# 算术表达式语法分析器实验

**姓名：吕文禧　　学号：19335151**

# 实验要求

> ## 实验描述
>
> - 算术表达式语法分析器的设计与实现
>
> ## 实验要求
>
> - 使用LL(1)分析法和LR分析法设计实现算术表达式的语法分析器
> - 算数表达式至少支持加减乘除以及括号操作，即（+，-，*，/，（））。
>
> ## 提交内容
>
> 1. 实验报告，报告内容必须包含：
>
> - 算术表达式所依据的文法；
> - LL(1)和LR分析法所使用的分析表，以及简要分析；
> - 程序执行流程；
> - 程序运行结果展示。
>
> 2. 语法分析源程序：source.c（源程序包）
> 3. 可执行文件
> 4. 程序测试文件：test.txt（实验输入，将测试案例写入程序的可没有此项）

# 实验思路

## 功能设计

> 本次实验要求设计一个可以识别算术表达式语法的语法分析器，对于语法正确的算术表达式，返回分析正确，若算术表达式中有语法错误则需要返回错误分析报告，算数表达式至少支持加减乘除以及括号操作，即（+，-，*，/，（））。

对于本次实验，设计了一个基本满足C语言算术表达式的语法分析器，支持负数、自增、自减 (e.g. : i++, ++i, --i, i--) 运算操作的语法，基本全部的二元运算符以及括号操作。对于以下语法，能够正确判断表达式中有无语法错误，并且能够准确定位错误可能发生的地方。

$$
\begin{aligned}
E &\to T \mid E\omega_0 T \\
T &\to F \mid T\omega_1 F \\
F &\to I \mid (E)
\end{aligned}
$$

实验所设计的语法分析器以词法分析器作为上游，将词法分析器解析得到的token序列作为输入，分析token序列中有无不合上述语法的错误。只对算术表达式语法正确性进行分析，不对表达式语义是否符合逻辑负责。

# LL(1)设计

## 文法修改

由于算术表达式的原文法存在左递归，不属于LL1文法，需要修改得到以下LL1文法：

$$E \to TE_1 \quad (1)$$
$$E_1 \to \omega_0 TE_1\ (2) \mid \varepsilon\ (3)$$
$$T \to FT_1 \quad (4)$$
$$T_1 \to \omega_1 FT_1\ (5) \mid \varepsilon\ (6)$$
$$F \to I\ (7) \mid (E)\ (8)$$

得到LL1文法后，求得每个产生式的 select 集， 以获取LL1分析表。

| 产生式 | select集 |
| --- | --- |
| 1 | {l，( } |
| 2 | {ω0} |
| 3 | { ) , #} |
| 4 | {l，( } |
| 5 | {ω1} |
| 6 | {ω0, ) , #} |
| 7 | {l} |
| 8 | { ( } |

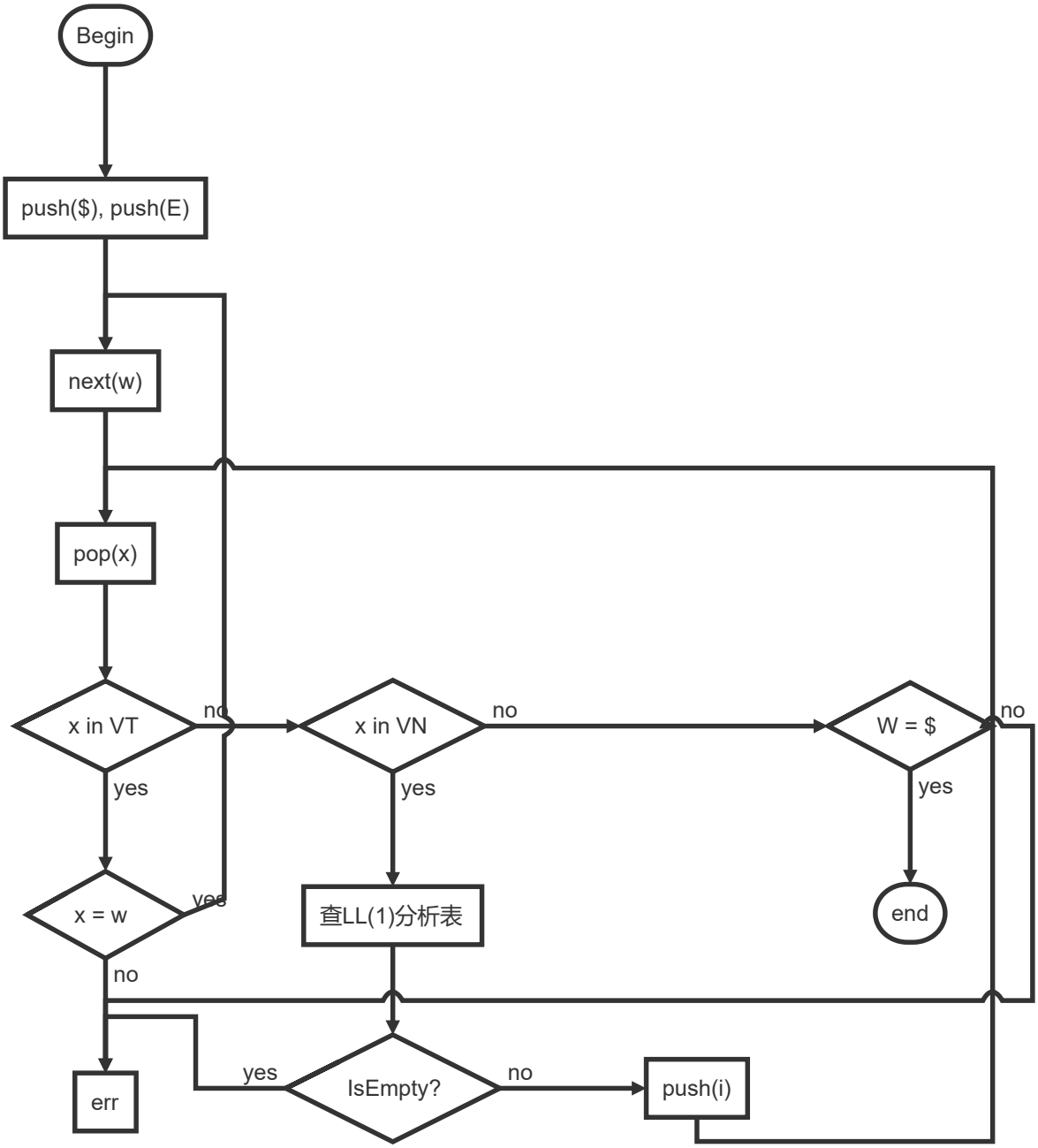
## LL1分析表

| | l | ω0 | ω1 | ( | ) | # |
| --- | --- | --- | --- | --- | --- | --- |
| $E$ | 1 | | | 1 | | |
| $E_1$ | | 2 | | | 3 | 3 |
| $T$ | 4 | | | 4 | | |
| $T_1$ | | 6 | 5 | | 6 | 6 |
| $F$ | 7 | | | 8 | | |

## 分析流程

对于LL1的语法分析器，需要一个分析栈一张分析表，一开始向栈内压入#E作为起始状态。开始时读入一个token，检查栈顶元素，若栈顶元素为非终结符Vn，则从LL1分析表中查找对应产生式的右部，若存在则逆序压入栈中，继续查看下一个栈顶元素而不移进字符，若该表项为空则返回错误信息；若栈顶元素为终结符Vt，则与当前移进的字符检查匹配，若不匹配则返回错误信息，若匹配则继续移进字符。重复上述步骤直至出现错误或者匹配到终结符`#`。

语法分析器的程序逻辑如下：




## 负数处理

在LL1分析中处理负数，我们在移进`-`后发现栈顶元素为`I`时，就判定该`-`为负号而不是减号，这时需要向后多看一位字符，对负数的形式分为三种情况讨论：符号后接变量，负号后接常量，负号后接（...）。前两种情况比较简单，分析器只需简单地移进两个符号，即把形如`-a`，`-100`的符号视为一个终结符`I`，而负号后接（...）则需要检查括号构成的子表达式是否符合语法，此处将括号形成的子表达式切片出来，使用递归检查即可。

对于`--`,`++`符号，我实现的语法规定`--`,`++`符号只可以接变量，这使得`--`,`++`符号的处理简单许多，只需在栈顶元素为`I`，且移进`--`,`++`符号时向后多看一位，若是变量则直接移进，否则返回错误信息。

# LR(0)设计

## DFA设计

开始时使用下标方法设计DFA，得到一个包含冲突有限状态机，并且难以解决冲突问题，因为其follow集与相冲突状态的转换边有交集。

所以采用求状态闭包与goto函数构造DFA，具体算法为

1. 求{$S' \to \cdot S$}的闭包，得到初态项目集$I_0$；
2. 对初态项目集和其他项目集，应用状态转移函数goto（I,X）求出新的项目集；
3. 重复第2步直到不出现新项目集。

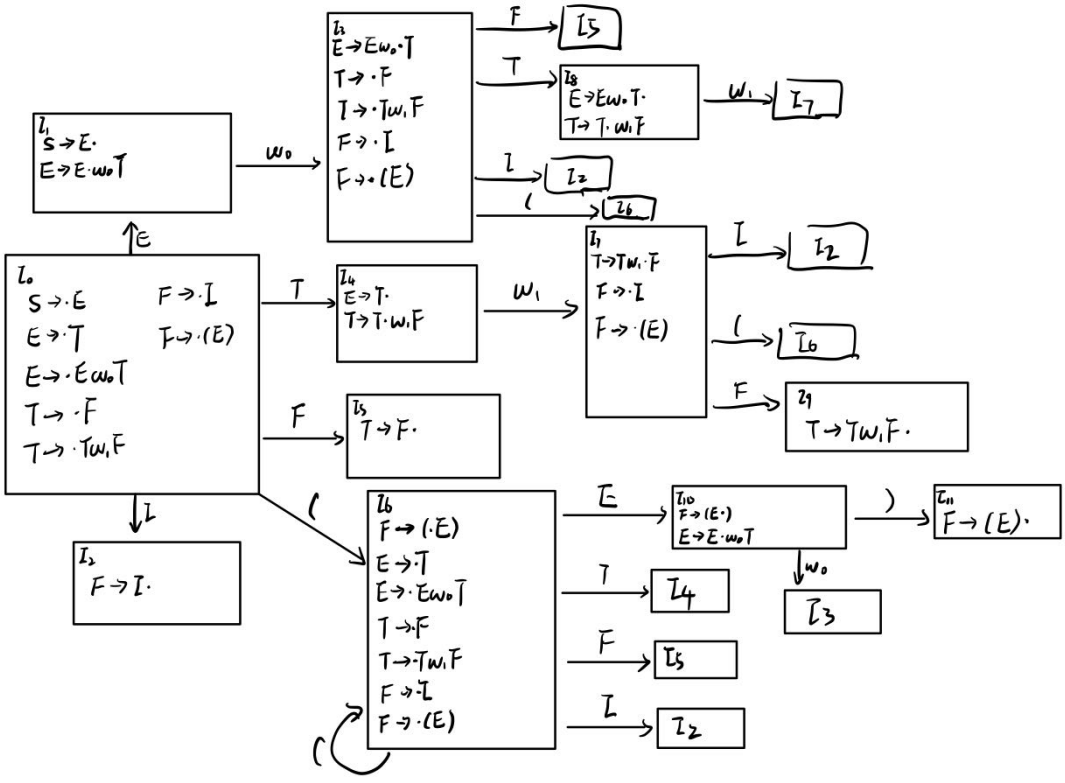
状态的闭包定义如下：对闭包函数closure(I)

1. $I \in closure(I)$;
2. 若项目
   $A \to \alpha \cdot B\beta \in closure(I)$,则对所有左侧为$B$的产生式$B \to r \in P$,项目$B \to \cdot r \in closure(I)$
   ;
3. 重复2直至closure(I)不再增大。

goto函数定义：

1. $goto(I, X) = closure(J), J = \{A \to \alpha X \cdot \beta \mid A \to \alpha \cdot X\beta \in I\}$

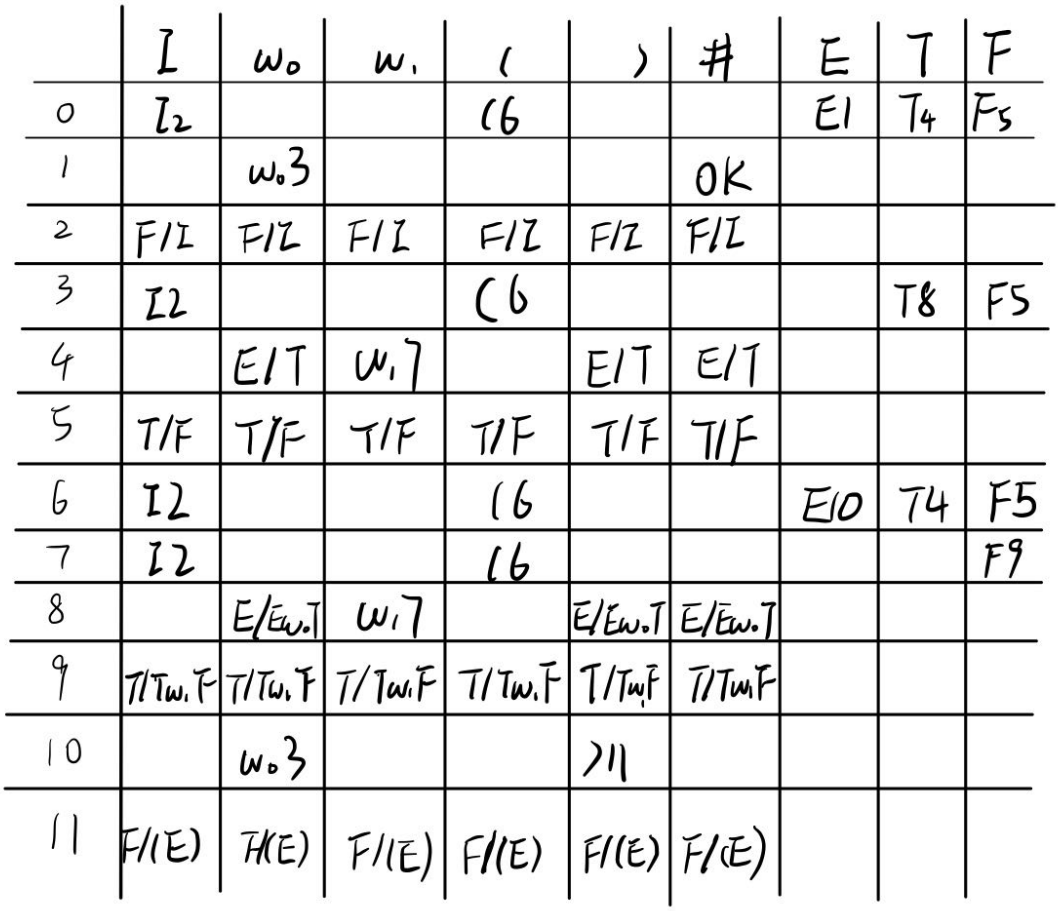
通过上述算法可以得到如下DFA




## LR分析表

上述DFA也有冲突，观察可知$I_4, I_8$状态既有规约项目，也有移进项目，这时我们对规约项目的左部求follow集可得 $follow(E) = \{\omega_0, ), 终止符\}$，与状态转移边无交集，所以只需在相应表项中填入对应规约的产生式即可。

由上述DFA可构建得到如下LR分析表

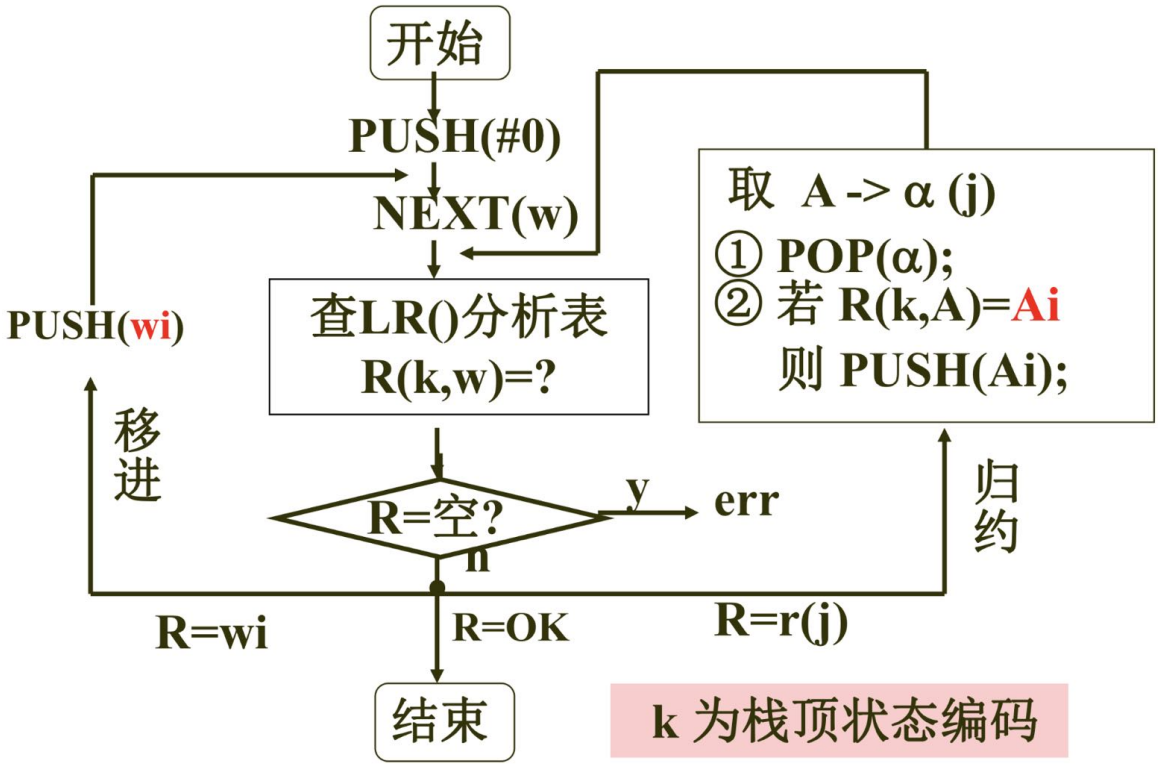


## 分析流程

LR算法分析的主程序逻辑如下，需要一个分析栈，一个LR分析表。

先读入一个字符，查找LR分析表，若表项为空，则返回报错信息；若表项为OK，说明分析完成，程序结束；若读到一个符号及其状态码，则将符号和状态码一起压栈，继续读入字符；若表项为一个产生式，则进行规约，将栈中属于产生式右部的元素弹出，再查LR表产生式左部对应的状态码，与左部一起压栈，对之前读到的字符重新查LR表；

重复上述过程，直到报错或是返回OK。




### 负数处理

在LR中负数的处理，与LL1类似，因为负数对应终结符`I`，所以一定不会是规约项，我们都按移进处理，关键是要移进多少个字符。对负数的形式分为三种情况讨论：符号后接变量，负号后接常量，负号后接（...）。前两种情况比较简单，分析器只需简单地移进两个符号，即把形如`-a`，`-100`的符号视为一个终结符`I`，而负号后接（...）则需要检查括号构成的子表达式是否符合语法，此处将括号形成的子表达式切片出来，使用递归检查即可。

对于`--`,`++`符号，我实现的语法规定`--`,`++`符号只可以接变量，这使得`--`,`++`符号的处理简单许多，只需在移进`--`,`++`符号时向后多看一位，若是变量则直接移进，否则返回错误信息。

# 具体实现

由于具体实现的代码过于冗长（硬编码代码占较大篇幅），其实现的逻辑与设计思路一致，所以具体代码请看压缩包文件中的parser.h代码实现。

下面仅简要讲述实验中所设计的类、数据结构与接口。

## LL(1)实现

LL1实现中所用到一些复杂数据结构：

StringMap：将token也就是终结符与可转换的产生式建立映射。

LL1_table：构建非终结符的LL1分析表。

symbol_set：记录$w_0, w_1$包含的符号。

```
typedef unordered_map<string, string> StringMap;
typedef unordered_map<char, StringMap> LL1_table;
typedef unordered_map<char, unordered_set<string>> symbol_set;
```

LL1类的实现如下：

```
class LL1{
    private:
        LL1_table analysis_table;   //LL1分析表
        // stack<char> analysis_stack; //作为check函数的局部变量更合适，便于递归。
        symbol_set Vt; //记录记录$w_0,w_1$包含的符号。

    public:
        LL1(){
            bulid_analysis_table();
        }
        void bulid_analysis_table();    //完成初始化，LL1分析表硬编码的填表工作
        bool check(vector<string> token_arr);   //接受token按token_seq顺序构成的数组，按照LL1算法逻辑检查语法正确性
};
```

## LR(0)实现

LR_StringMap: 为了分辨是否为规约项，加入了bool项，false为移进，true为规约。

LR_table: 记录LR分析表，构建state与符号对应项目的映射。

```
typedef unordered_map<string, pair<bool, string> > LR_StringMap;
typedef unordered_map<string, LR_StringMap> LR_table;
```

LR类实现如下：

```
class LR{
    private:
        //实现为check局部变量，便于递归
        // stack<char> symbol_stack;
        // stack<string> state_stack;
        LR_table analysis_table; //LR分析表


    public:
        LR(){ build_analysis_table(); }
        void build_analysis_table();    //LR分析表填表函数
        bool check(vector<string> token_arr);   //LR语法分析主函数
};
```

# 实验结果

下面只展示部分测试样例, 更多可以通过修改test.cpp文件尝试结果, 结果符合分析逻辑与语法规则.

## LL(1)结果

## 错误处理

对于语法中的错误, LL1可以大致定位到出错的位置,但由于LL1是自顶向下的语法分析算法,是由字符驱动的,所以某些只有遇到字符才会入栈的状态无法在匹配字符的过程中被检查出来,例如缺失左括号,这种情况下栈中不会压入左括号,因此会在匹配终结符的时候得到右括号而返回终结符不匹配的错误.

但大部分的错误都能正确报告. 下面展示部分例子:

该表达式中 缺失一个右括号 LL1分析器可以正确发现错误.

```
PS D:\学\编译原理\lab2> .\source.exe
test.cpp
n -= -a + -(1 + 1 - (1 + 1);
<00><47><46><00><43><46><53><03><43><03><46><53><03><43><03><54><58>
Missing ')' !
invaild sub-expression
```

该表达式中 缺失一个`;` LL1分析器可以正确发现错误. (C语言表达式以分号或者逗号作为终止符)

```
PS D:\学\编译原理\lab2> .\source.exe
test.cpp
n -= -a + 1
<00><47><46><00><43><03>
Missing terminator!
```

## 正确结果

以下是分析正规表达式时, 程序运行的结果, 可见LL1语法分析器可以处理负数以及自增运算.

```
PS D:\学\编译原理\lab2> .\source.exe
test.cpp
n -= -a + 1 + -(-1 % 1) ;
a = a << b, d = c > d;
a = --a + ++b + c++ - d--;
Expression LL1 correct!
PS D:\学\编译原理\lab2>
```

# LR(0)结果

## 错误处理

LR语法分析可以比较精确的定位到语法错误发生的地方,因为LR是自底向上的分析法, 分析表中有每种符号对于的分析行为,若不匹配则可以通过LR分析表得到什么状态在遇到什么字符的情况下, 分析得到语法错误.

此处将最后一个`-`视为符号所以报告负数不合法的错误.

```
PS D:\学\编译原理\lab2> .\source.exe
test.cpp
n -= -a + 1 + - ;
<00><47><46><00><43><03><43><46><58>
invaild negative number!
```

此处的表达式中,最后的括号里加号的右边缺失一个运算数, 所以返回运算数缺失或是左括号缺少的错误. 递归返回子表达式错误的报告.

```
PS D:\学\编译原理\lab2> .\source.exe
test.cpp
n -= -a + 1 + -(-1 + ) ;
<00><47><46><00><43><03><43><46><53><46><03><43><54><58>
Error!
We got #0(6E01B3 in stack!
We can accept token: F T 48 45 46 53 03 00
To regress (6 I2 T8 F5
But we got 54 now.
Error may be:
        Missing operand or '('!
invaild sub-expression
```

### 正确结果

可以看到, 对于正确的表达式, LR分析器可以正确分析并处理负数以及嵌套的子表达式.

```
PS D:\学\编译原理\lab2> .\source.exe
test.cpp
n -= -a + 1 + -(-1 % 1) ;
a = a+ b, c * d;
a = --a + ++b + c++ - d--;
Expression correct!
```

# 实验心得

通过本次实验, 我对LL1语法分析算法以及LR分析算法有了更深刻的理解, 如何构造LL1分析表、LR分析表，如何解决LR算法对于DFA产生的冲突都有了实践经验。本次实验中对负数以及自增自减操作在算术表达式中的处理给我带来了一定的挑战，如何使其适配到预设的文法之中，如何正确识别负号与减号，都需要思考与设计，但整体上实现起来只需要根据数学的符号规则对原有的框架加入负号的识别匹配，也能够解决该问题。总体上讲，本次实验有一定难度，但使我对理论知识的认识更加深刻，也为以后的下游语义分析器提供了良好的上游代码基础。