

04 | 语法分析（二）：解决二元表达式中的难点

2019-08-21 宫文学 来自北京

《编译原理之美》



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。

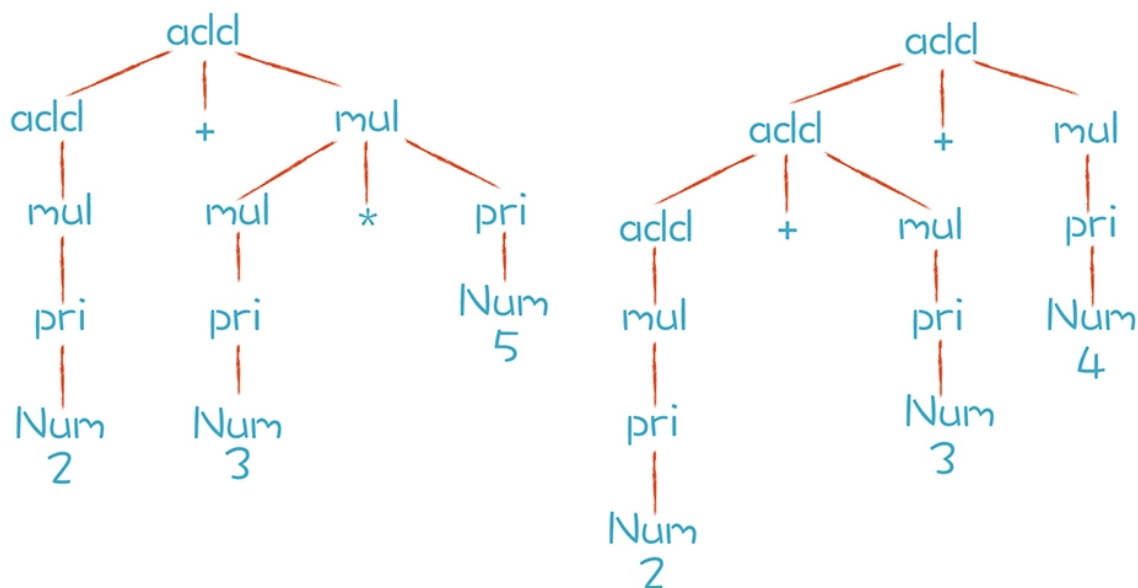
书写语法规则，并进行推导

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：

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

 复制代码

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：

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

 复制代码


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 中的乘法节点一定會在加法节点的下层，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：


```
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
```

 复制代码

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

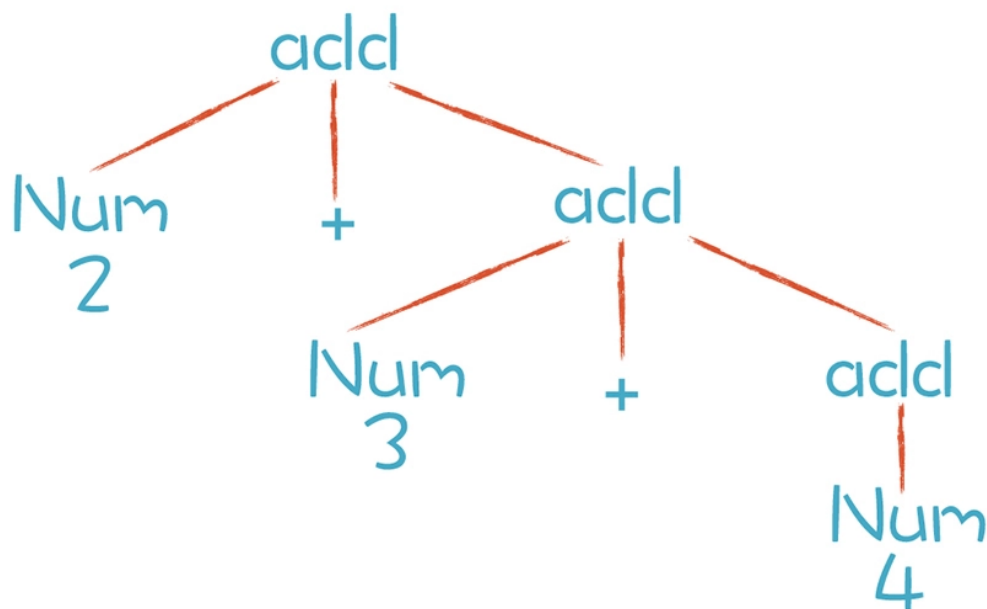
在上一讲中，我针对算术表达式写的第二个文法是错的，因为它的计算顺序是错的。

“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：

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

复制代码

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

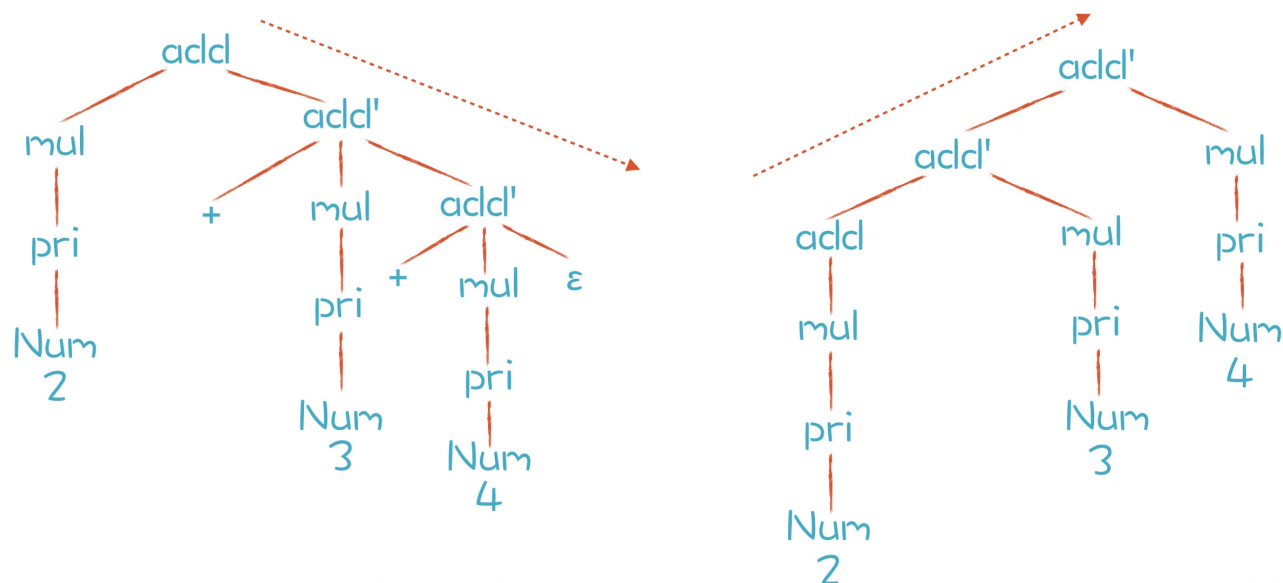
我提到过左递归的情况，也指出递归下降算法不能处理左递归。这里我要补充一点，并不是所有的算法都不能处理左递归，对于另外一些算法，左递归是没有问题的，比如 LR 算法。

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：


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

[复制代码](#)

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:

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

[复制代码](#)

写成这样有什么好处呢？能够优化我们写算法的思路。对于 $(+ \text{ mul})^*$ 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：

 复制代码

```
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：


 复制代码

```
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.getT
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
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：

 复制代码

```
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。

© 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。

精选留言 (56)



blackhole

2020-02-19

产生式 $add \rightarrow add + mul \mid mul$ 是如何改写成产生式 $add \rightarrow mul\ add'$ 和 $add' \rightarrow +\ mul\ add' \mid \epsilon$ 的，文中并未交代，评论中有人提出来了这个问题，老师依然没有回答。

我查阅了“龙书”，找到了答案。我的理解是这样的：

1, add 有两个产生式：① $add \rightarrow add + mul$ ，② $add \rightarrow mul$ 。如果只使用①来推导 add ，那么推导过程无法终结，会一直持续下去，形成 $add + mul + mul \dots + mul$ 这样的序列。因为不会有无限长的表达式，所以，推导过程必然会使用到②，且是在最后一步使用。也即，使用①②推导 add 后得到的序列最左边为 mul 。因而， add 的产生式可以改写为 $add \rightarrow mul\ add'$ 。

2, add' 的产生式需要满足 $+ mul + mul \dots + mul$ 这样的序列，所以可以写为 $add' \rightarrow + mul\ add'$ 。因为序列长度必然有限，所以，需要再加一个产生式以终结 add' 的推导过程： $add' \rightarrow \epsilon$ 。

作者回复：对。总结得很好！描述得很直观。

改写左递归的算法，实际上是有公式的。限于篇幅，我没有去陷入这个公式的细节。

在编译原理里面还有很多这样的细节。我希望能后续出的书里都包含到，并且仍然保持容易理解。

共 7 条评论 >

👍 65



毕达哥瓦斯

2020-10-25

希望老师多讲讲背后的原理和为什么会想到这么做

比如为什么想到改文法来消除左递归，为什么会想到ebnf

作者回复：你往深里又想了一层，探究背后的why，这很好，值得肯定！所以，我也拿出比较多的篇幅来回复你的问题。

1.为什么可以改写文法。

这其实有个前提，就是存在多个文法能生成相同语句。你可能有这样的经验，当需要写一个正则表达式来匹配字符串的时候，你能写出多个等价的正则表达式。对于上下文无关文法也一样，存在多个文法，能生成相同模式的语句。

既然存在等价的文法，那么自然可以去选择一个文法，能够更好的与某个算法去适配。LL算法是不能处理左递归的，那么就找到一个等价的文法，并且避免左递归就好了。

需要注意的是，虽然多个文法可以生成相同的语句，但是生成过程是不一样的。这也就导致解析树是不一样的。所以，有时候需要把解析树重新变换，来生成AST。

2.为什么想到EBNF

EBNF实际上等价于产生式，只不过写法不一样而已。实际上有很多跟EBNF等价的文法书写方式。它们都是用来描述一种语言的结构，或者是一种文档（如XML文档）的结构，所以它们也被叫做元语言（Metalinguage）。我在后面的元编程一章对Meta的级别有阐述，你也可以看一下。

所以，你的问题实际就变成了，为什么一个语言的文法会想到用产生式或者EBNF来描述。

实际上，一门语言不是必须用产生式或EBNF来描述的。有些类型的语言用其他方式描述更简单和方便，比如Indexed Language (https://en.wikipedia.org/wiki/Indexed_language)。不过，对于大多数计算机语言来说，用上下文无关文法描述是比较合适的，而上下文无关文法采用的是一种字符串重写规则（String Rewriting System, SRS），也就是把一个字符串中的一部分不断地替换成另外的字符串。采用这种工具没有别的原因，就是因为它在描述语言的语法方面是很有效的。如果你追求它的数学根基，你可以去看半图厄理论，在数理逻辑里有。

SRS这种工具出现的历史比较早，最早是用来研究自然语言的。后来，在逻辑学（作为哲学的一部分）、数学中也得到了广泛的使用。比如现代数学的公理化运动，也就是把数学（比如欧几里得几何）看做一个形式系统；把数学定理的推导，看做是一个纯粹的形式化的变换过程。所以，它首先需要一门形式化的语言来描述数学中的命题，然后再基于一套推导逻辑去变换它们。然后再来看是否在有限的时间内一定能够推导出来，这也就是图灵的停机问题。

总结起来，我们在编译原理里面用到了一些形式语言方面的工具，它和被数学、语言学、逻辑学等多个学科共享的。它们都认为，该学科被研究的对象某种意义上是一些纯形式的变换。这种严谨的形式变换的过程，构筑了西方现代科学的严密推理体系，是那么多科学发现的底层根基。从这个角度，你其实可以体会到编译原理搞的是很基础的东西，是这个世界的一些底层的思维逻辑。

再次非常肯定你的思考精神。通过这种思考，你可以越挖越深，这个过程非常有趣。而且，你挖到一定程度，会发现很多知识体系都是通着的。比如，通过今天的探讨，你已经知道现代数学和编译原理是通着的。顺着这条线，你还会发现更多通着的知识。比如，逻辑学和集成电路的底层是通着的；计算机的底层逻辑跟数学的底层逻辑是一回事；计算过程又跟物理学的某些原理是一回事。很有意思。

共 2 条评论 >

👍 29



Enzo

2019-09-06

老师 看不懂以下的公式

add -> mul | add + mul

mul -> pri | mul * pri

pri -> Id | Num | (add)

是需要找本书看看吗？

作者回复：我给你解释一下吧：

以 add -> mul | add + mul 为例，

-> 意思是推导出；

| 意思是“或者”

这个加号，我回头修改一下吧，可以用引号引起来， '+' 只是匹配一个 + 号字符的意思，没有别的意思。

所以，这个产生式的意思是：

加法表达式，要么是一个乘法表达式，要么是一个加法表达式，后面跟个 + 号，然后再跟一个乘法表达式。

参考书的话，看看这个链接：<https://time.geekbang.org/column/article/125948>

共 3 条评论 >

👍 23



Lafite

2019-09-03

请问宫老师

add -> mul add'

`add' -> + mul add' | ε`

这两个产生式的推导过程应该是怎么样的，为什么可以转化为EBNF的写法呢。

作者回复: 转化成EBNF: `add ::= mul ('+' mul)*`

一个表达式就解决了，更简洁。不需要`add'`了。

推导过程，要看算法。每种算法采用的推导过程是不一样的。如果用递归下降算法，推导：`2+3*5`
我们按照调用过程分成几层：

第1层：采用 `mul add'`，因为`mul`能完整的匹配`2`，不能再往后匹配了，所以第一个子节点建立完毕。
接着用`add'`去建立第二个子节点。

第2层：运用`add'`的第一个产生式，先匹配上了`+`号，之后去匹配`mul`，也就是`3*5`，也是成功的。然后再去匹配`add'`。

第3层：这次用`add'`的时候，还是先尝试第一个产生式，失败。为什么呢？因为没有`+`号。回溯。尝试第二个产生式，即`epsilon`。也就是返回空。那么第3层就完成了。

第3层成功后第2层，第2层也就成功完成了。

同理，返回第1层，第1层也成功。

这个过程是否听得清楚？

可以换着不同的例子多推导几遍，就会变得很熟练了！

共 3 条评论 >

👍 16



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。

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。



15



knull

2019-08-28

老师，我简单研究了下bnf，我觉得你写法最好修正下，不然不好看。比如：

原来的写法：add -> mul (+ mul)*

现在的写法：add -> mul ('+' mul)*

'+'表示关键字；

+ 直接用，表示1个或多个；

加单引号以示区分，看起来方便一点

作者回复：有道理！用EBNF的话，+号有特殊含义。

我们修改一下文稿。

谢谢你的建议，你很细心，并且自己去做研究了！



11



xxx

2021-02-15

左递归这块确实蛮烧脑，总结一下吧：

1. 左递归会造成无限递归，从而造成递归下降法无法结束。
2. 可以将左递归改成右递归，这样便能够结束。但结合性会出现问题。
3. 改成右递归之后，就成了尾递归，那么可以用循环代替递归。而这里不是为了优化性能，而是为了修改行为！（本来直接代替后应该是后面的token产生的树会成为前面的子树，但我们就是硬改为后面产生的树变成根）

作者回复：嗯，你总结出了其中的要点！

后面的课程里，还有别的方法来破除这些障碍，比如LR算法不怕左递归。

在另一门课，《编译原理实战课》中，你会看到常用语言其实用一个很优雅的运算符优先级算法就能解决常见的二元运算的表达式的解析问题。

共 2 条评论 >

9



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

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:

`add -> mul add'`

`add' -> + mul add' | ε`

来推导 $2+3+4$ 的过程不清楚吗?



👍 5



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!

共 2 条评论 >

👍 5



侯不住

2020-04-17

老师, 我需要帮助。。。.

`add ::= mul | add + mul`

`mul ::= pri | mul * pri`

`pri ::= Id | Num | (add)`

前面一讲 $2+3$ 这个表达式使用`add ::= mul | add + mul`这个产生式会有左递归的问题, 为何到了这一讲上面的产生式就能分析 $2+3*4$ 这样的表达式, 这应该跟前面的一样左递归就会有问题啊。不明白不明白, 求指导

共 2 条评论 >

👍 4



pwlazy

2019-08-21

$2+3+4+5$ 生产的AST 是否是这样的?

Programm Calculator

```
AdditiveExp +  
  AdditiveExp +  
    AdditiveExp +  
      IntLiteral 2  
      IntLiteral 3  
    IntLiteral 4  
  IntLiteral 5
```

作者回复: 是, 没错。

共 2 条评论 >

👍 3



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!



👍 3



草戊

2019-12-14

antlr4能处理直接左递归了, 表达式文法写起来直观很多

作者回复: 是的。



👍 2



nil

2019-09-11

老师你好，问个问题。最终通过循环来消除递归带来的二元预算符的结合性问题？能否直接在递归中消除结合性问题？

作者回复: 理论上是可以的，但需要给算法提供额外的信息。

采用递归下降算法的时候，我们在函数中标准的处理放肆，都是创建一个AST节点，并返回给调用者。调用者都是把返回的AST作为自己的子节点。

如果要改变结合性，相当于要知道什么时候把返回的节点作为自己的父节点。

Antlr里用属性标注的方法，来提供这个额外的信息。这种信息在标准的上下文无关文法中是无法提供的。



👍 2



Enzo

2019-09-06

```
exp -> or | or = exp
or -> and | or || and
and -> equal | and && equal
equal -> rel | equal == rel | equal != rel
rel -> add | rel > add | rel < add | rel >= add | rel <= add
add -> mul | add + mul | add - mul
mul -> pri | mul * pri | mul / pri
```

老师不懂这里的 + - >= 等符号的意思 能推荐本书 吗

作者回复: 所有的操作符，你可以加上引号，也就是去匹配这样一个字符而已。没有太复杂的意思。

```
add -> mul | add '+' mul | add '-' mul
```

参考书籍的话，参见这篇攻略：<https://time.geekbang.org/column/article/125948>



👍 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。



2



不会魔法
2020-07-27

不明白为啥上一节的

`add -> Int | add + Int` 不能匹配 `2+3`，会出现左递归，到了这一节

`add -> mul | add + mul` 又能匹配 `2+ 3 * 5` 中的 `2 + 3` 了

共 1 条评论 >



1



杨涛
2020-06-01

老师，请问`exp -> or | or = exp`这一句中`or = exp`对应出来是一个如何的表达式呢？

作者回复: 是赋值表达式。

你可以注意一个细节：这里递归项（`exp`）是在右边的，而后面的其他产生式，递归项是在左边的。

你知不知道为什么？

共 2 条评论 >



1



简玉
2019-12-23

学习这门课的时候 结合前后和留言问答 会好理解一些

作者回复: 嗯。留言中别人遇到的问题，对自己也会有用。你如果有问题的话，也多提问！



1



阿尔伯特
2019-09-11

<https://github.com/albertabc/compiler>

继续攒代码。

有了上节课的基础，这节相对比较容易理解。用文法的形式推导，最终消除了左递归的思路我觉得很有意思，用左右递归代表结合性，用文法上下级实现了优先级，这些可以作为解决问题的一个思路和方法，用比较平常的普通方法解决了这些问题。感觉比中缀后缀表达式容易掌握。

作者回复: 嗯。编译且运行了一下。



1