

17 | First和Follow集合：用LL算法推演一个实例

2019-09-20 宫文学 来自北京

《编译原理之美》



在前面的课程中，我讲了递归下降算法。这个算法很常用，但会有回溯的现象，在性能上会有损失。所以我们要把算法升级一下，实现带有预测能力的自顶向下分析算法，避免回溯。而要做到这一点，就需要对自顶向下算法有更全面的了解。

另外，在留言区，有几个同学问到了一些问题，涉及到对一些基本知识点的理解，比如：

基于某个语法规则做解析的时候，什么情况下算是成功，什么情况下算是失败？

使用深度优先的递归下降算法时，会跟广度优先的思路搞混。

要搞清这些问题，也需要全面了解自顶向下算法。比如，了解 Follow 集合和 \$ 符号的用法，能帮你解决第一个问题；了解广度优先算法能帮你解决第二个问题。

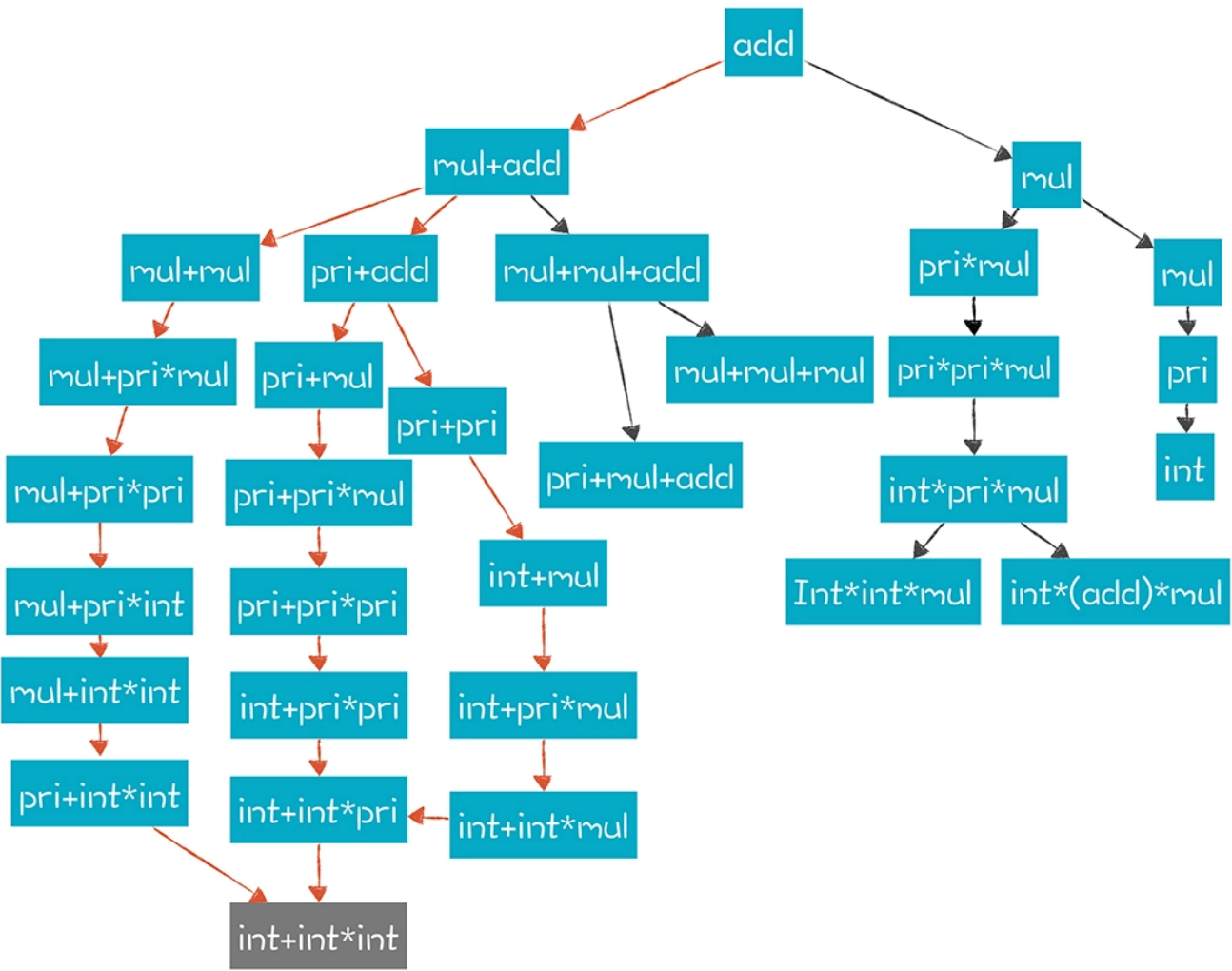
所以，本节课，我先把自顶向下分析的算法体系梳理一下，让你先建立更加清晰的全景图，然后我再深入剖析 LL 算法的原理，讲清楚 First 集合与 Follow 集合这对核心概念，最终让你把

自顶向下的算法体系吃透。

自顶向下分析算法概述

自顶向下分析的算法是一大类算法。总体来说，它是从一个非终结符出发，逐步推导出跟被解析的程序相同的 Token 串。

这个过程可以看做是一张图的搜索过程，这张图非常大，因为针对每一次推导，都可能产生一个新节点。下面这张图只是它的一个小角落。



算法的任务，就是在大图中，找到一条路径，能产生某个句子（Token 串）。比如，我们找到了三条橘色的路径，都能产生“2+3*5”这个表达式。


根据搜索的策略，有**深度优先 (Depth First)** 和**广度优先 (Breadth First)** 两种，这两种策略的推导过程是不同的。

深度优先是沿着一条分支把所有可能性探索完。以 “add->mul+add” 产生式为例，它会先把 mul 这个非终结符展开，比如替换成 pri，然后再把它的第一个非终结符 pri 展开。只有把这条分支都向下展开之后，才会回到上一级节点，去展开它的兄弟节点。

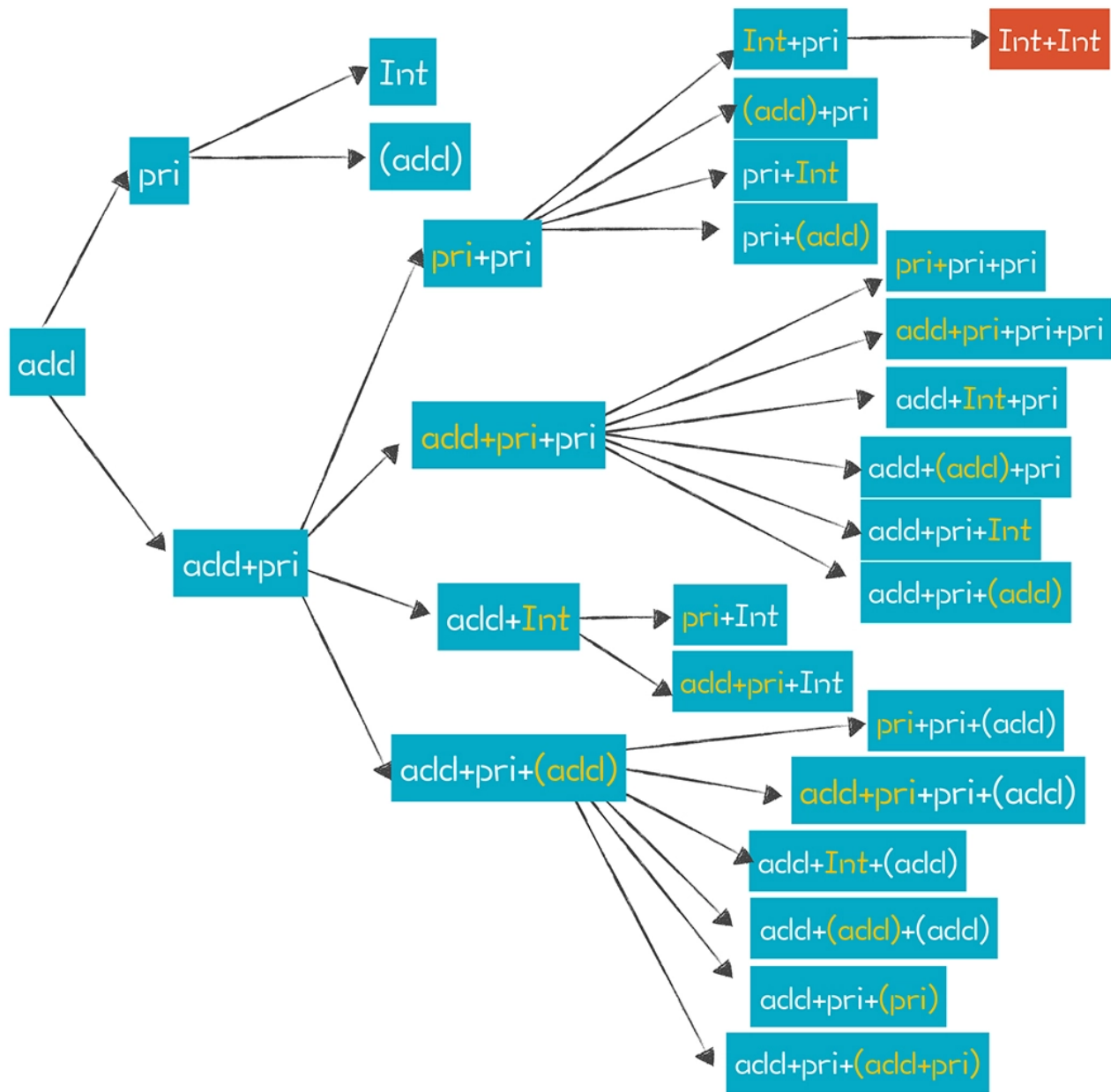
递归下降算法就是深度优先的，这也是它不能处理左递归的原因，因为左边的分支永远也不能展开完毕。

而针对 “add->add+mul” 这个产生式，**广度优先**会把 add 和 mul 这两个都先展开，这样就形成了四条搜索路径，分别是 mul+mul、add+mul+mul、add+pri 和 add+mul*pri。接着，把它们的每个非终结符再一次展开，会形成 18 条新的搜索路径。

所以，广度优先遍历，需要探索的路径数量会迅速爆炸，成指数级上升。哪怕用下面这个最简单的语法，去匹配 “2+3” 表达式，都需要尝试 20 多次，更别提针对更复杂的表达式或者采用更加复杂的语法规则了。

 复制代码

```
1 //一个简单的语法
2 add -> pri           //1
3 add -> add + pri     //2
4 pri -> Int           //3
5 pri -> (add)         //4
```



这样看来，指数级上升的内存消耗和计算量，使得广度优先根本没有实用价值。虽然上面的算法有优化空间，但无法从根本上降低算法复杂度。当然了，它也有可以使用左递归文法的优点，不过我们不会为了这个优点去忍受算法的性能。

而深度优先算法在内存占用上是线性增长的。考虑到回溯的情况，在最坏的情况下，它的计算量也会指数式增长，但我们可以通过优化，让复杂度降为线性增长。

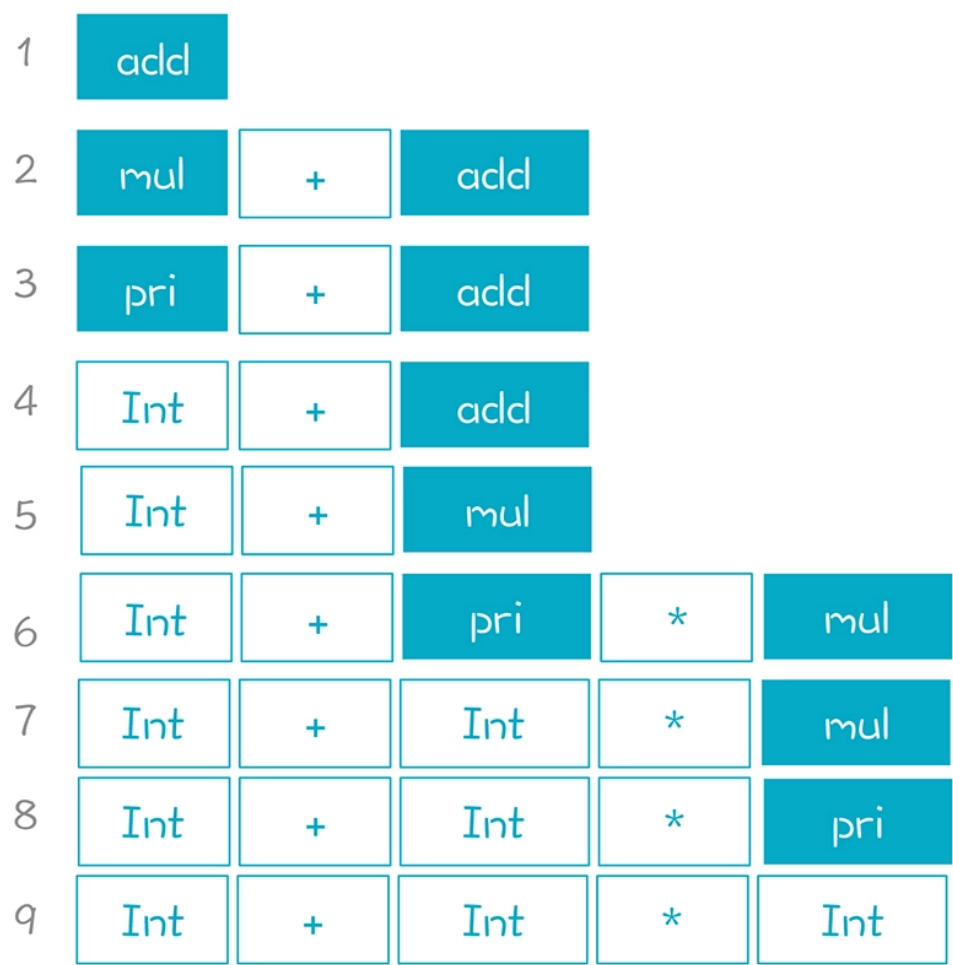
了解广度优先算法，你的思路会得到拓展，对自顶向下算法的本质有更全面的理解。另外，在写算法时，你也不会一会儿用深度优先，一会儿用广度优先了。

针对深度优先算法的优化方向是减少甚至避免回溯，思路就是给算法加上预测能力。比如，我在解析 statement 的时候，看到一个 if，就知道肯定这是一个条件语句，不用再去尝试其他产生式了。

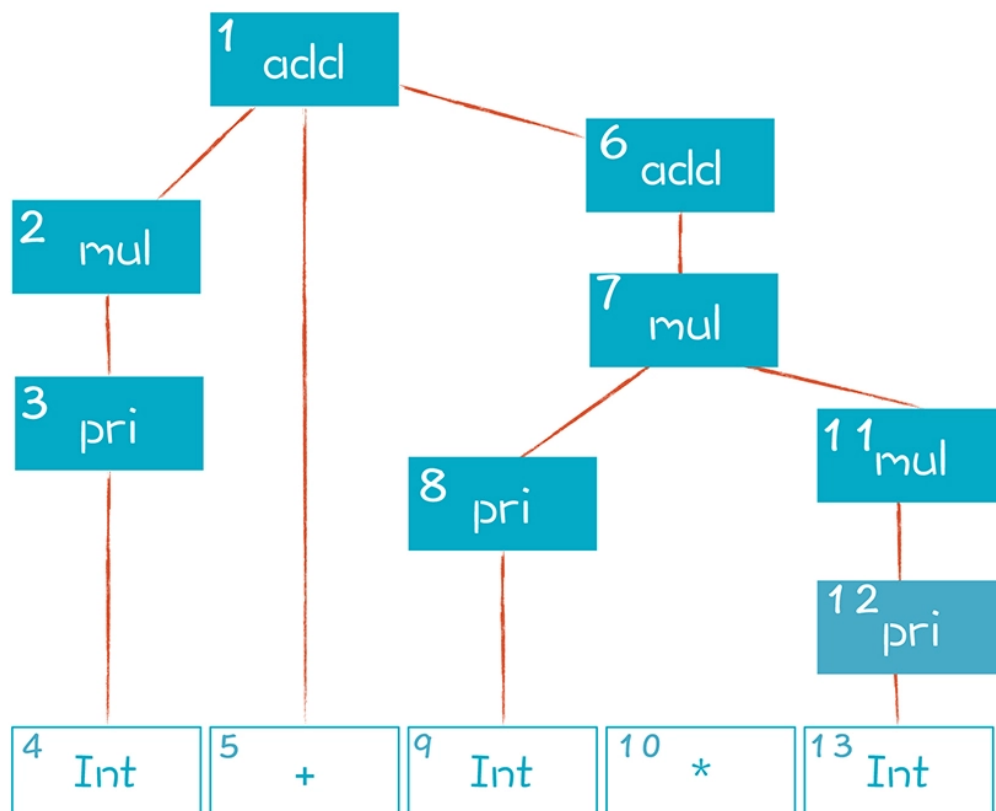
LL 算法就属于这类预测性的算法。第一个 L，是 Left-to-right，代表从左向右处理程序代码。第二个 L，是 Leftmost，意思是最左推导。

按照语法规则，一个非终结符展开后，会形成多个子节点，其中包含终结符和非终结符。最左推导是指，从左到右依次推导展开这些非终结符。采用 Leftmost 的方法，在推导过程中，句子的左边逐步都会被替换成终结符，只有右边的才可能包含非终结符。

以 “2+3*5” 为例，它的推导顺序从左到右，非终结符逐步替换成了终结符：



下图是上述推导过程建立起来的 AST，“1、2、3.....” 等编号是 AST 节点创建的顺序：



好了，我们把自顶向下分析算法做了总体概述，并讲清楚了最左推导的含义，现在来看看 LL 算法到底是怎么回事。

计算和使用 First 集合

LL 算法是带有预测能力的自顶向下算法。在推导的时候，我们希望当存在多个候选的产生式时，瞄一眼下一个（或多个）Token，就知道采用哪个产生式。如果只需要预看一个 Token，就是 LL(1) 算法。

拿 statement 的语法举例子，它有好几个产生式，分别产生 if 语句、while 语句、switch 语句.....

[复制代码](#)

```

1 statement
2   : block
3   | IF parExpression statement (ELSE statement)?
4   | FOR '(' forControl ')' statement
5   | WHILE parExpression statement
6   | DO statement WHILE parExpression ';'
7   | SWITCH parExpression '{' switchBlockStatementGroup* switchLabel*
  
```

```
8      | RETURN expression? ';'
9      | BREAK IDENTIFIER? ';'
10     | CONTINUE IDENTIFIER? ';'
11     | SEMI
12     | statementExpression=expression ';'
13     | identifierLabel=IDENTIFIER ':' statement
14     ;
```

如果我看到下一个 Token 是 if，那么后面跟着的肯定是 if 语句，这样就实现了预测，不需要一个一个产生式去试。

问题来了，if 语句的产生式的第一个元素就是一个终结符，这自然很好判断，可如果是一个非终结符，比如表达式语句，那该怎么判断呢？

我们可以为 statement 的每条分支计算一个集合，集合包含了这条分支所有可能的起始 Token。如果每条分支的起始 Token 是不一样的，也就是这些集合的交集是空集，那么就很容易根据这个集合来判断该选择哪个产生式。我们把这样的集合，**就叫做这个产生式的 First 集合。**

First 集合的计算很直观，假设我们要计算的产生式是 x：

如果 x 以 Token 开头，那么 First(x) 包含的元素就是这个 Token，比如 if 语句的 First 集合就是{IF}。

如果 x 的开头是非终结符 a，那么 First(x) 要包含 First(a) 的所有成员。比如 expressionStatement 是以 expression 开头，因此它的 First 集合要包含 First(expression) 的全体成员。

如果 x 的第一个元素 a 能够产生 ϵ ，那么还要再往下看一个元素 b，把 First(b) 的成员也加入到 First(x)，以此类推。如果所有元素都可能返回 ϵ ，那么 First(x) 也应该包含 ϵ ，意思是 x 也可能产生 ϵ 。比如下面的 blockStatements 产生式，它的第一个元素是 blockStatement*，也就意味着 blockStatement 的数量可能为 0，因此可能产生 ϵ 。那么 First(blockStatements) 除了要包含 First(blockStatement) 的全部成员，还要包含后面的“;”。

```
1 blockStatements
2     : blockStatement*
3     ;
```

[复制代码](#)

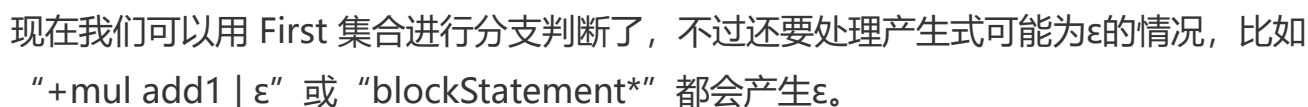
最后，如果 x 是一个非终结符，它有多个产生式可供选择，那么 $\text{First}(x)$ 应包含所有产生式的 $\text{First}()$ 集合的成员。比如 `statement` 的 First 集合要包含 `if`、`while` 等所有产生式的 First 集合的成员。并且，如果这些产生式只要有一个可能产生 ϵ ，那么 x 就可能产生 ϵ ，因此 $\text{First}(x)$ 就应该包含 ϵ 。

在本讲的示例程序里，我们可以用 [SampleGrammar.expressionGrammar\(\)](#) 方法获得一个表达式的语法，把它 `dump()` 一下，这其实是消除了左递归的表达式语法：

```
1 expression : assign ;
2 assign    : equal | assign1 ;
3 assign1   : '=' equal assign1 |  $\epsilon$  ;
4 equal     : rel equal1 ;
5 equal1    : ('==' | '!=') rel equal1 |  $\epsilon$  ;
6 rel       : add rel1 ;
7 rel1      : ('>=' | '>' | '<=' | '<') add rel1 |  $\epsilon$  ;
8 add       : mul add1 ;
9 add1      : ('+' | '-') mul add1 |  $\epsilon$  ;
10 mul      : pri mul1 ;
11 mul1     : ('*' | '/') pri mul1 |  $\epsilon$  ;
12 pri      : ID | INT_LITERAL | LPAREN expression RPAREN ;
```

[复制代码](#)


我们用 `GrammarNode` 类代表语法的节点，形成一张语法图（蓝色节点的下属节点之间是“或”的关系，也就是语法中的竖线）。



计算和使用 Follow 集合


对 ϵ 的处理分成两种情况。

第一种情况，是产生式中的部分元素会产生 ϵ 。比如，在 Java 语法里，声明一个类成员的时候，可能会用 `public`、`private` 这些来修饰，但也可以省略不写。在语法规则中，这个部分是 “`accessModifier?`”，它就可能产生 ϵ 。

 复制代码

```
1 memberDeclaration : accessModifier? type identifier ';' ;
2 accessModifier : 'public' | 'private' ;
3 type : 'int' | 'long' | 'double' ;
```

所以，当我们遇到下面这两个语句的时候，都可以判断为类成员的声明：


 复制代码

```
1 public int a;
2 int b;
```

这时，`type` 能够产生的终结符 ‘`int`’、‘`long`’ 和 ‘`double`’ 也在 `memberDeclaration` 的 First 集合中。这样，我们实际上把 `accessModifier` 给穿透了，直接到了下一个非终结符 `type`。所以这类问题依靠 First 集合仍然能解决。在解析的过程中，如果下一个 Token 是 ‘`int`’，我们可以认为 `accessModifier` 返回了 ϵ ，忽略它，继续解析下一个元素 `type`，因为它的 First 集合中才会包含 ‘`int`’。

第二种情况是产生式本身（而不是其组成部分）产生 ϵ 。这类问题仅仅依靠 First 集合是无法解决的，要引入另一个集合：Follow 集合。它是所有可能跟在某个非终结符之后的终结符的集合。

以 `block` 语句为例，在 `PlayScript.g4` 中，大致是这样定义的：

 复制代码

```
1 block
```

```


2      : '{' blockStatements '}'
3      ;
4
5 blockStatements
6      : blockStatement*
7      ;
8
9 blockStatement
10     : variableDeclarators ';'
11     | statement
12     | functionDeclaration
13     | classDeclaration
14     ;

```

也就是说，block 是由 blockStatements 构成的，而 blockStatements 可以由 0 到 n 个 blockStatement 构成，因此可能产生 ϵ 。

接下来，我们来看看解析 block 时会发生什么。

假设花括号中一个语句也没有，也就是 blockStatements 实际上产生了 ϵ 。那么在解析 block 时，首先读取了一个 Token，即 “{”，然后处理 blockStatements，我们再预读一个 Token，发现是 “}”，那这个右花括号是 blockStatement 的哪个产生式的呢？实际上它不在任何一个产生式的 First 集合中，下面是进行判断的伪代码：

 复制代码


```

1 nextToken = tokens.peek();           //得到'}'
2 nextToken in First(variableDeclarators) ? //no
3 nextToken in First(statement) ?       //no
4 nextToken in First(functionDeclaration) ? //no
5 nextToken in First(classDeclaration) ? //no

```

我们找不到任何一个可用的产生式。这可怎么办呢？除了可能是 blockStatements 本身产生了 ϵ 之外，还有一个可能性就是出现语法错误了。而要继续往下判断，就需要用到 Follow 集合。

像 blockStatements 的 Follow 集合只有一个元素，就是右花括号 “}” 。所以，我们只要再检查一下 nextToken 是不是花括号就行了：

 复制代码

```
1 //伪代码
2 nextToken = tokens.peek();           //得到'}'
3 nextToken in First(variableDeclarators) ? //no
4 nextToken in First(statement) ?      //no
5 nextToken in First(functionDeclaration) ? //no
6 nextToken in First(classDeclaration) ? //no
7
8 if (nextToken in Follow(blockStatements)) //检查Follow集合
9     return Epsilon;                     //推导出ε
10 else
11     error;                             //语法错误
```

那么怎么计算非终结符 x 的 Follow 集合呢？

扫描语法规则，看看 x 后面都可能跟哪些符号。

对于后面跟着的终结符，都加到 Follow(x) 集合中去。

如果后面是非终结符，就把它的 First 集合加到自己的 Follow 集合中去。

最后，如果后面的非终结符可能产出 ϵ ，就再往后找，直到找到程序终结符号。

这个符号通常记做 \$，意味一个程序的结束。比如在表达式的语法里，expression 后面可能跟这个符号，expression 的所有右侧分支的后代节点也都可能跟这个符号，也就是它们都可能出现在程序的末尾。但另一些非终结符，后面不会跟这个符号，如 blockstatements，因为它后面肯定会有 “}” 。

你可以参考 [LLParser](#) 类的 `calcFollowSets()` 方法，这里也要用到不动点法做计算。运行程序可以打印出示例语法的 Follow 集合。我把程序打印输出的 First 和 follow 集合整理如下（其实打印输出还包含一些中间节点，这里就不展示了）：

非终结符	First集合	Follow集合
expression	IntLiteral ID () \$
assign	IntLiteral ID () \$
assign 1	= ϵ) \$
equal	IntLiteral ID () \$ =
equal 1	!= == ϵ) \$ =
rel	IntLiteral ID () \$!= == =
rel 1	>= > <= < ϵ) \$!= == =
add	IntLiteral ID () \$!= == >= > <= < =
add 1	+ - ϵ) \$!= == >= > <= < =
mul	IntLiteral ID () \$!= == >= > <= < + - =
mul 1	* / ϵ) \$!= == >= > <= < + - =
pri	IntLiteral ID () \$!= == >= > <= < + - * / =

在表达式的解析中，我们会综合运用 First 和 Follow 集合。比如，对于 “add1 -> + mul add1 | ϵ ”，如果预读的下一个 Token 是 +，那就按照第一个产生式处理，因为 + 在 First(“+ mul add1”) 集合中。如果预读的 Token 是 > 号，那它肯定不在 First(add1) 中，而我们要看它是否属于 Follow(add1)，如果是，那么 add1 就产生一个 ϵ ，否则就报错。

LL 算法和文法


现在我们已经建立了对 First 集合、Follow 集合和 LL 算法计算过程的直觉认知。这样再写出算法的实现，就比较容易了。用 LL 算法解析语法的时候，我们可以选择两种实现方式。

第一种，还是采用递归下降算法，只不过现在的递归下降算法是没有任何回溯的。无论走到哪一步，我们都能准确地预测出应该采用哪个产生式。

第二种，是采用表驱动的方式。这个时候需要基于我们计算出来的 First 和 Follow 集合构造一张预测分析表。根据这个表，查找在遇到什么 Token 的情况下，应该走哪条路径。


这两种方式是等价的，你可以根据自己的喜好来选择，我用的是第一种。关于算法，我们就说这么多，接下来，我们谈谈如何设计符合 LL(k) 特别是 LL(1) 算法的文法。

我们已经知道左递归的文法是要避免的，也知道要如何避免。除此之外，我们要尽量抽取左公因子，这样可以避免 First 集合产生交集。举例来说，变量声明和函数声明的规则在前半截都差不多，都是类型后面跟着标识符：

 复制代码

```
1 statement : variableDeclare | functionDeclare | other;
2 variableDeclare : type Identifier ('=' expression)? ;
3 funcationDeclare : type Identifier '(' parameterList ')' block ;
```


具体例子如下：

 复制代码

```
1 int age;
2 int cac1(int a, int b){
3     return a + b;
4 }
```

这样的语法规则，如果按照 LL(1) 算法，First(variableDeclare) 和 First(funcationDeclare) 是相同的，没法决定走哪条路径。你就算用 LL(2)，也是一样的，要用到 LL(3) 才行。但对于 LL(k) $k > 1$ 来说，程序开销有点儿大，因为要计算更多的集合，构造更复杂的预测分析表。

不过这个问题很容易解决，只要把它们的左公因子提出来就可以了：

 复制代码

```
1 statement: declarator | other;
2 declarator : declarePrefix (variableDeclarePostfix
3             |functionDeclarePostfix) ;
4 variableDeclarePostfix : ('=' expression)? ;
5 functionDeclarePostfix : '(' parameterList ')' block ;
```


这样，解析程序先解析它们的公共部分，即 `declarePrefix`，然后再看后面的差异。这时，他俩的 First 集合，一个 `{ = ; }`，一个是 `{ (}`，两者没有交集，能够很容易区分。

课程小结

本节课我们比较全面地梳理了自顶向下算法。语法解析过程可以看做是对图的遍历过程，遍历时可以采取深度优先或广度优先的策略，这里要注意，你可能在深度优先遍历的时候，误用广度优先的思路。

针对 LL 算法，我们通过实例分析了 First 集合和 Follow 集合的使用场景和计算方式。掌握了这两个核心概念，特别是熟悉它们的使用场景，你会彻底掌握 LL 算法。

一课一思

处理 ϵ 是 LL 算法中的关键点。在你熟悉的语言中，哪些语法会产生 ϵ ，你在做语法解析的时候会怎样处理它们？欢迎在留言区分享你的思考。

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。

本节课的示例代码我放在了文末，供你参考。

lab/16 ~ 18（算法篇的示例代码）：[码云](#) [GitHub](#)

LLParser.java（LL 算法的语法解析器）：[码云](#) [GitHub](#)

© 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 (17)



沉淀的梦想

2019-09-22

Antlr 中 LL(k) 中 k 是多少，是 Antlr 根据我们的文法动态决定的吗？还是老师文中说的那些写 LL 文法的注意点，我们在写 Antlr 文法的时候需要注意吗？Antlr 会帮助我们自动处理这些吗？

作者回复: 是的。它会自动处理。

Antlr这个工具做了大量的工作, 让开发者编写语法的时候, 可以效率更高。

这样的工具, 也不是采用固化的LL(1)或LL(2)算法, 而是有能力根据语法去决定解析策略。实在不行了还可以回溯。你会发现: 1.生产中使用的编译器, 会综合采用多种技术, 而不仅仅是单纯的采用某个算法。

2.如果要写符合LL(1)算法的语法, 其实语法很啰嗦。这是在编译技术发展的早期, 计算能力有限, 大家更重视执行的效率。而现在, 计算能力很强, 可以更加照顾开发者效率, 而不是计算效率。所有Antlr的语法很友好, 很人性化。从表达式的语法就可见一斑。



👍 9



czh

2020-01-03

今日份总结: 今天是一个扫盲的学习, 有以下两点总结

1.编译的过程: 词法分析 语法分析 语义分析

1.1词法分析: 读取的内容是字符, 根据词法规则输出token。几乎不涉及语言的语法特性, 是编译器的基础。

1.2语法分析:读取的内容是token, 输出的是语法树AST。语言的表达式等功能又这部分中定义的上下文无关文法来实现。

1.3语义分析:操作的对象是AST, 所谓语义主要完成上下文相关的推理逻辑, 如类型问题, 定义声明问题等

2.说说我对编译原理的初次见面感觉: 编译原理相比于其他计算机基础知识而言, 他的难主要集中在需要高度的对现实生活规则的抽象能力、逻辑思维能力, 否则写不出没问题的上下文无关文法规则, 以及无法发现、处理其中蕴含的一些“逻辑坑”, 如左递归等问题。而一些其他的知识点, 如算法部分, 这些其实相比于抽象能力来说, 就要简单、通用、好理解的多, 更加考验你的编程基础, 而不是脑子。

作者回复: 谢谢你分享自己的感受。

我再加几句。你说的对。编译器的前端, 带有很强的形式体系的特征。形式语言能够描述程序的语法, 也能用于描述数学上的公理体系。逻辑学、哲学领域, 通常也需要这样抽象级别的体系。所以抽象程度确实挺高。



👍 4



沉淀的梦想

2019-09-22

还是不太明白为什么要有Follow集这个东西，如果First集中查找不到的话，直接将推导为 ϵ ，然后接着去推导下一个，如果发现不在下一个的First集中再报错，好像也不会有什么性能损失，那为什么要费那么大力气构建Follow集呢？

作者回复：把箭头指在线上这种画法确实不大好，会有歧义。我回头更新一版图，让箭头指向每个存储位置的格子上。

共 3 条评论 >

👍 3



军

2021-11-08

first集合和子集构造法很像呢



👍 1



墨灵

2020-06-28

<https://github.com/moling3650/Frontend-01-Template/blob/master/week12/ast.js>

用JavaScript写了一个四则计算器，总算搞明白产生式和LL算法的对应关系了，这课真是太不容易了，对于一个前端来说。

作者回复：恭喜你！

并且，你不是孤独的。我们公司的一名前端工程师，已经被我忽悠到编译的道路上了:-)

而且，编译技术在完成很多高级的前端工作方面大有可为！



👍 1



Geek_f9ea2d

2019-09-28

老师好，对First集合我基本能理解，对Follow集合的计算，我看的有点懵，这个方法：addToRightChild 为什么需要：把某个节点的Follow集合，也给它所有右边分枝的后代节点？

作者回复：因为这些孩子节点是父节点最右边的。那么父节点后面会跟什么终结符，这些子节点也会跟这些终结符。

如果一个非终结符位于上一级产生式的最右边，比如：A->abcdB中的B，（我们用大小写区分终结符和非终结符）那么找到可能出现在它右边的终结符，实际上不是那么好找。要看看A后面都可能跟啥，比如：C->abcAb，那么A的Follow集合中有b，B的Follow集合中也要有b。

实际上，我觉得自己的这个实现比较笨拙，受限于我采用的GrammarNode这样的数据结构。后面有时间的话，我再写个更加简洁的算法给大家参考。

共 2 条评论 >

👍 1



温雅小公子

2022-10-09 来自河北

那个pri结点应该是蓝色吧，他的子节点是或的关系。



if...else...

2021-10-16

哈哈，看到一愣一愣的



coconut

2021-04-11

和某评论一样有一个疑问，为什么要计算Follow集合？

似乎用First集合就可以实现不回溯的递归下降算法。

遇到下面的文法，如果token不在 $\text{First}(+ \text{ mul add1})$ 中，就直接匹配 ϵ 。也不一定要计算 $\text{Follow}(\text{add1})$

$\text{add1} \rightarrow + \text{ mul add1} \mid \epsilon$

共 2 条评论 >



yydsx

2020-04-15

class LLParser 里面的 241行

```
if (i == grammar.getChildCount()) {  
    rightChildren.add(left);  
}
```

是不是应为

```
if (i == grammar.getChildCount()-1) {  
    rightChildren.add(left);  
}
```

}

如果 `i == grammar.getChildCount()` 那么花括号里面的代码将永远不会执行



瓜瓜

2020-02-05

这个符号通常记做 \$，意味一个程序的结束。比如在表达式的语法里，`expression` 后面可能跟这个符号，`expression` 的所有右侧分支的后代节点也都可能跟这个符号，也就是它们都可能出现在程序的末尾。但另一些非终结符，后面不会跟这个符号，如 `blockstatements`，因为它后面肯定会有“}”。

这一段看了好几遍，没有看懂，老师能不能再解释下？

作者回复：就是说，根据语法规则，有的非终结符可能出现在程序的末尾的，另一些非终结符永远也不可能出现在程序末尾。

比如，在语法规则中，`blockStatements`是`block`的一部分，也只出现在这一个地方。而`block`呢，前后一定环绕着花括号，这就导致了`blockStatements`后面必然是跟着“}”的。

换句话说，`block`是可能出现在程序结尾的，而`blockStatement`不可能。



LeeR

2019-12-01

老师你好，\$ 是不是就是EOF符号，表示程序和文件的结束？

作者回复：\$是整个输入串右边的结束标记（endmarker）。

我们可以用EOF表示这个结束标记，因为这个时候源代码文件已经到结尾了。但理论上你还可以用某个特殊的Token来表示程序结束，只不过不常见罢了。



余晓飞

2019-10-29

我把程序打印输出的 First 和 follow 集合整理如下（其实打印输出还包含一些中间节点，这里就不展示了）：

这段下面的图中 `assign1` 的First 集合应该包含 `Epsilon`

作者回复: 你说的没错。谢谢你的细心!

带1的非终结符 (assign1, equal1, rel1, add1, mul1) 的First结合的都包含Epsilon。

运行LLParser.java会输出正确的First集合。

我让编辑同学改一下图。



余晓飞

2019-10-23

```
expression : assign ;  
assign : equal | assign1 ;  
assign1 : '=' equal assign1 | ε;
```

文中这里第二行 assign 是不写错了?

我看代码SimpleGrammar.java中有这一行GrammarNode assign = exp.createChild("assign", GrammarNodeType.And);

注释中刚好缺了关于assign的内容。

作者回复: 没有写错。

是注释没有跟代码同步, 少了赋值表达式的规则, 已经修改过了。

运行LLParser, 可以dump这个语法规则。



余晓飞

2019-10-02

下图是上述推导过程建立起来的 AST, “1、2、3……”等编号是 AST 节点创建的顺序

对这段话后前后两幅图有疑惑, 前面一副图中的第4行是怎么直接到第5行的,

如果通过下面右递归版的产生式推导似乎省略了一步?

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

```
mul -> pri | pri * mul
```

```
pri -> Id | Num | (add)
```

后面一幅图中节点8, 9, 10在节点12, 13之前生成, 似乎这与前一幅图第6到8行的展开顺序不一致?

作者回复: 你看得很细。上下两个图没配起来, 两张图用的语法规则是不同的, 插图的时候插错了, 而且推导过程跳了步骤, 我修改一下!
多谢你帮我发现!



余晓飞

2019-10-01

这样就形成了四条搜索路径, 分别是 $\text{mul}+\text{mul}$ 、 $\text{add}+\text{mul}+\text{mul}$ 、 $\text{add}+\text{pri}$ 和 $\text{add}+\text{mul}+\text{pri}$ 。这里最后一个是不是应该为 $\text{add}+\text{mul}*\text{pri}$

作者回复: 没错, mul 展开成 $\text{mul}*\text{pri}$ 。笔误了。多谢指出!



沉淀的梦想

2019-09-22

<https://github.com/RichardGong/PlayWithCompiler/blob/master/lab/16-18/src/main/java/play/parser/LLParser.java#L242>

作者回复: 这句看不懂? 我抽空多加点注释。

