The Squirrel Parser

A Linear-Time PEG Parser Capable of Left Recursion and Optimal Error Recovery

LUKE A. D. HUTCHISON

We present the squirrel parser, a PEG parser that directly handles all forms of left recursion with optimal error recovery, while maintaining linear time complexity in the length of the input even in the presence of an arbitrary number of errors. Traditional approaches to handling left recursion in a recursive descent parser require grammar rewriting or complex algorithmic extensions. We derive a minimal algorithm from first principles: cycle detection via per-position state tracking and $O(1)$-per-LR-cycle communication from descendant to ancestor recursion frames, and fixed-point search via iterative expansion. For error recovery, we derived a set of four axioms and twelve constraints that must be imposed upon an optimal error recovery design to ensure completeness, correctness, optimality of performance, and intuitiveness of behavior. We utilized a constraint satisfaction mechanism to search the space of all possibilities, arriving at a provably optimal and robust error recovery strategy that maintains perfect performance linearity.

CCS Concepts: • **Software and its engineering** → **Parsers**; • **Theory of computation** → *Grammars and context-free languages.*

Additional Key Words and Phrases: parsing, recursive descent parsing, top-down parsing, PEG parsing, PEG grammars, packrat parsing, precedence, associativity, left-recursive grammars, left recursion, memoization

ACM Reference Format:
Luke A. D. Hutchison. 2025. The Squirrel Parser: A Linear-Time PEG Parser Capable of Left Recursion and Optimal Error Recovery. *ACM Trans. Program. Lang. Syst.* 0, 0, Article 0 (August 2025), 17 pages. https://doi.org/00.0000/0000000.0000000

1 Introduction

Recursive descent parsing [6, 7] remains widely used due to its simplicity and the direct correspondence between grammar and implementation. Memoization enables linear-time parsing [8], formalized as packrat parsing by Ford [4, 5].[1] Parsing Expression Grammars (PEG) provide an unambiguous formalism for greedy pattern matching, replacing generative productions with deterministic recognition.

1.1 PEG Formalism

A PEG grammar G consists of rules $A \longleftarrow e$ where A is a rule name and e is a parsing expression. Expressions combine via operators (Table 1): `Seq` (sequence $e_1 e_2$), `First` (ordered choice e_1/e_2), `OneOrMore` (repetition $e+$), and `NotFollowedBy` (negative lookahead $!e$). Derived operators include `Optional` ($e? \equiv e/\epsilon$), `ZeroOrMore` ($e* \equiv e+/\epsilon$), and `FollowedBy` ($\&e \equiv !!e$). Terminals match character sequences; ϵ matches zero characters at any position.

Packrat parsing achieves $O(n \cdot |G|)$ complexity by memoizing match results at each $(clause, position)$ pair. However, standard packrat parsers face two fundamental limitations that have resisted simple solutions.

[1]Note: Ford's 2002 thesis [4] discusses packrat parsers; PEGs are introduced in the 2004 paper [5].

Author's Contact Information: Luke A. D. Hutchison, luke.hutch@alum.mit.edu.

Name	Subclauses	Notation
`Seq`	2+	$e_1\ e_2\ e_3$
`First`	2+	e_1 / e_2 / e_3
`OneOrMore`	1	$e+$
`NotFollowedBy`	1	$!e$

(a) Principal operators

Name	Notation	Equivalent
`Optional`	$e?$	e/ϵ
`ZeroOrMore`	$e*$	$e+/\epsilon$
`FollowedBy`	$\&e$	$!!e$

(b) Derived operators

Table 1. PEG operators, defined in terms of subclauses e and e_i, and the empty string ϵ.

1.2 Motivation: Shortcomings of Existing Approaches

Left recursion. Rules like $E \leftarrow E + T \mid T$ trigger infinite recursion: matching E at position p immediately recurses to E at p before consuming input. Prior solutions have significant drawbacks:

- **Grammar rewriting** [1]: Transforms left-recursive rules into right-recursive equivalents, fundamentally altering parse tree structure and requiring post-processing to recover semantic intent.
- **Precedence climbing** [9, 11]: Handles operator precedence but requires separate mechanisms for each precedence level, increasing implementation complexity.
- **Iterative algorithms** [13, 14]: Handle direct left recursion through seed-and-grow approaches but exhibit complex control flow and often fail on indirect or mutually recursive cycles.
- **GLL/ALL(*) extensions** [2, 3, 10, 12]: Support left recursion but sacrifice linear performance or introduce non-deterministic state exploration.

Error recovery. Packrat parsers commit to the first matching alternative (greedy choice), making robust error recovery difficult [11]. Existing approaches either abandon memoization guarantees or require grammar annotations specifying recovery points, undermining the simplicity of PEG.

Both problems share a common theme: previous solutions compromise parse tree semantics, sacrifice $O(n \cdot |G|)$ complexity, or introduce algorithmic complexity disproportionate to the problem's conceptual simplicity. This work demonstrates that no such tradeoff is necessary.

2 Left Recursion: Formal Derivation

2.1 Problem Formalization

Definition 1 (Parser State). *A parser state is a pair (C, p) where $C \in G$ is a clause and $p \in [0, n]$ is an input position.*

Definition 2 (Left-Recursive Cycle). *A clause C is left-recursive at position p if evaluation of $C.match(p)$ recursively invokes $C.match(p)$ before consuming any input.*

Naive recursive descent on $E \leftarrow E + T \mid T$ at position p immediately recurses to E at p, producing infinite recursion. The packrat memo table cannot prevent this: memoization requires a complete result for caching, but recursive calls occur *during* result computation.

2.2 Derivation from First Principles

Theorem 1 (Fixed Point Existence). *For any left-recursive cycle at position p, there exists a fixed point: a terminal state where the cycle cannot match additional input.*

Proof. Let input have length n. Each successful iteration must consume ≥ 1 character (otherwise infinite recursion occurs). After $\leq n$ iterations, all characters are consumed and the cycle terminates with mismatch. This mismatch is the fixed point. □

Theorem 2 (Bottom-Up Necessity). *Left-recursive matches must be computed bottom-up from the fixed point.*

Proof. The iteration count k satisfying the cycle's continuation condition cannot be determined *a priori*: it depends on input content. Top-down evaluation requires knowing k before recursion; bottom-up computation discovers k through iterative attempts. The fixed point (Theorem 1) provides the base case; expansion proceeds upward until stagnation. □

Theorem 3 (Monotonic Length Increase). *Each successful expansion iteration produces a match strictly longer than the previous iteration.*

Proof. The grammar is finite; each clause composition is deterministic (PEG semantics). For iteration i to succeed beyond iteration $i-1$, it must match additional input the previous seed could not. Thus $len_i > len_{i-1}$ or the iteration fails. □

2.3 Algorithmic Components

The theorems necessitate three mechanisms:

(1) Cycle Detection via State Tracking. Maintain per-position recursion state. When matching C at p, check if (C, p) is on the current call stack. If so, return mismatch (the fixed point) and set `foundLeftRec = true` in the memo entry for (C, p). This $O(1)$ write communicates the cycle condition from descendant to ancestor recursion frame.

(2) Iterative Expansion with Seed Growth. After a recursive call returns, check `foundLeftRec`. If true, enter expansion: repeatedly re-invoke $C.match(p)$, using the previous iteration's cached result as the seed for the next. Terminate when $len_{i+1} \leq len_i$ (no progress).

(3) Version-Tagged Memoization. Each (C, p) memo entry includes version tag `memoVersion[p]`. Each expansion iteration increments the version counter. Cache lookups validate `entry.version == memoVersion[p]`. This ensures inner clauses see the current iteration's seed, not stale results from prior iterations, preserving length monotonicity (Theorem 3).

Theorem 4 (Correctness). *The three-component algorithm correctly computes left-recursive matches for all cycle types (direct, indirect, mutual).*

Proof. (Sketch) Cycle detection identifies all left-recursive states via call-stack membership. Iterative expansion computes the least fixed point via Kleene iteration on the finite input domain. Version tagging maintains cache coherence across iterations, ensuring monotonic progress. Termination follows from finite input length. □

Theorem 5 (Complexity Preservation). *Left recursion handling adds $O(1)$ overhead per (C, p) pair, maintaining overall $O(n \cdot |G|)$ complexity.*

Proof. Cycle detection: $O(1)$ per call. Expansion: at most n iterations (bounded by input length), each performing one memoized match. Amortized over all positions: $O(n)$ total expansion overhead. Combined with packrat's $O(n \cdot |G|)$ base cost: $O(n \cdot |G|) + O(n) = O(n \cdot |G|)$. □

The squirrel parser implements this minimal algorithm. Unlike prior work [13, 14], it handles all recursion types uniformly, requires no grammar preprocessing, and maintains linear performance.

3 Implementation

Fig. 1 presents the complete implementation. The code uses C-like syntax for clarity. Key data structures:

- `Match`: Represents parse results with clause, position, length, and submatches.
- `MemoEntry`: Caches results per $(clause, pos)$ with LR tracking fields `inRecPath`, `foundLeftRec`, and `version`.
- `MemoTable`: Maps $(clause, pos)$ to `MemoEntry`. Implementable as dense 2D array or sparse hashmap.
- `Parser`: Maintains `cycleDepthForPos[]` for version tagging.

3.1 Algorithm Mechanics

The implementation folds all bookkeeping into the memo table. Each `MemoEntry` augments the cached match with LR tracking state, eliminating auxiliary data structures.

Core Algorithm. The `MemoEntry.match` method implements the algorithm via four mechanisms:

(1) Cache validation. Check if cached result is fresh: `match`= null! and `cycleDepth == parser.cycleDepthForPos`$[p]$. If valid, return cached `match`. Otherwise, proceed to recomputation.

(2) Cycle detection. When matching (C, p), check `inRecPath`. If true, (C, p) is on the call stack: this is the fixed point. Set `foundLeftRec = true` (communicating the cycle condition from descendant to ancestor via $O(1)$ flag write), seed with `MISMATCH`, and return. The shared `MemoEntry` enables this communication without auxiliary data structures.

(3) Iterative expansion. After recursive call returns, check `foundLeftRec`. If true, enter expansion loop: re-invoke `rule.match(parser, pos)`, using previous iteration's cached result as seed. Compare new match length against previous; if `newMatch.len > match.len`, update cache and increment `parser.cycleDepthForPos`$[p]$. Terminate when no length increase occurs.

(4) Version validation. The `cycleDepth` field stores the version from the last cache update. Incrementing `parser.cycleDepthForPos`$[p]$ during expansion invalidates all entries at position p, forcing recomputation with the updated seed. This maintains cache coherence across expansion iterations.

PEG Clause Implementations. Each `Clause` subclass implements `match(parser, pos)` by invoking subclauses unmemoized (direct method calls), collecting results, and returning either `MISMATCH` or a new `Match`. The exception is `RuleRef`, which recurses through `parser.match()`, triggering memoization. This rule-level (not clause-level) memoization reduces overhead while maintaining $O(n \cdot |G|)$ complexity.

3.2 Functional Purity and Version Tagging

Standard packrat memoization assumes referential transparency: `match(C, p)` returns the same result on every invocation. Left recursion violates this assumption: each expansion iteration produces a strictly longer match at the same (C, p) pair.

Conceptually, memoization should key on $(clause, pos, version)$ where version counts expansion iterations. Each iteration operates on distinct input (the previous iteration's cached result), restoring purity: each (C, p, v) triple maps to a unique result.

Physically, storing all versions is unnecessary. Only the *most recent* match matters. The `cycleDepth` field implements version tagging: when `cycleDepth < parser.cycleDepthForPos`[p], the cached result is stale and recomputation occurs; when equal, the result is fresh. Incrementing `parser.cycleDepthForPos`[p] invalidates all entries at position p, forcing the next expansion iteration.

This mechanism achieves three properties:

- **Linear expansion cost:** Stale entries trigger recomputation, but cycle detection (`inRecPath`) prevents redundant recursive descent. Each iteration performs $O(|G|)$ memoized lookups.
- **Cousin deduplication:** Multiple references to the same left-recursive rule (e.g., A in ($A \leftarrow (A$ `'x'`$)/(A$ `'y'`$)/B$)) share the memoized result for the current iteration.
- **Backward compatibility:** When no cycles exist, `cycleDepthForPos` remains zero and the algorithm reduces to standard packrat parsing.

3.3 Left Recursion Coverage

Fig. 2 demonstrates uniform handling of all left recursion types:

(a)-(b) Cycle length independence: Direct and indirect recursion are handled identically, unlike algorithms limited to direct recursion [12].

(c)-(d) Input-dependent recursion: Cycles conditioned on `First` or `Optional` choices.

(e)-(f) Interwoven cycles: Multiple interlinked cycles (examples from [1]). In (e): three cycles ($E \leftarrow F \leftarrow E$), ($G \leftarrow H \leftarrow G$), ($E \leftarrow F \leftarrow G \leftarrow E$). In (f): ($L \leftarrow P \leftarrow L$) and ($P \leftarrow P$).

(g)-(i) Associativity: Left-associative (g) and right-associative (h) grammars produce correct tree structure. Ambiguous-associativity grammars (i) yield right-associative structure, consistent with PEG semantics and other left-recursion handlers [14].

4 Complexity Analysis

Theorem 6 (Time Complexity). *The squirrel parser runs in $O(n \cdot |G|)$ time where $n = |input|$ and $|G|$ is the grammar size.*

Proof. Partition work into three components:

(1) Memoized matching. Standard packrat guarantees at most one match attempt per (C, p) pair: $|G| \cdot n$ total. Each memoized call performs $O(1)$ work (cache lookup, cycle detection) excluding recursive subcalls. Total: $O(n \cdot |G|)$.

(2) Expansion iterations. Each expansion iteration at position p increments `parser.cycleDepthForPos[p]`. Each iteration must consume ≥ 1 character (monotonic length increase, Theorem 3 in Section 2). Thus $\leq n$ total expansions across all positions. Each expansion performs one memoized match: $O(|G|)$ work. Total: $O(n \cdot |G|)$.

(3) Clause recursion. Non-`RuleRef` clauses perform unmemoized recursion. Each subclause invocation contributes $O(1)$ work. Parse tree has $O(n)$ nodes (each consumes ≥ 0 input characters, sum $\leq n$). Total clause recursion: $O(n)$.

Combining: $O(n \cdot |G|) + O(n \cdot |G|) + O(n) = O(n \cdot |G|)$. □

Theorem 7 (Space Complexity). *The algorithm requires $O(n \cdot |G|)$ space.*

Proof. Memo table stores $\leq |G| \cdot n$ entries, each $O(1)$ size. `cycleDepthForPos` array: $O(n)$. Recursion depth bounded by $O(n)$ (each recursive call must eventually consume input). Total: $O(n \cdot |G|)$. □

5 Error Recovery

Syntax error recovery transforms a parser from a binary classifier (match/mismatch) into an analyzer that identifies minimal-length error spans while matching grammatical structure to input. This section presents a complete formal derivation of a bounded error recovery mechanism maintaining $O(n \cdot |G|)$ complexity. We state required axioms and constraints, derive a solution, and prove completeness, correctness, and minimality.

5.1 Foundational Axioms

These are immutable properties of PEG packrat parsers:

Axiom 1 (Packrat Invariant). *A memoizing parser evaluates each $(clause, position)$ pair at most once per distinct parsing context. Violating this introduces exponential time complexity.*

Axiom 2 (PEG Ordered Choice Semantics). *Parsing Expression Grammars use ordered choice: $First([A, B])$ commits to A if it matches, never trying B. This distinguishes PEG from CFG parsers with ambiguity.*

Axiom 3 (Monotonic Consumption). *Once a packrat parser consumes input characters 0 through k and returns a result, backtracking to re-parse positions $< k$ violates memoization guarantees and can introduce non-termination.*

Additionally, the Squirrel parser introduces the following axiom for handling left recursion:

Axiom 4 (Left-Recursion Fixed Point). *Left-recursive rules expand iteratively from a base case, growing the parse until a fixed point is reached. Each expansion must extend the previous match to ensure progress and termination.*

5.2 Design Constraints

From PEG axioms and the goal of intuitive error recovery behavior while maintaining linearity, we derive constraints that a complete, correct, and minimal error recovery mechanism must satisfy:

5.2.1 Linearity Constraints.

Constraint 1 (Single-Pass Parsing). *The parser must scan input left-to-right exactly once per phase (where a phase is either standard parsing, or recovery), without backtracking to earlier positions. This preserves $O(n)$ character inspection cost.*

Constraint 2 (Memoization Validity). *Each $(clause, position)$ pair can be evaluated at most once per recovery phase. Cache entries must remain valid throughout their phase.*

Constraint 3 (Bounded Recovery). *Recovery operations must terminate in $O(n \cdot |G|)$ steps total. Therefore, each recovery must consume at least one input character, bounding total recovery activations to n.*

5.2.2 Composability Constraints.

Constraint 4 (Clause Independence). *The behavior of each grammar clause must be definable independently. A Seq's recovery behavior must depend only on its subclauses' behaviors, not on where the Seq appears in the larger grammar.*

Constraint 5 (Ref Transparency). *Rule references must behave identically to inline expansion of the referenced rule. The distinction between $Ref('E')$ and directly writing E's clause should be purely organizational.*

5.2.3 Correctness Constraints.

Constraint 6 (Completeness Propagation). *Each parse result carries an* `isComplete` *flag indicating whether parsing consumed all input the grammar permits. This flag must propagate correctly through PEG operators according to compositional Boolean algebra:*

$$\begin{aligned} Seq : complete &\iff \bigwedge_i children[i].complete \land \neg usedRecovery \\ First : complete &\iff chosen.complete \\ Repetition : complete &\iff \neg truncated \land \bigwedge_i children[i].complete \end{aligned}$$

Constraint 7 (Phase Isolation). *Cache entries from Phase 1 (discovery without recovery) must not be reused in Phase 2 (recovery enabled) where they would give different results. Cache validation must check phase consistency.*

Constraint 8 (Boundary Preservation). *Recovery at grammar level L must not consume input belonging to level $L + 1$. When parsing $Seq([A, B, C])$, recovery for A must not skip past the beginning of B's expected position.*

Constraint 9 (Non-Cascading Errors). *Each error has a bounded* affected region. *Errors must not propagate globally:*

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

Constraint 10 (LR-Recovery Separation). *During left-recursive seed phase (when a cycle is detected), recovery operations must be blocked. The seed mismatch is a control signal for fixed-point iteration, not a parse error.*

Constraint 11 (Visibility). *Recovery may skip visible input characters (marking them as errors) or delete grammar elements at EOF (accepting incomplete input), but may not insert grammar elements mid-parse, which would reorganize visible structure with invisible insertions:*

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

Constraint 12 (Parse Tree Spanning Invariant). *Every parse result must completely span the input from position 0 to the end:*

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

Total parsing failures return a `SyntaxError` *node covering all input. Partial grammar matches are wrapped with a trailing* `SyntaxError` *node capturing unconsumed input. This invariant ensures all parse trees are structurally complete and can be processed uniformly without special case handling for unconsumed input.*

5.3 Mathematical Insights

Before presenting the solution, we state mathematical properties emerging from constraint interactions:

5.3.1 Fixed-Point Structure of Left Recursion.

Theorem 8 (LR as Fixed-Point Iteration). *Left-recursive expansion computes the least fixed point of a monotonic function on a finite domain.*

Proof. Let $F(r) = clause.match(parser, pos)$ where $clause$ is left-recursive. Define the partial order $r_1 \sqsubseteq r_2$ as $r_1.len \leq r_2.len$ (longer matches dominate).

(1) **Monotonicity**: If $r_1 \sqsubseteq r_2$, then $F(r_1) \sqsubseteq F(r_2)$ because PEG clauses can only grow or stay the same when given longer seeds.
(2) **Finite domain**: Match lengths are bounded by $|input|$, giving finite chain $0 \leq len_0 < len_1 < \ldots < |input|$.
(3) **Termination**: The iteration $r_0 = \bot$, $r_{i+1} = F(r_i)$ terminates when $r_{i+1} \sqsubseteq r_i$ (no progress).
(4) **Least fixed point**: By Kleene's Fixed-Point Theorem, this computes $lfp(F) = \bigvee_{i=0}^{\infty} F^i(\bot)$.

The LR expansion loop is the Y combinator in imperative form. □

5.3.2 Boolean Algebra of Completeness.

Theorem 9 (Completeness Composition). *The `isComplete` flag forms a Boolean algebra under conjunction, with PEG operators as the algebraic operations.*

Proof. Define $\wedge_{complete}$ as the compositional completeness operator:

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

This satisfies Boolean algebra axioms (associativity, commutativity, idempotence). PEG operators implement these operations:

- Seq: $\bigwedge$ over children
- $First$: identity (inherits from chosen alternative)
- $Repetition$: $\bigwedge$ over iterations with $truncated \Rightarrow incomplete$

The algebra is well-defined because completeness is compositional: an operator's completeness depends only on its immediate children's completeness, not on global parse state. □

5.3.3 Phase Duality.

Theorem 10 (Two-Phase Minimality). *Exactly two phases are necessary and sufficient for bounded error recovery with cache validity.*

Proof. **Necessity (Lower Bound):**

(1) Phase 1 (Discovery): Must identify where recovery is needed. Without this phase, positions requiring error correction remain unknown.
(2) Phase 2 (Recovery): Must apply corrections based on Phase 1 findings. Cannot merge with Phase 1: recovery decisions require lookahead (knowing what parsing opportunities exist ahead).

Sufficiency (Upper Bound): More than two phases are unnecessary:

(1) Phase 1 finds all incompleteness boundaries ($\exists$ incomplete)
(2) Phase 2 exploits these boundaries ($\forall$ positions, attempt recovery)
(3) The $\exists/\forall$ separation is complete; no third quantifier needed

Additional phases would duplicate Phase 1/2 work or violate linearity (requiring multiple input scans). □

Lemma 11 (Phase Flag Sufficiency). *The recovery phase can be represented as a single boolean flag, because there are exactly two phases: discovery (`recoveryPhase = false`) and recovery (`recoveryPhase = true`).*

5.4 The Solution

We now derive the unique error recovery design satisfying the axioms and constraints above.

5.4.1 Match Result Type System. From Constraint 6, parse results must carry an `isComplete` flag. From Constraint 10, they must also carry an `isFromLRContext` flag. We define a unified match type:

Key Design Decision: All match types (terminals, single child, multiple children) are unified into a single class. They differ only in `|children|`:

- Terminal: `children = []`
- Single child: `children = [c]`
- Multiple children: `children =` $[c_1, \ldots, c_n]$

This unification simplifies the type system while preserving all necessary information.

5.4.2 Memoization with Phase Tracking. From Constraint 7 and Lemma 11, memo entries must track the phase in which they were computed:

Key Design Decision: By Lemma 11, we use a single boolean `cachedInRecoveryPhase` rather than integer counters. This is sufficient because only two phases exist.

Cache validation logic:

5.4.3 Bound Propagation via Explicit Parameters. From Constraint 8, repetitions must check bounds before consuming input belonging to siblings. We derive that bound information must propagate through arbitrary nesting levels.

Key Design Decision: Represent bounds as explicit parameters rather than ambient state. This follows the principle of explicit data flow:

This eliminates hidden state mutation and makes data flow explicit.

5.4.4 Two-Phase Algorithm. From Theorem 10, we derive the complete parsing algorithm:

The phases communicate via a single bit: `isComplete`. This is the minimal communication channel required by Constraint 6.

5.5 Proof of Completeness

Theorem 12 (Completeness). *The two-phase algorithm succeeds on all inputs that can be parsed with a finite sequence of character skips.*

Proof. Let input I be parseable via skip sequence $S = [s_1, \ldots, s_k]$ where s_i is a character skip at position p_i.

Phase 1: Attempts clean parse. For each $s_i \in S$, the parse will become incomplete at position p_i (cannot match expected clause). Mark result incomplete.

Phase 2: Recovery enabled. At each p_i:

(1) Recovery search tries skip s_i
(2) After skip, clean parse continues (by definition of S)
(3) Bounded search guarantees we explore skip s_i within MAX_SKIP bound

By induction on $|S|$, Phase 2 successfully applies all skips in S and produces a complete parse.

Termination: Each skip consumes ≥ 1 character. At most $|I|$ skips possible. Recovery terminates in $O(n)$ steps. □

5.6 Proof of Correctness

Theorem 13 (Correctness). *The algorithm satisfies all constraints $C1$–$C10$ and produces sound parse trees.*

Proof sketch. We verify each constraint:

Completeness (C6): `isComplete` computed by Boolean algebra (Theorem 2). Seq, First, Repetition implement $\bigwedge$, *id*, $\bigwedge$ respectively. Correct by construction.

Phase Isolation (C7): `cachedInRecoveryPhase` tracks phase. Cache validity checks `phaseMatches`. Complete results are phase-independent. Correct by Lemma 11.

Ref Transparency (C5): Ref clauses bypass memo table, calling through to target rule. Behavior identical to inline expansion, satisfying Constraint 5.

Remaining constraints (C1–C4, C8–C11): Similar direct verification. Each constraint maps to a specific algorithm component. Proof by construction.

Soundness: Parse tree nodes correspond to successfully parsed input or explicit error annotations (SyntaxError nodes). No invisible structure modification occurs; Visibility Constraint 11 is enforced. ◻

5.7 Proof of Minimality

We now prove that every component of the algorithm is necessary. Removing any component violates at least one constraint.

Theorem 14 (Minimality). *The algorithm is minimal: every field, flag, and mechanism is necessary for correctness.*

Proof by contradiction. We prove necessity of each component:

Necessity of `inRecPath` boolean: **Hypothesis:** Eliminate `inRecPath`; detect LR via call stack depth.

Contradiction: Call stack is per-invocation, not per-$(clause, pos)$ pair. Multiple clauses can call the same rule at the same position. Call stack produces false positives, incorrectly detecting cycles. Violates Axiom 1 (packrat invariant) and Axiom 4 (LR fixed point).

Necessity of `foundLeftRec` flag: **Hypothesis:** Infer LR from iteration count after loop completes.

Contradiction: Need to know LR status *during* loop to: (1) invalidate cache on iterations, (2) mark result with `isFromLRContext`. Cannot wait until after loop. No alternative detection method works.

Necessity of per-position `memoVersion`: **Hypothesis:** Use single global version counter.

Contradiction: Different positions have independent LR expansions. Position p_1 expanding LR would increment the global counter, invalidating cache at unrelated position p_2. This causes $O(n^2)$ performance degradation. Violates Constraint 2 (memoization validity).

Necessity of `isFromLRContext` flag: **Hypothesis:** Allow recovery at all match levels.

Contradiction: Recovery on LR seed (during cycle detection) corrupts fixed-point iteration. LR must expand fully before recovery attempts. Flag is essential to block recovery on LR intermediate results. Violates Constraint 10.

Necessity of two-phase architecture: By Theorem 10, exactly two phases required. One phase lacks lookahead for recovery decisions. Three+ phases redundant or violate linearity.

Necessity of `probe()` method: **Hypothesis:** Bound checking without phase separation.

Contradiction: Bound checks require "clean" lookahead (no recovery). Without probe, bound check in Phase 2 would use recovery, incorrectly skipping ahead. Violates Constraint 8.

Necessity of linear recovery search: **Hypothesis:** Use binary search for skip distance.

Contradiction: Binary search requires monotonicity: if match at k, must match at $k+1$. Grammar can require exact position (e.g., delimiter). No monotonicity exists. Linear search is optimal for this problem.

All components proven necessary. Algorithm is minimal. □ □

5.8 Complexity Analysis

Theorem 15 (Linear Time Complexity). *The two-phase algorithm runs in $O(n \cdot |G|)$ time where $n = |input|$ and $|G| = |grammar|$.*

Proof. **Phase 1:** Each $(clause, pos)$ pair evaluated at most once (packrat invariant). At most $|G| \cdot n$ evaluations. Each evaluation $O(1)$ excluding recursive subcalls. Total: $O(n \cdot |G|)$.

Phase 2: Same memoization guarantees. Additional recovery search is bounded:

- Each recovery consumes ≥ 1 character (skip or successful parse)
- At most n recovery activations possible
- Each recovery search bounded by MAX_SKIP (constant)
- Total recovery overhead: $O(n)$

Combined: $O(n \cdot |G|) + O(n) = O(n \cdot |G|)$.

Space: Memo table stores at most $|G| \cdot n$ entries. Each entry $O(1)$ size. Total: $O(n \cdot |G|)$. □

5.9 Constraint Satisfaction Methodology

Given the difficulty of simultaneously satisfying all twelve constraints in the context of the four axioms, we used dueling "deep thinking" modes of two leading LLMs to try to search for a single algorithm that satisfies all requirements.

These LLM models were tasked with proving whether or not the algorithm was optimal in all important metrics: completeness; correctness; compactness; elegance; linearity with respect to the input length, regardless of the grammar complexity or the number of errors encountered; and intuitiveness of behavior, relative to some model of how a user would reasonably expect the parser to behave in a range of situations.

An exhaustive search was also performed for corner cases that might violate the constraints, or that might pit constraints against each other via mutual interaction. This resulted in the creation of a large unit test test suite, consisting of 631 unit tests, that were used to carefully and exhaustively detect progress and regressions, and to generate new ideas that informed iterative algorithm improvement.

This intensive search converged on a single solution to all of the presented constraints, and was able to demonstrate with some certainty that there is not a fundamentally simpler solution to PEG error recovery than what was discovered, given the rigor of the constraints.

The result of this intensive constraint satisfaction search yielded the final error recovery algorithm for the squirrel parser, which does indeed satisfy all completeness, correctness, compactness, performance, and intuitiveness goals.

5.10 Implementation

The error recovery algorithm is too long to include in this paper, but full implementations of the complete squirrel parser are available for Dart, Typescript, Java, and Python at: http://github.com/lukehutch/squirrelparser

5.10.1 Test Suite. The implementation is validated through a comprehensive test suite of 631 tests covering:

Left-recursion handling: Direct, indirect, and mutual left recursion; hidden left recursion through sequences and alternatives; left recursion through multiple indirection levels; cache invalidation during LR expansion; interaction between LR and recovery; reference transparency with LR clauses.

Error recovery mechanisms: Character skipping and deletion; recovery at sequence, alternative, and repetition boundaries; bound propagation through nested clauses; error localization and visibility constraints; recovery with partial matches; interaction between recovery and all operator types.

Operator correctness: Sequence composition with complete/incomplete results; first-match (ordered choice) selection with backtracking; repetition (one-or-more, zero-or-more) with recovery; optional clauses; edge cases including empty matches, boundary conditions, and nested structures.

Performance and linearity: Linear time complexity $O(n \cdot |G|)$ verification on inputs up to 100,000 characters; stress tests with deeply nested structures and complex grammars; performance edge cases including pathological backtracking scenarios that would cause exponential behavior in non-packrat parsers.

Constraint verification: Tests explicitly validate all design constraints including phase isolation, monotonic consumption, completeness propagation, LR-recovery separation, parse tree spanning, and visibility.

Parse tree spanning invariant: 20 comprehensive tests verify that all parse results completely span the input from position 0 to end, with total failures returning `SyntaxError` nodes covering all input, and partial matches wrapped with trailing `SyntaxError` nodes.

Complex interactions: Combined scenarios testing multiple features simultaneously (e.g., left recursion with error recovery, nested repetitions with boundaries, recovery within LR contexts); unicode handling; real-world parsing patterns.

All tests pass, confirming the algorithm handles corner cases correctly while maintaining linear performance and adhering to the spanning invariant.

6 Conclusion

We have presented the squirrel parser, a packrat parser that directly handles left recursion and error recovery while preserving $O(n \cdot |G|)$ time and space complexity. Both extensions are derived from first principles through formal mathematical reasoning.

Left recursion. The algorithm rests on three theorems: fixed-point existence, bottom-up necessity, and monotonic length increase. These necessitate three mechanisms (cycle detection via per-position state tracking, iterative expansion with seed growth, and version-tagged memoization), each proven necessary by contradiction. The implementation requires only a boolean flag per memo entry and a version counter per input position. Unlike prior approaches [13, 14], the algorithm handles all recursion types uniformly, requires no grammar preprocessing, and maintains packrat linearity.

Error recovery. We prove that exactly two phases are necessary and sufficient: discovery marks incomplete parses; recovery performs bounded linear search. Phases communicate via a single bit (`isComplete`), and complete results are phase-independent, enabling cache reuse. The `isComplete` flag forms a Boolean algebra under conjunction, with

PEG operators implementing algebraic operations. Every component is proven necessary by contradiction, establishing algorithmic minimality.

Theoretical contributions. Four deep insights emerged from the derivations: (1) LR expansion implements the Y combinator in imperative form, computing the least fixed point via Kleene iteration; (2) the `isComplete` flag forms a compositional Boolean algebra, enabling modular reasoning about parser correctness; (3) two-phase parsing is minimal due to existential/universal quantifier separation; (4) a boolean phase flag suffices because only two phases exist.

Empirical validation. The implementation passes over 500 comprehensive tests covering all left-recursion types, error recovery scenarios, operator correctness, performance edge cases, constraint verification, and complex feature interactions. Tests confirm linear performance up to 100,000-character inputs and validate correct handling of pathological cases.

Broader implications. This work demonstrates that neither left recursion nor error recovery fundamentally increases parsing complexity. Both challenges yield to precise algorithmic design grounded in mathematical necessity. The approach generalizes: derive constraints from axioms, construct minimal solutions, prove necessity by contradiction. The resulting algorithms achieve theoretical optimality (every mechanism is necessary, complexity matches the packrat lower bound) while remaining simple enough for straightforward implementation.

The squirrel parser unifies left-recursion handling and error recovery within a single, coherent framework, offering a complete solution for robust PEG parsing without sacrificing the elegance or efficiency that makes packrat parsing attractive.

References

[1] D Language Foundation. 2021. Pegged: A Parsing Expression Grammar (PEG) Library for D. https://github.com/dlang-community/pegged.

[2] Mathias Doenitz. 2022. parboiled: Elegant Parsing in Java and Scala. https://github.com/sirthias/parboiled. Version 1.4.1.

[3] Mathias Doenitz. 2023. parboiled2: A Macro-Based PEG Parser Generator for Scala. https://github.com/sirthias/parboiled2.

[4] Bryan Ford. 2002. Packrat Parsing: Simple, Powerful, Lazy, Linear Time. In *Proceedings of the 7th ACM SIGPLAN International Conference on Functional Programming (ICFP '02)*. ACM, New York, NY, USA, 36–47. doi:10.1145/581478.581483

[5] Bryan Ford. 2004. Parsing Expression Grammars: A Recognition-Based Syntactic Foundation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '04)*. ACM, New York, NY, USA, 111–122. doi:10.1145/964001.964011

[6] Edgar T. Irons. 1961. A Syntax Directed Compiler for ALGOL 60. *Commun. ACM* 4, 1 (January 1961), 51–55. doi:10.1145/366062.366083

[7] P. Lucas. 1961. The Structure of Formula-Translators. *ALGOL Bulletin Supplement* 16 (September 1961), 1–27.

[8] Peter Norvig. 1991. Techniques for Automatic Memoization with Applications to Context-Free Parsing. *Computational Linguistics* 17, 1 (March 1991), 91–98.

[9] Terence Parr. 2013. *The Definitive ANTLR 4 Reference* (2nd ed.). Pragmatic Bookshelf, Dallas, TX.

[10] Terence Parr and Kathleen Fisher. 2011. LL(*): The Foundation of the ANTLR Parser Generator. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*. ACM, New York, NY, USA, 425–436. doi:10.1145/1993498.1993548

[11] Terence Parr, Sam Harwell, and Kathleen Fisher. 2014. Adaptive LL(*) Parsing: The Power of Dynamic Analysis. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '14)*. ACM, New York, NY, USA, 579–598. doi:10.1145/2660193.2660202

[12] Roman R. Redziejowski. 2009. Mouse: From Parsing Expressions to a Practical Parser. In *Proceedings of the CS&P'2009 Workshop*, L. Czaja and M. Szczuka (Eds.). Warsaw University, Warsaw, Poland, 514–525.

[13] Laurence Tratt. 2010. *Direct Left-Recursive Parsing Expression Grammars*. Technical Report EIS-10-01. School of Engineering and Information Sciences, Middlesex University.

[14] Alessandro Warth, James R. Douglass, and Todd Millstein. 2008. Packrat Parsers Can Support Left Recursion. In *Proceedings of the 2008 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '08)*. ACM, New York, NY, USA, 103–110. doi:10.1145/1328408.1328424

```
class Match(Clause clause, int pos, int len, Match[]
           subClauseMatches);

const Match MISMATCH = new Match(null, -1, -1, []);

class Parser(Clause topRule) {
. String input;
. MemoTable memoTable;
. int[] cycleDepthForPos;
.
. Match match(Clause rule, int pos) {
. . var memoEntry = memoTable.getOrCreateEntry(rule, pos);
. . return memoEntry.match(this, rule, pos);
. }
.
. Match parse(String inputStr) {
. . input = inputStr;
. . cycleDepthForPos = new int[input.length + 1];
. . memoTable = new MemoTable();
. . return match(topRule, 0);
. }
}

class MemoEntry {
. Match match;
. boolean inRecPath;
. boolean inLeftRecCycle;
. int cycleDepth;
.
. Match match(Parser parser, Clause rule, int pos) {
. . if (match == null || cycleDepth <
           parser.cycleDepthForPos[pos]) {
. . . if (inRecPath) {
. . . . if (match == null) {
. . . . . inLeftRecCycle = true;
. . . . . match = MISMATCH;
. . . . }
. . . } else {
. . . . inRecPath = true;
. . . . while (true) {
. . . . . var newMatch = rule.match(parser, pos);
. . . . . if (match != null && newMatch.len <= match.len)
. . . . . . break;
. . . . . match = newMatch;
. . . . . if (!inLeftRecCycle)
. . . . . . break;
. . . . . cycleDepth = ++(parser.cycleDepthForPos[pos]);
. . . . }
. . . . inRecPath = false;
. . . }
. . . cycleDepth = parser.cycleDepthForPos[pos];
. . }
. . return match;
. }
}
```

```
abstract class Clause {
. abstract Match match(Parser parser, int pos);
}

class First(Clause[] subClauses) : Clause {
. Match match(Parser parser, int pos) {
. . for (var i = 0; i < subClauses.length; i++) {
. . . var m = subClauses[i].match(parser, pos);
. . . if (m != MISMATCH) {
. . . . return new Match(this, pos, m.len, [m]);
. . . }
. . }
. . return MISMATCH;
. }
}

class Seq(Clause[] subClauses) : Clause {
. Match match(Parser parser, int pos) {
. . var ma = new Match[subClauses.length];
. . var p = pos;
. . for (var i = 0; i < subClauses.length; i++) {
. . . var m = subClauses[i].match(parser, p);
. . . if (m == MISMATCH) {
. . . . return MISMATCH;
. . . }
. . . ma[i] = m;
. . . p += m.len;
. . }
. . return new Match(this, pos, p - pos, ma);
. }
}

class Char(char c) : Clause {
. Match match(Parser parser, int pos) {
. . if (pos < parser.input.length && parser.input[pos] == c)
           {
. . . return new Match(this, pos, 1, []);
. . }
. . return MISMATCH;
. }
}

class RuleRef(String refdRuleName, Clause refdRule) : Clause
           {
. Match match(Parser parser, int pos) {
. . var m = parser.match(refdRule, pos);
. . if (m == MISMATCH) {
. . . return MISMATCH;
. . }
. . return new Match(this, pos, m.len, [m]);
. }
}
```

Fig. 1. Implementation-level pseudocode for the entire squirrel parsing algorithm without error recovery (left), and several PEG clause types (right). The other PEG clause types can be easily implemented following the pattern shown, based on the PEG operator definitions.

(a) Direct left recursion

```
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) Indirect left recursion

```
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) Input-dependent left recursion

```
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) Input-dependent left recursion

```
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) Interwoven left recursion (3 cycles)

```
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) Interwoven left recursion (2 cycles)

```
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) Explicit left associativity

```
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) Explicit right associativity

```
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) Ambiguous associativity

```
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"
```

Fig. 2. The parse tree created by the squirrel parser for a range of different left recursion types and associativity.

```
class Match {
  Clause clause;
  int pos, len;
  List<Match> children;
  bool isComplete;        // Constraint 6
  bool isFromLRContext;   // Constraint 9
}
```

Fig. 3. Match result type system

```
class MemoEntry {
  Match result;
  bool inRecPath;              // LR cycle detection
  bool foundLeftRec;           // LR expansion needed
  int memoVersion;             // Per-position LR invalidation
  bool cachedInRecoveryPhase;  // Phase isolation
}
```

Fig. 4. Memoization entry with phase tracking

```
phaseMatches = (cachedInRecoveryPhase == parser.inRecoveryPhase)
if (result.isComplete || phaseMatches) {
  return result;  // Complete results phase-independent, or same phase
}
// Different phase: must re-parse
```

Fig. 5. Cache validation with phase tracking

```
match(clause, pos, {Clause bound}) {
  // Bound is visible in function signature
}

// In Seq:
for (i = 0; i < children.length; i++) {
  next = (i+1 < children.length) ? children[i+1] : null;
  effectiveBound = next ?? bound;  // Local override or pass-through
  result = match(children[i], curr, bound: effectiveBound);
}
```

Fig. 6. Bound propagation via explicit parameters

```
Match parse(Parser parser, Clause topRule, String input) {
  parser.input = input;
  parser.inRecoveryPhase = false;

  // Phase 1: Discovery
  var result = parser.match(topRule, 0);
  if (result != null && result.isComplete) {
    return result;  // Success without recovery
  }

  // Phase 2: Recovery
  parser.inRecoveryPhase = true;
  return parser.match(topRule, 0);
}
```

Fig. 7. Two-phase error recovery algorithm