

03 | 语法分析（一）：纯手工打造公式计算器

2019-08-19 宫文学 来自北京

《编译原理之美》



我想你应该知道，公式是 Excel 电子表格软件的灵魂和核心。除此之外，在 HR 软件中，可以用公式自定义工资。而且，如果你要开发一款通用报表软件，也会大量用到自定义公式来计算报表上显示的数据。总而言之，很多高级一点儿的软件，都会用到自定义公式功能。

既然公式功能如此常见和重要，我们不妨实现一个公式计算器，给自己的软件添加自定义公式功能吧！

本节课将继续“手工打造”之旅，让你纯手工实现一个公式计算器，借此掌握**语法分析的原理**和**递归下降算法 (Recursive Descent Parsing)**，并初步了解上下文无关文法 (Context-free Grammar, CFG)。

我所举例的公式计算器支持加减乘除算术运算，比如支持“ $2 + 3 * 5$ ”的运算。

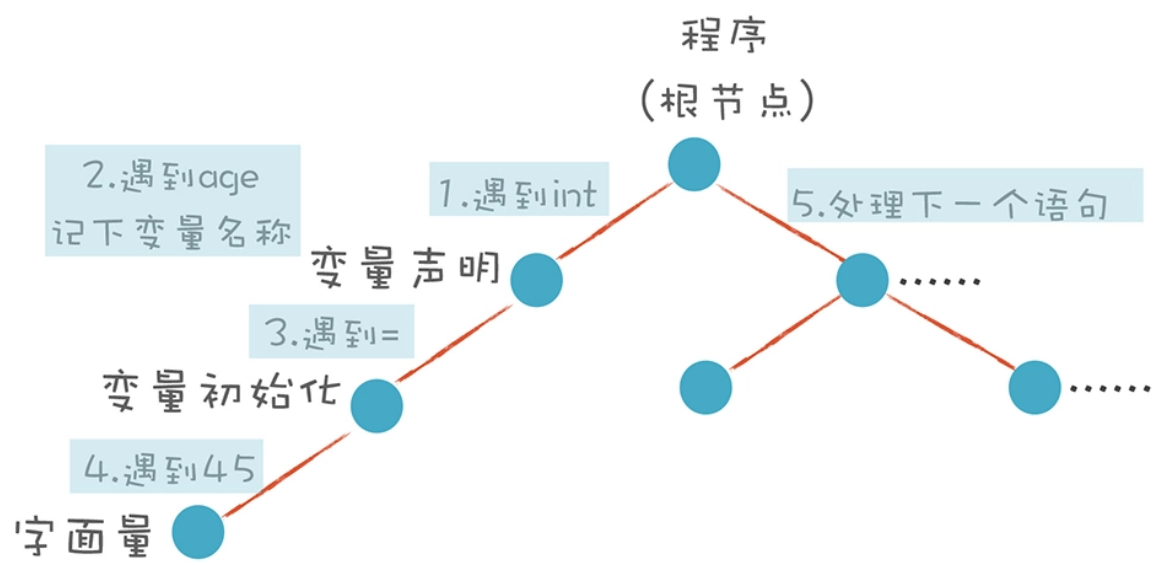
在学习语法分析时，我们习惯把上面的公式称为表达式。这个表达式看上去很简单，但你能借此学到很多语法分析的原理，例如左递归、优先级和结合性等问题。

当然了，要实现上面的表达式，你必须能分析它的语法。不过在此之前，我想先带你解析一下变量声明语句的语法，以便让你循序渐进地掌握语法分析。

解析变量声明语句：理解“下降”的含义

在“[01 | 理解代码：编译器的前端技术](#)”里，我提到语法分析的结果是生成 AST。算法分为自顶向下和自底向上算法，其中，递归下降算法是一种常见的自顶向下算法。

与此同时，我给出了一个简单的代码示例，也针对“`int age = 45`”这个语句，画了一个语法分析算法的示意图：



我们首先把变量声明语句的规则，用形式化的方法表达一下。它的左边是一个非终结符（Non-terminal）。右边是它的产生式（Production Rule）。在语法解析的过程中，左边会被右边替代。如果替代之后还有非终结符，那么继续这个替代过程，直到最后全部都是终结符（Terminal），也就是 Token。只有终结符才可以成为 AST 的叶子节点。这个过程，也叫做推导（Derivation）过程：

```
1 intDeclaration : Int Identifier ('=' additiveExpression)?;
```

[复制代码](#)

你可以看到，int 类型变量的声明，需要有一个 Int 型的 Token，加一个变量标识符，后面跟一个可选的赋值表达式。我们把上面的文法翻译成程序语句，伪代码如下：

```
1 //伪代码
2 MatchIntDeclare(){
3     MatchToken(Int);          //匹配Int关键字
4     MatchIdentifier();        //匹配标识符
5     MatchToken(equal);        //匹配等号
6     MatchExpression();        //匹配表达式
7 }
```

[复制代码](#)

实际代码在 SimpleCalculator.java 类的 IntDeclare() 方法中：

```
1 SimpleASTNode node = null;
2 Token token = tokens.peek(); //预读
3 if (token != null && token.getType() == TokenType.Int) { //匹配Int
4     token = tokens.read(); //消耗掉int
5     if (tokens.peek().getType() == TokenType.Identifier) { //匹配标识符
6         token = tokens.read(); //消耗掉标识符
7         //创建当前节点，并把变量名记到AST节点的文本值中，
8         //这里新建一个变量子节点也是可以的
9         node = new SimpleASTNode(ASTNodeType.IntDeclaration, token.getText());
10        token = tokens.peek(); //预读
11        if (token != null && token.getType() == TokenType.Assignment) {
12            tokens.read(); //消耗掉等号
13            SimpleASTNode child = additive(tokens); //匹配一个表达式
14            if (child == null) {
15                throw new Exception("invalid variable initialization, expecting");
16            }
17            else{
18                node.addChild(child);
19            }
20        }
21    } else {
22        throw new Exception("variable name expected");
23    }
24 }
```

[复制代码](#)

直白地描述一下上面的算法：

解析变量声明语句时，我先看第一个 Token 是不是 int。如果是，那我创建一个 AST 节点，记下 int 后面的变量名称，然后再看后面是不是跟了初始化部分，也就是等号加一个表达式。我们检查一下有没有等号，有的话，接着再匹配一个表达式。

我们通常会对产生式的每个部分建立一个子节点，比如变量声明语句会建立四个子节点，分别是 int 关键字、标识符、等号和表达式。后面的工具就是这样严格生成 AST 的。但是我这里做了简化，只生成了一个子节点，就是表达式子节点。变量名称记到 ASTNode 的文本值里去了，其他两个子节点没有提供额外的信息，就直接丢弃了。

另外，从上面的代码中我们看到，程序是从一个 Token 的流中顺序读取。代码中的 peek() 方法是预读，只是读取下一个 Token，但并不把它从 Token 流中移除。在代码中，我们用 peek() 方法可以预先看一下下一个 Token 是否是等号，从而知道后面跟着的是不是一个表达式。而 read() 方法会从 Token 流中移除，下一个 Token 变成了当前的 Token。

这里需要注意的是，通过 peek() 方法来预读，实际上是对代码的优化，这有点儿预测的意味。我们后面会讲带有预测的自顶向下算法，它能减少回溯的次数。

我们把解析变量声明语句和表达式的算法分别写成函数。在语法分析的时候，调用这些函数跟后面的 Token 串做模式匹配。匹配上了，就返回一个 AST 节点，否则就返回 null。如果中间发现跟语法规则不符，就报编译错误。

在这个过程中，上级文法嵌套下级文法，上级的算法调用下级的算法。表现在生成 AST 中，上级算法生成上级节点，下级算法生成下级节点。**这就是“下降”的含义。**

分析上面的伪代码和程序语句，你可以看到这样的特点：**程序结构基本上是与语法规则同构的。这就是递归下降算法的优点，非常直观。**

接着说回来，我们继续运行这个示例程序，输出 AST：

```
1 Programm Calculator
2     IntDeclaration age
3         AssignmentExp =
4             IntLiteral 45
```

前面的文法和算法都很简单，这样级别的文法没有超出正则文法。也就是说，并没有超出我们做词法分析时用到的文法。

好了，解析完变量声明语句，带你理解了“下降”的含义之后，我们来看看如何用上下文无关文法描述算术表达式。

用上下文无关文法描述算术表达式

我们解析算术表达式的时候，会遇到更复杂的情况，这时，正则文法不够用，我们必须用上下文无关文法来表达。你可能会问：“正则文法为什么不能表示算术表达式？”别着急，我们来分析一下算术表达式的语法规则。

算术表达式要包含加法和乘法两种运算（简单起见，我们把减法与加法等同看待，把除法也跟乘法等同看待），加法和乘法运算有不同的优先级。我们的规则要能匹配各种可能的算术表达式：

$2+3*5$

$2*3+5$

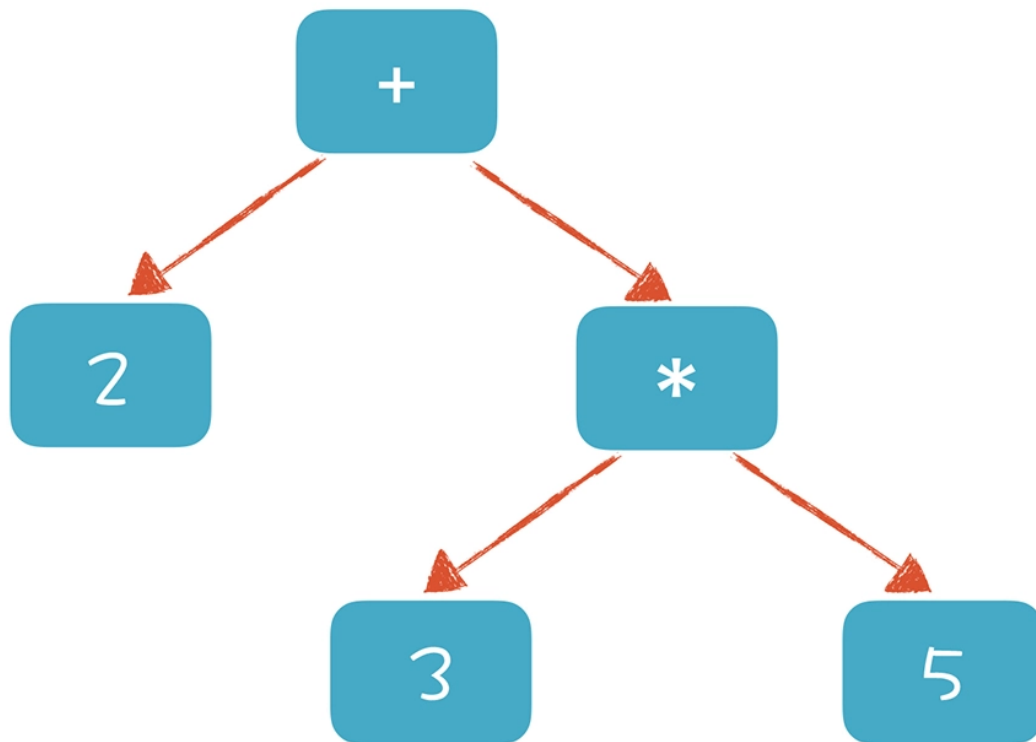
$2*3$

.....

思考一番之后，我们把规则分成两级：第一级是加法规则，第二级是乘法规则。把乘法规则作为加法规则的子规则，这样在解析形成 AST 时，乘法节点就一定是加法节点的子节点，从而被优先计算。

```
1 additiveExpression
2   :   multiplicativeExpression
3   |   additiveExpression Plus multiplicativeExpression
4   ;
5
6
7 multiplicativeExpression
8   :   IntLiteral
9   |   multiplicativeExpression Star IntLiteral
10  ;
```


你看，我们可以通过文法的嵌套，实现对运算优先级的支持。这样我们在解析 “2 + 3 * 5” 这个算术表达式时会形成类似下面的 AST：



如果要计算表达式的值，只需要对根节点求值就可以了。为了完成对根节点的求值，需要对下级节点递归求值，所以我们先完成 “3 * 5 = 15”，然后再计算 “2 + 15 = 17”。

有了这个认知，我们在解析算术表达式的时候，便能拿加法规则去匹配。在加法规则中，会嵌套地匹配乘法规则。我们通过文法的嵌套，实现了计算的优先级。

应该注意的是，加法规则中还递归地又引用了加法规则。通过这种递归的定义，我们能展开、形成所有各种可能的算术表达式。比如“ $2+3*5$ ”的推导过程：

 复制代码

```
1 -->additiveExpression + multiplicativeExpression
2 -->multiplicativeExpression + multiplicativeExpression
3 -->IntLiteral + multiplicativeExpression
4 -->IntLiteral + multiplicativeExpression * IntLiteral
5 -->IntLiteral + IntLiteral * IntLiteral
```

这种文法已经没有办法改写成正则文法了，它比正则文法的表达能力更强，叫做“**上下文无关文法**”。正则文法是上下文无关文法的一个子集。它们的区别呢，就是上下文无关文法允许递归调用，而正则文法不允许。

上下文无关的意思是，无论在任何情况下，文法的推导规则都是一样的。比如，在变量声明语句中可能要用到一个算术表达式来做变量初始化，而在其他地方可能也会用到算术表达式。不管在什么地方，算术表达式的语法都一样，都允许用加法和乘法，计算优先级也不变。好在你见到的大多数计算机语言，都能用上下文无关文法来表达它的语法。

那有没有上下文相关的情况需要处理呢？也是有的，但那不是语法分析阶段负责的，而是放在语义分析阶段来处理的。


解析算术表达式：理解“递归”的含义

在讲解上下文无关文法时，我提到了文法的递归调用，你也许会问，是否在算法上也需要递归的调用呢？要不怎么叫做“递归下降算法”呢？

的确，我们之前的算法只算是用到了“下降”，没有涉及“递归”，现在，我们就来看看如何用递归的算法翻译递归的文法。

我们先按照前面说的，把文法直观地翻译成算法。但是，我们遇到麻烦了。这个麻烦就是出现了无穷多次调用的情况。我们来看个例子。

为了简单化，我们采用下面这个简化的文法，去掉了乘法的层次：

 复制代码

```
1 additiveExpression
2     :   IntLiteral
3     |   additiveExpression Plus IntLiteral
4     ;
```

在解析 “2 + 3” 这样一个最简单的加法表达式的时候，我们直观地将其翻译成算法，结果出现了如下的情况：


首先匹配是不是整型字面量，发现不是；

然后匹配是不是加法表达式，这里是递归调用；

会重复上面两步，无穷无尽。


“additiveExpression Plus multiplicativeExpression” 这个文法规则的第一部分就递归地引用了自身，这种情况叫做**左递归**。通过上面的分析，我们知道左递归是递归下降算法无法处理的，这是递归下降算法最大的问题。

怎么解决呢？把 “additiveExpression” 调换到加号后面怎么样？我们来试一试。

 复制代码

```
1 additiveExpression
2     :   multiplicativeExpression
3     |   multiplicativeExpression Plus additiveExpression
4     ;
```

我们接着改写成算法，这个算法确实不会出现无限调用的问题：

 复制代码

```
1 private SimpleASTNode additive(TokenReader tokens) throws Exception {
2     SimpleASTNode child1 = multiplicative(); //计算第一个子节点
3     SimpleASTNode node = child1; //如果没有第二个子节点，就返回这个
4     Token token = tokens.peek();
```



```


5     if (child1 != null && token != null) {
6         if (token.getType() == TokenType.Plus) {
7             token = tokens.read();
8             SimpleASTNode child2 = additive(); //递归地解析第二个节点
9             if (child2 != null) {
10                node = new SimpleASTNode(ASTNodeType.AdditiveExp, token.getText())
11                node.addChild(child1);
12                node.addChild(child2);
13            } else {
14                throw new Exception("invalid additive expression, expecting the r
15            }
16        }
17    }
18    return node;
19 }

```

为了便于你理解，我解读一下上面的算法：

我们先尝试能否匹配乘法表达式，如果不能，那么这个节点肯定不是加法节点，因为加法表达式的两个产生式都必须首先匹配乘法表达式。遇到这种情况，返回 null 就可以了，调用者就这次匹配没有成功。如果乘法表达式匹配成功，那就再尝试匹配加号右边的部分，也就是去递归地匹配加法表达式。如果匹配成功，就构造一个加法的 ASTNode 返回。

同样的，乘法的文法规则也可以做类似的改写：

 复制代码

```

1 multiplicativeExpression
2     :   IntLiteral
3     |   IntLiteral Star multiplicativeExpression
4     ;

```

现在我们貌似解决了左递归问题，运行这个算法解析 “2+3*5” ，得到下面的 AST：

 复制代码


```

1 Programm Calculator
2     AdditiveExp +
3         IntLiteral 2
4         MulticativeExp *

```

```
5         IntLiteral 3
6         IntLiteral 5
```

是不是看上去一切正常？可如果让这个程序解析 “2+3+4” 呢？

 复制代码

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

问题是什么呢？计算顺序发生错误了。连续相加的表达式要从左向右计算，这是加法运算的结合性规则。但按照我们生成的 AST，变成从右向左了，先计算了 “3+4”，然后才跟 “2” 相加。这可不行！

为什么产生上面的问题呢？是因为我们修改了文法，把文法中加号左右两边的部分调换了一下。造成的影响是什么呢？你可以推导一下 “2+3+4” 的解析过程：

首先调用乘法表达式匹配函数 `multiplicative()`，成功，返回了一个字面量节点 2。

接着看看右边是否能递归地匹配加法表达式。

匹配的结果，真的返回了一个加法表达式 “3+4”，这个变成了第二个子节点。错误就出在这里了。这样的匹配顺序，“3+4” 一定会成为子节点，在求值时被优先计算。

所以，我们前面的方法其实并没有完美地解决左递归，因为它改变了加法运算的结合性规则。那么，我们能否既解决左递归问题，又不产生计算顺序的错误呢？答案是肯定的。不过我们下一讲再来解决它。目前先忍耐一下，凑合着用这个 “半吊子” 的算法吧。

实现表达式求值

上面帮助你理解了 “递归” 的含义，接下来，我要带你实现表达式的求值。其实，要实现一个表达式计算，只需要基于 AST 做求值运算。这个计算过程比较简单，只需要对这棵树做深度

优先的遍历就好了。

深度优先的遍历也是一个递归算法。以上文中 “2 + 3 * 5” 的 AST 为例看一下。

对表达式的求值，等价于对 AST 根节点求值。

首先求左边子节点，算出是 2。

接着对右边子节点求值，这时候需要递归计算下一层。计算完了以后，返回是 15 (3*5) 。

把左右节点相加，计算出根节点的值 17。

代码参见 SimpleCalculator.Java 中的 evaluate() 方法。

还是以 “2+3*5” 为例。它的求值过程输出如下，你可以看到求值过程中遍历了整棵树：

 复制代码

```
1      Calculating: AdditiveExp           //计算根节点
2          Calculating: IntLiteral        //计算第一个子节点
3      Result: 2                          //结果是2
4          Calculating: MultiplicativeExp //递归计算第二个子节点
5              Calculating: IntLiteral
6              Result: 3
7              Calculating: IntLiteral
8              Result: 5
9      Result: 15                          //忽略递归的细节，得到结果是15
10     Result: 17                          //根节点的值是17
```

你可以运行一下示例程序看看输出结果，而且我十分建议你修改表达式，自己做做实验，并试着让表达式不符合语法，看看语法分析程序能不能找出错误来。

课程小结

今天我们实现了一个简单的公式计算器，尽管简单，相信你已经有了收获。那么我来总结一下今天的重点：

初步了解上下文无关文法，知道它能表达主流的计算机语言，以及与正则文法的区别。

理解递归下降算法中的“下降”和“递归”两个特点。它跟文法规则基本上是同构的，通过文法一定能写出算法。

通过遍历 AST 对表达式求值，加深对计算机程序执行机制的理解。

在后面的课程中，我们会在此基础上逐步深化，比如在变量声明中可以使用表达式，在表达式中可以使用变量，例如能够执行像这样的语句：

 复制代码

```
1 int A = 17;  
2 int B = A + 10*2;
```

实现了上述功能以后，这个程序就越来越接近一个简单的脚本解释器了！当然，在此之前，我们还必须解决左递归的问题。所以下一讲，我会带你填掉左递归这个坑。我们学习和工作的过程，就是在不停地挖坑、填坑，你要有信心，只要坚强走过填坑这段路，你的职业生涯将会愈发平坦！

一课一思

递归算法是很好的自顶向下解决问题的方法，是计算机领域的一个核心的思维方式。拥有这种思维方式，可以说是程序员相对于非程序员的一种优势。

那么，你是否用递归算法或递归思维解决过工作中或者生活中存在的某些问题？你能否再找一些证据证明一下，哪些语法规则只能用上下文无关文法表达，用正则文法是怎样都写不出来的？欢迎在留言区和我一起讨论。

最后，十分感谢你的阅读，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。

另外，为了便于你更好地学习，我将本节课的示例程序放到了 [码云](#) 和 [GitHub](#) 上，你可以看一下。

精选留言 (109)



Sam 置顶

2019-08-22

初学者看了 8 遍, 终于看懂了, 不急不燥, 慢慢看就行了

作者回复: 点赞!

共 5 条评论 >

👍 87



许童童

2019-08-19

老师你好,

additiveExpression

```
: multiplicativeExpression
| additiveExpression Plus multiplicativeExpression
;
```

multiplicativeExpression

```
: IntLiteral
| multiplicativeExpression Star IntLiteral
;
```

这种DSL怎么理解?

作者回复: 这个实际上就是语法规则, 是用BNF表达的。以addtive为例, 它有两个产生式。

产生式1: 一个乘法表达式

产生式2: 一个加法表达式 + 乘法表达式。

通过上面两个产生式的组合, 特别是产生式2的递归调用, 就能推导出所有的加减乘数算术表达式。

比如, 对于2*3这个表达式, 运用的是产生式1。

对于2+3*5, 运用的是产生式2。

我上面用的语法规则的写法, 实际上是后面会用到的Antlr工具的写法。你也可以这样书写, 就是一般教材上的写法:

A -> M | A + M

M -> int | M * int

我们每个非终结符只用了一个大写字母代表，比较简洁。我在文稿中用比较长的单词，是为了容易理解其含义。

其中的竖线，是选择其一。你还可以拆成最简单的方式，形成4条规则：

$A \rightarrow M$

$A \rightarrow A + M$

$M \rightarrow \text{int}$

$M \rightarrow M * \text{int}$

上面这些不同的写法，都是等价的。你要能够看习惯，在不同的写法中自由切换。

不知道是否解答了你的疑问。

共 9 条评论 >

👍 101



阿尔伯特

2019-09-05

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

读了几遍老师的讲义。才逐渐理解了语法解析中用的推导。接着前一讲，攒了个程序。

就这个推导说说我目前的理解，其中最开始不能理解的根本原因就是没能理解语法规则之间的相互关系，以及与此相关的token的消耗。

比如例子 $A \rightarrow \text{Int} \mid A + \text{Int}$

在最开始的理解中，错误以为，这两条是顺序关系，与此相应就想当然认为token的消耗是像字符串匹配一样“一个接一个”的进行。这种错误思路是这样的：2+3，首先看token 2，它是nt所以消耗掉，然后类推。

而实际上，这两条规则是从某种程度上是“互斥”的关系。也就是说，2+3 要么是Int，要么是A+Int，在没有找到合适的规则前，token是不会被消耗的。由此，在深度优先实现中，就有老师所说的推导实现过程。总的要解决的问题是，2+3 是不是A，能不能用这条A规则来解释。那么就看它是否满足A的具体规则。首先，2+3 显然不是Int，因此没有token消耗。然后，在匹配A + Int时，上来就要看 2+3 是不是A，不断要解决原来的问题，从而就产生了所谓左递归。

所以在深度优先情况下，打破无穷递归，就把规则改为 $A \rightarrow \text{Int} \mid \text{Int} + A$ 。这时，推导，2+3显然不是Int。于是看Int + A。2显然是Int，于是消耗掉；再看+，消耗掉；再看3是不是A，3显然是Int，所以返回。

作为老师的示例程序，并没有体现出对 $A \rightarrow M \mid M + A$ 两条“互斥”标准的分别处理，所以可能造成了一定疑惑。我是这样理解的，程序事实上合并了对于M的处理，一段代码，处理了第一全部和第二一部分。比如2+3*5，机械按照刚才的理解，2+3*5显然不是M，于是任何token都不消耗，退回。再匹配第二条，第二条上来就会找，它是不是M开头，如果是就消耗掉+之前

的token；然后消耗+；然后再看看A。程序是不管如何，上来就看，是不是M开头。如果不是，那肯定就不是A，就返回NULL。如果是，就看你有没有“+”，如果没有，你就直接是规则第一条，如果有，就看你是不是第二条。从而就实现了两条M的合并处理。

在看了评论后，又看到了广度优先的推导，以及老师说有大量回溯，刚开始不甚理解。后来有点理解， $A \rightarrow \text{Int} | A + \text{Int}$ 。该规则在深度优先中，会导致左递归。如果用广度优先，则会有如下方式。所谓广度优先，通俗理解就是“横”着来。那我理解是， $2+3$ 显然不是Int。因此要找第二条规则那就是首先要从头扫描，找“+”，然后再“回头”看2是不是A，这就带来了回溯吧。但是由于只用了部分token，即判断2而不是 $2+3$ 是不是A，所以，避免了左递归。

请老师和各位同学有空帮忙指正。谢谢

作者回复：哇，这么认真，这么仔细:-)

竖线“|”是或者的关系，怪我忘了强调这一点了。在正则文法、上下文无关文法中，“|”都是代表几个不同的选项。

另外，在前端技术的算法篇，会再把我们对算法的理解提升一下。我尽量做几个示例程序，演示出深度优先和广度优先的差别来。特别是，为什么广度优先的回溯会太多。

当然，如果你能先于我写一个，也可以分享给大家，就省了我的事了:-)

为你的认真精神点赞！

共 12 条评论 >

👍 48



鸠摩智

2019-08-19

老师您好，请问语法和文法有什么区别和联系？

作者回复：你提的问题特别好！其他同学可能也会有这种疑问。

文法，英文叫做Grammar，是形式语言（Formal Language）的一个术语。所以也有Formal Grammar这样的说法。这里的文法有定义清晰的规则。比如，我们的词法规则、语法规则和属性规则，使用形式文法来定义的。我们的课程里讲解了正则文法(Regular Grammar)、上下文无关文法(Context-free Grammar)等不同的文法规则，用来描述词法和语法。

语法分析中的这个语法，英文是Syntax，主要是描述词是怎么组成句子的。一个语言的语法规则，通常指的是这个Syntax。

问题是，Grammar这个词，在中文很多应用场景中也叫做语法。这是会引起混淆的地方。我们在使用的时候要小心一点就行了。

比如，我做了一个规则文件，里面都是一些词法规则（Lexer Grammar），我会说，这是一个词法规则文件，或者词法文法文件。这个时候，把它说成是一个语法规则文件，就有点含义模糊。因为这

里面并没有语法规则 (Syntax Grammar) 。

为你的认真思考点赞！

共 4 条评论 >

👍 44



长方体混凝土移动工程...

2019-08-22

2 + 3 的推导过程就是要找到一个表达式可以正确的表达这个计算规则。顺序的消耗掉三个token,找到能表达这个式子的公式推导过程完成,并成功。

如果使用A: $M \mid A + M$ 的方式来递归代入,步步推导无法消耗完三个token的情况下就会陷入无限循环

推导过程:

1. 2 + 3 不是M表达式,使用A + M的方法匹配
2. A + M 在推导A的时候重复第1步操作,因为此时我们并没有消耗掉token,将完整的token代入A重复第1步推导,无限循环

但如果使用A: $M \mid M + A$ 的方式来递归代入

推导过程:

1. 2 + 3 不是一个M, 使用M + A推导,变成M + A
2. 使用2去匹配M可以顺序推导并消耗掉2这个字面量token,此时流中剩下 + 3两个token
3. 使用M + A规则中的+号消耗掉 + 3中的+号token
4. 将M + A中的A再次推导成M
- 5.最终推导成M + M,此时剩下的最后一个字面量token 3被消耗掉

作者回复: 没错。很好。

既然你已经理解了, 那么我再增加一点难度。当前推导是最左推导 (LeftMost) 推导的算法。也就是总是先把左边的非终结符展开。而且是深度优先的。

你再广度优先推演一下看看?

你再最右推导一下看看?

可能你的感受又不一样。很有意思的。可以作为消遣游戏 :-D

**张辽儿**

2019-08-20

为什么出现左递归无限调用我还没有理解，例如 $2+3$ ；当进入加法表达式递归的时候，参数不是已经变成了2吗，然后就是乘法表达式，最后形成字面常量。请老师解答下我的疑问，谢谢

作者回复: 为了方便讨论，我们把规则简化一下，去掉乘法那一层。否则在乘法那就已经无限递归下去了。修改后为：

`additive -> IntLiteral | additive IntLiteral ;`

我们假设是最左推导，也就是总是先展开左边的非中介符。

第一遍: `additive->IntLiteral`，但因为后面还有Token没处理完，所以这个推导过程会失败，要退回来。

这可能是你没理解的地方。我们是要用`additive`匹配整个Token串，而不仅仅是第一个Token。

第二遍: 用第二个产生式，`additive->additive->IntLiteral`，还是一样失败。

第三遍: `additive->additive->additive->IntLiteral`。

第四遍:

这样说，有没有帮助？

**炎发灼眼**

2020-04-11

老师，又把文章读了好几遍，然后仔仔细细看了你所有问题的回复，重新理解了，是不是这样；

例如: $2+3$ 这个式子，用`A->Int | A + Int`去推导，就是用 $2+3$ 去匹配第一个式子`Int`，不满足，然后看是否满足第二个式子`A + Int`，

这个时候，因为我们能直接看到整个表达式是什么样子的，现在是 $2+3$ ，所以我们本能的就使用了广度优先算法，觉得用2匹配A，+自然匹配，Int刚好消耗掉3，完美；

但是计算机拿到TOKENS的时候，是不知道这个是什么样子的，所以按照写好的深度优先算法来匹配，每一次的匹配，都想尽办法尽可能多的

消耗掉TOKENS中的TOKEN，所以，在`A + Int`的时候，用整个TOKENS来和A匹配，看看最多能消耗掉多少个TOKEN，其实这个时候，

对于计算机来说，是不知道这个式子后面还有 + Int这个的，然后回到了那一步，先用TOKEN S匹配Int，不对，退回来，进行另一个式子的尝试，又回到了A + Int，然后又是对A + Int中的A进行尽可能的多匹配，周而复始，就是所谓的左递归了

作者回复: 不错。你已经思考得挺细致的了！值得表扬！

如果你想继续做一下脑体操，可以看看17讲中与广度优先有关的算法，看看能否把深度优先和广度优先在大脑里转换自如！

共 4 条评论 >

👍 24



kaixiao7

2019-08-21

老师您好：

additiveExpression

```
: multiplicativeExpression
| multiplicativeExpression Plus additiveExpression
;
```

multiplicativeExpression

```
: IntLiteral
| IntLiteral Star multiplicativeExpression
;
```

在用上述文法求解 $2+3*5$ 时，首先会匹配乘法规则，根据代码，这一步返回字面量2，显然是产生式1匹配的结果，我的问题是这里不应该用 产生式1 匹配 $2+3*5$ 整个token串吗？

另外，再计算表达式 $2*3*5$ 时，返回的AST为 $2*3$ ，而 $*5$ 丢失了，因此multiplicative()方法中的SimpleASTNode child2 = primary(tokens); 是不是应该递归调用multiplicative()呢？

期待您的解惑！

作者回复: 算法可以首先尝试产生式1。推导顺序是这样的：

```
additive -> multiplicative(加法的产生式1)
          -> Intliteral (2) (乘法的产生式1)
```

这时候只消化了一个Token呀。我们是要用一个表达式把这5个Token都消化掉才行。所以会继续尝试乘法的产生式2。

```
additive -> multiplicative(加法的产生式1)
          -> Intliteral * multiplicative (乘法的产生式2)
```

这次尝试不成功，因为我们下一个Token是加号，不是乘号。

现在，退回来尝试加法的产生式2。

additive \rightarrow multiplicative + additive(加法的产生式2)

\rightarrow Intliteral(2) + additive

\rightarrow Intliteral(2) + multiplicative

\rightarrow Intliteral(2) + Intliteral(3) 不行，因为还有Token

\rightarrow Intliteral(2) + Intliteral(3) * multiplicative 又用上乘法的产生式2了

\rightarrow Intliteral(2) + Intliteral(3) * Intliteral(5)

这是严格的推导过程。我在示例代码的实现中，因为提取了左公因子，所以没用多次回溯。

这样说，你能明白吗？如果还不明白，就再问。

共 7 条评论 >

👍 19



阿名

2019-08-19

如果没有基础 比较难听得懂 比如文法推导 终结符 非终结符 这些概念 本身就不好理解

作者回复: 实际上，这些看上去比较正式的术语，是我在这篇文稿的最后一版才加上去的。其实，你忽略这些术语，也完全能看懂文稿。加上这些术语，是为后面正式讲算法做个铺垫。

我知道编译原理的术语本身就能吓倒很多人。但是这门课程的重点在于帮你建立直觉（Intuition）。建立起直觉来以后，你其实已经明白了语法分析的过程，你已经对它有熟悉感了。之后你再把这些直觉跟术语联系在一起，就不觉得困难了。

再次强调一点，首先建立直觉，然后再追求对术语和算法的严格理解。

学编译原理最大的困难不是这门课本身的难度，而是我们对它的畏惧心理。相信你自己！

共 3 条评论 >

👍 17



朱天超

2019-08-19

课下可以参考下:《编译系统透视: 图解编译原理》



👍 15



Rockbean

2019-08-25

小白读得有些吃力

> "我们首先把变量声明语句的规则，用形式化的方法表达一下。它的左边是一个非终结符（Non-terminal）。右边是它的产生式（Production Rule）。"

"它的左边"的"它"是指变量声明语句"int age = 45"呢还是什么，如果是变量声明语句,那左边是左到哪里，是"int age"还是什么？非终结符，是什么，往前翻了几个课也没有找到，或者说终结符是什么？同样的右边是右从哪里开始算右边？产生式是"=45"吗？小白对这些基础词汇有点蒙，见笑了

作者回复: 1.终结符跟非终结符在04讲得更细一点，可以在04讲再体会一下。

2.它的左边，是指：

intDeclaration : Int Identifier ('=' additiveExpression)?;

这个规则，冒号的左边。

共 2 条评论 >

👍 11



∇(●°▽°●)/∇

2019-08-19

递归容易表达很多算法，但是计算机本身执行递归有栈溢出和效率等问题，如何平衡呢？

作者回复: 你说的很对！

实际上，你提到了递归的优化问题。这是一个专门的研究领域。在SICP（《计算机程序的构造和解释》）这本书中，对这个问题也很重视。

我们下一讲会提到尾递归的情形，也就是线性迭代的递归函数。它实际上可以转化成循环语句，就没有对栈的消耗了。这是在编译技术中常用的一种优化策略。你可以提前了解一下尾递归：)

共 2 条评论 >

👍 9



蜉蝣

2020-05-04

同行们，看不懂没关系，继续往下看，把下面两个小节都看完之后照着 github 上面的代码敲一遍，再回来看，你就能懂了。因为天资愚钝的我就是这样搞懂的。



👍 5



中年男子

2019-08-21

总结一下：开头讲的推导过程就是递归过程
针对加法表达式 $2+3$

最初规则：

additive

```
:multiplicative
| additive Plus multiplicative
;
```

multiplicative

```
: IntLiteral
| multiplicative Star IntLiteral
;
```

简化：

additive

```
: IntLiteral
| additive Plus IntLiteral
;
```

multiplicative

```
: IntLiteral
| multiplicative Star IntLiteral
;
```

遍历整个token串，运用产生式1，不是 IntLiteral，运用产生式2，这里会出现左递归

解决左递归，把additive 调换到 加号 (plus) 后边去。相应的multiplicative 也调换位置

additive

```
: IntLiteral
| IntLiteral Plus multiplicative
;
```

multiplicative

```
: IntLiteral
| IntLiteral Star multiplicative
;
```

再解析 “2+3+4”

这里我就不明白了，为什么首先调用乘法表达式匹配函数，就能成功返回字面量2呢？
文法规则里的 “Star” 是什么意思？ 还请老师解惑！

作者回复: 我觉得你在认真分析，点赞！

在讨论左递归会无穷次递归的时候，我们把语法简化了一下，是根本就不要乘法运算了，只看加法运算。这样来推演左递归更加方便一点。

简化后的规则为：

additive -> IntLiteral | additive Intliteral ;

解析过程：

第一遍：additive->IntLiteral，但因为后面还有Token没处理完，所以这个推导过程会失败，要退回来。

第二遍：additive->additive->IntLiteral，还是一样失败。

第三遍：additive->additive->additive->IntLiteral。

第四遍：....

Star就是*号，是一个Token符号。是词法分析过程中形成的。这样的问题建议你看看源代码，甚至运行一下，就更清楚了。

如果不清楚，继续问我。

共 2 条评论 >

👍 5



William

2019-08-20

前端开发，表示有些吃力。很好奇Babel、Node.js的编译机制。

作者回复: 学完课程，你应该会理解这两个的运作机制。

Babel，只是做语言翻译，只需要前端技术就可以了。翻译成AST，做完语义分析，再转成另一个版本的js。

Node.js基于v8，不仅仅做前端工作，更重要的是在后端运行时做各种优化。



👍 5



小广

2019-08-20

解析“2 + 3”遇到左递归问题那一段，需要解析到 + 号的时候，才会发生下面的递归循环的问题，一开始看有点断档，因为第一个字符2是不会遇最递归的问题的，如果老师可以提示一下话，可能看起来会更加流畅一点 $O(n_n)O\sim$

作者回复: 嗯。谢谢你的建议。我看看是否需要把文稿表达得更细致一点。

如果不要乘法那一层，说明起来可能更简洁一些。否则，其实进入到乘法以后，就已经递归个不停了，根本回不到加法规则这来。

修改规则为:

additive -> IntLiteral | additive Intliteral ;

第一遍: additive->IntLiteral, 但因为后面还有Token没处理完, 所以这个推导过程会失败, 要退回来。

第二遍: additive->additive->IntLiteral, 还是一样失败。

第三遍: additive->additive->additive->IntLiteral。

第四遍:

共 4 条评论 >



4



恩佐

2019-11-07

https://github.com/shaojintian/learn_compiler/blob/master/calculator/calculator_test.go

老师我完全自己实现了calculator, 可否看一下, 指点一下, 多谢

作者回复: 看到你的工程经常更新, 我已经在github上加了关注。

简单地用go test运行了一下你的lexer和calculator。运行的输出挺漂亮!

如果有小的建议的话, 就是再稍微多写点注释。否则过一阵你自己看代码会想不起来了...

共 2 条评论 >



3



Sun Fei

2019-09-03

宫老师, 看了几遍, 还是没有理解 下面所表达的含义。

它的左边是一个非终结符 (Non-terminal) 。右边是它的产生式 (Production Rule) 。在语法解析的过程中, 左边会被右边替代。如果替代之后还有非终结符, 那么继续这个替代过程, 直

到最后全部都是终结符（Terminal）。

谢谢。

作者回复: 这个地方确实写得不够细, 没有交代清楚什么是非终结符, 什么是终结符。后来在下一讲里有更多的描述。

总体来说, 终结符, 就是我们在词法分析阶段获得的Token。在建立AST的时候, 它们是叶子节点。

因为不管是表达式也好, 语句也好, 最终都是由这些Token构成的。

非终结符就相当于AST非叶子节点, 它们是由Token构成的一些语法结构, 比如表达式、语句。

如果把AST这种直观的理解换成文法的推导过程, 那么就是反着来的。从非终结符一步步替换, 直到全部替换成终结符。也就是从树根, 一步步生成一棵AST。

共 2 条评论 >

👍 3



不会魔法

2020-07-27

关于为什么 $A \rightarrow M|A+M$, 为什么这样推导, 为什么推导的规则是这样的说下自己的理解。

首先用中文来翻一下这个表达式, 把A理解为一个句子, 把A和M理解为句子A中包含的元素。

比如, 对 '你好' 这个句子进行推导, 可推导为

你好 - \rightarrow 你|好|你好

构成这个句子包含这几种元素可能。

进行抽象, 句子你好 = 主语 + 谓语, 主语记做S, 谓语记做V。

以bnf表达式形式描述。

$SV(\text{你好}) \rightarrow S(\text{你})|V(\text{好})|SV(\text{你好})$

这个句子可能由这三种元素组合而成。

再简化

$SV \rightarrow S|V|SV$

简化sv为s, 右侧大写转为小写方便区分

$S \rightarrow s|v$

是不是有点内个味道了。

然后咱们基本就明白了啥是推导了吧, 就是说左边的集合(句子是单词的集合, 文法是词法的集合)包含右边元素的可能。找出来这个句子中可能包含的单词有哪些, 抽象一下就是找出这个句子中可能包含的单词类型有哪些。

你好中包含，主语谓语

再拉回来说本章中的

$A \rightarrow M | A + M$

等于

additiveExpression

 : multiplicativeExpression

 | additiveExpression Plus multiplicativeExpression

;

a=additiveExpression，是加法表达式的意思。

M = multiplicativeExpression，是乘法表达式的意思。

那么上面个bnf表达式的意思就是说。

一个加法表达式中可能包含元素有乘法表达式或者加法表达式 + 乘法表达式的意思。

这是简化后的答案，如果不简化的话，大家可以穷举一下。

本结中为了简单只使用了加法和乘法，没有减法除法等。

加法表达式包含，加法，乘法，加法 + 乘法，加法 + 加法，加法 + 加法 + 乘法 等等。

以上所有可能最终都可以用

乘法和加法 + 乘法，两种元素代替。

至于为什么我就知道了，我也是个萌新哈哈。看到这里大家应该都能差不多明白

$A \rightarrow M | A + M$

究竟是个什么东西了吧。



👍 2



Leoorz

2020-06-16

老师真是太细致了，基本每个回答都回复，这一节上来就是劝退的节奏.....基本是看下面回复看懂的哈哈，给老师点赞

作者回复: 分享一点我的观点：学习在很大程度上是个心理问题。

我们这门课的所有学员呢，会形成一个气场。当你看到别的同学也能学会的时候呢，你自然就会觉

得，其实我也OK。

20多年前，我读研的时候，考GRE。以前一天背几十个单词就了不起了。到了GRE的班上，发现大家每天是背200~300个单词，于是自己也就接受了这个标准。

学计算机科学其实也一样。所有这些基础知识，其实都是每个人都有能力学会的。



2