CODEP: Grammatical Seq2Seq Model for General-Purpose Code Generation

Yihong Dong
Key Lab of High Confidence Software Technology, MoE (Peking University)
Beijing, China
dongyh@stu.pku.edu.cn

Ge Li*
Key Lab of High Confidence Software Technology, MoE (Peking University)
Beijing, China
lige@pku.edu.cn

Zhi Jin
Key Lab of High Confidence Software Technology, MoE (Peking University)
Beijing, China
zhijin@pku.edu.cn

ABSTRACT

General-purpose code generation aims to automatically convert the natural language description to code snippets in a general-purpose programming language (GPL) such as Python. In the process of code generation, it is essential to guarantee the generated code satisfies grammar constraints of GPL. However, existing sequence-to-sequence (Seq2Seq) approaches neglect grammar rules when generating GPL code. In this paper, we devise a pushdown automaton (PDA)-based methodology to make the first attempt to consider grammatical Seq2Seq models for general-purpose code generation, exploiting the principle that PL is a subset of PDA recognizable language and code accepted by PDA is grammatical. Specifically, we construct a PDA module and design an algorithm to constrain the generation of Seq2Seq models to ensure grammatical correctness. Guided by this methodology, we further propose CODEP, a code generation framework equipped with a PDA module, to integrate the deduction of PDA into deep learning. This framework leverages the state of PDA deduction (including state representation, state prediction task, and joint prediction with state) to assist models in learning PDA deduction. To comprehensively evaluate CODEP, we construct a PDA for Python and conduct extensive experiments on four public benchmark datasets. CODEP can employ existing sequence-based models as base models, and we show that it achieves 100% grammatical correctness percentage on these benchmark datasets. Consequently, CODEP relatively improves 17% CodeBLEU on CONALA, 8% EM on DJANGO, and 15% CodeBLEU on JUICE-10K compared to base models. Moreover, PDA module also achieves significant improvements on the pre-trained models.

CCS CONCEPTS

• **Software and its engineering** → **Software creation and management**; • **Computing methodologies** → **Artificial intelligence**.

[*] Corresponding author.

The code of CODEP and PDA is avaliable on https://github.com/YihongDong/CODEP.

ISSTA '23, July 17–21, 2023, Seattle, WA, United States
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0221-1/23/07…$15.00
https://doi.org/10.1145/3597926.3598048

KEYWORDS

Code generation, Code intelligence, Seq2Seq, PDA, PL.

ACM Reference Format:
Yihong Dong, Ge Li, and Zhi Jin. 2023. CODEP: Grammatical Seq2Seq Model for General-Purpose Code Generation. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '23), July 17–21, 2023, Seattle, WA, United States.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3597926.3598048

1 INTRODUCTION

Code generation techniques are critical in artificial intelligence and software engineering, which aims to help facilitate software development and revolutionize user programming [7, 14, 19, 22, 26, 38–41, 44]. For example, in Fig. 1, a programmer wants to terminate the program under the specific condition “list *l* is empty" in Python, but does not know the relevant prior knowledge to realize their intention. By using code generation techniques, the target code “if not *l*: exit(0)” can be obtained with ease. A well-designed code generation approach requires consideration of grammar in addition to semantics, as semantics ensures the functionality of code meets developer's intention, while satisfying grammar is a basic requirement for programming. The two complement each other to produce a satisfactory code.

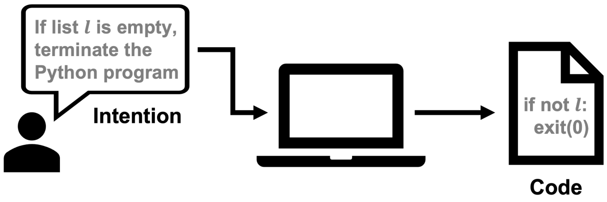


Figure 1: Illustration of code generation.

Seq2Seq approaches are the most commonly used in code generation [4, 11, 16, 23, 27, 38], which have the following three advantages: 1) High efficiency. Seq2Seq approach directly generates a token sequence of code along the order of human writing in one pass. 2) Convenient operation. It can obtain partially generated code with ease even if the result is incomplete or the generation is not finished, thus can be adopted for various code generation scenarios [1]. 3) Easy data accessibility. Sequence structured data is the

[1] For example, code completion, a popular variant of code generation in practice, which completes code with contexts.

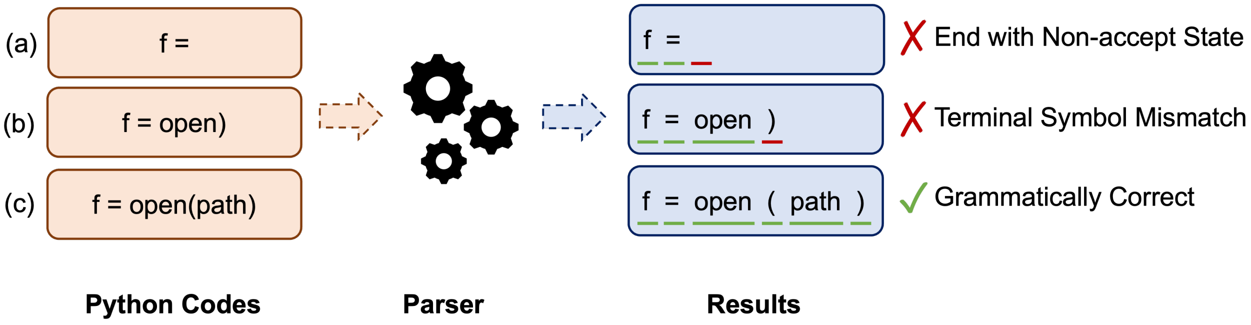


Figure 2: Examples of a parser parsing Python code.

most general form that can be obtained without much effort compared to other structured data. Thus, pre-trained methods with big data-driven usually employ Seq2Seq approaches [2, 8, 11, 14, 23, 38]. However, most Seq2Seq approaches generate codes with no guarantee of grammatical correctness (GC), which hinders the usability of generated codes.

Sequence-to-tree (Seq2Tree)-based approaches for code generation are generally used to ensure GC by deriving the syntactic structure [26], which outputs a node sequence of abstract syntax tree (AST) that can be further mapped to well-formed code. At a cost, they are inferior to Seq2Seq approaches in the preceding three aspects. Therefore, an ideal code generation approach could preserve the advantages of Seq2Seq approaches while ensuring GC.

According to automata theory, we find that PDA is suitable for PL grammar and has the following properties to deal with non-grammatical problems of Seq2Seq-based code generation. The first property is a valid terminal symbol set for next token prediction, and the second is an accept state set for recognizing completed code. In a practical Seq2Seq-based code generation process, what we really care about are two typical grammatical error: terminal symbol mismatch (TSM) error and end with non-accept state (ENS) error, where ENS error can be tolerable until the end of Seq2Seq-based code generation process. As a result, satisfying grammar constraints during Seq2Seq-based code generation require the preceding two properties, and constraining Seq2Seq-based code generation based on PDA to guarantee GC is a matter of course.

It is inspired by the operating principle of programming language parsers in practice. Fig. 2 demonstrates some examples of a parser parsing python code, which contains two specific grammatical error codes (a) and (b), and a grammatically correct code (c). Case (a) indicates ENS error, and case (b) indicates TSM error. All cases are recognized by a parser according to grammar. Therefore, a straightforward way is applying a parser to constrain Seq2Seq-based code generation, but it has the following two problems. First, Seq2Seq models need to recognize grammatical errors during code generation process. However, the vast majority of codes generated during this situation are incomplete, i.e., parsing these codes will produce ENS errors like case (a). Second, a parser can only recognize TSM errors like case (b) if terminal symbol is fed to the parser. It implies that Seq2Seq models need to attempt each generated code token in the candidate set one by one, which is strenuous [2]. In short, although code can be grammatically bounded by a parser, it is still challenging to work directly with a parser on Seq2Seq-based code generation.

In this paper, we propose CODEP (a CODE generation framework based on Pushdown automaton module) to ensure GC for Seq2Seq models. PDA module facilitates CODEP to generate bounded next prediction in a valid set and end with an accept state, utilizing the principle that PDA can recognize grammatical codes. CODEP not only ensures GC but also maintains the advantages of commonly used Seq2Seq-based code generation. We conduct a series of experiments on four public benchmark datasets. Extensive experimental results and analyses verify the effectiveness and generality of CODEP.

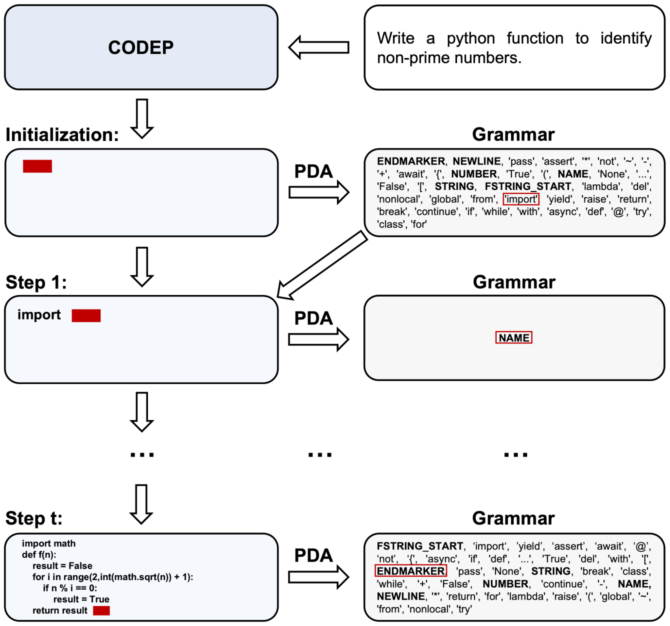
The main contribution of this paper can be summarized in fourfold:

- We devise a PDA-based methodology to guarantee grammatical correctness for code generation, which consists of a PDA module and an algorithm to simulate the deduction of PDA.
- We propose CODEP, a novel Seq2Seq-based code generation framework incorporating the PDA module. In addition to using the PDA-based methodology, CODEP leverages PDA state through state representation, state prediction task, and joint prediction with state, which helps learn PDA deduction.
- CODEP significantly enhances the performance of base models. Even in zero-shot setting, pre-trained models still show remarkable improvements.
- Under the premise of ensuring grammatical correctness, CODEP outperforms the state-of-the-art Seq2Tree models without pre-training.

The remainder of this paper is organized as follows: Section 2 demonstrates a motivation example of our work. Section 3 introduces the background and related work. Section 4 and 5 details the proposed PDA-based methodology and CODEP. Sections 6 and 7 describe the experimental setup and experimental results. Section 8 describes the major limitations of our work. Section 9 concludes this paper and discusses future work.

[2] For example, there are only 83 syntax-strings and 10 token-types in Python, and for a 10k-length vocabulary, more than 99% of tokens belong to three token-types NAME, STRING, and NUMBER. If models want to generate these types of tokens based on semantics but are prevented by grammar, the cost of attempts could be overwhelming.

2 MOTIVATION EXAMPLE


CODEP
Write a python function to identify non-prime numbers.
Initialization:
Grammar
PDA
ENDMARKER, NEWLINE, 'pass', 'assert', '*', 'not', '~', '-', '+', 'await', '{', NUMBER, 'True', '(', NAME, 'None', '...', 'False', '[', STRING, FSTRING_START, 'lambda', 'del', 'nonlocal', 'global', 'from', 'import' 'yield', 'raise', 'return', 'break', 'continue', 'if', 'while', 'with', 'async', 'def', '@', 'try', 'class', 'for'
Step 1:
import
Grammar
PDA
NAME
...
...
...
Step t:
Grammar
PDA
FSTRING_START, 'import', 'yield', 'assert', 'await', '@', 'not', '{', 'async', 'if', 'def', '...', 'True', 'del', 'with', '[', ENDMARKER, 'pass', 'None', STRING, 'break', 'class', 'while', '+', 'False', NUMBER, 'continue', '-', NAME, NEWLINE, '*', 'return', 'for', 'lambda', 'raise', '(', 'global', '~', 'from', 'nonlocal', 'try'

```
import math
def f(n):
    result = False
    for i in range(2,int(math.sqrt(n)) + 1):
        if n % i == 0:
            result = True
    return result
```

Figure 3: Motivation Example.

An example of Seq2Seq-based code generation constrained by PDA module is shown in Fig. 3. Given an natural language (NL) description, CODEP needs to generate code that satisfies its intended functionality with the guarantee of GC. PDA module gives a valid set of terminal symbols (i.e., candidate syntax-strings and token-types) depending on grammar constraints, to which the generated token of CODEP should belong. Specifically, in the beginning, CODEP wants to generate first token from the vocabulary of a model according to NL. Based on production rules, PDA module shrinks the candidate set from entire vocabulary to 35 syntax-strings and 6 token-types, where each candidate token is constrained by grammar. As a result, at step 1, CODEP picks a token (i.e., 'import') from the candidate set according to semantics. At each subsequent generation step, CODEP predicts a token in the same manner, which satisfies grammar constraints. Until the end of generation steps, CODEP outputs the token of ENDMARKER token-type (e.g., </s>), which ends with an accept state. Thus, CODEP generates a grammatical code with the help of PDA module.

In the way of the preceding example, we first introduce PDA into deep learning (DL), which drastically shrinks the candidate set at each step of generation and reduces the difficulty for the model to select tokens from it, thus ensuring GC and improving the quality of Seq2Seq-based code generation.

3 RELATED WORK

3.1 General-Purpose Code Generation

In recent years, two kinds of prevailing approaches have been used for general-purpose code generation.

3.1.1 Seq2Seq-based Code Generation. Ling et al. [22] considered the code generation task as a conditional text generation task and adopted the Seq2Seq model to address it. Some later work [4, 16] followed this generation approach. For example, the work [39] proposed a dual training framework to train both code generation and representation tasks simultaneously. CodeT5 [38], UniXcoder [14], Codegen [23], and InCoder [11] applied pre-trained models to code generation task. The work [38] proposed a unified pre-trained encoder-decoder transformer model, named CodeT5, to better exploit the code semantics conveyed by the developer-assigned identifiers. The work [14] proposed a unified cross-modal pre-trained model, named UniXcoder, for code generation, which used the mask attention matrix with prefix adapters to control the behavior of the model and leveraged cross-modal contents to enhance the code representations. In addition, the authors in [37] considered code compilability as a training objective based on pre-trained models, but it still fails to guarantee GC.

Although Seq2Seq-based approaches are most commonly used for code generation, they usually cannot guarantee GC.

3.1.2 Seq2Tree-based Code Generation. Dong et al. [6] first considered the Seq2Tree model for code generation. To exploit the grammatical information of the code, the work [43] and [26] adopted the encoder-decoder architecture to output the AST node sequence. On this basis, TRANX [44] was proposed and became an effective and widely used Seq2Tree model. Lots of work was based on and improved upon TRANX, such as ML-TRANX [40], TRANX-RL [18], ASED [17], and APT [9]. Moreover, the work [33] and [34] applied the Seq2Tree convolutional neural network and Seq2Tree Transformer for code generation. The work [32] proposed Subtoken-TranX that was adopted by Alibaba's BizCook platform, which is the first domain code generation system adopted in industrial development environments.

Table 1: Comparisons of Seq2Seq and Seq2Tree code generation.

	Seq2Seq	Seq2Tree
Output length	Same as token sequence of code	Much longer than token sequence of code (about 1.5-2 times)
Operation / NLP technologies transfer	Easy	Medium
Data collection	Easy	Hard (Especially for large-scale data)
Grammatical correctness	No	Yes

A comparative view of the Seq2Seq and Seq2Tree code generation is shown in Table 1. Although Seq2Tree approaches can ensure GC, they have three major disadvantages: 1) AST node sequence is much longer (about 1.5-2 times) than token sequence [9], leading to increased difficulty in generation. 2) The Seq2Tree approach has to generate code with a complete tree structure, which makes the operation of obtaining arbitrary code snippets more difficult. 3) Assembling data for the Seq2Tree approach is laborious because it requires the generation of AST based on syntactically complete code.

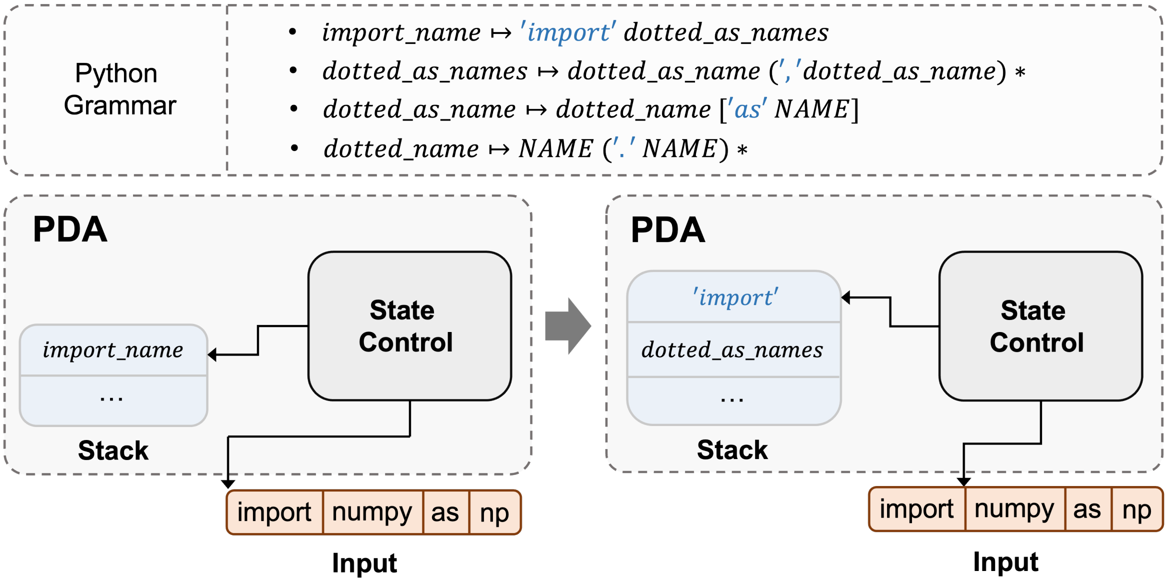

Python Grammar
• import_name ↦ 'import' dotted_as_names
• dotted_as_names ↦ dotted_as_name (','dotted_as_name) ∗
• dotted_as_name ↦ dotted_name ['as' NAME]
• dotted_name ↦ NAME ('.' NAME) ∗
PDA
State Control
import_name
…
Stack
import numpy as np
Input
PDA
'import'
dotted_as_names
…
Stack
State Control
import numpy as np
Input

Figure 4: Schematic diagram of a Python grammar PDA parsing Python code.

Compared to Seq2Tree approaches, our proposed approach not only ensures GC but also avoids the disadvantages of Seq2Tree approaches.

3.2 Domain-specific Code Generation

3.2.1 Grammatical Seq2Seq-based Code Generation. Scholak et al. [30] proposed PICARD that constrains auto-regressive decoders of language models through incremental parsing for a domain-specific language (DSL), i.e., Structured Query Language (SQL). PICARD used a parser for filtering in the beam search phase, meaning its candidate hypotheses might have terminal symbol mismatch errors. The work [25] proposed constrained semantic decoding (CSD) to address this problem for DSLs. However, both of them cannot extend to GPLs like Python. For the work [30], it focuses on solving the grammatical correctness of SQL generation tasks and uses a lot of prior knowledge of SQL, making it difficult to extend to other PLs. For the work [25], it uses a intuitive method to deal with the grammar of DSLs, but GPLs usually have more complicated grammar than DSLs, so as mentioned at the end of the paper [25], CSD cannot handle GPLs like Python.

For all we know, the grammatical Seq2Seq model for code generation within the context of GPLs has not yet been studied. There is a great need to ensure GC of Seq2Seq approaches for all PLs (including DSLs and GPLs), thus we introduce PDA to ensure GC for Seq2Seq code generation, which can be applied to all PLs.

4 METHODOLOGY

We first describe the core module of the proposed approach – a PDA-based methodology consisting of a PDA module and the corresponding algorithm to obtain a valid set for ensuring GC during code generation.

PLs (such as Python, Java, C++, etc.) belong to context-free languages, which can be accepted by a PDA [5, 10, 31]. A PDA M can be defined as:

$$M = (S, \Sigma, \Gamma, s_0, g_0, A, \delta) \tag{1}$$

where

- S is a finite set of states.
- Σ is a finite set of input symbols.
- Γ is a finite set of stack symbols.
- $s_0 \in S$ is the start state.
- $g_0 \in \Gamma$ is the starting stack symbol.
- $A \subseteq S$, where A is the set of accept states.
- δ is a transition function, mapping $S \times \Gamma \times (\Sigma \cup \{\varepsilon\})$ into the finite subsets of $S \times \Gamma^*$, where $*$ is the Kleene star.

To construct a PDA module for a PL grammar, we first define S, Σ, Γ, s_0, g_0, and A. In this paper, we adopt non-terminal symbols $\mathcal{N}'s$ of production rules in PL grammar as S, which can be changed with the PDA you build. Σ is set to terminal symbols $\mathcal{T}'s$ of production rules and Γ is the union of $\mathcal{T}'s$ and $\mathcal{N}'s$. In PL grammar, s_0 and g_0 are the starting non-terminal symbol, and A is the set of ending terminal symbols. As an example, in Python grammar PDA, s_0 and g_0 can be chosen as 'file_input', and A is the set of ENDMARKER token-type. Then, the definition of δ is:

- For each $\mathcal{N}$, δ(s, $\mathcal{N}$,ϵ) = {(s, β) | $\mathcal{N} \longrightarrow \beta$ is a production rule in PL grammar}.
- For each $\mathcal{T}$, δ(s, $\mathcal{T}$, $\mathcal{T}$) = {(s, ϵ)}.

The input symbols $\mathcal{I}'s$ in Σ are finite, because grammar of PLs merges the same type of tokens. For instance, each $\mathcal{I}$ in Python belongs to one of 83 syntax-strings and 10 token-types. As shown in Fig. 4, 'import' and 'as' are syntax-strings, while 'numpy' and 'np' belong to NAME token-type. Fig. 4 illustrates a Python grammar PDA parsing 'import numpy as np'. For a valid $\mathcal{I}$ = 'import', PDA jumps to valid states and stacks based on δ.

As shown in Algorithm 1, given current state s and current stack g, PDA M is able to provide the set of valid $\mathcal{I}'s$ according to δ. If ε belongs to the set of valid $\mathcal{I}'s$, the state and stack transferred after entering ε should be taken into account as well. Eventually, we merge all sets of $\mathcal{I}'s$ under each state and stack and remove ε from them.

Due to computer memory constraints, most of PLs are deterministic context-free languages, which can be accepted by a deterministic pushdown automaton (DPDA) [29]. A PDA M is deterministic only if both the following two conditions are satisfied:

- $\forall s \in S, g \in \Gamma, \mathcal{I} \in \Sigma \cup \{\varepsilon\}$, $|\delta(s, g, \mathcal{I})| = 1$ where $|\delta(s, g, \mathcal{I})|$ indicates the element number of the set $\delta(s, g, \mathcal{I})$.
- $\forall s \in S, g \in \Gamma, \mathcal{I} \in \Sigma$, if $\delta(s, g, \varepsilon) \neq \emptyset$, then $\delta(s, g, \mathcal{I}) = \emptyset$.

For the same deterministic context-free grammar, a general PDA is more accessible to construct than a DPDA. In addition, deterministic context-free languages are a proper subset of context-free languages. Therefore, in this paper, our proposed approach is based on the general PDA, which is also applicable to the DPDA.

Algorithm 1 Pseudocode for PDA module to obtain the valid set.

```
Require: PDA $M$ and current state $s$ as well as stack $g$.
Ensure: The set of valid $\mathcal{I}'s$ and corresponding states and stacks.
Initial empty valid set $V$ and empty queue $Q$.
Enqueue($Q$, $(s, g)$).
repeat
    $s, g \leftarrow$ Dequeue($Q$).
    for $(s, g, \mathcal{I}) \in \delta.keys()$ do
        if $\mathcal{I} = \varepsilon$ then
            Enqueue($Q$, $\delta(s, g, \varepsilon)$).
        else
            $V \leftarrow V \cup (\mathcal{I}, \delta(s, g, \mathcal{I}))$.
        end if
    end for
until $Q$ is empty
return $V$
```

5 CODEP

In this section, we introduce CODEP, which is constrained by the PDA module constructed under methodology. This framework can be compatible with existing Seq2Seq models, no matter it is an encoder-decoder or a decoder only. In this paper, we employ Transformer-based encoder-decoder model [35] as the backbone. We mainly modify the decoder side, which adds PDA module, state representation, state prediction task, and joint prediction to help models generate grammatical code.

Fig. 5 shows the model architecture of CODEP. At the encoder side of models, it maps an input sequence of NL utterances $\boldsymbol{x} = \{x_1, x_2, \cdots, x_n\}$ to a sequence of continuous NL representations $\boldsymbol{z} = \{z_1, z_2, \cdots, z_n\}$. At the decoder side of models, given the NL representation sequence $\boldsymbol{z}$, it then generates a token sequence of code, one element at a time. At each step, models generate the next token in an auto-regressive manner [13], which deploys previous generation results as additional input.

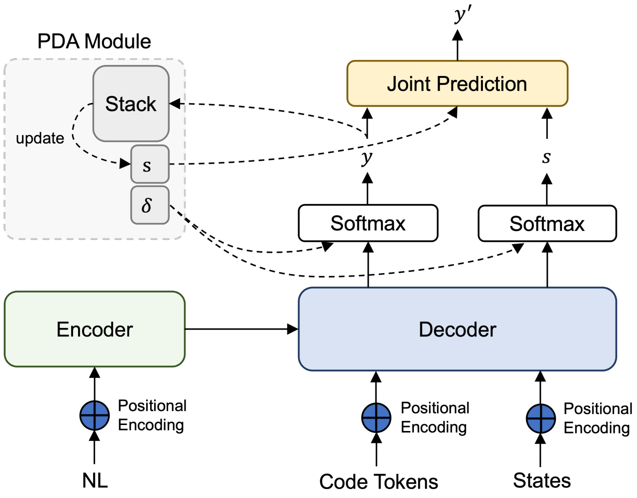


Figure 5: Diagram of CODEP.

5.1 Token and State Representation

PDA state is a type of valuable information to comprehend the status of the generated valid prefix in PDA module. In order to exploit states provided by PDA module, we set both tokens and corresponding states as the input of decoder. Specifically, given tokens $\boldsymbol{y} = \{y_1, y_2, \cdots, y_{t-1}\}$, we can obtain corresponding states $\boldsymbol{s} = \{s_1, s_2, \cdots, s_{t-1}\}$ from PDA module. Then, $\boldsymbol{y}$ and $\boldsymbol{s}$ are converted into $\boldsymbol{h}$ as follows:

$$\boldsymbol{h}_y = \boldsymbol{e}_y + \mathrm{PE}(\boldsymbol{y}) \tag{2}$$

$$\boldsymbol{h}_s = \boldsymbol{e}_s + \mathrm{PE}(\boldsymbol{s}) \tag{3}$$

$$\boldsymbol{h}_t = \mathrm{Concat}(\boldsymbol{h}_y, \boldsymbol{h}_s) \tag{4}$$

where $\boldsymbol{e}_y$ and $\boldsymbol{e}_s$ indicate the embedding of $\boldsymbol{y}$ and $\boldsymbol{s}$, respectively. $\mathrm{Concat}(\boldsymbol{h}_y, \boldsymbol{h}_s)$ indicates the concatenation of $\boldsymbol{h}_y$ and $\boldsymbol{h}_s$, and PE indicates the positional encoding [12]:

$$\mathrm{PE}_{(pos,2j)} = \sin\left(pos/10000^{2j/d}\right)$$

$$\mathrm{PE}_{(pos,2j+1)} = \cos\left(pos/10000^{2j/d}\right)$$

where pos is the position, j is the dimension, and d is the number of dimensions (i.e., embedding size).

5.2 Token Prediction Task

To generate code, $\boldsymbol{h}_t$ and NL description representation $\boldsymbol{z}$ are fed into the decoder:

$$\boldsymbol{a}_t = \mathrm{Decoder}(\boldsymbol{h}_t, \boldsymbol{z}) \tag{5}$$

The output of the multi-headed self-attention in the decoder layer is computed via:

$$\boldsymbol{Q_i} = \boldsymbol{HW_i^Q}, \boldsymbol{K} = \boldsymbol{HW_i^K}, \boldsymbol{V} = \boldsymbol{HW_i^V}, \tag{6}$$

$$\boldsymbol{head_i} = \mathrm{softmax}\left(\frac{\boldsymbol{Q_i K_i^\top}}{\sqrt{d_k}}\right)\boldsymbol{V_i}, \tag{7}$$

$$\boldsymbol{head} = \mathrm{Concat}(head_1, head_2, ..., head_h)\boldsymbol{W^O} \tag{8}$$

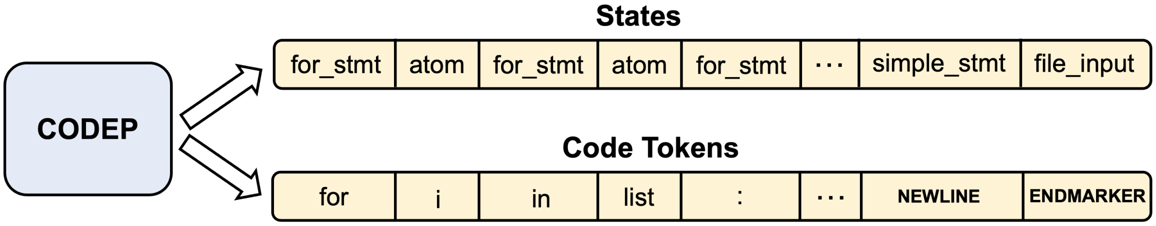


Figure 6: An example of state prediction task.

Where $\boldsymbol{W}_{\boldsymbol{i}}^{\boldsymbol{Q}} \in \mathbb{R}^{d_x \times d_k}, \boldsymbol{W}^{\boldsymbol{K}} \in \mathbb{R}^{d_x \times d_k}, \boldsymbol{W}^{\boldsymbol{V}} \in \mathbb{R}^{d_x \times d_h}$ are learnable parameter matrices. After that, a feed-forward layer and layer normalization are followed.

According to $\boldsymbol{a}_t$, we can compute the following probabilities:

$$p(\text{gen} \mid \boldsymbol{a}_t) = \text{softmax}(\mathbf{W}_x \boldsymbol{a}_t), \tag{9}$$

$$p(\text{copy} \mid \boldsymbol{a}_t) = 1 - p(\text{gen} \mid \boldsymbol{a}_t), \tag{10}$$

$$p(v_c \mid \text{gen}, \boldsymbol{a}_t) = \text{softmax}(\boldsymbol{e}_{v_c}^T \mathbf{W}_v \boldsymbol{a}_t), \tag{11}$$

$$p(x_i \mid \text{copy}, \boldsymbol{a}_t, \boldsymbol{x}) = \text{softmax}(\boldsymbol{h}_{x_i}^T \mathbf{W}_x \boldsymbol{a}_t). \tag{12}$$

where $\mathbf{W}_g$, $\mathbf{W}_v$, and $\mathbf{W}_x$ are three different parameter matrices, v_c indicates one of tokens in the vocabulary, and $\boldsymbol{h}_{x_i}$ is calculated by the pointer network [36]. Finally, at t step, the probability of generated token y_t can be defined as:

$$\begin{aligned} p(y_t \mid y_{<t}, s_{<t}, \boldsymbol{x}) = {} & p(\text{gen} \mid \boldsymbol{a}_t) p(v_c \mid \text{gen}, \boldsymbol{a}_t) \\ & + p(\text{copy} \mid \boldsymbol{a}_t) p(x_i \mid \text{copy}, \boldsymbol{a}_t, \boldsymbol{x}). \end{aligned} \tag{13}$$

5.3 State Prediction Task

To better understand the deduction process of PDA, we introduce an auxiliary task of PDA state prediction. Similar to token prediction, given $y_{<t}$, $s_{<t}$, and $\boldsymbol{x}$, we can calculate the probability of a state at t step as follows:

$$p(s_t \mid y_{<t}, s_{<t}, \boldsymbol{x}) = \text{softmax}(\boldsymbol{e}_{v_s}^T \mathbf{W}_s \boldsymbol{a}_t), \tag{14}$$

where v_s indicates one of states in vocabulary, $\mathbf{W}_s$ is another parameter matrix, and $\boldsymbol{a}_t$ is calculated via (5). In this paper, we adopt the non-terminal symbols of production rules in GPL grammar as states, which can be changed with PDA you build. Fig. 6 shows an example of state prediction task. We can see that state prediction task also gives guidance for generating tokens. For example, 'for', 'in', and ':' are the fixed pairing of 'for_stmt' in Python.

5.4 Training and Inference

The purpose of the training stage is to minimize the sum of cross-entropy loss for two tasks, which is defined as:

$$\begin{aligned} \mathcal{L}^{\text{ce}}(\boldsymbol{x}, \boldsymbol{y}, s; \boldsymbol{\theta}) = {} & \mathcal{L}_y^{\text{ce}}(\boldsymbol{x}, \boldsymbol{y}, s; \boldsymbol{\theta}) \\ & + \alpha \cdot \mathcal{L}_s^{\text{ce}}(\boldsymbol{x}, \boldsymbol{y}, s; \boldsymbol{\theta}), \end{aligned} \tag{15}$$

$$\mathcal{L}_y^{\text{ce}}(\boldsymbol{x}, \boldsymbol{y}, s; \boldsymbol{\theta}) = -\sum_{t=1}^{T} \log p\left(y_t \mid y_{<t}, s_{<t}, \boldsymbol{x}; \boldsymbol{\theta}\right), \tag{16}$$

$$\mathcal{L}_s^{\text{ce}}(\boldsymbol{x}, \boldsymbol{y}, s; \boldsymbol{\theta}) = -\sum_{t=1}^{T} \log p\left(s_t \mid y_{<t}, s_{<t}, \boldsymbol{x}; \boldsymbol{\theta}\right), \tag{17}$$

where $\boldsymbol{\theta}$ is CODEP's parameter, $\mathcal{L}_y^{\text{ce}}$ and $\mathcal{L}_s^{\text{ce}}$ are losses of token prediction task and state prediction task, respectively, and α is used to control the impact of them.

In the inference stage, we are able to calculate $p(y_t \mid y_{<t}, s_{<t}, \boldsymbol{x})$ via (13) with additional constraints of PDA module $y_t \in \{\mathcal{I} \mid (\mathcal{I}, s, g) \in V\}$, where V is obtained according to Algorithm 1. Similarly, we can calculate $p(s_t \mid y_{<t}, s_{<t}, \boldsymbol{x})$ via (14).

5.4.1 *Joint Prediction with State.*

We can leverage both probabilities to jointly predict the result as follows:

$$\begin{aligned} p(y'_t \mid y_{<t}, s_{<t}, \boldsymbol{x}) = {} & \frac{1}{1+\alpha} \cdot p(y_t \mid y_{<t}, s_{<t}, \boldsymbol{x}) \\ & + \frac{\alpha}{1+\alpha} \cdot p(\hat{s_t} \mid y_{<t}, s_{<t}, \boldsymbol{x}), \end{aligned} \tag{18}$$

where $\hat{s_t}$ indicates the actual state of PDA module, which is obtained from $\delta(s_{t-1}, g_{t-1}, y_t)$. Joint prediction wants to prevent CODEP from generating tokens of inappropriate PDA states.

We provide the pseudocode for the inference procedure of CODEP in Algorithm 2.

Algorithm 2 The inference procedure of CODEP.

```
Require: The PDA $M$, hyperparameter $\alpha$, parameters of CODEP
    $\boldsymbol{\theta}$, and NL utterances $\boldsymbol{x}$.
Ensure: The generated code tokens $\boldsymbol{y'}$.
Initial $t \leftarrow 0$ and obtain $s_0$ and $g_0$.
repeat
    Obtain $V$ according to Algorithm 1.
    Calculate $p(y_t \mid y_{<t}, s_{<t}, \boldsymbol{x})$ via (13), where
    $y_t \in \{\mathcal{I} | (\mathcal{I}, s, g) \in V\}$.
    Calculate $p(s_t \mid y_{<t}, s_{<t}, \boldsymbol{x})$ via (14).
    Calculate $p(y'_t \mid y_{<t}, s_{<t}, \boldsymbol{x})$ via (18).
    $y'_t \leftarrow \arg\max p(y'_t \mid y_{<t}, s_{<t}, \boldsymbol{x})$.
    $y_t \leftarrow y'_t$.
    $s_{t+1}, g_{t+1} = \delta(s_t, g_t, y'_t)$
    $t \leftarrow t + 1$
until $s_t \in A$
return $\boldsymbol{y'}$
```

6 EXPERIMENT SETUP

In this section, we delve into a comprehensive explanation of the experiment setup, covering aspects such as GPL and datasets, baselines, evaluation metrics, implementation details, and research questions.

6.1 GPL and Datasets

We build a PDA for the most popular PL Python (both Python2 and Python3) and conduct experiments on four public benchmark datasets, i.e., CONALA [42], DJANGO [24], JUICE-10K [1], and MBPP [3], as follows:

CONALA [42] contains 2,879 real-world data of manually annotated NL questions and their Python3 solutions on STACK OVERFLOW.

DJANGO [24] contains 18805 NL-annotated python2 code extracted from the Django Web data.

JUICE-10K contains 10K training samples randomly selected from the training set of JUICE [1], and the validation and test sets of JUICE-10K are consistent with those of JUICE. Due to the high demand for training resources, we use a subset instead of the full JuICe dataset.

MBPP [3] contains 974 Python programming problems, which consist of a task description, code solution, and 3 automated test cases.

The detailed statistics of the four datasets are shown in Table 2.

Table 2: Statistics of datasets.

Dataset	Examples Num			Avg Length	
	Train	Dev	Test	NL	Code
CONALA	2,175	200	500	10.2	15.1
DJANGO	16000	1000	1805	14.1	10.6
JUICE	10000	1,831	2,115	40.4	43.4
MBPP	-	-	974	15.5	32.5

6.2 Baselines

We select two typical **Seq2Seq models**, i.e., LSTM [15] and Transformer [35], as our base models.

For **non-pre-trained Seq2Tree methods**, we compare CODEP with TRANX [45], ML-TRANX [40], TRANX-RL [18], APT [9], and Transformer-AST, which can ensure GC of generated codes relying on AST.

TRANX [45] uses a Seq2Tree model to generate the AST as the intermediate representation of code.

ML-TRANX [40] adopts a mutual learning framework to train models for different traversals-based decodings jointly.

TRANX-RL [18] uses a context-based branch selector to determine optimal branch expansion orders for multi-branch nodes dynamically.

APT [9] uses antecedent prioritized loss to help Seq2Tree models attach importance to antecedent predictions by exploiting the position information of the generated AST nodes.

Transformer-AST indicates a Seq2Tree method based on Transformer instead of LSTM adopted in other Seq2Tree methods mentioned preceding.

For **pre-trained Seq2Seq methods**, we choose encoder-decoder models, including BART [21] and CodeT5 [38], and decoder-only models, including CodeGen [23] and InCoder [11].

BART [21] is a pre-training approach for text generation tasks that learns to map corrupted documents to the original.

CodeT5 [38] is a unified pre-trained encoder-decoder Transformer model that makes better use of the code semantics conveyed from developer-assigned identifiers.

CodeGen [23] is a unified generation model that allows left-to-right code generation and code infilling/editing by the causal mask language modeling training objective. In this paper, we employ the CodeGen-Multi version.

InCoder [11] is a collection of large-scale language models trained on NL and programming data for conversation-based program synthesis.

6.3 Evaluation Metrics

To evaluate the effectiveness of different methods, we use three widely-used evaluation metrics in code generation: exact matching accuracy (EM), corpus-level BLEU-4 (BLEU), and CodeBLEU [28], which considers important grammar and semantic features of codes. We also use GC and GC percentage (GCP), where GCP indicates the percentage of grammatically generated codes in all generated codes.

6.4 Implementation Details

We train our model with Adam [20] optimizer on a single GPU of Tesla A100-PCIe-40G. We set sizes of the word embedding, code embedding, state embedding, and hidden state as 256, 256, 256, and 512, respectively. The learning rate is set to 0.0001 for Transformer-based CODEP and 0.001 for LSTM-based CODEP. For additional hyperparameter α, we pick $\alpha \in [0, 1]$ on validation sets. The beam size is set to 2 for MBPP and 15 for the other datasets. For the approaches mentioned in Section 6.2, we follow the settings in their paper. To mitigate the instability of the model training, we exhibit the average performance of the model running five times.

6.5 Research Questions

Our study is centered around five main research questions (RQs):

RQ1: How does the proposed CODEP perform compared to the state-of-the-art Seq2Tree methods, which guarantees the syntactical correctness of the generated code relying on AST, without pre-training?

RQ2: How does each part of our proposed approach contribute to CODEP?

RQ3: How does the proposed PDA module improve the effectiveness of the state-of-the-art pre-trained Seq2Seq methods?

RQ4: What is the performance of code generated with the help of the PDA module?

RQ5: How well does the code generated by CODEP compile, and are there any solutions to compile errors?

7 EXPERIMENTAL RESULTS

In this section, we present the results of our empirical experiments, conducted specifically to answer the aforementioned RQs.

7.1 RQ1: CODEP vs. Seq2Tree Methods

From Tables 3, we can observe that CODEP substantially outperforms SOTA Seq2Tree methods without pre-training on three public benchmark datasets in terms of BLEU, CodeBLEU, and EM. It indicates that although both Seq2Tree methods and CODEP generate

Table 3: Comparison of CODEP and Seq2Tree methods.

Model	CONALA			DJANGO	JUICE-10K	
	BLEU	CodeBLEU	EM	EM	BLEU	CodeBLEU
Seq2Tree methods w/o pre-training						
TRANX [45]	24.35 ± 0.4	26.80 ± 0.6	2.5 ± 0.7	77.3 ± 0.4	4.63 ± 0.2	13.09 ± 0.4
ML-TRANX [40]	24.42 ± 0.8	27.59 ± 1.0	2.2 ± 0.4	79.6 ± 0.3	4.75 ± 0.4	13.43 ± 0.5
TRANX-RL [18]	25.47 ± 0.7	25.14 ± 1.0	2.6 ± 0.4	77.9 ± 0.5	6.08 ± 0.3	13.78 ± 0.6
APT [9]	27.56 ± 0.7	28.54 ± 0.7	2.9 ± 0.7	79.3 ± 0.5	5.78 ± 0.5	14.19 ± 0.5
Transformer-AST	26.50 ± 0.4	27.31 ± 0.4	1.8 ± 0.2	77.1 ± 0.5	4.80 ± 0.3	13.57 ± 0.4
LSTM	24.23 ± 0.7	25.76 ± 0.8	2.0 ± 0.7	70.5 ± 0.4	4.54 ± 0.5	13.13 ± 0.7
CODEP (LSTM-based)	**28.22 ± 0.8**	**30.12 ± 0.9 (↑ 17%)**	**3.0 ± 0.6**	79.6 ± 0.6 (↑ 13%)	6.32 ± 0.6	15.24 ± 0.6 (↑ 16%)
Transformer	24.21 ± 0.8	25.40 ± 0.9	2.0 ± 0.6	74.1 ± 0.7	4.67 ± 0.4	14.08 ± 0.5
CODEP (Transformer-based)	27.87 ± 0.9	29.49 ± 0.9 (↑ 16%)	2.6 ± 0.8	**79.7 ± 0.5 (↑ 8%)**	**7.26 ± 0.5**	**16.09 ± 0.6 (↑ 15%)**

Table 4: Ablation study of CODEP, where SR represents state representation, SPT represents state prediction task, and JP represents joint prediction.

Model	CONALA			DJANGO		JUICE-10K		
	BLEU	CodeBLEU	GCP	EM	GCP	BLEU	CodeBLEU	GCP
CODEP (LSTM-based)	28.22	30.12	100%	79.6	100%	6.32	15.24	100%
-JP	27.81	29.54	100%	78.7	100%	5.96	15.04	100%
-SPT	27.36	28.95	100%	77.3	100%	5.38	14.26	100%
-SR	27.97	29.98	100%	77.6	100%	5.54	14.47	100%
-PDA-JP	26.37	27.49	80.8%	74.1	91.1%	5.09	13.87	54.7%
-PDA-SR-SPT-JP	24.23	25.76	69.4%	70.5	89.5%	4.54	13.13	43.9%
CODEP (Transformer-based)	27.87	29.49	100%	79.7	100%	7.26	16.09	100%
-JP	27.05	28.85	100%	78.9	100%	7.12	15.82	100%
-SPT	26.72	27.79	100%	77.8	100%	6.59	15.52	100%
-SR	27.13	28.83	100%	78.3	100%	6.97	15.68	100%
-PDA-JP	26.32	27.46	81.4%	76.7	92.3%	5.45	15.20	57.6%
-PDA-SR-SPT-JP	24.21	25.40	70.6%	74.1	90.1%	4.67	14.79	45.3%

grammatical codes, CODEP takes advantage of the shorter generation sequence length of code tokens than AST nodes. We also find that Transformer-based CODEP achieves competitive performance against LSTM-based CODEP on CONALA. However, on DJANGO and JUICE-10K, Transformer-based CODEP exhibits its superiority due to more extensive training data and longer generated token sequence. Therefore, for large dataset and long code generation, we recommend employing Transformer as the base model of CODEP. Since the decoder of most Seq2Tree methods without pre-training is based on LSTM, we implement Transformer-AST to verify that the effect of CODEP is derived from our proposed approach rather than the base model. The experimental results demonstrate that our enhancements are not primarily attributable to the changes of the base model.

7.2 RQ2: Ablation Study

In Table 4, we demonstrate the performance of LSTM-based and Transformer-based CODEP with the reduction of some parts of them. The results indicate that each part of CODEP contributes, and PDA module is the most critical part of CODEP. When PDA module is removed, GCP drops sharply, which is inversely correlated with the average code length of datasets, and other evaluation metrics also have degradation. JP becomes unavailable due to the dependence of JP on the mapping from the token to the corresponding state provided by PDA module. In addition, it can be seen that only reducing JP or SR leads to a relatively small decrease in performance, because they have some overlap with other parts in helping the model. '-PDA-SR-SPT-JP' represents that only the base model is used, which has a significant performance degradation compared to CODEP (both LSTM-based and Transformer-based). As shown in Tables 3, it relatively improves 17%/16% CodeBLEU on CONALA, 13%/8% EM on DJANGO, and 16%/15% CodeBLEU on JUICE-10K compared to the base models.

7.3 RQ3: Improvement of Seq2Seq Methods with PDA Module

It is apparent from Table 6 that pre-trained models fail to work on MBPP in zero-shot setting. For encoder-decoder models, BART is only pre-trained on the data of text generation tasks, while CodeT5 is pre-trained on paired NL-code data of six PLs, but cannot distinguish what PL needs to be generated. For decoder-only models, we explore their performance under conditions that no prompt is given or only "def" is given as prompt, because traditional prompt 'def

Table 5: Examples of code generation for each pre-trained model with zero-shot setting, where NL is 'Write a function to find the similar elements from the given two tuple lists' and GOLD denotes the reference code.

Model	Code
BART-125M	Write a function to find the similar elements from the given two tuple lists.
CodeT5-220M	def f_elements( )
CodeT5-770M	to find the similar elements ." ," ." ," a function to find the similar elements from the given two tuple lists .
CodeGen-350M	def find_similar (t1 , t2): \n """ \n Finds the similar elements from the given two tuples. \n \n
InCoder-1B	def similar_elements (t1, t2): \n \n \n \n \n \n \n \n \n \n \n \n
CodeGen-350M +PDA	def find (a , b): \n for i in range (len (a)): \n if a [i] == b [i]: \n return a[i]
GOLD	def similar_elements (test_tup1, test_tup2) : \n res = tuple (set (test_tup1) & set (test_tup2)) \n return (res)

Table 6: The results of each pre-trained Seq2Seq method with PDA module in zero-shot setting.

Model	MBPP (Zero-shot)		
	BLEU	CodeBLEU	GC
Encoder-decoder methods w/ pre-training			
BART-125M	0.02	3.74	False
+PDA	4.83	9.17	True
CodeT5-220M	0.01	0.57	False
+PDA	4.75	9.79	True
CodeT5-770M	0.02	3.98	False
+PDA	6.19	9.31	True
Decoder-only methods w/ pre-training			
CodeGen-350M	1.55	3.21	False
+prompt	0.33	10.13	False
+PDA	**14.44**	**21.54**	True
InCoder-1B	0.12	7.41	False
+prompt	0.82	7.22	False
+PDA	9.84	17.38	True

+ function signature' contains much valid information of function signature, including function name, argument type and name, and even return value type. Therefore, their failures are to be expected since they are working in unfamiliar territory.

As can be seen from Table 6, pre-trained sequence-based models achieve significant improvements with the help of PDA module. The reason may be that pre-training models have the ability to answer questions, but do not know how to organize them into well-formed codes. Under the constraints of PDA module, pre-training models are guided to exert their capabilities by generating grammatical codes. Especially, (BLEU, CodeBLEU) of CodeGen-350M without prompt improvement from (1.55, 3.21) to (14.44, 21.54) on MBPP in zero-shot setting.

7.4 RQ4: Case Study

In Table 5, we demonstrate examples of code generation for each pre-trained model in zero-shot setting. We can see that the output of BART-125M directly replicates NL, which is related to its pre-training task. CodeT5-220M understands some syntactic structures, but the short length of their generated code is the main factor limiting their performance. CodeT5-770M tries to capture some semantic information at the beginning and continues to copy later. InCoder only generates the function signature and fails to generate its main body. CodeGen-350M generates the function signature and some comments capturing semantic information, which shows the ability to generate complete code if under proper guidelines. Under the constraints of PDA module, CodeGen-350M generates grammatical code. Although the code generated by CodeGen-350M + PDA still falls short of GOLD, we still think it is an impressive result considering that it is under the zero-shot setting.

7.5 RQ5: Compilation Error Analysis

Program execution needs to be compiled correctly in addition to being grammatically correct, and grammatical correctness is the foundation of compilation. The compilation rate of CODEP on CONALA dataset is 99.2%, and Table 7 demonstrates examples of three types of compilation error. The first type of compilation error is because the number type cannot be assigned, the second type of compilation error is the error of the duplicate parameter, and the third type of compilation error is the mapping error. We found that the preceding types of errors also occur in the Seq2Tree methods. For the first two types of errors, we are able to modify the transfer function, while the last type of error can be avoided using engineering methods.

Table 7: Three types of compilation error generated by CODEP on CONALA dataset.

Type	Example
I	1 = [(i , 2) for i in range(1)]
II	plt.plot(x, var1, color='str0', color='str1')

III	html = response.read(str0, 'str0')
	slot_map={str0: http://www.example.com/}

8 LIMITATION

There are three major limitations of our work:

- First, a specific PDA should be built for each PL, and we only built PDA for Python (including both Python2 and Python3). However, PDA can recognize context-free language to which PL belongs, due to time and memory limitations of running PL [29]. In future work, we plan to build PDAs for more PLs.

- Second, our work ensures GC, but neither our work nor other works can prevent compilation errors. With our proposed training framework, our work outperforms other works on compilation rates, e.g., CODEP has 99.2% compilation rates after training on CONALA.
- Third, using PDA will increase CPU operation and memory overhead, but it is still acceptable.

9 CONCLUSION AND FUTURE WORK

In this paper, we have proposed a Seq2Seq-based framework, namely CODEP, that first integrates the deduction of PDA into deep learning for code generation. CODEP adds PDA module, state representation, state prediction task, and joint prediction for the decoder of CODEP. PDA module guarantees the GC of generated codes, while the others make use of valid information provided by PDA module (i.e., PDA state) to generate codes better. As a result, CODEP dramatically enhances the performance of base models and outperforms SOTA Seq2Tree methods without pre-training on three benchmark datasets. The ablation study demonstrates each component of CODEP contributes. With the help of PDA module, pre-trained models also achieve significant improvements.

PDA module shows excellent potential in code generation tasks in zero-shot setting, which can help pre-trained models apply to niche PLs that have little data. Furthermore, PDA can accommodate context-free language to satisfy grammar constraints, not just PL. We hope this work sheds light on future work in this direction.

10 ACKNOWLEDGMENTS

This research is supported by the National Key R&D Program under Grant No. 2021ZD0110303, the National Natural Science Foundation of China under Grant Nos. 62192733, 62192730, 62192731, 61751210, 62072007, and 61832009.

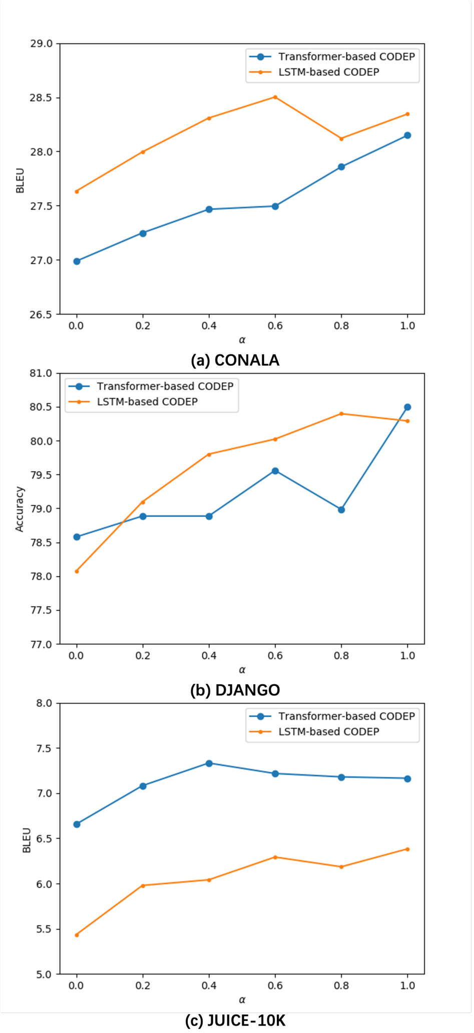
A APPENDIX

A.1 Syntax-strings and Token-types in Python

In Table 8, we show each of the 83 syntax-strings and 10 token-types in Python.

Table 8: Terminal symbols of Python grammar.

	Python
Syntax-string	‘for’, ‘in’, ‘try’, ‘finally’, ‘with’, ‘except’, ‘lambda’, ‘or’, ‘and’, ‘not’, ‘del’, ‘pass’, ‘break’, ‘continue’, ‘return’, ‘raise’, ‘from’, ‘import’, ‘as’, ‘nonlocal’, ‘global’, ‘assert’, ‘if’, ‘else’, ‘elif’, ‘while’, ‘async’, ‘def’, ‘@’, ‘(’, ‘)’, ‘->’, ‘:’, ‘**’, ‘*’, ‘,’, ‘=’, ‘;’, ‘^=’, ‘%=’, ‘//=’, ‘@=’, ‘«=’, ‘**=’, ‘&=’, ‘*=’, ‘|=’, ‘»=’, ‘-=’, ‘+=’, ‘/=’, ‘.’, ‘...’, ‘<=’, ‘>=’, ‘is’, ‘==’, ‘<’, ‘>’, ‘<>’, ‘!=’, ‘|’, ‘^’, ‘&’, ‘«’, ‘»’, ‘+’, ‘-’, ‘%’, ‘/’, ‘//’, ‘~’, ‘!’, ‘await’, ‘False’, ‘[’, ‘{’, ‘True’, ‘None’, ‘]’, ‘}’, ‘class’, ‘yield’
Token-type	‘NAME’, ‘STRING’, ‘NUMBER’, ‘INDENT’, ‘DEDENT’, ‘FSTRING_START’, ‘FSTRING_END’, ‘FSTRING_STRING’, ‘NEWLINE’, ‘ENDMARKER’



(a) CONALA

(b) DJANGO

(c) JUICE-10K

Figure 7: Effects of α on the validation sets.

A.2 Effect of α

The coefficient α is an important hyperparameter that controls the relative impacts of the state prediction task in the training stage and that of the joint prediction with the state probability in the inference stage. Therefore, we investigate the effect of α on our proposed framework in Fig. 7, which varies α from 0 to 1 with an increment of 0.2 on the validation set of CONALA, DJANGO, and JUICE-10K datasets. The experimental results show that as α increases, the performance of CODEP increases when $\alpha \leq 0.4$, and its tendency is related to the base model and the dataset when $\alpha > 0.4$. What stands out in Fig. 7 is that CODEP performs relatively well on each of the preceding datasets when α is set to 1.0, which indicates the effectiveness of the state prediction task and joint prediction. For experiments in this paper, we set α as the optimal α in Fig. 7. In particular, the $\alpha's$ of the LSTM-based CODEP and the transformer-based CODEP are set to (0.6, 1.0), (0.8, 1.0), and (1.0, 0.4) on CONALA, DJANGO, and JUICE-10K dataset, respectively.

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)*. 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] Jacob Austin, Augustus Odena, Maxwell I. Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. 2021. Program Synthesis with Large Language Models. *CoRR* abs/2108.07732 (2021).

[4] Ruisheng Cao, Su Zhu, Chen Liu, Jieyu Li, and Kai Yu. 2019. Semantic Parsing with Dual Learning. In *ACL (1)*. 51–64.

[5] Noam Chomsky. 1962. Context-free grammars and pushdown storage. *MIT Res. Lab. Electron. Quart. Prog. Report.* 65 (1962), 187–194.

[6] Li Dong and Mirella Lapata. 2016. Language to Logical Form with Neural Attention. In *ACL (1)*.

[7] Yihong Dong, Jiazheng Ding, Xue Jiang, Zhuo Li, Ge Li, and Zhi Jin. 2023. CodeScore: Evaluating Code Generation by Learning Code Execution. *CoRR* abs/2301.09043 (2023).

[8] Yihong Dong, Xue Jiang, Zhi Jin, and Ge Li. 2023. Self-collaboration Code Generation via ChatGPT. *CoRR* abs/2304.07590 (2023).

[9] Yihong Dong, Ge Li, and Zhi Jin. 2022. Antecedent Predictions Are Dominant for Tree-Based Code Generation. *CoRR* abs/2208.09998 (2022).

[10] R James Evey. 1963. Application of pushdown-store machines. In *Proceedings of the November 12-14, 1963, fall joint computer conference*. 215–227.

[11] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. 2022. InCoder: A Generative Model for Code Infilling and Synthesis. *CoRR* abs/2204.05999 (2022).

[12] Jonas Gehring, Michael Auli, David Grangier, Denis Yarats, and Yann N. Dauphin. 2017. Convolutional Sequence to Sequence Learning. In *ICML (Proceedings of Machine Learning Research, Vol. 70)*. PMLR, 1243–1252.

[13] Alex Graves. 2013. Generating Sequences With Recurrent Neural Networks. *CoRR* abs/1308.0850 (2013).

[14] 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.

[15] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long Short-Term Memory. *Neural Comput.* 9, 8 (1997), 1735–1780.

[16] Robin Jia and Percy Liang. 2016. Data Recombination for Neural Semantic Parsing. In *ACL (1)*.

[17] Hui Jiang, Linfeng Song, Yubin Ge, Fandong Meng, Junfeng Yao, and Jinsong Su. 2022. An AST Structure Enhanced Decoder for Code Generation. *IEEE ACM Trans. Audio Speech Lang. Process.* 30 (2022), 468–476.

[18] 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*. 5076–5085.

[19] Xue Jiang, Yihong Dong, Lecheng Wang, Qiwei Shang, and Ge Li. 2023. Self-planning Code Generation with Large Language Model. *CoRR* abs/2303.06689 (2023).

[20] Diederik P. Kingma and Jimmy Ba. 2015. Adam: A Method for Stochastic Optimization. In *ICLR*.

[21] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. 2020. BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension. In *ACL*. Association for Computational Linguistics, 7871–7880.

[22] 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)*.

[23] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. A Conversational Paradigm for Program Synthesis. *CoRR* abs/2203.13474 (2022).

[24] Yusuke Oda, Hiroyuki Fudaba, Graham Neubig, Hideaki Hata, Sakriani Sakti, Tomoki Toda, and Satoshi Nakamura. 2015. Learning to generate pseudo-code from source code using statistical machine translation. In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 574–584.

[25] Gabriel Poesia, Oleksandr Polozov, Vu Le, Ashish Tiwari, Gustavo Soares, Christopher Meek, and Sumit Gulwani. 2022. Synchromesh: Reliable code generation from pre-trained language models. *CoRR* abs/2201.11227 (2022).

[26] Maxim Rabinovich, Mitchell Stern, and Dan Klein. 2017. Abstract Syntax Networks for Code Generation and Semantic Parsing. In *ACL (1)*. 1139–1149.

[27] Veselin Raychev, Martin T. Vechev, and Eran Yahav. 2014. Code completion with statistical language models. In *PLDI*. 419–428.

[28] 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).

[29] Arto Salomaa, Derick Wood, and Sheng Yu. 2001. *A half-century of automata theory: celebration and inspiration*. World scientific.

[30] Torsten Scholak, Nathan Schucher, and Dzmitry Bahdanau. 2021. PICARD: Parsing Incrementally for Constrained Auto-Regressive Decoding from Language Models. In *EMNLP (1)*. Association for Computational Linguistics, 9895–9901.

[31] Marcel Paul Schützenberger. 1963. On context-free languages and push-down automata. *Information and control* 6, 3 (1963), 246–264.

[32] Sijie Shen, Xiang Zhu, Yihong Dong, Qizhi Guo, Yankun Zhen, and Ge Li. 2022. Incorporating domain knowledge through task augmentation for front-end JavaScript code generation. In *ESEC/SIGSOFT FSE*. ACM, 1533–1543.

[33] Zeyu Sun, Qihao Zhu, Lili Mou, Yingfei Xiong, Ge Li, and Lu Zhang. 2019. A Grammar-Based Structural CNN Decoder for Code Generation. In *AAAI*. 7055–7062.

[34] 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*. 8984–8991.

[35] 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.

[36] Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. 2015. Pointer Networks. In *NIPS*. 2692–2700.

[37] 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.

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

[39] 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.

[40] 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*. 14121–14128.

[41] Mengfei Yang, Bin Gu, Zhenhua Duan, Zhi Jin, Naijun Zhan, Yunwei Dong, Cong Tian, Ge Li, Xiaogang Dong, and Xiaofeng Li. 2022. Intelligent program synthesis framework and key scientific problems for embedded software. *Chinese Space Science and Technology* (2022), 1.

[42] Pengcheng Yin, Bowen Deng, Edgar Chen, Bogdan Vasilescu, and Graham Neubig. 2018. Learning to mine aligned code and natural language pairs from stack overflow. In *MSR*. 476–486.

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

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

[45] Pengcheng Yin and Graham Neubig. 2019. Reranking for Neural Semantic Parsing. In *ACL*. 4553–4559.

Received 2023-02-16; accepted 2023-05-03