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 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。

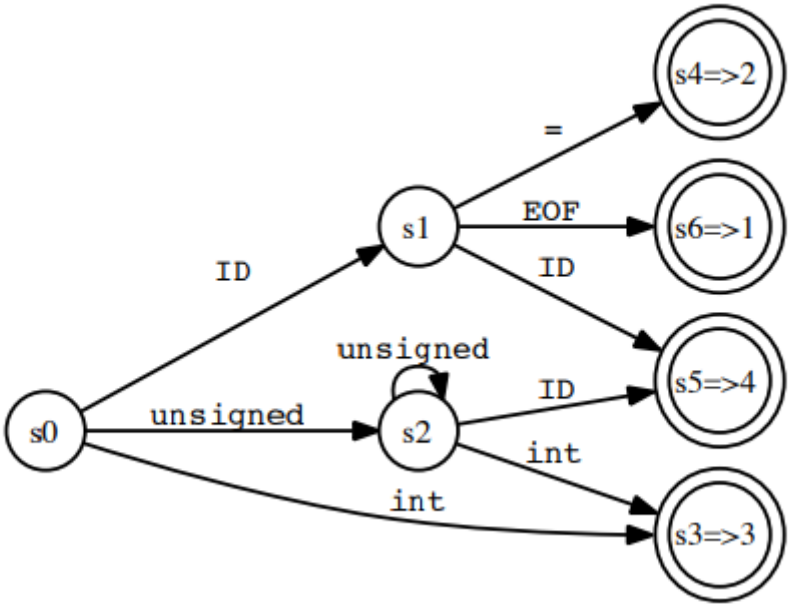

图1. 语法规则s的前看DFA，符号$sn => i$表示"预测(predict)第i个备选分支"

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。

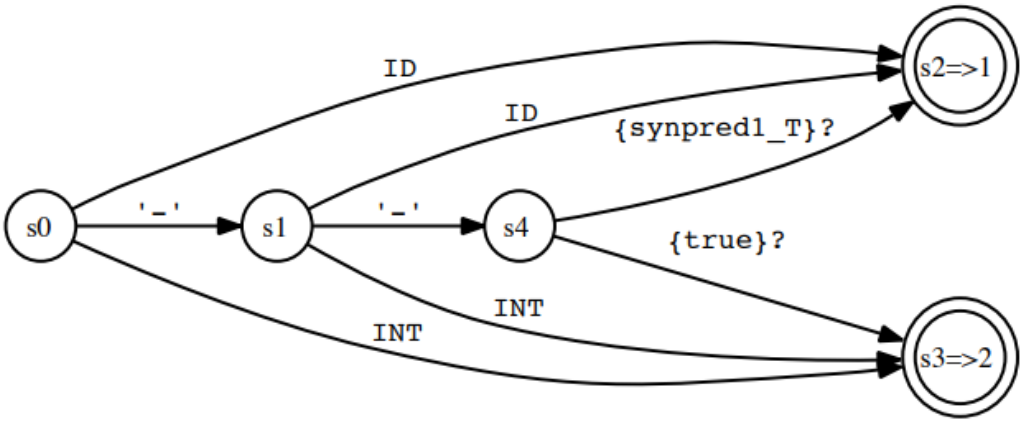

图2. 语法规则$s2$的解析决策DFA，使用了混合的$k3 \leq 3$前看和回溯（译者：synpred1_T这里指的是syntactic predicates number T，是否达到了递归展开的阈值常量$m=1$)

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：

$A\,\epsilon\,N$	非终结符号
$a\,\epsilon\,T$	终结符号
$X\,\epsilon\,(N\cup T)$	文法符号
$\alpha,\,\beta,\,\delta\,\epsilon\,X^*$	文法符号序列
$u,\,x,\,y,\,w\,\epsilon\,T^*$	终结符号序列
$w_r\,\epsilon\,T^*$	剩余的输入终结符
ϵ	空串
$\pi\,\epsilon\,\Pi$	和实现语言相关的谓词
$\mu\,\epsilon\,\mathcal{M}$	和实现语言相关的动作
$\lambda\,\epsilon\,(N\,\cup\,\Pi\,\cup\,\mathcal{M})$	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
$A\rightarrow\,(A_i')\Rightarrow\,\alpha_i$	基于$(predicated)$文法A_i'的i^{th}(第i个)生成物
$A\rightarrow\,\{\pi_i\}?\alpha_i$	基于$(predicated)$语义$(semantics)$的i^{th}(第i个)生成物
$A\rightarrow\,\{\mu_i\}$	修改器$(mutators)$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：

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

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

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

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

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

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。