编译原理 词法分析 实验报告

姓名：陈俊卉　　班级：2020219111　　学号：2020212256

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回复，使词法分析可以继续进行，对源程序进行一次扫描，即可检查并报告源程序中存在的所有词法错误。
实现要求：可以选择一下两种方案中的一种实现。
方法1：采用编程语言，手工编写词法分析程序；
方法2：基于LEX，自动生成词法分析程序。

本次实验采用方法1、2实现，使用的编程语言为C++。

二、实验环境

- windows10系统
- Visual Stdio Code
- C++ 17

三、方法一原理分析

1、 词法分析程序的功能

词法分析是编译过程的第一步，主要任务是从左到右逐个字符对源程序进行扫描，按照源语言的词法规则识别出单词符号，将识别出来的标识符存入符号表中，并产生用于语法分析的记号序列。其主要过程如下：

- 扫描源程序的字符流；
- 根据目标语言的各种词法规则识别出程序中各种单词、符号的种类，跳过注释和空格；
- 对单词、符号进行检查，若无问题则产生记号序列，否则抛出错误；
- 将识别出的序列放入创建符号表中，给出可能出现的错误的行数。

2、 词法分析程序单词的归类

- 变量名(variable)
- 关键字(keyword)　C语言共有32个关键字
- 数字常量(number)
  - 整数
  - 小数
  - 科学计数的数字
- 分隔符(界符)、运算符(operator)

keyword与operator具体如下所示。

```
// 关键字数组
// C语言关键字共有32个
char *key_word[] = {
    // 数据类型关键字
    "char",
    "short",
    "int",
    "long",
    "signed",
    "unsigned",
    "float",
    "double",
    "struct",
    "union",
    "enum",
    "void",
    // 控制语句关键字
    "for",
    "do",
    "while",
    "break",
    "continue",
    "if",
    "else",
    "goto",
    "switch",
    "case",
    "default",
    "return",
```

```
    // 存储类型关键字
    "auto",
    "extern",
    "register",
    "static",
    "typedef",
    // 其他关键字
    "const",
    "sizeof",
    "volatille",
};


// 运算符和界符
char *sign[45] = {
    "[",
    "]",
    "(",
    ")",
    ".",
    "->",
    "-",
    "++",
    "--",
    "*",
    "&",
    "!",
    "~",
    "/",
    "%",
    "+",
    "-",
    "<<",
    ">>",
    ">",
    ">=",
    "<",
    "<=",
    "==",
    "!=",
    "^",
    "|",
    "&&",
    "||",
    "=",
    "/=",
    "*=",
    "%=",
    "+=",
    "-=",
    "<<=",
    ">>=",
    "&=",
    "^=",
    "!=",
    ",",
    ";",
    "{",
```

```
    "}",
    "#",
};

```

3、 设计思路

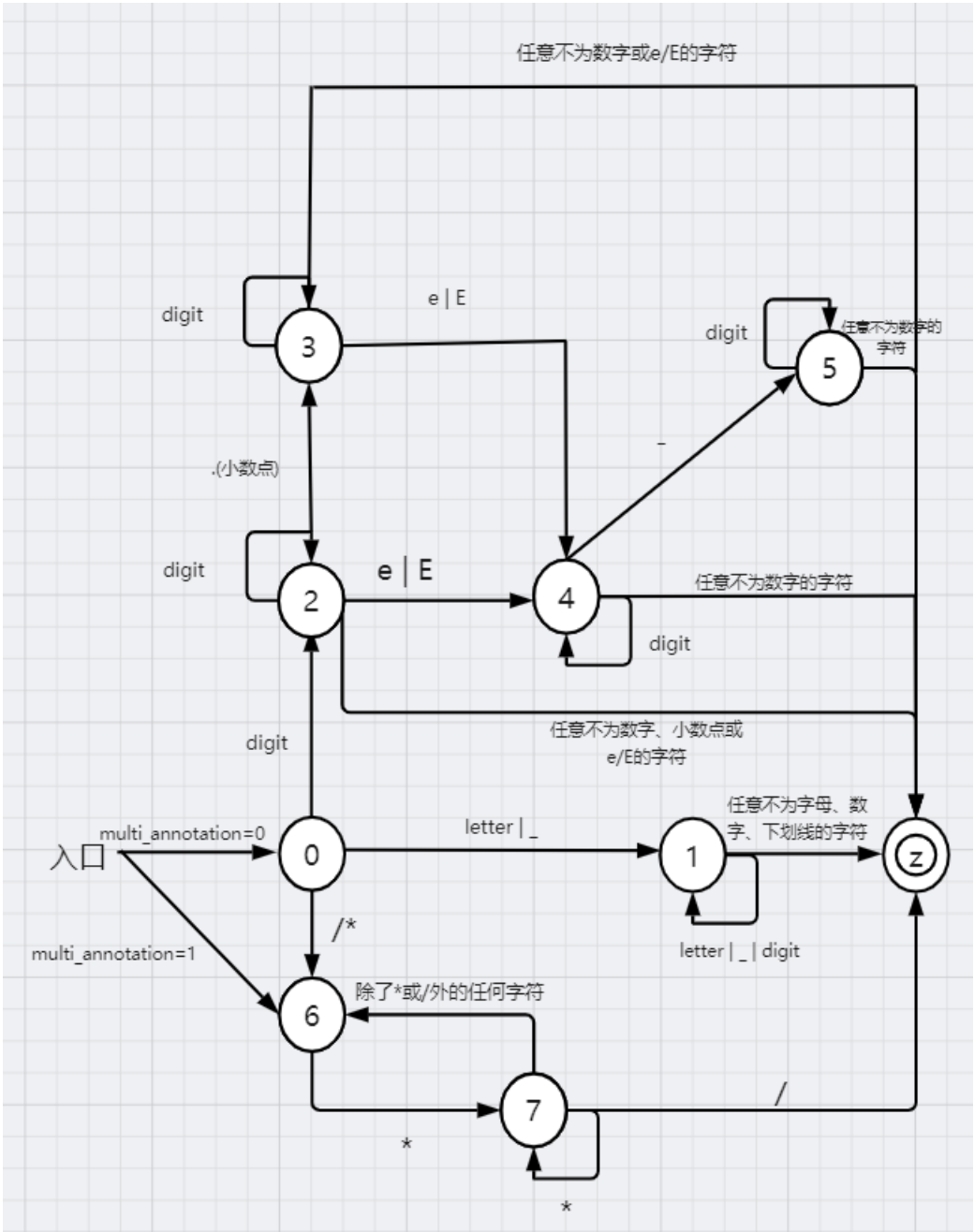
代码模板参考的是课本上的基础代码，并对其加以改进。

输出结构：　line x <type， word>

① 给出描述该语言各种单词符号的词法以及规则

- 该点已经在上文给出。

② 构建状态转换图



解释：

- 大部分匹配分界符和运算符，以及单行注释的过程都在state = 0时已经处理完毕，由于种类过于繁多，所以这里并没有在状态转换图中体现出来，将会在下一部分的代码展示中给出。
- 由于该状态图表示的是提取一个词法记号的过程，而本次实验为了标注行数，笔者采用了一行一行输入词法分析函数的方式进行读取。而多行注释是跨行的，所以我们需要一个全局变量 multi-annotation 表示当前是否为多行注释状态（跨行后由于词法分析函数的重新调用，state会被清空）。
- 各个状态的解释：
    - 0：初始状态。处理第一个字符判断后续状态的过程。在此状态，简单的分界符和运算符，与单行注释已经处理完毕。
    - 1：word状态。直到读取到不符合变量名规范的字符才会停止。
    - 2：整数状态。遇到小数点会进入小数状态，遇到e或E进入科学计数法状态，否则结束。
    - 3：小数状态。在C中，即使小数点后没有数字，也是正确的。遇到e或E进入科学计数法状态。
    - 4：科学计数法状态。遇到负号进入负幂次状态（事实上，负号前面如果是数字，也是没有错误的，只不过负号后面不是幂次而是减法了）
    - 5：科学计数法负幂次状态。
    - 6：多行注释中读取到 /*的状态。
    - 7：多行注释中读取到 /* *的状态。

四、方法一代码展示

一个需要注意的地方：

- 当前版本的C++中，当想要将string直接当作char *使用时会发生强制类型转换，这会引发编译警告：

```
warning: ISO C++ forbids converting a string constant to 'char*' [-Wwrite-strings]
```

可以用c_str()或data()解决该问题。但为了代码的简洁起见，我们仍保留这样的方式。

```
#include<iostream>
#include<string.h>
#include<stdlib.h>
#include<fstream>
using namespace std;

#define MAX_BUFFER_SIZE 10000
#define MAX_STR_SIZE 10000
// 每一行的缓冲区
char buffer[MAX_BUFFER_SIZE];
// 读取文件中被单引号和双引号引住的字符串 或 一串数字 或 变量
char str_in_file[MAX_STR_SIZE];

// 记录各种数据
// 行数
int line_num = 0;
// 字符总数（包括空格）
int sign_num = 0;
// 单词总数（关键字+变量名）
```

```c
int word_num = 0;
// 关键字数目
int keyword_num = 0;
// 变量名数目
int varible_num = 0;
// 数字常量数目
int number_num = 0;
// 分界符和操作符数目
int operate_num = 0;
// 字符串常量数目
int str_num = 0;
// 单行注释数目
int single_ann_num = 0;
// 多行注释数目
int multi_ann_num = 0;


// 多行注释标记
int multi_annotation = 0;


// 关键字数组
// C语言关键字共有32个
char *key_word[] = {
    // 数据类型关键字
    "char",
    "short",
    "int",
    "long",
    "signed",
    "unsigned",
    "float",
    "double",
    "struct",
    "union",
    "enum",
    "void",
    // 控制语句关键字
    "for",
    "do",
    "while",
    "break",
    "continue",
    "if",
    "else",
    "goto",
    "switch",
    "case",
    "default",
    "return",
    // 存储类型关键字
    "auto",
    "extern",
    "register",
    "static",
    "typedef",
    // 其他关键字
    "const",
```

```
    "sizeof",
    "volatille",
};


 // 获取下一个字符的函数，并让forward指针向后移动一位
char get_char(char *&forward){
    char this_char = *forward;
    forward ++;
    sign_num ++;
    return this_char;
}

// 回退一格
void get_backward(char *&forward){
    forward --;
}

/*
    打印结果
    结构:       行数 <word type,  word>
*/
void print(int line_index, char *type, char *str){
    cout << "line " << line_index << " <" << type << ", " << str << ">" << endl;
}

// 是否为数字
bool is_num(char ch){
    if (ch >= '0' && ch <= '9') return true;
    else return false;
}

// 是否为字母
bool is_letter(char ch){
    if ((ch >= 'a' && ch <= 'z' ) || (ch >= 'A' && ch <= 'Z')) return true;
    else return false;
}

// 是否为关键字 一共有32个关键字
bool is_keyword(char *str){
    for (int i = 0; i < 32; i++){
        if(strcmp(str, key_word[i]) == 0){
            // 如果是关键字 返回下标（从1开始）
            return true;
        }
    }
    // 否则返回false
    return false;
}



/*
    词法分析函数，传入forward指针(当前行)和当前行的下标
    该函数参考课本中程序的结构
*/
```

```
void lexical_analysis(int line_index, char *forward){
    // 状态初始化
    int state = 0;

    // 因为分行 所以只可能到状态6
    if (multi_annotation == 1){
        state = 6;
    }

    // 定义当前字符
    char now_char = ' ';
    // 标记字符串位置
    int pos = 0;
    // 如果当前还没有到达该行的结束
    while (now_char != '\0') {
        switch (state)
        {
            case 0:
            {
                // 当前字符
                now_char = get_char(forward);

                // 根据第一个字符进行状态的切换
                switch (now_char) {
                    // 空格跳过
                    case ' ':
                    {
                        pos = 0;
                        break;
                    }


                    // 分界符
                    case '(':
                    case ')':
                    case '[':
                    case ']':
                    case '{':
                    case '}':
                    {
                        pos = 0;

                        char op[2];
                        op[0] = now_char;

                        // 分界符和操作符的数量+1
                        operate_num ++;
                        // 输出
                        print(line_index, "delimiter", op);
                        break;
                    }
                    // 单引号和双引号
                    case '\'':
                    {
                        state = 0;
                        // 累计分界符和操作符数目
                        operate_num ++;
                        print(line_index, "delimiter", "\'");
```

```
192
193                         // 读取字符串内的内容
194                         now_char = get_char(forward);
195
196                         // 记录是否有第二个引号
197                         int mark_flag = 1;
198
199                         while ((now_char != '\'')){
200                             // 不为终止符则加入
201                             if (now_char != '\0'){
202                                 str_in_file[pos] = now_char;
203                                 pos ++;
204                                 now_char = get_char(forward);
205                             }
206                             // 如果在第二个单引号出现之前出现了终止符号 抛出一个错误
207                             else{
208                                 print(line_index, "error:the str end without another quotation mark", str_in_file);
209                                 mark_flag = 0;
210                                 break;
211                             }
212                         }
213
214                         if (mark_flag == 1) {
215                             // 正确生成了字符串 为其加入终结符 避免之前的影响
216                             str_in_file[pos] = '\0';
217                             // 输出正确内容
218                             print(line_index, "string", str_in_file);
219                             // 输出第二个引号
220                             print(line_index, "delimiter", "\'");
221                             // 累计字符串数目
222                             str_num ++;
223                             // 累计分界符和操作符数目
224                             operate_num ++;
225                         }
226
227                         // reset
228                         pos = 0;
229                         break;
230                     }
231                     // 与单引号一样
232                     case '"':
233                     {
234                         state = 0;
235                         // 累计分界符和操作符数目
236                         operate_num ++;
237                         print(line_index, "delimiter", "\"");
238
239                          // 读取字符串内的内容
240                         now_char = get_char(forward);
241
242                         // 记录是否有第二个引号
243                         int mark_flag = 1;
244
245                         while ((now_char != '"')){
246                             // 不为终止符则加入
247                             if (now_char != '\0'){
```

```
                                str_in_file[pos] = now_char;
                                pos ++;
                                now_char = get_char(forward);
                            }
                            // 如果在第二个双引号出现之前出现了终止符号 抛出一个错误
                            else{
                                print(line_index, "error:the str end without another quotation mark", str_in_file);
                                mark_flag = 0;
                                break;
                            }
                        }

                        if (mark_flag == 1) {
                            // 正确生成了字符串 为其加入终结符 避免之前的影响
                            str_in_file[pos] = '\0';
                            // 输出正确内容
                            print(line_index, "string", str_in_file);
                            // 输出第二个引号
                            print(line_index, "delimiter", "\"");
                            // 累计字符串数目
                            str_num ++;
                            // 累计分界符和操作符数目
                            operate_num ++;
                        }

                        // reset
                        pos = 0;
                        break;
                    }

                    // +
                    // 可能只有一个+ 也可能是++ +=
                    case '+':
                    {
                        state = 0;
                        // 累计分界符和操作符数目
                        operate_num ++;
                        now_char = get_char(forward);

                        switch (now_char) {
                            // ++
                            case '+':
                            {
                                print(line_index, "operator", "++");
                                pos = 0;
                                break;
                            }
                            // +=
                            case '=':
                            {
                                print(line_index, "operator", "+=");
                                pos = 0;
                                break;
                            }
                            // 其他情况 只有第一个加号或是错误的操作符
                            // 这里不考虑对形如“+-”的错误操作符
```

```
                            default:
                            {
                                // 因为刚才读了第一个+的下一个字符，这里只输出一个+，所以指针需要返回
                                get_backward(forward);
                                print(line_index, "operator", "+");
                                pos = 0;
                                break;
                            }
                        }
                        break;
                    }

                    // - 与 +类似
                    case '-':
                    {
                        state = 0;
                        // 累计分界符和操作符数目
                        operate_num ++;
                        now_char = get_char(forward);

                        switch (now_char) {
                            case '-':
                            {
                                print(line_index, "operator", "--");
                                pos = 0;
                                break;
                            }
                            case '=':
                            {
                                print(line_index, "operator", "-=");
                                pos = 0;
                                break;
                            }
                            // 可能为指针符号
                            case '>':
                            {
                                print(line_index, "operator", "->");
                                pos = 0;
                                break;
                            }
                            default:
                            {
                                get_backward(forward);
                                print(line_index, "operator", "-");
                                pos = 0;
                                break;
                            }
                        }
                        break;
                    }

                    // = 与 +类似
                    case '=':
                    {
                        state = 0;
                        // 累计分界符和操作符数目
                        operate_num ++;
```

```
                        now_char = get_char(forward);

                        switch (now_char) {
                            case '=':
                            {
                                print(line_index, "operator", "==");
                                pos = 0;
                                break;
                            }
                            default:
                            {
                                get_backward(forward);
                                print(line_index, "operator", "=");
                                pos = 0;
                                break;
                            }
                        }
                        break;
                    }

                    // < 与 +类似
                    case '<':
                    {
                        state = 0;
                        // 累计分界符和操作符数目
                        operate_num ++;
                        now_char = get_char(forward);

                        switch (now_char) {
                            case '<':
                            {
                                // 可能为<<=
                                char now_char = get_char(forward);
                                if (now_char == '='){
                                    print(line_index, "operator", "<<=");
                                    pos = 0;
                                    break;
                                }
                                else{
                                    // 回退
                                    get_backward(forward);
                                    print(line_index, "operator", "<<");
                                    pos = 0;
                                    break;
                                }
                            }
                            case '=':
                            {
                                print(line_index, "operator", "<=");
                                pos = 0;
                                break;
                            }
                            default:
                            {
                                get_backward(forward);
                                print(line_index, "operator", "<");
                                pos = 0;
                                break;
```

```
                            }
                        }
                        break;
                    }

                    // > 与 +类似
                    case '>':
                    {
                        state = 0;
                        // 累计分界符和操作符数目
                        operate_num ++;
                        now_char = get_char(forward);

                        switch (now_char) {
                            case '>':
                            {
                                // 可能为>>=
                                char now_char = get_char(forward);
                                if (now_char == '='){
                                    print(line_index, "operator", ">>=");
                                    pos = 0;
                                    break;
                                }
                                else{
                                    // 回退
                                    get_backward(forward);
                                    print(line_index, "operator", ">>");
                                    pos = 0;
                                    break;
                                }
                            }
                            case '=':
                            {
                                print(line_index, "operator", ">=");
                                pos = 0;
                                break;
                            }
                            default:
                            {
                                get_backward(forward);
                                print(line_index, "operator", ">");
                                pos = 0;
                                break;
                            }
                        }
                        break;
                    }

                    case '*':
                    {
                        state = 0;
                        // 累计分界符和操作符数目
                        operate_num ++;
                        now_char = get_char(forward);

                        switch (now_char) {
                            case '=':
                            {
```

```
                                print(line_index, "operator", "*=");
                                pos = 0;
                                break;
                            }
                            default:
                            {
                                get_backward(forward);
                                print(line_index, "operator", "*");
                                pos = 0;
                                break;
                            }
                        }
                        break;
                    }

                    case '/':
                    {
                        state = 0;
                        // 累计分界符和操作符数目
                        operate_num ++;
                        now_char = get_char(forward);

                        switch (now_char) {
                            case '=':
                            {
                                print(line_index, "operator", "/=");
                                pos = 0;
                                break;
                            }

                            // 行内注释
                            case '/':
                            {
                                // reset
                                state = 0;
                                pos = 0;
                                single_ann_num ++;
                                // 该行剩下的均已经不需要遍历 直接到下一行即可 直接return
                                cout << "line " << line_index << " have a single annotation skipping" << endl;
                                return ;
                            }

                            // 多行注释
                            case '*':
                            {
                                // 到多行注释寻找*状态
                                state = 6;
                                pos = 0;
                                multi_ann_num ++;
                                break;
                            }

                            default:
                            {
                                get_backward(forward);
                                print(line_index, "operator", "/");
```

```
                                pos = 0;
                                break;
                            }
                        }
                        break;
                    }

                    case '&':
                    {
                        state = 0;
                        // 累计分界符和操作符数目
                        operate_num ++;
                        now_char = get_char(forward);

                        switch (now_char) {
                            case '&':
                            {
                                print(line_index, "operator", "&&");
                                pos = 0;
                                break;
                            }
                            case '=':
                            {
                                print(line_index, "operator", "&=");
                                pos = 0;
                                break;
                            }
                            default:
                            {
                                get_backward(forward);
                                print(line_index, "operator", "&");
                                pos = 0;
                                break;
                            }
                        }
                        break;
                    }

                    case '|':
                    {
                        state = 0;
                        // 累计分界符和操作符数目
                        operate_num ++;
                        now_char = get_char(forward);

                        switch (now_char) {
                            case '|':
                            {
                                print(line_index, "operator", "||");
                                pos = 0;
                                break;
                            }
                            default:
                            {
                                get_backward(forward);
                                print(line_index, "operator", "|");
                                pos = 0;
                                break;
```

```
                            }
                        }
                        break;
                    }

                    case '!':
                    {
                        state = 0;
                        // 累计分界符和操作符数目
                        operate_num ++;
                        now_char = get_char(forward);

                        switch (now_char) {
                            case '=':
                            {
                                print(line_index, "operator", "!=");
                                pos = 0;
                                break;
                            }
                            default:
                            {
                                get_backward(forward);
                                print(line_index, "operator", "!");
                                pos = 0;
                                break;
                            }
                        }
                        break;
                    }

                    case '%':
                    {
                        state = 0;
                        // 累计分界符和操作符数目
                        operate_num ++;
                        now_char = get_char(forward);

                        switch (now_char) {
                            case '=':
                            {
                                print(line_index, "operator", "%=");
                                pos = 0;
                                break;
                            }
                            default:
                            {
                                get_backward(forward);
                                print(line_index, "operator", "%");
                                pos = 0;
                                break;
                            }
                        }
                        break;
                    }

                    case '^':
                    {
                        state = 0;
```

```
                        // 累计分界符和操作符数目
                        operate_num ++;
                        now_char = get_char(forward);

                        switch (now_char) {
                            case '=':
                            {
                                print(line_index, "operator", "^=");
                                pos = 0;
                                break;
                            }
                            default:
                            {
                                get_backward(forward);
                                print(line_index, "operator", "^");
                                pos = 0;
                                break;
                            }
                        }
                        break;
                    }

                    case ',':
                    {
                        // 累计分界符和操作符数目
                        operate_num ++;
                        state = 0;
                        pos = 0;
                        print(line_index, "operator", ",");
                        break;
                    }

                    case ';':
                    {
                        // 累计分界符和操作符数目
                        operate_num ++;
                        state = 0;
                        pos = 0;
                        print(line_index, "operator", ";");
                        break;
                    }

                    case '#':
                    {
                        // 累计分界符和操作符数目
                        operate_num ++;
                        state = 0;
                        pos = 0;
                        print(line_index, "operator", "#");
                        break;
                    }

                    case '.':
                    {
                        // 累计分界符和操作符数目
                        operate_num ++;
                        state = 0;
                        pos = 0;
```

```
                        print(line_index, "operator", ".");
                        break;
                    }

                    case '~':
                    {
                        // 累计分界符和操作符数目
                        operate_num ++;
                        state = 0;
                        pos = 0;
                        print(line_index, "operator", "~");
                        break;
                    }

                    // 剩下的就是数字或者word（变量名或关键字）
                    default:
                    {
                        // 数字
                        if (is_num(now_char)){
                            str_in_file[pos] = now_char;
                            pos ++;
                            number_num ++;
                            // 将状态转为2，往下查找整个数字
                            state = 2;
                        }

                        else{
                            // 变量名或关键字
                            if (is_letter(now_char) || now_char == '_'){
                                str_in_file[pos] = now_char;
                                pos ++;
                                word_num ++;
                                // 将状态转为1，往下查找整个word
                                state = 1;
                            }
                        }
                        break;
                    }
                }

                break;
            }

            // 处理word
            case 1:
            {
                // 不读完就不停
                while (1) {
                    now_char = get_char(forward);
                    // 因为第一个已经确定是字母，所以接下来的可以是字母、数字或者下划线
                    if (is_num(now_char) || is_letter(now_char) || now_char == '_'){
                        str_in_file[pos] = now_char;
                        pos ++;
                    }
                    // 如果出现了空格等别的字符
                    else {
```

```
                        // 给当前要输出的字符串加上结束符
                        str_in_file[pos] = '\0';
                        // 判断该word是不是关键字
                        bool keyword_res = is_keyword(str_in_file);
                        // 如果是关键字
                        if (keyword_res) {
                            print(line_index, "keyword", str_in_file);
                            // word_num 已经加过了 这里只加 keyword_num
                            keyword_num ++;
                        }
                        // 不是关键字
                        else {
                            print(line_index, "variable", str_in_file);
                            // word_num 已经加过了 这里只加 varible_num
                            varible_num ++;
                        }

                        // 处理完毕 进行reset
                        // 这个字符不是属于要输出的字符串之内的，回退
                        get_backward(forward);
                        pos = 0;
                        state = 0;
                        break;
                    }
                }
                break;
            }



            // 处理数字
            case 2:
            {
                // 同样不读完不停
                while (1) {
                    now_char = get_char(forward);

                    // 如果是数字 继续读
                    if (is_num(now_char)){
                        str_in_file[pos] = now_char;
                        pos ++;
                    }

                    // 如果下一个不是数字
                    else {

                        // 可能是小数
                        if (now_char == '.'){
                            str_in_file[pos] = now_char;
                            pos ++;
                            // 切换到小数状态
                            state = 3;
                            // 退出while
                            break;
                        }

                        // 有可能是科学计数法 如1e-3
                        else if (now_char == 'e' || now_char == 'E'){
```

```
821                             str_in_file[pos] = now_char;
822                             pos ++;
823                             // 切换到科学计数法状态
824                             state = 4;
825                             // 退出while
826                             break;
827                         }
828 
829                         // 不是上述的任何符号 则结束数字模式
830                         else {
831                             // 补终结符
832                             str_in_file[pos] = '\0';
833                             print(line_index, "integral number",
str_in_file);
834                             // 该字符没输出 需要回退
835                             get_backward(forward);
836 
837                             // reset
838                             pos = 0;
839                             state = 0;
840                             // 退出while
841                             break;
842                         }
843 
844                     }
845                 }
846                 break;
847             }
848 
849 
850             // 小数状态
851             case 3:
852             {
853                 while (1) {
854                     now_char = get_char(forward);
855                     // 是数字
856                     if (is_num(now_char)) {
857                         str_in_file[pos] = now_char;
858                         pos ++;
859                     }
860 
861                     // eg:1.5e-3   转移到科学计数法状态
862                     else if (now_char == 'e' || now_char == 'E') {
863                         str_in_file[pos] = now_char;
864                         pos ++;
865                         state = 4;
866                         break;
867                     }
868 
869                     // 不是数字或e
870                     else {
871                         str_in_file[pos] = '\0';
872                         print(line_index, "fraction number", str_in_file);
873                         // 该字符没输出 需要回退
874                         get_backward(forward);
875                         // reset
876                         pos = 0;
877                         state = 0;
```

```
878                         // 退出while
879                         break;
880                     }
881                 }
882                 break;
883             }
884 
885             // 科学计数状态(无负号)
886             case 4:
887             {
888                 while (1) {
889 
890                     now_char = get_char(forward);
891                     // 数字
892                     if (is_num(now_char)) {
893                         str_in_file[pos] = now_char;
894                         pos ++;
895                     }
896 
897                     // 负号  科学计数法的负次幂
898                     else if (now_char == '-') {
899                         str_in_file[pos] = now_char;
900                         pos ++;
901                         state = 5;
902                         break;
903                     }
904 
905                     // 结束幂次正数的科学计数法
906                     else {
907                         str_in_file[pos] = '\0';
908                         print(line_index, "scientific notation number (pos
    power)", str_in_file);
909                         // 该字符没输出 需要回退
910                         get_backward(forward);
911                         // reset
912                         pos = 0;
913                         state = 0;
914                         // 退出while
915                         break;
916                     }
917                 }
918                 break;
919             }
920 
921             // 科学技术状态 负幂次
922             case 5:
923             {
924                 while (1) {
925                     now_char = get_char(forward);
926                     // 数字
927                     if (is_num(now_char)) {
928                         str_in_file[pos] = now_char;
929                         pos ++;
930                     }
931 
932                     // 结束科学计数法
933                     else {
934                         str_in_file[pos] = '\0';
```

```
935                         print(line_index, "scientific notation number (neg
    power)", str_in_file);
936                         // 该字符没输出 需要回退
937                         get_backward(forward);
938                         // reset
939                         pos = 0;
940                         state = 0;
941                         // 退出while
942                         break;
943                     }
944                 }
945                 break;
946             }
947 
948             // 多行注释 已经检索到/*      寻找*
949             case 6:
950             {
951                 while (1) {
952                     now_char = get_char(forward);
953                     // 如果该行已经遍历完毕
954                     if (now_char == '\0'){
955                         // 设置多行注释标记
956                         multi_annotation = 1;
957                         break;
958                     }
959 
960                     // 找到 *
961                     else if (now_char == '*') {
962                         // 寻找 /
963                         state = 7;
964                         pos = 0;
965                         break;
966                     }
967 
968                     // 没找到就继续找
969                     else {
970                         continue;
971                     }
972                 }
973                 break;
974             }
975 
976             // 多行注释 已经检索到/* *      寻找/
977             case 7:
978             {
979                 now_char = get_char(forward);
980 
981                 // 已经检索完多行注释
982                 if (now_char == '/') {
983                     // 直接跳过 回到状态0
984                     multi_annotation = 0;
985                     cout << "have a multi-annotation skipping" << endl;
986                     state = 0;
987                     pos = 0;
988                     break;
989                 }
990 
991                 // 该行已经遍历完毕 (存在这种情况,因为是从case 6 过来的)
```

```
                // 出去之后该函数会结束 读下一行 所以不能设置state = 6 还是得用标记
                else if (now_char == '\0') {
                    // 设置多行注释标记
                    multi_annotation = 1;
                    break;
                }

                // 再接收到*
                else if (now_char == '*') {
                    // 仍然为state = 7
                    state = 7;
                    pos = 0;
                    break;
                }

                // 该行没结束 且不是/  则回到状态6
                else {
                    state = 6;
                    pos = 0;
                    break;
                }
                break;
            }

            // 不在任何情况内 抛出一个错误
            default:
            {
                print(line_index, "error! no state is matching.", "-");
                break;
            }

        }
    }
}



int main()
{
    // 打开文件
    FILE *file = fopen("s.txt", "r");
    // 标记当前行数
    int line_index = 0;

    // 为了标记行,选择一行行输入
    while (1){
        char *now_line_str = fgets(buffer, 10000, file);

        if (now_line_str != NULL){
            line_index ++;
            lexical_analysis(line_index, buffer);
        }
        else break;
    }

    // 如果多行注释始终没有配对成功 则报错
    if (multi_annotation == 1) {
```

```
        print(line_index, "error: multi-annotation compared unsuccessfully", "-");
    }

    cout <<endl;
    cout <<endl;
    cout << "行数:" << line_index << endl;
    cout << "字符总数:" << sign_num << endl;
    cout << "单词总数:" << word_num << endl;
    cout << "关键字总数:" << keyword_num <<endl;
    cout << "变量名数目:" << varible_num << endl;
    cout << "数学常量数目:" << number_num << endl;
    cout << "单行注释数目:" << single_ann_num << endl;
    cout << "多行注释数目:" << multi_ann_num << endl;
    cout << "分界符和操作符数目(包括注释):" << operate_num << endl;
    cout << "字符串常量数目(包括printf内的输出):" << str_num << endl;
}

```

五、方法一测试

寻找了一段判断素数的C语言代码，并对其增加了一些测试的必要语句，如一些注释等。

① 测试代码：（完全正确）

```c
/***
    use for lex-test.
**/

#include <math.h>
#include <stdio.h>
#define harry 916

//this is the program
int main()
{
    int m,flag;
    scanf("%d", &m);
    flag = 1;
    for(int i = 2;i <= sqrt(m);i++){
        if(m%i == 0)
        {
            flag = 0;
            break;
        }
    }
    if(flag){
        //output
        printf("%d is a primer.\n", m);
    }
    else{
        //output
        printf("%d is not a primer.\n", m);
    }
    return 0;
```

```
}
```

输出:

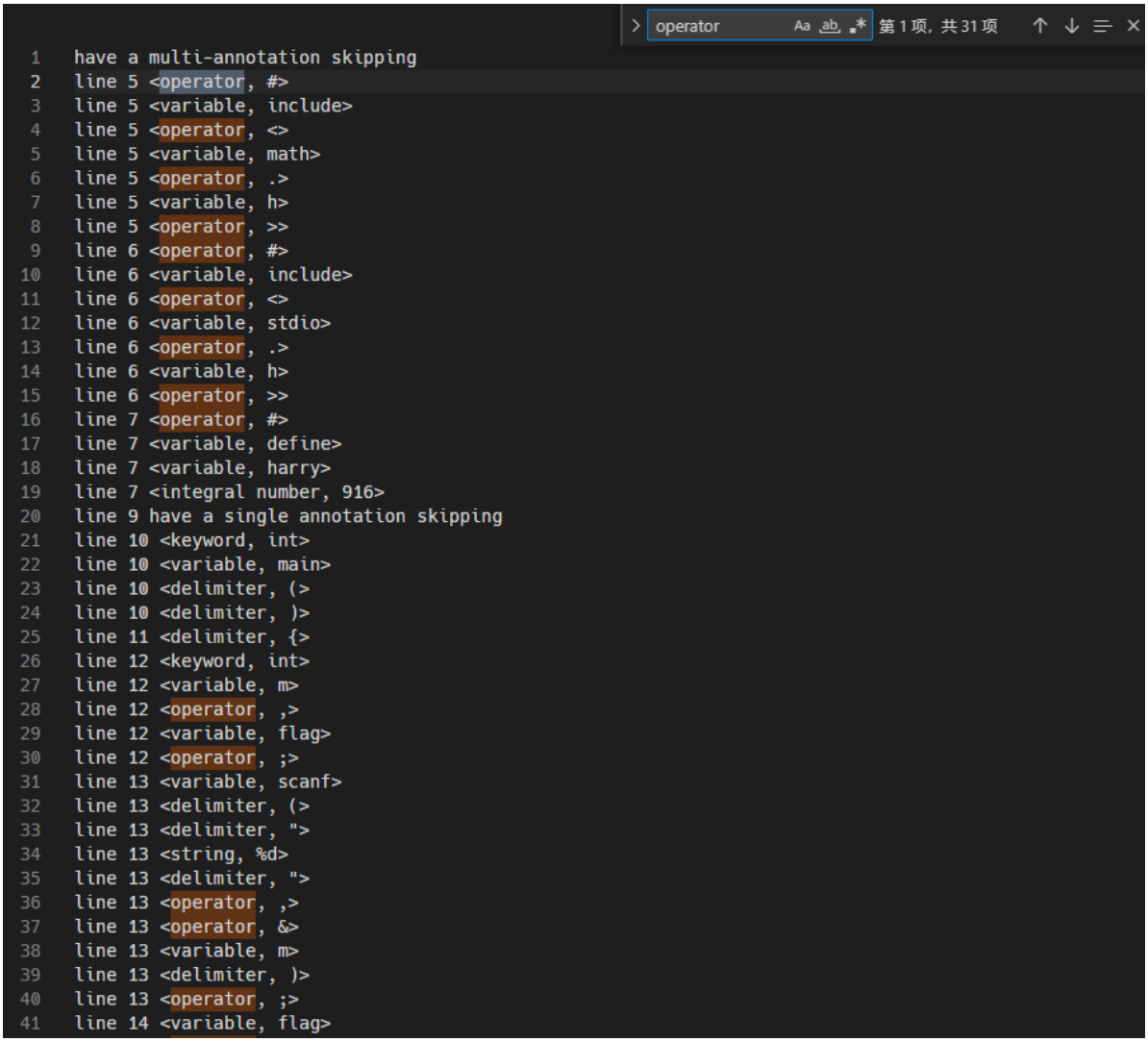
```
have a multi-annotation skipping
line 5 <operator, #>
line 5 <variable, include>
line 5 <operator, <>
line 5 <variable, math>
line 5 <operator, .>
line 5 <variable, h>
line 5 <operator, >>
line 6 <operator, #>
line 6 <variable, include>
line 6 <operator, <>
line 6 <variable, stdio>
line 6 <operator, .>
line 6 <variable, h>
line 6 <operator, >>
line 7 <operator, #>
line 7 <variable, define>
line 7 <variable, harry>
line 7 <integral number, 916>
line 9 have a single annotation skipping
line 10 <keyword, int>
line 10 <variable, main>
line 10 <delimiter, (>
line 10 <delimiter, )>
line 11 <delimiter, {>
line 12 <keyword, int>
line 12 <variable, m>
line 12 <operator, ,>
line 12 <variable, flag>
line 12 <operator, ;>
line 13 <variable, scanf>
line 13 <delimiter, (>
line 13 <delimiter, ">
line 13 <string, %d>
line 13 <delimiter, ">
line 13 <operator, ,>
line 13 <operator, &>
line 13 <variable, m>
line 13 <delimiter, )>
line 13 <operator, ;>
line 14 <variable, flag>
line 14 <operator, =>
line 14 <integral number, 1>
line 14 <operator, ;>
line 15 <keyword, for>
line 15 <delimiter, (>
line 15 <keyword, int>
line 15 <variable, i>
line 15 <operator, =>
line 15 <integral number, 2>
line 15 <operator, ;>
```

```
line 15 <variable, i>
line 15 <operator, <=>
line 15 <variable, sqrt>
line 15 <delimiter, (>
line 15 <variable, m>
line 15 <delimiter, )>
line 15 <operator, ;>
line 15 <variable, i>
line 15 <operator, ++>
line 15 <delimiter, )>
line 15 <delimiter, {>
line 16 <keyword, if>
line 16 <delimiter, (>
line 16 <variable, m>
line 16 <operator, %>
line 16 <variable, i>
line 16 <operator, ==>
line 16 <integral number, 0>
line 16 <delimiter, )>
line 17 <delimiter, {>
line 18 <variable, flag>
line 18 <operator, =>
line 18 <integral number, 0>
line 18 <operator, ;>
line 19 <keyword, break>
line 19 <operator, ;>
line 20 <delimiter, }>
line 21 <delimiter, }>
line 22 <keyword, if>
line 22 <delimiter, (>
line 22 <variable, flag>
line 22 <delimiter, )>
line 22 <delimiter, {>
line 23 have a single annotation skipping
line 24 <variable, printf>
line 24 <delimiter, (>
line 24 <delimiter, ">
line 24 <string, %d is a primer.\n>
line 24 <delimiter, ">
line 24 <operator, ,>
line 24 <variable, m>
line 24 <delimiter, )>
line 24 <operator, ;>
line 25 <delimiter, }>
line 26 <keyword, else>
line 26 <delimiter, {>
line 27 have a single annotation skipping
line 28 <variable, printf>
line 28 <delimiter, (>
line 28 <delimiter, ">
line 28 <string, %d is not a primer.\n>
line 28 <delimiter, ">
line 28 <operator, ,>
line 28 <variable, m>
line 28 <delimiter, )>
line 28 <operator, ;>
line 29 <delimiter, }>
line 30 <keyword, return>
```

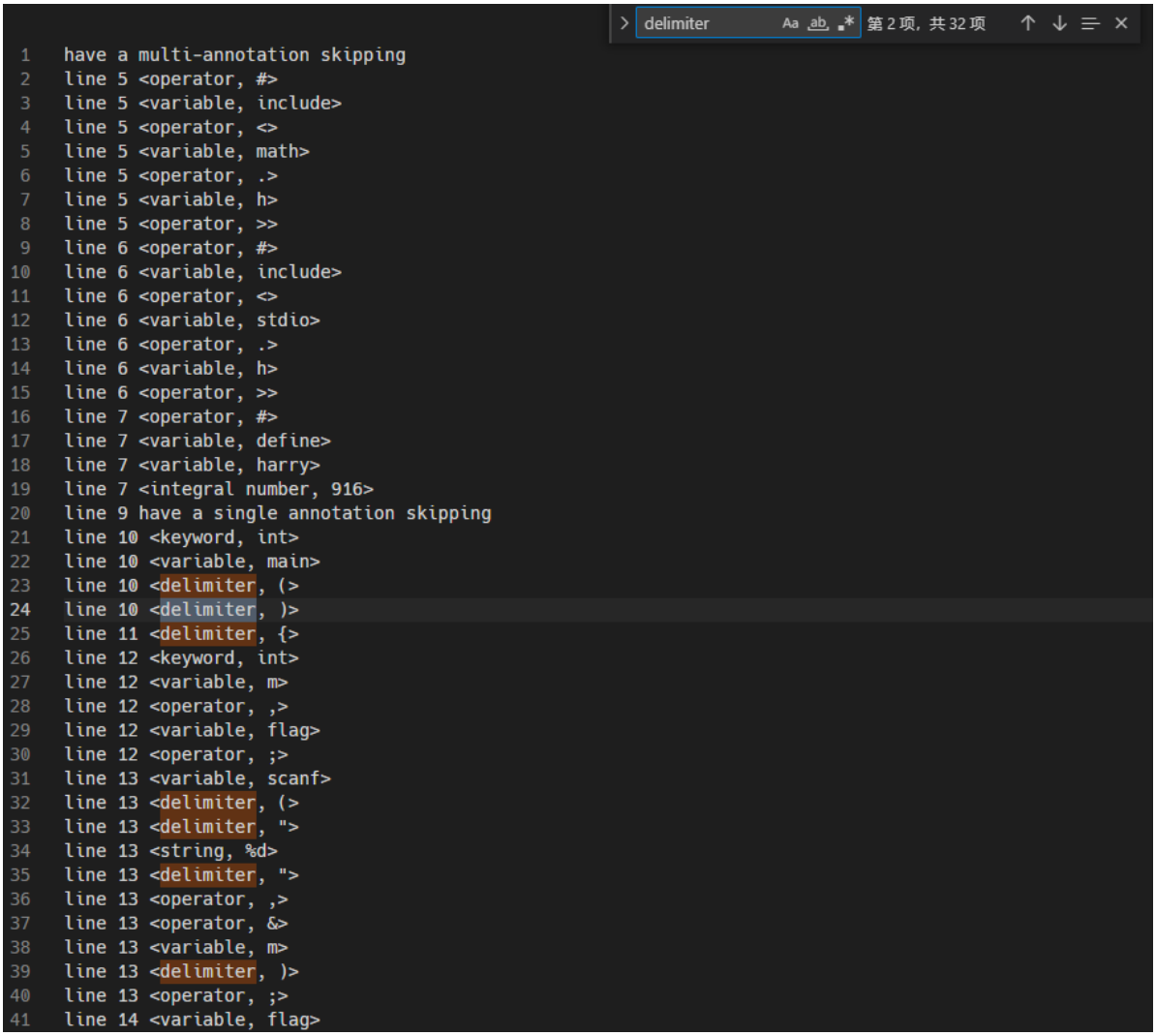
```
line 30 <integral number, 0>
line 30 <operator, ;>
line 31 <delimiter, }>


行数:31
字符总数:513
单词总数:36
关键字总数:9
变量名数目:27
数学常量数目:6
单行注释数目:3
多行注释数目:1
分界符和操作符数目(包括注释):67
字符串常量数目(包括printf内的输出):3
```

验证:

operator有31项。

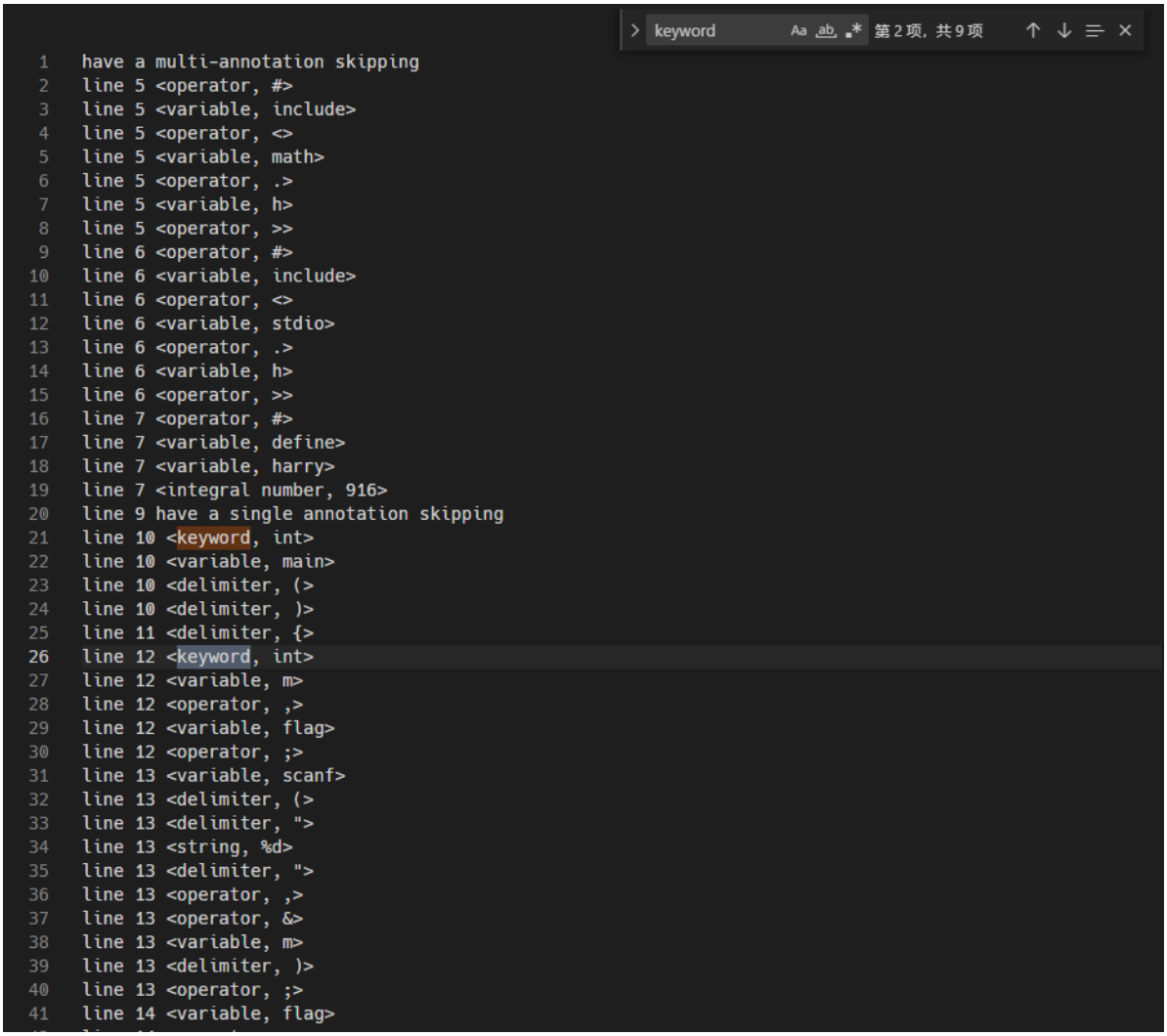


delimiter有32项。



注释有4项，所以分界符和操作符数量为67项，符合输出。

关键词有9项，符合输出。



经查证，其他数据也均符合输出。

② 测试代码：（28行双引号没有配对）

```
/***
    use for lex-test.
**/

#include <math.h>
#include <stdio.h>
#define harry 916

//this is the program
int main()
{
    int m,flag;
    scanf("%d", &m);
    flag = 1;
    for(int i = 2;i <= sqrt(m);i++){
        if(m%i == 0)
        {
            flag = 0;
            break;
```

```
        }
    }
    if(flag){
        //output
        printf("%d is a primer.\n", m);
    }
    else{
        //output
        printf("%d is not a primer.\n, m);
    }
    return 0;
}
```

输出：在102行抛出了异常

```
have a multi-annotation skipping
line 5 <operator, #>
line 5 <variable, include>
line 5 <operator, <>
line 5 <variable, math>
line 5 <operator, .>
line 5 <variable, h>
line 5 <operator, >>
line 6 <operator, #>
line 6 <variable, include>
line 6 <operator, <>
line 6 <variable, stdio>
line 6 <operator, .>
line 6 <variable, h>
line 6 <operator, >>
line 7 <operator, #>
line 7 <variable, define>
line 7 <variable, harry>
line 7 <integral number, 916>
line 9 have a single annotation skipping
line 10 <keyword, int>
line 10 <variable, main>
line 10 <delimiter, (>
line 10 <delimiter, )>
line 11 <delimiter, {>
line 12 <keyword, int>
line 12 <variable, m>
line 12 <operator, ,>
line 12 <variable, flag>
line 12 <operator, ;>
line 13 <variable, scanf>
line 13 <delimiter, (>
line 13 <delimiter, ">
line 13 <string, %d>
line 13 <delimiter, ">
line 13 <operator, ,>
line 13 <operator, &>
line 13 <variable, m>
line 13 <delimiter, )>
line 13 <operator, ;>
```

```
line 14 <variable, flag>
line 14 <operator, =>
line 14 <integral number, 1>
line 14 <operator, ;>
line 15 <keyword, for>
line 15 <delimiter, (>
line 15 <keyword, int>
line 15 <variable, i>
line 15 <operator, =>
line 15 <integral number, 2>
line 15 <operator, ;>
line 15 <variable, i>
line 15 <operator, <=>
line 15 <variable, sqrt>
line 15 <delimiter, (>
line 15 <variable, m>
line 15 <delimiter, )>
line 15 <operator, ;>
line 15 <variable, i>
line 15 <operator, ++>
line 15 <delimiter, )>
line 15 <delimiter, {>
line 16 <keyword, if>
line 16 <delimiter, (>
line 16 <variable, m>
line 16 <operator, %>
line 16 <variable, i>
line 16 <operator, ==>
line 16 <integral number, 0>
line 16 <delimiter, )>
line 17 <delimiter, {>
line 18 <variable, flag>
line 18 <operator, =>
line 18 <integral number, 0>
line 18 <operator, ;>
line 19 <keyword, break>
line 19 <operator, ;>
line 20 <delimiter, }>
line 21 <delimiter, }>
line 22 <keyword, if>
line 22 <delimiter, (>
line 22 <variable, flag>
line 22 <delimiter, )>
line 22 <delimiter, {>
line 23 have a single annotation skipping
line 24 <variable, printf>
line 24 <delimiter, (>
line 24 <delimiter, ">
line 24 <string, %d is a primer.\n>
line 24 <delimiter, ">
line 24 <operator, ,>
line 24 <variable, m>
line 24 <delimiter, )>
line 24 <operator, ;>
line 25 <delimiter, }>
line 26 <keyword, else>
line 26 <delimiter, {>
line 27 have a single annotation skipping
```

```
line 28 <variable, printf>
line 28 <delimiter, (>
line 28 <delimiter, ">
line 28 <error:the str end without another quotation mark, %d is not a primer.\n, m);
>
line 29 <delimiter, }>
line 30 <keyword, return>
line 30 <integral number, 0>
line 30 <operator, ;>
line 31 <delimiter, }>


行数:31
字符总数:511
单词总数:35
关键字总数:9
变量名数目:26
数学常量数目:6
单行注释数目:3
多行注释数目:1
分界符和操作符数目(包括注释):63
字符串常量数目(包括printf内的输出):2
```

分析：

- 该词法分析程序能够识别用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；
- 可以识别并且跳过源程序的注释；
- 可以统计源程序的语句行数，各类单词的个数，字符总数，输出统计结果；
- 可以检查源程序中存在的词法错误，并报告错误所在的位置；
- 可以对出现的错误进行适当回复，继续进行词法分析。

上述要求都能按要求完成。

六、方法二分析

lex实际上就是根据所输入的正则表达式对相应类型的词语进行辨识，识别到相应的词语后根据输入的对应c语句进行操作或输出。

lex首先通过下列语句进行编译，

```
flex lex.i
```

生成一个名为lex.yy.c的文件。再对该文件进行编译：

```
gcc lex.yy.c -lfl
```

生成.out文件。执行该文件可以输出结果。

正则表达式由书中第三章推导而来。

七、方法二代码展示

```
%{
#include<stdlib.h>
#include<stdio.h>
%}

digit       [0-9]
letter      [a-zA-Z]
id          ({letter}|_)({letter}|{digit}|_)*
integer     {digit}+
realnum     {digit}+"."{digit}+
operator    "+"|-|"*"|"/"|<=|<|==|=|>=|>|>>|<<|&|&&|"%"
delim       [\t\n\r\0\v\040]
empty       {delim}+
string      \"[^"]*\"

%%
{id}            {printf("identifier: %s\n",yytext);}
{integer}       {printf("integer: %s\n",yytext);}
{realnum}       {printf("realnum: %s\n",yytext);}
{operator}      {printf("operator: %s\n",yytext);}
"("             {printf("left bracket: %s\n",yytext);}
")"             {printf("right bracket: %s\n",yytext);}
"{"             {printf("left bracket: %s\n",yytext);}
"}"             {printf("right bracket: %s\n",yytext);}
","             {printf("comma: %s\n",yytext);}
":"             {printf("colon: %s\n",yytext);}
";"             {printf("semicolon: %s\n",yytext);}
"#".*           {printf("head file: %s\n",yytext);}
{empty}         { }
{string}        {printf("string: %s\n",yytext);}
.           {printf("can't not identify: %s\n",yytext);}

%%
int main(int argc,char **argv)
{
    yyin  = fopen("./test.c", "r");
    yylex();
    return 0;
}
int yywrap()
{
    return 1;
}
```

八、方法二测试

输入test.c进行测试：

```
#include <math.h>
#include <stdio.h>
int main()
{
    int m,flag;
    scanf("%d", &m);
```

```
    flag = 1;
    for(int i = 2;i <= sqrt(m);i++){
        if(m%i == 0)
        {
            flag = 0;
            break;
        }
    }
    if(flag){
        printf("%d is a primer.\n", m);
    }
    else{
        printf("%d is not a primer.\n", m);
    }
    return 0;
}
```

测试结果：

```
head file: #include <math.h>
head file: #include <stdio.h>
identifier: int
identifier: main
left bracket: (
right bracket: )
left bracket: {
identifier: int
identifier: m
comma: ,
identifier: flag
semicolon: ;
identifier: scanf
left bracket: (
string: "%d"
comma: ,
operator: &
identifier: m
right bracket: )
semicolon: ;
identifier: flag
operator: =
integer: 1
semicolon: ;
identifier: for
left bracket: (
identifier: int
identifier: i
operator: =
integer: 2
semicolon: ;
identifier: i
operator: <=
identifier: sqrt
left bracket: (
identifier: m
```

```
right bracket: )
semicolon: ;
identifier: i
operator: +
operator: +
right bracket: )
left bracket: {
identifier: if
left bracket: (
identifier: m
operator: %
identifier: i
operator: ==
integer: 0
right bracket: )
left bracket: {
identifier: flag
operator: =
integer: 0
semicolon: ;
identifier: break
semicolon: ;
right bracket: }
right bracket: }
identifier: if
left bracket: (
identifier: flag
right bracket: )
left bracket: {
identifier: printf
left bracket: (
string: "%d is a primer.\n"
comma: ,
identifier: m
right bracket: )
semicolon: ;
right bracket: }
identifier: else
left bracket: {
identifier: printf
left bracket: (
string: "%d is not a primer.\n"
comma: ,
identifier: m
right bracket: )
semicolon: ;
right bracket: }
identifier: return
integer: 0
semicolon: ;
right bracket: }

```

九、实验总结与心得

通过本次实验，我深刻地认识到了词法分析对于程序编译的重要性。而对词法分析来说，状态的确定与相互的转换时十分重要的。没有前置的思考与统筹，是很难系统地写出一个结构化的程序结构的。

将词法与语法分析安排在同一遍中，词法分析作为语法分析的子程序时，避免了中间文件，省去了取送符号的工作，有利于编译程序的效率。而词法分析与语法分析以生产者和消费者的关系同步运行时，也能够提高效率。

将词法分析独立出来，让我们至少能够简化词法、语法分析中的一项任务。例如，如果一个语法分析器必须把空白字符和注释当做语法单元进行处理，那么它就会比那些假设空白字符和注释已经被词法分析器过滤掉的处理器复杂得多。如果我们正在设计一个新语言，将词法和语法分开考虑有助于我们得到一个更加清晰地的语言设计方案。同时可以提高编译器的效率：把词法分析器独立出来使我们能够使用专用于词法分析任务、不进行语法分析的技术。此外，我们可以使用专门的用于读取输入字符的缓冲技术来显著提高编译器的速度。还可以增强编译器的可移植性。输入设备相关的特殊性可以被限制在词法分析器中。通过实验，我深刻了解了上述益处。

我也尝试了使用lex进行词法分析。lex的完备性与便捷性令我惊讶。这更加吸引我对词法分析进行更加深入的学习。