Algot: An Educational Programming Language with Human-Intuitive Visual Syntax

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

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

***Abstract*—Empirical research suggests that programming language syntax is a common impediment for beginners, a concern that is mitigated to a varying degree by visual programming. In this paper, we introduce a novel visual programming language that is founded on program synthesis and the programming-by-demonstration paradigm. By using an intuitive visual syntax, we show how we can meet our primary goal of providing support to computer programming novices in exploring foundational programming concepts. We present the language's current and planned use in computer science education, provide preliminary evidence for its effectiveness, and discuss its future possibilities.**

***Index Terms*—computer science education, visual programming, programming-by-demonstration**

I. Introduction

Computer programming is often perceived as a demanding cognitive activity which is difficult for beginners to learn and for experts to perform well. In particular, courses on introductory programming are reported to have high failure rates, a contested assertion [1], [2] that is nevertheless used to argue in favor of new paradigms, approaches, and interventions in computer science education.

For over a decade, the block-based programming language *Scratch* has been at the forefront of efforts to make programming easier and more accessible. Originally designed to appeal explicitly to children via youth-oriented media manipulation activities, such as animated stories and games [3], *Scratch* has been an enormously successful experiment with millions of users [4] inside and outside of the classroom. Aside from presenting programming in a simple and visually appealing environment, *Scratch* makes syntax errors—a significant practical concern for beginners—effectively impossible; its drag-and-drop language design means that the user cannot join two blocks if doing so results in an invalid statement [5].

Despite this radical departure in code composition when compared to textual programming, the resulting imperative code resembles familiar textual languages. Since *Scratch* is often considered a stepping stone for beginners before they arrive at more conventional languages [6], [7], keeping similar syntax can be an attractive design property. However, it also means that some conceptual problems with textual programming syntax may persist, even if the users cannot compose code with syntax errors.

In this paper, we present a minimal, non-arithmetic visual language prototype (Fig. 1), *Algot*, that is also motivated by the needs of beginners to programming. However, we have taken a different approach in the sense that our language syntax, both in terms of composition and outcome, imitates its visual semantics to an extreme extent. In doing so, we do not only eliminate syntax errors, but also aim to make programming more intuitive. We will briefly discuss possible applications of the language and our vision for its future use.

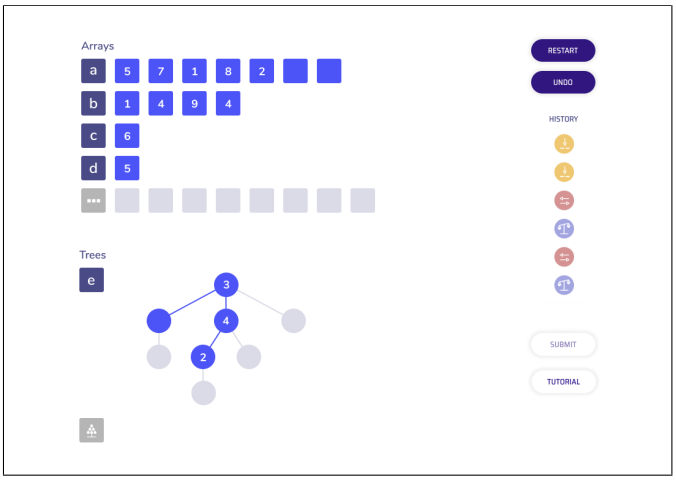

Fig. 1: A screenshot of the *Algot* system. The picture shows a snapshot of a program construction where the user has instantiated several arrays and a tree. For a full system description, see Section III.

II. Background

A. Introductory Computer Science

When preparing an illuminating classroom study that was published in 2006 [8], researchers designed a categorization of CS student errors with items such as "stuck on basic design", "problem naming things", and errors pertaining to specific programming concepts such as loops and boolean expressions. This list, which has since been adopted by other CS education researchers [9], was then used to classify the types of errors that prompted students to ask for help. The category "trivial mechanics", which covers mistakes such as missing semicolons and indentation errors, contained by far

the most frequent problem types that prompted students to seek assistance.

Although the results of this study have been slightly misrepresented,[1] they are important since they quantify to which extent syntax errors impact beginner programmers. Generally speaking, syntax has been considered a cognitive load for students [11], a barrier [12], [13], and a source of frustration and boredom [14]. Efforts to circumvent or change syntax education have taken shape both via innovative programming language design and by making syntax and problem-solving separate parts of the introductory CS curriculum [15].

Algorithm visualization (AV) as an intervention in computer science education dates back to at least the late 1970s with the production of the short film *Sorting Out Sorting* [16]. Since then, hundreds of AVs have been implemented [17], and many of them remain in use in the classroom, often as a complement to other educational materials. AV research is an active field, and recent developments include the JavaScript AV development library (JSAV) [18], a library that allows instructors to construct their own visualizations using modern web technologies.

A review in 2002 by Naps et al. [19] found that AV tools have limited educational benefits unless they engage the user in an active learning activity. Some suggested types of engagement are to ask students to create their own input dataset, to make predictions regarding future visualization states, and to construct their own visualizations. Mirroring Bloom's knowledge-based taxonomy [20], Naps et al. [19] define a non-hierarchical *engagement taxonomy* of AVs that emphasizes interaction and presentation. A study by Hundhausen et al. [21] also found that students do not benefit from passively viewing AVs and that the manner in which students engage with the tool is the most important factor in predicting its effectiveness.

B. Programming-by-Demonstration

The process under which a computer program is generated from an algorithm may be considered a mapping from a syntactic domain (e.g., symbol manipulation or visual operations in a programming language) to a semantic domain [22]. This process is an example of an *implementation*, which in a broad sense means a realization of a specification [23]. Under the *programming-by-example* (PbE) paradigm [24], [25], implementation occurs when the user generates a program by generating input-output examples and some mechanism synthesizes the examples to a program, for example a rule-based expert system or a machine learning algorithm. Distinct demonstrations (e.g., input-output examples) can be used to generate the same program.

Research into PbE is closely tied to its parent field of program synthesis, which develops mechanisms for constructing programs from input-output examples or other forms of user intent. As such, this is not a new line of research. In 1998, Lau and Weld suggested that PbE systems should use a machine learning methodology for program generation; they consider the alternative of rule-based heuristics (popular in early artificial intelligence research) to be "brittle, laborious to construct, and difficult to extend" [26]. In the modern day, machine-learning based PbE is an active area of research within robotics, where the paradigm is instead called *programming-by-demonstration* and is used to allow non-experts to program robots by demonstrating behavior via examples [27] [28].

[1] Leinonen et al. [10] claim that a majority of the issues that students faced were due to "trivial mechanics," but during both years in which the study took place, it was only the plurality of the issues; for instance, the general category "background problems" contained a greater source of errors.

TABLE I: Design principles from Victor's essay *Learnable Programming* [30]. Items (a)–(e): principles for the programming environment. Items (i)–(v): principles for the programming language. Each interpretation is directly from the essay.

The learner can	Interpretation
(a) read the vocabulary	*what do these words mean?*
(b) follow the flow	*what happens when?*
(c) see the state	*what is the computer thinking?*
(d) create by reacting	*start somewhere, then sculpt*
(e) create by abstracting	*start concrete, then generalize*
The language provides	**Interpretation**
(i) identity and metaphor	*how can I relate the computer's world to my own?*
(ii) decomposition	*how do I break down my thoughts into mind-sized pieces?*
(iii) recomposition	*how do I glue pieces together?*
(iv) readability	*what do these words mean?*

C. Creation-Design Connection

Bret Victor, software engineering innovator and interface designer, held an influential lecture in 2011 titled *Inventing on Principle* [29]. In this talk, Victor presents his principle that creators need an "immediate connection" with their creations. To support his claim, Victor used several examples to demonstrate how direct manipulation on graphical components can foster exploration and creativity.

In Victor's follow-up essay *Learnable Programming* [30], he claims that the goals of a programming system are to "support and encourage powerful ways of thinking" and "to enable programmers to see and understand the execution of their programs." To that end, some design principles for new programming languages and their environment are put forward (see Table I).

III. System

A. Language Description

Algot is a minimal programming language that was created under the following design principles:

1) The program state should always be visible to the user (design principle *c* in Table I). Composed commands are both executed immediately and have an immediate visible effect on the program state.
2) Whenever appropriate, the operations of the program share the same syntactic and semantic meaning (design principle *a* in Table I).

(1) To COMPARE two elements, click on them in any order. The selected elements will change their hue to indicate that they are under comparison; if one element is larger than the other, it will be displayed in green while the other one is displayed in red. If the elements are equal, they are displayed as gray.
(2) To SWAP elements, one can either

(a) click on an element that is under comparison. If there exist two such elements, they will now be swapped.
(b) drag one element onto another. The two elements will be swapped.

(3) To CANCEL an element comparison, the user will click on an unmarked part of the screen. This is considered a command and will be displayed as such in the history bar; in programming terms, it is equivalent to explicitly terminating the scope initialized by the previous command.
(4) To INSERT one or more elements anywhere, drag them to the corresponding index.
(5) Similarly, to COPY variables to a data structure, the user will drag it onto the data structure name.
(6) To initialize a NEW DATA STRUCTURE, select the corresponding gray icon.

a) For arrays, its number of empty elements in the array is determined by which gray element is selected. To initialize an empty array, the user initializes an array with only a single element.
b) For trees, the user can initialize a new tree by selecting the gray tree icon (bottom left on Fig. 1). To insert a node to an existing tree, the user should either (i) select a gray child node to insert an node with no value, or (ii) drag a node onto a gray element.

(7) To concatenate (MERGE) two arrays, the user will drag an array name onto another.

Fig. 2: A brief description of the central *Algot* array commands along with each corresponding icon.

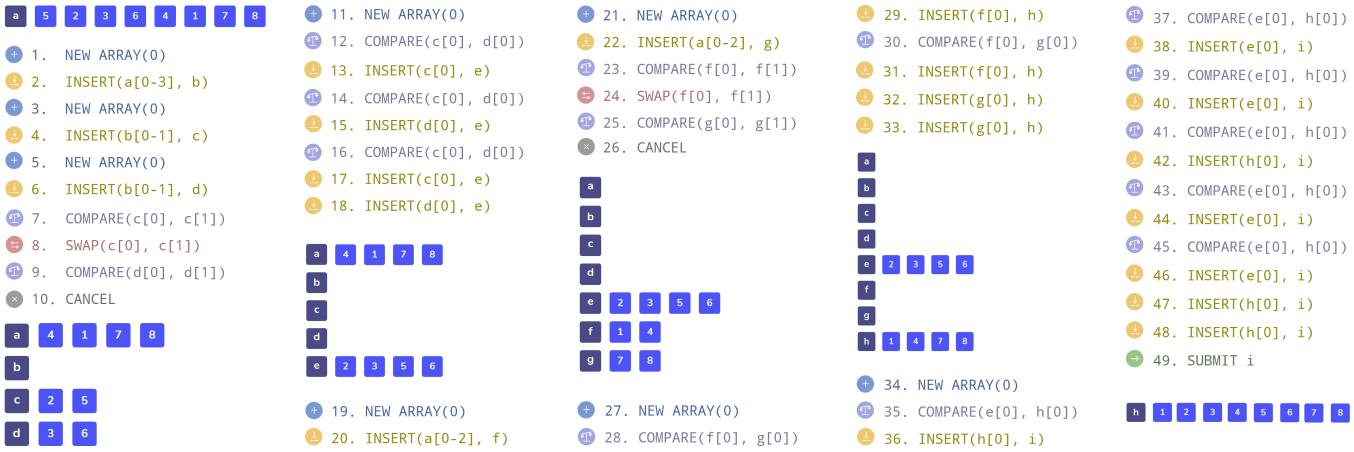

Fig. 3: A demonstration of an *Algot* implementation of the algorithm *merge sort*. The textual representations of the language operations have been colored for clarity and diagrams of the program state have been embedded.

3) The language environment should have a mechanism to automatically repeat similar operations if a pattern is detected.
4) The language should support the creation of algorithm visualizations that are on the highest level of Naps et al.'s engagement taxonomy.

To realize this vision, we implemented the programming environment in the following way:

1) The environment contains the following three modules:
   a) Visual representation of all variables and data structures that have been initialized. This module supports direct manipulation to modify the program state.
   b) A scrollable history bar that displays all the commands that have been composed and executed since the program was initialized. The user can interact with the icons in the history bar by hovering and clicking.
   c) A control bar with some added functionality, i.e., the buttons *repeat* (see below), *undo* and *restart*. A screenshot of this can be seen in Fig. 1.
2) The commands that the language supports can be seen in Fig. 2. For example, the INSERT operation can be performed by dragging an element onto a new area (such as an array index), causing the respective elements to change place. This is a visual metaphor that shows how we use the language syntax to imitate the corresponding semantics.
3) The environment has a *repeat*-button that uses a simple rule table to make programming less tedious for the user. If a match is found in the rule table, the *repeat* button becomes selectable. For example, Rule 1 indicates that if the user has been swapping adjacent elements in an array, selecting the *repeat* button will swap the next

Fig. 4: Four snapshots of the COMPARE and SWAP operations in action.

adjacent pair, provided it exists.

4) As the user creates the programs by directly manipulating visual representations of data structures, the resulting programs function as algorithm visualizations; learners can therefore use the language to both construct and present their own visualizations.

We believe that the advantage of using a rule-based system to synthesize programming instructions, in comparison to a machine learning-based approach (Section II-b), is that the language becomes simpler to use, understand and reason about.

By describing the language as minimal, we mean that its expressiveness is limited. The language supports the following concepts:

a) primitive conditionals, i.e., only one level of nesting. *Algot*'s central operation is number comparison. Comparing two numbers creates a new scope for a single command.

b) iteration by means of program synthesis. When it is possible to select the *repeat* button, the prior commands have followed a pattern that can be continued. Selecting the button will continue this pattern.

c) variable assignment. One-dimensional variables can be initialized by initializing an array of length one, and the variable can be assigned a value with the INSERT operation.

More system information can be found on the project homepage (algot.com).

B. Examples

To give a glimpse into the capabilities of the system, we include two example programs and a discussion of their mechanics. We refer to the visual commands of the language by using the textual labels in Table 2.

The example program in Fig. 3 shows how the user can use the *merge sort* algorithm to sort an input array with 8

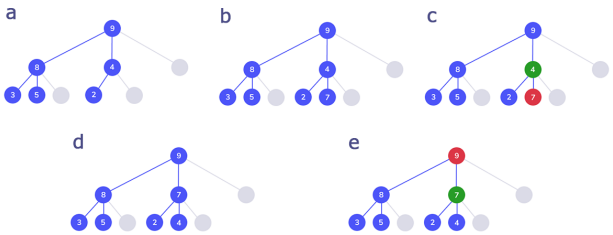

Fig. 5: A demonstration of adding a new value to a *max-heap* using *Algot*.

integer elements. To demonstrate how the operations impact the program state, the code has been illustrated with UI screenshots after certain operations have been executed.

Fig. 5 shows how the program state changes when performing operations on a tree (a). In this case, the user wants to insert a value to a *binary max heap*. To do so, the typical procedure is to insert the element at the end of the heap and then percolate up, i.e., iteratively compare and swap values with its parent until the root of the tree is reached or a parent node with a higher value is found. Here, the user inserts the value 7 as a child to the node 4 (b), then compares (c) and swaps (d) the parent node 4 and the child node 7. Comparing the root node 9 and its child node 7 (e) shows that the root node is larger, so no change is made (d).

IV. Applications

In its current form, *Algot* could be described as an algorithm-focused programming language; it can be used for precise algorithm demonstrations without using or understanding textual programming syntax. As such, a natural application for the language and its environment is to assess its users' procedural understanding of algorithms. Furthermore, the programming-by-demonstration paradigm makes the language highly suitable for knowledge-based assessment; if *Algot* can identify what the user wants, it should also mean that the user understands it themselves.

A second educational application is exploration. Exploration is considered by contemporary learning theorists important for learners [31], and there is ongoing research interest in combining direct instruction with constructivist-inspired exploratory learning activities [32], [33]. In computer science education, *Algot* can be used to support exploration during problem-solving activities, either on its own or as a complement to textual programming tasks.

Beyond education, it is our aim to expand the language to support more general programming. To do this, we plan to start by adding support for a visual handling of arithmetic and boolean connectives, input-output communication with files outside of the system, and additional data structures. Future work includes how to achieve this while preserving our language design principles and keeping the complexity of the system to a minimum.

V. Acknowledgements

We would like to thank the members of the online community *Future of Coding* (futureofcoding.org) for the helpful conversations during the development of the project.

References

[1] A. V. Robins, "Novice programmers and introductory programming," *The Cambridge handbook of computing education research*, vol. 1, pp. 327–376, 2019.

[2] J. Bennedsen and M. E. Caspersen, "Failure rates in introductory programming: 12 years later," *ACM inroads*, vol. 10, no. 2, pp. 30–36, 2019.

[3] J. Maloney, M. Resnick, N. Rusk, B. Silverman, and E. Eastmond, "The Scratch programming language and environment," *ACM Transactions on Computing Education (TOCE)*, vol. 10, no. 4, pp. 1–15, 2010.

[4] D. Weintrop, "Block-based programming in computer science education," *Communications of the ACM*, vol. 62, no. 8, pp. 22–25, 2019.

[5] D. Weintrop and U. Wilensky, "Comparing block-based and text-based programming in high school computer science classrooms," *ACM Transactions on Computing Education (TOCE)*, vol. 18, no. 1, pp. 1–25, 2017.

[6] R. B. Shapiro and M. Ahrens, "Beyond blocks: Syntax and semantics," *Communications of the ACM*, vol. 59, no. 5, pp. 39–41, 2016.

[7] D. Bau, J. Gray, C. Kelleher, J. Sheldon, and F. Turbak, "Learnable programming: blocks and beyond," *Communications of the ACM*, vol. 60, no. 6, pp. 72–80, 2017.

[8] A. Robins, P. Haden, and S. Garner, "Problem distributions in a cs1 course," in *Proceedings of the 8th Australasian Conference on Computing Education-Volume 52*, 2006, pp. 165–173.

[9] P. A. Silva, B. J. Polo, and M. E. Crosby, "Adapting the studio based learning methodology to computer science education," in *New directions for computing education*. Springer, 2017, pp. 119–142.

[10] A. Leinonen, H. Nygren, N. Pirttinen, A. Hellas, and J. Leinonen, "Exploring the applicability of simple syntax writing practice for learning programming," in *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, 2019, pp. 84–90.

[11] R. Lister, "Computing education research programming, syntax and cognitive load," *ACM Inroads*, vol. 2, no. 2, pp. 21–22, 2011.

[12] P. Denny, A. Luxton-Reilly, E. Tempero, and J. Hendrickx, "Understanding the syntax barrier for novices," in *Proceedings of the 16th annual joint conference on Innovation and technology in computer science education*, 2011, pp. 208–212.

[13] A. Stefik and S. Siebert, "An empirical investigation into programming language syntax," *ACM Transactions on Computing Education (TOCE)*, vol. 13, no. 4, pp. 1–40, 2013.

[14] N. Bosch and S. D'Mello, "The affective experience of novice computer programmers," *International journal of artificial intelligence in education*, vol. 27, no. 1, pp. 181–206, 2017.

[15] J. M. Edwards, E. K. Fulton, J. D. Holmes, J. L. Valentin, D. V. Beard, and K. R. Parker, "Separation of syntax and problem solving in introductory computer programming," in *2018 IEEE Frontiers in Education Conference (FIE)*. IEEE, 2018, pp. 1–5.

[16] R. Baecker, "Sorting Out Sorting: A case study of software visualization for teaching computer science," *Software visualization: Programming as a multimedia experience*, vol. 1, pp. 369–381, 1998.

[17] C. A. Shaffer, M. L. Cooper, A. J. D. Alon, M. Akbar, M. Stewart, S. Ponce, and S. H. Edwards, "Algorithm visualization: The state of the field," *ACM Transactions on Computing Education (TOCE)*, vol. 10, no. 3, pp. 1–22, 2010.

[18] V. Karavirta and C. A. Shaffer, "Creating engaging online learning material with the JSAV JavaScript algorithm visualization library," *IEEE Transactions on Learning Technologies*, vol. 9, no. 2, pp. 171–183, 2015.

[19] T. L. Naps, G. Rößling, V. Almstrum, W. Dann, R. Fleischer, C. Hundhausen, A. Korhonen, L. Malmi, M. McNally, S. Rodger *et al.*, "Exploring the role of visualization and engagement in computer science education," in *Working group reports from ITiCSE on innovation and technology in computer science education*, 2002, pp. 131–152.

[20] B. S. Bloom, "Taxonomy of educational objectives: The classification of educational goals," *Cognitive domain*, 1956.

[21] C. D. Hundhausen, S. A. Douglas, and J. T. Stasko, "A meta-study of algorithm visualization effectiveness," *Journal of Visual Languages & Computing*, vol. 13, no. 3, pp. 259–290, 2002.

[22] W. J. Rapaport, "Implementation is semantic interpretation," *The Monist*, vol. 82, no. 1, pp. 109–130, 1999.

[23] R. Turner and N. Angius, "The philosophy of computer science," 2013.

[24] A. Cypher and D. C. Halbert, *Watch what I do: programming by demonstration*. MIT press, 1993.

[25] D. C. Halbert, "Programming by example," Ph.D. dissertation, University of California, Berkeley, 1984.

[26] T. A. Lau and D. S. Weld, "Programming by demonstration: An inductive learning formulation," in *Proceedings of the 4th international conference on Intelligent user interfaces*, 1998, pp. 145–152.

[27] F. Steinmetz, V. Nitsch, and F. Stulp, "Intuitive task-level programming by demonstration through semantic skill recognition," *IEEE Robotics and Automation Letters*, vol. 4, no. 4, pp. 3742–3749, 2019.

[28] S. Calinon, "Learning from demonstration (programming by demonstration)," *Encyclopedia of Robotics*, pp. 1–8, 2018.

[29] B. Victor, "Inventing on principle," https://vimeo.com/36579366, February 2012.

[30] ——, "Learnable Programming: designing a programming system for understanding programs." Published online: http://worrydream.com/LearnableProgramming, 2012.

[31] A. M. Loehr, E. R. Fyfe, and B. Rittle-Johnson, "Wait for it... delaying instruction improves mathematics problem solving: A classroom study," *The Journal of Problem Solving*, vol. 7, no. 1, p. 5, 2014.

[32] J. P. Weaver, R. J. Chastain, D. A. DeCaro, and M. S. DeCaro, "Reverse the routine: Problem solving before instruction improves conceptual knowledge in undergraduate physics," *Contemporary Educational Psychology*, vol. 52, pp. 36–47, 2018.

[33] K. Loibl and T. Leuders, "Errors during exploration and consolidation—the effectiveness of productive failure as sequentially guided discovery learning," *Journal für Mathematik-Didaktik*, vol. 39, no. 1, pp. 69–96, 2018.