Benchmarking Meaning Representations in Neural Semantic Parsing

Jiaqi Guo[1]*, Qian Liu[2]*, Jian-Guang Lou[3], Zhenwen Li[4]
Xueqing Liu[5], Tao Xie[4], Ting Liu[1]
[1]Xi'an Jiaotong University, Xi'an, China [2]Beihang University, Beijing, China
[3]Microsoft Research, Beijing, China [4]Peking University, Beijing, China
[5]Stevens Institute of Technology, Hoboken, New Jersey, USA
jasperguo2013@stu.xjtu.edu.cn, qian.liu@buaa.edu.cn
jlou@microsoft.com, {lizhenwen, taoxie}@pku.edu.cn
xueqing.liu@stevens.edu, tingliu@mail.xjtu.edu.cn

Abstract

Meaning representation is an important component of semantic parsing. Although researchers have designed a lot of meaning representations, recent work focuses on only a few of them. Thus, the impact of meaning representation on semantic parsing is less understood. Furthermore, existing work's performance is often not comprehensively evaluated due to the lack of readily-available execution engines. Upon identifying these gaps, we propose UNIMER, a new unified benchmark on meaning representations, by integrating existing semantic parsing datasets, completing the missing logical forms, and implementing the missing execution engines. The resulting unified benchmark contains the complete enumeration of logical forms and execution engines over three datasets × four meaning representations. A thorough experimental study on UNIMER reveals that neural semantic parsing approaches exhibit notably different performance when they are trained to generate different meaning representations. Also, program alias and grammar rules heavily impact the performance of different meaning representations. Our benchmark, execution engines and implementation can be found on: https://github.com/JasperGuo/Unimer.

1 Introduction

A remarkable vision of artificial intelligence is to enable human interactions with machines through natural language. Semantic parsing has emerged as a key technology for achieving this goal. In general, semantic parsing aims to transform a natural language utterance into a logic form, i.e., a formal, machine-interpretable *meaning representation* (MR) (Zelle and Mooney, 1996; Dahl et al., 1994).[1] Thanks to the recent development of neural networks techniques, significant improvements have been made in semantic parsing performance (Jia and Liang, 2016; Yin and Neubig, 2017; Dong and Lapata, 2018; Shaw et al., 2019).

MR	Geo	ATIS	Job
Prolog	-	-	91.4
Lambda	90.4	91.3	85.0
FunQL	92.5	-	-
SQL	78.0	69.0	-
Prolog	89.6	-	92.1
Lambda	-	-	-
FunQL	-	-	-
SQL	82.5	79.2	-

Table 1: State-of-the-art performance for MRs on Geo, ATIS, and Job. The top table shows exact-match accuracy whereas the bottom table shows execution-match accuracy. Most existing work focuses on evaluating only a small subset of dataset × MR pairs, leaving most of the table unexplored. A finer-grained table is available in the supplementary material (Table 8).

Despite the advancement in performance, we identify three important biases in existing work's evaluation methodology. First, although multiple MRs are proposed, most existing work is evaluated on only one or two of them, leading to less comprehensive or even unfair comparisons. Table 1 shows the state-of-the-art performance of semantic parsing on different dataset × MR combinations, where the rows are the MRs and the columns are the datasets. We can observe that while Lambda Calculus is intensively studied, the other MRs have not been sufficiently studied. This biased evaluation is partly caused by the absence of target logic forms in the missing cells. Second, existing work often compares the performance on different MRs directly (Sun et al., 2020; Shaw et al., 2019; Chen et al., 2020) without considering the confounding

*Work done during an internship at Microsoft Research.

[1] In this paper, we focus on *grounded* semantic parsing, where meaning representations are grounded to specific knowledge bases, instead of ungrounded semantic parsing.

Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing, pages 1520–1540,
November 16–20, 2020. ©2020 Association for Computational Linguistics

role that MR plays in the performance,[2] causing unfair comparisons and misleading conclusions. Third, a more comprehensive evaluation methodology would consider both the exact-match accuracy and the execution-match accuracy, because two logic forms can be semantically equivalent yet do not match precisely in their surface forms. However, as shown in Table 1, most existing work is only evaluated with the exact-match accuracy. This bias is potentially due to the fact that execution engines are not available in six out of the twelve dataset × MR combinations.

Upon identifying the three biases, in this paper, we propose UNIMER, a new unified benchmark, by unifying four publicly available MRs in three of the most popular semantic parsing datasets: Geo, ATIS and Jobs. First, for each natural language utterance in the three datasets, UNIMER provides annotated logical forms in four different MRs, including Prolog, Lambda Calculus, FunQL, and SQL. We identify that annotated logical forms in some MR × dataset combinations are missing. As a result, we complete the benchmark by semi-automatically translating logical forms from one MR to another. Second, we implement six missing execution engines for MRs so that the execution-match accuracy can be readily computed for all the dataset × MR combinations. Both the logical forms and their execution results are manually checked to ensure the correctness of annotations and execution engines.

After constructing UNIMER, to obtain a preliminary understanding on the impact of MRs on semantic parsing, we empirically study the performance of MRs on UNIMER by using two widely-used neural semantic parsing approaches (a seq2seq model (Dong and Lapata, 2016; Jia and Liang, 2016) and a grammar-based neural model (Yin and Neubig, 2017)), under the supervised learning setting.

In addition to the empirical study above, we further analyze the impact of two operations, i.e., *program alias* and *grammar rules*, to understand how they affect different MRs differently. First, *Program alias*. A semantically equivalent program may have many syntactically different forms. As a result, if the training and testing data have a difference in their syntactic distributions of logic forms, a naive maximum likelihood estimation can suffer from this difference because it fails to capture the semantic equivalence (Bunel et al., 2018). As different MRs have different degrees of syntactic difference, they suffer from this problem differently. Second, *Grammar rules*. Grammar-based neural models can guarantee that the generated program is syntactically correct (Yin and Neubig, 2017; Wang et al., 2020; Sun et al., 2020). For a given set of logical forms in an MR, there exist multiple sets of grammar rules to model them. We observe that when the grammar-based neural model is trained with different sets of grammar rules, it exhibits a notable performance discrepancy. This finding alias with the one made in traditional semantic parsers (Kate, 2008) that properly transforming grammar rules can lead to better performance of a traditional semantic parser.

In summary, this paper makes the following main contributions:

- We propose UNIMER, a new unified benchmark on meaning representations, by integrating and completing semantic parsing datasets in three datasets × four MRs; we also implement six execution engines so that execution-match accuracy can be evaluated in all cases;
- We provide the baseline results for two widely used neural semantic parsing approaches on our benchmark, and we conduct an empirical study to understand the impact that program alias and grammar rule plays on the performance of neural semantic parsing;

2 Preliminaries

In this section, we provide a brief description of the MRs and neural semantic parsing approaches that we study in the paper.

2.1 Meaning Representations

We investigate four MRs in this paper, namely, Prolog, Lambda Calculus, FunQL, and SQL, because they are widely used in semantic parsing and we can obtain their corresponding labeled data in at least one semantic parsing domain. We regard Prolog, Lambda Calculus, and FunQL as domain-specific MRs, since the predicates defined in them are specific for a given domain. Consequently, the execution engines of domain-specific MRs need to be significantly customized for different domains, requiring plenty of manual efforts. In contrast, SQL is a domain-general MR for querying relational

[2] In (Kate et al., 2005; Liang et al., 2011; Guo et al., 2019), it was revealed that using an appropriate MR can substantially improve the performance of a semantic parser.

MR	Logical Form
Prolog	answer(A, (flight(A) , tomorrow(A) , during_day(A, B) , const (B, period (morning)), from(A, C) , const(C, city(Pittsburgh)), to(A, D), const (D, city(Atlanta))))
Lambda Calculus	(lambda A:e ((flight A) $\wedge$ (during_day A morning:pd) $\wedge$ (from A Pittsburgh:ci) $\wedge$ (to A Atlanta:ci) $\wedge$ (tomorrow A)))
FunQL	answer (flight (tomorrow (intersect (during_day (period (morning)) , from (city (Pittsburgh)) , to (city (Atlanta))))))
SQL	SELECT flight_id FROM … WHERE city_1.city_name = 'pittsburgh' AND city_2.city_name = 'atlanta' AND date_day_1.year = 1991 AND date_day_1.month_number = 1 AND date_day_1.day_number = 20 AND departure_time BETWEEN 0 AND 1200

Table 2: Examples of meaning representations for utterance "what flights do you have in tomorrow morning from pittsburgh to atlanta?" in the ATIS domain.

databases. Its execution engines (e.g., MySQL) can be used directly in different domains. Table 2 shows a logical form for each of the four MRs in the ATIS domain.

Prolog has long been used to represent the meaning of natural language (Zelle and Mooney, 1996; Kate and Mooney, 2006). Prolog includes first-order logical forms, augmented with some higher-order predicates, e.g., `most`, to handle issues such as quantification and aggregation. Take the first logical form in Tables 2 as an example. The uppercase characters denote variables, and the predicates in the logical form specify the constraints between variables. In this case, character `A` denotes a variable, and it is required to be a flight, and the flight should depart tomorrow morning from Pittsburgh to Atlanta. The outer predicate `answer` indicates the variable whose binding is of interest. One major benefit of Prolog-style MRs is that they allow predicates to be introduced in the order where they are actually named in the utterance. For instance, the order of predicates in the logical form strictly follows their mentions in the natural language utterance.

Lambda Calculus is a formal system to express computation. It can represent all first-order logic and it naturally supports higher-order functions. It represents the meanings of natural language with logical expressions that contain constants, quantifiers, logical connectors, and lambda abstract. These properties make it prevalent in semantic parsing. Consider the second logical form in Table 2. It defines an expression that takes an entity `A` as input and returns true if the entity satisfies the constraints defined in the expressions. Lambda Calculus can be typed, allowing type checking during generation and execution.

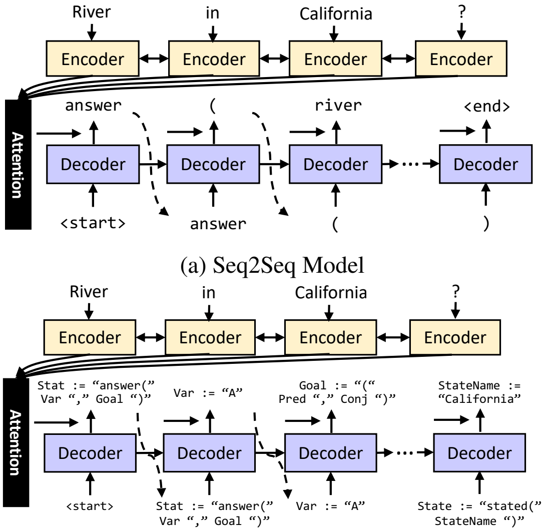
FunQL, abbreviated for Functional Query Language, is a variable-free language (Kate et al., 2005). It abstracts away variables and encodes compositionality via its nested function-argument structure, making it easier to implement an efficient execution engine for FunQL. Concretely, unlike Prolog and Lambda Calculus, predicates in FunQL take a set of entities as input and return another set of entities that meet certain requirements. Considering the third logical form in Table 2, the predicate `during_day(period(morning))` returns a set of flights that depart in the morning. With this function-argument structure, FunQL can directly return the entities of interest.

SQL is a popular relational database query language. Since it is domain-agnostic and has well-established execution engines, the subtask of semantic parsing, Text-to-SQL, has received a lot of interests. Compared with domain-specific MRs, SQL cannot encapsulate too much domain prior knowledge in its expressions. As shown in Table 2, to query flights that depart tomorrow, one needs to specify the concrete values of year, month, and day in the SQL query. However, these values are not explicitly mentioned in the utterance and may even change over time.

It is important to note that although these MRs are all expressive enough to represent all meanings in some domains, they are not equivalent in terms of their general expressiveness. For example, FunQL is less expressive than Lambda Calculus in general, partially due to the elimination of variables and quantifiers.

2.2 Neural Semantic Parsing Approaches

During the last few decades, researchers have proposed different approaches for semantic parsing. Most state-of-the-art approaches are based on neural models and formulate the semantic parsing problem as a sequence transduction problem. Due to the generality of sequence transduction, these ap-



(a) Seq2Seq Model

(b) Grammar-based Model

Figure 1: Illustrations of the seq2seq model and the grammar-based model with utterance “Rivers in California?” and its corresponding logical form in Prolog.

proaches can be trained to generate any MRs. In this work, without loss of generality, we benchmark MRs by evaluating the seq2seq model (Dong and Lapata, 2016; Jia and Liang, 2016) and the grammar-based model (Yin and Neubig, 2017) under the supervised learning setting. We select the two models because most neural approaches are designed based on them.

Seq2Seq Model. Dong and Lapata (2016) and Jia and Liang (2016) formulated the semantic parsing problem as a neural machine translation problem and employed the sequence-to-sequence model (Sutskever et al., 2014) to solve it. As illustrated in Figure 1a, the encoder takes an utterance as input and outputs a distributed representation for each word in the utterance. A decoder then sequentially predicts words in the logical form. When augmented with the attention mechanism (Bahdanau et al., 2014; Luong et al., 2015), the decoder can better utilize the encoder's information to predict logical forms. Moreover, to address the problem caused by the long tail distribution of entities in logical forms, Jia and Liang (2016) proposed an attention-based copying mechanism. That is, at each time step, the decoder takes one of two types of actions, one to predict a word from the vocabulary of logical forms and the other to copy a word from the input utterance.

Grammar-based Model. By treating a logical form as a sequence of words, the seq2seq model cannot fully utilize the property that logical forms are well-formed and must conform to certain grammars of an MR. To bridge this gap, Yin and Neubig (2017) proposed a grammar-based decoder that outputs a sequence of *grammar rules* instead of words, as presented in Figure 1b. The decoded grammar rules can deterministically generate a valid abstract syntax tree (AST) of a logical form. In this way, the generated logical form is guaranteed to be syntactically correct. This property makes it widely used in a lot of code generation and semantic parsing tasks (Sun et al., 2020; Wang et al., 2020; Bogin et al., 2019). The grammar-based decoder can also be equipped with the attention-based copying mechanism to address the long-tail distribution problem.

3 Benchmark

To provide an infrastructure for exploring MRs, we construct UNIMER, a unified benchmark on MRs, based on existing semantic parsing datasets. Currently, UNIMER covers three domains, namely Geo, ATIS, and Job, each of which has been extensively studied in previous work and has annotated logical forms for at least two MRs. All natural language utterances in UNIMER are written in English.

Geo focuses on querying a database of U.S. geography with natural language. To solve the problem, Zelle and Mooney (1996) designed a Prolog-style MR and annotated 880 (utterance, logical form) pairs. Popescu et al. (2003) and Kate et al. (2005) proposed to use SQL and FunQL to represent the meanings, respectively. Almost the same time, Zettlemoyer and Collins (2005) proposed to use Lambda Calculus and manually converted the Prolog logical forms to equivalent expressions in Lambda Calculus. Following their work, we adopt the standard 600/280 training/test split.

ATIS is a dataset of flight booking questions. It consists of 5,418 questions and their corresponding SQL queries. Zettlemoyer and Collins (2007) proposed to use Lambda Calculus to represent the meanings of natural language and automatically map these SQL queries to its equivalent logical forms in Lambda Calculus. Following the work of Kwiatkowski et al. (2011), we use the standard 4480/480/450 training/dev/test split.

Job is a dataset about job announcements posted in the newsgroup austin.jobs (Califf and Mooney, 1999). It consists of 640 utterances and their corresponding Prolog logical forms that query computer-related job postings. Similarly, in Geo, Zettlemoyer and Collins (2005) proposed using Lambda Calculus and manually converted them to equivalent expressions in Lambda Calculus. We use the same training-test split with them, containing 500 training and 140 test instances.

Since not all the four MRs that we introduce in Section 2.1 are used in these three domains, we semi-automatically translate logical forms in one MR into another. This effort enables researchers to explore MRs in more domains and make a fair comparison among them. Take the translation of Lambda Calculus to FunQL in ATIS as an example. We first design predicates for FunQL based on those defined in Lambda Calculus and implement an execution engine for FunQL. Then, we translate logical forms in Lambda Calculus to FunQL and compare the execution results to verify the correctness of the translation. In this process, we find that there is no ready-to-use Lambda Calculus execution engine for the three domains. Hence, we implement one for each domain. These engines, on the one hand, enable evaluations of semantic parsing approaches with both exact-match accuracy and execution-match accuracy. On the other hand, they enable exploration of weakly supervised semantic parsing with Lambda Calculus. In addition, we find some annotation mistakes in logical forms and several bugs in existing execution engines of Prolog and FunQL. By correcting the mistakes and fixing the bugs in the engines, we create a refined version of these datasets. Section A.1 in the supplementary material provides more details about the construction process.

We plan to cover more domains and more MRs in UNIMER. We have made UNIMER along with the execution engines publicly available.[3] We believe that UNIMER can provide fertile soil for exploring MRs and addressing challenges in semantic parsing.

4 Experimental Setup

Based on UNIMER, we take the first attempt to study the characteristics of different MRs and their impact on neural semantic parsing.

4.1 Experimental Design

Meaning Representation Comparison. To understand the impact of MRs on neural semantic parsing, we first experiment with the two neural approaches described in Section 2.2 on UNIMER, and we compare the resulting performance of different MRs with two metrics: *exact-match accuracy* (a logical form is regarded as correct if it is syntactically identical to the gold standard),[4] and *execution-match accuracy* (regarded as correct if a logical form's execution result is identical to that of the gold standard).[5]

Rule	Description
Shuffle	Shuffle expressions in Select, From, Where, and Having clauses
Argmax	Express Argmax/min with OrderBy and Limit clause instead of subquery
In2Join	Replace In clause with Join clause

Table 3: Three basic transformation rules for SQL.

Program Alias. To explore the effect of program alias, we replace a different proportion of logical forms in a training set with their aliases (semantically equivalent but syntactically different logical forms), and we re-train the neural approaches to quantify its effect. To search for aliases of a logical form, we first derive multiple transformation rules for each MR. Then, we apply these rules to the logical form to get its aliases and randomly sample one. We compare the execution results of the resulting logical forms to ensure their equivalence in semantics. Table 3 presents three transformation rules for SQL. We provide a detailed explanation of transformation rules and examples for each MR in Section A.3 of the supplementary material.

Grammar Rules. To understand the grammar rules' impact on grammar-based models, we provide two sets of grammar rules for each MR. Each set of rules can cover all the logical forms in the three domains. We compare the performance of models trained with different sets of rules. Specifically, Wong and Mooney (2006) and Wong and Mooney (2007) have induced a set of grammar rules for Prolog and FunQL in Geo. We directly use them in Geo and extend them to support logical forms in ATIS and Job. As for SQL, Bogin et al. (2019) have induced a set of rules for SQL in the Spider benchmark, and we adapt it to support the SQL queries in the three domains that we study.

[3] Our benchmark, execution engines and and our implementation can be found on: https://github.com/JasperGuo/Unimer

[4] Following Dong and Lapata (2016), we sort the sub-expressions in the conjunction predicate of Lambda Calculus before comparison.

[5] It should be acknowledged that using the execution-match accuracy to evaluate a parser is not enough, as there exist spurious programs that lead to the same execution results with the gold standard but have different semantics.

When it comes to Lambda Calculus, we use the one induced by Yin and Neubig (2018). For comparison, we also manually induce another set of grammar rules for the four MRs. Section A.4 in the supplementary material provides definitions of all the grammar rules.

4.2 Implementations

We implement each approach with the AllenNLP (Gardner et al., 2018) and PyTorch (Paszke et al., 2019) frameworks. To make a fair comparison, we tune the hyper-parameters of approaches for each MR on the development set or through cross-validation on the training set, with the NNI platform.[6] Due to the limited number of test data in each domain, we run each approach five times and take the average number. Section A.2 in the supplementary material provides the search space of hyper-parameters for each approach and the preprocessing procedures of logical forms.

Multiple neural semantic parsing approaches (Dong and Lapata, 2016; Iyer et al., 2017; Rabinovich et al., 2017) adopt the data anonymization techniques to replace entities in utterances with placeholders. However, the techniques are usually ad-hoc and specific for domains and MRs, and they sometimes require manual efforts to resolve conflicts (Finegan-Dollak et al., 2018). Hence, we do not apply data anonymization to avoid bias.

5 Experimental Results

5.1 Meaning Representation Comparison

Table 4 presents our experimental results on UNIMER. Since we do not use data anonymization techniques, the performance is generally lower than that shown in Table 1 and Table 8, but the performance is on par with the numbers reported in ablation studies of previous work (Dong and Lapata, 2016; Jia and Liang, 2016; Finegan-Dollak et al., 2018). We can make the following three observations from the table.

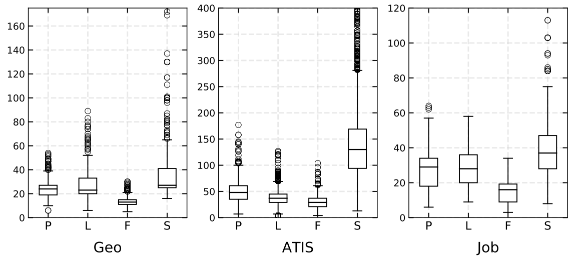


Figure 2: Statistics of logical forms in the training set of Prolog (P), Lambda Calculus (L), FunQL (F) and SQL (S). The y-axis indicates the number of production rules in each logical form.

First, neural approaches exhibit notably different performance when they are trained to generate different MRs. The difference can vary by as much as 20% in both exact-match and execution-match metrics. This finding tells us that an apple-to-apple comparison is extremely important when comparing two neural semantic parsing approaches. However, we notice that some papers (Sun et al., 2020; Shaw et al., 2019; Chen et al., 2020) do not clearly note the MRs used in baselines and compare with them using different metrics; the attained result can be somewhat misleading.

Second, domain-specific MRs (Prolog, Lambda Calculus, and FunQL) tend to outperform SQL (domain-general) by a large margin. For example, in Geo, the execution-match accuracy of FunQL is substantially higher than that of SQL in all approaches. This result is expected because a lot of domain knowledge is injected into domain-specific MRs. Consider the logical forms in Table 2. There is a predicate `tomorrow` in all three domain-specific MRs, and this predicate can directly align to the description in the utterance. However, one needs to explicitly express the concrete date values in the SQL query; this requirement can be a heavy burden for neural approaches, especially when the values will change over time. In addition, a recent study (Finegan-Dollak et al., 2018) in Text-to-SQL has shown that domain-specific MRs are more robust against generating never-seen logical forms than SQL, because their surface forms are much closer to natural language.

Third, among all the domain-specific MRs, FunQL tends to outperform the others in neural approaches. In Geo, FunQL outperforms the other MRs in both metrics by a large margin. In Job, the grammar-based (w/ copy) model trained with FunQL achieves the state-of-the-art performance. One possible reason is that FunQL is more compact than the other MRs, due to its elimination of variables and quantifiers. Figure 2 shows box plots about the number of grammars rules in the AST of a logical form. We can observe that while FunQL has almost the same number of grammar rules with the other MRs (Table 5), it has much fewer grammar rules involved in a logical form than the others

[6] https://github.com/microsoft/nni

Approach	Prolog		Lambda Calculus		FunQL		SQL	
	Exact	Execution	Exact	Execution	Exact	Execution	Exact	Execution
Geo								
Seq2Seq	70.0 ±2.1	73.9 ±2.4	64.7 ±2.3	70.1 ±2.1	**76.8** ±2.4	**79.4** ±2.2	58.0 ±2.4	68.7 ±3.7
w/ Copy	72.9 ±2.1	78.7 ±2.4	75.4 ±0.5	80.1 ±1.3	**80.3** ±1.4	**87.1** ±0.9	72.3 ±1.1	76.8 ±1.8
Grammar	68.6 ±1.2	75.7 ±2.3	70.7 ±1.1	75.1 ±1.4	**76.1** ±0.3	**78.2** ±0.3	63.3 ±2.0	70.8 ±1.9
w/ Copy	74.3 ±2.1	79.5 ±1.1	75.6 ±1.5	80.7 ±0.7	**81.8** ±0.3	**86.2** ±0.4	67.9 ±0.7	72.1 ±1.0
ATIS								
Seq2Seq	65.9 ±1.5	73.8 ±1.3	68.7 ±1.8	**76.3** ±0.9	**70.8** ±0.6	**76.3** ±1.0	5.6 ±0.3	61.1 ±4.6
w/ Copy	73.7 ±1.9	80.4 ±0.5	75.4 ±2.4	**83.1** ±1.3	**78.0** ±1.1	82.7 ±1.0	8.0 ±0.6	70.0 ±1.5
Grammar	70.9 ±0.9	76.3 ±1.0	71.7 ±1.5	77.4 ±1.3	**72.1** ±0.8	**77.5** ±0.9	5.5 ±0.2	63.7 ±1.3
w/ Copy	73.4 ±1.2	79.2 ±1.1	75.7 ±0.4	82.1 ±0.8	**76.5** ±0.7	**82.7** ±1.2	7.2 ±0.6	61.0 ±1.1
Job								
Seq2Seq	68.1 ±2.3	75.7 ±1.7	65.8 ±1.3	78.6 ±1.5	**71.4** ±2.1	**81.4** ±2.3	68.5 ±3.0	75.6 ±3.0
w/ Copy	71.4 ±3.6	79.1 ±2.2	77.9 ±2.8	**88.0** ±1.4	75.6 ±2.4	85.9 ±3.0	**78.7** ±2.1	87.3 ±2.3
Grammar	70.4 ±2.4	79.4 ±2.4	68.4 ±3.9	82.9 ±4.1	72.9 ±2.7	**86.3** ±2.5	**74.6** ±1.5	83.3 ±1.8
w/ Copy	75.9 ±1.1	84.4 ±2.1	**80.2** ±1.8	91.0 ±1.3	78.4 ±2.1	**92.4** ±1.5	78.7 ±1.4	87.6 ±2.1

Table 4: Experimental results of the seq2seq model and the grammar-based model. We highlight the best performance of each approach and MR in both metrics. The poor SQL exact-match accuracy in ATIS is caused by the different distribution of SQL queries in test set. Iyer et al. (2017) rewrote the SQL queries in test set to improve their execution efficiency.

MR	Geo		ATIS		Job	
	# Vo	# Ru	# Vo	# Ru	# Vo	# Ru
Prolog	146	234	503	732	170	230
Lambda	180	245	466	532	200	208
FunQL	152	188	494	706	195	208
SQL	187	269	530	656	211	227

Table 5: Statistics of logical forms in different MRs. '# Vo' and '# Ru' indicate the vocabulary size and the number of grammar rules of an MR, respectively.

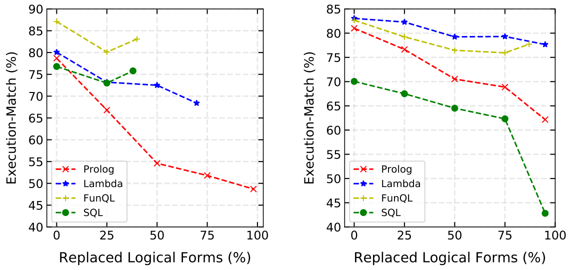
on average. This statistic is crucial for neural semantic parsing approaches as it directly determines the number of decoding steps in decoders.

A similar reason can be used to explain that the performance on SQL is lower than others. As Figure 2 shows, SQL has larger medians of the number of grammar rules, and it also has much more outliers than domain-specific MRs. It makes neural models more challenging to learn.

Interestingly, this finding contradicts the finding in CCG-based semantic parsing approaches (Kwiatkowksi et al., 2010), in which they show that Lambda Calculus outperforms FunQL in the Geo domain. The reason is that compared with Lambda Calculus, the deeply nested structure of FunQL makes it more challenging to learn a high-quality CCG lexicon, which is crucial for CCG parsing. In contrast, neural approaches do not rely on a lexicon and directly learn a mapping between source and target languages.

5.2 Program Alias

Figure 3 shows the execution-match accuracy of the seq2seq (w/ copy) model with different proportions of logical forms in the training set replaced with aliases.[7] Since not all the logical forms have aliases, the curves in Figure 3 stop at different points. Among all the domain-specific MRs, FunQL has the fewest logical forms with aliases, while Prolog has the largest. For example, in Geo, only 42% of FunQL logical forms in the training set have aliases, while more than 70% of logical forms in Lambda Calculus and Prolog have aliases.



(a) Geo (b) ATIS

Figure 3: Execution-match accuracy of the seq2seq (w/ copy) model when different proportions of logical forms in training set are replaced with their aliases.

MR	Geo		ATIS	
	Exact	Execution	Exact	Execution
Prolog	24.2%	15.1%	7.0%	5.4%
Lambda	11.9%	8.6%	6.2%	0.9%
FunQL	24.7%	8.0%	7.0%	4.1%
SQL	19.4%	4.9%	9.1%	3.6%

Table 6: Relative decline of performance when 25% of logical forms in training data are replaced with aliases.

[7] We have experimented with the grammar-based model and observed similar results.

MR	Grammar	Geo	ATIS
Prolog	G1	72.6 ±1.2	**77.5** ±0.9
	G2	**75.7** ±2.3	76.2 ±1.0
Lambda	G1	75.1 ±1.4	74.9 ±1.5
	G2	**75.6** ±1.5	**77.4** ±1.3
FunQL	G1	74.5 ±1.7	74.0 ±2.2
	G2	**78.2** ±0.4	**77.5** ±0.9
SQL	G1	68.2 ±2.4	**63.7** ±1.3
	G2	**70.2** ±1.0	61.3 ±1.7

Table 7: Execution-match accuracy of the grammar-based model (w/o Copy). 'G1' denotes the grammar rules induced by (Wong and Mooney, 2006, 2007; Yin and Neubig, 2018; Bogin et al., 2019); 'G2' denotes the grammar rules induced by ourselves.

From the figure, we have two main observations. First, in both domains, as more logical forms are replaced, the performance of all MRs declines gradually. Among all the MRs, the performance of Prolog declines more seriously than the others in both domains. In other words, it suffers from the program alias problem more seriously. The trends of Lambda Calculus and FunQL in ATIS are impressive, as their performance decreases only slowly. Selecting an MR with the less effect of program alias could be a better choice when we need to develop a semantic parser for a new domain, because we can save many efforts in defining annotation protocols and checking consistency, which could be extremely tedious. Second, the exact-match accuracy declines more seriously than execution-match. Table 6 provides the relative declines in both exact-match and execution-match metrics when 25% of logical forms are replaced. We find that the exact-match accuracy declines more seriously than execution-match, indicating that under the effect of program alias, exact-match may not be suitable as it may massively underestimate the performance. At last, given a large number of semantically equivalent logical forms, it would be valuable to explore whether they can be leveraged to improve semantic parsing (Zhong et al., 2018).

5.3 Grammar Rules

Table 7 presents the experimental results of the grammar-based (w/o copy) model trained with different sets of grammar rules. As the table shows, there is a notable performance discrepancy between different sets of rules. For example, in ATIS, we can observe 2.5% absolute improvement when the model is trained with G2 for Lambda Calculus. Moreover, G2 is not always better than G1. While the model trained with G2 for Prolog outperforms G1 in Geo, it lags behind G1 in ATIS. This observation motivates us to consider what factors contribute to the discrepancy. We had tried to explore the search space of logical forms defined by different grammar rules and the distribution drift between the AST of logical forms in the training and test set. However, the exploration results cannot consistently explain the performance discrepancy. As our important future work, we would explore whether or not the discrepancy is caused by better alignments between utterances and grammar rules. Intuitively, it would be easier for decoders to learn the set of grammar rules having better alignments with utterances.

We can learn from these results that similar to traditional semantic parsers, properly transforming grammar rules for MRs can also lead to better performance in neural approaches. Therefore, grammar rules should be considered as a very important hyper-parameter of grammar-based models, and it is recommended to mention the used grammar rules in research papers clearly.

6 Related Work

Meaning representations for semantic parsing. Recent work has shown that properly designing new MRs often helps improve the performance of neural semantic parsing. Except for the four MRs that we study, Liang et al. (2011); Liang (2013) presented DCS and lambda DCS for querying knowledge bases and demonstrated their advantages over Lambda Calculus. Guo et al. (2019) proposed SemQL, an MR for relational database queries, and they showed improvements in the Spider benchmark. Wolfson et al. (2020) designed an MR, named QDMR, for representing the meaning of questions through question decomposition. Instead of designing new MRs, Cheng et al. (2019) proposed a transition-based semantic parsing approach that supports generating tree-structured logical forms in either top-down or bottom-up manners. Their experimental results on various semantic parsing datasets using FunQL showed that top-down generation outperforms bottom-up in various settings. Our work aims to investigate the characteristics of different MRs and their impact on neural semantic parsing.

Ungrounded semantic parsing. Except for the grounded MRs studied in this work, there are also ungrounded MRs that are not tied to any particular applications, such as AMR (Banarescu et al., 2013), MRS (Copestake et al., 2005; Flickinger et al., 2017), and UCCA (Abend and Rappoport, 2013). Abend and Rappoport (2017) conducted a survey on ungrounded MRs to assess their achievements and shortcomings. Hershcovich et al. (2019) evaluated the similarities and divergences in the content encoded by ungrounded MRs and syntactic representation. Lin and Xue (2019) carried out a careful analysis on AMR and MRS to understand the factors contributing to the discrepancy in their parsing accuracy. Partly inspired by this line of work, we conduct a study on grounded MRs and investigate their impact on neural semantic parsing. Hershcovich et al. (2018) proposed to leverage annotated data in different ungrounded MRs to improve parsing performance. With UNIMER, we can explore whether it is feasible in grounded semantic parsing.

Extrinsic parser evaluation. Another line of research that is closely related to our work is extrinsic parser evaluation. Miyao et al. (2008) benchmarked different syntactic parsers and their representations, including dependency parsing, phrase structure parsing, and deep parsing, and evaluated their impact on an information extraction system. Oepen et al. (2017) provided a flexible infrastructure, including data and software, to estimate the relative utility of different types of dependency representations for a variety of downstream applications that rely on an analysis of grammatical structure of natural language. There has not been work on benchmarking MRs for grounded semantic parsing in neural approaches, to the best of our knowledge.

Weakly supervised semantic parsing. In this paper, we focus on supervised learning for semantic parsing, where each utterance has its corresponding logical form annotated. But the similar evaluation methodology could be applied to weakly supervised semantic parsing, which receives wide attention because parsers are only supervised with execution results and annotated logical forms are no longer required (Berant et al., 2013; Pasupat and Liang, 2015; Goldman et al., 2018; Liang et al., 2018; Mueller et al., 2019). We also notice that various MRs have been used in weakly supervised semantic parsing, and it would be valuable to explore the impact of MRs in such settings.

7 Conclusion

In this work, we propose UNIMER, a unified benchmark on meaning representations, based on established semantic parsing datasets; UNIMER covers three domains and four different meaning representations along with their execution engines. UNIMER allows researchers to comprehensively and fairly evaluate the performance of their approaches. Based on UNIMER, we conduct an empirical study to understand the characteristics of different meaning representations and their impact on neural semantic parsing. By open-sourcing our source code and benchmark, we believe that our work can facilitate the community to inform the design and development of next-generation MRs.

Implications. Our findings have clear implications for future work. First, according to our experimental results, FunQL tends to outperform Lambda Calculus and Prolog in neural semantic parsing. Additionally, FunQL is relatively robust against program alias. Hence, when developers need to design an MR for a new domain, FunQL is recommended to be the first choice. Second, to reduce program alias' negative effect on neural semantic parsing, developers should define a concrete protocol for annotating logical forms to ensure their consistency. Specifically, given an MR, developers should identify as many as possible sources where program alias can occur. Take SQL as an example. To express the `argmax` semantics, one can either use subquery or the OrderBy clause.[8] Having identified these sources, developers need to determine using which expression in what context, e.g., `argmax` is always expressed with subquery, and the unordered expressions in conjunctions are always sorted by characters.

Acknowledgments

We would like to thank Bei Chen for helpful discussions and thank the anonymous reviewers for their constructive comments. Ting Liu was partially supported by NSFC (61632015, 61721002) and State Grid R&D Program (5226SX1800FC).

References

Omri Abend and Ari Rappoport. 2013. UCCA: A semantics-based grammatical annotation scheme. In *Proceedings of the 10th International Conference*

[8] A concrete example is provided in Section A.3 in the supplementary material.

on Computational Semantics, pages 1–12, Potsdam, Germany. Association for Computational Linguistics.

Omri Abend and Ari Rappoport. 2017. The state of the art in semantic representation. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics*, pages 77–89, Vancouver, Canada. Association for Computational Linguistics.

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2014. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*.

Laura Banarescu, Claire Bonial, Shu Cai, Madalina Georgescu, Kira Griffitt, Ulf Hermjakob, Kevin Knight, Philipp Koehn, Martha Palmer, and Nathan Schneider. 2013. 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.

Jonathan Berant, Andrew Chou, Roy Frostig, and Percy Liang. 2013. Semantic parsing on Freebase from question-answer pairs. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*, pages 1533–1544, Seattle, Washington, USA. Association for Computational Linguistics.

Ben Bogin, Matt Gardner, and Jonathan Berant. 2019. Global reasoning over database structures for text-to-SQL parsing. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing*, pages 3659–3664, Hong Kong, China. Association for Computational Linguistics.

Rudy Bunel, Matthew Hausknecht, Jacob Devlin, Rishabh Singh, and Pushmeet Kohli. 2018. Leveraging grammar and reinforcement learning for neural program synthesis. In *International Conference on Learning Representations*.

Mary Elaine Califf and Raymond J. Mooney. 1999. Relational learning of pattern-match rules for information extraction. In *Proceedings of the 16th National Conference on Artificial Intelligence and the 7th Innovative Applications of Artificial Intelligence Conference Innovative Applications of Artificial Intelligence*, page 328–334, USA. American Association for Artificial Intelligence.

Ruisheng Cao, Su Zhu, Chen Liu, Jieyu Li, and Kai Yu. 2019. Semantic parsing with dual learning. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 51–64, Florence, Italy. Association for Computational Linguistics.

Bo Chen, Xianpei Han, Ben He, and Le Sun. 2020. Learning to map frequent phrases to sub-structures of meaning representation for neural semantic parsing. In *Proceedings of the 34th AAAI Conference on Artificial Intelligence*, pages 7546–7553. AAAI Press.

Jianpeng Cheng, Siva Reddy, Vijay Saraswat, and Mirella Lapata. 2019. Learning an executable neural semantic parser. *Computational Linguistics*, 45(1):59–94.

Ann Copestake, Dan Flickinger, Carl Pollard, and Ivan A. Sag. 2005. Minimal recursion semantics: An introduction. *Research on Language & Computation*, 3(4):281–332.

Deborah A. Dahl, Madeleine Bates, Michael Brown, William Fisher, Kate Hunicke-Smith, David Pallett, Christine Pao, Alexander Rudnicky, and Elizabeth Shriberg. 1994. Expanding the scope of the atis task: The atis-3 corpus. In *Proceedings of the Workshop on Human Language Technology*, page 43–48, USA. Association for Computational Linguistics.

Li Dong and Mirella Lapata. 2016. Language to logical form with neural attention. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics*, pages 33–43, Berlin, Germany. Association for Computational Linguistics.

Li Dong and Mirella Lapata. 2018. Coarse-to-fine decoding for neural semantic parsing. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics*, pages 731–742, Melbourne, Australia. Association for Computational Linguistics.

Catherine Finegan-Dollak, Jonathan K. Kummerfeld, Li Zhang, Karthik Ramanathan, Sesh Sadasivam, Rui Zhang, and Dragomir Radev. 2018. Improving text-to-SQL evaluation methodology. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics*, pages 351–360, Melbourne, Australia. Association for Computational Linguistics.

Dan Flickinger, Stephan Oepen, and Emily M. Bender. 2017. *Sustainable Development and Refinement of Complex Linguistic Annotations at Scale*, pages 353–377. Springer Netherlands, Dordrecht.

Matt Gardner, Joel Grus, Mark Neumann, Oyvind Tafjord, Pradeep Dasigi, Nelson F. Liu, Matthew Peters, Michael Schmitz, and Luke Zettlemoyer. 2018. Allennlp: A deep semantic natural language processing platform. In *Proceedings of Workshop for NLP Open Source Software*, pages 1–6, Melbourne, Australia. Association for Computational Linguistics.

Omer Goldman, Veronica Latcinnik, Ehud Nave, Amir Globerson, and Jonathan Berant. 2018. Weakly supervised semantic parsing with abstract examples. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics*, pages 1809–1819, Melbourne, Australia. Association for Computational Linguistics.

Jiaqi Guo, Zecheng Zhan, Yan Gao, Yan Xiao, Jian-Guang Lou, Ting Liu, and Dongmei Zhang. 2019. Towards complex text-to-SQL in cross-domain database with intermediate representation. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 4524–4535, Florence, Italy. Association for Computational Linguistics.

Daniel Hershcovich, Omri Abend, and Ari Rappoport. 2018. Multitask parsing across semantic representations. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics*, pages 373–385, Melbourne, Australia. Association for Computational Linguistics.

Daniel Hershcovich, Omri Abend, and Ari Rappoport. 2019. Content differences in syntactic and semantic representation. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 478–488, Minneapolis, Minnesota. Association for Computational Linguistics.

Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, Jayant Krishnamurthy, and Luke Zettlemoyer. 2017. Learning a neural semantic parser from user feedback. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics*, pages 963–973, Vancouver, Canada. Association for Computational Linguistics.

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

Rohit J. Kate. 2008. Transforming meaning representation grammars to improve semantic parsing. In *Proceedings of the 12th Conference on Computational Natural Language Learning*, pages 33–40, Manchester, England. Coling 2008 Organizing Committee.

Rohit J. Kate and Raymond J. Mooney. 2006. Using string-kernels for learning semantic parsers. In *Proceedings of the 21st International Conference on Computational Linguistics and 44th Annual Meeting of the Association for Computational Linguistics*, pages 913–920, Sydney, Australia. Association for Computational Linguistics.

Rohit J. Kate, Yuk Wah Wong, and Raymond J. Mooney. 2005. Learning to transform natural to formal languages. In *Proceedings of the 20th National Conference on Artificial Intelligence*, page 1062–1068. AAAI Press.

Tom Kwiatkowksi, Luke Zettlemoyer, Sharon Goldwater, and Mark Steedman. 2010. Inducing probabilistic CCG grammars from logical form with higher-order unification. In *Proceedings of the 2010 Conference on Empirical Methods in Natural Language Processing*, pages 1223–1233, Cambridge, MA. Association for Computational Linguistics.

Tom Kwiatkowski, Luke Zettlemoyer, Sharon Goldwater, and Mark Steedman. 2011. Lexical generalization in CCG grammar induction for semantic parsing. In *Proceedings of the 2011 Conference on Empirical Methods in Natural Language Processing*, pages 1512–1523, Edinburgh, Scotland, UK. Association for Computational Linguistics.

Chen Liang, Mohammad Norouzi, Jonathan Berant, Quoc V Le, and Ni Lao. 2018. Memory augmented policy optimization for program synthesis and semantic parsing. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, pages 9994–10006. Curran Associates, Inc.

Percy Liang. 2013. Lambda dependency-based compositional semantics. *arXiv preprint arXiv:1309.4408*.

Percy Liang, Michael Jordan, and Dan Klein. 2011. Learning dependency-based compositional semantics. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, pages 590–599, Portland, Oregon, USA. Association for Computational Linguistics.

Zi Lin and Nianwen Xue. 2019. Parsing meaning representations: Is easier always better? In *Proceedings of the First International Workshop on Designing Meaning Representations*, pages 34–43, Florence, Italy. Association for Computational Linguistics.

Thang Luong, Hieu Pham, and Christopher D. Manning. 2015. Effective approaches to attention-based neural machine translation. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 1412–1421, Lisbon, Portugal. Association for Computational Linguistics.

Yusuke Miyao, Rune Sætre, Kenji Sagae, Takuya Matsuzaki, and Jun'ichi Tsujii. 2008. Task-oriented evaluation of syntactic parsers and their representations. In *Proceedings of the 46th Annual Meeting of the Association for Computation Linguistics: Human Language Technologies*, pages 46–54, Columbus, Ohio. Association for Computational Linguistics.

Thomas Mueller, Francesco Piccinno, Peter Shaw, Massimo Nicosia, and Yasemin Altun. 2019. Answering conversational questions on structured data without logical forms. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing*, pages 5902–5910, Hong Kong, China. Association for Computational Linguistics.

Stephan Oepen, Lilja Øvrelid, Jari Björne, Richard Johansson, Emanuele Lapponi, Filip Ginter, and Erik Velldal. 2017. The 2017 shared task on extrinsic parser evaluation. towards a reusable community infrastructure. In *Proceedings of the 2017 Shared Task on Extrinsic Parser Evaluation at the Fourth International Conference on Dependency Linguistics and* *the 15th International Conference on Parsing Technologies. Pisa, Italy*, pages 1–16.

Panupong Pasupat and Percy Liang. 2015. Compositional semantic parsing on semi-structured tables. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing*, pages 1470–1480, Beijing, China. Association for Computational Linguistics.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. Pytorch: An imperative style, high-performance deep learning library. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, pages 8024–8035. Curran Associates, Inc.

Ana-Maria Popescu, Oren Etzioni, and Henry Kautz. 2003. Towards a theory of natural language interfaces to databases. In *Proceedings of the 8th International Conference on Intelligent User Interfaces*, page 149–157, New York, NY, USA. Association for Computing Machinery.

Maxim Rabinovich, Mitchell Stern, and Dan Klein. 2017. Abstract syntax networks for code generation and semantic parsing. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics*, pages 1139–1149, Vancouver, Canada. Association for Computational Linguistics.

Peter Shaw, Philip Massey, Angelica Chen, Francesco Piccinno, and Yasemin Altun. 2019. Generating logical forms from graph representations of text and entities. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 95–106, Florence, Italy. Association for Computational Linguistics.

Zeyu Sun, Qihao Zhu, Yingfei Xiong, Yican Sun, Lili Mou, and Lu Zhang. 2020. Treegen: A tree-based transformer architecture for code generation. In *Proceedings of the 34th AAAI Conference on Artificial Intelligence*, pages 8984–8991. AAAI Press.

Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. 2014. Sequence to sequence learning with neural networks. In *Proceedings of the 27th International Conference on Neural Information Processing Systems*, page 3104–3112, Cambridge, MA, USA. MIT Press.

Adrienne Wang, Tom Kwiatkowski, and Luke Zettlemoyer. 2014. Morpho-syntactic lexical generalization for CCG semantic parsing. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing*, pages 1284–1295, Doha, Qatar. Association for Computational Linguistics.

Bailin Wang, Richard Shin, Xiaodong Liu, Oleksandr Polozov, and Matthew Richardson. 2020. RAT-SQL: Relation-aware schema encoding and linking for text-to-SQL parsers. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 7567–7578, Online. Association for Computational Linguistics.

Tomer Wolfson, Mor Geva, Ankit Gupta, Matt Gardner, Y. Goldberg, D. Deutch, and Jonathan Berant. 2020. Break it down: A question understanding benchmark. *Transactions of the Association for Computational Linguistics*, 8:183–198.

Yuk Wah Wong and Raymond Mooney. 2006. Learning for semantic parsing with statistical machine translation. In *Proceedings of the Human Language Technology Conference of the NAACL, Main Conference*, pages 439–446, New York City, USA. Association for Computational Linguistics.

Yuk Wah Wong and Raymond Mooney. 2007. Learning synchronous grammars for semantic parsing with lambda calculus. In *Proceedings of the 45th Annual Meeting of the Association of Computational Linguistics*, pages 960–967, Prague, Czech Republic. Association for Computational Linguistics.

Pengcheng Yin and Graham Neubig. 2017. A syntactic neural model for general-purpose code generation. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics*, pages 440–450, Vancouver, Canada. Association for Computational Linguistics.

Pengcheng Yin and Graham Neubig. 2018. TRANX: A transition-based neural abstract syntax parser for semantic parsing and code generation. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 7–12, Brussels, Belgium. Association for Computational Linguistics.

John M. Zelle and Raymond J. Mooney. 1996. Learning to parse database queries using inductive logic programming. In *Proceedings of the 13th National Conference on Artificial Intelligence*, page 1050–1055. AAAI Press.

Luke Zettlemoyer and Michael Collins. 2007. Online learning of relaxed CCG grammars for parsing to logical form. In *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*, pages 678–687, Prague, Czech Republic. Association for Computational Linguistics.

Luke S. Zettlemoyer and Michael Collins. 2005. Learning to map sentences to logical form: Structured classification with probabilistic categorial grammars. In *Proceedings of the Twenty-First Conference on Uncertainty in Artificial Intelligence*, page 658–666, Arlington, Virginia, USA. AUAI Press.

Kai Zhao and Liang Huang. 2015. Type-driven incremental semantic parsing with polymorphism. In *Proceedings of the 2015 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 1416–1421, Denver, Colorado. Association for Computational Linguistics.

Zexuan Zhong, Jiaqi Guo, Wei Yang, Jian Peng, Tao Xie, Jian-Guang Lou, Ting Liu, and Dongmei Zhang. 2018. SemRegex: A semantics-based approach for generating regular expressions from natural language specifications. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 1608–1618, Brussels, Belgium. Association for Computational Linguistics.

A Supplemental Material

Algorithm 1: Translation of Lambda Calculus Logical Forms to FunQL

```
Input: A Lambda Calculus logical form $p$
Output: A FunQL logical form $f$
$types$ = InferTypes($p$);
$tokens$ = Tokenize($p$);
$expressions, stack$ = [], [];
$i = 0$;
for $t \in tokens$ do
    if $t$ == '(' then
        Append($stack$,$i$);
    else if $t$ == ')' then
        $j =$ = Pop($stack$);
        $e$ = Search($expressions$, $j$, $i$);
        Remove($e$);
        $ne$ = Translate($tokens[j : i]$, $e$,
          $types$);
        Append($expressions$, $ne$);
    $i+ = 1$;
end
$f = expressions[0]$;
```

A.1 Details of Benchmark Construction

Geo. Since the four MRs we study have annotated logical forms and have execution engines except for Lambda Calculus in Geo, we directly use them (SQL taken from (Finegan-Dollak et al., 2018), Prolog taken from (Jia and Liang, 2016), FunQL taken from (Wong and Mooney, 2006), and Lambda Calculus taken from (Kwiatkowksi et al., 2010)). We implement an execution engine with Haskell for Lambda Calculus. With these resources, we cross-validate the correctness of annotations and execution engines by comparing the execution results of logical forms. As a result, we found nearly 30 Prolog logical forms with annotation mistakes and two bugs in the execution engines of Prolog and FunQL.

ATIS. There are only annotated logical forms for Lambda Calculus and SQL in ATIS. We directly use Lambda Calculus logical forms provided by (Jia and Liang, 2016) and SQL queries provided by (Iyer et al., 2017). To provide annotations for FunQL and Prolog, we semi-automatically translate Lambda Calculus logical forms to equivalent logical forms in FunQL and Prolog. Algorithm 1 presents the pseudo code for translating a Lambda Calculus logical form to FunQL. Basically, we first

Approach	MR	Exact	Exec
Geo			
Kwiatkowksi et al. (2010)	FunQL	84.3	-
Shaw et al. (2019)*	FunQL	89.3	-
Jia and Liang (2016)*	Prolog	-	89.3
- data augmentation	Prolog	-	85.0
Chen et al. (2020)*	Prolog	-	89.6
Kwiatkowksi et al. (2010)	λ	87.9	-
Wang et al. (2014)	λ	90.4	-
Zhao and Huang (2015)	λ	88.9	-
Dong and Lapata (2016)*†	λ	87.1	-
Rabinovich et al. (2017)*†	λ	87.1	-
Dong and Lapata (2018)*†	λ	88.2	-
Sun et al. (2020)*†	λ	89.1	-
Iyer et al. (2017)*†	SQL	-	82.5
ATIS			
Wang et al. (2014)	λ	91.3	-
Zhao and Huang (2015)	λ	84.2	-
Jia and Liang (2016)*	λ	83.3	-
- data augmentation	λ	76.3	-
Dong and Lapata (2016)*†	λ	84.6	-
Rabinovich et al. (2017)*†	λ	85.9	-
Yin and Neubig (2018)*†	λ	88.2	-
Dong and Lapata (2018)*†	λ	87.7	-
Cao et al. (2019)*†	λ	89.1	-
Sun et al. (2020)*†	λ	89.6	-
Shaw et al. (2019)*	λ	87.1	-
Iyer et al. (2017)*†	SQL	-	79.2
Jobs			
Zettlemoyer and Collins (2005)	λ	79.3	-
Zhao and Huang (2015)	λ	85.0	-
Dong and Lapata (2016)*†	Prolog	90.0	-
Rabinovich et al. (2017)*†	Prolog	91.4	-
Chen et al. (2020)*	Prolog	-	92.1

Table 8: Performance of semantic parsing approaches, where $*$ denotes neural approaches and $\dagger$ denotes approaches using data anonymization techniques. 'λ' denotes Lambda Calculus. 'Exact' and 'Exec' denote exact-match and execution-match, respectively.

perform a type inference for variables in the logical form, since some predicates in Lambda Calculus can take different types of variables as input, e.g., for the predicate `to(A,B)`, variable `B` can be either an airport or a city. Then, we tokenize the input logical form and recursively translate each sub-expression in the logical form into FunQL expressions based on the predicates we design for FunQL. Translation from Lambda Calculus to Prolog is performed in a similar way. The source code for translation is also available in our Github repository. We also implement execution engines for Lambda Calculus, Prolog, and FunQL.

Job. There are only annotated logical forms for Lambda Calculus and Prolog in Job. We directly use Prolog logical forms provided on the website.[9] To provide annotations for FunQL, we semi-automatically translate Prolog logical forms to equivalent logical forms in FunQL with an algorithm similar to Algorithm 1. In terms of SQL, the logical forms in Job are relatively simple and can be expressed with the `SELECT`, `FROM` and `WHERE` clauses of SQL. We simply translate each sub-expression in Prolog to an expression in `WHERE` clause and use a conjunction to join the resulting expressions. Since we cannot find the annotated Lambda Calculus logical forms provided by (Zettlemoyer and Collins, 2005), we also translate the Prolog logical forms to Lambda Calculus. We implement execution engines for Lambda Calculus and FunQL.

A.2 Model Configuration

Preprocessing of Prolog and Lambda Calculus As Prolog and Lambda Calculus have variables, we need to standardize their variable naming before training. Following (Jia and Liang, 2016), we preprocess the Prolog logical forms to the De Brujin index notation. We standardize the variable naming of Lambda Calculus based on their occurrence order in logical forms.

Attention Copying Mechanism. In Geo and Job, we use the standard copy mechanism, i.e., directly copying a source word to a logical form. In ATIS, following (Jia and Liang, 2016), we leverage an external lexicon to identify potential copy candidates, e.g., `slc:ap` can be identified as a potential entity for description "salt lake city airport" in utterance. When we copy a source word that is part of a phrase in the lexicon, we write the entity associated with that lexicon entry to a logical form.

Hyper-Parameters. For the seq2seq model, the embedding dimension of both source and target languages ranges over $\{100, 200\}$. We select a one-layer bi-directional LSTM as an encoder. The hidden dimension of the encoder ranges over $\{32, 64, 128, 256\}$. Similarly, a one-layer LSTM is selected as the decoder. Its hidden dimension is as same as the encoder. In terms of attention, we select bi-linear as the activation function, where the hidden dimension is 2 times that of the encoder. We employ dropout at training time with rate ranging over $\{0.1, 0.2, 0.3\}$. We select batch size from $\{16, 32, 48, 64\}$, and select learning rate from $\{0.001, 0.0025, 0.005, 0.01, 0.025, 0.05\}$. Following (Dong and Lapata, 2016), we use the RMSProp algorithm to update the parameters. The smoothing constant of RMSProp is 0.95. We initialize all

[9] http://www.cs.utexas.edu/ ml/nldata/jobquery.html

parameters uniformly at random within the interval $[-0.1, 0.1]$.

Similarly, for the grammar-based model, a one-layer bi-directional LSTM is used as an encoder and another LSTM is employed as a decoder. The layers of the decoder is selected from $\{1, 2\}$. The hidden dimension of the encoder ranges over {64, 128, 256}. The hidden dimension of the decoder is 2 times that of the encoder. The hidden dimension of both the grammar rule and non-terminal is selected from {64, 128, 256}. We also employ dropout in the encoder and decoder at training time with rate selected from {0.1, 0.2, 0.3}. We select batch size from {16, 32, 48, 64}, and select learning rate from {0.001, 0.0025, 0.005, 0.01, 0.025, 0.05}. We use the Adam algorithm to update the parameters.

For both models, gradients are clipped at 5 to alleviate the exploding gradient problem, and early stopping is used to determine the number of epochs. We provide the detailed configurations of the NNI platform in our Github repository.

A.3 Search for Aliases for Logical Forms.

Algorithm 2: Search for Program Alias

```
Input: A logical form $p$
Output: A set of aliases of the input logical
        form $aliases$
$ast$ = Parse($p$);
$aliases$ = [];
for $r \in Rules$ do
    $C$ = Apply($ast$, $r$);
    for $c \in C$ do
        if IsEquivalent($p$, $c$) then
            Append($aliases$,$c$);
    end
end
```

Algorithm 2 presents the way we search for aliases for a logical form. Transformation rules can be categorized into two groups based on whether they are domain-specific. Considering the following two logical forms:

(lambda A:e (exists B (and (flight B) (fare B A))))

(lambda A:e (exists B (and (flight B) (equals (fare B) A))))

they are semantically equivalent due to the multiple meaning definitions of `fare`. There are also domain-general transformation rules, e.g., permuting the expressions in the conjunction predicate:

(lambda A:e (exists B (and (flight B) (fare B A))))

(lambda A:e (exists B (and (fare B A) (flight B))))

Rule	Description
Prolog	
Shuffle	Shuffle expressions in conjunction
Remove	Remove redundant expressions
Merge	Put expressions into higher-order predicates
Lambda Calculus	
Shuffle	Shuffle expressions in conjunction
Replace	Replace semantically equivalent predicates
FunQL	
Shuffle	Shuffle expressions in conjunction
Remove	Remove redundant expressions
Swap	Swap unit relation predicate
Replace	Replace semantically equivalent predicates
SQL	
Shuffle	Shuffle expressions in Select, From, Where, and Having clauses
Argmax	Express Argmax/min with OrderBy and Limit clause instead of subquery
In2Join	Replace In clause with Join clause

Table 9: Transformation rules for Prolog, Lambda Calculus and FunQL.

In this work, we primarily consider domain-general transformation rules and only when there is limited aliases found by domain-general rules, we use domain-specific rules. Table 9 presents the transformation rules we used in Geo domains. Rules in ATIS are similar. We provide examples below to illustrate the rules.

Prolog Shuffle:

answer(A,(area(B,A),const(B,stateid(Texas))))

answer(A,(const(B,stateid(texas)),area(B,A),))

Prolog Remove:

answer(A,(loc(B,A),city(B),const(B,cityid(Austin,_))))

answer(A,(loc(B,A),const(B,cityid(Austin,_))))

Prolog Merge:

answer(A,(state(A),loc(B,A),highest(B,place(B))))

answer(A,(highest(B,(place(B),loc(B,A),state(A)))))

FunQL Swap:

answer(count(major(city(loc_2(stateid(Texas))))))

answer(count(city(major(loc_2(stateid(Texas))))))

FunQL Replace:

answer(city(loc_2(largest(state(all)))))

answer(city(loc_2(largest_one(area_1(state(all))))))

SQL Argmax:

SELECT city.city_name FROM city WHERE city.population = (SELECT MAX(c1.population) FROM city as c1 WHERE c1.state_name = 'arizona') and city.state_name = 'arizona';

MR	Geo	ATIS
Prolog	98.0%	96.2%
Lambda Calculus	69.7%	95.3%
FunQL	42.5%	87.9%
SQL	38.5%	95.5%

Table 10: Number of logical forms in training set of Geo and ATIS that have aliases.

SELECT city.city_name FROM city WHERE state_name = 'arizona' ORDER BY city.population DESC LIMIT 1;

SQL In2Join:

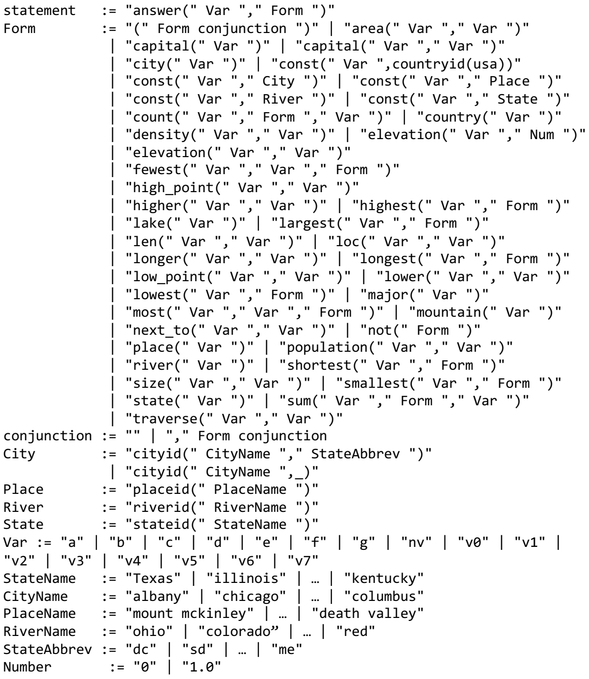
SELECT river.river_name FROM river WHERE river.traverse IN (SELECT state.state_name FROM state WHERE state.area = (SELECT MAX(s1.area) FROM state as s1));

SELECT river.river_name FROM river, state WHERE river.traverse = state.state_name AND state.area = (SELECT MAX(s1.area) FRORM state as s1);

Table 10 presents the number of logical forms in the training set of Geo and ATIS that have aliases, from which we can see that with these simple rules, we are able to find a lot of aliases. Since the logical forms in Geo are generally simpler than that of ATIS, the number of Geo is lower than ATIS.

A.4 Grammar

We present the grammar rules used in our experiments for Geo (Figure 4-11). The grammar rules for different MRs in ATIS follow a similar definition. It is important to note that although all logical forms in an MR can be modeled by both G1 and G2, they are not necessarily equivalent in expressiveness. For example, `answer(largest(place(all)))` is a syntactically correct but semantically incorrect FunQL logical form in Geo, because the predicate `largest` is not allowed to take places as input. However, G2 of FunQL still can accept this logical form, while G1 cannot, because G1 explicitly encodes type constraints in the its rules. Nevertheless, the performance of G1 is lower than G2 in both Geo and ATIS domains, which is surprising. For all the grammar rules we present, G1 and G2 for Prolog and Lambda Calculus and equivalent. G1 and G2 for FunQL and SQL are not equivalent.



```
statement   := "answer(" Var "," Form ")"
Form        := "(" Form conjunction ")" | "area(" Var "," Var ")"
             | "capital(" Var ")" | "capital(" Var "," Var ")"
             | "city(" Var ")" | "const(" Var ",countryid(usa))"
             | "const(" Var "," City ")" | "const(" Var "," Place ")"
             | "const(" Var "," River ")" | "const(" Var "," State ")"
             | "count(" Var "," Form "," Var ")" | "country(" Var ")"
             | "density(" Var "," Var ")" | "elevation(" Var "," Num ")"
             | "elevation(" Var "," Var ")"
             | "fewest(" Var "," Var "," Form ")"
             | "high_point(" Var "," Var ")"
             | "higher(" Var "," Var ")" | "highest(" Var "," Form ")"
             | "lake(" Var ")" | "largest(" Var "," Form ")"
             | "len(" Var "," Var ")" | "loc(" Var "," Var ")"
             | "longer(" Var "," Var ")" | "longest(" Var "," Form ")"
             | "low_point(" Var "," Var ")" | "lower(" Var "," Var ")"
             | "lowest(" Var "," Form ")" | "major(" Var ")"
             | "most(" Var "," Var "," Form ")" | "mountain(" Var ")"
             | "next_to(" Var "," Var ")" | "not(" Form ")"
             | "place(" Var ")" | "population(" Var "," Var ")"
             | "river(" Var ")" | "shortest(" Var "," Form ")"
             | "size(" Var "," Var ")" | "smallest(" Var "," Form ")"
             | "state(" Var ")" | "sum(" Var "," Form "," Var ")"
             | "traverse(" Var "," Var ")"
conjunction := "" | "," Form conjunction
City        := "cityid(" CityName "," StateAbbrev ")"
             | "cityid(" CityName ",_)"
Place       := "placeid(" PlaceName ")"
River       := "riverid(" RiverName ")"
State       := "stateid(" StateName ")"
Var := "a" | "b" | "c" | "d" | "e" | "f" | "g" | "nv" | "v0" | "v1" |
"v2" | "v3" | "v4" | "v5" | "v6" | "v7"
StateName   := "Texas" | "illinois" | … | "kentucky"
CityName    := "albany" | "chicago" | … | "columbus"
PlaceName   := "mount mckinley" | … | "death valley"
RiverName   := "ohio" | "colorado” | … | "red"
StateAbbrev := "dc" | "sd" | … | "me"
Number       := "0" | "1.0"
```

Figure 4: The grammar rules for Prolog in Geo induced by (Wong and Mooney, 2007) (G1).

```
statement     := "answer(" var "," goal ")"
goal          := "(" predicate conjunction ")" | meta | unit_relation
conjunction   := "" | "," predicate conjunction
predicate     := "not" declaration | "not((" predicate conjunction "))"
               | binary_relation | declaration | meta | unit_relation
meta          := count | fewest | highest | largest | longest | lowest
               | most | shortest | smallest | sum
unit_relation := is_capital | is_city | is_lake | is_major
               | is_mountain | is_place | is_river | is_state
binary_relation := is_area | is_captial_of | is_density | is_elevation
               | is_equal | is_high_point | is_higher | is_len
               | is_located_in | is_longer | is_low_point | is_lower
               | is_next_to | is_population | is_size | is_traverse
declaration   := "const(" var "," object ")"
object        := "countryid(usa)" | city | place | river | state
retrieve      := area | len | population
var           := "a" | "b" | "c" | "d" | "e" | "f" | "g" | "nv" | "v0"
               | "v1" | "v2" | "v3" | "v4" | "v5" | "v6" | "v7"
is_capital    := "capital(" var ")"
is_captial_of := "capital(" var "," var ")"
is_city       := "city(" var ")"
is_density    := "density(" var "," var ")"
is_elevation  := "elevation(" var "," literal ")"
               | "elevation(" var "," var ")"
is_equal      := "equal(" var "," var ")"
is_high_point := "high_point(" var "," var ")"
is_higher     := "higher(" var "," var ")"
is_lake       := "lake(" var ")"
is_len        := "len(" var "," var ")"
is_located_in := "loc(" var "," var ")"
is_longer     := "longer(" var "," var ")"
is_low_point  := "low_point(" var "," var ")"
is_lower      := "lower(" var "," var ")"
is_major      := "major(" var ")"
is_mountain   := "mountain(" var ")"
is_next_to    := "next_to(" var "," var ")"
is_place      := "place(" var ")"
is_population := "population(" var "," var ")"
is_river      := "river(" var ")"
is_size       := "size(" var "," var ")"
is_state      := "state(" var ")"
is_traverse   := "traverse(" var "," var ")"
largest       := "largest(" var "," goal ")"
len           := "len(" var ")"
longest       := "longest(" var "," goal ")"
lowest        := "lowest(" var "," goal ")"
most          := "most(" var "," var "," goal ")"
shortest      := "shortest(" var "," goal ")"
smallest      := "smallest(" var "," goal ")"
statement     := "answer(" var "," goal ")"
sum           := "sum(" var "," goal "," var ")"
count         := "count(" var "," goal "," var ")"
fewest        := "fewest(" var "," var "," goal ")"
highest       := "highest(" var "," goal ")”
state         := "stateid(" state_name ")"
river         := "riverid(" river_name ")"
city          := "cityid(" city_name "," state_abbre ")"
place         := "placeid(" place_name ")"
river         := "riverid(" river_name ")"
state_name    := "Texas" | "illinois" | … | "kentucky"
city_name     := "albany" | "chicago" | … | "columbus"
place_name    := "mount mckinley" | … | "death valley"
river_name    := "ohio" | "colorado” | … | "red"
state_bbbrev  := "dc" | "sd" | … | "me"
literal       := "0" | "1.0"
```

Figure 5: The grammar rules that we induce for Prolog in Geo (G2).

```
statement   := expression
expression  := abstraction | application | constant | variable
abstraction := "(lambda" variable_definition expression ")"
application := "(" function ")"
constant    := "0:i" | "death_valley:lo" | "usa:co" | city
             | mountain | names | place | river | state
variable    := "$0" | "$1" | "$2" | "$3" | "$4"
polyvariadic_expression := "" | application polyvariadic_expression
variable_definition     := "$0:e" | "$0:i" | "$1:e" | "$2:e"
                         | "$3:e" | "$4:e"
function    := "<:<i,<i,t>>" expression expression
             | "=:<i,<i,t>>" expression expression
             | ">:<i,<i,t>>" expression expression
             | "and:<t*,t>" application polyvariadic_expression
             | "area:<lo,i>" expression
             | "argmax:<<e,t>,<<e,i>,e>>" expression expression
             | "argmin:<<e,t>,<<e,i>,e>>" expression expression
             | "capital2:<s,<c,t>>" expression expression
             | "capital:<c,t>" expression
             | "capital:<s,<c,t>>" expression expression
             | "capital:<s,c>" expression | "city:<c,t>" expression
             | "count:<<e,t>,i>" expression
             | "density:<lo,<i,t>>" expression expression
             | "density:<lo,i>" expression
             | "elevation:<lo,<i,t>>" expression expression
             | "elevation:<lo,i>" expression
             | "equals:<e,<e,t>>" expression expression
             | "exists:<<e,t>,t>" expression
             | "forall:<<e,t>,t>" expression
             | "high_point:<e,<e,t>>" expression expression
             | "high_point:<e,l>" expression
             | "in:<lo,<lo,t>>" expression expression
             | "lake:<l,t>" expression | "len:<r,i>" expression
             | "loc:<lo,<lo,t>>" expression expression
             | "major:<lo,t>" expression | "mountain:<m,t>" expression
             | "named:<e,<n,t>>" expression expression
             | "next_to:<lo,<lo,t>>" expression expression
             | "not:<t,t>" expression
             | "or:<t*,t>" application polyvariadic_expression
             | "place:<p,t>" expression
             | "population:<lo,<i,t>>" expression expression
             | "population:<lo,i>" expression
             | "river:<r,t>" expression | "size:<lo,i>" expression
             | "state:<s,t>" expression
             | "sum:<<e,t>,<<e,i>,i>>" expression expression
             | "the:<<e,t>,e>" expression
             | "town:<lo,t>" expression
state       := "oklahoma:s" | "mississippi:s" | … | "arkansas"
city        := "albany_ny:c" | "chicago_il:c" | … | "columbus_oh:c"
place       := "mount_mckinley:p" | "mount_whitney:p"
river       := "mississippi_river:r" | … | "colorado_river:r"
mountain    := "mount_mckinley:m" | … | "mount_whitney:m"
name        := "austin:n" | … | "springfield:n"
```

Figure 6: The grammar rules that for Lambda Calculus in Geo (G1).

```
statement   := expression
expression  := abstraction | application | constant | variable
abstraction := "(lambda" variable_definition expression ")"
application := "(" function ")"
constant    := "0:i" | "death_valley:lo" | "usa:co" | city
              | mountain | names | place | river | state
variable    := "$0" | "$1" | "$2" | "$3" | "$4"
polyvariadic_expression := "" | application polyvariadic_expression
variable_definition     := "$0:e" | "$0:i" | "$1:e" | "$2:e"
                         | "$3:e" | "$4:e"
function    := binary_relation | entity_function | meta_predicate
              | unit_relation
binary_relation := "capital2:<s,<c,t>>" expression expression
              | "capital:<s,<c,t>>" variable variable
              | "density:<lo,<i,t>>" expression expression
              | "elevation:<lo,<i,t>>" expression expression
              | "high_point:<e,<e,t>>" variable variable
              | "in:<lo,<lo,t>>" expression expression
              | "loc:<lo,<lo,t>>" expression expression
              | "named:<e,<n,t>>" expression expression
              | "next_to:<lo,<lo,t>>" expression expression
              | "population:<lo,<i,t>>" variable variable
entity_function := "area:<lo,i>" expression | "capital:<s,c>" expression
              | "density:<lo,i>" expression
              | "elevation:<lo,i>" expression
              | "high_point:<e,l>" expression
              | "len:<r,i>" expression
              | "population:<lo,i>" expression
              | "size:<lo,i>" expression | "the:<<e,t>,e>" expression
meta_predicate  := "<:<i,<i,t>>" expression expression
              | "=:<i,<i,t>>" expression expression
              | ">:<i,<i,t>>" expression expression
              | "and:<t*,t>" application polyvariadic_expression
              | "argmax:<<e,t>,<<e,i>,e>>" abstraction abstraction
              | "argmin:<<e,t>,<<e,i>,e>>" abstraction abstraction
              | "count:<<e,t>,i>" abstraction
              | "equals:<e,<e,t>>" variable expression
              | "exists:<<e,t>,t>" abstraction
              | "forall:<<e,t>,t>" abstraction
              | "not:<t,t>" application
              | "or:<t*,t>" application polyvariadic_expression
              | "sum:<<e,t>,<<e,i>,i>>" abstraction abstraction
unit_relation   := "capital:<c,t>" variable | "city:<c,t>" variable
              | "lake:<l,t>" variable | "major:<lo,t>" variable
              | "mountain:<m,t>" variable | "place:<p,t>" variable
              | "river:<r,t>" variable | "state:<s,t>" variable
              | "town:<lo,t>" variable
state       := "oklahoma:s" | "mississippi:s" | … | "arkansas"
city        := "albany_ny:c" | "chicago_il:c" | … | "columbus_oh:c"
place       := "mount_mckinley:p" | "mount_whitney:p"
river       := "mississippi_river:r" | … | "colorado_river:r"
mountain    := "mount_mckinley:m" | … | "mount_whitney:m"
name        := "austin:n" | … | "springfield:n"
```

Figure 7: The grammar rules that for Lambda Calculus in Geo (G2).

```
statement := Query
Query     := "answer(" City ")" | "answer(" Country ")"
            | "answer(" Num ")" | "answer(" Place ")"
            | "answer(" River ")" | "answer(" State ")"
City      := "capital(" City ")" | "capital(" Place ")" | "capital(all)"
            | "capital_1(" Country ")" | "capital_1(" State ")"
            | "city(" City ")" | "city(all)"
            | "cityid(" CityName "," StateAbbrev ")"
            | "cityid(" CityName ",_)"
            | "each(" City ")" | "exclude(" City "," City ")"
            | "fewest(" City ")" | "intersection(" City "," City ")"
            | "largest(" City ")" | "largest_one(density_1(" City "))"
            | "largest_one(population_1(" City "))"
            | "loc_1(" Place ")" | "loc_2(" Country ")"
            | "loc_2(" State ")" | "major(" City ")"
            | "most(" City ")" | "smallest(" City ")"
            | "smallest_one(population_1(" City "))"
            | "traverse_1(" River ")"
Country   := "country(all)" | "countryid('usa')" | "each(" Country ")"
            | "exclude(" Country "," Country ")"
            | "intersection(" Country "," Country ")"
            | "largest(" Country ")" | "loc_1(" City ")"
            | "loc_1(" Place ")" | "loc_1(" River ")"
            | "loc_1(" State ")" | "most(" Country ")"
            | "smallest(" Country ")" | "traverse_1(" River ")"
Num       := "area_1(" City ")" | "area_1(" Country ")"
            | "area_1(" Place ")" | "area_1(" State ")"
            | "count(" City ")" | "count(" Country ")"
            | "count(" Place ")" | "count(" River ")"
            | "count(" State ")" | "density_1(" City ")"
            | "density_1(" Country ")" | "density_1(" State ")"
            | "elevation_1(" Place ")" | "len(" River ")"
            | "population_1(" City ")" | "population_1(" Country ")"
            | "population_1(" State ")" | "size(" City ")"
            | "size(" Country ")" | "size(" State ")"
            | "smallest(" Num ")" | "sum(" Num ")" | Digit
Place     := "each(" Place ")" | "elevation_2(" Num ")"
            | "exclude(" Place "," Place ")" | "fewest(" Place ")"
            | "high_point_1(" State ")" | "higher_1(" Place ")"
            | "higher_2(" Place ")" | "highest(" Place ")"
            | "intersection(" Place "," Place ")" | "lake(" Place ")"
            | "lake(all)" | "largest(" Place ")" | "loc_2(" City ")"
            | "loc_2(" Country ")" | "loc_2(" State ")"
            | "low_point_1(" State ")" | "lower_1(" Place ")"
            | "lower_2(" Place ")" | "lowest(" Place ")"
            | "major(" Place ")" | "mountain(" Place ")"
            | "mountain(all)" | "place(" Place ")"
            | "place(all)" | "placeid(" PlaceName ")"
            | "smallest(" Place ")"
River     := "each(" River ")" | "exclude(" River "," River ")"
            | "fewest(" River ")" | "intersection(" River "," River ")"
            | "loc_2(" Country ")" | "loc_2(" State ")"
            | "longer(" River ")" | "longest(" River ")"
            | "major(" River ")" | "most(" State ")"
            | "river(" River ")" | "river(all)"
            | "riverid(" RiverName ")" | "shortest(" River ")"
            | "traverse_2(" City ")" | "traverse_2(" Country ")"
            | "traverse_2(" State ")"
State     := "capital_2(" City ")" | "each(" State ")"
            | "exclude(" State "," State ")" | "fewest(" State ")"
            | "high_point_2(" Place ")"
            | "intersection(" State "," State ")"
            | "largest(" State ")" | "largest_one(area_1(" State "))"
            | "largest_one(density_1(" State "))"
            | "largest_one(population_1(" State "))"
            | "loc_1(" City ")" | "loc_1(" Place ")"
            | "loc_1(" River ")" | "loc_2(" Country ")"
            | "low_point_2(" Place ")" | "most(" City ")"
            | "most(" Place ")" | "most(" River ")"
            | "most(" State ")" | "next_to_1(" State ")"
            | "next_to_2(" River ")" | "next_to_2(" State ")”
            | "smallest(" State ")" | "smallest_one(area_1(" State "))"
            | "smallest_one(density_1(" State "))"
            | "smallest_one(population_1(" State "))"
            | "state(" State ")" | "state(all)"
            | "stateid(" StateName ")" | "traverse_1(" River ")"
StateName   := "Texas" | "illinois" | … | "kentucky"
CityName    := "albany" | "chicago" | … | "columbus"
PlaceName   := "mount mckinley" | … | "death valley"
RiverName   := "ohio" | "colorado” | … | "red"
StateAbbrev := "dc" | "sd" | … | "me"
Digit       := "0" | "1.0"
```

Figure 8: The grammar rules for FunQL in Geo induced by (Wong and Mooney, 2006) (G1).

```
answer     := "answer(" predicate ")"
predicate  := "exclude(" predicate "," predicate ")"
            | "intersection(" predicate "," predicate ")"
            | collection | meta | object | relation
collection := all_capital_cities | all_cities | all_lakes
            | all_mountains | all_places | all_rivers | all_states
meta       := count | fewest | highest | largest | largest_one_area
            | largest_one_density | largest_one_population
            | longest | lowest | most | shortest | smallest
            | smallest_one_area | smallest_one_density
            | smallest_one_population | sum
object     := "countryid('usa')" | city | place | river | state
relation   := is_area_state | is_captial | is_captial_city
            | is_captial_country | is_city | is_density_place
            | is_elevation_place | is_elevation_value
            | is_high_point_place | is_high_point_state
            | is_higher_place_2 | is_lake | is_len | is_loc_x
            | is_loc_y | is_longer | is_low_point_place
            | is_low_point_state | is_lower_place_2 | is_major
            | is_mountain | is_next_to_state_1 | is_next_to_state_2
            | is_place | is_population | is_river | is_size
            | is_state | is_traverse_river | is_traverse_state
all_capital_cities := "capital(all)"
all_cities         := "city(all)"
all_lakes          := "late(all)"
all_mountains      := "mountain(all)"
all_places         := "place(all)"
all_rivers         := "river(all)"
all_states         := "state(all)"
count              := "count(" predicate ")"
fewest             := "fewest(" predicate ")"
highest            := "highest(" predicate ")"
largest            := "largest(" predicate ")"
largest_one_area   := "largest_one(area_1(" predicate "))"
largest_one_density     := "largest_one(density_1(" predicate "))"
largest_one_population  := "largest_one(population_1(" predicate "))"
longest            := "longest(" predicate ")"
lowest             := "lowest(" predicate ")"
most               := "most(" predicate ")”
shortest           := "shortest(" predicate ")"
smallest           := "smallest(" predicate ")"
smallest_one_area  := "smallest_one(area_1(" predicate "))"
smallest_one_density    := "smallest_one(density_1(" predicate "))"
smallest_one_population := "smallest_one(population_1(" predicate "))"
sum                := "sum(" predicate ")"
city               := "cityid(" city_name "," state_abbre ")"
state              := "stateid(" state_name ")"
place              := "placeid(" place_name ")"
river              := "riverid(" river_name ")"
is_area_state      := "area_1(" predicate ")"
is_captial         := "capital(" predicate ")"
is_captial_city    := "capital_2(" predicate ")"
is_captial_country := "capital_1(" predicate ")"
is_city            := "city(" predicate ")"
is_density_place   := "density_1(" predicate ")"
is_elevation_place := "elevation_1(" predicate ")"
is_elevation_value := "elevation_2(" number ")"
is_high_point_place:= "high_point_2(" predicate ")"
is_high_point_state:= "high_point_1(" predicate ")"
is_higher_place_2  := "higher_2(" predicate ")"
is_lake            := "lake(" predicate ")"
is_len             := "len(" predicate ")"
is_loc_x           := "loc_1(" predicate ")"
is_loc_y           := "loc_2(" predicate ")"
is_longer          := "longer(" predicate ")"
is_low_point_place := "low_point_2(" predicate ")"
is_low_point_state := "low_point_1(" predicate ")"
is_lower_place_2   := "lower_2(" predicate ")"
is_major           := "major(" predicate ")"
is_mountain        := "mountain(" predicate ")"
is_next_to_state_1 := "next_to_1(" predicate ")"
is_next_to_state_2 := "next_to_2(" predicate ")"
is_place           := "place(" predicate ")"
is_population      := "population_1(" predicate ")"
is_river           := "river(" predicate ")"
is_size            := "size(" predicate ")"
is_state           := "state(" predicate ")"
is_traverse_river  := "traverse_1(" predicate ")"
is_traverse_state  := "traverse_2(" predicate ")"
state_name         := "Texas" | "illinois" | … | "kentucky"
city_name          := "albany" | "chicago" | … | "columbus"
place_name         := "mount mckinley" | … | "death valley"
river_name         := "ohio" | "colorado” | … | "red"
state_abbrev       := "dc" | "sd" | … | "me"
number             := "0" | "1.0"
```

Figure 9: The grammar rules that we induce for FunQL in Geo (G2).

```
statement  := mquery
mquery     := select_clause from_clause groupby_clause having_clause orderby_clause limit
            | select_clause from_clause groupby_clause having_clause orderby_clause
            | select_clause from_clause groupby_clause having_clause
            | select_clause from_clause groupby_clause orderby_clause limit
            | select_clause from_clause groupby_clause orderby_clause
            | select_clause from_clause groupby_clause
            | select_clause from_clause orderby_clause limit
            | select_clause from_clause orderby_clause
            | select_clause from_clause where_clause groupby_clause having_clause orderby_clause limit
            | select_clause from_clause where_clause groupby_clause having_clause orderby_clause
            | select_clause from_clause where_clause groupby_clause having_clause
            | select_clause from_clause where_clause groupby_clause orderby_clause limit
            | select_clause from_clause where_clause groupby_clause orderby_clause
            | select_clause from_clause where_clause groupby_clause
            | select_clause from_clause where_clause orderby_clause limit
            | select_clause from_clause where_clause orderby_clause
            | select_clause from_clause where_clause
            | select_clause from_clause
select_clause        := select_with_distinct select_results
select_with_distinct := "select distinct" | "select"
select_results       := select_result "," select_results | select_result
select_result        := subject "as" column_alias | subject selectop subject | subject
selectop             := "+" | "-" | "/"
subject              := col_ref | function
col_ref              := column_name | table_alias "." column_name
function             := fname "(distinct" col_ref ")" | fname "(" col_ref ")"
table_alias          := "border_infoalias0" | … | "statealias3" | "statealias4" | "statealias5" | "tmp"
table_name           := "border_info" | "city" | "highlow" | "lake" | "mountain" | "river" | "state"
column_alias         := "derived_fieldalias0" | "derived_fieldalias1"
column_name          := "*" | "area" | "border" | "capital" | … | "population" | "river_name" | "state_name"
from_clause          := "from" source | "from" table_source join_clauses
source               := single_source "," source | single_source
single_source        := source_subq | table_source
source_subq          := "(" mquery ") as" table_alias | "(" mquery ")" table_alias | "(" mquery ")"
table_source         := table_name "as" table_alias | table_name
join_clauses         := join_clause join_clauses | join_clause
join_clause          := joinop table_source "on" join_condition_clause
joinop := "join" | "left outer join"
join_condition_clause := join_condition "and" join_condition_clause | join_condition
join_condition       := col_ref "=" col_ref
groupby_clause       := "groupby" group_clause
group_clause         := subject "," group_clause | subject
where_clause         := "where" expr where_conj | "where" expr
where_conj           := "and" expr where_conj | "and" expr | "or" expr where_conj | "or" expr
expr                 := subject "in (" mquery ")" | subject "not in(" mquery ")"
                      | subject binaryop "(" mquery ")" | subject binaryop "all(" mquery ")"
                      | subject binaryop "any(" mquery ")" | subject binaryop value
value                := col_ref | non_literal_number | string
binaryop             := "!=" | "<" | "<=" | "<>" | "=" | ">" | ">=" | "like" | "not like"
having_clause        := "having" expr having_conj | "having" expr
having_conj          := "and" expr having_conj | "and" expr | "or" expr having_conj | "or" expr
orderby_clause       := "orderby" order_clause
order_clause         := ordering_term "," order_clause | ordering_term
ordering_term        := subject ordering | subject
ordering             := "asc" | "desc"
limit                := "limit" non_literal_number
non_literal_number.  := "150000" | "750" | "0" | "1" | "2" | "3" | "4"
string               := "\'chattahoochee\'" | … | "\'rio grande\'" | "\'potomac\'"
```

Figure 10: The grammar rules that we adapt from (Bogin et al., 2019) for SQL Geo (G1).

```
statement    := mquery
mquery       := query
query        := select_core groupby_clause orderby_clause "limit 1" | select_core groupby_clause orderby_clause
               | select_core groupby_clause | select_core orderby_clause "limit 1" | select_core orderby_clause
               | select_core
select_core  := select_with_distinct select_results from_clause where_clause
               | select_with_distinct select_results from_clause
select_with_distinct := "select distinct" | "select”
select_results       := select_result "," select_results | select_result
select_result        := col_ref selectop col_ref | col_ref | function "as" column_alias
                       | function selectop function | function
selectop     := "+" | "-" | "/"
col_ref      := column_name "as" column_alias | column_name | table_alias "." column_name
function     := fname "(distinct" arg_list_or_star ")" | fname "(" arg_list_or_star ")"
fname        := "all" | "avg" | "count" | "max" | "min" | "sum"
arg_list_or_star     := "*" | col_ref
column_alias         := "derived_fieldalias0" | "derived_fieldalias1"
column_name          := "*" | "area" | "border" | "capital" | … | "population" | "river_name" | "state_name"
table_alias          := "border_infoalias0" | … | "statealias3" | "statealias4" | "statealias5" | "tmp"
table_name           := "border_info" | "city" | "highlow" | "lake" | "mountain" | "river" | "state"
from_clause          := "from" source | "from" table_source join_clauses
table_source         := table_name "as" table_alias
source               := single_source "," source | single_source
single_source        := "(" mquery ") as" table_alias | table_source
join_clauses         := join_clause join_clauses | join_clause
join_clause          := joinop table_source "on" join_condition_clause
join_condition_clause:= join_condition "and" join_condition_clause | join_condition
join_condition       := col_ref "=" col_ref
joinop               := "join" | "left outer join"
where_clause         := "where" expr where_conj | "where" expr
where_conj           := "and" expr where_conj | "and" expr | "or" expr where_conj | "or" expr
expr                 := col_ref "in" source_subq | col_ref "not in" source_subq
                       | col_ref binaryop "all" source_subq | col_ref binaryop "any" source_subq
                       | col_ref binaryop source_subq | col_ref binaryop value
source_subq          := "(" query ")"
binaryop             := "!=" | "*" | "+" | "-" | "/" | "<" | "<=" | "<>" | "=" | ">" | ">=" | "like"
                       | "not like"
value                := col_ref | string
groupby_clause       := "group by" group_clause having_clause | "group by" group_clause
group_clause         := col_ref "," group_clause | col_ref
having_clause        := "having" having_expr having_conj | "having" having_expr
having_conj          := "and" having_expr having_conj | "and" having_expr | "or" having_expr having_conj
                       | "or" having_expr
having_expr          := function "in" source_subq | function "notin" source_subq
                       | function binaryop "all" source_subq | function binaryop "any" source_subq
                       | function binaryop source_subq
orderby_clause       := "orderby" order_clause
order_clause         := ordering_term "," order_clause | ordering_term
ordering_term        := ordering_expr ordering | ordering_expr
ordering             := "asc" | "desc"
ordering_expr        := col_ref | function
string               := "'red'" | "'usa'" | city_name | digit_value | mountain_name | place
                       | river_name | state_name
city_name            := "\'detroit\'" | … | "\'plano\'" | "\'des moines\'"
digit_value          := "750" | "0" | "150000"
mountain_name        := "\'mckinley\'" | "\'whitney\’”
place                := "\'death valley\'" | "\'mount mckinley\'" | "\'guadalupe peak\'"
river_name           := "\'north platte\'" | "\'chattahoochee\'" | "\'rio grande\'" | "\'potomac\'"
state_name           := "\'oregon\'" | "\'georgia\'" | … | "\'wisconsin\'" | "\'montana\'"
```

Figure 11: The grammar rules that we induce for SQL Geo (G2).