Universal Semantic Parsing

Siva Reddy†∗ Oscar Täckström‡ Slav Petrov‡ Mark Steedman†† Mirella Lapata††
†Stanford University
‡ Google Inc.
††University of Edinburgh
sivar@stanford.edu, {oscart, slav}@google.com, {steedman, mlap}@inf.ed.ac.uk

Abstract

Universal Dependencies (UD) offer a uniform cross-lingual syntactic representation, with the aim of advancing multilingual applications. Recent work shows that semantic parsing can be accomplished by transforming syntactic dependencies to logical forms. However, this work is limited to English, and cannot process dependency graphs, which allow handling complex phenomena such as control. In this work, we introduce UD\textsubscript{EP-LAMBDA}, a semantic interface for UD, which maps natural language to logical forms in an almost language-independent manner and can process dependency graphs. We perform experiments on question answering against Freebase and provide German and Spanish translations of the WebQuestions and GraphQuestions datasets to facilitate multilingual evaluation. Results show that UD\textsubscript{EP-LAMBDA} outperforms strong baselines across languages and datasets. For English, it achieves a 4.9 $F_1$ point improvement over the state-of-the-art on GraphQuestions.

1 Introduction

The Universal Dependencies (UD) initiative seeks to develop cross-linguistically consistent annotation guidelines as well as a large number of uniformly annotated treebanks for many languages (Nivre et al., 2016). Such resources could advance multilingual applications of parsing, improve comparability of evaluation results, enable cross-lingual learning, and more generally support natural language understanding.

Seeking to exploit the benefits of UD for natural language understanding, we introduce UD\textsubscript{EP-LAMBDA}, a semantic interface for UD that maps natural language to logical forms, representing underlying predicate-argument structures, in an almost language-independent manner. Our framework is based on DEPLAMBDA (Reddy et al., 2016) a recently developed method that converts English Stanford Dependencies (SD) to logical forms. The conversion process is illustrated in Figure 1 and discussed in more detail in Section 2. Whereas DEPLAMBDA works only for English, UD\textsubscript{EP-LAMBDA} applies to any language for which UD annotations are available.\footnote{As of v1.3, UD annotations are available for 47 languages at http://universaldependencies.org.} Moreover, DEPLAMBDA can only process tree-structured inputs whereas UD\textsubscript{EP-LAMBDA} can also process dependency graphs, which allow to handle complex constructions such as control. The different treatments of various linguistic constructions in UD compared to SD also require different handling in UD\textsubscript{EP-LAMBDA} (Section 3.3).

Our experiments focus on Freebase semantic parsing as a testbed for evaluating the framework’s multilingual appeal. We convert natural language to logical forms which in turn are converted to machine interpretable formal meaning representations for retrieving answers to questions from Freebase. To facilitate multilingual evaluation, we provide translations of the English WebQuestions (Berant et al., 2013) and GraphQuestions (Su et al., 2016) datasets to German and Spanish. We demonstrate that UD\textsubscript{EP-LAMBDA} can be used to derive logical forms for these languages using a minimal amount of language-specific knowledge. Aside from developing the first multilingual semantic parsing tool for Freebase, we also experimentally show that UD\textsubscript{EP-LAMBDA} outperforms strong baselines across...
languages and datasets. For English, it achieves the strongest result to date on GraphQuestions, with competitive results on WebQuestions. Our implementation and translated datasets are publicly available at https://github.com/sivareddyg/udeplambda.

2 DEPLambda

Before describing DEPLAMBDA, we provide an overview of DEPLAMBDA (Reddy et al., 2016) on which our approach is based. DEPLAMBDA converts a dependency tree to its logical form in three steps: binarization, substitution, and composition, each of which is briefly outlined below. Algorithm 1 describes the steps of DEPLAMBDA in lines 4-6, whereas lines 2 and 3 are specific to UDEPLAMBDA.

Binarization A dependency tree is first mapped to a Lisp-style s-expression indicating the order of semantic composition. Figure 1(b) shows the s-expression for the sentence Disney won an Oscar for the movie Frozen, derived from the dependency tree in Figure 1(a). Here, the sub-expression (dobj won (det Oscar an)) indicates that the logical form of the phrase won an Oscar is derived by composing the logical form of the label dobj with the logical form of the word won and the logical form of the phrase an Oscar, derived analogously. The s-expression can also be interpreted as a binarized tree with the dependency label as the root node, and the left and right expressions as subtrees.

A composition hierarchy is employed to impose a strict traversal ordering on the modifiers to each head in the dependency tree. As an example, won has three modifiers in Figure 1(a), which according to the composition hierarchy are composed in the order dobj > nmod > nsubj. In constructions like coordination, this ordering is crucial to arrive at the correct semantics. Lines 7-17 in Algorithm 1 describe the binarization step.

Substitution Each symbol in the s-expressions is substituted for a lambda expression encoding its semantics. Words and dependency labels are assigned different types of expressions. In general, words have expressions of the following kind:

ENTITY \( \Rightarrow \lambda x. \text{word}(x_a); \text{e.g. Oscar} \Rightarrow \lambda x. \text{Oscar}(x_a) \)
EVENT \( \Rightarrow \lambda x. \text{word}(x_e); \text{e.g. won} \Rightarrow \lambda x. \text{won}(x_e) \)
FUNCTIONAL \( \Rightarrow \lambda x. \text{TRUE}; \text{e.g. an} \Rightarrow \lambda x. \text{TRUE} \)

Here, the subscripts \( a \) and \( e \) denote the types of individuals (IND) and events (EVENT), respectively, whereas \( x \) denotes a paired variable \((x_a, x_e)\).

As an example of COPY, consider the lambda expression for dobj in (dobj won (det Oscar an)): \( \lambda f.gx.\exists y. f(x) \land g(y) \land \text{arg}_2(x_e, y_a). \) This expression takes two functions \( f \) and \( g \) as input, where \( f \) represents the logical form of won and \( g \) represents the logical form of an Oscar. The predicate-argument structure \( \text{arg}_2(x_e, y_a) \) indicates that the \( \text{arg}_2 \) of the event \( x_e \), i.e. \( \text{won} \), is the individual \( y_a \), i.e. the entity Oscar. Since \( \text{arg}_2(x_e, y_a) \) mimics the dependency structure dobj (won, Oscar), we refer to the expression kind evoked by dobj as COPY.
Expressions that invert the dependency direction are referred to as INVERT (e.g. amod in running horse); expressions that merge two subexpressions without introducing any relation predicates are referred to as MERGE (e.g. compound in movie Frozen); and expressions that simply return the parent expression semantics are referred to as HEAD (e.g. case in for Frozen). While this generalization applies to most dependency labels, several labels take a different logical form not listed here, some of which are discussed in Section 3.3. Sometimes the mapping of dependency label to lambda expression may depend on surrounding part-of-speech tags or dependency labels. For example, amod acts as INVERT when the modifier is a verb (e.g. in running horse), and as MERGE when the modifier is an adjective (e.g. in beautiful horse).\(^2\) Lines 26-32 in Algorithm 1 describe the substitution procedure.

**Composition** The final logical form is computed by beta-reduction, treating expressions of the form \(f(x, y)\) as the function \(f\) applied to the arguments \(x\) and \(y\). For example, \(\text{amod}\) \((\text{det Oscar an})\) results in \(\lambda x. \exists z. \text{won}(x, z) \land \text{Oscar}(z) \land \text{arg2}(x, z)\) when the expression for \(\text{amod}\) is applied to those for \(\text{won}\) and \((\text{det Oscar an})\). Figure 1(c) shows the logical form for the s-expression in Figure 1(b). The binarized s-expression is recursively converted to a logical form as described in lines 18-25 in Algorithm 1.

3 **UDEPLAMBDA**

We now introduce UDEPLAMBDA, a semantic interface for Universal Dependencies.\(^3\) Whereas DEPLAMBDA only applies to English Stanford Dependencies, UDEPLAMBDA takes advantage of the cross-lingual nature of UD to facilitate an (almost) language independent semantic interface. This is accomplished by restricting the binarization, substitution, and composition steps described above to rely solely on information encoded in the UD representation. As shown in Algorithm 1, lines 4-6 are common to both DEPLAMBDA and UDEPLAMBDA, whereas lines 2 and 3 applies only to UDEPLAMBDA. Importantly, UDEPLAMBDA is designed to not rely on lexical forms in a language

\(^2\)We use Tregex (Levy and Andrew, 2006) for substitution mappings and Cornell SPF (Artzi, 2013) as the lambda-calculus implementation. For example, in running horse, the tregex /label:amod/@target < /postag:verb/ matches amod to its INVERT expression \(\lambda x. \exists y. f(x) \land g(y) \land \text{amod}(y, x)\).

\(^3\)In what follows, all references to UD are to UD v1.3.

<table>
<thead>
<tr>
<th>Algorithm 1: UDEPLAMBDA Steps</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Function UDEPLAMBDA(depTree):</td>
</tr>
<tr>
<td>2 depGraph = Enhancement(depTree) #See Figure 2(a) for a depGraph.</td>
</tr>
<tr>
<td>3 bindedTree = SplitLongDistance(depGraph) #See Figure 2(b) for a bindedTree.</td>
</tr>
<tr>
<td>4 binarizedTree = Binarization(bindedTree) #See Figure 1(b) for a binarizedTree.</td>
</tr>
<tr>
<td>5 logicalForm = Composition(binarizedTree)</td>
</tr>
<tr>
<td>6 return logicalForm</td>
</tr>
<tr>
<td>7 Function Binarization(tree):</td>
</tr>
<tr>
<td>8 parent = GetRootNode(tree);</td>
</tr>
<tr>
<td>9 { (label1, child1), (label2, child2) ... }</td>
</tr>
<tr>
<td>10 GetChildNodes(parent)</td>
</tr>
<tr>
<td>11 sortedChildren = SortUsingLabelHierarchy</td>
</tr>
<tr>
<td>12 { ((label1, child1), (label2, child2)) ... }</td>
</tr>
<tr>
<td>13 binarizedTree.root = parent</td>
</tr>
<tr>
<td>14 for label, child \in sortedChildren:</td>
</tr>
<tr>
<td>15 temp.root = label</td>
</tr>
<tr>
<td>16 temp.left = binarizedTree</td>
</tr>
<tr>
<td>17 temp.right = Binarization(child)</td>
</tr>
<tr>
<td>18 binarizedTree = temp</td>
</tr>
<tr>
<td>19 return binarizedTree</td>
</tr>
<tr>
<td>18 Function Composition(binarizedTree):</td>
</tr>
<tr>
<td>19 mainLF = Substitution(binarizedTree.root)</td>
</tr>
<tr>
<td>20 if binarizedTree has left and right children:</td>
</tr>
<tr>
<td>21 leftLF = Composition(binarizedTree.left)</td>
</tr>
<tr>
<td>22 rightLF = Composition(binarizedTree.right)</td>
</tr>
<tr>
<td>23 mainLF = BetaReduce(mainLF, leftLF)</td>
</tr>
<tr>
<td>24 mainLF = BetaReduce(mainLF, rightLF)</td>
</tr>
<tr>
<td>25 return mainLF</td>
</tr>
<tr>
<td>26 Function Substitution(node):</td>
</tr>
<tr>
<td>27 logicalForms = [ ]</td>
</tr>
<tr>
<td>28 for tregexRule, template \in substitutionRules:</td>
</tr>
<tr>
<td>29 if tregexRule.match(node):</td>
</tr>
<tr>
<td>30 { if = GenLambdaExp(node, template)</td>
</tr>
<tr>
<td>31 logicalForms.add(if)</td>
</tr>
<tr>
<td>32 return logicalForms</td>
</tr>
</tbody>
</table>

We use Tregex (Levy and Andrew, 2006) for substitution mappings and Cornell SPF (Artzi, 2013) as the lambda-calculus implementation. For example, in running horse, the tregex /label:amod/@target < /postag:verb/ matches amod to its INVERT expression \(\lambda x. \exists y. f(x) \land g(y) \land \text{amod}(y, x)\).

In what follows, all references to UD are to UD v1.3.

We now introduce UDAMBDA, a semantic interface for Universal Dependencies. Whereas DEPLAMBDA only applies to English Stanford Dependencies, UDEPLAMBDA takes advantage of the cross-lingual nature of UD to facilitate an (almost) language independent semantic interface. This is accomplished by restricting the binarization, substitution, and composition steps described above to rely solely on information encoded in the UD representation. As shown in Algorithm 1, lines 4-6 are common to both DEPLAMBDA and UDEPLAMBDA, whereas lines 2 and 3 applies only to UDEPLAMBDA. Importantly, UDEPLAMBDA is designed to not rely on lexical forms in a language to assign lambda expressions, but only on information contained in dependency labels and postags.

However, some linguistic phenomena are language specific (e.g. pronoun-dropping) or lexicalized (e.g. every and the in English have different semantics, despite being both determiners) and are not encoded in the UD schema. Furthermore, some cross-linguistic phenomena, such as long-distance dependencies, are not part of the core UD representation. To circumvent this limitation, a simple enhancement step enriches the original UD representation before binarization takes place (Section 3.1). This step adds to the dependency tree missing syntactic information and long-distance dependencies, thereby creating a graph. Whereas DEPLAMBDA is not able to handle graph-structured input, UDEPL-

In what follows, all references to UD are to UD v1.3.
LAMBDA is designed to work with dependency graphs as well (Section 3.2). Finally, several constructions differ in structure between UD and SD, which requires different handling in the semantic interface (Section 3.3).

3.1 Enhancement

Both Schuster and Manning (2016) and Nivre et al. (2016) note the necessity of an enhanced UD representation to enable semantic applications. However, such enhancements are currently only available for a subset of languages in UD. Instead, we rely on a small number of enhancements for our main application—semantic parsing for question-answering—with the hope that this step can be replaced by an enhanced UD representation in the future. Specifically, we define three kinds of enhancements: (1) long-distance dependencies; (2) types of coordination; and (3) refined question word tags. These correspond to line 2 in Algorithm 1.

First, we identify long-distance dependencies in relative clauses and control constructions. We follow Schuster and Manning (2016) and find these using the labels acl (relative) and xcomp (control). Figure 2(a) shows the long-distance dependency in the sentence Anna wants to marry Kristoff. Here, marry is provided with its missing nsubj (dashed arc). Second, UD conflates all coordinating constructions to a single dependency label, conj. To obtain the correct coordination scope, we refine conj to conj:verb, conj:vp, conj:np, and conj:adj, similar to Reddy et al. (2016). Finally, unlike the PTB tags (Marcus et al., 1993) used by SD, the UD part-of-speech tags do not distinguish question words. Since these are crucial to question-answering, we use a small lexicon to refine the tags for determiners (DET), adverbs (ADV) and pronouns (PRON) to DET:WH, ADV:WH and PRON:WH, respectively. Specifically, we use a list of 12 (English), 14 (Spanish) and 35 (German) words, respectively. This is the only part of UDEPLAMBDA that relies on language-specific information. We hope that, as the coverage of morphological features in UD improves, this refinement can be replaced by relying on morphological features, such as the interrogative feature (INT).

3.2 Graph Structures and BIND

To handle graph structures that may result from the enhancement step, such as those in Figure 2(a), we propose a variable-binding mechanism that differs from that of DEPLAMBDA. This is indicated in line 3 of DEPLAMBDA. First, each long-distance dependency is split into independent arcs as shown in Figure 2(b). Here, Ω is a placeholder for the subject of marry, which in turn corresponds to Anna as indicated by the binding of Ω via the pseudo-label BIND. We treat BIND like an ordinary dependency label with semantics MERGE and process the resulting tree as usual, via the s-expression:

\[(\text{nsubj} (\text{xcomp} \text{wants} (\text{nsubj} (\text{mark} (\text{dobj} \text{Kristoff})) \Omega)) \text{BIND} \text{Anna} \Omega)\],

with the lambda-expression substitutions:

\[\text{wants}, \text{marry} \in \text{EVENT}; \text{to} \in \text{FUNCTIONAL}; \text{Anna}, \text{Kristoff} \in \text{ENTITY}; \text{mark} \in \text{HEAD}; \text{BIND} \in \text{MERGE}; \text{xcomp} = \lambda f. g. \exists y. (f(x) \land g(y) \land \text{xcomp}(y, y))\].

These substitutions are based solely on unlexicalized context. For example, the part-of-speech tag PROP of Anna invokes an ENTITY expression.

The placeholder Ω has semantics \(\lambda x. \text{EQ}(x, \omega)\), where \(\text{EQ}(u, \omega)\) is true iff \(u\) and \(\omega\) are equal (have the same denotation), which unifies the subject variable of wants with the subject variable of marry.

After substitution and composition, we get:

\[\lambda z. \exists y w v. \text{wants}(z_r) \land \text{Anna}(x_r) \land \text{arg1}(z_e, x_e) \land \text{EQ}(x, \omega) \land \text{marry}(y_v) \land \text{xcomp}(y_e, y_e) \land \text{arg1}(y_e, y_e) \land \text{EQ}(v, \omega) \land \text{Kristoff}(w_u) \land \text{arg2}(y_v, w_u)\].

This expression may be simplified further by replacing all occurrences of \(v\) with \(x\) and removing the unification predicates EQ, which results in:

\[\lambda z. \exists y w v. \text{wants}(z_r) \land \text{Anna}(x_r) \land \text{arg1}(z_e, x_e) \land \text{marry}(y_v) \land \text{xcomp}(y_e, y_e) \land \text{arg1}(y_e, x_e) \land \text{Kristoff}(w_u) \land \text{arg2}(y_v, w_u)\].

Figure 2: The original and enhanced dependency trees for Anna wants to marry Kristoff.
This expression encodes the fact that Anna is the arg₁ of the marry event, as desired. DEPLAMBDA, in contrast, cannot handle graph-structured input, since it lacks a principled way of generating s-expressions from graphs. Even given the above s-expression, BIND in DEPLAMBDA is defined in a way such that the composition fails to unify v and x, which is crucial for the correct semantics. Moreover, the definition of BIND in DEPLAMBDA does not have a formal interpretation within the lambda calculus, unlike ours.

3.3 Linguistic Constructions

Below, we highlight the most pertinent differences between UDEPLAMBDA and DEPLAMBDA, stemming from the different treatment of various linguistic constructions in UD versus SD.

Prepositional Phrases UD uses a content-head analysis, in contrast to SD, which treats function words as heads of prepositional phrases. Accordingly, the s-expression for the phrase president in 2009 is (nmod president (case 2009 in)) in UD-LAMBDA and (prep president (pobj in 2009)) in DEPLAMBDA. To achieve the desired semantics, λx. ∃y. president(x₁) ∧ president_event(x₁) ∧ arg₁(x₁, y₁) ∧ 2009(y₁) ∧ prep.in(x₁, y₁).

DEPLAMBDA relies on an intermediate logical form that requires some post-processing, whereas UDEPLAMBDA obtains the desired logical form directly through the following entries:

in ∈ FUNCTIONAL; 2009 ∈ ENTITY; case ∈ HEAD; president = λx. president(x₁) ∧ president_event(x₁) ∧ arg₁(x₁, y₁); nmmod = λf.gx. ∃y. f(x) ∧ g(y) ∧ nmmod.in(x₁, y₁).

Other nmmod constructions, such as possessives (nmmod:poss), temporal modifiers (nmmod:tmod) and adverbial modifiers (nmmod:nmod), are handled similarly. Note how the common noun president, evokes both entity and event predicates above.

Passives DEPLAMBDA gives special treatment to passive verbs, identified by the fine-grained part-of-speech tags in the PTB tag together with dependency context. For example, An Oscar was won is analyzed as λx. won.pass(x₁) ∧ Oscar(y₁) ∧ arg₁(x₁, y₁), where won.pass represents a passive event. However, UD does not distinguish between active and passive forms. While the labels nsubjpass or auxpass indicate passive constructions, such clues are sometimes missing, such as in reduced relatives. We therefore opt to not have separate entries for passives, but aim to produce identical logical forms for active and passive forms when possible (for example, by treating nsubjpass as direct object). With the following entries, won ∈ EVENT; an, was ∈ FUNCTIONAL; auxpass ∈ HEAD; nsubjpass = λf gx. ∃y. f(x) ∧ g(y) ∧ arg₂(x₁, y₂).

the lambda expression for An Oscar was won becomes λx. won(x₁) ∧ Oscar(y₁) ∧ arg₂(x₁, y₁), identical to that of its active form. However, not having a special entry for passive verbs may have undesirable side-effects. For example, in the reduced-relative construction Pixar claimed the Oscar won for Frozen, the phrase the Oscar won ... will receive the semantics λx. Oscar(y₁) ∧ won(x₁) ∧ arg₁(x₁, y₁), which differs from that of an Oscar was won. We leave it to the target application to disambiguate the interpretation in such cases.

Long-Distance Dependencies As discussed in Section 3.2, we handle long-distance dependencies evoked by clausal modifiers (acl) and control verbs (xcomp) with the BIND mechanism, whereas DEPLAMBDA cannot handle control constructions. For xcomp, as seen earlier, we use the mapping λf gx. ∃y. f(x) ∧ g(y) ∧ xcomp(x₁, y₁). For acl we use λf gx. ∃y. f(x) ∧ g(y), to conjoin the main clause and the modifier clause. However, not all acl clauses evoke long-distance dependencies, e.g. in the news that Disney won an Oscar, the clause that Disney won an Oscar is a subordinating conjunction of news. In such cases, we instead assign acl the INVERT semantics.

Questions Question words are marked with the enhanced part-of-speech tags DET:WH, ADV:WH and PRON:WH, which are all assigned the semantics λx. $\{\text{word}\}(x₁) \land TARGET(x₁)$. The predicate TARGET indicates that x₁ represents the variable of interest, that is the answer to the question.

3.4 Limitations In order to achieve language independence, UDEPLAMBDA has to sacrifice semantic specificity, since in many cases the semantics is carried by lexical information. Consider the sentences John broke the window and The window broke. Although it is the window that broke in both cases, our inferred logical forms do not canonicalize the relation between broke and window. To achieve this, we

would have to make the substitution of \texttt{nsbj} depending on lexical context, such that when \texttt{window} occurs as \texttt{nsbj} with \texttt{broke}, the predicate \texttt{arg2} is invoked rather than \texttt{arg1}. UDEPLAMBDA does not address this problem, and leave it to the target application to infer context-sensitive semantics of \texttt{arg1} and \texttt{arg2}. To measure the impact of this limitation, we present UDEPLAMBDA SRL in Section 4.4 which addresses this problem by relying on semantic roles from semantic role labeling (Palmer et al., 2010).

Other constructions that require lexical information are quantifiers like every, some and most, negation markers like no and not, and intentional verbs like believe and said. UD does not have special labels to indicate these. We discuss how to handle quantifiers in this framework in the supplementary material.

Although in the current setup UDEPLAMBDA rules are hand-coded, the number of rules are only proportional to the number of UD labels, making rule-writing manageable.\footnote{UD v1.3 has 40 dependency labels, and the number of substitution rules in UDEPLAMBDA are 61, with some labels having multiple rules, and some representing lexical semantics.} Moreover, we view UDEPLAMBDA as a first step towards learning rules for converting UD to richer semantic representations such as PropBank, AMR, or the Parallel Meaning Bank (Palmer et al., 2005; Banarescu et al., 2013; Abzianidze et al., 2017).

4 Cross-lingual Semantic Parsing

To study the multilingual nature of UDEPLAMBDA, we conduct an empirical evaluation on question answering against Freebase in three different languages: English, Spanish, and German. Before discussing the details of this experiment, we briefly outline the semantic parsing framework employed.

### 4.1 Semantic Parsing as Graph Matching

UDEPLAMBDA generates ungrounded logical forms that are independent of any knowledge base, such as Freebase. We use GRAPHPARSER (Reddy et al., 2016) to map these logical forms to their grounded Freebase graphs, via corresponding ungrounded graphs. Figures 3(a) to 3(c) show the ungrounded graphs corresponding to logical forms from UDEPLAMBDA, each grounded to the same Freebase graph in Figure 3(d). Here, rectangles denote entities, circles denote events, rounded rectangles denote entity types, and edges between events and entities denote predicates or Freebase relations. Finally, the TARGET node represents the set of values of \texttt{x} that are consistent with the Freebase graph, that is the answer to the question.

GRAPHPARSER treats semantic parsing as a graph-matching problem with the goal of finding the Freebase graphs that are structurally isomorphic to an ungrounded graph and rank them according to a model. To account for structural mismatches, GRAPHPARSER uses two graph transformations: CONTRACT and EXPAND. In Figure 3(a) there are two edges between \texttt{x} and \texttt{Ghana}. CONTRACT collapses one of these edges to create a graph isomorphic to Freebase. EXPAND, in contrast, adds edges to connect the graph in the case of disconnected components. The search space is explored by beam search and model parameters are estimated with the averaged structured perceptron (Collins, 2002) from training data consisting of question-answer pairs, using answer $F_1$-score as the objective.

### 4.2 Datasets

We evaluate our approach on two public benchmarks of question answering against Freebase: WebQuestions (Berant et al., 2013), a widely used benchmark consisting of English questions and their answers, and GraphQuestions (Su et al., 2016), a recently released dataset of English questions with both their answers and grounded logical forms.
While WebQuestions is dominated by simple entity-attribute questions, GraphQuestions contains a large number of compositional questions involving aggregation (e.g. *How many children of Eddard Stark were born in Winterfell?*) and comparison (e.g. *In which month does the average rainfall of New York City exceed 86 mm?*). The number of training, development and test questions is 2644, 1134, and 2032, respectively, for WebQuestions and 1794, 764, and 2608 for GraphQuestions.

To support multilingual evaluation, we created translations of WebQuestions and GraphQuestions to German and Spanish. For WebQuestions two professional annotators were hired per language, while for GraphQuestions we used a trusted pool of 20 annotators per language (with a single annotator per question). Examples of the original questions and their translations are provided in Table 1.

### 4.3 Implementation Details

Here we provide details on the syntactic analyzers employed, our entity resolution algorithm, and the features used by the grounding model.

**Dependency Parsing** The English, Spanish, and German Universal Dependencies (UD) treebanks (v1.3; Nivre et al. 2016) were used to train part of speech taggers and dependency parsers. We used a bidirectional LSTM tagger (Plank et al., 2016) and a bidirectional LSTM shift-reduce parser (Kiperwasser and Goldberg, 2016). Both the tagger and parser require word embeddings. For English, we used GloVe embeddings (Pennington et al., 2014) trained on Wikipedia and the Gigaword corpus. For German and Spanish, we used SENNA embeddings (Collobert et al., 2011; Al-Rfou et al., 2013) trained on Wikipedia corpora (589M words German; 397M words Spanish). Measured on the UD test sets, the tagger accuracies are 94.5 (English), 92.2 (German), and 95.7 (Spanish), with corresponding labeled attachment parser scores of 81.8, 74.7, and 82.2.

**Entity Resolution** We follow Reddy et al. (2016) and resolve entities in three steps: (1) potential entity spans are identified using seven handcrafted part-of-speech patterns; (2) each span is associated with potential Freebase entities according to the Freebase/KG API; and (3) the 10-best entity linking lattices, scored by a structured perceptron, are input to GRAPHPARSER, leaving the final disambiguation to the semantic parsing problem. Table 2 shows the 1-best and 10-best entity disambiguation $F_1$-scores for each language and dataset.

**Features** We use features similar to Reddy et al. (2016): basic features of words and Freebase relations, and graph features crossing ungrounded events with grounded relations, ungrounded types with grounded relations, and ungrounded answer type crossed with a binary feature indicating if the answer is a number. In addition, we add features encoding the semantic similarity of ungrounded events and Freebase relations. Specifically, we used the cosine similarity of the translation-invariant embeddings of Huang et al. (2015).\(^7\)

### 4.4 Comparison Systems

We compared UD$	ext{EP}$LAMBDA to four versions of GRAPHPARSER that operate on different representations, in addition to prior work.

**SINGLE EVENT** This model resembles the learning-to-rank model of Bast and Haussmann (2015). An ungrounded graph is generated by connecting all entities in the question with the TARGET node, representing a single event. Note that this

---

\(^6\)https://sites.google.com/site/myeid/projects/polyglot.

\(^7\)http://128.2.220.95/multilingual/data.
baseline cannot handle compositional questions, or those with aggregation or comparison.

**DEPTREE** An ungrounded graph is obtained directly from the original dependency tree. An event is created for each parent and its dependents in the tree. Each dependent is linked to this event with an edge labeled with its dependency relation, while the parent is linked to the event with an edge labeled arg0. If a word is a question word, an additional TARGET predicate is attached to its entity node.

**CCGGGRAPH** This is the CCG-based semantic representation of Reddy et al. (2014). Note that this baseline exists only for English.

**UDEPLAMBDA** This is similar to UDEPLAMBDA except that instead of assuming nsubj, dobj and nsubjpass correspond to arg1, arg2 and arg3, we employ semantic role labeling to identify the correct interpretation. We used the systems of Roth and Woodsend (2014) for English and German and Bjørkelund et al. (2009) for Spanish trained on the CoNLL-2009 dataset (Haji et al., 2009). 8

### 4.5 Results

Table 3 shows the performance of GRAPHPARSER with these different representations. Here and in what follows, we use average F1-score of predicted answers (Berant et al., 2013) as the evaluation metric. We first observe that UDEPLAMBDA consistently outperforms the SINGLEEVENT and DEPTREE representations in all languages.

For English, performance is on par with CCGGRAPH, which suggests that UDEPLAMBDA does not sacrifice too much specificity for universality. With both datasets, results are lower for German compared to Spanish. This agrees with the lower performance of the syntactic parser on the German portion of the UD treebank. While UDEPLAMBDA performs better than UDEPLAMBDA SRL.

8The parser accuracies (%) are 87.33, 81.38 and 79.91 for English, German and Spanish respectively.

| Method                  | WebQuestions | | |
|-------------------------|--------------|------------------|------------------|------------------|------------------|
|                         | en de es     | en de es         | en de es         | en de es         | en de es         |
| SINGLEEVENT             | 48.5 45.6 46.3 | 15.9 8.8 11.4 | 48.8 45.9 46.4 | 16.0 8.3 11.3 | 49.5 46.1 47.5 |
| DEPTREE                 | 48.8 45.9 46.4 | 16.0 8.3 11.3 | 49.5 46.1 47.5 | 17.7 9.5 12.8 | 49.8 46.2 47.0 |
| CCGGRAPH                | 49.5 – –     | 15.9 – –        | 49.5 46.1 47.5 | 17.7 9.5 12.8 | 49.8 46.2 47.0 |
| UDEPLAMBDA              | 49.5 46.1 47.5 | 16.0 8.3 11.3 | 49.5 46.1 47.5 | 17.7 9.5 12.8 | 49.8 46.2 47.0 |
| UDEPLAMBDA SRL          | 49.8 46.2 47.0 | 17.7 9.1 12.7 | 49.8 46.2 47.0 | 17.7 9.1 12.7 | 49.8 46.2 47.0 |

Table 3: F1-scores on the test data.

**5 Related Work**

Our work continues the long tradition of building logical forms from syntactic representations initiated by Montague (1973). The literature is rife with

<table>
<thead>
<tr>
<th>Method</th>
<th>GraphQ.</th>
<th>WebQ.</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEMPRE (Berant et al., 2013)</td>
<td>10.8</td>
<td>35.7</td>
</tr>
<tr>
<td>JACANA (Yao and Van Durme, 2014)</td>
<td>5.1</td>
<td>33.0</td>
</tr>
<tr>
<td>PARA_SEMPRE (Berant and Liang, 2014)</td>
<td>12.8</td>
<td>39.9</td>
</tr>
<tr>
<td>QA (Yao, 2015)</td>
<td>–</td>
<td>44.3</td>
</tr>
<tr>
<td>AQUA (Bast and Haussmann, 2015)</td>
<td>–</td>
<td>49.4</td>
</tr>
<tr>
<td>AGENDAIL (Berant and Liang, 2015)</td>
<td>–</td>
<td>49.7</td>
</tr>
<tr>
<td>DEPLAMBDA (Reddy et al., 2016)</td>
<td>–</td>
<td>50.3</td>
</tr>
<tr>
<td>STAGG (Yih et al., 2015)</td>
<td>–</td>
<td>48.4 (52.5)</td>
</tr>
<tr>
<td>BILSTM (Türe and Jojic, 2016)</td>
<td>–</td>
<td>24.9 (52.2)</td>
</tr>
<tr>
<td>MCNN (Xu et al., 2016)</td>
<td>–</td>
<td>47.0 (53.3)</td>
</tr>
<tr>
<td>AGENDAIL-RANK (Yavuz et al., 2016)</td>
<td>–</td>
<td>51.6 (52.6)</td>
</tr>
<tr>
<td>UDEPLAMBDA</td>
<td>17.7</td>
<td>49.5</td>
</tr>
</tbody>
</table>

Table 4: F1-scores on the English GraphQuestions and WebQuestions test sets (results with additional task-specific resources in parentheses).
attempts to develop semantic interfaces for HPSG (Copestake et al., 2005), LFG (Kaplan and Bresnan, 1982; Dalrymple et al., 1995; Crouch and King, 2006), TAG (Kallmeyer and Joshi, 2003; Gardent and Kallmeyer, 2003; Nesson and Shieber, 2006), and CCG (Baltr Aid and Kruijff, 2002; Bos et al., 2004; Artzi et al., 2015). Unlike existing semantic interfaces, UDEPLAMBDA uses dependency syntax, a widely available syntactic resource.

A common trend in previous work on semantic interfaces is the reliance on rich typed feature structures or semantic types coupled with strong type constraints, which can be very informative but unavoidably language specific. Instead, UDEPLAMBDA relies on generic unlexicalized information present in dependency treebanks and uses a simple type system (one type for dependency labels, and one for words) along with a combinatory mechanism, which avoids type collisions. Earlier attempts at extracting semantic representations from dependencies have mainly focused on language-specific dependency representations (Spreyer and Frank, 2005; Simov and Osenova, 2011; Hahn and Meurers, 2011; Reddy et al., 2016; Falke et al., 2016; Beltagy, 2016), and multi-layered dependency annotations (Jakob et al., 2010; Bédaride and Gardent, 2011). In contrast, UDEPLAMBDA derives semantic representations for multiple languages in a common schema directly from Universal Dependencies. This work parallels a growing interest in creating other forms of multilingual semantic representations (Akbik et al., 2015; Vanderwende et al., 2015; White et al., 2016; Evang and Bos, 2016).

We evaluate UDEPLAMBDA on semantic parsing for question answering against a knowledge base. Here, the literature offers two main modeling paradigms: (1) learning of task-specific grammars that directly parse language to a grounded representation (Zelle and Mooney, 1996; Zettlemoyer and Collins, 2005; Berant et al., 2013; Flanigan et al., 2014; Pasupat and Liang, 2015; Groschwitz et al., 2015); and (2) converting language to a linguistically motivated task-independent representation that is then mapped to a grounded representation (Kwiatkowski et al., 2013; Reddy et al., 2014; Krishnamurthy and Mitchell, 2015; Gardner and Krishnamurthy, 2017). Our work belongs to the latter paradigm, as we map natural language to Freebase indirectly via logical forms. Capitalizing on natural-language syntax affords interpretability, scalability, and reduced duplication of effort across applications (Bender et al., 2015). Our work also relates to literature on parsing multiple languages to a common executable representation (Cimiano et al., 2013; Haas and Riezler, 2016). However, existing approaches still map to the target meaning representations (more or less) directly (Kwiatkowski et al., 2010; Jones et al., 2012; Jie and Lu, 2014).

6 Conclusions

We introduced UDEPLAMBDA, a semantic interface for Universal Dependencies, and showed that the resulting semantic representation can be used for question-answering against a knowledge base in multiple languages. We provided translations of benchmark datasets in German and Spanish, in the hope to stimulate further multilingual research on semantic parsing and question answering in general. We have only scratched the surface when it comes to applying UDEPLAMBDA to natural language understanding tasks. In the future, we would like to explore how this framework can benefit applications such as summarization (Liu et al., 2015) and machine reading (Sachan and Xing, 2016).

Acknowledgements

This work greatly benefited from discussions with Michael Collins, Dipanjan Das, Federico Fancellu, Julia Hockenmaier, Tom Kwiatkowski, Adam Lopez, Valeria de Paiva, Martha Palmer, Fernando Pereira, Emily Pitler, Vijay Saraswat, Nathan Schneider, Bonnie Webber, Luke Zettlemoyer, and the members of ILCC Edinburgh University, the Microsoft Research Redmond NLP group, the Stanford NLP group, and the UW NLP and Linguistics group. We thank Reviewer 2 for useful feedback. The authors would also like to thank the Universal Dependencies community for the treebanks and documentation. This research is supported by a Google PhD Fellowship to the first author. We acknowledge the financial support of the European Research Council (Lapata; award number 681760).

References


98


