Learning Compositional Rules via Neural Program Synthesis

Maxwell I. Nye [1] **Armando Solar-Lezama** [1] **Joshua B. Tenenbaum** [1] **Brenden M. Lake** [2 3]

Abstract

Many aspects of human reasoning, including language, require learning rules from very little data. Humans can do this, often learning systematic rules from very few examples, and combining these rules to form compositional rule-based systems. Current neural architectures, on the other hand, often fail to generalize in a compositional manner, especially when evaluated in ways that vary systematically from training. In this work, we present a neuro-symbolic model which learns entire rule systems from a small set of examples. Instead of directly predicting outputs from inputs, we train our model to induce the explicit system of rules governing a set of previously seen examples, drawing upon techniques from the neural program synthesis literature. Our rule-synthesis approach outperforms neural meta-learning techniques in three domains: an artificial instruction-learning domain used to evaluate human learning, the SCAN challenge datasets, and learning rule-based translations of number words into integers for a wide range of human languages.

1. Introduction

Humans have a remarkable ability to learn compositional rules from very little data. For example, a person can learn a novel verb "to dax" from a few examples, and immediately understand what it means to "dax twice" or "dax around the room quietly." When learning their native language, children must learn many interrelated concepts simultaneously, including the meaning of both verbs and modifiers ("twice", "quietly", etc.), and how they combine to form complex meanings. Moreover, these compositional skills do not depend on detailed knowledge of a particular language (Lake et al., 2019); people can also learn novel artificial languages and generalize systematically to new compositional meanings (see Figure 2).

Fodor and Marcus have argued that systematic compositionality, while critical to human language and thought, is incompatible with classic neural networks (i.e., eliminative connectionism) (Fodor & Pylyshyn, 1988; Marcus, 1998; 2003). Recent work shows that these issues still plague contemporary neural architectures, which struggle to generalize in systematic ways when directly learning rule-like mappings between input sequences and output sequences (Lake & Baroni, 2018; Loula et al., 2018). Basic forms of compositionality can be acquired by training memory-augmented neural models with meta-learning (Lake, 2019), but this approach has yet to address the hardest challenges of learning genuinely new, systematic rules from examples. Given these classic and recent findings, Marcus continues to postulate that hybrid neural-symbolic architectures (implementational connectionism) are needed to achieve genuine compositional, human-like generalization (Marcus, 2003; 2018; Marcus & Davis, 2019).

An important goal of AI is to build systems which possess this sort of systematic rule-learning ability, while retaining the speed and flexibility of neural inference. In this work, we present a neural-symbolic framework for learning entire rule systems from examples. As illustrated in Figure 1B, our key idea is to frame the problem as explicit rule-learning through fast neural proposals and rigorous symbolic checking. Instead of training a model to predict the correct output given a novel input (Figure 1A), we train our model to induce the explicit system of rules governing the behavior of all previously seen examples (Figure 1B; Grammar proposals). Once inferred, this rule system can be used to predict the behavior of any new example (Figure 1B; Symbolic application).

This explicit rule-based approach confers several advantages compared to a pure input-output based approach. Instead of learning a blackbox input-output mapping, and applying it to each new query item for which we would like to predict an output (Figure 1A), we instead search for an explicit *program* which we can check against previous examples (the support set). This allows us to propose and check candidate programs, only terminating search when the proposed solution is consistent with prior data.

This framing also allows immediate and automatic gener-

[1]MIT [2]NYU [3]Facebook AI Research. Correspondence to: Maxwell Nye <mnye@mit.edu>.
A PyTorch implementation of this work can be found at `https://github.com/mtensor/rulesynthesis`.

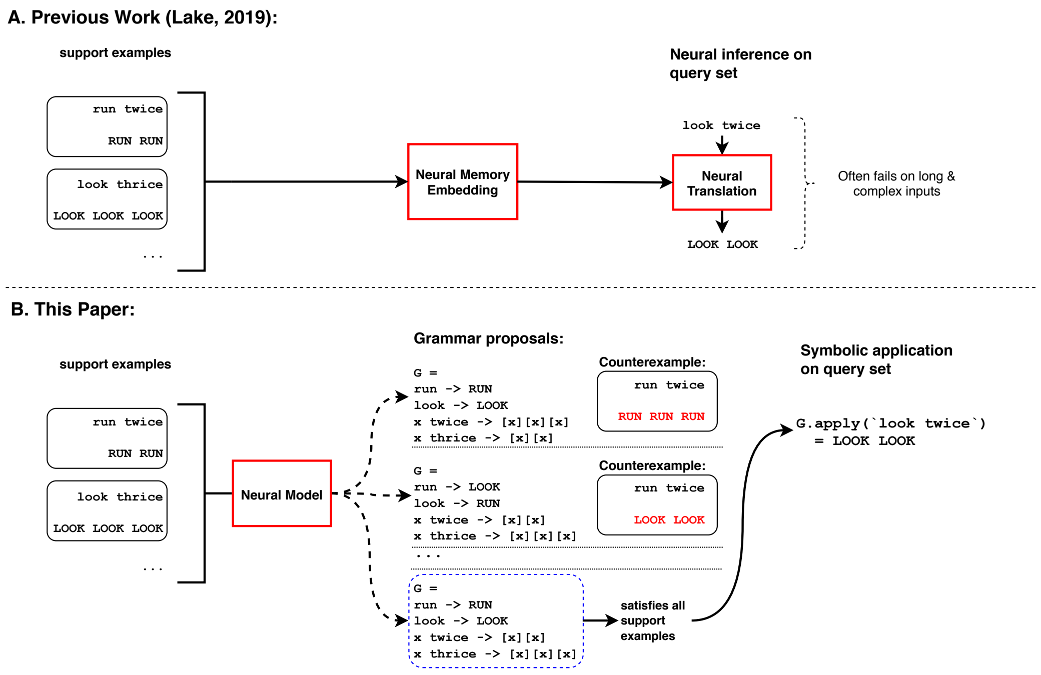


Figure 1. Illustration of our synthesis-based rule learner and comparison to previous work. A) Previous work (Lake, 2019): Support examples are encoded into an external neural memory. A query output is predicted by conditioning on the query input sequence and interacting with the external memory via attention. B) Our model: Given a support set of input-output examples, our model produces a distribution over candidate grammars. We sample from this distribution, and symbolically check consistency of each grammar against the support set until a grammar is found which satisfies the input-output examples in the support set. This approach allows much more effective search than selecting the maximum likelihood grammar from the network.

alization: once the correct rule system is learned, it can be correctly applied in novel scenarios which are a) arbitrarily complex and b) outside the distribution of previously seen examples. To build our rule-learning system, we draw on work in the neural program synthesis literature (Ellis et al., 2019; Devlin et al., 2017), allowing us to solve complex rule-learning problems which are difficult for both neural and traditional symbolic methods.

Our training scheme is inspired by meta-learning. Assuming a distribution of rule systems, or a "meta-grammar," we train our model by sampling grammar-learning problems and training on these sampled problems. We can interpret this as a kind of approximate Bayesian grammar induction, where our goal is to maximize the likelihood of finding a latent program which explains all of the data (Le et al., 2016).

We demonstrate that, when trained on a general meta-grammar of rule-systems, our rule-synthesis method can outperform neural meta-learning techniques.

Concretely, our main contributions are:

- We present a neuro-symbolic program induction model which can learn novel rule systems from few examples. Our model employs a symbolic program representation for compositional generalization and neural program induction for fast and flexible inference. This allows us to leverage search in the space of programs, for a guess-and-check approach.
- We show that our model can learn to interpret artificial languages from few examples, and further demonstrate that our model can solve tasks in the SCAN compositional learning domain.
- Finally, we show that our model can outperform baselines in learning how to interpret number words in unseen languages from few examples.

2. Related Work

Meta-Learning: Lake (2019) uses meta-learning to induce a sequence-to-sequence model for predicting query input-output transformations, given a small number of support examples (Figure 1A). They show significant improvements over standard sequence-to-sequence methods, and demonstrate that their model captures human biases in approaching few-shot sequence-to-sequence tasks. Our work uses a similar training scheme, but we instead learn an explicit program which can be applied to held out query items (Figure 1B).

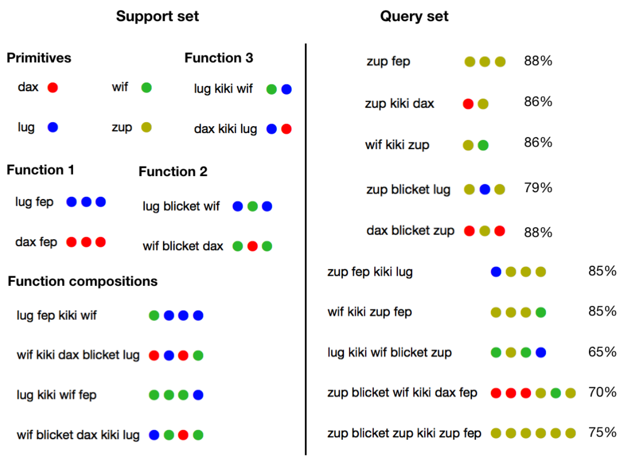


Figure 2. An example of few-shot learning of instructions. In Lake et al. (2019), participants learned to execute instructions in a novel language of nonce words by producing sequences of colored circles. Human performance is shown next to each query instruction, as the percent correct across participants. When conditioned on the support set, our model is able to predict the correct output sequences on the held out query instructions by synthesizing the grammar in Figure 3.

Program Synthesis: Our approach derives from the field of neural program synthesis, also called neural program induction.[1] We are inspired by work such as Devlin et al. (2017), and other program synthesis approaches, including enumerative approaches (Balog et al., 2016), execution guided work (Chen et al., 2018; Zohar & Wolf, 2018; Ellis et al., 2019; Yang & Deng, 2019), and hybrid models (Murali et al., 2017; Nye et al., 2019) However, a key difference in our work is the number of input-output examples provided to the system. Previous neural program synthesis systems condition on a handful of (less than 10) examples. We demonstrate that our method is able to synthesize very long programs while conditioned on up to 100 examples, and can attend to the relevant examples for decoding each program sub-component.

3. Our Approach

Overview: Given a small support set of input-output examples, $\mathcal{X} = \{(x_i, y_i)\}_{i=1..n}$, our goal is to produce the outputs corresponding to a query set of inputs $\{q_i\}_{i=1..m}$ (see Figure 2). To do this, we build a neural program induction model $p_\theta(\cdot|\mathcal{X})$ which accepts the given examples and synthesizes a symbolic program G, which we can execute on query inputs to predict the desired query outputs, $r_i = G(q_i)$. Our symbolic program consists of an "interpretation grammar," which is a sequence of *rewrite rules*, each of which represents a transformation of token sequences. The details of the interpretation grammar are discussed below. At test time, we employ our neural program induction model to drive a simple search process. This search process proposes candidate programs by sampling from the program induction model and symbolically checks whether candidate programs satisfy the support examples by executing them on the support inputs, i.e., checking that $G(x_i) = y_i$ for all $i = 1..n$. During each training episode, our model is given a support set $\mathcal{X}$ and is trained to infer an underlying program G which explains the support and held-out query examples.

[1]In this work we use the terms *program synthesis* and *program induction* interchangeably.

Model: A schematic of our architecture is shown in Figure 3. Our neural model $p_\theta(G|\mathcal{X})$ is a distribution over programs G given the support set $\mathcal{X}$. Our implementation is quite simple and consists of two components: an encoder $Enc(\cdot)$, which encodes each support example (x_i, y_i) into a vector h_i, and a decoder $Dec(\cdot)$, which decodes the program while attending to the support examples:

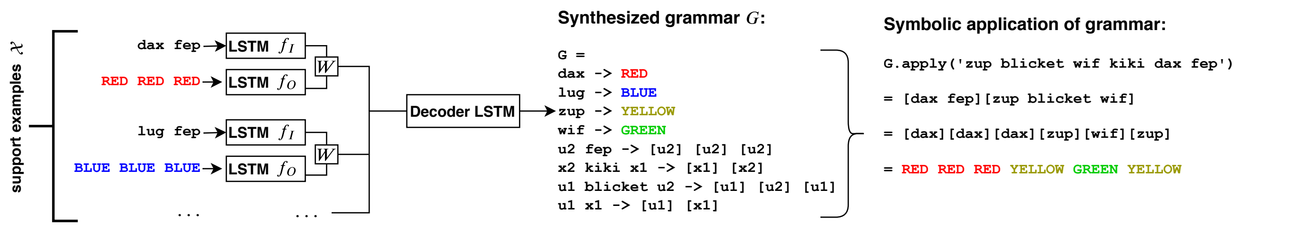
$$p_\theta(\cdot|\mathcal{X}) = Dec(\{h_i\}_{i=1..n}),$$
$$\text{where } \{h_i\}_{i=1..n} = Enc(\mathcal{X})$$

Encoder: For each support example (x_i, y_i), the input sequence x_i and output sequence y_i are each encoded into a vector by taking the final hidden state of an input BiLSTM encoder $f_I(x_i)$ and an output BiLSTM encoder $f_O(y_i)$, respectively (Figure 3; left). These hidden states are then combined via a single feedforward layer with weights W to produce one vector h_i per support example:

$$h_i = ReLU(W[f_I(x_i); f_O(y_i)])$$

Decoder: We use an LSTM for our decoder (Figure 3; center). The decoder hidden state u_0 is initialized with the sum of all of the support example vectors, $u_0 = \sum_i h_i$, and the decoder produces the program token-by-token while attending to the support vectors h_i via Luong attention (Luong et al., 2015). The decoder outputs a tokenized program, which is then parsed into an interpretation grammar object.

Interpretation Grammar: The programs in this work are instances of an *interpretation grammar*, which is a form of term rewriting system (Kratzer & Heim, 1998). The interpretation grammar used in this work consists of an ordered list of rules. Each rule consists of a left hand side (LHS) and a right hand side (RHS). The left hand side consists of the input words, string variables `x` (regexes that match entire strings), and primitive variables `u` (regexes that match single words). Evaluation proceeds as follows: An input sequence is checked against the rules in order of the rule priority. If the rule LHS matches the input sequence, then the sequence is replaced with the RHS. If the RHS contains bracketed variables (i.e., `[x]` or `[u]`), then the contents of these variables are evaluated recursively through the same process. In Figure 3 (right), we observe grammar application on the



Synthesized grammar G:

```
G =
dax -> RED
lug -> BLUE
zup -> YELLOW
wif -> GREEN
u2 fep -> [u2] [u2] [u2]
x2 kiki x1 -> [x1] [x2]
u1 blicket u2 -> [u1] [u2] [u1]
u1 x1 -> [u1] [x1]
```

Symbolic application of grammar:

```
G.apply('zup blicket wif kiki dax fep')
= [dax fep][zup blicket wif]
= [dax][dax][dax][zup][wif][zup]
= RED RED RED YELLOW GREEN YELLOW
```

Figure 3. Illustration of our synthesis-based rule learner neural architecture and grammar application. Support examples are encoded via BiLSTMs. The decoder LSTM attends over the resulting vectors and decodes a grammar, which can be symbolically applied to held out query inputs. Middle: an example of a fully synthesized grammar which solves the task in Fig. 2.

input sequence `zup blicket wif kiki dax fep`. The first matching rule is the `kiki` rule,[2] so its RHS is applied, producing `[dax fep] [zup blicket wif]`, and the two bracketed strings are recursively evaluated using the `fep` and `blicket` rules, respectively.

Search: At test time, we sample candidate programs from our neural program induction model. If the new candidate program G satisfies the support set —i.e., if $G(x_i) = y_i$ for all $i = 1..n$ —then search terminates and the candidate program G is returned as the solution. The program G is then applied to the held-out query set to produce final query predictions $r_i = G(q_i)$. During search, we maintain the best program so far, defined as the program which satisfies the largest number of support examples. If the search timeout is exceeded and no program has been found which solves all of the support examples, then the best program so far is returned as the solution.

This search procedure confers major advantages compared to previous approaches. In a pure neural induction model (Figure 1A), given a query input and corresponding output prediction, there is no way to check consistency with the support set. Conversely, casting the problem as a search for a satisfying program allows us to explicitly check each candidate program against the support set, to ensure that it correctly maps support inputs to support outputs. The benefit of such an approach is shown in Section 4.2, where we can achieve perfect accuracy on SCAN by increasing our search budget and searching until a program is found which satisfies all of the support examples.

Training: We train our model in a similar manner to Lake (2019). During each training episode, we randomly sample an interpretation grammar G from a distribution over interpretation grammars, or "meta-grammar" $\mathcal{M}$. We then randomly sample a set of input sequences consistent with the sampled interpretation grammar, and apply the interpretation grammar to each input sequence to produce the corresponding output sequence. This gives us a support set of input-output examples $\mathcal{X}_G$. We train the parameters θ of our network p_θ via supervised learning to output the interpretation grammar when conditioned on the support set of input-output examples, maximizing the following objective $\mathcal{L}$ by gradient descent:

$$\mathcal{L} = \mathop{\mathbb{E}}_{(G,\mathcal{X}_G)\sim\mathcal{M}} [\log p_\theta(G|\mathcal{X}_P)]$$

[2]Note that the `fep` rule is not applied first because `u2` is a primitive variable, so it only matches when `fep` is preceded by a single primitive word.

4. Experiments

All models were implemented in PyTorch. All testing and training was performed on one Nvidia GTX 1080 Ti GPU. For all models, we used LSTM embedding and hidden sizes of 200, and trained using the Adam optimizer (Kingma & Ba, 2014) with a learning rate of 1e-3. Training and testing runs used a batch size of 128. For all experiments, we report standard error in the supplement.

4.1. Experiment: MiniSCAN

Our first experimental domain is the paradigm introduced in Lake et al. (2019), informally dubbed "MiniSCAN." The goal of this domain is to learn compositional, language-like rules from a very limited number of examples. In Lake et al. (2019), human subjects were allowed to study the 14 example 'support instructions' in Figure 2, which demonstrate how to transform a sequence of nonce words into a sequence of colored circles. Participants were then tested on the 10 'query instructions' in Figure 2, to determine how well they had learned to execute instructions in this novel language. Our aim is to build a model which learns this artificial language from few examples, similar to humans.

Training details: To perform these rule-learning tasks, we trained our model on a series of meta-training episodes. During each training episode, a grammar was sampled from the meta-grammar distribution, and our model was trained to recover this grammar given a support set of example sequences. In our experiments, the meta-grammar randomly sampled grammars with 3-4 *primitive* rules and 2-4 *higher-order* rules. Primitive rules map a word to a color (e.g. `dax -> RED`), and higher order rules encode variable transformations given by a word (e.g. `x1 kiki x2 -> [x2]`

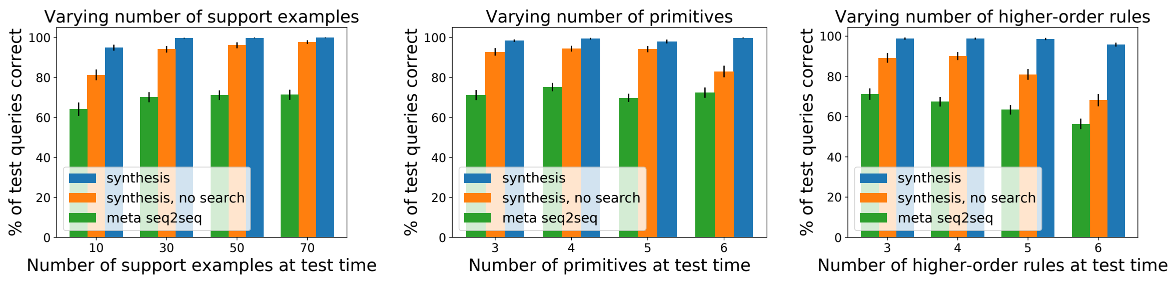


Figure 4. MiniSCAN generalization results. We train on random grammars with 3-4 primitives, 2-4 higher order rules, and 10-20 support examples. At test time, we vary the number of support examples (left), primitive rules (center), and higher-order rules (right). The synthesis-based approach using search achieves near-perfect accuracy for most test conditions.

`[x1]`). (In a higher-order rule, the LHS can be one or two variables and a word, and the RHS can be any sequence of bracketed forms of those variables.) For each grammar, we trained with a support set of 10-20 randomly sampled examples. We trained our models for 12 hours. Meta-grammar details can be found in Section A.1 of the supplement.

Alternate Models: We compare our model against two alternate models. The first alternate model is the **meta seq2seq** model introduced in Lake (2019). This model is also trained on episodes of randomly sampled grammars. However, instead of synthesizing a grammar, the meta seq2seq model conditions on support examples and attempts to translate query inputs directly to query outputs in a sequence to sequence manner (Figure 1A). This baseline allows us to compare with models which use a learned representation instead of a symbolic program representation.

The second alternate model is a lesioned version of our synthesis approach, dubbed the **no search** baseline. This model does not perform guess-and-check search, and instead returns the grammar which results from greedily decoding the most likely token at each step. This baseline allows us to determine how much of our model's performance is due to its ability to perform guess-and-check search.

Test Details: Our synthesis methods were tested by sampling from the network for the best grammar, or until a candidate grammar was found which was consistent with all of the support examples. We used a sampling timeout of 30 sec, and the model samples approx. 35 prog/second, resulting in a maximum search budget of approx. 1000 candidate programs. For each of our experiments, we tested on 50 held-out test grammars, each containing 10 query examples.

Results: To evaluate our rule-learning model and baselines, we test the models on a battery of evaluation schemes. In general, we observe that the synthesis methods are much more accurate than than the pure neural meta seq2seq method, and only the search-based synthesis method is able to consistently predict the correct query output sequence for all test conditions. Our main results are shown in Figure 4. We observed that, when the support set is too small, there

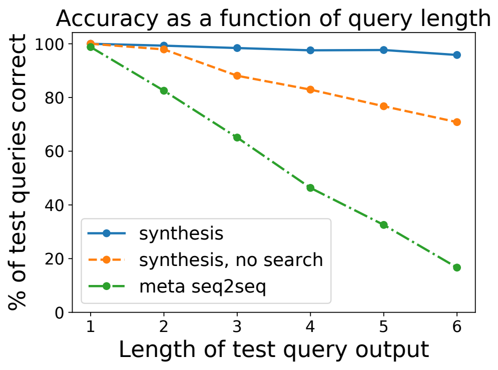


Figure 5. MiniSCAN length generalization results. A key challenge for compositional learning is generalization across lengths. We plot accuracy as a function of query output length for the "4 higher-order rules" test condition in Figure 4 above. The accuracy of our synthesis approach does not degrade as a function of query output length, whereas the performance of baselines decreases.

are often not enough examples to disambiguate between several grammars which all satisfy the support set, but may not satisfy the query set. Thus, in our first experiment, we varied the number of support examples during test time and evaluated the accuracy of each model. We observed that, when we increased the number of support elements to 50 or more, the probability of failing any of the query elements fell to less than 1% for our model. However, we also wanted to determine how well these models could generalize to grammars systematically different than those seen during training. For our second and third experiments, we varied the number of primitives in the test grammars (Figure 4 center), and the number of higher-order functions in the test grammars (Figure 4 right). For these experiments, each support set contained 30 examples.

Both synthesis models are able to correctly translate query items with high accuracy (89% or above) when tested on held-out grammars within the training distribution (3-4 primitive rules and 3-4 higher order rules). However, only the performance of the search-based synthesis model does not drop below 95% as the number of primitives and higher order rules increases beyond the training distribution, indicating that the ability to search for a consistent program plays a large role in out-of-sample generalization.

Furthermore, in instances where the synthesis based methods have perfect accuracy because they recover exactly the generating grammar (or some equivalent grammar), they would also be able to trivially generalize to query examples of any size or complexity, as long as these examples followed the same generating grammar. On the other hand, as reported in many previous studies (Graves et al., 2014; Lake & Baroni, 2018; Lake, 2019), approaches which attempt to neurally translate directly from inputs to outputs struggle to generate sequences much longer than those seen during training. This is a clear conceptual advantage of the synthesis approach; symbolic rules, if accurately inferred, necessarily allow correct translation in every circumstance. To investigate this property, we plot the performance of our models as a function of the query example length for the 4 higher-order rule test condition above (Figure 5). The performance of the baselines decays as the length of the query examples increases, whereas the search-based synthesis model experiences no such decrease in performance.

This indicates a key benefit of the program synthesis approach: When a correct program is found, it trivially generalizes correctly to arbitrary query inputs, regardless of how out-of-distribution they may be compared to the support inputs, as long as those query inputs follow the same rules as the support inputs. The model's ability to search the space of programs also plays a crucial role, as it allows the system to find a grammar which satisfies the support examples, even if it is not the most likely grammar under the neural network distribution.

We also note that our model is able to solve the task in Figure 2; averaged over 20 runs, our model achieves a score of 98.75% on the query set, which is higher than the average score for human participants in Lake et al. (2019). The no search and meta seq2seq model are not able to solve the task, achieving scores of 37.5% and 25%, respectively.

4.2. Experiment: SCAN Challenge

Our second experimental domain is the SCAN dataset, introduced in Lake & Baroni (2018) and Loula et al. (2018). The goal of SCAN is to test the compositional abilities of neural network systems, to determine how well they can generalize to held out test data which varies systematically from the training data. The SCAN dataset consists of simple English commands paired with corresponding discrete actions (see Figure 6). The dataset has approximately 21,000 command-to-action examples, which are arranged in several test-train splits to examine different aspects of compositionality. We focus on four splits: The **simple** split randomly sorts data into the train and test sets. The **length** split places all examples with output length of up to 22 tokens into the train split, and all other examples (24 to 48 tokens long) into the test split. This split tests whether a model can learn to generalize from short examples to longer ones. The **add jump** split teaches the model how to 'jump' in isolation, along with the compositional uses of other primitives, and then evaluates it on all compositional uses of jump, such as 'jump twice' or 'jump around to the right and walk twice.' The **add around right** split is similar to the 'add jump' split, except the phrase 'around right' is held out from the training set and the goal is to piece together the composite meaning from the meaning of its components. The 'add jump' and 'add around right' splits test if a model can learn to compositionally use words or phrases which had previously only been seen in isolation.

walk
```
WALK
```
walk left twice
```
LTURN WALK LTURN WALK
```
jump
```
JUMP
```
jump around left
```
LTURN JUMP LTURN JUMP LTURN JUMP LTURN JUMP
```
walk right
```
RTURN WALK
```

```
walk -> WALK
jump -> JUMP
run -> RUN
look -> LOOK
left -> LTURN
right -> RTURN
turn -> 'EMTPY_STRING'
u1 opposite u2 -> [u2] [u2] [u1]
u1 around u2 ->
  [u2] [u1] [u2] [u1] [u2] [u1] [u2] [u1]
x2 twice -> [x2] [x2]
x1 thrice -> [x1] [x1] [x1]
x2 after x1 -> [x1] [x2]
x1 and x2 -> [x1] [x2]
u1 u2 ->[u2] [u1]
```

Figure 6. Top: Example SCAN data. Each example consists of a synthetic natural language command (top) paired with a discrete action sequence (bottom). Fig. adapted from Andreas (2019). Bottom: Example of induced grammar which solves SCAN.

Training Setup: Previous work on SCAN has used a variety of techniques, including data augmentation (Andreas, 2019), meta-learning (Lake, 2019), and syntactic attention (Russin et al., 2019). Most related to our approach, Lake (2019) trained a model to solve related problems using a meta-training procedure. At test time, samples from the SCAN train split were used as support items, and samples from the SCAN test split were used as query items. However, in Lake (2019), the meta-training distribution consisted of different permutations of assigning the SCAN primitive actions ('run', 'jump', 'walk', 'look') to their commands ('RUN', 'JUMP', 'WALK', 'LOOK'), while maintaining the same SCAN task structure between meta-train and meta-test. Therefore, in these experiments, the goal of the learner is to assign primitive actions to commands within a known task structure, while the higher-order rules, such as 'twice', and 'after', remain constant between meta-train and meta-test.

In contrast, we approach learning the entire SCAN grammar from few examples, without knowledge about the form of the particular SCAN grammar itself beforehand. We meta-train on a general and broad meta-grammar for SCAN-like rule systems, similar to our approach above in Section 4.1. Specifically, we train on random grammars with between 4 and 9 primitives and 3 and 7 higher order rules, with random assignment of words to meanings. Examples of random grammars are given in the supplement. Models are trained on 30-50 support examples, and we train for 48 hours, viewing approximately 9 million grammars. Meta-grammar details can be found in Section A.2 of the supplement.

Testing Setup: We test our fully trained model on each split of SCAN as if it were a new few-shot test episode with support examples and a held out query set, as above. For each SCAN split, we use the training set as test-time support elements, and input sequences from the SCAN test set are used as query elements. The SCAN training sets have thousands of examples, so it is infeasible to attend over the entire training set at test time. Therefore, at test time, we randomly sample 100 examples from the SCAN training set to use as the support set for our network. We can then run program inference, conditioned on just these 100 examples from the SCAN training set.

Because of the large number of training examples, we are also able to slightly modify our test-time search algorithm to increase performance: We select 100 examples as the initial support set for our network, and search for a grammar which perfectly satisfies them. If no satisfying grammar is found within a set timeout of 20 seconds, we resample another 100 support examples and retry searching for a grammar. We repeat this process until a satisfying grammar is found. This methodology, inspired by RANSAC (Fischler & Bolles, 1981), allows us to utilize many examples in the training set without attending over thousands of examples at once.

Because the SCAN grammar lies within the support of the meta-grammar distribution, we additionally test two probabilistic inference baselines: MCMC and rejection sampling directly from the meta-grammar. We implement these baselines in the `pyprob` probabilistic programming language (Le et al., 2016). For both baselines, we allow a maximum timeout of 180 seconds. Both MCMC and sampling evaluate more candidate programs than our baseline, achieving about 60 programs/sec, compared to the synthesis model, which evaluates about 35 programs/sec.

Results: Table 1 shows the overall performance of our model compared to baselines. Using search, our synthesis model is able to achieve perfect performance on each SCAN split. Without search, the synthesis approach cannot solve SCAN, never achieving performance greater than 15%.

Table 1. Accuracy on SCAN splits.

	length	simple	jump	right
Synth (Ours)	**100**	**100**	**100**	**100**
Synth (no search)	0.0	13.3	3.5	0.0
Meta Seq2Seq	0.04	0.88	0.51	0.03
MCMC	0.02	0.0	0.01	0.01
Sample from prior	0.04	0.03	0.03	0.01
GECA (Andreas, 2019)	–	–	87	82
Meta Seq2Seq (perm)	16.64	–	99.95	98.71
Syntactic attention	15.2	–	78.4	28.9
Seq2Seq[3]	13.8	99.8	0.08	–

Table 2. Required search budget for our synthesis model on SCAN.

	length	simple	jump	right
Search time (sec)	39.1	33.7	74.6	36.1
Number of prog. seen	1516	1296	2993	1466
Number of ex. used	149.4	144.8	209.2	143.8
Fraction of ex. used	0.88%	0.86%	1.6%	0.94%

Likewise, meta seq2seq, using neither a program representation nor search, cannot solve SCAN when trained on a very general meta-grammar, solving less than 1% of the test set.

One advantage of our approach is that we don't need to retrain the model for each split. Once meta-training has occurred, the model can be tested on each of the splits and is able to induce a satisfying grammar for all four splits.

Compared to previous approaches, we also require many fewer examples to solve SCAN. Table 2 reports how many examples and how much time are required, on average, in order to find a grammar which solves all examples in the support set. Previous approaches use the entire training set, whereas we require less than 2% of the training set data. In the supplement, Table 6 reports the results of running our algorithm with a fixed time budget and without swapping out support sets when no perfectly satisfying grammar is found, averaged over 20 evaluation runs. Under this test condition, 180 seconds is sufficient to achieve perfect performance on the length and simple splits, and nearly perfect performance on the add around right split (98.4%). The add jump split is more difficult; we achieve 43.3% ($\pm 10\%$) accuracy.

4.3. Experiment: Learning Number Words

Our final experimental domain is the real-world problem of learning to infer the integer meaning of a number word sequence from few examples. This domain provides a real-world example of compositional rule learning. Figure 7 provides an example of this number learning task for Japanese.

Setup: In this domain, each grammar G is an ordered list of rules which defines a transformation from strings

[3]Seq2Seq results from Lake & Baroni (2018).

Table 3. Accuracy on few-shot number-word learning, using a maximum timeout of 45 seconds.

	English	Spanish	Chinese	Japanese	Italian	Greek	Korean	French	Vietnamese
Synth (Ours)	**100**	**88.0**	**100**	**100**	**100**	**94.5**	**100**	**75.5**	**69.5**
Synth (no search)	**100**	0.0	**100**	**100**	**100**	70.0	**100**	0.0	**69.5**
Meta Seq2Seq	68.6	59.1	63.6	46.1	73.7	89.0	45.8	40.0	36.6

```
一 -> 1        x1 万 y1 -> [x1] * 10000 + [y1]
二 -> 2        千 y1    -> 1000 * 1 + [y1]
三 -> 3        x1 千 y1 -> [x1] * 1000 + [y1]
...            百 y1    -> 100 * 1 + [y1]
十 -> 10       x1 百 y1 -> [x1] * 100 + [y1]
百 -> 100      十 y1    -> 10 * 1 + [y1]
千 -> 1000     x1 十 y1 -> [x1] * 10 + [y1]
               u1 x1     -> [u1] + [x1]
```

Figure 7. Induced grammar for Japanese numbers. Given the words for necessary numbers (1-10, 100, 1000, and 10000), as well as 30 random examples, our system is able to recover an interpretable symbolic grammar which can convert Japanese words to integers for any number up to 99,999,999.

to integers (i.e, G(`four thousand five hundred`) $\rightarrow$ 4500, or G(`ciento treinta y siete`) $\rightarrow$ 137). We modified our interpretation grammar to allow for the simple mathematics (multiplication, addition, division, and modulo) necessary to compute integer values. The modified interpretation grammar can be found in the supplement. Using this modified interpretation grammar, we designed a training meta-grammar by examining the number systems for three languages: English, Spanish and Chinese. The meta-grammar includes features common to these three languages, including regular and irregular words for powers of 10 and their multiples, exception words, and features such as zeros or conjunctive words. More details can be found in Section A.3 in the supplement.

We designed the task to mimic how it might be encountered when learning a foreign language: When presented with a core set of "primitive" words, such as the words for 1-20, 100, 1000,[4] and a small number of examples which show how to compose these primitives (e.g., `forty five` $\rightarrow$ 45 shows how to compose `forty` and `five`), an agent should be able to induce a system of rules for decoding the integer meaning of any number word. Therefore, for each train and test episode, we condition each model on a support set of primitive number words and several additional compositional examples. The goal of the model is to learn the system of rules for composing the given primitive words.

Training: We trained our model on programs sampled from the constructed meta-grammar. For each training program, we sampled 60-100 string-integer pairs to use as support examples, and sampled 10 more pairs as held-out query set. We train and test on numbers up to 99,999,999. We trained all models for 12 hours.

[4]This core set of primitive words varies by language. In English, multiples of ten are irregular and would thus be included.

Test Setup: To test our trained model on real languages, we used the PHP international number conversion tool to gather data for several number systems. On the input side, the neural model is trained on a large set of input tokens labeled by ID; at test time, we arbitrarily assign each word in the test language to a specific token ID. Character-level variation, such as elision, omission of final letters, and tone shifts were ignored. For integer outputs, we tokenized integers by digit. For testing, we conditioned on a core set of primitive examples, plus 30 additional compositional examples. At test time, we increased the preference for longer compositional examples compared to the training time distribution, in order to test generalization.

Results: Our results are reported in Figure 3. We test our model on three languages used to build the generative model (English, Spanish and Chinese), and test on six additional unseen languages, averaging over 5 evaluation runs for each. For many languages, our model is able to achieve perfect generalization to the held out query set. The no search baseline is able perform comparably for several languages, however for some (Spanish, French) it is not able to generalize at all to the query set because the generated grammar is invalid and does not parse. The meta seq2seq baseline is outperformed by the synthesis approaches, especially when longer examples are demanded at test time.

5. Conclusion

We present a neuro-symbolic program induction model which can learn rule-based systems from a small set of diverse examples. We demonstrate the effectiveness of our model in three domains: a few-shot artificial language-learning domain previously tested on humans, the SCAN challenge, and number-word learning in several natural languages. In all three domains, the use of a program representation and explicit search provide strong out-of-sample generalization. We believe that explicit rule learning is a key part of human intelligence, and is a necessary ingredient for building human-level and human-like artificial intelligence.

In future work, we hope to learn the symbolic structure of the meta-grammar and interpretation grammar from data, allowing our technique to be applied to a broader range of domains with less supervision. We also aim to build hybrid systems that learn a combination of implicit neural rules and explicit symbolic rules, to capture the dual intuitive and deliberate characteristics of human thought (Kahneman, 2011).

Acknowledgements

The authors gratefully acknowledge Kevin Ellis, Yewen Pu, Luke Hewitt, Tuan Anh Le and Eric Lu for productive conversations and helpful comments. We additionally thank Tuan Anh Le for assistance using the `pyprob` probabilistic programming library. M.N. is supported by an NSF Graduate Fellowship and an MIT BCS Hilibrand Graduate Fellowship.

References

Andreas, J. Good-enough compositional data augmentation. *arXiv preprint arXiv:1904.09545*, 2019.

Balog, M., Gaunt, A. L., Brockschmidt, M., Nowozin, S., and Tarlow, D. Deepcoder: Learning to write programs. *arXiv preprint arXiv:1611.01989*, 2016.

Chen, X., Liu, C., and Song, D. Execution-guided neural program synthesis. 2018.

Devlin, J., Uesato, J., Bhupatiraju, S., Singh, R., Mohamed, A.-r., and Kohli, P. Robustfill: Neural program learning under noisy i/o. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pp. 990–998. JMLR. org, 2017.

Ellis, K., Nye, M., Pu, Y., Sosa, F., Tenenbaum, J., and Solar-Lezama, A. Write, execute, assess: Program synthesis with a repl. In *Advances in Neural Information Processing Systems*, pp. 9165–9174, 2019.

Fischler, M. A. and Bolles, R. C. Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography. *Communications of the ACM*, 24(6):381–395, 1981.

Fodor, J. A. and Pylyshyn, Z. W. Connectionism and cognitive architecture: A critical analysis. *Cognition*, 28:3–71, 1988.

Graves, A., Wayne, G., and Danihelka, I. Neural Turing Machines. *arXiv preprint*, 2014. URL `http://arxiv.org/abs/1410.5401v1`.

Kahneman, D. *Thinking, fast and slow*. Macmillan, 2011.

Kingma, D. P. and Ba, J. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Kratzer, A. and Heim, I. *Semantics in generative grammar*, volume 1185. Blackwell Oxford, 1998.

Lake, B. and Baroni, M. Generalization without systematicity: On the compositional skills of sequence-to-sequence recurrent networks. In *35th International Conference on Machine Learning, ICML 2018*, pp. 4487–4499. International Machine Learning Society (IMLS), 2018.

Lake, B. M. Compositional generalization through meta sequence-to-sequence learning. In *Advances in Neural Information Processing Systems*, pp. 9788–9798, 2019.

Lake, B. M., Linzen, T., and Baroni, M. Human few-shot learning of compositional instructions. *Proceedings of the 41st Annual Conference of the Cognitive Science Society*, 2019.

Le, T. A., Baydin, A. G., and Wood, F. Inference compilation and universal probabilistic programming. *arXiv preprint arXiv:1610.09900*, 2016.

Loula, J., Baroni, M., and Lake, B. M. Rearranging the familiar: Testing compositional generalization in recurrent networks. *arXiv preprint arXiv:1807.07545*, 2018.

Luong, M.-T., Pham, H., and Manning, C. D. Effective approaches to attention-based neural machine translation. *arXiv preprint arXiv:1508.04025*, 2015.

Marcus, G. Deep Learning: A Critical Appraisal. *arXiv preprint*, 2018.

Marcus, G. and Davis, E. *Rebooting AI: Building Artificial Intelligence We Can Trust*. Knopf Doubleday Publishing Group, 2019. ISBN 9781524748265. URL `https://books.google.com/books?id=OmeEDwAAQBAJ`.

Marcus, G. F. Rethinking Eliminative Connectionism. *Cognitive Psychology*, 282(37):243–282, 1998.

Marcus, G. F. *The Algebraic Mind: Integrating Connectionism and Cognitive Science*. MIT Press, Cambridge, MA, 2003.

Murali, V., Qi, L., Chaudhuri, S., and Jermaine, C. Neural sketch learning for conditional program generation. *arXiv preprint arXiv:1703.05698*, 2017.

Nye, M., Hewitt, L., Tenenbaum, J., and Solar-Lezama, A. Learning to infer program sketches. In *International Conference on Machine Learning*, pp. 4861–4870, 2019.

Russin, J., Jo, J., O'Reilly, R. C., and Bengio, Y. Compositional generalization in a deep seq2seq model by separating syntax and semantics. *arXiv preprint*, 2019. URL `http://arxiv.org/abs/1904.09708`.

Yang, K. and Deng, J. Learning to prove theorems via interacting with proof assistants. *arXiv preprint arXiv:1905.09381*, 2019.

Zohar, A. and Wolf, L. Automatic program synthesis of long programs with a learned garbage collector. In *Advances in Neural Information Processing Systems*, pp. 2094–2103, 2018.

A. Supplementary Material

A.1. Experimental details: MiniSCAN

Meta-grammar As discussed in the main text, each grammar contained 3-4 *primitive* rules and 2-4 *higher-order* rules. Primitive rules map a word to a color (e.g. `dax -> RED`), and higher order rules encode variable transformations given by a word (e.g. `x1 kiki x2 -> [x2] [x1]`). In a higher-order rule, the left hand side can be one or two variables and a word, and the right hand side can be any sequence of bracketed forms of those variables. The last rule of every grammar is a concatenation rule: `u1 x1 -> [u1] [x1]`, which dictates how a sequence of tokens can be concatenated. Figure 8 shows several example training grammars sampled from the meta-grammar.

Generating input-output examples To generate a set of support input-output sequences $\mathcal{X}$ from a program G, we uniformly sample a set of input sequences from the CFG formed by the left hand side of each rule in G. We then apply the program G to each input sequence x_i to find the corresponding output sequence $y_i = G(x_i)$. This gives a set of examples $\{(x_i, y_i)\}$, which we can divide into support examples and query examples.

A.2. Experimental details: SCAN

Meta-grammar The meta-grammar used to train networks for SCAN is based on the meta-grammar used in the MiniSCAN experiments above. Each grammar has between 4 and 9 primitives and 3 and 7 higher order rules. This meta-grammar has two additional differences from the MiniSCAN meta-grammar, allowing it to produce grammars which solve SCAN:

1. Primitives can rewrite to empty tokens, e.g., `turn -> 'EMPTY_STRING'`.
2. The last rule for each grammar can either be the standard concatenation rule above, or, with 50% probability, a different concatenation rule: `u1 u2 -> [u2] [u1]`, which acts only on two adjacent single primitives. This is to ensure that the SCAN grammar, which does not support general string concatenation, is within the support of the training meta-grammar, while maintaining compatibility with MiniSCAN grammars.

Example training grammars sampled from the meta-grammar are shown in Figure 9.

At training time, we use the same process as for MiniSCAN to sample input-output examples for the support and query set.

Selecting support examples at test time The distribution of input-output example sequences in each SCAN split is very different than the training distribution. Therefore, selecting a random subset of 100 examples uniformly from the SCAN training set would lead to a support set very different from support sets seen during training. We found that two methods of selecting support examples from each SCAN training set allowed us to achieve good performance:

1. To ensure that support sets during testing matched the distribution of support sets during training, we selected our test-time support examples to match the empirical distribution of input sequence lengths seen at training time. We used rejection sampling to ensure consistent sequence lengths at train and test time.
2. We found that results were improved when words associated with longer sequences were seen in more examples in the test-time support set. Therefore, we upweighted the probability of seeing the words 'opposite' and 'around' in the support set.

The implementation details of support example selection can be found in `generate_episode.py`

Results Table 4 and Table 5 show the numerical results for the SCAN experiments reported with standard error. Table 6 shows the fixed example budget results, averaged over 20 evaluation runs.

A.3. Experimental details: Number Words

Meta-grammar We designed a meta-grammar for the number domain, relying on knowledge of English, Spanish, and Chinese. We assume a base 10 number system, where powers of 10 can have "regular" words (e.g., "one hundred", "two hundred", "three hundred") or "irregular" words ("ten", "twenty", "thirty"). Additional features include exceptions to regularity, conjunctive words (e.g., "y" in Spanish), and words for zero. The full model can be found in `pyro_num_distribution.py`, and example training grammars are shown in Figure 10.

Generating input-output examples For each grammar, example pairs (x_i, y_i) come in two categories: a core set of "necessary" primitive words, and a set of compositional examples.

1. Necessary words: The core set of "necessary words" are analogous to the primitives for the MiniSCAN and SCAN domains. This set comprises examples with only one token as well as examples for powers of 10. For both training and testing, we produce an example for every necessary word in the language. For the synthesis

```
G =
mup -> BLACK
kleek -> WHITE
wif -> PINK
u2 dax u1 -> [u1] [u1] [u2]
u1 lug -> [u1]
x1 gazzer -> [x1]
u2 dox x1 -> [x1] [u2]
u1 x1 -> [u1] [x1]

G =
tufa -> PINK
zup -> RED
gazzer -> YELLOW
kleek -> PURPLE
u2 mup x2 -> [u2] [x2]
x2 dax -> [x2]
u2 lug x2 -> [u2] [x2]
u1 dox -> [u1] [u1] [u1]
u1 x1 -> [u1] [x1]

G =
gazzer -> PURPLE
wif -> BLACK
lug -> GREEN
x2 kiki -> [x2] [x2]
x1 dax x2 -> [x2] [x1]
x1 mup x2 -> [x2] [x1] [x2] [x1] [x1]
u1 x1 -> [u1] [x1]
```

Figure 8. Samples from the training meta-grammar for MiniSCAN.

models, we automatically convert the core primitive examples into rules.

2. Compositional examples: At test time, to provide random compositional examples for each language, we sample numbers from a distribution over integers and convert them to words using the `NumberFormatter` class (see `convertNum.php`). To ensure a similar process during training time, to produce compositional example pairs (x_i, y_i) for a training grammar G, we sample numbers y_i from a distribution over integers. We then construct the inverse grammar G^{-1}, which transforms integers to words, and use this to find the input sequence examples $x_i = G^{-1}(y_i)$. At test time, the compositional example distribution is slightly modified to encourage longer compositional examples. The sampling distribution can be found in `test_langs.py`. At training time, we produce between 60 and 100 compositional examples for the support set, and 10 for the held out query set. At test time, we produce 30 compositional examples for the support set and 30-70 examples for the held out query set.

Results Table 7 shows the results in the number word domain with standard error, averaged over 5 evaluation runs for each language.

```
G =
turn -> GREEN
left -> BLUE
right -> WALK
thrice -> RUN
blicket -> RED
u2 and x1 -> [x1] [x1] [x1] [u2] [u2] [u2] [x1]
u1 after x2 -> [u1] [u1] [x2] [x2] [u1] [x2] [x2]
u2 opposite -> [u2] [u2]
u1 lug x2 -> [u1] [x2]
u1 x1 -> [u1] [x1]

G =
and -> JUMP
kiki -> LTURN
blicket -> BLUE
walk -> LOOK
thrice -> RED
run -> GREEN
dax -> RUN
after -> RTURN
x2 twice u1 -> [u1] [x2] [x2] [x2] [x2]
u2 right x1 -> [x1] [u2] [u2]
u1 look x2 -> [u1] [x2] [x2]
u1 jump -> [u1] [u1]
u2 turn u1 -> [u2] [u1]
u1 lug -> [u1] [u1]
x2 left u1 -> [x2] [u1]
u1 x1 -> [u1] [x1]

G =
twice -> WALK
jump -> RTURN
turn -> JUMP
walk ->
blicket -> GREEN
kiki -> RUN
right -> RED
run -> BLUE
x2 left -> [x2] [x2] [x2] [x2] [x2]
x1 dax u1 -> [u1] [x1] [u1]
u1 thrice x2 -> [u1] [x2] [x2] [u1] [u1]
x1 look u2 -> [x1] [x1] [u2] [x1]
x2 around -> [x2]
u1 u2 -> [u2] [u1]

G =
twice -> WALK
jump -> RTURN
turn -> JUMP
walk ->
blicket -> GREEN
kiki -> RUN
right -> RED
run -> BLUE
x2 left -> [x2] [x2] [x2] [x2] [x2]
x1 dax u1 -> [u1] [x1] [u1]
u1 thrice x2 -> [u1] [x2] [x2] [u1] [u1]
x1 look u2 -> [x1] [x1] [u2] [x1]
x2 around -> [x2]
u1 u2 -> [u2] [u1]
```

Figure 9. Samples from the training meta-grammar for SCAN.

Table 4. Accuracy on SCAN splits with standard error.

	length	simple	jump	right
Synth (Ours)	**100**	**100**	**100**	**100**
Synth (no search)	0.0	13.3 (3.3)	3.5 (0.7)	0.0
Meta Seq2Seq	0.04 (0.02)	0.88 (0.13)	0.51 (0.06)	0.03 (0.03)
MCMC	0.02 (0.01)	0.0	0.01 (0.01)	0.01 (0.01)
Sample from prior	0.04 (0.02)	0.03 (0.03)	0.03 (0.02)	0.01 (0.01)

Table 5. Required search budget for our synthesis model on SCAN, with standard error.

	length	simple	jump	right
Search time (sec)	39.1 (11.9)	33.7 (10.0)	74.6 (48.5)	36.1 (13.4)
Number of prog. seen	1516 (547)	1296 (358.2)	2993 (1990.1)	1466 (541)
Number of ex. used	149.4 (28.9)	144.8 (24.7)	209.2 (91.3)	143.8 (28.6)

Table 6. Accuracy on SCAN splits, using a fixed budget of 100 examples.

Model	length	simple	jump	right
Synth (180 s)	**100**	**100**	**43.3** (10.0)	**98.4** (1.6)
Synth (120 s)	**100**	98.4 (1.6)	**53.9** (10.3)	94.2 (2.9)
Synth (60 s)	92.2 (3.8)	97.5 (1.3)	**44.3** (9.6)	80.75 (6.8)
Synth (30 s)	85.6 (4.6)	95.6 (2.3)	24.2 (8.6)	60.0 (8.7)

Table 7. Accuracy on few-shot number-word learning, using a maximum timeout of 45 seconds. Results shown with standard error over 5 evaluation runs.

Model	English	Spanish	Chinese	Japanese	Italian	Greek	Korean	French	Vietnamese
Synthesis (Ours)	**100**	**88.0** (3.8)	**100**	**100**	**100**	**94.5** (4.9)	**100**	**75.5** (2.4)	**69.5** (2.3)
Synthesis (no search)	**100**	0.0	**100**	**100**	**100**	70.0 (10.2)	**100**	0.0	**69.5** (2.3)
Meta Seq2Seq	68.6 (10.0)	59.1 (4.5)	63.6 (4.0)	46.1 (3.5)	73.7 (3.2)	89.0 (2.5)	45.8 (3.7)	40.0 (5.3)	36.6 (6.2)

```
G =
token14 -> 1
token16 -> 2
token50 -> 3
token31 -> 4
token49 -> 5
token28 -> 6
token17 -> 7
token03 -> 8
token06 -> 9
token14 token10 -> 10
token13 -> 100
token14 token36 -> 1000
token01 -> 1000000
token08 y1 -> 1000000* 1 + [y1]
token05 token01 y1 -> 1000000* 9 + [y1]
x1 token01 y1 -> [x1]*1000000 + [y1]
x1 token36 y1 -> [x1]*1000 + [y1]
token32 y1 -> 100* 1 + [y1]
x1 token13 y1 -> [x1]*100 + [y1]
x1 token10 y1 -> [x1]*10 + [y1]
u1 token09 x1 -> [u1] + [x1]
u1 x1 -> [u1] + [x1]

G =
token20 -> 1
token22 -> 2
token37 -> 3
token14 -> 4
token01 -> 5
token13 -> 6
token48 -> 7
token05 -> 8
token16 -> 9
token47 -> 10
token07 -> 20
token08 -> 30
token35 -> 40
token02 -> 50
token40 -> 60
token31 -> 70
token43 -> 80
token29 -> 90
token20 token38 -> 100
token20 token18 -> 1000
token20 token33 -> 10000
token28 token33 y1 -> 10000* 7 + [y1]
x1 token33 y1 -> [x1]*10000 + [y1]
x1 token18 y1 -> [x1]*1000 + [y1]
x1 token38 y1 -> [x1]*100 + [y1]
u1 x1 -> [u1] + [x1]
```

Figure 10. Samples from the training meta-grammar for number word learning. Note that the model is trained on a large set of generic input tokens labeled by ID. At test time, we arbitrarily assign each word in the test language to a specific token ID.