Semantically Informed Data Augmentation for Unscoped Episodic Logical Forms

Mandar Juvekar*
Boston University
Boston, MA, USA 02215
mandarj@bu.edu

Gene Louis Kim
University of South Florida
Tampa, FL, USA 33620
genekim@usf.edu

Lenhart Schubert
University of Rochester
Rochester, NY, USA 14627
schubert@cs.rochester.edu

Abstract

Unscoped Logical Form (ULF) of Episodic Logic is a meaning representation format that captures the overall semantic type structure of natural language while leaving certain finer details, such as word sense and quantifier scope, underspecified for ease of parsing and annotation. While a learned parser exists to convert English to ULF, its performance is severely limited by the lack of a large dataset to train the system. We present a ULF dataset augmentation method that samples type-coherent ULF expressions using the ULF semantic type system and filters out samples corresponding to implausible English sentences using a pretrained language model. Our data augmentation method is configurable with parameters that trade off between plausibility of samples with sample novelty and augmentation size. We find that the best configuration of this augmentation method substantially improves parser performance beyond using the existing unaugmented dataset.[1]

1 Introduction

Kim and Schubert (2019) introduced Unscoped Episodic Logical Form (ULF) as a semantic representation that captures syntactic type structure within the Episodic Logic formalism, while staying close to the surface form for ease of annotation and parsing. Kim et al. (2021a) presented a learned approach to parsing English sentences to ULF which showed promising results. Their parsing efforts, however, were limited by the size of the training data available. They released a dataset of 1,738 sentences with corresponding manual ULF annotations alongside their parser which—to the best of our knowledge—remains the only dataset of ULF annotations to date. Our work aims to alleviate this limitation of data sparsity.

```
(|Mary| ((past place.v)
         |Glenn|
         (under.p (k anesthesia.n))))
```

Figure 1: An example ULF for the sentence "*Mary placed Glenn under anesthesia.*"

In this paper, we present a method of augmenting ULF datasets. Our method leverages ULF's underlying type structure and works by replacing subtrees of seed ULFs with other subtrees of the same semantic type. This, combined with the use of pretrained language models to prune out the most incoherent sentences, allows us to expand relatively small datasets of ULF, such as that of Kim et al. (2021a), into datasets several times larger in size. We evaluate the efficacy of our system by looking at the performance of the existing ULF parser when trained on augmented versions of the original training set.

The importance of our work, and more generally of ULF parsing, comes from the role of ULF in the broader Episodic Logic (EL) framework. Episodic Logic (EL) is an extended first-order logic designed to closely match the form and expressivity of natural language (Schubert, 2000). EL is a powerful representation with rich model-theoretic semantics which enable a variety of inferences including deductive inference, uncertain inference, and natural logic-like inference (Morbini and Schubert, 2009; Schubert and Hwang, 2000; Schubert, 2014). However, parsing ordinary English sentences into fully resolved EL forms is a difficult task.

ULF is an underspecified form of EL designed to balance encoding adequate semantic information with ease of parsing. It fully specifies the semantic type structure of EL by marking the types of the atoms and of all the predicate-argument relationships while leaving issues such as quantifier scope, word sense, and anaphora unresolved. ULF is the critical first step in parsing full-fledged EL formu-

*Work done in part while at the University of Rochester.

[1]The code is available at https://github.com/genelkim/subtree-sampled-ulf-data-augmentation.

las. A detailed description of how ULF fits into the EL interpretation process is given by Kim and Schubert (2019). ULF is also a useful interpretation in its own right. It is capable of generating inferences based on clause-taking verbs, counterfactuals, questions, requests, and polarity (Kim et al., 2019, 2021b,c), and has been an effective representation in schema-based story understanding (Lawley et al., 2019) and spatial reasoning-related dialogue (Platonov et al., 2020).

2 Background

ULFs are trees written in parenthesized list form. The leaves of these trees, which we will refer to as *atoms*, can be:

- Surface words marked with suffix tags of their semantic types (e.g. `.v`, `.n`, `.pro`, `.d` for verbs, nouns, pronouns, and determiners, respectively);
- Case-sensitive symbols such as names and titles marked with pipes (e.g. `|Glenn|`). Pipe-marked symbols may be left without a semantic tag, in which case they are interpreted as having an entity type;
- One of a closed set of logical and macro symbols (e.g. `k` and `mod-n` for denoting kind-forming and noun modifier-forming operators, respectively). These symbols have unique types and are left without suffix tags.

Figure 1 contains an example ULF for the sentence "*Mary placed Glenn under anesthesia.*" The different types of atoms described above are all present here. The names "Mary" and "Glenn" are enclosed in pipes and the other surface words have POS-related semantic tags (e.g. `place.v`). The type-shifter `k` is used to turn the nominal predicate `anesthesia.n` into a *kind*, which is an abstract individual whose instances are entities. The special operator `past` is used to specify the tense of the verb `place.v`.

As mentioned before, there is a machine learning-based parser to convert English sentences (Kim et al., 2021a) to ULFs. A brief description of how the parser works is given in Appendix A. In the other direction, Kim et al. (2019) introduced a simple ULF-to-English translator, `ulf2english`, which they reported as achieving 78% grammaticality. Broadly speaking, `ulf2english` works by analyzing the ULF type of each clause, adding morphological details based on that analysis, removing purely logical operators, and mapping logical symbols to their corresponding surface forms. A more up-to-date version (whose performance exceeds the evaluation in that paper to an unknown degree)[2] is used in our sampling system.

2.1 The ULF Type System

The EL/ULF type system is the backbone upon which our data augmentation system is built. The semantics of EL are defined over a domain of individuals denoted by $\mathcal{D}$ and a set of truth values denoted by $\mathbf{2}$. A set of situations $\mathcal{S} \subset \mathcal{D}$ consisting of first-class individuals provides the basis for intensionality.[3] Since EL is a first-order logic, the domain $\mathcal{D}$ contains all the individuals that can be spoken about directly. $\mathcal{D}$ not only contains ordinary individuals and situations, but also collections, kinds of entities, propositions, and more. Special type-shifting operators are used to access these other individuals. For example, the so-called *kind* operator `k` can be applied to the nominal predicate `dog.n` (i.e. `(k dog.n)`) to talk about "dogs" as a whole (as opposed to any particular dog or collection of dogs). Predicates can be thought of as true/false-valued functions that take a certain number of objects from the domain and a situation as input. Viewing that in a curried form gives us the type of an arbitrary predicate: $(\mathcal{D} \to (\mathcal{D} \to (\cdots \to (\mathcal{D} \to (\mathcal{S} \to \mathbf{2})) \cdots)))$. For convenience, we shorten this to $(\mathcal{D}^n \to (\mathcal{S} \to \mathbf{2}))$ where n is the number of $\mathcal{D}$s in the previous type.[4] For our purposes (where we are mostly concerned with ULFs) intensionality is not very relevant, and so henceforth we will abbreviate $(\mathcal{S} \to \mathbf{2})$ by $\widehat{\mathbf{2}}$. Since monadic predicates (type $(\mathcal{D} \to \widehat{\mathbf{2}})$) commonly occur in the type system, we will use $\mathcal{N}$ as a shorthand for $(\mathcal{D} \to \widehat{\mathbf{2}})$.

A couple of key differences exist between the ULF and EL type systems. ULF types may have syntactic restrictions, denoted by subscripts, e.g., a verbal monadic predicate is denoted by $\mathcal{N}_V$. Determiners are denoted with the type $(\mathcal{N} \to \mathcal{D})$, which anticipates their replacement in EL by a variable of type $\mathcal{D}$ bound by a restricted quantifier.

Each ULF atom can be one of a few related

[2] https://github.com/genelkim/ulf2english

[3] The description of EL semantics we give is informal and limited to our purposes. For a more detailed, formal discussion we recommend reading Schubert and Hwang (2000, pp. 9–14).

[4] For technical reasons, EL supplies the situations last.

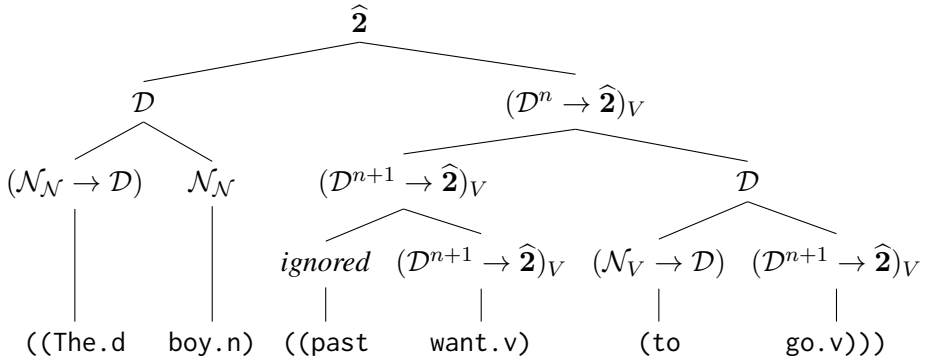


Figure 2: An example of how atomic ULF types combine to give the type of the ULF.

semantic types. Logical operators have a unique semantic type, whereas suffix-tagged atoms are restricted by the semantic types that correspond to their tags. A detailed correspondence between ULF atoms and their semantic types is given by Kim (2022, pp. 34–40). The types of atoms can *combine* (or *compose*) via function application to give the type of the ULF composed of those atoms. For example, `a.d` which has type $(\mathcal{N} \to \mathcal{D})$, and `dog.n` which has type $\mathcal{N}$ can compose to give `(a.d dog.n)`, with type $\mathcal{D}$. Such ULFs can further compose to give types for more complex ULFs. Figure 2 gives an example of such a type composition. Here, the entire ULF has type $\widehat{\mathbf{2}}$, the type for a complete sentence. Notice that `want.v` has type $(\mathcal{D}^{n+1} \to \widehat{\mathbf{2}})_V$. The variable n (taken to be a non-negative integer) is used to account for the fact that we do not have prior knowledge of how many arguments the verb takes. It is treated as an integer variable until the last step, where we instantiate it to 1 so that $(\mathcal{D}^n \to \widehat{\mathbf{2}})_V$ can combine with $\mathcal{D}$ to give $\widehat{\mathbf{2}}$. Such treatment is typical for verbs and other types that can take a variable number of arguments. We will call trees similar to the one in Figure 2 without the actual ULF atoms *type derivation trees*. A type derivation tree shows one way the types at the leaves can combine to give the type at the root.

All properly annotated ULFs, including ULFs that do not correspond to complete sentences, should have a valid type that can be found by composing the types of its atoms. This fact is what we use to build our ULF sampler. Our method of sampling ULFs produces new ULFs from a *seed* ULF by picking a random subtree of the seed, finding the semantic type of that subtree, and then replacing the subtree with another ULF of the same type. In our experiments, these seed ULFs are ULFs in the training set of the manually annotated ULF dataset released by Kim et al. (2021a). The type structure helps ensure that the result is a valid ULF where at least the composition of semantic types is coherent, and limiting our sampler to small subtrees makes sampling meaningful sentences significantly more likely than generating entire sentences from scratch.

3 System Description

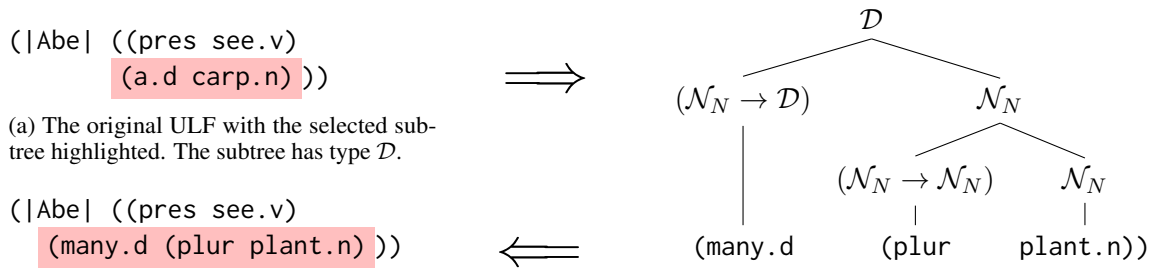
Our system can be broadly broken into two parts: a *sampler* that takes a single seed ULF as input and generates one new ULF-English pair, and a *handler* which uses the sampler repeatedly to augment a given dataset. Pseudocode for the salient parts of this process is given in Appendix E.

3.1 The Sampler

The sampler goes through four phases: (1) picking a random subtree, (2) finding its type, (3) sampling a ULF of that type, and (4) replacing the original subtree in-place. In this subsection we describe that process, illustrating it by walking through the process with the seed ULF `(|Abe| ((pres see.v) (a.d carp.n))` (see Figure 3 for an overview).

3.1.1 Picking a random subtree

This phase involves two parameters that can be tweaked: a maximum size M for the subtree picked, and a "recursion probability" p. Given these parameters and an input ULF, our algorithm first descends the ULF (viewed as a tree) top-down by picking uniformly random children at each level until it reaches a subtree with size (number of leaves) less than or equal to M. Then at each step where it is not at a leaf node it descends another level (by picking a random child) with probability p, and returns the subtree with the current node at its root with probability $1 - p$. If the algorithm ever



(a) The original ULF with the selected subtree highlighted. The subtree has type $\mathcal{D}$.

(c) The final sampled ULF with the replaced subtree highlighted.

(b) The derivation tree sampled for $\mathcal{D}$ with sampled atoms for the leaf nodes.

Figure 3: The sampling process illustrated.

reaches a leaf node it simply returns it. Pseudocode for this procedure is given in Algorithm 1 in Appendix E. In our running example (in Figure 3a) the recursion goes down the right side of the tree and stops with the subtree `(a.d carp.n)`.

3.1.2 Computing the subtree's type

The selected subtree's type is computed using ULF's type composition rules. We use a pre-existing ULF type system implementation[5] which finds the semantic type of a given ULF fragment by recursively composing types from the atoms in a bottom-up fashion. Due to the presence of variables in some types of leaf nodes (for example for verbs which can have multiple arities), the type system can return a list of possible types corresponding to different values of the variables. In such a case, we pick a random type from this list. Since variables in ULF type compositions rarely take high values (for example, verbs do not frequently take more than three arguments), we pick types corresponding to smaller values of the variable with higher probability. Specifically, if the number of options is less than 4, we pick uniformly. If the number of options is 4 or more, we pick uniformly from the first three options with probability $3/4$, and uniformly from all the options with probability $1/4$. Picking from multiple possible types in a more principled manner (for example by looking at the type composition tree of the seed ULF) could be an avenue for future work in improving our sampler.

Using this process, we find that the chosen subtree in the running example has type $\mathcal{D}$.

3.1.3 Sampling a ULF with a given type

This phase involves one parameter: the maximum size M' for the sampled ULF fragment; and takes one argument: τ_{root}, the desired ULF type (in our running example this is $\mathcal{D}$). To sample a ULF with the given type, we first sample a type derivation tree with τ_{root} at the root. Then, for each leaf type in the derivation tree, we sample a ULF atom with that type. Combining those atoms with the tree structure of the derivation tree gives us a ULF with the desired type.

Sampling a type derivation tree. Sampling a derivation tree is done via three functions: SAMPLETYPEDERIVATION, SAMPLETYPESOURCE, and SAMPLEARGDERIVATIONS. The top-level function is SAMPLETYPEDERIVATION which, as the name suggests, generates a type derivation tree with type τ_{root}. To do so it first uses SAMPLETYPESOURCE to sample a *source type*, τ_{src}, which is a type which can give τ_{root} when supplied 0 or more arguments *and* which is known to be the type of an atomic ULF. It then calls SAMPLEARGDERIVATIONS which takes τ_{root} and τ_{src} and returns a list of derivation trees for the argument types that need to be supplied to τ_{src} to obtain τ_{root}. Finally, SAMPLETYPEDERIVATION combines the source and argument into a derivation tree for τ_{root} which it returns.

SAMPLETYPESOURCE takes one argument, τ_0, and returns a type that can be combined with 0 or more arguments to obtain τ_0 and which can be the type of an atomic ULF. Let T be the set of all types that can be taken by atomic ULFs, and let μ_T be a distribution over T. We take μ_T to be the uniform distribution in our implementation. We leave the selection of a more informed distribution for future work.[6] SAMPLETYPESOURCE iteratively finds all the types in T that can combine with 0 or more arguments to give τ_0 and adds them to a set T'.

[5] https://github.com/genelkim/ulf-lib

[6] For example, while a four-argument verb is possible (e.g. in "I sold my car to John for $400."), it is far less likely than a one- or two-argument verb. A good choice for μ_T might account for that.

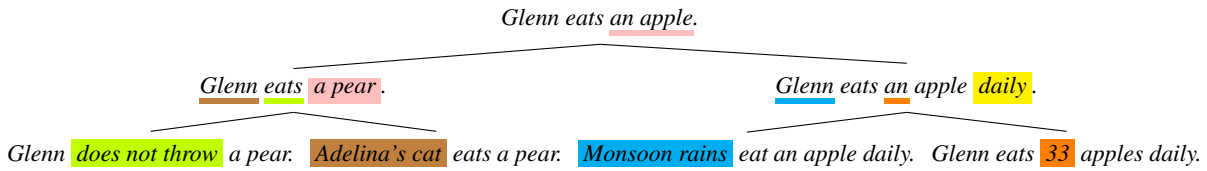


Figure 4: An example of what a tree of sentences (ULFs omitted for brevity) generated from the seed "*Glenn eats an apple*" with depth of 2 and branching factor of 2 might look like. Newly sampled text segments are highlighted. The corresponding replaced text segment in the parent (if any) is underlined with the same color.

It then returns a sample from T' with distribution weights from μ_T.Details on how exactly T' is computed are provided in Algorithm 2 in Appendix E.

SAMPLEARGDERIVATIONS takes parameters τ_{cur} and τ_{src}, and computes a list of derivation trees for types that can be composed with τ_{src} to get τ_{cur}. This starts with τ_{cur} and "grows" outward to get τ_{src}. It begins by finding the first type τ_{next} that needs to be prepended to τ_{cur} to get τ_{src}. For instance, if $\tau_{src} = (A \rightarrow (B \rightarrow (\mathcal{S} \rightarrow \mathbf{2})))$ and $\tau_{cur} = (\mathcal{S} \rightarrow \mathbf{2})$, then $\tau_{next} = B$. On finding τ_{next}, the algorithm makes a mutually recursive call to SAMPLETYPEDERIVATION to compute a derivation tree D_{next} for τ_{next}. It then recurses with $\tau_{cur} = (\tau_{next} \rightarrow \tau_{cur})$ and the same τ_{src} to obtain a list of derivations, ℓ_D. The algorithm returns $[D_{next}] + \ell_D$. Algorithm 2 in Appendix E contains pseudocode for the entire derivation tree sampling process.

Example. In our running example (Figure 3), the top-level function call is SAMPLETYPEDERIVATION($\mathcal{D}$). That function calls SAMPLETYPESOURCE($\mathcal{D}$), which returns the source type $(\mathcal{N}_N \rightarrow \mathcal{D})$. This is a valid source type since it can combine with one or more type to give $\mathcal{D}$, and since there are atoms (e.g. `the.d`) which have type $(\mathcal{N}_N \rightarrow \mathcal{D})$. The top level function then calls SAMPLEARGDERIVATIONS($\tau_{cur} = \mathcal{D}$, $\tau_{src} = (\mathcal{N}_N \rightarrow \mathcal{D})$). That function identifies that $\mathcal{N}_N$ can be combined with $(\mathcal{N}_N \rightarrow \mathcal{D})$ to get $\mathcal{D}$, and thus calls SAMPLETYPEDERIVATION($\mathcal{N}_N$).

In turn, that call does the same process as above, but with $\mathcal{N}_N$ as the root. It samples a source which, let us say, turns out to be $(\mathcal{N}_N \rightarrow \mathcal{N}_N)$. It then calls SAMPLEARGDERIVATIONS($\tau_{cur} = \mathcal{N}_N$, $\tau_{src} = (\mathcal{N}_N \rightarrow \mathcal{N}_N)$), which deduces that the required argument type is $\mathcal{N}_N$ and calls SAMPLETYPEDERIVATION($\mathcal{N}_N$) to find a derivation tree for the argument. In our example, the mutual recursion will end here: the call just mentioned will sample $\mathcal{N}_N$ as the source, which needs no further arguments to get to $\mathcal{N}_N$.

Putting everything together, this process leads to the derivation tree in Figure 3b.

Sampling ULF atoms. After generating a type derivation tree, we sample ULF atoms that have the types at the leaves of the derivation tree. Those atoms are then put in the structure induced by the derivation tree to obtain the ULF sampled. Sampling atoms with given types is done using the ULF lexicon used by Kim et al. (2021a). The sampling is weighted by probabilities computed by normalizing unigram counts from the Google n-gram dataset (Franz and Brants, 2006). In our example there are three leaf nodes with types $(\mathcal{N}_N \rightarrow \mathcal{D})$, $(\mathcal{N}_N \rightarrow \mathcal{N}_N)$, and $\mathcal{N}_N$. Suppose they are instantiated to the atoms `many.d`, `plur`, and `plant.n`.

3.1.4 Replacing in place

The final sampled ULF is obtained from the input ULF by replacing the random subtree picked in the first phase with the ULF fragment sampled in the previous phase. This is done using simple tree operations. In our example, this leads to the final sampled ULF, `(|Abe| ((pres see.v) (many.d (plur plant.n))))`.

3.2 The Handler

The sampling handler takes three inputs: the dataset that is to be augmented, a depth d, and a branching factor b. For each ULF U in the dataset, the handler performs the following steps:

1. Use the sampler b times on input ULF U to get b different samples from the seed U.
2. On each new ULF U' sampled in the previous step, use the sampler b times.
3. Repeat step 2 d times, thus obtaining a tree of ULFs with depth d and branching factor b. In this tree, each node is obtained from its parent via an application of the sampler. Figure 4 shows an example of such a tree.

4. Collect all the ULFs in this tree along with their English translations (which are found using the ULF-to-English library) into a set.

Combining all the sets obtained from the above process gives us a raw augmented dataset.

After generating a raw augmented dataset we assign a quality score using language model perplexity. The final dataset consists of the top $F * N$ ULF-English pairs according to the quality score, where N is total number of samples and F is a preset constant proportion ($0 \leq F \leq 1$). We use the GPT-2 language model (Radford et al., 2019) in our implementation. This last pruning step is done in order to remove highly incoherent results. Algorithm 3 contains pseudocode for the handler.

The reason we branch out from the seed instead of repeatedly modifying in a linear fashion is that in a linear design, if the sampler ever returns an incoherent result, every sentence generated from then onwards is likely to be incoherent too. This leads to a lot of "wasted" seeds leading to a smaller yield of good ULF-English pairs. In our branching-based design, even if one sample ends up being incoherent, the other branches of the algorithm still remain viable.

3.3 ULF Macros

One notable limitation of our sampler is that it does not support most ULF macros. ULF macros perform unique transformations over their arguments to handle complex but regular mappings from syntax to semantic structure (e.g., topicalization, post-nominal modification, etc.) and do not fit directly into the type-compositional system.

4 Experiments

We evaluate our sampler on the hand-annotated ULF 1.0 dataset by Kim et al. (2021a), the only dataset of gold ULF annotations that we are aware of. This dataset has 1,378, 180, and 180 sentences of ULF-English pairs in the training, development, and test sets, respectively.

Metrics. Following prior work on this dataset, we use SEMBLEU as the primary evaluation metric and use EL-SMATCH secondarily for analysis, since it is broken down into F1, precision, and recall components. The SEMBLEU score better reflects the the parser's ability to generate coherent ULFs because it takes into account chains of multiple nodes and edges that EL-SMATCH does not.

d	b	M	M'	p	N
1	3	5	5	0.5	5,035
2	3	5	5	0.5	14,503
3	1	5	5	0.5	4,777
3	2	5	5	0.5	16,050
3	3	5	5	0.5	40,708
3	4	5	5	0.5	83,383
4	3	5	5	0.5	116,112

Table 1: Sampling parameters and the resulting dataset sizes. The table uses the same variable conventions as Section 3 for sampling parameters and dataset size.

Thus, SEMBLEU is used as the primary evaluation metric for ULF parsing. Kim and Schubert (2016) describes EL-SMATCH in detail, which includes a method for representing ULFs as a set of triples similar to AMRs. When SEMBLEU is run on ULF, the same set-of-triples representation is used for ULFs so that the metric designed for AMR can be run on ULF.

4.1 Settings

Model. In order to isolate the benefits of the data augmentation method, we use the current state-of-the-art ULF parsing model (Kim et al., 2021a). This parser is described in detail in Appendix A. While Kim et al. (2021a) released the code for their model, it runs on PyTorch 1.2 with Python 3.6 which are incompatible with the drivers in some of our more up-to-date computing machines. We updated the code to use PyTorch 1.11 and Python 3.10. This initially led to a reduction in parser performance, but we found that we could replicate the original parser performance by reducing the step size by a factor 0.25 and doubling the number of training epochs. We detail the replication experiment in Appendix D, including the model hyperparameters. We use the model that successfully replicated the original results in our experiments.

Sampled Datasets. The sampling parameter combinations we test are listed in Table 1 along with the number of unique examples that result from this sampling process. We vary the handler parameters: depth and branching factors, which largely determine the number of samples. We fix the subtree sampling parameters: maximum sample size to 5, maximum replacement size to 5, and recursion probability to 0.5. During the development process, we found this to lead to the best balance of quality and speed. We remove duplicated samples

d	b	SEMBLEU	EL-SMATCH		
			F1	Precision	Recall
Reported		47.4	**59.8**	60.7	**59.0**
Replicated		47.1	58.7	60.6	56.9
1	3	48.2	59.5	61.6	57.6
2	3	46.0	58.2	59.5	57.0
3	1	47.9	59.0	**61.8**	56.5
3	2	46.1	57.9	59.8	56.1
3	3	48.3	**59.8**	61.5	57.8
3	4	47.8	58.1	60.1	56.3
4	3	**49.0**	59.3	60.9	57.8

Table 2: Test set parser performance for augmented training on various sampling parameters and no filtering—the average of 5 runs with different random seeds.

to reduce unintended bias towards these sentences.

LM Filtering. To evaluate the trade-off between sample quality and quantity, we vary the number of LM-filtered samples in our final augmented datasets. For each sampled dataset, we retain the following proportions of samples: 0.1, 0.25, 0.5, and 1.0. We limit our filtering experiments to the three largest sampled sets. This ensures that sufficient samples remain even after aggressive filtering.

4.2 Training & Hyperparameters

We modify the training process of the baseline model to include some number of epochs where the model trains on both the manually annotated ULF examples and the type-sampled dataset. After that, the remaining epochs are trained using only the manually annotated ULF examples. Other than this new hyperparameter, the only hyperparameter that is changed from the original model is the total number of epochs. We reduce the number of total epochs trained since the model begins to overfit earlier when a larger sampled dataset is added.

We estimated the number of epochs at which the model begins to overfit with sample augmentation using $d = 3$ and $b = 3$ at 1.0, 0.5, 0.25, and 0.1 filtering proportions. For these parameters, we set the augmented training epochs to 1 greater than where we consistently saw overfitting.[7] We then generalize this to other experiments under the assumption that similarly sized datasets will begin to overfit at similar numbers of epochs. The training epoch specifics are provided in Appendix B.

4.3 Results

In this section, we report only the average test set metrics. Appendix B reports the full results including the development set metrics and standard deviations for both test and development sets.

Handler Parameters. We first compare the performance of the baseline model when augmented with the unfiltered samples from the sampler with sampling parameters specified in Table 1. These results are reported in Table 2. The model augmented with $d = 4$ and $b = 3$ has the best performance, with a SEMBLEU score that is 1.6 points over the reported baseline and 1.9 points over the replicated baseline. Augmenting the training with sampled pairs improves SEMBLEU scores for most sampler parameters. Under closer inspection, we find a curious pattern in these results. When we fix b to 3 and vary d from 1 to 4, we see a U-shaped SEMBLEU performance curve. Similarly, when we fix d to 3 and vary b from 1 to 4, we see a similar pattern, though the performance drops a bit again when $b = 4$.

The rise in SEMBLEU scores with data augmentation is not reflected as strongly in the EL-SMATCH scores. The EL-SMATCH F1 scores are typically slightly higher than the replicated baseline, but still under the score reported by Kim et al. (2021a). This suggests that the augmented samples push the model towards overall parse coherence without much changing the expected performance on a particular node or edge.[8]

LM Filtering. Table 3 shows the parser performance when the augmented dataset is filtered at different levels based on LM perplexity. Moderate filtering (0.5) tends to result in a small improvement, leading to the best SEMBLEU results in this paper of 49.1 on the $d = 3$, $b = 3$ dataset. Curiously, moderate filtering seems to push the model toward higher EL-SMATCH recall over precision.

Aggressive filtering (0.1) consistently degrades performance, even relative to the baseline model. This does not seem to be due to dataset size, since similarly sized augmented datasets in Table 2 ($d = 1, b = 3$ and $d = 3, b = 1$) still improve over the baseline. This suggests that aggressive LM filtering

[7] We consider an increase in development set perplexity to be an overfit model.

[8] EL-SMATCH scores are based on overlaps of individual nodes and edges whereas SEMBLEU scores are based on chains of node-edge-node links.

d	b	Filter	SEMBLEU	EL-SMATCH		
				F1	Precision	Recall
Reported			47.4	**59.8**	60.7	**59.0**
Replicated			47.1	58.7	60.6	56.9
3	3	1.00	48.3	**59.8**	**61.5**	57.8
		0.50	**49.1**	**59.8**	61.0	58.6
		0.25	46.6	58.3	59.3	57.4
		0.10	46.9	59.7	60.8	58.7
3	4	1.00	47.8	58.1	60.1	56.3
		0.50	47.9	59.0	60.4	57.5
		0.25	47.5	59.0	60.1	57.9
		0.10	46.6	58.4	59.8	56.4
4	3	1.00	49.0	59.3	60.9	57.8
		0.50	48.1	59.5	60.2	58.9
		0.25	48.1	59.0	60.0	58.1
		0.10	45.3	57.9	58.9	56.9

Table 3: Test set parser performance for LM-filtered augmented data for larger type sampling parameters—the average of 5 runs with different random seeds.

removes useful variance in the samples and leads to overfitting to low-perplexity sentences.

4.4 Qualitative Evaluation

We performed a qualitative analysis of the sampled sentences in an early version of the sampler[9] to evaluate the syntactic and semantic coherence of the generated samples. This experiment used $d = 3, b = 2$ sampling parameters and LM filtering to a dataset size of 5,000 samples. 400 randomly selected examples from this set were scored by human evaluators for both syntactic and semantic coherence, each on a 5-point scale. This resulted in a mean syntax score of 3.87 and a mean semantics score of 3.96, showing that the sampler typically succeeds in generating ULFs corresponding to well-formed and understandable text. Appendix C provides exact prompts given to human evaluators and more details of the results.

5 Related Work

Gibson and Lawley (2022) fine-tune GPT2-large on the ULF 1.0 dataset to learn both an English to ULF parser and a ULF to English generator. Their ULF parser underperforms Kim et al.'s (2021a) on the primary SEMBLEU metric but achieves the state-of-the-art on the EL-SMATCH metric. Their ULF to English generator matches or outperforms the `ulf2english` system on automatic machine translation metrics, BLEU (Papineni et al., 2002), chrF++ (Popović, 2017), and METEOR (Banerjee and Lavie, 2005) but uses more compute resources.

Data augmentation is far from a new idea for training neural networks. Data augmentation in computer vision is common via translation, rotation, cropping, flipping, noise injection, and color space transformations (Shorten and Khoshgoftaar, 2019). NLP has its own suite of data augmentation techniques that have been explored with token-level perturbations (Wei and Zou, 2019), graph-level perturbations (Şahin and Steedman, 2018), example interpolation (Zhang et al., 2018; Verma et al., 2019; Faramarzi et al., 2022), and distributional model-based synthetic sampling (Sennrich et al., 2016; Yang et al., 2020; Kobayashi, 2018) covering the major common approaches. Feng et al. (2021) provide a comprehensive survey of the NLP data augmentation approaches.

Focusing in on semantic parsing, Jia and Liang (2016) and Yu et al. (2021) learn synchronous context-free grammars using available data from which new examples are sampled. Andreas (2020) infers shared lexical environments and performs substitutions of words between them to encourage compositionality in semantic parsers. van Noord and Bos (2017) cross-reference two independent AMR parsers to automatically generate likely-high-quality examples which led to major parsing performance gains. None of these methods are able to exploit the knowledge that we have about ULF types and the rules that mediate their composition. Some of the approaches described in this section, such as van Noord and Bos' (2017), could be used in conjunction with our approach.

6 Conclusions

We presented a data augmentation method for ULFs that leverages ULF's underlying semantic type structure. This method helps alleviate the data sparsity problem that currently exists for ULF parsing, leading to a new state-of-the-art in this task without any change in the parsing model. Though we tested our data augmentation method on ULFs, this technique is applicable to any semantic parsing task with an underlying tree-structured compositional type system. For example, parsing in combinatory categorical grammar (Steedman, 2000) is another appropriate candidate for this sampling technique. Some details of the sampling procedure can also be improved in obvious, but not trivial ways. For example, our ULF atom sampling procedure uses word frequencies without ULF type

[9] This version failed to properly propagate certain syntactic restrictions leading to sampling failures, in which case we repeated the sampling process.

information.This leads to an over-representation of type-ambiguous words in our generated samples.

We think that type system-driven data augmentation for ULF is a promising way to further improve ULF parser performance. We expect further parsing improvements through refinement of the sampling parameters and expansion of the sampler to include macros. The additional data provided by such augmentation would support more general neural network-based semantic parsers as have been successful in other semantic representations (van Noord et al., 2018; Liu et al., 2018; Buys and Blunsom, 2017; Konstas et al., 2017). We are hopeful to see an improved semantic parser find utility in ULF-related tasks such as those mentioned at the end of section 1.

Acknowledgements

This work was supported in part by NSF grant IIS-1940981. We thank Omar Abdelrahman for assisting in the replication of the ULF parser performance for the newer Python version. We are grateful to the anonymous reviewers for their helpful feedback.

References

Jacob Andreas. 2020. Good-enough compositional data augmentation. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 7556–7566, Online. Association for Computational Linguistics.

Satanjeev Banerjee and Alon Lavie. 2005. METEOR: An automatic metric for MT evaluation with improved correlation with human judgments. In *Proceedings of the ACL Workshop on Intrinsic and Extrinsic Evaluation Measures for Machine Translation and/or Summarization*, pages 65–72, Ann Arbor, Michigan. Association for Computational Linguistics.

Jan Buys and Phil Blunsom. 2017. Robust incremental neural semantic graph parsing. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1215–1226, Vancouver, Canada. Association for Computational Linguistics.

Mojtaba Faramarzi, Mohammad Amini, Akilesh Badrinaaraayanan, Vikas Verma, and Sarath Chandar. 2022. Patchup: A feature-space block-level regularization technique for convolutional neural networks. *Proceedings of the AAAI Conference on Artificial Intelligence*, 36(1):589–597.

Steven Y. Feng, Varun Gangal, Jason Wei, Sarath Chandar, Soroush Vosoughi, Teruko Mitamura, and Eduard Hovy. 2021. A survey of data augmentation approaches for NLP. In *Findings of the Association for Computational Linguistics: ACL-IJCNLP 2021*, pages 968–988, Online. Association for Computational Linguistics.

Alex Franz and Thorsten Brants. 2006. All our n-gram are belong to you. https://ai.googleblog.com/2006/08/all-our-n-gram-are-belong-to-you.html. Google AI Blog.

Erin Gibson and Lane Lawley. 2022. Language-model-based parsing and english generation for unscoped episodic logical forms. *The International FLAIRS Conference Proceedings*, 35.

Daniel Gildea, Giorgio Satta, and Xiaochang Peng. 2018. Cache transition systems for graph parsing. *Computational Linguistics*, 44(1):85–118.

Robin Jia and Percy Liang. 2016. Data recombination for neural semantic parsing. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 12–22, Berlin, Germany. Association for Computational Linguistics.

Gene Kim, Viet Duong, Xin Lu, and Lenhart Schubert. 2021a. A transition-based parser for unscoped episodic logical forms. In *Proceedings of the 14th International Conference on Computational Semantics (IWCS)*, pages 184–201, Groningen, The Netherlands (online). Association for Computational Linguistics.

Gene Kim, Mandar Juvekar, Junis Ekmekciu, Viet Duong, and Lenhart Schubert. 2021b. A (mostly) symbolic system for monotonic inference with unscoped episodic logical forms. In *Proceedings of the 1st and 2nd Workshops on Natural Logic Meets Machine Learning (NALOMA)*, pages 71–80, Groningen, the Netherlands (online). Association for Computational Linguistics.

Gene Kim, Mandar Juvekar, and Lenhart Schubert. 2021c. Monotonic inference for underspecified episodic logic. In *Proceedings of the 1st and 2nd Workshops on Natural Logic Meets Machine Learning (NALOMA)*, pages 26–40, Groningen, the Netherlands (online). Association for Computational Linguistics.

Gene Kim, Benjamin Kane, Viet Duong, Muskaan Mendiratta, Graeme McGuire, Sophie Sackstein, Georgiy Platonov, and Lenhart Schubert. 2019. Generating discourse inferences from unscoped episodic logical formulas. In *Proceedings of the First International Workshop on Designing Meaning Representations*, pages 56–65, Florence, Italy. Association for Computational Linguistics.

Gene Kim and Lenhart Schubert. 2016. High-fidelity lexical axiom construction from verb glosses. In *Proceedings of the Fifth Joint Conference on Lexical and Computational Semantics*, pages 34–44, Berlin, Germany. Association for Computational Linguistics.

Gene Louis Kim. 2022. *Corpus annotation, parsing, and inference for Episodic Logic type structure*. Ph.D. thesis, University of Rochester.

Gene Louis Kim and Lenhart Schubert. 2019. A type-coherent, expressive representation as an initial step to language understanding. In *Proceedings of the 13th International Conference on Computational Semantics - Long Papers*, pages 13–30, Gothenburg, Sweden. Association for Computational Linguistics.

Sosuke Kobayashi. 2018. Contextual augmentation: Data augmentation by words with paradigmatic relations. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 2 (Short Papers)*, pages 452–457, New Orleans, Louisiana. Association for Computational Linguistics.

Ioannis Konstas, Srinivasan Iyer, Mark Yatskar, Yejin Choi, and Luke Zettlemoyer. 2017. Neural AMR: Sequence-to-sequence models for parsing and generation. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 146–157, Vancouver, Canada. Association for Computational Linguistics.

Lane Lawley, Gene Louis Kim, and Lenhart Schubert. 2019. Towards natural language story understanding with rich logical schemas. In *Proceedings of the Sixth Workshop on Natural Language and Computer Science*, pages 11–22, Gothenburg, Sweden. Association for Computational Linguistics.

Jiangming Liu, Shay B. Cohen, and Mirella Lapata. 2018. Discourse representation structure parsing. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 429–439, Melbourne, Australia. Association for Computational Linguistics.

Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. Roberta: A robustly optimized BERT pretraining approach. *CoRR*, abs/1907.11692.

Fabrizio Morbini and Lenhart Schubert. 2009. Evaluation of EPILOG: a reasoner for Episodic Logic. In *Proceedings of the Ninth International Symposium on Logical Formalizations of Commonsense Reasoning*, Toronto, Canada.

Rik van Noord, Lasha Abzianidze, Antonio Toral, and Johan Bos. 2018. Exploring neural methods for parsing discourse representation structures. *Transactions of the Association for Computational Linguistics*, 6:619–633.

Rik van Noord and Johan Bos. 2017. Neural semantic parsing by character-based translation: Experiments with abstract meaning representations. *Computational Linguistics in the Netherlands*, 7.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, pages 311–318, Philadelphia, Pennsylvania, USA. Association for Computational Linguistics.

Xiaochang Peng, Linfeng Song, Daniel Gildea, and Giorgio Satta. 2018. Sequence-to-sequence models for cache transition systems. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1842–1852, Melbourne, Australia. Association for Computational Linguistics.

Jeffrey Pennington, Richard Socher, and Christopher Manning. 2014. GloVe: Global vectors for word representation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543, Doha, Qatar. Association for Computational Linguistics.

Georgiy Platonov, Lenhart Schubert, Benjamin Kane, and Aaron Gindi. 2020. A spoken dialogue system for spatial question answering in a physical blocks world. In *Proceedings of the 21th Annual Meeting of the Special Interest Group on Discourse and Dialogue*, pages 128–131, 1st virtual meeting. Association for Computational Linguistics.

Maja Popović. 2017. chrF++: words helping character n-grams. In *Proceedings of the Second Conference on Machine Translation*, pages 612–618, Copenhagen, Denmark. Association for Computational Linguistics.

Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language models are unsupervised multitask learners. *OpenAI blog*.

Gözde Gül Şahin and Mark Steedman. 2018. Data augmentation via dependency tree morphing for low-resource languages. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 5004–5009, Brussels, Belgium. Association for Computational Linguistics.

Lenhart Schubert. 2014. From treebank parses to episodic logic and commonsense inference. In *Proceedings of the ACL 2014 Workshop on Semantic Parsing*, pages 55–60, Baltimore, MD. Association for Computational Linguistics.

Lenhart K. Schubert. 2000. The situations we talk about. In Jack Minker, editor, *Logic-based Artificial Intelligence*, pages 407–439. Kluwer Academic Publishers, Norwell, MA, USA.

Lenhart K. Schubert and Chung Hee Hwang. 2000. Episodic Logic meets Little Red Riding Hood: A comprehensive natural representation for language understanding. In Lucja M. Iwańska and Stuart C. Shapiro, editors, *Natural Language Processing and*

Knowledge Representation, pages 111–174. MIT Press, Cambridge, MA, USA.

Rico Sennrich, Barry Haddow, and Alexandra Birch. 2016. Improving neural machine translation models with monolingual data. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 86–96, Berlin, Germany. Association for Computational Linguistics.

Connor Shorten and Taghi M. Khoshgoftaar. 2019. A survey on image data augmentation for deep learning. *Journal of Big Data*, 6(60).

Mark Steedman. 2000. *The Syntactic Process*, volume 24. MIT press, Cambridge, MA.

Vikas Verma, Alex Lamb, Christopher Beckham, Amir Najafi, Ioannis Mitliagkas, David Lopez-Paz, and Yoshua Bengio. 2019. Manifold mixup: Better representations by interpolating hidden states. In *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 6438–6447. PMLR.

Jason Wei and Kai Zou. 2019. EDA: Easy data augmentation techniques for boosting performance on text classification tasks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 6382–6388, Hong Kong, China. Association for Computational Linguistics.

Yiben Yang, Chaitanya Malaviya, Jared Fernandez, Swabha Swayamdipta, Ronan Le Bras, Ji-Ping Wang, Chandra Bhagavatula, Yejin Choi, and Doug Downey. 2020. Generative data augmentation for commonsense reasoning. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1008–1025, Online. Association for Computational Linguistics.

Tao Yu, Chien-Sheng Wu, Xi Victoria Lin, Bailin Wang, Yi Chern Tan, Xinyi Yang, Dragomir R. Radev, Richard Socher, and Caiming Xiong. 2021. Grappa: Grammar-augmented pre-training for table semantic parsing. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net.

Hongyi Zhang, Moustapha Cissé, Yann N. Dauphin, and David Lopez-Paz. 2018. mixup: Beyond empirical risk minimization. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net.

A Baseline ULF Parser Description

Kim et al.'s (2021a) ULF parser is a transition system-based parser where the transition actions are selected using an LSTM. This parser modifies the cache transition parser (Gildea et al., 2018) to better model ULFs. At a high level, the modification introduces methods of generating ULF symbols on the fly from input words, rather than assuming a sequence of symbols as input. These symbol generation methods are further designed to reflect the relationship between ULF symbols and their corresponding words rather than assuming that an arbitrary mapping can exist between them. The cache transition system oracle, which is needed for training, is similarly modified to support these changes in the possible actions.

The LSTM is then trained to take a concatenation of the relevant input word, the relevant ULF symbol, and the current transition system state features as input and predicts the next action for the transition system. The transition system is inspected to determine which word is relevant, this is called hard attention (Peng et al., 2018). The relevant ULF symbol is similarly inferred from the transition system state and action history. Either we find which symbol we generated based on the current word, or if it has not been generated yet, the most recently generated symbol. The word features include the RoBERTa (Liu et al., 2019) embedding, GloVe embedding (Pennington et al., 2014), and learned embeddings of the lemma, POS tag, and NER tag. The symbol tokens are learned. The transition state features further includes information about the dependency tree distances of relevant words and the transition system phase. We refer you to Kim et al.'s (2021a) paper for further details of the parser.

B Additional Experiment Details

B.1 Augmented Epoch Determination

Filtering	N	overfit epoch
1.00	40,708	2
0.50	20,354	4
0.25	10,177	9

Table 4: Epochs values at which the model begins to overfit when trained with an augmented dataset using $d = 3$ and $b = 3$ parameters at various GPT-2 filtering levels.

d	b	N	F	Aug.	Total
1	3	5,035	1.00	25	45
2	3	14,503	1.00	10	30
3	1	4,777	1.00	25	45
3	2	16,050	1.00	10	30
3	3	40,708	1.00	2	20
3	3	20,354	0.50	5	25
3	3	10,177	0.25	10	30
3	3	5,083	0.10	25	45
3	4	83,383	1.00	2	20
3	4	41,691	0.50	2	20
3	4	20,845	0.25	5	25
3	4	10,422	0.10	10	30
4	3	116,112	1.00	2	20
4	3	58,056	0.50	2	20
4	3	29,028	0.25	5	25
4	3	14,514	0.10	10	30

Table 5: Number of epochs trained on the augmented dataset and in total for each sampling and filtering configuration. "Aug." is the number of augmented epochs. "Total" is the total number of epochs trained. Includes the total size of each sampling configuration results to help interpret the motivation behind the epoch values.

Table 4 shows when the model would begin to overfit at various augmented dataset levels. Specifically, we use the augmented dataset with $d = 3$ and $b = 3$, filtered with GPT-2 at various proportions. We use this to determine the number of epochs that we should train the model with sampling augmented data before only training on the manually annotated dataset. The procedure we use here is to add 1 to the results from Table 4. We do not add 1 to the full $d = 3$ and $b = 3$ dataset. Due to the size of the dataset, 1 additional epoch would likely severely overfit the model. For filtering at a 0.1 level, we extrapolate from the 0.5 and 0.25 levels, assuming a linear relationship between the number of augmenting examples and epochs.

We then generalize these results to other sampling settings under the assumption that similarly sized datasets will begin to overfit at similar numbers of epochs. We select the filtering level for the $d = 3$, $b = 3$ dataset whose N value is the closest lower value to the augmenting dataset in question. Table 5 lists the number of epochs that we trained each model on the augmented set and in total.

As with the rest of the parser details, we follow Kim et al.'s (2021a) approach to selecting the test model. After all training epochs, we select

Model		SEMBLEU		EL-SMATCH					
d	b			F1		Precision		Recall	
		Dev ($\pm\sigma$)	Test ($\pm\sigma$)	Dev ($\pm\sigma$)	Test ($\pm\sigma$)	Dev ($\pm\sigma$)	Test ($\pm\sigma$)	Dev ($\pm\sigma$)	Test ($\pm\sigma$)
Reported		46.4 ± 1.4	47.4 ± 1.3	58.4 ± 0.7	$\mathbf{59.8} \pm 1.0$	59.1 ± 1.1	60.7 ± 1.5	57.8 ± 0.5	$\mathbf{59.0} \pm 0.7$
Replicated		46.4 ± 2.7	47.1 ± 1.4	$56.9 \pm 1.9^{\dagger}$	58.7 ± 1.4	$59.5 \pm 1.8^{\dagger}$	60.6 ± 1.7	$54.6 \pm 2.2^{\dagger}$	56.9 ± 1.3
1	3	48.7 ± 1.6	48.2 ± 1.2	58.4 ± 0.7	59.5 ± 0.7	61.3 ± 1.5	61.6 ± 1.3	55.7 ± 0.9	57.6 ± 0.6
2	3	46.9 ± 1.9	46.0 ± 1.6	56.9 ± 1.7	58.2 ± 1.0	59.6 ± 1.6	59.5 ± 1.1	54.4 ± 1.9	57.0 ± 1.7
3	1	49.2 ± 1.1	47.9 ± 1.5	58.4 ± 1.0	59.0 ± 0.8	61.9 ± 0.6	$\mathbf{61.8} \pm 0.9$	55.3 ± 1.5	56.5 ± 1.2
3	2	48.0 ± 3.3	46.1 ± 3.3	57.5 ± 2.0	57.9 ± 1.7	60.7 ± 2.2	59.8 ± 1.8	54.7 ± 2.3	56.1 ± 2.4
3	3	49.6 ± 1.6	48.3 ± 1.7	59.2 ± 1.5	$\mathbf{59.8} \pm 1.4$	62.1 ± 1.7	61.5 ± 1.4	56.7 ± 1.6	57.8 ± 2.2
3	4	49.3 ± 3.8	47.8 ± 4.0	58.3 ± 2.2	58.1 ± 2.4	61.1 ± 1.8	60.1 ± 2.0	55.7 ± 2.6	56.3 ± 3.2
4	3	50.4 ± 0.8	$\mathbf{49.0} \pm 1.0$	58.7 ± 1.1	59.3 ± 1.7	61.2 ± 1.0	60.9 ± 1.2	56.5 ± 1.3	57.8 ± 2.5

Table 6: Detailed parser performance for augmented training on various sampling parameters and no filtering—the average & standard deviation of 5 runs. See the caption for Table 10 regarding the meaning of the † superscript.

Model			SEMBLEU		EL-SMATCH					
d	b	F			F1		Precision		Recall	
			Dev ($\pm\sigma$)	Test ($\pm\sigma$)	Dev ($\pm\sigma$)	Test ($\pm\sigma$)	Dev ($\pm\sigma$)	Test ($\pm\sigma$)	Dev ($\pm\sigma$)	Test ($\pm\sigma$)
Reported			46.4 ± 1.4	47.4 ± 1.3	58.4 ± 0.7	$\mathbf{59.8} \pm 1.0$	59.1 ± 1.1	60.7 ± 1.5	57.8 ± 0.5	$\mathbf{59.0} \pm 0.7$
Replicated			46.4 ± 2.7	47.1 ± 1.4	$56.9 \pm 1.9^{\dagger}$	58.7 ± 1.4	$59.5 \pm 1.8^{\dagger}$	60.6 ± 1.7	$54.6 \pm 2.2^{\dagger}$	56.9 ± 1.3
3	3	1.00	49.6 ± 1.6	48.3 ± 1.7	59.2 ± 1.5	$\mathbf{59.8} \pm 1.4$	62.1 ± 1.7	$\mathbf{61.5} \pm 1.4$	56.7 ± 1.6	57.8 ± 2.2
		0.50	49.5 ± 0.8	$\mathbf{49.1} \pm 1.7$	58.7 ± 0.5	$\mathbf{59.8} \pm 1.0$	61.2 ± 0.9	61.0 ± 1.5	56.4 ± 0.5	58.6 ± 0.8
		0.25	47.6 ± 1.8	46.6 ± 1.8	57.4 ± 1.3	58.3 ± 1.4	59.8 ± 1.3	59.3 ± 1.9	55.3 ± 1.4	57.4 ± 1.3
		0.10	47.2 ± 1.5	46.9 ± 1.5	57.9 ± 0.6	59.7 ± 0.8	59.7 ± 1.8	60.8 ± 1.5	56.2 ± 0.7	58.7 ± 0.9
3	4	1.00	49.3 ± 3.8	47.8 ± 4.0	58.3 ± 2.2	58.1 ± 2.4	61.1 ± 1.8	60.1 ± 2.0	55.7 ± 2.6	56.3 ± 3.2
		0.50	48.6 ± 1.1	47.9 ± 1.4	58.4 ± 1.0	59.0 ± 0.9	61.1 ± 0.8	60.4 ± 1.2	55.8 ± 1.4	57.5 ± 0.9
		0.25	47.3 ± 2.7	47.5 ± 2.4	57.4 ± 1.6	59.0 ± 2.2	59.8 ± 2.5	60.1 ± 2.7	55.1 ± 0.8	57.9 ± 1.9
		0.10	46.7 ± 2.4	46.6 ± 2.2	57.2 ± 2.0	58.4 ± 2.4	60.1 ± 1.8	59.8 ± 1.7	54.7 ± 2.4	56.4 ± 3.3
4	3	1.00	50.4 ± 0.8	49.0 ± 1.0	58.7 ± 1.1	59.3 ± 1.7	61.2 ± 1.0	60.9 ± 1.2	56.5 ± 1.3	57.8 ± 2.5
		0.50	49.0 ± 3.1	48.1 ± 3.6	59.2 ± 1.8	59.5 ± 2.1	60.9 ± 2.2	60.2 ± 2.4	57.7 ± 1.5	58.9 ± 1.9
		0.25	48.3 ± 1.3	48.1 ± 1.7	57.8 ± 0.8	59.0 ± 0.8	60.0 ± 0.9	60.0 ± 1.4	55.7 ± 0.8	58.1 ± 0.6
		0.10	45.5 ± 2.7	45.4 ± 2.7	56.9 ± 2.2	57.9 ± 2.1	58.9 ± 2.3	58.9 ± 1.8	55.1 ± 2.3	56.9 ± 1.8

Table 7: Detailed parser performance for LM-filtered augmented data for larger type sampling parameters—the average & standard deviation of 5 runs.

the epoch at which the model has the best development set SEMBLEU performance and restore that checkpoint for testing.

B.2 Detailed Parser Results

Table 6 shows the full detailed parsing results with full augmented datasets, no filtering. These results include the development set results and standard deviations. These details should help in checking replication. It also shows that adding the augmented data tends to lead to more overfitting of the model. That is, the development set performance is relatively higher compared to the test set performance when using data augmentation. Still, the average test set performance is only a point or two below the average development set performance so the overfitting does not tend to be very severe. The standard deviations also show that certain sampling configurations lead to very unstable training. $d = 3, b = 4$ for example has a 4-point standard deviation in SEMBLEU scores. Table 7 shows similarly detailed results for the filtering experiments.

B.3 Hyperparameters

Model hyperparameters are listed in Table 8. All of them except for the learning rate are grandfathered in from Kim et al.'s (2021a) parser.

C Qualitative Evaluation Details

For the qualitative analysis, we sampled an augmented dataset using the following parameters $d = 5, b = 2, M = 5, M' = 5, p = 0.5$. This earlier version of the parser performed filtering based on a maximum augmented size, including the seed examples, rather than filtering proportional to only the sampled set. We set the maximum size to 5,000. Excluding the 1,378 seed sentences (the training set of ULF 1.0), this results in 3,622 new samples. Of these, we uniformly randomly select 400 and had human evaluators rank the English translations (using `ulf2english`) for both syntactic and semantic coherence. Each example was

GloVe.840B.300d	
dim	300
RoBERTa embeddings	
source	RoBERTa-Base
dim	768
POS tag embeddings	
dim	100
Lemma embeddings	
dim	100
CharCNN	
num_filters	100
ngram_filter_sizes	[3]
Action embeddings	
dim	100
Transition system feature embeddings	
dim	25
Word encoder	
hidden_size	256
num_layers	3
Symbol encoder	
hidden_size	128
num_layers	2
Action decoder	
hidden_size	256
num_layers	2
MLP decoder	
hidden_size	256
activation_function	ReLU
num_layers	1
Optimizer	
type	ADAM
learning_rate	**0.0025**
max_grad_norm	5.0
dropout	0.33
num_epochs	25
Beam size	3
Vocabulary	
word vocab size	9200
symbol vocab size	7300
Batch size	32

Table 8: Model hyperparameters. The learning rate, which differs from Kim et al.'s (2021a) parser, is bolded.

ranked. The samples were distributed among three in-person human volunteers for ranking. Volunteers were given descriptions for the meaning of each score value. These are provided in Table 9.

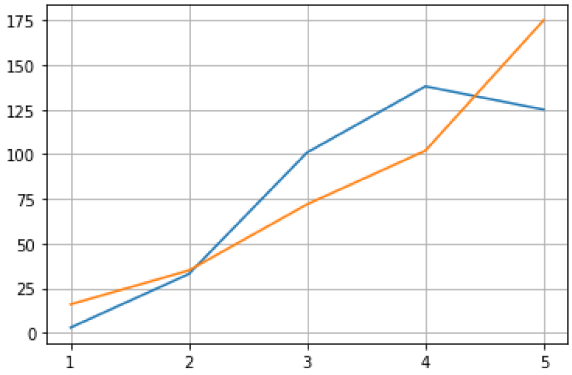


Figure 5: Frequencies for each score value reported by scorers. Score frequencies for syntax are in blue, those for meaning are in orange.

They were also asked to treat syntax and meaning as orthogonal properties as far as possible.

Figure 5 plots the frequencies for the qualitative scores. The mean syntax score was 3.87 with a standard deviation of 0.97. The mean meaning score was 3.96 with a standard deviation of 1.15. The medians for both scores were 4. About 65.7% of examples scored 4s and 5s on syntax, and about 69.2% scored 4s and 5s on meaning. Very few (less than 40 each) scored 1s and 2s on either categories. According to the descriptions given to the volunteers, this means that the average sentence was somewhere between "some syntactic inaccuracies, but overall not bad" (a score of 3) and "minor syntax errors" (a score of 4) leaning heavily towards the latter, and was just a little below "meaning is clear but a little strange for the average ear" (a score of 4) for meaning.

D Baseline Replication

The results for the baseline replication experiments are presented in Table 10. These results are based on 5 random runs, however, due to technical challenges, a few results are based on 4 random runs. This was the first experiment run for this paper so the infrastructure was still brittle. We did not redo these failed runs since a single additional run would not affect our conclusions in this circumstance.

When we run the unmodified parameters with our code updated to newer Python and PyTorch releases, we see that our SEMBLEU performance drops by 4.5 points. However, when we reduce the learning rate from 0.001 to 0.00025 and increase the total epochs from 25 to 60, the performance difference is only 0.3. Considering that the standard deviations of the SEMBLEU scores are 1.3 and

Score	Description
1	Completely garbled
2	Garbled up but there are sizable chunks that are coherent
3	Some inaccuracies in grammar, but overall not bad
4	Minor syntax errors
5	Grammatical

Score	Description
1	This doesn't mean anything
2	You could speculate what it means, but it isn't very coherent
3	Either somewhat clear but still unclear, or quite implausible
4	Meaning is clear but a little strange for the average ear
5	Makes sense, is plausible

Table 9: Descriptions of scores given to volunteers. The first table corresponds to syntax scores and the second corresponds to scores for meaning.

1.4 for the reported and our modified runs, respectively, 0.3 is within the range of sample variance. EL-SMATCH results are similar, though our replicated runs are relatively stronger on precision over recall.

E Pseudocode for Algorithms

Algorithms 1 to 3 are the pseudocode algorithms for the PICKRANDOMSUBTREE, SAMPLETYPEDERIVATION, and AUGMENTDATASET, respectively, which are described in Section 3.

Algorithm 2, however, elides some implementational caveats. First, in practice, we add a global parameter M' that imposes a maximum on the number of leaves in the sampled tree. This is implemented by limiting the amount of mutual recursion that happens between SAMPLETYPEDERIVATION and SAMPLEARGDERIVATIONS. Second, while the pseudocode uses simple equality to compare types, in practice we use a TYPEMATCH function which takes two types τ and τ' and returns true if and only if τ' is the same as τ, except possibly with *additional* syntactic restrictions. Third, in practice SAMPLETYPESOURCE can return some non-atomic ULF types which are known to be types of atomic ULFs when operated on with specific operators. This is to account for operators (such as sentential operators) which are ignored during type composition. An example of this is that SAMPLETYPESOURCE can returned a "tensed verb" type which can be instantiated in the next step to a tense operator operating on a verb (e.g. `(pres eat.v)`).

Model	SEMBLEU		EL-SMATCH					
			F1		Precision		Recall	
	Dev ($\pm\sigma$)	Test ($\pm\sigma$)	Dev ($\pm\sigma$)	Test ($\pm\sigma$)	Dev ($\pm\sigma$)	Test ($\pm\sigma$)	Dev ($\pm\sigma$)	Test ($\pm\sigma$)
Reported	46.4 ± 1.4	47.4 ± 1.3	58.4 ± 0.7	59.8 ± 1.0	59.1 ± 1.1	60.7 ± 1.5	57.8 ± 0.5	59.0 ± 0.7
Unmod.†	44.2 ± 1.7	42.9 ± 2.5	56.2 ± 0.4	56.7 ± 0.6	58.7 ± 1.9	58.3 ± 1.7	53.9 ± 1.2	55.3 ± 1.2
Modified	46.4 ± 2.7	47.1 ± 1.4	$56.9 \pm 1.9^\dagger$	58.7 ± 1.4	$59.5 \pm 1.8^\dagger$	60.6 ± 1.7	$54.6 \pm 2.2^\dagger$	56.9 ± 1.3

Table 10: Results for the baseline replication experiments. Results are based on 5 random runs. A "†" superscript indicates results based on 4 runs due to a system failure on one of the runs. The "Unmod." row contains the results of running our code updated to PyTorch 1.11 and Python 3.10 using the exact same parameters as the original. The "Modified" row contains the results where the learning rate is lowered four-fold and total epochs are increased from 25 to 60.

Algorithm 1 Picking a random subtree of a ULF.

global parameters: $M \in \mathbb{N}$, the maximum size of the subtree picked; $p \in (0, 1)$, a probability.
function PICKRANDOMSUBTREE(U)
  input: U, a ULF.
  if SIZE(U) $> M$ **then**
    $U' \leftarrow$ (uniformly) random child of U.
    return PICKRANDOMSUBTREE(U').
  else if U is atomic **then**
    return U.
  else
    $U' \leftarrow$ (uniformly) random child of U.
    return $\begin{cases} \text{PICKRANDOMSUBTREE}(U') \\ \qquad \text{with probability } p; \\ U \text{ otherwise.} \end{cases}$
  end if
end function

Algorithm 2 Sampling a type derivation tree for a given type. This pseudocode ignores some implementation details. Those details are explained in the text description of this algorithm.

function SAMPLETYPEDERIVATION(τ_{root})
  $\tau_{src} \leftarrow$ SAMPLETYPESOURCE(τ_{root})
  $\overrightarrow{\tau_{args}} \leftarrow$ SAMPLEARGDERIVATIONS(τ_{root}, τ_{src})
  return $(\tau_{src}, \overrightarrow{\tau_{args}})$
end function
function SAMPLETYPESOURCE(τ_0)
  global parameters: T, the set of possible types of ULF atoms; μ_T, a distribution over T.
  $T' \leftarrow \emptyset$
  for $\tau_a \in T$ **do**
    $\tau_{tmp} \leftarrow \tau_0$
    while $\tau_{tmp} \neq$ NIL **do**
      if $\tau_a = \tau_{tmp}$ **then**
        T'.append(τ_a)
      end if
      if $\tau_{tmp} \in \{\mathcal{D}, \mathcal{S}, \mathbf{2}\}$ **then**
        $\tau_{tmp} \leftarrow$ NIL
      else
        $\tau_{tmp} \leftarrow$ CODOMAIN(τ_{tmp})
      end if
    end while
  end for
  return $\tau_{src} \sim \mu_T(T')$
end function
function SAMPLEARGDERIVATIONS(τ_{cur}, τ_{src})
  if $\tau_{src} = \tau_{cur}$ **then**
    return []
  end if
  $\tau_{arg} \leftarrow$ NEXTARGTYPE(τ_{cur}, τ_{src})
  $D_{arg} \leftarrow$ SAMPLETYPEDERIVATION(τ_{arg})
  $\tau_{next} \leftarrow (\tau_{arg} \rightarrow \tau_{cur})$
  return $[D_{arg}] +$ SAMPLEARGDERIVATIONS(τ_{next}, τ_{src})
end function

Algorithm 3 The handler. We assume that the function SAMPLEFROMSEED is the top-level function for the sampler. It takes a ULF as input and runs the sampler to produce a single new (ULF, English) pair.

function AUGMENTDATASET($\mathscr{D}$, d, b, F)
  input: $\mathscr{D}$, a set of (ULF, English) pairs; d, the branching depth; b, the branching factor; F, top fraction of augmented set to keep.
  $\mathscr{D}' \leftarrow \mathscr{D}$
  for $(U, E) \in \mathscr{D}$ **do**
    $S \leftarrow [U]$
    for $i \in \{1, 2, \ldots, d\}$ **do**
      $S' \leftarrow \emptyset$
      for $U \in S$ **do**
        $U' \leftarrow$ POPFIRST(S)
        for $j \in \{1, 2, \ldots, b\}$ **do**
          $(U'', E'') \leftarrow$ SAMPLEFROMSEED(U')
          $\mathscr{D}'$.append((U'', E''))
          S'.append(U')
        end for
      end for
      $S \leftarrow S'$
    end for
  end for
  ORDERBYLANGUAGEMODELSCORE($\mathscr{D}'$)
  return first $F * |\mathscr{D}|$ elements of $\mathscr{D}'$
end function