UC Berkeley
UC Berkeley Electronic Theses and Dissertations

Title

Learning-Based Program Synthesis: Towards Synthesizing Complex Programs from Multi-Modal Specifications in the Wild

Permalink

https://escholarship.org/uc/item/74d6s4sd

Author

Chen, Xinyun

Publication Date

2022

Peer reviewed|Thesis/dissertation

Learning-Based Program Synthesis: Towards Synthesizing Complex Programs from Multi-Modal Specifications in the Wild

by

Xinyun Chen

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Dawn Song, Chair
Professor Alvin Cheung
Professor Bruno Olshausen

Spring 2022

Learning-Based Program Synthesis: Towards Synthesizing Complex Programs from Multi-Modal Specifications in the Wild

Abstract

Learning-Based Program Synthesis: Towards Synthesizing Complex Programs from
Multi-Modal Specifications in the Wild

by

Xinyun Chen

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Dawn Song, Chair

With the advancement of modern technologies, programming becomes ubiquitous not only among professional software developers, but also for general computer users. However, gaining programming expertise is time-consuming and challenging. Therefore, program synthesis has many applications, where the computer automatically synthesizes programs from specifications such as natural language descriptions and input-output examples. In this dissertation, we present our work on learning-based program synthesis, where we demonstrate deep learning techniques for synthesizing programs from different specification formats.

First, we present our work on synthesizing programs from multi-modal specifications with real-world applications. In particular, our SpreadsheetCoder work has been integrated into Google Sheets to support the formula suggestion feature, showing the power of learning-based program synthesis in real products. Second, we present our work on execution-guided program synthesis, which brings significant performance gain for synthesizing more complex programs from input-output examples. Our work on program translation and code optimization then demonstrate the importance of representing the program structures and designing learning algorithms correspondingly, which improve the generalization of the learned model and the complexity of programs that can be correctly generated. Finally, our work on neural-symbolic frameworks show that integrating symbolic components into neural networks empower the models with better reasoning and generalization capabilities.

To my parents.

Contents

List of Figures

List of Tables

Acknowledgments

I am very grateful for all the support and help I have received in my Ph.D. journey. First, I would like to thank my advisor Dawn Song. I have been working with her since my junior undergraduate year when I interned in her lab as a visiting student. Our first project is to design deep neural networks to translate natural language descriptions into a single program statement, which led to my first publication at top-tier conferences as the first author. I became deeply interested in learning-based program synthesis, and continued working in this field for my Ph.D. research. Dawn's support, guidance and encouragement are crucial in guiding me to build my own research agenda.

Next, I would like to thank Alvin Cheung and Bruno Olshausen for serving on both my dissertation committee and qualifying exam committee, and thank Koushik Sen for being the chair of my qualifying exam. Thank you all for giving many insightful comments and suggestions on my thesis research. I also enjoy my collaboration with Alvin Cheung on our PlotCoder project for synthesizing visualization code in Python Jupyter notebooks, where I learn more about the challenges of dealing with ambiguous program specifications for program synthesis in the wild.

I am fortunate to have the opportunities of interning at several great industrial research labs and working with amazing people. First, I want to thank Yuandong Tian and Denny Zhou, who hosted my internships at Meta AI and Google Brain. I met both of them when I was a senior undergraduate, and I didn't expect that I will have long-term collaboration with them throughout my Ph.D. journey. They not only closely work with me on my internship projects, but also give me a lot of career advice and regularly chat with me even if my internships already ended. Their mentorship has been playing a key role in my Ph.D. life.

Meanwhile, I would like to thank Petros Maniatis, Rishabh Singh and Charles Sutton, who are my mentors in the program synthesis team of Google Brain. I am super excited about our SpreadsheetCoder work, which shows the production impact of our developed techniques. The success will not be possible without the support from many people in Google Brain and Google Sheets teams. I also want to thank Yujia Li, who introduced me to the DeepMind AlphaCode team and brought me into this project in my internship. Building an agent that outperforms human programmers on competitive programming has been a dream since I started working on program synthesis, and I am excited to join the effort of tackling this challenge. It is a wonderful experience working in this amazing team, and I have been consistently impressed by the rapid progress of the team.

I would like to thank my colleagues and friends in Dawn's group for hanging out together and chatting about research and life, including Min Du, Linyuan Gong, Ruoxi Jia, Bo Li, Chang Liu, Jian Liu, Xiaoyuan Liu, Richard Shin, Chenguang Wang, Lun Wang, Tiancheng Xie, Gai Yu, Jiaheng Zhang, Siyuan Zhuang, and others. I also want to thank my collaborators and friends outside Berkeley, including Bo Dai, Hanjun Dai, Cheng Fu, Yujian Gan, Chen Liang, Hongyu Ren, Hui Shi, Adams Wei Yu, and many others.

Finally, I would like to thank my parents for their unconditional love and support.

Chapter 1

Introduction

Nowadays, programming is ubiquitous among not only professional software developers, but also general computer users. However, gaining programming expertise is time-consuming and challenging. Therefore, program synthesis has many applications, where the computer automatically synthesizes the programs from the specification, i.e., the required program functionality that can be described in various formats such as natural language descriptions and input-output examples. Program synthesis transforms the way we interact with computers, which makes programming more friendly and accessible to general users, aids data scientists in data processing, and improves the programming efficiency of software developers.

Classic program synthesis techniques are largely based on heuristic-guided search and rule-based generation. These hand-engineered systems require a lot of manual effort to tune the search heuristics and synthesis rules for different applications, and they are not capable of handling program specifications that are noisy and less well-defined, such as natural language descriptions. On the other hand, recent advancements in deep learning have shown impressive performance in a variety of areas. With abundant open-source projects available online, deep neural networks have become a great fit for representing different specification formats and efficiently learning the synthesis rules from data. In the end, learning-based techniques are necessary for broadening the impact of program synthesis.

Despite the recent progress of program synthesis, including learning-based approaches, prior works still suffer from limited complexity and generalizability, i.e., the predicted programs tend to become inconsistent with the specification when the specification and its corresponding program are long and complicated. Meanwhile, understanding heterogeneous specification formats remains a challenge for the real-world deployment of program synthesizers.

In this dissertation, we present deep learning-based techniques towards addressing the aforementioned challenges and demonstrating the real-world impact of learning-based program synthesis. Meanwhile, our work also aim to address the core challenges of artificial intelligence and machine learning regarding generalization, compositionality, interpretability, and reasoning. Specifically, we have designed neural-symbolic frameworks that interleave neural and symbolic modules, which learn to produce problem solutions represented as programs.

In terms of program synthesis applications, we present learning-based program synthesis

approaches to synthesize programs from various types of specifications, including natural language descriptions (Part I), input-output examples (Part II), and reference programs (Part III). Our SpreadsheetCoder model, which predicts spreadsheet formulas from the tabular context, was integrated into Google Sheets. The formula suggestion feature could potentially benefit hundreds of millions of users, and makes data analysis easier and more efficient (Chapter 2). Meanwhile, our execution-guided synthesis technique brings significant performance gains for synthesizing more complex programs from input-output examples (Chapters 4 and 5) .

Furthermore, we also introduce a new methodology to reason over data via program synthesis, where the neural networks are trained to make predictions as programs (Part IV). Existing deep neural networks have been primarily designed to learn what to predict, instead of the rationale behind the predictions. As a result, despite the remarkable success of deep neural networks in various applications, they are insufficient for more complex reasoning beyond superficial pattern matching, such as numerical calculation and logical reasoning. Furthermore, deep neural networks have exposed limitations in generalization, even if the test input only slightly deviates from the training distribution. Facing these challenges of reasoning and generalization, we have developed neural-symbolic techniques that empower neural networks with the ability to synthesize programs that represent the reasoning process. By integrating the symbolic component into deep neural networks, our neural-symbolic reader demonstrates decent performance on challenging numerical reasoning over text, which is not naturally achievable even with massive pre-training (Chapter 8). Meanwhile, our neural-symbolic stack machines learn execution traces that reveal the compositional rules for language understanding, which achieve full generalization to unseen test cases (Chapter 9).

Part I

Synthesis with Natural Language

Chapter 2

SpreadsheetCoder: Formula Prediction from Semi-structured Context

Spreadsheet formula prediction has been an important program synthesis problem with many real-world applications. Previous works typically utilize input-output examples as the specification for spreadsheet formula synthesis, where each input-output pair simulates a separate row in the spreadsheet. However, this formulation does not fully capture the rich context in real-world spreadsheets. First, spreadsheet data entries are organized as tables, thus rows and columns are not necessarily independent from each other. In addition, many spreadsheet tables include headers, which provide high-level descriptions of the cell data. However, previous synthesis approaches do not consider headers as part of the specification. In this chapter, we present the first approach for synthesizing spreadsheet formulas from tabular context, which includes both headers and semi-structured tabular data. In particular, we propose SpreadsheetCoder, a BERT-based model architecture to represent the tabular context in both row-based and column-based formats. We train our model on a large dataset of spreadsheets, and demonstrate that SpreadsheetCoder achieves top-1 prediction accuracy of 42.51%, which is a considerable improvement over baselines that do not employ rich tabular context. Compared to the rule-based system, SpreadsheetCoder assists 82% more users in composing formulas on Google Sheets [1].

2.1 Introduction

Spreadsheets are ubiquitous for data storage, with hundreds of millions of users. Helping users write formulas in spreadsheets is a powerful feature for data analysis. Although spreadsheet formula languages are relatively simpler than general-purpose programming languages for data manipulation, writing spreadsheet formulas could still be tedious and error-prone for end users [98, 111, 59]. Systems such as FlashFill [98, 99] help end-users perform string transformation tasks in spreadsheets using a few input-output examples by automatically

---
[1]The material in this chapter is based on Chen et al. [57].

synthesizing a program in a domain-specific language (DSL). Recently, several learning approaches based on different neural architectures have been developed for learning such programs from examples, and have demonstrated promising results [201, 74, 252].

All these previous works formalize the spreadsheet program prediction problem as a *programming by example* task, with the goal of synthesizing programs from a small number of input-output examples. We argue that this choice engenders three key limitations. First, this setup assumes that each data row is independent, and each formula is executed on data cells of the same row. However, real spreadsheets are less structured than this. Data in spreadsheets is typically organized as semi-structured tables, and cells in different rows could be correlated. As shown in Figure 2.1, in the same table, different data blocks could have different structures, and formulas can take cell values in other rows as function arguments. Second, because spreadsheets are semi-structured, they also contain rich metadata. In particular, many spreadsheet tables include headers that provide high-level descriptions of the data, which could provide important clues for formula prediction. However, table headers are not utilized in prior work. Finally, programming-by-example methods output programs in a DSL, which is typically designed to facilitate synthesis, and is much less flexible than the language in which users write formulas. For example, the FlashFill DSL only covers a subset of spreadsheet functions for string processing, and it does not support rectangular ranges, a common feature of spreadsheet formulas. In contrast, spreadsheet languages also support a wide variety of functions for numerical calculation, while the argument selection is more flexible and takes the spreadsheet table structure into account. In total, these limitations can compromise the applicability of such prior efforts to more diverse real-world spreadsheets and to richer language functionality.

Instead, we propose synthesizing spreadsheet formulas *without* an explicit specification. To predict a formula in a given cell, the context of data and metadata is used as an *implicit* (partial) specification of the desired program. For example (Figure 2.1b), if predicting a formula at the end of a column of numbers labeled "Score", and a cell in the same row contains the text "Total", this context might specify the user's intent to compute a column sum. Our problem brings several new challenges compared to related work in programming by example [98, 35, 22], semantic parsing [211, 292, 283] and source code completion [215, 157, 242]. Spreadsheet tables contain rich two-dimensional relational structure and natural language metadata, but the rows do not follow a fixed schema as in a relational database. Meanwhile, our tabular context is more ambiguous as the program specification, and the spreadsheet language studied in this work is more flexible than languages studied in the program synthesis literature.

In this paper, we present SpreadsheetCoder, a neural network architecture for spreadsheet formula prediction. SpreadsheetCoder encodes the spreadsheet context in its table format, and generates the corresponding formula in the target cell. A BERT-based encoder [72] computes an embedding vector for each input token, incorporating the contextual information from nearby rows and columns. The BERT encoder is initialized from the weights pre-trained on English text corpora, which is beneficial for encoding table headers. To handle cell references, we propose a two-stage decoding process inspired by sketch learning for program
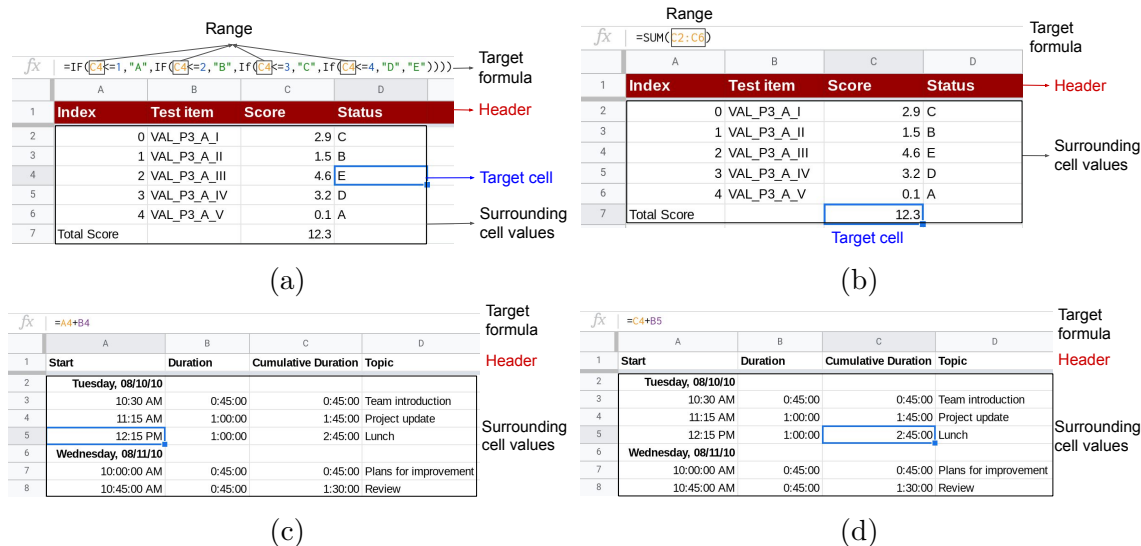
Figure 2.1: Illustrative synthetic examples of our spreadsheet formula prediction setup. (a): The formula manipulates cell values in the same row. (b): The formula is executed on the rows above. (c) and (d): Formulas involve cells in different rows and columns. The data value in the target cell is excluded from the input. All of these formulas can be correctly predicted by our model.

synthesis [234, 185, 79, 192]. Our decoder first generates a formula sketch, which does not include concrete cell references, and then predicts the corresponding cell ranges to generate the complete formula.

For evaluation (Section 2.4), we construct a large-scale benchmark of spreadsheets publicly shared within our organization. We show that SpreadsheetCoder outperforms neural network approaches for programming by example [74], and achieves 42.51% top-1 full-formula accuracy, and 57.41% top-1 formula-sketch accuracy, both of which are already high enough to be practically useful. In particular, SpreadsheetCoder assists 82% more users in composing formulas than the rule-based system on Google Sheets. Moreover, SpreadsheetCoder can predict cell ranges and around a hundred different spreadsheet operators, which is much more flexible than DSLs used in prior works. With various ablation experiments, we demonstrate that both implicit specification from the context and text from the headers are crucial for obtaining good performance.

2.2 Problem Setup

In this section, we discuss the setup of our spreadsheet formula prediction problem. We first describe the input specification, then introduce the language and representation for spreadsheet formulas.

Input specification. We illustrate the input context in Figure 2.1. The input context consists of two parts: (a) context surrounding the target cell (e.g., all cell values in rows 2–7, and columns A–D, excluding cell D4 in Figure 2.1a), and (b) the header row (e.g., row 1).

In contrast to prior programming-by-example approaches [98, 201, 74, 252], our input specification features (a) tabular input, rather than independent rows as input-output examples, and (b) header information. Tabular input is important for many cases where formulas are executed on various input cells from different rows and columns (Figure 2.1), and headers hold clues about the purpose of a column as well as its intended type, e.g, the header cell "Score" in Figure 2.1b is likely to indicate that the column data should be numbers.

Note that we do not include the intended *output* of the target cell in our input specification, for three reasons. First, unlike programming-by-example problems, we do not have multiple independent input-output examples available from which to induce a formula, so providing *multiple* input-output examples is not an option. Second, even for our single input instance, the evaluated formula value may not be known by the spreadsheet user yet. Finally, we tried including the intended formula execution *result* in our specification, but it did not improve the prediction accuracy beyond what the contextual information alone allowed.

The spreadsheet language. Our model predicts formulas written in the Google Sheets language[2]. Compared to the domain-specific language defined in FlashFill, which focuses on string transformations, the spreadsheet language supports a richer set of operators. Besides string manipulation operators such as `CONCATENATE`, `LOWER`, etc., the spreadsheet language also includes operators for numerical calculations (e.g., `SUM` and `AVERAGE`), table lookups (e.g., `VLOOKUP`) and conditional statements (`IF`, `IFS`). As will be discussed in Section 2.4, around a hundred different base formula functions appear in our dataset, many more than the operators defined in the FlashFill DSL.

In this work, we limit our problem to formulas with references to *local* cells in a spreadsheet tab, thus we exclude formulas with references to other tabs or spreadsheets, and absolute cell ranges. As will be discussed in Section 2.3, we also exclude formulas with relative cell references outside a bounded range, i.e., farther than $D = 10$ rows and columns in our evaluation. We consider improving the computational efficiency to support larger D and enabling the synthesis of formulas with more types of cell references as future work.

Formula representation. One of the key challenges in formula representation is how to represent cell references, especially ranges, which are prevalent in spreadsheet formulas. Naively using the absolute cell positions, e.g., `A5`, may not be meaningful across different spreadsheets. Meanwhile, a single spreadsheet can have millions of cells, thus the set of possible ranges is very large.

To address this, we design a representation for formula sketches inspired by prior work on sketch learning for program synthesis [234, 185, 79, 192]. A formula sketch includes every token in the prefix representation of the parse tree of the spreadsheet formula, except for cell references. References, which can be either a single cell or a range of cells, are replaced with a special placeholder `RANGE` token. For example, the sketch of the formula in Fig-

---

[2]Google Sheets function list: `https://support.google.com/docs/table/25273?hl=en`.

```
<Range>  ::=  $R$ <R> <C> $ENDR$
          |   $R$ <R> <C> $SEP$ <R> <C> $ENDR$
    <R>  ::=  R[-10] | R[-9] | ...R[9] | R[10]
    <C>  ::=  C[-10] | C[-9] | ...C[9] | C[10]
```

Figure 2.2: The full grammar for range representation.

ure 2.1a is `IF <= RANGE 1 "A" IF <= RANGE 2 "B" IF <= RANGE 3 "C" IF <= RANGE 4 "D" "E" $ENDSKETCH$`, where `$ENDSKETCH$` denotes the end of the sketch. Notice that the sketch includes literals, such as the constants `1` and `"A"`.

To complete the formula representation, we design an intermediate representation for ranges, *relative* to the target cell, as shown in Figure 2.2. For example, `B5` in Figure 2.1c is represented as `$R$ R[0] C[1] $ENDR$` since it is on the next column but the same row as the target cell `A5`, and range `C2:C6` in Figure 2.1b is represented as `$R$ R[-5] C[0] $SEP$ R[-1] C[0] $ENDR$`. The special tokens `$R$` and `$ENDR$` start and conclude a concrete range, respectively, and `$SEP$` separates the beginning and end (relative) references of a rectangular multi-cell range.

A complete spreadsheet formula includes both the sketch and any concrete ranges; e.g., the formula in Figure 2.1b is represented as `SUM RANGE $ENDSKETCH$ $R$ R[-5] C[0] $SEP$ R[-1] C[0] $ENDR$ EOF`, where `EOF` denotes the end of the formula. In Section 2.3, we will discuss our two-stage decoding process, which sequentially predicts the formula sketch and ranges.

2.3   SpreadsheetCoder Model Architecture

In this section, we present our SpreadsheetCoder model architecture for spreadsheet formula prediction. We provide an overview of our model design in Figure 2.3.

Tabular Context Encoder

Input representation. Our model input includes the surrounding data values of the target cell as a table, and the first row is the header. When there is no header in the spreadsheet table, we set the header row to be an empty sequence. We include data values in cells that are at most D rows and D columns away from the target cell, so that the input dimension is $(2D + 2) \times (2D + 1)$, and we set $D = 10$ in our experiments.

Row-based BERT encoder. We first use a BERT encoder [72] to compute a row-based contextual embedding for each token in the target cell's context. Since our $2D + 1 + 1$ rows contain many tokens and we use a standard BERT encoder of 512-token inputs, we *tile* our rows into bundles of $N = 3$ adjacent data rows, plus the header row, which is included in every bundle. Then we compute a token-wise BERT embedding for each bundle separately;

Skip Connection

Rowwise
Convolution

Row-Based
BERT Encoder

Columnwise
Convolution

Header
Attention

Sketch
Predictor

Input Table

Aggregated
Token-wise
Embedding

LSTM
Decoder

Column-Based
BERT Encoder

Rowwise
Convolution

Columnwise
Convolution

Cell Data
Attention

Range
Predictor

Skip Connection
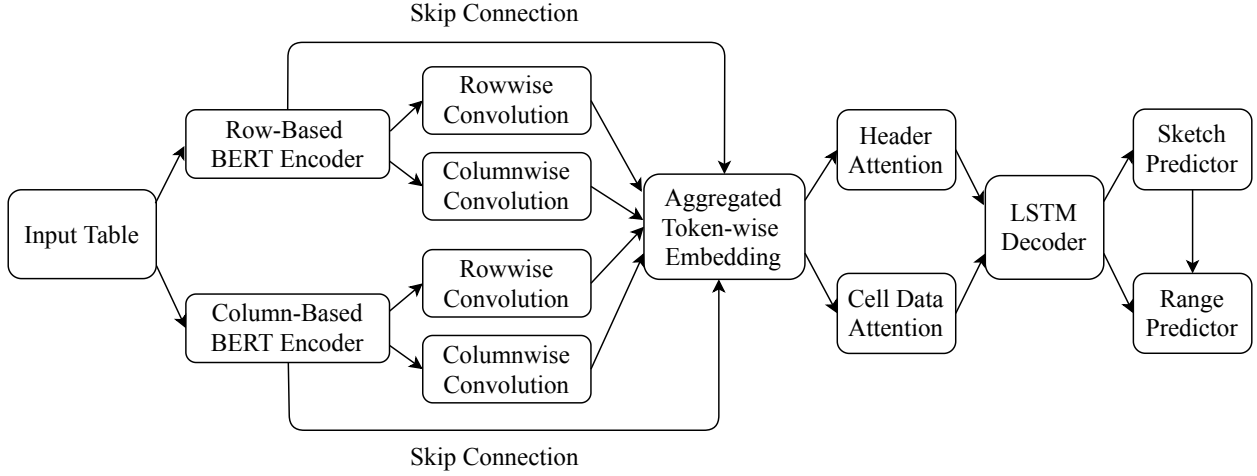
Figure 2.3: An overview of our model architecture.

the BERT weights are initialized from a pre-trained checkpoint for English. Specifically, in our experiments where $D = 10$, we concatenate all cell values for each row i in the context into a token sequence R_i, which has length $L = 128$ (we trim and pad as needed). We combine rows in bundles $S_{rb} = [H_r, R_{3b-1}, R_{3b}, R_{3b+1}]$, for $b \in [-3, 3]$; here H_r is the header row. We set the BERT segment IDs to 0 for the header tokens, and 1 for data tokens in each bundle. There are $2D + 1 = 21$ rows of context, so each of the 21 data rows is covered exactly once by the seven bundles. The header row is assigned a different BERT representation in each bundle. To obtain a single representation of the header row, we average per token across the embeddings from all of the bundles.

The number of data rows $N = 3$ is set to seek the balance between the size of the tabular context fed into the encoder and the computational efficiency. Since the BERT we use takes 512 input tokens, we can feed at most $L = 512/(N + 1)$ tokens per row. To generate formulas referring to cells within $D = 10$ rows and columns, $L = 128$ is a good fit in our evaluation. If we further decrease N and increase L, it imposes extra computational overhead due to more forward passes over BERT $(21/N)$.

Column-based BERT encoder. As shown in Figure 2.1b, some formulas manipulate cells in the same column, in which case a column-based representation may be more desirable. Therefore, we also compute a column-based contextual embedding for all context tokens. We perform similar tiling as for the row-based BERT encoding, yielding column bundles S_{cb} for $b \in [-3, 3]$. Unlike with row-wise tiling, where we include the header row H_r with every bundle, for column-wise tiling we use the column of the target cell, $H_c = C_0$, as the "header column" in every bundle. After obtaining all token embeddings from this tiled computation by the BERT encoder, we discard token embeddings of C_0 in its role as header column, and only use its regular token embeddings from the bundle S_{c0}.

Row-wise and column-wise convolution layers. Although the output vectors of BERT encoders already contain important contextual information, such as headers, nearby rows and columns, they still do not fully embed the entire input table as the context. To encode the context from more distant rows and columns, we add a row-wise convolution layer and a column-wise convolution layer on top of each BERT encoder. Specifically, the row-wise convolution layer has a kernel size of $1 \times L$, and the column-wise convolution layer has a kernel size of $(2D+2) \times 1$ for row-based BERT, and $(2D+1) \times 1$ for column-based BERT. In this way, the convolution layer aggregates across BERT embeddings from different bundles, allowing the model to take longer range dependencies into account. For each input token, let e_b be its BERT output vector, c_r be the output of the row-wise convolution layer, and c_c be the output of the column-wise convolution layer. The final embedding of each input token is the concatenation of the BERT output and the output of convolution layers, i.e., $e = [c_r + c_c; e_b]$.

Two-stage Formula Decoder

We train an LSTM [115] decoder to generate the formula as a token sequence. Meanwhile, we use the standard attention mechanism [20] to compute two attention vectors, one over the input header, and one over the cell data. We concatenate these two attention vectors with the LSTM output, and feed them to a fully-connected layer with the output dimension $|V|$, where $|V|$ is the vocabulary size of formula tokens. Note that the token vocabularies are different for sketches (formula operators, literals, and special tokens) and ranges (relative row and column tokens and special range tokens). The output token prediction is computed with the softmax.

As mentioned in Section 2.2, we design a two-stage decoding process, where the decoder first generates the formula sketch, and then predicts the concrete ranges. In the first stage, the sketch is predicted as a sequence of tokens by the LSTM, and the prediction terminates when an `$ENDSKETCH$` token is generated. Then in the second stage, the range predictor sequentially generates formula ranges corresponding to each `RANGE` token in the sketch, and the prediction terminates when an `EOF` token is generated. Both sketch and range predictors share the same LSTM, but with different output layers.

2.4 Experiments

We evaluate SpreadsheetCoder on spreadsheet formula prediction tasks in different settings. We first describe our dataset, then introduce our experimental setup and discuss the results [3].

---

[3]The code and data are available at `https://github.com/google-research/google-research/tree/master/spreadsheet_coder`.

Dataset

We constructed our dataset from a corpus of Google Sheets publicly shared within our organization. We collected 46K Google Sheets with formulas, and split them into 42K for training, 2.3K for validation, and 1.7K for testing.

Although in principle, our model could generate formulas using any operator in the spreadsheet language, some kinds of value references are impossible to predict from local context, thus we remove formulas with such values from our dataset. Specifically, we exclude formulas that use the HYPERLINK function with a literal URL, since those are merely "stylistic" formulas that perform no computation beyond presenting a URL as a clickable link. As discussed in Section 2.2, we also filtered out formulas with cross-references from other tabs or spreadsheets, with cell references farther than 10 rows or columns from the target cell in either direction, or with absolute cell ranges. Finally, our dataset includes 770K training samples, 42K for validation, and 34K for testing.

About the length distribution of target spreadsheet formulas, about 32% formulas have sketch lengths of 2, 53% formulas have sketch lengths of 3, 11% formulas have sketch lengths of 4-5, and 4% formulas have sketch lengths of at least 6. As discussed in Section 2.2, even if the formula sketches are mostly short, it is still challenging to generate the full formulas correctly. For example, the formula in Figure 2.1b is represented as SUM RANGE $ENDSKETCH$ R R[-5] C[0] SEP R[-1] C[0] $ENDR$ EOF, which has a sketch length of 2, but the full formula length is 10 if excluding the EOF token for length calculation. In total, around a hundred operators are covered in our output vocabulary, including 82 spreadsheet-specific functions, and other general-purpose numerical operators (e.g., +, -). We defer more details about dataset construction process and dataset statistics to Appendix A.3.

By default, each sample includes both the header row and surrounding data values of relative cell positions within $[-10, 10]$. Note that we do not include the data of the target cell, and we leave an empty value there. We perform the header detection according to the spreadsheet table format, i.e., we recognize the first row of a table as the header when it is frozen. Though some spreadsheet tables may include header-like descriptions in the leftmost column, e.g., "Total Score" in Figure 2.1a, we only extract headers as a row, to ensure the precision of header detection. In Section 2.4, we also discuss settings when the model input does not include headers, and when we only include a few data rows above the target cell as the input context.

Evaluation Setup

Metrics. We evaluate the following metrics: (1) *Formula accuracy*: the percentage of predicted formulas that are the same as the ground truth. (2) *Sketch accuracy*: the percentage of predictions with the same formula sketches as the ground truth. As discussed in Section 2.2, formula sketches do not include ranges, but include both functions and literals. (3) *Range accuracy*: the percentage of predictions with the same ranges as the ground truth. Note that the order of predicted ranges should also be the same as the ground truth. In addition, the

model may predict the ranges correctly even if the sketch prediction is wrong, as shown in Figure 2.5b.

Note that our formula accuracy metric could be an underestimate of the semantic equivalence, because different spreadsheet formulas may be semantically equivalent. For example, to predict arguments for `SUM` and `MULTIPLY`, different orders of the cell ranges have the same meaning. However, it is hard to systematically define the semantic equivalence in our evaluation, because we aim to support a wide range of operators in the spreadsheet language. Some existing works on program synthesis have evaluated the semantic equivalence based on the execution results [74, 35, 238]. However, it is hard to sample different input spreadsheets requiring the same formula, thus evaluating the execution accuracy is challenging. Therefore, we still focus on our current metric to measure the formula accuracy, where we compare whether the predicted formula is exactly the same as the single ground truth formula included in the spreadsheet.

Model details. For models with the BERT encoder [72], including our full SpreadsheetCoder model, we use the BERT-Medium architecture, and initialize from the English pre-trained model by default.[4] We compared our full model with several variants:

(1) Different encoder architectures. i) Using a single BERT encoder, either row-based or column-based; ii) removing convolution layers, where the BERT output is directly fed into the decoder.

(2) Different decoding approaches. We compare our two-stage decoding discussed in Section 2.3 to a simpler model that uses the same predictor for both the sketch and ranges, with a single joint output vocabulary for both.

(3) Different model initialization. When not using the pre-trained BERT model weights, we randomly initialize BERT encoders. This tests whether pre-training on generic natural language text is useful for our spreadsheet data.

We compare to previous approaches for related program synthesis tasks. First, we evaluate RobustFill, which demonstrates the state-of-the-art performance on string manipulation tasks for Excel spreadsheets [74]. Specifically, RobustFill encodes the cell context as independent rows, rather than a 2D table as in SpreadsheetCoder. Afterwards, at each decoding step, a shared LSTM decoder generates a hidden state per data row, which are then fed into a max pooling layer. Finally, the pooled hidden state is fed into a fully-connected layer to predict the formula token. We trained two variants of RobustFill on our dataset: one encodes each row independently, and another encodes each column independently, denoted as *row-based RobustFill* and *column-based RobustFill* respectively. In addition, we compared to a baseline that does not utilize any input context, thus the model only includes the LSTM decoder, similar to prior work on language modeling [239, 136].

---

[4]We downloaded the pre-trained BERT from: `https://github.com/google-research/bert`.

Results

In this section, we present the results using different variants of spreadsheet contexts as the model inputs. We perform a beam search during the inference time. Empirically, we find that results with different beam sizes (2, 4, 8, 16, 32, 64, 128) are similar, i.e., the accuracies vary within 0.3%. Therefore, we set the beam size to be 64 for all settings.

Results with the Full Input Context

Using both headers and the full surrounding data cell values as the model input, we present the formula accuracy in Table 2.1, where top-k accuracy measures how often the ground truth appears in the top k predictions using beam search. Compared to the model without the input context, all other models are able to use the contextual data to provide more accurate predictions. In particular, our full model achieves over 40% top-1 full formula prediction accuracy, which is 4 times as high as the model without context. We also observe that the full SpreadsheetCoder model has much better accuracy than either of the RobustFill models, demonstrating that our model is more capable of leveraging the implicit specification provided by the tabular context.

Different encoder architectures. Appropriately encoding the input context is important. Comparing with RobustFill models, we observe that it is beneficial to model the dependency among different rows and columns, instead of encoding each row or column independently. Meanwhile, adding convolution layers brings additional performance gain, because it enables the representation of each input token to aggregate broader contextual information beyond a few nearby rows or columns, i.e., 3 for our BERT encoders as discussed in Section 2.3. Finally, although models representing the input context as column-based tables generally perform worse than those using row-based tables, including both row-based and column-based encoders improves the overall accuracies by 2–3 percentage points. Note that the improvement is not due to the larger model size: to test this, we trained row-based and column-based BERT models with the larger BERT-base and BERT-large architectures, but the results were no better, while taking longer to train. In addition, initializing from pre-trained BERT encoders increases the formula accuracy by around 10 percentage points, suggesting that although spreadsheet headers are generally short natural language phrases, pre-training on a large-scale text corpus with much more complex text still enables the model to better understand the spreadsheet context.

Breakdown analysis of sketch and range prediction. We present the sketch and range accuracies in Table 2.2. On the one hand, sketch accuracies are generally much higher than range accuracies, since formulas are more likely to share common sketches with similar spreadsheet context, while range prediction requires a more careful investigation of the table structure. On the other hand, sketch prediction becomes more challenging when literals are included. In Figure 2.5a, we present a prediction with the correct sketch but the wrong range. Specifically, the model could easily infer that the formula should call a `SUM` function, since it is a common prediction given the input token "Total". However, the model wrongly selects all

Table 2.1: Formula accuracy on the test set. "−" means the corresponding component is removed from our full model.

	Approach	Top-1	Top-5	Top-10
	Full Model	**42.51%**	**54.41%**	**58.57%**
−	Column-based BERT	39.42%	51.68%	56.50%
−	Row-based BERT	20.37%	40.87%	48.37%
−	Convolution layers	38.43%	51.31%	55.87%
−	Two-stage decoding	41.12%	53.57%	57.95%
−	Pretraining	31.51%	42.64%	49.77%
	Row-based RobustFill	31.14%	40.09%	47.10%
	Column-based RobustFill	20.65%	39.69%	46.96%
	No context	10.56%	23.27%	31.96%

cells above as the function argument, and ignores the fact that the cell `B5` is already the sum of cells `B2`–`B4`, indicated by the text "Total price" in cell `A5`. Figure 2.5b shows a prediction with the correct range but the wrong sketch, where the predicted formula misses a "/" as an argument to the string concatenation operator "&". Two-stage decoding disentangles the generation of sketches and ranges, so that the two predictors could focus on addressing different difficulties in formula prediction, and this mechanism improves the overall accuracy.

Prediction on formulas with different sketch lengths. We present the top-1 formula accuracy on formulas with different sketch lengths in Figure 2.4. Note that we exclude the `$ENDSKETCH$` token from length calculation. First, all models achieve higher performance on formulas with sketch lengths of 2–3 than longer formulas. It is harder to make exactly the same prediction as the ground truth when the formula becomes longer, especially given that the input context is often an ambiguous specification for formula prediction. Fortunately, users typically do not need to write complicated formulas for spreadsheet data manipulation. Specifically, 85% of our collected formulas have sketch lengths of 2–3. Despite the performance degradation, our full model consistently performs better than other models on formulas with different sketch lengths.

The Effect of Header Information

In this section, we evaluate the effect of including the header row as the model input, which usually provides a short description of the table in natural language. For all models, we remove the headers from the context by replacing the header tokens with empty values. Thus the models can only use surrounding data cells as the spreadsheet context.

In Table 2.3, we observe a notable accuracy drop compared to Table 2.1, indicating that leveraging headers is critical. Figure 2.7a shows an example that can be correctly predicted by our full model, but is wrongly predicted by the model without input headers. We can observe
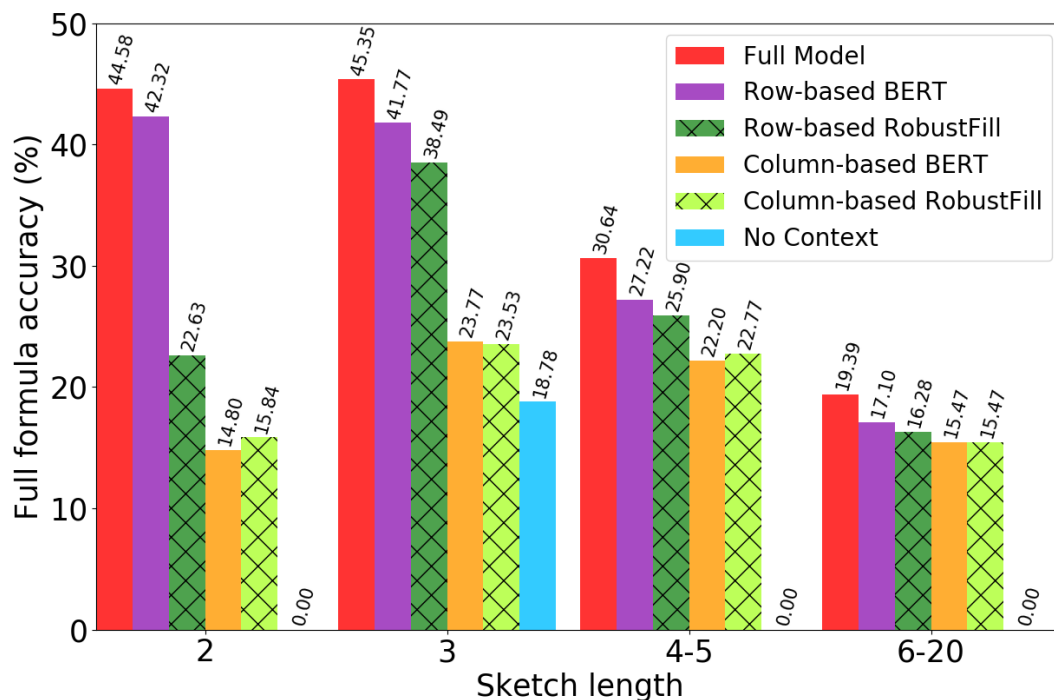
Figure 2.4: Top-1 formula accuracies for different sketch lengths.

that without the header "Average", it is much harder to figure out that the formula should call the `AVERAGE` function instead of a division. Interestingly, without input headers, using row-based or column-based table representation no longer makes much difference. However, our tabular input context encoders still perform better than RobustFill models, suggesting the importance of modeling the dependency among different rows and columns. In addition, initializing from pre-trained BERT model weights does not improve the results, and even slightly hurts the performance. The main reason is that the cell data values are mostly numeric and string literals. Breakdown results are deferred to Appendix A.2.

Results in the FlashFill-like Setting

In this section, we conduct experiments in the FlashFill-like setting, where formulas are always executed on cells in the same row. In total, 2.5K formulas in the test set only include cells with the relative row position `R[0]`, which constitute around 73% of the test set. More details are in Appendix A.4.

In Figure 2.6, we present the top-1 formula accuracies with different numbers of input data rows. We observe that even for spreadsheet formulas that only refer to cells in the same row, our models with tabular input encoders still perform better. In particular, with the increase of the number of input data rows, the accuracy of the RobustFill model does not

Table 2.2: Sketch and range accuracy on the test set.

(a) Sketch accuracy.

	Approach	Top-1	Top-5	Top-10
	Full Model	**57.41%**	**72.04%**	**78.52%**
−	Column-based BERT	55.50%	70.88%	77.73%
−	Row-based BERT	27.49%	61.95%	73.95%
−	Convolution layers	53.68%	69.38%	75.67%
−	Two-stage decoding	56.47%	72.02%	78.30%
−	Pretraining	41.26%	64.67%	76.36%
	Row-based RobustFill	40.23%	61.50%	72.20%
	Column-based RobustFill	29.50%	59.97%	71.31%
	No context	25.19%	47.08%	52.70%

(b) Range accuracy.

	Approach	Top-1	Top-5	Top-10
	Full Model	**46.93%**	**59.60%**	**63.51%**
−	Column-based BERT	43.60%	57.12%	62.27%
−	Row-based BERT	22.57%	47.84%	55.29%
−	Convolution layers	42.84%	56.64%	61.03%
−	Two-stage decoding	44.59%	58.52%	62.48%
−	Pretraining	36.03%	49.85%	54.71%
	Row-based RobustFill	33.88%	48.16%	54.83%
	Column-based RobustFill	23.97%	47.09%	52.75%
	No context	11.80%	25.54%	38.07%

show much improvement, while the accuracies of the other two models increase considerably, especially our full model. This demonstrates that our model could better utilize the available cell data context for prediction. Figure 2.7b shows a formula that can be correctly predicted by our model when the full input context is given, but is wrongly predicted when the input only contains the header row and one data row. This example shows that understanding the cell data is especially important when the header is not informative enough. Notice that including only a few input rows or columns does not fit our encoder design well, since our BERT encoders simultaneously embed 3 data rows at a time, while the RobustFill model independently encodes each row by design. This could be the main reason why models with BERT-based encoders may perform worse than RobustFill when less than 3 data rows are presented. In addition, including headers still consistently provides a significant performance gain.
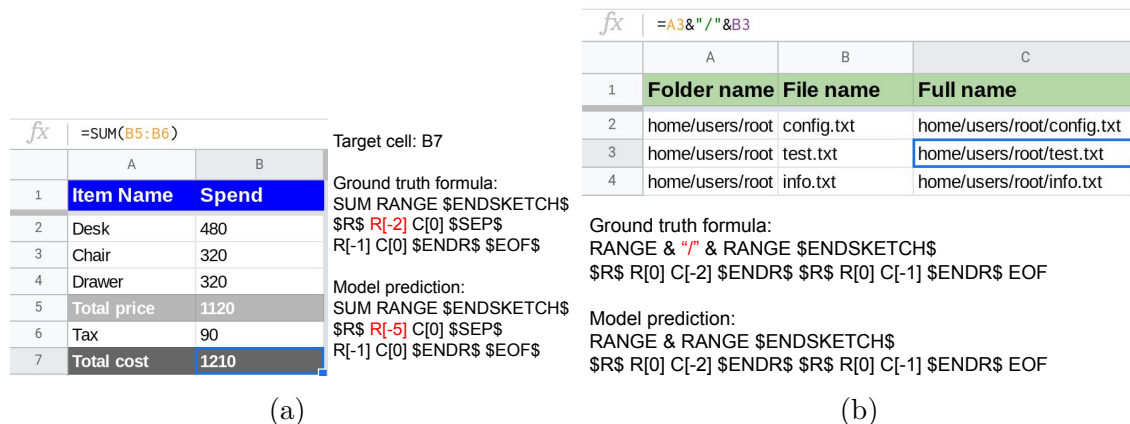
Figure 2.5: Examples of wrong formula predictions by our full model. (a) The sketch prediction is correct, but the range is wrong. (b) The range prediction is correct, but the sketch is wrong. These are synthetic examples for illustrative purposes.

Table 2.3: Formula accuracy on the test set, excluding headers in the context. Corresponding results with headers are in Table 2.1.

	Approach	Top-1	Top-5	Top-10
	Full Model	20.47%	40.23%	47.40%
−	Column-based BERT	20.63%	40.40%	**48.70%**
−	Row-based BERT	20.38%	40.11%	47.88%
−	Pretraining	**20.94%**	**40.64%**	48.51%
	Row-based RobustFill	19.02%	33.60%	37.38%
	Column-based RobustFill	17.64%	30.45%	36.79%
	No context	10.56%	23.27%	31.96%

Results on Public Excel Spreadsheets

Finally, we evaluate SpreadsheetCoder on the Enron corpus [5], which includes over 17K Excel Spreadsheets extracted from the Enron email corpus [141, 109]. We preprocess the Enron corpus in the same way as our Google Sheets corpus, and our final dataset includes 178K samples in the training set, 41K samples in the validation set, and 33K samples in the validation set. About 55% formulas have sketch lengths of 2, 18% formulas have sketch lengths of 3, 13% formulas have sketch lengths of 4-5, 9% formulas have sketch lengths of 6-7, and 5% formulas have sketch lengths of at least 8. The formulas utilize 13 spreadsheet functions, and 4 general-purpose numerical operators (i.e., +, -, *, and /). Compared to our

---

[5]The raw spreadsheet corpus is here: `https://github.com/SheetJS/enron_xls`.
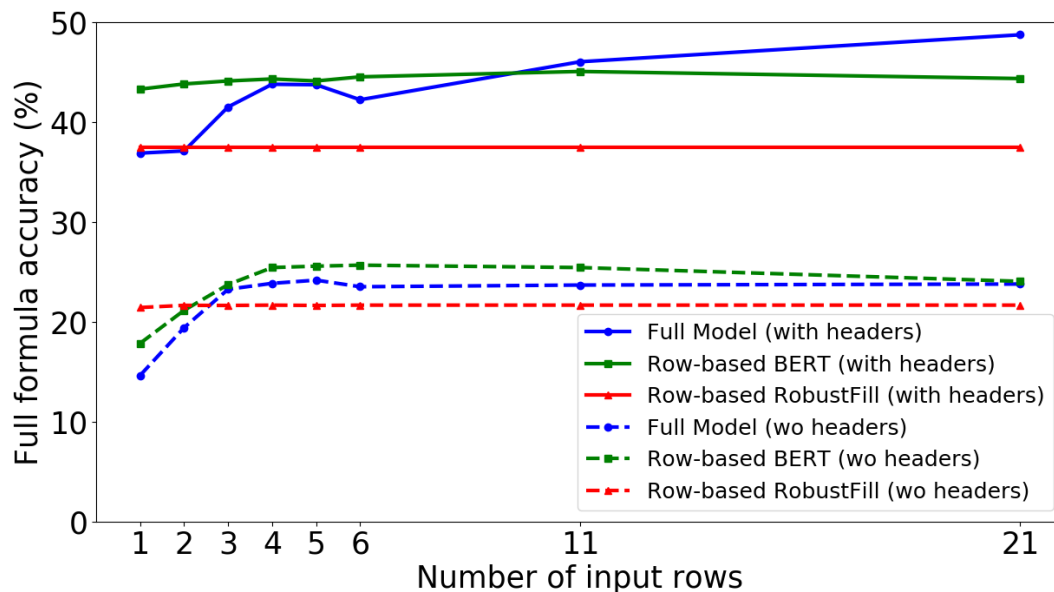
Figure 2.6: Top-1 formula accuracy in the FlashFill-like setting, with different number of input rows.

Google Sheets corpus, the Enron dataset is smaller and the formulas include fewer types of spreadsheet functions, but it contain more formulas with long sketches. More details about the dataset are deferred to Appendix A.3.

On the Enron test set, SpreadsheetCoder achieves 29.8% top-1 accuracy, 41.8% top-5 accuracy, and 48.5% top-10 accuracy. These numbers are lower than the results on our Google Sheets corpus. When investigating into the model predictions, we observe that the main reason is due to the spreadsheet format difference. Specifically, because Enron spreadsheets are in Excel, while our data preprocessing pipeline is implemented for Google Sheets, we import Enron spreadsheets into Google Sheets for data preprocessing. Therefore, a larger proportion of table headers are not properly detected. However, when comparing to the prediction results without headers, as shown in Table 2.3, the accuracies on the Enron test set are still better.

2.5 Related Work

In this section, we present a high-level overview of the related work, and we defer a more in-depth discussion to Appendix A.1. *Program synthesis* has been a long-standing challenge, and various types of specifications have been discussed, including input-output examples [99, 22, 35, 25, 227, 47], natural language descriptions [100, 283, 278, 161, 159, 257], and images [267,
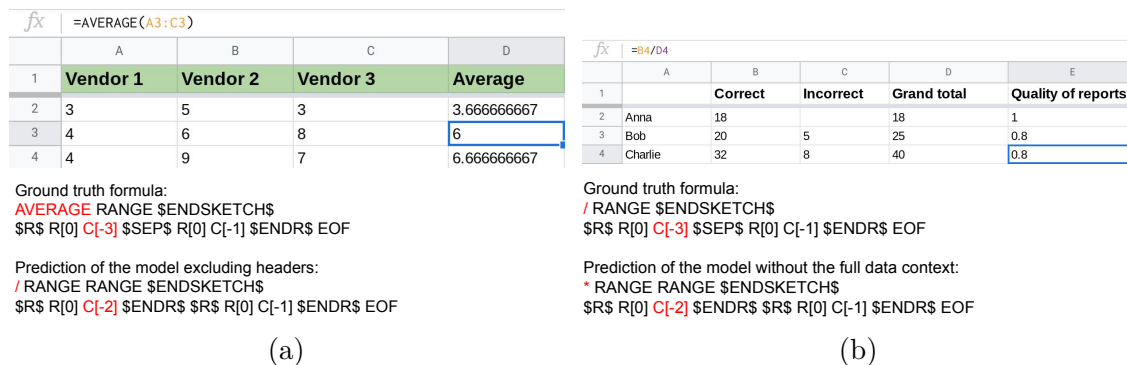
Figure 2.7: Examples of formulas that are correctly predicted by our full model with the full context, but wrongly predicted with missing context. (a) The wrong prediction when the model input does not include headers. Note that the model with headers predicts it correctly even if only one data row is provided. (b) The wrong prediction when the model input only includes headers and one data row. These are synthetic examples for illustrative purposes.

167, 238]. In particular, the FlashFill benchmark [99] is the most related to our task, and their goal is to generate string transformation programs to manipulate the Excel spreadsheet data, given input-output examples as the specification. Various neural network approaches have been proposed for FlashFill [201, 74, 252]. On the other hand, Nlyze [100] translates natural language specifications to programs in an SQL-like DSL for spreadsheet data manipulation; and Autopandas [25] synthesizes dataframe transformation functions implemented with the Python Pandas library, given input-output dataframe examples. The spreadsheet formula prediction task in our work considers the semi-structured tabular spreadsheet context as the specification, rather than standardized input-output examples or natural language descriptions. Therefore, our formula specifications are more ambiguous and diverse. Furthermore, we show that including the header information is a key factor to improving the formula prediction performance.

In terms of the model input format, our spreadsheet formula prediction task is related to existing benchmarks on *semantic parsing* over a tabular database [124, 292, 283]. There are two key differences between these tasks and ours. First, their program specification contains a natural language question, while our work predicts spreadsheet formulas based on the tabular context only. Therefore, our input specification is much more ambiguous. Meanwhile, our spreadsheet tables are typically less structured than the database tables. As shown in Figure 2.1, spreadsheet tables do not necessarily satisfy a consistent row-based schema, and data cell values may be dependent on cells from other rows.

Our tabular context encoder is related to prior works on tabular BERT models, including TAPAS [113], TaBERT [279], and Table-BERT [46]. Our encoder design differs from these works in the following ways. First, these models are designed for question answering [113, 279] or fact verification [46], thus their inputs are the concatenation of a natural language

question/statement and a table. In contrast, our model input only contains a spreadsheet table. Second, both TAPAS and Table-BERT require that the maximum table size is 512 tokens, which is not enough for our problem. SpreadsheetCoder encodes larger tabular input by tiling multiple rows/columns in multiple forward passes over BERT, and then doing the convolution to capture broader context. TaBERT independently embeds each table row with the question, then applies an attention mechanism over other tokens in the same column but different rows. This is similar to our row-based BERT without the row-wise convolution. As shown in Table 2.1, this alternative underperforms our full model.

Our spreadsheet formula prediction problem is also related to *code completion* tasks [215, 157, 242, 241, 243]. Specifically, the goal of code completion tasks is to synthesize the subsequent program tokens given the code context, while we aim to generate the formula in the cell with the missing value to complete the spreadsheet. However, instead of providing a token sequence to represent the code context, our data context is a semi-structured table, where data values in different cells are connected in a two-dimensional space.

2.6 Discussion

We presented the first technique to synthesize spreadsheet formulas given a tabular context, including both headers and cell values. In particular, we develop SpreadsheetCoder, a BERT-based model to capture the two-dimensional relational structure of the spreadsheet context, which are typically semi-structured tables. We demonstrate that incorporating the table headers significantly facilitates the prediction. Furthermore, modeling the dependency among cells of different rows and columns is important for generating formulas in real-world spreadsheets with diverse table structures. Compared to the rule-based system on Google Sheets, SpreadsheetCoder assists 82% more users in composing formulas.

There are a number of promising directions for future research about spreadsheet applications. First, developing a paradigm for pre-training on spreadsheet data could enable the encoder to be more specialized for spreadsheet applications. Second, we could infer more fine-grained knowledge of the table structure from the spreadsheet format information, such as colors and fonts, which could be utilized to develop more advanced encoder architectures. Finally, we could also extend our approach to support more spreadsheet applications, such as bug detection and clone detection.

Chapter 3

PlotCoder: Synthesizing Visualization Code in Programmatic Context

Creating effective visualization is an important part of data analytics. While there are many libraries for creating visualizations, writing such code remains difficult given the myriad of parameters that users need to provide. In this chapter, we propose the new task of synthesizing visualization programs from a combination of natural language utterances and code context. To tackle the learning problem, we introduce PlotCoder, a new hierarchical encoder-decoder architecture that models both the code context and the input utterance. We use PlotCoder to first determine the template of the visualization code, followed by predicting the data to be plotted. We use Jupyter notebooks containing visualization programs crawled from GitHub to train PlotCoder. On a comprehensive set of test samples from those notebooks, we show that PlotCoder correctly predicts the plot type of about 70% samples, and synthesizes the correct programs for 35% samples, performing 3-4.5% better than the baselines. [1]

3.1   Introduction

Visualizations play a crucial role in obtaining insights from data. While a number of libraries [120, 225, 32] have been developed for creating visualizations that range from simple scatter plots to complex 3D bar charts, writing visualization code remains a difficult task. For instance, drawing a scatter plot using the Python matplotlib library can be done using both the `scatter` and `plot` methods, and the `scatter` method [176] takes in 2 required parameters (the values to plot) along with 11 other optional parameters (marker type, color, etc), with some parameters having numeric types (e.g., the size of each marker) and some being arrays (e.g., the list of colors for each collection of the plotted data, where each color is specified as a string or another array of RGB values). Looking up each parameter's meaning

---

[1]The material in this chapter is based on Chen et al. [55]. Our code and data are available at `https://github.com/jungyhuk/plotcoder`.

and its valid values remains tedious and error-prone, and the multitude of libraries available further compounds the difficulty for developers to create effective visualizations.

In this paper, we propose to *automatically synthesize* visualization programs using a combination of natural language utterances and the programmatic context that the visualization program will reside (e.g., code written in the same file as the visualization program to load the plotted data), focusing on programs that create static visualizations (e.g., line charts, scatter plots, etc). While there has been prior work on synthesizing code from natural language [288, 195, 264, 278], and with addition information such as database schemas [292, 283, 282, 281] or input-output examples [209, 287], synthesizing general-purpose code from natural language remains highly difficult due to the ambiguity in the natural language input and complexity of the target. Our key insight in synthesizing visualization programs is to leverage their properties: they tend to be short, do not use complex programmatic control structures (typically a few lines of method calls without any control flow or loop constructs), with each method call restricted to a single plotting command (e.g., `scatter`, `pie`) along with its parameters (e.g., the plotted data). This influences our model architecture design as we will explain.

To study the visualization code synthesis problem, we use the Python Jupyter notebooks from the JuiCe dataset [3], where each notebook contains the visualization program and its programmatic context. These notebooks are crawled from GitHub and written by various programmers, thus a main challenge is understanding the complexity and the noisiness of real-world programmatic contexts and the huge variance in the quality of natural language comments. Unfortunately, using standard LSTM-based models and Transformer architectures [249] fails to solve the task, as noted in prior work [3].

We observe that while data to be plotted is usually stored in pandas dataframes [200], they are not explicitly annotated in JuiCe. Hence, unlike prior work, we augment the programmatic context with dataframe names and their schema when available in predicting the plotted data.

We next utilize our insight above and design a *hierarchical* deep neural network code generation model called PlotCoder that decomposes synthesis into two subtasks: generating the plot command, then the parameters to pass in given the command. PlotCoder uses a pointer network architecture [253], which allows the model to directly select code tokens in the previous code cells in the same notebook as the plotted data. Meanwhile, inspired by the schema linking techniques proposed for semantic parsing with structured inputs, such as text to SQL tasks [122, 257, 101], PlotCoder's encoder connects the embedding of the natural language descriptions with their corresponding code fragments in previous code cells within each notebook. Although the constructed links can be noisy because the code context is less structured than the database tables in text-to-SQL problems, we observe that our approach results in substantial performance gain.

We evaluate PlotCoder's ability to synthesize visualization programs using Jupyter notebooks of homework assignments or exam solutions. On the gold test set where the notebooks are official solutions, our best model correctly predicts the plot types for over 80% of samples, and precisely predicts both the plot types and the plotted data for over 50%

Natural Language

Explore the relationship between rarity and a skill of your choice. Choose one skill ('Attack','Defense' or 'Speed') and do the following. Use the scipy package to assess whether Catch_Rate predicts the skill. Create a scatterplot to visualize how the skill depends upon the rarity of the pokemon. Overlay a best fit line onto the scatterplot.

Local Code Context

```
slope, intercept, r_value, p_value, std_err =
    linregress(df['Catch_Rate'], df['Speed'],)
x = np.arange(256)
y = slope * x + intercept
```

Distant Dataframe Context

```
df['Weight_kg'].describe()
df['Color'].value_counts().plot(kind='bar')
df['Body_Style'].value_counts().plot(kind='bar')
grouped = df.groupby(['Body_Style','hasGender',]).mean()
df.groupby('Color')['Attack'].mean()
df.groupby('Color')['Pr_Male'].mean()
df.sort_values('Catch_Rate',ascending=False).head()
```

Dataframe Schema

```
df: ['Catch_Rate', 'Speed', 'Weight_kg', 'Color',
    'Body_Style']
```

Ground Truth

```
plt.scatter(df['Catch_Rate'], df['Speed'])
plt.plot(x,y)
```

Figure 3.1: An example of plot code synthesis problem studied in this work. Given the natural language, code context within a few code cells from the target code, and other code snippets related to dataframes, PlotCoder synthesizes the data visualization code.

of the samples. On the more noisy test splits with notebooks written by students, which may include work-in-progress code, our model still achieves over 70% plot type prediction accuracy, and around 35% accuracy for generating the entire code, showing how PlotCoder's design decisions improve our prediction accuracy.

3.2 Visualization Code Synthesis Problem

We now discuss our problem setup of synthesizing visualization code in programmatic context, where the model input includes different types of specifications. We first describe the model inputs, then introduce our code canonicalization process to make it easier to train our models and evaluate the accuracy, and finally our evaluation metrics.

Program Specification

We illustrate our program specification in Figure 3.1, which represents a Jupyter notebook fragment. Our task is to synthesize the visualization code given the natural language description and code from the preceding cells. To do so, our model takes in the following inputs:

- The natural language description for the visualization, which we extract from the natural language markdown above the target code cell containing the gold program in the notebook.

- The local code context, defined as a few code cells that immediately precede the target code cell. The number of cells to include is a tunable hyper-parameter to be described in Section 3.4.

- The code snippets related to dataframe manipulation that appear before the target code cell in the notebook, but are not included in the local code context. We refer to such code as the distant dataframe context. When such context contains code that uses dataframes, they are part of the model input by default.

As mentioned in Section 3.1, unlike JuiCe, we also extract the code snippets related to dataframes, and annotate the dataframe schemas according to their syntax trees. As shown in Section 3.1, knowing the column names in each dataframe is important for our task, as dataframes are often used for plotting.

Code Canonicalization

One way to train our models is to directly utilize the plotting code in Jupyter notebooks as the ground truth. However, due to the variety of plotting APIs and coding styles, such a model rarely predicts exactly the same code as written in Jupyter notebooks. For example, there are at least four ways in Matplotlib to create a scatter plot for columns 'y' against 'x' from a dataframe `df: plt.scatter(df['x'], df['y'])`, `plt.plot(df['x'], df['y'], 'o')`, `df.plot.scatter(x='x', y='y')`, `df.plot(kind='scatter', x='x', y='y')`. Moreover, given that the natural language description is ambiguous, many plot attributes are hard to precisely predict. For example, from the context shown in Figure 3.1, there are many valid ways to specify the plot title, the marker style, axis ranges, etc. In our experiments, we find that when trained on raw target programs, fewer than 5% predictions are exactly the same as the ground truth, and a similar phenomenon is also observed earlier [3].

Therefore, we design a canonical representation for plotting programs, which covers the core of plot generation. Specifically, we convert the plotting code into one of the following templates:

- `LIB.PLOT_TYPE(X,{Y}*)`, where `LIB` is a plotting library, and `PLOT_TYPE` is the plot type to be created. The number of arguments may vary for different `PLOT_TYPE`, e.g., 1 for histograms and pie charts, and 2 for scatter plots.

- L_0 `\n` L_1 `\n` ... L_m, where each L_i is a plotting command in the above template, and `\n` are separators.

For example, when using `plt` as the library (a commonly used abbreviation of `matplotlib.pyplot`), we convert `df.plot(kind='scatter',x='x',y='y')` into `plt.scatter(df['x'],df['y'])`,

where `LIB = plt` and `PLOT_TYPE = scatter`. Plotting code in other libraries could be converted similarly.

The tokens that represent the plotted data, i.e., `X` and `Y`, are annotated in the code context as follows:

- `VAR`, when the token is a variable name, e.g., `x` and `y` in Figure 3.1.

- `DF`, when the token is a Pandas dataframe or a Python dictionary, e.g., `df` in Figure 3.1.

- `STR`, when the token is a column name of a dataframe, or a key name of a Python dictionary, such as '`Catch_Rate`' and '`Speed`' in Section 3.1.

The above annotations are used to cover different types of data references. For example, a column in a dataframe is usually referred to as `DF[STR]`, and sometimes as `DF[VAR]` where `VAR` is a string. In Section 3.3, we will show how to utilize these annotations for hierarchical program decoding, where our decoder first generates a program sketch that predicts these token types without the plotted data, then predicts the actual plotted data subsequently.

Evaluation Metrics

Plot type accuracy.   To compute this metric, we categorize all plots into several types, and a prediction is correct when it belongs to the same type as the ground truth. In particular, we consider the following categories: (1) scatter plots (e.g., generated by `plt.scatter`); (2) histograms (e.g., generated by `plt.hist`); (3) pie charts (e.g., generated by `plt.pie`); (4) a scatterplot overlaid by a line (e.g., such as that shown in Figure 3.1, or generated by `sns.lmplot`); (5) a plot including a kernel density estimate (e.g., plots generated by `sns.distplot` or `sns.kdeplot`); and (6) others, which are mostly plots generated by `plt.plot`.

Plotted data accuracy.   This metric measures whether the predicted program selects the same data to plot as the ground truth. Unless otherwise specified, the ordering of variables must match the ground truth as well, i.e., swapping the data used to plot x and y axes result in different plots.

Program accuracy.   We consider a predicted program to be correct if both the plot type and plotted data are correct. As discussed in Section 3.2, we do not evaluate the correctness of other plot attributes because they are mostly unspecified.

3.3 PlotCoder Model Architecture

In this section, we present PlotCoder, a hierarchical model architecture for synthesizing visualization code from natural language and code context. PlotCoder includes an LSTM-based encoder [115] to jointly embed the natural language and code context, as well as a hierarchical decoder that generates API calls and selects data for plotting. We provide an overview of our model architecture in Figure 3.2.
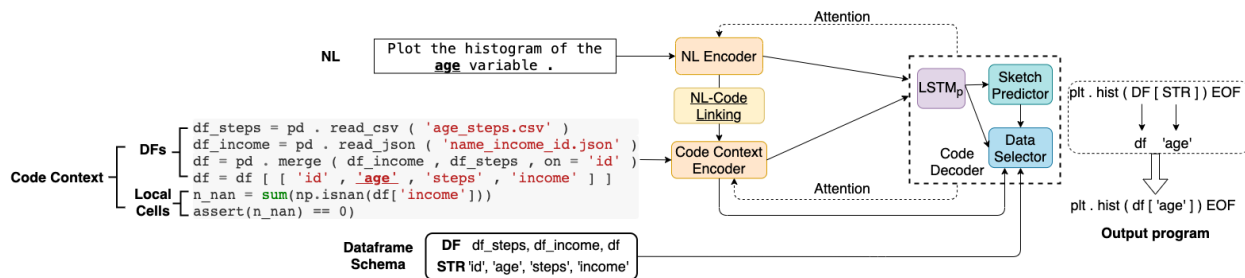
Figure 3.2: Overview of the PlotCoder architecture. The NL-Code linking component connects the embedding vectors for underscored tokens in natural language and code context, i.e., "age".

NL-Code Context Encoder

PlotCoder's encoder computes a vector representation for each token in the natural language description and the code context, where the code context is the concatenation of the code snippets describing dataframe schemas and the local code cells, as described in Section 3.2.

NL encoder. We build a vocabulary for the natural language tokens, and train an embedding matrix for it. Afterwards, we use a bi-directional LSTM to encode the input natural language sequence (denoted as LSTM_{nl}), and use the LSTM's output at each timestep as the contextual embedding vector for each token.

Code context encoder. We build a vocabulary V_c for the code context, and train an embedding matrix for it. V_c also includes the special tokens $\{\texttt{VAR}, \texttt{DF}, \texttt{STR}\}$ used for sketch decoding in Section 3.3. We train another bi-directional LSTM (LSTM_c), which computes a contextual embedding vector for each token in a similar way to the natural language encoder. We denote the hidden state of LSTM_c at the last timestep as H_c.

NL-code linking. Capturing the correspondence between the code context and natural language is crucial in achieving a good prediction performance. For example, in Figure 3.2, PlotCoder infers that the dataframe column "age" should be plotted, as this column name is mentioned in the natural language description. Inspired by this observation, we design the NL-code linking mechanism to explicitly connect the embedding vectors of code tokens and their corresponding natural language words. Specifically, for each token in the code context that also occurs in the natural language, let h_c and h_{nl} be its embedding vectors computed by LSTM_c and LSTM_{nl}, respectively, we compute a new code token embedding vector as:

$$h'_c = W_l([h_c; h_{nl}])$$

where W_l is a linear layer, and $[h_c; h_{nl}]$ is the concatenation of h_c and h_{nl}. When no natural language word matches the code token, h_{nl} is the embedding vector of the $\texttt{[EOS]}$ token at the end of the natural language description. When we include this NL-code linking component in the model, h'_c replaces the original embedding h_c for each token in the code context, and the

new embedding is used for decoding. We observe that many informative natural language descriptions explicitly state the variable names and dataframe columns for plotting, which makes our NL-code linking effective. Moreover, this component is especially useful when the variable names for plotting are unseen in the training set, thus NL-code linking provides the only cue to indicate that these variables are relevant.

Hierarchical Program Decoder

We train another LSTM to decode the visualization code sequence, denoted as LSTM_p. Our decoder generates the program in a hierarchical way. At each timestep, the model first predicts a token from the code token vocabulary that represents the program sketch. As shown in Figure 3.2, the program sketch does not include the plotted data. After that, the decoder predicts the plotted data, where it employs a copy mechanism [97, 253] to select tokens from the code context.

First, we initiate the hidden state of LSTM_p with H_c, the final hidden state of LSTM_c, and the start token is [GO] for both sketch and full program decoding. At each step t, let s_{t-1} and o_{t-1} be the sketch token and output program token generated at the previous step. Note that s_{t-1} and o_{t-1} are different only when $s_{t-1} \in \{\text{VAR}, \text{DF}, \text{STR}\}$, where o_{t-1} is the actual data name with the corresponding type. Let es_{t-1} and eo_{t-1} be the embedding vectors of s_{t-1} and o_{t-1} respectively, which are computed using the same embedding matrix for the code context encoder. The input of LSTM_p is the concatenation of the two embedding vectors, i.e., $[es_{t-1}; eo_{t-1}]$.

Attention. To compute attention vectors over the natural language description and the code context, we employ the two-step attention in [123]. Specifically, we first use hp_t to compute the attention vector over the natural language input using the standard attention mechanism [20], and we denote the attention vector as attn_t. Then, we use attn_t to compute the attention vector over the code context, denoted as attp_t.

Sketch decoding. For sketch decoding, the model computes the probability distribution among all sketch tokens in the code token vocabulary V_c:

$$Pr(s_t) = \text{Softmax}(W_s(hp_t + \text{attn}_t + \text{attp}_t))$$

Here W_s is a linear layer. For hierarchical decoding, we do not allow the model to directly decode the names of the plotted data during sketch decoding, so s_t is selected only from the valid sketch tokens, such as library names, plotting function names, and special tokens for plotted data representation in templates discussed in Section 3.2.

Data selection. For $s_t \in \{\text{VAR}, \text{DF}, \text{STR}\}$, we use the copy mechanism to select the plotted data from the code context. Specifically, our decoder includes 3 pointer networks [253] for selecting data with the type VAR, DF, and STR respectively, and they employ similar architectures but different model parameters.

Split	Train	Dev (gold)	Test (gold)	Dev (hard)	Test (hard)
All	38971	57	48	827	894
Scatter	11895	19	17	254	276
Hist	8856	14	11	182	175
Pie	574	1	1	14	13
Scatter+Plot	1533	3	1	34	57
KDE	2609	3	5	51	64
Others	13504	17	13	292	309

Table 3.1: Dataset statistics.

We take variable name selection as an instance to illustrate our data selection approach using the copy mechanism. We first compute $v_t = W_v(\text{attn}_t)$, where W_v is a linear layer. For the i-th token c_i in the code context, let hc_i be its embedding vector, we compute its prediction probability as:

$$Pr(c_i) = \frac{\exp v_t^T hc_i}{\sum_j \exp v_t^T hc_j}$$

After that, the model selects the token with the highest prediction probability as the next program token o_t, and uses the corresponding embedding vectors for s_t and o_t as the input for the next decoding step of LSTM_p.

The decoding process terminates when the model generates the `[EOF]` token.

3.4 Experiments

In this section, we first describe our dataset for visualization code synthesis, then introduce our experimental setup and discuss the results.

Dataset Construction

We build our benchmark upon the JuiCe dataset, and select those that call plotting APIs, including those from `matplotlib.pyplot` (`plt`), `pandas.DataFrame.plot`, `seaborn` (`sns`), `ggplot`, `bokeh`, `plotly`, `geoplotlib`, `pygal`. Over 99% of the samples use `plt`, `pandas.DataFrame.plot`, or `sns`. We first extract plot samples from the original dev and test splits of JuiCe to construct *Dev (gold)* and *Test (gold)*. However, the gold splits are too small to obtain quantitative results. Therefore, we extract around 1,700 Jupyter notebooks of homeworks and exams from JuiCe's training set, and split them roughly evenly into *Dev (hard)* and *Test (hard)*. All remaining plot samples from the JuiCe training split are included in our training set. The length of the visualization programs to be generated varies between 6 and 80 tokens, but the code context is typically much longer. We summarize the dataset statistics in Table 3.1.

Evaluation Setup

Implementation details. Unless otherwise specified, for the input specification we include $K = 3$ previous code cells as the local context, which usually provides the best accuracy. We set 512 as the length limit for both the natural language and the code context. For all model architectures, we train them for 50 epochs, and select the best checkpoint based on the program accuracy on the *Dev (hard)* split. More details are deferred to Appendix B.1.

Baselines. We compare the full PlotCoder against the following baselines: (1) - *Hierarchy*: the encoder is the same as in the full PlotCoder, but the decoder directly generates the full program without predicting the sketch. (2) - *Link*: the encoder does not use NL-code linking, and the decoder is not hierarchical. (3) *LSTM*: the model does not use NL-code linking, copy mechanism, and hierarchical decoding. The encoder still uses two separate LSTMs to embed the natural language and code context, which performs better than the LSTM baseline in prior work [3]. (4) *+ BERT*: we use the same hierarchical decoder as the full model, but replace the encoder with a Transformer architecture [249] initialized from a pre-trained model, and we fine-tune the encoder with other part of the model. We evaluated two pre-trained models. One is RoBERTa-base [166], an improved version of BERT-Base [72] pre-trained on a large text corpus. Another is codeBERT [89], which has the same architecture as RoBERTa-base, but is pre-trained on GitHub code in several programming languages including Python, and has demonstrated good performance on code retrieval tasks. To demonstrate the effectiveness of target code canonicalization discussed in Section 3.2, we also compare with models that are directly trained on the raw ground truth code from the same set of Jupyter notebooks.

Results

We present the program prediction accuracies in Table 3.2. First, training on the canonicalized code significantly boosts the performance for all models, suggesting that canonicalization improves data quality and hence prediction accuracies. When trained with target code canonicalization, the full PlotCoder significantly outperforms other model variants on different data splits. On the hard data splits, the hierarchical PlotCoder predicts 35% of the samples correctly, improving over the non-hierarchical model by $3 - 4.5\%$. Meanwhile, NL-code linking enables the model to better capture the correspondence between the code context and the natural language, and consistently improves the performance when trained on canonicalized target code. Without the copy mechanism, the baseline LSTM cannot predict any token outside of the code vocabulary. Therefore, this model performs worse than other LSTM-based models, especially on plotted data accuracies, as shown in Table 3.3.

Interestingly, while our hierarchical decoding, NL-code linking, and copy mechanism are mainly designed to improve the prediction accuracy of the plotted data, as shown in Table 3.4, we observe that the plot type accuracies of our full model are also mostly better, especially on the hard splits. To better understand this, we break down the results by plot type, and observe that the most significant improvement comes from the predictions

Model	Test (hard)	Dev (hard)	Test (gold)	Dev (gold)
With code canonicalization				
Full Model	**34.79%**	**34.70%**	**56.25%**	**47.37%**
− Hierarchy	30.20%	31.56%	45.83%	**47.37%**
− Link	29.98%	28.05%	43.75%	45.61%
LSTM	26.17%	24.67%	41.67%	40.35%
+ CodeBERT	33.11%	34.58%	54.17%	35.09%
+ RoBERTa	32.77%	33.37%	50.00%	26.32%
Without code canonicalization				
Full Model	20.58%	22.73%	22.92%	28.07%
− Hierarchy	20.25%	22.85%	18.75%	26.32%
− Link	20.02%	21.77%	20.83%	24.56%
LSTM	16.22%	16.93%	16.67%	24.56%
+ CodeBERT	20.92%	22.61%	22.92%	24.56%
+ RoBERTa	20.47%	22.37%	20.83%	24.56%

Table 3.2: Evaluation on program accuracy.

of scatter plots ("S") and plots in "Others" category. We posit that these two categories constitute the majority of the dataset, and the hierarchical model learns to better categorize plot types from a large number of training samples. In addition, we observe that the full model does not always perform better than other baselines on data splits of small sizes, and the difference mainly comes from the ambiguity in the natural language description. We defer more discussion to Section 3.4.

Also, using BERT-like encoders does not improve the results. This might be due to the difference in data distribution for pre-training and vocabularies. Specifically, RoBERTa is pre-trained on English passages, which does not include many visualization-related descriptions and code comments. Therefore, the subword vocabulary utilized by RoBERTa breaks down important keywords for visualization, e.g., "scatterplots" and "histograms" into multiple words, which limits model performance, especially for plot type prediction. Using codeBERT improves the performance of RoBERTa, but it still does not improve over the LSTM-based models, which may again due to vocabulary mismatch. As a result, in Table 3.4, the plot type accuracies of both models using BERT-like encoders are considerably lower than the LSTM-based models.

To better understand the plotted data prediction performance, in addition to the default plotted data accuracy that requires the data order to be the same as the ground truth, we also evaluate a relaxed version without ordering constraints. Note that the ordering includes two factors: (1) the ordering of the plotted data for the different axes; and (2) the ordering of plots when multiple plots are included. We observe that the ordering issue happens for around 1.5% of samples, and is more problematic for scatter plots ("S") and "Others." Figure 3.3 shows sample predictions where the model selects the correct set of data to plot, but the ordering is wrong. Although sometimes the natural language explicitly specifies which axes to

Model	Test (hard)	Dev (hard)	Test (gold)	Dev (gold)
With code canonicalization				
Full Model	**40.16%**	**38.69%**	**60.42%**	**49.12%**
− Hierarchy	35.91%	37.00%	47.92%	47.37%
− Link	35.46%	35.67%	47.92%	47.37%
LSTM	29.87%	28.05%	43.75%	40.35%
+ codeBERT	38.14%	38.33%	58.33%	40.35%
+ RoBERTa	37.47%	38.33%	58.33%	29.82%
Without code canonicalization				
Full Model	24.94%	27.69%	29.17%	33.33%
− Hierarchy	26.73%	27.93%	31.25%	31.58%
− Link	25.39%	27.21%	25.00%	28.07%
LSTM	18.90%	21.04%	18.75%	26.32%
+ CodeBERT	26.85%	27.21%	29.17%	31.58%
+ RoBERTa	25.28%	27.81%	27.08%	28.07%

Table 3.3: Evaluation on plotted data accuracy.

Model	Test (hard)	Dev (hard)	Test (gold)	Dev (gold)
With code canonicalization				
Full Model	**70.58%**	**71.46%**	83.33%	78.95%
− Hierarchy	64.65%	68.92%	**87.50%**	82.46%
− Link	65.32%	64.09%	81.25%	73.68%
LSTM	66.67%	67.47%	85.42%	**85.96%**
+ codeBERT	65.44%	67.96%	75.00%	57.89%
+ RoBERTa	65.21%	66.38%	66.67%	54.39%
Without code canonicalization				
Full Model	63.53%	65.66%	72.92%	80.70%
− Hierarchy	61.41%	67.47%	66.67%	73.68%
− Link	61.30%	63.72%	64.58%	77.19%
LSTM	64.65%	65.78%	81.25%	70.18%
+ CodeBERT	56.04%	57.07%	60.42%	56.14%
+ RoBERTa	61.30%	61.91%	68.75%	49.12%

Table 3.4: Evaluation on plot type accuracy.

plot (e.g., Figure 3.3 (a)), such descriptions are mostly implicit (e.g., Figure 3.3 (b)), making it hard for the model to learn. Full results on different plot types are in Section 3.4.

The Effect of Different Model Inputs

To evaluate the effect of including different input specifications, we present the results in Table 3.5. Specifically, - *NL* means the model input does not include the natural language,

(a) Natural Language
Create a scatter plot of the observations in the 'credit' dataset for the attributes 'Duration' and 'Age' (age should be shown on the xaxis).
Local Code Context
`duration = credit['Duration'].values` `age = credit['Age'].values`
Ground Truth
`plt.scatter(age, duration)`
Prediction
`plt.scatter(duration, age)`

(b) Natural Language
This graph provides more evidence that the higher a state's participation rates, the lower that state's averages scores are likely to be. The higher the participation rate, the lower the expected average verbal scores.
Local Code Context
`plt.plot(sat_data['Math'], sat_data['Verbal'])`
Dataframe Schema
`sat: ['Rate', 'Math', 'Verbal']`
Ground Truth
`plt.plot(sat_data['Rate'], sat_data['Math'])` `plt.plot(sat_data['Rate'], sat_data['Verbal'])`
Prediction
`plt.plot(sat_data['Math'], sat_data['Verbal'])` `plt.plot(sat_data['Rate'], sat_data['Verbal'])`

Figure 3.3: Examples of predictions where the model selects the correct set of data to plot, but the order is wrong.

Input	Test (hard)	Dev (hard)	Test (gold)	Dev (gold)
Full input	**34.79%**	**34.70%**	**56.25%**	**47.37%**
− Distant DFs	34.34%	34.10%	52.08%	45.61%
− NL	27.52%	28.42%	43.75%	21.05%

Table 3.5: Evaluation on the full hierarchical model with different inputs.

and - *Distant DFs* means the code context only includes the local code cells. Interestingly, even without the natural language description, PlotCoder correctly predicts a considerable number of samples. Figure 3.4 shows sample correct predictions without relying on the natural language description. To predict the plotted data, a simple yet effective heuristic is to select variable names appearing in the most recent code context. This is also one possible reason that causes the wrong data ordering prediction in Figure 3.3(a); in fact, the prediction is correct if we change the order of assignment statements for variables `age` and `duration` in the code context.

Meanwhile, we evaluated PlotCoder by varying the number of local code cells K. The

(a) Natural Language
Plot a Gaussian by looping through a range of x values and creating a resulting list of Gaussian values, g
Local Code Context
| ```python
x_axis = np.arange(-20, 20, 0.1)
g = []
for x in x_axis:
    g.append(f(mu, sigma2, x))
``` |
| **Ground Truth & Prediction** |
| ```python
plt.plot(x_axis, g)
``` |

(b) Natural Language
Like in Q9, let's start by thinking about two dice
Local Code Context
| ```python
results = []
for i in range(1,7):
    for j in range(1,7):
        print((i,j),max(i,j))
        results.append(max(i,j))
``` |
| **Ground Truth & Prediction** |
| ```python
plt.hist(results)
``` |

Figure 3.4: Examples of model predictions even without the natural language input.

results show that the program accuracies converge or start to decrease when $K > 3$ for different models, as observed in [3]. However, the accuracy drop of our hierarchical model is much less noticeable than the baselines, suggesting that our model is more resilient to the addition of irrelevant code context. See Appendix B.2 for more discussion.

Prediction Results Per Plot Type

We present the breakdown results per plot type in Tables 3.6 and 3.7. To better understand the plotted data prediction performance, in addition to the default plotted data accuracy that requires the data order to be the same as the ground truth, we also evaluate a relaxed version without ordering constraints, described as *permutation invariant* in Table 3.7. We compute the results on Test (hard), which has more samples per plot type than the gold splits. Compared to the non-hierarchical models, the most significant improvement comes from the predictions of scatter plots ("S") and plots in "Others" category. We posit that these two categories constitute the majority of the dataset, and the hierarchical model learns to better categorize plot types from a large number of training samples. The accuracy of the hierarchical model on some categories is lower than the baseline's, but the difference is not statistically significant since those categories only contain a few examples. A more detailed discussion is included in Appendix B.3.

Natural Language
Problem 5. Age groups (1 point) Create a histogram of all people's ages. Use the default settings. Add the label "Age" on the x-axis and "Count" on the y-axis.
Local Code Context
| ```
income_data.columns = ["age","workclass","fnlwgt",
"education","education_num", "marital_status",
"occupation","relationship","race","sex",
"capital_gain","capital_loss","hours_per_week",
"native_country","income_class"]
...
married_af_peoples = \\
income_data[income_data["marital_status"].str.contains(
"Married-AF-spouse")].shape[0]
...
``` |
| **Dataframe Schema** |
| ```
income_data: ['age', 'workclass', ..., 'income_class']
married_af_peoples: ['age', 'workclass', ..., 'income_class']
``` |
| **Ground Truth** |
| `plt.hist(income_data.age)` |
| **Prediction** |
| `plt.hist(married_af_peoples.age)` |

Figure 3.5: A sample prediction that requires a good understanding of the code context.

Model	S	H	Pie	S+P	KDE	Others
With code canonicalization						
Full Model	**77.17%**	70.86%	61.54%	12.28%	29.69%	**84.14%**
− Hierarchy	70.65%	68.00%	**76.92%**	15.79%	**39.06%**	71.20%
− Link	73.55%	68.00%	69.23%	**21.05%**	35.94%	70.55%
LSTM	73.91%	**71.43%**	69.23%	**21.05%**	28.13%	73.79%
+ codeBERT	67.39%	66.29%	**76.92%**	**21.05%**	35.94%	77.02%
+ RoBERTa	61.59%	62.29%	61.54%	10.53%	34.38%	80.58%
Without code canonicalization						
Full Model	71.01%	74.29%	76.92%	12.28%	37.50%	65.05%
− Hierarchy	75.00%	72.00%	61.54%	14.04%	31.25%	58.25%
− Link	72.10%	60.57%	69.23%	22.81%	37.50%	63.75%
LSTM	74.64%	74.29%	69.23%	19.30%	29.69%	65.70%
+ codeBERT	71.01%	56.00%	46.15%	14.04%	35.94%	55.02%
+ RoBERTa	73.91%	47.13%	46.15%	10.53%	29.69%	74.43%

Table 3.6: Plot type accuracy on Test (hard) per type.

Error Analysis

To better understand the challenges of our task, we conduct a qualitative error analysis and categorize the main reasons of error predictions. We investigate all error cases on *Test (gold)* split for the full hierarchical model, and present the results in Table 3.8. We summarize the key observations below, and defer more discussion to Appendix B.5.

Model	All	S	H	Pie	S+P	KDE	Others
			Plotted data accuracy				
Full Model	**40.16%**	**42.39%**	**41.14%**	61.54%	**10.53%**	**21.88%**	**45.95%**
− Hierarchy	35.91%	35.87%	40.00%	**69.23%**	8.77%	**21.88%**	40.13%
− Link	35.46%	36.96%	39.43%	53.85%	8.77%	14.06%	40.45%
LSTM	29.87%	30.43%	33.14%	61.54%	8.77%	12.50%	33.66%
+ codeBERT	38.14%	38.41%	39.43%	61.54%	8.77%	20.31%	44.98%
+ RoBERTa	37.47%	39.13%	36.57%	**69.23%**	3.51%	17.19%	45.63%
			Plotted data accuracy (permutation invariant)				
Full Model	**41.50%**	**44.57%**	**41.14%**	61.54%	**12.28%**	**21.88%**	**47.57%**
− Hierarchy	37.47%	38.04%	40.00%	**69.23%**	10.53%	**21.88%**	42.39%
− Link	41.05%	40.58%	39.43%	53.85%	8.77%	15.62%	43.04%
LSTM	30.65%	31.88%	33.14%	61.54%	10.53%	12.50%	34.30%

Table 3.7: Plotted data accuracy on Test (hard) per type. All models are trained with canonicalized target code.

- Around half of error cases are due to the ambiguity of the natural language description. (1-3)

- About 10% samples require longer code context for prediction, because the program selects the plotted data from distant code context that exceeds the input length limit. (4)

- Sometimes the model generates semantically same but syntactically different programs from the ground truth, which can happen when two variables or data frames contain the same data.(5)

- Besides understanding complex natural language description, as shown in Figure 3.3, another challenge is to understand the code context and reason about the data stored in different variables. For example, in Figure 3.5, although both dataframes `income_data` and `married_af_peoples` include the `age` column, the model must infer that `married_af_peoples` is a subset of `income_data`, thus it should select `income_data` to plot the statistics of people from all groups. (6-7)

3.5 Related Work

There has been work on translating natural language to code in different languages [288, 264, 195, 278, 292, 283, 161]. While the input specification only includes the natural language for most tasks, prior work also uses additional information for program prediction, including database schemas and contents for SQL query synthesis [292, 283, 282, 281], input-output examples [209, 287], and code context [123, 3]. There has also been work on synthesizing

Error Category	%
(1) NL only suggests the plot type	28.57
(2) NL only suggests the plotted data	9.52
(3) NL has no plotting information	9.52
(4) Need more code context	9.52
(5) Semantically correct	14.29
(6) Challenging NL understanding	19.05
(7) Challenging code context understanding	9.52

Table 3.8: Error analysis on *Test (gold)* with the hierarchical model.

data manipulation programs only from input-output examples [82, 258]. In this work, we focus on synthesizing visualization code from both natural language description and code context, and we construct our benchmark based on the Python Jupyter notebooks from the JuiCe [3]. Compared to JuiCe's input format, we also annotate dataframe schema if available, which is especially important for visualization code synthesis.

Prior work has studied generating plots from other specifications. Falx [261, 259] synthesizes plots from input-output examples, but do not use any learning technique, and focuses on developing a domain-specific language for plot generation instead. In [75], the authors apply a standard LSTM-based sequence-to-sequence model with attention for plot generation, but the model takes in only raw data to be visualized with no natural language input. The visualization code synthesis problem studied in our work is much more complex, where *both* the natural language and the code context can be long, and program specifications are implicit and ambiguous.

Our design of hierarchical program decoding is inspired by prior work on sketch learning for program synthesis, where various sketch representations have been proposed for different applications [234, 185, 79, 192]. Compared to other code synthesis tasks, a key difference is that our sketch representation distinguishes between dataframes and other variables, which is important for synthesizing visualization code.

Our code synthesis problem is also related to code completion, i.e., autocompleting the program given the code context [215, 157, 241]. However, standard code completion only requires the model to generate a few tokens following the code context, rather than entire statements. In contrast, our task requires the model to synthesize complete and executable visualization code. Furthermore, unlike standard code completion, our model synthesizes code from both the natural language description and code context. Nevertheless, when the prefix of the visualization code is given, our model could also be used for code completion, by including the given partial code as part of the code context.

3.6 Discussion

In this chapter, we present the first study of visualization code synthesis from natural language and programmatic context. Built upon the JuiCe dataset, we construct a large-scale benchmark with Python Jupyter notebooks including natural language descriptions, code context, and dataframes. We describe PlotCoder, a model architecture that includes an encoder that links the natural language description and code context, and a hierarchical program decoder that synthesizes plotted data from the code context and dataframe items. Results on real-world Jupyter notebooks show that PlotCoder can synthesize visualization code for different plot types, and outperforms various baseline models. We consider extending our approach to synthesize other parts of visualization programs (e.g., titles and legends) as future work, which could require additional specification besides the natural language to achieve good prediction results, such as input-output examples.

Part II

Synthesis from Input-Output Examples

Chapter 4

Execution-Guided Neural Program Synthesis

Neural program synthesis from input-output examples has attracted an increasing interest from both the machine learning and the programming language community. Most existing neural program synthesis approaches employ an encoder-decoder architecture, which uses an encoder to compute the embedding of the given input-output examples, as well as a decoder to generate the program from the embedding following a given syntax. Although such approaches achieve a reasonable performance on simple tasks such as FlashFill, on more complex tasks such as Karel, the state-of-the-art approach can only achieve an accuracy of around 77%. We observe that the main drawback of existing approaches is that the semantic information is greatly under-utilized. In this work, we propose two simple yet principled techniques to better leverage the semantic information, which are *execution-guided synthesis* and *synthesizer ensemble*. These techniques are general enough to be combined with any existing encoder-decoder-style neural program synthesizer. Applying our techniques to the Karel dataset, we can boost the accuracy from around 77% to more than 90% [1].

4.1 Introduction

Program synthesis is a traditional challenging problem. Such a problem typically takes a *specification* as the input, and the goal is to generate a *program* within a target *domain-specific language* (DSL). One of the most interesting forms of the specifications is input-output examples, and there have been several applications, such as FlashFill [98, 99].

Recently, there is an increasing interest of applying neural network approaches to tackle the program synthesis problem. For example, Devlin et al. have demonstrated that using an encoder-decoder-style neural network, their neural program synthesis algorithm called RobustFill can outperform the performance of the traditional non-neural program synthesis approach by a large margin on the FlashFill task [74].

---

[1]The material in this chapter is based on Chen et al. [47].

Despite their promising performance, we identify several inefficiencies of such encoder-decoder-style neural program synthesis approaches. In particular, such a neural network considers program synthesis as a sequence generation problem; although some recent work take the syntactical information into consideration during program generation [35, 212, 277, 201, 271], the semantic information, which is typically well-defined in the target DSL, is not effectively leveraged by existing work.

In light of this observation, in this work, we develop simple yet principled techniques that can be combined with any existing encoder-decoder-style neural program synthesizers. The main novel technique is called *execution-guided synthesis*. The basic idea is to view the program execution as a sequence of manipulations to transform each input state into the corresponding output state. In such a view, executing a *partial program* can result in intermediate states; thus, synthesizing the rest of the program can be conditioned on these intermediate states, so that the synthesizer can take the state changes into account in the followup program generation process. Therefore, we can leverage this idea to combine with any existing encoder-decoder-style neural synthesizer, and we observe that it can significantly improve the performance of the underlying synthesizers.

In addition, we also propose a simple yet effective technique called *synthesizer ensemble*, which leverages the semantic information to ensemble multiple neural program synthesizers. In particular, for the input-output program synthesis problem, we can easily verify if a synthesized program satisfies the input-output specification, which allows us to remove invalid predictions from the ensemble during inference time. Albeit its simplicity, to the best of our knowledge, we are not aware of any previous neural program synthesis work applying this approach. We observe that this technique further boosts the performance substantially.

We evaluate our techniques on the Karel task [35, 73], the largest publicly available benchmark for input-output program synthesis, on which the most performant model in the past can achieve only an accuracy of around 77% [35]. We observe that our proposed techniques can gain better performance than the previous state-of-the-art results. In particular, by combining both of our techniques, we can achieve an accuracy of 92%, which is around 15 percentage points better than the state-of-the-art results. This shows that our approach is effective in boosting the performance of algorithms for neural program synthesis from input-output examples.

4.2 Problem Setup

In this section, we first introduce the input-output program synthesis problem and existing encoder-decoder-style neural program synthesis approaches, then present an overview of our approaches.

Problem Definition

We follow the literature [74, 35, 48] to formally define the input-output program synthesis problem below.

Problem Definition 1 (Program emulation). *Let $\mathcal{L}$ be the space of all valid programs in the domain-specific language (DSL). Given a set of input-output pairs $\{(I^k, O^k)\}_{k=1}^{K}$ (or $\{IO^K\}$ in short), where there exists a program $P \in \mathcal{L}$, such that $P(I^k) = O^k, \forall k \in \{1, ..., K\}$. Our goal is to compute the output O^{test} for a new test input I^{test}, so that $O^{\text{test}} = P(I^{\text{test}})$.*

Although the problem definition only requires to compute the output for a test input, a typical method is to synthesize a program $P' \in \mathcal{L}$ such that P' is consistent with all input-output pairs $\{IO^K\}$, and then use P' to compute the output. In this case, we say program P' *emulates* the program P corresponding to $\{IO^K\}$.

In particular, in this work, we are mainly interested in the following formulation of the problem.

Problem Definition 2 (Program synthesis). *Let $\mathcal{L}$ be the space of all valid programs in the domain-specific language (DSL). Given a training dataset of $\{IO^K\}_i$ for $i = 1, ..., N$, where N is the size of the training data, compute a synthesizer Γ, so that given a test input-output example set $\{IO^K\}_{\text{test}}$, the synthesizer $\Gamma(\{IO^K\}_{\text{test}}) = P$ produces a program P, which emulates the program corresponding to $\{IO^K\}_{\text{test}}$.*

Encoder-decoder-style Neural Program Synthesis Approaches

There have been many approaches proposed for different neural program synthesis tasks, and most of them follow an encoder-decoder-style neural network architecture [35, 74, 201]. Figure 4.1 shows a general neural network architecture for input-output program synthesis. First, an encoder converts input-output examples $\{IO^K\}$ into an embedding. For example, RobustFill [74] deals with the string transformation tasks, thus it uses LSTMs as the encoder. For the Karel task, both inputs and outputs are 2D grids (see Figure 4.2); therefore, [35] employ a CNN as the encoder.

Once the IO embeddings are computed, these approaches employ an LSTM decoder to generate the programs conditioned on the embeddings. For program synthesis, one unique property is that the generated program should satisfy the synthax of $\mathcal{L}$. Therefore, a commonly used refinement is to filter syntactically invalid program prefixes during generation [201, 74, 35].

In the above approaches, only the syntax information is leveraged; the semantics of $\mathcal{L}$ is not utilized. In particular, standard supervised training procedure could suffer from *program aliasing*: for the same input-output examples, there are multiple semantically equivalent programs, but all except the one provided in the training data will be penalized as wrong programs. To mitigate this issue, Bunel et al. propose to train with reinforcement learning, so that it rewards all semantically correct programs once they are fully generated [35]. In our

Figure 4.1: A neural network architecture for input-output program synthesis (e.g., [35]). At each timestep t, the decoder LSTM generates a program token g_t conditioned on both the input-output pairs $\{IO^K\}$ and the previous program token g_{t-1}. Each IO pair is fed into the LSTM individually, and a max-pooling operation is performed over the hidden states $\{h_t^k\}_{k=1}^K$ of the last layer of LSTM for all IO pairs. The resulted vector is fed into a softmax layer to obtain a prediction probability distribution over all the possible program tokens in the vocabulary. More details can be found in Appendix C.3.

work, we demonstrate that we can leverage the semantic information in an effective way that provides a better performance.

An Overview of our Approaches

In this work, we propose two general and principled techniques that can improve the performance over existing work, which are *execution-guided synthesis* (Section 4.3) and *synthesizer ensemble* (Section 4.4). The main idea of our techniques is to better leverage the semantics of the language $\mathcal{L}$ during synthesis. Meanwhile, our techniques are compatible with any existing encoder-decoder-style neural program synthesis architecture. We will describe these techniques in detail in the following sections.

4.3   Execution-Guided Synthesis

Existing approaches generate the entire program only based on the input-output examples before execution. However, this is an inefficient use of the semantics of $\mathcal{L}$. For example, when a program consists of a sequence of statements, we can view the output to be a result by continuously executing each statement in the sequence to convert the input state into a sequence of intermediate states. Figure 4.2 illustrates such an example. From this perspective,

Program: turnLeft() ; move() ; putMarker()

States:

Figure 4.2: An example of the execution of partial programs to reach the target state in the Karel domain. The blue dot denotes the marker put by the Karel robot.

$$
\begin{aligned}
P &:= B; \bot \\
B &:= \bot \mid \mathcal{S} \mid B; B \\
&\quad \mid \quad \textbf{if } \mathcal{C} \textbf{ then } B \textbf{ else } B \textbf{ fi} \\
&\quad \mid \quad \textbf{while } \mathcal{C} \textbf{ do } B \textbf{ end} \\
\mathcal{S}, \mathcal{C} &\in \mathcal{L}
\end{aligned}
$$

Table 4.1: Syntax of $\mathcal{L}_{\text{ext}}$.

instead of generating the whole program at once, we can generate one statement at a time based on the intermediate/output state pairs.

However, most interesting programs are not just sequential. In this work, we explore this idea using a general control-flow framework. In particular, given any language $\mathcal{L}$, we extend it with three classical types of control-flow: sequential, branching, and looping. The extended language is call $\mathcal{L}_{\text{ext}}$. Then, we develop our above idea based on $\mathcal{L}_{\text{ext}}$, called *execution-guided synthesis*. In the following, to make our discussion concise, we first formalize $\mathcal{L}_{\text{ext}}$ (Section 4.3), and then present the idea of execution-guided synthesis (Section 4.3).

The Formal Specification of the Extended Language $\mathcal{L}_{\text{ext}}$

In this work, we assume some additional control-flow syntax on top of $\mathcal{L}$. We define the extended language $\mathcal{L}_{\text{ext}}$ in Table 4.1. In particular, we assume that a code block B can be composed by a sequence of statements $\mathcal{S} \in \mathcal{L}$ or sub code blocks, and each code block can also be an if-statement or a while-statement. We use $\mathcal{C}$ to indicate a condition expression in $\mathcal{L}$, and $\bot$ to indicate the termination of a program execution.

The semantics of $\mathcal{L}_{\text{ext}}$ is specified in Figure 4.3. These rules are largely standard following the convention in programming language literature. In particular, each rule's name starts with S- indicating that they are semantics rules; the suffixes indicate the constructors each rule specifies, (e.g., Stmt for statements, Seq for sequences, etc.). The judgment $\langle B, s \rangle \to \langle B', s' \rangle$ indicates a small-step execution of program B over state s to result in a new program B' and a new state s'. The judgments $\langle \mathcal{S}, s \rangle \Downarrow s'$ and $\langle C, s \rangle \Downarrow b$ capture the big-step execution in $\mathcal{L}$ that statement $\mathcal{S}$ evaluates to a new state s' from s, and condition $\mathcal{C}$ evaluates to a boolean value b from s. Following the semantics of $\mathcal{L}_{\text{ext}}$, we can formally define a program execution.

$$\text{S-Stmt} \frac{\langle \mathcal{S}, s \rangle \Downarrow s'}{\langle \mathcal{S}, s \rangle \rightarrow \langle \bot, s' \rangle} \qquad \text{S-Seq} \frac{\langle B_1, s \rangle \rightarrow \langle B_1', s' \rangle}{\langle B_1; B_2, s \rangle \rightarrow \langle B_1'; B_2, s' \rangle}$$

$$\text{S-Seq-Bot } \langle \bot; B_2, s \rangle \rightarrow \langle B_2, s \rangle \qquad \text{S-If} \frac{\langle \mathcal{C}, s \rangle \Downarrow b \quad i = \begin{cases} 1 & b \text{ is true} \\ 2 & b \text{ is false} \end{cases}}{\langle \textbf{if } \mathcal{C} \textbf{ then } B_1 \textbf{ else } B_2 \textbf{ fi}, s \rangle \rightarrow \langle B_i, s \rangle}$$

$$\text{S-While } \langle \textbf{while } \mathcal{C} \textbf{ do } B \textbf{ end}, s \rangle \rightarrow \langle \textbf{if } \mathcal{C} \textbf{ then } B; \textbf{while } C \textbf{ do } B \textbf{ end else } \bot \textbf{ fi}, s \rangle$$

Figure 4.3: Semantic rules $\langle B, s \rangle \rightarrow \langle B', s' \rangle$ for $\mathcal{L}_{\text{ext}}$.

Definition 1 (Program execution). *Given a program $P \in \mathcal{L}_{\text{ext}}$ and an input I, the execution is a sequence $s_0...s_T$, such that (1) $s_0 = I$; (2) $B_0 = P$; (3) $\langle B_i, s_i \rangle \rightarrow \langle B_{i+1}, s_{i+1} \rangle$ for $i = 0, ..., T-1$; and (4) $B_T = \bot$. The output of the program is $O = s_T$.*

Execution-Guided Synthesis Algorithm

In Definition 1, we can observe that the initial and final states are simply two special states provided as the input-output examples of the synthesis problem. Thus, a synthesizer Γ for input-output pairs should also be able to take any state-pairs as inputs. Our execution-guided synthesis algorithm takes advantage of this fact to improve the performance of the synthesizer. In the following, we discuss three cases from the easiest to the hardest to present our approach.

Sequential programs. We now consider the simplest case, where the program is in the form of $\mathcal{S}_1; ...; \mathcal{S}_T$, to illustrate the basic idea of execution-guided synthesis algorithm. We present the algorithm in Algorithm 1. Assuming the input-output examples are $\{IO^K\}$, we can treat them as K state-pairs $\{(s_{\textbf{i}}^k, s_{\textbf{o}}^k)\}_{k=1}^K$, where $s_{\textbf{i}}^k = I^k, s_{\textbf{o}}^k = O^k$. The Exec algorithm takes the synthesizer Γ and the input-output pairs $\{IO^K\}$ as its input. It also takes an additional input Δ, which is the *ending token* to be synthesized. For the top-level program, Δ will be the $\bot$ token. Later we will see that when synthesizing the sub-program for If- and While-blocks, different ending tokens will be used.

The synthesized program is initially empty (line 3). Then the algorithm iteratively generates one statement $\mathcal{S}$ at a time (line 4-10 and 17), and appends it to the end of P (line 16). Importantly, if $\mathcal{S}$ is not an if-statement or a while-statement, for which we handle separately, the algorithm executes the newly generated statement $\mathcal{S}$ to transit $s_{\textbf{i}}^k$ into $s_{\textbf{new}}^k$ (line 11-14). Therefore, in the subsequent iteration, the synthesizer can start from the new states $s_{\textbf{new}}^k$ after executing the partial program P generated so far. In doing so, the synthesizer

---

Algorithm 1 Execution-guided synthesis (sequential case)

1: **function** EXEC(Γ, $\{(s_{\mathbf{i}}^k, s_{\mathbf{o}}^k)\}_{k=1}^K$, Δ)
2:     // The main algorithm is called using Exec (Γ, $\{IO^K\}$, $\bot$)
3:     $P \leftarrow \bot$
4:     $\mathcal{S} \leftarrow \Gamma(\{(s_{\mathbf{i}}^k, s_{\mathbf{o}}^k)\}_{k=1}^K)$
5:     **while** $\mathcal{S} \neq \Delta$ **do**
6:         **if** $\mathcal{S} = \mathbf{if}$-token **then**   // If-statement synthesis
7:             $\mathcal{S}, \{(s_{\mathbf{i}}^k, s_{\mathbf{o}}^k)\}_{k=1}^K \leftarrow$ ExecIf(Γ, $\{(s_{\mathbf{i}}^k, s_{\mathbf{o}}^k)\}_{k=1}^K$)
8:         **else**
9:             **if** $\mathcal{S} = \mathbf{while}$-token **then**   // While-statement synthesis
10:                 $\mathcal{S}, \{(s_{\mathbf{i}}^k, s_{\mathbf{o}}^k)\}_{k=1}^K \leftarrow$ ExecWhile(Γ, $\{(s_{\mathbf{i}}^k, s_{\mathbf{o}}^k)\}_{k=1}^K$)
11:             **else**   // Execution of $\mathcal{S}$
12:                 $\langle \mathcal{S}, s_{\mathbf{i}}^k \rangle \rightarrow \langle \bot, s_{\mathbf{new}}^k \rangle$ for $k = 1, ..., K$
13:                 $s_{\mathbf{i}}^k \leftarrow s_{\mathbf{new}}^k$ for $k = 1, ..., K$
14:             **end if**
15:         **end if**
16:         $P \leftarrow P; \mathcal{S}$
17:         $\mathcal{S} \leftarrow \Gamma(\{(s_{\mathbf{i}}^k, s_{\mathbf{o}}^k)\}_{k=1}^K)$
18:     **end while**
19:     **return** P
20: **end function**

---

can see all intermediate states to better adjust the followup synthesis strategies to improve the overall synthesis performance.

Branching programs.   Dealing with if-statements is slightly more complicated than sequential programs, since in an if-statement, not all statements will be executed on all inputs. Following the execution in Algorithm 1 naively, we have to use Γ to synthesize the entire if-statement before being able to execute the partially generated program to derive intermediate states.

Therefore, in Algorithm 2, we extend the above idea to handle if-statements. When the next predicted token is an **if**-token, our execution-guided synthesizer first predicts the condition of the if-statement $\mathcal{C}$ (line 2). Then, we evaluate $\mathcal{C}$ over all state-pairs. Based on the evaluation results, we can divide the IO pairs into two sets $\mathcal{I}_{\mathbf{t}}$ and $\mathcal{I}_{\mathbf{f}}$ (line 3-4), so that all states in the former meet the branching condition to go to the true branch, and all states in the latter go to the false branch. Therefore, in the followup synthesis, we do not need to consider $\mathcal{I}_{\mathbf{f}}$ (or $\mathcal{I}_{\mathbf{t}}$) when synthesizing the true branch (or the false branch) (line 5-6). Note that in line 5-6, synthesizing both true-branch and false-branch employ execution-guided synthesis algorithm to leverage intermediate states, and different ending tokens (i.e., **else** and **fi**) are supplied respectively. Once we have done the synthesis of both branches, we

---

Algorithm 2 Execution-guided synthesis (if-statement)

1: **function** EXECIF(Γ, $\mathcal{I}$)
2:     $\mathcal{C} \leftarrow \Gamma(\mathcal{I})$
3:     $\mathcal{I}_t \leftarrow \{(s_\mathbf{i}, s_\mathbf{o}) \in \mathcal{I} | \langle \mathcal{C}, s_\mathbf{i} \rangle \Downarrow \text{true}\}$
4:     $\mathcal{I}_f \leftarrow \{(s_\mathbf{i}, s_\mathbf{o}) \in \mathcal{I} | \langle \mathcal{C}, s_\mathbf{i} \rangle \Downarrow \text{false}\}$
5:     $B_t \leftarrow \text{Exec}(\Gamma, \mathcal{I}_t, \textbf{else}\text{-token})$
6:     $B_f \leftarrow \text{Exec}(\Gamma, \mathcal{I}_f, \textbf{fi}\text{-token})$
7:     $\mathcal{I}'_t \leftarrow \{(s_\mathbf{new}, s_\mathbf{o}) | (s_\mathbf{i}, s_\mathbf{o}) \in \mathcal{I}_t \wedge \langle B_t, s_\mathbf{i} \rangle \Downarrow s_\mathbf{new}\}$
8:     $\mathcal{I}'_f \leftarrow \{(s_\mathbf{new}, s_\mathbf{o}) | (s_\mathbf{i}, s_\mathbf{o}) \in \mathcal{I}_f \wedge \langle B_f, s_\mathbf{i} \rangle \Downarrow s_\mathbf{new}\}$
9:     $\mathcal{I} \leftarrow \mathcal{I}'_t \cup \mathcal{I}'_f$
10:     $\mathcal{S} \leftarrow \textbf{if } C \textbf{ then } B_t \textbf{ else } B_f \textbf{ fi}$
11:     **return** $\mathcal{S}, \mathcal{I}$
12: **end function**

---

can execute the generated branches to get the new states $\mathcal{I}$ (line 7-9), and return the newly generated if-statement and updated states to the caller of this algorithm.

In Algorithm 2, we use $\langle B, s \rangle \Downarrow s'$ to indicate a big-step execution of code block B over state s to get s'. In particular, this means that $\langle B, s \rangle \rightarrow \langle B_1, s_1 \rangle \rightarrow ... \rightarrow \langle \bot, s' \rangle$.

Looping programs. The remaining problem is to handle while-statements. Due to the rule S-While (see Figure 4.3), a while-statement

$$\textbf{while } \mathcal{C} \textbf{ do } B \textbf{ end} \tag{4.1}$$

is equivalent to

$$\textbf{if } \mathcal{C} \textbf{ then } (B; \textbf{while } \mathcal{C} \textbf{ do } B \textbf{ end}) \textbf{ else } \bot \textbf{ fi} \tag{4.2}$$

Therefore, we can employ a procedure similar to Algorithm 2 once a **while**-token is predicted. However, there are two differences. First, in (4.2), the false-branch is empty, thus we do not need to deal with the false-branch. Second, although the true-branch is $B; \textbf{while } \mathcal{C} \textbf{ do } B \textbf{ end}$, once we have generated B, we do not need to generate the rest of the true-branch, since both $\mathcal{C}$ and B have been generated. The detailed algorithm can be found in Appendix C.2.

Remarks. The final algorithm is called by Exec (Γ, $\{IO^K\}$, $\bot$). Note that our execution-guided synthesis algorithm can be applied to any neural synthesizer Γ, and we can train the synthesizer Γ using any supervised or reinforcement learning algorithms that have been proposed before [74, 35]. In our evaluation, we demonstrate that our execution-guided synthesis technique helps boost the performance of both supervised and reinforcement learning algorithms proposed in existing work [35].

4.4   Synthesizer Ensemble

In our experiments, we observe that when we use different random initializations of the synthesizer during training, even if the synthesizer architectures are the same, they will be effective on different subsets of the dataset, although the overall prediction accuracy is similar to each other. Thus, a natural idea is to train multiple synthesizers, and ensemble them to build a more performant synthesizer.

Different from other deep learning tasks, for program synthesis task, without knowing the ground truth, we can already filter out those wrong predictions that cannot satisfy the input-output specification. Thus, we ensemble multiple synthesizers as follows: we run all synthesizers to obtain multiple programs, and select from programs that are consistent with all input-output examples. This provides us with a better chance to select the correct final prediction that generalizes to held-out IO pairs.

The main subtlety of such an approach is to deal with the case when multiple generated programs satisfy the input-output examples. In this work, we consider several alternatives as follows:

- **Majority vote.** We can choose the most frequently predicted program as the final prediction.

- **Shortest.** Following the Occam's razor principle, we can choose the shortest program as the final prediction.

4.5   Evaluation

In this section, we demonstrate the effectiveness of our approaches on the Karel dataset [204, 35]. We first introduce the task, discuss the experimental details, and present the results.

The Karel Task

Karel is an educational programming language proposed in the 1980s [204]. Using this language, we can control a robot to move inside a 2D grid world and modify the world state, and our goal is to synthesize a program given a small number of input and output grids as the specification. Such tasks have been used in Stanford CS introductory courses [65] and the Hour of Code [114], and have been studied recently in several neural program synthesis works [73, 35, 228]. Figure 4.2 shows an example in the Karel domain. We provide the grammar specification and the state representation in Appendix C.1. In particular, the Karel DSL includes control flow constructs such as conditionals and loops, which is more challenging than problems well-studied before, such as FlashFill [98, 74].

Our evaluation follows the setup in [35]. We train and evaluate our approaches on their dataset, which is built by randomly sampling programs from the DSL. For each program, 5 IO pairs serve as the specification, and the sixth one is the held-out test sample. In total,

there are 1,116,854 programs for training, 2,500 in the validation set, and 2,500 in the test set. We evaluate the following two metrics, which are the same as in [35]:

- **Exact Match.** The predicted program is an *exact match* if it is the same as the ground truth.

- **Generalization.** The predicted program is a *generalization* if it satisfies the input-output examples in both the specification and the held-out examples.

Training dataset construction for the Exec algorithm. Note that the original Karel dataset only provides the input-output examples and the ground truth programs. To train the synthesizer Γ with our Exec algorithm in the supervised learning setting, we need the supervision on intermediate states as well, which can be obtained easily by executing the ground truth programs following the semantics (Figure 4.3). In particular, for each sample $\langle \{IO^K\}, P \rangle$ in the original training set, we construct a sequence of training samples $\langle \{(s_{i-1}^k, O^k)\}_{k=1}^{K_i}, S_i \rangle$ $(i = 1, 2, ..., T)$, with each sample containing $K_i \leq K$ state pairs and a program $S_i \in P$. The algorithm to construct this set is largely analogous to the semantics specification, and we defer the details to Appendix C.2.

Training algorithms. Once the training set is constructed, the neural synthesizer Γ can be trained on this new dataset using the same algorithm as the one for training Γ on the original dataset. Therefore, our Exec algorithm can be applied to both supervised learning and reinforcement learning algorithms proposed in [35] for evaluation.

Model details. We employ the same neural network architecture as in [35] to synthesize the programs, which is briefly discussed in Section 4.2. During the inference time, we set the beam size $B = 64$, and select the one with the highest prediction probability from the remaining programs. More details can be found in Appendix C.3.

Results

We present our main results in Table 4.2. We report the results of ensembling 15 models for our ensemble techniques. For reference, we include MLE and RL results in [35], which were the state-of-the-art on the Karel task for the exact match and generalization metrics respectively. We first apply our ensemble techniques to these approaches, and observe that the performance could be significantly boosted by around 7%.

We next observe that our execution-guided synthesis alone can significantly improve the generalization accuracy over all approaches from [35], even after we accompany their approaches with our ensemble techniques. In particular, without the ensemble, "Exec+SL" already improves "MLE+RL" by 8 points on generalization accuracy; and when ensemble approaches are applied to "MLE+RL", this gap is shrunk, but still positive. Similar to [35], we can also train our Exec model using the RL technique, which improves the generalization

Training		Ensemble	Generalization	Exact Match	From
MLE	SL	-	71.91%	39.94%	[35]
		S	78.80%	46.68%	This work
		MV	78.80%	**47.08%**	This work
	RL	-	77.12%	32.17%	[35]
		S	84.84%	46.04%	This work
		MV	84.16%	46.36%	This work
Exec	SL	-	85.08%	40.88%	This work
		S	91.60%	45.84%	
		MV	91.52%	45.36%	
	RL	-	86.04%	39.40%	
		S	91.68%	46.36%	
		MV	**92.00%**	45.64%	

Table 4.2: Accuracy on the Karel test set. In the "Training"column, we use "MLE" and "Exec" to indicate the training approaches proposed in [35] and this work, "SL" and "RL" to indicate supervised learning and reinforcement learning respectively. In the "Ensemble" column, dash indicates that no ensemble is used, "S" and "MV" indicate the shortest and majority vote principles respectively. For the single model accuracy, we report the results of the model with the best generalization accuracy. We include 15 models in each ensemble.

accuracy by another 1 point, while slightly decreases the exact match accuracy. These results show that utilizing intermediate execution states alone is already an effective approach to boost the performance.

Note that the improvement of Exec on the exact match accuracy is relatively minor, and sometimes negative when applying the ensemble to baseline training algorithms. This is because our Exec algorithm is not designed to optimize for exact match accuracy. In particular, we decouple the full programs in the original training dataset into small pieces, thus our synthesizer Γ is trained with segments of the original training programs instead of the full programs. In doing so, our synthesizer is more capable of generating programs piece-by-piece and thus tends to generate semantically correct programs (i.e., with a better generalization accuracy) rather than the same programs in the training and testing sets (i.e., with a better exact match accuracy). In fact, for real-world applications, the generalization accuracy is more important than the exact match accuracy, because exact match accuracy is more about evaluating how well the synthesizer recovers the language model of the pCFG sampler used to generate the dataset. More discussion can be found in Appendix C.4.

Finally, we apply our ensemble approaches on top of Exec+SL and Exec+RL. We observe that this can further improve the generalization accuracy and exact match accuracy by around 6% on top of the best single model. These results show that our ensemble approaches consistently boost the performance, regardless of the underlying models used for ensembling.

In addition, we investigate the performance of ensembling different number of models.

Figure 4.4: Results of the ensemble model trained with Exec + RL approach. Left: generalization accuracy. Right: exact match accuracy. The corresponding figures using models trained with Exec approach can be found in Appendix C.4.

We present the results of ensembling Exec + RL models in Figure 4.4, and defer the results of ensembling Exec models to Appendix C.4. We observe that using the shortest principle is generally more effective than using the majority vote principle, especially when fewer number of models are included in the ensemble. However, when there are more models, majority vote may achieve a better generalization accuracy than the shortest principle. This is reasonable, since when there are too few models, there might not be enough effective models to form the majority.

Interestingly, we observe that Exec+RL+Ensemble does not significantly improve the performance over Exec+SL+Ensemble. This may be due to that the improvement from ensemble hides the improvement from RL. More discussion of our evaluation results can be found in Appendix C.4.

To summarize, we make the following key observations:

1. Our execution-guided synthesis technique can effectively improve previous approaches, which only use the syntactic information, or the final program execution outputs.

2. Our ensemble approaches can effectively improve the performance regardless of the underlying models being used.

3. The different modules of our proposed approaches, i.e., execution-guided synthesis and ensemble techniques, can work independently to improve the performance, and thus they can be applied independently to other tasks as well.

4. By combining all our novel techniques, we improve the state-of-the-art on the Karel task by 14.88 points (generalization) and 7.14 points (exact match).

4.6 Related Work

Synthesizing a program from input-output examples is an important challenging problem with many applications [74, 99, 98, 35, 48, 38, 156, 217, 286, 285, 90, 268, 91]. There has been an emerging interest in studying neural networks for program synthesis. A line of work studies training a neural network to directly generate the outputs given the inputs [74, 73, 95, 131, 133]. In particular, Devlin et al. study the Karel domain [73]. However, as shown in [73], this approach is incapable of handling the case when the number of input-output examples is small, and is hard to generalize to unseen inputs.

Recent work study using neural networks to generate programs in a domain-specific language (DSL) from a few input-output examples [74, 35, 201, 209, 295]. Several work synthesize programs for FlashFill tasks, which are in the string transformation domain [74, 201]. Other work synthesize programs in a LISP-style DSL for array manipulation [209, 295]. In particular, [295] also study the idea of encoding the state of the transformed inputs as it is updated during execution. However, these DSLs only include sequential programs, and do not support more complex control flows such as loops and conditionals in our studied Karel problem. Prior works also consider incorporating syntax constraints and information from program execution to facilitate program synthesis [74, 260, 35]. However, all these works generate the whole program, and use its execution results to guide the synthesis process; in contrast, our work leverages more fine-grained yet generic semantic information that can be gathered during executing programs in most imperative languages. As a result, our approach's performance is significantly better than previous work [35].

Previous work also study program synthesis given intermediate states. For example, [238] propose to synthesize the program from demonstration videos, which can be viewed as sequences of states. In such a problem, all intermediate states can be extracted from the videos. On the contrary, in the input-output program synthesis problem studied in our work, the input to the program synthesizer provides only the initial state and the final state. Thus, our synthesizer is required to address the challenge of inferring intermediate states, which is mainly tackled by our execution-guided synthesis algorithm.

In contrast to training a neural network to generate the entire program, a recent line of research studies using a neural network to guide the symbolic program search based on the input-output specification, so that the search process prioritizes the operators that have higher domain-specific scores predicted by the neural networks [22, 252]. Instead of predicting such domain-specific scores to guide the program search, we directly incorporate the domain knowledge by executing partial programs, and utilize the execution results for program generation of the neural network synthesizer. Meanwhile, recent work propose to leverage the full execution traces in the context of program repair [262]. Our work is the first to leverage partial execution traces for the program synthesis task, which is a much harder task than program repair.

4.7 Discussion

In this work, we propose two general and principled techniques to better leverage the semantic information for neural program synthesis: (1) execution-guided synthesis; and (2) synthesizer ensemble. On a rich DSL with complex control flows, we achieve a significant performance gain over the existing work, which demonstrates that utilizing the semantic information is crucial in boosting the performance of neural program synthesis approaches. We believe that our techniques are also beneficial to other program generation applications, and we consider extending our techniques to handle programming languages with richer semantics as important future work. At the same time, we have observed that utilizing existing reinforcement learning techniques does not provide much performance gain when combined with our approaches. We believe that there is plenty of room left for further improvement, and we are also interested in exploring this problem in the future.

Chapter 5

Latent Execution for Neural Program Synthesis

Program synthesis from input-output (IO) examples has been a long-standing challenge. While recent works demonstrated limited success on domain-specific languages (DSL), it remains highly challenging to apply them to real-world programming languages, such as C. Due to complicated syntax and token variation, there are three major challenges: **(1)** unlike many DSLs, programs in languages like C need to compile first and are not executed via interpreters; **(2)** the program search space grows exponentially when the syntax and semantics of the programming language become more complex; and **(3)** collecting a large-scale dataset of real-world programs is non-trivial. As a first step to address these challenges, we propose LaSynth and show its efficacy in a *restricted-C* domain (i.e., C code with tens of tokens, with sequential, branching, loop and simple arithmetic operations but no library call). More specifically, LaSynth learns the latent representation to approximate the execution of partially generated programs, even if they are incomplete in syntax (addressing **(1)**). The learned execution significantly improves the performance of next token prediction over existing approaches, facilitating search (addressing **(2)**). Finally, once trained with randomly generated ground-truth programs and their IO pairs, LaSynth can synthesize more concise programs that resemble human-written code. Furthermore, retraining our model with these synthesized programs yields better performance with fewer samples for both Karel and C program synthesis, indicating the promise of leveraging the learned program synthesizer to improve the dataset quality for input-output program synthesis (addressing **(3)**). When evaluating on whether the program execution outputs match the IO pairs, LaSynth achieves 55.2% accuracy on generating simple C code with tens of tokens including loops and branches, outperforming existing approaches without executors by around 20%. [1]

---

[1] The material in this chapter is based on Chen et al. [50]. The code is available at `https://github.com/Jungyhuk/latent-execution`.

5.1 Introduction

Program synthesis from input-output (IO) pairs, also called programming by example (PBE), requires high-level reasoning and remains a challenging problem for deep models. Unlike Natural Language Processing (NLP) [20, 72] and perceptual tasks such as Computer Vision (CV) [69, 107], the mapping from IO pairs to the program itself is hard to model. Many works attempt to learn a direct mapping from training samples, but often found that it is already difficult to achieve a low training error, and generalization to new problems is even harder. Alternatively, one might choose to formulate program synthesis as a search problem: to find the program that satisfies IO pairs. Unfortunately, the search space of programs is often vast and highly non-smooth, i.e., a small perturbation of the program often leads to a complete change of the output.

While there are many previous works on programming by example tasks [22, 74, 35], they mainly focus on Domain Specific Languages (DSLs), and cannot be easily applied to popular general-purpose programming languages. For example, to synthesize C programs, we need to deal with both high-level control flows (e.g., branching and loop) and low-level operations (e.g., which variable is the target of assignment). Moreover, unlike DSLs (e.g., Karel) for which it is feasible to implement a per-line interpreter, C programs need compilation and a partial C program cannot execute. On the other hand, some recent works investigate natural language descriptions as the auxiliary information of the program specification, and they evaluate neural program synthesis models on constrained or simplified competitive programming problems [144, 6, 108, 43, 17]. Although some of these works demonstrate promising results for synthesizing Python or C code, they require manual annotations of natural language specifications [144] or large-scale pre-training on human-written programs [43, 17], and the performance significantly degrades when only input-output examples are fed into the synthesizer [6].

To synthesize C programs from input-output examples only, we propose LaSynth, which generates the program in a recurrent and token-by-token manner. As the first contribution on model architectures for program synthesis, we propose to use two latent *parallel representations* in the recurrent model. One representation is learned from regular recurrent models as in autoregressive language models [115], with the double attention mechanism over IO pairs proposed in RobustFill [74] and an operation predictor that models the arithmetic relationship between the program input and output. The second representation, named *Latent Execution Trace (LaET)*, models the hypothetical input signal for the remaining partial program to execute to get to the desired output. Motivated by the line of work on execution-guided program synthesis [238, 85, 295, 47], we learn a latent representation for C programs which are not executed via interpreters, and train the model given only IO pairs without the intermediate program execution states. The two parallel representations are trained end-to-end.

As the second contribution on dataset construction, we demonstrate that it is possible to automatically construct a C codebase that is of high quality, controllable and concise through our proposed program synthesis procedure. Specifically, starting from randomly generated C programs that might contain a lot of redundant statements, we show that via *iterative*

retraining, the subsequent generated code from our learned model becomes more concise and similar to human-written ones. Moreover, learning directly from the generated code leads to better performance given the same amount of samples, and improves the sample efficiency. We observe similar results when applying our iterative retraining technique to Karel [35], another programming by example benchmark consisting of randomly generated programs. Although the initial Karel dataset includes a large proportion of complicated programs with different control flow constructs, we demonstrate that nearly half of the problems can be solved by straight-line programs, which again confirms that randomly generated programs tend to be unnecessarily complicated. We envision that the iterative retraining procedure could greatly reduce laborious efforts in human codebase collection in future research.

As the third contribution, we show for the first time that short C code in a restricted domain (tens of tokens, no library call) with sequential, branching, loop and simple arithmetic operations can be effectively synthesized from IO pairs only. In particular, while LaSynth tends to generate more concise programs (and does not have exact token match with random generated ground truth code), when measuring whether the program execution outputs match the IO pairs, LaSynth achieves 55.2% accuracy, and outperforms existing neural program synthesis models by around 20%. These results demonstrate the effectiveness of learning latent execution traces.

5.2 Problem Setup

In programming by example tasks, the program specification is a set of input-output examples [74, 35]. Specifically, we provide the synthesizer with a set of K input-output pairs $\{(I^{(k)}, O^{(k)})\}_{k=1}^{K}$ ($\{IO\}^{K}$ in short). These input-output pairs are annotated with a ground truth program $P^{\star}$, so that $P^{\star}(I^{(k)}) = O^{(k)}$ for any $k \in \{1, 2, ..., K\}$. To measure the program correctness, we include another set of held-out test cases $\{IO\}_{test}^{K_{test}}$ that differs from $\{IO\}^{K}$. The goal of the program synthesizer is to predict a program P from $\{IO\}^{K}$, so that $P(I) = P^{\star}(I) = O$ for any $(I, O) \in \{IO\}^{K} + \{IO\}_{test}^{K_{test}}$.

C Program Synthesis. In this work, we make the first attempt of synthesizing C code in a restricted domain from input-output examples only, and we focus on programs for list processing. List processing tasks have been studied in some prior works on input-output program synthesis, but they synthesize programs in restricted domain-specific languages instead of full-fledged popular programming languages [22, 196, 197].

Our C code synthesis problem brings new challenges for programming by example. Compared to domain-specific languages, the syntax and semantics of C are much more complicated, which significantly enlarges the program search space. Meanwhile, learning good representations for partially decoded programs also becomes more difficult. In particular, prior neural program synthesizers that utilize per-line interpreters for the programming language to guide the synthesis and representation learning [47, 227, 193, 85, 197] are not directly applicable to C. Although it is possible to dump some intermediate variable states during C code execution [39], since partial C programs are not executable, we are able to

Figure 5.1: Illustration of the C program synthesis pipeline. For dataset construction, we develop a random program generator to sample random C programs, then execute the program over randomly generated inputs and obtain the outputs. The input-output pairs are fed into the neural program synthesizer to predict the programs. Note that the synthesized program can be more concise than the original random program.

Figure 5.2: (a) An overview of LaSynth model architecture. (b), (c), and (d) present the details of the program decoder, latent executor, and the operation predictor. Note that the operation predictor is specialized for numerical calculation, and thus is not used for the Karel domain.

obtain all the execution states only until a full C code is generated, which is too late to include them in the program decoding process. In particular, the intermediate execution state is not available when the partial program is syntactically invalid, and this happens more frequently for C due to its syntax design.

5.3 Program Synthesis with Learned Execution

In this section, we present LaSynth which learns to represent the execution of partial programs to guide the synthesis process. Figure 5.2(a) provides an overview of LaSynth model architecture which consists of two components, the *program decoder* and the *latent executor*. We present the core design below, and defer more details to Appendix D.1 and Appendix D.2.

Model Overview

At a high level, the program decoder (Fig. 5.2(b)) takes a latent vector h_{t-1} that represents the generated partial program, the previous (generated) program token p_{t-1}, and outputs the latent vector h_t and the next program token p_t to be generated at time step t:

$$(h_t, p_t) = \text{ProgramDecoder}(h_{t-1}, p_{t-1}; IO_{t-1}) \tag{5.1}$$

Here the recurrent model is conditioned on the IO pair IO_{t-1}. When $IO_t = IO := (I, O)$ for every t, i.e., IO_t remains *constant* over the entire recurrent generation process, Eqn. 5.1 represents the standard recurrent architecture used in most autoregressive natural language models [115, 249], and is also used in prior works on program synthesis from input-output examples [74, 35].

For program decoding, the decoder first takes two attention vectors s_t^I and s_t^O computed from IO pairs and latent vector h_{t-1} via double attention [74], and utilizes a max pooling layer to compute an aggregated embedding m_t for all IO pairs (Fig. 5.2(b)):

$$m_t = \text{MaxPool}_{j \in \{1,2,\dots,K\}}(\tanh(W[s_t^{I(j)}; s_t^{O(j)}])) \tag{5.2}$$

Here the superscript (j) indicates that the representation is for the j-th IO pair, $[a; b]$ is vector concatenation of a and b, and W is a trainable matrix. To facilitate the prediction of long programs, we compute an attention vector d_t over previously generated program tokens using the standard attention mechanism [20, 171]:

$$d_t = \text{Attention}(m_t, \{p_0, \dots, p_{t-1}\}) \tag{5.3}$$

Finally, the next token p_t is sampled from $\mathbb{P}[p_t] = \text{Softmax}(V d_t)_{p_t}$ where V is a trainable matrix.

Latent Executor Design

As shown in our experiments (Section 5.5), the standard program decoder architecture may not be able to achieve strong performance in program synthesis when the program complexity increases. One main reason is that the standard program decoder only takes the initial IO pairs as the input without considering the program execution, thus the learned representation

for the partial program does not effectively guide the synthesis process. Motivated by prior works that utilize execution traces for Karel program synthesis [47, 227, 238], in this paper, we introduce *latent executor* (Fig. 5.2(c)) which maintains a second representation $\hat{I}_t$ during program decoding. Intuitively, $\hat{I}_{t-1}$ models the *hypothetical input* of the partial program $p_{t...T}$ so that its output becomes O. Given the estimated input $\hat{I}_{t-1}$ and the latent vector h_t, the latent executor returns $\hat{I}_t$ at the next time step t:

$$\hat{I}_t = \text{LatentExecutor}(\hat{I}_{t-1}, h_t) \tag{5.4}$$

The collection of $\{\hat{I}_t\}_{t=0}^T$ is the *latent execution trace (LaET)*. With the help of latent executor, we now use the IO pairs $IO_{t-1} := (\hat{I}_{t-1}, O)$ instead of (I, O) for the program decoder (Eqn. 5.1).

End-to-end Training

We train our model with supervised learning, by minimizing the sum of token prediction loss $\mathcal{L}_{Prog}$, and the latent executor loss $\mathcal{L}_{Exec}$:

$$\mathcal{L} = \mathcal{L}_{Prog} + \mathcal{L}_{Exec} \tag{5.5}$$

Specifically, $\mathcal{L}_{Prog} := \sum_{t=1}^T \text{Loss}(p_t, p_t^\star)$ is the step-by-step cross-entropy loss between the predicted programs $p_{1...T}$ and the ground truth programs $p_{1...T}^\star$.

For latent executor, since the semantics of partial programs (e.g., partial C programs) are not always well-defined, there is no step-by-step training supervision. However, the output of the executor should be consistent with the program specification after taking the annotated ground truth program as the input. Therefore, we set $\hat{I}_0 = I$ (true input) and minimize the distance between $\hat{I}_T$ and O (true output) after the program finishes:

$$\mathcal{L}_{Exec} = \text{Loss}(\hat{I}_T, O) \tag{5.6}$$

Note that $\mathcal{L}_{Exec}$ does not rely on any assumptions of the partial program semantics, and thus is applicable to both domain-specific languages and general-purpose programming languages such as C. In our evaluation, equipping with the latent executor significantly improves the program prediction performance, where each program could include up to 256 tokens.

Data Regeneration and Iterative Retraining

Interestingly, once our model is trained on the initial random generated programs $\mathcal{D}_0$, the predicted program becomes more concise and resembles human-written code. While the exact token match accuracy is low even on the training set, the model still satisfies the IO pairs for many problems. We leverage such a phenomenon to construct a new dataset $\mathcal{D}_1$ with higher-quality programs from $\mathcal{D}_0$. Specifically, we run beam search on the trained model to predict program $p_{0...T}$ given input-output pairs in the training set. If model prediction $p_{0...T}$

Table 5.1: The comparison between our restricted C domain and existing programming by example tasks.

	Control flow	Variables	Arithmetics	No helper functions
Restricted C (Ours)	✓	✓	✓	✓
Karel [35]	✓	–	–	–
DeepCoder [22]	–	✓	✓	–
FlashFill [98]	–	–	–	–

satisfies all the input-output examples and held-out cases, we replace the original program $p^\star_{0...T}$ with $p_{0...T}$ in $\mathcal{D}_1$, and keep $p^\star_{0...T}$ otherwise. Afterward, we re-train the model on $\mathcal{D}_1$. In Sec. 5.5, we will demonstrate that the retraining process further improves the model performance, especially with smaller training datasets.

5.4 Restricted C Program Synthesis Domain

In this section, we discuss our restricted C program synthesis domain, and our operation predictor design for improving the numerical reasoning ability of program synthesis models.

Data Generation

Collecting large-scale high-quality datasets for program synthesis requires a lot of human efforts, and we aim to reduce the manual work for dataset construction.

Our data generator is built upon Csmith [274], a random C code generation tool originally designed for finding bugs in compilers. Following the common practice of generating input-output pairs, for each program, we randomly sample 5 numerical lists as the program inputs, and execute the program to obtain the corresponding output lists. This is similar to existing works on PBE problems that sample programs based on a probabilistic context-free grammar, randomly generate valid inputs for the programs and obtain the outputs [201, 73, 22]. This creates infinite samples for synthesizing programs in domain-specific languages. While the programs sampled in this way differ from human-written code, Sec. 5.3 shows that they can be converted to be more concise and human-like.

The subset of language features used. Our generated program has variable declaration, variable assignment, and expressions with addition or subtraction operations. The programs also have non-sequential statements, including `If` statements, `For` loops, `Continue` and `Break` statements. Except for the input argument which is a list, all variables declared are integers, and all program statements are integer manipulation. Each expression has at most 2 mathematical operations, and chaining the full C program could perform multi-step numerical calculation (e.g., `p0 = p0 - p1 + p2; p0 = p0 - 1;`). Looping statements other than `For` (i.e., `While` or `Do-While` loops) are not supported. Note that we only constrain the final

program length (≤ 256 tokens) and the program can have nested for-loops and complicated if-conditions.

Post-processing. We perform a few post-processing steps to obtain our final programs from programs generated by Csmith (see Fig. 5.1 for an example). We resample different components of the program, so that (1) each constant numerical value lies in $[-4, 4]$, (2) mathematical operators only contain addition and subtraction, and (3) upper/lower limits of `For` loops are positive and within the length of the list. Programs are discarded if they are trivial (e.g., constant or identity mappings), or the input-output examples include values out of the range $[-4, 4]$.

Final dataset. We reweight the program distribution so that at least half of them include `For` loops. Our full dataset includes $500K$ samples in the training set, $1K$ samples in the validation set, and $1K$ samples in the test set. As shown in Fig. 5.1, the randomly sampled program may contain redundant statements, which can be easily avoided by human programmers. We compare our restricted C domain to prior datasets of programming by example in Table 5.1.

Program Decoding with the Operation Predictor

For program decoder, predicting the next program token p_t is non-trivial, especially when mathematical reasoning is required [222, 152]. To improve the program synthesis performance for domains involving numerical calculation, such as our restricted C domain, we design an associative memory structure named *operation predictor* (Fig. 5.2(d)), based on the following intuition: given the input $I = 2$ and output $O = 4$, human would infer that "$O = I + 2$" might be the desired operation and write down the code accordingly. To materialize such an intuition, we create a pre-computed table that covers all possible integer addition and subtraction operations for valid input and output list values. We defer the details of the model architecture to Appendix D.1. The program decoding process remains similar to the one described in Sec. 5.3, and we highlight the key differences as follows.

The operation predictor takes two attention vectors s_t^I and s_t^O as the representations of input-output examples, and yields an operator embedding $\hat{op}_t$. To compute the aggregated embedding vector for all input-output examples, we modify Eqn. 5.2 to also take $\hat{op}_t$ as an input of the max pooling layer:

$$m_t = \text{MaxPool}_{j \in \{1,2,...,K\}}(\tanh(W[s_t^{I(j)}; s_t^{O(j)}; \hat{op}_t^{(j)}])) \tag{5.7}$$

To train the operation predictor, we add an additional loss $\mathcal{L}_{Op}$:

$$\mathcal{L} = \mathcal{L}_{Prog} + \mathcal{L}_{Exec} + \mathcal{L}_{Op} \tag{5.8}$$

$\mathcal{L}_{Op}$ is designed to ensure that the operation predictor predicts operations related to IO pairs, and we defer the details to Appendix D.1.

Limitations. In our current implementation of the operation predictor, the operation table is only able to enumerate the arithmetic operations over a pre-defined constant set, thus

Table 5.2: The comparison between LaSynth and baseline neural program synthesis models in our evaluation.

	LaSynth	Exec [47]	Shin et al. [227]	Bunel et al. [35]	RobustFill [74]	Property Signatures [196]
+ Program execution	✓	✓	✓	−	−	−
No interpreter needed	✓	−	−	✓	✓	✓

it requires that the set of possible numerical values in input-output pairs is finite. One way of extending our operation predictor to support potentially unbounded numerical calculation is to combine it with the subword tokenizer, which has been commonly used in recent language models [72, 43, 17]. We consider designing general-purpose number representation for better mathematical reasoning as future work.

5.5 Experiments

In this section, we discuss our results on synthesizing programs in Karel and C languages. We first show that LaSynth achieves competitive performance on Karel benchmark. Then we present the results on our restricted C benchmark, and demonstrate that our approach significantly outperforms existing neural program synthesis models. Finally, we discuss the effect of iterative retraining.

Karel Program Synthesis

Evaluation Setup

Karel domain. Karel is an educational programming language [204], and has been studied in recent works on neural program synthesis from input-output examples [73, 35, 47, 227]. A Karel program controls a robot in a 2D grid world. There are instructions that control the robot, e.g., `move`, `turnLeft` and `PutMarker`, as well as conditionals and loops, i.e., `if`, `repeat` and `while`.

We train and evaluate all models on the Karel dataset introduced in [35]. The dataset contains randomly sampled programs from the Karel DSL ($1.1M$ training samples, $2.5K$ samples in the validation set and $2.5K$ samples in the test set). Each program includes 5 input-output pairs as the specification, and the sixth pair as the held-out test case. Following the prior work, we evaluate two metrics: (1) **Exact Match**: the predicted program is the same as the ground truth; (2) **Generalization**: the predicted program satisfies both the input-output pairs and the held-out input-output test case.

Baselines. *Bunel et al.* [35] designed the first program synthesis model for the Karel benchmark with a similar high-level design as RobustFill, but they use convolutional neural networks (CNN) to encode the Karel grid maps. Compared to LaSynth, this model does

Table 5.3: Results on Karel dataset. **Gen** and **Exact** denote generalization and exact match accuracies.

Approach	Gen	Exact
LaSynth	83.68%	41.12%
Exec [47]	**86.04%**	39.40%
Bunel et al. [35]	77.12%	32.17%
Shin et al. [227]	81.30%	**42.80%**

Figure 5.3: Generalization accuracies with different training data sizes on Karel. With the full training set, the accuracies are 86.04%, 89.28% and 89.36% for training on random programs, retraining for 1 and 2 iterations.

not utilize any program execution information, and does not include our latent executor. Instead of directly synthesizing the program from input-output examples, the model in *Shin et al.* [227] first predicts the execution traces containing the robot actions from the input-output pairs, then decodes the program based on the execution traces. This model improves the prediction performance over Bunel et al., but it requires the full execution traces for model training and an interpreter for execution. *Exec* [47] leverages the execution states of partial generated programs to guide the subsequent synthesis process, but the execution states are obtained from the Karel interpreter rather than learned by the model, thus this approach represents the ideal scenario where the partial programs could be executable.

Our model architecture for Karel is largely similar to the model for C code synthesis, except that we employ the CNN encoder in Bunel et al. [35] in our program decoder and latent executor. The comparison with baseline models is shown in the middle block of Table 5.2. All models use the beam search for decoding programs, with the beam size of 64.

Results

We present the results of LaSynth and baseline model architectures in Table 5.3. First, LaSynth outperforms all baselines that do not incorporate the partial program execution information, and achieves competitive performance compared with the Exec algorithm that requires an interpreter to obtain the partial program execution states. In particular, LaSynth achieves a higher generalization accuracy than Shin et al. with lower exact match accuracy, showing that decoded programs by LaSynth are more different from randomly generated programs. Although Shin et al. also model the program execution by predicting the robot actions, the prediction of the action traces does not take the program structure into account, resulting in the inferior performance.

Figure 5.4: Program distributions after iterative retraining on Karel. (a) The distributions of different program types. *Seq-only*: no control flows. *If-only*: the program includes If statements but no loops. *Repeat/While-only*: the program includes Repeat/While loops, but no other control flow constructs. *Mixture*: the program includes at least two types of control flow constructs. (b) The distributions of programs with different token lengths.

C Code Synthesis

Evaluation Setup

Given the variety of C programs, we observe that the exact match accuracies of models are mostly nearly 0. Therefore, we focus on evaluating the generalization accuracy, and we consider the predicted program to be correct when it satisfies both the 5 input-output examples and 5 held-out test cases.

Baselines. We compare the full LaSynth with its multiple ablated versions:

- `NoExecutor`. The program decoder (Eqn. 5.1) always takes the initial input-output pairs as the input; i.e,. $\hat{I}_t = I_0$ for every t.

- `NoPartialExecutor`. $\hat{I}_t = I_0 = I$ for every t and additionally h_T is regularized so that LatentExecutor(I_0, h_T) matches the output O under loss $\mathcal{L}_{Exec}$. Therefore, no partial latent execution.

- `NoOpPredictor`. The max pooling layer only takes the vectors computed by the double attention as the input (Eqn. 5.2).

- `NoAttentionInDecoding`. There is no attention over decoded program tokens, and the output of the max pooling layer is directly fed into the output softmax layer; i.e., $\mathbb{P}[p_t] = \text{Softmax}(V m_t)_{p_t}$ (compared to Eqn. 5.3).

We also compare with existing neural program synthesis models with good performance on related tasks, as shown in the rightmost block of Table 5.2. *RobustFill* [74] is the state-of-the-art neural network architecture on FlashFill benchmark, which synthesizes string

```
int * func_1(int a[])
{
    int p_0 = 0;
    int l_25 = 4;
    a[p_0] = 1;
    --a[l_25];
    return a;
}
```

```
int * func_1(int a[])
{
    int p_0 = 2;
    int l_12 = 3;
    for (p_0 = 1; p_0 <= 2; p_0++)
    {
        a[p_0]--;
    }
    a[l_12] = a[l_12] + 4;
    return a;
}
```

```
int * func_1(int a[])
{
    int p_0 = 0;
    int l_7 = 3;
    int l_8 = 1;
    a[l_8] = (a[l_7] - a[p_0]);
    for (p_0 = 3; p_0 <= 4; p_0++)
    {
        for (int p_1 = 1; p_1 <= 2; p_1++)
        {
            a[p_1] = a[p_1] + a[p_0];
            a[p_1] = a[p_1] + 2;
        }
    }
    return a;
}
```

Figure 5.5: Sample programs that could be correctly predicted by LaSynth, but wrongly predicted by models without the latent executor. These programs require multiple different operations for different input list elements.

manipulation programs in a domain-specific language. As described in Sec. 5.3, the input-output encoder and the program decoder architectures in RobustFill are similar to LaSynth, except that it does not include the latent executor, operation predictor, and the attention on the decoded program sequence.

Property Signatures [196] was designed for synthesizing list manipulation programs in domain-specific languages, but instead of taking the raw input and output lists as the neural network input, they design some properties that distinguish different programs, then take the values of these properties as the model input. A sample property could be whether the program output is the same as the input, and the property values could be "All True", "All False", or "Mixed", indicating that the property always holds for any input-output pair in the specification, never holds, and holds for some pairs but not others, respectively. We customize the original design [196] for our setting. First, our property set takes the format of $O = C + I$? and $O = C - I$?, where $C \in [-4, 4]$. For example, $O = 2 + I$? means whether the output O could be calculated by adding 2 to the input I. These properties focus more on numerical calculation, similar to our operation predictor. Second, different from the task in [196], our C programs sometimes manipulate only a subset of the input lists, thus encoding the list with a single property value is inappropriate. Instead, we compute the property value per element in input-output pairs, use a bi-directional LSTM to encode the property values as a sequence, then take the outputs of the bi-LSTM for program prediction.

Results

Table 5.4 presents the results, where all models are trained on the initial random programs. The full LaSynth outperforms other variants, and improves the performance of RobustFill and Property Signatures by around 20%. We also increase the model size of RobustFill

Table 5.4: Results on C dataset.

Approach	Accuracy
LaSynth	**55.2%**
NoAttentionInDecoding	53.5%
NoOpPredictor	53.7%
NoPartialExecutor	42.9%
NoExecutor	38.6%
RobustFill [74]	37.6%
Property Signatures [196]	34.5%

Figure 5.6: Accuracies of different program types on C dataset.

Figure 5.7: Results of iterative retraining on the C dataset. (a) Accuracies with different training data sizes. With the full training set, the accuracies are 55.2%, 56.0% and 56.5% for training on random programs, retraining for 1 and 2 iterations, respectively. (b) The program distributions after each retraining iteration.

to see if the improvement comes from larger model size, but the results are not better. In particular, the latent executor significantly increases the prediction accuracy, and achieves better results than NoPartialExecutor, which shows that learning latent execution traces leads to better partial program representations. In Fig. 5.5, we present sample programs that could be correctly synthesized by LaSynth, but models without the latent executor provide the wrong prediction. We observe that the latent executor is beneficial when the program involves different manipulations for different list elements, e.g., more than one For loop and different mathematical calculations. Our breakdown results on programs of different complexity also justify this observation. We first present the results on programs with different control flow constructs in Fig. 5.6. Specifically, *Seq-only* includes programs with no control flow constructs, *For-only* includes programs with For loops but no If statements, and *Mixture* includes programs with both For loops and If statements. Then we demonstrate

Figure 5.8: Results on programs of different token lengths on the C dataset. (a) The program token length distributions after each retraining iteration. (b) The accuracies on programs of different token lengths.

the results on different program lengths in Fig. 5.8b. We show that LaSynth achieves decent performance on long and complicated programs, while the accuracies of baseline models drop dramatically.

Discussion of Iterative Retraining

In Fig. 5.3, we show the effectiveness of retraining on decoded Karel programs (Sec. 5.3). We observe that retraining for one iteration is sufficient, and it significantly improves the generalization accuracy by over 3%. To understand the differences between predicted programs and randomly generated programs, we demonstrate the changes of dataset distributions after each retraining iteration in Fig. 5.4a and 5.4b. We observe that the model learns to predict more concise programs than the ground truth for a large proportion of input-output examples, and considerably alters the dataset distribution so that it becomes more concentrated on short programs with simplified control flow structures. Specifically, from Fig. 5.4a, although the initial Karel dataset seems to include a large proportion of complicated programs with different control flow constructs, our model synthesizes straight-line programs for nearly half of the samples, which means that many loops and branches in the annotated ground truth programs are unnecessary. This distribution shift also explains the gap between the exact match and generalization accuracies. The program distribution after the second retraining iteration is largely similar to the first iteration, thus retraining for more iterations does not considerably improve the performance. Note that in the second iteration, the synthesizer tends to generate slightly more complicated programs than the first iteration, in order to deal with the cases when the input-output examples oversimplify the intended program functionality. For example, sometimes the input-output examples do not cover the edge cases that the robot may encounter, thus adding additional `If` branches could avoid the crashes when testing on held-out cases.

Fig. 5.7a presents the results of retraining on decoded C programs. Similarly, retraining

improves the prediction accuracy, especially when the training set is small. From Fig. 5.7b and 5.8a, we again observe that the model tends to predict shorter programs than the random code, and it eliminates unnecessary control flows to simplify the programs. We present more examples in Appendix D.3.

5.6 Related Work

Programming by example. Programming by example problems have been widely studied with various applications, and recent works have developed deep neural networks as program synthesizers [99, 201, 74, 35]. Most prior works focus on synthesizing programs in domain-specific languages, such as FlashFill [201, 74, 252] for string transformation, Karel [35, 227, 47, 102] for simulated robot navigation, and LISP-style languages for list manipulation [22, 209, 295, 192]. In this work, we make the first attempt of synthesizing C code in a restricted domain from input-output examples only, and we focus on the list manipulation domain.

Some recent works investigate the limitations of synthetic datasets and the ambiguity in program specifications for neural program synthesis [229, 62, 237, 147]. These works focus on reducing the bias of data distributions and generating more diverse input-output pairs, while our data regeneration aims to improve the quality of programs. We consider incorporating both lines of work to further improve the dataset quality as future work. In addition, drawing the inspiration from self-training and bootstrapping techniques developed for other applications [184, 1, 178, 269] to extend our iterative retraining scheme is also another future direction.

Execution-guided program synthesis. To learn better program representations, some recent works incorporate the execution information to guide the synthesis process [238, 295, 227, 47, 85, 247, 23, 102, 197, 193, 172]. In particular, leveraging partial program execution states improves the performance for several program synthesis tasks [47, 295, 85, 193]. However, existing approaches rely on program interpreters to provide the intermediate execution results whenever applicable. In contrast, we demonstrate that our latent executor learns the latent execution traces (LaET) without such a requirement. Besides program synthesis, execution traces have also been utilized for other software engineering applications [5, 179].

Neural execution. Our latent executor is related to prior works on learning to execute algorithms [284, 251, 273] and programs [29]. They focus on predicting execution results for full algorithms and programs, but do not utilize them for program synthesis. Latent state prediction has also been studied in other applications such as task-oriented dialogue systems [180, 291] and robotics [205].

5.7   Discussion

In this work, we propose LaSynth, which learns the latent representation to approximate the execution of partial programs, even if their semantics are not well-defined. We demonstrate the possibility of synthesizing elementary C code from input-output examples only, and leveraging learned execution significantly improves the prediction performance by around 20%. Meanwhile, compared to the randomly generated programs, LaSynth synthesizes more concise programs that resemble human-written code, and training on these synthesized programs further improves the prediction performance for both Karel and C program synthesis. Our results indicate the promise of leveraging the learned program synthesizer to improve the dataset quality for programming by example tasks.

We consider extending our approach to synthesize more complicated real-world code as future work. For example, we will integrate our latent executor into large-scale pre-trained language models, which could further improve the performance of those program synthesis models taking natural language specifications. We will also study program synthesis problems with unbounded input ranges and different type signatures, which could be approached with the usage of subword tokenizers.

Part III

Synthesis for Software Engineering Applications

Chapter 6

Tree-to-tree Neural Networks for Program Translation

Program translation is an important tool to migrate legacy code in one language into an ecosystem built in a different language. In this chapter, we employ deep neural networks toward tackling this problem. We observe that program translation is a modular procedure, in which a sub-tree of the source tree is translated into the corresponding target sub-tree at each step. To capture this intuition, we design a tree-to-tree neural network to translate a source tree into a target one. Meanwhile, we develop an attention mechanism for the tree-to-tree model, so that when the decoder expands one non-terminal in the target tree, the attention mechanism locates the corresponding sub-tree in the source tree to guide the expansion of the decoder. We evaluate the program translation capability of our tree-to-tree model against several state-of-the-art approaches. Compared against other neural translation models, we observe that our approach is consistently better than the baselines with a margin of up to 15 points. Further, our approach can improve the previous state-of-the-art program translation approaches by a margin of 20 points on the translation of real-world projects [1].

6.1 Introduction

Programs are the main tool for building computer applications, the IT industry, and the digital world. Various programming languages have been invented to facilitate programmers to develop programs for different applications. At the same time, the variety of different programming languages also introduces a burden when programmers want to combine programs written in different languages together. Therefore, there is a tremendous need to enable program translation between different programming languages.

Nowadays, to translate programs between different programming languages, typically programmers would manually investigate the correspondence between the grammars of the two languages, then develop a rule-based translator. However, this process can be inefficient

---

[1]The material in this chapter is based on Chen et al. [49].

and error-prone. In this work, we make the first attempt to examine whether we can leverage deep neural networks to build a program translator automatically.

Intuitively, the program translation problem in its format is similar to a natural language translation problem. Some previous work propose to adapt phrase-based statistical machine translation (SMT) for code migration [189, 134, 188]. Recently, neural network approaches, such as sequence-to-sequence-based models, have achieved the state-of-the-art performance on machine translation [20, 60, 86, 106, 249]. In this work, we study neural machine translation methods to handle the program translation problem. However, a big challenge making a sequence-to-sequence-based model ineffective is that, unlike natural languages, programming languages have rigorous grammars and are not tolerant to typos and grammatical mistakes. It has been demonstrated that it is very hard for an RNN-based sequence generator to generate syntactically correct programs when the lengths grow large [136].

In this work, we observe that the main issue of an RNN that makes it hard to produce syntactically correct programs is that it entangles two sub-tasks together: (1) learning the grammar; and (2) aligning the sequence with the grammar. When these two tasks can be handled separately, the performance can typically boost. For example, Dong et al. employ a tree-based decoder to separate the two tasks [80]. In particular, the decoder in [80] leverages the tree structural information to (1) generate the nodes at the same depth of the parse tree using an LSTM decoder; and (2) expand a non-terminal and generate its children in the parse tree. Such an approach has been demonstrated to achieve the state-of-the-art results on several semantic parsing tasks.

Inspired by this observation, we hypothesize that the structural information of both source and target parse trees can be leveraged to enable such a separation. Inspired by this intuition, we propose tree-to-tree neural networks to combine both a tree encoder and a tree decoder. In particular, we observe that in the program translation problem, both source and target programs have their parse trees. In addition, a cross-language compiler typically follows a modular procedure to translate the individual sub-components in the source tree into their corresponding target ones, and then compose them to form the final target tree. Therefore, we design the workflow of a tree-to-tree neural network to align with this procedure: when the decoder expands a non-terminal, it locates the corresponding sub-tree in the source tree using an attention mechanism, and uses the information of the sub-tree to guide the non-terminal expansion. In particular, a tree encoder is helpful in this scenario, since it can aggregate all information of a sub-tree to the embedding of its root, so that the embedding can be used to guide the non-terminal expansion of the target tree.

We follow the above intuition to design the tree-to-tree translation model. Some existing work [233, 146] propose tree-based autoencoder architectures. However, in these models, the decoder can only access to a single hidden vector representing the source tree, thus they are not performant on the translation task. In our evaluation, we demonstrate that without an attention mechanism, the translation performance is 0% in most cases, while using an attention mechanism could boost the performance to > 90%. Another work [33] proposes a tree-based attentional encoder-decoder architecture for natural language translation, but their model performs even worse than the attentional sequence-to-sequence baseline model.

One main reason is that their attention mechanism calculates the attention weights of each
node independently, which does not well capture the hierarchical structure of the parse trees.
In our work, we design a *parent attention feeding* mechanism that formulates the dependence
of attention maps between different nodes, and show that this attention mechanism further
improves the performance of our tree-to-tree model considerably, especially when the size of
the parse trees grows large (i.e., $20\% - 30\%$ performance gain). To the best of our knowledge,
this is the first successful demonstration of tree-to-tree neural network architecture proposed
for translation tasks in the literature.

To test our hypothesis, we develop two novel program translation tasks, and employ a
Java to C# benchmark used by existing program translation works [188, 189]. First, we
compare our approach against several neural network approaches on our proposed two tasks.
Experimental results demonstrate that our tree-to-tree model outperforms other state-of-
the-art neural networks on the program translation tasks, and yields a margin of up to 5%
on the token accuracy and up to 15% on the program accuracy. Further, we compare our
approach with previous program translation approaches on the Java to C# benchmark, and
the results show that our tree-to-tree model outperforms previous state-of-the-art by a large
margin of 20% on program accuracy. These results demonstrate that our tree-to-tree model
is promising toward tackling the program translation problem. Meanwhile, we believe that
our proposed tree-to-tree neural network could also be adapted to other tree-to-tree tasks,
and we consider it as future work.

6.2   Program Translation Problem

In this work, we consider the problem of translating a program in one language into another.
One approach is to model the problem as a machine translation problem between two
languages, and thus numerous neural machine translation approaches can be applied.

For the program translation problem, however, a unique property is that each input
program unambiguously corresponds to a unique parse tree. Thus, rather than modeling the
input program as a sequence of tokens, we can consider the problem as translating a source
tree into a target tree. Note that most modern programming languages are accompanied
with a well-developed parser, so we can assume that the parse trees of both the source and
the target programs can be easily obtained.

The main challenge of the problem in our consideration is that the cross-compiler for
translating programs typically does not exist. Therefore, even if we assume the existence of
parsers for both the source and the target languages, the translation problem itself is still
non-trivial. We formally define the problem as follows.

Definition 2 (Program translation). *Given two programming languages $\mathcal{L}_s$ and $\mathcal{L}_t$, each
being a set of instances (p_k, T_k), where p_k is a program, and T_k is its corresponding parse tree.
We assume that there exists a translation oracle π, which maps instances in $\mathcal{L}_s$ to instances
in $\mathcal{L}_t$. Given a dataset of instance pairs (i_s, i_t) such that $i_s \in \mathcal{L}_s, i_t \in \mathcal{L}_t$ and $\pi(i_s) = i_t$, our
problem is to learn a function F that maps each $i_s \in \mathcal{L}_s$ into $i_t = \pi(i_s)$.*

Figure 6.1: Translating a CoffeeScript program into JavaScript. The sub-component in the
CoffeeScript program and its corresponding translation in JavaScript are highlighted.

In this work, we focus on the problem setting that we have a set of paired source and
target programs to learn the translator. Note that all existing program translation works [134,
188, 189] also study the problem under such an assumption. When such an alignment is
lacking, the program translation problem is more challenging. Several techniques for NMT
have been proposed to handle this issue, such as dual learning [106], which have the potential
to be extended for the program translation task. We leave these more challenging problem
setups as future work.

6.3   Tree-to-tree Neural Network

In this section, we present our design of the tree-to-tree neural network. We first motivate
the design, and then present the details.

Program Translation as a Tree-to-tree Translation Problem

Figure 6.1 presents an example of translation from CoffeeScript to JavaScript. We observe
that an interesting property of the program translation problem is that the translation process
can be modular. The figure highlights a sub-component in the source tree corresponding
to x=1 and its translation in the target tree corresponding to x=1;. This correspondence is
independent of other parts of the program. Consider when the program grows longer and this
statement may repetitively occur multiple times, it may be hard for a sequence-to-sequence
model to capture the correspondence based on only token sequences without structural
information. Thus, such a correspondence makes it a natural solution to locate the referenced
sub-tree in the source tree when expanding a non-terminal in the target tree into a sub-tree.

Figure 6.2: Tree-to-tree workflow: The arrows indicate the computation flow. Blue solid arrows indicate the flow from/to the left child, while orange dashed arrows are for the right child. The black dotted arrow from the source tree root to the target tree root indicates that the LSTM state is copied. The green box denotes the expanding node, and the grey one denotes the node to be expanded in the queue. The sub-tree of the source tree corresponding to the expanding node is highlighted in yellow. The right corner lists the formulas to predict the value of the expanding node.

Tree-to-tree Neural Network

Inspired by the above motivation, we design the tree-to-tree neural network, which follows an encoder-decoder framework to encode the source tree into an embedding, and decode the embedding into the target tree. To capture the intuition of the modular translation process, the decoder employs an attention mechanism to locate the corresponding source sub-tree when expanding the non-terminal. We illustrate the workflow of a tree-to-tree model in Figure 6.2, and present each component of the model below.

Converting a tree into a binary one. Note that the source and target trees may contain multiple branches. Although we can design tree-encoders and tree-decoders to handle trees with arbitrary number of branches, we observe that encoder and decoder for binary trees can be more effective. Thus, the first step is to convert both the source tree and the target tree into a binary tree. To this end, we employ the Left-Child Right-Sibling representation for this conversion.

Binary tree encoder. The encoder employs a Tree-LSTM [244] to compute embeddings for both the entire source tree and each of its sub-tree. In particular, consider a node N with the value t_s in its one-hot encoding representation, and it has two children N_L and N_R, which are its left child and right child respectively. The encoder recursively computes the embedding for N from the bottom up.

Assume that the left child and the right child maintain the LSTM state (h_L, c_L) and
(h_R, c_R) respectively, and the embedding of t_s is x. Then the LSTM state (h, c) of N is
computed as

$$(h, c) = \text{LSTM}(([h_L; h_R], [c_L; c_R]), x) \tag{6.1}$$

where $[a; b]$ denotes the concatenation of a and b. Note that a node may lack one or both of
its children. In this case, the encoder sets the LSTM state of the missing child to be zero.

Binary tree decoder.   The decoder generates the target tree starting from a single root
node. The decoder first copies the LSTM state (h, c) of the root of the source tree, and
attaches it to the root node of the target tree. Then the decoder maintains a queue of all
nodes to be expanded, and recursively expands each of them. In each iteration, the decoder
pops one node from the queue, and expands it. In the following, we call the node being
expanded *the expanding node.*

First, the decoder will predict the value of expanding node. To this end, the decoder
computes the embedding e_t of the expanding node N, and then feeds it into a softmax
regression network for prediction:

$$t_t = \textbf{argmax}\ \textbf{softmax}(W e_t) \tag{6.2}$$

Here, W is a trainable matrix of size $V_t \times d$, where V_t is the vocabulary size of the outputs
and d is the embedding dimension. Note that e_t is computed using the attention mechanism,
which we will explain later.

The value of each node t_t is a non-terminal, a terminal, or a special $\langle \text{EOS} \rangle$ token. If
$t_t = \langle \text{EOS} \rangle$, then the decoder finishes expanding this node. Otherwise, the decoder generates
one new node as the left child and another new node as the right child of the expanding one.
Assume that (h', c'), (h'', c'') are the LSTM states of its left child and right child respectively,
then they are computed as:

$$(h', c') = \text{LSTM}_L((h, c), B t_t) \tag{6.3}$$
$$(h'', c'') = \text{LSTM}_R((h, c), B t_t) \tag{6.4}$$

Here, B is a trainable word embedding matrix of size $d \times V_t$. Note that the generation of
the left child and right child use two different sets of parameters for LSTM_L and LSTM_R
respectively. These new children are pushed into the queue of all nodes to be expanded.
When the queue is empty, the target tree generation process terminates.

Notice that although the sets of terminal and non-terminal are disjoint, it is necessary
to include the $\langle \text{EOS} \rangle$ token for the following reasons. First, due to the left-child-right-
sibling encoding, although a terminal does not have a child, since it could have a right child
representing its sibling in the original tree, $\langle \text{EOS} \rangle$ is still needed for predicting the right
branch. Meanwhile, we combine the terminal and non-terminal sets into a single vocabulary
V_t for the decoder, and do not incorporate the knowledge of grammar rules into the model,
thus the model needs to infer whether a predicted token is a terminal or a non-terminal itself.

In our evaluation, we find that a well-trained model never generates a left child for a terminal, which indicates that the model can learn to distinguish between terminals and non-terminals correctly.

Attention mechanism to locate the source sub-tree.   Now we consider how to compute e_t. One straightforward approach is to compute e_t as h, which is the hidden state attached to the expanding node. However, in doing so, the embedding will soon forget the information about the source tree when generating deep nodes in the target tree, and thus the model yields a very poor performance.

To make better use of the information of the source tree, our tree-to-tree model employs an attention mechanism to locate the source sub-tree corresponding to the sub-tree rooted at the expanding node. Specifically, we compute the following probability:

$$P(N_s \text{ is the source sub-tree corresponding to } N_t | N_t)$$

where N_t is the expanding node. We denote this probability as $P(N_s|N_t)$, and we compute it as

$$P(N_s|N_t) \propto \mathbf{exp}(h_s^T W_0 h_t) \tag{6.5}$$

where W_0 is a trainable matrix of size $d \times d$.

To leverage the information from the source tree, we compute the expectation of the hidden state value across all N_s conditioned on N_t, i.e.,

$$e_s = \mathrm{E}[h_{N_s}|N_t] = \sum_{N_s} h_{N_s} \cdot P(N_s|N_t) \tag{6.6}$$

This embedding can then be combined with h, the hidden state of the expanding node, to compute e_t as follows:

$$e_t = \mathbf{tanh}(W_1 e_s + W_2 h) \tag{6.7}$$

where W_1, W_2 are trainable matrices of size $d \times d$ respectively.

Parent attention feeding.   In the above approach, the attention vectors e_t are computed independently to each other, since once e_t is used for predicting the node value t_t, e_t is no longer used for further predictions. However, intuitively, the attention decisions for the prediction of each node should be related to each other. For example, for a non-terminal node N_t in the target tree, suppose that it is related to N_s in the source tree, then it is very likely that the attention weights of its children should focus on the descendants of N_s. Therefore, when predicting the attention vector of a node, the model should leverage the attention information of its parent as well.

Following this intuition, we propose a *parent attention feeding* mechanism, so that the attention vector of the expanding node is taken into account when predicting the attention vectors of its children. Formally, besides the embedding of the node value t_t, we modify the inputs to LSTM_L and LSTM_R of the decoder in Equations (6.3) and (6.4) as below:

$$(h', c') = \text{LSTM}_L((h, c), [Bt_t; e_t]) \tag{6.8}$$

$$(h'', c'') = \text{LSTM}_R((h, c), [Bt_t; e_t]) \tag{6.9}$$

Notice that these formulas in their formats coincide with the input-feeding method for sequential neural networks [171], but their meanings are different. For sequential models, the input attention vector belongs to the previous token, while here it belongs to the parent node. In our evaluation, we will show that such a parent attention feeding mechanism significantly improves the performance of our tree-to-tree model.

6.4 Evaluation

In this section, we evaluate our tree-to-tree neural network with several baseline approaches on the program translation task. To do so, we first describe three benchmark datasets in Section 6.4 for evaluating different aspects; then we evaluate our tree-to-tree model against several baseline approaches, including the state-of-the-art neural network approaches and program translation approaches.

Datasets

To evaluate different approaches for the program translation problem, we employ three tasks: (1) a synthetic translation task from an imperative language to a functional language; (2) translation between CoffeeScript and JavaScript, which are both full-fledged languages; and (3) translation of real-world projects from Java to C#, which has been used as a benchmark in the literature. Due to the space limit, we present the translation tasks of real-world programming languages (i.e., task (2) and (3)) below, and we discuss the synthetic task in Appendix E.5.

For the CoffeeScript-JavaScript task, CoffeeScript employs a Python-style succinct syntax, while JavaScript employs a C-style verbose syntax. To control the program lengths of the training and test data, we develop a pCFG-based program generator and a subset of the core CoffeeScript grammar. We also limit the set of variables and literals to restrict the vocabulary size. We utilize the CoffeeScript compiler to generate the corresponding ground truth JavaScript programs. The grammar used to generate the programs in our experiments can be found in Appendix E.4. In doing so, we obtain a set of CoffeeScript-JavaScript pairs, and thus we can build a CoffeeScript-to-JavaScript dataset, and a JavaScript-to-CoffeeScript dataset by exchanging the source and the target. To build the dataset, we randomly generate 100,000 pairs of source and target programs for training, 10,000 pairs as the development set, and 10,000 pairs for testing. We guarantee that there is no overlap among training, development and test sets, and all samples are unique in the dataset. More statistics of the dataset can be found in Appendix E.2.

For the evaluation on Java to C#, we tried to contact the authors of [188] for their dataset, but our emails were not responded. Thus, we employ the same approach as in [188] to crawl several open-source projects, which have both a Java and a C# implementation. Same as in [188], we pair the methods in Java and C# based on their file names and method names. The statistics of the dataset is summarized in Appendix E.2. Due to the change of the versions of these projects, the concrete dataset in our evaluation may differ from [188]. For each project, we apply ten-fold validation on matched method pairs, as in [188].

Metrics

The main metric evaluated in our evaluation is the *program accuracy*, which is the percentage of the predicted target programs that are exactly the same as the ground truth in the dataset. Note that the program accuracy is an underestimation of the true accuracy based on semantic equivalence, and this metric has been used in [188]. This metric is more meaningful than other previously proposed metrics, such as syntax-correctness and dependency-graph-accuracy, which are not directly comparable to semantic equivalence. We also measure another metric called *token accuracy*, and we defer the details to Appendix E.3.

Model Details

We evaluate our tree-to-tree model against a sequence-to-sequence model [20, 254], a sequence-to-tree model [80], and a tree-to-sequence model [86]. Note that for a sequence-to-sequence model, there can be four variants to handle different input-output formats. For example, given a program, we can simply tokenize it into a sequence of tokens. We call this format as *raw program*, denoted as P. We can also use the parser to parse the program into a parse tree, and then serialize the parse tree as a sequence of tokens. Our serialization of a tree follows its depth-first traversal order, which is the same as [254]. We call this format as *parse tree*, denoted as T. For both input and output formats, we can choose either P or T. For a sequence-to-tree model, we have two variants based on its input format being either P or T; note that the sequence-to-tree model generates a tree as output, and thus requires its output format to be T (unserialized). Similarly, the tree-to-sequence model has two variants, and our tree-to-tree only has one form. Therefore, we have 9 different models in our evaluation.

The hyper-parameters used in different models can be found in Appendix E.1. The baseline models have employed their own input-feeding or parent-feeding method that is analogous to our parent attention feeding mechanism.

Results on the CoffeeScript-JavaScript Task

For the CoffeeScript-JavaScript task, we create several datasets named as XY-ZW: X and Y (C or J) indicate the source and target languages respectively; Z (A or B) indicates the vocabulary; and W (S or L) indicates the program length. In particular, vocabulary A uses {x,y} as variable names and {0,1} as literals; vocabulary B uses all alphabetical characters

	Tree2tree			Seq2seq				Seq2tree		Tree2seq	
	T→T	T→T (-PF)	T→T (-Attn)	P→P	P→T	T→P	T→T	P→T	T→T	T→P	T→T
CoffeeScript to JavaScript translation											
CJ-AS	**99.57%**	98.80%	0.09%	90.51%	79.82%	92.73%	89.13%	86.52%	88.50%	96.96%	92.18%
CJ-BS	**99.75%**	99.67%	0%	97.44%	16.26%	98.05%	93.89%	91.97%	88.22%	96.83%	78.77%
CJ-AL	**97.15%**	71.52%	0%	21.04%	0%	0%	0%	80.82%	78.60%	82.55%	46.94%
CJ-BL	**95.60%**	78.61%	0%	19.26%	9.98%	25.35%	42.08%	76.12%	76.21%	83.61%	26.83%
JavaScript to CoffeeScript translation											
JC-AS	**87.75%**	85.11%	0.09%	83.07%	86.13%	73.88%	86.31%	86.86%	86.99%	71.61%	86.53%
JC-BS	**86.37%**	80.35%	0%	80.49%	85.94%	69.77%	85.28%	85.06%	84.25%	66.82%	85.31%
JC-AL	**78.59%**	54.93%	0%	77.10%	77.30%	65.52%	75.70%	77.11%	77.59%	60.75%	75.75%
JC-BL	**75.62%**	44.40%	0%	73.14%	73.96%	61.92%	74.51%	74.34%	71.56%	57.09%	73.86%

Table 6.1: Program accuracy for the translation between CoffeeScript and JavaScript.

as variable names, and all single digits as literals. S means that the CoffeeScript programs
has 10 tokens on average; and L for 20.

The program accuracy results are presented in Table 6.1. We can observe that our
tree2tree model outperforms all baseline models on all datasets. Especially, on the dataset
with longer programs, the program accuracy significantly outperforms all seq2seq models
by a large margin, i.e., up to 75%. Its margin over a seq2tree model can also reach around
20 points. These results demonstrate that tree2tree model is more capable of learning the
correspondence between the source and the target programs; in particular, it is significantly
better than other baselines at handling longer inputs.

Meanwhile, we perform an ablation study to compare the full tree2tree model with (1)
tree2tree without parent attention feeding (T→T (-PF)) and (2) tree2tree without attention
(T→T (-Attn)). We observe that the full tree2tree model significantly outperforms the other
alternatives. In particular, on JC-BL, the full tree2tree's program accuracy is 30 points
higher than the tree2tree model without parent attention feeding.

More importantly, we observe that the program accuracy of tree2tree model without
the attention mechanism is nearly 0%. Note that such a model is similar to a tree-to-tree
autoencoder architecture. This result shows that our novel architecture can significantly
outperform previous tree-to-tree-like architectures on the program translation task.

However, although our tree2tree model performs better than other baselines, it still could
not achieve 100% accuracy. After investigating into the prediction, we find that the main
reason is because the translation may introduce temporary variables. Because such temporary
variables appear very rarely in the training set, it could be hard for a neural network to
infer correctly in these cases. Actually, the longer the programs are, the more temporary
variables that the cross-compiler may introduce, which makes the prediction harder. We
consider further improving the model to handle this problem as future work.

In addition, we observe that for the translation from JavaScript to CoffeeScript, the
improvements of the tree2tree model over the baselines are much smaller than for CoffeeScript

	Tree2tree	J2C#	1pSMT	mppSMT
		Reported in [188]		
Lucene	**72.8%**	21.5%	21.6%	40.0%
POI	**72.2%**	18.9%	34.6%	48.2%
Itext	**67.5%**	25.1%	24.4%	40.6%
JGit	**68.7%**	10.7%	23.0%	48.5%
JTS	**68.2%**	11.7%	18.5%	26.3%
Antlr	31.9% (**58.3%**)	10.0%	11.5%	49.1%

Table 6.2: Program accuracy on the Java to C# translation. In the parentheses, we present the program accuracy that can be achieved by increasing the training set.

to JavaScript translation. We attribute this to the fact that the target programs are much shorter. For example, for a CoffeeScript program with 20 tokens, its corresponding JavaScript program may contain more than 300 tokens. Thus, the model needs to predict much fewer tokens for a CoffeeScript program than a JavaScript program, so that even seq2seq models can achieve a reasonably good accuracy. However, still, we can observe that our tree2tree model outperforms all baselines.

Results on Real-world Projects

We now compare our approach with three state-of-the-art program translation approaches, i.e., J2C# [126], 1pSMT [189], and mppSMT [188], on the real-world benchmark from Java to C#. Here, J2C# is a rule-based system, 1pSMT directly applies the phrase-based SMT on sequential programs, and mppSMT is a multi-phase phrase-based SMT approach that leverages both the raw programs and their parse trees.

The results are summarized in Table 6.2. For previous approaches, we report the results from [188]. We can observe that our tree2tree approach can significantly outperform the previous state-of-the-art on all projects except Antlr. The improvements range from 20.2% to 41.9%.

On Antlr, the tree2tree model performs worse. We attribute this to the fact that Antlr contains too few data samples for training. We test our hypothesis by constructing another training and validation set from all other 5 projects, and test our model on the entire Antlr. We observe that our tree2tree model can achieve a test accuracy of 58.3%, which is 9 points higher than the state-of-the-art. Therefore, we conclude that our approach can significantly outperform previous program translation approaches when there are sufficient training data.

6.5   Related Work

Statistical approaches for program translation.   Some recent work have applied statistical machine translation techniques to program translation [7, 134, 188, 189, 190, 195].

For example, several works propose to adapt phrase-based statistical machine translation models and leverage grammatical structures of programming languages for code migration [134, 188, 189]. In [190], Nguyen et al. propose to use Word2Vec representation for APIs in libraries used in different programming languages, then learn a transformation matrix for API mapping. On the contrary, our work is the first to employ deep learning techniques for program translation.

Neural networks with tree structures. Recently, various neural networks with tree structures have been proposed to employ the structural information of the data [80, 212, 201, 277, 9, 244, 294, 232, 86, 289, 233, 146, 33]. In these work, different tree-structured encoders are proposed for embedding the input data, and different tree-structured decoders are proposed for predicting the output trees. In particular, in [233, 146], they propose tree-structured autoencoders to learn vector representations of trees, and show better performance on tree reconstruction and other tasks such as sentiment analysis. Another work [33] proposes to use a tree-structured encoder-decoder architecture for natural language translation, where both the encoder and the decoder are variants of the RNNG model [84]; however, the performance of their model is slightly worse than the sequence-to-sequence model with attention, which is mainly due to the fact that their attention mechanism can not condition the future attention weights on previously computed ones. In this work, we are the first to demonstrate a successful design of tree-to-tree neural network for translation tasks.

Neural networks for parsing. Other work study using neural networks to generate parse trees from input-output examples [80, 254, 4, 212, 277, 9, 84, 48, 53]. In [80], Dong et al. propose a seq2tree model that allows the decoder RNN to generate the output tree recursively in a top-down fashion. This approach achieves the state-of-the-art results on several semantic parsing tasks. Some other work incorporate the knowledge of the grammar into the architecture design [277, 212] to achieve better performance on specific tasks. However, these approaches are hard to generalize to other tasks. Again, none of them is designed for program translation or proposes a tree-to-tree architecture.

Neural networks for code generation. A recent line of research study using neural networks for code generation [22, 74, 201, 162, 212, 277]. In [162, 212, 277], they study generating code in a DSL from inputs in natural language or in another DSL. However, their designs require additional manual efforts to adapt to new DSLs in consideration. In our work, we consider the tree-to-tree model as a generic approach that can be applied to any grammar.

6.6   Discussion

In this work, we are the first to consider neural network approaches for the program translation problem, and are the first to demonstrate a successful design of tree-to-tree neural network

combining both a tree-RNN encoder and a tree-RNN decoder for translation tasks. Extensive evaluation demonstrates that our tree-to-tree neural network outperforms several state-of-the-art models. This renders our tree-to-tree model as a promising tool toward tackling the program translation problem. In addition, we believe that our proposed tree-to-tree neural network has the potential to generalize to other tree-to-tree tasks, and we consider it as future work.

At the same time, we observe many challenges in program translation that existing techniques are not capable of handling. For example, the models are hard to generalize to programs longer than the training ones; it is unclear how to handle an infinite vocabulary set that may be employed in real-world applications; further, the training requires a dataset of aligned input-output pairs, which may be lacking in practice. We consider all these problems as important future work in the research agenda toward solving the program translation problem.

Chapter 7

Neural Rewriter for Code Optimization and beyond

Search-based methods for hard combinatorial optimization are often guided by heuristics. Tuning heuristics in various conditions and situations is often time-consuming. In this paper, we propose NeuRewriter that learns a policy to pick heuristics and rewrite the local components of the current solution to iteratively improve it until convergence. The policy factorizes into a region-picking and a rule-picking component, each parameterized by a neural network trained with actor-critic methods in reinforcement learning. NeuRewriter captures the general structure of combinatorial problems and shows strong performance in three versatile tasks: expression simplification, online job scheduling and vehicle routing problems. NeuRewriter outperforms the expression simplification component in Z3 [68]; outperforms DeepRM [174] and Google OR-tools [92] in online job scheduling; and outperforms recent neural baselines [186, 142] and Google OR-tools [92] in vehicle routing problems. [1]

7.1 Introduction

Solving combinatorial problems is a long-standing challenge and has a lot of practical applications (e.g., job scheduling, theorem proving, planning, decision making). While problems with specific structures (e.g., shortest path) can be solved efficiently with proven algorithms (e.g, dynamic programming, greedy approach, search), many combinatorial problems are NP-hard and rely on manually designed heuristics to improve the quality of solutions [2, 218, 135].

Although it is usually easy to come up with many heuristics, determining when and where such heuristics should be applied, and how they should be prioritized, is time-consuming. It takes commercial solvers decades to tune to strong performance in practical problems [68, 235, 92].

---

[1] The material in this chapter is based on Chen et al. [51]. The code is available at `https://github.com/facebookresearch/neural-rewriter`.

To address this issue, previous works use neural networks to predict a complete solution from scratch, given a complete description of the problem [253, 174, 142, 95]. While this avoids search and tuning, a direct prediction could be difficult when the number of variables grows.

Improving iteratively from an existing solution is a common approach for continuous solution spaces, e.g, trajectory optimization in robotics [177, 245, 154]. However, such methods relying on gradient information to guide the search, is not applicable for discrete solution spaces due to indifferentiablity.

To address this problem, we directly learn a neural-based policy that improves the current solution by iteratively rewriting a local part of it until convergence. Inspired by the problem structures, the policy is factorized into two parts: the region-picking and the rule-picking policy, and is trained end-to-end with reinforcement learning, rewarding cumulative improvement of the solution.

We apply our approach, NeuRewriter, to three different domains: expression simplification, online job scheduling, and vehicle routing problems. We show that NeuRewriter is better than strong heuristics using multiple metrics. For expression simplification, NeuRewriter outperforms the expression simplification component in Z3 [68]. For online job scheduling, under a controlled setting, NeuRewriter outperforms Google OR-tools [92] in terms of both speed and quality of the solution, and DeepRM [174], a neural-based approach that predicts a holistic scheduling plan, by large margins especially in more complicated setting (e.g., with more heterogeneous resources). For vehicle routing problems, NeuRewriter outperforms two recent neural network approaches [186, 142] and Google OR-tools [92]. Furthermore, extensive ablation studies show that our approach works well in different situations (e.g., different expression lengths, non-uniform job/resource distribution), and transfers well when distribution shifts (e.g., test on longer expressions than those used for training).

7.2   Problem Setup

Let $\mathcal{S}$ be the space of all feasible solutions in the problem domain, and $c : \mathcal{S} \to \mathbb{R}$ be the cost function. The goal of optimization is to find $\arg\min_{s \in \mathcal{S}} c(s)$. In this work, instead of finding a solution from scratch, we first construct a feasible one, then make incremental improvement by iteratively applying *local rewriting rules* to the existing solution until convergence. Our rewriting formulation is especially suitable for problems with the following properties: **(1)** a feasible solution is easy to find; **(2)** the search space has well-behaved local structures, which could be utilized to incrementally improve the solution. For such problems, a complete solution provides a full context for the improvement using a rewriting-based approach, allowing additional features to be computed, which is hard to obtain if the solution is generated from scratch; meanwhile, different solutions might share a common routine towards the optimum, which could be represented as local rewriting rules. For example, it is much easier to decide whether to postpone jobs with large resource requirements when an existing job schedule is provided. Furthermore, simple rules like swapping two jobs could improve the performance.

Formally, each solution is a *state*, and each local region and the associated rewriting rule is an *action*.

Optimization as a rewriting problem. Let $\mathcal{U}$ be the rewriting ruleset. Suppose s_t is the current solution (or state) at iteration t. We first compute a state-dependent *region set* $\Omega(s_t)$, then pick a region $\omega_t \in \Omega(s_t)$ using the *region-picking policy* $\pi_\omega(\omega_t|s_t)$. We then pick a rewriting rule u_t applicable to that region ω_t using the *rule-picking policy* $\pi_u(u_t|s_t[\omega_t])$, where $s_t[\omega_t]$ is a subset of state s_t. We then apply this rewriting rule $u_t \in \mathcal{U}$ to $s_t[\omega_t]$, and obtain the next state $s_{t+1} = f(s_t, \omega_t, u_t)$. Given an initial solution (or state) s_0, our goal is to find a sequence of rewriting steps $(s_0, (\omega_0, u_0)), (s_1, (\omega_1, u_1)), ..., (s_{T-1}, (\omega_{T-1}, u_{T-1})), s_T$ so that the final cost $c(s_T)$ is minimized.

To tackle a rewriting problem, rule-based rewriters with manually-designed rewriting routines have been proposed [105]. However, manually designing such routines is not a trivial task. An incomplete set of routines often leads to an inefficient exhaustive search, while a set of kaleidoscopic routines is often cumbersome to design, hard to maintain and lacks flexibility.

In this paper, we propose to train a neural network instead, using reinforcement learning. Recent advance in deep reinforcement learning suggests the potential of well-trained models to discover novel effective policies, such as demonstrated in Computer Go [230] and video games [198]. Moreover, by leveraging reinforcement learning, our approach could be extended to a broader range of problems that could be hard for rule-based rewriters and classic search algorithms. For example, we can design the reward to take the validity of the solution into account, so that we can start with an infeasible solution and then move towards a feasible one. On the other hand, we can also train the neural network to explore the connections between different solutions in the search space. In our evaluation, we demonstrate that our approach (1) mitigates laborious human efforts, (2) discovers novel rewriting paths from its own exploration, and (3) finds better solution to optimization problem than the current state-of-the-art and traditional heuristic-based software packages tuned for decades.

7.3 Neural Rewriter Model

In the following, we present the design of our rewriting model, i.e., NeuRewriter. We first provide an overview of our model framework, then present the design details for different applications.

Model Overview

Figure 7.5 illustrates the overall framework of our neural rewriter, and we describe the two key components for rewriting as follows. More details can be found in Appendix F.3.

Score predictor. Given the state s_t, the score predictor computes a score $Q(s_t, \omega_t)$ for every $\omega_t \in \Omega(s_t)$, which measures the benefit of rewriting $s_t[\omega_t]$. A high score indicates that rewriting $s_t[\omega_t]$ could be desirable. Note that $\Omega(s_t)$ is a problem-dependent region set. For

expression simplification, $\Omega(s_t)$ includes all sub-trees of the expression parse trees; for job scheduling, $\Omega(s_t)$ covers all job nodes for scheduling; and for vehicle routing, it includes all nodes in the route.

Rule selector. Given $s_t[\omega_t]$ to be rewritten, the rule-picking policy predicts a probability distribution $\pi_u(s_t[\omega_t])$ over the entire ruleset $\mathcal{U}$, and selects a rule $u_t \in \mathcal{U}$ to apply accordingly.

Training Details

Let $(s_0, (\omega_0, u_0)), ..., (s_{T-1}, (\omega_{T-1}, u_{T-1})), s_T$ be the rewriting sequence in the forward pass.

Reward function. We define $r(s_t, (\omega_t, u_t))$ as $r(s_t, (\omega_t, u_t)) = c(s_t) - c(s_{t+1})$, where $c(\cdot)$ is the task-specific cost function in Section 7.2.

Q-Actor-Critic training. We train the region-picking policy π_ω and rule-picking policy π_u simultaneously. For $\pi_\omega(\omega_t|s_t; \theta)$, we parameterize it as a softmax of the underlying $Q(s_t, \omega_t; \theta)$ function:

$$\pi_\omega(\omega_t|s_t; \theta) = \frac{\exp(Q(s_t, \omega_t; \theta))}{\sum_{\omega_t} \exp(Q(s_t, \omega_t; \theta))} \tag{7.1}$$

and instead learn $Q(s_t, \omega_t; \theta)$ by fitting it to the cumulative reward sampled from the current policies π_ω and π_u:

$$L_\omega(\theta) = \frac{1}{T} \sum_{t=0}^{T-1} (\sum_{t'=t}^{T-1} \gamma^{t'-t} r(s'_t, (\omega'_t, u'_t)) - Q(s_t, \omega_t; \theta))^2 \tag{7.2}$$

Where T is the length of the episode (i.e., the number of rewriting steps), and γ is the decay factor.

For rule-picking policy $\pi_u(u_t|s_t[\omega_t]; \phi)$, we employ the Advantage Actor-Critic algorithm [240] with the learned $Q(s_t, \omega_t; \theta)$ as the critic, and thus avoid boot-strapping which could cause sample insufficiency and instability in training. This formulation is similar in spirit to soft-Q learning [104]. Denoting $\Delta(s_t, (\omega_t, u_t)) \equiv \sum_{t'=t}^{T-1} \gamma^{t'-t} r(s'_t, (\omega'_t, u'_t)) - Q(s_t, \omega_t; \theta)$ as the advantage function, the loss function of the rule selector is:

$$L_u(\phi) = -\sum_{t=0}^{T-1} \Delta(s_t, (\omega_t, u_t)) \log \pi_u(u_t|s_t[\omega_t]; \phi) \tag{7.3}$$

The overall loss function is $L(\theta, \phi) = L_u(\phi) + \alpha L_\omega(\theta)$, where α is a hyper-parameter.

7.4 Applications

In the following sections, we discuss the application of our rewriting approach to three different domains: expression simplification, online job scheduling, and vehicle routing. In expression simplification, we minimize the expression length using a well-defined semantics-preserving

Figure 7.1: The instantiation of NeuRewriter for different domains: **(a)** expression simplification; **(b)** job scheduling; and **(c)** vehicle routing. In **(a)**, s_t is the expression parse tree, where each square represents a node in the tree. The set $\Omega(s_t)$ includes every sub-tree rooted at a non-terminal node, from which the region-picking policy selects $\omega_t \sim \pi_\omega(\omega_t|s_t)$ to rewrite. Afterwards, the rule-picking policy predicts a rewriting rule $u_t \in \mathcal{U}$, then rewrites the sub-tree ω_t to get the new tree s_{t+1}. In **(b)**, s_t is the dependency graph representation of the job schedule. Each circle with index greater than 0 represents a job node, and node 0 is an additional one representing the machine. Edges in the graph reflect job dependencies. The region-picking policy selects a job ω_t to re-schedule from all job nodes, then the rule-picking policy chooses a moving action u_t for ω_t, then modifies s_t to get a new dependency graph s_{t+1}. In **(c)**, s_t is the current route, and ω_t is the node selected to change the visit order. Node 0 is the depot, and other nodes are customers with certain resource demands. The region-picking policy and the rule-picking policy work similarly to the job scheduling ones.

rewriting ruleset. In online job scheduling, we aim to reduce the overall waiting time of jobs. In vehicle routing, we aim to minimize the total tour length.

Expression Simplification

We first apply our approach to expression simplification domain. In particular, we consider expressions in Halide, a domain-specific language for high-performance image processing [213], which is widely used at scale in multiple products of Google (e.g., YouTube) and Adobe Photoshop. Simplifying Halide expressions is an important step towards the optimization of the entire code. To this end, a rule-based rewriter is implemented for the expressions, which is carefully tuned with manually-designed heuristics. The grammar of the expressions considered in the rewriter is specified in Appendix F.1.

Notice that the grammar includes a more comprehensive operator set than previous works on finding equivalent expressions, which consider only boolean expressions [8, 87] or a subset of algorithmic operations [8]. The rewriter includes hundreds of manually-designed rewriting templates. Given an expression, the rewriter checks the templates in a pre-designed order, and applies those rewriting templates that match any sub-expression of the input.

After investigating the rewriting templates in the rule-based rewriter, we find that a large number of rewriting templates enumerate specific cases for an *uphill rule*, which lengthens the expression first and shortens it later (e.g., "min/max" expansion). Similar to momentum terms in gradient descent for continuous optimization, such rules are used to escape a local

optimum. However, they should only be applied when the initial expression satisfies certain *pre-conditions*, which is traditionally specified by manual design, a cumbersome process that is hard to generalize.

Observing these limitations, we hypothesize that a neural network model has the potential of doing a better job than the rule-based rewriter. In particular, we propose to only keep the core rewriting rules in the ruleset, remove all unnecessary pre-conditions, and let the neural network decide which and when to apply each rewriting rule. In this way, the neural rewriter has a better flexibility than the rule-based rewriter, because it can learn such rewriting decisions from data, and has the ability of discovering novel rewriting patterns that are not included in the rule-based rewriter.

Ruleset. We incorporate two kinds of templates from Halide rewriting ruleset. The first kind is simple rules (e.g., $v - v \to 0$), while the second one is the uphill rules after removing their manually designed pre-conditions that do not affect the validity of the rewriting. In this way, a ruleset with $|\mathcal{U}| = 19$ categories is built. See Appendix F.2 for more details.

Model specification. We use expression parse trees as the input, and employ the N-ary Tree-LSTM designed in [244] as the input encoder to compute the embedding for each node in the tree. Both the score predictor and the rule selector are fully connected neural networks, taken the LSTM embeddings as the input. More details can be found in Appendix F.3.

Job Scheduling Problem

We also study the job scheduling problem, using the problem setup in [174].

Notation. Suppose we have a machine with D types of resources. Each job j is specified as $v_j = (\boldsymbol{\rho}_j, A_j, T_j)$, where the D-dimensional vector $\boldsymbol{\rho}_j = [\rho_{jd}]$ denotes the required portion $0 \leq \rho_{jd} \leq 1$ of the resource type d, A_j is the arrival timestep, and T_j is the duration. In addition, we define B_j as the scheduled beginning time, and $C_j = B_j + T_j$ as the completion time.

We assume that the resource requirement is fixed during the entire job execution, each job must run continuously until finishing, and no preemption is allowed. We adopt an online setting: there is a pending job queue that can hold at most W jobs. When a new job arrives, it can either be allocated immediately, or be added to the queue. If the queue is already full, to make space for the new job, at least one job in the queue needs to be scheduled immediately. The goal is to find a time schedule for every job, so that the average waiting time is as short as possible.

Ruleset. The set of rewriting rules is to re-schedule a job v_j and allocate it after another job $v_{j'}$ finishes or at its arrival time A_j. See Appendix F.2 for details of a rewriting step. The size of the rewriting ruleset is $|\mathcal{U}| = 2W$, since each job could only switch its scheduling order with at most W of its former and latter jobs respectively.

Representation. We represent each schedule as a directed acyclic graph (DAG), which describes the dependency among the schedule time of different jobs. Specifically, we denote each job v_j as a node in the graph, and we add an additional node v_0 to represent the machine. If a job v_j is scheduled at its arrival time A_j (i.e., $B_j = A_j$), then we add a directed edge

$\langle v_0, v_j \rangle$ in the graph. Otherwise, there must exist at least one job $v_{j'}$ such that $C_{j'} = B_j$ (i.e., job j starts right after job j'). We add an edge $\langle v_{j'}, v_j \rangle$ for every such job $v_{j'}$ to the graph. Figure 7.1(b) shows the setting, and we defer the embedding and graph construction details to Appendix F.3.

Model specification. To encode the graphs, we extend the Child-Sum Tree-LSTM architecture in [244], which is similar to the DAG-structured LSTM in [293]. Similar to the expression simplification model, both the score predictor and the rule selector are fully connected neural networks, and we defer the model details to Appendix F.3.

Vehicle Routing Problem

In addition, we evaluate our approach on vehicle routing problems studied in [142, 186]. Specifically, we focus on the Capacitated VRP (CVRP), where a single vehicle with limited capacity needs to satisfy the resource demands of a set of customer nodes. To do so, we construct multiple routes starting and ending at the depot, i.e., node 0 in Figure 7.1(c), so that the resources delivered in each route do not exceed the vehicle capacity, while the total route length is minimized.

We represent each vehicle routing problem as a sequence of the nodes visited in the tour, and use a bi-directional LSTM to embed the routes. The ruleset is similar to the job scheduling, where each node can swap with another node in the route. The architectures of the score predictor and rule selector are similar to job scheduling. More details can be found in Appendix F.3.

7.5 Experiments

We present the evaluation results in this section. To calculate the inference time, we run all algorithms on the same server equipped with 2 Quadro GP100 GPUs and 80 CPU cores. Only 1 GPU is used when evaluating neural networks, and 4 CPU cores are used for search algorithms. We set the timeout of search algorithms to be 10 seconds per instance. All neural networks in our evaluation are implemented in PyTorch [203].

Expression Simplification

Setup. To construct the dataset, we first generate random pipelines using the generator in Halide, then extract expressions from them. We filter out those irreducible expressions, then split the rest into 8/1/1 for training/validation/test sets respectively. See Appendix F.1 for more details.

Metrics. We evaluate the following two metrics: (1) *Average expression length reduction*, which is the length reduced from the initial expression to the rewritten one, and the length is defined as the number of characters in the expression; (2) *Average tree size reduction*, which is the number of nodes decreased from the initial expression parse tree to the rewritten one.

Figure 7.2: Experimental results of the expression simplification problem. In **(b)**, we train NeuRewriter on expressions of different lengths (described in the brackets).

Figure 7.3: Experimental results of the job scheduling problem varying the following aspects: **(a)** the number of resource types D; **(b)** job frequency; **(c)** resource distribution; **(d)** job length. For NeuRewriter, we describe training job distributions in the brackets. Workloads in **(a)** are with steady job frequency, non-uniform resource distribution, and non-uniform job length. In **(b)**, **(c)** and **(d)**, $D = 20$. In **(b)** and **(c)**, we omit the comparison with some approaches because their results are significantly worse; for example, the average slowdown of `EJF` is 14.53 on the dynamic job frequency, and 11.06 on the uniform resource distribution. More results can be found in Appendix F.4.

Figure 7.4: Experimental results of the vehicle routing problem with different number of customer nodes and vehicle capacity; e.g., VRP100, Cap50 means there are 100 customer nodes and the vehicle capacity is 50. **(a)** NeuRewriter outperforms multiple baselines and previous works [142, 186]. More results can be found in Appendix F.5. **(b)** We evaluate the generalization performance of NeuRewriter on problems from different distributions, and we describe the training problem distributions in the brackets.

Baselines. We examine the effectiveness of NeuRewriter against two kinds of baselines. The first kind of baselines are heuristic-based rewriting approaches, including `Halide-rule` (the rule-based Halide rewriter in Section 7.2) and `Heuristic-search`, which applies beam search to find the shortest rewriting with our ruleset at each step. Note that NeuRewriter does not use beam search.

In addition, we also compare our approach with Z3, a high-performance theorem prover developed by Microsoft Research [68]. Z3 provides two tactics to simplify the expressions: `Z3-simplify` performs some local transformation using its pre-defined rules, and `Z3-ctx-solver-simplify` traverses each sub-formula in the input expression and invokes the solver to find a simpler equivalent one to replace it. This search-based tactic is able to perform simplification not included in the Halide ruleset, and is generally better than the rule-based counterpart but with more computation. For `Z3-ctx-solver-simplify`, we set the timeout to be 10 seconds for each input expression.

Results. Figure 7.2a presents the main results. We can notice that the performance of `Z3-simplify` is worse than `Halide-rule`, because the ruleset included in this simplifier is more restricted than the Halide one, and in particular, it can not handle expressions with "max/min/select" operators. On the other hand, NeuRewriter outperforms both the rule-based rewriters and the heuristic search by a large margin. In particular, NeuRewriter could reduce the expression length and parse tree size by around 52% and 59% on average; compared to the rule-based rewriters, our model further reduces the average expression length and tree size by around 20% and 15% respectively. We observe that the main performance gain comes from learning to apply uphill rules appropriately in ways that

are not included in the manually-designed templates. For example, consider the expression $5 \leq \max(\max(v0, 3) + 3, \max(v1, v2))$, which could be reduced to *True* by expanding $\max(\max(v0, 3) + 3, \max(v1, v2))$ and $\max(v0, 3)$. Using a rule-based rewriter would require the need of specifying the pre-conditions recursively, which becomes prohibitive when the expressions become more complex. On the other hand, heuristic search may not be able to find the correct order of expanding the right hand size of the expression when more "min/max" are included, which would make the search less efficient.

Furthermore, NeuRewriter also outperforms `Z3-ctx-solver-simplify` in terms of both the result quality and the time efficiency, as shown in Figure 7.2a and Table 7.1a. Note that the implementation of Z3 is in C++ and highly optimized, while NeuRewriter is implemented in Python; meanwhile, `Z3-ctx-solver-simplify` could perform rewriting steps that are not included in the Halide ruleset. More results can be found in Appendix F.6.

Generalization to longer expressions. To measure the generalizability of our approach, we construct 4 subsets of the training set: $Train_{\leq 20}$, $Train_{\leq 30}$, $Train_{\leq 50}$ and $Train_{\leq 100}$, which only include expressions of length at most 20, 30, 50 and 100 in the full training set. We also build $Test_{>100}$, a subset of the full test set that only includes expressions of length larger than 100. The statistics of these datasets can be found in Appendix F.1.

We present the results of training our model on different datasets above in Figure 7.2b. Even trained on short expressions, NeuRewriter is still comparable with the Z3 solver. Thanks to local rewriting rules, our approach can generalize well even when operating on very different data distributions.

Job Scheduling Problem

Setup. We randomly generate $100K$ job sequences, and use $80K/10K/10K$ for training, validation and testing. Typically each job sequence includes ~ 50 jobs. We use an online setting where jobs arrive on the fly with a pending job queue of length $W = 10$. Unless stated otherwise, we generate initial schedules using *Earliest Job First* (`EJF`), which can be constructed with negligible overhead.

When the number of resource types $D = 2$, we follow the same setup as in [174]. The maximal job duration $T_{max} = 15$, and the latest job arrival time $A_{max} = 50$. With larger D, except changing the resource requirement of each job to include more resource types, other configurations stay the same.

Metric. Following `DeepRM` [174], we use the *average job slowdown* $\eta_j \equiv (C_j - A_j)/T_j \geq 1$ as our evaluation metric. Note that $\eta_j = 1$ means no slow down.

Job properties. To test the stability and generalizability of NeuRewriter, we change job properties (and their distributions): (1) *Number of resource types D*: larger D leads to more complicated scheduling; (2) *Average job arrival rate*: the probability that a new job will arrive, *Steady job frequency* sets it to be 70%, and *Dynamic job frequency* means the job arrival rate changes randomly at each timestep; (3) *Resource distribution*: jobs might require different resources, where some are *uniform* (e.g., half-half for resource 1 and 2) while others are *non-uniform* (see Appendix F.1 for the detailed description); (4) *Job lengths*: *Uniform*

job length: length of each job in the workload is either $[10, 15]$ (long) or $[1, 3]$ (short), and *Non-uniform job length*: workload has both short and long jobs. We show that NeuRewriter is fairly robust under different distributions. When trained on one distribution, it can generalize to others without performance collapse.

We compare NeuRewriter with three kinds of baselines.

Baselines on Manually designed heuristics: *Earliest Job First* (`EJF`) schedules each job in the increasing order of their arrival time. *Shortest Job First* (`SJF`) always allocates the shortest job in the pending job queue at each timestep, which is also used as a baseline in [174]. *Shortest First Search* (`SJFS`) searches over the shortest k jobs to schedule at each timestep, and returns the optimal one. We find that other heuristic-based baselines used in [174] generally perform worse than `SJF`, especially with large D. Thus, we omit the comparison.

Baselines on Neural network. We compare with `DeepRM` [174], a neural network also trained with RL to construct a solution from scratch.

Baselines on Offline planning. To measure the optimality of these algorithms, we also take an offline setting, where the entire job sequence is available before scheduling. Note that this is equivalent to assuming an unbounded length of the pending job queue. With such additional knowledge, this setting provides a strong baseline. We tried two offline algorithms: (1) `SJF-offline`, which is a simple heuristic that schedules each job in the increasing order of its duration; and (2) Google OR-tools [92], which is a generic toolbox for combinatorial optimization. For OR-tools, we set the timeout to be 10 seconds per workload, but we find that it can not achieve a good performance even with a larger timeout, and we defer the discussion to Appendix F.4.

Results on Scalability. As shown in Figure 7.3, NeuRewriter outperforms both heuristic algorithms and the baseline neural network `DeepRM`. In particular, while the performance of `DeepRM` and NeuRewriter are similar when $D = 2$, with larger D, `DeepRM` starts to perform worse than heuristic-based algorithms, which is consistent with our hypothesis that it becomes challenging to design a schedule from scratch when the environment becomes more complex. On the other hand, NeuRewriter could capture the bottleneck of an existing schedule that limits its efficiency, then progressively refine it to obtain a better one. In particular, our results are even better than offline algorithms that assume the knowledge of the entire job sequence, which further demonstrates the effectiveness of NeuRewriter. Meanwhile, we present the running time of OR-tools, `DeepRM` and NeuRewriter in Table 7.1b. We can observe that both `DeepRM` and NeuRewriter are much more time-efficient than OR-tools; on the other hand, the running time of NeuRewriter is comparable to `DeepRM`, while achieving much better results. More discussion can be found in Appendix F.4.

Results on Robustness. As shown in Figure 7.3, NeuRewriter excels in almost all different job distributions, except when the job lengths are uniform (short or long, Figure 7.3d), in which case existing methods/heuristics are sufficient. This shows that NeuRewriter can deal with complicated scenarios and is adaptive to different distributions.

Results on Generalization. Furthermore, NeuRewriter can also generalize to different distributions than those used in training, without substantial performance drop. This shows the power of local rewriting rules: using local context could yield more generalizable solutions.

	Time (s)
Z3-solver	1.375
NeuRewriter	0.159

(a)

	Time (s)
OR-tools	10.0
DeepRM	0.020
NeuRewriter	0.037

(b)

	VRP20	VRP50	VRP100
OR-tools	0.010	0.053	0.231
Nazari et al.	0.162	0.232	0.445
AM	0.036	0.168	0.720
NeuRewriter	0.133	0.211	0.398

(c)

Table 7.1: Average runtime (per instance) of different solvers (OR-tools [92] and the tactic `Z3-ctx-solver-simplify` of Z3 [68]) and RL-based approaches (NeuRewriter, DeepRM [174], Nazari et al. [186] and AM [142]) over the test set of: **(a)** expression simplification; **(b)** job scheduling; **(c)** vehicle routing.

Vehicle Routing Problem

Setup and Baselines. We follow the same training setup as [142, 186] by randomly generating vehicle routing problems with different number of customer nodes and vehicle capacity. We compare with two neural network approaches, i.e., AM [142] and Nazari et al. [186], and both of them train a neural network policy using reinforcement learning to construct the route from scratch. We also compare with OR-tools and several classic heuristics studied in [186].

Results. We first demonstrate our main results in Figure 7.4a, where we include the variant of each baseline that performs the best, and defer more results to Appendix F.5. Note that the initial routes generated for NeuRewriter are even worse than the classic heuristics; however, starting from such sub-optimal solutions, NeuRewriter is still able to iteratively improve the solutions and outperforms all the baseline approaches on different problem distributions. In addition, for VRP20 problems, we can compute the exact optimal solutions, which provides an average tour length of 6.10. We observe that the result of NeuRewriter (i.e., 6.16) is the closest to this lower bound, which also demonstrates that NeuRewriter is able to find solutions with better quality.

We also compare the runtime of the most competitive approaches in Table 7.1c. Note that the OR-Tools solver for vehicle routing problems is highly tuned and implemented in C++, while the RL-based approaches in comparison are implemented in Python. Meanwhile, following [186], to report the runtime of RL models, we decode a single instance at a time, thus there is potential room for speed improvement by decoding multiple instances per batch. Nevertheless, we can still observe that NeuRewriter achieves a better balance between the result quality and the time efficiency, especially with a larger problem scale.

Results on Generalization. Furthermore, in Figure 7.4b, we show that NeuRewriter can generalize to different problem distributions than training ones. In particular, they still exceed the performance of the classic heuristics, and are sometimes comparable or even better than the OR-tools. More discussion can be found in Appendix F.5.

Figure 7.5: The framework of our neural rewriter. Given the current state (i.e., solution to the optimization problem) s_t, we first pick a region ω_t by the region-picking policy $\pi_\omega(\omega_t|s_t)$, and then pick a rewriting rule u_t using the rule-picking policy $\pi_u(u_t|s_t[\omega_t])$, where $\pi_u(u_t|s_t[\omega_t])$ gives the probability distribution of applying each rewriting rule $u \in \mathcal{U}$ to the partial solution. Once the partial solution is updated, we obtain an improved solution s_{t+1} and repeat the process until convergence.

7.6   Related Work

Methods. Using neural network models for combinatorial optimization has been explored in the last few years. A straightforward idea is to construct a solution directly (e.g., with a Seq2Seq model) from the problem specification [253, 26, 174, 138]. However, such approaches might meet with difficulties if the problem has complex configurations, as our evaluation indicates. In contrast, our paper focuses on iterative improvement of a *complete* solution.

Trajectory optimization with local gradient information has been widely studied in robotics with many effective techniques [177, 34, 255, 245, 155, 154]. For discrete problems, it is possible to apply continuous relaxation and apply gradient descent [36]. In contrast, we *learn the gradient* from previous experience to optimize a complete solution, similar to data-driven descent [248] and synthetic gradient [125].

At a high level, our framework is closely connected with the local search pipeline. Specifically, we can leverage our learned RL policy to guide the local search, i.e., to decide which neighbor solution to move to. We will demonstrate that in our evaluated tasks, our approach outperforms several local search algorithms guided by manually designed heuristics, and softwares supporting more advanced local search algorithms, i.e., Z3 [68] and OR-tools [92].

Applications. For expression simplification, some recent work use deep neural networks to discover equivalent expressions [37, 8, 284]. In particular, [37] trains a deep neural network to rewrite algebraic expressions with supervised learning, which requires a collection of ground truth rewriting paths, and may not find novel rewriting routines. We mitigate these limitations using reinforcement learning.

Job scheduling and resource management problems are ubiquitous and fundamental in computer systems. Various work have studied these problems from both theoretical and empirical sides [30, 94, 16, 224, 246, 174, 45]. In particular, a recent line of work studies deep reinforcement learning for job scheduling [174, 45] and vehicle routing problems [142, 186].

Our approach is tested on multiple domains with extensive ablation studies, and could also be extended to other closely related tasks such as code optimization [223, 44], theorem

proving [118, 153, 19, 116], text simplification [63, 199, 88], and classical combinatorial optimization problems beyond routing problems [71, 138, 27, 253, 135], e.g., Vertex Cover Problem [24].

7.7 Discussion

In this work, we propose to formulate optimization as a rewriting problem, and solve the problem by iteratively rewriting an existing solution towards the optimum. We utilize deep reinforcement learning to train our neural rewriter. In our evaluation, we demonstrate the effectiveness of our neural rewriter on multiple domains, where our model outperforms both heuristic-based algorithms and baseline deep neural networks that generate an entire solution directly.

Meanwhile, we observe that since our approach is based on local rewriting, it could become time-consuming when large changes are needed in each iteration of rewriting. In extreme cases where each rewriting step needs to change the global structure, starting from scratch becomes preferrable. We consider improving the efficiency of our rewriting approach and extending it to more complicated scenarios as future work.

Part IV

Neural-Symbolic Reasoning for Language Understanding

Chapter 8

Neural Symbolic Reader for Reading Comprehension

Integrating distributed representations with symbolic operations is essential for reading comprehension requiring complex reasoning, such as counting, sorting and arithmetics, but most existing approaches rely on specialized neural modules and are hard to adapt to multiple domains or multi-step reasoning. In this chapter, we propose the **Ne**ural Symbolic **R**ea**d**er (**NeRd**), which includes a *reader*, e.g., BERT, to encode the passage and question, and a *programmer*, e.g., LSTM, to generate a program for multi-step reasoning. By using operators like span selection, the program can be executed over text to generate the answer. Compared to previous works, NeRd is more *scalable* in two aspects: (1) *domain-agnostic*, i.e., the same neural architecture works for different domains; (2) *compositional*, i.e., complex programs can be generated by compositionally applying the symbolic operators. Furthermore, to overcome the challenge of training NeRd with weak supervision, we apply data augmentation techniques and hard Expectation-Maximization (EM) with thresholding. On DROP, a challenging reading comprehension dataset requiring discrete reasoning, NeRd achieves 1.37%/1.18% absolute gain over the state-of-the-art on Exact-Match/F1 metrics. With the same architecture, NeRd significantly outperforms the baselines on MathQA, a math problem benchmark that requires multiple steps of reasoning, by 25.5% absolute gain on accuracy when trained on all the annotated programs, and more importantly, still beats the baselines even with only 20% of the program annotations [1].

8.1 Introduction

Deep neural networks have achieved remarkable successes in natural language processing recently. In particular, pretrained language models, e.g., BERT [72], have significantly advanced the state-of-the-art in reading comprehension. While neural models have demonstrated performance superior to humans on some benchmarks, e.g., SQuAD [214], so far such progress

---

[1]The material in this chapter is based on Chen et al. [54].

is mostly limited to extractive question answering, in which the answer is a single span from the text. In other words, this type of benchmarks usually test the capability of text pattern matching, but not of reasoning. Some recent datasets, e.g., DROP [83] and MathQA [10], are collected to examine the capability of both language understanding and discrete reasoning, where the direct application of the state-of-the-art pre-trained language models, such as BERT or QANet [280], achieves very low accuracy. This is especially challenging for pure neural network approaches, because discrete operators learned by neural networks, such as addition and sorting, can hardly generalize to inputs of arbitrary size without specialized design [217, 38, 133]. Therefore, integrating neural networks with symbolic reasoning is crucial for solving those new tasks.

The recent progress on neural semantic parsing [128, 160] is sparked to address this problem. However, such success is mainly restricted to question answering with structured data sources, e.g., knowledge graphs [28] or tabular databases [202]. Extending it to reading comprehension by parsing the text into structured representations suffers severely from the cascade errors, i.e., the issues of the structured parsing for data preprocessing account for the poor performance of the learned neural model [83].

A recent line of work [83, 117, 11] extends BERT/QANet to perform reasoning on the DROP dataset. However, they cannot easily scale to multiple domains or multi-step complex reasoning because: (1) they usually rely on handcrafted and specialized modules for each type of questions; (2) they don't support compositional applications of the operators, so it is hard to perform reasoning of more than one step.

In this work, we propose the **Ne**ural Symbolic **R**ea**d**er (NeRd) for reading comprehension, which consists of (1) a *reader* that encodes passages and questions into vector representations; and (2) a *programmer* that generates programs, which are executed to produce answers. The key insights behind NeRd are as follows: (1) by introducing a set of span selection operators, the compositional programs, usually executed against structured data such as databases in semantic parsing, can now be executed over text; (2) the same architecture can be applied to different domains by simply extending the set of symbolic operators.

A main challenge of training NeRd is that it is often expensive to collect program annotations, so the model needs to learn from weak supervision, i.e., with access only to the final answers. This raises two problems for learning: (1) cold start problem. There are no programs available at the beginning of training, so the training cannot proceed. We address this problem through data augmentation that generates noisy training data to bootstrap the training; (2) spurious program problem, where some programs produce the right answer for wrong rationales. We propose an iterative process using hard EM with thresholding, which filters out the spurious programs during training.

In our evaluation, NeRd demonstrates three major advantages over previous methods: (1) better accuracy. It outperforms the previous state-of-the-art on DROP by 1.37%/1.18% on EM/F1, and the baselines on MathQA by a large margin of 25.5% on accuracy if trained with all annotated programs. Notably, it still outperforms the MathQA baselines using only 20% of the program annotations; (2) more scalable (domain-agnostic and compositional). Unlike previous approaches, which rely on specialized modules that do not support compositional

Figure 8.1: Comparison of NeRd with previous approaches for reading comprehension requiring complex reasoning. The components in grey boxes are the neural architectures. Previous works mainly take two approaches: (1) augmenting pre-trained language model such as BERT with specialized modules for each type of questions, which is hard to scale to multiple domains or multi-step complex reasoning; (2) applying neural semantic parser to the structured parses of the passage, which suffers severely from the cascade error. In contrast, the neural architecture of NeRd is domain-agnostic, which includes a *reader*, e.g., BERT, and a *programmer*, e.g., LSTM, to generate compositional programs that are directly executed over the passages.

application of the operators, NeRd can be applied to tasks of different domains, e.g., DROP and MathQA, without changing the architecture, and more complex programs can be simply generated by extending the set of operators and compositionally applying them; (3) better interpretability. It is easier to interpret and verify an answer by inspecting the program that produces it, especially for the questions involving complex reasoning such as counting and sorting.

8.2 Neural Symbolic Reader

In this section, we present the design of NeRd. It consists of a *reader* that encodes the passages and questions into vector representations, and a *programmer* that generates programs in a domain specific language. The overall comparison between NeRd and previous works is visualized in Figure 8.1.

Neural Architecture

We provide an overview of the two components in NeRd, and defer more details to Appendix G.3.

Reader. Given the natural language text including a question and a passage, the reader component encodes each token t_i in the text into an embedding e_i. Note that our framework is agnostic to the architecture choice of the encoder, so any neural module that turns words into vectors is applicable, e.g., BERT [72].

Programmer. The programmer takes the output of the reader as input, and then decodes a program as a sequence of tokens. Again, our model is agnostic to the design of decoder. For simplicity, we use an LSTM [115] decoder with attention [20] over the encoded text, and self-attention [249] over the previously generated tokens.

A major advantage of our architecture is that it is *domain-agnostic*, i.e., the same architecture can be used for different domains. Compared to previous approaches that craft separate specialized modules for each answer type, we use a unified programmer component to generate programs for multi-step reasoning, and we can simply extend the operator set in the domain specific language (see the next section) to adapt to a different domain. See Section 8.4 for a more detailed discussion.

Domain Specific Language

In this section, we introduce our domain specific language (DSL), which is used to interpret the tokens generated by the programmer component as an executable program.

We list the operators in our DSL in Table 8.1. To handle discrete reasoning, the DSL includes operators that perform arithmetics (`DIFF, SUM`), counting (`COUNT`) and sorting (`ARGMAX, ARGMIN, MAX, MIN`). These operators have been used in previous work in semantic parsing over structured data sources such as a knowledge graph or a tabular database.

However, the main challenge of applying such operations for reading comprehension is that the model needs to manipulate unstructured data, i.e., natural language text, and parsing the text into structured representations may introduce a lot of cascade errors. For example, Dua et al. [83] found that their best performing semantic parsing pipeline using SRL [40] can only find the logical forms for 35% of the questions, resulting in poor performance.

To address this issue, a key insight in our DSL design is to introduce the span selection operators, so that all the arithmetics, counting and sorting operators can be applied to text. Specifically, we introduce `PASSAGE_SPAN, QUESTION_SPAN, VALUE, KEY-VALUE` for selecting spans or numbers from the passage and question. For example, `COUNT` can use `PASSAGE_SPAN` to pick out the spans that mention the relevant entities or events, e.g., touchdowns made by a certain person, and then returns the total number; `ARGMAX` relies on applying `KEY-VALUE` to pick out the spans (keys) for relevant mentions and their associated numbers (values), e.g., touchdowns and their lengths, and then returns the key with the highest value, e.g.,

Operator	Arguments	Outputs	Description
PASSAGE_SPAN QUESTION_SPAN	**v0**: the start index. **v1**: the end index.	a span.	Select a span from the passage or question.
VALUE	**v0**: an index.	a number.	Select a number from the passage.
KEY-VALUE (KV)	**v0**: a span. **v1**: a number.	a key-value pair.	Select a key (span) value (number) pair from the passage.
DIFF SUM	**v0**: a number or index. **v1**: a number or index.	a number.	Compute the difference or sum of two numbers.
COUNT	**v**: a set of spans.	a number.	Count the number of given spans.
MAX MIN	**v**: a set of numbers.	a number.	Select the maximum / minimum among the given numbers.
ARGMAX ARGMIN	**v**: a set of key-value pairs.	a span.	Select the key (span) with the highest / lowest value.

Table 8.1: Overview of our domain-specific language. See Table 8.7 for the sample usage.

the player kicking the longest touchdown. More examples can be found in Table 8.7. In summary, the introduction of span selection operators in the DSL enables the application of the discrete reasoning operators to text, and the resulting programs act as executable and interpretable representations of the reasoning process.

As mentioned above, our architecture is domain-agnostic and the only change needed, to apply to a different domain, is to extend the DSL with new operators. For example, MathQA benchmark requires adding more advanced mathematical operations beyond addition and subtraction, which are defined in [10]. We defer the details to Section 8.4.

A major advantage of our DSL is its *compositionality*, i.e., complex programs can be generated by compositionally applying the operators. Previous works [11] only allow applying the operators for one step, which requires them to introduce operators to mimic two-step compositions, e.g., `Merge` (selecting two spans) and `Sum3` (summing up three numbers). However, this would not scale to more steps of reasoning, as the number of required operators will grow exponentially w.r.t the number of steps. In contrast, NeRd can compose different operators to synthesize complex programs for multi-step reasoning. For example, on MathQA, the average number of operations per question is 5, and some programs apply more than 30 operations to compute the final answer.

8.3 Training with Weak Supervision

Although it is relatively easy to collect question-answer pairs, it is often hard and expensive to obtain program annotations that represent the reasoning behind the answers. Thus, how

to train NeRd with only weak supervision becomes a main challenge. In this section, we revisit the cold start and spurious program problems described in Section 8.1, and present our solutions.

Data Augmentation for Cold Start

The cold start problem means that the training cannot get started when there isn't any program available. For example, a question "How many touchdowns did Brady throw" annotated with only an answer "3" cannot be directly used to train our model due to the lack of the target program to optimize on. To obtain program annotations from question-answer pairs, we first follow previous work to find programs for questions answerable by span selection or arithmetic operations via an exhaustive search, and we defer the details to Section 8.4. However, for questions involving counting or sorting operations, the space becomes too large for an exhaustive search, since these operations rely on the span selection as their sub-routines. For example, the number of possible spans in a text with 200 words is in the order of 10^4, and what's more, counting and sorting operators usually include more than one span as their arguments.

   We apply data augmentation to address the search space explosion problem for counting and sorting operations. For counting, we augment the span selection questions by replacing the interrogatives, e.g., "what" and "who", with "how many" when applicable, and adding a call to COUNT over the selected spans in the answer. For example, a question "What areas have a Muslim population of more than 50000 people?" is changed into "How many areas...". For sorting, we extract the key-value pairs by first applying CoreNLP [173] for entity recognition, and then heuristically find an associated number for each entity. If including them as the arguments of any sorting operator yields the correct answer, then such programs are added to the training set. More details can be found in Appendix G.4. Although the programs found for counting and sorting through this data augmentation process is noisy, they help bootstrap the training. Throughout the training, we also use the model to decode programs, and add those leading to correct answers into our training set.

Hard EM with thresholding against Spurious Programs

After collecting a set of programs for each question-answer pair, another obstacle is the spurious program problem, the phenomenon that a wrong program accidentally predicts a right answer. For example, per arithmetic question in DROP, there are on average 9.8 programs that return correct answers, but usually only one of them is semantically correct. To filter out spurious programs, we adopt hard EM [159, 181] due to its simplicity and efficiency. Specifically, this approach uses the current model to select the program with the highest model probability among the ones that return the correct answer, and then maximizes the likelihood of the selected program. In other words, it relies on the neural model itself to filter out spurious programs. This algorithm is usually faster than the marginalized approach [28]

because at most one program per question-answer pair is used to compute the gradient, and the selection process is fast since it only has a forward pass.

Algorithm 3 Hard EM with Thresholding

   Input: question-answer pairs $\{(x_i, y_i)\}_{i=1}^N$, a model p_θ, initial threshold α_0, decay factor γ
   for each (x_i, y_i) **do**
      $Z_i \leftarrow \text{DataAugmentation}(x_i, y_i)$
   end for
   $T \leftarrow 0$
   repeat
      $\alpha \leftarrow \alpha_0 * \gamma^T$
      $\mathcal{D} \leftarrow \emptyset$
      for each (x_i, y_i) **do**
         $z_i^* = \arg\max_k p_\theta(z_i^k|x_i), z_i^k \in Z_i$
         if $p_\theta(z_i^*) > \alpha$ or $T = 0$ and $|Z_i| = 1$ **then**
            $\mathcal{D} \leftarrow \mathcal{D} \cup (x_i, z_i^*)$
         end if
      end for Update θ by maximizing $\sum_{\mathcal{D}} \log p_\theta(z^*|x)$
      $T \leftarrow T + 1$
   until converge or early stop

Hard EM assumes that for any question-answer pair, at least one of the generated programs is correct. However, there exist questions without any semantically correct program found, e.g., when the annotated answer itself is wrong. In this case, when directly applying the hard EM algorithm, even if the model probabilities for all the programs are very small, it will still select a program for training. RL-based approaches such as MAPO [159] avoid this issue by optimizing the expected return, which weighs the gradient by the model probability. Thus, when all the programs of a question-answer pair have very small probabilities, they will be largely ignored during training. We incorporate this intuition into hard EM by introducing a decaying threshold α, so that a program's probability has to be at least α in order to be included for training. Our experiments show that both hard EM and thresholding are crucial for successful training. The pseudo-code of our training procedure is presented in Algorithm 3, and we defer more details to Appendix G.4.

8.4   Evaluation

In this section, we demonstrate the effectiveness of our approach on DROP [83] and MathQA [10], two recent benchmarks that require discrete reasoning over passages.

Datasets

DROP. DROP (Discrete Reasoning Over Paragraphs) [83] is designed to combine the challenges from both reading comprehension and semantic parsing communities. Specifically, the passages are collected from Wikipedia, each having at least twenty numbers. The question-answer pairs are crowdsourced in an adversarial way that they are accepted only when the questions cannot be correctly answered by the BiDAF model [226]. The dataset has 96.6K question-answer pairs from 6.7K passages. Unlike most existing datasets that are solely

Question	Answer
Someone on a skateboard is traveling 8 miles per hour. How many feet does she travel in 5 seconds? (1 mile = 5280 feet)	**Program:** multiply(5,divide(multiply(8,5280),const_3600)) **Result:** 5 * ((8 * 5280) / 3600) = 58.67 ft

Table 8.2: An example in MathQA dataset.

based on the single span selection, the questions in DROP require complex reasoning, such as selecting multiple spans, arithmetic operations over numbers in the passage, counting and sorting, etc., which poses extra challenge for existing models. For example, vanilla BERT only gets around 30% F1 score. Table 8.7 provides some sample questions in DROP, and their corresponding programs in our DSL (Table 8.1).

For evaluation, we use the same metrics in [83]: (1) Exact Match (EM), where the score is 1 if the prediction exactly matches the ground truth, and 0 otherwise; (2) F1 score, which gives partial credits to a prediction that is not exactly the same as the ground truth, but overlaps with it.

MathQA. MathQA [10] is a dataset with 37K question-answer pairs selected from AQuA [163], but it is further annotated with gold programs in their domain-specific language. The passage length in MathQA is 38 on average, much shorter than DROP with 224. However, the questions in MathQA require more complex and advanced mathematical reasoning than DROP. To this aim, they design 58 math operations, which cover various advanced math topics including geometry, physics, probability, etc. Accordingly, we augment our DSL with those operators to support more advanced numerical reasoning. In these annotated programs, the average number of operations per question is 5, and some programs involve more than 30 steps of computation. Table 8.2 shows an example from MathQA.

Note that each question in MathQA is accompanied with 4 options, where 1 of them is the correct answer. However, since we do not have the full knowledge of the operation semantics, we choose a conservative metric to evaluate the accuracy: a predicted program is considered to be correct only if it is exactly the same as the annotated program. Thus, this metric is an under-estimation of the accuracy based on the execution results. Despite that we use a much stricter measurement in our evaluation, we show that NeRd still outperforms the baselines by a large margin.

Implementation Details

DROP. Similar to previous work [83], for span prediction, we perform an exhaustive search to find all mentions of the ground truth spans in the passage, then include all of them as candidate programs. For numerical questions, we perform another exhaustive search over

all expressions applying addition and subtraction over up to 3 numbers. In this way, we are able to find at least one program for over 95% of the training samples with a number as the answer. Our data augmentation approach for counting and sorting questions can be seen in Section 8.3.

MathQA. Besides the setting where all the ground truth programs are provided during training, we also evaluate the weak supervision setting on MathQA. Due to the lack of program executor, we are unable to perform the search similar to what we have done on DROP. To enable the first training iteration of the model, we assume that we have access to the ground truth programs for a small fraction of training samples at the beginning, and only know the final answer for the rest of training samples. In the first training iteration, the model only trains on the samples annotated with programs. In each of the following iterations, we first run a beam search with a beam size 64 to generate programs for each training sample that has not been annotated in previous iterations, and add the generated program only if it is exactly the same as the ground truth annotation.

For a fair comparison, our reader uses the same pre-trained model as [117, 11], i.e., BERT$_{\text{LARGE}}$. For both benchmarks, we perform greedy decoding during the evaluation.

Baselines

DROP. We evaluate NeRd against three types of baselines: (1) previous models on DROP; (2) NeRd with and without counting and sorting operations; (3) NeRd with different training algorithms, and we discuss the details below.

Previous approaches. We compare with NAQANet [83], NABERT [117], MTMSN [117], and BERT-Calc [11]. We have discussed the key differences between NeRd and BERT-Calc, the baseline with the best performance, in Section 8.2. On the other hand, NAQANet, NABERT, MTMSN share the same overall framework, where they augment an existing model to include individual modules for span selection, numerical expression generation, counting, negation, etc. While NAQANet is based on QANet, other baselines as well as NeRd are based on BERT. Note that the span selection modules themselves are not able to handle questions that return multiple spans as the answer, which causes the exact match accuracy to be zero on multiple-span selection questions for both NAQANet and NABERT. To tackle this issue, MTMSN adapts the non-maximum suppression algorithm [219] to select multiple spans from the candidates with the top prediction probabilities.

Operator variants of NeRd. To show that NeRd learns to apply counting and sorting operations appropriately, we also evaluate the following two variants: (1) *NeRd without counting*: we remove the `COUNT` operation in Table 8.1, and introduce 10 operations `COUNT_0`, `COUNT_1`, ..., `COUNT_9`, where the execution engine returns the number x for operation `COUNT_X`. This counting process is the same as [11]. (2) *NeRd without sorting*: we remove `ARGMAX`, `ARGMIN`, `MAX` and `MIN` operations, so that the model needs to use span selection operations for sorting questions.

	Overall Dev		Overall Test		Number (62%)		Span (32%)		Spans (4.4%)		Date (1.6%)	
	EM	F1	EM	F1	EM	F1	EM	F1	EM	F1	EM	F1
NAQANet	46.75	50.39	44.24	47.77	44.9	45.0	58.2	64.8	0.0	27.3	32.0	39.6
NABERT$_{\text{LARGE}}$	64.61	67.35	−	−	63.8	64.0	75.9	80.6	0.0	22.7	55.7	60.8
MTMSN$_{\text{LARGE}}$	76.68	80.54	75.85	79.85	80.9	81.1	77.5	82.8	25.1	62.8	55.7	69.0
BERT-Calc	78.09	81.65	76.96	80.53	82.0	82.1	78.8	83.4	5.1	45.0	58.1	61.8
N$^{\text{eRd}}$	**78.55**	**81.85**	**78.33**	**81.71**	**82.4**	**82.6**	76.2	81.8	**51.3**	**77.6**	**58.3**	67.2
	± 0.27	± 0.20			± 0.3	± 0.2	± 0.4	± 0.2	± 0.8	± 1.2	± 1.8	± 1.7

Table 8.3: Results on DROP dataset. On the development set, we present the mean and standard error of 10 NeRd models, and the test result of a single model. For all models, the performance breakdown of different question types is on the development set. Note that the training data of BERT-Calc model [11] for test set evaluation is augmented with CoQA [216].

Training variants of NeRd. To show the effectiveness of our training algorithm, we compare with the following baselines: (1) *Hard EM* described in Section 8.3; and (2) *Maximum Likelihood*, which maximizes the likelihood of each program that returns the correct answer for a training sample.

MathQA. We compare with Seq2prog and Seq2prog+cat models in [10], which are LSTM-based encoder-decoder architectures implemented in OpenNMT [140]. In particular, Seq2prog+cat extracts the category label of each question, then trains separate LSTMs to handle different categories, which improves the accuracy by 2.3%.

Results

DROP. Table 8.3 summarizes our main evaluation results on DROP dataset, with 9.5K samples in the development set and 9.6K hidden samples in the test set. Note that NABERT$_{\text{LARGE}}$ was not evaluated on the test set [117]. Specifically, we train 10 NeRd models with the best configuration from different random initialization, present the mean and standard error of the results on the development set, and submit a single model to obtain the result on the hidden test set. We can observe that on test set, NeRd outperforms previous models by 1.37% on exact match, and 1.18% on F1 score. Notice that in [11], they train their BERT-Calc model on CoQA [216] in addition to DROP, and they also evaluate an ensemble with 6 models, resulting in the exact match of 78.14, and F1 score of 81.78 on test set. However, we can see that without additional training data and ensembling, NeRd still beats their single model, and the performance is on par with their ensemble model.

To understand the strengths of NeRd, we first show examples of correct predictions in Table 8.7. We can observe that NeRd is able to compose multiple operations so as to obtain the correct answer, which helps boost the performance. In particular, for questions that require the selection of multiple spans, the exact match accuracy of NeRd is more than double of the best previous approach that specially designed for multi-span prediction, and the F1

	with Count Op	w/o Count op
EM	**73.1**	71.2
F1	**73.1**	71.2

(a)

	with Sort Ops	w/o Sort Ops
EM	**83.9**	82.1
F1	**86.8**	85.5

(b)

Table 8.4: Results of counting and sorting questions on DROP development set, where we compare variants of NeRd with and without the corresponding operations. **(a)**: counting; **(b)**: sorting. For each setting, we present the best results on development set.

	EM	F1
Hard EM with thresholding	**80.58**	**83.42**
Hard EM	73.72	77.46
Maximum Likelihood	63.96	67.98

Table 8.5: Results of different training algorithms on DROP development set. For each setting, we present the best results on the development set.

score also improves around 15%. Meanwhile, NeRd is able to generate more complicated arithmetic expressions than [11], thanks to the compositionality of our approach.

We further present our ablation studies of counting and sorting operations in Tables 8.4 and 8.8. Specifically, we evaluate on two subsets of DROP development set that include counting and sorting questions only, using the variants of NeRd with and without the corresponding operations. We can observe that adding these advanced operations can not only boost the performance, but also enable the model to provide the rationale behind its predictions. For counting problems, NeRd is able to select the spans related to the question. For sorting problems, NeRd first associates the entities with their corresponding values to compose the key-value pairs, then picks the most relevant ones for prediction. None of the previous models is able to demonstrate such reasoning processes, which suggests better interpretability of NeRd.

Finally, we present the results of different training algorithms in Table 8.5. First, we observe that by filtering spurious programs, the hard EM significantly boosts the performance of the maximum likelihood training for 10%, which may be due to the fact that the exhaustive search finds plenty of spurious programs that yield the correct answer. Adding the threshold for program selection provides further improvement of about 7%, indicating that our training algorithm can better handle the issue of spurious programs and be more tolerant to the noise of answer annotations. In Appendix G.5, we show some examples discarded by NeRd using the threshold, which mostly have the wrong answer annotations, e.g., incorrect numerical operations or missing part of the information in the question.

	Accuracy
Seq2prog	51.9
Seq2prog+cat	54.2
NeRd	**79.7**
NeRd (-pretraining)	71.6
NeRd (20%)	56.5

Table 8.6: Results on MathQA test set, with NeRd and two variants: (1) no pre-training; (2) using 20% of the program annotations in training.

MathQA. We present the results on MathQA test set with around 3K samples in Table 8.6. NeRd dramatically boosts the accuracy of the baselines by 25.5%. In addition, we also evaluate a variant of NeRd with the same model architecture, but the BERT encoder is not pre-trained and is randomly initialized. We observe that this variant still yields a performance gain of 17.4%. Note that NeRd is measured by the program accuracy, which is a much stricter criterion and thus is an underestimation of the execution accuracy computed in [10]. Moreover, even with only 20% training data labeled with ground truth programs, NeRd still outperforms the baseline.

8.5 Related Work

Reading comprehension and question answering have recently attracted a lot of attention from the NLP community. A plethora of datasets have been available to evaluate different capabilities of the models, such as SQuAD [214], CoQA [216], GLUE [256], etc. A bunch of representative models are proposed for these benchmarks, including BiDAF [226], r-net [263], DrQA [42], DCN [270] and QANet [280]. More recently, massive text pre-training techniques, e.g., ELMo [206], BERT [72], XLNet [275] and Roberta [166], have achieved superior performance on these tasks. However, for more complicated tasks that require logical reasoning, pre-trained models alone are insufficient.

On the other hand, semantic parsing has recently seen a lot of progress from the neural symbolic approaches. One line of work applied neural sequence-to-sequence and sequence-to-tree models to semantic parsing with full supervision [128, 80, 292] . Some other work have advanced the state-of-the-art in weakly supervised semantic parsing on knowledge graphs and tabular databases [160, 187, 143, 103, 159]. However, most of the successes of semantic parsing are limited to structured data sources. In contrast, our work naturally extends the complex reasoning in semantic parsing to reading comprehension by introducing the span selection operators. Several methods for training with weak supervision have been proposed in the context of weakly supervised semantic parsing including Maximum Marginal Likelihood [28, 143, 66, 103], RL [160, 159] and Hard EM [160, 181]. Our approach is based on Hard EM

due to its simplicity and efficiency, and extends it by adding a decaying threshold, which improves its robustness against spurious programs.

In the broader context, neural symbolic approaches have been applied to Visual Question Answering [13, 175, 130], where the neural architecture is composed with sub-modules based on the structured parses of the questions. Another line of work studied neural symbolic approaches to learn the execution of symbolic operations such as addition and sorting [95, 217, 38, 78]. In this work, we study neural symbolic approaches for reading comprehension tasks that require discrete reasoning over the text [83, 117, 11, 10].

8.6   Discussion

We presented the Neural Symbolic Reader (NeRd) as a scalable integration of distributed representations and symbolic operations for reading comprehension. NeRd architecture consists of a reader that encodes text into vector representation, and a programmer that generates programs, which will be executed to produce the answer. By introducing the span selection operators, our *domain-agnostic* architecture can generate *compositional* programs to perform complex reasoning over text for different domains by only extending the set of operators. We also overcome the challenge of weak supervision by applying data augmentation techniques and hard EM with thresholding. In our evaluation, using the same model architecture without any change, NeRd significantly surpasses previous state-of-the-arts on two challenging reading comprehension tasks, DROP and MathQA. We hope to motivate future works to introduce complex reasoning to other domains or other tasks in NLP, e.g., machine translation and language modeling, by extending the set of operators.

Passage	Question & Answer
	Multiple spans
...the population was spread out with **26.20% under the age of 18**, 9.30% from 18 to 24, **26.50% from 25 to 44**, **23.50% from 45 to 64**, and 14.60% who were 65 years of age or older...	**Question:** Which groups in percent are larger than 16%? **Program:** PASSAGE_SPAN(26,30), PASSAGE_SPAN(46,48), PASSAGE_SPAN(55,57) **Result:** 'under the age of 18', '25 to 44', '45 to 64'
	Date
When major general Nathanael Greene took command in the south, Marion and lieutenant colonel Henry Lee were ordered in January **1781**... On **August 31**, Marion rescued a small American force trapped by 500 British soldiers...	**Question:** When did Marion rescue the American force? **Program:** PASSAGE_SPAN(71,71), PASSAGE_SPAN(72,72), PASSAGE_SPAN(32,32) **Result:** 'August', '31', '1781'
	Numerical operations
...Lassen county had a population of **34,895**. The racial makeup of Lassen county was **25,532 (73.2%) white** (U.S. census), **2,834 (8.1%) African American** (U.S. census)...	**Question:** How many people were not either solely white or solely African American? **Program:** DIFF(9,SUM(10,12)) **Result:** 34895 - (25532 + 2834) = 6529
	Counting
...the Bolshevik party came to power in November 1917 through the **simultaneous election in the soviets** and an **organized uprising supported by military mutiny**...	**Question:** How many factors were involved in bringing the Bolsheviks to power? **Program:** COUNT(PASSAGE_SPAN(62, 66), PASSAGE_SPAN(69, 74)) **Result:** COUNT('simultaneous election in the soviets', 'organized uprising supported by military mutiny') = 2
	Sorting
...Jaguars kicker **Josh Scobee** managed to get a 48-yard field goal...with kicker Nate Kaeding getting a 23-yard field goal...	**Question:** Who kicked the longest field goal? **Program:** ARGMAX(KV(PASSAGE_SPAN(50,53),VALUE(9)), KV(PASSAGE_SPAN(92,94),VALUE(11))) **Result:** ARGMAX(KV('Josh Scobee', 48), KV('Nate Kaeding', 23)) = 'Josh Scobee'
...Leftwich flipped a **1-yard touchdown pass** to Wrighster...Leftwich threw a 16- yard touchdown pass to Williams for a 38-0 lead...	**Question:** How many yards was the shortest touchdown pass? **Program:** MIN(VALUE(17), VALUE(19)) **Result:** MIN(1, 16) = 1

Table 8.7: Examples of correct predictions on DROP development set.

Passage	Question & Prediction
...with field goals of **38** and **36 yards** by kicker Dan Carpenter ... followed by a **43-yard** field goal by Carpenter ... **52-yard** field goal ...	**Question:** How many total field goals were kicked in the game?

	Predicted Program:
	COUNT(
	    PASSAGE_SPAN(75,75), PASSAGE_SPAN(77,78),
	    PASSAGE_SPAN(133,135), PASSAGE_SPAN(315,317))
	Result: COUNT('38','36 yards', '43-yard','52-yard') = 4

	Predicted Program (-counting): COUNT5 **Result:** 5
... with the five most common surgeries being breast augmentation, **liposuction**, breast reduction, **eyelid surgery** and **abdominoplasty** ...	**Question:** How many of the five most common procedures are not done on the breasts?
	Predicted Program: COUNT(PASSAGE_SPAN(132,135), PASSAGE_SPAN(140,142), PASSAGE_SPAN(144,149)) **Result:** COUNT('liposuction', 'eyelid surgery', 'abdominoplasty') = 3
	Predicted Program (-counting): COUNT4 **Result:** 4

(a)

Passage	Question & Prediction
...In the third quarter, Arizona's deficit continued to climb as Cassel completed a 76-yard touchdown pass to wide receiver Randy Moss ... quarterback **Matt Leinart** completed a 78-yard touchdown pass to wide receiver Larry Fitzgerald ...	**Question:** Who threw the longest touchdown pass?

	Predicted Program:
	ARGMAX(
	    KV(PASSAGE_SPAN(205,208),VALUE(18)),
	    KV(PASSAGE_SPAN(142,143), VALUE(14)))
	Result: ARGMAX(KV('Matt Leinart', 78),KV('Cassel', 76)) = 'Matt Leinart'

	Predicted Program (-sorting): PASSAGE_SPAN(82,84) **Result:** Matt Cassel
... Carney got a 38-yard field goal ... with Carney connecting on a **39-yard field goal** ...	**Question:** How many yards was the longest field goal?
	Predicted Program: MAX(VALUE(14),VALUE(11))
	Result: MAX(39, 38) = 39
	Predicted Program (-sorting): VALUE(11) **Result:** 38

(b)

Table 8.8: Examples of counting and sorting questions on DROP development set, where NeRd with the corresponding operations gives the correct predictions, while the variants without them do not. **(a)**: counting; **(b)**: sorting.

Chapter 9

Compositional Generalization via Neural-Symbolic Stack Machines

Despite achieving tremendous success, existing deep learning models have exposed limitations in compositional generalization, the capability to learn compositional rules and apply them to unseen cases in a systematic manner. To tackle this issue, we propose the Neural-Symbolic Stack Machine (NeSS). It contains a neural network to generate traces, which are then executed by a symbolic stack machine enhanced with sequence manipulation operations. NeSS combines the expressive power of neural sequence models with the recursion supported by the symbolic stack machine. Without training supervision on execution traces, NeSS achieves 100% generalization performance in four domains: the SCAN benchmark of language-driven navigation tasks, the task of few-shot learning of compositional instructions, the compositional machine translation benchmark, and context-free grammar parsing tasks [1].

9.1 Introduction

Humans have an exceptional capability of compositional reasoning. Given a set of basic components and a few demonstrations of their combinations, a person could effectively capture the underlying compositional rules, and generalize the knowledge to novel combinations [61, 183, 151, 150]. In contrast, deep neural networks, including the state-of-the-art models for natural language understanding, typically lack such generalization abilities [148, 137, 168], although they have demonstrated impressive performance on various applications.

To evaluate the compositional generalization, [148] proposes the SCAN benchmark for natural language to action sequence generation. When SCAN is randomly split into training and testing sets, neural sequence models [20, 115] can achieve perfect generalization. However, when SCAN is split such that the testing set contains unseen combinations of components in the training set, the test accuracies of these models drop dramatically, though the training accuracies are still nearly 100%. Some techniques have been proposed to improve the

---

[1]The material in this chapter is based on Chen et al. [52].

performance on SCAN, but they either still fail to generalize on some splits [221, 149, 158, 93, 12], or are specialized for SCAN-like grammar learning [194].

In this paper, we propose the **Ne**ural-**S**ymbolic **S**tack machine (NeSS), which integrates a symbolic stack machine into a sequence-to-sequence generation framework, and learns a neural network as the controller to operate the machine. NeSS preserves the capacity of existing neural models for sequence generation; meanwhile, the symbolic stack machine supports recursion [38, 48], so it can break down the entire sequence into components, process them separately and then combine the results, encouraging the model to learn the primitives and composition rules. In addition, we propose the notion of *operational equivalence*, which formalizes the intuition that semantically similar sentences often imply similar operations executed by the symbolic stack machine. It enables NeSS to categorize components based on their semantic similarities, which further improves the generalization. To train our model without the ground truth execution traces, we design a curriculum learning scheme, which enables the model to find correct execution traces for long training samples.

We evaluate NeSS on four benchmarks that require compositional generalization: (1) the SCAN benchmark discussed above; (2) the task of few-shot learning of compositional instructions [150]; (3) the compositional machine translation task [148]; and (4) the context-free grammar parsing tasks [48]. NeSS achieves 100% generalization performance on all these benchmarks.

9.2 Neural-Symbolic Stack Machine (NeSS)

In this section, we demonstrate NeSS, which includes a symbolic stack machine enhanced with sequence manipulation operations, and a neural network as the machine controller that produces a trace to be executed by the machine. We present our stack machine in Section 9.2, describe the model architecture of our machine controller in Section 9.2, and discuss the expressiveness and generalization power of NeSS in Section 9.2.

Enhanced Stack Machine for Sequence Manipulation

We design a stack machine that supports recursion, a key component to achieving compositional generalization. In particular, this machine supports general-purpose sequence-to-sequence tasks, where an input sequence in a source language is mapped into an output sequence in a target language. We present an overview of the machine operations in Table 9.1. Specifically, `SHIFT` is used for reading the input sequence, `PUSH` and `POP` are standard stack operations, `REDUCE` is used for output sequence generation, `CONCAT_M` and `CONCAT_S` concatenate the generated sequences to form a longer one, and the `FINAL` operation terminates the machine operation and produces the output. We provide an illustrative example in Figure 9.1, and defer more descriptions to the supplementary.

Table 9.1: Instruction semantics of our stack machine. See Figure 9.1 for the sample usage.

Operator	Arguments	Description
SHIFT	–	Pull one token from the input stream to append to the end of the stack top.
REDUCE	$[t_1, t_2, ..., t_l]$	Reduce the stack's top to a sequence $[t_1, t_2, ..., t_l]$ in the target language.
PUSH	–	Push a new frame to the stack top.
POP	–	Pop the stack top and append the popped data to the new stack top.
CONCAT_M	$[i_1, i_2, ..., i_l]$	Concatenate the items from the stack top and the memory with indices $i_1, i_2, ..., i_l$, then put the concatenated sequence in the memory.
CONCAT_S	$[i_1, i_2, ..., i_l]$	Concatenate the items from the stack top and the memory with indices $i_1, i_2, ..., i_l$, then put the concatenated sequence in the stack top.
FINAL	–	Terminate the execution, and return the stack top as the output.

Figure 9.1: An illustrative example of how to use the stack machine for SCAN benchmark. A more complex example can be found in the supplementary material.

Operational Equivalence

Inspired by Combinatory Categorial Grammar (CCG) [236], we use component categorization as an ingredient for compositional generalization. As shown in Figure 9.2, from the perspective of categorical grammars, categories for source language may be considered as the primitive types of the lexicon, while predicting categories for the target language may be considered

run	and	walk	twice
Cs_1: RUN	Cs_2: concat(0, 2)	Cs_1: WALK	Cs_3: repeat(0, 0)

Ct_1: WALK WALK

Ct_1: RUN WALK WALK

jump	and	walk	thrice
Cs_1: JUMP	Cs_2: concat(0, 2)	Cs_1: WALK	Cs_3: repeat(0, 0, 0)

Ct_1: WALK WALK WALK

Ct_1: JUMP WALK WALK WALK

Figure 9.2: An illustration of component categorization, where Cs_i and Ct_i denote the i-th category of source and target languages respectively.

as the type inference. By mapping "jump" and "run" into the same category, we can easily infer the meaning of "run and walk" after learning to "jump and walk". Meanwhile, mapping "twice" and "thrice" into the same category indicates the similarity of their combinatorial rules, e.g., both of them should be processed before parsing the word "and". From the perspective of parsing, categorical information is encoded in non-terminals of the (latent) parse tree, which provides higher-level abstraction of the terminal tokens' semantic meaning. However, annotations of tree structures are typically unavailable or expensive to obtain. Faced with this challenge similar to unsupervised parsing and grammar induction [14, 67, 31], we leverage the similarities between the execution traces to induce the latent categorical information. This intuition is formalized as *operational equivalence* below.

Operational Equivalence (OE). Let $\mathcal{L}_s$, $\mathcal{L}_t$ be the source and target languages, π be a one-to-one mapping from $\mathcal{L}_s$ to $\mathcal{L}_t$; $Op_\pi(s)$ be the operator to perform the mapping π, given the current machine status s; $\mathcal{S}$ be the set of valid machine statuses; $s' = R(s, s_i, s'_i)$ means replacing the occurrences of s_i in s with s'_i. Components s_i and s'_i are operationally equivalent if and only if $\forall s \in \mathcal{S}$, $s' = R(s, s_i, s'_i) \in \mathcal{S}$ and $Op_\pi(s) = Op_\pi(s')$.

In Figure 9.3, we present some examples of operational equivalence captured by the execution traces. We observe that, when two sequences only differ in the arguments of REDUCE, their corresponding tokens could be mapped to the same category, which is the main focus of most prior work on compositional generalization [93, 158, 149]. For example, [93] proposes the notation of *local equivariance* to capture such information. On the other hand, by grouping sequences only differing in CONCAT_M and CONCAT_S arguments, we also allow the model to capture structural equivalence, as shown in Figure 9.3b, which is the key to enabling generalization beyond primitive substitutions.

Neural Controller

With the incorporation of a symbolic machine into our sequence-to-sequence generation framework, NeSS does not directly generate the entire output sequence. Instead, the neural network in NeSS acts as a controller to operate the machine. The machine runs the execution trace generated by the neural network to produce the output sequence. Meanwhile, the design of our machine allows the neural controller to make predictions based on the local context of the input, which is a key factor to achieving compositional generalization. We provide an overview of the neural controller in Figure 9.4, describe the key components below, and defer more details to the supplementary.

Figure 9.3: An illustration of the operational equivalence captured by the execution traces on SCAN benchmark. (a) With primitive replacement, e.g., changing "walk" into "jump", the operator trace remains the same, while the REDUCE arguments differ, thus "walk" and "jump" can be grouped into the same category. Such equivalence is also characterized by local equivariance defined in [93]. (b) By changing "twice" into "thrice", the operator trace remains the same, while the CONCAT_M and CONCAT_S arguments could differ, thus "twice" and "thrice" are in the same category. Such equivalence is crucial in achieving length generalization on SCAN, which is not characterized by primitive equivariance studied in prior work [93, 158, 149]

Machine status encoder. A key property of NeSS is that it enables the neural controller to focus on the local context that is relevant to the prediction, thanks to the recursion supported by the stack machine. Specifically, the input to the neural controller consists of three parts: (1) the next token in the input queue *tok*, e.g., the token "around" before executing step 4 in Figure 9.1; (2) the top 2 frames of the stack; and (3) the memory. Note that including the second stack frame from the top is necessary for determining the association rules of tokens, as discussed in [48]. Take arithmetic expression parsing as an example, when the top stack frame includes a variable "y" and the next input token is "*", we continue processing this token when the previous stack frame includes a "+", while we need a POP operation when the previous stack frame includes "*". We use 4 bi-directional LSTMs as the machine status encoders to encode the input sequence, the top 2 stack frames and the memory respectively. Then we denote the 4 computed embedding vectors as e_{tok}, e_{cur}, e_{pre} and e_M.

Figure 9.4: An overview of the neural architecture for the machine controller. A more detailed illustration is included in the supplementary material.

Operator predictor. The operator predictor is a multi-layer fully connected network with a $|Op|$-dimensional softmax output layer, where $|Op| = 7$ is the number of operators supported by the machine, as listed in Table 9.1. Its input is the concatenation of e_{tok}, e_{cur}, e_{pre} and e_M.

Argument predictors. We include three neural modules for argument prediction, which are used for REDUCE, CONCAT_M and CONCAT_S respectively. We design the REDUCE argument predictor as a standard LSTM-based sequence-to-sequence model with attention, which generates a token sequence in the target vocabulary as the arguments. For both CONCAT_M and CONCAT_S argument predictors, we utilize the pointer network architecture [253] to select indices from an element list as arguments, where the list contains the elements from the top stack frame and the memory.

Latent category predictors. We introduce two latent category predictors for source and target languages. The source category predictor, denoted as $p_{sc}(e_{tok})$, computes an embedding vector ec_{tok} to indicate the categorical information given the input e_{tok}. Similarly, we denote the target category predictor as $p_{tc}(e_s)$, where the input e_s is the embedding of the token sequence s in the target language. For the input to the operator predictor, we replace e_{tok} with ec_{tok} as the representation of the next input token tok, which encourages the neural controller to predict the same operator for tokens of the same category. Similarly, the categorical predictions for the target language are used for subsequent instruction predictions.

Discussion

In the following, we discuss the expressiveness and generalization power of NeSS. In particular, NeSS preserves the same expressive power as sequence-to-sequence models, while our neural-symbolic design enhances its compositional generalization capability.

Expressive power. When we impose no constraints to regularize the machine, e.g., we do not restrict the argument length for REDUCE instruction, there is a *degenerate* execution trace that is valid for every input-output example. Specifically, this trace keeps running the

`SHIFT` instruction until an `[EOS]` is observed as the next input token, then executes a `REDUCE` instruction with the entire output sequence as its argument. In this way, NeSS preserves the capacity of existing sequence-to-sequence models [20, 266, 115, 249, 72]. To leverage the recursion property of the machine, we could set a length limit for `REDUCE` arguments, so that the neural model mainly calls `REDUCE` instruction to generate phrases in the target language, and utilizes other instructions to combine the generated primitives to form a longer sequence. We call such compositional traces as *non-degenerate* traces hereafter.

Generalization. Some recent work proposes neural-symbolic architectures to achieve length generalization for program induction [38, 48, 268, 207]. The core idea is to incorporate a stack into the symbolic machine, so that the neural network model could restrict its attention to only part of the input important for the current decision. This recursion design principle is also crucial in achieving compositional generalization in our case. Meanwhile, capturing the operational equivalence enables NeSS to simultaneously obtain generalization capability and expressive power.

9.3 Training

As discussed in Section 9.2, without the ground truth execution traces as training supervision, the model may exploit the `REDUCE` argument generator and predict degenerate traces without compositionality. To avoid this degenerate solution, we apply a curriculum learning scheme. At a high level, the model first performs a trace search for each sample in the lesson, and then categorizes components based on the operational equivalence. We discuss the key sub-routines below. The full training procedure can be found in the supplementary material.

Curriculum learning. We sort the training samples in the increasing order of their input and output lengths to construct the curriculum. Before training on the first lesson, the neural model is randomly initialized. Afterwards, for each new lesson, the model is initialized with the parameters learned from previous lessons. To obtain training supervision, for each input sequence within the current lesson, we search for an execution trace that leads to the ground truth output, based on the probability distribution predicted by the neural model, and we prioritize the search for non-degenerate execution traces. If the model could not find any non-degenerate execution trace within the search budget, a degenerate execution trace is used for training. The model proceeds to the new lesson when no more non-degenerate execution traces can be found.

Learning to categorize. To provide training supervision for latent category predictors, we leverage the operational equivalence defined in Section 9.2. Specifically, after the trace search, we collect the non-degenerate operator traces, and compare among different samples within a training batch. If two samples share the same operator trace, we first assign their target sequences with the same target category for training. Note that their input sequences have the same length, because the same `SHIFT` instructions are executed. Therefore, we

enumerate each index within the input length, pair up the tokens with the same index, and assign them with the same source category for training.

9.4 Experiments

We evaluate NeSS in four domains: (1) the SCAN splits that require compositional generalization [148]; (2) the task of few-shot learning of compositional instructions proposed in [150]; (3) the compositional machine translation benchmark proposed in [148]; and (4) the context-free grammar parsing benchmarks proposed in [48]. We present the setup and key results below, and defer more experimental details to the supplementary material. Note that we perform greedy decoding to generate the execution traces during the inference time, without any search.

SCAN Benchmark

The SCAN benchmark has been widely used to evaluate the compositional generalization of neural networks, where the input sequence is a navigation command in English, and the output is the corresponding action sequence [148]. See Figure 9.1 for a sample usage of NeSS for the SCAN tasks.

Evaluation setup. Similar to prior work [149, 93, 194], we evaluate the following four settings. (1) **Length generalization**: the output sequences in the training set include at most 22 actions, while the output lengths in the test set are between 24 and 48. (2) **Template generalization for "around right"**: the phrase "around right" is held out from the training set; however, both "around" and "right" occurs in the training set separately. For example, the phrases "around left" and "opposite right" are included in the training set. (3) **Primitive generalization for "jump"**: all commands not including "jump" are included in the training set, but the primitive "jump" only appears as a single command in the training set. The test set includes commands combining "jump" with other primitives and templates, such as "jump twice" and "jump after walk". (4) **Simple split**: randomly split samples into training and test sets. In this case, no compositional generalization is required.

Previous approaches. We compare NeSS with two classes of existing approaches on SCAN benchmark. The first class of approaches propose different neural network architectures, without additional data to provide training supervision. Specifically, sequence-to-sequence models (seq2seq) [148] and convolutional neural networks (CNN) [70] are standard neural network architectures, Stack LSTM learns an LSTM to operate a differentiable stack [96], while the equivariant sequence-to-sequence model [93] incorporates convolution operations into the recurrent neural networks, so as to achieve local equivariance discussed in Section 9.2. On the other hand, the syntactic attention model [221] and primitive substitution [158] learn two attention maps for primitives and templates separately. The second class of approaches design different schemes to generate auxiliary training data. Specifically, GECA [12] performs data

augmentation by replacing fragments of training samples with different fragments from other similar samples, while the meta sequence-to-sequence model [149] and the rule synthesizer model (synth) [194] are trained with samples drawn from a meta-grammar with the format close to the SCAN grammar.

Table 9.2: Learned categories on SCAN. The words in a pair of brackets belong to the same category. The categories contained in the three lines are respectively learned from input sequences of length 1, 2 and 3.

{run, look, jump, look}
{left, right}, {twice, thrice}, {turn}
{and, after}, {around}, {opposite}

Results. Table 9.3 summarizes our results on SCAN tasks. In the top block, we compare with models trained without additional data. Among these approaches, NeSS is the only one achieving 100% test accuracies on tasks that require compositional generalization, and the performance is consistent among 5 independent runs. In particular, the best generalization accuracy on the length split is only around 20% for the baseline models. Note that the stack LSTM does not achieve better results, demonstrating that without a symbolic stack machine that supports recursion and sequence manipulation, augmenting neural networks with a stack alone is not sufficient. Meanwhile, without category predictors, NeSS still achieves 100% test accuracy in 2 runs, but the accuracy drops to around 20% for other 3 runs. A main reason is that existing models could not generalize to the input template "around left/right thrice", when the training set only includes the template "around left/right twice". Although NeSS correctly learns the parsing rules for different words, without category predictors, NeSS still may not learn that the parsing rule for "thrice" has the same priority as "twice". For example, in the test set, there is a new pattern "jump around right thrice". The correct translation is to parse "jump around right" first, then repeat the action sequence thrice, resulting in 24 actions. Without category prediction, NeSS could mistakenly parse "right thrice" first, concatenate the action sequences of "jump" and "right thrice", then repeat it for four times, resulting in 16 actions. Such a model could still achieve 100% training accuracy, because this pattern does not occur in the training set, but the test accuracy drops dramatically due to the wrong order for applying rules. Therefore, to achieve generalization, besides the parsing rules for each individual word, the model also needs to understand the order of applying different rules, which is not captured by the primitive equivalence [149, 158, 221] or local equivariance [93] studied in prior work. On the other hand, as shown in Table 9.2, the operational equivalence defined in Section 9.2 enables the model to learn the priorities of different parsing rules, e.g., categorizing "twice" and "thrice" together, which is crucial in achieving length generalization.

Next, we compare NeSS with models trained with additional data. In particular, the meta sequence-to-sequence model is trained with different permutations of primitive assignment, i.e., different one-to-one mapping of {run, look, jump, walk} to {RUN, LOOK, JUMP, WALK}, denoted as "(perm)". We consider two evaluation modes of the rule synthesizer model (synth) [194], where the first variant performs greedy decoding, denoted as "(no search)"; the second one performs a search process, where the model samples candidate

Table 9.3: Test accuracy on SCAN splits. All models in the top block are trained without additional data. In the bottom, GECA is trained with data augmentation, while Meta Seq2seq (perm) and both variants of Synth are trained with samples drawn from a meta-grammar, with a format close to the SCAN grammar. In particular, Synth (with search) performs a search procedure to sample candidate grammars, and returns the one that matches the training samples; instead, other models always return the prediction with the highest decoding probability.

Approach	Length	Around right	Jump	Simple
NeSS (ours)	**100.0**	**100.0**	**100.0**	**100.0**
Seq2seq [148]	13.8	—	0.08	99.8
CNN [70]	0.0	56.7	69.2	100.0
Stack LSTM [96]	17.0	0.3	0.0	100.0
Syntactic Attention [221]	15.2	28.9	91.0	—
Primitive Substitution [158]	20.3	83.2	98.8	99.9
Equivariant Seq2seq [93]	15.9	92.0	99.1	100.0
GECA [12]	—	82	87	—
Meta Seq2seq (perm) [149]	16.64	98.71	99.95	—
Synth (no search) [194]	0.0	0.0	3.5	13.3
Synth (with search) [194]	100.0	100.0	100.0	100.0

grammars, and returns the one that matches the training samples. We observe that even with additional training supervision, Synth (with search) is the only baseline approach that is able to achieve 100% generalization on all these SCAN splits.

Although both Synth and NeSS achieve perfect generalization, there are key differences we would like to highlight. First, the meta-grammar designed in Synth restricts the search space to only include grammars with a similar format to the SCAN grammar [194]. For example, each grammar has between 4 and 9 primitive rules that map a single word to a single primitive (e.g., run → RUN), and 3 to 7 higher order rules that encode variable transformations given by a single word (e.g., x1 and x2 → [x1] [x2]). Therefore, Synth cannot be applied to other two benchmarks in our evaluation. Unlike Synth, NeSS does not make such assumptions about the number of rules nor their formats. Also, NeSS does not perform any search during the inference, while Synth requires a search procedure to ensure that the synthesized grammar satisfies the training samples.

Few-shot Learning of Compositional Instructions

Next, we evaluate on the few-shot learning benchmark proposed in [150], where the model learns to produce abstract outputs (i.e., colored circles) from pseudowords (e.g., "dax"). Compared to the SCAN benchmark, the grammar of this task is simpler, with 4 primitive

Table 9.4: Accuracy on the few-shot learning task proposed in [150].

Approach	Accuracy
NeSS (ours)	**100.0**
Seq2seq [158]	2.5
Primitive Substitution [158]	76.0
Human [150]	84.3

Table 9.5: Accuracy on the compositional machine translation benchmark in [148], measured by semantic equivalence.

Approach	Accuracy
NeSS (ours)	**100.0**
Seq2seq [148]	12.5
Primitive Substitution [158]	100.0

Table 9.6: Results on the context-free grammar parsing benchmarks proposed in [48]. "Test-LEN" indicates the testset including inputs of length LEN.

Test	**NeSS (ours)**	Neural Parser	Seq2seq	Seq2tree	Stack LSTM
Training	**100%**	100%	81.29%	100%	100%
Test-10	**100%**	100%	0%	0.8%	0%
Test-5000	**100%**	100%	0%	0%	0%

rules and 3 compositional rules. However, while the SCAN training set includes over 10K examples, there are only 14 training samples in this benchmark, thus models need to learn the grammar from very few demonstrations. In [150], they demonstrate that humans are generally good at such few-shot learning tasks due to their inductive biases, while existing machine learning models struggle to obtain this capability.

Results. We present the results in Table 9.4, where we compare with the standard sequence-to-sequence model, the primitive substitution approach discussed in Section 9.4, and the human performance evaluated in [150]. We didn't compare with the meta sequence-to-sequence model and the rule synthesizer model discussed in Section 9.4, because they require meta learning with additional training samples. Despite that the number of training samples is very small, NeSS achieves 100% test accuracy in 5 independent runs, demonstrating the benefit of integrating the symbolic stack machine to capture the grammar rules.

Compositional Machine Translation

Then we evaluate on the compositional machine translation benchmark proposed in [148]. Specifically, the training set includes 11,000 English-French sentence pairs, where the English sentences begin with phrases such as "I am", "you are" and "he is", and 1,000 of the samples are repetitions of "I am daxy" and its French translation, which is the only sentence that introduces the pseudoword "daxy" in the training set. The test set includes different combinations of the token "daxy" and other phrases, e.g., "you are daxy", which do not

appear in the training set. Compared to the SCAN benchmark, the translation rules in this task are more complex and ambiguous, which makes it challenging to be fully explained with a rigorous rule set.

Results. We present the results in Table 9.5, where we compare NeSS with the standard sequence-to-sequence model [148], and the primitive substitution approach discussed in Section 9.4. Note that instead of measuring the exact match accuracy, where the prediction is considered correct only when it is exactly the same as ground truth, we measure the semantic equivalence in Table 9.5. As discussed in [158], only one reference translation is provided for each sample in the test set, but there are 2 different French translations of "you are" that appear frequently in the training set, which are both valid translations. Therefore, if we measure the exact match accuracy, the accuracy of the Primitive Substitution approach is 62.5%, while NeSS achieves 100% in 2 runs, and 62.5% in 3 other runs. Although both NeSS and the Primitive Substitution approach achieves 100% generalization, by preserving the sequence generation capability of sequence-to-sequence models with the `REDUCE` argument generator, NeSS is the only approach that simultaneously enables length generalization for rule learning tasks and achieves 100% generalization on machine translation with more diverse rules, by learning the phrase alignment.

Context-free Grammar Parsing

Finally we evaluate NeSS on the context-free grammar parsing tasks in [48]. Following [48], we mainly consider the curriculum learning setting, where we train the model with their designed curriculum, which includes 100 to 150 samples enumerating all constructors in the grammar. NeSS parses the inputs by generating the serialized parse trees, as illustrated in the supplementary material. The average input length of samples in the curriculum is around 10. This benchmark is mainly designed to evaluate length generalization, where the test samples are much longer than training samples.

Results. We present the main results in Table 9.6, where we compare NeSS with the sequence-to-sequence model [254], sequence-to-tree model [80], LSTM augmented with a differentiable stack structure [96], and the neural parser [48]. All these models are trained on the curriculum of the While language designed in [48], and we defer the full evaluation results of more setups and baselines to the supplementary material, where we draw similar conclusions. Again, NeSS achieves 100% accuracy in 5 independent runs. We notice that none of the models without incorporating a symbolic machine generalizes to test inputs that are $500 \times$ longer than training samples, suggesting the necessity of the neural-symbolic model design. Meanwhile, compared to the neural parser model, NeSS achieves the same capability of precisely learning the grammar production rules, while it supports more applications that are not supported by the neural parser model, as discussed in Section 9.2.

9.5 Related Work

There has been an increasing interest in studying the compositional generalization of deep
neural networks for natural language understanding [148, 137, 21, 220]. A line of literature
develops different techniques for the SCAN domain proposed in [148], including architectural
design [221, 158, 93], training data augmentation [12], and meta learning [149, 194]. Note that
we have already provided a more detailed discussion in Section 9.4. In particular, the rule
synthesis approach in [194] also achieves 100% generalization performance as NeSS. However,
they design a meta-grammar space to generate training samples, which contains grammars
with the format close to the SCAN grammar, and their model requires a search process to
sample candidate grammars during the inference time. On the other hand, NeSS does not
assume the knowledge of a restricted meta-grammar space as in [149, 194]. In addition, no
search is needed for model evaluation, thus NeSS could be more efficient especially when the
task requires more examples as the test-time input specification.

Some recent work also studies compositional generalization for other applications, including
semantic parsing [137, 12], visual question answering [129, 21, 175, 250, 276, 119], image
captioning [191], and other grounded language understanding domains [220]. In particular, a
line of work proposes neural-symbolic approaches for visual question answering [175, 250, 276,
119], and the main goal is to achieve generalization to new composition of visual concepts,
as well as scenes with more objects than training images. Compared to vision benchmarks
measuring the compositional generalization, our tasks do not require visual understanding,
but typically need much longer execution traces.

On the other hand, length generalization has been emphasized for program induction,
where the learned model is supposed to generalize to longer test samples than the training
ones [95, 286, 217, 38, 48]. A line of approaches learn a neural network augmented with a
differentiable data structure or a differentiable machine [95, 286, 131, 133, 145, 96]. However,
these approaches either can not achieve length generalization, or are only capable of solving
simple tasks, as also shown in our evaluation. Another class of approaches incorporate a
symbolic machine into the neural network [217, 38, 285, 48, 268, 207], which enables length
generalization either with training supervision on execution traces [38], or well-designed
curriculum learning schemes [48, 268, 207]. In particular, our neural-symbolic architecture
is inspired by the neural parser introduced in [48], which designs a parsing machine based
on classic SHIFT-REDUCE systems [64, 182, 164, 48]. By serializing the target parse tree,
our machine fully covers the usages supported by the parsing machine. Meanwhile, the
incorporation of a memory module and enhanced instructions for sequence generation enables
NeSS to achieve generalization for not only algorithmic induction, but also natural language
understanding domains. Some recent work also studies length generalization for other tasks,
including relational reasoning [78], multi-task learning [41], and structure inference [169].

9.6   Discussion

In this work, we presented NeSS, a differentiable neural network to operate a symbolic
stack machine supporting general-purpose sequence-to-sequence generation, to achieve com-
positional generalization. To train NeSS without supervision on ground truth execution
traces, we proposed the notation of operational equivalence, which captured the primitive and
compositional rules via the similarity of execution traces. NeSS achieved 100% generalization
performance on four benchmarks ranging from natural language understanding to gram-
mar parsing. For future work, we consider extending our techniques to other applications
that require the understanding of compositional semantics, including grounded language
understanding and code translation.

Chapter 10

Conclusion

In this thesis, we present our work on learning-based techniques for program synthesis, and neural-symbolic reasoning for language understanding. For program synthesis applications from different specification formats, we have developed neural network architectures that learn structured representations of the input specifications and output programs, which better capture the syntactic and semantic characteristics of the programming languages under consideration. Our work demonstrate the importance of structured representation learning for a wide range of applications, including spreadsheet formula synthesis (Chapter 2), visualization code synthesis (Chapter 3), program translation (Chapter 6), and code optimization (Chapter 7).

For program synthesis from input-output examples, we present our execution-guided synthesis techniques to incorporate partial program execution states (Part II). The principle of execution-guided synthesis is to view the program execution as a sequence of manipulations to transform a program input into the corresponding output. From this perspective, when a partial program is synthesized, we can obtain the intermediate execution state, which explicitly reveals the synthesis progress, and guides the followup program generation process to move on to reach the target program output. When an interpreter is available to obtain the execution states of partial programs, we demonstrate that feeding the execution states as the synthesizer input significantly improves the synthesis performance (Chapter 4). For programming languages that do not support partial program execution, such as C, we further show that we can learn a neural executor to approximate the partial program execution states, which still provides remarkable performance gain (Chapter 5).

Besides developing learning techniques for program synthesis applications, in Part IV, we introduce program synthesis as a new learning formulation. Specifically, we present neural-symbolic frameworks that integrate symbolic modules into neural networks, which allows the neural network to compose and execute symbolic operators to represent its knowledge of the data. In particular, our neural-symbolic models learn to interpret and reason over complex text (Chapter 8), comprehend grammar rules that reveal the compositionality in languages, and generalize the knowledge to new inputs (Chapter 9).

10.1 Future Work

We envision that low-code development is the future of the programming paradigm, where coding skills are no longer required for programming computers, hence everyone is able to create new software. There are several grand challenges of developing learning-based program synthesis techniques towards the goal: (1) limited scalability and efficiency of program search; (2) a lack of understanding of the mechanism driving the predicted outputs; and (3) the weaknesses and vulnerabilities of learning models. In the following, we highlight some concrete future directions.

Learning-based program synthesis for scalable software tool development. Our past work has demonstrated the feasibility of learning-based program synthesis approaches for various synthetic benchmarks and real-world applications. However, we still observe significant challenges in scaling up the techniques to handle more sophisticated code in large-scale projects. Moreover, existing learning-based program synthesis models generally suffer from sample inefficiency for synthesizing general-purpose code; e.g., tens or even hundreds of samples might be required to correctly predict a Python utility function implemented in 10 lines of code. We plan to continue improving neural network architectures to better capture the underlying semantics of programs. Meanwhile, we aim to develop new program search algorithms to further improve the sample efficiency. For example, we will extend our execution-guided synthesis framework to more specification formats, and draw inspiration from classic divide-and-conquer algorithms to leverage the compositionality in programming languages and explore the program search space more efficiently.

Human-friendly interactive programming from multi-modal specifications. Our work on program improvement shows the importance of iteratively updating the predicted program according to the execution results [102]. Besides program execution, we can also directly seek user feedback to gradually refine the program. In our work on visualization code synthesis (Chapter 3), we demonstrated that we can learn a model to synthesize code in real-world Python Jupyter notebooks crawled from GitHub, where the input specification contains interleaved code blocks and natural language markdown. This interactive programming paradigm allows users to break down the full program and provide step-by-step natural language descriptions of each building block, so that the program synthesizer does not have to absorb the full program specification all at once. One important future research direction is to develop interactive program synthesis systems that learn to adapt the predictions according to the user feedback, and ask for clarifications when necessary. To give users more flexibility in specifying the program intents, we also plan to develop neural network architectures to effectively aggregate the information from input specifications of different types, e.g., simultaneously supporting natural language descriptions, input-output examples, and other external resources as the reference.

Symbolic reasoning towards better robustness and generalization. In our work on neural-symbolic reasoning (Part IV), we have revealed several types of weaknesses and limitations of existing deep neural networks, including their generalizability and reasoning capability. Besides that, our work on adversarial machine learning also highlighted the security risks of deep neural networks, such as test-time attacks [165, 272] and training-time data poisoning [58, 56]. By learning to generate a symbolic representation as the model output, neural-symbolic models could potentially be more robust to distribution shift, while the predictions are also more interpretable and easier to verify. In the future, we plan to work on more systematic investigation of the success and failure modes of deep neural networks, including large-scale pre-trained models for program synthesis and other domains. Meanwhile, we are passionate about extending our neural-symbolic framework to support more diverse and noisy inputs, such as open-domain images and natural language text, and designing new pre-training and data augmentation schemes to strengthen the reasoning capability.

Bibliography

[1]   Steven Abney. "Bootstrapping". In: *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*. 2002, pp. 360–367.

[2]   Michael Affenzeller and Rene Mayrhofer. "Generic heuristics for combinatorial optimization problems". In: *Proc. of the 9th International Conference on Operational Research*. 2002, pp. 83–92.

[3]   Rajas Agashe, Srinivasan Iyer, and Luke Zettlemoyer. "JuICe: A Large Scale Distantly Supervised Dataset for Open Domain Context-based Code Generation". In: *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*. 2019, pp. 5439–5449.

[4]   Roee Aharoni and Yoav Goldberg. "Towards string-to-tree neural machine translation". In: *ACL*. 2017.

[5]   Mejbah Alam et al. "A zero-positive learning approach for diagnosing software performance regressions". In: *Advances in Neural Information Processing Systems* 32 (2019), pp. 11627–11639.

[6]   Ferran Alet et al. "A large-scale benchmark for few-shot program induction and synthesis". In: *International Conference on Machine Learning*. PMLR. 2021, pp. 175–186.

[7]   Miltiadis Allamanis et al. "A Survey of Machine Learning for Big Code and Naturalness". In: *arXiv preprint arXiv:1709.06182* (2017).

[8]   Miltiadis Allamanis et al. "Learning Continuous Semantic Representations of Symbolic Expressions". In: *International Conference on Machine Learning*. 2017, pp. 80–88.

[9]   David Alvarez-Melis and Tommi S Jaakkola. "Tree-structured decoding with doubly-recurrent neural networks". In: *ICLR*. 2017.

[10]   Aida Amini et al. "MathQA: Towards Interpretable Math Word Problem Solving with Operation-Based Formalisms". In: *arXiv preprint arXiv:1905.13319* (2019).

[11]   Daniel Andor et al. "Giving BERT a Calculator: Finding Operations and Arguments with Reading Comprehension". In: *arXiv preprint arXiv:1909.00109* (2019).

[12] Jacob Andreas. "Good-enough compositional data augmentation". In: *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. 2020.

[13] Jacob Andreas et al. "Learning to Compose Neural Networks for Question Answering". In: *arXiv:1601.01705* (2016).

[14] Dana Angluin. "Learning regular sets from queries and counterexamples". In: *Information and computation* 75.2 (1987), pp. 87–106.

[15] Antlr. *Antlr*. https://github.com/antlr/. 2018.

[16] Michael Armbrust et al. "A view of cloud computing". In: *Communications of the ACM* 53.4 (2010), pp. 50–58.

[17] Jacob Austin et al. "Program synthesis with large language models". In: *arXiv preprint arXiv:2108.07732* (2021).

[18] Awais Azam, Khubaib Amjad Alam, and Areeba Umair. "Spreadsheet Smells: A Systematic Mapping Study". In: *2019 International Conference on Frontiers of Information Technology (FIT)*. IEEE. 2019, pp. 345–3455.

[19] Leo Bachmair and Harald Ganzinger. "Rewrite-based equational theorem proving with selection and simplification". In: *Journal of Logic and Computation* 4.3 (1994), pp. 217–247.

[20] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. "Neural machine translation by jointly learning to align and translate". In: *International Conference on Learning Representations*. 2015.

[21] Dzmitry Bahdanau et al. "CLOSURE: Assessing Systematic Generalization of CLEVR Models". In: *arXiv preprint arXiv:1912.05783* (2019).

[22] Matej Balog et al. "DeepCoder: Learning to Write Programs". In: *International Conference on Learning Representations*. 2017.

[23] Matej Balog et al. "Neural program synthesis with a differentiable fixer". In: *arXiv preprint arXiv:2006.10924* (2020).

[24] Reuven Bar-Yehuda and Shimon Even. "A linear-time approximation algorithm for the weighted vertex cover problem". In: *Journal of Algorithms* 2.2 (1981), pp. 198–203.

[25] Rohan Bavishi et al. "AutoPandas: neural-backed generators for program synthesis". In: *Proceedings of the ACM on Programming Languages* 3.OOPSLA (2019), pp. 1–27.

[26] Alessandro Bay and Biswa Sengupta. "Approximating meta-heuristics with homotopic recurrent neural networks". In: *arXiv preprint arXiv:1709.02194* (2017).

[27] Irwan Bello et al. "Neural combinatorial optimization with reinforcement learning". In: *arXiv preprint arXiv:1611.09940* (2016).

[28] Jonathan Berant et al. "Semantic Parsing on Freebase from Question-Answer Pairs." In: *EMNLP* 2.5 (2013), p. 6.

[29] David Bieber et al. "Learning to Execute Programs with Instruction Pointer Attention Graph Neural Networks". In: *Advances in Neural Information Processing Systems*. 2020.

[30] Jacek Błażewicz, Wolfgang Domschke, and Erwin Pesch. "The job shop scheduling problem: Conventional and new solution techniques". In: *European journal of operational research* 93.1 (1996), pp. 1–33.

[31] Rens Bod. "An all-subtrees approach to unsupervised parsing". In: *Proceedings of the 21st International Conference on Computational Linguistics and the 44th annual meeting of the Association for Computational Linguistics*. Association for Computational Linguistics. 2006, pp. 865–872.

[32] Michael Bostock, Vadim Ogievetsky, and Jeffrey Heer. "D^3 data-driven documents". In: *IEEE transactions on visualization and computer graphics* 17.12 (2011), pp. 2301–2309.

[33] James Bradbury and Richard Socher. "Towards Neural Machine Translation with Latent Tree Attention". In: *arXiv preprint arXiv:1709.01915* (2017).

[34] Steven J Bradtke, B Erik Ydstie, and Andrew G Barto. "Adaptive linear quadratic control using policy iteration". In: *Proceedings of the American control conference*. Vol. 3. Citeseer. 1994, pp. 3475–3475.

[35] Rudy Bunel et al. "Leveraging Grammar and Reinforcement Learning for Neural Program Synthesis". In: *International Conference on Learning Representations*. 2018. URL: https://openreview.net/forum?id=H1Xw62kRZ.

[36] Rudy R Bunel et al. "Adaptive neural compilation". In: *Advances in Neural Information Processing Systems*. 2016, pp. 1444–1452.

[37] Cheng-Hao Cai et al. "Learning of human-like algebraic reasoning using deep feedforward neural networks". In: *Biologically Inspired Cognitive Architectures* 25 (2018), pp. 43–50.

[38] Jonathon Cai, Richard Shin, and Dawn Song. "Making Neural Programming Architectures Generalize via Recursion". In: *ICLR*. 2017.

[39] Brian Campbell. "An executable semantics for CompCert C". In: *International Conference on Certified Programs and Proofs*. Springer. 2012, pp. 60–75.

[40] Xavier Carreras and Lluis Marquez. "Introduction to the CoNLL-2004 shared task: Semantic role labeling". In: *Proceedings of the Eighth Conference on Computational Natural Language Learning (CoNLL-2004) at HLT-NAACL 2004*. 2004, pp. 89–97.

[41] Michael B Chang et al. "Automatically composing representation transformations as a means for generalization". In: *International Conference on Learning Representations*. 2019.

[42] Danqi Chen et al. "Reading Wikipedia to Answer Open-Domain Questions". In: *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, ACL 2017, Vancouver, Canada, July 30 - August 4, Volume 1: Long Papers*. 2017, pp. 1870–1879.

[43] Mark Chen et al. "Evaluating large language models trained on code". In: *arXiv preprint arXiv:2107.03374* (2021).

[44] Tianqi Chen et al. "Learning to Optimize Tensor Programs". In: *NIPS* (2018).

[45] Weijia Chen, Yuedong Xu, and Xiaofeng Wu. "Deep Reinforcement Learning for Multi-Resource Multi-Machine Job Scheduling". In: *arXiv preprint arXiv:1711.07440* (2017).

[46] Wenhu Chen et al. "Tabfact: A large-scale dataset for table-based fact verification". In: *International Conference on Learning Representations*. 2020.

[47] Xinyun Chen, Chang Liu, and Dawn Song. "Execution-guided neural program synthesis". In: *International Conference on Learning Representations*. 2019.

[48] Xinyun Chen, Chang Liu, and Dawn Song. "Towards Synthesizing Complex Programs from Input-Output Examples". In: *ICLR*. 2018.

[49] Xinyun Chen, Chang Liu, and Dawn Song. "Tree-to-tree neural networks for program translation". In: *Advances in neural information processing systems* 31 (2018).

[50] Xinyun Chen, Dawn Song, and Yuandong Tian. "Latent execution for neural program synthesis beyond domain-specific languages". In: *Advances in Neural Information Processing Systems* 34 (2021).

[51] Xinyun Chen and Yuandong Tian. "Learning to perform local rewriting for combinatorial optimization". In: *Advances in Neural Information Processing Systems* 32 (2019).

[52] Xinyun Chen et al. "Compositional generalization via neural-symbolic stack machines". In: *Advances in Neural Information Processing Systems* 33 (2020), pp. 1690–1701.

[53] Xinyun Chen et al. "Latent attention for if-then program synthesis". In: *Advances in Neural Information Processing Systems*. 2016, pp. 4574–4582.

[54] Xinyun Chen et al. "Neural symbolic reader: Scalable integration of distributed and symbolic representations for reading comprehension". In: *International Conference on Learning Representations*. 2020.

[55] Xinyun Chen et al. "Plotcoder: Hierarchical decoding for synthesizing visualization code in programmatic context". In: *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*. 2021, pp. 2169–2181.

[56] Xinyun Chen et al. "Refit: a unified watermark removal framework for deep learning systems with limited data". In: *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*. 2021, pp. 321–335.

[57] Xinyun Chen et al. "Spreadsheetcoder: Formula prediction from semi-structured context". In: *International Conference on Machine Learning*. PMLR. 2021, pp. 1661–1672.

[58] Xinyun Chen et al. "Targeted backdoor attacks on deep learning systems using data poisoning". In: *arXiv preprint arXiv:1712.05526* (2017).

[59] Shing-Chi Cheung et al. "CUSTODES: automatic spreadsheet cell clustering and smell detection using strong and weak features". In: *Proceedings of the 38th International Conference on Software Engineering*. 2016, pp. 464–475.

[60] Sébastien Jean Kyunghyun Cho, Roland Memisevic, and Yoshua Bengio. "On Using Very Large Target Vocabulary for Neural Machine Translation". In: *ACL*. 2015.

[61] Noam Chomsky and David W Lightfoot. *Syntactic structures*. Walter de Gruyter, 2002.

[62] Judith Clymo et al. "Data generation for neural programming by example". In: *International Conference on Artificial Intelligence and Statistics*. PMLR. 2020, pp. 3450–3459.

[63] Trevor Anthony Cohn and Mirella Lapata. "Sentence compression as tree transduction". In: *Journal of Artificial Intelligence Research* 34 (2009), pp. 637–674.

[64] James Cross and Liang Huang. "Span-Based Constituency Parsing with a Structure-Label System and Provably Optimal Dynamic Oracles". In: *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*. 2016, pp. 1–11.

[65] CS106A. *Stanford CS106A course page*. https://see.stanford.edu/Course/CS106A. 2018.

[66] Pradeep Dasigi et al. "Iterative Search for Weakly Supervised Semantic Parsing". In: *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. 2019, pp. 2669–2680.

[67] Colin De la Higuera. *Grammatical inference: learning automata and grammars*. Cambridge University Press, 2010.

[68] Leonardo De Moura and Nikolaj Bjørner. "Z3: An efficient SMT solver". In: *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2008, pp. 337–340.

[69] Jia Deng et al. "Imagenet: A large-scale hierarchical image database". In: *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*. IEEE. 2009, pp. 248–255.

[70] Roberto Dessi and Marco Baroni. "CNNs found to jump around more skillfully than RNNs: Compositional Generalization in Seq2seq Convolutional Networks". In: *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*. 2019, pp. 3919–3923.

[71]   Michel Deudon et al. "Learning heuristics for the tsp by policy gradient". In: *International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research*. Springer. 2018, pp. 170–181.

[72]   Jacob Devlin et al. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding". In: *NAACL-HLT (1)*. 2019.

[73]   Jacob Devlin et al. "Neural Program Meta-Induction". In: *Advances in Neural Information Processing Systems*. 2017, pp. 2077–2085.

[74]   Jacob Devlin et al. "RobustFill: Neural Program Learning under Noisy I/O". In: *ICML*. 2017.

[75]   Victor Dibia and Cagatay Demiralp. "Data2vis: Automatic generation of data visualizations using sequence-to-sequence recurrent neural networks". In: *IEEE computer graphics and applications* 39.5 (2019), pp. 33–46.

[76]   Haoyu Dong et al. "Semantic Structure Extraction for Spreadsheet Tables with a Multi-task Learning Architecture". In: *Workshop on Document Intelligence at NeurIPS 2019*. 2019.

[77]   Haoyu Dong et al. "Tablesense: Spreadsheet table detection with convolutional neural networks". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 33. 2019, pp. 69–76.

[78]   Honghua Dong et al. "Neural logic machines". In: *International Conference on Learning Representations*. 2019.

[79]   Li Dong and Mirella Lapata. "Coarse-to-Fine Decoding for Neural Semantic Parsing". In: *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 2018, pp. 731–742.

[80]   Li Dong and Mirella Lapata. "Language to logical form with neural attention". In: *ACL*. 2016.

[81]   Wensheng Dou et al. "Detecting table clones and smells in spreadsheets". In: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 2016, pp. 787–798.

[82]   Ian Drosos et al. "Wrex: A Unified Programming-by-Example Interaction for Synthesizing Readable Code for Data Scientists". In: *CHI '20: CHI Conference on Human Factors in Computing Systems, Honolulu, HI, USA, April 25-30, 2020*. ACM, 2020, pp. 1–12.

[83]   Dheeru Dua et al. "DROP: A Reading Comprehension Benchmark Requiring Discrete Reasoning Over Paragraphs". In: *Proc. of NAACL*. 2019.

[84]   Chris Dyer et al. "Recurrent neural network grammars". In: *NAACL*. 2016.

[85]   Kevin Ellis et al. "Write, Execute, Assess: Program Synthesis with a REPL". In: *Advances in Neural Information Processing Systems*. 2019.

[86] Akiko Eriguchi, Kazuma Hashimoto, and Yoshimasa Tsuruoka. "Tree-to-sequence attentional neural machine translation". In: *ACL*. 2016.

[87] Richard Evans et al. "Can Neural Networks Understand Logical Entailment?" In: *ICLR* (2018).

[88] Dan Feblowitz and David Kauchak. "Sentence simplification as tree transduction". In: *Proceedings of the Second Workshop on Predicting and Improving Text Readability for Target Reader Populations*. 2013, pp. 1–10.

[89] Zhangyin Feng et al. "Codebert: A pre-trained model for programming and natural languages". In: *arXiv preprint arXiv:2002.08155* (2020).

[90] Roy Fox et al. "Parametrized Hierarchical Procedures for Neural Programming". In: *ICLR*. 2018.

[91] Yaroslav Ganin et al. "Synthesizing Programs for Images using Reinforced Adversarial Learning". In: *arXiv preprint arXiv:1804.01118* (2018).

[92] Google. *Google Or-Tools*. https://developers.google.com/optimization/. 2019.

[93] Jonathan Gordon et al. "Permutation equivariant models for compositional generalization in language". In: *International Conference on Learning Representations*. 2020.

[94] Robert Grandl et al. "Multi-resource packing for cluster schedulers". In: *ACM SIGCOMM Computer Communication Review* 44.4 (2015), pp. 455–466.

[95] Alex Graves, Greg Wayne, and Ivo Danihelka. "Neural turing machines". In: *arXiv preprint arXiv:1410.5401* (2014).

[96] Edward Grefenstette et al. "Learning to transduce with unbounded memory". In: *Advances in Neural Information Processing Systems*. 2015, pp. 1828–1836.

[97] Jiatao Gu et al. "Incorporating Copying Mechanism in Sequence-to-Sequence Learning". In: *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 2016, pp. 1631–1640.

[98] Sumit Gulwani. "Automating string processing in spreadsheets using input-output examples". In: *ACM SIGPLAN Notices*. Vol. 46. 1. ACM. 2011, pp. 317–330.

[99] Sumit Gulwani, William R Harris, and Rishabh Singh. "Spreadsheet data manipulation using examples". In: *Communications of the ACM* 55.8 (2012), pp. 97–105.

[100] Sumit Gulwani and Mark Marron. "Nlyze: Interactive programming by natural language for spreadsheet data analysis and manipulation". In: *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. 2014, pp. 803–814.

[101] Jiaqi Guo et al. "Towards Complex Text-to-SQL in Cross-Domain Database with Intermediate Representation". In: *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*. 2019, pp. 4524–4535.

[102] Kavi Gupta et al. "Synthesize, execute and debug: Learning to repair for neural program synthesis". In: *Advances in Neural Information Processing Systems*. 2020.

[103] Kelvin Guu et al. "From Language to Programs: Bridging Reinforcement Learning and Maximum Marginal Likelihood". In: *ACL* (2017).

[104] Tuomas Haarnoja et al. "Reinforcement learning with deep energy-based policies". In: *ICML*. JMLR. org. 2017, pp. 1352–1361.

[105] Halide. *Halide Simplifier*. https://github.com/halide/Halide. 2018.

[106] Di He et al. "Dual learning for machine translation". In: *Advances in Neural Information Processing Systems*. 2016, pp. 820–828.

[107] Kaiming He et al. "Deep residual learning for image recognition". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778.

[108] Dan Hendrycks et al. "Measuring Coding Challenge Competence With APPS". In: *arXiv preprint arXiv:2105.09938* (2021).

[109] Felienne Hermans and Emerson Murphy-Hill. "Enron's spreadsheets and related emails: A dataset and analysis". In: *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. Vol. 2. IEEE. 2015, pp. 7–16.

[110] Felienne Hermans, Martin Pinzger, and Arie van Deursen. "Detecting and visualizing inter-worksheet smells in spreadsheets". In: *2012 34th International Conference on Software Engineering (ICSE)*. IEEE. 2012, pp. 441–451.

[111] Felienne Hermans, Martin Pinzger, and Arie van Deursen. "Measuring spreadsheet formula understandability". In: *arXiv preprint arXiv:1209.3517* (2012).

[112] Felienne Hermans et al. "Data clone detection and visualization in spreadsheets". In: *2013 35th International Conference on Software Engineering (ICSE)*. IEEE. 2013, pp. 292–301.

[113] Jonathan Herzig et al. "TAPAS: Weakly Supervised Table Parsing via Pre-training". In: *Annual Meeting of the Association for Computational Linguistics (ACL)*. 2020.

[114] HoC. *Hour of Code*. https://codehs.com/hourofcode/. 2018.

[115] Sepp Hochreiter and Jürgen Schmidhuber. "Long short-term memory". In: *Neural computation* 9.8 (1997), pp. 1735–1780.

[116] Jieh Hsiang et al. "The term rewriting approach to automated theorem proving". In: *The Journal of Logic Programming* 14.1-2 (1992), pp. 71–99.

[117] Minghao Hu et al. "A Multi-Type Multi-Span Network for Reading Comprehension that Requires Discrete Reasoning". In: *arXiv preprint arXiv:1908.05514* (2019).

[118] Daniel Huang et al. "GamePad: A Learning Environment for Theorem Proving". In: *arXiv preprint arXiv:1806.00608* (2018).

[119] Drew A Hudson and Christopher D Manning. "Compositional Attention Networks for Machine Reasoning". In: *International Conference on Learning Representations*. 2018.

[120] John D Hunter. "Matplotlib: A 2D graphics environment". In: *Computing in science & engineering* 9.3 (2007), pp. 90–95.

[121] IText. *IText*. http://sourceforge.net/projects/itext/. 2018.

[122] Srinivasan Iyer et al. "Learning a Neural Semantic Parser from User Feedback". In: *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 2017, pp. 963–973.

[123] Srinivasan Iyer et al. "Mapping Language to Code in Programmatic Context". In: *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*. 2018, pp. 1643–1652.

[124] Mohit Iyyer, Wen-tau Yih, and Ming-Wei Chang. "Search-based neural structured learning for sequential question answering". In: *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 2017, pp. 1821–1831.

[125] Max Jaderberg et al. "Decoupled neural interfaces using synthetic gradients". In: *Proceedings of the 34th International Conference on Machine Learning-Volume 70*. JMLR. org. 2017, pp. 1627–1635.

[126] Java2CSharp. *Java2CSharp*. http://sourceforge.net/projects/j2cstranslator/. 2018.

[127] JGit. *JGit*. https://github.com/eclipse/jgit/. 2018.

[128] Robin Jia and Percy Liang. "Data recombination for neural semantic parsing". In: *ACL* (2016).

[129] Justin Johnson et al. "Clevr: A diagnostic dataset for compositional language and elementary visual reasoning". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2017, pp. 2901–2910.

[130] Justin Johnson et al. "Inferring and executing programs for visual reasoning". In: *Proceedings of the IEEE International Conference on Computer Vision*. 2017, pp. 2989–2998.

[131] Armand Joulin and Tomas Mikolov. "Inferring algorithmic patterns with stack-augmented recurrent nets". In: *NIPS*. 2015.

[132] JTS. *JTS*. http://sourceforge.net/projects/jts-topo-suite/. 2018.

[133] Łukasz Kaiser and Ilya Sutskever. "Neural gpus learn algorithms". In: *arXiv preprint arXiv:1511.08228* (2015).

[134] Svetoslav Karaivanov, Veselin Raychev, and Martin Vechev. "Phrase-based statistical translation of programming languages". In: *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*. ACM. 2014, pp. 173–184.

[135] Richard M Karp. "Reducibility among combinatorial problems". In: *Complexity of computer computations*. Springer, 1972, pp. 85–103.

[136] Andrej Karpathy, Justin Johnson, and Li Fei-Fei. "Visualizing and understanding recurrent networks". In: *arXiv preprint arXiv:1506.02078* (2015).

[137] Daniel Keysers et al. "Measuring Compositional Generalization: A Comprehensive Method on Realistic Data". In: *International Conference on Learning Representations*. 2020.

[138] Elias Khalil et al. "Learning combinatorial optimization algorithms over graphs". In: *Advances in Neural Information Processing Systems*. 2017, pp. 6348–6358.

[139] Diederik P Kingma and Jimmy Ba. "Adam: A method for stochastic optimization". In: *ICLR*. 2015.

[140] Guillaume Klein et al. "OpenNMT: Neural Machine Translation Toolkit". In: *arXiv preprint arXiv:1805.11462* (2018).

[141] Bryan Klimt and Yiming Yang. "Introducing the Enron corpus." In: *CEAS*. 2004.

[142] Wouter Kool, Herke van Hoof, and Max Welling. "Attention, Learn to Solve Routing Problems!" In: *International Conference on Learning Representations*. 2019. URL: https://openreview.net/forum?id=ByxBFsRqYm.

[143] Jayant Krishnamurthy, Pradeep Dasigi, and Matt Gardner. "Neural semantic parsing with type constraints for semi-structured tables". In: *EMNLP* (2017).

[144] Sumith Kulal et al. "Spoc: Search-based pseudocode to code". In: *Advances in Neural Information Processing Systems*. 2019.

[145] Karol Kurach, Marcin Andrychowicz, and Ilya Sutskever. "Neural random-access machines". In: *arXiv preprint arXiv:1511.06392* (2015).

[146] Matt J Kusner, Brooks Paige, and José Miguel Hernández-Lobato. "Grammar Variational Autoencoder". In: *arXiv preprint arXiv:1703.01925* (2017).

[147] Larissa Laich, Pavol Bielik, and Martin Vechev. "Guiding program synthesis by learning to generate examples". In: *International Conference on Learning Representations*. 2019.

[148] Brenden Lake and Marco Baroni. "Generalization without Systematicity: On the Compositional Skills of Sequence-to-Sequence Recurrent Networks". In: *International Conference on Machine Learning*. 2018, pp. 2873–2882.

[149] Brenden M Lake. "Compositional generalization through meta sequence-to-sequence learning". In: *Advances in Neural Information Processing Systems*. 2019, pp. 9788–9798.

[150] Brenden M Lake, Tal Linzen, and Marco Baroni. "Human few-shot learning of compositional instructions". In: *arXiv preprint arXiv:1901.04587* (2019).

[151] Brenden M Lake et al. "Building machines that learn and think like people". In: *Behavioral and brain sciences* 40 (2017).

[152] Guillaume Lample and François Charton. "Deep learning for symbolic mathematics". In: *International Conference on Learning Representations*. 2020.

[153] Gil Lederman, Markus N Rabe, and Sanjit A Seshia. "Learning Heuristics for Automated Reasoning through Deep Reinforcement Learning". In: *arXiv preprint arXiv:1807.08058* (2018).

[154] Sergey Levine and Pieter Abbeel. "Learning neural network policies with guided policy search under unknown dynamics". In: *Advances in Neural Information Processing Systems*. 2014, pp. 1071–1079.

[155] Sergey Levine and Vladlen Koltun. "Guided policy search". In: *International Conference on Machine Learning*. 2013, pp. 1–9.

[156] Chengtao Li et al. "Neural Program Lattices". In: *ICLR*. 2017.

[157] Jian Li et al. "Code completion with neural attention and pointer networks". In: *Proceedings of the 27th International Joint Conference on Artificial Intelligence*. 2018, pp. 4159–25.

[158] Yuanpeng Li et al. "Compositional Generalization for Primitive Substitutions". In: *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*. 2019, pp. 4284–4293.

[159] Chen Liang et al. "Memory augmented policy optimization for program synthesis and semantic parsing". In: *Advances in Neural Information Processing Systems*. 2018, pp. 9994–10006.

[160] Chen Liang et al. "Neural Symbolic Machines: Learning Semantic Parsers on Freebase with Weak Supervision". In: *ACL* (2017).

[161] Xi Victoria Lin et al. "NL2Bash: A Corpus and Semantic Parser for Natural Language Interface to the Linux Operating System". In: *Proceedings of the Eleventh International Conference on Language Resources and Evaluation (LREC 2018)*. 2018.

[162] Wang Ling et al. "Latent predictor networks for code generation". In: *ACL*. 2016.

[163] Wang Ling et al. "Program Induction by Rationale Generation: Learning to Solve and Explain Algebraic Word Problems". In: *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 2017, pp. 158–167.

[164] Jiangming Liu and Yue Zhang. "Shift-reduce constituent parsing with neural lookahead features". In: *Transactions of the Association for Computational Linguistics* 5 (2017), pp. 45–58.

[165] Yanpei Liu et al. "Delving into transferable adversarial examples and black-box attacks". In: *International Conference on Learning Representations*. 2016.

[166] Yinhan Liu et al. "RoBERTa: A Robustly Optimized BERT Pretraining Approach". In: *arXiv:1907.11692* (2019).

[167] Yunchao Liu and Zheng Wu. "Learning to describe scenes with programs". In: *International Conference on Learning Representations*. 2019.

[168]   Joao Loula, Marco Baroni, and Brenden M Lake. "Rearranging the familiar: Testing compositional generalization in recurrent networks". In: *arXiv preprint arXiv:1807.07545* (2018).

[169]   Sidi Lu et al. "Neurally-Guided Structure Inference". In: *International Conference on Machine Learning*. 2019.

[170]   Lucene. *Lucene*. http://lucene.apache.org/. 2018.

[171]   Minh-Thang Luong, Hieu Pham, and Christopher D Manning. "Effective approaches to attention-based neural machine translation". In: *arXiv preprint arXiv:1508.04025* (2015).

[172]   Shantanu Mandal et al. "Learning Fitness Functions for Machine Programming". In: *Proceedings of Machine Learning and Systems* 3 (2021).

[173]   Christopher Manning et al. "The Stanford CoreNLP natural language processing toolkit". In: *Proceedings of 52nd annual meeting of the association for computational linguistics: system demonstrations*. 2014, pp. 55–60.

[174]   Hongzi Mao et al. "Resource management with deep reinforcement learning". In: *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*. ACM. 2016, pp. 50–56.

[175]   Jiayuan Mao et al. "The neuro-symbolic concept learner: Interpreting scenes, words, and sentences from natural supervision". In: *International Conference on Learning Representations*. 2019.

[176]   Matplotlib. *Matplotlib Scatter method documentation*. `https://matplotlib.org/3.3.3/api/_as_gen/matplotlib.pyplot.scatter.html`. 2020.

[177]   DAVID Q MAYNE. "Differential Dynamic Programming–A Unified Approach to the Optimization of Dynamic Systems". In: *Control and Dynamic Systems*. Vol. 10. Elsevier, 1973, pp. 179–254.

[178]   David McClosky, Eugene Charniak, and Mark Johnson. "Effective self-training for parsing". In: *Proceedings of the Human Language Technology Conference of the NAACL, Main Conference*. 2006, pp. 152–159.

[179]   Charith Mendis et al. "Ithemal: Accurate, portable and fast basic block throughput estimation using deep neural networks". In: *International Conference on machine learning*. PMLR. 2019, pp. 4505–4515.

[180]   Qingkai Min et al. "Dialogue State Induction Using Neural Latent Variable Models". In: *International Joint Conferences on Artificial Intelligence*. 2020.

[181]   Sewon Min et al. "A Discrete Hard EM Approach for Weakly Supervised Question Answering". In: *arXiv preprint arXiv:1909.04849* (2019).

[182]   Dipendra Misra and Yoav Artzi. "Neural shift-reduce ccg semantic parsing". In: *Proceedings of the 2016 conference on empirical methods in natural language processing*. 2016, pp. 1775–1786.

[183] Richard Montague. "Universal grammar". In: *Theoria* 36.3 (1970), pp. 373–398.

[184] Christopher Z Mooney et al. *Bootstrapping: A nonparametric approach to statistical inference.* 95. sage, 1993.

[185] Vijayaraghavan Murali et al. "Neural Sketch Learning for Conditional Program Generation". In: *International Conference on Learning Representations.* 2018.

[186] MohammadReza Nazari et al. "Reinforcement Learning for Solving the Vehicle Routing Problem". In: *Advances in Neural Information Processing Systems.* 2018, pp. 9861–9871.

[187] Arvind Neelakantan et al. "Learning a natural language interface with neural programmer". In: *arXiv preprint arXiv:1611.08945* (2016).

[188] Anh Tuan Nguyen, Tung Thanh Nguyen, and Tien N Nguyen. "Divide-and-conquer approach for multi-phase statistical migration for source code (t)". In: *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on.* IEEE. 2015, pp. 585–596.

[189] Anh Tuan Nguyen, Tung Thanh Nguyen, and Tien N Nguyen. "Lexical statistical machine translation for language migration". In: *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering.* ACM. 2013, pp. 651–654.

[190] Trong Duc Nguyen, Anh Tuan Nguyen, and Tien N Nguyen. "Mapping API elements for code migration with vector representations". In: *Software Engineering Companion (ICSE-C), IEEE/ACM International Conference on.* IEEE. 2016, pp. 756–758.

[191] Mitja Nikolaus et al. "Compositional Generalization in Image Captioning". In: *Proceedings of the 23rd Conference on Computational Natural Language Learning (CoNLL).* 2019, pp. 87–98.

[192] Maxwell Nye et al. "Learning to Infer Program Sketches". In: *International Conference on Machine Learning.* 2019, pp. 4861–4870.

[193] Maxwell Nye et al. "Representing Partial Programs with Blended Abstract Semantics". In: *International Conference on Learning Representations.* 2021.

[194] Maxwell I Nye et al. "Learning Compositional Rules via Neural Program Synthesis". In: *arXiv preprint arXiv:2003.05562* (2020).

[195] Yusuke Oda et al. "Learning to generate pseudo-code from source code using statistical machine translation (t)". In: *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE).* IEEE. 2015, pp. 574–584.

[196] Augustus Odena and Charles Sutton. "Learning to represent programs with property signatures". In: *International Conference on Learning Representations.* 2020.

[197] Augustus Odena et al. "BUSTLE: Bottom-up program-Synthesis Through Learning-guided Exploration". In: *arXiv preprint arXiv:2007.14381* (2020).

[198] OpenAI. *OpenAI Dota 2 Bot.* https://openai.com/the-international/. 2018.

[199] Gustavo H Paetzold and Lucia Specia. "Text simplification as tree transduction". In: *Proceedings of the 9th Brazilian Symposium in Information and Human Language Technology.* 2013.

[200] Pandas. *Pandas Dataframe documentation.* https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.html. 2020.

[201] Emilio Parisotto et al. "Neuro-Symbolic Program Synthesis". In: *International Conference on Learning Representations.* 2017.

[202] Panupong Pasupat and Percy Liang. "Compositional Semantic Parsing on Semi-Structured Tables". In: *ACL* (2015).

[203] Adam Paszke et al. "Automatic differentiation in PyTorch". In: *NIPS-W.* 2017.

[204] Richard E Pattis. *Karel the robot: a gentle introduction to the art of programming.* John Wiley & Sons, Inc., 1981.

[205] Chris Paxton et al. "Prospection: Interpretable plans from language by predicting the future". In: *2019 International Conference on Robotics and Automation (ICRA).* IEEE. 2019, pp. 6942–6948.

[206] Matthew E. Peters et al. "Deep Contextualized Word Representations". In: *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2018, New Orleans, Louisiana, USA, June 1-6, 2018, Volume 1 (Long Papers).* 2018, pp. 2227–2237.

[207] Thomas Pierrot et al. "Learning compositional neural programs with recursive tree search and planning". In: *Advances in Neural Information Processing Systems.* 2019, pp. 14646–14656.

[208] POI. *POI.* http://poi.apache.org/. 2018.

[209] Illia Polosukhin and Alexander Skidanov. "Neural program search: Solving programming tasks from description and examples". In: *arXiv preprint arXiv:1802.04335* (2018).

[210] Oleksandr Polozov and Sumit Gulwani. "FlashMeta: a framework for inductive program synthesis". In: *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications.* 2015, pp. 107–126.

[211] Ana-Maria Popescu, Oren Etzioni, and Henry Kautz. "Towards a theory of natural language interfaces to databases". In: *Proceedings of the 8th international conference on Intelligent user interfaces.* 2003, pp. 149–157.

[212] Maxim Rabinovich, Mitchell Stern, and Dan Klein. "Abstract Syntax Networks for Code Generation and Semantic Parsing". In: *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers).* Vol. 1. 2017, pp. 1139–1149.

[213] Jonathan Ragan-Kelley et al. "Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines". In: *ACM SIGPLAN Notices* 48.6 (2013), pp. 519–530.

[214] Pranav Rajpurkar et al. "SQuAD: 100,000+ Questions for Machine Comprehension of Text". In: *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing, EMNLP 2016, Austin, Texas, USA, November 1-4, 2016.* 2016, pp. 2383–2392.

[215] Veselin Raychev, Martin Vechev, and Eran Yahav. "Code completion with statistical language models". In: *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation.* 2014, pp. 419–428.

[216] Siva Reddy, Danqi Chen, and Christopher D. Manning. "CoQA: A Conversational Question Answering Challenge". In: *TACL* 7 (2019), pp. 249–266.

[217] Scott Reed and Nando De Freitas. "Neural programmer-interpreters". In: *ICLR.* 2016.

[218] Colin R Reeves. *Modern heuristic techniques for combinatorial problems. Advanced topics in computer science.* Vol. 15. Mc Graw-Hill, 1995.

[219] Azriel Rosenfeld and Mark Thurston. "Edge and curve detection for visual scene analysis". In: *IEEE Transactions on computers* 5 (1971), pp. 562–569.

[220] Laura Ruis et al. "A Benchmark for Systematic Generalization in Grounded Language Understanding". In: *arXiv preprint arXiv:2003.05161* (2020).

[221] Jake Russin et al. "Compositional generalization in a deep seq2seq model by separating syntax and semantics". In: *arXiv preprint arXiv:1904.09708* (2019).

[222] David Saxton et al. "Analysing mathematical reasoning abilities of neural models". In: (2019).

[223] Eric Schkufza, Rahul Sharma, and Alex Aiken. "Stochastic superoptimization". In: *ACM SIGARCH Computer Architecture News.* Vol. 41. 1. ACM. 2013, pp. 305–316.

[224] Ziv Scully et al. "Optimally scheduling jobs with multiple tasks". In: *ACM SIGMETRICS Performance Evaluation Review* 45.2 (2017), pp. 36–38.

[225] Seaborn. *mwaskom/seaborn library documentation.* 2020.

[226] Min Joon Seo et al. "Bidirectional Attention Flow for Machine Comprehension". In: *ICLR.* 2017.

[227] Eui Chul Shin, Illia Polosukhin, and Dawn Song. "Improving neural program synthesis with inferred execution traces". In: *Advances in Neural Information Processing Systems.* 2018, pp. 8917–8926.

[228] Richard Shin, Illia Polosukhin, and Dawn Song. "Towards Specification-Directed Program Repair". In: (2018).

[229] Richard Shin et al. "Synthetic datasets for neural program synthesis". In: *International Conference on Learning Representations.* 2019.

[230] David Silver et al. "Mastering the game of Go without human knowledge". In: *Nature* 550.7676 (2017), p. 354.

[231] Rishabh Singh, Benjamin Livshits, and Benjamin Zorn. "Melford: Using neural networks to find spreadsheet errors". In: (2017).

[232] Richard Socher et al. "Parsing natural scenes and natural language with recursive neural networks". In: *Proceedings of the 28th international conference on machine learning (ICML-11)*. 2011, pp. 129–136.

[233] Richard Socher et al. "Semi-supervised recursive autoencoders for predicting sentiment distributions". In: *Proceedings of the conference on empirical methods in natural language processing*. Association for Computational Linguistics. 2011, pp. 151–161.

[234] Armando Solar-Lezama. "Program Synthesis by Sketching". PhD thesis. UNIVERSITY OF CALIFORNIA, BERKELEY, 2008.

[235] Niklas Sorensson and Niklas Een. "Minisat v1. 13-a sat solver with conflict-clause minimization". In: *SAT* 2005.53 (2005), pp. 1–2.

[236] Mark Steedman. *The syntactic process*. Vol. 24. MIT press Cambridge, MA, 2000.

[237] Alexander Suh and Yuval Timen. "Creating Synthetic Datasets via Evolution for Neural Program Synthesis". In: *arXiv preprint arXiv:2003.10485* (2020).

[238] Shao-Hua Sun et al. "Neural program synthesis from diverse demonstration videos". In: *International Conference on Machine Learning*. 2018, pp. 4790–4799.

[239] Martin Sundermeyer, Ralf Schlüter, and Hermann Ney. "LSTM neural networks for language modeling". In: *Thirteenth annual conference of the international speech communication association*. 2012.

[240] Richard S Sutton, Andrew G Barto, et al. *Reinforcement learning: An introduction*. 1998.

[241] Alexey Svyatkovskiy et al. "IntelliCode Compose: Code Generation Using Transformer". In: *arXiv preprint arXiv:2005.08025* (2020).

[242] Alexey Svyatkovskiy et al. "Pythia: AI-assisted code completion system". In: *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 2019, pp. 2727–2735.

[243] Alexey Svyatkovskoy et al. "Fast and Memory-Efficient Neural Code Completion". In: *arXiv preprint arXiv:2004.13651* (2020).

[244] Kai Sheng Tai, Richard Socher, and Christopher D Manning. "Improved semantic representations from tree-structured long short-term memory networks". In: *Proceedings of the Annual Meeting of the Association for Computational Linguistics*. 2015.

[245] Yuval Tassa, Tom Erez, and Emanuel Todorov. "Synthesis and stabilization of complex behaviors through online trajectory optimization". In: *Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on*. IEEE. 2012, pp. 4906–4913.

[246] Daria Terekhov, Douglas G Down, and J Christopher Beck. "Queueing-theoretic approaches for dynamic scheduling: a survey". In: *Surveys in Operations Research and Management Science* 19.2 (2014), pp. 105–129.

[247] Yonglong Tian et al. "Learning to infer and execute 3d shape programs". In: *International Conference on Learning Representations*. 2019.

[248] Yuandong Tian and Srinivasa G Narasimhan. "Hierarchical data-driven descent for efficient optimal deformation estimation". In: *Proceedings of the IEEE International Conference on Computer Vision*. 2013, pp. 2288–2295.

[249] Ashish Vaswani et al. "Attention is all you need". In: *Advances in neural information processing systems*. 2017, pp. 5998–6008.

[250] Ramakrishna Vedantam et al. "Probabilistic Neural Symbolic Models for Interpretable Visual Question Answering". In: *International Conference on Machine Learning*. 2019, pp. 6428–6437.

[251] Petar Veličković et al. "Neural execution of graph algorithms". In: *International Conference on Learning Representations*. 2020.

[252] Ashwin J Vijayakumar et al. "Neural-Guided Deductive Search for Real-Time Program Synthesis from Examples". In: *ICLR*. 2018.

[253] Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. "Pointer networks". In: *Advances in neural information processing systems*. 2015, pp. 2692–2700.

[254] Oriol Vinyals et al. "Grammar as a foreign language". In: *NIPS*. 2015.

[255] Draguna Vrabie et al. "Adaptive optimal control for continuous-time linear systems based on policy iteration". In: *Automatica* 45.2 (2009), pp. 477–484.

[256] Alex Wang et al. "GLUE: A Multi-Task Benchmark and Analysis Platform for Natural Language Understanding". In: *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. 2019.

[257] Bailin Wang et al. "Rat-sql: Relation-aware schema encoding and linking for text-to-sql parsers". In: *Annual Meeting of the Association for Computational Linguistics (ACL)*. 2020.

[258] Chenglong Wang, Alvin Cheung, and Rastislav Bodik. "Synthesizing highly expressive SQL queries from input-output examples". In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI*. ACM, 2017, pp. 452–466.

[259] Chenglong Wang et al. "Falx: Synthesis-Powered Visualization Authoring". In: *CHI '21: CHI Conference on Human Factors in Computing Systems*. ACM, 2021, 106:1–106:15.

[260] Chenglong Wang et al. "Robust Text-to-SQL Generation with Execution-Guided Decoding". In: *arXiv preprint arXiv:1807.03100* (2018).

[261] Chenglong Wang et al. "Visualization by example". In: *Proceedings of the ACM on Programming Languages* 4.POPL (2019), pp. 1–28.

[262] Ke Wang, Rishabh Singh, and Zhendong Su. "Dynamic Neural Program Embedding for Program Repair". In: *ICLR*. 2018.

[263] Wenhui Wang et al. "Gated Self-Matching Networks for Reading Comprehension and Question Answering". In: *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, ACL 2017, Vancouver, Canada, July 30 - August 4, Volume 1: Long Papers*. 2017, pp. 189–198.

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

[265] Ronald J Williams. "Simple statistical gradient-following algorithms for connectionist reinforcement learning". In: *Reinforcement Learning*. Springer, 1992, pp. 5–32.

[266] Sam Wiseman and Alexander M Rush. "Sequence-to-sequence learning as beam-search optimization". In: *EMNLP*. 2016.

[267] Jiajun Wu, Joshua B Tenenbaum, and Pushmeet Kohli. "Neural scene de-rendering". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2017, pp. 699–707.

[268] Da Xiao, Jo-Yu Liao, and Xingyuan Yuan. "Improving the Universality and Learnability of Neural Programmer-Interpreters with Combinator Abstraction". In: *ICLR*. 2018.

[269] Qizhe Xie et al. "Self-training with noisy student improves imagenet classification". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2020, pp. 10687–10698.

[270] Caiming Xiong, Victor Zhong, and Richard Socher. "Dynamic Coattention Networks For Question Answering". In: *CoRR* abs/1611.01604 (2016). arXiv: `1611.01604`. URL: `http://arxiv.org/abs/1611.01604`.

[271] Xiaojun Xu, Chang Liu, and Dawn Song. "SQLNet: Generating Structured Queries From Natural Language Without Reinforcement Learning". In: *arXiv preprint arXiv:1711.04436* (2017).

[272] Xiaojun Xu et al. "Fooling vision and language models despite localization and attention mechanism". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2018, pp. 4951–4961.

[273] Yujun Yan et al. "Neural execution engines: Learning to execute subroutines". In: *Advances in Neural Information Processing Systems*. 2020.

[274] Xuejun Yang et al. "Finding and understanding bugs in C compilers". In: *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*. 2011, pp. 283–294.

[275] Zhilin Yang et al. "XLNet: Generalized Autoregressive Pretraining for Language Understanding". In: *CoRR* abs/1906.08237 (2019).

[276] Kexin Yi et al. "Neural-symbolic vqa: Disentangling reasoning from vision and language understanding". In: *Advances in neural information processing systems*. 2018, pp. 1031–1042.

[277] Pengcheng Yin and Graham Neubig. "A Syntactic Neural Model for General-Purpose Code Generation". In: *ACL*. 2017.

[278] Pengcheng Yin et al. "Learning to mine aligned code and natural language pairs from stack overflow". In: *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*. IEEE. 2018, pp. 476–486.

[279] Pengcheng Yin et al. "TaBERT: Pretraining for Joint Understanding of Textual and Tabular Data". In: *Annual Meeting of the Association for Computational Linguistics (ACL)*. 2020.

[280] Adams Wei Yu et al. "QANet: Combining Local Convolution with Global Self-Attention for Reading Comprehension". In: *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. 2018.

[281] Tao Yu et al. "CoSQL: A Conversational Text-to-SQL Challenge Towards Cross-Domain Natural Language Interfaces to Databases". In: *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*. 2019, pp. 1962–1979.

[282] Tao Yu et al. "SParC: Cross-Domain Semantic Parsing in Context". In: *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*. 2019, pp. 4511–4523.

[283] Tao Yu et al. "Spider: A Large-Scale Human-Labeled Dataset for Complex and Cross-Domain Semantic Parsing and Text-to-SQL Task". In: *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*. 2018, pp. 3911–3921.

[284] Wojciech Zaremba and Ilya Sutskever. "Learning to execute". In: *arXiv preprint arXiv:1410.4615* (2014).

[285] Wojciech Zaremba and Ilya Sutskever. "Reinforcement Learning Neural Turing Machines-Revised". In: *arXiv preprint arXiv:1505.00521* (2015).

[286] Wojciech Zaremba et al. "Learning Simple Algorithms from Examples". In: *Proceedings of The 33rd International Conference on Machine Learning*. 2016, pp. 421–429.

[287] Maksym Zavershynskyi, Alex Skidanov, and Illia Polosukhin. "NAPS: Natural program synthesis dataset". In: *arXiv preprint arXiv:1807.03168* (2018).

[288] Luke S Zettlemoyer and Michael Collins. "Learning to map sentences to logical form: Structured classification with probabilistic categorial grammars". In: *arXiv preprint arXiv:1207.1420* (2012).

[289] Xingxing Zhang, Liang Lu, and Mirella Lapata. "Top-down Tree Long Short-Term Memory Networks". In: *Proceedings of NAACL-HLT*. 2016, pp. 310–320.

[290] Yakun Zhang et al. "Learning to detect table clones in spreadsheets". In: *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 2020, pp. 528–540.

[291] Yichi Zhang et al. "A Probabilistic End-To-End Task-Oriented Dialog Model with Latent Belief States towards Semi-Supervised Learning". In: *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. 2020.

[292] Victor Zhong, Caiming Xiong, and Richard Socher. "Seq2sql: Generating structured queries from natural language using reinforcement learning". In: *arXiv preprint arXiv:1709.00103* (2017).

[293] Xiaodan Zhu, Parinaz Sobhani, and Hongyu Guo. "Dag-structured long short-term memory for semantic compositionality". In: *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. 2016, pp. 917–926.

[294] Xiaodan Zhu, Parinaz Sobihani, and Hongyu Guo. "Long short-term memory over recursive structures". In: *International Conference on Machine Learning*. 2015, pp. 1604–1612.

[295] Amit Zohar and Lior Wolf. "Automatic Program Synthesis of Long Programs with a Learned Garbage Collector". In: *Advances in Neural Information Processing Systems*. 2018, pp. 2094–2103.

Appendix A

SpreadsheetCoder: Formula Prediction from Semi-structured Context

A.1 An Extended Discussion of Related Work

Various neural network approaches have been proposed for the FlashFill benchmark [201, 74, 252]. Specifically, both R3NN [201] and RobustFill [74] are purely statistical models, and RobustFill performs better. In a RobustFill model, each formula is executed on a single data row, thus each row is independently fed into a shared encoder. Afterwards, at each decoding step, a shared LSTM decoder generates a hidden state per data row, which are then fed into a max pooling layer. Finally, the pooled hidden state is fed into a fully-connected layer to predict the formula token. On the other hand, in [252], they design a neural network to guide the deductive search performed by PROSE [210], a commercial framework for input-output program synthesis. A recent work proposes neural-guided bottom-up search for program synthesis from input-output examples, and they extend the domain-specific language of FlashFill to support more spreadsheet programs [197].

Besides formula prediction, some previous work has studied other applications related to spreadsheets, including smell detection [110, 59, 231, 18], clone detection [112, 81, 290], and structure extraction for spreadsheet tables [76, 77]. Our proposed encoder architecture could potentially be adapted for these spreadsheet tasks as well, and we leave it for future work.

A.2 More Experimental Results

For the setting where the model input does not include headers, corresponding to Table 2.3 in Section 2.4, we present the sketch and range accuracies in Table A.1, and the breakdown accuracies on formulas of different sketch lengths in Figure A.1. We observe that the performance degradation is more severe for formulas of sketch lengths 2–3.

Figure A.1: Top-1 formula accuracies for different sketch lengths, excluding headers in the context.

A.3 More Dataset Details

Although in principle, our model could generate formulas using any operator in the spreadsheet language, some kinds of value references are impossible to predict from local context, thus we remove formulas with such values from our dataset. Specifically, we exclude formulas that use the HYPERLINK function with a literal URL, since those are merely "stylistic" formulas that perform no computation beyond presenting a URL as a clickable link. As discussed in Section 2.2, we also filtered out formulas with cross-references from other tabs or spreadsheets. In total, the formulas filtered out after these two steps constitute around 40% of all formulas. We further filtered out formulas with cell references farther than 10 rows or columns from the target cell in either direction, and formulas with absolute cell ranges. In this way, about 45% of the original set of formulas are kept in our dataset.

Meanwhile, we observe that some spreadsheets may have tens of thousands of rows including the same formula, and including all of them in the dataset could bias our data distribution. Therefore, when multiple rows in the same spreadsheet table include the same formula in the same column, we keep the first 10 occurrences of such a formula, and create one data sample per formula. In this way, we extract around 846K formulas from 20M formulas before this filtering step, and we split them into 770K training samples, 42K for validation,

Table A.1: Breakdown accuracies on the test set, excluding headers in the context.

(a) Sketch accuracy.

	Approach	Top-1	Top-5	Top-10
	Full Model	28.33%	**62.55%**	72.89%
−	Column-based BERT	28.40%	61.60%	**74.92%**
−	Row-based BERT	27.71%	60.84%	73.43%
−	Pretraining	**28.78%**	62.37%	74.61%
	Row-based RobustFill	25.78%	42.66%	50.17%
	Column-based RobustFill	26.15%	47.78%	57.72%
	No context	25.19%	47.08%	52.70%

(b) Range accuracy.

	Approach	Top-1	Top-5	Top-10
	Full Model	22.60%	47.11%	53.84%
−	Column-based BERT	22.82%	**47.76%**	**54.98%**
−	Row-based BERT	22.47%	46.14%	54.51%
−	Pretraining	**23.48%**	47.27%	54.59%
	Row-based RobustFill	21.01%	38.21%	43.89%
	Column-based RobustFill	21.27%	37.80%	43.77%
	No context	11.80%	25.54%	38.07%

and 34K for testing.

In total, around 100 operators are covered in our output vocabulary. Among all spreadsheet formulas included in our filtered dataset, we list the 30 most commonly used spreadsheet functions and operators with their types [1] as follows: SUM (Math), + (Operator, equivalent to ADD), – (Operator, equivalent to MINUS), * (Operator, equivalent to MULTIPLY), / (Operator, equivalent to DIV), & (Operator, equivalent to CONCAT), AVERAGE (Statistical), LEN (Text), UPLUS (Operator), STDEV (Statistical), COUNTA (Statistical), MAX (Statistical), LEFT (Text), IFERROR (Logical), ABS (Math), MEDIAN (Statistical), UMINUS (Operator), CONCATENATE (Text), ROUND (Math), WEEKNUM (Date), AVERAGEA (Statistical), MIN (Statistical), COUNT (Statistical), TRIM (Text), COS (Math), SIN (Math), SINH (Math), TODAY (Date), IF (Logical), MONTH (Date). We observe that most of these functions and operators are for mathematical calculation, statistical computation, and text manipulation. However, people also write conditional statements, and spreadsheet formulas for calculating the dates.

The spreadsheet functions and operators utilized in the Enron corpus are: + (Operator, equivalent to ADD), SUM (Math), – (Operator, equivalent to MINUS), UPLUS (Operator), * (Operator, equivalent to MULTIPLY), / (Operator, equivalent to DIV), AVERAGE (Statistical), MIN (Statistical), MAX (Statistical), UMINUS (Operator), COUNT (Statistical), COUNTA (Statistical),

---

[1]The function types are based on the Google Sheets function list here: `https://support.google.com/docs/table/25273?hl=en`.

`ABS` (Math), `LN` (Math), `DAY` (Date), `WEEKDAY` (Date), and `STDEV` (Statistical).

A.4 More Discussion of the FlashFill-like Setting

Following prior work on FlashFill [74, 201, 252], we evaluate model performance when different numbers of data rows are presented to the model as input. Specifically, when the input includes 1–11 data rows, we grow the input from the target row upward. Our full data context includes 21 data rows, with 10 rows above the target cell, 10 rows below the target cell, and 1 row where the target cell locates. Consistent with prior work, when we vary the number of input data rows during inference, we always evaluate the same model trained with the full data context including 21 data rows. Since RobustFill independently encodes each row, it supports variable number of input rows by design. For our models with the tabular input representation, we set the rows to be empty when they are out of the input scope, and apply a mask to indicate that the corresponding data values are invalid.

A.5 Implementation Details

Data preprocessing. The content in each cell includes its data type and value, and we concatenate them as a token sequence. For example, `A2` in Figure 2.1a is represented as `num 0`. As discussed in Section 2.3, we concatenate all cell values in the same row as a token sequence, where values of different cells are separated by the `[SEP]` token. Each data row fed into the model includes $L = 128$ tokens, and when the concatenated token sequence exceeds the length limit, we discard cells that are further away from the target cell. For column-wise representation, we produce token embeddings independently for each column-wise bundle $S_{cb} = [H_c, C_{3b-1}, C_{3b}, C_{3b+1}]$ for $b \in [-3, 3]$, where C_i is a token sequence produced by concatenating all tokens of the cells in column C_i.

Output vocabulary construction. To construct the output formula token vocabulary, we filtered out tokens that appear less than 10 times in the training set, so that the vocabulary contains 462 tokens, out of 2625 tokens before filtering. In total, around a hundred operators are covered in our output vocabulary, including 82 spreadsheet-specific functions, and other general-purpose numerical operators (e.g., `+`, `-`).

Hyper-parameters. The formula decoder is a 1-layer LSTM with the hidden size of 512. We train the model with the Adam optimizer, with an initial learning rate of 5e-5. We train models for 200K minibatch updates, with a batch size 64. We set the dropout rate to be 0.1 for training. The norm for gradient clipping is 1.0.

Appendix B

PlotCoder: Synthesizing Visualization Code in Programmatic Context

B.1   Implementation Details

For the model input, we select the suffix of the code sequence when it exceeds the length limit, and we select the prefix for the natural language. To construct the vocabularies, we include natural language words that occur at least 15 times in the training set, and code tokens that occur at least 1,000 times, so that each vocabulary includes around $10,000$ tokens. We include an `[UNK]` token in both vocabularies, which is used to encode all input tokens not appeared in our vocabularies.

The model parameters are randomly initialized within $[-0.1, 0.1]$. Each LSTM has 2 layers, and a hidden size of 512. The embedding size of all embedding matrices is 512, and the hidden size of the linear layers is 512. For training, the batch size is 32, the initial learning rate is `1e-3`, with a decay rate of 0.9 after every $6,000$ batch updates. The dropout rate is 0.2, and the norm for gradient clipping is 5.0.

For models using the Transformer architecture as the encoder, we use the pre-trained RoBERTa-base and codeBERT from their official repositories.[1] The hyper-parameters are largely the same as the LSTM-based models, except that we added a linear learning rate warmup for the first $6,000$ training steps, which is the common practice for fine-tuning BERT-like models.

Figure B.1: Program accuracy with different number of input code cells. (a) Results of different model architectures. (b) The comparison between the accuracy of the hierarchical model and the upper bounds.

B.2   Training with Varying Number of Contextual Code Cells

As discussed in Section 3.4, we provide the results of including different number of local code cells as the model input in Figure B.1. We also evaluated the upper bounds of program accuracies for different values of K, where we consider an example to be predictable if all plotted data in the target program are covered in the input code context. We observe that including dataframe manipulation code in distant code cells improves the coverage, especially when K is small.

B.3   Detailed Analysis on Results Per Plot Type

In Section 3.4, we present the breakdown results per plot type in Tables 3.6 and 3.7, where we observed that "Scatter" and "Others" constitute the majority of the dataset, and the hierarchical model learns to better categorize plot types from a large number of training samples.

Note that for categories that the hierarchical model does not perform better than baselines, even if the accuracy differences are noticeable, the numbers of correct predictions do not differ much. For example, among the 13 samples in the "Pie" category, the hierarchical model correctly classifies 8 samples, while the non-hierarchical version makes 10 correct predictions. When looking at the predictions, we observe that the 2 different predictions are mainly due to the ambiguity of the natural language descriptions. Specifically, the text descriptions are "The average score of group A is better than average score of group B in 51% of the

---

[1]RoBERTa: https://github.com/pytorch/fairseq/tree/master/examples/roberta
codeBERT: https://github.com/microsoft/CodeBERT

state" and "I am analyzing the data of all male passengers". In fact, for both examples, the hierarchical model still generates a program including the plotted data in the ground truth. However, the hierarchical model wrongly selects `plt.bar` as the plotting API for the former sample, and selects `plt.scatter` for the latter sample, where it additionally selects another variable for the x-axis. For these 2 samples, we observe that the code context includes plotting programs that use other data to generate pie charts, and the non-hierarchical model picks a heuristic to select the same plot type in the code context when there is no cue provided in the natural language description, while the hierarchical model selects plot types that happen more frequently in the training distribution. A similar phenomenon holds for other categories or data splits with a small number of examples.

B.4   Other Plot Types

In the "Others" category discussed in Section 3.2, besides the plots generated by `plt.plot`, there are also other plot types, with much smaller data sizes than `plt.plot`. In Table B.1, we present the breakdown accuracies of some plot types, which constitute the largest percentages in the "Others" category excluding `plt.plot` samples. Specifically, around 4% samples use `boxplot`, and each of the other 3 plot types include around 1% samples. Due to the lack of data for such plot types, the results are much lower than the overall accuracies of all plot categories, but still non-trivial.

Plot Type	Plot Type Acc	Plotted Data Acc	Program Acc
boxplot	51.04%	10.42%	7.29%
pairplot	42.31%	34.62%	23.00%
jointplot	36.36%	9.09%	4.55%
violinplot	47.06%	5.88%	5.88%

Table B.1: Breakdown accuracies of plots in "Others" category on Test (hard), using the full hierarchical model.

B.5   More Discussion of Error Analysis

As discussed in Section 3.4, the lack of information in natural language descriptions is the main reason for a large proportion of wrong predictions (categories 1-3 in Table 3.8).

- Many natural language descriptions only mention the plot type, e.g., "Make a scatter plot," which is one reason that the plot type accuracy is generally much higher than the plotted data accuracy. (1)

- Sometimes the text only mentions the plotted data without specifying the plot type, e.g., "Plot the data x_1 and x_2," where both `plt.plot(x₁,x₂)` and `plt.scatter(x₁,x₂)` are possible predictions, and the model needs to infer the plot type from the code context. (2)

- The text description includes no plotting information at all, e.g., "Localize your search around the value you found above," where the model needs to infer which variables are search results and could be plotted. (3)

We consider several directions to address different error categories as future work. To mitigate the ambiguity of natural language descriptions, we could incorporate additional program specifications such as input-output examples. Input-output examples are also helpful for evaluating the execution accuracy, which considers all semantically correct programs as correct predictions even if they differ from the ground truth. Most Jupyter notebooks from GitHub do not contain sufficient execution information, e.g., many of them load external data for plotting, and the data sources are not public. Therefore, developing techniques to automatically synthesize input-output examples is a promising future direction. Designing new models for code representation learning is another future direction, which could help address the challenge of embedding long code context.

Appendix C

Execution-Guided Neural Program Synthesis

C.1 More Descriptions of the Karel Domain

Figure C.1 presents the grammar specification of the Karel DSL.

Each Karel grid world has a maximum size of 18×18, and is represented as a $16 \times 18 \times 18$ tensor, where each cell of the grid is represented as a 16-dimensional vector corresponding to the features described in Table C.1.

```
Prog p   ::=  def run() : s
Stmt s   ::=  while(b) : s | repeat(r) : s |  s₁ ; s₂ | a
         |    if(b) : s | ifelse(b) : s₁ else : s₂
Cond b   ::=  frontIsClear() | leftIsClear() | rightIsClear
         |    markersPresent() | noMarkersPresent() | not b
Action a ::=  move() | turnRight() | turnLeft()
         |    pickMarker() | putMarker()
Cste r   ::=  0 | 1 | ... | 19
```

Figure C.1: Grammar for the Karel task.

C.2 More Details about the Execution-guided Algorithm

While-statement synthesis algorithm. Algorithm 4 demonstrates the execution-guided algorithm for while-statement synthesis.

Robot facing North
Robot facing East
Robot facing South
Robot facing West
Obstacle
Grid boundary
1 marker
2 markers
3 markers
4 markers
5 markers
6 markers
7 markers
8 markers
9 markers
10 markers

Table C.1: Representation of each cell in the Karel state.

Training dataset construction for supervised learning. Consider a program $P = S_1; ...; S_T; \perp$, where each S_i is in one of the following forms: (1) $S_i \in \mathcal{L}$; (2) **if** $\mathcal{C}$ **then** $B_{\mathbf{t}}$ **else** $B_{\mathbf{f}}$ **fi**; and (3) **while** $\mathcal{C}$ **do** B **end**. For each $S_i \in \mathcal{L}$, we construct a sample of $\langle \{(s_{i-1}^k, O^k)\}_{k=1}^K, \mathcal{S}_i \rangle$ directly.

For $S_i = $ **if** $\mathcal{C}$ **then** $B_{\mathbf{t}}$ **else** $B_{\mathbf{f}}$ **fi**, we first construct a training sample $\langle \{(s_{i-1}^k, O^k)\}_{k=1}^K, \mathbf{if}\ \mathcal{C}\ \mathbf{then} \rangle$. Afterwards, we split the input-output examples into two subsets $\mathcal{I}_{\mathbf{t}} \uplus \mathcal{I}_{\mathbf{f}}$, where for all $(s, O) \in \mathcal{I}_{\mathbf{t}}$, we have $\langle \mathcal{C}, s \rangle \Downarrow$ true; and $\mathcal{C}$ evaluates to false for all $(s, O) \in \mathcal{I}_{\mathbf{f}}$ on the other hand. Then we obtain two derived samples $\langle \mathcal{I}_{\mathbf{t}}, B_{\mathbf{t}}\ \mathbf{else}; S_{i+1}; ...; S_T; \perp \rangle$ and $\langle \mathcal{I}_{\mathbf{f}}, B_{\mathbf{f}}\ \mathbf{fi}; S_{i+1}; ...; S_T; \perp \rangle$, from which we construct training samples respectively using the same approach as discussed above.

In a similar way, we can deal with $S_i = $ **while** $\mathcal{C}$ **do** B **end**. Finally, we include $\langle \{(O^k, O^k)\}_{k=1}^K, \perp \rangle$ in our constructed training set.

C.3   Model Details

Neural Network Architecture

Our neural network architecture can be found in Figure 4.1, which follows the design in [35]. In particular, the IO Encoder is a convolutional neural network to encode the input and output grids, which outputs a 512-dimensional vector for each input-output pair. The decoder

---

Algorithm 4 Execution-guided synthesis (while-statement)

1: **function** EXECWHILE(Γ, $\mathcal{I}$)
2:     $\mathcal{C} \leftarrow \Gamma(\mathcal{I})$
3:     $\mathcal{I}_\text{t} \leftarrow \{(s_\mathbf{i}, s_\mathbf{o}) \in \mathcal{I} | \langle \mathcal{C}, s_\mathbf{i} \rangle \Downarrow \text{true}\}$
4:     $\mathcal{I}_\text{f} \leftarrow \{(s_\mathbf{i}, s_\mathbf{o}) \in \mathcal{I} | \langle \mathcal{C}, s_\mathbf{i} \rangle \Downarrow \text{false}\}$
5:     $B_\text{t} \leftarrow \text{Exec}(\Gamma, \mathcal{I}_\text{t}, \textbf{end}\text{-token})$
6:     $\mathcal{I}'_\text{t} \leftarrow \{(s_\mathbf{new}, s_\mathbf{o}) | (s_\mathbf{i}, s_\mathbf{o}) \in \mathcal{I}_\text{t} \wedge \langle B_\text{t}, s_\mathbf{i} \rangle \Downarrow s_\mathbf{new}\}$
7:     $\mathcal{I} \leftarrow \mathcal{I}'_\text{t} \cup \mathcal{I}_\text{f}$
8:     $\mathcal{S} \leftarrow \textbf{while } \mathcal{C} \textbf{ do } B_\text{t} \textbf{ end}$
9:     **return** $\mathcal{S}, \mathcal{I}$
10: **end function**

---

is a 2-layer LSTM with a hidden size of 256. The embedding size of the program tokens is 256.

Each program is represented as a sequence of tokens $G = [g_1, g_2, ..., g_L]$, where each program token g_i belongs to a vocabulary Σ. At each timestep t, the decoder LSTM generates a program token g_t conditioned on both the input-output pair and the previous program token g_{t-1}, thus the input dimension is 768. Each IO pair is fed into the LSTM individually, and we a max-pooling operation is performed over the hidden states $\{h_t\}_{k=1}^{K}$ of the last layer of LSTM for all IO pairs. The resulted 256-dimensional vector is fed into a softmax layer to obtain a prediction probability distribution over all the 52 possible program tokens in the vocabulary.

Notice that this neural network architecture can also be applied to other program synthesis problems, with modifications of the IO encoder architectures for different formats of input-output pairs. For example, in the domain where input-output examples are text strings, such as FlashFill [98], the IO encoders can be recurrent neural networks (RNNs) [74].

Training Objective Functions

To estimate the parameters θ of the neural network, we first perform supervised learning to maximize the conditional log-likelihood of the referenced programs [201, 74, 35]. In particular, we estimate θ^* such that

$$\theta^* = \arg\max_\theta \prod_{i=1}^{N} p_\theta(\pi_i | \{IO_i^k\}_{k=1}^{K}) = \arg\max_\theta \sum_{i=1}^{N} \log p_\theta(\pi_i | \{IO_i^k\}_{k=1}^{K}) \tag{C.1}$$

Where π_i are the ground truth programs provided in the training set.

When training with reinforcement learning, we leverage the policy gradient algorithm REINFORCE [265] to solve the following objective:

$$\theta^* = \arg\max_\theta \sum_{i=1}^{N} \sum_{G} \log p_\theta(G|\{IO_i^k\}_{k=1}^{K}) R_i(G) \tag{C.2}$$

Where $R_i(G)$ is the reward function to represent the quality of the sampled program G. In our evaluation, we set $R_i(G) = 1$ if G gives the correct outputs for given inputs, and $R_i(G) = 0$ otherwise.

Training Hyper-parameters

We use the Adam optimizer [139] for both the supervised training and the RL training. The learning rate of supervised training is 10^{-4}, and the learning rate of reinforcement learning is 10^{-5}. We set the batch size to be 128 for supervised training, and 16 for RL training.

C.4 Evaluation Details

More Analysis of Evaluation Results

In our evaluation, we observe that while our Exec approach significantly boosts the generalization accuracy, the performance gain of the exact match accuracy is much smaller, and sometimes even negative. We attribute this to the fact that the ground truth program in the Karel benchmark is not always the simplest one satisfying the input-output examples; on the other hand, our approach tends to provide short predictions among all programs consistent with the input-output specification. For example, Figure C.2 shows a predicted program that is simpler than the ground truth, while also satisfies the input-output pairs. Notice that different from the MLE approach in [35], our model is not directly optimized for the exact match accuracy, since the training set not only includes the input-output examples in the original training set, but also the intermediate state pairs, which constitute a larger part of our augmented training set. Meanwhile, in our training set, for state pairs resembling $\{(s_1^k, s_2^k)\}_{k=1}^{K}$ in Figure C.2, it is more common for the ground truth program to be a single "move()" statement than other more complicated ones. Therefore, when training with our approach, the model is less prone to overfitting to the sample distribution of the original dataset, and focuses more on the program semantics captured by the intermediate execution.

More Details of the Ensemble

For different training approaches of a single model, we train 15 models with different random initializations. To do the ensemble, we first sort the 15 models according to the descending order of their generalization accuracies on the validation set, then select the first k models to compute the results of the k-model ensemble. When multiple programs satisfy the ensemble criterion, e.g., with the shortest length for the *Shortest* method, we choose the one from the models with better generalization accuracies on the validation set.

Ground truth: putMarker(); while (markersPresent()): move()

Prediction: putMarker(); move()

States:

$s_0^1 (I^1)$ s_1^1 $s_2^1 (O^1)$

$s_0^2 (I^2)$ s_1^2 $s_2^2 (O^2)$

Figure C.2: An example of the predicted program that generalizes to all input-output examples, but is different from the ground truth. Here, we only include 2 out of 5 input-output examples for simplicity. Notice that the predicted program is simpler than the ground truth.

Figure C.3: Results of the ensemble model trained with our Exec approach. Left: generalization accuracy. Right: exact match accuracy.

Figure C.3 shows the results of the ensemble trained with our Exec approach. Tables C.2 and C.3 show the numerical results of applying ensemble to our Exec approach and Exec + RL approach.

Ensemble	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Exec (S)	40.88%	42.16%	42.64%	43.12%	43.00%	43.36%	44.32%	44.64%	44.72%	45.64%	45.32%	45.60%	45.52%	45.92%	45.84%
Exec (MV)	40.88%	41.24%	41.36%	41.52%	41.96%	41.76%	42.64%	43.12%	43.48%	43.56%	43.92%	44.44%	44.80%	45.32%	45.36%
Exec + RL (S)	39.40%	42.80%	43.56%	43.84%	44.32%	44.96%	45.16%	45.44%	45.52%	46.36%	46.24%	46.04%	46.28%	46.16%	46.04%
Exec + RL (MV)	39.40%	40.76%	41.56%	42.92%	42.84%	43.84%	43.68%	44.36%	44.48%	44.60%	45.24%	45.52%	45.48%	45.64%	45.64%

Table C.2: Exact match accuracy of the ensemble.

Ensemble	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Exec (S)	85.08%	86.44%	87.04%	87.52%	87.76%	88.20%	89.04%	89.72%	90.16%	90.28%	90.68%	91.16%	91.44%	91.40%	91.60%
Exec (MV)	85.08%	86.08%	86.84%	87.36%	87.72%	87.84%	88.68%	89.36%	89.80%	90.00%	90.44%	91.12%	91.48%	91.40%	91.52%
Exec + RL (S)	86.04%	87.20%	88.64%	89.40%	89.64%	90.24%	90.32%	90.44%	90.48%	90.60%	90.96%	91.24%	91.32%	91.64%	91.68%
Exec + RL (MV)	86.04%	86.88%	88.24%	89.16%	89.20%	90.04%	90.08%	90.20%	90.56%	90.68%	91.28%	91.48%	91.76%	91.96%	92.00%

Table C.3: Generalization accuracy of the ensemble.

Appendix D

Latent Execution for Neural Program Synthesis

D.1 Details in Model Architecture

Program Decoder

Our model follows the encoder-decoder framework in prior work on neural program synthesis from input-output examples [74, 35], which includes an encoder for the input-output pairs, and a decoder to synthesize the program.

The program decoder is an LSTM (denoted as LSTM_D), which decodes the program as a token sequence. Let p_{t-1} be the decoded program token at step $t-1$, $E_p(p_{t-1})$ be the embedding vector of p_{t-1}, h_{t-1} be the hidden state of the program decoder at step $t-1$, and $\hat{I}_{t-1}$ and O be the sequences of vectors representing the input list elements and output list elements. We first compute attention vectors over both the input and output lists, following the double attention mechanism in RobustFill:

$$s_t^O = \text{Attention}(h_{t-1}, O), \quad s_t^I = \text{Attention}([h_{t-1}; s_t^O], \hat{I}_{t-1})$$

The notation $[a; b]$ means the concatenation of vectors a and b. Then we calculate the output vector of the program decoder at step t as $h_t = \text{LSTM}_D(h_{t-1}, [E_p(p_{t-1}); s_t^I; s_t^O])$.

Input-Output Encoder. For C program synthesis, our input-output encoder architecture is similar to RobustFill [74]. For each input-output pair, we use two bi-directional LSTMs [115] to encode the input and output lists respectively. To capture the relationship between the input and output lists, the output list encoder computes attention vectors over the input list elements, using the standard attention mechanism [20, 171]. We also employ different encoder architectures for program synthesis tasks with other formats of input-output examples, as discussed in Sec. 5.5.

To capture the required arithmetic operation to convert from the program input to the output, we also include the output of the operation predictor $\hat{op}_t$ for program decoding, and we discuss the details later. Afterwards, the max pooling layer aggregates the representation of different IO pairs to generate a single vector:

$$m_t = \text{MaxPool}_{j \in \{1,2,...,K\}}(\tanh(W[s_t^{I(j)}; s_t^{O(j)}; \hat{op}_t^{(j)}]))$$

Here the superscript (j) indicates that the representation is for the j-th IO pair, and W is a trainable weight matrix.

To facilitate the prediction of long programs, we compute an attention vector over previously generated program tokens as follows:

$$d_t = \text{Attention}(m_t, \{E_p(p_0), E_p(p_1), ..., E_p(p_{t-1})\})$$

Finally, the next token p_t is sampled from $\mathbb{P}[p_t] = \text{Softmax}(V d_t)_{p_t}$ where V is a trainable matrix.

Operation Predictor for Restricted C Domain

Training neural networks for mathematical reasoning is a challenging problem itself [222, 152], and jointly predicting the mathematical calculations and other program operations imposes extra burden on the program decoder. To mitigate the difficulty, we include a pre-computed table as part of the model input, which describes possible mathematical operations to derive an output value given the input number. For example, Fig. 5.2(d) shows that by applying the $O = 2 + I$ operation to the input $I = 2$, the output $O = 4$. For each valid input list value C, we include two operations $O = C + I$ and $O = C - I$ in the table. Then for each operation $O = C + I$, we enumerate all valid integer list values I, and we include the row $[O = C + I, I, O]$ in the table when O is also within our bounded range. In this way, the table covers all possible integer addition and subtraction operations for valid input and output list values.

With the pre-computed table, the operation predictor aims to predict the most possible program operation at the next step. First, we re-use the same embedding matrices as those in the input-output encoder, and compute the embedding vectors for each numerical element in the table. Let R be the number of table rows. For the i-th row, we refer to the embedding vector of the input and output values as $r^{[i]}$ and $c^{[i]}$, respectively. Then we utilize s_t^I and s_t^O to compute the attention weights over the table columns of input and output values as follows:

$$wi_t^{[i]} = \text{AttentionWeight}(s_t^I, \{r^{[i]}|i \in \{1, 2, ..., R\}\})$$

$$wo_t^{[i]} = \text{AttentionWeight}(s_t^O, \{c^{[i]}|i \in \{1, 2, ..., R\}\})$$

Let $op^{[i]}$ be the operation in row i, then the probability of selecting the operation in the i-th row at step t is

$$\mathbb{P}[op_t = op^{[i]}] \propto wi_t^{[i]} \cdot wo_t^{[i]}$$

Let $E_{op}(op)$ be the embedding vector of the operation op, then the operation predictor output is

$$\hat{op}_t = \sum_i \mathbb{P}[op_t = op^{[i]}] E_{op}(op^{[i]})$$

To train the operation predictor, we provide the training supervision at step 0, when no transformation has been applied to the program input:

$$\mathcal{L}_{Op} = \text{Loss}(wi_0^{[i]}, \mathbb{1}[r^{[i]} = \hat{I}_0 = I]) + \text{Loss}(wo_0^{[i]}, \mathbb{1}[c^{[i]} = O]) \tag{D.1}$$

Latent Executor

In RobustFill, the encoder only takes the initial input-output pairs as the input. On the other hand, in recent work on execution-guided program synthesis [47, 238, 295, 85, 197, 193], the execution states of partial programs are leveraged as the model input to guide the subsequent program prediction. However, existing approaches mostly assume that the programs are sequential [295, 85], or require an interpreter of partial programs [47]. To address these limitations, Nye et al. design neural networks to represent the partial program semantics when they are not well-defined [193]. However, they need to train a separate neural module to represent each program operation, thus it is hard to scale beyond domain-specific languages.

In this work, we include another LSTM to approximate the program execution states, denoted as LSTM_E. Let $\hat{I}_{t-1}$ be the input of LSTM_E, which is the program input at step $t-1$. The output of LSTM_E is:

$$\text{Exec}_t = \text{LSTM}_E(h_t, \hat{I}_{t-1})$$

Implementation for Restricted C Domain

For our restricted C domain, the length of Exec_t is the same as $\hat{I}_{t-1}$, i.e., the input list length. Let L be the length of input and output lists. Let $\mathbb{P}[I_t = v]$ be the probability that the execution result at step t is v, then:

$$\mathbb{P}[I_{t,l} = v_l] = \text{Softmax}(W_E \text{Exec}_{t,l})_{v_l}$$

Here the subscript l denotes that the representation is for the l-th list element, and W_E is a trainable weight matrix.

Finally, the approximated execution state $\hat{I}_t$ is the weighted sum of the embedding vectors of all possible program input integers $c \in [-4, 4] \cap \mathbb{Z}$ (where $\mathbb{Z}$ is the set of all integers):

$$\hat{I}_{t,l} = \sum_{c \in [-4,4] \cap \mathbb{Z}} \mathbb{P}[I_{t,l} = c] E_{io}(c)$$

Here $E_{io}(c)$ denotes the embedding vector of the list value c. At the next program decoding step, $\hat{I}_t$ will be fed into the encoder to replace the previous input list $\hat{I}_{t-1}$.

Implementation for Karel Domain

Similar to our restricted C domain, in our latent executor implementation for Karel domain, $\hat{I}_{t,l}$ is also the weighted sum of all possible execution states. Each Karel state describes the following variables: (1) $(\text{robot}_X, \text{robot}_Y)$ denotes the position of the Karel robot, where $0 \leq \text{robot}_X, \text{robot}_Y < 18$; (2) $\text{robot}_{dir} \leq \{\text{North, South, West, East}\}$ denotes the robot orientation at $(\text{robot}_X, \text{robot}_Y)$; and (3) the number of markers in each grid. Therefore, we train 3 predictors on top of LSTM_E to predict these variables: (1) a trainable layer that outputs a (18×18)-dimensional vector, representing the robot position; (2) a trainable layer that outputs a 4-dimensional vector, representing the robot orientation; and (3) an LSTM that generates an 11-dimensional vector at each step, representing the number of markers in each grid. We apply the softmax to all output vectors to obtain the probability distributions of different variables.

Afterward, we combine the outputs of the predictors to construct a $16 \times 18 \times 18$-dimensional vector representing the Karel state, according to Table C.1, with the value of each dimension in $[0, 1]$. Note that Karel programs can not change the grid boundary and obstacles, thus we apply a mask on the predicted intermediate execution states to ensure that the features representing the grid boundary and obstacles remain the same, which are the last 2 dimensions described in Table C.1.

D.2   Implementation Details

All encoders and decoders in our models are 2-layer bi-directional LSTMs with the hidden size of 512. The embedding size is 1024. We use the Adam optimizer [139] for training. The learning rate starts from 1e-3, and is decayed by 0.9 for every 6000 timesteps. The batch size is 8. The training converges in 200K batch updates. The norm for gradient clipping is 5.0. All models are trained on a single GPU. The beam size is 64 for evaluating the model performance, and is 8 for iterative retraining due to the large size of the training set.

About the implementation of the Property Signatures [196], we further illustrate the key difference between our adaption for the restricted C domain and the original implementation in [196] with the following example. Suppose an input-output pair is $([-4, 3, 1, 2, 1], [-4, 3, 3, 3, 3])$, when the feature is "Input == Output?", the corresponding property signature is "False" according to the implementation in [196], while the signature is "[True, True, False, False, False]" in our adapted implementation. Compared to the original implementation of property signatures, our adaptation better reveals which specific list elements are manipulated in

Table D.1: Results of iterative retraining on Karel dataset.

Iters	100%	10%	20%	30%	40%	50%
		Generalization Accuracy				
1	86.04%	70.92%	75.16%	78.84%	80.88%	82.08%
2	89.28%	76.20%	78.40%	81.08%	82.40%	83.40%
3	89.36%	78.12%	81.20%	83.68%	84.24%	86.32%
		Exact Match Accuracy				
1	39.40%	36.20%	37.20%	38.36%	40.20%	40.04%
2	41.56%	37.24%	37.28%	39.24%	39.72%	39.16%
3	41.16%	36.56%	38.16%	38.68%	38.72%	39.64%

Table D.2: Results of iterative retraining on C dataset.

Iters	100%	10%	20%	30%	40%	50%
1	55.2%	11.9%	26.4%	39.1%	45.2%	48.5%
2	56.0%	39.6%	43.9%	48.7%	51.9%	54.1%
3	56.5%	41.7%	44.4%	49.4%	52.8%	54.4%

the program. This modification makes our implementation of property signatures a much stronger baseline for the restricted C domain, because our C programs do not always perform the same manipulation steps over all elements in the input list, and sometimes change the values of only a subset of the input numbers.

D.3 More Results of Iterative Retraining

Figure D.1 presents more examples of predicted correct programs that are more concise than the randomly generated ground truth programs on C dataset.

Figure D.2 presents more examples of predicted correct programs that are more concise than the randomly generated ground truth programs on Karel dataset. Note that the predicted Karel program is not semantically equivalent to the annotated ground truth in many cases. The main reason is because the randomly generated ground truth program might include redundant branching statements, i.e., the conditions always evaluate to true or false for all program inputs in the specification and the held-out test cases.

We present the numerical results of iterative retraining on Karel and C benchmarks in Table D.1 and Table D.2 respectively.

```
I1: [2, 4, 1, 2, -3]
O1: [2, 4, 3, 2, -3]
I2: [1, 0, 1, -3, 4]
O2: [1, 0, 3, -3, 4]
I3: [2, 2, -4, 2, 0]
O3: [2, 2, 3, 2, 0]
I4: [0, -2, 3, 1, 3]
O4: [0, -2, 3, 1, 3]
I5: [-2, 1, 4, 0, 0]
O5: [-2, 1, 3, 0, 0]
```

```c
int * func_1(int a[])
{
    int p_0 = 4;
    int l_7 = 2;
    int l_8 = 4;
    a[l_7] = 3;
    a[l_8] = a[p_0];
    return a;
}
```

```c
int * func_1(int a[])
{
    int p_0 = 2;
    a[p_0] = 3;
    return a;
}
```

```c
int * func_1(int a[])
{
    int p_0 = 2;
    int l_10 = 0;
    int l_1 = 4;
    l_10 = 2;
    for (p_0 = 2; p_0 >= 1; p_0--)
    {
        a[p_0] = 3;
        a[p_0] = 2;
        if (a[p_0])
            break;
        a[p_0] = a[l_1];
        a[p_0]++;
    }
    return a;
}
```

```
I1: [3, 1, 3, -2, -4]
O1: [3, 1, 2, -2, -4]
I2: [2, 0, -1, -1, 3]
O2: [2, 0, 2, -1, 3]
I3: [2, 0, -1, 4, 0]
O3: [2, 0, 2, 4, 0]
I4: [-2, -1, 3, 2, -4]
O4: [-2, -1, 2, 2, -4]
I5: [-4, 0, 3, 0, 1]
O5: [-4, 0, 2, 0, 1]
```

```c
// Training on random programs
int * func_1(int a[])
{
    int p_0 = 2;
    int l_7 = 2;
    a[l_7] = 2;
    return a;
}

// After iterative retraining
int * func_1(int a[])
{
    int p_0 = 2;
    a[p_0] = 2;
    return a;
}
```

```
I1: [0, 4, 0, 4, 2]
O1: [0, 4, 0, 1, 1]
I2: [4, 0, 1, 1, 4]
O2: [4, 0, 1, 1, 1]
I3: [3, 2, 3, 0, 0]
O3: [3, 2, 3, 1, 1]
I4: [1, 1, 4, 0, 4]
O4: [1, 1, 4, 1, 1]
I5: [1, 3, 0, 1, 1]
O5: [1, 3, 0, 1, 1]
```

```c
int * func_1(int a[])
{
    int p_0 = 0;
    int l_10 = 3;
    for (p_0 = 4; p_0 >= 0; p_0--)
    {
        a[p_0] = 3;
        a[p_0] = a[p_0];
        a[p_0] = 1;
        if (a[p_0])
            break;
    }
    a[l_10] = a[l_10];
    a[l_10] = a[p_0];
    return a;
}
```

```c
int * func_1(int a[])
{
    int p_0 = 4;
    for (p_0 = 3; p_0 <= 4; p_0++)
    {
        a[p_0] = 1;
    }
    return a;
}
```

```
I1: [0, 3, -1, 0, 0]
O1: [4, 3, -1, 4, 4]
I2: [4, -3, 3, 4, 2]
O2: [4, -3, 3, 4, 4]
I3: [-4, 1, 0, 4, -2]
O3: [4, 1, 0, 4, 4]
I4: [0, 4, 3, 0, 4]
O4: [4, 4, 3, 4, 4]
I5: [2, 2, 0, 3, 2]
O5: [4, 2, 0, 4, 4]
```

```c
int * func_1(int a[])
{
    int p_0 = 0;
    int l_11 = 3;
    for (p_0 = 2; p_0 >= 1; p_0--)
    {
        for (int p_1 = 4; p_1 >= 3; p_1--)
        {
            a[p_1] = 4;
        }
    }
    a[p_0] = a[l_11];
    return a;
}
```

```c
int * func_1(int a[])
{
    int p_0 = 3;
    int l_7 = 0;
    a[l_7] = 4;
    for (p_0 = 4; p_0 >= 3; p_0--)
    {
        a[p_0] = 4;
    }
    return a;
}
```

Figure D.1: More examples of predicted correct programs that are more concise than the randomly generated ground truth programs on C dataset. Left: input-output examples. Middle: the randomly generated ground truth program. Right: the predicted programs. Unless otherwise specified, the predicted programs come from the model trained on random programs.

```
def run():
  repeat (5):
    ifelse (rightIsClear):
      move
    else:
      move
    putMarker
```

```
def run():
  repeat (5):
    move
    putMarker
```

```
def run():
  move
  turnRight
  ifelse (noMarkersPresent):
    repeat (2):
      putMarker
  else:
    pickMarker
  repeat (5):
    turnRight
```

```
def run():
  move
  turnLeft
  turnLeft
  ifelse (markersPresent):
    pickMarker
  else:
    putMarker
    putMarker
```

```
def run():
  pickMarker
  move
  ifelse (not rightIsClear):
    putMarker
    move
  else:
    move
    putMarker
    while (not rightIsClear):
      move
      putMarker
  putMarker
  turnRight
  move
```

```
def run():
  pickMarker
  move
  move
  putMarker
  putMarker
  turnRight
  move
```

```
def run():
  move
  turnRight
  repeat (5):
    pickMarker
  putMarker
```

```
def run():
  move
  repeat (4):
    pickMarker
  turnRight
```

```
def run():
  move
  ifelse (markersPresent):
    ifelse (frontIsClear):
      putMarker
    else:
      pickMarker
  else:
    while (rightIsClear):
      turnRight
  repeat (2):
    repeat (2):
      putMarker
    turnLeft
```

```
def run():
  move
  while (leftIsClear):
    turnLeft
  repeat (4):
    putMarker
```

```
def run():
  putMarker
  move
  ifelse (not leftIsClear):
    putMarker
  else:
    turnRight
  if (rightIsClear):
    pickMarker
```

```
def run():
  putMarker
  move
  putMarker
  if (rightIsClear):
    pickMarker
```

Figure D.2: Examples of predicted correct programs that are more concise than the randomly generated ground truth programs on Karel dataset. 1st and 3rd columns: the randomly generated ground truth programs. 2nd and 4th: the corresponding predicted programs. The predictions come from the model trained on random programs.

Appendix E

Tree-to-Tree Neural Networks for Program Translation

E.1   Hyper-parameters of Neural Network Models

	Seq2seq	Seq2tree	Tree2seq	Tree2tree
Batch size	100	20	100	100
Number of RNN layers	3	1	1	1
Encoder RNN cell	LSTM	LSTM	Tree LSTM	Tree LSTM
Decoder RNN cell	LSTM			
Initial learning rate	0.005			
Learning rate decay schedule	Decay the learning rate by a factor of 0.8× when the validation loss does not decrease for 500 mini-batches			
Hidden state size	256			
Embedding size	256			
Dropout rate	0.5			
Gradient clip threshold	5.0			
Weights initialization	Uniformly random from [-0.1, 0.1]			

Table E.1: Hyper-parameters chosen for each neural network model.

We present the hyper-parameters of different neural networks in Table E.1. These hyper-parameters are chosen to achieve the best accuracy on the development set through a grid search.

E.2 More Statistics of the Datasets

We present more detailed statistics of the datasets for the CoffeeScript-JavaScript task and the translation of real-world projects from Java to C# in Table E.2 and E.3 respectively.

	CJ-(A/B)S	CJ-(A/B)L
Average input length (P)	10	20
Minimal output length (P)	23	33
Maximal output length (P)	151	311
Average output length (P)	44	69
Minimal input length (T)	34	69
Maximal input length (T)	61	111
Average input length (T)	48	85
Minimal output length (T)	38	73
Maximal output length (T)	251	531
Average output length (T)	71	129

Table E.2: Statistics of the datasets used for the CoffeeScript-JavaScript task.

Project	# of matched methods
Lucene [170]	5,516
POI [208]	3,153
Itext [121]	3,079
JGit [127]	2,780
JTS [132]	2,003
Antlr [15]	465
Total	16,996

Table E.3: Statistics of the Java to C# dataset.

E.3 More Results on the CoffeeScript-JavaScript Task

Besides the program accuracy, we also measure the token accuracy of different approaches, which is the percentage of the tokens that are exactly the same as the ground truth. This metric is a finer-grained measurement of the correctness, thus provides some additional insights of the performance of different models.

Table E.4 shows the token accuracy of different approaches for the translation between CoffeeScript and JavaScript.

	Tree2tree			Seq2seq				Seq2tree		Tree2seq	
	T→T	T→T (-PF)	T→T (-Attn)	P→P	P→T	T→P	T→T	P→T	T→T	T→P	T→T
CoffeeScript to JavaScript translation											
CJ-AS	**99.97%**	**99.97%**	56.21%	93.51%	92.30%	95.46%	95.05%	93.29%	95.94%	98.96%	98.09%
CJ-BS	**99.98%**	**99.98%**	47.54%	99.08%	87.51%	99.11%	96.14%	98.31%	98.09%	99.27%	98.10%
CJ-AL	**99.37%**	98.16%	32.99%	85.84%	25.65%	19.13%	36.18%	95.64%	94.74%	94.18%	84.71%
CJ-BL	**99.36%**	99.27%	31.80%	80.22%	63.49%	87.27%	79.85%	94.09%	94.64%	93.85%	78.07%
JavaScript to CoffeeScript translation											
JC-AS	**99.14%**	98.81%	65.42%	88.44%	96.27%	88.46%	98.34%	98.20%	99.06%	86.93%	98.36%
JC-BS	**98.84%**	98.18%	55.22%	86.85%	97.92%	85.98%	98.09%	96.93%	**98.84%**	84.81%	97.94%
JC-AL	**96.95%**	92.65%	42.23%	88.09%	95.94%	87.19%	95.04%	93.51%	96.59%	84.57%	94.63%
JC-BL	**96.48%**	92.49%	39.89%	87.31%	94.12%	85.70%	96.24%	94.79%	96.33%	83.03%	94.68%

Table E.4: Token accuracy of different approaches for translation between CoffeeScript and JavaScript.

E.4 Grammar for the CoffeeScript-JavaScript Task

The grammar used to generate the CoffeeScript-JavaScript dataset, which is a subset of the core CoffeeScript grammar, is provided in Figure E.1.

E.5 Evaluation on the Synthetic Task

In the following, we discuss our synthetic translation task from an imperative language to a functional language.

Evaluation Setup

For the synthetic task, we design an imperative source language and a functional target language. Such a design makes the source and target languages use different programming paradigms, so that the translation can be challenging. Figure E.2 illustrates an example of the translation, which demonstrates that a for-loop is translated into a recursive function. We manually implement a translator, which is used to acquire the ground truth. The grammar specifications of the source language (FOR language) and the target language (LAMBDA language) are provided in Figure E.3 and Figure E.4 respectively. The python source code to implement the translator from a FOR program to a LAMBDA program is provided in Figure E.5.

To build the dataset, similar to the CoffeeScript-JavaScript task, we randomly generate 100,000 pairs of source and target programs for training, 10,000 pairs as the development set, and 10,000 pairs for testing. We guarantee that there is no overlap among training, development and test sets, and all samples are unique in the dataset. More statistics of the dataset can be found in Table E.6.

```
        <Expr>  ::=  <Var>
                 |   <Const>
                 |   <Expr> +  <Var>
                 |   <Expr> +  <Const>
                 |   <Expr> *  <Var>
                 |   <Expr> *  <Const>
                 |   <Expr> ==  <Var>
                 |   <Expr> ==  <Const>
      <Simple>  ::=  <Var> = <Expr>
                 |   <Expr>
     <IfShort>  ::=  <Simple> if <Expr>
                 |   <IfShort> if <Expr>
  <WhileShort>  ::=  <Simple> while <Expr>
                 |   <WhileShort> while <Expr>
<ShortStatement>  ::=  <Simple> | <IfShort> | <WhileShort>
   <Statement>  ::=  <ShortStatement>
                 |   if <Expr> <br> <indent+> <Block> <indent->
                 |   while <Expr> <br> <indent+> <Block> <indent->
                 |   if <Expr> <br> <indent+> <Block> <indent-> <br>
                     else <br> <indent+> <Block> <indent->
                 |   if <Expr> then <ShortStatement> else <ShortStatement>
       <Block>  ::=  <Statement>
                 |   <Block> <br> <Statement>
```

Figure E.1: A subset of the CoffeeScript grammar used to generate the CoffeeScript-JavaScript dataset. Here, `<br>` denotes the newline character.

Results on the Synthetic Task

We create two datasets for the synthetic task: one with an average length of 20 (*SYN-S*) and the other with an average length of 50 (*SYN-L*). Here, the length of a program indicates the number of tokens in the source program.

We present the results in Table E.5. Our observations are consistent with the results of the CoffeeScript-JavaScript task: our tree2tree model outperforms all baseline models; all models perform worse on longer inputs; both the attention and the parent attention feeding mechanisms boost the performance of our tree2tree model significantly.

```
Source program        Target program
for i=1; i<10; i+1 do  letrec f i =
    if x>1 then              if i<10 then
        y=1                          let _ = if x>1 then
    else                                 let y=1 in ()
        y=2                          else let y=2 in ()
    endfor                       in f i+1
                             else ()
                         in f 1
```

Figure E.2: An example of the translation for the synthetic task.

	Tree2tree			Seq2seq				Seq2tree		Tree2seq	
	T→T	T→T (-PF)	T→T (-Attn)	P→P	P→T	T→P	T→T	P→T	T→T	T→P	T→T
	Token accuracy										
SYN-S	**99.99%**	99.95%	55.60%	99.75%	99.59%	99.90%	99.73%	99.70%	99.51%	99.88%	99.82%
SYN-L	**99.60%**	96.68%	34.48%	68.31%	45.28%	67.37%	35.01%	96.95%	97.41%	97.08%	95.88%
	Program accuracy										
SYN-S	**99.76%**	98.61%	0%	97.92%	97.35%	98.38%	98.18%	96.14%	98.01%	98.51%	98.36%
SYN-L	**97.50%**	57.42%	0%	12.19%	0%	9.19%	0%	67.34%	68.11%	91.35%	87.84%

Table E.5: Token accuracy and program accuracy of different approaches for the synthetic task.

	SYN-S	SYN-L
Average input length (P)	20	50
Minimal output length (P)	22	46
Maximal output length (P)	44	96
Average output length (P)	30	71
Minimal input length (T)	40	100
Maximal input length (T)	56	134
Average input length (T)	49	111
Minimal output length (T)	41	90
Maximal output length (T)	82	177
Average output length (T)	55	133

Table E.6: Statistics of the datasets used for the synthetic task.

```
   <Expr>  ::=  <Var>
                 |  <Const>
                 |  <Expr> + <Var>
                 |  <Expr> + <Const>
                 |  <Expr> − <Var>
                 |  <Expr> − <Const>
    <Cmp>  ::=  <Expr> == <Expr>
                 |  <Expr> > <Expr>
                 |  <Expr> < <Expr>
 <Assign>  ::=  <Var> = <Expr>
     <If>  ::=  if <Cmp> then <statement>
                else <statement> endif
    <For>  ::=  for <Var> = <Expr> ;
                <Cmp> ; <Expr> do
                <Statement> endfor
 <Single>  ::=  <Assign> | <If> | <For>
    <Seq>  ::=  <Single> ; <Single>
                 |  <Seq> ; <Single>
<Statement>  ::=  <Seq> | <Single>
```

Figure E.3: Grammar for the source language FOR in the synthetic task.

```
   <Unit>  ::=  ()
    <App>  ::=  <Var> <Expr>
                 |  <App> <Expr>
   <Expr>  ::=  <Var>
                 |  <Expr> + <Var>
                 |  <Expr> − <Var>
    <Cmp>  ::=  <Expr> == <Expr>
                 |  <Expr> > <Expr>
                 |  <Expr> < <Expr>
   <Term>  ::=  <LetTerm> | <Expr> | <Unit>
                 |  <IfTerm> | <App>
<LetTerm>  ::=  let <Var> = <Term> in <Term>
                 |  letrec <Var> <Var> = <Term>
                in <Term>
 <IfTerm>  ::=  if <Cmp> then <Term>
                else <Term>
```

Figure E.4: Grammar for the target language LAMBDA in the synthetic task.

```python
def translate_from_for(self, ast):
    if type(ast) == type([]):
        if ast[0] == '<SEQ>':
            t1 = self.translate_from_for(ast[1])
            t2 = self.translate_from_for(ast[2])
            if t1[0] == '<LET>' and t1[-1] == '<UNIT>':
                t1[-1] = t2
                return t1
            else:
                return ['<LET>', 'blank', t1, t2]
        elif ast[0] == '<IF>':
            cmp = ast[1]
            t1 = self.translate_from_for(ast[2])
            t2 = self.translate_from_for(ast[3])
            return ['<IF>', cmp, t1, t2]
        elif ast[0] == '<FOR>':
            var = self.translate_from_for(ast[1])
            init = self.translate_from_for(ast[2])
            cmp = self.translate_from_for(ast[3])
            inc = self.translate_from_for(ast[4])
            body = self.translate_from_for(ast[5])
            tb = ['<LET>', 'blank', body, ['<APP>', 'func', inc]]
            func_body = ['<IF>', cmp, tb, '<UNIT>']
            translate = ['<LETREC>', 'func', var, func_body,
                         ['<APP>', 'func', inc]]
            return translate
        elif ast[0] == '<ASSIGN>':
            return ['<LET>', ast[1], ast[2], '<UNIT>']
        elif ast[0] == '<Expr>':
            return ast
        elif ast[0] == '<Op+>':
            return ast
        elif ast[0] == '<Op->':
            return ast
        elif ast[0] == '<CMP>':
            return ast
    else:
        return ast
```

Figure E.5: The Python code to translate a FOR program into a LAMBDA program in the synthetic task.

Appendix F

Neural Rewriter for Code Optimization and beyond

F.1 More Details of the Dataset

Expression Simplification

Figure F.1 presents the grammar of Halide expressions in our evaluation. We use the random pipeline generator in the Halide repository to build the dataset [1]. Table F.1 presents the statistics of the datasets.

Job Scheduling

Description of different resource distributions. For each job j, we define *dominant resources* d_{dom} as the resources with $0.5 \leq \rho_{jd_{\mathrm{dom}}} \leq 1$, and *auxiliary resources* d_{aux} as those with $0.1 \leq \rho_{jd_{\mathrm{aux}}} \leq 0.2$. We refer to a job with both dominant and auxiliary resources as a job with *non-uniform resources*. We also evaluate on workloads including only jobs with *uniform resources*, where each job only includes either dominant resources or auxiliary resources.

Vehicle Routing

Our data generation follows the setup in [186, 142]. The positions of the depot and customer nodes are uniformly randomly sampled from the unit square $[0, 1] \times [0, 1]$. Each node is denoted as $v_j = ((x_j, y_j), \delta_j)$, where (x_j, y_j) is the position, and δ_j is the resource demand. We set $\delta_0 = 0$ for the depot (i.e., node 0), and $\delta_j \in \{1, 2, ..., 9\}$ for customer nodes (i.e., $j > 0$).

---

[1]`https://github.com/halide/Halide/tree/new_autoschedule_with_new_simplifier/apps/random_pipeline`.

```
      <Expr>  ::=  <AlgExpr> | <BoolExpr>
  <BoolExpr>  ::=  <AlgExpr> < <AlgExpr>
               |   <AlgExpr> <= <AlgExpr>
               |   <AlgExpr> == <AlgExpr>
               |   (!<BoolExpr>)
               |   (<BoolExpr> && <BoolExpr>)
               |   (<BoolExpr> —— <BoolExpr>)
   <AlgExpr>  ::=  <Term>
               |   (<AlgExpr> + <Term>)
               |   (<AlgExpr> - <Term>)
               |   (<AlgExpr> * <Term>)
               |   (<AlgExpr> / <Term>)
               |   (<AlgExpr> % <Term>)
      <Term>  ::=  <Var> | <Const>
               |   max(<AlgExpr>, <AlgExpr>)
               |   min(<AlgExpr>, <AlgExpr>)
               |   select(<BoolExpr>, <AlgExpr>, <AlgExpr>)
```

Figure F.1: Grammar of the Halide expressions in our evaluation. "select $(c, e1, e2)$" means that when the condition c is satisfied, this term is equal to $e1$, otherwise is equal to $e2$. In our dataset, all constants are integers ranging in $[-1024, 1024]$, and variables are from the set $\{v0, v1, ..., v12\}$.

Number of expressions in the dataset	Length of expressions	Size of expression parse trees
Total: 1.36M	Average: 106.84	Average: 27.39
Train/Val/Test: 1.09M/136K/136K	Min/Max: 10/579	Min/Max:3/100
$Train_{\leq 20}$: 17K	Average: 16.76	Average: 4.66
$Train_{\leq 30}$: 48K	Average: 22.91	Average: 6.43
$Train_{\leq 50}$: 170K	Average: 35.62	Average: 10.18
$Train_{\leq 100}$: 588K	Average: 63.49	Average: 18.72
$Test_{>100}$: 53K	Average: 142.22	Average: 42.20

Table F.1: Statistics of the dataset for expression simplification.
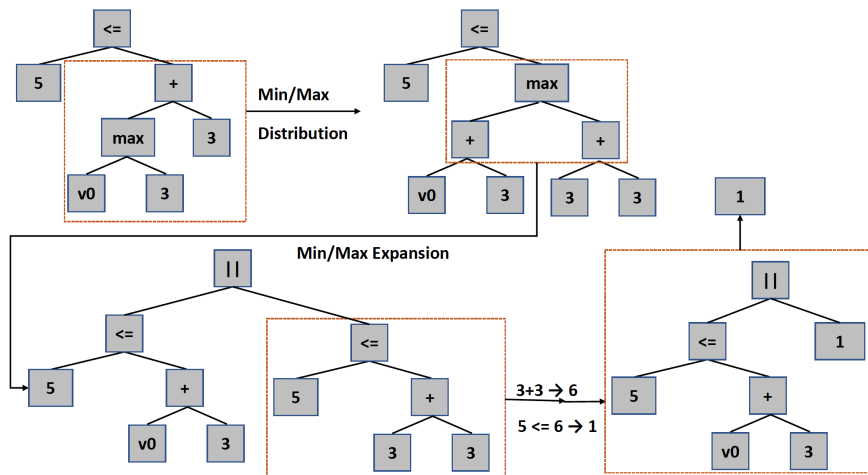
Figure F.2: An example of the rewriting process for Halide expressions. The initial expression is $5 \leq max(v0, 3) + 3$, which could be reduced to 1, i.e., $True$.

F.2 More Details on the Rewriting Ruleset

More Details for Expression Simplification Problem

The ruleset implemented in the Halide rule-based rewriter can be found in their public repository [2].

More discussions about the uphill rules. A commonly used type of uphill rules is "min/max" expansion, e.g., $\texttt{min}(a, b) < c \rightarrow a < c || b < c$. Dozens of templates in the ruleset of the Halide rewriter are describing conditions when a "min/max" expression could be simplified. Notice that although applying this rewriting rule has no benefit in most cases, since it will increase the expression length, it is necessary to include it in the ruleset, because when either $a < c$ or $b < c$ is always true, expanding the "min" term could reduce the entire expression to a tautology, which ends up simplifying the entire expression. Figure F.2 shows an example of the rewriting process using uphill rules properly.

More Details for Job Scheduling Problem

Algorithm 5 describes a single rewriting step for job scheduling problem.

---

[2] `https://github.com/halide/Halide`.

Algorithm 5 Algorithm of a Single Rewriting Step for Job Scheduling Problem

---

1: **function** REWRITE($v_j, v_{j'}, s_t$)
2:     **if** $C_{j'} < A_j$ or $C_{j'} == B_j$ **then**
3:         **return** s_t
4:     **end if**
5:     **if** $j' \neq 0$ **then** $B'_j = C_{j'}$ **else** $B'_j = A_j$ **fi**
6:     $C'_j = B'_j + T_j$
7:
8:     //Resolve potential resource occupation overflow within $[B'_j, C'_j]$
9:     J = all jobs in s_t except v_j that are scheduled within $[B'_j, C'_j]$
10:     Sort J in the topological order
11:     **for** $v_i \in J$ **do**
12:         B'_i = the earliest time that job v_i can be scheduled
13:         $C'_i = B'_i + T_i$
14:     **end for**
15:     For $v_i \notin J$, $B'_i = B_i$, $C'_i = C_i$
16:     $s_{t+1} = \{(B'_i, C'_i)\}$
17:     **return** s_{t+1}
18: **end function**

F.3   More Details on Model Architectures

Model Details for Expression Simplification

Input embedding.   Notice that in this problem, each non-terminal has at most 3 children. Thus, let x be the embedding of a non-terminal, $(h_L, c_L), (h_M, c_M), (h_R, c_R)$ be the LSTM states maintained by its children nodes, the LSTM state of the non-terminal node is computed as

$$(h, c) = \text{LSTM}(([h_L; h_M; h_R], [c_L; c_M; c_R]), x) \tag{F.1}$$

Where $[a; b]$ denotes the concatenation of vectors a and b. For non-terminals with less than 3 children, the corresponding LSTM states are set to be zero. We use d to represent the size of h and c, i.e., the hidden size of the LSTM.

Input representation.   For each sub-tree ω_i, its input to both the score predictor and the rule-picking policy is represented as a $2d$-dimensional vector $[h_0; h_i]$, where h_0 is the embedding of the root node encoding the entire tree. The reason why we include h_0 in the input is that looking at the sub-tree itself is sometimes insufficient to determine whether it is beneficial to perform the rewriting. For example, consider the expression $max(a, b) + 2 < a + 2$, by looking at the sub-expression $max(a, b) + 2$ itself, it does not seem necessary to rewrite it as $max(a + 2, b + 2)$. However, given the entire expression, we can observe that this rewriting is an important step towards the simplification, since the resulted expression
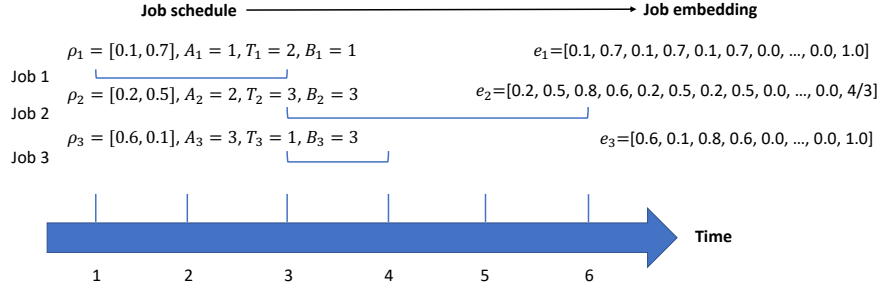
Figure F.3: An example to illustrate the job embedding approach for the job scheduling problem.
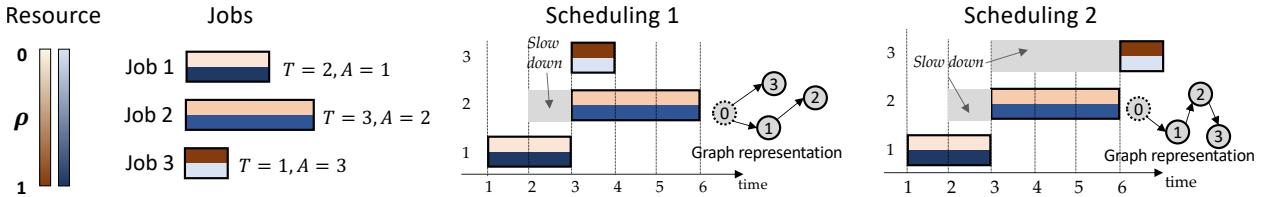

Figure F.4: An example to illustrate two possible job schedules on a single machine and their corresponding graph representations. Node 0 was added to represent the start of the scheduling process. For multiple machines, multiple node 0 will be added.

$max(a + 2, b + 2) < a + 2$ could be reduced to *False*. We have tried other approaches of combining the parent information into the input, but we find that including the embedding of the entire tree is the most efficient way.

Score predictor. The score predictor is an L_P-layer fully connected network with a hidden size of N_P. For each sub-tree ω_i, its input to the score predictor is represented as a $2d$-dimensional vector $[h_0; h_i]$, where h_0 embeds the entire tree.

Rule selector. The rule selector is an L_S-layer fully connected network with the hidden size N_S, and its input format is the same as the score predictor. A $|\mathcal{U}|$-dimensional softmax layer is used as the output layer.

More Details for Job Scheduling Problem

Job embedding. We embed each job into a $(D \times (T_{max} + 1) + 1)$-dimensional vector e_j, where T_{max} is the maximal duration of a job. This vector encodes the information of the job attributes and the machine status during its execution. We describe the details of job embedding as follows. Consider a job $v_j = (\boldsymbol{\rho}_j, A_j, T_j)$. We denote the amount of resources occupied by all jobs at each timestep t as $\boldsymbol{\rho}'_t = (\rho'_{t1}, \rho'_{t2}, ..., \rho'_{tD})$. Each job v_j is represented as

a $(D \times (T_{max} + 1) + 1)$-dimensional vector, where the first D dimensions of the vector are $\boldsymbol{\rho}_j$, representing its resource requirement. The following $D \times T_j$ dimensions of the vector are the concatenation of $\boldsymbol{\rho}'_{B_j}, \boldsymbol{\rho}'_{B_j+1}, ..., \boldsymbol{\rho}'_{B_j+T_j-1}$, which describes the machine usage during the execution of the job v_j. When $T_j < T_{max}$, the following $D \times (T_{max} - T_j)$ dimensions are zero. The last dimension of the embedding vector is the slowdown of the job in the current schedule. We denote the embedding of each job v_j as e_j. The embedding of the machine (i.e., v_0) is a zero vector $e_0 = \mathbf{0}$. Figure F.3 shows an example of our job embedding approach, and Figure F.4 illustrates an example of the graph construction.

Model specification. To encode the graphs, we extend the Child-Sum Tree-LSTM architecture in [244], which is similar to the DAG-structured LSTM in [293]. Specifically, for a job v_j, suppose $(h_1, c_1), (h_2, c_2), ..., (h_p, c_p)$ are the LSTM states of all parents of v_j, then its LSTM state is:

$$(h, c) = \text{LSTM}((\sum_{i=1}^{p} h_i, \sum_{i=1}^{p} c_i), e_j) \tag{F.2}$$

For each node, the d-dimensional hidden state h is used as the embedding for other two components.

Score predictor. This component is an L_P-layer fully connected neural network with a hidden size of N_P, and the input to the predictor of job v_j is h_j.

Rule selector. The rewriting rules are equivalent to moving the current job v_j to be a child of another job $v_{j'}$ or v_0 in the graph, which means allocating job v_j after job $v_{j'}$ finishes or at its arrival time A_j. Thus, the input to the rule selector not only includes h_j, but also $h_{j'}$ of all other $v_{j'}$ that could be used for rewriting. The rule selector has two modules. The first module is an L_S-layer fully connected neural network with a hidden size of N_S. For each job v_j, let N_j be the number of jobs that could be the parent of v_j, and $\{v_{j'_k}\}$ denotes the set of such jobs. For each $v_{j'_k}$, the input is $[h_j; h_{j'_k}]$, and this module computes a d-dimensional vector h'_k to encode such a pair of jobs. The second module of the rule selector is another L_S-layer fully connected neural network with a hidden size of N_S. For this module, the input is a $(|\mathcal{U}| \times d)$-dimensional vector $[h'_1; h'_2; ...; h'_{|\mathcal{U}|}]$, where $|\mathcal{U}| = 2W$. When $N_j < |\mathcal{U}|$, $h'_{N_j+1}, h'_{N_j+2}, ..., h'_{|\mathcal{U}|}$ are set to be zero. The output layer of this module is a $|\mathcal{U}|$-dimensional softmax layer, which predicts the probability of each different move of v_j.

More Details for Vehicle Routing Problem

Node embedding. We embed each node into a 7-dimensional vector e_j. This vector encodes the information of the node position, node resource demand, and the current status of the vehicle. We describe the details of node embedding as follows. Consider a node $v_j = ((x_j, y_j), \delta_j)$, where (x_j, y_j) is the position, and δ_j is the resource demand. We set $\delta_0 = 0$ for the depot (i.e., node 0). Denote Cap as the vehicle capacity. The first three dimensions

of e_j are x_j, y_j, and δ_j/Cap. The next three dimensions of e_j are the coordinates of the node visited at the previous step (set as the depot position for the first visited node) and the Euclidean distance between v_j and the previous node. The last dimension is the amount of remaining resources carried by the vehicle at the current step, which is also normalized by the vehicle capacity.

Score predictor. This component is an L_P-layer fully connected neural network with a hidden size of N_P, and the input to the predictor of the node v_j is h_j, where h_j is the output of the bi-directional LSTM used to encode each node in the route.

Rule selector. The rewriting rules are equivalent to moving a node in the route v_j after another node $v_{j'}$, similar to the job scheduling setting. However, different from job scheduling, the number of such nodes $v_{j'}$ varies among different problems. Thus, we train an attention module to select $v_{j'}$, with a similar design to the pointer network [253].

Model hyper-parameters

For both the expression simplification and job scheduling tasks, $L_S = L_P = 1$. For the vehicle routing task, $L_S = L_P = 2$. For all the three tasks, $N_S = N_P = 256$, $d = 512$.

F.4 More Results for Job Scheduling Problem

We observe that while OR-tools is a high-performance solver for generic combinatorial optimization problems, it is less effective than both heuristic-based scheduling algorithms and neural network approaches on our job scheduling problem, especially with more resource types. After looking into the schedules computed by OR-tools, we find that they often prioritize long jobs over short jobs, while swapping the scheduling order between them would clearly decrease the job waiting time. On the other hand, both our neural rewriter and heuristic algorithms based on the job length would usually schedule short jobs very soon after their arrival, which results in better schedules.

Table F.2 and F.3 present the results of ablation study on job frequency and resource distribution respectively.

To examine how the initial schedules affect the final results, besides earliest-job-first schedules, we also evaluate initial schedules with different average slowdown. Specifically, for each job sequence, we generate different initial schedules by randomly allocating one job at a time.

In Table F.4, we present the results with $D = 20$ types of resources. For each job sequence, we randomly generate 10 different initial schedules. We can observe that although the effectiveness of initial schedules affects the final schedules, the performance is still consistently better than other baseline approaches, which demonstrates that our neural rewriter is able to substantially improve the initial solution regardless of its quality.

	Dynamic Job Frequency	Steady Job Frequency
Earliest Job First (EJF)	14.53	24.23
Shortest Job First (SJF)	3.62	5.00
SJF-offline	2.70	4.26
NeuRewriter (dynamic)	**2.56**	3.99
NeuRewriter (steady)	2.59	**3.94**

Table F.2: Experimental results of the job scheduling problem with different distribution of job frequency.

	Uniform Job Resources	Non-uniform Job Resources
Earliest Job First (EJF)	11.06	24.23
Shortest Job First (SJF)	4.51	5.00
SJF-offline	2.76	4.26
NeuRewriter (uniform)	**2.73**	4.05
NeuRewriter (non-uniform)	3.13	**3.94**

Table F.3: Experimental results of the job scheduling problem with different distribution of job resources.

Initial average slowdown	≤ 10	$10 - 25$	> 25
Final average slowdown	3.88	3.90	4.06
Earliest Job First (EJF)	24.23		
Shortest Job First (SJF)	5.00		
Shortest First Search (SJFS)	4.98		
DeepRM	10.18		
OR-tools	15.18		
SJF-offline	4.26		
NeuRewriter	**3.94**		

Table F.4: Experimental results of the job scheduling problem using initial schedules with different average slowdown. The number of resource types $D = 20$.

F.5   More Discussion of the Evaluation on Vehicle Routing Problem

We generate the initial routes for NeuRewriter in the following way: starting from the depot, at each timestep, the vehicle visits the nearest node that is either: (1) a customer node that has not been visited yet, and its resource demand can be satisfied; or (2) the depot node, and the resources carried by the vehicle is less than its capacity. See Figure F.5 for examples of

Model	VRP20, Cap30	VRP50, Cap40	VRP100, Cap50
NeuRewriter	**6.16**	**10.51**	**16.10**
AM-Greedy	6.40	10.98	16.80
AM-Sampling	6.25	10.62	16.23
Nazari et al. (RL-Greedy)	6.59	11.39	17.23
Nazari et al. (RL-BS(5))	6.45	11.22	17.04
Nazari et al. (RL-BS(10))	6.40	11.15	16.96
CW-Greedy	7.22	12.85	19.72
CW-Rnd(5,5)	6.89	12.35	19.09
CW-Rnd(10,10)	6.81	12.25	18.96
SW-Basic	7.59	13.61	21.01
SW-Rnd(5)	7.17	13.09	20.47
SW-Rnd(10)	7.08	12.96	20.33
OR-Tools	6.43	11.31	17.16
Gurobi (optimal)	6.10	-	-

Table F.5: Experimental results of the vehicle routing problems.

the initial solutions. In this way, the average tour length is 7.74 for VRP20, 13.47 for VRP50, and 20.36 for VRP100. Note that these results are even worse than the classic heuristics compared in Table F.5.

Table F.5 presents more results for vehicle routing problems, and Figure F.5 shows an example of the rewriting steps performed by NeuRewriter.

For generalization results, note that after training on VRP50, NeuRewriter achieves an average tour length of 17.33 on VRP100 (See Figure 7.4b in the mainbody of the paper). This is better than 18.00 reported in [186], suggesting that our approach could adapt better to different problem distributions.

F.6   More Results for Expression Simplification

In Figures F.6 and F.7, we present some success cases of expression simplification, where we can simplify better than both the Halide rule-based rewriter and the Z3 solver.

(a) Step 0.

(b) Step 1.

(c) Steps 2-5.

(d) Step 6.
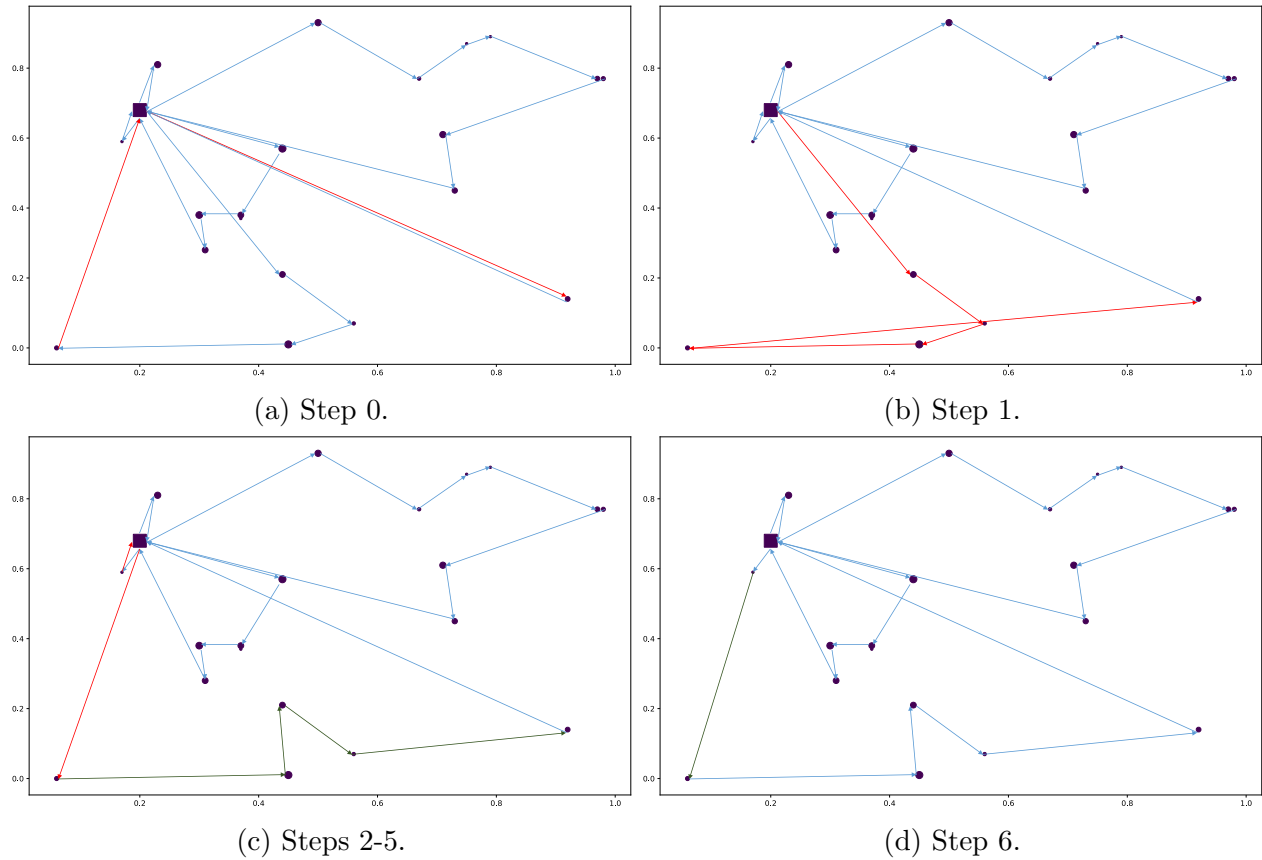
Figure F.5: An example of the rewriting steps for a VRP20 problem. The square is the depot, and circles are customer nodes. The customer node sizes are proportional to their resource demands. At each stage, red edges are to be rewritten at the next step, and green edges are rewritten ones. The tour length of the initial route is 7.31, and the final tour length after rewriting is 5.98.

(a) Step 0.

(b) Step 1.

(c) Step 2.

(d) Step 3.

(e) Step 4.
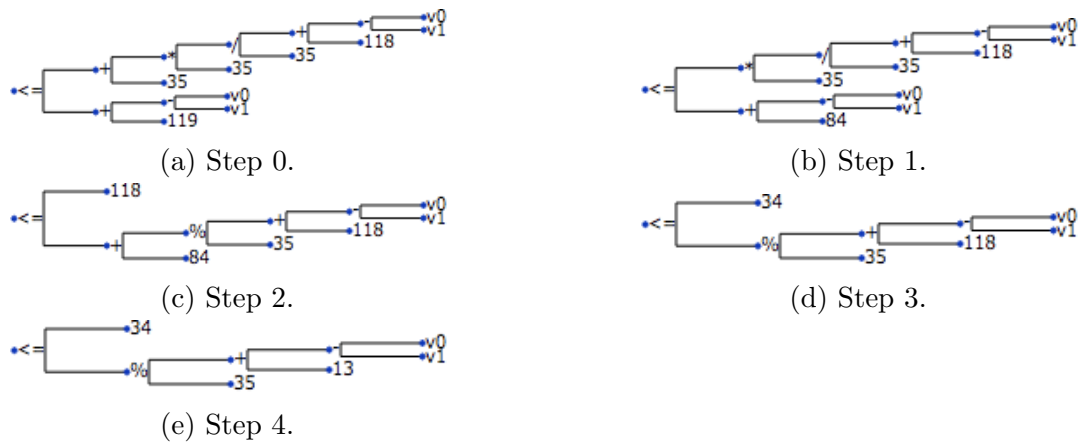
Figure F.6: The rewriting process that simplifies the expression $((v0 - v1 + 18)/35 * 35 + 35) \leq v0 - v1 + 119$ to $34 \leq (v0 - v1 + 13)\%35$.

(a) Step 0.

(b) Step 1.

(c) Step 2.

(d) Step 3.

(e) Step 4.

(f) Step 5.

(g) Step 6.

(h) Step 7.

(i) Step 8.

(j) Step 9.

(k) Step 10.
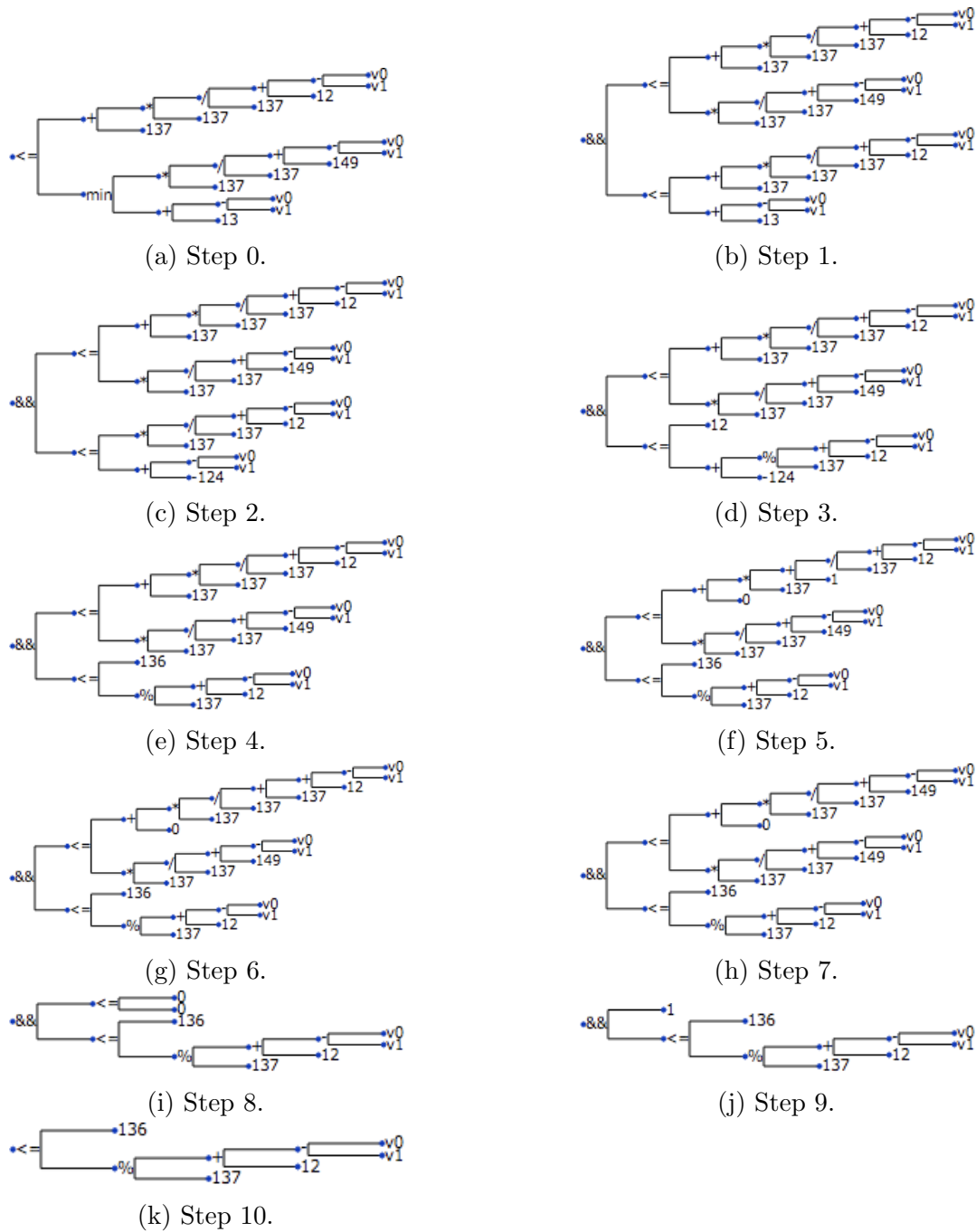
Figure F.7: The rewriting process that simplifies the expression $((v0 - v1 + 12)/137 * 137 + 137) \leq min((v0 - v1 + 149)/137 * 137, v0 - v1 + 13)$ to $136 \leq (v0 - v1 + 12)\%137$.

Appendix G

Neural-Symbolic Reader for Reading Comprehension

G.1   More details about the input preprocessing

We preprocess the input passages and questions in a similar way as the input preprocessing of DROP dataset described in [11]. Specifically, to facilitate the usage of BERT, we split up the documents longer than $L = 512$ tokens. Meanwhile, we extract the locations and values of the numbers, so that they can be retrieved via indices when applying numerical operators. We apply the same input preprocessing on MathQA as well.

G.2   More discussion about the domain specific language

To better support numerical reasoning, sometimes we need to leverage pre-defined constants for our computation. On MathQA, we have shown that applying the constant 3600, which is provided in their pre-defined question-agnostic constant list, is necessary for the calculation in Table 8.2. Meanwhile, we find that defining such a constant list is also helpful on DROP benchmark. For example, a variant of the sample numerical operation question in Table 8.7 is "How many people, in terms of percentage, were not either solely white or solely African American?", and such questions are included in DROP dataset as well. In this case, unless we are able to use the number 100 in our calculation, there is no way to obtain the correct answer. Again, previous works design specialized modules to deal with such questions, which is the main role of the negation module illustrated in Figure 8.1. On the contrary, we introduce a constant list that is callable for every question, so that the model can learn to apply any constant covered in the list, without the need of manually designing separate modules for questions requiring different constants.

In our evaluation, for DROP, we used $[100, 12, 28, 29, 30, 31, 1, 0]$ as the constant list, which

is helpful for percentage and date time calculation. For MathQA, we used the constant list provided in their public dataset, which includes 23 constants that cover common conversion between different units, domain-specific constants for geometry, physics and probability, etc.

G.3   More details about the model architecture

Reader

The reader implementation is largely the same as [11]. Specifically, for the embedding representation of the reader component, we feed the question and passage jointly into BERT, which provides the output vector of each input token t_i as e_i. Unless otherwise specified, the encoder is initialized with the uncased whole-word-masking version of BERT$_{\text{LARGE}}$. We denote the size of e_i as H_0.

Programmer

The core architecture of the programmer is a 1-layer LSTM with the hidden size of $H = 512$. To formally describe the input space and output space of the programmer, we denote R as the size of the reserved tokens, which include both operators and constants in a domain-specific language, and the special start and end tokens [GO] and [EOF]; and $L = 512$ as the total number of the question and passage tokens in a single sample. Samples with fewer than $L = 512$ tokens will be padded with [EOF] tokens to achieve this length. In the following, we discuss the details of each component.

Input embedding.   At each timestep, the programmer could generate a program token from: (1) the reserved tokens of the domain-specific language; and (2) the input question and passage tokens. The embedding of the i-th reserved token is

$$hr_i = E_r^T r_i$$

Where E_r is a trainable embedding matrix of size $R \times H$, and r_i is the one-hot encoding of the token.

For the i-th token in the input question and passage token list, their embedding is

$$ht_i = P_t e_i$$

Where P_t is a trainable projection matrix of size $H \times H_0$.

Attention module over the input.   At each timetstep T, let $[p_1, p_2, ..., p_{T-1}]$ denote the list of program tokens that are already generated in previous timesteps, and we define $[hp_0, hp_1, hp_2, ..., hp_{T-1}]$ as the *decoder history*, where hp_0 is the embedding vector of the [GO]

token calculated as above; $[hp_1, hp_2, ..., hp_{T-1}]$ are H-dimensional vectors corresponding to the generated program token list, and we will discuss how they are computed later.

Denote $(h_T, c_T) = \text{LSTM}(hp_{T-1}, (h_{T-1}, c_{T-1}))$ as the hidden state of the LSTM decoder at timestep T, where (h_0, c_0) is the trainable initial state, and hp_{T-1} is the LSTM input.

For each of hp_i in the decoder history, we compute

$$vh_i = W_h hp_i$$

Where W_h is a trainable matrix of size $H \times H$.

The attention weight of each hp_i in the decoder history is computed as

$$wh_i = \frac{\exp(h_T^T vh_i)}{\sum_{j=0}^{T-1} \exp(h_T^T vh_j)}$$

The attention vector of the decoder history is thus

$$att_h = \sum_{i=0}^{T-1} wh_i \cdot hp_i$$

This formulation is similar to the attention mechanism introduced in prior work [20]. Correspondingly, we compute the attention vector of the passage tokens att_p, and the attention vector of the question tokens att_q.

Afterwards, we compute

$$v_T = W_v[att_h; att_q; att_p; h_T]$$

Where W_v is a trainable matrix of size $H \times 4H$, and $[a; b]$ denotes the concatenation of a and b.

Program token prediction.   We compute another attention vector of the question tokens att_q' in a similar way as above, but with a different set of trainable parameters. Then for each input token, we have

$$ht_i' = P'[ht_i; ht_i \circ att_q']$$

$$hr_i' = P'[hr_i; hr_i \circ att_q']$$

Where P' is a trainable matrix of size $H \times 2H$, and $\circ$ is the Hadamard product.

Let H_T' be a $(R+L) \times H$-dimensional matrix, where the first R rows are hr_i' for $0 \leq i < R$, and the next L rows are ht_i' for $0 \leq i < L$. Then we compute

$$w_T' = H_T' \cdot v_T$$

Where w_{Ti}' denotes the weight of selecting the i-th token as the next program token. This design is similar to the pointer network [253].

Note that a valid program should satisfy the grammar constraints, for instance, those listed in Table 8.1 on DROP dataset. Therefore, we compute a mask m_T as an $(R + L)$-dimensional vector, where $m_{Ti} = 1$ when the i-th token is a valid next program token, and $m_{Ti} = 0$ if it is invalid. In the following, we take the DROP dataset as the example, and list some sample rules for mask generation:

(1) At the beginning of the program generation, $m_{Ti} = 1$ iff the i-th token denotes an operator;

(2) When the previous generated program token p_{T-1} is `PASSAGE_SPAN`, then $m_{Ti} = 1$ iff the i-th token is from the passage. Similarly, if p_{T-1} is `QUESTION_SPAN`, then $m_{Ti} = 1$ iff the i-th token is from the question.

(3) As discussed in Appendix G.1, we preprocess the data to extract the locations and values of numbers in the input question and passage, thus we can leverage it to generate masks for numerical calculation operators. Specifically, when $p_{T-1} \in \{\text{DIFF}, \text{SUM}, \text{VALUE}\}$, $m_{Ti} = 1$ iff the i-th token is from the constant list, or a number from either the input question or the passage.

With the generated program mask, we compute

$$w_T = w'_T - C(1 - m_T)$$

Where C is a large positive constant to ensure that the weight of an invalid program token is much smaller than the valid program tokens. In practice, we use $C = 1e6$. Such a grammar-based decoding process is a common practice in order to ensure the syntactic correctness of the generated programs [143, 160, 35].

Afterwards, the model predicts $p_T = \arg\max_i(w_T)$ as the next program token. We can also apply the beam search for decoding, but we find that the greedy decoding is already sufficient to provide good results, while the inference process is also much faster than the beam search.

Finally, $hp_T = H'_{Tp_T}$ is the vector representation corresponding to p_T, which is appended to the decoder history for generating the next program token.

G.4 More details about training

Data augmentation

In this section, we discuss the details of our data augmentation process for counting and sorting questions on DROP. To obtain training samples for counting questions with ground truth annotations, starting from the span selection questions in the training set, we filter out those questions that either can be answered by using the `QUESTION_SPAN` operation, or do not start with any interrogative in ["What", "Which", "Who", "Where"]. Afterwards, we replace the interrogative with "How many", and modify the ground truth program correspondingly. In this way, we can augment 15K additional questions for counting in DROP training set.

To annotate the key-value pairs, for each entity recognized by the CoreNLP tool, we search for the numbers that are in the same clause as the entity, i.e., not separated by any punctuation mark, and discard those entities that do not have any nearby number satisfying this constraint. Afterwards, we filter out those questions that do not include any superlative in ["longest", "shortest", "largest", "smallest", "most" and "least"]. For the remaining questions, we call each of the sorting operations, i.e., `ARGMAX`, `ARGMIN`, `MAX`, `MIN`, with all extracted key-value pairs as the arguments. For `ARGMAX` and `MAX` operators, the key-value pairs are sorted in the descending order of their values; for `ARGMIN` and `MIN` operators, they are sorted in the increasing order of their values. If any of the resulted sorting program yields the correct answer, the program is included into the training set. In this way, we can annotate 0.9K questions using `ARGMAX` or `ARGMIN` operations, and 1.8K questions using `MAX` or `MIN` operations in DROP training set.

Training configuration

For the training algorithm described in Algorithm 3, the initial threshold $\alpha_0 = 0.5$, and the decay factor $\gamma = 0.5$. We perform early stopping when both exact match and F1 score on the development set do not improve for two consecutive training iterations. For both DROP and MathQA datasets, the training typically takes around $50K \sim 60K$ training steps.

For both tasks in our evaluation, we train the model with Adam optimizer, with an initial learning rate of 5e-5, and batch size of 32. Gradients with L_2 norm larger than 1.0 are clipped.

G.5 Examples of wrong annotations on DROP

Table G.1 lists some examples of wrong annotations in DROP training set. Specifically, the first annotation is wrong because the crowd worker simply counts the number of field goals included in the entire passage, without considering the constraints of lengths and the kicker's name; on the other hand, the second mistake comes from the wrong numerical calculations. For both samples, the highest likelihood among all programs with the annotated answer is smaller than 1e-4, thus are not included during training, which is why the thresholding helps significantly.

G.6 Examples of wrong predictions on DROP

Table G.2 presents some error cases of NeRd on DROP development set.

Passage	Question	Ground truth
... but had to settle for a **23-yard field goal by kicker Matt Bryant** ...	How many field goals shorter than 30 yards did Matt Bryant kick?	3
... from a sample of 40 Sherman tanks, **33 tanks burned** (82 percent) and **7 tanks remained unburned** ...	How many more Sherman tanks burned out than survived in the Normandy Campaign?	22

Table G.1: Some samples in DROP training set with the wrong annotations, which are discarded by NeRd because none of the annotated programs passes the threshold of our training algorithm.

Question type	Passage	Question	Prediction
Question span	The **campaigns of 1702 and 1703** showed his limitations as a field officer... In early 1704 , he spoke with the envoy of Savoy about possible opportunities in their army ...	What happened first, the Hague campaigns as field officer or he spoke with envoy of Savoy for opportunities in the army?	**Prediction:** QUESTION_SPAN(7,10) **Result:** "campaigns as field officer" **Ground truth:** "campaigns of 1702 and 1703"
Counting	... The five regions with the lowest fertility rates were Beijing (0.71), **Shanghai (0.74)**, **Liaoning (0.74)**, Heilongjiang (0.75) ...	How many areas had a fertility rate of .74?	**Prediction:** COUNT(PASSAGE_SPAN(216, 216), PASSAGE_SPAN(223, 223), PASSAGE_SPAN(230, 231)) **Result:** COUNT(''Beijing", "Shanghai", "Liaoning") = 3 **Ground truth:** 2
Sorting	... to set up Nugent's career-long **54-yard field goal** to give the Jets a 9-3 lead ... The half ended when Brown came up five yards short on a 59-yard field goal attempt ...	How many yards was the longest field goal?	**Program:** MAX(VALUE(16), VALUE(20)) **Result:** MAX(54, 59) = 59 **Ground truth:** 54

Table G.2: Examples of wrong predictions on DROP dev set.

Appendix H

Compositional Generalization via Neural-Symbolic Stack Machines

H.1 Discussion of the Benchmark Selection for Evaluation

Given that NeSS achieves impressive results on synthetic natural language benchmarks in our evaluation, one question is whether it could also improve the performance on commonly used natural language datasets, e.g., large-scale machine translation benchmarks. However, note that most existing natural language benchmarks are not designed for evaluating the compositional generalization performance of models. Instead, the main challenge of those datasets is to handle the inherently ambiguous and potentially noisy natural language inputs. Specifically, their training and test sets are usually from the same distribution, and thus do not evaluate compositional generalization. As a result, we did not run experiments on these datasets. Instead, our evaluation focuses on standard sequence-to-sequence generation benchmarks used in previous works on compositional generalization. Such benchmarks are typically constructed with synthetic grammars, so that it is easier to change training and test distributions. We consider improving compositional generalization for more natural inputs as future work.

H.2 More Details of the Stack Machine

We present the sample usage of our machine with a more complex example on SCAN benchmark in Figure H.1. In the following, we provide a more detailed explanation of our CONCAT_M and CONCAT_S operations based on this example. Specifically, when executing the CONCAT_M operation at step 10, we first concatenate all items in the stack top and the memory as a list, i.e., [[JUMP], around, [RTURN]] in this case. Next, according to the argument [2, 0], the items with indices 2 and 0 are selected and concatenated, which results in the

sequence [RTURN, JUMP]. Afterwards, this new sequence replaces the original content in the memory, and the selected item in the stack top, i.e., [JUMP], is removed from the stack top. On the other hand, the token "around" is kept in the stack top, because it is not selected for `CONCAT_M`. The argument selection for `CONCAT_S` is similar, except that this operation puts the generated sequence in the stack top.

In Figure H.2, we present how our machine supports the `REDUCE` operation defined in the parsing machine of [48], which is designed for context-free grammar parsing.

H.3 More Details of the Neural Controller Architecture

We present the neural controller architecture in Figure H.3, and we describe the details below.

Machine status encoder. We use a bi-directional LSTM $LSTM_{inp}$ to encode the token sequence in the input queue, and use the LSTM output for *tok* as its vector representation e_{tok}. We use two separate bi-directional LSTMs $LSTM_{cur}$ and $LSTM_{pre}$ to encode the top 2 stack frames respectively. The LSTM output at the last timestep is used as the embedding of the 2 stack frames, denoted as e_{cur} and e_{pre}. Similarly, another LSTM $LSTM_M$ is used to encode the memory, and we denote the embedding as e_M. Note that we always add an [EOS] token when computing the embedding for stack frames and memory, even if they are empty.

`REDUCE` **argument predictor.** The `REDUCE` instruction takes the top stack frame as the input, and outputs a token sequence in the target vocabulary as the arguments, denoted as $p_{\texttt{REDUCE}}(arg|e_{cur})$. We design the `REDUCE` argument predictor as a standard LSTM-based sequence-to-sequence model with attention, and the argument generation process terminates when an [EOS] token is predicted. The output at the last timestep of the LSTM decoder is used as the embedding of the entire reduced sequence, and it replaces the embedding vectors originally in the top stack frame.

`CONCAT_M` **and** `CONCAT_S` **argument predictors.** We design the same architecture for `CONCAT_M` and `CONCAT_S` argument predictors, but with different sets of model parameters, and we discuss the details for `CONCAT_M` as follows. Firstly, we use a bi-directional LSTM to compute an embedding vector for each element in the top 2 stack frames and the memory. Note that the stack frames and memory include tokens in the source vocabulary that are directly moved from the input queue using the `SHIFT` instruction, as well as sequences generated by previous `REDUCE`, `CONCAT_M` and `CONCAT_S` instructions that consist of tokens in the target vocabulary. To select the arguments for `CONCAT_M` and `CONCAT_S`, we only consider token sequences in the target vocabulary that are included in the top stack frame and the memory as the candidates. However, when computing the embedding vectors, we still include other elements in the top 2 stack frames and the memory, so as to encode the context
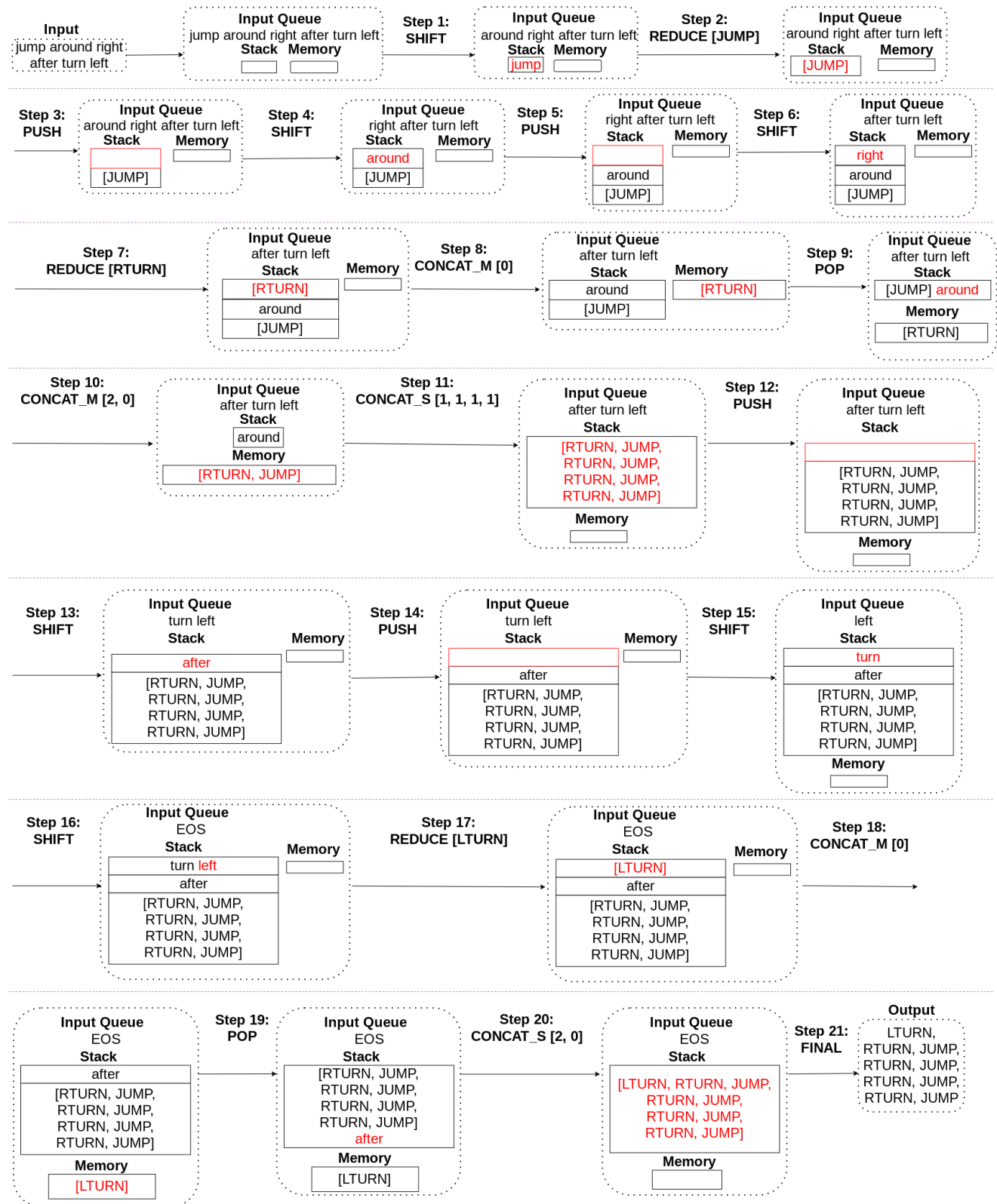
Figure H.1: A more complicated usage of the stack machine for SCAN benchmark.
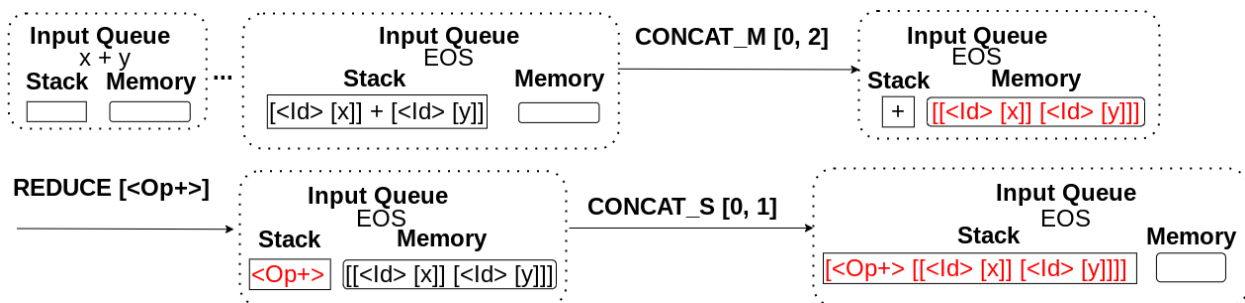
Figure H.2: An illustrative example of our stack machine for context-free grammar parsing. This example showcases the execution steps that are equivalent to a REDUCE operation defined in the parsing machine of [48]. CONCAT_M is used to select the children for the generated tree, REDUCE is used to generate the non-terminal, and CONCAT_S is used to construct the tree.
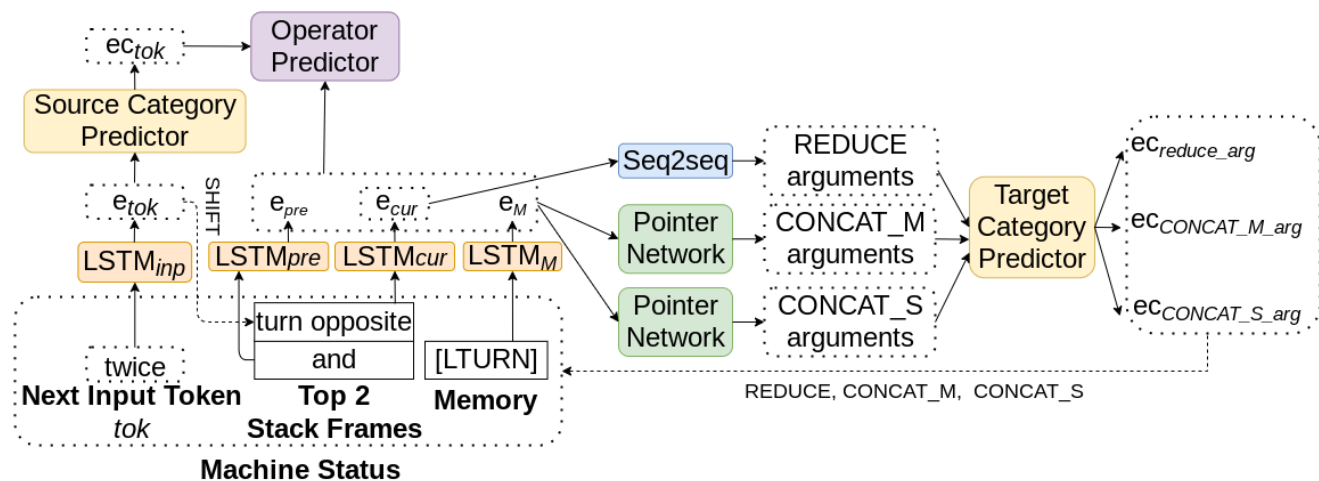

Figure H.3: The neural architecture for the machine controller. The dotted arrows indicate the update of machine status representation after executing the corresponding instructions.
.

information. We keep an additional embedding vector of the `[EOS]` token for argument prediction, which could be selected to terminate the argument generation process. We utilize a pointer network architecture [253] to select indices from the input element list as arguments, and the argument generation process terminates when it selects `[EOS]` token as the argument. The output at the last timestep of the pointer network is used as the embedding of the generated sequence, and it replaces the embedding vectors originally in the memory. We denote the two generators as $p_{\texttt{CONCAT\_M}}(arg|e_{cur}, e_{pre}, e_M)$ and $p_{\texttt{CONCAT\_S}}(arg|e_{cur}, e_{pre}, e_M)$.

Latent category predictors. Both source and target category predictors include a classification layer followed by an embedding matrix. Specifically, for the source category predictor, given e_{tok} as the input, the classification layer is a $|C_s|$-dimensional softmax layer, which predicts a probability distribution of the category that the input word *tok* belongs to. Let c_{tok} be the category that *tok* belongs to, another embedding matrix E_c is used to compute an embedding vector ec_{tok}. Similarly, given an embedding vector of a token sequence s in the target language, denoted as e_s, the classification layer of the target category predictor predicts a $|C_t|$-dimensional probability distribution indicating the category of the sequence s, then another embedding matrix is used to compute an embedding vector describing the categorical information of the token sequence. Note that when a `SHIFT` instruction is executed, we still put the embedding vector e_{tok} to the stack top instead of its categorical embedding, since different tokens in the same category could be processed with different `REDUCE` arguments. For example, "left" should be reduced into "LTURN", while "right" should be reduced into "RTURN". On the other hand, the categorical predictions for the target language are used for subsequent predictions of both operators and arguments. We set the number of categories $|C_s|$ and $|C_t|$ for source and target languages as their vocabulary sizes, to support the degenerate mapping that considers each token as a separate category.

H.4 More Details for Training

We outline the training algorithm in Algorithm 6. In the algorithm, we denote the prediction probability distribution of the operator predictor as p_{op}, and the argument prediction probability distribution as p_{args}.

Rule extraction. Our recursive machine design enables us to extract rules learned from previous lessons. For each execution step in a learned trace, we denote a tuple of (machine status, operator) as an extracted rule for operator prediction, where the machine status includes the contents of the top 2 stack frames, the memory, and the next token *tok* in the input queue.

    Similarly, we keep 3 rule sets for `REDUCE`, `CONCAT_M` and `CONCAT_S` argument prediction respectively, where the machine status includes the information used as the input to the corresponding predictors. For example, the ruleset for `REDUCE` argument prediction includes tuples of (stack top, argument). Therefore, after we extract the rules from NeSS trained
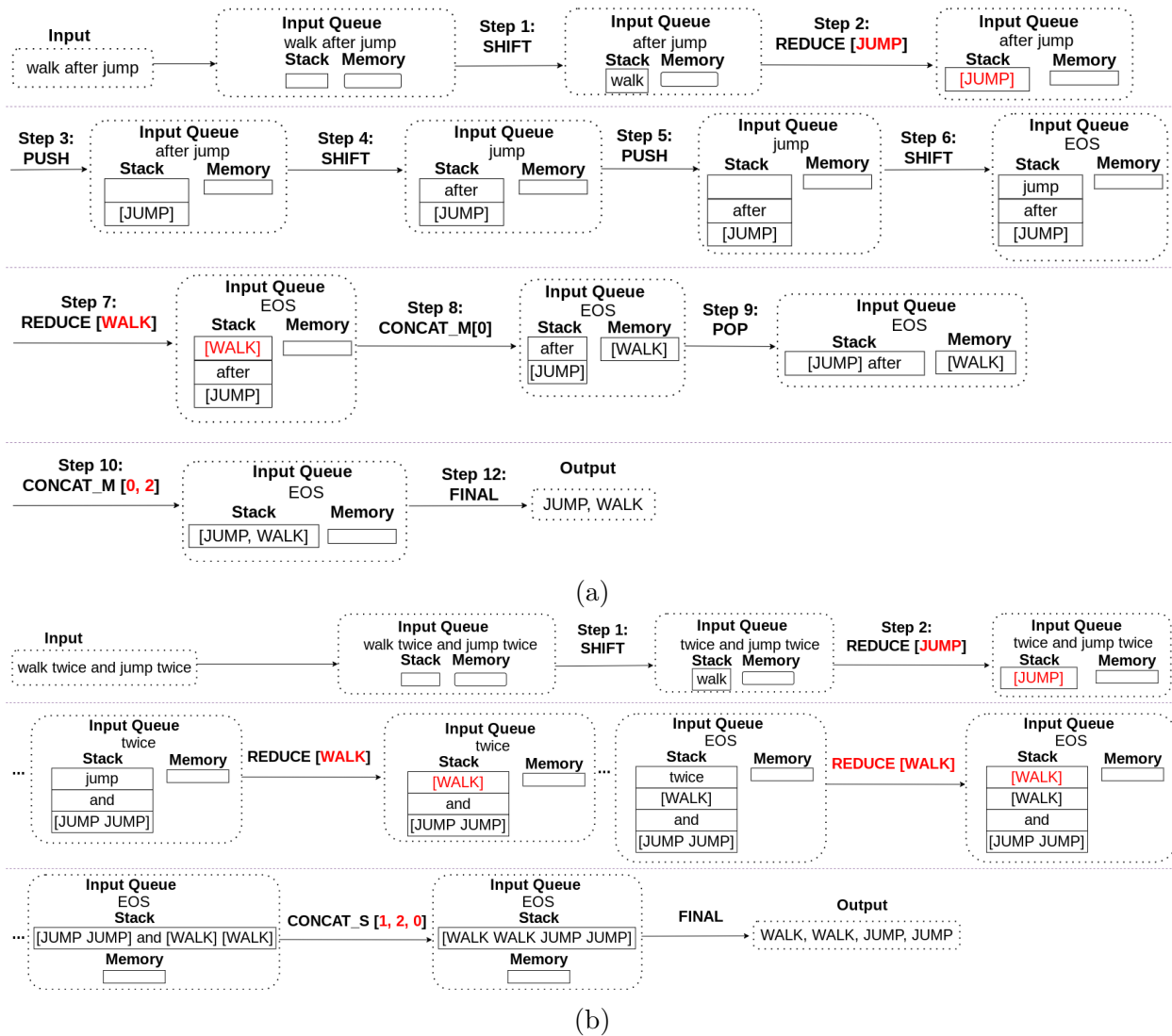
Figure H.4: Sample spurious traces on SCAN benchmark, which could be pruned by rule extraction. The wrong predictions of operators and arguments are marked with red.

Algorithm 6 Training algorithm for NeSS

Input: training lessons L, the i-th lesson $L_i = \{(x_{ij}, y_{ij})\}_{j=1}^{N_i}$, a model p^θ

The extracted ruleset $R \leftarrow \emptyset$

The current training data $D \leftarrow \emptyset$

for $L_i \in L$ **do**

    $D \leftarrow D \cup L_i$

    repeat

        for each batch $B_j = \{(x_k, y_k)\}_{k=1}^{|B_j|} \in D$ **do**

            $T_j \leftarrow \text{TraceSearch}(p^\theta, B_j, R)$

            $Ops_j \leftarrow$ operator traces in T_j

            $args_j = $ `REDUCE, CONCAT_M, CONCAT_S` arguments in T_j

            // OEs, OEt: latent category supervision for the source and target languages.

            $OEs_j, OEt_j \leftarrow \text{OEExtraction}(B_j, T_j)$

            $Loss_{\text{op}} \leftarrow -\log p^\theta_{\text{op}}(Ops_j)$, $Loss_{args} \leftarrow -\log p^\theta_{\text{args}}(args_j)$

            $Loss_{\text{OE}} \leftarrow -(\log p^\theta_{sc}(OEs_j) + \log p^\theta_{tc}(OEt_j))$

            $Loss \leftarrow Loss_{\text{op}} + Loss_{\text{args}} + Loss_{\text{OE}}$

            Update θ to minimize $Loss$

        end for

    until No more non-degenerate execution traces are found with the search.

    $R \leftarrow \text{RuleExtraction}(p^\theta, L_i)$

end for

---

on SCAN, its `REDUCE` ruleset should be as follows: {(run, [RUN]), (jump, [JUMP]), (look, [LOOK]), (walk, [WALK]), (left, [LTURN]), (right, [RTURN]), (turn left, [LTURN]), (turn right, [RTURN])}. The extracted ruleset for the few-shot learning and context free grammar parsing tasks also largely follow the pre-defined ground truth grammar. For the compositional machine translation benchmark, the main extracted `REDUCE` rules include: {(i am, [je suis]), (i am not, [je ne suis pas]), (you are, [tu es]), (you are not, [tu n es pas]), (he is, [il est]), (he is not, [il n est pas]), (she is, [elle est]), (she is not, [elle n est pas]), (we are, [nous sommes]), (we are not, [nous ne sommes pas]), (they are, [elles sont]), (they are not, [elles ne sont pas]), (very, [tres]), (daxy, [daxiste])}.

Note that we do not extract rules for degenerate execution traces, unless the length of the output sequence is 1, which suggests that the degenerate execution trace is the most appropriate one.

Speed up the trace search with extracted rules. To further speed up the trace search during the training process, we utilize the rules extracted from previous lessons, and prioritize their usage for the trace search in the current lesson. In Figure H.4, we provide some examples of spurious traces without leveraging the rules extracted from previous lessons. For example, in the spurious trace for "walk after jump" shown in Figure H.4a, "walk" and "jump" are wrongly reduced into "JUMP" and "WALK" respectively, and with the wrong `CONCAT_S` argument, the output sequence still matches the ground truth. Besides the wrong

arguments, a spurious trace could also get the operators wrong, as shown in Figure H.4b. In this spurious trace, a `REDUCE` operation is applied to the word "twice". Given that "jump", "walk" and "twice" already appear in previous lessons including shorter sentences, ideally, a well-trained model is supposed to perfectly memorize them. However, since our trace search is a sampling process, such spurious traces are still possible, especially when the input sequences become long. With the rule extraction process for training, NeSS prioritizes traces where the operators and arguments do not conflict with the learned rules, e.g., those with the correct `REDUCE` arguments for "walk" and "jump", and the correct operations for "twice". Specifically, when NeSS encounters a machine status that is already included in the rule set extracted from previous lessons, NeSS directly applies the corresponding rule, and only searches for other operations when it cannot find any trace consistent with the extracted rule.

Training for latent category predictors. During the training process, when we encounter two instances that are considered as potentially operationally equivalent, we first feed one of the instances into the latent category predictor, and randomly sample a category index based on the probability distribution computed by the predictor. Afterwards, we set this category index to be the ground truth category for both the two instances. If the first occurrence of one instance is in an earlier lesson than another one, then we sample the category index based on the prediction probability distribution computed for this instance, otherwise we arbitrarily select one instance from them.

H.5 Implementation Details

Curriculum design. For SCAN benchmark, we split the training set into 6 lessons. The first 4 lessons include samples with an input sequence length or an output sequence length of 1, 2, 3 and 4 respectively. The fifth lesson includes all samples with an input sequence length larger than 4, and a maximal output action sequence length of 8. The sixth lesson includes the rest of training samples.

For the compositional machine learning benchmark, the curriculum is designed with the increasing order of length of the English sentences, where the first lesson includes the shortest sentences with 3 words, e.g., "I am daxy" and "you are good", the second lesson includes the sentences with 4 words, etc. Note that each English sentence in this dataset includes no more than 9 words.

Trace search for training. When searching for non-degenerate execution traces, the length limit of the `REDUCE` argument predictor is 2 for SCAN and the context-free grammar parsing tasks, and 5 for the compositional machine translation task. No length limit is set for the `CONCAT_M` and `CONCAT_S` argument predictors. No length limit is set for the `REDUCE` argument predictor during the inference time, which allows it to produce degenerate execution traces.

For each training sample, the model searches for at most 256 execution steps to find a non-degenerate trace, and if no such trace is found, a degenerate trace is used for training. The execution steps are counted by the number of operators, e.g., the trace in Figure H.1 includes 21 execution steps. We use a simple trick to further speed up the trace search. Note that when the sequence generated in an intermediate execution step is already not a substring of the ground truth output sequence, this operation cannot be correct. In this case, we backtrack to the previous step, and sample another different operation to execute.

Other training hyper-parameters. We train the model with the Adam optimizer, the learning rate is 1e-3 without decaying, and the batch size is 256. We do not use dropout for training. The model parameters are uniformly randomly initialized within [-1.0, 1.0]. The norm for gradient clipping is 5.0. We perform an evaluation for the model after every 200 training steps, and the model usually converges to the optimum within 3000 training steps.

Model hyper-parameters. Each bi-directional LSTM used in the neural controller includes 1 layer, with the hidden size of 256. The embedding size is 512.

H.6 More Results on the Context-free Grammar Parsing Task

In Table H.1, we present the results including all different setups and baselines in [48]. Specifically, Stack LSTM, Queue LSTM, and DeQue LSTM are designed in [96], where they augment an LSTM with a differentiable data structure.

H.7 More Details of the Few-shot Learning Task

Figure H.5 shows the full dataset used for the few-shot learning task in our evaluation.
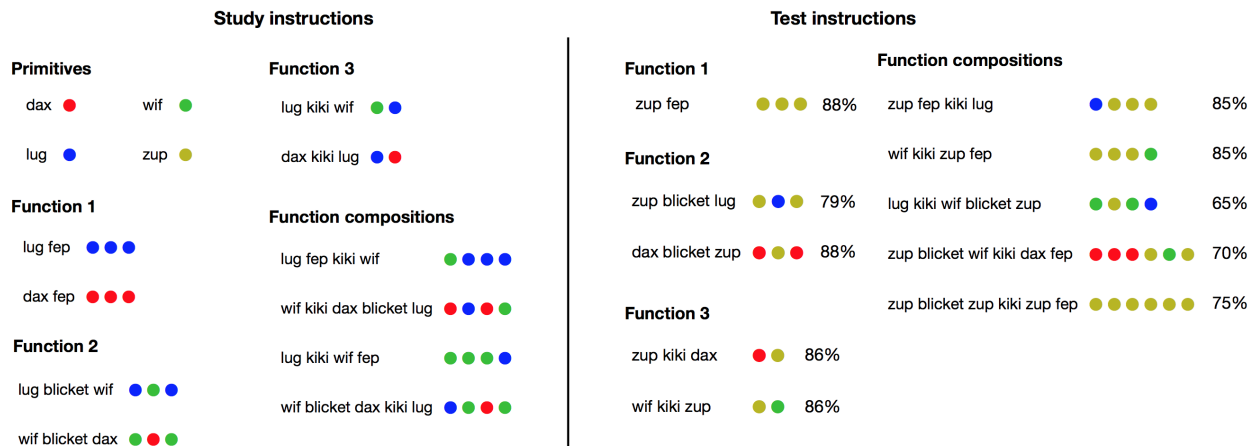
Figure H.5: The full dataset used for the few-shot learning of compositional instructions. This figure is taken from [150], where the percentage after each test sample is the proportion of human participants who predict the correct output.

.

Table H.1: The full experimental results on context-free grammar parsing benchmarks proposed in [48].

While-Lang

Train	Test	NeSS (ours)	Neural Parser	Seq2seq	Seq2tree	Stack LSTM	Queue LSTM	DeQue LSTM
Curriculum	Training	**100%**	100%	81.29%	100%	100%	100%	100%
	Test-10	**100%**	100%	0%	0.8%	0%	0%	0%
	Test-100	**100%**	100%	0%	0%	0%	0%	0%
	Test-1000	**100%**	100%	0%	0%	0%	0%	0%
	Test-5000	**100%**	100%	0%	0%	0%	0%	0%
Std-10	Training	**100%**	100%	94.67%	100%	81.01%	72.98%	82.59%
	Test-10	**100%**	100%	20.9%	88.7%	2.2%	0.7%	2.8%
	Test-100	**100%**	100%	0%	0%	0%	0%	0%
	Test-1000	**100%**	100%	0%	0%	0%	0%	0%
Std-50	Training	**100%**	100%	87.03%	100%	0%	0%	0%
	Test-50	**100%**	100%	86.6%	99.6%	0%	0%	0%
	Test-500	**100%**	100%	0%	0%	0%	0%	0%
	Test-5000	**100%**	100%	0%	0%	0%	0%	0%

Lambda-Lang

Train	Test	NeSS (ours)	Neural Parser	Seq2seq	Seq2tree	Stack LSTM	Queue LSTM	DeQue LSTM
Curriculum	Training	**100%**	100%	96.47%	100%	100%	100%	100%
	Test-10	**100%**	100%	0%	0%	0%	0%	0%
	Test-100	**100%**	100%	0%	0%	0%	0%	0%
	Test-1000	**100%**	100%	0%	0%	0%	0%	0%
	Test-5000	**100%**	100%	0%	0%	0%	0%	0%
Std-10	Training	**100%**	100%	93.53%	100%	0%	95.93%	2.23%
	Test-10	**100%**	100%	86.7%	99.6%	0%	6.5%	0.1%
	Test-100	**100%**	100%	0%	0%	0%	0%	0%
	Test-1000	**100%**	100%	0%	0%	0%	0%	0%
Std-50	Training	**100%**	100%	66.65%	89.65%	0%	0%	0%
	Test-50	**100%**	100%	66.6%	88.1%	0%	0%	0%
	Test-500	**100%**	100%	0%	0%	0%	0%	0%
	Test-5000	**100%**	100%	0%	0%	0%	0%	0%