LL(*)：ANTLR 解析器生成器的基础

草案

已被 2011 年 PLDI 接收

特伦斯-帕尔
旧金山大学
parrt@cs.usfca.edu

Kathleen S. Fisher
AT&T 实验室研究
kfisher@research.att.com

摘要

尽管解析器表达式语法 (PEG) 和 GLR 功能强大，但解析问题并没有解决。在传统的 *LL* 和 *LR 解析器中*添加非终结性（解析器推测）会导致意想不到的解析时行为，并在错误处理、单步除错和嵌入式语法动作的副作用等方面产生实际问题。本文介绍了 *LL(*)* 解析策略和相关的语法分析算法，该算法可从 ANTLR 语法构建 *LL(*)* 解析决策。在解析时，决策会根据解析决策和输入符号的复杂性，从传统的固定 $k \geq 1$ 的前瞻性（lookahead）优雅地节流到任意前瞻性（anywhere lookahead），最后失效到反向跟踪（backtracking）。*LL(*)*的解析能力达到了上下文敏感语言的水平，在某些情况下甚至超出了GLR和PEG所能表达的范围。通过静态消除尽可能多的猜测，*LL(*) 在*提供 PEG 表达能力的同时，还保留了 *LL* 良好的错误处理能力和不受限制的语法操作。ANTLR 的广泛使用（每年下载量超过 70,000 次）表明，它在各种应用中都非常有效。

1. 导言

尽管解析问题非常重要，而且学术研究历史悠久，但它并不是一个已经解决的问题。由于手工编写解析器既乏味又容易出错，研究人员花费了数十年时间研究如何从高级语法生成高效的解析器。尽管如此，解析器生成器仍然存在表达能力和可用性方面的问题。

解析理论最初提出时，机器资源稀缺，因此解析器的效率是最重要的考虑因素。在那个时代，迫使程序员修改语法以适应 *LALR*(1) 或 *LL*(1) 解析器生成器的要求是合理的。与此相反，现代计算机的速度非常快，程序员的效率变得更加重要。为了应对这一发展，研究人员开发出了功能更强大但成本更高的非确定性解析策略，既采用了 "自下而上 "的方法（*LR 式解析*），也采用了 "自上而下 "的方法（*LL 式*解析）。

在 "自下而上 "的世界里，*广义 LR*（GLR）[16] 解析器的解析时间从线性时间到立方时间不等，这取决于语法与经典 *LR 的*符合程度。GLR 本质上是 "分叉 "新的子解析器，从非确定的 *LR* 状态出发，追踪所有可能的动作，并终止任何导致无效解析的子解析器。这样就形成了一个包含所有可能的输入解释的解析森林。*猎犬*[10] 是一种非常高效的 GLR 实现，当语法为 *LALR*(1) 时，它能达到*类似 yacc 的*解析速度。不过，不熟悉 *LALR* 解析理论的专业人员很容易得到非线性 GLR 解析器。

在 "自上而下 "的世界里，福特引入了 *Packrat* 解析器和相关的解析*器表达式语法*（PEG）[5, 6]。PEG 只排除使用左递归语法规则。Packrat 解析器是一种回溯式解析器，会按照指定的顺序尝试备选生成物。在输入位置上最先匹配的生成语句胜出。包鼠解析器是线性的，而不是指数型的，因为它们将部分结果记忆化，确保输入状态不会被同一个生成器解析多次。*老鼠们*
[7] 基于 PEG 的工具大力优化消除 memoization 事件，以提高速度并减少内存占用。

GLR 和 PEG 解析器生成器的一大优势是，它们可以接受任何符合其元语言的语法（左递归 PEG 除外）。专业语法分析人员无需再阅读大量的冲突信息。尽管有这样的优势，但 GLR 和 PEG 解析器都不能完全令人满意，原因有很多。首先，GLR 和 PEG 解析器并不总是能达到预期的效果。GLR 默认接受*模棱两可的语法*，即以多种方式匹配同一输入的*语法*，这就迫使程序员动态地检测模棱两可的*语法*。PEG 没有语法冲突的概念，因为它们总是选择 "第一 "解释，这可能导致意想不到或不方便的行为。例如，PEG 规则 $A \rightarrow a|ab$（意思是 "*A* 要么匹配 *a*，要么匹配 *ab*"）的第二种解释永远不会被使用。因为第一个符号 *a* 匹配的 是 第一个选项，所以输入 *ab* 永远不会匹配第二个选项。在大型语法中，这种危险并不总是显而易见的，即使是经验丰富的开发人员也可能会错过没有经过详尽的测试。

其次，调试非确定性解析器可能非常困难。在自下而上的解析过程中，状态通常代表语法中的多个位置，因此程序员很难预测下一步会发生什么。自上而下的解析器更容易理解，因为从 *LL* 语法元素到解析器操作之间存在一一对应的映射关系。此外，递归后裔 *LL* 实现允许程序员使用标准的源代码级调试器来逐步完成解析器和嵌入式操作，从而便于理解。然而，对于回溯递归后裔的 packrat 分析程序来说，这一优势被大大削弱了。嵌套回溯非常难以跟踪！

第三，在非终结式解析器中生成高质量的错误信息非常困难，但对商业开发人员来说却非常重要。提供良好的语法错误支持取决于解析器的上下文。例如，要从无效表达式中很好地恢复，解析器需要知道它是在解析数组索引还是赋值。在第一种情况下，解析器应通过跳转到] 标记来重新同步。在第二种情况下，解析器应该跳转到;标记。自上而下的解析器有一个规则调用堆栈，可以报告类似 "*数组索引中的表达式无效*"的情况。自下而上的解析器

另一方面，它们只能确定它们正在匹配一个表达式。它们通常无法很好地处理错误输入。Packrat 分析程序的上下文也很模糊，因为它们总是在猜测。事实上，它们无法从语法错误中恢复，因为在看到整个输入之前，它们无法检测到错误。

最后，非确定性解析策略无法轻松支持任意的嵌入式语法操作，而这些操作对于操作符号表、构建数据结构*等*非常有用。推测解析器无法执行打印语句等有副作用的操作，因为推测的操作可能永远不会真正发生。在 GLR 解析器中，即使是计算规则返回值等无副作用的操作也会很麻烦[10]。例如，由于解析器可以用多种方式匹配同一规则，它可能不得不执行多个相互竞争的操作。(它是应该以某种方式合并所有结果，还是只选择其中一个？）GLR 和 PEG 工具解决这个问题的方法是禁止操作、禁止任意操作，或者依靠程序员来避免可能被投机执行的操作产生副作用。

1.1 ANTLR

本文介绍的 ANTLR 解析器生成器 3.3 版本及其底层自上而下的解析策略（称为 *LL(*)*）可以解决这些缺陷。ANTLR 的输入是一个无上下文语法，并添加了*句法*[14]和*语义*谓词以及嵌入动作。语法谓词允许任意前瞻，而语义谓词则允许在谓词之前构建的状态来指导解析。句法谓词以语法片段的形式给出，必须与下面的输入相匹配。语义谓词以解析器宿主语言中的任意布尔值代码形式给出。操作是用解析器的宿主语言编写的，可以访问当前状态。与 PEG 一样，ANTLR 要求程序员避免使用左递归语法规则。

本文的贡献在于：1）自上而下的解析策略 *LL(*)*；2）从 ANTLR 语法构建 *LL(*)* 解析决策的相关静态语法分析算法。*LL(*)*解析器背后的关键思想是使用正则表达式，而不是固定常量或使用完整解析器的回溯来进行前瞻。分析器*会为语法中的每个非终端*构建一个确定性有限自动机（DFA），以区分不同的生成物。如果分析无法为某个非终端找到合适的 DFA，就会退回到反向跟踪。因此，*LL(*)* 分析程序可以从传统的固定 $k \geq 1$ 的前瞻性（lookahead）到任意的前瞻性（lookahead），最后根据分析决定的复杂程度，从容地过渡到反向跟踪（backtracking）。即使在同一解析决策中，解析器也会根据输入序列动态决定策略。一个决策*可能*需要超前扫描或回溯，但这并不意味着它在解析时会对每个输入序列都*这样做*。在实践中，*LL(*)* 分析程序平均只向前看一到两个词组，尽管偶尔需要回溯（第 6 节）。*LL(*)* 分析程序是具有超强决策引擎的 *LL* 分析程序。

这种设计使 ANTLR 具有自顶向下解析的优点，而没有频繁推测的缺点。尤其是，ANTLR 接受除左递归上下文自由语法之外的所有语法，因此与 GLR 或 PEG 解析一样，程序员不必为了适应解析策略而扭曲自己的语法。与 GLR 或 PEG 不同，ANTLR 可以静态识别某些语法歧义和死产。ANTLR 生成的语法是自上而下、递归-后裔的，大多是非

这意味着它支持源代码级除错、生成高质量的错误信息，并允许程序员嵌入任意操作。对 sorceforce.net 和 code.google.com 提供的 89 个 ANTLR 语法[1]的调查显示，保守计算，75% 的 ANTLR 语法都嵌入了动作，这表明此类动作在 ANTLR 社区是一个有用的功能。

广泛的使用表明，*LL(*)*符合语法舒适区的要求，对各种语言应用都很有效。根据 Google Analytics 的数据（2008 年 1 月 9 日至 2010 年 10 月 28 日的独立下载次数），ANTLR 3.x 已被下载 41,364 次（二进制 jar 文件）+ 62,086 次（集成到 ANTLRworks 中）+ 31,126 次（源代码）= 134,576 次。使用 ANTLR 的项目包括谷歌应用引擎（Python）、IBM Tivoli Identity Man- ager、BEA/Oracle WebLogic、Yahoo！查询语言、Ap- ple XCode IDE、Apple Keynote、Oracle SQL Developer IDE、Sun/Oracle JavaFX 语言和 NetBeans IDE。

本文的结构如下。我们首先举例介绍 ANTLR 语法（第 2 节）。接下来，我们正式定义*谓语语法*和一种称为*谓语* LL-正则语法的特殊子类（第 3 节）。然后，我们将介绍 *LL(*)* 解析器（第 4 节），它可以实现谓语 *LL 不规则*语法的解析决策。接下来，我们给出了一种从 ANTLR 语法构建前瞻 DFA 的算法（第 5 节）。最后，我们支持我们关于 *LL(*)* 效率和减少猜测的主张（第 6 节）。

2. 实验室简介*(*)*

在本节中，我们将通过举例说明 *LL(*)* 解析是如何在两个 ANTLR 语法片段中运行的，从而给出 *LL(*)* 解析的直觉。考虑非终结符 s，它使用（省略的）非终结符 expr 来匹配算术表达式。

```
s : ID
  | ID '=' expr
  | 'unsigned'* 'int' ID
  | 'unsigned'* ID ID
  ;
```

非终端 s 可匹配一个标识符 (ID)、一个 ID 后跟一个等号然后跟一个表达式、零次或多次出现无符号字面后跟 int 字面后跟一个 ID，或者零次或多次出现无符号字面后跟两个 ID。ANTLR 语法使用类似 *yacc 的*语法，带有扩展 BNF (EBNF) 操作符（如 Kleene star (*)）和单引号中的标记字面。

当应用到该语法片段时，ANTLR 的语法分析产生了图 1 中的 *LL(*)* 前瞻 DFA。在 s 的决策点，ANTLR 对输入运行该 DFA，直到达到接受状态，然后选择接受状态预测的 s 的替代方案。

尽管我们需要任意的前瞻性来区分 3 个*rd* 和 4 个*th* 备选方案，但前瞻性 DFA 使用的是*每个输入序列的*最小前瞻性。当输入

时，DFA 会立即预测出第三个备选方案（$k = 1$）。当 T（—个 ID）来自 T x 时，DFA

需要看到 $k = 2$ 标记来区分备选方案 1、2 和 4。只有在无符号的情况下，DFA 才需要任意向前扫描，寻找能区分备选方案 3 和 4 的符号（int 或 ID）。

s 的前瞻语言是有规则的，因此我们可以用 DFA 与之匹配。然而，对于递归规则，我们通常会发现前瞻语言是无上下文的，而不是有规则的。在这种情况下，如果

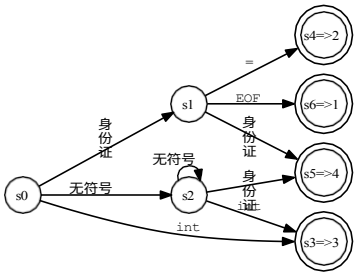


图 1.规则 s 的 *LL(*)* 前瞻 DFA。
sn => *i* 表示 "预测第 i 个备选方案"。

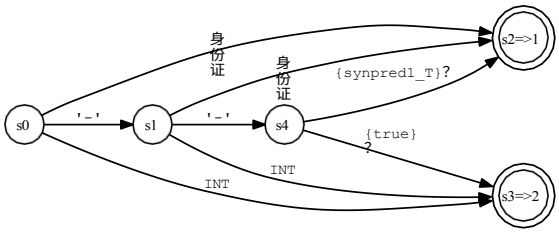


图 2.规则 s2 的 *LL(*)* 解析决策 DFA
使用混合 $k \le 3$ 的前瞻和回溯方法

程序员要求通过添加语法谓词来实现这一功能。为了方便起见，选项 backtrack=true 会自动将语法谓词插入到每个语句中，我们称之为 *"PEG 模式"*，因为它模仿了 PEG 分析器的行为。不过，在采用回溯之前，ANTLR 的分析算法会构建一个 DFA，添加一些额外的状态，使其在许多输入情况下都能避免回溯。在下面的规则 s2 中，两个备选方案都可以从任意数量的-否定符号开始；第二个备选方案使用递归规则 expr 来这样做。

```
选项 {backtrack=true;}// auto-insert syntactic preds s2 :
'-'* ID | expr ';' ;
expr : INT |  '-' expr ;
```

图 2 显示了 ANTLR 针对该输入构建的前瞻性 DFA。在输入 x 或 1 时，该 DFA 只需查看第一个符号，就能立即选择合适的替代方案。当输入-个符号时，DFA 会先匹配几个-，然后再反向追踪。ANTLR 在回溯之前解除递归规则的次数为由内部常数 *m* 控制，我们将其设置为 1 的例子。尽管有可能走回头路，在实践中，除非输入的信息是正确的，否则决定不会出错。以"--"开头，这是一个不太可能的表达式前缀。

3. 预设语法

要精确描述 *LL(*)* 分析，我们首先需要正式定义谓词语法，并从中衍生出 *LL(*)* 分析。一个谓词语法 $G = (N, T, P, S, \Pi, M)$ 包含多个元素：

- N 是非终结项（规则名称）的集合
- T 是终端（标记）集合
- P 是产品集
- $S \in N$ 是起始符号
- Π 是一组无副作用语义谓词
- M 是一组动作（突变器）

$A \in N$	非终端
$a \in T$	终端
$X \in (N \cup T)$	语法符号
$\alpha, \beta, \delta \in X^*$	语法符号序列
$u, x, y, w \in T^*$	终端序列
$w_r \in T^*$	其余输入端子
g	空字符串
$\pi \in \Pi$	宿主语言中的谓词
$\mu \in M$	宿主语言中的动作
$\lambda \in (N \cup \Pi \cup M)$	还原标签
$\rightarrow^{\lambda_1 \ldots \lambda_n}$	还原标签序列
生产规则：	
$A \rightarrow \alpha_i$	i^{th} A 的无上下文生产
$A \rightarrow (A')^i \Rightarrow \alpha_i$	i^{th} 以语法 A 为前提的生产
$A \rightarrow \{\pi_i\}?\alpha_i$	i^{th} 以语义为前提的生产
$A \rightarrow \{\mu_i\}$	i^{th} production with mutator

图 3.谓语语法符号

谓语语法使用图 3 所示的符号编写。语法产物用编号来表示优先级，以此来消除歧义。第一种生成形式表示标准的无上下文语法规则。第二种表示由 *语法谓词* 控制的生成：只有当当前输入也符合 A' 所描述的语法时，符号 A 才会扩展为 α_i。语法谓词可以实现任意的、程序员指定的无上下文前瞻。第三种形式表示由 *语义谓词* 控制的生产：只有当谓词 π_i 对目前构建的状态成立时，符号 A *才会*扩展为 α_i。最后的形式表示动作：应用这样的规则会根据突变器 μ_i 更新状态。

图 4 中的推导规则定义了谓词语法的含义。为了支持语义谓词和突变体，规则引用了状态 S，它在解析过程中抽象了用户状态。为了支持句法谓词，规则引用了 w_r，表示要匹配的剩余输入。判断形式 $(S, \alpha) \Rightarrow^{\lambda} (S', \beta)$ 可以理解为"在机器状态 S 中，语法序列 α 一步还原为修改后的状态 S' 和语法序列 β，同时发出迹 λ"。判断 $(S, \alpha) \Rightarrow (S, \beta)$ 表示重复的 $\rightarrow^{\lambda}$ 一步缩减规则的应用，累积过程中的所有行动。当 λ 与讨论无关时，我们省略它。这些还原规则指定了最左侧的派生。带有语义谓词 π_i 的生成可以只有当当前状态 S 的 π_i 为真时，才会触发。一个带有句法谓词 A *的*生产'，只有当从当前状态下*的*，是剩余输入的前缀，写为 $w \le w_r$。尝试过程中发生的行动来解析 A'^{i} 是*以投机方式*执行的。它们被撤销

$_i$ 是否匹配。最后，动作制作
使用指定的突变器 $_{\mu i}$ 更新状态。

形式上，语法序列 α 生成的语言是 $L(\mathrm{S}, \alpha) = \{w \mid (\mathrm{S}, \alpha) \Rightarrow^* (\mathrm{S}', w)\}$ ，语法 G 的语言是 $L(G) = \{w \mid (g, S) \Rightarrow^* (\mathrm{S}, w)\}$ 。从理论上讲，$L(G)$ 的语言类是递归可数的，因为每个突变器都可能是图灵机。在实践中，语法书写者不会使用这种通用性，因此我们认为该语言类是对上下文敏感的语言。该类语言是上下文敏感语言，而不是无上下文语言，因为谓词可以检查左上下文和右上下文。这种形式主义有各种实际 ANTLR 输入中不存在的语法限制，例如，强制谓词位于规则的左边缘，以及强制突变体进入自己的规则。我们可以在不损失

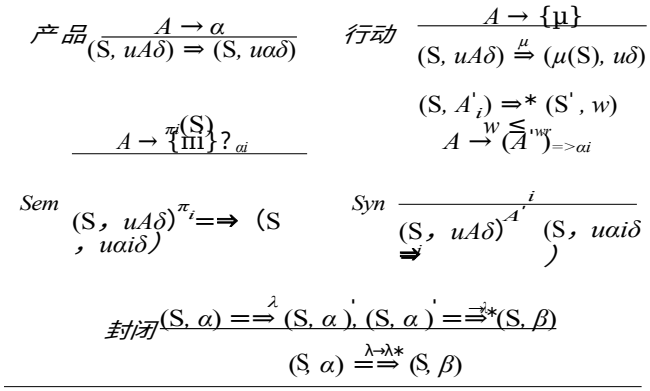


图 4.谓词语法最左派生规则

因为任何一般形式的语法都可以翻译成这种更受限制的形式 [1]。

解析背后的一个关键概念是，在解析过程中的某一特定时刻，生成的结果所匹配的语言。

定义 1.$C(\alpha) = \{w \mid (g, S) \Rightarrow^* (S, u\alpha\delta) \Rightarrow^* (S', uw)\}$ *是生产 α 的*延续语言。

最后，语法位置 $\alpha \cdot \beta$ 表示 "*在 α 之后但在β 之前*"。*在生成或解析过程中*"。

3.1 解决模棱两可的问题

模棱两可的语法是指同一字符串可能有多种识别方式。图 4 中的规则并不排除模糊性。但是，对于一个实用的解析器来说，我们希望每个输入都对应一个唯一的解析。为此，ANTLR 使用语法中的生成顺序来解决歧义问题，冲突将以生成数最小的规则为优先。程序员接到的指令是，让语义谓词对所有可能存在歧义的输入序列互斥，从而使此类语义生成无歧义。然而，由于谓词是用图灵完备语言编写的，因此这一条件无法执行。如果程序员无法满足这一条件，ANTLR 就会使用生产顺序来解决歧义问题。这种策略与 PEG [5, 7] 中的做法相匹配，对于简明地表示先例非常有用。

3.2 谓语 LL *规则*语法

最后还有一个概念有助于理解 *LL(*)* 分析框架，即*预指 LL* 正规 *语法的*概念。在之前的工作中，Jarzabek 和 Krawczyk [8] 以及 Nijholt [13] 将 LL-regular 语法定义为非左递归、无歧义 CFG 的一个特定子集。在这项工作中，我们将 *LL-regular 语法的*概念扩展到谓语 *LL-regular 语法，并*将为其构建高效的 *LL(*)* 分析器。我们要求输入语法是非左递归的；我们使用规则排序来确保语法的无歧义性。

LL-regular 语法与 *LL(k)* 语法的不同之处在于，对于任何给定的非终端，解析器都可以使用整个再主语输入来区分可供选择的生成，而不仅仅是 *k 个*符号。LL *无规则，R 是一个*规则的分区。*如果 x、y ∈ $_{Ri}$，我们写为* $x \equiv y \pmod R$.

定义 3.*如果对于每个非终端 A 的任意两个可供选择的生成扩展，G 都是谓语 LL* 规则的
对 $_{\alpha i}$ 和 $_{\alpha j}$ 而言，存在规则分区 R，使得

$$(g, S) \Rightarrow^* (S, {}_{wiA\delta i}) \Rightarrow (S, {}_{wi\alpha i\delta i}) \Rightarrow^* ({}_{Si}, wix) \qquad (1)$$

$$(g, S) \Rightarrow^* (S, {}_{wj\,A\delta j}) \Rightarrow (S, {}_{wj\alpha j\delta j}) \Rightarrow^* ({}_{Sj}, wjy) \qquad (2)$$

$$x \equiv y \text{（模 } R\text{）} \qquad (3)$$

*规律*语法要求每个非终端 *A 的*所有终端序列集合存在一个有规律的分区，该分区的每个块正好对应 *A* 的一种可能生成。形式上

定义 2.*设 R =* ($_{R1}$, $_{R2}$, … , $_{Rn}$) *是 Rn 的一个分区。*
T 分成 n 个非空且不相交的集合 $_{Ri}$。如果每个集合块 $_{Ri}$*

总是意味着 $\alpha i = \alpha j$ *和* $Si = Sj$。[1]

4. *LL(*)* 分析器

Ni- jholt [13] 和 Poplawski [15] 提出的现有 *LL 规则*语法分析器是线性的，但往往不切实际，因为它们无法解析无限流，如套接字协议和交互式解释器。在两次传递中的第一次传递中，这些解析器必须从右向左读取输入。取而代之的是，我们提出了一种更简单的从左到右、一次通过的方法，称为 *LL(*)*，它将*前瞻性 DFA 移植*到 *LL* 解析器上。前瞻 DFA 与特定非终端相关的正则分区 *R* 匹配，每个 R_i 都有一个接受状态。在决策点，如果 R_i 与剩余的输入相匹配，*LL(*)* 解析器就会扩展生产 i。因此，*LL(*)* 解析器的运算量为 $O(n^2)$，但在实际应用中，它们通常只检查一个或两个标记（第 6 节）。与以前的解析策略一样，每个 *LL* 正规语法都有一个 *LL(*)* 解析器。与之前的工作不同，*LL(*)* 分析器可以将*预先设定的* LL *正则*语法作为输入；它们通过在*前瞻 DFA* 中插入特殊的边来处理谓词，这些边对应于谓词。

定义 4.前瞻性 DFA *是添加了谓词的 DFA，并接受产生预测生产数的状态。形式上，给定谓语语法* $G = (N, T, P, S, \Pi, M)$, *DFA* $M = (S, Q, \Sigma, \Delta, D0, F)$ *其中：*

- S 是继承自周围解析器的系统状态
- Q 是状态集合
- $\Sigma = T \Pi$ 是边缘字母表
- Δ 是映射 $Q \times \Sigma \rightarrow Q$ *的*过渡函数
- $D0 \in Q$ 是起始状态
- $F = \{f1, f2, ., fn\}$ 是最终状态的集合，其中有一个每个常规分区块 R_i（生产 i）的 $fi \in Q$

在 Δ 中，符号 $a \in \Sigma$ 从状态 p 到状态 q *的*转换形式为 $p \xrightarrow{a} q$。谓词转换（写成 $p \xrightarrow{\pi} f$）必须以最终状态为目标，但从 p *开始*的这种转换可以不止一个。DFA 的瞬时配置 c 是（S, p, wr），其中 S 是系统状态，p 是当前状态；初始配置是（$S, D0, wr$）。$c \rightarrow c'$ 表示 DFA 使用图 5 中的规则从配置 c 变为 c'。与谓词格式一样，这些规则并不禁止由谓词转换产生的模棱两可的 DFA 路径。在实践中，ANTLR 会对边缘进行测试，以解决模棱两可的问题。

为了提高效率，前瞻性 DFA 与*前瞻性集合*相匹配而不是延续语言。给定 $R = (\{ac^*\}, \{bd^*\})$、

[1] 严格来说，这个定义对应的是*强 LL*-正则，而不是 Nijholt [13] 指出的 *LL*-正则。*强 LL* 解析器在做决定时会忽略左语境。

i

$$\frac{p \xrightarrow{a} q}{(S、p、aw) \rightarrow (S、q、w)} \qquad \frac{\pi i(S)\ p \xrightarrow{\pi_i} f}{(S、p、w) \rightarrow (S、fi、w)}$$

$$\frac{(S, fi, w)}{\text{接受，预测产量 } i}$$

图 5.前瞻性 DFA 配置更改规则

举例来说，如果我们把目光局限于第一个符号。前瞻集为 ({a}, {b})。

定义 5.*给定分区 R 区分了 n 个不同的产品，产品 i 的前瞻集是 $_{Ri}$ 中仍能唯一预测 i 的最小前缀集：*

$$_{LAi} = \{ w \mid ww' \in {}_{Ri}, w \notin {}_{LAj} \text{ for } j \neq i \text{ and no strict}$$

*w 的前缀具有相同的*属性}

4.1 删除句法谓词

为了避免为句法谓词建立单独的识别机制，我们将句法谓词简化为语义谓词，从而启动推测性解析。为了 "清除 "句法谓词 (A_i')=>，我们用语义谓词代替它。 cate {synpred(A_i')}?.如果 A 匹配当前输入，否则返回 false。要超在移植 PEG "非谓词 "时，我们可以按照福特（Ford）的建议[6]，翻转调用函数 synpred 的结果。

4.2 谓词语法中的任意行为

正式谓词语法在推测过程中会分叉出新的状态 S。在实践中，复制系统状态是不可行的。因此，ANTLR 默认在推测过程中停用突变器，防止动作以推测方式 "发射导弹"。然而，某些语义谓词依赖于突变器所做的更改，例如解析 C 所需的符号表操作。尽可能避免推测可以减轻这一问题，但仍会留下语义隐患。为了解决这个问题，ANTLR 支持一种特殊的操作，用双括号{{...}}括起来，即使在推测过程中也能执行。ANTLR 要求程序员验证这些操作是无副作用的或可撤销的。幸运的是，符号表操作（最常见的 {{...}} 操作）通常可以自动撤销。例如，代码块的规则通常会推入一个符号作用域，但在退出时会弹出。弹出有效地撤销了代码块期间产生的副作用。

5. *LL(*)* 语法分析

对于 *LL(*)*，分析一个语法意味着为每一个解析决定，*即*为语法中具有多重生成的每一个非终端找到一个前瞻性 DFA。在我们的讨论中，我们用 A 作为相关的非终端，用 $i \in 1...n$ 的 αi 作为相应的右边集合。我们的目标是为每个 A 找到一个由 DFA 表示的规则分区 R，它能区分不同的乘积。要想成功，A *必须是 LL 规则的*：分区块 $_{Ri}$ 必须包含 C(α_i)（$_{\alpha i}$ 的延续语言）中的每个句子，并且 $_{Ri}$ 必须是不相交的。为了提高效率，DFA 会匹配前瞻集而不是分区块。

需要指出的是，我们并不是在使用 DFA 进行解析，而只是在预测解析器应该扩展哪种生成。延续语言 C($_{\alpha i}$) 通常是上下文自由的，而不是正则表达式，但经验表明，通常有一种近似的正则表达式可以区分

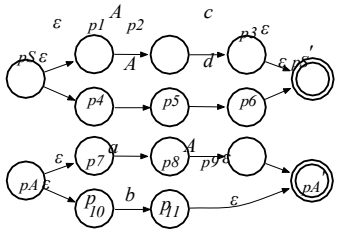


图 6.*P* ={*S* → *Ac* | *Ad*, *A* → *aA* | b} **的** *G* **的 ATN**

输入语法元素	结果 ATN 过渡
$A \rightarrow \alpha_i$	$p_A \xrightarrow{c} p_{A,i}$ $\boxed{\alpha_i}$ $\xrightarrow{c} p'_A$
$A \rightarrow \{\pi\}?$	$p_A \xrightarrow{c} p_{A,i} \xrightarrow{\pi_i}$ $\boxed{\alpha_i}$ $\xrightarrow{c} p'_A$
$A \rightarrow \{\mu\}$	$p_A \xrightarrow{c} p_{A,i} \xrightarrow{\mu_i} p'_A$
$A \rightarrow g$	$p_A \xrightarrow{c} p_{A,i} \xrightarrow{c} p'_A$
$\boxed{\ }_{\alpha i} = X1X2 \cdots Xm$ 对于 $Xj \in NUT, j = 1...m$	$p_0 \xrightarrow{X1} p_1 \xrightarrow{X2} \cdots \xrightarrow{X_n} p_m$

图 7.预设语法到 ATN 的转换

α_i。例如，考虑匹配以下规则的规则 $A \rightarrow$ **[** A **]** | **id** 标识符的平衡括号，*即*无上下文的语言 {**[nid]**^{n}}。用规则表达式逼近 C($_{\alpha i}$) 集，可以得到满足 LL *规则*条件的分区：R = {{[* **id**]* }, {id}}。事实上，第一个输入符号就足以预测替代方案：LA = {{**[**}, {id}}。判定结果为 *LL*(1)。

更糟糕的是，Poplawski [15] 指出 LL-regular 条件是不可判定的，因此我们必须使用启发式方法来防止非终结，有时即使 A 是 *LL-regular 的，*算法也会在找到 R 之前被迫放弃。在这种情况下，我们会采用第 5.3 节和第 5.4 节讨论的其他策略，而不是不创建 DFA。

LL(*) 分析算法首先将输入语法转换为等效的*增强转换网络* (ATN)[17]。然后，它通过模拟 ATN 的动作来计算前瞻性 DFA，这一过程模仿了著名的*子集构造*算法计算模拟 NFA 动作的 DFA 的过程。

5.1 增强过渡网络

给定谓语法 G = (N, T, P, S, Π, M), 相应的 ATN $_{MG}$ = (Q, Σ, Δ, E, F) 有元素：

- Q 是状态集合
- Σ是边缘字母 $N \cup T \cup \Pi \cup M$
- Δ 是过渡关系映射 $Q \times (\Sigma \cup g) \rightarrow Q$
- E = {$_{pA}$ | A∈N } 是子机入口状态集
- F = {p' |A∈ N } 是子机最终状态的集合

我们将很快介绍如何计算 Q 和 Δ。

ATN 类似于记录编程语言的语法图，每个非终端都有一个 ATN 子机。例如，图 6 给出了一个简单语法的 ATN。非终端边 $p \xrightarrow{A} p'$ 就像是函数调用。它们将 ATN 的控制权转移给 A *的*子机器，将返回状态 p' 推入状态

栈，以便在到达 A 的子机器的停止状态后，继续从 p 。'

要从语法中得到 ATN，我们要为每个非终端 A 创建一个子机器，如图 7 所示。起始状态 $_{pA}$ 针对从 $_{ai}$ 左边缘创建的 $pA_{,i}$。最后一个状态

由 α_i 目标 p 创建' 。ATN 匹配的语言与原始语法的语言相同。

语法分析就像是程序间的流程分析，它静态地跟踪程序的 ATN 类图表示，发现从顶层调用站点可到达的所有节点。就流程而言，程序的唯一配置就是图节点和用于到达该节点的调用栈。根据分析类型的不同，它还可能跟踪一些语义上下文，如来自顶级调用站点的 参数。

同样，语法分析会静态追踪从生产 α_i 的 "调用点"（即左边缘状态 $pA_{,i)}$ 出发的 ATN 路径。分析工作一直持续到每个前瞻序列对特定备选方案都是唯一的为止。分析还需要跟踪来自 α_i 左边缘的任何语义谓词 πi ， 以防需要它来解决歧义。因此，*ATN 配置*是一个元组（p, i, γ, π），包含 ATN 状态 p、预测生产 i、ATN 调用堆栈 γ 和可选谓词 π。我们将使用 $c.p$、$c.i$、$c.\gamma$ 和 $c.\pi$ 分别表示从配置 c 中投射状态、可选生产、堆栈和谓词。分析时忽略机器存储空间 S，因为在分析时它是未知的。

5.2 修改后的子集构建算法

出于语法分析的目的，我们修改了子集构造，以处理 ATN 而非 NFA 配置。每个 DFA 状态 D 代表 ATN 从状态 $pA_{,i}$ 开始匹配剩余输入前缀后可能的配置集：

- *闭合*操作模拟 ATN 非终端调用的推送和弹出。
- 如果新发现的状态中的所有配置都预决算了相同的备选方案，则分析不会将该状态添加到工作列表中；无需再进行前瞻性分析。
- 为解决模糊问题，如果存在适当的谓词，算法会在最终状态中添加谓词转换。

该算法的结构与子集结构类似。算法首先创建 DFA 起始状态 $_{D0}$，并将其添加到工作列表中。在没有剩余工作之前，算法会添加新的 DFA 状态，这些状态由*移动*和*闭包*函数计算，模拟 ATN 的转换。我们假定与输入语法 G 相对应的 ATN、$_{MG} = (_{QM}, N \cup T \cup \Pi \cup \mathrm{M}, _{\Delta M}, _{EM}, _{FM})$ 以及我们要分析的非终端 A 都在算法所有操作的范围内。

函数 *createDFA*（如算法 8 所示）是入口点：调用 *createDFA*($_{pA}$)可以为 A 构建前瞻 DFA。为了创建起始状态 $_{D0}$, 算法为每个生产 $A \rightarrow_{\pi i} \alpha_i$ 添加配置（$pA_{,i,} i,$ [], πi），并为每个生产 $A \rightarrow_{\alpha i\ 添加}$配置（$pA_{,i,} i,$ [], -）；符号"-"表示没有谓词。*createDFA* 的核心是一个组合的*移动-关闭*操作，它通过查找每个输入终端符号 a∈ T 可直接到达的 ATN 状态集来创建新的 DFA 状态：

Alg.8: *createDFA*(ATN State $_{pA}$) **返回** DFA

work := []; Δ := {}; $_{D0}$:= {}；

F := $_{fi}$ | $_{fi}$:= 新 DFA 状态，1 . . *numAlts*(*A*)};

前导 $p \xrightarrow{c} p'$; p' $_{A,i} \in \Delta_M$ **做**

如果 $p \xrightarrow{\pi_i} p'$ **则** π := $_{\pi i}$ **否则** π := -;

$\xrightarrow{c}_{D0} p'$ = *closure*($_{D0}$, ($pA_{,i}$, *i*, [], π))；

最后

work += $_{D0}$; *Q* += $_{D0}$；

dfa := *dfa*（-，*q*，*t* ∪ π，Δ，$_{d0}$，*f*）；

取 *mv* := *move*(*D*, *a*) **取** *a*∈*T* **do**

D' := $\bigcup_{c \in mv}$ *closure*(*D*，*c*)；

如果 *D*' ∈/ *Q*

，那么

resolve(*D*')；

switch *findPedredictAlt*(*D*') **do**

case *None: work* += *D*'；

情况*只是 j*：$_{fj}$:= *D*'；

结束语

Q += *D*'；

最后

Δ += $D \xrightarrow{a} D'$；

Δ += $D \xrightarrow{\epsilon \to \pi} f_{c.i}$

最后

最后

工作 -= *D*

foreach *c*∈*D such that wasResolved(c)* **do**

；

最后

返回 *DFA*；

$$move(D, a) = \{(q, i, \gamma, \pi) \mid p \xrightarrow{a} q, (p, i, \gamma, \pi) \in D\}$$

然后添加这些配置的闭包。一旦算法确定了新的状态 *D*'，它就会调用*解析*来检查并解决歧义。如果 *D*' 中的所有配置都预测了相同的备选方案 *j*，那么 *D*' 就会被标记为

$_{fj}$，备选方案 *j* 的接受状态，而 *D*' 不被添加到工作列表：一旦算法能够唯一确定要预测的生产，就没有必要再检查更多的输入。算法就是通过这种优化来构建与最小前瞻集 $_{LAj}$ 匹配的 DFA ，而不是整个剩余输入。接下来，算法会添加一条从 *D* 到 *D* 的边'，这条边的末端是 *a* 。最后，对于每个配置 c∈ *D* ，*如果*有一个谓词解决了一个不连续点，*createDFA 就*会添加一个以 *c.π 为*谓词的过渡，从 *D 到 c.i* 备选方案的最终状态。"*已解决*" （*wasResolved* ）测试会检查 "*解决* "步骤是否将配置 *c* 标记为已被谓词解决。

闭包 *LL(*) 闭包*操作如算法 9 所示，由于 ATN 栈的存在，它比 NFA 子集构造中的*闭包*函数更为复杂。尽管如此，直觉是相同的。当在配置 *c* 上调用时，*closure* 会通过遍历所有非三元组（*即*谓词、非终结符和突变体）的 ATN 边，递归地找到可从 *c* 的状态到达的所有 ATN 状态。

调用 *closure*(*D*, *c*) 时，会将 *c* 所属的 DFA 状态 *D* 作为附加参数。函数开始时会将参数配置添加到*繁忙*列表中，以避免冗余计算和无限循环。为了模拟

ATN 非终端过渡 *p* $\xrightarrow[A]{}$ *p*'，*关闭*重复

配置 *c*，将返回状态 *p*' 推入堆栈。在子机器停止状态 *p*' 时，*闭包*复制 *c*，从堆栈中弹出 *p*' 。如果*闭包*到达 $_{pA}$ *时*堆栈是空的，我们就无法静态地知道是哪条规则调用了闭包。

A.(这种情况只发生在有一条路径从 $pA_{,i}$ 到 p'^{A} 没有末端边缘）。在这种情况下，我们有

Alg.9: *closure*(DFA State D, $c = (p, i, \gamma, \pi)$)
返回集合*封闭*
如果 c∈*D.busy* **则返回** {}; **否则** *D.busy* += *c*;
closure := {c};
如果 $p \in F$ (*即 p 为停止状态*)
if $\gamma = p'\gamma'$ **then** *closure* += *closure*(D, (p', i, γ', π));
else *closure* += *closure*(D, (p_2, i, [], π));
最后
$\forall\, p2 : p1 \xrightarrow{A} p2 \in \Delta_M$
*对 ATN 状态 p 开始的***每个***过渡 t 都要***做**
switch t **do**
case $p \xrightarrow{A} p'$
:
深度 := A 在 γ 中出现的次数;
如果*深度* = 1**，那么**
D.recursiveAlts += i;
if |D.recursiveAlts| > 1 **then**
抛出 *LikelyNonLLRegularException*;
最后
如果*深度*≥ m， (即*最大递归深度*)**，那么**
将 D 标记为递归溢出;
返回*关闭*;
情况 $p \rightarrow q$, $p \xrightarrow{\mu}$ 或 $p \xrightarrow{c} q$ *过渡:*
最后
closure := *closure*(D, (q, i, γ, π))
;
closure += *closure*(D, (p_A, i, $p'\gamma$, π));
结束
返回*关闭*;

假定输入语法中的任何生产 $p \xrightarrow{A} p$ 都可能调用了 A，因此*封闭*必须追逐所有这样的状态 p_2。

如果*闭包*在一个以上的备选方案中检测到递归非终端调用（子机器直接或间接调用自身），它就会抛出异常终止 A *的* DFA 构建；第 5.4 节描述了我们的后退策略。如果*闭包*检测到的递归深度超过内部常数 m，*闭包就会将*状态参数 D 标记为*溢出*。在这种情况下，A 的 DFA 构建会继续进行，但*闭包*不再追踪从 c 的状态和堆栈中得出的路径。我们将在第 5.3 节详细讨论这种情况。

DFA 状态等价分析算法依赖于 DFA 状态等价的概念：

定义 6.*如果两个 DFA 状态的配置集相等，则 $D \equiv D'$ 相等。两个 ATN 配置集*
如果 p、i 和 π 分量相等，$c \equiv c'$，如果 p、i 和 π 分量相等，则 $c \equiv c'$。
相等，它们的堆栈也相等。两个堆栈相等
$\gamma_1 \equiv \gamma_2$*，如果它们相等，如果至少有一个是空的，或者如果一个是另一个的后缀。*

堆栈等价的这一定义反映了*关闭*遇到子机停止状态时的 ATN con- text 信息。由于分析会在 ATN 中搜索所有可能的前瞻序列，因此空堆栈就像是通配符。任何

过渡 p_1 $\xrightarrow{A}$

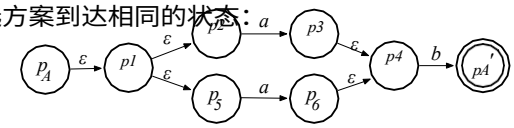
mar $S \rightarrow a|g$, $A \rightarrow SS$。开始状态构造计算 $S \rightarrow \cdot a$ 和 $S \rightarrow \cdot g$ 位置的*闭包*，然后 $S \rightarrow g \cdot$，S 的停止状态。状态栈是空的，因此*闭包会*追逐引用 S *之后的状态*，例如位置 $A \rightarrow S \cdot S$。最后，*闭包*重新进入 S，这次的状态栈不是空的。

给定 D 中的配置（p, , $\gamma 1$, ）和（p, , $\gamma 1\gamma 2$, ），*闭包*按照最近子机器调用的相同序列到达 p，$\gamma 1$。一旦 $\gamma 1$ 跳出，*闭包*就有配置（p, , [], ）和（p, , $\gamma 2$, ）。

解析 静态分析的好处之一是，它有时可以检测到模棱两可的非参数，并向用户发出警告。*闭包*结束后，*解析*函数（算法 10）会在其参数状态 D *中*查找*冲突*配置。

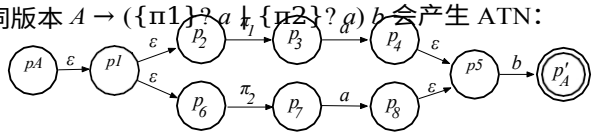
定义 7.*如果 DFA 状态 D 包含配置 $c = (p, i, \gamma_i, \pi_i)$ 和 $c' = (p, j, \gamma_j, \pi_j)$，使得 $i \neq j$ 和 $\gamma_i \equiv \gamma_j$，那么 D 是一个*模棱两可的 DFA 状态，c *和 c' 是*冲突配置。*属于 D 的冲突配置的所有备选数的集合就是 D 的*冲突集。

例如，$A \rightarrow (a|a)\ b$ 中子规则 $(a|a)$ 的 ATN 合并在一起，因此分析可以通过相同（空）的堆栈上下文从两个备选方案到达相同的状态：



D_0 = {（p_2，1），（p_5，2）}，其中我们缩写为（p_2，1，[]，-）。为清晰起见，将(p_2, 1)改为($p2$, *1*)。D_1 = {(p_3, 1), (p_4, 1), (p_6, 2), (p_4, *2*)} 用符号 a 可以到达，其配置 (p_4, 1) 和 (p_4, 2) 互有冲突。由于 *ab 在这*两个备选方案的延续语言中，所以再往前看也无法解决这个歧义。

如果 *resolve* 检测到歧义，它就会调用 *resolveWithPred- icate*（算法 11），查看相互冲突的配置是否有可以解决歧义的谓词。例如，前一个语法的谓词版本 $A \rightarrow (\{\pi 1\}?\ a \mid \{\pi 2\}?\ a)\ b$ 会产生 ATN：



D_0 表示决策状态 p_0，以 {(p_1, 1, [], π_1), (p_2, 2, [], π_2)} 开始我们在其中添加 {(p_3, 1, [], π_1), (p_7, 2, [], π_2)} 以实现*闭合*。D_1 和之前一样存在配置冲突，但现在预设可以在运行时用 DFA $D_0 \xrightarrow{\pi_1} D_1$, $D_0 \xrightarrow{\pi_2} D_2$ 解决这个问题、

如果 *resolve* 找到了谓词，它会返回而不发出警告，让 *createDFA* 将谓词纳入 DFA。如果没有谓词，就无法重新

p_2 可以调用 A，因此分析必须

在运行时*解　决问题*，因此静态解决时会移除
包括每一个这样的 $_{p2}$ 的*闭包*。因此，一个闭包集可以有两种配置 c 和 c'，*它们具有*相同的 ATN 状态，但 $c.\gamma = g$ 和 $c'.\gamma /= g$。只有*当闭*包以空堆栈到达一个非终端的子机器停止状态，并通过追逐对该非终端的引用状态，*闭包*重新进入该子机器时，才会发生这种情况。例如，考虑一下 S 的 DFA 在克-

通过赋予 A 的最低冲突优先权来消除歧义。
通过移除与更高编号的冲突备选方案相关的配置，DFA 可以生成一个新的备选方案。例如，在上述 A 的无预设语法中，得到的 DFA
是 $D' \rightarrow f$，因为分析解决冲突的方法是删除与最高优先级产状 1 无关的配置，剩下 $\{(_{p3}, 1), (_{p4}, I)\}$.

Alg.10: *resolve*(DFA State *D*)
conf licts := D 的冲突集；
if |*conf* licts| = 0 *and not overflowed(D)* **then return**; **if** *resolveW ithPreds*(*D, conf licts*) **then return**；
从 *D* 中删除所有 *c*，使得 *c*.i∈ *conf licts*，从而解决这个问题
且 *c.i* /= *min*（*conf licts*）；
if *overflowed(D)* **则**报告递归溢出；
否则报告语法模糊；

Alg.11: *resolveWithPreds*（DFA State *D*，set *conf licts*）
返回布尔值
pconf igs := []； // 为 alt *i* 配置谓词
foreach *i* ∈ *conf licts* **do**
pconf igs[*i*] := 任选一个代表（, *i*, , *π*）∈ $\bar{D}$；
最后
if |*pconf igs*| < |*conf licts*| **then return false**; **foreach** *c* ∈ *pconf igs* **do** mark *c* as *wasResolved* ; **return** **true**；

如果*闭合*触发了递归溢出警报，*解析*可能不会在 *D* 中看到相互冲突的配置，但 *D* 仍可能预测出多个备选方案，因为分析提前终止以避免非终止。如果存在谓词，算法可以在运行时使用谓词来解决潜在的歧义。如果不存在，算法会再次选择最小的备选方案编号，并向用户发出警告。

5.3 避免分析的棘手性

由于 *LL-regular* 条件是不可判定的，我们预计在任何前瞻性 DFA 构建算法的某个地方都有可能出现无限循环。递归规则是非终止的根源。给定配置 $c = (p, i, \gamma)$（为简洁起见，我们在下文中省略配置中的π），*c* 在过渡 $p \xrightarrow{A} p'$ 时的*闭包*包括 $(p, i, p' \gamma)$。如果*闭包*再次到达 *p*，它将包括 $(_{pA}, i, p\, p'' \gamma)$。最终、*闭包*将永远 "泵送 "递归规则，导致栈爆炸。例如，对于图 6 所示的带有递归非终端 *A* 的 ATN，$S \rightarrow Ac \mid Ad$ 的 DFA 开始状态 $_{D0}$ 是：

$$_{D0} = \{ (_{p1}, 1, []), (_{pA}, 1, _{p2}), (_{p7}, 1, _{p2}), (_{p10}, 1, _{p2}), (_{p4}, 2, []), (_{pA}, 2, _{p5}), (_{p7}, 2, _{p5}), (_{p10}, 2, _{p5})\}$$

函数 *move*($_{D0}$, *a*) 到达（$_{p8}$, 1, $_{p2}$），通过 $_{D0}$ 中的（$_{p7}$, 1, $_{p2}$）创建 $_{D1}$。（$_{p8}$, 1, $_{p2}$）的*闭合*遍历隐含的 *g* 边到 $_{pA}$，增加了三个新配置：（$_{pA}$, 1, $_{p9p2}$）、（$_{p7}$, 1, $_{p9p2}$）、（$_{p10}$, 1, $_{p9p2}$）。$_{D1}$ 的 $_{p7}$ 配置与 $_{D0}$ 相同，$_{9}$ 但堆栈更大。在递归深度为 *m* 时，随着 $p^m{}_{p2}$，配置堆栈永远增长、^{aaa} 产生越来越大的 DFA 路径：$\mathcal{D} \rightarrow \mathcal{D} \rightarrow \cdots \rightarrow D$ 。这个问题有两种解决方案。要么我们或者干脆避免对任何特定子机起始状态进行 *m 次*递归调用来计算配置*闭合*。我们选择后者是因为它保证了 *LL(k)* 的严格超集（当 $m \geq k$ 时）。我们不必担心近似值会在改变者之间引入共同的无效序列。相反，Bermudez 和 Schimpf 给出了一个 *LALR*(1) 语法族，对于这个语法族中的每个语法，都没有固定的 *m* 可以给出有效的 *LAR*(*m*) 解析器。对递归深度的硬性限制并不是一个严重的限制。

Java1.5：本地 ANTLR 语法，使用 PEG 模式。

RatsC, RatsJava：语法，使用 PEG 模式手动转换为 ANTLR 语法，同时保留了基本结构。我们删除了 *Rats!* 支持但 ANTLR 不支持的左递归。

vb.net, tsql, c#：米氏商业语法
由 *Temporal Wave, LLC* 提供的 crosoft 语言。

图 12.基准 ANTLR 语法

实践。程序员可能会使用（常规）语法 $S \rightarrow a^*\ \mathrm{bc} \mid a^*\ \mathrm{bd}$，而不是图 6 中的版本。

5.4 中止 DFA 构建

作为一种启发式方法，当我们发现 *A 的递归*不止一个时，我们就会终止非递归 *A 的* DFA 构建（图 6 中 S 的分析实际上会在触发递归溢出之前终止）。(图 6 中对 *S 的*分析实际上会在触发递归溢出之前终止）。这种决策不太可能有精确的规则分区，而且由于我们的算法并不近似前瞻，因此没有必要徒劳地进行 DFA。ANTLR 的实现对 *A 采用 LL*(1) lookahead，如果*解析*检测到递归不止一个，则采用回溯或其他谓词。

5.5 起重谓词

这种算法和第 3 节中的正式谓词语法语法要求谓词出现在生成的左边缘。这种限制在实践中非常麻烦，会迫使用户重复使用谓词。ANTLR 中的完整算法可以自动发现并*提升*所有对判定可见的谓词，即使是派生链上更下游的生产也不例外。(ANTLR 的分析还能处理产词右侧的 EBNF 运算符，*例如*通过在 ATN 中添加循环来处理 $A \rightarrow a^*\ b$。

6. 经验结果

本文就我们的分析算法和 *LL(*)* 解析策略的适用性和效率提出了一些主张。在本节中，我们在图 12 所示的六个大型真实语法上运行 ANTLR 3.3，并在图 13 所示的大型样本输入句子上对生成的解析器进行了测试，从而为这些说法提供了支持。我们包括两个源于 PEG 的语法，以显示 ANTLR 可以从 PEG 生成有效的解析器。(读者可以查看非商业语法、样本输入、测试方法和原始分析结果[1]）。

6.1 静态语法分析

我们声称，我们的语法分析算法能静态地将回溯从绝大多数解析决策中排除，并且能在合理的时间内完成。表 1 总结了 ANTLR 对六个语法的分析。除了 8,231 行-SQL 语法需要 13.1 秒外，ANTLR 处理每个语法的时间都在几秒之内。分析时间包括输入语法分析、分析（包括 EBNF 结构处理和谓词提升 [1]）以及生成分析器。(与指数级复杂的类集子集构建算法一样，ANTLR 的分析在极少数情况下也会触及 "地雷"；ANTLR 提供了一种方法来等效延迟违规决策，并手动设置其前瞻参数。这些语法都不会触及 "地雷"）。

所有语法都在一定程度上使用了回溯功能，这证明不值得对大型语法进行扭曲。

RatsJavaParser.java：由 ANTLR 生成的解析器，用于 RatsJava 语法；测试两个 Java 解析器。

预 javaParser.c：预处理 ANTLR 生成的解析器用于 Java1.5 语法；测试 RatsC 解析器。

LinqToSqlSamples.cs：微软示例代码；测试 C#；

big.sql：收集微软 SQL 示例；测试 TSQL。

北风.vb.NET Microsoft 示例代码；测试 VB.NET。

图 13.基准输入句

语法	行 *n*		固定	循环	回溯	运行时间
Java1.5	1,022	170	150	1	20 (11.8%)	3.1s
RatsC	1,133	143	111	0	32 (22.4%)	2.8s
RatsJava	744	87	73	6	8 (9.2%)	3s
VB.NET	3,505	348	332	0	16 (4.6%)	6.75s
TSQL	8,231	1,120	1,053	10	57 (5.1%)	13.1s
C#	3,481	217	189	2	26 (12%)	6.3s

表 1.语法决策特征行 "是语法的大小，*n* 是语法中的解析决策数。

固定 "是纯 *LL(k)* 判决的数量，"循环 "是纯循环 DFA（无回溯）的数量，"回溯 "是可能回溯的判决数量，"运行时间 "是处理语法并生成 *LL(*)* 分析器的时间。在以下条件下进行的测试

OS X，2x3.2Ghz 四核英特尔至强处理器，14G 内存。

语法	*LL(k)*	*LL*(1)	前瞻性深度 *k*					
			1	2	3	4	5	6
Java1.5	88.24%	74.71%	127	20	2	1		
RatsC	77.62%	72.03%	103	7	1			
RatsJava	83.91%	73.56%	64	8	1			
VB.NET	95.40%	88.79%	309	18	4	1		
TSQL	94.02%	83.48%	935	78	11	14	9	6
C#	87.10%	78.34%	170	19				

表 2.固定前瞻决策特征

mar，使其成为 *LL(k)*。前三个语法使用的是 PEG 模式，在这种模式下，ANTLR 会自动将一个句法谓词放在每个生成语的左边缘。不过，与 PEG 解析器不同的是，ANTLR 可以在很多情况下静态地避免回溯。例如，在 Java1.5 中，除了 11.8% 的判定外，ANTLR 在其他所有判定中都去除了句法谓词。不出所料，RatsC 语法的回溯判定比例最高，达到 22.4%，因为 C 语言的变量和函数声明与定义从左边缘看是一样的。三个商业语法的作者手动指定了语法谓词，从而降低了前瞻性要求。在 6 个语法中的 4 个中，ANTLR 能够构建循环 DFA 以避免回溯。

表 1 显示，样本语法中的绝大多数判定都是在某个 *k* 条件下的固定 *LL(k)*。表 2 报告了每个前瞻深度的判定数量，显示大多数判定实际上都是 *LL*(1)。该表还显示，ANTLR 几乎能够一直静态地确定 *k*，尽管这个问题

语法	输入行	解析时间	*n*	平均值 *k*	背面 *k*	最大 *k*
Java1.5	12,394	471ms	111	1.09	3.95	114
RatsC	37,019	1,375ms	131	1.88	5.87	7,968
RatsJava	12,394	527ms	78	1.85	5.95	1,313
VB.NET	4,649	339ms	166	1.07	3.25	12
TSQL	794	164ms	309	1.08	2.63	20
C#	3,807	524ms	146	1.04	1.60	9

表 3：解析器决策前瞻深度解析器决策前瞻深度。*n* 是解析过程中覆盖的决策点数量。"avg *k* "是除以决策事 件 的数量。*k* "是平均推测值。深度仅适用于回溯决策事件。"max *k* "是解析过程中遇到的最深前瞻深度。

在 OS X 下使用 Java 1.6.0 运行 *LL(*)* 分析程序时，使用了

2x3.2Ghz 四核英特尔至强处理器，14G 内存。

语法	可以背面	回来	决策事	后轨	返回
Java1.5	19	16	462,975	0.36%	45.22%
RatsC	30	24	1,343,176	16.85%	65.27%
RatsJava	8	7	628,340	14.07%	74.68%
VB.NET	6	3	109,257	0.46%	20.84%
TSQL	29	19	17,394	3.38%	27.01%
C#	24	19	141,055	3.68%	40.22%

在一般情况下是不可判定的。

6.2 解析器运行时剖析

我们声称，在运行时，*LL(*)*解析决策平均只使用几个前瞻性标记，而且解析速度合理。表 3 显示，对于每个示例输入文件，每个决策事件的平均前瞻深度约为一个标记，而 PEG 模式解析器需要近两个标记。PEG 模式解析器的平均解析速度约为每秒 26,000 行。

表 4.解析器决策回溯行为。"能回溯。"是指可能回溯的决策数量。"已回溯 "是样本输入中已回溯的决策数量。"回溯 "是回溯决策事件的百分比。"回溯率 "是指可能回溯的决策在触发时实际回溯的可能性。

前三个语法的树状结构开销为 9,000 美元，其他语法的树状结构开销为 9,000 美元。

仅反向跟踪决策的平均前瞻深度就不到 6 个词组，这说明尽管反向跟踪可以扫描很远的前方，但通常并不如此。例如，解析器可能需要回溯以匹配 C 类型的指定符，但大多数情况下这种指定符看起来像 `int` 或 `int *`，因此可以很快识别出来。没有从 PEG 派生的语法的最大前瞻深度要小得多，这表明作者进行了一些结构重构，以利用 $LL(k)$ 的效率。与此相反，RatsC 语法会回溯整个函数（在一个决策事件中会前瞻 7968 个标记），而不是前瞻到足以区分声明和定义，*即* `int f();` 和 `int f() {...}`。

尽管我们从 *Rats* 中提取了两个样本语法，但它们的语法结构并不完全相同*！*，

我们没有比较解析器的执行时间，也没有比较内存的使用情况。ANTLR 和 *Rats!* 都是实用的解析器生成器，解析速度和内存占用都适合大多数应用。与 ANTLR 不同，*Rats！* 也是无扫描的，这使得记忆缓存难以比较。

如表 1 所示，ANTLR 可以静态地从大多数解析器决策中移除回溯。在运行时，ANTLR 的回溯比静态分析预测的还要少。例如，在静态分析中，我们发现样本语法平均有 10.9% 的决策会出现回溯，但表 4 显示，生成的解析器平均只有 6.8% 的决策事件会出现回溯。非 PEG 派生语法的回溯率仅为 2.5%。部分原因是一些回溯决策管理不常见的语法结构。例如，有 1,120 个决策

但示例输入只练习了其中的 309 种。

最重要的是，决策*可以*回溯并不意味着它一定会回溯。表 4 的最后一列显示，在所有样本语法中，可能回溯的决策（PBD）平均只有一半的时间会回溯。商用 VB.NET 和 TSQL 语法产生的 PBD 只在大约 30% 的决策事件中回溯。有些 PBD 从未触发回溯事件。将表 4 中的前两列进行细分，可以得出完全避免回溯的 PBD 的数量。

我们应该指出，如果没有 memoization，在最坏的情况下，反向跟踪解析器的复杂程度将呈指数级增长。这对于进行大量嵌套反向跟踪的语法来说非常重要。例如，如果我们关闭 ANTLR 备忘化支持，RatsC 语法似乎不会终止。与此相反，VB.NET 和 C# 解析器在没有该支持的情况下却很好。Packrat 解析 [5] 以增加 memoization 缓存的内存为代价，实现了线性解析结果。在最坏的情况下，我们需要隐藏 $O(|N| * n)$ 个决策事件结果（每个非终端决策在每个输入位置上都有一个）。由于 ANTLR 只在推测时进行记忆，因此回溯越少，缓存就越小。

7. 相关工作

解析技术有很多，但目前最主要的两种策略是富田（Tomita）的自下而上的 GLR [16] 和福特（Ford）的自上而下的 packrat 解析 [5]，通常用其相关的元语言 PEG [6] 来表示。两者都是非确定性的，因为解析器可以使用某种形式的推测来做出决定。*LL(*)* 是对 packrat parsing 的优化，正如 GLR 是对 Earley 算法的优化 [4]。语法越符合基本*的 LL* 或 *LR* 策略，解析器在时间和空间上的效率就越高。*LL(*)* 从 $O(n)$ 到 $O(n^2)$ 不等，而 GLR 则从 $O(n)$ 到 $O(n^3)$ 不等。令人惊讶的是，$O(n^2)$ 的潜力来自循环前瞻 DFA 而非回溯（假设我们记忆了）。ANTLR 生成的 *LL(*)* 解析器在实践中是线性的，而且大大减少了猜测，与纯粹的 packrat 解析器相比，减少了记忆化开销。

GLR 和 PEG 往往是无扫描仪的，这在下列情况下是必要的工具需要支持*语法合成*。语法合成意味着程序员可以轻松地将一种语言整合到另一种语言中，或者通过修改和合成现有语法的片段来创建新语法。例如，*Rats*！Jeannie语法优雅地组合了所有C语言和Java语言。*LL(*)* 背后的理念源于 20 世纪 70 年代。*LL(*)* 剖析器没有预设的前瞻 DFA 边沿，可实现 *LL 不规则语法*，该语法由 Jarz- abek 和 Krawczyk [8] 以及 Nijholt [13] 提出。

Nijholt[13]和 Poplawski[15]给出了线性两遍 LL-正则解析策略，在第一遍中必须从右向左解析，需要有限输入流（*即*不是套接字或活动间流）。他们没有考虑语义谓词。

Milton 和 Fischer [11] 在 *LL*(1) 语法中引入了语义谓词，但只允许每个生成词使 用 一个语义谓词来指导解析，并且要求用户指定谓词应该在哪个前瞻集下进行评估。Parr 和 Quong [14] 引入了语法谓词和语义谓词提升（semantic predicate hoisting），即把其他非终结词的语义谓词纳入解析决策的概念。他们没有提供正式的谓词语法规范，也没有提供发现可见谓词的算法。在本文中，我们给出了正式的语法规范，并演示了在语法分析过程中发现有限谓词的方法。

的 DFA 构造。Grimm 在其基于 PEG 的 *Rats* 中支持受限语义谓词[7]和任意动作，但依赖于程序员避免无法撤销的副作用！[7]和任意操作中支持受限语义谓词，但依赖于程序员避免无法撤销的副作用。最近，Jim *等人在*能用 Yakker 处理所有 CFG 的转换器中添加了语义谓词[9]。

ANTLR 1.x 引入了语法谓词作为人工控制的回溯机制。指定生产优先级的能力是一个值得欢迎的副作用，但却没有得到重视。福特用 PEG 形式化了有序替代的概念。在 3.x 之前的 ANTLR 版本中，如果不进行备忘录化，回溯的时间复杂度将达到指数级。福特还通过引入 packrat 解析器解决了这一问题。ANTLR 3.x 用户可以通过一个选项开启记忆化。

*LL-regular 语法*是 *LR-regular* 语法的类似语法 [3]。Bermudez 和 Schimpf [2] 为 *LR-正则语法*提供了一种名为 *LAR*(*m*) 的解析策略。参数 *m* 是一个堆栈治理器，与我们的类似，可以防止分析算法非终止。*LAR*(*m*)建立了一个*LR*(0)模型，并嫁接了前瞻DFA来处理非确定状态。为每一个 *LL-regular* 或 LR-regular 解析决策找到一个规则分区是无法判定的。*LL(*)* 和 *LAR*(*m)* 的分析算法可分别为 LL-regular 或 LR-regular 语法分析决策的一个子集构建一个有效的 DFA。在自然语言领域，Nederhof [12] 使用 DFA 近似整个 CFG，大概是为了获得更快但不太准确的语言成员检查。Nederhof 将规则调用内嵌到特定深度 *m*，有效地模仿了 *LAR*(*m*) 中的常数。

8. 结论

由于语义谓词的存在，*LL(*)* 分析程序的表达能力与 PEG 不相上下，甚至更强。虽然 GLR *可以*接受左递归语法，但它不能像 *LL(*)* 那样识别上下文敏感语言。与 PEG 或 GLR 不同，*LL(*)* 解析器可以执行任意动作，并为调试和错误处理提供良好的支持。*LL(*)* 分析算法构建循环前瞻 DFA 来处理非 LL*(k)* 结构，然后在找不到合适的 DFA 时通过句法谓词进行回溯。实验表明，ANTLR 生成的解析器效率很高，几乎消除了所有的回溯。ANTLR 在实践中得到了广泛应用，这表明 *LL(*) 在*解析频谱中占据了有利位置。

最后，我们要感谢 Sriram Srinivasan 对 *LL(*)*一词的贡献和创造。

参考资料

[1] 文件附录。
http://antlr.org/papers/LL-star/index.html.

[2] BERMUDEZ, M. E., AND SCHIMPF, K. M. Practical arbitrary lookahead LR parsing.*JCSS 41*, 2 (1990).

[3] COHEN, R., AND CULIK, K. LR-regular grammars：LR(k) 语法的扩展。In *SWAT '71*.

[4] EARLEY, J. 一种高效的无上下文解析算法 *CACM 13*, 2 (1970), 94-102.

[5] FORD, B. Packrat parsing：简单、强大、懒惰、线性时间。In *ICFP'02*, pp.

[6] FORD, B. Parsing expression grammars：基于识别的句法基础。In *POPL'04*, pp.

[7] GRIMM, R. 通过模块化语法实现更好的可扩展性。在 *PLDI'06*, pp.

[8] JARZABEK, S., AND KRAWCZYK, T. LL-Regular grammars. *Information Processing Letters 4*, 2 (1975), 31 - 37。

[9] JIM, T., MANDELBAUM, Y., AND WALKER, D. 数据依赖语法的语义和算法。*POPL '10*.

[10] MCPEAK, S., AND NECULA, G. C. Elkhound：快速、实用的 GLR 解析器生成器。In *CC'04*.

[11] MILTON, D. R., AND FISCHER, C. N. LL(k) parsing for attributed grammars.In *ICALP* (1979).

[12] NEDERHOF，M.-J. 无上下文语言正则近似的实际实验。*Comput.Linguist.26*, 1 (2000), 17-44.

[13] NIJHOLT, A. 论 LL 不规则语法的解析。在 *MFCS* (1976), Springer Verlag, pp.

[14] PARR, T. J., AND QUONG, R. W. Adding semantic a n d syntactic predicates to *LL*(*k*)-*pred*-*LL*(*k*).在 *CC'94*.

[15] POPLAWSKI, D. A. 论 LL 规则语法.*JCSS 18*, 3 (1979), 218 - 227.

[16] TOMITA, M. *Efficient Parsing for Natural Language*.Kluwer Academic Publishers, 1986.

[17] WOODS, W. A. Transition Network grammars for natural language analysis.*CACM 13*, 10 (1970), 591-606.