

# 第 12 章 编译器词法语法分析项目

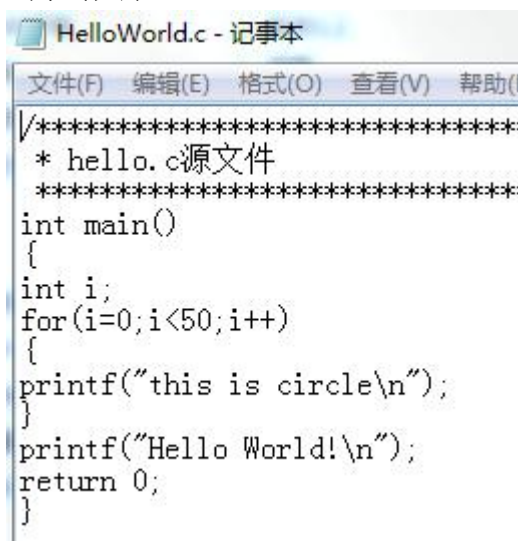
(视频讲解：2 小时)

本章我们将开始尝试做一个简单的编译器，编译一个程序，分为词法分析→语法分析→语义分析→中间代码生成→目标代码生成，因为我们没有学习汇编语言，不做编译器的后端（这个在工作中用的不多，目前 LLVM 非常流行，已经帮我们做好了后端工作，有兴趣的同学可以搜索 LLVM 进行学习），我们主要做词法分析，语法分析，王道训练营有位同学进入公司后，老大要求根据公司的编码规范写一个语法检查，针对不符合公司编程规范的进行提醒，这就需要我们掌握词法分析与语法分析。

## 12.1 词法分析项目

### 12.1.1 项目需求描述

该项目需求为分析一份 C 语言程序的内容，将函数名变量名设置为灰色，将关键字设置为绿色，将整型常量，浮点型常量，字符常量，字符串常量等常量设置为褐色，将运算符设置为红色，如图 12.1.1-1 所示，HelloWorld.c 是一个 txt 文档，测试程序



```
/* **** */
* hello.c源文件
* **** */
int main()
{
    int i;
    for(i=0;i<50;i++)
    {
        printf("this is circle\n");
    }
    printf("Hello World!\n");
    return 0;
}
```

图 12.1.1-1

经过编译器词法分析后，得到如图 12.1.1-2 效果。



图 12.1.1-2

注意：每行的结束符分号，也作为运算符，显示红色，书籍为黑白印刷，如果需要上图的彩色图，可以加入前言QQ群，进行获取。

## 12.2 词法分析模块设计

### 12.2.1 建立字典模块

词法分析是编译过程的第一阶段。它的任务就是对输入的字符串形式的源程序按顺序进行扫描，根据源程序的词法规则识别具有独立意义的单词（符号），并输出与其等价的TOKEN序列。

首先通过枚举对所有运算符及关键字进行编号

```
/* 单词编码 */
enum e_TokenCode
{
    /* 运算符及分隔符 */
    TK_PLUS,      // + 加号
    TK_MINUS,     // - 减号
    TK_STAR,      // * 星号
    TK_DIVIDE,    // / 除号
    TK_MOD,       // % 求余运算符
    TK_EQ,        // == 等于号
    TK_NEQ,       // != 不等于号
    TK_LT,        // < 小于号
    TK_LEQ,       // <= 小于等于号
    TK_GT,        // > 大于号
    TK_GEQ,       // >= 大于等于号
    TK_ASSIGN,    // = 赋值运算符
    TK_POINTSTO,  // -> 指向结构体成员运算符
    TK_DOT,       // . 结构体成员运算符
    TK_AND,       // & 地址与运算符
    TK_OPENPA,    // ( 左圆括号
```

```

TK_CLOSEPA,      // ) 右圆括号
TK_OPENBR,       // [ 左中括号
TK_CLOSEBR,      // ] 右圆括号
TK_BEGIN,        // { 左大括号
TK_END,          // } 右大括号
TK_SEMICOLON,    // ; 分号
TK_COMMA,        // , 逗号
TK_ELLIPSIS,     // ... 省略号
TK_EOF,          // 文件结束符

/* 常量 */
TK_CINT,         // 整型常量
TK_CCHAR,        // 字符常量
TK_CSTR,         // 字符串常量

/* 关键字 */
KW_CHAR,         // char关键字
KW_SHORT,        // short关键字
KW_INT,          // int关键字
KW_VOID,         // void关键字
KW_STRUCT,       // struct关键字
KW_IF,           // if关键字
KW_ELSE,         // else关键字
KW_FOR,          // for关键字
KW_CONTINUE,     // continue关键字
KW_BREAK,        // break关键字
KW_RETURN,       // return关键字
KW_SIZEOF,       // sizeof关键字

KW_ALIGN,        // __align关键字
KW_CDECL,        // __cdecl关键字 standard c call
KW_STDCALL,      // __stdcall关键字 pascal c call

/* 标识符 */
TK_IDENT //函数
};

```

（为了让项目不过于复杂，未考虑逻辑与，逻辑或，逻辑非等逻辑运算符，自增自减运算符不考虑，但原理完全一致）

编号以后，方便通过值进行判断，然后打印颜色，【例 12.2.1】中颜色打印 ch 是对应的字符串，也就是打印内容，token 是打印内容的 TOKEN 值，因为我们提前设置了 ch 对应字符串内容的 TOKEN 值，所以内容打印出后，会附带颜色。

【例 12.2.1】printColor 颜色打印函数

```

void printColor(char *ch, tokencode token) {
    HANDLE h = GetStdHandle(STD_OUTPUT_HANDLE);

```

```

    if (token >= TK_IDENT)
        SetConsoleTextAttribute(h, FOREGROUND_INTENSITY); //函数灰色
    else if (token >= KW_CHAR)
        SetConsoleTextAttribute(h, FOREGROUND_GREEN | FOREGROUND_INTENSITY); //
关键字绿色
    else if (token >= TK_CINT)
        SetConsoleTextAttribute(h, FOREGROUND_RED | FOREGROUND_GREEN); //常量褐色
    else
        SetConsoleTextAttribute(h, FOREGROUND_RED | FOREGROUND_INTENSITY); //运算
符号红色
    if (-1 == ch[0]) {
        printf("\n End_Of_File!");
        SetConsoleTextAttribute(h, FOREGROUND_RED | FOREGROUND_GREEN |
FOREGROUND_BLUE | FOREGROUND_INTENSITY);
    }
    else {
        printf("%s", ch);
    }
} //根据token值获取对应字符串类型，然后打印

```

为什么通过 token 做标识，因为对于编译时，源码文件中每一个都是字符或者字符串，全部作为字符串，读取后，将对应字符串存成一个链表（也可以采用动态字符串指针数组），链表的每一个节点同时存储对应字符串的 token 值，这样当打印输出时，可以通过上面的方法，判断 token 的值，从而设定不同的颜色，然后再次打印输出对应的字符串即可。

## 12.2.2 字符串存储及其 token 值的快速识别设计

为了能够存储每一个读到的字符串，可以采用动态数组的方式（也可以用链表的形式），当空间不够时，可以通过 realloc 来增加空间，动态指针数组定义方法如下：

```

/*动态数组定义*/
typedef struct DynArray
{
    int count;           // 动态数组元素个数
    int capacity;        // 动态数组缓冲区长度度
    void **data;         // 指向数据指针数组
} DynArray;

```

每一个指针指向的空间，存储一个结构体，这个结构体用来存储单词，这个只是说明一下动态字符串的设计，可以让 data 里的每一个指针直接指向实际的字符串。当我们从文件中读取一个字符串时，如何快速判断得到该字符串的 TOKEN 值呢？接下来我们要使用到哈希算法。

```

/* 单词存储结构定义 */
typedef struct TkWord
{
    int tkcode;           // 单词编码，也就是token值
    struct TkWord *next;  // 指向哈希冲突的其它单词
}

```

```

    char *spelling;                // 单词字符串
} TkWord;
TkWord* tk_hashtable[MAXKEY]; // 单词哈希表

```

当得到一次单词时，通过下面的哈希公式得到其哈希值，通过哈希值定位在单词哈希表中对应位置是否为NULL，如果不为NULL，说明该单词存在，直接取出tkcode，也就是token值，即可打印颜色。如果单词哈希表中为NULL，那么说明该字符串不在哈希表（也就是字符串指针数组）中，这时需要将其添加进去。

```

int elf_hash(char *key)
{
    int h = 0, g;
    while (*key)
    {
        h = (h << 4) + *key++;
        g = h & 0xf0000000;
        if (g)
            h ^= g >> 24;
        h &= ~g;
    }
    return h % MAXKEY;
}

```

如何判断出现哈希冲突，当得到字符串的哈希值后，同时要拿字符串与实际指针指向的字符串进行strcmp比较，如果发现不相等，说明发生哈希冲突，这时将冲突的字符串插入到下一个节点。

**放入哈希表时，我们如何知道放入的字符串的TOKEN值呢？**

首先我们需要定义一个如下的静态结构体数组：

```

static TkWord keywords[] = {
    {TK_PLUS, NULL, "+"},
    {TK_MINUS, NULL, "-"},
    {TK_STAR, NULL, "*"},
    {TK_DIVIDE, NULL, "/"},
    {TK_MOD, NULL, "%"},
    {TK_EQ, NULL, "=="},
    {TK_NEQ, NULL, "!="},
    {TK_LT, NULL, "<"},
    {TK_LEQ, NULL, "<="},
    {TK_GT, NULL, ">"},
    {TK_GEQ, NULL, ">="},
    {TK_ASSIGN, NULL, "="},
    {TK_POINTSTO, NULL, "->"},
    {TK_DOT, NULL, "."},
    {TK_AND, NULL, "&"},
    {TK_OPENPA, NULL, "("},
    {TK_CLOSEPA, NULL, ")"},
}

```

```

{TK_OPENBR, NULL, "["},
{TK_CLOSEBR, NULL, "}"},
{TK_BEGIN, NULL, "{"},
{TK_END, NULL, "}"},
{TK_SEMICOLON, NULL, ";"},
{TK_COMMA, NULL, ","},
{TK_ELLIPSIS, NULL, "..."},
{TK_EOF, NULL, "End_Of_File"},

{TK_CINT, NULL, "整型常量"},
{TK_CCHAR, NULL, "字符常量"},
{TK_CSTR, NULL, "字符串常量"},

{KW_CHAR, NULL, "char"},
{KW_SHORT, NULL, "short"},
{KW_INT, NULL, "int"},
{KW_VOID, NULL, "void"},
{KW_STRUCT, NULL, "struct"},

{KW_IF, NULL, "if"},
{KW_ELSE, NULL, "else"},
{KW_FOR, NULL, "for"},
{KW_CONTINUE, NULL, "continue"},
{KW_BREAK, NULL, "break"},
{KW_RETURN, NULL, "return"},
{KW_SIZEOF, NULL, "sizeof"},
{KW_ALIGN, NULL, "__align"},
{KW_CDECL, NULL, "__cdecl"},
{KW_STDCALL, NULL, "__stdcall"},
{0}
};

```

哈希表的建立，第一步是先遍历上面的结构体数组keywords，得到每一个字符串的哈希值时，将其存入哈希表，这样读取文件，当我们读到一个字符串时，如果是整型常量或者浮点常量，TOKEN值设置为TK\_CINT，如果是字符常量，TOKEN值设置为TK\_CCHAR，如果是字符串常量，TOKEN设置为TK\_CSTR，如果是函数名，我们设置其TOKEN为TK\_IDENT如果是变量名，我们可以设置其TOKEN值为44，因为上面的函数编号TK\_IDENT值为43。这样我们放入哈希表中的每个字符串都有TOKEN值，当我们从文件中读取到一串字符进行识别时，我们可以快速得到其TOKEN值，这样我们在打印输出时，就可以调用printColor函数，实现词法分析。

使用哈希表的目的是提高词法分析的速度，这样就提高了编译效率，针对整个代码文件，我们解析时，将每一个符号串用一个队列存储起来，可以用动态数组，也可以用链表实现，前言QQ群内给出了源码实现，使用的是链表，也就是用链表构造一个队列。

词法分析的源码如【例12.2.2-1】：

【例12.2.2-1】词法分析使用的源码试例

```
int main() {
```

```

int i;
for(i=0;i<=50;i=i+1) {
    printf("this is circle\n");
}
Insert(5,6);
printf("Hello World!\n");
return 0;
}

```

实际构造的队列结构如下：

名称	值
Text	0x0074a8b8 {tkcode=30 n
tkcode	30 1
next	0x0074ba68 {tkcode=43 n
tkcode	43 2
next	0x0074bb00 {tkcode=15 n
tkcode	15 3
next	0x0074bb98 {tkcode=16 n
spelling	0x0074bb50 "(" 3
row	1
spelling	0x0074bab8 "main" 2
row	1
spelling	0x0074b988 "int" 1
row	1

从上图中可以看出，第一个符号串是int，其TOKEN值为30，第二个符号串为main，其TOKEN值为43，第三个符号串为左括号（，其TOKEN值为15，当然代码实现和之前给的结构体TkWord有一些差异，多一个行号row，目的是为了在编译报错时，能够提示报错在第几行。

## 12.3 词法及语法分析简单样例研究

### 12.3.1 算术表达式的合法性判断

问题：给定一个字符串，只包含 +，-，\*，/，数字，小数点，（，）。要求判断该算术表达式是否合法。学过《编译原理》的人都知道，利用里面讲的分析方法可以对源代码进行解析。而**算术表达式**也是源代码的一部分，所以利用编译方法也可以很容易地判断表达式的合法性。与源代码相比，算术表达式只包含有很少的字符，所以解析起来也简单很多。下面从词法分析和语法分析两个方面来说明。

### 12.3.2 词法分析

下面先定一下表达式涉及到的单词的种类编码，见表 12.3.2-1：

单词符号	种类编码
+	1
-	2
*	3

/	4
数字	5
(	6
)	7

表 12.3.2-1

识别上表所列的单词的状态转换图见 12.3.2-1：

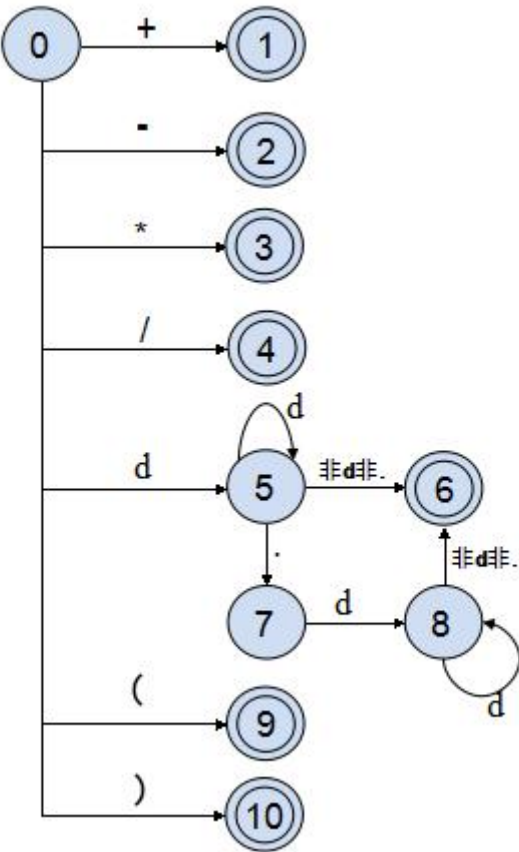


图 12.3.2-1

圆圈代表起始状态及中间状态，双层圆圈代表状态结束，状态结束后，就将对应字符串及其 TOKEN 值存入队列，【例 12.3.2-1】中函数 word\_analysis 根据上面的状态图，实现对算术表达式的词法分析，下面代码是用数组来表示队列，将算术表达式 char expr[50] = "(1.5+5.789)\*82-10/2+9"通过 word\_analysis 进行分析。

【例 12.3.2-1】算术表达式词法分析

```
typedef struct{
    char first[20]; //用来存储字符串
    int second; //用来存储字符串对应的TOKEN值
}Grammar_t;
```



```

Grammar_t word[100]={0};
int wordLen=0;

int word_analysis(Grammar_t* word, const char* expr)
{
    unsigned int i;
    for(i=0; i<strlen(expr); ++i)
    {
        // 如果是 + - * / ( )
        if(expr[i] == '(' || expr[i] == ')') || expr[i] == '+'
            || expr[i] == '-' || expr[i] == '*' || expr[i] == '/')
        {
            char tmp;
            tmp=expr[i];
            switch (expr[i])
            {
                case '+': //对应1结束状态
                    word[wordLen].first[0]=expr[i];
                    word[wordLen].second=1;
                    wordLen++;
                    break;
                case '-': //对应2结束状态
                    word[wordLen].first[0]=expr[i];
                    word[wordLen].second=2;
                    wordLen++;
                    break;
                case '*': //对应3结束状态
                    word[wordLen].first[0]=expr[i];
                    word[wordLen].second=3;
                    wordLen++;
                    break;
                case '/': //对应4结束状态
                    word[wordLen].first[0]=expr[i];
                    word[wordLen].second=4;
                    wordLen++;
                    break;
                case '(': //对应9结束状态
                    word[wordLen].first[0]=expr[i];
                    word[wordLen].second=6;
                    wordLen++;
                    break;
                case ')': //对应10结束状态
                    word[wordLen].first[0]=expr[i];
                    word[wordLen].second=7;
                    wordLen++;
            }
        }
    }
}

```

```

        break;
    }
}
// 如果是数字开头
else if(expr[i]>='0' && expr[i]<='9')
{
    char tmp[50]={0};
    int k=0;
    while(expr[i]>='0' && expr[i]<='9')//对应状态5
    {
        tmp[k++]=expr[i];
        ++i;
    }
    if(expr[i] == '.')//对应状态7
    {
        ++i;
        if(expr[i]>='0' && expr[i]<='9')
        {
            tmp[k++]='.';
            while(expr[i]>='0' && expr[i]<='9')//对应状态状态8
            {
                tmp[k++]=expr[i];
                ++i;
            }
        }
        else
        {
            return -1; // . 后面不是数字，词法错误，对应状态6
        }
    }
    strcpy(word[wordLen].first,tmp);
    word[wordLen].second=5;
    wordLen++;
    --i;
}
// 如果以. 开头
else
{
    return -1; // 以. 开头，词法错误，对应状态6
}
}
return 0;
}

```

对应算法表达式分析后，结果存在 word 结构体数组中，我们通过下面代码进行打印，

```

for(i=0;i<wordLen;i++)
{
    printf("%s %d\n",word[i].first,word[i].second);
}

```

最终得到的打印结果如图 12. 3. 2-2 所示，至此词法分析完毕：

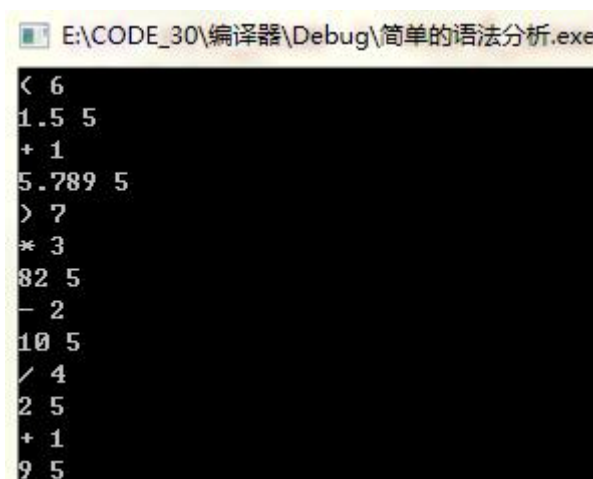


图 12. 3. 2-2

### 12.3.3 算术表达式的语法分析

语法分析(Syntax Analysis)是编译程序的第二阶段，也是编译程序的核心部分。语法分析的任务是，根据语言的语法规则，对源程序进行语法检查，并识别出相应的语法成分。语法分析的输入是从词法分析器输出的源程序的 Token 序列形式，然后根据语言的文法规则进行分析处理，语法分析输出是将有语法错误的地方进行打印提示，无语法错误的地方不提示，如果没有任何语法错误，提示语法分析通过（代表我们的程序编译通过，可以转为汇编代码）。

针对语法分析，首先我们需要对表达式进行文法分析，算术表达式的文法  $G[E]$  如下：

```

E → E+T | E-T | T
T → T*F | T/F | F
F → (E) | d

```

如何得出上面的文法  $G[E]$ ，通俗易懂的方法是我们将每一类运算符进行优先级分级，同一优先级的看为一个整体，另外要有递归的思想，一个表达式可能是  $2+3$ ，也可能是  $2+3+4$ ，但是对于文法  $E \rightarrow E+T$  来讲，他们是等价的，只是分析时多循环一次而已，针对  $|$  代表是或的意思，也就是  $E$  可能是  $E+T$ ，可能是  $E-T$ ，也可能是  $T$ ，对于最后的  $F \rightarrow (E) | d$ ，因为括号是最高优先级，但是括号内又可能是一个表达式  $E$ ，所以这是一个多函数递归的原理，当然  $F$  也可能是一个数字。

有  $G[E]$  后，如何进行语法分析的程序实现呢，这时有两种方法，一种是 **LL(1) 文法**，LL 分析器是一种处理某些上下文无关文法的自顶向下分析器。因为它从左（Left）到右处理输

入，再对句型执行最左推导出语法树（Leftderivation，相对于 LR 分析器）。能以此方法分析的文法称为 LL 文法。

一个 LL 分析器若被称为 LL(k) 分析器，表示它使用 k 个词法单元作向前探查。对于某个文法，若存在一个分析器可以在不用回溯法进行回溯的情况下处理该文法，则称该文法为 LL(k) 文法。这些文法中，较严格的 LL(1) 文法相当受欢迎，因为它的分析器只需多看一个词法单元就可以产生分析结果。那些需要很大的 k 才能产生分析结果的编程语言，在分析时的要求也比较高。

消去非终结符 E 和 T 的左递归后，改写 G[E] 文法如下：

```
E → TE'  
E' → +TE' | -TE' | ε  
T → FT'  
T' → *FT' | /FT' | ε  
F → (E) | d
```

可以证明上述无递归文法是 LL(1) 文法，它的基本思想是：对文法中的每一个非终结符编写一个函数（或子程序），每个函数（或子程序）的功能是识别由该非终结符所表示的语法成分。

构造递归下降分析程序时，每个函数名是相应的非终结符，函数体是根据规则右部符号串的结构编写：

1、当遇到终结符 a 时，则编写语句

if（当前读来的输入符号 == a）读下一个输入符号；

2、当遇到非终结符 A 时，则编写语句调用 A（）；

3、当遇到  $A \rightarrow \epsilon$  规则时，则编写语句

if（当前读来的输入符号 不属于 FOLLOW(A)）error（）；

当某个非终结符的规则有多个候选式时，按 LL(1) 文法的条件能唯一地选择一个候选式进行推导。

由于 LL(1) 文法较难理解，因此我们针对上面的代码实现，在书中不再给出，有兴趣的同学可以加入前言 QQ 群，获取代码进行理解，结合代码才能更清晰理解 LL(1) 文法。

另外一种非常流行的方法是递归下降法，递归下降法是语法分析中最易懂的一种方法。不少手写编译器的书籍均采用这种方法，它的主要原理是，对每个非终极符按其产生式结构构造相应语法分析子程序，其中终极符产生匹配命令，而非终极符则产生过程调用命令。因为文法递归相应子程序也递归，所以称这种方法为递归子程序下降法或递归下降法。其中子程序的结构与产生式结构几乎是一致的，从而使编写程序也变的更加简单。对 G[E] 通过递归下降法进行改写，相当于去除非终极符。

```

E → T { +T | -T }
T → F { *F | /F }
F → (E) | d

```

然后通过上面的公式直接生成如【例 13.3.3-1】的语法分析代码。

【例 13.3.3-1】算术表达式语法分析源码

```

#include<stdio.h>
#include<stdlib.h>
#include<string.h>

typedef struct{
    char first[20]; //用来存储符号串
    int second; //用来存储符号串对应的TOKEN值
}Grammar_t;

char expr[50] = "(1.5+5.789)*82-10/2+9"; //需要进行语法分析的算术表达式
Grammar_t word[100]={0}; //用于存储每个符号串及其TOKEN值
int wordLen=0; //用于存储最终word里有多少符号串
int idx = 0; //代表到达了第几个符号串，数组下标从零开始，
int sym; //存储当前符号串的TOKEN值
int err = 0; //错误标识

void T();
void F();
//词法分析, 上面的代码已经书写，这里只进行一个声明
int word_analysis(Grammar_t* word, const char* expr);
//执行Next的作用就是读取下一个符号串，word中每个符号串挨个存储
void Next()
{
    if(idx < wordLen)
        sym = word[idx++].second;
    else
        sym = 0;
}
// 上面的递归下降文法表达式 E → T { +T | -T }
//首先调用函数T，针对大括号内的+T或者-T，我们这里一定要用while，因为有可能E是
T+T+T，比如3*4+5*6+7/8
void E()
{
    T();
    while(sym == 1 || sym == 2)
    {
        Next();
        T();
    }
}

```

```
}
```

$T \rightarrow F \{ *F \mid /F \}$

// T有可能只有F，也有可能是F\*F，或者F\*F\*F等，所以和上面加法和减法类似，同样采用循环，符号是3或者4，也就是乘号，或者除号，就Next读取下一个符号串，然后再次调用F

```
void T()
{
    F();
    while(sym == 3 || sym == 4)
    {
        Next();
        F();
    }
}
```

$F \rightarrow (E) \mid d$

//代码实现中，我们先判断的是是否是数字，也就是sym是否为5，当然也可以先判断是否是左括号，当是左括号时，在括号内括住的是一个的表达式，这时我们重新调用E，前面我们学习过递归，这里是多函数形成的一种递归，如果是一个表达式，这时解析完毕后，就需要判断是否有右括号，如果不是右括号，这时就报错，同时如果既不是数字又不是左括号，也需要报错。

```
void F()
{
    if (sym == 5)
    {
        Next();
    }
    else if(sym == 6)
    {
        Next();
        E();
        if (sym == 7)
        {
            Next();
        }
        else
        {
            err = -1;
        }
    }
    else
    {
        err = -1;
        puts("Wrong Expression.");
        system("pause");
    }
}
```

```

}

int main()
{
    int i;
    //对表达式进行词法分析
    int err_num = word_analysis(word, expr);
    if (-1 == err_num)
    {
        puts("Word Error!");
    }
    else
    {
        //词法分析成功，测试输出
        for(i=0;i<wordLen;i++)
        {
            printf("%s %d\n",word[i].first,word[i].second);
        }
        // 词法正确，进行语法分析
        Next();
        E();//开始语法分析
        if (sym == 0 && err == 0) // 注意要判断两个条件
            puts("Right Expression.");
        else
            puts("Wrong Expression.");
    }
    system("pause");
    return 0;
}

```

执行结果如图 12.3.3-1:

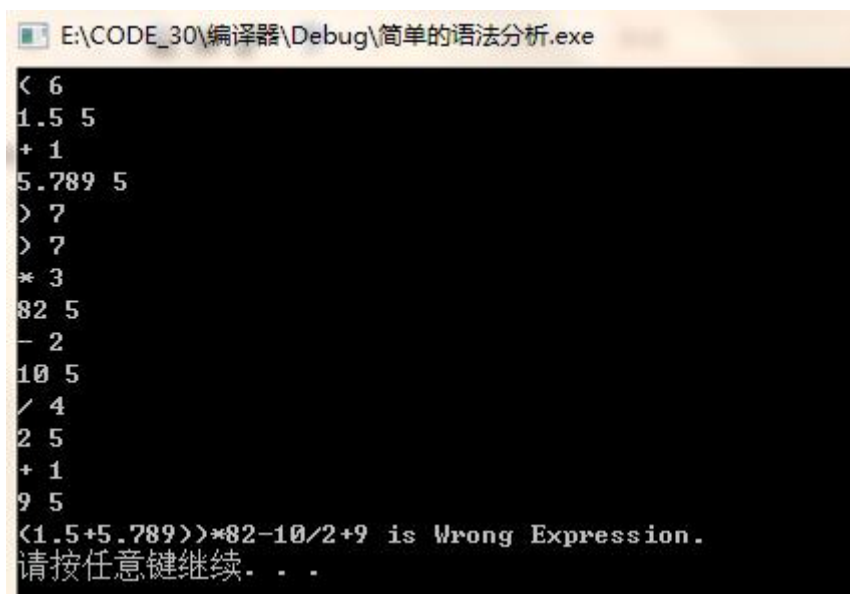
```

E:\CODE_30\编译器\Debug\简单的语法分析.exe
< 6
1.5 5
+ 1
5.789 5
> 7
* 3
82 5
- 2
10 5
/ 4
2 5
+ 1
9 5
<1.5+5.789>*82-10/2+9 is Right Expression.
请按任意键继续. . .

```

图 12.3.3-1

假如我们将表达式改为 `char expr[50] = "(1.5+5.789))*82-10/2+9"`，这时再次执行程序，得到如图 12.3.3-2 的执行结果，可以看出，表达式语法分析失败，输出 Wrong Expression，



```
E:\CODE_30\编译器\Debug\简单的语法分析.exe
< 6
1.5 5
+ 1
5.789 5
> 7
> 7
* 3
82 5
- 2
10 5
/ 4
2 5
+ 1
9 5
(1.5+5.789))*82-10/2+9 is Wrong Expression.
请按任意键继续. . .
```

图 12.3.3-2

思考题：失败时我们可以打印编译失败符号串的位置，请问如何修改程序进行打印？

## 12.4 升级版功能，编译器语法分析

常用的语法分析方法有自顶向下分析和自底向上分析两大类。自顶向下分析又分为确定的和不确定的两种，这里采用的是确定的自顶向下分析方法，也就是不带回溯的那种。确定的自顶向下分析方法又有两种，一种是递归子程序法，另一种是预测分析法

这里采用的语法分析方法是 自顶向下的 确定的 递归子程序法，即递归下降法

词法分析是语法分析的基础，有了词法分析，我们可以得到了每一个符号串极其 TOKEN 值，把每一部分都分清楚，语法分析作用就是分析代码是否有编译错误，编译问题比较复杂，我们要分场景，同时每一个单独的功能要封装好，这样才能重复调用，下面来依次分析。

### 12.4.1 整体流程分析

代码最外层分析：

重点：

第一：是不是函数声明



第二：是不是函数实现

第三：是不是外部声明 `int i, j, k` 等，或者外部声明初始化 `int i=3`

整体流程图如图 12.4.1-1：

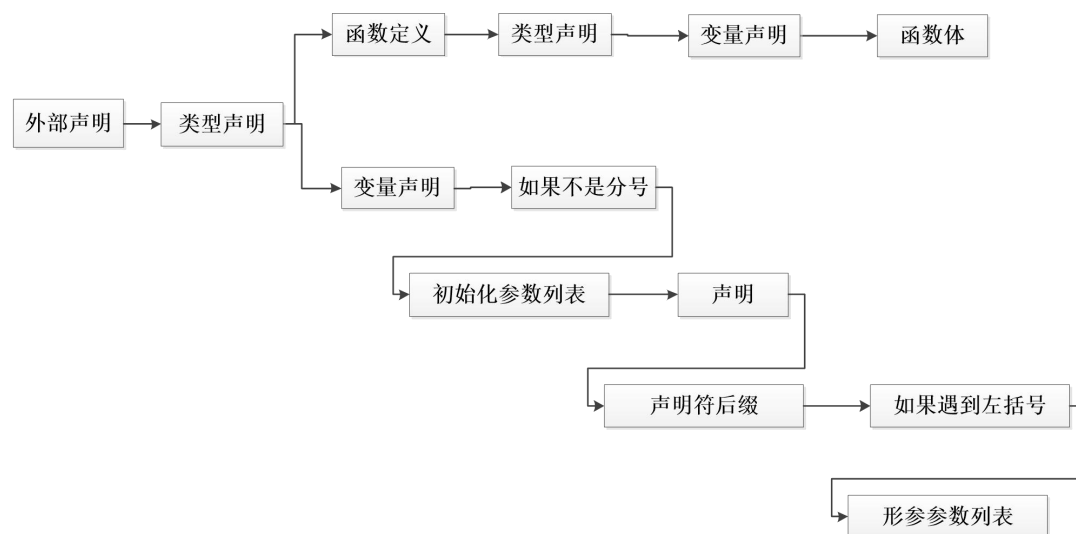


图 12.4.1-1

可以把代码都看为外部声明，最初就是判断对应符号串的TOKEN是不是KW\_CHAR与KW\_STRUCT这些类型说明符之间，然后怎么区分函数与外部变量声明呢？判断类型声明后，我们判断下下个符号串是不是左括号（因为下一个是函数名），就可以进行跳转。

如果跳转到变量声明，首先要看后面是不是分号，如果是分号，那么是多个变量声明，不是一个，就需要调用初始化参数列表，多个变量声明，每一个都需要调用声明，然后针对声明符后缀，如果定义的变量是一个函数指针，那么就会遇到左括号，这时又是形参参数列表，当然针对函数指针可以不写形参名。

针对函数定义，因为函数的内部依然是一个一个的变量声明，所以这时调用的与上面的变量声明的过程是类似的，继续调用对应的函数即可。

## 12.4.2 函数体内流程分析

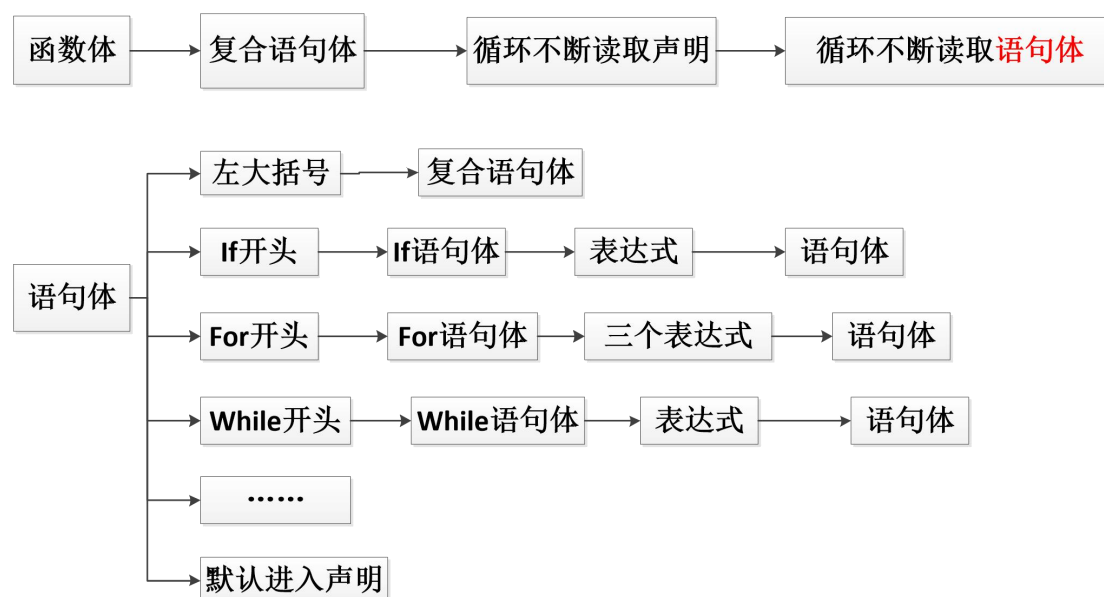


图12.4.2-1

通过图 12.4.1-1，进行函数体内部分析：

重点：

首先是大括号，然后是当读到的如果是int，char等标识符说明是一个声明，不断读取声明，如果没有声明了，开始匹配是不是语句体，语句体就可以通过switch分情况，判断是左大括号，就是复合语句，调用上面的复合语句体函数，是if，先判断if语句体，就是左括号，中间是表达式，针对表达式后面单独画图解析。是for，就是for语句，做括号，然后三个表达式，右括号，接着回归到语句体。是return，就是return语句，是break，就是break语句，是continue，就是continue语句，都不是，就使用switch的default手法，接着再次进入声明，如果没有声明，就是语句体结束了。

读取得到if关键字，然后判断是否是左括号，然后判断是否是表达式，调用语句体函数进行语句解析，然后如果读到else，再次调用语句体函数进行语句解析

首先读取for关键字，然后判断是否是左括号，然后判断是否是分号，如果是，说明第一个表达式缺失，如果不是，调用表达式解析函数解析表达式，接着跳过一个分号，再次判断下一个是否是分号，如果是，说明第二个表达式缺失，如果不是，调用表达式解析函数解析表达式，解析后跳过一个分号，然后判断是否等于右括号，不是右括号，调用表达式解析函数解析第三个表达式，然后跳过右括号。接着使用语句体函数解析语句部分。

针对continue语句，下一个判断是否是分号，不是就报错；

针对break语句，判断是否是分号，不是就报错（这就是跳过分号）；

针对return语句，判断是否是分号，是分号，直接返回，如果不是分号，说明有表达式，调用表达式解析函数，然后再跳过分号

### 12.4.3 表达式解析流程分析

首先我们编写一下表达式解析的文法分析，每种符号含义如表12.4.3-1：

缩写	英文函数名称	汉语含义
E	Expression	表达式函数
A	Assignment_expression	赋值表达式函数
EQ	Equality_espression	相等类表达式函数
RE	Relational_expression	关系类表达式函数
AD	Additive_expression	加减类表达式
MU	Multiplicative_expression	乘除类表达式
UN	Unary_expression	一元表达式（&, *, -, sizeof等）
PO	Postfix_expression	后缀表达式（主要是[]和（））
PR	Primary_expression	初等表达式

表 12. 4. 3-1

上面的英文函数名大家在编写编译器时可以参考，后面汉语解释，使用缩写的目的是下面的文法分析表达式编写方便，相等类表达式是指==是否等于和 !=是否不等于，因为这两个运算符的优先级低于小于（<），大于（>），小于等于（<=），大于等于（>=）等运算符的优先级。

```
E → A { , A }
A → EQ { = EQ }
```

```

EQ → RE { ==RE | !=RE }
RE → AD { =AD | >AD | <=AD | >=AD }
AD → MU { +MU | -MU }
MU → UN { *UN | /UN | %UN }
UN → &UN | *UN | -UN | sizeof UN | PR
PR → (E) | [E]

```

#### 功能:解析表达式语句

判断是否是分号，如果不是，说明有表达式，调用expression，然后跳过分号。例如if（表达式）；这种情况

#### 功能:解析相等类表达式

判断是不是关系类表达式，如果不是判断token是否为!=或者==，如果是再次读取内容，调用关系类表达式

#### 功能:解析关系表达式

判断是不是加减类表达式，如果不是判断token是否为<, 或者<=, >, >=, 如果是再次读取内容，调用加减类表达式

#### 功能:解析加减类表达式

判断是否为乘除类表达式，如果不是判断token是否为+, -, 如果是，再次读取内容，调用乘除类表达式

#### 功能:解析乘除类表达式

判断是否是一元表达式，如果不是判断token是否为\*, /, %, 如果是，再次读取内容，调用一元表达式

#### 功能:解析一元表达式

判断是否是&, \*, -, sizeof等运算符，如果是，继续递归调用自身，如果都不是，就是后缀表达式，调用postfix\_expression

在判断时注意对一个规则进行校验：

- 1、左右括号必须匹配(可用栈来实现)
- 2、操作符不能跟着操作符
- 3、第一个字符不能是操作符(-是负号可以)

## 12.4.4 总结

通过 12.4.1 到 12.4.3 的分析，相信大家对于实现语法分析有了一个清晰的思路，首先编写整体流程，然后是函数体内的流程分析，接着是表达式的语法分析，注意一定要采用增量编写法，比如一开始进行语法分析的文件，就只有外部声明，函数声明，没有函数实现，测试是否可以语法分析通过，然后再把源代码文件改错，再次执行，看是否能够针对错误的地方，提示语法分析错误。接着编写函数体分析的代码，编写好后，增加一个函数实现，验证是否提示编译通过，最后再增加表达式分析的代码，和前面一样，首先测试编译通过，然后自行制造各种错误，看是否能够正确提示。

针对编译错误提示，不用达到目前微软的 VS 里的编译器那么智能，只需提示第几行编译错误，在什么符号串位置出错即可，报错后，不想采用让函数 return 返回，想直接跳转到出错提示函数，可以采用 setjmp 与 longjmp 设计，或者在出错提示函数中使用 exit，让可执行程序结束。

做项目过程中有任何疑问，要及时与老师沟通，交流，避免在不清楚需求的情况下，闭门造车，同时注意项目一定是在设计清晰的情况下，再去编写代码，切记设计不清晰的情况下编写。

采用敏捷开发模式，比如表达式，一开始没有设计所有的运算符类型，不需要过于担心，逐步加进去即可，注意函数的封装。