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

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 。

2） 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。

3） 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 $X \in V_T$，则 $First(X) = \{X\}$。

Step2. 如果 $X \in V_N$，且有产生式 $X \to a\omega$，$a \in V_T$，则把 a 加入到 First(X)中。若存在 $X \to \varepsilon$，则把 ε 也加入到 First(X)中。

Step3.若存在产生式 $X \to a_1 a_2 \cdots a_n$，$a_1, a_2, \cdots, a_{k-1} \in V_N$，且 $\varepsilon \in First(a_i)(i = 1,2,\cdots,k-1)$，则把 $First(a_i)(i = 1,2,\cdots,k-1)$中除 ε 的一切符号加入到 First(X)中。

Step4. 若 $\varepsilon \in First(a_i)(i = 1,2,\cdots,n)$，则把 ε 加入到 First(X)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 Follow(E)中，其中 E 是文法开始符号。

Step2. 如果存在产生式 $A \to \alpha X \beta$，$X \in V_N$，则把 $First(\beta)$中一切除 ε 的符号都加入到 Follow(X)中。

Step3. 如果存在产生式 $A \to \alpha X$，$X \in V_N$，则把 Follow(A)的所有符号都加入到 Follow(X)中。

Step4. 如果存在产生式 $A \to \alpha X \beta$，$X \in V_N$，且 $\beta \Rightarrow^* \varepsilon$，则把 Follow(A)的所有符号都加入到 Follow(X)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：

```
#define TERM_NUM 8              // 终结符数量
#define UNTERM_NUM 5            // 非终结符数量
#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。

```
// 储存记号流中每个词以及相关信息
typedef struct words
{
    char sign;                  // 记号
    string attribute;           // 属性
}WORDS;
```

5.3 全局变量说明：

```
vector<string> G_LEFT;            // 储存产生式左边
vector<string> G_RIGHT;           // 储存产生式右边
vector<string> TERM;              // 储存终结符
vector<string> UNTERM;            // 储存非终结符
vector<string> FIRST[TERM_NUM];   // 所有非终结符的 FIRST 集
vector<string> FOLLOW[TERM_NUM];  // 所有非终结符的 FOLLOW 集
vector<vector<string>> table;     // LL 的预测分析表
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

```
table:
+               -               *               /               (               )               d               $
E -> NULL       E -> NULL       E -> NULL       E -> NULL       E -> TA E -> NULL       E -> TA E -> NULL
A -> +TA        A -> -TA        A -> NULL       A -> NULL       A -> NULL       A -> @  A -> NULL       A -> @
T -> NULL       T -> NULL       T -> NULL       T -> NULL       T -> FB T -> NULL       T -> FB T -> NULL
B -> @  B -> @  B -> *FB        B -> /FB        B -> NULL       B -> @  B -> NULL       B -> @
F -> NULL       F -> NULL       F -> NULL       F -> NULL       F -> (E)        F -> NULL       F -> d  F -> NULL
```

8.2 接着输出处理过的记号流

```
input stream:
(       (
d       12
+       +
d       34
)       )
*       *
d       5
-       -
d       6
/       /
d       8
$       $
```

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：

```
analyse:
Step 1:
        stack: $E
        input: (12+34)*5-6/8$
        Pop 'E' from stack
        Match production: E -> TA
        Push 'TA' reversed
Step 2:
        stack: $AT
        input: (12+34)*5-6/8$
        Pop 'T' from stack
        Match production: T -> FB
        Push 'FB' reversed
```

...

```
Step 32:
        stack: $AB
        input: $
        Pop 'B' from stack
        Match production: B -> @
Step 33:
        stack: $A
        input: $
        Pop 'A' from stack
        Match production: A -> @
Analyse success
```

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。

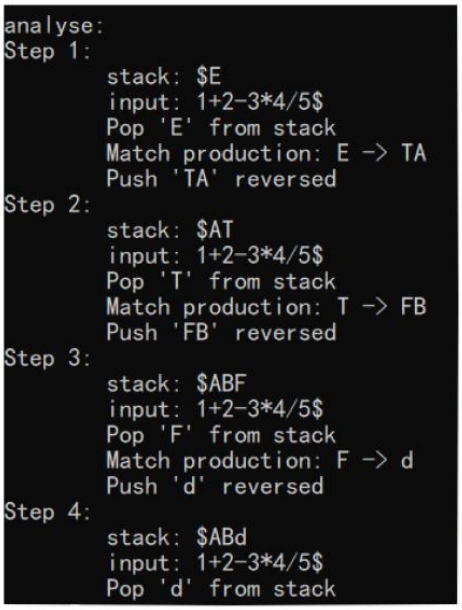
词法分析程序输入为：

```
1+2-3*4/5
```

输出记号流文件：

```
Token stream: 9
sign        attribute
num         1
opera       +
num         2
opera       -
num         3
opera       *
num         4
opera       /
num         5
```

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。

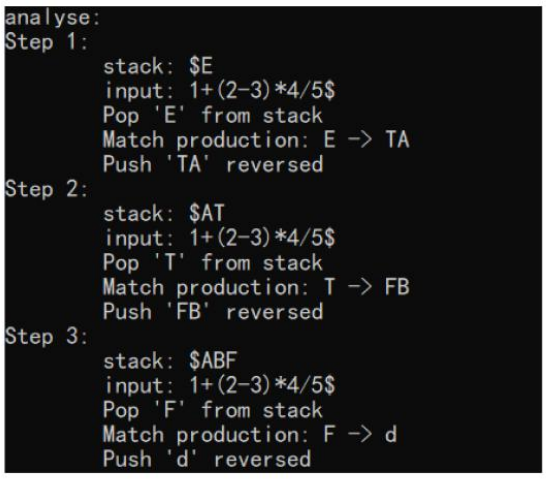
9.2 综合测试 1：

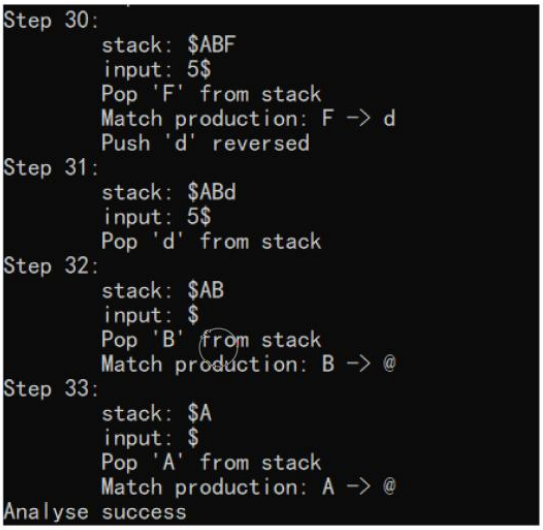
包含全元素(四种运算符与括号)

输入：

```
1+(2-3)*4/5
```

输出：





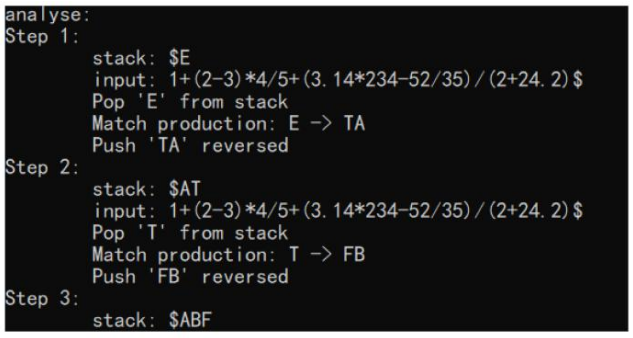
9.3 复杂测试 1：

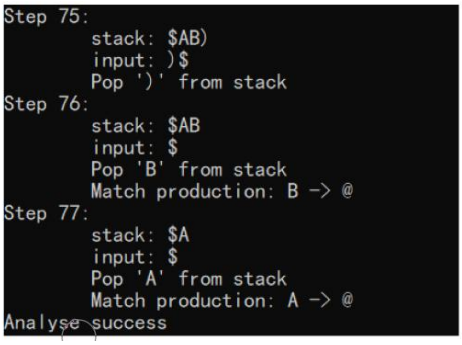
一个较长并且含有小数的包含全元素的式子

输入：

```
1+(2-3)*4/5+(3.14*234-52/35)/(2+24.2)
```

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：

```
1+(2-3)*4/5+(3.14*234-52/35)(2+24.2)
```

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：

```
Token stream: 26
sign            attribute
num             1
opera           +
delimi          (
num             2
```

...

```
Errors:0
Total characters: 36
```

输出：

```
Step 59:
        stack: $AB
        input:  (2+24.2)$
        Error: can't find production when 'B' meet '('.
```

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

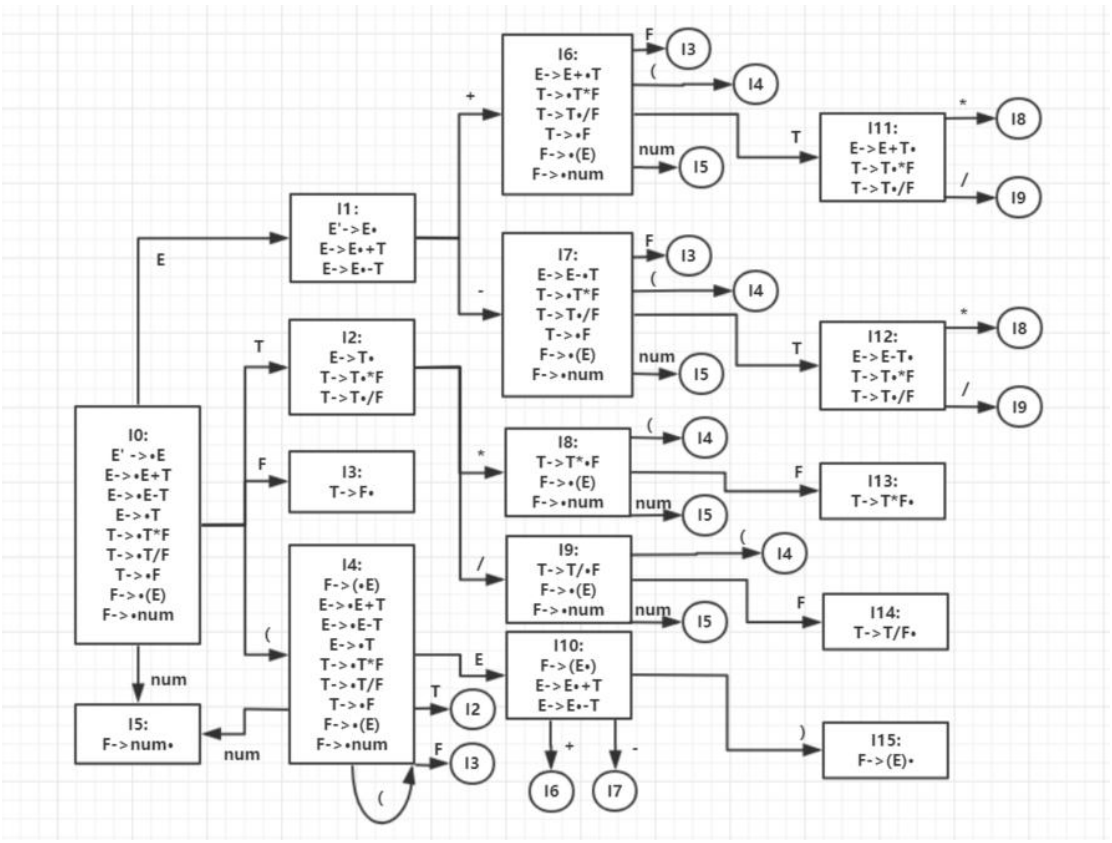
2. 求 FIRST 集和 FOLLOW 集：

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：

```
#define STATE_NUM 16          // 状态数
#define TERM_NUM 8            // 终结符数量，即 action 列数
#define UNTERM_NUM 3          // 非终结符数量，即 goto 列数
#define G_NUM 9               // 产生式数
#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。

```
// 储存记号流中每个词以及相关信息
typedef struct words
{
    char sign;              // 记号
    string attribute;       // 属性
}WORDS;

// 储存分析表的表项
typedef struct item
{
    string action;          // action 动作(归约 R/移入 S/接受 ACC)或 goto 标记(G)
    int state;              // 属性
}ITEM;
```

5.3 全局变量说明：

```
// 储存终结符
vector<string> TERM;
// 储存非终结符
vector<string> UNTERM;
// 左部生成式
string LEFT[] = { "E'", "E", "E", "E", "T", "T", "T", "F", "F"};
// 右部生成式
string RIGHT[] = { "E", "E+T", "E-T", "T", "T*F", "T/F", "F", "(E)", "d" };
// 右部生成式长度
int LEN[] = { 1, 3, 3, 1, 3, 3, 1, 3, 1 };
// 分析表
ITEM table[STATE_NUM][TERM_NUM + UNTERM_NUM];
// 输入流(来自词法分析)
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

```
table:
NULL    NULL    NULL    NULL    S4      NULL    S5      NULL    G1      G2      G3
S6      S7      NULL    NULL    NULL    NULL    NULL    ACC     NULL    NULL    NULL
R3      R3      S8      S9      NULL    R3      NULL    R3      NULL    NULL    NULL
R6      R6      R6      R6      NULL    R6      NULL    R6      NULL    NULL    NULL
NULL    NULL    NULL    NULL    S4      NULL    S5      NULL    G10     G2      G3
R8      R8      R8      R8      NULL    R8      NULL    R8      NULL    NULL    NULL
NULL    NULL    NULL    NULL    S4      NULL    S5      NULL    NULL    G11     G3
NULL    NULL    NULL    NULL    S4      NULL    S5      NULL    NULL    G12     G13
NULL    NULL    NULL    NULL    S4      NULL    S5      NULL    NULL    NULL    G13
NULL    NULL    NULL    NULL    S4      NULL    S5      NULL    NULL    NULL    G14
S6      S7      NULL    NULL    NULL    S15     NULL    NULL    NULL    NULL    NULL
R1      R1      S8      S9      NULL    R1      NULL    R1      NULL    NULL    NULL
R2      R2      S8      S9      NULL    R2      NULL    R2      NULL    NULL    NULL
R4      R4      R4      R4      NULL    R4      NULL    R4      NULL    NULL    NULL
R5      R5      R5      R5      NULL    R5      NULL    R5      NULL    NULL    NULL
R7      R7      R7      R7      NULL    R7      NULL    R7      NULL    NULL    NULL
```

8.2 接着输出处理过的记号流

```
input stream:
d       1
+       +
(       (
d       2
-       -
d       3
)       )
*       *
d       4
/       /
d       5
+       +
(       (
d       3.14
*       *
d       234
```

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：

```
Step 1:
        state: 0
        symbol: E'
        input: 1+(2-3)*4/5+(3.14*234-52/35)(2+24.2)$
        Shift 5 into state stack
Step 2:
        state: 0 5
        symbol: E' d
        input: +(2-3)*4/5+(3.14*234-52/35)(2+24.2)$
        Reduce by F -> d
```

...

```
Step 64:
        state: 0 1 6 11
        symbol: E' E + T
        input: $
        Reduce by E -> E+T
Step 65:
        Analyse success
```

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：

```
1+2-3*4/5
```

输出记号流文件：

```
Token stream: 9
sign        attribute
num         1
opera       +
num         2
opera       -
num         3
opera       *
num         4
opera       /
num         5
```

语法分析输出：

```
Step 1:
        state: 0
        symbol: E'
        input: 1+2-3*4/5$
        Shift 5 into state stack
Step 2:
        state: 0 5
        symbol: E' d
        input: +2-3*4/5$
        Reduce by F -> d
```

...

```
Step 21:
        state: 0 2 9 14
        symbol: E' T / F
        input: $
        Reduce by T -> T/F
Step 22:
        state: 0 2
        symbol: E' T
        input: $
        Reduce by E -> T
Step 23:
        Analyse success
```

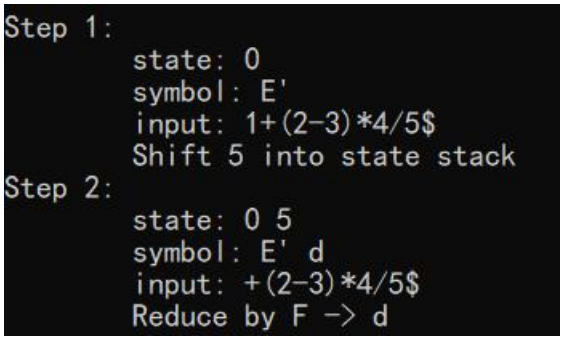
经过 23 步的分析，成功分析输入串，说明输入合法。

9.2 综合测试 1：
包含全元素(四种运算符与括号)

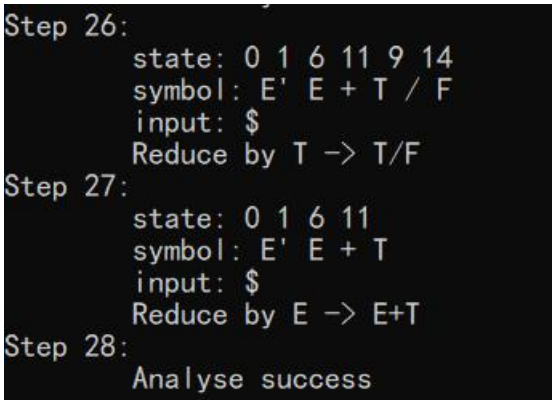
输入：

```
1+(2-3)*4/5
```

输出：



...

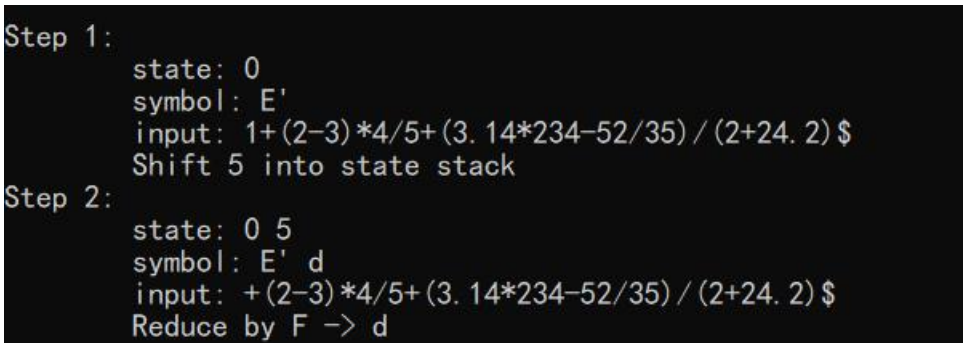


9.3 复杂测试 1：
一个较长并且含有小数的包含全元素的式子

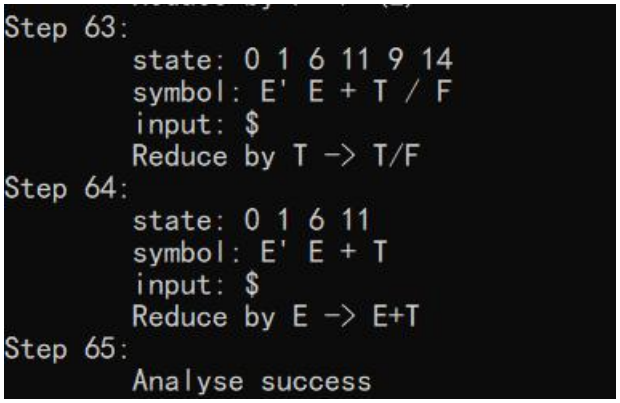
输入：

```
1+(2-3)*4/5+(3.14*234-52/35)/(2+24.2)
```

输出：



...

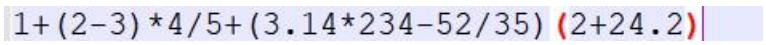


可以看到，经过较长过程(65 步)的分析，最终正确分析成功。

9.4 错误测试：

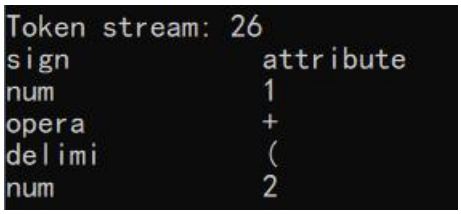
输入串不符合文法规则，考验程序检错能力：

输入：



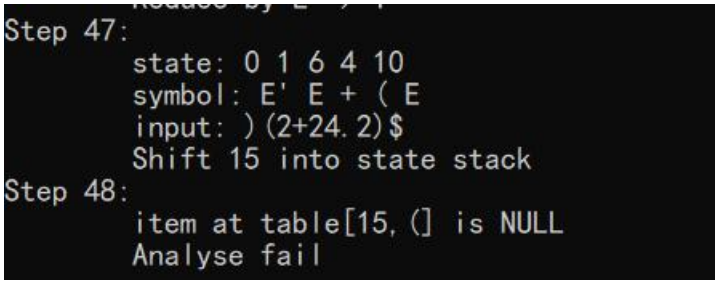
即去掉测试 9.3 中的左后一个除号，导致两个括号连在一起出现。

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：

```
#include<iostream>
#include<fstream>
#include<vector>
#include<string>
#include<stack>

#define TERM_NUM 8              // 终结符数量
#define UNTERM_NUM 5            // 非终结符数量
#define G_NUM 10                // 产生式数量
#define file_path "lexical.txt"

using namespace std;

// 储存记号流中每个词以及相关信息
typedef struct words
{
    char sign;                  // 记号
    string attribute;           // 属性
}WORDS;
```

```
20.
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)
30.
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.  }
39.
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.  }
48.
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.  }
58.
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. }
68. 
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
77. 
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.     }
88. 
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.     }
103. 
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");
117. 
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(")");
136. 
137.     return;
138. }
139. 
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;
147. 
148.     // 打开由词法分析输出的文件
```

```
    infile.open(file_path);

    // 去除无用符号
    infile >> str;
    infile >> str;

    // 获取记号流数量
    infile >> count;

    // 去除无用符号
    infile >> str;
    infile >> str;

    // 获取记号详细属性与信息
    for (int i = 0; i < count; i++)
    {
        infile >> str;
        if (str == "num")
        {
            tmpWord.sign = 'd';
            infile >> str;
            tmpWord.attribute = str;
        }
        else
        {
            infile >> str;
            tmpWord.sign = str[0];
            tmpWord.attribute = str;
        }
        input.push_back(tmpWord);
    }

    tmpWord.attribute = "$";
    tmpWord.sign = '$';
    input.push_back(tmpWord);

    // 输出得到的记号流(测试用)
    for (int i = 0; i < input.size(); i++)
    {
        cout << input[i].sign << "\t" << input[i].attribute << endl;
    }
}

// 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. }
200. 
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. }
209. 
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
```

```
        {
            table[findIndex(G_LEFT[i])][findIndexT(tmp)] = G_RIGHT[i];
        }
    }
}

// 算法 4.1——预测分析程序
int LLanalyse()
{
    vector<char> LLstack;
    LLstack.push_back('$');
    LLstack.push_back('E');
    int step = 1;           // 记录分析步骤数
    int ip = 0;             // 记录输入串读到的位置
    int i, j;
    string tmpX;
    string tmpa;

    char X = LLstack.back();
    tmpX.clear();
    tmpX.push_back(X);
    char a = input[ip].sign;
    tmpa.clear();
    tmpa.push_back(a);

    while (X != '$')
    {
        cout << "Step " << step << ": " << endl;
        outPutStack(LLstack);
        outPutIn(ip);
        if (isTerm(tmpX))  // 如果是终结符
        {
            if (X == a)  // 匹配成功，弹栈并继续分析
            {
                cout << "\tPop '" << X << "' from stack" << endl;
                LLstack.pop_back();
                ip++;
            }
            else  //匹配失败，报错并结束分析
            {
                cout << "\tError: can't match '" << X << "' and '" << a << "'." << endl;
                return -1;
            }
        }
```

```
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.         }
301.
302.         step++;
303.
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;
```

```
    }
}


int main()
{
    initProgram();
    genTable();

    cout << "table: " << endl;
    cout << "+\t\t-\t\t*\t\t/\t\t(\t\t)\t\td\t\t$" << endl;
    for (int i = 0; i < UNTERM_NUM; i++)
    {
        for (int j = 0; j < TERM_NUM; j++)
            cout << UNTERM[i] << " -> " << table[i][j] << "\t";
        cout << endl;
    }
    cout << endl;

    cout << "input stream: " << endl;
    getInput();
    cout << endl;

    cout << "analyse: " << endl;
    LLanalyse();

    return 0;
}
```

2. LR 分析：

```
#include<iostream>
#include<fstream>
#include<vector>
#include<string>
#include<stack>

#define STATE_NUM 16        // 状态数
#define TERM_NUM 8          // 终结符数量，即 action 列数
#define UNTERM_NUM 3        // 非终结符数量，即 goto 列数
#define G_NUM 9             // 产生式数
```

```
#define file_path "C:/Users/87290/Desktop/PROGRAM/编译原理-词法分析/我的代码/lexical_analysis/lexical.txt"

using namespace std;

// 储存记号流中每个词以及相关信息
typedef struct words
{
    char sign;              // 记号
    string attribute;       // 属性
}WORDS;

// 储存分析表的表项
typedef struct item
{
    string action;          // action 动作(归约 R/移入 S/接受 ACC)或 goto 标记(G)
    int state;              // 属性
}ITEM;


// 储存终结符
vector<string> TERM;
// 储存非终结符
vector<string> UNTERM;
// 左部生成式
string LEFT[] = { "E'", "E", "E", "E", "T", "T", "T", "F", "F"};
// 右部生成式
string RIGHT[] = { "E", "E+T", "E-T", "T", "T*F", "T/F", "F", "(E)", "d" };
// 右部生成式长度
int LEN[] = { 1, 3, 3, 1, 3, 3, 1, 3, 1 };
// 分析表
ITEM table[STATE_NUM][TERM_NUM + UNTERM_NUM];
// 输入流(来自词法分析)
vector<WORDS> input;

// 判断是否为终结符
int isTerm(string str)
{
    for (int i = 0; i < TERM.size(); i++)
        if (str == TERM[i])
            return 1;
    return 0;
}

```

```
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. }
62. 
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. }
72. 
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. }
82. 
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. }
```

```
98.
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. }
107.
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;
115.
116.     // 打开由词法分析输出的文件
117.     infile.open(file_path);
118.
119.     // 去除无用符号
120.     infile >> str;
121.     infile >> str;
122.
123.     // 获取记号流数量
124.     infile >> count;
125.
126.     // 去除无用符号
127.     infile >> str;
128.     infile >> str;
129.
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.         {
```

```
            infile >> str;
            tmpWord.sign = str[0];
            tmpWord.attribute = str;
        }
        input.push_back(tmpWord);
    }

    tmpWord.attribute = "$";
    tmpWord.sign = '$';
    input.push_back(tmpWord);

    // 输出得到的记号流(测试用)
    for (int i = 0; i < input.size(); i++)
    {
        cout << input[i].sign << "\t" << input[i].attribute << endl;
    }
}

// 算法 4.3——LR 分析
int LRanalyse()
{
    vector<int> State;        // 状态栈
    vector<string> Symbol;    // 符号栈
    int step = 1;             // 记录分析步骤数
    int ip = 0;               // 记录输入串读到的位置
    int i, j;
    int S = 0;                // 栈顶状态
    char a;                   // ip 所指符号
    string tmpa;              // a 的 string 形式的副本，便于契合库函数

    State.push_back(0);
    Symbol.push_back("E'");

    a = input[ip].sign;
    tmpa.clear();
    tmpa.push_back(a);

    while (1)
    {
        cout << "Step " << step << ": " << endl;

        if (isTerm(tmpa))
            j = findIndexT(tmpa);
        else
```

```
186.         {
187.             cout << "ERROR: can't find '" << a << "' in Termin Set" << endl;
188.             return -1;
189.         }
190.
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.         }
234.
235.         step++;
236.
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.     }
245.
246.     return 0;
247. }
248.
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", "$" };
256.
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.     }
266.
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;
```

```
272.
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 };
278.
279.    table[1][0] = { "S", 6 };
280.    table[1][1] = { "S", 7 };
281.    table[1][7] = { "ACC", -1 };
282.
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 };
289.
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 };
296.
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 };
302.
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 };
309.
310.    table[6][4] = { "S", 4 };
311.    table[6][6] = { "S", 5 };
312.    table[6][9] = { "G", 11 };
313.    table[6][10] = { "G", 3 };
314.
315.    table[7][4] = { "S", 4 };
```

```
316.     table[7][6] = { "S", 5 };
317.     table[7][9] = { "G", 12 };
318.     table[7][10] = { "G", 13 };
319.
320.     table[8][4] = { "S", 4 };
321.     table[8][6] = { "S", 5 };
322.     table[8][10] = { "G", 13 };
323.
324.     table[9][4] = { "S", 4 };
325.     table[9][6] = { "S", 5 };
326.     table[9][10] = { "G", 14 };
327.
328.     table[10][0] = { "S", 6 };
329.     table[10][1] = { "S", 7 };
330.     table[10][5] = { "S", 15 };
331.
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 };
338.
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 };
345.
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 };
352.
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 };
359.
```

```
    table[15][0] = { "R", 7 };
    table[15][1] = { "R", 7 };
    table[15][2] = { "R", 7 };
    table[15][3] = { "R", 7 };
    table[15][5] = { "R", 7 };
    table[15][7] = { "R", 7 };

    // 输出
    cout << "table:" << endl;
    for (int i = 0; i < STATE_NUM; i++)
    {
        for (int j = 0; j < TERM_NUM + UNTERM_NUM; j++)
            if (table[i][j].state != -1)
                cout << table[i][j].action << table[i][j].state << "\t";
            else
                cout << table[i][j].action << "\t";
        cout << endl;
    }

}


int main()
{
    initProgram();

    cout << "input stream: " << endl;
    getInput();
    cout << endl;

    LRanalyse();

    return 0;
}
```