19335030-陈至雪-实验1-手动

实验题目

词法分析扫描器的设计

实验要求

- 在报告中明确词法分析器所支持的单词范围
- 有自动机的设计
- 有实验结果说明

实验过程

此处实现的是 C 语言的词法分析器。

（1）首先定义标识符、保留字、常数、运算符、界符和需略除的符号的范围。

标识符	**标识符可以是字母、数字、下划线(A～Z, a～z, 0~9, _)组成的字符串，并且第一个字符必须是字母或下划线。**
保留字	指在高级语言中已经定义过的字，使用者不能再将这些字作为变量名或过程名使用。
常数	数字0~9
运算符	包括算术运算符、关系运算符、逻辑运算符、赋值运算符、位运算符、其他运算符
界符	限制界限所用，有单字节界符和双字节界符
需略除的符号	空格，制表符，换行符

标识符：

1. 标识符是严格**区分大小写**的。
2. 标识符**不能是C语言的关键字和保留标识符。**
3. 标识符长度限制跟C语言标准和编译器环境有关。

保留字：

字	**int,long,short,float,double,char,unsigned,signed,const,void,volatile,enum,struct,union**
语句定义保留字	if,else,goto,switch,case,do,while,for,continue,break,return,default,typedef
存储类说明保留字	auto,register,extern,static
长度运算符保留字	sizeof

运算符：

算术运算符	**+，-，*，/,%,++,--**
关系运算符	==，!=, >, <, >=, <=
逻辑运算符	&&, ||, !
赋值运算符	=, +=, -=, *=, /=, %=, <<=, >>=, &=, ^=, |=
位运算符	&, |, ^, ~, <<, >>
其他运算符	Condition ? X:Y, . , ->, &, *, ,

界符：

单字节界符	**；**
双字节界符	{},(),[], “和”，‘和’

需略除的符号：

注释符	**\ \ ，/* 和 */，**
其他	空格，制表符，换行符

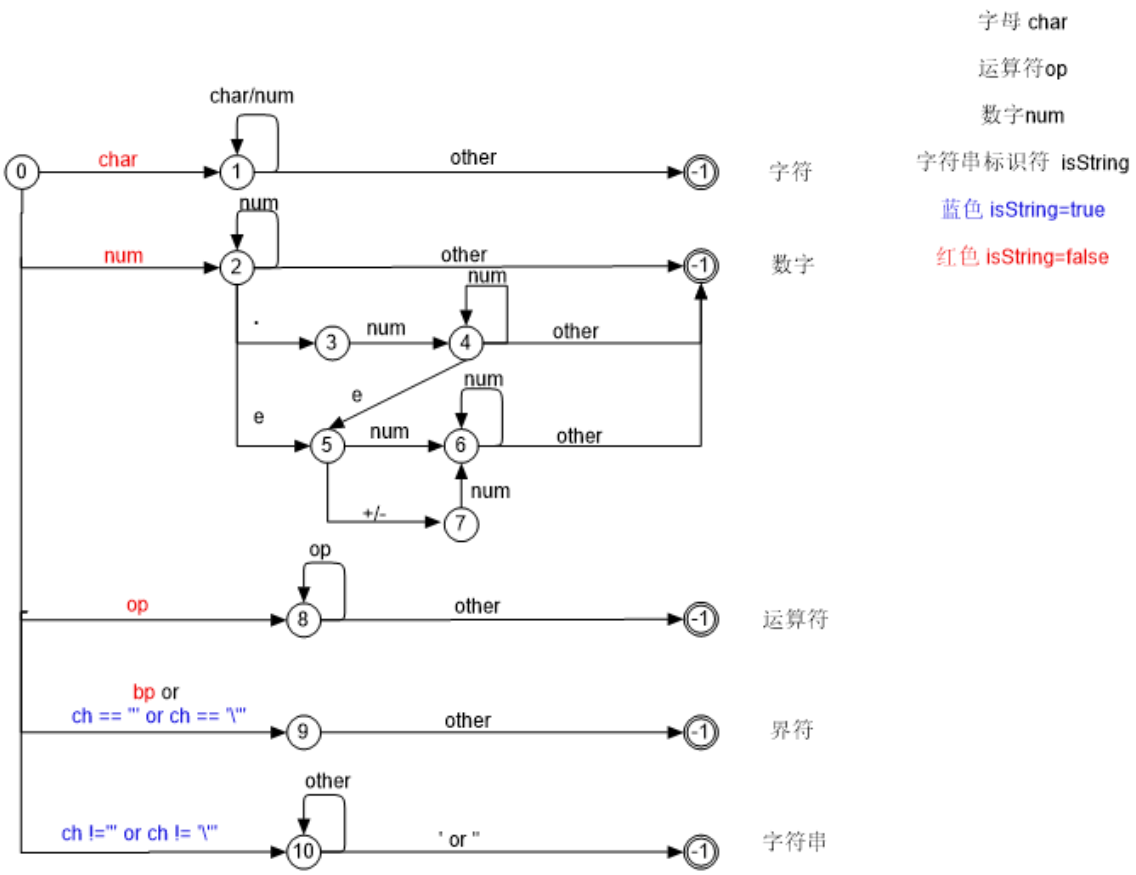
（2）进行种别编码：

单词符号	种别编码	助记符	内码值
常数	0	num	num在常数表中的位置
标识符	1	id	id在符号表中的位置
字符串	2	s	s在字符串表中的位置
int	3	int	
long	4	long	
short	5	short	
float	6	float	
double	7	double	
char	8	char	
unsigned	9	unsigned	
signed	10	signed	
const	11	const	
void	12	void	
volatile	13	volatile	
enum	14	enum	
struct	15	struct	
union	16	union	
if	17	if	-
else	18	else	-
goto	19	goto	-
switch	20	switch	-
case	21	case	-
do	22	do	-
while	23	while	-
for	24	for	-
continue	25	continue	-
break	26	break	-
return	27	return	-
default	28	default	-
typedef	29	typedef	-

单词符号	种别编码	助记符	内码值
auto	30	auto	-
register	31	register	-
extern	32	extern	-
static	33	static	-
sizeof	34	sizeof	-
+	35	+	-
-	36	-	-
*	37	*	-
/	38	/	-
%	39	%	-
++	40	++	-
--	41	--	-
==	42	==	-
!=	43	!=	-
>	44	>	-
<	45	<	-
>=	46	>=	-
<=	47	<=	-
&&	48	&&	-
||	49	||	-
!	50	!	-
=	51	=	-
+=	52	+=	-
-=	53	-=	-
*=	54	*=	-
/=	55	/=	-
%=	56	%=	-
<<=	57	<<=	-
>>=	58	>>=	-
&=	59	&=	-

单词符号	种别编码	助记符	内码值
^=	60	^=	-
|=	61	|=	-
&	62	&	-
|	63	|	-
^	64	^	-
~	65	-	-
<<	66	<<	-
>>	67	>>	-
;	68	;	-
{	69	{	-
}	70	}	-
(	71	(	-
)	72	)	-
[	73	[	-
]	74	]	-
"	75	"	-
'	76	'	-
,	77	,	-
#	78	#	-
.	79	.	-

（3）建立自动机



（4）代码实现

4.1 用数组存储关键字、运算符、界符等常量信息。

```
const char *keyword[] = //定义关键字
{"int","long","short","float","double","char","unsigned",
"signed","const","void","volatile","enum","struct","union",
"if", "else", "goto", "switch", "case", "do", "while",
"for", "continue", "break", "return", "default",
"typedef", "auto", "register", "extern", "static", "sizeof"};
int keyword_length = sizeof(keyword) / sizeof(char*);
const char *Opera[] = //定义运算符
{ "+", "-", "*", "/", "%", "++", "--", "==", "!=", ">",
"<", ">=", "<=", "&&", "||", "!", "=", "+=", "-=", "*=",
"/=", "%=", "<<=", ">>=", "&=", "^=", "|=", "&", "|", "^",
"~", "<<", ">>"};
const char *BS[] = {";","{","}","(",")","[","]","\"", "'",",","#","."};  //
定义界符
```

4.2 函数和变量说明

```
fstream inFile; //需要编译的文件
int BS_length = sizeof( BS ) / sizeof(char*);
int operator_length = sizeof(Opera) / sizeof(char*);
char id_table[255][255];//标识符表，存放构成的标识符
int ptr_id = 0; //标识符表的下标
char num_table[255][255];//常数表，存放构成的常熟
int ptr_num = 0; //常数表的下标
char string_table[255][255];
int ptr_string = 0;
```

```
char ch;//字符变量，存放最新读入的源程序字符
int ptr_token = 0; //token数组的下标
char token[255] = ""; //存放当前构成单词符号的字符串
bool iserror = false; // 发生错误的标志
bool isdigit(); //判断当前读取的字符是否是数字
bool isletter(); //判断当前读取的字符是否是字母
bool isoperator(); //判断当前字符是否是运算符
bool isbp();//判断当前字符是否是界符
void getbe(); //过滤无用符号
void concatenation(); //将ch中的字符链接到token后作为新的字符串
void error(int num); // 发现错误，显示错误
int reserve(); //判断token是否是keyword，若是则返回它的种别编码，若不是返回-1
int opra(); //判断token是否是操作符，若是，则返回它的种别编码，若不是返回-1
int bp(); //判断token是否是界符，若是，则返回它的种别编码，若不是返回-1
void buildtable();//建表（标识符表或常数表）
void automata(); //自动机
void LexicalAnalyzer(char*);//整个词法分析器
void Getchar();
int isString = 0;//用于标记token是字符串还是标识符，根据前一个字符是否是'或"决定，若是字符串isString=1,否则为0
char pre; //保存上一个字符
bool useNext = false;
char nextCh;
```

4.3 函数实现

- `bool isdigit()`

```
bool isdigit(){
    if( ch >= '0' && ch <= '9' )
        return true;
    else
        return false;
}
```

此函数用于判断当前字符是否是数字 0 ~ 9。

- `bool isletter()`

```
bool isletter(){
    if( (ch >= 'a' && ch <= 'z') || (ch >='A' && ch <= 'Z') )
        return true;
    else
        return false;
}
```

此函数用于判断当前字符是否是字母。

- `bool isoperator()`

```
bool isoperator(){
    if( ch == '+' || ch == '-' || ch == '*' || ch == '/' || ch == '%' ||
    ch == '=' || ch == '!' || ch == '>' || ch == '<' || ch == '&' ||
    ch == '|' || ch == '^' || ch == '~')
        return true;
    return false;
}
```

此函数用于判断当前字符是否是操作符，直接枚举判断。

- `bool isbp()`

```
bool isbp(){
    if( ch == ';' || ch == '{' || ch == '}' ||ch == '(' || ch == ')' || ch == '['
    || ch ==']' || ch == '"' || ch == '\'' || ch == ',' || ch == '#'|| ch == '.')
        return true;
    return false;
}
```

此函数用于判断当前字符是否是界符，直接枚举判断。

- `void getbe()`

```
void getbe(){
    while(ch == ' ' || ch == '\t' || ch == '\n')
        Getchar();
}
```

此函数用于过滤无用字符。其中定义无用字符为空格，制表符和换行符。

- `void error(int num)`

```
void error(int num){
    if( num == 0 )
        cout << "The token is too long\n" ;
    if( num == 1 )
        cout << "Illegal character: " << ch << endl;
}
```

此函数用于函数执行到错误处时报错，并输出错误类型的日志信息。传进的参数 `num` 为 error 的编码。这里定义了两种错误，第一种是 token 的长度超出最大限制，这里限制 token 最长为 254 个字符；第二种是接收到了非法字符。

- `void concatenation()`

```
void concatenation(){
    if (ptr_token > 254){
        error(0);
        return;
    }
    token[ptr_token] = ch;
    ptr_token ++;
    token[ptr_token] = '\0';
}
```

此函数将当前字符加入 `token`，如果 `token` 的长度大于 254，则返回错误。否则将当前字符加入 `token` 尾端。

- `int reserve()`

```
int reserve(){
    int i = 0;
    while( i < keyword_length ){
        if( !strcmp( keyword[i], token) )
            return i + 3; //返回种别编码
        i++;
    }
    return -1; //不是关键字，返回-1
}
```

此函数用于判断 `token` 是否是关键字，若是则返回它的种别编码，若不是返回-1。判断方法是将 `token` 与关键字数组中的每个元素进行对比，判断它们是否相等，如果相等说明 `token` 是关键字。

- `int opra()`

```
int opra(){
    int i = 0;
    while( i < operator_length ){
        if( !strcmp(Opera[i], token) )
            return i + 35;
        i++;
    }
    if( ptr_token <= 4 && ptr_token > 2) // 有可能是两个操作符连在一起，判断不出，将其拆分后重新进行判断。
    {
        if(  pre == '+' || pre == '-' || pre == '*' || pre == '/' || pre == '%' ||
    pre == '=' || pre == '!' || pre == '>' || pre == '<' || pre == '&' ||
    pre == '|' || pre == '^' || pre == '~' ){
            ch = pre; //开始回滚
            pre = token[ptr_token - 2];
            token[ptr_token - 1] = '\0';
            useNext = true; // 回滚，char下次更新时使用next的值，而不直接读取文件
            i = 0;
            while( i < operator_length ){
                if( !strcmp(Opera[i], token) ){
                    return i + 35;
                }
                i++;
            }
        }
```

```
    }
    return -1;
}
```

此函数用于判断 `token` 是否是操作符，若是则返回它的种别编码，若不是返回-1。判断方法是将 `token` 与操作符数组中的每个元素进行对比，判断它们是否相等，如果相等说明 `token` 是操作符。返回的编码是 i + 35，原因是第一个操作符在上述表格中的种别编码为 35。另外，根据上述自动机的逻辑，当出现两个操作符连在一起，又不是合法操作符时，可能无法正确判别，此时需要进行回滚操作。

- `int bp()`

```
int bp(){
    int i = 0;
    while( i < BS_length ){
        if( !strcmp(BS[i], token) )
            return i + 68;
        i++;
    }
    return -1;
}
```

该函数用于判断 `token` 是否是界符，若是，则返回它的种别编码，若不是返回-1。返回的编码是 i + 68，原因是第一个界符在上述表格中的种别编码为 68。

- `void Getchar()`

```
void Getchar(){
    if (useNext){
        ch = nextCh;
        useNext = false;
    }
    else{
        inFile >> ch;
        nextCh = ch;
    }
}
```

该函数用于读取字符。用一个布尔变量 `useNext` 判断是否要回滚，如果 `useNext` 为真，则不读取文件，而是用 `nextCh` 的值。`nextCh` 的值保存的是最近一次读取文件的字符。

- `void buildtable()`

```
void buildtable(){
    if(isString != 0){
        int k = 0;
        while( k < ptr_string){
            if( !strcmp( string_table[k], token ) ) {
                break;
            }
            k++;
        }
        if( k == ptr_string ){
            strcpy(string_table[k], token);
            ptr_string ++;
        }
        //isString = 0;
```

```
    }
    else{
        char head = token[0]-'0';
        if(!(head >= 0 && head <= 9)){ //token是标识符
            int i = 0;
            while( i < ptr_id ){
                if( !strcmp(id_table[i], token) ){ //两个字符串相等
                    break;
                }
                i++;
            }
            if( i == ptr_id ){
                strcpy(id_table[ptr_id], token);
                ptr_id++;
            }
        }

        else{ //token是常数
            int j = 0;
            while( j < ptr_num ){
                if( !strcmp(num_table[j], token) ){ //两个字符串相等
                    break;
                }
                j ++;
            }
            if( j == ptr_num ){
                strcpy(num_table[ptr_num], token);
                ptr_num ++;
            }
        }
    }
}
```

该函数用于建表（标识符表或常数表），为被判定为标识符或常数的 `token` 建表，保存出现过的标识符和常数。如果当前 `token` 被判断为标识符（常数），则在标识符表（常数表）逐个判断 `token` 是否已经存在于标识符表（常数表），若存在，则结束。否则，在标识符表（常数表）中保存该标识符（常数）。

- `void automata()`

```
void automata(){
    int state = 0;
    while(state != -1){ //规定-1为退出状态
        if(state == 0){
            if(ch == '"' || ch == '\'' )//不能直接isString == 1就跳到10
                state = 9;
            else if( isString == 1 ){
                state = 10;
            }
            else{
                getbe();
                if(isletter())
                    state = 1;
                else if( isdigit() )
                    state = 2;
                else if( isoperator() )
                    state = 8;
```

```
                else if( isbp() )
                    state = 9;
                else{
                    error(1);
                    cout << "191" << endl;
                    iserror = true;
                }
            }
        }

        else if( state == 1 ){
            if( isletter() || isdigit() )
                state = 1;
            else
                state = -1;
        }
        else if( state == 2 ){
            if( isdigit() )
                state = 2;
            else if( ch == '.' )
                state = 3;
            else if( ch == 'e' )
                state = 5;
            else
                state = -1;
        }
        else if( state == 3 ){
            if( isdigit() )
                state = 4;
            else{
                error(1);
                cout << "217" << endl;
                iserror = true;
            }
        }
        else if( state == 4 ){
            if( isdigit() )
                state = 4;
            else if( ch == 'e' )
                state = 5;
            else
                state = -1;
        }
        else if( state == 5 ){
            if( isdigit() ){
                state = 6;
            }
            else if( ch == '+' || ch == '-' ){
                state = 7;
            }
            else{
                error(1);
                cout << "238" << endl;
                iserror = true;
            }
        }
        else if( state == 6 ){
            if( isdigit() ){
```

```
                state = 6;
            }
            else
                state = -1;
        }
        else if( state == 7){
            if( isdigit() )
                state = 6;
            else {
                error(1);
                cout << "254" << endl;
                iserror = true;
            }
        }
        else if( state == 8){
            if( isoperator() )
                state = 8;
            else
                state = -1;
        }
        else if( state == 9 ){
            state = -1;
        }
        else if( state == 10 ){
            if( ch == '"' || ch == '\'' ){
                state = -1;
            }
        }
        if( iserror ) {
            cout << "error happened in automata" << endl;
            getchar();
            //exit(EXIT_FAILURE);
        }
        if(state != -1){
            concatenation();
            pre = ch;
            Getchar();
        }
    }
    return;
}
```

该函数为自动机的实现，逻辑见（3）中的图。代码中，用一个变量 `state` 表示当前状态，然后根据当前字符，判断应该进入哪个状态。终止状态为 -1.

- `void LexicalAnalyzer(char* fileName)`

```
void LexicalAnalyzer(char* fileName){
    ofstream outFile;
    inFile.open(fileName, ios::in);
    inFile.unsetf(ios_base::skipws);//逐字符读取文件，不跳过空白符
    outFile.open("./LexicalAnalyzerOut.txt",ios::out);
    if( !inFile.is_open() ){
        cout << "Could not find the file!\n";
        getchar();
        //exit(EXIT_FAILURE);
    }
```

```
    Getchar();
    outFile.setf(ios::left);
    outFile << setfill(' ')<<setw(66) << "Characters";
    outFile  << "Code" << endl;
    while(!inFile.eof()){
        ptr_token = 0;
        automata();
        int flag = -1;
        if( !( pre == '"' || pre == '\'' ) && isString == 1 ){ //token是字符串
            flag = 2;
            outFile.setf(ios::left);
            outFile << setfill(' ')<<setw(75) << token;
            outFile  << flag << endl;
            //isString = 0;
        }
        else{
            flag = reserve();
            if( flag != -1 ){ //token为关键字
                outFile.setf(ios::left);
                outFile << setfill(' ')<<setw(75) << token ;
                outFile  << flag << endl;
            }
            else{
                flag = opra();
                if( flag != -1 ){
                    outFile.setf(ios::left);
                    outFile << setfill(' ')<<setw(75) << token;
                    outFile  << flag << endl;
                }
                else{
                    flag = bp();
                    if( flag != -1 ){ //token是界符
                        outFile.setf(ios::left);
                        outFile << setfill(' ')<<setw(75) << token;
                        outFile  << flag << endl;
                        buildtable();
                        if( pre == '"' || pre == '\'' )
                            isString = isString==1 ? 0 : 1;

                    }
                    else{ //token是标识符或常数
                        if( token[0] >= '0' && token[0] <= '9' ) {//token是常数
                            flag = 0;
                            outFile.setf(ios::left);
                            outFile << setfill(' ')<<setw(75) << token;
                            outFile  << flag << endl;
                            buildtable();
                        }
                        else{//token是标识符
                            flag = 1;
                            outFile.setf(ios::left);
                            outFile << setfill(' ')<<setw(75) << token;
                            outFile  << flag << endl;
                            buildtable();
                        }
                    }
```

```
                }
            }
        }
    }
    outFile.close();
    inFile.close();
}
```

该函数实现整个词法分析器的功能。首先是打开要读取的文件，读取文件字符，调用自动机实现函数，得到 `token` ，再调用判断 `token` 的函数，得到该 `token` 的种别编码信息。如果到此还没有报错，则将 `token` 和其种别编码信息写入文件 `./LexicalAnalyzerOut.txt` 。 循环读取文件字符，直到字符串读取完毕。最后关闭打开的两个文件。

- `int main()`

```
int main(){
    char* fileName = "test.c";
    LexicalAnalyzer(fileName);
    cout << "Finished\n";
    getchar();
    return 0;
}
```

在 `main` 函数中指定要读取的文件名字，分析完毕后输出提示信息 "Finished" 。

至此，我的词法分析器已实现完毕。

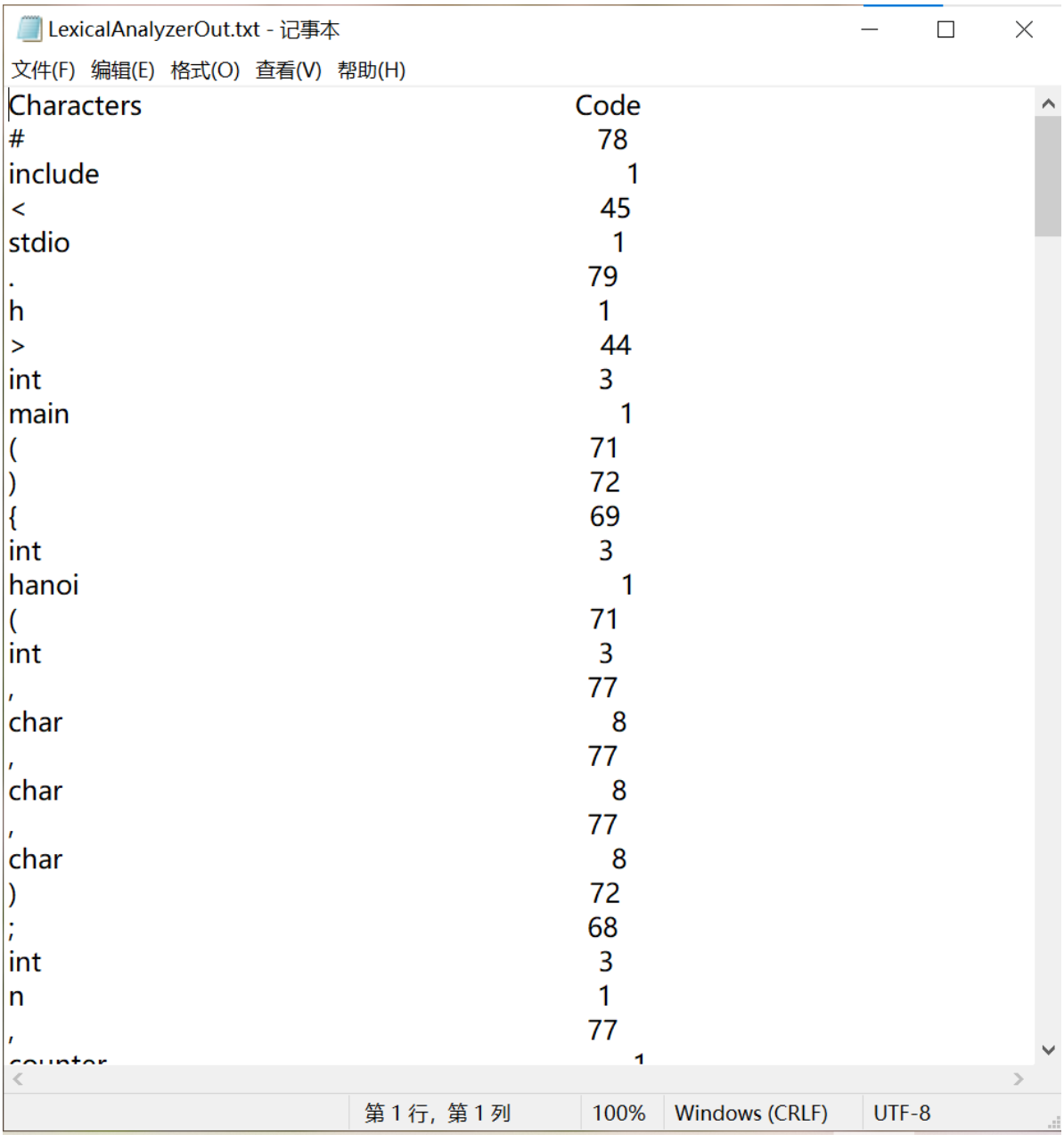
实验测试

测试代码为在网上 copy 的一段实现汉诺塔的 C 语言代码，如下：

```
#include <stdio.h>
int main()
{
    int hanoi(int,char,char,char);
    int n,counter;
    printf("Input the number of diskes: ");
    scanf("%d",&n);
    printf("\n");
    counter=hanoi(n,'A','B','C');
    return 0;
}
int hanoi(int n,char x,char y,char z)
{
    int move(char,int,char);
    if(n==1)
        move(x,1,z);
    else
    {
        hanoi(n-1,x,z,y);
        move(x,n,z);
        hanoi(n-1,y,x,z);
    }
    return 0;
}
```

```c
int move(char getone,int n,char putone)
{
    static int k=1;
    printf("%2d:%3d # %c---%c\n",k,n,getone,putone);
    if(k++%3==0)
        printf("\n");
    return 0;
}
```

实验结果



实验输出文件`LexicalAnalyzerOut.txt`将打包在压缩文件中。

实验感想

（1）此次实验让我对词法分析器的作用有了深刻的了解。

词法分析器的主要任务是读入源程序的输入字符、将它们组成词素，生成并输出一个词法单元序列，每个词法单元对应于一个词素。

当词法分析器发现了一个标识符的词素时，要将这个词素添加到符号表中。

同时词法分析器还需要完成：

- 过滤掉源程序中的注释和空白。
- 将编译器生成的错误信息与源程序的位置联系起来。记录行号等。

（2）在设计词法分析器时，先要确定该语言的保留字、支持的运算符以及界符有哪些，然后再确定你要实现的词法分析器能支持到什么程度，然后进一步设计自动机，和进行代码的实现。

（3）自动机的设计体现了词法分析器的运行逻辑，自动机设计得好可以让代码更简洁明了。

（4）此处实现的功能有

- 对C语言全部关键字、运算符、界符进行识别；
- 对常数、字符串和标识符进行识别和存储；
- 对字符串的合法性进行判断。