Map, Filter, and Conquer: A Visual Tool for Learning Higher-Order Functions

Silvan Renggli
sili.renggli@gmail.com
ETH Zurich
Zurich, Switzerland

Sverrir Thorgeirsson
sverrir.thorgeirsson@inf.ethz.ch
ETH Zurich
Zurich, Switzerland

Theo B. Weidmann
theo.weidmann@inf.ethz.ch
ETH Zurich
Zurich, Switzerland

Zhendong Su
zhendong.su@inf.ethz.ch
ETH Zurich
Zurich, Switzerland

Abstract

Higher-order functions are increasingly common in modern programming languages, yet there is a shortage of evidence-based tools and teaching strategies to help students learn them effectively. We introduce a visual tool that lets learners construct, view, and execute higher-order functions using direct manipulation and programming by demonstration. To evaluate its effectiveness, we conducted a randomized, within-subjects study with 27 university students, comparing our tool against Python as a control. The results show that students performed significantly better and reported lower cognitive load when solving simple problems with our tool. However, both groups showed similar performance on tasks that involved mapping input-output pairs to the correct higher-order function. Our findings suggest that visual, direct-manipulation tools can help students develop stronger procedural knowledge of higher-order functions, although additional scaffolding may be needed to foster deeper conceptual understanding.

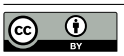
CCS Concepts

• **Social and professional topics** → **Computing education**; • **Human-centered computing** → **Empirical studies in HCI**.

Keywords

higher-order functions, functional programming, visual programming, live programming, direct manipulation, undergraduate education

ACM Reference Format:
Silvan Renggli, Sverrir Thorgeirsson, Theo B. Weidmann, and Zhendong Su. 2025. Map, Filter, and Conquer: A Visual Tool for Learning Higher-Order Functions. In *Proceedings of the 30th ACM Conference on Innovation and Technology in Computer Science Education V. 1 (ITiCSE 2025), June 27-July 2, 2025, Nijmegen, Netherlands.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3724363.3729111



ITiCSE 2025, Nijmegen, Netherlands

ACM ISBN 979-8-4007-1567-9/2025/06
https://doi.org/10.1145/3724363.3729111

1 Introduction

Functional programming has gained renewed interest in modern software development because of its potential to produce reliable, scalable, and maintainable systems [8]. Many traditionally imperative languages now incorporate functional programming constructs—such as Python's `lambda` expressions and JavaScript's `map` and `reduce`—which means that there is a growing need for computer science students and software engineers to develop proficiency in these techniques. However, research indicates that learners frequently encounter difficulties with fundamental functional programming concepts, including recursive functions [16, 18], pattern matching [22], and higher-order functions [1] (HOFs).

Among these core functional concepts, we consider HOFs particularly important to computing education for two reasons. First, by discovering effective pedagogical approaches and tools that foster a deeper understanding of HOFs, students will develop a stronger foundation for more advanced topics in functional programming. Second, HOFs have also become increasingly important in themselves; for example, in concurrency and parallel programming, HOFs enable developers to abstract common coordination patterns and support complex interactions among multiple processes. They also play a pivotal role in data-centric applications, where constructs such as `map`, `filter`, and `reduce` are frequently used for batch and real-time processing tasks.

In this paper, we introduce a system that allows students to learn about and implement HOFs visually without the need for any textual code. To the best of our knowledge, this system is the first of its kind to achieve that. The purpose behind our system is to make the topic more accessible for novices and to lower the barrier to entry, to support experimentation, to provide fast and live visual feedback so as to reduce misconceptions, and to promote active learning pedagogy where students can immediately start working with the subject without much preparatory work on behalf of the instructor. To achieve this, we built our system as an extension to the visual programming language Algot. We found that Algot is a suitable choice given the success of other Algot extensions [6], that controlled, experimental studies have found evidence that the language is helpful for teaching recursion compared to other visual alternatives [18], that students programming in Algot experience lower or comparable cognitive load than when programming in Python [20, 21], and that it works effectively for program comprehension [7].

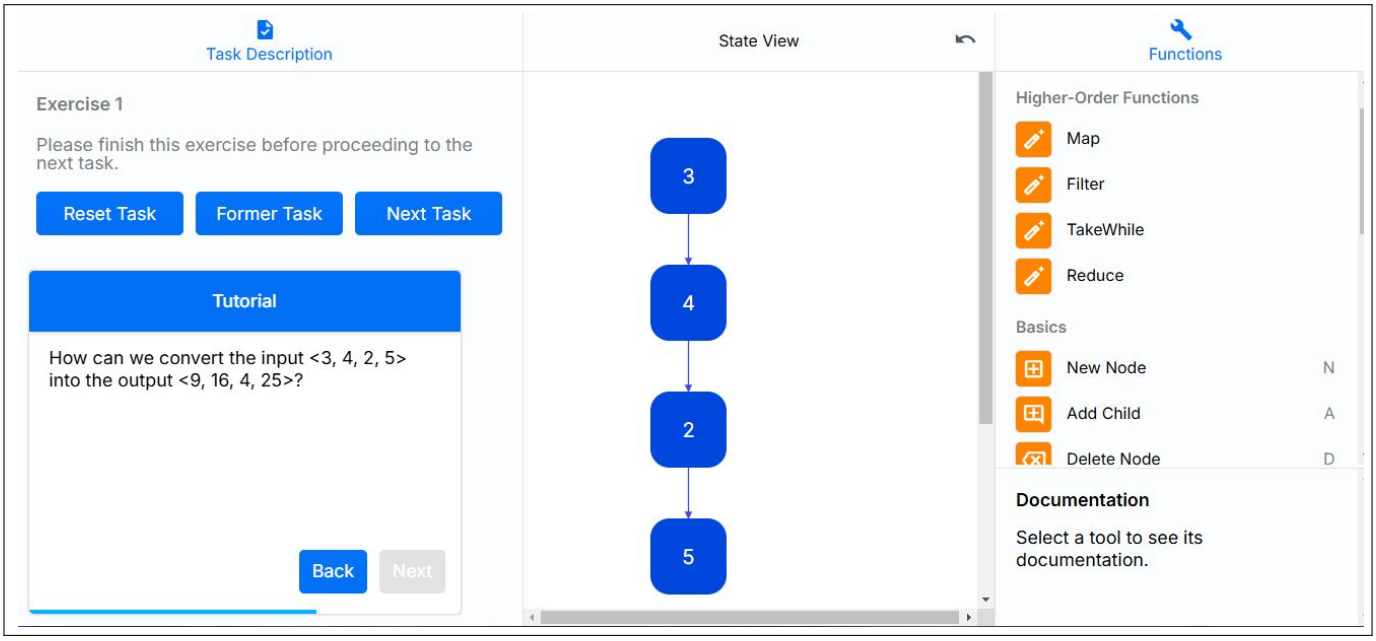


Figure 1: A screenshot of our system, which is an extension to the visual programming language Algot. The task shown here is from the tutorial used in our study.

To evaluate whether the system is useful for helping students learn about HOFs, we conducted a within-subject study on tertiary-level students with Python as a control environment. The participants followed interactive tutorials on HOFs in our system and in Python and solved tasks in each respective language. They also solved tasks based on behavioral properties of HOFs as introduced in a 2021 paper by Krishnamurthi and Fisler [11]. Our aim with the study was to answer the following research questions:

RQ1 Do novices experience lower cognitive load after receiving instruction and solving tasks on HOFs in our system than in Python?

RQ2 Do novices perform better at tasks involving HOFs in our system than in Python?

RQ3 Do novices perform better at conceptual tasks involving HOFs after working in our system than after working in Python?

Our hypothesis was that the research questions would be resolved affirmatively, i.e., that students would perform better on all counts using our system and also experience lower cognitive load.

We consider the contribution of this paper to be twofold. First, the system we introduce provides a novel approach to teaching HOFs in a purely visual environment, thereby eliminating many syntactic and conceptual barriers that can hinder novices. A second contribution is our study methodology, which we believe is a thorough yet simple way to help future investigations into the teaching of functional programming concepts.

2 Background

A central tenet of functional programming is the treatment of functions as first-class citizens, meaning functions can be passed as arguments to other functions or returned as results. The behavior of a function as a value is governed by its type signature, which specifies the types of its inputs and the type of its output. A function that either accepts another function as an argument or returns a function as its result is referred to as a *higher-order function* (HOF).

HOFs play a pivotal role in numerous applications. In particular, they are frequently used for data processing [3] and machine learning [5], and they enable developers to write more modular, reusable, and composable code [9]. As these benefits have become increasingly important, many traditionally imperative programming languages—such as Python (with `lambda` functions) and Java (with its stream API)—now integrate HOFs as well.

There are not many educational studies on how HOFs should be taught or how students learn to use them. However, a new line of research on the subject began with a 2021 study by Krishnamurthi and Fisler [11], which introduced a novel method for assessing students' understanding of higher-order functions through input-output behavior. Specifically, their approach required students to cluster and classify input-output pairs based on whether they could have been produced by the same underlying HOF. A follow-up experience report by Rivera and Krishnamurthi [14] also used input-output pairs as an instrument to analyze what makes some HOF problems more difficult than others, and another paper [15] looked into, among else, how well students understand individual HOFs based on their behavior.

More generally, some studies find that introducing students to programming through functional programming can offer notable advantages, including the potential for better-structured code and fewer errors [10]. Additionally, functional programming languages often resemble the algebraic or mathematical notations students

already know, thereby reducing the learning curve [4]. However, starting with functional programming also has drawbacks. Beyond HOFs, students frequently struggle with concepts common in functional programming such as recursion [16, 18] and the representation of types and data structures [1] in functional programs.

3 System

Our system (see Figure 1) builds on top of the live, visual programming language Algot, which implements direct manipulation, programming by demonstration, and liveness [19, 23]. In Algot, the program state is represented as a directed graph in which nodes hold values and edges act as references. The programmer can directly modify this graph by applying either built-in, atomic operations or user-defined operations on individual nodes. For example, applying the Sum operation to two nodes and storing the result in a third can be done with a few clicks, while Add Child attaches a new child node to an existing node. Lists are implemented as graphs where each node has no more than one child.

Programs in Algot are composed using a visual semantic approach based on programming by demonstration. Rather than typing code in a conventional text-based language, users execute operations by selecting the operation and then clicking on the relevant input nodes. Algot immediately reflects the results in the program state that is kept visible at all times [17], maintaining a live, interactive environment. More complex, repeated behavior is achieved by defining custom operations through a sequence of these atomic steps, and recursion is used instead of loops, which are common in imperative languages. Conditional execution is similarly facilitated through *queries*, which ask natural-language questions about node values (e.g., *Is Zero?*) and allow the user to branch based on the query's result.

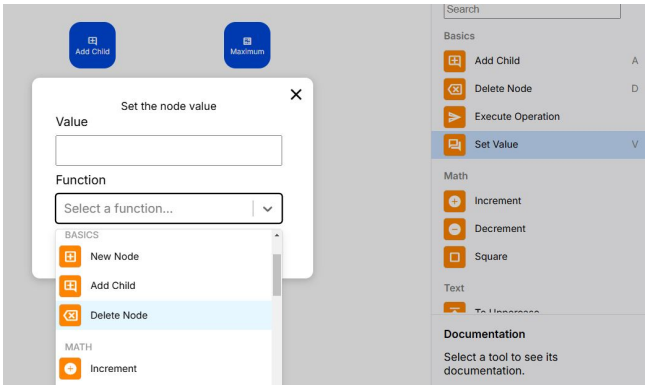


Figure 2: A screenshot from the system showing two nodes with operation values. The user is currently setting the value of a node using the value dialog with the operation selection.

Our extension to Algot allows a node to hold an operation or query as value. Internally, this is handled by saving the operation or query's unique identifier, while the interface displays the associated icon and name. To support this functionality, we modified the existing Set Value and Set Example Value operations, allowing users to select any available operation or query for assignment to a node (see Figure 2). To allow a user to execute an operation that is stored in a node, we added the Execute Operation base operation to our new version of the system. This operation takes a node containing an operation as the first argument and then executes the stored operation on the remaining inputs. A design challenge arises from the fact that different operations may require different numbers of arguments. To address this, Execute Operation dynamically adapts its parameter list to match the arity of the stored operation. Users must therefore provide an example where the maximum number of parameters for a given operation is specified; if an operation receives fewer arguments than required, it is not executed, and if it receives more, it simply ignores the extra inputs, consuming only those needed.

4 Method

After receiving ethics approval from our institution, we conducted a controlled, experimental study on tertiary-level students in a computer laboratory setting, with screen recordings used to keep track of the their performance. The students received 25 CHF per hour in compensation for their participation. We adopted a within-subjects design in which each participant used our system (the experimental condition) and Python in a Jupyter notebook environment (the control condition) to learn about HOFs. After each session, the students' learning was measured. To limit learning effects, we used counterbalancing and a short distractor task between the two sessions.

Our participants were recruited from a large pool (>10,000) of students enrolled in universities in Zürich, Switzerland. The inclusion criteria were to have completed at least one computer programming course at university level, to have at least a basic level of experience of Python, and to have no prior knowledge of HOFs. To determine the number of participants needed, we performed power analysis using the `TTestPower` function in the `statsmodels` module in Python, assuming a moderate effect size (0.5), a standard significance level (alpha = 0.05), one-sided hypothesis testing, and a desired power of 0.80. We recruited **27** participants after finding that this was the required number.

```
1: ("cs019", "ma054", "cs033", "cs018", "visa039") -> ("cs019")
2: (1, 2, 3, 4) -> (1, 2, 3, 4)
3: ("cs019", "ma054", "cs033", "cs018", "visa039") -> ("ma054",
     "visa039")
4: ("a", "b", "d", "e") -> ("a", "e")
5: (4, 6, 2, 1) -> <>
6: ("red", "green", "blue") -> (3, 5, 4)
7: (true, true, false, true, false, true, false) -> true
8: (1, 4, 4, 2, 6, 1) -> 3
9: (4, 6, 2, 1) -> (1, 1, 1, 1)
10: (true, true, false, true, false, true, false) -> (true, true)
11: (4, 6, 2, 5) -> (1, 1, 1)
12: (1, 4, 4, 2, 6, 1) -> (1, 4, 2, 6, 1)
13: ("cs019", "ma054", "cs033", "cs018", "visa039") -> 2
14: (1, 2, 3, 4, 5) -> (1, 4, 9, 16, 25)
15: (1, 7, 2, 3, -1, 4, 2, 6, 8, 7, 9, -5) -> (1, 7, 2, 3)
16: (1, 2, 3, 2, 1) -> 5
```

Figure 3: The input-output pairs provided in both the pretest and the behavioral questions of the study. The list is a modified version from Krishnamurthi and Fisler [11].

The total study duration was two hours. Each participant began by solving a pretest which was a modified version of the one used in the earlier study by Krishnamurthi and Fisler [11], in which input-output pairs were offered and the participants were asked to assign them to five clusters based on "similarity". The participants were given the same instructions as documented in the Krishnamurthi and Fisler paper, but we reduced the number of input-output pairs and simplified some of them. The total list can be seen in Figure 3. The pretest helped us discover if, due to random chance, there was any significant skill difference between the groups of students that began with either intervention, which could possibly undermine the results.

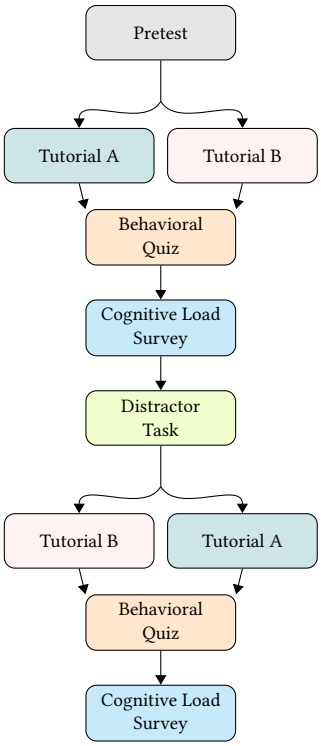


Figure 4: The within-subjects crossover design that we used in the study. Each participant was given each tutorial once.

Following the pretest, half of the participants were given an up-to-45-minute tutorial on HOFs, one using our system and the other half in Jupyter. Each tutorial instructed the students on how to use four HOFs: `map`, `filter`, `take-while` and `reduce` (also known as `fold`). These four functions were the same ones that were used by Krishnamurthi and Fisler, with the exception of `ormap`, which we chose to omit due to the time constraints of the study and because it is less common than the others. Both tutorials were interactive and required students to test what they had learned. For example, the part of the Python tutorial on `take-while` function began by defining how the function works, and then demonstrated how it would transform the list `<ant, ape, dog, rabbit, cow, goat>` when given a boolean function that returns true only when the input begins with the letter 'a'. As this was presented in a Jupyter notebook, students could run the code by themselves. They were then asked to modify the code (change the condition to only keep strings with three characters) and observe how the output changed. The instruction on the other three functions was similar. The tutorial in our system was designed to be as similar as possible to the one in Jupyter, in which the same four HOFs were presented using the same text and students asked to implement them using concrete examples.

After each tutorial ended, the participants were asked to solve questions on the behavioral features of HOFs, similarly to the Krishnamurthi and Fisler study. They were then asked to self-assess their cognitive load during the tutorial using the Paas scale [13], which is a short instrument that is often used for that purpose. After this, they were given a short distractor task (simple arithmetic questions), and then proceeded with the other tutorial and the same behavioral questions as before, followed by submitting their response on the Paas scale again. Figure 4 shows the overall study design.

At the end of each tutorial, the participants were asked to use the HOFs that they learned about to map a given input to a given output. We measured the accuracy of their solutions on these questions and the time that they needed to solve them. The four input-output pairs that we asked about were: **(1)** [3, 4, 2, 5] → [9, 16, 4, 25], **(2)** ['h', 'o', 'f', '!'] → 'hof!', **(3)** [3, 4, -1, 6] → [3, 4], and **(4)** ['bob', 'alice', 'daniel', 'andy'] → ['alice', 'andy'].

All calculations were conducted in the statistical software JASP (v. 0.18.1). For more information on our statistical analysis methods, we refer to the JASP user guide [12].

5 Results

5.1 Participants and pretest

27 students participated in the study, of which there were 11 women and 16 men. The median age of the participants was 23 (standard deviation 4.2 years). 14 participants were enrolled in engineering and technology-related programs. One participant claimed to have used the programming language Algot before, while the other 26 did not. All 27 participants completed the study.

To grade the performance on the prestest, we used the same procedure as Krishnamurthi and Fisler [11], calculating the sum of the Jaccard similarities of the elements in the student-created sets and the ground truth sets. The students who started with the experimental condition had similar pretest scores (avg. 1.34) as those that began with the control condition (avg. 1.47), and there was no significant difference ($p = 0.50$), suggesting that the order randomization was likely balanced.

5.2 Cognitive load

After first conducting a Shapiro-Wilk normality test and not finding evidence for deviations of normality in the data ($p = 0.16$), we conducted Student's paired samples t-test to compare the difference in perceived cognitive load for the two conditions (see Table 1). We found strong evidence that students experienced lower perceived cognitive load in the experimental condition ($p < 0.001$). Cohen's d was 1.03, indicating a strong effect size according to Cohen's benchmarks [2]. The average result for the control environment (6.04) was closest to the label "rather high mental effort," while

the average result for the experimental environment (4.78) was in between "rather low mental effort" and "neither low nor high mental effort'.' A frequency distribution can be found in Figure 5.

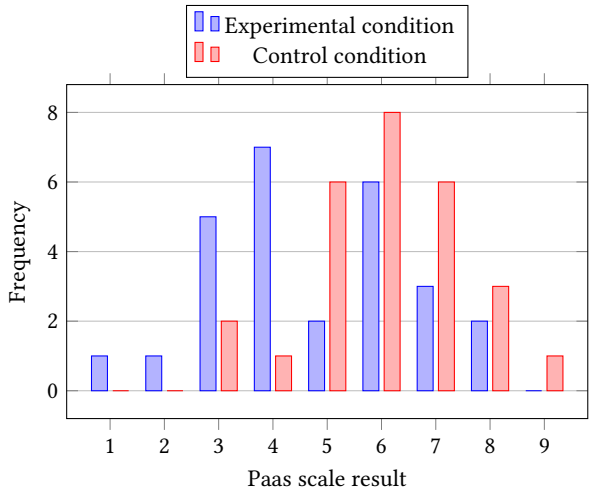


Figure 5: Frequency distribution of cognitive load scores for the experimental and control conditions. The scale covers the range of 1 to 9, with 1 indicating "very, very low mental effort" and a score of 9 suggesting "very, very high mental effort."

Descriptive statistics		
	Control CL	**Experimental CL**
Mean	6.04	4.78
SD	1.45	1.83
SE	0.28	0.35
Coeff. of variance	0.24	0.38
Paired Student's T-Test		
p-value	< .001	
Cohen's d	1.03	
SE (Cohen's d)	0.17	

Table 1: The cognitive load (CL) scores according to the Paas scale for the Python environment (control) and for our direct manipulation system (experimental). We report the p-value for our alternative hypothesis that students would experience lower CL under the experimental condition.

5.3 Behavioral tasks

Next, we looked into the performance on the behavioral concept questions (see Figure 3). We tested two separate grading scales; first, a binary scale where either one or zero points were awarded for correctly identifying exactly which of the four HOFs that could map the given input to the given output, and second, a partial-credit method that captures more information, under which each student's score was the fraction of correct HOFs they selected minus the fraction of incorrect HOFs they chose, with a floor of zero. Under both scales, the difference between the two groups was minimal (score pairs of 0.43 and 0.41, and 0.53 and 0.51, respectively, with $p > 0.05$ in both cases under our hypothesis testing).

5.4 Program implementations

For analyzing the performance on the programming tasks provided in each environment, we again computed a Student's t-test. For each participant, we counted the number of errors they made in each task. For grading the Python performance in the control condition, we used two grading mechanisms; one in which we ignored syntax errors, and one where we did not. To account for the fact that students could use trial-and-error to discover the correct solution to the tasks, we used a strict grading scheme that only graded the first attempt of solving each task. This meant that we only considered the first time a cell was run in the Python environment and the first time a HOF was executed in the Algot environment.

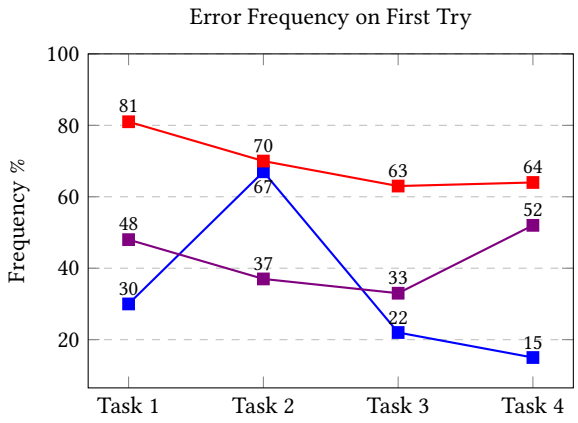


Figure 6: The percentage of errors that were made in the four programming tasks on the first attempt. The blue line (30 on task 1) corresponds to the experimental condition, the red line (81 on Task 1) corresponds to the control condition, and the purple line (48 on Task 1) corresponds to the control condition if syntax errors are ignored.

Figure 6 shows the error frequency for the four tasks in each environment. The error rate was lowest in the experimental condition. We use a paired-samples Student's t test to analyze the significance of the difference. The experimental condition had the lowest error frequency (M = 1.33, SD = 1.11), followed by the control condition without syntax errors (M = 1.70, SD = 1.41) and then the control condition with syntax errors (M = 2.89, SD = 0.89). The Student's t test revealed a significant difference between the conditions $t(26) = -7.211$, $p < 0.01$ and a large effect size ($d = -1.388$), whereas the difference between Algot and Python with no syntax errors was non-significant $t(26) = -1.629$, $p = 0.058$. These results suggest that the inability of users to make syntax errors contributed to the stronger performance in our system.

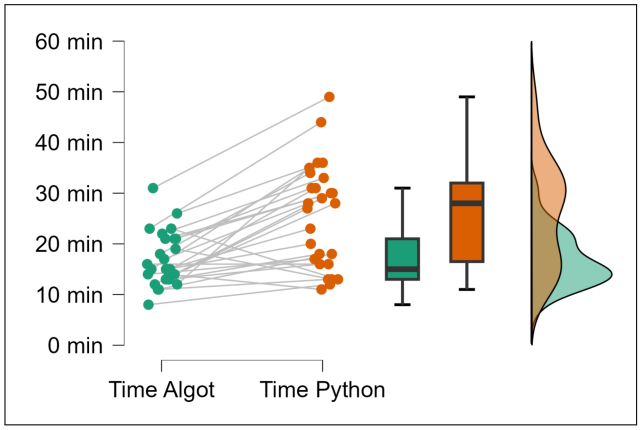


Figure 7: A raincloud plot showing the time used by each student on the parts of the experiment.

5.5 Time use

We found a significant difference between the time usage between the Algot and Python environments; on average, the participants spent 16.8 minutes (SD = 5.3) working through the Algot (experimental) environment, while they spent 25.6 minutes (SD = 10.3) on in the Python (control) environment. A Student's t-test indicated the difference was significant ($p < 0.001$) and the effect size was large (Cohen's $d = 1.07$). This is consistent with the cognitive load results, which suggested that the control condition was more challenging. Figure 7 shows a raincloud plot of the difference in time use between the two groups; the plot and our inferential statistics indicate that the difference was not driven by outliers.

5.6 Textual feedback

At the end of the study, participants were invited to offer optional feedback on their experience. Due to the low number of responses, we did not conduct qualitative analysis, but we found that the responses might help us put the results into context. One student wrote that "[A]lgot was easier, as syntax played no role and I could focus more on the meaning than in [J]upyter," which is an observation that is well-aligned with the results. Another student wrote that "[v]isualiz[ing] Algot makes my understanding easier" and another that "Algot was easier to comprehend than writing codes myself," and another that "It was quite pleasant and mostly well explained, even when I take intro account that I had a Python [i]ntroductory [c]ourse but did not use Python for a while." One student offered a contrasting perspective, writing that it was "[v]ery [repetitive], [I] can't imagine it being more useful than just straight up code, [because] of complexity reasons and also overview. The environment looked great though and it was easy and needed no coding experience."

6 Discussion

In response to RQ1, we found that students' self-reported cognitive load was significantly lower when using our system than when using Python, For RQ2, we also found that novices perform significantly better at HOF tasks in our system than in Python, even though the participants in our study had prior experience with Python by design. However, for RQ3, we did not find evidence that the students were better at the conceptual tasks (the behavior-based questions) after exposure to our system than after exposure to Python, which may possibly be due to the brevity of the intervention. This indicates that our hypothesis was confirmed for the first two research questions and not the third one.

Overall, we find that these results suggest at least that direct manipulation systems for teaching advanced computer programming concepts have some partial utility. By allowing learners to visualize and interact with higher-order functions directly without syntactic overhead, such systems can promote experimentation and reduce cognitive barriers. Direct manipulation tools also offer immediate feedback on how operations transform data, which may accelerate students' comprehension and help them identify misconceptions early on. As a result, students may be more willing to explore complex notions, including function composition and parameter passing, when their programming environment clearly illustrates each step of the process.

As for the threats to the validity of our study, we note that the conclusions are limited by the task and HOF selection; although we took care to select representative tasks, a different choice might have resulted in a different outcome. We also note that although the power analysis indicates that we recruited a sufficient number of participants, our study might be underpowered if the true effect size is lower than 0.5, which could have contributed to the non-significant results for our third research question. We also note that although we took steps to limit learning effects, they might still have moderated the difference in results of the two conditions. A larger, between-subjects version of our study design would offer more conclusive evidence. Last, we note that our grading criteria had an impact on the outcome; by only scoring participants' first attempt on each task, we eliminated the possibility that novices would eventually guess their way to a solution through repeated trial-and-error. However, this choice also omitted any evidence of iterative refinement or incremental learning that might occur after an initial misconception. Future studies could address this trade-off by tracking partial credit for progressive improvements, by not giving students the option to test their solution before they submit it, or by more complex tasks that reduce the likelihood of arriving at the correct solution through guesswork alone.

7 Conclusion

In this work, we introduced a live, direct manipulation system designed to help novices learn higher-order functions by interacting with visual representations instead of writing textual code. Through a controlled study of 27 students, we found that participants reported significantly lower cognitive load in our system compared to Python and made significantly fewer errors in basic HOF programming tasks. A comparison of conceptual understanding after each intervention was inconclusive. Overall, our findings suggest that visual, direct-manipulation environments have some promise for teaching higher-order functions. Future efforts can build on our approach by expanding the range of topics, employing longer-term interventions, and exploring how visual methods integrate with more advanced functional paradigms.

References

[1] Christopher Chambers, Sheng Chen, Duc Le, and Christopher Scaffidi. 2012. The function, and dysfunction, of information sources in learning functional programming. *J. Comput. Sci. Coll.* 28, 1 (Oct. 2012), 220–226.

[2] Jacob Cohen. 1988. *Statistical Power Analysis for the Behavioral Sciences* (2nd ed.). Routledge. https://doi.org/10.4324/9780203771587

[3] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. *Commun. ACM* 51, 1 (Jan. 2008), 107–113. https://doi.org/10.1145/1327452.1327492

[4] Robert Bruce Findler, John Clements, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Paul Steckler, and Matthias Felleisen. 2002. DrScheme: a programming environment for Scheme. *J. Funct. Program.* 12, 2 (March 2002), 159–182. https://doi.org/10.1017/S0956796801004208

[5] Deqing Fu, Tian-Qi Chen, Robin Jia, and Vatsal Sharan. 2024. Transformers learn to achieve second-order convergence rates for in-context linear regression. *Advances in Neural Information Processing Systems* 37 (2024), 98675–98716.

[6] Maximilian Georg Barth, Sverrir Thorgeirsson, and Zhendong Su. 2024. A Direct Manipulation Programming Environment for Teaching Introductory and Advanced Software Testing. In *Proceedings of the 24th Koli Calling International Conference on Computing Education Research (Koli Calling '24)*. Association for Computing Machinery, New York, NY, USA, Article 2, 11 pages. https://doi.org/10.1145/3699538.3699564

[7] Oliver Graf, Sverrir Thorgeirsson, and Zhendong Su. 2024. Assessing Live Programming for Program Comprehension. In *Proceedings of the 2024 on Innovation and Technology in Computer Science Education V. 1* (Milan, Italy) *(ITiCSE 2024)*. Association for Computing Machinery, New York, NY, USA, 520–526. https://doi.org/10.1145/3649217.3653547

[8] Zhenjiang Hu, John Hughes, and Meng Wang. 2015. How functional programming mattered. *National Science Review* 2, 3 (07 2015), 349–370. https://doi.org/10.1093/nsr/nwv042 arXiv:https://academic.oup.com/nsr/article-pdf/2/3/349/31566307/nwv042.pdf

[9] J. Hughes. 1989. Why Functional Programming Matters. *Comput. J.* 32, 2 (1989), 98–107. https://doi.org/10.1093/comjnl/32.2.98

[10] Stef Joosten, Klaas Berg, and Gerrit Hoeven. 1993. Teaching Functional Programming to First-Year Students. *Journal of Functional Programming* 3 (01 1993), 49–65. https://doi.org/10.1017/S0956796800000599

[11] Shriram Krishnamurthi and Kathi Fisler. 2021. Developing Behavioral Concepts of Higher-Order Functions. In *Proceedings of the 17th ACM Conference on International Computing Education Research* (Virtual Event, USA) *(ICER 2021)*. Association for Computing Machinery, New York, NY, USA, 306–318. https://doi.org/10.1145/3446871.3469739

[12] 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.

[13] Fred GWC Paas. 1992. Training strategies for attaining transfer of problem-solving skill in statistics: a cognitive-load approach. *Journal of educational psychology* 84, 4 (1992), 429.

[14] Elijah Rivera and Shriram Krishnamurthi. 2022. Structural versus pipeline composition of higher-order functions (experience report). *Proceedings of the ACM on Programming Languages* 6, ICFP (2022), 343–356.

[15] Elijah Rivera, Shriram Krishnamurthi, and Robert Goldstone. 2022. Plan Composition Using Higher-Order Functions. In *Proceedings of the 2022 ACM Conference on International Computing Education Research-Volume 1*. 84–104.

[16] Judith Segal. 1994. Empirical studies of functional programming learners evaluating recursive functions. *Instructional Science* 22, 5 (1994), 385–411. http://www.jstor.org/stable/23369999

[17] Sverrir Thorgeirsson, Oliver Graf, and Zhendong Su. 2024. The Hidden Program State Hurts Everyone. In *Proceedings of the 2024 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (Pasadena, CA, USA) *(Onward! '24)*. Association for Computing Machinery, New York, NY, USA, 266–274. https://doi.org/10.1145/3689492.3689813

[18] Sverrir Thorgeirsson, Lennart C. Lais, Theo B. 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 V. 1 (SIGCSE 2024)*. Association for Computing Machinery, 1321–1327. https://doi.org/10.1145/3626252.3630916

[19] Sverrir Thorgeirsson and Zhendong Su. 2021. Algot: An Educational Programming Language with Human-Intuitive Visual Syntax. *2021 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)* (2021), 1–5. https://api.semanticscholar.org/CorpusID:240156985

[20] Sverrir Thorgeirsson, Theo B 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 V. 1*. 1328–1334. https://doi.org/10.1145/3626252.3630808

[21] Sverrir Thorgeirsson, Chengyu Zhang, Theo B. Weidmann, Karl-Heinz Weidmann, and Zhendong Su. 2024. An Electroencephalography Study on Cognitive Load in Visual and Textual Programming. In *Proceedings of the 2024 ACM Conference on International Computing Education Research (ICER '24)*. ACM, Melbourne, VIC, Australia. https://doi.org/10.1145/3626252.3630808

[22] Isomöttönen V. Tirronen V, Uusi-Mäkelä S. 2015. Understanding beginners' mistakes with Haskell. *Journal of Functional Programming* 25 (2015). https://doi.org/10.1017/S0956796815000179

[23] 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. https://doi.org/10.1145/3563835.3567668