Unobserved Local Structures Make Compositional Generalization Hard

Ben Bogin[1] **Shivanshu Gupta**[2] **Jonathan Berant**[1]
[1]Tel-Aviv University [2]University of California Irvine
{ben.bogin,joberant}@cs.tau.ac.il, shivag5@uci.edu

Abstract

While recent work has convincingly showed that sequence-to-sequence models struggle to generalize to new compositions (termed *compositional generalization*), little is known on what makes compositional generalization hard on a particular test instance. In this work, we investigate what are the factors that make generalization to certain test instances challenging. We first substantiate that indeed some examples are more difficult than others by showing that different models consistently fail or succeed on the same test instances. Then, we propose a criterion for the difficulty of an example: a test instance is hard if it contains a *local structure* that was not observed at training time. We formulate a simple decision rule based on this criterion and empirically show it predicts instance-level generalization well across 5 different semantic parsing datasets, substantially better than alternative decision rules. Last, we show local structures can be leveraged for creating difficult adversarial compositional splits and also to improve compositional generalization under limited training budgets by strategically selecting examples for the training set.

1 Introduction

Recent analyses of pre-trained sequence-to-sequence (seq2seq) models have revealed that they do not perform well in a *compositional generalization* setup, i.e., when tested on structures that were not observed at training time (Lake and Baroni, 2018; Finegan-Dollak et al., 2018; Keysers et al., 2020). However, while accuracy on unseen structures drops on average, models do not *categorically* fail in compositional setups, and in fact are often able to successfully emit new unseen structures. This raises a natural question: **What are the conditions under which compositional generalization occurs in seq2seq models?**

Measuring compositional generalization at the *dataset level* obscures the fact that for a particular *test instance*, performance depends on the instance sub-structures and the examples that the model observed during training. Consequently, it might be possible to predict how difficult a test instance is given the test instance and the training set. Indeed, papers that provide multiple compositional splits (Kim and Linzen, 2020; Bogin et al., 2021a) have demonstrated that there is high variance in accuracy across splits, where some are easy to solve and some are much harder.

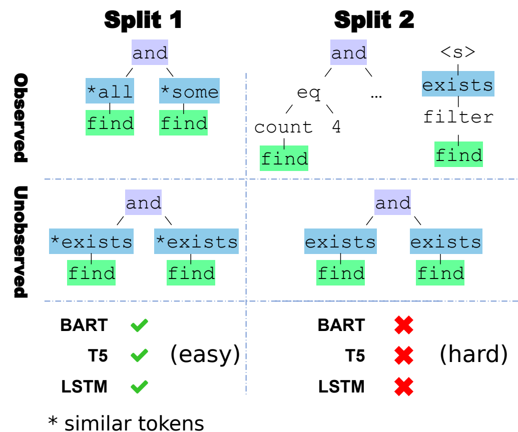

Figure 1: Unobserved local structures are harder for models to generalize to whenever there are no *similar* structures that were observed during training.

In this paper, we investigate the question of what makes compositional generalization hard in the context of semantic parsing, the task of mapping natural language utterances to executable programs. First, we create a large set of *compositional* train/test data splits over multiple datasets, in which there is *no* overlap between the programs of the test set and the training set. We then fine-tune and evaluate multiple seq2seq models on these compositional splits. Our first finding is that different models tend to agree on which test examples are difficult and which are not. This indicates example difficulty can be mostly explained by the example itself and the training data, independent of the

model. This calls for a better characterization of what makes a test instance hard.

To this end, we analyze the factors that make compositional generalization hard *at the instance level*. We formulate a simple decision rule that predicts the difficulty of test instances across multiple splits and datasets. Our main observation is that a test instance is hard if it contains a *local structure* that was not observed at training time. An unobserved local structure is defined as a small connected sub-graph that occurs in the program of the test instance, but does not occur in the training set. Moreover, unobserved structures render an instance difficult only if there are no observed structures that are *similar* to the unobserved one, where similarity is defined through a simple distributional similarity metric. Fig. 1 presents two splits that contain a tree with the same local structure in the test set. Split 1 is easy because the training set contains similar local structures: `exists` is similar to `all` and `some`, thus the model is likely to generalize to this new structure. Conversely, in Split 2 emitting the unobserved structure will be hard, as there is no similar observed structure in the training set.

We empirically evaluate our decision rule on five different datasets with diverse semantic formalisms, and show it predicts instance-level generalization well for both synthetic and natural language inputs, with an area under curve (AUC) score ranging from 78.4 to 93.3 (across datasets), substantially outperforming alternative rules that do not consider the program structure. Moreover, we compare our approach to MCD (Keysers et al., 2020), a metric that has been used to characterize difficulty at the dataset level (and *not* the instance level). We show our rule can be generalized to the dataset level and outperforms MCD in predicting the difficulty of various compositional splits. Last, we find that our decision rule applies not just to Transformer-based models, but also to an LSTM-based seq2seq model.

With the insight provided by our analysis, we use our decision rule for two purposes. First, we develop a method for creating difficult compositional splits by picking a set of similar local structures, and holding out instances that include any of these structures. We show that indeed, seq2seq models get much lower accuracy on these splits compared to prior approaches. Second, we propose a data-efficient approach for selecting training examples in a way that improves generalization to new structures. Given a large pool of examples, we select a training set that maximizes local structure similarity between examples in the training set and examples that are outside of it. We show this leads to better compositional generalization for a fixed budget of training examples.

To conclude, unobserved local structures are a key factor in compositional generalization, and they can be used to create challenging compositional splits and to effectively sample training data. We hope these insights will lead to further studies on factors behind compositional generalization and to methods for improving generalization in seq2seq models. Our code, data and models are available at: `https://github.com/benbogin/unobserved-local-structures`.

2 Setup

We focus on semantic parsing, where the task is to parse an utterance x into an executable program z. In a compositional generalization setup, examples are split into a training set and a test set, such that there is no overlap between programs in the two sets. We describe the two methods used to generate compositional splits in §2.1, and present the analyzed datasets in §2.2.

2.1 Generating Splits

Template split The first method is the random template split proposed in Finegan-Dollak et al. (2018). In this method, programs are first converted into *abstract templates* using a program anonymization function, that replaces certain program tokens such as string values, numbers and entities with their abstract type (e.g., for the input *"are there people that works for crossbars ?"* in Table 1, the string value `crossbars` is replaced with `STR_VAR`). We refer to App. A for the exact anonymization function used in each dataset.

We group examples according to their abstract template, randomly split the templates into a training set and a test set, and place each example in the train/test set according to their abstract template. While not a part of the original procedure, to make sure splits are "solvable", we verify in each split that there is no token that appears in the test set but does not appear in the training set (otherwise, we discard the split). To obtain multiple splits, we perform multiple random splits of templates.

Grammar split We propose a grammar split, which can be used when we have the set of context-

Dataset	Example
COVR (synthetic)	*What is the number of gray square cat that is looking at mouse?* `count (with_relation (filter (gray, filter (square, find(cat))), looking_at, find (mouse)))`
Overnight (synthetic†, natural°)	†*number of played games of player kobe bryant whose number of assists is 3* °*how many games has kobe bryant made more than 3 assists* `(listValue (getProperty (filter (getProperty kobe_bryant (reverse player)) num_assists = 3) num_games_played))`
Schema2QA (synthetic)	*are there people that works for crossbars ?* `( Person ) filter worksFor contains "crossbars"`
ATIS (natural)	*round trip flights between new york and miami* `( lambda $0 e ( and ( flight $0 ) ( round_trip $0 ) ( from $0 new_york : ci ) ( to $0 miami : ci ) ) )`

Table 1: An example utterance-program pair for each of the datasets used in this work.

free grammar rules $\mathcal{G}$ that generate the dataset programs. This can create meaningful splits, since generation is not random. For example, this can create a split where the test set contains exactly the set of examples that have the token `and` as a parent of the token `exists` (in the program tree). Such a split is unlikely to emerge in a template split (see App. B.2 for example grammar splits).

To create a grammar split for a set of examples, we look at the *derivation* $d = (r_1, ..., r_{N_d})$ of each example program, i.e., the sequence of production rules from the grammar that generate that program, where $r_i \in \mathcal{G}$. We create a compositional split by holding out a small number N_U of *pairs* of grammar rules, $\mathcal{U} = \{(r_{1_i}, r_{2_i})\}_{i=1}^{N_\mathcal{U}}$, and define the test set to contain any example whose program derivation contains a pair of rules r_1 and r_2, where $(r_1, r_2) \in \mathcal{U}$. For example, if $\mathcal{U}$ contains a single pair of rules (r_1, r_2), where r_1 is a rule that produces the terminal `and` and r_2 is a rule that produces the terminal `exists`, then the test set will include only (and all) examples that have both these rules in their derivation. Due to the structure of the grammar, this means the test programs will always have `exists` as a child of `and` in the program tree. Importantly, the training set will still have examples with both of these two terminals separately. We repeat this process with different instances of $\mathcal{U}$, see App. B.2 for details.

2.2 Datasets

We use five different datasets in our analysis, covering a wide variety of domains and semantic formalisms. Importantly, since we focus on compositional generalization, we choose datasets for which a random, i.i.d split, yields *high* model accuracy. Thus, errors that the models make in the compositional setup can be attributed to the compositional challenge, and not to conflating issues such as lexical alignments or ambiguity.

We consider both synthetic datasets, generated with a synchronous context-free grammar (SCFG) that generates utterance-program pairs, and datasets with natural language utterances. The datasets we use are (see Table 1 for examples):

COVR A synthetic dataset that uses a variable-free functional query language. The grammar is adapted from a visual question answering dataset that carries the same name (Bogin et al., 2021a).

Overnight (Wang et al., 2015): A dataset that contains both synthetic and natural utterances, covering 11 domains such as *housing*, *calendar* and *publications*, and uses the Lambda-DCS formal language (Liang et al., 2011). Utterance-program pairs are generated using a SCFG, providing synthetic utterances, and then manually paraphrased to natural language through crowdsourcing.

Schema2QA (**S2Q**, Xu et al. 2020): A dataset covering 6 domains such as *restaurants*, *movies* and *people* that uses the ThingTalk language (Campagna et al., 2019). This dataset contains both synthetic and paraphrased utterances, similar to Overnight, however we do not use these provided paraphrased examples, and instead only use the synthethic examples from the *people* domain, which were provided by Oren et al. (2021).

ATIS (Hemphill et al., 1990; Dahl et al., 1994): A dataset with natural language questions about aviation, and λ-calculus as its formal language.

In total, we use three datasets with synthetic language, and two with natural language.

3 Do Models Agree on the Difficulty of Test Examples?

Our goal is to predict how difficult a test instance is based on the instance itself and the training set. However, to conduct such an analysis, we must confirm that (1) splits contain both easy and difficult instances and (2) different models generally agree on how difficult a specific instance is. To check this, we evaluate different models on multiple splits.

Splits We experiment with the five datasets listed in §2.2. For COVR, we use the grammar split, which generates 124 splits. For Overnight, we generate 5 template splits for each domain, and use the same splits for both the synthetic and the natural language versions. For ATIS and S2Q, we generate 5 template splits. The size of the training sets across datasets ranges from 600-18,000, however, for a single dataset (and in Overnight, a single domain), the size of the training set is roughly similar across splits. For Overnight and ATIS, we hold out 20% of all program templates. For Schema2QA, we hold out 70%, since otherwise model performance is too high. We provide details of the splits in each dataset in App. B.3.

Models and experimental setting We experiment with four pre-trained seq2seq models: T5-Base, T5-large, BART-Base and BART-large (Raffel et al., 2020; Lewis et al., 2020). All models are fine-tuned for a total of 64 epochs with a batch-size ranging between 8 to 28, depending on model size and the maximum number of example tokens in each dataset. We use a learning rate of $3e^{-5}$ with polynomial decay.

We evaluate performance with *exact match (EM)*, that is, whether the predicted output is equal to the gold output. However, since EM is often too strict and results in false-negatives, we use a few manually-defined relaxations (detailed in App. A). We do early stopping using the test set, since we do not have a development set. As our goal is not to improve performance but only analyze instance difficulty, we argue this is an acceptable choice in our setting.

Results Table 2 (top) shows the average results across splits on all datasets, where we see that the accuracy of different models is roughly similar per

Model	COVR	Overnight syn. / nat.	S2Q	ATIS
T5-Base	88.2	59.0 / 23.6	83.4	78.4
T5-Large	87.8	62.3 / 27.4	88.0	77.6
BART-Base	86.3	60.4 / 27.8	77.9	76.0
BART-Large	85.4	60.3 / 28.2	82.0	78.9
Agree. (3+/4)	94.5	96.2 / 93.5	93.1	96.4
Agree. (4/4)	82.5	86.8 / 76.9	75.3	85.6
Rnd-agree. (4/4)	57.1	15.8 / 29.3	47.0	36.7

Table 2: Average test EM across splits for all datasets, agreement rate for at least 3 models (3+/4) and all 4 models (4/4), and random agreement rate (4/4). For Overnight, we show results both on the synthetic and the natural language settings.

dataset. Models tend to do well on most compositional splits, except in Overnight with natural language inputs. To measure agreement across models, we compute *agreement rate*, that is, the fraction of times all (or most) models either output a correct prediction or output an erronous one. We observe that agreement rate is high (Table 2, middle): at least 3 models agree (3+/4) in 93.1%-96.4% of the cases, and all (4/4) models agree in 75.3%-86.8%. Moreover, we compare this to *random agreement rate* (Table 2, bottom), where we compute agreement rate assuming that a model with accuracy p outputs a correct prediction for a random subset of p the examples and an incorrect prediction for the rest. We observe that agreement rate is dramatically higher than random agreement rate.

Importantly, the fact that models have high agreement rate suggests that instance difficulty depends mostly on the instance itself and the training set, and not on the model.

4 What makes an instance hard?

The results in §3 provide a set of test instances of various difficulties from a variety of compositional splits. We analyze these instances and propose a hypothesis for what makes generalization to a test instance difficult.

4.1 Unobserved Local Structure

We conjecture that a central factor in determining whether a test instance is difficult, is whether its program contains any **unobserved local structures**. We represent a program as a graph, and a local structure as a connected sub-graph within it. We formally define these concepts next.

The output z in our setting is a sequence of to-

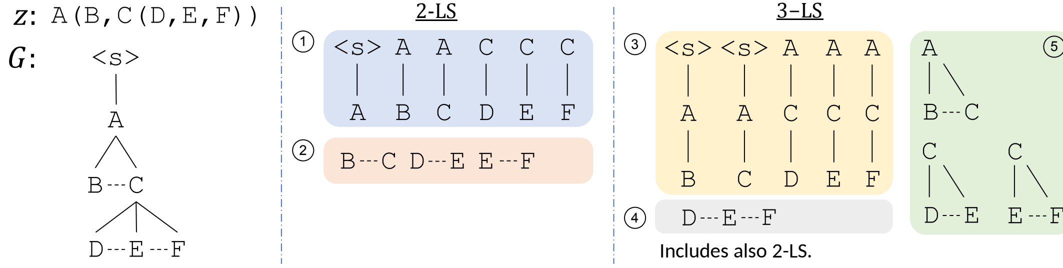

Figure 2: An example program z and the structure of its program graph G (left), with solid edges for parent-child relations and dashed edges for consecutive siblings. In the middle and right we enumerate all 2-LS and 3-LS structures over this graph.

kens which defines a program. Each token in z represents a program *symbol*, which is a function or a value, except for structure tokens (namely, parentheses and commas) that define parent-child relations between function symbols and their arguments. We parse z into a tree $T = (\mathcal{V}, \mathcal{E})$, such that each node $v \in \mathcal{V}$ is labeled by the symbol it represents in z, and the set of edges $\mathcal{E} = \{(p, c)\}$ expresses **parent-child** relations between the nodes. We additionally add a root node <s> connected as a parent to the original root in T.

To capture also sibling relations, and not only parent-child relations, we define a graph based on the tree T that contains an additional edge set $\mathcal{E}_{\text{sib}}$ of **sibling** edges: $G = (\mathcal{V}, \mathcal{E} \cup \mathcal{E}_{\text{sib}})$. Specifically, for each parent node p, the program z induces an order over the children of p: $(c_1^p, ..., c_{N_p}^p)$, where N_p is the number of children. We then define $\mathcal{E}_{\text{sib}} = \bigcup_p \{c_i^p, c_{i+1}^p\}_{i=1}^{N_p}$, that is, all *consecutive* siblings will be connected by edges. Fig. 2 (left) shows an example program z and its graph G.

We define local structures as connected subgraphs in G, with $2 \leq n \leq 4$ nodes, that have a particular structure, presented next. We term a local structure with n nodes as n-LS. The set of 2-LSs refers to all pairs of parents and their children, and all pairs of consecutive siblings (Fig. 2, structures 1 and 2). The set of 3-LSs includes all 2-LSs, and also structures with (1) two parent-child relations, (2) two siblings relations and (3) a parent with two siblings (structures 3, 4 and 5 in the figure, respectively). The structure 4-LS is a natural extension of 2-LS and 3-LS, defined in App. C. Importantly, the structures we consider are local since they are *connected*: We do not consider, for example, a grandparent-grandchild pair, or non-consecutive siblings.

4.2 Similarity of Local Structures

Our hypothesis is that if a model observes a test instance with an unobserved local structure, this instance will be difficult to predict. We relax this hypothesis and propose that an example might be easy even if it contains an unobserved local structure s_1, if the training set contains a structure s_2 that is *similar* to s_1.

The similarity $\text{sim}(s_1, s_2)$ of two structures is defined as follows. Two structures can have a positive similarity if (1) they are *isomorphic*, that is, they have the same number of nodes and the same types of edges between the nodes, and (2) if they are identical up to flipping the label of a *single* node. In any other case, their similarity will be 0.0. If all nodes are identical, similarity is 1.0. In the case where they are identical up to flipping the label of a single node, we define their similarity using the *symbol similarity* of the two symbols m_1 and m_2 that are different, which is computed using a distributional similarity metric $\widehat{\text{sim}}(m_1, m_2)$.

To compute symbol similarity we take advantage of the set of programs $\mathcal{P}_o$ in all training examples. We use this set to find the *context* that co-occurs with each symbol in the training set – that is, the set of symbols that have appeared as parents, children or siblings of a given symbol m. Specifically, we consider four types of contexts $c \in \mathcal{C}$, including children, parents, left siblings, and right siblings. We define $\text{ctx}_c(m)$ as the set of symbols that have appeared in the context c of the symbol m. Finally, given two symbols m_1 and m_2, we average the Jaccard similarity across the set of *relevant* context types $\tilde{\mathcal{C}} \subseteq \mathcal{C}$:

$$\widehat{\text{sim}}(m_1, m_2) = \frac{1}{|\tilde{\mathcal{C}}|} \sum_{c \in \tilde{\mathcal{C}}} \text{jac}(\text{ctx}_c(m_1), \text{ctx}_c(m_2)),$$

where $\tilde{\mathcal{C}}$ contains a context type c iff the set of

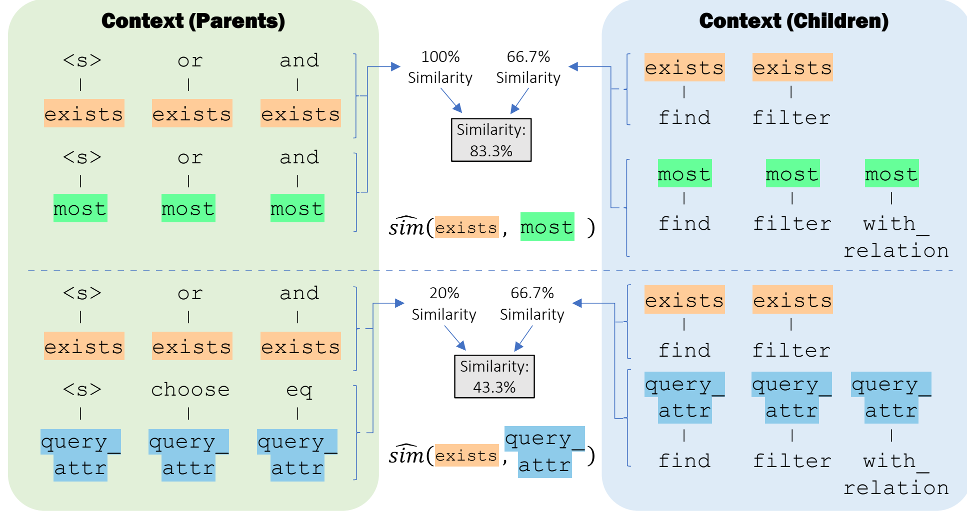

Figure 3: Two Examples for the computation of similarity $\widehat{\text{sim}}(m_1, m_2)$ between two symbols m_1, m_2.

context symbols $\text{ctx}_c(\cdot)$ is not empty for either m_1 or m_2.

Consider the top example in Fig. 3. The token `exists` appears with 3 different parents: `<s>`, `or` and `and`. The token `most` appears with exactly the same set of parents, thus their "parent" similarity is 100%. Likewise, the "children" similarity of the two is 66.7% since they share 2 out of 3 distinct children (`find` and `filter`). Finally, the similarity between the tokens is the average of the two types of similarities, 83.3% (for brevity, sibling contexts are not considered in the figure).

4.3 Decision Rule

We can now use the similarity between structures to predict the difficulty of an unobserved program p_u given a set of observed programs $\mathcal{P}_o$. For exposition purposes, instead of predicting *difficulty*, we predict its complement: *easiness*.

We predict the easiness of a program p_u by comparing its local structures with the local structures in $\mathcal{P}_o$. Thus, we parse p_u into a graph and extract the set $\mathcal{S}_u$ of n-LSs as defined in §4.1, for a chosen n. Similarly, we extract the set $\mathcal{S}_o$ of all n-LSs in the set of training programs $\mathcal{P}_o$, and define the easiness of p_u:

$$\hat{e}(p_u) = \min_{s_u \in \mathcal{S}_u} \max_{s_o \in \mathcal{S}_o} \left(\text{sim}(s_u, s_o)\right),$$

that is, the easiness of p_u is determined based on the *least easy* (most difficult) unobserved structure in $\mathcal{S}_u$. The easiness of a particular structure s_u is determined by the structure in $\mathcal{S}_o$ that is *most similar* to it.

4.4 Alternative Decision Rules

Our proposed decision rule is a result of manual analysis. Next, we discuss alternative decision rules, which will be evaluated as baselines in §5.

N-grams over sequence While we define our structures over program *graphs*, sequence to sequence models observe both input and output as a flat sequence of symbols, without any structure. Thus, it is possible that unobserved *n-grams* in test instances, rather than local structures, explains difficulty. To test this, we define our decision rule to be identical, but replace local structures with consecutive n-grams in the program sequence, and consider only two context types (left/right co-occurrence) to compute symbol similarity.

Length It is plausible that long sequences are more difficult to generalize to. To test this, given a set of training programs $\mathcal{P}_o$, we measure the number of symbols m_l in the longest program in $\mathcal{P}_o$, and define the easiness of a program p_u of length m_u to be $\max\left(1 - \frac{m_u}{m_l}, 0\right)$.

TMCD The MCD and TMCD methods (Keysers et al., 2020; Shaw et al., 2021) have been used to create compositional splits, by maximizing *compound divergence* across the training and test splits. A compound, which is analogous to our n-LS, is defined as any sub-graph of up to a certain size in the program tree, and divergence is computed over the distributions of compounds across the two sets (see papers for details). We can use this method also to *predict* difficulty instead of creating splits and compare it to our approach. While the two

methods are not directly comparable, since we focus on *instance-level generalization*, we extend our approach for computing the easiness of *a split* and compare to TMCD in §5.

5 Experiments

We now empirically test how well our decision rule predicts the easiness of test instances.

Setup We formalize our setup as a binary classification task where we predict the easiness $\hat{e}(p_u) \in [0, 1]$ of a test instance (utterance-program pair) with program p_u, and compare it to the "gold" easiness $e(p_u) \in \{0, 1\}$, as defined next. For each test instance, we have the EM accuracy of four models (see §3). We thus define $e(p_u)$ to be the *majority* EM on p_u between the four models. We discard examples where there was no majority (this happens in 3.6% to 6.9% of the cases in each dataset). In each dataset, we combine all test instances across all splits. We evaluate with Area Under the Curve (AUC), a popular metric for binary classification that does not require setting an "easiness threahold", where we compute the area under the curve of the true positive rate (TPR) against the false positive rate (FPR).

Decision rules We compare three variations of our n-LS decision rule with $n \in \{2, 3, 4\}$. Additionally, we evaluate the N-grams over sequence baseline (2-BIGRAM) and LENGTH, as described in §4.4. The RANDOM baseline samples a number between 0 to 1 uniformly at random.

We also conduct three ablations to our decision rule. The first, 2-LS-NOSIB ignores sibling relations in program graphs. This affects not only the set of local structures considered, but also the context types used in the symbol similarity function. Similarly, 2-LS-NOPC ignores parent-child relations. Our third ablation 2-LS-NOSIM tests a more strict decision rule that ignores structure similarity, i.e., $\hat{e}(p_u) = 0$ for a test program p_u that has *any* unobserved 2-LS, even if the unobserved structures have similar observed structures in the training set.

5.1 Results

The AUC scores for all decision rules are presented in Table 3, showing that our n-LS classifiers get high AUC scores, ranging from 78.4-93.3 depending on the dataset and the order n of structures, outperforming the baselines and ablations.

Decision Rule	COVR	Overnight syn. / nat.	S2Q	ATIS
2-LS	**93.3**	84.6 / 67.9	78.9	75.8
3-LS	93.1	**91.0** / 74.8	**81.6**	78.7
4-LS	92.1	88.0 / **78.4**	79.9	**81.4**
2-LS-NOSIB	91.9	78.0 / 65.8	75.9	67.7
2-LS-NOPC	77.2	69.6 / 59.7	61.0	64.1
2-LS-NOSIM	85.1	82.3 / 67.2	79.0	73.5
2-BIGRAM	88.5	58.2 / 52.4	69.7	69.9
LENGTH	49.7	45.9 / 55.3	43.4	47.4
RANDOM	50.5	49.5 / 51.6	48.2	49.2

Table 3: AUC scores of different decision rules for each dataset, computed across the test instances in all splits.

Comparing performance across the order n of LSs, there is some variance across datasets, and the best n may depend on the dataset. Still, local structure explains generalization better than the baselines practically for all n's.

The two graph-relation ablations (2-LS-NOSIB and 2-LS-NOPC) show that parent-child relations are more important than sibling relations, but both contribute to the final easiness score. The strict similarity ablation suggests that considering similarity between structures is important in COVR, but less significant in Overnight and ATIS, and not at all in S2Q. We hypothesize that similarity is important in COVR since the splits were created with a grammar, and not with templates. In grammar splits, often a single group of similar structures is split across the training and test sets (e.g., half of the quantifiers are in the training set, and half in the test set). In such cases, considering local structure similarity is important, since such test instances are easier according to our conjecture. Such splits are unlikely to emerge in a template split. The experiment we conduct in §6.1, where we specifically create such splits, supports this claim.

The low accuracy of 2-BIGRAM indicates that unobserved structures in the program space are much better for predicting generalization compared to unobserved sequences. This is surprising since models train with symbol sequences only. Last, we see that predicting difficulty by example length is as bad as a random predictor.

Performace on datasets with natural language (ATIS and Overnight-paraphrased) are lower than synthetic datasets. One reason for this is that natural language introduces additional challenges such as ambiguity, lexical alignment and evaluation errors – mistakes that cannot be explained by our decision rule. We further discuss limitations of our

Utterance	Either the number of dog is greater than the number of white animal or there is dog
Gold	`or(gt(count(find(dog)),count(filter(white,find(animal)))),exists(find(dog)))`
T5	`or(gt(count(find(dog)),count(filter(white,find(animal)))),there(find(dog)))`
BART	`or(gt(count(find(dog)),count(filter(white,find(animal)))),ists(find(dog)))`
T5-L	`or(gt(count(find(dog)),count(filter(white,find(animal)))),eq(find(dog)))`
BART-L	`or(gt(count(find(dog)),count(filter(white,find(animal)))),with_relation(...`

Table 4: Example predictions of four models, showing a typical case where all models emit wrong tokens exactly when encountering an unobserved parent-child structure: `or-exists` (`L` in model name stands for a large model).

Split	COVR	Overnight
2-LS	**0.79**	0.84
3-LS	0.66	**0.91**
4-LS	0.62	0.83
TMCD	-0.36	-0.60

Table 5: Pearson correlation between the easiness score of a split and the average model EM, shown for n-LS and TMCD, for COVR and Overnight (synthetic).

Decision Rule	COVR	Overnight syn / nat.	S2Q	ATIS
2-LS (Trans.)	93.3	84.6 / 67.9	78.9	75.8
3-LS (Trans.)	93.1	91.0 / 74.8	81.6	78.7
4-LS (Trans.)	92.1	88.0 / 78.4	79.9	81.4
2-LS (LSTM)	80.0	77.3 / 66.5	56.8	71.8
3-LS (LSTM)	81.6	84.2 / 71.0	60.7	76.9
4-LS (LSTM)	**82.6**	**85.8 / 72.8**	**60.9**	**79.6**

Table 6: AUC scores of our decision rule across each dataset, for experiments with an LSTM-based decoder (bottom) compared to a Transfomer (top).

approach in §8.

Token-level analysis We analyzed model errors at the instance level. Next, we analyze errors at the *token level*. Specifically, we check the relation between the first incorrect token the models emit and the unobserved structures. Consider the example in Table 4: given a test example with an unobserved parent-child structure `or-exists`, all models emit a wrong token precisely where they should have output the child, `exists`.

We measure the frequency of this by looking at model outputs where (1) the model is wrong and (2) unobserved 2-LSs $\mathcal{S}_u$ were found in the gold program. For each such model output $\hat{z}$ and gold output z, we find the index i of the first token where $\hat{z}^i \neq z^i$. We then count the fraction of cases where there is an unobserved 2-LS (a pair of symbols) $(m_1, m_2) \in \mathcal{S}_u$ such that both m_1 appears in the program prefix $\{z\}_{j=1}^{i-1}$ and $m_2 = z^i$. For COVR, this happens in 76.5% of the cases. For Overnight, S2Q and ATIS, this happens in 28.4%, 31.8% and 45.1% of the cases respectively, providing strong evidence that models struggle to emit local structures they have not observed during training.

5.2 Comparison to TMCD

As discussed (§4.4), Maximum Compound Divergence (MCD) and its variation TMCD, are recently proposed metrics for estimating the difficulty of a test set, while we measure difficulty at the instance level. To measure how well can local local structures predict performance at the split level, we average the EM scores that the 4 models get on each split and use this average as the "gold" easiness of that split. We then average the easiness predictions of our decision rule across all instances in a split to obtain an easiness prediction for that split. For TMCD, we compute the compound divergence of each split (high compound divergence indicates a more difficult split, or lower easiness) following Shaw et al. (2021), see details in App.D.

We evaluate by measuring Pearson correlation between the predicted scores and gold scores. We show results for COVR and Overnight only, since the number of splits in these two datasets is large enough (124 and 50 splits respectively). The results, shown in Table 5, demonstrate that n-LS correlates better with the EM of models, for any order n of structures.

5.3 Model Architecture Effect

We have shown that unobserved structures are a key factor in explaining what makes compositional generalization hard. We now check if this insight generalizes beyond Transformer-based pre-trained seq2seq models to other architectures.

To that end, we repeat our experiments with an LSTM (Hochreiter and Schmidhuber, 1997)

decoder. Precisely, we use an LSTM decoder with a copying mechanism (Gu et al., 2016), and $\text{BERT}_{\text{BASE}}$ as the encoder.[1]

Results are given in Table 6, showing that AUC scores when using an LSTM are close to the scores received with a Transfomer (except S2Q, see below). This indicates that even with a different architecture, the LSTM mostly errs due to unobserved local structures. In addition, our decision rule works well even though the EM of the LSTM is lower (70.0 for COVR, 50.8/20.5 for Overnight synthetic and paraphrased, 54.1 for S2Q, and 64.4 for ATIS), indicating that the cases where the LSTM is wrong and the Transfomer is correct, are often when the easiness prediction is lower. The only exception is S2Q that gets much lower EM compared to Transformers and a large drop in AUC as well.

6 Leveraging Local Structures

We now show we can take advantage of the insights presented to (1) create challenging compositional splits (§6.1) and (2) to improve data sampling efficiency for compositional generalization (§6.2).

6.1 n-LS Splits

We have shown that when splits are created with either the template or grammar split methods, a key reason for explaining model failure is unobserved local structures. In the next experiment, we test our conjecture from the opposite direction: we evaluate the accuracy of models when tested on adversarial splits that were designed to contain unobserved local structures. We compare the accuracy of models on such adversarial splits against accuracy on a random template split, where we hold out $K = 0.3$ of the program templates.

To create an adversarial split, we split the entire set of programs $\mathcal{P}$ into a set of training programs $\mathcal{P}_o$ and test programs $\mathcal{P}_u$, such that if we look at the set $\mathcal{S}_o$ of observed n-LSs in $\mathcal{P}_o$, and the set $\mathcal{S}_u$ of unobserved n-LSs in $\mathcal{P}_u$, the similarity between any structure in $\mathcal{S}_o$ and any structure in $\mathcal{S}_u$ will be minimal. Given $\mathcal{S}_u$, a split of examples is created by setting the test set to contain all examples that have any of the structures in $\mathcal{S}_u$, and the training set to contain all other examples.

To create adversarial splits, we go over the set of all n-LSs, $\mathcal{S}$, in the set of programs $\mathcal{P}$, and from each $s \in \mathcal{S}$ we attempt to create an adversarial split

[1] Accuracy with an LSTM encoder was too low even in an i.i.d split.

Split	COVR	Overnight
2-LS	32.8 ± 43.6	23.1 ± 33.6
2-LS-NoSib	**19.7** ± 34.6	**8.4** ± 15.8
Template	94.6 ± 21.7	54.2 ± 25.7
2-LS-NoSib-Half	92.6 ± 20.0	38.6 ± 32.4

Table 7: EM on adversarial splits compared to a template split. Numbers are averaged across all created splits and across 4 models. We show standard deviation across the EM scores of all models on all generated splits.

independently. Given an n-LS, s, and a similarity threshold t, a candidate set of unobserved structures is $\mathcal{S}_u(t) = \{s_u \in \mathcal{S} \text{ s.t. } \text{sim}(s_u, s) > t\}$, that is, the set of all structures that are similar to s (which includes s by definition). When t is low, splits are hard, since structures in $\mathcal{S}_o$ will be less similar to $\mathcal{S}_u$. On the other hand, when t is too low, we might hold out a large fraction of programs, which might (a) create invalid splits, where symbols in the test set do not occur in the training set, or (b) the fraction of test set programs will exceed K, rendering comparison to a random template split unfair. Thus, we pick $\mathcal{S}_u$ with the lowest t such that (1) $\mathcal{S}_u(t)$ is valid (all test symbols occur in the training set) and (2) the fraction of program templates in the test set is at most K. Because our process does not guarantee a difficult split, we discard any split, if its easiness score is higher than a threshold τ (see App. B.4 for details and number of generated splits), and merge identical splits that are created from different structures $s, s' \in \mathcal{S}$.

We test two variations: one where $\mathcal{S}$ includes all 2-LS structures, and one where we ignore sibling relations (2-LS-NoSib). In addition, we create another set of splits where instead of setting $\mathcal{S}_u$ to include *all* structures that are similar to s, we instead randomly sample *half* of them, meaning some similar structures will exist in the training set (2-LS-NoSib-Half). To avoid excessive computation, we sample 15 splits if the number of generated splits is higher (see App. B.4 for further details). We compare our algorithm to five random template splits, generated with different random seeds.

Table 7 shows that EM of models on our adversarial splits are dramatically lower than on template splits, especially for the 2-LS-NoSib variation, with a difference of 74.9 or 45.8 absolute points. Moreover, the scores on 2-LS-NoSib-Half are much higher, re-enforcing our hypothesis that un-

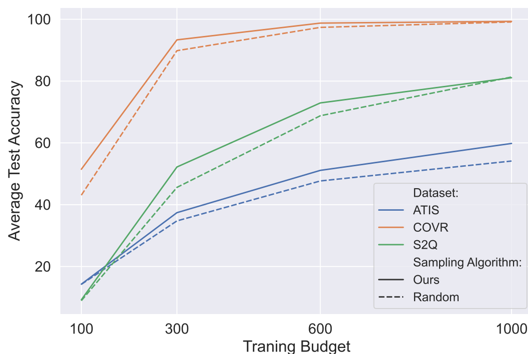

Figure 4: Average test accuracy when sampling based on local structures vs. random sampling on COVR, ATIS and S2Q as training budget is varied.

observed structures are only hard if there are no similar observed structures.

6.2 Efficient Sampling

We now test if we can leverage unobserved local structures to strategically choose examples that will lead to better compositional generalization. We assume access to a large set of examples $\mathcal{D}_{\text{pool}}$, but we can only use a small subset $\mathcal{D}_{\text{train}} \subset \mathcal{D}_{\text{pool}}$ for fine-tuning, where the budget for training is $|\mathcal{D}_{\text{train}}| \leq B$. Our goal is to improve accuracy on an unseen compositional test set $\mathcal{D}_{\text{test}}$, by choosing examples that are likely to reduce the number of unobserved structures in the test set. To simulate this setting, we use the template split method to generate $\mathcal{D}_{\text{pool}}$ and $\mathcal{D}_{\text{test}}$ for COVR, ATIS and S2Q, holding out 20% of the program templates for ATIS and S2Q, and 50% for COVR (the number of templates in each domain in Overnight was too low to use). Improving compositional generalization under budget constraints was recently explored by Oren et al. (2021), but as our experimental setup is different, results are not directly comparable.

We propose a simple iterative algorithm, starting with $\mathcal{D}_{\text{train}} = \phi$. We want our model to observe a variety of different local structures to increase the chance of seeing all local structures in $\mathcal{D}_{\text{test}}$. Thus, at each step, we add an example $e \in \mathcal{D}_{\text{pool}}$ such that e contains a local structure that is *least* similar to all other local structures already in $\mathcal{D}_{\text{train}}$. We do this by first finding the least similar local structure, and then randomly picking an example that contains this structure. We continue until $|\mathcal{D}_{\text{train}}| = B$.

We compare our algorithm against random sampling without replacement across a spectrum of training budgets. We evaluate both methods on 5 different template splits for each dataset, and for each template split, budget and sampling algorithm we perform the experiment with 3 different random seeds. Figure 4 shows the average compositional test accuracies on $\mathcal{D}_{\text{test}}$ for the two algorithms on the three datasets. Our sampling scheme outperforms or matches the random sampling method on all datasets and training budgets.

7 Related Work

Benchmarks Compositional splits are typically generated in one of two broad approaches: (a) researchers manually design dataset-specific splits (Bahdanau et al., 2019; Lake and Baroni, 2018; Bastings et al., 2018; Bogin et al., 2021a; Kim and Linzen, 2020; Ruis et al., 2020), and (b) compositional splits are automatically generated in a dataset-agnostic manner (as proposed in § 6.1). Automatic methods include splitting examples by output length (Lake and Baroni, 2018), by anonymizing programs (Finegan-Dollak et al., 2018), and by maximizing divergence between distribution across the training and test set (Keysers et al., 2020; Shaw et al., 2021).

Improving generalization Many approaches have been recently proposed to examine and improve generalization, including the effect of training size and architecture (Furrer et al., 2020), data augmentation (Andreas, 2020; Akyürek et al., 2021; Guo et al., 2021), data sampling (Oren et al., 2021), model architecture (Herzig and Berant, 2021; Bogin et al., 2021b; Chen et al., 2020), intermediate representations (Herzig et al., 2021) and different training techniques (Oren et al., 2020; Csordás et al., 2021).

Measuring compositional difficulty The most closely related methods to our work are MCD and its variation TMCD (Keysers et al., 2020; Shaw et al., 2021), designed to create compositional splits. In both methods, a tree is defined over the program of a given instance. In MCD, it is created from the tree of derivation rules that generates the program, and in TMCD from the program parse tree, similar to our approach. Then, certain sub-trees are considered as compounds, which are analogous to n-LSs (with the main exception that n-LSs contain consecutive siblings). Splits are created to maximize the divergence between the distributions of compounds in the training and test

sets.

However, since MCD and TMCD were designed to generate compositional splits, they were not tested on whether they *predict* difficulty of other splits, such as the template split. In addition, while in TMCD the difficulty is over an entire test set, we predict the difficulty of specific instances. Instance-level analysis can better characterize the challenges of compositional generalization, and as we show in §5.2 it is a better predictor of difficulty compared to TMCD even at the split level. Moreover, our approach can also be used to create challenging compositional splits (§6.1). Another important distinction is that we introduce the concept of structure similarity, which further improves the ability to predict the difficulty of instances.

8 Discussion

Limitations A limitation of our analysis is that we only use the *programs* of test instances to predict difficulty, but ignore the input *utterance*. Thus, we do not take into consideration the variance in natural language, which could play an important factor in determining difficulty, especially in compositional splits. In addition, we perform our analysis on datasets where models get high accuracies on i.i.d splits. Our decision rule may not work in datasets with other difficulties that conflate with the compositional challenge.

Conclusion We have shown that unobserved local structures have a critical role in explaining the difficulty of compositional generalization over a variety of datasets, formal languages, and model architectures, and demonstrated how these insights can be used for the evaluation of compositional generalization by creating challenging splits, and by improving data efficiency given a limited training budget. We hope our insights would be useful in future work for improving generalization in sequence to sequence models.

Acknowledgements

This research was partially supported by The Yandex Initiative for Machine Learning and the European Research Council (ERC) under the European Union Horizons 2020 research and innovation programme (grant ERC DELPHI 802800). We thank Elad Segal for his code to measure compound divergence (MCD), and Jonathan Herzig, Ankit Gupta and Sam Bowman for their helpful feedback. This work was completed in partial fulfillment for the Ph.D degree of Ben Bogin.

References

Ekin Akyürek, Afra Feyza Akyürek, and Jacob Andreas. 2021. Learning to recombine and resample data for compositional generalization. In *International Conference on Learning Representations*.

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

Dzmitry Bahdanau, Harm de Vries, Timothy J. O'Donnell, Shikhar Murty, Philippe Beaudoin, Yoshua Bengio, and Aaron C. Courville. 2019. CLOSURE: assessing systematic generalization of CLEVR models. *CoRR*, abs/1912.05783.

Jasmijn Bastings, Marco Baroni, Jason Weston, Kyunghyun Cho, and Douwe Kiela. 2018. Jump to better conclusions: SCAN both left and right. In *Proceedings of the 2018 EMNLP Workshop BlackboxNLP: Analyzing and Interpreting Neural Networks for NLP*, pages 47–55, Brussels, Belgium. Association for Computational Linguistics.

Ben Bogin, Shivanshu Gupta, Matt Gardner, and Jonathan Berant. 2021a. COVR: A test-bed for visually grounded compositional generalization with real images. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 9824–9846, Online and Punta Cana, Dominican Republic. Association for Computational Linguistics.

Ben Bogin, Sanjay Subramanian, Matt Gardner, and Jonathan Berant. 2021b. Latent compositional representations improve systematic generalization in grounded question answering. *Transactions of the Association for Computational Linguistics*, 9:195–210.

Giovanni Campagna, Silei Xu, Mehrad Moradshahi, Richard Socher, and Monica S. Lam. 2019. Genie: A generator of natural language semantic parsers for virtual assistant commands. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, page 394–410, New York, NY, USA. Association for Computing Machinery.

Xinyun Chen, Chen Liang, Adams Wei Yu, Dawn Song, and Denny Zhou. 2020. Compositional generalization via neural-symbolic stack machines. In *Advances in Neural Information Processing Systems*, volume 33, pages 1690–1701. Curran Associates, Inc.

Róbert Csordás, Kazuki Irie, and Juergen Schmidhuber. 2021. The devil is in the detail: Simple tricks improve systematic generalization of transformers. In

Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, pages 619–634, Online and Punta Cana, Dominican Republic. Association for Computational Linguistics.

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 *Human Language Technology: Proceedings of a Workshop held at Plainsboro, New Jersey, March 8-11, 1994*.

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 (Volume 1: Long Papers)*, pages 351–360, Melbourne, Australia. Association for Computational Linguistics.

Daniel Furrer, Marc van Zee, Nathan Scales, and Nathanael Schärli. 2020. Compositional generalization in semantic parsing: Pre-training vs. specialized architectures. *CoRR*, abs/2007.08970.

Jiatao Gu, Zhengdong Lu, Hang Li, and Victor O.K. Li. 2016. Incorporating copying mechanism in sequence-to-sequence learning. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1631–1640, Berlin, Germany. Association for Computational Linguistics.

Yinuo Guo, Hualei Zhu, Zeqi Lin, Bei Chen, Jian-Guang Lou, and Dongmei Zhang. 2021. Revisiting iterative back-translation from the perspective of compositional generalization. In *Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021, Thirty-Third Conference on Innovative Applications of Artificial Intelligence, IAAI 2021, The Eleventh Symposium on Educational Advances in Artificial Intelligence, EAAI 2021, Virtual Event, February 2-9, 2021*, pages 7601–7609. AAAI Press.

Charles T. Hemphill, John J. Godfrey, and George R. Doddington. 1990. The atis spoken language systems pilot corpus. In *Proceedings of the Workshop on Speech and Natural Language*, HLT '90, page 96–101, USA. Association for Computational Linguistics.

Jonathan Herzig and Jonathan Berant. 2021. Span-based semantic parsing for compositional generalization. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 908–921, Online. Association for Computational Linguistics.

Jonathan Herzig, Peter Shaw, Ming-Wei Chang, Kelvin Guu, Panupong Pasupat, and Yuan Zhang. 2021. Unlocking compositional generalization in pre-trained models using intermediate representations. *arXiv preprint arXiv:2104.07478*.

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

Daniel Keysers, Nathanael Schärli, Nathan Scales, Hylke Buisman, Daniel Furrer, Sergii Kashubin, Nikola Momchev, Danila Sinopalnikov, Lukasz Stafiniak, Tibor Tihon, Dmitry Tsarkov, Xiao Wang, Marc van Zee, and Olivier Bousquet. 2020. Measuring compositional generalization: A comprehensive method on realistic data. In *International Conference on Learning Representations*.

Najoung Kim and Tal Linzen. 2020. COGS: A compositional generalization challenge based on semantic interpretation. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 9087–9105, Online. Association for Computational Linguistics.

Brenden Lake and Marco Baroni. 2018. Generalization without systematicity: On the compositional skills of sequence-to-sequence recurrent networks. In *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 2873–2882. PMLR.

Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. 2020. BART: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 7871–7880, Online. Association for Computational Linguistics.

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.

Inbar Oren, Jonathan Herzig, and Jonathan Berant. 2021. Finding needles in a haystack: Sampling structurally-diverse training sets from synthetic data for compositional generalization. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 10793–10809, Online and Punta Cana, Dominican Republic. Association for Computational Linguistics.

Inbar Oren, Jonathan Herzig, Nitish Gupta, Matt Gardner, and Jonathan Berant. 2020. Improving compositional generalization in semantic parsing. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 2482–2495, Online. Association for Computational Linguistics.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research*, 21(140):1–67.

Laura Ruis, Jacob Andreas, Marco Baroni, Diane Bouchacourt, and Brenden M Lake. 2020. A benchmark for systematic generalization in grounded language understanding. In *Advances in Neural Information Processing Systems*, volume 33, pages 19861–19872. Curran Associates, Inc.

Peter Shaw, Ming-Wei Chang, Panupong Pasupat, and Kristina Toutanova. 2021. Compositional generalization and natural language variation: Can a semantic parsing approach handle both? In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 922–938, Online. Association for Computational Linguistics.

Yushi Wang, Jonathan Berant, and Percy Liang. 2015. Building a semantic parser overnight. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 1332–1342, Beijing, China. Association for Computational Linguistics.

Silei Xu, Giovanni Campagna, Jian Li, and Monica S. Lam. 2020. *Schema2QA: High-Quality and Low-Cost Qamp;A Agents for the Structured Web*, page 1685–1694. Association for Computing Machinery, New York, NY, USA.

A Datasets

In this appendix we enumerate the different preprocessing steps, anonymization functions and specific evaluation methods used for each dataset, if any, and provide the number of splits/instances created in each splitting method. Note that the described anonymization functions are used only in the template method splits, and are not used in the grammar splits or adversarial splits.

A.1 COVR

Generation The COVR dataset that we use is generated with a manually written grammar that is based on the visual question answering (VQA) dataset of the same name (Bogin et al., 2021a), with utterances that contain either true/false statement or questions regarding different objects in an image. However, the dataset we use is separate from the VQA dataset since it generates only the utterances and the executable programs, without any dependence on a scene graph or image. While we use a grammar to generate the pairs, during generation some pre-defined pruning rules prevent specific cases where illogical programs are produced. The entire grammar, pruning rules and generation code are available in our codebase.

Anonymization We anonymize the following groups of symbols (symbols in each group are replaced with a group-specific constant): numbers, entities (`dog`,`cat`, `mouse`, `animal`), relations (`chasing`, `playing_with`, `looking_at`), types of attributes (`color`, `shape`), attribute values (`black`, `white`, `brown`, `gray`, `round`, `square`, `triangle`) and logical operators (`and`, `or`).

A.2 Overnight

Preprocessing We remove redundant parts of the programs, namely, the scope prefix that is used in functions and entities (`edu.stanford.nlp.sempre.overnight.SimpleWorld.XXX`) and declarations of types (`string`, `number`, `date`, `call`). The *regex* dataset is not used due to parsing issues.

Anonymization We anonymize the following groups of symbols: strings, entities and numbers. We use the type declaration that are removed in the preprocessing part to identify these groups.

Evaluation When evaluating the synthetic versions of Overnight, we did not encounter any evaluation issues. However, on the paraphrased version, some of the instances yielded false negatives, mostly due to inconsistent order of filter conditions. For example, the program for the paraphrased utterance *"find an 800 sq ft housing unit posted on january 2"* first defines the posting date as a condition, and only then the square feet size. This could happen whenever the crowdsource workers changed the order of the conditions as part of paraphrasing. In such cases, most of the times, models output conditions in the exact order given in the utterance, which would result in a negative exact match accuracy, even though the program is essentially correct. We normalize the order of the conditions to prevent such cases. We have encountered other evaluation issues as well which we did not address, since they were not as common.

A.3 S2Q

Preprocessing The raw generated S2Q dataset provided by Oren et al. (2021) has several issues which made i.i.d results of different models to be too low (due to issues that are not related to compositionality, described below), and n-LSs to be sometimes non-meaningful (since the Thingtalk language is not entirely functional). Thus, we perform several preprocessing steps that were not necessary for the other dataset.

First, similar to Overnight, we remove redundant parts that define scope (e.g. `@org.schema.Person.Person` is replaced with `Person`). Next, we perform slight changes to the programs of S2Q such that their parsed tree better describes the hierarchy of the function and arguments. For example, see Tab. 8 (top), where we replace the positions of `filter` and `Person`, such that `filter` will be the function that calls `Person` as its argument. Next, since S2Q programs were generated with thousands of different random entities names which are often interleaved in a non-natural way in the utterance (e.g. *"people which are alumni of seems like some people , , with job title containing clarifier , or"*, where *"seems like some people"* and *"clarifier , or"* are entities), we anonymize these strings by replacing their occurrences in the program with a constant value `STR_VAL` (we do so for numbers as well). To anonymize the utterance, we do not replace string values

Orig. program	`now => ( @org.schema.Person.Person ) filter count ( param:award:Array (String) ) >= 8 => notify`
Preprocessed	`filter ( Person ) ( count ( award ) >= NUMBER_VAL )`
Orig. utterance	*which are the person which have either* ***learning of true university*** *or* ***ky.*** *in the works for and having job title containing* ***animal health technician*** *that have* ***alger lake , mi***
Anonymized	*which are the person which have either* ***worksFor*** *or* ***worksFor*** *in the works for and having job title containing* ***jobTitle*** *that have* ***workLocation***

Table 8: An example for the preprocessing and anonymization we perform for the S2Q dataset. In the top example, we remove redundant parts, anonymize numbers and do minor modifications such that `filter` will be the function that calls the other arguments. In the bottom example, we anonymize the utterance to prevent ambiguity for the column type of the entity *alger lage , mi* (see description in text). Entities are bold.

with just a constant value, due to another related issue: in some cases, filter conditions are used in the program without the utterance mentioning the relevant column that should be filtered. For example, the phrase *"which are the person ... that have alger lake , mi"* refers to people that have *"alger lake , mi"* in the column `workLocation`, however, the usage of this specific column cannot be inferred from the name of the entity. We thus replace every string value in the utterance with the name of the column that it should use. See Tab. 8 (bottom) for an example. Note that unlike the other datasets, here anonymization is used not only for the cause of template splitting, but rather models are trained and evaluated with these anonymized utterances and programs.

Anonymization String values and numbers are already anonymized, as described above. For the purpose of template splitting, we additionally replace the names of fields (e.g. `worksFor`, `alumniOf`, `faxNumber`, etc.) in the programs with a constant value, and the different operators (==, >=, <=, =, `contains`, `asc` and `desc`) as well.

Evaluation We normalization S2Q programs during evaluation to prevent false negative cases, where the predicted program is different than the gold, but the two have the same meaning (i.e. would provide the same answer given any set of input). We address two specific cases. First, we normalize `and` and `or` clauses, such that clauses are sorted by alphabetic order while taking into account the precedence between the two operators. Second, we remove a redundant call of the `compute count` function that is used twice and can thus be ommitted (see our codebase for exact implementation).

A.4 ATIS

Preprocessing The ATIS dataset contains numbered variables (`$0`, `$1`, etc). However, it has some inconsistencies in the way these variables are given: sometimes the number appears directly after the dollar sign (`$0`), and sometimes the letter "v" appears between them (`$v0`). Additionally, we standardize the order of these numbers by setting the number n_i of the i-th variable (where the order of variables is defined by the position of their first appearance in the program), will be equal to $n_{i-1} + 1$, except for the first variable, which is 0, so that variables will be numbered in a consistent manner.

B Splits

B.1 Template Split

We generate template splits using the method described in §2.1, and the anonymization functions described in App. A. To make sure the distribution of number of examples per template is not very skewed, we limit the number of examples for each program template to $k_{\text{train}} = 1000$ in the training set. In case there are more, we randomly sample k_{train} examples for that template and discard the rest. We follow the same procedure for the test split, with $k_{\text{test}} = 10$.

B.2 Grammar Split

In §2.1 we define how a split is created given a set of pairs of grammar rules, $\mathcal{U}$, and as explained, we repeat this process with different instances of $\mathcal{U}$, where for each $\mathcal{U}$ we create a new split. Next, we explain how we pick different instances of $\mathcal{U}$.

A grammar rule $r_i = A \rightarrow \gamma \in \mathcal{G}$ is comprised of its left hand side (LHS) A, which is a non-terminal, and its right hand side (RHS) γ which is a sequence of terminals and/or non-terminals. In

$\mathcal{G}$:
```
...
(1) number_objects → count ref
(2) count → "count"
(3) boolean_pair → boolean_and
(4) boolean_pair → boolean_or
(5) boolean_single → exists ref
(6) boolean_single → compare_count
...
```

$\mathcal{L}$:
```
(1) boolean_pair
(2) boolean_single
```

l_1: `boolean_pair`

l_2: `boolean_single`

$\mathcal{G}_{l_1}$:
```
(1) boolean_pair → boolean_and
(2) boolean_pair → boolean_or
```

$\mathcal{G}_{l_2}$:
```
(1) boolean_single → exists ref
(2) boolean_single → compare_count
```

$\mathcal{U}_{\text{comb}}$:
```
(1) (boolean_pair → boolean_and,
     boolean_single → exists ref)
(2) (boolean_pair → boolean_and,
     boolean_single → compare_count)
(3) (boolean_pair → boolean_or,
     boolean_single → exists ref)
(4) (boolean_pair → boolean_or,
     boolean_single → compare_count)
```

(Output)

$\mathcal{U}_1$:
```
(1) (boolean_pair → boolean_and,
     boolean_single → exists ref)
(2) (boolean_pair → boolean_and,
     boolean_single → compare_count)
(3) (boolean_pair → boolean_or,
     boolean_single → exists ref)
(2) (boolean_pair → boolean_or,
     boolean_single → compare_count)
```

$\mathcal{U}_2$:
```
(1) (boolean_pair → boolean_and,
     boolean_single → exists ref)
```

$\mathcal{U}_3$:
```
(1) (boolean_pair → boolean_and,
     boolean_single → compare_count)
```

$\mathcal{U}_4$:
```
(1) (boolean_pair → boolean_or,
     boolean_single → exists ref)
```

$\mathcal{U}_5$:
```
(1) (boolean_pair → boolean_or,
     boolean_single → compare_count)
```

$\mathcal{U}_6$:
```
(1) (boolean_pair → boolean_and,
     boolean_single → exists ref)
(2) (boolean_pair → boolean_and,
     boolean_single → compare_count)
```

$\mathcal{U}_7$:
```
(1) (boolean_pair → boolean_or,
     boolean_single → exists ref)
(2) (boolean_pair → boolean_or,
     boolean_single → compare_count)
```

Figure 5: An illustration of our grammar split. Bottom part shows 7 produced different splits. See text for details.

theory, we could have iterated over all possible *sets* of pairs, and create a split out of each of these sets, but this would be impractical. Instead, we propose the following method to to pick different instances of $\mathcal{U}$.

First, we consider only the set $\mathcal{L}$ of non-terminals that are *"meaningful"*. We consider a non-terminal l to be meaningful iff (1) there are at least two rules in $\mathcal{G}$ where l is their LHS, or (2) l is a non-terminal such that for at least two different rules $r_a, r_b \in \mathcal{G}$, l belongs to the RHS sequence of both r_a and r_b. By selecting only meaningful non-terminals, we can avoid considering redundant rules that always appear together with other rules. For example, in Fig. 5, the rule `count` is a non-terminal of just a single rule, and it belongs to the RHS of just a single rule, thus it is not considered meaningful. In this example, $\mathcal{L}$ will consist of the meaningful non-terminals `boolean_pair` and `boolean_single`.

We then iterate over all possible unique pairs (l_1, l_2) of non-terminals in $\mathcal{L}$. For each such pair, we take the set $\mathcal{G}_{l_1}$ of grammar rules for which l_1 is the LHS of, and the set $\mathcal{G}_{l_2}$ of grammar rules for which l_2 is the LHS of. In the example in Fig. 5, we will have one possible such pair of non-terminals (l_1, l_2), `boolean_pair` and `boolean_single`. The figure shows the set $\mathcal{G}_{l_1}$ that contains all rules with `boolean_pair` as its non-terminal, and similarly for $\mathcal{G}_{l_2}$.

Given the two sets of rules $\mathcal{G}_{l_1}$ and $\mathcal{G}_{l_2}$, we set $\mathcal{U}_{\text{comb}}$ to be the set of all possible combinations from $\mathcal{G}_{l_1}$ and from $\mathcal{G}_{l_2}$ (the Cartesian product of the two sets). Next, we use the following pseudo code to generate different instances of $\mathcal{U}$ (each set that we yield is a different generated instance of $\mathcal{U}$):

1: yield $\mathcal{U}_{\text{comb}}$
2: **for** $(r_1, r_2) \in \mathcal{U}_{\text{comb}}$ **do**
3:    yield $\{(r_1, r_2)\}$
4: **end for**
5: **for** $r \in (\mathcal{G}_{l_1} \cup \mathcal{G}_{l_2})$ **do**
6:    yield $\{(r_1, r_2) \in \mathcal{U}_{\text{comb}}$ if $r_1 = r$ or $r_2 = r\}$
7: **end for**

Following the figure, $\mathcal{U}_6$ will be generated in line 1, $\mathcal{U}_{2..5}$ in line 3, and $\mathcal{U}_{6..7}$ in line 6. Finally, for each instance of generated $\mathcal{U}$, we discard it if it generates an invalid split (where symbols in the test set do not occur in the training set).

Example grammar splits We show a few selected examples of the generated grammar splits in

$\mathcal{U}$	1. (eq → 'eq', boolean_pair → boolean_or) 2. (eq → 'eq', boolean_pair → boolean_and)
Train examples	1. *either there is black cat or all of mouse are looking at animal* **or**(exists(filter(black,find(cat))),all(find(mouse),with_relation (scene(),looking_at,find(animal)))) 2. *the number of black dog is equal to the number of white cat* **eq**(count(filter(black,find(dog))),count(filter(white,find(cat))))
Test example	*either the color of cat that is playing with cat is equal to black or there is mouse* **or**(**eq**(query_attr[color](with_relation(find(cat),playing_with,find(cat))), black),exists(find(mouse)))
Predicted easiness for split (2-LS)	0.09
$\mathcal{U}$	1. (ref → with_relation, ref → filter_object)
Train examples	1. *the number of white cat is less than the number of gray mouse* lt(count(**filter**(white,find(cat))),count(**filter**(gray,find(mouse)))) 2. *is the color of cat that is looking at animal black or white ?* choose(query_attr[color](**with_relation**(find(cat),looking_at ,find(animal))),black,white)
Test example	1. *the number of animal that is looking at white animal is greater than 4* gt(count(**with_relation**(find(animal),looking_at,**filter**(white, find(animal)))),4)
Predicted easiness for split (2-LS)	0.43
$\mathcal{U}$	1. (boolean_single → exists ref, boolean_pair → boolean_or)
Train examples	1. *either some of animal are brown or the number of cat is less than 2* **or**(some(find(animal),filter(brown,scene())),lt(count(find(cat)),2)) 2. *both the number of dog that is chasing cat that is chasing dog is equal to 4 and there is brown mouse that is chasing mouse* and(eq(...),**exists**(with_relation(filter(brown,find(mouse)),chasing, find(mouse))))
Test example	1. *either most of round white mouse are square or there is animal that is playing with cat* **or**(most(...),**exists**(with_relation(find(animal),playing_with ,find(cat))))
Predicted easiness for split (2-LS)	0.94

Table 9: Selected splits generated with the grammar split method, with selected examples for each split.

Table 9.

B.3 Sizes of Splits

For each dataset, we list the number of splits, the total number of instances over which we compute AUC scores, and the average number of train/test templates (after anonymization) in Table 10.

B.4 n-LS Splits

In §6.1 we describe how we create adversarial splits using n-LSs, and mention that we discard any splits if their easiness score is higher than a threshold τ. We choose this threshold by using the n-LS classifiers we evaluate in §5.1, for which we have shown AUC scores in Table 3. For each decision rule and for each dataset, we find an optimal threshold that optimizes the F_1 score on our test instances, and use it as the threshold τ for the matching n-LS split.

For 2-LS, COVR, $\tau = 0.58$ (11 splits generated, none discarded) and for Overnight, $\tau = 0.13$ (112 splits are generated, 38 kept after filtering, 15 are sampled for experiments). For 2-LS-NoSib, COVR, $\tau = 0.3$ (5 splits generated, 1 split is discarded) and for Overnight, $\tau = 0.2$ (44 splits are generated, 11 kept after filtering). For 2-LS-NoSib-Half, COVR, we do not use τ, but instead we take the top 15 hardest splits, according to the easiness score. We cannot use τ in these cases since all splits were higher than the thresholds (by the definition of our decision rule, easiness score is high in splits where all unobserved structures have similar observed structures).

Dataset	Split	# Splits	# Total instances	Avg # training templates / ex.	Avg # test templates / ex.
COVR	Grammar	124	62,000	1,510.0 / 3000.0	372.5 / 500.0
Overnight	Template	5	2,707	43.9 / 1,491.8	10.8 / 215.0
S2Q	Template	5	4,089	42.0 / 15,940.2	97.0 / 831.6
Atis	Template	5	2,587	920.0 / 4,026.2	229.0 / 772.8

Table 10: Different splits statistics, including the number of generated splits per each split method and dataset, the total number of test instances across all splits and average train/test templates and examples across the splits.

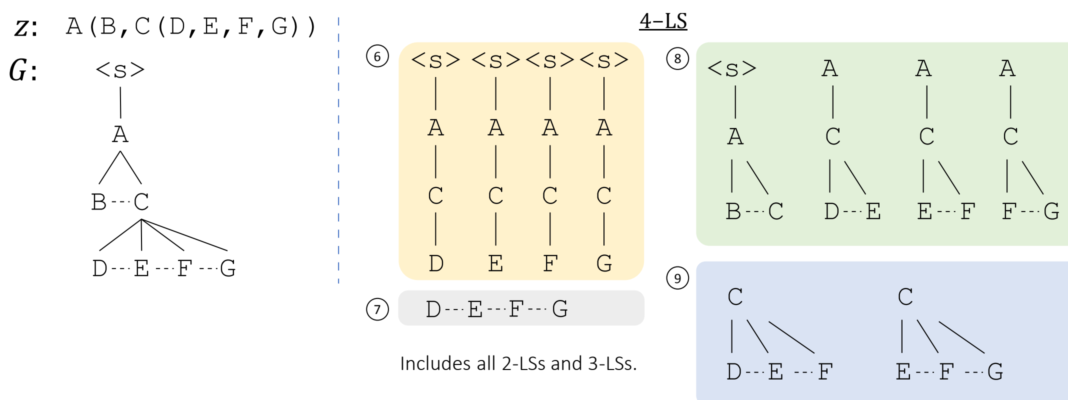

Figure 6: An example program z and the structure of its program graph G (left), with solid edges for parent-child relations and dashed edges for consecutive siblings. The right side enumerates all 4-LS structures over this graph.

C 4-LS

In §4.1, we define 2-LS and 3-LS over the program graph G. The structure 4-LS is a natural extension of these structures. It includes all 2-LSs, and 3-LS, and also structures with (1) three parent-child relations, (2) three siblings relations and (3) a grandparent with its child and two sibling grandchildren and (4) a parent with three siblings (structures 6, 7, 8 and 9 in Figure 6, respectively).

D Measuring TMCD

We measure compound divergence of the distributions of compounds and atoms on the program graph, following Keysers et al. (2020) and Shaw et al. (2021). Similarly to TMCD, we define atoms and compounds over the program tree T, over the graph defined in §4.1. Each tree node is considered an atom, and the compounds are all sub-trees of up to depth 2. We use the same Chernoff coefficient as in the original paper, $\alpha = 0.5$, to compute compound divergence.