A Generative Model For Lambek Categorial Sequents

Jinman Zhao, Gerald Penn
Department of Computer Science
University of Toronto
Toronto, Canada
{jzhao, gpenn}@cs.toronto.edu

Abstract

In this work, we introduce a generative model, PLC+, for generating Lambek Categorial Grammar(LCG) sequents. We also introduce a simple method to numerically estimate the model's parameters from an annotated corpus. Then we compare our model with probabilistic context-free grammars (PCFGs) and show that PLC+ simultaneously assigns a higher probability to a common corpus, and has greater coverage. Our code is available at https://github.com/zhaojinm/Probabilistic_Lambek_Categorial_Sequent.

Keywords: Lambek Categorial Grammar, PCFG, stochastic

1. Introduction

The success of large language models like GPT (et al., 2020), LLaMa (et al., 2023) and PaLM (et al., 2022) has cast a shadow over the role of grammar in NLP: perhaps scaling up language models and feeding them massive amounts of data will converge on a similar syntactic mastery of language. These large models have their drawbacks: they demand extensive resources and are challenging to interpret and scrutinize. More importantly, they require vast amounts of data. Some work, furthermore, has shown that syntax-based models can improve the performance of transformer-based language models (Bai et al., 2021; Zhang et al., 2022), text generation with autoencoders (Zhang et al., 2019), named entity recognition (Yu et al., 2020) and so on. Therefore, grammar is still essential for NLP research.

It probably is true, however, that our effective choice of context-free grammars (CFGs) for annotating syntactic corpora has made it relatively easy to disparage the use of grammar altogether. CFGs massively over-generate. While statistical methods help us choose the right tree for grammatical sentences to some extent, the underlying discrete formalism does impose limits upon this, as it does upon how much probability must be wasted upon ungrammatical sentences. Besides, CFG articulates a view of syntax with almost no connection to semantics, and even then cannot capture long-distance dependencies.

Categorial Grammars (CGs) such as Lambek Categorial Grammar(LCG) addresses these issues. LCG connects syntax and semantics through a very transparent, compositional labeled-term-deduction system based upon the lambda calculus. LCG lexicalizes grammar to the point that it requires only four, fixed grammar schemata. Recent research in Quantum NLP, as discussed in the Wu et al. (2021) has seen the adoption of pregroup grammar for mapping linguistic structures into quantum systems. Pregroups are an efficient variant of LCG. While parsing with LCG is theoretically intractable (Pentus, 2006; Savateev, 2012), **?** demonstrates that natural-language corpora pose no real challenge to parsing because of empirical limits on the order of functional categories.

There are stochastic context-free grammars(Huang and Fu, 1971) for generative purposes. There have been stochastic frameworks proposed for CG before (Osborne and Briscoe, 1997), with the earliest merely transforming derivations from CFG directly. (Bonfante and de Groote, 2004) proposed a stochastic model for LCG that was not generative. To our knowledge, (Zhao and Penn, 2021) is the only stochastic generative model for LCG sequents, and even that has some observable drawbacks: the procedure is rather complicated, and sequents like:

- $NP/N/N\ N\ N \models N$, and
- $NP/N\ N/N\ N \models NP$

receive the same probability, even though the second is far more common for Noun Phrases.

In this work, we propose a generative model that fixes these drawbacks. Section 2 presents the background information necessary to follow the technical details. In section 3, we present a method for representing LCG sequent derivations in a more tree-like structure that simplifies the overall formalism, separating the stochastic generation of a proof net into a phase that generates the tree first, followed by a phase that generates a proof net, the LCG equivalent of a phrase structure tree, directly from this tree. In section 4, we compare our model with PCFGs on a corpus using MLE.

LREC-COLING 2024, pages 584–593
20-25 May, 2024. © 2024 ELRA Language Resource Association: CC BY-NC 4.0

2. Preliminaries

2.1. Lambek Calculus

Lambek (1958) first proposed Lambek calculus(L). $Prim = \{p_1, p_2, p_3, \ ...\}$ is a set of letters. There are three binary connectives $/, \backslash$ and $\cdot$. The primitive types, together with their closure under the available connectives, will be called *categories*, also known as *types*. The intuition is that a functional category such as $S\backslash NP$ takes an NP on its left to produce an S, as a verb phrase would, and dually, a category such as NP/N takes a noun on its right to produce an NP, as a determiner would. This is not the traditional notation for LCG, which is far less readable, but it is the one we will use in this paper.

There are several variations of the Lambek calculus itself, the system of rules for deriving sequents with these types. We will focus on the variant in which the LHS of a sequent cannot be empty, and only types with $/$ and $\backslash$ connectives are allowed, the so-called product-free fragment of the calculus L.

2.2. Proof Nets

Proof nets are commonly used to determine the derivability of an LCG sequent. Roorda (1991) adapted the proof nets of linear logic to the Product-free Lambek calculus. Constructing a (Lambek) proof net from a sequent consists of four steps:

1. Labeling each category. All LHS categories will receive a negative polarity and the RHS category will receive a positive polarity. We also label each category with a unique variable.

2. Unfolding a sequent to *terminal* formulae. Apply the substitution rules to each labelled category from the previous step recursively until no more rules can be applied:

$$(A\backslash B)^{-}{:}t \rightarrow A^{+}{:}u \ \ B^{-}{:}tu \quad (1)$$
$$(A\backslash B)^{+}{:}v \rightarrow B^{+}{:}v' \ \ A^{-}{:}u[v := \lambda u.v'] \quad (2)$$
$$(A/B)^{-}{:}t \rightarrow A^{-}{:}tu \ \ B^{+}{:}u \quad (3)$$
$$(A/B)^{+}{:}v \rightarrow B^{-}{:}u \ \ A^{+}{:}v'[v := \lambda u.v'] \quad (4)$$

When unfold positive-polarity categories with variable v, new variables u and v' will be added to the proof framework. v is known as a *lambda node* or *lambda variable*. u and v' will likewise be referred to as the *daughters* of v.

3. Adding a half planar linkage between formulae. Link each pair of formulae that has same primitives and opposite polarities. The edges of half-planar graph will be known as *axiom links*.

4. Variable substitution.

Let us consider the sequent for "Who eats cake?" as an example:

$$S/(NP\backslash S) \ \ (NP\backslash S)/NP \ \ NP \models S$$

One of the possible linkages Figure 1:

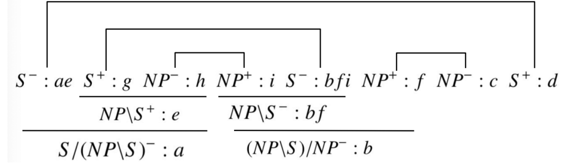


Figure 1: Example proof net.

2.3. LC-graph

The LC-graph(Penn, 2004) of a proof net is a directed graph $G = \langle V, E\rangle$, such that V is the set of all variables that appear in its category labels and E is the smallest set such that:

- for every $v \in V$, if v is a lambda-variable, then for both daughter variables of v, u and v', $(v, u) \in E$, and $(v, v') \in E$, and
- for every axiom link matching $p^+ : u$ and $p^- : t$ and for every v in the string t, $(u, v) \in E$.

The LG-graph for the above linkage and sequence of axiomatic formulae is Figure 2.

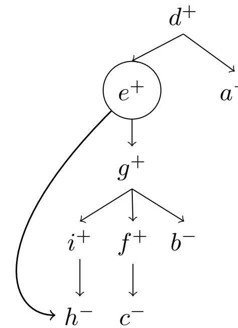


Figure 2: LC-graph example.

An LCG sequent is derivable iff the following three *integrity criteria* are true of its LC graph, G:

- **I(1)** there is a unique node in G with in-degree 0, from which all other nodes are path-accessible.
- **I(2)** G is acyclic.
- **I(3)** for every lambda-node $v \in V$, there is a path from its plus-daughter, v', to its minus-daughter, u.

- **I(CT)** for every lambda-node $v \in V$, there is a path in G, $v \rightsquigarrow x$, such that x labels a negative-polarity category, x has out-degree 0 and there is no lambda-node $v' \in V$ such that $v \rightsquigarrow v' \rightarrow x$.

Then we say that G is *integral*.

3. Simplifying the LC-Graph

LC-graphs are very useful for checking derivability, but not for generation, because the mapping between proof nets and LC-graphs is not bijective. We perform the following modifications.

3.1. Imposing Variable Order

For every axiom link matching $p^+ : u$ and $p^- : t$, t is a string $t_1 ... t_n$, where t_i are children of u. Let t_1 be the leftmost child of u, t_2 be the second leftmost child, and so on. We will reflect this in our depiction of the above graph, as shown in Figure 3.

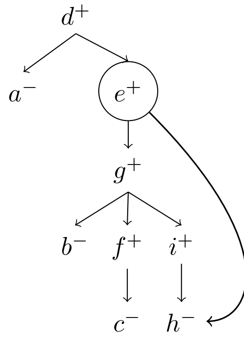


Figure 3: Ordered Variables

3.2. Adding Edge and Node Labels

We will add labels to some edges. For each edge:

1. If u is a positive non-lambda node and v is positive, add as a label the connective that we use in the unfolding rule when v is created.

2. Otherwise, add nothing.

We will label every node. Some of these labels will resemble primitive types. For each node u, define its *augment* as $\psi(u)$:

1. If u is a lambda node, $\psi(u) = \lambda$.

2. If u is a positive non-lambda node, then there must be exactly one axiomatic formula $p^+ : u$. Let $\psi(u) = p$.

3. If u is a negative non-lambda node, then there must be exactly one axiomatic formula $p^- : t$, where t is a string and $u \in t$. Let $\psi(u) = p$

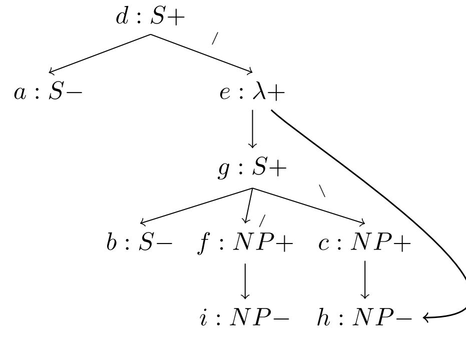


Figure 4: Adding augment.

The above graph changes to the one shown in Figure 4.

We name this augmented graph **LC-graph+**. For each LCG sequent, all primitive types, lambda nodes and connectives are stored in the LC-graph+ with a certain arrangement.

Theorem 3.1. *The mapping from proof nets to LC-graphs+ by the above procedures is* ***bijective****.*

The proofs of this and the other theorems of this paper can be found in Section 6.

3.3. Pruning Extra Edges

Theorem 3.2. *Let P be a valid proofnet and $G = \langle V, E \rangle$ be its corresponding integral LC-graph+. Let $E' = \{(u,v)|(u,v) \in E, u\ is\ a\ lambda\ node\ , v\ is\ negative\}$ and $T = \langle V, E - E' \rangle$. Given T, the rest of G can be inferred.*

This means that drawing the edge from the lambda node to its negative daughter is not necessary. Thus, the above graph can be represented as a tree. But we can further simplify this tree. The variable of each node itself is redundant since all variable names are distinct and the mapping from proof nets to LC-graphs+ is injective. Also, if an LC-graph is integral, it is obvious that leaves are only negative and all internal nodes are positive. So we can eliminate the polarities of nodes. In addition, each leaf must have the same primitive type as its parent. So all leaves' polarities and primitives are determined by the internal nodes. This means we do not need to draw any of the leaves in the previous tree. We will call the result of these changes, Figure 5, the **LC-tree**.

One assumption of Theorem 3.2 is that the LC-graph+ is integral. Removing edges from the lambda node to its negative daughter makes the LC-tree unsuitable for checking the derivability of prior input because we lose some essential information. But we are focusing on generating a string in this work.

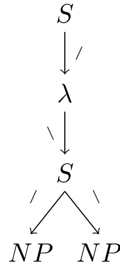


Figure 5: LC-graph+ simplified into an LC-tree.

3.4. Integrity Condition

Given LCG proof nets for a training corpus, we first transfer them to LC-trees, then learn how to generate these trees using different machine learning methods such as neural networks, MLE, etc. But not all trees with primitive types as nodes slashed edges can be called LC-trees. There is one remaining condition left over from L in order for it to correspond to a valid proof net:

- T(CT): For each sub-tree at v, the number of nodes labelled λ is less than the number of nodes not labelled λ.

The mapping of LC-trees for which this condition holds to proof nets is bijective. We call these *integral trees.*

Without this extra condition, it is relatively simple to generate LC-trees: we can simply use a top-down (i.e., LHS-normalized) PCFG for LC-trees and their labels. T(CT) interferes with this because it violates the context-freeness of the generated subtrees, and would require a balance condition to ensure that the number of λ-nodes is bounded. Our approach is to use a top-down PCFG anyway, without T(CT), and then to repair the broken trees.

3.5. Repairing T(CT)

For any tree that violates this condition, we can do the following recursively until it holds:

1. Pick any node u such that $u = \lambda$, its child tree satisfies T(CT), but the tree rooted at u violates T(CT).
2. Delete u and the edge from u to its child.
3. Connect u's parent to u's child.

Figure 6 provides an example of repairing a tree in one step. We can repeatedly apply this procedure, until T(CT) is satisfied. Because the tree only becomes smaller, termination is guaranteed. Because only λ-nodes are removed, T(CT) will eventually be satisfied.

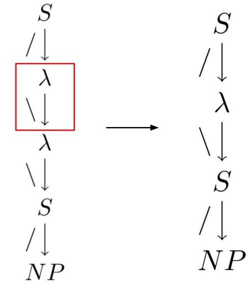


Figure 6: Left tree violates integral, we fix it by removing nodes and edges in the red box.

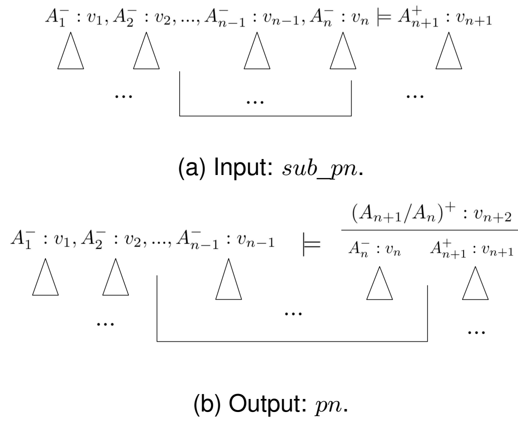
3.6. From Tree To Proof Net

Given a tree that is generated by the model, we can build a proof net recursively from proof nets of sub-tree(s). Algorithm 1 demonstrates such a procedure.

Algorithm 1: Generating a proof net from a tree.

```
   Result: proofnet
1  Algorithm gen(t)
2    p = t.prim;
3    if t.child = NIL then
4      return proof net of p ⊨ p;
5    con = (t, t.child[0]).connective;
6    if c.prim = λ & con = / then
7      sub_pn = gen(c);
8      if len(sub_pn LHS)=1 then
9        return sub_pn;
10     else
11       pn = sub_pn apply Figure 7;
12       return pn;
13   else if c.prim = λ & con = \ then
14     sub_pn = gen(c);
15     if len(sub_pn LHS)=1 then
16       return sub_pn;
17     else
18       pn = sub_pn apply Figure 8;
19       return pn;
20   pn = proof net of p ⊨ p;
21   foreach c ∈ t.child do
22     sub_pn = gen(c);
23     con = (t, c).connective;
24     if con = / then
25       pn = pn and sub_pn apply
           Figure 9
26     else
27       pn = pn and sub_pn apply
           Figure 10
28   end
29   return pn;
```

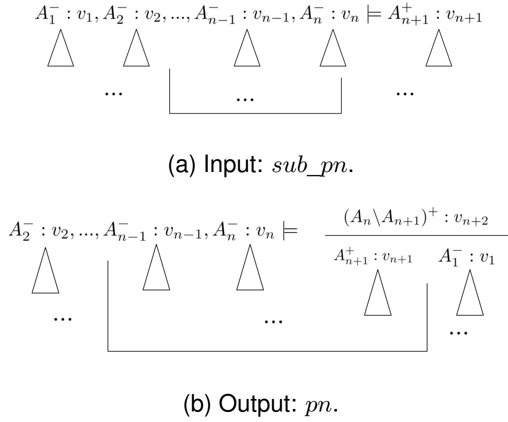
Instead of actually modifying the tree during repair, Algorithm 1 merely ignores those λ-nodes



(a) Input: sub_pn.

(b) Output: pn.

Figure 7: We do not change the sequence of *terminal* formulae, and the linkage, only step is to combine left most two categories using the inverse of unfolding rule 4. This will create a new variable v_{n+2}. And change the proofnet of $A_1, A_2, ..., A_{n-1}, A_n \models A_{n+1}$ to $A_1, A_2, ..., A_{n-1} \models A_{n+1}/A_n$.



(a) Input: sub_pn.

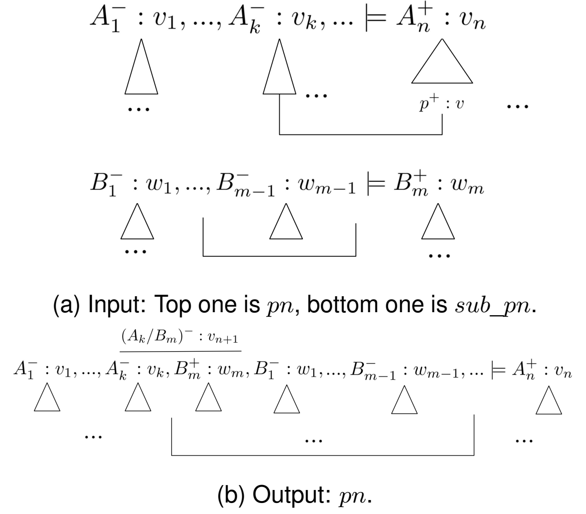
(b) Output: pn.

Figure 8: We move all the formulae generated by first category to the end. So A_1 becomes the left most category in the sequent. But do not change the endpoints of linkages. Linkage is still planar. Then combine left most two categories using the inverse of unfolding rule 2. This will also create new variable v_{n+2}. This procedure will change $A_1, A_2, ..., A_{n-1}, A_n \models A_{n+1}$ to $A_2, ..., A_n \models A_1 \backslash A_{n+1}$.

while generating the corresponding proof net in lines 8 to line 19.

3.7. Probability Convergence

We must then define the probability of a proof net, P, as the sum of the probabilities of all of the trees for which there is a sequence of repairs ending in the integral tree that converts directly to P. There are infinitely many such trees, and yet that infinite sum does converge:



(a) Input: Top one is pn, bottom one is sub_pn.

(b) Output: pn.

Figure 9: Let $p^+ : v$ be the root of tree t, there is unique A_k that contains the axiom primitive that links to $p^+ : v$. First move B_m in the sub_pn from right most to left most without modifying the endpoints of the linkage. So it is still planar. Then move entire sub_pn inside pn between A_k and A_{k+1}. Then combine A_k and B_m by the inverse of unfolding rule 3. This procedure will result a new proofnet $A_1, ..., A_k/B_m, B_1, ..., B_{m-1}, A_{k+1}, ... \models A_n$.

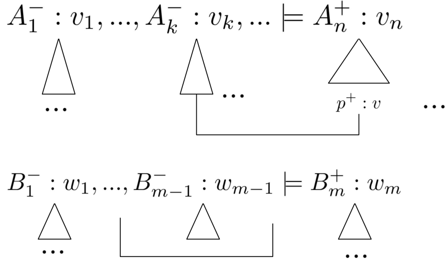
Theorem 3.3. *Given a proof net P and its corresponding integral LC-tree T, let $p_1 = Prob(T)$, $p_2 = Prob(\lambda \to \backslash\lambda) + Prob(\lambda \to /\lambda)$ and n be the number of nodes v in T such that, in the subtree rooted at v, the number of λ-nodes is exactly one less than the number of non-λ-nodes. Then:*

$$Prob(P) = \frac{p_1}{(1 - p_2)^n}$$

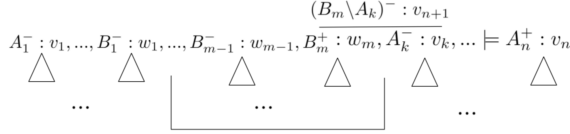
3.8. N-parent LCG

PLCG extends in the same way as Markov assumptions do. In the previous generative model, the current state only depends on one previous state. This assumption leads to a problem, however. For some sequent, there might be multiple ambiguous proof nets that derive them, each receiving the same probabilities. For example, $A/A \;\; A \;\; A\backslash A \models A$ has two LC-trees as shown in Figures 11 and 12. If the training set contains only one sample which is Figure 11, it has two production rules $A \to \backslash A$ and $A \to /A$. And both have conditional probability $\frac{1}{2}$. Both Figures 11 and 12 then receive $\frac{1}{4}$ as probabilities. But Figure 11 should have a higher probability than Figure 12 since this is the one attested in the training set.

We name the previous model Uni-parent PLC+, because it depends on only one previous state. We call the model that depends on two previous states Bi-parent PLC+. In Bi-parent PLC+, the training set that contains only Figure 11 will learn



(a) Input: Top one is pn, bottom one is sub_pn.



(b) Output: pn.

Figure 10: Let $p^+ : v$ be the root of tree t, there is unique A_k that contains the axiom primitive that links to $p^+ : v$.We do not change anything to sub_pn. Move entire sub_pn inside pn between A_k and A_{k+1}. Then combine A_k and B_m by the inverse of unfolding rule 1. This procedure will result a new proofnet $A_1, ..., B_1, ..., B_{m-1}, B_m \backslash A_k, A_{k+1}, ... \models A_n$.

two production rules: $A \rightarrow \backslash A$ and $A \backslash A \rightarrow /A$. Therefore, Figure 12 will receive 0 probability, as wanted.

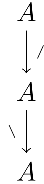


Figure 11: LC-tree for $A/A \;\; A \;\; A \backslash A \models A$.

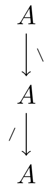


Figure 12: LC-tree for $A/A \;\; A \;\; A \backslash A \models A$.

4. Results

4.1. Dataset

LCGbank(Bhargava et al., 2024) is a conversion of CCGbank(Hockenmaier and Steedman, 2007) to LCG. CCGbank in turns a conversion of the Penn TreeBank(Marcus et al., 1993). We train and test on LCGbank, using section 23 as the test set, which contains 2416 proof nets. For the training set, we use sections 1-22 and 24, which contain 44870 proof nets. Unlike the Penn TreeBank, which uses more than 50 POS tags, LCGBank only has 4 primitives: NP, S, N and conj,

4.2. Comparison with Probabilistic Context Free Grammar

We name our models *-parent PLC+, and compare our results with PLC(Zhao and Penn, 2021) and PCFG, all using MLE parameter estimation. The PCFG is trained and tested on the PTB with the same sections. Also, we exclude the lexica which, for PLC(+), means that we are generating sequents instead of sentences. For PCFG, this means we only generate POS tag sequences, for uniformity.

Table 1 shows the coverage of six models, including four PLC+ variants. PCFG assigns zero probability to 272 sentences in the test set. That is because many production rules in the PTB test set never appear in the training set. PLC is capable of generating all sequents in the test set. Uni-parent PLC+ cannot generate all test set sequents, although only 8 are missing. Even Quad-parent PLC+ has better coverage than PCFG.

	$P \geq 0$	$P = 0$
PCFG	2144	272
PLC	2416	0
Uni-parent PLC+	2408	8
Bi-parent PLC+	2373	43
Tri-parent PLC+	2283	133
Quad-parent PLC+	2149	267

Table 1: Number of sentences receiving positive and zero probabilities.

Table 2 shows that the majority of sequents are assigned a higher probability by all of the PLC+ variants than by PCFG.

	>0	<0	0
Uni-parent PLC+	528	1883	5
Bi-parent PLC+	298	2098	20
Tri-parent PLC+	234	2136	46
Quad-parent PLC+	205	2132	79

Table 2: Sign of $p(PCFG) - p(PLC{+})$.

Table 3 presents the normalized negative log likelihood (NLL) of the test set, which is calculated by adding the log probabilities of each sequent and then dividing by the total number of tokens. Note that we only compute NLL using inputs with

positive probability. These two tables together suggest that PLC+ is far better at saving probability for sequents that we know to exist.

Although PLC has excellent coverage, its NLL performance is very poor. It does not generalize as well beyond its training data. PLC+ achieves great NLL with just a slight loss in coverage.

PCFG	2.95
PLC	4.06
Uni-parent PLC+	2.36
Bi-parent PLC+	1.96
Tri-parent PLC+	1.75
Quad-parent PLC+	**1.71**

Table 3: Negative log probability of test set normalized by the number of tokens.

4.3. LCG Proofnet Ranker

We can also employ PLC+ for ranking proof nets. A given LCG sequent may have numerous possible proof nets (“matches”). In Figure 13, the sequent with the maximum number of proofnets reads slightly over 14 on a (natural) log scale, which corresponds to more than 2 million possible proof nets.

Only one of these is the gold-standard proof net found in the corpus, however. As shown in Figure 14, 419 sequents among the 2285 within the test set of length ≤ 40 have a unique proof net. An additional 205 sequents have two possible proof nets (matches), and so forth. 567 sequents have more than 50 possible proof nets. The Bi-parent PLC+ assigns the highest probability to the gold-standard proof net for 1329 of these sequents (including the 419 with only one to offer). The gold standard ranks second for another 264 sequents and so on. The Bi-parent variant was chosen for this figure because it has the best distribution. This demonstrates the power of the probabilistic model for proofnet disambiguation. On the other hand, when it is wrong, it can be very wrong. While the median gold-standard rank is 1, the mean is 82.

5. Conclusion and Future work

In this work, we have presented a probabilistic generative model for sequent derivability in the Lambek calculus. We compared it with PCFG and with the only other known approach to generative LCG using MLE, both trained and tested on comparable corpora. The results show that PLC+ has the highest data likelihood and the best overall combination of data likelihood and test-set coverage.

The performance of PLC+ may be further improved if we use a different corpus that utilizes

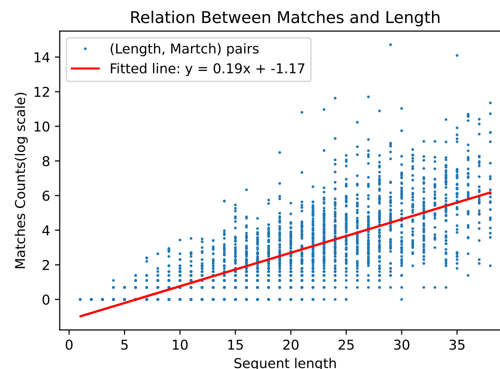


Figure 13: Number of proof nets for sequents in the LCGbank test set of length ≤ 40, sorted by sequent length. The red line is the natural-log-scale linear regression between number of proof nets and length, which has a slope of 0.19.

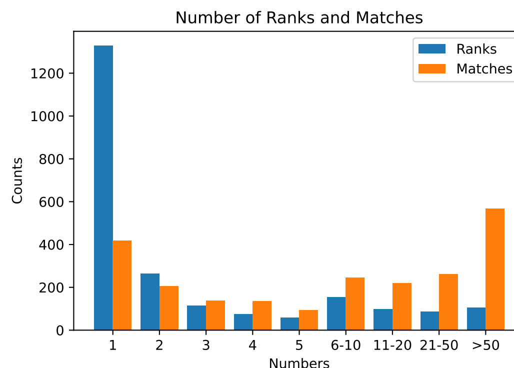


Figure 14: Gold-standard ranks and number of proof nets for sequents in the test set with length ≤ 40.

more primitives than just 4. This model can also be used to determine the best proof net for a sequent — the first such algorithm known to exist for the Lambek Calculus. This opens the door to using LCG for other downstream NLP tasks.

6. Proofs

Lemma 6.1. *Let f be a mapping from proofnets to LC-graphs+ by the above procedures. f is* ***surjective****.*

Proof. Induction on the number of nodes. We want to prove that, for each LC-graph+ G, there exists a proofnet P such that $f(P) = G$. We denote the root of G as a.
Base case: $n = 2$.
Let p denote the primitive of each node. The proofnet of $p \models p$ generates the LC-graph+.
Inductive Step:
case11: the root is a lambda node.
Then root only has one positive child. b denotes

this node. Then by IH, there must exist a proofnet P_1 of sequent A_1 ... $A_n \models A_{n+1}$ that generates the LC-graph+ rooted at b. If the augment of edge $a \to b$ is $/$, then the proofnet P of A_1 ... $A_{n-1} \models A_{n+1}/A_n$ generates G. Otherwise, the proofnet P of A_2 ... $A_n \models A_1\backslash A_{n+1}$ generates G. P and P_1 have the same linkages.
case 2: the root is not a lambda node.
Let p be the augmented primitive of the root. Let b_1 ... b_m be the positive children of a, with this order. Each LC-graph+ rooted at b_i can be generated by a proofnet P_i such that $f(P_i) = b_i$. Each P_i is a proofnet for the sequent $\Gamma_i \models X_i$. Then the following proof net generates G, starting from the proof net of $p \models p$. Let A be the LHS category that contains p:

- for i:=1 to m:
- if the augment of the edge $a \to b_i$ is $/$, update P as the proofnet of $A/X_i \;\; \Gamma_i \models p$.
- if the augment of the edge $a \to b_i$ is $\backslash$, update P as the proofnet of $\Gamma_i \;\; X_i\backslash A \;\models p$.

Apparently all linkages in P_i are also in P, with an extra link from root to its negative child. □

Lemma 6.2. *Let f be a mapping from proofnets to LC-graphs+ by the above procedures. f is* ***injective****.*

Proof. Induction on the number of nodes in the LC-graph+. We want to prove that if $f(P_1) = f(P_2) = G$ then $P_1 = P_2$.
Base case: $n = 2$. Let p be the augmented primitive of the root of G. The only case is $p \models p$.
Inductive Step:
case 1: The root is a lambda node.
Let a be the root of the LC-graph+ and b, the positive child of a. Assume P_1 and P_2 are different.
First consider $a \to b = /$. P_1 is for the sequent A_1 ... $A_n \models A_{n+1}/A_{n+2}$ and P_2 is for the sequent B_1 ... $B_m \models B_{m+1}/B_{m+2}$. There must exist a proofnet P_1' such that all the links are the same as in P_1 but the sequent is A_1 ... A_n $A_{n+2} \models A_{n+1}$. Also, there must exist a proofnet P_2' such that all links are the same as in P_2 but the sequent is B_1 ... B_m $B_{m+2} \models B_{m+1}$. If $P_1 \neq P_2$ then $P_1' \neq P_2'$. But $f(P_1') = f(P_2')$, which is the LC-subgraph+ rooted at b, so by IH, $P_1' = P_2'$, which is a contradiction.
The proof of the dual case $a \to b = \backslash$ is the same, except P_1 is for the sequent A_1 ... $A_n \models A_{n+1}\backslash A_{n+2}$, P_2 is for the sequent B_1 ... $B_m \models B_{m+1}\backslash B_{m+2}$, P_1' is for the sequent A_{n+1} A_1 ... $A_n \models A_{n+2}$, and P_2' is for the sequent B_{m+1} B_1 ... $B_m \models B_{m+2}$.
case 2: The root is not a lambda node.
case 2a: G contains at least one lambda node. Let b be any anterior lambda node. Let G' be the LC-graph+ of G but remove the sub-graph rooted at b. There must exist a unique proofnet $P1$ for sequent $\Gamma \models A$ for the LC-graph+ rooted at b.
By Lemma 4.9 and Lemma 4.10 in (Zhao and Penn, 2021), everything in the tree rooted at b can be removed to form a new proofnet of G', say P'. By IH, P' is unique. $P1$ is also unique, and so the proofnet of G is also unique.
case2b: G contains no lambda node. This obvious case is proved by Lemma 4.12 in (Zhao and Penn, 2021) and IH. □

Theorem 6.3. *Let f be a mapping from proofnets to LC-graphs+ by the above procedures. f is* ***bijective****.*

Proof. From Lemma 6.1 and Lemma 6.2 □

Theorem 6.4. *Let P be a valid proofnet and $G = \langle V, E\rangle$ be its corresponding LC-graph+. Let $E' = \{(u, v)|(u, v) \in E, u\ is\ a\ lambda\ node\ , v\ is\ negative\}$ and $T = \langle V, E - E'\rangle$. Given T, the rest of G can be inferred.*

Proof. It is sufficient to prove that for each LC-Graph+ rooted at lambda-node a, if all its descendent lambda nodes are inferrable then the edge from a to its negative daughter is also inferrable. If this holds then G itself is inferrable from bottom to top.
Let b be the positive child of a. Consider the subgraph G' rooted at b.
case 1: edge $a \to b = /$. Theorem 6.3 implies that there exists a unique P' with sequent A_1 ... A_n $A_{n+1} \models A_{n+2}$ such that $f(P') = G'$. Consider the proof net P of sequent A_1 ... $A_n \models A_{n+2}/A_{n+1}$ that has same linkage as P'. The LC-graph+ should be the same as the subgraph of G rooted at a. By Lemma 6.2, such P is unique. So the edge from a to its negative daughter is also unique. a should link to the node representing the category A_{n+2}.
case 2: edge $a \to b = \backslash$. Similar to case 1, but P is for the sequent A_2 ... A_n $A_{n+1} \models A_1\backslash A_{n+1}$. □

Theorem 6.5. *Given a proof net P and its corresponding LC-tree T, let $p_1 = Prob(T)$, $p_2 = Prob(\lambda \to \backslash\lambda) + Prob(\lambda \to /\lambda)$ and n be the number of nodes v in T such that, in the subtree rooted at λ, (the number of lambda nodes) = (the number of non-lambda nodes) - 1. Then:*

$$Prob(P) = \frac{p_1}{(1 - p_2)^n}$$

Proof. Let us consider a simple case of LC-tree from the right side of Figure 6, where $n = 1$. Then all the trees in Figure 15 generate the same proofnet. So to compute the probability of the proofnet, we need to compute the probability of the sum of the trees. For the case $n > 1$, all the

trees with an extra lambda node right above the n nodes described in the statement of the theorem are able to generate P.

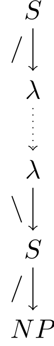


Figure 15: Set of trees that generates the same proofnet.

Let us define the tree $T_{k_1,\dots k_n}$ with n subscripts $k_i, 1 \leq i \leq n$. k_i decorates a tree that adds k_i extra lambda nodes above the ith node for which (the number of lambda nodes) = (the number of non-lambda nodes) - 1. Note that if $\forall i, k_i = 0$, then $T_{k_1,\dots k_n} = T$.

Let us consider the proposition that, for any m:

$$\sum_{\substack{k_i \geq 0 \forall i \leq m,\\ k_i = 0 \forall i > m}} prob(T_{k_1,\dots k_n}) = \frac{p_1}{(1-p_2)^m}$$

We prove this by induction on m.
base case: $m = 0$. then

$$\sum_{\substack{k_i \geq 0 \forall i \leq m,\\ k_i = 0 \forall i > m}} prob(T_{k_1,\dots k_n}) = Prob(T)$$

$$= p_1$$

$$= \frac{p_1}{(1-p_2)^n}$$

as wanted.
Inductive Step:

$$\sum_{\substack{k_i \geq 0 \forall i \leq m,\\ k_i = 0 \forall i > m}} prob(T_{k_1,\dots k_n})$$

$$= \sum_{k_m=0}^{\infty} \sum_{\substack{k_i \geq 0 \forall i \leq m-1,\\ k_i = 0 \forall i > m}} prob(T_{k_1,\dots k_n})$$

$$= \sum_{i=0}^{\infty} p_2^i \sum_{\substack{k_i \geq 0 \forall i \leq m-1,\\ k_i = 0 \forall i > m\\ k_m = 0}} prob(T_{k_1,\dots k_n})$$

$$= \sum_{i=0}^{\infty} p_2^i \sum_{\substack{k_i \geq 0 \forall i \leq m,\\ k_i = 0 \forall i > m}} prob(T_{k_1,\dots k_n})$$

$$= \sum_{i=0}^{\infty} p_2^i \frac{p_1}{(1-p_2)^{m-1}} \quad \text{ByIH} \qquad = \frac{p_1}{(1-p_2)^m}$$

Then we are able to compute $Prob(P)$, since all $T_{k_1,\dots k_n}$ are able to generate P by Algorithm 1. Therefore, the probability of P is :

$$Prob(P) = \sum_{k_i \geq 0 \forall i \leq n} prob(T_{k_1,\dots k_n})$$

$$= \frac{p_1}{(1-p_2)^n}$$

□

7. References

Jiangang Bai, Yujing Wang, Yiren Chen, Yaming Yang, Jing Bai, Jing Yu, and Yunhai Tong. 2021. Syntax-BERT: Improving pre-trained transformers with syntax trees. In *Proceedings of the 16th Conference of the European Chapter of the Association for Computational Linguistics: Main Volume*, pages 3011–3020, Online. Association for Computational Linguistics.

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*, Turin, Italy.

Guillaume Bonfante and Philippe de Groote. 2004. Stochastic lambek categorial grammars. *Electronic Notes in Theoretical Computer Science*, 53:34–40. Proceedings of the joint meeting of the 6th Conference on Formal Grammar and the 7th Conference on Mathematics of Language.

Aakanksha Chowdheryand et al. 2022. Palm: Scaling language modeling with pathways.

Hugo Touvron et al. 2023. Llama 2: Open foundation and fine-tuned chat models.

Tom B. Brown et al. 2020. Language models are few-shot learners. *CoRR*, abs/2005.14165.

Timothy A. D. Fowler. 2010. A polynomial time algorithm for parsing with the bounded order lambek calculus. In *The Mathematics of Language*.

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.

T. Huang and K.S. Fu. 1971. On stochastic context-free languages. *Information Sciences*, 3(3):201–224.

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

Mitchell P. Marcus, Beatrice Santorini, and Mary Ann Marcinkiewicz. 1993. Building a large annotated corpus of English: The Penn Treebank. *Computational Linguistics*, 19(2):313–330.

Miles Osborne and Ted Briscoe. 1997. Learning stochastic categorial grammars. In *CoNLL97: Computational Natural Language Learning*.

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.

M. Pentus. 1993. Lambek grammars are context free. In *[1993] Proceedings Eighth Annual IEEE Symposium on Logic in Computer Science*, pages 429–433.

Mati Pentus. 1997. Product-free lambek calculus and context-free grammars. *Journal of Symbolic Logic*, 62(2):648–660.

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

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

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.

Sixuan Wu, Jian Li, Peng Zhang, and Yue Zhang. 2021. Natural language processing meets quantum physics: A survey and categorization. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 3172–3182, Online and Punta Cana, Dominican Republic. Association for Computational Linguistics.

Juntao Yu, Bernd Bohnet, and Massimo Poesio. 2020. Named entity recognition as dependency parsing. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 6470–6476, Online. Association for Computational Linguistics.

Shuai Zhang, Wang Lijie, Xinyan Xiao, and Hua Wu. 2022. Syntax-guided contrastive learning for pre-trained language model. In *Findings of the Association for Computational Linguistics: ACL 2022*, pages 2430–2440, Dublin, Ireland. Association for Computational Linguistics.

Xinyuan Zhang, Yi Yang, Siyang Yuan, Dinghan Shen, and Lawrence Carin. 2019. Syntax-infused variational autoencoder for text generation. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 2069–2078, Florence, Italy. Association for Computational Linguistics.

Jinman Zhao and Gerald Penn. 2021. A generative process for Lambek categorial proof nets. In *Proceedings of the 17th Meeting on the Mathematics of Language*, pages 1–13, Umeå, Sweden. Association for Computational Linguistics.