R", 1 };
337. table[11][7] = { "R", 1 };
339. table[12][0] = { "R", 2 };
340. table[12][1] = { "R", 2 };
341. table[12][2] = { "S", 8 };
342. table[12][3] = { "S", 9 };
343. table[12][5] = { "R", 2 };
344. table[12][7] = { "R", 2 };
346. table[13][0] = { "R", 4 };
347. table[13][1] = { "R", 4 };
348. table[13][2] = { "R", 4 };
349. table[13][3] = { "R", 4 };
350. table[13][5] = { "R", 4 };
351. table[13][7] = { "R", 4 };
353. table[14][0] = { "R", 5 };
354. table[14][1] = { "R", 5 };
355. table[14][2] = { "R", 5 };
356. table[14][3] = { "R", 5 };
357. table[14][5] = { "R", 5 };
358. table[14][7] = { "R", 5 };
360. table[15][0] = { "R", 7 };
361. table[15][1] = { "R", 7 };
362. table[15][2] = { "R", 7 };
363. table[15][3] = { "R", 7 };
364. table[15][5] = { "R", 7 };
365. table[15][7] = { "R", 7 };
367. // 输出
368. cout << "table:" << endl;
369. **for** (**int** i = 0; i < STATE\_NUM; i++)
370. {
371. **for** (**int** j = 0; j < TERM\_NUM + UNTERM\_NUM; j++)
372. **if** (table[i][j].state != -1)
373. cout << table[i][j].action << table[i][j].state << "\t";
374. **else**
375. cout << table[i][j].action << "\t";
376. cout << endl;
377. }
379. }

382. **int** main()
383. {
384. initProgram();
386. cout << "input stream: " << endl;
387. getInput();
388. cout << endl;
390. LRanalyse();
392. **return** 0;
393. }