# Write, Execute, Assess: Program Synthesis with a REPL

**Kevin Ellis** [*1] **Maxwell Nye** [*1] **Yewen Pu** [*1] **Felix Sosa** [*2] **Joshua B. Tenenbaum** [1] **Armando Solar-Lezama** [1]

## Abstract

We present a neural program synthesis approach integrating components which write, execute, and assess code to navigate the search space of possible programs. We equip the search process with an interpreter or a read-eval-print-loop (REPL), which immediately executes partially written programs, exposing their semantics. The REPL addresses a basic challenge of program synthesis: tiny changes in syntax can lead to huge changes in semantics. We train a pair of models, a policy that proposes the new piece of code to write, and a value function that assesses the prospects of the code written so-far. At test time we can combine these models with a Sequential Monte Carlo algorithm. We apply our approach to two domains: synthesizing text editing programs and inferring 2D and 3D graphics programs.

## 1. Introduction

When was the last time you typed out a large body of code all at once, and had it work on the first try? If you are like most programmers, this hasn't happened much since "Hello, World." Writing a large body of code is a process of *trial and error* that alternates between trying out a piece of code, executing it to see if you're still on the right track, and trying out something different if the current execution looks buggy. Crucial to this human work-flow is the ability to *execute* the partially-written code, and the ability to *assess* the resulting execution to see if one should continue with the current approach. Thus, if we wish to build machines that automatically write large, complex programs, designing systems with the ability to effectively transition between states of writing, executing, and assessing the code may prove crucial.

[*]Equal contribution. Listed alphabetically. [1]Massachusetts Institute of Technology [2]Harvard University. Correspondence to: Kevin Ellis <ellisk@mit.edu>, Maxwell Nye <mnye@mit.edu>, Yewen Pu <yewenpu@mit.edu>, Felix Sosa <fsosa@fas.harvard.edu>.

*Proceedings of the $36^{th}$ International Conference on Machine Learning*, Long Beach, California, PMLR 97, 2019. 

In this work, we present a model that integrates components which write, execute, and assess code to perform a stochastic search over the *semantic* space of possible programs. We do this by equipping our model with one of the oldest and most basic tools available to a programmer: an interpreter, or read-eval-print-loop (REPL), which immediately executes partially written programs, exposing their semantics. The REPL addresses a fundamental challenge of program synthesis: tiny changes in syntax can lead to huge changes in semantics. By conditioning the search solely on the execution states rather than the program syntax, the search is performed entirely in the *semantic* space.

In the spirit of systems such as AlphaGo (1), we train a pair of models – a *policy* that proposes new pieces of code to write, and a *value function* that evaluates the long-term prospects of the code written so far, and deploy both at test time in a symbolic tree search. Specifically, we combine the policy, value, and REPL with a Sequential Monte Carlo (SMC) search strategy at inference time. We sample next actions using our learned policy, execute the partial programs with the REPL, and re-weight the candidates by the value of the resulting partial program state. This algorithm allows us to naturally incorporate writing, executing, and assessing partial programs into our search strategy, while managing a large space of alternative program candidates.

Integrating learning and search to tackle the problem of program synthesis is an old idea experiencing a recent resurgence (2; 3; 4; 5; 6; 7; 8; 9; 10; 11). Our work builds on recent ideas termed 'execution-guided neural program synthesis,' independently proposed by (12) and (13), where a neural network writes a program conditioned on intermediate execution states. We extend these ideas along two dimensions. First, we cast these different kinds of execution guidance in terms of interaction with a REPL, and use reinforcement learning techniques to train an agent to both interact with a REPL, *and* to assess when it is on the right track. Prior execution-guided neural synthesizers do not learn to *assess* the execution state, which is a prerequisite for sophisticated search algorithms, like those we explore in this work. Second, we investigate several ways of interleaving the policy and value networks during search, finding that an SMC sampler provides an effective foundation for an agent that writes, executes and assesses its code. We validate our framework on two different domains (see Figure 1):

inferring 2D and 3D graphics programs (in the style of computer aided design, or CAD) and synthesizing text-editing programs (in the style of FlashFill (14)).

## 2. An Illustrative Example

To make our framework concrete, consider the following program synthesis task of synthesizing a constructive solid geometry (CSG) representation of a simple 2D scene (see Figure 2). CSG is a shape-modeling language that allows the user to create complex renders by combining simple primitive shapes via boolean operators. The CSG program in our example consists of two boolean combinations: union $+$ and subtraction $-$ and two primitives: circles $C^r_{x,y}$ and rectangles $R^{w,h,\theta}_{x,y}$, specified by position $x, y$, radius $r$, width and height $w, h$, and rotation $\theta$. The synthesis task is to find a CSG program that renders to $spec$. Our policy constructs this program one piece at a time, conditioned on the set of expressions currently in scope. Starting with an empty *set* of programs in scope, $pp = \{\}$, the policy proposes an action $a$ that extends it. This proposal process is iterated to incrementally extend $pp$ to contain longer and more complex programs. In this CSG example, the action $a$ is either adding a primitive shape, such as a circle $C^3_{2,8}$, or applying a boolean combinator, such as $p_1 - p_2$, where the action also specifies its two arguments $p_1$ and $p_2$.

To help the policy make good proposals, we augment it with a REPL, which takes a set of programs $pp$ in scope and executes each of them. In our CSG example, the REPL renders the set of programs $pp$ to a set of images. The policy then takes in the REPL state (a set of images), along with the specification $spec$ to predict the next action $a$. This way, the input to the policy lies entirely in the semantic space, akin to how one would use a REPL to iteratively construct a working code snippet. Figure 2 demonstrates a potential roll-out through a CSG problem using only the policy.

However, code is brittle, and if the policy predicts an incorrect action, the entire program synthesis fails. To combat this brittleness, we use Sequential Monte Carlo (SMC) to search over the space of candidate programs. Crucial to our SMC algorithm is a learned value function $v$ which, given a REPL state, assesses the likelihood of success on this particular search branch. By employing $v$, the search can be judicious about which search branch to prioritize in exploring and withdraw from branches deemed unpromising. Figure 3 demonstrates a fraction of the search space leading up to the successful program and how the value function $v$ helps to prune out unpromising search candidates.

## 3. Our Approach

### 3.1. The Semantic Search Space of Programs

The space of possible programs is typically defined by a context free grammar (CFG), which specifies the set of syntactically valid programs. However, when one is writing the code, the programs are often constructed in a piece-wise fashion. Thus, it is natural to express the search space of programs as a markov decision process (MDP) over the set of partially constructed programs.

**State** The state is a tuple $s = (pp, spec)$ where $pp$ is a *set* of partially-constructed program trees (intuitively, 'variables in scope'), and $spec$ is the goal specification. Thus, our MDP is goal conditioned. The start state is $(\{\}, spec)$.

**Action** The action $a$ is a production rule from the CFG (a line of code typed into the REPL).

**Transitions** The transition, $T$, takes the set of partial programs $pp$ and applies the action $a$ to either:
1. instantiate a new sub-tree if $a$ is a terminal production: $T(pp, a) = pp \cup \{a\}$
2. combine multiple sub-trees if $a$ is a non-terminal: $T(pp, a) = (pp \cup \{a(t_1 \dots t_k)\}) - \{t_1 \dots t_k\}$

Note that in the case of a non-terminal, the children $t_1 \dots t_k$ are removed, or 'garbage-collected' (13).

**Reward** The reward is 1 if there is a program $p \in pp$ that satisfies the spec, and 0 otherwise.

Note that the state of our MDP is defined jointly in the syntactic space, $pp$, and the semantic space, $spec$. To bridge this gap, we use a REPL, which evaluates the set of partial programs $pp$ into a semantic or "executed" representation. Let $pp$ be a set of $n$ programs, $pp = \{p_1 \dots p_n\}$ and let $[\![p]\!]$ denote the execution of a program $p$, then we can write the REPL state as $[\![pp]\!] = \{[\![p_1]\!] \dots [\![p_n]\!]\}$.

### 3.2. Training the Code-Writing Policy $\pi$ and the Code-Assessing Value $v$

Given the pair of evaluated program states and spec ($[\![pp]\!]$, $spec$), the policy $\pi$ outputs a distribution over actions, written $\pi(a \mid [\![pp]\!], spec)$, and the value function $v$ predicts the expected reward starting from state ($[\![pp]\!]$, $spec$).

**Pretraining $\pi$.** Because we assume the existence of a CFG and a REPL, we can generate an infinite stream of training data by sampling random programs from the CFG, executing them to obtain a spec, and then recovering the ground-truth action sequence. Specifically, we draw samples from a distribution over synthetic training data, $\mathcal{D}$, consisting of triples of the spec, the sequence of actions, and the set of partially constructed programs at each step:

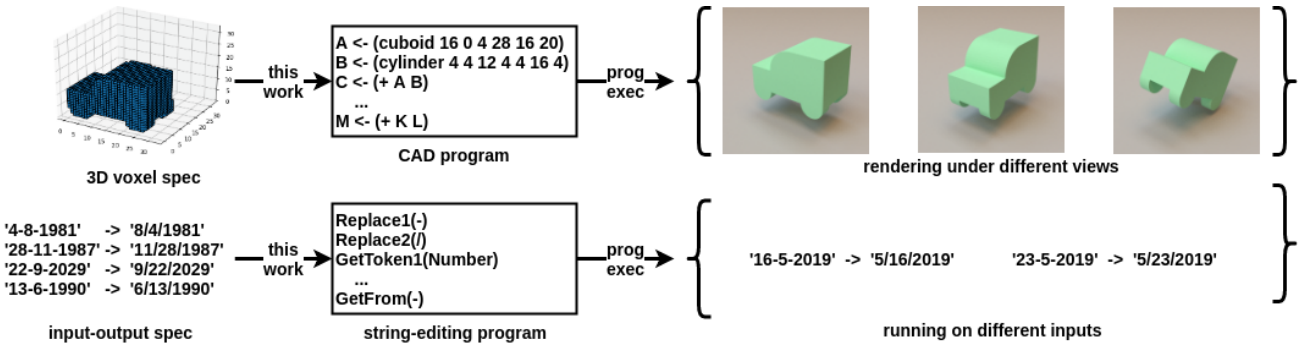


*Figure 1.* Examples of programs synthesized by our system. Top, graphics program from voxel specification. Bottom, string editing program from input-output specification.

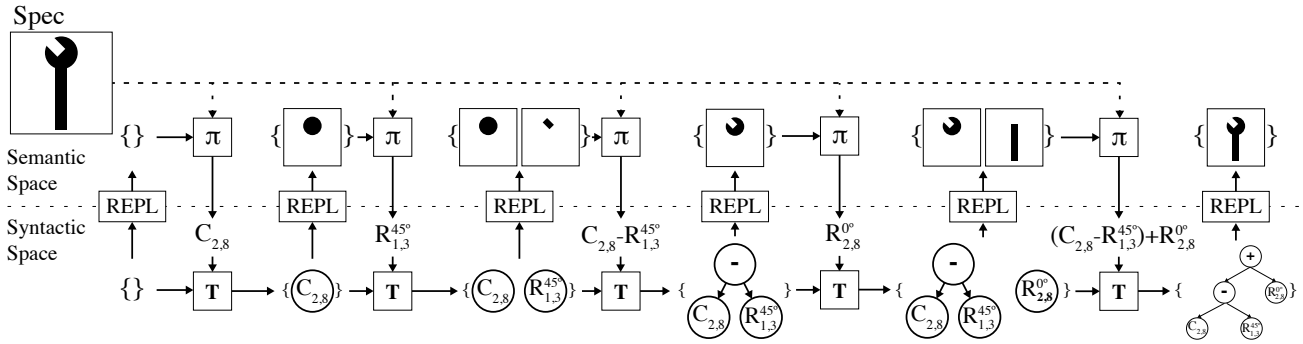


*Figure 2.* A particular trajectory of the policy building a 2D wrench. At each step, the REPL renders the set of partial programs *pp* into the semantic (image) space. These images are fed into the policy $\pi$ which proposes how to extend the program via an action $a$, which is incorporated into *pp* via the transition $T$.

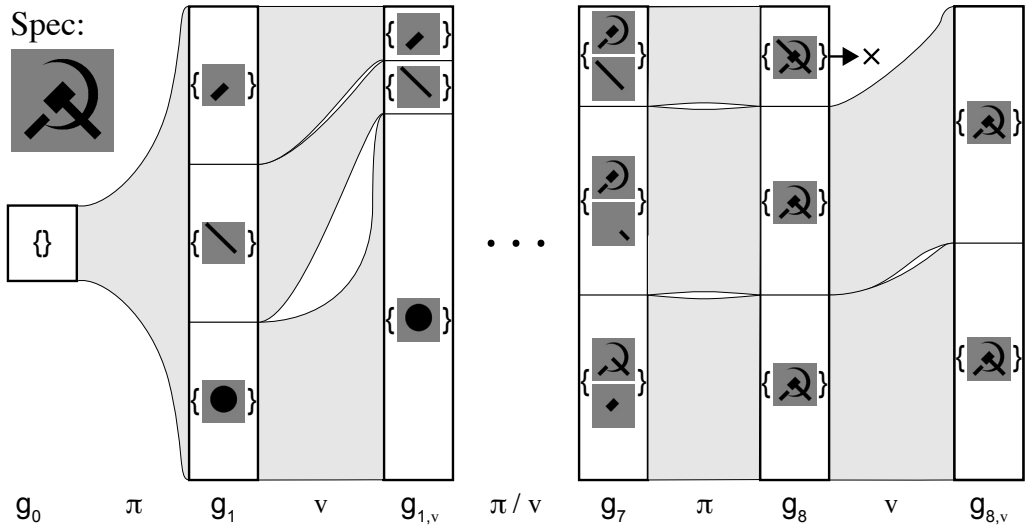


*Figure 3.* SMC sampler maintains a population of particles (i.e. programs), which it evolves forward in time by (1) sampling from policy $\pi$ to get a new generation of particles, then (2) reweighting by the value function $v$ and resampling to prune unpromising candidates and up-weight promising ones.

$(spec, \{a_t\}_{t\leq T}, \{pp_t\}_{t\leq T}) \sim \mathcal{D}$, and maximize:

$$\mathcal{L}^{\text{pretrain}}(\pi) = \mathbb{E}_{\mathcal{D}}\left[\sum_{t\leq T} \log \pi(a_t|[\![pp_t]\!], spec)\right] \quad (1)$$

**Training $\pi$ and $v$.** We fine-tune the policy and train the value function by sampling the policy's roll-outs against $spec \sim \mathcal{D}$ in the style of REINFORCE. Specifically, given $spec \sim \mathcal{D}$ and reward $R$, we train $v$ and $\pi$ to maximize:

$$\begin{aligned}\mathcal{L}^{\text{RL}}(v, \pi) = \; & R\sum_{t\leq T} \log v([\![pp_t]\!], spec) \\ & +(1-R)\sum_{t\leq T} \log(1 - v([\![pp_t]\!], spec)) \\ & +R\sum_{t\leq T} \log \pi(a_t|[\![pp_t]\!], spec) \end{aligned} \quad (2)$$

### 3.3. An SMC Inference Algorithm That Interleaves Writing, Executing, and Assessing Code

At test time we interleave *code writing*, i.e. drawing actions from the policy, and *code assessing*, i.e. querying the value function (and thus also interleave execution, which always occurs before running these networks). We implement this by constructing a Sequential Monte Carlo sampler (15) that proposes search moves using $\pi$, and then reweights and resamples in proportion to $v$. SMC techniques are not the only reasonable approach: one could perform a beam search, seeking to maximize $\log \pi(\{a_t\}_{t<T}|spec) + \log v([\![pp_T]\!], spec)$; or, A* search by interpreting $-\log \pi(\{a_t\}_{t<T}|spec)$ as cost-so-far and $-\log v([\![pp_T]\!], spec)$ as heuristic cost-to-go. SMC confers two main benefits: (1) it is a stochastic search procedure, immediately yielding a simple any-time algorithm where we repeatedly run the sampler and keep the best program found so far; and (2) the sampling/resampling steps are easily batched on a GPU, giving high throughput unattainable with serial algorithms like A*.

## 4. Experiments

To assess the relative importance of the policy, value function, and REPL, we study a spectrum of models and test-time inference strategies in both of our domains. For each model and inference strategy, we are interested in how efficiently it can search the space of programs, i.e. the best program found as a function of time spent searching. We trained a pair of models: our REPL model, which conditions on intermediate execution states (architectures in appendix), and a 'no REPL' baseline, which decodes a program in one shot using only the spec and syntax. This baseline is inspired by the prior work CSGNet (16) and RobustFill (8) for CAD and string editing, respectively.

### 4.1. Inverse CAD

Modern mechanical parts are created using Computer Aided Design (CAD), a family of programmatic shape-modeling techniques. Here we consider two varieties of *inverse* CAD: inferring programs generating 3D shapes, and programs generating 2D graphics. We use CSG as our CAD modeling language, and the the goal is to write a program that renders to the target image by algebraically combining parametric primitive drawing commands via addition and subtraction. Our REPL renders each partial program $p \in pp$ to a distinct canvas, which the policy and value networks take as input.

**Experimental evaluation** We train our models on randomly generated scenes with up to 13 objects. Figure 4 (bottom) measures the quality of the best program found so far as a function of time, where we measure the quality of a program by the intersection-over-union (IoU) with the spec. Incorporating the value function proves important for both beam search and sampling methods such as SMC. Given a large enough time budget the 'no REPL' baseline is competitive with our ablated alternatives: inference time is dominated by CNN evaluations, which occur at every step with a REPL, but only once without it. Qualitatively, an integrated policy, value network, and REPL yield programs closely matching the spec (Figure 4, top). Together these components allow us to infer very long programs, despite a branching factor of ≈1.3 million per line of code: the largest programs we successfully infergo up to 19 lines of code/102 tokens for 3D and 22 lines/107 tokens for 2D, but the best-performing ablations fail to scale beyond 3 lines/19 tokens for 3D and 19 lines/87 tokens for 2D.

### 4.2. String Editing Programs

Learning programs that transform text is a classic program synthesis task (17) made famous by the FlashFill system, which ships in Microsoft Excel (14). We apply our framework to string editing programs using the RobustFill programming language (8), which was designed for neural program synthesizers. Our formulation suggests modifications to the RobustFill language so that partial programs can be evaluated into a semantically coherent state (i.e. they execute and output something meaningful). Along with edits to the original language, we designed and implemented a REPL, which, in addition to the original inputs and outputs, includes additional features of the intermediate program state, described in the appendix.

**Experimental Evaluation** We trained our model and a reimplementation of RobustFill on string editing programs randomly sampled from the CFG. We originally tested on string editing programs from (6) (comprising training tasks from (4) and the test corpus from (18)), but found performance was near ceiling for our model. We designed a more difficult dataset of 87 string editing problems from 34

templates comprising address, date/time, name, and movie review transformations. This dataset required synthesis of long and complex programs, making it harder for pure neural approaches such as RobustFill.

The performance of our model and baselines is plotted in Figure 5 (bottom), and examples of best performing programs are shown in Figure 5 (top). The value-guided SMC sampler leads to the highest overall number of correct programs, requiring less time and fewer nodes expanded compared to other inference techniques. We also observe that beam search attains higher overall performance with the value function than beam search without value. Our model demonstrates strong out-of sample generalization: Although it was trained on programs whose maximum length was 30 actions and average length approximately 8 actions, during test time we regularly achieved programs with 40 actions or more, representing a recovery of programs with description length greater than 350 bits.

## 5. Discussion

**Related Work** Within the program synthesis community, both text processing and graphics program synthesis have received considerable attention (14).We are motivated by works such as InverseCSG (19), CSGNet (16), and RobustFill (8), but our goal is not to solve a specific synthesis problem in isolation, but rather to push toward more general frameworks that demonstrate robustness across domains.

We draw inspiration from recent neural "execution-guided synthesis" approaches (13; 12) which leverage partial evaluations of programs, similar to our use of a REPL. We build on this line of work by explicitly formalizing the task as an MDP, which exposes a range of techniques from the RL and planning literatures. Our addition of a learned value network demonstrates marked improvements on methods that do not leverage such learned value networks. Prior work (20) combines tree search with $Q$-learning to synthesize small assembly programs, but do not scale to large programs with extremely high branching factors, as we do (e.g., the $> 40$ action-programs we synthesize for text editing or the $>1.3$ million branching factor per line of code in our 3D inverse CAD).

**Outlook** We have presented a framework for learning to write code combining two ideas: allowing the agent to explore a tree of possible solutions, and executing and assessing code as it gets written. This has largely been inspired by previous work on execution-guided program synthesis, value-guided tree search, and general behavior observed in how people write code.

An immediate future direction is to investigate programs with control flow like conditionals and loops. A Forth-style stack-based (21) language offer promising REPL-like representations of these control flow operators. But more broadly, we are optimistic that many tools used by human programmers, like debuggers and profilers, can be reinterpreted and repurposed as modules of a program synthesis system. By integrating these tools into program synthesis systems, we believe we can design systems that write code more robustly and rapidly like people.

### ACKNOWLEDGMENTS

We gratefully acknowledge many extended and productive conversations with Tao Du, Wojciech Matusik, and Siemens research. In addition to these conversations, Tao Du assisted by providing 3D ray tracing code, which we used when rerendering 3D programs. Work was supported by Siemens research, AFOSR award FA9550-16-1-0012 and the MIT-IBM Watson AI Lab. K. E. and M. N. are additionally supported by NSF graduate fellowships.

## References

[1] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484, 2016.

[2] Yaroslav Ganin, Tejas Kulkarni, Igor Babuschkin, SM Eslami, and Oriol Vinyals. Synthesizing programs for images using reinforced adversarial learning. *arXiv preprint arXiv:1804.01118*, 2018.

[3] Jürgen Schmidhuber. Optimal ordered problem solver. *Machine Learning*, 54(3):211–254, 2004.

[4] Kevin Ellis, Lucas Morales, Mathias Sablé-Meyer, Armando Solar-Lezama, and Josh Tenenbaum. Learning libraries of subroutines for neurally–guided bayesian program induction. In *Advances in Neural Information Processing Systems*, pages 7805–7815, 2018.

[5] Matej Balog, Alexander L Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. Deepcoder: Learning to write programs. *arXiv preprint arXiv:1611.01989*, 2016.

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

[7] Kevin Ellis, Daniel Ritchie, Armando Solar-Lezama, and Josh Tenenbaum. Learning to infer graphics programs from hand-drawn images. In *Advances in Neural Information Processing Systems*, pages 6059–6068, 2018.

[8] Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. Robustfill: Neural program learning under noisy i/o. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 990–998. JMLR. org, 2017.

[9] Ashwin Kalyan, Abhishek Mohta, Oleksandr Polozov, Dhruv Batra, Prateek Jain, and Sumit Gulwani. Neural-guided deductive search for real-time program synthesis from examples. *arXiv preprint arXiv:1804.01186*, 2018.

[10] Yonglong Tian, Andrew Luo, Xingyuan Sun, Kevin Ellis, William T Freeman, Joshua B Tenenbaum, and Jiajun Wu. Learning to infer and execute 3d shape programs. *arXiv preprint arXiv:1901.02875*, 2019.

[11] Yewen Pu, Zachery Miranda, Armando Solar-Lezama, and Leslie Kaelbling. Selecting representative examples for program synthesis. In *International Conference on Machine Learning*, pages 4158–4167, 2018.

[12] Xinyun Chen, Chang Liu, and Dawn Song. Execution-guided neural program synthesis. 2018.

[13] Amit Zohar and Lior Wolf. Automatic program synthesis of long programs with a learned garbage collector. In *Advances in Neural Information Processing Systems*, pages 2094–2103, 2018.

[14] Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In *ACM Sigplan Notices*, volume 46, pages 317–330. ACM, 2011.

[15] Arnaud Doucet, Nando De Freitas, and Neil Gordon. An introduction to sequential monte carlo methods. In *Sequential Monte Carlo methods in practice*, pages 3–14. Springer, 2001.

[16] Gopal Sharma, Rishabh Goyal, Difan Liu, Evangelos Kalogerakis, and Subhransu Maji. Csgnet: Neural shape parser for constructive solid geometry. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 5515–5523, 2018.

[17] Tessa Lau. *Programming by demonstration: a machine learning approach*. PhD thesis, 2001.

[18] Rajeev Alur, Dana Fisman, Rishabh Singh, and Armando Solar-Lezama. Sygus-comp 2016: Results and analysis. *arXiv preprint arXiv:1611.07627*, 2016.

[19] Tao Du, Jeevana Priya Inala, Yewen Pu, Andrew Spielberg, Adriana Schulz, Daniela Rus, Armando Solar-Lezama, and Wojciech Matusik. Inversecsg: Automatic conversion of 3d models to csg trees. *ACM Trans. Graph.*, 37(6):213:1–213:16, December 2018.

[20] Riley Simmons-Edler, Anders Miltner, and Sebastian Seung. Program synthesis through reinforcement learning guided tree search. *arXiv preprint arXiv:1806.02932*, 2018.

[21] Elizabeth D Rather, Donald R Colburn, and Charles H Moore. The evolution of forth. In *ACM SIGPLAN Notices*, volume 28, pages 177–199. ACM, 1993.

[22] Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. Pointer networks. In *Advances in Neural Information Processing Systems*, pages 2692–2700, 2015.

[23] Manzil Zaheer, Satwik Kottur, Siamak Ravanbakhsh, Barnabas Poczos, Ruslan R Salakhutdinov, and Alexander J Smola. Deep sets. In *Advances in neural information processing systems*, pages 3391–3401, 2017.

# A. Appendix

## A.1. Neural network architectures

## A.2. Graphics Programming Language

**CFG** The programs are generated by the CFG below:

```
P -> P + P | P - P | S

# For 2D graphics
S -> circle(radius=N, x=N, y=N)
   | quadrilateral(x0=N, y0=N,
                   x1=N, y1=N,
                   x2=N, y2=N,
                   x3=N, y3=N)

N -> [0 : 31 : 2]

# For 3D graphics
S -> sphere(radius=N, x=N, y=N, z=N)
   | cube(x0=N, y0=N, z0=N,
          x1=N, y1=N, z1=N)
   | cylinder(x0=N, y0=N, z0=N,
              x1=N, y1=N, z1=N, radius=N)
N -> [0 : 31 : 4]
```

In principle the 2D language admits arbitrary quadrilaterals. When generating synthetic training data we constrain the quadrilaterals to be take the form of rectangles rotated by 45 increments, although in principle one could permit arbitrary rotations by simply training a higher capacity network on more examples.

## A.3. String Editing Programming Language

**CFG** Our modified string editing language, based on (8) is defined as follows:

```
Program P -> concat(E,.., E) (at most 6 E's)
Expression E -> F | N | N(N)
              | N(F) | Const
Substring F -> Sub1(k) Sub2(k)
               | Span1(r) Span2(i) Span3(y)
                 Span4(r) Span5(i) Span6(y)
Nesting N -> GetToken1(t) GetToken2(i)
             | ToCase(s) | GetUpTo(r)
             | GetFrom(r) | GetAll(t)
             | GetFirst1(t) GetFirst2(i)
Regex r -> t | d
Type t -> Number | Word | AlphaNum
             | Digit | Char | AllCaps
             | Proper | Lower
Case s -> AllCaps | PropCase | Lower
Delimiter d ->  &,.?!@()[]%{}/#$:;"'
Index i -> {-5 .. 6}
Boundary y -> Start | End
```

**REPL** Our read-eval-print-loop exposes the following intermediate program states: The **committed** string maintains, for each example input, the output of the expressions synthesized so far. The **scratch** string maintains, for each example input, the partial results of the expression currently being synthesized until it is complete and ready to be added to the *committed* string. Finally, the binary valued **mask** features indicate, for each character position, the possible locations on which transformations can occur.

### A.4. Training details

**String Editing** We performed supervised pretraining for 24000 iterations with a batch size of 4000. We then performed REINFORCE for 12000 epochs with a batch size of 2000. Training took approximately two days with one p100 GPU. We use the Adam optimizer wiht default settings.

Our Robustfill baseline was a re-implementation of the "Attn-A" model from (8). We implemented the "DP-beam" feature, wherein during test-time beam search, partial programs which lead to an output string which is not a prefix of the desired output string are removed from the beam. We trained for 50000 iterations with a batch size of 32. Training also took approximately two days with one p100 GPU.

**2D/3D Graphics** We performed supervised pretraining with a batch size of 32, training on a random stream of CSG programs with up to 13 shapes, for approximately three days with one p100 GPU. We use the Adam optimizer wiht default settings. Over three days the 3D model saw approximately 1.7 million examples and the 2D model saw approximately 5 million examples. We fine-tuned the policy using REINFORCE and trained the value network for approximately 5 days on one p100 GPU. For each gradient step during this process we sampled $B_1 = 2$ random programs and performed $B_2 = 16$ rollouts for a total batch size of $B = B_1 \times B_2 = 32$. During reinforcement learning the 3D model saw approximately 0.25 million examples and the 2D model saw approximately 9 million examples.

For both domains, we performed no hyperparameter search. We expect that with some tuning, results could be marginally improved, but our goal is to design a general approach which is not sensitive to fine architectural details.

### A.5. Data and test-time details

For both domains, we used a 2-minute timeout for testing for each problem, and repeatedly doubled the beam/number of particles until timeout is reached.

**String Editing** We originally tested on string editing programs from (6) (comprising training tasks from (4) and the test corpus from (18)), but found our performance was near ceiling for our model (Figure 8). Thus, we designed our own dataset, as described in the main text. Generation code for this dataset can be found in our supplement, in the file `generate_test_robust.py`.

**2D/3D Graphics** We generate a scene with up to $k$ objects by sampling a number between 1 to $k$, and then sampling a random CSG tree with that many objects. We then remove any subtrees that do not affect the final render (e.g., subtracting pixels from empty space). Our held-out test set is created by sampling 30 random scenes with up to $k = 20$ objects for 3D and $k = 30$ objects for 2D. Running `python driver.py demo -maxShapes 30` using the attached supplemental source code will generate example random scenes. Figure 7 illustrates ten random 3-D/2-D scenes and contrasts different model outputs.

### A.6. Architecture details

#### A.6.1. String Editing

For this domain, our neural architecture involves encoding each example state separately and pooling into a hidden state, which is used to decode the next action. To encode each example, we learn an embedding vector of size 20 for each character and apply it to each position in the input string, output string, committed string, and scratch string. For each character position, we concatenate these embedding vectors, additionally concatenating the values of the masks for that spatial position. We then perform a 1-d convolution with kernel size 5 across the character positions. Following (13), we concatenate the vectors for all the character positions, and pass this through a dense block with 10 layers and a growth rate of 128 to produce a hidden vector for a single example. We perform an average pooling on the hidden vector for each example. We then concatenate the

resulting vector with a 32-dim embedding of the previous action and apply a linear layer, which results in the final state embedding, from which we decode the next action. Our value network is identical, except the final layer instead decodes a value.

#### A.6.2. INVERSE CAD

The policy is a CNN followed by a pointer network which decodes into the next line of code. A pointer network (22) is an RNN that uses a differentiable attention mechanism to emit pointers, or indices, into a set of objects. Here the pointers index into the set of partial programs $pp$ in the current state, which is necessary for the union and difference operators. Because the CNN takes as input the current REPL state – which may have a variable number of objects in scope – we encode each object with a separate CNN and sum their activations, i.e. a 'Deep Set' encoding (23). The value function is an additional 'head' to the pooled CNN activations.

Concretely the neural architecture has a *spec encoder*, which is a CNN inputting a single image, as well as a *canvas encoder*, which is a CNN inputting a single canvas in the REPL state, alongside the spec, as a two-channel image. The canvas encoder output activations are summed and concatenated with the spec encoder output activations to give an embedding of the state:

$$\text{stateEncoding}(spec, pp) = \text{specEncoder}(spec) \otimes \sum_{p \in pp} \text{canvasEncoder(spec,}[\![\text{p}]\!]) \tag{3}$$

for $W_1$, $W_2$ weight matrices.

For the policy we pass the state encoding to a pointer network, implemented using a GRU with 512 hidden units and one layer, which predicts the next line of code. To attend to canvases $p \in pp$, we use the output of the canvas encoder as the 'key' for the attention mechanism.

For the value function we passed the state in coding to a MLP with 512 hidden units w/ a hidden ReLU activation, and finally apply a negated 'soft plus' activation to the output to ensure that the logits output by the value network is nonpositive:

$$v(spec, pp) = -\text{SoftPlus}(W_2\text{ReLU}(W_1\text{stateEncoding}(spec, pp))) \tag{4}$$

$$\text{SoftPlus}(x) = \log(1 + e^x) \tag{5}$$

**2-D CNN architecture:** The 2D spec encoder and canvas encoder both take as input $64 \times 64$ images, passed through 4 layers of 3x3 convolution, with ReLU activations after each layer and 2x2 max pooling after the first two layers. The first 3 layers have 32 hidden channels and the output layer has 16 output channels.

**3-D CNN architecture:** The 3D spec encoder and canvas encoder both take as input $32 \times 32 \times 32$ voxel arrays, passed through 3 layers of 3x3 convolution, with ReLU activations after each layer and 4x4 max pooling after the first layer. The first 2 layers have 32 hidden channels and the output layer has 16 output channels.

**No REPL baseline:** Our "No REPL" baselines using the same CNN architecture to take as input the *spec*, and then use the same pointer network architecture to decode into the program, with the sole difference that, rather than attend over the CNN encodings of the objects in scope (which are hidden from this baseline), the pointer network attends over the hidden states produced at the time when each previously constructed object was brought into scope.

### A.7. String editing additional results

Figure 8 shows results on the string editing dataset from (6).

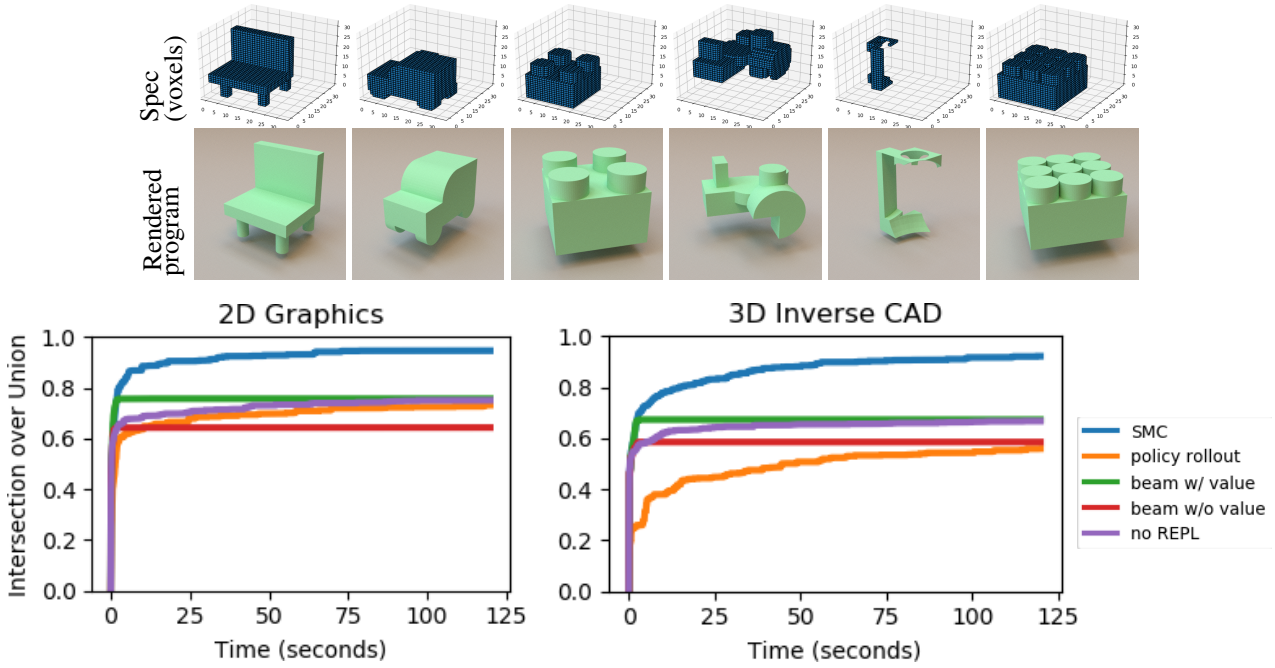


*Figure 4.* **Top:** Qualitative inverse CAD results: Rerendering program inferred from voxels from novel viewpoints. **Bottom:** Quantitative results for CAD on out-of-sample testing problems. Both models trained on scenes with up to 13 objects. Left: 2D models tested on scenes with up to 30 objects. Right: 3D models tested on scenes with up to 20 objects. SMC achieves the highest test accuracy.

| Spec: | | |
|---|---|---|
| 6/12/2003 | → | date: 12 mo: 6 year: 2003 |
| 3/16/1997 | → | date: 16 mo: 3 year: 1997 |
| Held out test instance: | | |
| 12/8/2019 | → | date: 8 mo: 12 year: 2019 |
| Results: | | |
| **SMC (Ours)** | → | **date: 8 mo: 12 year: 2019** |
| Rollout | → | date: 8 mo: 1282019 |
| Beam w/value | → | date: 8 mo: 12 year:2019 |
| Beam | → | date: 8 mo: 12 year: |
| RobustFill | → | date:12/8/2019 |

| Spec: | | |
|---|---|---|
| Dr Mary Jane Lennon | → | Lennon, Mary Jane (Dr) |
| Mrs Isaac McCormick | → | McCormick, Isaac (Mrs) |
| Held out test instance: | | |
| Dr James Watson | → | Watson, James (Dr) |
| Results: | | |
| | → | **Watson, James (Dr)** |
| | → | Watson, James |
| | → | Watson, JamesDr |
| | → | Watson, James ( |
| | → | Dr James Watson |

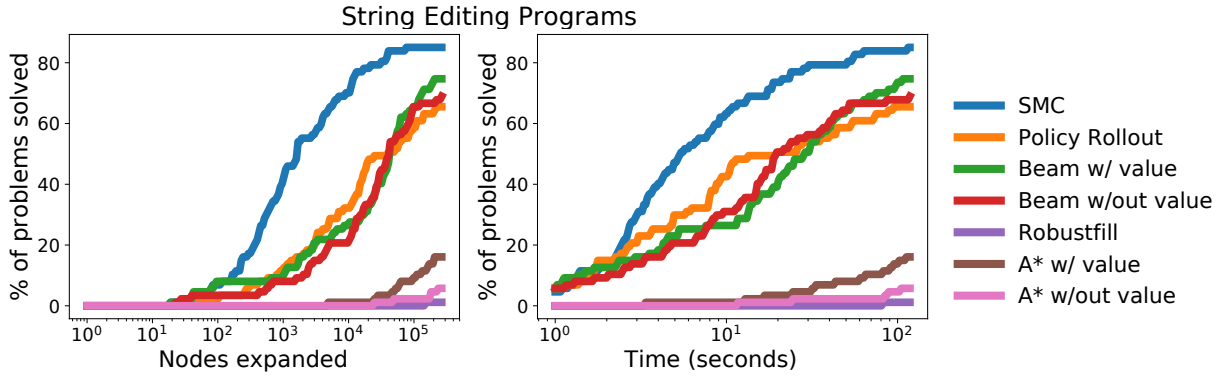


*Figure 5.* **Top:** Comparison of best programs on held-out inputs. Best programs determined by Levenshtein distance of program outputs to spec outputs. Leveraging the policy network, value network, and REPL-based execution guidance, SMC is able to consistently find programs with the desired behavior. **Bottom:** Results for String Editing tasks. Left: tasks solved vs number of nodes expanded. Right: tasks solved vs total time per task. Our SMC-based search algorithm solves more tasks using 10x fewer node expansions and less time than previous approaches. Note that x-axes are log scale.

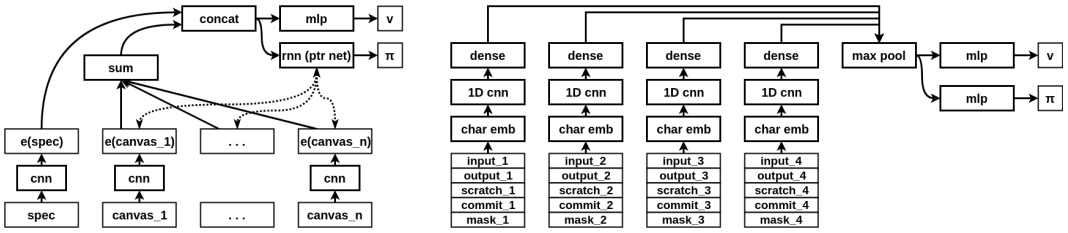


*Figure 6.* Left: CAD architecture. The policy is a CNN followed by a pointer network (22) (attending over the set of partial programs *pp*) which decodes into the next line of code. The value function is an additional 'head' to the pooled CNN activations. Right: String Editing architecture. We encode each example using an embedding layer, apply a 1-d convolution and a dense block, and pool the example hidden states to predict the policy and value.

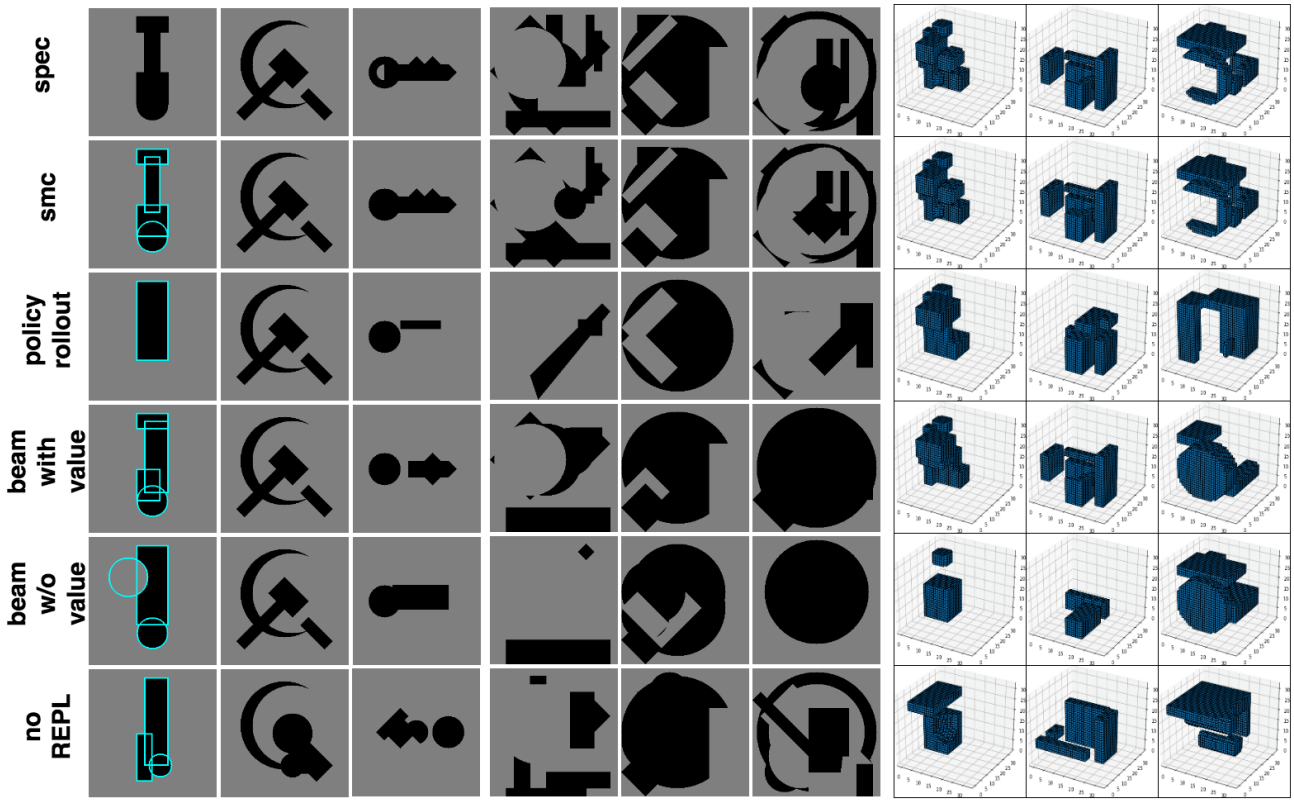


*Figure 7.* Derendering random scenes vs. ablations and no-REPL baseline. Teal outlines show shape primitives in synthesized 2D programs.

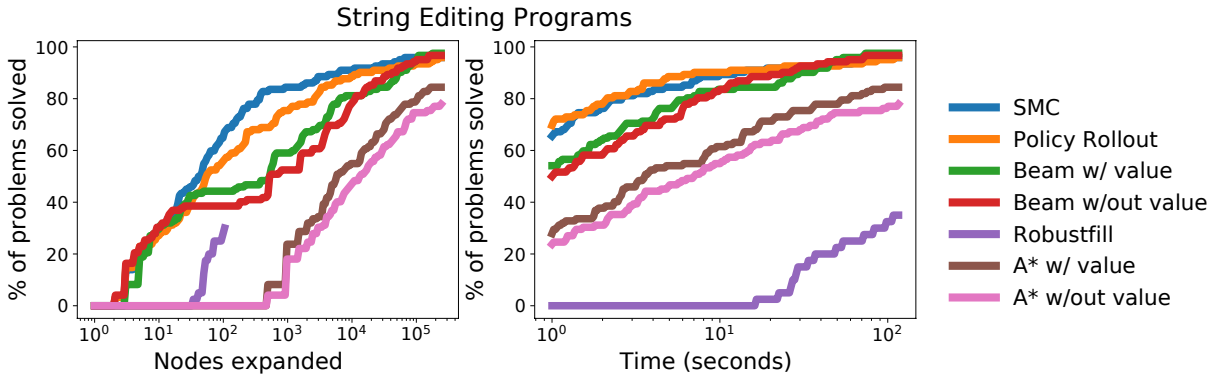


*Figure 8.* Results for String Editing tasks on dataset from (6). Left: tasks solved vs number of nodes expanded. Right: tasks solved vs total time per task. Note that x-axes are log scale.

inputs:
('3/16/1997', '4/17/1986', '6/12/2003', '4/23/1997')
outputs:
('date: 16 mo: 3 year: 1997', 'date: 17 mo: 4 year: 1986', 'date: 12 mo: 6 year: 2003', 'date: 23 mo: 4 year: 1997')
Const(d), Commit, Const(a), Commit, Const(t), Commit, Const(e), Commit, Const(:), Commit, Const( ), Commit, Replace1(/), Replace2( ), GetToken1(Number), GetToken2(1),
Commit, Const( ), Commit, Const(m), Commit, Const(o), Commit, Const(:), Commit, Const( ), Commit, GetUpTo(Number), Commit, Const( ), Commit, Const(y), Commit,
Const(e), Commit, Const(a), Commit, Const(r), Commit, Const(:), Commit, Const( ), Commit, GetFrom(/), Commit

inputs:
('April 19, 2:45 PM', 'July 5, 8:42 PM', 'July 13, 3:35 PM', 'May 24, 10:22 PM')
outputs:
('April 19, approx. 2 PM', 'July 5, approx. 8 PM', 'July 13, approx. 3 PM', 'May 24, approx. 10 PM')
GetUpTo( ), Commit, GetFirst1(Number), GetFirst2(-3), Commit, Const(,), Commit, Const( ), Commit, Const(a), Commit, Const(p), Commit, Const(p), Commit, Const(r),
Commit, Const(o), Commit, Const(x), Commit, Const(.), Commit, Const( ), Commit, GetFrom(,), GetFirst1(Digit), GetFirst2(3), GetFirst1(Digit), GetFirst2(-3), Commit,
Const( ), Commit, Const(P), Commit, Const(M), Commit

inputs:
('cell: 322-594-9310', 'home: 190-776-2770', 'home: 224-078-7398', 'cell: 125-961-0607')
outputs:
('(322) 5949310 (cell)', '(190) 7762770 (home)', '(224) 0787398 (home)', '(125) 9610607 (cell)')
Const((), Commit, ToCase(Proper), GetFirst1(Number), GetFirst2(1), GetFirst1(Char), GetFirst2(2), Commit, Const()), Commit, Const( ), Commit, GetFirst1(Number),
GetFirst2(5), GetFirst1(Char), GetFirst2(-2), GetToken1(Char), GetToken2(3), Commit, SubStr1(-16), SubStr2(17), GetFirst1(Number), GetFirst2(4), GetToken1(Char),
GetToken2(-5), Commit, GetFirst1(Number), GetFirst2(5), GetToken1(Char), GetToken2(-5), Commit, GetToken1(Number), GetToken2(2), Commit, Const( ), Commit,
Const((), Commit, GetUpTo(-), GetUpTo(Word), Commit, Const()), Commit

inputs:
('(137) 544 1718', '(582) 431 0370', '(010) 738 6792', '(389) 820 9649')
outputs
('area code: 137, num: 5441718', 'area code: 582, num: 4310370', 'area code: 010, num: 7386792', 'area code: 389, num: 8209649')
Const(a), Commit, Const(r), Commit, Const(e), Commit, Const(a), Commit, Const( ), Commit, Const(c), Commit, Const(o), Commit, Const(d), Commit, Const(e), Commit,
Const(:), Commit, Const( ), Commit, GetFirst1(Number), GetFirst2(0), Commit, Const(,), Commit, Const( ), Commit, Const(n), Commit, Const(u), Commit, Const(m),
Commit, Const(:), Commit, Const( ), Commit, GetFrom()), GetFirst1(Number), GetFirst2(2), Commit

*Figure 9.* Examples of long programs inferred by our system in the string editing domain.