A Survey on Natural Language Processing for Programming

Qingfu Zhu[♯], Xianzhen Luo[♯], Fang Liu[♭], Cuiyun Gao[♮], Wanxiang Che[♯*]

[♯]Harbin Institute of Technology, Harbin, China
[♭]Beihang University, Beijing, China
[♮]Harbin Institute of Technology, Shenzhen, China
{qfzhu,xzluo,car}@ir.hit.edu.cn
fangliu@buaa.edu.cn
gaocuiyun@hit.edu.cn

Abstract

Natural language processing for programming aims to use NLP techniques to assist programming. It is increasingly prevalent for its effectiveness in improving productivity. Distinct from natural language, a programming language is highly structured and functional. Constructing a structure-based representation and a functionality-oriented algorithm is at the heart of program understanding and generation. In this paper, we conduct a systematic review covering tasks, datasets, evaluation methods, techniques, and models from the perspective of the structure-based and functionality-oriented property, aiming to understand the role of the two properties in each component. Based on the analysis, we illustrate unexplored areas and suggest potential directions for future work.

Keywords: Natural language processing for programming, Program Generation, Program Understanding

1. Introduction

Natural language processing for programming (NLP4P) is an interdisciplinary field of NLP and software engineering (SE), aiming to use NLP techniques for assisting programming (Lachmy et al., 2021). It could relieve developers from laborious work, e.g., by automatically writing documentation for a program. Meanwhile, it provides easy access for non-professional users to improve efficiency, e.g., by performing a cross-application operation with natural language (NL) interface (Liu et al., 2016a). Therefore, it is beneficial for improving the productivity of the whole society.

Distinct from NL, a programming language (PL) is characterized by two properties: **structure-based** and **functionality-oriented**, as shown in Figure 1. First, PL is highly structure-based since it typically contains multiple sophisticated structures, such as hierarchy, loops, and recursions. Appropriately modeling the components and obtaining a structure-based representation is the key to program understanding (Mou et al., 2016; Allamanis et al., 2018; Hu et al., 2018; Guo et al., 2020; Wang et al., 2021; Guo et al., 2022). Second, PL is functionality-oriented since it is executable and ought to convert given input into expected output. Developing an algorithm oriented to the functionality is at the heart of generating a logically correct program (Chen et al., 2021; Hendrycks et al., 2021; Li et al., 2022; Nijkamp et al., 2022; Le et al., 2022). Despite the benefits, the two properties cannot be directly modeled by conventional NLP approaches due to the heterogeneity between NL and PL, making the integration of the properties a fundamental topic in NLP4P.

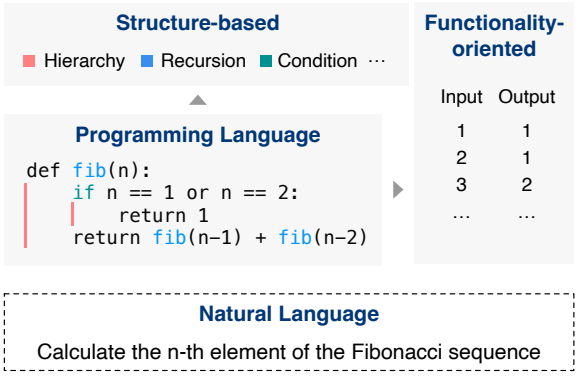

Figure 1: An example of the structure-based and functionality-oriented property of programming language. Colored fonts and indents denote different aspects of the structure-based property. Functionality-oriented refers to the property that PL should convert given input into expected output.

From the perspective of pre-training, Niu et al. (2022), Zan et al. (2023) have summarized the recent advances. Nevertheless, the role of the structure-based and functionality-oriented property has not been sufficiently discussed. In this paper, we focus on the properties and systematically review their effect in defining tasks (§2), constructing datasets (§3), forming evaluation methods (§4), supporting techniques (§5), and achieving SOTA performance (§6). Based on the analysis, we illustrate unexplored areas of current NLP4P and

*Corresponding author.

LREC-COLING 2024, pages 1690–1704

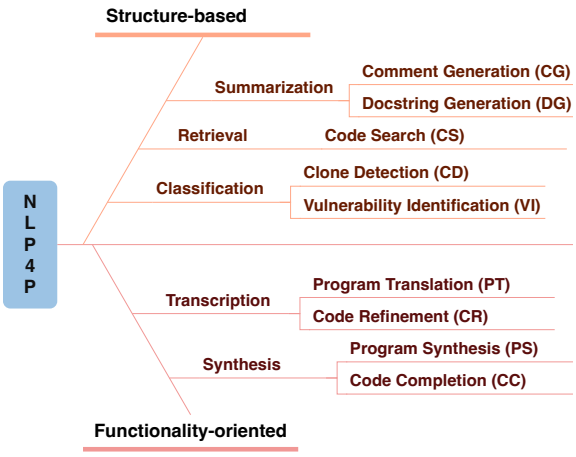

Figure 2: Categories of NLP4P tasks.

potential directions for future work (§7). The contributions of this paper are summarized as follows:

- We identify two properties of PL: structure-based and functionality-oriented, which are essential for program understanding and generation, respectively.
- From the perspective of the properties, we systematically review current work, covering tasks, datasets, evaluation methods, techniques, and representative models that achieve SOTA performance.
- By analysis of current NLP4P, we illustrate unexplored areas and suggest potential directions for future work.

2. Tasks

As shown in Figure 2, we classify a task as functionality-oriented if it aims at program generation; otherwise as structure-based. Within each category, we further divide the tasks according to application scenarios to cluster related tasks and highlight subtle differences between them. The partition of the structure-based and functionality-oriented roughly aligns with the partition of understanding and generation in NLP. In addition to the classical tasks, there is a trend in program-aided task which improves the performance of downstream tasks by programming. The two paradigms of the program-aided task are also consistent with our structure-based and functionality-oriented perspective, which will be introduced in detail at the end of this section.

2.1. Summarization Tasks

The summarization task summarizes a program into an NL description. It is crucial for the maintenance of software, especially those involving multiple developers. From the perspective of NLP, abstractive summarization is a generation task. We classify it as structure-based since PL lies in its input side, and the key to the task is understanding the content of PL by the structure. According to the format of the output, it can be further divided into **comment generation** (Nie et al., 2022) and **docstring generation** (Clement et al., 2020). The output of the latter contains some structural information, such as parameters and input/output examples.

2.2. Retrieval Tasks

The retrieval task mainly refers to the **code search**. It aims to retrieve relevant programs given NL query (Husain et al., 2019). It has a similar application scenario and input/output format to program synthesis. The difference is that its output is extracted from existing programs, rather than being synthesized from scratch. Sourcegraph[1] is a typical application of the code search. It enables rapid search of code from large codebases, aiding developers in finding relevant code snippets.

2.3. Classification Tasks

The classification task detects whether given programs have specific characteristics, e.g., being cloned (**clone detection**), or being vulnerable (**vulnerability identification**). They are essential in protecting software from the effects of ad-hoc reuse (Svajlenko et al., 2014) (e.g., Code Insight developed by Pattern Insight[2]) and cyber attacks (Zhou et al., 2019). The granularity of the input ranges from a coarse-grained software repository (Hovsepyan et al., 2012) to a fine-grained function (Russell et al., 2018; Zhou et al., 2019).

Despite the fact that NL does not explicitly occur in either input or output, we include tasks of such form for two reasons. First, PL has been demonstrated to contain abundant statistical properties similar to NL (Mou et al., 2016). Second, most of the ways that PL is processed are derived from NLP, like machine translation techniques (Tufano et al., 2019) in the transcription task (§ 2.5).

2.4. Synthesis Tasks

The synthesis task generates a program given a context (which can be NL, PL, or their mixture), thus can accelerate the development process. It can be further divided into program synthesis and code completion by the formal completeness of the output. The output of program synthesis is a

[1] https://sourcegraph.com/
[2] https://patterninsight.com

relatively independent unit, such as a function and a class, while the output of code completion is less restricted, ranging from tokens to code snippets.

Program synthesis is also called code generation. It is the systematic derivation of a program from a given specification (Manna and Waldinger, 1980). Conventional deductive approaches (Manna and Waldinger, 1980; Polozov and Gulwani, 2015) take logical specifications, which are logically complete but hard to write. Inductive approaches (Lieberman, 2001) list input-output examples as specifications, which are more accessible but incomplete. In contrast, an NL specification is sufficient to describe the logic of a program. Meanwhile, it is compatible with input-output examples by including them in a docstring. Therefore, it can take advantage of both the deductive and inductive approaches.

Code completion is also called code suggestion in early research (Tu et al., 2014; Hindle et al., 2016). It suggests the next program token given a context and has been widely applied to IDEs (Li et al., 2018). The application scenario includes the completion of method calls, keywords, variables, and arguments. With the bloom of the pre-trained models, the scenario has been extended to punctuations, statements, and even code snippets (Svyatkovskiy et al., 2020), further blurring the line between program synthesis and code completion. Copilot[3] is exactly a representative product that can accomplish both two tasks.

2.5. Transcription Tasks

The transcription task converts a given program to meet a specific requirement. Concretely, **program translation** aims to convert between high-level PL, e.g., C++ and Java (Zhu et al., 2022). It can accelerate the update of projects written by deprecated PL, and the migration of algorithms implemented by various PLs, represented by the TransCoder (Roziere et al., 2020). **Code refinement** aims to convert a buggy program into correct one (Wang et al., 2021) or optimize an existing program. It is closely related to vulnerability identification but is required to fix the detected bugs simultaneously. For instance, the Snyk[4] can identify and fix security issues in code. The transcription task differs from the synthesis task in two aspects. First, its input program is formally complete (input program is None or a function header in program synthesis, a partial code snippet in code completion). Second, its output can be strictly aligned with the input in both the format and the content.

[3]https://github.com/features/copilot
[4]https://snyk.io/

2.6. Program-aided Tasks

The program-aided approach improves the performance of a target task by the following two main paradigms: using program structure to construct intermediate representation, and executing program functionality to obtain the expected output.

Typically, the target task of the first paradigm involves structured commonsense reasoning, whose output is a graph (Madaan et al., 2022; Wang et al., 2023a). Since language models cannot directly generate a graph, a sequential intermediate representation is generated instead, and subsequently converted into a graph. In this way, the program is intrinsically superior to intermediate representations in other formats. First, the program is highly structured and qualified to represent graphical information. Second, the program is an essential source of the training data of LLM (especially code LLM), thus is more readily to be processed than other intermediate representations.

The second paradigm generates a program whose functionality accomplishes the target task. The program can be simply an equation, which can be calculated by an interpreter to avoid numerical error (Gao et al., 2023), or API commands to invoke external tools (Surís et al., 2023).

3. Datasets

Datasets are the basis for supporting the learning process of tasks. We thus classify current datasets (exhaustively searched from 2016 to date) into general, structure-based, and functionality-oriented, following the categories of tasks (as shown in Table 1). The general dataset is less processed and can be used in the early learning stage regardless of tasks. The structure-based and functionality-oriented datasets are dedicated data that properly processed for specific tasks.

3.1. General vs. Dedicated

There are two primary sources of general datasets: 1) open-source platforms such as GitHub, GitLab, and SeCold, 2) community-based spaces like Stack Overflow. The datasets are automatically collected and large in scale, thus can be applied to pre-training to ensure generated PL is grammatically correct and logically valid. However, sometimes they are noisy and non-informative. For instance, a commit message like “update” is of little substantial content; a code snippet answer might be irrelevant to its question (Iyer et al., 2018).

Structure-based datasets are specially formatted to support particular tasks. Concrete structure information is available via open-source parsers, e.g.,

		Data	Source	PL	Size	Type
General Dataset		PanguCoder (2022)	GitHub	Python	147	NL-PL PL
		The Pile (2020)	GitHub, ArXiv,...	-	825	NL PL
		Ahmad et al., 2021	GitHub, Stack Overflow	Java, Python	655	NL PL
		Fried et al., 2022	GitHub, GitLab, Stack Overflow	28	216	NL PL
		The Stack	GitHub	30	3,100	PL
		Li et al., 2022	GitHub	12	715	PL
		BigQuery	GitHub	C/C++, Go, Java, JS, Python	340	PL
		BIGPYTHON (2022)	GitHub	Python	217	PL
		CodeParrot	GitHub	Python	180	PL
		Chen et al., 2021	-	Python	159	PL
		GCPY (2022)	GitHub	Python	-	PL
Struc-based	CG	CodeNN (2016)	Stack Overflow	C#, SQL	<1	NL-PL
	CS	CodeSearchNet (2019)	GitHub	Go, Java, JS, PHP, Python, Ruby	17	NL-PL PL
	CD	BigCloneBench (2014)	SeCold	Java	2	PL
		POJ-104 (2016)	-	C/C++	<1	PL
	VI	Devign (2019)	QEMU, FFmpeg	C	<1	PL
Functionality-oriented	PS	CONCODE (2018)	GitHub	Java	13	NL-PL
		CodeNet (2021)	AIZU, AtCoder	55	8	NL-PL
		CodeContests (2022)	CodeNet, Codeforces, Caballero et al., 2016	C/C++, Java, Python	3	NL-PL
		APPS (2021)	Codewars, AtCoder, Kattis, Codeforces	Python	1	NL-PL
		Code Alpaca (2023)	Machine-generated	Python	<1	NL-PL
		HumanEval (2021)	Hand-craft	Python	<1	NL-PL
		HumanEval-X (2023)	Hand-craft	C++, Go, Java, JS, Python	<1	NL-PL
		MBPP (2021)	Hand-craft	Python	<1	NL-PL
		DS-1000 (2023)	Stack Overflow	Python	<1	NL-PL
		CoderEval (2023)	Github	Java, Python	<1	NL-PL
		AixBench (2022b)	-	Java	<1	NL-PL
		AixBench-L (2023a)	Github	Java	<1	NL-PL
	CC	PY150 (2016)	GitHub	Python	<1	PL
		Github Java (2013)	GitHub	Java	<1	PL
	PT	CodeTrans	Lucene, POI, JGit, Antlr	C#, Java	<1	PL
	CR	Bugs2Fix (2019)	GitHub	Java	15	PL

Table 1: An overview of datasets. Struc-based denotes the structure-based. Abbreviations in upper case denote tasks (Figure 2). For datasets that contain numerous kinds of PL, the total number of PLs is reported instead of concrete PL types. The unit of data size is GB. NL-PL in the last column denotes a parallel dataset whose all samples contain paired NL and PL. Single NL or PL denotes a monolingual dataset whose dominant language is NL or PL.

Tree-sitter.[5] Most functionality-oriented datasets contain a number of test cases for each sample to verify the functional correctness of synthesized programs. Therefore, the datasets are typically hand-crafted (Chen et al., 2021; Austin et al., 2021) or collected from online judge websites (Iyer et al., 2018; Puri et al., 2021; Hendrycks et al., 2021; Li et al., 2022), including AIZU, AtCoder, Codeforces, Codewars, and Kattis.

[5]https://tree-sitter.github.io/tree-sitter/

3.2. Parallel vs. Monolingual

To further explore the potential of datasets outside their original tasks, we divide them into parallel (denoted as NL-PL) and monolingual (denoted as NL or PL). We define a dataset as parallel if **all** samples include paired NL and PL, otherwise as monolingual. The type of monolingual datasets is denoted by their dominant language. For instance, Stack Overflow QA pairs with optional code snippets are denoted as NL, and GitHub programs with optional comments are denoted as PL. Note that a dataset may consist of multiple subsets of different

types, we explicitly list them in the last column of Table 1. Generally, parallel datasets are relatively homogeneous, and thus can support other tasks whose dataset is of the same type. For example, CodeSearchNet can also be used for comment generation (Lu et al., 2021). In contrast, a monolingual dataset is usually task-related with specific labels, limiting its generalization ability for other tasks.

4. Evaluation Methods

The evaluation metric is also closely related to the task. Considering that the retrieval and classification tasks are well-defined and their metrics (such as F1, MRR, and accuracy) are universally accepted, we focus on the summarization, synthesis, and transcription tasks, whose evaluation remains an open question. Concretely, the output of summarization is NL. Thus the evaluation can directly refer to NLP. For the synthesis and transcription task, whose output is PL, the functionality-oriented property is the main concern, assisted by the structure-based property as an auxiliary.

4.1. NL Evaluation

NL evaluation can refer to NLP and be conducted by the following two complementary approaches. **Automatic Evaluation** is usually implemented by comparing the n-grams between the predicted output and given references. Concrete metric includes BLEU (Papineni et al., 2002), MENTOR (Banerjee and Lavie, 2005), and ROUGE (Lin, 2004). However, limited by the number of references, they might correlate weakly with the real quality (Liu et al., 2016b). Hence, it is crucial to conduct a human evaluation simultaneously. **Human Evaluation** consists of several independent dimensions, such as naturalness, diversity, and informativeness. Human evaluation is more accurate, fine-grained, and comprehensive than automatic evaluation. However, it is also time-consuming and labor-intensive, and thus can only be conducted on a small subset of the test set.

4.2. PL Evaluation

PL evaluation can be conducted by the following two methods using the references and the test cases as evidence, respectively.

Reference based Evaluation Regarding a program as a sequence of tokens, PL can also be evaluated by n-gram based NL metrics. To further capture the structure-based property, Ren et al. (2020) propose the CodeBLEU metric, which takes AST and data flow graph into consideration. Similar to NL, PL is expressive in that a program can be implemented differently, leading to the same weak correlation issue with a limited number of references.

Test Case based Evaluation Hendrycks et al. (2021) propose two metrics based on test cases: Test Case Average and Strict Accuracy. Suppose there is a single generated program and a varying number of test cases for each sample. Test Case Average computes the average test case pass rate over all samples. Strict Accuracy is a relatively rigorous metric. A program is regarded as accepted if and only if it passes all test cases, and the final Strict Accuracy is the ratio of accepted programs.

Actually, we can generate more than one (e.g., K) program for each sample to improve the performance. In this way, Strict Accuracy regards a sample as accepted if any of the K programs pass all test cases. Therefore, it is also called $p@k$ in some literature. The sampling size could be huge, but the number of submissions sometimes is limited, like the competition scenario. To highlight the difference between the sampling and submission, Li et al. (2022) further propose the $n@k$ metric, which computes the acceptance ratio when sampling k and submitting n programs.

The test case based evaluation is a remarkable progress, which has already in turn improved the training process via reinforcement learning (Le et al., 2022). Currently, the associated datasets are only available in program synthesis. Extending it to other functionality-oriented tasks is expected to gain similar improvement.

5. Techniques

The heterogeneity between NL and PL requires extra effort in techniques to process programs. First, the key to understanding the content of a program is appropriately representing its structure information. Second, at the heart of program generation is elaborately designing an algorithm to achieve functional correctness. Therefore, we introduce the techniques by structure-based understanding and functionality-oriented generation, respectively.

5.1. Structure-based Understanding

Compared with NL, PL has more sophisticated structures, such as hierarchy, loops, and recursions. Generally, it would benefit the performance by explicitly representing the structures with appropriate data structure, including relative distance, abstract syntax tree, control flow graph, program dependence graph, and code property graph.

Relative Distance typically refers to the distance between two tokens in the source code sequence.

In this way, it can be easily combined into token representations as a feature. Ahmad et al. (2020) represent the relative distance as a learnable embedding and introduce it into transformer models by biasing the attention mechanism. Results show that the relative distance is an effective alternative to AST to capture the structure information. Based on that, Zugner et al. (2021) further extend the concept of relative distance from textual context to AST. Jointly training a model with the two types of relative distance achieves further improvement.

Abstract Syntax Tree (AST) is a tree representation that carries the syntax and structure information of a program (Shi et al., 2021). It simplifies inessential parts (e.g., parentheses) of the parse tree by implying the information in its hierarchy. Each node of AST has **arbitrary number of children** organized in a **specific order**. Therefore, a lossless representation of AST should capture the two characteristics simultaneously. Despite that, some AST can be complex with a deep hierarchy (Guo et al., 2020), which delays the parsing time and increases the input length (up to 70%) (Guo et al., 2022).

Control Flow Graph (CFG) represents a program as a graph. Its node (also called a basic block) contains a sequence of successive statements executed together. Edges between nodes are directed, denoting the order of execution (Allen, 1970). CFG makes it convenient to locate specific syntactic structures (such as loops and conditional statements) and redundant statements.

Program Dependence Graph (PDG) is another graphical representation of a program. Nodes in PDG are statements and predicate expressions, and edges denote both data dependencies and control dependencies (Ferrante et al., 1987). The data dependencies describe the partial order between definitions and usages of variables, and have been demonstrated to be beneficial for program understanding (Krinke, 2001; Allamanis and Brockschmidt, 2017; Allamanis et al., 2018; Guo et al., 2020). Similar to CFG, control dependencies also model the execution order, but it highlights a statement or a predicate itself by determining edges according to its value (Liu et al., 2020a).

Code Property Graph (CPG) is a joint graph that merges AST, CFG, and PDG (Yamaguchi et al., 2014). In this way, it takes advantage of all the representations, and thus can comprehensively represent a program for structure-based tasks, such as vulnerability identification (Zhou et al., 2019) and comment summarization (Liu et al., 2020a).

Call Graph and Inter-procedural Control Flow Graph (ICFG) both provide rich dependency relationships among functions. As a reference, the recently released DeepSeek-Coder (Guo et al., 2024) has achieved a new SOTA among open-source code models. A significant innovation of its data processing is exactly the incorporation of file-level dependency. It would be interesting to explore if a more fine-grained function-level dependency in Call Graph and ICFG could gain further improvement in future work.

In summary, a structure-based representation benefits program understanding. Among the representations, relative distance takes the most concise form but has the minimum structure information, while CPG is the other extreme. AST, CFG, and PDG are a balance between conciseness and information capacity. As a tree representation, AST can be more easily integrated by a backbone model than the graphical CFG and PDG, and thus is the most widely used structure-based representation.

5.2. Functionality-oriented Generation

Distinct from NL, PL is executable and ought to convert an input into an expected output to implement specific functionality. The techniques to ensure functional correctness involve the entire period of developing a model, including preparing training data (self-instruct), generation (interactive programming), and post-processing (sampling & filtering).

Self-Instruct Supervised fine-tuning (SFT) is crucial for generating correct programs that aligned with human intentions (Hendrycks et al., 2021). However, instruction-code pairs for SFT are hard to collect in practice. To this end, Chaudhary (2023) employ the self-instruct approach (Wang et al., 2022) to automatically construct an instruction-code dataset named Code Alpaca via LLM. Based on the dataset, Luo et al. (2023) propose an evol-instruct approach to evolve existing samples into more complex and diverse samples. The evolution process can be iterated up to three times, gradually improving the functional correctness.

Interactive Programming Generally, programming is an evolutionary process involving multiple iterations, rather than writing from scratch at one time. For instance, it is difficult to solve a problem with a single submission despite the developers being experienced. As a reference, the average accept rate of Codeforce,[6] a competitive programming website, is only 50.03%. Interactive programming, a progressive paradigm with a natural language interface, may shed light on this problem. Concretely, the interaction can be conducted by

[6] http://codeforces.com/

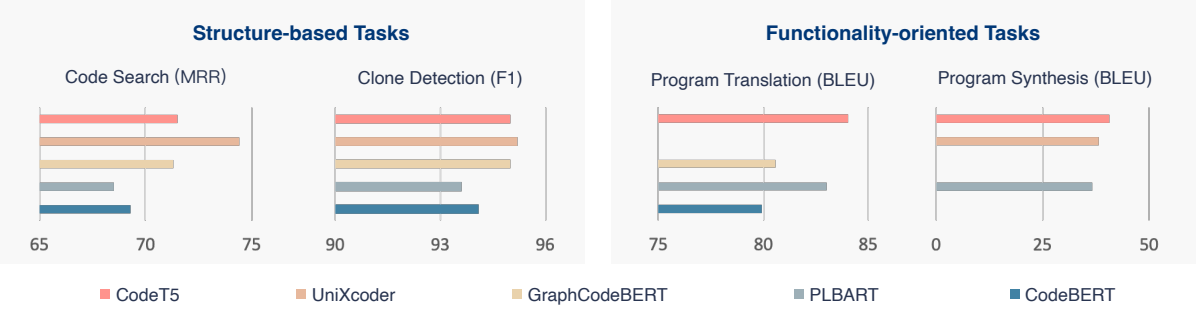

Figure 3: The results of structure-based models on CodeXGLUE. CodeT5, UniXcoder, and GraphCodeBERT, whose inputs have structure information, significantly outperform others on structure-based tasks. For functionality-oriented tasks like program translation, the performance of encoder-decoder architecture (CodeT5 and PLBART) is higher than that of decoder-only architecture (UniXcoder, GraphCodeBERT, and CodeBERT).

disentangling a problem into several simple subproblems and resolving them step by step (Nijkamp et al., 2022). Meanwhile, the interaction can also occur among iterations by taking into account the execution feedback (Zhang et al., 2023).

Sampling & Filtering To further improve the functional correctness, NLP4P has developed a sampling-based paradigm, which first samples a large volume of candidates and subsequently selects the desired program by test case based filtering and clustering techniques (Chen et al., 2021; Li et al., 2022; Nijkamp et al., 2022). To ensure good coverage of the desired program, first, the number of sampling should be as large as possible (up to 1M per problem in Li et al., 2022). Second, it would be better to employ the standard sampling with temperature or the top-k sampling algorithm, rather than the beam search, whose generated candidates can be pretty similar to each other (Li et al., 2016). The resulting programs are subsequently filtered by checking the functional correctness on given test cases (Li et al., 2022; Chen et al., 2022). However, the number of programs after filtering can still be huge if there are too many programs sampled. To fit the scenario where the number of submissions is limited, Li et al. (2022) propose a clustering strategy. It first clusters the programs according to their behaviors on generated test cases. Then it selects and submits a program from the clusters one by one. This strategy avoids repetitively submitting programs with identical bugs.

5.3. Backbone Models

Most of the functionality-oriented algorithms are model-agnostic and have little impact on the choice of a backbone model. In this section, we focus on the match between the structure-based representation (e.g., AST) and backbone models.

Recurrent Neural Network (RNN, Mikolov et al., 2010) and its variant LSTM (Hochreiter and Schmidhuber, 1997) are capable of processing variable-length inputs. Therefore, it is well-suited for representing NL description (Liu et al., 2016a; Weigelt et al., 2020) and PL token sequence (Wei et al., 2019). Meanwhile, it accepts the structure-based representation formatted as a sequence, e.g., the pre-order traversal of AST. However, such transforms are lossy in that the AST cannot be recovered. To this end, Hu et al. (2018) propose a structure-based traversal (SBT) approach, adding parentheses into the sequence to mark hierarchical relationships. Distinct from SBT that adapts data to a model, Shido et al. (2019) propose a Multi-way Tree-LSTM, which directly takes input as AST. It first encodes the children of a node with a standard LSTM, and subsequently integrates the results into the node with a Tree-LSTM.

Convolutional Neural Network (CNN, LeCun et al., 1989) extracts the features by scanning an input with a sliding window and applying stacked convolution and pooling operations on the window. Both two operations can be parallelized, making CNN more time-efficient than RNN. CNN in NLP4P usually takes input as execution traces (Gupta et al., 2020), input-output pairs (Bunel et al., 2018), and encodes them into an embedding as the output. Similar to Tree-LSTM, CNN can also be adapted to the structure-based representation. For instance, Mou et al. (2016) propose a tree-based convolutional neural network (TBCNN), which encodes AST by a weight-base and positional features.

Transformer (Vaswani et al., 2017) has a similar interface to RNN. It is more time-efficient and can better capture long-term dependencies, which is essential for processing PL since programs can be pretty long (Ahmad et al., 2020). Despite these approaches, some studies explore the usage of

feed-forward neural network (Iyer et al., 2016; Loyola et al., 2017), recursive neural network (Liang and Zhu, 2018), and graph neural network (Liu et al., 2020a). The architecture of the models, as well as RNN and CNN, can be flexibly adapted to customized data, e.g., the primitive AST. While for large-scale general data, the transformer is suggested due to its high capacity and easy access to pre-training.

6. Representative Pre-training Models

SOTA pre-training models can be roughly divided into two categories according to the benchmarks they are evaluated. The first category focuses on the CodeXGLUE benchmark (Lu et al., 2021), which is composed primarily of structure-based datasets, as shown in Figure 3. The second aims at passing the test cases of functionality-oriented datasets, as typified by HumanEval (Chen et al., 2021) in Figure 4. We denote the two categories as structure-based models and functionality-oriented models, respectively.

6.1. Structure-based Models

The performance of this category is shown in Figure 3. GraphCodeBERT (Guo et al., 2020), UniXcoder (Guo et al., 2022), and CodeT5 (Wang et al., 2021) incorporate data dependencies of PDG, AST, and node types of AST, respectively. As a reference, we also report the result of an encoder-only based CodeBERT (Feng et al., 2020) and an encoder-decoder based PLBART (Ahmad et al., 2021), neither of which utilize the structure-based representation.

Generally, incorporating structure-based representation can boost the performance of NLP4P tasks. Concretely, UniXcoder performs better on understanding tasks, while CodeT5 outperforms others on generation tasks. For program synthesis, although CodeT5 has the highest BLEU score, it would be better to use its variant CodeT5+ (Wang et al., 2023b) or other functionality-oriented models.

6.2. Functionality-oriented Models

Figure 4 shows the p@1 HumanEval results of representative functionality-oriented models, including CodeGen2.5 (Nijkamp et al., 2023), CodeT5+, StarCoder (Li et al., 2023b), and Code Llama (Rozière et al., 2023). We employ a unified "ModelName-Variant-#Parameter" format to highlight the differences between models. The suffixes "Python" and "Instruct" denote the variant that trained on extra Python data and SFT data, respectively.

Concretely, models in Figure 4 are roughly divided into three groups by the number of parameters. There is an increase from the 7B group to the 34B group, indicating that model size is the primary factor for improving functional correctness. Meanwhile, each "Python" variant significantly outperforms its "Base" counterpart, which indicates that constantly learning a specific PL is promising to gain improvement on the PL.[7]

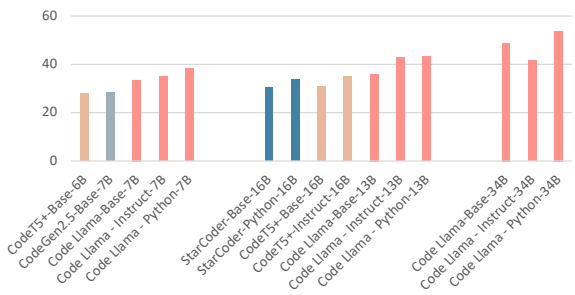

Figure 4: The p@1 results of functionality-oriented models on HumanEval benchmark. The two suffixes split by "-" in each model denote a variant and the number of parameters, respectively.

7. Future Directions

Taking advantage of both SE and NLP, NLP4P has achieved remarkable performance. However, some features of the two fields (e.g., multilingual learning in NLP) have not been sufficiently explored. Incorporating them is expected to further improve the performance on NLP4P tasks.

7.1. Multilingual NLP4P

As the bloom of the open source software platform, e.g., GitHub, source code, along with their NL descriptions, has accumulated to a considerable amount, making it possible to learn a data-driven NLP4P model. However, the distribution of these data is highly unbalanced. Most of the NL part is English, and the PL part is Java and Python. As a result, the performance of low-resource NL and PL is much worse than the average performance. For example, Ruby only takes a minor proportion in CodeSearchNet dataset and is inferior to other PL in both code search and comment generation tasks (Feng et al., 2020).

To bridge the gap between different languages, the simplest way is to translate a low-resource language into its high-resource counterpart. For tasks whose input is low-resource NL, we can translate it into English before sending it to the model. For tasks whose output is low-resource PL, we can first generate a Java program and subsequently translate it into the desired PL. However, it introduces

[7]The PL of HumanEval is Python.

extra effort and cascading errors during the translation. Multilingual learning approaches (Conneau et al., 2020; Liu et al., 2020b; Xue et al., 2021) provide access to address the issue. It can efficiently utilize the data presented in various languages, representing them in a unified semantic space and avoiding cascading errors.

7.2. Multi-modal NLP4P

NL specification may refer to other modalities (e.g., figures) for better understanding. For instance, the "Seven Bridge Problem", a classical graph problem, is hard to understand by plain NL descriptions. At the heart of the multi-modal approaches is the alignment of various modalities. However, there is no such dataset in the area of NLP4P, and annotating a new one is costly. Therefore, it would be crucial to utilize the knowledge entailed in the existing multi-modal datasets (e.g., COCO Lin et al., 2014) and the language-vision pre-trained models (such as CLIP Radford et al., 2021, Flamingo Alayrac et al., 2022, and METALM Hao et al., 2022a).

7.3. Long-Context Code Processing

Distinct from NL, PL has a long-distance dependency across multiple lines and files, which is hard for conventional NLP approaches to process. Potential solutions include (1) hierarchical approaches (Wu et al., 2021), which break long documents into segments and then hierarchically integrate them, and (2) retrieval-augmented approaches (Zhu et al., 2019; Cai et al., 2022), which retrieve the most relevant segments to compress the context.

8. Ethical Considerations

NLP4P techniques can potentially be misused for creating malicious software or leaking personally identifiable information. Fortunately, NLP4P itself can also be employed to detect malicious intent, e.g., detecting malicious emails with an AUC of 0.99 (Muralidharan and Nissim, 2023). In addition, NLP4P might potentially reduce the demand for repetitive programming work. However, it also creates opportunities for more creative programming roles, where NLP4P is utilized to augment human capabilities, rather than replacing them.

9. Conclusion

In this paper, we review a broad spectrum of NLP4P work. We identify two intrinsic properties of PL: structure-based and functionality-oriented, which are at the heart of program understanding and generation, respectively. They naturally partition the tasks, datasets, techniques, and models, highlighting the characteristics of each category. Additionally, the structure-based property is the key to the choice of backbone models, and the functionality-oriented property is the primary concern of evaluation methods. Through the analysis, we list topics that have yet to be fully considered and might be worth researching in the future.

10. Acknowledgments

We gratefully acknowledge the support of the National Natural Science Foundation of China (NSFC) via grant 62236004 and 62206078, and the support of Du Xiaoman (Beijing) Science Technology Co., Ltd.

11. Bibliographical References

Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2020. A transformer-based approach for source code summarization. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 4998–5007, Online. Association for Computational Linguistics.

Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified pre-training for program understanding and generation. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 2655–2668, Online. Association for Computational Linguistics.

Jean-Baptiste Alayrac, Jeff Donahue, Pauline Luc, Antoine Miech, Iain Barr, Yana Hasson, Karel Lenc, Arthur Mensch, Katie Millican, Malcolm Reynolds, et al. 2022. Flamingo: a visual language model for few-shot learning. *arXiv preprint arXiv:2204.14198*.

Miltiadis Allamanis and Marc Brockschmidt. 2017. Smartpaste: Learning to adapt source code. *arXiv preprint arXiv:1705.07867*.

Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2018. Learning to represent programs with graphs. In *International Conference on Learning Representations*.

Miltiadis Allamanis and Charles Sutton. 2013. Mining source code repositories at massive scale using language modeling. In *2013 10th working conference on mining software repositories (MSR)*, pages 207–216. IEEE.

Frances E Allen. 1970. Control flow analysis. *ACM Sigplan Notices*, 5(7):1–19.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. 2021. Program synthesis with large language models. *CoRR*, abs/2108.07732.

Satanjeev Banerjee and Alon Lavie. 2005. METEOR: An automatic metric for MT evaluation with improved correlation with human judgments. In *Proceedings of the ACL Workshop on Intrinsic and Extrinsic Evaluation Measures for Machine Translation and/or Summarization*, pages 65–72, Ann Arbor, Michigan. Association for Computational Linguistics.

Rudy Bunel, Matthew Hausknecht, Jacob Devlin, Rishabh Singh, and Pushmeet Kohli. 2018. Leveraging grammar and reinforcement learning for neural program synthesis. In *International Conference on Learning Representations*.

Ethan Caballero, . OpenAI, and Ilya Sutskever. 2016. Description2Code Dataset.

Deng Cai, Yan Wang, Lemao Liu, and Shuming Shi. 2022. Recent advances in retrieval-augmented text generation. In *Proceedings of the 45th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 3417–3419.

Sahil Chaudhary. 2023. Code alpaca: An instruction-following llama model for code generation. https://github.com/sahil280114/codealpaca.

Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2022. Codet: Code generation with generated tests. *arXiv preprint arXiv:2207.10397*.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code.

Fenia Christopoulou, Gerasimos Lampouras, Milan Gritta, Guchun Zhang, Yinpeng Guo, Zhongqi Li, Qi Zhang, Meng Xiao, Bo Shen, Lin Li, et al. 2022. Pangu-coder: Program synthesis with function-level language modeling. *arXiv preprint arXiv:2207.11280*.

Colin B Clement, Dawn Drain, Jonathan Timcheck, Alexey Svyatkovskiy, and Neel Sundaresan. 2020. Pymt5: multi-mode translation of natural language and python code with transformers. *arXiv preprint arXiv:2010.03150*.

Alexis Conneau, Kartikay Khandelwal, Naman Goyal, Vishrav Chaudhary, Guillaume Wenzek, Francisco Guzmán, Edouard Grave, Myle Ott, Luke Zettlemoyer, and Veselin Stoyanov. 2020. Unsupervised cross-lingual representation learning at scale. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 8440–8451, Online. Association for Computational Linguistics.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547, Online. Association for Computational Linguistics.

Jeanne Ferrante, Karl J Ottenstein, and Joe D Warren. 1987. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(3):319–349.

Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. 2022. Incoder: A generative model for code infilling and synthesis. *arXiv preprint arXiv:2204.05999*.

Leo Gao, Stella Biderman, Sid Black, Laurence Golding, Travis Hoppe, Charles Foster, Jason Phang, Horace He, Anish Thite, Noa Nabeshima, et al. 2020. The pile: An 800gb dataset of diverse text for language modeling. *arXiv preprint arXiv:2101.00027*.

Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. 2023. PAL: Program-aided language models. In *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pages 10764–10799. PMLR.

Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. UniXcoder: Unified cross-modal pre-training for code representation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 7212–7225, Dublin, Ireland. Association for Computational Linguistics.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, LIU Shujie, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. 2020. Graphcodebert: Pre-training code representations with data flow. In *International Conference on Learning Representations*.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y Wu, YK Li, et al. 2024. Deepseek-coder: When the large language model meets programming–the rise of code intelligence. *arXiv preprint arXiv:2401.14196*.

Kavi Gupta, Peter Ebert Christensen, Xinyun Chen, and Dawn Song. 2020. Synthesize, execute and debug: Learning to repair for neural program synthesis. *Advances in Neural Information Processing Systems*, 33:17685–17695.

Yaru Hao, Haoyu Song, Li Dong, Shaohan Huang, Zewen Chi, Wenhui Wang, Shuming Ma, and Furu Wei. 2022a. Language models are general-purpose interfaces. *arXiv preprint arXiv:2206.06336*.

Yiyang Hao, Ge Li, Yongqiang Liu, Xiaowei Miao, He Zong, Siyuan Jiang, Yang Liu, and He Wei. 2022b. Aixbench: A code generation benchmark dataset. *arXiv preprint arXiv:2206.13179*.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, et al. 2021. Measuring coding challenge competence with apps. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2)*.

Abram Hindle, Earl T Barr, Mark Gabel, Zhendong Su, and Premkumar Devanbu. 2016. On the naturalness of software. *Communications of the ACM*, 59(5):122–131.

Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation*, 9(8):1735–1780.

Aram Hovsepyan, Riccardo Scandariato, Wouter Joosen, and James Walden. 2012. Software vulnerability prediction using text analysis techniques. In *Proceedings of the 4th international workshop on Security measurements and metrics*, pages 7–10.

Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018. Deep code comment generation. In *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*, pages 200–20010. IEEE.

Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*.

Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2016. Summarizing source code using a neural attention model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2073–2083, Berlin, Germany. Association for Computational Linguistics.

Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2018. Mapping language to code in programmatic context. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 1643–1652, Brussels, Belgium. Association for Computational Linguistics.

Jens Krinke. 2001. Identifying similar code with program dependence graphs. In *Proceedings Eighth Working Conference on Reverse Engineering*, pages 301–309. IEEE.

Royi Lachmy, Ziyu Yao, Greg Durrett, Milos Gligoric, Junyi Jessy Li, Ray Mooney, Graham Neubig, Yu Su, Huan Sun, and Reut Tsarfaty. 2021. Proceedings of the 1st workshop on natural language processing for programming (nlp4prog 2021). In *Proceedings of the 1st Workshop on Natural Language Processing for Programming (NLP4Prog 2021)*.

Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Wen-tau Yih, Daniel Fried, Sida Wang, and Tao Yu. 2023. Ds-1000: A natural and reliable benchmark for data science code generation. In *International Conference on Machine Learning*, pages 18319–18345. PMLR.

Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven Hoi. 2022. CodeRL: Mastering code generation through pretrained models and deep reinforcement learning. In *Advances in Neural Information Processing Systems*.

Yann LeCun, Bernhard Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne Hubbard, and Lawrence D Jackel. 1989. Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1(4):541–551.

Jia Li, Yongmin Li, Ge Li, Zhi Jin, Yiyang Hao, and Xing Hu. 2023a. Skcoder: A sketch-based approach for automatic code generation. *arXiv preprint arXiv:2302.06144*.

Jian Li, Yue Wang, Michael R Lyu, and Irwin King. 2018. Code completion with neural attention and pointer networks. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence*, pages 4159–25.

Jiwei Li, Will Monroe, and Dan Jurafsky. 2016. A simple, fast diverse decoding algorithm for neural generation. *arXiv preprint arXiv:1611.08562*.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. 2023b. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. 2022. Competition-level code generation with alphacode. *arXiv preprint arXiv:2203.07814*.

Yuding Liang and Kenny Zhu. 2018. Automatic generation of text descriptive comments for code blocks. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 32.

Henry Lieberman. 2001. *Your wish is my command: Programming by example*. Morgan Kaufmann.

Chin-Yew Lin. 2004. ROUGE: A package for automatic evaluation of summaries. In *Text Summarization Branches Out*, pages 74–81, Barcelona, Spain. Association for Computational Linguistics.

Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. 2014. Microsoft coco: Common objects in context. In *European conference on computer vision*, pages 740–755. Springer.

Chang Liu, Xinyun Chen, Eui Chul Shin, Mingcheng Chen, and Dawn Song. 2016a. Latent attention for if-then program synthesis. *Advances in Neural Information Processing Systems*, 29.

Chia-Wei Liu, Ryan Lowe, Iulian Serban, Mike Noseworthy, Laurent Charlin, and Joelle Pineau. 2016b. How NOT to evaluate your dialogue system: An empirical study of unsupervised evaluation metrics for dialogue response generation. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pages 2122–2132, Austin, Texas. Association for Computational Linguistics.

Shangqing Liu, Yu Chen, Xiaofei Xie, Jing Kai Siow, and Yang Liu. 2020a. Retrieval-augmented generation for code summarization via hybrid gnn. In *International Conference on Learning Representations*.

Yinhan Liu, Jiatao Gu, Naman Goyal, Xian Li, Sergey Edunov, Marjan Ghazvininejad, Mike Lewis, and Luke Zettlemoyer. 2020b. Multilingual denoising pre-training for neural machine translation. *Transactions of the Association for Computational Linguistics*, 8:726–742.

Pablo Loyola, Edison Marrese-Taylor, and Yutaka Matsuo. 2017. A neural architecture for generating natural language descriptions from source code changes. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 287–292, Vancouver, Canada. Association for Computational Linguistics.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1)*.

Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2023. Wizardcoder: Empowering code large language models with evol-instruct. *arXiv preprint arXiv:2306.08568*.

Aman Madaan, Shuyan Zhou, Uri Alon, Yiming Yang, and Graham Neubig. 2022. Language models of code are few-shot commonsense learners. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 1384–1403, Abu Dhabi, United Arab Emirates. Association for Computational Linguistics.

Zohar Manna and Richard Waldinger. 1980. A deductive approach to program synthesis. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2(1):90–121.

Tomas Mikolov, Martin Karafiát, Lukas Burget, Jan Cernockỳ, and Sanjeev Khudanpur. 2010. Recurrent neural network based language model. In *Interspeech*, volume 2, pages 1045–1048. Makuhari.

Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. 2016. Convolutional neural networks over tree structures for programming language processing. In *Thirtieth AAAI conference on artificial intelligence*.

Trivikram Muralidharan and Nir Nissim. 2023. Improving malicious email detection through novel designated deep-learning architectures utilizing entire email. *Neural Networks*, 157:257–279.

Pengyu Nie, Jiyang Zhang, Junyi Jessy Li, Ray Mooney, and Milos Gligoric. 2022. Impact of

evaluation methodologies on code summarization. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 4936–4960.

Erik Nijkamp, Hiroaki Hayashi, Caiming Xiong, Silvio Savarese, and Yingbo Zhou. 2023. Codegen2: Lessons for training llms on programming and natural languages. *ICLR*.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. A conversational paradigm for program synthesis. *arXiv preprint arXiv:2203.13474*.

Changan Niu, Chuanyi Li, Bin Luo, and Vincent Ng. 2022. Deep learning meets software engineering: A survey on pre-trained models of source code. *arXiv preprint arXiv:2205.11739*.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, pages 311–318, Philadelphia, Pennsylvania, USA. Association for Computational Linguistics.

Oleksandr Polozov and Sumit Gulwani. 2015. Flashmeta: A framework for inductive program synthesis. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 107–126.

Ruchir Puri, David S Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladimir Zolotov, Julian Dolby, Jie Chen, Mihir Choudhury, Lindsey Decker, et al. 2021. Codenet: A large-scale ai for code dataset for learning a diversity of coding tasks. *arXiv preprint arXiv:2105.12655*.

Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, et al. 2021. Learning transferable visual models from natural language supervision. In *International Conference on Machine Learning*, pages 8748–8763. PMLR.

Veselin Raychev, Pavol Bielik, and Martin Vechev. 2016. Probabilistic model for code with decision trees. *ACM SIGPLAN Notices*, 51(10):731–747.

Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. Codebleu: a method for automatic evaluation of code synthesis. *arXiv preprint arXiv:2009.10297*.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

Baptiste Roziere, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. 2020. Unsupervised translation of programming languages. *Advances in Neural Information Processing Systems*, 33:20601–20611.

Rebecca Russell, Louis Kim, Lei Hamilton, Tomo Lazovich, Jacob Harer, Onur Ozdemir, Paul Ellingwood, and Marc McConley. 2018. Automated vulnerability detection in source code using deep representation learning. In *2018 17th IEEE international conference on machine learning and applications (ICMLA)*, pages 757–762. IEEE.

Ensheng Shi, Yanlin Wang, Lun Du, Hongyu Zhang, Shi Han, Dongmei Zhang, and Hongbin Sun. 2021. Cast: Enhancing code summarization with hierarchical splitting and reconstruction of abstract syntax trees. *arXiv preprint arXiv:2108.12987*.

Yusuke Shido, Yasuaki Kobayashi, Akihiro Yamamoto, Atsushi Miyamoto, and Tadayuki Matsumura. 2019. Automatic source code summarization with extended tree-lstm. In *2019 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE.

Dídac Surís, Sachit Menon, and Carl Vondrick. 2023. Vipergpt: Visual inference via python execution for reasoning. *arXiv preprint arXiv:2303.08128*.

Jeffrey Svajlenko, Judith F Islam, Iman Keivanloo, Chanchal K Roy, and Mohammad Mamun Mia. 2014. Towards a big data curated benchmark of inter-project code clones. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 476–480. IEEE.

Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. 2020. Intellicode compose: Code generation using transformer. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1433–1443.

Zhaopeng Tu, Zhendong Su, and Premkumar Devanbu. 2014. On the localness of software. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 269–280.

Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2019. An empirical study on learning bug-fixing patches in the wild via neural machine translation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 28(4):1–29.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc.

Xingyao Wang, Sha Li, and Heng Ji. 2023a. Code4Struct: Code generation for few-shot event structure prediction. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 3640–3663, Toronto, Canada. Association for Computational Linguistics.

Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A Smith, Daniel Khashabi, and Hannaneh Hajishirzi. 2022. Self-instruct: Aligning language model with self generated instructions. *arXiv preprint arXiv:2212.10560*.

Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi DQ Bui, Junnan Li, and Steven CH Hoi. 2023b. Codet5+: Open code large language models for code understanding and generation. *arXiv preprint arXiv:2305.07922*.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859*.

Bolin Wei, Ge Li, Xin Xia, Zhiyi Fu, and Zhi Jin. 2019. Code generation as a dual task of code summarization. *Advances in neural information processing systems*, 32.

Sebastian Weigelt, Vanessa Steurer, Tobias Hey, and Walter F. Tichy. 2020. Programming in Natural Language with fuSE: Synthesizing Methods from Spoken Utterances Using Deep Natural Language Understanding. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 4280–4295, Online. Association for Computational Linguistics.

Chuhan Wu, Fangzhao Wu, Tao Qi, and Yongfeng Huang. 2021. Hi-transformer: Hierarchical interactive transformer for efficient and effective long document modeling. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 2: Short Papers)*, pages 848–853, Online. Association for Computational Linguistics.

Linting Xue, Noah Constant, Adam Roberts, Mihir Kale, Rami Al-Rfou, Aditya Siddhant, Aditya Barua, and Colin Raffel. 2021. mT5: A massively multilingual pre-trained text-to-text transformer. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 483–498, Online. Association for Computational Linguistics.

Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. 2014. Modeling and discovering vulnerabilities with code property graphs. In *2014 IEEE Symposium on Security and Privacy*, pages 590–604. IEEE.

Hao Yu, Bo Shen, Dezhi Ran, Jiaxin Zhang, Qi Zhang, Yuchi Ma, Guangtai Liang, Ying Li, Tao Xie, and Qianxiang Wang. 2023. Codereval: A benchmark of pragmatic code generation with generative pre-trained models. *arXiv preprint arXiv:2302.00288*.

Daoguang Zan, Bei Chen, Fengji Zhang, Dianjie Lu, Bingchao Wu, Bei Guan, Wang Yongji, and Jian-Guang Lou. 2023. Large language models meet NL2Code: A survey. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 7443–7464, Toronto, Canada. Association for Computational Linguistics.

Kechi Zhang, Zhuo Li, Jia Li, Ge Li, and Zhi Jin. 2023. Self-edit: Fault-aware code editor for code generation. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 769–787, Toronto, Canada. Association for Computational Linguistics.

Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Lei Shen, Zihan Wang, Andi Wang, Yang Li, et al. 2023. Codegeex: A pre-trained model for code generation with multilingual benchmarking on humaneval-x. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 5673–5684.

Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. *Advances in neural information processing systems*, 32.

Ming Zhu, Karthik Suresh, and Chandan K Reddy. 2022. Multilingual code snippets training for program translation.

Qingfu Zhu, Lei Cui, Wei-Nan Zhang, Furu Wei, and Ting Liu. 2019. Retrieval-enhanced adversarial training for neural response generation. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 3763–3773, Florence, Italy. Association for Computational Linguistics.

Daniel Zugner, Tobias Kirschstein, Michele Catasta, Jure Leskovec, and Stephan Gunnemann. 2021. Language-agnostic representation learning of source code from structure and context. In *International Conference on Learning Representations (ICLR)*.

12. Language Resource References

Jacob Austin and Augustus Odena and Maxwell Nye and Maarten Bosma and Henryk Michalewski and David Dohan and Ellen Jiang and Carrie J. Cai and Michael Terry and Quoc V. Le and Charles Sutton. 2021. *Program Synthesis with Large Language Models*. PID https://github.com/google-research/google-research/tree/master/mbpp.

Chen, Mark and Tworek, Jerry and Jun, Heewoo and Yuan, Qiming and de Oliveira Pinto, Henrique Ponde and Kaplan, Jared and Edwards, Harrison and Burda, Yuri and Joseph, Nicholas and Brockman, Greg and others. 2021. *Evaluating Large Language Models Trained on Code*. PID https://github.com/openai/human-eval.

Hendrycks, Dan and Basart, Steven and Kadavath, Saurav and Mazeika, Mantas and Arora, Akul and Guo, Ethan and Burns, Collin and Puranik, Samir and He, Horace and Song, Dawn and Steinhardt, Jacob. 2021. *Measuring Coding Challenge Competence With APPS*. Curran. PID https://github.com/hendrycks/apps.

Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2018. Mapping language to code in programmatic context. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 1643–1652, Brussels, Belgium. Association for Computational Linguistics.

Li, Yujia and Choi, David and Chung, Junyoung and Kushman, Nate and Schrittwieser, Julian and Leblond, Rémi and Eccles, Tom and Keeling, James and Gimeno, Felix and Lago, Agustin Dal and others. 2022. *Competition-level code generation with alphacode*. PID https://github.com/google-deepmind/code_contests.

Puri, Ruchir and Kung, David and Janssen, Geert and Zhang, Wei and Domeniconi, Giacomo and Zolotov, Vladimir and Dolby, Julian T and Chen, Jie and Choudhury, Mihir and Decker, Lindsey and Thost, Veronika and Thost, Veronika and Buratti, Luca and Pujar, Saurabh and Ramji, Shyam and Finkler, Ulrich and Malaika, Susan and Reiss, Frederick. 2021. *CodeNet: A Large-Scale AI for Code Dataset for Learning a Diversity of Coding Tasks*. Curran. PID https://github.com/IBM/Project_CodeNet/tree/main.