```c
// 文件: parser.c
// 内容: 实现语法分析器( C 语言版 )
// 作者: WXQ#2018


/* 重要说明
  本文件是在单纯的语法分析器模块 parser/parser.c 的
  基础上，增加了语义处理所需的代码。

  这些代码均用
  #ifdef _PARSER_WITH_SEMANTICS
  ...
  #endif
  包围。
 */

#include <stdio.h>   // for fopen, fclose
#include <stdlib.h>  // for malloc(), free(), atexit()
#include <stdarg.h>  // for va_start(), va_end(), va_arg()

#include "errlog.h"  // 错误处理模块的接口
#include "parser.h"

#ifdef _PARSER_WITH_SEMANTICS // 带有语义计算的语法分析器 = 语义分析器
#include "semantics.h"
#endif
//
// 非终结符对应的递归下降子程序，这里先声明，后面再定义
//

void program();
void statement();
void for_statement();
void origin_statement();
void rot_statement();
void scale_statement();
ExprNode_Ptr expression();
ExprNode_Ptr term();
ExprNode_Ptr factor();
ExprNode_Ptr component();
ExprNode_Ptr atom();


// ~~~~~ 先声明辅助函数，后面再给出定义
int InitParser(const char* file_name); // 语法分析器的初始化
void CloseParser();
void destroy_all_tree();

void FetchToken ();                          // 获取记号
void MatchToken (enum Token_Type expected);// 匹配记号
void SyntaxError (int case_of);                // 指出语法错误(调用error_msg)
void PrintSyntaxTree(ExprNode_Ptr root, int indent);        // 打印语法树
ExprNode_Ptr MakeExprNode(enum Token_Type opcode,...);    // 构造语法树

// 下述函数用于跟踪调试，显示进入、退出哪个非终结符的函数。
void enter(char * x);
void back(char * x);
void call_match(char * x);
void tree_trace(ExprNode_Ptr x);


// 语法分析器的辅助数据类型.
```

```
63 // 用结构体将杂乱的数据打包成一个整体，便于管理。
64 // 若不用这个结构体类型，则你就必须将这些数据定义为全局变量 :-(
65 struct t_parser_stuff
66 {
67     int         indent;    // 显示信息的缩进量
68     ExprNode_Ptr  start_ptr,     // 绘图起点表达式的语法树
69             end_ptr,       // 绘图终点表达式的语法树
70             step_ptr,      // 步长表达式的语法树
71             x_ptr,         // 点的横坐标表达式的语法树
72             y_ptr,         // 点的横坐标表达式的语法树
73             angle_ptr;     // 旋转角度表达式的语法树
74     struct Token  curr_token;  // 当前记号
75
76     // 语义相关数据
77     double        parameter;     // 参数T的存储空间
78 };
79
80 // 本文件内使用的数据
81 static struct t_parser_stuff  parser_data;
82 #define current_token parser_data.curr_token
83
84 #ifdef _PARSER_WITH_SEMANTICS
85 double * getTmemory() { return &(parser_data.parameter); }
86 #endif
87
88
89 // 绘图语言语法分析器的入口
90 void Parser(const char * file_name)
91 {
92     // 语法分析器的初始化（含初始化词法分析器）
93     if( InitParser(file_name) == 0 )
94         return;
95
96     FetchToken();       // 获取第一个记号
97     program();          // 递归下降分析
98
99     CloseParser();
100    return;
101 }
102
103
104 // ------------------------ program 的递归子程序
105 void program ()
106 {
107     enter("program");
108     while (current_token.type != NONTOKEN)
109     {
110         statement();
111         MatchToken (SEMICO) ;
112     }
113     back("program");
114 }
115
116 // ------------------------ statement 的递归子程序
117 void statement()
118 {
119     enter("statement");
120     switch (current_token.type) // LL(1)
121     {
122     case ORIGIN :    origin_statement();    break;
123     case SCALE  :    scale_statement();     break;
124     case ROT    :    rot_statement();       break;
```

```
    case FOR    :    for_statement();       break;
    default     :    SyntaxError(2);
    }

    // 销毁可能存在的语法树，否则内存泄漏。
    // 这里假设这些树不会被再次使用!!!
    destroy_all_tree();

    back("statement");
}

// ------------------------ origin_statement 的递归子程序
void origin_statement()
{
    enter("origin_statement");

    MatchToken  (ORIGIN);
    MatchToken  (IS);
    MatchToken  (L_BRACKET);
                    parser_data.x_ptr = expression();
                    tree_trace(parser_data.x_ptr);  // print AST
    MatchToken  (COMMA);
                    parser_data.y_ptr = expression();
                    tree_trace(parser_data.y_ptr);  // print AST
    MatchToken  (R_BRACKET);

#ifdef _PARSER_WITH_SEMANTICS  // 解释器所需的语义计算
    setOrigin(parser_data.x_ptr, parser_data.y_ptr);
#endif

    back("origin_statement");
}

// ------------------------ scale_statement 的递归子程序
void scale_statement ()
{
    enter("scale_statement");

    MatchToken  (SCALE);
    MatchToken  (IS);
    MatchToken  (L_BRACKET);
                    parser_data.x_ptr = expression();
                    tree_trace(parser_data.x_ptr); // print AST
    MatchToken  (COMMA);
                    parser_data.y_ptr = expression();
                    tree_trace(parser_data.y_ptr); // print AST
    MatchToken  (R_BRACKET);

#ifdef _PARSER_WITH_SEMANTICS  // 解释器所需的语义计算
    setScale(parser_data.x_ptr, parser_data.y_ptr);
#endif

    back("scale_statement");
}

// ------------------------ rot_statement 的递归子程序
void rot_statement ()
{
    enter("rot_statement");

    MatchToken  (ROT);
    MatchToken  (IS);
```

```
187                     parser_data.angle_ptr = expression();
188                     tree_trace(parser_data.angle_ptr); // print AST
189
190 #ifdef _PARSER_WITH_SEMANTICS  // 解释器所需的语义计算
191     setRotate( parser_data.angle_ptr );
192 #endif
193
194     back("rot_statement");
195 }
196
197
198 // ------------------------ for_statement 的递归子程序
199 void for_statement ()
200 {
201     enter("for_statement");
202
203     MatchToken (FOR);
204     MatchToken (T);
205     MatchToken (FROM);
206                     parser_data.start_ptr = expression();   // 构造参数起始表达式语法树
207                     tree_trace(parser_data.start_ptr);
208     MatchToken (TO);
209                     parser_data.end_ptr = expression();     // 构造参数终结表达式语法树
210                     tree_trace(parser_data.end_ptr);
211     MatchToken (STEP);
212                     parser_data.step_ptr = expression();    // 构造参数步长表达式语法树
213                     tree_trace(parser_data.step_ptr);
214     MatchToken (DRAW);
215     MatchToken (L_BRACKET);
216                     parser_data.x_ptr = expression();       // 构造横坐标表达式语法树
217                     tree_trace(parser_data.x_ptr);
218     MatchToken (COMMA);
219                     parser_data.y_ptr = expression();       // 构造纵坐标表达式语法树
220                     tree_trace(parser_data.y_ptr);
221     MatchToken (R_BRACKET);
222
223 #ifdef _PARSER_WITH_SEMANTICS  // 解释器所需的语义计算
224     DrawLoop (parser_data.start_ptr, parser_data.end_ptr,
225               parser_data.step_ptr,
226               parser_data.x_ptr, parser_data.y_ptr
227              ); // 绘图
228 #endif
229
230     back("for_statement");
231 }
232
233 // ------------------------ expression 的递归子程序
234 ExprNode_Ptr expression()
235 {
236     ExprNode_Ptr left, right;            // 左右子树节点的指针
237     Token_Type  lastType;
238
239     enter("expression");
240     left = term();                    // 分析左操作数且得到其语法树
241     while (current_token.type==PLUS || current_token.type==MINUS)
242     {
243         lastType = current_token.type;
244         MatchToken (lastType);
245         right = term();                // 分析右操作数且得到其语法树
246         left = MakeExprNode(lastType, left, right);
247                             // 构造运算的语法树，结果为左子树
248     }
```

```
//  none sub AST
//    tree_trace(left);                // 打印表达式的语法树
//
    back("expression");
    return left;                    // 返回最终表达式的语法树
}

// ----------------------- term 的递归子程序
ExprNode_Ptr  term()
{
    ExprNode_Ptr left, right;
    Token_Type lastType;

    left = factor();
    while (current_token.type==MUL || current_token.type==DIV)
    {
        lastType = current_token.type;
        MatchToken (lastType);
        right = factor();
        left = MakeExprNode(lastType, left, right);
    }
    return left;
}

// ----------------------- factor 的递归子程序
ExprNode_Ptr factor ()
{
    ExprNode_Ptr left, right;

    if(current_token.type == PLUS)             // 匹配一元加运算
    {
        MatchToken (PLUS);
        right = factor();        //  表达式退化为仅有右操作数的表达式
    }
    else if(current_token.type == MINUS)        // 匹配一元减运算
    {
        MatchToken (MINUS);         // 表达式转化为二元减运算的表达式
        right = factor();
        left = (ExprNode_Ptr)malloc(sizeof(ExprNode));
        left->OpCode = CONST_ID;
        left->content.CaseConst = 0.0;
        right = MakeExprNode(MINUS, left, right);
    }
    else right = component();        // 匹配非终结符component
    return right;
}

// ----------------------- component 的递归子程序
ExprNode_Ptr component()
{
    ExprNode_Ptr left, right;

    left = atom();
    if(current_token.type == POWER)
    {
        MatchToken (POWER);
        right = component();    // 递归调用component以实现POWER的右结合性质
        left  = MakeExprNode(POWER, left, right);
    }
    return left;
}

```

```
311 // ------------------------ atom 的递归子程序
312 ExprNode_Ptr atom()
313 {
314     struct Token t = current_token;
315     ExprNode_Ptr ptr, tmp;
316 
317     switch (current_token.type)
318     {    case CONST_ID :
319             MatchToken  (CONST_ID) ;
320             ptr = MakeExprNode(CONST_ID,t.value);
321             break;
322         case T:
323             MatchToken (T);
324             ptr = MakeExprNode(T);
325             break;
326         case FUNC :
327             MatchToken  (FUNC);
328             MatchToken  (L_BRACKET);
329             tmp = expression ();
330             ptr = MakeExprNode(FUNC, t.FuncPtr, tmp);
331             MatchToken  (R_BRACKET);
332             break ;
333         case L_BRACKET :
334             MatchToken  (L_BRACKET);
335             ptr = expression ();
336             MatchToken  (R_BRACKET);
337             break ;
338         default :
339             SyntaxError (2);
340     }
341     return ptr;
342 }
343 
344 
345 //
346 //  下面是辅助函数的定义
347 //
348 
349 
350 
351 void destroy_tree(ExprNode_Ptr root)
352 {
353     if(NULL == root) return;
354 
355     switch( root->OpCode )
356     {
357     case PLUS:
358     case MINUS:
359     case MUL:
360     case DIV:
361     case POWER:
362         destroy_tree(root->content.CaseOperator.left);
363         destroy_tree(root->content.CaseOperator.right);
364         break;
365     case FUNC:
366         destroy_tree(root->content.CaseFunc.child);
367         break;
368     default:
369         break;
370     }
371 
372     free( root );
```

```
373     return;
374 }
375 void destroy_all_tree()
376 {
377     destroy_tree(parser_data.start_ptr); parser_data.start_ptr = NULL;
378     destroy_tree(parser_data.end_ptr);   parser_data.end_ptr   = NULL;
379     destroy_tree(parser_data.step_ptr);  parser_data.step_ptr  = NULL;
380     destroy_tree(parser_data.x_ptr);     parser_data.x_ptr     = NULL;
381     destroy_tree(parser_data.y_ptr);     parser_data.y_ptr     = NULL;
382     destroy_tree(parser_data.angle_ptr); parser_data.angle_ptr = NULL;
383 }
384
385 void CloseParser()
386 {
387     CloseScanner();      // 关闭词法分析器
388     destroy_all_tree();
389 }
390
391
392 int InitParser(const char* file_name) // 语法分析器的初始化
393 {
394     parser_data.indent = 0;
395
396     parser_data.parameter = 0;
397     parser_data.start_ptr = NULL;
398     parser_data.end_ptr   = NULL;
399     parser_data.step_ptr  = NULL;
400     parser_data.x_ptr     = NULL;
401     parser_data.y_ptr     = NULL;
402     parser_data.angle_ptr = NULL;
403
404     atexit(CloseParser);
405
406     if( InitScanner(file_name) == 0)  // 初始化词法分析器
407     {
408         logPrint("打开文件[%s]失败 !\n", file_name);
409         return 0;
410     }
411
412     logPrint("分析文件[%s]...\n", file_name);
413
414     return 1;
415 }
416
417 //  通过词法分析器接口 get_token 获取一个记号
418 void FetchToken ()
419 {
420     current_token = GetToken () ;
421     if (current_token.type == ERRTOKEN)    SyntaxError(1);
422 }
423
424 //  匹配记号
425 void MatchToken(enum Token_Type expected)
426 {
427     if (current_token.type != expected)
428         SyntaxError(2);
429     else // 若匹配上了记号，则打印记号文本
430     {
431         int i;
432         for(i=0; i<parser_data.indent; i++) logPrint( " " );
433         logPrint("matchtoken %s\n", current_token.lexeme);
434     }
```

```
435     FetchToken(); //接着读取下一记号
436 }
437
438 // 语法错误处理
439 void SyntaxError (int case_of)
440 {
441     switch(case_of)
442     {
443     case 1:
444         error_msg (current_token.where.line,"非法单词 ", current_token.lexeme) ;
445         break;
446     case 2:
447         error_msg (current_token.where.line, current_token.lexeme, "不是预期记号") ;
448         break;
449     }
450     exit(1);
451 }
452
453 // 先序遍历并打印表达式的语法树
454 void PrintSyntaxTree(ExprNode_Ptr root, int indent)
455 {
456     int  L;
457
458     if(NULL == root) return;
459
460     L =   indent
461         + parser_data.indent; // 与 "enter..." 对齐
462
463     for ( ; L > 0; --L) logPrint(" ");  // 缩进
464     switch(root->OpCode)  // 打印子树
465     {
466     case PLUS:
467         logPrint("+\n"); break;
468     case MINUS:
469         logPrint("-\n"); break;
470     case MUL:
471         logPrint("*\n"); break;
472     case DIV:
473         logPrint("/\n"); break;
474     case POWER:
475         logPrint("**\n"); break;
476     case T:
477         logPrint("T\n");
478         return;
479     case FUNC:
480         logPrint("%p\n", root->content.CaseFunc.MathFuncPtr);
481         // 递归打印孩子节点
482         PrintSyntaxTree(root->content.CaseFunc.child, indent+2);
483         return;
484     case CONST_ID:
485         logPrint("%lf\n", root->content.CaseConst);
486         return;
487     default:
488         logPrint("非法的树节点\n");
489         return;
490     }
491
492     // !! 只有操作符结点才会走到这里!!
493     // 递归打印两个孩子的节点
494     PrintSyntaxTree(root->content.CaseOperator.left,  indent+2);
495     PrintSyntaxTree(root->content.CaseOperator.right, indent+2);
496     return;
```

```
}

//  生成语法树的一个节点
ExprNode_Ptr MakeExprNode(enum Token_Type opcode, ...)
{
    ExprNode_Ptr ptr = (ExprNode_Ptr)malloc(sizeof(ExprNode));   // 分配节点存储空间
    ptr->OpCode = opcode;                // 接收记号的类别
    va_list arg_ptr ;
    va_start (arg_ptr, opcode);
    switch(opcode)                        // 根据记号的类别构造不同的节点
    {    case CONST_ID:                    // 常数节点
            ptr->content.CaseConst = (double)va_arg(arg_ptr, double);
            break;
        case T:                          // 参数节点
            ptr->content.CaseParmPtr = &(parser_data.parameter);
            break;
        case FUNC:                      // 函数调用节点
            ptr->content.CaseFunc.MathFuncPtr = (t_func)va_arg(arg_ptr, t_func);
            ptr->content.CaseFunc.child
                    = (ExprNode_Ptr)va_arg (arg_ptr, ExprNode_Ptr);
            break;
        default:                    // 二元运算节点
            ptr->content.CaseOperator.left
                    = (ExprNode_Ptr)va_arg (arg_ptr, ExprNode_Ptr);
            ptr->content.CaseOperator.right
                    = (ExprNode_Ptr)va_arg (arg_ptr, ExprNode_Ptr);
            break;
    }
    va_end(arg_ptr);
    return ptr;
}



//  用于语法分析器中的跟踪调试

void enter(char * x)
{
    int i;
    for(i=0; i<parser_data.indent; ++i) logPrint( " " );
    logPrint( "enter in %s\n", x);
    parser_data.indent += 2;
}
void back(char * x)
{
    int i;
    parser_data.indent -= 2;
    for(i=0; i<parser_data.indent; ++i) logPrint( " " );
    logPrint( "exit from  %s\n",  x);
}

void tree_trace(ExprNode_Ptr x)
{
/*
    int L = parser_data.indent; // 与 "enter..." 对齐
    for ( ; L > 0; --L) logPrint(" ");  // 缩进
    logPrint( "TREE:\n");
*/
    PrintSyntaxTree(x, 0);
}

```