C语言词法分析器实验报告

2019211315班 2019211243 谭恩浩

1 实验题目

C语言词法分析程序的设计与实现

2 实验内容

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。

3 实验要求

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；
- 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。

4 文法定义及状态转换图

4.1 语言说明

C语言定义有以下记号及单词：

1. 标识符：C语言中的标识符只能由字母、数字和下划线三种字符组成,且第一个字符必须是字母或下划线；

2. 关键字：标识符集合的子集。C语言共有33个关键字（不包含C11新增的关键字）：

   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；

3. 无符号数：由整数部分、可选的小数部分和可选的指数部分构成；
4. 运算符：
   a. 关系运算符6种：`<`, `<=`, `>`, `>=`, `==`, `!=`；
   b. 算术运算符7种：`+`, `-`, `*`, `/`, `%`, `++`, `--`；
   c. 逻辑运算符3种：`&&`, `||`, `!`；
   d. 位操作运算符6种：`&`, `|`, `~`, `^`, `<<`, `>>`；
   e. 赋值运算符11种：`=`, `+=`, `-=`, `*=`, `/=`, `%=`, `&=`, `|=`, `^=`, `>>=`, `<<=`；
   f. 选择运算符1种：`?:`；
   g. 逗号运算符1种：`,`；
   h. 指针运算符2种：`*`, `&`；
   i. 特殊运算符4种：`()`, `[]`, `->`, `.`；
   j. 条件运算符1种：`?:`；
5. 标点符号：`{`, `}`, `:`, `'`, `"`, `;`, `,`, `#`；
6. 注释标记：
   a. 以`/*`开始，以`*/`结束；
   b. 以`//`开始，以换行符结束；
7. 单词符号间的分隔符：空格。

4.2 记号的正规文法

1. 标识符的文法：

   $$id \rightarrow \text{letter}\ rid$$
   $$rid \rightarrow \epsilon \mid \text{letter}\ rid \mid \text{digit}\ rid$$

2. 无符号整数的文法：

$digits \rightarrow \text{digit}\ remainder$
$remainder \rightarrow \epsilon \mid \text{digit}\ remainder$

3. 无符号数的文法：

$num \rightarrow digit\ num1$
$num1 \rightarrow digit\ num1 \mid .\ num2 \mid \text{E}\ num4 \mid \epsilon$
$num2 \rightarrow digit\ num3$
$num4 \rightarrow +\ digits \mid -\ digits \mid \text{digit}\ num5$
$digits \rightarrow \text{digit}\ num5$
$num5 \rightarrow \text{digit}\ num5 \mid \epsilon$

4. 关系运算符的文法：

$relop \rightarrow < \mid < equal \mid = \mid > \mid > equal \mid !equal$
$equal \rightarrow =$

5. 算术运算符的文法：

$ariop \rightarrow + \mid - \mid * \mid / \mid = \mid + plus \mid - minus$
$plus \rightarrow +$
$minus \rightarrow -$

6. 逻辑运算符的文法：

$logop \rightarrow \&and \mid |or \mid !$
$and \rightarrow \&$
$or \rightarrow |$

7. 位操作运算符的文法：

$bitop \rightarrow \& \mid | \mid \sim \mid \hat{} \mid < left \mid > right$
$left \rightarrow <$
$right \rightarrow >$

8. 赋值运算符的文法：

$assop \rightarrow = \mid = equal \mid + equal \mid - equal \mid * equal \mid / equal \mid$
$\%equal \mid \&equal \mid |equal \mid \hat{}equal \mid > right \mid < left$
$right \rightarrow > equal$
$left \rightarrow < equal$
$equal \rightarrow =$

9. 条件运算符的文法：

$conop \rightarrow ?$

10. 逗号运算符的文法：

    $comop \rightarrow ,$

11. 指针运算符的文法：

    $ponop \rightarrow * \mid \&$

12. 特殊运算符的文法：

    $speop \rightarrow (\mid) \mid [\mid] \mid . \mid -right$
    $right \rightarrow >$

13. 标点符号的文法：

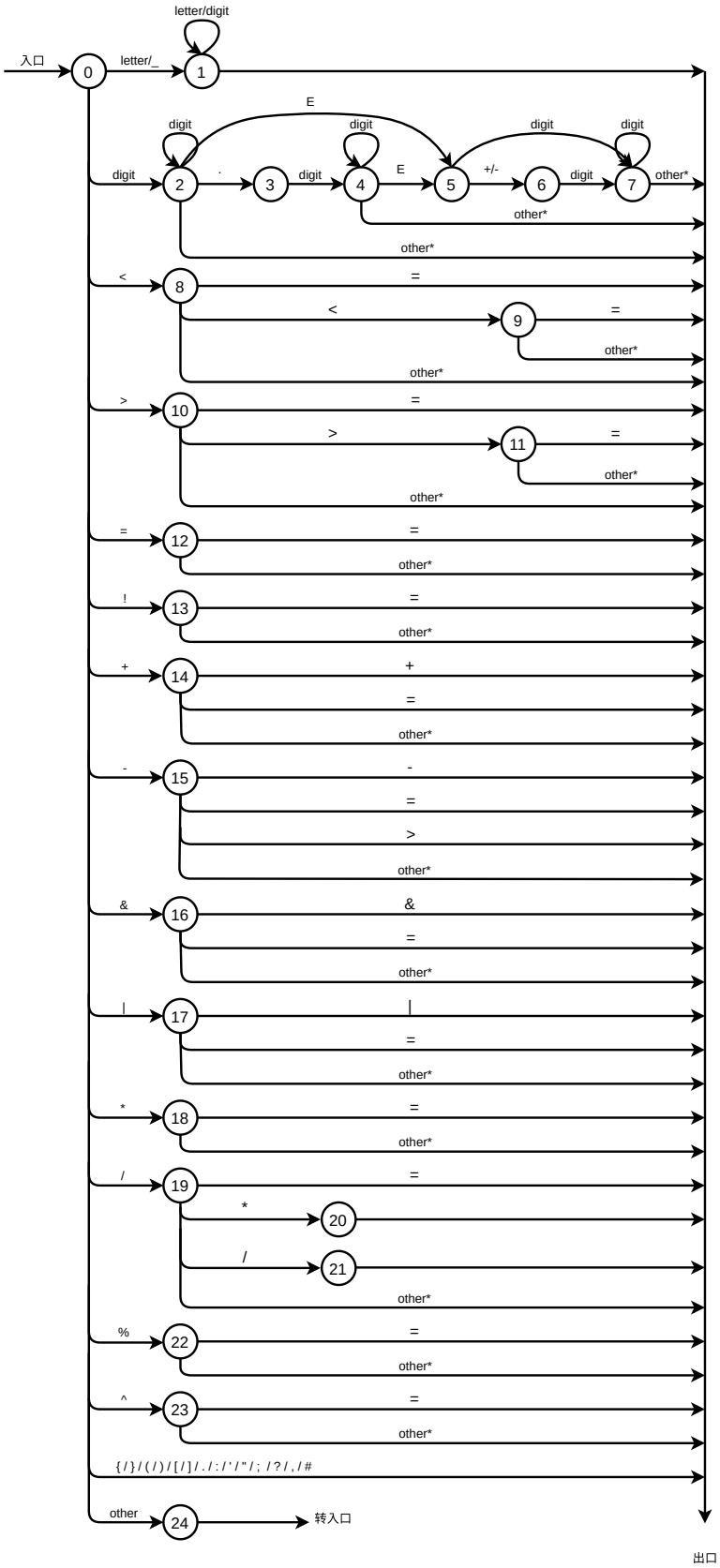
    $symbol \rightarrow \{ \mid \} \mid : \mid ' \mid " \mid ; \mid , \mid \#$

14. 注释头符号的文法：

    $note \rightarrow /star \mid /slash$
    $star \rightarrow *$
    $slash \rightarrow /$

4.3 状态转换图

为每种记号的文法构造出相应的状态转换图，让这些状态转换图共用一个初态，并对相似的文法进行一定的调整以简化转换图，可以得到词法分析程序的状态转换图如下图所示：


入口
letter/digit
letter/_
digit
E
.
+/-
other*
=
<
>
!
+
-
&
|
*
/
%
^
{ / } / (/) / [/] / . / : / ' / " / ; / ? / , / #
other
转入口
出口

5 词法分析程序的构造

按照上图所示状态转换图，每一个状态对应一段程序，用switch-case结构组合，就可以构造出相应的词法分析程序。

- 在开始状态，首先要读进一个字符。若读入的字符是一个空格或换行，就跳过它，继续读字符，直到读进一个非空字符为止。接下来的工作就是根据所读进的非空字符转相应的程序段进行处理。
- 在标识符状态，识别并组合出一个标识符之后，还必须加入一些动作，如查关键字表，以确定识别出的单词符号是关键字还是用户自定义标识符，并输出相应的记号。
- 在无符号数状态，可识别出各种常数，包括整数、小数和无符号数。在组合常数的同时，还要进行从字符串到在数字的转换。
- 在"<"(">")状态，若读进的下一个字符是"="，则输出关系运算符"<="(">=")；若读进的下一个字符是"<"(">")，则进入"<<"(">>")状态；否则输出关系运算符"<"(">")。
- 在"<<"(">>")状态，若读进的下一个字符是"="，则输出赋值运算符"<<="(">>=")；否则输出位操作运算符"<<"(">>")。
- 在"="("!")状态，若读进的下一个字符是"="，则输出赋值运算符"=="("!=")；否则输出赋值运算符"="(逻辑运算符"!")。
- 在"+"("&"/"|")状态，若读进的下一个字符是"="，则输出赋值运算符"+="("&="/"|=")；若读进的下一个字符是"+"("&"/"|")，则输出算术运算符"++"(逻辑运算符"&&"/"||")；否则输出算术运算符"+"(位操作运算符"&"/"|")。
- 在"-"状态，若读进的下一个字符是"="，则输出赋值运算符"-="；若读进的下一个字符是"-"，则输出算术运算符"--"；若读进的下一个字符是">"，则输出指针运算符"->"；否则输出算术运算符"-"。
- 在"*"("%"/"^")状态，若读进的下一个字符是"="，则输出赋值运算符"*="("%="/"^=")；否则输出算术运算符"*"("%"/位操作运算符"^")。
- 在"/"状态，若读进的下一个字符是"="，则输出赋值运算符"/="；若读进的下一个字符是"*"，则进入"/*"状态；若读进的下一个字符是"/"，则进入"//"状态；否则输出算术运算符"/"。
- 在其他状态，直接输出相应的记号即可。
- 若进入错误处理状态，表示词法分析程序从源程序中读入了一个不合法的字符。所谓不合法的字符是指该语言不包括以此字符开头的单词符号。词法分析程序发现不合法

的字符时，要做错误处理，其主要工作是显示或打印错误信息，并跳过这个字符，然后转开始状态继续识别和分析下一个单词符号。

6 词法分析程序的实现

6.1 输出形式

该词法分析程序使用的翻译表如下所示：

C语言词法分析程序的翻译表

Aa 正规表达式	记号	属性
auto	auto	0
break	break	1
case	case	2
char	char	3
const	const	4
continue	continue	5
default	default	6
do	do	7
double	double	8
else	else	9
enum	enum	10
extern	extern	11
float	float	12
for	for	13
goto	goto	14
if	if	15
inline	inline	16
int	int	17
long	long	18
register	register	19

正规表达式	记号	属性
restrict	restrict	20
return	return	21
short	short	22
signed	signed	23
sizeof	sizeof	24
static	static	25
struct	struct	26
switch	switch	27
typedef	typedef	28
union	union	29
unsigned	unsigned	30
void	void	31
volatile	volatile	32
numi	NUMI	整形常数值
numf	NUMF	浮点常数值
id	ID	单词在符号表的下标
<	REL	0
<=	REL	1
>	REL	2
>=	REL	3
==	REL	4
!=	REL	5
=	ASS	0
+=	ASS	1
-=	ASS	2
*=	ASS	3
/=	ASS	4
%=	ASS	5
&=	ASS	6

正规表达式	记号	属性
|=	ASS	7
^=	ASS	8
>>=	ASS	9
<<=	ASS	10
&	BIT	0
|	BIT	1
~	BIT	2
^	BIT	3
<<	BIT	4
>>	BIT	5
&&	LOG	0
||	LOG	1
!	LOG	2
+	ARI	0
-	ARI	1
*	ARI	2
/	ARI	3
%	ARI	4
++	ARI	5
--	ARI	6
(	SPE	0
)	SPE	1
[	SPE	2
]	SPE	3
->	SPE	4
.	SPE	5
/*	NOT	0
//	NOT	1
{	SYM	0

Aa 正规表达式	记号	属性
}	SYM	1
:	SYM	2
'	SYM	3
"	SYM	4
;	SYM	5
,	SYM	6
#	SYM	7
?	SEL	0

6.2 定义全局变量和过程

此例中的词法分析程序所用的全局变量、函数及数据结构的说明如下图所示

```
#define KEYSIZE 33 //关键字的数量
#define BUFFERSIZE 1000 //半个缓冲区的大小

//13个记号
#define KEY 0
#define ID 1
#define NUMI 2
#define NUMF 3
#define REL 4
#define ASS 5
#define BIT 6
#define LOG 7
#define ARI 8
#define SPE 9
#define NOT 10
#define SYM 11
#define SEL 12

int token_num[13]; //存放各种记号单词的数量

union attributes {
        long intnum;
        double fltnum;
}; //存放二元组中的属性

struct tokens {
    int token;
    union attributes attribute;
};
struct tokens binary; //存放当前单词二元组
```

```
char signTable[1000][32]; //存放用户自定义字符的符号表
int signPos; //符号表当前末尾位置
char keyTable[KEYSIZE][10] = {"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"}; //存放关键字字符
int state; //当前状态指示
char C; //存放当前读入的字符
int isKey; //值为-1表示识别出的单词是用户自定义的标识符，否则，表示识别出的单词是关键字，其值为
           //关键字的记号
char token[1000]; //存放当前正在识别的单词字符串
int pos; //token中的字符串尾部；
int forward; //向前指针
int line; //当前行数
int total; //不带换行和空格的字符总数
int total_ns; //带换行和空格的字符总数
char buffer[BUFFERSIZE * 2]; //输入缓冲区
FILE* fp; //文件指针
int re_flag; //向前指针回退时的标记，避免重复加载缓冲区

void get_char(); //根据向前指针forward的指示从输入缓冲区中读一个字符放入变量C中，然后移动
                 //forward，使之指向下一个字符
void get_nbc(void); //检查C中的字符是否为空格，若是，则反复调用get_char，直到C中进入一个非空
                    //字符为止
void cat(void); //把C中字符链接在token中的字符串后面
void retract(void); //向前指针forward后退一个字符
void combine(int token, double attribute); //把单词的记号和属性组合成一个二元组
void error(int log); //对发现的错误进行相应的处理
int letter(void); //判断C中的字符是否为字母，若是则返回1，否则返回0
int digit(void); //判断C中的字符是否为数字，若是则返回1，否则返回0
int reserve(void); //根据token中的单词查关键字表，若token中的单词是关键字，则返回该关键字的
                   //记号，否则，返回值“-1”
void table_insert(void); //将识别出来的用户自定义标识符，即token中的单词，插入符号表，返回该
                         //单词在符号表中的位置指针；
void token_print(void); //打印识别出来的记号
void buffer_fill(int start); //填充一半的缓冲区
void outcome_print(void); //打印统计结果
```

6.3 编制词法分析程序

此处简要说明一下程序中的主要内容，完整程序见附件*源代码*。

该函数在从缓冲区获取下一个字符的同时，前移向前指针，统计程序的总行数，并在向前指针到达缓冲区边界时更新缓冲区。

```
void get_char() {
    C = buffer[forward];
    if (C == '\n') {
        line++;
    }
    forward++;
    if (forward == BUFFERSIZE && re_flag == 0) {
            buffer_fill(BUFFERSIZE);
    }
    else if (forward == BUFFERSIZE * 2 && re_flag == 0) {
            buffer_fill(0);
    }
    re_flag = 0;
    forward = (forward + BUFFERSIZE * 2) % (BUFFERSIZE * 2);
}
```

该函数向后回退一个向前指针，并将回退标记置1，防止重复更新缓冲区；同时若超前扫描时导致行数计数器增加，在这里一起回退。

```
void retract(void) {
    re_flag = 1;
    forward = (forward + BUFFERSIZE * 2 - 1) % (BUFFERSIZE * 2);
    if (C == '\n') {
        line--;
    }
}
```

该函数将每个单词的记号和属性组合成一个二元组，并清空token，为读新的单词作准备。

```
void combine(int tokens, double attribute) {
    token[pos] = '\0';
    binary.token = tokens;
    if (binary.token == NUMF) {
        binary.attribute.fltnum = attribute;
    }
    binary.attribute.intnum = (int)attribute;
    token_print();
    pos = 0;
}
```

该函数是错误处理函数，针对词法分析时可能出现的各种错误给出相应的报告及错误所在的行数，由于行数的增加是在get_char()函数中，若导致错误的字符恰好是换行符，line

中储存的行数就会不正确，这里用局部变量eline(error line)解决这个问题。

```
void error(int log) {
    int eline;
    if (C == '\n') {
        eline = line - 1;
    }
    else {
        eline = line;
    }
    switch (log) {
        case 0:
           printf("\t小数点后必须有数字(line %d)\n", eline);
           break;
        case 1:
            printf("\t指数符号E后必须有数字(line %d)\n", eline);
            break;
        case 2:
            printf("\t正负号后必须有数字(line %d)\n", eline);
            break;
        case 3:
            printf("\t无法识别的输入(line %d)\n", eline);
    }
    pos = 0;
}
```

该函数更新缓冲区，并在文件结束时即时停止读入，同时在读入字符的同时统计程序总字符数。为方便之后的判断，将文件结束符设置为'\0'。

```
void buffer_fill(int start) {
    int i = 0;
    char c;
    while (i < BUFFERSIZE && (c = fgetc(fp)) != EOF) {
        buffer[start + i] = c;
        i++;
        total_ns++;
        if (c != ' ' && c != '\n') {
            total++;
        }
    }
    if (c == EOF) {
        buffer[start + i] = '\0';
    }
}
```

词法分析器的主体函数，程序逻辑比较简单，就是用switch-case实现上述状态转换图，每个状态就是一个case，状态的转换就是case的跳转。

```
void lexicalAnalyzer(void) {
    state = 0;
    pos = 0;
    do {
        switch(state) {
            case 0: //初始状态
                get_char();
                get_nbc();
                if (letter() || C == '_') {
                    state = 1; //设置标识符状态
                }
                else if (digit()) {
                    state = 2; //设置常数符状态
                }
                else {
                    switch(C) {
                        case '<': state = 8; break;
                        case '>': state = 10; break;
                        case '=': state = 12; break;
                        case '!': state = 13; break;
                        case '+': state = 14; break;
                        case '-': state = 15; break;
                        case '&': state = 16; break;
                        case '|': state = 17; break;
                        case '*': state = 18; break;
                        case '/': state = 19; break;
                        case '%': state = 22; break;
                        case '^': state = 23; break;
                        case '{':
                            cat(); state = 0; combine(SYM, 0); break;
                        case '}':
                            cat(); state = 0; combine(SYM, 0); break;
                        case '(':
                            cat(); state = 0; combine(SPE, 0); break;
                        case ')':
                            cat(); state = 0; combine(SPE, 1); break;
                        case '[':
                            cat(); state = 0; combine(SPE, 2); break;
                        case ']':
                            cat(); state = 0; combine(SPE, 3); break;
                        case '.':
                            cat(); state = 0; combine(SPE, 5); break;
                        case ':':
                            cat(); state = 0; combine(SYM, 2); break;
                        case '\'':
                            cat(); state = 0; combine(SYM, 3); break;
                        case '\"':
                            cat(); state = 0; combine(SYM, 4); break;
                        case ';':
                            cat(); state = 0; combine(SYM, 5); break;
                        case '?':
                            cat(); state = 0; combine(SEL, 0); break;
```

```
                    case ',':
                        cat(); state = 0; combine(SYM, 6); break;
                    case '#':
                        cat(); state = 0; combine(SYM, 7); break;
                    default: state = 22; break;
                }
            }
            break;

        case 1: //标识符状态
            cat();
            get_char();
            if (letter() || digit()) {
                state = 1;
            }
            else {
                retract();
                state = 0;
                isKey = reserve();
                if (isKey != -1) {
                    combine(KEY, isKey);
                }
                else {
                    table_insert(); combine(ID, signPos);
                }
            }
            break;

        case 2: //常数状态
            cat();
            get_char();
            switch (C) {
                case '0':
                case '1':
                case '2':
                case '3':
                case '4':
                case '5':
                case '6':
                case '7':
                case '8':
                case '9': state = 2; break;
                case '.': state = 3; break;
                case 'E': state = 5; break;
                default: //识别出常整数
                    retract(); state = 0; combine(NUMI, atoi(token)); break;
            }
            break;

        case 3: //小数点状态
            cat();
            get_char();
            if (digit()) {
                state = 4;
```

```
                }
                else {
                    error(0); state = 0;
                }
                break;

            case 4: //小数状态
                cat();
                get_char();
                switch (C) {
                    case '0':
                    case '1':
                    case '2':
                    case '3':
                    case '4':
                    case '5':
                    case '6':
                    case '7':
                    case '8':
                    case '9': state = 4; break;
                    case 'E': state = 5; break;
                    default: //识别出小数
                        retract(); state = 0; combine(NUMF, atof(token)); break;
                }
                break;

            case 5: //指数状态
                cat();
                get_char();
                switch (C) {
                    case '0':
                    case '1':
                    case '2':
                    case '3':
                    case '4':
                    case '5':
                    case '6':
                    case '7':
                    case '8':
                    case '9': state = 7; break;
                    case '+':
                    case '-':state = 6; break;
                    default:
                        retract(); error(1); state = 0; break;
                }
                break;

            case 6:
                cat();
                get_char();
                if (digit()) {
                    state = 7;
                }
                else {
```

```
                retract(); error(2); state = 0;
            }
            break;

        case 7:
            cat();
            get_char();
            if(digit()) {
                state = 7;
            }
            else {
                retract(); state = 0; combine(NUMF, atof(token));
            }
            break;

        case 8: // <
            cat();
            get_char();
            switch (C) {
                case '=':
                    cat(); state = 0; combine(REL, 1); break; // <=
                case '<':
                    state = 9; break; // <<?
                default:
                    retract(); state = 0; combine(REL, 0); break; // <
            }
            break;

        case 9: // <<
            cat();
            get_char();
            if (C == '=') {
                cat(); state = 0; combine(ASS, 10); // <<=
            }
            else {
                retract(); state = 0; combine(BIT, 4); // <<
            }
            break;
                        .
                        .
                        .

        case 23: // ^
            cat();
            get_char();
            if (C == '=') {
                cat(); state = 0; combine(ASS, 8); // ^=
            }
            else {
                retract(); state = 0; combine(BIT, 3); // ^
            }
            break;

        case 24: //error
```

```
                error(3);
                state = 0;
                break;
            default: break;
        }
    } while (C != '\0');
}
```

7 词法分析程序测试

本报告指展示一个简单的HelloWorld程序（附带一个人为设置的小错误）的测试结果，其他复杂的程序测试结果见附件*测试报告*。

本词法分析程序以txt文件形式读入测试程序。

```
//
// Created by Tan Enhao on 2021/10/10.
//

#include <stdio.h>

int main(void) {
    printf("Hello World!");
    int a = 12.23E;
    return 0;
}
```

```
词法分析结果为：
  <//, NOT, 1>
  <//, NOT, 1>
  <//, NOT, 1>
  <#, SYM, 7>
  <include, ID, 0>
  <<, REL, 0>
  <stdio, ID, 0>
  <., SPE, 5>
  <h, ID, 0>
  <>, REL, 2>
  <int, KEY, 17>
  <main, ID, 0>
  <(, SPE, 0>
  <void, KEY, 31>
  <), SPE, 1>
  <{, SYM, 0>
  <printf, ID, 0>
  <(, SPE, 0>
```

```
  <", SYM, 4>
  <Hello, ID, 0>
  <World, ID, 0>
  <!, LOG, 2>
  <", SYM, 4>
  <), SPE, 1>
  <;, SYM, 5>
  <int, KEY, 17>
  <a, ID, 0>
  <=, ASS, 0>
  指数符号E后必须有数字(line 9)
  <;, SYM, 5>
  <return, KEY, 21>
  <0, NUMI, 0>
  <;, SYM, 5>
  <}, SYM, 0>


该程序共有 11 行
该程序的字符总数为 110 / 146
各种记号的的个数为：
  KEY:  4
  ID:   8
  NUMI: 1
  NUMF: 0
  REL:  2
  ASS:  1
  BIT:  0
  LOG:  1
  ARI:  0
  SPE:  5
  NOT:  3
  SYM:  8
  SEL:  0
```

8 需要注意的问题及解决方法

1. 由于向前指针`forward`有回退的需求（超前扫描时），且缓冲区的更新是以向前指针`forward`是否到达边界为条件，当向前指针`forward`恰好在边界处回退时，会导致缓冲区在同一个位置更新两次，覆盖第一次读进的数据。

本程序设置了一个回退标记`re_flag`，当向前指针`forward`回退时将其置为1，否则为0。当需要更新缓冲区时要检查该标记的值，只有当回退标记`re_flag`为0时才允许更新。

2. 统计行数时仅以检测到的换行符`\n`为指标，同样因为向前指针`forward`的回退问题会导致计数器`line`对同一个换行符`\n`重复计数。

   🤗 只要在向前指针`forward`回退时检测一下，如果当前字符为换行符`\n`，行数计数器`line`同时回退一个即可

3. 由于属性值有时需要是浮点数（记号为浮点数），有时需要是整数（其他情况），若设计成三元组会浪费空间。

   🤗 本程序使用了集合`union`将一个整形变量和浮点型变量集合到一起，解决了空间浪费的问题