
# 《编译原理与技术》课程实验报告

# 词法分析程序的设计与实现

付容天 学号 2020211616

班级 2020211310

计算机科学与技术系

计算机学院（国家示范性软件学院）

2022 年 9 月 20 日

# 目录

# 1 实验概览

## 1.1 实验介绍

词法分析（Lexical Analysis）是计算机科学中将字符序列转换为单词（Token）序列的过程。进行词法分析的程序或者函数叫作词法分析器（Lexical analyzer，简称 Lexer），也叫扫描器（Scanner）。词法分析器一般以函数的形式存在，供语法分析器调用。完成词法分析任务的程序称为词法分析程序或词法分析器或扫描器。

完成词法分析任务的程序称为词法分析程序或词法分析器或扫描器。从左至右地对源程序进行扫描，按照语言的词法规则识别各类单词，并产生相应单词的属性字。

在本次实验中，北京邮电大学计算机学院（国家示范性软件学院）编译原理与技术课程组的老师设计了“词法分析程序的设计与实现”这一实验任务，让我们可以通过知识的综合运用加深对词法分析工作原理的认识、建立清晰的词法分析概念，从而培养科学研究的独立工作能力、取得工程设计与编程调试的实践经验。

## 1.2 实验任务

本次实验具有以下要求：

- 选定源语言，比如：C、Pascal、Python、Java 等，任何一种语言均可
- 可以识别出用源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
- 可以识别并跳过源程序中的注释
- 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
- 检查源程序中存在的词法错误，并报告错误所在的位置
- 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，即可检查并报告源程序中存在的所有词法错误

并且，在实验方法上，可以选择下面两种方法：

- 方法 1：采用 C/C++ 作为实现语言，手工编写词法分析程序（必做）
- 方法 2：编写 LEX 源程序，利用 LEX 编译程序自动生成词法分析程序

实验报告具有以下要求：

- 内容：
  - 实验题目、要求
  - 程序设计说明
  - 源程序
  - 可执行程序
  - 测试报告：输入、运行结果、分析说明
- 提交：
  - 个人资料打包
  - 命名规则：班级（3 位班号）-学号-姓名
  - 教学云平台上提交

## 1.3 实验成果简述

在本实验中，我完整地完成了既定的实验任务，我的实验成果简要介绍如下：

- **基础词法分析**：在这一部分中，我们小组基于老师给出的基础指令集，拓展了若干实用指令，并按照给定数据格式和数据通路，结合 Altera CPM7128 芯片自行设计并完成了基于硬布线控制器的顺序模型处理机。我们设计的机器支持修改 PC 指针，并且进行了指令拓展，而且引入了流水线思路，优化了程序执行效率
- **简单纠错功能**：我编写的词法分析程序支持有限的自动纠错，这使得词法分析在遇到一些错误的时候可以继续进行下去而不至于崩溃。例如“小数点前没有数字”的错误中，我编写的程序将会自动在小数点前加上数字 0
- **词法分析数据统计**：我编写的词法分析程序可以在分析结束后输出相关统计信息，包括行数、字符总数、各类字符的数量、发现的错误数、程序自动纠正的错误数等
- **用户友好地界面显示**：我编写的词法分析程序利用 SetConsoleTextAttribute 函数给控制台上输出的不同内容赋予了不同颜色，使用户可以更方便地使用，比如：错误信息显示为红色、提示信息显示为蓝色、纠错信息显示为绿色等等

# 2 实验原理、方案设计与代码实现

## 2.1 词法分析介绍

词法分析阶段是编译过程的第一个阶段，是编译的基础。这个阶段的任务是从左到右一个字符一个字符地读入源程序，即对构成源程序的字符流进行扫描然后根据构词规则识别单词（也称单词符号或符号）。词法分析程序实现这个任务。词法分析程序可以使用 LEX 等工具自动生成。

词法分析是编译程序的第一个阶段且是必要阶段；词法分析的核心任务是扫描、识别单词且对识别出的单词给出定性、定长的处理；实现词法分析程序的常用途径：自动生成，手工生成。

## 2.2 利用 C++ 语言进行词法分析的思路

在本实验中，我使用 C++ 语言进行词法分析，对由 C 语言编写的程序 example.c 进行词法分析，并输出相应的结果。根据《编译原理与技术》课本上的伪代码的相关内容，不难发现如果我们要使用 C++ 语言来模拟词法分析的自动机（转移）过程，一个有效的思路就是使用循环结构（如 for 循环）和分支结构（如 switch 结构）来进行模拟，同时考虑当前读入的字符，由此决定自动机的转移方向，通过设计一个全局变量 state 来标识当前自动机处于什么样的状态以及下一次循环时自动机应进入什么样的状态。

## 2.3 词法分析自动机的构造与代码实现

为了用 C++ 语言模拟词法分析过程，我必须首先构造出详细的词法分析自动机，总体来看，词法分析自动机如下图所示：

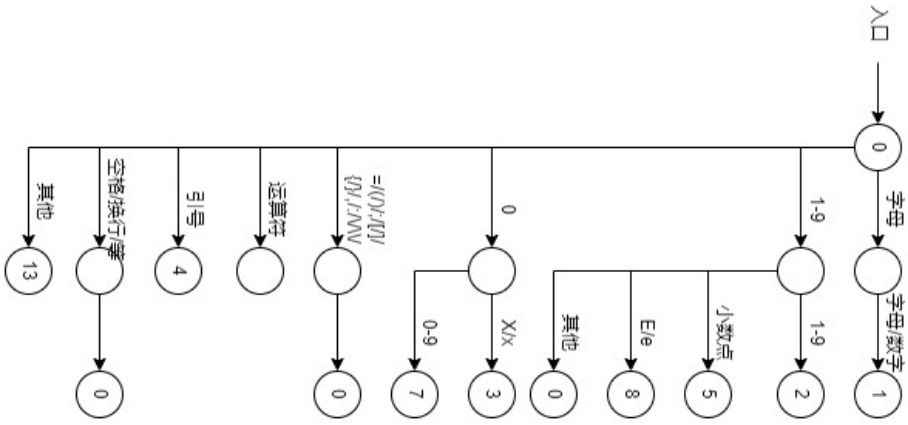


图 1: 词法分析自动机（略去细节，只看第一次状态识别与转移的部分）

上图所示的自动机通过检测当前输入从而进行转移，在自动机进入到不同

状态的时候，会自动执行对应情况下的“子自动机”，比如，“1”状态表示当前读入的字符为“标识符”的一部分，进一步，“1”状态下的自动机为：

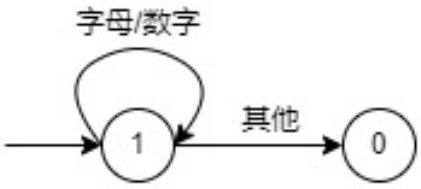


图 2: “1”状态下的自动机

根据图 2 所示的自动机，可以写出如下代码：

```
------------------------------state=1 BEGIN----------------------------
        output_table[table_iterms_num].sign += c; //更新符号表
        if(!is_letter(c_next) && !is_digit(c_next)){
                state = 0;
                print(1, i, j);
        }
------------------------------state=1 END------------------------------
```

剩下所有状态的实现思路都是类似的（即：先设计出自动机、再使用 C++ 语言中的语法结构来模拟自动机的识别的转移过程），比如，“0”状态意味着当前需要读取一个字符，这是所有状态中最为复杂的，因为其直接决定了接下来进入什么样的状态、识别什么样的单词（如图 1 所示），我们可以编写如下代码来实现“0”状态的自动机转换：

```
------------------------------state=0 BEGIN----------------------------
//c 为当前读取的字符 c_next 为当前读取的字符的下一个字符
if(is_letter(c)){
        output_table[table_iterms_num].sign += c;
        if(is_letter(c_next) || is_digit(c_next))
        state = 1;
        else{
                state = 0;
                print(1, i, j);
        }
}
else if(is_digit(c)){
        output_table[table_iterms_num].sign += c;
        if(c == '0'){
                if(c_next == 'X' || c_next == 'x'){ //十六进制数
                        state = 3;  //state=3 状态处理十六进制数
                }
```

```
                else if(is_digit(c_next)){
                        if(c_next == '8' || c_next == '9') is_wrong_octnum = 1;
                        state = 7;  //state=7 状态处理八进制数
                }
        }
        if(is_digit(c_next))
        state = 2; //十进制数
        else if(c_next == '.')
        state = 5; //小数
        else if(c_next == 'E' || c_next == 'e')
        state = 8; //科学计数法
        else{
                state = 0; //仅有一位数
                print(3, i, j);
        }
}
else if(c == '=' || c == '(' ||... 此处略去...|| c == '\\'){
        output_table[table_iterms_num].sign += c;
        state = 0;
        if(c == '=' && c_next == '='){
                output_table[table_iterms_num].sign += c_next;
                j++;
                print(5, i, j);
        }
        else
        print(4, i, j);
}
else if(c == '+' || c == '-' ||... 此处略去...|| c == '.'){
        if(c == '.' && is_digit(c_next)){
                output_error(" 小数点前没有数字", i, j);
                output_table[table_iterms_num].sign += '0'; //自动纠正
                output_table[table_iterms_num].sign += c;
                SetConsoleTextAttribute(hout, FOREGROUND_GREEN);
                printf("\n该错误被修正!\n\n");
                SetConsoleTextAttribute(hout, ... 此处略去...);
                solved_num++;
                state = 5;
        }
        else if((c == '+' ||... 此处略去...|| c == '|') && c_next == '='){
                output_table[table_iterms_num].sign += c;
                output_table[table_iterms_num].sign += c_next;
                j++;
                state = 0;
                print(5, i, j);
```

```
        }
        else if(c == '<'){
                output_table[table_iterms_num].sign += c;
                if(c_next == '>' || c_next == '=' || c_next == '<'){
                        output_table[table_iterms_num].sign += c_next;
                        j++;
                }
                if(c_next == '<' && buff[i][j + 1] == '='){
                        output_table[table_iterms_num].sign += buff[i][j+1];
                        j++;
                }
                state = 0;
                print(5, i, j);
        }
        else if(c == '>'){
                output_table[table_iterms_num].sign += c;
                if(c_next == '=' || c_next == '>'){
                        output_table[table_iterms_num].sign += c_next;
                        j++;
                }
                if(buff[i][j + 1] == '=' && c_next == '>'){
                        output_table[table_iterms_num].sign += buff[i][j + 1];
                        j++;
                }
                state = 0;
                print(5, i, j);
        }
        else if((c == '&' && c_next == '&') || (c == '|' && c_next == '|')){
                output_table[table_iterms_num].sign += c;
                output_table[table_iterms_num].sign += c_next;
                j++;
                state = 0;
                print(5, i, j);
        }
        else{
                output_table[table_iterms_num].sign += c;
                state = 0;
                print(5, i, j);
        }
}
else if(c == '"'){
        output_table[table_iterms_num].sign += c;
        is_string = 1;
        state = 4; // "" 中的字符串
```

```
        print(1, i, j);
}
else if(c == ' ' || c == '\0' || c == '\r' || c == '\n') return;
else{
        output_table[table_iterms_num].sign += c;
        output_error(" 出现未知字符", i, j);
        state = 13;
}
-----------------------------state=0 END-----------------------------
```

这样，我们就可以完整地用 C++ 语言实现自动机的转换过程。

下面来考虑如何构造出自动机本体。除了实现每个状态的识别与转换过程以外，一个很重要的内容就是实现语法分析程序中的“循环-检测-分支”这一循环。对于自动机本体（这是分析程序 analyse_buff 的主干部分），可以编写出如下代码：

```
-----------------------------whole BEGIN-------------------------------
int flag = 0;                 // 用于判断当前读入字符所属的结构
int is_string = 0;            // 是否是字符串
int is_wrong_octnum = 0;      // 是否是错误的八进制数
int state = 0;                // 表示自动机的当前状态

cout << " 词法分析开始" << endl << endl;
        for(int i=0; i<buff_num; i++){ // 依次对每一行进行分析
        if(flag == 1) flag = 0;
        state = 0;

        for(int j=0; j<int(strlen(buff[i])); j++){
                if(is_string == 0)
                        flag = is_comment(strlen(buff[i]), i, &j, flag);
                if(flag == 0){
                        char c = buff[i][j];
                        char c_next = buff[i][j+1];

                        switch(state){ //根据当前 state 进入不同状态
                                case 0: state_0(必要参数); break;
                                case 1: state_1(必要参数); break;
                                case 2: state_2(必要参数); break;
                                case 3: state_3(必要参数); break;
                                case 4: state_4(必要参数); break;
                                case 5: state_5(必要参数); break;
                                case 6: state_6(必要参数); break;
                                case 7: state_7(必要参数); break;
                                case 8: state_8(必要参数); break;
```

```
                                case 9: state_9(必要参数); break;
                                case 13: state_13(必要参数); break;
                                default: break;
                            }
                        }
                    }
}
------------------------------whole END--------------------------------
```

## 2.4 其他功能的设计与实现

### 2.4.1 符号表数据结构

在本次实验中,我设计了合理的符号表的数据结构,命名为OUTPUT_TABLE,并使用 C++ 语言中的 struct 进行了实现，其内容包括：

- string 类型的 sign：符号的具体内容
- int 类型的 property：符号的属性
- int 类型的 i 和 j：符号出现的行数（i）和结束的列数（j）

### 2.4.2 程序文件读取功能

我在 main 函数的一开始内嵌了程序文件的读取功能，通过将程序文件读取为 char 类型的二维数组 buff[MAX_BUFF][MAX_BUFF]，将欲进行词法分析的程序存到缓存中。

读取文件的细节为：首先创建一个 ifstream 类型的变量用以标识位于路径 path 处的程序文件，然后使用 while 循环不断读入文件内容至二位数组 buff 中，直到程序结束。代码如下所示：

```
------------------------------读文件 BEGIN----------------------------
ifstream input_file(path, ios::in|ios::binary);
while(input_file.getline(buff[buff_num], MAX_LINE_LEN)) buff_num++;
if(buff_num==0) cout << " 文件不存在或出错!" << endl;
input_file.close(); // 关闭输出文件
------------------------------读文件 END------------------------------
```

## 2.5 字符属性判断设计与实现

在词法分析的过程中，往往需要判断当前字符的属性，用以指导自动机状态的转移。在我的设计方案中，我设计了三个函数分别判断字符的三种属性：

- int 类型的 is_letter(char c) 函数：用来判断 c 是否为“字母”，注意此处的“字母”包括了下划线
- int 类型的 is_digit(char c) 函数：用来判断 c 是否为“数字”
- int 类型的 is_comment(int buff_size, int i_line, int *pos, int flag) 函数：用来判断从 i_line 行 pos 位置开始的内容是否为注释内容

下面是这三个函数的代码实现：

```
------------------------------属性判断 BEGIN------------------------------
int is_letter(char c){
        if((c>='a'&&c<='z') || (c>='A'&&c<='Z') || c=='_') return 1;
        else return 0;
}

int is_digit(char c){
        if(c>='0'&&c<='9') return 1;
        else return 0;
}

int is_comment(int buff_size, int i_line, int *pos, int flag){
        int j = *pos;
        // '///'形式的单行注释
        if(j+1<buff_size&&buff[i_line][j]=='/'&&buff[i_line][j+1]=='/'){
                printf("\nThe \"//\" comment starts in: %d:%d\n\n", i_line, j);
                return 1;
        }
        // '/*'形式的注释开头
        else if(j+1<buff_size&&buff[i_line][j]=='/'&&buff[i_line][j+1]=='*'){
                printf("\nThe \"/*\" comment starts in: %d:%d\n\n", i_line, j);
                return 2;
        }
        // '*/'形式的注释结尾
        else if(j+1<buff_size&&buff[i_line][j]=='*'&&buff[i_line][j+1]=='/'){
                printf("\nThe \"/*\" comment ends in: %d:%d\n\n", i_line, j);
                *pos += 2;
                return 0;
        }
        return flag;
}
------------------------------属性判断 END------------------------------
```

## 2.6 中间信息输出设计

在词法分析的过程中，需要输出中间信息从而帮助使用者了解词法分析程序进行到的步骤。在我编写的词法分析程序中，中间信息输出主要包括：

- 当前符号的序号
- 当前符号的内容
- 当前符号的属性
- 注释开始与结束信息

如下图所示：

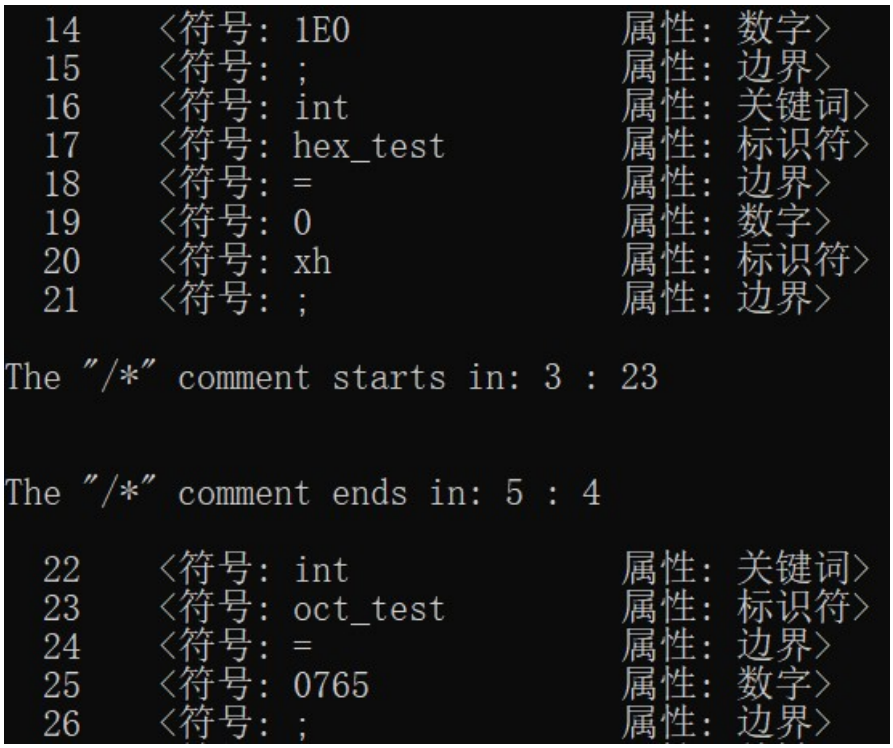


图 3: 中间信息输出示例

## 2.7 错误信息收集

在词法分析程序中，如果检测到当前输入存在错误，那么通常会输出错误出现的位置，在我编写的错误信息收集功能中，具有下面的特点：

- 错误信息通过红色进行表述
- 包括错误类型以及所在的行和列（通过“?”标识错误所在的列）

- 通过一个函数实现，错误类型的描述作为一个字符串变量传入该函数

比如，下图显示了“超过一个小数点”错误的输出情况：

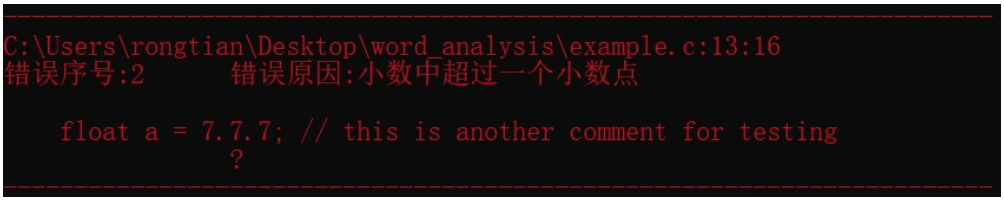


图 4: 错误信息输出示例

## 2.8 自动纠错功能

对于一些常见的错误，我编写的程序支持将其进行自动纠正，主要支持自动修正以下错误：

- 小数点前没有数字：自动加上 0
- 小数点后没有数字：自动加上 0
- 0 开头但是出现了 8 或 9 的数字：删除 0，将其当成非八进制数处理
- 指数部分没有数字：自动加上 0

## 2.9 统计信息收集

为了直观地观察到词法分析程序的分析结果，我在程序中嵌入了统计信息收集的功能，具体的的方案是通过设置合理的全局变量，并在程序的相应位置进行调用和更新，从而在词法分析结束之后可以同时看到收集到的统计信息。我设计的全局变量主要包括：

- OUTPUT_TABLE 类型的 output_table[MAX_WORDS]：词法分析输出表
- int 类型的 table_iterms_num：词法分析输出表中元素个数
- char 类型的 buff[MAX_BUFF][MAX_BUFF]：读入字符的缓存
- int 类型的 buff_num：缓存中字符个数
- int 类型的 key_num：关键字个数
- int 类型的 identifier_num：标识符个数
- int 类型的 number_num：数字个数

- int 类型的 operator_num：操作符个数
- int 类型的 string_num：字符串个数
- int 类型的 error_num：错误个数
- int 类型的 solved_num：解决的错误个数

并且，统计信息在控制台上的颜色被设置为黄色，如下图所示：

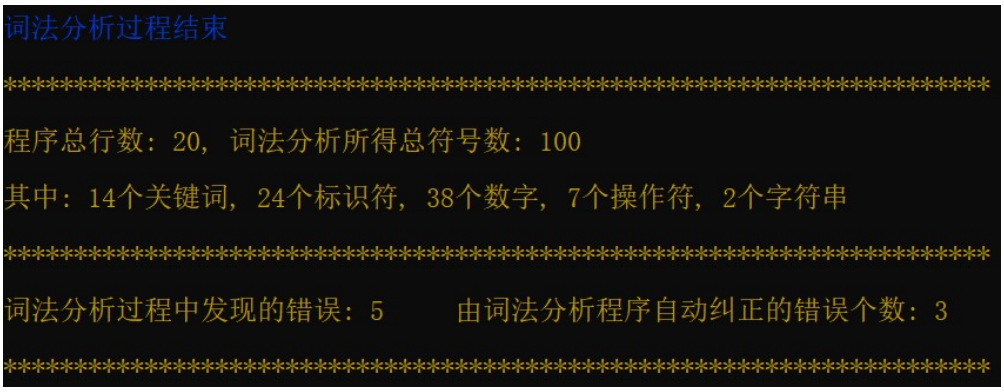


图 5: 统计信息输出示例

# 3 样例设计与测试结果

## 3.1 测试用例设计

我为编写的词法分析程序设计了简单的 C 语言测试用例，命名为 example.c，具体内容如下：

```
------------------------------测试用例 BEGIN------------------------------
int main(){
        int num_test = 1E;
        int hex_test = 0xh;/*
        this is a comment for testing
        */
        int oct_test = 0765;
        int aaa = 09;
        int bbb = 0x1a;
        int ccc = 1.39E1;
        for(int i=0; i<10; i++)
        int ans = aa + bb;
        printf("hello");
        float a = 7.7.7; // this is another comment for testing
```

```
    float b = .1;
    float c = 1.;
    printf("///**/");

    return 0;
}
```

------------------------------测试用例 END--------------------------------

## 3.2 测试结果展示

经过词法分析程序之后，输出的测试结果如下所示：


图 6: 结果 1

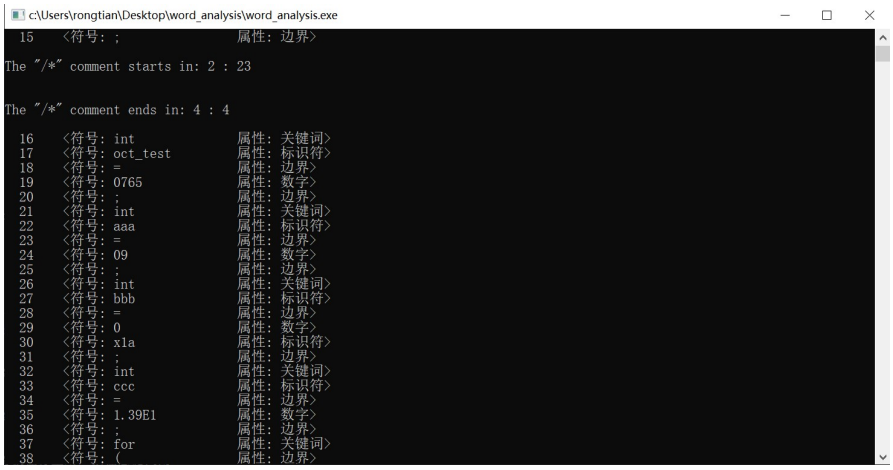


图 7: 结果 2

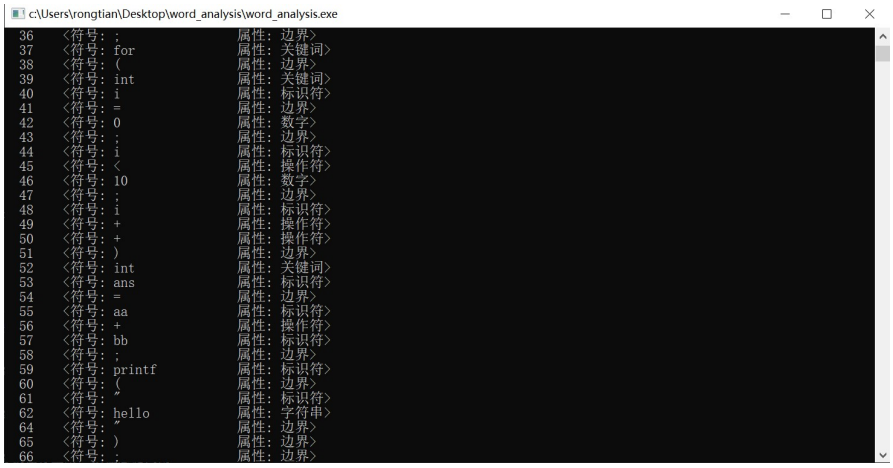


图 8: 结果 3

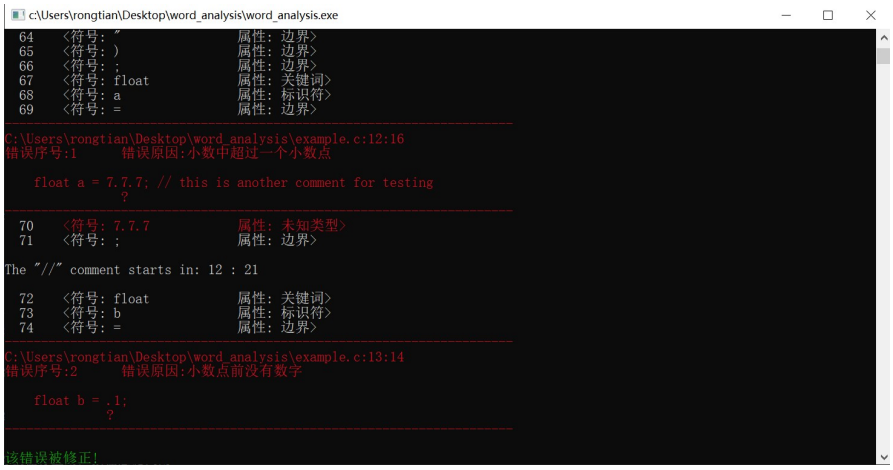


图 9: 结果 4

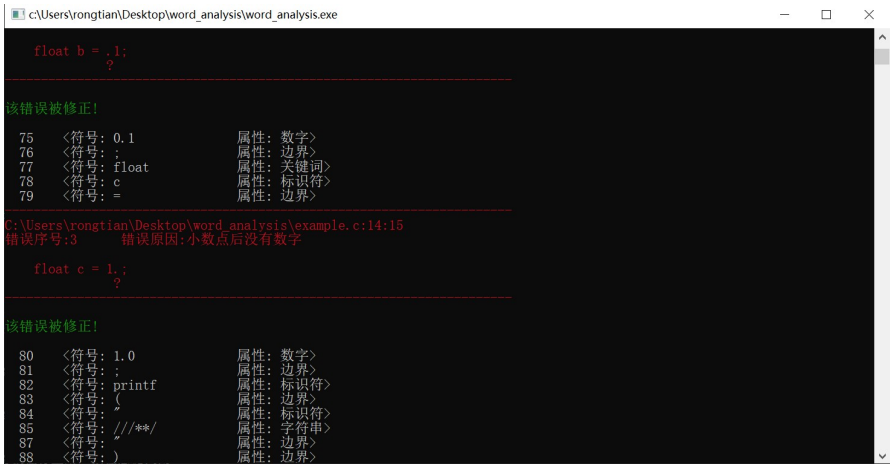


图 10: 结果 5

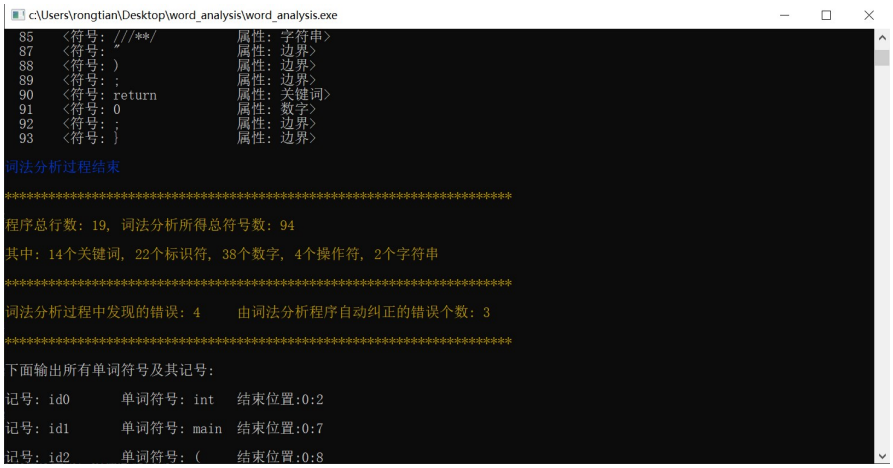


图 11: 结果 6

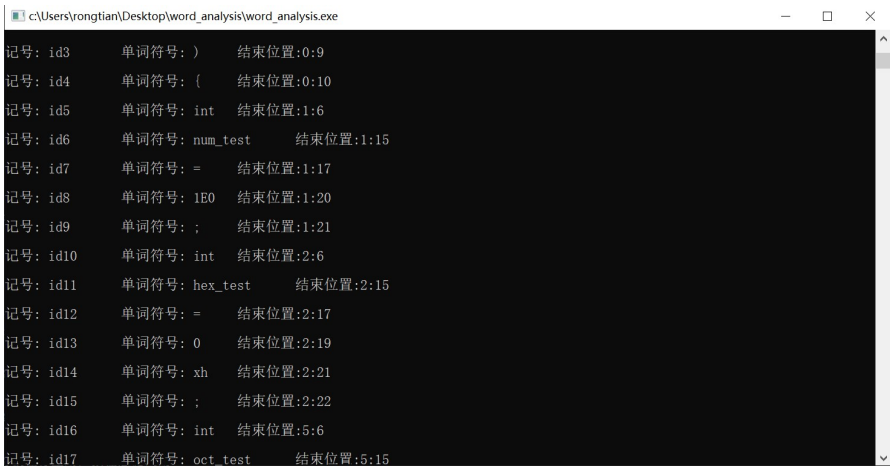


图 12: 结果 7

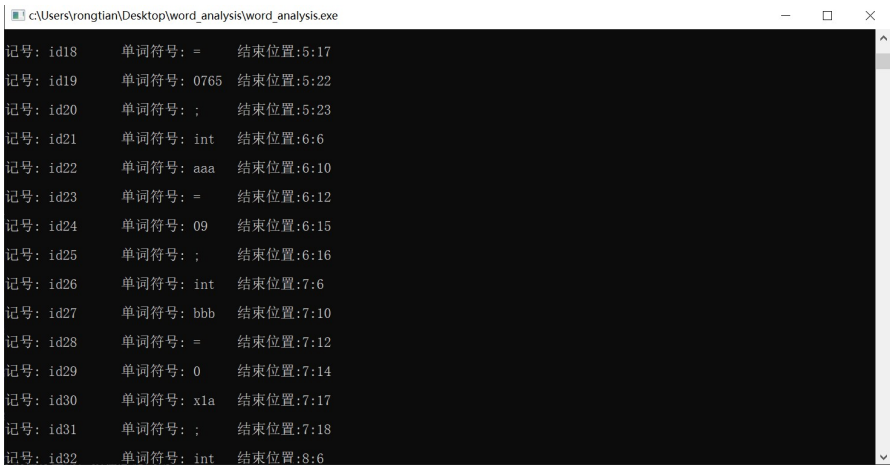


图 13: 结果 8

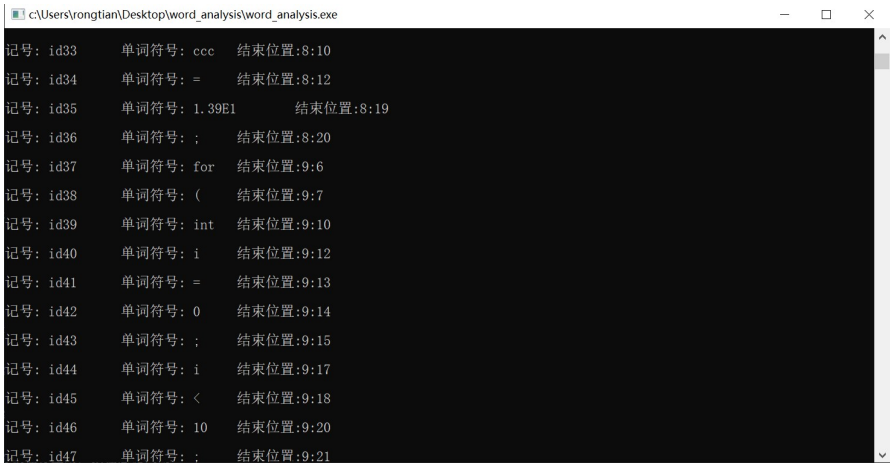


图 14: 结果 9

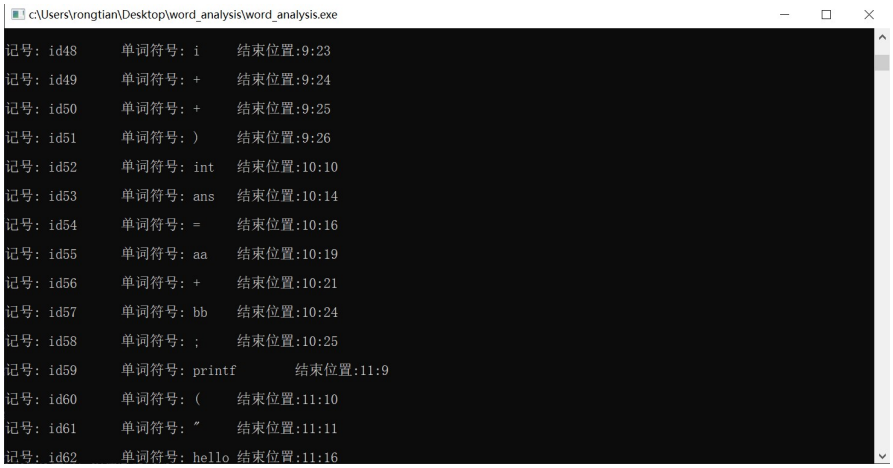


图 15: 结果 10

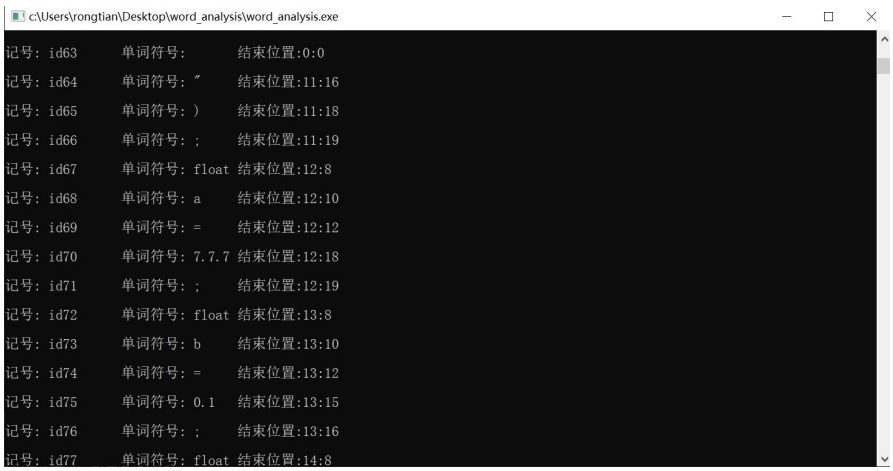


图 16: 结果 11

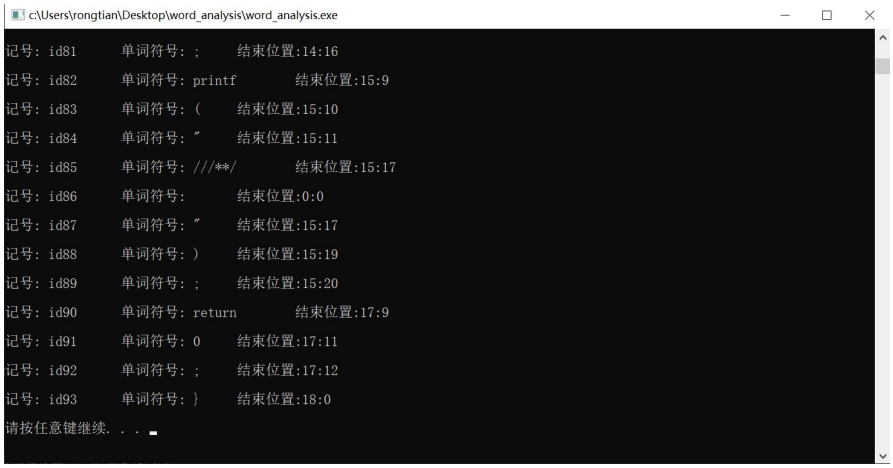


图 17: 结果 12

# 4 实验总结与心得

在本次实验中，我完整了词法分析程序的设计与编写，先后攻克了基础知识不熟、C++ 语言部分特性遗忘、理论设计与代码实现之间的差异等诸多问题，最终完成了既定的实验任务。在本次实验期间我复习了基础知识、完成了理论设计、编写了代码实现、经历了代码调试，收获颇多。

在本次课程设计的过程中，我的心情有高潮、也有低谷。一开始拿到词法分析的题目时，我认为这个题目是十分浅显易懂的。对于这种一眼看懂的实验题目，一开始我并没有放在心上，认为我一定可以很快做出来。但是随着实验的不断进行，我对于本次实验认识也不断深入，渐渐认识到设计到该实验的理论设计和代码实现具有同等重要的地位，在理论上我们可以很容易地设计出指令周期流程图，但是涉及到具体的代码实现，则需要不断编写与调试，并据此改进已经设计出来的自动机、完善程序漏洞。这让我知道了做什么都要从实际出发、从实际入手，只有真正理解了理论与实践的所有细节，才算真正地完成这个实验。