



**实验报告： 词法分析程序的设计与实现**

**——手工实现**

**学院：计算机学院（国家示范性软件学院）**

**专业： 计算机科学与技术**

**班级： 2022211305**

**学号： 2022211683**

**姓名： 张晨阳**

**2024年10月9号**

**目录**

[1 实验概述 1](#_Toc179485121)

[1.1 实验内容及要求 1](#_Toc179485122)

[1.2 实验方法要求 1](#_Toc179485123)

[1.3 实验环境说明 1](#_Toc179485124)

[2 程序设计说明 2](#_Toc179485125)

[2.1 程序结构设计概述 2](#_Toc179485126)

[2.1.1 token.h & token.cpp 2](#_Toc179485127)

[2.1.2 error.h & error.cpp 2](#_Toc179485128)

[2.1.3 mylex.h & mylex.cpp 3](#_Toc179485129)

[2.1.4 main.cpp 3](#_Toc179485130)

[2.2 程序具体逻辑说明 4](#_Toc179485131)

[2.2.1 辅助函数 4](#_Toc179485132)

[2.2.2 注释及空字符 5](#_Toc179485133)

[2.2.3 标识符和关键字 5](#_Toc179485134)

[2.2.4 数字 6](#_Toc179485135)

[2.2.5 字符和字符串 6](#_Toc179485136)

[2.2.6 运算符和标点 7](#_Toc179485137)

[2.3 两种版本程序对比 8](#_Toc179485138)

[3 测试设计与分析 9](#_Toc179485139)

[3.1 测试方法 9](#_Toc179485140)

[3.2 test1.c 10](#_Toc179485141)

[3.3 test2.c 15](#_Toc179485142)

[3.4 test3.c 20](#_Toc179485143)

[4 总结 24](#_Toc179485144)

# 1 实验概述

## 实验内容及要求

1. 选定源语言，比如：C、Pascal、Python、Java 等，任何一种语言均可；
2. 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。
3. 可以识别并跳过源程序中的注释。
4. 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。
5. 检查源程序中存在的词法错误，并报告错误所在的位置。
6. 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。

## 实验方法要求

采用 C/C++ 作为实现语言，手工编写词法分析程序。

## 实验环境说明

* Windows 11
* Visual Studio Code

# 2 程序设计说明

## 2.1 程序结构设计概述

根据编写LEX版本的词法分析器的理解，我将手工版本（C++）的词法分析器分为了：token部分（token.h & token.cpp）、error部分（error.h & error.cpp）、mylex部分（mylex.h & mylex.cpp）、main部分（main.cpp）。

各部分功能概述如下：

### token.h & token.cpp

token.h 和 token.cpp 主要负责定义和实现与词法单元（token）相关的结构和方法。

**token.h**：

定义了词法分析过程中使用的 TokenType 枚举类型（如标识符、关键字、整数常量、字符常量等）。此外，定义了 Token 类来表示每个词法单元，包含token类型、文本内容、所在行号和列号等信息。

**token.cpp**：

实现了 Token 类的具体功能，例如构造函数、获取token属性的方法，以及用于调试或输出token信息的方法。

这部分的功能是将词法分析得到的字符序列标识为具体的token，并通过 Token 对象传递给分析器的其他部分。

### error.h & error.cpp

error.h 和 error.cpp 负责处理词法分析过程中的特殊错误情况。

定义了 ErrorHandler 类，该类提供了报告错误的接口。当遇到不需要返回token的错误（如注释错误，因为注释需要跳过，故不需要返回token）时，调用该接口报告错误。

### 2.1.3 mylex.h & mylex.cpp

mylex.h 和 mylex.cpp 是词法分析器的核心部分，负责实现词法分析的具体逻辑。

**mylex.h**：

定义了 Lexer 类及其主要成员函数。Lexer 类包含解析源代码的功能，能够从输入字符串中逐字符分析并生成对应的 Token 对象。同时定义了一些辅助函数，用于处理标识符、数字、字符串、注释等。

**mylex.cpp**：

实现了 Lexer 类的具体逻辑。包括跳过空白和注释、识别标识符、常量和标点符号，检测非法字符等。每次调用 getNextToken() 函数时，Lexer 会从输入中获取下一个token，返回给调用方。同时，Lexer 还负责记录每个token的数量和处理字符的总数，能够输出详细的统计信息。

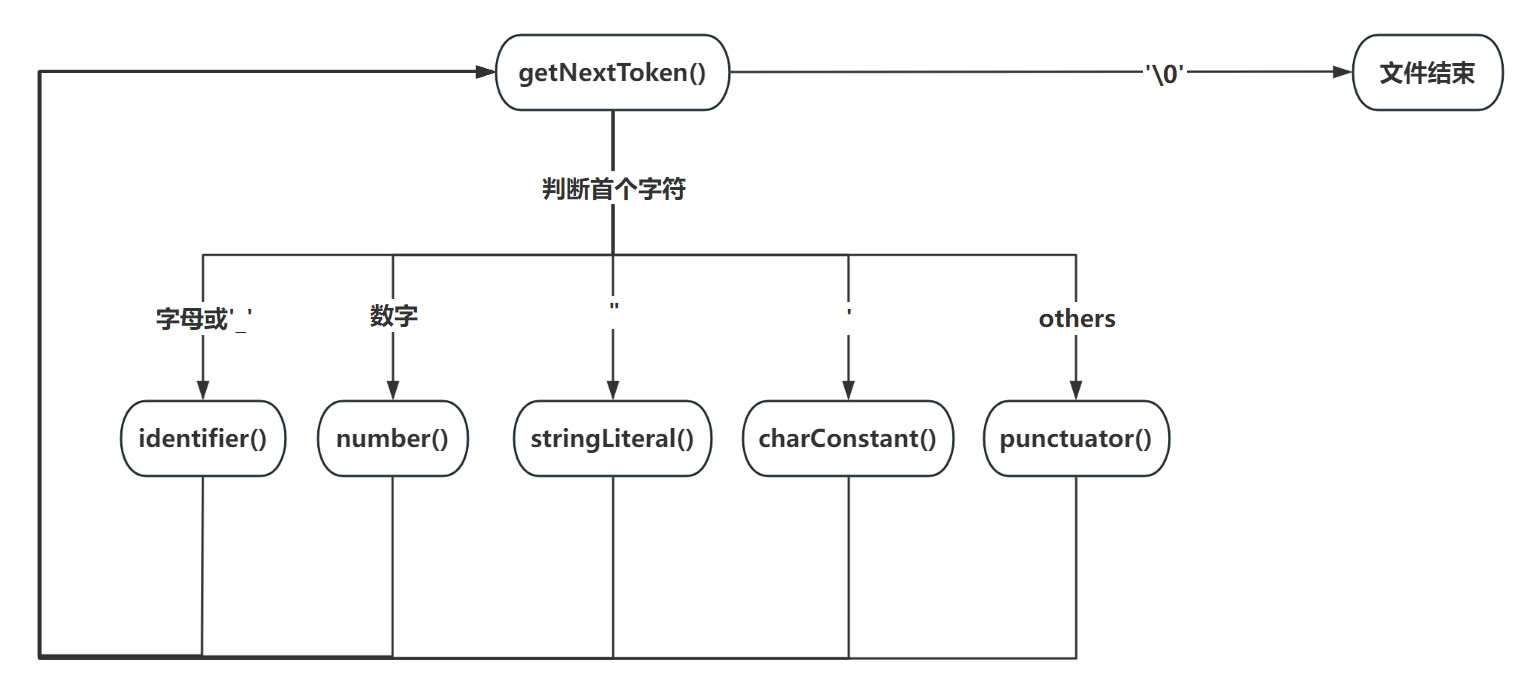
这一部分是词法分析器的主要实现，负责根据源代码生成token序列。

### 2.1.4 main.cpp

main.cpp 是程序的入口，负责将各个模块整合在一起，执行词法分析器的主流程。

## 2.2 程序具体逻辑说明

mylex.cpp 是词法分析器的核心部分，负责逐字符扫描输入源代码，识别并生成对应的 Token 对象。主要逻辑如下图：



具体代码见源文件。

以下将分模块解释该文件的主要逻辑。

**注意：**由于实现方法以及C++语言的特性，有些token种类无法画出状态转换图，在这里我只提供了识别标点、运算符的**状态转换图**。

### 2.2.1 辅助函数

在我实现的词法分析器中，辅助函数 peek() 和 advance() 是分析器执行字符操作的基础。

**peek()**：

该函数返回当前 position 处的字符，但不移动字符指针 position。通过它可以检查当前字符的值，用于决定接下来的解析逻辑。

**advance()**：

该函数不仅返回当前 position 处的字符，还会将 position 向前移动，更新当前的行号 line 和列号 column，并更新总字符计数 totalCharacters。通过它，分析器能够逐步前进，读取下一个字符。

### 注释及空字符

**skipWhitespaceAndComments()**：该函数的作用是跳过源代码中的空白字符、单行注释和多行注释。

**空白字符**（如空格、制表符和换行符）通过循环调用 advance() 跳过处理。

**单行注释：**遇到 // 时，忽略其后直到换行符的所有字符。

**多行注释：**遇到 /\* 时，忽略其后直到找到结束符 \*/。如果注释未闭合，报错 "Unterminated block comment"。

该函数通过识别和跳过无关字符，确保词法分析器只处理实际代码部分，避免空白字符和注释干扰分析。

### 标识符和关键字

**identifier()**：标识符和关键字的识别逻辑在此函数中实现。

当 peek() 返回字母或下划线时，identifier() 函数开始识别标识符，读取所有由字母、数字或下划线组成的字符，形成完整的标识符。

识别完标识符后，函数检查该标识符是否是C语言中的关键字。

实现方法为：列出所有的关键字，进行一对一的比对。

如果是，则将其识别为关键字 T\_KEYWORD；否则将其识别为普通标识符 T\_IDENTIFIER。

例如：int 作为关键字处理，而 variable\_name 则作为标识符处理。

该函数帮助词法分析器区分用户定义的标识符和C语言内置的关键字。

### 数字

我实现的词法分析器能够识别整数、八进制数、十六进制数以及浮点数。数字的识别逻辑主要通过以下几个函数实现。

**number()**：该函数处理十进制数字，首先从当前字符开始，读取所有数字字符。如果遇到小数点或科学计数法标志（e 或 E），则调用 floatNumber() 处理浮点数。

**octalNumber()**：处理以 0 开头的八进制数字。在识别过程中，若发现非法八进制（例如 09 中的 9 是非法的八进制数字），则报错，并继续解析后续数字。

**hexNumber()**：识别以 0x 或 0X 开头的十六进制数，允许出现数字 0-9 和字母 a-f 或 A-F。

**floatNumber()**：该函数处理浮点数，包括小数点和科学计数法表示。如果发现非法的小数或科学计数法表示，返回错误。

### 字符和字符串

**stringLiteral()** 和 **charConstant()** 负责处理源代码中的字符串和字符常量。

**stringLiteral()**：识别由双引号包围的字符串字面量，并处理常见的转义字符（如 \n 表示换行符）。如果字符串缺少关闭的双引号，则报错 "Unterminated string literal"。

**charConstant()**：处理单引号包围的字符常量，包括转义字符。如果字符常量长度非法（例如 'ab' 或 '\\'），则返回错误。

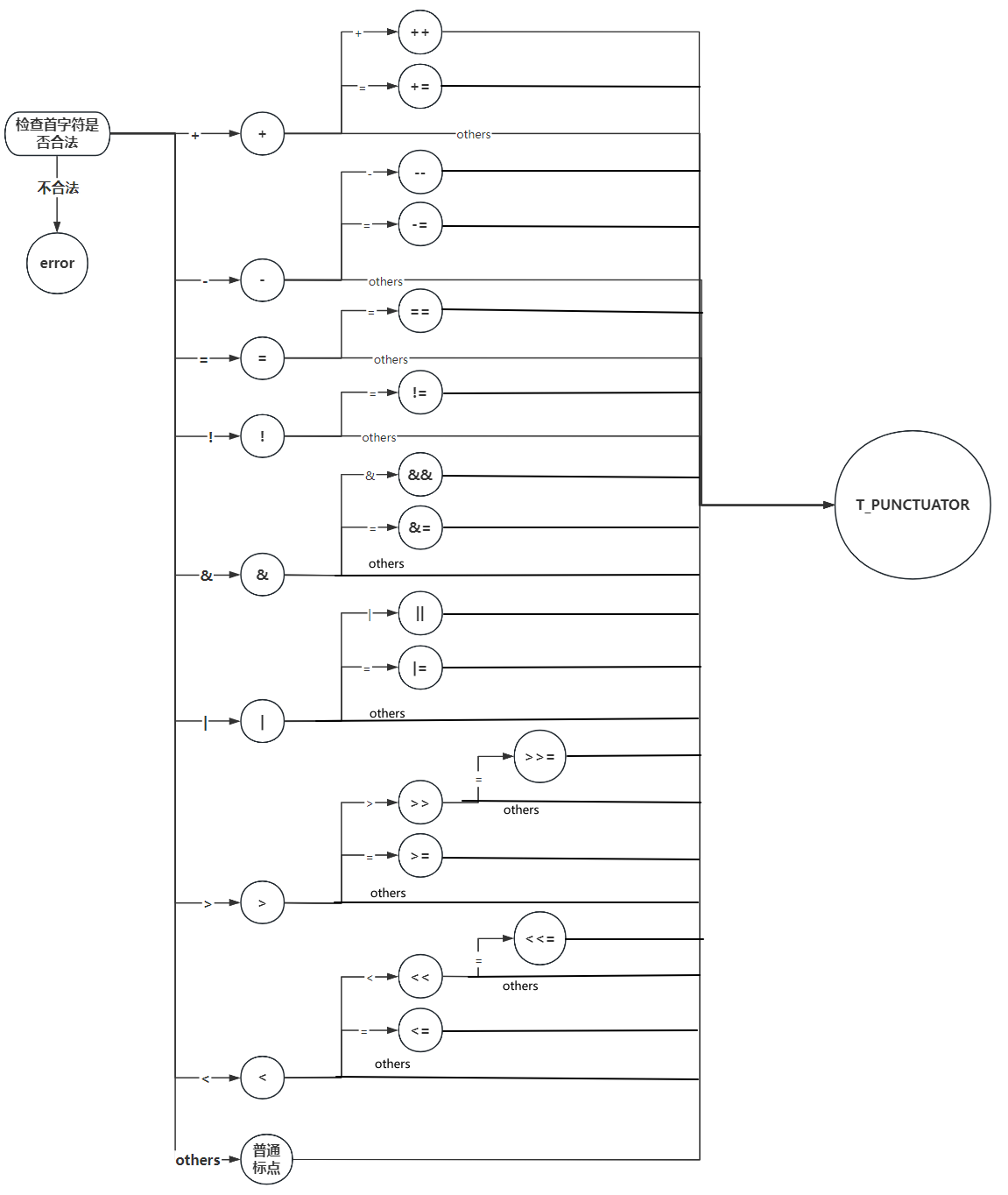
### 运算符和标点

**punctuator()**该函数用于处理源代码中的运算符和标点符号。

首先，它检查当前字符是否为合法的C语言标点符号。然后处理组合运算符（如 +=，!=，<<= 等），并返回相应的token。

如果遇到非法的标点符号，函数会返回错误。

具体的状态转换图如下：



## 2.3 两种版本程序对比

经过三种测试对比，两种版本的词法分析器都有各自的优缺点。

因为在实现的过程中，两种版本我都准备实现同样的功能，所以可以进行横向对比。

具体的测试结果可见两份文档的测试部分。对比结论如下：

1. **错误注释的识别**

在LEX版本中，由于正则表达式的限制，我假设错误的多行注释一定是两行的注释，其他行数没有进行处理；

而在C++版本中，对于任意行数的多行注释错误都可以识别。

1. **错误浮点数的识别**

在LEX版本中，由于正则表达式的不够完善，在识别出错误的浮点数后，会继续错误的将 ; 认为是错误的内容，导致少识别一个标点符号；

而在手工版本中不存在这个问题。

1. **科学计数法的识别**

LEX版本无法正确的识别科学计数法格式的浮点数；

C++版本可以。

1. **调试过程**

LEX版本的调试大部分都是优化正则表达式，比较单一，只要正则表达式写的足够完美即可。

而在手工版本中，错误的识别、各种bug都可能来自各种问题，需要对自己的代码足够熟悉，一步一步的进行调试修改。

1. **代码逻辑**

正则表达式逻辑并不如直接的代码逻辑清晰，导致优化、添加新功能时较为困难；

C++版本逻辑清晰，想要添加新功能只需要添加新的函数、接口即可，更便于后续版本迭代。

# 3 测试设计与分析

## 3.1 测试方法

**1. 编译可执行文件**

在源代码所在路径下输入编译命令：

> g++ -o test token.cpp error.cpp mylex.cpp main.cpp

生成test.exe可执行文件

**2. 运行测试用例**

在相同路径下输入测试命令：

> test test1.c

即运行可执行文件，后面接需要进行测试的案例 .c文件。

## 3.2 test1.c

test.1主要用于测试C语言的关键字、注释、常见符号等内容，不包含任何词法错误。内容如下：

1. #include <stdio.h>

2.

3. // This is a single line comment

4.

5. /\*

6.  \*  This is a multi-line comment

7.  \*  It spans multiple lines

8.  \*/

9.

10. int main() {

11.     int a = 10;          // This is an integer

12.     float b = 20.5;      // This is a floating point number

13.     char c = 'a';        // This is a character

14.     double d = 30.5e-2;  // This is a double

15.

16.     if (a > 5) {

17.         printf("a is greater than 5\n");

18.     } else {

19.         printf("a is not greater than 5\n");

20.     }

21.

22.     while (a < 20) {

23.         a++;

24.     }

25.

26.     do {

27.         b -= 1.5;

28.     } while (b > 0);

29.

30.     for (int i = 0; i < 10; i++) {

31.         c = 'A' + i;

32.     }

33.

34.     switch (c) {

35.         case 'A':

36.             printf("Uppercase A\n");

37.             break;

38.         case 'a':

39.             printf("Lowercase a\n");

40.             break;

41.         default:

42.             printf("Other character\n");

43.     }

44.

45.     return 0;

46. }

我们编写的词法分析器对test1.c文件的分析结果如下：

10:1: <keyword, int>

10:5: <identifier, main>

10:9: <punctuator, (>

10:10: <punctuator, )>

10:12: <punctuator, {>

11:5: <keyword, int>

11:9: <identifier, a>

11:11: <punctuator, =>

11:13: <integer constant, 10>

11:15: <punctuator, ;>

12:5: <keyword, float>

12:11: <identifier, b>

12:13: <punctuator, =>

12:15: <floating constant, 20.5>

12:19: <punctuator, ;>

13:5: <keyword, char>

13:10: <identifier, c>

13:12: <punctuator, =>

13:14: <char constant, 'a'>

13:17: <punctuator, ;>

14:5: <keyword, double>

14:12: <identifier, d>

14:14: <punctuator, =>

14:16: <floating constant, 30.5e-2>

14:23: <punctuator, ;>

16:5: <keyword, if>

16:8: <punctuator, (>

16:9: <identifier, a>

16:11: <punctuator, >>

16:13: <integer constant, 5>

16:14: <punctuator, )>

16:16: <punctuator, {>

17:9: <identifier, printf>

17:15: <punctuator, (>

17:16: <string literal, "a is greater than 5\n">

17:39: <punctuator, )>

17:40: <punctuator, ;>

18:5: <punctuator, }>

18:7: <keyword, else>

18:12: <punctuator, {>

19:9: <identifier, printf>

19:15: <punctuator, (>

19:16: <string literal, "a is not greater than 5\n">

19:43: <punctuator, )>

19:44: <punctuator, ;>

20:5: <punctuator, }>

22:5: <keyword, while>

22:11: <punctuator, (>

22:12: <identifier, a>

22:14: <punctuator, <>

22:16: <integer constant, 20>

22:18: <punctuator, )>

22:20: <punctuator, {>

23:9: <identifier, a>

23:10: <punctuator, ++>

23:12: <punctuator, ;>

24:5: <punctuator, }>

26:5: <keyword, do>

26:8: <punctuator, {>

27:9: <identifier, b>

27:11: <punctuator, -=>

27:14: <floating constant, 1.5>

27:17: <punctuator, ;>

28:5: <punctuator, }>

28:7: <keyword, while>

28:13: <punctuator, (>

28:14: <identifier, b>

28:16: <punctuator, >>

28:18: <integer constant, 0>

28:19: <punctuator, )>

28:20: <punctuator, ;>

30:5: <keyword, for>

30:9: <punctuator, (>

30:10: <keyword, int>

30:14: <identifier, i>

30:16: <punctuator, =>

30:18: <integer constant, 0>

30:19: <punctuator, ;>

30:21: <identifier, i>

30:23: <punctuator, <>

30:25: <integer constant, 10>

30:27: <punctuator, ;>

30:29: <identifier, i>

30:30: <punctuator, ++>

30:32: <punctuator, )>

30:34: <punctuator, {>

31:9: <identifier, c>

31:11: <punctuator, =>

31:13: <char constant, 'A'>

31:17: <punctuator, +>

31:19: <identifier, i>

31:20: <punctuator, ;>

32:5: <punctuator, }>

34:5: <keyword, switch>

34:12: <punctuator, (>

34:13: <identifier, c>

34:14: <punctuator, )>

34:16: <punctuator, {>

35:9: <keyword, case>

35:14: <char constant, 'A'>

35:17: <punctuator, :>

36:13: <identifier, printf>

36:19: <punctuator, (>

36:20: <string literal, "Uppercase A\n">

36:35: <punctuator, )>

36:36: <punctuator, ;>

37:13: <keyword, break>

37:18: <punctuator, ;>

38:9: <keyword, case>

38:14: <char constant, 'a'>

38:17: <punctuator, :>

39:13: <identifier, printf>

39:19: <punctuator, (>

39:20: <string literal, "Lowercase a\n">

39:35: <punctuator, )>

39:36: <punctuator, ;>

40:13: <keyword, break>

40:18: <punctuator, ;>

41:9: <keyword, default>

41:16: <punctuator, :>

42:13: <identifier, printf>

42:19: <punctuator, (>

42:20: <string literal, "Other character\n">

42:39: <punctuator, )>

42:40: <punctuator, ;>

43:5: <punctuator, }>

45:5: <keyword, return>

45:12: <integer constant, 0>

45:13: <punctuator, ;>

46:1: <punctuator, }>

19 keyword

21 identifier

71 punctuator

7 integer constant

3 floating constant

4 char constant

5 string literal

0 error

total: 130 tokens, 858 characters, 46 lines

从输出结果中我们可以发现，test1.c文件含有130个词，858个字符，46行。其中，含有19个关键词，21个标识符，71个标点，7个整数常量，3个浮点数常量，4个字符常量，5个字符串。

经过比对，该测试完全通过。

## 3.3 test2.c

test.2主要用于测试C语言的十进制、八进制、十六进制的数字等内容，不包含任何词法错误。内容如下：

1. #include <stdio.h>

2.

3. int main() {

4.     // Decimal numbers

5.     int decimal = 100;  // Decimal number

6.

7.     // Octal numbers

8.     int octal = 0144;  // Octal number

9.

10.     // Hexadecimal numbers

11.     int hex = 0x64;  // Hexadecimal number

12.

13.     // Float numbers with different bases

14.     float pi = 3.14159;         // Decimal float

15.     float e = 2.71828;          // Decimal float

16.     float hexFloat = 0x1.2p10;  // Hexadecimal float

17.

18.     // Exponential notation

19.     double exp = 1e10;  // Exponential notation

20.

21.     printf("Decimal: %d\n", decimal);

22.     printf("Octal: %o\n", octal);

23.     printf("Hex: %x\n", hex);

24.     printf("Float in hex: %a\n", hexFloat);

25.     printf("Exponential: %e\n", exp);

26.

27.     // Array with different bases

28.     int bases[] = {10, 07, 0x1A};

29.

30.     // Loop to print array elements

31.     for (int i = 0; i < sizeof(bases) / sizeof(bases[0]); i++) {

32.         printf("Array element in base 10: %d\n", bases[i]);

33.     }

34.

35.     return 0;

36. }

我们编写的词法分析器对test2.c文件的分析结果如下：

3:1: <keyword, int>

3:5: <identifier, main>

3:9: <punctuator, (>

3:10: <punctuator, )>

3:12: <punctuator, {>

5:5: <keyword, int>

5:9: <identifier, decimal>

5:17: <punctuator, =>

5:19: <integer constant, 100>

5:22: <punctuator, ;>

8:5: <keyword, int>

8:9: <identifier, octal>

8:15: <punctuator, =>

8:17: <integer constant, 0144>

8:21: <punctuator, ;>

11:5: <keyword, int>

11:9: <identifier, hex>

11:13: <punctuator, =>

11:15: <integer constant, 0x64>

11:19: <punctuator, ;>

14:5: <keyword, float>

14:11: <identifier, pi>

14:14: <punctuator, =>

14:16: <floating constant, 3.14159>

14:23: <punctuator, ;>

15:5: <keyword, float>

15:11: <identifier, e>

15:13: <punctuator, =>

15:15: <floating constant, 2.71828>

15:22: <punctuator, ;>

16:5: <keyword, float>

16:11: <identifier, hexFloat>

16:20: <punctuator, =>

16:22: <floating constant, 0x1.2p10>

16:30: <punctuator, ;>

19:5: <keyword, double>

19:12: <identifier, exp>

19:16: <punctuator, =>

19:18: <floating constant, 1e10>

19:22: <punctuator, ;>

21:5: <identifier, printf>

21:11: <punctuator, (>

21:12: <string literal, "Decimal: %d\n">

21:27: <punctuator, ,>

21:29: <identifier, decimal>

21:36: <punctuator, )>

21:37: <punctuator, ;>

22:5: <identifier, printf>

22:11: <punctuator, (>

22:12: <string literal, "Octal: %o\n">

22:25: <punctuator, ,>

22:27: <identifier, octal>

22:32: <punctuator, )>

22:33: <punctuator, ;>

23:5: <identifier, printf>

23:11: <punctuator, (>

23:12: <string literal, "Hex: %x\n">

23:23: <punctuator, ,>

23:25: <identifier, hex>

23:28: <punctuator, )>

23:29: <punctuator, ;>

24:5: <identifier, printf>

24:11: <punctuator, (>

24:12: <string literal, "Float in hex: %a\n">

24:32: <punctuator, ,>

24:34: <identifier, hexFloat>

24:42: <punctuator, )>

24:43: <punctuator, ;>

25:5: <identifier, printf>

25:11: <punctuator, (>

25:12: <string literal, "Exponential: %e\n">

25:31: <punctuator, ,>

25:33: <identifier, exp>

25:36: <punctuator, )>

25:37: <punctuator, ;>

28:5: <keyword, int>

28:9: <identifier, bases>

28:14: <punctuator, [>

28:15: <punctuator, ]>

28:17: <punctuator, =>

28:19: <punctuator, {>

28:20: <integer constant, 10>

28:22: <punctuator, ,>

28:24: <integer constant, 07>

28:26: <punctuator, ,>

28:28: <integer constant, 0x1A>

28:32: <punctuator, }>

28:33: <punctuator, ;>

31:5: <keyword, for>

31:9: <punctuator, (>

31:10: <keyword, int>

31:14: <identifier, i>

31:16: <punctuator, =>

31:18: <integer constant, 0>

31:19: <punctuator, ;>

31:21: <identifier, i>

31:23: <punctuator, <>

31:25: <keyword, sizeof>

31:31: <punctuator, (>

31:32: <identifier, bases>

31:37: <punctuator, )>

31:39: <punctuator, />

31:41: <keyword, sizeof>

31:47: <punctuator, (>

31:48: <identifier, bases>

31:53: <punctuator, [>

31:54: <integer constant, 0>

31:55: <punctuator, ]>

31:56: <punctuator, )>

31:57: <punctuator, ;>

31:59: <identifier, i>

31:60: <punctuator, ++>

31:62: <punctuator, )>

31:64: <punctuator, {>

32:9: <identifier, printf>

32:15: <punctuator, (>

32:16: <string literal, "Array element in base 10: %d\n">

32:48: <punctuator, ,>

32:50: <identifier, bases>

32:55: <punctuator, [>

32:56: <identifier, i>

32:57: <punctuator, ]>

32:58: <punctuator, )>

32:59: <punctuator, ;>

33:5: <punctuator, }>

35:5: <keyword, return>

35:12: <integer constant, 0>

35:13: <punctuator, ;>

36:1: <punctuator, }>

14 keyword

27 identifier

69 punctuator

9 integer constant

4 floating constant

0 char constant

6 string literal

0 error

total: 129 tokens, 939 characters, 36 lines

从输出结果中我们可以发现，test2.c文件含有129个词，939个字符，36行。其中，含有14个关键词，27个标识符，69个标点，9个整数常量，4个浮点数常量，6个字符串。

经过比对，该测试完全通过。

## 3.4 test3.c

test3.c主要用于测试C语言的几种词法错误，包括注释未结束、非法的八进制数、非法的字符常量、非法的浮点数常量、非法标识符等。内容如下：

1. /\*

2.  \* test3.c- This program contains several intentional lexical errors

3.  \*           to test the error detection and recovery capabilities

4.  \*           of the lexical analyzer.

5.  \*/

6.

7. int main() {

8.     int number = 123;     // valid integer

9.     float pi = 3.14;      // valid float

10.     char ch = 'a';        // valid character constant

11.     char\* str = "Hello";  // valid string literal

12.

13.     /\* Missing closing comment delimiter \*/

14.     int x = 10;

15.     /\* This is a valid comment but it's incomplete

16.     \*

17.

18.     int y = 020; // valid

19.     char \*z = "abcd";

20.

21.     // Below are some lexical errors

22.

23.     // Invalid: '09' is not a valid octal number

24.     int invalid\_number = 09;

25.

26.     // Invalid: too many characters in character constant

27.     char invalid\_char = 'ab';

28.

29.     // Invalid: incomplete exponent part

30.     float invalid\_float = 1.2e+;

31.

32.     // Invalid: '@' is not allowed in an identifier

33.     int incomplete\_identifier = @var;

34.

35.     return 0;

36. }

我们编写的词法分析器对test3.c文件的分析结果如下：

7:1: <keyword, int>

7:5: <identifier, main>

7:9: <punctuator, (>

7:10: <punctuator, )>

7:12: <punctuator, {>

8:5: <keyword, int>

8:9: <identifier, number>

8:16: <punctuator, =>

8:18: <integer constant, 123>

8:21: <punctuator, ;>

9:5: <keyword, float>

9:11: <identifier, pi>

9:14: <punctuator, =>

9:16: <floating constant, 3.14>

9:20: <punctuator, ;>

10:5: <keyword, char>

10:10: <identifier, ch>

10:13: <punctuator, =>

10:15: <char constant, 'a'>

10:18: <punctuator, ;>

11:5: <keyword, char>

11:9: <punctuator, \*>

11:11: <identifier, str>

11:15: <punctuator, =>

11:17: <string literal, "Hello">

11:24: <punctuator, ;>

14:5: <keyword, int>

14:9: <identifier, x>

14:11: <punctuator, =>

14:13: <integer constant, 10>

14:15: <punctuator, ;>

15:5: <error, Unterminated block comment>

18:5: <keyword, int>

18:9: <identifier, y>

18:11: <punctuator, =>

18:13: <integer constant, 020>

18:16: <punctuator, ;>

19:5: <keyword, char>

19:10: <punctuator, \*>

19:11: <identifier, z>

19:13: <punctuator, =>

19:15: <string literal, "abcd">

19:21: <punctuator, ;>

24:5: <keyword, int>

24:9: <identifier, invalid\_number>

24:24: <punctuator, =>

24:26: <error, Invalid octal number>

24:28: <punctuator, ;>

27:5: <keyword, char>

27:10: <identifier, invalid\_char>

27:23: <punctuator, =>

27:25: <error, Invalid character constant>

27:29: <punctuator, ;>

30:5: <keyword, float>

30:11: <identifier, invalid\_float>

30:25: <punctuator, =>

30:27: <error, Invalid exponent in floating point number>

30:32: <punctuator, ;>

33:5: <keyword, int>

33:9: <identifier, incomplete\_identifier>

33:31: <punctuator, =>

33:33: <error, @>

33:34: <identifier, var>

33:37: <punctuator, ;>

35:5: <keyword, return>

35:12: <integer constant, 0>

35:13: <punctuator, ;>

36:1: <punctuator, }>

13 keyword

13 identifier

29 punctuator

4 integer constant

1 floating constant

1 char constant

2 string literal

5 error

total: 68 tokens, 958 characters, 37 lines

从输出结果中我们可以发现，test3.c文件含有68个词，958个字符，37行。其中，含有13个关键词，13个标识符，29个标点，4个整数常量，1个浮点数常量，1个字符常量，2个字符串，5个错误。

5个错误分别为：

注释缺少结束标志：

15:5: <error, Unterminated block comment>

非法的八进制数字：

24:26: <error, invalid octal number>

非法的char字符常量：

27:25: <error, Invalid character constant>

不完整的浮点数：

30:27: <error, Invalid exponent in floating point number >

非法标识符：

33:33: <error, @>

经过比对，该测试完全通过。

# 4 总结

本次实验加深了我对词法分析过程的理解，也让我对C++的相关stl库更熟悉。

我构建了一个功能较全面的C语言词法分析器，该分析器可以识别C语言的基本词法元素，包括关键字、标识符、常量、字符和字符串字面量等。此外，它还具备一定的错误处理能力，能够检测诸如不完整的注释、不完整的字符串、不合法的八进制数等错误，并且在识别到错误时可以恢复继续处理。

除此之外，在调试C++版本，即手工版本的词法分析器的时候，我也发现了LEX版本的不足，二者相互参考优化，使我的代码逻辑、能力更清晰了。

在进行各种token的识别过程的代码编写的过程中，我也对C语言的各种特性、语法格式有了更深的印象，让我知道以前的学习还不够深入，之后仍要多加注意细节。

总之，这次实验加深了我对C语言的认识和理解，也让我对编译原理中的词法分析过程有了新的理解和认识。