A Transition-based Parser for Unscoped Episodic Logical Forms

Gene Louis Kim♡**, Viet Duong**♢**, Xin Lu**♠**, and Lenhart Schubert**♣
University of Rochester
Department of Computer Science
{gkim21♡, schubert♣}@cs.rochester.edu
{vduong♢, xlu32♠}@u.rochester.edu

Abstract

"Episodic Logic: Unscoped Logical Form" (EL-ULF) is a semantic representation capturing predicate-argument structure as well as more challenging aspects of language within the Episodic Logic formalism. We present the first learned approach for parsing sentences into ULFs, using a growing set of annotated examples. The results provide a strong baseline for future improvement. Our method learns a sequence-to-sequence model for predicting the transition action sequence within a modified cache transition system. We evaluate the efficacy of type grammar-based constraints, a word-to-symbol lexicon, and transition system state features in this task. Our system is available at https://github.com/genelkim/ulf-transition-parser. We also present the first official annotated ULF dataset at https://www.cs.rochester.edu/u/gkim21/ulf/resources/.

1 Introduction

EL-ULF was recently introduced as a semantic representation that accurately captures linguistic semantic structure within an expressive logical formalism, while staying close to the surface language, facilitating annotation of a dataset that can be used to train a parser (Kim and Schubert, 2019). The goal is to overcome the limitations of fragile rule-based systems, such as the Episodic Logic (EL) parser used for gloss axiomatization (Kim and Schubert, 2016) and domain-specific ULF parsers used for schema generation and dialogue systems (Lawley et al., 2019; Platonov et al., 2020). EL's rich model-theoretic semantics enables deductive inference, uncertain inference, and natural logic-like inference (Morbini and Schubert, 2009; Schubert and Hwang, 2000; Schubert, 2014); and the unscoped version, EL-ULF, supports Natural Logic-like monotonic inferences (Kim et al., 2020)

```
(i.pro ((pres want.v)
        (to (dance.v
             (adv-a (in.p (my.d ((mod-n new.a)
                                  (plur shoe.n)))))))))
```

Figure 1: An example ULF for the sentence, *"I want to dance in my new shoes"*.

and inferences based on some classes of entailments, presuppositions, and implicatures which are common in discourse (Kim et al., 2019). The lack of robust parsers have prevented large scale experiments using these powerful representations. We will refer to EL-ULF as simply ULF in the rest of this paper.

In this paper we present the first system that learns to parse ULFs of English sentences from an annotated dataset, and provide the first official release of the annotated ULF corpus, whereon our system is trained. We evaluate the parser using SEMBLEU (Song and Gildea, 2019) and a modified version of SMATCH (Cai and Knight, 2013), establishing a baseline for future work.

An initial effort in learning a parser producing a representation as rich as ULF is bound to face a data sparsity issue.[1] Thus a major goal in our choice of a transition-system-based parser has been to reduce the search space of the model. We investigate three additional methods of tackling this issue: (1) constraining actions in the decoding phase based on faithfulness to the ULF type system, (2) using a lexicon to limit the possible word-aligned symbols that the parser can generate, and (3) defining learnable features of the transition system state.

2 Unscoped Logical Form

Episodic Logic is an extension of first-order logic (FOL) that closely matches the form and ex-

[1] The training set in our initial release is only 1,378 sentences.

Proceedings of the 14th International Conference on Computational Semantics, pages 184–201
June 17–18, 2021. ©2021 Association for Computational Linguistics

pressivity of natural language, using reifying operators to enrich the domain of basic individuals and situations with propositions and kinds, keeping the logic first-order. It also uses other type-shifters, e.g., for mapping predicates to modifiers, and allows for generalized quantifiers (Schubert, 2000). ULF fully specifies the semantic type structure of EL by marking the types of the atoms and all of the predicate-argument relationships while leaving operator scope, anaphora, and word sense unresolved (Kim and Schubert, 2019). ULF is the critical first step to parsing full-fledged EL formulas. Types are marked on ULF atoms with a suffixed tag resembling the part-of-speech (e.g., .v, .n, .pro, .d for verbs, nouns, pronouns, and determiners, respectively). Names are instead marked with pipes (e.g. |John|) and a closed set of logical and macro operators have unique types and are left without a type marking. Each suffix denotes a set of possible semantic denotations, e.g. .pro always denotes an *entity* and .v denotes an *n-ary predicate* where *n* can vary. The symbol without the suffix or pipes is called the *stem*.

Type shifters in ULF maintain coherence of the semantic type compositions. For example, the type shifter adv-a maps a predicate into a verbal predicate modifier as in the prepositional phrase *"in my new shoes"* in Figure 1, as opposed to its predicative use *"A spider is in my new shoes"*.

The syntactic structure is closely reflected in ULF even under syntactic movement through the use of rewriting *macros* which explicitly mark these occurrences and upon expansion make the exact semantic argument structure available. Also, further resembling syntactic structure, ULFs are trees. The operators in operator-argument relations of ULF can be in first or second position, disambiguated by the types of the participating expressions. This further reduces the amount of word reordering between English and ULFs. The EL type system only allows function application for combining types, $\langle A, B\rangle, A \rightarrow B$, much like Montagovian semantics (Montague, 1970), but without type-raising.

3 Background

Currently, there is semantic parsing research occurring on multiple representational fronts, which is showcased by the cross-framework meaning representation parsing task (Oepen et al., 2019). The key differentiating factor of ULF from other meaning representations is the model-theoretic expressive capacity. To highlight this, here are a few limitations of notable representations: AMR (Banarescu et al., 2013a) neglects issues such as articles, tense, and nonintersective modification in favor of a canonicalized form that abstracts away from the surface structure; Minimal Recursion Semantics (Copestake et al., 2005) captures meta-level semantics for which inference systems cannot be built directly based on model-theoretic notions of truth and entailment; and extant semantic parsers for DRSs generate FOL-equivalent LFs, thus precludes proper treatment of phenomena such as generalized quantifiers, modification, and reification. Due to space limitations, we refer to Kim and Schubert (2019) for an in-depth description and motivation of ULF, including comparisons to other representations. We also refer to Schubert (2015) which places EL—the antecedent of ULF—in a broad context.

Our ULF parser development draws inspiration from the body of semantic parsing research on graph-based formalism of natural language, in particular, the recent advances in AMR parsing (Peng et al., 2018; Zhang et al., 2019a). The core organization of our parser is based on Peng et al. (2018), which uses a sequence-to-sequence model to predict the transition action sequence for a cache transition system with transition system features and hard attention alignment.

There are many transition-based parsers that were developed for parsing meaning representations (Zhang et al., 2016; Buys and Blunsom, 2017; Damonte et al., 2017; Hershcovich et al., 2017). These are mainly based on what's called an arc-eager parsing method, termed by Abney and Johnson (1991). Arc-eager parsing greedily adds edges between nodes before full constituents are formed, which keeps the partial graph as connected as possible during the parsing process (Nivre, 2004). They modify arc-eager parsing in various ways to generalize to the graph structures. Our transition system can be considered a modification of bottom-up arc-standard parsing due to restrictions on arc formation. While this leads to a longer action sequence for parsing, the parser's access to complete constituents allows promotion-based symbol generation for unary operators such as type shifters and standard bottom-up type analysis for constrained parsing.

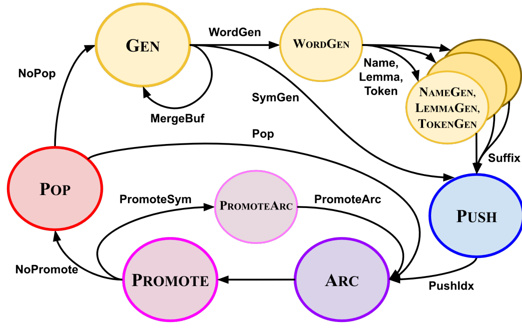

Figure 2: State transition diagram of the node generative transition system. Nodes in the figure are phases and edges are actions. An unlabeled edge means that this state transition occurs no matter what action is taken in that phase. The transition system starts in the GEN phase.

4 Our Transition System

Our transition system is a modification of the cache transition system (Gildea et al., 2018) which has been shown to be effective in AMR parsing (Peng et al., 2018). The distinctive aspect of our version is that the transition system generates nodes that are derived, but distinct, from the input sequence. We call it a node generative transition system. This eliminates the two-stage parsing framework of Peng et al. (2018). Our transition system also restricts the parses to be bottom-up to enable node generation and decoding constraints by the available constituents since ULF has an bottom-up compositional type system. The transition parser configuration is

$$C = (\sigma, \eta, \beta, G_p) \tag{1}$$

where σ is the stack, η is the cache, β is the buffer, and G_p is the partial graph. The parser is initialized with ([],[\$, . . . , \$],[$w_1, \ldots, w_n$],$\emptyset$), that is an empty stack, the cache with null values (\$), the buffer with the input sequence of words, where each word is a token, lemma, POS tuple, $w_i = (t_i, l_i, p_i)$, and an empty partial graph, $G_p = (V_p, E_p)$, where V_p is ordered. A vertex, $v_i = (s_i, a_i) \in V$, is a pair of a ULF symbol s_i, and its alignment a_i—the index of the word from which s_i was produced. We will refer to the leftmost element in β as w_{next}.

While the size of the cache is a hyperparameter that can be set for the cache transition parser, we restrict the cache size to 2 in order to keep the oracle simple despite the newly added actions. This means that only tree structures can be parsed. In describing the transition system, we differentiate between *phases* and *actions*. Phases are classes of states in the transition system and the actions move between states. Figure 2 shows the full state transition diagram and shows how the phases dictate which actions can be taken and how actions move between phases. Actions may take variables to specify how to move into the next phase. Phases also determine which features go into the determining the next action. We will write phases in small caps (e.g. GEN) and actions in bold (e.g. **TokenGen**) for clarity.

The GEN and PROMOTE phases are novel to our transition system. The GEN phase generates graph vertices that are transformations of the buffer values. This allows us to put words of the input sentence in β instead of a pre-computed ULF atom sequence. The PROMOTE phase enables context-sensitive symbol generation. It generates unaligned symbols in the context of an existing constituent in the partial graph. (Use of logical operators without word alignments only makes sense with respect to something for the operators to act on.) We now describe each of the actions in the transition system. The following are parser actions that were almost directly inherited from the vanilla cache transition parser.

- **PushIndex**(i) pushes (i, v) onto σ, where v is the vertex currently at index i of η. Then it moves the vertex generated by the prior GEN phase to index i in η.
- **Arc**(i, d, l) forms an arc with label l in direction d (i.e. left or right) between the vertex at index i of the cache and the rightmost vertex in the cache. The **NoArc** action is used if no arc is made.
- **Pop** pops (i, v) from σ where i is the index of η which v came from. v is placed at index i of η and shifts the appropriate elements to the right.

We introduce two sets, S_p and S_s, which define the vocabulary of the two unaligned symbol generation actions: **PromoteSym** and **SymGen**, respectively. S_p consists of logical and macro operators that do not align with English words. S_s consists of symbols that could not be aligned in the training set and are not members of S_p.

4.1 Promotion-based Symbol Generation

PROMOTE includes a subordinate PROMOTEARC phase for modularizing the parsing decision. The following parsing actions are in this phase.

- **PromoteSym**(s_p) generates a promotion symbol,

Stack	Cache	Buffer	Edges	Actions taken
[]	[$, $]	[Do, you, want, to, see, me, ?]	$\emptyset$	—
[$\$^0$]	[$, do.aux-s]	[you, want, to, see, me, ?]	$\emptyset$	**Lemma**(aux-s); **Push**(0)
[$\$^0$]	[$, pres]	[you, want, to, see, me, ?]	E_1	**NoArc**; **PSym**(pres); **PArc**(arg0)
[$\$^0$]	[$, χ_0]	[you, want, to, see, me, ?]	E_2	**NoArc**; **PSym**(χ_0); **PArc**(ι)
[$\$^0$, $\$^0$]	[χ_0, you.pro]	[want, to, see, me, ?]	E_2	**NoArc**; **NoP**; **Lemma**(pro); **Push**(0)
[$\$^0$]	[$, χ_0]	[want, to, see, me, ?]	E_3	**Arc**(0, R, arg0); **NoP**; **Pop**
[$\$^0$, $\$^0$]	[χ_0, want.v]	[to, see, me, ?]	E_3	**NoArc**; **NoP**; **Lemma**(v); **Push**(0)
[$\$^0$, $\$^0$, χ_0^0]	[want.v, to]	[see, me, ?]	E_3	**NoArc**; **NoP**; **Lemma**($\varnothing$); **Push**(0)
[$\$^0$, $\$^0$, χ_0^0, want.v^0]	[to, see.v]	[me, ?]	E_3	**NoArc**; **NoP**; **Lemma**(v); **Push**(0)
[$\$^0$, $\$^0$, χ_0^0, want.v^0, to^0]	[see.v, me.pro]	[?]	E_3	**NoArc**; **NoP**; **Token**(pro); **Push**(0)
[$\$^0$, $\$^0$, χ_0^0, want.v^0]	[to, see.v]	[?]	E_4	**Arc**(0, R, arg0); **NoP**; **Pop**
[$\$^0$, $\$^0$, χ_0^0]	[want.v, to]	[?]	E_5	**Arc**(0, R, arg0); **NoP**; **Pop**
[$\$^0$, $\$^0$]	[χ_0, want.v]	[?]	E_6	**Arc**(0, R, arg0); **NoP**; **Pop**
[$\$^0$]	[$, χ_0]	[?]	E_7	**Arc**(0, R, arg1); **NoP**; **Pop**
[$\$^0$]	[$, χ_1]	[?]	E_8	**NoArc**; **PSym**(χ_1); **PArc**(ι)
[$\$^0$, $\$^0$]	[χ_1, ?]	[]	E_8	**NoArc**; **NoP**; **Lemma**($\varnothing$); **Push**(0)
[$\$^0$]	[$, χ_1]	[]	E_9	**Arc**(0, R, arg0); **NoP**; **Pop**
[]	[$, $]	[]	E_9	**NoArc**; **NoP**; **Pop**

Figure 3: Example run of the transition system running on the sentence *"Do you want to see me?"* from our parser. The left four columns show the parser configuration after taking the actions shown in the rightmost column. We make the following modifications for brevity. When a **WordGen** action takes place, it is always followed by one of **Name**, **Lemma**, or **Token** and then a **Suffix**(e) action. Thus we omit the **WordGen** and **Suffix** actions and transfer the argument of **Suffix** to the **Name**, **Lemma**, or **Token** action. "Promote" is abbreviated as "P" (e.g., **PromoteSym** as **PSym**) and **PushIdx** as **Push**. Stack item indices (i, v) are written as v^i instead. χ and ι stand for COMPLEX and INSTANCE which are the special node and edge labels, respectively, for constructing non-atomic ULF operators in penman format. Edge labels arg0 and arg1 simply indicate the argument position in ULF. $E_n = \{e_i \mid 0 \leq i < n\}$ where $e_0 = (\text{do.aux-s} \xleftarrow{arg0} \text{pres})$, $e_1 = (\text{pres} \xleftarrow{\iota} \chi_0)$, $e_2 = (\chi_0 \xrightarrow{arg0} \text{you})$, $e_3 = (\text{see.v} \xrightarrow{arg0} \text{me.pro})$, $e_4 = (\text{to} \xrightarrow{arg0} \text{see.v})$, $e_5 = (\text{want.v} \xrightarrow{arg0} \text{to})$, $e_6 = (\chi_0 \xrightarrow{arg0} \text{want.v})$, $e_7 = (\chi_0 \xleftarrow{\iota} \chi_1)$, $e_8 = (\chi_1 \xrightarrow{arg0} ?)$.

$s_p \in S_p$, appends the vertex (s_p, NONE) to V_p, and proceeds to the PROMOTEARC phase.

- **NoPromote** skips the PROMOTE phase and proceeds to the POP phase.
- **PromoteArc**(l) makes an arc from the last added vertex, v_p, to the vertex at the rightmost position of the cache, v_{η_r}, by adding (v_p, v_{η_r}, l) to E_p. v_p then takes the place of v_{η_r} in the cache and v_{η_r} is no longer accessible by the transition system. The system proceeds to the ARC phase.

4.2 Sequential Symbol Generation

We replace the **Shift** action with the GEN phase to generate ULF atoms based on the tokenized text input. This phase allows the parser to generate a symbol using w_{next} as a foundation, or generate an arbitrary symbol that is not aligned to any word in β. GEN includes subordinate phases WORDGEN, LEMMAGEN, TOKENGEN, and NAMEGEN for modularizing the decision process.

- **WordGen** proceeds to WORDGEN phase, in which the following actions are available.
  1. **Name** proceeds to the NAMEGEN phase.
  2. **Lemma** proceeds to the LEMMAGEN phase.
  3. **Token** proceeds to the TOKENGEN phase.
- **Suffix**(e) is the only action available in the NAMEGEN, LEMMAGEN, and TOKENGEN phases. It generates a symbol s consisting of a stem and suffix extension e from w_{next}. In the NAMEGEN phase, the stem is t_{next} with surrounding pipes; in the TOKENGEN phase, the stem is t_{next}; and in the LEMMAGEN phase, the stem is l_{next}. (s, i) where i is the index of w_{next} is added to V_p and we move forward one word in β. The system proceeds to the PUSH phase.
- **SymGen**(s) adds an unaligned symbol (s, NONE) to V_p and proceeds to the PUSH phase.
- **SkipWord** skips word in β and returns to the GEN phase.
- **MergeBuf** takes w_{next} and merges it with the word after it $w_{\text{next}+1}$. This is stored at the front of the buffer as a pair $(v_\beta, v_{\beta+1})$. This forms a single stem with a space delimiter in the NAMEGEN phase and an underscore delimiter in the LEMMAGEN and TOKENGEN phases. The system returns to the GEN phase. This is used to handle multi-word expressions (e.g. *"had better"*).

The transition system begins in the GEN phase.

4.3 Oracle Extraction Algorithm

In order to train a model of the parser actions, we need to extract the desired action sequences from gold graphs. We modify the oracle extraction algorithm for the vanilla cache transition parser, described by Gildea et al. (2018). The oracle starts with a gold graph $G_g = (V_g, E_g)$ and maintains the partial graph $G_p = (V_p, E_p)$ of the parsing process, where V_g is sequenced by the preorder traversal of G_g. The oracle maintains s_{next}, the symbol in the foremost vertex of V_g that has not yet been added to G_p. The oracle begins with a transition system configuration, C, initialized with the input sequence, $w_1, ..., w_n$.

The oracle is also provided with an approximate alignment, $A = \{(w_i, v_j) \mid 1 \leq i \leq n, 1 \leq j \leq m\}$, between the input sequence, $w_{i:n}$, to the nodes in the gold graph, $V_g, |V_g| = m$, which is generated with a greedy matching algorithm. The matching algorithm uses a manually-tuned similarity heuristic built on the superficial similarity of English words, POS, and word order to the stems, suffixes, and preorder positions of the corresponding ULF atoms. A complete description of the alignment algorithm is in appendix B. This alignment is not necessary to maintain correctness of the oracle, but it is used to cut the losses when the input words become out of sync with the gold graph vertex order.[2] Steps 5-7 of the GEN phase uses the alignments to identify whether the buffer or the vertex order is ahead of the other and appropriately sync them back together.

The oracle uses the following procedure, broken down by parsing phase, to extract the action sequence to build the $G_p = G_g$ with C and A.

- GEN phase: Let $b = \text{Stem}(s_{\text{next}})$, $e = \text{Suffix}(s_{\text{next}})$, $n = \text{IsName}(s_{\text{next}})$.[3]
  1. If n and $t_{\text{next}} = b$, **NameGen**(e)
  2. If not n and $t_{\text{next}} =_i b$, **TokenGen**(e)
  3. If not n and $l_{\text{next}} =_i b$, **LemmaGen**(e)
  4. **MergeBuf** if
     n and $\text{Pre}(\text{Concat}(t_{\text{next}}, \text{`` ''}, t_{\text{next}+1}), b)$ or
     not n and $\text{Pre}_i(\text{Concat}(l_{\text{next}}, \text{``_''}, l_{\text{next}+1}), b)$
     or
     not n and $\text{Pre}_i(\text{Concat}(t_{\text{next}}, \text{``_''}, t_{\text{next}+1}), b)$
  5. If $(w_i, v_{\text{next}}) \in A$ for w_i before w_{next} or $v_{\text{next}} \in S_s$, then **SymGen**(v_{next})
  6. If $(w_{\text{next}}, v_j) \in A$ for v_j which comes after v_{next} or $v_j \in V_p$, then **SkipWord**.
  7. Otherwise, **SymGen**(v_{next})

  Step 5-7 allow the oracle to handle the generation of symbols that are not in word order, by skipping any words that come earlier than the symbol order; and generating symbols that cannot be aligned with **SymGen** for any reason.
- PUSH phase: The push phase of the vanilla cache transition parser's oracle—viz., choosing the cache position whose closest edge into β is farthest away—is extended to account not only for direct edges, but also for paths that include only unaligned-symbols.[4]
- ARC phase: The vanilla cache transition system's rule of generating the ARC action for any edge, $e \in E_g \wedge e \notin E_p$ between the rightmost cache position and the other positions, is extended to also require the child vertex to be fully formed. That is, for the vertex v_{child}, $|\text{descendants}(v_{\text{child}}, G_g)| = |\text{descendants}(v_{\text{child}}, G_p)|$. This enforces bottom-up parsing, which is necessary for both the promotion-based symbol generation and type composition constraint.
- PROMOTE phase: If the vertex in the rightmost cache position, v_{η_r}, is fully formed ($|\text{descendants}(v_{\eta_r}, G_g)| = |\text{descendants}(v_{\eta_r}, G_p)|$) and has a parent node in the PROMOTE lexicon ($\text{label}(\text{parent}(v_{\eta_r}, G_g)) \in S_p$), then the parser generates the action sequence **PromoteSym**($\text{parent}(v_r, G_g)$), **PromoteArc**(l_p) where l_p is the label for the edge from the parent of v_{η_r} to v_{η_r} in G_g ($\text{EdgeLabel}(\text{parent}(v_{\eta_r}, G_g), v_{\eta_r}, G_g)$).

5 Model

Our model has three basic components: (1) a word sequence encoder, (2) a ULF atom sequence encoder, and (3) an action decoder, all of which are

[2] When the words become out-of-sync with the gold graph the oracle must rely on **SymGen** to generate the graph nodes. Since **SymGen** requires selecting the correct value out the entire vocabulary of ULF atoms, it is much more difficult to predict correctly than **NameGen**, *TokenGen*, and *LemmaGen* which require only selecting the correct type tag.

[3] $=$ is string match, $=_i$ is case-insensitive string match, Pre determines whether its first argument is a prefix of the second and Pre_i is the case-insensitive counterpart.

[4] The motivation for this is that if only unaligned symbols exist in the path, the full path can be made without changing the relative status of any other node in the transition system. Let v_1 and v_2 be the end points of the path. With v_1 in the cache and the word aligned to v_2, $w_{v_2} = w_{\text{next}}$, **SymGen** and PROMOTE can generate all nodes in the path without interacting with the rest of the transition system.

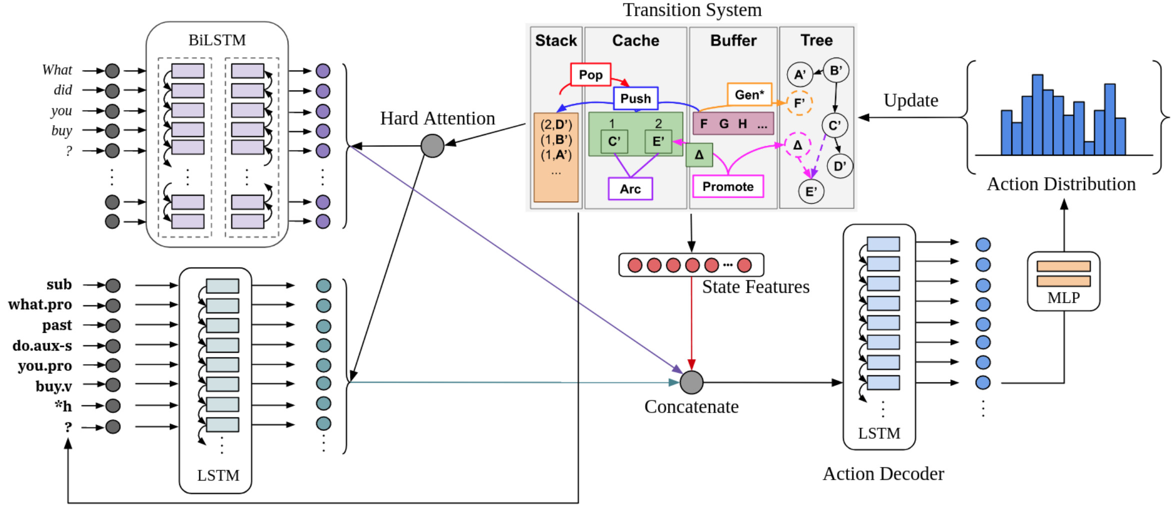

Figure 4: The model consists of a sentence-encoding BiLSTM, a symbol-encoding LSTM, and an action-decoding LSTM. New symbols generated in the GEN and PROMOTE phases of the transition system are appended to the symbol sequence. The transition system supplies hard attention pointers that select the relevant word and symbol embeddings. These are concatenated with the transition state feature vector and supplied as input to the action decoder, which predicts the next action that updates the transition system.

LSTMs. During decoding, the transition system configuration, C, is updated with decoded actions and used to organize the action decoder inputs using the sequence encoders. The system models the following probability

$$P(a_{1:q}|w_{1:n}) = \prod_{t=1}^{q} P(a_t|a_{1:t-1}, w_{1:n}; \theta) \quad (2)$$

where $a_{1:q}$ is the action sequence, $w_{1:n}$ is the input sequence, and θ is the set of model parameters. Figure 4 is a diagram of the full model structure.

5.1 Word and Symbol Sequence Encoders

The input word embedding sequence $w_{1:n}$ is encoded by a stacked bidirectional LSTM (Hochreiter and Schmidhuber, 1997) with L_w layers. Each word embedding sequence is a concatenation of embeddings of GloVe (Pennington et al., 2014), lemmas, part-of-speech (POS) and named entity (NER) tags, RoBERTa (Liu et al., 2019), and features learned by a character-level convolutional neural network (CharCNN, Kim et al., 2016). As ULF symbols are generated during the parsing process, the symbol embedding sequence $s_{1:m}$, which is the concatenation of a symbol-level learned embedding and the CharCNN feature vector over the symbol string, is encoded by a stacked *uni*directional LSTM of L_s layers.

5.2 Hard Attention

Peng et al. (2018) found that for AMR parsing with cache transition systems, a hard attention mechanism, tracking the next buffer node position and its aligned word, works better than a soft attention mechanism for selecting the embedding used during decoding. We take this idea and modify the tracking mechanism to find the most relevant word, w_i, and symbol, s_j, for each phase.

- ARC and PROMOTE*: The symbol s_j in the rightmost cache position and aligned word w_i.
- PUSH: The symbol s_j generated in the previous action and aligned word w_i.
- Otherwise: The last generated symbol s_j and the word w_i in the leftmost β position.

This selects the output sequences $\boldsymbol{h}_{w_i}^{L_w}$ and $\boldsymbol{h}_{s_j}^{L_s}$ from the encoders for the action decoder.

5.3 Transition State Features

Similar to Peng et al. (2018), we extract features from the current transition state configuration, C, to feed into the decoder as additional input in the form of learned embeddings

$$\boldsymbol{e}_f(C) = [\boldsymbol{e}_{f_1}(C); \boldsymbol{e}_{f_2}(C); ...; \boldsymbol{e}_{f_l}(C)] \quad (3)$$

where $\boldsymbol{e}_{f_k}(C)$ $(k = 1, ..., l)$ is the k-th feature embedding, with l total features. Our features, which are heavily inspired by Peng et al. (2018), are as follows.

- Phase: An indicator of the phase in the transition system.
- POP, GEN features: *Token features*[5] of the rightmost cache position and the leftmost buffer position; the number of rightward dependency edges from the cache position word and the first three of their labels; and the number of outgoing ULF arcs from the cache position and their first three labels.
- ARC, PROMOTE features: For the two cache positions, their token features and the word, symbol[6], and dependency distance between them; furthermore, their first three outgoing and single incoming dependency arc labels and their first two outgoing and single incoming ULF arc labels.
- PROMOTEARC features: Same as the PROMOTE features but for the rightmost cache position use the node/symbol generated in the preceding **PromoteSym** action.
- PUSH features: Token features for the leftmost buffer position and all cache positions.

5.4 Action Encoder/Decoder

The action sequence is encoded by a stacked unidirectional LSTM with L_a layers where the action input embeddings, $\boldsymbol{h}_{a_{1:q}}$ are concatenations of the word and symbol encodings.

$$\boldsymbol{h}_{a_k} = [\boldsymbol{h}_{w_i}^{L_w}; \boldsymbol{h}_{s_j}^{L_s}; \boldsymbol{e}_f(C)] \quad (4)$$

The state features $\boldsymbol{h}_{a_k}^{L_a}$ are then decoded into prediction weights with a linear transformation and ReLU non-linearity.

6 Parsing

The model is trained on the cross-entropy loss of the model probability (2) using the oracle action sequence. Both training and decoding are limited to a maximum action length of 800. For the training set the oracle has an average action length of 134 actions and a maximum action length of 1477.

6.1 Constrained Decoding

We investigate two methods of constraining the decoding process with prior knowledge of ULF to overcome the challenge of using a small dataset. These automatic methods filter out clearly incorrect choices at the cost of some decoding speed and further tailor the parser to ULFs.

ULF Lexicon To improve symbol generation, we introduce a lexicon with possible ULF atoms for each word. Nouns, verbs, adjectives, adverbs, and preposition entries are automatically converted from the Alvey lexicon (Carroll and Grover, 1989) with some manual editing. Pronouns, determiners, and conjunctions entries are extracted from Wiktionary[7] category lists. Auxiliary verbs entries are manually built from our ULF annotation guidelines. When generating a word-aligned symbol the stem is searched in the lexicon. If the string is present in the lexicon, only corresponding symbols in the lexicon are allowed to be generated. Since the lexicon is not completely comprehensive, this constraint may lead to some additional errors.

Type Composition The type system constraint adds a list of types, T_v, to accompany $|V_p|$ (the vertices of the partial graph), which stores the ULF type of each vertex. When a vertex, v, is added to G_p, its ULF type, t_v is added to T_v. This ULF type system is generalized with placeholders for macros and each stage in processing them. When the parser predicts an arc action during decoding, the types source, t_s, and target, t_t nodes are run through a type composition function. If the types can compose, $t_c = (t_s.t_t), t_c \neq \emptyset$, the type of the source node is replaced with t_c. Otherwise, the resulting C is not added to the search beam.

7 Experiments

We ran our experiments on a hand-annotated dataset of ULFs totaling 1,738 sentences (1,378 train, 180 dev, 180 test). The dataset is a mixture of sentences from crowd-sourced translations, news text, a question dataset, and novels. The distribution of sentences leans towards more questions, requests, clause-taking verbs, and counterfactuals because a portion of the dataset comes from the dataset used by Kim et al. (2019) for generating inferences from ULFs of those constructions.

The data is split by segmenting the dataset into 10 sentence segments and distributing them in a round-robin fashion, with the training set receiving eight chunks in each round. This splitting method is designed to allow document-level topics to distribute into splits while limiting any performance inflation of the dev and test results that can result

[5]The token features are the ULF symbol and the word, lemma, POS, and NER tags of the aligned index of the input.

[6]Symbol distance is based on the order in which the symbols are generated by the parser.

[7]https://en.wiktionary.org/

when localized word-choice and grammatical patterns are distributed into all splits.

Kim and Schubert (2019) found that interannotator agreement (IA) on ULFs using the EL-SMATCH metric (Kim and Schubert, 2016) is 0.79.[8] We add a second pass to further reduce variability in our annotations.[9] Further details about the dataset are available in appendix A and the complete annotation guidelines are available as part of the dataset.

ULF-AMR In order to use parsing and evaluation methods developed for AMR parsing (Banarescu et al., 2013a), we rewrite ULFs in penman format (Kasper, 1989) by introducing a node for each ULF atom and generating left-to-right arcs in the order that they appear (:ARG0, :ARG1, etc.), assuming the leftmost constituent is the parent. In order to handle non-atomic operators in penman format which only allows atomic nodes, we introduce a COMPLEX node with an :INSTANCE edge to mark the identity of the non-atomic operator.

Setup We evaluate the model with SEMBLEU (Song and Gildea, 2019), a metric for parsing accuracy of AMRs (Banarescu et al., 2013b). This metric extends BLEU (Papineni et al., 2002) to node- and edge-labeled graphs. We also measure EL-SMATCH, a generalization of SMATCH to graphs with non-atomic nodes, for analysis of the model since it has F1, precision, and recall components.

The tokens, lemmas, POS tags, NER tags, and dependencies are all extracted using the Stanford CoreNLP toolkit (Manning et al., 2014). In all experiments the model was trained for 25 epochs. Starting at the 12th epoch we measured the SEMBLEU performance on the dev split with beam size 3. Hyperparameters were tuned manually on the dev split performance of a smaller, preliminary version of the annotation corpus. We use RoBERTa-Base embeddings with frozen parameters, 300 dimensional GloVe embeddings, and 100 dimensional t_i, l_i, p_i, action, and symbol embeddings. The word encoder is 3 layers. The symbol encoder and action decoder are 2 layers. Experiments were run on a single NVIDIA Tesla K80 or GeForce RTX 2070 GPU. Training the full model takes about 6 hours. The full tables of results and default parameters are available in appendix D.

[8] cf. AMR is reported to have about 0.8 IA using the SMATCH metric (Tsialos, 2015)

[9] We did not measure IAA on our dataset and take the prior report as an lower-end estimate given the similarity of our annotations methods and our additional review phase. Our annotation process was collaborative and result in a single annotation per sentence so IAA cannot be measured.

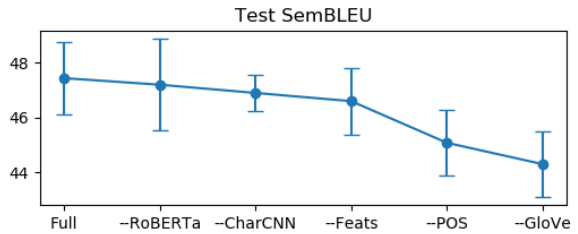

Figure 5: Ablation tests with standard deviation error bars of 5 runs of different random seeds.

7.1 Results

Ablations In our ablation tests, the model from the training epoch with the highest dev set SEMBLEU score is evaluated on the test split with beam size 3.[10] The results are shown in Figure 5.

CharCNN and RoBERTa are the least important components—to the point that we cannot conclude that they are of any benefit to the model due to the large overlap in the performance of models with and without them. The GloVe, POS, and feature embeddings are more important. The importance of POS is not surprising given the tight correspondence between POS tags and ULF type tags.

Model	SEMBLEU	EL-SMATCH
(Zhang et al., 2019a)	12.3	34.3
(Cai and Lam, 2020)	34.2	52.6
Our best model	**47.4**	**59.8**

Table 1: Comparison to AMR parsers.

Comparison to Baselines We compare our parser performance against two AMR parsers with minimal AMR-specific assumptions. The major recent efforts by the research community in AMR parsing make these parsers strong baselines. Specifically, we compare against the sequence-to-graph (STOG) parser (Zhang et al., 2019a) and Cai and Lam's (2020) graph-sequence iterative inference (GS) parser.[11] The ULF dataset is preprocessed for these parsers by stripping pipes from names to support the use of a copy mechanism and splitting node labels with spaces into multiple nodes to make the labels compatible with their data

[10] Our initial experiments re-evaluated the top-5 choices with a beam size of 10, but we found that the performance consistently degraded and abandoned this step.

[11] We do not compare our model against the existing rule-based ULF parsers since they are domain specific and cannot handle the range of sentences that appear in our dataset.

pipelines. Table 1 shows the results.[12] The STOG parser fares poorly on both metrics. A review of the results revealed that the parser struggles with node prediction in particular. This is likely the result of the dataset size not properly supporting the parser's latent alignment mechanism.[13] The GS parser performs better than the STOG parser by a large margin, but is still far from our parser's performance. The GS parser also struggles with node prediction, but is more successful in maintaining the correct edges in spite of incorrect node labels.

Investigating the dev set results reveals that our parser is quite successful in node generation, since by design the node generation process reflects the design of ULF atoms. Despite the theoretical capacity to generate node labels without a corresponding uttered word or phrase, our parser only does this for common logical operators such as reifiers and modifier constructors. The GS parser on the other hand, is relatively successful on node labels without uttered correspondences, correctly generating the elided "you" in imperatives and the logical operators ! and multi-sent which indicate imperatives and multi-sentence annotations, respectively. Our parser also manages to correctly generate a variety of verb phrase constructions, but fails to recognize reified infinitives as arguments of less frequent clausal verbs such as "neglect", "attach", etc. (as opposed to "have", "tell") and instead interprets "to" as either an argument-marking prepositions or reification of an already reified verb. Examples of parses and a discussion of specific errors are omitted here due to space constraints and provided in appendix E.

Constrained Decoding When evaluating decoding constraints, we select the model by re-running the five best performing epochs with constraints. When using the type composition constraint, we additionally increase the beam size to 10 so that the parser has backup options when its top choices are filtered out. Table 2 presents these results. We see a increase in precision for +Lex, but a greater loss in recall. +Type reduces performance on all metrics. Due to the bottom-up parsing procedure, a filtering of choices can cascade into fragmented

	SEMBLEU	EL-SMATCH		
		F1	Precision	Recall
Full	**47.4**	**59.8**	60.7	**59.0**
+Lex	46.2	57.5	**61.5**	54.1
+Type	40.0	55.8	59.1	52.8

Table 2: Statistics of model performances with constraints added—the average of 5 runs.

parses. The outputs for an arbitrarily selected run of the model has on average 2.9 fragments per sentence when decoding with the type constraint and 1.4 without. This and the relative performance on the precision metric suggest that constraints improve individual parsing choices, but are too strict, leading to fragmented parses.

Dependence on Length To investigate the performance dependence on the problem size, we partition the test set into quartiles by oracle action length. The 0 seed of our full model has SEMBLEU scores of 52, 47, 48, and 31 on the quartiles of increasing length. As expected, the parser performs better on shorter tasks. The parser performance is relatively stable until the last quartile. This is likely due to a long-tail of sentence lengths in our dataset. This last quartile includes sentences with oracle action length ranging from 148 to 1474.

8 Conclusion

We presented the first annotated ULF dataset and the first parser trained on such a dataset. We showed that our parser is a strong baseline, outperforming existing semantic parsers from a similar task. Surprisingly, our experiments showed that even in this low-resource setting, constrained decoding with a lexicon or a type system does more harm than good. However, the symbol generation method and features designed for ULFs result in a performance lead over using an AMR parser with minimal representational assumptions.

We hope that releasing this dataset will spur other efforts into improving ULF parsing. This of course includes expanding the dataset, using our comprehensive annotation guidelines and tools; but we see many additional avenues of improvement. The type grammar opens up many promising possibilities: sampling of silver data (in conjunction with ULF to English generation (Kim et al., 2019)), use as a weighted constraint, or direct incorporation into a model to avoid the pitfalls we observed in our simple approach to semantic type enforcement.

[12] Our parser gets the exact ULF for 6 out of the 180 sentences (3.3%). They were all yes-no questions which tend to be a bit shorter than informative declarative sentences (e.g. *"Can't you do something?"*).

[13] The STOG parser is improved by (Zhang et al., 2019b) with about 1 point of improvement on SMATCH. Unfortunately, the code for this parser is not released to the public.

9 Acknowledgments

This work was supported by NSF EAGER grant NSF IIS-1908595, DARPA CwC subcontract W911NF-15-1-0542, and a Sproull Graduate Fellowship from the University of Rochester. We are grateful to the anonymous reviewers for their helpful feedback.

References

Steven P Abney and Mark Johnson. 1991. Memory requirements and local ambiguities of parsing strategies. *Journal of Psycholinguistic Research*, 20(3):233–250.

Laura Banarescu, Claire Bonial, Shu Cai, Madalina Georgescu, Kira Griffitt, Ulf Hermjakob, Kevin Knight, Philipp Koehn, Martha Palmer, and Nathan Schneider. 2013a. Abstract meaning representation for sembanking. In *Proceedings of the 7th linguistic annotation workshop and interoperability with discourse*, pages 178–186.

Laura Banarescu, Claire Bonial, Shu Cai, Madalina Georgescu, Kira Griffitt, Ulf Hermjakob, Kevin Knight, Philipp Koehn, Martha Palmer, and Nathan Schneider. 2013b. Abstract Meaning Representation for sembanking. In *Proceedings of the 7th Linguistic Annotation Workshop and Interoperability with Discourse*, pages 178–186, Sofia, Bulgaria. 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.

Deng Cai and Wai Lam. 2020. AMR parsing via graph-sequence iterative inference. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 1290–1301, Online. Association for Computational Linguistics.

Shu Cai and Kevin Knight. 2013. Smatch: an evaluation metric for semantic feature structures. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 748–752, Sofia, Bulgaria. Association for Computational Linguistics.

J. Carroll and C. Grover. 1989. The derivation of a large computational lexicon of english from LDOCE. In Boguraev B. and Briscoe E., editors, *Computational Lexicography for Natural Language Processing*, pages 117–134. Longman, Harlow, UK.

Ann Copestake, Dan Flickinger, Carl Pollard, and Ivan A. Sag. 2005. Minimal Recursion Semantics: An introduction. *Research on Language and Computation*, 3(2):281–332.

Marco Damonte, Shay B. Cohen, and Giorgio Satta. 2017. An incremental parser for Abstract Meaning Representation. In *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics: Volume 1, Long Papers*, pages 536–546, Valencia, Spain. Association for Computational Linguistics.

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

Daniel Hershcovich, Omri Abend, and Ari Rappoport. 2017. A transition-based directed acyclic graph parser for UCCA. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1127–1138, Vancouver, Canada. Association for Computational Linguistics.

Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation*, 9(8):1735–1780.

Robert T. Kasper. 1989. A flexible interface for linking applications to Penman's sentence generator. In *Speech and Natural Language: Proceedings of a Workshop Held at Philadelphia, Pennsylvania, February 21-23, 1989*.

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, Mandar Juvekar, and Lenhart Schubert. 2020. Monotonic inference for underspecified episodic logic. In *Proceedings of the 1st Workshop on Natural Logic Meets Machine Learning (NALOMA)*. Association for Computational Linguistics.

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.

Yoon Kim, Yacine Jernite, David Sontag, and Alexander M Rush. 2016. Character-aware neural language models. In *Thirtieth AAAI Conference on Artificial Intelligence*.

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.

Xin Li and Dan Roth. 2002. Learning question classifiers. In *COLING 2002: The 19th International Conference on 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. *arXiv preprint arXiv:1907.11692*.

Christopher Manning, Mihai Surdeanu, John Bauer, Jenny Finkel, Steven Bethard, and David McClosky. 2014. The Stanford CoreNLP natural language processing toolkit. In *Proceedings of 52nd Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, pages 55–60, Baltimore, Maryland. Association for Computational Linguistics.

Richard Montague. 1970. Universal grammar. *Theoria*, 36(3):373–398.

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.

Joakim Nivre. 2004. Incrementality in deterministic dependency parsing. In *Proceedings of the workshop on incremental parsing: Bringing engineering and cognition together*, pages 50–57.

Stephan Oepen, Omri Abend, Jan Hajic, Daniel Hershcovich, Marco Kuhlmann, Tim O'Gorman, Nianwen Xue, Jayeol Chun, Milan Straka, and Zdenka Uresova. 2019. MRP 2019: Cross-framework meaning representation parsing. In *Proceedings of the Shared Task on Cross-Framework Meaning Representation Parsing at the 2019 Conference on Natural Language Learning*, pages 1–27, Hong Kong. Association for Computational Linguistics.

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.

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 Schubert. 2015. Semantic representation. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, AAAI'15, pages 4132–4138. AAAI Press.

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.

Linfeng Song and Daniel Gildea. 2019. SemBleu: A robust metric for AMR parsing evaluation. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 4547–4552, Florence, Italy. Association for Computational Linguistics.

Aristeidis Tsialos. 2015. Abstract meaning representation for sembanking. Available at `www.inf.ed.ac.uk/teaching/courses/tnlp/2014/Aristeidis.pdf`, accessed December 8, 2018.

Florian Wolf. 2005. *Coherence in natural language : data structures and applications*. Ph.D. thesis, Massachusetts Institute of Technology, Dept. of Brain and Cognitive Sciences.

Meishan Zhang, Yue Zhang, and Guohong Fu. 2016. Transition-based neural word segmentation. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 421–431, Berlin, Germany. Association for Computational Linguistics.

Sheng Zhang, Xutai Ma, Kevin Duh, and Benjamin Van Durme. 2019a. AMR parsing as sequence-to-graph transduction. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 80–94, Florence, Italy. Association for Computational Linguistics.

Sheng Zhang, Xutai Ma, Kevin Duh, and Benjamin Van Durme. 2019b. Broad-coverage semantic parsing as transduction. 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 3786–3798, Hong Kong, China. Association for Computational Linguistics.

A Dataset Details

We chose a variety of text sources for constructing this dataset to reduce genre-effects and provide good coverage of all the phenomena we are investigating. Some of these datasets include annotations, which we use only to identify sentence and token boundaries. The dataset includes 1,738 sentences, with a mean, median, min, and max sentence lengths of 10.275, 8, 2, and 128 words, respectively.

A.1 Data Sources

- **Tatoeba**

  The Tatoeba dataset[14] consists of crowd-sourced translations from a community-based educational platform. People can request the translation of a sentence from one language to another on the website and other members will provide the translation. Due to this pedagogical structure, the sentences are fluent, simple, and highly-varied. The English portion downloaded on May 18, 2017 contains 687,274 sentences.

- **Discourse Graphbank**

  The Discourse Graphbank (Wolf, 2005) is a discourse annotation corpus created from 135 newswire and WSJ texts. We use the discourse annotations to perform sentence delimiting. This dataset is on the order of several thousand sentences.

- **Project Gutenberg**

  Project Gutenberg[15] is an online repository of texts with expired copyright. We downloaded the top 100 most popular books from the 30 days prior to February 26, 2018. We then ignored books that have non-standard writing styles: poems, plays, archaic texts, instructional books, textbooks, and dictionaries. This collection totals to 578,650 sentences.

- **UIUC Question Classification**

  The UIUC Question Classification dataset (Li and Roth, 2002) consists of questions from the TREC question answering competition. It covers a wide range of question structures on a wide variety of topics, but focuses on factoid questions. This dataset consists of 15,452 questions.

[14] `https://tatoeba.org/eng/`

[15] `https://www.gutenberg.org`

Most of the dataset is annotated by random selection of a single or some contiguous sequence of sentences by annotators. However, part of the annotated dataset comes from inference experiments run by Kim et al. (2019) regarding questions, requests, counterfactuals, and clause-taking verbs. Therefore, the dataset has a bias towards having these phenomena at a higher frequency than expected from a random selection of English text.

A key issue regarding the dataset is its difficulty. We primarily quantify this with the AMR parser baseline, the sequence-to-graph (STOG) parser (Zhang et al., 2019a), in the main text, which performs quite poorly on this dataset. Its performance indicates that the patterns in this dataset are too varied for a modern parsing model to learn without built in ULF-specific biases. Although, part of this is due to the size of the dataset, if the dataset consisted only of short and highly-similar sentences, we would expect a modern neural model, such as the AMR baseline, to be able to learn successful parsing strategy for it.

This reflects the design of the dataset construction. Although the dataset indeed includes many short sentences, especially from the Tatoeba and UIUC Question Classification datasets, the sentences cover a wide range of styles and topics. The Tatoeba dataset is built from a crowd-sourced translation community, so the sentences are not limited in genre and style and has a bias toward sentences that give people trouble when learning a second language. We consider this to be valuable for a parsing dataset since, while the sentences from Tatoeba are usually short, they vary widely in topic and tend to focus on tricky phenomena that give language-learners—and likely parsers—trouble. Sentences from the Discourse Graphbank (news text) and Project Gutenberg (novels) further widen the scope of genres and styles in the dataset. This should make it difficult for a parsing model to overfit to dataset distribution. The dataset also has a considerable representation of longer sentences ($\sim$10% of the dataset is $>$20 words) including dozens of sentences exceeding 40 words, reaching up to 128 words.

A.2 Annotation Interface & Interannotator Agreement

We use the same annotation interface as Kim and Schubert (2019), which includes (1) syntax and bracket highlighting, (2) a sanity checker based on the underlying type grammar, and (3) uncertainty marking to trigger a review by a second annotator. The complete English-to-ULF annotation guideline is attached as a supplementary document.

Kim and Schubert (2019) reports interannotator agreement (IA) of ULF annotations using this annotation procedure. In summary, they found that agreement among sentences that are marked as *certain* are 0.79 on average and can be up-to 0.88 when we filter for well-trained annotators. For comparison, AMR annotations have been reported to have annotator vs consensus IA of 0.83 for newswire text and 0.79 for webtext using the *smatch* metric (Tsialos, 2015).

In order to mitigate the issue of low agreement of some annotators in the IA study, each annotation in our dataset was reviewed by a second annotator and corrected if necessary. There was an open discussion among annotators to clear up uncertainty and handle tricky cases during both the original annotation and the reviewing process so the actual dataset annotations are more consistent than the test of IA agreement (which had completely independent annotations) would suggest.

A.3 Dataset Splits

The data split is done by segmenting the dataset into 10 sentence segments and distributing them in a round-robin fashion, with the training set receiving eight chunks in each round. This splitting method is designed to allow document-level topics to distribute into splits while limiting any performance inflation of the dev and test results that can result when localized word-choice and grammatical patterns are distributed into all splits.

The Tatoeba dataset further exacerbates the issue of localized word-choice and grammatical patterns since multiple sentences using the same phrase or grammatical construction often appear back-to-back. We suspect that this is because the Tatoeba dataset is ordered chronologically and users often submit multiple similar sentences in order to help understand a particular phrase or grammatical pattern in a language that they are learning.

B Full ULF Alignment Details

The ULF-English alignment system takes into account the similarity of the English word to the ULF atom without the type extension, the similarity of the type extension with the POS tag, and the relative distance of the word and symbol in question.

Given a sentence $s = w_{1:n}$, which is tokenized, $t_{1:n}$, lemmatized, $l_{1:n}$, and POS tagged, $p_{1:n}$, a set of symbols that are never aligned S_u, and a list of ULF atoms $a_{1:m}$, which can be broken up into the base stems, $b_{1:m}$, and suffix extensions, $e_{1:m}$, in order of appearance in the formula (i.e. DFS preorder traversal), the word/atom similarity is defined using the following formulas.

$$\text{Sim}(w, a) = \max(\text{Olap}(t, b), \text{Olap}(l, b)) \\ + 0.5 * (\text{Olap}(p, e) + (1 - |\text{RL}(w, n) - \text{RL}(a, m)|))$$

where token overlap, Olap, is defined as

$$\text{Olap}(x, y) = \frac{2 * |\text{MaxSharedSubstr}(x, y)|}{|x| + |y|}$$

and relative location RL is defined as

$$\text{RL}(x, n) = \frac{\text{IndexOf}(x)}{n}$$

Next, in order of $\text{Sim}(w, a)$, we consider each word-atom pair, (w_i, a_i), $1 \leq i \leq n$ until $\text{Sim}(w, a) < \text{MinSim}$, where MinSim is set to 1.0, based on cursory results. We further disregard any alignments that include an atom which shouldn't be aligned (a_i s.t. $a_i \in S_u$). We assume that spans of words align to connected subgraphs, so we cannot accept all word-atom pairs. An word-atom pair, (w_i, a_i), is accepted into the set of token alignments, A_t, if and only if the following conditions are met:

1. w_i has no alignments or a_i is connected to an atom, a', that is already aligned to w_i.
2. a_i is not in any other alignment or w_i is adjacent to another, w' which is already aligned to a_i.

The token-level (word-atom) alignment, A_t, is then converted to connected (span-subgraph) alignment, A. This is done with the following algorithm:

1. For every atom a_i in one of the aligned pairs of A_t, merge all of the words aligned to a_i into a single span, s_i. During the initial alignment, we ensured that these words would form a span.
2. Merge all overlapping spans into single spans and collect the set of atoms that are aligned to each of these spans into a subgraph.[16] These collected subgraphs will be connected because we ensured that for any word the nodes that it is aligned to forms a connected subgraph.

[16]This can be done in $\mathcal{O}(n \log n)$ time by sorting the spans, then doing a single pass of merging overlapping elements.

C RoBERTa Handling Details

Except for RoBERTa, all other embeddings are fetched from their corresponding learned embedding lookup tables. RoBERTa uses OpenAI GPT-2 tokenizer for the input sequence and segments words into subwords prior to generating embeddings, which means one input word may correspond to multiple hidden states of RoBERTa. In order to accurately use these hidden states to represent each word, we apply an average pooling function to the outputs of RoBERTa according to the alignments between the original and GPT-2 tokenized sequences.

D Full Tables

Tables of the full set of raw results and parameters are presented in this section. Table 3 shows the ablations on the model without decoding constraints. This is the basis of Figure 5 in the main text. Table 4 shows the performance change with the lexicon constraint and Table 5 shows the performance change with the composition constraint. These tables are the basis of Table 2 in the main text. Our experiments with the lexicon constraint were more extensive since the type constraint takes considerably longer to run due to requiring a larger beam size and more computational overhead. Table 7 presents all of the model parameters in our experiments.

E Parse Examples

Figure 6 shows six parse examples of our parser and the GS parser in reference to the gold standard. Generally, we find that our parser does much better on node generation for nodes that correspond to an input word. For example, the GS parser on example 1 uses *(plur *s)* for the word "speech" and *iron.n* for the words "silver" and "silence". This isn't to say that our parser doesn't make mistakes. But the mistakes are not as open-ended. For example, our parser mistakenly annotates "silver" as a noun in example 1 when in fact it should be an adjective (compared against "golden"). The GS parser seems to pick the closest word in its vocabulary, which is generated from the training set and closed. This leads to strange annotations like *iron.n* for the word "silence". If there is nothing close available, then it can derail the entire parse. In example 4, the GS parser is unable to find a node label for the word "device" which derails the parse to generate *(mod-n*

Ablation	SEMBLEU		EL-SMATCH					
			F1		Precision		Recall	
	Dev	Test	Dev	Test	Dev	Test	Dev	Test
Full	46.4 ± 1.4	$\mathbf{47.4} \pm 1.3$	58.4 ± 0.7	$\mathbf{59.8} \pm 1.0$	59.1 ± 1.1	$\mathbf{60.7} \pm 1.5$	57.8 ± 0.5	$\mathbf{59.0} \pm 0.7$
-RoBERTa	45.5 ± 2.4	47.2 ± 1.7	58.3 ± 1.4	59.3 ± 1.0	59.1 ± 1.6	60.5 ± 1.1	57.5 ± 1.2	58.3 ± 0.9
-CharCNN	46.4 ± 1.0	46.9 ± 0.7	58.8 ± 0.8	59.3 ± 0.4	59.4 ± 1.3	60.1 ± 0.5	58.1 ± 0.6	58.5 ± 0.5
-$e_f(C)$ Feats	47.0 ± 1.2	46.6 ± 1.2	58.6 ± 0.5	58.8 ± 1.1	60.4 ± 1.2	60.2 ± 1.1	56.9 ± 0.4	57.4 ± 1.2
-POS	43.8 ± 1.7	45.1 ± 1.2	56.9 ± 1.1	58.3 ± 1.1	56.8 ± 1.0	58.7 ± 1.1	56.9 ± 1.2	57.9 ± 1.2
-GloVe	43.2 ± 1.8	44.3 ± 1.2	56.6 ± 1.0	57.1 ± 0.9	56.9 ± 2.7	58.3 ± 2.2	56.4 ± 1.7	56.1 ± 2.2

Table 3: Ablation results without decoding constraints, mean and standard deviation of 5 runs.

Ablation	SEMBLEU		EL-SMATCH					
			F1		Precision		Recall	
	Dev	Test	Dev	Test	Dev	Test	Dev	Test
Full	47.3 ± 0.6	46.2 ± 0.3	56.3 ± 0.7	57.5 ± 0.8	60.2 ± 0.5	61.5 ± 1.2	52.9 ± 0.9	54.1 ± 1.5
$\Delta\bar{x}$		-1.2		-2.3		+0.8		-4.9
-RoBERTa	46.6 ± 1.3	46.9 ± 0.6	56.1 ± 0.6	57.8 ± 0.4	60.0 ± 0.7	60.5 ± 0.9	52.6 ± 0.6	55.3 ± 0.5
-CharCNN	45.8 ± 2.3	45.5 ± 2.5	56.1 ± 1.4	56.9 ± 1.1	59.3 ± 2.4	59.6 ± 1.8	53.3 ± 1.1	54.5 ± 1.5
-$e_f(C)$ Feats	45.9 ± 1.5	45.6 ± 0.9	56.5 ± 0.6	57.0 ± 0.5	62.0 ± 0.8	61.4 ± 0.6	52.0 ± 1.1	53.3 ± 0.5
-POS	44.1 ± 2.0	44.5 ± 0.9	55.3 ± 0.2	56.6 ± 0.7	58.5 ± 2.2	60.4 ± 0.8	52.6 ± 2.3	53.2 ± 1.4
-GloVe	46.1 ± 1.1	45.4 ± 1.4	55.9 ± 0.9	57.0 ± 0.6	59.5 ± 1.5	60.3 ± 0.8	52.7 ± 1.0	54.0 ± 0.7

Table 4: Ablation results with the lexicon constraint, mean and standard deviation of 5 runs. $\Delta\bar{x}$ is the difference in the mean score between the test set results of the model with the lexicon constraint and without, i.e. Table 3. We only list this for the full model, but the pattern of higher precision but lower scores on other metrics generally holds for the other variants as well.

Ablation	SEMBLEU		EL-SMATCH					
			F1		Precision		Recall	
	Dev	Test	Dev	Test	Dev	Test	Dev	Test
Full	38.3 ± 2.3	40.0 ± 1.4	54.2 ± 1.2	55.8 ± 1.2	57.6 ± 1.0	59.1 ± 1.2	51.1 ± 1.5	52.8 ± 1.4
$\Delta\bar{x}$		-7.4		-4.0		-1.6		-6.2

Table 5: Ablation results with the type composition constraint, mean and standard deviation of 5 runs. $\Delta\bar{x}$ is the difference in the mean score between the test set results of the model with the type constraint and without, i.e. Table 3. We only ran the full model for this test because this constraint takes much longer to run.

Model	Fragments/Sentence	
	α	τ
Full	1.4	2.9
-CharCNN	1.1	3.5
-$e_f(C)$ Feats	1.4	3.9
-POS	1.5	3.2
$\bar{x}$	1.4	3.4

Table 6: Fragments per sentence on the test set decoding results for a subset of the ablated lexicon-constrained models (Table 4). α is the original model and τ is with the type composition constraint.

(mod-n man.n) (mod-n man.n iron.n) mod-n mod-n) for the text span "device is attached firmly to the ceiling".

This isn't to say that the GS parser always performs worse than our parser. When it comes to words that are elided (*{you}.pro* in example 4), nodes generated from multiple words (*had_better.aux-s* in example 3), or logical symbols unassociated with a particular word (*multi-sent* in example 6), the GS parser consistently performs better than our parser. Our parser has no special mechanism for these handling these cases and prefers to avoid generating node labels without an anchoring word.

A common mistake by our parser seems to be nested reifiers, which is not possible in the EL type system (e.g. *(to (ka come.v))* in example 5 and *(to (ka (show.v ..)))* in example 6). Other common mistakes that could be fixed by type coherence enforcement is mistakenly shifting a term into a modifier (e.g. *(adv-a (to ...))* in example 6). In the EL type system only predicates can be shifted into modifiers.

GloVe.840B.300d embeddings	
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	0.001
max_grad_norm	5.0
dropout	0.33
num_epochs	25
Beam size	
without type composition filtering	3
with type composition filtering	10
Vocabulary	
word_encoder_vocab_ size	9200
symbol_encoder_vocab_ size	7300
Batch size	32

Table 7: Default model parameters.

1. *"Speech is silver but silence is golden."*

   Gold: (((k speech.n) ((pres be.v) silver.a)) but.cc ((k silence.n) ((pres be.v) golden.a)))

   Ours: (((k speech.n) ((pres be.v) silver.n)) ▌ (k silence.n) ((pres be.v) golden.a))

   GS: (((k (plur *s)) ((pres be.v) (= (k iron.n)))) but.cc ((k iron.n) ((pres be.v) =)))

2. *"You neglected to tell me to buy bread."*

   Gold: (you.pro ((past neglect.v) (to (tell.v me.pro (to (buy.v (k bread.n)))))))

   Ours: (you.pro ((past neglect.v) (adv-e (to (tell.v me.pro (to (buy.v (k bread.n)))))))

   GS: (you.pro ((past fail.v) (to (tell.v me.pro {ref}.pro))))

3. *"You'd better knuckle down to work."*

   Gold: (you.pro ((pres had_better.aux-s) (knuckle.v down.adv-a (to work.v))))

   Ours: (you.pro ((pres would.aux-s) (knuckle.v down.a (adv-a (to.p work.v)))))

   GS: (you.pro ((pres had_better.aux-s) (go.v (to.p-arg (k work.n)) (adv-a (to.p (ka work.v))))))

4. *"Make sure that the device is attached firmly to the ceiling."*

   Gold: ({you}.pro ((pres make.v) sure.a
   (that ((the.d device.n)
   ((pres (pasv attach.v)) firmly.adv-a (to.p-arg (the.d ceiling.n)))))))

   Ours: (▌ ((pres make.v) sure.a that.pro (tht
   ((the.d device.n) ((pres be.v) ▌ (k (n+preds attach.v (to.p-arg ▌ ceiling.n))))))))

   GS: (({you}.pro ((pres make.v) (sure.a
   (that (the.d (mod-n (mod-n man.n) (mod-n man.n iron.n) mod-n mod-n)))))) !)

5. *"Can't I persuade you to come?"*

   Gold: (((pres can.aux-v) not i.pro (persuade.v you.pro (to come.v)) ?)

   Ours: (sub ((pres can.aux-v) not i.pro (persuade.v you.pro (to (ka come.v)) ?)))

   GS: (((pres can.aux-v) not i.pro (▌ come.v (to come.v) you.pro)) ?)

6. *"Look carefully. I'm going to show you how it's done."*

   Gold: (multi-sent (({you}.pro ((pres look.v) carefully.adv-a)) !)
   (i.pro ((pres be-going-to.aux-v)
   (show.v you.pro (ans-to (sub how.pq (it.pro ((pres (pasv do.v)) *h))))))))

   Ours: (▌((pres look.v) carefully.adv-a) ▌
   (tht (i.pro ((pres be.v) (go.v
   (adv-a (to (ka (show.v you.pro (sub how.pq (it.pro ((pres be.v) do.n ▌)))))))))))

   GS: (multi-sent (({you}.pro ((pres be.v) you.pro fine.a)) !)
   (i.pro ((pres be.aux-v) (go.v (to (do.v you.pro *h))))))

Figure 6: Several parse examples comparing behavior of our parser with the stronger baseline, the GS parser. For each example, the top is the gold parse, the center is our parser, and the bottom is the GS (Cai and Lam, 2020) parser. Errors are written in red. If something from the gold parse is omitted, a red highlighted block marks the location.