03｜支持表达式：解析表达式和解析语句有什么不同？

2021-08-13 宫文学

《手把手带你写一门编程语言》 课程介绍 >

讲述：宫文学

时长 15:52 大小 14.54M

你好，我是宫文学。

到目前为止，我们已经学习了一些语法分析的算法。不过，我们主要是分析了如何来解析语句，比如函数声明、函数调用，没有把重点放在解析表达式上。

其实我是刻意为之的，故意把表达式的解析往后推迟一下。原因是表达式解析，特别是像“2+3*5”这样的看似特别简单的二元运算的表达式解析，涉及的语法分析技术反而是比较复杂的。所以，从循序渐进的角度来说，我们要把它们放在后面。

表达式的解析复杂在哪里呢？是这样，我们在解析二元表达式的时候，会遇到递归下降算法最大的短板，也就是不支持左递归的文法。如果遇到左递归的文法，会出现无限循环的情况。

在这一节里，我会给你分析这种左递归的困境，借此加深你对递归下降算法运算过程的理解。

同时，我也要给出避免左递归问题的方法。这里，我没有采用教科书上经常推荐的改写文法的方法，而是使用了业界实际编译器中更常用的算法：**运算符优先级解析器（Operator-precedence parser）**。JDK 的 Java 编译器、V8 的 JavaScript 编译器和 Go 语言的 GC 编译器，都毫无例外地采用了这个算法，所以这个算法非常值得我们掌握。

好了，那我们首先来了解一下用递归下降算法解析算术表达式会出现的这个左递归问题。

左递归问题

我们先给出一种简化的加法表达式的语法规则：

复制代码

```
add : add '+' IntLiteral
    | IntLiteral
    ;
```

对这个规则的解读是这样的：一个加法表达式，它要么是一个整型字面量，要么是另一个加法表达式再加上一个整型字面量。在这个规则下，2、2+3、2+3+4 都是合格的加法表达式。

那如果用递归下降算法去解析 2+3，我们会采用“add ‘+’ IntLiteral”的规则。而这个规则呢，又要求匹配出一个 add 来，从而算法又会递归地再次调用“add ‘+’ IntLiteral”规则，导致无限递归下去。

复制代码

```
2+3是不是一个add表达式？
  ->先匹配出一个add表达式来，再是+号，再是整型字面量
    ->先匹配出一个add表达式来，再是+号，再是整型字面量
      ->先匹配出一个add表达式来，再是+号，再是整型字面量
        ->无限递归...
```

这就是著名的**左递归问题**，是递归下降算法或者 LL 算法都无法解决的。

你可能会问，如果把产生式的写法换一下，把 add 放在后面，不就会避免左递归了吗？

```
add : IntLiteral '+' add
    | IntLiteral
    ;
```

这个也是不行的，因为这样会导致运算的结合性出错。如果执意按照这个语法解析，解析 2+3+4 这个表达式所形成的 AST 会是右结合的：

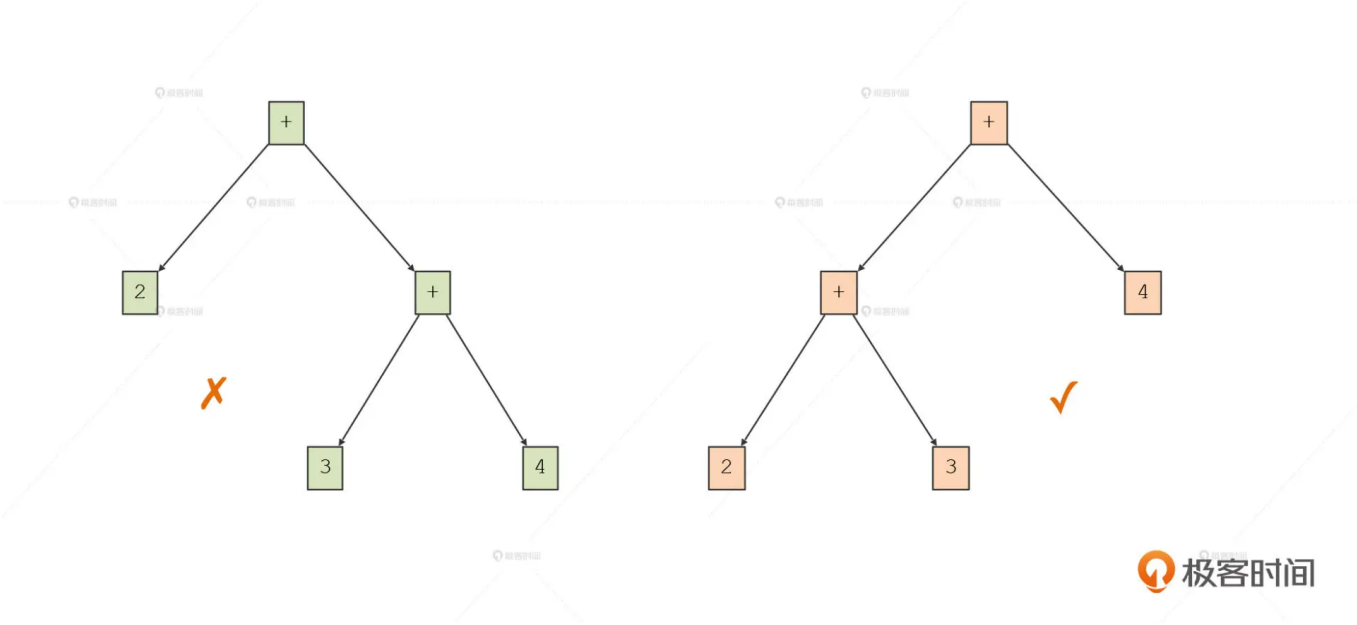

图1：基于右递归语法会生成右结合的AST

你会看到，基于使用右递归文法生成的 AST，实际是先计算 3+4，再跟 2 相加。这违背了加法运算的结合性的规定。正确的运算顺序，应该是先计算 2+3，然后再加上 4，是左结合，对应的 AST 应该是右边的那个。这种结合性的错误，看上去对于加法影响不大，但如果换成减法或者除法，那计算结果就完全错误了。

好了，现在你已经理解了左递归问题了。那我们要如何解决这个问题呢？一个可行的解决方法就是改写文法，并且要在解析算法上做一些特殊的处理，你可以参考《编译原理之美》课程04 讲。除了改写文法的方法以外，还有一些研究者提出了其他一些算法，也能解决左递归问题。

不过，针对二元表达式的解析，今天我要采用的是被实际编译器广泛采用的**运算符优先级算法**。

运算符优先级算法

这是个怎么样的算法呢？我想先用简单的方式帮你理解运算符优先级算法的原理，然后再一步步深化。

在01 讲介绍递归下降算法的时候，我提到，它对应的是人类的一种思维方式，也就是从顶向下逐步分解。但人类还有另一种思维方式：自底向上逐步归纳。**而运算符优先级算法，对应的就是自底向上的一种思维方式。**

首先我们来看 2+3+4 这个表达式，如果我们用自底向上归纳的思路做语法分析是怎么样一个思考过程呢？

第 1 步，首先看到 2。你心里想，这里有一个整数了，那它是不是一个算术表达式的组成部分呀？是一个加法表达式的，还是乘法表达式的一部分呢？我们再往下看一看就知道。

第 2 步，看到一个 + 号。噢，你说，原来是一个加法表达式呀。这时候，我们知道，2 肯定是要参与加法运算的，所以是加法的左子树。但加法后面可以跟很多东西的，比如另一个整数，或者是一个乘法表达式什么的，都有可能。那我们继续向下看。

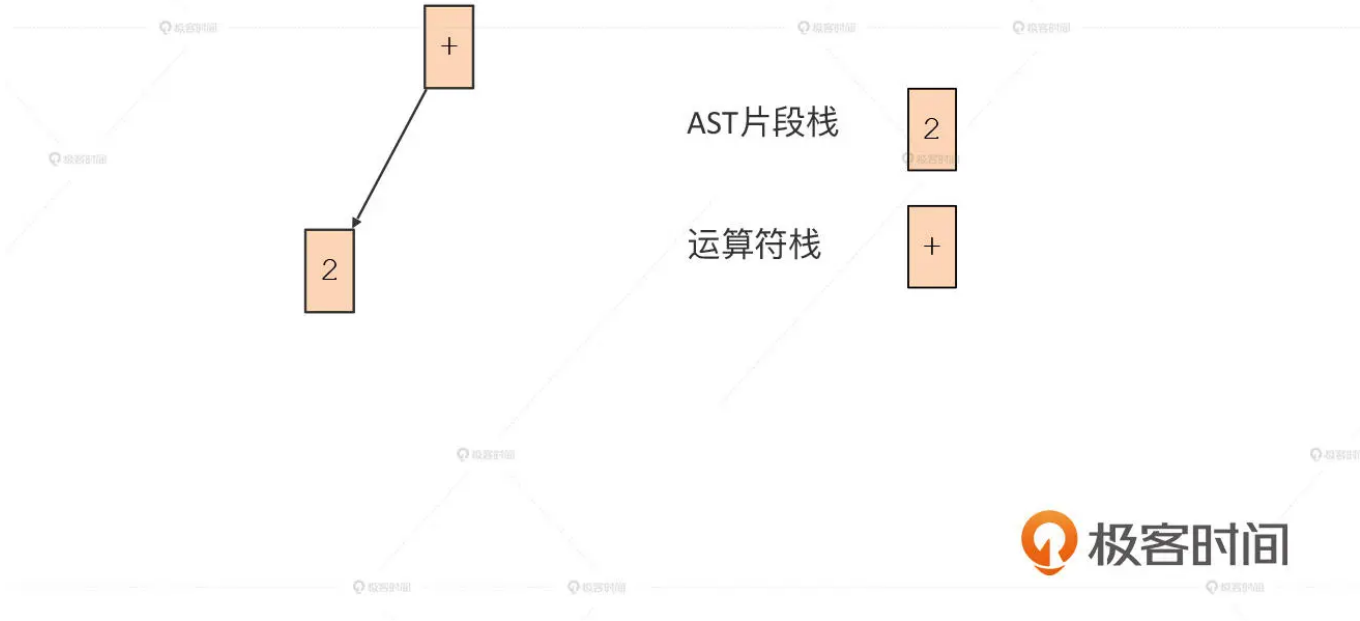

图2：第2步时，2是肯定与+号结合的

第 3 步，看到整数 3。奥，你心里想，原来是 2+3 呀。那我现在根据这三个 Token 是不是可以先凑出一棵 AST 的子树来呢？先等一等，我们现在还不知道 3 后面跟的是什么。

如果 3 后面跟的是 + 号或者 - 号，那没问题，3 是先参与前面这个 + 号的计算，再把 2+3 的结果一起，去参与后面的计算的。

但如果 3 后面遇到的是 * 号呢？那么 3 就要先参与乘法运算，计算完的才参与前面的加法的。这两个不同的计算顺序，导致 AST 的结构是不一样的，**而影响 AST 结构的，其实就是 3 前后的两个运算符的优先级。**

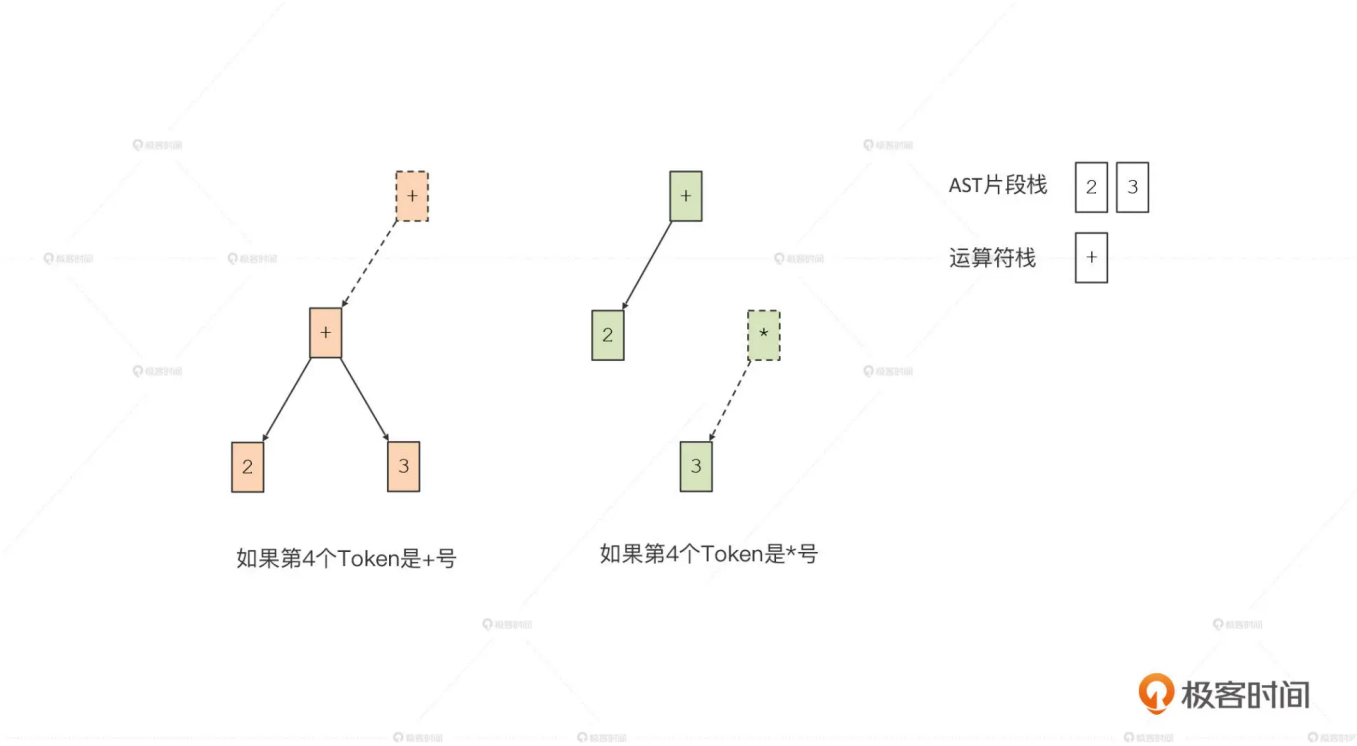

图3：根据第4个Token的优先级，3会与不同的运算符结合

第 4 步，看到第 2 个 + 号。这个时候，你心里知道了，原来 3 后面的运算符的优先级跟前面的是一样的呀，那么按照结合性的规定，应该先算前面的加法，再算后面的加法，所以 3 应该跟前面的 2 和 + 号一起凑成一个 AST 子树。并且，这棵子树会作为一个稳定的单元，参与后面的 AST 的构建。

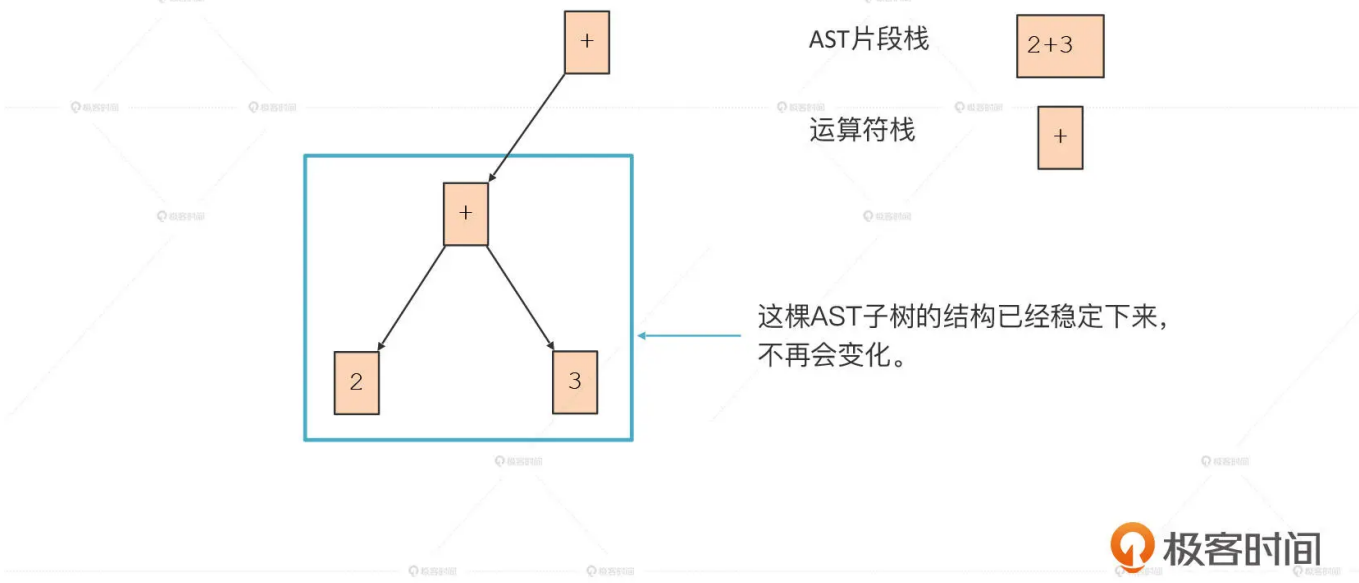

图4：2+3对应的AST子树已经稳定，不会被拆散了

第 5 步，看到整数 4。现在的情况跟第 3 步是一样的，我们不知道 4 后面跟着的是什么。如果 4 后面跟着一个 * 号，那么 4 还要先参与后面的计算，然后再跟前面这一堆做加法。如果 4 后面也是一个加法运算符，那 4 就要先参与前面的计算，4 在 AST 中的位置也就会变得确定。

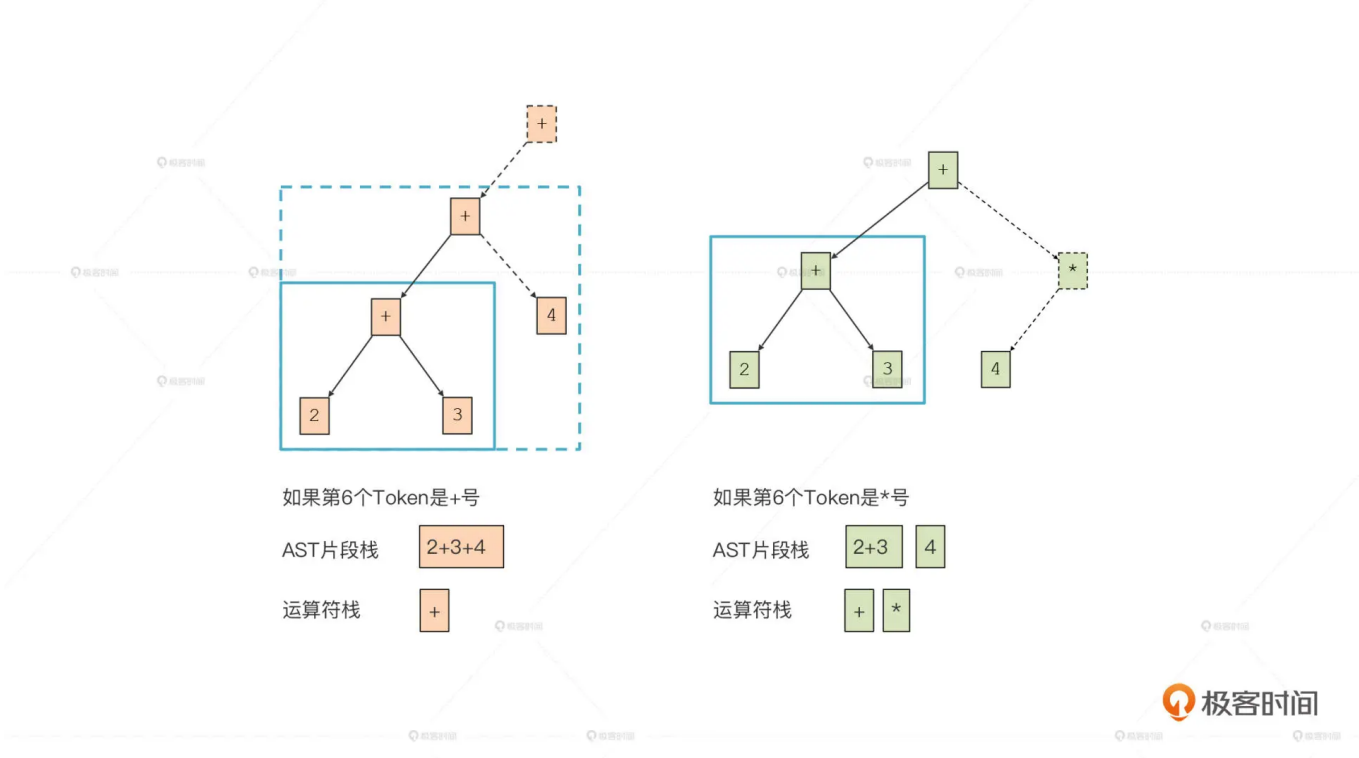

图5：根据第6个Token的运算符的优先级，AST结构会不同

第 6 步，再往下看，发现后面的 Token 既不是 + 号，也不是 * 号，而是 EOF，也就是 Token 串的结尾。这样的话，整个 AST 就可以确定下来了。

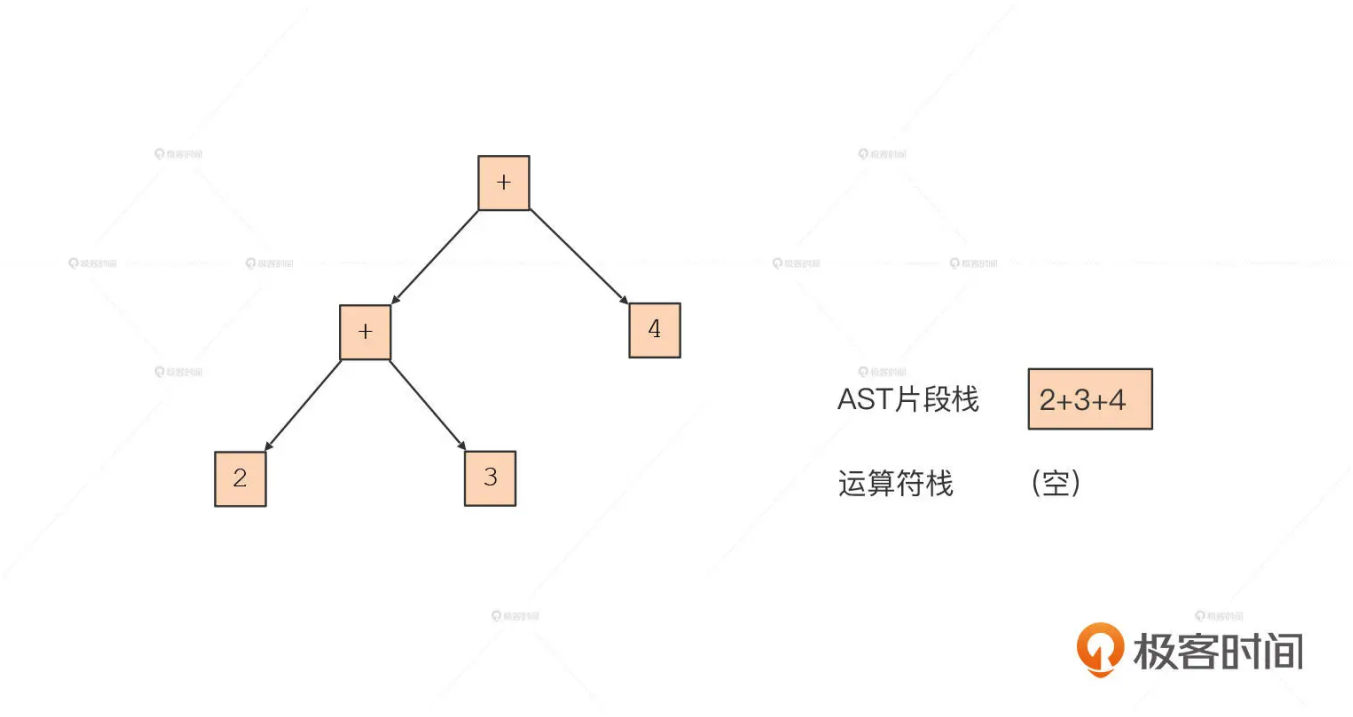

图6：遇到EOF后，解析结束

好了，这是一个比较简单的算法运行的场景。你可以多读几遍，借此找找自底向上分析的直观感觉。

接下来，我们再换一个任务，分析一下 2+3*5。你会发现，跟前一个例子相比，一直到第 5 步的时候，也就是读入了 5 以后，仍然没有形成一棵稳定的 AST 子树：

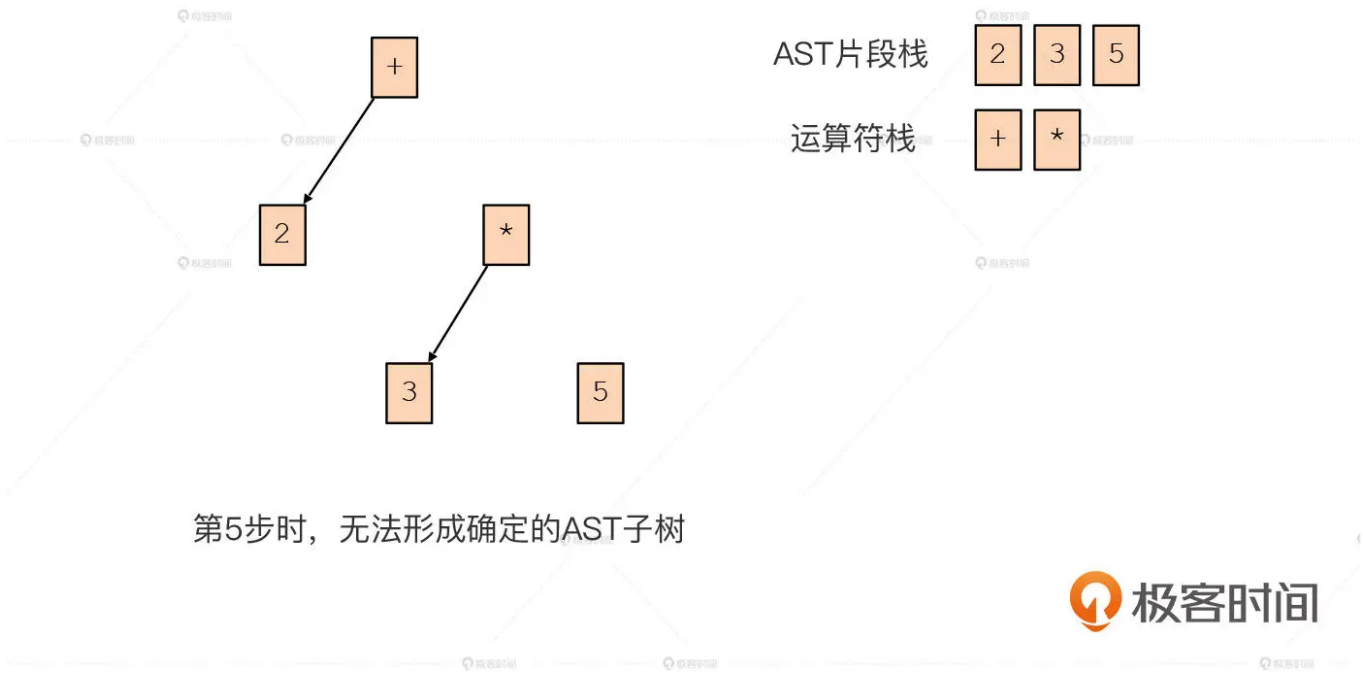

图7：第5步时的AST结构

这是为什么呢？

因为根据 5 后面读入的 Token 的不同，形成的 AST 的结构会有很大的区别。这里我们展示 3 种情形：

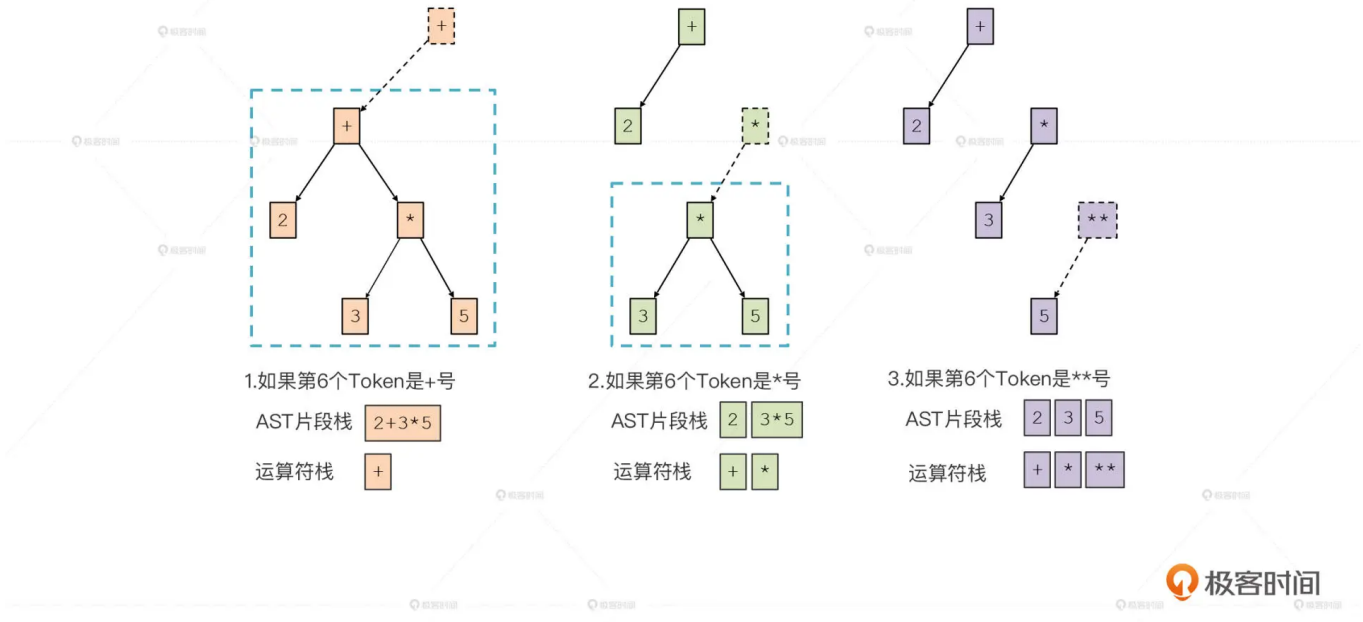

图8：AST的结构取决于第6个Token的运算符优先级

情形 1，第 6 个 Token 是 + 号：它的优先级不高于最后一个运算符 * 号，所以 3 * 5 这棵子树的结构就是确定的；进一步看，它也不高于第一个运算符的优先级，所以整个 2+3 * 5 这棵子树的结构都可以确定下来，并且肯定是最后一个 + 号的左子树。

情形 2，第 6 个 Token 是 * 号：这时候它的优先级仍然不高于最后一个运算符 * 号，所以 3 * 5 这棵子树的结构也可以确定下来，并成为新的 * 号的左子树。

情形 3，如果第 6 个 Token 是 ** 号：也就是做幂运算，那么 5 就会首先参与幂运算，而不是参与前面的乘法运算。

当然，还有最后一种情形，就是**第六个 Token 是 EOF**：这会跟第一种情形相似，形成一棵确定的 AST。

好了，这就是对第二个例子的分析。它比第一个例子要复杂一些，也能让你对运算符优先级算法的理解更深化。

目前我们的表达式，用到了两个优先级的运算符。你还可以增加更多的优先级进来，比如“2+3 * 5>10”这个关系表达式，有 3 个优先级；“2+3 * 5>10 && ture”是一个逻辑表达式，有 4 个优先级。

那么怎么把它实现成算法呢？其实我在上面的图里已经有所铺垫。在每个图里，我都画了两个工作区，也就是两个栈。一个栈用来放运算符，一个栈用来放待装配的 AST 片段。

算法的设计也很简单，每次新取出一个运算符的时候，都跟栈顶的运算符做比较。如果新的运算符的优先级高于栈顶的运算符，就把该运算符继续压栈；否则，就从栈里弹出所存的运算符，并把它跟 AST 片段栈中的元素组合，变成一棵新的 AST 子树，重新压入 AST 片段栈。怎么样，很简单吧？

从上面的描述中，你还能得出一个推论：**运算符栈里的元素，总是一个比一个的优先级高。**

具体实现算法的时候，可以按照刚才描述的逻辑，用两个栈作为工作区，来实现解析。不过，你可能知道，基于栈的算法往往有等价的递归算法，而且递归算法会更简洁。这里我

把等价的递归算法写出来。你可以用这个算法把前面的两个例子推演一遍，看看它们为什么是等价的。

复制代码

```
/**
 * 解析一个二元表达式，形成一棵AST子树，其根节点的优先级不超过prec
 */
parseBinary(prec:number):Expression|null{
    // 首先解析一个基础表达式，作为左子节点
    let exp1 = this.parsePrimary();
    if (exp1 != null){
        //预读运算符
        let t = this.scanner.peek();
        //获取运算符的优先级
        let tprec = this.getPrec(t.text);
        //只要优先级比当前要求的优先级大
        while (t.kind == TokenKind.Operator &&  tprec > prec){
            this.scanner.next();  //跳过运算符
            //针对优先级更高的运算符，获取一棵AST子树
            let exp2 = this.parseBinary(tprec);
            if (exp2 != null){
                //创建一棵新的AST，把刚刚获取的AST子树作为右子树
                let exp:Binary = new Binary(t.text, exp1, exp2);
                exp1 = exp;
                t = this.scanner.peek()
                tprec = this.getPrec(t.text);
            }
            else{//报编译错误}
        }
        return exp1;
    }
    else{//报编译错误}
}
```

在实际的编译器中，有的采用的是基于栈的算法，如 JDK（以 JDK14 为例）；有的采用的是递归算法，如 Go 语言编译器（以 1.14.2 版本为例）。你可以根据自己的喜好采用。

注意，我们这节课只讨论了二元运算。而一元运算，比如 ++i，或 i++，用我们之前02讲中讲过的 LL 算法已经能够解决，你再去复习就好了。

掌握了运算符优先级算法以后，我们语法解析器实际上就混合了两种算法，主体是用 LL 算法，能够解析语句等各种成分；唯独留下二元表达式，用运算符优先级算法来实现。这也是像 Java、V8、Go 语言等这些编译器采用的较为成熟的实现方法。

在完成了这节课的主要任务之后，我再基于运算符优先级算法，把自底向上算法的知识面给你稍微扩展一下，这也让你能够基于更大的背景来理解运算符优先级算法。

了解 LR 算法

前面也说了，我们今天介绍的运算符优先级算法，属于自底向上的算法。而自底向上的算法，除了运算符优先级算法之外，最著名的就是 LR 算法家族。LR 算法在工作的时候和运算符优先级算法一样，都是采用一个工作区来组装 AST 的片段。其中读取 Token 的过程叫做 Shift，也就是移进；把工作区里的 Token 组装成 AST 片段的过程叫做 Reduce，也就是规约。所以，这种特点的算法又被叫做移进 - 规约算法。

说到 LR，你会想起上一节课，我还有一个名词没有给你展开介绍，就是 LL。那在这里，我结合 LR 一起给你解释一下。

LL 是中的第一个 L 是“Left-to-right” ” 的意思，代表从左向右处理输入的字符串；第二个 L，是“Left-most Derivation”的意思，也就是最左推导。

那什么是最左推导呢？就是对于语法规则而言，我们每次总展开最左边的非终结符，之后再处理右边的非终结符。

我们还是拿简化版的加法计算的文法做例子。为了便于推导，我这次采用了产生式的写法。此文法要求加法表达式必须用括号括起来，这样就是一个合格的 LL 文法，避免了左递归：

复制代码

```
1 add -> (add + add)
2 add -> IntLiteral
```

基于这个规则，如果要推导出“((2+3)+4)”这个串，推导顺序是：

复制代码

```
1 add -> (add + add)    //采用第一个产生式展开
2      //展开最左边的元素，仍然采用第一个产生式
3      -> ((add + add)+add)
4      //从左到右依次展开每个add，采用第二个产生式
5      -> ((IntLiteral + add) + add)
6      -> ((IntLiteral + IntLiteral) + add)
```

```
7      -> ((IntLiteral + IntLiteral) + IntLiteral)
```

理解了 LL 的意思，那么 LR 的意思你大概也能猜出来了。第一个字母 L 仍然是“Left-to-right”的意思，而第二字母 R 的意思是“Right-most Derivation”，也就是最右推导。你可以把上面的例子用最右推导来试一下其展开过程。

而 LR 算法的实际执行过程，是最右推导过程的逆过程。也就是从 ((IntLiteral + IntLiteral) + IntLiteral) 一步步的反向做规约，最后规约成一个 add 非终结符的过程。

如果你想了解 LR 算法的细节，可以看看《编译原理之美》的18 讲。

课程小结

今天这节课，我们借助解析二元表达式的任务，首先介绍了递归下降算法的一个短板，就是不能处理左递归语法。

接着，我们介绍了业界编译器为解决二元表达式的解析问题而普遍采用的一个算法，也就是运算符优先级算法。我希望你能像这节课里示范的一样，去推导解析二元表达式的过程，从而帮助你建立直觉认知。在建立了这种直觉认知以后，写成算法就不是难事了。借此，你也能触摸到所有自底向上的算法的思维方式，比如 LR 算法家族。

最后，我花了很小的篇幅，介绍了 LR 和 LL 这两个词汇的准确含义。再次强调一下，我们这门课没有专门花精力追求理论方面的全面性，而是更重视最佳实现技术。

思考题

在这节课中，我们提到了加减乘除等运算是左结合的。那么有没有右结合的运算呢？对于右结合的运算，我们应该如何实现呢？说说你的想法。你也可以看看我们本节的示例代码，看看跟你的想法是否一致。

欢迎你和我一起学习，如果你觉得这节课讲得还不错，也欢迎分享给更多感兴趣的朋友。我是宫文学，我们下节课见。

分享给需要的人，Ta的阅后你可得 20 元现金奖励

赞 0 提建议

上一篇 02 | 词法分析：识别Token也可以很简单吗？

下一篇 04 | 如何让我们的语言支持变量和类型？

精选留言 (2)

写留言

写点啥呢
2021-08-16

请问宫老师，在2+3+4这个例子中在判断4之后的运算符是*的情况所画的AST，感觉绿色AST将4*的AST节点和2+3的节点画在同一个+号节点的两个字节点是不对的，这会导致2+3先于4*x的表达式求值而违反运算符优先级

展开

1

有学识的兔子
2021-08-15

常见运算中，赋值运算和幂运算是符合右结合的。在计算ast时，碰到幂运算时多向前获取一个token，构建一个独立的ast片段，不受其它运算符的影响。