Advancements in Neural Program Synthesis

Yufan Zhuang (yz3453)[1] Saikat Chakraborty (sc4537)[1]

[1]Columbia University

April 4, 2019

Overview

Problem Overview (contd.)

The Goal

Given a set of Inputs generate the corresponding Outputs.
2 ways to address this:

- **Program Induction :** Model the relation between Input and Output by a neural network.
    - Difficult to model due to the discrete nature (often) of the I/O.
    - Also because of non-differentiability of I/O relation.
- **Program Synthesis :** Generate (or synthesize) a program that takes in the Input and generate Output
    - Do not have to model the input output behavior.
    - Often some complex relations between I/O are implemented as functionality of code elements.

Problem Overview

Given set of inputs and outputs, the model is asked to give the correct program.

Existed Major Approaches and Corresponding Drawbacks

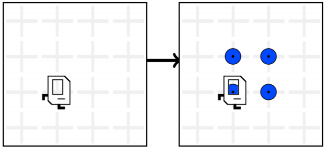
- Neural-guided Search (Exponential Search Space)
- **Direct Synthesis** (Could be Syntactically Wrong)
- Differentiable Program with Gradient Methods (Difficult to Model)

Shortcomings of Current Direct Synthesis Methods

The model is typically a Neural Machine Translation Model (Inputs/Outputs -> Programs)

Two primary drawbacks

- **May not be a program**: syntax correctness cannot be guaranteed (BLEU evaluation for NLP problems)
- **Program Aliasing**: multiple programs can be semantically correct, but the supervised model would just go for the ground truth


(a) Partial Specification as IO Pair

Program A

```
def run():
  repeat(4):
    putMarker()
    move()
    turnLeft()
```

Program B

```
def run():
  while(noMarkersPresent):
    putMarker()
    move()
    turnLeft()
```

Domain: Karel

Note that it has control flows but no data flows

$$
\begin{array}{rcl}
\text{Prog } p & := & \texttt{def run() : } s \\
\text{Stmt } s & := & \texttt{while}(b) : s \mid \texttt{repeat}(r) : s \mid s_1 ; s_2 \mid a \\
 & \mid & \texttt{if}(b) : s \mid \texttt{ifelse}(b) : s_1 \texttt{ else} : s_2 \\
\text{Cond } b & := & \texttt{frontIsClear}() \mid \texttt{leftIsClear}() \mid \texttt{rightIsClear}() \\
 & \mid & \texttt{markersPresent}() \mid \texttt{noMarkersPresent}() \mid \texttt{not } b \\
\text{Action a} & := & \texttt{move}() \mid \texttt{turnRight}() \mid \texttt{turnLeft}() \\
 & \mid & \texttt{pickMarker}() \mid \texttt{putMarker}() \\
\text{Cste } r & := & 0 \mid 1 \mid \cdots \mid 19
\end{array}
$$

Figure 4: The Domain-specific language for Karel programs.

Leveraging Grammar and Reinforcement Learning for Neural Program Synthesis

Rudy Bunel[1], Matthew Hausknecht[2], Jacob Devlin[3], Rishabh Singh[2], Pushmeet Kohli[4]

[1]University of Oxford

[2]Microsoft Research

[3]Google

[4]Deepmind

April 4, 2019

Goal of this Paper

- Deep Learning Model to Generate Tokens Directly
- Set up an Reinforcement Learning Framework to Overcome the Program Aliasing Problem
- Introduce a syntax checker to prune the space of possible programs

Problem Formulation

At training time, assumed to have access to N training samples, each with K input/output states and a ground truth program λ.

$$D = \{(\{IO_i^k\}_{k=1...K}, \lambda_i\}_{i=1...N}, s.t.\ \lambda_i(I_i^k) = O_i^k,\ \forall i \in 1...N,\ \forall k \in 1...K$$

The goal is to learn a synthesizer $\sigma : \{IO_i^k\}_{k=1...K} \rightarrow \hat{\lambda}$
At testing time, the test data is consisted of both specification examples and held-out examples.

$$D_{test} = \{\{IO_j^{k_{spec}}\}_{k_{spec}=1...K}, \{IO_j^{k_{test}}\}_{k_{test}=K+1...K'}\}_{j=1...N_{test}}$$

In testing the output program $\hat{\lambda}$ is based on the specification sets, then evaluated on the entire test set

Model Architecture

Each program is represented by a sequence of tokens

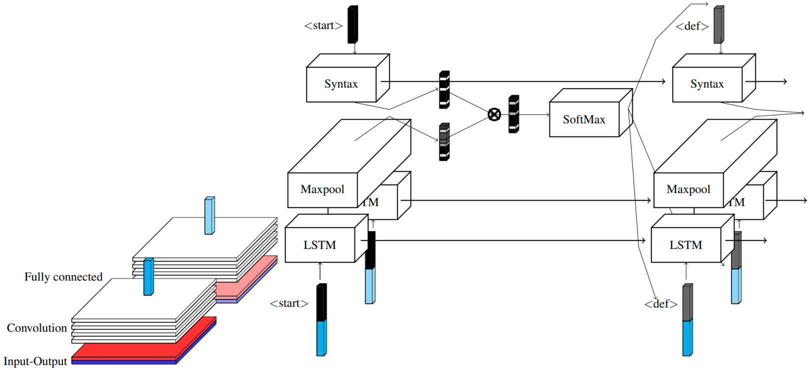
$$\lambda = [s_1, s_2, ..., s_L]$$

where each token belongs to an alphabet Σ (dictionary).

The form of the model:

$$p_\theta(\lambda_i | \{IO_i^k\}_{k=1...K}) = \prod_{t=1}^{L_i} p_\theta(s_t | s_1, s_2, ..., s_{t-1}, \{IO_i^k\}_{k=1...K})$$

Model Architecture



Model Architecture

1. Each pair of IO is embedded jointly by a CNN (share wights)
2. One LSTM is run for each example (share wights)
3. At each time step, the IO pair embedding and the previous token is fed into the LSTM
4. The output of the LSTMs goes through a maxpool layer and a linear layer to generate the next prediction after softmax
5. The syntax checker produces a mask on the output of the linear layer to filter out those have incorrect syntax

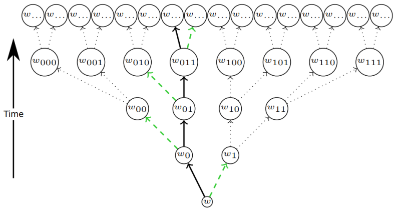
Objective Function

Maximum Likelihood Estimation The default solution is to do MLE, since this is a supervised learning problem.

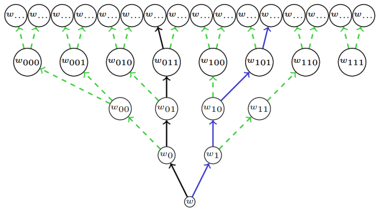
$$\theta^* = \operatorname{argmax}_\theta \prod_i p_\theta(\lambda_i | \{IO_i^k\}_{k=1...K})$$

Drawbacks

- Still has the programming aliasing problem
- Exposure bias (the model won't explore enough sample paths, since it only sees the data distribution)



Training with exposure bias



Training in expectation (Reinforce)

Objective Function

Reinforcement Learning A potentially better objective is to optimize over the ecpectations

$$\theta^* = \mathrm{argmax}_\theta L_R(\theta),\ \textit{where } L_R(\theta) = \sum_i (\sum_\lambda p_\theta(\lambda | \{IO_i^k\}_{k=1\ldots K}) R_i(\lambda))$$

where $R_i(\lambda)$ denotes the reward for sampled programs. If a simulator is accessible, R can be designed to optimize for generalization and for preventing over-fitting. Other things like conciseness can also be taken account in R.

Objective Function

REINFORCE Trick/score function estimator Notice that the sum over all possible λ is intractable.

$$\theta^* = \mathrm{argmax}_\theta L_R(\theta),\ \textit{where}\ L_R(\theta) = \sum_i (\sum_\lambda p_\theta(\lambda | \{IO_i^k\}_{k=1\ldots K}) R_i(\lambda))$$

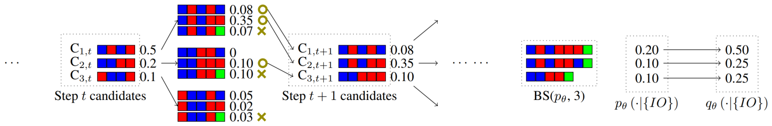
We can only use monte-carlo methods to sample gradient, but how?

$$\begin{aligned}\nabla_\theta L_R(\theta) &= \sum_i \sum_\lambda \nabla_\theta p_\theta(\lambda | \{IO_i^k\}_{k=1\ldots K}) R_i(\lambda) \\ &= \sum_i \sum_\lambda \nabla_\theta \log(p_\theta(\lambda | \{IO_i^k\}_{k=1\ldots K})) R_i(\lambda) p_\theta(\lambda | \{IO_i^k\}_{k=1\ldots K}) \\ &\approx \sum_i \sum_{r=1}^{S} \frac{1}{S} \nabla_\theta \log(p_\theta(\lambda_r | \{IO_i^k\}_{k=1\ldots K})) R_i(\lambda)\end{aligned}$$

where $\lambda_r \sim p_\theta(\lambda | \{IO_i^k\}_{k=1\ldots K})$

Objective Function

Approximate p_θ with q_θ Since the domain is discrete with small amount of training data per program, it is difficult to sample different programs according to the learned distribution. Thus, the model wouldn't explore much, and this is not desirable.



At every time step, S most likely samples returned by a Beam Search are used to construct the approximate distribution q_θ.

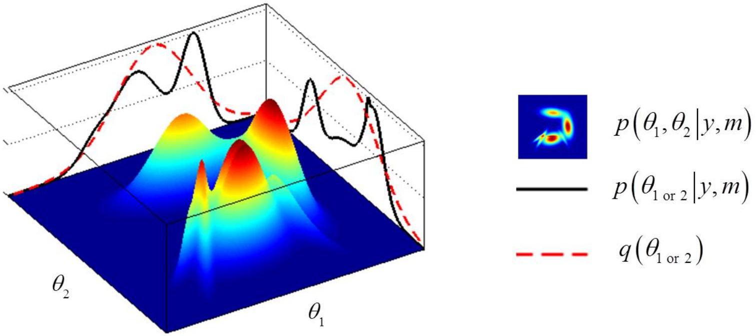
$$q_\theta(\lambda_r|\{IO_i^k\}_{k=1\dots K}) = \frac{p_\theta(\lambda_r|\{IO_i^k\}_{k=1\dots K})}{\sum_{\lambda_r \in BS(p_\theta, S)} p_\theta(\lambda_r|\{IO_i^k\}_{k=1\dots K})}, \; \textit{otherwise } 0$$

Objective Function

Approximate p_θ **with** q_θ Then the objective becomes tractable

$$L_R(\theta) \approx \sum_i \left(\sum_\lambda q_\theta(\lambda | \{IO_i^k\}_{k=1 \dots K}) R_i(\lambda) \right)$$

q_θ approximates p_θ at modes



Objective Function

Approximate p_θ with q_θ Using q_θ, the objective function can be more complicated for a better exploration. Instead of optimizing over the expected reward, a bag of C programs can be sampled and the best one is kept. By doing this, probability mass would be assigned to several candidate programs, thus resulting a higher diversity of outputs.

$$\theta^* = \mathrm{argmax}_\theta \sum_i \Big(\sum_{\lambda \in BS(p_\theta, S)^C} [\max_{j \in \{1,..,C\}} R_i(\lambda_j)] \Big(\prod_{r \in 1,\dots,C} q_\theta(\lambda_r | \{IO_i^k\}_{k=1\dots K}) \Big) \Big)$$

Syntax Checker

Condition on the syntax If there is an syntax checker (serves like IDE auto-complete). Include the syntax checker in the likelihood, denote *stx* as the event that the sampled program is syntactically correct.

$$\begin{aligned} p(\lambda|\{IO_i^k\}_{k=1...K}, stx) &= \frac{p(stx|\lambda, \{IO_i^k\}_{k=1...K}) \cdot p(\lambda|\{IO_i^k\}_{k=1...K})}{p(stx|\{IO_i^k\}_{k=1...K})} \\ &\propto p(stx|\lambda, \{IO_i^k\}_{k=1...K}) \cdot p(\lambda|\{IO_i^k\}_{k=1...K}) \\ &\propto p(stx|\lambda) \cdot p(\lambda|\{IO_i^k\}_{k=1...K}) \end{aligned}$$

At the token level, it is the same

$$\begin{aligned} p(s_t|s_1, ..., s_{t-1}, \{IO_i^k\}_{k=1...K}, stx_{1,...,t}) \propto\ & p(stx_{1,...,t}|s_1, ..., s_t) \\ & \cdot p(s_t|s_1, ..., s_{t-1}, \{IO_i^k\}_{k=1...K}) \end{aligned}$$

Syntax Checker

Jointly Learned Syntax If there isn't syntax checker, we can train a network to learn that. Denote the checker network as *syntaxLSTM* g_ϕ. Its activation is $x \to -\exp(x)$

In training, one additional loss term is added:

$$L_{syntax} = -\sum_i \sum_t g_\phi(s_t^i | s_1^i, ..., s_{t-1}^i),\ where\ \lambda_i = [s_1^i, s_2^i, ..., s_L^i]$$

To penalize negative output $(-\exp g_\phi)$ for valid syntax.

Evaluation

Dataset: Synthetic dataset from sampling programs and sampling IO pairs. In the testing phase, 5000 programs are left out with 6 IO pairs split into a specification (5 pairs) and a held-out pair.

Top-1 Generalization Accuracy: the accuracy of the most likely program synthesized having the correct behaviour across all input-output examples.

Exact Match Accuracy: the accuracy of the most likely program synthesized is exactly the same as the reference program.

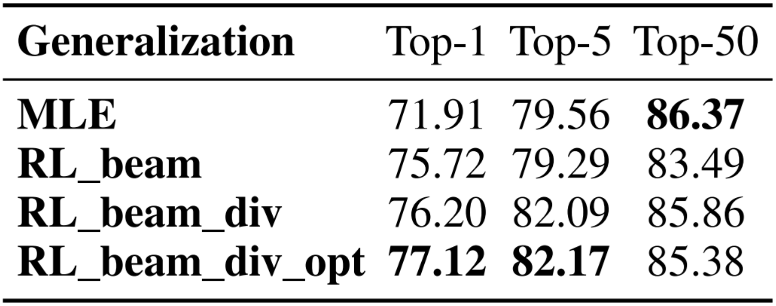
Experiment Results

	Full Dataset		Small Dataset	
Top-1	**Generalization**	**Exact Match**	**Generalization**	**Exact Match**
MLE	71.91	**39.94**	12.58	8.93
RL	68.39	34.74	0	0
RL_beam	75.72	8.21	**25.28**	**17.63**
RL_beam_div	76.20	31.25	23.72	16.31
RL_beam_div_opt	**77.12**	32.17	24.24	16.63

Full dataset denotes the 1 million program dataset, small dataset contains only 10,000 examples. RL_beam_div employs the richer objective function. RL_beam_div_opt adds penalty for long programs on top of RL_beam_div.

Experiment Results

Top-k accuracy



Generalization	Top-1	Top-5	Top-50
MLE	71.91	79.56	**86.37**
RL_beam	75.72	79.29	83.49
RL_beam_div	76.20	82.09	85.86
RL_beam_div_opt	**77.12**	**82.17**	85.38

MLE shows greater relative accuracy increases as k increases than RL. Methods employing beam search and diversity objectives reduce this accuracy gap by encouraging diversity in the beam of partial programs.

Experiment Results

Impact of Syntax

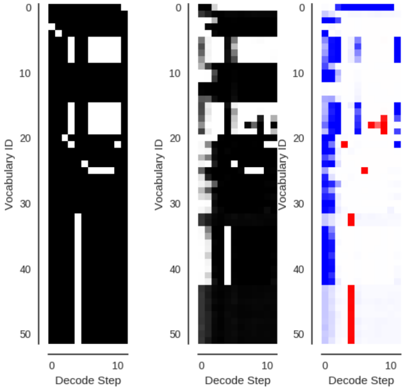
Top-1 Generalization	**Full Dataset**	**Small Dataset**
MLE	71.91	12.58
MLE_learned	69.37	**17.02**
MLE_handwritten	72.07	9.81
MLE_large	**73.67**	13.14

MLE_leaned denotes the usage of syntaxLSTM, MLE_handwritten denotes the usage of a handwritten syntax checker, MLE_large simply increases the size of parameters to the same number as MLE_leaned.

Just get a larger network ¯_(ツ)_/¯

Experiment Results

Learned Syntax (Program A)
Black indicates valid, white indicates invalid. Blue indicates the syntaxLSTM predicts a valid token to be invalid, red indicates the syntaxLSTM predicts a invalid token to be valid.



(a) Manual (b) Learned (c) Diff

Improving Neural Program Synthesis with Inferred Execution Trace

Richard Shin[1], Illia Polosukhin[2], Dawn Song[1]

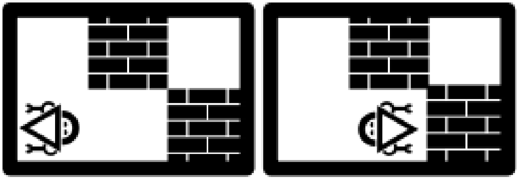
[1]UC Berkley

[2]NEAR Protocol

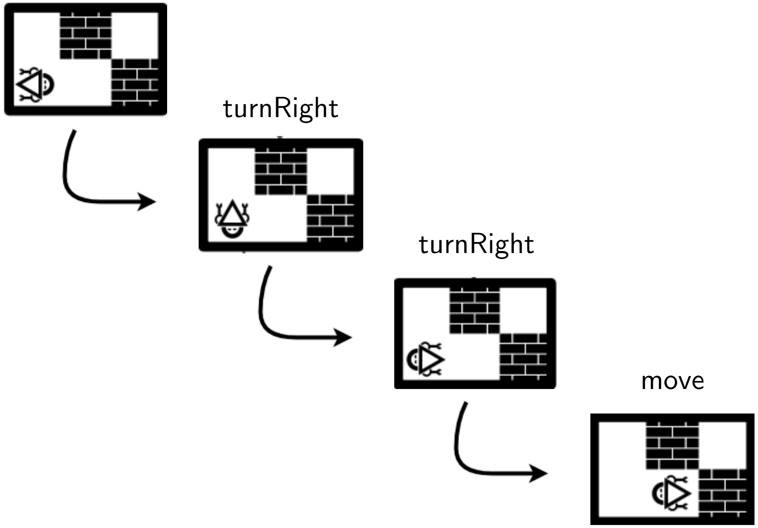
April 4, 2019

Motivation

- How to human thinks about problem solving
  - Given an Input and corresponding output, we think of sequence of steps.



Motivation Contd.



Motivation Contd.

- Given the sequence of actions (*i.e.* execution trace), it is much easier to reason about the program.
- For instance,
  - Given, $< turnRight, turnRight, move >$
  - It is much easier to reason about the actual program
  - The actual program is $repeat(2)\ \{\ turnRight()\ \}\ move()$

Overview of the Method

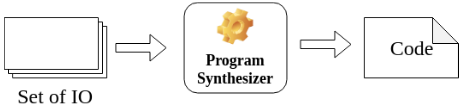


Figure: Traditional Program Synthesis approach

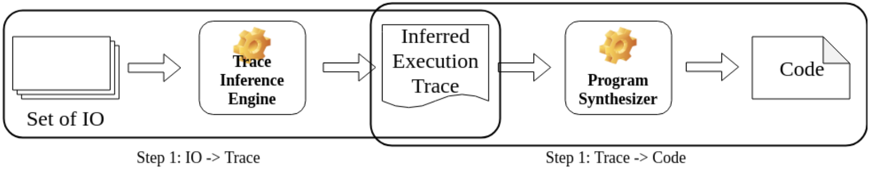


Figure: Program Synthesis With Inferred Execution Trace

Step 1: Trace Inference

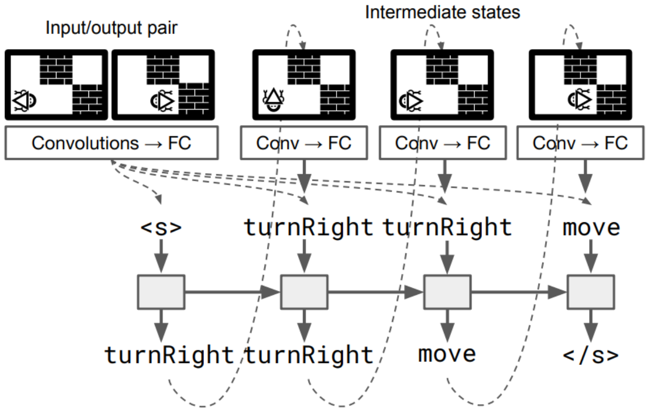
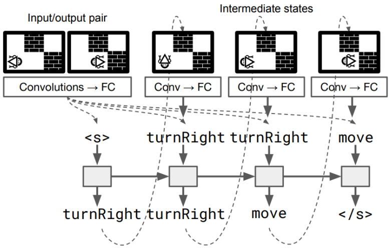


Figure: Inference of the Execution Trace from Each of the Examples

Step 1: Trace Inference (Contd.)



- For Each IO pair, a trace sequence is generated.
- **Convolution network** is used for encoding the state and Intermediate States.
- **LSTM** is used for generating sequence of actions.
- Ideally, LSTM should mimic the change in state, but experiment shows, if the actual state after every action is provided as auxiliary input, it performs better.

Step 2: Code Synthesis

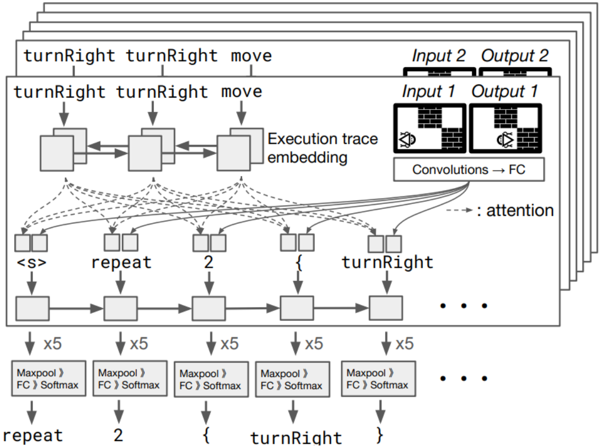
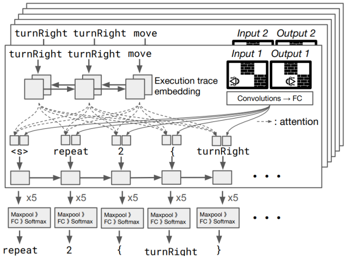


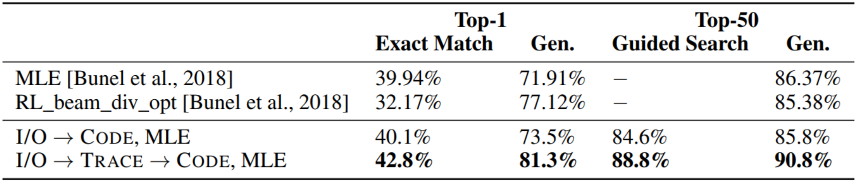
Figure: Synthesis of the actual code from the IO and also the inferred trace.

Step 2: Code Synthesis (Contd.)



- **LSTM** is used to generated final for as a token sequence.
- Inputs to the LSTM are:
  1. Previously generated code token.
  2. Attention context over inferred action sequence.
  3. Embedding of the IO pair through a convolution network.
- Output of LSTM at time i corresponding to j'th IO is $\hat{o}_{i,j}$
- Final token is generated as $softmax(W.MaxPool(\hat{o}_{i,1}, \hat{o}_{i,2}, ..., \hat{o}_{i,N}))$ where N is the number of IO pair.

Evaluation : Overall performance



	Top-1		Top-50	
	Exact Match	**Gen.**	**Guided Search**	**Gen.**
MLE [Bunel et al., 2018]	39.94%	71.91%	–	86.37%
RL_beam_div_opt [Bunel et al., 2018]	32.17%	77.12%	–	85.38%
I/O $\rightarrow$ CODE, MLE	40.1%	73.5%	84.6%	85.8%
I/O $\rightarrow$ TRACE $\rightarrow$ CODE, MLE	**42.8%**	**81.3%**	**88.8%**	**90.8%**

- Exact match is most stringent metric.
- Gen. (generalization) is when the generated program passes all the test cases, i.e. generates expected output.
- Guided search is generation of K code sequence using guided beam search with the softmax probabilities from previous step. (very similar to DeepCoder[Balog et.al. ICLR 2017])

Evaluation : On different data stripe

Slice	% of dataset	I/O → CODE	I/O → TRACE → CODE	Δ%
No control flow	26.4%	100.0%	100.0%	+0.0%
Only Conditions	15.6%	87.4%	91.0%	+3.6%
Only Loops	29.9%	91.3%	94.3%	+3.0%
With all control flow	73.6%	79.0%	84.8%	+5.8%
Program length 0-15	44.8%	99.5%	99.5%	+0.0%
Program length 15-30	40.7%	80.8%	86.9%	+6.1%
Program length 30+	14.5%	48.6%	61.0%	+12.4%

Observations:

- In **no control flow**, there is not branch, loop etc. hence code is sequencetial, sequential models(i.e. LSTM) used in Step 1 and 2 generates 100% correct code.
- For programs **with branches (and also loops)**, other models (Tree based, graph based:Allamanis et.al ICLR'18) are worth looking.
- For **Longer code sequences (30+ tokens)**, LSTM loses information due to long term dependency. Self attention models (Vaswani et.al. NIPS'17) can be worth looking here.

Neural Program Synthesis : Summary

- Neural Program Synthesis is and interesting field that is gaining focus in past few years.
- In last few years there are tons of papers published in different prestigious conferences (i.e. NeurIPS, ICLR, ICML, POPL etc.)

Neural Program Synthesis : Summary

Caveats:

- The dataset this papers are on has no concept of variables.
- All the programs are data dependent on a global state, and hence no sophisticated data dependency.
- Synthesizing code for general purpose programming languages (i.e. Java, Python) with complex syntactic and semantic structure is an open challenge as of now.

Opinion

- For generating general purpose programs, some form of concrete program analysis technique is needed to augment ML based techniques.
- The well-defined grammars and structural information in general programs should be utilized

The End