# PROGRAM SYNTHESIS WITH LEARNED CODE IDIOMS

**Anonymous authors**
Paper under double-blind review

## ABSTRACT

Program synthesis of general-purpose source code from natural language specifications is challenging due to the need to reason about high-level patterns in the target program and low-level implementation details at the same time. In this work, we present PATOIS, the first system that allows a neural program synthesizer to explicitly interleave high-level and low-level reasoning at every generation step. It accomplishes this by automatically mining common *code idioms* from a given corpus and then incorporating them into the underlying language for neural synthesis. We evaluate PATOIS on a challenging program synthesis dataset NAPS and show that using learned code idioms improves the synthesizer's accuracy.

## 1 INTRODUCTION

Program synthesis is a task of translating a specification, expressed as a combination of natural language and input-output examples, into the most likely program that satisfies this specification in a given language (Gulwani et al., 2017). In the last decade, it has advanced dramatically thanks to the appearance of novel neural and neuro-symbolic techniques (Balog et al., 2017; Devlin et al., 2017b; Kalyan et al., 2018), first mass-market applications (Polozov & Gulwani, 2015), and new datasets (Devlin et al., 2017a; Ling et al., 2016; Yin et al., 2018a; Zavershynskyi et al., 2018; Zhong et al., 2017). Figure 1 shows a few examples of typical program synthesis tasks. Most of the successful applications apply program synthesis to manually crafted domain-specific languages (DSLs), such as FlashFill and Karel, or to subsets of general-purpose functional languages, such as SQL and Lisp. However, scaling program synthesis to algorithmic programs in a general-purpose language with complex control flow remains an open challenge.

We conjecture that one of the main current challenges of synthesizing a program lies in the insufficient separation between high-level and low-level reasoning. In a typical program generation process, be it a neural generative model or a symbolic search, the target program is generated in terms of its *surface tokens*, which typically represent low-level implementation details of the latent higher-level *patterns* in the program. In contrast, expert programmers switch between high-level reasoning ("a binary search over an array") and low-level implementation ("`m = (l + r)/2`") multiple times when completing a single function. Reasoning over multiple levels of abstraction simultaneously complicates the program generation task for a model.

This conjecture is supported by two key observations. First, recent work (Dong & Lapata, 2018; Murali et al., 2018) has proposed explicitly separating the program synthesis process into *sketch generation* and *sketch completion*. The first stage generates a high-level sketch of the target program, and the second stage fills in missing details in the sketch. Such separation improves the accuracy

| Dataset | NL Spec | Examples | Program |
|---|---|---|---|
| NAPS (Zavershynskyi et al., 2018) | *Given a string S, if its first letter is uppercase, then make all letters of S lowercase. Otherwise, make the first letter of S uppercase and make other letters of S lowercase.* | `cAPS` → `Caps`<br>`TEST` → `test`<br>⋮ | `(if (&& (>= (array_index S 0) 65)`<br>`(<= (array_index S 0) 90))`<br>`(lower S)`<br>`(concat (upper (array_index S 0))`<br>`(lower (substring S 1 -1))))` |

Figure 1: Representative program synthesis tasks from the NAPS dataset.

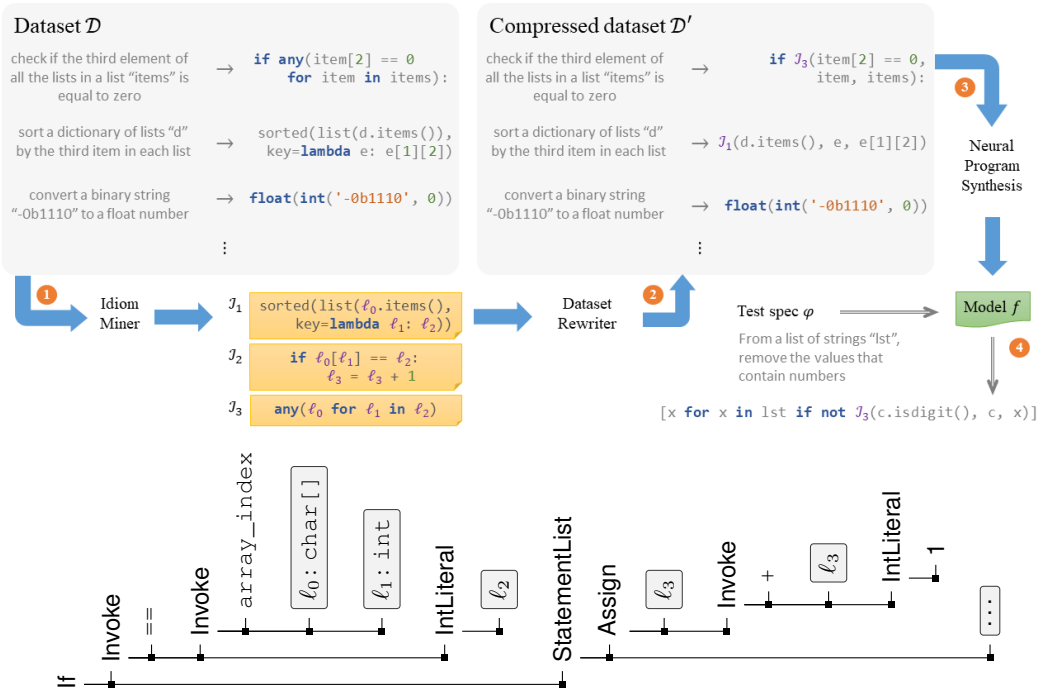


Figure 2: *Top:* An overview of PATOIS. A miner ① extracts common idioms from the programs in a given dataset. The dataset programs are ② compressed by replacing AST subtrees of idiom occurrences with new idiom operators. The compressed dataset ③ is used to train a neural generative model. At inference time, the model ④ generates programs with named idioms, which are inlined before program execution. Notice that idioms involve named subexpressions, may be repeated, and may occur at any program level. For clarity, we typeset idioms using an operator-like syntax $\mathcal{I}_j(\ell_1, \ldots, \ell_k)$, although they are actually represented as AST fragments with no syntax. *Bottom:* AST fragment representation of the idiom $\mathcal{I}_2$ in the NAPS dataset grammar. Here sans-serif nodes are fixed non-terminals, `monospaced` nodes are fixed terminals, boxed nodes are named arguments.

of program synthesis as compared to an equivalent end-to-end generation. However, it allows only exactly one stage of high-level reasoning at the root level of the program, whereas **(a)** real-life programs commonly involve patterns at all syntactic levels, and **(b)** programmers often interleave multiple steps of high-level and low-level reasoning during the implementation process.

Second, many successful applications of program synthesis from input-output examples such as FlashFill (Gulwani, 2011) rely on a manually designed DSL to make the underlying search process scalable. Such DSLs contain high-level operators that implement common patterns for solving subtasks in a given domain. This approach has two key benefits: **(i)** it compresses the search space, making most syntactically valid programs in the DSL uniquely express some useful task in the domain, and **(ii)** it enables logical reasoning over the domain-specific operator semantics, which makes the search process of program synthesis more efficient. However, DSL design is laborious and requires substantial domain expertise. Moreover, DSLs usually do not include domain-independent patterns such as "binary search" or "updating an entry in a dictionary".

In this work, we present a system, called PATOIS, that equips a program synthesizer with automatically learned high-level *code idioms* (*i.e.* common program fragments) and trains it to use these idioms during program generation. While syntactic by definition, code idioms often represent useful semantic concepts. They *compress* and *abstract* the programs by explicitly representing common latent patterns with new unique tokens. When explicitly named and tagged in the training data, idioms are associated with language patterns in the associated task specifications, and simplify the generative process for the synthesis model.

PATOIS has three main components, illustrated in Figure 2. First, it employs nonparameteric Bayesian inference to learn a DSL of code idioms that frequently occur in a given dataset. Second, it rewrites

the training dataset, extending each program with explicit usages of named idioms. Finally, it trains a neural generative model on the rewritten dataset, which allows it to learn a model of idiom usage conditioned on a task specification. During generation, the model has the ability to emit entire idioms in a single step, instead of multiple steps of individual AST nodes or program tokens constituting their definitions. As a result, PATOIS interleaves high-level idioms with low-level tokens at any level during program synthesis, generalizing beyond fixed top-level sketch generation.

We evaluate PATOIS on NAPS, a challenging dataset for synthesis of large programs from algorithmic competitions from natural language specifications with optional input-output examples (Zavershynskyi et al., 2018). We find that equipping the synthesizer with a moderate amount of learned idioms improves its accuracy in generating programs that satisfy the task description.

## 2 OVERVIEW

### 2.1 BACKGROUND

**Program Synthesis** We consider the following formulation of the *program synthesis* problem. Assume an underlying programming language $\mathcal{L}$ of programs. Each program $P \in \mathcal{L}$ can be represented either as a sequence $t_1 \cdots t_{|P|}$ of its *tokens*, or, equivalently, as an *abstract syntax tree (AST)* $T$ parsed according to the context-free grammar (CFG) $\mathcal{G}$ of the language $\mathcal{L}$. A *specification* $\varphi = \langle X, [\langle i_1, o_1\rangle, \ldots, \langle i_m, o_m\rangle]\rangle$ consists of **(i)** a *natural language task description* $X$, represented as a sequence of words $x_1 \cdots x_{|X|}$, and **(ii)** an *optional* set of *input-output examples* $\{\langle i_k, o_k\rangle\}_{k=1}^{m}$, $m \geq 0$. The goal of a program synthesis model $f\colon \varphi \mapsto P$ is to generate a program $P$ that **(i)** maximizes the conditional probability $\Pr(P \mid \varphi)$ *i.e.* the most likely program given the specification, and **(ii)** satisfies the given input-output examples if available: $P(i_k) = o_k \; \forall 1 \leq k \leq m$. We also assume a training set $\mathcal{D} = \{\langle \varphi_j, P_j\rangle\}_{j=1}^{|\mathcal{D}|}$, sampled from an unknown true distribution $\mathfrak{D}$, from which we wish to estimate the conditional probability $\Pr(P \mid \varphi)$.

In this work, we consider imperative programming languages $\mathcal{L}$ with a known context-free grammar $\mathcal{G}$, such as Python. We do not impose any restrictions on the architecture of the generative model $f$. The PATOIS framework is applicable to sequence-to-sequence models (Sutskever et al., 2014), common in semantic parsing, as well as to more sophisticated sequence-to-tree models (Yin & Neubig, 2017) and graph neural networks (Brockschmidt et al., 2018; Li et al., 2016).

**Code Idioms** The key idea of PATOIS is to simplify program synthesis by learning and explicitly demarcating common *code idioms* in the programs $P \in \mathcal{D}$. Following Allamanis & Sutton (2014), we define code idioms as *fragments* $\mathcal{I}$ of valid ASTs $T$ in the CFG $\mathcal{G}$, *i.e.* trees of nonterminals and terminals from $\mathcal{G}$ that may occur as subtrees of valid parse trees from $\mathcal{G}$. A leaf nonterminal of an idiom is an "argument", which must be instantiated with a concrete subtree in each AST instantiation of this idiom. See Figure 2 for an example.

Additionally, we associate a (non-unique) *label* $\ell$ with each nonterminal in every idiom, and require that every instantiation of an idiom $\mathcal{I}$ in a concrete AST $T$ must have its nonterminals with the same label instantiated to identical subtrees. Intuitively, this formulation enables the role of idioms as *higher-order functions*, allowing them to have "named arguments" that are used multiple times in the "body" of an idiom.

### 2.2 THE PATOIS FRAMEWORK

Algorithm 1 shows an overview of the PATOIS framework. At training time, PATOIS performs the following three steps. First, it mines the most common code idioms $\widetilde{\mathcal{I}} = \{\mathcal{I}_1, \ldots, \mathcal{I}_N\}$ from the training set $\mathcal{D}$. Then, it extends the underlying language $\mathcal{L}$ with $N$ new high-level operators representing the idioms, thus creating a higher-level language $\mathcal{L}'$. For this, PATOIS finds all the occurrences of these idioms in the golden programs and replaces them with the newly introduced operators. Finally, it trains a neural program synthesis model $f$ on the rewritten dataset.

At inference time, PATOIS is given a task spec $\varphi$ for a desired program and a new test input $i$. Since the model emits programs in the new language $\mathcal{L}'$ (including new operators representing high-level idioms),

PATOIS rewrites the generated program back in terms of the original language $\mathcal{L}$ before evaluating it on the input $i$.

The framework in Algorithm 1 includes four parameterizable components. Of them, REPLACEOCCURRENCES and INLINEOCCURRENCES involve straightforward tree-to-tree rewriting as defined by the *tree substitution grammar* formalism (Allamanis & Sutton, 2014; Cohn et al., 2010). We detail our implementation of MINECOMMONIDIOMS and TRAINMODEL in Section 3 and Section 4, respectively.

**Algorithm 1** The PATOIS training and inference.

**function** TRAIN-PATOIS(Dataset $\mathcal{D}$)
1: $\{\mathcal{I}_1, \ldots, \mathcal{I}_N\} \leftarrow$ MINECOMMONIDIOMS($\mathcal{D}$)
2: $\mathcal{D}' \leftarrow \varnothing$
3: **for all** training instances $\langle \varphi, P \rangle \in \mathcal{D}$ **do**
4: $\quad P' \leftarrow$ REPLACEOCCURRENCES($P, \mathcal{I}_1, \ldots, \mathcal{I}_N$)
5: $\quad \mathcal{D}' \leftarrow \mathcal{D}' \cup \langle \varphi, P' \rangle$
6: Model $f \leftarrow$ TRAINMODEL($\mathcal{D}'$)
7: **return** $f$

**function** INFER-PATOIS(Model $f$, task spec $\varphi$, input $i$, idioms $\mathcal{I}_1, \ldots, \mathcal{I}_N$)
8: Program $P' \leftarrow f(\varphi)$
9: $P \leftarrow$ INLINEOCCURRENCES($P', \mathcal{I}_1, \ldots, \mathcal{I}_N$)
10: **return** $P(i)$

## 3 MINING CODE IDIOMS

The goal of MINECOMMONIDIOMS is to obtain a set of common and useful AST fragments that we designate as idioms. The trade-off between frequency and usefulness is crucial: it is trivial to simply mine commonly occurring short patterns, but they are often meaningless (Aggarwal & Han, 2014). Instead, we employ and extend the methodology of Allamanis et al. (2018), and frame idiom mining as a nonparameteric Bayesian problem.

We represent idiom mining as inference over *probabilistic tree substitution grammars* (pTSG). A pTSG is a probabilistic context-free grammar extended with production rules that expand to a whole AST fragment instead of a single level of symbols (Cohn et al., 2010; Post & Gildea, 2009). The grammar $\mathcal{G}$ of our original language $\mathcal{L}$ induces a pTSG $\mathcal{G}_0$ with no fragment rules and choice probabilities estimated from the corpus $\mathcal{D}$. To construct a pTSG corresponding to the desired idiom-extended language $\mathcal{L}'$, we define a distribution $\mathfrak{G}$ over pTSGs as follows.

We first choose a Pitman-Yor process (Teh & Jordan, 2010) as a prior distribution $\mathfrak{G}_0$ over pTSGs. It is a nonparameteric process that has proven to be effective for mining code idioms in prior work thanks to its modeling of production choices as a Zipfian distribution (in other words, it implements the desired "rich get richer" effect, which encourages a smaller number of larger *and* more common idioms). Formally, it is a "stick-breaking" process (Sethuraman, 1994) that defines $\mathfrak{G}_0$ as a distribution for each *set of idioms* $\widetilde{\mathcal{I}}_N$ *rooted at a nonterminal symbol* $N$ as

$$\Pr(\mathcal{I} \in \widetilde{\mathcal{I}}_N) \stackrel{\text{def}}{=} \sum_{k=0}^{\infty} \pi_k \, \delta(\mathcal{I} = \mathcal{I}_k) \qquad \mathcal{I}_k \sim \mathcal{G}_0 \tag{1}$$

$$\pi_k \stackrel{\text{def}}{=} u_k \prod_{j=1}^{k-1} (1 - u_j) \qquad u_k \sim \text{Beta}(1 - d,\ \alpha + k \cdot d) \tag{2}$$

where $\delta(\cdot)$ is a delta function, $\alpha$ and $d$ are hyperparameters. See Allamanis et al. (2018) for details.

PATOIS uses $\mathfrak{G}_0$ to compute a posterior distribution $\mathfrak{G}_1 = \Pr(\mathcal{G}_1 \mid T_1, \ldots, T_N)$ via Bayes' rule, where $T_1, \ldots, T_N$ are concrete AST fragments in the training set $\mathcal{D}$. As this calculation is computationally intractable, we approximate it using type-based MCMC (Liang et al., 2010). At each iteration $t$ of the MCMC process, PATOIS generates a pTSG $\mathcal{G}_t$ whose distribution approaches $\mathfrak{G}_1$ as $t \to \infty$. The final idioms are extracted from the pTSG obtained at the last MCMC iteration.

While the Pitman-Yor process definition helps avoid overfitting the idioms to $\mathcal{D}$, not all sampled idioms are useful for the subsequent synthesis process. Thus we *rank* and *filter* the sampled idioms before using them to compress the dataset. In this work, we use the ranking function defined by Allamanis et al. (2018) and introduce a new filtering criterion:

$$\begin{aligned}\text{Coverage-entropy ranking } \mathsf{Score}_{\mathsf{CXE}}(\mathcal{I}) &\stackrel{\text{def}}{=} \text{coverage} \cdot \text{cross-entropy gain} \\ &= \frac{\mathsf{count}(T \in \mathcal{D} \mid \mathcal{I} \in T)}{|\mathcal{D}|} \cdot \frac{1}{|\mathcal{I}|} \log \frac{\Pr_{\mathfrak{G}_1}(\mathcal{I})}{\Pr_{\mathfrak{G}_0}(\mathcal{I})}\end{aligned} \tag{3}$$

$$\text{Reused argument criterion } \mathsf{Filter}_{\mathsf{arg}}(\mathcal{I}) \stackrel{\text{def}}{=} 1 \text{ iff } \exists\, \ell_j \in \mathcal{I} \text{ s.t. } \mathsf{count}(\ell \in \mathcal{I} \mid \ell = \ell_j) > 1 \tag{4}$$

In other words, $\mathsf{Filter}_{\mathsf{arg}}(\mathcal{I})$ accepts an idiom $\mathcal{I}$ for usage iff it contains a named argument $\ell_j$ that is used more than once in the idiom's body.

## 4 USING IDIOMS IN PROGRAM SYNTHESIS

After mining, PATOIS obtains a set of common idioms $\widetilde{\mathcal{I}} = \{\mathcal{I}_1, \ldots, \mathcal{I}_N\}$. We wish for our program synthesis model $f$ to explicitly emit entire idioms $\mathcal{I}_j$ during generation instead of individual AST nodes or program tokens that constitute the definition of $\mathcal{I}_j$ as a fragment. We achieve this by **(a)** extending the programming language $\mathcal{L}$ using the mined idioms as new named operators in the language, and **(b)** rewriting the training set $\mathcal{D}$ *in terms of* the found idioms to allow $f$ to learn a distribution of the idiom usage conditioned on a specification.

First, PATOIS represents each idiom $\mathcal{I}$ as a new named operator $\mathsf{op}_{\mathcal{I}}(\ell_1, \ldots, \ell_k)$ where $\ell_1, \ldots, \ell_k$ are unique nonterminal labels in the definition of $\mathcal{I}$ and $\mathsf{op}_{\mathcal{I}}$ is a unique name. It then extends the language $\mathcal{L}$ by adding these new operators at appropriate grammar levels: $\mathcal{L}' \stackrel{\text{def}}{=} \mathcal{L} \cup \left\{\mathsf{op}_{\mathcal{I}_1}, \ldots, \mathsf{op}_{\mathcal{I}_N}\right\}$.

Next, PATOIS trains a neural synthesis model to emit programs in the new language $\mathcal{L}'$. To provide it with information about occurrences of idioms in the training set $\mathcal{D}$, we take the simplest approach and *rewrite* $\mathcal{D}$ by incorporating the new operators $\mathsf{op}_{\mathcal{I}}$ wherever the corresponding idiomatic code fragments appear in the ground programs. This rewriting process is non-deterministic, since a program may contain multiple overlapping occurrences of different idioms and CFG rules. To resolve this non-determinism, we rely on ranking and filtering of idioms as defined in Section 3, and find their occurrences greedily.[1]

After rewriting, PATOIS obtains a new training set $\mathcal{D}'$, which is used to train a neural program synthesis model $f$. The approach is orthogonal to the choices of model architecture and training objectives, as long as the training is supervised by rewritten new golden programs in $\mathcal{D}'$, either as sequences of tokens or as trees/graphs of AST nodes. In this work, we apply the PATOIS framework to sequence-to-sequence models, which still obtain competitive performance in numerous semantic parsing tasks (Dong & Lapata, 2018; Ling et al., 2016). Extending PATOIS to grammar-based tree decoders would require only to extend the programming language grammar $\mathcal{G}$ using the mined idioms.

To apply sequence-to-sequence models to program synthesis with code idioms, we assume a linearization process on the AST $T$ of the target program $P$. While prior neural synthesis work (*e.g.* by Devlin et al. (2017b)) applies sequence-to-sequence models to the terminal tokens of the program, we assume a linearization that transforms the entire AST, similarly to Jia & Liang (2016). This is because **(i)** we want to explicitly expose the model to the new nonterminal choices in the grammar of $\mathcal{L}'$, and **(ii)** the new idiom operators in $\mathcal{L}'$ have no new surface syntax, hence their syntactic representation as a sequence of terminal tokens is identical to the original program. For example, the idiom fragment in Figure 2 is linearized as

```
( If ( Invoke ( == ( Invoke array_index ℓ0 ℓ1 ) ) )
     ( ( Assign ℓ3 ( Invoke (+ ℓ3 1 ) ) ) ... ) ).
```

In practice, we shorten the linearization as much as possible by omitting obvious nonterminals (*e.g.* `StatementList`), types, inlining single-element lists, etc.

After linearization, each program in the dataset $\mathcal{D}'$ is represented as a sequence $Y$ of tokens $y_1 \cdots y_{|Y|}$ where each token $y_j$ is either a terminal token $t \in P$, a nonterminal symbol token $N \in T$, or an idiom operator token $\mathsf{op}_{\mathcal{I}}$. PATOIS thus trains a sequence-to-sequence model $f : \hat{X} \mapsto Y$ where

$$\hat{X} \stackrel{\text{def}}{=} X \,||\, [\mathsf{emb}(i_1); \mathsf{emb}(o_1)] \,||\, \ldots \,||\, [\mathsf{emb}(i_m); \mathsf{emb}(o_m)] \tag{5}$$

is a sequence representation of the specification $\varphi$ (here $||$ denotes concatenation). In pure semantic parsing (*i.e.*, $m = 0$) we get $\hat{X} = X$. In practice, for the model $f$ we use a bidirectional encoder-

[1] It is possible to *learn* the rewriting process jointly with training the synthesis model $f$. This requires extending the training process with a *non-deterministic oracle*: whenever a idiom appears in a golden program, mark the corresponding token/node choices from $\mathcal{L}$ *as well as* from $\mathcal{L}'$ as correct in the training objective. The objective would thus be amended to allow multiple correct continuations (*i.e.* idiom-based rewritings) of the program being generated in a current training instance, based on the tokens/nodes generated so far. We leave this instantiation of the PATOIS framework to future work.

decoder RNN (Sutskever et al., 2014) with multiplicative attention (Luong et al., 2015) and a pointer mechanism (See et al., 2017) to enable copying tokens from the problem description $X$.

After generation, the linearized AST $Y$ is transformed back into a tree and all the idioms are inlined in order to translate the generated program back into the original language $\mathcal{L}$ for evaluation.

# 5 EVALUATION

## 5.1 DATASETS

We implement and evaluate the PATOIS framework on a challenging program synthesis dataset NAPS (Zavershynskyi et al., 2018). It is a recent dataset of 19,126 problems in the style of algorithmic programming competitions, along with their solutions in an imperative Java-like UAST ("Universal AST") language. It includes a mix of synthetic and real problems, and every problem description is paraphrased by an expert to eliminate commonsense or algorithmic reasoning and make the description more similar to the code that solves the problem. The synthetic (resp. real) descriptions are on average 173 (resp. 93) words long. Every problem also includes a set of $7 \pm 2$ input-output examples.

## 5.2 IMPLEMENTATION

**Idiom mining** We mine the idioms separately using the *training* set of each dataset. Thus, by construction, PATOIS *cannot* indirectly overfit to the test set by learning its idioms, but on the other hand, it cannot generalize beyond the idioms that occur in the training set.

We run type-based MCMC (Section 3) for 10 iterations with $\alpha = 5$ and $d = 0.5$. After ranking and filtering, we take $K$ top-ranked idioms and use them to rewrite the dataset, as described in Section 4. We ran ablation experiments with $K \in \{10, 100, 1000\}$.

**Program synthesis model** As described in Section 4, for all our experiments we used a sequence-to-sequence model with attention and pointer mechanism to represent the synthesizer $f$.[2] Specifically, we use 500-dimensional hidden representations in the encoder and decoder, two-layer RNNs, and a dropout of 0.2 between the RNN layers.

The models are trained using the Adam optimizer (Kingma & Ba, 2015) with $\beta_1 = 0.9$, $\beta_2 = 0.999$ for 25,000 steps, with a starting learning rate of $5 \times 10^{-4}$ and a learning rate decay of $0.5^{t/5000}$. We clip the gradients at 0.5 to improve training stability. For NAPS, we follow the training procedure of Zavershynskyi et al. (2018) and draw batches of size 100 from synthetic or real problem dataset uniformly at random during training. At inference time, we run a beam search of size 64 and pick the best model.

## 5.3 EXPERIMENTAL RESULTS

In each configuration, we compare the performance of equivalent trained models on the same dataset with and without idiomatic rewriting by PATOIS. Following Zavershynskyi et al. (2018), we measure **(i)** exact match accuracy, **(ii)** the number of programs that pass all the input-output examples, **(iii)** the number of programs that pass at least 50% of the input-output examples. Since generated programs are not guaranteed to be resource-efficient, we execute them in a sandbox with a timeout of $10\times$ the runtime of the corresponding ground truth program.

Table 1 shows the experimental results on NAPS test set (NAPS has no predefined dev split). NAPS is an exceedingly difficult dataset: no model, including ones in prior work (Zavershynskyi et al., 2018), has ever achieved a non-zero exact match accuracy on it. However, the models can still generate *semantically equivalent* programs that satisfy the given input-output examples. Our results show that including a moderate number of filtered idioms improves the synthesizer's performance from $3.92\%$ to $4.74\%$ on the perfect accuracy metric and from $5.57\%$ to $5.77\%$ on the "50% examples" accuracy metric. Including more idioms worsens the performance, probably due to the increased sparsity in

[2]Experiments with more advanced tree-based architectures should be completed by the review period; we will post them on the submission forum and update the draft later.

Table 1: Experimental results on the NAPS test set.

| Model | Exact match | IO Accuracy | 50% IO Accuracy |
|---|---|---|---|
| Baseline Seq2Seq | 0.0 | 3.92 | 5.57 |
| Seq2Seq + PATOIS ($K = 1000$, filtering) | 0.0 | 2.06 | 2.88 |
| Seq2Seq + PATOIS ($K = 100$, filtering) | 0.0 | 1.65 | 2.06 |
| Seq2Seq + PATOIS ($K = 10$, filtering) | 0.0 | **4.74** | **5.77** |

the vocabulary. However, we believe that this can be alleviated with *learned dataset rewriting* (as described in Section 4), as it will increase the number of observed idiom occurrences in the data.

## 6 RELATED WORK

**Program synthesis & Semantic parsing** Program synthesis from natural language and input-output examples has a long history in Programming Languages (PL) and Machine Learning (ML) communities (see Gulwani et al. (2017) for a survey). When an input specification is limited to natural language, the resulting problem is an instance of *semantic parsing* (Liang, 2016). There has been a lot of recent interest in applying recurrent sequence-based and tree-based neural networks to semantic parsing (Dong & Lapata, 2016; Jia & Liang, 2016; Li et al., 2016; Yin & Neubig, 2017; Yin et al., 2018b). These approaches commonly use insights from the PL literature, such as grammar-based constraints to reduce the search space, non-deterministic training oracles to enable multiple executable interpretations of intent, and supervision from program execution. They typically either supervise the training on one or more golden programs, or use reinforcement learning to supervise the training from a neural program execution result (Neelakantan et al., 2017). Our PATOIS approach is applicable to any underlying neural semantic parsing model, as long as it is supervised by a corpus of golden programs. In this work, we evaluate PATOIS on sequence-to-sequence models, but more advanced tree-based and graph-based architectures would equally benefit from training on learned code idioms.

**Sketch generation** Two recent works (Dong & Lapata, 2018; Murali et al., 2018) learn abstractions of the target program to compress and abstract the reasoning process of a neural synthesizer. Both of them split the generation process into *sketch generation* and *sketch completion*, wherein the first stage emits a partial tree/sequence (*i.e.* a *sketch* of the program) and the second stage fills in the holes in this sketch. While sketch generation is typically implemented with a neural model, sketch completion can be either a different neural model or a combinatorial search. In contrast to PATOIS, both works define the grammar of sketches explicitly by a manual *program abstraction* procedure and only allow a single top-level sketch for each program. In PATOIS, we learn the program abstractions (code idioms) automatically from a given corpus and allow them to appear anywhere in the program, as is common in real-life programming.

**Learning abstractions** Concurrently with this work, Ellis et al. (2018) developed a Search, Compress & Compile (SCC) framework for automatically learning DSLs for program synthesis from input-output examples (such as the DSLs used by FlashFill (Gulwani, 2011) and DeepCoder (Balog et al., 2017)). The workflow of SCC is similar to PATOIS, with three stages: **(a)** learn new DSL subroutines from a corpus of tasks, **(b)** train a neural recognition model that maps a task specification to a distribution over DSL operators, similarly to DeepCoder (Balog et al., 2017), and **(c)** use these operators in a program synthesizer. PATOIS differs from SCC in three aspects: **(i)** we assume a natural language specification instead of examples, **(i)** to handle NL specifications, our synthesizer is a neural semantic parser model instead of enumerative search, and **(iii)** we discover syntactic idioms that compress general-purpose languages instead of extending DSLs. In both systems, the underlying subroutine mining procedure is a Bayesian process, sampling program fragments that optimize for the subroutines' expressiveness and size.

As described previously, our code idiom mining is an extension of the procedure developed by Allamanis & Sutton (2014); Allamanis et al. (2018). While they are the first to use the tree substitution grammar formalism and Bayesian inference to find non-trivial common idioms in a corpus of code, their problem formalization does not involve any application for the learned idioms beyond their explanatory power. In this work, we show that the mined code idioms correlate with latent semantic

abstractions that are involved in solving programming tasks, and thus using these idioms as explicit programming abstractions improves the performance of neural program synthesis models.

## 7 CONCLUSION

Neural program generation, or synthesis from natural language and input-output examples, has made tremendous progress over the past years, but state-of-the-art models still struggle with generating complex programs that involve multiple levels of abstraction. In this work, we present a framework that allows directly incorporating learned program patterns from the code corpus into the vocabulary of neural program synthesizer, thus enabling it to emit high-level or low-level program constructs interchangeably at each generation step. Our current instantiation of the PATOIS framework uses Bayesian inference to mine common code idioms, inlines them into the dataset as new named operators, and trains a sequence-to-sequence model on the rewritten linearized ASTs. The same workflow is applicable to more advanced tree-based and graph-based generative models, as well as to learned AST rewritings. We show that explicit abstraction of the dataset using idioms improves the performance of neural program synthesis.

PATOIS represents the first step toward learned abstractions in program synthesis. While code idioms correlate with latent semantic concepts that we would like to be used during program generation, our current method does not mine them with the intent to directly optimize their usefulness for program generation. In future work, we want to alleviate this by jointly learning the mining, rewriting, and synthesis models, thus optimizing our idioms for their usefulness for synthesis by construction. We also want to explore incorporating more semantic patterns into the idiom definition, such as natural language phrases from task descriptions or data flow patterns. We believe that such a workflow will push neural synthesis models more toward human-like programming by allowing them to reason directly about semantic program abstractions.

## REFERENCES

Charu C. Aggarwal and Jiawei Han. *Frequent pattern mining*. Springer, 2014.

Miltiadis Allamanis and Charles Sutton. Mining idioms from source code. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, pp. 472–483. ACM, 2014.

Miltiadis Allamanis, Earl T. Barr, Christian Bird, Premkumar Devanbu, Mark Marron, and Charles Sutton. Mining semantic loop idioms. *IEEE Transactions on Software Engineering*, 2018.

Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. DeepCoder: Learning to write programs. In *Proceedings of the 5th International Conference on Learning Representations (ICLR)*, 2017.

Marc Brockschmidt, Miltiadis Allamanis, Alexander L. Gaunt, and Oleksandr Polozov. Generative code modeling with graphs. *CoRR*, abs/1805.08490, 2018.

Trevor Cohn, Phil Blunsom, and Sharon Goldwater. Inducing tree-substitution grammars. *Journal of Machine Learning Research*, 11(Nov):3053–3096, 2010.

Jacob Devlin, Rudy Bunel, Rishabh Singh, Matthew Hausknecht, and Pushmeet Kohli. Neural program meta-induction. In *Advances in Neural Information Processing Systems (NIPS)*, pp. 2080–2088, 2017a.

Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. RobustFill: Neural program learning under noisy I/O. In *Proceedings of the 34th International Conference on Machine Learning (ICML)*, 2017b.

Li Dong and Mirella Lapata. Language to logical form with neural attention. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (ACL)*, 2016.

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 (ACL)*, 2018.

Kevin Ellis, Lucas Morales, Mathias Sablé-Meyer, Armando Solar-Lezama, and Josh Tenenbaum. Search, compress, compile: Library learning in neurally-guided Bayesian program learning. In *Advances in Neural Information Processing Systems (NIPS)*, 2018. To appear.

Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the 38th ACM Symposium on Principles of Programming Languages (POPL)*, volume 46, pp. 317–330, 2011.

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

Robin Jia and Percy Liang. Data recombination for neural semantic parsing. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (ACL)*, volume 1, pp. 12–22, 2016.

Ashwin Kalyan, Abhishek Mohta, Oleksandr Polozov, Dhruv Batra, Prateek Jain, and Sumit Gulwani. Neural-guided deductive search for real-time program synthesis from examples. In *Proceedings of the 6th International Conference on Learning Representations (ICLR)*, 2018.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *Proceedings of 5th International Conference on Learning Representations (ICLR)*, 2015.

Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. Gated graph sequence neural networks. In *Proceedings of the 4th International Conference on Learning Representations (ICLR)*, 2016.

Percy Liang. Learning executable semantic parsers for natural language understanding. *Communications of the ACM*, 59(9):68–76, 2016.

Percy Liang, Michael I. Jordan, and Dan Klein. Type-based MCMC. In *Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the Association for Computational Linguistics*, pp. 573–581. Association for Computational Linguistics, 2010.

Wang Ling, Phil Blunsom, Edward Grefenstette, Karl Moritz Hermann, Tomáš Kočiskỳ, Fumin Wang, and Andrew Senior. Latent predictor networks for code generation. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (ACL)*, volume 1, pp. 599–609, 2016.

Thang Luong, Hieu Pham, and Christopher D. Manning. 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.

Vijayaraghavan Murali, Letao Qi, Swarat Chaudhuri, and Chris Jermaine. Neural sketch learning for conditional program generation. In *Proceedings of the 6th International Conference on Learning Representations (ICLR)*, 2018.

Arvind Neelakantan, Quoc V. Le, Martin Abadi, Andrew McCallum, and Dario Amodei. Learning a natural language interface with neural programmer. In *Proceedings of the 5th International Conference on Learning Representations (ICLR)*, 2017.

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 (OOPSLA)*, pp. 107–126, 2015.

Matt Post and Daniel Gildea. Bayesian learning of a tree substitution grammar. In *Proceedings of the ACL-IJCNLP 2009 Conference Short Papers*, pp. 45–48. Association for Computational Linguistics, 2009.

Abigail See, Peter J. Liu, and Christopher D. Manning. Get to the point: Summarization with pointer-generator networks. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (ACL)*, volume 1, pp. 1073–1083, 2017.

Jayaram Sethuraman. A constructive definition of Dirichlet priors. *Statistica sinica*, pp. 639–650, 1994.

Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks. In *Advances in Neural Information Processing Systems (NIPS)*, pp. 3104–3112, 2014.

Yee Whye Teh and Michael I. Jordan. Hierarchical Bayesian nonparametric models with applications. *Bayesian nonparametrics*, 1:158–207, 2010.

Pengcheng Yin and Graham Neubig. A syntactic neural model for general-purpose code generation. In *The 55th Annual Meeting of the Association for Computational Linguistics (ACL)*, Vancouver, Canada, July 2017.

Pengcheng Yin, Bowen Deng, Edgar Chen, Bogdan Vasilescu, and Graham Neubig. Learning to mine aligned code and natural language pairs from StackOverflow. In *International Conference on Mining Software Repositories (MSR)*, pp. 476–486. ACM, 2018a.

Pengcheng Yin, Chunting Zhou, Junxian He, and Graham Neubig. StructVAE: Tree-structured latent variable models for semi-supervised semantic parsing. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (ACL)*, 2018b.

Maksym Zavershynskyi, Alex Skidanov, and Illia Polosukhin. NAPS: Natural program synthesis dataset. In *2nd Workshop on Neural Abstract Machines & Program Induction*, 2018.

Victor Zhong, Caiming Xiong, and Richard Socher. Seq2SQL: Generating structured queries from natural language using reinforcement learning. *arXiv preprint arXiv:1709.00103*, 2017.