手把手教你构建 C 语言编译器（8）- 表达式

Table of Contents

这是整个编译器的最后一部分，解析表达式。什么是表达式？表达式是将各种语言要素的一个组合，用来求值。例如：函数调用、变量赋值、运算符运算等等。

表达式的解析难点有二：一是运算符的优先级问题，二是如何将表达式编译成目标代码。我们就来逐一说明。

手把手教你构建 C 语言编译器系列共有10个部分：

1. 手把手教你构建 C 语言编译器（0）——前言
2. 手把手教你构建 C 语言编译器（1）——设计
3. 手把手教你构建 C 语言编译器（2）——虚拟机
4. 手把手教你构建 C 语言编译器（3）——词法分析器
5. 手把手教你构建 C 语言编译器（4）——递归下降
6. 手把手教你构建 C 语言编译器（5）——变量定义
7. 手把手教你构建 C 语言编译器（6）——函数定义
8. 手把手教你构建 C 语言编译器（7）——语句

9. 手把手教你构建 C 语言编译器（8）——表达式
10. 手把手教你构建 C 语言编译器（9）——总结

运算符的优先级

运算符的优先级决定了表达式的运算顺序，如在普通的四则运算中，乘法 `*` 优先级高于加法 `+`，这就意味着表达式 `2 + 3 * 4` 的实际运行顺序是 `2 + (3 * 4)` 而不是 `(2 + 3) * 4`。

C 语言定义了各种表达式的优先级，可以参考 C 语言运算符优先级。

传统的编程书籍会用“逆波兰式”实现四则运算来讲解优先级问题。实际上，优先级关心的就是哪个运算符先计算，哪个运算符后计算（毕竟叫做“优先级”嘛）。而这就意味着我们需要决定先为哪个运算符生成目标代码（汇编），因为汇编代码是顺序排列的，我们必须先计算优先级高的运算符。

那么如何确定运算符的优先级呢？答曰：栈（递归调用的实质也是栈的处理）。

举一个例子：`2 + 3 - 4 * 5`，它的运算顺序是这样的：

1. 将 `2` 入栈
2. 遇到运算符 `+`，入栈，此时我们期待的是`+`的另一个参数
3. 遇到数字 `3`，原则上我们需要立即计算 `2+3` 的值，但我们不确定数字 `3` 是否属于优先级更高的运算符，所以先将它入栈。

4. 遇到运算符 `-`，它的优先级和 `+` 相同，此时判断参数 `3` 属于这前的 `+`。将运算符 `+` 出栈，并将之前的 `2` 和 `3` 出栈，计算 `2+3` 的结果，得到 `5` 入栈。同时将运算符 `-` 入栈。
5. 遇到数字`4`，同样不能确定是否能立即计算，入栈
6. 遇到运算符 `*` 优先级大于 `-`，入栈
7. 遇到数字`5`，依旧不能确定是否立即计算，入栈
8. 表达式结束，运算符出栈，为 `*`，将参数出栈，计算 `4*5` 得到结果 `20` 入栈。
9. 运算符出栈，为 `-`，将参数出栈，计算 `5-20`，得到 `-15` 入栈。
10. 此时运算符栈为空，因此得到结果 `-15`。

```
// after step 1, 2
|      |
+------+
| 3    |   |      |
+------+   +------+
| 2    |   | +    |
+------+   +------+

// after step 4
|      |   |      |
+------+   +------+
| 5    |   | -    |
+------+   +------+

// after step 7
|      |
+------+
| 5    |
+------+   +------+
| 4    |   | *    |
+------+   +------+
| 5    |   | -    |
+------+   +------+
```

综上，在计算一个运算符‘x’之前，必须先查看它的右方，找出并计算所有优先级大于‘x’的运算符，之后再计算运算符‘x’。

最后注意的是优先通常只与多元运算符相关，单元运算符往往没有这个问题（因为只有一个参数）。也可以认为“优先级”的实质就是两个运算符在抢参数。

一元运算符

上节中说到了运算符的优先级，也提到了优先级一般只与多元运算符有关，这也意味着一元运算符的优先级总是高于多元运算符。因为我们需要先对它们进行解析。

当然，这部分也将同时解析参数本身（如变量、数字、字符串等等）。

关于表达式的解析，与语法分析相关的部分就是上文所说的优先级问题了，而剩下的较难较烦的部分是与目标代码的生成有关的。因此对于需要讲解的运算符，我们主要从它的目标代码入手。

常量

首先是数字，用 `IMM` 指令将它加载到 `AX` 中即可：

```
if (token == Num) {
    match(Num);

    // emit code
    *++text = IMM;
    *++text = token_val;
    expr_type = INT;
}
```

接着是字符串常量。它比较特殊的一点是 C 语言的字符串常量支持如下风格：

```
char *p;
p = "first line"
```

```
    "second line";
```

即跨行的字符串拼接，它相当于：

```
char *p;
p = "first linesecond line";
```

所以解析的时候要注意这一点：

```
else if (token == '"') {
    // emit code
    *++text = IMM;
    *++text = token_val;

    match('"');
    // store the rest strings
    while (token == '"') {
        match('"');
    }

    // append the end of string character '\0', all the data are defaul
    // to 0, so just move data one position forward.
    data = (char *)(((int)data + sizeof(int)) & (-sizeof(int)));
    expr_type = PTR;
}
```

sizeof

`sizeof` 是一个一元运算符，我们需要知道后面参数的类型，类型的解析在前面的文章中我们已经很熟悉了。

```
else if (token == Sizeof) {
    // sizeof is actually an unary operator
    // now only `sizeof(int)`, `sizeof(char)` and `sizeof(*...)` are
    // supported.
    match(Sizeof);
    match('(');
    expr_type = INT;

    if (token == Int) {
        match(Int);
    } else if (token == Char) {
        match(Char);
        expr_type = CHAR;
    }

    while (token == Mul) {
        match(Mul);
        expr_type = expr_type + PTR;
    }

    match(')');

    // emit code
    *++text = IMM;
    *++text = (expr_type == CHAR) ? sizeof(char) : sizeof(int);

    expr_type = INT;
}
```

注意的是只支持 `sizeof(int)`, `sizeof(char)` 及 `sizeof(pointer type...)`。并且它的结果是 `int` 型。

变量与函数调用

由于取变量的值与函数的调用都是以 `Id` 标记开头的，因此将它们放在一起处理。

```
else if (token == Id) {
    // there are several type when occurs to Id
    // but this is unit, so it can only be
    // 1. function call
    // 2. Enum variable
    // 3. global/local variable
    match(Id);

    id = current_id;

    if (token == '(') {
        // function call
        match('(');

        // ①
        // pass in arguments
        tmp = 0; // number of arguments
        while (token != ')') {
            expression(Assign);
            *++text = PUSH;
            tmp ++;

            if (token == ',') {
                match(',');
            }
        }
        match(')');

        // ②
        // emit code
        if (id[Class] == Sys) {
            // system functions
            *++text = id[Value];
        }
        else if (id[Class] == Fun) {
            // function call
            *++text = CALL;
            *++text = id[Value];
        }
        else {
```

```
            printf("%d: bad function call\n", line);
            exit(-1);
        }

        // ③
        // clean the stack for arguments
        if (tmp > 0) {
            *++text = ADJ;
            *++text = tmp;
        }
        expr_type = id[Type];
    }
    else if (id[Class] == Num) {
        // ④
        // enum variable
        *++text = IMM;
        *++text = id[Value];
        expr_type = INT;
    }
    else {
        // ⑤
        // variable
        if (id[Class] == Loc) {
            *++text = LEA;
            *++text = index_of_bp - id[Value];
        }
        else if (id[Class] == Glo) {
            *++text = IMM;
            *++text = id[Value];
        }
        else {
            printf("%d: undefined variable\n", line);
            exit(-1);
        }

        //⑥
        // emit code, default behaviour is to load the value of the
        // address which is stored in `ax`
        expr_type = id[Type];
        *++text = (expr_type == Char) ? LC : LI;
```

```
    }
  }
```

①中注意我们是顺序将参数入栈，这和第三章：虚拟机中讲解的指令是对应的。与之不同，标准 C 是逆序将参数入栈的。

②中判断函数的类型，同样在第三章：“虚拟机”中我们介绍过内置函数的支持，如 `printf`，`read`，`malloc` 等等。内置函数有对应的汇编指令，而普通的函数则编译成 `CALL <addr>` 的形式。

③用于清除入栈的参数。因为我们不在乎出栈的值，所以直接修改栈指针的大小即可。

④：当该标识符是全局定义的枚举类型时，直接将对应的值用 `IMM` 指令存入 `AX` 即可。

⑤则是用于加载变量的值，如果是局部变量则采用与 `bp` 指针相对位置的形式（参见第 7章函数定义）。而如果是全局变量则用 `IMM` 加载变量的地址。

⑥：无论是全局还是局部变量，最终都根据它们的类型用 `LC` 或 `LI` 指令加载对应的值。

关于变量，你可能有疑问，如果遇到标识符就用 `LC/LI` 载入相应的值，那诸如 `a[10]` 之类的表达式要如何实现呢？后面我们会看到，根据标识符后的运算符，我们可能会修改或删除现有的 `LC/LI` 指令。

强制转换

虽然我们前面没有提到，但我们一直用 `expr_type` 来保存一个表达式的类型，强制转换的作用是获取转换的类型，并直接修改 `expr_type` 的值。

```
else if (token == '(') {
    // cast or parenthesis
    match('(');
    if (token == Int || token == Char) {
        tmp = (token == Char) ? CHAR : INT; // cast type
        match(token);
        while (token == Mul) {
            match(Mul);
            tmp = tmp + PTR;
        }

        match(')');

        expression(Inc); // cast has precedence as Inc(++)

        expr_type  = tmp;
    } else {
        // normal parenthesis
        expression(Assign);
        match(')');
    }
}
```

指针取值

诸如 `*a` 的指针取值，关键是判断 `a` 的类型，而就像上节中提到的，当一个表达式解析结束时，它的类型保存在变量 `expr_type` 中。

```
else if (token == Mul) {
    // dereference *<addr>
    match(Mul);
    expression(Inc); // dereference has the same precedence as Inc(++)

    if (expr_type >= PTR) {
        expr_type = expr_type - PTR;
    } else {
        printf("%d: bad dereference\n", line);
        exit(-1);
    }

    *++text = (expr_type == CHAR) ? LC : LI;
}
```

取址操作

这里我们就能看到“变量与函数调用”一节中所说的修改或删除 `LC/LI` 指令了。前文中我们说到，对于变量，我们会先加载它的地址，并根据它们类型使用 `LC/LI` 指令加载实际内容，例如对变量 `a`：

```
IMM <addr>
LI
```

那么对变量 `a` 取址，其实只要不执行 `LC/LI` 即可。因此我们删除相应的指令。

```
else if (token == And) {
    // get the address of
    match(And);
```

```
        expression(Inc); // get the address of
        if (*text == LC || *text == LI) {
            text --;
        } else {
            printf("%d: bad address of\n", line);
            exit(-1);
        }

        expr_type = expr_type + PTR;
    }
```

逻辑取反

我们没有直接的逻辑取反指令，因此我们判断它是否与数字 0 相等。而数字 0 代表了逻辑 “False”。

```
    else if (token == '!') {
        // not
        match('!');
        expression(Inc);

        // emit code, use <expr> == 0
        *++text = PUSH;
        *++text = IMM;
        *++text = 0;
        *++text = EQ;

        expr_type = INT;
    }
```

按位取反

同样我们没有相应的指令，所以我们用异或来实现，即 `~a = a ^ 0xFFFF`。

```
else if (token == '~') {
    // bitwise not
    match('~');
    expression(Inc);

    // emit code, use <expr> XOR -1
    *++text = PUSH;
    *++text = IMM;
    *++text = -1;
    *++text = XOR;

    expr_type = INT;
}
```

正负号

注意这里并不是四则运算中的加减法，而是单个数字的取正取负操作。同样，我们没有取负的操作，用 `0 - x` 来实现 `-x`。

```
else if (token == Add) {
    // +var, do nothing
    match(Add);
    expression(Inc);

    expr_type = INT;
}
else if (token == Sub) {
    // -var
    match(Sub);

    if (token == Num) {
        *++text = IMM;
        *++text = -token_val;
        match(Num);
    } else {
```

```
            *++text = IMM;
            *++text = -1;
            *++text = PUSH;
            expression(Inc);
            *++text = MUL;
        }

        expr_type = INT;
    }
```

自增自减

注意的是自增自减操作的优先级是和它的位置有关的。如 `++p` 的优先级高于 `p++`，这里我们解析的就是类似 `++p` 的操作。

```
else if (token == Inc || token == Dec) {
    tmp = token;
    match(token);
    expression(Inc);
    // ①
    if (*text == LC) {
        *text = PUSH;  // to duplicate the address
        *++text = LC;
    } else if (*text == LI) {
        *text = PUSH;
        *++text = LI;
    } else {
        printf("%d: bad lvalue of pre-increment\n", line);
        exit(-1);
    }
    *++text = PUSH;
    *++text = IMM;
    // ②
    *++text = (expr_type > PTR) ? sizeof(int) : sizeof(char);
    *++text = (tmp == Inc) ? ADD : SUB;
    *++text = (expr_type == CHAR) ? SC : SI;
}
```

对应的汇编代码也比较直观，只是在实现 `++p` 时，我们要使用变量 `p` 的地址两次，所以我们需要先 `PUSH` （①）。

②则是因为自增自减操作还需要处理是指针的情形。

二元运算符

这里，我们需要处理多运算符的优先级问题，就如前文的“优先级”一节提到的，我们需要不断地向右扫描，直到遇到优先级 **小于** 当前优先级的运算符。

回想起我们之前定义过的各个标记，它们是以优先级从低到高排列的，即 `Assign` 的优先级最低，而 `Brak` （`[`） 的优先级最高。

```
enum {
  Num = 128, Fun, Sys, Glo, Loc, Id,
  Char, Else, Enum, If, Int, Return, Sizeof, While,
  Assign, Cond, Lor, Lan, Or, Xor, And, Eq, Ne, Lt, Gt, Le, Ge, Shl, Sh
};
```

所以，当我们调用 `expression(level)` 进行解析的时候，我们其实通过了参数 `level` 指定了当前的优先级。在前文的一元运算符处理中也用到了这一点。

所以，此时的二元运算符的解析的框架为：

```
while (token >= level) {
    // parse token for binary operator and postfix operator
}
```

解决了优先级的问题，让我们继续讲解如何把运算符编译成汇编代码吧。

赋值操作

赋值操作是优先级最低的运算符。考虑诸如 `a = (expession)` 的表达式，在解析 `=` 之前，我们已经为变量 `a` 生成了如下的汇编代码：

```
IMM <addr>
LC/LI
```

当解析完`=`右边的表达式后，相应的值会存放在`ax`中，此时，为了实际将这个值保存起来，我们需要类似下面的汇编代码：

```
IMM <addr>
PUSH
SC/SI
```

明白了这点，也就能理解下面的源代码了：

```
tmp = expr_type;
if (token == Assign) {
    // var = expr;
    match(Assign);
    if (*text == LC || *text == LI) {
        *text = PUSH; // save the lvalue's pointer
    } else {
        printf("%d: bad lvalue in assignment\n", line);
        exit(-1);
    }
    expression(Assign);

    expr_type = tmp;
    *++text = (expr_type == CHAR) ? SC : SI;
}
```

三目运算符

这是 C 语言中唯一的一个三元运算符：`? :`，它相当于一个小型的 If 语句，所以生成的代码也类似于 If 语句，这里就不多作解

释。

```
else if (token == Cond) {
    // expr ? a : b;
    match(Cond);
    *++text = JZ;
    addr = ++text;
    expression(Assign);
    if (token == ':') {
        match(':');
    } else {
        printf("%d: missing colon in conditional\n", line);
        exit(-1);
    }
    *addr = (int)(text + 3);
    *++text = JMP;
    addr = ++text;
    expression(Cond);
    *addr = (int)(text + 1);
}
```

逻辑运算符

这包括 `||` 和 `&&`。它们对应的汇编代码如下：

```
<expr1> || <expr2>      <expr1> && <expr2>

  ...<expr1>...           ...<expr1>...
  JNZ b                   JZ b
  ...<expr2>...           ...<expr2>...
b:                      b:
```

所以源码如下：

```
else if (token == Lor) {
    // logic or
    match(Lor);
    *++text = JNZ;
    addr = ++text;
    expression(Lan);
    *addr = (int)(text + 1);
    expr_type = INT;
}
else if (token == Lan) {
    // logic and
    match(Lan);
    *++text = JZ;
    addr = ++text;
    expression(Or);
    *addr = (int)(text + 1);
    expr_type = INT;
}
```

数学运算符

它们包括 `|`, `^`, `&`, `==`, `!=` `<=`, `>=`, `<`, `>`, `<<`, `>>`, `+`, `-`, `*`, `/`, `%`。它们的实现都很类似，我们以异或 `^` 为例：

```
<expr1> ^ <expr2>

...<expr1>...          <- now the result is on ax
PUSH
...<expr2>...          <- now the value of <expr2> is on ax
XOR
```

所以它对应的代码为：

```
else if (token == Xor) {
    // bitwise xor
    match(Xor);
    *++text = PUSH;
    expression(And);
    *++text = XOR;
    expr_type = INT;
}
```

其它的我们便不再详述。但这当中还有一个问题，就是指针的加减。在 C 语言中，指针加上数值等于将指针移位，且根据不同的类型移动的位移不同。如 `a + 1`，如果 `a` 是 `char *` 型，则移动一字节，而如果 `a` 是 `int *` 型，则移动 4 个字节（32位系统）。

另外，在作指针减法时，如果是两个指针相减（相同类型），则结果是两个指针间隔的元素个数。因此要有特殊的处理。

下面以加法为例，对应的汇编代码为：

```
<expr1> + <expr2>

normal          pointer

<expr1>         <expr1>
PUSH            PUSH
<expr2>         <expr2>      |
ADD             PUSH         | <expr2> * <unit>
                IMM <unit>   |
                MUL          |
                ADD
```

即当 `<expr1>` 是指针时，要根据它的类型放大 `<expr2>` 的值，因此对应的源码如下：

```
else if (token == Add) {
    // add
    match(Add);
    *++text = PUSH;
    expression(Mul);

    expr_type = tmp;
    if (expr_type > PTR) {
        // pointer type, and not `char *`
        *++text = PUSH;
        *++text = IMM;
        *++text = sizeof(int);
        *++text = MUL;
    }
    *++text = ADD;
}
```

相应的减法的代码就不贴了，可以自己实现看看，也可以看文末给出的链接。

自增自减

这次是后缀形式的，即 `p++` 或 `p--`。与前缀形式不同的是，在执行自增自减后， `ax` 上需要保留原来的值。所以我们首先执行类似前缀自增自减的操作，再将 `ax` 中的值执行减/增的操作。

```
// 前缀形式 生成汇编代码
*++text = PUSH;
*++text = IMM;
*++text = (expr_type > PTR) ? sizeof(int) : sizeof(char);
```

```
    *++text = (tmp == Inc) ? ADD : SUB;
    *++text = (expr_type == CHAR) ? SC : SI;

    // 后缀形式 生成汇编代码
    *++text = PUSH;
    *++text = IMM;
    *++text = (expr_type > PTR) ? sizeof(int) : sizeof(char);
    *++text = (token == Inc) ? ADD : SUB;
    *++text = (expr_type == CHAR) ? SC : SI;
    *++text = PUSH;                                          //
    *++text = IMM;                                           // 执行相反
    *++text = (expr_type > PTR) ? sizeof(int) : sizeof(char);  //
    *++text = (token == Inc) ? SUB : ADD;                    //
```

数组取值操作

在学习 C 语言的时候你可能已经知道了，诸如 `a[10]` 的操作等价于 `*(a + 10)`。因此我们要做的就是生成类似的汇编代码：

```
else if (token == Brak) {
    // array access var[xx]
    match(Brak);
    *++text = PUSH;
    expression(Assign);
    match(']');

    if (tmp > PTR) {
        // pointer, `not char *`
        *++text = PUSH;
        *++text = IMM;
        *++text = sizeof(int);
        *++text = MUL;
    }
    else if (tmp < PTR) {
        printf("%d: pointer type expected\n", line);
        exit(-1);
```

```
        }
        expr_type = tmp - PTR;
        *++text = ADD;
        *++text = (expr_type == CHAR) ? LC : LI;
    }
```

代码

除了上述对表达式的解析外，我们还需要初始化虚拟机的栈，我们可以正确调用 `main` 函数，且当 `main` 函数结束时退出进程。

```
int *tmp;
// setup stack
sp = (int *)((int)stack + poolsize);
*--sp = EXIT; // call exit if main returns
*--sp = PUSH; tmp = sp;
*--sp = argc;
*--sp = (int)argv;
*--sp = (int)tmp;
```

当然，最后要注意的一点是：所有的变量定义必须放在语句之前。

本章的代码可以在 Github 上下载，也可以直接 clone

```
git clone -b step-6 https://github.com/lotabout/write-a-C-interpreter
```

通过 `gcc -o xc-tutor xc-tutor.c` 进行编译。并执行 `./xc-tutor hello.c` 查看结果。

正如我们保证的那样，我们的代码是自举的，能自己编译自己，所以你可以执行 `./xc-tutor xc-tutor.c hello.c`。可以看到和之前有同样的输出。

小结

本章我们进行了最后的解析，解析表达式。本章有两个难点：

1. 如何通过递归调用 `expression` 来实现运算符的优先级。
2. 如何为每个运算符生成对应的汇编代码。

尽管代码看起来比较简单（虽然多），但其中用到的原理还是需要仔细推敲的。

最后，恭喜你！通过一步步的学习，自己实现了一个C语言的编译器（好吧，是解释器）。