```c
// 文件: dfa.c
// 内容: 实现表驱动型的词法分析器之驱动器
// 作者: WXQ#2018


// 表驱动型、直接编码型的词法分析器的差异从这里开始
// {{{{

/*  !!!! important information !!!
 * (1) 本文基于“压缩存储稀疏矩阵”的原理，采用[三元组]存放一个有效转移，
 *     且将行下标、列下标合并到一个int来存放。
 * (2) 本实现要求 DFA 的每个终态最多只能识别一类记号。
 *     所以要对《习题解答》中的DFA进行改造 !!!
 */

#define NULL 0
#include "scanner.h"

//
// Part I  状态转移及定义
//

#define CK_CHAR   (0  << 16)   // 单个字符
#define CK_LETTER (1U << 16)   // 字母 [a-zA-Z]
#define CK_DIGIT  (2U << 16)   // 数字 [0-9]
#define CK_NULL   (0x80U << 16)


/* struct t_state_trans
 * 定义一个状态转移，包括
 *   起始状态（矩阵的行下标），
 *   字符（矩阵的列下标），
 *   目标状态（矩阵单元格的值）.
 *
 * 为查找方便及效率考虑，该结构将行下标、列下标合并存放到 idx 字段，
 * 其中第1字节对应行下标(要求状态编号范围为 [0-254])，
 * 第2字节对应字符种类(上述宏 CK_*)，
 * 第3-4字节为字符值（且仅当第3字节为 0 时有效）。
 */
typedef unsigned int t_key;
struct t_state_trans
{
    t_key idx;      // move(s,ch)的参数合成的查询关键字
    int   state_to; // 转移的目标状态
};

#define MK_IDX(from, c)      ((t_key)(from<<24) | (c))
#define MOVE(from, c, to)    { MK_IDX(from, c),  to }
#define TRANS_END            MK_IDX(255, CK_NULL)

// 状态转移矩阵定义
// 这里将该矩阵当作“稀疏矩阵”，且每个转移采用“三元组”结构。
// 为方便查询，将当前状态、字符组合为一个整数（Ref: MK_IDX）
struct t_state_trans  myTransfers[] =
{
    MOVE(0, CK_LETTER, 1 ),    // from state 0
    MOVE(0, CK_DIGIT , 2 ),
    MOVE(0,  '*'     , 4 ),
    MOVE(0,  '/'     , 6 ),
    MOVE(0,  '-'     , 7 ),
    MOVE(0,  '+'     , 8 ),
    MOVE(0,  ','     , 9 ),
```

```
    MOVE(0,  ';'     , 10),
    MOVE(0,  '('     , 11),
    MOVE(0,  ')'     , 12),
    MOVE(1, CK_LETTER, 1 ),    // from state 1
    MOVE(1, CK_DIGIT , 1 ),
    MOVE(2, CK_DIGIT , 2 ),    // from state 2
    MOVE(2,  '.'     , 3 ),
    MOVE(3, CK_DIGIT , 3 ),    // from state 3
    MOVE(4,  '*'     , 5 ),    // from state 4
    MOVE(6,  '/'     , 13),    // from state 6
    MOVE(7,  '-'     , 13),    // from state 7

    {TRANS_END, 255}             // 结束标志
};

//
// Part II  终态及定义
//

struct  t_final_state // 终态的数据结构
{
    int        state; // 终态
    Token_Type kink;  // 该终态所识别的记号类别
};

//所有终态的定义，形成一个数组
struct  t_final_state myFinalStates[] =
{
    {1 , ID },      // 识别后应进一步确定是哪个关键字
    {2 , CONST_ID}, // 识别后应得到对应的数值
    {3 , CONST_ID}, // 同2

    {4 , MUL},
    {5 , POWER},
    {6 , DIV},
    {7 , MINUS},

    {8 , PLUS},    // 以下终态需从《解答》P66的DFA中拆解出来
    {9 , COMMA},
    {10, SEMICO},
    {11, L_BRACKET},
    {12, R_BRACKET},

    {13, COMMENT}     // 识别后应丢弃直到行尾的字符

    ,{-1, ERRTOKEN} // 该元素是结束标志
};


//
// Part III   DFA 的定义
//

struct DFA_definition   // DFA 的完整数据结构
{
    int    start_state;                // 初态，唯一
    struct t_final_state * final_state; // 终态，不唯一
    struct t_state_trans * transfers;   // 所有的状态转移，即状态转移矩阵

};

struct DFA_definition  myDFA =
```

```
{
    0, myFinalStates, myTransfers
};


//
// Part IV DFA 为记号识别提供的接口
//

// 查询初态是哪个
int  get_start_state()
{
    return myDFA.start_state;
}

// 判断指定状态是否为DFA的终态。
// 若是，则返回该终态对应的记号类别，否则返回 ERRTOKEN.
enum Token_Type  state_is_final(int state)
{
    struct  t_final_state *p = myDFA.final_state;
    for( ; p->state > 0; ++p)
    {
        if( p->state == state )
            return p->kink;
    }
    return ERRTOKEN;
}

// 函数 get_next_state()
// 功能：
//   根据当前状态、当前遇到的字符，进行状态转移。
//   该函数相当于NFA定义中的 move() 函数.
// 返回值定义：
//   -1: 此时没有状态转移（假设DFA所有状态编号均 >= 0）
//  >=0: 新的状态
//
// 注意：
//   若遇到当前 DFA 不接受的字符时，则没有状态转移.

int  move(
            int state_src, // 当前状态(0~255)
            char ch        // 当前字符
         )
{
    // 先计算字符 ch 的种类码
    int ck_of_ch = CK_CHAR;
    if( '0' <= ch && ch <= '9' )
        { ck_of_ch = CK_DIGIT;  ch =0; }
    else if( ('a' <= ch && ch <= 'z') || ('A' <= ch && ch <= 'Z') )
        { ck_of_ch = CK_LETTER; ch =0; }
    else
        ck_of_ch = CK_CHAR;

    // 生成查询的关键字
    t_key key = MK_IDX(state_src, ck_of_ch | ch);

    //查找转移
    struct t_state_trans * pTransfer =  myDFA.transfers;
    for( ; pTransfer->idx != TRANS_END; ++pTransfer )
    {
        if(pTransfer->idx == key)
            return pTransfer->state_to;
```

```
    }

    return -1;
}
// 表驱动型、直接编码型的词法分析器的差异到这里结束
// }}}}

```