```c
// 文件: scanner.c
// 内容: 实现词法分析器( C 语言版 )
// 作者: WXQ#2018

#include <stdio.h>   // fopen(), fclose(), getc(), ungetc()
#include <string.h>  // strcmp(), strlen(), memset()
#include <ctype.h>   // isspace(), toupper()
#include <stdlib.h>  // atof() of GCC
#include <math.h>    // atof(), sin(), cos(), ...

#include "scanner.h" // interface of Lexer

/* BEGIN 辅助的全局变量定义 */

struct Token TokenTab[] =    // 符号表内容
{
    {CONST_ID,    "PI",       3.1415926,  NULL, {0,0}}, // 命名常数
    {CONST_ID,    "E",        2.71828,    NULL, {0,0}},
    {T,           "T",        0.0,        NULL, {0,0}}, // 参数

    {FUNC,        "SIN",      0.0,        sin , {0,0}},  // 支持的函数名
    {FUNC,        "COS",      0.0,        cos , {0,0}},  // 这些函数都是 math.h 的
    {FUNC,        "TAN",      0.0,        tan , {0,0}},  // 当然你也可以自己实现!
    {FUNC,        "LN",       0.0,        log , {0,0}},
    {FUNC,        "EXP",      0.0,        exp , {0,0}},
    {FUNC,        "SQRT",     0.0,        sqrt, {0,0}},

    {ORIGIN,      "ORIGIN",   0.0,        NULL, {0,0}}, // 保留关键字
    {SCALE,       "SCALE",    0.0,        NULL, {0,0}},
    {ROT,         "ROT",      0.0,        NULL, {0,0}},
    {IS,          "IS",       0.0,        NULL, {0,0}},
    {FOR,         "FOR",      0.0,        NULL, {0,0}},
    {FROM,        "FROM",     0.0,        NULL, {0,0}},
    {TO,          "TO",       0.0,        NULL, {0,0}},
    {STEP,        "STEP",     0.0,        NULL, {0,0}},
    {DRAW,        "DRAW",     0.0,        NULL, {0,0}}
};

FILE * in_flie      =NULL;  // 指向输入文件的指针
struct position  current_pos={1, 0};  // 当前记号所在行、列号
/* END 辅助的全局变量定义 */



// 接口操作1: 初始化词法分析器
int InitScanner(const char* fileName)
{
    current_pos.line = 1;
    current_pos.col  = 0;

    in_flie = fopen(fileName, "r");
    if (in_flie != NULL)
        return 1;
    else
        return 0;
}

// 接口操作2: 关闭词法分析器
void CloseScanner()
{
    if (in_flie != NULL)
        fclose (in_flie) ;
```

```
}

// 三个辅助函数，定义在后面
int pre_process(struct Token *pToken);
int scan_move(struct Token * pToken, int first_char);
int post_process(struct Token *pToken, int last_state);

// 接口操作3：识别并返回一个记号。
// 遇到非法输入时 .type=ERRTOKEN、文件结束时 .type=NONTOKEN
struct Token GetToken()
{
    int first_char;                 // 记号开始的第1个字符
    int last_state = -1;            // 识别记号结束时的状态
    struct position where;          // 当前记号的起始位置
    struct Token    theToken;       // 被识别到的记号对象
    int  to_bo_continue;

    do
    {
    // 第1步：预处理，跳过空白字符
        first_char = pre_process(&theToken);
        if (first_char == -1)  // 文件结束了
        {
            theToken.type = NONTOKEN;
            return theToken;
        }
        theToken.where = where = current_pos;  // 当前记号开始的位置

    // 第2步：边扫描输入，边转移状态
        last_state = scan_move(&theToken, first_char);

    // 第3步：后处理：根据终态所标记的记号种类信息，进行特殊处理
        to_bo_continue = post_process(&theToken, last_state);
    } while (to_bo_continue != 0);

    theToken.where = where; //修正记号的位置
    return theToken;
}

//
// 下面是辅助函数、辅助变量等定义
//

// 从输入源程序中读入一个字符并返回它
// 若遇到文件结束则返回 -1.
char GetChar(void)
{
    int next_char = getc(in_flie);
    if( EOF == next_char )
        return -1;
    else
    {
        if('\n' == next_char)
        {
            ++ (current_pos.line);  current_pos.col = 0;
            return next_char;
        }
        else
        {
            ++ (current_pos.col);
            return toupper(next_char); // 统一返回大写
        }
```

```
    }
}

// 把预读的字符退回到输入源程序中
void BackChar(char next_char)
{
    // 文件结束标志、换行不回退
    if( next_char == EOF || next_char == '\n')
        return;

    ungetc(next_char, in_flie);
    --(current_pos.col);
}

//  判断所给的字符串是否在符号表中
struct Token JudgeKeyToken(const char * c_str)
{
    int count;
    struct Token err_token;

    for (count=0; count<sizeof(TokenTab)/sizeof(TokenTab[0]); ++count)
    {
        if (strcmp(TokenTab[count].lexeme, c_str)==0)
            return TokenTab[count];
    }

    memset(&err_token, 0, sizeof(err_token));
    err_token.type = ERRTOKEN;
    return err_token;
}


// 将字符c追加到记号文本末尾.
// 若超长则不再追加并返回-1，否则返回0.
int AppendTokenTxt(struct Token* pToken, char c)
{
    size_t len;
    len = strlen (pToken->lexeme);
    if (len + 1 >= sizeof (pToken->lexeme))
         return -1;
    pToken->lexeme[len]   = c;
    pToken->lexeme[len+1] = '\0';
    return 0;
}


int is_space(char c)
{
    if( c<0 || c > 0x7e ) // 处理非 ASCII 字符，如（半个）中文字符
        return 0;
    return isspace(c);
}



// 识别一个记号的前处理：跳过空白字符，并读取第1个非空白字符.
// 返回值：  -1 文件结束，其他值表示字符本身.
int pre_process(struct Token *pToken)
{
    int current_char; // 当前读到的字符
    memset(pToken, 0, sizeof(struct Token)); // 记号内存清零
    for (;;)
```

```
187     {
188         current_char = GetChar() ;
189         if (current_char == -1)
190         {
191             return -1;
192         }
193         if (!is_space(current_char))   break ;
194     } // end of for
195     //
196     // 此时，current_char 就是记号的第1个字符
197 
198     return current_char;
199 }
200 
201 
202 // DFA 提供的接口操作，定义在文件 dfa.c
203 extern int  get_start_state();
204 extern int  move( int state_src,  char ch );
205 extern enum Token_Type  state_is_final(int state);
206 
207 // 识别记号的核心操作
208 int scan_move(struct Token * pToken, int first_char)
209 {
210     int current_state, next_state; // 当前状态，下一状态
211     int current_char;              // 当前字符
212 
213     current_char  = first_char;
214     current_state = get_start_state();
215     for (;;)
216     {
217         next_state = move(current_state, current_char);
218         if (next_state < 0) // 没有转移了
219         {
220             //第一个字符就无效，则丢弃它.否则因为反复读到该字符而陷入死循环
221             if (pToken->lexeme[0] == '\0')
222             {
223                 snprintf(pToken->lexeme, sizeof(pToken->lexeme),
224                     "\\X%02X", (unsigned char)current_char);
225             }
226             else
227             {
228                 BackChar(current_char); // 退回当前字符，它应该是下一记号的开始
229             }
230             break;
231         }
232 
233         AppendTokenTxt(pToken, current_char);  // 追加记号的文本
234 
235         current_state = next_state;
236         current_char = GetChar();
237         if (current_char == -1) // 文件结束了
238             break;
239     }
240 
241     return current_state;
242 }
243 
244 
245 // 根据终态所标记的记号种类信息，进行特殊处理.
246 // 若返回非0，则表示当前刚处理完了“注释”，需要调用者接着获取下一个记号。
247 int post_process(struct Token *pToken, int last_state)
248 {
```

```
249     int  to_bo_continue = 0; // FALSE
250     enum Token_Type tk = state_is_final( last_state );
251     switch( tk )
252     {
253     case ID:  // 查符号表，进一步计算记号信息
254         {
255             struct Token id = JudgeKeyToken(pToken->lexeme);
256             if(ERRTOKEN == id.type)
257                 pToken->type = ERRTOKEN;
258             else
259                 *pToken = id;
260         }
261         break;
262     case CONST_ID:
263         pToken->type = tk;
264         pToken->value = atof(pToken->lexeme); // 转为数值
265         break;
266     case COMMENT:  // 行注释：忽略直到行尾的文本，并读取下一个记号
267         {   int c;
268             while (1) {
269                 c = GetChar();
270                 if (c == '\n' || c == -1)
271                     break;
272             }
273         }
274         to_bo_continue = 1; // TRUE
275         break;
276     default:
277         pToken->type = tk;
278         break;
279     }
280
281     return to_bo_continue;
282 }
283
284 //
285 // 下面的代码仅为显示记号种类的名称之文本
286 //
287 struct tk_print
288 {
289     enum Token_Type tk;
290     const char*     str;
291 };
292
293 #define MKSTR(x) { x , #x }
294 struct tk_print tk_names[]=
295 {
296     MKSTR( ORIGIN ),    MKSTR( SCALE ),    MKSTR( ROT ),
297     MKSTR( IS ),        MKSTR( TO ),       MKSTR( STEP ),
298     MKSTR( DRAW ),      MKSTR( FOR ),      MKSTR( FROM ),
299     MKSTR( T ),
300
301     MKSTR( SEMICO ),    MKSTR( L_BRACKET ),
302     MKSTR( R_BRACKET ), MKSTR( COMMA ),
303
304     MKSTR( PLUS ),      MKSTR( MINUS ),    MKSTR( MUL ),
305     MKSTR( DIV ),       MKSTR( POWER ),
306     MKSTR( FUNC ),
307     MKSTR( CONST_ID ),
308     MKSTR( ERRTOKEN ),
309
310     MKSTR( NONTOKEN ) // flag item for end
```

```
311 };
312
313 const char* token_type_str(enum Token_Type tk)
314 {
315     struct tk_print * p = tk_names;
316     for( ; p->tk != NONTOKEN ; ++p)
317     {   if( p->tk == tk )
318             return p->str;
319     }
320     return "UNKNOW";
321 }
322
```