Recursion in Secondary Computer Science Education: A Comparative Study of Visual Programming Approaches

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

Lennart C. Lais*
lelais@student.ethz.ch
ETH Zürich
Zürich, Switzerland

Theo B. Weidmann
tweidmann@ethz.ch
ETH Zürich
Zürich, Switzerland

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

Abstract

While recursion is a fundamental technique in computer programming, it is challenging for novices, for example since it requires tracing non-linear and hierarchical sequences of execution. Though algorithm visualizations and visual programming may be helpful, such tools need to offer sufficiently expressive environments that support active, constructivist learning via exploration and experimentation. In this study, we investigated whether Algot, a visual programming language that relies on a novel programming-by-demonstration paradigm, is effective for teaching recursion to 14-17 year old students, and whether it compares favorably to the popular visual programming language Scratch. We conducted an experimental study with 23 participants where they learned recursion in a video tutorial using Algot and Scratch, worked out code exercises in each respective language, and then solved a post-test on recursion. Despite the participants being more familiar with Scratch than Algot, our results indicated that students instructed with Algot demonstrated a significantly better understanding of recursion (Bayes Factor = 14.09, p = 0.005, Cohen's d = 1.30). We also found that students reported a similar level of enjoyment of each language. These findings provide preliminary evidence about the effectiveness of the programming-by-demonstration paradigm, as implemented in Algot, in aiding the comprehension of complex programming concepts like recursion.

CCS Concepts: • **Human-centered computing** → **Empirical studies in visualization**; • **Social and professional topics** → **K-12 education**.

Keywords: visual programming, block-based programming, recursion, programming by demonstration, educational technology, secondary education

1 Introduction

Recursion is a fundamental technique in computer programming in which tasks are broken down into smaller subtasks that are a self-similar or simpler version of the original problem. In computer science (CS) education, it is also a representative computational thinking strategy that exemplifies, for example, Piaget's notion of reflective abstraction [4], but also decomposition, pattern recognition, and algorithmic problem-solving. However, it is considered a difficult topic for novices to grasp [1, 19, 38]. This may be partially due the non-linear control flow which makes it hard to trace the flow of execution, but also since recursive algorithms introduce complexities when it comes to function calls, parameter passing, and scope of variables.

In this paper, we investigated whether Algot [35, 39], a visual programming language specifically designed for CS education, is effective as a tool for teaching recursion at the secondary education level. Unlike most other programming languages, Algot is based on a programming-by-demonstration computational model where users can define, run, and interact with their programs using direct manipulation while seeing an explicit visual representation of control flow and data structures. Our hypothesis is that Algot provides an effective way to teach CS concepts that are normally considered particularly difficult in K12 education. To evaluate how well Algot can teach recursion at the secondary school level, and to understand how Algot compares against other visual programming languages, we designed a controlled study where Algot was compared against Scratch [27], a block-based programming language that is popular in CS education and is familiar to most students in the local school district. We asked participants aged 14 to 17 to first complete a pre-test, then watch a tutorial for each programming language, solve recursion-based programming tasks in each language, fill out a post-test based on a recursion concept inventory [11], and lastly to complete a survey about their experience.

*Sverrir Thorgeirsson and Lennart Lais are co-primary authors.

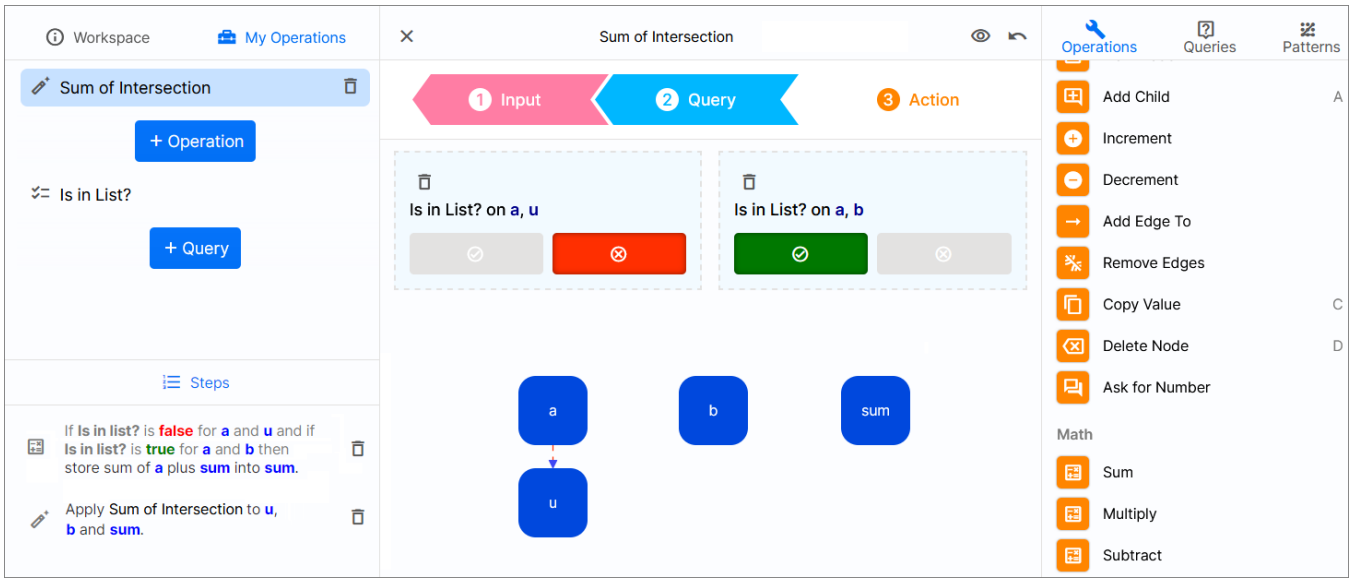

Figure 1. A screenshot of Algot (lightly edited for space reasons) showing the user-defined operation *Sum of Intersection*. The bottom left shows the semantics of the operation.

We had two primary research questions that we sought to answer with our study:

RQ1 Can Algot be used effectively to teach recursion to 14-17 year old students?

RQ2 For the same age group, how does Algot's effectiveness in teaching recursion compare to an established educational programming language like Scratch?

We also collected ancillary data to help answer two secondary research questions, namely (i) whether the combined use of Algot and Scratch affects students' understanding and application of recursion in programming, and (ii) whether students find a difference in enjoyment level after programming in each environment.

2 Related Work

While recursion has been described by Papert as "particularly able to evoke an excited response" among children [24], it is also frequently considered a "notoriously difficult" topic to learn or master [1, 19, 38]. Attempts to identify recursion-related misconceptions and interventions span many decades. One explanation is that students struggle to cultivate a mental model of the program stack, without which it is hard to understand the backward flow of control after reaching a recursive function's base case [7, 30]. Additionally, a lack of real-world examples can compound the conceptual difficulties [1, 25].

In early secondary education, some authors have proposed and experimented with teaching recursion with "unplugged" activities [9, 20, 33], which means group problem-solving activities without the direct use of technology, by using specific programming languages such as Logo [9] or Python [18], or by using a combination of tools [33]. Many authors have advocating for a conceptual approach [2, 5, 8, 26, 31], stressing the importance of examples [25, 30, 32] and the use of visualization [7, 9, 14]. Two controlled experiments on high-school students have found that specific approaches are more effective than others; one that found logic programming helped students gain a better mental model of recursion than procedural programming [10], and another study found that the computer game Cargo-Bot, which allows the user to write sequenced, nested instructions to control a robot, was more helpful in improving the students' understanding of recursion than a direct instruction approach with Java [34].

3 Background

3.1 Algot

Algot is a visual programming language specifically designed for computer science education [35, 39]. Unlike block-based programming languages where the program logic is represented by interlocked blocks of code, programming in Algot does not require comprehending or manipulating code as a semantic structure. Instead, Algot uses programming-by-demonstration as its computational model where programs are created by selecting from pre-defined and user-created queries and functions which are applied on graphs or specific types of graphs such as trees, linked lists, and individual nodes. The motivation behind Algot is to remove a barrier to learning by making the meaning of programs more transparent; defining a program in Algot is similar to how the program is executed, echoing the concepts put forward in Bret Victor's 2012 essay on learnable programming [37]. A comparative study found that undergraduates experienced

lower cognitive load and performed better when solving simple programming tasks in Algot than in the textual language Python [36].

To explain how Algot can be used, we provide an illustrated example of using Algot to compute the sum of the intersection of two linked lists. Figure 1 shows how a new operation is defined in Algot using the *Demonstration View* of Algot, which consists of three stages called the *Input Stage*, *Query Stage* and *Action Stage*:

1. Input Stage: The input arguments are specified. Nodes that belong to the same connected component as the input nodes can also be identified via pattern matching. In this case, three input nodes are specified (the head of two lists, `a` and `b`, and the node `sum` storing the sum) but also `a`'s child node `u`.
2. Query Stage: The programmer specifies binary questions about the nodes by applying queries (e.g., whether two nodes have the same value). Five base queries are provided by default and the programmer can create their own, for example *Is in List?*, as shown here, which checks whether the value of its first argument is contained in the list headed by its second argument.
3. Action Stage: Custom or pre-defined operations are called on the input arguments or newly created nodes. If any queries are active, the operations will be called conditionally. In this example, if two conditions are correct, namely that the value of `a` is not in the list headed by its child `u`, and that `a` is in the list headed by `b`, then the node *sum* is increased by the value of *a*. Then the operation *Sum of Intersection* is called recursively on `u`, `b` and `sum`.

We have two clarifying remarks. First, note that when an operation is called on non-existent input nodes, this is not considered an error but the operation will simply not be executed. This means that in the next recursive call of *Sum of Intersection*, if the child node *u* does not exist, the operation will terminate at this point. In general, this behaviour can eliminate the need for explicitly defining a base case or stating when the algorithm should terminate. Second, when operations changes the structure of the state, such as when new nodes or edges are created or deleted, this is shown visually in the demonstration view.

Figure 2 shows the results of applying the operation *Sum of Intersection* in the Algot *state view*, an interactive part of Algot which resembles the action stage of the demonstration view. Here, the operation was called on input arguments representing two linked lists and a singleton node that initially had the value zero.

This example is sufficiently complex to show the main capabilities of Algot but also simple enough to be described concisely. For more details on the computational model behind Algot, we refer to a recent paper by its creators [39].

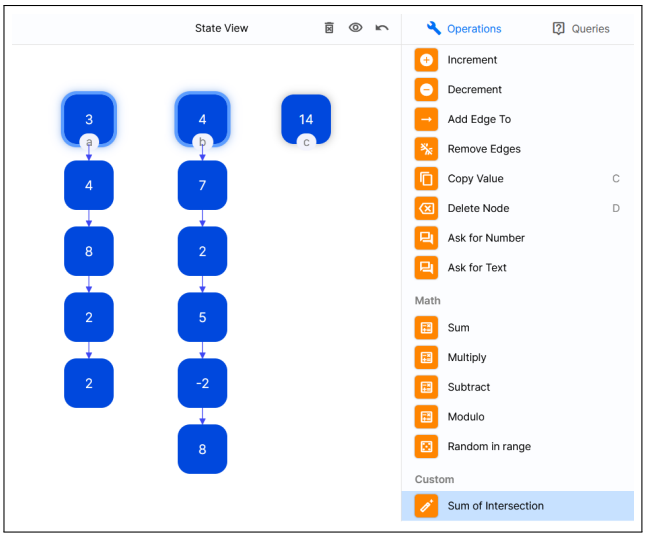

Figure 2. A screenshot showing the user-defined function *Sum of Intersection* applied on the two input linked lists marked `a` and `b` in the Algot state view. The intersection of the two lists is $\{8, 4, 2\}$ and thus the sum 14 is stored in the third function argument `c`.

3.2 Scratch

First released in 2007, Scratch is a block-based programming language specifically designed to introduce computer programming to young people [27]. Its visual interface allows learners to create programs by putting together predefined blocks of code. This approach helps beginners in several ways, for example by preventing syntax errors, but also by reducing the need for prior knowledge by visually presenting the language's capabilities [40]. More than just an educational programming language, Scratch is now a very large online community whose usage doubled over the COVID-19 pandemic [3] and now boosts over a million active monthly users as of the 2023 school year [6].

Soon after Scratch was released, Harvey and Möning [13] noted that Scratch lacked support for "the impressive phenomenon of recursion, one of the central ideas of computer science" which, among other reasons, inhibited the use of Scratch in the introductory CS curriculum. To address this, the authors introduced a Scratch extension named *Build Your Own Block* (BYOB), later reimplemented as its own spin-off language *Snap!*, which is explicitly designed to teach recursion and higher-order functions [12]. Perhaps inspired by Snap!, Scratch 2.0 (released in 2013) contains support for custom procedures via the *Make a Block* functionality [16, 23], which enables recursive programming, but unlike Snap!, does not support higher-order functions by allowing custom blocks to generate their own blocks. Like user-defined operations in Algot, these custom blocks do not possess a return statement, but can modify global variables via side effects.

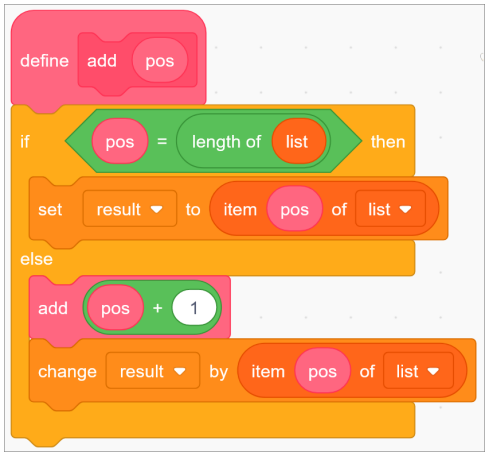

Figure 3. A screenshot depicting a function in Scratch whose control-flow is determined by several different blocks of code. This example shows a recursive function which computes the sum of a list. In our study, students were given the function block and asked to complete it.

Figure 3 shows an example of a recursive function in Scratch 3.0 using this feature.

While the *Make a Block* feature in Scratch is not as expressive as custom blocks in Snap!, its simplicity may make it easier to understand and use in certain contexts. Some Scratch textbooks [15, 22] and online educational materials [21, 29] geared towards young students use this feature to teach recursion, for example by featuring examples with repeating geographical patterns. However, to our best knowledge, the efficiency or usage of *Make a Block* in Scratch for teaching recursion has not yet been subject to empirical research.

4 Method

Upon securing ethics approval from our institution, we recruited 23 individuals ranging from 14 to 17 years old who provided their informed consent to take part in our study. These individuals were compensated with gift cards worth 35 Swiss Francs. All participants were recruited from a scout group in Zürich, Switzerland. This study required some familiarity with computer programming, a prerequisite commonly met in this age group due to the inclusion of computer science in the local secondary education curriculum.

When arriving at the study location, participants were randomly divided into an experimental group and a control group. Both groups (i) filled out a pre-study survey on their perceived programming skills and programming language knowledge, (ii) underwent instruction on recursion and Algot and Scratch, respectively, via a video tutorial, (iii) proceeded to solve exercises in each respective language (Figures 3 and 5) and then (iv) completed a post-test (Figure 4). Lastly, participants filled out a survey on their study

a) Heating Function

The heating function controls how much a heater heats. At what level is the heating after the heating function has been executed with input 4? Select a number or "Infinite recursion" if you think the function never stops.

```
Heater with input X:
    When X equals 5:
        Set the heater to level 3
    If X is not equal to 5:
        Run Heater with input X+1
        Reduce X by 1
        Set the heater to level 1
```

The alternatives given were 1, 2, 3 and *Infinite recursion*.

b) Countdown Function

When the Countdown function is executed, the screen should count down from the second input to the first input. For example, if Countdown is called with inputs 2 and 5, the screen should show 5, 4, 3, 2. Complete the function by filling in the missing line.

```
Countdown with inputs X and Y:
    If X is smaller or equal to Y:
        Show Y on the screen
        [missing line]
```

Alternatives:

```
a) Run Countdown with inputs X-1 and Y
b) Run Countdown with inputs X and Y-1
c) Run Countdown with inputs X+1 and Y
d) Run Countdown with inputs X and Y+1
e) None of the above
```

c) Recursion Explanation

In general terms, explain the approach of solving a problem with a recursive program.

Figure 4. Three questions from the post-test. The last questions was graded with the assistance of a simple grading rubric. The multiple-choice questions shown are adapted from the first and third questions on the second iteration of the Basic Recursion Concept Inventory [11].

experience that asked on a 5-point scale whether they enjoyed programming in each respective language and if they had some suggestions for improvements. Participants could view the tutorial at any time during the study until the post-test was administered. The tasks were completed on the

participants' own laptop computers, but in person and under supervision. Participants were given an hour in total to complete the experiment.

When designing the video tutorials on recursion, we incorporated the aforementioned guidelines and suggestions on how recursion should be taught, for example by introducing students to a concrete, real-world example in the form of Matryoshka dolls, using a three-step approach proposed by Ginat for solving problems recursively [8], and by showing a visual step-by-step execution of a recursive function.

> **Task 1**: Complete the operation *GenerateTree* which takes as input a node with value *n* and generates a tree starting with the value *n* such that the value of a node should be equal to its height on the graph, every node should have two children except for the nodes with the value 0, and the nodes with the value 0 should have the color green.
>
> **Task 2**: In this task, all the values in a list should be added together. Complete the function *ListSum* for this. The input `head` is the start node of the list and its sum is to be stored in the input node `sum`. The operation should work regardless of the initial value of node `sum`.

Figure 5. The two tasks that students were asked to complete in Algot. Students received similar instructions in Scratch. Both groups were also shown illustrations to better understand the tasks (for instance Figure 6).

The post-test was loosely adapted from the basic recursion concept inventory by Hamouda et al. [11], a set of questions that aims to identify misconceptions about recursion. The existing version of the concept inventory was not suitable since it is explicitly designed for C-like languages and tests misconceptions about recursion that we believe are unlikely to occur in Algot or Scratch. For example, errors that arise from misconceptions about the return statement cannot arise in Scratch or Algot, as they lack a return statement. We, nonetheless, selected parts that seemed appropriate and adjusted them to our needs. Two different versions of the post-test were used to prevent potential knowledge leakage between sessions and to discourage self-discovery of answers outside the study's framework, but we intended the versions to be equally difficult. The first three items were multiple-choice questions with four or five alternatives, and the last question (included on both versions) asked for a free-form response on how recursion works. Examples are shown in Figure 4.

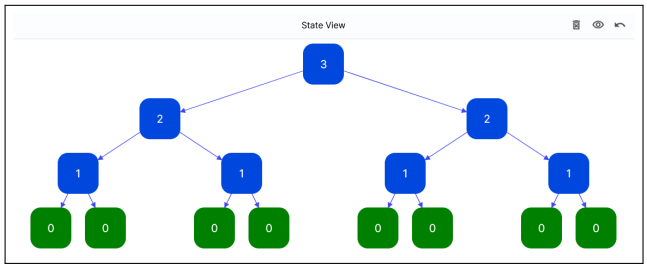

Figure 6. A screenshot showing an Algot representation of a perfect binary tree whose node values are equal to the node depth minus the tree height. In one task of the study, students were provided with a similar diagram and asked to create functions that can generate such trees.

Two weeks after the experimental session, participants were asked to return and complete the experiment again, but under the conditions of the opposite group and while taking the other version of the post-test. The purpose of this was to allow participants to experience both environments so that we could gather comparative feedback—after the end of the second session, participants completed a survey asking them to compare their experience with the two programming environments—and to help determine whether a combined use of Algot and Scratch is effective by estimating whether the average post-test score increased across sessions. The purpose of the second session was, however, not to gather additional research data to answer our first two research questions due to the strong possibility of transfer or carryover effects.

5 Results

23 participants were divided randomly into two groups. The results of one participant had to be excluded due to technical difficulties, so in the final sample, the control group and the test group had 11 participants each. The remaining participants (7 male and 15 female) were balanced in both age (avg. age 15.5) and self-reported programming background. The pre-test survey indicated that 16 of the 22 remaining participants were familiar with programming in Scratch. Of the remaining 6, 4 had some exposure to Python, 1 to C/C++, and 1 was unsure. 5 participants considered themselves to have a below-average programming background, 15 reported an average background, and 2 claimed to be above average. No student had any prior exposure to recursion.

Table 1 and the visualization of the results in Figure 7 shows that the test group (Algot) had higher post-test scores than the control group (Scratch) with a large effect size (Cohen's d is 1.30). We used Bayesian and classical statistical tests to evaluate the null hypothesis against the alternative hypothesis, namely that Algot-instructed students perform better on the test than Scratch-instructed students under our experimental conditions. A Bayesian independent samples Student's t-test returned the Bayes factor 14.09, meaning that the data is approximately 14 times more likely under the alternative hypothesis than the null hypothesis. A Bayes

Group	Test (Algot)	Control (Scratch)
Group size	11	11
Average age	15.5	15.5
Post-test score		
Mean score	3.46	1.55
Std. deviation	1.92	0.82
Task performance		
One task solved	3	2
Two tasks solved	2	0
No tasks solved	6	9

Table 1. The performance of 22 participants on the programming tasks and on the post-test. The effect size was 1.30 (Cohen's d). The maximum post-test score is 8.

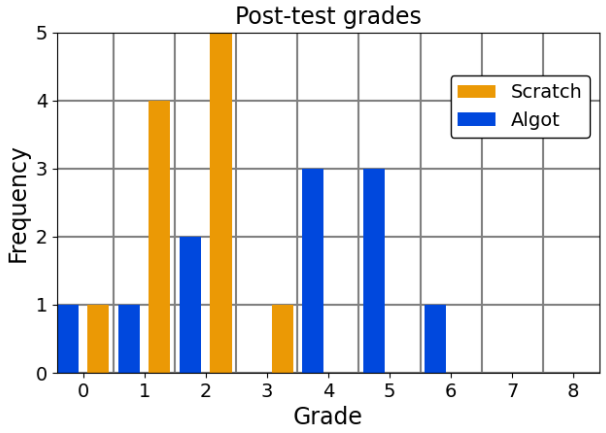

Figure 7. A bar chart showing the post-test performance of the 22 participants. The highest possible grade is 8. The grades that share a column are equivalent.

factor above 10 is considered strong evidence [17]. A classical Welch's t-test, chosen because it makes no assumption about the variances of the populations being equal, gave the p-value $5 \cdot 10^{-3}$, meaning the null hypothesis could be rejected ($p < 0.05$).

To help determine if the performance difference could be attributed to the control group receiving an inadequate introduction to Scratch in our tutorial, we also tested excluding the results of the participants who had no prior experience with Scratch. Under this exclusion, the average post-test scores for the test and control group 3.43 and 1.78, respectively, which is a slightly lower gap (1.65 vs 1.92).

Table 1 also shows the performance on the programming tasks. 45% (5 of 11) of the experimental group could solve at least one task perfectly and 18% (2 of 11) could do so in Scratch. Among those who could not solve any tasks correctly, the performance was varied; some came very close to correct solutions, while others made little or no progress.

18 participants were able to attend two experimental sessions two weeks apart where those from the original Algot group used Scratch ($n = 9$), and vice versa ($n = 9$). The average post-test performance improved from 2.50 to 3.11, but a Bayesian paired samples Student's t-test resulted in $BF_{+0} = 0.38$ against the alternative hypothesis that the performance would improve. As for average enjoyment of programming in Algot and Scratch, the results were similar (2.20 and 2.40, respectively, on a 5-point scale where 4.00 is the most enjoyment). Using the same statistical test, the Bayes Factor was 0.38 when comparing the null hypothesis against the alternative hypothesis that the enjoyment was different.

The participants' textual feedback about both Scratch and Algot was generally positive. Students were particularly appreciative of the video tutorials. A common negative remark about Scratch was that it contains too many blocks and too much selection. A common negative remark about Algot was the query system required too many clicks (currently, to remain in the same scope within an Algot program, the user must frequently re-select the same query result). When students who used both languages were asked if they could suggest improvements by borrowing any features from the other, the responses were varied, with some neutral responses (translated from German) such as "I think both programs should stay the same," and "I find both good. Each in it's own way. But I have have already done a lot in Scratch, therefore it was easier for me to work with Scratch and I made better progress." However, some were particularly appreciative of Algot ("Algot doesn't have to add anything from Scratch. Scratch could be made a bit more clear by offering less blocks and colors," "I like Algot better," and "Algot has a simpler design. Once you have understood it, you really can do it.") and two went in the other direction ("Scratch is clearer," and "I didn't understand Algot at all.").

6 Discussion and Threats to Validity

Since recursion is a topic that is typically reserved for older students, and because highly diverse learning outcomes are commonly reported in CS education [28], we correctly predicted significant diversity in performance and high failure rates among our participants. However, a significant number of participants (5/11) in the test group were able to solve at least one task successfully. These students were able to learn our new programming language Algot, learn about recursion, and apply what they learned to solve a non-trivial programming task, all within the time span of one hour. We believe that with more exposure and practice with Algot, students would be able to perform even better. Altogether, we believe we have found at least promising preliminary evidence in favor of RQ1, that Algot can be used effectively to teach recursion to 14-17 year old students. Further research

may explore longer-term engagement with Algot to fully harness its potential in enhancing programming proficiency.

For RQ2, we found strong evidence (BF_{+0} = 14.09) that within our experimental setting, students who are instructed in Algot rather than Scratch achieve a better understanding of recursion and can demonstrate greater ability to solve simple programming tasks that require recursion. This is in spite of the disparity in prior programming knowledge between the two groups; the participants learned Algot only with a brief tutorial, but most participants were familiar with Scratch (and some noted explicitly in their textual feedback that this was helpful). We believe that the results are significant given that Algot is not only designed with a specific focus on recursion but also functions as a standalone programming language. Given the promise shown in this study, we are optimistic that Algot could serve as an effective tool for imparting broader computer science concepts and computational thinking skills to this age group.

Our ancillary data revealed that the average post-test performance improved on average between sessions (from 2.50 to 3.11), but this improvement was weak and due to the low Bayes Factor, we cannot reject that it happened due to random chance. Note that while similar learning outcomes on both the first and second session would support the null hypothesis, score improvements could be attributed not only to the alternative hypothesis but also to general learning effects. Similarly, the difference in enjoyment of the two programming languages was low and the overall enjoyment seemed neutral on average. Students might report different levels of enjoyment to a different set of tasks, for example ones with clear real-world applications.

Note that our choice of a post-test is a potential threat to the construct validity of the study; as noted before, we developed our own drawing from the Basic Recursion Concept Inventory (BRCI) [11] due to the lack of an established assessment instrument suitable for the study. Our post-test has not been validated and the BRCI has not been thoroughly tested. However, we made every effort to ensure that our test was fair and as unbiased as possible within the context of the study, and while it may not be a comprehensive measure of proficiency with recursion, we believe that it does assess key recursion concepts such as recursive calls, state at various recursion depths, and the understanding of recursive control flow. As for the internal validity, we note that most students were familiar with Scratch but had not programmed in Algot. However, we think that this strengthens the direction of our results since the experimental group performed better on the post-tests and the tasks themselves.

To conclude, we find the results promising and encourage further investigation into the long-term potential and effectiveness in CS education of expressive visual programming languages such as Algot that support direct manipulation and programming-by-demonstration.

References

[1] John R Anderson, Peter Pirolli, and Robert Farrell. 2014. Learning to program recursive functions. In *The nature of expertise*. Psychology Press, 153–183.

[2] Alan C Benander and Barbara A Benander. 2008. Student monks–Teaching recursion in an IS or CS programming course using the Towers of Hanoi. *Journal of Information Systems Education* 19, 4 (2008), 455.

[3] Bryan Braun. 2022. Scratch is a big deal. https://www.bryanbraun.com/2022/07/16/scratch-is-a-big-deal/#scratch-at-scale Accessed: 6 August 2023.

[4] Ibrahim Cetin and Ed Dubinsky. 2017. Reflective abstraction in computational thinking. *The Journal of Mathematical Behavior* 47 (2017), 70–80.

[5] Jeffrey Edgington. 2007. Teaching and Viewing Recursion as Delegation. *J. Comput. Sci. Coll.* 23, 1 (oct 2007), 241–246.

[6] Scratch Foundation. 2023. Scratch Community Statistics. https://scratch.mit.edu/statistics/ Accessed: 6 August 2023.

[7] Carlisle E. George. 2000. EROSI—visualising Recursion and Discovering New Errors. *SIGCSE Bull.* 32, 1 (mar 2000), 305–309. https://doi.org/10.1145/331795.331875

[8] David Ginat and Eyal Shifroni. 1999. Teaching recursion in a procedural environment - How much should we emphasize the computing model? *ACM Sigcse Bulletin* 31, 127–131. https://doi.org/10.1145/299649.299718

[9] Katherine Gunion, Todd Milford, and Ulrike Stege. 2009. Curing Recursion Aversion. In *Proceedings of the 14th Annual ACM SIGCSE Conference on Innovation and Technology in Computer Science Education* (Paris, France) *(ITiCSE '09)*. Association for Computing Machinery, New York, NY, USA, 124–128. https://doi.org/10.1145/1562877.1562919

[10] Bruria Haberman. 2004. How Learning Logic Programming Affects Recursion Comprehension. *Computer Science Education* 14, 1 (2004), 37–53. https://doi.org/10.1076/csed.14.1.37.23500 arXiv:https://doi.org/10.1076/csed.14.1.37.23500

[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] Brian Harvey. 2012. The beauty and joy of computing: Computer science for everyone. *Proceedings of Constructionism* (2012), 33–39.

[13] Brian Harvey and Jens Mönig. 2010. Bringing "no ceiling" to scratch: Can one language serve kids and computer scientists. *Proc. Constructionism* (2010), 1–10.

[14] Wen-Jung Hsin. 2008. Teaching Recursion Using Recursion Graphs. *J. Comput. Sci. Coll.* 23, 4 (apr 2008), 217–222.

[15] A.B. Joshi and R. Pande. 2016. *Advanced Scratch Programming: Learn to Design Programs for Challenging Games, Puzzles, and Animations*. CreateSpace Independent Publishing Platform. https://books.google.se/books?id=_e8ktAEACAAJ

[16] Ivan Kalas and Laura Benton. 2017. Defining procedures in early computing education. In *Tomorrow's Learning: Involving Everyone. Learning with and about Technologies and Computing: 11th IFIP TC 3 World Conference on Computers in Education, WCCE 2017, Dublin, Ireland, July 3-6, 2017, Revised Selected Papers 11*. Springer, 567–578.

[17] Robert E Kass and Adrian E Raftery. 1995. Bayes factors. *Journal of the american statistical association* 90, 430 (1995), 773–795.

[18] Dennis Komm. 2021. Teaching Recursion in High School. In *Informatics in Schools. Rethinking Computing Education*, Erik Barendsen and Christos Chytas (Eds.). Springer International Publishing, Cham, 69–80.

[19] Elynn Lee, Victoria Shan, Bradley Beth, and Calvin Lin. 2014. A structured approach to teaching recursion using cargo-bot. In *Proceedings of the tenth annual conference on International computing education research*. 59–66.

[20] Violetta Lonati, Dario Malchiodi, Mattia Monga, and Anna Morpurgo. 2017. Nothing to Fear but Fear Itself: Introducing Recursion in Lower Secondary Schools. In *2017 International Conference on Learning and Teaching in Computing and Engineering (LaTICE)*. 91–98. https://doi.org/10.1109/LaTiCE.2017.23

[21] CoderDojo Malahide. 2013. Scratch Level 3: Custom Blocks and Recursion. http://coderdojomalahide.com/wp-content/uploads/2013/05/Scratch_Level3_Custom-Blocks-and-Recursion.pdf Accessed: 9 August 2023.

[22] Majed Marji. 2014. *Learn to program with Scratch: A visual introduction to programming with games, art, science, and math.* No Starch Press.

[23] Giulia Paparo, Marco Hartmann, and Mareen Grillenberger. 2021. A Scratch Challenge: Middle School Students Working with Variables, Lists and Procedures. In *Proceedings of the 21st Koli Calling International Conference on Computing Education Research*. 1–10.

[24] Seymour Papert. 1980. *Mindstorms: Children, Computers, and Powerful Ideas.* Basic Books, New York, NY.

[25] Peter L. Pirolli and John R. Anderson. 1985. The role of learning from examples in the acquisition of recursive programming skills. *Canadian Journal of Psychology / Revue canadienne de psychologie* 39, 2 (June 1985), 240–272. https://doi.org/10.1037/h0080061

[26] Irene Polycarpou, Ana Pasztor, and Malek Adjouadi. 2008. A Conceptual Approach to Teaching Induction for Computer Science. *SIGCSE Bull.* 40, 1 (mar 2008), 9–13. https://doi.org/10.1145/1352322.1352142

[27] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, et al. 2009. Scratch: programming for all. *Commun. ACM* 52, 11 (2009), 60–67.

[28] Anthony V Robins. 2019. 12 novice programmers and introductory programming. *The Cambridge handbook of computing education research* (2019), 327.

[29] Dylan Ryder. 2014. Scratch Programming: Advanced Fractal Fun. https://www.edutopia.org/blog/scratch-programming-advanced-fractal-fun-dylan-ryder Accessed: 6 August 2023.

[30] Ian Sanders and Tamarisk Scholtz. 2012. First year students' understanding of the flow of control in recursive algorithms. *African Journal of Research in Mathematics, Science and Technology Education* 16, 3 (Jan. 2012), 348–362. https://doi.org/10.1080/10288457.2012.10740750

[31] Raja Sooriamurthi. 2001. Problems in Comprehending Recursion and Suggested Solutions. In *Proceedings of the 6th Annual Conference on Innovation and Technology in Computer Science Education* (Canterbury, United Kingdom) *(ITiCSE '01)*. Association for Computing Machinery, New York, NY, USA, 25–28. https://doi.org/10.1145/377435.377458

[32] Linda Stern and Lee Naish. 2002. Visual Representations for Recursive Algorithms. In *Proceedings of the 33rd SIGCSE Technical Symposium on Computer Science Education* (Cincinnati, Kentucky) *(SIGCSE '02)*. Association for Computing Machinery, New York, NY, USA, 196–200. https://doi.org/10.1145/563340.563414

[33] Maciej M. Syslo and Anna Beata Kwiatkowska. 2014. Introducing Students to Recursion: A Multi-facet and Multi-tool Approach. In *Informatics in Schools. Teaching and Learning Perspectives*, Yasemin Gülbahar and Erinç Karataş (Eds.). Springer International Publishing, Cham, 124–137.

[34] Joe Tessler, Bradley Beth, and Calvin Lin. 2013. Using Cargo-Bot to Provide Contextualized Learning of Recursion. In *Proceedings of the Ninth Annual International ACM Conference on International Computing Education Research* (San Diego, San California, USA) *(ICER '13)*. Association for Computing Machinery, New York, NY, USA, 161–168. https://doi.org/10.1145/2493394.2493411

[35] 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.

[36] 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 (SIGCSE 2024)*. Portland, Oregon. In Press.

[37] Bret Victor. 2012. Learnable programming: Designing a programming system for understanding programs. *URL: http://worrydream.com/LearnableProgramming* (2012).

[38] Juan Diego Tascón Vidarte, Christian Rinderknecht, Jee-In Kim, and HyungSeok Kim. 2010. A tangible interface for learning recursion and functional programming. In *2010 International Symposium on Ubiquitous Virtual Reality*. IEEE, 32–35.

[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] David Weintrop. 2019. Block-based programming in computer science education. *Commun. ACM* 62, 8 (2019), 22–25.