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
{xiye,qchen,xwang,isil,gdurrett}@cs.utexas.edu

Abstract

Recent systems for converting natural language descriptions into regular expressions have achieved some success, but typically deal with short, formulaic text and can only produce simple regular expressions, limiting their applicability. Real-world regular expressions are complex, hard to describe with brief sentences, and sometimes require examples to fully convey the user's intent. We present a framework for regular expression synthesis in this setting where both natural language 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 regular expression containing holes to denote missing components. Then a program synthesizer enumerates the regular expression space defined by the sketch and finds a regular expression that is consistent with the given string examples. Our semantic parser can be trained from supervised or heuristically-derived sketches and additionally fine-tuned with weak supervision based on correctness of the synthesized regex. We conduct experiments on two public large-scale datasets (Kushman and Barzilay, 2013; Locascio et al., 2016) and a real-world dataset we collected from StackOverflow. Our system achieves state-of-the-art performance on the public datasets and successfully solves 57% of the real-world dataset, which existing neural systems completely fail on.

1 Introduction

Regular expressions are widely used in various domains, but are notoriously difficult to write: regex is one of the most popular tags of posts on StackOverflow, with over 200,000 posts. Recent research has attempted to build semantic parsers that can translate natural language descriptions into regular expressions, via rule-based techniques (Ranta, 1998), semantic parsing (Kushman and Barzilay, 2013), or sequence-to-sequence neural network models (Locascio et al., 2016; Zhong et al., 2018a). However, while this prior work has achieved relatively high accuracy on benchmark datasets, trained models are still not able to generate expected regular expressions in real-world applications. In these benchmarks, the natural language inputs are primarily short sentences with limited vocabulary, often following simple patterns, and only describe relatively simple regexes.

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, language descriptions can sometimes be underspecified or ambiguous, which leads the models to predict incorrect regexes. One way to supplement such descriptions is by including positive and negative examples of strings for the target regex to match. In fact, providing examples alongside language descriptions is the typical way that questioners post regex-related questions on StackOverflow. Previous methods only take into account language descriptions but cannot leverage the guidance of examples.

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. Second, we use an off-the-shelf program synthesizer, mildly customized for our task, to produce a regex consistent with both the sketch and

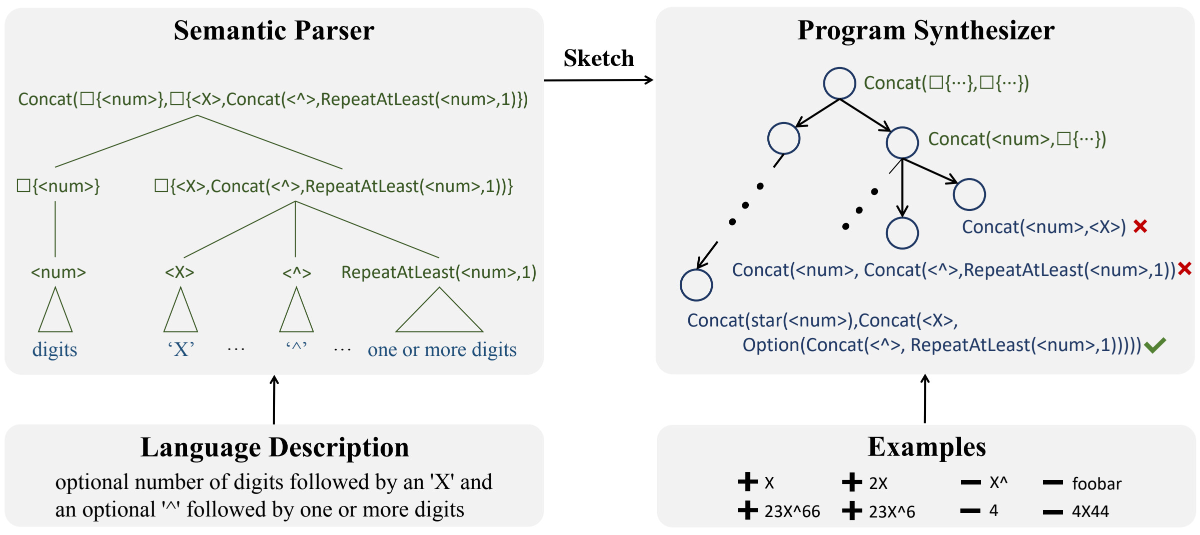

Semantic Parser
Sketch
Program Synthesizer
Concat(□{<num>},□{<X>,Concat(<^>,RepeatAtLeast(<num>,1)})
□{<num>}
□{<X>,Concat(<^>,RepeatAtLeast(<num>,1))}
<num>
<X>
<^>
RepeatAtLeast(<num>,1)
digits
'X'
'^'
one or more digits
Concat(□{···},□{···})
Concat(<num>,□{···})
Concat(<num>,<X>) ✗
Concat(<num>, Concat(<^>,RepeatAtLeast(<num>,1)) ✗
Concat(star(<num>),Concat(<X>, Option(Concat(<^>, RepeatAtLeast(<num>,1))))) ✓
Language Description
optional number of digits followed by an 'X' and an optional '^' followed by one or more digits
Examples
+ X
+ 2X
− X^
− foobar
+ 23X^66
+ 23X^6
− 4
− 4X44

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.

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 datasets. Because these datasets vary in scale, we consider two sketch generation approaches: neural network-based with a sequence-to-sequence model (Luong et al., 2015) or grammar-based with a conventional semantic parsing framework (Berant et al., 2013). We use two large-scale datasets explored in past work, a dataset from Kushman and Barzilay (2013) and the NL-Turk dataset of Locascio et al. (2016), augmented with automatically-produced positive and negative examples for each regex. Our neural parsing model is able to exploit the large amounts of data available here, allowing our sketch-driven approach to outperform existing sequence-to-sequence methods, even when those methods are modified to take advantage of examples. However, we note that the regexes in NL-Turk especially are both short and highly artificial, resulting in very regular natural language that corresponds closely to the regex.

To test our model in a more realistic setting, we collect a dataset of real-world regex synthesis problems from StackOverflow. These problems organically have natural language descriptions and paired examples, since this was the most natural way for the user to communicate their intent. For this dataset, we instantiate our sketch framework with a grammar-based semantic parser due to the limited data. On this dataset, our approach can solve 57% of the benchmarks whereas the performance of existing deep learning approaches is below 10%; the number of samples is insufficient for training neural models and pretrained models do not generalize well to this domain. We believe that this more realistic 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 StackOverflow.[1] In the StackOverflow post, the user describes the desired regex as "*optional number of digits followed by an 'X' and an optional '^' followed by one or more digits*". Additionally, the user provides in total eleven positive and negative examples to further specify their intent. In this example, the natural language description is under-specified and ambiguous: "*optional number of digits*" doesn't clearly specify whether having no digits at all is allowed, and "*and*" in the sentence means concatenation rather than logical and. Therefore, we cannot directly generate the target regex based only on natural language de-

[1]https://stackoverflow.com/questions/10589948/regular-expression-validation-fails-while-egrep-validates-just-fine

scription in this case.

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). In general, the purpose of the sketch is to capture useful components from the natural language 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 is (likely) obtained by composing `<num>`. We later feed this sketch, together with positive/negative examples, into the synthesizer, which enumeratively instantiates holes with DSL constructs until a consistent regex is found.

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

Regex Language Our regex language is shown in Appendix A. The syntax is similar to the one presented in (Locascio et al., 2016) however it is more expressive. In fact, our DSL has the same expressiveness as a standard regular language.

Sketch Language Our sketch language builds on top of our regex DSL by adding a new construct called a "constrained hole," which we denote by $\square_d\{S_1, \ldots, S_m\}$. Here, d is an integer pre-specified by the user and each S_i is a sketch.[2] A program r belongs to the space of programs defined by a constrained hole if *at least* one of the leaf nodes of r belongs to the language of one of the S_i's and r has depth at most d. Note that the program only needs to match *a single* leaf node for *a single* sketch S_i. As we can see, the sketch serves as a hint for some of the leaf nodes of the regex. 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 `RepeatAtLeast(<num>,1)`), whereas the final correct regex happens to include every mentioned component.

3 Semantic Parser

Given a natural language description $L = l_1, l_2, \ldots, l_m$, the task of our semantic parser is to generate a sketch S in our sketch language that

[2]Since d is fixed, we sometimes omit it for simplicity.

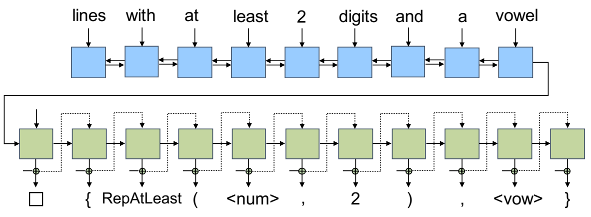


Figure 2: Our neural parser model with a BiLSTM encoder and an LSTM decoder.

encapsulates the user's intent. When put through the synthesizer with examples, 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 seq2seq neural network parser for KB13 and NL-TURK, and grammar-based parser for STACKOVERFLOW, as well as two ways of training the parser, maximum likelihood estimation (MLE, based on a "gold" sketch) and maximum marginal likelihood (MML, based only on whether the sketch leads to the correct synthesis result).

3.1 Neural Parser

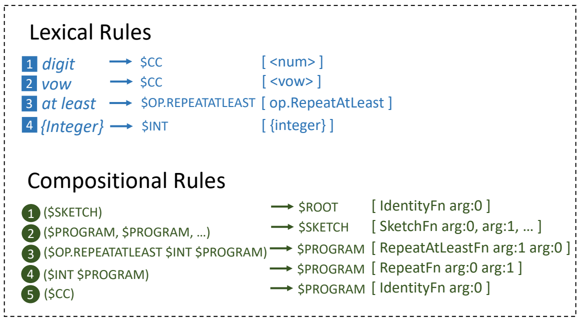
Following recent research (Locascio et al., 2016), we employ a sequence-to-sequence model with attention (Luong et al., 2015) as our neural parser (Figure 2). Here, we treat the sketch as a sequence of tokens $S = s_1, s_2, \ldots, s_n$ and model $P(S|L)$ autoregressively.

We use a single-layer bidirectional LSTM in the encoder and a single-layer unidirectional LSTM for the decoder. While encoding, each token in natural language description is encoded into hidden state by the BiLSTM:

$$\bar{h}_1, \bar{h}_2, ..., \bar{h}_m = \text{BiLSTM}(\bar{w}_1, \bar{w}_2, ..., \bar{w}_m),$$

where $\bar{w}_i$ is the embedded word vector of l_i.

When decoding, we initialize the hidden state of the decoder LSTM with the final encoding of the entire description L. At each timestamp t, we concatenate the decoder hidden state $\hat{h}_t$ with an context vector c_t computed based on bilinear attention, and the probability distribution of the out-



Lexical Rules
1 digit → $CC [<num>]
2 vow → $CC [<vow>]
3 at least → $OP.REPEATATLEAST [op.RepeatAtLeast]
4 {Integer} → $INT [{integer}]
Compositional Rules
1 ($SKETCH) → $ROOT [IdentityFn arg:0]
2 ($PROGRAM, $PROGRAM, …) → $SKETCH [SketchFn arg:0, arg:1, …]
3 ($OP.REPEATATLEAST $INT $PROGRAM) → $PROGRAM [RepeatAtLeastFn arg:1 arg:0]
4 ($INT $PROGRAM) → $PROGRAM [RepeatFn arg:0 arg:1]
5 ($CC) → $PROGRAM [IdentityFn arg:0]
Parse Tree
$ROOT
1
$SKETCH: □{RepeatAtLeast (<num>,2),<vow>}
2
$PROGRAM: RepeatAtLeast (<num>,2)
$PROGRAM: <vow>
3
5
$OP.REPEATATLEAST
$INT:2
$PROGRAM: <num>
3
4
5
$CC: <num>
$CC:<vow>
1
2
lines with
at least
2
digits
and a
vow

Figure 3: Examples of rules and the parse tree for building one possible derivation. The LHS of a rule is the source sequence of tokens or syntactic categories (marked with a "$" sign). The RHS 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.

put token s_t is given as:

$$\begin{aligned}
\hat{h}_1, \hat{h}_2, ..., \hat{h}_t &= \mathrm{LSTM}(\hat{w}_1, \hat{w}_2, ..., \hat{w}_t) \\
a_{i,t} &= \mathrm{softmax}(\bar{h}_i^\top W_q \hat{h}_t) \\
c_t &= \sum_i a_{i,t} \bar{h}_i \\
p(s_t | L, s_{<t}) &= \mathrm{softmax}(W_z[\hat{h}_t; c_t]),
\end{aligned}$$

where $\hat{w}_i$ is the embedded word vector of z_i. Detailed hyperparameters of the model are given in the appendix.

3.2 Grammar-Based Parser

We also explore a grammar-based semantic parser on top of SEMPRE, which has proven effective for semantic parsing tasks in various domains. This approach is less data hungry than deep neural networks and promises better generalizability, as it is less likely to fit annotation artifacts of crowdsourced datasets (Gururangan et al., 2018).

Given a natural language description, our semantic parser relies on 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: $\alpha_1 ... \alpha_n \rightarrow c[\beta]$. Such a rule maps the sequence $\alpha_1 ... \alpha_n$ into target derivation β with syntactic category c.

As shown in Figure 3, a 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 derivations to build more complex ones. Having the predefined rules, our semantic parser constructs the possible derivations by recursively applying the rules. More specifically, we first apply lexical rules to generate leaf derivations for any matching span. Later, the derivations of larger spans are constructed by applying compositional rules to derivations built over non-overlapping constituent spans. Finally, we take the the derivations over the entire natural language description with a designated $ROOT category as the final output.

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, since 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 general and covers the narrow regex domain.

Our parser allows skipping arbitrary tokens (Figure 3) and usually over-generates many derivations. To pick up the best ones, we define a log-linear model to determine the distributions over derivations $Z \in \mathcal{D}(L)$ given natural language 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

[3]The grammar is included in Appendix C.

derivation and description. The features used in our semantic parser mainly characterize the relation between description and applied composition, including

- the co-occurrences of a token l and a rule r fired in the same span: $\mathbb{1}\{r \text{ is fired in the span containing } l\}$,
- the indicator whether a particular rule r is fired: $\mathbb{1}\{r \text{ is fired when deriving } Z\}$,
- and the tree-hierarchy of fired rules: $\mathbb{1}\{r \text{ is fired in } Z \text{ and } r' \text{ is fired in its child } Z'\}$.

3.3 Training

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

MLE MLE maximizes the likelihood of mapping the natural language description to a corresponding gold sketch:

$$\theta^* = \arg\max_\theta \sum \log(p_\theta(S|L)).$$

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 leverage MML to maximize the 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:

$$\theta^* = \arg\max_\theta \sum \log \sum_Z p(r|Z,R)p_\theta(Z|L).$$

It is usually intractable to compute the MML objective because it involves traversing over every possible sketches. Therefore, we only sample sketches from beam search to approximate the gradient (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 does a guided exploration over the space of programs defined by the sketch, while additionally being guided by the examples. In what follows, we describe the general enumeration-based synthesis framework we operate in, then describe the enumeration order of the synthesizer.

Enumerative Synthesis from Sketches We implement a enumeration-based program synthesizer that is a generalized version of the regex synthesizer proposed by Lee et al. (2016). At a high level, the inputs to the synthesizer are a program sketch and a set of positive and negative examples. Using these, the synthesizer searches in the space of programs that could be instantiated by the given sketch and returns a concrete regular expression that accept all positive examples and reject 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>`, `<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 and then we will introduce new holes for its arguments, and we require the components for at least one of the holes to be the original holes' components. For example, in Figure 1, we showed some 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 therefore 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 is different from the work by Lee et al. (2016) in two ways. First, their regex language is extremely restricted (only allowing a binary alphabet) whereas our regex language is much more expressive and allows all common ASCII characters. As a result, our synthesizer can generate both a richer and more human-readable class of regexes. 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 faster.

Dataset	KB13	Turk	SO
size	824	10,000	62
#. unique words	207	557	301
Avg. NL length	8.1	11.5	25.4
Avg. regex length	14.3	18.4	37.2
Avg. regex depth	2.5	2.3	4.0

Table 1: Statistics of three datasets. Compared to KB13 and NL-Turk, StackOverflow contains more sophisticated descriptions and regexes.

KB13
lines where there are two consecutive capital letters
NL-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. NL-Turk tends to be very formulaic, while StackOverflow is longer and much more complex.

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 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.

5 Datasets

We evaluate our framework on two public datasets, KB13 and NL-Turk, and a new dataset, StackOverflow. Statistics about these datasets are given in Table 1, and we describe them in more detail below. As our framework requires string examples which are absent in the public 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 natural 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 natrual language description and regex are generated.

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

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. Since 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 kind of 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 address these issues in past data, we explore regex generation in real-word settings, namely posts on StackOverflow, one of the most popular programming forums. We search posts tagged as regex on StackOverflow and then filter the collected posts with several rules, including:

- the post should include both an natural language description as well as positive and negative examples

- the post should not contain *high-level concepts* (e.g., "months", "US phone numbers") or *visual formatting* (e.g., "AB-XX-XX") in description.

We first collect 105 posts[4] that contain both natural language description and regex, and then select 62 of them using our rules. In addition, we manually clean the description by fixing typos and marking string constants with quotation symbols.

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 length 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. Although we are not able to conduct comprehensive large-scale experiments on STACKOVERFLOW because of its limited size, STACKOVERFLOW can still serve as a pioneer dataset for an initial exploration of real-world regex generation tasks and help provide solid evidence that our sketch-driven approach is effective at tackling real-world problems.

5.2 Dataset Preprocessing

Generating Gold Sketches A "gold" sketch is not uniquely defined given a regex, but having a canonical one is useful for training with maximum likelihood estimation (as discussed in Section 3.3). We generate gold sketches from ground truth regexes in a heuristic fashion as follows. For any regex whose AST has depth more than 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 $\square$\{`<num>`,`<let>`\}. For regex with depth 1, we just wrapped the ground truth regex within a constrained hole. For example, the gold sketch for the regex `<num>` is $\square$\{`<num>`\}. We apply this method to NL-TURK and KB13. For the smaller STACKOVERFLOW dataset, we manually label gold sketches based on information from the gold sketch that should be "indisputable" (obviously true about the ground truth regex).

Generating Positive/Negative Examples The STACKOVERFLOW dataset organically has positive and negative examples, but, for the other datasets, we need to generate examples to augment the existing datasets. We use the automaton (Møller, 2017) library 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 make the examples generated as diverse as possible, we first randomize the path when traversing the automaton. Second, at each transition, we randomly choose a character that belongs to the label of the transition. Finally, we limit the number of times that we visit each transition so that the example generator avoids taking the same transition in a loop. 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 small set of "corner cases" that humans would write.

[4]Despite the fact that more data is available on the website, we only view the top posts because the process requires heavy human involvement.

6 Experiments

6.1 Setup

We implement all neural models in PYTORCH (Paszke et al., 2017). While training, we use the Adam optimizer (Kingma and Ba, 2014) with a learning rate of 0.0001, and a batch size of 25. We train our models until the loss on the development set converges. For models trained with the MML objective, we first pre-train with the MLE objective for efficiency, then use a sample size of 10 to estimate the gradient. In addition, we do beam search a with beam size of 20 when evaluating.

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 parsers with the training set and use heuristic sketches as supervision. On STACKOVERFLOW, we use manually written sketches to train the parser and employ 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 a beam size of 200.

6.2 Evaluation: KB13 and NL-TURK

Baselines We compare two variants of our model (grammar-based or deep learning-based

Approach	KB13		NL-Turk	
	Acc	Consistent	Acc	Consistent
DeepRegex (Locascio et al.)	65.6%	-	58 .2%	-
DeepRegex (Ours)	66.5%	-	60.2%	-
SemRegex	78.2%	-	62.3%	-
DeepRegex + Exs	77.7%	91.0%	83.8%	93.0%
GrammarSketch + MLE	68.9%	95.8%	69.6%	97.2%
DeepSketch + MLE	84.0%	95.3%	**85.2%**	98.4%
DeepSketch + MML	**86.4%**	96.3%	84.8%	97.8%

Table 2: Comparison 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 neural model (DeepSketch) achieves the best results on both datasets, but even our grammar-based method (GrammarSketch) achieves reasonable results, outperforming past systems that do not use examples.

sketches) against three baselines. DeepRegex directly translates language descriptions with a seq-to-seq model without looking at the examples. Note that we compare against both reported numbers as well as our own implementation of this (DeepRegex (Ours))[5], since we could not reproduce those results using the released code. Our reimplemented DeepRegex outperforms the original one by 0.9% and 2.0% on KB13 and NL-Turk, respectively; we use this version in all other reported experiments.

SemRegex (Zhong et al., 2018a)[6] 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 with the annotation.

While Zhong et al. (2018a) use examples to check semantic correctness at *training* time, neither this method nor DeepRegex uses examples at *test* time. To compare these methods to our setting, we extend our version of DeepRegex 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 call this DeepRegex+Exs.

Results Table 2 summarizes our experimental results on these two prior datasets. We find a significant performance boost (11.2% on KB13 and 21.5% on NL-Turk) by filtering the output beams using examples. Such performance gain demonstrates one advantage we can take from examples, i.e., examples help verify the produced regexes. Furthermore, our sketch-driven approach outperforms previous approaches even if they are extended to benefit from examples. We achieve new state-of-the-art results on both datasets by an accuracy increase of 8.7% and 1.4% on KB13 and NL-Turk, respectively. This performance increase demonstrates that our decomposition of the problem is effective. In addition, we compare the fraction of consistent regexes. Because we allow uncertainty in the sketches and use examples to guide the construction of regexes, our framework, using either neural parser or grammar-based parser, produces over 4% more regexes consistent with examples compared to DeepRegex + Exs baseline.

We also find that our GrammarSketch approach achieves near 70% accuracy on both datasets, which is better than DeepRegex. The performance of GrammarSketch lags that of DeepRegex + Exs 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. However, we still see that, even with a simple grammar (see Appendix C), GrammarSketch can achieve strong enough results to demonstrate the generalizability of this approach across different datasets.

6.3 Evaluation: StackOverflow

Baselines Given the small data size, it is impractical to train a deep neural model from scratch to translate natural language descriptions into either

[5]Detailed model architecture is included in Appendix B.

[6]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.

Approach	Top-N Acc		
	top-1	top-5	top-25
DEEPREGEX + EXS (transferred model)	0%	0%	0%
DEEPREGEX + EXS (curated language)	0%	0%	6.6%
DEEPSKETCH (transferred model)	3.2%	3.2%	4.8%
DEEPSKETCH (curated language)	9.7%	11.3%	11.3%
GRAMMARSKETCH LONGEST DERIV	16.1%	34.4%	45.2%
GRAMMARSKETCH + MLE	**34.4%**	48.4%	53.2%
GRAMMARSKETCH + MML	31.1%	**54.1%**	**56.5%**

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. DEEPSKETCH is slightly better, but still not effective when learned on such small data. Our GRAMMARSKETCH model can do significantly better, and benefits from better training techniques.

regexes or sketches. We derive several baselines from the exiting literature so as to make comparison. First, we train a transferred model that first pre-trains on NL-TURK and fine-tunes on STACKOVERFLOW. Second, we manually rewrite the descriptions in STACKOVERFLOW to make them conform to the style of NL-TURK, 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* in into *line that does not contain "space" two or more times*, and apply DEEPREGEX + EXS or DEEPSKETCH method on the curated descriptions (without fine-tuning). Note that crafting the input to the expected style is highly generous to the models trained on NL-TURK.

In addition, we test our grammar-based parser with a straightforward re-ranking rule, i.e., picking the derivation that is built with the largest number rules applied (GRAMMARSKETCH LONGEST DERIV). This is to show the effect of learning on the grammar-based model.

Results Table 3 shows the results on this dataset. Since STACKOVERFLOW is a challenging dataset, we report the top-N semantic accuracy, on which the model is considered correct if any of the top-N derivations (regexes or sketches) yields a correct answer.

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. However, the examples solved are very simple, such as the one mentioned above, and the baseline model is extremely sensitive to the text. Our DEEPSKETCH approach is still limited here, only solving 3.2% of the examples, or around 10% if we curate the language descriptions. Our GRAMMARSKETCH approach achieves the best results, 34.4% top-1 accuracy using MLE and 57.4% top-25 accuracy using the MML objective, while the best baseline only reaches 0% and 6.6%, respectively. Because the sketch 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., Figure 1).

Furthermore, we observed that our model is not simply relying on the grammar, but it is important to learn good parameter values as well (compare the heuristic GRAMMARSKETCH LONGEST DERIV method with the others). While training with the MML objective performs better on top-25, it does not perform better on top-1. We theorize that this is because the pattern of which sketches successfully synthesize is sometimes complex, and the model may fail to learn a consistent ranking as these correct sketches might not follow a learnable pattern.

6.4 Evaluation: Data efficiency

As shown on the existing datasets, one advantage of our GRAMMARSKETCH approach seems to be that it doesn't rely on large training sets. In Figure 5, we evaluate this explicitly, comparing the

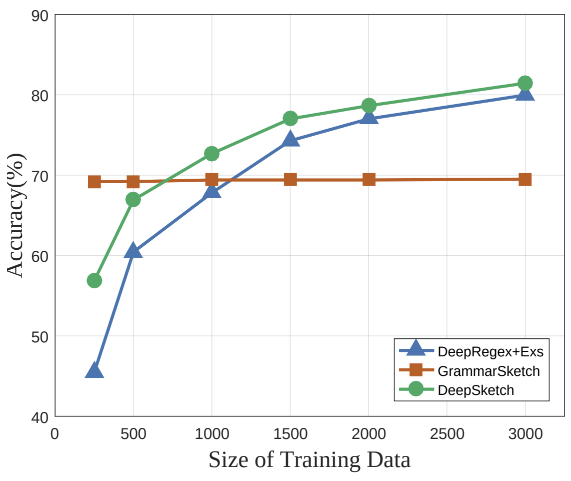


Figure 5: Accuracy on NL-TURK when data is limited. Our DEEPSKETCH and GRAMMARSKETCH approaches outperform DEEPREGEX + EXS baseline when training data is limited.

performance of our DEEPSKETCH, GRAMMARSKETCH, and DEEPREGEX + EXS baseline with respect to different size of training data of NL-TURK. When the data size is extremely limited (i.e. no more than 500), our GRAMMARSKETCH approach is much better than the DEEPREGEX + Exs baseline. Our DEEPSKETCH approach is more flexible and achieves stronger performance for larger training data sizes, consistently outperforming the DEEPREGEX + EXSs 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. Our comparison results on KB13 and NL-TURK also provides evidence that the sketch-driven approach achieves larger improvement on the smaller KB13 dataset.

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.

6.5 Analysis: Success And Failure Pairs

NL-TURK We now analyze the output of the DEEPREGEX + EXS and DEEPSKETCH. Figure 6 provides some success pairs that DEEPSKETCH solves while DEEPREGEX + EXS does not. Examples of success pairs suggest that our approach can deal with under-specified natural language 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 since 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 string *truck* (because ~`(truck)` matches any string but *truck*). The semantics are hard to pin down with examples, but the correct regex is also quite artificial and unlikely to appear in real word applications. Our DEEPSKETCH fails on this pair since the synthesizer fails to catch the *at least 7 times* constraint when strings that have less 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 Finally, 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). These examples are similar in that they both contain multiple sentences with each one describing certain a component or constraint. This seems to be a common pattern of describing real world regexes, and 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 a must, 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 gram-

NL-TURK		
Success Pairs:		
(a)	description:	*lines with 3 more capital letters*
	ground truth:	`(<cap>){3,})(.*)`
(b)	description:	*none of the lines should have a vowel , a capital letter , or the string "dog"*
	ground truth:	`~((<vow>)|(dog)|(<cap>))`
Failure Pairs:		
(c)	description:	*lines with "dog" or without "truck" , at least 7 times*
	ground truth:	`((dog)|(~(truck))){7,}`
	our output:	`(dog)|(~(truck))`
(d)	description:	*lines ending with lower-case letter or not the string "dog"*
	ground truth:	`(.*)(([<low>])|(~(dog)))`
	our output:	`(([<low>])|(~(dog)))*`
STACKOVERFLOW		
Success Pairs:		
(e)	description:	*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.*
	ground truth:	`((<let>|<num>|(.)|(%d4%)|(%t7%)){1,}`
(f)	description:	*The input box should accept only if either (1) first 2 letters alpha + 6 numeric or (2) 8 numeric*
	ground truth:	`(<let>{2}<num>{6})|(<num>{8})`
Failure Pairs:		
(g)	description:	*I'm trying to devise a regular expression which will accept decimal number up to 4 digits*
	ground truth:	`(<num>{1,})(.)(<num>{1,4})`
	wrong output:	`(<num>{1,})((.)(<num>{1,4}))?`
(h)	description:	*the first letter of each string is in upper case*
	ground truth:	`<cap>(<let>)*(( )<cap>(<let>)*)*`
	wrong output:	`((( )<cap>)|(<let>)){1,}`

Figure 6: Examples of success and failure pairs from NL-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.

mar 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 works lie in using semantic parsing to synthesize programs. Particularly, several techniques have been proposed to translate natural language to SQL queries (Yaghmazadeh et al., 2017), "if-this-then-that" recipes (Quirk et al., 2015), bash commands (Lin et al., 2018), Java expressions (Gvero and Kuncak, 2015) and etc. This work is different from prior work in that it utilizes both the natural language as well as input-output examples.

Another line of work uses deep learning to directly predict programs from natural language. Recent work has built encoder decoder models to generated logic forms represented by sequences (Dong and Lapata, 2016), and ASTs (Rabinovich et al., 2017; Yin and Neubig, 2017). Our work differs from theirs in that we truly execute the generated programs on real data, while previous work only evaluates the produced strings by exact match accuracy or BLEU score.

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, we implement an enumeration-based synthesizer that incorporates domain heuristics that speed up its performance. Our method also shares similarity with sketching-based approaches (Solar-Lezama, 2008) in that our synthesizer starts with a sketch. However, our sketches are produced automatically from the natural language description whereas traditional sketch-based synthesis relies on a user-provided sketch.

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.

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*, pages 1–8.

Jonathan Berant, Andrew Chou, Roy Frostig, and Percy Liang. 2013. Semantic parsing on Freebase from question-answer pairs. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*, pages 1533–1544, Seattle, Washington, USA. Association for Computational Linguistics.

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, Berlin, Germany. Association for Computational Linguistics.

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 2018, pages 420–435, New York, NY, USA. ACM.

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 '11, pages 317–330, New York, NY, USA. ACM.

Suchin Gururangan, Swabha Swayamdipta, Omer Levy, Roy Schwartz, Samuel Bowman, and Noah A. Smith. 2018. Annotation artifacts in natural language inference data. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 2 (Short Papers)*, pages 107–112, New Orleans, Louisiana. Association for Computational Linguistics.

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 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1051–1062, Vancouver, Canada. Association for Computational Linguistics.

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 2015, pages 416–432, New York, NY, USA. ACM.

Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

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*, pages 826–836, Atlanta, Georgia. Association for Computational Linguistics.

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 2016, pages 70–80, New York, NY, USA. ACM.

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 Eleventh International Conference on Language Resources and*

Evaluation LREC 2018, Miyazaki (Japan), 7-12 May, 2018.

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 2016 Conference on Empirical Methods in Natural Language Processing*, pages 1918–1923, Austin, Texas. Association for Computational Linguistics.

Thang Luong, Hieu Pham, and Christopher D. Manning. 2015. Effective approaches to attention-based neural machine translation. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 1412–1421, Lisbon, Portugal. Association for Computational Linguistics.

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

Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in PyTorch. In *NIPS Autodiff 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 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 878–888, Beijing, China. Association for Computational Linguistics.

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.

Aarne Ranta. 1998. A multilingual natural-language interface to regular expressions. In *Finite State Methods in Natural Language Processing*.

Armando Solar-Lezama. 2008. *Program Synthesis by Sketching*. Ph.D. thesis, University of California at Berkeley, Berkeley, CA, USA. AAI3353225.

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 2016, pages 195–213, New York, NY, USA. ACM.

Navid Yaghmazadeh, Yuepeng Wang, Isil Dillig, and Thomas Dillig. 2017. SQLizer: Query Synthesis from Natural Language. *Proc. ACM Program. Lang.*, 1(OOPSLA):63:1–63:26.

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.

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 2018 Conference on Empirical Methods in Natural Language Processing*, pages 1608–1618, Brussels, Belgium. Association for Computational Linguistics.

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?

Appendix A: Regex Language

Non-terminals r		
StartsWith(r) $\rightarrow r.*$	EndsWith(r) $\rightarrow .*r$	Contains(r) $\rightarrow .*r.*$
Not(r) $\rightarrow \sim r$	Optional(r) $\rightarrow r?$	KleeneStar(r) $\rightarrow r*$
Concat(r_1, r_2) $\rightarrow r_1 r_2$	And(r_1, r_2) $\rightarrow r_1 \& r_2$	Or(r_1, r_2) $\rightarrow r_1 \mid r_2$
Repeat(r, k) $\rightarrow r\{k\}$	RepeatAtLeast(r, k) $\rightarrow r\{k,\}$	RepeatRange(r, k_1, k_2) $\rightarrow r\{k_1, k_2\}$

Terminals			
<let> $\rightarrow$ [A-Za-z]	<vow> $\rightarrow$ [AEIOUaeiou]	<cap> $\rightarrow$ [A-Z]	<low> $\rightarrow$ [a-z]
<num> $\rightarrow$ [0-9]	<alphanum> $\rightarrow$ [A-Za-z0-9]	<any> $\rightarrow$.	
<eps> $\rightarrow \epsilon$		<null> $\rightarrow \emptyset$	

Figure 7: Our regex language (k is a positive integer). The right arrow $\rightarrow$ shows how to translate our language to a standard regular expression language.

Appendix B: Hyperparameters Of Our Neural Model

	Ours	Locascio et al. (2016)
input dim	128	64
encoder type	BiLSTM	LSTM
encoder layer	1	2
encoder size	256	512
output dim	128	64
decoder type	LSTM	LSTM
decoder layer	1	2
decoder size	256	512

Table 4: Comparison of the neural model used in our approach and the original model used in DEEPREGEX(Locascio et al., 2016). We use a single layer BiLSTM encoder and a single layer LSTM decoder while original DEEPREGEX uses a double-layer LSTM encoder and a double-layer LSTM decoder.

Appendix C: Grammar

8.1 Notes about the parsing rules

Recall from the paper that we specify our grammar using rules of the following form: $\alpha_1 ... \alpha_n \rightarrow c[\beta]$.

The β can be exactly the target derivation, or a semantic function (such as IdentityFn, SelectFn, sketch.UnarySketchFn etc.) that is applied to produce target derivation with arguments.

There are two types of arguments:

- arg:i represents selecting the i^{th} position from the matched $\alpha_1 ... \alpha_n$ and passing it to the function. For example, the rule ($Skip $SKETCH $\rightarrow$ $ROOT [SelectFn arg:1]) selects the first category from the source sequence, which is the $SKETCH (Notes that Sempre starts indexing from 0).
- val:n represents passing the integer value n to the function. For example, the rule ($CC1 $MARKER_ONLY $\rightarrow$ $PROGRAM[sketch.RepeatatleastFn arg:0, val:1]) passes the first category in the matched source sequence $MARKER_ONLY and the integer value 1 to the semantic function sketch.RepeatatleastFn.

8.2 Compositional Rules

Root
$Sketch → $ROOT [IdentityFn arg:0]
$Skip $SKETCH → $ROOT [SelectFn arg:1]

Skip tokens rule
$LEMMA_PHRASE → $Skip [ConstantFn arg:null]

Parse a number
$LEMMA_PHRASE → $INT1 [NumberFn val:NUMBER]

Parse a character class or constant to a regex
$CC → $PROGRAM [IdentifyFn arg:0]
$CONST → $PROGRAM [IdentifyFn arg:0]

Hierarchical sketch parse rules
$LIST_PROGRAM → $SKETCH [sketch.UnarySketchFn arg:0]
$PROGRAM → $LIST_PROGRAM [IdentityFn arg:0]
$PROGRAM $LIST_PROGRAM → $LIST_PROGRAM [sketch.SketchJoinFn arg:0]

Operator: NotContain
$MARKER_NOTCONTAIN $SKETCH → $SKETCH [sketch.NotContainFn arg:1]
$MARKER_NOTCONTAIN $PROGRAM → $PROGRAM [sketch.NotContainFn arg:1]

Operator: Not
$MARKER_NOT $SKETCH → $SKETCH [sketch.NotFn arg:1]
$MARKER_NOT $PROGRAM → $PROGRAM [sketch.NotFn arg:1]
$MARKER_NON1 $CONST → $PROGRAM [sketch.NotccFn arg:1]

Operator: Optional
$MARKER_NOT $CC → $PROGRAM [sketch.OptionalFn arg:1]

Operator: StartWith, EndWith
$MARKER_STARTWITH $PROGRAM → $PROGRAM [sketch.StartwithFn arg:1]
$MARKER_ENDWITH $PROGRAM → $PROGRAM [sketch.EndwithFn arg:1]
$PROGRAM $MARKER_ATEND → $PROGRAM [sketch.EndwithFn arg:0]

Operator: Concat
$PROGRAM $MARKER_CONCAT $PROGRAM → $PROGRAM [sketch.ConcatFn arg:0, arg:2]
$PROGRAM $MARKER_CONCAT $SKETCH → $SKETCH [sketch.ConcatFn arg:0, arg:2]
$SKETCH $MARKER_CONCAT $PROGRAM → $SKETCH [sketch.ConcatFn arg:0, arg:2]
$SKETCH $MARKER_CONCAT $SKETCH → $SKETCH [sketch.ConcatFn arg:0, arg:2]
$PROGRAM $MARKER_FOLLOW $PROGRAM → $PROGRAM [sketch.ConcatFn arg:2, arg:0]
$PROGRAM $MARKER_FOLLOW $SKETCH → $SKETCH [sketch.ConcatFn arg:2, arg:0]
$SKETCH $MARKER_FOLLOW $PROGRAM → $SKETCH [sketch.ConcatFn arg:2, arg:0]
$SKETCH $MARKER_FOLLOW $SKETCH → $SKETCH [sketch.ConcatFn arg:2, arg:0]

Operator: Repeat

$INT $CC → $PROGRAM [sketch.RepeatFn arg:1, arg:0]
$CC $MARKER_LENGTH $INT → $PROGRAM [sketch.RepeatFn arg:0, arg:2]
$MARKER_LENGTH $INT $CC → $PROGRAM [sketch.RepeatFn arg:2, arg:1]
$INT1 $MARKER_OR1 $INT1 $CC → $PROGRAM [sketch.RepeatAOrBFn arg:3, arg:0, arg:2]

Operator: RepeatAtLeast
$MARKER_ONLY1 $CC → $PROGRAM [sketch.RepeatatleastFn arg:1, val:1]
$CC1 $MARKER_ONLY → $PROGRAM [sketch.RepeatatleastFn arg:0, val:1]
$INT1 $MARKER_ORMORE1 $CC → $PROGRAM [sketch.RepeatatleastFn arg:2, arg:0]
$PROGRAM $INT1 $MARKER_ORMORE1 → $PROGRAM [sketch.RepeatatleastFn arg:0, arg:1]

Operator: RepeatRange
$MARKER_ATMAX1 $INT $CC → $PROGRAM [sketch.RepeatrangeFn arg:2, val:1, arg:1]
$MARKER_ATMAX $INT $CC → $PROGRAM [sketch.RepeatrangeFn arg:2, val:1, arg:1]
$INT $CC → $PROGRAM [sketch.RepeatrangeFn arg:1, val:1, arg:0]

Extract constants
$CONST_SET → $CC1 [IdentityFn arg:0]
$CONST1 $CONST_SET → $CONST_SET [sketch.ConstUnionFn arg:0, arg:1]
$CONST1 $CONST1 → $CONST_SET [sketch.ConstUnionFn arg:0, arg:1]
$CONST1 $MAKRKER_CONSTSETUNION1 $CONST1 → $CONST_SET [sketch.ConstUnionFn arg:0, arg:2]
$CONST1 $MAKRKER_CONSTSETUNION1 $CONST_SET → $CONST_SET [sketch.ConstUnionFn arg:0, arg:2]
$CCPHRASE1 $MAKRKER_CONSTSETUNION1 $CONST1 → $CONST_SET [sketch.ConstUnionFn arg:0, arg:2]
$CCPHRASE1 $MAKRKER_CONSTSETUNION1 $CCPHRASE1 → $CONST_SET [sketch.ConstUnionFn arg:0, arg:2]
$CCPHRASE1 $MAKRKER_CONSTSETUNION1 $CONST_SET → $CONST_SET [sketch.ConstUnionFn arg:0, arg:2]
“ $PHRASE ” → $CONST1 [sketch.ConstFn arg:0]

“Separated/Split by”
$SKETCH $PROGRAM $MARKER_SEP → $SKETCH [sketch.SepFn arg:0,arg:1]
$PROGRAM $PROGRAM $MARKER_SEP → $PROGRAM [sketch.SepFn arg:0,arg:1]
$PROGRAM $MARKER_BETWEEN $SKETCH → $SKETCH [sketch.SepFn arg:2,arg:0]
$PROGRAM $MARKER_BETWEEN $PROGRAM → $PROGRAM [sketch.SepFn arg:2,arg:0]
$SKETCH $MARKER_SPLITBY $PROGRAM → $SKETCH [sketch.SepFn arg:0,arg:2]

“Decimal”
$PROGRAM $MARKER_DECIMAL $PROGRAM → $SKETCH [sketch.DecimalFn arg:0, arg:2]
$MARKER_DECIMAL $PROGRAM $PROGRAM → $SKETCH [sketch.DecimalFn arg:1, arg:2]
$PROGRAM $PROGRAM $MARKER_DECIMAL → $SKETCH [sketch.DecimalFn arg:0, arg:1]
$MARKER_DECIMALNUM → $SKETCH [sketch.DecimalFn]

Skip Rules: we present one example here as a skip rule that is required by Sempre to allow skipping tokens when matching compositional rules. These rules can be generated automatically and hence we don’t count these as part of the compositional rules.
$Skip optional → $CC [SelectFn arg:0]

Lexicon mapping rules: we present one example here that matches lexicons in the lexicon files to base-case target category to allow compositional rules build up on lexicons. These rules can be generated automatically and hence we don’t count these as part of the compositional rules.
$CCPHRASE1 → $CC1 [IdentityFn arg:0]

8.3 Lexical Rules

number → $CC [<num>]
numeric → $CC [<num>]
numeral → $CC [<num>]
decimal → $CC [<num>]
digit → $CC [<num>]
alphanumeric → $CC [<alphanum>]
hexadecimal → $CC [<hex>]
string → $CC [<any>]
character → $CC [<let>]
letter → $CC [<let>]
word → $CC [<let>]
alphabet → $CC [<let>]

lower case letter → $CC [<low>]
small letter → $CC [<low>]
upper case letter → $CC [<cap>]
capital letter → $CC [<cap>]
vowel → $CC [<vow>]
special character → $CC [<spec>]
special char → $CC [<spec>]
comma → $CONST [<,>]
colon → $CONST [<:>]
semicolon → $CONST [<;>]
space → $CONST [<space>]
underscore → $CONST [<_>]
dash → $CONST [<->]
percentage sign → $CONST [<%>]
percentage sign → $CONST [<%>]
not → $OP.NOT [op.not]
non → $OP.NON [op.non]
or → $OP.OR [op.or]
optional → $OP.OPTIONAL [op.optional]
not contain → $OP.NOTCONTAIN [op.notcontain]
not allow → $OP.NOTCONTAIN [op.notcontain]
or more → $OP.ORMORE [op.ormore]
or more time → $OP.ORMORE [op.ormore]
max → $OP.MAX [op.max]
decimal → $OP.DECIMAL [op.decimal]
double number → $OP.DECIMALNUM [op.decimalnum]
length → $OP.LENGTH [op.length]
, → $OP.CONSTSETUNION [op.constsetunion]
(, optional) or → $OP.CONSTSETUNION [op.constsetunion]
(, optional) and → $OP.CONSTSETUNION [op.constsetunion]
separate → $OP.SEP [op.sep]
delimit → $OP.SEP [op.sep]
between → $OP.BETWEEN [op.between]
separated → $OP.BETWEEN [op.between]
split by → $OP.SPLITBY [op.splitby]
divide by → $OP.SPLITBY [op.splitby]
end with → $OP.ENDWITH [op.endwith]
finish with → $OP.ENDWITH [op.endwith]
end in → $OP.ENDWITH [op.endwith]
terminate → $OP.ENDWITH [op.endwith]
at end → $OP.ATEND [op.atend]
start with → $OP.STARTWITH [op.startwith]
start in → $OP.STARTWITH [op.startwith]
at the begin → $OP.STARTWITH [op.startwith]
before → $OP.CONCAT [op.concat]
follow by → $OP.CONCAT [op.concat]
next → $OP.CONCAT [op.concat]
then → $OP.CONCAT [op.concat]
prior to → $OP.CONCAT [op.concat]
precede → $OP.CONCAT [op.concat]
after → $OP.FOLLOW [op.follow]
bulletpoint → $OP.FOLLOW [op.follow]
up to → $OP.ATMAX [op.atmax]
at max → $OP.ATMAX [op.atmax]
only → $OP.ONLY [op.only]