Explaining Code with a Purpose: An Integrated Approach for Developing Code Comprehension and Prompting Skills

Paul Denny
University of Auckland
Auckland, New Zealand
paul@cs.auckland.ac.nz

David H. Smith IV
University of Illinois
Urbana, IL, USA
dhsmith2@illinois.edu

Max Fowler
University of Illinois
Urbana, IL, USA
mfowler5@illinois.edu

James Prather
Abilene Christian University
Abilene, TX, USA
james.prather@acu.edu

Brett A. Becker
University College Dublin
Dublin, Ireland
brett.becker@ucd.ie

Juho Leinonen
Aalto University
Espoo, Finland
juho.2.leinonen@aalto.fi

ABSTRACT

Reading, understanding and explaining code have traditionally been important skills for novices learning programming. As large language models (LLMs) become prevalent, these foundational skills are more important than ever given the increasing need to understand and evaluate model-generated code. Brand new skills are also needed, such as the ability to formulate clear prompts that can elicit intended code from an LLM. Thus, there is great interest in integrating pedagogical approaches for the development of both traditional coding competencies and the novel skills required to interact with LLMs. One effective way to develop and assess code comprehension ability is with "Explain in plain English" (EiPE) questions, where students succinctly explain the purpose of a fragment of code. However, grading EiPE questions has always been difficult given the subjective nature of evaluating written explanations and this has stifled their uptake. In this paper, we explore a natural synergy between EiPE questions and code-generating LLMs to overcome this limitation. We propose using an LLM to generate code based on students' responses to EiPE questions – not only enabling EiPE responses to be assessed automatically, but helping students develop essential code comprehension and prompt crafting skills in parallel. We investigate this idea in an introductory programming course and report student success in creating effective prompts for solving EiPE questions. We also examine student perceptions of this activity and how it influences their views on the use of LLMs for aiding and assessing learning.

CCS CONCEPTS

• **Social and professional topics → Computing education**; • **Computing methodologies → Artificial intelligence**.

KEYWORDS

Explain in plan English, EiPE, Large language models, LLMs, Code comprehension, Prompting, Introductory programming, CS1

ITiCSE 2024, July 8–10, 2024, Milan, Italy
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0600-4/24/07.
https://doi.org/10.1145/3649217.3653587

ACM Reference Format:
Paul Denny, David H. Smith IV, Max Fowler, James Prather, Brett A. Becker, and Juho Leinonen. 2024. Explaining Code with a Purpose: An Integrated Approach for Developing Code Comprehension and Prompting Skills. In *Proceedings of the 2024 Innovation and Technology in Computer Science Education V. 1 (ITiCSE 2024), July 8–10, 2024, Milan, Italy.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3649217.3653587

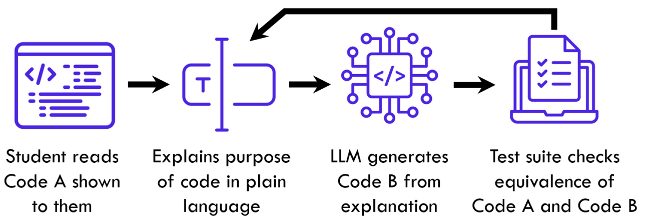

Student reads Code A shown to them
Explains purpose of code in plain language
LLM generates Code B from explanation
Test suite checks equivalence of Code A and Code B

Figure 1: Explaining the purpose of a code fragment to an LLM until it generates functionally equivalent code, targeting code comprehension and prompt crafting skills in parallel.

1 INTRODUCTION

The ability to read and comprehend source code is an essential skill for all programmers, from novices to professionals [26]. Indeed, due to the ease of generating code with large language models (LLMs), programmers will be spending an increasing proportion of their time understanding and evaluating LLM-generated code [3]. Of course, teaching students to comprehend code is not a new challenge; reading and tracing code execution and answering comprehension questions about code have been common and effective strategies in programming courses for many years [6, 19, 34, 35]. In particular, EiPE ("Explain in Plain English") questions are a widely studied format for assessing how well students can read and understand code at an abstract level [9, 28].

Manual grading approaches for EiPE questions have been informed by the "Structure of the Observed Learning Outcome" (SOLO) taxonomy where a given response can be categorized based on the degree to which it integrates all elements of a given segment of code in order to describe that code's purpose rather than its implementation [4, 23]. As a response relating the various elements of a topic together is treated as one of the highest levels of comprehension in the SOLO taxonomy, a typical EiPE question asks students to describe the *purpose* of a provided piece of code in natural language [27]. Through interviews with members of the computing education research community, Fowler et al. [17] found that EiPE questions were valued for their abstraction aspect which was seen to aid in debugging, communication and functional decomposition. Despite this clear potential, the difficulty of grading EiPE questions – due to the subjective nature of evaluating students' written explanations – has been a longstanding barrier to their wide-scale adoption [16, 22, 36]. To address this shortcoming, Smith IV and Zilles [33] created an EiPE grading approach that uses LLM generated code from a student's EiPE response to determine if their description is accurate.

In this paper, we explore this potential synergy between EiPE questions and code-generating LLMs. Illustrated in Figure 1, our method begins by presenting a student with a fragment of code, much like in a traditional EiPE question. The student then reads and attempts to understand what the code is doing, and crafts an explanation of the code in natural language. To assess this explanation, we use the method proposed by Smith IV and Zilles [33] where the student's response is provided as the input prompt to a code-generating LLM. The code that is generated is automatically tested for equivalence with the original code using a test suite. This approach enables the objective evaluation of responses to EiPE questions and supports the development of both code comprehension skills and skills related to clearly formulating prompts for LLMs.

To investigate the potential of this idea, we deployed a series of these problems to students in a large introductory programming course (n≈900). First, we examined how well students solved these tasks, by analysing success rates and prompt lengths (with and without a character limit enforced) and we classified prompts with respect to the SOLO taxonomy. Second, we investigated students' perceptions of this activity which was their first experience using an LLM to assess code comprehension skill. Students compared the activity to more familiar programming tasks, and indicated the extent to which they felt it was a valid way to evaluate their learning. We organise our study, and the presentation of results, around the following three research questions:

RQ1: How successful are students at crafting code explanation prompts that induce an LLM to generate functionally equivalent code?

RQ2: What is the relationship, if any, between the success of a student's prompt and the classification of that prompt with respect to the categories of the SOLO taxonomy?

RQ3: What are students' thoughts about the code explanation activity in comparison to more traditional code writing tasks, and do they see it as an accurate way to assess their comprehension of code?

2 RELATED WORK

2.1 Large Language Models in Introductory Programming Courses

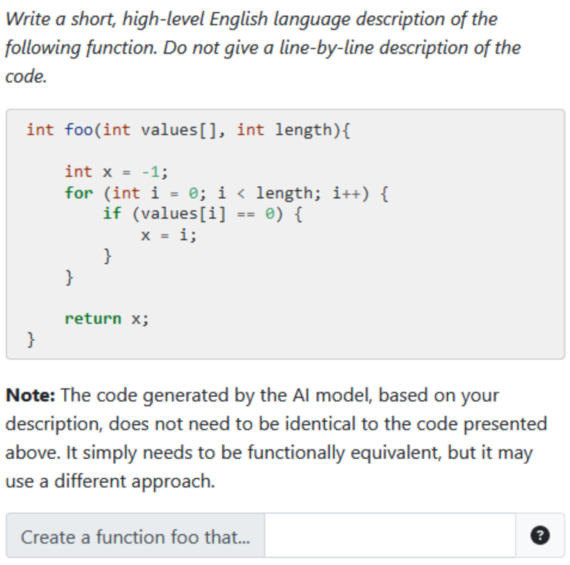
The advent of large language models (LLMs) has introduced significant changes to the landscape of programming education [13]. Numerous open-source and proprietary models are being developed and evaluated [21, 31], and tools like GitHub Copilot have been seamlessly integrated into widely-used Integrated Development Environments (IDEs). This has begun to revolutionize the code-writing process for both seasoned developers and new learners. From a computing education perspective, the ubiquity and capability of LLMs has raised questions regarding academic integrity and how and when such tools should be integrated into CS1 courses [11, 24, 29, 30].

Studies investigating the capabilities of these models, namely Codex, GPT-3, and GPT-4, have shown that they can solve typical programming problems at least as well as an average introductory programming student [8, 14, 15]. Recent work by Denny et al. evaluated the ability of the Codex model to solve introductory programming exercises from natural language specifications [10]. Of the 166 problems they evaluated, approximately 50% could be solved by simply supplying the original question prompt, and 80% could be solved after small manual modifications were made to the prompts. Though this may raise concerns related to the ease with which students are able to generate solutions, it also highlights how a human can work iteratively with an LLM to refine a prompt to generate desired code. This suggests that explicit instruction on how to formulate effective prompts, supported by appropriate practice opportunities, could be beneficial for students learning programming.

2.2 Explain in Plain English Questions & Prompt Problems

The issue of problem formulation when prompting for code generation bears some resemblance to "Explain in Plain English" (EiPE) questions. Both require students to formulate a description of some code. In the case of EiPE questions, "correctness" of a response is generally evaluated based on whether it unambiguously conveys the functionality of the code at a high-level [2, 18]. This approach to grading has been largely inspired by the SOLO taxonomy as it has been applied to code comprehension. This differentiates between a student describing the structures present in a segment of code from it's higher level purpose. The latter is considered to demonstrate a higher level of comprehension and is more typical of descriptions produced by experts [23]. On the other hand, successful prompting might be characterized by the ability to provide a description that elicits code that functions as the prompter intended.

Beyond the relationship between EiPE questions and prompting there is an emerging notion that prompting, evaluating the resulting code, and then potentially re-prompting is an emerging skill set which CS students should be explicitly taught [13]. Early work in this direction includes the research around "Prompt problems" [11, 12], where students are shown images that illustrate how inputs should be transformed to outputs and are tasked with constructing solutions in natural language. Similarly, Smith IV and Zilles [33] evaluated a grading approach for EiPE questions which they term "Code Generation Based Grading" (CGBG). This involved collecting students' responses to EiPE questions taken from a large historical dataset, and grading them based on the correctness of the code they produce when used as a prompt to an LLM. They note that future work should investigate the utility of this grading approach as a tool for teaching prompting skills, given that effectively utilizing the system's feedback requires students to evaluate the relationship


Write a short, high-level English language description of the following function. Do not give a line-by-line description of the code.

```
int foo(int values[], int length){

    int x = -1;
    for (int i = 0; i < length; i++) {
        if (values[i] == 0) {
            x = i;
        }
    }

    return x;
}
```

Note: The code generated by the AI model, based on your description, does not need to be identical to the code presented above. It simply needs to be functionally equivalent, but it may use a different approach.

Create a function foo that...

Figure 2: A problem presented in the form of a function implementation with obfuscated identifier names; students enter a natural language description that will be used to prompt an LLM to generate equivalent code to that shown.

between their prompt, the code it produced, and the code they were attempting to describe.

3 METHODS

We conducted our study in an introductory programming course at the University of Auckland, a large public research University in New Zealand. The 12-week course is required for all students in the Engineering programme, and introduces standard CS1 topics using a combination of MATLAB and C. In the spring term of 2023, when our data was collected, 889 students were enrolled.

3.1 Code Explanation Tasks

Throughout the course, students complete weekly laboratory exercises primarily consisting of sets of programming tasks that are automatically graded. For the purposes of the current study, we included the new code explanation tasks in two lab sessions. For convenience in the paper, we will refer to these lab sessions as Lab A and Lab B, although they were held in Week 2 and Week 4 of the C programming module respectively. The topic of Lab A was "Loops, Arrays and Functions" and the topic of Lab B was "Strings, Text Processing and 2D arrays". The new tasks were delivered using a modified version of the open-source PrairieLearn platform [37]. Alongside regular programming tasks, both Lab A and Lab B included four new code explanation tasks. Every task consisted of a single function (named 'foo') that had to be explained – Lab A did not enforce any length limit on the explanations that students wrote, whereas for Lab B a 250-character limit was imposed.

A short description of each task is listed in Table 2 when presenting the results (e.g. "Index of last zero"). Figure 2 shows a screenshot of the PrairieLearn platform with one of the code explanation tasks from Lab A. The task shown is the "Index of last zero" task, where the code returns the index position of the rightmost occurrence of the value 0, or -1 if it is not present. Students complete the prompt, which begins *"Create a function foo that..."*, and submit it for grading. Students were also given the following information about the task:

"The goal of this task is for you to read and understand the purpose of the code shown below. Describe the purpose of the code using plain language; your description will be given to an AI language model, and the model will generate code matching your description. You will have solved the task when the code generated by the AI model is functionally equivalent to the code you have described." Upon submitting a prompt, the code generated by the LLM is displayed along with the results of the test cases.

The tasks were graded, but contributed only a small fraction towards each student's final score (approximately 1%). No penalties were given for incorrect submissions, but for each task a maximum of 20 attempts were allowed.

3.2 Analysis of Students' Prompts

In addressing **RQ1** and **RQ2** we operationalize success at completing the prompting task as the ability to provide a prompt that (1) is successful in generating code that passes the provided test cases and (2) demonstrates comprehension of the code's purpose. To evaluate the latter, we use the "Structure of the Observed Learning Outcome" (SOLO) taxonomy [4]. Specifically, we use the adapted SOLO taxonomy presented by Lister et al. [23] as well as their process for applying these codes to student's responses. The taxonomy and definitions we use when coding students' prompts are:

- **Prestructural:** A student demonstrates one or more significant misconceptions or no understanding of the code.
- **Unistructural:** A student demonstrates some understanding of the code or the code's purpose but provides an incomplete description or the description contains some misconceptions.
- **Multistructural** The student provides a correct and complete description of the code and its structures but does not fully join these descriptions together to describe the code's overall purpose.
- **Relational:** The student demonstrates a correct and high-level understanding of the code's purpose by relating all of its elements together and describing the code's purpose.

We also include a fifth category which we term **Direct Recitation**. This category includes responses where a student directly recited the code in a line-by-line fashion or directly copied the code or elements of the code verbatim into the prompt without demonstrating any understanding of the code's structures.

Student prompts were categorized by two members of the research team. From each of the eight questions, 200 prompts were randomly selected for deductive coding using the categories described above. In the event that a student's prompt contained a *multistructural* description in addition to a *relational* summary, that response was graded as *relational*. For the purposes of establishing inter-rater reliability (IRR) the response set from two questions were coded independently by each of the researchers. Inter-rater reliability was calculated using Cohen's kappa and found to be 0.79, well above the accepted threshold for high IRR [25]. The researchers then met to reconcile those disagreements that did exist. Given the high IRR, the researchers then each coded 100 prompts from each

Type	Question
Open (prompted)	Please reflect on the code comprehension tasks and comment on what you think about them compared to typical programming tasks.
Likert (SD,D,N,A,SA)	Having AI language models generate code from natural language descriptions is an accurate way to evaluate code comprehension skills.
Open (unprompted)	Do you have any comments about this lab?

Table 1: Reflection questions related to RQ3.

of the remaining six questions independently for a total of 1600 prompts.

3.3 Student Perceptions

To address **RQ3**, after completing (or attempting) the code explanation tasks students were asked to reflect on the activity by responding to three questions. Two of these were open-response questions, and the other used a standard 5-point Likert scale. The first open-response question directly asked students to comment on the code explanation task, and thus we consider this 'prompted' feedback. The other open-response question was a generic question about any aspect of the lab, and thus we consider any comments relating to the code explanation task in response to this question to be 'unprompted'. The three questions are listed in Table 1. We summarize responses to the Likert item using a diverging stacked bar chart. We analyzed open-response data using the guidelines for reflexive thematic analysis outlined by Braun and Clarke [5]. This includes a coding phase in which responses are tagged with succinct labels, followed by phases of generating and developing higher-level themes collated from these labels[1]. When presenting the results in Section 4.3, we report the most common themes and illustrate these with examples of student responses.

4 RESULTS

4.1 RQ1: Task Completion Success

Overall, students experienced a high degree of success in both Lab A and Lab B, with almost all students completing all tasks (Table 2). Despite being given 20 attempts per task, the majority of students completed each of the tasks using only one or two attempts.

Lab B differed from Lab A in that it imposed a 255 character limit on students' prompts. Examining the distribution of prompt lengths for each of the two labs reveals that the median response length for each of the two labs was similar (Figure 3). The primary impact of including the character limit appears to be that it eliminated the small number of extremely verbose prompts seen in Lab A. The inclusion of the character limit does not appear to have had a significant impact on students' success at the task as the average number of attempts for each of the labs is quite similar (Table 2).

4.2 RQ2: SOLO Category & Prompt Success

From Figure 4, we see that prompts that contained small or significant misconceptions (*Unistructural* and *Prestructural*) generated code which was near universally graded as incorrect. Though this might be expected for prestructural it is positive that the LLM, in

[1]https://www.thematicanalysis.net/doing-reflexive-ta/

#	Task Description	μ	σ	% Correct
Lab A	Sum between a and b inclusive	2.04	2.53	98.2
	Count even numbers in array	1.37	1.00	99.0
	Index of last zero	2.18	3.44	99.8
	Sum positive values	1.43	0.75	99.2
Lab B	Reverse a string	1.65	1.56	99.6
	Calculate sum of row in 2D array	2.03	3.11	98.1
	Is a vowel contained in a string?	1.49	1.89	98.1
	Does a string contain a substring?	2.58	6.43	96.3

Table 2: The number of submission attempts and percentage of students who successfully completed each task.

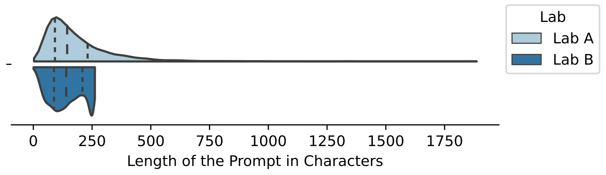


Figure 3: Distribution of prompt lengths for each of the two lab sessions (Lab B enforced a 255 character limit).

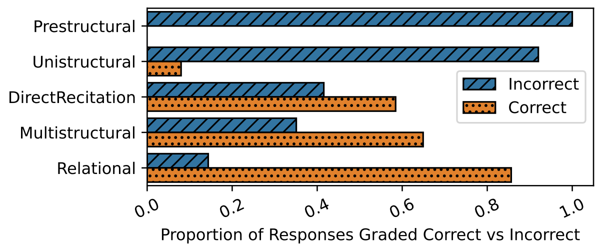


Figure 4: Proportion of prompts generating correct and incorrect code at each SOLO level.

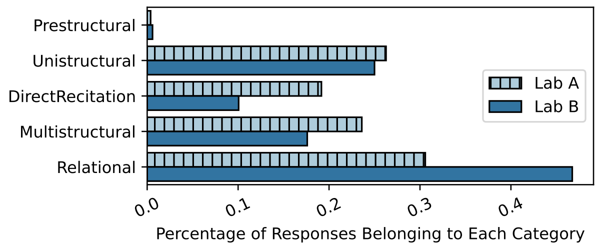


Figure 5: Proportion of prompts classified at each SOLO-level for each lab.

general, did not generate correct code for a prompt that contained a misconception. If it had done so, then this approach to grading EiPE questions could reinforce misconceptions.

Prompts classified as *Direct Recitation* and *Multistructural* more often generated code which was graded as correct rather than incorrect, but not overwhelmingly so. It may be the case that such prompts can contain certain ambiguities or lack specific details which human graders are willing to overlook in the event the prompt contains the core ideas the grader is looking for. However, these ambiguities may lead an LLM to generating plausible though ultimately incorrect interpretations of a student's prompt.

Finally, *Relational* prompts generated code which was overwhelmingly graded as correct. This finding is encouraging as it indicates a synergy between successful prompting, which focuses on successfully generating code, and high quality EiPE responses, which focus on high-level descriptions of code.

In comparing the results for Lab A and Lab B, we found that Lab B included a higher proportion of relational responses than Lab A (Figure 5). It may be the case that the character limit imposed for Lab B not only reduced the presence of extremely long explanations but increased the presence of relational ones as well. However, there are alternative explanations. Students' code comprehension skills may have improved between the two labs, and students' may have found relational responses more successful in Lab A and thus deliberately aimed to construct them in Lab B. In any case, given the success of relational prompts for generating code, future work seeking to teach students how to successfully prompt LLMs should further explore approaches to encouraging relational prompts.

4.3 RQ3: Student Perceptions

A total of 812 non-empty responses were submitted to the first open-response question, where students were directly 'prompted' on their perceptions of the code explanation exercises in comparison to traditional code writing tasks. Thematic analysis of these responses revealed several key themes. Generally speaking, students enjoyed the exercises and found them interesting, and took a positive view of their educational value.

4.3.1 Novelty and Engagement. The most prominent theme that emerged related to the novelty and engaging nature of the activity. Students frequently expressed that the task was *"very interesting and fun as it's nice seeing AI included"* and that it provides a *"nice change of pace from writing code"*. This sentiment is reflected clearly in the following responses:

> *"I actually found it a little bit entertaining since it was unbelievable to me that a code could be written directly from a sentence or two."*

> *"The task was a lot of fun to do and it was interesting to see how the ai understood my work based on my statements."*

Many similar comments were observed, indicating that students generally enjoyed the activity, found it novel, and appreciated that some aspects of AI were integrated into the course.

4.3.2 Enhanced Comprehension of Code. The next most prevalent theme related to the perception that the activity enhanced students' comprehension of code. Students often remarked on the usefulness of the exercise to articulate their understanding in *"plain English,"* which helped *"improve my skills of understanding code"*:

> *"I think it is very useful in really comprehending chunks of code in terms of their purpose rather than its individual tasks."*

> *"It actually gets people to read and understand code, and try to figure out what the original code was supposed to do. I feel that my ability to describe the actual action not just the steps going on improved as I read the code more."*

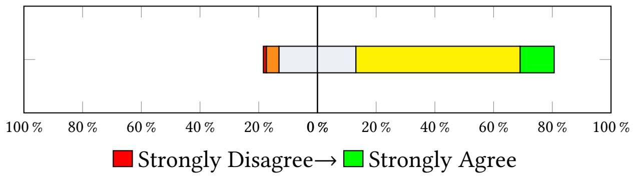


Figure 6: Student perceptions of whether the task is an accurate way to evaluate code comprehension skills.

This theme suggests that translating code to natural language prompts encouraged deeper cognitive processing of programming concepts and improved understanding, highlighting the educational value of the task. Moreover, student responses to the Likert item (summarized in Figure 6) illustrated generally strong agreement that using an LLM to generate code from their explanations is an accurate way to evaluate code comprehension skills.

4.3.3 Concerns About Effectiveness for Practice and Assessment. Not all views about the task were positive. One relatively common theme exposed concerns regarding the effectiveness of the task, especially how well it would scale to problems where the code is more complex. In addition, some mostly positive comments were qualified by skeptical statements about the value of the task in comparison to more traditional activities, or at least a desire to see a balance between the use of AI-supported and traditional tasks:

> *"Pretty cool. The code was simplistic enough for an AI to generate a valid result - but I don't think this would work in more complex problems."*

> *"The task was fairly helpful in understanding what code means. However, I feel like it only has limited use in helping me to write code more easily and efficiently."*

> *"So, I think I mixture of the AI model question and regular coding question will be good, but more regular coding question than the other one."*

4.3.4 General feedback. The second of the two open-response questions, which asked for general feedback on any aspect of the lab, elicited a total of 80 'unprompted' responses that made some mention of the code explanation activity. The vast majority of these responses were positive, and most related to a general theme around "Enjoyment or fun", aligning with the most common theme observed in responses to the 'prompted' question around the tasks. Many students highlighted the novelty of the activity. For instance, one student reflected, *"The lab was super fun, especially PrairieLearn as it was a completely new experience"*. Others examples included *"I just want to say I loved the PrairieLearn tasks"* and *"The PraireLearn part was also very fun to do and use"*.

Another positive theme that was common related to "Learning and Understanding", where students appreciated that the task aided their comprehension of code. One student commented, *"PrairieLearn gave me much better insight into the connection between natural language and coding, as small changes in wording had a large impact on the code"*. Others noted the benefits for developing new skills, with one even referring to the skill of 'writing code in plain language': *"I would've liked to use prairielearn a bit more and continue to improve my skills of writing code in plain language"*.

Negative feedback was much less common, and included comments around technical issues and interface design such as wanting

Task description	Longest correct response	Shortest correct response
Count even numbers in array	takes an array of values called 'values[]' and the length (or number of values contained within the array) of the array. The program then uses a for loop with the conditionals i initialized to equal zero, looping the for loop for as long as i is less than the length of the array and adding 1 (using i++) every iteration. This is essentially scans the length of the array of values and operates on each item in the array. The operation for each item in the array (i.e. values[i], the operation on the number at any given index) is to test wether the value leaves any remainder when divided by two (i.e. using mod 2). If the number leaves no remainder when divided by two it is even and the count for x is incremented by one. This continues until every value in the array has been tested. The returned value 'x' is the count of how many numbers left no remainder when divided by two, which is a count of how many even numbers were in the array of values.	find how many numbers divisible by 2
Reverse a string	The provided code is a C function named ‘foo‘ that reverses the characters in a given character array (string) ‘str‘. Here's a breakdown of how the code works: 1. ‘#include <string.h>‘: This line includes the standard C library header file ‘string.h‘, wh	flips a string
Does a string contain a substring	The code defines a C function named foo that checks if the second string str2 is contained within the first string str1. It iterates through str1 and compares substrings of the same length as str2. If a match is found, it returns 1; otherwise, it returns	checks if str2 is a in str1

Table 3: The longest and shortest correct prompts submitted for a selection of the exercises (the last two, from Lab B, had a 255 character limit enforced). Some prompts show evidence of being generated by LLMs, given tell-tale language such as "Here's a breakdown of what happens" and the start of a list of line-by-line explanations (e.g., the longest response in the middle row).

to continue experimenting with the tool even after successfully solving a problem in order to *"push the ai"*, as well as a small number of students reporting either the tasks were too easy or too difficult.

5 DISCUSSION

When the capabilities of large language models (LLMs) became apparent several years ago, there was initially widespread concern in academia regarding their potential misuse by students. Now, as their presence becomes ubiquitous, there are calls to explore the integration of LLMs into teaching practice and to use them to power novel educational tools [13]. This is especially relevant in the field of computing education, where the dominant pedagogy has involved frequent and repeated practice at *writing* code [1]. The ease with which LLMs can now be used to generate code from natural language prompts suggests a need to re-evaluate teaching strategies that focus on the mechanics of syntax and code writing [15, 32]. Indeed, a recent global survey of computing educators revealed a strong expectation that students will need to be taught how to use generative AI tools, and yet concrete pedagogical approaches are only just beginning to emerge [29].

Though these exercises may not be considered traditional EiPE questions given that they lack explicit checks for many of the requirements present in those rubrics [7, 17], they do represent a similar form of code comprehension task. From our results we see there appears to exist overlap between what makes a successful EiPE response and prompts which successfully generate code. This is echoed in the qualitative results where students not only found the tasks engaging but often mentioned being engaged in the process of reading code and attempting to comprehend its purpose. These findings are promising in that they suggest the task was successful in engaging students with the process of code comprehension. Furthermore, their success at the tasks appears to be related to the level at which they were able to express that comprehension.

As the name suggests, an implicit requirement for traditional "Explain in Plain English" questions is that the descriptions be provided in English. This could place students with poorer English language skills – but equally good code comprehension skills – at a disadvantage. Indeed, this is the motivation for "refute" questions, proposed by Kumar and Raman [20], which allow students to demonstrate their comprehension of code but without requiring English language skill. The powerful language translation capabilities of modern LLMs means that code explanations can be provided in a wide variety of languages. We observed several students submitting accurate descriptions of code in languages other than English. For example, a total of six submissions that successfully solved the tasks were made in Chinese. Although still fairly infrequent in our data, this suggests that the current approach could be used to provide an equitable assessment option in diverse classrooms, and this would be an interesting direction for future work.

Supported by the initially positive findings of this study, there exist several avenues for future work. First, grading is not entirely reliable as the same prompt may produce correct code on some occasions and incorrect code on others. A modified version of the activity could generate multiple completions from the prompt, to assess how reliable the prompt is. In addition, a poor "Explain in Plain English" prompt, like a line by line explanation, might still produce the correct code thus reinforcing that behavior. Given both the success of relational prompts in this activity and their alignment with the goals of teaching students code comprehension, future work should focus on additional measures which can be taken to nudge students towards providing relational prompts.

6 CONCLUSION

As the computing education community continues to grapple with questions around the integration of large language models (LLMs) into the classroom, in this work we offer some insights into their potential role for developing both code comprehension and prompting skills. We propose an approach where code comprehension is assessed through the use of "Explain in Plain English" (EiPE) questions, by passing student explanations of code to an LLM for evaluation. In an empirical study in a large introductory classroom, we observe high rates of success for students attempting these kinds of tasks, and find that higher-level, relational descriptions of code are much more likely to succeed. Feedback from students indicates that they felt the activity not only helps in improving understanding of code, but was also novel and highly engaging. Our work demonstrates just one possible way that LLMs could be integrated into programming classrooms, and highlights the need for continued work in this direction.

ACKNOWLEDGMENTS

This research was partially supported by the Research Council of Finland (Academy Research Fellow grant number 356114).

REFERENCES

[1] Joe Michael Allen, Frank Vahid, Alex Edgcomb, Kelly Downey, and Kris Miller. 2019. An Analysis of Using Many Small Programs in CS1 *(SIGCSE '19)*. ACM, NY, NY, USA, 585–591. https://doi.org/10.1145/3287324.3287466

[2] Sushmita Azad, Binglin Chen, Maxwell Fowler, Matthew West, and Craig Zilles. 2020. Strategies for deploying unreliable AI graders in high-transparency high-stakes exams. In *Artificial Intelligence in Education: 21st International Conference, AIED 2020, Ifrane, Morocco, July 6–10, 2020, Proceedings, Part I 21*. Springer, 16–28.

[3] Brett A. Becker, Paul Denny, James Finnie-Ansley, Andrew Luxton-Reilly, James Prather, and Eddie Antonio Santos. 2023. Programming Is Hard - Or at Least It Used to Be: Educational Opportunities and Challenges of AI Code Generation. In *Proc of the 54th ACM Tech Symp on CS Education V. 1* (Toronto ON, Canada) *(SIGCSE 2023)*. ACM, NY, USA, 500–506. https://doi.org/10.1145/3545945.3569759

[4] John B Biggs and Kevin F Collis. 2014. *Evaluating the quality of learning: The SOLO taxonomy (Structure of the Observed Learning Outcome)*. Academic Press.

[5] Virginia Braun and Victoria Clarke. 2006. Using thematic analysis in psychology. *Qualitative Research in Psychology* 3, 2 (2006), 77–101. https://doi.org/10.1191/1478088706qp063oa

[6] Teresa Busjahn, Carsten Schulte, and Andreas Busjahn. 2011. Analysis of Code Reading to Gain More Insight in Program Comprehension. In *Proc of the 11th Koli Calling Int. Conf. on Computing Education Research* (Koli, Finland) *(Koli Calling '11)*. ACM, New York, NY, USA, 1–9. https://doi.org/10.1145/2094131.2094133

[7] Binglin Chen, Sushmita Azad, Rajarshi Haldar, Matthew West, and Craig Zilles. 2020. A validated scoring rubric for explain-in-plain-english questions. In *Proc of the 51st ACM Technical Symposium on CS Education*. 563–569.

[8] Bruno Pereira Cipriano and Pedro Alves. 2023. Gpt-3 vs object oriented programming assignments: An experience report. In *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1*. 61–67.

[9] Malcolm Corney, Sue Fitzgerald, Brian Hanks, Raymond Lister, Renee McCauley, and Laurie Murphy. 2014. Explain in Plain English Questions Revisited: Data Structures Problems. In *Proceedings of the 45th ACM Technical Symposium on Computer Science Education* (Atlanta, Georgia, USA) *(SIGCSE '14)*. ACM, NY, USA, 591–596. https://doi.org/10.1145/2538862.2538911

[10] Paul Denny, Viraj Kumar, and Nasser Giacaman. 2023. Conversing with Copilot: Exploring Prompt Engineering for Solving CS1 Problems Using Natural Language. In *Procof the 54th ACM Technical Symposium on CS Education V. 1* (Toronto, Canada) *(SIGCSE 2023)*. ACM, NY, USA, 1136–1142. https://doi.org/10.1145/3545945.3569823

[11] Paul Denny, Juho Leinonen, James Prather, Andrew Luxton-Reilly, Thezyrie Amarouche, Brett A. Becker, and Brent N. Reeves. 2023. Promptly: Using Prompt Problems to Teach Learners How to Effectively Utilize AI Code Generators. arXiv:2307.16364 [cs.HC]

[12] Paul Denny, Juho Leinonen, James Prather, Andrew Luxton-Reilly, Thezyrie Amarouche, Brett A. Becker, and Brent N. Reeves. 2024. Prompt Problems: A New Programming Exercise for the Generative AI Era. In *Proc of the 55th ACM Technical Symposium on CS Education V. 1* (Portland, USA) *(SIGCSE 2024)*. ACM, NY, USA, 296–302. https://doi.org/10.1145/3626252.3630909

[13] Paul Denny, James Prather, Brett A. Becker, James Finnie-Ansley, Arto Hellas, Juho Leinonen, Andrew Luxton-Reilly, Brent N. Reeves, Eddie Antonio Santos, and Sami Sarsa. 2024. Computing Education in the Era of Generative AI. *Commun. ACM* 67, 2 (Jan 2024), 56–67. https://doi.org/10.1145/3624720

[14] James Finnie-Ansley, Paul Denny, Brett A. Becker, Andrew Luxton-Reilly, and James Prather. 2022. The Robots Are Coming: Exploring the Implications of OpenAI Codex on Introductory Programming. In *Proc of the 24th Australasian Comp Ed Conference* (Virtual Event, Australia) *(ACE '22)*. ACM, New York, NY, USA, 10–19. https://doi.org/10.1145/3511861.3511863

[15] James Finnie-Ansley, Paul Denny, Andrew Luxton-Reilly, Eddie Antonio Santos, James Prather, and Brett A. Becker. 2023. My AI Wants to Know If This Will Be on the Exam: Testing OpenAI's Codex on CS2 Programming Exercises *(ACE '23)*. ACM, NY, NY, USA, 97–104. https://doi.org/10.1145/3576123.3576134

[16] Max Fowler, Binglin Chen, Sushmita Azad, Matthew West, and Craig Zilles. 2021. Autograding "Explain in Plain English" Questions Using NLP. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education* (Virtual Event, USA) *(SIGCSE '21)*. Association for Computing Machinery, New York, NY, USA, 1163–1169. https://doi.org/10.1145/3408877.3432539

[17] Max Fowler, Binglin Chen, and Craig Zilles. 2021. How Should We 'Explain in Plain English'? Voices from the Community. In *Proceedings of the 17th ACM Conference on International Computing Education Research* (Virtual Event, USA) *(ICER 2021)*. Association for Computing Machinery, New York, NY, USA, 69–80. https://doi.org/10.1145/3446871.3469738

[18] Max Fowler, Binglin Chen, and Craig Zilles. 2021. How should we 'Explain in plain English'? Voices from the Community. In *Proceedings of the 17th ACM conference on international computing education research*. 69–80.

[19] Mohammed Hassan and Craig Zilles. 2021. Exploring 'reverse-Tracing' Questions as a Means of Assessing the Tracing Skill on Computer-Based CS 1 Exams. In *Proc. of the 17th ACM Conf. on Int. Comp. Ed. Research* (Virtual Event, USA) *(ICER 2021)*. ACM, New York, NY, USA, 115–126. https://doi.org/10.1145/3446871.3469765

[20] Viraj Kumar and Arun Raman. 2023. Helping Students Develop a Critical Eye with Refute Questions. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 2* (Toronto ON, Canada) *(SIGCSE 2023)*. ACM, New York, NY, USA, 1181. https://doi.org/10.1145/3545947.3569636

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

[22] Tiffany Wenting Li, Silas Hsu, Max Fowler, Zhilin Zhang, Craig Zilles, and Karrie Karahalios. 2023. Am I Wrong, or Is the Autograder Wrong? Effects of AI Grading Mistakes on Learning. In *Proc. of the 2023 ACM Conference on International Computing Education Research - Volume 1* (Chicago, IL, USA) *(ICER '23)*. ACM, New York, NY, USA, 159–176. https://doi.org/10.1145/3568813.3600124

[23] Raymond Lister, Beth Simon, Errol Thompson, Jacqueline L Whalley, and Christine Prasad. 2006. Not seeing the forest for the trees: novice programmers and the SOLO taxonomy. *ACM SIGCSE Bulletin* 38, 3 (2006), 118–122.

[24] Stephen MacNeil, Joanne Kim, Juho Leinonen, Paul Denny, Seth Bernstein, Brett A Becker, Michel Wermelinger, Arto Hellas, Andrew Tran, Sami Sarsa, et al. 2023. The Implications of Large Language Models for CS Teachers and Students. In *Proc. of the 54th ACM Technical Symposium on Computer Science Education*, Vol. 2.

[25] Mary L McHugh. 2012. Interrater reliability: the kappa statistic. *Biochemia medica* 22, 3 (2012), 276–282.

[26] Maria Medvidova and Jaroslav Porubän. 2022. Program Comprehension and Quality Experiments in Programming Education. In *Third International Computer Programming Education Conference (ICPEC 2022)*, Vol. 102. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 14:1–14:12. https://doi.org/10.4230/OASIcs.ICPEC.2022.14

[27] Laurie Murphy, Sue Fitzgerald, Raymond Lister, and Renée McCauley. 2012. Ability to 'explain in Plain English' Linked to Proficiency in Computer-Based Programming. In *Proc. of the 9th Annual International Conference on International Computing Education Research* (Auckland, New Zealand) *(ICER '12)*. ACM, New York, NY, USA, 111–118. https://doi.org/10.1145/2361276.2361299

[28] Laurie Murphy, Renée McCauley, and Sue Fitzgerald. 2012. 'Explain in Plain English' Questions: Implications for Teaching. In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education* (Raleigh, North Carolina, USA) *(SIGCSE '12)*. Association for Computing Machinery, New York, NY, USA, 385–390. https://doi.org/10.1145/2157136.2157249

[29] James Prather, Paul Denny, Juho Leinonen, Brett A. Becker, Ibrahim Albluwi, Michelle Craig, Hieke Keuning, Natalie Kiesler, Tobias Kohn, Andrew Luxton-Reilly, Stephen MacNeil, Andrew Petersen, Raymond Pettit, Brent N. Reeves, and Jaromir Savelka. 2023. The Robots Are Here: Navigating the Generative AI Revolution in Computing Education. In *Proc of the 2023 Working Group Reports on Innovation and Technology in CS Education* (Turku, Finland) *(ITiCSE-WGR '23)*. ACM, New York, NY, USA, 108–159. https://doi.org/10.1145/3623762.3633499

[30] Arun Raman and Viraj Kumar. 2022. Programming pedagogy and assessment in the era of AI/ML: A position paper. In *Proceedings of the 15th Annual ACM India Compute Conference*. 29–34.

[31] Sami Sarsa, Paul Denny, Arto Hellas, and Juho Leinonen. 2022. Automatic Generation of Programming Exercises and Code Explanations Using Large Language Models. In *Proc of the 2022 ACM Conference on International Computing Education Research - Volume 1* (Lugano and Virtual Event, Switzerland) *(ICER '22)*. ACM, NY, USA, 27–43. https://doi.org/10.1145/3501385.3543957

[32] Jaromir Savelka, Arav Agarwal, Marshall An, Chris Bogart, and Majd Sakr. 2023. Thrilled by Your Progress! Large Language Models (GPT-4) No Longer Struggle to Pass Assessments in Higher Education Programming Courses. In *Proceedings of the 2023 ACM Conference on International Computing Education Research - Volume 1* (Chicago, IL, USA) *(ICER '23)*. Association for Computing Machinery, New York, NY, USA, 78–92. https://doi.org/10.1145/3568813.3600142

[33] David H. Smith IV and Craig Zilles. [n. d.]. Code Generation Based Grading: Evaluating an Auto-grading Mechanism for "Explain-in-Plain-English" Questions. arXiv:2311.14903 [cs.CY]

[34] Leigh Ann Sudol-DeLyser, Mark Stehlik, and Sharon Carver. 2012. Code Comprehension Problems as Learning Events. In *Proceedings of the 17th ACM Annual Conference on Innovation and Technology in Computer Science Education* (Haifa, Israel) *(ITiCSE '12)*. Association for Computing Machinery, New York, NY, USA, 81–86. https://doi.org/10.1145/2325296.2325319

[35] Alaaeddin Swidan and Felienne Hermans. 2019. The Effect of Reading Code Aloud on Comprehension: An Empirical Study with School Students. In *Proceedings of the ACM Conference on Global Computing Education* (Chengdu,Sichuan, China) *(CompEd '19)*. Association for Computing Machinery, New York, NY, USA, 178–184. https://doi.org/10.1145/3300115.3309504

[36] Renske Weeda, Cruz Izu, Maria Kallia, and Erik Barendsen. 2020. Towards an Assessment Rubric for EiPE Tasks in Secondary Education: Identifying Quality Indicators and Descriptors. In *Proc. of the 20th Koli Calling International Conf. on Computing Ed. Research* (Koli, Finland) *(Koli Calling '20)*. ACM, New York, NY, USA, Article 30, 10 pages. https://doi.org/10.1145/3428029.3428031

[37] Matthew West, Geoffrey L Herman, and Craig Zilles. 2015. PrairieLearn: Mastery-based online problem solving with adaptive scoring and recommendations driven by machine learning. In *2015 ASEE Annual Conference & Exposition*. 26–1238.