SPoC: Search-based Pseudocode to Code
Sumith Kulat, Panupong Pasupat, Kartik Chandra, Mina Lee, Oded Padon, Alex Aiken, Percy Liang
Department of Computer Science, Stanford University

Summary: Given pseudocode and test cases, the task is to synthesize a program, which will be evaluated on functional correctness. We release the SPoC dataset (18K programs + human-authored pseudocode), a search-based synthesizer, and error localization models to guide search based on information from compilation errors.

Task and Motivation

Goal: Synthesize programs that are long (10–20 lines) and functionally correct.

Semantic parsing
natural language
usually short
(e.g., SQR, logical forms)
yes
(e.g., database execution)
Language to code
natural language
long
(e.g., methods, classes)
mostly no
(e.g., exact match, BLEU)
Test-driven synthesis
test cases
usually short
yes
This work
natural language
long
program
yes

SPoC Dataset
bit.ly/spoc-dataset

Main features:
• Complex programs from programming competitions + test cases, inspired by the NAPS dataset (Kaveshnyksy et al., NAPPI 2018).
• 18356 programs.
• All programs come with human-authored pseudocode.

Local-level challenges: Translating each line is non-trivial.

Semantic parsing

Language to code

Test-driven synthesis

This work

SPoC Dataset

bit.ly/spoc-dataset

Main features:
• Complex programs from programming competitions + test cases, inspired by the NAPS dataset (Kaveshnyksy et al., NAPPI 2018).
• 18356 programs.
• All programs come with human-authored pseudocode.

Local-level challenges: Translating each line is non-trivial.

High-level descriptions:
read n values into array a and array b
for (int i = 0; i < n; i++)
    cin >> a[i] >> b[i];
read n and m in a loop, printing
while (cin >> a) cout << a[0] / a[1] / <end>
for (int i = 0; i < n; i++)
    cout << a[i];
Complex sentences and diverse operations:
change max to i if tree[i] > tree[max]
max = tree[i] > tree[max] ? i : max;
... or max otherwise
if m and n are odd
if (a % 2 == 0 && b % 2 == 0)
if (a == b = 0 && c % 2 == 0)
if (a is a digit return 1
Complex-context interpretation:
add x to y (x is a set)
q.insert(x);
add ok to ans (ans is an integer)
ans += ok;
add element a to v (v is a vector)
v.push_back(a);

Global-level challenges:
The programs are 14 lines long on average.
One wrong code line can make the whole program incorrect!
And most programs have at least one difficult line. (See the experiments)

Two data splits: TestP (split by problem) and TestW (split by pseudocode author).

Evaluation:
pseudocode lines
Input Output size

Evaluate functional correctness?

Base Framework

When a program fails, we want to avoid using the source of that failure over again.

Proposal: When a compilation error occurs, use an error localization method to infer the offending code line(s), then demote or blacklist them.

In the example figure above, (c11, c22, c33) satisfies the test cases. Best-first search iterates in the order of decreasing probabilities and succeeds in 4 compiler calls. The error localization method decreases weights c33, leading to an earlier success.

Method 1: Multiclass classification. Predict the offending line from the error message.

Method 2: Prefix-based pruning. Spend a few trials to validate code prefixes.

Experiments and Takeaways

Takeaway 1: Long programs → more chances to go wrong. Even though line-level translation accuracy is 85%, stitching the top translations gives a success rate of 24.6%.

Takeaway 2: Search increases the success rate. Under the budget of 100 trials, the success rate goes up to 44.7%.

Takeaway 3: Error localization reduces the number of trials needed:
• The multiclass classification model reduces the number of trials needed in 15.5% of the programs (median reduction of 26 trials).
• Prefix-based pruning increases the number of trials on easy problems (since we need to compile prefixes) but greatly helps on harder programs.