An Efficient Parser for Bounded-Order Product-Free Lambek Categorial Grammar via Term Graph

Jinman Zhao and Gerald Penn
Dept. of Computer Science
University of Toronto
CANADA
{jzhao,gpenn}@cs.toronto.edu

Abstract

Lambek Categorial Grammar (LCG) parsing has been proved to be an NP-complete problem. However, in the bounded-order case, the complexity can be reduced to polynomial time. Fowler (2007) first introduced the *term graph*, a simple graphical representation for LCG parsing, but his algorithm for using it remained largely inscrutable. Pentus (2010) later proposed a polynomial algorithm for bounded-order LCG parsing based on cyclic linear logic, yet both approaches remain largely theoretical, with no open-source implementations available. In this work, we combine the term-graph representation with insights from cyclic linear logic to develop a novel parsing algorithm for bounded-order LCG. Furthermore, we release our parser as an open-source tool.

1 Introduction

Many studies have shown that transformer-based models such as large language models (LLMs) effectively capture certain aspects of syntactic structure (Niu et al., 2022; Strobl et al., 2024; Ramesh et al., 2024; Cagnetta and Wyart, 2024). Coming to terms with better representations of syntax could play a significant role in future LLM research, contributing to advancements in areas such as mitigating hallucinations (Wu and Liu, 2025) and reasoning (Barke et al., 2024).

While most current research on syntax in NLP primarily focuses on context-free grammars (CFGs), categorial grammar (CG) deserves greater attention due to its unique advantages. Unlike CFGs, which rely on a predefined set of production rules, CG is inherently lexicalized, meaning that all grammatical variations are captured within the lexicon itself. This allows syntactic processing to be driven directly by the lexical categories present in a sentence, rather than by a global rule set. Additionally, CG strongly adheres to the principle of compositionality, as seen in Montague grammar, ensuring that syntactic and semantic derivations align closely. This property makes semantic interpretation more transparent and directly extractable from syntax, and could be particularly beneficial for improving the still fraught understanding of the interplay between structure and meaning by the neural language modellling community.

Downstream tasks that leverage CG's syntactic representations to interpret sentence structure generally involve two stages: 1) supertagging (Bhargava and Penn, 2020; Tian et al., 2020; Kogkalidis and Moortgat, 2023), in which each word is assigned a syntactic category, and 2) sequent derivation (Yamaki et al., 2023; Clark, 2015; Fowler, 2007), which organizes these categories into a coherent graphical structure that captures the sentence's grammatical composition.

Like other CG formalisms, Lambek Categorial Grammar (LCG) parsing is amenable to this two-step process. A useful supertagger (Zhao and Penn, 2024) for LCG has already been proposed, allowing us to focus on the second sequent derivation step. But LCG sequent derivation has been proved to be NP-complete (Pentus, 2006). Fortunately, it becomes polynomially solvable under a bounded-order assumption (Fowler, 2007; Pentus, 2010). This assumption is not only theoretically appealing but also empirically justified: in practical scenarios, the syntactic category order tends to remain low. For instance, in both the CCGbank (Hockenmaier and Steedman, 2007) and LCGbank corpus (Bhargava et al., 2024), the maximum order is only 5 (Fowler, 2008), suggesting that bounded-order parsing is sufficient for most real-world applications. Fowler (2007) introduced the *term graph*, a simple graphical representation for LCG parsing. While it also proposed a polynomial-time algorithm with complexity $O(n^3)$ for bounded-order parsing, that approach was never properly explicated and its proof of correctness is overly complex. Pentus (2010) developed an alternative $O(n^4)$ algorithm based on cyclic linear logic. In this work, we prove that the insights from cyclic linear logic also work for term graphs and use these to propose an efficient yet simple algorithm for bounded-order LCG parsing using term graphs that remains $O(n^3)$. We release our parser and demonstrate our parser on LCGbank.[1] For expository purposes, only the recognition (yes/no) version of the algorithm is presented in the text.

2 Related Work

LCG was first introduced by Lambek (1958), and since then, numerous variants have been developed, including ones that are product-free, using only $/$ and $\backslash$ as connectives, unidirectional, with only one of $/$ or $\backslash$, and lexicalized, that prohibit the derivation of the empty sequent, among others. The parsing complexity of LCG has been an ongoing topic of research, leading to the introduction of various frameworks aimed at addressing parsing challenges, such as proof nets (Roorda, 1991), LC-Graphs (Penn, 2004), term graphs (Fowler, 2007), and cyclic linear logic (Girard, 1989; Yetter, 1990).

A key milestone in this line of work was the proof proposed by Pentus (2006) that derivability in the original LCG is NP-complete. Subsequent studies further demonstrated that derivability in the product-free (Savateev, 2012) and semidirectional (Dörre, 1996) LCGs is also NP-complete, while unidirectional (Savateev, 2009) derivability has been shown to be solvable in polynomial time. Despite this theoretical complexity, in practical settings, both the original LCG (Pentus, 2010) and its product-free (Fowler, 2007) variant have been proved to be polynomial-time solvable under reasonable constraints, making them more feasible for real-world applications.

3 Preliminary

3.1 Lambek Categorial Grammar

A *Lambek Categorial Grammar* (LCG) is a formal system used to model natural language syntax through category-based inference. The set of categories C is built from a set of *atomic categories* (e.g., $\{S, NP, N, PP\}$) along with three *binary connectives*: the *forward slash* ($/$), the *backward slash* ($\backslash$) and the *product* ($\cdot$), which encode directional function application. In this work, we focus on the **product-free** (resulting in only two connectives) LCG since the product connective has limited contribution to linguistic.

A *Lambek grammar* G is defined as a four-tuple:

$$G = \langle \Sigma, A, R, S \rangle$$

where Σ is a finite alphabet of symbols (lexical items). A is a set of atomic categories from which complex categories are constructed. R is a relation that maps symbols in Σ to categories in C. S is the set of sentence categories, determining well-formed sentence structures.

Lambek calculus L has the following rules of inference:

$$\frac{\Gamma X \to Y}{\Gamma \to Y/X}\ (/R) \quad \Gamma \text{ is not empty}$$

$$\frac{X\Gamma \to Y}{\Gamma \to X\backslash Y}\ (\backslash R) \quad \Gamma \text{ is not empty}$$

$$\frac{\Gamma \to X \quad \Delta Y \Theta \to Z}{\Delta Y/X \Gamma \Theta \to Z}\ (/L)$$

$$\frac{\Gamma \to X \quad \Delta Y \Theta \to Z}{\Delta \Gamma X\backslash Y \Theta \to Z}\ (\backslash L)$$

$$\frac{\Gamma \to X \quad \Delta X \Theta \to Y}{\Delta \Gamma \Theta \to Y}\ (\text{CUT})$$

Lambek calculus allowing empty premises, denoted as L^*, is a special case where Γ can be empty.

Sequent derivability problem is to determine whether a sequent $\Gamma \vdash s, s \in S$ is derivable under L (or L^*).

3.2 Bounded-Order

We define the *order* of a category, denoted $o(\alpha)$, as a measure of the depth of argument implication nesting. The definition proceeds recursively as follows:

- $o(\alpha) = 0$, if α is a basic (atomic) category;
- $o(\alpha/\beta) = o(\beta\backslash\alpha) = \max(o(\alpha), o(\beta) + 1)$, for complex categories.

We have $o(\text{NP}) = 0$, $o((\text{NP}\backslash\text{S})/\text{NP}) = 1$, and $o((\text{S}/\text{NP})\backslash(\text{S}/\text{NP})) = 2$ as examples.

The maximum order of a category can also be interpreted as the depth of the corresponding term frame structure

3.3 Term Graph

Proof nets (Roorda, 1991; Buch, 2009) are a widely used graphical framework for representing derivations. One key advantage is their ability to merge ambiguous derivations, effectively capturing multiple syntactic structures that share the same semantic interpretation. For instance, as shown in

[1]https://github.com/zhaojinm/LCG_parser.git

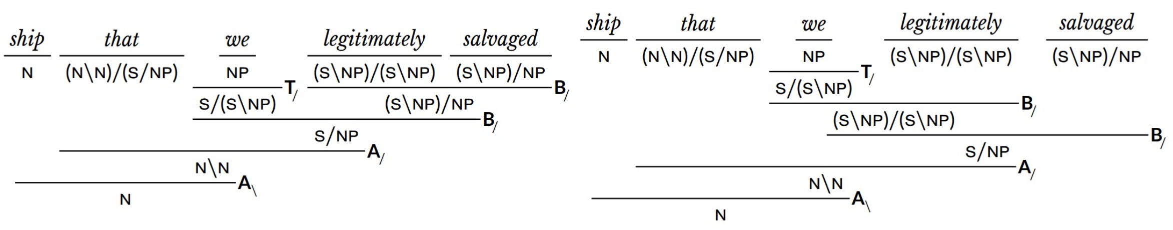

Figure 1: An example of LCG sequent derivation with distinct derivation but same semantics (Bhargava et al., 2024).

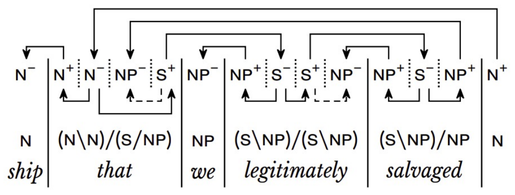

Figure 2: An example of proofnet (Bhargava et al., 2024).

Figure 1, although the two derivations appear structurally different, they convey the same meaning as Figure 2 demonstrated.

There are in fact two algorithmic formalizations of proof nets subsequent to Roorda (1991), however. The more conservative one (Penn, 2004) in terms of natural deduction is what led to term graphs (Fowler, 2007), a simplification that requires less structure to be explicitly maintained. Pentus (2010), which comes from the other formalization, never really embraced all of its advantages.

Constructing a term graph for a given sequent follows a two-step process. Let us use the following sequent as an example:

$$S/(S\backslash NP) \quad (S\backslash NP)/NP \quad NP \quad \vdash S$$

Step 1: Graph Frame Construction

The first step is deterministic and begins by assigning a polarity to each category in the sequent:

- **Negative** polarity is assigned to antecedents(left-hand category).
- **Positive** polarity is assigned to the succedent(right-hand category).

After polarity assignment, the above sequent become:

$$S/(S\backslash NP)^{-} \quad (S\backslash NP)/NP^{-} \quad NP^{-} \quad \vdash S^{+}$$

Each polarized category is treated as a node, and categories containing slashes are decomposed according to the following rewriting rules recursively until no rule can be applied:

$$(\alpha/\beta)^{-} \Rightarrow \alpha^{-} \rightarrow \beta^{+}$$
$$(\beta\backslash\alpha)^{-} \Rightarrow \beta^{+} \leftarrow \alpha^{-}$$
$$(\alpha/\beta)^{+} \Rightarrow \beta^{-} \dashleftarrow \alpha^{+}$$
$$(\beta\setminus\alpha)^{+} \Rightarrow \alpha^{+} \dashrightarrow \beta^{-}$$

In these transformations, the left-hand side of each rule determines the neighborhood of α. The *dashed* edges introduced in this step are referred to as *Lambek edges*, while other connections are called *regular edges*. This process effectively translates syntactic categories into tree structures.

Next, rooted Lambek edges are introduced, connecting the root of the succedent tree to the roots of the antecedent trees. This makes the whole sequent from a forest to a bigger tree. See Figure 3 as a frame for the sequent.

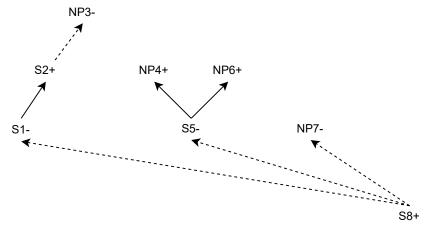

Figure 3: An example of term graph frame.

Step 2: Atom Matching

The second step is *non-deterministic* and involves computing a complete matching of the polarized atoms. The matching must satisfy two constraints:

1. **Planarity**: The edges connecting atoms must not cross when visualized.

2. **Opposite Polarity Pairing**: Every atomic category instance must be paired with exactly one instance of the same category but with *opposite polarity*.

These pairings, called *matches* or *links*, are represented by *regular* edges, are directed from positive atoms to negative atoms.

An example of a *term graph* for a sequent derivation is shown in Figure 4.

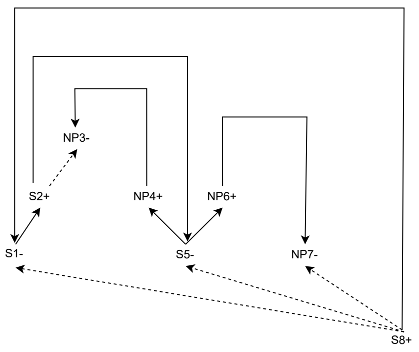

Figure 4: An example of term graph.

Correctness criteria

A term graph G is considered L^**-integral* if it satisfies the following conditions:

1. T(0): It is regular acyclic, for all vertices, there is no regular path to itself.
2. T(1): For every Lambek edge $\langle s, t\rangle$ in G, there exists a regular path from s to t.

A term graph is called *integral* if it is L^**-integral* and additionally satisfies:

3. T(CT): For every Lambek edge $\langle s, t\rangle$ in G, there exists a regular path from s to some vertex x in G. If x has a non-rooted Lambek in-edge $\langle s', x\rangle$, then there must not be a regular path from s to s'.

Theorem 3.1. *A sequent is derivable in L if and only if it has an integral term graph. A sequent is derivable in* L^* *if and only if it has an* L^**-integral term graph.*

Proof. Fowler (2007) □

If we use the naive chart-based parser, the complexity would be NP-complete. Fowler (2007) proposed a method that claims $O(n^3)$ complexity for LCG bounded order sequent derivability, however, this algorithm is very complex and difficult to understand.

3.4 Cyclic linear logic

Cyclic linear logic framework, which, while easy to understand, involves numerous steps. Due to space constraints, we refer readers to Pentus (2010) for a more detailed explanation. Here, we provide only a brief overview of the key idea: they transform sequent derivability in Lambek calculus into sequent derivability in cyclic linear logic. Through a series of transformations, they further convert sequents into a tree-like structure (Figure 5 as an example), allowing for axiom matching.

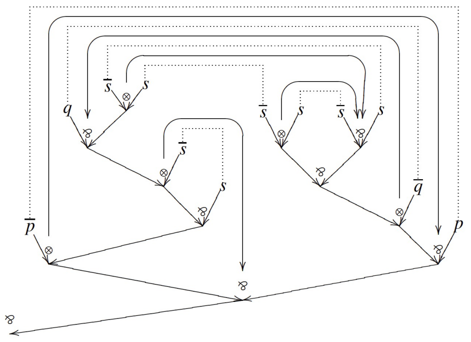

Figure 5: An example of CMLL-based framework (Pentus, 2010). Note that unlike term graph, *order* in CMLL does not equal to the depth of the framework, CMLL's depth is unbounded.

The core idea of their algorithm is that for a span (i, j) in the subtree, it is unnecessary to store a subgraph containing all vertices from i to j. Instead, only the information from two paths in the tree is relevant: one from the root to $axiom_i$ and another from the root to $axiom_j$. Since the depth remains constant under the bounded order condition, the chart-based parser achieves cubic complexity $O(n^3)$ for L^*. However, for L, additional information is required, increasing the complexity to $O(n^4)$.

4 Bounded Order Parser

We combine the strengths of two existing frameworks and propose a simple and easily understandable algorithm based on term graphs. Our approach retains the cubic-time complexity for both $L*$ and L, making it both efficient and practical.

In this section, we first introduce the naive chart parser and our proposed algorithm, followed by a

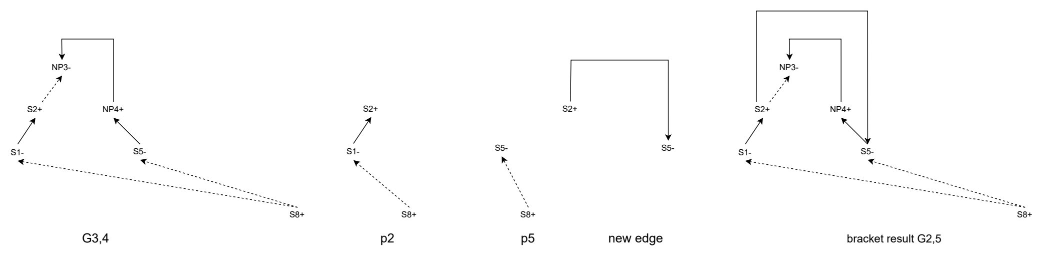

Figure 6: An example of naive bracket operation.

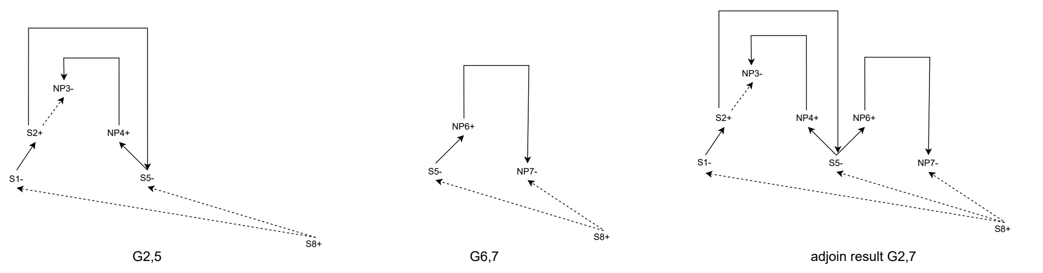

Figure 7: An example of naive adjoin operation.

proof of its correctness.

4.1 Naive Chart Parser for $L*$

A natural approach is to use dynamic programming, following the standard chart parsing paradigm. For each span (i, j), we consider two possible operations: `bracket` and `adjoin`. The `bracket` operation introduces a new link between positions i and j, on top of the subgraph constructed over the inner span $(i + 1, j - 1)$. Figure 6 illustrates an example of the `bracket` operation. Mathematically, the `bracket` operation can be viewed as the union of graphs:

$$G_{i,j} = G_{i+1,j-1} \cup p_i \cup p_j \cup new\_edge$$

The other operation is `adjoin`, which merges two adjacent subgraphs $G_{i,k}$ and $G_{k+1,j}$ into a larger graph $G_{i,j}$. Figure 7 illustrates an example of the `adjoin` operation. From a mathematical perspective, this corresponds to the composition or union of the two subgraphs:

$$G_{i,j} = G_{i,k} \cup G_{k+1,j}$$

We can apply certain early stopping heuristics during parsing. For example, if a candidate graph $G_{i,j}$ contains a cycle, we can immediately discard it from the chart. However, despite such pruning strategies, the number of possible graphs that may be stored in each chart entry $F_{i,j}$ can still be exponential in the worst case. This is because the number of nodes in $G_{i,j}$ is unbounded, and thus the number of possible subgraph configurations grows exponentially. As a result, the overall complexity of this naive chart parser remains exponential.

4.2 Efficient Parser for $L*$

The key bottleneck of the naive chart parser described above lies in its failure to prune intermediate nodes: each chart entry may store an exponential number of subgraph variants due to the unbounded number of nodes.

We begin by introducing the core insight behind our parser design. Our approach incrementally merges pairs of subgraphs. However, unlike standard methods that may retain the full internal structure of each subgraph, we observe that it is sufficient to preserve only the node information along the two outermost boundary paths. Nodes in the interior of the merged span will no longer be accessed by any subsequent operations from outside the span and thus can be safely ignored. This simplification significantly reduces the complexity.

Frame construction Our method's first step aligns with Section 3.3, and we start from Figure 3.

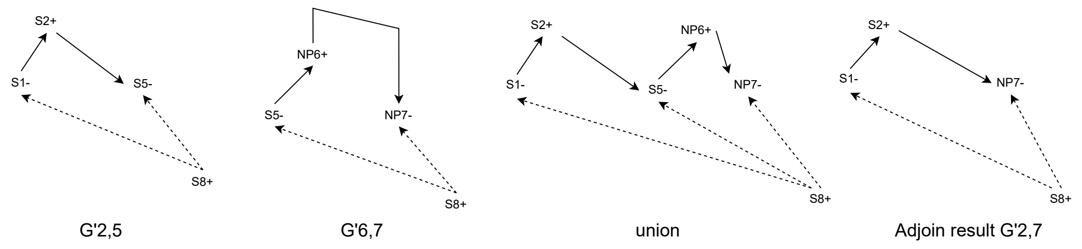

Figure 8: An example of updated adjoin operation. We only illustrate the new `adjoin` operation. However, the graph G'_{25} shown here can be seen as an updated version of the bracket-derived G_{25} from Figure 6. Additionally, the edge $\mathrm{S}_2 \to \mathrm{NP}_7$ in G'_{27} exists because there is a regular path $\mathrm{S}_2 \rightsquigarrow \mathrm{NP}_7$ in the union graph. Similarly, the dash edge $\mathrm{S}_8 \dashrightarrow \mathrm{S}_5$ in the union graph corresponds to $\mathrm{S}_8 \dashrightarrow \mathrm{S}_1$ in G'_{27}, since S_1 (i.e., w in line 9 of Algorithm 1) satisfies $\mathrm{S}_1 \rightsquigarrow \mathrm{S}_5$.

4.2.1 Chart Parser

We continue to define our parser in terms of two operations: `adjoin` and `bracket`, each corresponding to the union of subgraphs. However, since each chart entry $F_{i,j}$ now stores a *simplified graph*, we denote it by $G'_{i,j}$. For the both operation, the update rule remains structurally the same with extra `Simplify` function, for `adjoin`:

$$G'_{i,j} = G'_{i,k} \cup G'_{k+1,j}$$

$$G'_{i,j} = \texttt{Check\_and\_Simplify}(G'_{i,j})$$

and for `bracket`:

$$G'_{i,j} = G'_{i+1,j-1} \cup p_i \cup p_j \cup new\_edge$$

$$G'_{i,j} = \texttt{Check\_and\_Simplify}(G'_{i,j})$$

Here, `Check_and_Simplify` removes redundant internal nodes and preserves only the necessary boundary information. The function `Check_and_Simplify` (Algorithm 1) performs two key check steps and one simplify step sequentially:

1. For all node pairs (u, v) in $G'_{i,j}$, if both $u \rightsquigarrow v$ and $v \rightsquigarrow u$ hold, then the cyclic constraint T(0) is violated. In this case, the function returns `"CYCLIC"` and terminates further computation for this span. (line 1)

2. For every dash edge $u \dashrightarrow v$, if $u \rightsquigarrow v$, then all incoming dash edges to v (i.e., $w \dashrightarrow v$) are deleted. This ensures that the T(1) condition is satisfied for node v and no need to keep this in the future. (line 2)

3. Simplify graph(after line 3). Line 3-4 initialize a new graph where V are nodes on two boundary paths and E is empty. For all (u, v) where $u \rightsquigarrow v$ we can add edge $u \to v$ in the result graph (line 5-7). Line 8-14 deal with the dashed edge, from Lemma 4.7, if $u \dashrightarrow v$, then the dashed path $u \rightsquigarrow v$ is unique and it must pass through w since $w \rightsquigarrow v$. Note that S contains all the nodes the u access to. So the dashed constraint becomes a bunch of constraints; either node in S access to w will make T(1) satisfy for v. Lines 8–14 and line 2 should be considered as a unified process: both are designed to enforce the T(1) constraint.

Figure 8 is the update result for Figure 6 and 7.

Algorithm 1: `Check_and_Simplify`($G'_{i,j}$)

```
   Input: $G'_{i,j}$
   Output: A simplified graph
1  T0 check;
2  T1 check;
3  $V = node(pi \cup pj)$
4  $new\_G = \{V, E = \emptyset\}$
5  foreach $u, v \in V$ do
6      if $u \rightsquigarrow v \in G'_{i,j}$ then
7          $E.append(u \to v)$;
8  foreach $u \dashrightarrow v$ in $G'_{i,j}$ do
9      Follow $v$ backwards along regular edges
         to the furthest $w \in V$ such that $w \rightsquigarrow v$.
10     $S = \{s | s \in V, u \rightsquigarrow s\}$
11     if $S = \emptyset$ or $w$ is NULL then
12         return 'NO REGULAR ACCESS'
13     foreach $s \in S$ do
14         $E.append(s \dashrightarrow w)$;
15 return "VALID", $new\_G$
```

4.2.2 Correctness

Let G be the original term graph, $G_{i,j}$ be the partial term graph with span (i, j). And let $G'_{i,j}$ be the simplified graph that only contains nodes in two paths p_i and p_j where p_i is the path in Figure 3 from root to i. We want to prove that $G'_{i,j}$ stores enough information for LCG parsing as $G_{i,j}$.

Lemma 4.1. *If the term graph G is acyclic, the method will not return 'CYCLIC'.*

Proof. By induction, for each edge (u, v) in $G'_{i,j}$, there must exist a path that $u \rightsquigarrow v$ in $G_{i,j}$. Therefore, no cycle in G indicates no cycle can be detected by G'. □

Lemma 4.2. *If $G_{i,j}$ is the adjoin of $G_{i,k}$ and $G_{k+1,j}$, then there is no cross match (a, b) such that $i \leq a \leq k$ and $k + 1 \leq b \leq j$.*

Proof. This can be easily proved by induction where the base case is $i = j - 1$. □

Lemma 4.3. *For each $G_{i,j}$, for all $u, v \in p_i \cup p_j$, if $u \rightsquigarrow v$ in $G_{i,j}$, then $u \rightsquigarrow v$ in $G'_{i,j}$.*

Proof. Prove by Induction.
Base: $G_{i,i}$ = $G'_{i,i}$, trivial case.
Induction Step:
Assume $G'_{i+1,j-1}$, $G'_{i+1,k}$ and $G'_{k,j}$ are both satisfiable.
Case 1: `bracketing` where:

$$G_{i,j} = G_{i+1,j-1} \cup p_i \cup p_j \cup new\_edge$$

Assume there is an edge $(u, v) \in p_i \cup p_j$ such that $u \rightsquigarrow v$ in $G_{i,j}$, and $u \not\rightsquigarrow v$ in $G'_{i,j}$. By IS, $u, v \notin (p_i \cap p_{i+1}) \cup (p_{j-1} \cap p_j)$. Thus, there must be a node $x \in (p_i \cap p_{i+1}) \cup (p_{j-1} \cap p_j)$ and $w \in G_{i,j} \setminus (p_i \cup p_{i+1} \cup p_{j-1} \cup p_j)$ such that $x \rightarrow w$ or $w \rightarrow x$, and such edge does not exists based on our graph construction rule. Contradiction.
Case 2: `adjoin`, where:

$$G_{i,j} = G_{i,k} \cup G_{k+1,j}$$

Since both subgraphs are satisfied by IS and there is no cross-match by Lemma 4.2, the result also holds for $G_{i,j}$. □

Lemma 4.4. *If the term graph G is cyclic, the method will return 'CYCLIC'.*

Proof. If there is a cycle in G, let i be the left most and j be the right most, then by Lemma 4.3, $G_{i,j}$ must contains two edges of (i, j) and (j, i). Then, the cycle check will return 'CYCLIC'. □

Theorem 4.5. *(T(0)) Term graph G is regular acyclic if and only if the method returns 'ACYCLIC'.*

Proof. By Lemma 4.1 and Lemma 4.4. □

Lemma 4.6. *If a dashed edge $u \dashrightarrow v$ in G has no regular access, then the method will return 'NO REGULAR ACCESS'.*

Proof. By induction, for each edge (u, v) in G', there must exist a path that $u \rightsquigarrow v$ in G. Therefore, no regular path in G indicates no regular path can be detected by G'. □

Lemma 4.7. *Regular in degree is 1 for all nodes.*

Proof. Induction on the term graph construction. □

Lemma 4.8. *If the method returns 'NO REGULAR ACCESS' for a dashed edge $u \dashrightarrow v$, then this edge has no regular access in G.*

Proof. By Lemma 4.7, such a regular path $u \rightsquigarrow v$ is unique. According to the construction procedure of G', the constraint $u \dashrightarrow v$ is initialized as the pair $(\{u\}, v)$, and is iteratively updated through `adjoin` or `bracket` operations. Suppose an intermediate step yields a span $[i, j]$; the result is then represented as a pair (S, w), where S is a set of nodes and w is a single node, both of which lie on the path $p_i \cup p_j$. This representation implies that at least one node in S must maintain *regular access* to w.

If, during an `adjoin` or `bracket` operation, the method determines that regular access does not hold, then one of the following conditions must be true:

1. No node on $p_i \cup p_j$ has access to w; or
2. No node in S has access to any node on $p_i \cup p_j$.

In either case, by Lemma 4.2, it follows that there exists no path of regular access from S to w. □

Theorem 4.9. *(T(1)) For each dashed edge $u \dashrightarrow v$ in term graph G, $u \rightsquigarrow v$ in G if and only if the method does not return 'NO REGULAR ACCESS'.*

Proof. By Lemma 4.6 and Lemma 4.8. □

4.2.3 Complexity of L*

All operations strictly follow the procedure of a chart parser. Therefore, the overall complexity is

$$O(n^3) \cdot |F_{ij}| \cdot \max(O(\texttt{adjoin}), O(\texttt{bracket}))$$

where $|F_{ij}|$ denotes the number of possible graph configurations for the span $[i, j]$ stored in the chart entry F_{ij}. Since the category order is bounded(i.e., constant), the number of nodes within each G'_{ij} is constant, and the number of possible configurations is also bounded. Moreover, the cost of each graph operation (such as `adjoin` and `bracket`) deals with the constant number of nodes, so both $O(\texttt{adjoin})$ and $O(\texttt{bracket})$ are bounded.

Therefore, the overall complexity is $O(n^3)$.

4.3 Parser for L

While the constraint T(CT) may initially appear to be a condition on Lambek edges, a closer examination reveals that it is in fact a constraint on each positive node. Specifically, we can restate T(CT) as:

- T(CT): for every positive node s^+, there must exist a negative node x^- such that either $root \dashrightarrow x$, or $s \not\rightsquigarrow \texttt{dash\_parent}(x)$.

Moreover, for any positive node s^+, the search for a negative node x^- satisfying the T(CT) condition can terminate as soon as one such x^- is found. This is because T(CT) can no longer be violated once the condition holds for any such x^-. Specifically, by T(1), we have $\texttt{dash\_parent}(x) \rightsquigarrow x$. Suppose at some point we observe that $s^+ \rightsquigarrow x$ but $s^+ \not\rightsquigarrow \texttt{dash\_parent}(x)$. The only way this could occur is if $\texttt{dash\_parent}(x) \rightsquigarrow s^+ \rightsquigarrow x$, which would imply a cycle. However, by T(0), cycles are disallowed, so it must be the case that $s^+ \not\rightsquigarrow \texttt{dash\_parent}(x)$.

This formulation of T(CT) is naturally compatible with our parsing algorithm. For each positive node x_i, we maintain a set $\texttt{CT}_{x_i} = \{x_i\}$, initialized with the node itself. During parsing, whenever an element $v \in \texttt{CT}_{x_i}$ no longer appears on the boundary paths, we update $\texttt{CT}_{x_i}$ by replacing v with all nodes that are reachable from v via regular access and that lie on the current boundary path.

If $\texttt{CT}_{x_i}$ ever becomes empty, this indicates that T(CT) can no longer be satisfied for x_i. On the other hand, once any node in $\texttt{CT}_{x_i}$ finds a matching negative node that satisfies the T(CT) condition, we consider the constraint for x_i satisfied and can safely discard $\texttt{CT}_{x_i}$ from further tracking.

Complexity for L It is worth noting that maintaining the T(CT) constraint introduces only minimal overhead. Since the number of nodes in each simplified subgraph $G'_{i,j}$ is bounded, the number of elements in each $\texttt{CT}_{x_i}$ set remains constant throughout the parsing process. Consequently, the additional computation required to update and check T(CT) constraints does not affect the overall complexity. The total time complexity remains $O(n^3)$.

4.4 Number of Derivations

An additional advantage of our chart-based parser is its ability to naturally track all valid derivations for a given sequent. During parsing, the algorithm maintains distinct derivation paths, allowing it to enumerate all possible syntactic analyses.

4.5 Experimental Results

To illustrate the practical performance and correctness of our parser, we apply it to LCGbank (Bhargava et al., 2024), a Lambek Categorial Grammar variant of the Penn Treebank (Marcus et al., 1993) containing 44,870 labeled sentences. Our parser successfully derives every sentence in the dataset, demonstrating both the efficiency and robustness of the proposed algorithm.

5 Conclusion

In this work, we presented a practical and efficient parser for bounded-order, product-free Lambek Categorial Grammar, based on a refined term-graph framework. Inspired in part by the theoretical insights from cyclic linear logic (Pentus, 2010), our algorithm achieves the same asymptotic complexity as Fowler (2007)'s chart-based parser at $O(n^3)$, while offering significantly improved simplicity and implementability. In contrast, Pentus's original method incurs a higher $O(n^4)$ complexity due to the need for additional structural tracking in cyclic linear logic.

A key innovation in our parser lies in the use of boundary-only representations, which eliminate the need to track internal nodes and allow for aggressive graph simplification without sacrificing correctness. Our method unifies term graph derivability conditions with a lightweight dynamic programming architecture, resulting in a parser that is both fast and easy to understand. Crucially, we provide the first open-source parser that is bounded-order polynomial, enabling further research and application of Lambek grammar in modern NLP workflows.

References

Shraddha Barke, Emmanuel Anaya Gonzalez, Saketh Ram Kasibatla, Taylor Berg-Kirkpatrick, and Nadia Polikarpova. 2024. HYSYNTH: Context-free LLM approximation for guiding program synthesis. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*.

Aditya Bhargava, Timothy A. D. Fowler, and Gerald Penn. 2024. LCGbank: A corpus of syntactic analyses based on proof nets. In *Proceedings of the 2024 Joint International Conference on Computational Linguistics, Language Resources and Evaluation (LREC-COLING 2024)*, pages 10225–10236, Torino, Italia. ELRA and ICCL.

Aditya Bhargava and Gerald Penn. 2020. Supertagging with CCG primitives. In *Proceedings of the 5th Workshop on Representation Learning for NLP*, pages 194–204, Online. Association for Computational Linguistics.

Armin Buch. 2009. Mildly non-planar proof nets for ccg. *OF THE EUROPEAN SUMMER SCHOOL FOR LOGIC, LANGUAGE, AND INFORMATION*, page 160.

Francesco Cagnetta and Matthieu Wyart. 2024. Towards a theory of how the structure of language is acquired by deep neural networks. *Preprint*, arXiv:2406.00048.

Stephen Clark. 2015. The java version of the c&c parser: Version 0.95.

Jochen Dörre. 1996. Parsing with semidirectional Lambek grammar is NP-complete. In *34th Annual Meeting of the Association for Computational Linguistics*, pages 95–100, Santa Cruz, California, USA. Association for Computational Linguistics.

Timothy AD Fowler. 2007. A polynomial time algorithm for parsing with the bounded order lambek calculus. In *Conference on Mathematics of Language*, pages 36–43.

Timothy AD Fowler. 2008. Efficiently parsing with the product-free lambek calculus. In *Proceedings of the 22nd International Conference on Computational Linguistics (Coling 2008)*, pages 217–224.

Jean-Yves Girard. 1989. Towards a geometry of interaction. *Categories in computer science and logic*, 92:69–108.

Julia Hockenmaier and Mark Steedman. 2007. Ccgbank: a corpus of ccg derivations and dependency structures extracted from the penn treebank. *Computational Linguistics*, 33(3):355–396.

Konstantinos Kogkalidis and Michael Moortgat. 2023. Geometry-aware supertagging with heterogeneous dynamic convolutions. In *Proceedings of the 2023 CLASP Conference on Learning with Small Data (LSD)*, pages 107–119, Gothenburg, Sweden. Association for Computational Linguistics.

Joachim Lambek. 1958. The mathematics of sentence structure. *The American Mathematical Monthly*, 65(3):154–170.

Mitch Marcus, Beatrice Santorini, and Mary Ann Marcinkiewicz. 1993. Building a large annotated corpus of english: The penn treebank. *Computational linguistics*, 19(2):313–330.

Jingcheng Niu, Wenjie Lu, and Gerald Penn. 2022. Does BERT rediscover a classical NLP pipeline? In *Proceedings of the 29th International Conference on Computational Linguistics*, pages 3143–3153, Gyeongju, Republic of Korea. International Committee on Computational Linguistics.

Gerald Penn. 2004. A graph-theoretic approach to sequent derivability in the lambek calculus. *Electronic Notes in Theoretical Computer Science*, 53:274–295. Proceedings of the joint meeting of the 6th Conference on Formal Grammar and the 7th Conference on Mathematics of Language.

Mati Pentus. 2006. Lambek calculus is np-complete. *Theoretical Computer Science*, 357(1):186–201. Clifford Lectures and the Mathematical Foundations of Programming Semantics.

Mati Pentus. 2010. A polynomial-time algorithm for lambek grammars of bounded order. *Linguistic Analysis*, 36(1):441–471.

Rahul Ramesh, Ekdeep Singh Lubana, Mikail Khona, Robert P. Dick, and Hidenori Tanaka. 2024. Compositional capabilities of autoregressive transformers: A study on synthetic, interpretable tasks. *Preprint*, arXiv:2311.12997.

Dirk Roorda. 1991. *Resource Logics: Proof-Theoretical Investigations*. Ph.D. thesis.

Yu V Savateev. 2009. Recognition of derivability for the lambek calculus with one division. *Moscow University Mathematics Bulletin*, 64:73–75.

Yury Savateev. 2012. Product-free lambek calculus is np-complete. *Annals of Pure and Applied Logic*, 163(7):775–788. The Symposium on Logical Foundations of Computer Science 2009.

Lena Strobl, William Merrill, Gail Weiss, David Chiang, and Dana Angluin. 2024. What formal languages can transformers express? a survey. *Transactions of the Association for Computational Linguistics*, 12:543–561.

Yuanhe Tian, Yan Song, and Fei Xia. 2020. Supertagging Combinatory Categorial Grammar with attentive graph convolutional networks. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 6037–6044, Online. Association for Computational Linguistics.

Katherine Wu and Yanhong A Liu. 2025. Lp-lm: No hallucinations in question answering with logic programming. *arXiv preprint arXiv:2502.09212*.

Ryosuke Yamaki, Tadahiro Taniguchi, and Daichi Mochihashi. 2023. Holographic CCG parsing. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 262–276, Toronto, Canada. Association for Computational Linguistics.

David N Yetter. 1990. Quantales and (noncommutative) linear logic. *The Journal of Symbolic Logic*, 55(1):41–64.

Jinman Zhao and Gerald Penn. 2024. LLM-supertagger: Categorial grammar supertagging via large language models. In *Findings of the Association for Computational Linguistics: EMNLP 2024*, pages 697–705, Miami, Florida, USA. Association for Computational Linguistics.