Sketch Then Generate: Providing Incremental User Feedback and Guiding LLM Code Generation through Language-Oriented Code Sketches

Chen Zhu-Tian
ztchen@umn.edu
University of Minnesota
Minneapolis, MN, USA

Zeyu Xiong
zeyuxionghci@outlook.com
University of Minnesota
Minneapolis, MN, USA

Xiaoshuo Yao
yaoxiaos@usc.edu
USC
Los Angeles, CA, USA

Elena Glassman
glassman@seas.harvard.edu
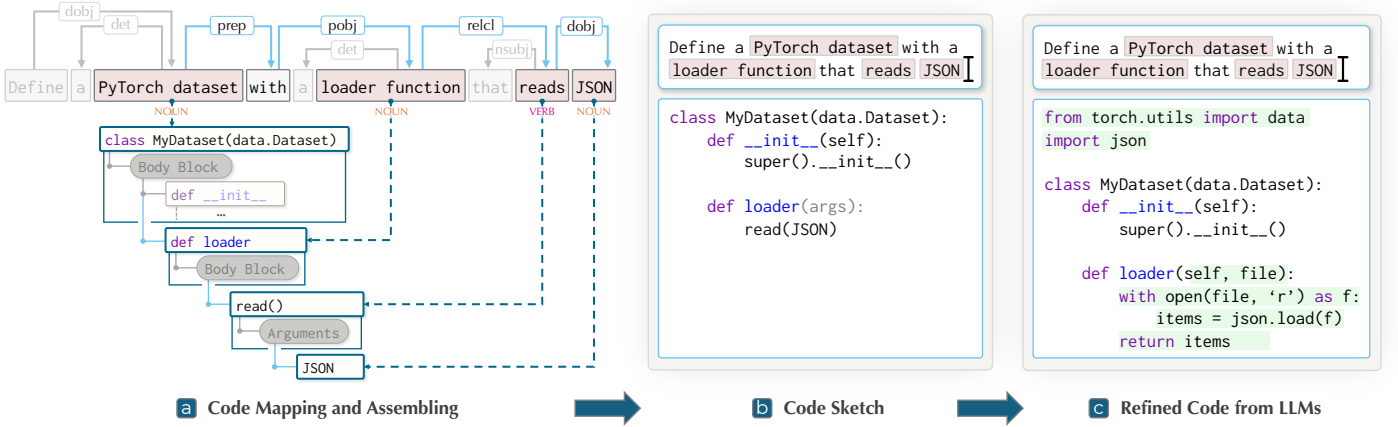Harvard University
Boston, MA, USA

Figure 1: We derive a code sketch from the user's prompt a) by mapping the phrases to code elements then assembling them. The code sketch can b) offer user feedback on the intended code structure and c) guide LLMs in generating the final code.

ABSTRACT

Crafting effective prompts for code generation or editing with Large Language Models (LLMs) is not an easy task. Particularly, the absence of immediate, stable feedback during prompt crafting hinders effective interaction, as users are left to mentally imagine possible outcomes until the code is generated. In response, we introduce *Language-Oriented Code Sketching*, an interactive approach that provides instant, incremental feedback in the form of code sketches (i.e., incomplete code outlines) during prompt crafting. This approach converts a prompt into a code sketch by leveraging the inherent linguistic structures within the prompt and applying classic natural language processing techniques. The sketch then serves as an intermediate placeholder that not only previews the intended code structure but also guides the LLM towards the desired code, thereby enhancing human-LLM interaction. We conclude by discussing the approach's applicability and future plans.

KEYWORDS

Language-Oriented Interaction, Code Generation, LLMs, HAI

1 INTRODUCTION

The advent of Large Language Models (LLMs) represents a paradigm shift in programming, enabling an era where natural language prompts can directly inform code generation and editing [31, 32]. Despite this advancement, programmers frequently encounter a tedious loop of trial and error, continuously refining their prompts to guide LLMs to the desired code, e.g., as shown in the user studies of GitHub Copilot [2, 10, 27]. Existing solutions (e.g., [11, 21, 25, 30]), while useful, fall short in providing user feedback during the prompt crafting process. The lack of such feedback can lead to uncertainty about the expected output and potential inefficiencies in the development process, as users are left to mentally imagine possible outcomes until the code is generated [12, 29].

However, providing user feedback during prompt crafting presents significant challenges. Ideally, feedback should not only be *instant* but also evolve *incrementally* as the user types their prompt, offering clear and understandable intermediate results that progressively guide the LLM towards the final desired code. The most straightforward approach—continuously sending the user's prompt to LLMs for feedback—is currently impractical, due to the computational demands and the unpredictable nature of LLM outputs, which can vary greatly as the prompt evolves [27]. These limitations highlight the need for innovative approaches that extend beyond mere reliance on LLMs.

One observation, however, is that users' prompts often include phrases directly referencing code elements. Moreover, these phrases are embedded within narratives that reflect the relationships among these elements in the intended code structure. For example, consider the prompt *"Define a PyTorch dataset with a loader function ..."* (Figure 1a). Here, `PyTorch dataset` and `loader function` reference a class and a function, respectively. Their linguistic relationship — where `loader function` is a child of `PyTorch dataset`

associated through *"with"* — reveals that the function is meant to be a component of the class. This linguistic structure of the prompt offers an interesting opportunity to derive a *code sketch* — an incomplete code outline [23]) — by mapping these phrases to predefined code elements and assembling them based on their linguistic relationship. The resultant sketch (Figure 1b) not only offers a preview of the intended code structure of the prompt, but also serves as a foundational layer for LLMs to further refine and complete (Figure 1c). Such a process can be achieved using classic natural language processing (NLP) techniques, bypassing the constraints inherent in solutions based solely on LLMs.

Based on the observation, we develop **Language-Oriented Code Sketching**, an interactive approach that incrementally derives a code sketch as the user types a prompt. We next introduce the technical details of our approach and discuss its applicability, along with user feedback collected through a technology probe.

2 RELATED WORK

The interaction between programmer and LLMs in coding has been the subject of extensive research, e.g., [2, 9, 10, 15, 16, 19, 22, 27]. Among the challenges in coding with AI assistants, the crafting of effective prompts has repeatedly been identified as a primary concern. This challenge stems from the need for a deep understanding of LLMs' interpretative mechanisms and the ability to articulate intentions in natural language. To address this challenge, researchers have develop various interactive systems and approaches to assist users in writing effective prompts [1, 3, 11, 21, 25, 30]. For instance, Coladder [30] introduces a block-based user interface that enables programmers to hierarchically decompose and refine their prompts. Liu el al. [11] proposed Grounded Code Matching, a method where the LLM generates code and accompanying explanations, allowing users to refine these explanations to iteratively improve the LLM's output. Despite useful, these solutions fall short in providing user feedback during the prompt crafting process. The slow feedback loop between writing a prompt, submitting it, and awaiting the generated code can undermines productivity and escalates frustration.

This work aims to provide instant, incremental user feedback during the prompt crafting process by deriving code sketches from the latent linguistic structures of the prompts on-the-fly. Our approach shares a similar spirit to methods in semantic parsing [8], which translates natural language into formal, executable structures, e.g., SQL or code. However, our approach is different in two key aspects: 1) we focus on generating incomplete code sketches for user feedback rather than executable code; and 2) our method emerges naturally in the user's prompt typing process, without imposing additional effort on the user. In this sense, our method is closer to systems that treats the natural language as implicit data sources to foster fluid and streamlined interactions. Such a method has been applied in various domains, such as creating data reports [33], unit animations [4], augmented videos [34], and graphics [28]. Additionally, inspired by structural code editors [5], when deriving the sketch, we manipulate the underlying abstract syntax tree of the code sketch, instead of directly editing its textual representation. This enables us to fully leverage the structures in both the prompt and the code itself.

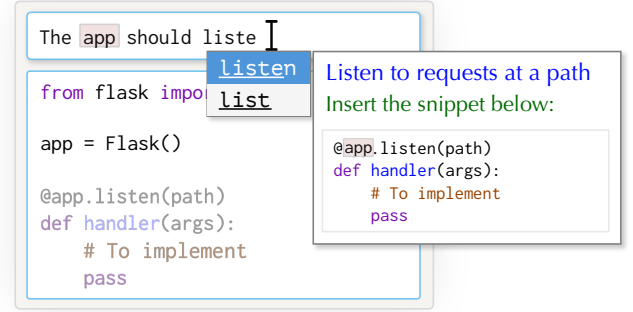## 3 LANGUAGE-ORIENTED CODE SKETCHING

Figure 2: As the user types a phrase, our system suggests corresponding code elements and previews their integration into the existing code. When the user accepts a suggestion, the code element is inserted and the association between the phrase and the code element is preserved.

Our approach leverages word completion as an interface (Figure 2) to facilitate seamless collaboration between human and machine intelligence. It includes three key steps:

(1) **Map**: As a user types, the system maps the current phrase to a list of potential corresponding code elements.
(2) **Assemble**: For each code element, the system assembles it with the existing code by comparing the linguistic relationships between the current phrase and previous phrases against a predefined rule set. Valid assemblies are presented as suggestions and previewed for the user in the code editor upon selection.
(3) **Preserve**: Once the user accepts a suggestion, the system inserts the corresponding code elements to the code editor, and completes the typing phrase accordingly. The association between the phrase and the code element is preserved.

The user can submit their prompt and the resultant code sketch to LLMs at any time to get the completed code. We next explain the details of these three steps.

3.1 Map

When a user types a phrase, we aim to map it to potential corresponding code elements. We identify that a phrase can refer to a code element in two distinct ways – *independently* or *dependently* – and thus employ two strategies to derive the potential mappings.

Independent Phrase: Matching Predefined Code Snippets or Existing Code. A phrase can independently point to specific code elements using exact identifiers (e.g., variable or function names) or reserved keywords. For example, in the prompt *"The app should liste ..."*, the word `app` directly refers to a variable in the code. Thus, we use fuzzy string matching to match the phrase with two sources of code elements:

- *Predefined Code Snippets.* We provide a list of code snippets as potential matches for the typed word. A code snippet is a small block of re-usable code, ranging from simple code fragments to complex function or class templates. Modern

IDEs often provide predefined snippets. For example, Figure 2 shows a snippet for an API function from the Python Flask library [17].

- *Existing Code.* We also dynamically parses the existing code to suggest identifiers, such as function names or variables, as potential matches for the typed word.

Dependent Phrase: Inferring Based on Linguistic Features.
A phrase can also refer to code elements based on its relationship with other phrases within the prompt. This type of phrase requires linguistic context to resolve the code element it refers to. We use two NLP techniques to infer the corresponding code elements:

- *Part-of-Speech.* In user's prompt, nouns usually indicate variables, while verbs suggest function calls. Recognizing these patterns, we employ a *part-of-speech tagging* [18] of the typed phrase and map it to a variable or function accordingly. For example, Figure 1a shows that the system maps the words `reads` (a verb) and `JSON` (a noun) to a function call and an argument, respectively.
- *Co-reference.* Certain phrases, such as *"it"* or *"the function"*, commonly refer back to other specific phrases mentioned earlier in the user's prompt. Through *co-reference analysis* [24], we map these phrases to the corresponding code elements of the earlier phrases they refer to.

3.2 Assemble

Once a phrase is mapped to a potential code element, the next challenge is assembling this element with other existing code elements. To achieve this, we leverage the linguistic structures of the prompt that indicate relationships between code elements. Specifically, given the current phrase and its corresponding code elements, our approach involves three main steps:

(1) **Dependency Parsing**. We begin with *dependency parsing* [20] to uncover the linguistic relationships between the current phrase and other phrases in the prompt. This parsing technique organizes phrases into a dependency tree, showing how each phrase grammatically relates to and depends on its parent. The links above the words in Figure 1a shows an example of dependency tree.

(2) **Rule Matching**. We then iteratively exam the linguistic relationships between the current phrase to other phrases associated with code elements following a reverse reading order. If a relationship aligns with a predefined rule in our rule set, we proceed with the assembly. Figure 3 shows a list of rule examples implemented in our current system.

(3) **AST Assembly**. Instead of merely assembling plain text, we construct the Abstract Syntax Tree (AST) of these elements, which provides a tree-like structure of the code that facilitates easier management. Each rule in our rule set also specifies how the ASTs of the two code elements involved in the rule should be constructed. This process also adheres to the programming language's grammar, discarding any assemblies that do not comply.

Finally, only the validated assemblies will be suggested to the user. The user can select a suggestion (Figure 2) and review how the code element will be inserted into the code editor.

Assembly	Rule

The incoming code element is a **constituents** of an existing code element.	`nsubj` – `ROOT` | `conj*` – `dobj` | `conj*`
	
	 Example: The `function` returns a `list` …
	
	 `nsubj` | `dobj` – `relcl` | `conj*` – `dobj` | `conj*`
	
	 Example: … a `class` that contains a `function` …
	
	 `nsubj` | `dobj` – `prep` – `pobj` | `conj*`
	
	 Example: … a `class` with a `function` …

The incoming code element is a **function call** of or within an existing code element.	`nsubj` – `ROOT` | `conj*`
	
	 Example: The `handler` first `verifies` …
	
	 `nsubj` | `dobj` – `relcl` | `conj*`
	
	 Example: … a `loader function` that `reads` …

The incoming code element is an **argument** of an existing function call.	`ROOT` | `relcl` | `conj*` – `dobj` | `conj*`
	
	 Example: … `reads` a `file` …
	
	 `ROOT` | `relcl` | `conj*` – `prep` – `pobj` | `conj*`
	
	 Example: … `listen` at an `endpoint` …

The incoming code element is a **value or name** of an existing code element.	`dobj` | `conj*` – `appos` | `conj*`
	
	 Example: … two `layers`, a `Conv2D` and a `ReLU` …
	
	 `dobj` | `conj*` – `acl` – `oprd`
	
	 Example: … a `function` called `MyFunc` …

Figure 3: Example rules for code assembly, each represented as a regular expression that specifies the linguistic relationship between the incoming phrase and an existing code-associated phrase within the dependency tree. The labels used (e.g, `nsubj` and `dobj`) are established terms in NLP.

3.3 Preserve

Once a user accepts a suggestion, our system preserves the established association between the phrase and its corresponding code element. This preserved association is crucial for the system to support incrementally building the code sketch. As the user inputs additional words, our system utilizes these prior associations to assemble the newly coming code elements as introduce in Sec. 3.2.

3.4 Guiding LLMs in Code Generation

The derived code sketch not only provides instant, incremental feedback to the user but also can guide the subsequent code generation process. For example, we fill the prompt template below with the prompt and code sketch shown in Figure 1b.

> **Prompt template:**
> Based on the description and a code sketch (which is incomplete and buggy) below, do your best to complete the code with **minimal** editions. Generate the code within a triple quotes: ```python ... ```
>
> ### Code Description
> {prompt}
>
> ### Current Code Draft
> {code sketch}

Once the LLM responds, we can streamingly display the edits made by the LLM on the code sketch, as illustrated in Figure 1c. This process aids users in better understanding the generated code, as the sketch previews the code during the prompt-typing process.

4 PRELIMINARY USER FEEDBACK

We implement our approach in a technology probe to gather preliminary feedback from colleagues who were unfamiliar with the project. A formal study is planned to further evaluate the usability of our approach, identify areas for improvement, and explore its real-world applications. Below, we briefly outline the implementation of the probe and the initial feedback received.

Implementation. We implemented our approach using a web-based code editor built with React [13], CodeMirror [7], and TypeScript [14], with NLP techniques powered by SpaCy [6]. The ASTs for the code are obtained using Tree Sitter [26], an incremental parsing library. Furthermore, a Flask-based backend server was crafted to manage user prompts and facilitate interactions with the GPT-4 API from OpenAI. While currently supporting only Python, we envision extending our implementation to additional programming languages in subsequent iterations.

User Feedback. Feedback on the internal review was highly positive about the sketch, noting that it reduces the cognitive load for users by visualizing the potential structure of the code. Interestingly, users found the code sketch useful even when the assembly of code elements was not entirely accurate (e.g., a function inserted with incorrect indentation) because they expected the LLM to correct these errors. Additionally, users reported that it helps them better control the LLM. We believe this is due to two factors: 1) the code sketch provides an outline for the LLM, and 2) the feedback nudges users to write longer and more detailed prompts. The most significant concern regarding our approach is its robustness, which is unsurprising since we have only implemented a minimal set of rules in this technology probe. We anticipate that this issue can be alleviated with better engineering and more comprehensive rule sets. Feedback also indicated that sometimes the action of suggestion acceptance can be disruptive, particularly when the default suggestion is the intended one.

5 DISCUSSION AND FUTURE WORK

Target Contexts for Language-Oriented Code Sketching. Previous research [2] has found that the use of AI coding assistants varies according to the user's clarity about the final code solution.

For example, Copilot Chat is favored for exploratory coding when the solution is unclear, whereas Copilot Code Completion is preferred for more targeted coding tasks with clearer objectives. Our approach is particularly effective when users possess an intermediate level of clarity about their coding objectives. This typically occurs when users have a general understanding of the necessary functions, variables, or code structures, yet lack details on how to implement these elements. For example, a user may know they need to create a *"REST API with a GET handler to read from a database"*, but are uncertain about the specifics of database interaction. In such cases, users frequently reference code elements as they refine their ideas and align the emerging code with their overall architectural goals.

Figure 4: Our approach is more effective with prompts (a) that include more code-relevant phrases, allowing for the creation of more detailed code sketches, in contrast to prompts (b) that contain fewer such phrases.

Effectiveness Depends on Phrase-Code Correspondence. The effectiveness of our approach depends on the number of phrases that correspond to code elements. Intuitively, if a prompt contains more phrases that can be mapped to code elements, we can derive a more detailed code sketch. For example, Figure 4a contains two code-relevant phrases, while Figure 4b contains only one. Consequently, our approach can create a more detailed sketch for the first prompt. Nonetheless, even in the worst case—where no sketch can be derived for a prompt—our approach will simply default to a conventional LLM-based text-to-code system.

Limitations. Our approach is not without its limitations. As previously mentioned, the effectiveness of our approach depends on the number of phrases that are correspondent to code elements within the user's prompt. At present, our system supports only a minimal set of rules, which means that prompts that are ambiguous, unstructured, or overly generic may result in less precise code sketches. While it's theoretically possible to expand our rule set in the future, the practicality of such enhancements is yet to be determined, and scaling up may reveal additional challenges. Moreover, our current system is limited to English and Python only.

Future Work. The transformation from a natural language utterance to a linguistic dependency tree, and ultimately to an AST, is essentially a parsing process. We aim to streamline this by developing a compiler that would automate these steps and move beyond our current handcrafted, rule-based methodology, thereby increasing the system's robustness. By using code as an intermediate representation, the concept of *sketch-then-generate* could be expanded to other domains, such as graphic design. Furthermore, we envision that our approach could blur the distinctions between natural language and programming languages, opening up opportunities to apply features developed for programming languages to natural language prompts, such as static analysis, refactoring, and other enhancements. The latest updates will be posted here: https://pages.github.umn.edu/lang-sketch

REFERENCES

[1] Tyler Angert, Miroslav Suzara, Jenny Han, Christopher Pondoc, and Hariharan Subramonyam. 2023. Spellburst: A Node-based Interface for Exploratory Creative Coding with Natural Language Prompts. In *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology*. 1–22.

[2] Shraddha Barke, Michael B James, and Nadia Polikarpova. 2023. Grounded copilot: How programmers interact with code-generating models. *Proceedings of the ACM on Programming Languages* 7, OOPSLA1 (2023), 85–111.

[3] Yuzhe Cai, Shaoguang Mao, Wenshan Wu, Zehua Wang, Yaobo Liang, Tao Ge, Chenfei Wu, Wang You, Ting Song, Yan Xia, et al. 2023. Low-code LLM: Visual Programming over LLMs. *arXiv preprint arXiv:2304.08103* (2023).

[4] Yining Cao, Jane L. E, Chen Zhu-Tian, and Haijun Xia. 2023. DataParticles: Block-based and Language-oriented Authoring of Animated Unit Visualizations. In *CHI Conference on Human Factors in Computing Systems*. ACM, 808:1–808:15. https://doi.org/10.1145/3544548.3581472

[5] Veronique Donzeau-Gouge, Gerard Huet, Bernard Lang, and Gilles Kahn. 1980. *Programming environments based on structured editors: The MENTOR experience*. Ph. D. Dissertation. Inria.

[6] Explosion.Ai. 2024. SpaCy. https://spacy.io

[7] Marijn Haverbeke. 2024. CodeMirror. https://codemirror.net

[8] Aishwarya Kamath and Rajarshi Das. 2018. A survey on semantic parsing. *arXiv preprint arXiv:1812.00978* (2018).

[9] Majeed Kazemitabaar, Justin Chow, Carl Ka To Ma, Barbara J. Ericson, David Weintrop, and Tovi Grossman. 2023. Studying the effect of AI Code Generators on Supporting Novice Learners in Introductory Programming. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems* (<conf-loc>, <city>Hamburg</city>, <country>Germany</country>, </conf-loc>) *(CHI '23)*. Association for Computing Machinery, New York, NY, USA, Article 455, 23 pages. https://doi.org/10.1145/3544548.3580919

[10] Jenny T. Liang, Chenyang Yang, and Brad A. Myers. 2023. A Large-Scale Survey on the Usability of AI Programming Assistants: Successes and Challenges. arXiv:2303.17125 [cs.SE]

[11] Michael Xieyang Liu, Advait Sarkar, Carina Negreanu, Benjamin Zorn, Jack Williams, Neil Toronto, and Andrew D Gordon. 2023. "What It Wants Me To Say": Bridging the Abstraction Gap Between End-User Programmers and Code-Generating Large Language Models. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*. 1–31.

[12] Michael Xieyang Liu, Advait Sarkar, Carina Negreanu, Benjamin Zorn, Jack Williams, Neil Toronto, and Andrew D. Gordon. 2023. "What It Wants Me To Say": Bridging the Abstraction Gap Between End-User Programmers and Code-Generating Large Language Models. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems* (<conf-loc>, <city>Hamburg</city>, <country>Germany</country>, </conf-loc>) *(CHI '23)*. Association for Computing Machinery, New York, NY, USA, Article 598, 31 pages. https://doi.org/10.1145/3544548.3580817

[13] Meta. 2024. React. https://react.dev

[14] Microsoft. 2024. TypeScript. https://www.typescriptlang.org

[15] Hussein Mozannar, Gagan Bansal, Adam Fourney, and Eric Horvitz. 2022. Reading between the lines: Modeling user behavior and costs in AI-assisted programming. *arXiv preprint arXiv:2210.14306* (2022).

[16] Sydney Nguyen, Hannah McLean Babe, Yangtian Zi, Arjun Guha, Carolyn Jane Anderson, and Molly Q Feldman. 2024. How Beginning Programmers and Code LLMs (Mis) read Each Other. *arXiv preprint arXiv:2401.15232* (2024).

[17] Pallets. 2024. Flask. https://flask.palletsprojects.com

[18] Slav Petrov, Dipanjan Das, and Ryan McDonald. 2011. A universal part-of-speech tagset. *arXiv preprint arXiv:1104.2086* (2011).

[19] James Prather, Brent N. Reeves, Paul Denny, Brett A. Becker, Juho Leinonen, Andrew Luxton-Reilly, Garrett Powell, James Finnie-Ansley, and Eddie Antonio Santos. 2023. "It's Weird That it Knows What I Want": Usability and Interactions with Copilot for Novice Programmers. *ACM Trans. Comput.-Hum. Interact.* 31, 1, Article 4 (nov 2023), 31 pages. https://doi.org/10.1145/3617367

[20] Peng Qi, Timothy Dozat, Yuhao Zhang, and Christopher D Manning. 2019. Universal dependency parsing from scratch. *arXiv preprint arXiv:1901.10457* (2019).

[21] Steven I Ross, Fernando Martinez, Stephanie Houde, Michael Muller, and Justin D Weisz. 2023. The programmer's assistant: Conversational interaction with a large language model for software development. In *Proceedings of the 28th International Conference on Intelligent User Interfaces*. 491–514.

[22] Advait Sarkar, Andrew D Gordon, Carina Negreanu, Christian Poelitz, Sruti Srinivasa Ragavan, and Ben Zorn. 2022. What is it like to program with artificial intelligence? *arXiv preprint arXiv:2208.06213* (2022).

[23] Armando Solar-Lezama. 2013. Program sketching. *International Journal on Software Tools for Technology Transfer* 15 (2013), 475–495.

[24] Rhea Sukthanker, Soujanya Poria, Erik Cambria, and Ramkumar Thirunavukarasu. 2020. Anaphora and coreference resolution: A review. *Information Fusion* 59 (2020), 139–162.

[25] Yuan Tian, Toby Jia-Jun Li, Jonathan K Kummerfeld, and Tianyi Zhang. 2023. Interactive Text-to-SQL Generation via Editable Step-by-Step Explanations. *arXiv preprint arXiv:2305.07372* (2023).

[26] Tree-Sitter. 2024. Tree Sitter. https://tree-sitter.github.io/tree-sitter

[27] Priyan Vaithilingam, Tianyi Zhang, and Elena L Glassman. 2022. Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models. In *Chi conference on human factors in computing systems extended abstracts*. 1–7.

[28] Haijun Xia. 2020. Crosspower: Bridging Graphics and Linguistics. In *ACM Symposium on User Interface Software and Technology*. ACM, 722–734. https://doi.org/10.1145/3379337.3415845

[29] Frank F. Xu, Bogdan Vasilescu, and Graham Neubig. 2022. In-IDE Code Generation from Natural Language: Promise and Challenges. *ACM Trans. Softw. Eng. Methodol.* 31, 2, Article 29 (mar 2022), 47 pages. https://doi.org/10.1145/3487569

[30] Ryan Yen, Jiawen Zhu, Sangho Suh, Haijun Xia, and Jian Zhao. 2023. CoLadder: Supporting Programmers with Hierarchical Code Generation in Multi-Level Abstraction. *arXiv preprint arXiv:2310.08699* (2023).

[31] 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)*. 7443–7464.

[32] Ziyin Zhang, Chaoyu Chen, Bingchang Liu, Cong Liao, Zi Gong, Hang Yu, Jianguo Li, and Rui Wang. 2024. Unifying the Perspectives of NLP and Software Engineering: A Survey on Language Models for Code. arXiv:2311.07989 [cs.CL]

[33] Chen Zhu-Tian and Haijun Xia. 2022. CrossData: Leveraging Text-Data Connections for Authoring Data Documents. In *CHI Conference on Human Factors in Computing Systems*. ACM, 95:1–95:15. https://doi.org/10.1145/3491102.3517485

[34] Chen Zhu-Tian, Qisen Yang, Xiao Xie, Johanna Beyer, Haijun Xia, Yingcai Wu, and Hanspeter Pfister. 2023. Sporthesia: Augmenting Sports Videos Using Natural Language. *IEEE Trans. Vis. Comput. Graph.* 29, 1 (2023), 918–928. https://doi.org/10.1109/TVCG.2022.3209497