Structural Language Models of Code

Uri Alon [1] **Roy Sadaka** [1] **Omer Levy** [2 3] **Eran Yahav** [1]

Abstract

We address the problem of *any-code completion* – generating a missing piece of source code in a given program without any restriction on the vocabulary or structure. We introduce a new approach to any-code completion that leverages the strict syntax of programming languages to model a code snippet as a tree – *structural language modeling* (SLM). SLM estimates the probability of the program's abstract syntax tree (AST) by decomposing it into a product of conditional probabilities over its nodes. We present a neural model that computes these conditional probabilities by considering all AST paths leading to a target node. Unlike previous techniques that have severely restricted the kinds of expressions that can be generated in this task, our approach can generate arbitrary code in any programming language. Our model significantly outperforms both seq2seq and a variety of structured approaches in generating Java and C# code. Our code, data, and trained models are available at `http://github.com/tech-srl/slm-code-generation/`. An online demo is available at `http://AnyCodeGen.org`.

1. Introduction

Code completion is the problem of generating code given its surrounding code as context. In its most general form, this problem is extremely challenging as it requires reasoning over an unbounded number of syntactic structures and user-defined symbols. Previous approaches have avoided this issue by limiting the generation problem: program synthesis approaches are often tailored to domain-specific languages (Gulwani, 2011; Polozov & Gulwani, 2015; Devlin et al., 2017; Ellis et al., 2019), while other recent approaches generate code in general languages like Java and C#, but severely restrict the syntax, vocabulary, domain, or nature of the generated programs (Murali et al., 2018; Brockschmidt et al., 2019; Young et al., 2019).

We introduce the task of *any-code completion* – generating code in a general-purpose programming language without any restriction on its vocabulary or structure. Specifically, we focus on generating code in context: given a program $\mathcal{P}$ and some part of the program p, the task is to predict p from the rest of the program $\mathcal{P}^{-}=\mathcal{P}\backslash p$. Any-code completion thus generalizes the restricted completion task of Brockschmidt et al. (2019), in which the target code contained only primitive types (e.g., `int` and `string`) and excluded user-defined functions. Figure 1 shows two any-code completion examples.

In related tasks such as semantic parsing (Dong & Lapata, 2018; Yu et al., 2018; Iyer et al., 2019), natural-language-to-code (Allamanis et al., 2015; Iyer et al., 2018), and edit-to-code (Yin et al., 2019; Zhao et al., 2019), models must use separate encoders and decoders because of the different modalities of the input (e.g. natural language text) and the output (code). In contrast, we leverage the fact that our input and output are of the *same modality* (code), and pursue better generalization by modeling them *jointly*.

We present a new approach that explicitly models the source and the target code as the same tree – *structural language modeling* (SLM). SLM estimates the probability of the program's abstract syntax tree (AST) by decomposing it into a product of conditional probabilities over its *nodes*. We present a neural model that computes these conditional probabilities by considering all AST paths leading to a target node, generalizing over traditional language models that consider sequences of words. While prior work uses AST paths to *read* programs (Alon et al., 2019b), we *generate* code by predicting the next node along the set of paths, generating the target AST node-by-node.

We evaluate SLMs on Java any-code completion, achieving a new state of the art: exact-match accuracy@1 of 18.04% and accuracy@5 of 24.83% (previous SOTA: 16.93% and 23.17%). SLMs also outperform existing models in the restricted completion task of Brockschmidt et al. (2019) in C# by a wide margin, 37.61% accuracy@1 compared to

[1]Technion, Israel [2]Tel Aviv University [3]Facebook AI Research. Correspondence to: Uri Alon <urialon@cs.technion.ac.il>, Roy Sadaka <roysadaka@gmail.com>, Omer Levy <omerlevy@gmail.com>, Eran Yahav <yahave@cs.technion.ac.il>.

Proceedings of the 37th International Conference on Machine Learning, Online, PMLR 119, 2020. Copyright 2020 by the author(s).

```
public static Path[] stat2Paths(
    FileStatus[] stats) {
  if (stats == null) return null;
  Path[] ret = new Path[stats.length];
  for (int i = 0; i < stats.length; ++i){
    ret[i] =                     ;
  }
  return ret;
}
```

True ref (Java):		`stats[i].getPath()`
SLM top-5:	(25.2%)	**`stats[i].getPath()`**
	(3.3%)	`Path(stats[i])`
	(2.5%)	`new Path(stats[i], charset)`
	(1.7%)	`stat(stats[i], ret)`
	(0.8%)	`new Path(stats[i])`

(a)

```
public static string Camelize(
    this string input)
{
    var word = input.Pascalize();
    return word.Length > 0 ?
                              .ToLower()
            + word.Substring(1)
        : word;
}
```

True ref (C#):		`word.Substring(0, 1)`
SLM top-5:	(14.1%)	**`word.Substring(0, 1)`**
	(8.2%)	`word.trim()`
	(5.8%)	`word.Substring(1)`
	(2.4%)	`input.Substring(0, 1)`
	(1.9%)	`wordValue.Substring(0, 1)`

(b)

Figure 1. Examples from the Java (left) and C# (right) test sets. The highlighted expression in each example is the target p, which our models correctly generated from the rest of the snippet. Additional and larger examples can be found in the supplementary material.

26.42%. Our ablation study reveals the importance of *joint modeling* of the source and target code, rather than separating encoders from decoders. Finally, we discuss the theoretical advantages of SLMs, and show how they generalize many previous structural approaches for code generation. An interactive demo of our model is presented at `http://AnyCodeGen.org`.

2. Code Generation as Structural Language Modeling

We model the task of any-code completion by computing the probability of a program $Pr(\mathcal{P})$, similar to how a language model computes the probability of a natural language sentence. While language models typically assume a *sequence* as their input, our input is an abstract syntax *tree* $\mathcal{A_P}$. We thus introduce a *structural* language modeling approach (SLM).

The intuition behind this idea is that a language model could *generalize better* by modeling the *tree* rather than the sequential form of the program. Further, learning from the AST allows a model to save learning capacity, instead of having to re-learn known syntactic patterns from the text.

We first show a chain-rule decomposition of the tree's probability $Pr(\mathcal{A_P})$ into a product of conditional *node* probabilities, and then describe our path-based model for computing the individual conditional probabilities. We explain how to construct a tree from local node predictions, and finally discuss how our approach differs from previous work on production-based tree generation.

Representing Code as a Tree A program $\mathcal{P}$ is a sequence of tokens that can be unambiguously mapped to an abstract syntax tree (AST) $\mathcal{A_P}$, where every node represents an element in the language (e.g. conditions, loops, variable declarations) from a set $\mathcal{T}$. Each AST leaf (terminal) has an associated user-defined value $v \in \mathcal{V}$. Nonterminal nodes can have a varying number of children nodes.

Decomposing the Probability of a Tree Given a tree $\mathcal{A_P}$, we first traverse the tree, depth-first,[1] to induce an ordering over its nodes $a_0, \ldots, a_{|\mathcal{A_P}|} \in \mathcal{A_P}$. We decompose the probability of a tree $Pr(\mathcal{A_P})$ using the chain rule, akin to the standard approach in language modeling:

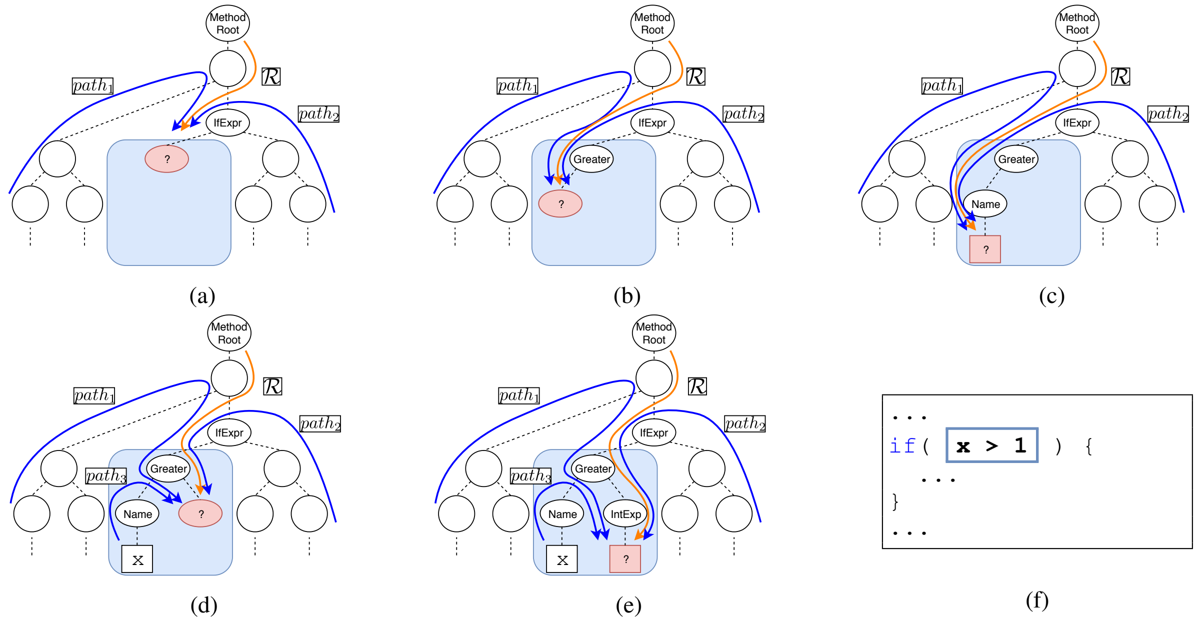
$$Pr(\mathcal{A_P}) = \prod_t Pr(a_t | a_{<t}) \tag{1}$$

where $a_{<t}$ are all the nodes that were traversed before a_t.

In any-code completion, part of the tree ($\mathcal{A}_{\mathcal{P}^-}$) is already observed. Therefore, we order the nodes of $\mathcal{A}_{\mathcal{P}^-}$ to be before the nodes of the target p, and compute only the conditional probabilities over the nodes in p, essentially conditioning on the observed tree $\mathcal{A}_{\mathcal{P}^-}$.

Representing Partial Trees via Paths How can we represent the partial tree composed of $a_{<t}$ when computing $Pr(a_t|a_{<t})$? In standard language modeling, the structure is linear, and $a_{<t}$ is a sequence. One way to represent a partial tree is to linearize it according to the traversal order (Xiao et al., 2016); however, this creates artificially long distances between the current node a_t and ancestor nodes (e.g., the root a_0). Another option is to use only the path from the root node to a_t (Rabinovich et al., 2017), but this ignores a lot of contextual information (e.g., sibling nodes).

[1]Depth-first ordering is a common practice in tree generation (Maddison & Tarlow, 2014; Raychev et al., 2016), but in principle our framework also allows for other orderings.



```
...
if( x > 1 ) {
  ...
}
...
```

Figure 2. The subtree representing `x > 1` is generated given its surrounding tree. At each step, the model generates the next node (denoted by ?) of $path_1$, $path_2$ and $path_3$ using the root path $\mathcal{R}$. Dashed lines denote the AST structure; solid lines denote AST paths. Most AST paths are omitted from the figure, for clarity.

We follow Alon et al. (2018) and use *the set of paths* from every leaf to a_t together with the path from the root to a_t. Intuitively, each path captures the effect of a different, possibly distant, program element on a_t, along with the syntactic relationship between them. For example, in Figure 1 (left) the three paths originating from `Path[] ret` inform the model about the existence of `ret` which is an array of type `Path`. Thus, when completing `ret[i] = ...` – the completion should be a `Path` object. Other paths inform the model that the target is inside a `For` loop, iterated `stats.length` times. Considering the information flowing from all paths, our model correctly generates `stats[i].getPath()`.

We denote the (candidate) node at time t as a_t, its (given) parent, which is currently expanded, by $\pi(a_t)$, and the set of all paths as $\mathcal{S}_t$:

$$\mathcal{S}_t = \{\ell \rightsquigarrow \pi(a_t) \mid \ell \in \text{leaves}(a_{<t})\}$$

where $\ell \rightsquigarrow \pi(a_t)$ is the (only) path in the tree between a leaf ℓ and the current node to expand $\pi(a_t)$. We denote the path from the root of the program as $\mathcal{R}_t = a_0 \rightsquigarrow \pi(a_t)$, which represents the current, relative position of $\pi(a_t)$ in the program (marked as $\mathcal{R}$ in Figure 2). Whereas prior work used *whole* paths (between two leaf nodes) to encode ASTs (Alon et al., 2019a;b), our model observes *partial* paths (between a leaf and any other node) and learns to extend them by predicting their next node.

Figure 2 illustrates the traversal order of a subtree that represents the expression `x > 1` and some of the paths used to compute the probability at each step. At each step, the probability of the next node is computed given the paths $\mathcal{S}_t$ from the root and every given leaf up to the current node to expand. Figure 2(d) shows how after the terminal node with the value `x` is given, $path_3$ originating from this leaf is also used to compute the probability of the next nodes.

Our path-based approach generalizes previous approaches such as "parent feeding" and "previous action" encoding (Yin & Neubig, 2017), context nodes (Bielik et al., 2016), and some of the graph-edges of Brockschmidt et al. (2019). See Section 8 for further discussion.

Generating Trees In sequence generation, the length of the target sequence is controlled by generating an `EOS` token to stop. When generating trees, we require a more sophisticated mechanism to control arity and depth. We augment $\mathcal{A}_{\mathcal{P}}$ in two ways to allow node-by-node generation.

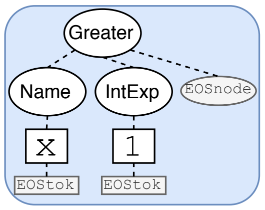


Figure 3. Augmenting the AST with EOS_{node} and EOS_{tok} nodes.

First, we add a special EOS_{node} node to every nonterminal to control for *arity*. Generating this node indicates that the parent node has no more children nodes. Second, we end each subtoken sequence with a special EOS_{tok} node to control for *depth* during generation; we decompose each terminal node n_v into a sequence of terminal nodes T_v by splitting up the node's value v into *subtokens* based on camel notation. For example, if $v =$ `toLowerCase`, then $T_v =$ `to` $\rightarrow$ `lower` $\rightarrow$ `case` $\rightarrow$ $\texttt{EOS}_{tok}$. Figure 3 shows an example of both $\texttt{EOS}_{node}$ and $\texttt{EOS}_{tok}$ in action.

Node Trees vs. Production Trees While we predict a single *node* at each step, previous work (Iyer et al., 2018; 2019) predicts a grammar production rule. This representation decomposes the code in a way that often forces the model to predict with partial information. For instance, consider generating the expression **`str.Substring(3)`**. The model of Brockschmidt et al. (2019) would first predict the rule **`Expr`**$\rightarrow$**`Expr.Substring(Expr)`**, and only then expand **`Expr`**$\rightarrow$**`str`** and **`Expr`**$\rightarrow$**`3`**. That is, the model needs to predict the method name (`Substring`) *before* the invoking object (`str`). Further, the `Substring` method can get either one *or* two arguments, forcing the model to choose whether to use the one- or two-argument rule in advance. Node generation, however, allows us to predict the presence of a function call and only then to predict its object and method name, rather than predicting these a priori.

3. Model Architecture

In the previous section, we described how we can generate code given the probabilities $Pr\left(a_t|a_{<t}\right)$, where $a_{<t}$ is represented by the set of partial AST paths $\mathcal{S}_t$. Here, we present a neural model that estimates $Pr\left(a_t|\mathcal{S}_t\right)$. We first encode each path in $\mathcal{S}_t$ as a vector (Section 3.1); then, we contextualize and aggregate the entire set. Finally, we predict the target node a_t by combining a subtoken vocabulary with a syntactic copy mechanism (Section 3.3).

3.1. Encoding AST Paths

Given a partial AST path, i.e., a sequence of nodes $n_1, \ldots, n_k$, our goal is to create a vector representation.

We first represent each node n_i using embeddings. A subtoken node is represented by the index of its subtoken w in the embedding matrix E^{subtoken}; AST nodes are represented as a pair $n_i = (\tau, \kappa)$ where τ is the node type, e.g. `IfStatement`, and κ is the node index among its sibling nodes. We represent node types using a learned embedding matrix E^{type} and the child indices using a learned matrix E^{index}. The node's vector representation is the concatenation of the type and index vectors.

$$e\left(n_i\right) = \begin{cases} E^{\text{subtoken}}_{w} & n_i \text{ is the subtoken } w \\ \left[E^{\text{type}}_{\tau}; E^{\text{index}}_{\kappa}\right] & n_i \text{ is the AST node } (\tau, \kappa) \end{cases}$$

We encode the entire path using a uni-directional LSTM stack, and take the final states:[2]

$$\widetilde{f}\left(n_1, \ldots, n_k\right) = \text{LSTM}\left(e\left(n_1\right), \ldots, e\left(n_k\right)\right)$$

Given a set of partial paths $\mathcal{S}$ (omitting the iterator t for simplicity), we denote their encodings as $H = \{\widetilde{f}\left(n_1, \ldots, n_k\right) \mid \left(n_1, \ldots, n_k\right) \in \mathcal{S}\}$.

Efficient Computation When modeling a subtree, there are large overlaps between paths from different time steps. In particular, paths that originate from the same leaf share the same *prefix*. We therefore apply the LSTM on the prefix *once* and cache the intermediate state across suffixes, speeding up both training and inference significantly. An example is shown in the supplementary material (Fig. 2).

3.2. Aggregating Multiple Paths

Given the set of paths $\mathcal{S}$ leading up to the parent $\pi(a)$ of the target node a, our goal is to represent $\mathcal{S}$ in the context of predicting a. To do so, we introduce the aggregation function $g\left(H, r, i\right)$. As its input, g takes the set of encoded paths H, the encoded root path r, and the child index i of the currently predicted child node a relative to its parent.

We first contextualize the path encodings H using a transformer encoder (Vaswani et al., 2017).[3] In parallel, we apply a non-linear transformation to the encoding of the root path $r = \widetilde{f}\left(\mathcal{R}\right)$, in order to inform it that we wish to predict the i-th child of $\pi(a)$:

$$Z = \text{Transformer}\left(H\right) \qquad \widetilde{r} = W_a \cdot \text{ReLU}\left(C_i \cdot r\right)$$

In this formulation, the parameter matrix C_i is used when the child index is i, while the parameter matrix W_a is used for every instance.

We then compute attention over the set of contextualized path encodings Z using the index-informed root-path encoding $\widetilde{r}$ as the query; we pass the weighted average $\widetilde{z}$ and the root-path encoding $\widetilde{r}$ through another fully-connected layer; we denote the resulting vector representation as $\widetilde{h}$:

$$\boldsymbol{\alpha} = \text{softmax}\left(Z \cdot \widetilde{r}\right) \qquad \widetilde{z} = \sum_j \alpha_j \cdot Z_j$$

$$\widetilde{h} = g\left(H, r, i\right) = \text{ReLU}\left(W_g\left[\widetilde{z}; \widetilde{r}\right]\right)$$

where semicolons (;) denote vector concatenation.

3.3. Predicting with a Syntactic Copy Mechanism

We can now predict a from the representation $\widetilde{h}$. If the target node's parent $\pi(a)$ is a nonterminal AST node, then a must be an AST node; otherwise, a is a subtoken.

[2]Replacing the LSTMs with transformers yielded similar results in preliminary experiments.

[3]Since H is a set, we do not use positional embeddings.

Predicting AST Nodes If a is an AST node, we predict a using a softmax over the node type embeddings E^{type}:

$$Pr\left(a|\mathcal{S}\right) = \text{softmax}\left(E^{\text{type}} \cdot \widetilde{h}\right) \quad (\pi(a) \text{ is a nonterminal})$$

Predicting Subtokens Programs repeatedly refer to previously declared symbols, resulting in highly repetitive usage of identifiers. We therefore use a copy mechanism (Gu et al., 2016) to allow our model to predict either entire tokens or individual subtokens that exist in the context. As we show in Section 6, copying greatly improves our model's performance. For brevity, we describe how entire tokens are copied, and elaborate on the copy of *sub*tokens in the supplementary material. We score each leaf ℓ using a bilinear function (W_c) between its path's encoding H_ℓ and $\widetilde{h}$. At the same time, we score the token w, which is the token associated with ℓ, from a limited vocabulary using the inner product between its representation in the subtoken embedding matrix E^{subtoken} and $\widetilde{h}$.

$$s_{\text{copy}}\left(\ell\right) = H_\ell \cdot W_c \cdot \widetilde{h} \qquad s_{\text{gen}}\left(w\right) = E^{\text{subtoken}}_w \cdot \widetilde{h}$$

The scores s_{copy} and s_{gen} are then summed over all occurrences that correspond to the same symbol and subsequently normalized via softmax. A key difference from most previous work (Ling et al., 2016; Yin & Neubig, 2017) is that our copy mechanism uses the *syntactic* relation to the source (the path H_ℓ), rather than the sequential relation or the graph-node representation (Yin et al., 2019).

4. Experimental Setup

4.1. Benchmarks

Any-Code Completion: Java We take the Java-small dataset of Alon et al. (2019a), which is a re-split of the dataset of Allamanis et al. (2016). It contains 11 GitHub projects, broken down into a single method per example, and split to train/dev/test by project to reduce code overlap. This dataset was found to contain the least code duplication by Allamanis (2019). We create any-code completion examples by selecting every expression larger than a single AST node as the target, using the remainder of the method as the context. We remove methods containing the word "test" in their body or file name, and omit 10% of the examples by filtering out methods longer than 20 lines to avoid configurations, initializations, and auto-generated code. To make the task even harder, we remove examples where the target appears as-is in the context. Ultimately, this dataset contains 1.3M/10k/20k train/dev/test examples.

Restricted Completion: C# To provide a fair comparison to Brockschmidt et al. (2019), we create an additional benchmark where the missing code is more limited. We use the code of Brockschmidt et al. (2019) which filters out examples where the targets contain non-primitive types or user-defined functions. We extract the exact same types of limited expressions. Since the dataset of Brockschmidt et al. (2019) is not publicly available, we consulted with Brockschmidt et al. directly and extracted examples from the raw dataset of Allamanis et al. (2018) using their "unseen projects test" set. This dataset contains 30 GitHub projects broken down to one method per example. This dataset contains 16k/8k/3k train/dev/test examples.

Our datasets are available at: `http://github.com/tech-srl/slm-code-generation/`. Detailed statistics are provided in the supplementary material.

Metrics Following Brockschmidt et al. (2019), we report exact match accuracy at 1 and 5. We also introduce a new *tree@k* metric which counts a prediction as correct if the entire tree structures, ignoring leaf values, are identical. For example, `x > 1` and `y > 2` would *not* count as identical in *exact match*, but *would* count as "tree-match identical" because both express that an identifier is greater than an integer (`NAME > INT`). The *tree@k* metric is interesting because it allows us to tease apart the model's syntactic errors from incorrect subtoken predictions.

4.2. Baselines

We compare our model to a variety of original implementations and adaptations of existing models. We put significant effort to perform a fair comparison, including adding a copy mechanism to the NMT baselines and *sub*tokenization as in our model. We adapt strong baselines from the literature to our task, even if they were designed to different tasks such as NL→code and code→NL. We re-train all the following baselines on the same datasets as our models.

NMT We use standard autoregressive sequence-to-sequence NMT baselines, in which we subtokenize the given code snippet, replace the target in the source with a special `PRED` symbol, and train the network to predict the target as a sequence of subtokens. *Transformer$_{base}$+copy* (Vaswani et al., 2017) uses the implementation of OpenNMT (Klein et al., 2017) with a copy mechanism (Gu et al., 2016). *Transformer$_{small}$+copy* uses d_{model}=256, d_{ff}=1024, and 4 self attention heads per layer. *BiLSTM→LSTM+copy* is a 2-layer bidirectional LSTM encoder-decoder with d=512 and attention. *seq2tree+copy* follows Aharoni & Goldberg (2017) and learns to generate the linearized, subtokenized target AST.

Java-specific Baselines We use the original implementation of Iyer et al. (2018), and also their *seq2prod* baseline which is a re-implementation of Yin & Neubig (2017); these are designed for NL→code tasks, in which we feed the code context as the NL input. The model of Iyer et al. (2018) is designed to get additional input of the available variables *and their types*, for which we do not feed types.

Model	acc@1	acc@5	tree@1	tree@5
code2seq (Alon et al., 2019a)	10.68	15.56	30.46	43.94
Iyer et al. (2018)	5.94	9.19	25.54	36.75
seq2prod (Yin & Neubig, 2017)	8.05	11.82	30.77	41.73
Transformer$_{\text{small}}$ (Vaswani et al., 2017)+copy	14.23	21.35	31.83	47.40
Transformer$_{\text{base}}$ (Vaswani et al., 2017)+copy	16.65	24.05	34.68	50.52
BiLSTM→LSTM (Luong et al., 2015)+copy	16.93	23.17	34.29	49.72
seq2tree (Aharoni & Goldberg, 2017)+copy	16.81	23.04	38.14	52.36
SLM (this work)	**18.04**	**24.83**	**39.10**	**55.32**

Table 1. Results on any-code completion in Java.

While these models could also be applied to other languages, their implementation only supports Java.

C#-specific Baselines We compare our model to the graph-based *GNN*$\rightarrow\mathcal{NAG}$ model using the implementation of Brockschmidt et al. (2019). Bielik et al. (2016) kindly trained and tested their non-neural PHOG model on our C# dataset. We note that PHOG does not have an explicit copy mechanism, and considers only context to the left of the target code, while we consider also context to the right. Extending PHOG could potentially improve its results.

In both Java and C#, we compare to *code2seq* (Alon et al., 2019a), which is a strong code→NL model. We train it to generate the target code as a *sequence* of subtokens.

4.3. Implementation and Hyperparameter Settings

Architecture We use embeddings of size 512, 2 layers of LSTMs with 256 units, and 4 transformer layers with 8 attention heads. We kept a small subtoken vocabulary of size 1000 to encourage the model to learn to copy; larger vocabularies did not show an improvement. These resulted in a very lightweight model of only 15M parameters, which is close to *Transformer$_{small}$* (11.8M parameters). In comparison, *Transformer$_{base}$* had more than 45M parameters (3× more parameters than our model).

Training We train the model end-to-end on a single V100 GPU, using cross entropy and the Adam optimizer (Kingma & Ba, 2015), an initial learning rate of 10^{-4} multiplied by 0.95 every $20k$ steps. We bucket examples based on the number of predictions in the target subtree (nodes + subtokens + `EOS`), and vary the batch size such that each batch contains about 512 targets. We train the model to prefer copying entire tokens rather than copying subtokens, if possible, by optimizing for the entire token as the true label. We apply dropout of 0.25 in the Transformer layers, and a recurrent dropout of 0.5 in the LSTMs.

Inference We perform beam search with width of 5 and optimize for accuracy@1.

Model	acc@1	acc@5	tree@1	tree@5
GNN$\rightarrow\mathcal{NAG}$	15.19	27.05	26.48	40.09
code2seq	6.20	10.05	21.97	30.89
seq2seq+copy	26.42	37.94	34.10	49.23
seq2tree+copy	22.29	35.86	31.85	48.53
PHOG	7.40	12.00	–	–
SLM (this work)	**37.61**	**45.51**	**51.10**	**59.82**

Table 2. Results on restricted completion in C#.

5. Results

Any-Code Completion: Java Table 1 shows that our SLM achieves over 1.1% and 0.78% better *acc@1* and *acc@5* (respectively) over the two strongest baselines. The improvement over *Transformer$_{small}$*, which is closer to our model in the number of parameters, is even higher: over 3.8% and 3.4% in *acc@1* and *acc@5*.

The NMT baselines performed better than code-specific baselines. We hypothesize that the reason is that the NMT baselines are more generic, while the code-specific baselines are designed for different tasks: *seq2prod* is designed for tasks which involve generating code *given natural language input*; Iyer et al. (2018) additionally expects all member methods, fields, and their types as input; *code2seq* is designed to generate sequences rather than code, and does not have a copy mechanism. An approximation of *code2seq* with a copy mechanism is presented in Section 6.

Interestingly, the syntactically-informed *seq2tree* baseline achieved the highest *tree@k* among the baselines, while our model achieved higher *acc@k* and *tree@k*. This shows that leveraging the syntax can benefit NMT models as well.

Restricted Completion: C# Table 2 shows the results for the restricted completion task in C#, where *seq2seq+copy* is the *BiLSTM→LSTM+copy* model which performed the best among the Java baselines. We first observe that the *seq2seq+copy* and the *seq2tree+copy* baselines outperform the *GNN*$\rightarrow\mathcal{NAG}$ of Brockschmidt et al. (2019), who introduced this task. Although Brockschmidt et al. (2019) did compare to a seq2seq baseline, their *GNN*$\rightarrow\mathcal{NAG}$ model

Ablation	acc@1	acc@5	tree@1	tree@5
Paths→Seq	12.95	18.52	33.44	43.43
Seq→Path	12.12	17.12	28.68	43.99
Paths→Paths	17.63	24.62	37.78	53.98
No Root Att	14.43	18.48	28.20	35.65
No Copy	10.72	15.70	30.61	44.35
SLM (original)	18.04	24.83	39.10	55.32

Table 3. Ablations on any-code completion in Java.

could copy symbols from the context, but their baseline did not. To conduct a fair comparison with our SLM model, we equipped the seq2seq and seq2tree baselines with a copy mechanism. Even though the *seq2seq+copy* and the *seq2tree+copy* baselines perform substantially better than the state of the art in this setting, our SLM model is able to go beyond, achieving significant gains over all models.

The superiority of our model over *GNN*$\rightarrow\mathcal{NAG}$ may also be related to the GNN bottleneck (Alon & Yahav, 2020), which hinders GNNs from propagating long-range messages. In contrast, propagating long-range messages using paths is natural for our model.

6. Ablation Study

To understand the importance of the various components and design decisions in our model, we conducted an extensive ablation study.

Paths→Seq follows *code2seq* (Alon et al., 2019a) and separates the model to an encoder and a decoder, where the decoder generates the target code as a sequence of subtokens. The main difference from *code2seq* is that *Paths→Seq* includes a copy mechanism, as in our model.

Seq→Path follows Rabinovich et al. (2017) and separates our model to an encoder and a decoder (including a copy mechanism), where the encoder encodes the context as a sequence of subtokens using a BiLSTM, and the decoder generates the missing subtree using the root path and the index of the generated child.

Paths→Paths is similar to our SLM model except that it uses separate encoder and decoder. These encoder and decoder have untied weights, unlike our SLM model which models the source and the target jointly.

No Root Attention uses max pooling instead of attention in aggregating multiple paths (see Section 3.2). The index-informed path from the root to the target's parent ($\mathcal{R}$ in Figure 2) is concatenated with the result, instead of being used as attention query.

No Copy replaces copy mechanism with a much larger vocabulary (25k subtokens instead of 1k).

Results Table 3 shows the results of these alternatives. As our SLM model performs better than *Paths→Paths*, this ablation shows the importance of joint modeling of the context and the target subtree by parameter tying.

Each of *Paths→Paths* and the seq2seq baselines (Table 1) performs better than *Paths→Seq* and *Seq→Path*; this shows the importance of *using the same type of encoder and decoder* for any-code completion, rather than combining "an optimal encoder" with "an optimal decoder". While this distinction between encoder and decoder types might be necessary for semantic parsing (Rabinovich et al., 2017; Dong & Lapata, 2018), NL→code (Yin & Neubig, 2017) and code→NL (Alon et al., 2019a; Fernandes et al., 2019) tasks because of the different modalities of the input and the output, this discrepancy may hurt generalization when the output is essentially a missing part of the input's AST.

Paths→Paths performs better than the seq2seq baselines (Table 1), showing the advantage of using paths over textual sequences, even without parameter tying.

No Root Attention degrades *acc@1* and *acc@5* by 3.6% to 6.3%. This shows that dynamically attending to the context paths given the current root path is crucial.

Not using a copying mechanism results in a degradation of 7.3% to 9.1%. Programs use symbols and identifiers repetitively, thus the ability to copy symbols from the context is crucial for this task. For this reason, we included a copying mechanism in all NMT baselines in Section 4.

7. Qualitative Analysis

Our main results (Table 1 and Table 2) reveal a gap between *acc@k* and *tree@k*: when ignoring identifier values and comparing only the tree structure, accuracy is significantly higher across all models. While our SLM model performs better than all baselines in *acc@k*, our model also shows greater potential for improvement in its *tree@k* results, which are much higher than the baselines'. We thus focus on studying the cases where the tree was predicted correctly, but the model failed to generate the code exactly including names.

Figure 4(a) shows an example of this case: the ground truth has a structure of the form: `NAME.NAME() > INT`. Our model predicts `value.length() > 0` (a tree-match) as its first candidate and `value.length() > 55` (the ground truth) as its second. Null-checking a string is often followed by checking that it is also not empty, making the first candidate a reasonable prediction as well.

Figure 4(b) shows another example: in this case, the ground truth `thisValue == thatValue ? 0 : 1` was pre-

```
private static void log(String value) {
  if (value != null
      && 🟩)
    value = value.substring(0, 55)+"...";
  LOG.info(value);
}
```

True ref:		`value.length() > 55`	
SLM top-5:	(9.6%)	`value.length() > 0`	🌳
	(7.3%)	**`value.length() > 55`**	✓
	(1.8%)	`value.startsWith("...")`	
	(1.5%)	`!value.startsWith("...")`	
	(0.9%)	`value.charAt(0) == '.'`	

(a)

```
public int compareTo(LongWritable o) {
    long thisValue = this.value;
    long thatValue = o.value;
    return (thisValue < thatValue ? -1 :
        (🟩));
}
```

	`thisValue == thatValue ? 0 : 1`	
(16.3%)	`thisValue == thisValue ? 0 : 1`	🌳
(11.0%)	**`thisValue == thatValue ? 0 : 1`**	✓
(9.5%)	`thisValue == value ? 0 : 1`	🌳
(6.6%)	`thisValue > thatValue ? 0 : 1`	
(6.1%)	`(thisValue == thatValue) ? 0 : 1`	↔

(b)

Figure 4. Examples for cases where the top candidate is a "tree-match" (marked with 🌳), but only the second candidate is an "exact match" (marked with ✓ in bold). Predictions that are logically equivalent to the ground truth are marked with ↔. Additional (and larger) examples along with the predictions of the baselines are shown in the supplementary material.

dicted correctly only as the second candidate. Nevertheless, the top-3 candidates are tree-matches since all of them are of the form: `NAME == NAME ? INT : INT`. Interestingly, the fifth candidate `(thisValue == thatValue) ? 0 : 1` is logically-equivalent to the ground truth.

In both examples, our model's top candidate differs from the ground truth by *a single identifier or literal*: in Figure 4(a) the model predicted `0` instead of `55`; in Figure 4(b) the model predicted `thisValue` instead of `thatValue`. Such single *sub*token errors are responsible for 30% of the cases where the model's top prediction is a tree-match but not an exact match. Single *token* (whole identifier or literal) mismatches are responsible for 74% of these cases. Thus, improving our model's ability to predict the right names has the potential to enhance our gains furthermore. Detailed results of allowing such mistakes in our model and in the baselines can be found in the supplementary material.

Additional possible post-filtering could filter out candidates that do not compile. In Figure 5, the first, third and fourth candidates do not compile, because the `this.currentAttempt` object does not have `getCount`, `get`, nor `getTime` methods. If the model's predictions would have been considered in the context of the entire project including its dependencies, these candidates could have been filtered out, and the (correct) fifth candidate would be ranked *second*. We leave compiler-guided code generation to future work.

Additional examples can be found in the supplementary material and in our interactive demo at `http://AnyCodeGen.org`.

8. Related Work

Generalizing Previous Approaches Our approach frames code generation as predicting the next node in all partial AST paths. This simple framing generalizes most previous work, without hand-crafted edges and special actions:

- Models that use information about ancestor nodes only (Rabinovich et al., 2017), as well as the "Parent Feeding" of Yin & Neubig (2017), are generalized by our model, since all paths that go into a node a_t pass through its parent, and the path from the root is the attention query.
- The "previous action encoding" of Yin & Neubig (2017) is also a special case of our approach, because $\mathcal{S}_t$ contains the paths starting from the *previously expanded* leaves of $\mathcal{A}_p$ into the currently expanded node $\pi(a_t)$, such as $path_3$ in Figure 2(e).
- The "context node" of PHOG (Bielik et al., 2016) is just one of the previously-traversed leaf nodes in $a_{<t}$. Thus, not only that our model conditions on this context node as well, our model also takes into account the *syntactic relation*, i.e., the path, between the context and $\pi(a_t)$. Moreover, while PHOG conditions on a single leaf, SLMs condition on *every* leaf in $a_{<t}$.
- Finally, Brockschmidt et al. (2019) define special graph edges (e.g., "NextSib" and "Child") to capture relations on the AST. Allamanis et al. (2018) further defines data-flow and control-flow graph edges such as "ComputedFrom" and "GuardedByNegation". Most of these relations can be expressed as partial AST paths without manually designing them.

Program Generation Learning to generate programs is one of the oldest problems in machine learning (Waldinger & Lee, 1969) and has been considered by some as the "holy

```
public float getProgress() {
    this.readLock.lock();
    try {
        if (this.currentAttempt != null) {
            return                                  ;
        }
        return 0;
    } finally {
      this.readLock.unlock();
    }
}
```

True ref:		`this.currentAttempt.getProgress()`			
SLM top-5:	(31.3%)	`this.currentAttempt.getCount()`		🌳	
	(30.6%)	`-1`			⚙
	(1.5%)	`this.currentAttempt.get()`		🌳	
	(1.2%)	`this.currentAttempt.getTime()`		🌳	
	(0.9%)	**`this.currentAttempt.getProgress()`**	✓	🌳	⚙

Figure 5. An example from our test set in which a compiler-guided generation could filter out non-compiling candidates, and thus rank the ground truth *second* instead of *fifth*. Four out of the five candidates are "tree-match" (marked with 🌳), the fifth candidate is an "exact match" (marked with ✓ in bold), and only the second and the fifth candidate compile (marked with ⚙).

grail of computer science" (Pnueli & Rosner, 1989; Gulwani et al., 2017). Typically, the task is to generate a program given some form of input or context, such as complete formal specifications (Green, 1981; Si et al., 2019) or input-output examples (Gulwani, 2011; Devlin et al., 2017; Parisotto et al., 2017; Balog et al., 2017; Gaunt et al., 2017). While these approaches work well in some cases, they are often bounded to DSLs that prevent them from being applied to realistic, general-purpose code.

Bielik et al. (2016) learn a dynamic DSL expression that points to a *single* context that guides the generation of a JavaScript program. Maddison & Tarlow (2014) and Amodio et al. (2017) generate general-purpose unconditional code, and do not deal with the challenge of fitting the code to a given context.

Brockschmidt et al. (2019) addressed a similar code completion task as ours using a graph encoder and a neural attribute grammar decoder. However, they limit their model to generate only primitive types or arrays of these; use a closed vocabulary; and omit user-defined functions. In this paper, we lift these constraints and allow any, general-purpose, generation of code, of all types and containing any names. As we show in Section 5, our model performs significantly better.

Murali et al. (2018) generate code given a set of APIs in a "Java-like" language; they state that their approach is thus intrinsically limited to generate only API-heavy programs. Yin et al. (2019) generate general-purpose code by applying a given edit to a given code snippet. Brody et al. (2020) predict code edits directly given other edits that occurred in the context. Yin & Neubig (2017) and Rabinovich et al. (2017) used a top-down syntactic approach for generating general-purpose code given a natural language description. Models that address APIs→code, edit→code, or NL→code tasks must model the input separately and differently from the output code. As we show in Section 6, modeling the source and the target differently perform poorly in our task, in which the input is code as well.

Chen et al. (2018) addressed JavaScript↔CoffeeScript translation with a tree-to-tree approach, which required a strong alignment between the source and target trees.

9. Conclusion

We presented a novel approach for any-code completion: joint modeling of an AST and its missing subtree using a structural language model. Our approach generalizes most previous work in this area while reaching state-of-the-art performance on challenging benchmarks. We demonstrate our approach in generating general-purpose code, in restricted and unrestricted settings, in two languages. Our model outperforms a variety of strong baselines, including programming language-oriented models and strong NMT models applied in our settings.

We believe that structural language modeling enables a wide range of future applications, similarly to how language modeling research has contributed to NLP in recent years. Our approach also has a variety of direct applications such as code completion, detecting and fixing unlikely existing code, and re-ranking solutions produced by another synthesizer or solver. To these ends, we make all our code, datasets, and trained models publicly available.

Acknowledgments

We would like to thank Guy Waldman for developing the AnyCodeGen.org website, Pavol Bielik and Martin Vechev for training their PHOG model on our dataset, Srinivasan Iyer for his useful advice, the guidance in training his model and adapting it to our task, Marc Brockschmidt for his useful implementation tips and guidance in training his model, and Miltiadis Allamanis for the guidance in reproducing his C# dataset.

References

Aharoni, R. and Goldberg, Y. Towards string-to-tree neural machine translation. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pp. 132–140, 2017.

Allamanis, M. The adverse effects of code duplication in machine learning models of code. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, pp. 143–153. ACM, 2019.

Allamanis, M., Tarlow, D., Gordon, A., and Wei, Y. Bimodal modelling of source code and natural language. In *International conference on machine learning*, pp. 2123–2132, 2015.

Allamanis, M., Peng, H., and Sutton, C. A convolutional attention network for extreme summarization of source code. In *International conference on machine learning*, pp. 2091–2100, 2016.

Allamanis, M., Brockschmidt, M., and Khademi, M. Learning to represent programs with graphs. In *International Conference on Learning Representations*, 2018.

Alon, U. and Yahav, E. On the bottleneck of graph neural networks and its practical implications. *arXiv preprint arXiv:2006.05205*, 2020.

Alon, U., Zilberstein, M., Levy, O., and Yahav, E. A general path-based representation for predicting program properties. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 404–419, 2018.

Alon, U., Brody, S., Levy, O., and Yahav, E. code2seq: Generating sequences from structured representations of code. In *International Conference on Learning Representations*, 2019a.

Alon, U., Zilberstein, M., Levy, O., and Yahav, E. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, 3 (POPL):1–29, 2019b.

Amodio, M., Chaudhuri, S., and Reps, T. Neural attribute machines for program generation. *arXiv preprint arXiv:1705.09231*, 2017.

Balog, M., Gaunt, A. L., Brockschmidt, M., Nowozin, S., and Tarlow, D. Deepcoder: Learning to write programs. In *International Conference on Learning Representations*, 2017.

Bielik, P., Raychev, V., and Vechev, M. Phog: probabilistic model for code. In *International Conference on Machine Learning*, pp. 2933–2942, 2016.

Brockschmidt, M., Allamanis, M., Gaunt, A. L., and Polozov, O. Generative code modeling with graphs. In *International Conference on Learning Representations*, 2019.

Brody, S., Alon, U., and Yahav, E. Neural edit completion. *arXiv preprint arXiv:2005.13209*, 2020.

Chen, X., Liu, C., and Song, D. Tree-to-tree neural networks for program translation. In *Advances in Neural Information Processing Systems*, pp. 2547–2557, 2018.

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

Dong, L. and Lapata, M. 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)*, pp. 731–742, 2018.

Ellis, K., Nye, M., Pu, Y., Sosa, F., Tenenbaum, J., and Solar-Lezama, A. Write, execute, assess: Program synthesis with a repl. In *Advances in Neural Information Processing Systems*, pp. 9169–9178, 2019.

Fernandes, P., Allamanis, M., and Brockschmidt, M. Structured neural summarization. In *International Conference on Learning Representations*, 2019.

Gaunt, A. L., Brockschmidt, M., Kushman, N., and Tarlow, D. Differentiable programs with neural libraries. In *International Conference on Machine Learning*, pp. 1213–1222, 2017.

Green, C. Application of theorem proving to problem solving. In *Readings in Artificial Intelligence*, pp. 202–222. Elsevier, 1981.

Gu, J., Lu, Z., Li, H., and Li, V. O. 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)*, pp. 1631–1640, 2016.

Gulwani, S. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 317–330, 2011.

Gulwani, S., Polozov, O., Singh, R., et al. Program synthesis. *Foundations and Trends® in Programming Languages*, 4(1-2):1–119, 2017.

Iyer, S., Konstas, I., Cheung, A., and Zettlemoyer, L. Mapping language to code in programmatic context. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pp. 1643–1652, 2018.

Iyer, S., Cheung, A., and Zettlemoyer, L. Learning programmatic idioms for scalable semantic parsing. 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)*, pp. 5429–5438, 2019.

Kingma, D. and Ba, J. Adam: A method for stochastic optimization. In *International Conference on Learning Representations*, 2015.

Klein, G., Kim, Y., Deng, Y., Senellart, J., and Rush, A. M. Opennmt: Open-source toolkit for neural machine translation. In *Proceedings of ACL 2017, System Demonstrations*, pp. 67–72, 2017.

Ling, W., Blunsom, P., Grefenstette, E., Hermann, K. M., Kočiskỳ, T., Wang, F., and Senior, A. Latent predictor networks for code generation. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 599–609, 2016.

Luong, M.-T., Pham, H., and Manning, C. D. Effective approaches to attention-based neural machine translation. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pp. 1412–1421, 2015.

Maddison, C. and Tarlow, D. Structured generative models of natural source code. In *International Conference on Machine Learning*, pp. 649–657, 2014.

Murali, V., Qi, L., Chaudhuri, S., and Jermaine, C. Neural sketch learning for conditional program generation. In *International Conference on Learning Representations*, 2018.

Parisotto, E., Mohamed, A.-r., Singh, R., Li, L., Zhou, D., and Kohli, P. Neuro-symbolic program synthesis. In *International Conference on Learning Representations*, 2017.

Pnueli, A. and Rosner, R. On the synthesis of a reactive module. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 179–190. ACM, 1989.

Polozov, O. and Gulwani, S. Flashmeta: a framework for inductive program synthesis. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pp. 107–126, 2015.

Rabinovich, M., Stern, M., and Klein, D. 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)*, pp. 1139–1149, 2017.

Raychev, V., Bielik, P., Vechev, M., and Krause, A. Learning programs from noisy data. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 761–774, 2016.

Si, X., Yang, Y., Dai, H., Naik, M., and Song, L. Learning a meta-solver for syntax-guided program synthesis. In *International Conference on Learning Representations*, 2019.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. Attention is all you need. In *Advances in Neural Information Processing Systems*, pp. 6000–6010, 2017.

Waldinger, R. J. and Lee, R. C. Prow: A step toward automatic program writing. In *Proceedings of the 1st international joint conference on Artificial intelligence*, pp. 241–252, 1969.

Xiao, C., Dymetman, M., and Gardent, C. Sequence-based structured prediction for semantic parsing. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 1341–1350, 2016.

Yin, P. and Neubig, G. A syntactic neural model for general-purpose code generation. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 440–450, 2017.

Yin, P., Neubig, G., Allamanis, M., Brockschmidt, M., and Gaunt, A. L. Learning to represent edits. In *International Conference on Learning Representations*, 2019.

Young, H., Bastani, O., and Naik, M. Learning neurosymbolic generative models via program synthesis. In *International Conference on Machine Learning*, pp. 7144–7153, 2019.

Yu, T., Li, Z., Zhang, Z., Zhang, R., and Radev, D. Typesql: Knowledge-based type-aware neural text-to-sql generation. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 2 (Short Papers)*, pp. 588–594, 2018.

Zhao, R., Bieber, D., Swersky, K., and Tarlow, D. Neural networks for modeling source code edits. *arXiv preprint arXiv:1904.02818*, 2019.