arXiv:2601.05012v1 [cs.PL] 8 Jan 2026

松鼠解析器

一种支持左递归和最优错误恢复的线性时间 PEG Packrat 解析器

LUKE A. D. HUTCHISON

我们介绍 squirrel 解析器：一种 PEG packrat 解析器，能够直接处理所有形式的左递归，并实现最优错误恢复；即使存在任意数量的错误，其时间复杂度仍与输入长度保持线性。传统的递归下降解析器处理左递归时，需要重写文法或采用复杂的算法扩展。我们从第一性原理推导出一种最简算法：通过按位置状态跟踪进行循环检测，并在每个左递归循环中以 $O(1)$ 的代价在后代与祖先递归帧之间传递信息；通过迭代展开进行不动点搜索。对于错误恢复，我们推导出一组由四条公理和十二项约束组成的条件。最优错误恢复设计必须满足这些条件，才能确保完备性、正确性、性能最优以及行为直观。我们利用约束满足机制搜索所有可能方案，最终得出一种经证明最优且稳健的错误恢复策略，同时保持完美的线性性能。

CCS 概念： • **软件及其工程** → **解析器**； • **计算理论** → 文法与上下文无关语言

附加关键词与短语：语法分析、递归下降语法分析、自顶向下语法分析、PEG 解析、PEG文法、Packrat 解析、优先级、结合律、左递归文法、左递归、记忆化

ACM 引用格式：
Luke A. D. Hutchison。2025。松鼠解析器：一种支持左递归和最优错误恢复的线性时间 PEG Packrat 解析器。*ACM程序设计语言与系统汇刊* 0，0，文章 0（2025 年 8 月），17 页。https://doi.org/00.0000/0000000.0000000

1 引言

递归下降语法分析 [6, 7] 因其简单性，以及文法与实现之间的直接对应关系，至今仍被广泛使用。记忆化使线性时间语法分析成为可能 [8],，Ford 将其形式化为 Packrat 解析 [4, 5]。[1]解析表达式文法（PEG）为贪心模式匹配提供了一种无歧义的形式体系，以确定性识别取代生成式产生式。

1.1 解析表达式文法形式体系

PEG语法 G 由规则组成 $A \leftarrow e$，其中 A 是规则名称， e 是解析表达式。表达式通过运算符组合（表 1）： Seq（序列 $e_1 e_2$）、First（有序选择 e_1/e_2）、OneOrMore（重复 $e+$）和 NotFollowedBy（负前瞻 $!e$）。派生运算符包括 Optional（$e? \equiv e/\epsilon$）、ZeroOrMore（$e* \equiv e+/\epsilon$）和 FollowedBy（$\&e \equiv !!e$）。终结符用于匹配字符序列； ϵ 在任意位置匹配零个字符。

Packrat 解析通过在每个（子句，位置）对上记忆匹配结果，实现了 $O(n \cdot |G|)$ 复杂度。然而，标准 packrat 解析器面临两项难以通过简单方法解决的根本性局限。

[1]注意：Ford 的 2002年论文 [4] 讨论了 packrat 解析器；解析表达式文法在 2004年论文 [5]中首次提出。

作者联系方式：Luke A. D. Hutchison，luke.hutch@alum.mit.edu。

允许免费制作本作品的全部或部分内容的数字副本或纸质副本，供个人使用或课堂教学使用，但前提是不得为营利或商业利益而制作或分发副本，且副本须附有本声明以及首页上的完整引文。对于本作品中由作者以外人士拥有版权的部分，必须尊重其版权。注明出处后允许摘录。若要以其他方式复制、重新发表、发布至服务器或重新分发至列表，须事先获得明确许可和/或支付费用。请向 permissions@acm.org 申请许可。© 2025 版权所有者/作者保留版权。出版权已授权给 ACM。提交至 ACM 的稿件

Name	子句	记法
Seq	2+	$e_1\ e_2\ e_3$
First	2+	$e_1 \mid e_2 \mid e_3$
一个或多个	1	$e+$
不被后续匹配	1	$!e$

(a) 主要运算符

Name	记法	等价
可选	$e?$	e/ϵ
零次或多次	$e*$	$e+/\epsilon$
后跟	$\&e$	$!!e$

（b）派生运算符

表 1. 以子句 e 和 e_i以及空字符串 ϵ 定义的解析表达式文法运算符。

1.2 动机：现有方法的缺陷

左递归。 像 $E \leftarrow E + T \mid T$ 这样的规则会导致无限递归：在位置 p 匹配 E 时，会立即递归到 p 处的 E，而此时尚未消耗任何输入。以往的解决方案存在显著缺陷：

- **文法重写** [1]：将左递归规则转换为右递归等价规则，从根本上改变解析树结构，并且需要进行后处理才能还原语义意图。
- **优先级爬升** [9, 11]：处理运算符优先级，但每个优先级层级都需要单独的机制，从而增加实现复杂度。
- **迭代算法** [13, 14]：通过种子增长方法处理直接左递归，但控制流复杂，且常常无法处理间接递归或相互递归循环。
- **GLL/ALL(*) 扩展** [2, 3, 10, 12]：支持左递归，但会牺牲线性性能，或引入非确定性状态探索。

错误恢复。 Packrat 解析器会提交到第一个匹配的备选项（贪心选择），因此难以实现可靠的错误恢复 [11]。现有方法要么放弃记忆化保证，要么要求通过文法注释指定恢复点，从而削弱了解析表达式文法的简洁性。

这两个问题有一个共同点：以往的解决方案会损害解析树语义、牺牲 $O(n \cdot |G|)$ 复杂度，或引入与问题概念上的简洁性不相称的算法复杂度。本文证明，这种权衡并非必要。

2 左递归：形式化推导

2.1 问题形式化

定义 1（解析器状态）。解析器状态是一个二元组(C, p)，其中 $C \in G$是一个子句， $p \in [0, n]$ 是一个输入位置。

定义 2（左递归循环）。如果在位置 p对 $C.match(p)$ 求值时，在消耗任何输入之前递归调用了 $C.match(p)$，则子句 C 在该位置是左递归的。

在位置 p 对 $E \leftarrow E + T \mid T$进行朴素递归下降，会立即递归到位置 p的 E，从而导致无限递归。Packrat 记忆表无法阻止这种情况：记忆化需要先得到完整结果才能缓存，但递归调用发生在结果计算期间 。

2.2 从第一性原理推导

定理 1（不动点存在性）。对于位置 p处的任意左递归循环，都存在一个不动点：一个无法再匹配更多输入的终止状态。

证明。设输入长度为 n。每次成功迭代都必须消耗 ≥ 1 个字符（否则会发生无限递归）。经过 $\leq n$ 次迭代后，所有字符均已消耗，该循环以不匹配终止。这个不匹配就是不动点。□

理论 引理 2（自底向上的必要性）。左递归匹配必须从不动点开始进行自底向上计算。

证明。满足循环延续条件的迭代次数 k 无法预先确定：它取决于输入内容。自顶向下求值要求在递归之前知道 k；自底向上计算则通过反复尝试发现 k。不动点（定理 1）提供基本情况；扩展逐步向上进行，直到停滞。□

定理 3（长度单调递增）。每次成功的扩展迭代都会产生一个严格长于前一次迭代的匹配结果。

证明。文法是有限的；每个子句组合都是确定性的（PEG 语义）。要使迭代 i 在迭代 $i-1$之后继续成功，它必须匹配前一个种子无法匹配的额外输入。因此 $len_i > len_{i-1}$，否则该迭代将失败。□

2.3 算法组件

这些定理需要三种机制：

（*1*）通过状态跟踪进行循环检测。维护每个位置的递归状态。匹配 C位于 p时，检查（C，p）是否在当前调用栈中。如果是，则返回不匹配（不动点），并在的备忘录条目中将 foundLeftRec = **设为 true**，对应（C，p）。这次 O（1）写入会将循环条件从后代递归帧传递给祖先递归帧。

（*2*）通过种子增长进行迭代展开。递归调用返回后，检查 foundLeftRec。如果为 true，则进入扩展阶段：反复重新调用 $C.match(p)$，将上一轮缓存的结果用作下一轮的种子。直到$len_{i+1} \leq len_i$（不再有进展）时终止。

（*3*）带版本标签的备忘录。每个（C，p）备忘录条目都包含版本标签 memoVersion[p]。每次扩展迭代都会递增版本计数器。缓存查找会验证 entry.version == memoVersion[p]。这样可以确保内部子句使用当前迭代的种子，而不是之前迭代的过时结果，从而保持长度单调性（定理 3）。

定理 4（正确性）。该三组件算法能够正确计算所有循环类型（直接递归、间接递归、相互递归）的左递归匹配。

证明。（概要）循环检测通过调用栈成员关系识别所有左递归状态。迭代展开在有限输入域上通过克莱尼迭代计算最小不动点。版本标记在各次迭代间维护缓存一致性，确保单调进展。由于输入长度有限，算法必然终止。□

定理 5（复杂度保持）。左递归处理为每个(1) (C, p) 对增加 O的开销，同时保持整体 $O(n \cdot |G|)$ 复杂度。

证明。循环检测：每次调用耗时 $O(1)$。扩展：最多迭代 n 次（受输入长度限制），每次执行一次记忆化匹配。对所有位置进行摊销：扩展总开销为 $O(n)$。结合 packrat 的基础开销 $O(n \cdot |G|)$：$O(n \cdot |G|) + O(n) = O(n \cdot |G|)$。□

松鼠解析器实现了这一最简算法。与以往的工作不同，[13, 14], 它能够统一处理所有类型的递归，无需进行文法预处理，并保持线性性能。

3 实现

图1展示了完整的实现。为清晰起见，代码采用类 C 语法。关键数据结构：

- 匹配：表示解析结果，包含子句、位置、长度和子匹配。
- 备忘录条目：按（子句，位置）缓存结果，并包含在递归路径中、发现左递归和版本等左递归跟踪字段。
- MemoTable：将（子句，位置）映射到备忘录条目。可用密集二维数组或稀疏哈希表实现。
- 解析器：维护 cycleDepthForPos[]，用于版本标记。

3.1 算法机制

实现将所有簿记信息整合进 MemoTable。每个 MemoEntry 都会在缓存的匹配结果上附加左递归跟踪状态，从而无需使用辅助数据结构。

核心算法。 MemoEntry.match 方法通过四种机制实现该算法：

(1) 缓存验证。 检查缓存结果是否仍然有效：match= null! 且 cycleDepth == parser.cycleDepthForPos[p]。如果有效，则返回缓存的匹配结果；否则，继续重新计算。

(2) 循环检测。 匹配 (C, p)时，检查 在递归路径中。如果为真，(C, p) 就在调用栈上：这就是不动点。将 `foundLeftRec = 设为 true` （通过 $O(1)$ 标志写入，将循环条件从后代传递给祖先），以 不匹配作为种子值并返回。共享的 `MemoEntry` 使这种传递无需辅助数据结构即可实现。

（3）迭代展开。递归调用返回后，检查 foundLeftRec。如果为 true，则进入扩展循环：重新调用 rule.match(parser, pos)，并将上一轮迭代的缓存结果用作种子。比较新匹配的长度与上一轮的长度；如果 newMatch.len > match.len，则更新缓存，并递增 parser.cycleDepthForPos[p]。当长度不再增加时，终止循环。

（4）版本验证。cycleDepth 字段存储上次缓存更新时的版本。在扩展过程中递增parser.cycleDepthForPos[p]，会使位置 p处的所有条目失效，从而强制使用更新后的种子重新计算。这确保了各轮扩展迭代之间的缓存一致性。

PEG 子句实现。每个 Clause 子类都会通过调用未记忆化的子句（直接调用方法）、收集结果，并返回 MISMATCH 或新的 Match，来实现 match(parser, pos)。例外是 RuleRef，它会通过 parser.match() 进行递归，从而触发记忆化。这种规则级（而非子句级）的记忆化在降低开销的同时，将复杂度维持在 $O(n \cdot |G|)$。

3.2 函数式纯粹性与版本标记

标准的 packrat 记忆化假设引用透明性成立：match(C, p) 每次调用都会返回相同结果。左递归违背了这一假设：每次扩展迭代都会在相同的 (C， p) 对上产生严格更长的匹配。

从概念上说，记忆化应以 (子句，位置，版本) 为键，其中版本记录扩展迭代的次数。每次迭代都对不同的输入（上一次迭代缓存的结果）进行操作，从而恢复纯粹性：每个 (C， p， v) 三元组都映射到唯一结果。

从物理上来说，存储所有版本并无必要。只有最新的 匹配结果才重要。cycleDepth 字段实现了版本标记：当 cycleDepth $<$ parser.cycleDepthForPos[p] 时，缓存的结果已过期，需要重新计算；当两者相等时，结果仍然有效。递增 parser.cycleDepthForPos[p] 会使位置p处的所有条目失效，迫使下一次扩展迭代重新计算。

该机制实现了三个特性：

- **线性扩展成本：**过期条目会触发重新计算，但循环检测（inRecPath）可防止冗余的递归下降。每次迭代都会执行 $O(|G|)$ 次记忆化查找。
- **同源规则去重：**对同一左递归规则的多个引用（例如， A in $(A \leftarrow (A\text{'x'})/(A\text{'y'})/B)$）会共享当前迭代的记忆化结果。
- **向后兼容性：**不存在循环时，cycleDepthForPos 保持为零，算法会退化为标准的 packrat 解析。

3.3 左递归覆盖

图 2 展示了对所有类型的左递归进行统一处理：

(a)-(b) 与循环长度无关： 直接递归和间接递归的处理方式相同，不同于仅限于处理直接递归的算法 [12]。

(c)-(d) 输入相关递归：由 `First` 或 可选 选择所决定的循环。

(e)-(f) 交织循环： 多个相互交织的循环（示例取自 [1]）。在 (e) 中：三个循环（$E \leftarrow F \leftarrow E$）、（$G \leftarrow H \leftarrow G$）、（$E \leftarrow F \leftarrow G \leftarrow E$）。在 (f) 中：（$L \leftarrow P \leftarrow L$）和（$P \leftarrow P$）。

(g)-(i) 结合律： 左结合（g）和右结合（h）文法会生成正确的树结构。结合律存在歧义的文法（i）会生成右结合结构，这与解析表达式文法语义及其他左递归处理器一致 [14]。

4 复杂度分析

定理 6（时间复杂度）。*squirrel* 解析器的运行时间为 $O(n \cdot |G|)$，其中 $n = |input|$，且 $|G|$为文法大小。

证明。将工作分为三个部分：

（1）记忆化匹配。标准 packrat 保证每个（C, p）对最多尝试匹配一次：总计 $|G| \cdot n$。每次记忆化调用执行 $O(1)$ 的工作（缓存查找、循环检测），不包括递归子调用。总计：$O(n \cdot |G|)$。

（2）扩展迭代。在位置 p 的每次扩展迭代都会递增 parser.cycleDepthForPos[p]。每次迭代都必须消耗 ≥ 1 个字符（长度单调递增，见第 2 节中的定理 3）。因此，所有位置的扩展总数为 $\leq n$。每次扩展执行一次记忆化匹配：工作量为 $O(|G|)$。总计：$O(n \cdot |G|)$。

（3）子句递归。非 RuleRef 子句会执行非记忆化递归。每次子子句调用贡献 $O(1)$ 的工作量。解析树有 $O(n)$ 个节点（每个节点消耗 ≥ 0 个输入字符，总计 $\leq n$）。子句递归总计：$O(n)$。

合并：$O(n \cdot |G|) + O(n \cdot |G|) + O(n) = O(n \cdot |G|)$。 □

定理 7（空间复杂度）。该算法需要 $O(n \cdot |G|)$ 空间。

证明。备忘录表存储 $\leq |G| \cdot n$ 个条目，每个条目的大小为 $O(1)$。cycleDepthForPos 数组：$O(n)$。递归深度上限为 $O(n)$（每次递归调用最终都必须消耗输入）。总计：$O(n \cdot |G|)$。□

5 错误恢复

语法错误恢复将解析器从二元分类器（匹配/不匹配）转变为分析器，使其在匹配语法结构与输入的同时，识别长度最小的错误跨度。本节将完整地形式化推导一种有界错误恢复机制，使复杂度保持为 $O(n \cdot |G|)$。我们陈述所需的公理和约束，推导出解决方案，并证明其完备性、正确性和最小性。

5.1 基础公理

以下是 PEG packrat 解析器的不可变属性：

公理 1（Packrat 不变量）。记忆化解析器针对每个不同的解析上下文，对每个 （$clause, position$）对最多求值一次。违反这一点会引入指数时间复杂度。

公理 2（PEG 有序选择语义）。解析表达式文法使用有序选择： $First$（$[A, B]$）一旦匹配成功，就会提交于 A，绝不会再尝试 B。这使 *PEG* 区别于存在歧义的 *CFG* 解析器。

公理 3（单调消耗）。一旦 *packrat* 解析器消耗了输入字符 0 到 k，并返回结果，再回溯到位置 $< k$ 重新解析，就会违反记忆化保证，并可能导致非终止。

此外，松鼠解析器还引入了以下公理来处理左递归：

公理 4（左递归不动点）。左递归规则从一个基本情况开始迭代扩展，不断扩展解析结果，直至达到不动点。每次扩展都必须在前一次匹配的基础上有所延伸，以确保取得进展并最终终止。

5.2 设计约束

根据解析表达式文法的公理，以及在保持线性性的同时实现直观错误恢复行为这一目标，我们推导出完整、正确且精简的错误恢复机制必须满足以下约束：

5.2.1 线性约束。

约束 1（单遍解析）。解析器必须在每个阶段（即标准解析或恢复）中严格按照从左到右的顺序扫描输入一次，不得回溯到较早的位置。这可保持 $O(n)$ 字符检查成本。

约束 2（记忆化有效性）。每个 $(clause, position)$ 对在每个恢复阶段中最多只能计算一次。缓存条目必须在其所属阶段内始终有效。

约束 3（有界恢复）。恢复操作必须在总计 $O(n \cdot |G|)$ 步内终止。因此，每次恢复都必须至少消耗一个输入字符，从而将恢复激活总次数限制在 n。

5.2.2 可组合性约束。

约束 4（子句独立性）。每个语法子句的行为都必须能够独立定义。*ASeq* 的恢复行为只能取决于其子句的行为，而不能取决于 *Seq* 在更大文法中的出现位置。

约束 5（引用透明性）。规则引用的行为必须与所引用规则的内联展开完全相同。区分 $Ref('E')$ 与直接写出 E的子句，纯粹是为了组织安排。提交给 ACM 的稿件

5.2.3 正确性约束。

约束 6（完整性传播）。每个解析结果都带有一个 *isComplete* 标志，用于指示语法分析是否已消耗文法允许的全部输入。此标志必须根据组合式布尔代数，在解析表达式文法运算符中正确传播：

$$Seq : complete \iff \bigwedge_i children[i].complete \wedge \neg usedRecovery$$

$$First : complete \iff chosen.complete$$

$$Repetition : complete \iff \neg truncated \wedge \bigwedge_i children[i].complete$$

约束 7（阶段隔离）。阶段 *1*（不启用恢复的发现阶段）的缓存条目不得在阶段 *2*（启用恢复）中重复使用，否则会导致不同的结果。缓存验证必须检查阶段一致性。

约束 8（边界保留）。文法层级 L 上的恢复不得消耗属于层级$L+1$的输入。解析 Seq（$[A, B, C]$）时， A 的恢复不得跳过 B预期位置的起始处。

约束 9（非级联错误）。每个错误都有范围有限的受影响区域。错误不得全局传播：

$$\forall error\ e: \quad e.affectedRegion \subseteq [e.pos, e.pos + e.len] \tag{1}$$

约束 10（左递归与错误恢复分离）。在左递归种子阶段（检测到循环时），必须禁止恢复操作。种子不匹配是不动点迭代的控制信号，而非解析错误。

约束 11（可见性）。恢复可以跳过可见输入字符（将其标记为错误），也可以在文件结束时删除文法元素（接受不完整输入），但不能在语法分析过程中间插入文法元素，因为这会通过不可见插入重组可见结构：

$$\forall edit \in Recovery: \quad (edit = InputSkip) \vee (edit = GrammarDelete \wedge pos = EOF) \tag{2}$$

约束 12（解析树跨度不变式）。每个解析结果都必须完整覆盖从位置 *0* 到末尾的输入：

$$\forall result: \quad result.len = |input| \tag{3}$$

完全解析失败时，返回一个覆盖全部输入的 *SyntaxError* 节点。部分文法匹配会用一个尾随 *SyntaxError* 节点包裹，以捕获未消耗的输入。这一不变式确保所有解析树在结构上完整，并且可以统一处理，无需针对未消耗的输入进行特殊处理。

5.3 数学见解

在介绍解决方案之前，我们先陈述由约束交互所产生的数学性质：

5.3.1 左递归的定点结构。

定理 8（左递归作为不动点迭代）。左递归展开计算的是有限域上某个单调函数的最小不动点。

证明。设 $F(r) = clause.match(parser,\ pos)$，其中 $clause$是左递归的。定义偏序 $r_1 \sqsubseteq r_2$为 $r_1.len \leq r_2.len$ （较长的匹配优先）。

(1) 单调性：如果 $r_1 \sqsubseteq r_2$，那么 $F(r_1) \sqsubseteq F(r_2)$，因为解析表达式文法子句在给定更长的种子时只会增长或保持不变。**(2) 有限域**：匹配长度受 $|input|$ 限制，因此形成有限链 $0 \leq len_0 < len_1 < \ldots < |input|$。**(3) 终止性**：当 $r_{i+1} \sqsubseteq r_i$（没有进展）时，迭代 $r_0 = \bot, r_{i+1} = F(r_i)$ 终止。**(4) 最小不动点**：根据 Kleene 不动点定理，这会计算出 $lfp(F) = \bigvee_{i=0}^{\infty} F^i(\bot)$。

LR 扩展循环是命令式形式的 Y 组合子。 □

5.3.2 完备性的布尔代数。

定理 9（完备性复合）。*isComplete* 标志在合取运算下构成一个布尔代数，解析表达式文法运算符则作为代数运算。

证明。定义 $\wedge_{complete}$ 为组合完备性算子：

$$complete \wedge_{complete} complete = complete$$

$$complete \wedge_{complete} incomplete = incomplete$$

$$incomplete \wedge_{complete} incomplete = incomplete$$

它满足布尔代数公理（结合律、交换律、幂等律）。解析表达式文法运算符实现了这些运算：

- *Seq*: $\wedge$ 遍历子节点
- *First*: 恒等（继承自选定的备选项）
- *Repetition*: $\wedge$ 遍历迭代，并使用 $truncated \Rightarrow incomplete$

该代数定义良好，因为完备性具有组合性：运算符的完备性仅取决于其直接子节点的完备性，而不取决于全局解析状态。□

5.3.3 阶段对偶性。

定理 10（两阶段最小性）。对于具备缓存有效性的有界错误恢复，恰好需要两个阶段。

证明。 **必要性（下界）：**

（1）阶段 1（发现）：必须确定哪些位置需要恢复。若没有这一阶段，就无法知道哪些位置需要进行错误纠正。（2）阶段 2（恢复）：必须根据阶段 1 的发现应用纠正。不能将其与阶段 1 合并：恢复决策需要前瞻（了解前方有哪些解析机会）。

充分性（上界）： 两个阶段以上并无必要：

（1）阶段 1 找出所有不完整性边界（∃ 不完整）（2）阶段 2
利用这些边界（∀位置，尝试恢复）（3） ∃/∀分离已完整；无需
第三个量词

增加阶段只会重复阶段 1/2 的工作，或破坏线性性（需要多次扫描输入） . □

引理 11（阶段标志充分性）。恢复阶段可以用一个布尔标志表示，因为恰好有两个阶段：发现（恢复阶段 = 假）和恢复（恢复阶段 = 真）。

5.4 解决方案

现在，我们推导出满足上述公理和约束的唯一错误恢复设计。

5.4.1 匹配结果类型系统。 根据约束 6，解析器结果必须携带 isComplete 标志。根据约束 10，它们还必须携带 isFromLRContext 标志。我们定义一种统一匹配类型：

关键设计决策：所有匹配类型（终结符、单个子节点、多个子节点）都统一为一个类。它们的区别仅在于 |子节点|：

- 终结符：子节点 = []
- 单个子节点：子节点 = [c]
- 多个子节点：子节点 = $[c_1, \ldots, c_n]$

这种统一简化了类型系统，同时保留了所有必要的信息。

5.4.2 带阶段跟踪的记忆化。 根据约束 7 和引理 11，记忆化条目必须记录其计算时所处的阶段：

关键设计决策：根据引理 11，我们使用单个布尔值 cachedInRecoveryPhase，而非整数计数器。这已经足够，因为只有两个阶段。

缓存验证逻辑：

5.4.3 通过显式参数传播界限。 根据约束 8，重复项在消耗属于兄弟节点的输入之前，必须先检查界限。由此可知，界限信息必须通过任意嵌套层级传播。

关键设计决策：将边界表示为显式参数，而不是环境状态。这遵循显式数据流原则：

这消除了隐藏状态变更，使数据流变得显式。

5.4.4 两阶段算法。 根据定理 10，我们推导出完整的解析算法：

各阶段通过单个位进行通信：isComplete。这是约束 6 所要求的最小通信通道。

5.5 完备性证明

定理 12（完备性）。 对于所有能够通过有限次字符跳过来完成语法分析的输入，两阶段算法都能成功。

证明。设输入 I 可通过跳过序列 $S = [s_1, \ldots, s_k]$ 进行解析，其中 s_i 是位置 p_i 处的字符跳过。

阶段 1： 尝试进行干净解析。对于每个 $s_i \in S$，解析将在位置 p_i 变得不完整（无法匹配预期子句）。将结果标记为不完整。

阶段 2： 启用恢复。在每个 p_i处：

(1) 恢复搜索尝试跳过 s_i(2) 跳过后，干净解析继续进行（根据 S的定义）

(3) 有界搜索保证我们会在 MAX_SKIP 上限内探索跳过 s_i

通过对 $|S|$ 进行归纳法，阶段 2 成功应用了 S 中的所有跳过操作，并生成完整的解析。**终止性：** 每次跳过都会消耗 ≥ 1 个字符。最多可能进行 $|I|$ 次跳过。恢复会在 $O(n)$ 步内终止。 □

5.6 正确性证明

Th定理 13（正确性）。该算法满足所有约束 $C1$–$C10$，并生成可靠的解析树。

证明概要。我们逐一验证各项约束：

完备性 (C6)： isComplete 通过布尔代数计算得出（定理 2）。序列、First、重复分别实现$\wedge$、id、$\wedge$。按构造即正确。

阶段隔离 (C7)： cachedInRecoveryPhase 用于跟踪阶段。缓存有效性检查阶段匹配。完整结果与阶段无关。由引理 11 可知正确。

引用透明性 (C5)： Ref 子句会绕过备忘录表，直接调用目标规则。其行为与内联展开相同，满足约束 5。

其余约束（C1–C4、C8–C11）： 类似的直接验证。每项约束都对应一个特定的算法组件。构造性证明。

可靠性： 解析树节点对应于成功解析的输入或明确的错误注释（SyntaxError 节点）。不会发生不可见的结构修改；可见性约束 11 得到强制执行。□

5.7 最小性的证明

现在我们证明算法的每个组件都是必要的。移除任何组件都会违反至少一项约束。

定理 14（最小性）。该算法是最小的：每个字段、标志和机制对于正确性都不可或缺。

反证法。我们证明每个组件的必要性：

在 在递归路径中 布尔值的必要性：**假设：** 移除在递归路径中的标记；通过调用栈深度检测左递归。

矛盾： 调用栈是按调用实例维护的，而不是按（子句，位置）对维护的。多个子句可能在同一位置调用同一规则。调用栈会造成误报，错误地检测出循环。这违反了公理 1（Packrat 不变量）和公理 4（左递归不动点）。

在 *foundLeftRec* 标志的必要性：**假设：** 在循环结束后，根据迭代次数推断左递归。

矛盾： 必须在循环过程中 了解左递归状态，才能：(1) 在迭代时使缓存失效；(2) 使用 isFromLRContext 标记结果。无法等到循环结束后再判断。没有其他可行的检测方法。

按位置维护 *memoVersion*的必要性：**假设：** 使用单一的全局版本计数器。

矛盾： 不同位置有各自独立的左递归展开。位置 p_1 处的左递归展开会递增全局计数器，使无关位置 p_2 处的缓存失效。这会导致 $O(n^2)$ 性能下降。违反了约束 2（记忆化有效性）。

isFromLRContext 标志的必要性：**假设：** 允许在所有匹配层级进行恢复。

矛盾： 在左递归种子上进行恢复（循环检测期间）会破坏不动点迭代。左递归必须在尝试恢复前完全展开。该标志对于阻止在左递归中间结果上进行恢复至关重要。违反了约束 10。

两阶段架构的必要性：根据定理 10，恰好需要两个阶段。一个阶段缺乏用于恢复决策的前瞻。三个+ 阶段要么是多余的，要么会违反线性性。稿件已提交至ACM

必要性 *probe()* 方法： **假设：** 未进行阶段分离的边界检查。

矛盾： 边界检查需要“干净的”前瞻（不进行恢复）。如果不进行探测，阶段 2 中的边界检查就会使用恢复，从而错误地跳过后续内容。这违反了约束 8。

线性恢复搜索的必要性： **假设：** 使用二分搜索确定跳过距离。

矛盾： 二分搜索要求满足单调性：如果在 k 处匹配，那么在 $k+1$ 处也必须匹配。文法可能要求精确位置（例如分隔符）。这里不存在单调性。对于这个问题，线性搜索是最优的。

所有组成部分都已证明是必要的。该算法是最精简的。 □ □

5.8 复杂度分析

定理 15（线性时间复杂度）。两阶段算法在 $O(n \cdot |G|)$ 时间内运行，其中 $n = |input|$ 且 $|G| = |grammar|$。

证明。**阶段 1：** 每个（$clause$，pos）对最多计算一次（packrat 不变量）。最多进行 $|G| \cdot n$ 次计算。每次计算为 $O(1)$，不包括递归子调用。总计： $O(n \cdot |G|)$。

阶段 2： 相同的记忆化保证。额外的恢复搜索有界：

- 每次恢复消耗 ≥ 1 个字符（跳过或成功解析）
- 最多可能进行 n 次恢复激活
- 每次恢复搜索最多跳过 MAX_SKIP（常数）
- 恢复的总开销： $O(n)$

合计： $O(n \cdot |G|) + O(n) = O(n \cdot |G|)$。

空间： 备忘录表最多存储 $|G| \cdot n$ 个条目。每个条目大小为 $O(1)$。总计： $O(n \cdot |G|)$。 □

5.9 约束满足方法论

鉴于在四条公理的背景下同时满足全部十二项约束十分困难，我们让两种领先的大语言模型分别采用对抗式的“深度思考”模式，尝试寻找一种满足所有要求的单一算法。

这些大语言模型受命证明该算法是否在所有重要指标上均为最优：完备性；正确性；紧凑性；优雅性；相对于输入长度的线性性（无论语法复杂度如何，也无论遇到多少错误）；以及相对于某种模型而言，解析器在各种情形下的行为是否直观、是否符合用户的合理预期。

此外，还对可能违反约束，或可能因相互作用而使约束彼此冲突的边界情况进行了穷举搜索。由此创建了一个大型单元测试套件，共包含 631 个单元测试，用于仔细而全面地检测进展和回归问题，并产生为算法迭代改进提供依据的新想法。

这次密集搜索最终为所有提出的约束收敛到一个单一解决方案，并且能够较为确定地证明：鉴于这些约束的严格性，PEG 错误恢复不存在从根本上比所发现方案更简单的解决方案。

这次密集的约束满足搜索最终产生了 squirrel 解析器的错误恢复算法；该算法确实满足完备性、正确性、紧凑性、性能和直观性方面的所有目标。

5.10 实现

错误恢复算法太长，无法纳入本文，但 Dart、TypeScript、Java 和 Python 版本的完整 squirrel 解析器实现可在以下网址获取：http://github.com/lukehutch/squirrelparser

5.10.1 测试套件。 该实现通过一套全面的测试进行了验证，共包含 631 项测试，涵盖：

左递归处理： 直接、间接和相互左递归；通过序列和选择项产生的隐式左递归；跨越多个间接层级的左递归；左递归展开期间的缓存失效；左递归与恢复之间的交互；Ref 对左递归子句的透明性。

错误恢复机制： 字符跳过和删除；在序列、选择项和重复边界处进行恢复；在嵌套子句中传播界限；错误定位和可见性约束；对部分匹配进行恢复；恢复与所有运算符类型之间的交互。

运算符正确性： 完整/不完整结果的序列组合；通过回溯选择首次匹配（有序选择）；带恢复的重复（一个或多个、零个或多个）；可选子句；包括空匹配、边界条件和嵌套结构在内的边缘情况。

性能与线性性：针对最多 100,000 个字符的输入验证线性时间复杂度 $O(n \cdot |G|)$；使用深层嵌套结构和复杂文法进行压力测试；测试性能边界情况，包括在非 Packrat 解析器中会导致指数级行为的病态回溯场景。

约束验证：测试明确验证所有设计约束，包括阶段隔离、单调消耗、完整性传播、左递归与错误恢复的分离、解析树覆盖以及可见性。

解析树覆盖不变量：20 项全面测试验证所有解析结果都完整覆盖从位置 0 到末尾的输入；完全失败时返回覆盖全部输入的 SyntaxError 节点，部分匹配则用尾随 SyntaxError 节点包装。

复杂交互：测试同时涉及多项功能的组合场景（例如，带错误恢复的左递归、带边界的嵌套重复、左递归上下文中的恢复）；Unicode 处理；实际解析模式。

所有测试均通过，证实该算法能够正确处理边界情况，同时保持线性性能并遵循覆盖不变量。

6 结论

我们介绍了 squirrel 解析器，这是一种 packrat 解析器，能够直接处理左递归和错误恢复，同时保持 $O(n \cdot |G|)$ 的时间和空间复杂度。这两项扩展均通过形式化数学推理，从第一性原理推导而来。

左递归。 该算法基于三个定理：不动点存在性、自底向上的必要性和长度单调递增。这些定理要求采用三种机制（通过按位置状态跟踪进行循环检测、结合种子增长进行迭代展开，以及带版本标签的备忘录），并且每种机制都通过反证法证明为必要。实现只需为每个备忘录条目设置一个布尔标志，并为每个输入位置设置一个版本计数器。与以往方法不同， [13, 14],该算法以统一方式处理所有类型的递归，无需进行文法预处理，并保持 packrat 的线性性。

错误恢复。我们证明，恰好需要两个阶段，且这已足够：发现阶段标记不完整解析；恢复阶段执行有界线性搜索。各阶段通过一个单比特（`isComplete`）进行通信，而完整结果与阶段无关，因此可以复用缓存。`isComplete` 标志在合取运算下构成布尔代数，稿件已提交至 ACM

实现代数运算的解析表达式文法运算符。通过反证法证明了每个组成部分都是必要的，从而确立了算法最小性。

理论贡献。 推导得出了四个深刻见解：(1) LR 展开以命令式形式实现 Y 组合子，通过克莱尼迭代计算最小不动点；(2) isComplete 标志构成组合布尔代数，使解析器正确性能够进行模块化推理；(3) 由于存在量词与全称量词分离，两阶段解析是最简方案；(4) 由于只有两个阶段，一个布尔阶段标志便已足够。

实证验证。 该实现通过了 500 多项全面测试，涵盖所有左递归类型、错误恢复场景、运算符正确性、性能边界情形、约束验证以及复杂功能交互。测试证实，对于长度达 100,000 个字符的输入，性能呈线性，并验证了对病态情形的正确处理。

更广泛的影响。 这项工作表明，左递归和错误恢复都不会从根本上增加语法分析的复杂度。通过以数学必要性为基础的精确算法设计，这两项挑战都能迎刃而解。这种方法具有普适性：从公理推导约束，构造最简解，并通过反证证明其必要性。由此得到的算法在理论上达到最优（每种机制都是必要的，复杂度与 packrat 下界相符），同时又足够简单，便于直接实现。

松鼠解析器在一个统一且连贯的框架中整合了左递归处理和错误恢复，为稳健的 PEG 解析提供了完整的解决方案，同时不牺牲 packrat 解析的优雅性或效率。

参考资料

[1] D 语言基金会。2021年。Pegged：面向 D 语言的解析表达式文法（PEG）库。https://github.com/dlang-community/pegged.[2] Mathias Doenitz。2022年。parboiled：Java 和 Scala 中优雅的语法分析。https://github.com/sirthias/parboiled。1.4.1 版。[3] Mathias Doenitz。2023年。parboiled2：基于宏的 Scala 解析表达式文法解析器生成器。https://github.com/sirthias/parboiled2.[4] Bryan Ford。2002年。Packrat 解析：简单、强大、惰性且线性时间。在第七届 *ACM SIGPLAN* 国际函数式编程会议论文集（*ICFP '02*）中。ACM，纽约，美国纽约州，36–47。doi:10.1145/581478.581483[5] Bryan Ford。2004年。解析表达式文法：一种基于识别的语法基础。在第 *31* 届 *ACM SIGPLAN-SIGACT* 编程语言原理研讨会论文集（*POPL '04*）中。ACM，纽约，美国纽约州，111–122。doi:10.1145/964001.964011[6] Edgar T. Irons。1961年。面向 ALGOL 60 的语法制导编译器。*ACM* 通讯 4，1（1961年1月），51–55。doi:10.1145/366062.366083[7] P. Lucas。1961年。公式翻译器的结构。《*ALGOL* 公报增刊》16（1961年9月），1–27。[8] Peter Norvig。1991年。自动记忆化技术及其在上下文无关解析中的应用。《计算语言学》17，1（1991年3月），91–98。[9] Terence Parr。2013年。*ANTLR 4* 权威指南 （第 2 版）。Pragmatic Bookshelf，达拉斯，美国得克萨斯州。[10] Terence Parr 和 Kathleen Fisher。2011年。LL(*)：ANTLR 解析器生成器的基础。在第 *32* 届 *ACM SIGPLAN* 编程语言设计与实现会议论文集（*PLDI '11*）中。ACM，纽约，美国纽约州，425–436。doi:10.1145/1993498.1993548[11] Terence Parr、Sam Harwell 和 Kathleen Fisher。2014年。自适应 LL(*) 解析：动态分析的力量。在*2014* 年 *ACM* 面向对象编程系统、语言与应用国际会议论文集（*OOPSLA '14*）中。ACM，纽约，美国纽约州，579–598。doi:10.1145/2660193.2660202[12] Roman R. Redziejowski。2009年。Mouse：从解析表达式到实用解析器。在*CS&P'2009* 研讨会论文集中，L. Czaja 和 M. Szczuka（编）。华沙大学，波兰华沙，514–525。[13] Laurence Tratt。2010年。直接左递归解析表达式文法。技术报告 EIS-10-01。密德萨斯大学工程与信息科学学院。[14] Alessandro Warth、James R. Douglass 和 Todd Millstein。2008年。Packrat 解析器可以支持左递归。在*2008* 年 *ACM SIGPLAN* 部分求值与基于语义的程序操纵研讨会论文集（*PEPM '08*）中。ACM，纽约，美国纽约州，103–110。doi:10.1145/ 1328408.1328424

```
类匹配(子句 子句,整数位置,整数长度, 匹配[]
                子句匹配);

常量匹配 不匹配 =新建匹配(空值          , -1, -1, []);

类解析器(子句 顶级规则) {
. 字符串 输入;
. MemoTable memoTable;
. 。整数[] 位置循环深度;
. .
. . 匹配 匹配(子句 规则,intpos){
. . var 备忘录条目 =memoTable.getOrCreateEntry(规则, pos);
. . 返回备忘录条目.匹配(this, 规则, pos);
. . }
. .
. . 匹配解析(String输入字符串) {
. . . 输入 = inputStr;
. . . 位置循环深度 =新建整数[输入.length + 1];
. . . memoTable =新建MemoTable();
. . . 返回匹配(topRule, 0);
. . }
. }

类MemoEntry{
. . 匹配 匹配;
. . 布尔值 inRecPath;
. . 布尔值 inLeftRecCycle;
. . int 循环深度;
. .
. . 匹配 match(解析器 parser, 子句 rule,intpos){
. . . 如果 (匹配 ==null ||循环深度 <
            parser.cycleDepthForPos[位置]) {
. . . .如果(在递归路径中) {
. . . . 如果(匹配 ==空){
. . . . . 在左递归循环中 =真;
. . . . . 匹配 =不匹配;
. . . . }
. . . .} 否则{
. . . . 在递归路径中 =true;
. . . . 当 (true){
. . . . .变量 newMatch =规则.match(解析器, pos);
. . . . .如果(匹配 !=null&&newMatch.len <=匹配.len)
. . . . . 跳出;
. . . . . 匹配 =newMatch;
. . . . .如果(在左递归循环中)
. . . . . 跳出循环;
. . . . . 循环深度 = ++(parser.cycleDepthForPos[位置]);
. . . . }
. . . . 在递归路径中 =假;
. . . }
. . . 循环深度 =parser.cycleDepthForPos[位置];
. . . }
. . 返回匹配;
. . }
}
```

```
59 抽象类 子句 { 60 . 抽象 匹配(解析器 解析器, 整数 位置); 61 } 62 63
类 F
{ 65 .
ij.匹
his,
75 类
置) {
于 (
析器
ma[
置, r
析器
] ==
97 .
{ 10
引用
106
```

图1。完整 squirrel 语法分析算法（不含错误恢复）的实现级伪代码（左），以及若干 PEG子句类型（右）。其他 PEG子句类型可以根据 PEG 运算符定义，按照所示模式轻松实现。

(a) 直接左递归

```
Grammar:
  A <- (A 'x') / 'x';

Parse tree for input: xxx
└─A : 0+3 : "xxx"
  └─A 'x' : 0+3 : "xxx"
    ├─A : 0+2 : "xx"
    │ └─A 'x' : 0+2 : "xx"
    │   ├─A : 0+1 : "x"
    │   │ └─'x' : 0+1 : "x"
    │   └─'x' : 1+1 : "x"
    └─'x' : 2+1 : "x"
```

(b) 间接左递归

```
Grammar:
  A <- B / 'x';
  B <- (A 'y') / (A 'x');

Parse tree for input: xxyx
└─A : 0+4 : "xxyx"
  └─B : 0+4 : "xxyx"
    └─A 'x' : 0+4 : "xxyx"
      ├─A : 0+3 : "xxy"
      │ └─B : 0+3 : "xxy"
      │   └─A 'y' : 0+3 : "xxy"
      │     ├─A : 0+2 : "xx"
      │     │ └─B : 0+2 : "xx"
      │     │   └─A 'x' : 0+2 : "xx"
      │     │     ├─A : 0+1 : "x"
      │     │     │ └─'x' : 0+1 : "x"
      │     │     └─'x' : 1+1 : "x"
      │     └─'y' : 2+1 : "y"
      └─'x' : 3+1 : "x"
```

(c) 输入依赖的左递归

```
Grammar:
  A <- B / 'z';
  B <- ('x' A) / (A 'y');

Parse tree for input: xxzyyy
└─A : 0+6 : "xxzyyy"
  └─B : 0+6 : "xxzyyy"
    └─'x' A : 0+6 : "xxzyyy"
      ├─'x' : 0+1 : "x"
      └─A : 1+5 : "xzyyy"
        └─B : 1+5 : "xzyyy"
          └─'x' A : 1+5 : "xzyyy"
            ├─'x' : 1+1 : "x"
            └─A : 2+4 : "zyyy"
              └─B : 2+4 : "zyyy"
                └─A 'y' : 2+4 : "zyyy"
                  ├─A : 2+3 : "zyy"
                  │ └─B : 2+3 : "zyy"
                  │   └─A 'y' : 2+3 : "zyy"
                  │     ├─A : 2+2 : "zy"
                  │     │ └─B : 2+2 : "zy"
                  │     │   └─A 'y' : 2+2 : "zy"
                  │     │     ├─A : 2+1 : "z"
                  │     │     │ └─'z' : 2+1 : "z"
                  │     │     └─'y' : 3+1 : "y"
                  │     └─'y' : 4+1 : "y"
                  └─'y' : 5+1 : "y"
```

(d) 输入依赖的左递归

```
Grammar:
  A <- 'x'? (A 'y' / A / 'y');

Parse tree for input: xxyyy
└─A : 0+5 : "xxyyy"
  ├─'x'? : 0+1 : "x"
  │ └─'x' : 0+1 : "x"
  └─(A 'y') / A / 'y' : 1+4 : "xyyy"
    └─A : 1+4 : "xyyy"
      ├─'x'? : 1+1 : "x"
      │ └─'x' : 1+1 : "x"
      └─(A 'y') / A / 'y' : 2+3 : "yyy"
        └─A : 2+3 : "yyy"
          ├─'x'? : 2+0 : ""
          └─(A 'y') / A / 'y' : 2+3 : "yyy"
            └─A 'y' : 2+3 : "yyy"
              ├─A : 2+2 : "yy"
              │ ├─'x'? : 2+0 : ""
              │ └─(A 'y') / A / 'y' : 2+2 : "yy"
              │   └─A 'y' : 2+2 : "yy"
              │     ├─A : 2+1 : "y"
              │     │ ├─'x'? : 2+0 : ""
              │     │ └─(A 'y') / A / 'y' : 2+1 : "y"
              │     │   └─'y' : 2+1 : "y"
              │     └─'y' : 3+1 : "y"
              └─'y' : 4+1 : "y"
```

(e) 交织左递归（3 个循环）

```
Grammar:
  S <- E;
  E <- F 'n' / 'n';
  F <- E '+' I* / G '-';
  G <- H 'm' / E;
  H <- G 'l';
  I <- '(' A+ ')';
  A <- 'a';

Parse tree for input: nlm-n+(aaa)n
└─E : 0+12 : "nlm-n+(aaa)n"
  └─F 'n' : 0+12 : "nlm-n+(aaa)n"
    ├─F : 0+11 : "nlm-n+(aaa)"
    │ └─E '+' I* : 0+11 : "nlm-n+(aaa)"
    │   ├─E : 0+5 : "nlm-n"
    │   │ └─F 'n' : 0+5 : "nlm-n"
    │   │   ├─F : 0+4 : "nlm-"
    │   │   │ └─G '-' : 0+4 : "nlm-"
    │   │   │   ├─G : 0+3 : "nlm"
    │   │   │   │ └─H 'm' : 0+3 : "nlm"
    │   │   │   │   ├─H : 0+2 : "nl"
    │   │   │   │   │ ├─G : 0+1 : "n"
    │   │   │   │   │ │ └─E : 0+1 : "n"
    │   │   │   │   │ │   └─'n' : 0+1 : "n"
    │   │   │   │   │ └─'l' : 1+1 : "l"
    │   │   │   │   └─'m' : 2+1 : "m"
    │   │   │   └─'-' : 3+1 : "-"
    │   │   └─'n' : 4+1 : "n"
    │   ├─'+' : 5+1 : "+"
    │   └─I* : 6+5 : "(aaa)"
    │     └─I : 6+5 : "(aaa)"
    │       ├─'(' : 6+1 : "("
    │       ├─A+ : 7+3 : "aaa"
    │       │ ├─A : 7+1 : "a"
    │       │ ├─A : 8+1 : "a"
    │       │ └─A : 9+1 : "a"
    │       └─')' : 10+1 : ")"
    └─'n' : 11+1 : "n"
```

(f) 交织左递归（2 个循环）

```
Grammar:
  M <- L;
  L <- P ".x" / 'x';
  P <- P "(n)" / L;

Parse tree for input: x.x(n)(n).x.x
└─L : 0+13 : "x.x(n)(n).x.x"
  └─P ".x" : 0+13 : "x.x(n)(n).x.x"
    ├─P : 0+11 : "x.x(n)(n).x"
    │ └─L : 0+11 : "x.x(n)(n).x"
    │   └─P ".x" : 0+11 : "x.x(n)(n).x"
    │     ├─P : 0+9 : "x.x(n)(n)"
    │     │ └─P "(n)" : 0+9 : "x.x(n)(n)"
    │     │   ├─P : 0+6 : "x.x(n)"
    │     │   │ └─P "(n)" : 0+6 : "x.x(n)"
    │     │   │   ├─P : 0+3 : "x.x"
    │     │   │   │ └─L : 0+3 : "x.x"
    │     │   │   │   └─P ".x" : 0+3 : "x.x"
    │     │   │   │     ├─P : 0+1 : "x"
    │     │   │   │     │ └─L : 0+1 : "x"
    │     │   │   │     │   └─'x' : 0+1 : "x"
    │     │   │   │     └─".x" : 1+2 : ".x"
    │     │   │   └─"(n)" : 3+3 : "(n)"
    │     │   └─"(n)" : 6+3 : "(n)"
    │     └─".x" : 9+2 : ".x"
    └─".x" : 11+2 : ".x"
```

(g) 显式左结合性

```
Grammar:
  E <- E '+' N / N;
  N <- [0-9]+;

Parse tree for input: 0+1+2+3
└─E : 0+7 : "0+1+2+3"
  └─E '+' N : 0+7 : "0+1+2+3"
    ├─E : 0+5 : "0+1+2"
    │ └─E '+' N : 0+5 : "0+1+2"
    │   ├─E : 0+3 : "0+1"
    │   │ └─E '+' N : 0+3 : "0+1"
    │   │   ├─E : 0+1 : "0"
    │   │   │ └─N : 0+1 : "0"
    │   │   │   └─[0-9] : 0+1 : "0"
    │   │   ├─'+' : 1+1 : "+"
    │   │   └─N : 2+1 : "1"
    │   │     └─[0-9] : 2+1 : "1"
    │   ├─'+' : 3+1 : "+"
    │   └─N : 4+1 : "2"
    │     └─[0-9] : 4+1 : "2"
    ├─'+' : 5+1 : "+"
    └─N : 6+1 : "3"
      └─[0-9] : 6+1 : "3"
```

(h) 显式右结合性

```
Grammar:
  E <- N '+' E / N;
  N <- [0-9]+;

Parse tree for input: 0+1+2+3
└─E : 0+7 : "0+1+2+3"
  └─N '+' E : 0+7 : "0+1+2+3"
    ├─N : 0+1 : "0"
    │ └─[0-9] : 0+1 : "0"
    ├─'+' : 1+1 : "+"
    └─E : 2+5 : "1+2+3"
      └─N '+' E : 2+5 : "1+2+3"
        ├─N : 2+1 : "1"
        │ └─[0-9] : 2+1 : "1"
        ├─'+' : 3+1 : "+"
        └─E : 4+3 : "2+3"
          └─N '+' E : 4+3 : "2+3"
            ├─N : 4+1 : "2"
            │ └─[0-9] : 4+1 : "2"
            ├─'+' : 5+1 : "+"
            └─E : 6+1 : "3"
              └─N : 6+1 : "3"
                └─[0-9] : 6+1 : "3"
```

(i) 有歧义的结合性

```
Grammar:
  E <- E '+' E / N;
  N <- [0-9]+;

Parse tree for input: 0+1+2+3
└─E : 0+7 : "0+1+2+3"
  └─E '+' E : 0+7 : "0+1+2+3"
    ├─E : 0+1 : "0"
    │ └─N : 0+1 : "0"
    │   └─[0-9] : 0+1 : "0"
    ├─'+' : 1+1 : "+"
    └─E : 2+5 : "1+2+3"
      └─E '+' E : 2+5 : "1+2+3"
        ├─E : 2+1 : "1"
        │ └─N : 2+1 : "1"
        │   └─[0-9] : 2+1 : "1"
        ├─'+' : 3+1 : "+"
        └─E : 4+3 : "2+3"
          └─E '+' E : 4+3 : "2+3"
            ├─E : 4+1 : "2"
            │ └─N : 4+1 : "2"
            │   └─[0-9] : 4+1 : "2"
            ├─'+' : 5+1 : "+"
            └─E : 6+1 : "3"
              └─N : 6+1 : "3"
                └─[0-9] : 6+1 : "3"
```

图 2. squirrel 解析器针对多种不同类型的左递归和结合律创建的解析树 .

```
类匹配 {
  子句 clause;
  整数 位置, len;
  列表<匹配> 子节点;
  布尔值 isComplete;              // 约束 6
  bool isFromLRContext;           // 约束 9
}
```

图 3. 匹配结果类型系统

```
类备忘录条目 {
  匹配结果;
  bool 在递归路径中;                 // LR 循环检测
  bool foundLeftRec;                 // 需要 LR 展开
  int memoVersion;                   // 按位置进行 LR 失效处理
  bool cachedInRecoveryPhase;        // 阶段隔离
}
```

图 4. 带阶段跟踪的记忆化条目

```
阶段匹配 =(cachedInRecoveryPhase  ==parser.inRecoveryPhase)  如果
(r
相
```

图 5. 带阶段跟踪的缓存验证

```
 1 匹配(clause, pos, {Clause bound}) { 2 // bound 在函数签名中可见 3 } 4 5 //
在
ch
地
10
```

图 6. 通过显式参数传播界限

```
匹配 parse(解析器 parser, 子句 topRule,字符串输入) {
  解析器.input = 输入;
  解析器.inRecoveryPhase = false;

  // 阶段 1：发现
  变量 结果 = 解析器.match(topRule, 0);
  如果 (结果 != 为空 && result.isComplete) {
    返回 result;      // 无恢复时成功
  }

  // 阶段 2：恢复
  解析器处于恢复阶段 = 为 true;
  返回 解析器匹配(topRule, 0);
}
```

图 7。两阶段错误恢复算法