手把手教你构建 C 语言编译器（4）- 递归下降

Table of Contents

本章我们将讲解递归下降的方法，并用它完成一个基本的四则运算的语法分析器。

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

什么是递归下降

传统上，编写语法分析器有两种方法，一种是自顶向下，一种是自底向上。自顶向下是从起始非终结符开始，不断地对非终结符进行分解，直到匹配输入的终结符；自底向上是不断地将终结符进行合并，直到合并成起始的非终结符。

其中的自顶向下方法就是我们所说的递归下降。

终结符与非终结符

没有学过编译原理的话可能并不知道什么是“终结符”，“非终结符”。这里我简单介绍一下。首先是 BNF 范式，就是一种用来描述语法的语言，例如，四则运算的规则可以表示如下：

```
<expr> ::= <expr> + <term>
         | <expr> - <term>
         | <term>

<term> ::= <term> * <factor>
         | <term> / <factor>
         | <factor>

<factor> ::= ( <expr> )
           | Num
```

用尖括号 `<>` 括起来的就称作 **非终结符**，因为它们可以用 `::=` 右侧的式子代替。`|` 表示选择，如 `<expr>` 可以是 `<expr> + <term>`、`<expr> - <term>` 或 `<term>` 中的一种。而没有出现

在`::=`左边的就称作 **终结符**，一般终结符对应于词法分析器输出的标记。

四则运算的递归下降

例如，我们对 `3 * (4 + 2)` 进行语法分析。我们假设词法分析器已经正确地将其中的数字识别成了标记 `Num`。

递归下降是从起始的非终结符开始（顶），本例中是 `<expr>`，实际中可以自己指定，不指定的话一般认为是第一个出现的非终结符。

```
1. <expr> => <expr>
2.           => <term>        * <factor>
3.              => <factor>     |
4.                 => Num (3)   |
5.                              => ( <expr> )
6.                                   => <expr>           + <term>
7.                                      => <term>          |
8.                                         => <factor>     |
9.                                            => Num (4)   |
10.                                                       => <factor>
11.                                                          => Num (2
```

可以看到，整个解析的过程是在不断对非终结符进行替换（向下），直到遇见了终结符（底）。而我们可以从解析的过程中看出，一些非终结符如`<expr>`被递归地使用了。

为什么选择递归下降

从上小节对四则运算的递归下降解析可以看出，整个解析的过程和语法的 BNF 表示是十分接近的，更为重要的是，我们可以很容易地直接将 BNF 表示转换成实际的代码。方法是为每个产生式（即 `非终结符 ::= ...`）生成一个同名的函数。

这里会有一个疑问，就是上例中，当一个终结符有多个选择时，如何确定具体选择哪一个？如为什么用 `<expr> ::= <term> * <factor>` 而不是 `<expr> ::= <term> / <factor>`？这就用到了上一章中提到的“向前看 k 个标记”的概念了。我们向前看一个标记，发现是 `*`，而这个标记足够让我们确定用哪个表达式了。

另外，递归下下降方法对 BNF 方法本身有一定的要求，否则会有一些问题，如经典的“左递归”问题。

左递归

原则上我们是不讲这么深入，但我们上面的四则运算的文法就是左递归的，而左递归的语法是没法直接使用递归下降的方法实现的。因此我们要消除左递归，消除后的文法如下：

```
<expr> ::= <term> <expr_tail>
<expr_tail> ::= + <term> <expr_tail>
              | - <term> <expr_tail>
              | <empty>

<term> ::= <factor> <term_tail>
<term_tail> ::= * <factor> <term_tail>
              | / <factor> <term_tail>
              | <empty>
```

```
<factor> ::= ( <expr> )
           | Num
```

消除左递归的相关方法，这里不再多说，请自行查阅相关的资料。

四则运算的实现

本节中我们专注语法分析器部分的实现，具体实现很容易，我们直接贴上代码，就是上述的消除左递归后的文法直接转换而来的：

```
int expr();

int factor() {
    int value = 0;
    if (token == '(') {
        match('(');
        value = expr();
        match(')');
    } else {
        value = token_val;
        match(Num);
    }
    return value;
}

int term_tail(int lvalue) {
    if (token == '*') {
        match('*');
        int value = lvalue * factor();
        return term_tail(value);
    } else if (token == '/') {
        match('/');
        int value = lvalue / factor();
```

```
        return term_tail(value);
    } else {
        return lvalue;
    }
}

int term() {
    int lvalue = factor();
    return term_tail(lvalue);
}

int expr_tail(int lvalue) {
    if (token == '+') {
        match('+');
        int value = lvalue + term();
        return expr_tail(value);
    } else if (token == '-') {
        match('-');
        int value = lvalue - term();
        return expr_tail(value);
    } else {
        return lvalue;
    }
}

int expr() {
    int lvalue = term();
    return expr_tail(lvalue);
}
```

可以看到，有了BNF方法后，采用递归向下的方法来实现编译器是很直观的。

我们把词法分析器的代码一并贴上：

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

```
enum {Num};
int token;
int token_val;
char *line = NULL;
char *src = NULL;

void next() {
    // skip white space
    while (*src == ' ' || *src == '\t') {
        src ++;
    }

    token = *src++;

    if (token >= '0' && token <= '9' ) {
        token_val = token - '0';
        token = Num;

        while (*src >= '0' && *src <= '9') {
            token_val = token_val*10 + *src - '0';
            src ++;
        }
        return;
    }
}

void match(int tk) {
    if (token != tk) {
        printf("expected token: %d(%c), got: %d(%c)\n", tk, tk, token,
        exit(-1);
    }
    next();
}
```

最后是`main`函数：

```
int main(int argc, char *argv[])
{
```

```
    size_t linecap = 0;
    ssize_t linelen;
    while ((linelen = getline(&line, &linecap, stdin)) > 0) {
        src = line;
        next();
        printf("%d\n", expr());
    }
    return 0;
}
```

小结

本章中我们介绍了递归下降的方法，并用它来实现了四则运算的语法分析器。

花这么大精力讲解递归下降方法，是因为几乎所有手工编写的语法分析器都或多或少地有它的影子。换句话说，掌握了递归下降的方法，就可以应付大多数的语法分析器编写。

同时我们也用实例看到了理论（BNF 语法，左递归的消除）是如何帮助我们的工程实现的。尽管理论不是必需的，但如果能掌握它，对于提高我们的水平还是很有帮助的。