Blocks Versus Text: Ongoing Lessons from Bootstrap

Emmanuel Schanzer, Shriram Krishnamurthi, Kathi Fisler

Bootstrap
www.bootstrapworld.org

Abstract

Block languages need to be studied from many perspectives. One important viewpoint that we do not believe has been explored in the literature is the effect on teachers during professional development workshops. We have acquiring experience comparing the use of block languages against (parenthetical) textual ones in the process of training teachers for the Bootstrap curriculum. We believe our observations are not only interesting in context, but might also lead to discovering interesting new subtleties about the use of blocks.

1. Context: the Curriculum

Bootstrap is a middle-school and high-school program that combines algebra and programming. Students with no prior programming experience begin by designing a simple videogame (given a set of constraints), and then learn programming and algebra concepts to implement it. In particular, the underlying programming language used in Bootstrap is purely functional, mirroring the semantics of algebra.

In the past, Bootstrap has exclusively used functional, student-friendly subsets of Racket (comparable to a functional core of Scheme). Some institutions used the DrRacket programming environment, which is desktop-based, but most use WeScheme, a browser- and cloud-based environment. It's noteworthy that Racket and Scheme have a parenthetical syntax.

Two years ago, Code.org—a non-profit running a large national effort to popularize computing education—chose Bootstrap's curriculum as its Middle School Mathematics (MSM) module. For the first year they used the Bootstrap curriculum verbatim. However, being strong believers in block-based languages, Code.org created a block language for MSM. They based their version on a Blockly-based prototype that the authors built with a student, Spencer Gordon.

This block language is intended to go live in classes in Fall 2015. During summer 2015, we have been training teachers to prepare for this curriculum. Thanks to our extensive experience training Bootstrap teachers with textual languages and the similar activities used in both curricula, we have been able to compare the experiences with these two languages in an apples-to-apples fashion. This document presents some of our preliminary observations, which we believe would be interesting to discuss at the workshop. (We believe the issue of *training teachers with blocks* has not been studied previously in the literature.)

2. Observations

- Blocks are more of an all-or-nothing pedagogical approach than we anticipated. The pedagogy of Bootstrap makes extensive use of on-paper exercises (designed to complement and extend math teachers' existing practices, and also useful in school settings with limited computing equipment). However, blocks do not lend themselves well to paper, and teachers end up wanting a "paper syntax" for writing programs. Our own experience with Bootstrap has found that the on-paper exercises are an essential ingredient for learning, and Code.org has also decided to provide an (optional) paper workbook. However, this imposes two syntactic constraints for the language used on paper:
  - The paper syntax must be as close to the block design as possible. Otherwise, the translation steps from problem to paper to code become a point of friction.

    For example: Math notation includes ternary expressions ($50 < x < 100$), whereas a particular programming language may require students to translate that expression into three binary operators ($50 < xandx < 100$). This transition has been problematic with our block language. In contrast, it has not been problematic in Bootstrap, which also uses binary comparison and logical operators (i.e., we do not exploit the Lispy ability to have variable arity operators, such as `(< 50 x 100)`). What might explain this discrepancy?

978-1-4673-8367-7/15/$31.00 ©2015 IEEE

When using a text-based language, this translation must happen at the textual level. When using a block-based language, this translation can happen at the moment that teachers write expressions on paper (text) or at the moment when they move from handwritten expressions into blocks. In our experience, the text-to-block transition was much more difficult for teachers than the text-to-text transition. This implies that a block language designed to be used alongside written materials may actually increase the need for rigorous textual syntax, rather than reducing it.

Of course, there is another possible approach. One could eliminate the restriction that these operators be binary, and allow a more direct translation. However, designing a block language that allows arbitrary arity may involve subtle user interface effects, which we have not thought through.

- The syntax must be well-specified and unambiguous in order for teachers to perform these translations accurately, and to establish a shared vocabulary for class- and group-discussion. This means confronting the very syntax issues that blocks are designed to gently avoid!

We conjecture that a similar need will be felt for writing code on the board in class, too. We assume this issue has been studied in the other literature on teaching with block-based languages.

In short, pedagogical techniques that are especially common in mathematics appear to require a formal, textual pseudocode—routing around the problem using blocks on the computer only shifts this burden to what happens on paper (and on the board).

- Properly modeling the abstraction provided by functions is not always a good idea! The Code.org language uses modal windows for definitions, which nicely reflects the function definition versus function use distinction. (Named constants—a.k.a., "variables"—are also defined modally.) In practice, this has proven to be a surprisingly bad idea. When writing one function should make use of another, teachers strongly resist the use of abstraction, because they are unable to see what is inside the callee block! This encapsulation trades visual space for working memory, and it is a much more expensive trade than we had anticipated. On the other hand, not having a modal mechanism means it is much harder for learners to track what it is they are doing on the screen at a given moment.
- The goal of our prototype—subsequently adopted by Code.org—was to explore the use of *types represented by colors*. Naturally, there are only so many color gradations that the eye can discern; our view was that once students advance to more sophisticated types, they should move beyond block-based programming anyway. Therefore, the language intentionally offered only a fixed set of types, represented by distinguishable colors.

  That said, some amount of type sophistication appears surprisingly early. Consider a conditional construct: what "color" is it? In fact, it's polymorphic: it takes on a type—and hence a color—based on the content of its branches. There appear to be three ways of handling this:

  - Our prototype gave such blocks a neutral color initially, and used Hindley-Milner inference to determine their type as the program evolved.
  - Code.org's version instead uses a neutral color and, eschewing an inference process, leaves the blocks in that color. This unfortunately grossly violates the precept of block languages, confusing the meaning of colors, thereby creating considerable confusion for learners. This is not to suggest that the inference-based solution is more *usable*; it is certainly more *accurate*, and its usability needs to be studied further.
  - A third possibility, which we are now exploring in our collaboration with Code.org, is to simply have each of these constructs be duplicated across the different colors. This simple expedient appears to gracefully address both of the above problems, but it creates its own issues: (a) the user must pre-select which colored conditional to use, and pay a high switching cost if they chose incorrectly; and (b) it suggests that there are N different notions of the conditional construct, which is at least unsavory (and arguably highly inaccurate) from a programming linguistic perspective.
- One of the strengths of a block-based language for students is that dragging a block may simply be faster than typing an expression. However, teacher workshops themselves rarely suffer from this constraint. We have found that for the teachers in our professional development, writing a program using blocks takes far more time than typing. As facilitators, our goal is often to spend most of our time discussing pedagogy and content, and expect the actual typing to be quite brief. The use of a block language interferes with this, requiring much longer workshops or less discussion of content and pedagogy.

Note that these problems largely disappear in most textual representations. Some of these are specific to professional development of teachers, but some are much more general. We therefore believe each of these represents a line of research necessary into the relative strengths—and designs—of textual and block languages.

Acknowledgements This work is partially supported by the US National Science Foundation, Code.org, Google, CSNYC, and TripAdvisor. We thank Rosanna Sobota and Emma Youndtsmith for their collaboration and insights.