

LL(*)文法: ANTLR语法规解析生成器的基础

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,但语法规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

尽管语法规解析一直被相当重视,并且有着一段很长的学术研究历史,但语法规解析并不是一个已经完全解决的问题。由于手工编写语法规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规解析器。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,因此一个语法规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 $LALR(1)$ 或者 $LL(1)$ 文法的语法规解析器生成器。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 文法规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,如果不调过彻底的调试,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向前看，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向前看符号的问题。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) 到升级到任意的向前看字符 (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sourceforce.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 Manager、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。

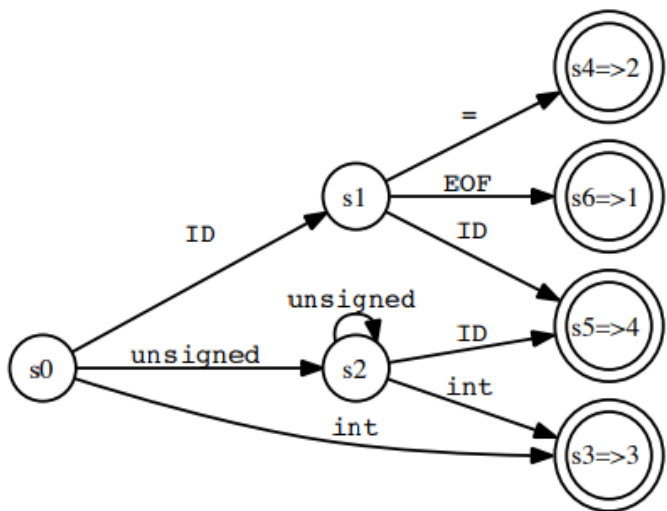


图1. 语法规则 s 的前看DFA，符号 $sn=>i$ 表示“谓词(predict)(选择)第 i 个备选分支”

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。

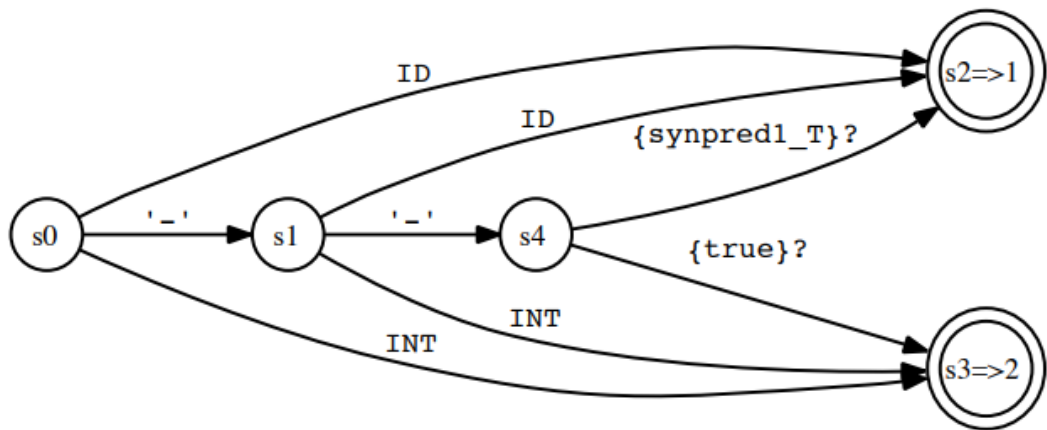


图2. 语法规则s2的解析决策DFA，使用了混合的 $k \leq 3$ 前看和回溯（译者：synpred1_T大概率指的是syntactic predicates times(这里是1) T(True)的缩写，大概是判断是否达到了递归展开数量的阈值常量 $m=1$ ）

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 $G = (N, T, P, S, \Pi, \mathcal{M})$ ，它有以下元素：

- N 是非终结符(规则名称)集合
- T 是终结符(词法单元)集合
- $S \in N$ 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)上下文无关文法生成物
$A \rightarrow (A'_i) \rightsquigarrow \alpha_i$	基于谓词判断后的语法 A'_i 的 i^{th} (第 i 个)产生式
$A \rightarrow \{\pi_i\} ? \alpha_i$	基于谓词判断后的语义(<i>semantics</i>)的 i^{th} (第 i 个)产生式
$A \rightarrow \{\mu_i\}$	修改器(<i>mutators</i>)生成的产生式

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：

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

$$Action \frac{A \rightarrow \{\mu\}}{(\mathbb{S}, uA\delta) \xRightarrow{\mu} (\mu(\mathbb{S}), u\delta)} \quad (2)$$

$$Sem \frac{\pi_i(\mathbb{S}) \quad A \rightarrow \{\pi_i\}?\alpha_i}{(\mathbb{S}, uA\delta) \xRightarrow{\pi_i} (\mathbb{S}, u\alpha_i\delta)} \quad (3)$$

$$Syn \frac{\begin{array}{c} (\mathbb{S}, A'_i) \Rightarrow^* (\mathbb{S}', w) \\ w \preceq w_r \\ A \rightarrow (A'_i) \rightsquigarrow \alpha_i \end{array}}{(\mathbb{S}, uA\delta) \xRightarrow{A'_i} (\mathbb{S}, u\alpha_i\delta)} \quad (4)$$

$$Closure \frac{(\mathbb{S}, \alpha) \xrightarrow{\lambda} (\mathbb{S}, \alpha'), (\mathbb{S}, \alpha') \xrightarrow{\vec{\lambda}} *(\mathbb{S}, \beta)}{(\mathbb{S}, \alpha) \xrightarrow{\lambda\vec{\lambda}} *(\mathbb{S}, \beta)} \quad (5)$$

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：

$(\mathbb{S}, \alpha) \xrightarrow{\lambda} (\mathbb{S}', \beta)$ (译者注：这里应该指的是的 $Closure$ 公式中分子的这个 $(\mathbb{S}, \alpha) \xrightarrow{\lambda} (\mathbb{S}, \alpha')$ 公式) 可以理解为：在当前的机器状态 \mathbb{S} ，输入文法序列 α 后，将在下一步将 \mathbb{S} 归约成 \mathbb{S}' 和新的文法序列 β ，并同时发射(emit)一个追踪(trace) λ 。判断式：

$(\mathbb{S}, \alpha) \xrightarrow{\vec{\lambda}} *(\mathbb{S}', \beta)$ (译者注：这里同样应该指的是的 $Closure$ 公式中分子的这个 $(\mathbb{S}, \alpha') \xrightarrow{\vec{\lambda}} *(\mathbb{S}, \beta)$ 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 α 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 $R = (R_1, R_2, \dots, R_n)$ 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 $x \equiv y \pmod{R}$

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

$$(\epsilon, S) \Rightarrow^* (S, w_i A \delta_i) \Rightarrow (S, w_i \alpha_i \delta_i) \Rightarrow^* (S_i, w_i x) \quad (1)$$

$$(\epsilon, S) \Rightarrow^* (S, w_j A \delta_j) \Rightarrow (S, w_j \alpha_j \delta_j) \Rightarrow^* (S_j, w_j y) \quad (2)$$

$$x \equiv y \pmod{R} \quad (3)$$

则我们称G是谓词的(predicated)LL - 正则(LL - regular)文法。

并且 $\alpha_i = \alpha_j$ 和 $S_i = S_j$ 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和其指定的非终结符相关联的正则分区R进行匹配(match)，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
- $\Sigma = T \cup \Pi$ 是边的符号表
- Δ 是映射(mapping) $Q \times \Sigma \rightarrow Q$ 的转换(transition)函数
- $D_0 \in Q$ 是起始状态
- $F = \{f_1, f_2, \dots, f_n\}$ 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' ：

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

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

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

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*)：

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

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 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，ANTLR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 $C(\alpha_i)$ 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 $C(\alpha_i)$ 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 $A \rightarrow [A]id$ ，即上下文无关文法 $\{[{}^n id]^n\}$ 。我们可以用这么一个满足 $LL -$ 正则文法条件的： $R = \{\{[*id]^*\}, \{id\}\}$ 正则表达式来近似的表示 $C(\alpha_i)$ 。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： $LA = \{\{[], \{id\}\}$ ，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。在上诉的情况下，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 是转换关系的映射 $Q \times (\Sigma \cup \epsilon) \rightarrow Q$
- $E = \{p_A | A \in N\}$ 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。

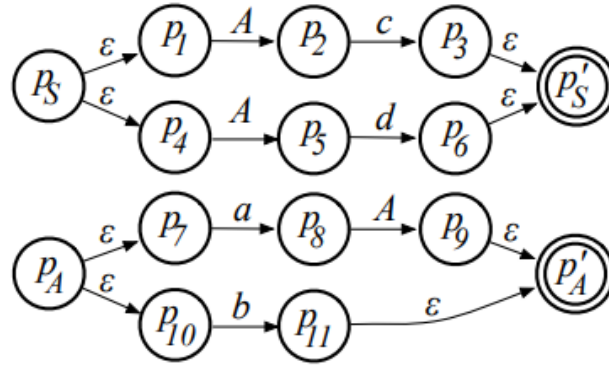


图3. $P = \{S \rightarrow Ac | Ad, A \rightarrow aA | b\}$ 时，文法 G 的 ATN 示意图

非终结符的转换边 $p \xrightarrow{A} p'$ 就像是函数调用。它们将 ATN 的控制权转移给 A 的子机，并将返回状态 p' 压入状态栈，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。初始状态 p_A 指向能够通过上文的路径转换得到 α_i 的 $p_{A,i}$ ，而最终状态则是从 α_i 指向下文的 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
$A \rightarrow \alpha_i$	$p_A \xrightarrow{\epsilon} p_{A,i} \xrightarrow{\epsilon} \boxed{\alpha_i} \xrightarrow{\epsilon} p'_A$
$A \rightarrow \{\pi_i\} ? \alpha_i$	$p_A \xrightarrow{\epsilon} p_{A,i} \xrightarrow{\pi_i} \boxed{\alpha_i} \xrightarrow{\epsilon} p'_A$
$A \rightarrow \{\mu_i\}$	$p_A \xrightarrow{\epsilon} p_{A,i} \xrightarrow{\mu_i} p'_A$
$A \rightarrow \epsilon$	$p_A \xrightarrow{\epsilon} p_{A,i} \xrightarrow{\epsilon} p'_A$
$\boxed{\alpha_i} = X_1 X_2 \dots X_m$ for $X_j \in N \cup T, j = 1..m$	$p_0 \xrightarrow{X_1} p_1 \xrightarrow{X_2} \dots \xrightarrow{X_m} p_m$

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达到的节点。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，通过可达到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 $p_{A,i}$ 出发的路径不断收敛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, γ 以及 c, π 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:

Alg. 1 : *createDFA*(ATN State p_A) **returns** DFA

$work := \square$; $\Delta := \{\}$; $D_0 = \{\}$;

//译者注: $numAlts(A)$ 即*number of alternative A*, 表示和备选的产生式数量有关

$F := \{f_i \mid f_i := \text{new DFA state}, 1 \dots numAlts(A)\}$;

$Q := F$;

foreach p_A

算法1. 从ATN构建DFA

函数*createDFA*, 是算法的入口: