



**实验报告： 词法分析程序的设计与实现**

**——工具生成**

**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8号**

**目录**

[1 实验概述 1](#_Toc179484380)

[1.1 实验内容及要求 1](#_Toc179484381)

[1.2 实验方法要求 1](#_Toc179484382)

[1.3 实验环境说明 1](#_Toc179484383)

[2 程序设计说明 2](#_Toc179484384)

[2.1 程序语法结构说明 2](#_Toc179484385)

[2.1.1 第一部分：声明 2](#_Toc179484386)

[2.1.2 第二部分：翻译规则 2](#_Toc179484387)

[2.1.3 第三部分：辅助过程 3](#_Toc179484388)

[2.2 程序具体实现 4](#_Toc179484389)

[2.2.1 C语言声明&辅助函数 4](#_Toc179484390)

[2.2.2 正则表达式 5](#_Toc179484391)

[2.2.3 翻译规则 5](#_Toc179484392)

[2.2.4 main函数 6](#_Toc179484393)

[2.3 两种版本程序对比 7](#_Toc179484394)

[3 测试设计与分析 8](#_Toc179484395)

[3.1 测试方法 8](#_Toc179484396)

[3.2 test1.c 9](#_Toc179484397)

[3.3 test2.c 14](#_Toc179484398)

[3.4 test3.c 19](#_Toc179484399)

[4 总结 23](#_Toc179484400)

# 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

编写 LEX 源程序，利用 LEX 编译程序自动生成词法分析程序。

## 实验环境说明

* Windows 11
* Visual Studio Code
* WinFlexBison

# 2 程序设计说明

## 2.1 程序语法结构说明

根据相关博客知识的学习，本实验的lex源文件，即lex.l文件语法组成结构如下：

### 2.1.1 第一部分：声明

该部分的代码由两部分组成：直接写入在编译生成的lex.yy.c文件里的C语言声明语句和需要在翻译规则部分出现的相关记号的正规表达式。

其中声明部分的语法与C语言相同，主要包含必要的头文件、声明全局变量、声明函数等。这部分必须用 %{ 和 %} 包起来。

正规表达式部分前面是类字符串名称，接一个空格，之后是匹配这种字符串的正则表达式，里面的元字符要匹配的话要用 \ 进行转义。对连续多个元字符进行转义可以使用一对双引号，元字符有大中小括号，+，-，\* 等等。

示例如下：

1. %{

2. #include <stdio.h>

3. #include <string.h>

4. #include <stdlib.h>

5. %}

6.

7. ws [ \f\n\r\t\v]

8. prep "#"[^\n]\*

9. line\_comment \/\/[^\n]\*

10. block\_comment \/\\*([^\*]|\\*+[^\*/])\*\\*+\/

### 2.1.2 第二部分：翻译规则

该部分的主要功能是实现匹配到字符串后程序的操作。语法规则为：{}中为匹配到的正规表达式，第二个{}中的内容为匹配后的具体操作，此处语法与C语言相同。yytext在此处表示匹配到的文本。

值得注意的是，lex有两个匹配规则：

1. 按最长匹配原则确定被选中的单词；

2. 如果一个字符串能被若干正规式匹配，则先匹配排在前面的正规式。

示例如下：

1. {ws}|{prep}|{line\_comment}|{block\_comment} {} // 对于合法的注释，跳过

2.

3. {incomplete\_comment} {

4. record\_token("error", "Unterminated block comment");

5. while (input() != '\n') ; // 跳过第二行，从第三行开始词法分析

6. }

### 2.1.3 第三部分：辅助过程

辅助过程用来定义一些翻译规则所需要的函数。通常包含main函数，该部分的代码会直接写入到lex.yy.c的末尾。语法规则与C语言相同。示例如下：

1. int main(int argc, char \*\*argv) {

2. FILE \*fin;

3. prev\_yylineno = 0;

4. if (argc > 1) {

5. fin = fopen(argv[1], "r");

6. if (fin == NULL) {

7. perror("cannot open file");

8. return 1;

9. }

10. yyset\_in(fin);

11. }

12.

13. int token;

14. while ((token = yylex()) != 0) {}

15.

16. print\_statistics();

17.

18. if (fin) {

19. fclose(fin);

20. }

21. return 0;

22. }

## 2.2 程序具体实现

该部分用于说明lex.l文件的具体实现内容、方法、功能。具体代码实现可见lex.l源文件。

### 2.2.1 C语言声明&辅助函数

在词法分析器的实现中，我使用了一些C语言的声明和辅助函数，以便处理输入文本并统计词法分析的结果。

**C语言声明：**

引入了头文件 stdio.h，string.h，和 stdlib.h，用于处理文件输入、字符串操作和动态内存分配。

声明了一些全局变量用于记录当前的分析状态，例如 yycolumn（当前列），char\_cnt（字符计数），start\_line 和 start\_column（用于记录当前token的起始位置），以及 prev\_yylineno（上一次匹配的行数）。

**辅助宏 YY\_USER\_ACTION：**

YY\_USER\_ACTION用于在每次成功匹配到一个词法单元时更新行、列等信息。它记录当前词法单元的起始行和列，并更新 yycolumn 以便正确显示词法单元的位置。

**辅助函数 record\_token：**

函数 record\_token 用于将识别到的token类型和内容进行记录。

它会判断当前token类型是否已存在，如果存在则增加相应的计数，否则会将该类型加入到 token\_types 数组中，同时计数器加一。

函数中还使用 printf 输出识别到的token的信息，包括行、列、类型和内容。

**辅助函数 print\_statistics：**

该函数用于在词法分析结束后，打印整个分析过程的统计信息。包括每种token类型的数量、字符总数和总行数。

### 2.2.2 正则表达式

正则表达式部分用于定义C语言中的各种词法元素。它们可以匹配关键字、标识符、数值、字符常量、字符串字面量等。

**关键词 (keyword)：**

匹配C语言中的保留字，例如 int, char, float 等。

**标识符 (identifier)：**

匹配以字母或下划线开头，后接字母、数字或下划线的合法标识符。

**整型常量 (decimal\_int, octal\_int, hex\_int)：**

匹配十进制、八进制和十六进制的整型常量。

**浮点数常量 (decimal\_float, hex\_float)：**

匹配符合C语言语法的浮点数，包括可能的指数部分。

**字符常量 (valid\_char\_constant) 和字符串字面量 (string\_literal)**：

匹配合法的字符常量和字符串字面量。包括转义字符的处理。

**注释 (line\_comment, block\_comment)：**

匹配单行注释 (//...) 和多行注释 (/\*...\*/)。

**其他特殊处理：**

针对不完整的注释、不完整的指数、不合法的字符常量等情况，定义了相应的正则表达式来检测并处理这些错误。

### 2.2.3 翻译规则

翻译规则部分定义了正则表达式与相应的操作之间的关系。当匹配到不同的C语言元素时，使用相应的函数来处理和记录这些元素。

**空白符和注释：**

匹配到空白符（ws）、预处理指令（prep）、单行注释（line\_comment）和多行注释（block\_comment）时，不执行任何操作，直接跳过。

**错误处理：**

匹配到不完整的注释（incomplete\_comment）时，记录为错误，并跳到下一行。

匹配到无效八进制数（invalid\_octal）、无效字符常量（invalid\_char\_constant）和不完整的指数（incomplete\_exp）时，均记录为相应的错误。

**关键字、标识符和标点符号：**

匹配到关键字（keyword）时，调用 record\_token 记录类型为 keyword 的token；匹配到标识符（identifier）和标点符号（punctuator）时，类似地调用 record\_token 记录相应的token。

**数值常量和字符串字面量：**

匹配到整型常量（decimal\_int, octal\_int, hex\_int）、浮点数常量（decimal\_float, hex\_float）、字符常量（valid\_char\_constant）和字符串字面量（string\_literal）时，分别记录为相应的token类型。

**其他：**

所有没有匹配到的字符，统一处理为错误，并记录为 error 类型。

### 2.2.4 main函数

main 函数用于初始化词法分析器并启动词法分析的过程，最后打印分析的统计结果。

**文件输入的处理：**

main 函数首先检查是否提供了输入文件。如果提供了文件，则通过 fopen 打开文件，并设置输入流为该文件。

**启动词法分析：**

使用 yylex() 函数启动词法分析器，逐一读取输入文件的每个token，直到 yylex() 返回0（表示输入结束）。

**统计信息的打印：**

调用 print\_statistics() 打印所有token的统计信息，包括每种类型的数量、字符总数和行数。

## 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lex.l文件**

在lex.l所在文件夹进入终端。输入命令：

> .\win\_flex lex.l

即将lex.l文件编译成为lex.yy.c文件。

**2. 编译lex.yy.c文件**

在相同的路径下，输入命令：

> gcc -o test.exe lex.yy.c

其中 -o后为所需编译完成的可执行文件的名称。

**3. 运行测试用例**

在相同的路径下，输入命令：

> .\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

total: 130 tokens, 853 characters, 46 lines

从输出结果中我们可以发现，test1.c文件含有130个词，853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：

1. 3:1: <keyword, int>

2. 3:5: <identifier, main>

3. 3:9: <punctuator, (>

4. 3:10: <punctuator, )>

5. 3:12: <punctuator, {>

6. 5:5: <keyword, int>

7. 5:9: <identifier, decimal>

8. 5:17: <punctuator, =>

9. 5:19: <integer constant, 100>

10. 5:22: <punctuator, ;>

11. 8:5: <keyword, int>

12. 8:9: <identifier, octal>

13. 8:15: <punctuator, =>

14. 8:17: <integer constant, 0144>

15. 8:21: <punctuator, ;>

16. 11:5: <keyword, int>

17. 11:9: <identifier, hex>

18. 11:13: <punctuator, =>

19. 11:15: <integer constant, 0x64>

20. 11:19: <punctuator, ;>

21. 14:5: <keyword, float>

22. 14:11: <identifier, pi>

23. 14:14: <punctuator, =>

24. 14:16: <floating constant, 3.14159>

25. 14:23: <punctuator, ;>

26. 15:5: <keyword, float>

27. 15:11: <identifier, e>

28. 15:13: <punctuator, =>

29. 15:15: <floating constant, 2.71828>

30. 15:22: <punctuator, ;>

31. 16:5: <keyword, float>

32. 16:11: <identifier, hexFloat>

33. 16:20: <punctuator, =>

34. 16:22: <floating constant, 0x1.2p10>

35. 16:30: <punctuator, ;>

36. 19:5: <keyword, double>

37. 19:12: <identifier, exp>

38. 19:16: <punctuator, =>

39. 19:18: <integer constant, 1>

40. 19:19: <identifier, e10>

41. 19:22: <punctuator, ;>

42. 21:5: <identifier, printf>

43. 21:11: <punctuator, (>

44. 21:12: <string literal, "Decimal: %d\n">

45. 21:27: <punctuator, ,>

46. 21:29: <identifier, decimal>

47. 21:36: <punctuator, )>

48. 21:37: <punctuator, ;>

49. 22:5: <identifier, printf>

50. 22:11: <punctuator, (>

51. 22:12: <string literal, "Octal: %o\n">

52. 22:25: <punctuator, ,>

53. 22:27: <identifier, octal>

54. 22:32: <punctuator, )>

55. 22:33: <punctuator, ;>

56. 23:5: <identifier, printf>

57. 23:11: <punctuator, (>

58. 23:12: <string literal, "Hex: %x\n">

59. 23:23: <punctuator, ,>

60. 23:25: <identifier, hex>

61. 23:28: <punctuator, )>

62. 23:29: <punctuator, ;>

63. 24:5: <identifier, printf>

64. 24:11: <punctuator, (>

65. 24:12: <string literal, "Float in hex: %a\n">

66. 24:32: <punctuator, ,>

67. 24:34: <identifier, hexFloat>

68. 24:42: <punctuator, )>

69. 24:43: <punctuator, ;>

70. 25:5: <identifier, printf>

71. 25:11: <punctuator, (>

72. 25:12: <string literal, "Exponential: %e\n">

73. 25:31: <punctuator, ,>

74. 25:33: <identifier, exp>

75. 25:36: <punctuator, )>

76. 25:37: <punctuator, ;>

77. 28:5: <keyword, int>

78. 28:9: <identifier, bases>

79. 28:14: <punctuator, [>

80. 28:15: <punctuator, ]>

81. 28:17: <punctuator, =>

82. 28:19: <punctuator, {>

83. 28:20: <integer constant, 10>

84. 28:22: <punctuator, ,>

85. 28:24: <integer constant, 07>

86. 28:26: <punctuator, ,>

87. 28:28: <integer constant, 0x1A>

88. 28:32: <punctuator, }>

89. 28:33: <punctuator, ;>

90. 31:5: <keyword, for>

91. 31:9: <punctuator, (>

92. 31:10: <keyword, int>

93. 31:14: <identifier, i>

94. 31:16: <punctuator, =>

95. 31:18: <integer constant, 0>

96. 31:19: <punctuator, ;>

97. 31:21: <identifier, i>

98. 31:23: <punctuator, <>

99. 31:25: <keyword, sizeof>

100. 31:31: <punctuator, (>

101. 31:32: <identifier, bases>

102. 31:37: <punctuator, )>

103. 31:39: <punctuator, />

104. 31:41: <keyword, sizeof>

105. 31:47: <punctuator, (>

106. 31:48: <identifier, bases>

107. 31:53: <punctuator, [>

108. 31:54: <integer constant, 0>

109. 31:55: <punctuator, ]>

110. 31:56: <punctuator, )>

111. 31:57: <punctuator, ;>

112. 31:59: <identifier, i>

113. 31:60: <punctuator, ++>

114. 31:62: <punctuator, )>

115. 31:64: <punctuator, {>

116. 32:9: <identifier, printf>

117. 32:15: <punctuator, (>

118. 32:16: <string literal, "Array element in base 10: %d\n">

119. 32:48: <punctuator, ,>

120. 32:50: <identifier, bases>

121. 32:55: <punctuator, [>

122. 32:56: <identifier, i>

123. 32:57: <punctuator, ]>

124. 32:58: <punctuator, )>

125. 32:59: <punctuator, ;>

126. 33:5: <punctuator, }>

127. 35:5: <keyword, return>

128. 35:12: <integer constant, 0>

129. 35:13: <punctuator, ;>

130. 36:1: <punctuator, }>

131.

132. 14 keyword

133. 28 identifier

134. 69 punctuator

135. 10 integer constant

136. 3 floating constant

137. 6 string literal

138. total: 130 tokens, 939 characters, 36 lines

从输出结果中我们可以发现，test2.c文件含有130个词，939个字符，36行。其中，含有14个关键词，28个标识符，69个标点，10个整数常量，3个浮点数常量，6个字符串。

然而，经过与手工实现的词法分析器对比，我发现lex实现的分析器无法正确识别1e10这种格式的浮点数，尽管它可以识别出错误的浮点数（见test3）。会识别出：1和e10.

后续考虑完善浮点数的正则表达式以优化该功能。

## 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>

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, Incomplete exponent part>

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

28 punctuator

4 integer constant

1 floating constant

1 char constant

2 string literal

5 error

total: 67 tokens, 959 characters, 36 lines

从输出结果中我们可以发现，test3.c文件含有67个词，959个字符，36行。其中，含有13个关键词，13个标识符，28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>

非法的char字符常量：

27:25: <error, Invalid character constant>

不完整的浮点数：

30:27: <error, Incomplete exponent part>

非法标识符：

33:33: <error, @>

经过比对发现，在识别不完整的浮点数的时候，lex版本会错误的将浮点数后面的 ; 计入在内，导致token、字符数量有误。

# 4 总结

本次实验让我学会了lex的基本使用方法和语法，也加深了我对词法分析过程的理解。

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。

特别是对于正则表达式的学习和使用，让我对正则表达式掌握的更加熟练。在编写lex代码的过程中，我时常遇到lex的报错和warning，之后根据各种方式的查询和学习，也认识到lex的最长匹配规则和优先匹配规则，让我学会了在编写代码时的细节处理。

此外，在尝试编写出可以识别代码bug的正则式的时候，也让我发现了自己最初写的正则式的不足之处。两者的相互补充学习，让我的正则式更加的严谨。在调试的过程中也加深了我对C语言的词法规则的认识。

对于后续代码的提升优化，我考虑加入更多的词法错误的正则表达式，以识别更多种类的错误并纠正。

总之，这次实验加深了我对C语言的认识和理解，让我学会了lex的使用和语法，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。