IDE Augmented with Human-Learning Inspired Natural Language Programming

Mitchell Young
North Carolina State University
Raleigh, North Carolina, USA
mgyoung@ncsu.edu

Zifan Nan
North Carolina State University
Raleigh, North Carolina, USA
znan@ncsu.edu

Xipeng Shen
North Carolina State University
Raleigh, North Carolina, USA
xshen5@ncsu.edu

ABSTRACT

Natural Language (NL) programming, the concept of synthesizing code from natural language inputs, has garnered growing interest among the software community in recent years. Unfortunately, current solutions in the space all suffer from the same problem, they require many labeled training examples due to their data-driven nature. To address this issue, this paper proposes an NLU-driven approach that forgoes the need for large numbers of labeled training examples. Inspired by how humans learn programming, this solution centers around Natural Language Understanding and draws on a novel graph-based mapping algorithm. The resulting NL programming framework, HISyn, uses no training examples, but gives synthesis accuracies comparable to data-driven methods trained on hundreds of samples. HISyn meanwhile demonstrates advantages in terms of interpretability, error diagnosis support, and cross-domain extensibility. To encourage adoption of HISyn among developers, the tool is made available as an extension for the Visual Studio Code IDE, thereby allowing users to easily submit inputs to HISyn and insert the generated code expressions into their active programs. A demo of the HISyn Extension can be found at https://youtu.be/KKOqJS24FNo.

CCS CONCEPTS

• **Software and its engineering → Source code generation**; **Domain specific languages**; **Integrated and visual development environments**.

KEYWORDS

Program synthesis, natural language programming, code editor

ACM Reference Format:
Mitchell Young, Zifan Nan, and Xipeng Shen. 2022. IDE Augmented with Human-Learning Inspired Natural Language Programming. In *44th International Conference on Software Engineering Companion (ICSE '22 Companion), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3510454.3516832

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ICSE '22 Companion, May 21–29, 2022, Pittsburgh, PA, USA

© 2022 Association for Computing Machinery.

ACM ISBN 978-1-4503-9223-5/22/05…$15.00
https://doi.org/10.1145/3510454.3516832

1 INTRODUCTION

Natural Language (NL) programming is the process of automatically generating program code for a target domain from natural language input queries. Figure 1 demonstrates this concept using an example for the ASTMatcher Domain Specific Language (DSL), a C++ library used for source code analysis of the Clang compiler abstract syntax tree (AST). From a simple English description of the desired program, the NL programming tool generates working code using the domain's API set. Because of the convenience such an intuitive programming interface offers to general users (e.g., IoT [12], ASTMatcher [9]), recent years have witnessed a growing interest in NL programming.

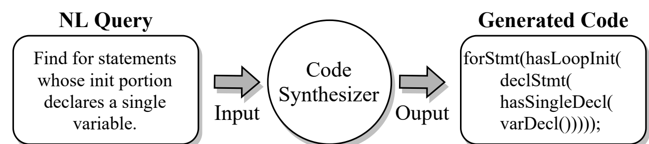

Figure 1: NL Programming example for ASTMatcher DSL.

Existing approaches to NL programming tools fall into two classes: data-driven and rule-driven approaches. The former features the reliance on many labeled input-code pairs as training data to build up some statistical models; the latter depends on predefined domain-specific rules. The rule-based approach showed success in the early stages of the field's development (e.g., Smartsynth [5]), but has gradually lost traction due to the lack of robustness and the difficulties in generalizing across domains. The data-driven approach has dominated recent efforts, represented by the adoption of deep learning to map NL queries to code via various neural networks (e.g., [1, 4, 6, 10, 11]). Although this approach has shown more promise than the previous rule-driven approach, its requirement of large numbers of labeled examples hinders its adoption, especially for domains where labeled examples are scarce. Recent proposals show the possibility of generating examples for a certain domain [2], but it is yet unclear how well these methods can generate truly representative examples in complex domains.

The pitfalls of these approaches leave us with the following challenge faced by NL programming: *Can we eliminate the cumbersome need for large numbers of training examples while still maintaining the robustness and cross-domain extensibility seen in data-driven approaches?* To address this challenge, we proposed HISyn (for "human learning inspired synthesizer") [8] , the first NL programming framework driven by natural language understanding (NLU), which attempts to model the same process through which humans approach writing programs. Rather than building solutions using

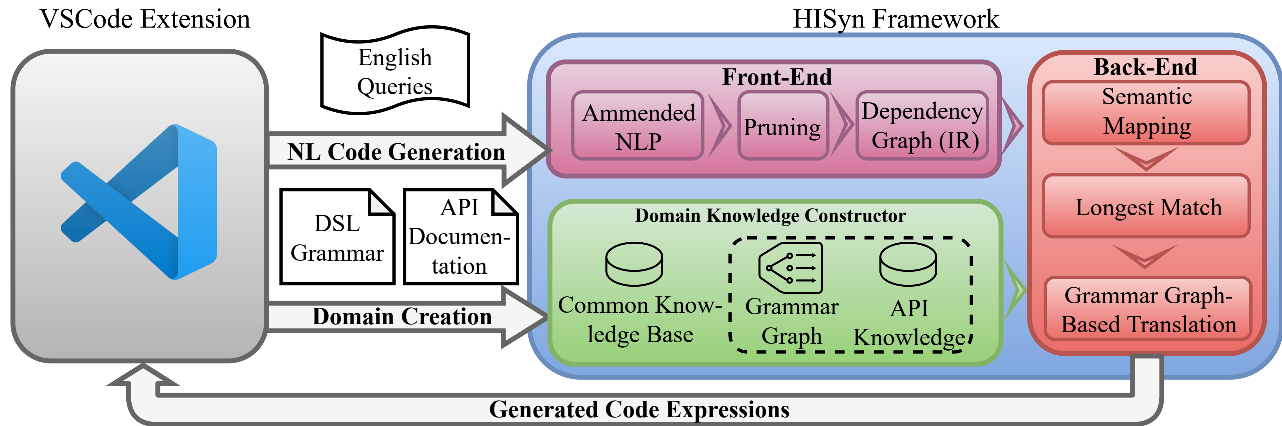

Figure 2: HISyn framework design and integration with VS Code Extension

snippets from training examples, as in the data-driven approach, HISyn uses natural language processing (NLP) to build a deeper understanding of a programmer's intentions and a domain's API documentation. From this natural language understanding, output code expressions are then generated by following the grammar for the DSL. The advantage of this approach is that it requires no training data and is general enough that the framework can easily be extended to new domains by building separate modules for each DSL.

To further improve the appeal of HISyn to programmers, the tool is made available as an extension to the Visual Studio Code IDE (VS Code)[1]. By integrating the tool with this development environment, users can interface with HISyn directly from their code editor and automatically incorporate generated code expressions into their active program. While similar developer tools augmented with NL programming exist (e.g. Github Copilot[2]), this is the first to allow users to add support for new domains and which requires no training data to do so. Furthermore, the extension also includes additional features such as snippet support and domain-specific tools, which when paired with HISyn's NL programming framework create a hollistic tool for aiding programmers writing DSL specific programs. In particular, the HISyn tool is best suited for programmers working with domains that have large and complex API sets (e.g., Python libraries [1]). In these cases, all but the most experienced programmers will have difficulty memorizing all the APIs and creating working programs from them. Thus, rather than having to constantly refer to API documentation for these domains, users can instead leverage the NL programming of the HISyn extension to more efficiently develop their programs.

In the sections to follow we will cover the NLU-driven approach of the HISyn framework and describe how target users can leverage the tool through its integration with VS Code. Then, we will describe the procedure used to evaluate the performance of HISyn and discuss the results on the following three domains: Text Editing Language, Air Travel Information System (ATIS), ASTMatcher.

[1]https://code.visualstudio.com/
[2]https://copilot.github.com/

2 APPROACH OF HISYN

The framework of HISyn [8] is shown in Figure 2. It features a modular design where domain-specific modules are separated from a generalized core. This modularity is what allows HISyn to be more easily extended to new domains compared to other NL programming approaches. The generalized core of the framework is comprised of three main components: (1) a domain knowledge constructor that processes the domain knowledge used during code synthesis; (2) a front-end that transforms each NL-based query to a dependency graph which serves as the basis for the intermediate representation (IR) that the back-end works on; (3) a back-end that employees grammar-graph-based translation to generate domain specific code from the IR. In the rest of this section we will describe each individual components at a high-level. For more details on the concepts discussed herein, see the original publication [8] on the approach.

The first main component of HISyn is the domain knowledge constructor, which processes and stores domain information leveraged by the back-end during code generation. Some of this information is shared between all domains. Namely a WordNet [7] synonym list, preposition dictionary, and Named Entities (NE). Collectively, we refer to this shared information as the common knowledge base. The WordNet synonym list and preposition dictionary are both used in the Semantic Mapping step of the back-end. The WordNet synonym list maps word tokens to their synonyms, while the preposition dictionary maps preposition tokens (which are ignored by WordNet) directly to semantically related words. For example, the preposition "from" maps to "start", "source", and "origin". The Named Entities are the labels assigned to words that represent real-world objects. For example, January's NE is Month, Baltimore's NE is City.

In addition to this common knowledge base, the domain knowledge constructor also processes the information unique to each distinct domain module. We refer to each of these sets as a domain knowledge base. For each domain, this will contain a grammar graph and an API knowledge base. The grammar graph defines the search space used during code generation for a given domain in the Grammar-Graph-Based Translation step of the back-end.

```
1)      forStmt
        input: ForStmt
        return: Stmt
        description: Matches for statements.
```

```
2)      ForStmt_arg     :=      empty
                        |       Matcher
                        |       hasBody(HasBody_arg)
                        |       hasCondition(HasCondition_arg)
                        |       hasIncrement(HasIncrement_arg)
                        |       hasLoopInit(HasLoopInit_arg)
                        |       ...
```

Figure 3: Example inputs for ASTMatcher domain knowledge base. 1) API documentation for "forStmt" API. 2) Grammar for "ForStmt_arg" type.

It is constructed from the context-free grammar for the domain written in Backus-Naur form (BNF). The API knowledge is used in the Semantic Mapping step of the back-end for mapping words from the input queries to candidate APIs. This API knowledge base is generated by parsing the API documentation and contains the API name, input/output types, and natural language description for all APIs in the domain. Figure 3 shows examples from the context-free grammar and API documentation inputs used to construct the ASTMatcher domain knowledge base.

The second main component of HISyn is the front-end, which is responsible for transforming the NL English query into an intermediary representation (IR) that is used for code generation in the back-end. To do so, the front-end first applies light regulation to the NL-based query to avoid term confusions for words that have domain-specific meaning. Next, it uses multiple NLP techniques including POS tagging, Lemmatization, NER, and dependency parsing to produce a dependency graph. Non-essential words (called function words) are then pruned from the dependency graph based on the POS tag and the dependency relations of each word. This pruned dependency graph is the final IR that is passed to the back-end for code generation.

Lastly, the back-end component of HISyn is responsible for employing the novel synthesizing algorithm *grammar graph-based translation* to generate code according to the pruned IR. Grammar graph-based translation first semantically maps each node in the IR to a set of APIs based on the lemma and synonyms of words in each API's description. Each node can be mapped to one, several, or no corresponding APIs, which we call candidate APIs. HISyn uses a longest match scheme to handle the phrases in the query. If multiple nodes map to the same candidate, HISyn groups these nodes as a cluster and selects a single set of candidate APIs for the cluster instead of for each node. With the IR now annotated with candidate APIs, we then search the grammar graph to find all possible subgraphs that contains exactly one candidate API from each node in the IR. Each of these subgraphs represents a grammatically correct candidate output code expression for this domain. Of these candidates, the one chosen for the final code generation output is the one with the minimum number of APIs. The justification for this being that all candidate subgraphs contain all the key information conveyed by the query, so additional APIs are likely to just contain redundant and/or unnecessary information. Hence, the final expression should contains as few as possible.

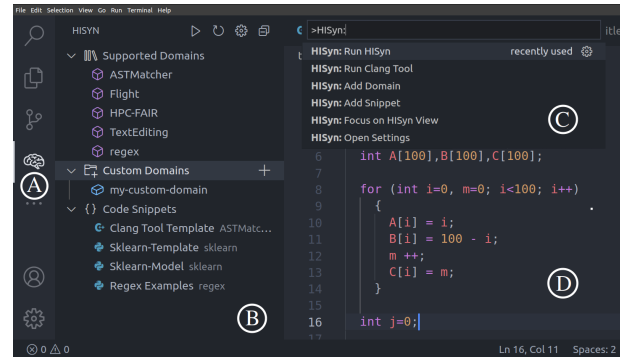

Figure 4: HISyn Extension Interface: (A) Side Panel Icon, (B) Main Interface, (C) Command Palette, (D) Code Editor.

3 AN IDE AUGMENTED WITH NL PROGRAMMING

We next propose an integration of the HISyn framework with the environment where the generated code will most likely be used, the IDE (Integrated Development Environment). Specifically, we introduce the HISyn Extension, an extension to the Visual Studio Code IDE that allows users to interface with HISyn and leverage its code generation functionality from the comfort of the text editor. Figure 4 shows the basic layout of the HISyn Extension interface.

The main features provided by the extension include: (1) NL-Based Code Generation; (2) Custom Domain Creation; (3) Snippet Support; (4) Domain Specific Tools. Each feature is exposed to the user through commands executable within VS Code. These commands can be accessed either directly through the extension interface, figure 4(B), or using VS Code's built-in command palette, figure 4(C). In the rest of this section, we will go into more detail on each of the above features regarding their usage scenarios and expected behaviors. For examples of these features in action, please refer to our demo video[3].

3.1 NL-Based Code Generation

The primary feature of the extension allows users to leverage the NL programming available through HISyn from within the VS Code editor. This functionality can be executed using the "HISyn: Run HISyn" command, which opens a dialog for users to submit their domain and a natural language description of their desired code expression to the HISyn tool. Within a few seconds of submission, the top code expressions generated by HISyn will be displayed in the dialog. The user can then select their preferred result which will automatically be inserted into the active editor. Figure 5 shows this process and an example output copied to the editor.

By default, this code generation is executed via API calls on a HISyn engine running on a remote server. As such, a stable internet connection is required and only the domains currently supported by the server will be available. For users who find that their preferred domain is not currently supported, the extension can instead be configured to run through a locally installed version of HISyn and the domain creation feature may be used to add new domains. To

[3]https://youtu.be/KKOqJS24FNo

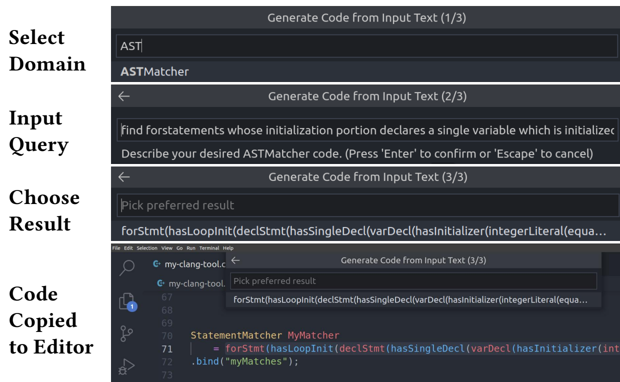

Figure 5: Series of input dialogues for NL-based code generation through HISyn Extension.

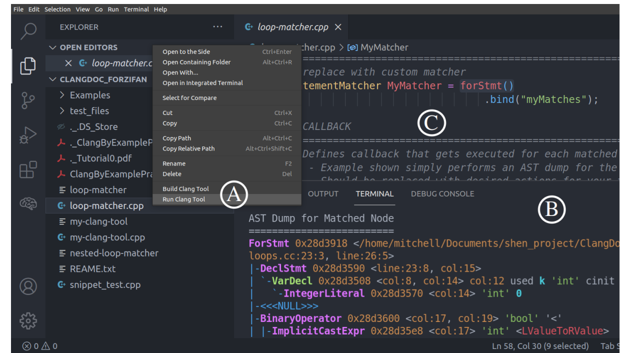

Figure 6: Additional support for ASTMatcher domain, including: (A) Build & Run Clang Tool Commands, (B) Tool Terminal Output, (C) Clang Tool Template Snippet.

add a new domain, users can execute the "HISyn: Add Domain" command, which opens a dialogue for submitting a name for the new domain, as well as its Grammer & API documentation. Once submitted, the domain knowledge constructor will automatically incorporate the new domain into the local HISyn installation and be made available for selection during code generation.

3.2 Domain Support

In addition to the code generation functionality exposed through HISyn, support for code snippets and domain specific tools has also been incorporated into the extension to further assist users when working with certain domains. For example, the code expressions generated by HISyn for many of the supported domains will not serve as executable code in and of themselves. These code expressions will need to be wrapped by more code blocks for importing the APIs, initializing the referenced variable names, outputting the results to files, etc. To assist users in these types of scenarios, the provided code snippets can be used to quickly generate the bulk of this wrapper code. Moreover, since the included snippets cannot possibly cover all possible usage-scenarios, the extension also allows users to create their own snippets or edit existing snippets.

Beyond the code snippets, additional commands have been added to the extension to further aid users working with the ASTMatcher domain. This domain is typically used to develop tools for the Clang compiler and LLVM-Project, a process that is not well documented and prone to errors. For this reason, the extension includes the "Build Clang Tool" and "Run Clang Tool" commands to allow users to easily compile their ASTMatcher tool into their Clang build and run the tool against selected source files. Figure 6 shows an example of a Clang tool being built, compiled, and executed using these additional tools. Note that the tool itself was built using the "Clang Tool Template" code snippet and modified using code expressions generated by HISyn for the ASTMatcher domain.

4 EVALUATION

To evaluate the efficacy of the HISyn framework, experiments were conducted across several different domains. These experiments were intended primarily to determine if HISyn can produce code expressions which are comparable in accuracy to data-driven methods and whether this accuracy is dependent on the difficulty of the NL query. In the rest of this section we will describe the methodology behind the experiments and discuss the results for each domain.

4.1 Methodology

Three different DSLs were used for our experiments on HISyn. For each domain, HISyn was tested against a dataset of English queries with corresponding ground truth DSL expressions. Below are brief descriptions of these DSLs and the datasets used for each.

(1) **Text Editing Language**[4]: a command language intended for performing text editing in Office suite applications. DSL consists of 52 APIs. Dataset includes 467 English query/DSL pairs.
(2) **Air Travel Information System (ATIS)**[5]: a SQL style language used as a benchmark for querying air traffic control data. DSL consists of 51 APIs. Dataset includes 535 English query/DSL pairs.
(3) **ASTMatcher**[6]: a library included in the Clang and LLVM Project used for constructing matchers that identify particular patterns in the Clang AST. DSL consists of 505 APIs and over 200 data types. Dataset includes 50 English query/DSL pairs.

The datasets of English query/DSL pairs for the first two domains, Text Editing Language and ATIS, both come from their respective work [3]. For the third dataset, ASTMatcher, the 50 English query/DSL pairs were collected manually, with the ASTMatcher expressions drawn from Clang-tidy and the English descriptions generated independently by 5 graduate students. Each ASTMatcher expression is described using a single English sentence from at least two students. Additionally, for the ASTMatcher DSL official API documentation was used for our experiments; however, no documentation is provided for the Text Editing and ATIS DSLs. For these domains, API documentation was created manually based off descriptions and examples.

All DSL test cases were tested on the HISyn framework as well as a baseline synthesizer for comparison. To evaluate the performance

[4]shorturl.at/npFIS
[5]shorturl.at/sxyS5
[6]https://clang.llvm.org/docs/LibASTMatchersReference.html

Table 1: Accuracy of HISyn vs. baseline model for 3 different domains broken down by query difficulty.

Domain	HISyn Acc. (%)			Comparison Acc. (%)
	Easy	Hard	Overall	Overall
Text Editing	88.03	59.3	82.59	82.3
ATIS	89.81	64.56	85.4	88.4
ASTMatcher	89.29	68.18	80.0	68.0

on these test cases, the *synthesis accuracy* evaluation metric was chosen. This accuracy measures the percentage of the total test cases for which HISyn generates the correct code expressions. A generated code expression is considered correct if it matches the ground truth DSL expression exactly (including APIs, variables, ordering, etc.). A data-driven synthesizer [3] was used for the Text Editing and ASIT baselines, while a rule-based synthesizer was used for the ASTMatcher domain. The chosen data-driven synthesizer was selected for comparison with HISyn because it gives accuracies on-par with existing domain-specific NL-based synthesizers and, like HISyn, is cross-domain extensible. For the ASTMatcher domain, a rule-based synthesizer [5] was used instead because satisfactory accuracies could not be achieved using a data-driven synthesizer given the limited set of training examples available.

4.2 Results

Table 1 reports the results of our experiments, including the overall accuracy of HISyn and the comparison baseline, as well as the HISyn accuracy broken down by the difficulty of the input query. It should be noted that the comparison accuracies for the Text Editing and ATIS domains come from the paper [3] directly. From these results, one can see that HISyn achieves accuracies comparable to the data-driven method for the Text Editing and ATIS domains. On the other hand, for the ASTMatcher domain HISyn achieves a significantly higher accuracy of 80% compared to the rule-based approach at 68%.

In the breakdown by query difficulty, we characterize easy and hard queries as those with token lengths shorter and longer than the domain average, respectively. The average query lengths of the Text Editing, ATIS, and ASTMatcher domains are 7, 12, and 9 tokens. This breakdown shows that for all 3 domains HISyn achieves much higher accuracies on the shorter queries, dropping by about 20% for longer queries. This reduced accuracy for longer queries has two major causes: (1) in the front-end, the NLP engine is more likely to generate an incorrect dependency graph for long queries; (2) in the back-end, incorrect candidate grammar subgraphs are generated due to the increased number of paths to select and combine.

5 CONCLUSION

This paper introduces the HISyn Extension, a NL programming tool for the VS Code IDE that features NLU-driven code synthesis, a novel approach to NL programming. Experiments on the HISyn framework demonstrate that an NLU-driven solution, without the use of training examples, can produce results comparable to data-driven methods. This makes the HISyn Extension a viable NL-programming tool for developers working with complex domains that lack extensive documented examples. Using the tool, programmers can easily generate DSL expressions through HISyn and incorporate the output into their active code editor. Furthermore, because the approach features a modular design and does not require example collection, users can easily extend the tool to support new domains. In the future, we hope to improve this tool further by adding support for more popular domains (e.g. Python libraries) and adding commands to help users generate DSL grammar/API documentation.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation (NSF) under Grants CNS-1717425 and the Department of Energy (DOE) under Grant DE-SC0013700. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF or DOE.

REFERENCES

[1] Rohan Bavishi, Caroline Lemieux, Roy Fox, Koushik Sen, and Ion Stoica. 2019. AutoPandas: neural-backed generators for program synthesis. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–27.

[2] Giovanni Campagna, Silei Xu, Mehrad Moradshahi, Richard Socher, and Monica S Lam. 2019. Genie: A generator of natural language semantic parsers for virtual assistant commands. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 394–410.

[3] Aditya Desai, Sumit Gulwani, Vineet Hingorani, Nidhi Jain, Amey Karkare, Mark Marron, Subhajit Roy, et al. 2016. Program synthesis using natural language. In *Proceedings of the 38th International Conference on Software Engineering*. ACM, 345–356.

[4] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. 2016. Deep API learning. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 631–642.

[5] Vu Le, Sumit Gulwani, and Zhendong Su. 2013. Smartsynth: Synthesizing smartphone automation scripts from natural language. In *Proceeding of the 11th annual international conference on Mobile systems, applications, and services*. ACM, 193–206.

[6] Xi Victoria Lin, Chenglong Wang, Luke Zettlemoyer, and Michael D Ernst. 2018. NL2Bash: A Corpus and Semantic Parser for Natural Language Interface to the Linux Operating System. *arXiv preprint arXiv:1802.08979* (2018).

[7] George A Miller. 1998. *WordNet: An electronic lexical database*. MIT press.

[8] Zifan Nan, Hui Guan, and Xipeng Shen. 2020. HISyn: human learning-inspired natural language programming. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 75–86.

[9] Zifan Nan, Hui Guan, Xipeng Shen, and Chunhua Liao. 2021. Deep NLP-Based Co-Evolvement for Synthesizing Code Analysis from Natural Language. In *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction (CC)*. 141–152.

[10] Emilio Parisotto, Abdel-rahman Mohamed, Rishabh Singh, Lihong Li, Dengyong Zhou, and Pushmeet Kohli. 2016. Neuro-symbolic program synthesis. *arXiv preprint arXiv:1611.01855* (2016).

[11] Illia Polosukhin and Alexander Skidanov. 2018. Neural Program Search: Solving Programming Tasks from Description and Examples. *arXiv preprint arXiv:1802.04335* (2018).

[12] Chris Quirk, Raymond Mooney, and Michel Galley. 2015. Language to code: Learning semantic parsers for if-this-then-that recipes. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*. 878–888.