Assessing Live Programming for Program Comprehension

Oliver Graf*
olgraf@ethz.ch
Department of Computer Science
ETH Zürich
Zürich, Switzerland

Sverrir Thorgeirsson*
sverrir.thorgeirsson@inf.ethz.ch
Department of Computer Science
ETH Zürich
Zürich, Switzerland

Zhendong Su
zhendong.su@inf.ethz.ch
Department of Computer Science
ETH Zürich
Zürich, Switzerland

ABSTRACT

Previous research on the effects of live program composition in computer science education has shown mixed results; while live programming is well-received by students and can improve the program composition process in some contexts, the resulting programs may be hard to understand, potentially making the paradigm unfeasible for collaborative and general-purpose programming. In this paper, we explore to what extent programs created in Algot, a live programming language, can be understood by tertiary-level students. We conducted an experimental, within-subjects study ($n = 41$) measuring how well students at this level could comprehend programs composed in Algot and Python. We asked our participants to explain the programs and answer questions on them related to tracing, reverse tracing, conceptual extrapolations, and (optionally) time complexity. Despite the participants' lack of familiarity with Algot, students performed better after viewing most programs in Algot than Python, but primarily for problems involving trees and matrices. Our results contribute to the body of research on live programming in computer science (CS) education and complement recent research on the benefits of Algot for program composition, suggesting that Algot can be useful as a more general learning resource in CS tertiary education.

CCS CONCEPTS

• **Human-centered computing** → *Visualisation systems and tools*; *Empirical studies in visualisation*; • **Applied computing** → **Interactive learning environments**.

KEYWORDS

visual programming, live programming, programming by demonstration, program comprehension, tertiary education

1 INTRODUCTION

Live programming refers to the process where changes in source code are immediately reflected in the program's execution, offering real-time feedback of the program's behaviour. This approach stands in contrast to traditional programming methods where code modifications necessitate a separate compile-and-run phase to observe changes [24, 33, 34]. Its purported advantage lies in its ability to enhance understanding and debugging of code by providing instant visual feedback, which can be particularly helpful for beginners in both exploring ideas and in grasping programming concepts and logic [7].

In general, the focus with live programming is on the program composition process rather than the resulting program itself. Creating programs is sometimes only a secondary goal of live programming [31], with the primary goal to create a one-time effect where the program plays no role and is discarded afterwards. In computer science (CS) education, besides its perceived benefit in making program composition easier, researchers have also explored the effect of live programming on learning. For instance, the live code visualisation tool Python Tutor, which has been used by millions of users [9], has been the subject of several such studies [18, 19, 27]. However, studies on the program comprehension and usefulness of the programs themselves that are generated from live programming are scarce and limited to specific domains such as robotics education; one such study found that programs written in a live language were no better for comprehension than ones written in a non-live language [4].

In this paper, we explore how well tertiary-level students can understand programs that were composed via live programming for solving a curated set of introductory CS exercises. To do so, we chose programs written in Algot, a live, visual language that supports direct manipulation on the program state via programming by demonstration [36, 39]. We chose Algot for two reasons. First, controlled, experimental studies indicate that Algot performs well in certain contexts when it comes to program composition [37] and learning outcomes [35], but to our best knowledge, no studies are available on how well existing Algot programs can be understood in the interactive, step-by-step presentation style that the language uses. Further clarity on this question would help establish whether Algot's effectiveness is limited to simple program composition activities or whether it can provide a more comprehensive solution in CS education. Second, as Algot is specifically designed for live programming, its use in this study will give insight into whether live programming can help or hinder program understanding.

We begin by explaining how we equipped Algot with a novel, example-based programming system as suggested by the Algot authors in the future work section of their 2022 paper [39]. Then we report on our experimental, within-subjects study on 41 tertiary-level students who were provided with programs written in Algot and Python and asked to solve different types of questions pertaining to the programs. Five of the six programs we used were solutions to CodeCheck problems [15], an online repository of introductory CS coding problems that has been used frequently in CS1 research in recent years [1, 6, 38].

The primary research questions for our study were:

RQ1 Is Algot's experimental way of program presentation a viable alternative to traditional text-based presentation?

RQ2 Do tertiary-level students report enjoying solving comprehension questions more in Algot or Python?

*Oliver Graf and Sverrir Thorgeirsson are co-primary authors.

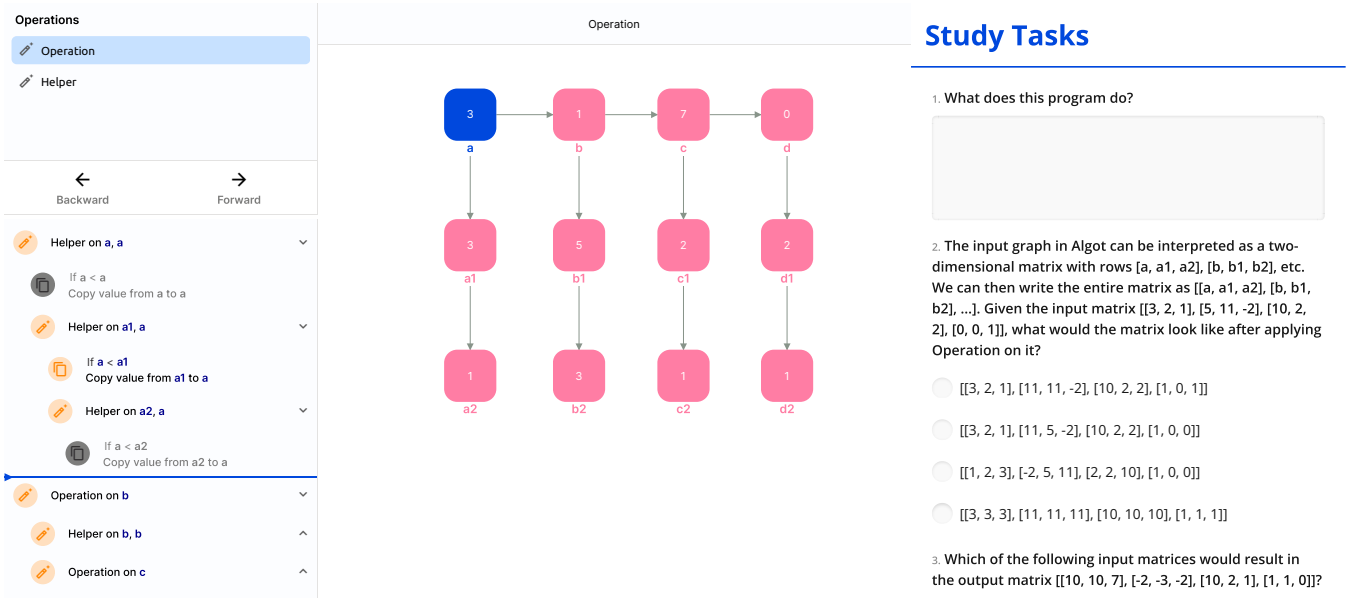

Figure 1: A screenshot of the Algot system used in the study. It shows the Algot version of task 5 (Matrix Update). Currently, the recursive calls to `Helper` are opened, as is the first recursive call to `Operation`. The screenshot is slightly cropped to reduce whitespace.

We hypothesised that the participants of the study would perform better or equally well using the Algot environment, as its live and example-based style may make program comprehension easier, and that they would also find it more enjoyable to use.

2 BACKGROUND AND RELATED WORK

2.1 Live Programming

In contrast to traditional text-based programming, which requires programmers to maintain a mental model of the program state as the code changes, live programming offers an immediate and interactive feedback loop, allowing a real-time representation of state changes when code is modified. However, the effectiveness of live systems in computer science education is mixed and implementation-dependent; for example, some have expressed concerns that live systems may overload users with information they do not need [25]. Their effectiveness can also be task-dependent; one study on the spreadsheet language Forms/3 found that liveness helped programmers with complex tasks, while it did not help or even hindered them when solving simpler tasks [40]. In a robotics education setting, a more recent study on a small number of participants failed to find any positive correlation between a live programming environment and program comprehension, even though liveness led to a more enjoyable development experience [4].

One of the most widely used live programming environments is Python Tutor, a Python environment that allows users to iteratively step through programs and observe the effects of individual instructions on a visualisation of the program state [10]. One study found that use of the tool (when embedded within another software) was correlated with higher learning performance [2, 16], but another study on the tool was less conclusive [17].

Some challenging topics in CS education can potentially be easier to teach in live systems. As an example, recursion is traditionally regarded as a very difficult topic to understand for novice programmers, and is often cited as a problem area in CS1/CS2 education [5]. However, studies have found that teaching recursion first does not lead to worse outcomes [26], and that recursive programs are sometimes better understood than iterative programs by more advanced programmers [3]. A live presentation of concrete program states has the potential to enhance students' ability to understand and apply concepts like recursion, as concrete examples tend to improve educational outcomes [41].

2.2 Algot

Algot is a visual, graph-based programming language which supports direct manipulation and programming by demonstration [36, 39]. In Algot, the program state is visually represented as a graph. The state graph can, for example, be a set of singleton nodes, linked lists, trees, or matrices, and it can be modified by applying operations on individual nodes, for example by deleting them, changing their values, or adding new edges between them, all with an immediate visual effect.

In Algot, certain operations are available in the system by default. User-defined operations are created by demonstrating base operations or other user-defined operations on specific input, which means that the programmer can define a new program simply by executing a list of operations in sequence. Since Algot is a live programming language, the structure of the input can visibly change or new nodes may be created while the program is being composed. The programmer can choose to apply operations conditionally based on the results of *queries*, which are yes-no questions about a given set of nodes, such as "Does this node have outgoing

edges?" or "Is the value of node B less than the value of node C?". Iteration is achieved via recursion, which means that the programmer calls the operation that is being defined. More information on the language is available in a 2022 paper [39].

We are aware of two empirical studies on the program composition mechanism in Algot. A comparative, controlled study on secondary school students found that they exhibited a better understanding of the recursion mechanism after programming in Algot rather than after programming with the Make-a-Block feature in Scratch [35]. Another comparative study on Algot and Python found that undergraduate students experienced lower cognitive load when programming in Algot, according to physiological measurements and a validated survey, and performed significantly better on the simple programming tasks that were used [37].

3 EXAMPLE-BASED PROGRAMMING IN ALGOT

After getting access to Algot, we implemented additional support for liveness ahead of our study. For example, we made adjustments to the program composition process in order to allow Algot programmers to directly see the effects of their programs on example values instead of relying on an abstract approximation. We call this example-based programming (not to be confused with program synthesis, as the term is sometimes used [29]), similar to a recent system for text-based programming called *Babylonian programming* [20, 28, 30]. In Algot, this means that the programmer is always working with concrete values of their own choice when defining new operations. For example, Figure 2 shows how one might define an operation on two input lists where the objective is to find the index of the first element where the lists differ. Before defining the operation, the programmer may choose to populate the input lists with the values `[2,5,3,4]` and `[2,5,1,4]`, allowing one to see in real-time how the value of the third argument *c* changes throughout the definition of the operation from 0 to its correct value 2. Our belief is that by making programs more tangible and contextual, our implementation of example-based programming can aid programmers with programming. It also requires the programmer to define and use at least one test case for any given operation.

We also added support for programmers to step through the execution of the program in order to observe the current state at any given time, similarly to adding break points in a textual debugger. This allows Algot programmers to examine the effects of recursive operations in a more fine-grained manner. Figure 1 shows a screenshot of the Algot system used in the user study. The left-hand side contains a textual representation of the program, including currently opened recursive calls. The center of the screen shows the graphical representation of the example program state. For our study, some features were deliberately omitted from the testing environment, such as the opportunity to change the example values at any point; we believed that this might make, for example, tracing questions very simple to solve.

4 METHOD

After receiving ethics approval from our institution, we conducted a controlled, experimental study under a within-subjects design where each participant was asked to solve questions on six tasks

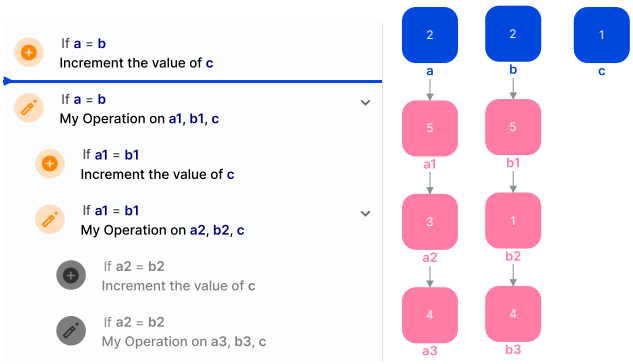

Figure 2: Implementation of the task *Two Lists* in Algot. Operations that are not executed due to the queries being evaluated as false are displayed in gray. Note the example values used for the task (see the visualisation on the right).

in each language. The tasks were to be solved sequentially with each program presented together with the corresponding questions. To minimise carryover effects, each participant first solved three randomly chosen tasks in one language and then the other three in the other language. This was then followed by the original three tasks in the second language and the next three in the first language in reverse order.

The study took place in Zürich, Switzerland. 43 participants were recruited. The inclusion criteria was to have successfully completed at least one introductory computer science course, to be a tertiary education student, and to have some experience with the programming language Python. As a pretest, students solved a set of Python exercises in different categories (writing, tracing, reverse tracing, and Parson problems), loosely adapted from a study on the hierarchy of computer programming skills [22] that was recently replicated [8]. We also included two questions from the Basic Recursion Concept Inventory [11] to understand how familiar the participants were with recursion.

After the pretest, participants watched a brief video introduction on how to navigate the Algot environment and received a double-sided handout they could use for the remaining part of the study, containing some general useful information on time complexity, graphs and trees, Python syntax (for example tuple unpacking and the meaning of special syntax such as `l[-1]` and `l[1:3]`) and the naming convention for Algot nodes. Afterwards, participants were asked to solve the six tasks of the study in both languages (see the next section). Following this, participants rated the Algot programming environment using three scales of the User Experience Questionnaire Plus (UEQ+) [21], meaning that the participants were given four different adjective-antonym pairs for each scale and asked to select a number from 1 to 7 indicating their agreement with each word in the pair. The three scales selected were usefulness, clarity and novelty, the first two of which are recommended by the UEQ+ authors for the evaluation of programming systems. Afterwards, participants were asked to rate their enjoyment of working with Algot and Python on a 5-point scale, and lastly to offer optional free-form comments on their study experience.

The participants were given up to 30 minutes to complete the pretest and two hours in total for the entire study (including the pretest and the posttest). The study was conducted over two different sessions occurring on the same day, with about half of the participants attending each session. Participants were compensated with 25 CHF for each hour, or 50 CHF in total.

Study Tasks

(1) **Simple String**: If the first and last characters of a string s are the same, remove them.
(2) **Character Swap**: In a string s, swap the characters adjacent to all occurrences of a character c.
(3) **Two Lists**: For two input lists, return the first index where the lists have different values.
(4) **Closest Number**: For an integer x and an input list l of integers, return the position of the first element in l that is closest to x.
(5) **Matrix Update**: In a two-dimensional list m of integers, update the value of the first element in each row of l with the maximum value of the row.
(6) **Heap Deletion**: In an input binary max-heap, remove the root r and restore the heap condition.

Figure 3: The six tasks that were selected for the study. The first five tasks were selected from the CodeCheck list of Python exercises [15]. Each task was implemented in Algot and Python without comments and with non-descriptive functions and variable names.

4.1 Tasks

A description of the tasks used for the study can be found in Figure 3. The code for each task was as identical in Algot and Python as possible, with minor adjustments to reflect different variable names and slightly different program structures used by the different languages. To make the comparison as fair as possible, we used recursion in both languages.

For each task, the participants were asked four questions: (i) to provide an explanation in plain English of the effect of the code; (ii) to select the output produced by the program for some concrete input (tracing), which were distinct from those used to define the Algot programs; (iii) to select inputs that produce a given output (reverse tracing), a more challenging problem type than tracing that requires a similar but distinct thought process [12]; (iv) to answer a conceptual question related to the program, for example how the algorithm might be adjusted to meet a different criteria. We also included a fifth bonus question on algorithmic time complexity, which was optional since we did not expect all participants to be familiar with this topic.

The first five tasks were taken from CodeCheck problems [15], an online repository of introductory CS coding problems. Our aim was to select tasks involving different types and data structures commonly used in CS1, such as strings, lists, matrices, and trees, and to have at most one task from each CodeCheck category. We made a minor adjustment to the fifth task selected (*Matrix Update*) to simplify the implementation in Algot and Python. We also added a separate sixth task on a canonical data structure operation involving heaps since there are no CodeCheck questions on trees. With a diverse representation of tasks, we also hoped to discover more about the effects of Algot's graphical representation of structures other than singleton nodes, linked lists, or matrices.

Figure 2 and Listing 1 show our implementation of the program for the third task (*Two Lists*) in Algot and in Python, respectively.

```
def f(l, m):
    def aux(l, m, i):
        if l[0] != m[0]:
            return i
        return aux(l[1:], m[1:], i+1)
    return aux(l, m, 0)
```

Listing 1: Implementation of *Two Lists* in Python

4.2 Data Analysis

After grading the students' performance, we used the Wilcoxon's signed-rank version of the Bayesian Paired Samples t-test with 10,000 samples to analyse the difference between the mean percentage of tasks solved in each language for each participant. This non-parametric statistical test is useful in this context since it does not require any assumptions about the distributions of the data and is suitable for small sample sizes [14]. The resulting Bayesian Factors (BF_{+0}) can be interpreted as the strength of the alternative hypothesis (that students would perform better in Algot) over the null hypothesis, with values above 3 typically considered moderate evidence and values over 10 typically considered strong evidence. All results were computed with the statistical software JASP [23].

We also applied simple quantitative analysis on the free-form feedback, using thematic coding to identify common themes and patterns in the responses. This way, we hoped to gain a contextual understanding of the qualitative results, in particular of students' perceptions of Algot and how it compares to Python for program comprehension.

5 RESULTS

Of the 43 participants initially recruited, 41 participants completed the study and two withdrew or did not submit any solutions. The participants were between the age of 18 and 36 with a median age of 23. There were 17 female and 24 male participants. None of them had used Algot before. The participants performed reasonably well on the Python pretest, solving on average 62.7% of the questions correctly, including 50.0% on the two questions on recursion.

A detailed analysis of the performance of the participants can be found in Table 2. Overall, the participants performed significantly better in Algot than in the Python environment, with an effect size (Cohen's d for repeated measures) of 0.67 and a Bayes Factor of 4860, indicating very strong support for the alternative hypothesis. Much of the difference in the results can be attributed to the performance on the fifth and sixth tasks (*Matrix Update* and *Heap Deletion*). The participants did perform better in Python than Algot on the second and fourth tasks (*Character Swap* and *Closest Number*), although by a smaller margin. These results match our hypothesis from Section 1 that participants would perform better in Algot overall. We also found that the participants performed better across every question

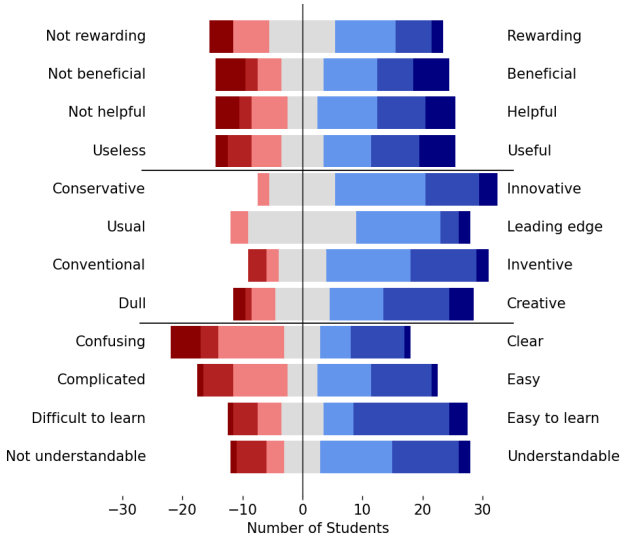

Figure 4: A visualisation of the UEQ+ survey results. Participants were asked to rate the following properties of the Algot environment: usefulness (top), novelty (middle) and perspicuity (bottom). The colors gradients represent scores from -3 (dark red) to 3 (dark blue).

type, with substantial improvements in the explaining and tracing skills (Cohen's *d* for repeated measures 0.61 and 0.62).

24 participants (59%) chose to answer the bonus questions on time complexity. The participants performed similarly well across languages (a performance of 54.3% in Algot and 52.6% in Python).

The results on the UEQ+ can be found in Figure 4. The ratings were generally positive, with only one adjective-pair (confusing-clear) rated below the middle point of 4. However, despite many students finding Algot confusing for program comprehension, they also found it understandable and easy to learn, and more found it to be helpful and useful than those who did not. The novelty of the system was rated most strongly, with a majority of participants finding the language at least somewhat creative and innovative. Additionally, the participants reported that they enjoyed solving the tasks in Algot more than in Python (Cohen's *d* for repeated measures 0.26; see more in Table 2).

21 participants chose to leave optional free-form feedback. After reading through them, we assigned three thematic codes: (i) *Language Feedback*, (ii) *Study Feedback*, and (iii) *Other*. Of the eight instances of language feedback, four responses contained only positive remarks on Algot such as "The Algot environment is great. The execution is very clear to understand and follow", "Some of the codes in Python, I was not able to understand. However, same codes in Algot were clear for me" and "i really liked algot. i wish i had more time to practice with it", three comments contained reserved positive feedback on Algot, but with the notion that the system would be hard for program composition or only useful when used together with Python (e.g., "i quite liked Algot it made understanding the program a lot easier but it's only beneficial as an extra to the codes"), and one comment was critical of Algot ("Not sure how useful this would be. I feel like looking at the original python program is less confusing sometimes"). Overall, the language feedback appears congruent with the UEQ+ results and the self-reported difference in enjoyment between the two languages.

11 responses were on the study design, thereof six remarks on minor technical parts such as perceived issues with some of the code, visualisation, or its presentation, three remarks were on the tasks being too difficult, and two others could be characterised as objections to the within-subjects design of the study. The two remaining comments that we categorised as *Other* were on the Python background of the student.

We also analyzed a few other metrics to help contextualise the results. We found that participants spent, on average, roughly an equal amount of time on the Algot and Python tasks (6.88 and 6.80 minutes per task, respectively). We also looked into whether the performance improved on a given task when solving it later in the other language, and found, surprisingly, that the performance was slightly worse, which may possibly be attributed to fatigue. We found a strong correlation between performance in the pretest and the main study for both the explaining and tracing skills (see Table 1).

Lastly, we note some minor experimental and implementation errors that occurred. During the first session, we found that the two programs on the *Two Lists* task were not as similar as we had intended, an issue that we addressed ahead of the second session. The performance of the participants in the first session on this task was therefore not included in the analysis. We also noted that the conceptual question on the *Heap Deletion* task was not displayed in both language environments, so to maintain fairness, we did not include the performance on this question for this task in our analysis. Also, one participant accidentally skipped a task due to a keybinding in the user interface that we were unaware of, so their performance on this task was excluded from our analysis.

	Mean (PT)	Corr PT→A	Corr PT→P
Overall	62.7	0.476	0.460
Explaining	52.4	0.571	0.556
Tracing	73.2	0.104	0.286

Table 1: The correlations between performances on the pretest (PT) and the Algot (A) and Python (P) environments on the different skills examined. The correlation columns contain the Pearson correlation coefficient.

6 DISCUSSION

In general, the performance results were aligned with our hypothesis that students would experience better program comprehension when viewing the tasks using Algot than Python. The strength of the results, however, varied between the tasks, with the two tasks on trees and matrices appearing to be better suited for Algot than the four others. It is possible that Algot is well-suited for tasks on graph-based or multidimensional data structures. In general, we find the results to be particularly convincing considering that all participants had some experience with Python, but were encountering Algot for the first time.

	Mean (A)	Mean (P)	SD (A)	SD (P)	Cohen's d (SE)	BF_{+0}
Overall	**58.2**	47.7	20.2	21.2	**0.667** (0.13)	4860
Task 1 (Simple String)	**78.2**	71.2	25.8	24.4	**0.218** (0.16)	1.12
Task 2 (Character Swap)	49.8	**57.2**	33.7	34.3	**-0.252** (0.14)	0.07
Task 3 (Two Lists)	**71.1**	59.7	30.5	33.5	**0.313** (0.17)	2.82
Task 4 (Closest Number)	28.4	**31.9**	31.7	36.4	**-0.118** (0.10)	0.12
Task 5 (Matrix Update)	**74.2**	41.9	34.4	40.5	**0.623** (0.23)	253
Task 6 (Heap Deletion)	**50.6**	29.5	32.0	28.8	**0.560** (0.21)	120
Code explanation	**57.1**	48.6	24.5	25.9	**0.607** (0.09)	776
Tracing	**68.6**	56.1	23.0	23.6	**0.619** (0.15)	305
Reverse tracing	**43.3**	32.1	22.9	23.7	**0.446** (0.18)	30.1
Conceptual questions	**63.9**	58.7	20.0	27.5	**0.203** (0.17)	0.385
Enjoyment	**3.51**	3.12	1.12	1.05	**0.258** (0.221)	1.249

Table 2: The student performance on the Algot (A) and Python (P) tasks, expressed as percentage points of the maximum score. SD is standard deviation, SE refers to the standard error for Cohen's d for repeated measures, and BF_{+0} is the Bayes Factor under the alternative hypothesis that score(A) > score(P). The bottom row contains the rating of how much the participants enjoyed the Algot and Python environments (ranking from 1 to 5).

We do not consider the results to reflect negatively on Python as a programming language in CS education. Many programming environments support the visualisation of Python code, for example Python Tutor [10] or visual debuggers [13, 32], and a direct comparison between Algot and such environments may well have given different results. The objective of the study was not to carry out such a comparison, but rather to find whether Algot's program representation is viable by comparing it against an established baseline, and secondly, to understand better the effects of liveness itself on program comprehension. On the first point, our results indicate that Algot's program representation is viable, and on the second point, we believe our study contributes to research on "the productivity advantages of live programming for understanding existing code," as put by Campusano [4], whose small study on the topic was the first of this type. To our knowledge, our experiment is only the second such study, and the first to find clear evidence in favour of liveness on program comprehension.

6.1 Threats to Validity

Regarding the ecological validity of the study, we note that it was conducted in an environment that is different from the typical setting in which comprehension skills are typically applied, as program comprehension is usually relevant when students or developers work with larger systems containing more code. Furthermore, the participants would likely be using their own software and hardware instead of those provided during the study.

Some priming effects may have occurred via the Algot video tutorial and the Python pretest, but we find it unlikely that this significantly shifted the direction or strength of the results. Some of the implementation errors described in the previous section may also have had a small effect on the results.

The results of the study are limited to the selected programming tasks and exercises. While we aimed for a balanced range of tasks, it is possible that the results would be different under a different selection, for example with more complex tasks, or ones that emphasised other topics commonly taught in introductory CS.

7 CONCLUSION

Using typical CS1 tasks, this study presents evidence that programs written in Algot, an experimental live programming environment based on programming by demonstration, are as easy to comprehend by tertiary-level students as programs written in the textual language Python. Algot worked particularly well on tasks related to matrices and trees, underscoring its potential effectiveness in visualizing and understanding graph-based data structures. The participants, despite their inexperience with Algot and familiarity with Python, demonstrated a slight preference for Algot in terms of enjoyment, and rated it highly on most usability metrics. These results indicate that at least some live programming environments can enhance program comprehension in CS1.

REFERENCES

[1] Nimisha Agarwal, Viraj Kumar, Arun Raman, and Amey Karkare. 2023. A Bug's New Life: Creating Refute Questions from Filtered CS1 Student Code Snapshots. In *Proceedings of the ACM Conference on Global Computing Education Vol 1*. 7–14.

[2] Christine Alvarado, Briana B Morrison, Barbara Ericson, Mark Guzdial, Brad Miller, and David L Ranum. 2012. Performance and use evaluation of an electronic book for introductory Python programming. (2012).

[3] Alan C Benander, Barbara A Benander, and Howard Pu. 1996. Recursion vs. iteration: An empirical study of comprehension. *Journal of Systems and Software* 32, 1 (1996), 73–82.

[4] Miguel Campusano, Alexandre Bergel, and Johan Fabry. 2016. Does Live Programming Help Program Comprehension?. In *A user study with Live Robot Programming. In Proceedings of the 7th International Workshop on Evaluation and Usability of Programming Languages and Tools. ACM, Amsterdam, Netherlands.*

[5] Nell B Dale. 2006. Most difficult topics in CS1: results of an online survey of educators. *ACM SIGCSE Bulletin* 38, 2 (2006), 49–53.

[6] Paul Denny, Viraj Kumar, and Nasser Giacaman. 2023. Conversing with copilot: Exploring prompt engineering for solving cs1 problems using natural language. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*. 1136–1142.

[7] Johan Fabry. 2019. The meager validation of live programming. In *Companion Proceedings of the 3rd International Conference on the Art, Science, and Engineering*

of Programming. 1–6.

[8] Max Fowler, David H Smith IV, Mohammed Hassan, Seth Poulsen, Matthew West, and Craig Zilles. 2022. Reevaluating the relationship between explaining, tracing, and writing skills in CS1 in a replication study. *Computer Science Education* 32, 3 (2022), 355–383.

[9] Philip Guo. 2021. Ten million users and ten years later: Python tutor's design guidelines for building scalable and sustainable research software in academia. In *The 34th Annual ACM Symposium on User Interface Software and Technology*. 1235–1251.

[10] Philip J Guo. 2013. Online python tutor: embeddable web-based program visualization for cs education. In *Proceeding of the 44th ACM technical symposium on Computer science education*. 579–584.

[11] Sally Hamouda, Stephen H Edwards, Hicham G Elmongui, Jeremy V Ernst, and Clifford A Shaffer. 2017. A basic recursion concept inventory. *Computer Science Education* 27, 2 (2017), 121–148.

[12] 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 *Proceedings of the 17th ACM conference on international computing education research*. 115–126.

[13] Juha Helminen et al. 2009. Jype–an education-oriented integrated program visualization, visual debugging, and programming exercise tool for python. *Master's Thesis* 1 (2009).

[14] Myles Hollander, Douglas A Wolfe, and Eric Chicken. 2013. *Nonparametric statistical methods*. John Wiley & Sons.

[15] Cay S. Horstmann. n. d.. CodeCheck Python Exercises. https://horstmann.com/codecheck/python-questions.html Retrieved December 1, 2023.

[16] Ruanqianqian Huang, Kasra Ferdowsi, Ana Selvaraj, Adalbert Gerald Soosai Raj, and Sorin Lerner. 2022. Investigating the impact of using a live programming environment in a CS1 course. In *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education-Volume 1*. 495–501.

[17] Oscar Karnalim and Mewati Ayub. 2017. The effectiveness of a program visualization tool on introductory programming: A case study with PythonTutor. *CommIT (Communication and Information Technology) Journal* 11, 2 (2017), 67–76.

[18] Oscar Karnalim and Mewati Ayub. 2017. The use of python tutor on programming laboratory session: Student perspectives. *Kinetik: Game Technology, Information System, Computer Network, Computing, Electronics, and Control* (2017), 327–336.

[19] Oscar Karnalim and Mewati Ayub. 2018. A Quasi-Experimental Design to Evaluate the Use of PythonTutor on Programming Laboratory Session. *International Journal of Online Engineering* 14, 2 (2018).

[20] Eva Krebs, Toni Mattis, Patrick Rein, and Robert Hirschfeld. 2023. Toward Studying Example-Based Live Programming in CS/SE Education. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Programming Abstractions and Interactive Notations, Tools, and Environments*. 17–24.

[21] Bettina Laugwitz, Theo Held, and Martin Schrepp. 2008. Construction and evaluation of a user experience questionnaire. In *HCI and Usability for Education and Work: 4th Symposium of the Workgroup Human-Computer Interaction and Usability Engineering of the Austrian Computer Society, USAB 2008, Graz, Austria, November 20-21, 2008. Proceedings 4*. Springer, 63–76.

[22] 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*. 101–112.

[23] Jonathon Love, Ravi Selker, Maarten Marsman, Tahira Jamil, Damian Dropmann, Josine Verhagen, Alexander Ly, Quentin F Gronau, Martin Šmíra, Sacha Epskamp, et al. 2019. JASP: Graphical statistical software for common statistical designs. *Journal of Statistical Software* 88 (2019), 1–17.

[24] Sean McDirmid. 2007. Living it up with a live programming language. *ACM SIGPLAN Notices* 42, 10 (2007), 623–638.

[25] Sean McDirmid. 2013. Usable live programming. In *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software*. 53–62.

[26] Claudio Mirolo. 2012. Is iteration really easier to learn than recursion for CS1 students?. In *Proceedings of the ninth annual international conference on International computing education research*. 99–104.

[27] Monika Mladenović, Žana Žanko, and Marin Aglić Čuvić. 2021. The impact of using program visualization techniques on learning basic programming concepts at the K–12 level. *Computer Applications in Engineering Education* 29, 1 (2021), 145–159.

[28] Fabio Niephaus, Patrick Rein, Jakob Edding, Jonas Hering, Bastian König, Kolya Opahle, Nico Scordialo, and Robert Hirschfeld. 2020. Example-based live programming for everyone: Building language-agnostic tools for live programming with lsp and graalvm. In *Proceedings of the 2020 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. 1–17.

[29] Hila Peleg, Sharon Shoham, and Eran Yahav. 2018. Programming not only by example. In *Proceedings of the 40th International Conference on Software Engineering*. 1114–1124.

[30] Patrick Rein, Jens Lincke, Stefan Ramson, Toni Mattis, Fabio Niephaus, and Robert Hirschfeld. 2019. Implementing babylonian/s by putting examples into contexts: Tracing instrumentation for example-based live programming as a use case for context-oriented programming. In *Proceedings of the 11th ACM International Workshop on Context-Oriented Programming*. 17–23.

[31] Patrick Rein, Stefan Ramson, Jens Lincke, Robert Hirschfeld, and Tobias Pape. 2018. Exploratory and live, programming and coding: a literature study comparing perspectives on liveness. *arXiv preprint arXiv:1807.08578* (2018).

[32] Joël Schneider. 2020. *Design and Implementation of a Graphics Window and Debugger for WebTigerJython*. Master's thesis. ETH Zurich.

[33] Ben Swift, Andrew Sorensen, Henry Gardner, and John Hosking. 2013. Visual code annotations for cyberphysical programming. In *2013 1st International Workshop on Live Programming (LIVE)*. IEEE, 27–30.

[34] Steven L Tanimoto. 2013. A perspective on the evolution of live programming. In *2013 1st International Workshop on Live Programming (LIVE)*. IEEE, 31–34.

[35] Sverrir Thorgeirsson, Lennart Lais, Theo Weidmann, and Zhendong Su. 2024. Recursion in Secondary Computer Science Education: A Comparative Study of Visual Programming Approaches. In *Proceedings of the 55th ACM Technical Symposium on Computer Science Education (SIGCSE 2024)*. Portland, Oregon. In Press.

[36] Sverrir Thorgeirsson and Zhendong Su. 2021. Algot: an educational programming language with human-intuitive visual syntax. In *2021 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 1–5.

[37] Sverrir Thorgeirsson, Theo Weidmann, Karl-Heinz Weidmann, and Zhendong Su. 2024. Comparing Cognitive Load Among Undergraduate Students Programming in Python and the Visual Language Algot. In *Proceedings of the 55th ACM Technical Symposium on Computer Science Education (SIGCSE 2024)*. Portland, Oregon. In Press.

[38] Varshini Venkatesh, Vaishnavi Venkatesh, and Viraj Kumar. 2023. Evaluating Copilot on CS1 Code Writing Problems with Suppressed Specifications. In *Proceedings of the 16th Annual ACM India Compute Conference*. 104–107.

[39] Theo B Weidmann, Sverrir Thorgeirsson, and Zhendong Su. 2022. Bridging the Syntax-Semantics Gap of Programming. In *Proceedings of the 2022 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. 80–94.

[40] Eric M Wilcox, J William Atwood, Margaret M Burnett, Jonathan J Cadiz, and Curtis R Cook. 1997. Does continuous visual feedback aid debugging in direct-manipulation programming systems?. In *Proceedings of the ACM SIGCHI Conference on Human factors in computing systems*. 258–265.

[41] Cheng-Chih Wu, Nell B Dale, and Lowell J Bethel. 1998. Conceptual models and cognitive learning styles in teaching recursion. In *Proceedings of the twenty-ninth SIGCSE technical symposium on Computer science education*. 292–296.