Beyond Blocks: Syntax and Semantics

R. Benjamin Shapiro, University of Colorado Boulder
Matthew Ahrens, Tufts University

Millions of kids learning to program today begin with blocks-based systems. Blocks are visual (see Figure 1) representations of program code that add shape, color, and drag-and-drop editing to concrete syntax.

Figure 1: IF/ELSE block from Scratch

Appearing in structured editors as early as 1992 [5], blocks offer a number of affordances: prevention of syntax errors (see Figure 2), type hinting (the diamond-shaped slot in Figure 1 only fits a Boolean-valued expression), and API discovery (block editors typically provide a palette of available objects and methods). Novices benefit from these affordances to overcome some common challenges of learning to program, and there is modest evidence that blocks can improve student learning compared to text [4, 7].

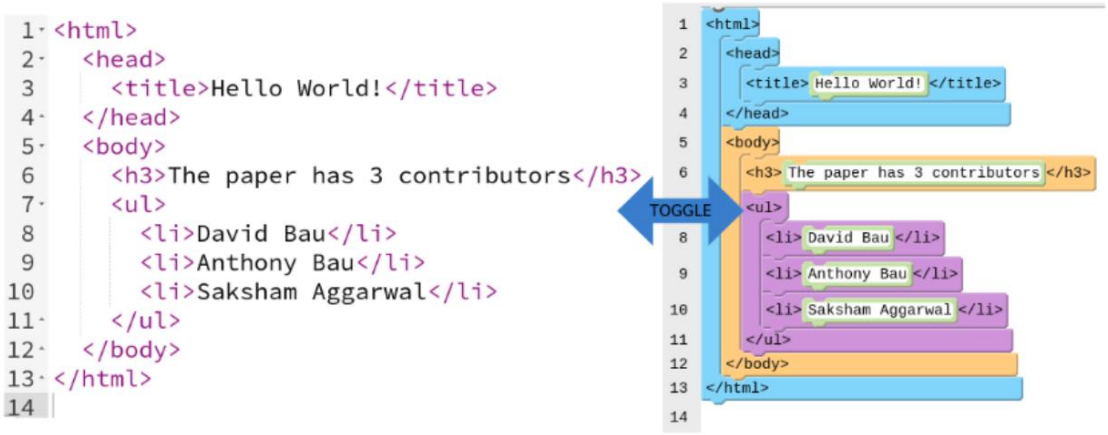

Figure 2: Pencilcode's block-based HTML editor only allows syntactically-correct code and permits toggling between isomorphic text and block representations. Image from [1].

Most users of these beginner-specific systems will probably never want to progress beyond them, but what about those who do? Will they stick with blocks or migrate to text? Will blocks-

based editing become a standard for industrial and academic language developers to consider, or will learners' progress from introductory environments necessarily involve leaving blocks behind?

Mainstreaming Blocks

Several research projects are exploring what general purpose programming tools with blocks-like features could be like. The Snap! Project investigates how blocks can scale up from simple to complex programs, such as those involving complex data structures, user-created abstractions, object-oriented programming, interaction with the Web, or even high-performance computing techniques [3].

Block and text hybrid environments can improve the efficiency and usability of both as shown in multiple current efforts. For example, Greenfoot 3 enables users to manage long segments of Java code through block-like elements called frames, while preserving the ability to readily edit the frames' code within using the keyboard and text, thereby eliminating one of the common complaints about blocks, which is that they are cumbersome for advanced programmers. Work like this may soon illustrate how the future of general-purpose programming tools could include blocks-based structured editing.

Meet the new code, same as the old code

Editing interfaces – and programming tools and culture generally – have proven surprisingly resistant to change. Despite being 40 years old, vi is still a very popular text editor among programmers, even while more user-friendly alternatives abound. Many programmers, including novices, are resistant to graphical interfaces that enhance the experience of programming: one study of entry-level high school programming students found that while the students overwhelmingly perceived blocks-based programming as easier, they also saw it as less powerful, slower, and inauthentic [8]. African-American students' preferences for blocks or text vary sharply depending upon their long-term career goals, with those who identify Programming as their career goal preferring text over blocks [2].

At least for now, it is impossible to participate in existing socio-technical ecosystems of general purpose programming without becoming fluent in reading and editing text, including text written by others. Regardless of whether blocks become the next general purpose standard, learners in the near future will need to transition from blocks toward text.

Blocks → Text; Novice-Specific → General-Purpose

Block-to-text translation tools can improve students' ability to learn traditional languages after using blocks [9]. But no published empirical research has actually examined the experiences that students have as they transition from blocks to text, including the challenges they face as they progress beyond the walled-gardens of novice-specific languages and frameworks.

Current efforts to support transitions from blocks to text are focused on systems building, rather than learning research, and involve transformation of blocks into isomorphic text

equivalents. A recent workshop at IEEE VL/HCC, *Blocks and Beyond* included demonstrations of several tools that involved 1:1 translation of block programs into text equivalents. But this focus on blocks vs text (and blocks *to* text) is a misallocation of our attention. Even though learning to use the concrete syntax of general-purpose programming languages is important to nascent developers, it is not the hardest problem that they face.

With few exceptions, block-based programming environments couple the *syntactic* support of blocks with the *semantic* assistance of purpose-built beginner-friendly APIs. For example, App Inventor drastically simplifies Android development through abstractions that permit creating mobile applications in just a few lines of code. But if learners wish to go beyond what is possible within App Inventor's wrappers of the Android SDK, they must simultaneously abandon blocks for text and learn Java and native Android. A text version of App Inventor's API would only address a limited slice of the problems learners face when they wish to develop more powerful programming capabilities. Focusing on transitions between isomorphic block and text languages (e.g., Pencilcode, shown in Figure 2) ignores the semantic problem.

We need research that investigates how we can support learners' transitions from syntactically- and semantically-assisted novice-specific programming tools into the less-assisted world of general-purpose languages and platforms.

Supporting Conceptual Deepening

We are investigating how to design programming tools that not only support transitions from blocks to text, but that also provide gradual exposure to the semantics of a general-purpose programming environment. To do so, we have created BlockyTalky, an Open Source educational programming environment for networked physical systems. It runs on low-cost computing devices like the Raspberry Pi and enables users to create a wide variety of inventions, from computer music systems to Internet of Things projects. Users can begin with a semantically simplified block environment and then transition to a more powerful and flexible text experience. Block-to-text translation eases this task by enabling users to begin their work in text with semantically equivalent and syntactically similar versions of their block programs, but it is only through programming activities that go beyond those isomorphisms that learners access the richness of the general-purpose environment.

Consider an example: Authoring block-based programs in BlockyTalky involves using simple abstractions for problems like handling physical inputs and communicating over the network with other devices. Figure 3 shows an event handler that asynchronously sends a message (`tickle`) to another device (`elmo`).

Figure 3: BlockyTalky Event Handler

BlockyTalky users can access textual versions of their block programs in a Domain Specific Language (DSL) that we have created to encapsulate event handling, scheduling, networking, and other facets of physical computing that are semantically complex. For example, the blocks in Figure 3 are rendered in the DSL as shown in Figure 4.

```
when_sensor "PORT_1" == 1 do
   send_message("tickle", "elmo")
end
```

Figure 4: Textual version of Figure 4 in the BlockyTalky DSL

We are just beginning research to understand how this block-to-text translation can support the beginnings of student-motivated pathways into additional semantic and syntactic complexity. Our approach is grounded in other researchers' findings that students perceive text-based programming tools as ways to access more powerful capabilities. We respond to these perceptions by limiting what users can do in blocks and their isomorphic textual equivalents, and then offering structured opportunities to gradually learn a more syntactically and semantically rich space.

We envision that students will use our environment in three stages. First, they'll work mostly in blocks, and then begin comparing their own block programs to textual equivalents. Because the semantics of the blocks and text will be identical, the comparisons will help them to understand the syntax differences. Next, we will encourage them to tinker with the text programs – that's when they'll face the changed semantics of the text language. Finally, they will do things in text (like calling functions in text-only libraries) that they can't do in the blocks that we provide.

Students' progress through these stages will primarily be motivated by their own creative goals. For instance, a student building a device to provide an ambient representation of the outside temperature might want to get data from a web service that provides weather information. To do so, they would write textual code to call an HTTP library and then parse the data. Though we could add HTTP blocks to make this easier, we have chosen not to. Students already believe that a major reason to use blocks over text is to create more powerful programs. We leverage this belief in our approach by limiting the functionality available in blocks and encouraging students to try text in order to accomplish their goals. In this way, learning to write text feels like a natural method of accomplishing of students' own goals, rather than something that we impose on them. We will study students' thinking as they work through the three stages just described, identifying areas of confusion along the way and developing new technological and pedagogical supports in response.

Conclusions

Blocks have quickly proliferated within programming tools for novices, offering many kids greater ease of entry to computing. While they are helpful, their use creates a new challenge for learning: learning to *not* use blocks. But rather than studying the blocks-to-text transition in isolation, we need research approaches that take a more holistic view, investigating how the

blocks-to-text transition is a part of syntactically and semantically rich development. We encourage others doing research in this area to investigate how their work on blocks and learning can take a more comprehensive view, looking beyond blocks and into the richer semantic challenges of learning to program.

Acknowledgements

Our work is supported by NSF Awards 1418463 and 1453201, the NCWIT Academic Alliance Seed Fund, and LEGO Education.

References

1. Aggarwal, S., Bau, D.A., Bau, D. (2015) A blocks-based editor for HTML code. Position paper for the Blocks & Beyond workshop at IEEE VL/HCC 2015.

2. DiSalvo, B. (2014). Graphical Qualities of Educational Technology: Using Drag-and-Drop and Text-Based Programs for Introductory Computer Science. *IEEE computer graphics and applications*, (6), 12-15.

3. Feng, A., Tilevich, E., Feng, W. (October 2015). Block-Based Programming Abstractions for Explicit Parallel Computing. In Proceedings of the Blocks and Beyond: Lessons and Directions for First Programming Environments, Atlanta, GA, USA. A VL/HCC 2015 Workshop.

4. Lewis, C.M. (2010). How programming environment shapes perception, learning and goals: logo vs. scratch. In *Proceedings of the 41st ACM technical symposium on Computer science education* (SIGCSE '10). ACM, New York, NY, USA, 346-350. DOI=http://dx.doi.org/10.1145/1734263.1734383

5. Minör, S. (1992). Interacting with structure-oriented editors. *International Journal of Man-Machine Studies*, *37*(4), 399-418.

6. Pane, J. F., & Myers, B. A. (2001). Studying the language and structure in non-programmers' solutions to programming problems. *International Journal of Human-Computer Studies*, *54*(2), 237-264.

7. Weintrop, D. (2015). Minding the Gap Between Blocks-Based and Text-Based Programming. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education* (pp. 720-720). ACM.

8. Weintrop, D., & Wilensky, U. (2015, June). To block or not to block, that is the question: students' perceptions of blocks-based programming. In *Proceedings of the 14th International Conference on Interaction Design and Children* (pp. 199-208). ACM.

9. Dann, W., Cosgrove, D., Slater, D., Culyba, D., & Cooper, S. (2012) Mediated transfer: Alice 3 to Java. In *Proceedings of the 43rd ACM technical symposium on Computer Science Education* (pp. 141-146). ACM.

This is the author's version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution. The definitive version was published in *Communications of the ACM*, Volume 59 Issue 5, May 2016, doi: 10.1145/2903751.