北京邮电大学计算机学院（国家示范性软件学院）本科2019级

编译原理与技术

语法分析实验报告



姓名： 江姝潼

班级: 2019211314

学号: 2019211653

专业： 网络工程

目录

[一、实验概述 3](#_Toc87114108)

[1、实验题目与要求 3](#_Toc87114109)

[2、开发环境 3](#_Toc87114110)

[二、结构设计 3](#_Toc87114111)

[1、实验准备 3](#_Toc87114112)

[2、文法类 4](#_Toc87114113)

[3、预测分析表类 5](#_Toc87114114)

[三、实现方案 6](#_Toc87114115)

[1、生成FIRST集 6](#_Toc87114116)

[2、生成FOLLOW集 8](#_Toc87114117)

[3、预测分析表的生成 10](#_Toc87114118)

[4、预测分析过程 12](#_Toc87114119)

[四、实验运行结果及分析说明 14](#_Toc87114120)

[输入输出： 14](#_Toc87114121)

[测试用例一： 16](#_Toc87114122)

[测试用例二： 17](#_Toc87114123)

[测试用例三： 19](#_Toc87114124)

[测试用例四： 20](#_Toc87114125)

[测试用例五： 22](#_Toc87114126)

[五、心得体会 24](#_Toc87114127)

[六、附录：源代码 25](#_Toc87114128)

# 一、实验概述

## 1、实验题目与要求

题目：语法分析程序的设计与实现。

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：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、开发环境

Windows 10 设备机，Visual Stdio Community 2019编程环境

# 二、结构设计

## 1、实验准备

本次实验需要对LL(1)文法进行分析预测处理，而题目所给文法是含左递归和回溯的，故需要人为改写文法消除。为了方便计算机对文法的处理，所有的终结符和非终结符均用单个字符表示，改写后的文法如下：

E->TA

A->+TA

A->-TA

A->#

T->FB

B->\*FB

B->/FB

B->#

F->(E)

F->n

其中为了方便计算机处理，用“#”表示空字符。

本次实验的编写采用了面向对象的方式，原因是从宏观的角度考虑，按照功能划分程序为文法和分析表两大类，而每类的操作又是相对独立的，互不干扰；同时，由于文法和分析表都要对多方的数据进行处理，若面向过程编程需要传大量参数，若采用类的方式处理则可以直接调用类内的变量，操作方便简洁。

故本实验定义了“文法”和“分析表”两大类整体结构，对各个子模块分别逐步细化，最终分解到具体语句为止。把各类的数据和操作封装在一起，并避免了局部变量的暴露，而只提供接口；同时，本次实验还使用了继承的方式，分析表是从文法类中继承而来，可以复用文法的操作，方便了程序的设计。

## 2、文法类

1. 首先定义产生式类，左侧为一个字符，右侧为产生式，用string表示；

class EXP//产生式

{

public:

    char left = '\0';//左部

    string right = "";//右部

};

2.接下来定义文法：

其中后缀为cnt的表示计数变量，分别用来存储非终结符、终结符和产生式的个数，N\_begin表示起始符；后缀为sym的用来存储终结和非终结符号；两个前缀为whe\_find的数组用于记录生成FIRST和FOLLOW集的时候，该符号是否已经生成过，避免重复递归。

class grammar

{

public:

    int N\_cnt;//非终结符个数

    int T\_cnt;//终结符个数

    int G\_cnt;//产生式个数

    EXP exp[MAXN];//产生式

    set<char> FIRST[MAXN];//FIRST集

    set<char> FOLLOW[MAXN];//FOLLOW集

    char N\_begin;            //起始符

    char N\_sym[MAXN] = { 0 };//非终结符

    char T\_sym[MAXN] = { 0 };//终结符

    bool whe\_find\_first[MAXN] = { false };  //表示该符号是否已经找过其FIRST集

    bool whe\_find\_follow[MAXN] = { false };  //表示该符号是否已经找过其FOLLOW集

    grammar();

    ~grammar();

    void input();   //输入文法

    bool isN(char ch);//判断一个字符是否为非终结符

    int getT(char ch);//获取终结符下标

    int getN(char ch);//获取非终结符下标

    void getFirst(char ch);//获取FIRST集

    void getFollow(char ch);//获取FOLLOW集

    void printSet();//打印集合

};

各函数所代表的含义如下：

|  |  |  |
| --- | --- | --- |
| **函数名** | **参数** | **功能** |
| void input(); | 无 | 输入文法 |
| bool isN(char ch); | 字符ch | 判断一个字符是否为非终结符，是返回true,否则返回false |
| int getT(char ch); | 字符ch | 获取终结符下标 |
| int getN(char ch); | 字符ch | 获取非终结符下标 |
| void getFirst(char ch); | 字符ch | 获取FIRST集 |
| void getFollow(char ch); | 字符ch | 获取FOLLOW集 |
| void printSet(); | 无 | 打印文法 |

## 3、预测分析表类

预测分析表的类定义如下。

其中detect数组表示分析表在某个位置的产生式个数，如果>1则不是LL（1）文法，用analyze\_t来存储预测分析表；定义了两个栈stack和remainder分别代表分析栈和输入缓冲区。

class analyze\_table:public grammar {

public:

    string str = "";//待分析的字符串

    int detect[MAXN][MAXN] = {};//计数该位置产生式的个数，用于判断是否是LL(1)文法

    int analyze\_t[MAXN][MAXN];//分析预测表

    vector<char> stack;       //分析栈

    vector<char>  remainder;  //输入缓冲区

    analyze\_table();

    ~analyze\_table();

    void getAT();  //生成预测分析表

    void printAT();//输出预测分析表

    bool judgeL1();//判断是否为LL(1)文法

    void inputS(); //输入字符串

    void analyze();//分析预测过程

};

各函数所代表的含义如下：

|  |  |  |
| --- | --- | --- |
| **函数名** | **参数** | **功能** |
| void getAT(); | 无 | 生成预测分析表 |
| void printAT(); | 无 | 输出预测分析表 |
| bool judgeL1(); | 无 | 判断该文法是否为LL(1)文法，是返回true,否则返回false |
| void inputS(); | 无 | 输入要分析的字符串 |
| void analyze(); | 无 | 分析预测过程 |

# 三、实现方案

## 1、生成FIRST集

getFirst(N)表示对当前非终结符N求FIRST集。用whe\_fine\_first[ch]存储之前是否已经找过该符号的FIRST集。

具体方法为，遍历所有的产生式，如果产生式左侧为要找的非终结符N，则遍历产生式的右侧，如果右侧字符为终结符，则把该终结符加入FIRST集中，并停止向后遍历；如果为非终结符，则求该非终结符FIRST集，并把FIRST中非空元素加进去，如果FIRST集中含有空，则继续向后遍历，反之则停止遍历，若能一直遍历到末尾，则说明产生式右侧可以完全转化为空，把空元素也加到FIRST集中,程序代码如下：

void grammar::getFirst(char ch) {

    //如果之前已经找过该符号的FIRST集，则退出

    if (whe\_find\_first[ch])

        return;

    bool nulls\_flag = false;   //用于存储First集是否有空串$

    int total = 0;             //记录非终结符产生$的个数

    int loc\_ch = getN(ch);

    for (int i = 0; i < G\_cnt; i++) {

        //如果当前产生式左部是需要找FIRST集的非终结符

        if (ch == exp[i].left) {

            for (int j = 0; j < (int)exp[i].right.length(); j++) {

                //如果当前字符是终结符

                if (!isN(exp[i].right[j])) {

                    FIRST[loc\_ch].insert(exp[i].right[j]);

                    break;

                }

                //如果是非终结符，则求它的FIRST集

                getFirst(exp[i].right[j]);

                //把该非终结符的FIRST集加到当前非终结符的FIRST集中

                set<char>::iterator it;

                int locj = getN(exp[i].right[j]);

                for (it = FIRST[locj].begin(); it != FIRST[locj].end(); it++) {

                    if (\*it == '#') {

                        nulls\_flag = true;

                    }

                    else {

                        FIRST[loc\_ch].insert(\*it);

                    }

                }

                //如果没有空串则不必再找下一个符号了

                if (!nulls\_flag)

                    break;

                else {

                    nulls\_flag = false;

                    total++;

                }

            }

            //如果右部所有符号的First集都有空串,则ch的First集也有空串

            if (total == exp[i].right.length())

                FIRST[loc\_ch].insert('#');

        }

    }

    whe\_find\_first[ch] = true;

}

## 2、生成FOLLOW集

getFollow(N)表示对当前非终结符N求FOLLOW集。用whe\_fine\_follow[ch]存储之前是否已经找过该符号的FOLLOW集。同FIRST集不同的是，**需要将$先放入起始符的FOLLOW集中**。

具体方法为，遍历所有的产生式，如果在产生式右侧找到要找的非终结符N，则遍历N位置后的所有符号。如果N不在最后一个位置，后面是终结符的话，则把该该终结符加入FOLLOW集中，并停止向后遍历，如果是非终结符，则把非终结符的除空的FRIST集加入FOLLOW中，若FIRST集中不含空，则停止向后遍历；若能一直遍历到末尾（表示N之后的部分可以完全转化为空）或N在最后一个位置，则把左部的FOLLOW集加入到当前符号N的FOLLOW集中,程序代码如下：

void grammar::getFollow(char ch) {

    //如果之前已经找过该符号的FOLLOW集，则退出

    if (whe\_find\_follow[ch])

        return;

    int loc\_ch = getN(ch);

    for (int i = 0; i < G\_cnt; i++) {

        int index = -1;

        int rlen = exp[i].right.length();

        //找当前字符在产生式中的位置

        for (int j = 0; j < rlen; j++) {

            if (ch == exp[i].right[j]) {

                index = j;

                break;

            }

        }

        //如果当前产生式找到了ch，且ch不在最后一个位置

        if (index != -1 && index < rlen - 1) {

            int total = 0;

            bool nulls\_flag = false;

            char current\_ch;

            for (int j = index + 1; j < exp[i].right.length(); j++) {

                current\_ch = exp[i].right[j];

                //如果是终结符，则加入到FOLLOW集中

                if (!isN(current\_ch)) {

                    FOLLOW[loc\_ch].insert(current\_ch);

                    break;

                }

                //如果是非终结符，则把非终结符的除空的FRIST集加入FOLLOW中

                else {

                    set<char>::iterator it;

                    for (it = FIRST[getN(current\_ch)].begin(); it != FIRST[getN(current\_ch)].end(); it++)

                    {

                        if (\*it == '#')

                            nulls\_flag = true;

                        else

                            FOLLOW[loc\_ch].insert(\*it);

                    }

                    //如果该符号不会变成空，则不需要再遍历后面的符号

                    if (!nulls\_flag)

                        break;

                    else {

                        nulls\_flag = false;

                        total++;

                    }

                }

            }

            //如果后面可以生成空串，则加入产生式左边的符号的FOLLOW集

            if (total == (exp[i].right.length() - index - 1)) {

                char temp = exp[i].left;

                getFollow(temp);

                set<char>::iterator it;

                for (it = FOLLOW[getN(temp)].begin(); it != FOLLOW[getN(temp)].end(); it++)

                    FOLLOW[loc\_ch].insert(\*it);

            }

        }

        //如果当前产生式ch在最后一个位置，则加左部的FOLLOW集,注意要避免循环递归的情况

        else if (index == rlen - 1 && ch != exp[i].left) {

            char temp = exp[i].left;

            getFollow(temp);

            set<char>::iterator it;

            for (it = FOLLOW[getN(temp)].begin(); it != FOLLOW[getN(temp)].end(); it++)

                FOLLOW[loc\_ch].insert(\*it);

        }

    }

    whe\_find\_follow[ch] = true;

}

## 3、预测分析表的生成

构造文法G的预测分析表了。其主要思想如下:

遍历文法的每一个产生式，对于左侧为A的非终结符，从头开始遍历产生式右侧的每一个符号。如果该符号是终结符a，则加入到分析表M[A,a]所对应位置，并停止向后遍历；如果是非终结符,则遍历其FIRST集，把FIRST集中所有的非空元素a放入M[A,a]中，如果FIRST集中不含空，则停止向后遍历，反之继续向后遍历，如果能够遍历到末尾，说明产生式右侧可以全部为空，则遍历左部的FOLLOW集，对集合中每个元素b放入M[A,b]中，具体代码如下：

void analyze\_table::getAT() {

    for (int i = 0; i < G\_cnt; i++) {

        int nulls\_flag = false;

        int total = 0;

        char temp;

        for (int j = 0; j < exp[i].right.length(); j++) {

            temp = exp[i].right[j];

            //如果是终结符

            if (!isN(temp)) {

                //如果产生式右侧是空，则标记，后续会处理加入FOLLOW集

                if (temp == '#') {

                    nulls\_flag = true;

                    total++;

                }

                //该终结符不是空串，更新预测分析表

                else {

                    analyze\_t[getN(exp[i].left)][getT(temp)] = i;

                    detect[getN(exp[i].left)][getT(temp)] ++;

                    break;

                }

            }

            //如果是非终结符,则遍历其FIRST集

            else {

                set<char>::iterator it;

                for (it = FIRST[getN(temp)].begin(); it != FIRST[getN(temp)].end(); it++)

                {

                    //把FIRST集中所有的非空元素a放入M[A,a]中

                    if (\*it == '#') {

                        nulls\_flag = true;

                        total++;

                    }

                    else {

                        analyze\_t[getN(exp[i].left)][getT(\*it)] = i;

                        detect[getN(exp[i].left)][getT(\*it)] ++;

                    }

                }

            }

            if (!nulls\_flag)

                break;

            else

                nulls\_flag = true;

        }

        //若右边可以全部为空，则遍历左部的follow集

        if (total == exp[i].right.length()) {

            set<char>::iterator it;

            for (it = FOLLOW[getN(exp[i].left)].begin();it != FOLLOW[getN(exp[i].left)].end(); it++)

            {

                detect[getN(exp[i].left)][getT(\*it)]++;

                analyze\_t[getN(exp[i].left)][getT(\*it)] = i;

            }

        }

    }

}

## 4、预测分析过程

预测分析过程开始先，需要先初始化，将$压入栈底，将文法开始符号S压入栈；将输入字符串+$放入输入缓冲区中，然后，预测分析控制程序根据分析表M对输入符号串w作出自顶向下的分析，过程如下：

预测分析控制程序：根据栈顶符号X和当前输入符号a来决定分析程序应采取的动作，有4种可能：

（1）若X=a=$，则分析程序宣告分析成功，停止分析。

（2）若X=a≠$，则分析程序从栈顶弹出X，向前指针前移一个位置，指向a的后继符号。

（3）若X是终结符号，但X≠a，则分析程序发现了输入符号串中存在的一个错误，此时停止分析，并报错。

（4）若X是非终结符号，则预测分析控制程序访问分析表M[X，a]。

·若M[X，a]是产生式，则先将X从栈顶弹出，然后把产生式的右部符号串按反序一一推入栈中。

·若M[X，a]是产生式X→空（本实验用的是“#”），则预测分析控制程序从栈顶弹出X。

·若M[X，a]是error，则停止分析，并报错。

具体代码如下：

void analyze\_table::analyze() {

    cout << "栈\t\t\t" << "输入\t\t\t" << "输出" << endl;

    remainder.push\_back('$');

    for (int i = str.length() - 1; i >= 0; i--) {

        remainder.push\_back(str[i]);

    }

    stack.push\_back('$');

    stack.push\_back(N\_begin);

    while (!remainder.empty()) {

        string outS = "";

        for (int i = 0; i < stack.size(); i++)

        {

            outS += stack[i];

        }

        if (outS.length() >= 8)

            cout << outS << "\t\t";

        else

            cout << outS << "\t\t\t";//输出分析栈内容,$栈底到栈顶

        outS = "";

        for (int i = remainder.size() - 1; i >= 0; i--)

        {

            outS += remainder[i];

        }

        if (outS.length() >= 8)

            cout << outS << "\t\t";

        else

            cout << outS << "\t\t\t";

        char stack\_top = stack[stack.size() - 1];//表示当前栈顶元素

        char remainder\_top = remainder[remainder.size() - 1];//表示剩余输入的第一个字符

        if (stack\_top == remainder\_top && stack\_top == '$') {

            cout << "acc" << endl;

            return;

        }

        if (stack\_top == remainder\_top) {//remainder\_top恒为终结符

            stack.pop\_back();

            remainder.pop\_back();

            cout << "匹配" << endl;

        }

        else if (analyze\_t[getN(stack\_top)][getT(remainder\_top)] != -1) {

            int loc = analyze\_t[getN(stack\_top)][getT(remainder\_top)];

            stack.pop\_back();

            //如果产生式右侧不为空，则逆序将产生式右式压入栈中

            if (exp[loc].right != "#") {

                for (int i = exp[loc].right.length() - 1; i >= 0; i--) {

                    stack.push\_back(exp[loc].right[i]);

                }

            }

            cout << exp[loc].left << "->" << exp[loc].right << endl;

        }

        else {

            cout << "error!" << endl;

            return;

        }

    }

}

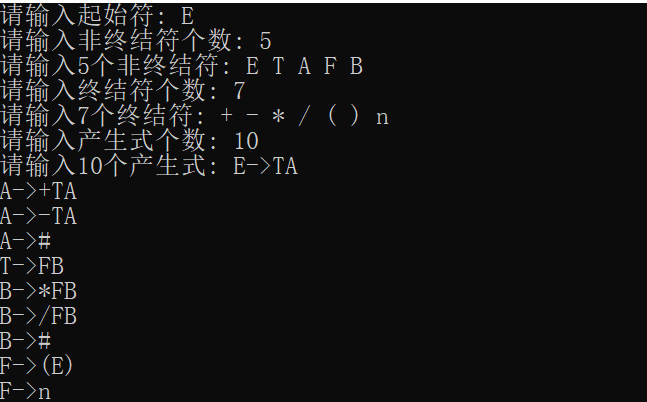
# 四、实验运行结果及分析说明

## 输入输出：

1．输入

输入主要分为两部分，第一部分是输入文法，第二部分输入待分析字符串。

第一部分输入根据提示一个一个输入即可，先输入非终结符再输入终结符，再依次输入各个产生式即可，具体输入见下图：

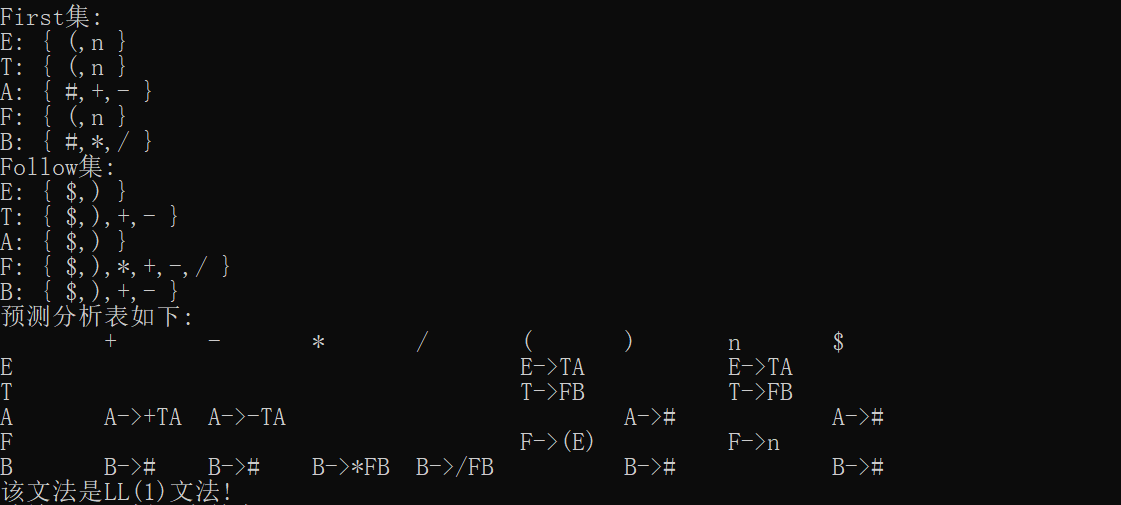


第二部分入出要分析的字符串：

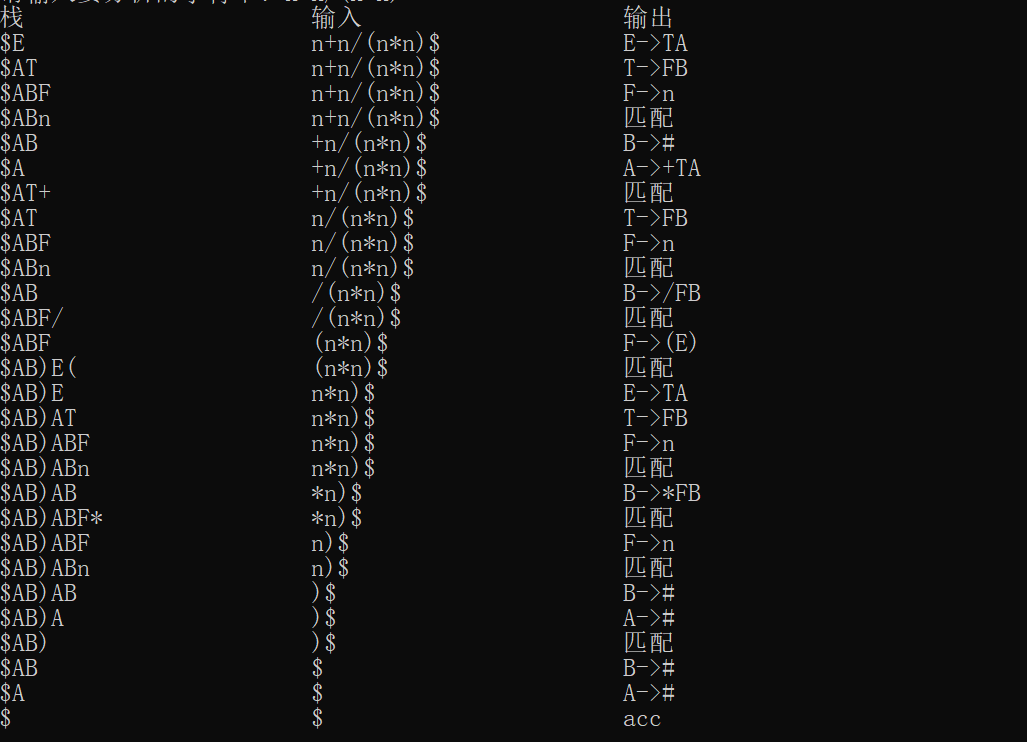


2．输出

输出也主要分为输出文法和分析表信息两部分，第一部分会根据输入的文法输出其FIRST和FOLLOW集，并输出对应的预测分析表；



第二部分则输出对应输入文法的预测分析过程，包括分析栈、输入字符和输出结果。



## 测试用例一：

1.正常输入：

*请输入起始符: E*

*请输入非终结符个数: 5*

*请输入5个非终结符: E T A F B*

*请输入终结符个数: 7*

*请输入7个终结符: + - \* / ( ) n*

*请输入产生式个数: 10*

*请输入10个产生式: E->TA*

*A->+TA*

*A->-TA*

*A->#*

*T->FB*

*B->\*FB*

*B->/FB*

*B->#*

*F->(E)*

*F->n*

*First集:*

*E: { (,n }*

*T: { (,n }*

*A: { #,+,- }*

*F: { (,n }*

*B: { #,\*,/ }*

*Follow集:*

*E: { $,) }*

*T: { $,),+,- }*

*A: { $,) }*

*F: { $,),\*,+,-,/ }*

*B: { $,),+,- }*

*预测分析表如下:*

*+ - \* / ( ) n $*

*E E->TA E->TA*

*T T->FB T->FB*

*A A->+TA A->-TA A-># A->#*

*F F->(E) F->n*

*B B-># B-># B->\*FB B->/FB B-># B->#*

*该文法是LL(1)文法!*

*请输入要分析的字符串: n+n/(n\*n)*

*栈 输入 输出*

*$E n+n/(n\*n)$ E->TA*

*$AT n+n/(n\*n)$ T->FB*

*$ABF n+n/(n\*n)$ F->n*

*$ABn n+n/(n\*n)$ 匹配*

*$AB +n/(n\*n)$ B->#*

*$A +n/(n\*n)$ A->+TA*

*$AT+ +n/(n\*n)$ 匹配*

*$AT n/(n\*n)$ T->FB*

*$ABF n/(n\*n)$ F->n*

*$ABn n/(n\*n)$ 匹配*

*$AB /(n\*n)$ B->/FB*

*$ABF/ /(n\*n)$ 匹配*

*$ABF (n\*n)$ F->(E)*

*$AB)E( (n\*n)$ 匹配*

*$AB)E n\*n)$ E->TA*

*$AB)AT n\*n)$ T->FB*

*$AB)ABF n\*n)$ F->n*

*$AB)ABn n\*n)$ 匹配*

*$AB)AB \*n)$ B->\*FB*

*$AB)ABF\* \*n)$ 匹配*

*$AB)ABF n)$ F->n*

*$AB)ABn n)$ 匹配*

*$AB)AB )$ B->#*

*$AB)A )$ A->#*

*$AB) )$ 匹配*

*$AB $ B->#*

*$A $ A->#*

*$ $ acc*

## 测试用例二：

还是使用题目所给文法，更复杂的测试语句：

输出结果：

*请输入起始符: E*

*请输入非终结符个数: 5*

*请输入5个非终结符: E T A F B*

*请输入终结符个数: 7*

*请输入7个终结符: + - \* / ( ) n*

*请输入产生式个数: 10*

*请输入10个产生式: E->TA*

*A->+TA*

*A->-TA*

*A->#*

*T->FB*

*B->\*FB*

*B->/FB*

*B->#*

*F->(E)*

*F->n*

*First集:*

*E: { (,n }*

*T: { (,n }*

*A: { #,+,- }*

*F: { (,n }*

*B: { #,\*,/ }*

*Follow集:*

*E: { $,) }*

*T: { $,),+,- }*

*A: { $,) }*

*F: { $,),\*,+,-,/ }*

*B: { $,),+,- }*

*预测分析表如下:*

*+ - \* / ( ) n $*

*E E->TA E->TA*

*T T->FB T->FB*

*A A->+TA A->-TA A-># A->#*

*F F->(E) F->n*

*B B-># B-># B->\*FB B->/FB B-># B->#*

*该文法是LL(1)文法!*

*请输入要分析的字符串: ((n))+n-n/(n)*

*栈 输入 输出*

*$E ((n))+n-n/(n)$ E->TA*

*$AT ((n))+n-n/(n)$ T->FB*

*$ABF ((n))+n-n/(n)$ F->(E)*

*$AB)E( ((n))+n-n/(n)$ 匹配*

*$AB)E (n))+n-n/(n)$ E->TA*

*$AB)AT (n))+n-n/(n)$ T->FB*

*$AB)ABF (n))+n-n/(n)$ F->(E)*

*$AB)AB)E( (n))+n-n/(n)$ 匹配*

*$AB)AB)E n))+n-n/(n)$ E->TA*

*$AB)AB)AT n))+n-n/(n)$ T->FB*

*$AB)AB)ABF n))+n-n/(n)$ F->n*

*$AB)AB)ABn n))+n-n/(n)$ 匹配*

*$AB)AB)AB ))+n-n/(n)$ B->#*

*$AB)AB)A ))+n-n/(n)$ A->#*

*$AB)AB) ))+n-n/(n)$ 匹配*

*$AB)AB )+n-n/(n)$ B->#*

*$AB)A )+n-n/(n)$ A->#*

*$AB) )+n-n/(n)$ 匹配*

*$AB +n-n/(n)$ B->#*

*$A +n-n/(n)$ A->+TA*

*$AT+ +n-n/(n)$ 匹配*

*$AT n-n/(n)$ T->FB*

*$ABF n-n/(n)$ F->n*

*$ABn n-n/(n)$ 匹配*

*$AB -n/(n)$ B->#*

*$A -n/(n)$ A->-TA*

*$AT- -n/(n)$ 匹配*

*$AT n/(n)$ T->FB*

*$ABF n/(n)$ F->n*

*$ABn n/(n)$ 匹配*

*$AB /(n)$ B->/FB*

*$ABF/ /(n)$ 匹配*

*$ABF (n)$ F->(E)*

*$AB)E( (n)$ 匹配*

*$AB)E n)$ E->TA*

*$AB)AT n)$ T->FB*

*$AB)ABF n)$ F->n*

*$AB)ABn n)$ 匹配*

*$AB)AB )$ B->#*

*$AB)A )$ A->#*

*$AB) )$ 匹配*

*$AB $ B->#*

*$A $ A->#*

*$ $ acc*

## 测试用例三：

对于输入错误的情况，停止程序并报错：

*请输入起始符: E*

*请输入非终结符个数: 5*

*请输入5个非终结符: E T A F B*

*请输入终结符个数: 7*

*请输入7个终结符: + - \* / ( ) n*

*请输入产生式个数: 10*

*请输入10个产生式: E->TA*

*A->+TA*

*A->-TA*

*A->#*

*T->FB*

*B->\*FB*

*B->/FB*

*B->#*

*F->(E)*

*F->n*

*First集:*

*E: { (,n }*

*T: { (,n }*

*A: { #,+,- }*

*F: { (,n }*

*B: { #,\*,/ }*

*Follow集:*

*E: { $,) }*

*T: { $,),+,- }*

*A: { $,) }*

*F: { $,),\*,+,-,/ }*

*B: { $,),+,- }*

*预测分析表如下:*

*+ - \* / ( ) n $*

*E E->TA E->TA*

*T T->FB T->FB*

*A A->+TA A->-TA A-># A->#*

*F F->(E) F->n*

*B B-># B-># B->\*FB B->/FB B-># B->#*

*该文法是LL(1)文法!*

*请输入要分析的字符串: n+nn*

*栈 输入 输出*

*$E n+nn$ E->TA*

*$AT n+nn$ T->FB*

*$ABF n+nn$ F->n*

*$ABn n+nn$ 匹配*

*$AB +nn$ B->#*

*$A +nn$ A->+TA*

*$AT+ +nn$ 匹配*

*$AT nn$ T->FB*

*$ABF nn$ F->n*

*$ABn nn$ 匹配*

*$AB n$ error!*

## 测试用例四：

本样例使用了一个更为特殊的文法G如下：

E->AB

A->aA|ε

B->bB|ε

这个文法较为特殊的是，E的产生式右侧可以完全变为空,此时计算E的FIRST集的时候需要含空，输出结果如下，可以看到FIRST集里是含有空符号的，输出结果正确！

*请输入起始符: E*

*请输入非终结符个数: 3*

*请输入3个非终结符: E A B*

*请输入终结符个数: 2*

*请输入2个终结符: a b*

*请输入产生式个数: 5*

*请输入5个产生式: E->AB*

*A->aA*

*A->#*

*B->bB*

*B->#*

*First集:*

*E: { #,a,b }*

*A: { #,a }*

*B: { #,b }*

*Follow集:*

*E: { $ }*

*A: { $,b }*

*B: { $ }*

*预测分析表如下:*

*a b $*

*E E->AB E->AB E->AB*

*A A->aA A-># A->#*

*B B->bB B->#*

*该文法是LL(1)文法!*

*请输入要分析的字符串: aaabb*

*栈 输入 输出*

*$E aaabb$ E->AB*

*$BA aaabb$ A->aA*

*$BAa aaabb$ 匹配*

*$BA aabb$ A->aA*

*$BAa aabb$ 匹配*

*$BA abb$ A->aA*

*$BAa abb$ 匹配*

*$BA bb$ A->#*

*$B bb$ B->bB*

*$Bb bb$ 匹配*

*$B b$ B->bB*

*$Bb b$ 匹配*

*$B $ B->#*

*$ $ acc*

## 测试用例五：

此样例测试了不是LL（1）文法的情况，采用了PPT上的一个例子：





输出结果：

*请输入起始符: S*

*请输入非终结符个数: 5*

*请输入5个非终结符: S A B C D*

*请输入终结符个数: 3*

*请输入3个终结符: a b c*

*请输入产生式个数: 8*

*请输入8个产生式: S->AB*

*A->Da*

*A->#*

*B->cC*

*C->aADC*

*C->#*

*D->b*

*D->#*

*First集:*

*S: { a,b,c }*

*A: { #,a,b }*

*B: { c }*

*C: { #,a }*

*D: { #,b }*

*Follow集:*

*S: { $ }*

*A: { $,a,b,c }*

*B: { $ }*

*C: { $ }*

*D: { $,a }*

*该文法不是LL(1)文法!*

结合PPT上的内容，可以看出FIRST集和FOLLOW集的生成是正确的，但是该文法不是LL(1)文法，所以不能分析预测程序。

# 五、心得体会

本次实验是构造了一个对所有LL（1）文法普适的文法分析处理程序。先生成文法的FIRST集和FOLLOW集，然后构造预测分析表，根据分析表结果判断是否为LL（1）文法，再对输入的字符串进行语法分析。

语法分析的核心部分是构造预测分析表，而为了构造分析表，需要定义与文法有关的两个集合FIRST和FOLLOW。对于G的任何非终结符号A ，FIRST(A)集表示由A推导出的所有开头终结符号的集合，FOLLOW（A）是该文法的所有句型中紧跟在A之后出现的终结符号或$组成的集合。本次实验的一大难点就是如何生成FIRST和FOLLOW集，以及在生成中对空字符的处理；本实验是采用了遍历并累计记录的方式，如果每一个字符都能变成空，则整个式子都可以为空，然后再进行对空的处理。此次实验中，对空的处理让人非常头疼，我也因为这个卡了很久，修改了很多个bug，这也为我提了个醒，在后面做题的时候，一定一定要注意对空的处理。

有了FIRST集和FOLLOW集，后面的处理相对就轻松了很多，根据书上的伪代码分类处理就可以，需要注意的点是更新栈中的内容时，还是要注意对空的处理，同时还要逆序放入栈中，因为栈是先进后出的，这样的处理方式也符合输入情况。值得一提的是，此次实验采用的面对对象的方式，让分析预测表analyze\_table继承了文法grammar，在类中写与该类有关的函数，过程清晰，也省去了传参的麻烦。

经过这次语法分析程序的编写实验，加深了我对语法分析的认识，它是一种常用的、确定的自顶向下分析方法，是一种不含递归调用的有效分析方法，它使用一张分析表和一个分析栈进行联合控制，实现对输入符号串的自顶向下分析，后续学习中还会自底向上等分析方法，此次实验加深了我对该种方法的了解，为后续的学习奠定了基础。

# 六、附录：源代码

**grammar.h**

#pragma once

#include <iostream>

#include <string>

#include <cstring>

#include <vector>

#include <algorithm>

#include <set>

using namespace std;

const int MAXN = 128;

class EXP//产生式

{

public:

    char left = '\0';//左部

    string right = "";//右部

};

class grammar

{

public:

    int N\_cnt;//非终结符个数

    int T\_cnt;//终结符个数

    int G\_cnt;//产生式个数

    EXP exp[MAXN];//产生式

    set<char> FIRST[MAXN];//FIRST集

    set<char> FOLLOW[MAXN];//FOLLOW集

    char N\_begin;            //起始符

    char N\_sym[MAXN] = { 0 };//非终结符

    char T\_sym[MAXN] = { 0 };//终结符

    bool whe\_find\_first[MAXN] = { false };  //表示该符号是否已经找过其FIRST集

    bool whe\_find\_follow[MAXN] = { false };  //表示该符号是否已经找过其FOLLOW集

    grammar();

    ~grammar();

    void input();   //输入文法

    bool isN(char ch);//判断一个字符是否为非终结符

    int getT(char ch);//获取终结符下标

    int getN(char ch);//获取非终结符下标

    void getFirst(char ch);//获取FIRST集

    void getFollow(char ch);//获取FOLLOW集

    void printSet();//打印文法

};

**grammar.cpp**

#include "grammar.h"

#pragma once

grammar::grammar(){

    input();

    for (int i = 0; i < N\_cnt; i++)

    {

        getFirst(N\_sym[i]);

    }

    for (int i = 0; i < N\_cnt; i++)

    {

        getFollow(N\_sym[i]);

    }

}

grammar::~grammar() {

}

void grammar::input() {

    cout << "请输入起始符: ";

    cin >> N\_begin;

    cout << "请输入非终结符个数: ";

    cin >> N\_cnt;

    cout << "请输入" << N\_cnt << "个非终结符: ";

    for (int i = 0; i < N\_cnt; i++) {

        cin >> N\_sym[i];

        //起始符号的FOLLOW集需要多添加一个$

        if (N\_sym[i] == N\_begin)

            FOLLOW[getN(N\_begin)].insert('$');

    }

    cout << "请输入终结符个数: ";

    cin >> T\_cnt;

    cout << "请输入" << T\_cnt << "个终结符: ";

    for (int i = 0; i < T\_cnt; i++) {

        cin >> T\_sym[i];

    }

    T\_sym[T\_cnt] = '$';

    T\_cnt++;

    cout << "请输入产生式个数: ";

    cin >> G\_cnt;

    cout << "请输入" << G\_cnt << "个产生式: ";

    for (int i = 0; i < G\_cnt; i++) {

        char tmp;

        cin >> exp[i].left;

        cin >> tmp >> tmp;

        cin >> exp[i].right;

    }

}

bool grammar::isN(char ch) {

    for (int i = 0; i < N\_cnt; i++) {

        if (ch == N\_sym[i])

            return true;

    }

    return false;

}

int grammar::getT(char ch) {

    for (int i = 0; i < T\_cnt; i++) {

        if (ch == T\_sym[i])

            return i;

    }

    return -1;

}

int grammar::getN(char ch) {

    for (int i = 0; i < N\_cnt; i++) {

        if (ch == N\_sym[i])

            return i;

    }

    return -1;

}

void grammar::getFirst(char ch) {

    //如果之前已经找过该符号的FIRST集，则退出

    if (whe\_find\_first[ch])

        return;

    bool nulls\_flag = false;   //用于存储First集是否有空串$

    int total = 0;             //记录非终结符产生$的个数

    int loc\_ch = getN(ch);

    for (int i = 0; i < G\_cnt; i++) {

        //如果当前产生式左部是需要找FIRST集的非终结符

        if (ch == exp[i].left) {

            for (int j = 0; j < (int)exp[i].right.length(); j++) {

                //如果当前字符是终结符

                if (!isN(exp[i].right[j])) {

                    FIRST[loc\_ch].insert(exp[i].right[j]);

                    break;

                }

                //如果是非终结符，则求它的FIRST集

                getFirst(exp[i].right[j]);

                //把该非终结符的FIRST集加到当前非终结符的FIRST集中

                set<char>::iterator it;

                int locj = getN(exp[i].right[j]);

                for (it = FIRST[locj].begin(); it != FIRST[locj].end(); it++) {

                    if (\*it == '#') {

                        nulls\_flag = true;

                    }

                    else {

                        FIRST[loc\_ch].insert(\*it);

                    }

                }

                //如果没有空串则不必再找下一个符号了

                if (!nulls\_flag)

                    break;

                else {

                    nulls\_flag = false;

                    total++;

                }

            }

            //如果右部所有符号的First集都有空串,则ch的First集也有空串

            if (total == exp[i].right.length())

                FIRST[loc\_ch].insert('#');

        }

    }

    whe\_find\_first[ch] = true;

}

void grammar::getFollow(char ch) {

    //如果之前已经找过该符号的FOLLOW集，则退出

    if (whe\_find\_follow[ch])

        return;

    int loc\_ch = getN(ch);

    for (int i = 0; i < G\_cnt; i++) {

        int index = -1;

        int rlen = exp[i].right.length();

        //找当前字符在产生式中的位置

        for (int j = 0; j < rlen; j++) {

            if (ch == exp[i].right[j]) {

                index = j;

                break;

            }

        }

        //如果当前产生式找到了ch，且ch不在最后一个位置

        if (index != -1 && index < rlen - 1) {

            int total = 0;

            bool nulls\_flag = false;

            char current\_ch;

            for (int j = index + 1; j < exp[i].right.length(); j++) {

                current\_ch = exp[i].right[j];

                //如果是终结符，则加入到FOLLOW集中

                if (!isN(current\_ch)) {

                    FOLLOW[loc\_ch].insert(current\_ch);

                    break;

                }

                //如果是非终结符，则把非终结符的除空的FRIST集加入FOLLOW中

                else {

                    set<char>::iterator it;

                    for (it = FIRST[getN(current\_ch)].begin(); it != FIRST[getN(current\_ch)].end(); it++)

                    {

                        if (\*it == '#')

                            nulls\_flag = true;

                        else

                            FOLLOW[loc\_ch].insert(\*it);

                    }

                    //如果该符号不会变成空，则不需要再遍历后面的符号

                    if (!nulls\_flag)

                        break;

                    else {

                        nulls\_flag = false;

                        total++;

                    }

                }

            }

            //如果后面可以生成空串，则加入产生式左边的符号的FOLLOW集

            if (total == (exp[i].right.length() - index - 1)) {

                char temp = exp[i].left;

                getFollow(temp);

                set<char>::iterator it;

                for (it = FOLLOW[getN(temp)].begin(); it != FOLLOW[getN(temp)].end(); it++)

                    FOLLOW[loc\_ch].insert(\*it);

            }

        }

        //如果当前产生式ch在最后一个位置，则加左部的FOLLOW集,注意要避免循环递归的情况

        else if (index == rlen - 1 && ch != exp[i].left) {

            char temp = exp[i].left;

            getFollow(temp);

            set<char>::iterator it;

            for (it = FOLLOW[getN(temp)].begin(); it != FOLLOW[getN(temp)].end(); it++)

                FOLLOW[loc\_ch].insert(\*it);

        }

    }

    whe\_find\_follow[ch] = true;

}

void grammar::printSet() {

    set<char>::iterator it;

    cout << "First集:" << endl;

    for (int i = 0; i < N\_cnt; i++)

    {

        cout << N\_sym[i] << ": { ";

        for (it = FIRST[i].begin(); it != FIRST[i].end(); it++) {

            cout << \*it;

            it++;

            if (it != FIRST[i].end())

                cout << ",";

            it--;

        }

        cout << " }" << endl;

    }

    cout << "Follow集:" << endl;

    for (int i = 0; i < N\_cnt; i++)

    {

        cout << N\_sym[i] << ": { ";

        for (it = FOLLOW[i].begin(); it != FOLLOW[i].end(); it++) {

            cout << \*it;

            it++;

            if (it != FOLLOW[i].end())

                cout << ",";

            it--;

        }

        cout << " }" << endl;

    }

}

**analyze\_table.h**

#pragma once

#include "grammar.h"

class analyze\_table:public grammar {

public:

    string str = "";//待分析的字符串

    int detect[MAXN][MAXN] = {};//计数该位置产生式的个数，用于判断是否是LL(1)文法

    int analyze\_t[MAXN][MAXN];//分析预测表

    vector<char> stack;       //分析栈

    vector<char>  remainder;  //输入缓冲区

    analyze\_table();

    ~analyze\_table();

    void getAT();  //生成预测分析表

    void printAT();//输出预测分析表

    bool judgeL1();//判断是否为LL(1)文法

    void inputS(); //输入字符串

    void analyze();//分析预测过程

};

**analyze\_table.cpp**

#include "analyze\_table.h"

analyze\_table::analyze\_table() {

    for (int i = 0; i < MAXN; i++) {

        for (int j = 0; j < MAXN; j++) {

            analyze\_t[i][j] = -1;

        }

    }

    getAT();

}

analyze\_table::~analyze\_table() {

}

void analyze\_table::getAT() {

    for (int i = 0; i < G\_cnt; i++) {

        int nulls\_flag = false;

        int total = 0;

        char temp;

        for (int j = 0; j < exp[i].right.length(); j++) {

            temp = exp[i].right[j];

            //如果是终结符

            if (!isN(temp)) {

                //如果产生式右侧是空，则标记，后续会处理加入FOLLOW集

                if (temp == '#') {

                    nulls\_flag = true;

                    total++;

                }

                //该终结符不是空串，更新预测分析表

                else {

                    analyze\_t[getN(exp[i].left)][getT(temp)] = i;

                    detect[getN(exp[i].left)][getT(temp)] ++;

                    break;

                }

            }

            //如果是非终结符,则遍历其FIRST集

            else {

                set<char>::iterator it;

                for (it = FIRST[getN(temp)].begin(); it != FIRST[getN(temp)].end(); it++)

                {

                    //把FIRST集中所有的非空元素a放入M[A,a]中

                    if (\*it == '#') {

                        nulls\_flag = true;

                        total++;

                    }

                    else {

                        analyze\_t[getN(exp[i].left)][getT(\*it)] = i;

                        detect[getN(exp[i].left)][getT(\*it)] ++;

                    }

                }

            }

            if (!nulls\_flag)

                break;

            else

                nulls\_flag = true;

        }

        //若右边可以全部为空，则遍历左部的follow集

        if (total == exp[i].right.length()) {

            set<char>::iterator it;

            for (it = FOLLOW[getN(exp[i].left)].begin();it != FOLLOW[getN(exp[i].left)].end(); it++)

            {

                detect[getN(exp[i].left)][getT(\*it)]++;

                analyze\_t[getN(exp[i].left)][getT(\*it)] = i;

            }

        }

    }

}

void analyze\_table::printAT() {

    cout << "预测分析表如下:" << endl << "\t";

    for (int i = 0; i < T\_cnt; i++) {

        cout << T\_sym[i] << "\t";

    }

    cout << endl;

    for (int i = 0; i < N\_cnt; i++) {

        cout << N\_sym[i] << "\t";

        for (int j = 0; j < T\_cnt; j++) {

            if (analyze\_t[i][j] >= 0) {

                cout << exp[analyze\_t[i][j]].left << "->" << exp[analyze\_t[i][j]].right << "\t";

            }

            else {

                cout << "\t";

            }

        }

        cout << endl;

    }

}

bool analyze\_table::judgeL1() {

    for (int i = 0; i < MAXN; i++)

        for (int j = 0; j < MAXN; j++)

        {

            if (detect[i][j] > 1)

            {

                cout << "该文法不是LL(1)文法!" << endl;

                return false;

            }

        }

    cout << "该文法是LL(1)文法!" << endl;

    return true;

}

void analyze\_table::inputS() {

    cout << "请输入要分析的字符串: ";

    cin >> str;

}

void analyze\_table::analyze() {

    cout << "栈\t\t\t" << "输入\t\t\t" << "输出" << endl;

    remainder.push\_back('$');

    for (int i = str.length() - 1; i >= 0; i--) {

        remainder.push\_back(str[i]);

    }

    stack.push\_back('$');

    stack.push\_back(N\_begin);

    while (!remainder.empty()) {

        string outS = "";

        for (int i = 0; i < stack.size(); i++)

        {

            outS += stack[i];

        }

        if (outS.length() >= 8)

            cout << outS << "\t\t";

        else

            cout << outS << "\t\t\t";//输出分析栈内容,$栈底到栈顶

        outS = "";

        for (int i = remainder.size() - 1; i >= 0; i--)

        {

            outS += remainder[i];

        }

        if (outS.length() >= 8)

            cout << outS << "\t\t";

        else

            cout << outS << "\t\t\t";

        char stack\_top = stack[stack.size() - 1];//表示当前栈顶元素

        char remainder\_top = remainder[remainder.size() - 1];//表示剩余输入的第一个字符

        if (stack\_top == remainder\_top && stack\_top == '$') {

            cout << "acc" << endl;

            return;

        }

        if (stack\_top == remainder\_top) {//remainder\_top恒为终结符

            stack.pop\_back();

            remainder.pop\_back();

            cout << "匹配" << endl;

        }

        else if (analyze\_t[getN(stack\_top)][getT(remainder\_top)] != -1) {

            int loc = analyze\_t[getN(stack\_top)][getT(remainder\_top)];

            stack.pop\_back();

            //如果产生式右侧不为空，则逆序将产生式右式压入栈中

            if (exp[loc].right != "#") {

                for (int i = exp[loc].right.length() - 1; i >= 0; i--) {

                    stack.push\_back(exp[loc].right[i]);

                }

            }

            cout << exp[loc].left << "->" << exp[loc].right << endl;

        }

        else {

            cout << "error!" << endl;

            return;

        }

    }

}

**GrammarAnalysis.cpp**

#include "analyze\_table.h"

using namespace std;

int main(){

    analyze\_table t;

    t.printSet();

    t.printAT();

    if (t.judgeL1()) {

        t.inputS();

        t.analyze();

    }

    return 0;

}