词法分析实验报告（lex）

一、实验要求

题目：C语言词法分析程序的设计与实现

实验内容及要求：

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。
4. 检查源程序中存在的词法错误，并报告错误所在的位置。
5. 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。

实现方法要求：分别用以下两种方法实现：

方法1：采用C/C++作为实现语言，手工编写词法分析程序。（必做）

方法2：编写LEX源程序，利用LEX编译程序自动生成词法分析程序。

二、实验环境

词法生成工具：Flex 2.5.4，GNU Bison 1.28

三、实验原理

3.1 Lex

Lex 是一种生成扫描器的工具。扫描器是一种识别文本中的词汇模式的程序。 这些词汇模式（或者常规表达式）在一种特殊的句子结构中定义，这个我们一会儿就要讨论。

一种匹配的常规表达式可能会包含相关的动作。这一动作可能还包括返回一个标记。 当 Lex 接收到文件或文本形式的输入时，它试图将文本与常规表达式进行匹配。 它一次读入一个输入字符，直到找到一个匹配的模式。 如果能够找到一个匹配的模式，Lex 就执行相关的动作（可能包括返回一个标记）。 另一方面，如果没有可以匹配的常规表达式，将会停止进一步的处理，Lex 将显示一个错误消息。

Lex 和 C 是强耦合的。一个 *.lex* 文件（Lex 文件具有 *.lex* 的扩展名）通过 lex 公用程序来传递，并生成 C 的输出文件。这些文件被编译为词法分析器的可执行版本。

3.2 Lex 的常规表达式

字符	含义
A-Z, 0-9, a-z	构成了部分模式的字符和数字。
.	匹配任意字符，除了 \n。
-	用来指定范围。例如：A-Z 指从 A 到 Z 之间的所有字符。
[ ]	一个字符集合。匹配括号内的 *任意* 字符。如果第一个字符是 ^ 那么它表示否定模式。例如: [abC] 匹配 a, b, 和 C中的任何一个。
*****	匹配 *0个*或者多个上述的模式。
+	匹配 *1个*或者多个上述模式。
?	匹配 *0个或1个*上述模式。
$	作为模式的最后一个字符匹配一行的结尾。
{ }	指出一个模式可能出现的次数。 例如: A{1,3} 表示 A 可能出现1次或3次。
*****	用来转义元字符。同样用来覆盖字符在此表中定义的特殊意义，只取字符的本意。
^	否定。
|	表达式间的逻辑或。
"<一些符号>"	字符的字面含义。元字符具有。
/	向前匹配。如果在匹配的模版中的“/”后跟有后续表达式，只匹配模版中“/”前 面的部分。如：如果输入 A01，那么在模版 A0/1 中的 A0 是匹配的。
( )	将一系列常规表达式分组。

3.3 Lex编程

Lex 编程可以分为三步：

1. 以 Lex 可以理解的格式指定模式相关的动作。
2. 在这一文件上运行 Lex，生成扫描器的 C 代码。
3. 编译和链接 C 代码，生成可执行的扫描器。

一个 Lex 程序分为三个段：第一段是 C 和 Lex 的全局声明，第二段包括模式（C 代码），第三段是补充的 C 函数。 这些段以%%来分界。

3.3.1 全局声明举例

```
%{
    int tot_token = 0;
    int tot_error=0;
    int col = 0,line= 1;
%}
```

这里声明了四个整型变量`tot_token,tot_error,col,line`，分别代表符号总数，错误总数，列号，行号。

3.3.2 模式举例

```
letter                  [A-Za-z]
digit                   [0-9]
```

letter的模式为[A-Za-z]，即所有的大写或小写字母

digit的模式为[0-9]，即0~9之间任意数字

3.3.3 补充的 C 函数举例

```
void addcol(int cnt) {
    col += cnt;
}
```

我在这里定义了一个列数增加函数，作用是根据当前识别出的符号的长度增加当前列数

3.4 Lex的使用

编写好Lex的程序后，将其保存为.l后缀的文件。首先要使用flex生成lex语法对应的c文件：

```
C:\Users\HTY\iCloudDrive\iCloud~QReader~MarginStudy\编译原理与技术\实验>flex lex.l
```

生成的文件名为lex.yy.c

然后将这个c文件用GNU编译

```
C:\Users\HTY\iCloudDrive\iCloud~QReader~MarginStudy\编译原理与技术\实验>gcc lex.yy.c -o lex.exe
```

如果编译无误，将会生成lex.exe可执行文件

最后，输入测试程序文件名调用lex.exe即可

```
C:\Users\HTY\iCloudDrive\iCloud~QReader~MarginStudy\编译原理与技术\实验>lex.exe <test.c >out.txt
```

结果保存在out.txt中

四、程序设计

由于完成用C语言设计词法分析器后，留给我用来做Lex词法分析器的时间并不多。为简化实验步骤，我将C语言的符号类型概括为以下九类，且未考虑错误情况：

1、标识符 IDENTIFIER

2、关键字 KEYWORD

3、常量字符 CONSTANT_CHAR

4、常量字符串 CONSTANT_STRING

5、常整数 CONSTANT_INT

6、常实数 CONSTANT_REAL

7、操作符号 SIGNAL

8、行注释 LINE_COMMENT

9、换行符 LINE_BREAK

4.1 Lex模式设计

以上九种模式对应的正则表达式为：

1. 标识符 IDENTIFIER

   标识符的特征为有字母开头的、字母数字组成的字符串

   `{letter}({letter}|{digit})*`

2. 关键字 KEYWORD

   这里直接枚举了所有C语言关键字：

   `"char"|"int"|"long"|"float"|"double"|"void"|"unsigned"|"signed"|"const"|"static"|"extern"|"struct"|"union"|"typedef"|"sizeof"|"if"|"else"|"do"|"while"|"for"|"switch"|"case"|"default"|"continue"|"break"|"goto"`

3. 常量字符 CONSTANT_CHAR

   考虑普通字符与转义字符两种情况，普通字符为两个`'`之间包裹一个非`'`、非`\`、非`"`的字符，可以用`[^'"\]`表示这个集合；转义字符为`\`跟上一个可以被转义的字符

   `'[^'"\]'|'\\[0nrtvabf\'\"\\]'`

4. 常量字符串 CONSTANT_STRING

   `\"[^\"]*\"`

5. 常整数 CONSTANT_INT

   `{digit}+`

6. 常实数 CONSTANT_REAL

   `{digit}*(\.{digit}+)?(e|E[+\-]?{digit}+)?`

7. 操作符号 SIGNAL

   这里直接枚举了所有C语言的合法符号：`"="|"+"|"++"|"+="|"-"|"--"|"-="|"*="|"/"|"/="|"%"|"%="|"&"|"&="|"|"|"|="|"^"|"^="|"~"|"<<"|"<<="|">>"|">>="|"&&"|"&&="|"||"|"||="|"!"|"<"|"<="|"=="|"!="|">"|">="|"##"|"?"|","|":"|";"|"."|"*"|"->"|"("|")"|"["|"]"|"{"|"}"`

8. 行注释 LINE_COMMENT

   行注释的开头两个`/`符号需要被转义

   `(\/\/.*\n)`

9. 换行符 LINE_BREAK

   \n

4.2 辅助变量与函数设计

```
void addline()   {  //换行时行数加1
    line ++;
}
void clearcol()  {  //换行时列数置1
    col = 1;
}
void addcol(int cnt) {  //识别完一个符号后列数加上这个符号的长度
    col += cnt;
}
void addWord()   {  //识别完一个符号符号总数加1
    tot_token ++;
}
```

4.3 完整代码

```
%{
    int tot_token = 0;
    int charCounter=0;
    int col =1,line=1;
    void addline();
    void addcol(int);
    void clearcol();
    void addWord();
%}
letter                  [A-Za-z_]
digit                   [0-9]
LINE_COMMENT            (\/\/.*\n)
CONSTANT_CHAR           '[^']'|'\\[0nrtvabf\'\"\\]'
CONSTANT_STRING         \"[^'"\]*\"

IDENTIFIER              {letter}({letter}|{digit})*
KEYWORD
 "char"|"int"|"long"|"float"|"double"|"void"|"unsigned"|"signed"|"const"|"static
"|"extern"|"struct"|"union"|"typedef"|"sizeof"|"if"|"else"|"do"|"while"|"for"|"s
witch"|"case"|"default"|"continue"|"break"|"goto"
CONSTANT_INT            {digit}+
CONSTANT_REAL           {digit}*(\.{digit}+)?(e|E[+\-]?{digit}+)?
SIGNAL                  "="|"+"|"++"|"+="|"-"|"--"|"-
="|"*="|"/"|"/="|"%"|"%="|"&"|"&="|"|"|"|="|"^"|"^="|"~"|"<<"|"
<<="|">>"|">>="|"&&"|"&&="|"||"|"||="|"!"|"<"|"
<="|"=="|"!="|">"|">="|"##"|"?"|","|":"|";"|"."|"*"|"->"|"("|")"|"["|"]"|"{"|"}"
LINE_BREAK              \n

%%

{LINE_COMMENT} {
    printf ("< LINE_COMMENT , \"");
    int i = 0;
    while (i < yyleng - 1)
        printf ("%c", yytext[i++]);
    printf ("\" > in ( %d , %d)\n", line, col);
    addcol(yyleng);
    addWord();
}
```

```
{CONSTANT_CHAR}  {
    printf("< LINE_COMMENT , %s > in ( %d , %d)\n",yytext, line, col);
    addcol(yyleng);
    addWord();
}
{CONSTANT_STRING}  {
    printf("< LINE_STRING , %s > in ( %d , %d)\n",yytext, line, col);
    addcol(yyleng);
    addWord();
}
{KEYWORD} {
    printf("< KEYWORD , \"%s\" > in ( %d , %d)\n",yytext, line, col);
    addcol(yyleng);
    addWord();
}
{IDENTIFIER}    {
    printf("< IDENTIFIER , \"%s\" > in ( %d , %d)\n",yytext, line, col);
    addcol(yyleng);
    addWord();
}

{CONSTANT_INT}   {
    printf("< CONSTANT_INT , \"%s\" > in ( %d , %d)\n",yytext, line, col);
    addcol(yyleng);
    addWord();
}
{CONSTANT_REAL} {
    printf("< CONSTANT_REAL , \"%s\" > in ( %d , %d)\n",yytext, line, col);
    addcol(yyleng);
    addWord();
}
{SIGNAL}  {
    printf("< SIGNAL , \"%s\" > in ( %d , %d)\n",yytext, line, col);
    addcol(yyleng);
    addWord();
}

{LINE_BREAK} {
    addline();
    clearcol();
}

%%
int main(void)
{
    printf ("开始分析\n");
    yylex();
    printf("统计结果： 共 %d 行, 共 %d 个单词\n",line, tot_token);
    return 0;
}
int yywrap()    {
    return 1;
}

void addline()   {
    line ++;
}
void addcol(int cnt) {
```

```
    col += cnt;
}
void clearcol()  {
    col = 1;
}
void addWord()   {
    tot_token ++;
}
```

五、程序测试

使用测试程序 test.c

```
int main ()
{
    pid_t fpid;
    int count=0;
    fpid=fork();
    if (fpid < 0)
        printf("error in fork!"); //Block
    else if (fpid == 0) {
        printf("i am the child process, my process id is %d/n",getpid());
        count++;
    }
    else {
        printf("i am the parent process, my process id is %d/n",getpid());
        count++;
    }
    printf("aaa: %d/n",count);
    return 0;
}
```

运行结果

```
开始分析
< KEYWORD , "int" > in ( 1 , 1)
< IDENTIFIER , "main" > in ( 1 , 4)
< OPERATOR , "(" > in ( 1 , 8)
< OPERATOR , ")" > in ( 1 , 9)
< OPERATOR , "{" > in ( 2 , 1)
< IDENTIFIER , "pid_t" > in ( 3 , 1)
< IDENTIFIER , "fpid" > in ( 3 , 6)
< OPERATOR , ";" > in ( 3 , 10)
< KEYWORD , "int" > in ( 4 , 1)
< IDENTIFIER , "count" > in ( 4 , 4)
< OPERATOR , "=" > in ( 4 , 9)
< CONSTANT_INT , "0" > in ( 4 , 10)
< OPERATOR , ";" > in ( 4 , 11)
< IDENTIFIER , "fpid" > in ( 5 , 1)
< OPERATOR , "=" > in ( 5 , 5)
< IDENTIFIER , "fork" > in ( 5 , 6)
< OPERATOR , "(" > in ( 5 , 10)
< OPERATOR , ")" > in ( 5 , 11)
< OPERATOR , ";" > in ( 5 , 12)
< KEYWORD , "if" > in ( 6 , 1)
```

```
< OPERATOR , "(" > in ( 6 , 3)
< IDENTIFIER , "fpid" > in ( 6 , 4)
< OPERATOR , "<" > in ( 6 , 8)
< CONSTANT_INT , "0" > in ( 6 , 9)
< OPERATOR , ")" > in ( 6 , 10)
< IDENTIFIER , "printf" > in ( 7 , 1)
< OPERATOR , "(" > in ( 7 , 7)
< LINE_STRING , "error in fork!" > in ( 7 , 8)
< OPERATOR , ")" > in ( 7 , 24)
< OPERATOR , ";" > in ( 7 , 25)
< LINE_COMMENT , "//Block" > in ( 7 , 26)
< KEYWORD , "else" > in ( 7 , 34)
< KEYWORD , "if" > in ( 7 , 38)
< OPERATOR , "(" > in ( 7 , 40)
< IDENTIFIER , "fpid" > in ( 7 , 41)
< OPERATOR , "==" > in ( 7 , 45)
< CONSTANT_INT , "0" > in ( 7 , 47)
< OPERATOR , ")" > in ( 7 , 48)
< OPERATOR , "{" > in ( 7 , 49)
< IDENTIFIER , "printf" > in ( 8 , 1)
< OPERATOR , "(" > in ( 8 , 7)
< LINE_STRING , "i am the child process, my process id is %d/n" > in ( 8 , 8)
< OPERATOR , "," > in ( 8 , 55)
< IDENTIFIER , "getpid" > in ( 8 , 56)
< OPERATOR , "(" > in ( 8 , 62)
< OPERATOR , ")" > in ( 8 , 63)
< OPERATOR , ")" > in ( 8 , 64)
< OPERATOR , ";" > in ( 8 , 65)
< IDENTIFIER , "count" > in ( 9 , 1)
< OPERATOR , "++" > in ( 9 , 6)
< OPERATOR , ";" > in ( 9 , 8)
< OPERATOR , "}" > in ( 10 , 1)
< KEYWORD , "else" > in ( 11 , 1)
< OPERATOR , "{" > in ( 11 , 5)
< IDENTIFIER , "printf" > in ( 12 , 1)
< OPERATOR , "(" > in ( 12 , 7)
< LINE_STRING , "i am the parent process, my process id is %d/n" > in ( 12 , 8)
< OPERATOR , "," > in ( 12 , 56)
< IDENTIFIER , "getpid" > in ( 12 , 57)
< OPERATOR , "(" > in ( 12 , 63)
< OPERATOR , ")" > in ( 12 , 64)
< OPERATOR , ")" > in ( 12 , 65)
< OPERATOR , ";" > in ( 12 , 66)
< IDENTIFIER , "count" > in ( 13 , 1)
< OPERATOR , "++" > in ( 13 , 6)
< OPERATOR , ";" > in ( 13 , 8)
< OPERATOR , "}" > in ( 14 , 1)
< IDENTIFIER , "printf" > in ( 15 , 1)
< OPERATOR , "(" > in ( 15 , 7)
< LINE_STRING , "aaa: %d/n" > in ( 15 , 8)
< OPERATOR , "," > in ( 15 , 19)
< IDENTIFIER , "count" > in ( 15 , 20)
< OPERATOR , ")" > in ( 15 , 25)
< OPERATOR , ";" > in ( 15 , 26)
< IDENTIFIER , "return" > in ( 16 , 1)
< CONSTANT_INT , "0" > in ( 16 , 7)
< OPERATOR , ";" > in ( 16 , 8)
< OPERATOR , "}" > in ( 17 , 1)
```

```
统计结果： 共 17 行，共 78 个单词
```