SemRegex: A Semantics-Based Approach for Generating Regular Expressions from Natural Language Specifications

Zexuan Zhong[1], **Jiaqi Guo**[2], **Wei Yang**[3], **Jian Peng**[1], **Tao Xie**[1],
Jian-Guang Lou[4], **Ting Liu**[2], **Dongmei Zhang**[4]
[1]University of Illinois at Urbana-Champaign
[2]Xi'an Jiaotong University, [3]University of Texas at Dallas
[4]Microsoft Research Asia
`{zexuan2,jianpeng,taoxie}@illinois.edu`
`jasperguo2013@stu.xjtu.edu.cn, tingliu@mail.xjtu.edu.cn`
`wei.yang@utdallas.edu, {jlou,dongmeiz}@microsoft.com`

Abstract

Recent research proposes syntax-based approaches to address the problem of generating programs from natural language specifications. These approaches typically train a sequence-to-sequence learning model using a syntax-based objective: maximum likelihood estimation (MLE). Such syntax-based approaches do not effectively address the goal of generating semantically correct programs, because these approaches fail to handle *Program Aliasing*, i.e., semantically equivalent programs may have many syntactically different forms. To address this issue, in this paper, we propose a semantics-based approach named SemRegex. SemRegex provides solutions for a subtask of the program-synthesis problem: generating regular expressions from natural language. Different from the existing syntax-based approaches, SemRegex trains the model by maximizing the expected semantic correctness of the generated regular expressions. The semantic correctness is measured using the DFA-equivalence oracle, random test cases, and distinguishing test cases. The experiments on three public datasets demonstrate the superiority of SemRegex over the existing state-of-the-art approaches.

1 Introduction

Translating natural language (NL) descriptions into executable programs is a fundamental problem for computational linguistics. An end user may have difficulty to write programs for a certain task, even when the task is already specified in NL. For some tasks, even for developers, who have experience in writing programs, it can be time consuming and error prone to write programs based on the NL description of the task. Naturally, automatically synthesizing programs from NL can help alleviate the preceding issues for both end users and developers.

Recent research proposes syntax-based approaches to address some tasks of this problem in different domains, such as regular expressions (regex) (Locascio et al., 2016), Bash scripts (Lin et al., 2017), and Python programs (Yin and Neubig, 2017). These approaches typically train a sequence-to-sequence learning model using maximum likelihood estimation (MLE). Using MLE encourages the model to output programs that are syntactically similar with the ground-truth programs in the training set. However, such syntax-based training objective deviates from the goal of synthesizing semantically equivalent programs. Specifically, these syntax-based approaches fail to handle the problem of *Program Aliasing* (Bunel et al., 2018), i.e., a semantically equivalent program may have many syntactically different forms. Table 1 shows some examples of the *Program Aliasing* problem. Both Program 1 and Program 2 are desirable outputs for the given NL specification but one of them is penalized by syntax-based approaches if the other one is used as the ground truth, compromising the overall effectiveness of these approaches.

In this paper, we focus on generating regular expressions from NL, an important task of the program-synthesis problem, and propose SemRegex, a semantics-based approach to generate regular expressions from NL specifications. Regular expressions are widely used in various applications, and "regex" is one of the most common tags in Stack Overflow[1] with more than $190,000$ related questions. The huge number of regex-related questions indicates the importance of this task.

Different from the existing syntax-based approaches, SemRegex alters the syntax-based training objective of the model to a semantics-based objective. To encourage the translation model to generate semantically correct regular expressions, instead of MLE, SemRegex trains the model by maximizing the expected semantic correctness of

[1]`https://stackoverflow.com/questions/tagged/regex`

Table 1: Examples of *Program Aliasing*: for each NL specification, Program 2 is semantically equivalent to Program 1; however, if Program 1 is the ground truth in the training set, Program 2 is penalized by syntax-based approaches although it is a desirable program.

Domain	**NL Specification**	**Program 1**	**Program 2**
Regex	Match lines that start with an uppercase vowel and end with 'X'	`([AEIOUaeiou]&[A-Z]).*X`	`([AEIOU].*)&(.*X)`
Bash	Rename file 'f1' to 'f1.txt'	`mv 'f1' 'f1.txt'`	`cp 'f1' 'f1.txt'; rm 'f1'`
Python	Assign the greater value of 'a' and 'b' to variable 'c'	`c = a if a > b else b`	`c = [b, a][a > b]`

generated regular expressions. We follow the technique of policy gradient (Williams, 1992) to estimate the gradients of the semantics-based objective and perform optimization.

The measurement of semantic correctness serves as a key part in the semantics-based objective, which should represent the semantics of programs. In this paper, we convert a regular expression to a minimal Deterministic Finite Automaton (DFA). Such conversion is based on the insight that semantically equivalent regular expressions have the same minimal DFAs. We define the semantic correctness of a generated regular expression as whether its corresponding minimal DFA is the same as the ground truth's minimal DFA.

When our approach is applied on domains other than regular expressions such as Python programs and Bash scripts, a perfect equivalence oracle such as minimal DFAs may not be feasibly available. To handle a more general case, we propose correctness assessment based on test cases for regular expression; such correctness assessment can be easily generalized for other tasks of program synthesis. Concretely, we generate test cases to represent semantics of the ground truth. For a generated regular expression, we assess its semantic correctness by checking whether it can pass all the test cases. However, a regular expression may have infinite positive (i.e., matched) or negative (i.e., unmatched) string examples; thus, we cannot perfectly represent the semantics. To use limited string examples to differentiate whether a generated regular expression is semantically correct or not, we propose an intelligent strategy for test generation to generate distinguishing test cases instead of just using random test cases.

We evaluate SemRegex on three public datasets: NL-RX-Synth, NL-RX-Turk (Locascio et al., 2016), and KB13 (Kushman and Barzilay, 2013). We compare SemRegex with the existing state-of-the-art approaches on the task of generating regular expressions from NL specifications. Our evaluation results show that SemRegex outperforms the start-of-the-art approaches on all of three datasets. The evaluation results confirm that by maximizing semantic correctness, the model can output more correct regular expressions even when the regular expressions are syntactically different from the ground truth.

In summary, this paper makes the following three main contributions. (1) We propose a semantics-based approach to optimize the semantics-based objective for the task of generating regular expressions from NL specifications. (2) We introduce the measurement of semantic correctness based on test cases, and propose a strategy to generate distinguishing test cases, in order to better measure the semantic correctness than using random test cases. (3) We evaluate our approach on three public datasets. The evaluation results show that our approach outperforms the existing state-of-the-art on all of the three datasets.

2 Problem Formulation

Consider the problem of automatically generating a regular expression R given an NL specification S as an input. Let $S = s_1, s_2, \ldots, s_m$ denote the NL specification, where s_i represents a word in the vocabulary; let $R = r_1, r_2, \ldots, r_n$ denote the regular expression, where r_i is a valid character in the regular expression.

We assume that we have a training set consisting of K NL and regular expression pairs:

$$\mathcal{D} = \left\{(S^{(i)}, R^{(i)})\right\}_{i=1..K}$$

Given an NL specification, it is possible to have multiple regular expressions fitting the specification. In the training set, only one regular expression is provided for each NL specification.

3 SemRegex Approach

In this section, we illustrate our SemRegex approach in detail. First, we introduce our model, which is a sequence-to-sequence learning model.

Next, we alter the standard Maximum Likelihood Estimation (MLE) objective to maximize semantic correctness. We leverage policy gradient to train the model with the semantics-based objective. Finally, we discuss how to measure semantic correctness.

3.1 Model

It is natural to apply a machine-translation model on the program-synthesis problem. We follow a previous attempt (Locascio et al., 2016) to use a sequence-to-sequence learning model (Sutskever et al., 2014) augmented with the attention mechanism (Bahdanau et al., 2014). The model consists of an encoder network and a decoder network. In both the encoder network and decoder network, we use LSTM (Hochreiter and Schmidhuber, 1997) units that can be summarized as follow:

$$\begin{aligned} i_t &= \sigma(W_i x_t + U_i h_{t-1} + b_i) \\ f_t &= \sigma(W_f x_t + U_f h_{t-1} + b_f) \\ o_t &= \sigma(W_o x_t + U_o h_{t-1} + b_o) \\ \tilde{c}_t &= \phi(W_c x_t + U_c h_{t-1} + b_c) \\ c_t &= f_t \circ c_{t-1} + i_t \circ \tilde{c}_t \\ h_t &= o_t \circ \phi(c_t) \end{aligned}$$

where σ is the sigmoid function, ϕ is the hyperbolic tangent function, and $\circ$ is the element-wise multiplication; weight matrices W and U along with biases b are learnable parameters of the model. In the encoder network, the input x_t is an embedding vector of the word s_t in the NL input sequence. In the decoder network, the input x_t is an embedding vector of the previous character r_{t-1} in the output regular expression. The hidden vectors h_t of the encoder network are fed into an attention layer (Bahdanau et al., 2014) to output an overall representation of the input sentence considering the output position. The hidden vectors h_t of the decoder network are fed into a dense layer $z_t = W_z h_t$, where z_t holds the dimension of the vocabulary size of the regular expression. z_t is the output of the decoder network to predict the output character $r_t = \arg\max_j z_{t,j}$.

A softmax function is applied on z_t to obtain a probability distribution on output character candidates. The probability of character j at output position t is as follow:

$$p(r_t = j | r_{<t}, S) = \frac{e^{z_{t,j}}}{\sum_{j'} e^{z_{t,j'}}}$$

3.2 Training

Let θ represent all learnable parameters in the model. We discuss two objective functions of θ to train the model.

Maximum Likelihood Estimation (MLE). A sequence-to-sequence learning model learns the distribution of regular expressions R given an input NL sentence S:

$$p_\theta(R|S) = \prod_{t=1}^{T} p_\theta(r_t | r_{<t}, S)$$

By default, the sequence-to-sequence learning model uses maximum likelihood estimation (MLE) for training, i.e., maximizing the likelihood of mapping the input sequence to the output sequence for each pair in the training set. Specifically, the optimal parameters θ^* are obtained as follow:

$$\begin{aligned} \theta^* &= \arg\max_\theta \prod_{(S^{(i)},R^{(i)}) \in \mathcal{D}} p_\theta(R^{(i)} | S^{(i)}) \\ &= \arg\max_\theta \sum_{(S^{(i)},R^{(i)}) \in \mathcal{D}} \log p_\theta(R^{(i)} | S^{(i)}) \end{aligned}$$

Gradient descent is used to search out optimal parameters θ^*.

However, MLE fails to consider the fact that semantically equivalent regular expressions might be syntactically different. The MLE objective function forces the model to generate syntactically similar regular expressions, but penalizes semantically equivalent and syntactically different regular expressions. Such a syntax-based training objective does not fit our task's objective (i.e., generating any semantically correct regular expression).

Maximizing Semantic Correctness. To encourage the model to generate any semantically correct regular expression, we alter the MLE training objective function to maximize semantic correctness.

For an NL specification, we define a reward of a predicted regular expression $\boldsymbol{r}(R)$ as its semantic correctness (we discuss how to measure the correctness later in this section). We encourage the model to generate regular expressions to maximize expected rewards instead of MLE. Concretely, we train the model parameters θ to maximize the following objective function:

$$\begin{aligned} J(\theta) &= \sum_{(S^{(i)},R^{(i)}) \in \mathcal{D}} \mathbb{E}_{R \sim p_\theta(\cdot | S^{(i)})} \boldsymbol{r}(R) \\ &= \sum_{(S^{(i)},R^{(i)}) \in \mathcal{D}} \sum_{R} p_\theta(R | S^{(i)}) \boldsymbol{r}(R) \end{aligned}$$

However, to compute the expected reward, we need to go over all possible regular expressions, and the number of all possible regular expressions is infinite. To address this problem, we use the Monte Carlo estimate as the approximation of the expected value. Specifically, M regular expressions $R_1, \ldots, R_M$ are sampled following the output probability of the model. We average the reward of each sample to estimate the expected reward:

$$J(\theta) \approx \sum_{(S^{(i)},R^{(i)}) \in \mathcal{D}} \sum_{j=1}^{M} \frac{1}{M} \boldsymbol{r}(R_j), \quad \text{where } R_j \sim p_\theta(\cdot | S^{(i)})$$

In order to compute the gradient of the expected reward and to maximize the objective using gradient descent, we employ the REINFORCE technique of policy gradient (Williams, 1992), which is based on the following estimation:

$$\nabla_\theta J(\theta) \approx \sum_{(S^{(i)},R^{(i)}) \in \mathcal{D}} \sum_{j=1}^{M} \frac{1}{M} \boldsymbol{r}(R_j) \nabla_\theta \log p_\theta(R_j | S^{(i)}), \quad \text{where } R_j \sim p_\theta(\cdot | S^{(i)})$$

In practice, we subtract the mean reward of all samples to reduce the variance of estimated gradient (Williams, 1992). The final gradient estimate is as follow:

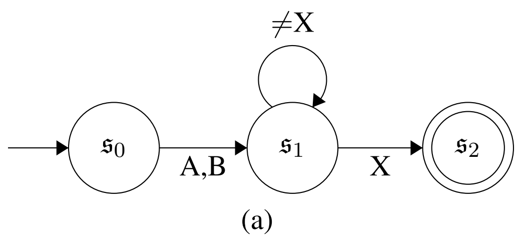
$$\nabla_\theta J(\theta) \approx \sum_{(S^{(i)},R^{(i)}) \in \mathcal{D}} \sum_{j=1}^{M} \frac{1}{M} \tilde{\boldsymbol{r}}(R_j) \nabla_\theta \log p_\theta(R_j | S^{(i)}),$$

$$\text{where } \tilde{\boldsymbol{r}}(R_j) = \boldsymbol{r}(R_j) - \sum_{j'=1}^{M} \frac{1}{M} \boldsymbol{r}(R_{j'})$$

The overall training algorithm is summarized in Algorithm 1. We initialize θ by pre-training the model using MLE on the training set. For each pair in training set, we sample M regular expressions to estimate the gradient.

Algorithm 1: Policy-gradient method to maximize semantic correctness

```
  Input: Training set: $\mathcal{D} = \{(S^{(i)}, R^{(i)})\}$
1 Initilize $\theta$ from pretrained model using
    MLE on $\mathcal{D}$ ;
2 for each epoch do
3     for $(S^{(i)}, R^{(i)}) \in \mathcal{D}$ do
4         Sample $R_1, \ldots, R_M$ using current
            model ;
5         Get rewards $\boldsymbol{r}(R_1), \ldots, \boldsymbol{r}(R_M)$ ;
6         Estimate $\nabla_\theta J(\theta)$ using (3.2) ;
7         Update $\theta$ using $\nabla_\theta J(\theta)$ by gradient
            descent ;
8     end
9 end
```



Path	String example
$\mathfrak{s}_0 \xrightarrow{A} \mathfrak{s}_1 \xrightarrow{X} \mathfrak{s}_2$	AX
$\mathfrak{s}_0 \xrightarrow{B} \mathfrak{s}_1 \xrightarrow{K} \mathfrak{s}_1 \xrightarrow{X} \mathfrak{s}_2$	BKX
$\mathfrak{s}_0 \xrightarrow{B} \mathfrak{s}_1 \xrightarrow{X} \mathfrak{s}_2$	BX

(b)

Figure 1: Minimal DFA converted from "`([ABab]&[A-Z]).*X`" and generated string examples, where $\mathfrak{s}_0$ represents the start state and $\mathfrak{s}_2$ is the only accept state.

3.3 Measurement of Semantic Correctness

In this paper, we propose two types of measuring semantic correctness based on minimal DFAs and test cases, respectively.

Minimal DFAs. We convert a regular expression to a minimal DFA and utilize the fact that equivalent regular expressions have the same minimal DFAs even when they are syntactically different (Hopcroft et al., 1979). For example, the minimal DFA of regular expression "`([ABab]&[A-Z]).*X`" is shown in Figure 1(a). A syntactically different regular expression "`((A|B).*)&(.*X)`" can be converted to the same minimal DFA as shown in Figure 1(a), indicating that these two regular expressions are semantically equivalent.

We check whether two regular expressions are equivalent by checking whether their corresponding minimal DFAs are the same. When the policy-gradient method is performed, if a sampled regular expression R is equivalent to the ground truth, then $\boldsymbol{r}(R) = 1$; otherwise $\boldsymbol{r}(R) = 0$.

Test Cases. A perfect equivalence oracle such as using the minimal DFA may not be feasibly available for some tasks, e.g., when our approach is applied on other domains such as generating Bash scripts and Python programs. To handle a more general case, we propose correctness measurement based on test cases. We generate test cases (i.e., inputs and expected outputs) and check whether a program can pass the test cases that are generated from the ground truth to approximately check whether the program and the ground truth are equivalent.

Given a regular expression R, we generate test cases that contain positive (acceptable/matched) and negative (unacceptable/unmatched) string examples. Here we consider only positive examples because negative examples can be obtained by generating positive examples of its complement regular expression $\sim R$. To generate positive string examples from regular expression R, we convert R to its corresponding minimal DFA. Each positive string example corresponds to a path from the start state of the minimal DFA to any accept state[2], and vice versa. Thus, we generate paths randomly from the start state to any accept state, and convert the paths to their corresponding strings as shown in Figure 1(b). To generate distinct string examples, we aim to generate paths to cover as many transitions as possible. In particular, we mask all transitions that have been covered by previously generated paths. When we generate a new path, the not-covered transitions have higher priority to be explored than covered ones.

Because complex regular expressions may accept/match or reject/unmatch infinite string examples, we augment random generation with a new strategy to generate distinguishing test cases to better represent the semantics. Considering that the generated test cases are used to check whether a Monte-Carlo sampled regular expression is equivalent to the ground truth in the policy-gradient method, only test cases that can differentiate an incorrect sample and the ground truth are useful. Based on such insight, we give preference to test cases that differentiate Monte-Carlo samples and the ground truth. A challenge here is that we do not know the samples before performing the policy-gradient method. However, we find that there is a high chance to get the same samples repeatedly when the model is pre-trained using MLE on the training set, because sampling is following the distribution learned by the pre-trained model. Based on the observation, we use the Beam Search algorithm on the pre-trained model to obtain B most likely samples $\hat{R}_1, \ldots, \hat{R}_B$. We generate string examples that can differentiate these samples and the ground truth, named as distinguishing string examples. For each $\hat{R}$ and ground truth R, we construct a new regular expression $R\&(\sim\hat{R})$, and generate its string examples that can differentiate R and $\hat{R}$.

[2]A DFA has one start state and a set of accept states.

The overall idea of our strategy for generating string examples is shown in Algorithm 2. Once we have a set of positive and negative string examples, we define the reward of a regular expression as $\boldsymbol{r}(R) = 1$ if it can pass all the test cases, and $\boldsymbol{r}(R) = 0$ otherwise.

When extending SemRegex on other languages where a perfect equivalence oracle is not available, it is desirable to use a technique to generate test cases for a program. There exist techniques (discussed in Section 5) to generate test cases for a general executable program.

Algorithm 2: Generating distinguishing test cases for regular expressions

```
Input: Training set: $\mathcal{D} = \{(S^{(i)}, R^{(i)})\}$,
       the number of examples to generate:
       $T$, and a pre-trained model
Output: Positive and negative example
        sets $\mathcal{P}^{(i)}$ and $\mathcal{N}^{(i)}$
1  for $(S^{(i)}, R^{(i)}) \in \mathcal{D}$ do
2      $\mathcal{P}^{(i)} \leftarrow \emptyset$ ;
3      $\mathcal{N}^{(i)} \leftarrow \emptyset$ ;
4      Beam search on pre-trained model to
        obtain $\hat{R}_1, \ldots, \hat{R}_B$ ;
5      repeat
6          Randomly pick a $j$ in $[1, B]$ ;
7          $R_p \leftarrow R^{(i)}\&(\sim\hat{R}_j)$ ;
8          $R_n \leftarrow (\sim R^{(i)})\&\hat{R}_j$ ;
9          Generate an example $p$ from $R_p$ ;
10         Generate an example $n$ from $R_n$ ;
11         $\mathcal{P}^{(i)} \leftarrow \mathcal{P}^{(i)} \cup \{p\}$ ;
12         $\mathcal{N}^{(i)} \leftarrow \mathcal{N}^{(i)} \cup \{n\}$ ;
13     until $|\mathcal{P}^{(i)}| \geq T$ && $|\mathcal{N}^{(i)}| \geq T$;
14 end
```

4 Experiments

We evaluate the effectiveness of SemRegex by comparing it to the state-of-the-art approaches. We also study how using different measurements of correctness impacts the effectiveness of SemRegex.

4.1 Experiment Setup

Datasets. We conduct our experiments on three public datasets for the task of generating regular expressions from NL specifications.

- **KB13.** KB13 (Kushman and Barzilay, 2013) includes 824 pairs of NL and regular expression. When conducting data labeling, labeling workers are asked to generate the NL specifications to capture a subset of the lines in a file. Then programmers are asked to generate regular expressions for these NL specifications written by the labeling workers. We split the data into 75% training and 25% testing sets, following what the authors of KB13 do.

- **NL-RX-Synth.** NL-RX-Synth (Locascio et al., 2016) is a synthetic dataset much larger than KB13. Its authors define a small grammar for parsing regular expressions to NL. The grammar is used to stochastically generate $10,000$ regular expressions and their corresponding synthetic NL specifications. We split the pairs into 65% training, 10% development, and 25% testing sets, following what the authors of NL-RX-Synth do.

- **NL-RX-Turk.** NL-RX-Turk (Locascio et al., 2016) comes from the NL-RX-Synth dataset. Instead of directly using synthetic NL descriptions in the dataset, the authors of NL-RX-Turk ask labeling workers to paraphrase the synthetic specifications. The dataset also consists of $10,000$ pairs of NL and regular expression. We split the pairs into 65% training, 10% development, and 25% testing sets, following what the authors of NL-RX-Turk do.

Training Setting. We use a two-layer stacked LSTM architecture in both the encoder and decoder networks. The dimensions of encoder and decoder hidden states are set to 256. We use random embedding layers with the dimension of 128 for both input and output words. We also tune our hyper-parameters on the development set. The best results are obtained when the learning rate $= 0.001$ and the batch size = 25. We use the Monte-Carlo method to sample $M = 10$ regular expressions to estimate the gradient. To generate distinguishing string examples, we perform Beam Search to obtain $B = 10$ most likely samples. Before performing the policy-gradient method, we pre-train the model using MLE for 100 epochs. Then we train the model for 40 epochs using the policy-gradient method, and choose the model with the best effectiveness on the development set. Our model is implemented in TensorFlow (Abadi et al., 2016).

4.2 Results and Analysis

Comparison Results. We demonstrate the effectiveness of our approach by comparing it to the existing approaches including Semantic-Unify (Kushman and Barzilay, 2013) and Deep-Regex(MLE) (Locascio et al., 2016). We also compare the results of our approach with different measurements of semantic correctness. Table 2 shows the comparison results of different approaches, with detailed discussion as follows.

- **Semantic-Unify.** Semantic-Unify (Kushman and Barzilay, 2013) learns to parse NL to regular expressions. Similarly, DFA equivalence is applied as a semantic unification when training the parser.

- **Deep-Regex(MLE).** Deep-Regex(MLE) (Locascio et al., 2016) regards the problem as a black-box task of machine translation without utilizing any domain knowledge of regular expressions. A syntax-based objective (MLE) is used to train the model. To the best of our knowledge, Deep-Regex(MLE) is the state-of-the-art approach on these three datasets.

- **SemRegex(DFA Oracle).** In SemRegex (DFA Oracle), we use the oracle of DFA equivalence to measure semantic correctness. SemRegex(DFA Oracle) outperforms Deep-Regex(MLE), the existing state-of-the-art approach, by an accuracy increase of 12.6% on KB13, 2.9% on NL-RX-Synth, and 4.1% on NL-RX-Turk, respectively. Compared to Deep-Regex(MLE), the results demonstrate the effectiveness of maximizing semantic correctness during the training phase. SemRegex(DFA Oracle) shows more improvement on the KB13 dataset over Deep-Regex(MLE) than on other datasets. Such result indicates that supervised learning based on MLE is less effective to learn from a small training set. When the policy-gradient method is used, Monte-Carlo samples can provide more information beyond only training samples especially on a small training set;

Table 2: Effectiveness comparison of different approaches (using DFA-equivalence accuracy as metrics)

Approach	KB13	NL-RX-Synth	NL-RX-Turk
Semantic-Unify	65.5%	46.3%	38.6%
Deep-Regex(MLE)	65.6%	88.7%	58.2%
SemRegex(DFA Oracle)	**78.2%**	**91.6%**	**62.3%**
SemRegex(Distinguishing Test Cases)	77.5%	90.2%	61.3%
SemRegex(Random Test Cases)	66.5%	90.2%	59.5%

such more information significantly improves the effectiveness.

- **SemRegex(Distinguishing Test Cases).** When we do not have access to an oracle such as DFA equivalence, we can generate test cases to define semantic correctness. SemRegex(Distinguishing Test Cases) uses Algorithm 2 to generate distinguishing test cases (10 positive examples and 10 negative examples) that differentiate the results returned by Beam Search and the ground truth. The results show that by using distinguishing test cases, SemRegex(Distinguishing Test Cases) outperforms Deep-Regex(MLE), an existing syntax-based approach, on all of three datasets. Meanwhile, the effectiveness of SemRegex(Distinguishing Test Cases) drops no more than 1.4% on accuracy compared to SemRegex(DFA Oracle). Such result indicates that limited distinguishing test cases generated by the proposed strategy can well represent the semantics.

- **SemRegex(Random Test Cases).** SemRegex(Random Test Cases) generates random test cases instead of distinguishing test cases. It outperforms the existing approaches (Semantic-Unify and Deep-Regex(MLE)) because random test cases can still represent the semantics and differentiate some inequivalent regular expressions. Compared to SemRegex(Distinguishing Test Cases), its effectiveness shows a big drop on KB13 and a slight drop on NL-RX-Turk. Such results indicate the benefit of distinguishing test cases over random test cases.

Effectiveness of Semantics-Based Objective. To understand the effect of using a semantics-based learning objective, we record the semantic accuracy (DFA equivalence) and syntactic accuracy (exact-match) on the NL-RL-Turk testing set after each epoch as shown in Figure 2. During pre-training (epochs 1 to 100), we use MLE to train the

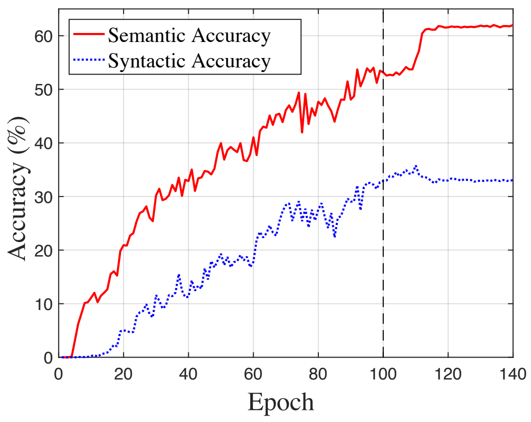


Figure 2: Semantic accuracy (DFA equivalence) and syntactic accuracy (exact-match) on the NL-RL-Turk testing set after each epoch. The training objective is replaced to maximize expected correctness after 100 epochs. The correctness is measured by the DFA-equivalence oracle in this figure.

model to increase both semantic accuracy and syntactic accuracy iteratively. Then, we alter the training objective to maximize the expected semantic correctness. We notice that while semantic accuracy continues increasing for about 10%, the syntactic accuracy does not show a significant growth after pre-training. Such result indicates that the model is no longer encouraged to generate regular expressions that are syntactically equivalent to the ground truths. Instead, the model learns to generate semantically correct regular expressions.

Analysis of Semantic Correctness Based on Test Cases. The correctness measurements based on test cases serve as an approximate oracle. Figure 3 shows an example of how the approximate oracle helps make improvement. Furthermore, we evaluate how the correctness based on test cases is close to the DFA-equivalence oracle. In Monte-Carlo estimate, we count the samples with the approximate oracle that equals to the minimal DFA oracle. When using random test cases, there are 89.8% samples with the approximate oracle that equals to the minimal DFA oracle. When using distinguish-

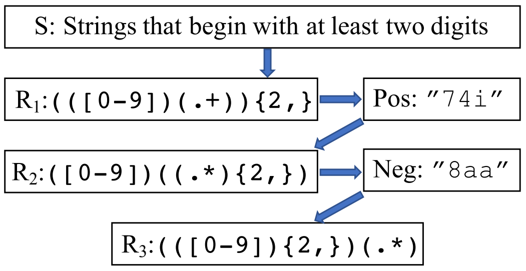


Figure 3: An example of how test cases help with training. At the beginning of the policy-gradient method, the model outputs an incorrect answer R_1, which cannot pass a positive test case. R_1 gets penalized because it receives a reward 0. Then the model changes to output an incorrect answer R_2, which cannot pass a negative test case. Similarly, R_2 gets penalized as training continues. R_3 receives a reward 1 because it passes all test cases, resulting in an increase of its likelihood from the model in iterations. Finally, the model outputs the correct answer R_3.

ing test cases, such percentage increases to 96.3%. Such result illustrates that test cases are able to approximately check the semantic equivalence even when the test cases are generated randomly. The result also suggests that distinguishing test cases represent the semantics more effectively than random test cases.

Impact of the Number of Test Cases. We study how the number of test cases impacts the effectiveness. We enumerate the number of distinguishing or random positive/negative string examples from $T = 1$ to $T = 10$ to show the impact on the effectiveness ($T = 0$ refers to using MLE to train the model). As shown in Figure 4, when more distinguishing test cases are used, higher accuracy is reached. However, more random test cases make limited improvement.

5 Related Work

Program Synthesis. Our work falls into the general topic of program synthesis. Program synthesis is the problem of automatically generating programs from high-level specifications (Gulwani et al., 2017). There has been a lot of progress made in this area, classified based on (1) the form of specifications, e.g., NL descriptions (Yin and Neubig, 2017; Guu et al., 2017; Lin et al., 2017; Krishnamurthy and Mitchell, 2012; Liu et al., 2018), input-output examples (Balog et al., 2017; Chen et al., 2018; Kalyan et al., 2018), and hybrid of the two preceding types of specifications (Manshadi

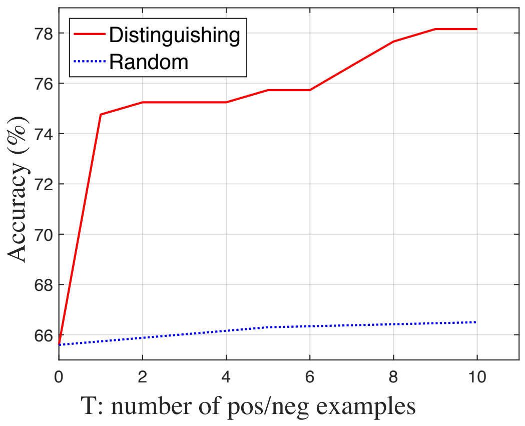


Figure 4: Impact of the number of distinguishing or random test cases on accuracy on the KB13 dataset.

et al., 2013; Raza et al., 2015); (2) the programming languages, e.g., LISP (Biermann, 1978), Python (Yin and Neubig, 2017; Rabinovich et al., 2017), SQL (Zhong et al., 2017; Sun et al., 2018), and Domain-Specific Languages (DSL) such as FlashFill (Gulwani, 2011). In this paper, we focus on an important subtask of the program-synthesis problem: generating regular expressions from NL.

Generating Regular Expressions. Recent research has attempted to automatically generate regular expressions from NL specifications. Ranta (1998) propose a rule-based approach to build an NL interface for regular expressions. Kushman and Barzilay (2013) develop an approach for learning a probabilistic grammar model to parse an NL description into a regular expression. Locascio et al. (2016) regard the problem as a black-box task of machine translation, and train a sequence-to-sequence learning model to address the problem. There exists also a lot of work focusing on generating regular expressions from string examples. Recent work typically uses an evolutionary algorithm to address the problem (Svingen, 1998; Cetinkaya, 2007; Bartoli et al., 2012, 2016).

Inspired by our previous study (Zhong et al., 2018), in this paper, we leverage the help of string examples generated from ground truths to improve the state of the art for the problem of generating regular expressions from NL. Compared with previous state-of-the-art approaches (Locascio et al., 2016) that maximize the likelihood of ground truths in the training set, SemRegex leverages the policy-gradient method to encourage the model to generate semantically correct regular expressions.

Generating Test Cases. When SemRegex is applied on domains other than synthesizing regular expressions, a perfect equivalence oracle such as using the minimal DFA may not be feasibly available. In order to handle a more general case, we propose to generate test cases from the ground truths to measure the semantic correctness of a program candidate. State-of-the-art test-generation techniques are typically based on Dynamic Symbolic Execution (DSE) (Godefroid et al., 2005). Given a program that we want to generate test cases for, DSE executes the program for some seed test cases, and at the same time collects symbolic constraints from branch statements along the execution path. Then DSE generates new test cases to cover different branches in iterations by flipping a branching node in previous execution path. In this way, DSE is able to generate test cases that can be used to approximately check the semantic equivalence. Furthermore, DSE can effectively generate distinguishing test cases for two executable programs by relating these two programs in a single execution (Taneja and Xie, 2008). Various DSE tools have been implemented for different programming languages, such as PyExZ3 (Python) (Ball and Daniel, 2015), JPF-SE (Java) (Anand et al., 2007), Pex (C#) (Tillmann and De Halleux, 2008; Tillmann et al., 2014; Li et al., 2009), and CUTE (C) (Sen et al., 2005).

6 Conclusion

We have proposed SemRegex, a semantics-based approach to generate regular expressions from NL specifications. SemRegex trains a sequence-to-sequence model by maximizing the expected semantic correctness. We measure the semantic correctness using the DFA-equivalence oracle, random test cases, and distinguishing test cases. Our evaluation results show that SemRegex outperforms the existing start-of-the-art approaches on three public datasets.

Acknowledgments

The work from the authors at the University of Illinois at Urbana-Champaign was supported in part by National Science Foundation under grants no. CNS-1513939, CNS-1564274, and CCF-1816615. The work from the authors at Xi'an Jiaotong University was supported by National Natural Science Foundation of China (61632015, 61772408, 61721002).

References

Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Gregory S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian J. Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Józefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Gordon Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul A. Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda B. Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: Large-scale machine learning on heterogeneous distributed systems. *CoRR*, abs/1603.04467.

Saswat Anand, Corina S Păsăreanu, and Willem Visser. 2007. JPF–SE: A symbolic execution extension to Java PathFinder. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 134–138.

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2014. Neural machine translation by jointly learning to align and translate. In *International Conference on Learning Representations*.

Thomas Ball and Jakub Daniel. 2015. Deconstructing dynamic symbolic execution. Technical Report MSR-TR-2015-95, Microsoft.

Matej Balog, Alexander L Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. 2017. DeepCoder: Learning to write programs. In *International Conference on Learning Representations*.

Alberto Bartoli, Giorgio Davanzo, Andrea De Lorenzo, Marco Mauri, Eric Medvet, and Enrico Sorio. 2012. Automatic generation of regular expressions from examples with genetic programming. In *Annual Conference Companion on Genetic and Evolutionary Computation*, pages 1477–1478.

Alberto Bartoli, Andrea De Lorenzo, Eric Medvet, and Fabiano Tarlao. 2016. Inference of regular expressions for text extraction from examples. *IEEE Transactions on Knowledge and Data Engineering*, 28(5):1217–1230.

Alan W Biermann. 1978. The inference of regular LISP programs from examples. *IEEE Transactions on Systems, Man, and Cybernetics*, 8(8):585–600.

Rudy Bunel, Matthew Hausknecht, Jacob Devlin, Rishabh Singh, and Pushmeet Kohli. 2018. Leveraging grammar and reinforcement learning for neural program synthesis. In *International Conference on Learning Representations*.

Ahmet Cetinkaya. 2007. Regular expression generation through grammatical evolution. In *Annual Conference Companion on Genetic and Evolutionary Computation*, pages 2643–2646.

Xinyun Chen, Chang Liu, and Dawn Song. 2018. Towards synthesizing complex programs from input-output examples. In *International Conference on Learning Representations*.

Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: directed automated random testing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 213–223.

Sumit Gulwani. 2011. Automating string processing in spreadsheets using input-output examples. In *ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 317–330.

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

Kelvin Guu, Panupong Pasupat, Evan Zheran Liu, and Percy Liang. 2017. From language to programs: Bridging reinforcement learning and maximum marginal likelihood. In *Annual Meeting of the Association for Computational Linguistics*, pages 1051–1062.

Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural Computation*, 9(8):1735–1780.

John E Hopcroft, Rajeev Motwani, and Jeffrey D Ullman. 1979. *Introduction to automata theory, languages, and computation*. Addison-wesley Reading.

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

Jayant Krishnamurthy and Tom M Mitchell. 2012. Weakly supervised training of semantic parsers. In *Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*, pages 754–765.

Nate Kushman and Regina Barzilay. 2013. Using semantic unification to generate regular expressions from natural language. In *Conference of the North American Chapter of the Association for Computational Linguistics*, pages 826–836.

Nuo Li, Tao Xie, Nikolai Tillmann, Jonathan de Halleux, and Wolfram Schulte. 2009. Reggae: Automated test generation for programs using complex regular expressions. In *IEEE/ACM International Conference on Automated Software Engineering*.

Xi Victoria Lin, Chenglong Wang, Deric Pang, Kevin Vu, Luke Zettlemoyer, and Michael D. Ernst. 2017. Program synthesis from natural language using recurrent neural networks. Technical Report UW-CSE-17-03-01, University of Washington Department of Computer Science and Engineering.

Evan Zheran Liu, Kelvin Guu, Panupong Pasupat, and Percy Liang. 2018. Reinforcement learning on web interfaces using workflow-guided exploration. In *International Conference on Learning Representations*.

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 *Conference on Empirical Methods in Natural Language Processing*, pages 1918–1923.

Mehdi Hafezi Manshadi, Daniel Gildea, and James F Allen. 2013. Integrating programming by example and natural language programming. In *AAAI Conference on Artificial Intelligence*, pages 661–667.

Maxim Rabinovich, Mitchell Stern, and Dan Klein. 2017. Abstract syntax networks for code generation and semantic parsing. In *Annual Meeting of the Association for Computational Linguistics*, pages 1139–1149.

Aarne Ranta. 1998. A multilingual natural-language interface to regular expressions. In *International Workshop on Finite State Methods in Natural Language Processing*, pages 79–90.

Mohammad Raza, Sumit Gulwani, and Natasa Milic-Frayling. 2015. Compositional program synthesis from natural language and examples. In *International Joint Conferences on Artificial Intelligence*, pages 792–800.

Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: a concolic unit testing engine for C. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 263–272.

Yibo Sun, Duyu Tang, Nan Duan, Jianshu Ji, Guihong Cao, Xiaocheng Feng, Bing Qin, Ting Liu, and Ming Zhou. 2018. Semantic parsing with syntax- and table-aware SQL generation. In *Annual Meeting of the Association for Computational Linguistics*, pages 361–372.

Ilya Sutskever, Oriol Vinyals, and Quoc V Le. 2014. Sequence to sequence learning with neural networks. In *Annual Conference on Neural Information Processing Systems*, pages 3104–3112.

Borge Svingen. 1998. Learning regular languages using genetic programming. *Genetic Programming*, pages 374–376.

Kunal Taneja and Tao Xie. 2008. DiffGen: Automated regression unit-test generation. In *IEEE/ACM International Conference on Automated Software Engineering*, pages 407–410.

Nikolai Tillmann and Jonathan De Halleux. 2008. Pex–White box test generation for .NET. In *International Conference on Tests and Proofs*, pages 134–153.

Nikolai Tillmann, Jonathan de Halleux, and Tao Xie. 2014. Transferring an automated test generation tool to practice: From Pex to Fakes and Code Digger. In *IEEE/ACM International Conference on Automated Software Engineering*, pages 385–396.

Ronald J Williams. 1992. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8:229–256.

Pengcheng Yin and Graham Neubig. 2017. A syntactic neural model for general-purpose code generation. In *Annual Meeting of the Association for Computational Linguistics*, pages 440–450.

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

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 *Workshop on NLP for Software Engineering*.