# Deep Learning Based Program Generation From Requirements Text: Are We There Yet?

Hui Liu, Mingzhu Shen, Jiaqi Zhu, Nan Niu, Ge Li, and Lu Zhang

**Abstract—**To release developers from time-consuming software development, many approaches have been proposed to generate source code automatically according to software requirements. With significant advances in deep learning and natural language processing, deep learning-based approaches are proposed to generate source code from natural language descriptions. The key insight is that given a large corpus of software requirements and their corresponding implementations, advanced deep learning techniques may learn how to translate software requirements into source code that fulfill such requirements. Although such approaches are reported to be highly accurate, they are evaluated on datasets that are rather small, lack of diversity, and significantly different from real-world software requirements. To this end, we build a large scale dataset that is composed of longer requirements as well as validated implementations. We evaluate the state-of-the-art approaches on this new dataset, and the results suggest that their performance on our dataset is significantly lower than that on existing datasets concerning the common metrics, i.e., BLEU. Evaluation results also suggest that the generated programs often contain syntactic and semantical errors, and none of them can pass even a single predefined test case. Further analysis reveals that the state-of-the-art approaches learn little from software requirements, and most of the successfully generated statements are popular statements in the training programs. Based on this finding, we propose a popularity-based approach that always generates the most popular statements in training programs regardless of the input (software requirements). Evaluation results suggest that none of the state-of-the-art approaches can outperform this simple statistics-based approach. As a conclusion, deep learning-based program generation requires significant improvement in the future, and our dataset may serve as a basis for future research in this direction.

**Index Terms—**Software requirements, code generation, deep learning, data set

## 1 Introduction

Software development is the process of writing and maintaining source code according to software requirements [1]. The resulting source code is in turn compiled automatically into executable applications that finally fulfill the requirements. However, with the increase in software complexity, software development is often expensive and error-prone [1] although many engineering approaches have been proposed to guide the development.

To release human beings from challenging, time-consuming, and error-prone software development (especially coding), many approaches have been proposed to generate source code automatically. During the last twenty years of the twentieth century, researchers proposed mathematics-based formal methods [2], [3] and tools [4], [5] to generate source code automatically according to formal specifications [6], [7]. Although formal methods are highly reliable, it remains challenging to create formal specifications that should be described in formal languages, e.g., Z [8]. Consequently, researchers turn to less formal approaches, e.g., Model Driven Architecture (MDA) [9]. MDA attempts to generate source code [10] according to models described in modeling languages, e.g., Unified Modelling Language (UML) [11]. UML is a graphical modelling language, and shares most of the concepts with object-oriented programming languages. Consequently, developers are willing to use UML compared to formal languages. However, because UML models usually focus on the architecture of the system under development, it is quite often that MDA generates nothing but sketch (e.g., signatures of methods) of the system. Detailed implementation, especially the body of methods, still has to be typed in manually in most cases. Extending UML with action semantics [12] makes it possible to present more detailed semantics in UML models, and thus we may generate more complete source code from UML models. However, they often employ DSMLs instead of general-purpose languages. It is also challenging and time-consuming to construct action semantics with the extended UML.

With significant advances in deep learning, researchers are turning to learning-based approaches to generate source code. The key insight of such approaches is that given a corpus of software requirements and their corresponding implementation (source code), advanced deep learning techniques may learn how to translate software requirements into source code that fulfill such requirements [13], [14]. Existing approaches have successfully generated source code from software requirements in natural language descriptions [15], [16],

- *Hui Liu, Mingzhu Shen, and Jiaqi Zhu are with the School of Computer Science and Technology, Beijing Institute of Technology, Beijing 100081, China. E-mail: {liuhui08, 3120181025, zhujiaqi}@bit.edu.cn.*
- *Nan Niu is with the Department of Electrical Engineering and Computer Science, University of Cincinnati, Cincinnati, OH 45221 USA. E-mail: nan.niu@uc.edu.*
- *Ge Li and Lu Zhang are with the Key Laboratory of High Confidence Software Technologies, Ministry of Education, Peking University, Beijing 100871, China. E-mail: lige@pku.edu.cn, zhanglu@sei.pku.edu.cn.*

*Manuscript received 27 Apr. 2020; revised 31 July 2020; accepted 18 Aug. 2020. Date of publication 21 Aug. 2020; date of current version 18 Apr. 2022.*
*(Corresponding author: Hui Liu.)*
*Recommended for acceptance by H. Rajan.*
*Digital Object Identifier no. 10.1109/TSE.2020.3018481*

images of Graphical User Interface (GUI) [17], [18], and input-output examples [19], [20]. In this paper, we focus on natural language requirements (requirements text) because in most cases software requirements in the industry are described in natural languages [21]. Deep learning-based code generation approaches have been reported to be highly accurate. For example, the syntactic neural model (*SNM*) proposed by Yin and Neubig [22] reaches a high Bilingual Evaluation Understudy (BLEU) [23] (0.845) in translating natural language descriptions into Python programs. In contrast, Google neural machine translation (*GNMT*), the widely used state-of-the-practice language translator, results in much smaller BLEU (0.4) in translating English into French [24]. For example, if the reference translation is "*I know tomorrow is another day*" and the generated translation is "*I know tomorrow is a new day*", the resulting BLEU is 0.42.

However, such deep learning-based code generation approaches have not yet been extensively evaluated, which prevents us from knowing the state of the art. The reported evaluation of such approaches is often conducted on datasets that are rather small, lack of diversity, and significantly different from real-world software requirements. For example, the widely used dataset *HS* [14] is composed of source code (and the associated 'requirements') from a single software project, which results in poor diversity. The average length of the source code in dataset *Django* [25] is 33 characters only, suggesting that the software programs in the dataset have very limited complexity. Natural language descriptions in dataset *CoNaLa* [16] are how-to questions automatically extracted from *Stack Overflow*, instead of real-world software requirements. As a result, evaluating existing approaches on such datasets may fail to reveal the state of the art.

To this end, in this paper, we build a large scale dataset and evaluate deep learning-based code generation approaches on it. Compared to existing datasets, our dataset is composed of longer and more comprehensive software requirements accompanied by their validated implementations (source code). We also develop an assisting tool to assess comprehensively the quality of the generated source code instead of simply counting the lexical similarity between the generated source code and reference implementations. The benefits of this dataset and its assisting tool are twofold. On one side, we can reassess the state of the art of deep learning-based code generation with the resulting dataset and assisting tool. On the other side, the resulting data set may serve as a publicly available training/testing dataset for future research in code generation. Lacking of large scale and high-quality datasets is preventing deep learning-based code generation approaches from reaching their maximal potential. The resulting dataset is an initial attempt to solve this problem.

We reassess the state of the art in code generation with our new dataset. Evaluation results suggest that the performance of the state-of-the-art approaches on our dataset is significantly lower than that on existing datasets. The programs generated by such approaches are significantly different from reference implementations, often contain syntactic and semantical errors, and fail to pass even a single predefined test case. We replace the input (requirements) with random noise, and the performance of the evaluated approaches is still comparable to that before the replacement. It may suggest that the evaluated approaches learn little from software requirements. Further analysis of the generated source code suggests that most of the successfully generated statements are popular statements in the training programs. Based on this finding, we propose a popularity-based approach that always generates the most popular statements in the training programs regardless of the input (software requirements). Evaluation results suggest that none of the state-of-the-art approaches can outperform this simple statistics-based approach.

The paper makes the following contributions:

- A large scale dataset for learning-based code generation. Compared to existing ones, it is larger and has improved diversity as well as validated programs. Besides that, the requirements in the dataset are longer than those in existing datasets.
- An assisting tool kit to assess the quality of generated programs. Unlike existing approaches that heavily rely on lexical similarity, the tool kit employs static syntactic checking, dynamic cross-validation, and lexical comparison to comprehensively assess the quality of generated source code.
- A comprehensive reassessment of the state of the art of deep learning-based code generation. Based on the new dataset and associated tool kit, we evaluate the state-of-the-art approaches. Evaluation results suggest that the generated programs are significantly different from references, and none of them can pass even a single test case associated with the dataset. It may suggest that deep learning-based program generation requires significant improvement in the future. Our dataset, as well as the assisting tool, may serve as a basis for future research in this direction.

The rest of the paper is structured as follows. Section 2 introduces related research. Section 3 introduces how we construct a new dataset. Section 4 specifies how we assess the state-of-the-art approaches on the resulting dataset whereas Section 5 presents the results. Section 6 discusses related issues. Section 7 makes conclusions.

## 2 Related Work

### 2.1 Generating Source Code From Requirements Text

As introduced in the preceding section, automatic generation of source code from requirements text has recently been a hot topic in both software engineering and artificial intelligence communities. To reduce the complexity of code generation, researchers try to limit the complexity of programming languages. As a result, many code generation approaches employ domain-specific languages (DSLs) to describe the generated source code [26], [27], [28], [29]. DSLs are much simpler than general-purpose programming languages, and thus DSL-based approaches often result in high accuracy in generating source code. However, DSLs are specific to predefined domains, and it is challenging to apply them to other domains [14].

Compared to DSLs, generating source code in general-purpose programming languages is more challenging. However, employing such languages results in a number of significant advantages [14]. First, such languages, e.g., Java, are well-known and widely used by developers, and thus developers can read and modify the generated source code expediently. Second, such languages are broadly applicable across domains. Third, such programming languages have better expression ability than DSLs, and thus could describe complex applications. Because of such significant advantages, researchers have proposed a number of approaches to generating source code in general-purpose programming languages [14], [22], [29], [30]. Ling *et al.* [14] propose a latent predictor network-based approach (called LPN) to generate source code in Python or Java. Evaluation results on MTG, HS, and Django suggest that the approach is accurate and the average BLEU is up to 0.776. To the best of our knowledge, it is the first one that generates source code in general-purpose programming languages. Yin and Neubig [22] propose a syntactic neural model (SNM), which for the first time leverages the syntax of target language as prior knowledge. Later, they propose *TRANX* that generalizes SNM from Python to other languages [31]. Rabinovich *et al.* [29] propose an abstract syntax network-based approach (called ASN). To the best of our knowledge, they are the first to employ multiple decoders in code generation, where different types of elements in abstract syntax trees are generated by different decoders. Stehnii [30] proposes a *tree-to-tree* model for code generation. The key insight of the approach is that requirements in English could be parsed into trees as well, and the parsing can help neural networks to better understand the requirements. Dong and Lapata [32] propose a structure-aware neural architecture (called *Coarse-to-Fine*) for code generation. They are the first to divide the decoding process of code generation into two stages: generating a sketch on the first stage, and generating other information (e.g., variables and parameters) on the second stage. Hayati *et al.* [33] propose an approach called ReCode. For the first time, they leverage the nearest neighbors for code generation. The key insight of the approach is that two highly similar requirements are likely to result in highly similar implementations. GrammerCNN [34] proposed by Sun *et al.* is the first to employ CNN-based decoders in code generation. Their evaluation results suggest that their approach improves the state of the art by five percentage.

Text-based code generation is also a hot topic in the software engineering community. Gvero and Kuncak [35] propose an approach, called *anyCode*, to synthesize Java expressions from free-form queries containing a mixture of English and Java. The purpose of *anyCode* is to help developers, especially new developers, to achieve a task of interest by leveraging related APIs. For example, the developer may type in "*copy file fname to bname*" where *fname* and *fname* are given file names. *AnyCode* would return Java expressions like "*FileUtils.copyFile(new File(fname), new File(bname))*". For a given query, *anyCode* selects a set of most likely API declarations according to the query and unigram models. After that, *anyCode* leverages probabilistic context free grammar and unigram model to unfold the declaration arguments of the selected APIs. Raghothaman *et al.* [36] propose another approach, called SWIM, to generate code snippets for given API-related natural language queries such as "*generate md5 hash code*". Different from *anyCode* that generates a single expression, SWIM can generate a code snippet containing a few statements. SWIM maps textual query into a set of APIs by leveraging a statistical model. To construct code snippets from the suggested APIs, SWIM collects structural call sequences for each API data type in projects on Github. From such pre-extracted call sequences, SWIM retrieves the one that is most similar to the suggested APIs based on cosine similarity. *T2API* proposed by Nguyen *et al.* [37] is also a statistics-based approach to synthesize API code snippets from textual queries. It differs from SWIM in the following aspects. First, it conducts context expansion to expand the related APIs. For example, if *Socket.open()* is in the initial set of APIs associated with the given query, *T2API* will add *Socket.close()* as related APIs as well because it frequently follows *Socket.open()*. Second, *T2API* presents code snippets as graphs, and generates code graphs instead of retrieving graphs/code snippets from a given library. Consequently, it may generate new API usages. Yan *et al.* [38] build a dataset and its associated tools for fair and convenient comparison among different query-based code search methods. Such approaches differ from the evaluated approaches in that they are often confined to APIs [35], [36], [37] and generate short code snippets (or even a single expression) only. As a result, they are not suitable for our scenario and thus they are not involved in the evaluation in Section 4.

## 2.2 Datasets for Code Generation From Requirements Text

It is well recognized that high-quality datasets are critical for learning-based code generation [14]. Consequently, researchers have built a number of datasets that contain textual description (requirements) as well as their implementations (source code). Table 1 presents an overview of existing datasets. The first column presents the names of the datasets. The second column presents a short explanation. The third column presents the program languages employed to describe the programs. The fourth and fifth columns present the numbers of software requirements and software programs, respectively. The last two columns present the average length of requirements and programs, respectively. Sample items (both requirements and their corresponding implementations) are presented in Table 2.

According to the programming languages involved in the datasets, existing datasets could be classified into two categories. The first category of datasets (i.e., ATIS, GEO, JOBS, and IFTTT) describes source code in domain-specific languages. ATIS [13] was initially built to evaluate air travel information systems. It is composed of database queries in English and the source code to accomplish the queries. Later, it is employed as a dataset for code generation [28] where the queries are taken as software requirements and the Lambda style source code is taken as reference implementation. GEO [39] and JOBS [15] are similar to ATIS [13] in that all of them are composed of database queries and their accomplishing source code in DSLs. IFTTT [40] is another DSL-based dataset. Source code (applets) within IFTTT follows a predefined pattern: IF *this* THEN *that*, and it is the reason why the dataset is called IFTTT. Such kind of applets are widely used to control devices (e.g., watches, smart phone, and lights). These DSL-based datasets together have significantly advanced research in code generation [28]. However, such datasets are domain-specific, and thus models trained on such datasets may not work in other domains.

TABLE 1
Existing Datasets for Requirements Text-Based Code Generation

| Dataset | Explanation | Programming Language | #REQs | # Programs | Average Length of REQs (in tokens) | Average Length of Programs character/LOC |
|---|---|---|---|---|---|---|
| ATIS | database query of traveling info | Lambda-style DSL | 5,373 | 5,373 | 11 | 64 / 1 |
| JOBS | database query of jobs | Prolog-style DSL | 640 | 640 | 9 | 52 / 1 |
| GEO | database query of geography | Lambda-style DSL | 880 | 880 | 7 | 45 / 1 |
| IFTTT | IF-*this*-THEN-*that* applets | If-Then Recipes | 86,960 | 86,960 | 7 | 62 / 1 |
| MTG | features from Magic the Gathering | Java | 13,297 | 13,297 | 58 | 981 / 31.4 |
| HS | features from Hearthstone | Python | 665 | 665 | 34 | 300 / 7.4 |
| Django | pseudo-code vs. code | Python | 18,805 | 18,805 | 14 | 33 / 1 |
| StaQC | how-to questions | Python | 1,441 | 2,169 | 10 | 247 / 10.1 |
| | | SQL | 1,221 | 2,056 | 10 | 218 / 10.2 |
| CoNaLa | how-to questions | Python | 2,879 | 2,879 | 14 | 39 / 1.1 |

**REQs: Requirements.*

The second category of datasets (i.e., Django, MTG, HS, StaQC, and CoNaLa) describes source code in general-purpose programming languages, e.g., Java and Python. Oda *et al.* [25] propose an automatic approach to generating pseudo-code from source code. They collect source code (Python statements) of a Python web framework called *Django* (available at https://www.djangoproject.com/), and generate pseudo-code automatically for each of the downloaded Python statements. Such Python statements accompanied by corresponding pseudo-code are later employed as code generation dataset [14], [22], [30], [30]. Different from *Django* that is a byproduct of a pseudo-code generation approach, MTG and HS are intentionally built for code generation [14]. MTG is built on a trading card game called *Magic the Gathering* [41]. Each item in MTG is composed of two parts: Textual description of a card (in English) and the source code associated with the card. HS is highly similar to MTG. The only difference is that HS is based on another card game called *Hearthstone* [42]. MTG and HS are frequently used in code generation tasks [14], [22], [29], [30]. Different from MTG and HS that are built on a given software project, StaQC [43] and CoNaLa [16] are created by mining QA forums (e.g., *Stack Overflow* [44]), i.e., extracting how-to questions and their code fragments in accepted answers.

Although such datasets employ general-purpose programming languages, they still have the following limitations:

- First, software requirements included in such datasets are essentially different from real ones in the industry. The '*requirements*' in Django are pseudo-code that is highly similar to the associated source code. Such pseudo-code is significantly different from requirements text. Translating requirements text into source code is much more challenging than the translation from pseudo-code to source code. Although programs in MTG are longer than those in our dataset, the '*requirements*' in MTG (and HS as well) are rather special: all of the '*requirements*' in the dataset together constitute the real complete requirements for a single application (Magic the Gathering). Consequently, models trained on MTG can generate only additional source code (i.e., expansion) for the given program. It is unlikely for them to generate programs that are irrelevant to the given program (Magic the Gathering). The '*requirements*' in StaQC and CoNaLa are automatically extracted how-to questions that are significantly different from common requirements text.
- Second, the requirements and their associated source code may not match exactly. For example, the source code extracted automatically from QA forums may not exactly fulfill the how-to questions in StaQC and CoNaLa.
- Finally, as shown in Table 1, programs within such datasets are rather short. It may suggest that programs in such datasets are of limited complexity.

As a result of the limitations, models trained on such datasets may fail to generate complex implementation for real-world software requirements. Assessing the state of the art on such datasets also suffers from significant threats to external validity.

## 2.3 Code Generation Based On Examples and Contexts

Code complete is to generate code expressions or short code snippets based on contexts, e.g., the source code preceding the locations where the suggested code should be inserted. Type-based code complete is widely supported by IDEs. For example, while developers type in "*System.*", IDEs like Eclipse would suggest a list of members (fields and methods) that could be accessed via *System*. More advanced approaches, like statistical language models, have been proposed to improve the accuracy of code complete [45], [46]. Such approaches are based on the assumption that source code, like natural languages, is likely to be repetitive and predictable [47]. To this end, the statistical language models, like n-gram, are employed to predict the next token or the next expression in code complete. Besides such generic

TABLE 2
Sample Requirements and Implementations From Existing Datasets

| Dataset | Sample Requirements | Corresponding Implementations |
|---|---|---|
| ATIS | dallas to san francisco leaving after 4 in the afternoon please | (lambda $0 e (and (>(departure time $0) 1600:ti) (from $0 dallas:ci) (to $0 san francisco:ci))) |
| JOBS | what microsoft jobs do not require a bscs? | answer(company(J,‘microsoft’), job(J), not((req deg(J, ‘bscs’)))) |
| GEO | what is the population of the state with the largest area? | (population:i (argmax $0 (state:t $0) (area:i $0))) |
| IFTTT | Turn on heater when temperature drops below 58 degree | TRIGGER:Weather - Current temperature drops below - ((Temperature (58)) (Degrees in (f)))<br>ACTION: WeMo Insight Switch - Turn on - ((Which switch? (“”))) |
| MTG | NAME: Mox Jet<br>ATK: NIL<br>DEF: NIL<br>COST: 0<br>DUR: NIL<br>TYPE: Artifact<br>PLAYER_CLS: Limited Edition Alpha<br>RACE: 262<br>RARITY: R<br>TAP: Add B to your mana pool. | public class MoxJet extends CardImpl {<br>public MoxJet(UUID ownerId) {<br>super(ownerId, 262, “Mox Jet”, Rarity.RARE, new<br>CardType[]CardType.ARTIFACT, “{0}”);<br>this.expansionSetCode = “LEA”;<br>this.addAbility(new BlackManaAbility());<br>}<br>public MoxJet(final MoxJet card) {<br>super(card);<br>}<br>@Override<br>public MoxJet copy() {<br>return new MoxJet(this);<br>}<br>} |
| HS | NAME: Acidic Swamp Ooze<br>ATK: 3<br>DEF: 2<br>COST: 2<br>DUR: -1<br>TYPE: Minion<br>PLAYER_CLS: Neutral<br>RACE: NIL<br>RARITY: Common<br>Battlecry: Destroy your opponent’s weapon. | class AcidicSwampOoze(MinionCard):<br>def __init__(self):<br>super().__init__(“Acidic Swamp Ooze”, 2, CHARACTER_CLASS.ALL,<br>CARD_RARITY.COMMON, battlecry=Battlecry(Destroy(),<br>WeaponSelector(EnemyPlayer())))<br>def create_minion(self, player):<br>return Minion(3, 2) |
| Django | call the function conf.copy, substitute it for params. | params = conf.copy() |
| StaQC | How to limit a number to be within a specified range? | def clamp(n, minn, maxn):<br>return max(min(maxn, n), minn) |
| CoNaLa | How to convert a list of multiple integers into a single integer? | r = int(‘’.join(map(str, x))) |

code complete approaches, some task-specific approaches have been proposed successfully to suggest specific tokens, like method names [48], [49] and arguments [50]

Code generation is also closely related to program synthesis that generates programs automatically according to input/output examples [51], called *programming by examples* [52]. For example, researchers have successfully synthesized string editing programs according to input/output examples [53], [54]. Feng *et al.* [55] propose a component-based approach to synthesis scripts from examples for table consolidation and transformation tasks. Feng *et al.* [56] propose a conflict-driven program synthesis technique that learns from past mistakes. Lee *et al.* [57] accelerate search-based program synthesis using learned probabilistic models. A systematic review of search-based program synthesis is available at [52]. Neural network-based program synthesis is one of the most promising directions in program synthesis [51], [58]. Balog *et al.* [59], however, propose *DeepCoder* that combines neural network-based program synthesis and search-based program synthesis. Their evaluation results suggest that the combination leads to an order of magnitude speedup over the Recurrent Neural Network approaches. The performance of such learning-based program synthesis approaches depends heavily on the performance of training data [58]. To improve the quality of training data, Shin *et al.* [58] propose an automatic approach

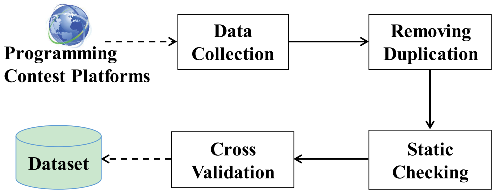


Fig. 1. Dataset creation.

to generate a high-quality dataset so that models trained on the resulting dataset could learn the full semantics of the selected DSL.

Representation of source code is also closely related to code generation [60]. The intuitive and straightforward representation of source code is to take it as natural language text (tokens) [47]. However, such plain text-based representation ignores the semantics of programs and the structures of source code. To this end, new approaches have been proposed to represent source code based on abstract syntax trees (AST) [60]. More advanced approaches can event leverage the paths within the AST trees [61], [62] and dependency among different source code elements [63].

## 3 New Dataset

As introduced in Section 2.2, existing datasets are preventing deep learning-based code generation approaches from reaching their maximal potential. To this end, in this section, we build a new dataset, as well as an assisting tool kit, for learning-based code generation.

### 3.1 Overview

Fig. 1 presents an overview on how we create the dataset for code generation. First, we extract task descriptions (software requirements) and their associated submissions from programming contest platforms. Second, we detect and remove duplicate tasks and duplicate implementations from the resulting dataset. Third, we compile the downloaded source code to make sure that the remaining source code is compilable. Third, we apply cross-validation to exclude incorrect implementations. Details of the creation are presented in the following sections.

### 3.2 Data Collection

We collect data from two programming contest platforms, i.e., Codeforces [64] and HackerEarth [65] because of the following reasons:

- First, the contests (software requirements) and their corresponding submissions (source code) on such platforms are publicly available;
- Second, the contests cover different topics instead of being confined to a specific domain, which may increase the diversity of the resulting dataset;
- Third, such platforms have manually designed test cases for each of the contests to ensure the correctness of the submissions, which may reduce the likelihood to include incorrect implementations in the resulting dataset;
- Finally, the contests are moderately challenging for automatic code generation. On one side, they are much more complex than most of the existing datasets whose implementation is often composed of only a couple of lines. Consequently, compared against existing datasets, the resulting dataset is more complex. On the other side, such contests are intentionally designed for beginners, and thus the complexity is limited. The limited complexity makes it potentially practical for deep learning techniques to generate the source code automatically.

With a Python-based crawler, we collect programming tasks (in English) from the selected platforms. We also collect their submissions (implementations) that have passed all of the associated test cases. The submissions are described in different programming languages, e.g., Java, Python, and C/C++. Comments within the source code are removed automatically because we focus only on code generation in our current work. Notably, for a single contest, there are often a large number (hundreds) of submissions. We only download its first $N$ submissions for each programming language. This number (empirically set to ten) is a result of the balance between the diversity of implementations and the size of the resulting dataset. We manually check the diversity of implementations (i.e., differences in algorithms, program structures, and identifiers) and find that the diversity increases significantly when $N$ increases from 1 to 10 whereas the diversity increases slightly when $N$ increases from 10. Consequently, we empirically set $N$ to 10. Notably, the diversity of dataset is not to increase the challenges in code generation, but to prevent overfitting of machine learning models.

An illustrating example task[1] is presented as follows:

> *You are given array consisting of n integers. Your task is to find the maximum length of an increasing subarray of the given array. A subarray is the sequence of consecutive elements of the array. Subarray is called increasing if each element of this subarray strictly greater than previous.*
> *Input: The first line contains single positive integer n — the number of integers. The second line contains n positive integers* $a_1, a_2, \ldots, a_n$ $(1 \leq a_i \leq 10^9)$.
> *Output: Print the maximum length of an increasing subarray of the given array.*

### 3.3 Removing Duplication

First, we detect and remove duplicate or nearly duplicate tasks from the resulting dataset. To avoid the pairwise comparison among thousands of tasks, we employ the well-known fingerprint algorithm *SimHash* [66] to transform textual description of each task into a fixed-length hash value (called fingerprint). The algorithm guarantees that fingerprints of nearly duplicate texts differ from each other in a small number of bit positions [66]. For each pair of the highly similar fingerprints, we manually check the corresponding tasks to exclude duplicate tasks only. Manual checking is conducted because two different tasks may happen to be lexically similar to each other, but the functionality of the intended software applications are essentially different. Consequently, the first two authors manually check the highly similar tasks. Two tasks $ts_1$ and $ts_2$ are duplicate if applications conforming to $ts_1$ conform to $ts_2$ as well, and vice versa.

[1] http://codeforces.com/problemset/problem/702/A

TABLE 3
Resulting Dataset

| Dataset | Explanation | #REQs | # Programs | Programming Language | Average Length of REQs (in tokens) | Average Length of Programs character/LOC |
|---|---|---|---|---|---|---|
| ReCa | from programing contests | 5,149 | 20,554 | C | 185 | 458 / 35.7 |
| | | | 35,092 | C++ | | 578 / 37.2 |
| | | | 32,306 | Java | | 1,121 / 63.8 |
| | | | 16,673 | Python | | 205 / 13.9 |

Second, we detect and remove duplicate implementations. For each of the tasks, we compare each pair of its submissions to exclude duplicate or nearly duplicate submissions. The comparison is based on the well-known edit distance [67] between two source code fragments:[2] if the distance is small, i.e., changing a few characters in one fragment can turn it into the other fragment, they are reported as potentially redundant implementations. Before removing such potentially duplicate implementations, the first two authors also manually check them to exclude false positives: two implementations of the same task are duplicate if and only if they are identical except for the difference in format (e.g., blank lines) and/or code comments.

### 3.4 Static Checking

Both of the websites have a long history, and thus some of the old submissions to the websites may be out of date and cannot be compiled with the up to date compilers. Assuming that up-to-date code generation approaches may target up-to-date compilers only, we filter out such outdated submissions by compiling them with the up-to-date compilers.

By compiling the submissions, we also remove low-quality submissions that result in warnings. Such submissions could be compiled and thus may be executed. However, warnings (e.g., dead code) reported by compilers suggest that such submissions deserve improvement. Consequently, to guarantee high-quality of the resulting dataset, we exclude such submissions that result in compiler's warnings. The exclusion, in turn, may improve the quality of code generation models trained on the resulting dataset (i.e., fewer compiler's warnings on the generated source code).

### 3.5 Cross-Validation by Software Testing

One of the biggest challenges in building code generation datasets is to guarantee that the included programs act exactly the same as what their associated software requirements specify. In other words, such programs should be accepted as correct implementations by users who propose the requirements. In our case, all submissions are specifically designed for the given tasks, and the websites have run some manually predefined test cases to guarantee that the submissions satisfy the requirements in the most common scenarios. This helps to improve the reliability of the resulting dataset. However, the number of such manually designed test cases is rather small, and thus it is likely that some buggy submissions can still pass such test cases.

To further improve the reliability, we carry out cross-validation by software testing. For each of the task $t$, the cross-validation is conducted as follows:

- First, according to the requirements we manually create a template to specify the input parameters, including their data types and value ranges.
- Second, based on the template, we automatically create test cases with fuzz testing, i.e., create random data as inputs to the programs under test.
- Third, for each test case, we automatically run each of the submissions (to the given task $t$) with the inputs in the test case. If the submissions result in different outputs, we manually check the results and remove the buggy submissions.

Notably, we do not employ popular test case generation tools like *EvoSuite*. The downloaded programs often receive input from *console* with statements like '*input()*' (Python) and '*scanf*' (C/C++). However, existing test case generation tools like *EvoSuite* generate test cases (more specially, input of the programs) according to parameters instead of console input. As a result, applying such tools to the downloaded programs results in few test cases. To this end, we manually create a template for each task to explicitly specify its input (including those from console), and generate test cases automatically based on the template.

### 3.6 Resulting Dataset

We call the resulting dataset *large scale dataset for Requirements text based Code generation* (ReCa). Details of the dataset is presented in Table 3. By comparing Table 3 against existing datasets in Table 1 (especially MTG, HS, Django, StaQC, and CoNala that describe implementations in general-purpose programming languages), we observe that our dataset has the following advantages:

- First, ReCa is composed of longer requirements of independent software applications. The average length of requirements in our dataset is significantly longer than that of existing datasets. As analyzed in the preceding sections, the textual descriptions in MTG, HS, Django, StaQC, or CoNala are not real-world software requirements. Instead, they are incremental features of a single software project (MTG and HS), pseudo-code (Django), or how-to questions (StaQC and CoNala). In contrast, each of the textual descriptions in our dataset represents a requirement of an independent software application. Software engineers have developed the intended applications successfully according to such textual descriptions.

[2] More advanced tools like MOSS (http://theory.stanford.edu/aiken/moss/) may ease the work.

- Second, ReCa contains more programs. Our dataset contains more than one hundred thousand software programs, much larger than existing datasets.
- Third, ReCa has longer programs. For example, the average length of Java programs in our dataset is 63.8 (LOC), much longer (at least twice) than that of existing datasets. Notably, however, such programs are still significantly smaller than real-world software applications in the industry. These real-world applications may contain millions of lines of source code, which makes them extremely challenging (if not impossible) to be generated automatically by up-to-date deep learning models.
- Fourth, the implementations in ReCa are in multiple general-purpose programming languages. For most of the requirements in our dataset, we provide corresponding implementations in different programming languages at the same time, e.g., Java and Python. It may facilitate the research on cross language code generation, as well as research on the impact of programming languages on code generation.
- Fifth, the implementations in ReCa are validated. Each of the implementations in our dataset has been validated by static checking as well as dynamic software testing to guarantee that they satisfy the declared requirements and they are of high quality.
- Sixth, ReCa provides multiple implementations for the same requirements. A single requirement has up to ten independent implementations in the same language (e.g., Java). Trained on such a dataset, learning-based code generation algorithms may learn equivalence among different code fragments, and thus may be smarter in appreciating the context while generating the next code token. Multiple reference implementations also facilitate more reasonable and more comprehensive quality assessment on generated programs by comparing them against diverse references. Existing approaches often assess the quality of a generated program by computing its lexical similarity (e.g., BLEU) with a single reference because existing datasets often provide a single reference only. The assessment is risky because two semantically equivalent programs may happen to be significantly different in text. Providing a number of diverse references helps to reduce the risk.

### 3.7 Quality Assessment and Tool Kit

To facilitate research on code generation, we develop an assisting tool to comprehensively assess the quality of generated programs. The tool computes automatically a list of quality metrics for the generated program against its reference programs. The first quality metrics are BLEU (bilingual evaluation understudy) [23] that is widely employed by existing approaches. BLEU was initially proposed to assess the quality of machine translation [23]. For code generation, BLEU scores are calculated by comparing the generated source code against a set of reference programs. The scores range between 0 and 1, suggesting how lexically similar the generated program is to the reference programs. Notably, BLEU for a generated program $p$ is the maximal similarity (BLEU) between $p$ and any of its reference programs: If it is highly similar (or even identical) to any of its reference programs, the generated program is of high quality even if it is essentially different from other reference programs.

The second code metrics are the number of errors and warnings compilers produce on the generated source code. Existing approaches rarely employ such metrics because most of the generated source code cannot be compiled successfully at all, i.e., they often contain syntactic errors. One of the reasons for such syntactic errors is that most of the reference programs (e.g., code fragments from Stack Overflow) in existing datasets are incomplete and thus cannot be compiled successfully. Consequently, code generation models trained on such datasets rarely generate compilable programs.

The third code metrics are the percentage of passed test cases, i.e., what percentages of the test cases the generate program has passed. In our dataset, we have generated automatically a large number of test cases for each of the tasks (requirements). Consequently, we can run such test cases on the generated programs to assess the extent to which the generated programs satisfy the functional requirements.

The fourth quality metrics are the edit distance-based lexical similarity. Levenshtein distance is widely employed to measure the minimum number of single-character edits (i.e., insertions, deletions, or substitutions) required to change one text (the generated source code in our case) into the other (reference implementation in our case). The edit distance-based lexical similarity (noted as $S_{ed}$) turns the Levenshtein distance (note as $dis$) into a similarity varying from zero to one: $S_{ed}(gc, ref) = 1 - \text{dis(gc,ref)}/max(|gc|, |ref|)$ where $gc$ is the generated source code and $ref$ is a reference implementation.

BLEU is selected because it is widely employed by existing approaches to evaluate the quality of code generation [23]. The number of compiler errors and warnings (the second metrics) is selected because it represents the syntactic quality of the generated source code. The percentage of passed test cases (the third metrics) is selected because it represents the functional quality of the generated source code. The edit distance is selected because it is widely used to measure the similarity between source code. BLEU and edit distance concern the lexical similarity between generated programs and references whereas the number of compiler errors and the percentages of passed test cases concerns the syntactics and functionality of the generated programs, respectively. Consequently, employing such diverse metrics facilitates comprehensive assessment of the generated programs. To facilitate more comprehensive assessment, however, the tool kit also provides additional metrics, i.e., NIST, WER, and Subtree Metric [68].

We employ additional quality metrics (as introduced in preceding paragraphs) besides BLEU for assessing the quality of generated source code because of the following reasons:

- First, although BLEU is frequently employed to assess the quality of generated source code, it has significant limitations [69] for assessing source code. Unlike nature languages, source code has less tolerance for noise and poor syntax/semantics. Consequently, programs with high BLEU could be syntactically incorrect and essentially different from reference programs in semantics.

- Second, even the implementations for the same task (requirements) are often diverse in text. Consequently, computing the lexical similarity between the generated source code and its diverse reference implementations may fail to reveal the quality of code generation.

## 4 Experimental Setup

As introduced in Section 2, researchers have achieved great advances recently in deep learning-based code generation. A number of approaches have been proposed, and evaluation results on different datasets suggest that they are highly accurate. For example, the syntactic neural model proposed by Yin and Neubig achieves a high BLEU (0.845) on *Django* dataset [22], which suggests that the generated source code is very close to the reference implementation. However, as introduced in the preceding sections, such datasets employed in the evaluations have significant limitations and thus good performance on such datasets may not necessarily lead to good performance in handling real-world software requirements. To assess the state of the art, in this section we evaluate such approaches on our new dataset.

### 4.1 Validation Questions

The evaluation investigates the following questions:

- *Q1*: How accurate are the state-of-the-art approaches on the new dataset?
- *Q2*: How often do the generated programs pass syntactic checking?
- *Q3*: How often do the generated programs pass predefined test cases?
- *Q4*: Is the generated source code useful for developers?
- *Q5*: Where and why do state-of-the-art approaches succeed?
- *Q6*: To what extent do state-of-the-art approaches understand software requirements?
- *Q7*: Can we propose a simple and intuitive approach whose performance is comparable to (or even better than) that of the state-of-the-art approaches?
- *Q8*: Can we improve the performance of the evaluated approaches if we keep only a single solution per requirement?
- *Q9*: Can we improve the performance of the evaluated approaches by unifying identifiers in requirements and associated source code?

Research question Q1 concerns the performance of the state-of-the-art approaches on our new dataset. Many of the state-of-the-art approaches are reported to be highly accurate on existing datasets [22]. Answering this question may reveal whether the reported high performance is owned to the limitations of the involved datasets.

Research question Q2 investigates how often the deep learning-based approaches generate syntactically correct programs, and how often such programs could be executed without exceptions. Investigating Q2 would reveal to what extent such approaches learn automatically the syntax of target programming languages.

Research question Q3 investigates to what extent the generated programs are semantically correct, i.e., consistent with the given software requirements. The investigation would reveal to what extent the approaches can learn the semantics of requirements that are described in English, and turn such semantics into implementations.

Research question Q4 investigates the usefulness of the generated programs. It is likely that developers cannot use the generated code as-is. However, if the effort to modify it to make it work is much smaller than the effort to write the correct code from scratch, the generated source code (and the generation approaches) could be considered useful.

Research question Q5 investigates what kind of tokens could be generated correctly, and potential reasons for the success. The investigation will reveal the strength of existing approaches, and the rational for the strength.

Research question Q6 investigates the influence of the input (textual requirements) on the output (generated source code). It is challenging for computers to fully understand natural languages. Consequently, it is likely that the deep learning-based code generation approaches cannot fully understand the requirements in English. Answering this question may reveal whether natural language understanding is the major obstacle to deep learning-based code generation.

Research question Q7 concerns the substitutability of the state-of-the-art deep learning-based complex approaches. Answering this question may reveal whether such deep learning-based complex approaches are really better (or much better) than simple and intuitive approaches.

Research question Q8 concerns the effect of removing redundant implementations for the same requirement. While answering the preceding research questions, we provide the evaluated approaches with multiple code snippets/solutions for the same requirement. However, this has not been done by the authors of the evaluated approaches and the loss function of the approaches is not prepared for this. Consequently, such neural networks may fail to learn anything from such different implementations. To this end, we repeat the evaluation after removing the redundant implementations, i.e., we keep only a single solution per requirement.

Research question Q9 concerns the identifiers in requirements and their associated implementation (source code). Such identifiers do not influence the syntax or semantics of programs. However, replacing them with unified tokens e.g., $var_0$ and $var_1$, could significantly reduce the size of vocabularies employed by automated code generation approaches. Research question Q9 investigates the effect of the preprocessing.

### 4.2 Evaluated Approaches

We select Seq2Seq [28], SNM [22], Tree2Tree [30], TRANX [31] and Coasr-to-Fine [32] for the evaluation because of the following reasons.

- First, they could generate source code in general-purpose programming languages according to software requirements in English, which makes it practical for them to work on our dataset.
- Second, their implementation is publicly available, which significantly facilitates the evaluation. Some well-known approaches [14], [29], [34] that could

generate source code from requirements text are not selected for evaluation because we either fail to get their implementations [14], [29] or fail to make them work on our dataset [34].

- Third, SNM [22], Tree2Tree [30], TRANX [31] and Coarse-to-Fine [32] were proposed recently, and represent the state of the art. To the best of our knowledge, they are the latest approaches that 1) have publicly available implementations and 2) can work on our dataset to generate Python programs according to requirements text.
- Although Seq2Seq [28] was initially proposed for semantic parsing, it is widely employed as a baseline in code generation [33]. Consequently, we include it for the evaluation as well.

### 4.3 Process

To investigate questions Q1, Q2, Q3, and Q5, we conduct the first empirical study as follows:

- First, we select all tasks for evaluation from our dataset that are accompanied by Python source code. To the best of our knowledge, no publicly available deep learning-based models/implementations can transform requirements text into programs in general-purpose programming languages other than Python. Although LPN [14] generates Java programs, its implementation is unavailable. Consequently, we select only Python programs for the empirical study. The resulting dataset is noted as *selected dataset*. It is composed of 2,740 tasks (requirements) and 16,673 Python programs.
- Second, from the selected dataset, we randomly select 300 tasks as testing dataset, 200 tasks as validation dataset, and other as training dataset.
- Third, the selected dataset is preprocessed. For the textual requirements, we leverage NLTK [70], [71] to replace acronyms (e.g.," *what's*") with separated words (e.g., "*what is*"), to turn characters into lowercase, to split the text into a sequence of word by word segmentation and special characters (e.g., splitting "*Java.System*" according to "."), to remove stop words, and to apply lemmatization on the resulting words. For example, the requirement:

  > "*Some natural number was written on the board. Its sum of digits was not less than k. But you were distracted a bit, and someone changed this number to n, replacing some digits with others. It's known that the length of the number didn't change.You have to find the minimum number of digits in which these two numbers can differ.*"

  is finally turned into:

  > "*some natural number write on board. its sum of digit not less than k. but you distract bit, someone change number to n, replace some digit with others. it know length of number do not change. you have to find minimum number of digit in which these two number can differ.*"

  after the preprocessing. For the selected source code, we remove comments and copyright declarations, and format the source code (with Autopep8 [72]). Listing 1 presents an illustrating example of source code preprocessing: the code before and after preprocessing.

**Listing 1.** Example of Source Code Preprocessing

```
/* Code before preprocessing */:
#␣get␣number
w␣=␣int(input())
if␣w%2␣==␣0␣and␣w!=2:
␣␣print(’YES’)
else:
␣␣print(’NO’)

/* Code after preprocessing */:
w␣=␣int(input())
if␣w␣%␣2␣==␣0␣and␣w␣!=␣2:
␣␣␣␣print(’YES’)
else:
␣␣␣␣print(’NO’)
```

- Fourth, for each of the selected approaches, we train it on the training and validation datasets, and test it on the testing dataset.
- Finally, we evaluate the quality of generated source code with the tool kit introduced in the preceding sections. The quality metrics generated by the tool kit are subsequently employed to answer the research questions.

To maximize the potential of the evaluated approaches, we perform hyper parameter tuning for each of the evaluated approaches. Basically, we follow the grid-search tuning approach [73] but pick up grids (i.e., to-be-tested values of parameters) dynamically and empirically to speed up the tuning process. Notably, for each of the to-be-tested setting, we train the selected approach with the given setting on the given training data (all of the requirements-code pairs regardless of their topics), and then validate the performance on the validation set. Based on the validation, we empirically select the next to-be-tested setting. For a given setting, we train the associated approach with the setting once and for all (instead of repeating the training and validation for several times) because the training is highly time-consuming: For each of the evaluated approaches, it takes more than one week to tune its hyper parameters on a GPU server (OS: Ubuntu 14.04.5; CPU: 56 * Intel(R) Xeon(R) CPU E5-2683 v3 @ 2.00GHz; GPU: 2* TITAN Xp; RAM: 64GB). The final parameters are presented in Table 4 where N/A suggests that the implementation of the given approach does not contain the parameter or the parameter does not deserve tuning.

To investigate question Q4, we randomly select eleven tasks from the dataset and invite thirty developers to conduct a controlled experiment. The participants have rich experience in Python. They did not know the intent of the experiment in advance, which helps to reduce potential bias. The experiment is conducted as follows:

- First, each of the participants is requested to code from scratch for a selected task (noted as *preTestTask*), and we record the time that developers take to finish the assigned task. Notably, a task is finished only if the submitted program has passed all predefined test cases. The top five (who spend the shortest

TABLE 4
Final Parameters for Evaluated Approaches

| Approaches \ Parameters | Embedding Size | Hidden Size | Epoch | Batch Size | Decoder Dropout | Learning Rate | Learning Rate Decay |
|---|---|---|---|---|---|---|---|
| Seq2Seq | 200 | N/A | 120 | 20 | 0.4 | 0.01 | 0.98 |
| SNM | 256 | 256 | 100 | 7 | 0.4 | 0.001 | N/A |
| Tree2Tree | 300 | 256 | 100 | 8 | 0.2 | 0.001 | N/A |
| TRANX | 128 | 256 | 100 | 10 | 0.3 | 0.001 | 0.5 |
| Coarse-to-Fine | 250 | N/A | 75 | 10 | 0.3 | 0.002 | 0.99 |

TABLE 5
Evaluation Results on Our Dataset

| Approaches | BLEU on New Dataset | BLEU on Django | BLEU on HS | Syntactically Correct Programs | Executable Programs | Functionally Correct Programs |
|---|---|---|---|---|---|---|
| Seq2Seq | 0.138 | 0.673 | 0.550 | 44.7% | 6.0% | 0% |
| SNM | 0.188 | 0.845 | 0.758 | 93.0% | 16.7% | 0% |
| Tree2Tree | 0.150 | 0.825 | 0.716 | 83.7% | 14.3% | 0% |
| TRANX | 0.184 | 0.856 | 0.695 | 81.7% | 9.0% | 0% |
| Coarse-to-Fine | 0.176 | 0.854 | 0.640 | 10.0% | 2.7% | 0% |
| Average | 0.167 | 0.811 | 0.672 | 62.6% | 9.7% | 0% |

time in finishing the given task) and the bottom five (who spent the longest time) are excluded from further evaluation. The other twenty participants are divided into two equally sized groups according to their coding speed: participants in *Group A* is faster than anyone from *Group B*. The purpose of this step is to construct two participant groups where participants within the same group have similar coding speed. Grouping the participants by coding speed may reduce the bias introduced by the difference in participants' programming ability/speed.

- Second, for each of the selected participants, we request him/her to code from scratch for five out of the remaining ten tasks (i.e., all selected tasks except for *preTestTask*), and to complete the other five tasks based on programs generated by *SNM*. *SNM* is selected because it achieves the best performance among the evaluated approaches (see Section 5.1 and Table 5 for details). The assignments of the tasks guarantee that exactly half of the participants from each group finish a task from scratch and another half the participants from the same group finish the same task by modifying the generated program.
- Third, we record the time that developers take to finish the assigned tasks, and analyze the results of the two groups.

To investigate question Q6, i.e., to what extent the input (software requirements) is exploited by code generation approaches, we conduct the third empirical study as follows:

- First, for each of the selected approaches, we train it on the training and validation datasets, and test it on the testing dataset (noted as $tdata$). We compute their performance, and call it *original performance* (noted as $P_{orig}$). Notably, it is the same as we do in the previous empirical study.
- Second, for each item (composed of requirements and their implementations) in the testing dataset $tdata$, we replace the requirements with random noise. The noise is generated automatically by picking tokens randomly from a large corpus. The length (in tokens) of the noise is equal to the original requirements. The resulting testing dataset is noted as $tdata'$.
- Third, on the revised testing dataset $tdata'$, we evaluate the resulting models that are trained on the first step. The resulting performance (noted as $P_{noise}$) is compared against the original one (i.e., $P_{orig}$).

If replacing requirements with random noise fails to reduce the performance significantly, it is likely that the code generation approaches learn little from the requirements.

To investigate question Q8, i.e., the effect of removing redundant implements, we conduct the fourth empirical study as follows:

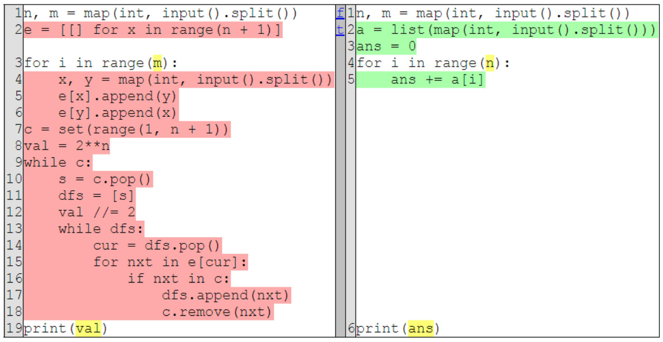
- First, for each of the tasks in the dataset, we randomly select and keep one of its implementations. Other implementations are removed from the dataset. We call the resulting dataset as *nonredundant dataset*
- Second, we repeat the first empirical study on the *nonredundant dataset*.
- Third, we compare the resulting performance on *nonredundant dataset* against that on the original dataset where multiple implementations for the same tasks are exploited.

## 5 Results and Analysis

### 5.1 Q1: Significant Reduction in BLEU

To answer question Q1, we evaluate the state-of-the-art code generation approaches on our new dataset, and evaluation results are presented in Table 5. The first column presents the evaluated approaches. The second column presents BLEU of such approaches on our new dataset. To facilitate comparison, we also present their BLEU on existing datasets (i.e., Django and HS) on the third and fourth columns. Columns 5-7 present the syntactical and semantic checking results of the generated source code (on our dataset).

From the first four columns, we make the following observations:



```
n, m = map(int, input().split())
e = [[] for x in range(n + 1)]

for i in range(m):
    x, y = map(int, input().split())
    e[x].append(y)
    e[y].append(x)
c = set(range(1, n + 1))
val = 2**n
while c:
    s = c.pop()
    dfs = [s]
    val //= 2
    while dfs:
        cur = dfs.pop()
        for nxt in e[cur]:
            if nxt in c:
                dfs.append(nxt)
                c.remove(nxt)
print(val)
```

```
n, m = map(int, input().split())
a = list(map(int, input().split()))
ans = 0
for i in range(n):
    ans += a[i]
print(ans)
```

Fig. 2. Visual comparison between reference implementation (left) and generated program (right).

- First, BLEU of the evaluated approaches is rather low. It varies from 0.138 to 0.188, with an average of 0.167. Such a low BLEU suggests that the generated source code is often significantly different from reference implementations, i.e., the validated implementations in the dataset.
- Second, switching from existing datasets to our new dataset reduces BLEU significantly. The average BLEU is reduced significantly from 0.811 (on Django) and 0.646 (on HS) to 0.167. The reduction is up to 79%=(0.811-0.167)/0.811, and 75%=(0.672-0.167)/0.672, respectively.

To figure out the reason for low BLEU, we employ *diff*, a popular and powerful tool, to visualize the difference between the generated programs and their references. A typical example is presented in Fig. 2. The right part of the figure presents the program generated by *SNM*. The left part presents a reference implementation that has the greatest BLEU with the generated one. The common part (i.e., successfully generated statements) is shown on white background. Missing part (i.e., statements that should have been generated) is shown on red background, added part (i.e., statements that should not have been generated) is on green background, and the modified part is on yellow background.

We randomly sample 100 generated programs for visual comparison. Based on the comparison, we make the following observations:

- First, most statements are not generated successfully. Around 75 percent of the statements in reference programs are missing in the generated programs. For example, in Fig. 2 sixteen out of the nineteen (84%=16/19) lines of source code in the reference implementation are on red background, suggesting that the evaluated approach fails to generate the majority of the reference implementation.
- Second, most of the generated source code is irrelevant, i.e., having no counterparts in the reference implementations. Around 81 percent of the generated source code is irrelevant (on green background). For example, in Fig. 2 three out of the six (50%=3/6) lines of source code in the generated program are on green background.

To validate whether the observations could be generalized to all generated programs, we compute automatically how often tokens in reference implementations are missed (i.e., shown on red or yellow background), and how often tokens in the generated programs are irrelevant (i.e., shown on green or yellow background). Results are presented in Table 6. The first column presents evaluated approaches. The second column presents the percentages of the tokens in the reference programs that are missed by the generated programs. The third column presents the percentages of the tokens in the generated programs that are irrelevant, i.e., having no counterparts in the reference implementations. From this table, we observe that on average, 76.9 percent of the tokens in reference implementations are missed, and 82.7 percent of the tokens in generated programs are irrevevent. In other words, only 23.1%(=1-76.9%) of the tokens in the reference implementations are generated successfully, and only 17.3%=(1-82.7%) of the generated programs tokens are really useful. The statistics confirm our preceding observation that the generated programs are often significantly different from references.

TABLE 6
Mismatch Between Generated Programs and References

| Approaches | Missing Tokens | Irrelevant Tokens |
|---|---|---|
| Seq2Seq | 81.8% | 87.5% |
| SNM | 71.2% | 74.8% |
| Tree2Tree | 75.4% | 81.0% |
| TRANX | 74.4% | 81.5% |
| Coarse-to-Fine | 81.6% | 88.5% |
| Average | 76.9% | 82.7% |

We also employ additional metrics [68] (i.e., NIST, WER, and Subtree Metric) besides BLEU. Evaluation results are presented in Table 7. The results confirm the conclusions drawn on the preceding paragraphs: the performance of the evaluated approaches is not promising on the new dataset.

One potential cause of the low accuracy could be the irregularity of the tokens in the dataset. If tokens in the testing dataset are often missing in the training dataset, it is likely that machine learning model cannot generate tokens accurately. To this end, we compare the vocabularies of the training dataset and testing dataset. The comparison results suggest that 96 percent of the (requirements) text tokens in the testing dataset are actually observed in the training dataset whereas 79 percent of the source code tokens are observed in the training dataset. The results may suggest that the difference in vocabularies of training data and testing data is not the major reason for low accuracy.

It is quite intuitive that the longer the text and programs are, the lower the generation accuracy would be. To quantitatively verify this, we partition the tasks into four equally sized groups according to their length of requirements and length of reference programs, respectively. Notably, for a single task, we have multiple reference implementations (programs). Consequently, we classify the task based on the average length of its reference programs (instead of the length of a single reference program). Evaluation results are presented in Tables 8 and 9. On Table 8, we present how the length of requirements influences the performance (BLEU). Q1 contains 75=300/4 tasks that have the shortest requirements whereas Q4 contains 75 tasks with the longest requirements. Each row of the table presents the performance (BLEU) of an evaluated approach on different groups of tasks. From this table, we make the following observations:

TABLE 7
Evaluation Results with Additional Metrics

| Metrics / Approaches | NIST | WER | Subtree |
|---|---|---|---|
| Seq2Seq | 1.369 | 8.089 | 0.117 |
| SNM | 1.453 | 0.785 | 0.200 |
| Tree2Tree | 1.185 | 0.866 | 0.139 |
| TRANX | 1.721 | 1.179 | 0.160 |
| Coarse-to-Fine | 1.988 | 1.643 | 0.116 |
| Average | 1.543 | 2.513 | 0.146 |

- First, all of the evaluated approaches result in the lowest performance on Q4 group that is composed of the longest requirements. It may suggest that extremely long requirements (varying from 228 tokens to 391 tokens) have significant negative impact on the performance of code generation.
- Second, all of the evaluated approaches result in the highest performance on Q2 group where the length of requirements varies from 125 tokens to 171 tokens. In contrast, they result in significantly lower performance on Q1 that is composed of the shortest requirements (varying from 16 tokens to 125 tokens). The results may suggest that the following assumption is not necessarily true: the shorter the requirements text is, the higher the generation accuracy would be.

Table 9 presents the influence of programs' length where Q1 contains 75 tasks with the shortest reference programs. From this table, we observe that the performance decreases with the increase of programs' length. The average BLEU reduces from 0.218 on Q1 (where the length of programs varies from 6 tokens to 77 tokens) to 0.0726 on Q4 (where the length of programs varies from 199 to 822 tokens). The evaluation results may suggest that the length of programs has a significant negative impact on the performance of automated code generation.

Based on the preceding analysis, we conclude that concerning the common performance metrics (i.e., BLEU) the state-of-the-art code generation approaches cannot reach a high performance on the new dataset as they do on existing datasets. Concerning other performance metrics like NIST, WER, STM, and Subtree metrics, the evaluated approaches also result in poor performance on the new dataset. Most tokens in reference implementations are missed whereas most of the generated tokens are irreverent. One possible reason for the significant reduction in performance is that the new dataset is more complex and more diverse than existing ones.

TABLE 8
Impact of Requirements' Length on BLEU

| Length of Requirements / Approaches | Q1 | Q2 | Q3 | Q4 |
|---|---|---|---|---|
| Seq2Seq | 0.131 | 0.157 | 0.141 | 0.125 |
| SNM | 0.195 | 0.214 | 0.185 | 0.156 |
| Tree2Tree | 0.155 | 0.173 | 0.146 | 0.125 |
| TRANX | 0.198 | 0.204 | 0.177 | 0.158 |
| Coarse-to-Fine | 0.164 | 0.192 | 0.182 | 0.167 |
| Average | 0.1686 | 0.188 | 0.1662 | 0.1462 |

TABLE 9
Impact of Programs' Length on BLEU

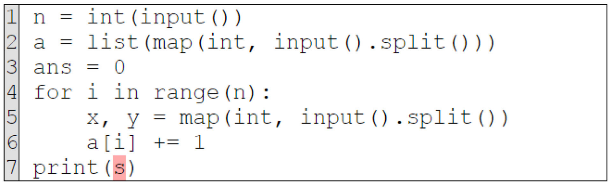
| Length of Reference Programs / Approaches | Q1 | Q2 | Q3 | Q4 |
|---|---|---|---|---|
| Seq2Seq | 0.150 | 0.181 | 0.146 | 0.076 |
| SNM | 0.287 | 0.240 | 0.170 | 0.054 |
| Tree2Tree | 0.257 | 0.192 | 0.120 | 0.029 |
| TRANX | 0.223 | 0.231 | 0.194 | 0.088 |
| Coarse-to-Fine | 0.173 | 0.215 | 0.201 | 0.116 |
| Average | 0.218 | 0.2118 | 0.1662 | 0.0726 |

### 5.2 Q2: Syntactic Checking

To answer question Q2, we conduct syntactic checking on the generated programs. The checking is composed of two parts. In the first part, we conduct static syntactic checking on the generated programs with the state-of-the-practice tool *Pylint* [74]. For convenience, we call programs that pass the static checking as *syntactically correct programs*. In the second part, we try to execute the programs (with sample input specified in the requirements) that pass the static checking on the first step. If the execution results in any syntactic error or runtime exception, the programs are non-executable. Results of the static syntactic checking are presented in the fifth column of Table 5 whereas the execution results are presented in the sixth column. From these two columns, we make the following observations.

The first observation is that most (up to 93.0 percent) of the programs generated by AST-based approaches (i.e., SNM, Tree2Tree, and TRANX) pass the static syntactic checking whereas programs generated by other approaches have significantly smaller chance (less than fifty percentage) to pass the static syntactic checking. The results may suggest that generating ASTs (and then transferring them into source code) helps much in avoiding syntactic errors. In contrast, generating source code (as generic text) directly is much riskier because the state-of-the-art approaches could not yet automatically recognize the complete syntax of programming languages that is embedded in the training programs.

To figure out what kind of syntax such approaches fail to learn automatically, we manually analyze the syntactic errors generated by such approaches. In general, the syntactic checking on generated programs (i.e., to compute how many of the generated programs are syntactically correct and how many of them are executable) is completely automated, and no manual checking is required. Manual checking is only employed to empirically reveal the common syntax errors in the generated programs. The results of the manual analysis suggest that *undefined-variable* is dominating. Undefined variable refers to usage of variables that have not yet been defined before the usage. An illustrating example is presented in Fig. 3 where *s* on Line 7 is *undefined*. *Undefined-variable* accounts for 59, 78, 70, 53, and 82 percent of the syntactical errors generated by *Seq2Seq*, *SNM*, *Tree2Tree*, *TRANX*, and *Coarse-to-Fine*, respectively. On average, it accounts for 68%(=2003/2965) of the syntactical errors we



```
n = int(input())
a = list(map(int, input().split()))
ans = 0
for i in range(n):
    x, y = map(int, input().split())
    a[i] += 1
print(s)
```

Fig. 3. *Undefined-variable* in generated program.

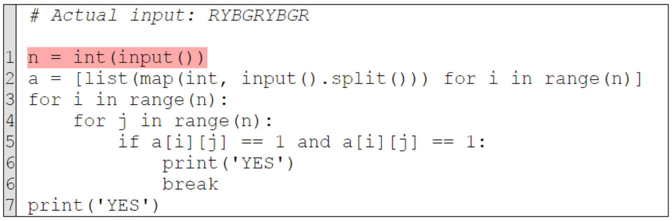
encounter during the evaluation. Consequently, to further improve the state of the art, researchers in the future should pay more attention to such kind of syntactic errors. For example, to reduce *Undefined-variable* errors, we may request the deep learning models to select from a short list of variables declared in the generated source code when variables are expected. In contrast, exiting models always select tokens from a generic large vocabulary, which often results in undefined-variables.

The second observation is that only a small part (less than 10 percent) of the generated programs could be executed without exceptions. Notably, Python is a dynamically typed programming language, and thus many type errors could not be identified by static syntactic checking. As a result, programs that pass static syntactic checking may still fail to run successfully. To figure out what kind of problems are preventing such programs from successful execution, we manually analyze the runtime exceptions that we encounter while executing such programs. Notably, we do not execute those who fail to pass static syntactic checking because they are bound to fail. The results of our analysis suggest that most of the exceptions are *ValueError*. According to Python documents [75], *ValueError* exception is 'raised when an operation or function receives an argument that has the right type but an inappropriate value, and the situation is not described by a more precise exception such as IndexError'. An illustrating example is presented in Fig. 4. The first input statement on Line 1 expects a string that could be parsed into an integer. However, the actual input "RYBGRYBGR" fails, and thus a *ValueError* is raised. *ValueError* exceptions account for 72.5%(=578/797) of the exceptions encountered during the evaluation.

Based on the preceding analysis, we conclude that AST-based code generation approaches have a great chance to generate syntactically correct Python programs. However, such programs are often non-executable because of various runtime exceptions.

### 5.3 Q3: Dynamic Validation

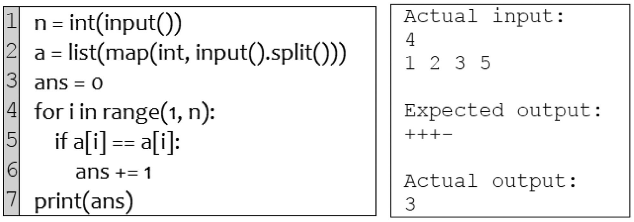
To answer question Q3, we run test cases in the dataset on the generated programs. Results are presented in the last column of Table 5. From this column, we observe that none of the generated programs passes any test case in the dataset. The results may suggest that even if some of the generated programs are syntactically correct and executable, they fail to fulfill the given requirements. One of the possible reasons for the failure is that the evaluated approaches do not really understand the software requirements (details are presented in Section 5.6). As a result of the incomprehension, such approaches cannot generate programs that fulfill the requirements. An illustrating example is presented in Fig. 5 where the expected output is a sequence of '+' and '-'. However, the generated program outputs a single integer ($ans$ on Line 7).



```
  # Actual input: RYBGRYBGR

1 n = int(input())
2 a = [list(map(int, input().split())) for i in range(n)]
3 for i in range(n):
4     for j in range(n):
5         if a[i][j] == 1 and a[i][j] == 1:
6             print('YES')
6             break
7 print('YES')
```

Fig. 4. *VauleError* exception thrown by generated program.



```
1 n = int(input())
2 a = list(map(int, input().split()))
3 ans = 0
4 for i in range(1, n):
5   if a[i] == a[i]:
6       ans += 1
7 print(ans)
```

(a) Generated Program

```
Actual input:
4
1 2 3 5

Expected output:
+++-

Actual output:
3
```

(b) Failed Test Case

Fig. 5. Sample program and failed test case.

Notably, the requirements in the dataset have explicitly specified the format of programs' input and output, and thus the failure should not be owned to the flexibility in the design of program interfaces. For the given example in Fig. 5, developers could figure out the exact format of the expected output based on the specification: "*Output: In a single line print the sequence of n characters '+' and '-', where the ith character is the sign that is placed in front of number* $a_i$"

Based on the preceding analysis, we conclude that the generated programs have little chance to pass the associated test cases. Consequently, manual interference (especially code revision and validation) is indispensable even if such state-of-the-art automatic code generation approaches are employed.

### 5.4 Q4: Usefulness of Generated Programs

To answer question Q4, we record the time that developers take to finish the tasks, with and without generated programs, respectively. Results are presented as box plots in Fig. 6 (for Group A) and Fig. 7 (for Group B). The blue boxes are associated with cases where developers create source code from scratch. The red ones are associated with cases where generated programs are modified to make them work.

From the box plots, we fail to observe significant difference between the two development models (i.e., coding from scratch or based on generated programs). For Group A, coding from scratch took 652 minutes in total whereas revision based on generated programs took 655.5 minutes. For Group B, coding from scratch took 714.5 minutes whereas revision based on generated programs took 707.4 minutes. Overall, the difference between the two coding models is minor. We also perform a significance test on the resulting data. Results suggest that there is no significant difference between the two coding models: the p-value=0.9696 and F=0.0015 for Group A and p-value=0.9318 and F=0.0074 for Group B. For both groups, the p-value is significantly greater than 0.05. We also compute the effect size (Cohen's d), and results suggest the effect size (-0.0077 for Group A and 0.0173 for Group B) is small.

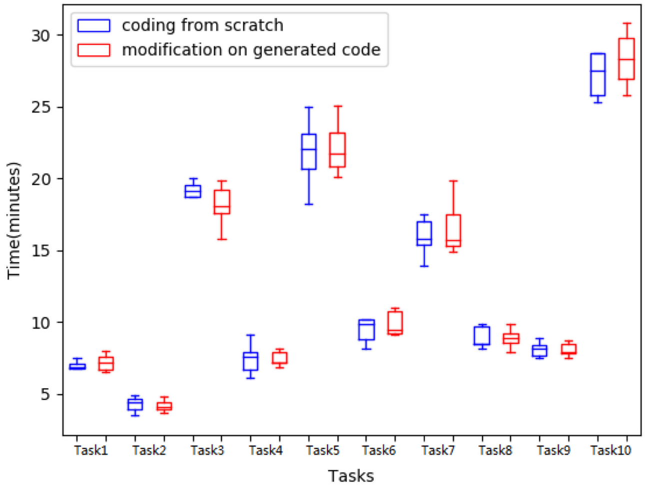


Fig. 6. Usefulness of generated programs (Group A).

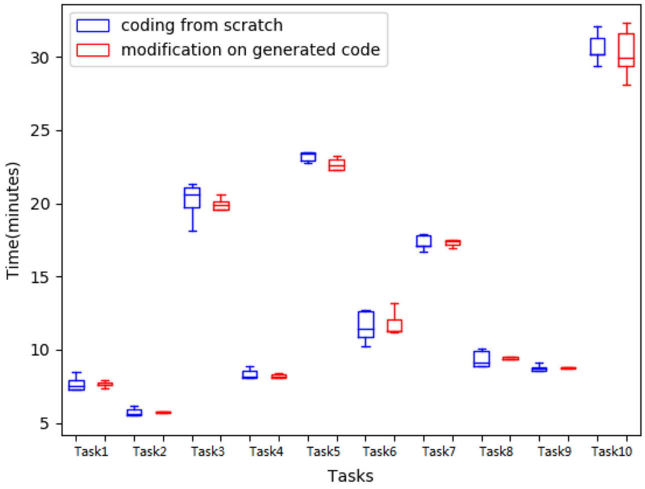


Fig. 7. Usefulness of generated programs (Group B).

We conclude based on the preceding analysis that the generated source code cannot significantly reduce the cost (time) of programming, i.e., modification of the generated programs is not significantly easier than creating programs from scratch.

### 5.5 Q5: Where and Why They Succeed

To answer question Q5, we manually analyze the generated source code. It is highly challenging and time-consuming to manually compare all of the 1,500 (=300 × 5) generated programs against 3,000 (=300 × 10) reference implementations. Consequently, we take the following measures to simplify the manual checking. First, we randomly select 30 (out of 300) software requirements from the testing dataset, and confine the manual checking to this subset. Second, for each generated program on this subset, we only compare it against one of its reference implementations that has the greatest similarity (BLEU) with it. We employ *diff* to visualize the difference (and common ground as well) between a generated program and its reference implementation. A typical example is presented in Fig. 2.

Based on the manual checking, we observe that the evaluated approaches often succeed or partially succeed in generating *input*, *output*, and *for* statements. As suggested by Fig. 2, *SNM* generates the input statement correctly (‘*n,m = map(int, input().split())*’), and places it in the right place, i.e., the very beginning of the program. It also succeeds in generating *output* statement ‘*print(val)*’ and *for* statement (Line 3 on the left part of Fig. 2) except for the variable names. Table 10 presents how often *input*, *output*, and *for* statements are generated successfully. The first column of Table 10 presents different approaches. The second column presents how often the evaluated approaches succeed or partially succeed (inside parentheses) in generating *input* statements. If the generated *input* statement is identical to that in the reference implementation, we say the generation is correct. Otherwise, we manually assess whether the generation is partially correct (with slight difference) or incorrect. The third and the fourth columns present how often the evaluated approaches succeed or partially succeed in generating *for* and *output* statements, respectively. From this table, we observe that all of the evaluated approaches are good at generating such statements. On average, around one fifth of the *input* and *for* statements are generated correctly, and more than half of them are generated partially successfully. Although *output* statements are more difficult to generate correctly (because of variables involved in the statements), in most cases (84 percent on average) the evaluated approaches know that an output statement (i.e., ‘print(*)’) should be generated and placed at the end of the generated programs.

One of the possible reasons for the success in generating *input*, *output*, and *for* statements is that such statements are highly popular in the training data. The popularity of related statements is presented in Table 11. The first column presents the popular statement (or part of a statement). The second column presents their popularity in training programs, i.e., how many percentages of the programs in the training dataset contain such statements. The third column presents their popularity in testing programs. Columns 4-8

TABLE 10
Well Generated Statements

| Approaches | *Input* Statement correct (partially correct) | *For* Statement correct (partially correct) | *Output* Statement correct (partially correct) |
|---|---|---|---|
| Seq2Seq | 17% (83%) | 24% (60%) | 3% (90%) |
| SNM | 17% (83%) | 26% (67%) | 7% (83%) |
| Tree2Tree | 13% (87%) | 11% (54%) | 7% (83%) |
| TRANX | 30% (70%) | 22% (59%) | 3% (83%) |
| Coarse-to-Fine | 27% (70%) | 26% (70%) | 7% (83%) |
| Average | 21% (79%) | 22% (62%) | 5% (84%) |

TABLE 11
Popularity of Well Generated Statements

| Statements | In Training Programs | In Testing Programs | In Generated Programs | | | | |
|---|---|---|---|---|---|---|---|
| | | | Seq2Seq | SNM | Tree2Tree | Tranx | Coarse-to-Fine |
| `input()` | 97% | 97% | 99% | 100% | 99% | 100% | 98% |
| `print(*)` | 99% | 99% | 86% | 86% | 95% | 89% | 91% |
| `for * in *` | 76% | 75% | 84% | 83% | 67% | 89% | 92% |
| `for i in range` | 50% | 47% | 77% | 79% | 57% | 85% | 85% |

TABLE 12
Change of BLEU When Normal Input Is Replaced with Random Noise

| | Seq2Seq | SNM | Tree2Tree | TRANX | Coarse-to-Fine | Average |
|---|---|---|---|---|---|---|
| Normal Input | 0.138 | 0.188 | 0.150 | 0.184 | 0.176 | 0.167 |
| Random Noise | 0.152 | 0.173 | 0.147 | 0.178 | 0.169 | 0.164 |

TABLE 13
Popularity of Well Generated Statements (Random Noise)

| Statements | Seq2Seq | SNM | Tree2Tree | Tranx | Coarse-to-Fine |
|---|---|---|---|---|---|
| `input()` | 99% | 100% | 99% | 99% | 99% |
| `print(*)` | 97% | 90% | 95% | 90% | 97% |
| `for * in *` | 82% | 80% | 76% | 85% | 96% |
| `for i in range` | 78% | 76% | 64% | 82% | 88% |

present their popularity in programs generated by different approaches. From the table, we observe that the *output* statement "`print(*)`" (where * is a wildcard character) appears in almost all of the training and testing programs, and thus the deep learning-based approaches learn to generate this statement frequently. For example, *SNM* and *TRANX* include this statement in each of their generated programs. The same is true for *input* statement "`input()`" and *for* statement "`for * in *`"

Based on the preceding analysis, we conclude that the state-of-the-art approaches have the ability to generate highly popular statements, like *input*, *output*, and *for* statements.

### 5.6 Q6: Little Learned From Requirements

To investigate to what extent the evaluated approaches understand software requirements (input of the approaches), we replace the requirements in the testing data with random noise, and repeat the evaluation. The random noise is created as follows. First, we collect all unique tokens from requirements in the training data, noted as $S_{token}$. Second, for each requirement $r_i$ in the testing data, we generate an empty noise $noise(r_i)$. Third, we randomly select a token from $S_{token}$, and append it to $noise(r_i)$. This step is repeated until $noise(r_i)$ and $r_i$ are equally sized.

Evaluation results are presented in Table 12 where the second and third rows present the BLEU of the evaluated approaches with normal input and noise input, respectively. From this table, we observe that replacing normal input with random noise results in small changes in BLEU of the evaluated approaches. The average BLEU (0.164) with random noise is comparable to that (0.167) with normal input. We also notice that the random noise even increases the performance of *Seq2Seq*, improving its BLEU from 0.138 to 0.152.

We also investigate how often the most popular statements (e.g., *input*, *print*, and *for* statements) are generated by the evaluated approaches when normal input is replaced with random noise. Results are presented in Table 13. From this table, we observe that such popular statements are generated frequently as well. By comparing Table 13 against Table 11, we conclude that replacing requirements text with random noise does not prevent the evaluated approaches from generating the most popular statements.

Based on the preceding analysis, we conclude that the evaluated approaches learn little from input requirements.

### 5.7 Q7: Simple Alternative Approach

As suggested by the preceding analysis in Section 5.5, the evaluated approaches work well in generating popular statements. Consequently, an intuitive and simple way to simulate the evaluated approaches is to generate popular statements only. We call it *popularity-based approach*.

The approach works as follows. First, it computes the average length of the programs in training data. In our case, the average length is 13 lines of source code, noted as $n = 13$. Second, for each unique line of source code in the training data, the approach computes its popularity, i.e., how often it appears in the training programs. Third, it sorts the unique lines according to their popularity, and inserts the top $n$ lines into a new program $p$. Finally, the approach always returns this program ($p$) as the *generated program* regardless of the input (requirements). Notably, this approach completely ignores the input (requirements), and thus it is of little value in practice. However, it may intuitively reveal the state of the art by comparing it against the state-of-the-art approaches.

We apply this simple popularity-based approach to our dataset. Evaluation results suggest it achieves a BLEU of

TABLE 14
Evaluation Results on Nonredundant Dataset

| Approaches | BLEU on New Dataset | NIST | WER | Subtree | Syntactically Correct Programs | Executable Programs | Functionally Correct Programs |
|---|---|---|---|---|---|---|---|
| Seq2Seq | 0.108 | 0.667 | 5.886 | 0.060 | 19.7% | 4.3% | 0% |
| SNM | 0.151 | 0.738 | 0.873 | 0.133 | 27.3% | 10.0% | 0% |
| Tree2Tree | 0.129 | 0.642 | 0.968 | 0.133 | 59.0% | 0.3% | 0% |
| TRANX | 0.149 | 0.836 | 1.292 | 0.113 | 41.0% | 9.3% | 0% |
| CoasetoFine | 0.155 | 1.006 | 1.602 | 0.029 | 3.0% | 1.0% | 0% |
| Average | 0.138 | 0.778 | 2.124 | 0.093 | 30.0% | 5.0% | 0% |

TABLE 15
Effect of Unifying Identifiers

| Applications | Unifying Identifiers | | | | Without Unifying Identifiers (Default Setting) | | | |
|---|---|---|---|---|---|---|---|---|
| | BLEU | Syntactically | Executable | Functionally | BLEU | Syntactically | Executable | Functionally |
| Seq2Seq | 0.135 | 47.3% | 7.7% | 0% | 0.138 | 44.7% | 6.0% | 0% |
| SNM | 0.181 | 80.0% | 16.3% | 0% | 0.188 | 93.0% | 16.7% | 0% |
| Tree2Tree | 0.177 | 72.3% | 12.0% | 0% | 0.150 | 83.7% | 14.3% | 0% |
| TRANX | 0.187 | 84.3% | 4.0% | 0% | 0.184 | 81.7% | 9.0% | 0% |
| Coarse-to-Fine | 0.151 | 3.3% | 0.1% | 0% | 0.176 | 10.0% | 2.7% | 0% |
| **Average** | **0.166** | **57.4%** | **8.0%** | **0%** | **0.167** | **62.6%** | **9.7%** | **0%** |

0.211, significantly higher than any of the evaluated deep learning-based approaches (as shown in Table 5). The comparison intuitively reveals the state of the art: the advanced deep learning-based code generation approaches cannot even outperform this intuitive and impractical approach.

Based on the preceding analysis, we conclude that it is likely for simple and intuitive approaches to outperform the state-of-the-art deep learning-based approaches concerning the common performance metrics BLEU.

### 5.8 Q8: Removing Redundant Implementations Does Not Help

Evaluation results on the nonredundant dataset are presented in Table 14. By comparing this table against Table 5 (performance on the original dataset where multiple implementations for the same tasks are exploited), we make the following observations:

- First, removing redundant implementations does not help. For example, all of the evaluated approaches result in lower BLEU on the nonredundant dataset than that on the original dataset. It reduces from 0.138 to 0.108 (Seq2Seq), from 0.188 to 0.151 (SNM), from 0.15 to 0.129 (Tree2Tree), from 0.176 to 0.149 (TRANX), and from 0.167 to 0.155 (CoasetoFine). The same is true for other performance metrics.
- Second, no functionally correct programs could be generated even if the evaluated approaches are fed with the nonredundant dataset.

Based on the preceding analysis, we conclude that removing redundant implementations from the dataset may not improve the performance of code generation.

### 5.9 Q9: Impact of Unifying Identifiers

To investigate the impact of identifier unification, we unify identifiers in requirements and source code (in the same way as TRANX unifies identifiers [31]), and repeat the first empirical study as introduced in Section 4.3. First, we replace constant strings (like *"URL is required"*) that appear in both requirements and associated source code with unified tokens "$str_i$". Second, we replace variables that appear in both requirements and associated source code with unified tokens "$var_i$". The variables are not further divided according to their types because Python is not a statically typed programming language (like Java). Notably, the same identifier unification is conducted on all of the data. Evaluation results of the identifier unification are presented in Table 15. To facilitate the comparison, we also present the performance of the default setting (i.e., without unifying identifiers). From Table 15, we make the following observations:

- First, unifying identifiers has minor and diverse impact on the performance of the evaluated approaches. For example, it improves the BLEU of TRANX and Tree2-Tree slightly from 0.184 to 0.187 and from 0.15 to 0.177, respectively. In the same time, however, it also decreases BLEU of Seq2seq (from 0.138 to 0.135), SNM (from 0.188 to 0.181), and Coarse-to-Fine (from 0.176 to 0.151). Overall, unifying identifiers slightly reduces the average BLEU of the evaluated approaches from 0.167 to 0.166. The same is true for other performance metrics, e.g., syntactically correct programs.
- Second, no functionally correct programs could be generated regardless of the application of unifying identifiers.

## 6 Discussions

### 6.1 Potential Reasons for Reduced Performance

Evaluation results in Section 5 suggest that switching from existing datasets to ours significantly reduces the performance of existing approaches. Potential reasons are discussed as follows.

First, some special characters in existing datasets facilitate learning-based code generation. For example, the requirements (pseudo-code) in *Django* are quite similar to their implementations. On average, 49.4 percent of the tokens in a program (source code) could be copied from the requirements associated with the program. As a result, learning-based approaches may achieve high performance by copying tokens from requirements to generated programs. In *HS*, different programs are highly similar to each other, which also significantly facilitates code generation. Because of the similarity, learning-based approaches can learn the common structures (also known as templates), and frequently generate source code successfully by '*filling learned code templates from training data with arguments copied from input*' [22].

Second, the requirements in our dataset are much more complex than the existing ones. As discussed in Section 5.6, a great challenge in code generation is natural language understanding (NLU), i.e., to understand requirements. The longer the requirements are, the harder NLU is. Compared to existing datasets, our dataset is composed of much longer and more complex requirements. The average length of such requirements is 185 tokens compared to 14 and 34 in *Django* and *HS*, respectively.

Third, the diversity of our dataset has a significant negative impact on the evaluated approaches. Such approaches have been trained in a specific domain with similar requirements. However, our dataset has very diverse requirements with no common tasks. As a result, except for the generic programming skills (especially algorithm related programming skills), little could be learned about the implementation of specific tasks. However, learning the generic programming skills (i.e., the ability to turn textual requirements into source code as a human developer does) is highly challenging. As a result, the performance of program generation is significantly reduced.

Fourth, the size of our dataset may have prevented the evaluated approaches from reaching their maximal potential. In total, the dataset is composed of 16,673 requirements-code pairs, making it comparable to other data sets that have been employed by the authors of the evaluated approaches. For example, SNM was originally evaluated on JBOS (with 640 items), GEO (with 880 items), ATIS (with 5,373 items), and IFTTT (with 86,960 items), independently. Our dataset is significantly bigger than such datasets except for IFTTT. However, our dataset contains 2,740 unique requirements only, which makes it smaller than ATIS and IFTTT concerning the number of unique requirements. Besides that, the increased complexity of the requirements and source code, together with the limited number of unique requirements, could prevent the evaluated approaches from reaching their maximal potential.

Fifth, our tuning of the hyper parameters for the evaluated approaches could be less effective than the tuning conducted by the original authors of the evaluated approaches. Such approaches have been fine-tuned on given datasets that were leveraged for evaluation by their authors, which often results in high performance on the given dataset. The original tuning is effective because the experts who tuned the parameters were familiar with the approaches. In contrast, we tuned the parameters without deep understanding of the evaluated approaches, and thus the tuning could be more time-consuming and less effective. This, in turn, prevents the evaluated approaches from reaching their maximal potential.

### 6.2 Experiment on More Datasets

There is a clear need for an empirical study on various datasets with the proposed approach and evaluate them by comparing it with other approaches. The experiment is conducted on a single dataset that we create in Section 3, which may limit its validity. As introduced in Section 2.2, existing datasets have significant limitations, and thus assessing the state of the art on such datasets may result in severe threats to validity. To this end, we create a new dataset. With this dataset, we assess the state of the art in code generation. To reduce threats to external validity, however, we should conduct similar experiments on other qualified datasets in the future when such datasets are available. Notably, we do not compare the proposed approach (popularity-based code generation) against other approaches on existing datasets. For example, each of the reference programs in Django is composed of a single unique statement, which makes it impractical to select the most *popular* statements in the dataset. As a result, the popularity-based approach cannot work on Django.

Other threats to validity exist as well, e.g., the size of the involved dataset and the representativeness of the evaluated approaches. The size of the involved dataset may influence the performance of the evaluated approaches. It is likely that increasing the size of the dataset could improve the performance. However, we have not yet investigated its exact influence. Selecting different code generation approaches for the evaluation may result in different conclusions because their performance on the same dataset (i.e., our new dataset) could vary significantly. To reduce the threats to validity, we select multiple state-of-the-art approaches for the evaluation.

### 6.3 Limited Diversity of the New Dataset

As specified in Section 3, the new dataset is created based on programming contest platforms, which may limit its diversity. Although the programming contest platforms do not post any explicit limitations on the domain of contests, most of the contests concern data structures, sorting algorithms, mathematic computation, text processing, or database management. They are rarely related to any specific application domains, e.g., financial systems, office software, or image processing. As a result, the diversity of the resulting dataset is limited. Approaches trained on such dataset may fail to generate applications whose creation strongly depends on domain knowledge.

Besides the limited diversity, the source code within the dataset could be different from applications in the industry in the following ways. First, most of the code in the new dataset is coded by novice programmers and the skillset levels of these developers are low when compared with industry standards. Second, most of the code written by programmers participating in such contests tend to algorithm driven and end up being implementations of some data structures. Third, real-world systems have a lot of inter-dependencies among the task whereas a majority of tasks in programming contests tend to be orthogonal in nature. Finally, there is lot of importance given to certain qualities in programming contests which is not necessarily true in real-world systems.

One future work to strengthen the dataset is to exploit additional data sources. Extracting additional data will increase both the size and diversity of the resulting dataset, and thus may help to facilitate the training of deep learning-based code generation models. It is also interesting to include non-English software requirements, and to investigate multiple-language code generation.

Although it is not novel to create dataset by crawling web pages, creating and publishing the dataset is valuable. On one side, the resulting dataset has significant advantages compared to existing ones. On the other side, publishing it releases other researchers from grueling and time-consuming dataset creation.

### 6.4 Performance Metrics for the Empirical Study

Besides BLEU, we also employ the number of compilation errors, the number of compilation warnings, and the number of failed/passed test cases to assess the quality of generated programs as presented in Table 5. However, such metrics are not suitable for existing datasets (e.g., *Django* and *HS*) because the reference programs (code fragments) within such datasets are incomplete and incompilable. Consequently, it is unfair/unpractical to require models trained on such datasets to generate complete and compilable/runnable programs. However, programs in our new dataset are complete and syntactically correct, and thus we compute such performance metrics for the evaluation on the new dataset.

### 6.5 Threats to Validity

Besides the threats (limitations) discussed in the preceding sections, the evaluation (especially the case study to evaluate the usefulness of generated programs) is subjected to the following threats to validity. A threat to external validity is that only ten programming tasks and twenty participants were involved in the evaluation. Conclusions drawn on such limited number of subjects may not be generalizable. We failed to increase the number of programming tasks or participants because it is time-consuming for participants to finish the selected programming tasks, and it is challenging for us to recruit a large number of qualified participants. A threat to internal validity is that the observations (coding speed) could be significantly influenced by the characters (e.g., knowledge in Python and programming skills) besides the investigated factor (i.e., with or without the generated programs). To reduce the threat, we recruited thirty participants, excluded the top and bottom ones (concerning their performance) with a pretest, and divided the remaining participants into two independent groups according to their performance in the pretest. As a result, the participants within the same group had similar performance in the pretest.

## 7 Conclusion and Future Work

Deep learning-based code generation is potentially promising, and a few approaches have been proposed. Although existing evaluations suggest that such approaches are highly accurate, they are evaluated on small datasets where ‘requirements’ are quite different from real-world requirements in the industry. To assess the state of the art, in this paper, we build a large scale dataset. Compared to existing ones, it is larger and more diverse. Besides the dataset, we also build an assisting tool to measure the quality of generated programs. We not only compute the widely used plain text-based metrics (BLEU), but also employ syntax sensitive static checking as well as test based dynamic cross-validation. Based on the resulting dataset and assisting tool, we reassess the state of the art in natural language-based program generation. Evaluation results suggest that the state-of-the-art approaches successfully learn to generate popular statements. However, the generated programs are often significantly different from their references. Besides that, they often contain syntactic and semantical errors, and none of them can pass even a single test case. Further analysis suggests that they learn little from the input (requirements). Consequently, to further improve the state of the art, researchers should pay more attention to the encoders of the neural networks that are in charge of requirements’ interpretation. The resulting dataset, the assisting tool, and evaluated approaches (all of them are publicly available at https://github.com/ds4an/CoDas4CG) could serve as a basis for future research in this direction.

One future work is to design more effective metrics to assess quality of code generation. It is well-known that BLEU alone is insufficient for assessing the quality of code generation [69] because source code has little tolerance for poor syntax or semantics. To this end, in this paper we propose additional metrics to assess the syntax and semantics of generated programs, i.e., the number of compilation errors, number of compilation warnings, and number of failed/passed test cases. However, as suggested by the empirical study in Section 5, most of the programs generated by the state-of-the-art approaches are not executable, which significantly prevents the proposed execution-based metrics from reaching their maximal potential. Consequently, it remains an open question to design effective metrics in the future to accurately and quantitatively assess the quality of programs automatically generated by the state-of-the-art approaches.

In the future, it is worthwhile to explore larger (not necessarily more complex) datasets to investigate whether the performance of deep learning-based program generation could be improved if there are more sample program implementations available for each task.

It is interesting to change the evaluation setting and repeat the evaluation in the future. The evaluation setting is that some task requirement-implementation pairs are used to train the deep learning models, and use other different task requirements to test if the resulting models can generate useful code. Such a setting is quite realistic but highly challenging. However, if we build a smaller dataset containing similar tasks only, repeating the same evaluation could result in significantly improved performance of the evaluated approaches because in this case the testing tasks are similar to those leveraged for model training.

Finally, further investigation into the weakness of the evaluated approaches could be valuable. In the evaluation, we analyze where and why the evaluated approaches work. However, we have no yet investigated where and why such approaches fail.

## ACKNOWLEDGMENTS

The authors would like to thank the associate editor and the anonymous reviewers for their insightful comments and constructive suggestions. This work was sponsored in part by the National Key Research and Development Program of China (2017YFB1001803), the National Natural Science Foundation of China (61772071, 61690205), and the National Science Foundation (CCF-1350487).

## REFERENCES

[1] I. Sommerville, *Software Engineering*. Boston, MA, USA: Addison-Wesley, 1992.

[2] M. W. Whalen, "An approach to automatic code generation for safety-critical systems," in *Proc. 14th IEEE Int. Conf. Autom. Softw. Eng.*, 1999, pp. 315–318.

[3] M. W. Whalen, "High-integrity code generation for state-based formalisms," in *Proc. Int. Conf. Softw. Eng.*, 2000, pp. 725–727.

[4] D. Harel *et al.*, "STATEMATE: A working environment for the development of complex reactive systems," *IEEE Trans. Softw. Eng.*, vol. 16, no. 4, pp. 403–414, Apr. 1990.

[5] H. Mei and L. Zhang, "Can big data bring a breakthrough for software automation?" *Sci. China Inf. Sci.*, vol. 61, no. 5, 2018, Art. no. 056101.

[6] G. O'Regan, *Concise Guide to Formal Methods: Theory, Fundamentals and Industry Applications*. Berlin, Germany: Springer, 2017.

[7] J. M. Wing, "A specifier's introduction to formal methods," *Computer*, vol. 23, no. 9, pp. 8–22, Sep. 1990.

[8] P. Linz, *An Introduction to Formal Languages and Automata*. Burlington, MA, USA: Jones and Bartlett Learning, 2011.

[9] R. Soley and the OMG Staff Strategy Group, "Model driven architecture," Object Management Group, Needham, MA, Rep. no. omg/00-11-05, Nov. 2000.

[10] F. A. Kraemer, "Engineering Android applications based on UML activities," in *Proc. 14th Int. Conf. Model Driven Eng. Lang. Syst.*, 2011, pp. 183–197.

[11] G. Booch, J. Rumbaugh, and I. Jacobson, *The Unified Modelling Language User Guide*. Boston, MA, USA: Addison-Wesley Professional, 2005.

[12] G. Sunyé, A. L. Guennec, and J.-M. Jézéquel, "Using UML action semantics for model execution and transformation," *Inf. Syst.*, vol. 27, no. 6, pp. 445–457, 2002. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0306437902000145

[13] D. A. Dahl *et al.*, "Expanding the scope of the ATIS task: The ATIS-3 corpus," in *Proc. Workshop held at Plainsboro Hum. Lang. Technol.*, 1994, pp. 43–48. [Online]. Available: http://aclweb.org/anthology/H/H94/H94-1010.pdf

[14] W. Ling *et al.*, "Latent predictor networks for code generation," in *Proc. 54th Annu. Meeting Assoc. Comput. Linguistics*, 2016, pp. 599–609. [Online]. Available: http://aclweb.org/anthology/P/P16/P16-1057.pdf

[15] L. R. Tang and R. J. Mooney, "Using multiple clause constructors in inductive logic programming for semantic parsing," in *Proc. 12th Eur. Conf. Mach. Learn.*, 2001, pp. 466–477. [Online]. Available: https://doi.org/10.1007/3-540-44795-4_40

[16] P. Yin, B. Deng, E. Chen, B. Vasilescu, and G. Neubig, "Learning to mine aligned code and natural language pairs from stack overflow," in *Proc. 15th Int. Conf. Mining Softw. Repositories*, 2018, pp. 476–486. [Online]. Available: http://doi.acm.org/10.1145/3196398.3196408

[17] Y. Deng, A. Kanervisto, J. Ling, and A. M. Rush, "Image-to-markup generation with coarse-to-fine attention," in *Proc. 34th Int. Conf. Mach. Learn.*, 2017, pp. 980–989. [Online]. Available: http://proceedings.mlr.press/v70/deng17a.html

[18] C. Chen, T. Su, G. Meng, Z. Xing, and Y. Liu, "From UI design image to GUI skeleton: A neural machine translator to bootstrap mobile GUI implementation," in *Proc. 40th Int. Conf. Softw. Eng.*, 2018, pp. 665–676. [Online]. Available: http://doi.acm.org/10.1145/3180155.3180240

[19] S. Gulwani, "Automating string processing in spreadsheets using input-output examples," in *Proc. 38th ACM SIGPLAN-SIGACT Symp. Princ. Program. Lang.*, 2011, pp. 317–330. [Online]. Available: http://doi.acm.org/10.1145/1926385.1926423

[20] C. Shu and H. Zhang, "Neural programming by example," in *Proc. 31st AAAI Conf. Artif. Intell.*, 2017, pp. 1539–1545.

[21] J. Dick, E. Hull, and K. Jackson, *Requirements Engineering*. 4th ed., Berlin, Germany: Springer, Aug. 23, 2017.

[22] P. Yin and G. Neubig, "A syntactic neural model for general-purpose code generation," in *Proc. 55th Annu. Meeting Assoc. Comput. Linguistics*, 2017, pp. 440–450. [Online]. Available: https://doi.org/10.18653/v1/P17-1041

[23] K. Papineni, S. Roukos, T. Ward, and W. Zhu, "BLEU: A method for automatic evaluation of machine translation," in *Proc. 40th Annu. Meeting Assoc. Comput. Linguistics*, 2002, pp. 311–318. [Online]. Available: http://www.aclweb.org/anthology/P02-1040.pdf

[24] Y. Wu *et al.*, "Google's neural machine translation system: Bridging the gap between human and machine translation," *CoRR*, 2016. [Online]. Available: http://arxiv.org/abs/1609.08144

[25] Y. Oda *et al.*, "Learning to generate pseudo-code from source code using statistical machine translation (T)," in *Proc. 30th IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2015, pp. 574–584. [Online]. Available: https://doi.org/10.1109/ASE.2015.36

[26] P. Liang, M. I. Jordan, and D. Klein, "Learning dependency-based compositional semantics," in *Proc. 49th Annu. Meeting Assoc. Comput. Linguistics*, 2011, pp. 590–599. [Online]. Available: http://www.aclweb.org/anthology/P11-1060

[27] A. Wang, T. Kwiatkowski, and L. S. Zettlemoyer, "Morpho-syntactic lexical generalization for CCG semantic parsing," in *Proc. Conf. Empir. Methods Natural Lang. Process.*, 2014, pp. 1284–1295. [Online]. Available: http://aclweb.org/anthology/D/D14/D14-1135.pdf

[28] L. Dong and M. Lapata, "Language to logical form with neural attention," in *Proc. 54th Annu. Meeting Assoc. Comput. Linguistics*, 2016, pp. 33–43. [Online]. Available: http://aclweb.org/anthology/P/P16/P16-1004.pdf

[29] M. Rabinovich, M. Stern, and D. Klein, "Abstract syntax networks for code generation and semantic parsing," in *Proc. 55th Annu. Meeting Assoc. Comput. Linguistics*, 2017, pp. 1139–1149. [Online]. Available: https://doi.org/10.18653/v1/P17-1105

[30] A. Stehnii, "Generation of code from text description with syntactic parsing and Tree2Tree model," Master's thesis, Dept. Comput. Sci., Ukrainian Catholic University, Lviv, Ukraine, 2018.

[31] P. Yin and G. Neubig, "TRANX: A transition-based neural abstract syntax parser for semantic parsing and code generation," in *Proc. Conf. Empir. Methods Natural Lang. Process.*, 2018, pp. 7–12. [Online]. Available: https://arxiv.org/abs/1810.02720

[32] L. Dong and M. Lapata, "Coarse-to-fine decoding for neural semantic parsing," in *Proc. 56th Annu. Meeting Assoc. Comput. Linguistics*, 2018, pp. 731–742. [Online]. Available: http://aclweb.org/anthology/P18-1068

[33] S. A. Hayati, R. Olivier, P. Avvaru, P. Yin, A. Tomasic, and G. Neubig, "Retrieval-based neural code generation," in *Proc. Conf. Empir. Methods Natural Lang. Process.*, 2018, pp. 925–930. [Online]. Available: https://www.aclweb.org/anthology/D18-1111/

[34] Z. Sun, Q. Zhu, L. Mou, Y. Xiong, G. Li, and L. Zhang, "A grammar-based structural CNN decoder for code generation," in *Proc. AAAI Conf. Artif. Intell.*, 2019, vol. 33, pp. 7055–7062.

[35] T. Gvero and V. Kuncak, "Synthesizing Java expressions from free-form queries," in *Proc. ACM SIGPLAN Int. Conf. Object-Oriented Program. Syst. Lang. Appl.*, 2015, pp. 416–432. [Online]. Available: http://doi.acm.org/10.1145/2814270.2814295

[36] M. Raghothaman, Y. Wei, and Y. Hamadi, "SWIM: Synthesizing what i mean - code search and idiomatic snippet synthesis," in *Proc. IEEE/ACM 38th Int. Conf. Softw. Eng.*, 2016, pp. 357–367.

[37] A. T. Nguyen, P. C. Rigby, T. Nguyen, D. Palani, M. Karanfil, and T. N. Nguyen, "Statistical translation of english texts to API code templates," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol.*, 2018, pp. 194–205.

[38] S. Yan, H. Yu, Y. Chen, B. Shen, and L. Jiang, "Are the code snippets what we are searching for? A benchmark and an empirical study on code search with natural-language queries," in *Proc. IEEE 27th Int. Conf. Softw. Anal. Evol. Reengineering*, 2020, pp. 344–354.

[39] J. M. Zelle and R. J. Mooney, "Learning to parse database queries using inductive logic programming," in *Proc. 13th Nat. Conf. Artif. Intell.*, 1996, pp. 1050–1055. [Online]. Available: http://www.aaai.org/Library/AAAI/1996/aaai96-156.php

[40] C. Quirk, R. J. Mooney, and M. Galley, "Language to code: Learning semantic parsers for if-this-then-that recipes," in *Proc. 53rd Annu. Meeting Assoc. Comput. Linguistics*, 2015, pp. 878–888. [Online]. Available: http://aclweb.org/anthology/P/P15/P15-1085.pdf

[41] Magic the gathering, 2016. [Online]. Available: http://github.com/magefree/mage/

[42] Hearthstone, 2016. [Online]. Available: http://github.com/danielyule/hearthbreaker/

[43] Z. Yao, D. S. Weld, W. Chen, and H. Sun, “StaQC: A systematically mined question-code dataset from stack overflow,” in *Proc. World Wide Web Conf. World Wide Web*, 2018, pp. 1693–1703. [Online]. Available: http://doi.acm.org/10.1145/3178876.3186081

[44] Stack OverFlow, 2019. [Online]. Available: https://stackoverflow.com/

[45] V. Raychev, M. Vechev, and E. Yahav, “Code completion with statistical language models,” in *Proc. 35th ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2014, pp. 419–428. [Online]. Available: http://doi.acm.org/10.1145/2594291.2594321

[46] J. Li, Y. Wang, M. R. Lyu, and I. King, “Code completion with neural attention and pointer networks,” in *Proc. 27th Int. Joint Conf. Artif. Intell.*, 2018, pp. 4159–4165. [Online]. Available: https://doi.org/10.24963/ijcai.2018/578

[47] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu, “On the naturalness of software,” in *Proc. 34th Int. Conf. Softw. Eng.*, 2012, pp. 837–847. [Online]. Available: http://dl.acm.org/citation.cfm?id=2337223.2337322

[48] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton, “Suggesting accurate method and class names,” in *Proc. 10th Joint Meeting Found. Softw. Eng.*, 2015, pp. 38–49. [Online]. Available: http://doi.acm.org/10.1145/2786805.2786849

[49] L. Jiang, H. Liu, and H. Jiang, “Machine learning based automated method name recommendation: How far are we,” in *Proc. 34th IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2019, pp. 602–614.

[50] H. Liu, Q. Liu, C.-A. Staicu, M. Pradel, and Y. Luo, “Nomen est Omen: Exploring and exploiting similarities between argument and parameter names,” in *Proc. 38th Int. Conf. Softw. Eng.*, 2016, pp. 1063–1073. [Online]. Available: http://doi.acm.org/10.1145/2884781.2884841

[51] X. Chen, C. Liu, and D. Song, “Towards synthesizing complex programs from input-output examples,” in *Proc. Int. Conf. Learn. Representations*, 2018. [Online]. Available: https://openreview.net/forum?id=Skp1ESxRZ

[52] R. Alur, R. Singh, D. Fisman, and A. Solar-Lezama, “Search-based program synthesis,” *Commun. ACM*, vol. 61, no. 12, pp. 84–93, Nov. 2018. [Online]. Available: https://doi.org/10.1145/3208071

[53] J. Devlin, J. Uesato, S. Bhupatiraju, R. Singh, A. Mohamed, and P. Kohli, “RobustFill: Neural program learning under noisy I/O,” in *Proc. 34th Int. Conf. Mach. Learn.*, 2017, pp. 990–998.

[54] E. Parisotto, A. Mohamed, R. Singh, L. Li, D. Zhou, and P. Kohli, “Neuro-symbolic program synthesis,” in *Proc. 5th Int. Conf. Learn. Representations*, 2017. [Online]. Available: https://www.microsoft.com/en-us/research/publication/neuro-symbolic-pro gram-synthesis-2/

[55] Y. Feng, R. Martins, J. Van Geffen, I. Dillig, and S. Chaudhuri, “Component-based synthesis of table consolidation and transformation tasks from examples,” in *Proc. 38th ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2017, pp. 422–436. [Online]. Available: https://doi.org/10.1145/3062341.3062351

[56] Y. Feng, R. Martins, O. Bastani, and I. Dillig, “Program synthesis using conflict-driven learning,” in *Proc. 39th ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2018, pp. 420–435. [Online]. Available: https://doi.org/10.1145/3192366.3192382

[57] W. Lee, K. Heo, R. Alur, and M. Naik, “Accelerating search-based program synthesis using learned probabilistic models,” in *Proc. 39th ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2018, pp. 436–449. [Online]. Available: https://doi.org/10.1145/3192366.3192410

[58] R. Shin *et al.*, “Synthetic datasets for neural program synthesis,” in *Proc. Int. Conf. Learn. Representations*, 2019. [Online]. Available: https://openreview.net/forum?id=ryeOSnAqYm

[59] M. Balog, A. Gaunt, M. Brockschmidt, S. Nowozin, and D. Tarlow, “DeepCoder: Learning to write programs,” in *Proc. Int. Conf. Learn. Representations*, 2017, pp. 1–20.

[60] J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang, and X. Liu, “A novel neural source code representation based on abstract syntax tree,” in *Proc. IEEE/ACM 41st Int. Conf. Softw. Eng.*, 2019, pp. 783–794.

[61] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, “Code2vec: Learning distributed representations of code,” in *Proc. 46th ACM SIGPLAN Symp. Princ. Program. Lang.*, 2019, pp. 1–29. [Online]. Available: http://doi.acm.org/10.1145/3290353

[62] L. Mou, G. Li, Z. Jin, L. Zhang, and T. Wang, “TBCNN: A tree-based convolutional neural network for programming language processing,” in *Proc. 30th AAAI Conf. Artif. Intell.*, 2016, pp. 1287–1293.

[63] M. Allamanis, M. Brockschmidt, and M. Khademi, “Learning to represent programs with graphs,” *CoRR*, 2017. [Online]. Available: http://arxiv.org/abs/1711.00740

[64] Codeforces, 2019. [Online]. Available: http://codeforces.com/

[65] HackerEarth, 2019. [Online]. Available: https://www.hackerearth.com/

[66] G. S. Manku, A. Jain, and A. D. Sarma, “Detecting near-duplicates for web crawling,” in *Proc. 16th Int. Conf. World Wide Web*, 2007, pp. 141–150. [Online]. Available: http://doi.acm.org/10.1145/1242572.1242592

[67] V. I. Levenshtein, “Binary codes capable of correcting deletions, insertions, and reversals,” *Soviet Phys. Doklady*, vol. 10, no. 8, pp. 707–710, 1966.

[68] D. Liu and D. Gildea, “Syntactic features for evaluation of machine translation,” in *Proc. Workshop Intrinsic Extrinsic Eval. Measures Mach. Transl. Summarization*, 2005, pp. 25–32.

[69] S. Karaivanov, V. Raychev, and M. Vechev, “Phrase-based statistical translation of programming languages,” in *Proc. ACM Int. Symp. New Ideas New Paradigms Reflections Program. Softw.*, 2014, pp. 173–184. [Online]. Available: http://doi.acm.org/10.1145/2661136.2661148

[70] S. Bird, E. Klein, and E. Loper, *Natural Language Processing with Python*. Sebastopol, CA, USA: O’Reilly, 2009. [Online]. Available: http://www.oreilly.de/catalog/9780596516499/index.html

[71] Natural language toolkit, 2020. [Online]. Available: https://github.com/nltk/nltk/

[72] autopep8, 2018. [Online]. Available: https://pypi.org/project/autopep8/

[73] P. M. Lerman, “Fitting segmented regression models by grid search,” *Appl. Statist.*, vol. 29, no. 1, pp. 77–84, 1980.

[74] Pylint, 2018. [Online]. Available: https://www.pylint.org/

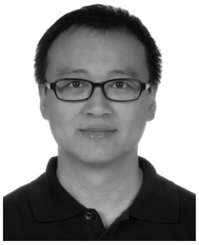
[75] Python documents, 2019. [Online]. Available: https://docs.python.org/3/library/exceptions.html

**Hui Liu** received the BS degree in control science from Shandong University, China, in 2001, the MS degree in computer science from Shanghai University, China, in 2004, and the PhD degree in computer science from the Peking University, China, in 2008. He is a professor with the School of Computer Science and Technology, Beijing Institute of Technology, China. He was a visiting research fellow in centre for research on evolution, search and testing (CREST) at University College London, United Kingdom. He served on the program committees and organizing committees of prestigious conferences, such as ICSME, RE, ICSR, and COMPSAC. He is serving as associate editor for the *IET Software*, and guest editor for the *Empirical Software Engineering* and the *Journal of Systems and Software*. He is particularly interested in deep learning-based software engineering, software refactoring, and software quality. He is also interested in developing practical tools to assist software engineers.

**Minzhu Shen** received the BS degree from the information management and system program, Northwest A&F University, China, in 2018. She is currently working toward the master’s degree in the School of Computer Science and Technology, Beijing Institute of Technology, China, under the supervision of Dr. Hui Liu. Her current research interests include software testing and AI-based software engineering.

**Jiaqi Zhu** received the BS degree from the College of Information Engineering, Northwest A&F University, China, in 2017. She is currently working toward the master’s degree in the School of Computer Science and Technology, Beijing Institute of Technology, China, under the supervision of Dr. Hui Liu. Her current research interests include code generation and software evolution.

**Nan Niu** received the BEng degree in computer science and engineering from the Beijing Institute of Technology, Beijing, China, in 1999, the MSc degree in computing science from the University of Alberta, Edmonton, AB, Canada, in 2004, and the PhD degree in computer science from the University of Toronto, Toronto, ON, Canada, in 2009. He is currently an associate professor with the Department of Electrical Engineering and Computer Science, University of Cincinnati, Cincinnati, Ohio. His current research interests include software requirements engineering, information seeking in software engineering, and human-centric computing. He was a recipient of the U.S. National Science Foundation Faculty Early Career Development (CAREER) Award, the IEEE International Requirements Engineering Conference's Best Research Paper Award, in 2016, and the Most Influential Paper Award, in 2018.



**Ge Li** received the PhD degree from Peking University, China, in 2006, and had been a visiting associate professor at Stanford University, Stanford, California, in 2013-2014. He is an associate professor with the Department of Computer Science and Technology, School of EECS. He is currently the deputy secretary general of CCF Software Engineering Society and the founder of the Software Program Generation Study Group. He was one of the earliest researchers engaged in the study of the computer program language model based on deep neural network, and the study of end-to-end program code generating techniques. His current research mainly concerns applications of probabilistic methods for machine learning, including program language process, natural language process, and software engineering.

**Lu Zhang** received the both BSc and PhD degrees in computer science from Peking University, China, in 1995 and 2000 respectively. He is a professor with the School of Electronics Engineering and Computer Science, Peking University, P.R. China. He was a postdoctoral researcher in Oxford Brookes University and University of Liverpool, United Kingdom. He served on the program committees of many prestigious conferences, such as FSE, OOPSLA, ISSTA, and ASE. He was a program co-chair of SCAM2008 and a program co-chair of ICSM17. He has been on the editorial boards of the *Journal of Software Maintenance and Evolution: Research and Practice* and the *Software Testing, Verification and Reliability*. His current research interests include software testing and analysis, program comprehension, software maintenance and evolution, software reuse, and program synthesis.