# Code Representation Learning Using Prüfer Sequences (Student Abstract)

**Tenzin Jinpa and Yong Gao**

Department of Computer Science
University of British Columbia, Okanagan
Kelowna, BC, Canada
tenzin.jinpa@ubc.ca , yong.gao@ubc.ca

## Abstract

An effective and efficient encoding of the source code of a computer program is critical to the success of sequence-to-sequence deep neural network models for code representation learning. In this study, we propose to use the Prüfer sequence of the Abstract Syntax Tree (AST) of a computer program to design a sequential representation scheme that preserves the structural information in an AST. Our representation makes it possible to develop deep-learning models in which signals carried by lexical tokens in the training examples can be exploited automatically and selectively based on their syntactic role and importance. Unlike other recently-proposed approaches, our representation is concise and lossless in terms of the structural information of the AST. Results from our experiment show that prüfer-sequence-based representation is indeed highly effective and efficient.

## Introduction

To use sequence-to-sequence(seq2seq) deep learning models for program understanding, the source code of a computer program has to be represented as a sequence of tokens. Unlike natural languages, which are unstructured and noisy, computer codes are highly structured, and it is thus critical to encode as much as possible the structural information in a seq2seq learning model and to take advantage of the encoded information in the training. For example, the Structure-Based Traversal (SBT) by Hu et al. (2020) represents AST by a sequence of syntactic tokens and is generated by a depth-first traversal of the AST with parentheses pairs to retain the sub-tree information. The model Code2Seq (Alon et al. 2019) uses the concatenation of the token sequences along the paths between pairs of terminal nodes in an AST to represent a computer program. While these methods and models have been shown to be effective in comparison to "flat" sequence of tokens of code, the choices of the traversal method and the ordering of the tokens appear to be still arbitrary.

We propose to use the Prüfer sequence of the AST of a computer program to design a sequential representation scheme that preserves the structural information in an AST. Our representation makes it possible to develop deep-learning models in which signals carried by lexical tokens in the training examples can be exploited automatically and selectively based on their syntactic role and importance. Unlike other recently-proposed approaches, our representation is concise and lossless due to the fact that an AST can be uniquely reconstructed from its Prüfer sequence.

## Prüfer Sequence of an AST

The Prüfer sequence (West 2000) of a node-labeled tree is a sequence of node labels from which the tree can be uniquely reconstructed. Given a tree $T$ with $n$ nodes labeled by the integers $\{1, \cdots, n\}$, its Prüfer sequence is a sequence of $(n-2)$ node labels (i.e., integers) and can be formed by successively removing the leaf with the smallest label and including the label of its parent as the next node label in the Prüfer sequence. The process stopped when only two nodes were left in the tree.

Since ASTs are labeled by syntactic and lexical tokens, we use a fixed mapping to map each token in the given token set to a unique integer and use it as the integer label of the AST-node that is labeled by the token. The Prüfer sequence constructed from this integer-labeled AST is then mapped back to a sequence of syntactic tokens, which we call the **"syntactic Prüfer sequence"** and is used as part of the input sequence to our learning model.

### Advantages of Learning with a Prüfer-Sequence Representation

The syntactic Prüfer sequence can be regarded as a "transformed" and "quantified" version of an AST and the corresponding source code where

1. the frequency with which a syntactic token appears is decided by the degree of the corresponding AST node and quantifies the "importance" of the token (measured the size of the code block it controls);
2. the positions of the appearances of syntactic tokens in the Prüfer sequence are decided by the position of the corresponding node in the tree; and
3. a lexical token labeling a leaf node of an AST never appears in the Prüfer sequence, but its "significance" can be measured by the syntactic importance of the parent of the leaf node.

| Model | S-BLEU | C-BLEU | METEOR | ROUGE-L |
|---|---|---|---|---|
| Lexical-Token-Only Model | 36.21 | 27.30 | 19.01 | 40.78 |
| Code2Seq (Alon et al. 2019) | 20.72 | 4.56 | 10.21 | 20.63 |
| TL-CodeSum (Hu et al. 2018) | 37.20 | 28.43 | 19.64 | 41.29 |
| BFS-Hybrid-DeepCom (Hu et al. 2020) | 37.98 | 29.08 | 19.72 | 41.03 |
| Hybrid-DeepCom (Hu et al. 2020) | 38.19 | 29.28 | 19.87 | 41.15 |
| Our Model (Prüfer Encoder + Hu's Source Encoder) | 38.38 (0.5%) | 29.43 (0.5%) | 20.13 (1.3%) | 41.82 (1.3%) |
| Our Model (Prüfer Encoder + Context Encoder) | 39.67 (3.3%) | 31.01 (5.7%) | 21.01 (5.6%) | 43.45 (5.1%) |

Table 1: Effectiveness of Models based on Machine Translation metrics for Dataset-1

| Model | S-BLEU | C-BLEU | METEOR | ROUGE-L |
|---|---|---|---|---|
| Lexical-Token-Only Model | 9.21 | 3.07 | 7.96 | 19.84 |
| Code2Seq (Alon et al. 2019) | 2.27 | 0.30 | 3.5 | 12.23 |
| BFS-Hybrid-DeepCom (Hu et al. 2020) | 13.41 | 3.47 | 7.29 | 20.42 |
| Hybrid-DeepCom (Hu et al. 2020) | 15.02 | 3.7 | 8.27 | 18.01 |
| Our Model (Prüfer Encoder + Hu's Source Encoder) | 15.50 (3.15%) | 3.85 (3.97%) | 8.9 (6.925%) | 20.79 (1.8%) |
| Our Model (Prüfer Encoder + Context Encoder) | 16.15(7.02%) | 4.49 (19.29%) | 9.72 (15.05%) | 24.73 (19.09%) |

Table 2: Effectiveness of Models based on Machine Translation metrics for Dataset-2

This is contrary to all the other recently proposed representations, where all tokens are treated equally, and their positions only partially capture their roles in the AST. Other properties that distinguish our representation are as follows.

1. A Uniqueness and Lossless Representation
   Our Prüfer-sequence representation is a lossless encoding because, given a fixed syntactic-token-to-integer mapping, there is a one-to-one correspondence between the set of ASTs and their syntactic Prüfer sequences.
2. Natural Separation of Lexical and Syntactic Tokens
   Properties of Prüfer sequences make it possible for us to define a structure-aware lexical context of an AST and to use it to design a Context Encoder in our deep learning model to learn more effectively from the lexical tokens from the source code.

## Prüfer-Sequence-Based Learning Model For Code Summarization

We developed an attention-based seq2seq model[1] for code summarization to study the effectiveness of the Prüfer-sequence-based representation, where we used two separate encoders, i.e., Prüfer Sequence Encoder is designed to learn from the structural information of the ASTs that are losslessly encoded in their syntactic Prüfer sequences. Gated Recurrent Units (GRUs) are used to map the syntactic Prüfer sequence $(X = x_1, ...., x_n)$ of a computer program to a sequence of hidden states as follows:

$$s_t = GRU(x_t, s_{t-1})$$

The Context Encoder, also consisting of GRUs, is designed to learn from the collection of lexical tokens (i.e., user-defined and program-specific values in the source code) organized in a way that reflects the structural information of the AST.

We perform our experiment on two Java datasets. Dataset-1 with 68469 pairs of java code and comments[2] and Dataset-2 with 163316 pairs of java code and comments[3].

## Results and Observations

As shown in the second last rows in Tables 1 and 2, our Prüfer Sequence Encoder and Hu's Source Encoder (second last rows in both Tables 1 and 2) has already had notable improvement over the baseline models, with the average performance improvement being $0.9\%$ for Dataset-1 and $3.96\%$ for Dataset-2. We attribute the performance improvement to the properties of the Prüfer sequence representation: a concise and lossless encoding that quantifies the "importance" of syntactic tokens and preserves their structural roles in describing the source code.

As shown in the last row in both Tables 1 and 2, the use of the Context Encoder boosted the performance of our model dramatically. The average performance improvement over baseline models is increased from $0.9\%$ to $5\%$ for Dataset-1 and $3.96\%$ to $15.11\%$ for Dataset-2. The significant performance gain of our model can be attributed to the context we have designed that helps amplify the learning-relevant lexical signals in the source code than the use of entire tokens as they appear in the source code.

## References

Alon, U.; Brody, S.; Levy, O.; and Yahav, E. 2019. code2seq: Generating Sequences from Structured Representations of Code. In *International Conference on Learning Representations*.

Hu, X.; Li, G.; Xia, X.; Lo, D.; and Jin, Z. 2020. Deep code comment generation with hybrid lexical and syntactical information. *Empirical Software Engineering*, 2179–2217.

Hu, X.; Li, G.; Xia, X.; Lo, D.; Lu, S.; and Jin, Z. 2018. Summarizing Source Code with Transferred API Knowledge. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence*, 2269–2275.

West, D. B. 2000. Introduction to Graph Theory. *Prentice Hall*, 81–83.

[1] Prüfer-Model-https://github.com/kardol123/Prufer

[2] https://github.com/xing-hu/TL-CodeSum/tree/master/data

[3] https://github.com/microsoft/CodeXGLUE/tree/ main /Code-Text/code-to-text