Automated Questionnaires About Students' JavaScript Programs: Towards Gauging Novice Programming Processes

Teemu Lehtinen
teemu.t.lehtinen@aalto.fi
Aalto University
Espoo, Finland

Lassi Haaranen
lassi.haaranen@aalto.fi
Aalto University
Espoo, Finland

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

ABSTRACT

Students sometimes manage to produce functionally correct program code while having a fragile understanding of the related learning goals. Such unproductive success could be intercepted by an educator who asks questions that target the structure and evaluation of the student's program using the constructs and identifiers in the code. We provide a tool that automatically generates multiple-choice questions of seven different types for this purpose. We integrated these questions into a web-based program writing exercises, which we also publish as a part of this work, and successfully used them on an introductory programming course. In our pilot evaluation of the tool, we found that the students who answer these questions repeatedly incorrectly are likely to drop out, have more challenges while writing a program, and resort to tinkering behavior.

CCS CONCEPTS

• **Applied computing** → **Interactive learning environments**; • **Social and professional topics** → **Computer science education**.

KEYWORDS

QLC, unproductive success, program comprehension, introductory programming, online education

ACM Reference Format:
Teemu Lehtinen, Lassi Haaranen, and Juho Leinonen. 2023. Automated Questionnaires About Students' JavaScript Programs: Towards Gauging Novice Programming Processes. In *Australasian Computing Education Conference (ACE '23), January 30-February 3, 2023, Melbourne, VIC, Australia.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3576123.3576129

1 INTRODUCTION

Online courses are an important way for people outside of formal education settings to acquire new skills and continue learning. The recent pandemic generated additional demand for distance learning for all levels of education. In computer science (CS) and programming, many systems can provide automated assessment to grade students' solutions [25] which supports running online and on-campus courses.

ACE '23, January 30-February 3, 2023, Melbourne, VIC, Australia

ACM ISBN 978-1-4503-9941-8/23/01.
https://doi.org/10.1145/3576123.3576129

All of the above has contributed to a large variety of online courses in CS of which many consist of instructional material and exercises in which students write programs that are automatically checked for correctness (e.g. [5, 36]). The exercise points that students earn by solving the assignments are commonly a substantial factor for final course grades. Presumably, solving exercises correctly is tantamount to learning. However, prior research has indicated that at least some students have difficulties explaining their own functionally correct code [16].

Kapur [12] notes that performance and learning are incommensurable. That is, a student might perform well with given exercises but the actual learning might be short term and not lead to longer term learning and understanding of the topic. He considers four design considerations for learning: unproductive failure, unproductive success, productive failure, and productive success. In our present work, we are particularly interested in unproductive success. These are cases where students succeed in solving assignments correctly, but their understanding of programming is fragile and prone to misconceptions. Evidence of unproductive success in CS education and programming tasks exists [14, 24, 29].

Identifying students who might be able to piece together a functionally correct solution to an assignment with little conceptual understanding is an important and interesting problem to solve in (online) CS courses. Prior work has investigated asking *Questions about Learner's Code* (QLCs) which have been suggested to catch cases of unproductive success in programming exercises [17]. This approach uses questions that target the structure and evaluation of the learners' own program using the constructs and identifiers in the code they wrote. Alternatively, the QLCs can be described as program comprehension questions with the addition that they are in the context of a program that the learner recently created.

Our present work offers a tool to generate multiple-choice QLCs for JavaScript programs. We provide an open sourced QLC library and open sourced program writing exercises that can deliver QLCs to students. The assignments are available via a content server which enables testing them in a web browser without installing anything as well as using them in different learning management systems (via supported protocols, e.g. LTI).

After reviewing related work in Section 2 we describe the tool contribution in Section 3. Section 4 describes how we successfully used the tool on an online introductory programming course collecting data and conducting a pilot evaluation. Finally, in Section 5, we discuss how the presented work, including the tool, may be useful in teaching as well as research, and conclude the work in Section 6.

2 RELATED WORK

2.1 Questions About Learners' Code

Questions about Learners' Code or QLCs have been defined as 1) being questions about code written by a student, 2) QLCs refer to constructs or patterns in the student's program, and 3) they are presented to the student. Automatic QLCs have the additional condition of being generated automatically by analysing the student's code [17].

Santos et al. [30] presented an automated QLC system, Jask, for a subset of Java programs in introductory programming context. The system can generate 17 different question types and the authors provide question templates which the system fills in to form actual question instances for specific programs. The system design is described at high-level steps. The answers were collected as text, including numeric values and identifiers, and automatically assessed by the system. The authors conducted an initial study with student volunteers from an introductory programming course who, on average, answered 75% of the QLCs correctly. The questions targeting dynamic properties of the programs, such as tracing variable values, were considerably more challenging having success rates below 50%.

Jask has not been offered publicly for adoption or evaluation at the time of writing. Furthermore, supporting different programming languages on the same code base is difficult because there are constructs, such as generators in Python or object notation in JavaScript, that don't easily map across on a syntactic level where questions need to be presented. The system we are presenting differs from Jask by targeting a different programming language, using multiple-choice questions, and integrating into an existing programming exercise platform. In practice, this means that all students receive QLCs once their program passes automated functional tests.

Lehtinen et al. [16] experimented with questions that were prepared manually to target constructs in programs that pass the functional tests for the selected exercises. Although all students received the same questions, they appeared as QLCs to the students who answered in the context of their own program code. On average, one third of the students struggled to answer these questions in open text answers. The students who answered QLCs better had higher course success and retention.

Henley et al. [7] described a system that aims to detect code smells and possible misconceptions while students are writing programs. They suggest an intervention that includes questions on details about the related program structures. We consider this as a different use-case for QLCs.

While a lot of programming education focuses on program writing, researchers have increasing interest and knowledge of benefits in teaching program comprehension [11, 23], including tasks such as program reading, self-explanation, and program tracing. We consider QLCs as one type of task that has potential to measure and teach program comprehension.

There are several recent approaches where program comprehension is practiced with the help of generated questions [28, 33, 34, 37]. Tamang et al. [35] presented an approach focusing on the scaffolding of self-explanation of programs. It can produce questions that approach human-generated quality. These approaches differ from QLCs in that the questions have not been generated and posed for the learner using a program that the learner created.

2.2 Programming Process

Experts approach solving programming problems differently from novices (e.g. [21]). However, there are also differences in problem solving styles within novices. Perkins et al. [27] described three different problem solving strategies in novice programmers: stoppers, movers, and tinkerers. Stoppers typically give up on solving the problem, movers explored alternative approaches, and tinkerers preferred small changes when trying to solve the problem.

Hosseini et al. [8] also created a categorisation of student behaviors. By looking at the sequences of steps students took in solving programming assignments they identified four types of behaviors: builder, massager, reducer, and struggler. Builders correspond to movers, that is, they increase the concepts used in the program as well as its correctness. They also found that some students "massage" their code by doing a series of small changes that do not change the number of concepts in the program or its correctness. A reducing pattern was also identified, where the student reduces the number of concepts used in the program. Finally, the struggling students try different changes to the code but have difficulties getting even the first steps correct.

Heinonen et al. [6] presented a tool to browse program snapshots of students' work in exercises. The tool was evaluated with experts examining programming processes and recording errors and challenges that students had. The students were divided into two groups and the recorded labels were used to describe their differences. This initial study only included 20 students and it suggested that students failing the course lacked a systematic approach and had difficulties with understanding and applying conditionals. We use a similar method to analyse programming processes using more fine-grained data.

Shrestha et al. [31] presented a tool that can visualise the relation between the completed program and the program state after each programmer's keystroke in a single diagram. In addition, the individual code states can be browsed by pointing at the diagram. Experts were instructed to use this tool to review different programming processes and they were able to efficiently detect interesting student behaviours, such as copy-pasting, removing parts of the program, and periods of linear program writing in contrast to jumping to edit different parts of the code. The tool may help reveal whether the student is using a top-down or bottom-up approach in programming.

Methodologically our work also bears similarities with Vihavainen et al. [36] where they investigated how novices wrote their first lines of code both qualitatively and quantitatively. They found out that in an IDE many students tended to start writing code linearly (as in typing it out character by character) in the beginning of the course. But already on the second week of the course, many beginners took advantage of the IDE shortcuts.

One approach to identifying students who are struggling with programming assignments is to look at their typing patterns. Earlier research has shown that identifying students is possible based on the keystroke data they produce when solving programming assignments [22]. Building on that Leinonen et al. [20] sought to infer the

Table 1: The types of generated QLCs including the question template and possible answer options. The values in italics will be substituted with their actual content.

ID	Question Template	Correct	Wrong options (distractors)
Q1	Which is the name of the function [declared on line *n*]?	*function-id*	function, *parameter-id(s)*, *variable-id(s)*, *keyword(s)*
Q2	Which are the parameter names of the function [declared on line *n*]?	*parameter-id(s)*	*function-id*, function, *variable-id(s)*, *keyword(s)*
Q3	Which value does *parameter-id* have when execution of *function-call* starts?	*input-value*	*other-input-value(s)*, *literal(s)*, *random-value(s)*
Q4	A program loop starts on line *n*. Which is the last line inside it?	*line-at-end*	*line(s)-before-loop*, *line(s)-after-loop*, *line(s)-inside-loop*
Q5	A value is [assigned to|accessed from] variable *variable-id* on line *n*. On which line is *variable-id* defined?	*declaration-line*	*reference-line(s)*, *other-random-line(s)*
Q6	Which is the ordered sequence of values that are assigned to variable *variable-id* while executing [*function-call*|the program]?	*variable-history*	*altered-variable-history*, *random-values*
Q7	Which best describes *property-name* on line *n*?	method	argument, keyword, operator, parameter

experience and performance of programmers based on the typing patterns. They looked at typing patterns and exam performance and found that it can be used to explain some of the variance. They also looked at classifying programmers to novices and non-novices based on the keystroke data and found that it shows some promise.

Leinonen et al. [18] showed that keystroke-level data provides a time-on-task metric that outperforms metrics based on the submission timestamps when predicting success on the course. They argue that the more fine-grained keystroke-level metric takes into account the breaks that students take in learning. We apply the same fine-grained time-on-task metric in our study.

A working group report [10] provides a broader literature review of research related to data in learning programming—including data from programming processes.

3 THE TOOL

3.1 QLCs Generation Library

The generation of QLCs about JavaScript programs is contained in an open sourced, documented library which supports development of different systems related to QLCs[1]. The library has an interface that takes a JavaScript code and a QLC configuration as inputs. It outputs the questions, multiple-choice answer options, and the pedagogical explanations for the options as plain data objects. The code is packaged using an industry standard development tools and can be built for both server and client use.

The process of generating QLCs starts from an abstract syntax tree (AST) of the program and a history of variable values that is recorded while evaluating the program. Then, suitable nodes and their properties are selected from the AST to decide which types of questions are generated. The QLC configuration can limit the types of questions that are considered. Finally, a previously demonstrated method is used to populate the selected question templates with the facts extracted from the AST and the recorded history [17, 30]. The library currently generates seven different types of questions and multiple-choice answer options as presented in Table 1.

The question templates were selected so that they target learning goals that are relevant for our pilot study. Most of the selected question templates have been used in previous QLC studies or are small adaptations so we could expect efficient implementation and that the students would mostly understand the questions posed to them. Additionally, a pedagogically aimed explanation text is provided for each answer option based on how that option was generated (see Figure 1 for examples).

3.2 Programming Exercises

Generation of QLCs remains a theoretical curiosity unless the necessary steps are taken to integrate them into students' workflow. This includes collecting learners' programs, posing the QLCs as a continuum of their programming process, providing automated feedback that supports learning, and recording all of the above learning data for analytics and research.

We constructed a system providing online JavaScript programming exercises with QLCs support. These exercises use a code editor in a web browser window. In addition, the interface includes a task description and a grade button that runs functional tests on the student code and displays test results as feedback. After all the tests pass, the system has an option to generate a set of multiple-choice QLCs and pose them to the students below their program code. Each question has 5 answer options.

Figure 1 presents a programming exercise at the time of answering the generated QLCs. When students choose an incorrect answer (distractor) the system displays a description of why this is not the correct answer. When students choose the correct answer the system displays descriptions for all the available options and rewards full points for the exercise.

[1] https://github.com/teemulehtinen/qlcjs

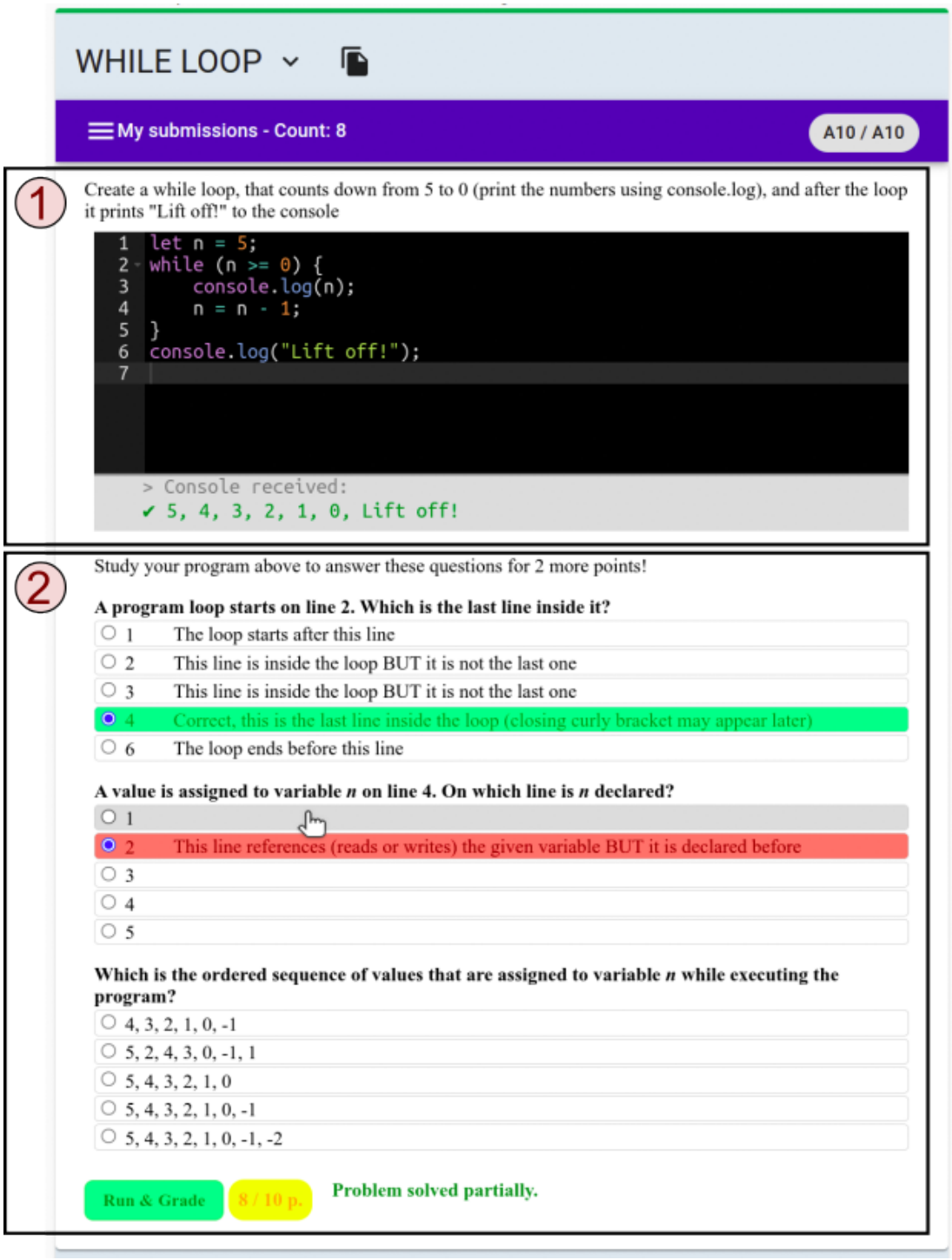

Figure 1: Screen capture from the exercice system. First the student is only shown the exercise (1). After the student successfully completes the exercise the QLCs are shown for the student to answer (2).

Table 2: The selected exercises including the description for students and the types of QLCs generated.

ID	Title	Description	QLC types
E1	Half number function	Define a function called `halveNumber`, that takes a number as parameter and returns the given number divided by 2	Q1 Function name Q2 Parameter name Q3 Parameter value
E2	While loop	Create a while loop, that counts down from 5 to 0 (print the numbers using console.log), and after the loop it prints "Lift off!" to the console	Q4 Loop end Q5 Variable declaration Q6 Variable trace
E3	Count word	Write a function called `countWord` that takes a search word as the first parameter, and an array of any words as the second parameter. The function must count the number of times the given search word is in the given array.	Q3 Parameter value Q4 Loop end Q5 Variable declaration
E4	Repeat note	Create a function called `repeatNote`, that takes as parameters a `note` as string and a number `n`. The function returns a string where the given `note` is repeated `n` numbers of times, each time separated by a white space, without white space at the end. For example calling `repeatNote("C#", 3)` should return `"C# C# C#"`. There are a number of ways to achieve this, for example by using the following methods: string.repeat() and string.trim()	Q7 Method call Q6 Variable trace

The exercises are defined using configuration files. An exercise requires a task description, possible code template to start with, automated tests as JavaScript functions, a list of active QLC types, and possible inputs for generating QLCs related to program evaluation.

The system to deliver JavaScript programming exercises and QLCs is provided as an open sourced content package for Acos content server.[2] The project documentation also links to online exercises that are available for testing them online – without the need of installing anything. The Acos content server offers integration to different learning management systems (LMS) which may use different interoperability protocols [32]. One of the widely used protocols is LTI that can be used to include the exercises to courses run in many of the commercial and open-source LMS products available.

Currently, the QLCs are enabled for four selected exercises (E1-4) for which we configured question types that were relevant for the learning goals of the exercises. Table 2 presents the exercise titles, descriptions as given to students, and the enabled question types. Additionally, E3 included the required function signature as a template in the program editor.

3.3 Enabling Research

The system collects each keystroke and other action event from the user interface to the system logs. Each event includes a timestamp and relevant details, such as the inserted/removed characters with their location, the mouse coordinates, the generated QLCs, and the selected answer option. These series of events are stored for each session along the exercise state and user identifier using a JSON-formatted row in the log files.

To support CS education research the system includes a feature to play the events from a logged session inside the browser interface. In a way, this allows examination of learners' work "over their shoulder" at a later time. The researcher can also move on the timeline freely. To enforce privacy there are no means to select stored logs from the browser. Instead, once the logs are retrieved from the server and research consents are handled properly an individual JSON session can be posted to the system's replay URL.

[2]https://github.com/teemulehtinen/acos-webdev-editor

4 PILOT EVALUATION

4.1 Context

We used the QLC system in the spring of 2022 on an online course that is offered by the university to life-long learners. The course had 56 student participants of whom 51 gave research consent. The aim of the course is to combine introductory programming with web development. The focus of the first half of the course is on learning basics of HTML & CSS and how to program with JavaScript in the browser. There are no prerequisites for the course leading to participants with very different backgrounds and demographics.

The course consists of an interactive ebook with automatically assessed exercises without any final exam. The course is organised completely online using the ebook without on-premise lectures or lab sessions. The course platform has a question and answer (Q&A) section where students can request help if they are stuck with an exercise or do not understand a particular concept.

Our evaluation targeted the third and fourth round of the course (R3 and R4). The QLCs tool was used on R3, and we include student points from R4 as a measure of student success. These rounds introduced programming using concepts such as variables, functions, conditional statements, and iteration statements.

R3 included four chapters of the ebook that comprised 23 programming exercises, three quizzes, eight short videos, two program visualisations, and a generous amount of annotated example code and text. While the programming exercise system we described in section 3 was used on the course regularly, we decided to pilot QLCs in four selected exercises (E1-4). In order to limit the new type of effort for students, E1-4 were located in separate sections of R3 with other exercises and learning activities between them.

In E1-4, students had 10 attempts to create a program that passed all of the automated tests. The input, output, and result for each test was provided as feedback. After all tests passed, the student was awarded 8/10 points for the exercise and a set of 2-3 QLCs were generated and posed to the student. Students had unlimited attempts and received descriptive feedback when answering the multiple-choice QLCs. Once they answered them correctly they were rewarded the full 10 points for the exercise. Considering this, our interest for E1-4 lies in the incorrect choices rather than final points.

We also investigated student success on the fourth round (R4) of the course that continued to practice and apply the rudimentary programming concepts using the web context. The rounds after R4 are arranged by different topics on web programming, such as a HTTP server or client session, and they are not required for passing the course. Previously some students have skipped rounds they found less interesting and many of the life-long learners are not on the course to get the best grades. We deemed R3 and R4 to provide an adequate understanding about students' success in learning rudimentary programming concepts without introducing too much variation from personal learning preferences. In the following, we present *total points* as a percentage of all exercise points available in R3 and R4.

4.2 Data Collection

As described in section 3, our programming exercise system collected event data on the keystroke-level so that any state of a program that a student edited in the system could be recovered. In addition to the log data, we had access to the course database including the student answers, automated grading results and feedback for each exercise or survey, as well as posts to the Q&A section. For this research we fetched the aforementioned data for the 51 students who gave their research consent during the enrolment survey. The pre-processing step combined the log and database data using cross-references and created anonymised data tables for researchers to analyse. We wanted to analyse answers to QLCs – thus we included 39 students who completed at least one of E1-4 i.e. created a program and answered the related set of QLCs.

4.3 Method

4.3.1 Quantitative. Previous research on QLCs has collected students' answers as text they could submit once. Our research uses multiple-choice answer options that students selected until they, sooner or later, arrived to the correct answer. We decided to use students' first selection as a comparison with the studies where students could submit only once. We report, separately for each question type, the ratio of students who answered correctly on first attempt as well as the numbers of times different wrong options were chosen.

We wanted to evaluate the correlation between answering QLCs and success in learning rudimentary programming concepts. Our study included students who had answered different numbers of QLCs. In order to compare between all students, we decided to measure success in QLCs by the student's average number of erroneous choices while answering them. The other variable, learning success, is measured as the total points i.e. percentage of all exercise points available in R3 and R4. The relevance of these rounds was discussed in subsection 4.1. Almost every student works linearly through R3 and R4 and rarely stops working on an exercise before they are awarded full points for the task i.e. a student having 50% of the points has most likely completed R3 and has not completed significant work in R4. Therefore, we see the total points as an equally good measure for course retention in our context. The more points students collect, the further they proceed in the ebook – and practice more principles of programming in JavaScript.

To evaluate correlation between students' average number of errors in QLCs and the total points we provide a visual correlation inspection for increased transparency. We report the Pearson correlation coefficient and p-value using a two-sided test.

4.3.2 Qualitative. We wanted to study how different students created their programs and whether those programming processes have any underlying potential reasons for how the students answered the following QLCs. Considering previous research on programming processes, they are not easily described or measured. We decided to qualitatively research the recorded programming processes in E1-4.

Using the replay feature described in section 3, we examined how students had produced their programs. We started with 132 processes of which we rejected processes where the majority of the source code was copy-pasted to the browser as that concealed possible struggles that students had during writing the program. We assume most of these students preferred to use an IDE over the online editor. In the end, 92 processes from 24 different students were included in the analysis.

We select 6 students to represent the population and cover the spectrum and outliers of the interplay between successes in QLCs and the total points. While we provide few numeric facts of those student processes, the result of the study are the qualitative descriptions of how they created the programs i.e what features the first author observed in the programming processes. In results, we focus on objective observations and strive to limit our interpretations until discussion.

4.4 Results

4.4.1 QLC answers. Table 3 reports how students answered the QLCs in E1-4. The results reveal that the initial success at multiple-choice QLCs, the first choice, is roughly similar to success in text answers for manually prepared QLCs [16] or text answers to automatically generated QLCs for another programming language [30]. Questions on code's surface-level about atomic concepts, relations, or code blocks were answered, at first attempt, above 70% of times correctly. Questions about details on execution-level had much lower 30-40% initial success.

Furthermore, Table 3 presents that when the same question type was repeated it was answered more correctly on the first attempt in 3 cases and less correctly in 1 case. The order in popularity of incorrect choices follows the intuition of the authors – harder to easier.

4.4.2 Success correlation. Figure 2 presents on the left the distribution of students' average number of errors while answering QLCs as well as the first and third quantiles that limit the 25% of students

Table 3: Student Answers to QLCs.

Exercise	QLC type	N	QLC correct on first	Incorrect answer options for QLC (number of times selected)
E1	Q1 Function name	35	34 (97%)	Keyword (1)
	Q2 Parameter name	35	29 (83%)	Function name (7) Keyword (6)
	Q3 Parameter value	35	15 (43%)	Random value (18) Parameter name (9) Literal in body (8)
E2	Q4 Loop end	30	21 (70%)	Line inside block (8) Line before block (7) Line after block (2)
	Q5 Variable declaration	30	23 (77%)	Reference line (16) Random line (4)
	Q6 Variable trace	30	10 (33%)	Miss last (19) Miss first (9) Shuffled (2) Extra at end (1)
E3	Q3 Parameter value	25	19 (76%)	Wrong parameter value (4) Parameter name (3)
	Q4 Loop end	25	14 (56%)	Line inside block (11) Line after block (6) Line before block (1)
	Q5 Variable declaration	25	22 (88%)	Reference line (4)
E4	Q6 Variable trace	18	10 (56%)	Miss first (4) Smaller value (3) Shuffled (3) Miss last (2) Extra at end (1)
	Q7 Method call	18	17 (94%)	Keyword (1)

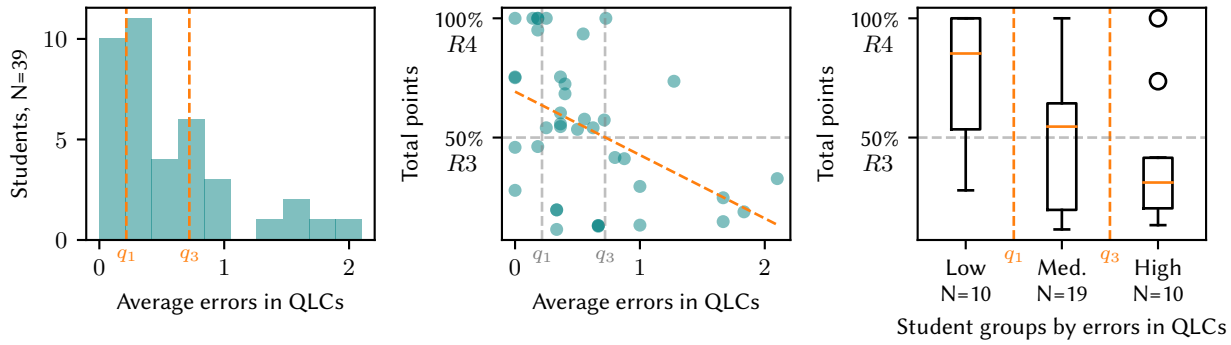

Figure 2: The distribution of students' average number of errors while answering QLCs and correlation with the total points they received in the two rounds R3 and R4. Quantiles q_1 and q_3 mark the limits for 25% of students with the least and most errors respectively.

with the least errors and the 25% of students with the most errors respectively.

In the middle, the students' average errors in QLCs is compared with their total points. Students who are below 50% did not successfully continue working on the fourth round, R4. In addition, a regression line is fitted for the data. The Pearson correlation coefficient for the average errors and points is −0.459. The p-value for a two-sided test for non-correlation is 0.003.

On the right, the students are divided into groups by the quantiles so that there are 10 students with the least errors, 19 students in the middle, and 10 students with the most errors in QLCs. Students in all of these groups received full points for the programming tasks related to the QLCs they answered. The representation of the distributions of their points highlights that 25% of students who had on average 0.72 or more errors in QLCs (the "high" group) were likely to drop out and not continue to learn more programming on the course. This supports the previous finding of correlation between success in manually prepared QLCs and course retention [16].

4.4.3 Processes. Figure 3 employs the previously presented correlation diagram to mark the six students (S1-6) that we selected to represent the population. The students marked in gray had a

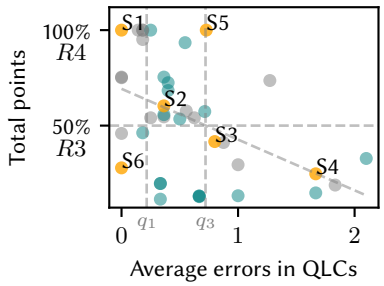

Figure 3: The students described qualitatively in this study.

process that included mostly copy-pasted code (i.e., likely written in another editor) and were rejected from qualitative analysis. Table 4 collects the available measures for the selected students. In following, we describe our observations of the programming processes, in turn for each selected student.

S1 had no recognisable challenges and did report to have up to 1000 hours of previous programming experience. They efficiently

Table 4: Students in Qualitative Analysis.

	Average errors in QLCs	Total points %	Self-reported experience, h	Average time-on-task*, min	Average attempts at program tests*	Q&A posts
S1	0.0	100	101–1000	4	1.0	0
S2	0.4	60	11–100	23	2.3	0
S3	0.8	42	101–1000	28	6.7	8
S4	1.7	25	0	91	8.0	6
S5	0.7	100	11–100	33	2.7	2
S6	0.0	28	n/a	40	7.0	5

* In tasks E2-4, E1 was excluded as a much simpler task that required less work

seemed to decide on the next subgoals and knew how to implement them for E1-4. S1 passed all functional tests for the programs on the first attempt. S1 answered QLCs in a timeline that suggests they derived the correct answers with thought. They did not use the Q&A service.

S2 wrote a conditional statement for a while-loop in E2 that had that counted up instead of down, but noticed it while writing the related decrement statement. In the related QLCs, they did take time to consider but first missed the last iterator value violating the loop condition in *Q6 Variable trace*. In E3, S2 had a plan on how to solve the problem but was little challenged with including all the necessary parentheses and seemed to lean towards Python type of syntax – they did report up to 100 hours of previous programming experience. Also, they were about to start array iteration from 1 instead of 0 but again, fixed all errors before running the program. In E4, S2 did first some experimentation and eventually ran a version that was showing the expected value instead of returning it as required in the task description. After fixing that they did again have off-by-one errors in *Q6 Variable trace*.

S3 ran into numerous challenges. In E2, they started to change the design of an incrementing while-loop into a decrementing one but quickly resorted to tinkering where they combined multiple conditionals and inserted break statements. They reduced it gradually to the simple solution 30 minutes later. In the related QLCs, they had errors in both *Q5 Variable declaration* and *Q6 Variable trace*. In E3 & E4, S3 was seemingly missing plans to proceed even while seeking help in the Q&A service and reporting up to 1000 hours of previous experience. They tried a lot of program runs and eventually passed functional tests in E4 with an incomplete solution using a corner case that was not realised when the instructor had prepared the tests. They had more errors in the QLCs *Q3 Parameter value* and *Q5 Variable declaration* (for the second time).

S4 completed only E1 and E2. In E2, they too attempted to start from a design of an incrementing loop. After initial attempts, they copied example code and placed a number from the task description to a printed string as they would expect that to have an effect in how many times a program loop will repeat. Next, S4 changed a plus operator to a minus without other considerations. We consider these actions tinkering that was based on the description but did not follow any plan for a program. S4 did complete E2 with help from the Q&A service after 10 editing sessions for 6 lines of code. They did not have previous programming experience. Considering QLCs, S4 did stop to think for each question but most of their first choices were incorrect and then they trialed the options until hitting the correct one.

S5 ended little off-road in E2 by failing to assign a variable when decrementing and by applying a condition to execute a statement that could have simply been located after a loop. However, it seems they proceeded with a subgoal at a time and managed to solve them in a matter of minutes. In E3 and E4, the processes were similar including initial challenges with handling arrays and padding a string with space. S5 selected many answers to QLCs promptly and had errors in *Q3 Parameter value* on two occasions as well as errors in *Q6 Variable trace* and *Q4 Loop end*. They did not explicitly use the Q&A service for E1-4 and reported up to 100 hours of previous experience.

S6 reached as far as to attempt E3. In E1, they struggled with syntax to declare a function and which name to use for calling it. Once they completed the program and received QLCs, they used time to consider and answered perfectly. Similarly in E2, they struggled with the concept of a code block and its syntax for 8 editing sessions and 7 attempts to pass the tests – yet they carefully chose correct answers to QLCs. Despite posting to Q&A service they could not finish E3 which in the end had a typo in "lenght" and was missing a return statement for the calculated value.

Based on these results, we argue that students who answered incorrectly to many types of QLCs had fewer plans for implementing a program and resorted to tinkering behaviour. We recognised more challenges per programming task for these subjects.

The stronger success students had in QLCs, the more systematically they recognised and resolved issues in their programs. Although, the outliers in data reveal that the care that students invest in answering QLCs and their attitude towards unlimited attempts may affect their average number of errors in QLCs significantly.

5 DISCUSSION

The main contribution of this work is the presented open sourced tool for automatically generating Questions about Learners' Code (QLC). Our system serves as a reference implementation for this type of software for JavaScript questions and it can be readily adopted into teaching and research practice.

5.1 QLC Tool

Considering practitioners, our tool provides a novel type of exercise for courses. There are many tools targeting productive successes, i.e. for cases where students get the exercise correct and learn from

it, but very few targeting unproductive successes which is where QLCs shine. As an example, prior research has shown benefits of Parsons problems [3, 4, 26], and multiple tools for presenting Parsons problems to students have been presented in prior work [9, 13, 15, 38]. Having a QLC tool that can be easily integrated to courses is a valuable addition to the toolbox of those teaching JavaScript courses.

Future research could look into QLCs correlation with time-on-task, number of grading attempts, or other measures that may already be available on a given course – we did not research such correlations due to the small size of our process data. However, adopting QLCs could have additional value to practitioners. They can help with identifying at-risk students early for targeted interventions, with the added benefit of not just being an early warning sign but also providing information of what concepts and ideas the student is having difficulties with. In addition, QLCs have been previously speculated to have potential to act as self-explanation prompts that help learning [17]. Research on that avenue is yet to be started.

One problem pointed out in prior work is the potential over-reliance on automated assessment tools that are ubiquitous in programming courses [2, 19] – Baniassad et al. dubbed this "autograder insanity" where students tinker their programs based on autograder feedback, never thinking about their program carefully but eventually ending up with a correct solution. We propose that QLCs could be a "cure" for the autograder insanity as students who tinker their way to a solution will most likely be less successful on QLCs, and can then be given additional support.

Another unfortunately common issue in online courses is plagiarism [1]. One way to try combat plagiarism would be using QLCs – if the student plagiarised their answer, they might not be able to correctly answer QLCs related to their answer. At the very least they will need to study the plagiarised code in more detail, potentially helping them learn and make the best of a bad situation.

5.2 Pilot Evaluation

Regarding QLC related research, though our pilot evaluation population is small, our observations are indicative towards making new hypotheses and designing further studies with QLCs. Our results regarding students' success rates are aligned with earlier studies [16, 30]. Supplementing earlier results, we showed that repeatedly answering QLCs incorrectly is a sign of immediate danger and the student is likely to drop out from the course as seen in Figure 2.

Based on the qualitative analysis of the students' programming processes, we found that students with a higher error rate in QLCs had more challenges while writing a program and resorted to tinkering behaviour. For example, students repeatedly run the program with small changes that do not seem to follow a logical plan and finally pass the tests. We argue such examples are likely to include cases of unproductive success and we know the same students did not progress very far in their studies. This highlights how QLCs can give us information that traditional unit test -based assessment can not or provides too late.

In our future work, we are interested in studying the types of interventions students should be given when they incorrectly answer QLCs. In smaller courses, such as the one where the pilot evaluation was conducted, student performance on QLCs could be visualised to the teacher, who could then decide the type of support personalised for the student. In larger courses, performance on QLCs could be used, for example, to decide whether the student should be given an additional exercise on the topic if they fail to answer QLCs correctly.

5.3 Limitations

There are limitations to the tool and the pilot evaluation that we outline here. Related to the tool, it currently only supports JavaScript, which could hinder adoption of the tool. As the tool has been developed with our specific context in mind, there are potentially features that would be useful, but we have not thought about.

Related to the pilot evaluation, the population in the course where the tool was evaluated is different from many other computing courses. Most of the students are lifelong learners, which likely affects their motivations for taking the course and backgrounds in general. Thus, it is possible that results would be different in more traditional contexts (e.g., a CS1 course with majors). The pilot evaluation had a relatively small number of students (n=51) and used relatively simple methods – correlational analysis and one researcher going through student processes qualitatively. In our future work, we are planning on conducting a more rigorous research study related to how the tool affects student performance.

Lastly, our qualitative analysis relied on observing the reconstructed programming processes where some behaviours were classified as challenges. However, we do not really know about students' motivations for their actions: for example, trying what happens when parts of the code are changed could be productive, intentional exploration by the student, but classified as a challenge in our analysis. In our future work, we are interested in conducting a think-aloud study to examine student challenges in more detail.

6 CONCLUSION

We presented a tool for automatically generating Questions about Learners' Code (QLC) for JavaScript courses. A pilot evaluation supports the value of the tool, showing that being unsuccessful on QLCs correlates with dropping out of the course and can give instructors information on what topics students struggle with. QLCs fill the gap often present in programming courses by focusing on unproductive successes where students get the exercise right but do not understand their solution, which is missing from many traditional tools.

ACKNOWLEDGMENTS

To Evan Cole, for advice and implementing QLCs in study-lenses which inspired us to this research.

REFERENCES

[1] Ibrahim Albluwi. 2019. Plagiarism in programming assessments: a systematic review. *ACM Transactions on Computing Education (TOCE)* 20, 1 (2019), 1–28.
[2] Elisa Baniassad, Lucas Zamprogno, Braxton Hall, and Reid Holmes. 2021. Stop the (autograder) insanity: Regression penalties to deter autograder overreliance. In

Proceedings of the 52nd ACM technical symposium on computer science education. 1062–1068.

[3] Yuemeng Du, Andrew Luxton-Reilly, and Paul Denny. 2020. A review of research on parsons problems. In *Proceedings of the Twenty-Second Australasian Computing Education Conference*. 195–202.

[4] Barbara J Ericson, Lauren E Margulieux, and Jochen Rick. 2017. Solving parsons problems versus fixing and writing code. In *Proceedings of the 17th Koli Calling International Conference on Computing Education Research*. 20–29.

[5] Lassi Haaranen, Giacomo Mariani, Peter Sormunen, and Teemu Lehtinen. 2020. Complex Online Material Development in CS Courses. In *Koli Calling '20: Proceedings of the 20th Koli Calling International Conference on Computing Education Research* (Koli, Finland) *(Koli Calling '20)*. Association for Computing Machinery, New York, NY, USA, Article 26, 5 pages. https://doi.org/10.1145/3428029.3428053

[6] Kenny Heinonen, Kasper Hirvikoski, Matti Luukkainen, and Arto Vihavainen. 2014. Using CodeBrowser to Seek Differences between Novice Programmers. In *Proceedings of the 45th ACM Technical Symposium on Computer Science Education* (Atlanta, Georgia, USA) *(SIGCSE '14)*. Association for Computing Machinery, New York, NY, USA, 229–234. https://doi.org/10.1145/2538862.2538981

[7] Austin Henley, Julian Ball, Benjamin Klein, Aiden Rutter, and Dylan Lee. 2021. An Inquisitive Code Editor for Addressing Novice Programmers' Misconceptions of Program Behavior. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)*. 165–170. https://doi.org/10.1109/ICSE-SEET52601.2021.00026

[8] Roya Hosseini, Arto Vihavainen, and Peter Brusilovsky. 2014. Exploring Problem Solving Paths in a Java Programming Course. In *Psychology of Programming Interest Group Conference, PPIG 2014*. 65 – 76. http://d-scholarship.pitt.edu/21832/

[9] Petri Ihantola and Ville Karavirta. 2010. Open source widget for parson's puzzles. In *Proceedings of the fifteenth annual conference on Innovation and technology in computer science education*. 302–302.

[10] Petri Ihantola, Arto Vihavainen, Alireza Ahadi, Matthew Butler, Jürgen Börstler, Stephen H. Edwards, Essi Isohanni, Ari Korhonen, Andrew Petersen, Kelly Rivers, Miguel Ángel Rubio, Judy Sheard, Bronius Skupas, Jaime Spacco, Claudia Szabo, and Daniel Toll. 2015. Educational Data Mining and Learning Analytics in Programming: Literature Review and Case Studies. In *Proceedings of the 2015 ITiCSE on Working Group Reports* (Vilnius, Lithuania) *(ITICSE-WGR '15)*. Association for Computing Machinery, New York, NY, USA, 41–63. https://doi.org/10.1145/2858796.2858798

[11] Cruz Izu, Carsten Schulte, Ashish Aggarwal, Quintin Cutts, Rodrigo Duran, Mirela Gutica, Birte Heinemann, Eileen Kraemer, Violetta Lonati, Claudio Mirolo, and Renske Weeda. 2019. Fostering Program Comprehension in Novice Programmers - Learning Activities and Learning Trajectories. In *Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education* (Aberdeen, Scotland Uk) *(ITiCSE-WGR '19)*. Association for Computing Machinery, New York, NY, USA, 27–52. https://doi.org/10.1145/3344429.3372501

[12] Manu Kapur. 2016. Examining Productive Failure, Productive Success, Unproductive Failure, and Unproductive Success in Learning. *Educational Psychologist* 51, 2 (2016), 289–299. https://doi.org/10.1080/00461520.2016.1155457

[13] Ville Karavirta, Juha Helminen, and Petri Ihantola. 2012. A mobile learning application for parsons problems with automatic feedback. In *Proceedings of the 12th koli calling international conference on computing education research*. 11–18.

[14] Cazembe Kennedy and Eileen T. Kraemer. 2019. Qualitative Observations of Student Reasoning: Coding in the Wild. In *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education* (Aberdeen, Scotland Uk) *(ITiCSE '19)*. Association for Computing Machinery, New York, NY, USA, 224–230. https://doi.org/10.1145/3304221.3319751

[15] Amruth N Kumar. 2018. Epplets: a tool for solving parsons puzzles. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*. 527–532.

[16] Teemu Lehtinen, Aleksi Lukkarinen, and Lassi Haaranen. 2021. Students Struggle to Explain Their Own Program Code. In *Proceedings of the 26th ACM Conference on Innovation and Technology in Computer Science Education V. 1* (Virtual Event, Germany) *(ITiCSE '21)*. Association for Computing Machinery, New York, NY, USA, 206–212. https://doi.org/10.1145/3430665.3456322

[17] Teemu Lehtinen, André L. Santos, and Juha Sorva. 2021. Let's Ask Students About Their Programs, Automatically. In *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*. 467–475. https://doi.org/10.1109/ICPC52881.2021.00054

[18] Juho Leinonen, Francisco Enrique Vicente Castro, and Arto Hellas. 2022. Time-on-Task Metrics for Predicting Performance. In *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education V. 1* (Providence, RI, USA) *(SIGCSE 2022)*. Association for Computing Machinery, New York, NY, USA, 871–877. https://doi.org/10.1145/3478431.3499359

[19] Juho Leinonen, Paul Denny, and Jacqueline Whalley. 2022. A Comparison of Immediate and Scheduled Feedback in Introductory Programming Projects. In *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education V. 1*. 885–891.

[20] Juho Leinonen, Krista Longi, Arto Klami, and Arto Vihavainen. 2016. Automatic Inference of Programming Performance and Experience from Typing Patterns. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education* (Memphis, Tennessee, USA) *(SIGCSE '16)*. Association for Computing Machinery, New York, NY, USA, 132–137. https://doi.org/10.1145/2839509.2844612

[21] 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. *SIGCSE Bull.* 38, 3 (jun 2006), 118–122. https://doi.org/10.1145/1140123.1140157

[22] Krista Longi, Juho Leinonen, Henrik Nygren, Joni Salmi, Arto Klami, and Arto Vihavainen. 2015. Identification of Programmers from Typing Patterns. In *Proceedings of the 15th Koli Calling Conference on Computing Education Research* (Koli, Finland) *(Koli Calling '15)*. Association for Computing Machinery, New York, NY, USA, 60–67. https://doi.org/10.1145/2828959.2828960

[23] Mike Lopez, Jacqueline Whalley, Phil Robbins, and Raymond Lister. 2008. Relationships between Reading, Tracing and Writing Skills in Introductory Programming. In *Proceedings of the Fourth International Workshop on Computing Education Research* (Sydney, Australia) *(ICER '08)*. Association for Computing Machinery, New York, NY, USA, 101–112. https://doi.org/10.1145/1404520.1404531

[24] Sandra Madison and James Gifford. 2002. Modular Programming. *Journal of Research on Technology in Education* 34, 3 (2002), 217–229. https://doi.org/10.1080/15391523.2002.10782346

[25] José Carlos Paiva, José Paulo Leal, and Álvaro Figueira. 2022. Automated Assessment in Computer Science Education: A State-of-the-Art Review. *ACM Trans. Comput. Educ.* 22, 3, Article 34 (jun 2022), 40 pages. https://doi.org/10.1145/3513140

[26] Dale Parsons and Patricia Haden. 2006. Parson's programming puzzles: a fun and effective learning tool for first programming courses. In *Proceedings of the 8th Australasian Conference on Computing Education-Volume 52*. 157–163.

[27] D. N. Perkins, Chris Hancock, Renee Hobbs, Fay Martin, and Rebecca Simmons. 1986. Conditions of Learning in Novice Programmers. *Journal of Educational Computing Research* 2, 1 (1986), 37–55. https://doi.org/10.2190/GUJT-JCBJ-Q6QU-Q9PL

[28] Seán Russell. 2022. Automated Code Tracing Exercises for CS1. In *Computing Education Practice 2022* (Durham, United Kingdom) *(CEP 2022)*. Association for Computing Machinery, New York, NY, USA, 13–16. https://doi.org/10.1145/3498343.3498347

[29] Jean Salac and Diana Franklin. 2020. If They Build It, Will They Understand It? Exploring the Relationship between Student Code and Performance. In *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education* (Trondheim, Norway) *(ITiCSE '20)*. Association for Computing Machinery, New York, NY, USA, 473–479. https://doi.org/10.1145/3341525.3387379

[30] André Santos, Tiago Soares, Nuno Garrido, and Teemu Lehtinen. 2022. Jask: Generation of Questions About Learners' Code in Java. In *Proceedings of the 27th ACM Conference on on Innovation and Technology in Computer Science Education Vol. 1* (Dublin, Ireland) *(ITiCSE '22)*. Association for Computing Machinery, New York, NY, USA, 117–123. https://doi.org/10.1145/3502718.3524761

[31] Raj Shrestha, Juho Leinonen, Arto Hellas, Petri Ihantola, and John Edwards. 2022. *CodeProcess Charts: Visualizing the Process of Writing Code*. Association for Computing Machinery, New York, NY, USA, 46–55. https://doi.org/10.1145/3511861.3511867

[32] Teemu Sirkiä and Lassi Haaranen. 2017. Improving online learning activity interoperability with Acos server. *Software: Practice and Experience* 47, 11 (2017), 1657–1676. https://doi.org/10.1002/spe.2492

[33] Emil Stankov, Mile Jovanov, and Ana Madevska Bogdanova. 2022. Smart generation of code tracing questions for assessment in introductory programming. *Computer Applications in Engineering Education* (2022). https://doi.org/10.1002/cae.22567

[34] Lasang Jimba Tamang, Zeyad Alshaikh, Nisrine Ait Khayi, Priti Oli, and Vasile Rus. 2021. A Comparative Study of Free Self-Explanations and Socratic Tutoring Explanations for Source Code Comprehension. 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, 219–225. https://doi.org/10.1145/3408877.3432423

[35] Lasang J. Tamang, Rabin Banjade, Jeevan Chapagain, and Vasile Rus. 2022. Automatic Question Generation for Scaffolding Self-explanations for Code Comprehension. In *Artificial Intelligence in Education*, Maria Mercedes Rodrigo, Noburu Matsuda, Alexandra I. Cristea, and Vania Dimitrova (Eds.). Springer International Publishing, Cham, 743–748. https://doi.org/10.1007/978-3-031-11644-5_77

[36] Arto Vihavainen, Juha Helminen, and Petri Ihantola. 2014. How Novices Tackle Their First Lines of Code in an IDE: Analysis of Programming Session Traces. In *Proceedings of the 14th Koli Calling International Conference on Computing Education Research* (Koli, Finland) *(Koli Calling '14)*. Association for Computing Machinery, New York, NY, USA, 109–116. https://doi.org/10.1145/2674683.2674692

[37] Anusha Vimalaksha, Abhijit Prekash, Viraj Kumar, and Gowri Srinivasa. 2021. DiGen: Distractor Generator for Multiple Choice Questions in Code Comprehension. In *2021 IEEE International Conference on Engineering, Technology & Education (TALE)*. 1073–1078. https://doi.org/10.1109/TALE52509.2021.9678662

[38] Nathaniel Weinman, Armando Fox, and Marti A Hearst. 2021. Improving instruction of programming patterns with faded parsons problems. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. 1–4.