

## 第 12 章 编译器词法、语法分析项目

(视频讲解: 2 小时)

本章尝试开发一个简单的编译器。编译一个程序的步骤如下: 词法分析→语法分析→语义分析→中间代码生成→目标代码生成。由于我们没有学习过汇编语言, 无法做编译器的后端工作(这在实际工作中用得不多, 目前非常流行的 LLVM 框架通常已帮我们做好了后端工作, 有兴趣的读者可以搜索 LLVM 自主学习), 所以我们主要做词法分析、语法分析。本章的难度较大, 读者可以结合视频理解项目需求。

### 12.1 词法分析项目

#### 12.1.1 项目需求描述

该项目需求是分析一个 C 语言程序的内容, 将函数的变量名设置为灰色, 将关键字设置为绿色, 将整型常量、浮点型常量、字符常量、字符串常量等设置为褐色, 将运算符设置为红色。如图 12.1.1 所示, HelloWorld.c 是一个文本文档, 用来测试程序。

经过编译器词法分析后, 得到的效果如图 12.1.2 所示。

```

/*****
 * 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 待分析的源文件

```

int main()
{
    int i;
    for(i=0;i<50;i++)
    {
        printf("this is circle\n");
    }
    printf("Hello World!\n");
    return 0;
}
```

图 12.1.2 词法分析后的效果

**注意:** 每行代码的结束符是分号, 它也作为运算符, 需要将其设置为红色。书籍为黑白印刷, 要查看彩色图, 可在 QQ 群中获取。

### 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，具体编码如下所示。

```
/* 具体编码 */
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,  // [ 左方括号
    TK_CLOSEBR, // ] 右方括号
    TK_BEGIN,   // { 左花括号
    TK_END,     // } 右花括号
    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
};

```

注意，为使得项目不过于复杂，未考虑逻辑与、逻辑或、逻辑非等逻辑运算符，自增和自减运算符也不考虑，但原理完全一致。

在例 12.2.1 中，颜色打印函数 `printColor` 中的“`ch`”是对应的字符串，即打印内容；`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，所以内容打印后会附带颜色。

【例 12.2.1】颜色打印函数 `printColor`。

```

void printColor(char *ch, tokencode token) {
    HANDLE h = GetStdHandle(STD_OUTPUT_HANDLE);
    if (token >= TK_IDENT)
        SetConsoleTextAttribute(h, FOREGROUND_INTENSITY);
        //函数显示灰色
    else if (token >= KW_CHAR)
        SetConsoleTextAttribute(h, FOREGROUND_GREEN
FOREGROUND_INTENSITY);
        //关键字显示为绿色
    else if (token >= TK_CINT)
        SetConsoleTextAttribute(h, FOREGROUND_RED | FOREGROUND_GREEN);
        //常量显示为褐色
    else

```

```

        SetConsoleTextAttribute(h, FOREGROUND_RED | FOREGROUND_INTENSITY);
        //运算符显示为红色
    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

```

为什么要通过 *token* 做标识？这是因为编译时，源代码文件中的每个数据都是字符或字符串，全部作为字符串读取后，将对应字符串存储在一个链表中（也可以采用动态字符串指针数组），链表的每个结点同时存储对应字符串的 *TOKEN* 值，这样，在打印输出时，可以通过上面的方法判断 *TOKEN* 的值，从而设定不同的颜色来显示打印内容，然后打印输出对应的字符串。

### 12.2.2 字符串存储及其 *TOKEN* 值的快速识别设计

为了能够存储每个读到的字符串，可以采用动态数组的形式（也可以采用链表的形式），当空间不够时，可以通过 *realloc* 函数来增加空间。动态数组的定义方法如下：

```

/*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
    void **data;      // 指向数据指针数组
} DynArray;

```

每个指针指向的空间都存储一个结构体，这个结构体是用来存储单词的，这里只说明一下动态字符串的设计，可以让 *data* 中的每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，如何快速判断并得到该字符串的 *TOKEN* 值？可以通过哈希算法来实现，单词存储结构定义如下：

```

/* 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;       // 指向哈希冲突的其他单词
    char *spelling;            // 单词字符串
} TkWord;

```

```
TkWord* tk_hashtable[MAXKEY]; // 单词哈希表
```

得到一个单词后,通过下面的哈希公式得到其哈希值,通过哈希值在单词哈希表中定位并查看对应位置是否为 *NULL*,如果不为 *NULL*,那么说明该单词存在,直接取出 *tkcode*(即 *TOKEN* 值)。如果在单词哈希表中为 *NULL*,那么说明该字符串不在哈希表(即字符串指针数组)中,那么这时需要将其添加进去。具体代码如下所示:

```
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;
}
```

那么如何判断是否出现哈希冲突呢?得到字符串的哈希值后,要同时将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,那么说明发生哈希冲突,这时要将冲突的字符串插入下一个结点。

将字符串添加到哈希表中时,如何知道添加的字符串的 *TOKEN* 值呢?

首先,我们需要定义一个如下所示的静态结构体数组:

```
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}
};

```

建立哈希表时，第一步是先遍历上面的结构体数组 *keywords*，得到每个字符串的哈希值，并将其存入哈希表。这样，当我们读到一个字符串时，如果是整型常量或浮点常量，那么 *TOKEN* 值设置为 *TK\_CINT*；如果是字符常量，那么 *TOKEN* 值设置为 *TK\_CCHAR*；如果是字符串常

量，那么 *TOKEN* 设置为 *TK\_CSTR*；如果是函数名，那么 *TOKEN* 值设置为 *TK\_IDENT*；如果是变量名，那么 *TOKEN* 值设置为 *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就都有了 *TOKEN* 值，当我们从文件中读取一串字符进行识别时，可以快速得到其 *TOKEN* 值，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，进而提高编译效率。针对整个代码文件，我们在解析时，需要用一个队列存储每个符号串，可以用动态数组或链表实现。QQ 群内给出的源代码实现使用的是链表，即用链表构造一个队列。

词法分析的源代码示例如例 12.2.2 所示。

【例 12.2.2】词法分析使用的源代码示例。

```
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图 12.2.1 所示。

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

图 12.2.1 队列结构

从图 12.2.1 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，代码实现和之前给出的结构体 *TkWord* 有一些差异（多了一个行号 *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

给定一个字符串，其中只包含+、-、\*、/、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”课程的读者都知道，利用“编译原理”课程中介绍的分析方法可以对源代码进行解析。算术表达式也是源代码的一部分，所以利用编译方法也能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，所以解析起来要简单得多。下面从词法分析和语法分析两个方面加以说明。

### 12.3.2 词法分析

下面先确定一些表达式涉及的常见单词的种类编码，如表 12.3.1 所示。

表 12.3.1 常见单词的种类编码

单词符号	种类编码
+	1
-	2
*	3
/	4
数字	5
(	6
)	7

识别上表所列的单词的状态转换图如图 12.3.1 所示。

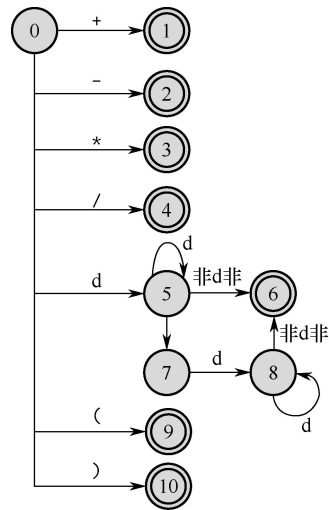


图 12.3.1 状态转换图

图中单圆圈代表起始状态及中间状态，双圆圈代表状态结束。状态结束后，就将对应字符串





```
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 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;
}

```

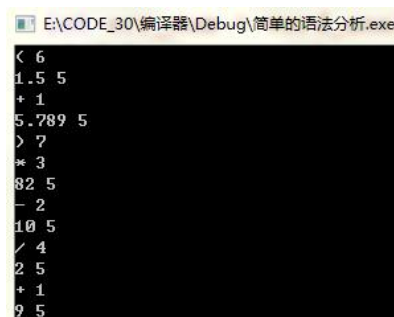
分析对应的算法表达式后，结果会存储在 `word` 结构体数组中，可以通过如下代码进行打印：

```

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 所示。至此，词法分析完毕。



```

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

```

图 12.3.2 词法分析效果图

### 12.3.3 算术表达式的语法分析

语法分析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，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，那么提示语法分析通过（代表程序编译通过，可以转为汇编代码）。

要进行语法分析，首先需要对表达式进行文法分析。算术表达式的文法  $G[E]$  如下：

$$\begin{aligned} E &\rightarrow E+T \mid E-T \mid T \\ T &\rightarrow T*F \mid T/F \mid F \\ F &\rightarrow (E) \mid d \end{aligned}$$

那么如何得出上面的文法  $G[E]$  呢？通俗易懂的方法是将每一类运算符进行优先级分级，同一优先级的视为一个整体。另外，要借助递归的思想，因为一个表达式可能是  $2+3$ ，也可能是  $2+3+4$ ，但是对于文法  $E \rightarrow E+T$  来讲，它们是等价的，只是分析时多循环一次而已，“ $|$ ”代表或，即  $E$  可能是  $E+T$ ，可能是  $E-T$ ，也可能是  $T$ 。对于最后的  $F \rightarrow (E)d$ ，因为括号的优先级是最高，但括号内又可能是一个表达式  $E$ ，所以这是一个多函数递归。当然， $F$  也可能是一个数字。

得到  $G[E]$  后如何进行语法分析的程序实现呢？这里有两种方法。

一种方法是  $LL(1)$  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。因为它先从左（*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，则表示它使用  $k$  个词法单元做向前探查。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。在这些文法中，较严格的  $LL(1)$  文法非常受欢迎，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  $k$  值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：

$$\begin{aligned} E &\rightarrow TE' \\ E' &\rightarrow +TE' \mid -TE' \mid \varepsilon \\ T &\rightarrow FT' \\ T' &\rightarrow *FT' \mid /FT' \mid \varepsilon \\ F &\rightarrow (E) \mid d \end{aligned}$$

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的。

（1）遇到终结符  $a$  时，编写语句

*if*（当前读来的输入符号 ==  $a$ ） 读下一个输入符号；

(2) 遇到非终结符  $A$  时, 编写语句调用  $A()$ 。

(3) 遇到  $A \rightarrow \varepsilon$  规则时, 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, 因此这里再给出针对上面的代码实现, 有兴趣的读者可以在 QQ 群中获取代码进行理解, 只有结合代码才能更清晰地理解  $LL(1)$  文法。

另一种非常流行的语法分析方法是递归下降法。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, 它的原理是, 对每个非终结符按其产生式结构构造相应语法分析子程序, 其中终结符产生匹配命令, 而非终结符产生过程调用命令。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得更加简单。对  $G[E]$  通过递归下降法进行改写, 相当于去除非终结符:

```
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 13.3.2 所示的语法分析源代码。

【例 13.3.2】算术表达式语法分析源代码。

```
#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 中有多少符号串
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  $E \rightarrow T \{ +T \mid -T \}$ 
// 首先调用函数 T, 针对花括号内的+T或-T, 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 有可能只是 F, 也可能是  $F*F$  或  $F*F*F$  等, 所以和上面的加法和减法类似, 同样采用循环,
// 符号是 3 或 4, 也就是乘号或除号, 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, 先判断是否是数字, 即 sym 是否为 5。当然, 也可以先判断是否是左括号, 是左括
//号时, 在括号内括住的是一个新表达式, 这时重新调用 E。前面学习过递归, 这里是多函数形成
//的一种递归形式。如果是一个表达式, 那么这时解析完毕后, 就需要判断是否是右括号, 如果不
//是右括号, 那么会报错, 同时如果既不是数字又不是左括号, 那么也需要报错。
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 12.3.2 所示。



图 12.3.2 执行结果

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

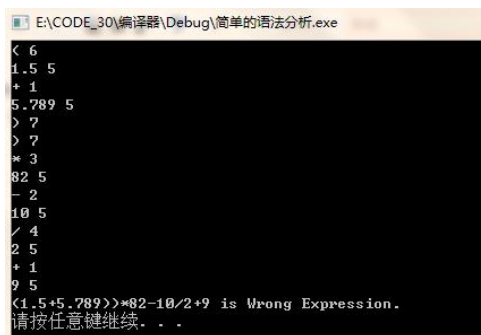


图 12.3.3 修改代码后的执行结果



**思考题：**语法分析失败时我们其实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方法和自底向上分析方法两大类。

自顶向下分析方法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，也就是不带回溯的分析方法。

确定的自顶向下分析方法也有两种，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，就可以得到每个符号串及其 *TOKEN* 值，把每一部分都划分清楚。语法分析的作用就是分析代码是否有编译错误。编译问题比较复杂，我们要分场景进行分析，同时要封装好每个单独的功能，以便重复调用。下面依次加以分析。

### 12.4.1 整体流程分析

代码最外层分析的内容如下：

- (1) 是不是函数声明？
- (2) 是不是函数实现？
- (3) 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 所示。

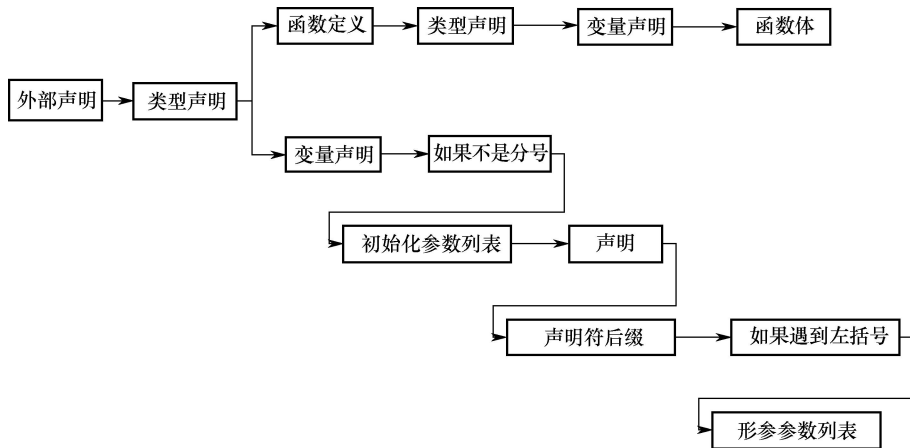


图 12.4.1 整体流程图

可以把代码都视为外部声明，最初判断对应符号串的 *TOKEN* 是否位于 *KW\_CHAR* 与 *KW\_STRUCT* 这些类型说明符之间。那么怎么区分函数与外部变量声明呢？判断类型声明后，判断下一个符号串是不是左括号（因为下一个是函数名），进而进行跳转。

如果跳转到变量声明，那么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而不是一个，此时需要调用初始化参数列表。多个变量声明即每个都需要调用声明，对于声明符后

缀，如果定义的变量是一个函数指针，那么就会遇到左括号，这时就又是形参参数列表，当然函数指针可以不写形参名。

对于函数定义，因为函数的内部依然是一个一个的变量声明，所以这时调用的过程与上面的变量声明过程是类似的，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 所示为函数体内的分析流程，其重点如下。

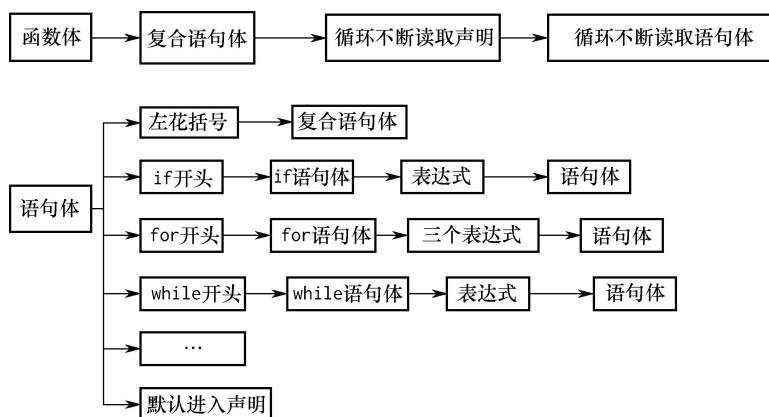


图 12.4.2 函数体内的分析流程

首先是花括号，读到的数据是 *int*、*char* 等标识符时，说明是一个声明，接着不断地读取声明，如果不再有声明，那么就开始匹配语句体，语句体可以通过 *switch* 分情况匹配：

- 判断是左花括号，就是复合语句，调用上面的复合语句体函数。
- 判断是 *if*，先判断 *if* 语句体，就是左括号。中间是表达式，针对表达式后面的内容单独画图解析。
- 判断是 *for*，就是 *for* 语句，先是左括号，后是三个表达式，再是右括号，接着回归到语句体。
- 判断是 *return*，就是 *return* 语句。
- 判断是 *break*，就是 *break* 语句。
- 判断是 *continue*，就是 *continue* 语句。
- 如果都不是，就使用 *switch* 的 *default* 方式，接着再次进入声明，如果没有声明，就是语句体结束。

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

若首先读取 *for* 关键字，则接着判断是否是左括号，然后判断是否是分号，如果是，那么说明第一个表达式缺失，如果不是，那么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，那么说明第二个表达式缺失，如果不是，那么调用表达式解

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，如果不是右括号，那么调用表达式解析函数解析第三个表达式，再后跳过右括号，接着使用语句体函数解析语句部分。

对于 *continue* 语句，判断是否是分号，不是就报错。

对于 *break* 语句，判断是否是分号，不是就报错（即跳过分号）。

对于 *return* 语句，判断是否是分号，是分号就直接返回，不是分号则说明有表达式，调用表达式解析函数，然后跳过分号。

12.4.3 表达式解析流程分析

首先我们编写表达式解析的文法分析，各种符号的含义如表 12.4.1 所示。

表 12.4.1 各种符号的含义

缩 写	英文函数名称	含 义
E	Expression	表达式函数
A	Assignment_expression	赋值表达式函数
EQ	Equality_expression	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

读者在编写编译器时可以参考表中的英文函数名。使用缩写的目的是方便下面的文法分析表达式的编写。相等类表达式是指等于（==）和不等（!=），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，那么说明有表达式，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。

判断是否是关系类表达式。如果不是，那么判断 *token* 是否为 !=、==；如果是，那么就再次读取内容，调用关系类表达式。

功能：解析关系表达式。

判断是否是加减类表达式。如果不是，那么判断 *token* 是否为 <或<=、>、>=；如果是，那么再次读取内容，调用加减类表达式。

功能：解析加减类表达式。

判断是否是乘除类表达式。如果不是，那么判断 *token* 是否为+、-；如果是，那么再次读取内容，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，那么判断 *token* 是否为\*、/、%；如果是，那么再次读取内容，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，那么就是后缀表达式，调用 *postfix\_expression*。

在判断时注意要对一些规则进行校验：

- (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（可以是负号）。

#### 12.4.4 总结

通过 12.4.1 节到 12.4.3 节的分析，相信读者对实现语法分析有了较为清晰的思路：首先编写整体流程，然后进行函数体内的流程分析，接着进行表达式的语法分析。注意，一定要采用增量编写法，比如对于一开始就进行语法分析的文件，只有外部声明、函数声明，而没有函数实现，测试是否可以通过语法分析，然后对源代码文件改错，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，不需要像目前 VS 的编译器那样智能，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，如果不想采用，那么让函数 *return* 返回，如果想直接跳转到出错提示函数，那么可以采用 *setjmp* 与 *longjmp* 设计，或在出错提示函数中使用 *exit* 让可执行程序结束。

在做项目的过程中，若有任何疑问，则要及时与他人沟通、交流，以避免在不清楚需求的情况下闭门造车，同时注意项目一定要在设计清晰的情况下才能编写代码，切忌在设计还不清晰的情况下就编写代码。

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，此时不需要过于担心，逐步加进去即可，但要注意函数的封装。