编译技术实验报告

姓名：刘铭源

学号：**2018214937**

班级：软件工程 **18-4** 班

时间：**2020.12.25**

目录

实验一词法分析

一、实验目的

通过本实验的编程实践，使学生了解词法分析的任务，掌握词法分析程序设计的原理和构造方法，使学生对编译的基本概念、原理和方法有完整的和清楚的理解，并能正确地、熟练地运用。

二、实验内容

用 VC++/VB/JAVA 语言实现对 C 语言子集的源程序进行词法分析。通过输入源程序从左到右对字符串进行扫描和分解，依次输出各个单词的内部编码及单词符号自身值；若遇到错误则显示“Error”，然后跳过错误部分继续显示 ；同时进行标识符登记符号表的管理。

以下是实现词法分析设计的主要工作：

（1）从源程序文件中读入字符。

（2）统计行数和列数用于错误单词的定位。

（3）删除空格类字符，包括回车、制表符空格。

（4）按拼写单词，并用（内码，属性）二元式表示。(属性值——token 的机内表示)

（5）如果发现错误则报告出错 7

（6）根据需要是否填写标识符表供以后各阶段使用。

单词的基本分类：

关键字：由程序语言定义的具有固定意义的标识符。也称为保留字例如 if、 for、while、printf ； 单词种别码为 1。

标识符：用以表示各种名字，如变量名、数组名、函数名；

常数： 任何数值常数。如 125, 1,0.5,3.1416;

运算符：+、-、*、/；

关系运算符： <、<=、= 、>、>=、<>；

分界符： ；、，、（、）、[、]；

三、词法分析实验设计思想及算法

1、主程序设计考虑：

程序的说明部分为各种表格和变量安排空间。

在具体实现时，将各类单词设计成结构和长度均相同的形式，较短的关键字后面补空。

k 数组------关键字表，每个数组元素存放一个关键字（事先构造好关键字表）。

s 数组------存放分界符表（可事先构造好分界符表）。为了简单起见，分界符、算术运算和关系运算符都放在 s 表中（编程时，应建立算术运算符表和关系运算符表，并且各有类号），合并成一类。

id 和 ci 数组分别存放标识符和常数。

instring 数组为输入源程序的单词缓存。

outtoken 记录为输出内部表示缓存。

还有一些为造表填表设置的变量。

主程序开始后，先以人工方式输入关键字，造 k 表；再输入分界符等造 p 表。

主程序的工作部分设计成便于调试的循环结构。每个循环处理一个单词；接收键盘上送来的一个单词；调用词法分析过程；输出每个单词的内部码。

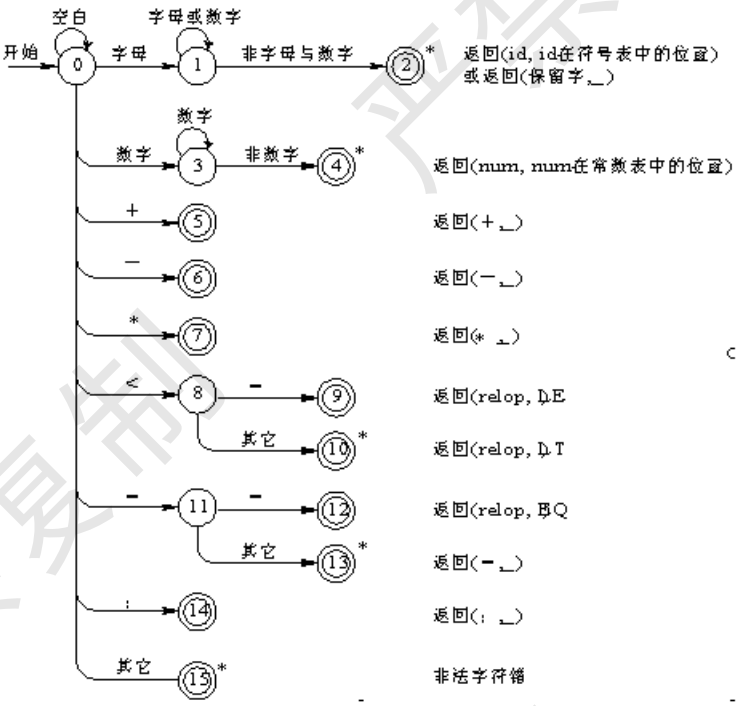
例如，把每一单词设计成如下形式:(type,pointer) 其中 type 指明单词的种类，例如：Pointer 指向本单词存放处的开始位置。

还有一些为造表填表设置的变量。

主程序开始后，先以人工方式输入关键字，造 k 表；再输入分界符等造 p 表。

主程序的工作部分设计成便于调试的循环结构。每个循环处理一个单词；接收键盘上送来的一个单词；调用词法分析过程；输出每个单词的内部码。

例如，把每一单词设计成如下形式:(type,pointer)其中 type 指明单词的种类，例如：Pointer 指向本单词存放处的开始位置。



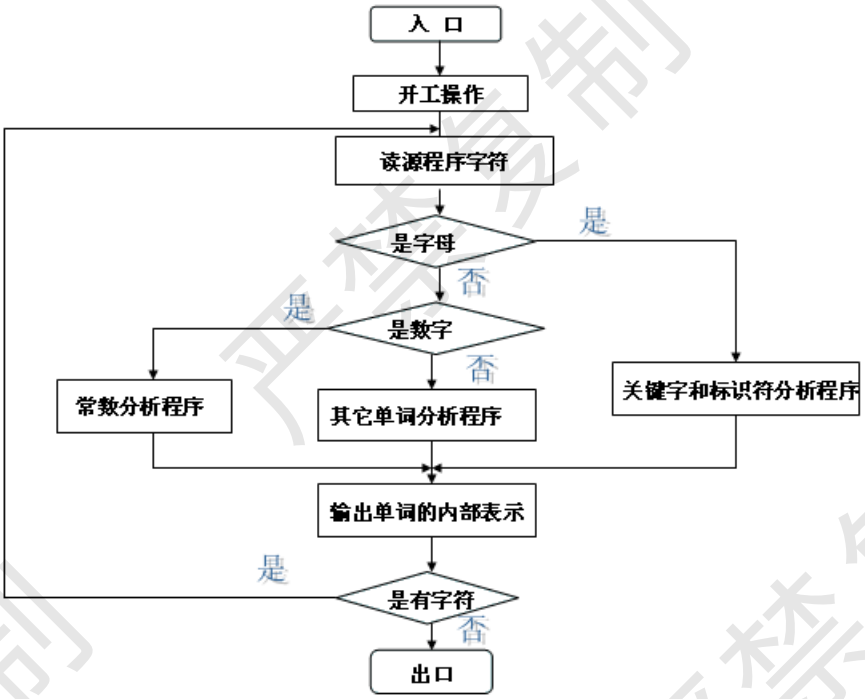


图 1 词法分析设计流程图

2、词法分析过程考虑

根据输入单词的第一个字符（有时还需读第二个字符），判断单词类，产生类号：以字符 k 表示关键字；id 表示标识符；ci 表

示常数；s 表示分界符。

对于标识符和常数，需分别与标识符表和常数表中已登记的元素相比较，如表中已有该元素，则记录其在表中的位置，如未出现过，将标识符按顺序填入数组 id 中，将常数变为二进制形式存入数组中 ci 中，并记录其在表中的位置。

lexical 过程中嵌有两个小过程：一个名为 getchar，其功能为从 instring 中按顺序取出一个字符，并将其指针 pint 加 1；另一个名为 error，当出现错误时，调用这个过程，输出错误编号。

要求：所有识别出的单词都用两个字节的等长表示，称为内部码。第一个字节为 t，第二个字节为 i。t 为单词的种类。关键字的 t=1；分界符的 t=2；算术运算符的 t=3；关系运算符的 t=4；无符号数的 t=5；标识符的 t=6。i 为该单词在各自表中的指针或内部码值。表 1 为关键字表；表 2 为 10 分界符表；表 3 为算术运算符的 i 值；表 4 为关系运算符的 i 值。

表1 关键字表

指针	关键字
0	do
1	end
2	for
3	if
4	printf
5	scanf
6	then
7	while

表2 分界符表

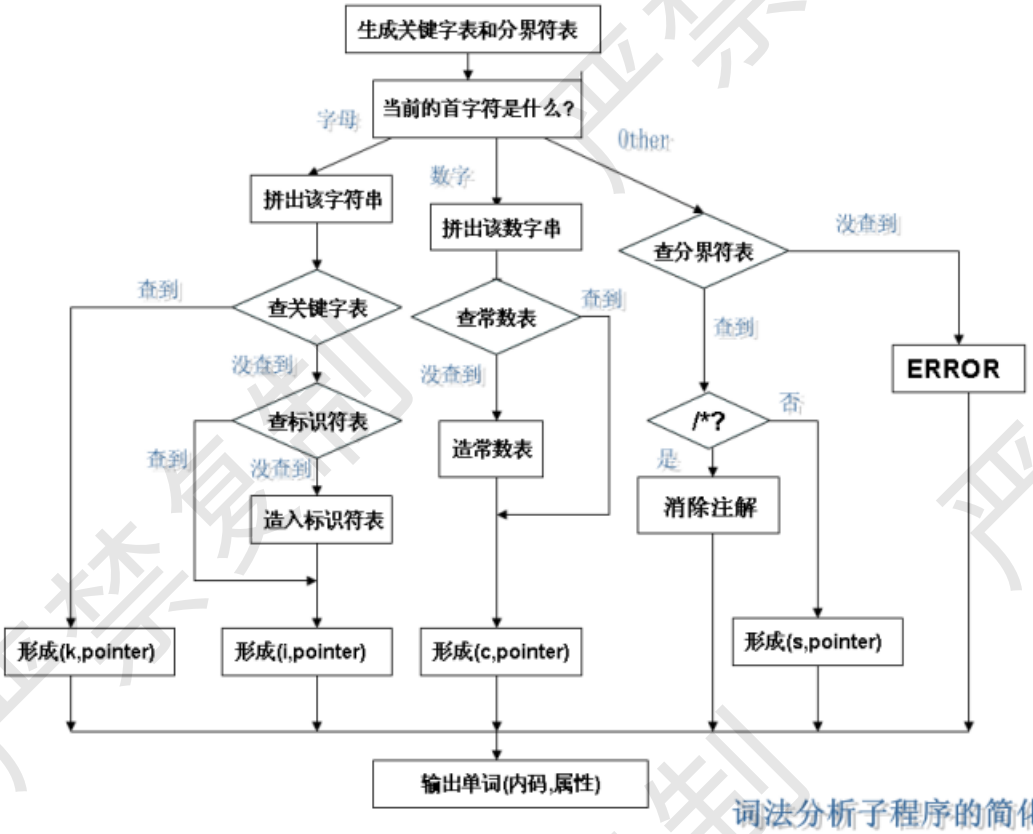
指针	分界符
0	,
1	;
2	(
3	)
4	[
5	]

表3 算术运算符

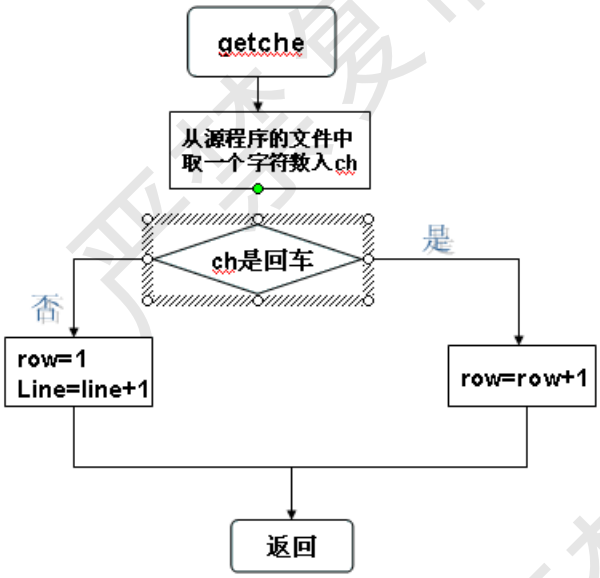
i值	算术运算符
10H	+
11H	-
20H	*
21H	/

表4 关系运算符

i值	关系运算符
00H	<
01H	<=
02H	=
03H	>
04H	>=
05H	<>



词法分析子程序的简化框图



四、实验要求

1、编程时注意编程风格：空行的使用、注释的使用、缩进的使用等。

2、将标识符填写的相应符号表须提供给编译程序的以后各阶段使用。

3、根据测试数据进行测试。测试实例应包括以下三个部分：

全部合法的输入。

各种组合的非法输入。

由记号组成的句子。

4、词法分析程序设计要求输出形式：

If i=0 then n++;

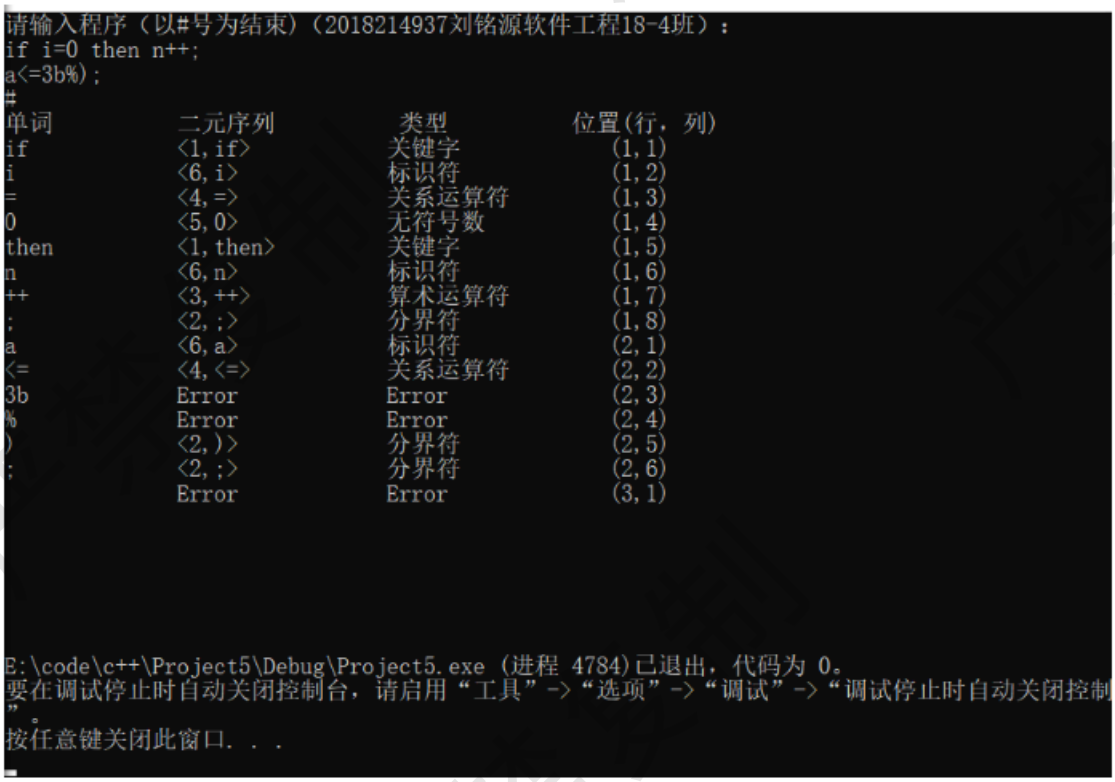
A<=3b%);

五、程序结构描述

函数名称	类型	作用
isnumber	bool	判断数字
isalpha	bool	判断字母
analysis	void	分析函数，调用其他分析函数
constant	void	分析常数
alphabet	void	分析标识符或者关键字
symbol	void	分析其他符号
show	void	根据 outtoken 和 t 输出

六、实验结果及分析

实验平台：VS2019、windows10



七、实验心得

通过此次实验，复习了一下 c++的内容，很久没用使用 c++去写程序了；在编写词法分析程序时候，编写分析函数是一个比较难的问题，需要结合不同字符去调用不同的函数分析函数，在编写词法分析的时候顺便复习一下数据结构的栈和队列。

八、代码

Main.cpp

```
#include<iostream>
```

```
#include<stdio.h>
#include<cstring>
#include<iomanip>
using namespace std;
const char* k[8] = { "do","end","for","if","printf","scanf","then","while" };     //关键字表
const char* s1[6] = { ",",";","(",")","[","]" };//分解符表
const char* s2[6] = { "+","-","*","/","++","--", };//算术运算符表
const char* s3[7] = { "<=",">",">=","=",">",">=","<>" };//关系运算符表

//全局变量
int row = 1, line = 1;//全局变量，保存行列数
int t, p = 0;//种别码,以及记录指针
char instring[100];//保存输入的程序缓存
char outtoken[10];//用于输出
char ci[8], id[8];//暂时保存数字和字符

//函数声明
void analysis();//分析函数,决定调用什么函数进行分析
void symbol();//分析以非字母数字开头的字符
void constant();//分析常数
void alphabet();//分析标识符和关键字
void show();//输出显示函数
bool isnumber(char x);//判断字母
bool isalpha(char x);//判断数字

//程序入口
int main()
{
    cout << "请输入程序（以#号为结束)（2018214937 刘铭源软件工程 18-4 班）：" << endl;
    do {                //写入源程序
        instring[p++] = getchar();
    } while (instring[p - 1] != '#');
    getchar();
    instring[p - 1] = '\0';//吃掉#
    p = 0;//指针归零
    cout << left;
    cout << setw(6) << "单词" << "              " << setw(6) << "二元序列" << "                " <<
setw(6) << "类型" << "            " << setw(6) << "位置(行，列)" << endl;
    while (p < strlen(instring))   //扫描输入字符
    {
        analysis();
        show();
    }
    cout << endl << endl << endl << endl << endl;
```

```
    return 0;
}

bool isnumber(char x)//判断数字
{
    return x >= '0' && x <= '9';
}

bool isalpha(char x)//判断字母
{
    return (x >= 'a' && x <= 'z' || x >= 'A' && x <= 'Z');
}

void analysis()     //分析函数，调用其他分析函数
{
    strcpy_s(outtoken, 2, "\0");//清空 outtoken
    while (instring[p] == ' ' || instring[p] == '\n')//不等于空格和回车则 p++指向下一个
    {
        if (instring[p] == '\n')      //回车即换行了，row++
        {
            row++;
            line = 1;
        }
        p++;
    }   //执行完之后指向之后第一个不是空格的字符
    char ch = instring[p];
    if (isalpha(ch))
        alphabet();
    else if (isnumber(ch))
        constant();
    else
        symbol();
}

void constant()//分析常数
{
    strcpy_s(ci, 2, "\0"); //使用之前清空 ci
    t = 5;
    int i = 0;
    while (isnumber(instring[p]))    //数字存入 ci
        ci[i++] = instring[p++];
    while (isalpha(instring[p]) || isnumber(instring[p])) //如果数字之后接着字符
    {
        ci[i++] = instring[p++];
```

```
            t = 7;      //即出错
        }
        ci[i] = '\0';//结束符
        strcpy_s(outtoken, strlen(ci) + 1, ci);
        line++;
        return;
}

void alphabet()//分析标识符或者关键字
{
        strcpy_s(id, 2, "\0");//使用之前清空 id
        int i = 0;
        while (isalpha(instring[p]) || isnumber(instring[p]))   //读取连续的字母数字序列
            id[i++] = instring[p++];   //p 指向连续序列之后的第一个
        id[i] = '\0';//结束符
        for (i = 0; i < 8; i++)     //查关键字表
            if (strcmp(id, k[i]) == 0)
            {
                t = 1;                  //表示关键字
                line++;
                strcpy_s(outtoken, strlen(id) + 1, id);
                return;     //是关键字直接退出程序
            }
        for (i = 0; i < strlen(id); i++)//查看是否是标识符
            if (!(isalpha(id[i]) || isnumber(id[i])))
            {
                t = 7;//即出错
                strcpy_s(outtoken, strlen(id) + 1, id);
                line++;
                return;
            }
        line++;
        t = 6;//不是关键字且没有出错即为标识符
        strcpy_s(outtoken, strlen(id) + 1, id);
}


void symbol()
{
        char ch = instring[p++];
        char ch2 = instring[p];
        t = 7;//先设置成出错标志，如果没有找到即为 error
        switch (ch)     //判断种别码 t
        {
```

```
case '+':
    if (ch2 == '+')      //++运算符
        t = 3; break;      //3 表示算数运算符
case '-':
    if (ch2 == '-')      //--运算符
        t = 3; break;    //3 表示算数运算符
case '>':      //>  和>=
    if (ch2 == '=')
        t = 4; break;      //4 表示关系运算符
case '<':   //<和  <=
    if (ch2 == '=' || ch2 == '>')
        t = 4; break;
}
//if 语句判断具有两个符号的运算符
if (ch == '>' && ch2 == '=' || ch == '<' && ch2 == '=' || ch == '<' && ch2 == '>' || ch == '+' &&
ch2 == '+' || ch == '-' && ch2 == '-')
{
    p++;
    outtoken[0] = ch;
    outtoken[1] = ch2;
    outtoken[2] = '\0';
    line++;
    return;
}
else
{
    char chq[2];
    chq[0] = ch;
    chq[1] = '\0';
    for (int i = 0; i < 6; i++)   //分解符比较
        if (strcmp(chq, s1[i]) == 0)
        {
            t = 2;                    //表示分界线符
            break;
        }
    for (int i = 0; i < 6; i++)   //算术运算符比较
        if (strcmp(chq, s2[i]) == 0)
        {
            t = 3;                    //表示算术运算符
            break;
        }
    for (int i = 0; i < 7; i++)   //关系运算符比较
        if (strcmp(chq, s3[i]) == 0)
        {
```

```
                    t = 4;
                    break;
                }
        }
        line++;
        outtoken[0] = ch;
        outtoken[1] = '\0';
        return;
}
void show()     //根据 outtoken 和 t 输出
{
        cout << left;
        if (t == 7)
                cout << setw(6) << outtoken << "               " << setw(6) << "Error" << setw(11) << " " <<
setw(10) << "Error";
        else
        {
                cout << left;
                cout << setw(6) << outtoken << "               " << "<" << t << "," << outtoken;
                cout << setw(6 - strlen(outtoken)) << ">" << "          ";
                switch (t)
                {
                case 1:cout << left << setw(10) << "关键字"; break;
                case 2:cout << left << setw(10) << "分界符"; break;
                case 3:cout << left << setw(10) << "算术运算符"; break;
                case 4:cout << left << setw(10) << "关系运算符"; break;
                case 5:cout << left << setw(10) << "无符号数"; break;
                case 6:cout << left << setw(10) << "标识符"; break;
                }
        }
        cout << "               " << "(" << row << "," << line - 1 << ")" << endl;
}
```

实验二 LL（1 分析法）

一、实验目的

通过完成预测分析法的语法分析程序，了解预测分析法和递归子程序法的区别和联系。使学生了解语法分析的功能，掌握语法分析程序设计的原理和构造方法，训练学生掌握开发应用程序的基本方法。有利于提高学生的专业素质，为培养适应社会多方面需要的能力。

二、实验内容

根据某一文法编制调试 LL （ 1 ）分析程序，以便对任意输入的符号串进行分析。

构造预测分析表，并利用分析表和一个栈来实现对上述程序设计语言的分析程序。

分析法的功能是利用 LL（1）控制程序根据显示栈栈顶内容、向前看符号以及 LL（1）分析表，对输入符号串自上而下的分析过程。

三、LL（1）分析法实验设计思想及算法

模块结构：

（1）定义部分：定义常量、变量、数据结构。

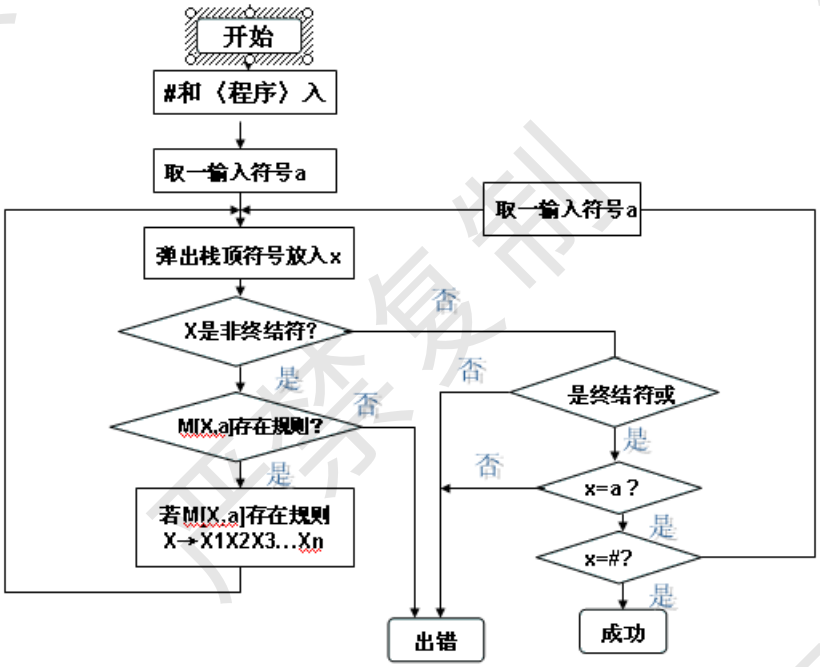
（2）初始化：设立 LL(1)分析表、初始化变量空间（包括堆栈、

结构体、数组、临时变量等）；

（3）控制部分：从键盘输入一个表达式符号串；

（4）利用 LL(1)分析算法进行表达式处理：根据 LL(1)分析表对表达式符号串进行堆栈（或其他）操作，输出分析结果，如果遇到错误则显示错误信息。

LL(1)预测分析流程图



LL(1)预测分析程序流程

四、实验要求

1、编程时注意编程风格：空行的使用、注释的使用、缩进的使用等。

2、如果遇到错误的表达式，应输出错误提示信息。

3、对下列文法，用 LL（1）分析法对任意输入的符号串进行分析：

（1）E->TG

（2）G->+TG|—TG

（3）G->ε

（4）T->FS

（5）S->*FS|/FS

（6）S->ε

（7）F->(E)

（8）F->i

五、程序结构描述

类型	函数	作用
Grammar 类	grammar	定义一个文法类对象
FIRST 类	first	定义 FIRST 集
FOLLOW 类	follow	定义·FOLLOW 集
string	analyseTable[N][N]	定义预测分析表
bool	isNonTerminal	检测一个字符是否为非终结字符
bool	isEmpty	检测一个字符是否为空字

bool	isTerminal	检测一个字符是否为终结字符
void	readGrammar	从控制台读取文法并保存
bool	canCalFIRST	判断一个产生式是否能求出 FIRST 集，能返回 true，否则 false
void	calFIRST	计算能够计算 FIRST 集的产生式
int	getNonTerminalIndex	获取其非终结字符所在的索引
bool	hasEmpty	检测第 i 个 FIRST 集是否有空字

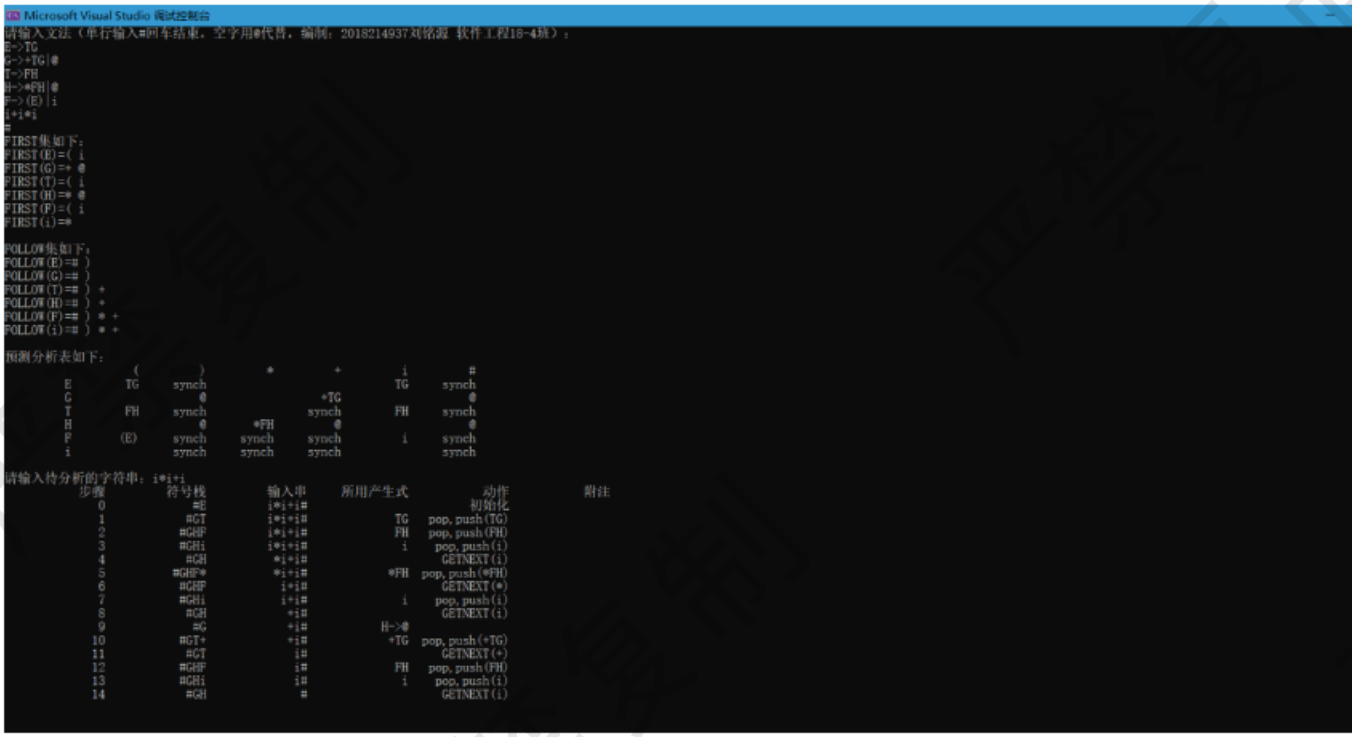
bool	adjustFIRST	判断是否能计算FIRST集(首字符含非终结符)
void	calSetUnion	计算两个集合的并集，即 set(i) = set(i) ∪ set(j)
int	reloadCalCount	更新 calCount
void	calFIRSTSet	计算 FIRST 集
void	printFIRST	输出 first 集
void	getPosition	获取索引（每一个非终结符在产生式的索引，索引保存在容器中）

void	calFollowAndFirstUnion	将 FIRST 集去空加入 FOLLOW 集，i 代表 FOLLOW,i 代表 FIRST 集
void	calFollowAndFollowUnion	计算两个 FOLLOW 集的并集,即 set(i) = set(i) ∪ set(j)
int	reloadFOLLOWCalCount	更新 FOLLOW 集的 calCount
void	calFOLLOWSet	计算 FOLLOW 集
void	getFollowSet	获取每一个非终结符的 FOLLOW 集
void	printFOLLOW	打印 FOLLOW 集
int	getTerminalIndex	获取终结符在 Grammar.terminal[]

		中的索引
string	charToString	将产生式字符转为字符串，i,j 为相应产生式的索引
void	bulidAnalyseTable	构建预测分析表
void	printAnalyseTable	打印预测分析表
string	veToString	将 vector 中的字符转化为字符串
strin	toString	将字符数组有选择的转化为字符串
void	analyseGrammar	核心函数，对语法进行分析

六、实验结果及分析

实验环境：windows10，vs2019



七、实验心得

通过此次实验，完成对 LL（1）文法的扎实学习，实验最难的部分就是 first 集和 follow 集的求解和建表，根据老师上课对 first 集和 follow 集的求解，我立马开始动手写实验，按着实验指导书的内容，学会了如何建立 first 集和 follow 集，如何建一个预测分析表，巩固了自己对 c++的熟练程度。

八、代码

1.Grammar.h

```
#pragma once
#ifndef _grammar_
#define _grammar_

#include <list>
#include <set>
#include <vector>
#define N 50

using namespace std;

//定义文法类，保存文法个数和记录所有文法
class Grammar {
public:
	//保存所有文法
	list<char> grammarTable[N][N];
	//保存终结字符
	char terminalChar[N];
	//保存终结字符的个数
	int terNum;
	//保存每行的产生式的个数
	int countEachRow[N];
	//定义文法数量
	int count;

	Grammar() {
		terNum = 0;
	}
};
//保存每个非终结符的 FIRST 集合
class FIRST {
public:
	//保存每个非终结符的 FIRST 集合
	set<char> First[N];
	//保存非终结符
	char nonTerminal[N];
	//保存是否计算出相应的 FIRST 集
	bool flag[N] = { 0 };
```

```
    //保存已计算 FIRST 集的个数
    int calCount;

    FIRST() {
        calCount = 0;
    }

};
class Position {
public:
    int x;
    int y;
    Position() {
        x = -1;
        y = -1;
    }
};
//保存每个非终结符的 FOLLOW 集合
class FOLLOW {
public:
    //保存每个非终结符的 FOLLOW 集合
    set<char> Follow[N];
    //保存非终结符
    char nonTerminal[N];
    //保存是否计算出相应的 FOLLOW 集
    bool flag[N] = { 0 };
    //保存已计算 Follow 集的个数
    int calCount;
    //保存产生式的索引
    vector<Position> position[N];

    FOLLOW() {
        calCount = 0;
    }
};

#endif // _grammar_
```

2.Mian.cpp

```
#include <iostream>
#include <cstring>
#include <algorithm>
#include <string>
```

```
#include <set>
#include <iomanip>
#include <stack>
#include "grammar.h"
using namespace std;
//定义一个文法类对象
Grammar grammar;
//定义 FIRST 集
FIRST first;
//定义·FOLLOW 集
FOLLOW follow;
//定义预测分析表
string analyseTable[N][N];
//检测一个字符是否为非终结字符
bool isNonTerminal(char var);
//检测一个字符是否为空字
bool isEmpty(char var);
//检测一个字符是否为终结字符
bool isTerminal(char var);
//从控制台读取文法并保存
void readGrammar();
//判断一个产生式是否能求出 FIRST 集，能返回 true，否则 false
bool canCalFIRST(int i);
//计算能够计算 FIRST 集的产生式
void calFIRST();
//获取其非终结字符所在的索引
int getNonTerminalIndex(char var);
//检测第 i 个 FIRST 集是否有空字
bool hasEmpty(int i);
//判断是否能计算 FIRST 集(首字符含非终结符)
bool adjustFIRST(int i);
//计算两个集合的并集，即 set(i) = set(i) ∪ set(j)
void calSetUnion(int i, int j);
//更新 calCount
int reloadCalCount();
//计算 FIRST 集
void calFIRSTSet();
//输出 first 集
void printFIRST();
//获取索引（每一个非终结符在产生式的索引，索引保存在容器中）
void getPosition();
//将 FIRST 集去空加入 FOLLOW 集，i 代表 FOLLOW,i 代表 FIRST 集
void calFollowAndFirstUnion(int i, int j);
//计算两个 FOLLOW 集的并集,即 set(i) = set(i) ∪ set(j)
```

```
void calFollowAndFollowUnion(int i, int j);
//更新 FOLLOW 集的 calCount
int reloadFOLLOWCalCount();
//计算 FOLLOW 集
void calFOLLOWSet();
//获取每一个非终结符的 FOLLOW 集
void getFollowSet();
//打印 FOLLOW 集
void printFOLLOW();
//获取终结符在 Grammar.terminal[]中的索引
int getTerminalIndex(char var);
//构建单个产生式的 First 集,i,j 为相应产生式的索引
set<char> buildFirstForOne(int i, int j);
//将产生式字符转为字符串,i,j 为相应产生式的索引
string charToString(int i, int j);
//构建预测分析表
void bulidAnalyseTable();
//打印预测分析表
void printAnalyseTable();
//将 vector 中的字符转化为字符串
string veToString(vector<char>& vec);
//将字符数组有选择的转化为字符串
string toString(char buf[], int start, int end);
//核心函数，对语法进行分析
void analyseGrammar();


int main()
{
    readGrammar();
    calFIRSTSet();
    printFIRST();

    getFollowSet();
    printFOLLOW();
    bulidAnalyseTable();
    printAnalyseTable();
    analyseGrammar();
    return 0;
}
//从控制台读取文法并保存
void readGrammar() {
    //保存输入的第 i 行文法
    string str;
```

```
    //把第 i 行文法转换为字符数组
    char buf[100] = { 0 };
    int i = 0;
    int index = 0;
    int count = 0;
    //临时保存非终结字符
    set<char> ter;
    cout << "请输入文法（单行输入#回车结束，空字用@代替，编制：2018214937 刘铭源  软件工程 18-4 班）： " << endl;
    cin >> str;
    strcpy_s(buf, str.c_str());
    while (str != "#") {
        i = 0;
        count = 0;
        grammar.grammarTable[index][count].push_back(buf[i]);
        //略去"->"
        i += 3;
        //检测是否到边界
        while ((int)buf[i] != 0) {
            //如果检测到"|"
            if ((int)buf[i] == 124) {
                count++;
                i++;
                //保存起始字符
                grammar.grammarTable[index][count].push_back(buf[0]);
                //保存产生式的每个字符
                grammar.grammarTable[index][count].push_back(buf[i]);
                //如果是终结字符则保存
                if (isTerminal(buf[i])) {
                    ter.insert(buf[i]);
                }
                i++;
            }
            else {
                //保存产生式的每个字符
                grammar.grammarTable[index][count].push_back(buf[i]);
                //如果是终结字符则保存
                if (isTerminal(buf[i])) {
                    ter.insert(buf[i]);
                }
                i++;
            }
        }
        grammar.countEachRow[index] = count + 1;
```

```
            index++;
            cin >> str;
            strcpy_s(buf, str.c_str());
        }
        //保留文法个数
        grammar.count = index;
        //保存终结字符
        set<char>::iterator it = ter.begin();
        for (it; it != ter.end(); it++) {
            grammar.terminalChar[grammar.terNum] = *it;
            grammar.terNum++;
        }
        //注意需要把特殊符号"#"，加入
        grammar.terminalChar[grammar.terNum] = '#';
        grammar.terNum++;
    }
    //检测一个字符是否为终结字符,注意空字@也不算终结字符
    bool isTerminal(char var) {
        if ((!isNonTerminal(var)) && (!isEmpty(var))) {
            return true;
        }
        else {
            return false;
        }
    }
    //检测一个字符是否为非终结字符
    bool isNonTerminal(char var) {
        if (((int)var > 64) && ((int)var < 91)) {
            return true;
        }
        else {
            return false;
        }
    }
    //检测一个字符是否为空字
    bool isEmpty(char var) {
        if ((int)var == 64) {
            return true;
        }
        else {
            return false;
        }
    }
    //获取其非终结字符所在的索引
```

```
int getNonTerminalIndex(char var) {
    int index = 0;
    //获取其终结字符所在的索引
    for (index; index < grammar.count; index++) {
        if ((int)var == (int)grammar.grammarTable[index][0].front()) {
            break;
        }
    }
    return index;

}
//检测第 i 个 FIRST 集是否有空字
bool hasEmpty(int i) {
    set<char>::iterator it = first.First[i].begin();
    for (it; it != first.First[i].end(); it++) {
        if ((int)*it == 64) {
            return true;
        }
    }
    return false;
}
//计算两个集合的并集，即 set(i) = set(i) ∪ set(j)，其中 set(j)中去除空字
void calSetUnion(int i, int j) {
    set<char>::iterator it = first.First[j].begin();
    //如果有空字，则去空字
    if (hasEmpty(j)) {
        for (it; it != first.First[j].end(); it++) {
            if (!isEmpty(*it)) {
                first.First[i].insert(*it);
            }
        }
    }
    else {
        for (it; it != first.First[j].end(); it++) {
            first.First[i].insert(*it);
        }
    }
}
//更新 calCount
int reloadCalCount() {
    int count = 0;
    for (int i = 0; i < grammar.count; i++) {
        if (first.flag[i] == true) {
            count++;
```

```
            }
        }
        first.calCount = count;
        return count;
}
//计算 FIRST 集
void calFIRSTSet() {
        while (reloadCalCount() != grammar.count) {
            //扫描每一个产生式
            for (int i = 0; i < grammar.count; i++) {
                //如果没有计算 FIRST 集
                if (!first.flag[i]) {
                    for (int j = 0; j < grammar.countEachRow[i]; j++) {
                        list<char>::iterator it = grammar.grammarTable[i][j].begin();
                        //获取产生式的首字符
                        it++;
                        //如果 it 没有到边界并且是非终结字符并且并且已经计算 FIRST 集并且 FIRST 含有空字
                        while (it != grammar.grammarTable[i][j].end() && isNonTerminal(*it) && first.flag[getNonTerminalIndex(*it)] && hasEmpty(getNonTerminalIndex(*it))) {
                            first.nonTerminal[i] = grammar.grammarTable[i][0].front();
                            // first.flag[i] = true;
                            calSetUnion(i, getNonTerminalIndex(*it));
                            it++;
                        }
                        //如果 it 到边界，说明每个非终结符的 FIRST 集都已经计算出来，并且都含有空字
                        if (it == grammar.grammarTable[i][j].end()) {
                            //把空字加入
                            first.First[i].insert('@');
                            first.flag[i] = true;
                            continue;
                        }
                        //否则，it 没有到边界
                        else {
                            //如果*it 为终结符
                            if (isTerminal(*it)) {
                                first.nonTerminal[i] = grammar.grammarTable[i][0].front();
                                first.flag[i] = true;
                                //把终结字符保存到 FIRST 集
                                first.First[i].insert(*it);
                            }
                            //如果是非终结符
                            else if (isNonTerminal(*it)) {
```

```
                                //如果已经计算过 FIRST 集，则把 FIrst 集加入
                                if (first.flag[getNonTerminalIndex(*it)]) {
                                    first.nonTerminal[i] =
grammar.grammarTable[i][0].front();
                                    first.flag[i] = true;
                                    calSetUnion(i, getNonTerminalIndex(*it));
                                }
                                //没有计算过
                                else {
                                    first.flag[i] = false;
                                }
                            }
                            //如果是空字
                            else {
                                first.nonTerminal[i] = grammar.grammarTable[i][0].front();
                                first.flag[i] = true;
                                //把终结字符保存到 FIRST 集
                                first.First[i].insert(*it);
                            }
                        }
                    }
                }
                //如果计算 FIRST 集
                else {
                    continue;
                }
            }
        }
    }
    //输出 first 集
    void printFIRST() {
        cout << "FIRST 集如下：" << endl;
        for (int i = 0; i < grammar.count; i++) {
            cout << "FIRST" << "(" << first.nonTerminal[i] << ")" << "=";
            set<char>::iterator it;
            for (it = first.First[i].begin(); it != first.First[i].end(); it++) {
                cout << *it << " ";
            }
            cout << endl;
        }
        cout << endl;
    }
    //获取索引（每一个非终结符在产生式的索引，索引保存在容器中）
    void getPosition() {
```

```
    for (int i = 0; i < grammar.count; i++) {
        list<char>::iterator it = grammar.grammarTable[i][0].begin();
        for (int j = 0; j < grammar.count; j++) {
            for (int k = 0; k < grammar.countEachRow[j]; k++) {
                list<char>::iterator itp = grammar.grammarTable[j][k].begin();
                itp++;
                for (itp; itp != grammar.grammarTable[j][k].end(); itp++) {
                    if ((int)*it == (int)*itp) {
                        Position pos;
                        pos.x = j;
                        pos.y = k;
                        //记下其位置
                        follow.position[i].push_back(pos);
                    }
                }
            }
        }
    }
}
//将 FIRST 集去空加入 FOLLOW 集，i 代表 FOLLOW,i 代表 FIRST 集
void calFollowAndFirstUnion(int i, int j) {
    set<char>::iterator it = first.First[j].begin();
    //如果有空字，则去空字
    if (hasEmpty(j)) {
        for (it; it != first.First[j].end(); it++) {
            if (!isEmpty(*it)) {
                follow.Follow[i].insert(*it);
            }
        }
    }
    else {
        for (it; it != first.First[j].end(); it++) {
            follow.Follow[i].insert(*it);
        }
    }
}
//更新 FOLLOW 集的 calCount
int reloadFOLLOWCalCount() {
    int count = 0;
    for (int i = 0; i < grammar.count; i++) {
        if (follow.flag[i] == true) {
            count++;
        }
    }
```

```
        follow.calCount = count;
        return count;
}
//计算两个 FOLLOW 集的并集,即 set(i) = set(i) ∪ set(j)
void calFollowAndFollowUnion(int i, int j) {
        set<char>::iterator it = follow.Follow[j].begin();
        for (it; it != follow.Follow[j].end(); it++) {
                follow.Follow[i].insert(*it);
        }
}
//计算 FOLLOW 集
void calFOLLOWSet() {
        //对于开始符号 S，需将"#"加入其 FOLLOW 集
        follow.Follow[0].insert('#');
        while (reloadFOLLOWCalCount() != grammar.count) {
                for (int i = 0; i < grammar.count; i++) {
                        //如果没有计算 FOLLOW 集，则计算
                        if (!follow.flag[i]) {
                                vector<Position>::iterator it = follow.position[i].begin();
                                for (it; it != follow.position[i].end(); it++) {
                                        int m = (*it).x;
                                        int n = (*it).y;
                                        list<char>::iterator  itp = grammar.grammarTable[m][n].begin();
                                        //使其指向首字符
                                        itp++;
                                        for (itp; itp != grammar.grammarTable[m][n].end(); itp++) {
                                                if ((int)(*itp) == (int)grammar.grammarTable[i][0].front()) {
                                                        itp++;
                                                        break;
                                                }
                                        }
                                        //itp 不指向结尾，并且是非终结符并 FIRST 集含有空字，则继续检
测
                                        while (itp != grammar.grammarTable[m][n].end() &&
isNonTerminal(*itp) && hasEmpty(getNonTerminalIndex(*itp))) {
                                                int index = getNonTerminalIndex(*itp);
                                                follow.nonTerminal[i] = grammar.grammarTable[i][0].front();
                                                //将非终结符去空字的 FIRST 集加入 FOLLOW 集
                                                calFollowAndFirstUnion(i, index);
                                                itp++;
                                        }
                                        //如果 itp 没有指向 end 指针，说明该字符为终结字符或非终结字符
或空字
                                        if (itp != grammar.grammarTable[m][n].end()) {
```

```
                    if (isTerminal(*itp)) {
                        follow.nonTerminal[i] = grammar.grammarTable[i][0].front();
                        //将非终结字符加入 FOLLOW 集
                        follow.Follow[i].insert(*itp);
                        //标记已经计算该非终结符的 FOLLOW 集
                        follow.flag[i] = true;
                    }
                    else if (isNonTerminal(*itp)) {
                        int index = getNonTerminalIndex(*itp);
                        follow.nonTerminal[i] = grammar.grammarTable[i][0].front();
                        //将非终结符去空字的 FIRST 集加入 FOLLOW 集
                        calFollowAndFirstUnion(i, index);
                        //标记已经计算该非终结符的 FOLLOW 集
                        follow.flag[i] = true;
                    }
                    //空字什么也不做
                    else {

                    }
                }
                //itp 指向 end 指针
                else {
                    if (!follow.flag[m]) {
                        //如果没有计算则标记 false
                        follow.flag[i] = false;
                    }
                    else {
                        follow.nonTerminal[i] = grammar.grammarTable[i][0].front();
                        calFollowAndFollowUnion(i, m);
                        //标记已经计算该非终结符的 FOLLOW 集
                        follow.flag[i] = true;
                    }
                }
            }
        }
    }
}
//获取每一个非终结符的 FOLLOW 集
void getFollowSet() {
    getPosition();
    calFOLLOWSet();
}
//打印 FOLLOW 集
```

```
void printFOLLOW() {
    cout << "FOLLOW 集如下：" << endl;
    for (int i = 0; i < grammar.count; i++) {
        cout << "FOLLOW" << "(" << follow.nonTerminal[i] << ")" << "=";
        set<char>::iterator it;
        for (it = follow.Follow[i].begin(); it != follow.Follow[i].end(); it++) {
            cout << *it << " ";
        }
        cout << endl;
    }
    cout << endl;
}
//获取终结符在 Grammar.terminal[]中的索引
int getTerminalIndex(char var)
{
    for (int i = 0; i < grammar.terNum; i++) {
        if ((int)grammar.terminalChar[i] == (int)var) {
            return i;
        }
    }
    //不存在返回-1
    return -1;
}
//构建单个产生式的 First 集,i,j 为相应产生式的索引
set<char> buildFirstForOne(int i, int j) {
    //定义集合
    set<char> temp;
    list<char>::iterator it = grammar.grammarTable[i][j].begin();
    it++;
    for (it; it != grammar.grammarTable[i][j].end(); it++) {
        //如果没有出界，并且是非终结字符，并且 FIRST 集含有空字
        while (it != grammar.grammarTable[i][j].end() && isNonTerminal(*it) &&
hasEmpty(getNonTerminalIndex(*it))) {
            int index = getNonTerminalIndex(*it);
            set<char>::iterator itp = first.First[index].begin();
            for (itp; itp != first.First[index].end(); itp++) {
                //如果不是空字则添加 temp 集合
                if (!isEmpty(*itp)) {
                    temp.insert(*itp);
                }
            }
            it++;
        }
        //没有出界
```

```
            if (it != grammar.grammarTable[i][j].end()) {
                //如果是终结字符或空字，则把终结字符填到 FIRST 集
                if (isTerminal(*it) || isEmpty(*it)) {
                    temp.insert(*it);
                    return temp;
                }
                //否则为非终结符
                else {
                    int index = getNonTerminalIndex(*it);
                    set<char>::iterator itpt = first.First[index].begin();
                    for (itpt; itpt != first.First[index].end(); itpt++) {
                        temp.insert(*itpt);
                    }
                    return temp;
                }
            }
            //如果出界，则退出
            else {
                //说明都是非终结字符，且都含有空字
                temp.insert('@');
                return temp;
            }
        }
    }
    //将产生式字符转为字符串,i,j 为相应产生式的索引
    string charToString(int i, int j) {
        char buf[100] = { 0 };
        int count = 0;
        list<char>::iterator it = grammar.grammarTable[i][j].begin();
        it++;
        for (it; it != grammar.grammarTable[i][j].end(); it++) {
            buf[count] = *it;
            count++;
        }
        buf[count] = '\0';
        string str(buf);
        return str;
    }
    //构建预测分析表
    void bulidAnalyseTable() {
        bool flag = false;
        //遍历每个非终结符
        for (int i = 0; i < grammar.count; i++) {
            //遍历每个非终结字符的产生式
```

```
            for (int j = 0; j < grammar.countEachRow[i]; j++) {
                flag = false;
                set<char> firstSet = buildFirstForOne(i, j);
                set<char>::iterator it = firstSet.begin();
                for (it; it != firstSet.end(); it++) {
                    //如果 FIRST 集存在空字，记上标记
                    if (isEmpty(*it)) {
                        flag = true;
                    }
                    //否则将相应的产生式加入预测分析表
                    else {
                        //将文法字符转为字符串
                        string str = charToString(i, j);
                        analyseTable[i][getTerminalIndex(*it)] = str;
                    }
                }
                //产生式的 FIRST 集中含有空字
                if (flag) {
                    //获取 i 为索引的非终结字符的 FOLLOW 集
                    set<char>::iterator it = follow.Follow[i].begin();
                    for (it; it != follow.Follow[i].end(); it++) {
                        if (isTerminal(*it)) {
                            analyseTable[i][getTerminalIndex(*it)] = (string)"@";
                        }
                    }
                }
                //产生式的 FIRST 集中不含有空字
                else {
                    //获取 i 为索引的非终结字符的 FOLLOW 集
                    set<char>::iterator it = follow.Follow[i].begin();
                    for (it; it != follow.Follow[i].end(); it++) {
                        analyseTable[i][getTerminalIndex(*it)] = (string)"synch";
                    }
                }
            }
        }
    }
    //打印预测分析表
    void printAnalyseTable() {
        cout << "预测分析表如下：" << endl;
        //占位符
        cout << setw(10) << " ";
        //循环输出每位终结符
        for (int i = 0; i < grammar.terNum; i++) {
```

```
            cout << setw(10) << grammar.terminalChar[i];
        }
        cout << endl;
        //输出每行
        for (int i = 0; i < grammar.count; i++) {
            //输出非终结字符
            cout << setw(10) << grammar.grammarTable[i][0].front();
            //输出相应的产生式
            for (int j = 0; j < grammar.terNum; j++) {
                cout << setw(10) << analyseTable[i][j];
            }
            cout << endl;
        }
        cout << endl;
    }
    //将 vector 中的字符转化为字符串
    string veToString(vector<char>& vec) {
        char buf[N] = { 0 };
        int index = 0;
        vector<char>::iterator it = vec.begin();
        for (it; it != vec.end(); it++) {
            buf[index] = *it;
            index++;
        }
        buf[index] = '\0';
        string str(buf);
        return str;
    }
    //将字符数组有选择的转化为字符串
    string toString(char buf[], int start, int end) {
        char temp[N];
        int index = 0;
        for (start; start <= end; start++) {
            temp[index] = buf[start];
            index++;
        }
        temp[index] = '\0';
        string str(temp);
        return str;
    }
    //核心函数，对语法进行分析
    void analyseGrammar() {
        cout << "请输入待分析的字符串：";
        string str;
```

```
        cin >> str;
        //将输入的字符串转化为字符数组
        char buf[N] = { 0 };
        strcpy_s(buf, str.c_str());
        //计算字符的数目
        int count = 0;
        for (int i = 0; buf[i] != 0; i++) {
            count++;
        }
        buf[count++] = '#';
        cout << setw(15) << "步骤" << setw(15) << "符号栈" << setw(15) << "输入串" << setw(15)
<< "所用产生式" << setw(15) << "动作" << setw(15) << "附注" << endl;
        //定义一个分析栈
        stack<char> analyseStack;
        //把'#'和文法开始符号入栈
        analyseStack.push('#');
        analyseStack.push(grammar.grammarTable[0][0].front());
        vector<char> vec;
        vec.push_back('#');
        vec.push_back(grammar.grammarTable[0][0].front());
        //把第一个字符读入 a 中
        char a = buf[0];
        //记录步骤
        int step = 0;
        cout << setw(15) << step << setw(15) << veToString(vec) << setw(15) << toString(buf, 0,
count - 1) << setw(15) << " " << setw(15) << "初始化" << setw(15) << " " << endl;
        //buf[]的索引
        int index = 0;
        bool flag = true;
        while (flag) {
            char ch;
            if (!analyseStack.empty()) {
                ch = analyseStack.top();
                analyseStack.pop();
                vec.pop_back();
            }
            if (isTerminal(ch) && ch != '#') {
                if (ch == a) {

                    index++;
                    a = buf[index];
                    step++;
                    cout << setw(15) << step << setw(15) << veToString(vec) << setw(15) <<
toString(buf, index, count - 1) << setw(15) << " " << setw(15) << (string)"GETNEXT(" + ch +
```

```
(string)")" << setw(15) << " " << endl;

                }
                else {
                    //出错
                    step++;
                    cout << setw(15) << step << setw(15) << veToString(vec) << setw(15) <<
toString(buf, index, count - 1) << setw(15) << " " << setw(15) << (string)"pop" << setw(15) << "
错误，栈顶终结符与输入符号不匹配 " << endl;
                }
            }
            else if (ch == '#') {
                if (ch == a) {
                    flag = false;
                }
                else {
                    //出错
                    cout << "出错";
                    return;
                }
            }
            else if (isNonTerminal(ch)) {
                string str = analyseTable[getNonTerminalIndex(ch)][getTerminalIndex(a)];
                //如果产生式不为空,且不为空字
                if (str != "@" && !str.empty() && str != "synch") {

                    int strSize = str.size();
                    char temp[N];
                    strcpy_s(temp, str.c_str());
                    for (int i = strSize - 1; i >= 0; i--) {
                        analyseStack.push(temp[i]);
                        vec.push_back(temp[i]);
                    }
                    step++;
                    cout << setw(15) << step << setw(15) << veToString(vec) << setw(15) <<
toString(buf, index, count - 1) << setw(15) << str << setw(15) << (string)"pop,push(" + str +
(string)")" << setw(15) << " " << endl;
                }
                //如果[M,a]为空,则跳过输入符号 a
                else if (str.empty()) {
                    //出错
                    index++;
                    a = buf[index];
                    step++;
```

```
                cout << setw(15) << step << setw(15) << veToString(vec) << setw(15) << toString(buf, index, count - 1) << setw(15) << " " << setw(15) << " " << setw(15) << "错，跳过" << endl;
            }
            else if (str == "synch") {
                //如果栈顶为文法开始符号,跳过输入符号
                if (ch == grammar.grammarTable[0][0].front()) {
                    index++;
                    a = buf[index];
                    //文法开始符号入栈
                    analyseStack.push(ch);
                    vec.push_back(ch);
                    step++;
                    cout << setw(15) << step << setw(15) << veToString(vec) << setw(15) << toString(buf, index, count - 1) << setw(15) << " " << setw(15) << " " << setw(15) << "错，跳过" << endl;
                }
                else {
                    step++;
                    cout << setw(15) << step << setw(15) << veToString(vec) << setw(15) << toString(buf, index, count - 1) << setw(15) << " " << setw(15) << " " << setw(15) << (string)"错,M[" + ch + (string)"," + a + (string)"]=synch" + "," + ch + (string)"已弹出栈" << endl;
                }
            }
            //若为空字，什么也不做
            else {
                step++;
                cout << setw(15) << step << setw(15) << veToString(vec) << setw(15) << toString(buf, index, count - 1) << setw(15) << ch + (string)"->" + (string)"@" << setw(15) << " " << setw(15) << "" << endl;
            }
        }
        else {
            //出错
            cout << "出错";
            return;
        }
    }
}
```

实验三 LR（1）分析法

一、实验目的

构造 LR(1)分析程序，利用它进行语法分析，判断给出的符号串是否为该文法识别的句子，了解 LR（K）分析方法是严格的从左向右扫描，和自底向上的语法分析方法。

二、实验内容

对下列文法，用 LR（1）分析法对任意输入的符号串进行分析：

（1）E-> E+T

（2）E->T

（3）T-> T*F

（4）T->F

（5）F-> (E)

（6）F-> i

三、LR（1）分析法实验设计思想及算法

(1)总控程序，也可以称为驱动程序。对所有的 LR 分析器总控程序都是相同的。

(2)分析表或分析函数，不同的文法分析表将不同，同一个文法采用的 LR 分析器

不同时，分析表将不同，分析表又可以分为动作表（ACTION）和状态转换（GOTO）

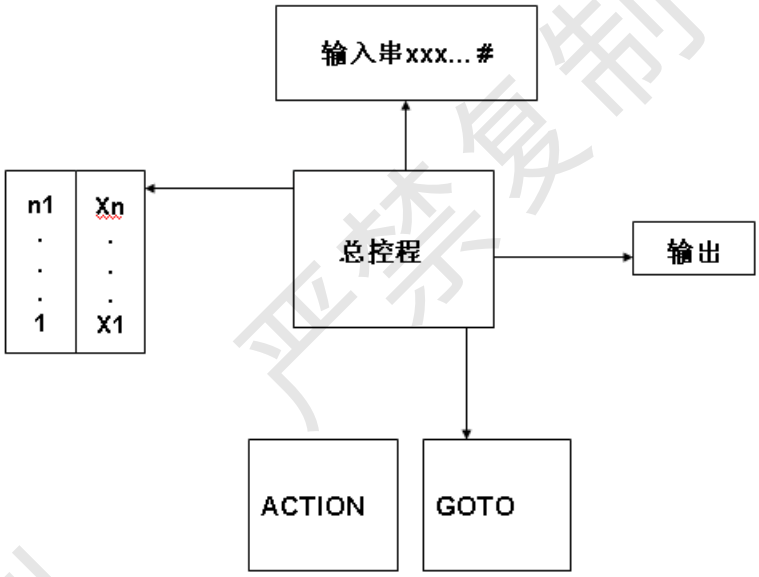
表两个部分，它们都可用二维数组表示。

(3)分析栈，包括文法符号栈和相应的状态栈，它们均是先进后出栈。

分析器的动作就是由栈顶状态和当前输入符号所决定。

LR 分析器由三个部分组成：

LR分析器结构：



其中:SP 为栈指针，S[i]为状态栈，X[i]为文法符号栈。状态转换表用 GOTO[i，X]=j 表示，规定当栈顶状态为 i，遇到当前文法符号为 X 时应转向状态 j，X 为终结符或非终结符。

ACTION[i，a]规定了栈顶状态为 i 时遇到输入符号 a 应执行。动作有四种

可能：

(1)移进：

action[i，a]= Sj：状态 j 移入到状态栈，把 a 移入到文法符号栈，其中 i,j 表示状态号。

(2)归约：

action[i，a]=rk：当在栈顶形成句柄时，则归约为相应的非终结符 A，即文法中有 A- B 的产生式，若 B 的长度为 R(即|B|=R)，则从状态栈和文法符号栈中自顶向下去掉 R 个符号，即栈指针 SP 减去 R，并把 A 移入文法符号栈内，j=GOTO[i,A]移进状态栈，其中 i 为修改指针后的栈顶状态。

(3)接受 acc:

当归约到文法符号栈中只剩文法的开始符号 S 时，并且输入符号串已结束即当前输入符是'#'，则为分析成功。

(4)报错:

当遇到状态栈顶为某一状态下出现不该遇到的文法符号时，则报错，说明输入端不是该文法能接受的符号串

四、实验要求

1、编程时注意编程风格：空行的使用、注释的使用、缩进的使用等。

2、如果遇到错误的表达式，应输出错误提示信息。

3、程序输入/输出实例：

输入一以#结束的符号串(包括+*（）i#)：在此位置输入符号串

五、程序结构描述

类型	函数名	作用
vector<int>	status;	定义状态栈
vector<char>	sign	定义符号栈
vector<char>	inputStr	定义输入的字符串
string	inputVal	记录输入的字符串
Grammar	grammar	定义文法
LRAnalyseTable	analyseTable	定义 LR 分析表
void	readStr	读取输入的字符串
string	vectTrancStr	对栈容器进行输出，i=0，返回 status 中的字符串，i=1，返回 sign 中的字符串，

		i=2 返回 inputStr
void	LRAnalyse	总控，对输入的字符串进行分析

六、实验结果及分析

```
请输入分析的字符串：i+i*i#
      步骤     状态栈     符号栈     输入串                动作说明
         1          0          #     i+i*i#          ACTION[0,i]=S5,状态5入栈
         2         05         #i      +i*i#          r6:F->i归约,GOTO(0,F)=3入栈
         3         03         #F      +i*i#          r4:T->F归约,GOTO(0,T)=2入栈
         4         02         #T      +i*i#          r2:E->T归约,GOTO(0,E)=1入栈
         5         01         #E      +i*i#          ACTION[1,+]=S6,状态6入栈
         6        016        #E+       i*i#          ACTION[6,i]=S5,状态5入栈
         7       0165       #E+i        *i#          r6:F->i归约,GOTO(6,F)=3入栈
         8       0163       #E+F        *i#          r4:T->F归约,GOTO(6,T)=9入栈
         9       0169       #E+T        *i#          ACTION[9,*]=S7,状态7入栈
        10      01697      #E+T*         i#          ACTION[7,i]=S5,状态5入栈
        11    016975     #E+T*i           #          r6:F->i归约,GOTO(7,F)=10入栈
        12   0169710     #E+T*F           #          r3:T->T*F归约,GOTO(6,T)=9入栈
        13      0169       #E+T           #          r1:E->E+T归约,GOTO(0,E)=1入栈
        14        01         #E           #          Acc:分析成功
i+i*i#为合法符号串

E:\code\c++\Project3\Debug\Project3.exe (进程 8032)已退出，代码为 0。
要在调试停止时自动关闭控制台，请启用“工具”->“选项”->“调试”->“调试停止时自动关闭控制台
”。
按任意键关闭此窗口. . .
```

```
请输入分析的字符串：i++i**@@@@#
      步骤     状态栈     符号栈     输入串                动作说明
         1          0          #i++i**@@@@#          ACTION[0,i]=S5,状态5入栈
         2         05         #i++i**@@@@#          r6:F->i归约,GOTO(0,F)=3入栈
         3         03         #F++i**@@@@#          r4:T->F归约,GOTO(0,T)=2入栈
         4         02         #T++i**@@@@#          r2:E->T归约,GOTO(0,E)=1入栈
         5         01         #E++i**@@@@#          ACTION[1,+]=S6,状态6入栈
出错
i++i**@@@@#为非法符号串

E:\code\c++\Project3\Debug\Project3.exe (进程 21156)已退出，代码为 0。
要在调试停止时自动关闭控制台，请启用“工具”->“选项”->“调试”->“调试停止时自动关闭控制台
”。
按任意键关闭此窗口. . .
```

七、实验心得

实验三和实验二是一口气写完了，综合上借鉴了很多实验二的内容，在老师讲解完 LR 分析法时就开始重新的将代码进行编写，发现了之前很多错误的地方，也通过这个实验深了自己的对 LR 分析的理解，也通过这个实验加深自己对编译原理的兴趣。

八、代码

1.LR.h

```
#pragma once
#ifndef _LR_
#define _LR_
#include<cstring>
using namespace std;

class Grammar {
public:
	//产生式的个数
	int grammarNum;
	//定义产生式数组
	string formula[100] = { " ","E->E+T","E->T","T->T*F","T->F","F->(E)","F->i" };

	Grammar() {
		grammarNum = 6;
	}
};
//定义 LR 文法的分析表
class LRAnalyseTable {
public:
	char terminalChar[100] = { 'i','+','*','(',')','#' };
	//定义终结符的个数
	int terNum = 6;
	char nonTerminalChar[100] = { 'E','T','F' };
	//定义非终结符的个数
```

```
    int nonTerNum = 3;
    //定义状态数
    int statusNum = 12;
    string action[12][6] =
{ {"s5","","","s4","",""},{"","s6","","","","acc"},{"","r2","s7","","r2","r2"},{"","r4","r4","","r4","r4"},{
"s5","","","s4","",""},{"","r6","r6","","r6","r6"},{"s5","","","s4","",""}
    ,{"s5","","","s4","",""},{"","s6","","","s11",""},{"","r1","s7","","r1","r1"},{"","r3","r3","","r3","r
3"},{"","r5","r5","","r5","r5"}};
    int goTo[12][3] =
{  {1,2,3},{-1,-1,-1},{-1,-1,-1},{-1,-1,-1},{8,2,3},{-1,-1,-1},{-1,9,3},{-1,-1,10},{-1,-1,-1},{-1,-1,-1},{-1,-1,-
1},{-1,-1,-1} };
    //获取终结符的索引
    int getTerminalIndex(char var) {
        for (int i = 0; i < terNum; i++) {
            if (terminalChar[i] == var) {
                return i;
            }
        }
        return -1;
    }
    //获取非终结符的索引
    int getNonTerminalIndex(char var) {
        for (int i = 0; i < nonTerNum; i++) {
            if (nonTerminalChar[i] == var) {
                return i;
            }
        }
        return -1;
    }
};

#endif // _LR_
```

2.Main.cpp

```
#include <iostream>
#include <vector>
#include <iomanip>
#include <cstring>
#include <sstream>
#include "LR.h"
using namespace std;
//定义状态栈
vector<int> status;
```

```
//定义符号栈
vector<char> sign;
//定义输入的字符串
vector<char> inputStr;
//记录输入的字符串
string inputVal;
//定义文法
Grammar grammar;
//定义 LR 分析表
LRAnalyseTable analyseTable;
//读取输入的字符串
void readStr();
//对栈容器进行输出,i=0,返回 status 中的字符串,i=1,返回 sign 中的字符串，i=2 返回 inputStr
string vectTrancStr(int i);
//总控，对输入的字符串进行分析
void LRAnalyse();

int main()
{
	readStr();
	LRAnalyse();
	return 0;
}
//读取输入的字符串
void readStr() {
	char ch;
	cout << "LR（1）分析程序请以#结束，编制人：刘铭源，2018214937，软件工程 18-4
班" << endl;
	cout << "请输入分析的字符串：";
	cin >> ch;
	while (ch != '#') {
		inputVal += ch;
		inputStr.push_back(ch);
		cin >> ch;
	}
	//把#加入容器
	inputStr.push_back('#');
	inputVal += '#';
}
//对栈容器进行输出,i=0,返回 status 中的字符串,i=1,返回 sign 中的字符串，i=2 返回 inputStr
中的字符串
string vectTrancStr(int i) {
	char buf[100];
	int count = 0;
```

```
        //输出状态栈
        if (i == 0) {
            vector<int>::iterator it = status.begin();
            //将数字转化为字符串
            string str, tempStr;
            for (it; it != status.end(); it++) {
                stringstream ss;
                ss << *it;
                ss >> tempStr;
                str += tempStr;
            }
            return str;
        }
        //输出符号栈
        else if (i == 1) {
            vector<char>::iterator it = sign.begin();
            for (it; it != sign.end(); it++) {
                buf[count] = *it;
                count++;
            }
        }
        //输出待分析的字符串
        else {
            vector<char>::iterator it = inputStr.begin();
            for (it; it != inputStr.end(); it++) {
                buf[count] = *it;
                count++;
            }
        }
        buf[count] = '\0';
        string str(buf);
        return str;
    }
    //总控，对输入的字符串进行分析
    void LRAnalyse() {
        //步骤
        int step = 1;
        //把状态 0 入栈
        status.push_back(0);
        //把#加入符号栈
        sign.push_back('#');
        //输出初始栈状态
        cout << setw(10) << "步骤" << setw(10) << "状态栈" << setw(10) << "符号栈" << setw(10)
    << "输入串" << setw(25) << "动作说明" << endl;
```

```
        //初始状态
        int s = 0;
        //保存之前的状态
        int oldStatus;
        //获取初始符号
        char ch = inputStr.front();
        //如果 action[s][ch] =="acc" ， 则分析成功
        while (analyseTable.action[s][analyseTable.getTerminalIndex(ch)] != "acc") {
            //获取字符串
            string str = analyseTable.action[s][analyseTable.getTerminalIndex(ch)];
            //如果 str 为空，报错并返回
            if (str.size() == 0) {
                cout << "出错" << endl;
                cout << inputVal << "为非法符号串" << endl;
                return;
            }
            //获取 r 或 s 后面的数字
            stringstream ss;
            ss << str.substr(1);
            ss >> s;
            //如果是移进
            if (str.substr(0, 1) == "s") {
                cout << setw(10) << step << setw(10) << vectTrancStr(0) << setw(10) <<
vectTrancStr(1) << setw(10) << vectTrancStr(2) << setw(10) << "A" << "CTION[" << status.back() <<
"," << ch << "]=S" << s << "," << "状态" << s << "入栈" << endl;
                //输入符号入栈
                sign.push_back(ch);
                inputStr.erase(inputStr.begin());
                //将状态数字入栈
                status.push_back(s);
            }
            //如果是归约
            else if (str.substr(0, 1) == "r") {
                //获取第 S 个产生式
                string formu = grammar.formula[s];
                //cout<<s<<endl;
                int strSize = formu.size();
                //将产生式转化为字符数组
                char buf[100];
                strcpy_s(buf, formu.c_str());
                //获取产生式的首字符
                char nonTerCh = buf[0];
                //获取符号栈的出栈次数
                int popCount = strSize - 3;
```

```
                //反向迭代
                vector<int>::reverse_iterator rit = status.rbegin();
                int i = 0;
                for (rit; rit != status.rend(); rit++) {
                    i++;
                    if (i == popCount + 1) {
                        oldStatus = *rit;
                        break;
                    }
                }
                int r = s;
                //修改 s
                s = analyseTable.goTo[oldStatus][analyseTable.getNonTerminalIndex(nonTerCh)];
                cout << setw(10) << step << setw(10) << vectTrancStr(0) << setw(10) << vectTrancStr(1) << setw(10) << vectTrancStr(2) << setw(10) << "r" << r << (string)":" + grammar.formula[r] + (string)"归约,GOTO{" << oldStatus << "," << nonTerCh << ")=" << s << "入栈" << endl;
                //对符号栈进行出栈和状态栈进行出栈
                for (int i = 0; i < popCount; i++) {
                    sign.pop_back();
                    status.pop_back();
                }
                //再对产生式的开始符号入栈
                sign.push_back(nonTerCh);
                //再把新的状态入栈
                status.push_back(s);
            }
            else {
                //什么都不处理
            }
            //步骤数加 1
            step++;

            //获取栈顶状态
            s = status.back();
            //获取输入的字符
            ch = inputStr.front();
        }
        cout << setw(10) << step << setw(10) << vectTrancStr(0) << setw(10) << vectTrancStr(1) << setw(10) << vectTrancStr(2) << setw(10) << "A" << "cc:分析成功" << endl;
        cout << inputVal << "为合法符号串" << endl;
    }
```