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

张梓良
2021212484
北京邮电大学
计算机科学与技术

日期：2023 年 10 月 6 日

1 概述

1.1 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。

1.2 实验环境

- flex version 2.5.4
- cmake version 3.27.0-rc4
- gcc version 8.1.0
- Visual Studio Code 1.82.2
- OS: Windows_NT x64 10.0.22621

1.3 实验目的

编写 LEX 源程序，利用 LEX 编译程序自动生成词法分析程序。

2 程序设计说明

2.1 模块划分

词法分析程序分为四个模块：`token`，`error`，`output`，`lex`，其中 `token` 模块定义了所需的数据结构，`error` 模块对词法分析过程中遇到的错误词法进行输出，`output` 模块将词法分析得到的结果输出，`lex` 模块对输入的文件流进行词法分析并将前三个模块组装构成整个词法分析程序。模块间的调用关系如 **图 1** 。

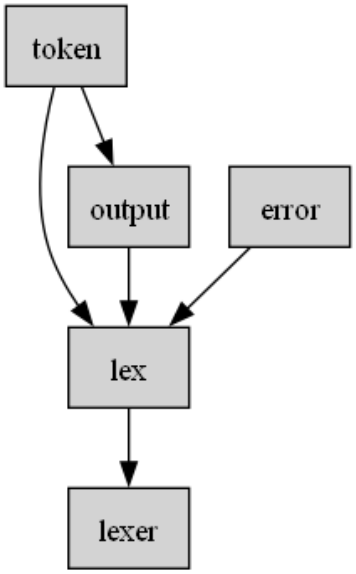

图 1: 模块调用关系

2.2 自定义数据结构

2.2.1 C 语言记号类型

```
enum TokenType
{
    KEYWORD,                // 关键字
    ID,                     // 标志符
    STRING,                 // 字符串常量
    CHAR,                   // 字符常量
    INT,                    // 整型常量
    UINT,                   // 无符号整型常量
    LONG,                   // 长整型常量
    ULONG,                  // 无符号长整型
    LONGLONG,               // 长长整型常量
    ULONGLONG,              // 无符号长长整型常量
    FLOAT,                  // 单精度浮点数
    DOUBLE,                 // 双精度浮点数
    LONGDOUBLE,             // 长双精度浮点数
    RELATION_OPERATOR,      // 关系运算符
```

```
    ASSIGN_OPERATOR,        // 赋值运算符
    AGORITHM_OPERATOR,      // 算术运算符
    LOGICAL_OPERATOR,       // 逻辑运算符
    BITWISE_OPERATOR,       // 位运算符
    QUESTION_MARK,          //"?"
    COLON,                  //":"
    SEMICOLON,              //";"
    LEFT_SQUARE_BRACKET,    //"["
    RIGHT_SQUARE_BRACKET,   //"]"
    LEFT_PARENTHESE,        //"("
    RIGHT_PARENTHESE,       //")"
    LEFT_BRACE,             //"{"
    RIGHT_BRACE,            //"}"
    DOT,                    //"."
    COMMA,                  //","
    ARROW,                  //"->"
    ANNOTATION,             // 注释
};
```

采用 enum 类型存储记号类型，建立记号类型与非负整数之间的映射，便于词法分析时函数之间记号类型的传递。

2.2.2 记号

```
struct Token
{
    enum TokenType type; // 记号类型
    char value[200];     // 记号值
    int line;            // 记号所在行数
    int column;          // 记号所在列数
};
```

一个记号中存储了记号类型、记号属性值、记号所在行数以及记号所在列数，其中记号类型采用先前已经自定义的数据类型。记号的表示形式为 <line:column><type, value> 。

2.3 词法分析

在 lex.l 中定义识别各类记号的正则表达式，再通过 flex 编译程序生成 lex.yy.c 文件来替代手工编写的 lexical_analysis.cpp 文件以实现词法分析程序。

2.3.1 设计流程

参考 ISO_C99_definition 中对各类记号的形式化定义画出识别各类记号的 DFA，再根据 DFA 写出相应正则表达式。

2.3.2 关键字的识别

ISO_C99_definition 中对关键字的定义如 **图 2** 所示。

keyword: one of

auto	**enum**	**restrict**	**unsigned**
break	**extern**	**return**	**void**
case	**float**	**short**	**volatile**
char	**for**	**signed**	**while**
const	**goto**	**sizeof**	**_Bool**
continue	**if**	**static**	**_Complex**
default	**inline**	**struct**	**_Imaginary**
do	**int**	**switch**	
double	**long**	**typedef**	
else	**register**	**union**	

图 2: C99 对关键字的规定

据此写出正则表达式：

```
keyword   (auto)|(break)|(case)|(char)|(const)|(continue)|(default)|(do)|(double)|(
    else)|(enum)|(extern)|(float)|(for)|(goto)|(if)|(inline)|(int)|(long)|(register)
    |(restrict)|(return)|(short)|(signed)|(sizeof)|(static)|(struct)|(switch)|(
    typedef)|(union)|(unsigned)|(void)|(volatile)|(while)|(_Bool)|(_Complex)|(
    _Complex)|(_Imaginary)
```

2.3.3 标识符的识别

ISO_C99_definition 中对标志符的定义如 **图 3** 所示。标识符只能由 _ \ letter 开头，可以包含 _ \ letter \ digit 。

Syntax

identifier:
identifier-nondigit
identifier identifier-nondigit
identifier digit

identifier-nondigit:
nondigit
universal-character-name
other implementation-defined characters

nondigit: one of

_	**a**	**b**	**c**	**d**	**e**	**f**	**g**	**h**	**i**	**j**	**k**	**l**	**m**
	n	**o**	**p**	**q**	**r**	**s**	**t**	**u**	**v**	**w**	**x**	**y**	**z**
	A	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**	**J**	**K**	**L**	**M**
	N	**O**	**P**	**Q**	**R**	**S**	**T**	**U**	**V**	**W**	**X**	**Y**	**Z**

digit: one of

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**

图 3: C99 对标识符的规定

采用 dot 语言实现识别标识符的 DFA：

```
digraph G{
    rankdir=LR
    node[shape = circle]
    ID [shape = doublecircle]
    0 -> 1 [label = "_ / letter"]
    1 -> 1 [label = "_ / letter / digit"]
    1 -> ID [label = "other"]
}
```

据此写出正则表达式：

```
letter      [A-Za-z]
digit       [0-9]
id          ({letter}|_)({letter}|{digit}|_)*
```

2.3.4 常数的识别

常数可以按照两类标准分类：

- 以 0 开头还是非 0 数字开头还是 . 开头
- 整型常数还是浮点型常数

其中以 0 开头的常数又可以分为：

- 整数 0
- 以 0 开头的八进制整型常数
- 以 0 开头的十六进制整型常数
- 以 0 开头的浮点型常数

其中以非 0 数字开头的常数又可以分为：

- 十进制整型常数
- 浮点型常数

其中以 . 开头的常数只能为浮点型常数

其中整形常数又可以分为：

- 十进制整型常数
- 八进制整型常数
- 十六进制整型常数

其中浮点型常数又可以分为：

- `float`（f / F 后缀）
- `double`（无后缀）

- long double (l / L 后缀)

各种进制的整型常数又可以分为:

- int (无后缀)
- unsigned int (u / U 后缀)
- long (l / L 后缀)
- unsigned long ((u / U)+ (l / L)后缀)
- long long ((l / L)+ (l / L)后缀)
- unsigned long long((u / U)+ (l / L)+ (l / L)后缀)

根据以上分类，采用 dot 语言对识别常数的 DFA 进行定义:

```
digraph G{
    rankdir = LR
    splines = ortho
    node[shape = circle]
    DOT[shape = doublecircle, label = "DOT(4)"]
    INT[shape = doublecircle]
    UINT[shape = doublecircle, label = "UINT(30)"]
    ULONG[shape = doublecircle, label = "ULONG(32)"]
    ULONGLONG[shape = doublecircle]
    LONG[shape = doublecircle, label = "LONG(31)"]
    LONGLONG[shape = doublecircle]
    FLOAT[shape = doublecircle]
    DOUBLE[shape = doublecircle]
    LONGDOUBLE[shape = doublecircle]
    0 -> 2 [label = "digit(except 0)"]
    0 -> 3 [label = "0"]
    0 -> DOT [label = "."]
    2 -> 2 [label = "digit"]
    2 -> 22 [label = "."]
    2 -> 23 [label = "e / E"]
    2 -> UINT [label = "u / U"]
    2 -> LONG [label = "l / L"]
    UINT -> ULONG [label = "l / L"]
    ULONG -> ULONGLONG [label = "l / L"]
    LONG -> LONGLONG [label = "l / L"]
    2 -> INT [label = "INT"]
    22 -> 22 [label = "digit"]
    22 -> 23 [label = "e / E"]
    22 -> FLOAT [label = "f / F"]
    22 -> LONGDOUBLE [label = "l / L"]
    22 -> DOUBLE [label = "other"]
    23 -> 24 [label = "+ / -"]
```

```
    23 -> 25 [label = "digit"]
    24 -> 25 [label = "digit"]
    25 -> 25 [label = "digit"]
    25 -> FLOAT [label = "f / F"]
    25 -> LONGDOUBLE [label = "l / L"]
    25 -> DOUBLE [label = "other"]
    3 -> 22 [label = "."]
    3 -> 26 [label = "oct_digt"]
    3 -> 27 [label = "8 / 9"]
    3 -> 23 [label = "e / E"]
    3 -> 28 [label = "x / X"]
    3 -> INT [label = "other"]
    3 -> UINT [label = "u / U"]
    3 -> LONG [label = "l / L"]
    26 -> 27 [label = "8 / 9"]
    26 -> 26 [label = "oct_digit"]
    26 -> 23 [label = "e / E"]
    26 -> UINT [label = "u / U"]
    26 -> LONG [label = "l / L"]
    26 -> 22 [label = "."]
    27 -> 27 [label = "digit"]
    27 -> 22 [label = "."]
    27 -> 23 [label = "e / E"]
    28 -> 29 [label = "hex_digit"]
    29 -> 29 [label = "hex_digit"]
    29 -> INT [label = "other"]
    29 -> UINT [label = "u / U"]
    29 -> LONG [label = "l / L"]
    DOT -> 22 [label = "digit"]
}
```

据此写出正则表达式：

```
digit       [0-9]
octal_digit [0-7]
hex_digit   [0-9a-fA-F]
int     [1-9]{digit}*|0{octal_digit}*|0[xX]{hex_digit}+
uint    {int}(u|U)
long    {int}(l|L)
ulong   {int}(u|U)(l|L)
longlong  {int}(l|L)(l|L)
ulonglong {int}(u|U)(l|L)(l|L)
double    ({digit}+\.{digit}*|\.{digit}+)([eE][\+\-]?{digit}+)?|{digit}+[eE
    ][\+\-]?{digit}+
float   {double}(f|F)
longdouble  {double}(l|L)
```

2.3.5 字符常量的识别

DOT 语言定义的识别字符常量的 DFA：

```
digraph G{
    rankdir=LR
    splines = line
    node[shape = circle]
    CHAR [shape = doublecircle, fontsize = 10]
    ERROR [shape = doublecircle, label = "ERROR(21)", fontsize = 10]
    0 -> 7 [label = "'"]
    7 -> ERROR [label = "' / \\n / EOF "]
    7 -> 35 [label = "other"]
    7 -> 34 [label = "\\"]
    34 -> 35 [label = "legal escape characters"]
    35 -> CHAR [label = "'"]
}
```

据此写出正则表达式：

```
char    \'([^\'\n]|\\.)+\'
```

除此之外，我们需要考虑字符常量为空，eg.''，以及字符常量未闭合的错误情况，eg.'a。

对以上错误情况进行匹配的正则表达式：

```
unclosed_char     \'[^\'\n]*
empty_char        \'\'
```

2.3.6 字符串常量的识别

字符串常量的识别与字符常量的识别类似，同样需要考虑未闭合的错误情况，但字符串常量允许为空。

下面给出用 DOT 语言定义的识别字符串常量的 DFA：

```
digraph G{
    rankdir=LR
    splines = line
    node[shape = circle]
    STRING [shape = doublecircle, fontsize = 10]
    ERROR [shape = doublecircle, label = "ERROR(21)", fontsize = 10]
    0 -> 6 [label = "\""]
    6 -> 6 [label = "other"]
    6 -> ERROR [label = "\\n / EOF"]
    6 -> 33 [label = "\\"]
    6 -> STRING [label = "\""]
    33 -> 6 [label = "legal escape characters"]
}
```

据此写出正则表达式：

```
str      \"([^\"\n]|\\.)*\"
unclosed_string      \"[^\"\n]*
```

2.3.7 运算符的识别

运算符分为五大类:RELATION_OPERATOR, ASSIGN_OPERATOR, AGORITHM_OPERATOR, LOGICAL_OPERATOR, BITWISE_OPERATOR

据此写出正则表达式:

```
relation   (\<)|(\>)|(\<=)|(\>=)|(==)|(!=)
assign     (=)|(\+=)|(\-=)|(\*=)|(\/=)|(\%=)|(&=)|(\|=)|(\^=)|(\<\<=)|(\>\>=)
agorithm   (\+)|(\-)|(\*)|(\/)|(\%)|(\+\+)|(\-\-)
logical    (&&)|(\|\|)|(!)
bitwise    (&)|(\|)|(\^)|(~)|(\<\<)|(\>\>)
```

2.3.8 注释的识别

注释分为单行注释和多行注释，二者都以 / 开头。单行注释遇到 \n 或 EOF 结束；多行注释当匹配到对应的 */ 时结束，因此需要进行闭合检测，若读到文件末尾还没有与 /* 匹配的 */，就输出多行注释未闭合的错误。

下面给出用 DOT 语言定义的识别注释的 DFA:

```
digraph G{
    rankdir=LR
    node[shape = circle]
    ANNOTATION [shape = doublecircle, fontsize = 10]
    0 -> 11 [label = "/"]
    11 -> 36 [label = "/"]
    11 -> 37 [label = "*"]
    36 -> 36 [label = "other"]
    36 -> ANNOTATION [label = "\\n / EOF"]
    37 -> 37 [label = "other"]
    37 -> 38 [label = "*"]
    38 -> 37 [label = "other"]
    38 -> 38 [label = "*"]
    38 -> ANNOTATION [label = "/"]
}
```

据此写出正则表达式:

```
singleline_annotation    (\/\/)[^\n]*
multiline_annotation     (\/\*)(.|\n)*(\*\/)

unclosed_annotation    (\/\*)(.*)
```

2.3.9 其他记号的识别

其他记号的识别较为简单，直接在翻译规则部分写出即可。

2.4 词法错误

2.4.1 “错误”类型的定义

```
enum ErrorType
{
    UNKNOWN_SYMBOL,      // 未知符号
    UNCLOSED_STRING,     // 未闭合字符串
    UNCLOSED_CHAR,       // 未闭合字符
    EMPTY_CHAR,          // 空字符
    UNCLOSED_ANNOTATION // 未闭合注释
};
```

2.4.2 “错误”的正则匹配

```
unclosed_string     \"[^\"\n]*
unclosed_char       \'[^\'\n]*
empty_char          \'\'
unclosed_annotation    (\/\*)(.*)
unknown_symbol      [‘@$]
```

2.4.3 “错误”信息的输出

错误输出函数 error 按照 *"Error: line:col: 错误类型: 错误内容"* 的格式输出错误信息。

eg. `Error: 29:14:Illegal const numbers: 09`

处理完错误后，词法分析函数继续执行，对剩余的文件流进行词法分析，实现了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的功能。

2.5 记号流的存储及输出

识别出的记号通过定义在 lex.l 中的 add_token 函数加入到记号流 `struct Token token_streams [MAX_TOKEN_NUM]` 中。

```
void add_token(TokenType type, char *value, int line, int col){
    token_streams[token_num].type = type;
    strcpy(token_streams[token_num].value, value);
    token_streams[token_num].line = line;
    token_streams[token_num].column = col;
    token_num++;
}
```

记号流通过定义在 output.h 中的 output 函数以特定格式输出，详细格式见**程序测试**中的**测试结果**。

```
/**
 * @brief 输出记号流
 *
```

```
 * @param token_streams 记号流
 * @param char_num 字符数
 * @param line 行数
 * @param token_num 记号数
 */
void output(struct Token *token_streams, int char_num, int line, int token_num);
```

3 程序测试

3.1 测试集 1

此测试集用于测试程序是否能够准确识别标识符及关键字。

3.1.1 测试内容

```
#include <stdio.h>

int main()
{
    int _123;
    unsigned int _abc;
    long abc_;
    long long abc_123;
    unsigned long long 123abc;
    double _double;
    char 345;
}
```

3.1.2 测试结果

```
========================Token Streams========================
<3:1><Keyword, int>     <3:5><Identifier, main>
<3:9><Left Parenthese, (>       <3:10><Right Parenthese, )>
<4:1><Left Brace, {>    <5:5><Keyword, int>
<5:9><Identifier, _123> <5:13><Semicolon, ;>
<6:5><Keyword, unsigned>        <6:14><Keyword, int>
<6:18><Identifier, _abc>        <6:22><Semicolon, ;>
<7:5><Keyword, long>    <7:10><Identifier, abc_>
<7:14><Semicolon, ;>    <8:5><Keyword, long>
<8:10><Keyword, long>   <8:15><Identifier, abc_123>
<8:22><Semicolon, ;>    <9:5><Keyword, unsigned>
<9:14><Keyword, long>   <9:19><Keyword, long>
<9:24><Int, 123>        <9:27><Identifier, abc>
<9:30><Semicolon, ;>    <10:5><Keyword, double>
<10:12><Identifier, _double>    <10:19><Semicolon, ;>
```

```
<11:5><Keyword, char>   <11:10><Int, 345>
<11:13><Semicolon, ;>   <12:1><Right Brace, }>

========Analysis Result========
Total lines: 12
Total characters: 156
Total tokens: 32
Total keywords: 12
Total identifiers: 7
Total const strings: 0
Total const characters: 0
Total const integers: 2
Total const floats: 0
Total operators: 0
Total others: 11
```

3.1.3 结果分析

对于两个非法标识符 123abc 和 345，词法分析程序对前者的识别结果为 <9:24><Int, 123> 和 <9:27><Identifier, abc>，符合预期设想；对后者的识别结果为 <11:10><Int, 345>，也符合预期设想。标识符虽然非法，但词法分析程序未输出错误信息，因为对于该类错误的判定属于语法分析的任务。

其余合法的标识符和关键字也都能够被准确识别。对于记号类别的统计信息也符合预期。

3.2 测试集 2

此测试集用于测试程序是否能够准确识别字符常量并判定其合法性。

3.2.1 测试内容

```c
#include <stdio.h>

int main()
{
    'a';
    '\0', '\a', '\b', '\t', '\n', '\v', '\f', '\r', '\"', '\'', '\?', '\\';
    '';
    'b;
    '\p';
}
```

3.2.2 测试结果

```
Error: 7:5: Empty const char: ''
Error: 8:5: Unclosed const char: 'b;
```

```
3
4  ======================Token Streams======================
5  <3:1><Keyword, int>      <3:5><Identifier, main>
6  <3:9><Left Parenthese, (>        <3:10><Right Parenthese, )>
7  <4:1><Left Brace, {>     <5:5><Char, 'a'>
8  <5:8><Semicolon, ;>      <6:5><Char, '\0'>
9  <6:9><Comma, ,> <6:11><Char, '\a'>
10 <6:15><Comma, ,>         <6:17><Char, '\b'>
11 <6:21><Comma, ,>         <6:23><Char, '\t'>
12 <6:27><Comma, ,>         <6:29><Char, '\n'>
13 <6:33><Comma, ,>         <6:35><Char, '\v'>
14 <6:39><Comma, ,>         <6:41><Char, '\f'>
15 <6:45><Comma, ,>         <6:47><Char, '\r'>
16 <6:51><Comma, ,>         <6:53><Char, '\"'>
17 <6:57><Comma, ,>         <6:59><Char, '\''>
18 <6:63><Comma, ,>         <6:65><Char, '\?'>
19 <6:69><Comma, ,>         <6:71><Char, '\\'>
20 <6:75><Semicolon, ;>     <7:7><Semicolon, ;>
21 <9:5><Char, '\p'>        <9:9><Semicolon, ;>
22 <10:1><Right Brace, }>
23
24 ========Analysis Result========
25 Total lines: 10
26 Total characters: 127
27 Total tokens: 35
28 Total keywords: 1
29 Total identifiers: 1
30 Total const strings: 0
31 Total const characters: 14
32 Total const integers: 0
33 Total const floats: 0
34 Total operators: 0
35 Total others: 19
```

3.2.3 结果分析

对于两个非法的字符常量 ‘’，‘b，词法分析程序都能识别出错误，并输出相应错误信息（测试结果 1-2 行），符合预期设想。

注：用 LEX 编写的词法分析程序未对转义字符进行检查，因此'\p' 不会被判定为错误的字符常量。

其余合法的字符常量也都能够被准确识别。对于记号类别的统计信息也符合预期。

3.3 测试集 3

此测试集用于测试程序是否能够准确识别字符串常量并判定其合法性。

3.3.1 测试内容

```
#include <stdio.h>

int main()
{
    "";
    "abc";
    "def;
    "abc\"";
    "def\p";
}
```

3.3.2 测试结果

```
Error: 7:5: Unclosed const string: "def;

=======================Token Streams=======================
<3:1><Keyword, int>     <3:5><Identifier, main>
<3:9><Left Parenthese, (>       <3:10><Right Parenthese, )>
<4:1><Left Brace, {>    <5:5><String, "">
<5:7><Semicolon, ;>     <6:5><String, "abc">
<6:10><Semicolon, ;>    <8:5><String, "abc\"">
<8:12><Semicolon, ;>    <9:5><String, "def\p">
<9:12><Semicolon, ;>    <10:1><Right Brace, }>

========Analysis Result========
Total lines: 10
Total characters: 71
Total tokens: 14
Total keywords: 1
Total identifiers: 1
Total const strings: 4
Total const characters: 0
Total const integers: 0
Total const floats: 0
Total operators: 0
Total others: 8
```

3.3.3 结果分析

对于非法的字符串常量 *"def*，词法分析程序都能识别出错误，并输出相应错误信息（测试结果 1 行），符合预期设想。

注：用 LEX 编写的词法分析程序未对转义字符进行检查，因此 *"def\p"* 不会被判定为错误的字符串常量。

其余合法的字符串常量也都能够被准确识别。对于记号类别的统计信息也符合预期。

3.4 测试集 4

此测试集用于测试程序是否能够准确识别常数并判定其合法性。

3.4.1 测试内容

```
#include <stdio.h>

int main()
{
    123, 123u, 123l, 123ul, 123ll, 123ull;
    123., 123.456, 123.456e5, 123.456e+5, 123.456e-5, 123e5, 123.e5;
    123.456e+5f, 123.456e+5l;
    0, 0U, 0L, 0UL, 0LL, 0ULL;
    0123, 0234U, 0345L, 0456UL, 0567LL;
    0789.;
    0789;
    0x123, 0x234U, 0x345L, 0x456uL, 0x789lL, 0x89aULl, 0x9ab,0xabc;
    0.123, 0.123e5, 0.123e+5, 0.123e-5;
    123e+a;
    123e-;
    .123F, .123e5L, .123e+5, .123e-5;
}
```

3.4.2 测试结果

```
========================Token Streams========================
<3:1><Keyword, int>     <3:5><Identifier, main>
<3:9><Left Parenthese, (>       <3:10><Right Parenthese, )>
<4:1><Left Brace, {>    <5:5><Int, 123>
<5:8><Comma, ,> <5:10><Unsigned Int, 123u>
<5:14><Comma, ,>        <5:16><Long, 123l>
<5:20><Comma, ,>        <5:22><Unsigned Long, 123ul>
<5:27><Comma, ,>        <5:29><Long Long, 123ll>
<5:34><Comma, ,>        <5:36><Unsigned Long Long, 123ull>
<5:42><Semicolon, ;>    <6:5><Double, 123.>
<6:9><Comma, ,> <6:11><Double, 123.456>
<6:18><Comma, ,>        <6:20><Double, 123.456e5>
<6:29><Comma, ,>        <6:31><Double, 123.456e+5>
<6:41><Comma, ,>        <6:43><Double, 123.456e-5>
<6:53><Comma, ,>        <6:55><Double, 123e5>
<6:60><Comma, ,>        <6:62><Double, 123.e5>
<6:68><Semicolon, ;>    <7:5><Float, 123.456e+5f>
<7:16><Comma, ,>        <7:18><Long Double, 123.456e+5l>
<7:29><Semicolon, ;>    <8:5><Int, 0>
<8:6><Comma, ,> <8:8><Unsigned Int, 0U>
<8:10><Comma, ,>        <8:12><Long, 0L>
```

```
<8:14><Comma, ,>        <8:16><Unsigned Long, 0UL>
<8:19><Comma, ,>        <8:21><Long Long, 0LL>
<8:24><Comma, ,>        <8:26><Unsigned Long Long, 0ULL>
<8:30><Semicolon, ;>    <9:5><Int, 0123>
<9:9><Comma, ,> <9:11><Unsigned Int, 0234U>
<9:16><Comma, ,>        <9:18><Long, 0345L>
<9:23><Comma, ,>        <9:25><Unsigned Long, 0456UL>
<9:31><Comma, ,>        <9:33><Long Long, 0567LL>
<9:39><Semicolon, ;>    <10:5><Double, 0789.>
<10:10><Semicolon, ;>   <11:5><Int, 07>
<11:7><Int, 89> <11:9><Semicolon, ;>
<12:5><Int, 0x123>      <12:10><Comma, ,>
<12:12><Unsigned Int, 0x234U>   <12:18><Comma, ,>
<12:20><Long, 0x345L>   <12:26><Comma, ,>
<12:28><Unsigned Long, 0x456uL> <12:35><Comma, ,>
<12:37><Long Long, 0x789lL>     <12:44><Comma, ,>
<12:46><Unsigned Long Long, 0x89aULl>   <12:54><Comma, ,>
<12:56><Int, 0x9ab>     <12:61><Comma, ,>
<12:62><Int, 0xabc>     <12:67><Semicolon, ;>
<13:5><Double, 0.123>   <13:10><Comma, ,>
<13:12><Double, 0.123e5>        <13:19><Comma, ,>
<13:21><Double, 0.123e+5>       <13:29><Comma, ,>
<13:31><Double, 0.123e-5>       <13:39><Semicolon, ;>
<14:5><Int, 123>        <14:8><Identifier, e>
<14:9><Agorithm Operator, +>    <14:10><Identifier, a>
<14:11><Semicolon, ;>   <15:5><Int, 123>
<15:8><Identifier, e>   <15:9><Agorithm Operator, ->
<15:10><Semicolon, ;>   <16:5><Float, .123F>
<16:10><Comma, ,>       <16:12><Long Double, .123e5L>
<16:19><Comma, ,>       <16:21><Double, .123e+5>
<16:28><Comma, ,>       <16:30><Double, .123e-5>
<16:37><Semicolon, ;>   <17:1><Right Brace, }>

========Analysis Result========
Total lines: 17
Total characters: 419
Total tokens: 104
Total keywords: 1
Total identifiers: 4
Total const strings: 0
Total const characters: 0
Total const integers: 29
Total const floats: 18
Total operators: 2
Total others: 50
```

3.4.3 结果分析

注：对于三个非法的常数 0789，123e+a，123e-，在此处不会被判定为非法常数，是因为定义的相关正则表达式会将以上常数拆分成两个合法的单词，例如 0789 会被拆分成 07 和 89。

其余合法的常数也都能够被准确识别。对于记号类别的统计信息也符合预期。

3.5 测试集 5

此测试集用于测试程序是否能够准确识别运算符、其他符号以及注释。

3.5.1 测试内容

```
#include <stdio.h>

int main()
{
    +, -, *, /, %, ++, --, &&, ||, !, &, |, ~, ^, << ;
    >>, =, +=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=, == ;
    !=, >, <, >=, <= ;

    ?, :, ;
    , [, ], (, ),
    { , }, ., , , ->;

    // this is a sigle line annotation

    /* this is a multiple line annotation


    */

    ` // UNKNOWN_SYMBOL
}

/* this is an unclosed multiple line annotation
```

3.5.2 测试结果

```
Error: 17:5: Unknown symbol: `
Error: 20:1: Unclosed annotation: /* this is an unclosed multiple line annotation

=======================Token Streams=======================
<3:1><Keyword, int>     <3:5><Identifier, main>
<3:9><Left Parenthese, (>       <3:10><Right Parenthese, )>
<4:1><Left Brace, {>    <5:5><Agorithm Operator, +>
<5:6><Comma, ,> <5:8><Agorithm Operator, ->
```

```
<5:9><Comma, ,> <5:11><Agorithm Operator, *>
<5:12><Comma, ,>          <5:14><Agorithm Operator, />
<5:15><Comma, ,>          <5:17><Agorithm Operator, %>
<5:18><Comma, ,>          <5:20><Agorithm Operator, ++>
<5:22><Comma, ,>          <5:24><Agorithm Operator, -->
<5:26><Comma, ,>          <5:28><Logical Operator, &&>
<5:30><Comma, ,>          <5:32><Logical Operator, ||>
<5:34><Comma, ,>          <5:36><Logical Operator, !>
<5:37><Comma, ,>          <5:39><Bitwise Operator, &>
<5:40><Comma, ,>          <5:42><Bitwise Operator, |>
<5:43><Comma, ,>          <5:45><Bitwise Operator, ~>
<5:46><Comma, ,>          <5:48><Bitwise Operator, ^>
<5:49><Comma, ,>          <5:51><Bitwise Operator, <<>
<5:54><Semicolon, ;>      <6:5><Bitwise Operator, >>>
<6:7><Comma, ,> <6:9><Assign Operator, =>
<6:10><Comma, ,>          <6:12><Assign Operator, +=>
<6:14><Comma, ,>          <6:16><Assign Operator, -=>
<6:18><Comma, ,>          <6:20><Assign Operator, *=>
<6:22><Comma, ,>          <6:24><Assign Operator, /=>
<6:26><Comma, ,>          <6:28><Assign Operator, %=>
<6:30><Comma, ,>          <6:32><Assign Operator, &=>
<6:34><Comma, ,>          <6:36><Assign Operator, |=>
<6:38><Comma, ,>          <6:40><Assign Operator, ^=>
<6:42><Comma, ,>          <6:44><Assign Operator, <<=>
<6:47><Comma, ,>          <6:49><Assign Operator, >>=>
<6:52><Comma, ,>          <6:54><Relation Operator, ==>
<6:57><Semicolon, ;>      <7:5><Relation Operator, !=>
<7:7><Comma, ,> <7:9><Relation Operator, >>
<7:10><Comma, ,>          <7:12><Relation Operator, <>
<7:13><Comma, ,>          <7:15><Relation Operator, >=>
<7:17><Comma, ,>          <7:19><Relation Operator, <=>
<7:22><Semicolon, ;>      <9:5><Question Mark, ?>
<9:6><Comma, ,> <9:8><Colon, :>
<9:9><Comma, ,> <9:11><Semicolon, ;>
<10:5><Comma, ,>          <10:7><Left Square Bracket, [>
<10:8><Comma, ,>          <10:10><Right Square Bracket, ]>
<10:11><Comma, ,>         <10:13><Left Parenthese, (>
<10:14><Comma, ,>         <10:16><Right Parenthese, )>
<10:17><Comma, ,>         <11:5><Left Brace, {>
<11:7><Comma, ,>          <11:9><Right Brace, }>
<11:10><Comma, ,>         <11:12><Dot, .>
<11:13><Comma, ,>         <11:15><Comma, ,>
<11:17><Comma, ,>         <11:19><Arrow, ->>
<11:21><Semicolon, ;>     <18:1><Right Brace, }>

========Analysis Result========
Total lines: 21
```

```
Total characters: 372
Total tokens: 96
Total keywords: 1
Total identifiers: 1
Total const strings: 0
Total const characters: 0
Total const integers: 0
Total const floats: 0
Total operators: 33
Total others: 61
```

3.5.3 结果分析

对于未知字符错误:`和未闭合的多行注释错误:/* this is an unclosed multiple line annotation，词法分析程序都能够准确识别，并输出相应错误信息（测试结果 1-2 行），符合预期设想。

对运算符的识别实现了**超前扫描**功能，能够区分例如 <，<=，<<，<<= 等前缀相同的运算符。同时对于 .，;，[等其他符号也能够准确识别。遇到注释能够识别并跳过。对于记号类别的统计信息也符合预期。

3.6 测试集 6

此测试集用于综合测试程序。

3.6.1 测试内容

```c
#include <stdio.h>
#include <math.h>

struct node
{
    int a;
    int b;
};

int main()
{
    int i = 123;
    unsigned int ui = 0456u;
    long long ll = 0x123456789abcdefll;

    // this is a single line comment

    float f = 1.f;
    double d = 1.234e+10;
    long double ld = 1.234e-10l;
```

```

    struct node n1;
    struct node *temp = &n1;
    temp->b = 123ull;

    /*
    this is a multi-line comment
    */

    char c = '\?';
    if (c <= '\f' && fabs(d - 123e7) > 1e-10)
    {
        c = '\f';
        i++;
        ui << 1;
        ll <<= 2;
        f = (i == 123) ? 1.0 : 0;
        ld -= 1.0;
        n1.a = 123;
    }
    printf("This is a string");

    return 0;
}
```

3.6.2 测试结果

```
========================Token Streams========================
<4:1><Keyword, struct>  <4:8><Identifier, node>
<5:1><Left Brace, {>    <6:5><Keyword, int>
<6:9><Identifier, a>    <6:10><Semicolon, ;>
<7:5><Keyword, int>     <7:9><Identifier, b>
<7:10><Semicolon, ;>    <8:1><Right Brace, }>
<8:2><Semicolon, ;>     <10:1><Keyword, int>
<10:5><Identifier, main>        <10:9><Left Parenthese, (>
<10:10><Right Parenthese, )>    <11:1><Left Brace, {>
<12:5><Keyword, int>    <12:9><Identifier, i>
<12:11><Assign Operator, =>     <12:13><Int, 123>
<12:16><Semicolon, ;>   <13:5><Keyword, unsigned>
<13:14><Keyword, int>   <13:18><Identifier, ui>
<13:21><Assign Operator, =>     <13:23><Unsigned Int, 0456u>
<13:28><Semicolon, ;>   <14:5><Keyword, long>
<14:10><Keyword, long>  <14:15><Identifier, ll>
<14:18><Assign Operator, =>     <14:20><Long Long, 0x123456789abcdefll>
<14:39><Semicolon, ;>   <18:5><Keyword, float>
<18:11><Identifier, f>  <18:13><Assign Operator, =>
<18:15><Float, 1.f>     <18:18><Semicolon, ;>
```

```
<19:5><Keyword, double> <19:12><Identifier, d>
<19:14><Assign Operator, =>     <19:16><Double, 1.234e+10>
<19:25><Semicolon, ;>   <20:5><Keyword, long>
<20:10><Keyword, double>        <20:17><Identifier, ld>
<20:20><Assign Operator, =>     <20:22><Long Double, 1.234e-10l>
<20:32><Semicolon, ;>   <22:5><Keyword, struct>
<22:12><Identifier, node>       <22:17><Identifier, n1>
<22:19><Semicolon, ;>   <23:5><Keyword, struct>
<23:12><Identifier, node>       <23:17><Agorithm Operator, *>
<23:18><Identifier, temp>       <23:23><Assign Operator, =>
<23:25><Bitwise Operator, &>    <23:26><Identifier, n1>
<23:28><Semicolon, ;>   <24:5><Identifier, temp>
<24:9><Arrow, ->>       <24:11><Identifier, b>
<24:13><Assign Operator, =>     <24:15><Unsigned Long Long, 123ull>
<24:21><Semicolon, ;>   <28:5><Keyword, char>
<28:10><Identifier, c>  <28:12><Assign Operator, =>
<28:14><Char, '\?'>     <28:18><Semicolon, ;>
<29:5><Keyword, if>     <29:8><Left Parenthese, (>
<29:9><Identifier, c>   <29:11><Relation Operator, <=>
<29:14><Char, '\f'>     <29:19><Logical Operator, &&>
<29:22><Identifier, fabs>       <29:26><Left Parenthese, (>
<29:27><Identifier, d>  <29:29><Agorithm Operator, ->
<29:31><Double, 123e7>  <29:36><Right Parenthese, )>
<29:38><Relation Operator, >>   <29:40><Double, 1e-10>
<29:45><Right Parenthese, )>    <30:5><Left Brace, {>
<31:9><Identifier, c>   <31:11><Assign Operator, =>
<31:13><Char, '\f'>     <31:17><Semicolon, ;>
<32:9><Identifier, i>   <32:10><Agorithm Operator, ++>
<32:12><Semicolon, ;>   <33:9><Identifier, ui>
<33:12><Bitwise Operator, <<>   <33:15><Int, 1>
<33:16><Semicolon, ;>   <34:9><Identifier, ll>
<34:12><Assign Operator, <<=>   <34:16><Int, 2>
<34:17><Semicolon, ;>   <35:9><Identifier, f>
<35:11><Assign Operator, =>     <35:13><Left Parenthese, (>
<35:14><Identifier, i>  <35:16><Relation Operator, ==>
<35:19><Int, 123>       <35:22><Right Parenthese, )>
<35:24><Question Mark, ?>       <35:26><Double, 1.0>
<35:30><Colon, :>       <35:32><Int, 0>
<35:33><Semicolon, ;>   <36:9><Identifier, ld>
<36:12><Assign Operator, -=>    <36:15><Double, 1.0>
<36:18><Semicolon, ;>   <37:9><Identifier, n1>
<37:11><Dot, .> <37:12><Identifier, a>
<37:14><Assign Operator, =>     <37:16><Int, 123>
<37:19><Semicolon, ;>   <38:5><Right Brace, }>
<39:5><Identifier, printf>      <39:11><Left Parenthese, (>
<39:12><String, "This is a string">     <39:30><Right Parenthese, )>
<39:31><Semicolon, ;>   <41:5><Keyword, return>
```

```
<41:12><Int, 0> <41:13><Semicolon, ;>
<42:1><Right Brace, }>

========Analysis Result========
Total lines: 42
Total characters: 648
Total tokens: 135
Total keywords: 18
Total identifiers: 31
Total const strings: 1
Total const characters: 3
Total const integers: 10
Total const floats: 7
Total operators: 23
Total others: 42
```

3.6.3 结果分析

经检验，词法分析程序能够准确识别出各类记号，对于记号的统计也准确无误。通过以上 6 个测试集，验证了程序的正确性和健壮性。

4 用户说明

.vscode 文件中配置了编译参数，通过编译源程序可以在根目录下得到可执行程序 Lexer.exe。运行该程序，并在命令行输入要进行词法分析的源文件，eg. test1.txt，即可对源文件进行词法分析。

详细说明见说明文档 README.md。

5 实验总结

通过本次实验，我用 LEX + C 语言实现了一个 C 语言的词法分析程序，让我对词法分析的过程和原理有了更加深刻的理解和体会。

本次实验的核心在于定义出识别各类记号的正则表达式,在这个过程中,我参考了 ISO_C99_definition 官方文档中词法分析的相关内容，提升了我阅读文档能力，同时也参考了教材中以及网络资料中对 LEX 语法的介绍。

在调试过程中，因为 LEX 语法的不规范导致编译一直提示错误 unrecognized rule，最终经过反复修改与尝试终于将错误找出:

```
{relation}{add_token(RELATION_OPERATOR, yytext, line + 1, col + 1); char_num +=
    yyleng; col += yyleng;}
```

经修改为:

```
{relation} {add_token(RELATION_OPERATOR, yytext, line + 1, col + 1); char_num +=
    yyleng; col += yyleng;}
```

因为少打了一个空格导致编译报错，这也充分体现了语法规整的重要性，在接触一门新的编程语言时首先必须将其语法规则了解清楚，否则就会犯这样难以发现的错误。

总的来说，在设计词法分析程序的过程中，提升了我对抽象问题具象化的能力，以及对大型问题分解的能力，通过实现各个子问题，以实现最终目的。同时设计过程中有许多边界以及错误情况需要考虑，这也提升了我思考问题的严谨性以及全面性。