第14章 移除直接左递归

在5.4节中我们看到，用自然方式处理算术表达式是具有歧义性的。例如，下列expr可以将1+2*3解释为（1+2）*3或者1+（2*3）。通过优先选择位置靠前的备选分支，ANTLR优雅地解决了歧义问题。

left-recursion-removal/Expr.g4

```
stat: expr ';' ;

expr:   expr '*' expr       // 优先级 4
    |   expr '+' expr       // 优先级 3
    |   INT                 // 主表达式（优先级 2）
    |   ID                  // 主表达式（优先级 1）
    ;
```

expr规则仍然是左递归的，传统的自顶向下的语法（例如ANTLR 3）无法处理这样的规则。在本章中，我们会探究ANTLR处理左递归和运算符优先级的方式。简单而言，ANTLR将左递归替换成一个（...）*，它会比较前一个和下一个运算符的优先级。

熟悉这样的规则变换是很重要的，因为生成的代码反映的是转换后的规则，而非原先的规则。更重要的是，当一份语法没有按照我们的期望对运算符进行分组和结合时，我们需要知道原因。大多数用户可以只阅读本章第一节中与有效递归备选分支模式相关的内容，对实现细节感兴趣的进阶用户可以继续阅读第二节。

让我们首先学习ANTLR采取的转换方案，然后通过一个例子来在实践中学习优先级上升[1]（precedence climbing）算法。

[1] Theodore Norvell创造了这个术语，但是最初的工作是由Keith Clarke完成的。

14.1 直接左递归备选分支模式

ANTLR通过检查下列四种子表达式运算模式来认定一条规则为左递归规则。

二元 expr规则的某个备选分支符合expr op expr或者expr（op1|op2|...|opN）expr的形式。op可以是单一词法符号或者多词法符号构成的运算符。例如，Java语法可能独立处理尖括号，而非将<=>或>=当作单一词法符号。下面的备选分支将比较运算符按照同一优先级处理：

```
expr: ...
    | expr ('<' '=' | '>' '=' | '>' | '<') expr
   ...
    ;
```

op可以是对另外一条规则的引用，例如，我们可能将若干个词法符号提出来，组成一条新的规则。

```
expr: ...
    | expr compareOps expr
   ...
    ;
compareOps : ('<' '=' | '>' '=' | '>' | '<') ;
```

三元 expr的某个备选分支符合expr op1 expr op2 expr的形式。op1和op2必须是单词法符号引用。这种模式的典型代表是类C语言中的“?:”运算符:

```
expr: ...
    | expr '?' expr ':' expr
   ...
    ;
```

一元前缀 expr的某条规则符合elements expr的形式。ANTLR将任意元素后的尾递归规则引用视作一元前缀模式，前提是它不符合二元模式和三元模式。下面是两个具有前缀运算符的备选分支:

```
expr: ...
    | '(' type ')' expr
   ...
    | ('+'|'-'|'++'|'--') expr
   ...
    ;
```

一元后缀 expr的某个备选分支符合expr elements形式。和前缀模式相同，ANTLR将任意元素前的直接左递归规则视作一元后缀模式，前提是它不符合二元模式和三元模式。下面是两个具有后缀运算符的备选分支:

```
expr: ...
    | expr '.' Identifier
    ...
    | expr '.' 'super' '(' exprList? ')'
    ...
    ;
```

其他形式的备选分支都被作为主表达式（primary expression）元素处理，例如标识符或者整数，也包括类似'（'expr'）'的形式，因为它不符合上述四种模式的任意一种。这是必要的，因为括号存在的意义是将其包含的表达式当作一个原子元素处理。这样的“ 其他形式”备选分支可以以任意顺序出现。ANTLR能够正确地处理它们。除此之外的备选分支顺序都是需要特别注意的。下面是一些主表达式备选分支的示例：

```
expr: ...
    |   literal
    |   Identifier
    |   type '.' 'class'
    ...
    ;
```

除非额外指定，ANTLR假设所有的运算符都是左结合的。换句话说，1+2+3会被分组为（1+2）+3。不过，某些运算符是右结合的，例如赋值运算符和指数运算符，我们已经在5.4节中见过它们的处理方式。通过assoc选项，可以指定右结合性[1]。

```
expr: expr '^'<assoc=right> expr
    ...
    | expr '='<assoc=right> expr
    ...
    ;
```

在下一节中，我们将会看到ANTLR翻译这些模式的方法。

[1] ANTLR 4.2之后，assoc的语法已经变更，详见5.4节译注。——译者注

14.2 左递归规则转换

如果你打开ANTLR命令行的“-Xlog”选项，你就可以在日志文件中看到转换后的左递归规则。下面是在先前Expr.g4语法中的stat和expr规则上发生的转换过程：

```
// 使用"antlr4 -Xlog Expr.g4"查看转换后的规则
stat:   expr[0] ';' ;  // 匹配包含优先级运算符的表达式

expr[int _p]              // _p是预期的最低优先级
:    ( INT           // 匹配主表达式（无运算符的表达式）
     | ID
     )
     // 匹配优先级大于等于预期最低值的运算符
     ( {4 >= $_p}? '*' expr[5]  // *具有优先级4
     | {3 >= $_p}? '+' expr[4]  // +具有优先级3
     )*
;
```

这些转换真是一项浩大的工程。不要被ANTLR处理expr的过程吓到，我们希望学习的是这些判定根据运算符优先级指导语法分析器进行正确分组的方法。

关键在于，究竟是在expr的当前调用中匹配下一个运算符，还是令expr的调用者匹配下一个运算符。(...)*能够匹配当前运算符和右

侧运算符。例如，对于输入1+2*3，该循环能够匹配+2和*3。循环中的判定能够决定，令语法分析器匹配这二者，还是放弃它们。如果操作符的优先级3低于当前子表达式预期的最低优先级_p，{3>=$_p}?就会关闭这个备选分支。

这不是带运算符优先级的语法分析

不要将这种机制和你可能在维基百科[1]读到的带运算符优先级的语法分析相混淆。带运算符优先级的语法分析无法处理一些特殊情况，例如具有两种不同优先级的负号，一种用于取负，一种用于二元减法。它也无法处理具有两条相邻的规则的备选分支，如expr ops expr。参考文献【Compilers：Principles，Techniques，and Tools[ALSU06]】给出了详细解释。

参数_p的值总是前一个运算符的优先级。_p从0开始，因为对expr的非递归调用会传递0，例如stat会调用expr[0]。为了解实际情况中_p的值，我们可以查看基于转换后的规则生成的语法分析树（参数_p的值在方括号中显示）。注意，这些语法分析树并不是ANTLR基于原先的左递归规则建立的。这些是转换后的规则对应的语法分析树。如图14-1所示为样例输入和相应的语法分析树。

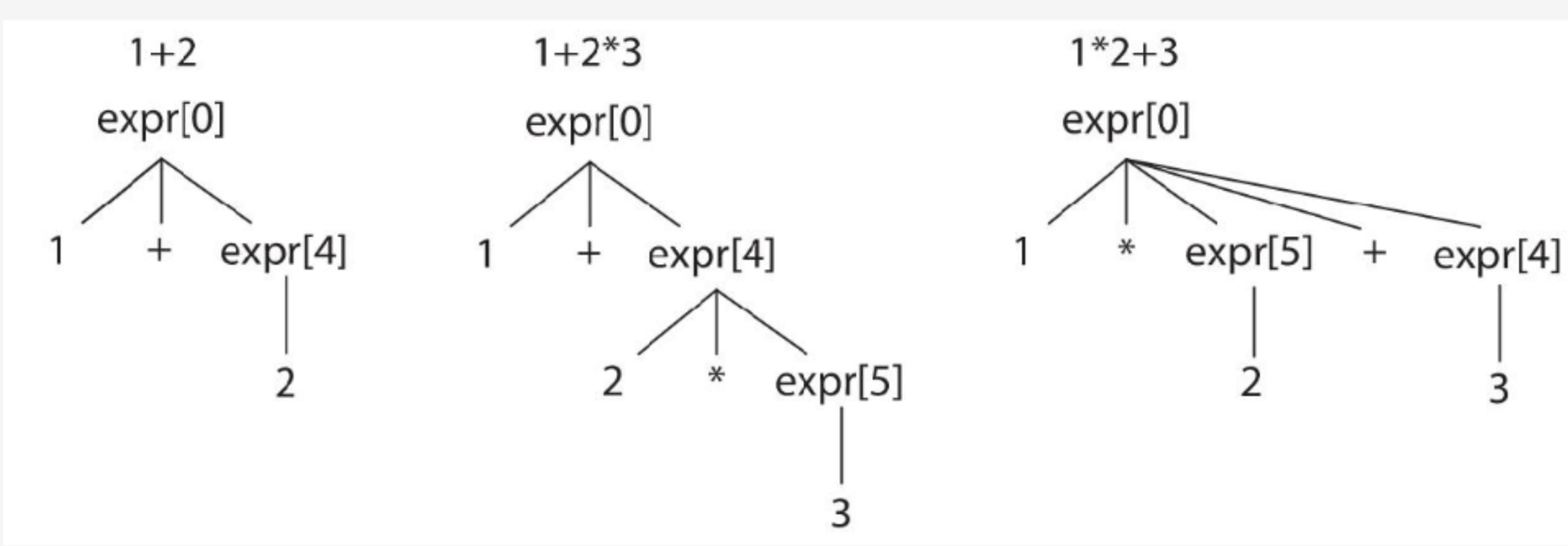

图14-1　样例输入及其相应的语法树

在第一棵树中，对expr的初始调用传递的_p是0，expr立刻匹配了（INT|ID）子规则对应的1。现在expr必须决定是匹配接下来的+，还是直接退出循环并返回。此时，执行的判定是{3>=0}?，因此，我们进入了循环，匹配到了+，然后递归调用了expr规则，传递了参数4。下一次调用匹配到了2并立即返回，因为没有更多的输入了。expr[0]随后返回到了最初的stat中对expr的调用。

第二棵树展示了expr[0]匹配1，以及又一次的{3>=0}?判定，它允许我们匹配+和下一次调用expr[4]。这次调用匹配到了2，然后执行判定{4>=4}?，它允许语法分析器通过expr[5]，进一步匹配之后的*。

第三棵语法分析树是最有趣的。最初的调用expr[0]匹配了1和*，因为{4>=0}?结果为真。此循环递归调用了expr[5]，并匹配了2。现在，在expr[5]的内部，语法分析器不应当匹配+，因为这样的话，2+3就会在乘法之前被执行（即在语法分析树中，我们会看到expr[5]的子节点是2+3）。判定{3>=5}?关闭了对应的备选分支，因此expr[5]没有匹配+就提前返回了。在返回之后，由于{3>=0}?为真，expr[0]匹配了+3。

我希望本章的内容能够加深大家对优先级上升机制的理解。欲了解更多细节，请参阅Norvell的论述[2]。

[1] http://en.wikipedia.org/wiki/Operator-precedence_parser

[2] http://www.engr.mun.ca/~theo/Misc/exp_parsing.htm