Sketch-Driven Regular Expression Generation from Natural Language and Examples

Xi Ye[◊] **Qiaochu Chen**[◊] **Xinyu Wang**[♠] **Isil Dillig**[◊] **Greg Durrett**[◊]

[◊]Department of Computer Science, The University of Texas at Austin
[♠]Computer Science and Engineering Department, University of Michigan, Ann Arbor
{xiye,qchen,isil,gdurrett}@cs.utexas.edu
xwangsd@umich.edu

Abstract

Recent systems for converting natural language descriptions into regular expressions (regexes) have achieved some success, but typically deal with short, formulaic text and can only produce simple regexes. Real-world regexes are complex, hard to describe with brief sentences, and sometimes require examples to fully convey the user's intent. We present a framework for regex synthesis in this setting where both natural language (NL) and examples are available. First, a semantic parser (either grammar-based or neural) maps the natural language description into an intermediate *sketch*, which is an incomplete regex containing holes to denote missing components. Then a program synthesizer searches over the regex space defined by the sketch and finds a regex that is consistent with the given string examples. Our semantic parser can be trained purely from weak supervision based on correctness of the synthesized regex, or it can leverage heuristically derived sketches. We evaluate on two prior datasets (Kushman and Barzilay, 2013; Locascio et al., 2016) and a real-world dataset from Stack Overflow. Our system achieves state-of-the-art performance on the prior datasets and solves 57% of the real-world dataset, which existing neural systems completely fail on.[1]

1 Introduction

Regular expressions (regexes) are widely used in various domains, but are notoriously difficult to write: `regex` is one of the most popular tags of posts on Stack Overflow, with over 200,000 posts. Recent research has attempted to build semantic parsers that can translate natural language descriptions into regexes, via rule-based techniques (Ranta, 1998), semantic parsing (Kushman and Barzilay, 2013), or seq-to-seq neural network models (Locascio et al., 2016; Zhong et al., 2018a; Park et al., 2019). Although this prior work has achieved relatively high accuracy on benchmark datasets, trained models still do not generalize to real-world applications: These benchmarks describe simple regexes with short natural language descriptions and limited vocabulary.

Real-world regexes are more complex in terms of length and tree-depth, requiring natural language descriptions that are longer and more complicated (Zhong et al., 2018b). Moreover, these descriptions may be under-specified or ambiguous. One way to supplement such descriptions is by including positive/negative examples of strings for the target regex to match. In fact, such examples are typically provided by users posting questions on Stack Overflow. Previous methods cannot leverage the guidance of examples at test time beyond naive postfiltering.

In this paper, we present a framework to exploit both natural language and examples for regex synthesis by means of a *sketch*. Rather than directly mapping the natural language into a concrete regex, we first parse the description into an intermediate representation, called a sketch, which is an incomplete regular expression that contains holes to denote missing components. This representation allows our parser to recognize partial structure and fragments from the natural language without fully committing to the regex's syntax. We then use an off-the-shelf program synthesizer, mildly customized for our task, to

[1]Code and data available at `https://github.com/xiye17/SketchRegex/`.

Transactions of the Association for Computational Linguistics, vol. 8, pp. 679–694, 2020. https://doi.org/10.1162/tacl_a_00339
Action Editor: Luke Zettlemoyer. Submission batch: 3/2020; Revision batch: 7/2020; Published 10/2020.

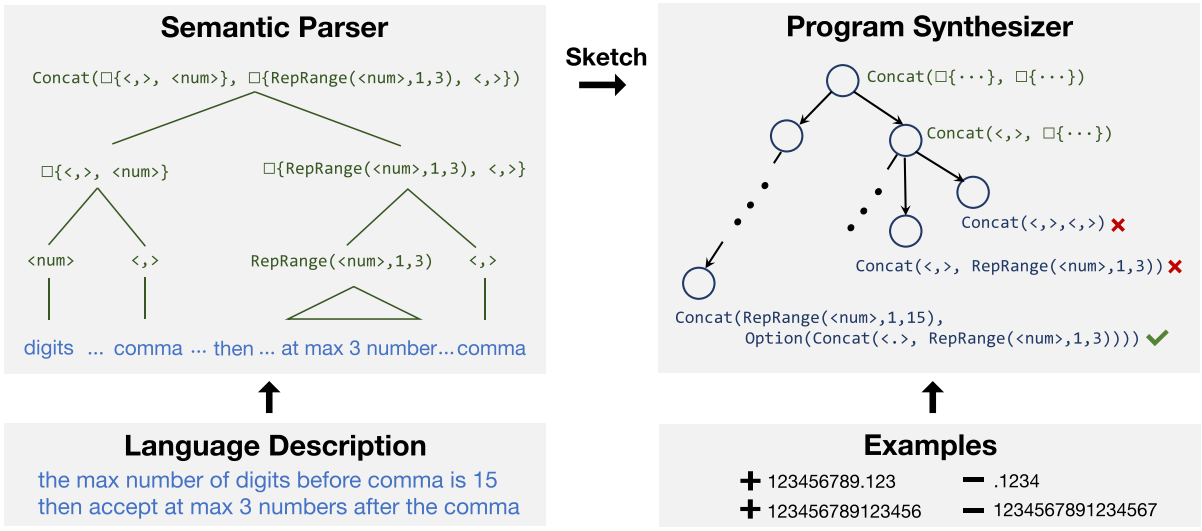

Figure 1: Our regex synthesis approach from language and positive/negative examples. Natural language is parsed into a sketch using a semantic parser. The finished sketch (the root node of the tree) is passed to a program synthesizer, which searches over programs consistent with the sketch and examples. Each leaf node in the search tree is a concrete regex; we return the first one consistent with all the examples.

produce a regex consistent with both the sketch and the provided examples. Critically, this two-stage approach modularizes the language interpretation and program synthesis, allowing us to freely swap out these components.

We evaluate our framework on several English datasets. Because these datasets vary in scale, we consider two sketch generation approaches: neural network-based with a seq-to-seq model (Luong et al., 2015) or grammar-based with a semantic parser (Berant et al., 2013). We use two large-scale datasets from past work, the KB13 dataset of Kushman and Barzilay (2013) and the TURK dataset of Locascio et al. (2016), augmented with automatically produced positive/negative examples for each regex. Our neural sketch model can exploit these large labeled datasets, allowing our sketch-driven approach to outperform existing seq-to-seq methods, even when those methods are modified to take advantage of examples.

To test our model in a more realistic setting, we also evaluate on a dataset of real-world regex synthesis problems from Stack Overflow. These problems organically have English language descriptions and paired examples that the user wrote to communicate their intent. This dataset is small, only 62 examples; to more robustly handle this setting without large-scale training data, we instantiate our sketch framework with a grammar-based semantic parser. Our approach can solve 57% of the benchmarks, where existing deep learning approaches solve less than 10%. More data is needed, but this dataset can motivate further work on more challenging regex synthesis problems.

2 Regex Synthesis Framework

In this section, we illustrate how our regex synthesis framework works using a real-world example from a Stack Overflow post.[2] In this post, the user describes the desired regex as ''*the max number of digits before comma is 15 then accept at max 3 numbers after the comma.*'' Additionally, the user provides eight positive/negative examples to further specify their intent. In this instance, the NL description is under-specified: The description doesn't clearly say whether the decimal part is compulsory, and a period (.) is mistakenly described as a comma. These issues in NL pose problems for systems attempting to directly generate the target regex based only on the description.

Figure 1 shows how our framework handles this example. The natural language description is first parsed into a sketch by a semantic parser, which in this case is grammar-based (Section 3.2) but could also be neural in nature (Section 3.1). The purpose of the sketch is to capture useful

[2] https://stackoverflow.com/questions/19076566/regular-expression-that-validates-decimal-18-3.

components from the description as well as the high-level structure of the regex. For example, the sketch in Figure 1 depicts the target regex as the concatenation of two regexes, where the first regex likely involves composition of `<num>` and `<,>` in some way. We later feed this sketch, together with positive/negative examples, into the synthesizer, which enumeratively instantiates holes with constructs from our regex DSL until a consistent regex is found.

We describe our semantic parsers in Section 3 and our synthesizer in Section 4.

Regex/Sketch DSL Our regex language (Figure 2) is similar to the one presented in Locascio et al. (2016), but more expressive. Our DSL adds some additional constructs, such as `Repeat(S,k)`, repeating a regex `S` exactly `k` times, in our DSL, which is not supported by Locascio et al. (2016). This DSL is equivalent in power to standard regular expressions, in that it can match any regular language.

Our sketch language builds on top of our regex DSL by adding a new construct called a ''constrained hole'' (the red rule in Figure 2). Our sketch DSL introduces an additional grammar symbol S and the notion of hole $\square$. Holes can be produced with the rule $\square\{S_1, \ldots, S_m\}$, where each S_i on the right-hand side is also a sketch. A concrete regex r belongs to the space of regexes defined by a constrained hole if at least one of its S_i defines any concrete regex r, that is, one of the subtrees in r. Put another way, the regex rooted at S must contain a subtree that matches at least one of the S_i, but it does not have to match all of them. However, the synthesizer we use supports using the S_i in its search heuristic to prefer certain programs. In this fashion, the constraint serves as a hint for the leaf nodes of the regex, but it only loosely constraints the structure.

For example, consider the sketch shown in Figure 1. Here, all programs on the leaf nodes of the search tree are included in the space of regexes defined by this sketch. Note that the first two explored regexes only include some of the components mentioned in the sketch (e.g., `<num>` and `RepRange(<num>,1,3)`), whereas the final correct regex happens to include every mentioned component.

Use of Holes There is no single correct sketch for a given example. A trivial sketch consisting of just a single hole could synthesize to a correct program if the examples precisely specify the semantics; this reduces to a pure programming-by-example setting. A sketch could also make no use of holes and fully specify a regex, in which case the synthesizer has no flexibility in its search.

$$
\begin{aligned}
S := {} & C \mid \texttt{StartsWith}(S) \mid \texttt{EndsWith}(S) \\
& \mid \texttt{Contains}(S) \mid \texttt{Optional}(S) \\
& \mid \texttt{Repeat}(S, k) \mid \texttt{KleeneStar}(S) \\
& \mid \texttt{RepAtLeast}(S, k) \\
& \mid \texttt{RepRange}(S, k_1, k_2) \\
& \mid \texttt{Concat}(S, S) \mid \texttt{And}(S, S) \mid \texttt{Or}(S, S) \\
& \mid \square\{S, \ldots, S\}
\end{aligned}
$$

Figure 2: Regex DSL (black) and Sketch DSL (all rules including the last rule in red). C represents either a character class such as `<let>`, `<num>` or a single character such as `<a>`, `<1>`. k represents an integer.

Our process maintains uncertainty over sketches in both training, by leaving them as latent variables, and test, by feeding a k-best list of sketches into our synthesizer. In practice, we observe a few patterns in how sketches are used. One successful pattern is when sketches balance concreteness with flexibility, as shown in Figure 1: They commit to some high-level details to constrain the search space while using holes with specified components further down in the tree. A second pattern we observe is when the sketch has a single hole at the root but enumerates a rich set of components S_i that are likely to appear; this prefers synthesizing sketches using these subtrees.

3 Semantic Parser

Given a natural language description $L = l_1, l_2, \ldots, l_m$, our semantic parser generates a sketch S that encapsulates the user's intent. When combined with examples in the synthesizer, this sketch should yield a regex matching the ground truth regex. As stated before, our semantic parser is a modular component of our system, so we can use different parsers in different settings. We investigate two paradigms of semantic parser: a seq-to-seq neural network parser and a grammar-based parser, as well as two ways of training the parser: maximum likelihood estimation based on a pseudo-gold sketch and maximum marginal likelihood based on whether the sketch leads to the correct synthesis result.

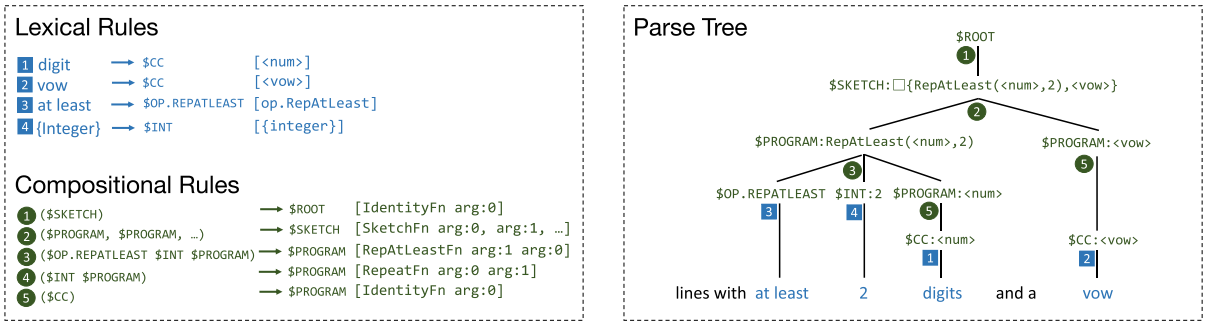

Figure 3: Examples of rules and the parse tree for building one possible derivation. The left side of a rule is the source sequence of tokens or syntactic categories (marked with a $ sign). The right side specifies the target syntactic category and then the target derivation or a semantic function (together with arguments) producing it. $PROGRAM denotes a concrete regex without holes and $SKETCH denotes sketches containing holes. Lexical rule 4 denotes mapping any token of an integer to its value.

3.1 Neural Parser

Following recent work (Locascio et al., 2016), we use a seq-to-seq model with attention (Luong et al., 2015) as our neural parser. Here, we treat the sketch as a sequence of tokens $S = s_1, s_2, \ldots, s_n$ and model $P(S|L)$ autoregressively.

Our encoder is a single-layer bidirectional LSTM, where the tokens l_i in the natural language description are encoded into a sequence of hidden states $\bar{h}_i$. Our decoder is a single-layer unidirectional LSTM, initialized with the encoder final state $\bar{h}_m$. At each timestep t, we concatenate the decoder hidden state $\hat{h}_t$ with a context vector c_t computed based on bilinear attention, and the probability distribution of the output token s_t is given as:

$$a_{i,t} = \text{softmax}\left(\bar{h}_i^\top W_q \hat{h}_t\right) \qquad c_t = \sum_i a_{i,t}\bar{h}_i$$

$$p(s_t|L, s_{<t}) = \text{softmax}\left(W_z[\hat{h}_t; c_t]\right),$$

where $\hat{w}_i$ is the embedded word vector of z_i. The final probability for a generating S conditioned on L is given as $p(S|L) = \prod_{t=1}^{n} p(s_t|L, s_{<t})$.

3.2 Grammar-Based Parser

We also explore a grammar-based semantic parser built using SEMPRE (Berant et al., 2013). This approach is less data-hungry than deep neural networks and promises better generalizability, as it is regulated by a grammar and has fewer parameters, which makes it less likely to fit annotation artifacts of crowdsourced datasets.

Given a natural language description, our semantic parser uses a grammar to construct possible sketches. The grammar consists of two sets of rules, lexical rules and compositional rules. Formally, a grammar rule is of the following form: $\alpha_1 \ldots \alpha_n \rightarrow c[\beta]$. Such a rule maps the sequence of tokens or syntactic categories $\alpha_1 \ldots \alpha_n$ into target derivation β with syntactic category c.

As shown in Figure 3, each lexical rule maps a word or a phrase in the description to a base concept in the DSL, including character classes, string constants, and operators. Compositional rules generally capture the higher-level DSL constructs, specifying how to combine one or more base concepts to build more complex ones. Our semantic parser constructs possible derivations of sketches by recursively applying these rules, first generating derivations for spans matching the lexical rules and then combining these with compositional rules. Finally, we take the the derivations over the entire natural language description with a designated $ROOT category as the final set of output sketches.

We design our grammar[3] according to our sketch DSL. For all the datasets in evaluation, we use a unified grammar that consists of approximately 70 lexical rules and 60 compositional rules. The size of grammar is reflective of the size of DSL, because either a terminal of a single DSL construct needs several rules to specify it (e.g., both *digit* or *number* can present `<num>`, and `Concat(X,Y)` can be described in multiple ways like *X before Y* or *Y follows X*). Despite the fact that the grammar is hand-crafted, it is sufficient to cover the fairly narrow domain of regex descriptions.

[3] A readable version of grammar is available at `https://github.com/xiye17/SketchRegex/blob/master/readable_grammar.pdf`.

Our parser allows skipping arbitrary tokens (Figure 3), resulting in a large number of derivations. We define a log-linear model to place a distribution over derivations $Z \in \mathcal{D}(L)$ given description L: $p_\theta(Z|L) = \frac{\exp(\theta^\top \phi(L,Z))}{\sum_{Z' \in \mathcal{D}(L)} \exp(\theta^\top \phi(L,Z'))}$ where θ is the vector of parameters to be learned, and $\phi(L, Z)$ is a feature vector extracted from the derivation and description. The features used in our semantic parser are standard features in the SEMPRE framework and mainly characterize the relation between description and applied composition, including indicators when rule r is fired over a span containing token l, indicators of whether a particular rule r is fired, and indicators of rule bigrams in the tree.

3.3 Training

For both the neural and grammar-based parsers, we can train the model parameters in two ways.

MLE Maximum likelihood estimation maximizes the probability of mapping the description to a corresponding gold sketch S^*:

$$\arg\max_\theta \sum_{(S^*,L)} \log p_\theta(S^* \mid L).$$

Gold sketches are not defined a priori; however, we describe ways to heuristically derive them in Section 5.2.

MML For a given natural language description and regex pair, multiple syntactically different sketches can yield semantically equivalent regexes. We can therefore maximize the marginal likelihood of generating a sketch that leads us to the semantically correct regex, instead of a generating a particular gold sketch. Namely, we learn the parameters by maximizing:

$$\arg\max_\theta \sum_{(r^*,L)} \log \sum_S 1[\text{synth}(S) = r^*] p_\theta(S \mid L)$$

where r^* is the ground truth regex and synth denotes running the synthesizer. Computing the sum over all sketches is intractable, so we sample sketches from beam search to approximate the gradients (Guu et al., 2017).

4 Program Synthesizer

In this section, we describe the program synthesizer, which takes as input a sketch and a set of examples and returns a regex that is consistent with the given examples. Specifically, our synthesizer explores the space of programs defined by the sketch while additionally being guided by the examples.

Enumerative Synthesis from Sketches We use an enumeration-based program synthesizer that is a generalized version of the regex synthesizer proposed by Lee et al. (2016). Given a program sketch and a set of positive/negative examples, the synthesizer searches the space of programs that can be instantiated by the given sketch and returns a concrete regex that accepts all positive examples and rejects all negative examples.

Specifically, the synthesizer instantiates each hole with our DSL constructs or the components for the hole. If a hole is instantiated with a DSL terminal such as `<num>` or `<let>`, the hole will just be replaced by the terminal. If a hole is instantiated using a DSL operator, the hole will first be replaced by this operator, we introduce new holes for its arguments, and we require the components for at least one of the holes to be the original holes' components. See Figure 1 for an example of regexes that could be instantiated from the given sketch.

Whenever the synthesizer produces a complete instantiation of the sketch (i.e., a concrete regex with no holes), it returns this regex if it is also consistent with the examples (accepts all positive and rejects all negative examples). Otherwise, the synthesizer moves on to the next program in the sketch language. The synthesizer terminates when it either finds a regex consistent with the examples or it has exhausted every possible instantiation of the sketch up to depth d.

Our synthesizer differs from that of Lee et al. (2016) in two main ways. First, their regex language is extremely restricted, only allowing the characters 0 and 1 (a binary alphabet). Second, their technique enumerates DSL programs from scratch, whereas our synthesizer performs enumeration based on an initial sketch. This significantly reduces the search space and therefore allows us to synthesize complex regexes much more quickly.

Enumeration Order Our synthesizer maintains a worklist of partial programs to complete, and enumerates complete programs in increasing order of depth. Specifically, at each step, we pop the

Dataset	KB13	TURK	SO
size	824	10,000	62
#. unique words	207	557	301
Avg. NL length	8	12	25
Avg. regex size	5	5	13
Avg. regex depth	3	2	4

Table 1: Statistics of our datasets. Compared with KB13 and TURK, STACKOVERFLOW contains more sophisticated descriptions and regexes.

KB13
lines where there are two consecutive capital letters
TURK
lines where words include a digit, upper-case letter, plus any letter
STACKOVERFLOW
I'm looking for a regular expression that will match text given the following requirements: contains only 10 digits (only numbers); starts with "9"

Figure 4: Examples of natural language description from each of the three datasets. TURK tends to be very formulaic, while STACKOVERFLOW is longer and much more complex.

next partial program with the highest overlap with our sketch, expand the hole given possible completions, and add the resulting partial programs back to the worklist. When a partial program is completed (i.e., no holes), it is checked against the provided examples. The program will be returned to the user if it is consistent with all the examples, otherwise the worklist algorithm continues.

Note that in this search algorithm, constrained holes are not just hard constraints on the search but are also used to score partial programs, favoring programs using more constructs derived from the natural language. This scoring helps the model prioritize programs that are more congruent with the natural language, lead to more accurate synthesis.

5 Datasets

We evaluate our framework on two datasets from prior work, KB13 and TURK, and a new dataset, STACKOVERFLOW. We list the statistics about these datasets and a typical example from each of them in Table 1 and Figure 4, respectively. Because our framework requires string examples which are absent in the existing datasets, we introduce a systematic way to generate positive/negative examples from ground truth regexes.

KB13 KB13 (Kushman and Barzilay, 2013) was created with crowdsourcing in two steps. First, workers from Amazon Mechanical Turk wrote the original English language descriptions to describe a subset of the lines in a file. Then, a set of programmers from oDesk are required to write the corresponding regex for each of these language descriptions. In total, 834 pairs of description and regex are generated.

Turk Locascio et al. (2016) collected the larger-scale TURK dataset to investigate the performance of deep neural models on regex generation. Because it is challenging and expensive to hire crowd workers with domain knowledge, the authors utilize a generate-and-paraphrase procedure instead. Specifically, 10,000 instances are randomly sampled from a predefined manually crafted grammar that synchronously generates both regexes and synthetic English language descriptions. The synthetic descriptions are then paraphrased by workers from Amazon Mechanical Turk.

The generate-and-paraphrase procedure is an efficient way to obtain description-regex pairs, but it also leads to several issues that we find in the dataset. The paraphrase procedure inherently limits the originality in natural language, leading to artificial descriptions. In addition, because the regexes are stochastically generated without being validated, many of them are syntactically correct but semantically meaningless. For instance, the regex `\b(<vow>)&(<num>)\b` for the description *lines with words containing a vowel and a number* is a valid regex but does not match any string values. These null regexes account for around 15% of the data. Moreover, other regexes have formulaic descriptions since their semantics are randomly made up (more examples can be found in Section 6).

5.1 StackOverflow

To explore regex generation in real-word settings, we collect a new dataset consisting of posts on Stack Overflow. We search posts tagged as regex on Stack Overflow and then filter the collected posts with two rules: (1) the post should include both an English language description as well as positive/negative examples; (2) the post should not contain *abstract concepts* (e.g., ''months'', ''US phone numbers'') or *visual formatting* (e.g.,

''AB-XX-XX'') in the description. We collected 62 posts[4] that contain both description and regex using our rules. In addition, we slightly preprocess the description by fixing typos and marking string constants, as has done in prior datasets (Locascio et al., 2016).

Although StackOverflow only includes 62 examples, the number of unique words in the dataset is higher than that in KB13 (Table 1). Moreover, its average description length and regex size are substantially higher than those of previous datasets, which indicates the complexity of regexes used in real-world settings and the sophistication of language used to describe them.

5.2 Dataset Preprocessing

Generating Positive/Negative Examples The StackOverflow dataset organically has positive/negative examples, but, for the other datasets, we need to generate examples to augment the existing datasets. We use the automaton library (Møller, 2017) for this purpose. For positive examples, we first convert the ground truth regex into an automaton and generate strings by sampling values consistent with paths leading to accepting states in the automaton. For negative examples, we take the negation of the ground truth regex, convert it into an automaton, and follow the same procedure as generating the positive examples. To ensure a diverse set of examples, we limit the number of times that we visit each transition so that the example generator avoids taking the same transitions repeatedly. For each of these datasets, we generate 10 positive and 10 negative examples. This is comparable to what was used in past work (Zhong et al., 2018a) and it is generally hard to automatically generate a smaller set of ''corner cases'' that humans would write.

Generating Heuristic Sketches Our approach does not require any notion of a gold sketch and can operate from weak supervision only. However, we can nevertheless derive *pseudogold* sketches using a heuristic and train with MLE to produce these in order to examine the effects of injecting human prior knowledge into learning. We generate pseudogold sketches from ground truth regexes as follows. For any regex whose Abstract Syntax Tree (AST) has depth > 1, we replace the operator at the root with a constrained hole and the components for this hole are arguments of the original operator. For example, the gold sketch for regex `concat(<num>,<let>)` is `□{<num>,<let>}`. For regexes with depth 1, we just wrap the ground truth regex within a constrained hole; for example, the gold sketch for the regex `<num>` is `□{<num>}`. We apply this method to Turk and KB13.

For the smaller StackOverflow dataset, we explored a more heavily supervised approach where we manually labeled gold sketches based on information from the gold sketch that we judged to be unambiguous about the ground truth regex. For example, the description ''*The input box should accept only if either (1) first 2 letters alpha + 6 numeric or (2) 8 numeric*'' is labeled with the sketch `Or(□{Repeat(<let>, 2),Repeat(<num>,6)}, □{Repeat(<num>), 8))}`, which clearly reflects both the user's intent and the compositional structure.

6 Experiments

Setup We implement all neural models in Pytorch (Paszke et al., 2019). While training with MLE, we use the Adam optimizer (Kingma and Ba, 2015) with a learning rate of 1e-3 and a batch size of 25. We train our models until the loss on the development set converges. When training with MML, we set the learning rate to be 1e-4 and use beam search with beam size 10 to approximate the gradients.

We build our grammar-based parsers on top of the SEMPRE framework (Berant et al., 2013). We use the same grammar for all three datasets. On datasets from prior work, we train our learning-based models with the training set. On StackOverflow, we use 5-fold cross-validation as described in Section 5 because of the limited data size. Our grammar-based parser is always trained for 5 epochs with a batch size of 50 and use a beam size of 200 when trained with MML.

During the testing phase, we produce a k-best list of sketches for a given NL description and run the synthesizer on each sketch in parallel. For a single sketch, the synthesizer either finds an example-consistent regex or running out of a specified time budget (timeout). We pick the output of the highest-ranked sketch yielding an example-consistent regex as the answer. For KB13

[4]These posts are filtered from roughly 1,000 top posts. Despite the fact that more data is available on the Web site, we only view the top posts because the process requires significant human involvement.

and TURK, we set the beam size k to be 20 and set the timeout of synthesizer to be 2s. For the more challenging STACKOVERFLOW dataset, we synthesize top 25 sketches and set the timeout to be 30s. In the experiments, the average time to synthesize a single sketch for a single benchmark in TURK, KB13, and STACKOVERFLOW is 0.4s, 0.8s, and 10.8s, respectively, and we synthesize the k-best lists in parallel using 10 threads.

6.1 Evaluation: KB13 and TURK

Baselines: Prior Work + Translation-based Approaches We compare our approach against several baselines. DEEPREGEX directly translates language descriptions with a seq-to-seq model without looking at the examples using the MLE objective. Note that we compare against both reported numbers from Locascio et al. (2016) as well as our own implementation of this (DEEPREGEX$^{\text{MLE}}$), which outperforms the original by 0.9% and 2.0% on KB13 and TURK, respectively; we use this version in all other reported experiments.

SEMREGEX (Zhong et al., 2018a)[5] uses the same model as DEEPREGEX but is trained to maximize semantic correctness of the gold regex, rather than having to produce an exact match. We implement a similar technique using maximum marginal likelihood training to optimize for semantic correctness (DEEPREGEX$^{\text{MML}}$).

Note that none of these methods assumes access to examples to check correctness at *test* time. To compare these methods to our setting, we extend them in order to exploit examples: we produce the model's k-best list of solutions, then take the highest element in the k-best list consistent with the examples as the answer. We apply this method to both types of training to yield DEEPREGEX$^{\text{MLE}}$+FILTER and DEEPREGEX$^{\text{MML}}$+FILTER.

Sketch-Driven We evaluate three broad types of our sketch-driven models.

Our **No Training** approaches only use untrained sketch procedures. As an example-only baseline, we include the results using an EMPTY SKETCH (a single hole), relying entirely on the synthesizer. We also use a variant of GRAMMARSKETCH method where we heuristically prefer sketch derivations that cover as many words in the input sentence as possible (GRAMMARSKETCH (MAX COVERAGE)).

In the **No Sketch Supervision** setting, we assume no access to labeled sketch data. However, it is challenging to train a neural sketch parser from randomly initialized parameters purely with the MML objective. We therefore warm start the neural models using GRAMMARSKETCH (MAX COVERAGE): We rank the sketches by their coverage of the input sentence, and take the highest-coverage sketch which synthesizes to the correct ground truth regex (if one can be found) as a gold sketch for warm-starting. We can train with the MLE objective for a few epochs and then continue with MML training (DEEPREGEX$^{\text{MML}}$). As a comparison, we can also evaluate the model trained only with MLE with these sketches (DEEPREGEX$^{\text{MLE}}$).

Models in the **Pseudogold Sketches** setting follow the approach described in the previous paragraph, but uses the pseudogold sketches described in Section 5.2 instead of bootstrapping with the grammar-based approach.

Results Table 2 summarizes our experimental results on these two datasets. Note that reported accuracy is semantic accuracy, which measures the functional equivalence of the regex compared to the ground truth. First, we find a significant performance boost by filtering the output of our DEEPREGEX variants using examples (11.2% on KB13 and 21.5% on TURK when applying this to DEEPREGEX$^{\text{MLE}}$), indicating the utility of examples in verifying the produced regexes.

However, our sketch-driven approach outperforms these previous approaches even when they are extended to benefit from examples. We achieve new state-of-the-art results on both datasets, with slightly stronger performance when pseudogold sketches are used. The results are particularly striking in terms of consistency (fraction of regexes produced consistent with the examples). Because we allow uncertainty in the sketches and use examples to guide the construction of regexes, our framework achieves 50% or more relative reduction in the rate of inconsistent regexes compared with DEEPREGEX+FILTER baseline (91.7% and 92.8% on the two datasets), which may fail if no consistent sketch is in the k-best list.

[5] Upon consultation with the authors of SEMREGEX (Zhong et al., 2018a), we were not able to reproduce the results of their model. Therefore, we only include the printed numbers of semantic accuracy on the prior datasets.

	KB13		TURK	
	Acc	Consistent	Acc	Consistent
Prior Work:				
DEEPREGEX (Locascio et al.)	65.6%	–	58.2%	–
SEMREGEX	78.2%	–	62.3%	–
Translation-Based Approaches:				
DEEPREGEX^{MLE}	66.5%	–	60.3%	–
DEEPREGEX^{MML}	68.2%	–	62.4%	–
DEEPREGEX^{MLE} + FILTER	77.7%	89.0%	82.8%	92.0%
DEEPREGEX^{MML} + FILTER	80.1%	91.7%	84.3%	92.8%
Sketch-Driven (No Training):				
EMPTY SKETCH	15.5%	18.4%	21.0%	34.4%
GRAMMARSKETCH (MAX COVERAGE)	68.0%	76.7%	60.2%	78.8%
Sketch-Driven (No Sketch Supervision):				
DEEPSKETCH^{MLE}	76.2%	88.8%	74.6%	92.8%
DEEPSKETCH^{MML}	82.5%	94.2%	84.3%	95.8%
Sketch-Driven (Pseudogold Sketches):				
GRAMMARSKETCH	72.8%	85.4%	69.4%	87.4%
DEEPSKETCH^{MLE} PSEUDOGOLD	84.0%	95.3%	85.4%	98.4%
DEEPSKETCH^{MML} PSEUDOGOLD	**8**6.4%	96.3%	**8**6.2%	98.9%

Table 2: Results on datasets from prior work. We evaluate on both accuracy (Acc) and the fraction of regexes produced consistent (Consistent) with the positive/negative examples. Our sketch-driven approaches outperform prior approaches even when those are modified to use examples. Our approach can leverage heuristic pseudogold sketches, but does not require them. Our DEEPSKETCH models achieve the best results, but even our grammar-based method (GRAMMARSKETCH) outperforms past systems that do not use examples.

We also find that our GRAMMARSKETCH approach, trained with pseudogold sketches, achieves nearly 70% accuracy on both datasets, which is better than DEEPREGEX. This indicates the generalizability of this approach. The performance of GRAMMARSKETCH lags that of DEEPREGEX+FILTER and DEEPSKETCH models, which can be attributed to the fact that GRAMMARSKETCH is more constrained by its grammar and is less capable of exploiting large amounts of data compared to neural approaches.

Finally, we turn to the source of the supervision. The untrained GRAMMARSKETCH (MAX COVERAGE) achieves over 60% accuracy on both datasets; recall that this provides the set of gold sketches as initial supervision in our warm-started model. Our sketch-driven approach trained with MML (DEEPSKETCH^{MML}) achieves 82.5% on KB13 and 84.3% on TURK, which is comparable with the performance obtained using pseudogold sketches, demonstrating that human labeling or curation of sketches is not required for this technique to work well.

6.2 Evaluation: Stack Overflow

Additional Baselines It is impractical to train a deep neural model from scratch on this dataset, so we modify our approach slightly to compare against such models. First, we train a model on TURK and fine-tune it on STACKOVERFLOW (Transferred Model). Second, we explore a modified version of the dataset where we rewrite the descriptions in STACKOVERFLOW to make them conform to the style of TURK (Curated Language), as users might do if they were knowledgeable about the capabilities of the regex synthesis system they are using. For example, we manually paraphrase the original description *''write regular expression in C# to validate that the input does not contain double spaces''* to *''line that does not contain 'space' two or more times''*, and apply DEEPREGEX+FILTER method on

Approach	Top-N Acc		
	top-1	top-5	top-25
DEEPREGEX+FILTER			
Transferred Model	0%	0%	0%
+Curated Language	0%	0%	6.6%
GRAMMARREGEX+FILTER	3.2%	9.7%	11.3%
EMPTY SKETCH	4.8%	–	–
DEEPSKETCH			
Transferred Model	3.2%	3.2%	4.8%
GRAMMARSKETCH			
MAX COVERAGE	16.1%	34.4%	45.2%
MLE, MANUAL SKETCHES	**3**4.4%	48.4%	53.2%
MML, NO SKETCH SUP	31.1%	**5**4.1%	**5**6.5%

Table 3: Results on the STACKOVERFLOW dataset. The DEEPREGEX method totally fails even when the examples are generously rewritten to conform to the model's ''expected'' style. Our GRAMMARSKETCH model can do significantly better, with or without manually labeled sketches.

the curated descriptions (without fine-tuning on them). Note that this simplifies the inputs for these baselines considerably by removing variation in the language.

We also construct a grammar-based regex parser from our GRAMMARSKETCH model by removing the grammar rules related to assembling sketches from regexes. We use our filtering technique as well and call this the GRAMMARREGEX+FILTER baseline.

Results Because STACKOVERFLOW is a challenging dataset, we report the top-N accuracy, where the model is considered correct if any of the top-N sketches synthesizes to a correct answer. Table 3 shows the results on this dataset. The transferred DEEPREGEX model completely fails on these real-world tasks. Rewriting and curating the language, we are able to get some examples correct among the top 25 derivations, but only on very simple cases. GRAMMARREGEX+FILTER is similarly unable to do well: this approach is too inflexible given the complexity of regexes in this dataset. Our transferred DEEPSKETCH approach is also still limited here, as the text is too dissimilar from TURK.

Our GRAMMARSKETCH approach, trained without explicit sketch supervision, achieves a top-1 accuracy of 31.1%. Surprisingly, this is comparable to the performance of GRAMMARSKETCH trained using manually written sketches, and even outperforms this model in terms of top-5 and top-25 accuracy.

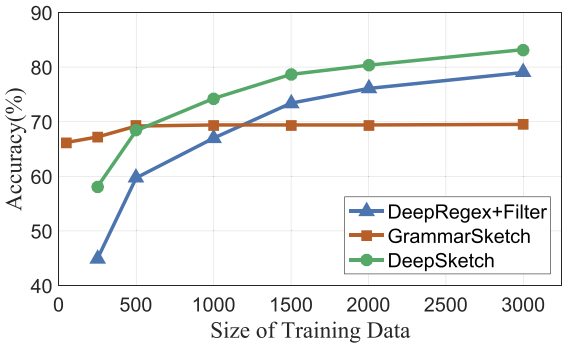

Figure 5: Accuracy on TURK for different training set sizes. Our DEEPSKETCH and GRAMMARSKETCH approaches outperform the DEEPREGEX+FILTER baseline when training data is limited.

This substantially outperforms all of the baseline approaches. We attribute this success to the problem decomposition: Because the sketches produced can be simpler than the full regex, our model is much more robust to the complex setting of this dataset. By examining the problems that are solved, we find our approach is able to solve several complicated cases with long descriptions and sophisticated regexes (e.g., the example in Figure 1).

6.3 Detailed Analysis

Data Efficiency In Figure 5, we explicitly evaluate the data efficiency of our techniques, comparing the performance of DEEPREGEX+FILTER, GRAMMARSKETCH, and DEEPSKETCH on TURK. When the data size is extremely limited (no more than 500), our GRAMMARSKETCH approach is much better than the DEEPREGEX+FILTER baseline. Our DEEPSKETCH approach is more flexible and achieves stronger performance for larger training data sizes, consistently outperforming the DEEPREGEX+FILTER baseline as the training data size increases. More importantly, the difference is particularly obvious when the size of training data is relatively small, in which case correct regexes are less likely to exist in the DEEPREGEX generated k-best lists due to lack of supervision. By contrast, it is easier to learn the mapping from the language to effective sketches, which are simpler than gold regexes.

Overall, these results indicate that DEEPREGEX, as a technique, is only effective when large training sets are available, even for a relatively simple set of natural language expressions.

	4	6	8	10
DEEPREGEX+FILTER	79.5	81.0	82.3	82.8
GRAMMARSKETCH	59.4	64.4	67.6	69.5
DEEPSKETCH	71.8	78.6	82.7	85.4

Table 4: Performance on TURK varying the number of positive/negative examples. Because our synthesizer depends on sufficient examples to constrain the semantics, our sketch-based approaches require a certain number of examples to work well.

	1	3	5	10	20
DEEPREGEX+FILTER	60.3	73.2	76.8	80.3	82.8
GRAMMARSKETCH	58.3	65.0	67.1	68.9	69.5
DEEPSKETCH	69.4	79.7	82.3	84.3	85.4

Table 5: Performance on TURK under different beam sizes.

Impact of Number of Examples We show how the number of positive/negative examples impacts the performance on TURK in Table 4. Our sketch-driven techniques rely on examples to search for the desired regexes in the synthesizer step, and therefore are inferior to DEEPREGEX+FILTER when only limited number of examples are provided. However, as more examples are included, our sketch-driven approaches are more effective in taking advantage of the multi-modality than simply filtering the outputs (DEEPREGEX+FILTER). Note that in the STACKOVERFLOW setting, we evaluate on sets of examples provided by actual users, and find that users typically provide enough examples for our model to be in an effective regime.

Impact of Beam Size We study the effect of varying beam size on the performance on TURK in Table 5. DEEPSKETCH outperforms DEEPREGEX+FILTER by a substantial gap with smaller beam sizes, as we naturally allow uncertainty using the holes in sketches. Note that using larger beam sizes is important for DEEPSKETCH because we feed the entire k-best list to the synthesizer instead of the top sketch only.

6.4 Examples of Success And Failure Pairs

TURK We now analyze the output of the DEEPREGEX+FILTER and DEEPSKETCH. Figure 6 provides some success pairs that DEEPSKETCH solves while DEEPREGEX+FILTER does not. Examples of success pairs suggest that our approach can deal with under-specified descriptions. For instance, in pair (a) from Figure 6, the language is ungrammatical (*3 more* instead of *3 or more*) and also ambiguous: Should the target string consist of only capital letters, or could it have capital letters as well as something else? Our approach is able to recover the faithful semantics using sketch and examples, whereas DEEPREGEX fails to find the correct regex. In pair (b), the description is fully clear but DEEPREGEX still fails because the phrase *none of* rarely appears in the training data. Our approach can solve this pair because it is less sensitive to the description.

We also give some examples of failure cases for our model. These are particularly common in cases of unnatural semantics. For instance, the regex in pair (c) accepts any string except the

TURK	
Success:	
(a) nl:	*lines with 3 more capital letters*
gt:	`(<cap>){3,})(.*)`
(b) nl:	*none of the lines should have a vowel , a capital letter , or the string "dog"*
gt:	`~((<vow>)|(dog)|(<cap>))`
Failure:	
(c) nl:	*lines with "dog" or without "truck" , at least 7 times*
gt:	`((dog)|(~(truck)))){7,}`
error:	`(dog)|(~(truck))`
(d) nl:	*lines ending with lower-case letter or not the string "dog"*
gt:	`(.*)(([<low>])|(~(dog)))`
error:	`(([<low>])|(~(dog)))*`
STACKOVERFLOW	
Success:	
(e) nl:	*valid characters are alphanumeric and "."(period). The patterns are "%d4%" and "%t7%". So "%" is not valid by itself, but has to be part of these specific patterns.*
gt:	`((<let>|<num>|(.)|(%d4%)|(%t7%)){1,}`
(f) nl:	*The input box should accept only if either (1) first 2 letters alpha + 6 numeric or (2) 8 numeric*
gt:	`(<let>{2}<num>{6})|(<num>{8})`
Failure:	
(g) nl:	*I'm trying to devise a regular expression which will accept decimal number up to 4 digits*
gt:	`(<num>{1,})(.)(<num>{1,4})`
error:	`(<num>{1,})((.)(<num>{1,4}))?`
(h) nl:	*the first letter of each string is in upper case*
gt:	`<cap>(<let>)*(( )<cap>(<let>)*)*`
error:	`((( )<cap>)|(<let>)){1,}`

Figure 6: Examples of success and failure pairs from TURK and STACKOVERFLOW. On pairs (a) and (b), our DEEPSKETCH is robust to the issues existing in natural language descriptions. On pairs (c) and (d), our approach fails due to the unrealistic semantics of the desired regexes. GRAMMARSKETCH succeeds in solving some complex pairs in STACKOVERFLOW, including (e) and (f). However, (g) and (h) fail because of insufficient examples or overly concise descriptions.

string *truck* (because ~ (truck) matches any string but *truck*). The semantics are hard to pin down with examples, but the correct regex is also artificial and unlikely to appear in real word applications. Our DEEPSKETCH fails on this pair because the synthesizer fails to catch the *at least 7 times* constraint when strings that have fewer than 7 characters can also be accepted (since *without truck* can match the empty string). DEEPREGEX is able to produce the ground-truth regex in this case, but this is only because the formulaic description is easy enough to translate directly into a regex.

STACKOVERFLOW We show some solved and unsolved examples using GRAMMARSKETCH from the STACKOVERFLOW dataset. Our approach can successfully deal with multiple-sentence inputs like pairs (e) and (f). They both contain multiple sentences with each one describing certain a component or constraint, which seems to be a common pattern of describing real world regexes. Our approach is effective for this structure because the parser can extract fragments from each sentence and hand them to the synthesizer for completion.

Some failure cases are due to lack of corner-case examples. For example, the description from pair (g) doesn't explicitly specify whether the decimal part is required and there are no corner-case negative examples that provide this clue. Our synthesizer mistakenly treats the decimal part as an option, failing to match the ground truth. In addition, pair (h) is an example in which the natural language description is too concise for the grammar parser to generate a useful sketch.

7 Related Work

Other NL and Program Synthesis There has been recent interest in synthesizing programs from natural language. One line of work uses either grammar-based or neural semantic parsing to synthesize programs. Particularly, several techniques have been proposed to translate natural language to SQL queries (Yaghmazadeh et al., 2017; Iyer et al., 2017; Suhr et al., 2018), ''if-this-then-that'' recipes (Quirk et al., 2015), bash commands (Lin et al., 2018), Java expressions (Gvero and Kuncak, 2015) and more. Our work is different from prior work in that it utilizes input-output examples in addition to natural language. Although several past approaches use both natural language and examples (Kulal et al., 2019; Polosukhin and Skidanov, 2018; Zhong et al., 2020), they only use the examples to verify the generated programs, whereas our approach heavily engages examples when searching for the instantiation of sketches to make the synthesizer more efficient.

Another line of work has focused on exploring which deep learning techniques are most effective for directly predicting programs from natural language. Recent work has built encoder-decoder models to generated logical forms or programs represented by sequences (Dong and Lapata, 2016), and ASTs (Rabinovich et al., 2017; Yin and Neubig, 2017; Iyer et al., 2019; Shin et al., 2019). However, some of the most challenging code settings such as the Hearthstone dataset (Ling et al., 2016) only evaluates the produced strings by exact match accuracy or BLEU score, rather than executing the programs on real data as we do.

There is also recent work using neural models to generate logical forms utilizing a coarse-to-fine approach (Zettlemoyer and Collins, 2009; Kwiatkowski et al., 2013; Artzi et al., 2015; Dong and Lapata, 2018; Wang et al., 2019), which first generates an abstract logical form and then concretizes it using neural modules, whereas we complete the sketch via a synthesizer.

Program Synthesis from Examples Recent work has studied program synthesis from examples in other domains (Gulwani, 2011; Alur et al., 2013; Wang et al., 2016; Feng et al., 2018). Similar to prior work (Balog et al., 2017; Kalyan et al., 2018; Odena and Sutton, 2020), we implement an enumeration-based synthesizer to search for the target program, but they use probability distribution of functions or production rules predicted by neural networks to guide the search, whereas our work relies on sketches.

Our method is closely related to sketch-based approaches (Solar-Lezama, 2008; Nye et al., 2019) in that our synthesizer starts with a sketch. However, we produce sketches automatically from the natural language description whereas traditional sketch-based synthesis (Solar-Lezama, 2008) relies on a user-provided sketch, and our sketches are hierarchical and constrained compared to other neural sketch-based approaches (Nye et al., 2019).

8 Conclusion

We have proposed a sketch-driven regular expression synthesis framework that utilizes both natural

language and examples, and we have instantiated this framework with both a neural and a grammar-based parser. Experimental results reveal the artificialness of existing public datasets and demonstrate the advantages of our approach over existing research, especially in real-world settings.

Acknowledgments

This work was partially supported by NSF grant IIS-1814522, NSF grant SHF-1762299, gifts from Arm and Salesforce, and an equipment grant from NVIDIA. The authors acknowledge the Texas Advanced Computing Center (TACC) at The University of Texas at Austin for providing HPC resources used to conduct this research. Thanks as well to our TACL action editor Luke Zettlemoyer and the anonymous reviewers for their helpful comments.

References

R. Alur, R. Bodik, G. Juniwal, M. M. K. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa. 2013. Syntax-guided synthesis. In *2013 Formal Methods in Computer-Aided Design (FMCAD)*. **DOI:** https://doi.org/10.1109/FMCAD.2013.6679385

Yoav Artzi, Kenton Lee, and Luke Zettlemoyer. 2015. Broad-coverage CCG semantic parsing with AMR. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*. **DOI:** https://doi.org/10.18653/v1/D15-1198

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

Jonathan Berant, Andrew Chou, Roy Frostig, and Percy Liang. 2013. Semantic parsing on Freebase from question-answer pairs. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*.

Li Dong and Mirella Lapata. 2016. Language to logical form with neural attention. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics (ACL)*. **DOI:** https://doi.org/10.18653/v1/P16-1004

Li Dong and Mirella Lapata. 2018. Coarse-to-fine decoding for neural semantic parsing. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics (ACL)*. **DOI:** https://doi.org/10.18653/v1/P18-1068, **PMCID:** PMC5995273

Yu Feng, Ruben Martins, Osbert Bastani, and Isil Dillig. 2018. Program synthesis using conflict-driven learning. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. **DOI:** https://doi.org/10.1145/3192366.3192382, **PMCID:** PMC5882843

Sumit Gulwani. 2011. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. **DOI:** https://doi.org/10.1145/1926385.1926423

Kelvin Guu, Panupong Pasupat, Evan Liu, and Percy Liang. 2017. From language to programs: Bridging reinforcement learning and maximum marginal likelihood. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics (ACL)*.

Tihomir Gvero and Viktor Kuncak. 2015. Synthesizing Java expressions from Free-form Queries. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. **DOI:** https://doi.org/10.1145/2814270.2814295

Srinivasan Iyer, Alvin Cheung, and Luke Zettlemoyer. 2019. Learning programmatic idioms for scalable semantic parsing. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*.

Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, Jayant Krishnamurthy, and Luke Zettlemoyer. 2017. Learning a neural semantic parser from user feedback. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics (ACL)*.

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

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

Sumith Kulal, Panupong Pasupat, Kartik Chandra, Mina Lee, Oded Padon, Alex Aiken, and Percy S. Liang. 2019. Spoc: Search-based pseudocode to code. In *Proceedings of the Conference on Advances in Neural Information Processing Systems (NeurIPS)*.

Nate Kushman and Regina Barzilay. 2013. Using semantic unification to generate regular expressions from natural language. In *Proceedings of the 2013 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT)*.

Tom Kwiatkowski, Eunsol Choi, Yoav Artzi, and Luke Zettlemoyer. 2013. Scaling semantic parsers with on-the-fly ontology matching. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*.

Mina Lee, Sunbeom So, and Hakjoo Oh. 2016. Synthesizing regular expressions from examples for introductory automata assignments. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE)*. **DOI:** https://doi.org/10.1145/2993236.2993244

Xi Victoria Lin, Chenglong Wang, Luke Zettlemoyer, and Michael D. Ernst. 2018. NL2Bash: A Corpus and Semantic Parser for Natural Language Interface to the Linux Operating System. In *Proceedings of the International Conference on Language Resources and Evaluation LREC*.

Wang Ling, Phil Blunsom, Edward Grefenstette, Karl Moritz Hermann, Tomáš Kočiský, Fumin Wang, and Andrew Senior. 2016. Latent predictor networks for code generation. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics (ACL)*.

Nicholas Locascio, Karthik Narasimhan, Eduardo DeLeon, Nate Kushman, and Regina Barzilay. 2016. Neural generation of regular expressions from natural language with minimal domain knowledge. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*. **DOI:** https://doi.org/10.18653/v1/D16-1197

Thang Luong, Hieu Pham, and Christopher D. Manning. 2015. Effective approaches to attention-based neural machine translation. In *Proceedings of the Conference on Empirical Methods-in Natural Language Processing (EMNLP)*. **DOI:** https://doi.org/10.18653/v1/D15-1166

Anders Møller. 2017. dk.brics.automaton–finite-state automata and regular expressions for Java. http://www.brics.dk/automaton/.

Maxwell Nye, Luke Hewitt, Joshua Tenenbaum, and Armando Solar-Lezama. 2019. Learning to infer program sketches. In *Proceedings of the International Conference on Machine Learning (ICML)*.

Augustus Odena and Charles Sutton. 2020. Learning to represent programs with property signatures. In *Proceedings of the International Conference on Learning Representations (ICLR)*.

Jun-U Park, Sang-Ki Ko, Marco Cognetta, and Yo-Sub Han. 2019. SoftRegex: Generating regex from natural language descriptions using softened regex equivalence. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing and the International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*. **DOI:** https://doi.org/10.18653/v1/D19-1677

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie

Bai, and Soumith Chintala. 2019. PyTorch: An imperative style, high-performance deep learning library, *Advances in Neural Information Processing Systems (NeurIPS)*.

Illia Polosukhin and Alexander Skidanov. 2018. Neural program search: Solving programming tasks from description and examples. In *Workshop at the International Conference on Learning Representations (ICLR Workshop)*.

Chris Quirk, Raymond Mooney, and Michel Galley. 2015. Language to code: Learning semantic parsers for if-this-then-that recipes. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics and the International Joint Conference on Natural Language Processing (EMNLP-IJCAI)*. **DOI:** https://doi.org/10.3115/v1/P15-1085

Maxim Rabinovich, Mitchell Stern, and Dan Klein. 2017. Abstract syntax networks for code generation and semantic parsing. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics (ACL)*. **DOI:** https://doi.org/10.18653/v1/P17-1105

Aarne Ranta. 1998. A multilingual natural-language interface to regular expressions. In *Finite State Methods in Natural Language Processing*. **DOI:** https://doi.org/10.3115/1611533.1611541

Richard Shin, Miltiadis Allamanis, Marc Brockschmidt, and Oleksandr Polozov. 2019. Program synthesis and semantic parsing with learned code idioms. In *Advances in Neural Information Processing Systems (NeurIPS)*.

Armando Solar-Lezama. 2008. Program Synthesis by Sketching. Ph.D. thesis, University of California at Berkeley.

Alane Suhr, Srinivasan Iyer, and Yoav Artzi. 2018. Learning to map context-dependent sentences to executable formal queries. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT)*. **DOI:** https://doi.org/10.18653/v1/N18-1203

Bailin Wang, Ivan Titov, and Mirella Lapata. 2019. Learning semantic parsers from denotations with latent structured alignments and abstract programs. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing and the International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*. **DOI:** https://doi.org/10.18653/v1/D19-1391, **PMID:** 31933765

Xinyu Wang, Sumit Gulwani, and Rishabh Singh. 2016. FIDEX: Filtering Spreadsheet Data Using Examples. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. **DOI:** https://doi.org/10.1145/2983990.2984030

Navid Yaghmazadeh, Yuepeng Wang, Isil Dillig, and Thomas Dillig. 2017. Sqlizer: Query synthesis from natural language. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):1–26. **DOI:** https://doi.org/10.3133887

Pengcheng Yin and Graham Neubig. 2017. A syntactic neural model for general-purpose code generation. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics (ACL)*.

Luke Zettlemoyer and Michael Collins. 2009. Learning context-dependent mappings from sentences to logical form. In *Proceedings of the Joint Conference of the Annual Meeting of the Association for Computational Linguistics ACL*. **DOI:** https://doi.org/10.3115/1690219.1690283

Ruiqi Zhong, Mitchell Stern, and Dan Klein. 2020. Semantic scaffolds for pseudocode-to-code generation. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics (ACL)*. **DOI:** https://doi.org/10.18653/v1/2020.acl-main.208

Zexuan Zhong, Jiaqi Guo, Wei Yang, Jian Peng, Tao Xie, Jian-Guang Lou, Ting Liu, and Dongmei Zhang. 2018a. SemRegex: A

semantics-based approach for generating regular expressions from natural language specifications. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*. **DOI:** https://doi.org/10.18653/v1/D18-1189

Zexuan Zhong, Jiaqi Guo, Wei Yang, Tao Xie, Jian-Guang Lou, Ting Liu, and Dongmei Zhang. 2018b. Generating regular expressions from natural language specifications: Are we there yet? In *Workshops at the AAAI Conference on Artificial Intelligence (AAAI)*.