

# 编译原理实验报告

## 语法分析程序的设计与实现

姓名：胡敏臻

班级：2019211307

学号：2019211424

日期：2021/11/1

## 1 概述

### 1.1 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：

$$E \rightarrow E+T \mid E-T \mid T$$
$$T \rightarrow T * F \mid T / F \mid F$$
$$F \rightarrow (E) \mid \text{num}$$

编写 LL(1) 语法分析程序，要求如下：

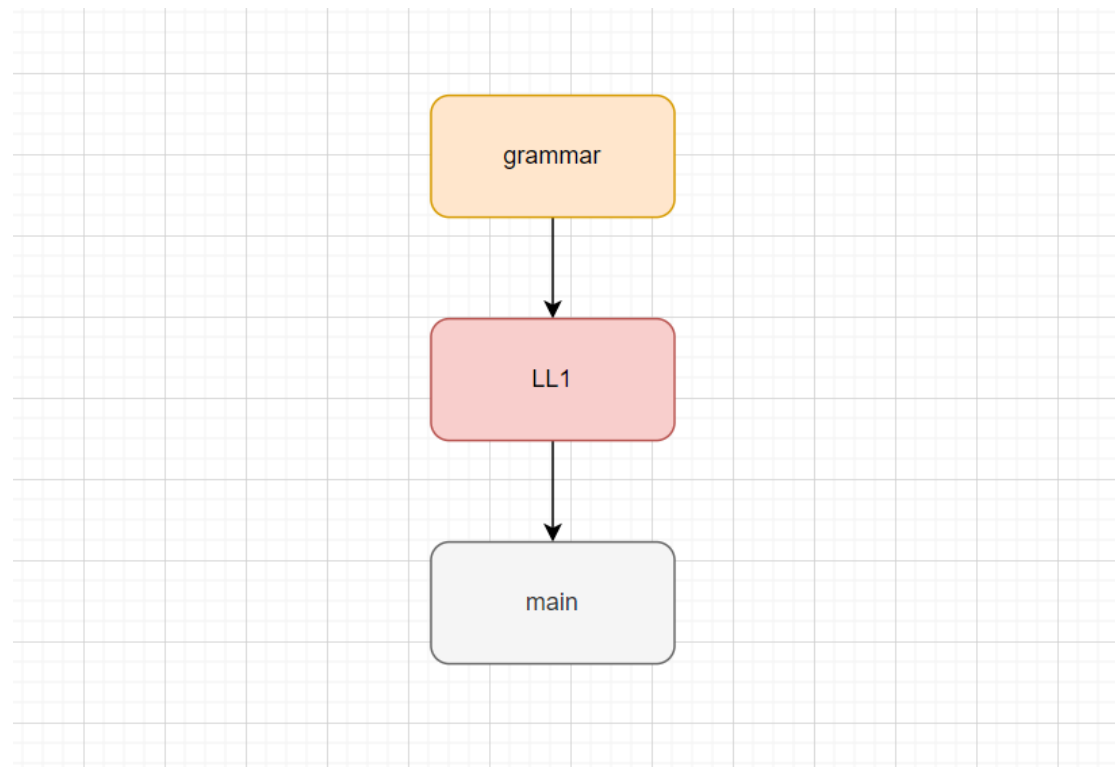
(1) 编程实现算法 4.2，为给定文法自动构造预测分析表；

(2) 编程实现算法 4.1，构造 LL(1) 预测分析程序。

### 1.2 实验环境

Visual Studio 2019

## 2 总体设计



grammar: 存储要识别的语法结构

LL1: 对于识别的语法求出 first 集和 follow 集，并构造预测分析表，利用预测分析程序进行分析并输出相应内容

main: 识别输入的字符串进行识别

## 3 实现说明

### 3.1 grammar 类的设计

grammar 类的设计如下：

```
class grammar
{
```

```

public:
    std::string S;//开始符号
    std::vector<std::string> terminal; //终结符
    std::vector<std::string> non_ter; //非终结符
    std::vector<std::vector<std::vector<std::string>>> all_gra; //所有文法关系

public:
    grammar();
    void read_grammar();
    void delete_left_recur(); //删除左递归
    //void delete_common_factor();
    void test_print_grammar(); //测试打印此时文法
};

```


在 grammar 类中实现了对文法的存储。本程序可以采用两种方式读取文法，一种是将文法关系内置，如下：

```

//初始化
grammar::grammar() {
    S = "E";
    terminal = { "+", "-", "*", "/", "(", ")", "num" };
    non_ter = { "E", "T", "F" };
    all_gra.resize(non_ter.size());
    all_gra[0].resize(3);
    all_gra[0][0] = { "E", "+", "T" };
    all_gra[0][1] = { "E", "-", "T" };
    all_gra[0][2] = { "T" };
    all_gra[1].resize(3);
    all_gra[1][0] = { "T", "*", "F" };
    all_gra[1][1] = { "T", "/", "F" };
    all_gra[1][2] = { "F" };
    all_gra[2].resize(2);
    all_gra[2][0] = { "(", "E", ")" };
    all_gra[2][1] = { "num" };
}

```

另一种方法是通过输入行输入，通过调用 read\_grammar() 的函数进行存储，以题目所给条件为例，以如下方式进行存储：

 F:\zzz\AllCode\编译原理\语法分析\语法分析\Debug\语法分析.exe

```

请输入开始符号
E
请输入非终结符个数及非终结符
3
E T F
请输入终结符个数及终结符
7
+ - * / ( ) num
请输入产生式行数及产生式 产生式以#结尾
3
E -> E + T | E - T | T #
T -> T * F | T / F | F #
F -> ( E ) | num #

```

因为在题目所给条件中，我们的文法是确定的，因此我们采用了内置的方法进行处理。并且为了让该程序更具有普遍性，我们同样还是保留了读取文法这个接口。

在该文法中我们读取之后，发现此文法是左递归文法，因为我们需要对文法做消除左递归操作，根据消除左递归算法，当出现式子  $A \rightarrow A\alpha \mid \beta$  时，我们可以做如下操作去消除左递归：

$$\begin{aligned} A &\rightarrow \beta A' \\ A' &\rightarrow \alpha A' \mid \varepsilon \end{aligned}$$

代码实现如下展示：

```

80 //消除左递归
81 void grammar::delete_left_recur() {
82     std::string temps;
83     int n = all_gra.size();
84     int flag = 0;
85     for (int i = 0; i < n; i++) {
86         temps = non_ter[i];
87         flag = 0;
88         for (int j = 0; j < all_gra[i].size(); j++) {
89             if (all_gra[i][j][0] == temps) {
90                 flag = 1;
91                 non_ter.push_back(temps + '\');
92                 break;
93             }
94         }
95     }
96     if (flag == 1) {
97         int newn = non_ter.size();
98         all_gra.resize(newn);
99         int tempn = all_gra[i].size();
100        for (int j = 0; j < all_gra[i].size(); j++) {
101            if (all_gra[i][j][0] == temps) {
102                all_gra[i][j].erase(all_gra[i][j].begin());
103                all_gra[i][j].push_back(non_ter[newn - 1]);
104                all_gra[newn-1].push_back(all_gra[i][j]);
105                all_gra[i].erase(all_gra[i].begin() + j);
106                j--;
107            }
108            else {
109                all_gra[i][j].push_back(non_ter[newn - 1]);
110            }
111        }
112        if (find(terminal.begin(), terminal.end(), "epsilon") == terminal.end()) {
113            terminal.push_back("epsilon");
114        }
115        std::vector<std::string> tempss;
116        tempss.push_back("epsilon");
117        all_gra[newn - 1].push_back(tempss);
118    }
119 }
120

```

其中，我们用 test\_print\_grammar() 这个函数去测试对文法操作之后的正确性。例如在做完消除左递归之后，文法结果如下：

```

E->TE'
T->FT'
F->(E) | num
E' ->+TE' | -TE' | epsilon
T' ->*FT' | /FT' | epsilon

```

### 3.2 LL1 类的设计

LL1 类的设计如下：

```

class LL1
{
    grammar Gra; //实例化Gra类
    int len_non = Gra.non_ter.size(); //非终结符的长度
    int len_ter = Gra.terminal.size(); //终结符长度
}

```

```

std::vector<std::vector<std::string>> first; //first集
std::vector<std::vector<std::string>> follow; //follow集
std::vector<int> has_first; //是否已有first集, 0表示无, 1表示有
std::vector<int> has_follow; //是否已有follow集, 0表示无, 1表示有
std::vector<std::vector<int>> include_of; //表示在follow集中的递归调用关系
std::vector<std::vector<std::vector<std::string>>> table; //预测分析表
std::vector<std::string> pre_stack; //预测分析栈
std::vector<std::string> parse_s; //分析字符串
public:
    void init(); //初始化
    void get_first(); //得到first集
    void get_first_element(int symbol); //得到first集的每个元素
    void print_first(); //打印first集
    void get_follow(); //得到follow集
    void get_follow_element(int symbol); //得到follow集的每个元素
    void print_follow(); //打印follow集
    void get_table(); //得到预测分析表
    void add_error_table(); //增加错误处理
    void print_table(); //打印预测分析表
    void parse(std::string s); //分析程序
    void parse_string(std::string s); //打印字符串
    void print_parse(int pi, std::vector<std::string> s); //预处理分析程序中此时的字符串, 主要是用dfa模型分析数字
};

```

LL(1)语法分析的大部分工作都是在 LL1 类中完成的,包括计算 first 集与 follow 集,构造预测分析表,进行预测分析,利用 dfa 模型来识别输入的字符都在此类中完成。

(1) **first 集的实现在书第 90 页给出了算法, 如下展示:**

- 若  $X \in V_T$ , 则  $FIRST(X) = \{X\}$ 。
- 若  $X \in V_N$ , 且有产生式  $X \rightarrow a \dots$ , 其中  $a \in V_T$ , 则把  $a$  加入到  $FIRST(X)$  中。  
若有产生式  $X \rightarrow \epsilon$ , 则  $\epsilon$  也加入到  $FIRST(X)$  中。
- 若有产生式  $X \rightarrow Y \dots$ , 且  $Y \in V_N$ , 则把  $FIRST(Y)$  中的所有非  $\epsilon$  元素加入到  $FIRST(X)$  中。  
若有产生式  $X \rightarrow Y_1 Y_2 \dots Y_k$ , 如果对某个  $i$ ,  $FIRST(Y_1), FIRST(Y_2), \dots, FIRST(Y_{i-1})$  都含有  $\epsilon$ , 即  $Y_1 Y_2 \dots Y_{i-1} \Rightarrow \epsilon$ , 则把  $FIRST(Y_i)$  中的所有非  $\epsilon$  元素加入到  $FIRST(X)$  中。

根据该算法, 进行代码的编写, 因为涉及到递归的调用, 因此用了一个数组 has\_first 来标记这个非终结符是否已被计算。在计算时套用了多层循环, 第一层循环为非终结符, 第二层遍历其产生式, 第三层遍历产生式中各个部分。代码实现如下:

```

void LL1::get_first() {
    init();
    for (int i = 0; i < len_non; i++) {
        if (has_first[i] != 0) {
            continue;
        }
        get_first_element(i);
    }
}

```

```

void L1::get_first_element(int symbol) {
    int i = symbol;
    for (int j = 0; j < Gra.all_gra[i].size(); j++) { //遍历其产生的每一个产生式
        if (find(Gra.terminal.begin(), Gra.terminal.end(), Gra.all_gra[i][j][0]) != Gra.terminal.end()) { //第一个就是终结符
            if (find(first[i].begin(), first[i].end(), Gra.all_gra[i][j][0]) == first[i].end()) { //如果first集中没有则加入
                first[i].push_back(Gra.all_gra[i][j][0]);
                sort(first[i].begin(), first[i].end());
            }
        }
        else { //当第一位是非终结符时
            std::vector<std::string> s = Gra.all_gra[i][j]; //s为当前产生式
            std::vector<std::string> temps_u;
            std::vector<std::string> temps_k;
            for (int k = 0; k < s.size(); k++) { //遍历某一个产生式
                if (find(Gra.terminal.begin(), Gra.terminal.end(), s[k]) != Gra.terminal.end()) { //如果是终结符,则直接退出
                    if (find(first[i].begin(), first[i].end(), s[k]) == first[i].end()) { //如果first集中没有则加入
                        first[i].push_back(Gra.all_gra[i][j][k]);
                        sort(first[i].begin(), first[i].end());
                    }
                }
                else { //是非终结符
                    break;
                }
            }
            int pk = std::distance(Gra.non_ter.begin(), find(Gra.non_ter.begin(), Gra.non_ter.end(), s[k])); //s[k]在非终结符中的下标
            if (has_first[pk] != 0) { //如果是已经计算过的
                temps_k = first[pk];
                //与现有的做并集
                std::set_union(first[i].begin(), first[i].end(), first[pk].begin(), first[pk].end(), std::back_inserter(temps_u));
                first[i] = temps_u;
                if (find(temps_k.begin(), temps_k.end(), "epsilon") == temps_k.end()) { //如果此情况不包括空集,则退出
                    break;
                }
                else if (k == s.size() - 1) { //如果包括空集且是表达式最后一位
                    if (find(first[i].begin(), first[i].end(), "epsilon") == first[i].end()) { //判断有没有空集,没有则加入
                        first[i].push_back("epsilon");
                        sort(first[i].begin(), first[i].end());
                    }
                }
            }
            else { //还未计算过的
                get_first_element(pk);
                k--;
            }
        }
    }
    has_first[symbol] = 1;
}

```

(2) 构造 follow 集的方法与 first 集类似, 同样都是采用书上给出的如下方法。

- 对文法开始符号  $S$ , 置  $\$$  于 FOLLOW( $S$ ) 中。
- 若有产生式  $A \rightarrow \alpha B \beta$ , 则把 FIRST( $\beta$ ) 中的所有非  $\epsilon$  元素加入到 FOLLOW( $B$ ) 中。
- 若有产生式  $A \rightarrow \alpha B$ , 或有产生式  $A \rightarrow \alpha B \beta$ , 但是  $\epsilon \in \text{FIRST}(\beta)$ , 则把 FOLLOW( $A$ ) 中的所有元素加入到 FOLLOW( $B$ ) 中。

需要注意的是 follow 集会出现循环递归的情况, 此时程序会陷入崩溃或者死循环, 需要特别注意。因此我们用了 include\_of 数组来表示不同终结符的调用关系, 若存在互相调用的情况出现, 我们就让这两个非终结符的 follow 集相等, 具体代码如下。

```

//follow[j]与follow[i]做交集
if (has_follow[j] == 0) {
    if (include_of[i][j] == 1 && include_of[j][i] == 1) { //互相递归调用
        std::vector<std::string> temps_u;
        std::set_union(follow[j].begin(), follow[j].end(), follow[i].begin(), follow[i].end(), std::back_inserter(temps_u));
        follow[i] = temps_u;
        follow[j] = temps_u;
    }
}

```

同时需要注意的是, 这时求出来的 follow 集中包括 epsilon 的元素, 在求取完 follow 集之后需要去除 epsilon 元素。

```

for (int i = 0; i < len_non; i++) {
    auto it = find(follow[i].begin(), follow[i].end(), "epsilon");
    if (it != follow[i].end()) {
        follow[i].erase(it);
    }
}

```

(3) 此时第3步需要**构造预测分析表**，首先因为预测分析表中包括的是”\$”符，因此在一开始我们需要整理终结符，将空字符串替换为”\$”符号。同时运用书上构造预测分析表的算法进行计算。

```
for(文法 G 的每一个产生式  $A \rightarrow \alpha$ ) {
    for(每个终结符号  $a \in \text{FIRST}(\alpha)$ ) 把  $A \rightarrow \alpha$  放入表项  $M[A, a]$  中;
    if( $\epsilon \in \text{FIRST}(\alpha)$ )
        for(每个  $b \in \text{FOLLOW}(A)$ ) 把  $A \rightarrow \alpha$  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。
```

实现的代码部分如下：

```
for (int i = 0; i < len_non; i++) {
    for (int j = 0; j < Gra.all_gra[i].size(); j++) {
        for (int k = 0; k < Gra.all_gra[i][j].size(); k++) {
            std::string s = Gra.all_gra[i][j][k];
            //如果是空，将空加入follow[i]
            if (s == "epsilon") {
                for (int m = 0; m < follow[i].size(); m++) {
                    auto pf = find(Gra.terminal.begin(), Gra.terminal.end(), follow[i][m]);
                    int dpf = std::distance(Gra.terminal.begin(), pf);
                    table[i][dpf] = Gra.all_gra[i][j];
                }
                break;
            }
            //如果是终结符
            else if (find(Gra.terminal.begin(), Gra.terminal.end(), s) != Gra.terminal.end()) {
                int pi = distance(Gra.terminal.begin(), find(Gra.terminal.begin(), Gra.terminal.end(), s));
                table[i][pi] = Gra.all_gra[i][j];
                break;
            }
            //如果是非终结符
            else {
                auto pf = find(Gra.non_ter.begin(), Gra.non_ter.end(), s);
                int dpf = std::distance(Gra.non_ter.begin(), pf);
                int flag = 0;
                for (int m = 0; m < first[dpf].size(); m++) {
                    if (first[dpf][m] == "epsilon") {
                        flag = 1;
                    }
                    else {
                        int pi = distance(Gra.terminal.begin(), find(Gra.terminal.begin(), Gra.terminal.end(), first[dpf][m]));
                        table[i][pi] = Gra.all_gra[i][j];
                    }
                }
                if (flag == 0) {
                    break;
                }
            }
        }
    }
}
```

构造出的预测分析表如下展示：

	+	-	*	/	(	)	num	\$
E					$E \rightarrow TE'$		$E \rightarrow TE'$	
T					$T \rightarrow FT'$		$T \rightarrow FT'$	
F					$F \rightarrow (E)$		$F \rightarrow \text{num}$	
$E'$	$E' \rightarrow +TE'$	$E' \rightarrow -TE'$					$E' \rightarrow \text{epsilon}$	$E' \rightarrow \text{epsilon}$
$T'$	$T' \rightarrow \text{epsilon}$	$T' \rightarrow \text{epsilon}$	$T' \rightarrow *FT'$	$T' \rightarrow /FT'$			$T' \rightarrow \text{epsilon}$	$T' \rightarrow \text{epsilon}$

(4) 同时为了能够实现进行一定的**错误处理**功能，我们对错误的输入进行处理。第一种处理错误的方法是弹出栈顶的终结符号，第二种情况是跳过剩余输入字符串中的若干个符号直到可以继续进行分析为止。我们通过判断在该状态读取到字符的 follow 集，给转移表格中空的地方加入”synch”，表示遇到这个字符需要弹出栈顶的符号。若分析表为空，我们采取第二种处理方式，跳过该字符。在 LL1 中 add\_error\_table() 函数就是完成该功能。实现的代码如下展示。

```

//增加错误处理
void LL1::add_error_table() {
    for (int i = 0; i < len_non; i++) {
        for (int j = 0; j < follow[i].size(); j++) {
            if (table[i][j].size() == 0) {
                table[i][j].push_back("synch");
            }
        }
    }
}

```

在加过错误处理之后的**预测分析表**如下展示，不足的是没有做好对齐处理。

	+	-	*	/	(	)	num	\$	
E	synch	synch			E→TE'		E→TE'		
T	synch	synch	synch	synch	T→FT'		T→FT'		
F	synch	synch	synch	synch	F→(E)	synch	F→num		
E'	E'→+TE'		E'→-TE'				E'→epsilon		E'→epsilon
T'	T'→epsilon		T'→epsilon		T'→*FT'		T'→/FT'	T'→epsilon	T'→epsilon

(5) 接下来我们需要准备做预测分析程序了，但是在做之前我们需要对读入的字符串先进行预处理。在本程序中特别**预处理的是数字类型**。因为在数字类型中有整数，小数，科学计数等形式，我们都要将其识别为 num，因此我们利用词法分析时的 dfa 转换图。同时为了方便标识这是 num 类型，我们在记录其值的时候在前面加上字符 '0' 标识这是 num 类型，这样就不用构造新的变量了。具体实现在 parse\_string(std::string s) 函数中，这部分内同与词法分析类似，因此代码就不在此贴出，具体内容见附件中的代码。在样例测试中，我们将会对不同类型的数字进行测试。

(6) 最后一步，在完成了预测分析表和字符串预处理之后，我们终于要开始进行预测分析了，预测分析的过程运用了书上 P89 页的算法，如下：

首先，初始化，即将 \$ 压入栈底，将文法开始符号 S 压入栈顶；将  $\omega$  放入输入缓冲区中，并置向前指针 ip 指向  $\omega$  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  $\omega$  作出自顶向下的分析，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 ( M[X,a]=X→Y1Y2...Yk ) {
            从栈顶弹出 X;
            依次把 Yk,Yk-1,...,Y2,Y1 压入栈;
            输出产生式 X→Y1Y2...Yk;
        };
        else error();
    } while (X!= $)

```

按照这个思路，代码实现如下。在代码中我们可以看到用 tempa[0] 是否为 '0' 来标注这个类型是否为 num 类型。并且对于处理到空的时候，会判断此时是错误并做了什么错误处理，这使得程序的鲁棒性更好，在遇到错误输入的时候仍可以继续识别。并需要注意此时 while 中判断的条件需要加上 (ip<len)，用来判断此时虽然栈还没空，但是字符输入的指示已经指向字符串末尾了。



```

do {
    topx = pre_stack.back();
    tempa = parse_s[ip];
    auto terp = find(Gra.terminal.begin(), Gra.terminal.end(), topx);
    int terni = std::distance(Gra.non_ter.begin(), find(Gra.non_ter.begin(), Gra.non_ter.end(), tempa));
    int terai = std::distance(Gra.terminal.begin(), find(Gra.terminal.begin(), Gra.terminal.end(), tempa));
    if (tempa[0] == '0') {
        terai = std::distance(Gra.terminal.begin(), find(Gra.terminal.begin(), Gra.terminal.end(), "num"));
    }

    print_parse(ip, parse_s);
    if (terp != Gra.terminal.end() || topx == "$") {
        if (topx == tempa) {
            pre_stack.pop_back();
            ip++;
            std::cout << "匹配成功\n";
        }
        else if (topx == "num" && tempa[0] == '0') {
            pre_stack.pop_back();
            ip++;
            std::cout << "匹配成功\n";
        }
        else {
            std::cout << "error!处理下一字符\n";
            //break;
            ip++;
        }
    }
}

else {
    if (table[terni][terai].size() != 0) {
        pre_stack.pop_back();
        if (table[terni][terai][0] == "synch") {
            std::cout << "error!字符退栈处理\n";
        }
        else {
            if (table[terni][terai][0] != "epsilon") {
                for (int i = table[terni][terai].size() - 1; i >= 0; i--) {
                    pre_stack.push_back(table[terni][terai][i]);
                }
            }
            std::cout << Gra.non_ter[terni] << "->";
            for (int i = 0; i < table[terni][terai].size(); i++) {
                std::cout << table[terni][terai][i];
            }
            std::cout << "\n";
        }
    }
    else {
        std::cout << "error!处理下一字符\n";
        ip++;
        //break;
    }
}

std::cout << "\n";
} while (topx != "$" && ip < len);

```

至此，语法分析的部分就已经全部完成了。在该类中以 print 开头的函数都是在编写程序时用来检查每一部分是否正确的。比如检验 first 集与 follow 集，预测分析表等。在进行语法分析时，也将每一个过程的栈、字符串识别情况、输出打印出来。

## 4 测试样例说明

### 4.1 输入整数运算式

输入：(2)+3\*(5/6)

输出：

```

$E      (2)+3*(5/6)$      E->TE'
$E' T    (2)+3*(5/6)$      T->FT'
$E' T' F  (2)+3*(5/6)$      F->(E)
$E' T')E( (2)+3*(5/6)$      匹配成功
$E' T')E  2)+3*(5/6)$      E->TE'
$E' T')E' T  2)+3*(5/6)$      T->FT'
$E' T')E' T' F  2)+3*(5/6)$      F->num
$E' T')E' T' num  2)+3*(5/6)$      匹配成功
$E' T')E' T' )+3*(5/6)$      T'->epsilon
$E' T')E' )+3*(5/6)$      E'->epsilon
$E' T') )+3*(5/6)$      匹配成功
$E' T' +3*(5/6)$      T'->epsilon
$E' +3*(5/6)$      E'->TE'
$E' T+ +3*(5/6)$      匹配成功
$E' T 3*(5/6)$      T->FT'
$E' T' F 3*(5/6)$      F->num
$E' T' num 3*(5/6)$      匹配成功
$E' T' * (5/6)$      T'->*FT'
$E' T' F* * (5/6)$      匹配成功
$E' T' F (5/6)$      F->(E)
$E' T')E( (5/6)$      匹配成功
$E' T')E 5/6)$      E->TE'
$E' T')E' T 5/6)$      T->FT'
$E' T')E' T' F 5/6)$      F->num
$E' T')E' T' num 5/6)$      匹配成功
$E' T')E' T' /6)$      T' ->/FT'
$E' T')E' T' F/ /6)$      匹配成功
$E' T')E' T' F 6)$      F->num

$E' T')E' T' num 6)$      匹配成功
$E' T')E' T' )$      T'->epsilon
$E' T')E' )$      E'->epsilon
$E' T' )$      匹配成功
$E' T' $      T'->epsilon
$E' $      E'->epsilon
$      $      匹配成功

```

## 4.2 输入有许多括号的

输入：((((150))))

输出：

```

$E      ((((((150))))))$      E->TE'
$E' T    ((((((150))))))$      T->FT'
$E' T' F  ((((((150))))))$      F->(E)
$E' T')E( ((((((150))))))$      匹配成功
$E' T')E  ((((((150))))))$      E->TE'
$E' T')E' T  ((((((150))))))$      T->FT'
$E' T')E' T' F  ((((((150))))))$      F->(E)
$E' T')E' T')E( ((((((150))))))$      匹配成功
$E' T')E' T')E  ((((((150))))))$      E->TE'
$E' T')E' T')E' T  ((((((150))))))$      T->FT'
$E' T')E' T')E' T' F  ((((((150))))))$      F->(E)
$E' T')E' T')E' T')E( ((((((150))))))$      匹配成功
$E' T')E' T')E' T')E  ((((((150))))))$      E->TE'
$E' T')E' T')E' T')E' T  ((((((150))))))$      T->FT'
$E' T')E' T')E' T')E' T' F  ((((((150))))))$      F->(E)
$E' T')E' T')E' T')E' T')E( ((((((150))))))$      匹配成功
$E' T')E' T')E' T')E' T')E  ((((((150))))))$      E->TE'
$E' T')E' T')E' T')E' T')E' T  ((((((150))))))$      T->FT'
$E' T')E' T')E' T')E' T')E' T' F  ((((((150))))))$      F->num
$E' T')E' T')E' T')E' T')E' T' num  ((((((150))))))$      匹配成功
$E' T')E' T')E' T')E' T')E' T' )$      T'->epsilon
$E' T')E' T')E' T')E' T')E' T' )$      E'->epsilon
$E' T')E' T')E' T')E' T')E' T' )$      匹配成功
$E' T')E' T')E' T')E' T')E' T' )$      T'->epsilon

```

```

$E'T')E'T')E'T')E'T')E'T'      ))))$      T'->epsilon
$E'T')E'T')E'T')E'T')E'      ))))$      E'->epsilon
$E'T')E'T')E'T')E'T')      ))))$      匹配成功
$E'T')E'T')E'T')E'T'      ))))$      T'->epsilon
$E'T')E'T')E'T')E'      ))))$      E'->epsilon
$E'T')E'T')E'T')      ))))$      匹配成功
$E'T')E'T')E'T'      ))$      T'->epsilon
$E'T')E'T')E'      ))$      E'->epsilon
$E'T')E'T')      ))$      匹配成功
$E'T')E'T'      )$      T'->epsilon
$E'T')E'      )$      E'->epsilon
$E'T')      )$      匹配成功
$E'T'      $      T'->epsilon
$E'      $      E'->epsilon
$      $      匹配成功

```

#### 4.3 输入有小数的

输入: 1.5\*2.8-3.7

输出:

```

请输入要识别的字符串
1.5*2.8-3.7
$E      1.5*2.8-3.7$      E->TE'
$E' T      1.5*2.8-3.7$      T->FT'
$E' T' F      1.5*2.8-3.7$      F->num
$E' T' num      1.5*2.8-3.7$      匹配成功
$E' T'      *2.8-3.7$      T'->*FT'
$E' T' F*      *2.8-3.7$      匹配成功
$E' T' F      2.8-3.7$      F->num
$E' T' num      2.8-3.7$      匹配成功
$E' T'      -3.7$      T'->epsilon
$E'      -3.7$      E'->-TE'
$E' T-      -3.7$      匹配成功
$E' T      3.7$      T->FT'
$E' T' F      3.7$      F->num
$E' T' num      3.7$      匹配成功
$E' T'      $      T'->epsilon
$E'      $      E'->epsilon
$      $      匹配成功

```

#### 4.4 输入有科学计数的

输入: 1.5E-3+17e5

输出:

```

$E      1.5E-3+17e5$      E->TE'
$E' T      1.5E-3+17e5$      T->FT'
$E' T' F      1.5E-3+17e5$      F->num
$E' T' num      1.5E-3+17e5$      匹配成功
$E' T'      +17e5$      T'->epsilon
$E'      +17e5$      E'->+TE'
$E' T+      +17e5$      匹配成功
$E' T      17e5$      T->FT'
$E' T' F      17e5$      F->num
$E' T' num      17e5$      匹配成功
$E' T'      $      T'->epsilon
$E'      $      E'->epsilon
$      $      匹配成功

```

#### 4.5 输入弹栈处理错误的和字符串移动的

输入: ((2+)+333

输出:

```

((2+)+333
$E      ((2+)+333$      E->TE'
$E' T   ((2+)+333$      T->FT'
$E' T' F ((2+)+333$      F->(E)
$E' T')E((2+)+333$      匹配成功
$E' T')E(2+)+333$      E->TE'
$E' T')E' T (2+)+333$      T->FT'
$E' T')E' T' F (2+)+333$      F->(E)
$E' T')E' T')E((2+)+333$      匹配成功
$E' T')E' T')E 2+)+333$      E->TE'
$E' T')E' T')E' T 2+)+333$      T->FT'
$E' T')E' T')E' T' F 2+)+333$      F->num
$E' T')E' T')E' T' num 2+)+333$      匹配成功
$E' T')E' T')E' T' +)+333$      T' ->epsilon
$E' T')E' T')E' T' +)+333$      E' ->+TE'
$E' T')E' T')E' T+ +)+333$      匹配成功
$E' T')E' T')E' T' )+333$      error!处理下一字符
$E' T')E' T')E' T' +333$      error!字符退栈处理
$E' T')E' T')E' T' +333$      E' ->+TE'
$E' T')E' T')E' T+ +333$      匹配成功
$E' T')E' T')E' T' 333$      T->FT'
$E' T')E' T')E' T' F 333$      F->num
$E' T')E' T')E' T' num 333$      匹配成功
$E' T')E' T')E' T' $ T' ->epsilon
$E' T')E' T')E' T' $ E' ->epsilon
$E' T')E' T') $ error!处理下一字符

```

## 5 实验总结

本次实验让我们编写了 LL(1) 语法分析，在有了编写词法分析的经验之后，上手编写语法分析变得简单了。不过这次的难点在于怎么把书上的算法用编程的方式表达出来，包括应该怎么设置数据，怎么存储信息量，怎么处理多重递归的问题。这和手算预测分析表还是有很大不同的。

本次实验依然有许多待改进的地方。例如我们已经完成了词法分析编写的程序了，写语法分析的时候可以考虑是否综合使用，这样在识别数字或者标识符的时候就不用重新编写 dfa 识别程序了。并且本次只是识别了特定的文法，在面对复杂的文法规则时，仅消除左递归是不够的，还需要消除公共左因子，因为这一点较难实现，在程序中并没有实现，想法是通过二叉树的方式搜寻公共左因子，这一点也是有待改进的。

综上，本次实验让我对 LL(1) 语法分析有了更为深入的理解，对其更为熟悉了，并且对于如何将算法转变为编程语言也有了新的锻炼，是十分不错的一次实验。