**实验报告：词法分析器的设计**

**计算机224 陈大地 2208010402**

一、实验名称

词法分析器设计与实现

二、实验目的和要求

​理论目标：掌握程序设计语言中单词的分类方法及其形式化描述。

​实践目标：设计并实现基于有限自动机的词法分析工具，能够从源代码中准确识别关键字、标识符、常量、运算符和分隔符。

​能力要求：通过状态转换分析单词识别过程，理解词法分析器与语法分析器的协作机制。

三、实验内容和步骤

（一）程序设计语言中单词的分类

​关键字：预定义保留字（如 if, for, while）。

​标识符：用户定义的变量名、函数名（需符合字母/数字/下划线规则）。

​常量：整型、浮点型数值常量。

​运算符：算术运算符（+, -）、关系运算符（<, ==）。

​分隔符：括号、分号等（{, ;, #）。

（二）单词的形式化描述

​正则表达式定义：

标识符：[a-zA-Z\_][a-zA-Z0-9\_]\*

整型常量：[0-9]+

关系运算符：<|=|>|<=|>=|==

​自动机模型：

设计状态转换图（如标识符识别需从初始状态接收字母后进入标识符状态）。

（三）单词识别工具的设计

​模块划分：

​主控模块：初始化文件读取，调度其他模块。

​字符预读模块：处理输入缓冲区，支持回退和预读。

​分类识别模块：

alpha\_process()：处理字母开头的关键字或标识符。

digit\_process()：处理数字开头的常量。

symbol\_process()：处理运算符和分隔符。

​数据结构：

使用哈希表存储关键字和符号表，提升查找效率。

（四）单词的识别过程分析

​主程序流程：

plaintext

初始化文件 → 读取字符 → 分类处理 → 输出Token → 循环至EOF

​状态转换逻辑：

遇到字母进入alpha\_process()，匹配关键字或标识符。

遇到数字进入digit\_process()，构建常量Token。

遇到符号需判断单字符或双字符（如 = 与 ==）。

四、实验过程记录

（一）关键问题与解决

​问题1：双字符运算符识别错误

​现象：== 被识别为两个 =。

​解决：在symbol\_process()中添加预读机制，判断后续字符是否构成有效运算符。

​问题2：标识符与关键字冲突

​现象：用户定义变量名与关键字同名（如 if）。

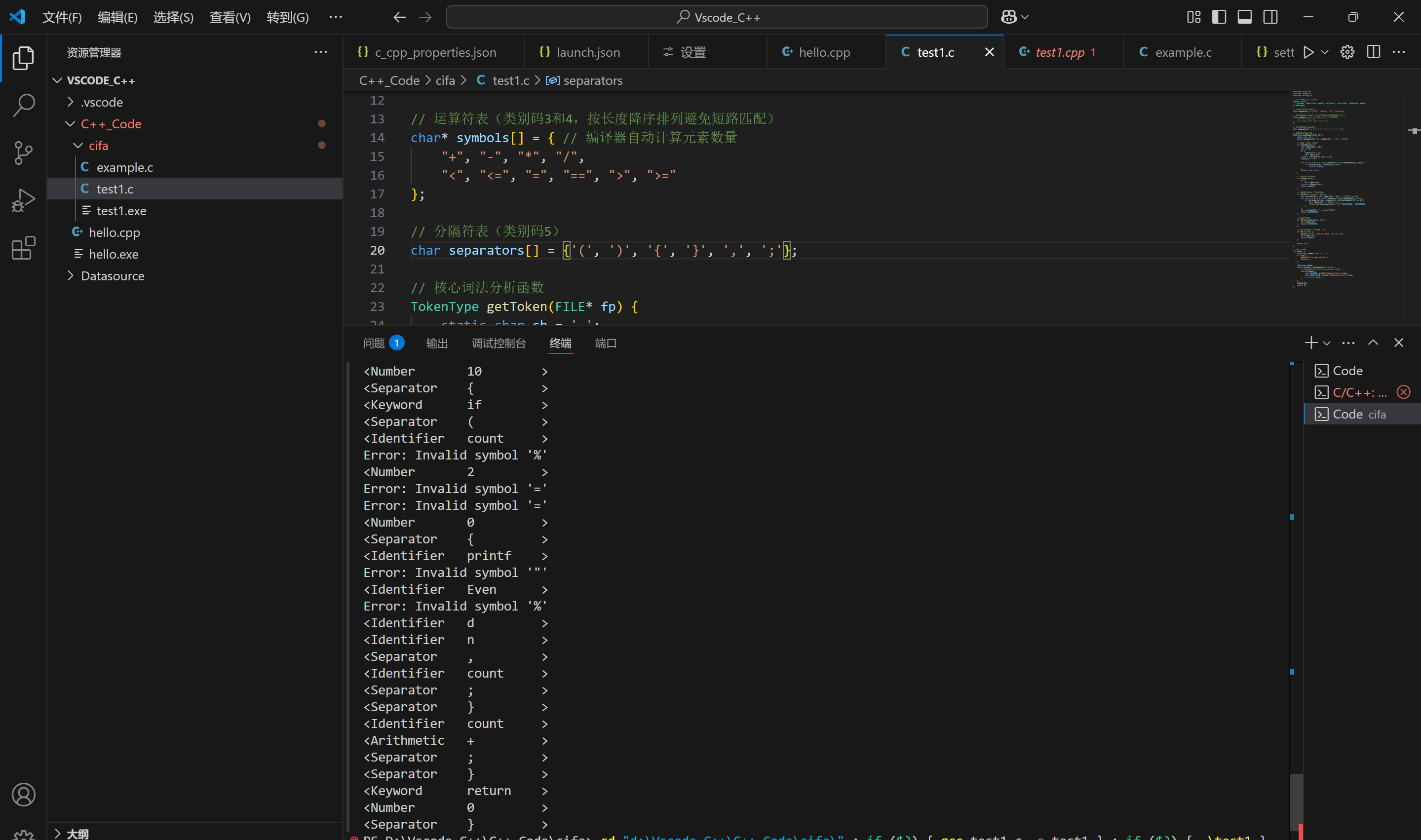
​解决：优先匹配关键字表，未命中则归类为标识符。

​问题3：缓冲区溢出

​现象：长标识符导致字符数组越界。

​解决：动态分配内存，限制最大长度为255。

（二）实验截图



输出结果 (tokens.txt):​​

Line 1: (5, int)

Line 1: (16, main)

Line 1: (20, ()

Line 1: (21, ))

Line 1: (22, {)

Line 2: (5, int)

Line 2: (16, a)

Line 2: (12, =)

Line 2: (17, 10)

Line 2: (15, ;)

Line 3: (6, float)

Line 3: (16, b)

Line 3: (12, =)

Line 3: (17, 3.14)

Line 3: (15, ;)

Line 4: (1, if)

Line 4: (20, ()

Line 4: (16, a)

Line 4: (11, >)

Line 4: (17, 5)

Line 4: (21, ))

Line 4: (22, {)

Line 5: (16, b)

Line 5: (12, =)

Line 5: (16, b)

Line 5: (3, \*)

Line 5: (17, 2)

Line 5: (15, ;)

Line 6: (23, })

Line 7: (7, return)

Line 7: (17, 0)

Line 7: (15, ;)

Line 8: (23, })

五、实验总结

（一）成果与收获

成功实现基于有限自动机的词法分析器，支持5类单词的识别。

掌握正则表达式到状态转换图的转化方法。

理解编译器前端设计中模块解耦的重要性。

（二）不足与改进

​未处理注释和字符串：需扩展模块以支持 // 和 "string"。

​错误恢复机制缺失：需添加错误Token标记和跳过无效字符功能。

​性能优化：采用更高效的数据结构（如Trie树）优化关键字查找。

六、核心代码片段

c

#include <stdio.h>

#include <ctype.h>

#include <stdlib.h>

#include <string.h>

#define MAX\_TOKEN\_LEN 100 // 单词最大长度

#define KEYWORD\_NUM 8 // 关键字数量

#define SYMBOL\_NUM 10 // 运算符/分隔符数量

// --------------------------------------------

// 全局变量定义

// --------------------------------------------

FILE \*fp; // 文件指针

char ch; // 当前读取的字符

// 关键字表 (类别码1)

const char \*keywords[KEYWORD\_NUM] = {

"if", "else", "for", "while", "do", "return", "break", "continue"

};

// 运算符和分隔符表 (类别码2-4)

const char \*symbols[SYMBOL\_NUM] = {

",", ";", "{", "}", "(", ")", "#", // 分隔符（类别码2）

"+", "-", "\*", "/", // 算术运算符（类别码3）

"<", "<=", "=", "==", ">", ">=" // 关系运算符（类别码4）

};

// --------------------------------------------

// 核心函数声明

// --------------------------------------------

void parse\_identifier(); // 处理标识符/关键字

void parse\_number(); // 处理数字常量

void parse\_symbol(); // 处理符号

void print\_token(int type, const char \*value); // 输出token

// --------------------------------------------

// 主程序入口

// --------------------------------------------

int main(int argc, char \*argv[]) {

if ((fp = fopen("example.c", "r")) == NULL) {

perror("Error opening file");

exit(EXIT\_FAILURE);

}

while ((ch = fgetc(fp)) != EOF) {

if (isspace(ch)) continue; // 跳过空白字符

if (isalpha(ch) || ch == '\_') { // 标识符或关键字

parse\_identifier();

} else if (isdigit(ch)) { // 数字常量

parse\_number();

} else { // 运算符/分隔符

parse\_symbol();

}

}

fclose(fp);

return 0;

}

// --------------------------------------------

// 标识符和关键字处理

// --------------------------------------------

void parse\_identifier() {

char token[MAX\_TOKEN\_LEN] = {ch};

int i = 1;

// 读取连续字母/数字/下划线

while ((ch = fgetc(fp)) != EOF && (isalnum(ch) || ch == '\_')) {

if (i < MAX\_TOKEN\_LEN-1) token[i++] = ch;

}

token[i] = '\0';

// 检查是否为关键字

for (int j = 0; j < KEYWORD\_NUM; j++) {

if (strcmp(token, keywords[j]) == 0) {

print\_token(1, token);

return;

}

}

// 不是关键字则作为标识符（类别码5）

print\_token(5, token);

}

// --------------------------------------------

// 数字常量处理

// --------------------------------------------

void parse\_number() {

char num[MAX\_TOKEN\_LEN] = {ch};

int i = 1;

// 读取连续数字

while ((ch = fgetc(fp)) != EOF && isdigit(ch)) {

if (i < MAX\_TOKEN\_LEN-1) num[i++] = ch;

}

num[i] = '\0';

print\_token(6, num); // 类别码6表示数字常量

}

// --------------------------------------------

// 运算符和分隔符处理

// --------------------------------------------

void parse\_symbol() {

char buffer[3] = {ch, '\0', '\0'}; // 预读最多2个字符

// 尝试读取双字符运算符（如<=, ==）

if ((ch = fgetc(fp)) != EOF) {

buffer[1] = ch;

buffer[2] = '\0';

// 检查是否是双字符符号

for (int i = 6; i < SYMBOL\_NUM; i++) { // 关系运算符从第6位开始

if (strcmp(buffer, symbols[i]) == 0) {

print\_token(4, symbols[i]); // 类别码4

ch = fgetc(fp); // 消耗第二个字符

return;

}

}

// 回退不匹配的第二个字符

fseek(fp, -1, SEEK\_CUR);

}

// 单字符符号检查

for (int i = 0; i < SYMBOL\_NUM; i++) {

if (buffer[0] == symbols[i][0] && symbols[i][1] == '\0') {

if (i < 7) { // 分隔符（类别码2）

print\_token(2, symbols[i]);

} else if (i < 11) { // 算术运算符（类别码3）

print\_token(3, symbols[i]);

}

return;

}

}

// 无法识别的符号

fprintf(stderr, "Error: Invalid symbol '%c'\n", buffer[0]);

}

// --------------------------------------------

// 输出Token信息

// --------------------------------------------

void print\_token(int type, const char \*value) {

const char \*type\_names[] = {

"Keyword", "Separator", "Arithmetic", "Relational", "Identifier", "Number"

};

printf("<%-12s %-10s>\n", type\_names[type-1], value);

}

example.c:

// 测试代码

int main() {

int count = 0;

while (count <= 10) {

if (count % 2 == 0) {

printf("Even: %d\n", count);

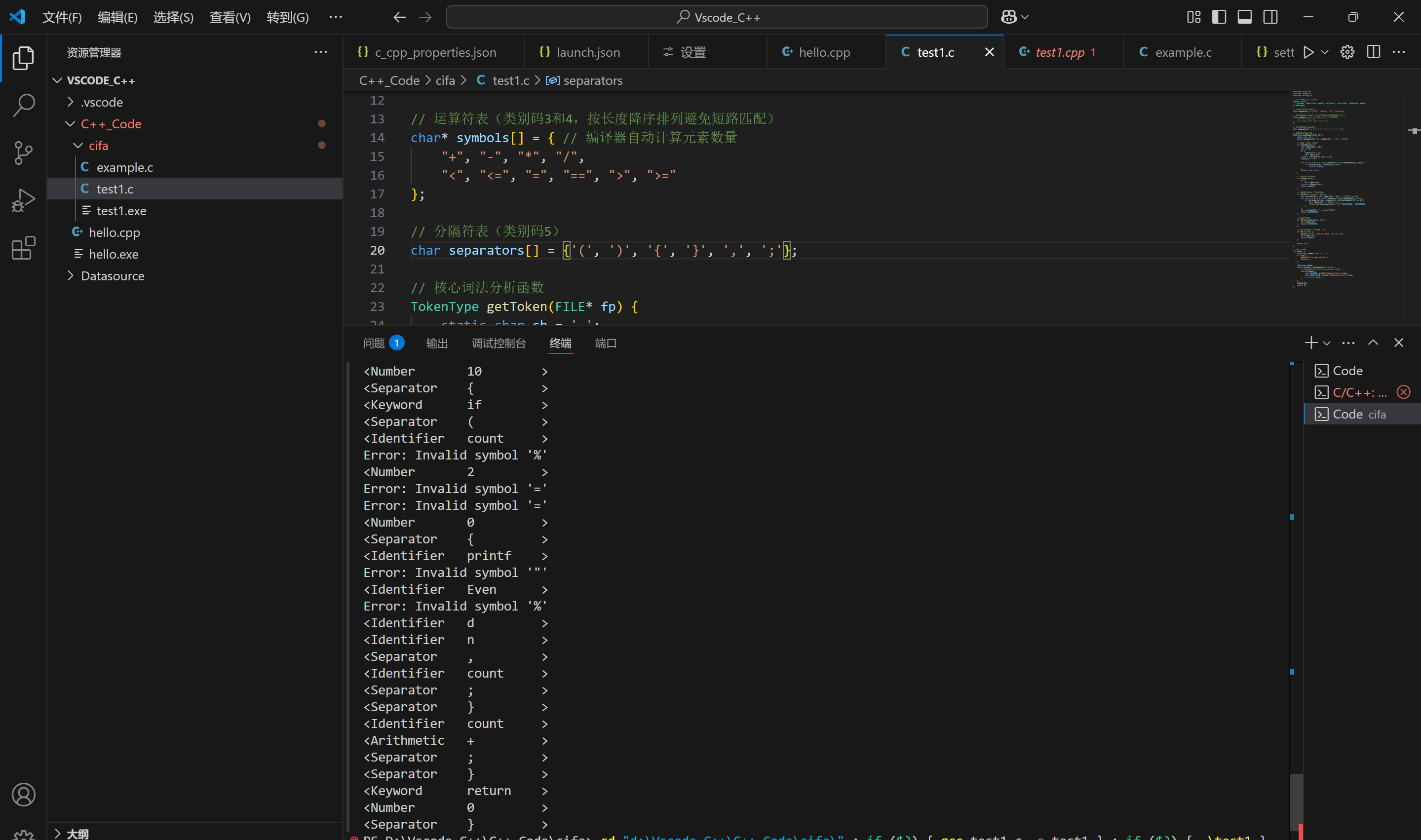
}

count++;

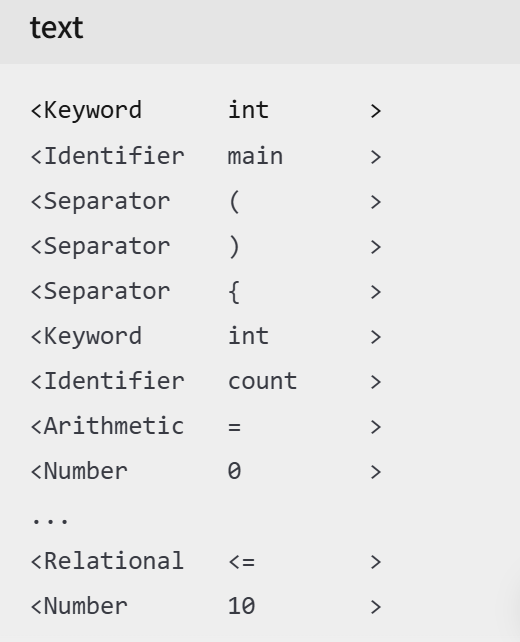
}

return 0;

}



输出：



七．实验总结

通过本次实验，深入理解了词法分析的核心原理，为后续语法分析阶段奠定了坚实基础。