自适应 *LL(*)* 解析：动态分析的力量

特伦斯-帕尔
旧金山大学
parrt@cs.usfca.edu

山姆-哈威尔
德克萨斯大学奥斯汀分校
samharwell@utexas.edu

凯瑟琳-费舍尔
塔夫茨大学
kfisher@eecs.tufts.edu

摘要

尽管 PEG、*LL(*)*、GLR 和 GLL 等现代解析策略取得了进步，但解析问题仍未得到解决。现有的方法存在许多缺陷，包括难以支持副作用嵌入式操作、性能缓慢和/或不可预测，以及匹配策略有悖直觉等。本文介绍的 *ALL(*)* 解析策略结合了传统自上而下 *LL(k)* 解析器的简单、高效和可预测性，以及类似 GLR 机制的强大解析决策功能。关键的创新之处在于将语法分析移到了解析时间，这使得 *ALL(*)* 可以处理任何非左递归上下文自由语法。从理论上讲，*ALL(*)* 的速度为 $O(n^4)$，但在实际语法分析中却始终保持线性增长，比 GLL 和 GLR 等一般策略的性能高出数个数量级。ANTLR 4 可生成 *ALL(*)* 解析器，并通过语法重写支持直接左递归。ANTLR 4 的广泛使用（2013 年的下载量为 5000 次/月）证明，*ALL(*)* 对各种应用都很有效。

1. 导言

尽管现代语法分析策略已经非常成熟，而且学术研究历史悠久，但计算机语言解析在实践中仍然不是一个已经解决的问题。当机器资源稀缺时，迫使程序员修改语法以适应确定性 *LALR*(*k*) 或 *LL*(*k*) 解析器生成器的约束是合理的。[1]随着机器资源的增长，再搜索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，这些策略同时采用了 "自下而上"（*LR* 风格）和 "自上而下"（*LL 风格）的*方法。这些策略包括 GLR [26]、解析器表达语法 (PEG) [9]、*LL(*)*

[20] 来自 ANTLR 3，以及最近的 GLL [25]，一种完全通用的自顶向下策略。

虽然这些新策略比 *LALR*(*k*) 和 *LL*(*k*) 剖析器生成器更容易使用，但它们也存在各种缺陷。首先，非确定性解析器有时会出现意想不到的行为。GLL 和 GLR 会返回含混语法的多重解析树（森林），因为它们是为处理自然语言语法而设计的，而自然语言语法通常是有意含混的。对于计算机语言来说，模糊性几乎总是一个错误。当然，我们也可以对构建的解析森林进行漫步来消除歧义，但这种方法需要花费额外的时间、空间和机器来处理不常见的情况。

[1] 我们使用 "*确定性*"一词，是指确定性有限自动机（*DFA*）与非确定性有限自动机（*NFA*）的不同之处：下一个或多个符号唯一地决定着动作。

根据定义，PEG 是无歧义的，但有一个怪癖，即规则 $A \rightarrow a \mid ab$（意思是 "A 要么匹配 a 要么匹配 ab"）永远无法匹配 ab，因为 PEG 会选择第一个匹配的选项的前缀。嵌套回溯使得消除 PEG 的错误变得困难。

其次，在任何持续推测（PEG）或支持对输入进行多重解释（GLL 和 GLR）的策略中，都应避免打印语句等由程序员提供的副作用（muta-tors），因为此类操作可能从未真正发生过[17]。(尽管 DParser [24] 支持 "最终 " 操作，当程序员确定缩减是明确的最终解析的一部分时）。如果没有副作用，Ac- tions 必须在不可变数据结构中缓冲所有解释的数据，或者提供撤销操作。前一种机制受到内存大小的限制，而后一种机制并不总是容易或可行的。避免突变器的典型方法是构建一棵解析树，用于解析后处理，但这种人工制品从根本上限制了对输入文件的解析，因为这些文件的解析树只能放在内存中。构建解析树的解析器无法分析大型数据文件或无限的数据流（如网络流量），除非它们可以按逻辑块进行处理。

第三，我们的实验（第 7 节）表明，GLL 和 GLR 在时间和空间上都很慢，而且不可预测。它们的复杂度分别为 $O(n^3)$ 和 $O(n^{p+1})$，其中 p 是语法中最长句子的长度[14]。(GLR 通常被引用为 $O(n^3)$ 是因为 Kipps [15] 给出了这样一个算法，尽管常数高得不切实际）。从理论上讲，一般解析器处理确定性语法应接近线性时间。在实践中，我们发现 GLL 和 GLR 在 12 920 个 Java 6 库源文件（123M）的语料库中，*ALL(*)* 的速度比 *ALL(*)* 慢 ˜135 倍；在一个 3.2M 的 Java 文件中，*ALL(*)* 的速度比 *ALL(*)* 慢 6 个数量级。

LL(*) 通过提供一种主要是去终结的解析策略来解决这些缺陷，这种策略使用正则表达式（代表为确定性有限自动机（DFA））来检查整个剩余输入，而不是 *LL(k)* 的固定 k 序列。尽管前瞻语言（所有可能的剩余输入短语的集合）通常是无上下文的，但使用 DFA 进行前瞻也会将 *LL(*)* 的决策限制在用常规前瞻语言区分备选方案上。但主要问题在于，*LL(*)* 语法条件在静态上是不可判定的，语法分析有时会找不到能区分替代结果的常规表达式。ANTLR 3 的静态分析能检测并避免潜在的不可判定情况，从而避免了反向跟踪分析的失败。这使得 LL(*) 具有与 PEG 相同的 a | ab 特性。这使得 *LL(*)* 具有与 PEG 相同的 $a \mid ab$ 特性

对于这样的决策来说。选择第一个匹配选项的回溯决策也无法检测到明显的歧义，如 $A \rightarrow \alpha \mid \alpha$，其中 α 是语法符号序列。
使 $\alpha \mid \alpha$ 非 LL*(*)*。

1.1 动态语法分析

在本文中，我们介绍了自适应 *LL(*)* 或 *ALL(*)* 剖析器，它将确定性自上而下剖析器的简单性与类似于 GLR 的强大机制结合起来，以做出剖析决策。具体来说，LL 分析会在每个预测决策点（非终端）暂停，然后在预词典机制选择了适当的扩展后恢复。这一创新的关键在于将语法分析转移到了解析时间；无需进行静态语法分析。这一选择让我们避免了静态 *LL(*)* 语法分析的不可判定性，让我们可以为任何非左递归无上下文语法 (CFG) 生成正确的解析器（定理 6.1）。静态分析必须考虑所有可能的输入序列，而动态分析只需考虑实际看到的输入序列的有限集合。

*ALL(*)*预测机制背后的理念是在决策点启动子解析器，每个备选方案都有一个子解析器。子解析器以伪并行方式运行，探索所有可能的路径。当子解析器的路径与剩余输入不匹配时，子解析器就会死亡。子解析器在输入过程中同步前进，这样分析就能在最小前瞻深度上识别出唯一的幸存者，从而唯一预测生产。如果多个子解析器聚集在一起或到达文件末尾，预测器就会宣布出现歧义，并以与存活的子解析器相关的最低生产编号来解决。(产品编号表示优先级，是自动解决 PEG 等歧义的一种方法；Bi-son 还通过选择首先指定的产品来解决冲突）。程序员还可以嵌入*语义*谓词[22]，在模棱两可的解释中做出选择。

ALL(*) 分析器将分析结果记忆化，以增量和动态的方式建立一个 DFA 缓存，将前瞻性短语映射到预测的产品。(我们使用 "*分析*"一词的意思是，*ALL(*)* 分析会像静态 *LL(*) 分析*一样产生前瞻性 DFA）。解析器可以通过查阅缓存，在相同的解析器决策和前瞻短语中快速做出未来预测。不熟悉的输入短语会触发语法分析机制，同时预测一个替代方案并更新 DFA。DFA 适合记录预测结果，尽管在给定决策下的前瞻语言通常是无上下文语言。动态分析只需考虑解析过程中遇到的有限无语境语言子集，而且任何有限子集都是有规律的。

为了避免非确定子解析器的指数性质，预测使用了*图结构堆栈*（GSS）[25]，以避免冗余计算。GLR 使用的策略与此基本相同，只是 *ALL(*)* 只预测使用此类子解析器的产品，而 GLR 则实际使用它们进行解析。因此，GLR 必须将终端推入 GSS，而 *ALL(*) 则*不需要。

ALL(*) 解析器以 *LL* 的简单性处理匹配终端和扩展非终端的任务，但其理论时间复杂度为 $O(n^4)$（定理 6.3），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成本为 $O(n^2)$。$O(n^4)$ 与 GLR 的复杂度一致。在第 7 节中，我们通过经验证明，常见语言的 *ALL(*)* 解析器是高效的，并且在实践中表现出线性行为。

ALL(*) 的优势在于将语法分析移至解析时间，但这一选择给语法功能测试带来了额外的负担。与所有动态方法一样，程序员必须覆盖尽可能多的语法位置和输入序列组合，以发现语法歧义。标准的源代码覆盖工具可以帮助程序员测量 *ALL(*)* 解析器的语法覆盖率。生成代码中的高覆盖率与高语法覆盖率相对应。

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（源代码、二进制文件或 ANTLRworks2 开发环境，使用唯一 IP 地址计算网络日志中的非机器人条目以获得下限）。这些活动证明 *ALL(*)* 是有用和可用的。

本文的其余部分安排如下。首先，我们介绍了 ANTLR 4 解析器生成器（第 2 节），并讨论了 *ALL(*)* 解析策略（第 3 节）。接下来，我们将定义*谓词语法*、其*增强转换网络*表示法和前瞻性 DFA（第 4 节）。然后，我们将描述 *ALL(*)* 语法分析并介绍解析算法本身（第 5 节）。最后，我们将证明 ALL(***) 的正确性（第 6 节）和效率（第 7 节），并考察相关工作（第 8 节）。附录 A 有 *ALL(*)* 关键定理的证明，附录 B 讨论了算法语法，附录 C 有消除左递归的细节。

2. ANTLR 4

ANTLR 4 接受任何不包含间接或隐藏左递归的无上下文语法作为输入。[2]ANTLR 4 会根据语法生成一个递归后裔解析器，该解析器使用 *ALL(*)* 生成预测函数（第 3 节）。ANTLR 目前使用 Java 或 C# 生成解析器。ANTLR 4 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量。为方便起见，语法包含词法规则和语法规则的组合规范。ANTLR 4 可根据组合规范生成词法和解析器。通过使用单个字符作为输入符号，ANTLR 4 语法可以是无扫描的和可组合的，因为 *ALL(*)* 语言在联合下是封闭的（定理 6.2），从而提供了以下好处

[2] 间接左递归规则通过另一条规则调用自己，例如：$A \rightarrow B, B \rightarrow A$. 当空生产暴露出左递归时，就会出现隐藏左递归，例如：$A \rightarrow BA, B \rightarrow \epsilon$。

Grimm [10] 所描述的模块性。(此后，我们将 ANTLR 4 称为 ANTLR，并明确标注早期版本）。

程序员可以在语法中嵌入用解析器宿主语言编写的副作用动作（*突变器*）。这些动作可以访问解析器的当前状态。解析器在推测过程中会忽略突变体，以防止动作推测性地 "发射导弹"。动作通常从输入流中提取信息并创建数据结构。

ANTLR 还支持*语义谓词*（*semantic predicates*），这些谓词是用宿主语言编写的无副作用布尔值表达式，用于确定特定语法的语义可行性。语义谓词如果在解析过程中评估为假，就会导致周围的语句不可行，从而在解析时改变语法生成的语言。[3]谓词可以检查解析堆栈和周围的输入上下文，从而提供一种非正式的上下文敏感解析能力，因此大大提高了解析策略的强度。语义操作和谓词通常共同作用，根据先前发现的信息改变解析。例如，一个 C 语言语法可以包含嵌入式动作，以便从上下文中定义类型符号。
结构体，如 `typedef int i32;`，以及用于区分
在随后的定义中，将类型名称与其他标识符区分开来，如 `i32 x;`.

2.1 语法示例

图 1 展示了 ANTLRs yacc-like 金属语言，给出了一种简单编程语言的语法，该语法包含以分号结束的符号和表达式语句。有两个语法特征使得该语法不符合 *LL(*)*，因此 ANTLR 3 无法接受。首先，规则 `expr` 是左递归的。ANTLR 4 会自动重写该规则，使其成为非左递归和无歧义的规则，详见第 2.2 节。其次，`stat` 规则的备选生成物有一个共同的递归前缀 (`expr`)，这足以让 `stat` 从 *LL(*)* 的角度变得不可判定。ANTLR 3 会检测生产左侧边的递归，并在运行时退回到回溯决策。

规则 `id` 中的谓词 `{!`
的预测时刻。当谓词为假时，解析器只将 `id` 视为 `id : ID;` 不允许将 `enum` 视为 `id`，因为词法词典会将 `enum` 作为 `ID` 以外的单独标记进行匹配。这个例子展示了谓词是如何让一个语法去描述同一种语言的子集或变体的。

2.2 删除左递归

ALL(*) 解析策略本身不支持左向递归，但 ANTLR 在生成解析器之前通过语法重写支持直接左向递归。直接左递归涵盖了最常见的情况，例如算术表达式的前
id，以及 C 声明符。我们在工程设计上决定不支持间接或隐藏的左递归

[3] ANTLR 以前的版本支持*语法谓词*，以消除静态语法分析失败的歧义；由于 *ALL(*)* 的动态分析，*ANTLR4* 中不需要这一功能。

```
grammar Ex; // 生成类 ExParser
// 操作定义 ExParser 成员：enum_is_keyword @members
{boolean enum_is_keyword = true;}
stat: expr '=' expr ';' // production 1
           | expr ';'          // 生产 2
    ;
expr: expr '*' expr
     | expr '+' expr
    | expr '(' expr ')'// f(x)
    | id
    ;
id : ID | {!enum_is_keyword}?枚举' ;
ID : [A-Za-z]+ ; // 用大、小写匹配 ID WS : [ \\tr\n]+
-> skip ; // 忽略空格
```

图 1.左递归 ANTLR 4 谓语语法 Ex 样本

因为这些形式并不常见，而且删除所有左递归会导致转换后的语法呈指数级增长。例如，C11 语言规范语法包含大量直接左递归，但没有间接或隐藏递归。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，它能完全匹配标记，而不是像 *ALL(*)* 分析程序那样只预测生成。预热后，词法生成器将建立一个 DFA，类似于 lex 等基于正则表达式的工具静态生成的 DFA。关键的区别在于，*ALL(*) 词法*是谓词化的无上下文语法，而不仅仅是正则表达式，因此它们可以识别无上下文的标记，例如嵌套连接词，并能根据语义上下文来控制标记的进出。这种设计之所以可行，是因为 *ALL(*)* 的速度足以处理词法和解析。

ALL(*) 也适用于无扫描器解析，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;
```

概念验证请参见 [19] 中的语法代码/extras/CSQL。

3. *ALL(*)* 解析简介

在本节中，我们将解释 *ALL(*)* 解析背后的想法和直觉。然后，第 5 节将更正式地介绍该算法。自上而下解析策略的优势与该策略如何选择对当前非词进行扩展有关。与 *LL(k)* 和 *LL(*) 剖析*器不同，*ALL(*)* 剖析器总是选择导致有效解析的第一个备选方案。因此，所有非左递归语法都是 *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句子进行调整。该解析器使用类似 GLR 的机制分析当前的决策点（有多个生成词的非终结词），以探索与当前 "调用 "堆栈中的在进程非终结词和剩余非终结词相关的所有可能决策路径。

```
void stat () { //parseacordingtorulestat
  switch ( adaptive Predict (" stat ", call stack)) {
    case 1 : //predictproduction1
      expr (); match (' ='); expr (); match (';');
      break ;
      情况 2 : //predictproduction2
      expr (); match (';'); break ;
  }
}
```

图 2.语法 Ex 中 stat 的递归-后裔代码

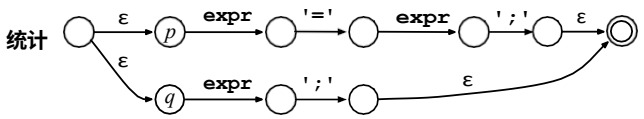


图 3.语法 Ex 中 ANTLR 规则 stat 的 ATN

*按需*输入。解析器会为每个决策增量、动态地构建一个前瞻性 DFA，记录从前瞻性序列到预测生产数量的映射。如果迄今为止构建的 DFA 与当前的前瞻性匹配，解析器就可以跳过分析，立即展开预测的备选方案。第 7 节中的实验表明，*ALL(*)* 分析程序通常会获得 DFA 缓存命中，而且 DFA 对性能至关重要。

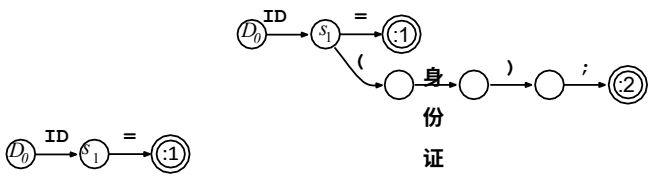
由于 *ALL(*)* 与确定性自上而下方法的区别仅在于预测机制，因此我们可以构建常规递归后裔 *LL* 解析器，但有一个重要的转折。*ALL(*)* 解析器会调用一个特殊的预测函数 *adap- tivePredict*，该函数会分析语法以构建前瞻 DFA，而不是简单地将前瞻与静态计算的标记集进行比较。函数 *adaptivePredict* 将非词法和解析器调用堆栈作为参数，并返回预测的生成数，如果没有生成数，则抛出异常。

可行的生产。例如
第 2.1 节产生了与图 2 类似的解析程序。

ALL(*) 预测的结构类似于著名的 NFA 到DFA *子集构建*算法。我们的目标是，相对于当前决定，分析器在看到部分或全部剩余输入后可能达到的状态集。与子集构造一样，*ALL(*)* DFA 状态是在匹配导致该状态的输入后可能出现的解析器配置的集合。然而，ALL(**)* 模拟的不是 NFA，而是语法的*增强递归转换网*（ATN）[27] 表示的动作，因为 ATN 与语法结构非常接近。(ATN 看起来就像同步图，可以有动作和语义谓词）。出于同样的原因，*LL(*)* 的静态分析也在 ATN 上运行。图 3 显示了规则统计的 ATN 子机器。

*ATN 配置*表示子解析器的执行状态，并跟踪 ATN 状态、预测生产编号和 ATN 子解析器调用堆栈：元组（*p*、*i*、*γ*）。[4]配置包括生产编号，因此预测可以确定*哪个*生产与当前的前瞻匹配。与静态 *LL(*)* 分析不同，*ALL(*)* 只考虑它所看到的前瞻序列，而不是所有可能的序列，以增量方式构建 DFA。

[4] 组件 *i* 不存在于 GLL、GLR 或 Earley [8] 的机器配置中。



(a) x=y 之后；　　(b) x=y 之后；以及 f(x)；

图 4.用于决策统计的预测 DFA

当解析首次到达某个判定时，*adap-tivePredict* 会通过创建 DFA 开始状态 D_0 来初始化该判定的前瞻 DFA。D_0 是 ATN 子解析器配置的集合，在不消耗输入符号的情况下，可以到达每个生产左边缘。例如，为图 3 中的非终端 stat 构建 D_0 时，首先要添加 ATN 配置（*p,* 1*,* []）和（*q,* 2*, []*），其中 *p* 和 *q 是分别*对应于产成 1 和 2 左边缘的 ATN 状态，而 [] 则是空的子解析器调用堆栈（如果 stat 是起始符号）。

接下来，分析计算出一个新的 DFA 状态，表明 ATN 模拟在消耗第一个 looka- head 符号后可能到达的位置，然后用一条标有该符号的边连接两个 DFA 状态。分析继续进行，不断添加新的 DFA 状态，直到新创建的 DFA 状态预测相同的结果：（-, *i*, -）。函数 *自适应预测*将该状态标记为接受状态，并返回解析器的生产编号。图 4a 显示了 *adaptivePredict 对*输入句 x=y；进行分析后的决策统计前瞻 DFA。DFA 不会超出 = 的范围，因为 = 足以唯一区分 expr 的生成。(符号 :1 表示 "预测结果 1"）。

在典型情况下，*adaptivePredict* 会为特定决策找到现有的 DFA。其目标是通过 DFA 找到或建立一条通往接受状态的路径。如果 *adaptivePredict* 到达的（非接受）DFA 状态没有当前 looka- head 符号的边，它就会返回到 ATN 模拟，以扩展 DFA（无需重新输入）。例如，为了分析第二个输入短语 stat，如 f(x);，*adaptivePredict 会*找到一个从 D_0 的现有 ID 边沿跳转到 s_1，不带 ATN 模拟。由于没有来自 s_1 的用于左侧父子关系的现有边，因此分析模拟 ATN 完成了一条通往接受状态的路径，该路径预测了第二个生成结果，如图 4b 所示。请注意，由于序列 ID(ID) 预测了两个生成结果，因此分析将继续进行，直到 DFA 有了用于两个生成结果的边为止。= 和 ; 符号。

如果 ATN 仿真计算出的新目标状态已经存在于 DFA 中，则仿真会添加一条新的边，目标是正在消失的状态，并从该状态开始切换回 DFA 仿真模式。以现有状态为目标就是循环在 DFA 中出现的方式。根据经验，扩展 DFA 以处理不熟悉的短语会降低未来 ATN 模拟的可能性，从而提高解析速度（第 7 节）。

3.1　对调用堆栈敏感的预测

解析器不能总是依靠前瞻性 DFA 来做出正确的决定。为了处理所有非左递归语法，*ALL(*)* 预测必须偶尔考虑预测开始时可用的解析器调用栈（在第 5 章中表示为 γ_0）。为了说明堆栈敏感预测的必要性，可以考虑在识别 Java 方法定义时进行的预测可能取决于该方法是否被定义为

接口或类定义中的方法（Java 接口方法不能有主体）。(下面是一个简化语法，在非终端 A 中显示了堆栈敏感判定：

$$S \to xB \mid yC \quad B \to Aa \quad C \to Aba \quad A \to b \mid \epsilon$$

如果没有解析器堆栈，再多的前瞻性也无法独特地区分 A 的制作。当 B 调用 A 时，head ba 预测 $A \to b$，但当 C 调用 A 时，head ba 预测 $A \to \epsilon$。
如果预测忽略了解析器调用堆栈，那么就会有 ba 时发生预测冲突。

忽略解析器调用栈进行预测的解析器称为*强 LL*（*SLL*）解析器。程序员手工创建的递归后裔解析器属于 *SLL* 类。按照惯例，文献将 *SLL* 称为 *LL*，但我们将这两个术语区分开来，因为处理所有语法都需要 "真正的 "*LL*。上述语法是 $LL(2)$语法，但对于任意 k *而言都*不是 $SLL(k)$ 语法，尽管在每个调用点复制 A *会*使语法成为 $SLL(2)$语法。

为每个可能的解析器创建不同的前瞻性 DFA 由于堆栈排列的次数与堆栈深度成指数关系，因此忽略调用堆栈是不可行的。相反，我们利用大多数决策对堆栈不敏感这一事实，忽略解析器调用堆栈来构建前瞻性 DFA。如果 *SLL* ATN 模拟发现预测冲突（第 5.3 节），则无法确定前瞻短语是模棱两可还是对堆栈敏感。在这种情况下，*adaptivePredict* 必须使用解析器堆栈 γ_0 重新检查 lookahead。这种混合或*优化的* LL *模式*通过在可能的情况下在 lookahead DFA 中缓存对堆栈不敏感的预测结果来提高性能，同时保留全部对堆栈敏感的预测能力。优化的 *LL* 模式适用于每次决策，但接下来介绍的*两阶段解析*通常可以完全避免 *LL* 模拟。(以下我们用 *SLL* 表示对堆栈不敏感的解析，用 *LL* 表示对堆栈敏感的解析）。

3.2 两阶段 *ALL(*)* 解析

SLL 比 *LL* 弱，但速度更快。由于我们发现在实践中大多数决定都是 *SLL* 决定，因此尝试在 "仅 *SLL* 模式 "下解析整个输入是有意义的，这也是两阶段 *ALL(*)* 解析算法的第一阶段。但是，如果 *SLL* 模式发现了语法错误，它可能发现了 *SLL* 弱项或真正的语法错误，因此我们必须使用优化的 *LL* 模式（即第二阶段）重试整个输入。与单级优化 *LL* *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能显著提高速度。例如，在解析 123M 的语料库时，使用 Java 语法（第 7 节）进行的两阶段解析比单阶段优化 *LL* 模式快 8 倍。两阶段策略依赖于这样一个事实，即 *SLL* 要么表现得像 *LL*，要么会出现语法错误（定理 6.5）。对于无效句子，无论解析器如何选择生成物，都无法对输入进行推导。对于有效句子，*SLL* *会*像 *LL* 一样选择构词法，或者选择最终导致语法错误的构词法（*LL* 认为该选择不可行）。即使存在歧义，*SLL* 也经常像 *LL* 一样解决冲突。例如，尽管我们的 Java 语法中存在一些歧义，*SLL* 模式仍能正确地解析我们尝试过的所有输入，而不会失败到

LL.不过，为了确保正确性，必须保留第二阶段（*LL*）。

4. 专用语法、ATN 和 DFA

为了使 *ALL(*)* 解析正规化，我们首先需要回顾一些基础材料，特别是谓词语法、ATN 和 Lookahead DFA 的正规定义。

4.1 谓语语法

要使 *ALL(*)* 解析正规化，我们首先需要正式定义谓词语法，并从中衍生出 *ALL(*)* 解析。一个谓词语法 $G = (N, T, P, S, \Pi, M)$ 有以下元素：

- 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）

谓词 *ALL(*)* 语法与 *LL(*)* [20] 语法的区别仅在于 *ALL(*)* 语法不需要或不支持句法谓词。本文正式章节中的谓词语法使用图 5 所示的符号。图 6 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，这些规则引用了

状态 S，它在解析过程中抽象了用户状态。判断形式 $(S, \alpha) \Rightarrow (S', \beta)$ 可以理解为"在机器状态 S 中，语法序列 α 一步还原为修正状态 S' 和语法序列 β"。判断 $(S, \alpha) \Rightarrow^* (S', \beta)$ 表示一步还原规则的重复应用。

这些还原规则指定了最左侧的派生。只有当当前状态 S 的 π_i 为真时，带有语义谓词 π_i 的生产才是可行的。最后，一个动作生产使用指定的突变器 μ_i 来更新状态。

形式上，语法序列 α 在用户状态 S 中生成的语言是 $L(S, \alpha) = \{w \mid (S, \alpha) \Rightarrow^* (S', w)\}$，语法 G 的语言是 $L(S_0, G) = \{w \mid (S_0, S) \Rightarrow^* (S, w)\}$，初始用户状态 S_0 (S_0 可以为空)。如果 u 是 w 的前缀或等于 w，我们就写成 $u \leq w$。*如果* L *存在* *ALL(*)* 语法，那么语言 L 就是 *ALL(*)*。

$L(G)$的类是递归可数的，因为每个突变体都可能是图灵机。在现实中，语法编写者不会使用这种通用性，因此标准做法是将语言类视为上下文敏感语言。该类语言是上下文敏感语言，而不是无上下文语言，因为谓词可以检查调用栈和左右终端。

这种形式主义有各种实际 ANTLR 语法中没有的语法限制，例如，强制突变体使用自己的规则，不允许使用常见的扩展 BNF（EBNF）符号，如 α^* 和 α^+ 闭包。我们可以做出这些限制，而不会丧失通用性，因为任何一般形式的语法都可以转换成这种更受限制的形式。

4.2 解决模棱两可的问题

模棱两可的语法是指同一输入句子可以有多种识别方式。图中的规则

$A \in N$	非终端
a、b、c、$d \in T$	终端
$X \in (N \cup T)$	生产要素
$\alpha, \beta, \delta \in X^*$	语法符号序列
u、v、w、x、$y \in T$	终端顺序
g	空字符串
文件 "符号 "	结束
宿主语言中的$\pi \in \Pi$	谓词
$\mu \in M$	宿主语言中的动作
$\lambda \in (N \cup \Pi \cup M)$	还原标签
$\lambda_1 \ldots \lambda_n$	还原标签序列
生产规则：	
$A \rightarrow \alpha i$	i^{th} A 的无上下文生产
$A \rightarrow \{\pi i\}? \alpha i$	i^{th} 以语义为前提的生产
$A \rightarrow \{\mu i\}$	i^{th} 用突变器生产

图 5.谓语语法符号

$$\text{产品}\ \frac{A \rightarrow \alpha}{(S,\ uA\delta) \Rightarrow (S,\ u\alpha\delta)}$$

$$Sem\ \frac{\pi(S)}{A \rightarrow \{\pi\}?} \qquad \text{行动}\ \frac{A \rightarrow \{\mu\}}{(S, uA\delta) \Rightarrow (\mu(S), u\delta)}$$

$$\text{闭合}\ \frac{(S, \alpha) \Rightarrow (S', \alpha'), (S', \alpha') \Rightarrow^* (S'', \beta)}{(S, \alpha) \Rightarrow^* (S'', \beta)}$$

图 6.谓词语法最左派生规则

图 6 并不排除歧义。但是，对于一个实用的语法分析器来说，每个输入都应该对应一个唯一的解析。为此，*ALL(*)* 使用规则中的前导句顺序来解决歧义问题，以支持数字最小的前导句。对于程序员来说，这种策略是自动解决歧义的简洁方法，它通过将 *else* 与最近的 *if* 相关联，以通常的方式重新解决了众所周知的 *if-then-else* 歧义。PEG 和 Bison 解析器具有相同的解决策略。

为了解决取决于当前状态 S 的含糊不清问题，程序员可以插入语义谓词，但必须使

在所有可能存在歧义的输入序列中，谓词都是互斥的，这样就能使生成的结果明确无误。由于谓词是用图灵完备语言编写的，因此无法在静态下执行互斥。不过，在解析时，*ALL(*)* 会对谓词进行评估，并动态报告有多个谓词生成的输入短语。如果程序员未能满足互斥性，*ALL(*)* 会使用生产顺序来解决歧义。

4.3 增强过渡网络

输入语法元素	结果 ATN 过渡
$A \rightarrow \alpha i$	$p_A \xrightarrow{\varepsilon} p_{A,i} \xrightarrow{\varepsilon} \boxed{\alpha i} \xrightarrow{c} p'_A$
$A \rightarrow \{\pi i\}?$	$p_A \xrightarrow{\varepsilon} p_{A,i} \xrightarrow{\pi} \boxed{\alpha_i} \xrightarrow{\varepsilon} p'_A$
$A \rightarrow \{\mu\}$	$p_A \xrightarrow{c} p_{A,i} \xrightarrow{\mu} p'_A$
$A \rightarrow gi$	$p_A \xrightarrow{c} p_{A,i} \xrightarrow{c} p'_A$
$\boxed{\alpha i} = X1X2 \cdots Xm$	$p0 \xrightarrow{X1} p1 \rightarrow \cdots \xrightarrow{X2} pm$
对于j $X \in N \cup T$, $j = 1 \ldots m$	

图 7.预设语法到 ATN 的转换

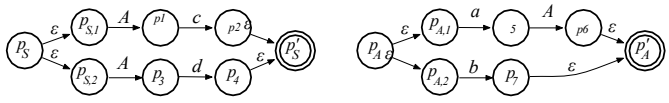


图 8.$P = \{S \rightarrow Ac \mid Ad, A \rightarrow aA \mid b\}$ 的 G 的ATN

是 $p_A \in Q$，目标是 p_{A_i}，由 α_i 的左边缘创建、中的一条边。由 α 创建的最后一个状态，目标 p。$_A$ $_i$ $_A$

非终端边 $p \dashrightarrow q$ 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 q 到状态调用栈上，以便在到达

给定谓语语法 $G = (N, T, P, S, \Pi, M)$，相应的 ATN $M_G = (Q, \Sigma, \Delta, E, F)$ 有元素 [20]：

- Q 是状态集合
- Σ是边缘字母表 $N \cup T \cup \Pi \cup M$
- Δ 是过渡关系映射 $Q \times (\Sigma \cup \epsilon) \rightarrow Q$
- $E \in Q = \{p_A \mid A \in N\}$ 是子机入口状态集
- $F \in Q = \{p' \mid A \in N\}$ 是子机器最终状态的集合

ATN 类似于记录原语的语法图，每个非终端都有一个 ATN 子机。图 7 展示了如何根据语法生成构建状态 Q 和边 Δ 的集合。A 的起始状态

A 的子机器的停止状态，p'。图 8 给出了简单语法的 ATN。ATN 所匹配的语言与原始语法的语言相同。

4.4 前瞻性 DFA

ALL(*) 分析器记录的是使用*前瞻性 DFA 进行* ATN 模拟所得到的预测结果，这些 DFA 增加了接受状态，从而得到预测的生产数量。每个决策的生产量都有一个接受状态。

定义 4.1.前瞻性 DFA *是具有增强接受状态的 DFA，可产生预测的生产数字。对于预言语法* $G = (N, T, P, S, \Pi, M)$，*DFA* $M = (Q, \Sigma, \Delta, D_0, F)$ *其中：*

- Q 是状态集合
- $\Sigma = T$ 是边缘字母表
- Δ 是映射 $Q \times \Sigma \rightarrow Q$ 的过渡函数
- $D_0 \in Q$ 是起始状态
- $F \in Q = \{f_1, f_2, ..., f_n\}$ 最终状态，每个产品有一个 f_i i

符号 $a \in \Sigma$ 在 Δ 中从状态 p 到状态 q *的*转换形式为 $p \rightarrow^{a} q$，我们要求 $p \rightarrow^{a} q'$ 意味着 $q = q'$。

5. *ALL(*)* 解析算法

有了语法、ATN 和前瞻 DFA 的正式定义，我们就可以介绍 *ALL(*)* 解析算法的关键功能了。本节首先总结了这些功能以及它们是如何结合在一起的，然后在介绍功能本身之前讨论了关键图数据结构。最后，我们将举例说明该算法的工作原理。解析从函数 *parse* 开始，该函数的行为类似于传统的自上而下 *LL(k) 解析*函数，但 *ALL(*)* 解析器使用一个名为 *adaptivePredict* 的特殊函数来预测生成，而不是通常的 "打开下 *k 个*标记类型 "机制。函数 *adaptivePredict* 模拟原始谓词语法的 ATN 重现，以选择一个 α_i 扩大决策点 A 的生产 $\rightarrow \alpha_1 | ... | \alpha_n$。

从概念上讲，预测是当前解析器调用栈 γ_0 、剩余输入 w_r 和用户状态 S（如果 A 有预测）的函数。为提高效率，预测尽可能忽略 γ_0 （第 3.1 节），并使用来自 w_r 的最小前瞻性。

为避免对相同输入和非终端重复进行 ATN 模拟，*adaptivePredict* 组装了 DFA，用于记忆输入到预测生产的映射，每个非终端一个 DFA。回想一下，每个 DFA 状态 D 是在匹配通向该状态的前瞻符号后可能出现的 ATN 配置集合。函数 *adaptivePredict* 调用 *startState* 创建初始 DFA 状态 D_0 ，然后调用 *SLLpredict* 开始模拟。

函数 *SLLpredict* 为前瞻性 DFA 添加了以下路径

通过重复调用 *target*，匹配部分或全部 w_r 。函数 *target* 使用*移动*和*闭包*操作，从当前状态 D 计算 DFA 目标状态 D' ，这些操作类似于在

子集构造。函数 *move* *查找在*当前输入符号上可以到达的所有 ATN 配置，而 *closure* 则查找在不穿越终端边的情况下可以到达的所有配置。与子集构造的主要区别在于，*closure* 模拟了与非终端相关的 ATN 子机的调用和返回。

如果 *SLL* 模拟发现了冲突（第 5.3 节），*SLLpredict* 会倒退输入，并调用 *LLpredict* *重新*进行预测，这次要考虑 γ_0 。函数 *LLpredict* 与 *SLLpre- dict* 类似，但不会更新非终端的 DFA，因为 DFA 必须对堆栈不敏感，才能适用于所有堆栈上下文。*LLpre- dict* 发现的 ATN 配置集内的冲突代表模糊性。这两个预测函数都使用 *get-ConflictSetsPerLoc* 来检测冲突，即代表相同解析器位置但不同制作的配置。为了避免不必要地切换到 *LLpredict*，当 *getConflictSetsPerLoc* 报告冲突时，*SLLpredict* 会使用 *getProdSetsPerState* 查看是否还存在潜在的非冲突 DFA 路径。如果是，则值得继续使用 *SLLpredict*，因为更多的前瞻性可能会解决冲突，而无需诉诸完整的 *LL* 解析。

在详细介绍这些功能之前，我们先回顾一下

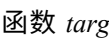
基本图数据结构，用于有效管理多个调用栈，*如* GLL 和 GLR。

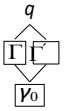
5.1 图结构调用堆栈

模拟会从它们的堆栈中弹出 q。预测可以通过合并堆栈将这些子解析器视为一个子解析器。我们将 DFA 状态中形式为（p, i, γ_1 ）和（p, i, γ_2 ）的所有配置的堆栈合并，形成一个一般配置（p, i, Γ），其中包括

图结构栈（GSS）[25] $\Gamma = \gamma_1 \rbrack \gamma_2$ 其中]表示图合并。Γ 可以是空栈 []，也可以是一个特殊的栈 # 用于

GSS：用于 *SLL* 预测的 GSS（稍后讨论）、单个堆栈或堆栈节点图。将单个堆栈合并为一个 GSS，可将潜在规模从指数复杂度降低为线性复杂度（定理 6.4）。为了表示 GSS，我们使用了节点共享最大化的不可变图形数据结构。下面是两个共享解析器栈 γ_0 的例子：

$p\gamma_0 \rbrack q\gamma_0 =$ 

$q\Gamma\gamma_0 \rbrack q\Gamma'\gamma_0 =$ 

实现 *ALL(*)* 预测的最简单方法是经典的反向跟踪方法，即为每个 α_i 启动一个子解析器。由于反向跟踪子解析器不知道何时停止解析--它们不知道其他子解析器的状态，因此子解析器将消耗掉所有剩余输入。独立的子解析器还会导致指数级的时间复杂性。我们通过让预测子解析器

$_r$当除一个子解析器外的所有子解析器都停止运行，或者当预测发现冲突时，预测会在消耗完前缀 $u \le w_r$ 后终止。同步运行

也为子解析器共享调用堆栈提供了机会，从而避免了多余的计算。

处于 ATN 状态 p *的*两个子解析器共享相同的 ATN 堆栈顶（$q\gamma_1$ 和 $q\gamma_2$ ），它们将相互映射对方的行为，直到

在接下来的函数中，对配置集的所有添加（如使用操作符 += 添加）都会隐式合并堆栈。

有一种特殊情况与预测开始时的堆栈条件有关。Γ 必须区分空堆栈和无堆栈信息。对于 *LL* 预测，初始 ATN 模拟堆栈是当前的解析器调用堆栈 γ_0。只有当判定入口规则是起始符时，初始堆栈才是空的，即 $\gamma_0 = []$。另一方面，对堆栈不敏感的 *SLL* 预测会忽略解析器调用堆栈，并使用 # 表示无堆栈信息的初始堆栈。在计算代表子机器停止状态的配置*闭合*（函数 7）时，这一区别非常重要。如果没有解析器堆栈信息，从决策入口规则 *A* 返回的子解析器必须考虑所有可能的调用位置；也就是说，*闭包可以*看到
配置 $(p', -, \#)$。

对于 *LL* 预测，空栈 [] 与其他节点一样处理：Γ ⟧[] 产生等价于集合 {Γ, []} 的图，这意味着 Γ 和空堆栈都是可能的。推
状态 *p* 到 [] 会产生 *p*[] 而不是 *p*，因为弹出 *p* 必须使 [] 堆栈符号为空。对于 *SLL* 预测，Γ ⟧# = # 适用于任何图形 Γ，因为 # 就像通配符，代表集合
的所有堆栈。因此通配符包含任何 Γ。将状态 *p* 推到 # 上，得到 *p*#。

5.2 *ALL(*)* 解析功能

现在我们可以介绍关键的 *ALL(*)* 功能，我们用方框突出了这些功能，并将其穿插在本节的正文中。我们的讨论遵循自上而下的顺序，并假设语法 *G* 对应的 ATN、语义状态 S、构造中的 DFA 和*输入*都在算法所有函数的范围内，而且语义谓词和操作可以直接访问 S。

函数解析主要入口点是函数*解析*
(该函数首先将模拟调用堆栈 γ 初始化为空堆栈，并将 ATN 状态 "光标 "*p* 设为 $p_{S,i}$，*即适应性预测（adaptivePredict*）所预测的 *S* 的执行号 *i* 左边缘上的 ATN 状态。函数循环运行，直到光标到达 *S* 的子机器停止状态 p'。如果光标到达另一个子机器停止状态 p'、

parse 通过从调用栈中弹出返回状态 *q* 并将 *p* 移至 *q* 来模拟 "返回"。

```
功能 1：解析（s）
γ := []; i := adaptivePredict(S, γ); p := p ；S,i
while true do
  如果 p ≠ p´ （即p 是规则停止状态）
    如果 B = S（完成匹配起始规则S），则返回;
    ，否则 let γ = qγ in γ := γ ; p := q;
  不然
    switch t where p →-t q do
      情况 b：（即终端符号过渡）
        if b = input.curr() then
          p := q; input.advance();
        否则解析错误;
      情况 B：γ := qγ； i := adaptivePredict(B，γ)； p:=p
      ；B,i
      情况 μ：S := μ(S); p := q;
      case π: if π(S) then p := q else parse error;
      情况 ϵ: p := q;
    结束语
```

对于不处于停止状态的 *p*，*解析* ATN 过渡 $p \xrightarrow{-t} q$ 的过程。如果 *t* 是终结边，并且符合

如果 t 是当前输入符号的非终端边，*parse* 会转换该边并移动到下一个符号。如果 *t* 是引用某个 *B* 的非终端边缘，*parse* 会模拟子机器调用，将返回状态 *q* 推入堆栈，并通过调用 adaptivePredict 和设置光标来选择 *B* 中合适的生产左边缘。对于动作边，*解析会*根据突变器 μ 更新状态并过渡到 *q*。对于谓词边，*解析*只有在谓词 π 评估为 true 时才会过渡。在解析过程中，失败的谓词就像不匹配的标记一样。函数*解析*不会明确检查解析是否在文件结束时停止，因为开发环境等应用程序需要解析输入的子句。

函数 adaptivePredict。*解析*调用 *adaptivePredict*（函数 2）来预测结果，该函数是判定非终端 *A* 和当前解析器堆栈 γ_0 的函数。由于预测只在完全 *LL* 模拟期间评估谓词，因此如果至少有一个结果是谓词，*adaptivePredict 就会*委托给 *LLpredict* 。[5] 对于还没有 DFA 的决策，*adaptivePredict* 会创建 DFA dfa_A，起始状态为 D_0，为 *SLLpredict* 添加 DFA 路径做准备。D_0 是 ATN 配置的集合，无需遍历 ter-minal 边即可到达。函数 *adaptivePredict* 还会构建最终状态集 F_{DFA}，其中包含 A 的每个生产的一个最终状态 f_i。DFA 状态集 Q_{DFA} 是 D_0、F_{DFA} 和错误状态 D_{error} 的联合。词汇表 Σ_{DFA} 是语法终端 *T* 的集合。对于具有现有 DFA 的未预测决策，输入游标，它的做法是在进入游标时捕捉输入索引作为*起点*，并在返回前倒退到*起点*。

```
函数 2：adaptivePredict(A, γ0 ) 返回 int alt
start := input.index(); // 检查点输入
如果 ∃ A → π_i α_i 那么
    alt := LLpredict(A, start, γ )； 0
    input.seek(start); // 撤销数据流位置更改
    return alt;
if ∄ dfa_A then
    D0 := startState(A, #);
    F_DFA := {f_i | f_i := DFA State(i) ∀ A → α_i };
    QDFA : = D0 ∪ FDFA ∪ Derror;
    dfa_A := DFA(Q_DFA , Σ_DFA = T, Δ_DFA = ∅, D0 , F )； DFA
alt := SLLpredict(A, D0 , start, γ0 );
input.seek(start); // undo stream position
changes return alt;
```

adaptivePredict 会调用 *SLLpredict* 从 DFA 中获取预测结果，并可能在此过程中通过 ATN 仿真扩展 DFA。最后，由于 *adaptivePredict 进行*的是前瞻而非解析，因此它必须撤销对 DFA 所做的任何更改。

[5] *SLL* 预测在本文中为了清晰起见没有加入谓词，但在实际应用中，ANTLR 会将谓词加入 DFA 的接受状态（第 B.2 节）。ANTLR 3 DFA 使用的是谓词边而不是谓词接受状态。

函数 startState。要创建 DFA 的起始状态 D_0，*startState*（函数 3）会为每个 $A \to \alpha_i$ 和 $A \to \pi_i \alpha_i$ 添加配置（$p_{A,i}$, i, γ），如果 π_i 评估为真。从 *adaptivePredict* 调用时，调用堆栈参数 γ 是所需的特殊符号 # 。

表示 "无解析器堆栈信息"。从 *LLpredict* 调用时，γ 是初始解析器堆栈 γ_0。完成配置*闭合后*，D_0。

函数 SLLpredict函数 *SLLpredict*（函数 4）同时执行 DFA 和 *SLL* ATN 仿真，逐步增加

向 DFA 添加路径。在最好的情况下，对于前缀 $u \leq w_r$ 和某个生产编号 i，已经存在一条从 D_0 到接受状态 f_i 的 DFA 路径。在最坏的情况下，ATN sim-

SLLpredict 的主循环会在 a 上找到一条从 DFA 状态游标 D *出发的*现有边，或者通过 *target* 计算一条新的边。D' 在这种情况下，函数 *target* 将返回 D′，因为 D' 可能已经计算过出边；通过替换 D' 来丢弃工作是低效的。在下一次迭代中，*SLLpre-dict* 将考虑来自 D' 的边，从而有效地切换回 *DFA* 模拟。

函数 3： *startState*(A, γ) **返回** DFA 状态 D_0
$D_0 := \varnothing$
预选 $p \xrightarrow{\epsilon} p_{A,i} \in \Delta_{ATN}$ **做**
  如果 $p \xrightarrow{\pi i} p$ **则** $\pi := \pi_i$，**否则** $\pi := \epsilon$
  如果 $\pi=\epsilon$ *或* $eval(\pi_i)$ **则** D_0 += *closure*({},D_0 ,($p_{A,i}$,i,γ))
return D_0；

函数 4： *SLLpredict*(A, D_0, *start*, γ_0) **返回** int prod
$a := input.curr()$; $D = D_0$；
while *true* **do**
  让 D' 成为 DFA 目标 $D \xrightarrow{a} D'$；
  if ∄ D' **then** $D' := target(D, a)$;
  如果 $D' = D_{error}$，**则***解析错误；*
  如果 D' 堆栈敏感，**那么**
    input.seek(*start*); **return** *LLpredict*(A, *start*, γ)；
  如果 $D' = f_i \in F_{DFA}$，**则返回** i；
  $D := D'$; $a := input.next()$；

SLLpredict 获取目标状态 D' 后，会检查错误、堆栈敏感性和完成情况。如果*目标状态*将 D' 标记为堆栈敏感，则预测需要完整的 *LL* 模拟，*SLL- predict* 调用 *LLpredict*。$'_i$在这种情况下，D' 中的所有配置都预测出了相同的结果 *i*；无需进一步分析，算法可以停止。对于任何其他 D'，算法会将 *D* 设为 D'，得到下一个符号，然后重复。

功能目标。通过组合*移动-关闭*操作，*target* 发现了单个终端符号 $a \in T$ 时可从 *D* 到达的 ATN 配置集。函数*移动*组合
通过遍历可直接到达 *a* 的配置
一个终端边缘：

$$\text{移动}(D, a) = \{(q, i \quad q, \quad (p, i, \Gamma) \in D\}$$
$$, \Gamma) \mid p \xrightarrow{a}$$

这些配置及其*闭包*构成 D'。如果 D' 为空，则没有任何替代方案可行，因为没有任何方案能与当前状态的 *a* 匹配，因此*目标*返回错误状态 D_{error}。如 ◆◆◆ D′ 中 ◆◆◆所有配置都预测了相*同的*生产编号 i，则*目标返回错误状态 D*。
添加边 $D \xrightarrow{a} f$ 并返回接受状态 *f*。如果 D' 有连接配置，*target* 会将 D' 标记为堆栈敏感。目标将 D 标记为堆栈敏感。
冲突可能是 *SLL* 缺乏解析器堆栈信息造成的歧义或缺陷。(冲突以及 *getConflictSetsPerLoc* 和 *getProdSetsPerState* 将在第 5.3 节中介绍）。如果 DFA 中还没有等价的状态 $\underline{D'}$，*a* 则函数将添加状态 D'，并添加边
$D \dashrightarrow D$
.

函数 5：*target*(*D*, *a*) **返回** DFA 状态 D'
mv := *move*（*D*，
D) := *closure*({}，*c*)；
如果 $D^{c \in mv} = \varnothing$ **则** Δ_{DFA} += D_{error}; **return** D_{error}；
如果 {*j* | (-, *j*, -) ∈ D' } = {i}
那么 Δ_{DFA} += $D \xrightarrow{a} f_i$; **return** f_i; // *预测规则 i*
// *寻找 D 配置之间的冲突*
a conflict := ∃*alts* ∈*getConflictSetsPerLoc*(D')：
|*alts*|>1; *viablealt* := ∃*alts* ∈ *getProdSetsPerState*(D')
：|*alts*| = 1; **如果***是冲突且不是 viablealt*，**则**
将 D' 标记为堆栈敏感；
如果 $D' = \underline{D'} \in Q_{DFA}$，**那么** D' := D′；**否则** Q_{DFA} += D'；
Δ_{DFA} += $D \xrightarrow{a} D'$；
return D'；

(第 5.3 节解释了模糊检测）。
D 移动到 D'，并考虑下一个输入符号。

函数 6：*LLpredict*(*A*, *start*, γ_0) **返回** int *alt D* :=
D_0 := *startState*(*A*, γ)；
mv := *move*（*D*，
while *true* **do**
D := *closure*({}，*c*)；
如果 $D^{c \in mv} = \varnothing$ **则***解析错误；*
if {*j* | (-, *j*, -) ∈ D' } = {i} **then return** *i*；
/* *如果所有 p, Γ 对的预测值都 > 1 alt 且所有这样的*
生产集相同，输入模糊。/
altsets := *getConflictSetsPerLoc*(D')；
如果 ∀ *x*, *y*∈ *altsets*, *x* = *y* *且* |x| > 1，**那么**
x := *altsets 中的任意集合；*
报告起始.....输入.index() 中不明确的 alts x；
return *min*(*x*)；
D := D'；*input.advance*()；

函数闭合*闭合*操作（函数 7）追逐从 *p* *（从*配置参数 *c* 投影的 ATN 状态）可到达的所有ϵ 边，并模拟子机的调用和返回。函数*闭合*将 *μ* 和 *π 边*视为ϵ 边，因为突变体不应在预测过程中执行，而谓词只在起始状态时进行评估。计算。由参数 *c* = (*p*, *i*, Γ) 和边 $p \xrightarrow{\epsilon} q$、
闭包将 (*q*, *i*, Γ) 添加到本地工作集 *C* 中。

称边 *p* B→*q*，*闭包*增加了（*p* $\xrightarrow{B}$ ）*i*，*q*Γ
从子机停止状态返回 p' $_B$ 添加了
配置 (*q*, *i*, Γ)，在这种情况下，*c* 的形式为 (p' , *i*, *q*Γ)。一般来说，配置栈 Γ 是代表多个单独栈的图。函数*闭包必须*
模拟从每个 Γ 栈顶返回。算法使用 qΓ′ ∈ Γ 表示 Γ 的所有栈顶 *q*。为了避免由于 *SLL* 右递归和ϵ 边导致的非终止
等分则中 ()+，*闭合*使用 *繁忙* 共享的一套
用于计算同一个 *D* 的所有闭包运算。
当*闭合*达到停止状态 *p* 时，，$_A$ 执行决策进入规则、*A*、*LL* 和 *SLL* 预测的行为不同。 *LL* 预测从解析器调用堆栈 γ_0 中弹出，"返回 "到调用 *A* 的子机器的状态。另一方面，*SLL* 预测无法访问解析器调用栈，

函数 LLpredict。当 *SLL* 模拟冲突时，*SLLpre- dict* 会倒转输入并调用 *LLpredict*（函数 6），以获得基于 *LL* ATN 模拟的 预测结果，该模拟考虑了整个解析器堆栈 γ_0。函数 *LLpredict* 与 *SLL- predict* 类似。它使用 DFA 状态 D 作为游标，并使用状态 D' 作为一致性转换目标，但不更新 A 的 DFA，因此 *SLL* 预测可以继续使用 DFA。*LL* 预测继续进行，直到 $D' = \varnothing$、D' 唯一预测了一个变化状态，或者 D' 发生冲突。如果 *LL* 模拟中*的*D' 和 *SLL* 一样有冲突，算法会报告有歧义的短语（从*开始*到当前索引的输入），并在有冲突的配置中找出最小的生产编号。

必须考虑所有可能的 A 调用位置。函数*闭包*发现 $\Gamma = \#$（和

$'' = p'$），因为 *startState* 会将初始堆栈设置为 # 而不是 γ_0。$^{p_B}{}^{A}$决策时的返回行为是
条目规则是 *SLL* 解析与 *LL* 解析的区别所在。

5.3 冲突和模糊检测

冲突配置的概念是 *ALL(*)* 分析的核心。在 *SLL* 预测期间，冲突会触发故障切换到完全 *LL* 预测，并在 *LL* 预测期间发出模糊信号。配置间冲突的充分条件是它们仅在预测的备选方案上存在差异：（p, i, Γ）和（p, j, Γ）。有两个函数可以帮助检测冲突。第一个函数是 *getConflictSetsPerLoc*（函数 8），用于收集
与所有（$p, -, \Gamma$）配置相关的生产数集。如果一个 p、Γ 对预测了不止一个产量，则一个

函数 7： *closure*(*busy*, *c* = (*p*, *i*, Γ)) **返回**集合 *C*
如果 c∈ *busy*，**则返回** ∅；**否则** *busy* += *c*； *C* := {c}；
如果 *p* ⇒ *p*′ *（即 p 是任何停止状态，包括 p*′ *）*，**则如果** Γ = ≤# *（即栈是 SLL 通配符）*，**则**
C += *closure*(*busy*, (*p*2 , *i*, #)); // *call site*
∀*closure*
p2 : *p1*→*p2*∈ΔATN
else // 非空 *SLL* 或 *LL* 堆栈
对于 qΓ′ ∈ Γ（即图 Γ 中的每个栈顶 *q*）**做**
C += *closure*(*busy*, (*q*, *i*, Γ′)); // *"return" to q*
返回 *C*；
最
foreach *p* –*边缘* *q* **do switch**
后*edge* **do**
情况 *B*： *C* += *closure*(*busy*, (*pB* , *i*, *q*Γ))；
case *π*, *μ*, *ϵ*： *C* += *closure*(*busy*, (*q*, *i*, Γ))；
返回 *C*；

存在冲突。下面是一组配置和相关冲突集的示例：

{(*p*，1，Γ)，(*p*，2，Γ)，(*p*，3，Γ)，(p，1，Γ′)，(*p*，2，Γ′)，(*r*，2，Γ″)}

{1,2,3} {1,2} {2}

这些冲突集表明，生产{1，2，3}可到达位置 *p*，Γ，生产{1，2}可到达位置 *p*，Γ′，生产{2}可到达位置 *r*，Γ″。

// 对于每个 *p*、Γ，从 (*p*、-、Γ) ∈ *D* configs 获取一组别名 {i}
函数 8： *getConflictSetsPerLoc*(*D*) **返回***冲突*集集合
s := ∅;
for (*p*, -, Γ) ∈ *D* **do** *prods* := {*i* | (*p*, *i*, Γ)}; *s* := s∪ *prods*；
返回 *s*;

第二个函数 *getProdSetsPerState*（函数 9）与之类似，但只收集与 ATN 状态 *p* 相关的生产编号：

{(*p*，1，Γ)，(*p*，2，Γ)，(*p*，3，Γ)，(p，1，Γ′)，(*p*，2，Γ′)，(*r*，2，Γ″)}

{1,2,3} {2}

从 *SLL* 跳转到 *LL* 预测（*LL- predict*）的充分条件是至少存在一组相互冲突的配置：*getConflictSetsPerLoc* 返回至少一组具有一个以上生产编号的配置。例如，参数 *D* 中存在配置 (*p*, *i*, Γ) 和 (*p*, *j*, Γ)。然而，我们的目标是尽可

如果从 *getConflictSetsPerLoc 得出的*结果预测到一个以上的变化，那么再多的前瞻也无法得出唯一的预测结果。分析必须通过 *LLpredict* 调用堆栈 γ_0 再试一次。

// For each *p* return set of alts *i* from (*p*, -, -) ∈ *D* configs.
函数 9： *getProdSetsPerState*(*D*) **返回**集合的集合
s := ∅;
for (*p*, -, -) ∈ *D* **do** *prods* := {*i* | (*p*, *i*, -)}; *s* := *s* ∪ *prods*；
返回 *s*;

当 *getConflictSetsPerLoc* 中的每个冲突集包含 1 个以上的产品（*D* 中的每个位置都可以从 1 个以上的产品到达）时，*LL* 模拟中的冲突就会发生。一旦多个子解析器到达
同样的（*p*，-，Γ），未来所有由（*p*，-，Γ）派生的模拟都将表现相同。更多的前瞻性将无法解决环境问题。
guity。预测可以在此时终止，并报告 looka- head 前缀 *u* 为模棱两可，但 *LLpredict 会*继续预测，直到确定 *u* 对于 *哪些*制作是模棱两可的。考虑冲突
集 {1,2,3} 和 {2,3}。由于这两个集合的度数都大于 1，因此这两个集合代表了一个模糊集合，但是额外的输入将确定 $u \le w_r$ 在 {1,2,3} 或 {2,3} 上是否有歧义。函数 *LLpredict 将*继续运行，直到所有冲突集都能识别{1,2,3}或{2,3}。
模糊度相等；条件 *x* = *y* 且 |x| > 1 ∀ *x*, *y* ∈ *altsets* 体现了这一测试。

为了检测冲突，算法会频繁比较图结构堆栈。从技术上讲，当配置（*p*，*i*，Γ）和（*p*，*j*，Γ′）出现在相同的配置中时，就会发生冲突
集，且至少有一个堆栈轨迹 γ 与
和 Γ′。因为检查图的交集是前
因此，该算法采用相等法（Γ = Γ′）作为启发式算法。由于共享子图，相等算法的速度要快得多。图相等算法通常可以检查节点身份来匹配两个完整的子图。在最糟糕的情况下，相等子集启发式会延迟冲突检测，直到冲突配置之间的 GSS 是简单的线性堆栈，其中图交集与图相等相同。这种启发式的代价是更深的前瞻性。

能长时间地继续使用 *SLL 预测*，因为 *SLL* 预测会更新前瞻 DFA 缓存。为此，如果存在至少一个不冲突的配置（当 *getProdSetsPer- State* 返回至少一个大小为 1 的集合时），*SLL 预测就会*继续。我们希望更多的前瞻性会导致一个配置集预测出唯一的

通过该不冲突的配置来决定 S → a|a|a .p b。例如，*S→a|a|a ·p b 的*决策在制作 1 和 2 之间的 *a 上*是模棱两可的，但在 *ab* 上则是明确无误的。位置

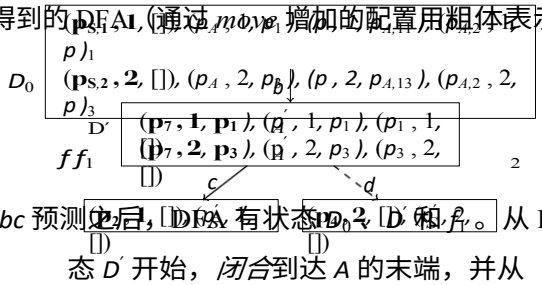
是 *a* 和 *b* 之间的 ATN 状态）匹配输入 *a* 后、

则配置集为 {(*p*, 1, []), (*p*, 2, *[]*), (p, 3, *[]*)} 。函数 *getConflictSetsPerLoc* 返回 {{1, 2}, {3}}。下一步对 *b* 的*移动封闭*会导致非冲突配置集

{(*p*, 3, [])} from (*p*, 3, []), 绕过冲突。如果所有集合 re

5.4 DFA 构建样本

为说明算法行为，请考虑图 8 中语法和 ATN 的输入 *bc* 和 *bd*。针对决策 *S 的* ATN 模拟在左边缘节点 $p_{S,1}$ 和 $p_{S,2}$ 启动子解析器，初始 D_0 配置为 ($p_{S,1}$, 1, []) 和 ($p_{S,2}$, 2, []) 。函数*闭包在* "调用 "*A 时，*在 "返回 "节点 p_1 和 p_3 *上为* D_0 增加了三个配置。以下是在 *bc* 和 *bd* 之后进行 ATN 模拟得到的 DFA（通过 *move* 增加的配置用粗体表示）：



在 *bc* 预测之后，DFA 有状态 D_0、D_1 和 D_2。从 DFA 状态 D_0 开始，*闭合*到达 *A* 的末端，并从

状态 f_1 独一无二地预测了第 1 个短语。在预测第二个短语 *bd* 时，状态 f_2 被创建并连接到 DFA（如虚线箭头所示）。函数 *adaptivePredict* 首先使用 DFA

在看到 *bd* 之前，模拟从 D_0 到 D'、

D' 没有 *d* 边，因此 *adaptivePredict* 必须使用 ATN 模拟添加边 $D' \xrightarrow{d} f$。[2]

6. 理论成果

本节指出了关键的 *ALL(*)* 定理，并展示了 解析器的时间复杂度。详细证明见附录 A。**定理 6.1** *（正确性）。针对非左-右-左语法的* ALL(*)

如果 w∈ *L*(*G*)，*递归 G 就能识别句子 w。*
定理 6.2.ALL(*) *语言在联合下是封闭的。*

定理 6.3.*对 n 个符号进行* ALL(*) *解析的时间为 O*(*n*)。[4]

定理 6.4.*对于 n 个输入符号，GSS 有 O*(*n*) *个节点。*

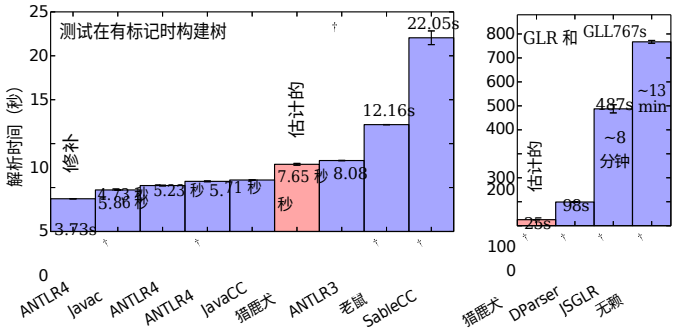
定理 6.5.如果 w∈ *L*(*G*)，则*非左递归 G 的两阶段解析会识别句子 w。*

7. 经验结果

我们通过实验比较了 *ALL(*)* Java 解析器与其他策略的性能，检查了 *ALL(*)* 对其他各种语言的吞吐量，强调了前瞻性 DFA 缓存对解析速度的影响，并提供了在实践中线性 *ALL(*) 性能的*证据。

7.1 *将 ALL(*)* 的速度与其他解析器进行比较

我们的第一个实验比较了 10 种工具和 8 种解析策略的 Java 解析速度：手工调整的递归-后裔优先解析、*LL(k)*、*LL(*)*、PEG、*LALR*(1)、*ALL(*)* GLR 和 GLL。图 9 显示了每种工具解析 12 920 个 Java 6 库和编译器源文件所需的时间。我们之所以选择 Java，是因为它是工具中最常用的语法，而且 Java 示例源也很多。除了 DParser 和 Elkhound 没有提供合适的 Java 语法外，本实验中使用的 Java 语法都直接来自相关工具。我们使用明确的表达式规则将 ANTLR 的 Java 语法移植到这些工具的元语法中。我们还在 Elkhound 语法中嵌入了合并动作，以便在解析过程中消除歧义，从而模仿 ANTLR 的歧义解决方法。所有输入文件都在解析前加载到 RAM 中，解析时间反映的是 10 次完整的语料传递所测得的平均时间，跳过前两次以确保 JIT 编译器预热。对于 *ALL(*)*，我们使用了第 3.2 节中的两阶段解析。测试机是运行 Java 7 虚拟机的 6 核 3.33Ghz 16G 内存 Mac OS X 10.7 台式机。Elkhound 和 DParser 解析器使用 C/C++ 实现，没有垃圾回收器同时运行。Elkhound 最后一次更新是在 2005 年，目前已无法在 Linux 或 OS X 上运行，但我们可以在 Windows 7（4 核 2.67Ghz 24G 内存）上运行。Elkhound



也只能从文件中读取，因此 Elkhound 的解析时间不具可比性。为了控制机器速度的差异以及 RAM 与 SSD 的对比，我们计算了在 OS X 机器上从 RAM 读取 Java 测试设备的时间与从 SSD 读取 Java 测试设备的时间之比。

图 9.在 Java 6 库和编译器源代码上比较 10 种工具和 8 种策略的 Java 解析时间（越小越快）。12 920 个文件，360 万行，大小 123M。工具描述符包括："**工具名称** *版本* [*策略*]"。**ANTLR4** 4.1 [*ALL(*)*]; **Javac** 7 [手工构建的递归-后裔和先例表达式解析器]；**JavaCC** 5.0 [*LL(k)]*；**Elkhound** 2005.08.22b [GLR]（*在 Windows 上测试*）；**ANTLR3** 3.5 [*LL(*)]*；**Rats！** 2.3.1 [PEG]; **SableCC** 3.7 [*LALR(*1)]; **DParser 1.**2 [GLR]; **JSGLR** (from Spoofax) 1.1 [GLR]；**Rascal** 0.6.1 [GLL]。测试在内存充足的情况下运行了 10 倍，平均值/st- ddev 按最后 8 个计算，以避免 JIT 成本。误差可忽略不计，但由于垃圾回收的原因，会出现一些变化。为避免使用对数刻度，我们对 GLR、GLL 解析时间使用了单独的图表。

测试设备在 Windows 系统上运行，从固态硬盘中提取数据。我们报告的 Elkhound 时间是 Windows 时间乘以 OS X 与 Windows 的比率。

在本实验中，*ALL(*)* 的表现优于其他解析器生成器，只比手工生成的解析器慢 20% 左右。

ANTLR 4 是 Java 编译器中的解析器。在比较树构建运行时（图 9 中用 † 标记），ANTLR 4 比我们测试过的最快 GLR 工具 Elkhound 快约 4.4 倍、

比 GLL（Rascal）快 135 倍。ANTLR 4 的非确定性 *ALL(*)* 解析器比 JavaCC 的确定性 *LL(k)* 解析器稍快，比 *Rats!*在另一项测试中，我们发现 *ALL(*)* 在将自己的 PEG 语法转换为 ANTLR 语法时的表现优于 *Rats!LALR*(1) 解析器在与 *LL* 工具的对比中表现不佳，但这可能是 SableCC 的实现问题，而不是 *LALR*(1) 的缺陷。(另一个 *LALR*(1) 工具 JavaCUP 的 Java 语法不完整，无法解析语料）。在重新解析语料库时，*ALL(*)* lookahead 在每次决策时都会获得缓存命中，解析速度比 LALR(1*)* 快 30%。

3.73s.当使用树形结构（未显示时间）进行重新解析时，*ALL(*) 的速度*超过了手工构建的 Javac（4.4 秒对 4.73 秒）。对于开发环境等工具来说，解析速度非常重要。

我们测试的 GLR 解析器在 Java 解析方面比 *ALL(*)* 慢两个数量级。在 GLR 工具中，Elkhound 的性能最好，这主要是因为它在可能的情况下使用线性 *LR*(1) 堆栈而不是 GSS。此外，我们还允许 Elkhound 像 *ALL(*)* 一样在解析过程中进行消歧。与 JSGLR 和 DParser 无扫描器不同，Elkhound 使用单独的词法器。观察到的 *ALL(*)* 性能差异的一个可能解释是，我们移植到 Elkhound 和 DParser 的 Java 语法偏向于 *ALL(*)*，但这一反对意见并不成立。GLR 也应受益于高确定性和非模糊语法。GLL 在这项测试中速度最慢，可能是因为 Rascal 团队移植了 SDF 的 GLR Java 语法、

	工具	时间	内存 (M)
	Javac†	89 毫秒	7
	ANTLR4	201 毫秒	8
	JavaCC	206 毫秒	7
	ANTLR4†	360 毫秒	8
	ANTLR3	1048 毫秒	143
	SableCC†	1 174 毫秒	201
	老鼠†	1 365 毫秒	22
	JSGLR†	15.4 秒	1,030
	无赖†	24.9 秒	2,622
(无DFA)	ANTLR4	42.5 秒	27
	驼鹿[a]	3.35 分钟	3
	DParser†	10.5 小时	100+
	驼鹿†	失效	5390+

图 10.解析和选择性建树所需的时间和空间

3.2M Java 文件。在解析过程中使用 `-XX:+PrintGC`选项（C++进程监控）。时间包括词法；所有输入均已预载。†建树。[a]在解析过程中消歧，无树，*估计时间*。

Rascal 也无需扫描仪，是目前唯一可用的 GLL 工具。

通用算法的最大问题在于，它们在时间和空间上的不可预测性很高，这可能使它们不适合某些商业应用。图 10 汇总了相同工具针对单个 3.2M Java 文件。Elkhound 解析 123M Java 语料库用时 7.65 秒，但解析 3.2M Java 文件用时 3.35 分钟。它在解析森林结构时崩溃（内存不足）。在 3.2M 的文件上，DParser 的时间从 98 秒的语料库时间跃升至 10.5 小时。Rascal 和 JSGLR 的速度在 3.2M 的文件上扩展合理，但分别使用了 2.6G 和 1G 内存。相比之下，*ALL(*)* 解析 3.2M 文件的速度为 360ms，而构建树的速度为 8M。ANTLR 3 的速度很快，但与 ANTLR 4 相比，速度更慢，内存消耗更大（由于采用了回溯记忆法）。

7.2 跨语言的 *ALL(*)* 性能

图 11 给出了 8 种语言 *ALL(*)* 解析器的每秒字节吞吐量，其中包括用于比较的 Java。测试文件的数量和文件大小差异很大（根据我们可以合理收集的输入）；文件越小，解析时间差异越大。

- **C** 源自 C11 规范；无间接左递归，修改了堆栈敏感规则以呈现 *SLL*（见下文）：813 个预处理文件，来自 `postgres` 数据库的 159.8M 源代码。
- **Verilog2001** 源自 Verilog 2001 规范，去除了间接的左递归：385 个文件，659k，来自 [3] 和网络。
- **JSON** 源自规范。4 个文件，331k 来自 twitter。
- **DOT**：源自规范。从网上收集 48 个文件，19.5M。
- **Lua**：源自 Lua 5.2 规范。751 个文件，123k 来自 github。
- **XML** 源自规范。1 个文件，117M 来自 XML 基准。
- **Erlang** 源自 *LALR(1)* 语法。500 个预处理文件，8M。

语法	KB/sec
XML	45,993
Java	24,972
JSON	17,696
DOT	16,152
Lua	5,698
C	4,238
Verilog2001	1,994
埃朗	751

图 11.吞吐量（千字节/秒

Lexing+parsing; 所有输入预载入 RAM。

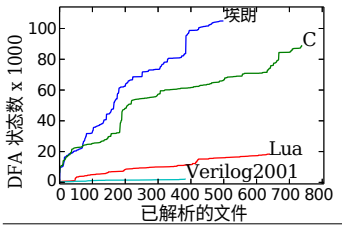


图 12.DFA 增长率与解析文件数量的关系。按磁盘顺序解析的文件。

其中一些语法的解析时间虽然合理，但与 Java 和 XML 相比要慢得多，这表明程序员可以将语言规范转换为 ANTLR 的元语法，而无需对语法进行重大修改。

语法规范。(根据我们的经验，语法规范很少会针对特定工具或解析策略进行调整，而且往往是模棱两可的）。之后，程序员可以使用 ANTLR 的剖析和诊断功能来提高性能，就像任何编程任务一样。例如，C11 规范语法是 *LL 语法*而非 *SLL* 语法，原因是其中的规则 declarationSpecifiers，我们在 C 语法中将其改为 *SLL* 语法（速度提高了 7 倍）。

7.3 前瞻性 DFA 对性能的影响

前瞻性 DFA 缓存对 *ALL(*)* 性能至关重要。为了证明缓存对解析速度的影响，我们禁用了 DFA 并重复了 Java 实验。从图 9 中可以看出，用纯缓存命中来重新解析 Java 语料需要 3.73 秒的解析时间。如果完全禁用前瞻性 DFA 缓存，解析器需要 12 分钟（717.6 秒）。图 10 显示，在 3.2M 文件上，禁用缓存会将解析时间从 203 毫秒增加到 42.5 秒。这一性能与 GLL 和 GLR 解析器的高成本相符，它们也没有通过将解析决策记忆化来减少解析器的猜测。作为中间值，在解析每个语料库文件前清除 DFA 缓存的总时间为 34 秒，而不是 12 分钟。这将缓存的使用隔离到了单个文件，并证明即使在单个文件中，缓存预热也会很快发生。

当解析器遇到新的前瞻性短语时，DFA 的大小呈线性增长。图 12 显示了图 11 中（最慢的四个）解析器遇到新文件时 DFA 状态数量的增长情况。像 C 语言这样具有
结构{...} x；和结构{...} f()；共享一个大的左前缀。相比之下，Verilog 2001
解析器使用的 DFA 状态很少（但由于非 *SLL* 规则，运行速度较慢）。同样，在查看了整个 123M Java 语料库后，Java 解析器只使用了 31,626 个 DFA 状态，平均每个解析文件增加了 ~2.5 个状态。不过，当解析器遇到不熟悉的输入时，DFA 的大小会继续增长。程序员可以清除缓存，*ALL(*)* 将适应后续输入。

7.4 经验解析时间复杂度

鉴于图 11 中的吞吐量范围很大，我们可以怀疑速度较慢的解析器存在非线性行为。为了进行研究，我们在图 13 中绘制了解析时间与文件大小的关系图，并绘制了最小二乘回归和 LOWESS [6] 数据拟合曲线。LOWESS 曲线在参数上不受限制（不要求是直线或任何特定的多项式），而且它们会对解析时间和文件大小之间的关系产生影响。

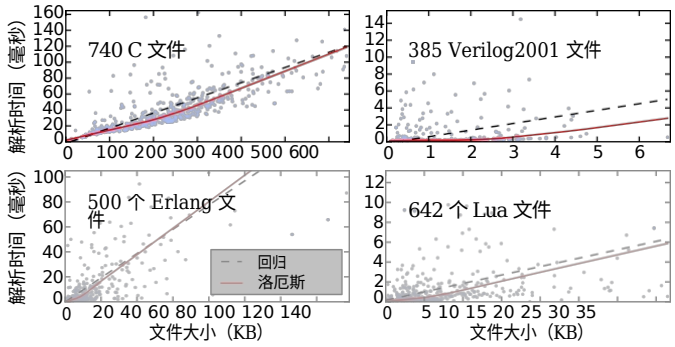


图 13.线性解析时间与文件大小的关系。线性回归（虚线）与 LOWESS 无约束曲线重合，有力地证明了线性关系。曲线是根据（底部）99% 的文件大小计算的，但放大后显示了底部 40% 的解析时间细节。

实际上反映了每条回归线，有力地证明了解析时间与输入大小之间的关系是线性的。同样的方法表明，从 C11 规范中提取的*非* SLL 语法（未显示）生成的解析器也是线性的，尽管比我们的 *SLL* 版本慢得多。

我们还没有在实践中看到非线性行为，但 *ALL(*)* 解析的理论最坏情况行为是 $O(n^4)$。对于输入 *a*、*aa*、*aaa*，以下假定的最坏情况语法的实验解析时间数据显示了四分行为、
……，一个^n （$n \le 120$ 个符号，我们可以在合理的时间内进行测试）。$S \to A\ \$, A \to aAA \mid aA \mid a$。
每个预测必须检查所有剩余输入。最后，在我们的实现中，合并两个 GSS 需要 $O(n)$，复杂度为 $O(n^4)$。

通过这些实验，我们得出结论：将语法分析转移到解析时间以获得 *ALL(*)* 强度不仅实用，而且能产生极其高效的解析器，可与 Java 编译器的手工调整递归-后裔解析器相媲美。用 DFA 记忆分析结果对这种性能至关重要。尽管理论复杂度为 $O(n^4)$，但 *ALL(*)* 在实践中似乎是线性的，并没有表现出一般算法的非凡性能或大量内存占用。

8. 相关工作

这种方法可以提高识别强度，避免静态语法分析的不可判定性问题，但却不可取，因为它存在嵌入式突变问题，降低了性能，并使单步调试变得复杂。Packrat 分析器（PEG）[9] 按顺序尝试决策产物，并选择第一个成功的。PEG 是 $O(n)$，因为它们将部分解析结果记忆化，但会出现以下问题
来自 $a \mid ab$ quirk，其中 ab 是无声的不可匹配的。
为了提高一般解析性能，Tomita [26] 引入了
GLR 是一种基于 *LR(k)*的通用算法，它可以在解析时在每个冲突的 *LR(k)* 状态下分叉子解析器，以探索所有可能的路径。富田展示了 GLR

几十年来，研究人员一直致力于提高高效但非通用的 *LL* 和 *LR* 解析器的识别能力，并提高通用算法的效率，如 Earley 的 $O(n^3)$ 算法 [8]。Parr [21] 和 Charles [4] 静态生成了 $k > 1$ 的 *LL(k)* 和 *LR(k)* 剖析器。Parr 和 Fisher 的 *LL(*)* [20] 以及 Bermudez 和 Schimpf 的 $LAR(m)$ [2] 静态计算了带有循环 DFA 的 *LL* 和 *LR* 剖析器，可以检查任意数量的前瞻性。这些解析器基于 LL-regular [12] 和 *LR-regular* [7] 解析器，而 *LL-regular* 和 LR-regular 解析器具有一般不可编码的不良特性。引入回溯功能可以极大地提高分析效率。

比 Earley 快 5-10 倍。GLR 解析的一个关键组件是图结构堆栈（GSS）[26]，它可以防止以相同的方式对相同的输入解析两次（GLR 在 GSS 上推送输入符号和 LR 状态，而 ALL(*) 则推送 ATN 状态）。(GLR 在 GSS 上推送输入符号和 *LR* 状态，而 *ALL(*)* 则推送 ATN 状态）。Elkhound [18] 引入了混合 GLR 分析器，该分析器的所有 *LR(*1) 决定都使用单一堆栈，必要时使用 GSS 来匹配输入的模糊部分。(我们发现 Elkhound 的解析器比其他 GLR 工具更快。）GLL [25] 是 GLR 的 *LL* 类似物，也使用子解析器和 GSS 来探索所有可能的路径；为了提高效率，GLL 尽可能使用 $k = 1$ lookahead。GLL 是 $O(n^3)$，而 GLR 是 $O(n^{p+1})$，其中 p 是最长语法生成的长度。

对于确定性语法，Earley 解析器可以从容地从 $O(n)$ 扩展到含混语法的最坏情况 $O(n^3)$，但对于一般应用来说，性能还不够好。*LR(k)* 状态机可以通过尽可能多的静态计算来提高此类解析器的性能。LRE [16] 就是这样一个例子。尽管进行了这些优化，但与使用深度前瞻的确定性解析器相比，一般算法的速度仍然很慢。

任意前瞻的问题在于，对于许多有用的语法来说，它是不可能静态计算的（*LL*-正则条件是不可判定的）。通过将前瞻分析转移到解析时间，*ALL(*)* 获得了处理任何没有左递归的语法的能力，因为它可以启动子解析器来确定哪条路径会导致有效的解析。与 GLR 不同的是，当所有剩余的子解析器都与单个替代生产相关联时，推测就会停止，从而计算出小型前瞻序列。为了提高性能，*ALL(*)* 使用 DFA 记录了从前瞻序列到预测结果的映射，供后续决策使用。解析过程中遇到的无上下文语言子集是有限的，因此 *ALL(*)* 前瞻语言是有规则的。Ancona *等人*[1] 也进行了解析时间分析，但他们只计算了固定的 *LR(k)* lookahead，并没有像 *ALL(*) 那样*适应实际输入。Perlin [23] 对类似 *ALL(*)* 的 RTN 进行了操作，并在解析过程中计算了 $k = 1$ 的前瞻性。

*ALL(*)*与厄利相似，都是自上而下和在解析时对语法表示进行操作，但厄利进行的是解析，而不是计算前瞻性 DFA。从这个意义上说，Earley 并不是在进行语法分析。在解析过程中，Earley 也不管理显式 GSS。相反，Earley 状态中的项目具有 "父指针"，这些指针指向其他状态，当这些状态串联在一起时，就形成了一个 GSS。Earley 的 *SCANNER* 操作与 *ALL(*)* 的*移动*功能相对应。*PREDIC*-

TOR 和 *COMPLETER* 操作对应于 *ALL(*) 闭包*函数中的 push 和 pop 操作。一个 Earley 状态是在某个绝对输入深度下可达到的所有解析器配置的集合，而一个 *ALL(*)* DFA 状态则是相对于当前决定的前瞻深度下可达到的配置的集合。与完全通用的算法不同，*ALL(*)* 算法只对输入进行一次解析，这就允许在解析过程中使用高效的 *LL* 栈。

持续推测或支持模糊性的解析策略很难使用突变器，因为它们很难撤销。缺少突变器会降低改变解析的语义谓词的通用性，因为它们无法测试解析过程中先前计算出的任意状态。*老鼠*[10]支持受限语义谓词，而 Yakker [13]则支持作为先前解析的 ter- minals 函数的语义谓词。由于 *ALL(*)* 不会在实际解析过程中进行推测，因此它支持任意的突变体和语义谓词。由于篇幅有限，我们无法在此对相关工作进行更详细的讨论；更详细的分析可参见参考文献 [20]。

9. 结论

ANTLR 4 可以为任何没有直接或隐藏左递归的 CFG 生成 *ALL(*)* 解析器。*ALL(*)* 结合了传统自上而下 *LL(k)* 分析器的简洁性、效率和可预测性，以及类似 GLR 机制的强大分析决策能力。它的关键创新之处在于将语法分析转移到了解析时间，并将分析结果缓存在前瞻性 DFA 中以提高效率。实验表明，*ALL(*)* 的性能比一般（Java）解析器高出几个数量级，在 8 种语言中表现出线性的时间和空间行为。*ALL(*)* Java 解析器的速度仅为 Java 编译器手工调整的递归-后裔解析器的 20%。理论上，*ALL(*) 的速度*为 $O(n^4)$，与 GLR 的低多项式边界一致。ANTLR 在实践中得到了广泛应用，这表明 *ALL(*) 在*提供实用解析能力的同时，并没有牺牲程序员所期望的递归-后裔解析器的灵活性和简单性。

10. 致谢

我们感谢 Elizabeth Scott、Adrian Johnstone 和 Mark Johnson 就解析算法复杂性进行的讨论。Eelco Visser 和 Jurgen Vinju 提供了测试 JS- GLR 和 Rascal 独立解析器的代码。Etienne Gagnon 生成了 SableCC 解析器。

参考资料

[1] ANCONA, M., DODERO, G., GIANUZZI, V., AND MORGAVI、M.高效构建 *LR*(*k*) 状态和表格。*ACM Trans.Program.Lang.Syst.13*, 1 (Jan. 1991), 150-178.

[2] BERMUDEZ, M. E., AND SCHIMPF, K. M. Practical arbitrary lookahead *LR* parsing.*Journal of Computer and System Sciences 41*, 2 (1990).

[3] 布鲁恩-S 和弗拉涅西奇-Z.《*数字逻辑与 Verilog 设计基础*》。McGraw-Hill ECE 系列。2003.

[4] CHARLES, P. *A Practical Method for Constructing Efficient LALR(k) Parsers with Automatic Error Recovery*.美国纽约州纽约市纽约大学博士论文，1991 年。

[5] CLARKE, K. The top-down parsing of expressions.未发表的技术报告，伦敦玛丽皇后学院计算机科学与统计系，1986 年 6 月。

[6] CLEVELAND, W. S. Robust Locally Weighted Regression and Smoothing Scatterplots.*Journal of the American Statistical Association 74* (1979), 829-836.

[7] COHEN, R., AND CULIK, K. *LR-Regular* grammars-an extension of *LR*(*k*) grammars.In *SWAT '71* (Washington, DC, USA, 1971), IEEE Computer Society, pp.

[8] EARLEY, J. An efficient context-free parsing algorithm.*Communications of the ACM 13*, 2 (1970), 94-102.

[9] FORD, B. Parsing Expression Grammars：基于识别的句法基础。In *POPL* (2004), ACM Press, pp.

[10] GRIMM, R. 通过模块化语法实现更好的可扩展性。 在 *PLDI* (2006)，ACM 出版社，第 38-51 页。

[11] HOPCROFT, J., AND ULLMAN, J. *Introduction to Automata Theory, Languages, and Computation*.Addison-Wesley, Reading, Massachusetts, 1979.

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

[13] JIM, T., MANDELBAUM, Y., AND WALKER, D. Semantics and 数据依赖语法的算法。见 *POPL* (2010)。

[14] JOHNSON, M. The computational complexity of GLR parsing.In *Generalized LR Parsing*, M. Tomita, Ed., Kluwer, Boston, 1991, pp.Kluwer, Boston, 1991, pp.

[15] KIPPS, J. *Generalized LR Parsing*.Springer, 1991, pp.

[16] MCLEAN, P., AND HORSPOOL, R. N. A faster Earley parser.在 *CC* (1996)，施普林格出版社，第 281-293 页。

[17] MCPEAK, S. Elkhound： A fast, practical GLR parser generator.Tech. rep., University of California, Berkeley (EECS), Dec. 2002.

[18] MCPEAK, S., AND NECULA, G. C. Elkhound：快速、实用的 GLR 解析生成器。In *CC* (2004), pp.

[19] PARR, T. *The Definitive ANTLR Reference：构建特定领域语言》* 。The Pragmatic Programmers, 2013.ISBN978-1-93435-699-9。

[20] Parr, T., and Fisher, K. *ll*(∗)：ANTLR 解析器生成器的基础。In *PLDI* (2011), pp.

[21] PARR, T. J. *Obtaining practical variants of LL*(*k*) *and LR*(*k*) *for k* > 1 *by splitting the atomic k-tuple*.PhD thesis, Purdue University, West Lafayette, IN, USA, 1993.

[22] PARR, T. J., AND QUONG, R. W. Adding Semantic and Syntactic Predic *Predicates* to *LL*(*k*)-pred-LL(*k*).In *CC* (1994).

[23] PERLIN, M. Earley 和 Tomita 解析的 LR 递归转换网络。*第 29 届计算语言学协会年会论文集*（1991 年），ACL '91、pp.98-105.

[24] PLEVYAK, J. DParser：GLR 解析器生成器，2013 年 10 月访问 。

[25] SCOTT, E., AND JOHNSTONE, A. GLL parsing.*Electron.Notes Theor.Comput.253*, 7 (Sept. 2010), 177-189.

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

[27] WOODS, W. A. Transition Network grammars for natural language analysis.*Comm. of the ACM 13*, 10 (1970), 591-606.

A. 正确性和复杂性分析

定理 A.1.ALL(*) *语言在联合下是封闭的。*

*证明*让谓词语法 $G_1 = (N_1, T, P_1, S_1, \Pi_1, M_1)$ 和 $G_2 = (N_2, T, P_2, S_2, \Pi_2, M_2)$ 分别描述 $L(G_1)$ 和 $L(G_2)$。为了同时适用于解析器和无扫描器解析器，假定终端空间 T 是有效字符集。假设 $N_1 \cap N_2 = \emptyset$，必要时重命名非终端。假设 G_1 和 G_2 的谓词和突变体在 S_1 和 S_2 这两个不相交的环境中运行：

$$G' = (N_1 \cup N_2, T, P_1 \cup P_2, S', \Pi_1 \cup \Pi_2, M_1 \cup M_2)$$

$S' = S \mid S_1 S_2$。那么，$L(G') = L(G_1) \cup L(G_2)$。 □

定理 A.1.如果 $w \in L(G)$，*则停用了前瞻 DFA 的非左递归 G 的*ALL(*) *分析器会识别句子 w。*

证明。如果 $w \in L(G)$，则 G 的 ATN 识别 *w*。因此，我们可以等价地证明 *ALL(*)* 是一个忠实地执行在没有前瞻性 DFA 的情况下，预测是一种直接的 ATN 模拟器。在没有前瞻性 DFA 的情况下，预测是一个直接的 ATN 模拟器：一个自顶向下的解析器，使用类似 GLR 的子解析器做出准确的解析决策，这些子解析器可以检查整个剩余输入和 ATN 子机器调用栈。

定理 A.2. *（正确性）。*如果 $w \in L(G)$，*那么非左递归 G 的* ALL(*) *解析器就能识别句子 w。*

*证明*定理 A.1 表明，*ALL(*)* 解析器可以在没有 DFA 缓存的情况下正确地识别 *w*。因此，证明的实质是表明 *ALL(*)* 的自适应前瞻 DFA 不会因为给出的预测决定与直接的 ATN 模拟不同而破坏解析。我们只需考虑无预测 *SLL* 解析的情况，因为 *ALL(*)* 只缓存这种情况下的判定结果。

*如果情况：*通过对任何给定决策 *A* 的前瞻性 DFA 状态*的*归纳。对 *A 的*第一次预测以空 DFA 开始，必须激活 ATN 模拟，以便选择使用前缀 $u \le w$ *的*替代选择 α_{ir}。由于 ATN 模拟会产生正确的预测，*ALL(*)* 分析程序会正确预测 α_i 从冷启动，然后在 DFA 中记录 $u : i$ 的映射。如果只

2. $w' = bx$，而之前所有的 $w_r = ay$，对于某个 $a \neq b$。这种情况简化为冷启动基本情况，因为不存在冷启动 $D_0 \xrightarrow{-b} D$ 边。ATN 模拟预测 α 并添加路径 $D_0 \xrightarrow{t} f_i$。
3. 对于 $u \le w'$，从 D_0 到 f_i $_{wr} = vax$ 和 $w_r = vby$，对于一些以前见过的具有共同前缀 v 和 $a \neq b$ *的* w_r。DFA 模拟从输入 v *的* D_0 到达 D。D 对 b 有一条边，但对 a 没有。ATN 模拟预测 α_i 并扩充 DFA，从 D *的* a 边开始，最终通向 f_i。

只有在这种情况下：*ALL(*)* 解析器会报告 $w \notin L(G)$ 的语法错误。假设相反，即解析器成功解析了 *w*。ration 派生序列 $(S, p_S, [], w) \rightarrow^* (S', p', [], \epsilon)$ for w 通过 *G* 的相应 ATN。但这需要 $w \in L(G)$，根据定义?因此，*ALL(*)* 解析器会报告一个 *w* 的语法错误。*ALL(*)* 前瞻缓存的准确性无关紧要，因为没有可能的路径通过 ATN 或解析器。 □

定理 A.2.*对于给定的决策 A 和剩余的输入字符串 w_r，*LL *的可行制作集总是*SLL *的可行制作集的子集。*

*证明*如果子机 *A* 的按键*移动-关闭*分析操作没有达到停止状态 p'，则 *SLL* 和 *LL 的*表现为

因此，它们共享同一套可行的生产系统。
际关系。
如果*关闭*达到决策输入规则的停止状态、p'，存在形式为 $(p', -, \gamma)$ 的配置，其中，对于有一个可行的替代方案，*i* 就是相关的生产数量。如果 ATN 模拟找到多个可行的替代方案，*i 就是*与该组替代方案相关的最小生产数量。

*归纳步骤。*假定前瞻性 DFA 能正确预判解析器在以下时间段看到的 w_r 的每个 *u* 前缀的生成结果 A.我们必须证明，从现有的 DFA 开始，*ALL(*) 可以*为 w' 的陌生 *u* 前缀正确添加一条通过 DFA 的路径。有几种情况：

1. $u \le w'$ 和 $u \le w_r$ 的前一个 w_r。根据归纳假设，前瞻 DFA 给出了 *u* 的正确答案。DFA 不会更新。

在 *LL* 预测模式下，$\gamma = \gamma_0$ ，*如果* $A = s$，*则*为单栈或空栈。在 *SLL* 模式下，$\gamma = \#$，表示无栈信息。函数*闭包*必须考虑所有可能的 γ_0 解析器调用栈。由于任何一个栈都必须包含在所有可能的调用栈集合中，因此 *LL* 闭包操作最多只能考虑与 *SLL* 相同数量的 ATN 路径。 □

定理 A.3.*对于* $w \notin L(G)$ *和非左递归* G，SLL *报告语法错误。*

*证明*与定理 6.1 中的 "*如果* "*情况*一样，*无论 AdaptivePredict* 如何选择产品，都不存在有效的 ATN 配置推导。 □

定理 A.3.如果 $w \in L(G)$，则*非左递归* G *的两阶段解析会识别句子* w。

*证明*根据 *Lemma* A.3，当 $w \notin L(G)$ 时，*SLL* 和 *LL* *的*行为相同。剩下的工作就是证明 *SLL* 预测要么在输入 $w \in L(G)$ 时表现与 *LL* 相似，要么报告语法错误，表明需要进行 *LL* 第二阶段。假设 u 和 u' 分别是 A 使用 *SLL* 和 *LL* 时的可行生成数集。根据 Lemma A.2，$u' \subseteq u$。有两种情况需要考虑：

1. 如果 $min(u) = min(u')$, *SLL* 和 *LL* 选择相同的推导。例如，$u = \{1, 2, 3\}$ 和 $u' = \{1, 2, 3\}$。$\{1, 3\}$ 或 $u = \{1\}$ 和 $u' = \{1\}$。
2. 如果 $min(u) \neq min(u')$ 则 $min(u) \notin u'$，因为 *LL* 认为 $min(u)$ 不可行。*SLL* 会报告语法错误。例如，$u = \{1, 2, 3\}$ 和 $u' = \{2, 3\}$ 或 $u = \{1, 2\}$ 和 $u' = \{2\}$。

在 u 和 u' 的所有可能组合中，*SLL* 的表现与 *LL* 相似或报告 $w \notin L(G)$ 的语法错误。 □

定理 A.4. *对于 n 个输入符号，GSS 有 $O(n)$ 个节点。*

证明 对于非终结点 N 和 ATN 状态 Q，存在 $|N| \times |Q|$ 个。如果每个语法位置都会调用每个非终端，则 $p \xrightarrow{A} q$ ATN 过渡。这就将闭合操作（不能转换终端边）的新 GSS 节点数量限制在 $|Q|^2$。*ALL(*)* 对 n 个输入符号执行 $n + 1$ 次闭包，给出 $|Q|^2 (n + 1)$ 个节点或 $O(n)$，因为 Q 不是输入的函数。 □

定理 A.4. *检查一个前瞻符号需要 $O(n^2)$ 时间。*

证明。 [2]Lookahead 是一种*移动关闭*操作，它以 D 中 ATN 配置的函数形式计算新的目标 *DFA* 状态 D'。的形式为 $(p, i,) \in D$，$|Q|$ ATN 状态和 m 备选

目前决定中的产品。*搬迁*费用不包括

的函数。D 的闭包计算 $closure(c)$
$\forall c \in D$ 和 $closure(c)$ 可以将整个 GSS 走回根（空栈）。这样，构建 D' 的成本为 $|Q|^2$ 配置乘以 $|Q|^2 (n + 1)$ GSS 节点（根据定理 A.4），或 $O(n)$ 个添加操作。添加一个配置的操作主要是

在我们的实现中，图合并的成本与图的深度成正比。*移动关闭的*总成本为 $O(n^2)$。 □

定理 A.5. *对 n 个输入符号的* ALL(*) *解析有 $O(n)^4$ 时间*

证明。 在最坏的情况下，解析器必须在预测过程中为 n 个输入符号中的每个符号检查所有剩余的输入符号，从而产生 $O(n^2)$ 的前瞻操作。根据 Lemma A.4，每次前瞻操作的成本为 $O(n^2)$，因此总体解析成本为 $O(n^4)$。 □

最初，我们预计在大型输入语料库上 "训练 "一个解析器，然后将前瞻 DFA 序列化到磁盘，以避免在后续解析器运行时重新计算 DFA。正如第 7 节所示，前瞻 DFA 的构建速度足够快，因此无需对 DFA 进行序列化和反序列化。

B.2 语义谓词评估

为清楚起见，本文所述算法对所有在生产左边缘上有语义谓词的决策都使用纯 ATN 模拟。在实践中，ANTLR 使用跟踪接受状态中谓词的前瞻性 DFA 来处理语义上下文敏感预测。如果在 *SLL* 模拟过程中对谓词进行评估时预测到了唯一的生成，那么跟踪 DFA 中的谓词就能使预测避免昂贵的 ATN 模拟。语义谓词并不常见，但对于解决某些上下文敏感的解析问题至关重要；例如，在重写左递归规则时，ANTLR 在内部使用谓词来编码运算符优先级。因此，在 *SLL* 谓词化过程中评估谓词的额外复杂性是值得的。请看第 2.1 节中的谓词规则：

```
id : ID | {!enum_is_keyword}?枚举' ;
```

只有当 !enum 是关键字时，第二种生成方式才可行。的值为真。抽象地说，这意味着解析器需要两个前瞻性 DFA，每个语义条件一个。相反，ANTLR 的 *ALL(*)* 实现创建了一个 DFA（通过

SLL 预测），边缘 D_0 f_2 其中 f_2 是一个

B. 语用学

本节将介绍与执行 *ALL(*)* 算法相关的一些实际考虑因素。

B.1 缩短热身时间

语法中的许多判定都是 *LL*(1)，而且很容易静态识别。ANTLR 并不总是在递归后裔解析器中生成 "switch on *adaptivePredict* "决策，而是尽可能生成 "switch on token type "决策。这种 *LL*(1) 优化不会影响生成解析器的大小，但会减少解析器必须计算的前瞻 DFA 的数量。

增强的 DFA 接受状态，测试 !enum 是关键字。如果出现以下情况，函数 *adaptivePredict* 将在枚举时返回结果 2 !enum 是关键字，否则会抛出一个不可行的替代例外。(一)……。

本文所述算法也不支持决策入口规则之外的语义谓词。在实践中，*ALL(*)* 分析必须评估所有可从决策入口规则到达的谓词，而无需跨过 ATN 中的终端边。例如，本文的简化 *ALL(*)* 算法只考虑以下（含糊）语法中 S 的原语π_1 和 π_2 。

$S \rightarrow \{\pi_1\}?Ab \mid \{\pi_2\}?Ab$
$A \rightarrow \{\pi_3\}?a \mid \{\pi_4\}?a$

输入 ab 与 S 的任一备选方案相匹配，实际上，ANTLR 会评估 "π_1 和（π_3 或 π_4 ）"，以测试 S 的第一个生产的可行性，而不仅仅是 π_1 。在模拟 S 和 A 的 ATN 子机后，S 的前瞻 DFA 将是 $D \xrightarrow{-a} D' \xrightarrow{-b} f$。增强的接受状态 f 预测亲
根据语义上下文 $\pi_1 \wedge (\pi_3 \vee \pi_4)$ 进行第 1 或第 2 项操作和 $\pi_2 \wedge (\pi_3 \vee \pi_4)$。为了在 *SLL* 模拟期间跟踪语义上下文，ANTLR ATN 配置
包含额外的元素 π：（p, i, Γ, π）。元素 π 带有语义上下文，ANTLR 将谓词到生产对存储在增强的 DFA 接受状态中。

B.3 错误报告和恢复

ALL(*) 预测可以任意向前扫描，因此错误的前瞻序列可能很长。默认情况下，ANTLR 生成的解析器会打印整个序列。为了恢复，解析器会消耗标记，直到出现可以跟随

现行规则。ANTLR 提供了覆盖报告和恢复策略的钩子。

ANTLR 解析器会对无效输入短语发出错误信息，并尝试恢复。对于不匹配的标记，ANTLR 会尝试插入和删除单个标记以重新同步。如果剩余的输入与当前非终端的任何产生都不一致，解析器就会消耗标记，直到找到可以合理地跟在当前非终端后面的标记为止。然后，解析器继续解析，就好像当前非终端已经成功。与 ANTLR 3 相比，ANTLR 在 EBNF 子规则的开始和 "循环 "继续测试时插入了同步检查，以避免过早退出子规则，从而改进了错误恢复。例如，请看下面的类定义规则。

```
classdef : 'class' ID '{' member+ '}' ;
member : 'int' ID ';' ;
```

成员列表中多余的分号（如 `int i;; int j;`）不应强制周围的规则 `classdef` 中止。相反，解析器会忽略多余的分号，并查找其他成员。为了减少层叠错误信息，解析器在正确匹配一个标记之前不会发出其他信息。

B.4 多线程执行

应用程序通常需要并行执行多个解析器实例，即使是同一种语言。例如，基于网络的应用服务器会使用同一解析器的多个实例解析多个传入的 XML 或 JSON 数据流。为了提高内存效率，特定语言的所有 *ALL(*)* 解析器实例必须共享前瞻性 DFA。ANTLR 生成的 Java 代码使用共享内存模型和线程进行控制，这意味着解析器必须以线程安全的方式更新共享 DFA。在其他线程为 DFA 添加状态和边的同时，多个线程可以模拟 DFA。我们的目标是线程安全，但并发也为前瞻性 DFA 构建提供了小幅提速（经验观察）。

在 Java 中实现线程安全并保持高吞吐量的关键在于避免过度锁定（同步块）。只有两种数据结构需要加锁：Q 是 DFA 状态集，而 Δ 是边集。我们的实现将状态添加（$Q \mathrel{+}= D'$）纳入 *addDFAState* 函数，该函数在测试 DFA 状态的成员性或添加状态之前等待 Q 的锁定。这并不是一个瓶颈，因为在 DFA 构建过程中，DFA 模拟可以在不锁定的情况下继续进行，因为它可以遍历边来访问现有的 DFA 状态，而无需检查 Q。

向现有状态添加 DFA 边需要细粒度锁定，但只针对特定的 DFA 状态，因为我们的实现为每个 DFA 状态维护一个边数组。我们允许多个读取器，但只有一个写入器。即使另一个线程正在争分夺秒地设置边缘，也不需要对测试边缘加锁。

如果边 $D \xrightarrow{a} D'$ 存在，模拟只需过渡到 D'。如果模拟未找到现有边，则会启动 ATN 仿真从 D 开始计算 D'，然后为 D 设置元素 $edge[a]$。两个线程可能会发现 a 上缺少一条边，并同时启动 ATN 仿真，竞相添加 $D \xrightarrow{a} D'$。

在这两种情况下，D' 都是一样的，因此只要使用同步化安全地更新特定的边阵列，就不会有危险。要遇到有争议的锁，两个或更多 ATN 模拟线程必须尝试将一条边添加到相同的 DFA 状态。

C. 消除左递归

ANTLR 支持直接左递归规则，将其重写为非左递归版本，同时消除任何歧义。例如，用于描述 arith- metic 表达式语法的自然语法就是最常见（含混）的左递归规则之一。下面的语法支持简单的模数和加法表达式。

$$E \rightarrow E\%E \mid E+E \mid \mathbf{id}$$

E 是*直接*左递归的，因为至少有一个生产以 E 开头（$\exists E \Rightarrow E\alpha$），这对自上而下的解析器来说是个问题。

用于自顶向下解析器的语法必须使用一种等价的非左递归语法，这种语法为每一级运算符优先级设置了一个单独的非终端：

$E' \rightarrow M\ (+\ M)^*$ 相加，下优先
$M \rightarrow P\ (\%\ P)^*$ 相乘，上优先
$P \rightarrow \mathbf{id}$ 初级（**id** 表示标识符）

调用的规则越深，优先级就越高。在解析时，匹配单个标识符 a 需要调用 l 个优先级的规则。

E 比 E' 更容易阅读，但左递归版本有歧义，因为输入 a+b+c 有两种解释：(a+b)+c 和 a+(b+c)。自下而上的解析器生成器（如 bison）使用运算符优先级指定符（如 %left '%'）来解决此类歧义。非左递归语法 E' 是无歧义的，因为它根据非终端嵌套深度隐式地编码了先例规则。

理想情况下，解析器生成器应支持左递归，并提供一种方法来隐式地通过语法本身来解决歧义，而无需求助于外部的优先级规范。ANTLR 通过直接左递归重写非终结词，并插入语义谓词来根据生成顺序解决歧义问题。重写过程会生成模仿 Clarke [5] 技术的解析器。

我们选择只消除直接左递归，因为一般的左递归消除会导致转换后的语法比原始语法大几个数量级[11]，而且产生的解析树与原始语法的解析树只有松散的联系。ANTLR 会自动构建适合原始左递归语法的解析树，因此原语法不会察觉到内部重组。直接左递归也涵盖了最常见的语法情况（根据长期构建语法的经验）。本文讨论的重点是算术表达式的语法，但变换

这些规则同样适用于其他左递归构造，如 C 声明符：$D \rightarrow *\ D$，$D \rightarrow \mathrm{D}\ [\ ]$，$\mathrm{D} \rightarrow D\ (\ )$，$D \rightarrow \mathbf{id}$。

消除直接左递归而不考虑模糊性是很简单的[11]。设 $j = 1...s$ 的 $A \to \alpha_j$ 为非左递归生成，$k = 1...r$ 时的 $A \to A\beta_k$ 为直接左递归生成，其中 α_j, β_k /⇒* ϵ。Re-将这些产品与

$$A \to \alpha A_1' \mid \ldots \mid \alpha$$
$$A_s' A' \to \beta A_1'$$
$$\mid \ldots \mid \beta A_r' \mid \epsilon$$

使用 EBNF 更容易理解这种转换：

$$A \to A'A''*$$
$$A' \to \alpha_1$$
$$\mid \ldots \mid \alpha_s$$
$$A'' \to \beta_1 \mid \ldots \mid \beta_r$$

或只是 $A \to (\alpha_1 \mid \ldots \mid \alpha_s)(\beta_1 \mid \ldots \mid \beta)_r*$ 。例如，左递归 E 规则变为：

$$E \to \mathbf{id}\ (\%\ E \mid +\ E)*$$

由于 a+b+c 有两个派生词，因此这个非左递归版本仍有歧义。默认的歧义解决策略会选择尽快匹配输入，从而产生 (a+b)+c 的解释。

对于使用单一运算符的表达式来说，关联性的差异并不重要，但使用混合运算符的表达式必须根据运算符的优先级来关联操作数和运算符。例如，解析器必须将 a%b+c 识别为 (a%b)+c 而不是 a%(b+c)。

为了选择适当的解释，生成的解析器必须在 (% E | + E)* "循环 "中比较前一个运算符的优先级和当前运算符的优先级。在
图 14 中，$\underline{E}$ 是 E 的临界扩展。
id 并立即返回，这样调用 E 就可以匹配 + 以形成（**a**）而不是（**b**）中的解析树。

为了支持这种比较，产品的优先序号与生产序号相反。对于 n 个原始产品，第 i^{th} 个产品的优先级为 $n - i + 1$ 。
将优先级 3 赋予 $E \to E$ % E，将优先级 4 赋予 E % E ，将优先级 5 赋予 E % E。
$E \to E + E$ 为优先级 2，$E \to$ **id** 为优先级 1。
接下来，对 E 的每次嵌套调用都需要以下信息
最简单的机制是向 E 和 require 传递一个优先级参数 pr ：$E[pr]$ 的扩展只能匹配优先级达到或超过 pr 的子扩展 。

为了执行这一点，左递归消除程序在 (% E | + E)* 循环中插入了谓词。下面是经过转换的无歧义、非左递归规则

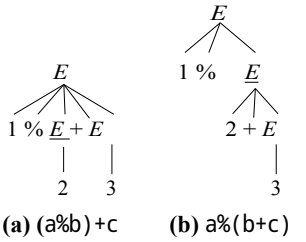
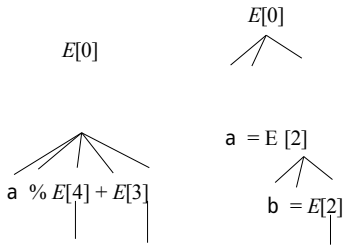


图 14a%b+c 和 $E \to$ **id** 的解析树 (% E|+ E)*



：

$$E[pr] \to \mathbf{id}\ (\{3 \geq \text{pr}\}?\ \%E[4] \mid \{2 \geq \text{pr}\}?\ + E[3])*$$

语法中其他地方对 E 的引用变为 E[0]；例如，$S \to E$ 变为 $S \to E$[0]。输入 a%b+c 会产生图 15 **(a)** 所示的 E[0] 解析树。

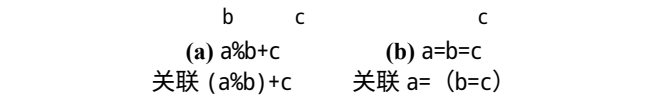


图 15.非终端的非终端扩展树

E[*pr*] → **id** ({3 ≥ pr}? *%E*[4] | {2 ≥ pr}? + *E[*3])*

生产"{3 ≥ pr}?%E[4]"在模运算前值 3 达到或超过参数 *pr 时*是可行的。第一次调用 *E* 时 *pr* = 0，由于 3 ≥ 0，所以解析器在 *E*[0] 中展开"%*E[*4]"。

解析调用 *E*[4] 时，谓词 {2 ≥ pr}? 失败，因为 + 运算符的优先级太低：2 /≥ 4。因此，*E*[4] 与 + 运算符不匹配，推迟了
给调用 *E*[0] 的人。

转换的一个关键因素是 *E* 参数*的*选择，在本语法中为 *E*[4] 和 *E*[3]。对于像 % 和 + 这样的左关联运算符，右操作数比运算符本身多一个前置级别。这保证了对右操作数调用 *E 时，*只匹配优先级更高的操作。

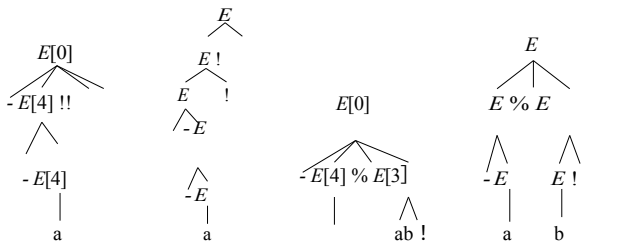
对于右关联运算，*E* 操作数的优先级与当前运算符相同。下面是表达式语法的一个变体，用右关联赋值运算符代替了加法运算符：

E → *E* % *E* | *E* =^{right} *E* | **id**

其中符号 =$_{right}$ 是 ANTLR 实际语法"|<assoc=right> *E* =^{right} *E* "的简写。a=b=c 的解释应该是右关联的，即 a=（b=c）。要得到这样的
在社会性方面，转换后的规则只需在右操作数（*E*[2] 与 *E*[3]）上有所不同：

E[pr] → **id** ({3 ≥ pr}? % *E*[4] | {2 ≥ pr}? = *E[*2])*

如图所示，*E*[2] 扩展可以与赋值匹配。
图 15 的 **(b)**，因为谓词 2 ≥ 2 为真。
一元前缀和后缀运算符被硬连线为右-
和左关联。考虑下面的 *E*



(a) --a!
((-(-a))!)!

(b) -a!

(c) -a%b!
(-a)%(b!)

(d) -a%b!

图 16. $E \rightarrow$ -E | E ！ | E%E | **id**

带有否定前缀和 "非 "后缀运算符。

$E \rightarrow -E \mid E$ ！ $\mid E$ % E | **id**

前缀运算符不具有左递归性，因此会进入第一个子规则，而具有左递归性的后缀运算符则会像二元运算符一样进入谓语循环：

E[pr] → (**id** | - E[4])
({3 ≥ pr}? ! | {2 ≥ pr}? % E[3])*

图 16 展示了 ANTLR 生成的解析器所产生的规则调用树（调用堆栈记录）和相关解析树。连续生产中的一元操作都具有相同的相对优先级，因此都是 "eval-"。
uated "的顺序。例如，$E \rightarrow -E \mid +E$ | **id** 必须将 -+a 解释为 -（+a），而不是 +（-a）。

.不符合要求的左递归结构 $E \rightarrow E$ 或 $E \rightarrow \epsilon$ 将使用典型的消去技术进行重写，而无需考虑歧义问题。

因为需要解决谓词的歧义并计算 A 参数、

C.1 左递归消除规则

为了消除非终结词中的直接左递归并解决歧义，ANTLR 会寻找四种模式：

$A_i \rightarrow A\alpha\, A_i$ *（二元和三元算子）*
$A_i \rightarrow A\alpha_i$ *（后缀运算符）*
$A_i \rightarrow \alpha\, A_i$ *（前缀运算符）*
$A_i \rightarrow \alpha_i$ *（主要或 "其他"）*。

在需要时，产品的下标 A_i 可以捕捉到原始语法中的产品编号。隐藏和间接左向递归会导致 ANTLR 的静态左向递归错误。从 G 到 G' 的转换过程是

1. 去掉直接左递归非终端引用
5. 重写 A 二进制、三元和前缀原语之间的引用导管为 A[*nextpr*(*i*，*关联*)]，其中 *nextpr*(*i, assoc*) = {*assoc* == *left* ? *i* + 1 : i}
6. 重写任何其他 P (包括 A' 和 A''）为 A[0]
7. 将原来的 A 规则改写为 A[pr] → $A\,A'''$*

实际上，ANTLR 使用的是 EBNF 形式，而不是 $A\,A'''$* 。

2. 将前缀、初级产品收集到新创建的 A'

3. 将二元、三元和后缀制作收集到新创建的 A''

4. A 中的前缀生成$''$，带有先例检查半语谓词 $\{pr(i) >= pr\}$? 其中 $pr(i) = \{n - i + 1\}$