实验二：语法分析器编程　Lab2: Syntax Parser Programming

一、实验目的 Motivation/Aim

本次语法分析器的编程实验目的在于：

1) 了解语法分析是编译过程的核心部分，它的主要任务是按照程序的语法规则，从由语法分析输出的源程序符号串中识别出各类语法成分，同时进行词法检查，为语义分析和代码生成做准备；
2) 了解递归下降分析方法、自顶向下的 LL(1)方法、自下而上的 LR(k)方法进行语法分析的方法；
3) 掌握语法分析程序设计的原理和构造方法。

二、实验内容 Content description

实验二要求如下：

a. 输入：字符流，上下文无关文法 CFG（CFGs 文法句子组合自行确定）
b. 输出：语法树
　如果使用自顶向下的语法分析方法，则为推导序列。
　如果使用自下而上的语法分析方法，则为规约序列。
a. 文法句号自行定义
b. 可能包括错误处理

确定实验内容如下：

例：针对 C++语言中简单算术表达式文法 G[E]：

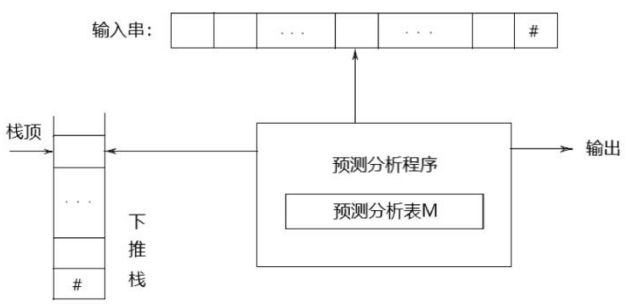
1. E→EAT | T
2. T→TMF | F
3. F→(E) | i
4. A→+ | -
5. M→* | /

求解相应的 FIRST、FOLLOW 集，构造预测分析表，并编写语法分析程序，并给出测试句子的分析过程。

三、实验方法 Ideas/Methods

本次实验采用自顶向下的 LL(1)方法：

1) 手工判断上述文法 G[E]是否 LL(1)文法，若不是，将其转变为 LL(1)文法；
2) 对转变后的 LL(1)文法手工建立 LL(1)分析表；
3) 根据编译原理教材上算法思想，构造 LL(1)分析程序；



规定 X 为下推栈栈顶字符，a 为当前读入符号；

若 X=a=“#”，则分析成功，停止分析；

若 X=a≠“#”，则把 X 从栈顶弹出，让 a 指向下一个读入符号；

若 $X \in V_N$，则查预测分析表 M，若 M[X][a]中存放着关于 X 的产生式，则弹出 X，并且将此产生式的右部以自右向左的顺序压入栈内，在输出带上记下产生式编号；若 M[X,a]中存放着“出错标志”，则调用相应出错程序 ERROR 去处理。

4) 用 LL(1)分析程序对任意键盘输入串进行语法分析，并根据栈的变化状态输出给定串的具体分析过程。

四、实验假设 Assumptions

由于目前不存在算法，能够在有限步数内确切判断一个文法是否为二义文法，我们假设实验输入文法已消除二义性。

接着，假设实验输入的文法是 LL(1)文法，因为我们为降低实验难度，可以先手动消除左递归与提取左公共因子，将无二义性文法转变为 LL(1)文法。

例如，我们自定的实验内容中给出的文法 G[E]是存在左递归的，我们可以手动消除左递归，如下：

1. E→TE’
2. E’→ATE’ | ε
3. T→FT’
4. T’→MFT’ | ε
5. F→(E) | i
6. A→+ | -
7. M→* | /

最后，假设实验输入的文法的 FIRST、FOLLOW 集也是已知的，并能够根据此建立 LL(1)分析表，同样是因为降低实验难度，可以先手动求解相应的 FIRST、FOLLOW 集，构造预测分析表，然后在键盘输入辅助语法分析。

从而，本实验内容 G[E]的 FIRST、FOLLOW 集手动计算如下：

	FIRST	FOLLOW
E→TE’	{(, i}	{# ,)}
E’→ATE’	{+ , -}	{# ,)}
E’→ ε	{ ε }	
T→FT’	{(, i}	{# ,) , + , -}
T’→MFT’	{* , /}	{# ,) , + , -}
T’→ ε	{ ε }	
F→(E)	{(}	{# ,) , + , - , * , /}
F→i	{i}	
A→+	{+}	{(, i}
A→-	{-}	
M→*	{*}	{(, i}
M→/	{/}	

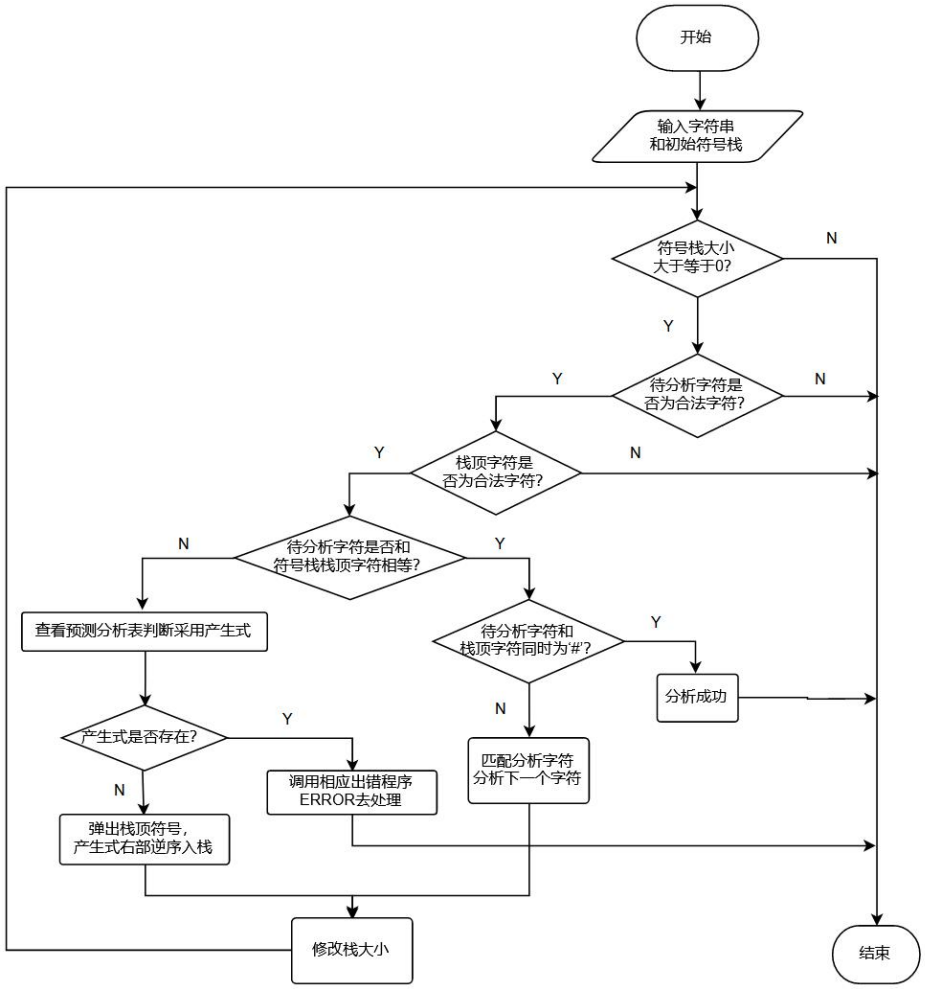
预测分析表如下：

	(	)	i	+	-	*	/	#
E	E→TE’		E→TE’					

E’		E’→ ε		E’→ATE’	E’→ATE’			E’→ ε
T	T→FT’		T→FT’					
T’		T’→ ε		T’→ ε	T’→ ε	T’→MFT’	T’→MFT’	T’→ ε
F	F→(E)		F→i					
A				A→+	A→-			
M						M→*	M→/	

五、相关流程图描述 Related flow chart descriptions

将构造 LL(1)分析程序的流程绘制如下：



六、核心数据结构描述 Description of important Data Structures

1) 根据输入的终结符、非终结符个数，动态构造 LL(1)预测分析表的大小

```
int terminalNum; // 终结符的个数
int nonTerminalNum; // 非终结符的个数
string** LL1ForecastAnalysisTable; // LL(1)预测分析表
LL1ForecastAnalysisTable = new string * [nonTerminalNum + 1];
for (int i = 0; i <= nonTerminalNum; i++)
    LL1ForecastAnalysisTable[i] = new string[terminalNum + 2];
```

2) 终结符(包含结束符#)的集合、非终结符的集合，全部用 map 映射表示在分析表的第几列/第几列，这样便于查找，提高效率。

```
map<string, int> terminalSet; // 终结符(包含结束符)的集合，map 映射表示在分析表的第几列
map<string, int> nonTerminalSet;// 非终结符的集合，map 映射表示在分析表的第几行
```

3) 下推栈采用 vector 数据结构表示

```
vector<string>pushdownStack; // 下推栈
```

4) 语法树采用树的数据结构表示

```
vector<vector<void*>>treeStack;//辅助构建语法树的栈
struct node
{
    string name;//节点名称
    int cnt;//子树宽度
    vector<node*>child;
};
typedef node* TreeNode;
struct Tree
{
    TreeNode root;
};
Tree syntaxTree;//语法树
```

七、核心算法描述 Description of core Algorithms

1) `void syntax_parser(string str)`

语法分析器，从 str 中按字符读入，与下推栈栈顶字符比较，逐字分析；

设 X 为下推栈栈顶字符，a 为当前读入符号，ptr 为输入条的指针；

```
string X, a(1, str[ptr]);// X 代表栈顶字符,a 为当前读入符号
```

若 X=a=“#”，则分析成功，停止分析；

若 X=a≠“#”，则把 X 从栈顶弹出，让 a 指向下一个读入符号；

若 $X \in V_N$，则查预测分析表 M，若 M[X][a]中存放着关于 X 的产生式，则弹出 X，并且将此产生式的右部以自右向左的顺序压入栈内，在输出带上记下产生式编号；若 M[X,a]中存放着“出错标志”，则调用相应出错程序 ERROR 去处理。

```
void syntax_parser(string str)
{
    cout << "步骤\t\t 栈内\t\t 输入串\t\t 所用产生式\t\t 动作" << endl;
    cout << "初态\t\t#" << LL1ForecastAnalysisTable[1][0] << "\t\t" << str << "#\t\t\n";
    str += "#";
    int i = 0, ptr = 0;// i 代表进行到步骤 i,ptr 代表分析到输入语言字符串第几个
    string X, a(1, str[ptr]);// X 代表栈顶字符,a 为当前读入符号
    string inStack, inputStr;// inStack 代表栈内，inputStr 代表输入串
    pushdownStack.clear();// 使下推栈为空
    pushdownStack.push_back("#");// 结束符#入栈
    pushdownStack.push_back(LL1ForecastAnalysisTable[1][0]);// 文法开始符号入栈
    syntaxTree.root = new node;//语法树根节点
```

```
auto cur_node = syntaxTree.root;
cur_node->name = LL1ForecastAnalysisTable[1][0];
treeStack.push_back(vector<void*>());//进入语法树的下一层
treeStack.back().push_back(0);//语法栈中两个局部变量
treeStack.back().push_back(cur_node);
while (++i){
    X = pushdownStack.back();// 下推栈栈顶符号
    if (X == a)
    {
        if (a == "#") {   // X=a='#',表示识别成功
            cout << "\t\t 识别成功" << endl; return;
        }
        else {      // X=a≠'#',表示匹配
            ptr++;//读头前进一格
            pushdownStack.pop_back();//弹出栈顶符号 X
            inStack = "";
            for (int i = 0; i < pushdownStack.size(); i++)
                inStack += pushdownStack[i];
            inputStr = "";
            for (int i = ptr; i < str.size(); i++)
                inputStr += str[i];
            cout << i << "\t\t" << inStack << "\t";
            if (pushdownStack.size() < 5)cout << "\t";
            cout << inputStr << "\t\t\t\t\t" << a << "匹配" << endl;
            a = str[ptr];
            // 语法树栈调整(到树叶节点时)，索引右移动
            while (treeStack.size() > 1){
                treeStack.back()[0] =       (void*)((int)treeStack.back()[0] + 1);
                int index = (int)treeStack.back()[0];
                int parent_idx = (int)treeStack[treeStack.size() - 2][0];
                TreeNode Node =
                        (TreeNode)treeStack[treeStack.size() - 2][parent_idx + 1];
                if (Node->child.size() == index){
                    treeStack.pop_back();
                    continue;
                }
                break;
            }
        }
    }
    else {
        if (terminalSet[X]) { //X∈终结符，但 X≠a，则调 ERROR 处理
            cout << i << "\t\t 出错\n"; return;
        }
```

```
else { //X∈非终结符，查预测分析表 M
        int row = nonTerminalSet[X], col = terminalSet[a];
        string production = LL1ForecastAnalysisTable[row][col]; // M[X,a]
        if (production == "_"){      //存放着出错标志
                cout << i << "\t\t 出错\n";
                return;
        }
        else {       // 存放着关于 X 的产生式
                pushdownStack.pop_back();      // 弹出栈顶符号 X
                if (production != "epsilon")// 将产生式右部以自右向左的顺序入栈
                        for (int i = production.size() - 1; i >= 0; i--)
                                if(production[i]=='\''){
                                   i--;
                                   pushdownStack.push_back(string(1,production[i])+"'");
                                }
                                else
                                        pushdownStack.push_back(string(1, production[i]));
                inStack = "";
                for (int i = 0; i < pushdownStack.size(); i++)
                        inStack += pushdownStack[i];
                inputStr = "";
                for (int i = ptr; i < str.size(); i++)
                        inputStr += str[i];
                cout << i << "\t\t" << inStack << "\t";
                if (pushdownStack.size() < 5) cout << "\t";
                cout << inputStr << "\t\t" << X << "->" << production << "\t\t";
                if (production.size() < 4) cout << "\t"; cout << "推导";
                if (production == "epsilon") cout << "空串";
                cout << endl;
                int index = (int)treeStack.back()[0];
                TreeNode Node = (TreeNode)treeStack.back()[index + 1];
                cout << Node->name << endl;
                treeStack.push_back(vector<void*>());//进入语法树下一层
                treeStack.back().push_back(0);
                if(production != "epsilon"){//产生式右部
                        for (int i = 0; i < production.size(); i++){
                                TreeNode newNode = new node;
                                if (i + 1 < production.size() && production[i + 1] == '\'') {
                                        newNode->name = string(1, production[i]) + "'";
                                        i++;
                                }
                                else
                                        newNode->name = string(1, production[i]);
                                Node->child.push_back(newNode);
```

```
                                    treeStack.back().push_back(newNode);
                                }
                            }
                            else {
                                TreeNode newNode = new node;
                                newNode->name = "epsilon";
                                Node->child.push_back(newNode);
                                treeStack.back().push_back(newNode);
                            }
                            if (production == "epsilon"){//推导出空串时也需调整语法树
                                while (treeStack.size() > 1){
                                    treeStack.back()[0] = (void*)((int)treeStack.back()[0] + 1);
                                    int index = (int)treeStack.back()[0];
                                    int parent_idx = (int)treeStack[treeStack.size() - 2][0];
                                    TreeNode Node =
                                    (TreeNode)treeStack[treeStack.size() - 2][parent_idx + 1];
                                    if (Node->child.size() == index){
                                        treeStack.pop_back();
                                        continue;
                                    }
                                    break;
                                }
                            }
                        }
                    }
                }
            }
            cout << "输出语法树如下：" << endl; printTree(syntaxTree.root,0);
        }
```

2) `void printTree(TreeNode Node,int deep)`

打印语法树（竖着打印），通过递归思维打印语法树的父子节点，deep 表示节点在树中位于的层数，先打印父节点，再打印以其子节点为父节点的子树。

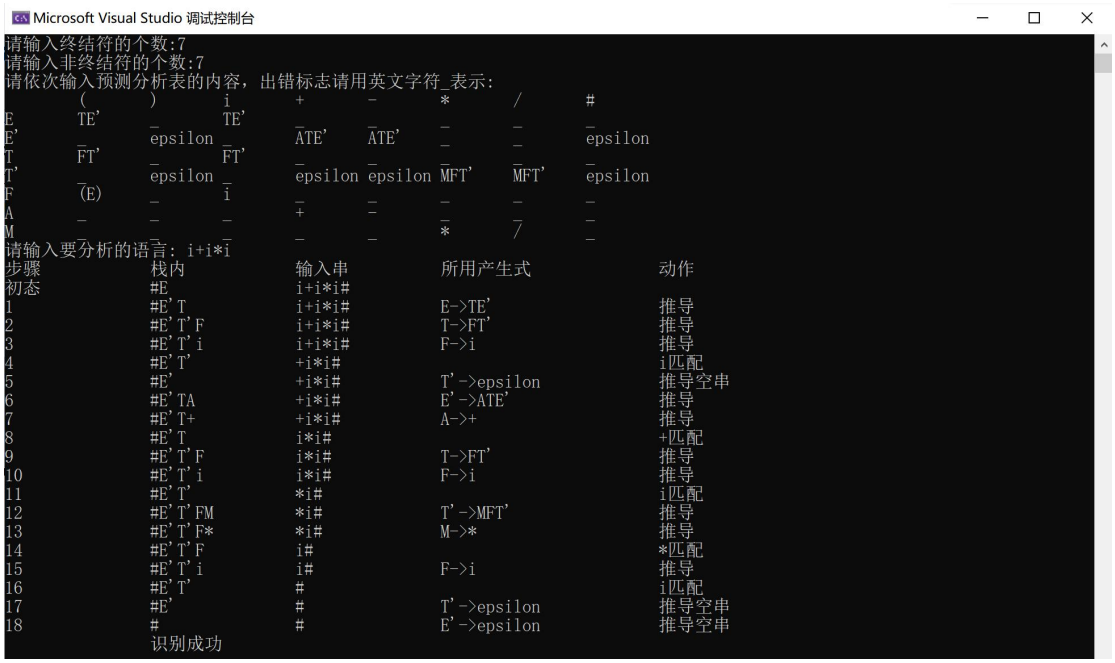
```
void printTree(TreeNode Node,int deep)
{
    for (int i = 0; i <= deep; i++)
        cout << "\t";
    cout << Node->name << endl;
    for (int i = 0; i < Node->child.size(); i++)
        printTree(Node->child[i], deep + 1);
}
```

八、测试用例 Use cases on running

例如 C++语言中简单算术表达式文法 G[E]（已手动转变为 LL(1)文法）：

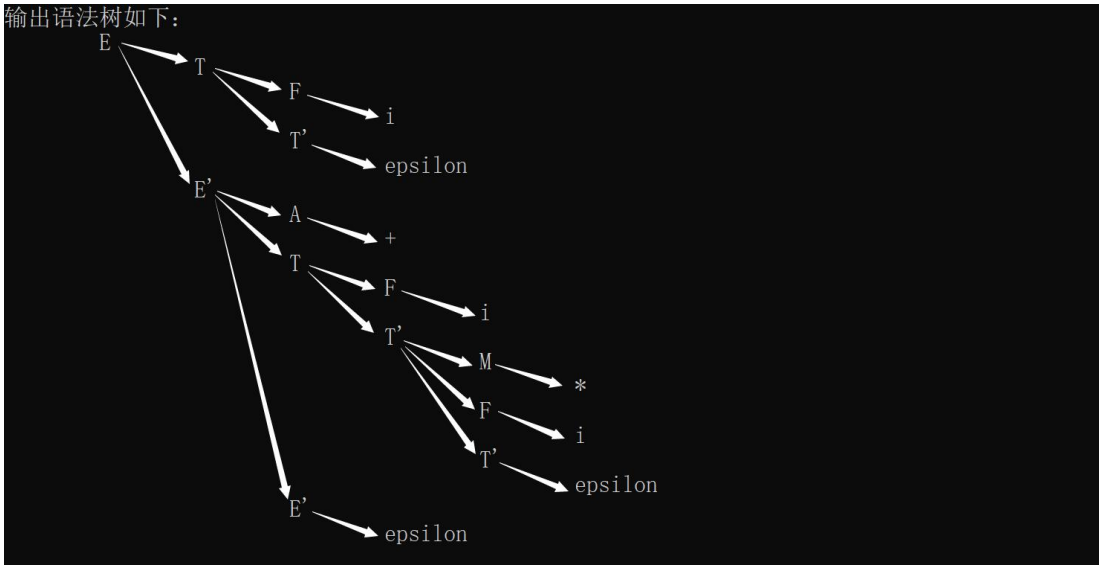
8. E→TE’
9. E’→ATE’ | ε
10. T→FT’
11. T’→MFT’ | ε
12. F→(E) | i
13. A→+ | -
14. M→* | /

测试：i+i*i，输出打印结果如下：

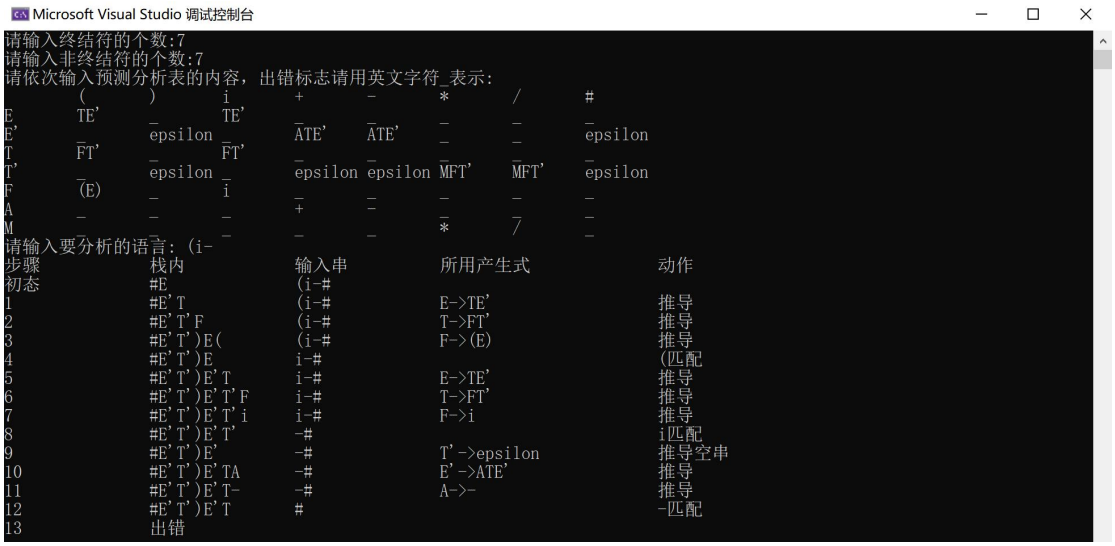


可见 i+i*i 识别成功，并且自上而下将推导过程打印出来了；

再输出语法树如下：（打印的同一列元素属于语法树同一层，推导的产生式位于斜下方）



测试：(i-，输出打印结果如下：



可见 (i- 这样的语法错误也检测出来了。

九、出现的问题与解决方案 Problems occurred and related solutions

出现的问题有：

1. 在将输入文法转换为 LL(1)文法时，经常会用到消除左递归的方法：
   对于产生式 P→Pa | b 可改写为非直接左递归的等价式：P→bP’ P’→aP’ | ε；
   但也因此在程序中输入时会被默认为两个分开的字符不是一个整体，同时将产生式反序入栈时也会出现问题；
2. 语法树打印；

解决方案为：

1. 将非终结符从 char 型转变为 string 型，从而 map 数据结构的声明也从 map<char,int>修改为 map<string,int>,vector 下推栈的声明也从 vector<char>修改为 vector<string>，产生式反序入栈时，也添加一个判断是否为 ’ 的判断，从而规避问题。
2. 采用 struct 构建树的数据结构以及利用一个栈来辅助构建语法树，具体解决方案见报告的第六、七部分有关数据结构与算法代码呈现。

十、心得体会 Your feelings and comments

通过动手实践，使我对构造语法分析器的基本原理有更为深入的理解和掌握，将语法分析过程的理论运用到实际，更加熟练地计算 FIRST、FOLLOW 集与构造预测分析表。其中，最大的锻炼是通过设计算法改变下推栈的大小，明确了构造 LL(1)分析程序的过程。

不足的是，算法采用的是基于已经手写 FIRST、FOLLOW 集构建好预测分析表的词法分析方法，人工手写可能并不高效，对于自动分析文法 FIRST、FOLLOW 集以及自动构建预测分析表的算法并没有尝试。以及，语法树的打印不够美观，但是仔细查看也是正确的。不过总的来说，这次实验还是达到了实验初衷，实现了一定的语法分析功能，得到了锻炼。