COMPOSITIONAL GENERALIZATION AND DECOMPOSITION IN NEURAL PROGRAM SYNTHESIS

Kensen Shi
Google Research
kshi@google.com

Joey Hong
UC Berkeley
joey_hong@berkeley.edu

Manzil Zaheer
Google Research
manzilzaheer@google.com

Pengcheng Yin
Google Research
pcyin@google.com

Charles Sutton
Google Research
charlessutton@google.com

ABSTRACT

When writing programs, people have the ability to tackle a new complex task by decomposing it into smaller and more familiar subtasks. While it is difficult to measure whether neural program synthesis methods have similar capabilities, what we can measure is whether they compositionally generalize, that is, whether a model that has been trained on the simpler subtasks is subsequently able to solve more complex tasks. In this paper, we focus on measuring the ability of learned program synthesizers to compositionally generalize. We first characterize several different axes along which program synthesis methods would be desired to generalize, e.g., length generalization, or the ability to combine known subroutines in new ways that do not occur in the training data. Based on this characterization, we introduce a benchmark suite of tasks to assess these abilities based on two popular existing datasets, SCAN and RobustFill. Finally, we make first attempts to improve the compositional generalization ability of Transformer models along these axes through novel attention mechanisms that draw inspiration from a human-like decomposition strategy. Empirically, we find our modified Transformer models generally perform better than natural baselines, but the tasks remain challenging.

1 INTRODUCTION

Program synthesis aims to assist programmers by automatically producing code according to a user's specification of what the code should do (Gulwani et al., 2017a). Search-based program synthesis approaches, such as programming by example (PBE) systems, have been effective for small self-contained tasks such as short Java functions (Shi et al., 2019), string manipulation (Gulwani, 2011; Devlin et al., 2017; Shi et al., 2022), and tensor manipulation (Shi et al., 2020). However, synthesizing complex or long programs can be expensive because the search space grows exponentially with respect to the program length. Furthermore, many search-based approaches require significant engineering effort to adapt to new programming libraries or languages. Similarly, program synthesizers that use constraint solving (Solar-Lezama et al., 2006; Torlak & Bodík, 2013) are successful in narrow domain-specific languages (DSLs), but they often become intractable for longer programs and can be difficult to extend beyond simple DSLs. Neural program synthesizers, especially those based on large language models (Chen et al., 2021; Austin et al., 2021; Li et al., 2022), can produce longer or more complex code at a lower computation cost, but their successes are often limited to examples similar to those present in the training data (Furrer et al., 2020). That is, they do not generalize well to new APIs, novel concepts, or even novel *combinations* of concepts.

It is desirable for program synthesizers to generalize in many ways. For example, an ideal synthesizer would produce longer code without a prohibitive increase in computational cost or a dramatic decrease in code quality. It would adapt its general programming skills to handle new APIs and concepts with minimal extra guidance or engineering. It would also be able to mix and match programming concepts, composing different code idioms in novel ways to solve novel problems. These are all *compositional generalization* skills that human programmers naturally develop but are often difficult for program

synthesizers. Compositional generality means the ability to generalize to test examples consisting of compositions of components seen during training, but where the distribution of compositional patterns is different (Keysers et al., 2020). While current program synthesizers are far from reaching these lofty goals, we can measure the compositional generalization abilities of different synthesis techniques to help push the state-of-the-art toward these desirable human-like abilities.

Prior work has evaluated whether natural language processing systems can compositionally generalize, proposing benchmark datasets to measure the ability of language understanding models in interpreting learned concepts, e.g., *jump*, in compositionally novel contexts, e.g., *jump twice* (Marcus, 2001; Lake & Baroni, 2018). We adapt those ideas to focus on how problem-solving in the form of *programming* is compositional. For instance, complex computer programs are typically built by composing individual functions and API calls, which can be composed in novel ways to solve novel problems. In this paper, we identify seven different compositional generalization tasks applicable to program synthesis and propose a new method of creating benchmark datasets that measure these forms of compositional generalization, for both zero-shot and few-shot generalization. We apply our benchmark-creation method to two popular domains: SCAN (Lake & Baroni, 2018), which involves generating a sequence of actions specified by a natural language-like command, and RobustFill (Devlin et al., 2017), which targets string manipulation programs specified by input-output examples. Our benchmark-creation method is agnostic to the kind of program specification used, and our RobustFill-based benchmark is the first compositional generalization dataset using input-output examples to our knowledge, making it particularly applicable to program synthesis.

In addition to proposing benchmark datasets to measure the compositional generality of program synthesizers, we furthermore hypothesize that *decomposition* is particularly useful for achieving this compositional generality. Decomposition is the problem-solving technique (broadly applicable even beyond programming) of breaking a complex task into multiple smaller parts, perhaps repeatedly, until each subtask is easy enough to handle. Decomposition is especially important within software engineering where implementations of subtasks can be combined and reused in modular ways. Applying decomposition is a skill so fundamental to software engineering that the first programming course at Stanford University begins teaching decomposition within the first week of class, immediately after introducing basic syntax, the concept of functions, and simple control flow (Parlante, 2022). Because compositional generality revolves around combining ideas in new ways, and the decomposition approach solves problems by isolating subtasks and combining their solutions, we argue that a decompositional programming strategy is likely to have high compositional generality (although this is not necessarily the only viable strategy). Hence, we propose variations of the Transformer architecture motivated by the decomposition strategy, where the model is trained to recognize boundaries between subtasks and focus on solving one subtask at a time. As a bonus, well-decomposed code is a hallmark of good coding style, so it is additionally desirable to encourage synthesizers to produce such code.

In our experiments, we find that our decomposition-based Transformer variations outperform the vanilla Transformer architecture for some but not all of the compositional generalization tasks, with a greater improvement over the baseline for SCAN than for RobustFill. Even so, our compositional generalization benchmarks remain difficult overall in both the zero-shot and few-shot settings. We hope that the datasets inspire continued research using different kinds of techniques toward the goal of compositionally general program synthesis.

2 Compositional Generalization in Programming

The goal in program synthesis is to find a program in a given language that is consistent with a specification. Formally, we are given a domain specific language (DSL) which defines a space $\mathcal{P}$ of programs. The task is described by a specification $X \in \mathcal{X}$ and is solved by an unknown program $P^* \in \mathcal{P}$. For example, a specification can be a set of input/output (I/O) examples denoted $X = \{(I_1, O_1), \ldots (I_N, O_N)\}$. Then, solving specification X means finding a program P (not necessarily P^*) that correctly solves all of the examples: $P(I_i) = O_i, \forall i$. A specification can also be a natural language description of a task, and the corresponding program implements said task.

Program synthesizers are more robust and more broadly applicable if they generalize well. In this section we discuss several distinct forms of generalization that are desirable in program synthesis.

Current program synthesizers do not achieve high generalization in all of these ways. At a high level, we identify three broad categories of generalization that an "ideal program synthesizer" should have:

- *Length generalization*: Produce longer code than appeared in the training set, but without a prohibitive increase in computational cost, and without a substantial decrease in quality.
- *Mix and match concepts*: Compose code idioms in novel ways to solve novel problems, but without combinatorially many training examples covering all combinations.
- *Apply general principles*: Adapt to new, updated, or custom APIs by drawing on knowledge of other similar APIs, without excessive guidance or engineering.

These kinds of generalization can be described as *compositional generalization*, which revolves around understanding how basic building blocks can be composed in different ways to create larger structures. Prior work in natural language processing has studied compositionality in natural language (Chomsky & Lightfoot, 2002; Talmor & Berant, 2018; Lake & Baroni, 2018; Keysers et al., 2020; Gu et al., 2021). For example, the SCAN dataset (Lake & Baroni, 2018) tests compositional generalization for translation models. The SCAN task is to translate from a language-like command such as "jump left twice and walk" to a sequence of actions, in this case `[LTURN, JUMP, LTURN, JUMP, WALK]`. One compositional generalization task would be to train a model on commands except for those including "jump right", and then test on commands containing "jump right". This zero-shot generalization task requires the model to understand "jump" and "right" individually, as well as the compositional pattern of an action verb followed by a direction. Such understanding can be drawn from other training examples including "jump left", "walk left", and "walk right".

We adapt compositional generalization to the context of program synthesis, focusing on how problem-solving (in the form of programs) is compositional. Regardless of the programming language or DSL, programs nearly always consist of compositions of smaller parts. In general, a "program part" could mean a line of code, a block of statements, a function, or some other natural DSL-specific portion of code, where multiple such portions can be combined into a full program. We can even view SCAN action sequences as programs (with SCAN commands being the specification). For SCAN, we choose to define "parts" to be the portions of the action sequence that were separated by conjunctions ("and" and "after") in the command. In the example above, the program parts would be `[LTURN, JUMP, LTURN, JUMP]` and `[WALK]`, corresponding to "jump left twice" and "walk" from the command. We use the notion of program parts to study how they are combined with different compositional patterns. Thus, the conventions for partitioning a program into its composed parts may vary depending on the DSL and the compositional patterns being tested.

We expand the three broad categories of generalization above into 7 concrete compositional generalization tasks applicable to program synthesis. Each generalization task describes a method of creating training and test sets with disjoint distributions, so that generalization may be tested in a zero-shot or few-shot setting. The generalization tasks are as follows:

1. ***Length-Generalization***: Can a model *produce longer code* than seen in training, when the problem requires it? Here, "length" is measured by the number of composed parts in the program, not simply the number of tokens, so there is more emphasis on generalizing to more complex compositional patterns. For this generalization task, we train on problems of lengths 1 to n and test on lengths $n+1$ to m (where $m > n$). In our experiments we choose $n = 6$ and $m = 10$.
2. ***Length-Generalization-Hard***: Similar to above, but train on problems of length exactly n and test on lengths 1 to m except n. To succeed on this generalization task, the synthesizer must recognize that problems may have varying difficulty with corresponding solutions of varying lengths, without seeing this fact demonstrated during training.
3. ***Length-Generalization-Hardest***: Similar to above, but train on tasks of length exactly 1 and test on lengths 2 to n. Because the training data has no examples of how program parts may be composed, we do not expect neural models to achieve high zero-shot generalization of this kind. Few-shot generalization is more interesting for this task—after mastering how to solve individual parts of a program, the synthesizer must quickly learn the compositional patterns for composing those parts into larger programs.
4. ***Compose-Different-Concepts*** (a form of "mix and match concepts"): Can a model *use concepts in different combinations* than seen in training? We partition the DSL operations into multiple groups

or *concepts*,[1] train on compositions of operations from the same concept, and test on compositions from different concepts. For example, if two concepts consist of operations $\{A_1, A_2, \ldots\}$ and $\{B_1, B_2, \ldots\}$ respectively, then this generalization task involves training on programs of the forms $A_i \circ A_j$ and $B_i \circ B_j$, and testing on the forms $A_i \circ B_j$ and $B_i \circ A_j$. As a real-world example, this generalization task is similar to training on scripts containing only TensorFlow or only NumPy, but synthesizing code using both libraries.

5. ***Switch-Concept-Order*** (a form of "mix and match concepts"): Can a model *compose concepts in different orders* than seen in training? We again partition the DSL operations into multiple concepts (groups). We train on compositions of operations drawn from one sequence of concepts and test on a different sequence of concepts, e.g., train on $A_i \circ B_j$ and test on $B_i \circ A_j$. As a real-world example, in the training data a function might be primarily used to validate inputs at the beginning of the code, but we want to use the function in a different context, e.g., to validate results at the end of the code.
6. ***Compose-New-Operation*** (a form of "apply general principles"): Can a model learn to *use a new isolated operation within a larger composition*? In this task, we train on the isolated operation and compositions without the operation, and test on compositions using the operation. For instance, in the SCAN domain, we could train on "walk left after run twice" and "jump", and test on "jump left after jump twice". A real-world example of this kind of generalization would be composing a new function with others in a larger solution, after seeing examples of the function used in isolation.
7. ***Add-Operation-Functionality*** (a form of "apply general principles"): Can a model *extend its understanding of an operation by drawing on parallels* to other operations? We omit from the training data some functionality of an operation that could be inferred from other context, and test on programs using that functionality. For instance, in the SCAN domain, we could train on commands that do not contain "around right" (but contain other similar constructions like "opposite left" and "opposite right"), and test on commands containing "around right". This task can occur in the real world when a library function is upgraded with a new parameter whose behavior can be inferred from other functions.

3 Benchmark Creation

We create benchmark datasets for the 7 kinds of compositional generalization tasks described in Section 2 for two popular domains, SCAN (Lake & Baroni, 2018) and RobustFill (Devlin et al., 2017). While Section 2 focused on general descriptions of the tasks, this section instantiates these tasks for our specific domains.

Our benchmark creation process works for both natural-language specifications (as in SCAN) and input-output (I/O) specifications (as in RobustFill). While the SCAN domain was used in prior work in natural language processing (Lake & Baroni, 2018), we have expanded the domain to make our benchmark more applicable to program synthesis. Furthermore, our RobustFill benchmark is the first dataset for measuring compositional generalization for program synthesis from I/O examples.

In the SCAN domain, the objective is to translate from a natural-language command to a program that is a sequence of actions. Lake & Baroni (2018) originally describes SCAN commands as having at most one "and" or "after" conjunction, but we generalize the domain so that commands can contain an arbitrary number of "and" and "after" conjunctions between parts of the command.[2] We treat these conjunctions as the boundaries between program parts, so a command with n parts will have $n-1$ conjunctions and should be translated to an action sequence containing n corresponding parts, although corresponding command and action sequence parts will appear in different orders whenever there is an "after" conjunction. We show the DSL for commands, as well as how commands are translated to programs in Appendix A.

In the RobustFill domain, the objective is to synthesize a string manipulation program from I/O examples. A RobustFill program is a concatenation of expressions, where an expression may be an

[1] Ideally, operations within a group should have meaningful commonalities that form one concept, and each concept should have roughly equal semantic complexity, but these are not strictly required.

[2] To eliminate ambiguity in the correct ordering of parts, we say that "and" has higher precedence than "after". For example, "jump and run after walk" should be translated to `[WALK, JUMP, RUN]`, and *not* `[JUMP, WALK, RUN]`.

operation that extracts a substring from the input, an operation that returns a modified version of the input, a special Compose operation (applying a modification operation to the result of another operation), or a constant string. See Appendix A for the full DSL of how programs are generated. Due to this program structure, we treat each expression as a *program part*.

Appendix B provides more details for the setup of specific generalization tasks for both domains.

4 MODELS

We approach our compositional synthesis benchmarks using a sequence-to-sequence (seq2seq) model, which has been shown to be successful on various natural language (Bahdanau et al., 2016; Vaswani et al., 2017) and program synthesis tasks (Parisotto et al., 2017; Devlin et al., 2017). In this paper, we choose our seq2seq model to be a Transformer due to its impressive performance on natural language tasks over traditional RNNs (Vaswani et al., 2017). Section 4.1 describes a baseline Transformer adapted to program synthesis. In Section 4.2, we present modifications to the baseline model to encourage decomposition. We call the modified architecture the *Decompositional Transformer*.

4.1 BASELINE TRANSFORMER

Our baseline Transformer consists of two modules. First, a Transformer encoder receives the specification X word-by-word and produces an encoding, $E \leftarrow \text{TransformerEncoder}(X)$. Then, a Transformer decoder takes the encoding and autoregressively generates a program token-by-token. Formally, let $P_{t-1} = [p_1, p_2, \ldots, p_{t-1}]$ be the program generated so far. The decoder predicts the next program token as $p_t \leftarrow \text{TransformerDecoder}(P_{t-1}, E)$. As described by Vaswani et al. (2017), the Transformer encoder and decoder both apply a stack of self-attention and feed-forward units.

In the case of specification X being multiple I/O examples, our Transformer architecture performs *double attention* analogous to Devlin et al. (2017). That is, for each example (I_i, O_i), the encoder behaves as $E_i \leftarrow \text{TransformerEncoder}(I_i, O_i)$, where the encoder now performs self-attention on input I_i followed by cross-attention on output O_i to I_i. Finally, the encoding E is simply the concatenation across examples $E \leftarrow \text{Concat}((E_i)_{i=1}^{N})$ where N is the number of examples.

Relative attention. Early self-attention mechanisms have added representations of the absolute positions of tokens to its inputs (Vaswani et al., 2017). However, we use representations of relative positions, or distances between tokens, in line with recent work showing that relative attention is advantageous, particularly on length-generalization tasks (Shaw et al., 2018; Csordás et al., 2021). By considering logarithmic distances, our model is also encouraged to attend to more recent tokens during decoding, which can be desirable when programs consist of multiple smaller parts.

4.2 DECOMPOSITIONAL TRANSFORMER

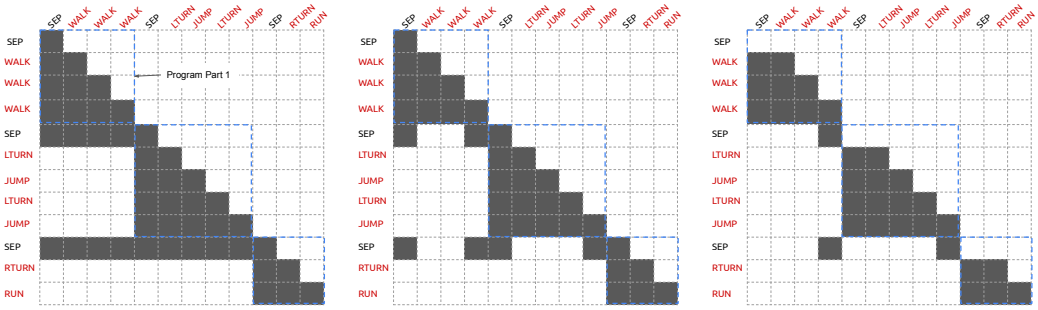
Inspired by the human problem-solving and programming strategy of decomposition, our Decompositional Transformer architecture leverages the baseline architecture but ensures that program parts are decoded independently of one another, using novel attention mechanisms. We accomplish this decomposition using two notable modifications to the baseline Transformer architecture:

Subprogram separators. We introduce a new separator token `SEP` to the program vocabulary.[3] This token will partition programs into sequences of *program parts*. The program parts can be composed, for instance via concatenation, to form the full program. We can straightforwardly add such tokens into the training data generated as described in Section 3. In this manner, we provide explicit training supervision for the compositional nature of the ground-truth programs. Thus, our models are trained to predict a `SEP` token after completing each program part. To compensate during evaluation, we remove all `SEP` tokens from the generated program before evaluating its correctness.

Decompositional attention masks. We incorporate novel attention mechanisms to ensure that program parts are decoded separately. As in the baseline architecture, we first use a Transformer

[3]In practice, we found it sufficient to have the `SEP` token be the same `BOS` token that marks the beginning of programs. In this manner, we avoid introducing a new token to the vocabulary.

Specification: "jump left twice and run right after walk thrice"
Program: WALK WALK WALK LTURN JUMP LTURN JUMP RTURN RUN



(a) Sep-Full-Attention (b) Sep-to-Sep-and-Last (c) Sep-to-Last

Figure 1: Illustrations of our proposed attention mechanisms on an example SCAN program.

encoder to produce an encoding $E \leftarrow \text{TransformerEncoder}(X)$. Let P_{t-1} again be the program generated so far. In contrast to the baseline Transformer, during decoding, the next token in the program is instead generated as $p_t \leftarrow \text{TransformerDecoder}(Q_{t-1}, E)$, where we apply an attention mask to the self-attention layers so that $Q_{t-1} \subseteq P_{t-1}$ consists of tokens relevant to solving the current subtask. More specifically, if p_{t-1} is not SEP, then Q_{t-1} consists of the program part generated so far, namely all program tokens starting from the most recent SEP (inclusive). Now, if p_{t-1} is SEP, then our model must identify the next subtask to solve, which naturally should depend on the previously-solved subtasks. We propose three different choices of Q_{t-1} when p_{t-1} is SEP, where the more tokens Q_{t-1} contains, the more information we give to our model to identify the next subtask:

1. *Sep-Full-Attention*: In the most general case, we provide the Transformer decoder with the entire program generated so far, i.e., $Q_{t-1} = P_{t-1}$ when p_{t-1} is SEP.
2. *Sep-to-Sep-and-Last*: In this case, Q_{t-1} contains all previous SEP tokens, as well as the last program token in each previous program part. We use the last tokens because they attend to their entire respective program part during decoding, thus providing a summary of what that part does.
3. *Sep-to-Last*: In the most restrictive case, Q_{t-1} only contains the last program token in each previously generated program part.

We provide an illustration of each attention mask on an example program in Figure 1. Note that relative attention encourages a similar behavior in attending more strongly to recent tokens; however, our attention masks are stricter and explicitly constrain attention to only include the necessary tokens.

5 Experiments and Discussion

We trained the various methods in Section 4 on each compositional generalization task in the SCAN and RobustFill datasets described in Section 3. We trained with batch size 128 and default settings for hyperparameters (details in Appendix C), training each method 3 times with different random initializations. We used 10K training steps for SCAN and 1M training steps for RobustFill, and we generated enough synthetic training data to avoid repeating examples. After training for each generalization task, we evaluate on 10K test examples to measure zero-shot generalization. Then, we fine-tune the trained models on a single batch containing 20 examples from the training distribution and 20 from the test distribution, repeated for 30 steps. Finally, we evaluate again on the same 10K test examples to measure few-shot generalization. SCAN results are in Table 1 and RobustFill results are in Table 2. For a more visual representation, Appendix D contains bar graphs for the same data.

All three improvements to vanilla Transformer (relative attention, separator tokens, and attention masks) generally help overall. In the most extreme case, our best model using the Sep-to-Last attention mask has 74.4% zero-shot generalization for SCAN *Compose-Different-Concepts*, compared

		Length Generalization			Compose Diff. Concepts	Switch Concept Order	Compose New Operation	Add Operation Func.
		1-6 to 7-10	6 to 1-10	1 to 2-6				
Zero-shot	Transformer	26.9 ±0.7	36.3 ±0.6	1.1 ±0.2	10.5 ±2.4	4.6 ±2.5	7.9 ±2.2	41.7 ±1.6
	+ Separators ⋆	26.7 ±0.6	23.6 ±0.6	0.9 ±0.2	37.3 ±2.8	10.0 ±3.8	11.4 ±1.1	43.7 ±1.7
	Relative Attention	46.1 ±3.0	39.8 ±1.8	1.1 ±0.1	29.2 ±4.6	9.4 ±4.0	9.3 ±1.9	15.6 ±6.8
	+ Separators ⋆	42.1 ±3.5	33.8 ±1.9	1.1 ±0.1	46.7 ±4.3	9.8 ±4.8	**12.9** ±2.6	46.6 ±7.0
	Sep-Full-Attention ⋆	**50.2** ±2.0	**41.2** ±0.6	1.1 ±0.1	36.7 ±2.4	7.6 ±5.5	11.6 ±2.5	59.8 ±4.4
	Sep-to-Sep-and-Last ⋆	47.2 ±2.5	34.4 ±1.2	**1.2** ±0.1	49.0 ±6.6	15.4 ±4.0	10.3 ±1.2	**65.5** ±15.1
	Sep-to-Last ⋆	41.9 ±3.3	24.1 ±1.9	1.0 ±0.2	**74.4** ±9.3	**17.5** ±8.8	12.5 ±2.5	39.8 ±6.8
Fine-tuning	Transformer	28.7 ±1.7	61.0 ±1.6	1.1 ±0.2	56.5 ±6.4	48.3 ±2.4	74.8 ±0.6	89.6 ±1.6
	+ Separators ⋆	32.6 ±2.2	59.5 ±3.6	1.0 ±0.2	**89.3** ±1.7	58.4 ±4.8	88.0 ±1.0	97.6 ±0.1
	Relative Attention	65.3 ±1.1	67.5 ±6.4	0.9 ±0.3	53.0 ±3.8	54.2 ±4.5	79.4 ±2.8	85.6 ±3.6
	+ Separators ⋆	74.5 ±6.9	71.2 ±3.5	1.2 ±0.2	76.4 ±6.3	65.7 ±1.0	89.0 ±2.4	98.9 ±0.6
	Sep-Full-Attention ⋆	72.3 ±2.8	**76.7** ±1.4	**1.6** ±0.1	71.2 ±9.3	64.4 ±3.1	88.9 ±3.1	99.8 ±0.1
	Sep-to-Sep-and-Last ⋆	**75.0** ±2.3	71.5 ±2.8	1.2 ±0.1	72.3 ±8.8	66.7 ±3.6	93.2 ±1.0	99.8 ±0.0
	Sep-to-Last ⋆	70.8 ±4.5	69.5 ±2.0	1.5 ±0.2	84.9 ±5.2	**66.8** ±1.5	**95.7** ±0.6	**100.0** ±0.0

Table 1: **SCAN** zero-shot generalization and few-shot fine-tuning results, with $\pm\sigma$ denoting a standard deviation of σ over 3 trials. Methods marked with ⋆ are newly proposed in this work.

		Length Generalization			Compose Diff. Concepts	Switch Concept Order	Compose New Operation	Add Operation Func.
		1-6 to 7-10	6 to 1-10	1 to 2-6				
Zero-shot	Transformer	39.1 ±1.2	28.7 ±1.0	1.8 ±0.0	51.3 ±0.5	4.7 ±0.3	45.9 ±0.1	55.3 ±0.2
	+ Separators ⋆	28.5 ±2.4	24.7 ±0.6	1.8 ±0.0	55.7 ±1.7	**6.6** ±0.5	46.2 ±0.1	55.6 ±0.2
	Relative Attention	43.6 ±3.0	**30.8** ±0.8	1.8 ±0.0	56.1 ±2.6	6.5 ±0.5	46.2 ±0.1	55.9 ±0.2
	+ Separators ⋆	43.5 ±3.0	29.5 ±2.3	**1.9** ±0.0	61.1 ±2.6	5.4 ±0.5	**46.4** ±0.2	**56.1** ±0.2
	Sep-Full-Attention ⋆	**48.6** ±2.5	28.5 ±1.6	**1.9** ±0.0	58.1 ±6.9	5.1 ±0.2	**46.4** ±0.2	55.4 ±0.0
	Sep-to-Sep-and-Last ⋆	42.4 ±1.2	30.1 ±0.7	**1.9** ±0.0	60.4 ±1.3	6.4 ±0.6	46.2 ±0.2	55.4 ±0.1
	Sep-to-Last ⋆	40.7 ±1.2	24.1 ±3.3	1.8 ±0.0	**62.0** ±1.9	5.5 ±0.5	46.0 ±0.0	55.5 ±0.2
Fine-tuning	Transformer	58.3 ±0.1	60.6 ±1.8	1.8 ±0.2	91.9 ±0.8	51.4 ±3.0	51.3 ±0.0	66.4 ±0.3
	+ Separators ⋆	58.2 ±0.6	61.1 ±2.2	1.9 ±0.1	92.4 ±0.4	**53.6** ±2.3	54.8 ±0.8	67.6 ±0.4
	Relative Attention	**61.6** ±0.5	**65.0** ±0.3	**2.7** ±0.3	91.8 ±1.3	51.5 ±1.2	55.6 ±0.7	66.2 ±1.0
	+ Separators ⋆	59.9 ±1.0	63.4 ±0.4	2.4 ±0.3	**93.0** ±0.4	45.1 ±2.7	**58.5** ±1.8	67.5 ±0.3
	Sep-Full-Attention ⋆	60.4 ±1.2	62.2 ±1.4	2.2 ±0.3	92.4 ±1.0	48.2 ±0.6	57.2 ±0.8	**67.7** ±0.3
	Sep-to-Sep-and-Last ⋆	58.4 ±0.6	63.0 ±1.1	2.2 ±0.2	92.5 ±0.5	50.2 ±2.2	56.5 ±0.7	66.6 ±0.9
	Sep-to-Last ⋆	59.8 ±0.7	60.7 ±2.0	2.6 ±0.2	92.4 ±0.5	51.3 ±0.9	57.9 ±0.9	67.5 ±0.5

Table 2: **RobustFill** zero-shot generalization and few-shot fine-tuning results.

to 29.2% for a baseline Transformer with relative attention, or 10.5% without any of the improvements. However, for some generalization tasks, the performance difference between models is much smaller.

Interestingly, adding separator tokens to the baseline Transformer (with and without relative attention) slightly decreases performance on zero-shot length generalization for both datasets. This may be because the number of program parts is more obvious to the model, so the model is less likely to predict more `SEP` tokens than it has seen during training. Thus, the model may have difficulty generalizing to out-of-distribution lengths. Despite this drawback, applying our attention masks (enabled by the separator tokens) leads to the best length generalization in most cases.

All models struggled with zero-shot *Switch-Concept-Order*. The ordering pattern is likely very obvious to the Transformer, much like how a language model would learn that sentences start with capital letters and end with punctuation. Again we observe that the easier a pattern is to see, the harder zero-shot generalization would be—the model is more likely to overfit to that particular pattern, making it unlikely to deviate from that pattern when needed for generalization.

For several tasks, in particular *Switch-Concept-Order* in both domains, few-shot fine-tuning is very effective. Even though the model only sees 20 examples from the test distribution, the best model improves from 6.6% to 53.6% for RobustFill, or 17.5% to 66.8% for SCAN. With fine-tuning, our

Decompositional Transformer models exceed 90% generalization on RobustFill's *Compose-Different-Concepts* and SCAN's *Compose-New-Operation* and *Add-Operation-Functionality*.

Finally, we note that none of the three variations of attention masks were consistently better than any others. Thus, it is possible that the best attention mask type could be application-specific. For instance, Sep-to-Last is the best for zero-shot *Compose-Different-Concepts* in both domains, which can be explained by the observation that a model would perform well on this generalization task if it can effectively "forget" what concept was used in previous program parts, and Sep-to-Last is the most sparse attention mask with the least attention to previous parts.

6 Related Work

Program Synthesis. For surveys on program synthesis and machine learning for software engineering, see Gottschlich et al. (2018); Solar-Lezama (2018); Gulwani et al. (2017b); Allamanis et al. (2018). Much attention has focused on machine learning for programming by example (Devlin et al., 2017; Bunel et al., 2018; Parisotto et al., 2017; Ellis et al., 2020). Many methods incorporate learning to guide the search over programs, such as using learned premise selection (Balog et al., 2017; Odena & Sutton, 2020), syntax-guided search (Yin & Neubig, 2017; Lee et al., 2018), bottom-up search (Barke et al., 2020; Shi et al., 2020), two-level search (Nye et al., 2019), per-example search (Shrivastava et al.), and execution-guided synthesis methods (Zohar & Wolf, 2018; Ellis et al., 2019; Chen et al., 2019; Odena et al., 2020; Shi et al., 2022). Another class of program synthesis methods are symbolic search methods (Solar-Lezama, 2018; Gulwani et al., 2017b), such as bottom-up search and satisfiability solvers. Since purely symbolic search will not have problems with compositional generalization (barring failures or timeouts in search), it is an intriguing question for future work whether learning-based search methods face challenges with compositional generalization. More generally, there is less work on systematic generalization for machine learning for code, although Bieber et al. (2020) studies length generalization in the context of the learning-to-execute task (Zaremba & Sutskever, 2014).

Compositional Generalization Benchmarks. Many datasets have been proposed by the NLP community to evaluate understanding of natural language sentences with compositionally novel structures, such as SCAN (Lake & Baroni, 2018). These benchmarks are either constructed by synthesizing examples based on a fine-grained schema of generalization patterns like this work (Bahdanau et al., 2019; Keysers et al., 2020; Kim & Linzen, 2020), or by repartitioning existing datasets with *i.i.d.* samples into splits with disjoint compositional structures (Finegan-Dollak et al., 2018; Shaw et al., 2021). Our dataset is related to the COGS benchmark (Kim & Linzen, 2020), which defines a taxonomy of compositional structures in English syntax for natural language understanding. While many generalization concepts are similar to those proposed in Section 2 (e.g., extend operation functionality), we focus on measuring and modeling compositional generalization of computer programs under task specifications in both natural language and I/O examples.

Improving Compositional Generalization. A large body of work develops specialized neural architectures with improved generalization performance (Russin et al., 2019; Li et al., 2019; Liu et al., 2020; Chen et al., 2020; Herzig & Berant, 2020), but is typically limited to specific domains and tasks. More generalized approaches have been proposed, such as meta-learning (Lake, 2019; Wang et al., 2020; Conklin et al., 2021), data augmentation (Andreas, 2020; Oren et al., 2021; Akyürek et al., 2021; Wang et al., 2021; Qiu et al., 2021), and improving the representation of programs (Furrer et al., 2020; Herzig et al., 2021). Related to our work, recent studies attempt to regularize the attention distribution over source tokens to improve generalization of language understanding models (Oren et al., 2020; Yin et al., 2021), which encourage the model to attend to the aligned concept in the input (e.g., "right") when predicting a function (e.g., `Right`). Instead of relying on such alignment information between the source and targets to regularize cross-attention distributions, we mask the self-attention scores in the decoder to capture the compositionality of programs, which is applicable to domains like RobustFill where the source-target alignments are not clearly defined.

7 Conclusion

We argue that compositional generalization is particularly important for neural program synthesis, for two reasons. On the practical side, we would like synthesizers to be able to length generalize,

generalize to novel combinations of concepts, and so on. On the conceptual side, measuring compositional generalization might give us insight into what problem-solving strategies are learned by neural program synthesizers. To that end, we propose a suite of generalization tasks, which measure different types of compositional generalization that are desirable for program synthesis. These tasks can be applied to different synthesis domains to produce a set of benchmarks; we have introduced benchmarks for string manipulation programs and a simple navigation domain. We show that these benchmarks are particularly difficult for current sequence to sequence models, and we present some early results on modifications to the Transformer attention mechanism to encourage better generalization. Future work could explore whether large pre-trained Transformers also have difficulty with these benchmarks, as well as further methods for improving compositional generalization.

REFERENCES

Ekin Akyürek, Afra Feyza Akyurek, and Jacob Andreas. Learning to recombine and resample data for compositional generalization. *ArXiv*, abs/2010.03706, 2021.

Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)*, 51(4):81, 2018.

Jacob Andreas. Good-enough compositional data augmentation. In *Proceedings of ACL*, 2020.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. Program synthesis with large language models. August 2021.

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. In *International Conference on Learning Representations (ICLR)*, 2016.

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

Matej Balog, Alexander L Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. DeepCoder: Learning to write programs. In *International Conference on Learning Representations (ICLR)*, 2017.

Shraddha Barke, Hila Peleg, and Nadia Polikarpova. Just-in-Time learning for Bottom-Up enumerative synthesis. In *Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2020.

David Bieber, Charles Sutton, Hugo Larochelle, and Daniel Tarlow. Learning to execute programs with instruction pointer attention graph neural networks. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2020.

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

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Xinyun Chen, Chang Liu, and Dawn Song. Execution-guided neural program synthesis. In *International Conference on Learning Representations (ICLR)*, 2019.

Xinyun Chen, Chen Liang, Adams Wei Yu, D. Song, and Denny Zhou. Compositional generalization via neural-symbolic stack machines. In *Proceedings of NeurIPS*, 2020.

N. Chomsky and D.W. Lightfoot. *Syntactic Structures*. De Gruyter Reference Global. Mouton de Gruyter, 2002. ISBN 9783110172799. URL https://books.google.com/books?id=a6a_b-CXYAkC.

Henry Conklin, Bailin Wang, Kenny Smith, and Ivan Titov. Meta-learning to compositionally generalize. *ArXiv*, abs/2106.04252, 2021.

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

Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. RobustFill: Neural program learning under noisy I/O. *ICML*, 2017.

Kevin Ellis, Maxwell I. Nye, Yewen Pu, Felix Sosa, Josh Tenenbaum, and Armando Solar-Lezama. Write, execute, assess: Program synthesis with a REPL. In *Neural Information Processing Systems (NeurIPS)*, 2019.

Kevin Ellis, Catherine Wong, Maxwell Nye, Mathias Sable-Meyer, Luc Cary, Lucas Morales, Luke Hewitt, Armando Solar-Lezama, and Joshua B. Tenenbaum. Dreamcoder: Growing generalizable, interpretable knowledge with wake-sleep bayesian program learning. *CoRR*, abs/2006.08381, 2020. URL https://arxiv.org/abs/2006.08381.

Catherine Finegan-Dollak, Jonathan K. Kummerfeld, Li Zhang, Karthik Ramanathan, Sesh Sadasivam, Rui Zhang, and Dragomir Radev. Improving text-to-SQL evaluation methodology. In *Proceedings of ACL*, 2018.

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

Justin Gottschlich, Armando Solar-Lezama, Nesime Tatbul, Michael Carbin, Martin Rinard, Regina Barzilay, Saman Amarasinghe, Joshua B Tenenbaum, and Tim Mattson. The three pillars of machine programming. In *ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, pp. 69–80. ACM, 2018.

Yu Gu, Sue Kase, Michelle Vanni, Brian Sadler, Percy Liang, Xifeng Yan, and Yu Su. Beyond iid: three levels of generalization for question answering on knowledge bases. In *Proceedings of the Web Conference 2021*, pp. 3477–3488, 2021.

S. Gulwani, O. Polozov, and R. Singh. *Program Synthesis*. Foundations and Trends(r) in Programming Languages Series. Now Publishers, 2017a. ISBN 9781680832921. URL https://books.google.com/books?id=mK5ctAEACAAJ.

Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In *PoPL'11, January 26-28, 2011, Austin, Texas, USA*, 2011.

Sumit Gulwani, Oleksandr Polozov, Rishabh Singh, et al. Program synthesis. *Foundations and Trends® in Programming Languages*, 4(1-2):1–119, 2017b.

Jonathan Herzig and Jonathan Berant. Span-based semantic parsing for compositional generalization. In *Proceedings of EMNLP*, 2020.

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

Daniel Keysers, Nathanael Schärli, Nathan Scales, H. Buisman, Daniel Furrer, Sergii Kashubin, Nikola Momchev, Danila Sinopalnikov, Lukasz Stafiniak, Tibor Tihon, D. Tsarkov, Xiao Wang, Marc van Zee, and O. Bousquet. Measuring compositional generalization: A comprehensive method on realistic data. In *Proceedings of ICLR*, 2020.

Najoung Kim and Tal Linzen. Cogs: A compositional generalization challenge based on semantic interpretation. *ArXiv*, abs/2010.05465, 2020.

Brenden M Lake. Compositional generalization through meta sequence-to-sequence learning. In *Proceedings of NeurIPS*, 2019.

Brenden M. Lake and Marco Baroni. Generalization without systematicity: On the compositional skills of sequence-to-sequence recurrent networks. *ICML*, 2018.

Woosuk Lee, Kihong Heo, Rajeev Alur, and Mayur Naik. Accelerating search-based program synthesis using learned probabilistic models. In *Conference on Programming Language Design and Implementation (PLDI)*, pp. 436–449, June 2018.

Yuanpeng Li, Liang Zhao, JianYu Wang, and Joel Hestness. Compositional generalization for primitive substitutions. In *Proceedings of EMNLP/IJCNLP*, 2019.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien De Masson D'autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando De Freitas, Koray Kavukcuoglu, and Oriol Vinyals. Competition-Level code generation with AlphaCode. https://storage.googleapis.com/deepmind-media/AlphaCode/competition_level_code_generation_with_alphacode.pdf, 2022. Accessed: 2022-2-26.

Qian Liu, Shengnan An, Jianguang Lou, B. Chen, Zeqi Lin, Yan Gao, Bin Zhou, Nanning Zheng, and Dongmei Zhang. Compositional generalization by learning analytical expressions. In *Proceedings of NeurIPS*, 2020.

Gary F. Marcus. The algebraic mind: Integrating connectionism and cognitive science. 2001.

Maxwell I. Nye, Luke B. Hewitt, Joshua B. Tenenbaum, and Armando Solar-Lezama. Learning to infer program sketches. In *International Conference on Machine Learning (ICML)*, pp. 4861–4870, 2019. URL http://proceedings.mlr.press/v97/nye19a.html.

Augustus Odena and Charles Sutton. Learning to represent programs with property signatures. In *International Conference on Learning Representations (ICLR)*, 2020.

Augustus Odena, Kensen Shi, David Bieber, Rishabh Singh, Charles Sutton, and Hanjun Dai. BUSTLE: Bottom-Up program synthesis through learning-guided exploration. In *International Conference on Learning Representations (ICLR)*, September 2020.

Inbar Oren, Jonathan Herzig, Nitish Gupta, Matt Gardner, and Jonathan Berant. Improving compositional generalization in semantic parsing. In *Proceedings of EMNLP-Findings*, 2020.

Inbar Oren, Jonathan Herzig, and Jonathan Berant. Finding needles in a haystack: Sampling structurally-diverse training sets from synthetic data for compositional generalization. In *Proceedings of EMNLP*, 2021.

Emilio Parisotto, Abdel-rahman Mohamed, Rishabh Singh, Lihong Li, Dengyong Zhou, and Pushmeet Kohli. Neuro-symbolic program synthesis. In *International Conference on Learning Representations (ICLR)*, 2017.

Nick Parlante. CS106A: Programming methodologies. Course at Stanford University, 2022.

Linlu Qiu, Peter Shaw, Panupong Pasupat, Paweł Krzysztof Nowak, Tal Linzen, Fei Sha, and Kristina Toutanova. Improving compositional generalization with latent structure and data augmentation. *arXiv preprint arXiv:2112.07610*, 2021.

Jake Russin, Jason Jo, R. O'Reilly, and Yoshua Bengio. Compositional generalization in a deep seq2seq model by separating syntax and semantics. *ArXiv*, abs/1904.09708, 2019.

Peter Shaw, Jakob Uszkoreit, and Ashish Vaswani. Self-attention with relative position representations. In *North American Chapter of the Association for Computational Linguistics (NAACL)*, 2018.

Peter Shaw, Ming-Wei Chang, Panupong Pasupat, and Kristina Toutanova. Compositional generalization and natural language variation: Can a semantic parsing approach handle both? In *Proceedings of ACL*, 2021.

Kensen Shi, Jacob Steinhardt, and Percy Liang. FrAngel: Component-based synthesis with control structures. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–29, 2019.

Kensen Shi, David Bieber, and Rishabh Singh. TF-Coder: Program synthesis for tensor manipulations. *arXiv preprint arXiv:2003.09040*, 2020.

Kensen Shi, Hanjun Dai, Kevin Ellis, and Charles Sutton. CrossBeam: Learning to search in bottom-up program synthesis. In *International Conference on Learning Representations (ICLR)*, 2022.

Disha Shrivastava, Hugo Larochelle, and Daniel Tarlow. Learning to combine per-example solutions for neural program synthesis. In *Advances in Neural Information Processing Systems*.

Armando Solar-Lezama. Introduction to program synthesis. https://people.csail.mit.edu/asolar/SynthesisCourse/TOC.htma, 2018. Accessed: 2018-09-17.

Armando Solar-Lezama, Liviu Tancau, Rastislav Bodík, Sanjit A. Seshia, and Vijay A. Saraswat. Combinatorial sketching for finite programs. In *Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2006, San Jose, CA, USA, October 21-25, 2006*, pp. 404–415. ACM, 2006.

Alon Talmor and Jonathan Berant. The web as a knowledge-base for answering complex questions. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, pp. 641–651, 2018.

Emina Torlak and Rastislav Bodík. Growing solver-aided languages with rosette. In Antony L. Hosking, Patrick Th. Eugster, and Robert Hirschfeld (eds.), *ACM Symposium on New Ideas in Programming and Reflections on Software, Onward! 2013, part of SPLASH '13, Indianapolis, IN, USA, October 26-31, 2013*, pp. 135–152. ACM, 2013. doi: 10.1145/2509578.2509586. URL https://doi.org/10.1145/2509578.2509586.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Neural Information Processing Systems (NeurIPS)*, 2017.

Bailin Wang, Mirella Lapata, and Ivan Titov. Meta-learning for domain generalization in semantic parsing. *arXiv:2010.11988*, 2020.

Bailin Wang, Wenpeng Yin, Xi Victoria Lin, and Caiming Xiong. Learning to synthesize data for semantic parsing. *ArXiv*, abs/2104.05827, 2021.

Pengcheng Yin and Graham Neubig. A syntactic neural model for General-Purpose code generation. In *Assocation for Computational Linguistics (ACL)*, 2017.

Pengcheng Yin, Hao Fang, Graham Neubig, Adam Pauls, Emmanouil Antonios Platanios, Yu Su, Sam Thomson, and Jacob Andreas. Compositional generalization for neural semantic parsing via span-level supervised attention. In *2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, 2021.

Wojciech Zaremba and Ilya Sutskever. Learning to execute, 2014.

Amit Zohar and Lior Wolf. Automatic program synthesis of long programs with a learned garbage collector. In *Neural Information Processing Systems (NeurIPS)*, 2018.

Appendices

A SCAN AND ROBUSTFILL DSLS

Figure 2 contains the DSL for SCAN translation tasks, and Figure 3 contains the DSL for our RobustFill programs.

Command C	:=	C_1 "after" C_2 | D
D	:=	D_1 "and" D_2 | P
Part P	:=	Q | Q "twice" | Q "thrice"
Q	:=	l | r | a
Left-concept l	:=	v "left" | v "opposite left" | v "around left"
Right-concept r	:=	v "right" | v "opposite right" | v "around right"
Verb v	:=	"turn" | a
Action a	:=	"walk" | "look" | "run" | "jump"

C_1 "after" C_2	→	C_2 C_1
D_1 "and" D_2	→	D_1 D_2
Q "twice"	→	Q Q
Q "thrice"	→	Q Q Q
"turn left"	→	LTURN
"turn right"	→	RTURN
a "left"	→	LTURN a
a "right"	→	RTURN a
"turn opposite left"	→	LTURN LTURN
"turn opposite right"	→	RTURN RTURN
a "opposite left"	→	LTURN LTURN a
a "opposite right"	→	RTURN RTURN a
"turn around left"	→	LTURN LTURN LTURN LTURN
"turn around right"	→	RTURN RTURN RTURN RTURN
a "around left"	→	LTURN a LTURN a LTURN a LTURN a
a "around right"	→	RTURN a RTURN a RTURN a RTURN a
"walk"	→	WALK
"look"	→	LOOK
"run"	→	RUN
"jump"	→	JUMP

Figure 2: The DSL for SCAN commands (top) with translations (bottom) from language-like commands (e.g., "jump left") to action sequences (e.g., LTURN JUMP). This is generalized from Lake & Baroni (2018) to allow arbitrarily-many "and" and "after" conjunctions.

B SCAN AND ROBUSTFILL GENERALIZATION TASK DETAILS

RobustFill Details. Unless stated otherwise, all programs described in the following paragraph have length between 2 and 6 (number of program parts). For *Compose-Different-Concepts*, we group together all of the substring operations into *substring concepts* and all of the modification operations plus constant strings as *non-substring concepts* (the Compose operation is omitted). We use the same concepts for *Switch-Concept-Order*, where training examples use only the substring concept for the first half of the parts and only the non-substring concept for the latter half, and test examples

$$
\begin{array}{rcl}
\text{Program } P & := & \texttt{Concat}(e_1, e_2, \ldots) \\
\text{Expression } e & := & s \mid m \mid o \mid \texttt{ConstStr}(c) \\
\text{Compose } o & := & m_1(m_2) \mid m(s) \\
\text{Substring } s & := & \texttt{SubStr}(k_1, k_2) \mid \texttt{GetSpan}(r_1, i_1, b_1, r_2, i_2, b_2) \mid \texttt{GetToken}(t, i) \\
 & & \mid \texttt{GetUpto}(r) \mid \texttt{GetFrom}(r) \\
\text{Modification } m & := & \texttt{ToCase}(a) \mid \texttt{Replace}(\delta_1, \delta_2) \mid \texttt{Trim}() \mid \texttt{GetFirst}(t, i) \mid \texttt{GetAll}(t) \\
 & & \mid \texttt{Substitute}(t, i, c) \mid \texttt{SubstituteAll}(t, c) \mid \texttt{Remove}(t, i) \mid \texttt{RemoveAll}(t) \\
\text{Regex } r & := & t_1 \mid \ldots \mid t_n \mid \delta_1 \mid \ldots \mid \delta_m \\
\text{Type } t & := & \texttt{NUMBER} \mid \texttt{WORD} \mid \texttt{ALPHANUM} \mid \texttt{ALL_CAPS} \mid \texttt{PROP_CASE} \mid \texttt{LOWER} \mid \texttt{DIGIT} \mid \texttt{CHAR} \\
\text{Case } a & := & \texttt{PROPER} \mid \texttt{ALL_CAPS} \mid \texttt{LOWER} \\
\text{Position } k & := & -100 \mid -99 \mid \ldots \mid 1 \mid 2 \mid \ldots \mid 100 \\
\text{Index } i & := & -5 \mid -4 \mid \ldots \mid -1 \mid 1 \mid 2 \mid \ldots \mid 5 \\
\text{Boundary } b & := & \texttt{START} \mid \texttt{END} \\
\text{Delimiter } \delta & := & \&\,,.?@()[]\%\{\}/\,:;\$\#"' \\
\text{Character } c & := & A - Z \mid a - z \mid 0 - 9 \mid \&,.?@\ldots
\end{array}
$$

Figure 3: The DSL for string transformation tasks in the RobustFill domain, slightly modified from (Devlin et al., 2017) to add more functionality.

have the ordering reversed. For *Compose-New-Operation*, 25% of training examples are length 1 programs containing only a Compose operation, the remainder of the training examples are length 2-6 programs without the Compose operation, and the test examples are length 2-6 programs that use the Compose operation. For *Add-Operation-Functionality*, all examples are length 1-6 programs, training examples are those where a substring operation is not used within a Compose operation, and test examples are those where a substring operation is used within a Compose operation.

SCAN Details. We create compositional generalization tasks for the SCAN domain in largely the same way as for the RobustFill domain, with the following differences. For *Compose-Different-Concepts* and *Switch-Concept-Order*, we use all left-direction phrases as one concept, all right-direction phrases as the other concept, and omit phrases without a direction. For *Compose-New-Operation*, 10% of training examples are (exactly) the length 1 "jump" command, the remaining training examples are length 1-6 commands that do not contain "jump", and the training examples are length 1-6 commands that do contain "jump" (but are not exactly "jump" itself). For *Add-Operation-Functionality*, training commands do not contain "around right", while test commands do contain "around right". The setup of the latter two generalization tasks closely mirrors tasks from Lake & Baroni (2018).

C Model and Training Hyperparameters

For our models, we used default hyperparameters already existing in the frameworks used in our implementation. In particular, the Transformer has 4 attention heads, 3 layers, a hidden dimension of 512, and an embedding dimension of 256. Relative attention uses 32 different buckets for relative positions, with a maximum distance of 128. We use a dropout rate of 0.1. During training, we use a learning rate schedule consisting of a base learning rate of 1×10^{-3}, linear warmup of 16000 steps, and square root decay. During fine-tuning, we use a constant learning rate of 1×10^{-4}.

D Plots for SCAN and RobustFill Results

Table 1 and Table 2 in Section 5 provide zero-shot and few-shot generalization results for SCAN and RobustFill. Figure 4, Figure 6, Figure 5, and Figure 7 provide bar graphs for a more visual representation of the same data.

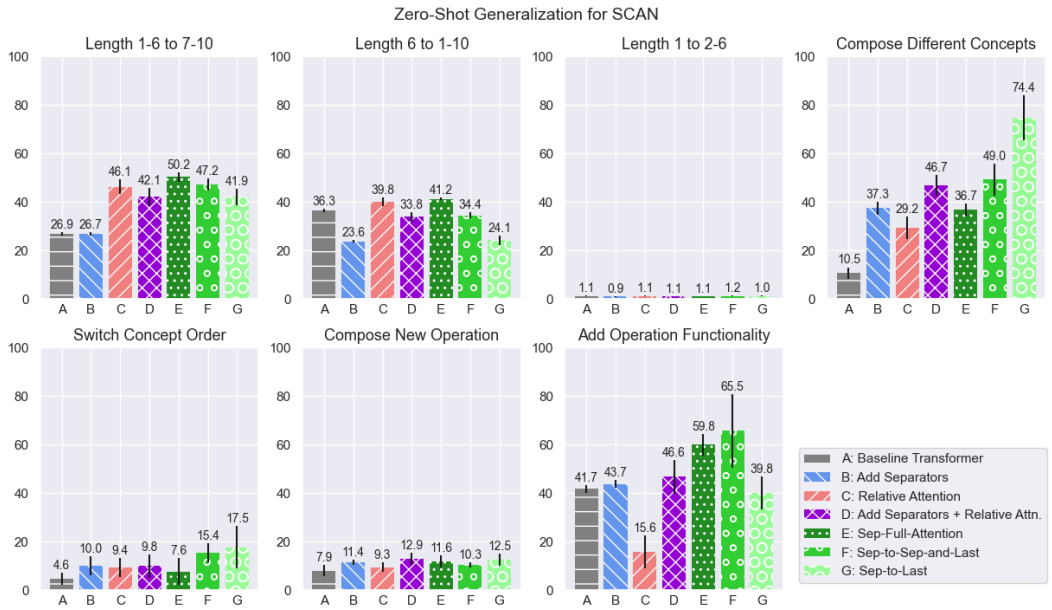


Figure 4: Zero-shot generalization for SCAN, trained for 10,000 steps with a batch size of 128. The bar heights represent the mean accuracy on the test set over 3 different random initializations, and the error bars represent one standard deviation above and below the mean.

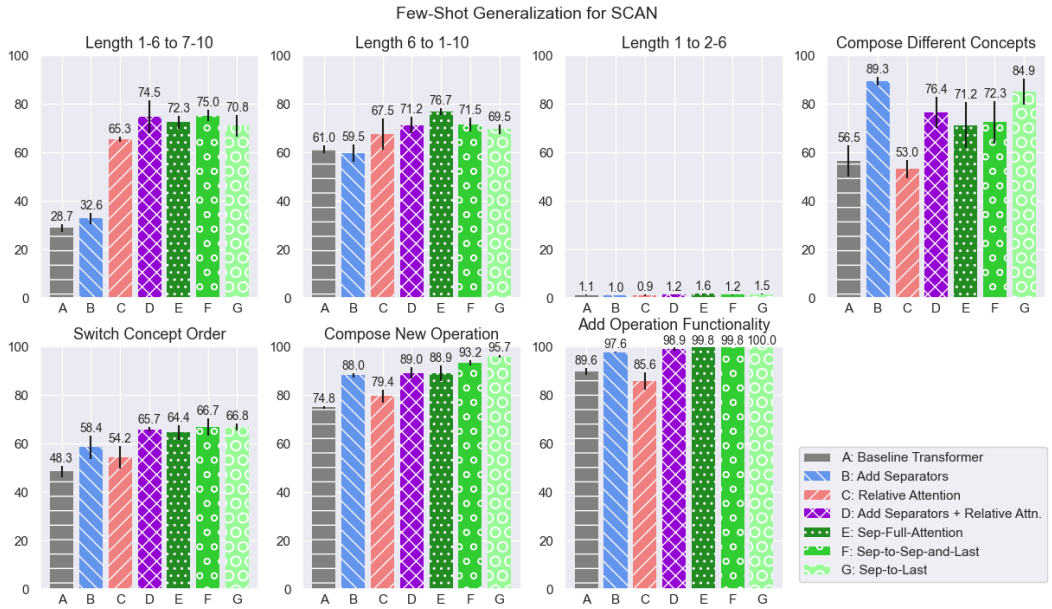


Figure 5: Few-shot generalization for SCAN, fine-tuning with 20 examples from the train distribution and 20 examples from the test distribution, for 30 epochs. As in the zero-shot case, the bars show the mean accuracy on the test set over the 3 random initializations with error bars representing one standard deviation above and below the mean.

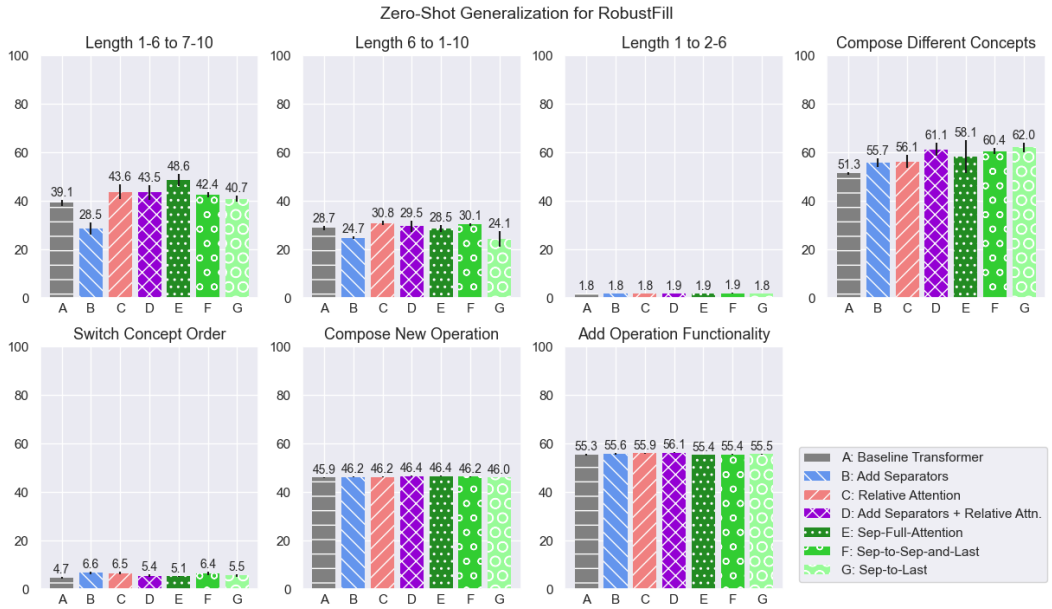


Figure 6: Zero-shot generalization for RobustFill, trained for 1 million steps with a batch size of 128.

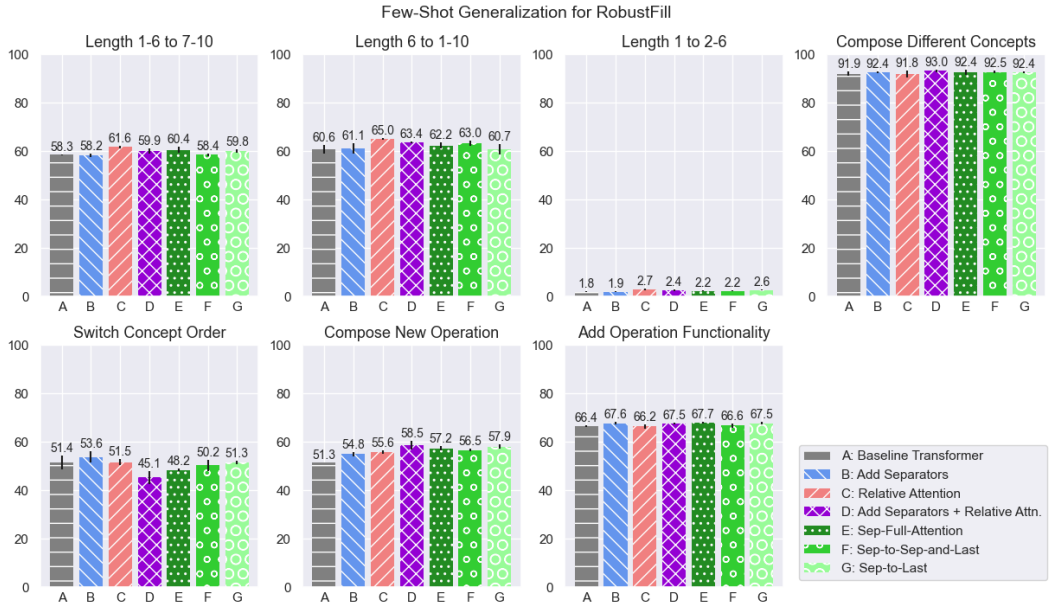


Figure 7: Few-shot generalization for RobustFill. The same models from the zero-shot experiment were fine-tuned on a set of 20 examples from the train distribution and 20 examples from the test distribution, for 30 epochs.