# Learning Programmatic Idioms for Scalable Semantic Parsing

**Srinivasan Iyer[†], Alvin Cheung[§] and Luke Zettlemoyer[†‡]**

[†]Paul G. Allen School of Computer Science and Engineering, Univ. of Washington, Seattle, WA
{sviyer, lsz}@cs.washington.edu
[§]Department of Electrical Engineering and Computer Sciences, UC Berkeley, Berkeley, CA
akcheung@cs.berkeley.edu
‡ Facebook AI Research, Seattle
lsz@fb.com

## Abstract

Programmers typically organize executable source code using high-level coding patterns or idiomatic structures such as nested loops, exception handlers and recursive blocks, rather than as individual code tokens. In contrast, state of the art (SOTA) semantic parsers still map natural language instructions to source code by building the code syntax tree one node at a time. In this paper, we introduce an iterative method to extract code idioms from large source code corpora by repeatedly collapsing most-frequent depth-2 subtrees of their syntax trees, and train semantic parsers to apply these idioms during decoding. Applying idiom-based decoding on a recent context-dependent semantic parsing task improves the SOTA by 2.2% BLEU score while reducing training time by more than 50%. This improved speed enables us to scale up the model by training on an extended training set that is 5× larger, to further move up the SOTA by an additional 2.3% BLEU and 0.9% exact match. Finally, idioms also significantly improve accuracy of semantic parsing to SQL on the ATIS-SQL dataset, when training data is limited.

## 1 Introduction

When programmers translate Natural Language (NL) specifications into executable source code, they typically start with a high-level plan of the major structures required, such as nested loops, conditionals, etc. and then proceed to fill in specific details into these components. We refer to these high-level structures (Figure 1 (b)) as code idioms (Allamanis and Sutton, 2014). In this paper, we demonstrate how learning to use code idioms leads to an improvement in model accuracy and training time for the task of semantic parsing, i.e., mapping intents in NL into general purpose source code (Iyer et al., 2017; Ling et al., 2016).
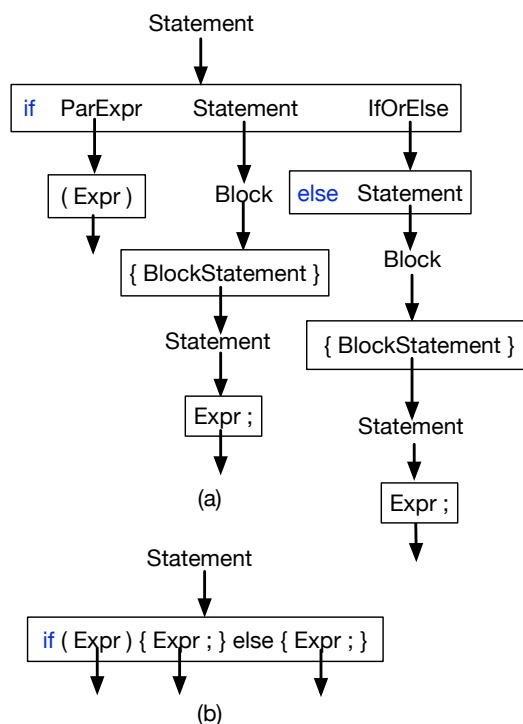


Figure 1: (a) Syntax tree based decoding for semantic parsing uses as many as 11 rules (steps) to produce the outer structure of a very frequently used `if-then-else` code snippet. (b) Direct application of an `if-then-else` idiom during decoding leads to improved accuracy and training time.

State-of-the-art semantic parsers are neural encoder-decoder models, where decoding is guided by the target programming language grammar (Yin and Neubig, 2017; Rabinovich et al., 2017; Iyer et al., 2018) to ensure syntactically valid programs. For general purpose programming languages with large formal grammars, this can easily lead to long decoding paths even for short snippets of code. For example, Figure 1 shows an intermediate parse tree for a generic `if-then-else` code snippet, for which the decoder requires as many as eleven decoding steps

5426

before ultimately filling in the slots for the `if` condition, the `then` expression and the `else` expression. However, the `if-then-else` block can be seen as a higher level structure such as shown in Figure 1 (b) that can be applied in one decoding step and reused in many different programs. In this paper, we refer to frequently recurring subtrees of programmatic parse trees as *code idioms*, and we equip semantic parsers with the ability to learn and directly generate idiomatic structures as in Figure 1 (b).

We introduce a simple iterative method to extract idioms from a dataset of programs by repeatedly collapsing the most frequent depth-2 subtrees of syntax parse trees. Analogous to the byte pair encoding (BPE) method (Gage, 1994; Sennrich et al., 2016) that creates new subtokens of words by repeatedly combining frequently occurring adjacent pairs of subtokens, our method takes a depth-2 syntax subtree and replaces it with a tree of depth-1 by removing all the internal nodes. This method is in contrast with the approach using probabilistic tree substitution grammars (pTSG) taken by Allamanis and Sutton (2014), who use the *explanation quality* of an idiom to prioritize idioms that are more interesting, with an end goal to suggest useful idioms to programmers using IDEs. Once idioms are extracted, we greedily apply them to semantic parsing training sets to provide supervision for learning to apply idioms.

We evaluate our approach on two semantic parsing tasks that map NL into 1) general-purpose source code, and 2) executable SQL queries, respectively. On the first task, i.e., context dependent semantic parsing (Iyer et al., 2018) using the CONCODE dataset, we improve the state of the art (SOTA) by 2.2% of BLEU score. Furthermore, generating source code using idioms results in a more than 50% reduction in the number of decoding steps, which cuts down training time to less than half, from 27 to 13 hours. Taking advantage of this reduced training time, we further push the SOTA on CONCODE to an EM of 13.4 and a BLEU score of 28.9 by training on an extended version of the training set (with 5× the number of training examples). On the second task, i.e., mapping NL utterances into SQL queries for a flight information database (ATIS-SQL; Iyer et al. (2017)), using idioms significantly improves denotational accuracy over SOTA models, when a limited amount of training data is used, and also

marginally outperforms the SOTA when the full training set is used (more details in Section 7).

## 2   Related Work

Neural encoder-decoder models have proved effective in mapping NL to logical forms (Dong and Lapata, 2016) and also for directly producing general purpose programs (Iyer et al., 2017, 2018). Ling et al. (2016) use a sequence-to-sequence model with attention and a copy mechanism to generate source code. Instead of directly generating a sequence of code tokens, recent methods focus on constrained decoding mechanisms to generate syntactically correct output using a decoder that is either grammar-aware or has a dynamically determined modular structure paralleling the structure of the abstract syntax tree (AST) of the code (Rabinovich et al., 2017; Krishnamurthy et al., 2017; Yin and Neubig, 2017). Iyer et al. (2018) use a similar decoding approach but use a specialized context encoder for the task of context-dependent code generation. We augment these neural encoder-decoder models with the ability to decode in terms of frequently occurring higher level idiomatic structures to achieve gains in accuracy and training time.

Another different but related method to produce source code is using sketches, which are code snippets containing slots in the place of low-level information such as variable names, method arguments, and literals. Dong and Lapata (2018) generate such sketches using programming language-specific sketch creation rules and use them as intermediate representations to train token-based seq2seq models that convert NL to logical forms. Hayati et al. (2018) retrieve sketches from a large training corpus and modify them for the current input; Murali et al. (2018) use a combination of neural learning and type-guided combinatorial search to convert existing sketches into executable programs, whereas Nye et al. (2019) additionally also generate the sketches before synthesising programs. Our idiom-based decoder learns to produce commonly used subtrees of programming syntax-trees in one decoding step, where the non-terminal leaves function as slots that can be subsequently expanded in a grammer-aware fashion. Code idioms can be roughly viewed as a tree-structured generalization of sketches, that can be automatically extracted from large code corpora for any programming language, and unlike sketches, can

also be nested with other idioms or grammar rules.

More closely related to the idioms that we use for decoding is Allamanis and Sutton (2014), who develop a system (HAGGIS) to automatically mine idioms from large code bases. They focus on finding interesting and explainable idioms, e.g., those that can be included as preset code templates in programming IDEs. Instead, we learn frequently used idioms that can be easily associated with NL phrases in our dataset. The production of large subtrees in a single step directly translates to a large speedup in training and inference.

Concurrent with our research, Shin et al. (2019) also develop a system (PATOIS) for idiom-based semantic parsing and demonstrate its benefits on the Hearthstone (Ling et al., 2016) and Spider (Yu et al., 2018) datasets. While we extract idioms by collapsing frequently occurring depth-2 AST subtrees and apply them greedily during training, they use non-parametric Bayesian inference for idiom extraction and train neural models to either apply entire idioms or generate its full body.

## 3   Idiom Aware Encoder-Decoder Models

We aim to train semantic parsers having the ability to use idioms during code generation. To do this, we first extract frequently used idioms from the training set, and then provide them as supervision to the semantic parser's learning algorithm.

Formally, if a semantic parser decoder is guided by a grammar $\mathscr{G} = (N, \Sigma, R)$, where $N$ and $\Sigma$ are the sets of non-terminals and terminals respectively, and $R$ is the set of production rules of the form $A \rightarrow \beta, A \in N, \beta \in \{N \cup \Sigma\}^*$, we would like to construct an idiom set $I$ with rules of the form $B \rightarrow \gamma, B \in N, \gamma \in \{N \cup \Sigma\}^*$, such that $B \stackrel{\geq 2}{\Longrightarrow} \gamma$ under $\mathscr{G}$, i.e., $\gamma$ can be derived in two or more steps from $B$ under $\mathscr{G}$. For the example in Figure 1, $R$ would contain rules for expanding each non-terminal, such as *Statement → if ParExpr Statement IfOrElse* and *ParExpr → { Expr }*, whereas $I$ would contain the idiomatic rule *Statement → if ( Expr ) { Expr ; } else { Expr ; }* .

The decoder builds trees from $\hat{\mathscr{G}} = (N, \Sigma, R \cup I)$. Although the set of valid programs under both $\mathscr{G}$ and $\hat{\mathscr{G}}$ are exactly the same, this introduction of ambiguous rules into $\mathscr{G}$ in the form of idioms presents an opportunity to learn shorter derivations. In the next two sections, we describe the idiom extraction process, i.e., how $I$ is chosen, and the idiom application process, i.e., how the de-

coder is trained to learn to apply idioms.

---

**1 Procedure** *Extract-Idioms($\mathscr{D}, \mathscr{G}, n$)*
    **Input:** $\mathscr{D} \rightarrow$ Training Programs
    **Input:** $\mathscr{G} \rightarrow$ Parsing Grammar
    **Input:** $n \rightarrow$ Number of idioms
**2**   $T \leftarrow \{\} \triangleright$ Stores all parse trees
**3**   **for** $d \in \mathscr{D}$ **do**
**4**   $\quad$ $T \leftarrow T \cup$ Parse-Tree$(d, \mathscr{G})$
**5**   **end**
**6**   $I \leftarrow \{\}$
**7**   **for** $i \leftarrow 1$ **to** $K$ **do**
**8**   $\quad$ $s \leftarrow$ Most-frequent
            Depth-2-Subtree$(T)$
**9**   $\quad$ **for** $t \in T$ **do**
**10**  $\quad\quad$ $t \leftarrow$ Collapse-Subtree(t, s)
**11**  $\quad$ **end**
**12**  $\quad$ $I \leftarrow I \cup \{s\}$
**13**  **end**
**14** **end**
**15 Procedure** *Collapse-Subtree($t, s$)*
    **Input:** $t \rightarrow$ parse-tree
    **Input:** $s \rightarrow$ Depth 2 subtree
**16**  **while** $s$ *subtree-of* $t$ **do**
**17**  $\quad$ Replace $s$ in $t$ with Collapse$(s)$
**18**  **end**
**19** **end**
**20 Procedure** *Collapse(s)*
    **Input:** $s \rightarrow$ Depth 2 subtree
**21**  frontier $\leftarrow$ leaves$(s)$
**22**  return Tree$(root(s),$frontier$)$
**23** **end**

**Algorithm 1:** Idiom Extraction.

## 4   Idiom Extraction

Algorithm 1 describes the procedure to add idiomatic rules, $I$, to the regular production rules, $R$. Our goal is to populate the set $I$ by identifying frequently occurring idioms (subtrees) from the programs in training set $\mathscr{D}$. Since enumerating all subtrees of every AST in the training set is infeasible, we observe that all subtrees $s'$ of a frequently occurring subtree $s$ are just as or more frequent than $s$, so we take a bottom-up approach by repeatedly collapsing the most frequent depth-2 subtrees. Intuitively, this can be viewed as a particular kind of generalization of the BPE (Gage, 1994; Sennrich et al., 2016) algorithm for sequences, where new subtokens are created by repeatedly combining frequently occurring adjacent pairs of subtokens. Note that subtrees of parse
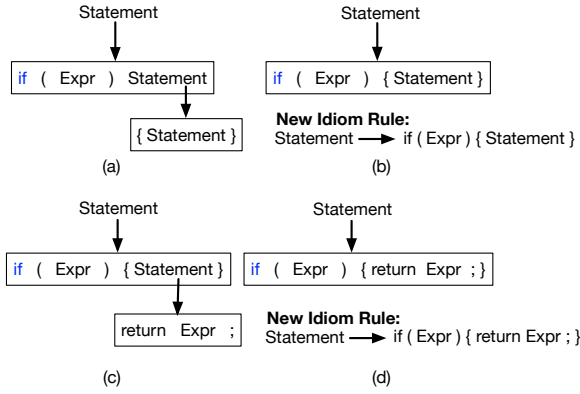
Figure 2: Two steps of the idiom extraction process described in Algorithm 1. (a) First, we find the most frequent depth-2 syntax subtree under Grammar $\mathcal{G}$ in dataset $\mathcal{D}$, collapse it to produce a new production rule (b), and replace all occurrences in the dataset with the collapsed version. Next, (c) is the most frequent depth-2 subtree which now happens to contain (b) within it. (c) is collapsed to form an idiom (d), which is effectively a depth-3 idiom.

trees have an additional constraint, i.e., either all or none of the children of non-terminal nodes are included, since a grammar rule has to be used entirely or not at all.

We perform idiom extraction in an iterative fashion. We first populate $T$ with all parse trees of programs in $\mathcal{D}$ using grammar $\mathcal{G}$ (Step 4). Each iteration then comprises retrieving the most frequent depth-2 subtree $s$ from $T$ (Step 8), followed by post-processing $T$ to replace all occurrences of $s$ in $T$ with a collapsed (depth-1) version of $s$ (Step 10 and 17). The collapse function (Step 20) simply takes a subtree, removes all its internal nodes and attaches its leaves directly to its root (Step 22). The collapsed version of $s$ is a new idiomatic rule (a depth-1 subtree), which we add to our set of idioms, $I$ (Step 12). We illustrate two iterations of this algorithm in Figure 2 ((a)-(b) and (c)-(d)). Assuming (a) is the most frequent depth-2 subtree in the dataset, it is transformed into the idiomatic rule in (b). Larger idiomatic trees are learned by combining several depth-2 subtrees as the algorithm progresses. This is shown in Figure 2 (c) which contains the idiom extracted in (b) within it owing to the post-processing of the dataset after idiom (b) is extracted (Step 10 of Algorithm 1) which effectively makes the idiom in (d), a depth-3 idiom. We perform idiom extraction for $K$ iterations. In our experiments we vary the value of $K$ based on the number of idioms we would like to extract.
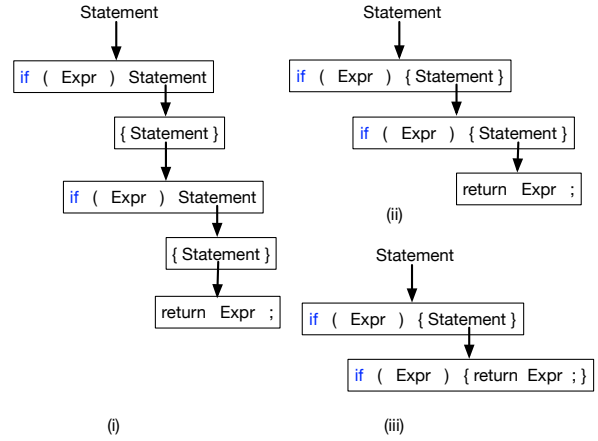


Figure 3: (i) Application of the two idioms extracted in Figure 2 on a new training example. (ii) We first perform two applications of the idiom in Figure 2b, (iii) followed by an application of the idiom in Figure 2d. The resulting tree can be represented with just 2 parsing rules instead of 5.

## 5 Model Training with Idioms

Once a set of idioms $I$ is obtained, we next train semantic parsing models to apply these idioms while decoding. We do this by supervising grammar rule generation in the decoder using a compressed set of rules for each example, using the idiom set $I$ (see Algorithm 2). More concretely, we first obtain the parse tree $t_i$ (or grammar rule set $p_i$) for each training program $y_i$ under grammar $\mathcal{G}$ (Step 3) and then greedily collapse each depth-2 subtree in $t_i$ corresponding to every idiom in $I$ (Step 5). Once $t_i$ cannot be further collapsed, we translate $t_i$ into production rules $r_i$ based on the collapsed tree, with $|r_i| \leq |p_i|$ (Step 7).

```
1  Procedure Compress(𝒟, 𝒢, I)
      Input: 𝒟 → Training Programs
      Input: 𝒢 → Parsing Grammar
      Input: I → Idiom set
2     for d ∈ 𝒟 do
3         t ← Parse-Tree(d, 𝒢)
4         for i ∈ I do
5             t ← Collapse-Subtree(t, i)
6         end
7         Train decoder using
              Production-Rules(t)
8     end
9  end
```

**Algorithm 2:** Training Example Compression.

This process is illustrated in Figure 3 where we

**NL query:** Adds a scalar to this vector in place

**Variables: [Type, Name]**
double[] vecElements
double[] weights
**Methods: [Return Type, Name]**
void inc ( )
float dotProduct ( )
float multiply ( )                                    Context

(a)

**Source code:**
```java
public void add(final double arg0) {
    for (int i = 0; i < vecElements.length(); i++){
        vecElements[i] += arg0;
    }
}
```
**AST Derivation:**
MemberDeclaration ➤ MethodDeclaration
MethodDeclaration ➤
    TypeTypeOrVoid IdentifierNT FormalParameters
    MethodBody
TypeTypeOrVoid ➤ void
IdentifierNT ➤ add
FormalParameters ➤ ( FormalParameterList )
FormalParameterList ➤ FormalParameter
        …
Primary ➤ IdentifierNT
IdentifierNT ➤ arg0

(b)

Figure 4: Context dependent code generation task of Iyer et al. (2018) that involves mapping an NL query together with a set of context variables and methods (each having an identifier name and data type) (a) into source code, represented as a sequence of grammar rules (b).

perform two applications of the first idiom from Figure 2 (b), followed by one application of the second idiom from Figure 2 (d), after which the tree cannot be further compressed using those two idioms. The final tree can be represented using $|r_i| = 2$ rules instead of the original $|p_i| = 5$ rules. The decoder is then trained similar to previous approaches (Yin and Neubig, 2017) using the compressed set of rules. We observe a rule set compression of more than 50% in our experiments (Section 7).

# 6   Experimental Setup

We apply our approach on 1) the context dependent encoder-decoder model of Iyer et al. (2018) on the CONCODE dataset, where we outperform an improved version of their best model, and 2) the task of mapping NL utterances to SQL queries on the ATIS-SQL dataset (Iyer et al., 2017) where an idiom-based model using the full training set outperforms the SOTA, also achieving significant

```sql
List all flights from Denver to Seattle

SELECT DISTINCT flight_1.flight_id FROM
 flight f1, airport_service as1, city c1,
 airport_service as2, city c2 WHERE
 f1.from_airport = as1.airport_code
AND as1.city_code = c1.city_code
AND c1.city_name = "Denver"
AND f1.to_airport = as2.airport_code
AND as2.city_code = c2.city_code
AND c2.city_name = "Seattle";
```

Figure 5: Example NL utterance with its corresponding executable SQL query from the ATIS-SQL dataset.

gains when using a reduced training set.

## 6.1   Context Dependent Semantic Parsing

The CONCODE task involves mapping an NL query together with a class context comprising a list of variables (with types) and methods (with return types), into the source code of a class member function. Figure 4 (a) shows an example where the context comprises variables and methods (with types) that would normally exist in a class that implements a vector, such as vecElements and dotProduct(). Conditioned on this context, the task involves mapping the NL query *Adds a scalar to this vector in place* into a sequence of parsing rules to generate the source code in Figure 4 (b).

Formally, the task is: Given a NL utterance $q$, a set of context variables $\{v_i\}$ with types $\{t_i\}$, and a set of context methods $\{m_i\}$ with return types $\{r_i\}$, predict a set of parsing rules $\{a_i\}$ of the target program. Their best performing model is a neural encoder-decoder with a context-aware encoder and a decoder that produces a sequence of Java grammar rules.

**Baseline Model**   We follow the approach of Iyer et al. (2018) with three major modifications in their encoder, which yields improvements in both speed and accuracy (Iyer-Simp). First, in addition to camel-case splitting of identifier tokens, we use byte-pair encoding (BPE) (Sennrich et al., 2016) on all NL tokens, identifier names and types and embed all these BPE tokens using a single embedding matrix. Next, we replace their RNN that contextualizes the subtokens of identifiers and types with an average of the subtoken embeddings instead. Finally, we consolidate their three separate RNNs for contextualizing NL, variable names with types, and method names with types, into a single shared RNN, which greatly reduces the number of model parameters. Formally, let

$\{q_i\}$ represent the set of BPE tokens of the NL, and $\{t_{ij}\}$, $\{v_{ij}\}$, $\{r_{ij}\}$ and $\{m_{ij}\}$ represent the $j$th BPE token of the $i$th variable type, variable name, method return type, and method name respectively. First, all these elements are embedded using a BPE token embedding matrix B to give us $\mathbf{q_i}$, $\mathbf{t_{ij}}$, $\mathbf{v_{ij}}$, $\mathbf{r_{ij}}$ and $\mathbf{m_{ij}}$. Using Bi-LSTM $f$, the encoder then computes:

$$h_1, \cdots, h_z = f(\mathbf{q_1}, \ldots, \mathbf{q_z}) \quad (1)$$

$$\mathbf{v_i} = Avg(\mathbf{v_{i1}}, \ldots, \mathbf{v_{ij}}) \quad (2)$$

$$\text{Similarly, compute } \mathbf{m_i}, \mathbf{t_i}, \mathbf{r_i} \quad (3)$$

$$\hat{t}_i, \hat{v}_i = f(\mathbf{t_i}, \mathbf{v_i}) \quad (4)$$

$$\hat{r}_i, \hat{m}_i = f(\mathbf{r_i}, \mathbf{m_i}) \quad (5)$$

Then, $h_1, \ldots, h_z$, and $\hat{t}_i, \hat{v}_i, \hat{r}_i, \hat{m}_i$ are passed to the attention mechanism in the decoder, exactly as in Iyer et al. (2018). The decoder remains the same as described in Iyer et al. (2018), and produces a probability distribution over grammar rules at each time step (full details in Supplementary Materials). This forms our baseline model (Iyer-Simp).

**Idiom Aware Training** To utilize idioms, we augment this decoder by retrieving the top-K most frequent idioms from the training set (Algorithm 1), followed by post-processing the training set by greedily applying these idioms (Algorithm 2; we denote this model as Iyer-Simp-K). We evaluate all our models on the CONCODE dataset which was created using Java source files from `github.com`. It contains 100K tuples of (NL, code, context) for training, 2K tuples for development, and 2K tuples for testing. We use a BPE vocabulary of 10K tokens (for matrix $B$) and get the best validation set results using the original hyper-parameters used by Iyer et al. (2018). Since idiom aware training is significantly faster than without idioms, it enables us to train on an additional 400K training examples that Iyer et al. (2018) released as part of CONCODE. We report exact match accuracy, corpus level BLEU score (which serves as a measure of partial credit) (Papineni et al., 2002), and training time for all these configurations.

### 6.2 Semantic Parsing to SQL

This task involves mapping NL utterances into executable SQL queries. We use the ATIS-SQL dataset (Iyer et al., 2017) that contains NL questions posed to a flight database, together with their SQL queries and a database with 25 tables

| Model | Exact | BLEU |
|---|---|---|
| Seq2Seq† | 3.2 (2.9) | 23.5 (21.0) |
| Seq2Prod† | 6.7 (5.6) | 21.3 (20.6) |
| Iyer et al. (2018)† | 8.6 (7.1) | 22.1 (21.3) |
| Iyer-Simp | 12.5 (9.8) | 24.4 (23.2) |
| Iyer-Simp + 200 idioms | 12.2 (9.8) | **26.6 (24.0)** |

Table 1: Exact Match and BLEU scores for our simplified model (Iyer-Simp) with and without idioms, compared with results from Iyer et al. (2018)† on the test (validation) set of CONCODE. Iyer-Simp achieves significantly better EM and BLEU score and reduces training time from 40 hours to 27 hours. Augmenting the decoding process with 200 code idioms further pushes up BLEU and reduces training time to 13 hours.

| Model | Exact | BLEU | Training Time (h) |
|---|---|---|---|
| Seq2Seq† | 2.9 | 21.0 | 12 |
| Seq2Prod† | 5.6 | 20.6 | 36 |
| Iyer et al. (2018)† | 7.1 | 21.3 | 40 |
| Iyer-Simp | 9.8 | 23.2 | 27 |
| + 100 idioms | 9.8 | 24.5 | 15 |
| + 200 idioms | 9.8 | 24.0 | 13 |
| + 300 idioms | 9.6 | 23.8 | 12 |
| + 400 idioms | 9.7 | 23.8 | 11 |
| + 600 idioms | 9.9 | 22.7 | 11 |

Table 2: Variation in Exact Match, BLEU score, and training time on the validation set of CONCODE with number of idioms used. After top-200 idioms are used, accuracies start to reduce, since using more specialized idioms can hurt model generalization. Training time plateaus after considering top-600 idioms.

to execute them against (Figure 5 shows an example). The dataset is split into 4,379 training, 491 validation, and 448 testing examples following Kwiatkowski et al. (2011).

The SOTA by Iyer et al. (2017) is a Seq2Seq model with attention and achieves a denotational accuracy of 82.5% on the test set. Since using our idiom-based approach requires a model that uses grammar-rule based decoding, we use a modified version of the Seq2Prod model described in Iyer et al. (2018) (based on Yin and Neubig (2017)) as a baseline model (Seq2Prod), and augment the decoder with SQL idioms (Seq2Prod-K).

Seq2Prod is an encoder-decoder model, where the encoder executes an n-layer bi-LSTM over NL embeddings and passes the final layer LSTM

| Model | Exact | BLEU |
|---|---|---|
| 1× Train | 12.0 (9.7) | 26.3 (23.8) |
| 2× Train | 13.0 (10.3) | 28.4 (25.2) |
| 3× Train | 13.3 (10.4) | 28.6 (26.5) |
| 5× Train | 13.4 (11.0) | 28.9 (26.6) |

Table 3: Exact Match and BLEU scores on the test (validation) set of CONCODE by training Iyer-Simp-400 on the extended training set released by Iyer et al. (2018). Significant improvements in training speed after incorporating idioms makes training on large amounts of data possible.

hidden states to an attention mechanism in the decoder. Note that the Seq2Prod encoder described in Iyer et al. (2018) encodes a concatenated sequence of NL and context, but ATIS-SQL instances do not include contextual information. Thus, if $q_i$ represents each lemmatized token of the NL, they are first embedded using a token embedding matrix $B$ to give us $\mathbf{q_i}$. Using Bi-LSTM $f$, the encoder then computes:

$$h_1, \cdots, h_z = f(\mathbf{q_1}, \ldots, \mathbf{q_z}) \quad (6)$$

Then, $h_1, \ldots, h_z$ are passed to the attention mechanism in the decoder.

The sequential LSTM-decoder uses attention and produces a sequence of grammar rules $\{a_t\}$. The decoder hidden state at time $t$, $s_t$, is computed based on an embedding of the current non-terminal $n_t$ to be expanded, an embedding of the previous production rule $a_{t-1}$, an embedding of the parent production rule, $\text{par}(n_t)$, that produced $n_t$, the previous decoder state $s_{t-1}$, and the decoder state of the LSTM cell that produced $n_t$, denoted as $s_{n_t}$.

$$s_t = \text{LSTM}_f(n_t, a_{t-1}, \text{par}(n_t), s_{t-1}, s_{n_t}) \quad (7)$$

$s_t$ is then used for attention and finally, produces a distribution over grammar rules.

We make two modifications in this decoder. First, we remove the dependence of $\text{LSTM}_f$ on the parent LSTM cell state $s_{n_t}$. Second, instead of using direct embeddings of rules $a_{t-1}$ and $\text{par}(n_t)$ in $\text{LSTM}_f$, we use another Bi-LSTM across the left and right sides of the rule (using separator symbol SEP) and use the final hidden state as inputs to $\text{LSTM}_f$ instead. More concretely, if a grammar

| Model | Accuracy |
|---|---|
| Iyer et al. (2017)† | 82.5 |
| Seq2Prod | 79.1 |
| **Seq2Prod + 400 idioms** | **83.2** |

Table 4: Denotational Accuracy for Seq2Prod with and without idioms, compared with results from Iyer et al. (2017)† on the test set of ATIS using SQL queries. Results averaged over 3 runs.
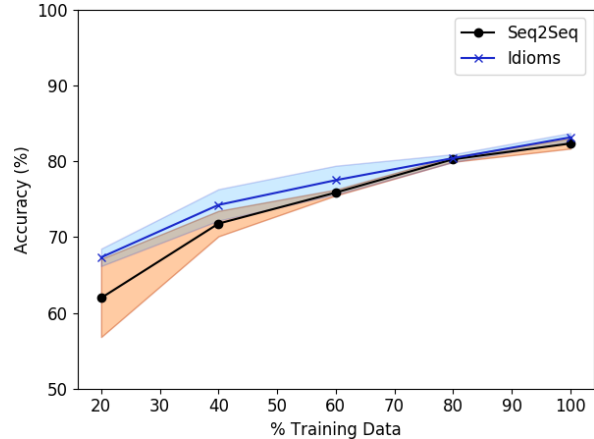


Figure 6: Accuracy of the Seq2Seq baseline (Iyer et al., 2017) compared with Seq2Prod-K, on the test set of ATIS-SQL for varying fractions of training data. Idioms significantly boost accuracy when training data is scarce. (Mean and Std. Dev. computed across 3 runs). 20-40% use 100 idioms and the rest use 400 idioms.

rule is represented as $A \rightarrow B_1 \ldots B_n$, then:

$$\begin{aligned} \text{Emb}(A \rightarrow B_1 \ldots B_n) &= \\ \text{LSTM}_g(\{A, \text{SEP}, B_1, \ldots, B_n\}) \\ s_t = \text{LSTM}_f(n_t, \text{Emb}(a_{t-1}), \text{Emb}(\text{par}(n_t)), \\ s_{t-1}) \quad (8) \end{aligned}$$

This modification can help the $\text{LSTM}_f$ cell locate the position of $n_t$ within rules $a_{t-1}$ and $\text{par}(n_t)$, especially for lengthy idiomatic rules. We present a full description of this model with all hyperparameters in the supplementary materials.

**Idiom Aware Training** As before, we augment the set of decoder grammar rules with top-K idioms extracted from ATIS-SQL. To represent SQL queries as grammar rules, we use the python `sqlparse` package.

## 7 Results and Discussion

Table 1 presents exact match and BLEU scores on the original CONCODE train/validation/test split.
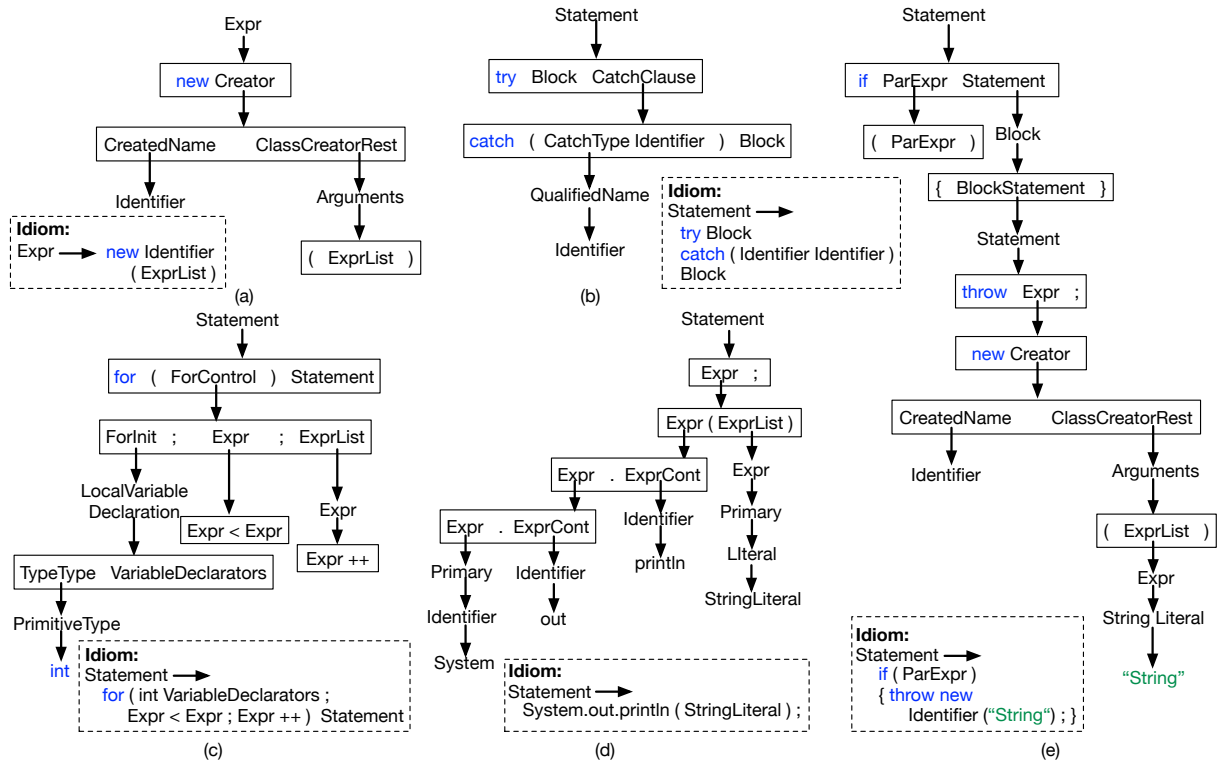
Figure 7: Examples of idioms learned from CONCODE. (a)-(c) represent idioms for instantiation of a new object, exception handling and integer-based looping respectively. (d) represents an idiom for applying a very commonly used library method (`System.out.println`). (e) is a combination of various idioms viz. an if-then idiom, an idiom for throwing exceptions and finally, reusing idiom (a) for creating new objects.

Iyer-Simp yields a large improvement of 3.9 EM and 2.2 BLEU over the best model of Iyer et al. (2018), while also being significantly faster (27 hours for 30 training epochs as compared to 40 hours). Using a reduced BPE vocabulary makes the model memory efficient, which allows us to use a larger batch size that in turn speeds up training. Furthermore, using 200 code idioms further improves BLEU by 2.2% while maintaining comparable EM accuracy. Using the top-200 idioms results in a target AST compression of more than 50%, which results in fewer decoder RNN steps being performed. This reduces training time further by more than 50%, from 27 hours to 13 hours.

In Table 2, we illustrate the variations in EM, BLEU and training time with the number of idioms. We find that 200 idioms performs best overall in terms of balancing accuracy and training time. Adding more idioms continues to reduce training time, but accuracy also suffers. Since we permit idioms to contain identifier names to capture frequently used library methods, having too many idioms hurts generalization, especially since the test set is built using repositories disjoint from

the training set. Finally, the amount of compression, and therefore the training time, plateaus after the top-600 idioms are incorporated.

Compared to the model of Iyer et al. (2018), our significantly reduced training time enables us to train on their extended training set. We run Iyer-Simp using 400 idioms (taking advantage of even lower training time) on up to 5 times the amount of data, while making sure that we do not include in training any NL from the validation or the test sets. Since the original set of idioms learned from the original training set are quite general, we directly use them rather than relearn the idioms from scratch. We report EM and BLEU scores for different amounts of training data on the same validation and test sets as CONCODE in Table 3. In general, accuracies increase with the amount of data with the best model achieving a BLEU score of 28.9 and EM of 13.4.

Figure 7 shows example idioms extracted from CONCODE: (a) is an idiom to construct a new object with arguments, (b) represents a `try-catch` block, and, (c) is an integer-based `for` loop. In (e), we show how small idioms are combined to

form larger ones; it combines an `if-then` idiom with a `throw-exception` idiom, which throws an object instantiated using idiom (a). The decoder also learns idioms to directly generate common library methods such as `System.out.println( StringLiteral )` in one decoding step (d).

For the NL to SQL task, we report denotational accuracy in Table 4. We observe that Seq2Prod underperforms the Seq2Seq model of Iyer et al. (2017), most likely because a SQL query parse is much longer than the original query. This is remedied by using top-400 idioms, which compresses the decoded sequence size, marginally outperforming the SOTA (83.2%). Finegan-Dollak et al. (2018) observed that the SQL structures in ATIS-SQL are repeated numerous times in both train and test sets, thus facilitating Seq2seq models to memorize these structures without explicit idiom supervision. To test a scenario with limited repetition of structures, we compare Seq2Seq with Seq2Prod-K for limited training data (increments of 20%) and observe that (Figure 6) idioms are additionally helpful with lesser training data, consistent with our intuition.

## 8 Conclusions

We presented a general approach to make semantic parsers aware of target idiomatic structures, by first identifying frequently used idioms, followed by providing models with supervision to apply these idioms. We demonstrated this approach on the task of context dependent code generation where we achieved a new SOTA in EM accuracy and BLEU score. We also found that decoding using idioms significantly reduces training time and allows us to train on significantly larger datasets. Finally, our approach also outperformed the SOTA for a semantic parsing to SQL task on ATIS-SQL, with significant improvements under a limited training data regime.

## Acknowledgements

## References

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

Li Dong and Mirella Lapata. 2016. Language to logical form with neural attention. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 33–43. Association for Computational Linguistics.

Li Dong and Mirella Lapata. 2018. 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)*, volume 1, pages 731–742.

Catherine Finegan-Dollak, Jonathan K Kummerfeld, Li Zhang, Karthik Ramanathan, Sesh Sadasivam, Rui Zhang, and Dragomir Radev. 2018. Improving text-to-sql evaluation methodology. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 351–360.

Philip Gage. 1994. A new algorithm for data compression. *The C Users Journal*, 12(2):23–38.

Shirley Anugrah Hayati, Raphael Olivier, Pravalika Avvaru, Pengcheng Yin, Anthony Tomasic, and Graham Neubig. 2018. Retrieval-based neural code generation. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 925–930. Association for Computational Linguistics.

Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, Jayant Krishnamurthy, and Luke Zettlemoyer. 2017. 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)*, pages 963–973, Vancouver, Canada. Association for Computational Linguistics.

Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2018. Mapping language to code in programmatic context. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 1643–1652. Association for Computational Linguistics.

Jayant Krishnamurthy, Pradeep Dasigi, and Matt Gardner. 2017. Neural semantic parsing with type constraints for semi-structured tables. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pages 1516–1526, Copenhagen, Denmark. Association for Computational Linguistics.

Tom Kwiatkowski, Luke Zettlemoyer, Sharon Goldwater, and Mark Steedman. 2011. Lexical generalization in CCG grammar induction for semantic parsing. In *Proceedings of the 2011 Conference on*

*Empirical Methods in Natural Language Processing*, pages 1512–1523. Association for Computational Linguistics.

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

Vijayaraghavan Murali, Letao Qi, Swarat Chaudhuri, and Chris Jermaine. 2018. Neural sketch learning for conditional program generation.

Maxwell Nye, Luke Hewitt, Joshua Tenenbaum, and Armando Solar-Lezama. 2019. Learning to infer program sketches. *arXiv preprint arXiv:1902.06349*.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting on association for computational linguistics*, pages 311–318.

Maxim Rabinovich, Mitchell Stern, and Dan Klein. 2017. 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)*, pages 1139–1149, Vancouver, Canada. Association for Computational Linguistics.

Rico Sennrich, Barry Haddow, and Alexandra Birch. 2016. Neural machine translation of rare words with subword units. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1715–1725, Berlin, Germany. Association for Computational Linguistics.

Richard Shin, Miltiadis Allamanis, Marc Brockschmidt, and Oleksandr Polozov. 2019. Program synthesis and semantic parsing with learned code idioms. *arXiv preprint arXiv:1906.10816*.

Pengcheng Yin and Graham Neubig. 2017. 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)*, pages 440–450, Vancouver, Canada. Association for Computational Linguistics.

Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, et al. 2018. 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*, pages 3911–3921.