Benchmarking Multimodal Regex Synthesis with Complex Structures

Xi Ye Qiaochu Chen Isil Dillig Greg Durrett
Department of Computer Science
The University of Texas at Austin
{xiye,qchen,isil,gdurrett}@cs.utexas.edu

Abstract

Existing datasets for regular expression (regex) generation from natural language are limited in complexity; compared to regex tasks that users post on StackOverflow, the regexes in these datasets are simple, and the language used to describe them is not diverse. We introduce StructuredRegex, a new regex synthesis dataset differing from prior ones in three aspects. First, to obtain structurally complex and realistic regexes, we generate the regexes using a probabilistic grammar with pre-defined macros observed from real-world StackOverflow posts. Second, to obtain linguistically diverse natural language descriptions, we show crowdworkers abstract depictions of the underlying regex and ask them to describe the pattern they see, rather than having them paraphrase synthetic language. Third, we augment each regex example with a collection of strings that are and are not matched by the ground truth regex, similar to how real users give examples. Our quantitative and qualitative analysis demonstrates the advantages of StructuredRegex over prior datasets. Further experimental results using various multimodal synthesis techniques highlight the challenge presented by our dataset, including non-local constraints and multi-modal inputs.[1]

1 Introduction

Regular expressions (regexes) are known for their usefulness and wide applicability, and yet they are hard to understand and write, even for many programmers (Friedl, 2006). Recent research has therefore studied how to construct regexes from natural language (NL) descriptions, leading to the emergence of NL-to-regex datasets including

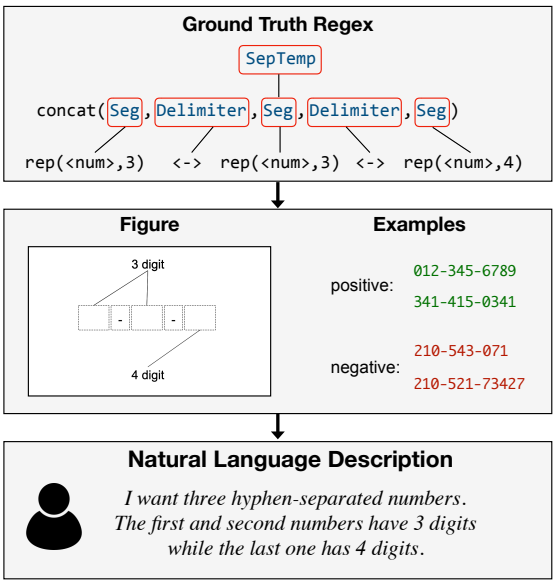


Figure 1: Our dataset collection process. A regex is sampled from our grammar, then we render an abstract figure and generate distinguishing positive/negative examples. We present the figure and examples to crowdworkers to collect natural language descriptions.

KB13 (Kushman and Barzilay, 2013) and NL-Turk (Locascio et al., 2016). However, KB13 is small in size, with only 814 NL-regex pairs with even fewer distinct regexes. Locascio et al. (2016) subsequently employed a generate-and-paraphrase procedure (Wang et al., 2015) to create the larger NL-Turk dataset. However, the regexes in this dataset are very simple, and the descriptions are short, formulaic, and not linguistically diverse because of the paraphrasing annotation procedure (Herzig and Berant, 2019). As a result, even when models achieve credible performance on these datasets, they completely fail when evaluated on the StackOverflow dataset (Ye et al., 2019), a real-world dataset collected from users seeking help on StackOverflow. The limited size of this dataset (only 62 NL-regex pairs) makes it

[1]Code and data available at https://www.cs.utexas.edu/~xiye/streg/.

(a)	*I need to validate the next pattern: starts with "C0" and finish with 4 digits exactly.* `and(startwith(<C0>)),endwith(rep(<num>,4)))`
(b)	*i need regular expression for : one or two digits then "." and one or two digits.* `concat(reprange(<num>,1,2),concat(<.>,reprange(<num>,1,2)))`
(c)	*The input will be in the form a colon (:) separated tuple of three values. The first value will be an integer, with the other two values being either numeric or a string.* `concat(repatleast(<num>,1),rep(concat(<:>,or(repatleast(<let>,1),repatleast(<num>,1))),2))`

Figure 2: Examples of complex regexes from StackOverflow. Each regex can be viewed as a set of components composed with a high-level template. Regex (a), for example, can be as viewed the intersection of two constraints specifying the characteristics of the desired regex. (`rep` means repeat).

unsuitable for large-scale training, and critically, the complexity of regexes it features means that regex synthesis systems must leverage the user-provided positive and negative examples (strings that should be matched or rejected by the target regex) in order to do well.

To enable the development of large-scale neural models in this more realistic regex setting, we present StructuredRegex, a new dataset of English language descriptions and positive/negative examples associated with complex regexes. Using a new data collection procedure (Figure 1), our dataset addresses two major limitations in NL-Turk. First, we generate our regexes using a structured probabilistic grammar which includes *macro* rules defining high-level templates and constructions that involve multiple basic operators. These grammar structures allow us to sample more realistic regexes, with more terminals and operators, while avoiding vacuous regexes. By contrast, the random sampling procedure in NL-Turk leads to simple regexes, and attempting to sample more complex regexes results in atypical regex structures or even contradictory regexes that do not match any string values (Ye et al., 2019). Second, to achieve more realistic language descriptions, we prompt Turkers to write descriptions based on abstract figures that show the desired regexes. We design a set of visual symbols and glyphs to *draw* a given regex with minimal textual hints. We thereby avoid priming Turkers to a particular way of describing things, hence yielding more linguistically diverse descriptions.

Using this methodology, we collect a total of 3,520 English descriptions, paired with ground truth regexes and associated positive/negative examples. We conduct a comprehensive analysis and demonstrate several linguistic features present in our dataset which do not occur in past datasets. We evaluate a set of baselines, including grammar-based methods and neural models, on our dataset. In addition, we propose a novel decoding algorithm that integrates constrained decoding using positive/negative examples during inference: this demonstrates the potential of our dataset to enable work at the intersection of NLP and program synthesis. The performance of the best existing approach on StructuredRegex only reaches 37%, which is far behind 84% on NL-Turk. However, this simple model can nevertheless solve 13% of the StackOverflow dataset, indicating that further progress on this dataset can be useful for real-world scenarios.

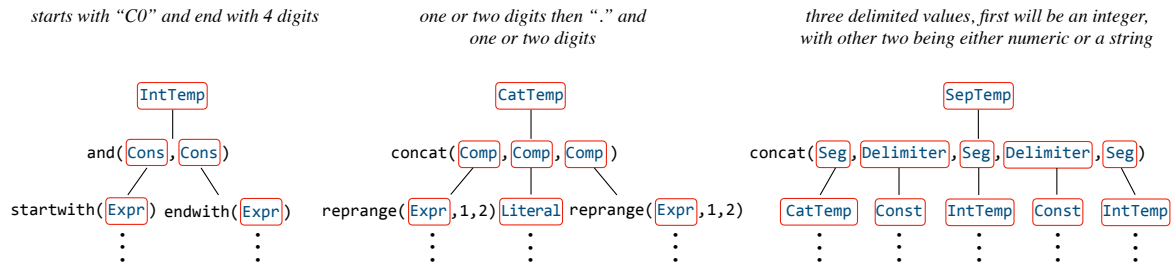
2 Structured Regex Generation Process

We first describe the structured generative process we adopt to produce the regexes in our dataset. For better readability, we denote regexes using a domain specific language (DSL) similar to regex DSLs in prior work (Locascio et al., 2016; Ye et al., 2019). Our DSL has the same expressiveness as a standard regular language and can be easily mapped back to standard regular expressions.[2]

To collect the NL-Turk dataset, Locascio et al. (2016) sampled regexes using a hand-crafted grammar similar to a standard regex DSL. However, regexes sampled from this process can easily have conflicts (e.g. `and(<let>,<num>)`) or redundancies (e.g. `or(<let>,<low>)`). One solution to this problem is rejection sampling, but this still does not yield regexes with compositional, real-world structure.

We show three prominent types of composition observed from StackOverflow in Figure 2. Each regex above is built by assembling several sub-regexes together according to a high-level template: regex (a) is the intersection of two base regexes expressing constraints, regex (b) is a sequence of three simple parts, and regex (c) is a

[2]Refer to the appendix for details of our DSL.



Cons	`start[end]with(Expr) | not(start[end]with(Expr))`	# must (not) start/end with
	`contain(Expr) | not(contain(Expr))`	# must (not) contain
	`rep(<any>,k) | repatleast(<any>,k) |reprange(<any>,k,k)`	# length constraints
	`AdvStartwithCons | AdvEndwithCons`	# adversative macro (e.g., start with capitals except A)
	`CondContainCons`	# conditional macro. (e.g. letter, if contained, must be after a digit)
Comp	`Literal | or(Literal,Literal,...)`	# literals like digits, letters, strings, or set of literals.
	`rep(Expr,k) | repatleast(Expr,k) | reprange(Expr,k,k)`	# e.g, 3 digits, 2 - 5 letter, etc.
	`optional(Comp)`	# components can be optional.

Figure 3: Examples of our top-level templates and how they cover the three regexes in Figure 2, and overview of sub-regexes (in table) that can possibly be derived from `Cons` and `Comp`. `Expr` as a category here indicates various different constrained sets of sub-regexes. More detail about this structure is available in the full grammar in the appendix.

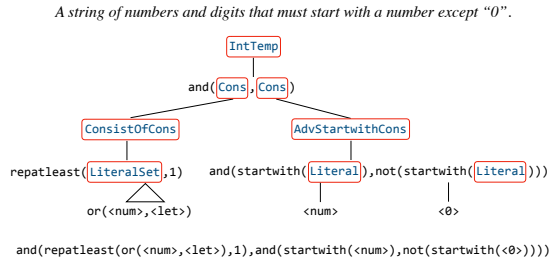


Figure 4: The generation of a deep and complex regex using our grammar. Here, `AdvStartwithCons` is a macro rule that yields a complex sub-tree with an adversative constraint.

list of three segments delimited by a constant. We observe that these three templates actually capture a wide range of possible regex settings. The first, for example, handles password validation-esque settings where we have a series of constraints to apply to a single string. The second and third reflect matching sequences of fields, which may have shared structured (regex (c)) or be more or less independent (regex (b)).

2.1 Structured Grammar

To generate realistic regexes in these forms, we rely on a structured hand-crafted grammar. The top level of our grammar specifies three templates distilled from StackOverflow examples: Intersection, Concatenation, and Separation, which mimic patterns of real-world regexes. In Figure 3, we show how regexes in Figure 2 can be derived from our templates. The Intersection template (left) intersects several base constraints with the `and` operator; the Concatenation template (middle) concatenates several base components with the `concat` operator. Separation (right) is a more complex type, generating a list of constant-separated Intersection or Concatenation regexes which may be identical or share common components.

Across all templates, the components are sub-regexes falling into a few high-level types (notably `Cons` and `Comp`), which are depth-limited to control the overall complexity (discussed in Appendix B.2). To make these component regexes more realistic as well, we design several macro rules that expand to more than one operator. The macros are also extracted from real-world examples and capture complex relations like adversative (Figure 4) and conditional (Table 2) relations.

Although our hand-crafted grammar does not cover every possible construction allowed by the regular expression language, it is still highly expressive. Based on manual analysis, our grammar covers 80% of the real-world regexes in StackOverflow, whereas the grammar of NL-Turk only covers 24% (see Section 4). Note that some constructions apparently omitted by our grammar are equivalent to ones supported by our grammar: e.g., we don't allow a global `startwith` constraint in the Concatenation template, but this constraint can be expressed by having the first component of the concatenation incorporate the desired

constraint.

2.2 Sampling from the Regex Grammar

Although our structural constraints on the grammar already give rise to more realistic regexes, we still want to impose further control over the generative process to mimic properties of real-world regexes. For example, there are sometimes repeating components in CONCATENATION regexes, such as regex (b) from Figure 2.

We encourage such regexes by dynamically modifying the probability of applying the grammar rules while we are expanding a regex based on the status of the entire tree that has currently been induced. For example, suppose we are building regex (b) from Figure 2, and suppose we currently have `concat(reprange(<num>,1,2),concat(<.>,Comp))`, where **`Comp`** is a non-terminal that needs to be expanded into a sub-regex. Because we already have `reprrange(<num>,1,2)` and `<.>` in the current tree, we increase the probability of expanding **`Comp`** to generate these particular two sub-regexes, allowing the model to copy from what it has generated before.[3]

In addition to copying, we also change the sampling distribution when sampling children of certain grammar constructs to control for complexity and encourage sampling of valid regexes. For example, the child of a `startwith` expression will typically be less complex and compositional than the child of a `Comp` expression, so we tune the probabilities of sampling compositional AST operators like `or` appropriately.

3 Dataset Collection

3.1 Positive/Negative Example Generation

The STACKOVERFLOW dataset (Ye et al., 2019) shows that programmers often provide both positive and negative examples to fully convey their intents while specifying a complicated regex. Therefore, we augment our dataset with positive and negative examples for each regex. Our model will use these examples to resolve ambiguity present in the natural language descriptions. However, the examples can also help Turkers to better understand the regexes they are describing during the data collection process.

[3]This component reuse bears some similarity to an Adaptor Grammar (Johnson et al., 2007). However, we modify the distributions in a way that violates exchangeability, making it not formally equivalent to one.

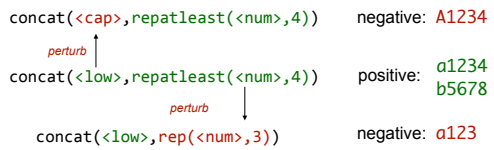


Figure 5: The process of generating distinguishing negative examples by minorly perturbing each of the sub-regexes in the ground truth regex.

We aim to generate diverse and distinguishing examples similar to human-written ones, which often include corner cases that differentiate the ground truth regex from closely-related spurious ones. We can achieve this by enumerating examples that cover the states in the deterministic finite automaton (DFA) defined by the given regex[4] and reject similar but incorrect regexes. We employ the Automaton Library (Møller, 2017) to generate the examples in our work. Positive examples are generated by stochastically traversing the DFA.

For negative examples, randomly sampling examples from the negation of a given regex will typically produce obviously wrong examples and not distinguishing negative examples as desired. Therefore, we propose an alternative approach shown in Figure 5 for generating negative examples. We apply minor perturbations to the ground truth regex to cause it to accept a set of strings that do not intersect with the set recognized by the original regex. The negative examples can be derived by sampling a positive string from one of these "incorrect" regexes.

For each regex in our dataset, we generate 6 positive examples and 6 negative examples. These numbers are comparable to the average number of examples provided by STACKOVERFLOW users.

3.2 Figure Generation

As stated previously, we avoid the paradigm of asking users to paraphrase machine-generated regex descriptions, as this methodology can yield formulaic and artificial descriptions. Instead, we ask users to describe regexes based on figures that illustrate how the regex is built. We show one example figure of a SEPARATION regex in Figure 6. In general, we abstract a given regex as a series of blocks linked with textual descriptions of its content and constraints. For instance, `startwith` and `endwith` are denoted by shading the head or tail of a block. By linking multiple blocks to shared tex-

[4]Recall that although our DSL is tree-structured, it is equivalent in power standard regexes, and hence our expressions can be mapped to DFAs.

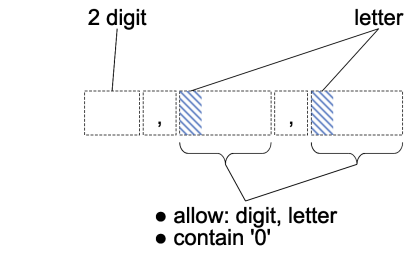


Figure 6: An example automatically generated figure of a SEPARATION regex and corresponding description annotated by a Turker.

tual descriptions, we hope to encourage Turkers to notice the correlation and write descriptions accordingly. Finally, we have different textual hints for the same concept: "contain x" in Figure 6 may appear as "have x" elsewhere. These figures are rendered for each regex in the MTurk interface using JavaScript.

3.3 Crowdsourcing

Task We collected the STRUCTUREDREGEX dataset on Amazon Mechanical Turk (MTurk). For each HIT, the Turkers are presented with a regex figure and a set of positive/negative examples. Then, they are asked to write down several sentences describing the regex, as well as one additional positive example that matches the regex. We only accept a description if the submitted positive example is matched by the ground-truth regex; this helps filter out some cases where the Turker may have misunderstood the regex. We show an example HIT in Appendix C.

In early pilot studies, we explored other ways of abstractly explaining regexes to Turkers, such as providing more examples and an associated set of keywords, yet none of these methods led to users generating sufficiently precise descriptions. By contrast, our figures fully specify the semantics of the regexes while only minimally biasing Turkers towards certain ways of describing them.

We generated 1,200 regexes (400 from each template), assigned each regex to three Turkers, and collected a total of 3,520 descriptions after rejecting HITs. In general, each Turker spent 2 to 3 minutes on each of the HITs, and we set the reward to be $0.35. The total cost of collecting our dataset was $1,512, and the average cost for each description is $0.43.

Dataset	KB13	TURK	STREG	SO
size	824	10000	3520	62
#. unique words	207	557	873	301
avg. NL length	8	12	33	25
avg. reg size	5	5	15	13
avg. reg depth	2.5	2.3	6.0	4.0

Table 1: Statistics of our dataset and prior datasets. Compared to KB13 and NL-TURK, our dataset contain more diverse language and more complex regexes, comparable to the real STACKOVERFLOW dataset.

Quality To ensure the quality of collected responses, we require the Turkers to first take a qualification test which simply requires describing one regex that we have specified in advance. We then check that the description for this regex is sufficiently long and that it contains enough of our manually-written correct base regex concepts.

We manually observed from the responses that various styles were adopted by different Turkers for describing the same type of regexes. For instance, given regex (b) in Figure 2, some Turkers tend to enumerate every component in order, describing it as *one or two digits followed by a dot followed by one or two digits*; some other Turkers prefer grouping identical components and describing the components out of order, describing it as *the first and third parts are one or two digits, and the second part is a dot*. These distinct styles lead to a diversity of linguistic phenomena, which is further analyzed in Section 4. Because we aim for high linguistic diversity in our dataset, we prohibited a single Turker from doing more than 300 HITs.

Furthermore, we found anecdotal evidence that the task was engaging for users, which we took as a positive signal for generation quality. We received messages about our HITs from some Turkers telling us that our HIT was "really interesting" and they "enjoyed doing it."

Splitting the Dataset Since our dataset consists of natural language descriptions written by annotators, there is possibly bias introduced by training and testing on the same annotators (Geva et al., 2019). Therefore, in addition to the standard Train/Development/Test splits, we also form a Test-E (excluded) which consists only of annotations from annotators unseen in the training set. We ensure that Train, Dev, and both two test sets (Test and Test-E) have mutually exclusive regexes from each other (Test and Test-E can have common regexes), and Test-E is annotated entirely by

	TURK	STREG	Example NL from STREG
multi-sentence	0%	70%	The string has 6 or more characters**.** The string must start with a digit.
ambiguity	2.3%	20.6%	The sequence starts with a letter followed by 2 numbers.
abstraction	0%	13.3%	The first part of a single string consists of 1 or more "0" followed by 2 capital letters. The second part of the string must follow **the same rules**.
non-local constraint	0%	16.7%	There are 3 dash separated strings. The first is 1 to 4 "A" . The **second and third** consist of 1 or 2 "x" followed by 1 to 3 numbers and 2 letters.
coreference	5.1%	29.7%	The string starts with a number. **It** ends with 1 to 4 lower or capital letters.
condition relation	0%	3.5%	**If** there is a capital letter it must be after a digit.
adversative relation	0%	3.7%	The string start with capital letter **but** it should not be a "A".

Table 2: Qualitative analysis on 150 descriptions from NL-TURK and our dataset (50 from each template). We show the percentage of examples containing each phenomenon. Our dataset features more of these challenging linguistic phenomena compared to prior synthetic datasets.

a disjoint set of annotators from those who annotated the training or development set. The final size of the splits are: 2173 (61.7%), 351 (10.0%), 629 (17.9%), 367 (10.4%).

4 Dataset Analysis

We demonstrate the advantages of our dataset over prior datasets (Kushman and Barzilay, 2013; Locascio et al., 2016) through both quantitative and qualitative analysis.

We list the key statistics of our dataset as well as KB13 and NL-TURK for comparison in Table 1. Compared to past synthetic datasets, our dataset has more diverse and sophisticated language. The average NL length of our dataset is twice as long as that of NL-TURK, and the descriptions contain many more unique words even though our dataset contains fewer regexes. In addition, our dataset contains more complex regexes that are closer to the complexity of real-world regexes found on StackOverflow, whereas regexes in previous datasets are significantly simpler.

Manual Analysis We further manually analyze 150 descriptions from past synthetic datasets and our dataset. Table 2 lists the proportion of descriptions containing each of several phenomena: examples that are *multi-sentence*, examples with clear syntactic or semantic *ambiguity*, examples using *abstraction* to refer to different parts of the regex, examples invoking *non-local constraints*, and examples with nontrivial *coreference*. The language from our dataset is organic and diverse, since we allow Turkers to compose their own descriptions. We find that macros and complex constraints in our structured grammar can successfully trigger interesting language. For instance, the abstraction reflects repetition in concatenation regexes, and the bottom part of Table 2 reflects the complex macros.

	KB13	TURK	STREG
Word Coverage	27.1%	34.4%	**55.9%**
Regex Coverage	23.5%	23.5%	**84.3%**

Table 3: Distribution mismatch analysis with respect to STACKOVERFLOW on past datasets and our dataset. Our dataset covers significantly more words and regexes, and is closer to the real-world dataset.

Furthermore, the complex and ambiguous language highlights the necessity of including examples together with language to fully specify a regex. For instance, ambiguity is common in our descriptions. However, many of the ambiguous descriptions can be resolved with the help of examples. Concretely, the description for *ambiguity* from Table 2 can be easily interpreted as `startwith(concat(<let>,repeat(<num>,2)))` while the ground truth is `concat(<let>,repeat(<num>,2))`. By simply adding one negative example, "a123", the ground truth can be distinguished from the spurious regex.

Comparison to STACKOVERFLOW Since our goal was to produce realistic regex data, we analyze how well the real-world STACKOVERFLOW dataset is covered by data from STRUCTUREDREGEX compared to other datasets (Kushman and Barzilay, 2013; Locascio et al., 2016). We ignore 11 of the STACKOVERFLOW examples that involve the high-level *decimal* concept, which is beyond the scope of our dataset and past synthetic datasets. In addition, we anonymize all the constants and integer parameters (e.g., `repeat(<x>,9)` is anonymized as `repeat(const,int)`). The statistics (Table 3) suggest that our dataset is more highly similar to real-world regexes on StackOverflow, especially in terms of regex distribution.

5 Methods

We evaluate the accuracy of both existing grammar-based approaches and neural models, as well as a novel method that targets the multi-modal nature of our dataset.

Existing Approaches SEMANTIC-UNIFY (Kushman and Barzilay, 2013) is a grammar-based approach that relies on a probabilistic combinatory categorical grammar to build the regexes. DEEPREGEX (Locascio et al., 2016) directly translates natural language descriptions into regexes using a seq-to-seq model enhanced with attention (Luong et al., 2015) without considering examples. We re-implemented DEEPREGEX with slightly different hyperparameters; we refer to our re-implementation as DEEPREGEX (OURS). DEEPREGEX+FILTER (Ye et al., 2019) adapts DEEPREGEX so as to take examples into account by simply filtering the k-best regexes based on whether a regex accepts all the positive examples and rejects all the negative ones.

Example-Guided Decoding Although DEEPREGEX+FILTER is able to take advantage of positive and negative string examples, these examples are completely isolated in the training and inference phase. We propose to make use of examples during inference with the technique of over- and under- approximation (Lee et al., 2016) used in the program synthesis domain. The core idea of our approach is that, for each partially completed regex during decoding, we use the approximation technique to infer whether the regex can possibly match all positive or reject all negative examples. If this is impossible, we can prune this partial regex from our search. This approach allows us to more effectively explore the set of plausible regexes without increasing the computational budget or beam size.

As an example, consider the ground truth regex `and(startwith(<low>),endwith(<num>))` with one corresponding positive example "00x". Suppose that the decoder has so far generated the incomplete regex `and(startwith(<cap>),`. To produce a syntactically valid regex, the decoder needs to generate a second argument for the `and`. By appending `star(<any>)` as its second argument, we can see that there is no completion here that will accept the given positive example, allowing us to reject this regex from the beam. Under-approximation works analogously, completing regexes with maximally restrictive arguments and checking that negative examples are rejected.

We integrate the aforementioned technique in the beam decoding process by simply pruning out bad partial derivations at each timestep. We refer to this approach as DEEPREGEX + APPROX.

Approach	KB13	TURK	STREG
SEMANTIC-UNIFY	65.5%	38.6%	1.8%
DEEPREGEX (Locascio et al.)	65.6%	58.2%	–
DEEPREGEX (Ours)	66.5%	60.2%	24.5%
DEEPREGEX + FILTER	77.7%	83.8%	37.2%

Table 4: DFA-equivalent accuracy on prior datasets and our dataset. The performance on our dataset using any model is much lower than the performance on existing datasets.

6 Experiments

6.1 Comparison to Prior Datasets

We evaluate the baseline models on KB13, NL-TURK, and our dataset (Table 4). The results show that our dataset is far more challenging compared to existing datasets. Traditional grammar baseline can scarcely solve our dataset. The best baseline, DEEPREGEX + FILTER, achieves more than 77.7% on KB13 and 83.8% NL-TURK when these datasets are augmented with examples, but can only tackle 37.2% of our dataset. Additionally, the comparison between DEEPREGEX and DEEPREGEX + FILTER demonstrates that simply filtering the outputs of neural model leads to a substantial performance boost on all the datasets. This supports the effectiveness of the way we specify regexes, i.e., using both natural language descriptions and examples.

6.2 Detailed Results on STRUCTUREDREGEX

Table 5 shows the detailed accuracy regarding different regex templates on both Test and Test-E sets. Our DEEPREGEX + APPROX achieves best accuracy with 5.6% and 7.9% improvement over DEEPREGEX + FILTER on Test and Test-E, respectively, since it can leverage examples more effectively using over- and under- approximations during search.

Accuracy varies on different types of regexes. Generally, models perform the best on concatenation regexes, slightly worse on intersection regexes, and the worst on separation regexes. Concatenation regexes usually have straightforward

Approach	Test				Test-E			
	Agg	Int	Cat	Sep	Agg	Int	Cat	Sep
SEMANTIC-UNIFY	2.1%	2.9%	3.1%	0.0%	1.4%	1.6%	2.4%	0.0%
DEEPREGEX (Ours)	27.8%	20.7%	42.2%	19.2%	18.8%	18.0%	23.6%	14.8%
DEEPREGEX + FILTER	42.6%	38.9%	55.2%	32.3%	28.1%	32.0%	32.5%	19.7%
DEEPREGEX + APPROX	48.2%	45.7%	59.6%	37.9%	36.0%	39.3%	40.7%	27.9%

Table 5: Results for models trained and tested on STRUCTUREDREGEX. Using the examples (the latter two methods) gives a substantial accuracy boost, and DEEPREGEX + APPROX is better than the post-hoc FILTER method, but still only achieves 48.2% accuracy on Test and 36.0% on Test-E. Separation regexes are more difficult than the other two classes, and performance for all models drops on Test-E.

Train Set	Model DEEPREGEX	Acc	Equiv Found	Consistent Found
TURK	w/o Example	0.0%	0.0%	7.8%
STREG	+ FILTER	9.8%	9.8%	21.6%
STREG	+APPROX	13.7%	17.6%	37.7%

Table 6: The performance on STACKOVERFLOW-51 with models trained on NL-TURK and our dataset. We report the fraction of examples where a DFA-equivalent regex is found (Acc), where a DFA-equivalent regex is found in the k-best list, and where a regex consistent with the examples appears in the k-best list. Models trained on NL-TURK do not perform well in this setting, while our models can solve some examples.

descriptions in the form of listing simple components one by one. Intersection descriptions can be more complicated because of the high-level macros specified by our grammar. Separation descriptions are the most complex ones that often involve coreferences and non-local features. Performance on Test-E is 12% lower than on Test for the models haven't been trained on patterns of the unseen annotators.

6.3 Transferability Results

Finally, we investigate whether a model trained on our dataset can transfer to the STACKOVERFLOW dataset. As in Section 4, we ignore instances requiring the decimal concept and only evaluate on the subset of STACKOVERFLOW with 51 instances. We compare our dataset with NL-TURK for this task. As shown in Table 6, DEEPREGEX trained on NL-TURK completely fails on STACKOVERFLOW and even fails to predict reasonable regexes that are consistent with the examples. This is caused by the fact that the NL-TURK dataset contains formulaic descriptions and shallow regexes that are not representative of real-world tasks. DEEPREGEX trained on our dataset can at least achieve 9.8% accuracy on STACKOVERFLOW dataset because the English descriptions in this dataset better match the desired task. Our DEEPREGEX + APPROX model successfully solves 13.7% and finds consistent regexes for 38% of the tasks, which is credible given that the performance of the same model on Test-E set is only 30%. Some additional challenges in STACKOVERFLOW are instances involving large numbers of constants or slightly more formal language since the SO users are mainly programmers. However, we believe the transfer results here show that improved performance on our dataset may transfer to STACKOVERFLOW as well, since some of the challenges also present in our Test-E set (e.g., unseen language).

6.4 Human Performance Estimate

It is difficult to hire Turkers to estimate a human performance upper bound, because our task requires reckoning with both the descriptions and positive/negative examples. Unlike many NLP tasks where an example with ambiguous language is fundamentally impossible, here the examples may actually still allow a human to determine the correct answer with enough sleuthing. But to perform this task, crowdworkers would minimally need to be trained to understand the DSL constructs and how they compose, which would require an extensive tutorial and qualification test. To do the task well, Turkers would need a tool to do on-the-fly execution of their proposed regexes on the provided examples.

We instead opted for a lighter-weight verification approach to estimate human performance. We adopted a post-editing approach on failure cases from our model, where we compared the model's output with the input description and examples and corrected inconsistencies.

Specifically, we sample 100 failure examples from the test set (Test plus Test-E) and manually assess the failure cases. We find **78%** of failure cases contain descriptions that describe all com-

ponents of the target regexes, but our seq-to-seq models are insufficient to capture these. There are truly some mis- or under-specified examples, such as not mentioning the optionality of one component or mistaking "I" for "l" in constants. An additional **9%** (out of 100) of the errors could be fixed using the provided examples. This leaves roughly 13% of failure cases that are challenging to solve.

Considering that the model already achieves 43.6% accuracy on the test set, we estimate human performance is around 90%.[5]

7 Related Work

Data collection in semantic parsing Collecting large-scale data for semantic parsing and related tasks is a long-standing challenge (Berant et al., 2013; Wang et al., 2015). Wang et al. (2015) proposed the generate-and-paraphrase framework, which has been adopted to collect datasets in various domains (Locascio et al., 2016; Ravichander et al., 2017; Johnson et al., 2017). However, this process often biases annotators towards using formulaic language (Ravichander et al., 2017; Herzig and Berant, 2019).

Similar to our work, past work has sought to elicit linguistically diverse data using visual elements for semantic parsing (Long et al., 2016), natural language generation (Novikova et al., 2016), and visual reasoning (Suhr et al., 2017, 2019). However, for these other tasks, the images used are depictions of an *inherently graphical* underlying world state; e.g., the NLVR dataset (Suhr et al., 2017) and NLVR2 (Suhr et al., 2019) are based on reasoning over the presented images, and the Tangrams dataset (Long et al., 2016) involves describing shape transformations. By contrast, regexes are typically represented as source code; there is no standard graphical schema for depicting the patterns they recognize. This changes the properties of the generated descriptions, leading to higher levels of compositionality and ambiguity because what's being described is not naturally an image.

Program and regex synthesis Recent research has tackled the problem of program synthesis from examples (Gulwani, 2011; Gulwani and Jain, 2017; Alur et al., 2013; Wang et al., 2016; Feng et al., 2018; Devlin et al., 2017; Nye et al., 2019). A closer line of work to ours uses both examples and natural language input (Yaghmazadeh et al., 2017; Ye et al., 2019; Andreas et al., 2018), which involves fundamentally different techniques. However, our work does not rely on the same sort of program synthesizer to build final outputs (Yaghmazadeh et al., 2017; Ye et al., 2019). Moreover, Andreas et al. (2018) only use language at train time, whereas we use NL at both train and test time.

Finally, while several datasets on regex synthesis specifically have been released (Kushman and Barzilay, 2013; Locascio et al., 2016), we are the first to incorporate examples in the dataset. Other methods have been proposed to parse natural language into regex via rule-based (Ranta, 1998), grammar-based (Kushman and Barzilay, 2013), or neural models (Locascio et al., 2016; Zhong et al., 2018; Ye et al., 2019). Notably, Zhong et al. (2018) also generate distinguishing examples to facilitate translation, but they require a trained model to generate examples, and we organically derive examples from the structure of regexes without additional input.

8 Conclusion

We introduce StructuredRegex, a new dataset for regex synthesis from natural language and examples. Our dataset contains compositionally structured regexes paired with linguistically diverse language, and organically includes distinguishing examples. Better methods are needed to solve this dataset; we show that such methods might generalize well to real-world settings.

Acknowledgments

This work was partially supported by NSF Grant IIS-1814522, NSF Grant SHF-1762299, a gift from Arm, 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 the anonymous reviewers for their helpful comments.

[5] In addition, the first author manually wrote regexes for 100 randomly sampled examples and achieved an accuracy of 95% (higher than the estimate). However, the author also has a strong prior over what synthetic regexes are likely to be in the data.

References

Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo MK Martin, Mukund Raghothaman, Sanjit A

Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. Syntax-guided Synthesis. In *2013 Formal Methods in Computer-Aided Design (FMCAD)*.

Jacob Andreas, Dan Klein, and Sergey Levine. 2018. Learning with Latent Language. In *Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL)*.

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)*.

Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. 2017. Robustfill: Neural Program Learning under Noisy I/O. In *Proceedings of the International Conference on Machine Learning (ICML)*.

Yu Feng, Ruben Martins, Osbert Bastani, and Isil Dillig. 2018. Program Synthesis Using Conflict-driven Learning. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

Jeffrey EF Friedl. 2006. *Mastering Regular Expressions*. " O'Reilly Media, Inc.".

Mor Geva, Yoav Goldberg, and Jonathan Berant. 2019. Are We Modeling the Task or the Annotator? An Investigation of Annotator Bias in Natural Language Understanding Datasets. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing and the International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*.

Sumit Gulwani. 2011. Automating String Processing in Spreadsheets Using Input-output Examples. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*.

Sumit Gulwani and Prateek Jain. 2017. Programming by Examples: PL Meets ML. In *Proceedings of the Asian Symposium on Programming Languages and Systems (APLAS)*.

Jonathan Herzig and Jonathan Berant. 2019. Don't paraphrase, detect! Rapid and Effective Data Collection for Semantic Parsing. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing and the International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*.

Justin Johnson, Bharath Hariharan, Laurens van der Maaten, Li Fei-Fei, C Lawrence Zitnick, and Ross Girshick. 2017. Clevr: A Diagnostic Dataset for Compositional Language and Elementary Visual Reasoning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

Mark Johnson, Thomas L. Griffiths, and Sharon Goldwater. 2007. Adaptor Grammars: A Framework for Specifying Compositional Nonparametric Bayesian Models. 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 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NACCL)*.

Mina Lee, Sunbeom So, and Hakjoo Oh. 2016. Synthesizing Regular Expressions from Examples for Introductory Automata Assignments. In *Proceedings of the ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE)*.

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)*.

Reginald Long, Panupong Pasupat, and Percy Liang. 2016. Simpler Context-Dependent Logical Forms via Model Projections. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics (ACL)*.

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)*.

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

Jekaterina Novikova, Oliver Lemon, and Verena Rieser. 2016. Crowd-sourcing NLG Data: Pictures Elicit Better Data. In *Proceedings of the International Natural Language Generation conference (INLG)*.

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)*.

Aarne Ranta. 1998. A Multilingual Natural-Language Interface to Regular Expressions. In *Finite State Methods in Natural Language Processing*.

Abhilasha Ravichander, Thomas Manzini, Matthias Grabmair, Graham Neubig, Jonathan Francis, and Eric Nyberg. 2017. How Would You Say It? Eliciting Lexically Diverse Dialogue for Supervised Semantic Parsing. In *Proceedings of the Annual SIGdial Meeting on Discourse and Dialogue (SIGDIAL)*.

Alane Suhr, Mike Lewis, James Yeh, and Yoav Artzi. 2017. A Corpus of Natural Language for Visual Reasoning. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics (ACL)*.

Alane Suhr, Stephanie Zhou, Ally Zhang, Iris Zhang, Huajun Bai, and Yoav Artzi. 2019. A Corpus for Reasoning about Natural Language Grounded in Photographs. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics (ACL)*.

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

Yushi Wang, Jonathan Berant, and Percy Liang. 2015. Building a Semantic Parser Overnight. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics (ACL)*.

Navid Yaghmazadeh, Yuepeng Wang, Isil Dillig, and Thomas Dillig. 2017. SQLizer: Query Synthesis from Natural Language. In *Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*.

Xi Ye, Qiaochu Chen, Xinyu Wang, Isil Dillig, and Greg Durrett. 2019. Sketch-Driven Regular Expression Generation from Natural Language and Examples. In *arXiv preprint arXiv:1908.05848*.

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

A Regex DSL

Nonterminals r :=	
startwith(r)	r.*
| endwith(r)	.*r
| contain(r)	.*r.*
| not(r)	~r
| optional(r)	r?
| star(r)	r*
| concat(r_1, r_2)	$r_1 r_2$
| and(r_1, r_2)	r_1&r_2
| or(r_1, r_2)	r_1|r_2
| rep(r,k)	r{k}
| repatleast(r,k)	r{k,}
| reprange(r,k_1,k_2)	r{k_1,k_2}
Terminals t :=	
<let>	[A-Za-z]
| <cap>	[A-Z]
| <low>	[a-z]
| <num>	[0-9]
| <any>	.
| <spec>	[-,;.+:!@#_$%&*=^]
| <null>	∅

Table 7: Our regex DSL and the corresponding constructions in standard regular language. Our regex DSL is as expressive as and can be easily translated to standard regex syntax.

B Details of Structured Grammar

B.1 Grammar Rules

See Figure 7.

B.2 Implementation Details

Intersection While building INTERSECTION regexes, we impose context-dependent constraints mainly to avoid combinations of regexes that are redundant or in conflict. Conflicts often occur between a `ComposedBy` constraint and the other constraints. A `ComposedBy` constraint indicates the allowed characters; e.g., `repeatatleast(or(<let>,<spec>),1)` means there can only be letters and special characters in the matched string. Therefore, when we already have such a constraint in the tree, we only allow the terminals to be selected from the valid subset of `<let>` and `<spec>` while expanding the other subtrees.

This greatly reduce the chances of yielding empty regexes as well as redundant regexes (e.g., in `and(repeatatleast(or(<let>,<spec>),1),not(contain(<num>)))`, the second constraint is actually redundant).

Concatenation CONCATENATION regexes are a sequence of simple components. As stated above, our grammar encourages the phenomenon of repetition that commonly occurs in real regexes by copying existing sub-trees.

Separation SEPARATION regexes have several subfields, which can be specified by either INTERSECTION regexes or CONCATENATION regexes, and which are delimited by a constant. The fields of real regexes are often related, i.e., they share common components. For instance, the format of U.S. phone numbers is "xxx-xxx-xxxx" where "x" is a digit. Here the three fields are all digits but differ in length. Similar to the CONCATENATION template, we alter the distribution so as to copy the already generated subtrees.

We also allow a class of SEPARATION with an arbitrary number of identical fields separated by a constant (e.g., *a list of comma-separated numbers*).

Complexity Control We aim to create a collection of complicated regexes, but we do not wish to make them needlessly complex along unrealistic axes. We assess the complexity of generated regexes using a measure we call *semantic complexity*, which roughly measures how many factors would need to be specified by a user. Generally, each constraint or components counts for one degree of semantic complexity, e.g., `not(contain(x))` and `repeat(x,4)` are of complexity level one. High-level macro constraints are of complexity level two since they need more verbal explanation. We limit the complexity degrees all of our generated regexes to be strictly no more than six. More details about the number of nodes and depth of our regexes can be found in Section 4.

C HIT Example

See Figure 8.

Intersection Template

```
IntTemp → Cons | and(Cons,IntTemp)
Cons → BasicCons | LengthCons | MacroCons
BasicCons → not(BasicCons)
BasicCons → startwith(ConsExpr)|endwith(ConsExpr)| contain(ConsExpr)
LengthCons → rep(<any>,k)| repatleast(<any>,k) |reprange(<any>,k,k)
MacroCons → ConsistOfCons|AdvStartwithCons| AdvEndwithCons | CondContainCons
ConsistOfCons → repatleast(LiteralSet,1)
AdvStartwithCons → and(startwith(Literal),not(startwith(Literal)))
AdvEndwithCons → and(endwith(Literal),not(endwith(Literal)))
CondContainCons → not(contain(concat(Literal,notcc(Literal))))
CondContainCons → not(contain(concat(notcc(Literal),Literal)))
ConsExpr → LiteralSet|MinConsExpr|concat(MinConsExpr,MinConsExpr)
MinConsExpr → Literal|rep(Literal,k)
```

Concatenation Template

```
CatTemp → Comp, concat(Comp, CatTemp)
Comp → optional(Comp)
Comp → BasicComp| MacroComp
BasicComp → CompExpr|rep(CompExpr,k)| repatleast(CompExpr,k) |reprange(CompExpr,k,k)
MacroComp → or(rep(<Literal>,k),rep(<Literal>,k))
MacroComp → or(repatleast(<Literal>,k),repatleast(<Literal>,k))
MacroComp → or(reprange(<Literal>,k,k),reprange(<Literal>,k,k))
CompExpr → Literal|LiteralSet
```

Separation Template

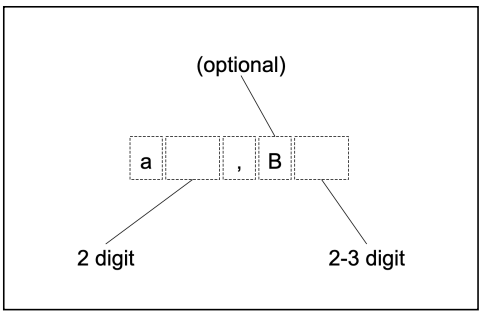
```
SepTemp → concat(Seg,Delimiter,Seg,Delimiter,Seg)
SepTemp → concat(Seg,star(concat(Delimiter,Seg))
Seg → IntTemp|CatTemp
Delimiter → CONST
```

Literals etc.

```
Literal → CC | CONST | STR   # CONST can be any const character, STR can be any string values.
CC → <num>|<let>|<low>|<cap>|<spec>
LiteralSet → Literal|or(Literal,LiteralSet)
```

Figure 7: Grammar rules for generating regexes in our dataset. Our grammar contains much more rules than a standard regex grammar, and is highly structured in that we have high-level templates and macros.

Instructions:
In this task, you will be writing down descriptions of the patterns you see in a group of strings. For each HIT, you'll be given a figure visually specifying a pattern and a few examples of strings following or not following the pattern to help you to understand it. Please write a description (generally 1-4 sentences) that describes the pattern. In addition, please write one additional string that follows the pattern.
Things to keep in mind:
• Please describe the pattern underlying the string examples, not the sequence of strings itself. Do not write things like "the first line ..., the second line ...."
• Try to be precise about describing the pattern, but also concise. Don't describe the same property of the strings in multiple ways.
• You are not required to use the keywords in the figure. If you can think of another way to express the intent, that's okay.
• Please try to write natural and fluent sentences.
• Additional string example must be different.



Example strings that follow the pattern:
a51,B457
a74,B23
a09,849
Example strings that do not follow the pattern:
b55,B193
a7,B23
a09,1

Figure 8: HIT prompt for the description writing task. We particularly emphasize in the instructions that Turkers should use precise and original language.