Seq2Seq or Seq2Tree: Generating Code Using Both Paradigms via Mutual Learning

Yunfei Zhao
Peking University
Beijing, China
zhaoyunfei@pku.edu.cn

Yihong Dong
Peking University
Beijing, China
dongyh@stu.pku.edu.cn

Ge Li*
Peking University
Beijing, China
lige@pku.edu.cn

ABSTRACT

Code generation aims to automatically generate the source code based on given natural language (NL) descriptions, which is of great significance for automated software development. Some code generation models follow a language model-based paradigm (LMBP) to generate source code tokens sequentially. Some others focus on deriving the grammatical structure by generating the program's abstract syntax tree (AST), i.e., using the grammatical structure-based paradigm (GSBP). Existing studies are trying to generate code through one of the above two models. However, human developers often consider both paradigms: building the grammatical structure of the code and writing source code sentences according to the language model. Therefore, we argue that code generation should consider both GSBP and LMBP. In this paper, we use mutual learning to combine two classes of models to make the two different paradigms train together. To implement the mutual learning framework, we design alignment methods between code and AST. Under this framework, models can be enhanced through shared encoders and knowledge interaction in aligned training steps. We experiment on three Python-based code generation datasets. Experimental results and ablation analysis confirm the effectiveness of our approach. Our results demonstrate that considering both GSBP and LMBP is helpful in improving the performance of code generation.

CCS CONCEPTS

• **Software and its engineering → Automatic programming**.

KEYWORDS

neural networks, mutual learning, code generation, abstract syntax tree

ACM Reference Format:
Yunfei Zhao, Yihong Dong, and Ge Li. 2023. Seq2Seq or Seq2Tree: Generating Code Using Both Paradigms via Mutual Learning. In *14th Asia-Pacific Symposium on Internetware (Internetware 2023), August 04–06, 2023, Hangzhou, China.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3609437.3609465

*Corresponding author.

Internetware 2023, August 04–06, 2023, Hangzhou, China

ACM ISBN 979-8-4007-0894-7/23/08...$15.00
https://doi.org/10.1145/3609437.3609465

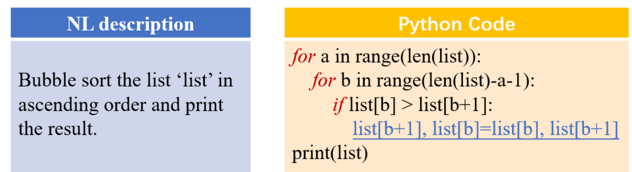


Figure 1: An NL description and the corresponding Python code snippet of the bubble sort algorithm. The italic code in red forms the grammatical structure of the algorithm, and the code in blue with underlined forms coherent code snippets.

1 INTRODUCTION

Code generation aims to automatically generate the source code according to natural language (NL) descriptions. Because of its potential in automated software development, code generation has become one of the research hotspots in software engineering. With the advantages of feature learning and generation capability, deep learning technology has been widely used in code generation [7, 16, 32, 33]. Some code generation models follow a language model-based paradigm (LMBP) to generate source code tokens sequentially. An example is the sequence-to-sequence (Seq2Seq) [23] model, consisting of an encoder and a decoder. The encoder takes an NL description as input, and the decoder outputs the corresponding code token sequence according to LMBP. Some studies [3, 6, 12, 16, 17, 20, 29] fall into this category, as do pre-trained models [2, 8, 28]. These Seq2Seq models inherit the advantages of the language model, capturing the information of the NL specification well and generating coherent code snippets. Some other models focus on deriving the grammatical structure by generating the abstract syntax tree (AST) of programs [5, 19, 22, 32, 33]. These models are based on the sequence-to-tree (Seq2Tree) model, using an encoder to process NL descriptions and a tree decoder to output a sequence of tree-construction actions, which correspond to the generation of an AST and can be further converted into the code. This generation is according to the grammatical structure-based paradigm (GSBP). These Seq2Tree models can generate well-formed code through AST structures and syntactic constraints.

We argue that code generation should consider both LMBP and GSBP, used by the Seq2Seq and Seq2Tree models. The reason is that humans consider both paradigms when programming. For example, consider writing an algorithm according to the NL description in Figure 1. The programmer needs to conceive the grammatical structure (red italic marks in Python code in Figure 1) and writes source code sentences in left-to-right order according to the language model (blue underlined marks in Python code in Figure 1). Inspired by human programming, code generation models should also combine these two distinct paradigms while leveraging the strengths of each. Intuitively, the output of the Seq2Tree models is an AST, which explicitly expresses the grammatical structure of the corresponding code. Therefore, the generation of Seq2Tree models is according to GSBP and has advantages in capturing the grammatical structure information. In contrast, the Seq2Seq models directly generate the token sequence of the source code according to LMBP and have the advantage of generating coherent fragments. We want to combine the two generation models, Seq2Seq and Seq2Tree, and take advantage of the different paradigms of the two models to generate more structural and natural [10] code snippets.

In this paper, we use the mutual learning method to realize the knowledge interaction between Seq2Tree and Seq2Seq models. Mutual learning [35] is a learning approach that enables different models to learn collaboratively by mimicking others' inferred probability distributions, and has been used in many fields. Our mutual learning framework integrates different generation paradigms and targets, called MutualTS (**mutual** learning framework combining the Seq2**T**ree and Seq2**S**eq models). Specifically, MutualTS consists of an encoder for NL input and two decoders, the Seq2Tree end and the Seq2Seq end. The Seq2Tree end is similar to the decoder of the Seq2Tree model and outputs AST results. The Seq2Seq end is similar to the decoder of the Seq2Seq model and outputs source code results. In the specific training step, we calculate the Kullback-Leibler (KL) divergence between the output probability distributions of the two decoders to achieve mutual learning. This allows each decoder to learn not only from the training data but also from the knowledge of another decoder to further enhance itself. In addition, the shared encoder can also help to integrate the knowledge of the two decoders. The critical challenge in the MutualTS framework is aligning the two different output formats (AST & code) to enable mutual learning. Therefore, we first align the generation steps of two decoders, i.e., establish a correspondence between code tokens and AST nodes. We then align the probability space on the Seq2Tree end and the Seq2Seq end, i.e., create a mapping between the tree-construction action table and the code token vocabulary. Thus, the Seq2Seq and Seq2Tree models in the MutualTS framework can realize knowledge interaction.

After the MutualTS framework is trained, one of the two decoders, Seq2Tree or Seq2Seq, can be used with the encoder to form an independent encoder-decoder model for a generation. Although this encoder-decoder model is still generated using a single paradigm, through mutual learning with another decoder based on another paradigm, both Seq2Tree and Seq2Seq end can gain knowledge about another generation paradigm and enhance themselves. Experimental results on three datasets prove this view and verify the performance of MutualTS. We further verify the effectiveness of each component in the MutualTS framework by ablation experiments. Through the analysis of samples, we find that different generation paradigms have advantages in different situations and can be mutually enhanced through mutual learning.

In summary, the contributions of this paper are:

- We argue that code generation should consider both the grammatical structure-based paradigm and the language model-based paradigm, which are used by the Seq2Tree models and the Seq2Seq models, respectively.
- We propose MutualTS, a mutual learning framework for code generation. Through mutual learning between two different decoders, our framework can refer to both paradigms simultaneously to improve the generation.
- Experimental results on three datasets show the improvements provided by our framework and confirm the validity of considering both paradigms.

2 RELATED WORK

Previous methods of code generation fall into two categories: using LMBP or GSBP. LPN [16] regarded code generation as a conditional text generation task and introduced the Seq2Seq model, using LMBP. Later, many methods optimize this generation method. For example, Wei et al. [29] uses dual learning to jointly train code generation and representation tasks. Korbak et al. [15] and Wang et al. [27] regard compilability as one of the training objectives of the Seq2Seq model. The most advanced Seq2Seq models are primarily based on Transformer, such as pre-trained models CodeT5 [28] and UniXcoder [8].

On the other hand, some works generate code according to grammatical structures, using GSBP. Dong et al. [5] generate hierarchical trees, ensuring the correctness of parenthesis pairs. Yin17 [32] introduced Abstract Syntax Description Language (ASDL) grammar and proposed the Seq2Tree model to generate an AST with a series of tree-construction actions. TRANX [33] improve the model structure and become a widely used Seq2Tree model, with various enhancements proposed to improve its effectiveness. For example, Xu et al. [31] and Norouzi et al. [18] introduce external knowledge to enhance the model, ML-TRANX [30] use mutual learning between different traversal sequences in tree construction, Yin and Neubig [34] explores the reordering of candidate results. Among them, ML-TRANX [30] and TRANX-RL [13] are representative works of exploring the model generation method.

Unlike the above models, which only use a single paradigm, we explore the mutual learning between two different paradigms used by the Seq2Seq and Seq2Tree models to enhance both simultaneously. Zhang et al. [35] proposed the mutual learning method for image classification tasks. Due to the universality of mutual learning, it has been widely used in object detection [11], speech translation [36], text generation [4], and other fields. ML-TRANX [30] introduce mutual learning into GSBP Code generation. Their purpose is to enable knowledge interaction between different AST generations orders to broaden the context view of models using GSBP. We differ from this work in multiple ways: (1) We aim to consider both GSBP and LMBP in code generation and further extend the mutual learning method between these two paradigms. (2) Our mutual learning framework has two different generation objectives, code text, and AST, representing different program representations. (3) We implement the alignment between these different representations to calculate the mutual learning loss.

3 MUTUALTS

This section describes the full details of our MutualTS framework, including model details, alignment methods, and the mutual learning training process.

3.1 Seq2Seq and Seq2Tree in MutualTS

The mutual learning framework we design is model-free and can be applied to any two models using LMBP and GSBP. To ensure that the performance gap between the two models is not large, we use Seq2Seq and Seq2Tree models of similar sizes and with consistent training data. This is because if one model significantly outperforms the other, it becomes difficult for the better model to learn from the worse one. For instance, if the Seq2Seq end is a large pre-trained model and the Seq2Tree end is a small model, the performance gap between the two ends makes it hard for the Seq2Seq end to benefit from the knowledge of the Seq2Tree end, even if there is a code fragment that is more suitable to generate using GSBP. Given that the core of our discussion is whether models can benefit from combining LMBP and GSBP, the two ends should be defined to have similar sizes and consistent training data to exclude these factors as much as possible. Unfortunately, although there has been a variety of large pre-training models based on LMBP and achieved good results [2, 8, 28], large pre-training models based on GSBP need to be improved. Some pre-training models use syntax-structure information such as AST [8, 24], but they still generate code only using LMBP when training. Therefore, we do not use pre-trained models, even though the performance of these models is higher.

We choose Transformer [25] for the Seq2Seq end and TRANX [33] for the Seq2Tree end as base models, both of which are typical models of LMBP and GSBP, respectively, and have been widely studied and used. We change the TRANX model to a Transformer architecture to balance the capabilities of the two models. In this section, we first present the model details of Transformer and TRANX, and then present our mutual learning framework.

Transformer [25] framework is widely used in various sequence generation tasks due to its outstanding performance. The canonical Transformer has an encoder-decoder structure where both encoder and decoder consist of stacked layers. The core component of a layer is the multi-head self-attention mechanism. Given an input feature sequence $\boldsymbol{X} = [x_1, ..., x_n] \in \mathbb{R}^{n \times d_x}$, each attention head computes a sequence of new features $\boldsymbol{H} = [h_1, ..., h_n] \in \mathbb{R}^{n \times d_h}$ as:

$$\boldsymbol{H} = \mathrm{softmax}\left(\frac{\boldsymbol{Q}\boldsymbol{K}^\top}{\sqrt{d_k}}\right)\boldsymbol{V}, \tag{1}$$

$$\boldsymbol{Q} = \boldsymbol{X}W^Q, \boldsymbol{K} = \boldsymbol{X}W^K, \boldsymbol{V} = \boldsymbol{X}W^V, \tag{2}$$

Where $W^Q \in \mathbb{R}^{d_x \times d_k}, W^K \in \mathbb{R}^{d_x \times d_k}, W^V \in \mathbb{R}^{d_x \times d_h}$ are learnable parameter matrices. The attention mechanism calculates the similarity between query matrix $Q \in \mathbb{R}^{n \times d_k}$ and key matrix $K \in \mathbb{R}^{n \times d_k}$ to get the attention weight between different positions and then aggregates the feature sequence according to the attention weight and the value matrix $V \in \mathbb{R}^{n \times d_h}$. The attention mechanism gives Transformer the ability to capture potential relationships.

TRANX [33] framework takes NL input and outputs a series of actions based on ASDL grammar. The AST can be constructed using these actions and predefined traversal order, and then converted into source code. There are three types of ASDL grammar-based actions used in TRANX:

ApplyRule$[c]$ actions apply a grammar rule constructor c to an opening composite frontier field, which has the same type as c. ApplyRule$[c]$ extends this field to generate an AST node and frontier fields.

Reduce actions mark the completion of the generation of child values for a field with optional(?) or multiple(★) cardinalities.

GenToken$[v]$ actions populate a primitive frontier field with a code token v.

As shown in Figure 2, for the target code in (a), TRANX uses the action sequence composed of ApplyRule, Reduce, and GenToken to gradually build the ASDL AST as shown in (b). The sequence of actions and corresponding opening fields are listed in (c). When a new non-terminal node is generated, new frontier fields f_i are also extended.

TRANX has an encoder-decoder structure to predict actions based on the programming language's ASDL rules and token vocabulary. TRANX uses a BiLSTM encoder to learn the word-level hidden state of an NL input x and an LSTM decoder to update the hidden state at each time step:

$$\mathbf{h}_t = f_{\mathrm{LSTM}}([\mathbf{E}(a_{t-1}) : \mathbf{s}_{t-1} : \mathbf{p}_t], \mathbf{h}_{t-1}), \tag{3}$$

where $\mathbf{E}(a_{t-1})$ is the embedding of the previous action, $\mathbf{s}_{t-1}$ is the previous decoder hidden state, and $\mathbf{p}_t$ indicates the information about the parent node. [:] denotes vector concatenation. Furthermore, the attention vector $\mathbf{s}_t$ is defined as

$$\mathbf{s}_t = \tanh(\mathbf{W}_s[\mathbf{c}_t : \mathbf{h}_t]), \tag{4}$$

where $\mathbf{c}_t$ is the weighted sum of input encoding by attention and $\mathbf{W}_s$ is a parameter matrix. Then the TRANX decoder calculates the probability of actions:

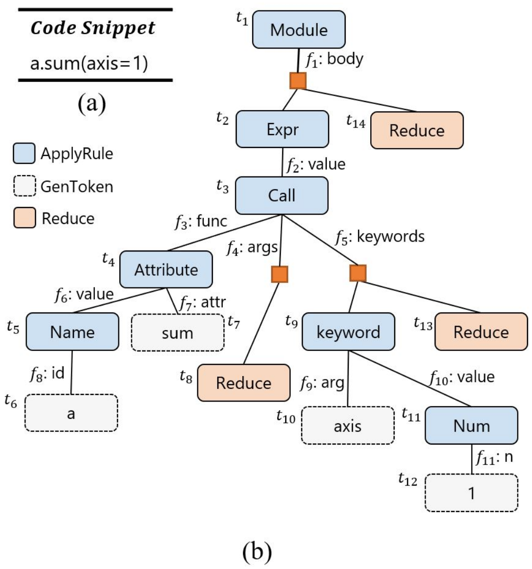
$$p(a_t = \textsc{ApplyRule}[c] | a_{<t}, \mathbf{x}) = \mathrm{softmax}(\mathbf{E}(c)^T \mathbf{W}\mathbf{s}_t), \tag{5}$$

$$p(a_t = \textsc{Reduce} | a_{<t}, \mathbf{x}) = \mathrm{softmax}(\mathbf{E}(\textsc{Reduce})^T \mathbf{W}\mathbf{s}_t), \tag{6}$$

$$\begin{aligned} p(a_t = \textsc{GenToken}[v] | a_{<t}, \mathbf{x}) &= p(\textsc{Gen}|a_{<t}, \mathbf{x}) p(v|\textsc{Gen}, a_{<t}, \mathbf{x}) \\ &+ (1 - p(\textsc{Gen}|a_{<t}, \mathbf{x})) p(v|\textsc{Copy}, a_{<t}, \mathbf{x}), \end{aligned} \tag{7}$$

where $\mathbf{W}$ is a parameter matrix used in generating ApplyRule action and Reduce action. There are three softmax functions based on $\mathbf{s}$ to calculate the probability of GenToken action. $p(\textsc{Gen}|a_{<t}, \mathbf{x})$ for choosing the generation operation, $p(v|\textsc{Gen}, a_{<t}, x)$ for generating v, and $p(v = x_i|\textsc{Copy}, a_{<t}, \mathbf{x})$ for selecting to copy x_i, respectively.

TRANX determines the set of syntactically allowed actions at each step of the prediction and selects from them based on the probability distribution. In this paper, the TranX model is changed from LSTM architecture to Transformer architecture (called TRANX-Trans) to share the encoder and achieve a relatively consistent effect with the Seq2Seq end. At time step t, the decoder of Tranx-trans accepts the sequence stacked by the historical information $[\mathbf{E}(a_i) : \mathbf{s}_i : \mathbf{p}_{i+1}], i \in [0, t-1]$ as input and outputs the hidden vector $\mathbf{h}_t$ for subsequent computation.



t	n_{f_t}	Action
t_1	root	$mod \rightarrow Module(stmt * body)$
t_2	f_1	$stmt \rightarrow Expr(expr\ value)$
t_3	f_2	$expr \rightarrow Call(expr\ func, expr * args, keyword * keywords)$
t_4	f_3	$expr \rightarrow Attribute(expr\ value, identifier\ attr)$
t_5	f_6	$expr \rightarrow Name(identifier\ id)$
t_6	f_8	$\boldsymbol{GenToken}[a]$
t_7	f_7	$\boldsymbol{GenToken}[sum]$
t_8	f_4	$\boldsymbol{Reduce}$
t_9	f_5	$keyword \rightarrow keyword(identifier?\ \text{arg}, expr\ value)$
t_{10}	f_9	$\boldsymbol{GenToken}[axis]$
t_{11}	f_{10}	$expr \rightarrow Num(object\ n)$
t_{12}	f_{11}	$\boldsymbol{GenToken}[1]$
t_{13}	f_5	$\boldsymbol{Reduce}$
t_{14}	f_1	$\boldsymbol{Reduce}$

(c)

Figure 2: Example of code generation procedure in TRANX. (a) The code snippet. (b) The corresponding ASDL AST for the code, f_i indicates field names. (c) The action sequence that TranX used to construct the AST, ApplyRule actions are represented by their constructors.

MutualTS The MutualTS framework combining Transformer and TRANX-Trans is shown in Figure 3, which contains a shared Transformer encoder and two Transformer decoders using different paradigms. The encoder is utilized to acquire semantic representations of the NL input, while the Seq2Seq end decoder is used to generate a sequence of code tokens and the Seq2Tree end decoder is used to generate a sequence of tree-construction actions. The two decoders use the same paradigms as the Seq2Seq models and the Seq2Tree models, respectively. By combining these two paradigms, the MutualTS framework can be enhanced in two ways. First, the two decoders achieve bidirectional information transfer through mutual learning, which helps each decoder obtain knowledge from the other. In addition, the shared encoder can receive feedback from both decoders simultaneously, which will enhance its encoding ability, enabling the encoder to consider the two different modes of code.

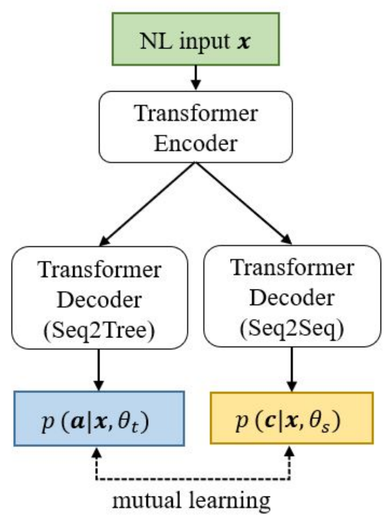


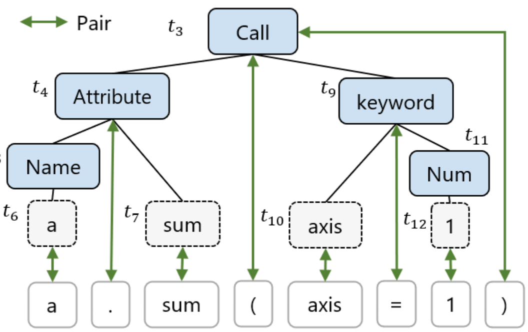
Figure 3: The mutual learning framework for the Seq2Seq and Seq2Tree models with a shared encoder.

3.2 Align generation steps and probability spaces

In order to realize the mutual learning between the Seq2Seq and Seq2Tree methods, we first need to establish the relationship between the generation steps of the two decoders. The Seq2Seq decoder generates one token per step in the code snippet, while the Seq2Tree decoder generates one action, i.e., one AST node, per step to build the AST. Therefore, we have to find a mapping between tokens in the code snippet and nodes in the AST.

We implement a tool to add the corresponding AST node information when converting the AST into a list of code tokens for Python (see Figure 4(a)). It is implemented based on `astor`[1], a Python library for round-trip an AST back to source code. For the input AST, the tool traverses the nodes, generates the corresponding token, and puts the token in the output list along with the current node number. The result is a list of code tokens and corresponding node numbers (see Figure 4(b)).

[1]https://pypi.org/project/astor/



(a)

Action	t_6	t_4	t_7	t_3	t_{10}	t_9	t_{12}	t_3
Token	a	.	sum	(	axis	=	1	)
Use	√	√	√	x	√	x	√	x

(b)

Figure 4: Process for creating mutual learning pairs for the example in Figure 1. (a) Pair code tokens with the corresponding AST nodes, noting that some AST nodes are omitted. (b) Determine whether a pair is used for mutual learning.

In addition, mutual learning requires calculating the difference between the probability distributions of two decoders, which means the output probability spaces of two decoders need to be the same. However, the output probability space of the Seq2Tree decoder is consistent with the Seq2Seq decoder only when predicting GenToken actions. As shown in Figure 4(b), when the elements of pairs of (‘a’, GenToken[a]), (‘sum’, GenToken[sum]), (‘axis’, GenToken[axis]), and (‘1’, GenToken[1]) are generated, the output probability spaces of the two decoders are consistent, and both are the code token vocabulary. However, when ApplyRule or Reduce actions are generated, the output probability space of the Seq2Tree decoder is the grammar rule table, and these actions may correspond to multiple tokens. For these actions, we try to correspond them to a unique synonym token that must appear in the (token, action) pairs corresponding to the action. Pairs that satisfy these conditions are also used for mutual learning. For example, n_4 in Figure 4 has the corresponding token ‘.’, and this token only corresponds to ApplyRule[Attribute]. This allows us to establish the probability space mapping (‘.’, ApplyRule[Attribute]), indicating that the two have similar semantics. Other mappings include (‘for’, ApplyRule[For]), (‘def’, ApplyRule[FunctionDef]), etc. As shown in Table 6, these nodes correspond to a code pattern, respectively, and the semantics of the nodes are basically the same as the corresponding tokens. Some actions do not have unique synonymous tokens, such as n_3 in Figure 4, whose corresponding action is ApplyRule[Call], and whose corresponding tokens are ‘(’ and ‘)’, which come in pairs and are widely used in code. The (token, node) pair corresponding to these actions will not be used for mutual learning. Therefore, we establish a one-to-one correspondence between a subset of the grammar rule list and a subset of the code token vocabulary. Please refer to Table 6 in Appendix for details.

Algorithm 1 The training procedure of MutualTS

Require: Training set D. Hyperparameters λ_g and λ_r
Ensure: The encoder parameters θ_e, the Seq2Tree decoder parameters θ_t and the Seq2Seq decoder parameters θ_s.
  Initial parameters θ_e, θ_t and θ_s.
  repeat
    for each batch $B \in D$ **do**
      $\mathcal{L}(B;\theta_e,\theta_t) = 0$
      $\mathcal{L}(B;\theta_e,\theta_s) = 0$
      for each instance $(\boldsymbol{x},\boldsymbol{a},\boldsymbol{c},\boldsymbol{\sigma}) \in B$ **do**
        Extract $\boldsymbol{\sigma_g}$ and $\boldsymbol{\sigma_r}$ from $\boldsymbol{\sigma}$
        $\mathcal{L}(B;\theta_e,\theta_s) \mathrel{+}= \mathcal{L}^{\text{MLE}}(\boldsymbol{x},\boldsymbol{c};\theta_e,\theta_s)$
          $+ \lambda_g \cdot \mathcal{L}_g^{\text{KL}}(\boldsymbol{x},\boldsymbol{c},\boldsymbol{\sigma_g};\theta_e,\theta_s|\boldsymbol{a};\theta_t)$
          $+ \lambda_r \cdot \mathcal{L}_r^{\text{KL}}(\boldsymbol{x},\boldsymbol{c},\boldsymbol{\sigma_r};\theta_e,\theta_s|\boldsymbol{a};\theta_t)$
        $\mathcal{L}(B;\theta_e,\theta_t) \mathrel{+}= \mathcal{L}^{\text{MLE}}(\boldsymbol{x},\boldsymbol{a};\theta_e,\theta_t)$
          $+ \lambda_g \cdot \mathcal{L}_g^{\text{KL}}(\boldsymbol{x},\boldsymbol{a},\boldsymbol{\sigma_g};\theta_e,\theta_t|\boldsymbol{c};\theta_s)$
          $+ \lambda_r \cdot \mathcal{L}_r^{\text{KL}}(\boldsymbol{x},\boldsymbol{a},\boldsymbol{\sigma_r};\theta_e,\theta_t|\boldsymbol{c};\theta_s)$
      end for
      Update θ_e,θ_s to minimize $\mathcal{L}(B;\theta_e,\theta_s)$
      Update θ_e,θ_t to minimize $\mathcal{L}(B;\theta_e,\theta_t)$
    end for
  until convergence

3.3 Mutual learning between Seq2Seq and Seq2Tree

After determining (token, action) pairs, probability space correspondence, and the details of MutualTS, this subsection describes the training method of mutual learning. In order for each decoder to learn from the other decoder, MutualTS performs bidirectional knowledge transfer at specific training steps that correspond to the generation of (node, token) pairs. Specifically, MutualTS is trained through Kullback-Leibler (KL) divergence loss between the outputs of the Seq2Seq and Seq2Tree ends at corresponding (token, action) pairs. This training occurs concurrently with the use of maximum likelihood estimation (MLE) loss between decoder output and target code token or AST action sequence. Through the interaction of these loss functions, the encoder of MutualTS tends to learn to extract better intermediate feature representations that meet both code text generation and AST generation, while the two decoders can learn knowledge of different generation paradigms from each other.

The specific training procedure of MutualTS is shown in Algorithm 1, where θ_e, θ_s, and θ_t represent the parameter sets of the NL encoder, the Seq2Seq decoder, and the Seq2Tree decoder, respectively. $(\boldsymbol{x},\boldsymbol{a},\boldsymbol{c},\boldsymbol{\sigma})$ represents an instance in the dataset, $\boldsymbol{x}$ is NL description input, $\boldsymbol{a}$ is target action sequence, $\boldsymbol{c}$ is target code sequence corresponding to $\boldsymbol{a}$, and $\boldsymbol{\sigma}$ is the (token, action) pairs between $\boldsymbol{c}$ and $\boldsymbol{a}$. $\mathcal{L}$ represents the sum of the losses of the corresponding end in an epoch, where λ_g and λ_r are coefficients that are used to control the impact of different losses. For each training instance, we first extracted two subsets of $\boldsymbol{\sigma}$, namely $\boldsymbol{\sigma_g}$ and $\boldsymbol{\sigma_r}$. The set $\boldsymbol{\sigma_g} \subseteq \boldsymbol{\sigma}$ includes all pairs containing the GenToken action, while $\boldsymbol{\sigma_r} \subseteq \boldsymbol{\sigma}$ includes pairs containing the ApplyRule action and are included in the (token, action) comparison table. Then, three loss functions for each decoder are obtained.

For the Seq2Seq end, the MLE loss $\mathcal{L}^{\text{MLE}}(\boldsymbol{x}, \boldsymbol{c}; \theta_e, \theta_s)$ is calculated by target action sequence $\boldsymbol{a}$ as follows:

$$\mathcal{L}^{\text{MLE}} = -\sum_{i=1}^{T} \log\ p(c_i|c_{<i}, \boldsymbol{x}, \theta_s), \tag{8}$$

The two KL loss $\mathcal{L}_g^{\text{KL}}(\boldsymbol{x}, \boldsymbol{c}, \boldsymbol{\sigma_g}; \theta_e, \theta_s|\boldsymbol{a}; \theta_t)$ and $\mathcal{L}_r^{\text{KL}}(\boldsymbol{x}, \boldsymbol{c}, \boldsymbol{\sigma_r}; \theta_e, \theta_s|\boldsymbol{a}; \theta_t)$ of Seq2Tree end are calculated as follows:

$$\mathcal{L}_g^{\text{KL}} = \sum_{(i,j)\in\sigma_g} \text{KL}_1(p_a(j)\|p_c(i)), \tag{9}$$

$$\mathcal{L}_r^{\text{KL}} = \sum_{(i,j)\in\sigma_r} \text{KL}_2(f_{a\mapsto c}(p_a(j))\|p_c(i)), \tag{10}$$

where (i, j) denotes the generation step of a (token, action) pair, respectively. $p_c(i) = p(\cdot|c_{<i}, \boldsymbol{x}; \theta_s)$ represents the probability distribution of Seq2Seq model at step i, $p_a(j) = p(\cdot|a_{<j}, \boldsymbol{x}; \theta_t)$ represents the probability distribution of Seq2Tree model at step j. $\text{KL}_1(\cdot\|\cdot)$ and $\text{KL}_2(\cdot\|\cdot)$ are KL divergence, the probability space of the former is the total code token vocabulary, and the latter's is code tokens in the (token, action) comparison table. Correspondingly, $f_{a\mapsto c}$ is used to convert the probability distribution on the grammar rule table to that on the code token vocabulary, based on the (token, action) comparison table. Similarly, for the Seq2Tree end, we have

$$\mathcal{L}^{\text{MLE}} = -\sum_{j=1}^{T} \log p(a_j|a_{<j}, \boldsymbol{x}, \theta_t), \tag{11}$$

$$\mathcal{L}_g^{\text{KL}} = \sum_{(i,j)\in\sigma_g} \text{KL}_1(p_c(i)\|p_a(j)), \tag{12}$$

$$\mathcal{L}_r^{\text{KL}} = \sum_{(i,j)\in\sigma_r} \text{KL}_2(p_c(i)\|f_{a\mapsto c}(p_a(j))), \tag{13}$$

The training steps are repeated until both the Seq2Seq and Seq2Tree end converge. After training, we combine the Seq2Seq end or the Seq2Tree end with the encoder to form an independent model for prediction.

4 EXPERIMENT SETUP

4.1 Datasets

To demonstrate the general validity of our method, we carry out experiments on three datasets. In addition to the benchmark dataset **CoNaLa**, we also constructed two datasets, named **JuICe_10k** and **JuICe_tiny**, based on the JuICe dataset [1].

CoNaLa[33]. This dataset consists of 2,879 examples of manually annotated NL questions and their Python solutions on StackOverflow. The examples in CoNaLa cover real-world NL queries issued by programmers with different intentions, with broad coverage and high composability.

JuICe_10k. The validation and test sets of this dataset are consistent with the JuICe dataset, and the train set contains 10,000 random samples from the JuICe train set. JuICe [1] is an open-domain large-scale dataset of over 659K publicly available Jupyter notebooks from Github, together with a manually curated evaluation set based on nbgrader [9]. Due to the high demand for the JuICe dataset on model size and training resources, we do not use the full JuICe dataset but divide a subset to reduce complexity. This dataset, like the JuICe dataset, is more challenging because of the extensive coverage nature of the training set and the domain differences between the training set and evaluation set.

JuICe_tiny. This dataset is a repartition of the JuICe dataset's validation set and test set, which are manually collected in-class programming assignment notebooks with solutions. This dataset is of higher quality than the noisy and large train set of the JuICe dataset. Besides, the training and evaluation set of JuICe_tiny belong to the same domain. These characteristics make models perform better on JuICe_tiny dataset than on the JuICe_10k dataset.

Table 1: Statistics of datasets

Dataset	CoNaLa	JuICe_tiny	JuICe_10k
Train Num	2,175	3,346	10,000
Dev Num	200	300	1,831
Test Num	500	300	2,115
Avg NL Tokens	10.2	58.0	40.4
Avg Code Tokens	14.9	38.4	43.4
Avg AST Nodes	23.2	55.2	58.8
$\boldsymbol{\sigma_g}$(%)	29.2	30.2	32.7
$\boldsymbol{\sigma_r}$(%)	6.4	7.0	6.6
Reduce(%)	23.7	23.4	20.9

The statistics of the three datasets are shown in Table 1. The values of $\boldsymbol{\sigma_g}$ and $\boldsymbol{\sigma_r}$ represent the average proportions of GenToken actions and ApplyRule actions contained in pairs for mutual learning in generating AST, respectively. The statistical values of Reduce represent the average proportions of Reduce actions in generating AST, which do not correspond to AST nodes. Our mutual learning framework can cover about 50% of AST nodes.

4.2 Baselines

The mutual learning framework we proposed can combine different Seq2Seq models and Seq2Tree models. In addition to the traditional TRANX model, we use the following models related to the models used in MutualTS as baselines.

TRANX-RL [13]. This method equips the TRANX model with a context-based branch selector. The selector is optimized by reinforcement learning to determine the optimal branching order of multi-branching nodes dynamically.

ML-TRANX [30]. This method uses mutual learning for different traversals-based decodings (depth-first preorder traversal vs. breadth-first traversal) of the TRANX model to improve the effect. ML-TRANX acts on AST, which is different from MutualTS integrating the two generation paradigms GSBP and LMBP. In addition, ML-Tranx calculates the KL loss at all AST generation actions without considering the alignment between AST and code.

Model	CoNaLa			JuICe-tiny			JuICe-10k
	CodeBLEU	BLEU	EM	CodeBLEU	BLEU	EM	BLEU
TRANX [33]	$26.80_{\pm 0.61}$	$^{\dagger}24.35_{\pm 0.4}$	$^{\dagger}2.5_{\pm 0.7}$	$18.06_{\pm 0.46}$	$12.36_{\pm 0.5}$	$1.2_{\pm 0.2}$	$4.63_{\pm 0.2}$
TRANX-RL [13]	$25.14_{\pm 0.96}$	$^{\dagger}25.47_{\pm 0.7}$	$^{\dagger}\mathbf{2.6}_{\pm 0.4}$	$19.79_{\pm 0.79}$	$13.85_{\pm 0.3}$	$1.0_{\pm 0.2}$	$6.08_{\pm 0.3}$
ML-TRANX [30]	$27.59_{\pm 0.98}$	$24.42_{\pm 0.8}$	$2.2_{\pm 0.4}$	$18.41_{\pm 0.51}$	$12.49_{\pm 0.6}$	$1.2_{\pm 0.3}$	$4.75_{\pm 0.4}$
Transformer [25]	$26.71_{\pm 0.94}$	$25.88_{\pm 0.5}$	$1.2_{\pm 0.3}$	$24.80_{\pm 0.38}$	$19.36_{\pm 0.2}$	$1.2_{\pm 0.2}$	$5.84_{\pm 0.1}$
TRANX-Trans	$27.31_{\pm 0.37}$	$26.50_{\pm 0.4}$	$1.8_{\pm 0.2}$	$22.62_{\pm 0.69}$	$16.89_{\pm 0.6}$	$2.1_{\pm 0.2}$	$4.80_{\pm 0.3}$
MutualTS(Seq2Seq)	$27.19_{\pm 0.50}$	$27.55_{\pm 0.2}$	$0.7_{\pm 0.3}$	$\mathbf{25.20}_{\pm 0.76}$	$\mathbf{19.59}_{\pm 0.1}$	$1.3_{\pm 0.0}$	$6.56_{\pm 0.1}$
MutualTS(Seq2Tree)	$\mathbf{28.33}_{\pm 0.71}$	$\mathbf{28.08}_{\pm 0.8}$	$2.5_{\pm 0.2}$	$25.01_{\pm 0.92}$	$19.52_{\pm 0.0}$	$\mathbf{2.2}_{\pm 0.3}$	$\mathbf{6.64}_{\pm 0.1}$

Table 2: Mean and standard deviation results of our model and baselines. All results were obtained from at least five experiments. † indicates the scores are previously reported ones.

TRANX-Trans. TRANX model implemented with Transformer structure, which generates a sequence of actions consistent with the TRANX model. The structure of the model is equivalent to removing the Seq2Seq decoder from the MutualTS framework, and the parameter settings are consistent with MutualTS.

Transformer [25]. A Seq2Seq model generating code token sequence. This model is equivalent to removing the Seq2Tree decoder from the MutualTS framework, and the parameter settings are consistent with MutualTS. Both the TRANX-Trans model and the Transformer model are used to explore the effectiveness of mutual learning methods.

4.3 Evaluation Metrics

Following the previous studies [8, 13, 28], we used three evaluation metrics including CodeBLEU, BLEU, and EM. CodeBLEU [21] is designed to evaluate code considering syntax and semantics. BLEU evaluates text based on n-gram similarity. We use a smooth BLEU-4 score similar to previous studies [13, 34]. EM means the proportion of exactly matched results.

4.4 Hyperparameters

We set the Transformer encoder and decoder parameters in our model as follows: (1) the number of layers $L = 6$. (2) the number of heads $H = 4$. (3) the feed-forward dimension $D_{FF} = 1024$. (4) the embedded dimension of actions and code tokens $D_{action} = D_{code} = 256$. (5) the embedded dimension of composite frontier field and field type $D_{field} = D_{type} = 64$. The weight coefficients λ_g and λ_r of loss functions are both set to 1. For decoding, the beam sizes for CoNaLa, JuICe_tiny, JuICe_10k are 15, 5, and 5, respectively. We add the pointer network [26] to the decoder as same as our baselines. In our experiment, we use one Tesla V100 GPU for training.

5 RESULTS

In this section, we evaluate the MutualTS framework in order to answer the following research questions:

- RQ1: How effective is the MutualTS framework in code generation task?
- RQ2: How effective are the different components in the MutualTS framework?
- RQ3: How effective is mutual learning between two different generation paradigms compared with the mutual learning within one paradigm?

RQ1: Compare MutualTS framework with baselines

The comparison results are shown in Table 2, including the mean and standard deviation. MutualTS(Seq2Seq) and MutualTS(Seq2Tree) represent the Seq2Seq and Seq2Tree ends of our framework. As seen, MutualTS exceeds all baselines in almost all metrics. The only exception was the exact match (EM) metrics on the CoNaLa dataset, in which the Seq2Tree end of MutualTS has a result 0.1 percent lower than TRANX-RL but better than most other baselines.

Comparing MutualTS with Transformer and TRANX-Trans shows that the mutual learning approach effectively combines the two models and improves their performance. Interestingly, the EM metrics on the seq2seq end are weaker than on the seq2tree end. This may be due to subtle differences between the code sequence transformed by AST and the original code, for example, the use of parentheses, even though they have the same function. Conversely, the training AST samples of the Seq2Tree end are consistent with the original data, and it achieves better effects on EM metrics through learning training data and mutual learning.

In addition, the CodeBLEU and BLEU metrics of the Seq2Tree and Seq2Seq ends were improved synchronously. Moreover, the comparison between the two ends is consistent with the comparison between the standalone Seq2Seq model (Transformer) and Seq2Tree model (TRANX-Trans). For example, in the JuICe-tiny dataset, the CodeBLEU metric of TRANX-Trans is lower than that of Transformer. Through mutual learning, the CodeBLEU metric of Seq2Tree and Seq2Seq ends reach a consistent level and are higher than models using a single paradigm. Moreover, the Seq2Seq end performs better than the Seq2Tree end. It shows that the mutual learning framework effectively combines the two paradigms with different output data forms, and can improve the model generation effect.

Table 3: Ablation study on mutual learning components. JointTS: Only share encoder. MutualTS$_g$: Mutual learning on (GenToken, token) pairs. MutualTS$_r$: Mutual learning on (ApplyRule, token) pairs.

Model	CoNaLa BLEU	JuICe-tiny BLEU
JointTS(Seq2Seq)	$25.99_{\pm 0.7}$	$19.29_{\pm 0.1}$
JointTS(Seq2Tree)	$27.41_{\pm 0.5}$	$19.23_{\pm 0.3}$
MutualTS$_g$(Seq2Seq)	$26.57_{\pm 0.1}$	$\mathbf{20.30}_{\pm 0.7}$
MutualTS$_g$(Seq2Tree)	$27.78_{\pm 0.4}$	$20.29_{\pm 0.2}$
MutualTS$_r$(Seq2Seq)	$26.88_{\pm 0.5}$	$19.44_{\pm 0.4}$
MutualTS$_r$(Seq2Tree)	$27.84_{\pm 0.5}$	$19.54_{\pm 0.7}$
MutualTS(Seq2Seq)	$27.55_{\pm 0.2}$	$19.59_{\pm 0.1}$
MutualTS(Seq2Tree)	$\mathbf{28.08}_{\pm 0.8}$	$19.52_{\pm 0.0}$

Table 4: Ablation study on different paradigms. MutualTT: Mutual learning between two Seq2Tree models. MutualSS: Mutual learning between two Seq2Seq models.

Model	CoNaLa BLEU	JuICe-tiny BLEU
MutualSS	$27.07_{\pm 0.2}$	$19.51_{\pm 0.4}$
MutualTT	$26.88_{\pm 0.6}$	$16.60_{\pm 0.4}$
MutualTS(Seq2Seq)	$27.55_{\pm 0.2}$	$\mathbf{19.59}_{\pm 0.1}$
MutualTS(Seq2Tree)	$\mathbf{28.08}_{\pm 0.8}$	$19.52_{\pm 0.0}$

RQ2: Effects of different mutual learning components

As described in section 3, MutualTS has two loss functions based on mutual learning, $\mathcal{L}_g^{\text{KL}}$ and $\mathcal{L}_r^{\text{KL}}$, corresponding to GenToken and ApplyRule actions generated by the Seq2Tree end. We designed three variants of MutualTS to explore the role of these components. The **JointTS** model removes $\mathcal{L}_g^{\text{KL}}$ and $\mathcal{L}_r^{\text{KL}}$ and still share encoder. The **MutualTS$_g$** model uses $\mathcal{L}_g^{\text{KL}}$ only. The **MutualTS$_r$** model uses $\mathcal{L}_r^{\text{KL}}$ only. The remaining settings of these three variants are consistent with the original model.

The ablation results of mutual learning loss functions are shown in Table 3. The additional content after the model name indicates whether the result is output by the Seq2Seq or Seq2Tree end of the model. The results show that the two kinds of mutual learning components can improve the effect independently, but their effects are not orthogonal. Compared with JointTS(Seq2Tree), the BLEU metrics of the MutualTS(Seq2Tree) improve by 0.67, which is lower than the sum of effect improvements of MutualTS$_g$(Seq2Tree) and MutualTS$_r$(Seq2Tree). In addition, the JointTS model shows an improvement over the separately trained models, indicating that a degree of knowledge fusion can be achieved through the shared encoder.

RQ3: Effects of interacting two different paradigms of code generation

To verify the interaction effect of two different paradigms (GSBP & LMBP), we designed two variants of MutualTS that apply mutual learning between models using a single paradigm. The **MutualTT** model combines two different initialized Seq2Tree models and shares the encoder. The **MutualSS** model combines two different initialized Seq2Seq models and shares the encoder. The hyperparameter settings of these models are consistent with MutualTS.

The ablation results are shown in Table 4. On the two datasets, mutual learning between two identical models with different initialization also helps to improve the generalization, and the MuTualTS model performs better than these models. The results show that the Seq2Seq end and the Seq2Tree end learn different features, and they can benefit from each other. Moreover, code generation can be improved by using both language model-based and grammatical structure-based paradigms.

6 DISCUSSION

6.1 Qualitative Analysis

In this section, we aim to analyze the specific ways in which mutual learning enhances the performance of models. To perform a thorough evaluation, we use a token-level code completion task rather than the code generation task used in previous experiments. Code completion tasks are widely used to measure the generation capability of a model on a single token or short code snippet [8, 14, 17]. In our experiments, the model is asked to predict the next token or action given all previous tokens or actions and the NL description. This allows us to make token-level comparisons and accurately assess the ability of different models to predict specific tokens. All experiments in this section are conducted using the test set of the CoNaLa dataset.

Table 5: Results of average MLE loss on code completion tasks for different models

Model	CoNaLa loss
Transformer	2.33
MutualTS(Seq2Seq)	**1.82**
TRANX-Trans	2.88
MutualTS(Seq2Tree)	**2.32**

The evaluation of generation performance is based on the probability of correct token predictions. Specifically, we calculate the MLE loss at specific token positions, which is negatively correlated with the correct probability. In the rest of this section, "loss" refers specifically to the MLE loss. As described in Section 3.3, let the parameter sets of a model be θ, the input be x and the current output step be t. That is, $y_{<t}$ represents the previous sequence and y_t represents the current target. The relationship between the current step loss l_t and the correct probability $p(y_t|y_{<t}, x)$ is $l_t = -\log p(y_t|y_{<t}, x)$. Table 5 shows the average loss of different models. The results show that models trained using our MuTualTS framework have a higher probability of correctly predicting the next token, which

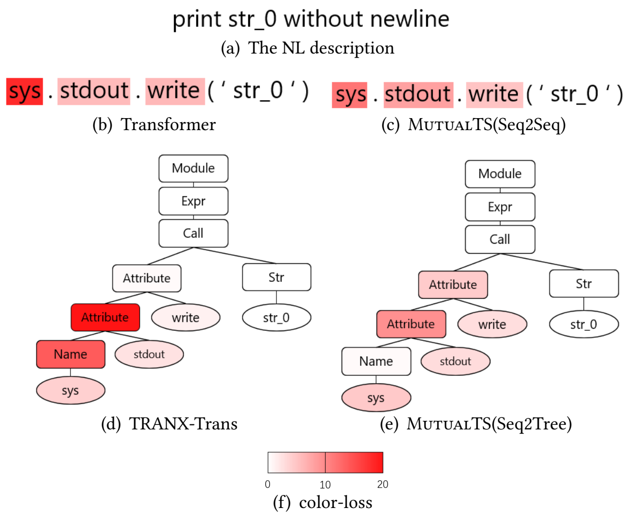


Figure 5: Results of example 1. (a) The input NL description. (b)-(e) The loss of different models on code tokens or AST nodes. The background red color of a token/node is proportional to the loss when predicting it. (f) The correspondence between color and loss.

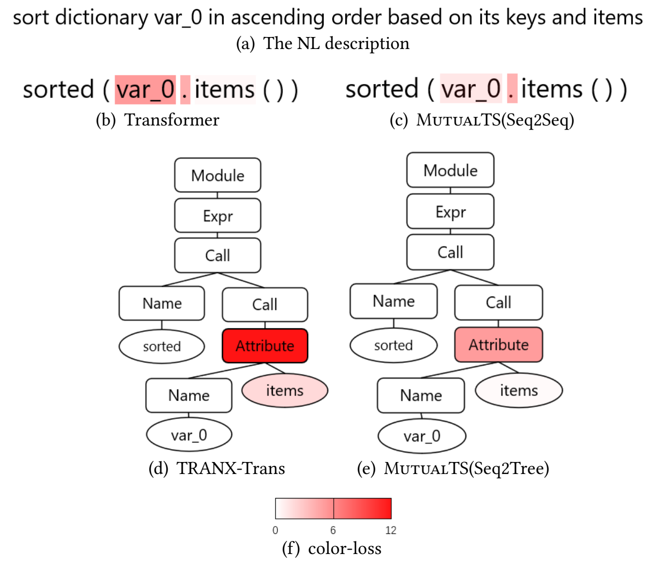


Figure 6: Results of example 2.

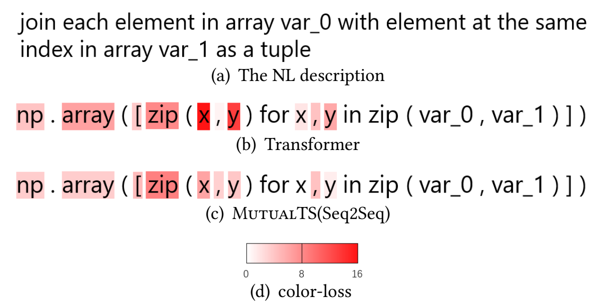


Figure 7: Results of example 3.

is consistent with the experimental results reported earlier in this paper.

Through detailed analysis, we have found that mutual learning improves the probability of correct predictions in a general way, but it is particularly effective in some cases. When models using different paradigms have different generation orders, they may have very different performance on specific code tokens. In these cases, mutual learning can integrate the knowledge of both models and significantly improve performance. Here are some examples to illustrate this.

The visualization results of example 1 are shown in Figure 5. The deeper the red background in a token/node, the greater the loss, that is, the lower the correct probability of the model prediction at this token/node. By comparing the loss difference between different models at the same position, we can analyze the specific ways in which mutual learning improves model performance. For Transformer using LMBP, the loss of the token '`sys`' is much higher than that of TRANX-Trans when predicting the `sys` node (the `sys` node is corresponding to the token '`sys`'), and is reduced from 18.17 to 11.76 through mutual learning. On the other hand, when predicting the second `Attribute` node and the subsequent `Name` node, TRANX-Trans has higher losses than that of Transformer when predicting the corresponding first '`.`' token, and these losses are reduced significantly through mutual learning. Some people may argue that models without mutual learning are attempting to generate correct code that is different from the target code, so mutual learning has not really been effective. However, we disagree with this perspective. The training data for the models is completely the same, so if a model without mutual learning wants to generate a correct but different code, it should still try to generate the same code after mutual learning. Whether mutual learning is used should not affect the losses at these tokens. In practice, however, mutual learning greatly reduces the losses at these tokens.

We believe that one of the reasons is that it is harder for a decoder to model underlying semantic dependency from back to front, compared to that from front to back. Here we say a code fragment c_1 semantically depends on another code fragment c_2, if the model needs some information from the code fragment c_2 when generating the code fragment c_1. As a result, models with different generation orders have different performances on learning dependencies, leading to differences in their predictions on specific tokens. As shown in example 1, the `Attribute` and `Name` nodes are not directly related to the input, but rather semantically depend on the function name '`write`' and package prefixes '`sys`' and '`stdout`'. However, those nodes are generated later, and when training the Seq2Tree model on nodes like `Attribute` and `Name`, those related nodes are not visible to the model yet. On the other hand, when generating the first '`.`' token, a Seq2Seq model can see '`sys`', so this dependency is easy to learn for the Seq2Seq model. Through the knowledge interaction of mutual learning, the Seq2Tree model can also better model this dependency, causing the loss to decrease.

Similarly, the '`sys`' token is a second-level package prefix for the '`write`' function, and is related to the subsequent "`.stdout.write()`"

fragment. However, the Seq2Seq model needs to generate 'sys' token first, making it difficult to model this underlying dependency. On the other hand, when generating the 'sys' node, the Seq2Tree model can see ancestor nodes `Call` and two `Attribute`, and this information can be transferred to the Seq2Seq model through mutual learning, improving its performance at 'sys' token.

Other examples include example 2 in Figure 6 and example 3 in Figure 7. Through mutual learning, high losses have been significantly reduced. At these positions, underlying semantic dependencies are from back to front in one paradigm, but they are from front to back in another. The MutualTS framework improves the correct probability of models at these positions, demonstrating that one of the advantages of combining different paradigms is to help the model more comprehensively learn underlying dependencies. Furthermore, this also inspires us to design methods that enhance the ability of code generation models to learn underlying dependencies from back to front better.

6.2 Threats to Validity

There are two main threats to the validity of our MutualTS framework. Firstly, we conduct experiments on three Python datasets. Our findings suggest that mutual learning between models using different paradigms can enhance performance and is particularly effective in some situations. But further research is needed to determine the generalizability of our results to other programming languages. For other programming languages with open-source ASDL grammar, it is possible to design the corresponding Seq2Tree model and align the generation steps of different paradigms, but with a lot of engineering effort. In future work, we plan to extend our study to include a broader range of programming languages. Secondly, We choose two typical models for mutual learning, neither of which is a pre-training model. It is exciting to explore whether our approach would also be effective in pre-training models. However, the pre-training generation model using GSBM has not been studied, so we cannot use the pre-training model as the Seq2Tree end. Additionally, there is a significant performance gap between pre-training models and other models. Therefore, only using a pre-training model as the Seq2Seq end would not achieve our purpose, that is, to analyze whether combining different paradigms can improve the performance of code generation models. In summary, our results are limited and we plan to conduct further experiments with pre-training models in the future.

7 CONCLUSION

This paper first argues that code generation should consider both the language model-based paradigm and grammatical structure-based paradigm used by Seq2Tree and Seq2Seq models, respectively. Then this paper proposes a mutual learning framework combining the Seq2Tree and Seq2Seq models to realize knowledge interaction between them. The results of main experiments and ablation studies show that combining the two code generation paradigms is effective.

ACKNOWLEDGMENTS

We thank all reviewers for their constructive comments. This research is supported by the National Key R&D Program under Grant No.2021ZD0110303, the National Natural Science Foundation of China under Grant No. 62072007, 62192733, 61832009, 62192731, 62192730.

REFERENCES

[1] Rajas Agashe, Srinivasan Iyer, and Luke Zettlemoyer. 2019. JuICe: A Large Scale Distantly Supervised Dataset for Open Domain Context-based Code Generation. In *EMNLP/IJCNLP (1)*. Association for Computational Linguistics, 5435–5445.
[2] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified Pre-training for Program Understanding and Generation. In *NAACL-HLT*. Association for Computational Linguistics, 2655–2668.
[3] Ruisheng Cao, Su Zhu, Chen Liu, Jieyu Li, and Kai Yu. 2019. Semantic Parsing with Dual Learning. In *ACL (1)*. Association for Computational Linguistics, 51–64.
[4] Huimin Chen, Yankai Lin, Fanchao Qi, Jinyi Hu, Peng Li, Jie Zhou, and Maosong Sun. 2021. Aspect-Level Sentiment-Controllable Review Generation with Mutual Learning Framework. In *AAAI*. AAAI Press, 12639–12647.
[5] Li Dong and Mirella Lapata. 2016. Language to Logical Form with Neural Attention. In *ACL (1)*. The Association for Computer Linguistics.
[6] Yihong Dong, Xue Jiang, Zhi Jin, and Ge Li. 2023. Self-collaboration Code Generation via ChatGPT. *CoRR* abs/2304.07590 (2023).
[7] Yihong Dong, Ge Li, and Zhi Jin. 2022. CODEP: Grammatical Seq2Seq Model for General-Purpose Code Generation. *CoRR* abs/2211.00818 (2022).
[8] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. UniXcoder: Unified Cross-Modal Pre-training for Code Representation. In *ACL (1)*. Association for Computational Linguistics, 7212–7225.
[9] Jessica B. Hamrick. 2016. Creating and Grading IPython/Jupyter Notebook Assignments with NbGrader. In *SIGCSE*. ACM, 242.
[10] Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar T. Devanbu. 2012. On the naturalness of software. In *ICSE*. IEEE Computer Society, 837–847.
[11] Peixian Hong, Tao Wu, Ancong Wu, Xintong Han, and Wei-Shi Zheng. 2021. Fine-Grained Shape-Appearance Mutual Learning for Cloth-Changing Person Re-Identification. In *CVPR*. Computer Vision Foundation / IEEE, 10513–10522.
[12] Robin Jia and Percy Liang. 2016. Data Recombination for Neural Semantic Parsing. In *ACL (1)*. The Association for Computer Linguistics.
[13] Hui Jiang, Chulun Zhou, Fandong Meng, Biao Zhang, Jie Zhou, Degen Huang, Qingqiang Wu, and Jinsong Su. 2021. Exploring Dynamic Selection of Branch Expansion Orders for Code Generation. In *ACL/IJCNLP (1)*. Association for Computational Linguistics, 5076–5085.
[14] Seohyun Kim, Jinman Zhao, Yuchi Tian, and Satish Chandra. 2021. Code Prediction by Feeding Trees to Transformers. In *ICSE*. IEEE, 150–162.
[15] Tomasz Korbak, Hady Elsahar, Marc Dymetman, and Germán Kruszewski. 2021. Energy-Based Models for Code Generation under Compilability Constraints. *CoRR* abs/2106.04985 (2021).
[16] Wang Ling, Phil Blunsom, Edward Grefenstette, Karl Moritz Hermann, Tomás Kociský, Fumin Wang, and Andrew W. Senior. 2016. Latent Predictor Networks for Code Generation. In *ACL (1)*. The Association for Computer Linguistics.
[17] Fang Liu, Jia Li, and Li Zhang. 2023. Syntax and Domain Aware Model for Unsupervised Program Translation. In *ICSE*. IEEE, 755–767.
[18] Sajad Norouzi, Keyi Tang, and Yanshuai Cao. 2021. Code Generation from Natural Language with Less Prior Knowledge and More Monolingual Data. In *ACL/IJCNLP (2)*. Association for Computational Linguistics, 776–785.
[19] Maxim Rabinovich, Mitchell Stern, and Dan Klein. 2017. Abstract Syntax Networks for Code Generation and Semantic Parsing. In *ACL (1)*. Association for Computational Linguistics, 1139–1149.
[20] Veselin Raychev, Martin T. Vechev, and Eran Yahav. 2014. Code completion with statistical language models. In *PLDI*. ACM, 419–428.
[21] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. CodeBLEU: a Method for Automatic Evaluation of Code Synthesis. *CoRR* abs/2009.10297 (2020).
[22] Zeyu Sun, Qihao Zhu, Yingfei Xiong, Yican Sun, Lili Mou, and Lu Zhang. 2020. TreeGen: A Tree-Based Transformer Architecture for Code Generation. In *AAAI*. AAAI Press, 8984–8991.
[23] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. 2014. Sequence to Sequence Learning with Neural Networks. In *NIPS*. 3104–3112.
[24] Sindhu Tipirneni, Ming Zhu, and Chandan K. Reddy. 2022. StructCoder: Structure-Aware Transformer for Code Generation. *CoRR* abs/2206.05239 (2022).
[25] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *NIPS*. 5998–6008.
[26] Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. 2015. Pointer Networks. In *NIPS*. 2692–2700.
[27] Xin Wang, Yasheng Wang, Yao Wan, Fei Mi, Yitong Li, Pingyi Zhou, Jin Liu, Hao Wu, Xin Jiang, and Qun Liu. 2022. Compilable Neural Code Generation with Compiler Feedback. In *ACL (Findings)*. Association for Computational Linguistics, 9–19.

[28] Yue Wang, Weishi Wang, Shafiq R. Joty, and Steven C. H. Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In *EMNLP (1)*. Association for Computational Linguistics, 8696–8708.

[29] Bolin Wei, Ge Li, Xin Xia, Zhiyi Fu, and Zhi Jin. 2019. Code Generation as a Dual Task of Code Summarization. In *NeurIPS*. 6559–6569.

[30] Binbin Xie, Jinsong Su, Yubin Ge, Xiang Li, Jianwei Cui, Junfeng Yao, and Bin Wang. 2021. Improving Tree-Structured Decoder Training for Code Generation via Mutual Learning. In *AAAI*. AAAI Press, 14121–14128.

[31] Frank F. Xu, Zhengbao Jiang, Pengcheng Yin, Bogdan Vasilescu, and Graham Neubig. 2020. Incorporating External Knowledge through Pre-training for Natural Language to Code Generation. In *ACL*. Association for Computational Linguistics, 6045–6052.

[32] Pengcheng Yin and Graham Neubig. 2017. A Syntactic Neural Model for General-Purpose Code Generation. In *ACL (1)*. Association for Computational Linguistics, 440–450.

[33] Pengcheng Yin and Graham Neubig. 2018. TRANX: A Transition-based Neural Abstract Syntax Parser for Semantic Parsing and Code Generation. In *EMNLP (Demonstration)*. Association for Computational Linguistics, 7–12.

[34] Pengcheng Yin and Graham Neubig. 2019. Reranking for Neural Semantic Parsing. In *ACL (1)*. Association for Computational Linguistics, 4553–4559.

[35] Ying Zhang, Tao Xiang, Timothy M. Hospedales, and Huchuan Lu. 2018. Deep Mutual Learning. In *CVPR*. Computer Vision Foundation / IEEE Computer Society, 4320–4328.

[36] Jiawei Zhao, Wei Luo, Boxing Chen, and Andrew Gilman. 2021. Mutual-Learning Improves End-to-End Speech Translation. In *EMNLP (1)*. Association for Computational Linguistics, 3989–3994.

A COMPARISON TABLE

Table 6: (token, ApplyRule action) comparison table.

Action	Code pattern	Token
stmt → FunctionDef(identifier name, arguments args, stmt* body, expr* decorator_list, expr? returns)	@[decorator_list] def [name]([args])->[returns]: [body]	def
stmt → ClassDef(identifier name, expr* bases, keyword* keywords, stmt* body, expr* decorator_list)	@[decorator_list] class [name]([bases], [keywords]): [body]	class

stmt → For(expr target, expr iter, stmt* body, stmt* orelse)	for [target] in [iter]:
	[body]
	else:
	[orelse]
		for

stmt → While(expr test, stmt* body, stmt* orelse)	while [test]:
	[body]
	else:
	[orelse]
		while

stmt → If(expr test, stmt* body, stmt* orelse)	if [test]:
	[body]
	else:
	[orelse]
		if

stmt → With(withitem* items, stmt* body)	with [items]: [body]	with
stmt → Raise(expr? exc, expr? cause)	raise [exc] from [cause]	raise

stmt → Try(stmt* body, excepthandler* handlers, stmt* orelse, stmt* finalbody)	try:
	[body]
	[handlers]
	else:
	[orelse]
	finally:
	[finalbody]
		try

stmt → Return(expr? value)	return [value]	return
stmt → Delete(expr* targets)	del [targets]	del
stmt → Assert(expr test, expr? msg)	assert [test], [msg]	assert
stmt → Import(alias* names)	import [names]	import
stmt → ImportFrom(identifier? module, alias* names, int? level)	from [module] import [names]	from
stmt → Global(identifier* names)	global [names]	global
stmt → Nonlocal(identifier* names)	nonlocal [names]	nonlocal
stmt → Pass()	pass	pass
stmt → Break()	break	break
stmt → Continue()	continue	continue
expr → Await(expr value)	await [value]	await
expr → Lambda(arguments args, expr body)	lambda [args]: [body]	lambda
expr → Ellipsis()	...	...
expr → Attribute(expr value, identifier attr)	[value].[attr]	.
boolop → And()	and	and
boolop → Or()	or	or
operator → Add()	+	+
operator → Sub()	-	-
operator → Mult()	*	*
operator → MatMult()	@	@
operator → Div()	/	/
operator → Mod()	%	%
operator → Pow()	**	**
operator → LShift()	<<	<<
operator → RShift()	>>	>>
operator → BitOr()	|	|
operator → BitXOr()	^	^
operator → BitAnd()	&	&
operator → FloorDiv()	//	//
unaryop → Invert()	~	~
unaryop → Not()	not	not
cmpop → Eq()	==	==
cmpop → NotEq()	!=	!=
cmpop → Lt()	<	<
cmpop → LtE()	<=	<=
cmpop → Gt()	>	>
cmpop → GtE()	>=	>=
cmpop → Is()	is	is
cmpop → In()	in	in
excepthandler → ExceptHandler(expr? type, identifier? name, stmt* body)	except [type] as [name]: [body]	except