

04 | 语法分析（二）：解决二元表达式中的难点

2019-08-21 宫文学

编译原理之美

[进入课程 >](#)



讲述：宫文学

时长 13:41 大小 12.54M



在“[03 | 语法分析（一）：纯手工打造公式计算器](#)”中，我们已经初步实现了一个公式计算器。而且你还在这个过程中，直观地获得了写语法分析程序的体验，在一定程度上破除了对语法分析算法的神秘感。

当然了，你也遇到了一些问题，比如怎么消除左递归，怎么确保正确的优先级和结合性。所以本节课的主要目的就是解决这几个问题，让你掌握像算术运算这样的二元表达式（Binary Expression）。

不过在课程开始之前，我想先带你简单地温习一下什么是左递归（Left Recursive）、优先级（Priority）和结合性（Associativity）。


在二元表达式的语法规则中，如果产生式的第一个元素是它自身，那么程序就会无限地递归下去，这种情况就叫做**左递归**。比如加法表达式的产生式“加法表达式 + 乘法表达式”，就是左递归的。而优先级和结合性则是计算机语言中与表达式有关的核心概念。它们都涉及了语法规则的设计问题。

我们要想深入探讨语法规则设计，需要像在词法分析环节一样，先了解如何用形式化的方法表达语法规则。“工欲善其事必先利其器”。熟练地阅读和书写语法规则，是我们在语法分析环节需要掌握的一项基本功。

所以本节课我会先带你了解如何写语法规则，然后在此基础上，带你解决上面提到的三个问题。

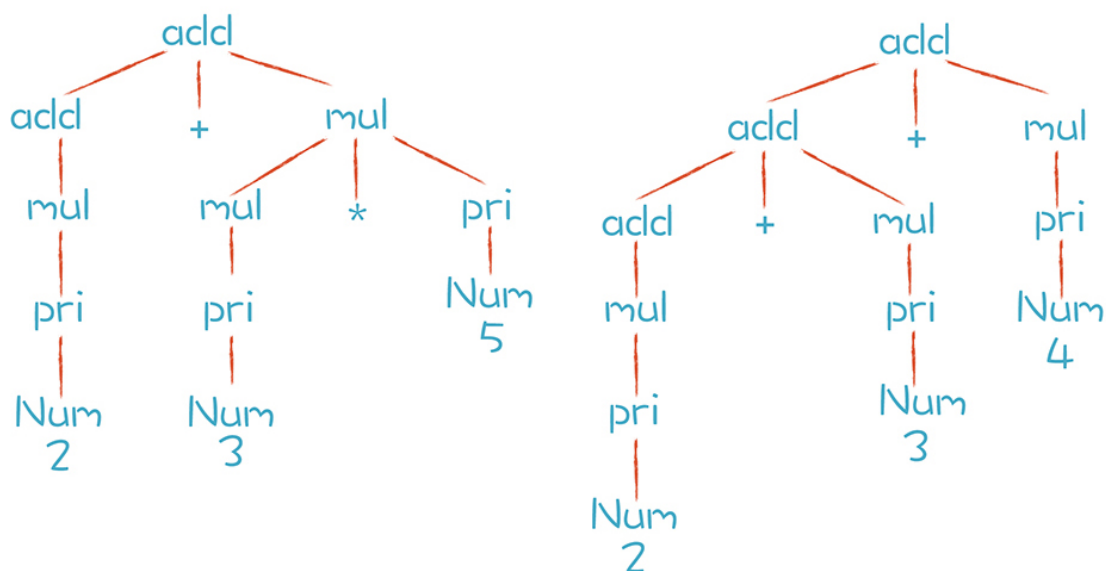
书写语法规则，并进行推导

我们已经知道，语法规则是由上下文无关文法表示的，而上下文无关文法是由一组替换规则（又叫产生式）组成的，比如算术表达式的文法规则可以表达成下面这种形式：

 复制代码

```
1 add -> mul | add + mul
2 mul -> pri | mul * pri
3 pri -> Id | Num | (add)
```


按照上面的产生式，add 可以替换成 mul，或者 add + mul。这样的替换过程又叫做“推导”。以“2+3*5”和“2+3+4”这两个算术表达式为例，这两个算术表达式的推导过程分别如下图所示：



通过上图的推导过程，你可以清楚地看到这两个表达式是怎样生成的。而分析过程中形成的这棵树，其实就是 AST。只不过我们手写的算法在生成 AST 的时候，通常会做一些简化，省略掉中间一些不必要的节点。比如，“add-add-mul-pri-Num”这一条分支，实际手写时会被简化成“add-Num”。其实，简化 AST 也是优化编译过程的一种手段，如果不做简化，呈现的效果就是上图的样子。

那么，上图中两颗树的叶子节点有哪些呢？Num、+ 和 * 都是终结符，终结符都是词法分析中产生的 Token。而那些非叶子节点，就是非终结符。文法的推导过程，就是把非终结符不断替换的过程，让最后的结果没有非终结符，只有终结符。

而在实际应用中，语法规则经常写成下面这种形式：


 复制代码

```
1 add ::= mul | add + mul
2 mul ::= pri | mul * pri
3 pri ::= Id | Num | (add)
```

这种写法叫做“**巴科斯范式**”，简称 BNF。Antlr 和 Yacc 这两个工具都用这种写法。为了简化书写，我有时会在课程中把“::=”简化成一个冒号。你看到的时候，知道是什么意思就可以了。

你有时还会听到一个术语，叫做**扩展巴科斯范式 (EBNF)**。它跟普通的 BNF 表达式最大的区别，就是里面会用到类似正则表达式的一些写法。比如下面这个规则中运用了 * 号，来

表示这个部分可以重复 0 到多次：

 复制代码


```
1 add -> mul (+ mul)*
```

其实这种写法跟标准的 BNF 写法是等价的，但是更简洁。为什么是等价的呢？因为一个项多次重复，就等价于通过递归来推导。从这里我们还可以得到一个推论：就是上下文无关文法包含了正则文法，比正则文法能做更多的事情。

确保正确的优先级

掌握了语法规则的写法之后，我们来看看如何用语法规则来保证表达式的优先级。刚刚，我们由加法规则推导到乘法规则，这种方式保证了 AST 中的乘法节点一定会在加法节点的下层，也就保证了乘法计算优先于加法计算。

听到这儿，你一定会想到，我们应该把关系运算（>、=、<）放在加法的上层，逻辑运算（and、or）放在关系运算的上层。的确如此，我们试着将它写出来：


 复制代码

```
1 exp -> or | or = exp
2 or -> and | or || and
3 and -> equal | and && equal
4 equal -> rel | equal == rel | equal != rel
5 rel -> add | rel > add | rel < add | rel >= add | rel <= add
6 add -> mul | add + mul | add - mul
7 mul -> pri | mul * pri | mul / pri
```

这里表达的优先级从低到高是：赋值运算、逻辑运算（or）、逻辑运算（and）、相等比较（equal）、大小比较（rel）、加法运算（add）、乘法运算（mul）和基础表达式（pri）。

实际语言中还有更多不同的优先级，比如位运算等。而且优先级是能够改变的，比如我们通常会在语法里通过括号来改变计算的优先级。不过这怎么表达成语法规则呢？

其实，我们在最低层，也就是优先级最高的基础表达式（`pri`）这里，用括号把表达式包裹起来，递归地引用表达式就可以了。这样的话，只要在解析表达式的时候遇到括号，那么就on知道这个是最优先的。这样的话就实现了优先级的改变：

 复制代码

```
1 pri -> Id | Literal | (exp)
```

了解了这些内容之后，到目前为止，你已经会写整套的表达式规则了，也能让公式计算器支持这些规则了。另外，在使用一门语言的时候，如果你不清楚各种运算确切的优先级，除了查阅常规的资料，你还多了一项新技能，就是阅读这门语言的语法规则文件，这些规则可能就是用 BNF 或 EBNF 的写法书写的。

弄明白优先级的问题以后，我们再来讨论一下结合性这个问题。

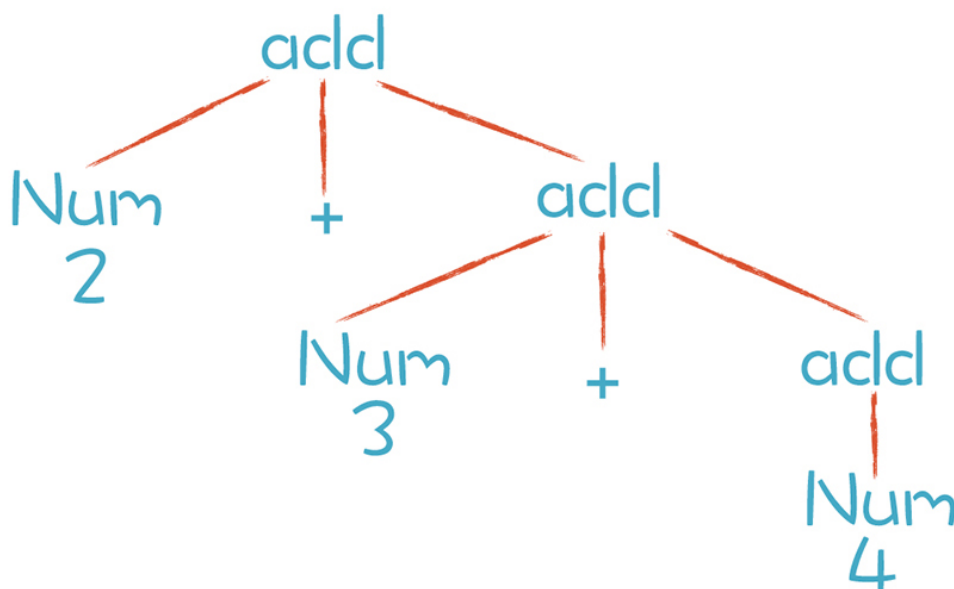
确保正确的结合性

在上一讲中，我针对算术表达式写的第二个文法是错的，因为它的计算顺序是错的。“`2+3+4`”这个算术表达式，先计算了“`3+4`”然后才和“`2`”相加，计算顺序从右到左，正确的应该是从左往右才对。

这就是运算符的结合性问题。什么是结合性呢？同样优先级的运算符是从左到右计算还是从右到左计算叫做结合性。我们常见的加减乘除等算术运算是左结合的，“`.`”符号也是左结合的。


比如“`rectangle.center.x`”是先获得长方形（`rectangle`）的中心点（`center`），再获得这个点的 `x` 坐标。计算顺序是从左向右的。那有没有右结合的例子呢？肯定是有的。赋值运算就是典型的右结合的例子，比如“`x = y = 10`”。

我们再来回顾一下“`2+3+4`”计算顺序出错的原因。用之前错误的右递归的文法解析这个表达式形成的简化版本的 AST 如下：



根据这个 AST 做计算会出现计算顺序的错误。不过如果我们将递归项写在左边，就不会出现这种结合性的错误。于是我们得出一个规律：**对于左结合的运算符，递归项要放在左边；而右结合的运算符，递归项放在右边。**

所以你能看到，我们在写加法表达式的规则的时候，是这样写的：

 复制代码


```
1 add -> mul | add + mul
```

这是我们犯错之后所学到的知识。那么问题来了，大多数二元运算都是左结合的，那岂不是都要面临左递归问题？不用担心，我们可以通过改写左递归的文法，解决这个问题。

消除左递归

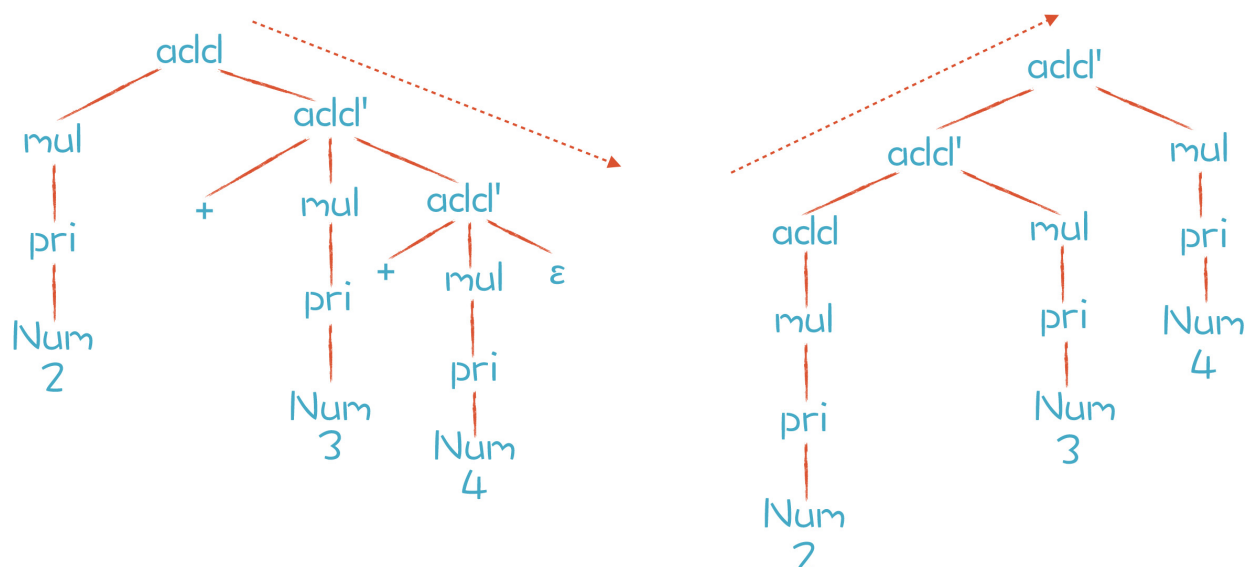
我提到过左递归的情况，也指出递归下降算法不能处理左递归。这里我要补充一点，并不是所有的算法都不能处理左递归，对于另外一些算法，左递归是没有问题的，比如 LR 算法。

消除左递归，用一个标准的方法，就能够把左递归文法改写成非左递归的文法。以加法表达式规则为例，原来的文法是 “add -> add + mul”，现在我们改写成：

 复制代码

```
1 add -> mul add'
2 add' -> + mul add' | ε
```


文法中， ϵ （读作 epsilon）是空集的意思。接下来，我们用刚刚改写的规则再次推导一下“2+3+4”这个表达式，得到了下图中左边的结果：



左边的分析树是推导后的结果。问题是，由于 `add'` 的规则是右递归的，如果用标准的递归下降算法，我们会跟上一讲一样，又会出现运算符结合性的错误。我们期待的 AST 是右边的那棵，它的结合性才是正确的。那么有没有解决办法呢？

答案是有的。我们仔细分析一下上面语法规则的推导过程。只有第一步是按照 `add` 规则推导，之后都是按照 `add'` 规则推导，一直到结束。

如果用 EBNF 方式表达，也就是允许用 `*` 号和 `+` 号表示重复，上面两条规则可以合并成一条：

复制代码

```
1 add -> mul (+ mul)*
```

写成这样有什么好处呢？能够优化我们写算法的思路。对于 `(+ mul)*` 这部分，我们其实可以写成一个循环，而不是一次次的递归调用。伪代码如下：

```

1 mul();
2 while(next token is +){
3     mul()
4     createAddNode
5 }

```

我们扩展一下话题。在研究递归函数的时候，有一个概念叫做**尾递归**，尾递归函数的最后一句是递归地调用自身。

编译程序通常都会把尾递归转化为一个循环语句，使用的原理跟上面的伪代码是一样的。相对于递归调用来说，循环语句对系统资源的开销更低，因此，把尾递归转化为循环语句也是一种编译优化技术。

好了，我们继续左递归的话题。现在我们知道怎么写这种左递归的算法了，大概是下面的样子：

```

1 private SimpleASTNode additive(TokenReader tokens) throws Exception {
2     SimpleASTNode child1 = multiplicative(tokens); // 应用 add 规则
3     SimpleASTNode node = child1;
4     if (child1 != null) {
5         while (true) { // 循环应用 add'
6             Token token = tokens.peek();
7             if (token != null && (token.getType() == TokenType.Plus || token.getType() :
8                 token = tokens.read(); // 读出加号
9                 SimpleASTNode child2 = multiplicative(tokens); // 计算下级节点
10                node = new SimpleASTNode(ASTNodeType.Additive, token.getText());
11                node.addChild(child1); // 注意，新节点在顶层，保证正确的结合性
12                node.addChild(child2);
13                child1 = node;
14            } else {
15                break;
16            }
17        }
18    }
19    return node;
20 }

```

修改完后，再次运行语法分析器分析“2+3+4+5”，会得到正确的 AST：


```
1 Programm Calculator
2     AdditiveExp +
3         AdditiveExp +
4         AdditiveExp +
5             IntLiteral 2
6             IntLiteral 3
7             IntLiteral 4
8         IntLiteral 5
```

这样，我们就把左递归问题解决了。左递归问题是我们用递归下降算法写语法分析器遇到的最大的一只“拦路虎”。解决这只“拦路虎”以后，你的道路将会越来越平坦。

课程小结

今天我们针对优先级、结合性和左递归这三个问题做了更系统的研究。我来带你梳理一下本节课的重点知识：

优先级是通过在语法推导中的层次来决定的，优先级越低的，越先尝试推导。

结合性是跟左递归还是右递归有关的，左递归导致左结合，右递归导致右结合。

左递归可以通过改写语法规则来避免，而改写后的语法又可以表达成简洁的 EBNF 格式，从而启发我们用循环代替右递归。

为了研究和解决这三个问题，我们还特别介绍了语法规则的产生式写法以及 BNF、EBNF 写法。在后面的课程中我们会不断用到这个技能，还会用工具来生成语法分析器，我们提供给工具的就是书写良好的语法规则。

到目前为止，你已经闯过了语法分析中比较难的一关。再增加一些其他的语法，你就可以实现出一个简单的脚本语言了！

一课一思

本节课提到了语法的优先级、结合性。那么，你能否梳理一下你熟悉的语言的运算优先级？你能说出更多的左结合、右结合的例子吗？可以在留言区与大家一起交流。

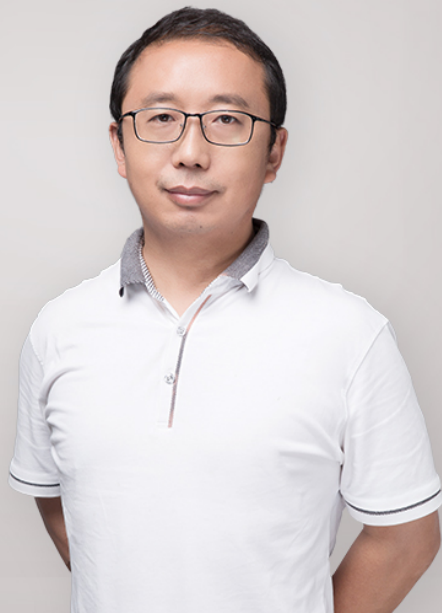
最后，感谢你的阅读，如果这篇文章让你有所收获，也欢迎你将它分享给更多的朋友。

编译原理之美

手把手教你实现一个编译器

宫文学

北京物演科技CEO



新版升级：点击「👤 请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 03 | 语法分析（一）：纯手工打造公式计算器

下一篇 05 | 语法分析（三）：实现一门简单的脚本语言

精选留言 (14)

写留言



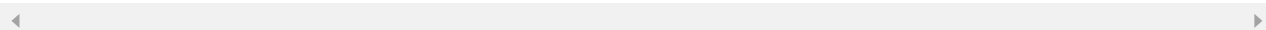
贾献华

2019-08-20

<https://github.com/iOSDevLog/Logo>

Swift 版《编译原理之美》代码，可以在 iOS 上运行。

作者回复：厉害！点赞！



1

3



pwlazy

2019-08-21

2+3+4+5生产的AST 是否是这样的？

Programm Calculator

AdditiveExp + ...

展开 ▾

作者回复: 是, 没错。

1 2



春去春又来

2019-08-21

老师 上一讲看懂了 这一讲在推导公式的时候迷糊了。可以加点推导过程的详细讲解嘛 而不是直接给一个推导的结果图

展开 ▾

作者回复: 好的, 我对于公式推导过程再加个图。加完了在回复中告诉你。

你指的是用:

$\text{add} \rightarrow \text{mul add}'$

$\text{add}' \rightarrow + \text{mul add}' \mid \epsilon$

来推导 $2+3+4$ 的过程不清楚吗?

2



半桶水

2019-08-21

是否可以给一些扩展资料的链接, 有些概念, 推导还是需要更多资料和练习才能掌握

作者回复: 如果想练习语法规则的推导, 那么随便买哪本教材都可以。一般也都会带些练习。

其他的扩展资料, 我后面有想到的, 会提供链接。

1



秋成

2019-08-25

没明白替换规则是为了做什么 为什么替换规则是这样的

$\text{add} \rightarrow \text{mul} \mid \text{add} + \text{mul}$

$\text{mul} \rightarrow \text{pri} \mid \text{mul} * \text{pri}$
 $\text{pri} \rightarrow \text{Id} \mid \text{Num} \mid (\text{add})$

作者回复: 这是文法理论的核心逻辑。

1. 替换规则是为了做什么？

替换过程，就是推导过程。这样不断替换，就是不断推导。

如果某个句子，能用某个文法推导出来，那就说这个句子符合某个文法。比如 $2+3*5$ 就符合我们上面的文法。

我们说语法解析，实际上是语法推导的反过程，是把它怎么推导的过程给逆向出来。

2. 为什么替换规则是这样的？

这就是文法设计的问题。这要根据问题域的特征来设计。比如，假设你为汉语设计一个文法，那么就知道分成“主谓宾”三个部分。而表达式是分为加减乘数运算，又分成优先级，所以就用上面的规则来表达。验证的方法就是看能否推导出所有可能的表达式。



秋成

2019-08-25

没明白 左递归的概念 "加法表达式 + 乘法表达式"

展开 ∨

作者回复: 动手改一下我们的示例代码，针对左递归的文法运行一下递归下降算法，就会知道会出现什么问题了。

基于你提的问题，我强烈建议你动动手。



中年男子

2019-08-23

优先级那里 $\text{exp} \rightarrow \text{or} \mid \text{or} = \text{exp}$ 是不是应该是 $\text{exp} \rightarrow \text{or} \mid \text{or} = \mid \text{exp}$?

上一讲中消除左递归，导致了结合性问题，这一讲再通过需要解决结合性问题，引出用循环来消除左递归。课程循序渐进，感觉老师说的直觉在一点点建立起来

有一个问题： $\text{add} \rightarrow \text{add} + \text{mul}$ 是怎么改成

$\text{add} \rightarrow \text{mul add}' \dots$

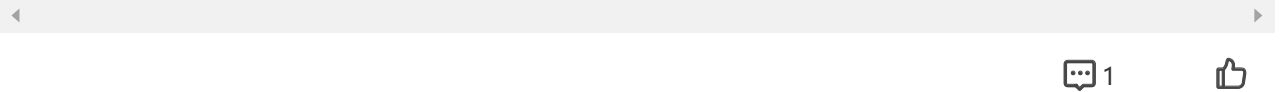
展开 ∨

作者回复: 看到你的进步很高兴！

mul 是 add' 的下一级，是乘法。它也可以像加法一样做类似的变换。

add'很容易解析这样的Token串： $+ 2 + 3 + 4 \dots$

或者： $+ 2 + 3 * 5 + 4 \dots$



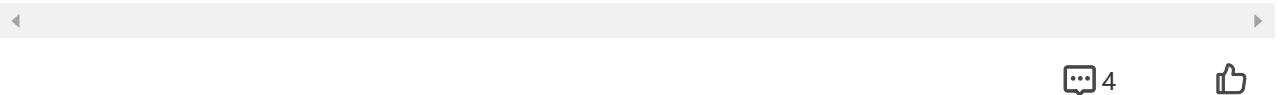
杨涛

2019-08-23

$\text{exp} \rightarrow \text{or} \mid \text{or} = \text{exp}$ 老师，一个逻辑运算=一个表达式是什么意思？前端表示看不懂！

作者回复: 那个等号，我省略了引号，是一个Token。

也就是一个or表达式，跟等号token，跟另一个表达式。

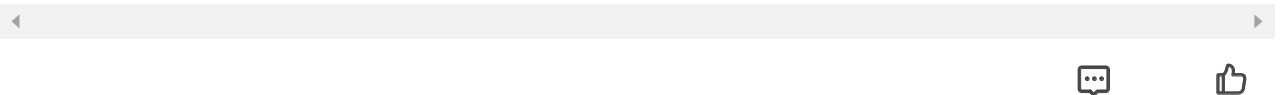


Giacomo

2019-08-23

老师老师，BNF能表达无限的情况吗？因为我发现EBNF是可以的，但你又说它们是等价的

作者回复: BNF带上递归就可以表达无限。BNF本身不行。但因为它是放在文法里，而文法是可以递归嵌套的。



谱写未来

2019-08-21

只有第一步用add，接下来都用add'，后面不是都是add'了，还是左边那张图不是吗？

作者回复: 是的。

我们通过改写规则的方法，能够避免左递归，但无法同时照顾结合性。这是很多教科书都没有提到的一件事情。

好在，这个事情比较简单，因为改写后的规则，是多了一个标准的“尾巴”。对，很多人都称呼它为尾巴。这个尾巴可以特别处理。

也就是说，结合性的信息已经不是单纯通过上下文无关文法提供了，要辅助额外的信息。

无独有偶，还有的作者用别的方法来解决算法优先级问题，比如LLVM的一个初学者教程，用的也

是标注算符优先级的方法，也要在文法的基础上提供额外的信息给算法。

<http://llvm.org/docs/tutorial/MyFirstLanguageFrontend/index.html>

本课程讲究实践。在实践中才会看到这些教科书上讲不到的点，但在面对实际问题时必须解决。



许童童

2019-08-21

老师可以说一下生成出来的AST怎么使用吗？

<https://github.com/jamiebuilds/the-super-tiny-compiler>

这个编译器写得怎么样，老师可以说一下吗？

作者回复: AST是对计算机语言的结构化表示，它是一切后续工作的基础，比如做语义分析，翻译成目标代码。

看了一下你发的那个链接。是从类似lisp语言的函数调用翻译到C语言的格式。这属于语言翻译的范畴。

我有两点点评：

- 1.lisp语言很容易翻译，一个递归下降算法肯定搞定。因为它的语法结构很简单，所有的语法结构都是一层层括号的嵌套。
- 2.翻译后得到AST，再生成C的格式，这就很简单了。基本上就是把括号位置改一下而已。

感谢你经常参与讨论！



Void_seT

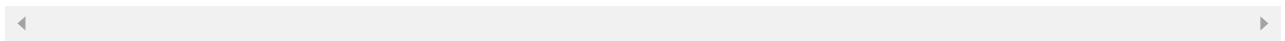
2019-08-21

介绍优先级的层级表示时，BNF的表达式最后一行
`mul -> pri | mul * pri | add / mul`，是不是应该是
`mul -> pri | mul * pri | mul / pri` 啊？

展开 ∨

作者回复: 唉，从加法规则里整行拷贝，修改时又少修改了一个地方。

已经在调整了。谢谢你仔细的阅读！



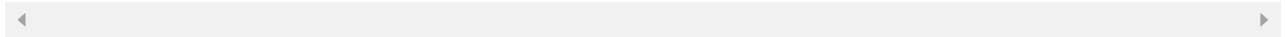
Varphp

2019-08-21

Bnf和ebnf使用会有冲突吗？还是类似于es和js的

展开 ▾

作者回复: 没有冲突。ebnf是bnf的超级。



w1sl1y

2019-08-21

老师，新代码中child2判空没加，不合法表达式会有空指针异常。

```
SimpleASTNode child2 = multiplicative(tokens);
    if (child2 != null) {
        node = new SimpleASTNode(ASTNodeType.Additive,token.getText
());...
```

展开 ▾

作者回复: 你说的对。要像前一节的代码一样判断一下child2是否为空。

代码库已经调整过来了！感谢！

我又加了两个测试用例，来检查这种处理错误语法的逻辑。你再看看！

