

编译原理与技术实验一：词法分析程序的设计与实现

实验报告

毛子恒

2019211397

北京邮电大学 计算机学院

日期：2021 年 9 月 28 日

1 概览

1.1 任务描述

设计并实现 C 语言的词法分析程序，要求如下。

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. 可以统计源程序汇总的语句行数、单词个数和字符个数，并输出统计结果。
4. 检查源程序中存在的错误，并可以报告错误所在的位置。
5. 发现源程序中存在的错误后，进行适当的恢复，使词法分析可以继续，对源程序进行一次扫描，即可检查并报告出源程序中存在的词法错误。

采用 C++ 作为实现语言，手工编写词法分析程序。

1.2 开发环境

- macOS Big Sur 11.6
- Apple clang version 12.0.5
- cmake version 3.19.6
- Clion 2021.2.1
- Visual Studio Code 1.60.2

2 模块介绍

2.1 模块划分

各模块及其关系如图 1。

其中，`token` 模块定义了记号的数据结构；`dict` 模块定义了用于快速检索关键字的字典树；`scanner` 模块对文件输入流进行简单封装，实现了一个简单的缓冲区，实现向前预先查看数个

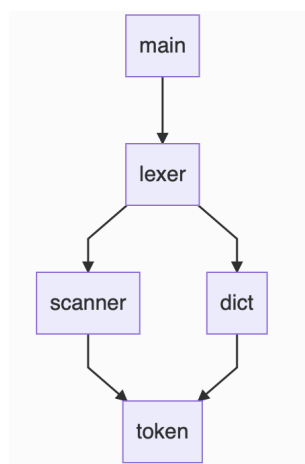


图 1: 模块关系图

字符的功能；`lexer` 模块是词法分析器，将文件流中的字符分成不同的记号，并用数据结构表示。

2.2 记号数据结构

在编写词法分析程序时，我参考了 ISO/IEC 9899:1999 标准，在标准的 6.4 节中有对词法单元的详细规定。我在标准的基础上做了一些改动：

1. 将满足以下条件的部分标记为预处理器，预处理器部分不作处理。
 - 以 `#` 开头，且此符号在行首或者此符号前都是空白字符。
 - 以换行结束，除非忽略本行末尾的空白字符后本行末尾的最后一个字符为 `\`。
2. 将在程序中其他地方出现的 `#` 和 `##` 符号视为非法符号。
3. 不处理 `Universal character`。

我将记号分为以下十类：

1. `Unknown`，未识别和出现错误的记号，本词法分析器能够识别出如下几种错误：
 - 非法的 `#` 和 `##` 符号。
 - 未知的符号。
 - 未闭合的块注释。
 - 不合法的数字。
 - 未闭合的字符常量。
 - 未闭合的字符串面值。
 - `Slash Newline at EOF`，在字符常量、字符串面值或者预处理器中，出现 `\` 符号后，直到 EOF 也没有新的行。
2. `Keyword`，关键字，在 C99 标准中共 37 个。
3. `Identifier`，标识符。
4. `Integer Constant`，整数常量。
5. `Float Constant`，浮点数常量。
6. `Char Constant`，字符常量。

7. String Literal, 字符串面值。
8. Punctuator, 标点符号, 在 C99 标准中共 49 个。
9. Preprocessing, 预处理器。
10. EOF, 文件末尾。

2.2.1 TokenLocation

此类用于存储一个记号的位置。

```
1 class TokenLocation
2 {
3 public:
4     TokenLocation();
5     TokenLocation(std::string file_name_, unsigned int row_, unsigned int
        ↪ column_);
6     friend std::ostream &operator<<(std::ostream &os, const TokenLocation
        ↪ &location);
7 private:
8     std::string file_name;
9     unsigned int row;
10    unsigned int column;
11 };
```

2.2.2 Token

此类用于表示一个记号。

```
1 class Token
2 {
3 public:
4     Token();
5     Token(TokenType type_, TokenLocation location_, Keyword keyword_,
        ↪ Punctuator punctuator_, Unknown unknown_, std::string literal_);
6     Token(TokenType type_, TokenLocation location_, Keyword keyword_);
7     Token(TokenType type_, TokenLocation location_, Punctuator punctuator_);
8     Token(TokenType type_, TokenLocation location_, std::string literal_);
9     Token(TokenType type_, TokenLocation location_, Unknown unknown_);
10    Token(TokenType type_, TokenLocation location_, Unknown unknown_,
        ↪ std::string literal_);
11    TokenType GetType() const;
12    friend std::ostream &operator<<(std::ostream &os, const Token &token);
13 private:
14     TokenType type;
15     TokenLocation location;
16     Keyword keyword;
17     Punctuator punctuator;
18     Unknown unknown;
```

```
19     std::string literal;
20 };
```

2.3 字典

`dict` 模块实现了一个简单的字典树，用于快速查找关键字。字典树提供了插入和查询接口。假设 n 为关键字的数量， l 为关键字的长度，相比于依次比较每个关键字的 $O(nl)$ 的复杂度，字典树的复杂度为 $O(l)$ 。

此外，无需过多修改，字典树还可以扩展为识别所有的标识符。

```
1 struct TrieNode
2 {
3     Keyword keyword;
4     TrieNode * child[63]{};
5 };
6 class Dict
7 {
8 public:
9     Dict() = default;
10    Keyword find(const std::string &str) const;
11    void insert(Keyword keyword, const std::string &str);
12 private:
13    TrieNode root;
14    static size_t char2Index(char c);
15 };
```

2.4 扫描器

由于在词法分析时，有在文件流中超前查看几个字符的需求，因此需要对文件输入流建立一个缓冲区，并且对缓冲区可以进行顺序查找。

不同于教材上的实现方式，我简单地采用了一个 `deque` 实现缓冲区，提供了判断是否读到文件末尾、读取一个字符、向后查看几个字符、跳过几个字符的接口。

此外，`Scanner` 类中还实现了获取当前字符的位置、以及行数和字符数的计数功能。

```
1 class Scanner
2 {
3 public:
4     explicit Scanner(std::string file_name_);
5     bool Eof();
6     char GetChar();
7     char PeekChar(size_t offset);
8     void SkipChar(int num);
9     const std::string &GetFileName() const;
10    unsigned int GetRow() const;
11    unsigned int GetCountChar() const;
```

```

12     bool GetStartOfRow() const;
13     TokenLocation GetLocation() const;
14 private:
15     std::string file_name;
16     std::ifstream source_file;
17     unsigned int row;
18     unsigned int column;
19     unsigned int count_char;
20     std::deque<char> buffer;
21     bool start_of_row;
22     void Read();
23 };

```

2.5 词法分析器

```

1  class Lexer
2  {
3  public:
4      Lexer(std::string file_name_, Dict dict_);
5      Token GetNextToken();
6      friend std::ostream &operator<<(std::ostream &os, const Lexer &lexer);
7  private:
8      Scanner scanner;
9      Dict dict;
10     std::string buffer;
11     TokenLocation token_location;
12     unsigned int lexeme_count[10];
13     Token GetTokenHandler();
14     void LineCommentHandler();
15     bool BlockCommentHandler();
16     Token PreprocessingHandler();
17     Token NumberTokenHandler();
18     Token CharTokenHandler();
19     Token StringTokenHandler();
20     Token IdentifierTokenHandler();
21     static bool isNumberCharacter(char c);
22     static bool isOctCharacter(char c);
23     static bool isIntegerSuffix(const std::string &str);
24     static bool isFloatSuffix(const std::string &str);
25     Token DecimalIntegerHandler(std::string::iterator iter);
26     Token OctIntegerHandler(std::string::iterator iter);
27     Token HexIntegerHandler(std::string::iterator iter);
28     Token DecimalFloatHandler(std::string::iterator iter);
29     Token HexFloatHandler(std::string::iterator iter);
30     static bool isIdentifierCharacter(char c);
31 };

```

词法分析器提供一个简单的接口 `GetNextToken` 来获取源文件中的下一个符号，并且可以通过重载的 `<<` 运算符输出分析结果。

Lexer 中的函数调用关系图如图 2。

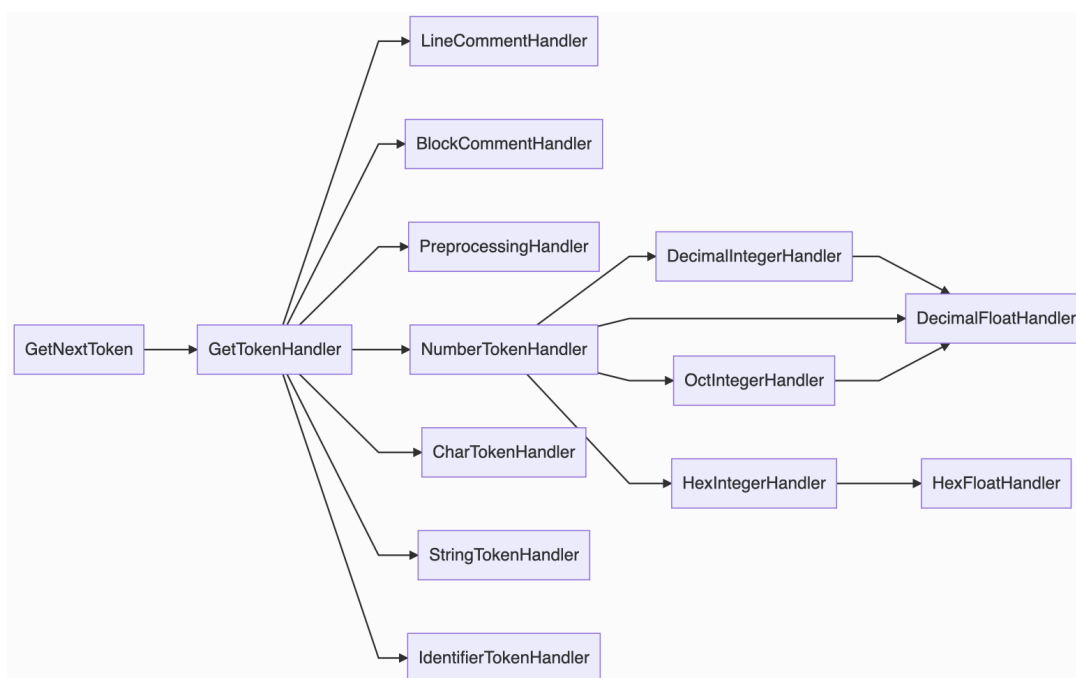


图 2: 词法分析器的函数调用关系图

`GetNextToken` 调用 `GetTokenHandler` 获取下一个记号，并且计数。

`GetTokenHandler` 中首先跳过所有的空白字符，之后依据符号的类型分别进行处理。

2.5.1 标点符号处理

在遇到标点符号时，词法分析器通过向前查看文件流中的字符判断标点符号的类型，此时注意符号的匹配过程是贪心的。例如表达式 `x+++++y` 会被解析为五个记号，依次为 `x`、`++`、`++`、`+`和 `y`。

2.5.2 单行注释处理

当匹配到符号 `//` 后进入单行注释处理函数 `LineCommentHandler`，此时词法分析器不断读取字符直到行末或者文件末尾。

2.5.3 块注释处理

当匹配到符号 `/*` 后进入块注释处理函数 `BlockCommentHandler`，此时词法分析器不断读取字符直到遇到符号 `*/`，如果读到文件末尾，则返回块注释未闭合错误。

2.5.4 预处理器处理

当匹配到符号 `#` 时，若该行中此符号前没有非空白字符，则进入预处理器处理函数 `PreprocessingHandler`。此后，词法分析器不断读取字符，直到：

- 如果遇到换行符或者文件结尾则结束。

- 如果遇到\符号，则跳过该反斜杠后的所有空白字符，如果遇到了换行，则跳过这个换行符号，之后继续预处理器的处理过程。

2.5.5 数字常量处理

当遇到数字或者符号，之后的字符是数字时，进入数字常量处理函数 `NumberTokenHandler`。

数字的处理过程分为两个步骤：第一步，从遇到的第一个符号开始，将所有的数字、大小写字母、紧跟在大小写 `e` 和 `p` 之后的 `+` 和 `-` 符号加入缓冲区中；第二步，判断缓冲区中的数字是否是一个合法数字。

其中，第二步可以调用系统函数 `stof` 完成，但是出于练习目的，我手动实现了判断数字合法性的过程。

通过检查缓冲区的前缀，调用不同的处理函数：

- 如果以 `.` 开头，作为十进制浮点数处理。
- 如果以 `0x` 或者 `0X` 开头，作为十六进制整数处理。
- 如果以 `0` 开头，作为八进制整数处理。
- 否则，作为十进制整数处理。

十进制整数处理 `DecimalIntegerHandler` 处理十进制整数。

函数不断向前读取数字，直到：

- 读取到缓冲区结束，则返回十进制整数记号。
- 读取到 `.`、`e` 或者 `E`，则转到十进制浮点数处理。
- 读取到其他字符，则判断从当前字符到缓冲区结束的部分是否是合法的整数后缀，如果是，则返回十进制整数记号，否则返回错误。

八进制整数处理 `OctIntegerHandler` 处理八进制整数。

函数不断向前读取八进制数字，直到：

- 读取到缓冲区结束，则返回八进制整数记号。
- 读取到 `.`、`e` 或者 `E`，则转到十进制浮点数处理。
- 读取到数字 `8` 或者 `9`，则继续向前读取数字，直到遇到 `.`、`e` 或者 `E`，则转到十进制浮点数处理，如果读取不到上述三种字符，则返回错误。
- 读取到其他字符，则判断从当前字符到缓冲区结束的部分是否是合法的整数后缀，如果是，则返回八进制整数处理，否则返回错误。

十六进制整数处理 `HexIntegerHandler` 处理十六进制整数。

在之前处理十进制数的过程中，我们保证了不存在浮点数小数点前后都没有数字的情况，但是对于十六进制数我们只判断了前缀 `0x`，因此如果十六进制前缀后紧跟一个小数点，需要确保小数点之后至少有一个十六进制字符。如果小数点之后不是十六进制字符，则返回错误。

此外，十六进制数除了前缀和后缀之外至少需要有一位十六进制数字，如果直接读取到非十六进制数字，则返回错误。

函数不断向前读取十六进制字符，直到：

- 读取到缓冲区结束，则返回十六进制整数记号。
- 读取到 `.`、`p` 或者 `P`，则转到处理十六进制浮点数。
- 读取到其他字符，则判断从当前字符到缓冲区结束的部分是否是合法的整数后缀，如果是，则返回十六进制整数记号，否则返回错误。

十进制浮点数处理 `DecimalFloatHandler` 处理十进制浮点数。

此函数默认浮点数的整数部分已经处理完成，从小数点或者 `e` 开始处理。

如果有小数点，小数点后可以跟随若干个十进制数字。

如果有 `e` 或者 `E`，其后跟随一个可选的+或-，之后跟随至少一个十进制数字。

如果此时读取到缓冲区结束，则返回十进制浮点数记号。

否则判断剩余的部分是否是合法的浮点数后缀，如果是，则返回十进制浮点数记号，否则返回错误。

十六进制浮点数处理 `HexFloatHandler` 处理十六进制浮点数。

此函数默认浮点数的整数部分已经处理完成，从小数点或者 `p` 开始处理。

如果有小数点，小数点后可以跟随若干个十进制数字。

如果有 `p` 或者 `P`，其后跟随一个可选的+或-，之后跟随至少一个十进制数字。

十六进制浮点数的指数部分是必须的，如果没有指数部分，则返回错误。

如果此时读取到缓冲区结束，则返回十六进制浮点数记号。

否则判断剩余的部分是否是合法的浮点数后缀，如果是，则返回十六进制浮点数记号，否则返回错误。

2.5.6 字符常量处理

当读取到符号 `'` 或者字母 `L` 紧跟一个 `'` 时，进入字符常量处理函数 `CharTokenHandler`。

此后，词法分析器不断读取字符：

- 如果遇到 `'` 则结束。
- 如果遇到换行符或者文件结尾，则返回未闭合字符常量错误。
- 如果遇到 `\` 符号，则读取该反斜杠后的所有空白字符，直到：
 - 如果遇到了换行，则跳过这个换行符号，并且丢弃刚才记录到的空白字符，继续字符常量的处理过程。
 - 否则，将刚才记录到的反斜杠符号和空白字符加入缓冲区，继续字符常量的处理过程。
- 否则，将该符号加入缓冲区。

2.5.7 字符串字面量处理

当读取到符号 `"` 或者字母 `L` 紧跟一个 `"`，进入字符串字面量处理函数 `StringTokenHandler`。

除了将单引号替换为双引号之外，过程和字符常量处理基本相同。

2.5.8 标识符处理

当读取到符号 `_` 或者大小写字母时，进入标识符处理函数 `IdentifierTokenHandler`。

该函数首先将连续的包含下划线和大小写字母的字符串加入缓冲区中，之后在字典中查找是否存在关键字，如果存在则返回关键字记号，否则返回标识符记号。

3 用户指南

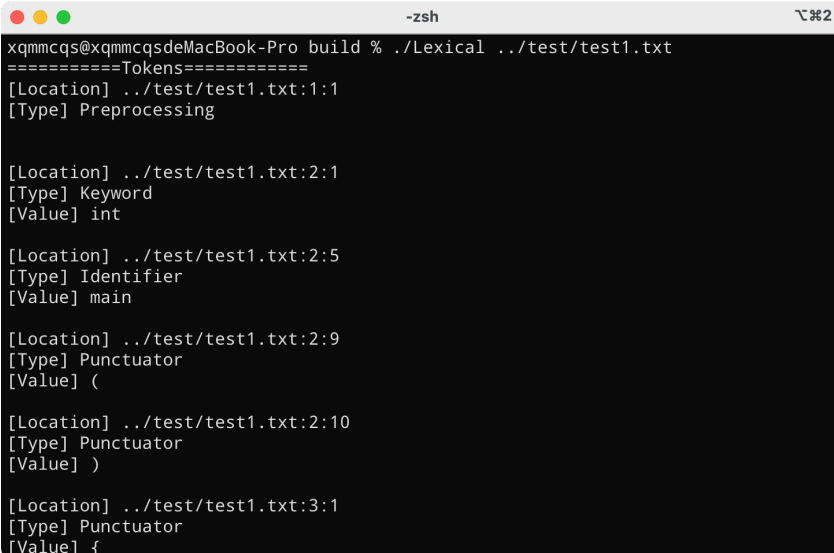
在项目目录中执行以下命令来编译：

```
mkdir build
cd build
cmake ..
make
```

编译完成后，运行：

```
./Lexical ../test/test1.txt
```

运行截图如图 3 所示。



```
xqmmcqs@xqmmcqsdeMacBook-Pro build % ./Lexical ../test/test1.txt
=====Tokens=====
[Location] ../test/test1.txt:1:1
[Type] Preprocessing

[Location] ../test/test1.txt:2:1
[Type] Keyword
[Value] int

[Location] ../test/test1.txt:2:5
[Type] Identifier
[Value] main

[Location] ../test/test1.txt:2:9
[Type] Punctuator
[Value] (

[Location] ../test/test1.txt:2:10
[Type] Punctuator
[Value] )

[Location] ../test/test1.txt:3:1
[Type] Punctuator
[Value] {
```

图 3: 运行截图

4 测试结果

4.1 测试集 1

此测试集用于简单地测试程序是否正常运行。

4.1.1 输入

```
1  #include <stdio.h>
2  int main()
3  {
4      int a, b;
5      scanf("%d%d", &a, &b);
6      printf("%d\n", a + b);
7      return 0;
8  }
```

4.1.2 输出

```
=====Tokens=====
[Location] test/test1.txt:1:1
[Type] Preprocessing

[Location] test/test1.txt:2:1
[Type] Keyword
[Value] int

[Location] test/test1.txt:2:5
[Type] Identifier
[Value] main

[Location] test/test1.txt:2:9
[Type] Punctuator
[Value] (

[Location] test/test1.txt:2:10
[Type] Punctuator
[Value] )

[Location] test/test1.txt:3:1
[Type] Punctuator
[Value] {

[Location] test/test1.txt:4:5
[Type] Keyword
[Value] int

[Location] test/test1.txt:4:9
[Type] Identifier
[Value] a

[Location] test/test1.txt:4:10
[Type] Punctuator
[Value] ,
```

[Location] test/test1.txt:4:12
[Type] Identifier
[Value] b

[Location] test/test1.txt:4:13
[Type] Punctuator
[Value] ;

[Location] test/test1.txt:5:5
[Type] Identifier
[Value] scanf

[Location] test/test1.txt:5:10
[Type] Punctuator
[Value] (

[Location] test/test1.txt:5:11
[Type] String Literal
[Value] %d%d

[Location] test/test1.txt:5:17
[Type] Punctuator
[Value] ,

[Location] test/test1.txt:5:19
[Type] Punctuator
[Value] &

[Location] test/test1.txt:5:20
[Type] Identifier
[Value] a

[Location] test/test1.txt:5:21
[Type] Punctuator
[Value] ,

[Location] test/test1.txt:5:23
[Type] Punctuator
[Value] &

[Location] test/test1.txt:5:24
[Type] Identifier
[Value] b

[Location] test/test1.txt:5:25
[Type] Punctuator
[Value])

[Location] test/test1.txt:5:26
[Type] Punctuator

[Value] ;

[Location] test/test1.txt:6:5
[Type] Identifier
[Value] printf

[Location] test/test1.txt:6:11
[Type] Punctuator
[Value] (

[Location] test/test1.txt:6:12
[Type] String Literal
[Value] %d\n

[Location] test/test1.txt:6:18
[Type] Punctuator
[Value] ,

[Location] test/test1.txt:6:20
[Type] Identifier
[Value] a

[Location] test/test1.txt:6:22
[Type] Punctuator
[Value] +

[Location] test/test1.txt:6:24
[Type] Identifier
[Value] b

[Location] test/test1.txt:6:25
[Type] Punctuator
[Value])

[Location] test/test1.txt:6:26
[Type] Punctuator
[Value] ;

[Location] test/test1.txt:7:5
[Type] Keyword
[Value] return

[Location] test/test1.txt:7:12
[Type] Integer Constant
[Value] 0

[Location] test/test1.txt:7:13
[Type] Punctuator
[Value] ;

[Location] test/test1.txt:8:1

```
[Type] Punctuator  
[Value] }
```

```
=====Analysis=====
```

```
[File Name] test/test1.txt  
[Rows] 8  
[Characters] 115  
[Unknown] 0  
[Keyword] 3  
[Identifier] 9  
[Integer Constant] 1  
[Float Constant] 0  
[Character Constant] 0  
[String Literal] 2  
[Punctuator] 19  
[Preprocessing] 1  
[End Of File] 1
```

4.2 测试集 2

此测试集是一个没有错误的程序，测试了所有关键字、字符、各种类型的常量是否能被正确识别。同时测试了一些略微复杂的数字常量、以及跨行的字符串字面量能否正确识别。

4.2.1 输入

```
1  #include <stdio.h>  
2  #define LL long long  
3  #define NAME(n)      \  
4      int name(n)      \  
5      {                \  
6          return name##n; \  
7      }  
8  
9  typedef struct Node  
10 {  
11     int a, b;  
12     long long c;  
13     union  
14     {  
15         float d;  
16         double e;  
17     };  
18     LL f;  
19 } Node;  
20  
21 char ch[10];  
22 enum TT  
23 {  
24     namea,
```

```

25     nameb,
26 };
27 extern eee;
28
29 static int func(unsigned long long var)
30 {
31     var += var * (var - var) + ((var / var) % var);
32     unsigned LL *restrict p = &var;
33     return var;
34 }
35
36 inline int f()
37 {
38     return sizeof(LL);
39 }
40
41 int main(int argc, char *argv[])
42 {
43     Node node;
44     Node *pnode = &node;
45     node.a = pnode->a;
46     node.b++;
47     auto unsigned int x = node.b & ((pnode->c | ch[0]) ^ node.a--);
48     volatile unsigned char m = x << 5 + 'c';
49     const float y = 3.14, z = 3e5;
50     _Bool t = x < z && y > z || x <= y && y >= z;
51     node.a = m++ + x;
52     _Complex n;
53     goto label;
54 #ifndef what
55 #define what
56 #endif
57     char *str = "str/*comment*/";
58 label:
59     if (y != z)
60         x <<= 2 ^ 3;
61     else if (!(x >> 2))
62         x >>= 2;
63     else
64         x &= 5;
65     m ^= (x |= ~node.b);
66     x -= (node.a *= (m /= (node.b %= 2u)));
67     while (1)
68     {
69         /* This is a loop */
70         if (5u)
71             break;
72         if (y <= z)
73             continue;
74     }
75     do // a comment

```

```

76     {
77         signed _temp = 0x0123456789abcdefUL, temp_temp = 00123251;
78         float __ = 0x12f.2p+4;
79         switch (_temp)
80         {
81             case 'd':
82                 /* Comment
83                 ???
84                 @ # `
85
86                 break;
87                 */
88                 break;
89
90             default:
91                 break;
92         }
93     } while (0);
94     // a comment "string" 'c' 123321
95     for(register int i = 1; i <= -1; ++i)
96         ;
97
98     printf("%llu\n", 3ul ? 21lu : 45ull);
99     printf("floats \
100 %f %lf\n",
101         1.e-2f, (double).356e+31);
102     return x, 0;
103 }

```

4.2.2 输出（节选）

```

=====Tokens=====
[Location] test/test2.txt:1:1
[Type] Preprocessing

[Location] test/test2.txt:2:1
[Type] Preprocessing

[Location] test/test2.txt:3:1
[Type] Preprocessing

[Location] test/test2.txt:9:1
[Type] Keyword
[Value] typedef

[Location] test/test2.txt:9:9
[Type] Keyword

```

[Value] struct

[Location] test/test2.txt:9:16
[Type] Identifier
[Value] Node

[Location] test/test2.txt:10:1
[Type] Punctuator
[Value] {

[Location] test/test2.txt:11:5
[Type] Keyword
[Value] int

[Location] test/test2.txt:11:9
[Type] Identifier
[Value] a

[Location] test/test2.txt:11:10
[Type] Punctuator
[Value] ,

[Location] test/test2.txt:11:12
[Type] Identifier
[Value] b

[Location] test/test2.txt:11:13
[Type] Punctuator
[Value] ;

[Location] test/test2.txt:12:5
[Type] Keyword
[Value] long

[Location] test/test2.txt:12:10
[Type] Keyword
[Value] long

[Location] test/test2.txt:12:15
[Type] Identifier
[Value] c

[Location] test/test2.txt:12:16
[Type] Punctuator
[Value] ;

[Location] test/test2.txt:13:5
[Type] Keyword
[Value] union

[Location] test/test2.txt:14:5


```

[Type] Punctuator
[Value] {

[Location] test/test2.txt:15:9
[Type] Keyword
[Value] float

[Location] test/test2.txt:15:15
[Type] Identifier
[Value] d

[Location] test/test2.txt:15:16
[Type] Punctuator
[Value] ;

[Location] test/test2.txt:16:9
[Type] Keyword
[Value] double

[Location] test/test2.txt:16:16
[Type] Identifier
[Value] e

[Location] test/test2.txt:16:17
[Type] Punctuator
[Value] ;

[Location] test/test2.txt:17:5
[Type] Punctuator
[Value] }

```

...

=====Analysis=====

```

[File Name] test/test2.txt
[Rows] 103
[Characters] 1847
[Unknown] 0
[Keyword] 61
[Identifier] 100
[Integer Constant] 20
[Float Constant] 5
[Character Constant] 2
[String Literal] 3
[Punctuator] 203
[Preprocessing] 6
[End Of File] 1

```

4.2.3 分析

通过将输出与输入比较，词法分析器正确地得到了每个记号。

4.3 测试集 3

此测试集主要测试了合法和非法的字面量能否正确识别，同时将识别结果和 gcc 的编译结果进行比较。

4.3.1 输入

```
1 int a[] = {0xe+2, 012823, 123F, 23LU23, 0xFFEU, 0xU, 0EU, 0x12bufb, 12_,  
  ↪ 12g+2, 12p+2};  
2 double b[] = {02582e2, 00.3e-4l, 0X5e5ep2, 0xp0, 3e+1, 0xfpl, 0x5e.2, 2.2.2,  
  ↪ 3e5.4, 0e+9U, 0x4h3e-1, .e4, 0xp+2F, 0x.p5L, 5.e+2e-4, 0x.2p+2al};  
3 char c = 'ccc  
4 const char * d = "fff  
5 char a = '\
```

4.3.2 输出 (节选)

```
=====Tokens=====  
...  
  
[Location] test/test3.txt:1:12  
[Type] Unknown  
[Value] Illegal Number 0xe+2  
  
...  
  
[Location] test/test3.txt:1:25  
[Type] Punctuator  
[Value] ,  
  
[Location] test/test3.txt:1:27  
[Type] Unknown  
[Value] Illegal Number 123F  
  
...  
  
[Location] test/test3.txt:1:33  
[Type] Unknown  
[Value] Illegal Number 23LU23  
  
...  
  
[Location] test/test3.txt:1:41  
[Type] Unknown
```

[Value] Illegal Number 0XFFEUU

...

[Location] test/test3.txt:1:50

[Type] Unknown

[Value] Illegal Number 0xU

...

[Location] test/test3.txt:1:55

[Type] Unknown

[Value] Illegal Number 0EU

...

[Location] test/test3.txt:1:60

[Type] Unknown

[Value] Illegal Number 0x12bufb

...

[Location] test/test3.txt:1:70

[Type] Integer Constant

[Value] 12

[Location] test/test3.txt:1:72

[Type] Identifier

[Value] _

...

[Location] test/test3.txt:1:75

[Type] Unknown

[Value] Illegal Number 12g

[Location] test/test3.txt:1:78

[Type] Punctuator

[Value] +

[Location] test/test3.txt:1:79

[Type] Integer Constant

[Value] 2

...

[Location] test/test3.txt:1:82

[Type] Unknown

[Value] Illegal Number 12p+2

...

[Location] test/test3.txt:2:15
[Type] Float Constant
[Value] 02582e2

...

[Location] test/test3.txt:2:24
[Type] Float Constant
[Value] 00.3e-41

...

[Location] test/test3.txt:2:34
[Type] Float Constant
[Value] 0X5e5ep2

...

[Location] test/test3.txt:2:44
[Type] Unknown
[Value] Illegal Number 0xp0

...

[Location] test/test3.txt:2:50
[Type] Unknown
[Value] Illegal Number 3e+1

...

[Location] test/test3.txt:2:56
[Type] Unknown
[Value] Illegal Number 0xfp1

...

[Location] test/test3.txt:2:63
[Type] Unknown
[Value] Illegal Number 0x5e.2

...

[Location] test/test3.txt:2:71
[Type] Unknown
[Value] Illegal Number 2.2.2

...

[Location] test/test3.txt:2:78
[Type] Unknown

[Value] Illegal Number 3e5.4

...

[Location] test/test3.txt:2:85
[Type] Unknown
[Value] Illegal Number 0e+9U

...

[Location] test/test3.txt:2:92
[Type] Unknown
[Value] Illegal Number 0x4h3e-1

...

[Location] test/test3.txt:2:102
[Type] Punctuator
[Value] .

[Location] test/test3.txt:2:103
[Type] Identifier
[Value] e4

...

[Location] test/test3.txt:2:107
[Type] Unknown
[Value] Illegal Number 0xp+2F

...

[Location] test/test3.txt:2:115
[Type] Unknown
[Value] Illegal Number 0x.p5L

...

[Location] test/test3.txt:2:123
[Type] Unknown
[Value] Illegal Number 5.e+2e-4

...

[Location] test/test3.txt:2:133
[Type] Unknown
[Value] Illegal Number 0x.2p+2a1

...

[Location] test/test3.txt:3:1

[Type] Keyword
[Value] char

[Location] test/test3.txt:3:6
[Type] Identifier
[Value] c

[Location] test/test3.txt:3:8
[Type] Punctuator
[Value] =

[Location] test/test3.txt:3:10
[Type] Unknown
[Value] Unclosed Character Constant ccc

[Location] test/test3.txt:4:1
[Type] Keyword
[Value] const

[Location] test/test3.txt:4:7
[Type] Keyword
[Value] char

[Location] test/test3.txt:4:12
[Type] Punctuator
[Value] *

[Location] test/test3.txt:4:14
[Type] Identifier
[Value] d

[Location] test/test3.txt:4:16
[Type] Punctuator
[Value] =

[Location] test/test3.txt:4:18
[Type] Unknown
[Value] Unclosed String Literal fff

[Location] test/test3.txt:5:1
[Type] Keyword
[Value] char

[Location] test/test3.txt:5:6
[Type] Identifier
[Value] a

[Location] test/test3.txt:5:8
[Type] Punctuator
[Value] =

```
[Location] test/test3.txt:5:10
[Type] Unknown
[Value] Slash Newline at EOF
```

=====Analysis=====

```
[File Name] test/test3.txt
[Rows] 5
[Characters] 280
[Unknown] 25
[Keyword] 6
[Identifier] 7
[Integer Constant] 2
[Float Constant] 3
[Character Constant] 0
[String Literal] 0
[Punctuator] 43
[Preprocessing] 0
[End Of File] 1
```

4.3.3 分析

第一行中的整数常量全部是非法的：

- `0xe+2`：根据2.5.5节说明的两个步骤，`0xe+2`会被认为是一个数字常量，而不是分开成三个记号`0xe`、`+`和`2`，而根据前缀可以判断这是一个十六进制整数，其中出现`+2`的后缀是非法的。
- `012823`：八进制整数中不应该出现数字`8`。
- `123F`：`F`不是一个合法的整数后缀。
- `23LU23`：`LU23`不是一个合法的整数后缀。
- `0XFFEuu`：`uu`不是一个合法的整数后缀。
- `0xu`：十六进制整数必须至少包含一位数字。
- `0eu`：八进制整数必须至少包含一位数字。
- `0x12bufb`：`ufb`不是一个合法的整数后缀。
- `12_`：被解析为两个记号`12`和`_`。
- `12g+2`：被解析为三个记号`12g`、`+`和`2`，其中`g`不是一个合法的整数后缀。
- `12p+2`：被解析为一个记号，但是`p+2`不是一个合法的整数后缀。

第二行中的浮点数常量只有前三个是合法的：

- `02582e2`：虽然以`0`开头并且整数部分出现了数字`8`，但是是一个合法的十进制浮点数。
- `00.3e-41`：同上。
- `0X5e5ep2`：十六进制浮点数中可以出现多个`e`。
- `0xp0`：十六进制浮点数的前缀和指数部分之间至少包含一个十六进制数字。
- `3e+1`：指数部分至少包含一个数字。
- `0xfpl`：同上。
- `0x5e.2`：指数部分不允许出现小数点。

- 2.2.2: 不能出现多个小数点。
- 3e5.4: 指数部分不允许出现小数点。
- 0e+9U: U 不是一个合法的浮点数后缀。
- 0x4h3e-1: 指数部分至少包含一个数字。
- .e4: 小数点前后至少有一个数字。
- 0xp+2F: 十六进制浮点数的前缀和指数部分之间至少包含一个十六进制数字。
- 0x.p5L: 十六进制浮点数的前缀和指数部分之间至少包含一个十六进制数字。
- 5.e+2e-4: 出现多个指数部分。
- 0x.2p+2a1: 指数部分只能出现十进制数字。

第三行和第四行分别是未闭合的字符常量和字符串字面量错误。第五行是通过反斜杠换行后遇到 EOF 的错误。

4.4 测试集 4

此测试集主要测试几个转义符号能否正常工作，以及另几个其他类型的错误。

4.4.1 输入

```

1  %:include<string.h>
2  int a<::> = <%0, 0%>
3  const char * str = L"12  \  \      \n\t\v\f      \
4  sd\"'\';
5  int main()
6  <%
7      a<:0:> = 1;
8      return 0;
9  %>#include<stdio.h>
10 /* Unclosed Comment

```

4.4.2 输出（节选）

```

=====Tokens=====
[Location] test/test4.txt:1:1
[Type] Preprocessing

[Location] test/test4.txt:2:1
[Type] Keyword
[Value] int

[Location] test/test4.txt:2:5
[Type] Identifier
[Value] a

[Location] test/test4.txt:2:6

```



```

[Type] Punctuator
[Value] [

[Location] test/test4.txt:2:8
[Type] Punctuator
[Value] ]

[Location] test/test4.txt:2:11
[Type] Punctuator
[Value] =

[Location] test/test4.txt:2:13
[Type] Punctuator
[Value] {

[Location] test/test4.txt:2:15
[Type] Integer Constant
[Value] 0

[Location] test/test4.txt:2:16
[Type] Punctuator
[Value] ,

[Location] test/test4.txt:2:18
[Type] Integer Constant
[Value] 0

[Location] test/test4.txt:2:19
[Type] Punctuator
[Value] }

[Location] test/test4.txt:3:1
[Type] Keyword
[Value] const

[Location] test/test4.txt:3:7
[Type] Keyword
[Value] char

[Location] test/test4.txt:3:12
[Type] Punctuator
[Value] *

[Location] test/test4.txt:3:14
[Type] Identifier
[Value] str

[Location] test/test4.txt:3:18
[Type] Punctuator
[Value] =

```

[Location] test/test4.txt:3:20
[Type] String Literal
[Value] 12 \ \ \n\t\v\f sd\"\'

[Location] test/test4.txt:4:8
[Type] Punctuator
[Value] ;

...

[Location] test/test4.txt:7:5
[Type] Identifier
[Value] a

[Location] test/test4.txt:7:6
[Type] Punctuator
[Value] [

[Location] test/test4.txt:7:8
[Type] Integer Constant
[Value] 0

[Location] test/test4.txt:7:9
[Type] Punctuator
[Value]]

[Location] test/test4.txt:7:12
[Type] Punctuator
[Value] =

[Location] test/test4.txt:7:14
[Type] Integer Constant
[Value] 1

...

[Location] test/test4.txt:9:3
[Type] Unknown
[Value] Unexpected Hash

[Location] test/test4.txt:9:4
[Type] Identifier
[Value] include

[Location] test/test4.txt:9:11
[Type] Punctuator
[Value] <

[Location] test/test4.txt:9:12
[Type] Identifier
[Value] stdio

```
[Location] test/test4.txt:9:17
[Type] Punctuator
[Value] .
```

```
[Location] test/test4.txt:9:18
[Type] Identifier
[Value] h
```

```
[Location] test/test4.txt:9:19
[Type] Punctuator
[Value] >
```

```
[Location] test/test4.txt:10:1
[Type] Unknown
[Value] Unclosed Comment
```

=====Analysis=====

```
[File Name] test/test4.txt
[Rows] 10
[Characters] 186
[Unknown] 2
[Keyword] 5
[Identifier] 7
[Integer Constant] 5
[Float Constant] 0
[Character Constant] 0
[String Literal] 1
[Punctuator] 21
[Preprocessing] 1
[End Of File] 1
```

4.4.3 分析

词法分析器正确解析出`#`、`[`等记号，并且正确识别出了换行的字符串字面量和未闭合的块注释。

同时，对于不在行首的预处理器，词法分析器报出了未识别的符号的错误，同时正常解析了其后的 `include` 等记号，这与 `gcc` 的表现一致。

4.5 测试集 5

此测试集主要测试非法符号是否能够识别。

4.5.1 输入

```
1 ~
2 @
3 $
```

```
4  ##
5  %: %:
6  /* ` @ # */
7  #define \
```

4.5.2 输出

```
=====Tokens=====
[Location] test/test5.txt:1:1
[Type] Unknown
[Value] Unknown Character `

[Location] test/test5.txt:2:1
[Type] Unknown
[Value] Unknown Character @

[Location] test/test5.txt:3:1
[Type] Unknown
[Value] Unknown Character $

[Location] test/test5.txt:4:1
[Type] Unknown
[Value] Unexpected Hash

[Location] test/test5.txt:5:1
[Type] Unknown
[Value] Unexpected Hash

[Location] test/test5.txt:7:1
[Type] Unknown
[Value] Slash Newline at EOF

=====Analysis=====
[File Name] test/test5.txt
[Rows] 7
[Characters] 37
[Unknown] 6
[Keyword] 0
[Identifier] 0
[Integer Constant] 0
[Float Constant] 0
[Character Constant] 0
[String Literal] 0
[Punctuator] 0
[Preprocessing] 0
[End Of File] 1
```

4.5.3 分析

对于非法符号，词法分析器能够正确识别，但是注释中的符号会被跳过。

5 实验总结

本次实验中我上手编写了一个词法分析程序，使我对词法分析的流程更加清楚，对相关知识的掌握更加牢固。

为了实现 C 语言的词法分析，需要有标准的 C 语言词法规则进行参照，而不是凭感觉编写程序。因此我找到了 C99 的 ISO 标准，并且依照标准中定义的词法规则编写程序。标准中对各类记号的文法定义十分清晰，这在一定程度上降低了我编程的难度，但是这其中也有不少细节值得注意。我在第一次编写过程中出现了不少的疏忽，在多次阅读文法的定义，并且精心设计样例调试的过程中解决了许多 bug。

本次实验中我没有完全照搬教材上的实现方式，没有明显地实现一个 DFA，也没有明显的状态转移过程，但是我将自动机的思想渗透到了代码中，通过不同的处理函数来实现对不同种类记号的处理。此外，不同于教材上的缓冲区实现方式，我使用了 C++ 的语法特性，使得维护缓冲区变得更加容易，大大减少编程复杂度。

本次实验除了让我对课内知识有了更多的认识，也使我的 C++ 编程能力和英文文献阅读能力得到提高，我从中收获颇丰。