《词法分析程序》实验报告—C 语言版本

一、实验内容

设计并实现 C 语言的词法分析程序，要求实现如下功能。

1) 可以识别出用 C 语言编写的源程序中的每个单词符号，并以记号的形式输出每个单词符号。
2) 可以识别并跳过源程序中的注释。
3) 可以统计源程序中的语句行数、各类单词的个数、以及字符总数，并输出统计结果。
4) 检查源程序中存在的词法错误，并报告错误所在的位置。
5) 对源程序中出现的错误进行适当的恢复，使词法分析可以继续进行，对源程序进行一次扫描，既可检查并报告源程序中存在的所有语法错误。

二、实验分析

(一) 语言说明

根据 C 语言标准，可将一个 C 源程序分为以下记号及单词：

1) 标识符：以字母开头的、后跟字母或数字组成的字符串
2) 关键字：标识符集合的子集，C 语言的关键字有 32 个

关键字	意义
void	声明函数无返回值或无参数，声明空类型指针
char	声明字符型变量
short	声明短整形变量
int	声明整形变量
long	声明长整型变量
float	声明浮点型变量
double	声明双精度型变量
sizeof	计算对象所占内存空间的大小
signed	声明有符号类型变量
unsigned	声明无符号类型变量
struct	声明结构体变量
union	声明联合数据类型
enum	声明枚举类型
typedef	给数据类型取别名
auto	声明自动变量，缺省时编译器一般默认为 auto
static	声明静态变量
extern	声明变量在其他文件中声明，亦引用变量
register	声明寄存器变量
const	声明只读变量
volatile	说明程序在执行中可被隐含的改变
if	条件语句
else	条件语句的否认分支，与 if 连用
for	一般循环语句
while	循环语句的循环条件
do	循环语句的循环体
break	跳出当前循环
goto	无条件跳转语句
continue	结束当前循环，开始下一轮循环
return	子程序返回语句，可以带参数，也可以不带参数
switch	用于开关语句
case	开关语句分支
default	开关语句中的其他分支

3) 无符号数：由整数部分、可选的小数部分和可选的指数部分构成
4) 关系运算符：<、<=、==、>、>=、!=
5) 算术运算符：+、-、*、/
6) 标点符号：(、)、{、}、[、]、;、,、\、’、”
7) 赋值号： =
8) 注释标记：以“/*”开始，以“*/”结束 或 “//”
9) 单词符号间的分隔符：空格、\t

(二) 文法产生式

1) 标识符文法

```
id->letter rid
rid-> ε | letter rid | digit rid
```

2) 无符号整数的文法

```
digits->digit remainder
remainder->ε | digit remainder
```

3) 无符号数的文法

```
num->digit num 1
num1->digit num 1 |. num2 | E num4 | ε
num2->digit num3
num3->digit num3 | E num4 | ε
num4->+digits | -digits | digits num5
digits->digit num 5
num5->digit num 5 | ε
```

4) 关系运算符的文法

```
relop-> < | < eq | > | > eq | ! eq
greater-> >
equal-> ==
eq-> =
```

5) 赋值号的文法

```
assign_op-> =
```

6) 算术运算符的文法

```
single->+ | - | * | /
```

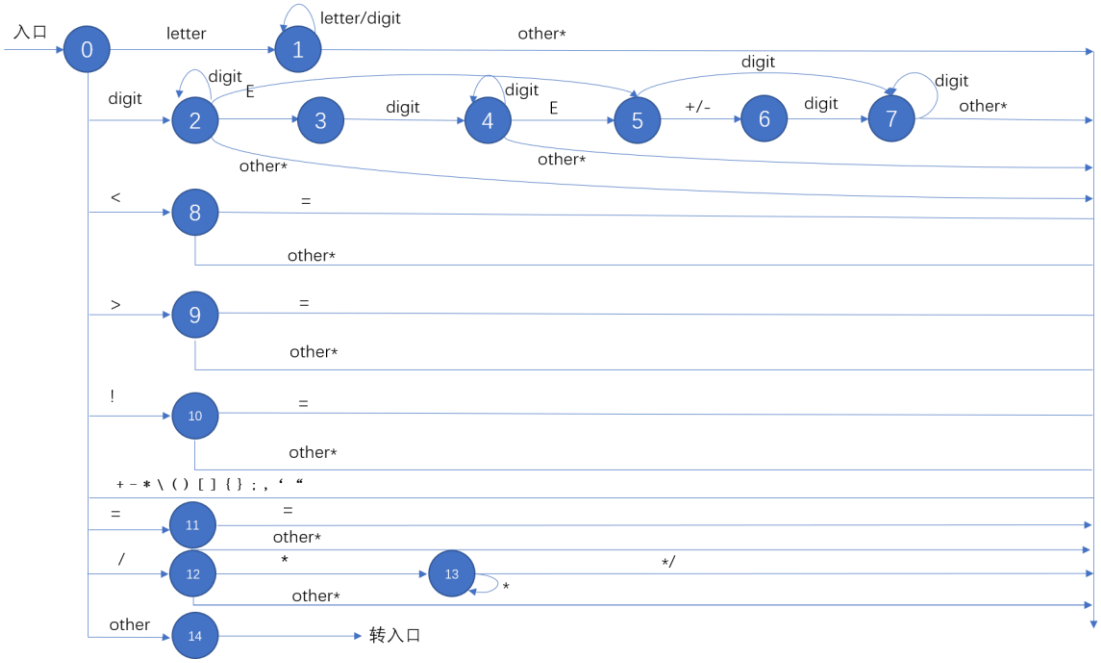
7) 标点符号的文法

```
symbol->( | ) | [ | ] | { | } | ; | , | \ | ’ | “
```

8) 注释头符号的文法

```
note->/star
star->* | /
```

根据(二)中的文法，我们可以构造出如下的状态转移图：



状态 0 是初始状态，若此时读入的字符是字母，则转换到状态 1，进入标识符识别过程，如果此时读入的字符是数字，则转换到状态 2，进入无符号数识别过程，在状态 2 时，若读入字符是”.”，则转入状态 3，而若读入其他，则转到状态 14 出口，状态 3 读入字符为数字时，转入状态 4，此时读入“e”或“E”时，转入状态 5，若读入其他则转入状态 14 出口，状态 5 读入数字则转入状态 7，读入+|-则转入状态 6，状态 6 读入数字转入状态 7，状态 7 若接着读入数字则循环状态 7，否则转入状态 14 出口。

初始时读入字符是“<”，转入状态 8

初始时读入字符是“>”，转入状态 9

初始时读入字符是“!”，转入状态 10

初始时读入字符是“=”，转入状态 11

初始时读入字符是“/”，转入状态 12,状态 12 再次读入字符“*”将转入状态 13，状态 13 为注释处理状态

状态 14 为错误处理状态

三、程序设计

(一) 输出形式

本词法分析程序使用下表所给出的翻译表，在分离出一个单词之后，对识别出的记号以二元式的形式加以输出，其形式为<记号，属性>

正则表达式	记号	属性
32 个关键字	关键字本身	Keyword
id	id	符号表入口指针
num	num	常数值

(	(	symbol
)	)	symbol
[	[	symbol
]	]	symbol
{	{	symbol
}	}	symbol
,	,	symbol
;	;	symbol
‘	‘	symbol
“	“	symbol
\	\	symbol
<	relop	LT
>	relop	GT
<=	relop	LE
>=	relop	GE
==	relop	EQ
!=	relop	NE
=	=	assign_op
+	+	--
-	-	--
*	*	--
:	:	--
&	&	--
%	%	--

（二） 全局变量和过程

1） kw 类

```
class kw
{
public:
    string re_exp;//正规表达式
    string mark;//记号
    string nature;//属性
    kw()//无参数构造函数
    {
        re_exp = "";
        mark = "";
        nature = "";
    }
    kw(string a, string b, string c)//有参数构造函数
    {
        re_exp = a;
        mark = b;
        nature = c;
    }
};
```

kw 类用于定义字符串，kw 类中有三个参数，re_exp 代表正则表达式，mark 代表记号，nature 代表属性，kw 类中定义了一个默认构造函数，一个带参构造函数，根据上述输出形式中定义的表，可给出每个识别字符串对应的 kw 类实例，程序伊始定义了 C 语言中的 32 个关键字：

```
kw kw_ex[32] = { kw("auto", "auto", "Z"),
                 kw("short", "stort", "Z"),
                 kw("int", "int", "Z"),
                 kw("long", "long", "Z"),
                 kw("float", "float", "Z"),
                 kw("double", "double", "Z"),
                 kw("char", "char", "Z"),
                 kw("struct", "struct", "Z"),
                 kw("union", "union", "Z"),
                 kw("enum", "enum", "Z"),
                 kw("typedef", "typedef", "Z"),
                 kw("const", "const", "Z"),
                 kw("unsigned", "unsigned", "Z"),
                 kw("signed", "signed", "Z"),
                 kw("extern", "extern", "Z"),
                 kw("register", "register", "Z"),
                 kw("static", "static", "Z"),
                 kw("volatile", "volatile", "Z"),
```

```
        kw("void", "void", "Z"),
        kw("if", "if", "Z"),
        kw("else", "else", "Z"),
        kw("switch", "switch", "Z"),
        kw("case", "case", "Z"),
        kw("for", "for", "Z"),
        kw("do", "do", "Z"),
        kw("while", "while", "Z"),
        kw("goto", "goto", "Z"),
        kw("continue", "continue", "Z"),
        kw("break", "break", "Z"),
        kw("default", "default", "Z"),
        kw("sizeof", "sizeof", "Z"),
        kw("return", "return", "Z")
    };
```

32 个关键字的记号 mark 值均为原关键字，属性 nature 值为 Z，另外定义了 kw 类的实例数组 kw tag[100]用于存储用于自定义的标识符，int 型变量 i_tag 用于表示用户自定义标识符的指针，当识别到新定义的标识符时，将标识符添加进数组，并将 i_tag 后移。

2) fw 类

```
class fw
{
public:
    int row;
    int line;
    fw()
    {
        row = 0;
        line = 0;
    }
};//指针 forward 类
```

fw 是 forward 的简写，代表字符指针，向前指针，fw 中定义了两个变量 row，line。分别代表当前指针在源文件中所处的行数和列数，fw 默认构造函数将 row 和 line 都置 0，在本次实验中，我们定义了 fw 的两个实例 f 和 lf，分别代表当前字符的指针位置和之前指针的字符位置(用于回退)

3) ch 字符串数组

```
string ch[11] = { "{", "[", "(", ")", "]", "}", ";", ",", "\'", "\"","\\"};
```

ch 数组是所有标点符号数组的集合，对于标点符号正则式，其记号定义为本身，属性定义为 symbol

4) 错误数 err， 总行数 all_line，单词个数 word_num，字符总数 all_ch

```
int err = 0;//错误数
int all_line = 0;//语句行数
int word_num = 0;//单词个数
```

```
int all_ch = 0;//字符总数
```

5) 其他变量

```
char C;
string token;
int state = 0;//当前状态指示
string iskey = "";
int line = 0;
int row = 0;//指针
```

C为字符变量，存放当前读入的字符

token为字符数组，存放当前正在识别的单词字符串

state为整型变量，为当前状态指示

iskey为字符串型变量，若值为“-1”则表示识别出的单词是用户自定义标识符，否则为所定义的关键字之一，其值为关键字的记号

line为整型变量，当前指针所指向的字符所在列数

row为整型变量，当前指针所指向的字符所在行数

6) 文件读入函数 readTxt()

```
void readTxt(string file)
{
    ifstream infile;
    infile.open(file);   //将文件流对象与文件连接起来
    assert(infile.is_open());   //若失败,则输出错误消息,并终止程序运行
    string s;
    int i = 0;
    while (getline(infile, s))
    {
        code[i] = s;
        all_ch += s.length();
        code[i] += '\n';
        i++;
        std::cout << s << endl;
    }
    all_line = i;
    infile.close();             //关闭文件输入流
    code[i] = "#";
    for (auto a : ch) {
        cout << a << " ";
    }
    cout << endl;
}
```

用于从指定的txt文件中读入代码字符串并处理，在这里我们按行读入并将每一行中的所有字符都依次存入数组code中，每一行末尾添加’\n’，用于字符指针的更新和处

理，统计当前代码文件所有行数，再整个code数组后中添加“#”代表整个代码的结束。

7） 字符读入函数 get_char()

```
void get_char()
{
    rf.row = f.row;
    rf.line = f.line;
    if (code[f.row][f.line] != '\n')
    {
        C = code[f.row][f.line];
        f.line++;
    }
    else
    {
        f.row++;
        f.line = 0;
        C = code[f.row][f.line];
        f.line++;
    }
}//根据向前指针forward的指示从输入缓冲区中读出一个字符并放入C中
```

在本实验中的get_char()主要用于从已经存储好的字符数组code中读入字符并将其放入C中，当未读取到‘\n’时，将当前字符指针f的line值加一，row值不变。当读取到‘\n’时，将f.row值加一，f.line值置为0，取出字符C

8） 空格跳过函数 get_nbc()

```
void get_nbc()
{
    while ((C == ' ') || (C == '\t'))
    {
        get_char();
    }
}//检查空格和\t
```

每次调用时，检查C中的字符是否为空格，若是，则反复调用过程get_char(),直到C中进入一个非空字符为止

9） 字符拼接函数 cat

```
void cat()
{
    token += C;
}//将C中字符拼接在token中的字符串后面
```

10) 判断字母函数 letter()

```
bool letter()
{
    if (((C >= 'a') && (C <= 'z')) || ((C >= 'A') && (C <= 'Z')) || (C == '_'))//字
```

符

```
            return true;
        else
            return false;
    }//判断字母或下划线
```

布尔函数，判断C中的字符是否为字母，若是则返回true，否则返回false

11) 判断数字函数 digit()

```
    bool digit()
    {
        if (((C >= '0') && (C <= '9')))
            return true;
        else
            return false;
    }//判断数字
```

布尔函数，判断C中的字符是否为字母，若是则返回true，否则返回false

12) 指针回退函数 restract()

```
    void restract()
    {
        f.row = rf.row;
        f.line = rf.line;
    }//指针forward后退一个字符
```

向前指针forward(f)后退一个字符

13) 关键字匹配函数 reverse()

```
    string reserve()
    {
        for (int i = 0; i < 32; i++)
        {
            if (token == kw_ex[i].re_exp)
                return kw_ex[i].mark;
        }
        return "-1";
    }
    //根据token中的单词查关键字表，若token中的单词是关键字，则返回值该关键字的记号，否则返回-1
```

14) 自定义标识符插入函数 table_insert()

```
    int table_insert(){
        for (int i = 0; tag[i].re_exp != ""; i++){
            if (token == tag[i].re_exp){
                return i;
            }
```

```
        }
        tag[i_tag].re_exp = token;
        tag[i_tag].mark = "id";
        tag[i_tag].nature = i_tag;
        i_tag++;
        return (i_tag - 1);
    }//将识别出来的用户自定义标识符插入符号表，返回该单词在符号表中的位置指针
```

将传入的字符串进行识别，若已经在符号表中找到，则返回自定义字符在符号表中的入口地址，否则将字符串插入符号表中(新增一个自定义标识符),并将该位置返回

15) 输出函数 ret()

```
    void ret(string A, string B)
    {
        int flag = -1;
        for (int i = 0; i < 11; i++)
            if (ch[i] == A)
                flag = i;
        if (flag >= 0)
            std::cout << "<" << A << " , " << "symbol" << ">" << endl;
        else if (A == "relop")
            std::cout << "<" << "relop" << " , " << B << ">" << endl;
        else if (B == "Z")
            std::cout << "<" << A << " , " << "keyword" << ">" << endl;
        else
            std::cout << "<" << A << " , " << B << ">" << endl;
    }//输出

    void ret(string A, int B)
    {
        std::cout << "<" << A << " , " << B << ">" << endl;
    }

    void ret(string A, float B)
    {
        std::cout << "<" << A << " , " << B << ">" << endl;
    }
```

根据传入参数的不同，ret函数有三种不同的重载，分别对应三种不用的输出形式

16) 错误输出函数 error()

```
    void error(int row, int line)
    {
        std::cout << "第" << row + 1 << "行" << "第" << line + 1 << "列出现错误" <<
endl;
        err++;
```

}//对发现的错误进行相应的处理

输出错误所在的指针位置

（三） 具体状态分析

状态0是初始状态，将token置空，将变量row和line置为当前字符所在位置指针f.row和f.line，通过get_char()函数读入字符并赋值给C，通过get_nbc()跳过所有的空格和\t，之后根据C的值更新状态值state

1） 状态 1

```
case 1:
    //标识符状态
    cat();
    get_char();
    if (letter() || digit())
        state = 1;
    else
    {
        restract();
        state = 0;
        iskey = reserve();
        word_num++;
        if (iskey != "-1")
            ret(iskey, "Z");
        else {
            int identry = table_insert(); //返回标识符在符号表的入口指针
            ret("id", identry);
        }
    }
    break;
```

当状态0读入的字符C为字母时，当前状态转换到状态1，状态1一直读入字符并将其拼接到token后，知道读入的字符不为字母或数字其一，此时转入处理。首先将字符指针回退，状态回退0，之后置iskey的值为reverse(),若iskey的值为”-1”表明当前识别到的标识符为关键字，通过ret()函数输出，否则，说明当前识别到的标识符为用户自定义标识符，执行table_insert()函数，返回符号表的入口指针地址或插入新的自定义标识符进入表中，通过ret()函数输出。

2） 状态2

```
case 2:
    //常数状态
    cat();
    get_char();
    switch (C) {
    case '0': case '1':
    case '2': case '3':
```

```
            case '4': case '5':
            case '6': case '7':
            case '8': case '9':
                state = 2;
                break;
            case '.':
                state = 3;
                break;
            case 'E':
            case 'e':
                state = 5;
                break;
            default:
                if (((C >= 'a') && (C <= 'z')) || ((C >= 'A') && (C <= 'Z')) || (C ==
'_'))
                {
                    while ((C != ' ') && (C != ',') && (C != ';'))
                        get_char();
                    state = 14;
                    restract();
                }
                else
                {
                    word_num++;
                    restract();
                    state = 0;
                    ret("num", stoi(token));//返回整数
                }
                break;
            }
            break;
```

当状态0读入的字符C为数字时，当前状态转换到状态2，之后接着读入字符，若字符一直为数字则一直循环状态2，若此时读入字符’.’，说明可能是小数，转入状态3，若此时读入字符’e’或’E’，说明可能是科学计数法表示数，转入状态5，当读入字符为字母或下划线时，此时词法出错，返回状态14，输出错误信息，否则说明仅有一个整数，指针回退，状态置0，通过ret()函数输出该整数

3） 状态3

```
        case 3:
            //小数点状态
            cat();
            get_char();
            if (digit())
                state = 4;
```

```
            else
            {
                error(row, line);
                state = 0;
            }
            break;
```

当状态2读取到的字符为小数点时，将转入状态3，处理小数点情况，状态3接着读取下一个字符，若下一个字符为数字将转换到状态4，否则该词法出错，转入错误输出语句，状态归0

4） 状态4

```
        case 4:
            //小数状态
            cat();
            get_char();
            switch (C) {
            case '0':case '1':
            case '2':case '3':
            case '4':case '5':
            case '6':case '7':
            case '8':case '9':
                state = 4;
                break;
            case 'E':
            case 'e':
                state = 5;
                break;
            default:
                if (((C >= 'a') && (C <= 'z')) || ((C >= 'A') && (C <= 'Z')) || (C ==
'_'))
                {
                    while ((C != ' ') && (C != ',') && (C != ';'))
                        get_char();
                    state = 14;
                    restract();
                }
                else
                {
                    word_num++;
                    restract();
                    state = 0;
                    ret("num", stof(token));
                }
                break;
```

```
    }
    break;
```

当状态3读取的下一位字符依旧为数字时，将转入状态4。状态4接着读入字符，若字符为数字，将一直处于状态4，读入字符为E或e时，将转入状态5处理指数，若读入字符为字母或下划线时，此时词法出错，转入状态14处理错误信息，否则为其他字符，说明识别成功，token此时为一个完整的小数，回退指针和状态归位，输出该小数

5） 状态5

```
case 5:
    //指数状态
    cat();
    get_char();
    switch (C) {
    case '0':case '1':
    case '2':case '3':
    case '4':case '5':
    case '6':case '7':
    case '8':case '9':
        state = 7;
        break;
    case '+':
    case '-':
        state = 6;
        break;
    default:
        restract();
        error(row, line);
        state = 0;
        break;
    }
    break;
```

状态5为指数处理状态，进入状态5时说明上一个读入字符为e或E，此时说明该数字使用指数法表示，此时根据接下来读入的字符再次进行分支处理，若读入字符为数字，则转入状态7中处理该数字以及后续字符，此时词法形式可能为AeXXXX…，状态7将后续字符串转为字符串并输出；若读入字符为“+”，则说明后续指数为正数，读入字符为“-”，说明后续指数为负数，两者均转入状态6进一步处理；若读入其他字符，则此时词法出错，指针回退，输出错误信息后，状态归0。

6） 状态6

```
case 6:
    cat();
    get_char();
    if (digit())
        state = 7;
```

```
            else
            {
                restract();
                error(row, line);
                state = 0;
            }
            break;
```

状态6判断下一步读入数是否为数字，若是则转入状态7取出该数字，否则输出错误信息

7） 状态7

```
        case 7:
            cat();
            get_char();
            if (digit())
                state = 7;
            else
            {
                if (((C >= 'a') && (C <= 'z')) || ((C >= 'A') && (C <= 'Z')) || (C == '_'))
                {
                    while ((C != ' ') && (C != ',') && (C != ';'))
                        get_char();
                    state = 14;
                    restract();
                }
                else
                {
                    word_num++;
                    restract();
                    state = 0;
                    ret("num", stof(token));//返回无符号数
                }
            }
            break;
```

状态7经过前面的层层铺垫，确保了可以用来取出该指数的环境，状态7不停读入字符，若字符为数字，则状态不变，将字符串在token后；若读入的字符为字母或下划线时，此处词法出错，状态置位14，输出错误信息；否则说明指数已读取完毕，将token转换为数字后输出。

8） 状态8

```
        case 8:
            //'<'状态
            cat();
```

```
get_char();
switch (C) {
case '=':
    cat();
    state = 0;
    ret("relop", "LE");
    break;
default:
    restract();
    cat();
    state = 0;
    ret("relop", "LT");
    break;
}
break;
```

状态8为状态0读入字符为“<”后转入的状态，状态8首先读入一个字符，若该字符为“=”，说明此时的运算符为小于等于，输出运算符信息；若为其他字符，则说明运算符仅为小于，则指针回退，状态归0后，输出运算符信息。

9） 状态9

```
case 9:
    //'>'状态
    cat();
    get_char();
    if (C == '='){
        cat();
        state = 0;
        ret("relop", "GE");
    }
    else{
        restract();
        state = 0;
        ret("relop", "GT");
    }
    break;
```

状态9为状态0读入字符为“>”后转入的状态，状态9首先读入一个字符，若该字符为“=”，说明此时的运算符为大于等于，输出运算符信息；若为其他字符，则说明运算符仅为大于，则指针回退，状态归0后，输出运算符信息。

10）状态10

```
case 10:
    //'!'状态
    cat();
    get_char();
```

```
if (C == '=') {
    cat();
    state = 0;
    ret("relop", "NE");
}
else {
    restract();
    state = 0;
    ret("!", "--");
}
break;
```

状态10为状态0读入字符为“!”后转入的状态，状态10首先读入一个字符，若该字符为“=”，说明此时的运算符为不等于，输出运算符信息；若为其他字符，则说明名该“!”并不代表运算符，则指针回退，状态归0后，输出字符信息。

11) 状态11

```
case 11:
    // '='状态
    cat();
    get_char();
    if (C == '=') {
        cat();
        state = 0;
        ret("relop", "EQ");
    }
    else {
        restract();
        state = 0;
        cout << "<=" << " , " << "assign_op>" << endl;
    }
    break;
```

状态11为状态0读入字符为“=”后转入的状态，状态11首先读入一个字符，若该字符为“=”，说明此时运算符为等于，输出运算符信息；若为其他字符，则说明名该“=”并不代表运算符，而代表的是分配符，指针回退，状态归0后，输出assign_op。

12) 状态12

```
case 12:
    //'/'状态
    cat();
    get_char();
    if (C == '*') {
        cout << "</*" << " , " << "annotation_start>" << endl;
        state = 13; //设置注释处理状态
    }
```

```
else if (C == '/') {
    cout << "//" << " , " << "annotation_start>" << endl;
    get_char();
    while ((f.row-rf.row)!=1)
        get_char();
    state = 0;
}
else{
    restract();
    state = 0;
    std::cout << "</ , div_op>" << endl;
}
break;
```

状态12为状态0读入字符为“/”后转入的状态，状态12首先读入一个字符，若该字符为“*”，说明此时出现了注释信息的开始标志，输出注释开始标志信息后，将状态置为13用以处理注释信息；若该字符为“/”，则说明出现了行注释的开始标志，将该行后所有字符均跳过，之后状态置0接着处理下一行；若为其他字符，则说明名该“/”仅代表分隔符，则指针回退，状态归0后，输出分隔符信息。

13) 状态13

```
case 13:
    //注释处理状态
    get_char();
    while (C != '*'&&C!='#')
        get_char();
        if (C == '#') {
            restract();
        }
    get_char();
    if (C == '/') {
        cout << "<*/" << " , " << "annotation_end>" << endl;
        state = 0;
    }
    else if (C == '#') {
        state = 14;
    }
    else state = 13;
    break;
```

状态13为状态12读入字符为“*”后转入的状态，状态13的处理相对复杂，状态13将一直读取字符，知道读取到的字符“*”或结束符“#”，若读取到结束符，说明该初始注释后所有字符均为注释后的内容，此时结束词法分析即可；若读取到的字符为“*”,则再次读入字符，若该字符为“/”，说明出现了注释结束标志，输出注释结束信息后状态归为0，若为“#”，说明读取到结束符，结束词法分析，若为其他字符，则状态归位13，接着循环该过程。

14）状态14

```
case 14:
    //错误处理状态
    error(row, line);
    state = 0;
    break;
```

状态14为错误处理状态，当词法出错时将转入该状态，输出错误信息并将状态归0

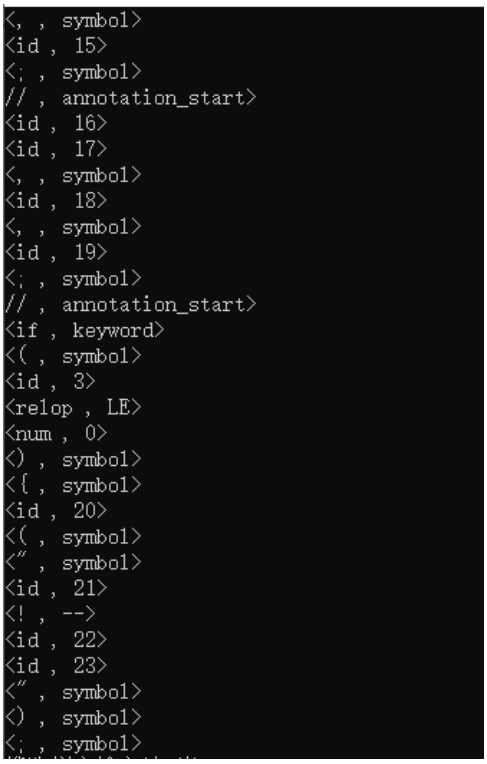
四、程序运行与结果分析

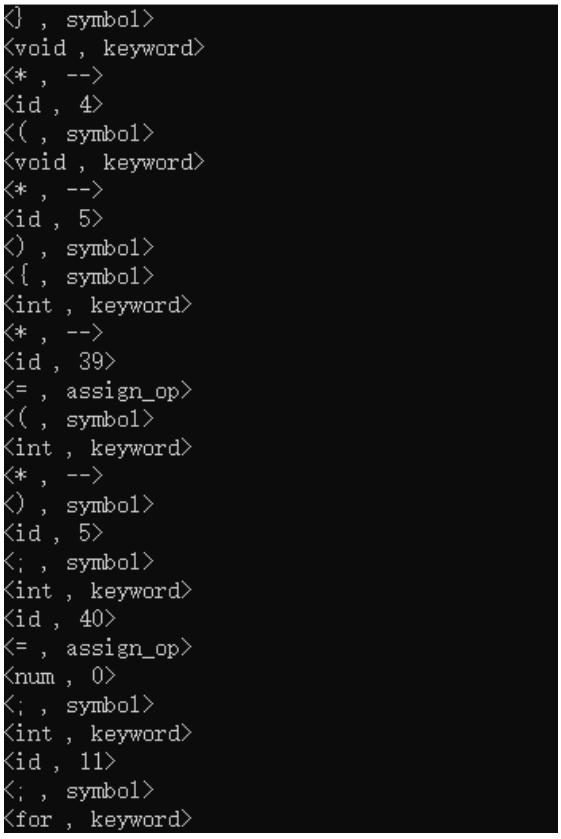
（一） 常规运行

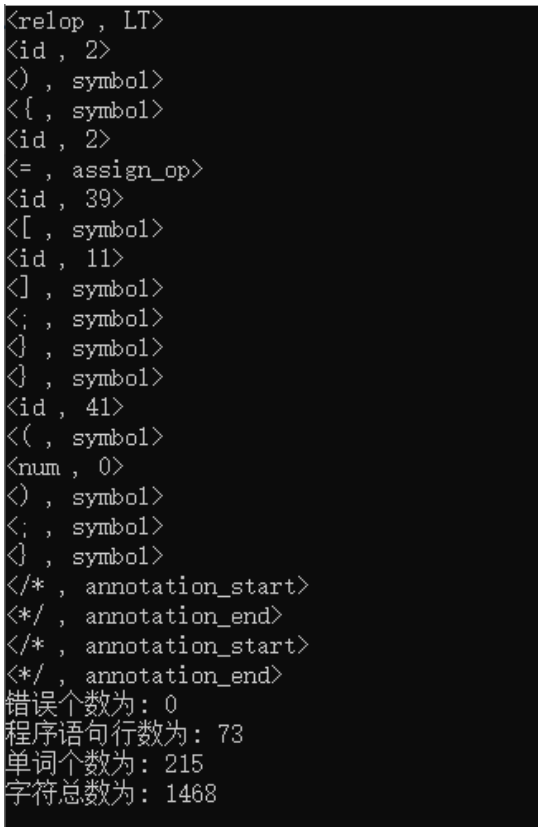
对一个词法正确的程序文本进行分析：

```
int average,max,min;
int count;
void *average_thread(void *param);
void *max_thread(void *param);
void *min_thread(void *param);
int main(int argc,char **argv){
        int i;
        count=argc-1;
        pthread_t Thread1,Thread2,Thread3;//this is a zhushi
        pthread_attr_t attr1,attr2,attr3;//this is a zhushi

        if(count<=0){
                printf("error!no input");
                return -1;
        }

        int *numbers=(int *)malloc(sizeof(int)*count);
        if(numbers!=NULL){
                printf("error!malloc error");
                return -1;
        }
        for(i=0;i<count;i++){
                numbers[i]=atoi(argv[i+1]);
        }
        pthread_attr_init(&attr1);
        pthread_attr_init(&attr2);
        pthread_attr_init(&attr3);
        pthread_create(&Thread1,&attr1,average_thread,numbers);
        pthread_join(Thread1,NULL);
        pthread_create(&Thread2,&attr2,max_thread,numbers);
        pthread_join(Thread2,NULL);
        pthread_create(&Thread3,&attr3,min_thread,numbers);
        pthread_join(Thread3,NULL);
        printf("The average value is %d\n",average);
        printf("The minimun value is %d\n",min);
        printf("The maximun value is %d\n",max);
        free(numbers);
        return 0;
}
void *average_thread(void *param){
        int *num=(int *)param;
        int sum=0;
        int i;
        for(i=0;i<count;i++){
                sum+=num[i];
        }
        average=sum/count;
        pthread_exit(0);
}
void *max_thread(void *param){
        int *num=(int *)param;
        max=num[0];
        int i;
        for(i=1;i<count;i++){
                if(num[i]>max){
                        max=num[i];
                }
        }
        pthread_exit(0);
}
void *min_thread(void *param){
        int *num=(int *)param;
        min=num[0];
        int i;
        for(i=1;i<count;i++){
                if(num[i]<min){
                        min=num[i];
                }
        }
        pthread_exit(0);
}
/*fjaiojffj*/
/*fawgGe*/
```

运行结果部分截图：







该程序并没有词法错误，并且每个字符或标识符均正确识别，识别并跳过了源程序中的注释，可以统计源程序中的语句行数，各类单词的个数，以及字符总数，并输出统计结果，完成了词法分析器的功能

（二） 错误检查与处理

对一个词法存在错误的程序文本进行词法分析

```
int main()
{
        int m = 2, z, k;
        z = 2*m;
        k = m*z;
        int c = 1e-1;
        int b = 888888.8912e2;
        int 6d;
        int 6.6dd;
/*bnmvbnmbn
Ghjghbnm
ghj*/
}
```

该程序中红框标注处词法存在问题

程序运行结果：

```
<id , 2>
<; , symbol>
<int , keyword>
<id , 4>
<= , assign_op>
<num , 0.1>
<; , symbol>
<int , keyword>
<id , 5>
<= , assign_op>
<num , 8.88889e+07>
<; , symbol>
<int , keyword>
第8行第5列出现错误
<; , symbol>
<int , keyword>
第9行第5列出现错误
<; , symbol>
</* , annotation_start>
<*/ , annotation_end>
<} , symbol>
错误个数为: 2
程序语句行数为: 13
单词个数为: 21
字符总数为: 127
```

当程序识别到6d时，发现了词法错误并输出了错误所在位置：第8行第5列，当程序识别到6.6dd时，发现了词法错误并输出了错误所在位置，第9行第5列，程序在遇到词法错误时并没有停止，而依旧接着分析错误语句后面的词法，保证了程序的正常执行。

《词法分析程序》实验报告—LEX 版本

实验目的和内容与C语言编写版本相同，本次使用LEX仅实现了简易的词法分析功能，相比C编写的版本来说，词法分析较为简洁

一、程序分析

简易的将C词法定义为以下几种类型：

- 关键字：囊括C语言标准中的32个关键字表(具体可见C语言版本报告)
- 运算符：包括"&="|"^="|"|="|"<<="|">>="|"*="|"/="|"%="|"+="|"-="|"="|"?:"|"||"|"&&"|"|"|"^"|"&"|"=="|"!="|">"|">="|"<"|"<="|"<<"|">>"|"+"|"-"|"*"|"/"|"%"|".*"|"->*"|"&"|"+"|"-"|"++"|"--"|"->"|"::" {等
- 界符：包括"{"|"}"|"("|")"|"#"|","|":"|";"|"."|"\"" {等
- 标识符：以字母开头的、后跟字母或数字组成的字符串
- 其他：空格或\t或换行符等

二、程序运行

编译与执行：

```
hexing@ubuntu:~$ vi lex.l
hexing@ubuntu:~$ flex lex.l
hexing@ubuntu:~$ gcc lex.yy.c
hexing@ubuntu:~$ ./a.out<test
```

测试用例：

```
int main()
{
        int m = 2, z, k;
        z = 2*m;
        k = m*z;
        int c = 1e-1;
        int b = 888888.8912e2;
        int 6d;
        int 6.6dd;
//bnmvbnmbnGhjghbnmghj
}
~
```

输出结果：

```
hexing@ubuntu:~$ ./a.out<test
 (关键字, int)
 (其它(blank or enter),  )
 (标识符, main)
 (界符, ()
 (界符, ))

 (界符, {)

 (关键字, int)
 (其它(blank or enter),  )
 (标识符, m)
 (其它(blank or enter),  )
 (运算符, =)
 (其它(blank or enter),  )
 (数字, 2)
 (界符, ,)
 (其它(blank or enter),  )
 (标识符, z)
 (界符, ,)
 (其它(blank or enter),  )
 (标识符, k)
 (界符, ;)

 (标识符, z)
 (其它(blank or enter),  )
 (运算符, =)
 (其它(blank or enter),  )
 (数字, 2)
 (运算符, *)
 (标识符, m)
 (界符, ;)

 (标识符, k)
 (其它(blank or enter),  )
 (运算符, =)
 (其它(blank or enter),  )
 (标识符, m)
 (运算符, *)
 (标识符, z)
 (界符, ;)

 (关键字, int)
 (其它(blank or enter),  )
 (标识符, c)
 (其它(blank or enter),  )
 (运算符, =)
 (其它(blank or enter),  )
 (数字, 1)
 (标识符, e)
 (数字, -1)
 (界符, ;)

 (关键字, int)
 (其它(blank or enter),  )
 (标识符, b)
 (其它(blank or enter),  )
 (运算符, =)
 (其它(blank or enter),  )
 (数字, 888888.8912)
 (标识符, e2)
 (界符, ;)

 (关键字, int)
 (其它(blank or enter),  )
 (数字, 6)
 (标识符, d)
 (界符, ;)

 (关键字, int)
 (其它(blank or enter),  )
 (数字, 6.6)
 (标识符, dd)
 (界符, ;)

 (注释, //bnmvbnmbnGhjghbnmghj)

 (界符, })

hexing@ubuntu:~$ vi test
hexing@ubuntu:~$ vi test
```

三、LEX 程序源码

```
%{
#include "stdio.h"
%}
%%
asm|do|if|return|typedef|auto|double|inline|short|typeid|bool|try|include|long|s
izeof|union|case|enum|mutable|static|unsigned|long|sizeof|union|case|enum|mutabl
e|static|unsigned|catch|explicit|namespace|using|char|export|int|signed|break|el
se|new|struct|virtual|class|extern|operator|switch|void|const|false|private|temp
late|volatile|float|protected|this|continue|for|public|throw|while|default|frien
d|register|true|delete|goto|try|include|std|iomanip|setw|setprecision|endl|setio
sflags|ios {
     printf(" (关键字, %s)\n" ,yytext);
}
[+-]?([0-9]*|0|([0-9]*\.[0-9]*)) {
printf(" (数字, %s)\n" ,yytext);
}
"&="|"^="|"|="|"<<="|">>="|"*="|"/="|"%="|"+="|"-="|"="|"?:"|"||"|"&&"|"|"|"^"|"
&"|"=="|"!="|">"|">="|"<"|"<="|"<<"|">>"|"+"|"-"|"*"|"/"|"%"|".*"|"->*"|"&"|"+"|
"-"|"++"|"--"|"->"|"::" {
printf(" (运算符, %s)\n" ,yytext);
}
"{"|"}"|"("|")"|"#"|","|":"|";"|"."|"\"" {
printf(" (界符, %s)\n" ,yytext);
}
'[^'\n]*'|\"[^\"]*\" {
printf(" (字符串, %s)\n" ,yytext);
}
\/\*[\s\S]*\*\/|\/\/.* {
printf(" (注释, %s)\n" ,yytext);
}
[A-Za-z]([A-Za-z]|[0-9]|_)* {
printf(" (标识符, %s)\n" ,yytext);
}

[\t]+  {}
[\n];
. {printf(" (其它(blank or enter), %s)\n" ,yytext);}
%%
int yywrap(){
return 1;
}
void main(){
yylex();
}
```

```
% {
#include "stdio.h"
    %}
%%
asm | do | if | return | typedef | auto | double | inline | short | typeid | bool | try
| include | long | sizeof | union | case | enum | mutable | static | unsigned | long |
sizeof | union | case | enum | mutable | static | unsigned | catch | explicit |
namespace | using | char | export | int | signed | break | else | new | struct |
virtual | class | extern | operator|switch | void | const | false | private | template
| volatile | float | protected | this | continue | for | public | throw | while |
default | friend | register | true | delete | goto | try | include | std | iomanip |
setw | setprecision | endl | setiosflags | ios{
     printf(" (关键字, %s)\n" ,yytext);
```

```
}
[+-] ? ([0 - 9] * | 0 | ([0 - 9] * \.[0 - 9] *)) {
    printf(" (数字, %s)\n", yytext);
}
"&=" | "^=" | "|=" | "<<=" | ">>=" | "*=" | "/=" | "%=" | "+=" | "-=" | "=" | "?:" |
"||" | "&&" | "|" | "^" | "&" | "==" | "!=" | ">" | ">=" | "<" | "<=" | "<<" | ">>" |
"+" | "-" | "*" | "/" | "%" | ".*" | "->*" | "&" | "+" | "-" | "++" | "--" | "->" |
"::" {
    printf(" (运算符, %s)\n", yytext);
}
"{" | "}" | "(" | ")" | "#" | "," | ":" | ";" | "." | "\"" {
    printf(" (界符, %s)\n", yytext);
}
'[^'\n] * '|\"[^\"]*\" {
printf(" (字符串, %s)\n", yytext);
}
\ / \ * [\s\S] * \ * \/|\ / \ / .* {
    printf(" (注释, %s)\n", yytext);
}
[A - Za - z]([A - Za - z] | [0 - 9] | _)* {
    printf(" (标识符, %s)\n", yytext);
}

[\t] + {}
[\n];
. {printf(" (其它(blank or enter), %s)\n", yytext); }
%%
int yywrap() {
    return 1;
}
void main() {
    yylex();
}
```

实验心得

通过此次词法分析器的实验，我编译原理第3章节词法分析的过程有了更加全面的了解和系统的学习，本次实验让我受益匪浅。第一个方面，对于问题解决能力的提升，在一开始的编写过程中，我遇到了很多逻辑上想不通的地方，对于庞大的词法分析一时不知该从何下手。第二个方面，对于我个人能力的提升，大到整个编译原理学习，小到一个函数的设计与调试，都让我有了更加深刻的理解。其中有许多前期工作以及做的过程中遇到的很多程序方面的问题是需要注意的，都需要静下心来慢慢的看和上网查资料，只有这样，我们才能够在做的过程中变得更加得心应手，从实验中得到的收获也更大。本次实验让我收获最大的便是对于词法分析过程的理解和贯彻，以及模块的整个设计过程的理解，同时也让我对一个程序的词法分析流程有了更加系统全面的认知，让书本上虚无缥缈的话语变得活灵活现，在这一部分我也耗费了最多的时间，对于实验内容的理解让我对书本上晦涩难懂的知识有了更鲜活的记忆和理解。