LL(*)文法: ANTLR语法解析生成器的基础

摘要

尽管**解析表达式语法**PEG**(Parser Expression Grammar)和通用**LR**(Generalized LR** GLR**)分析算法**十分强大，但语法解析仍然是一个没能彻底解决的问题。比如在传统的LL和LR解析器解的析过程中添加非终结符可能会导致未定义的解析行为，并且会在错误处理、单步调试中引入一些实际问题，以及给嵌入语法动作带来副作用。本文将介绍$LL(*)$文法的解析策略和与其相关的语法分析算法，该算法可以从ANTLR语法中构建$LL(*)$文法的解析决策。在解析时，这些决策能够根据解析决策以及输入符号的复杂程度，优雅的从前瞻字符数量$k \geq 1$转换为任意大小字符数的前瞻。$LL(*)$文法的解析能力可以支持上下文有关文法，并且在一些特殊情况下能够超出GLR和PEG文法的表达范围。通过在静态解析阶段尽可能多地消除语法猜测，$LL(*)$在提供PEG的表达能力的同时，还保留了LL文法优良的错误处理能力以及不受限制的语法动作。ANTLR的广泛使用(每年下载次数超7万次)表明，这种能力在各种应用和场景中都相当受欢迎。

1. 导论

尽管语法解析一直被相当重视，并且有着一段很长的学术研究历史，但语法解析并不是一个已经完全解决的问题。由于手工编写语法解析器相当繁琐并且容易出错，研究人员花费了数十年的时间去研究如何将高级编程语言生成对应的高效语法解析器。尽管如此，解析器生成器在语法的表达适用能力和可用性上仍然存在着问题。

在解析理论最开始被提出的时候，机器资源是十分稀缺的，因此一个语法解析器是否高效成为了最重要的考虑因素。在当时，这种窘境迫使程序员去改造自己语法来符合$LARL(1)$或者$LL(1)$文法的语法解析器生成器。但是时代变迁，现代计算机的性能已经十分强大了，所以程序员的开发效率成为了更加重要的考虑因素。为了应对这种发展趋势，研究人员开发了功能更加强大，但成本更加高昂的非确定行解析策略，包括遵循“自下而上”的LR文法以及遵循“自上而下”的LL文法。

在LR文法中，GLR文法解析器的解析性能根据语法定义对经典LR的符合程度，其解析时间从$O(n)$到$O(n^3)$不等。GLR的本质是“分叉”出新的子解析器，然后从非确定的LR状态开始，解析所有可能出现的动作(action)，并且当无效的解析器被子解析器生成时，终止该子解析器。最终生成一个包含所有可能解释输入流的解析森林(parse forest)。Elkhound(猎犬)是一个非常高效的GLR实现，当语法是$LALR(1)$时，其解析速度媲美yacc。不过，对$LALR$解析理论不熟悉的程序员很容易得到非线性(线性这里指$O(n)$)的GLR解析器。

在“自上而下”的世界里，Ford引入了Packrat解析器以及其相关的解析器表达式语法($PEGs$, Parser Expression Grammars)。PEG是一个不允许使用左递归的语法。Packrat解析器是一类回溯式解析器，按照指定的顺序尝试去产生可替代的产生式。匹配当前输入位置的第一个生成式将被解析规则采用。相比于指数型解析器，由于Packrat会将部分结果进行缓存，所以Packrat是一个线性的解析器。Packrat解析器保证输入状态不会被同一个生成式解析多次。基于Rats-PEG的工具大力优化了记忆化事件以提高运行速度和减少内存的占用。

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），程序员们不需要再艰难地处理大量的冲突冲突信息。不过尽管存在这样的优势，GLR和PEG解析器也不能完全符合所有的需求场景。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默认接受存在二义性的语法，即可以用多种解释方式匹配同一输入，这就迫使程序员们不断地去地检语法是否存在二义性。而PEG没有语法冲突的概念，因为PEG总是按照“第一"匹配原则去解释输入，这可能导致意想不到或麻烦的行为，例如，PEG规则$A \rightarrow a|ab$（意思是"A要么匹配a，要么匹配ab"）的第二个解释：“匹配ab”将永远不会被用上。因为第一个符号a匹配的是第一个解释选项，所以输入ab将永远不会匹配第二个解释选项。在大型语法中，这种危险是潜在的，如果不经过彻底的调试，即使是经验非常丰富的开发者也会放走这些错误。

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，状态通常会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“的解析器则相对更容易理解，因为从LL语法元素到解析器的操作之间存在一一对应的映射关系。此外，递归下降的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动作，更加便于理解。然而，对于存在回溯的递归下降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是非常困难的，尽管这种功能对于商业开发人员如此重要。能否提供良好的语法错误提示取决于解析器的上下文。例如 ，当识别到一个无效的表达式时，如果要进行有效的恢复工作并给出准确的错误信息，解析器需要知道它正在解析数组索引还是赋值语句。在第一种情况下，解析器应向前跳过直到"]"标记来重新同步。在第二种情况下，解析器应该跳转到";"标记。自上而下的解析器存在一个规则调用堆栈，可以发出类似 "数组索引中的表达式无效 "的错误提示。另一方面，"自下而上"的解析器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解析器也会存在二义性的上下文，因为总是在进行预测。事实上，它们也无法从语法错误中恢复：因为在看到整个输入之前，它们都无法检测到错误。

最后，非确定性解析策略无法轻松地支持任意的嵌入式语法动作，而这些操作对于使用符号表、构建数据结构等非常有用。预测解析器也不能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。当然，在GLR解析器中，即使是计算规则返回值这种无副作用的操作也会很棘手。例如，由于解析器可以用多种解释方式匹配同一规则，它可能会执行多个相互竞争的动作（那么这种情况下，究竟是多次执行合并成一个执行还是每个都单独执行呢？）。GLR 和PEG工具解决这个问题的方

法是禁止执行动作、禁止执行任意动作，或者干脆依赖程序员来避免这些可能被预测执行动作而产生出的副作用。

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$LL(*)$）可以解决上述的缺陷。ANTLR 的输入是一个无上下文的文法，并且增加了语法(syntactic)和语义(semantic)谓词(predicates)以及嵌入式动作(embedded actions)。语法谓词允许任意的前看，而语义谓词则允许构造谓词点之前的状态来指导解析工作。语法谓词以语法片段(grammar segment)的形式给出，并且必须与即将到来的输入相匹配。语义谓词则以解析器编写语言的任意布尔值给出。动作(actions)通过解析器的编写语言来实现，并且可以访问当前的状态。与PEG一样，ANTLR也要求程序员避免使用左递归的语法规则。

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，而不是使用固定常量或用整个解析器通过回溯来解决前看符号的问题。分析器(analysis)会为语法中的每个非终结符构建一个确定性的有限状态自动机(DFA deterministic finite automata)，用以区分不同的产生式(productions)。如果分析器无法为某个非终结符找到合适的DFA，那么这个非终结符的匹配策略就会变成回溯的方式。因此，$LL(*)$ 分析程序可以从传统的固定$k \geq 1$的前看字符（lookahead）到升级到任意的前看字符（lookahead），最后根据解析决策的复杂程度，合理地切换到回溯式解析（backtracking）。即使在同一解析决策中，解析器也会根据输入序列动态地决定出解析策略，因为并不是所有的输入序列都意味着让一个解析决策可能需要任意地向前或向后扫描。在实践中，$LL(*)$分析程序平均只前看一到两个词法单元（token），偶尔需要回溯。所以$LL(*)$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，而没有其他解析器需要频繁预测(speculation)的缺点。尤其是，ANTLR接受除左递归以外的所有上下文无关文法。因此ANTLR与GLR、PEG解析一样，程序员不必为了适应解析其策略而扭曲(contort)自己的语法。而与GLR、PEG不同的是，ANTLR可以静态识别某些语法的二义性(grammar ambiguities)以及一些无效的产生式(dead productions)。ANTLR生成的是自上而下、递归下降，(大多数情况下)非预测性的语法解析器，而这些优势意味着它支持源代码级别的调试、生成高质量的错误提示，并且允许程序员嵌入任意的动作 。根据sorceforce.net和code.google.com提供的89个ANTLR语法的调查结果，保守计算，75%的 ANTLR语法添加了都嵌入式动作，说明此特性在ANTLR社区中是一个有用且受欢迎的功能。

ANTLR的广泛使用表明LL(*)符合编程人员的舒适区，并且针对各种语言都很有成效。(以下为机译)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 ANTLR works 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、 Apple 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，我们正式定义谓词语法(predicated grammar)和一种称为【谓词-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【谓词-LL正则语法】的解析决策。接下来，我们给出了一种从ANTLR 语法构建前看DFA的算法（第 5 节）。最后，我们对关于$LL(*)$效率和减少预测的主张予以支持（第 6 节）。

2. LL(*)简介

在本节中，我们构造了两个ANTLR语法片段来说明$LL(*)$算法，从而给出$LL(*)$一个直观的感受。考虑一个非终结符s，它使用了另一个非终结符$expr$(已省略)来匹配一个算术表达式。

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

非终结符s可以匹配一个标识符(ID)、一个ID后面跟一个'='号然后跟一个表达式(expr)；也可以跟零次或多次出现的'unsigned'字符，接着跟一个'int'，最后跟一个ID；或者跟零次或多次出现的'unsigned'字符，然后跟两个ID。ANTLR使用了类似yacc的语法，利用拓展的BNF范式(EBNF)操作符(也可以叫算子operators)(比如Kleene闭包(*))和用单引号括起来的token字面量(literal)。

当应用这个语法片段时，ANTLR的语法分析就会为语法规则s产生如图1所示的$LL(*)$前看的DFA。

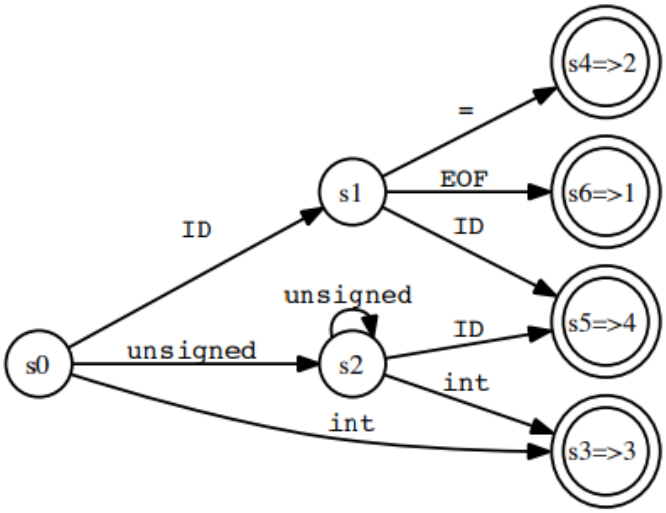


图1. 语法规则s的前看DFA，符号sn=>i表示“谓词(predict)(选择)第i个备选分支”

对于语法规则s的决策来说，ANTLR接受输入，然后运行这个DFA，它会根据当前不同的状态选选择备选分支，直到达到接受(accept)状态。

尽管我们可能需要多个(任意)前看输入才能将$s3$和$s4$进行区分，但是前看DFA会对每个输入序列(input sequence)使用最小的前看。当"int"从一个输入序列"int x"出现时，DFA会立即预测(predict)出$s3$这个备选分支($k = 1$，前看一个token)。当"T"(一个标识符(ID))从输入序列"T x"出现时，DFA需要$k = 2$(前看两个token)来区分$s1$、$s2$、$s4$。只有在出现'unsigned'符号的情况下，DFA才需要多次(任意)前看，寻找能区分备选分支$s3$和$s4$的符号('int'或'ID')。

语法规则s的前看是正则表达式形式的，所以我们可以通过DFA进行匹配。然而，对于递归的规则而言，我们能够发现它通常是上下文无关文法而不是正则表达式(使用递归而不是像正则表达式那样使用$*$、$+$、?来匹配零个、一个或多个)。在这种情况下，如果程序员通过添加语法谓词来实现这种功能，ANTLR会过渡到回溯式。为方便起见，我们使用选项"$backtrack = true$"来自动地将语法谓词插入到每个产生式中，这种我们称之为"PEG模式"，因为它模仿了PEG解析器的行为。不过，在回溯之前，ANTLR的分析算法会添加一些额外的状态来构建DFA，使其在许多输入情况下都能避免回溯。比如下面的语法规则$s2$，其两个备选分支都能以任意数量的'-'号开始（第二个备选分支expr可以通过递归多个带'-'号的expr来实现）。

```
options {
    backtrack = true    // auto-insert syntatic preds(predicates)
}

s2 : '-'* ID
   | expr ';'
   ;

expr : INT
     | '-' expr
     ;
```

图 2 则显示了 ANTLR 针对该输入构建的前看DFA。

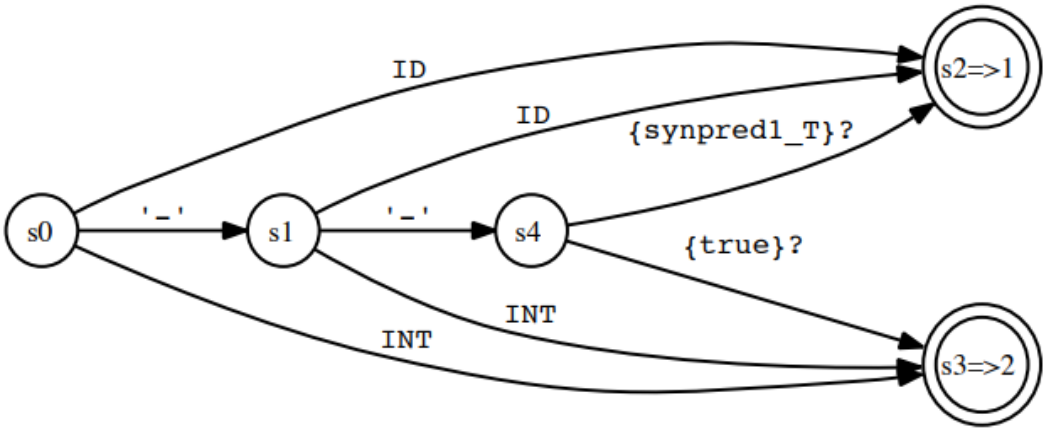


图2. 语法规则s2的解析决策DFA，使用了混合的k ≤ 3前看和回溯（译者：synpred1_T大概率指的是syntactic predicates times(这里是1) T(True)的缩写，大概是判断是否达到了递归展开数量的阈值常量m=1）

当输入'x'或'1'的时候，此DFA只需要根据当前的符号就能立即选择出合适的备选分支。当输入不定个数的'-'号时，此DFA会先匹配几个'-'号，然后再过渡到回溯式。在回溯之前，ANTLR展开递归规则的次数通过内部的一个常量m来控制，在这个例子中，我们设置这个常量m为1。尽管进行回溯的可能性很高，但在实践中，这个决策是不会回溯的，除非真的会有人输入'--'的序列，这样的前缀在表达式中是不太可能出现的。

3. 谓词($predicated$)文法

要精确描述$LL(*)$的解析，首先我们需要正式定义它们的谓词(predicated)文法，一个谓词文法$G = (N, T, P, S, \Pi, \mathcal{M})$，它有如下元素：

- N 是非终结符(规则名称)集合
- T 是终结符(词法单元)集合
- $S \epsilon N$ 表示S是起始符号，并且S属于非终结符集合
- Π 是无副作用的语义谓词集合
- $\mathcal{M}$ 是动作集合(或者说一组修改器(mutators，如在Java语言中，setter方法就是mutator))

谓词文法使用以下符号进行编写：

$A \in N$	非终结符号
$a \in T$	终结符号
$X \in (N \cup T)$	文法符号
$\alpha, \beta, \delta \in X^*$	文法符号序列
$u, x, y, w \in T^*$	终结符号序列
$w_r \in T^*$	剩余的输入终结符
ϵ	空串
$\pi \in \Pi$	和实现语言相关的谓词
$\mu \in \mathcal{M}$	和实现语言相关的动作
$\lambda \in (N \cup \Pi \cup \mathcal{M})$	归约标号
$\vec{\lambda} = \lambda_1..\lambda_2$	归约标号序列
产生式规则：	
$A \rightarrow \alpha_i$	A的i^{th}(第i个)上下文无关文法产生式
$A \rightarrow (A_i')\rightsquigarrow \alpha_i$	基于谓词判断后的语法A_i'的i^{th}(第i个)产生式
$A \rightarrow \{\pi_i\}?\alpha_i$	基于语义$(semantics)$谓词判断后的i^{th}(第i个)产生式
$A \rightarrow \{\mu_i\}$	具有修改器$(mutators)$的产生式

产生式通过编号来表示各自的优先级(precedence)，以此来消除文法规则的二义性。第一种产生式(译者注：产生式规则里的第一条)用来表示标准的上下文无关文法的产生式；第二种产生式表示基于**语法谓词(syntactic predicates)**生成的产生式：只有在当前输入也符合A_i'所描述的文法时，文法A才会拓展为α_i。语法谓词可以实现任意的、可以由程序员指定的、上下文无关的前看。第三种产生式表示基于**语义谓词(semantic predicates)**生成的产生式：只有谓词π_i和当前所构造的状态匹配时，文法A才会拓展为α_i。最后的一种产生式表示一个动作：根据修改器(mutator)μ_i，将对应规则的状态进行更新。

谓词文法可以用下面的**最左推导(leftmost derivation)**规则来定义：

$$Prod\ \frac{A \to \alpha}{(\mathbb{S}, uA\delta) \Rightarrow (\mathbb{S}, u\alpha\delta)} \tag{1}$$

$$Action\ \frac{A \to \{\mu\}}{(\mathbb{S}, uA\delta) \overset{\mu}{\Rightarrow} (\mu(\mathbb{S}), u\delta)} \tag{2}$$

$$Sem\ \frac{\begin{array}{c}\pi_i(\mathbb{S}) \\ A \to \{\pi_i\}?\alpha_i\end{array}}{(\mathbb{S}, uA\delta) \overset{\pi_i}{\Longrightarrow} (\mathbb{S}, u\alpha_i\delta)} \tag{3}$$

$$Syn\ \frac{\begin{array}{c}(\mathbb{S}, A_i') \Rightarrow^* (\mathbb{S}', w) \\ w \preceq w_r \\ A \to (A_i') \rightsquigarrow \alpha_i\end{array}}{(\mathbb{S}, uA\delta) \overset{A_i'}{\Longrightarrow} (\mathbb{S}, u\alpha_i\delta)} \tag{4}$$

$$Closure\ \frac{(\mathbb{S}, \alpha) \overset{\lambda}{\Longrightarrow} (\mathbb{S}, \alpha'), (\mathbb{S}, \alpha') \overset{\vec{\lambda}}{\Longrightarrow} {}^*(\mathbb{S}, \beta)}{(\mathbb{S}, \alpha) \overset{\lambda\vec{\lambda}}{\Longrightarrow} {}^*(\mathbb{S}, \beta)} \tag{5}$$

公式1. 谓词文法的最左推导公式

规则引用了状态$\mathbb{S}$来支持语义谓词(semantic predicates)和修改器(mutators)，它抽象地表示在解析过程中产生的各种用户状态(user state)，同样地，引入w_r来支持语法谓词(syntactic predicates)，用来表示剩余的待匹配的输入。判断式：$(\mathbb{S}, \alpha) \overset{\lambda}{\Rightarrow} (\mathbb{S}', \beta)$（译者注：这里应该指的的是$Closure$公式中分子的这个$(\mathbb{S}, \alpha) \overset{\lambda}{\Longrightarrow} (\mathbb{S}, \alpha')$公式）可以理解为：在当前的机器状态$\mathbb{S}$，输入文法序列$\alpha$后，将在下一步将$\mathbb{S}$归约成$\mathbb{S}'$和新的文法序列$\beta$，并同时发射(emit)一个追踪(trace)$\lambda$。判断式：$(\mathbb{S}, \alpha) \overset{\vec{\lambda}}{\Longrightarrow} {}^*(\mathbb{S}', \beta)$（译者注：这里同样应该指的的是$Closure$公式中分子的这个$(\mathbb{S}, \alpha') \overset{\vec{\lambda}}{\Longrightarrow} {}^*(\mathbb{S}, \beta)$公式）表示：将单步归约规则(one-step reduction rule)中重复的归约动作进行累积。如果λ对接下来的分析并不重要，我们就会省略(omit)它，这些归约规则指定了其最左推导。如果一个产生式带有**语义(semantic)**谓词π_i，那么只有在当前状态$\mathbb{S}$的谓词π_i为真(true)的；时候，该产生式才会生成；而如果一个产生式带有**语法(syntactic)**谓词A_i'，那么只有在当前状态下从A_i'派生出的字符串是剩余待输入的**前缀**时，该产生式才会生成，我们将其写作$w \preceq w_r$。动作将会在尝试解析A_i'的过程中以预测(speculatively)的方式执行，并且将会根据是否匹配A_i'来决定是否撤销这一过程中被预测执行的动作。最后，一个动作产生式将会指定其修改器μ_i来更新当前的状态。

形式上，文法序列α生成的语言是$L(\mathbb{S}, \alpha) = \{w | (\mathbb{S}, \alpha) \Rightarrow {}^*(\mathbb{S}', w)\}$，文法$G$生成的语言是$L(G) = \{w | (\epsilon, S) \Rightarrow {}^*(\mathbb{S}, w)\}$。从理论上来说，$L(G)$类语言是可以穷举的递归，因为每个修改器都可以是一个图灵机。而在实践中，文法的编写者通常不会用到这样的特性，我们考虑这类语言可以是上下文相关的：由于谓词(predicates)可以同时检查上文也可以检查下文，所以这类语言是上下文相关，而不是上下文无关的。

实际上这样的文法形式存在诸多语法限制，但并未在ANTLR的实际输入中展现出来，比如：谓词必须强制位于产生式规则的最左边、修改器(mutators)也必须侵入到本身的规则中。但是我们也能在不损失ANTLR这种通用性的前提下实现这些语法限制，因为任意的通用文法都可以被转换成更严格的形式。

解析背后的一个关键概念在于：在解析过程中的某一特定时刻，输入的语言能够被一个产生式所匹配。

定义1. $\mathcal{C}(\alpha) = \{w | (\epsilon, S) \Rightarrow {}^*(\mathbb{S}, u\alpha\delta) \Rightarrow {}^*(\mathbb{S}', uw)\}$为产生式$\alpha$的延续语言$(continuation\ language)$。

> 译者注：“延续语言”这个翻译并不准确，至于continuation在计算机科学中的含义，可以参考CPS编程风格中continuation所拥有的释义，或者google一下Continuation在计算机科学中的含义。

最后，文法的位置$\alpha \cdot \beta$表示"在生成或者解析的过程中其位于α之后β之前"。

3.1 解决二义性

文法的二义性指的是同一个字符串能够被多种方式匹配，上诉给出的谓词文法的5个推导公式并没有排除这种二义性。但是，对于一个实用的解析器来说，我们希望每种输入都只会对应一个唯一的解析匹配。为此，ANTLR 使用文法中的产生式顺序来解决二义性问 题，当冲突发生时，将以优先级最高(顺序最小)的产生式进行匹配。程序员收到的指示是：在所有的、可能存在二义性的输入序列的情况下，让语义谓词(semantic predicates)之间互斥，以此来消除产生式的二义性。然而，由于谓词是由图灵完备的语言来编写的，所以这一条件是无法得到执行的。如果程序员们无法满足这一条件限制，ANTLR 就会使用产生式顺序来解决歧义问题。这种策略对于简洁地表示产生式优先级(precedence)非常有用，并且与PEG中的做法是一致的。

3.2 谓词的($predicated$)LL-正则(LL-regular)文法

最后还有一个概念有助于理解$LL(*)$分析框架，即谓词的($predicated$)LL-正则(LL-regular)文法这一概念。在以往的工作中，Jarzabek和Krawczyk以及Nijholt将LL-正则语法定义为非左递归、无二义性CFG文法的一个特定子集。而在这项工作中，我们将LL-正则文法的概念拓展为谓词的LL-正则文法，并为其构造高效的$LL(*)$分析器。我们要求输入的语法是非左递归的，并且使用规则顺序来确保文法的无二义性。LL-正则文法与 $LL(k)$文法的不同之处在于，对于任何给定的非终结符，解析器都可以利用剩下的整个输入来区分可供选择的产生式，而不仅仅是k个符号。LL-正则文法要求非终结符A的所有终结符序列位于正则分区上，而每一个分区都正好对应非终结符A的一个可能的产生式。一个LL-正则解析器将决定剩下的输入属于哪一个正则集合并选择对应的产生式。

定义2. 设$R = (R_1, R_2, \ldots, R_n)$是$T^*$分为$n$个不为空且不相交的集合$R_i$。如果每个集合$R_i$是正则的，那么$R$也是一个正则的集合。如果$x, y \in R_i$，我们写作$x \equiv y (mod\ R)$

定义3. 如果对于每个非终结符A的任意两个可选的产生式α_i和α_j，存在正则分区R使得

$$(\epsilon, S) \Rightarrow^* (\mathbb{S}, w_i A \delta_i) \Rightarrow (\mathbb{S}, w_i \alpha_i \delta_i) \Rightarrow^* (\mathbb{S}_i, w_i x) \quad (1)$$

$$(\epsilon, S) \Rightarrow^* (\mathbb{S}, w_j A \delta_j) \Rightarrow (\mathbb{S}, w_j \alpha_j \delta_j) \Rightarrow^* (\mathbb{S}_j, w_j y) \quad (2)$$

$$x \equiv (mod\ R) \quad (3)$$

则我们称G是谓词的$(predicated)$$LL$-正则$(LL\text{-regular})$文法。

并且$\alpha_i = \alpha_j$和$\mathbb{S}_i = \mathbb{S}_j$恒成立。

4. $LL(*)$解析器

虽然现有的由Nijholt和Poplawski提出的LL-正则文法分析器是线性的，但往往不切实际，因为它们无法解析无穷的数据流，比如应用在套接字(socket)协议和交互式解释器(interactive interpreters)上。在上述的两种数据传输过程中，解析器就必须从右到左来读取输入数据。（译者：原文是："In the first of two passes, these parsers must read the input from right to left."。这一段看得有点懵，结合上下文也很难翻译，我估计这里作者指的是在举的那两种不适用的例子中，解析器要等到输入完全停止后才能开始解析）

取而代之的是，我们提出了一种更简单的、从左到右、一次遍历的解析策略，称之为$LL(*)$，它将前看$DFA(lookaheadDFA)$移植到了LL解析器上。前看DFA和其指定的的非终结符相关联的正则分区R进行匹配(math)，并且每个R_i都会有一个接受(accept)状态。在某个决策点上，如果R_i和剩余的输入相匹配，$LL(*)$解析器就会生成产生式i。因此，$LL(*)$解析器的复杂度为$O(n^2)$，但在实际应用中，解析器通常只会校验一到两个词法单元(token)。与此前的解析策略一样，每个LL-正则文法都会有一个$LL(*)$解析器。与之前的工作不同，$LL(*)$解析器可以将输入当做谓词的正则文法(predicated LL-regular grammar)来处理；通过在前看DFA中插入一些与谓词对应的特殊边(edges)来处理谓词。

定义4. 前看DFA是一种添加了谓词和一些接受状态的DFA，并且这些接受状态分别对应一个由其对应谓词判断通过后的产生式标号。正式地，给定谓词文法 $G = (N, T, P, S, \Pi, \mathcal{M})$, 则前看$DFA$为：

$$DFA\ M = (\mathbb{S}, Q, \Sigma, \Delta, D_0, F)$$

其中：

- $\mathbb{S}$是继承自周围解析器的一个系统状态
- Q是状态集合
- $\Sigma = T \cup \Pi$是边的的符号表
- Δ是映射(mapping)$Q \times \Sigma \to Q$的转换(transition)函数
- $D_0 \in Q$是起始状态
- $F = \{f_1, f_2, \ldots, f_n\}$是最终状态集合，并且每个正则分区$R_i$(产生式$i$)都会有一个$f_i \in Q$

我们给出Δ中转换函数的形式，例如状态p通过输入一个符号$a \in \Sigma$转换到状态q，则写作$p \xrightarrow{a} q$。而基于谓词的状态转换写作$p \xrightarrow{\pi} f_i$，即转换后的状态必须是最终状态。但是从状态p能够进行转换的状态肯定不止一种，在某个时刻，DFA的构造(configuration)c写作$(\mathbb{S}, p, w_r)$，其中$\mathbb{S}$是系统状态，p是当前状态；DFA的初始构造写作$(\mathbb{S}, D_0, w_r)$。使用$c \mapsto c'$表示DFA在使用以下公式中的规则从构造c变为c'：

$$\frac{p \xrightarrow{a} q}{(\mathbb{S}, p, aw) \mapsto (\mathbb{S}, q, w)} \quad (1)$$

$$\frac{\pi_i(\mathbb{S}) \quad p \xrightarrow{\pi_i} f_i}{(\mathbb{S}, p, w) \overset{\pi_i}{\mapsto} (\mathbb{S}, f_i, w)} \quad (2)$$

$$\frac{(\mathbb{S}, f_i, w)}{\text{接受状态，谓词产生式}i(Accept,\ predict\ production\ i)} \quad (3)$$

公式2. 前看DFA的构造转换规则

和谓词文法一样，这些规则并不禁止由谓词转换而产生的二义性DFA路径，在实践中，ANTLR会对每条转换边进行尝试和检测，以此来解决二义性问题。

为了提高匹配效率，相比于延续语言(continuation language)（译者注：定义1），前看DFA所匹配的是前看集合$(lookahead\ sets)$（译者注：应该就是编译原理中的first集）。假设一个文法$G = (\{ac^*\}, \{bd^*\})$，那么它的前看集合就是$(\{a\}, \{b\})$。

定义5. 给定一个正则分区R，并且R能够产生n个不同的产生式，那么产生式i的前看集合为R_i中能够唯一区分产生式i的最小前缀集$(minimal - prefix\ set)$：

$$LA_i = \{w | ww' \in R_i, w \notin LA_j, i \neq j且w的严格前缀都不具有和R_j相同的属性\}$$

4.1 语法谓词擦除

为避免单独给语法谓词建立识别机制，通过预测解析的方式，我们可以将语法谓词简化为语义谓词。例如，为擦除语法谓词$(A'_i) \rightsquigarrow$，我们使用语义谓词$\{synpred(A'_i)\}?$进行替换。如果A'_i和当前输入相匹配，函数$synpred$将会返回$true$，否则返回$false$。如果要支持PEG中的"非谓词(not predicates)"判断，我们可以像Ford建议的那样，对$synpred$函数的返回值取反即可。

4.2 谓词文法中的任意动作

标准的谓词文法会在预测的过程中分叉(fork)出新的状态系统$\mathbb{S}$。但在实践中，出现重复的系统状态是不符合预期的。因此，ANTLR会在预测的过程中会禁用掉修改器(mutators)，防止可能因预测从而执行错误的动作，以至产生出“灾难性”的bug。然而，一些语义谓词依赖于修改器所产生的变更，例如解析C所需要的符号表操作。我们可以尽可能地避免预测的发生，从而减轻这一问题，但这就会为语义谓词的操作留下隐患。为了解决上述的问题，ANLTR支持一种特殊的动作，即使在预测的过程中也能被执行，写法是用双括号$\{\{\dots\}\}$括起来。不过，ANTLR要求程序员确保这些动作是无副作用的或者是可以撤销的。幸运的是，符号表操作以及最常见的$\{\{\dots\}\}$操作都是可以被自动撤销的。例如：一个代码块(code_block)的文法规则通常会压入(push)一个符号作用域，然后在代码块退出时将这个作用域弹出来(pop)。”弹出(pop)“这一动作有效地撤销了代码块期间产生的副作用。

5. $LL(*)$文法分析

对于$LL(*)$而言，分析该文法意味着为每一个解析决策找到一个前看DFA，比如：文法中带有多个产生式的非终结符，其每个产生式都会对应一个前看DFA。在我们的讨论范围内，我们用A来表示议题中用到的非终结符，用$\alpha_i (i \in 1..n)$表示对应的产生式。我们的目的是为每个A的每个产生式找到其正则分区R，并以DFA的形式来表示，并且能够区分出不同的产生式。为达到目的，A必须是LL-正则文法：分区块R_i必须包含产生式α_i的延续语言$\mathcal{C}(\alpha_i)$的每个句子，而且R_i之前还必须互不相交。DFA会将剩下的输入挨个和R_i的谓词文法进行匹配，测试其相关性。当然，为了效率，DFA会匹配前看集合而不是分区块。

有一个很重要的点必须指出：我们是在用DFA进行谓词决策，找到解析器应当展开成哪个产生式，而不是用DFA进行文法解析。延续语言$\mathcal{C}(\alpha_i)$通常是上下文无关的，而不是正则表达式，但是经验表明，通常有一种近似正则表达式的方式可以区分不同的产生式α_i。比如，有这么一个匹配有效中括号的文法规则$A \rightarrow [A] | \text{id}$，即上下文无关文法$\{[^n \text{id}]^n\}$。我们可以用这么一个满足$LL$-正则文法条件的：$R = \{\{[^* \text{id}]^*\}, \{\text{id}\}\}$正则表达式来近似的表示$\mathcal{C}(\alpha_i)$。事实上，第一个输入符号就已经足够去判断不同的产生式了：$LA = \{\{[\}, \{\text{id}\}\}$，文法决策此时为$LL(1)$。

然而，并不是所有的文法都是LL-正则的，所以我们的这个算法在为A找到正则分区时可能会失败。并且更糟糕的是，Poplawski指出：要判断一个文法是否是LL-正则的是不可能的。所以我们必须使用一些引导或者启发式的方式来避免死循环，或者有时干脆强制终止算法来为A找到其正则分区R，即使A满足LL-正则的属性。在上述的情况下，我们会采用第5.3节和第5.4节中讨论的其他策略，而不是去创建DFA。

$LL(*)$分析算法首先会将输入文法转化为等效的增强转换网络$(argumented\ transition\ network)$(ATN)。然后，在解析过程中通过模拟ATN的动作来计算前看DFA，这一过程模仿了著名的将NFA转换为DFA的子集构造$(subset\ construction)$算法

5.1 增强转换网络(ATN)

给定谓词文法$G = (N, T, P, S, \Pi, \mathcal{M})$，其对应的ATN $M_G = (Q, \Sigma, \Delta, E, F)$具有以下元素：

- Q是状态集合
- Σ是应用在转换边的字母$N \cup T \cup \Pi \cup \mathcal{M}$
- Δ是映射$Q \times (\Sigma \cup \epsilon) \rightarrow Q$的转换关系
- $E = \{p_A | A \in N\}$是(状态机)子机的入口状态集合
- $F = \{p'_A | A \in N\}$是(状态机)子机的最终状态集合

接下来我们将马上介绍如何计算Q和Δ。

ATN类似于记录编程语言的语法图，每个非终结符都有一个ATN子机。例如，图3给出了一个简单语法的ATN。

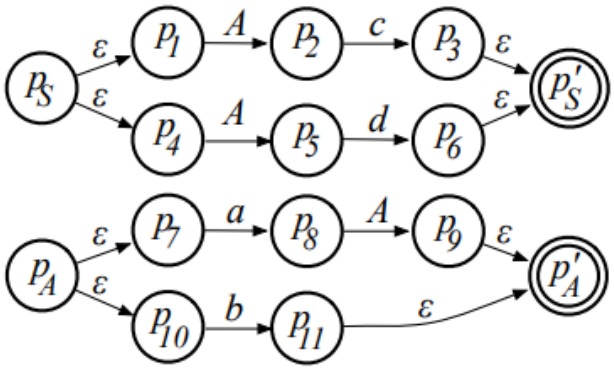


图3. $P = \{S \to Ac|Ad, A \to aA|b\}$时，文法$G$的ATN示意图

非终结符的转换边$p \xrightarrow{A} p'$就像是函数调用。它们将返回状态p'压入状态栈，并将ATN的控制权转移给A的子机，从而在A的子机达到终止态后能够继续从p'继续执行。

要从文法中得到ATN，我们要为每个非终结符创建一个子机，如图4所示。以$A \to \alpha_i$为例，我们为α_i构建了中间状态$p_{A,i}$和最终状态p'_A，在最开始，初始状态为p_A，则p_A的目标是不断通过Δ中的边转换到状态$p_{A,i}$，最后转换为p'_A。ATN匹配的文法和原始文法规则匹配的文法是相同的。

Input Grammar Element	Resulting ATN Transitions
$A \to \alpha_i$	$p_A \xrightarrow{\epsilon} p_{A,i} \xrightarrow{\epsilon} \boxed{\alpha_i} \xrightarrow{\epsilon} p'_A$
$A \to \{\pi_i\}?\ \alpha_i$	$p_A \xrightarrow{\epsilon} p_{A,i} \xrightarrow{\pi_i} \boxed{\alpha_i} \xrightarrow{\epsilon} p'_A$
$A \to \{\mu_i\}$	$p_A \xrightarrow{\epsilon} p_{A,i} \xrightarrow{\mu_i} p'_A$
$A \to \epsilon$	$p_A \xrightarrow{\epsilon} p_{A,i} \xrightarrow{\epsilon} p'_A$
$\boxed{\alpha_i} = X_1 X_2 \ldots X_m$ for $X_j \in N \cup T,\ j = 1..m$	$p_0 \xrightarrow{X_1} p_1 \xrightarrow{X_2} \ldots \xrightarrow{X_m} p_m$

图4. 谓词文法到ATN的转换

语法分析和程序的流程分析一样，程序的流程分析静态地遍历一个类似ATN的图，从图的根节点出发找到所有可到达的节点。不过就程序的流程分析而言，在构造上唯一的不同就是就是图的节点和用于找到该节点的调用栈。根据分析的类型不同，语法分析可能还会追踪一些语义上下文，比如来自根节点的调用参数。

类似的，语法分析也会静态地遍历由产生式α_i构建的ATN图，并从根节点出发，通过可到达的节点遍历所有的路径，就比如上述的路径p_A, i。用沿着从p_A, i出发的路径不断收聚非终结符方式代表一个前看序列。分析工作会持续进行，直到每个前看序列能够唯一区分出不同的产生式为止。分析工作也同样需要追踪一些来自α_i上文的语义谓词π_i，并在需要解决二义性的时候用到它们。因此，一个ATN构造(ATN configuration)是一个元组(p, i, γ, π)：由ATN的状态p、谓词产生式i、ATN调用栈γ以及可选地谓词π构成。在后面，我们会使用$c.p, c.i, c.\gamma$以及$c.\pi$来分别表示一个ATN构造c的状态、产生式、调用栈以及谓词。分析过程将会忽略机器存储空间$\mathbb{S}$，因为这块空间在分析的过程中是无法获知大小的。

5.2 修订版子集构造算法

出于语法分析的目的，我们修改了子集构造算法，从而能够处理ATN的构造而非NFA。每个DFA的状态D表示这样一个可能的构造集合：从状态p_A, i开始，匹配一个剩余输入的前缀后，ATN可能表现出的构造。修订版子集构造算法的关键修改包含以下：

- 闭包$(closure)$的操作，模拟了ATN非终结符调用的压入(push)和弹出(pop)。
- 如果在当前最新的状态下，所有的构造通过谓词都判断出同一个产生式，那么该状态将不会被添加到待解析的工作队列中，因为这已经足够判断出产生式了，也没有必要再进行前看了。
- 为解决文法二义性的问题，如果存在合适的谓词，该算法会为最终状态添加其谓词转换。

该算法的整体结构实际上和子集构造算法类似。首先创建DFA的起始状态D_0，然后将其添加到待解析的工作队列中。在这样的工作队列为空之前，算法都会将通过移动$move$和闭包$(closure)$函数计算出来的新状态添加到该工作队列中，这实际上就模拟了ATN的转换。我们假设一个和我们的输入文法G对应的ATN $M_G = (Q_M, N \cup T \cup \Pi \cup \mathcal{M}, \Delta_M, E_M, F_M)$以及我们要分析的非终结符$A$都在算法的操作范围内，如算法1所示：

算法1. 从ATN构建DFA

Alg. 1 : $createDFA(\text{ATN State } p_A)$ **returns** DFA

$work := []$; $\Delta := \{\}$; $D_0 = \{\}$;
//译者注：$numAlts(A)$即$number\ of\ alternative\ A$, 表示和备选的产生式数量有关
$F := \{f_i \mid f_i := \text{new DFA state}, 1...numAlts(A)\}$;
$Q := F$;
foreach $p_A \xrightarrow{\epsilon} p_{A,i} \in \Delta_M$ **do**
  if $p_A \xrightarrow{\epsilon} p_{A,i} \xrightarrow{\pi_i} p$ **then** $\pi := \pi_i$ **else** $\pi := -$;
  $D_0 \mathrel{+}= closure(D_0, (p_{A,i}, i, [], \pi))$;
end
$work \mathrel{+}= D_0$; $Q \mathrel{+}= D_0$;
$DFA := DFA(-, Q, T \cup \Pi, \Delta, D_0, F)$;
foreach $D \in work$ **do**
  foreach $a \in T$ **do**
    $mv := move(D, a)$;
    $D' := \bigcup_{c \in mv} closure(D, c)$;
    if $D' \notin Q$ **then**
      $resolve(D')$;
      switch $findPredictedAlt(D')$ **do**
        case $None : work \mathrel{+}= D'$;
        case $Just\ j : f_j := D'$;
      endsw
      $Q \mathrel{+}= D'$;
    end
    $\Delta \mathrel{+}= D \xrightarrow{a} D'$;
  end
  foreach $c \in D\ \ such\ that\ wasResolved(c)$ **do**
    $\Delta \mathrel{+}= D \xrightarrow{c.\pi} f_{c.i}$;
  end
  $work$ -= D;
end
return DFA;

函数$createDFA$（如算法1所示）是算法的入口：调用$createDFA(p_A)$可以为非终结符A创建一个前看DFA。为创建起始状态D_0，算法将会为A的每个含有谓词的产生式$A \to \pi_i\alpha_i$添加对应的构造$(p_{A,i}, i, [], \pi_i)$，并且也会为A的每个不含谓词的产生式$A \to \alpha_i$添加对应的构造$(p_{A,i}, i, [], -)$；符号"$-$"表示没有谓词。$createDFA$的核心是一系列的移动$-$闭包$(move-closure)$操作：通过每个输入的终结符$a \in T$，找到其直接可达的ATN状态集，然后创建出新的DFA状态：

$$\text{移动操作：}\ move(D, a) = \{(q, i, \gamma, \pi) | p \xrightarrow{a} q, (p, i, \gamma, \pi) \in D\}$$

接着，为这些构造添加闭包。一旦算法确定了一个新的状态D'，并且这个状态不在状态集合Q内，它就会调用$resolve$函数来进行检验并解决二义性。如果D'的所有构造都预测了相同的产生式j，那么D'就会被标记为f_j，表示备产生式j的接受状态，而D'则不会被添加到工作队列中：一旦算法能够唯一地确定哪个产生式该被预测时，就没有必要再去检测更多的输入了。该算法就是通过这种优化，来构建和最小前看集合LA_j匹配的DFA，而非剩余的整个输入。接下来，算法将会为终结符a添加一条从D到D'的边。最后，对于每个带有谓词并解决了二义性的构造$c \in D$，$createDFA$就会为产生式$c.i$添加一条从D通过谓词$c.\pi$到最终状态$f_{c.i}$的转换关系。测试函数$waResolved$将会检查构造c是否在$resolve$步骤中被标记为已被谓词解决。

闭包(Closure)。$LL(*)$的闭包操作如算法2所示，由于ATN栈的存在，它比NFA子集构造算法中的闭包$(Closure)$更为复杂。不过在直觉上两者又是几乎相同的。

算法2. 闭包算法

Alg. 2 : $closure(\text{DFA State } D, c = (p, i, \gamma, \pi))$
returns set $closure$

if $c \in D.busy$ **then return** $\{\}$; **else** $D.busy \mathrel{+}= c$;
$closure := \{c\}$;
if $p = p'_A$ (i.e., p *is stop state*) **then**
  if $\gamma = p'\gamma'$ **then** $closure \mathrel{+}= closure(D, (p', i, \gamma', \pi))$;
  else $closure \mathrel{+}= \bigcup_{\forall\, p_2 : p_1 \xrightarrow{A} p_2 \in \Delta_M} closure(D, (p_2, i, [], \pi))$
end
foreach *transition* t *emanating from ATN state* p **do**
  switch t **do**
    case $p \xrightarrow{A} p'$:
      $depth :=$ number of occurrences of A in γ;
      if $depth = 1$ **then**
        $D.recursiveAlts \mathrel{+}= i$;
        if $|D.recursiveAts| > 1$ **then**
          throw $LikelyNonLLRegularException$;
      end
      if $depth \geq m$, (i.e, *the max recursion depth*) **then**
        mark D to have recursion overflow;
        return $closure$;
      end
      $closure \mathrel{+}= closure(D, (p_A, i, p'\gamma, \pi))$;
    case $p \xrightarrow{\pi'} q$, $p \xrightarrow{\mu} q$, *or* $p \xrightarrow{\epsilon} q$ *transition* :
      $closure \mathrel{+}= closure(D, (q, i, \gamma, \pi))$;
  endsw
end
return $closure$;

当用构造c进行调用时，闭包算法会遍历所有的非终结符的ATN边（比如：谓词predicates、非终结符nonterminals以及修改器(mutators），找到所有能从构造c转换的ATN状态。

调用$closure(D, c)$时，会将c所属的DFA状态D作为一个额外参数。在函数开始时，会首先将c添加到D的繁忙$(busy)$队列中，避免冗余计算或者是死循环；在模拟ATN的非终结符转换：$p \xrightarrow{A} p'$时，闭包算法会复制一份构造c，并将返回状态p'压入到状态栈中，在子机为终止状态p'_A时，闭包算法同样会复制一份构造c，并将返回状态p'从状态栈中弹出。如果闭包算法到达状态p_A时状态栈是空的，我们便无法静态地知道哪条产生式规则调用了闭包A（不过这种情况只会发生在$p_{A,i}$到p'_A的转换不存在终结符的边时）。在这种情况下，我们只能假设输入语法中的任意产生式$p_1 \xrightarrow{A} p_2$都可能被调用了，然后闭包算法必须追踪所有的类似p_2这样的状态。

如果闭包算法检测到递归的非终结符调用（子机直接或间接地调用本身）在不止一个产生式中发生，那么闭包算法将会终止A的DFA构建并抛出异常。在第5.4节将会描述应对此场景的回退策略。如果闭包算法检测到递归的深度大于内部常量m，闭包算法会将状态参数D标记为溢出$(overflow)$，不过在这种场景下，A的DFA构建仍然继续，但是闭包算法将不再追踪从c衍生出的路径以及其状态栈。在5.3节，我们将会对此场景进行详细的讨论。

DFA状态等价(DFA State Equivalence)。$LL(*)$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D、D'的构造集相同，那么这两个DFA状态等价：$D \equiv D'$；
如果两个DFA构造c、c'中的p, i以及π分别相同以及它们的状态栈也相同，那么这两个DFA构造等价：$c \equiv c'$；
如果两个状态栈γ、γ'相同、或者其中一个为空、或者其中一个是另一个的后缀，那么这两个状态栈等价：$\gamma \equiv \gamma'$

状态栈等价定义反映了闭包算法在遇到子机终止态时ATN的上下文(context)信息。由于分析算法会为所有可能的前看序列查找ATN，故一个空栈就像一个通配符一样。任意转换$p_1 \xrightarrow{A} p_2$都会包含A，所以分析算法必须为类似p_2这样的状态进行闭包操作。因此，一个闭包集合可以包含两个构造c和c'，但是$c.\gamma = \epsilon$并且$c'.\gamma \neq \epsilon$。不过这种情况只会在闭包算法追踪引用了某个非终结符之后的所有状态时，重入了该非终结符的某个子机，并且到达了终止态，并且此时是空状态栈。举个例子，考虑一个S在语法$S \rightarrow a|\epsilon, A \rightarrow SS$下的DFA。起始状态的构造会计算$S \rightarrow \cdot a$和$S \rightarrow \cdot\epsilon$的闭包算法，然后计算$S$的终止态$S \rightarrow \epsilon\cdot$。此时状态栈是空的，所以闭包算法将会追踪引用了$S$之后的状态，比如$A \rightarrow S \cdot S$。最后，闭包算法重入了$S$，并且此时是空状态栈。

当$\gamma_2 \neq \epsilon$时，如果$\gamma_1 \equiv \gamma_1\gamma_2$将会退化到之前$\gamma_1 = \epsilon$的情况。那么给定$D$中的两个构造$(p, _, \gamma_1, _)$和$(p, _, \gamma_1\gamma_2, _)$，按照最近子机状态($\gamma_1$)调用的方式去处理相同的输入序列时，闭包算法都会到达状态p。而关于状态栈，一旦状态γ_1被弹出，闭包算法就会包含$(p, _, [], _)$和$(p, _, \gamma_2, _)$两个构造。

解析(Resolve)。静态分析的好处之一是，它有时可以检测到具有二义性的非终结符，并向使用者发出警告。当闭包操作结束后，解析$(resolve)$函数（如算法3所示）将会在参数状态D中查找具有冲突的构造(conflicting configuration)，而这样的构造表明ATN在同一输入下能够匹配多个产生式。

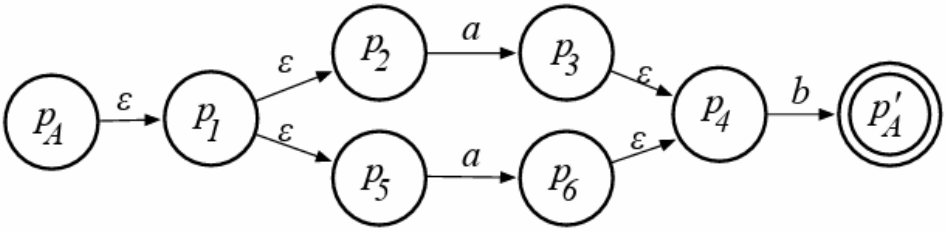
算法3. 二义性解析算法

Alg. 3 : $resolve(\text{DFA State } D)$

$conflicts :=$ the conflict set of D;
if $|conflicts| = 0$ *and not* $overflowed(D)$ **then return**;
if $resolveWithPreds(D, conflicts)$ **then return**;
resolve by removing all c from D such that $c.i \in conflicts$ and $c.i \neq min(conflicts)$;
if $overflowed(D)$ **then** report recursion overflow;
else report grammar ambiguity;

定义7. 如果DFA状态D具有构造$c = (p, i, \gamma_i, \pi_i)$以及构造$c' = (p, j, \gamma_j, \pi_j)$，其中，$i \neq j$并且$\gamma_i \equiv \gamma_j$，
那么D是一个二义性的DFA状态，并且c和c'是具有冲突的构造。
冲突构造的备选产生式集合也同样是冲突构造所属状态D的冲突集合。

举个例子，在规则$A \rightarrow (a|a)b$中的子规则$(a|a)$的ATN合并在一起，因此在同样的(空)状态栈下，分析算法将会到达同样的状态：



考虑状态D_0的构造：$D_0 = \{(p_2, 1), (p_5, 2)\}$，其中$(p_2, 1)$是$(p_2, 1, [], _)$更清晰的简便写法。同理，另一个状态$D_1 = \{(p_3, 1), (p_4, 1), (p_6, 2), (p_4, 2)\}$能够通过符号$a$到达，但是此时具有冲突构造$(p_4, 1)$和$(p_4, 2)$。由于$ab$同时是这两个备选产生式的延续性语言，所以此时再使用前分析看也无济于事了。

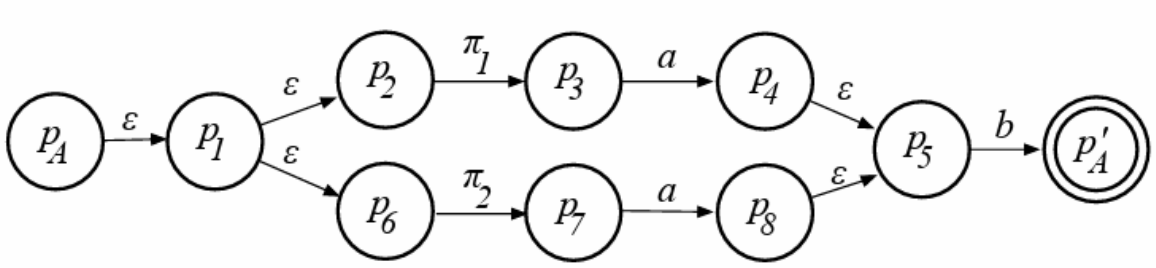
如果解析$(resolve)$函数检测到二义性，它会调用$resolveWithPreds$函数（如算法4所示），尝试从冲突构造中找到谓词来解决二义性问题。

算法4. 二义性能否进行谓词解析的判断算法

Alg. 4 : $resolveWithPreds(\text{DFA State } D, \text{ set } conflicts)$ **returns boolean**

$pconfigs :=$ [];　　//备选产生式i的谓词构造
foreach $i \in conflicts$ **do**
　$pconfigs[i] :=$ pick any representative $(\_, i, \_, \pi) \in D$;
end
if $|pconfigs| < |conflicts|$ **then return false**;
foreach $c \in pconfigs$ **do** mark c as $wasResolved$;
return true;

举个例子，前一个语法的谓词版本$A \rightarrow (\{\pi_1\}?a|\{\pi_2?\}a)b$将会产生如下的ATN构造：



D_0的决策状态p_1从构造$\{(p_2, 1, [], \pi_1), (p_6, 2, [], \pi_2)\}$开始，转换到我们在闭包算法中产生的构造$\{(p_3, 1, [], \pi_1), (p_7, 2, [], \pi_2)\}$。跟之前一样，$D_1$具有冲突构造，但是现在通过具有谓词的DFA转换：$D_0 \xrightarrow{a} D_1$，$D_1 \xrightarrow{\pi_1} f_1$，$D_1 \xrightarrow{\pi_2} f_2$，我们可以在运行时解决这个问题。

如果解析$(resolve)$函数找到了谓词，它便会直接返回而不会产生警告，让$createDFA$函数将谓词纳入到DFA中。如果没有谓词，我们将无法在运行时解决这样的问题，所以解析$(resolve)$函数会通过消除更高数字编号的冲突备选产生式构造，给予A的最低数字编号的冲突备选构造一个优先级，来静态地消除这样的二义性。例如，在上述A的无谓词版语法中，最后得到的DFA是$D_0 \xrightarrow{a} f_1$，因为分析算法通过移除与最高优先级产生式1无关的构造来解决冲突，保留$\{(p_3, 1), (p_4, 1)\}$。

如果闭包$(closure)$算法不小心触发了递归溢出警告，解析$(resolve)$算法可能将无法在D上看到冲突构造，在这种情况下，由于分析算法会为了避免死循环而提前终止分析，所以D仍然可能预测出多个备选产生式。如果谓词存在，算法可以使用谓词来解决运行时潜在的文法二义性。如果不存在，算法将会再次选择数字编号最小的备选方案，并向使用者发出警告。

5.3 避免分析成为"烫手的山芋"

由于LL-正则(LL-regular)条件通常是无法判定的，故我们能够预计到在任意的前看DFA构造算法中都有可能出现潜在的死循环。递归规则通常是死循环的源头。给定一个构造$c = (p, i, \gamma)$（为了间接起见，我们从现在开始忽略构造中的π），而c在转换$p \xrightarrow{A} p'$的闭包构造则为$(p_A, i, p'\gamma)$。如果闭包再次到达状态p，那么它的构造就会变成$(p_A, i, p'p'\gamma)$。最终，闭包将会永远"泵动"递归规则，导致爆栈。例如，**图3**中非终结符A的带有递归的ATN，其：$S \rightarrow Ac|Ad$的DFA起始状态D_0是：

$$D_0 = \{(p_1, 1, []), (p_A, 1, p_2), (p_7, 1, p_2), (p_{10}, 1, p_2)$$
$$(p_4, 2, []), (p_A, 2, p_5), (p_7, 2, p_5), (p_{10}, 2, p_5)\}$$

函数$move(D_0, a)$将会达到$(p_8, 1, p_2)$，通过D_0中的构造$(p_7, 1, p_2)$创建新的状态D_1。而$(p_8, 1, 2)$的闭包操作将会遍历隐式的通过ϵ到p_a的边，并因此创建3个新的构造：$(p_A, 1, p_9p_2)$，$(p_7, 1, p_9p_2)$，$(p_{10}, 1, p_9p_2)$。D_1的p_7构造和D_0是一样的，只不过具有更大的状态栈。在递归深度为m时，由于构造$p_9^m p_2$而无限扩大，从而产生一个越来越大的DFA路径：$D_0 \xrightarrow{a} D_1 \xrightarrow{a} \ldots \xrightarrow{a} D_m$。

而这样的问题具有两种解决方案。要么我们舍弃除了栈顶状态m以外的所有状态（正如Bermudez和Schimpf在$LAR(m)$文法中做的那样），要么我们直接避免对任意指定子机的起始状态进行m次针对构造的闭包算法的递归调用。最终我们选择了后者，因为它保证了一个更加严格的$LL(k)$超集（$m \geq k$）。通常情况下，我们无须担心一个近似值会在备选方案之间引入无效序列。相反，Bermudez和Schimpf给出了一个$LALR(1)$文法，不过对于这个文法家族中的每一个文法，都没有固定的m值来给出一个有效的$LAR(m)$解析器。在实践中，硬性的递归深度限制也并非是一个严重的局限。例如程序员们通常会使用(正则)文法：$S \rightarrow a^*bc|a^*bd$而不是如**图3**所示的版本。

5.4 中止DFA的构建

作为启发，当我们发现非终结符A在不止一个备选方案中包含递归时，我们将会中止非终结符A的DFA的构造（在**图3**中对S的分析将会在触发递归溢出前终止）。而这些决策明显不可能有准确的正则分区，并且我们的算法不会采用近似前看的策略，因此在这种情况下去追求一个DFA毫无意义。而ANTLR的实现则是为A回退到$LL(1)$前看，如当解析$(resolve)$算法在多个备选方案中检测递归时，通过回溯(backtracking)或者其他谓词来解决。

5.5 提升(hoisting)谓词

这种算法和第3节中给出的正式的谓词文法语义要求谓词必须出现在产生式左推导边。而这样的限制在实践中是不够灵活的，并且会迫使用户冗余地使用谓词，如产生式的派生式中的谓词等等。ANTLR的完整算法可以自动查找并为决策提升其所有谓词的可见性，即使是存在于派生链上更下游的产生式。ANTLR的分析同样也能处理在右侧产生式中出现的EBNF操作符，例如在ATN中添加循环来处理$A \rightarrow a^*b$。

6. 实验结果

本文为我们的分析算法和$LL(*)$解析策略的适用性和效率做了不少了的声明。在本节中，我们会在以下六个大型、真实的语法上运行ANTLR3.3：

- **Java1.5**：原生ANTLR语法，使用PEG模式

- **RatsC，RatsJava**：$Rats!$语法，使用PEG模式手动地将其转换为ANTLR语法，同时保留基本结构。但剔除了$Rats!$支持但ANTLR不支持的左递归
- **VB.NET，TSQL，C#**：由$Temporal\ Wave,\ LLC$提供的微软商业语法

并在大型输入样本上做以下生成的解析器做性能评估：

- **RatsJavaParser.java**：由ANTLR为RatsJava语法生成的解析器；会同时测试两种Java的解析器
- **pre_javaParser.c**：预处理ANTLR为Java1.5语法生成的解析器；用于测试RatsC解析器
- **LinqToSqlSamples.cs**：微软示例代码；用于测试c#
- **big.sql**：收集好的微软SQL示例；用于测试TSQL
- **Northwind.vb**：微软示例代码；用于测试VB.NET

从而支持本文中的许多声明。我们引入两个PEG派生语法来展示ANTLR能够从PEGs生成有效的解析器。

6.1 静态语法分析

本文曾表示我们的分析算法能够从绝大多数的解析决策中，静态地优化回溯部分，并且能够在合理的时间内完成。表1总结了ANTLR对上诉6种语法的分析结果：

Grammar	Lines	n	Fixed	Cyclic	Backtrack	Runtime
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. 不同语法的决策分析结果。"Lines"表示语法的行数，n是语法中解析决策的次数，"Fixed"是纯$LL(k)$决策的次数，"Cyclic"是纯循环（没有回溯）DFA的次数，"Backtrack"是可能的回溯次数，"Runtime"是处理语法并生成一个$LL(*)$解析器的时间。以上所有的测试都基于：OS X，2x3.2Ghz 4核Intel至强处理器，14G内存

除了8231行的SQL语法外，ANTLR处理每个语法的时间都在几秒内。解析时间包括语法分析、解析（包括EBNF结构的处理和谓词的提升）以及生成语法解析器。（和指数级复杂度的经典子集构造算法一样，ANTLR的解析在极少数情况下也会"踩雷"。不过ANTLR提供了一种方法来隔离这些违规的决策并且手动设置它们的的前看参数，之后，这些语法将能避免"踩雷"）。

所有语法在一定程度上都用到了回溯，这一定程度上说明使用ANTLR就没有必要再去迫使将大型文法修改成$LL(k)$了。前三个语法使用的是PEG模式，在这种模式下，ANTLR会自动为每个产生式的左推导路径上添加一个语法谓词(syntactic predicate)。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的发生。例如，在Java1.5中，除了11.8%的决策外，ANTLR在其他的决策中都移除了语法谓词。不出意外的是，RatsC语法的回溯决策占比最高，达到22.4%，这是由于C语言的变量声明和和函数声明从左边看是一样的。而三个商业语法的作者手动指定了谓词，从而降低了前看的标准。6个语法的4个中，ANTLR都能构建循环DFA来避免回溯。

Grammar	$LL(k)$	$LL(1)$	Lookahead depth 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. 不同语法的固定前看决策分析结果。译者注：$LL(k)$那一栏表示$LL(k)$占总决策的比例，$LL(1)$那一栏表示$LL(1)$占总决策的比例，总决策数量在表1的n列

表1数据显示，样本语法中的绝大多数决策都是某个k值下的固定$LL(k)$。而表2则显示了每个前看深度(Lookahead depth)的决策数量，而且还显示大多数的决策实际上都是$LL(1)$。上述的表分析结果同时还表明，尽管存在一般情况下无法进行判断的问题，但ANTLR几乎还是能在解析时一直静态地确定k的取值。

6.2 解析器运行时的剖析

之前我们提到，在运行时$LL(*)$解析决策通常只会使用前几个前看词法单元(token)，并且能够以一个合适的速度进行解析。表3则表示，对于每个输入示例文件，每个决策事件的前看深度大约是1个词法单元，而PEG模型的解析器通常需要2个词法单元。

Grammar	Input lines	parse-time	n	avg k	back. k	max 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. 解析器决策时的前看深度。n是解析时覆盖的决策点数量，“avg k”是决策前看深度之和除以决策数量之和，“back k”是仅在回溯决策事件时的平均推测(speculation)深度，“max k”是在解析过程中遇到的最深的前看深度。以上所有的测试都基于：Java1.6.0 OS X，2x3.2Ghz 4核Intel至强处理器，14G内存

表3显示，前三种语法的解析速度大约在26000行/秒，而其他存在树状构造的前看语法则大约在9000行/秒。

仅对于回溯决策来说，其平均前看深度不超过6个词法单元，彰显了一个事实：尽管回溯的前看可能往前扫描很深很深，但通常情况下这并不会发生。例如，一个解析器可能需要回溯来匹配一个C语言的类型关键字，但在大多数情况下，这些关键字看起来就像`int`或者是`int *`，因此可以被很快识别出来。而且不是由PEG派生的语法具有更小得多的最大前看深度，这表明语法的作者进行了一些重构来利用$LL(k)$的效率。与此相反，RatsC语法回溯了整个函数(在一个决策事件中前看了7968个词法单元)来区分函数声明还是函数定义，但实际上`int f();`和`int f() {...}`在`;`和`{`部分就已经做出了区分。

尽管我们从$Rats!$中派生了2种语法进行实验，但我们并没有比较它们的解析执行时间，也没有比较它们的内存利用情况。ANTLR和$Rats!$都是一种实践型解析器生成器，在大多数应用中，其解析速度和内存利用情况都是比较出色的。不像ANTLR，$Rats!$是一种弱扫描型的解析器生成器，所以二者在内存缓存的比较上会很棘手。

如表1所示，ANTLR能够静态地在大多数解析决策中移除回溯。而在实际运行时，ANTLR进行回溯的次数甚至比静态分析预测出的还要少。举个例子，在静态分析中，我们发现我们的样本语法会有平均10.9%的决策回溯，但是表4显示生成的解析器仅有平均6.8%的决策事件会出现回溯。非PEG派生的语法的回溯率仅为2.5%。部分原因是由于一些回溯决策具有不常见的语法结构。例如，TSQL语法共有1120种决策，但是我们用于示例输入的只有其中的309种。

最重要的是，一个决策能够回溯但并不意味着它一定会走向回溯。表4的最后一列显示，在样本语法中，可能会回溯的决策（potentially backtracking decisions，简称PBDs）也只有平均一半的时间走了回溯。商业语法VB.NET和TSQL产生的PBDs走了回溯的也只占其总决策的30%不到。而有些PBDs从来也不会触发回溯。如果将表4的前两列相减，我们就可以计算出PBDs中没有走回溯的数量。

Grammar	Can back.	Did back.	decision events	Back-track	Back. rate
Java1.5	19	16	462,975	2.36%	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%

表 4. 解析器决策时回溯行为分析。“Can back”是可能走回溯的数量，“Did back”是对应样本语法中实际走了回溯的数量，“Backtrack”是回溯次数占所有决策事件的比例，“Back. rate”是可能走回溯决策实际做出回溯的概率

应当指出，不考虑内存大小，最坏情况下，回溯解析器的复杂度将会呈指数级增加。这对于需要进行大量嵌套回溯的语法来说这点不可忽视。例如，如果我们禁止ANTLR的缓存策略，RatsC语法的解析工作将看不到尽头。但与之相反的是，VB.NET和C#解析器在这种情况却运行良好。Packrat解析以增加缓存为代价，换取线性的（$O(n)$）解析性能。在最坏的情况下，我们需要缓存$o(|N| * n)$个决策结果（在每个输入读取位置为每个非终结符做一个决策缓存）。因为ANTLR只会在预测的时候进行缓存，所以对于ANTLR来说，回溯的次数越少，内存的占用也就越小。

7. 相关工作

解析技术有很多，但目前最主要的两种测试是Tomita的自底向上的GLR分析法和Ford的自上而下的packrat分析法（通常用与其相关的元语言PEG来表示）。这两种语法都是非确定性的，因其对应的解析器可以使用某种形式的预测来做决策。$LL(*)$是对packrat解析器的优化，正如GLR是对Earley算法的优化。语法越接近底层的LL文法策略或LR文法策略，其对应解析器在时间复杂度和空间复杂度上也就越好。$LL(*)$解析的时间复杂度从$O(n)$到$O(n^2)$不等，而GLR则从$O(n)$到$O(n^3)$不等。令人惊讶的是，造成$O(n^2)$复杂度的潜在因素是循环前看DFA而不是回溯（假设我们在回溯做了缓存）。ANTLR生成的$LL(*)$解析器在实践中证明是线性的（$O(n)$的时间复杂度），并且与纯粹的packrat解析器相比，其大大减少了预测的次数和内存开销。

GLR和PEG是趋向于轻扫描的，而这种特性在某些语法工具需要支持语法合成$(grammar)$的情况十分有必要。语法合成意味着程序员可以轻易地通过修改源语法，或是将源语法的一些语法碎片进行合成，将一种语言整合到另一种语言中甚至是直接创建出新的语法。例如，$Rats!$-Jeannie语法优雅地组合了C和Java两种语法。

$LL(*)$文法背后的理念来源于20世纪70年代。当时的$LL(*)$解析器由Jarzabek、Krawczyk、Nijholt提出，这种解析器实现了LL-正则文法，但不具备谓词前看的DFA边。Nijholt和Poplawski给出了线性的两步$(two\text{-}pass)$$LL$-正则文法解析策略，第一次的解析方式必须从右向左解析，并且输入流必须是有限输入流（比如不能是套接字（socket）输入流或者是交互式（interactive streams）输入流）。最后，他们也没有考虑语义谓词。

Milton和Fischer在$LL(1)$文法中引入了语义谓词，但每个产生式只允许拥有一个语义谓词来指导解析策略，而且要求使用者将前看符集合和对应谓词进行绑定。Parr和Quong引入了语法谓词和语义谓词提升(semantic predicate hoisting)的概念，即将其他非终结符的语义谓词纳入合并到解析决策中。不过，他们都没有提供一个正式的谓词文法规范，也没有提供可见谓词的查找算法。在本文中，我们给出了一个正式的语法规范，并演示了在DFA构建过程中发现有限(limited)谓词的方法。Grimm在其基于PEG的$Rats!$文法中支持了受限(restricted)的语义谓词和任意动作(action)，但这些特性依赖于程序员去避免不可撤销的副作用。最近，Jim等人为使用Yakker处理CFG文法的转换器拓展了语义谓词（译者注：我也不知道作者写这句话想表达什么）。

ANTLR v1.x引入了语法谓词作为在解析阶段的一种能够手动控制的回溯装置。这种能为产生式确定优先级的能力出道即得到追捧，但其真正的实现却意料之外的未能得到重视。Ford用PEG将有序排列的产生式规范化。v3.x之前的ANTLR版本中，如果不进行解析缓存，回溯的时间复杂度将达到指数级别。不过Ford通过引入packrat解析器解决了这个问题。在ANTLR v3.x的版本中，用户可以通过一个选项开启解析缓存的功能。

LL-正则文法是LR-正则(LR-regular)文法的模拟产物。Bermudez和Schimpf为LR-正则文法提供了一种名为$LAR(m)$的解析策略。参数m是堆栈的约束，和ANTLR类似，也是用来避免解析算法变得无穷无尽。$LAR(m)$构建了一个$LR(0)$模型，并嫁接到前看DFA中来处理非终结符。为每个LL-正则和LR-正则解析决策找到对应的正则分区是不可能的。$LL(*)$和$LAR(m)$算法可以分别为LL-正则文法和LR-正则文法分析决策的子集构造一个有效的DFA。在自然语言领域，Nederhof使用DFA来近似模拟整个PEG，大概是为了得到更快的语言关联性检查，但这样却降低了检测精度。Nederhof将规则调用进行内联，并指定其最大调用深度m，实际上也成功地模仿了$LAR(m)$中的常数m。

8. 结论

由于语义谓词的存在，$LL(*)$解析器的表达能力和PEG不相上下，甚至更强。虽然GLR文法能够接受左递归的形式，但在识别一些上下文有关的文法上，其能力却无法与$LL(*)$媲美。与PEG或者GLR不同的是，$LL(*)$解析器能够执行任意动作(action)，并在调试和错误处理上提供了更好的能力支持。$LL(*)$分析算法能够构建循环前看DFA来处理非-$LL(k)$构造，并且在未能找到合适的DFA时，能够通过语法谓词进行回溯。实验表明，ANTLR解析器生成工具能够构建高效率的解析器，这些解析器几乎消除了所有的回溯。并且ANTLR在实践中得到了广泛的应用，这表明$LL(*)$在解析领域占据了非常有利的位置。

最后，我要感谢Sriram Srinivasan对$LL(*)$这一名词的提出和为此所做的所有贡献。