

# 动态分析之力：自适应(*Adaptive*) $LL(*)$ 解析算法

## 摘要

尽管如PEG,  $LL(*)$ , GLR和GLL等现代解析策略取得了不小的进步, 但解析本身并不是一个完全被解决了的问题。现有的方式存在许多缺陷, 如: 难以支持增强式嵌入动作(action), 性能低下或者说不可预测的解析速度, 以及有悖直觉的解析策略。本文介绍的 $ALL(*)$ 解析策略结合了传统的自上而下 $LL(k)$ 解析器的简单、高效、可预测能力, 以及类GLR机制的强大解析决策功能。其关键的创新之处在于将语法分析推迟到了解析运行时, 这使得 $ALL(*)$ 能够处理任何非左递归的上下文无关文法。理论上 $ALL(*)$ 的解析时间复杂度是 $O(n^4)$ , 但在实际的文法分析中却始终呈线性复杂度, 性能比GLL和GLR等一般的解析策略高出好几个数量级。ANTLR4生成的 $ALL(*)$ 解析器, 可以通过重写文法来支持直接左递归。ANTLR4的广泛使用 (2013 年的下载量为 5000 次/月) 表明,  $ALL(*)$ 对各种应用程序都卓有成效。

## 1. 导论

尽管现代语法解析策略已经非常先进成熟, 并且在学术研究上历史悠久, 但在实践中, 计算机语言的解析仍然未能被彻底解决。迫使程序员修改文法以匹配确定性 (注: 确定性一词, 指的是确定性有限自动机DFA, 与之对应的非确定性有限自动机NFA)  $LALR(k)$ 或 $LL(k)$ 解析器生成器的规范来提升解析性能, 在硬件资源稀缺时是无可厚非的。但随着硬件资源成本的不断降低, 研究人员开发出了功能更加强大、但解析成本更高的非确定性解析策略, 这些策略遵循"自下而上" ( $LR$ 风格) 或"自上而下" ( $LL$ 风格) 的解析方式。这些策略包括GLR, 解析器表达式语法(PEG Parser Expression Grammar), 来自ANTLR 3的 $LL(*)$ 以及最近提出的一种全通用自上而下的GLL解析策略。

虽然这些策略在使用上比 $LALR(k)$ 和 $LL(k)$ 解析器生成器更加便捷, 但它们也存在各种短板。

首先, 非确定性解析器有时会出现未预料的行为。由于GLL和GLR是专为处理自然语言语法设计的, 而通常情况下, 这种语法会潜在地包含二义性, 故GLL和GLR会为存在二义性的语法返回多种解析树 (森林)。对于计算机语言来说, 二义性几乎就是一种异常。当然为处理这种不常见的情况, 我们可以对构建出的解析森林进行遍历来消除此等二义性, 但这往往意味着需要消耗额外的时间、空间和硬件资源。根据语法定义, PEG是一种无二义性的文法, 但存在一种特殊情况, 即规则 $A \rightarrow a \mid ab$  (表示非终结符 $A$ 要么匹配 $a$ 要么匹配 $ab$ ) 永远无法匹配 $ab$ , 因为只要输入的前缀匹配 $a$ , PEG文法将会选择第一种产生式。存在嵌套的回溯时, 调试PEG将变得举步维艰。

其次, 像打印语句这种由程序员提供的能够对解析过程产生影响的动作(actions)或者是修改器(mutators)都应当在持续预测 (PEG) 或支持多种解释 (GLL和GLR) 的解析策略中避免, 因为这些动作或修改器可能并不应当执行 (虽然DParser支持'final'动作, 即如果程序员确定某次归约是一个二义性语法的最终产物之一, 那么这个动作或者修改器将得到执行)。在没有副作用的情况下, 可能触发的动作必须在某个不可变的数据结构被缓存下来, 或者是解析本身提供撤销操作。前一种机制受到内存大小的限制, 而后一种机制实现则相对复杂并且有可能无法实现。而避免上述副作用的一种典型解决方式是构建一个解析树, 用于解析后 (post-parse) 处理, 但这种刻意产物从根本上限制了对输入大小的限制, 因为要将整个输入的解析树放入内存中 (译者注: 要将整个输入整成一个解析树放内存里, 可不得要求输入尽量小点嘛)。构建解析树的解析器是无法为大文件输入或者是无边界流数据提供解析能力, 除非可以将这些数据按照逻辑分块(logical chunks)进行处理。

最后, 我们的实验结果 (第 7 节) 表明, GLL和GLR在时间和空间上都性能低下, 并且无法预测。它们的复杂度分别为 $O(n^3)$ 和 $O(n^{p+1})$ , 其中 $p$ 是语法中最长的产生式长度 (GLR的经典复杂度为 $O(n^3)$ , 那是因为Kipps给出的这个算法具有一个高得不合理的常数)。理论上, 通用解析器应当在处理确定性语法时具有线性 ( $O(n)$ ) 的时间复杂度。但在实践中, 我们发现GLL和GLR在12920个Java6的库源代码文件 (约123M) 的解析上, 要比 $ALL(*)$ 慢大约135倍, 而在一个3.2M大小的Java源文件解析上, 甚至要比 $ALL(*)$ 慢6个数量级。

$LL(*)$ 文法通过提供一种确定性的解析策略来解决上述的短板，该解析策略使用正则表达式（表现为一种确定性有限自动机DFA）来处理剩下的输入，而不是像 $LL(k)$ 那样的固定 $k$ 长度序列。即使前看序列（一个所有可能的剩余输入短语集）通常是上下文无关的，但使用正则的前看语法DFA用作前看策略便能够将 $LL(*)$ 的决策进行限制，并区分出不同的产生式。但 $LL(*)$ 语法存在一个主要的问题，它在静态阶段是无法进行文法判定的，并且有时候也无法找到能够区分不同产生式的正则表达式。ANTLR 3的静态分析能够检测并避免潜在的不可判定情况，并将这种情况回退到回溯解析策略。这使得 $LL(*)$ 具有与PEG解析类似 $A \rightarrow a \mid b$ 语法时相同的窘境。回溯解析策略也无法检测如 $A \rightarrow \alpha \mid \alpha$ 这样明显的二义性错误，因为回溯会始终选择第一个产生式进行匹配，故其中的文法符号序列 $\alpha$ 就使得 $\alpha \mid \alpha$ 不再是 $LL(*)$ 规范的了。

## 1.1 动态语法分析

在本文中，我们将介绍自适应（Adaptive） $LL(*)$ 文法，简称 $ALL(*)$ 文法，这种解析器将自顶向下解析器具有的简单易实现和类GLR解析器具有的一些强大机制进行结合，并在解析时做出决策。具体来说， $LL$ 解析会在每个预测决策点（非终结符）挂起，然后会在预测机制选择了合适的产生式去展开的时候恢复。而关键的创新在于将语法分析转移到解析时；不再需要静态的语法分析。这种方式可以使我们避免在静态 $LL(*)$ 语法分析的不可判定性，并且让我们能够为任何非左递归的上下文无关文法(Context Free Grammar, CFG)生成正确的解析器（定理6.1）。相比于静态分析需要考虑所有可能得输入序列，动态分析只需要考虑当前实际输入序列的有限结合。

$ALL(*)$ 预测机制背后的理念是在每个备选产生式的决策点启动(launch)一个子解析器。子解析器以一种假并行的方式去展开所有可能的路径。当子解析器展开的某条路径和剩余输入序列不匹配时，子解析器就会被销毁。子解析器之间步调一致的向前读取输入序列，这样分析过程就能以最小的前看深度，识别出唯一匹配的的产生式，这样就具备了唯一预测产生式的能力。当然，如果多个子解析器在最后碰面或者一起达到输入序列末尾，预测器就会告知一个二义性，并根据存活子解析器相关的产生式编号的最小值（即最高优先级）来解决这个二义性（产生式编号代表优先级，是类PEG解析器默认解决二义性的一种方式；Bison解析器还可以通过自选产生式来解决冲突）。当然，程序员可以嵌入语义谓词（semantic predicates）来在具有二义性的产生式之间做出选择。

$ALL(*)$ 解析器能够对分析结果做缓存，以增量和动态的方式为一个DFA做映射缓存，即前看短语序列到预测产生式的映射（在这里我们使用分析(*analysis*)一词的原因在于 $ALL(*)$ 的分析也能够像 $LL(*)$ 分析(*analysis*)时那样，产生前看DFA）。解析器可以通过查询缓存，快速地在同一个解析器决策和前看短语中做出后面的解析预测。而未命中缓存的输入短语将会触发语法分析机制，并同时预测一个备选产生式然后更新DFA到缓存中。尽管在给定决策下的前看语言通常是上下文无关的，但DFA仍然很适合用来记录预测结果。动态分析只需要考虑在解析时遇到的有限上下文无关文法子集，并且任意的有限子集都是有规律可循的。

为避免非确定性子解析器具有的指数复杂度这一性质，预测采用了一种叫图形化结构堆栈(*GSS*(*graph - structured stack*))的数据结构来避免冗余计算。GLR使用的策略与此基本相同，只不过 $ALL(*)$ 只预测使用了这种策略的子解析器的产生式，而GLR直接使用这种方式进行解析。因此，GLR必须将非终结符压入 $GSS$ 中，而 $ALL(*)$ 则不需要。

$ALL(*)$ 解析器以 $LL$ 的简易性来处理终结符的匹配和非终结符的展开任务，但其理论的时间复杂度为 $O(n^4)$ ，因为在最坏情况下，解析器必须对每个输入符号进行预测，而每次预测都必须检查整个剩余输入。检查一个输入符号的时间复杂度为 $O(n^2)$ 。 $O(n^4)$ 的时间复杂度和GLR一致。不过在第7节中，我们通过实测证明，常见语言的 $ALL(*)$ 解析器是十分高效的，并在实践中表现出线性的时间复杂度。

$ALL(*)$ 的优势在于将语法分析推迟到解析时，但这一选择给语法的功能测试带来了额外的负担。与所有动态解析方式一样，程序员必须覆盖尽可能多的语法定义和输入序列组合，以找到语法歧义（二义性）。标准的源代码覆盖工具可以帮助程序员测量 $ALL(*)$ 解析器的语法检查覆盖情况。生成代码中的语法覆盖率有多高语法检查覆盖率就有多高。

$ALL(*)$ 算法是ANTLR 4解析器生成器的基础（ANTLR 3基于 $LL(*)$ ）。ANTLR 4于2013年1月发布，每月下载量约为5000次（从web服务日志中过滤出人为下载痕迹，并使用唯一IP地址来计算包括源代码、二进制文件或ANTLRworks2 开发环境的下载次数）。这些活跃的数据恰恰证明 $ALL(*)$ 的用途广发和可用性。

本文的其余部分安排如下。首先，我们简要介绍ANTLR 4解析器生成器（第2节），并对 $ALL(*)$ 解析策略（第3节）进行详细讨论。接下来，我们将定义谓词文法(*predicated grammar*)、以及其增强转换网络(*ATN augmented transition network*)表示法和前看DFA（第4节）。然后，我们将描述 $ALL(*)$ 语法分析并介绍其解析算法本身（第5节）。最后，我们将证明 $ALL(*)$ 的正确性（第6节）和性能（第7节），以及其相关检测工作（第8节）。附录A提供了 $ALL(*)$ 定理的关键证明，附录B讨论了其语法算法，附录C将提供消除左递归的细节。

## 2. ANTLR 4

ANTLR 4能接收任何不包含直接或间接左递归（间接左递归通过另一条规则调用自身，如：

$A \rightarrow B, B \rightarrow A$ ，另外，但一个空产生式出现时，也会出现左递归，如： $A \rightarrow BA, \rightarrow \epsilon$ ）的上下文无关文法输入序列。ANTLR 4会根据编写的语法，生成一个使用 $ALL(*)$ 产生式预测函数的递归下降解析器。

ANTLR目前具有Java或者C#解析生成器。ANTLR 4的语法类似 $yacc$ ，使用带BNF(EBNF)范式操作符，如克林闭包(Kleene star( $*$ ))和单引号中的词法单元字面量。为方便起见，ANTLR 4的语法同时包含了语法规则和语法规则这样的组合规范。通过使用独立的字符作为输入符号，ANTLR 4的语法可以是弱扫描并且是可组合的，因为 $ALL(*)$ 语法在联合的情况下是封闭的（定理6.2），这种特性有助于Grimm所描述的模块特性（此后，我们将ANTLR 4简称为ANTLR，早期版本我们会额外标注）。

程序员们可以在语法中嵌入用解析器宿主语言（译者注：如使用的是Java的ANTLR 4，就用Java语法）编写的动作（actions，或者成为修改器(mutators)）。这些动作可以获取到当前解析器的状态。不过解析器会在预测的过程中忽略这些动作（修改器），从而尽力避免“踩雷”。通常来说，这些动作的主要目的是从输入流中提取信息然后构建一些数据结构。

ANTLR还支持语义谓词(*semantic predicates*)，这些谓词使用解析器宿主语法编写的不具备副作用的产生布尔值的表达式，用于确定某个特定产生式的语义可达性。如果在评估过程中，一个语义谓词产出为假。那么解析器就会标记这些产生式不可达，从而在解析时改变由编写语法生成的语言<sup>1</sup>。谓词可以检测解析栈以及相关的上下文，来提供一种非标准的上下文有关文法的解析能力，从而大大增强了解析策略的能力。基于先前检测的信息，语义动作和谓词共同作用于解析器，改变整个解析。举个例子，一个C语言语法可以嵌入动作，以便在复杂构造中定义类型符号，如`typedef int i32;`可以在随后的定义中，将类型名称和其他的标识符区分开来，如`i32 x;`。

### 2.1 语法示例

以下用了一个简单的语法片段展示ANTLR的类 $yacc$ 元语言语法：一个以分号结尾的赋值表达式语句

译者注：小写开头的表示语法规则，大写的表示词法规则

```
grammar Ex; // generates class ExParser
// action defines ExParser member: enum_is_keyword
@members {boolean enum_is_keyword = true;}
stat : expr '=' expr ';' // production 1    产生式1
    | expr ';'           // production 2    产生式2
    ;

expr : expr '*' expr
    | expr '+' expr
    | expr '(' expr ')' // f(x) 函数
    | id
    ;

id : ID | {!enum_is_keyword}? 'enum' ;
ID : [A-Za-z]+ ; // match id with upper, lowercase
WS : [ \t\r\n]+ -> skip ; // ignore whitespace
```

代码1. 一个左递归的ANTLR4谓词语法Ex示例

在上面的示例中，有两个特征使得这个语法示例不符合 $LL(*)$ 文法（因此ANTLR3无法识别上述语法定义）。首先，语法规则`expr`是左递归的，ANTLR4会自动重写该语法规则，使其成为非左递归和非二义性的，详见2.2节；其次，`stat`的两个备选产生式都具有一个共同的递归前缀（`expr`），这足以让`stat`从 $LL(*)$ 的角度变得不可判断。ANTLR3会在运行时检测到产生式左侧的递归，然后回退到回溯法进行决策。

词法规则`id`中的谓词`{!enum_is_keyword}`?能够在解析预测的时候根据此谓词允许或禁止`enum`是否能作为一个标识符（译者注：在某些语言中`enum`是保留关键字，上述的语法定义就表示词法规则`id`不能匹配`enum`这样的词法单元）。当谓词为`false`时，解析器只会将`id : ID ;`视为`id`而不会匹配`enum`输入序列，词法会将`enum`作为ID以外的单独词法单元进行匹配。当然，这个例子仅仅用来展示谓词是怎样作用于语法，从而能够描述同一语言的子集或变体。

## 2.2 删除左递归

$ALL(*)$ 解析策略本身不支持左递归，但是ANTLR在生成解析器之前通过语法重写来支持直接左递归。直接左递归涵盖了大多数语法情况，例如我们上面提到的`id`这样的语法规则以及C语言里的声明符。我们从工程设计的角度不支持间接左递归或者隐藏左递归，因为这些形式并不常见，并且消除这样的左递归后，会导致转换的语法呈指数级增长。例如，C11的语法规则包含了大量的直接左递归但却没有间接或隐藏左递归，详见附录2.2。

## 2.3 使用 $ALL(*)$ 进行词法分析

ANTLR使用 $ALL(*)$ 的一种变体来进行词法分析，并且能够对词法单元进行完整匹配，而不是像 $ALL(*)$ 语法解析器那样去预测产生式。在预热之后，词法解析器将会静态地构建一个基于正则表达式的类似DFA。而关键的区别在于 $ALL(*)$ 词法是谓词化的上下文无关文法，而不仅仅是正则表达式，因此可以识别上下文无关的词法单元比如嵌套的注释语法，并且能够根据语义的上下文来控制词法单元的吞吐。这种设计之所以可行，是因为 $ALL(*)$ 在处理词法和语法时足够快。

$ALL(*)$ 也适用于弱扫描的解析方式，因为它具有强大的识别能力，这在处理对上下文敏感的词法问题（如合并C语言和SQL语言）时非常有用。这种合并没有明确的词法哨兵(界限)来划分词法区域：

```
int next = select ID from users where name='Raj'+1;
int from = 1, select = 2;
int x = select * from;
```

概念证明请参见 [19] 中的语法代码/`extras/CSQL`。

## 3. $ALL(*)$ 简介

在本节中，我们将解释 $ALL(*)$ 解析背后的思想和直观感受。然后在第5节中将更加正式地介绍该算法。自上而下解析策略的优势和该策略如何对当前非终结符选择哪个产生式进行展开密切相关。与 $LL(k)$ 和 $LL(*)$ 不同， $ALL(*)$ 解析器会选择第一个有效解析的产生式。因此所有的非左递归语法都是 $ALL(*)$ 。

$ALL(*)$ 解析器不依赖静态语法分析，而是在解析时根据输入的语句进行调整。该解析器会采用类似GLR的机制来分析当前的决策点（因为非终结符往往具有多个备选产生式），并根据需要适当结合当前解析进程的“调用”栈中所有非终结符以及剩余的输入，来查找所有可能有效的决策路径。解析器会为每个决策增量、动态地构建一个前看DFA，并且该DFA记录了其前看序列具体映射到哪个备选产生式。如果到目前为止构建的DFA和当前的前看序列吻合，那么解析器就会跳过分析，并立即对该预测的产生式进行展开。第7节中的实验数据表明， $ALL(*)$ 解析器通常会命中DFA缓存，那么这就对性能产生了极大的影响。

由于 $ALL(*)$ 和确定性的自上而下解析方式的区别仅仅在于预测机制，因此我们可以构建一个传统的递归下降的 $LL$ 解析器，但是会做一个重大的改动。 $ALL(*)$ 解析器会调用一个特殊的预测函数：`adaptivePredict`，该函数会分析语法然后构建前看DFA，而不是简单的将前看序列和静态计算过的符号集进行比较。函数`adaptivePredict`的入参是一个非终结符和解析器调用栈，出参则是备选产生式的序号或者在没有产生式的情况下抛出异常。例如，在2.1节中语法规则`stat`产生了一个如下的解析程序：

```

void stat() {    // 根据语法规则stat进行解析
    switch (adaptivePredicate("stat", call statck)) {
        case 1 :    // 预测产生式1
            expr(); match('='); expr(); match(';');
            break;
        case 2 :    // 预测产生式2
            expr(); match(';');
            break;
    }
}

```

代码2. 语法Ex中规则stat的递归下降代码示例

$ALL(*)$ 进行预测的结构类似于著名的NFA到DFA的子集构造算法。目标是，根据当前的决策，解析器能在看到部分或全部的剩余输入后发现可能达到的状态集合。与子集构造一样， $ALL(*)$ DFA状态也是一个由解析器在匹配一个输入后可能达到的解析器状态集合。然后， $ALL(*)$ 模拟的不是NFA，而是语法上的增强递归转换网络ATN(*augmented recursive transition network*)，因为ATN和语法构造十分接近（ATN看起来就像具有动作和语义谓词的语法图）。因此， $LL(*)$ 的静态分析也是在操作ATN。图1展示了规则stat的ATN子机。

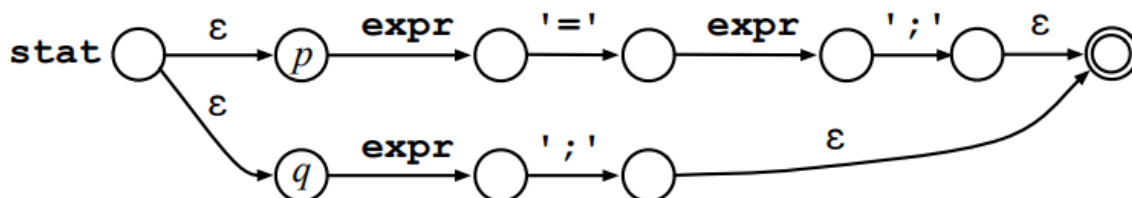


图1. 语法Ex中A的NTLR规则stat的ATN示例

译者注：接下来阅读时要注意区别ATN模拟和DFA模拟这两名词

一个ATN构造(*configuration*)代表了一个子解析器的执行状态，并且指示当前ATN的状态、预测的产生式序号以及ATN子解析器的调用栈：用一个三元组 $(p, i, \gamma)$ <sup>2</sup>表示。构造包含了产生式的序号，因此预测机制能够区分哪一个产生式能够匹配当前的前看序列。与静态 $LL(*)$ 分析不同， $ALL(*)$ 仅仅为它能够看见的前看序列增量地构建DFA而不是所有可能的输入序列。

当解析工作首次达到决策点时，*adaptivePredict*会为该决策点创建一个用起始状态为 $D_0$ 来初始化的前看DFA。 $D_0$ 是ATN子解析器的这样一种构造集合：从每个产生式的左边开始，在不消耗任何输入的情况下都能达到。例如，图1中的非终结符stat的 $D_0$ 构造会首先加入ATN构造 $(p, 1, [])$ 和 $(q, 2, [])$ ，其中 $p$ 、 $q$ 是和产生式1和2左起始相关的ATN状态，并且 $[]$ 表示子解析器此时的调用栈为空（如果stat是起始符号）。

接下来，分析算法会计算出一个新的DFA状态，用于指示ATN在消耗首次前看序列符号后能够到达的位置，并且用标记了该前看符号的边连接这两个DFA状态。紧接着，添加新的DFA状态，直到所有该ATN构造下，新创建的DFA状态都预测相同的产生式： $(-, i, -)$ 。函数*adaptivePredict*将会为该状态标记为接受(*accept*)状态并会为解析器返回对应产生式的序号。图2展示了stat决策在函数*adaptivePredict*分析了输入句子 $x = y;$ 后的前看DFA。该DFA前看的返回不会超过 $=$ 符号，因为一个 $=$ 符号就足够区分出expr产生式了（注：1表示“预测产生式1”）。

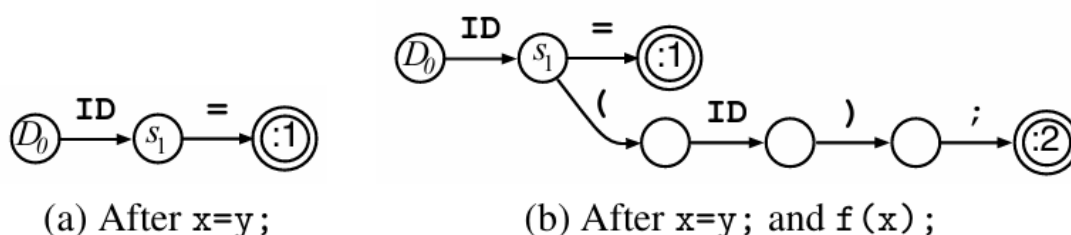


图2. 决策stat的预测DFA



在典型情况下，函数 $adaptivePredict$ 能够为特定的决策找到一个已存在的DFA。解析的目标是找到或者是构建一条能够通过DFA达到可接受状态的路径。如果 $adaptivePredict$ 达到了一个（非接受 $non - accept$ ）的DFA状态，即无法为当前的前看符号找到能够指向下一个DFA状态的边，它就会恢复到ATN模拟来展开DFA（无需从头开始）。例如，为解析 $stat$ 的接下来的输入： $f(x);$ ，函数 $adaptivePredict$ 将会为 $D_0$ 找到一条标有ID、通向DFA状态 $s_1$ 的边，在无需ATN模拟的情况下，沿着这条边转化为DFA状态 $s_1$ ，此时由于在状态 $s_1$ 的视角下，沿着左括号向前已经没有新的DFA状态了，分析算法就会产出一条可接受状态的路径，也就是直达产生式2，如图2中的(b)所示。注意，由于在输入序列ID(ID)的情况下，会同时预测产生式1和2，所以在这种情况下，分析算法会一直持续，直到DFA沿着=号和;号前进。

如果ATN模拟为某个输入计算出的新状态已经在DFA中存在，模拟工作将会添加一条指向那个已存在DFA状态的边，然后从那个已存在DFA状态开始，切换回DFA模拟。指向一个已存在的DFA状态则是DFA中一个递归循环的表现形式。在经验上，通过拓展DFA来处理不常见的输入短语，能够降低后面使用ATN模拟的可能性，从而提高解析速度（第7节）。

### 3.1 预测工作和解析调用栈息息相关

标题的原文是：Predictions sensitive to the call stack

解析器无法永远依赖于前看DFA来做正确的决策。为处理所有可能的非左递归语法，在预测开始时， $ALL(*)$ 必须在某些情况下考虑解析器调用栈(parser call stack)的可用性（在第5章中，调用栈用 $\gamma_0$ 表示）。为了说明栈敏感(stack-sensitive)预测的必要性，考虑如下场景：假设我们正在对Java语法的方法定义进行产生式预测：究竟是识别为接口的方法定义还是类的方法定义呢（在Java中，接口定义的方法不能具有方法体）。以下是一个简化的语法，用于展示非终结符 $A$ 中的一个栈敏感决策：

$$S \rightarrow xB|yC \quad B \rightarrow Aa \quad C \rightarrow Aba \quad A \rightarrow b|\epsilon$$

如果此时的待解析输入序列是： $ba$ ，在没有解析调用栈的情况下，即使有再多的前看也无法确定非终结符 $A$ 应该用哪种产生式来进行匹配：如果要将此序列匹配成非终结符 $B$ ，则可以用产生式 $A \rightarrow b$ 来匹配将非终结符 $A$ ，而如果要匹配非终结符 $C$ ，则可以用产生式 $A \rightarrow \epsilon$ 来匹配非终结符 $A$ ，如果不用解析调用栈进行辅助，这就是解析 $ba$ 时发生的二义性冲突。

忽略解析调用栈进行产生式预测的解析器称为 $Strong LL(SLL)$ 解析器。例如由程序员手工构建的递归下降解析器就属于 $SLL$ 解析器一类。按照惯例，文献将 $SLL$ 成为 $LL$ ，但我们将这两个术语区分开来，因为如果要处理所有可能的语法还是得“真” $LL$ 解析器来做（译者注：这里应该指的是 $SLL$ ）。刚才举的例子实际上对应 $LL(2)$ 语法，不是能够处理任意前看 $k$ 的 $SLL(k)$ 语法，尽管在每个调用点复制 $A$ 都会使语法成为 $SLL(2)$ 。

为每个可能的解析器创建不同的前看DFA是不可能的，因为要创建调用栈的排列组合数量和其深度称指数关系。相反，我们利用大多数决策对调用栈不敏感这一事实，忽略调用栈来构建前看DFA。如果 $SLL$  ATN模拟发现存在产生式预测冲突（第5.3节），它是无法确定这个前看短语究竟是二义性本身的还是说可以用栈敏感来解决。在这种情况下，函数 $adaptivePredict$ 必须重新利用解析调用栈 $\gamma_0$ 来重新检测这个前看短语。这种混合或称之为优化的 $LL$ 模式能够尽可能地在当前看DFA中缓存栈敏感的产生式预测结果，来提高解析性能，同时保留栈敏感方式对产生式进行预测的全部功能。优化的 $LL$ 模式适用于每次产生式决策，但接下来介绍的两段( $two-stage$ )解析法，通常能够完全避免 $LL$ 模拟（接下来我们用 $SLL$ 来表示对调用栈不敏感的解析， $LL$ 表示对调用栈敏感的解析）。

### 3.2 两段 $ALL(*)$ 解析法

$SLL$ 的解析能力比 $LL$ 弱，但是速度更快。但由于我们发现实践中大多数情况还是 $SLL$ 决策，因此尝试使用“仅使用 $SLL$ 模式”解析整个输入是具有意义的，这也是两段 $ALL(*)$ 解析算法的第一阶段。但是，如果 $SLL$ 模式下发现了一个语法错误，这种错误可能是由于 $SLL(*)$ 不具备栈敏感能力所产生的，当然也可能是一个真正的语法错误，所以必须使用优化的 $LL$ 模式，即第二阶段，来重试对整个输入的解析。与单阶段的优化的 $LL$ 解析模式相比，这种可能会对整个输入进行两次解析的反直觉策略反而能够提高解析速度。例如，在解析一个123M的Java源代码库时（第7节），这种两段解析模式比单段优化的 $LL$ 模式快8倍。不过两段解析策略是建立在 $SLL$ 要么表现得和 $LL$ 一样，要么则产生语法错误这一事实上的（定理6.5）。对于无效

的句子，无论解析器选择哪个产生式，都无法对输入进行推导。而对于有效的句子，*SLL*会像*LL*那样选择产生式进行推导，或者选择能够最终产生一个语法错误的产生式（*LL*认为该产生式是无效的）。即使存在二义性，*SLL*也能常常像*LL*那样去解决冲突。例如，尽管我们的Java语法存在一些二义性，*SLL*模式仍然能够正确地解析我们尝试过的所有输入，而不是回退到*LL*。不过，为了确保解析的正确性，必须保留第二阶段的*LL*模式。

## 4. 谓词(predicated)语法：ATN和DFA

在正式定义*ALL*(\*)解析之前，我们首先需要回顾一些背景资料，特别是谓词文法：ATN和前看DFA。

### 4.1 谓词语法

为正式定义*ALL*(\*)解析，我们首先需要正式定义该语法源自的谓词文法。一个谓词文法  $G = (N, T, P, S, \Pi, \mathcal{M})$  具有如下的元素：

- $N$ 是非终结符集合（规则名词）
- $T$ 是终结符集合（词法单元）
- $P$ 是产生式集合
- $S \in N$ 是起始符号
- $\Pi$ 是无副作用的语义谓词(semantic predicates)集合
- $\mathcal{M}$ 是动作集合（修改器）

谓词*ALL*(\*)文法和*LL*(\*)文法的区别仅在于*ALL*(\*)不再需要或者说不再支持语法谓词（syntactic predicates）。本文正式章节中的谓词文法使用以下标注来表示：

$A \in N$	非终结符号		
$a, b, c, d \in T$	终结符号		
$X \in (N \cup T)$	产生式元素		
$\alpha, \beta, \delta \in X^*$	文法符号序列		
$u, v, w, x, y \in T^*$	终结符号序列		
$\$$	文件末尾”符号”	$\epsilon$	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		
$A \rightarrow \{\pi_i\}?\alpha_i$	基于语义( <i>semantics</i> )谓词判断后的 $i^{th}$ (第 $i$ 个)产生式		
$A \rightarrow \{\mu_i\}$	具有修改器( <i>mutators</i> )的产生式		

谓词文法的具体含义可以用下面的推导(derivation)规则来定义：

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

$$Sem \frac{\pi(\mathbb{S}) \quad A \rightarrow \{\pi\}?\alpha}{(\mathbb{S}, uA\delta) \Rightarrow (\mathbb{S}, u\alpha\delta)} \quad (2)$$

$$Action \frac{A \rightarrow \{\mu\}}{(\mathbb{S}, uA\delta) \Rightarrow (\mu(\mathbb{S}), u\delta)} \quad (3)$$

$$Closure \frac{(\mathbb{S}, \alpha) \Rightarrow (\mathbb{S}', \alpha'), (\mathbb{S}', \alpha') \Rightarrow *(\mathbb{S}'', \beta)}{(\mathbb{S}, \alpha) \Rightarrow *(\mathbb{S}'', \beta)} \quad (4)$$

### 公式1. 谓词文法的最左推导公式

为支持语义谓词和修改器，这些规则引用了状态 $\mathbb{S}$ ，它表示在解析过程中抽象的用户状态。判别式 $(\mathbb{S}, \alpha) \Rightarrow (\mathbb{S}', \beta)$ 可以理解为：在机器状态 $\mathbb{S}$ ，语法序列 $\alpha$ 会在紧接着的一步后归约为机器状态 $\mathbb{S}'$ 和语法序列 $\beta$ 。而判别式 $(\mathbb{S}, \alpha) \Rightarrow *(\mathbb{S}', \beta)$ 则表示要重复若干次这样的一步规则规则。这些归约规则指定了最左推导原则。只有当前状态 $\mathbb{S}$ 的 $\pi_i$ 为 `true` 的时候，其带有语义谓词 $\pi_i$ 的产生式的才是有效的。最后，一个动作产生式可以使用指定的修改器 $\mu_i$ 来更新该状态。

正式地，语法序列 $\alpha$ 在用户态 $\mathbb{S}$ 生成的语言是： $L(\mathbb{S}, \alpha) = \{w | (\mathbb{S}, \alpha) \Rightarrow *(\mathbb{S}', w)\}$ ，以及文法 $G$ 的语言是： $L(\mathbb{S}_0, G) = \{w | (\mathbb{S}_0, S) \Rightarrow *(\mathbb{S}, w)\}$  ( $\mathbb{S}_0$ 可以为空)。如果 $\mu$ 是 $w$ 的前缀或者等于 $w$ ，我们就写作 $\mu \preceq w$ 。如果 $L$ 存在一个 $ALL(*)$ 的语法，那么 $L$ 就是 $ALL(*)$ 的。理论上， $L(G)$ 的语言类别是有限递归的，因为每个修改器都可以是一个图灵机。在现实中，语法编写者不会依赖该通用性，所以标准做法是考虑将此类语言归类为上下文有关的：因为谓词在此过程作用时，可以检查从左到右的推导过程中产生的调用栈以及终结符的情况，故而这类语言应该是上下文有关的而不是无关的文法。

以上的形式体系还有很多语法上的限制并，但并不存在实际的ANTLR语法中，例如强制修改器只能作用在各自的产生式规则上、禁止使用常见的拓展BNF(EBNF)符号，如 $\alpha^*$ 和 $\alpha^+$ 闭包。我们可以做出这些限制，而不丢失其通用性，因为任何的一般语法都可以转换为这种更加严格的受限形式。

## 4.2 解决二义性

二义性语法指的是对于同一个输入序列，可以识别出多个产生式。公式1中的规则并不具备排除二义性的能力。但是，对于一个实用性的编程语言解析器，每个输入都应该对应一个唯一的解析方式。

1. ANTLR以前的版本支持语法谓词(*syntactic predicates*)，以消除能够导致静态语法分析失败的二义性；但由于 $ALL(*)$ 具有动态解析的能力，故ANTLR4不需要这一功能 [↩](#)

2. 组件 $i$ 并不存在于GLL、GLR或Early的解析机构造中 [↩](#)