Adapting Student IDEs for Blind Programmers

Emmanuel Schanzer
schanzer@bootstrapworld.org
Brown University / Bootstrap
Providence, RI, USA

Sina Bahram
sina@sinabahram.com
Prime Access Consulting
Cary, NC, USA

Shriram Krishnamurthi
schanzer@bootstrapworld.org
Brown University / Bootstrap
Providence, RI, USA

ABSTRACT

What does it take to adapt a programming environment so students with low or no vision can comfortably use it? Every aspect of the environment needs attention, from toolbars to editors to interactive components. We describe the steps we had to take to adapt WeScheme, the environment used by Bootstrap:Algebra. We also summarize the experience of a group of blind students using the result, and present some lessons for other curricula to consider.

A particular challenge in Bootstrap:Algebra is its heavy reliance on images, which many other media-rich curricula also use. Visual computing is, almost by definition, inaccessible to students with low or poor vision. This poses curricular, legal, and moral obstacles for computing educators who want to use these curricula.

CCS CONCEPTS

• **Human-centered computing** → **Accessibility systems and tools**; • **Software and its engineering** → **Integrated and visual development environments**.

KEYWORDS

visual-impairment, blind, accessibility, IDEs, images

ACM Reference Format:
Emmanuel Schanzer, Sina Bahram, and Shriram Krishnamurthi. 2020. Adapting Student IDEs for Blind Programmers. In *Koli Calling '20: Proceedings of the 20th Koli Calling International Conference on Computing Education Research (Koli Calling '20), November 19–22, 2020, Koli, Finland*. ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3428029.3428051

1 INTRODUCTION

Many curricula [2, 3, 8] use visual media to motivate students. Students find images accessible; they are can be much friendlier than numbers; and they can easily be used to create content that is entertaining and immediately understandable even by non-technical people (such as family and friends), who students sometimes turn to for validation and support. For this reason, many curricula *begin* with visual content, even if they do not focus exclusively on it.

Some curricula depend even more heavily on images. For instance, the Bootstrap:Algebra [8] (henceforth BS:A) curriculum, designed for students in the 12–16 age range, begins by teaching students to write small programs that compose images. The focus of BS:A is (as the name suggests) algebra, and images are a particularly convenient way to teach *composition*, a key concept in algebra. Function composition could (and usually is) taught with numbers, but it can be quite difficult (and uninteresting) to spot errors: at best, it introduces a dependency on the very topics students may be struggling with. In contrast, mistakes in image composition are immediately visible, and students are typically deeply motivated to fix them to obtain the picture they wanted.

Hidden in these statements is, of course, the assumption that students can see what is on the screen. Naturally, this is very problematic for students who have low or no vision. This can be resolved by students with partial sight, e.g., by using screen magnifiers. However, no such partial solution exists for students who are completely blind. (Even these partial solutions are only so useful: for a student with color-blindness, magnification is not much use.)

Computing has traditionally been poor at addressing the needs of such students. The more computing is viewed as a key skill in a growing number of both jobs and academic disciplines, the more this weakness is significantly multiplied. When computing is a required discipline, teachers even face legal requirements to accommodate all the students in their class. Even in the absence of legal demands, our discipline has a moral obligation to accommodate the needs of all students.

This paper describes steps in our attempt to bringing visual accessibility to the BS:A curriculum. We set as our goal that a blind student could usefully complete the first half of the curriculum (the remainder we discuss in §3.3) using a screen-reader. This primarily requires addressing the treatment of visual output (to preserve the benefits discussed above), but requires attention to several other components as well to create a usable experience. We have been able to run a brief evaluation of the resulting system with a group of blind students, with positive results. At the same time there is much more to do. We hope this paper both documents what other implementations need to do (with tips on how to do it), and documents open tasks (for us and others to work on).

This paper is centered around WeScheme [12], the IDE that BS:A uses. WeScheme, shown in fig. 1, runs entirely inside the browser. Any adaptation of the tool can therefore exploit Web technology but is also subject to its constraints. The left panel is the Definitions area, where students write programs that persist (which can be saved to the cloud), while on the right is the Interactions area, which offers a read-eval-print loop (REPL), where students can run interactive expressions. The image shows a function definition (which generates an overlap of two circles, parameterized over their size) in the Definitions area, and a use of that function, to generate an image, in the REPL. We will refer back to WeScheme repeatedly in this paper.

Koli Calling '20, November 19–22, 2020, Koli, Finland

ACM ISBN 978-1-4503-8921-1/20/11…$15.00
https://doi.org/10.1145/3428029.3428051

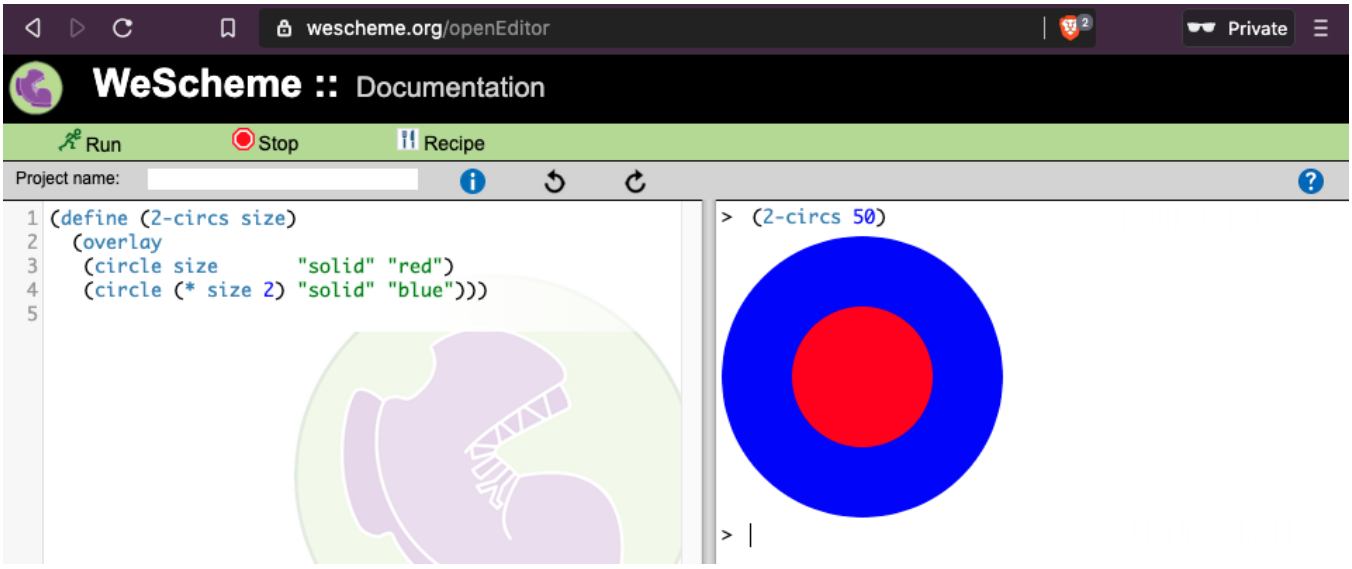

Figure 1: The WeScheme IDE

2 RELATED WORK

Though there is literature on adapting professional IDEs to users with low or no vision, their constraints are very different. Their users already have experience with computing and are more likely to select projects that accommodate their strengths and constraints. Students are a more vulnerable population, and are often given little to no choice on language, tool, or problem.

Relatively little has been written about adapting student IDEs. A notable exception is the work on StructJumper [1], but this focuses very narrowly on code entry, and has the problem we have mentioned in §5, which we work around. A related paper studies editable blocks [7], which is the technology that we use here.

The Quorum project [11] is perhaps the most notable attempt at creating a language with support for such students from scratch. Their effort addresses everything from the details of syntax—whose ordering principle we borrow (§4)—to the IDE, but they have the benefit of creating one from the ground-up and are not constrained by existing implementations. We have also not seen a detailed description by them of the many components of the IDE that need to be adapted.

Handling visual output is a vexing problem. Most curricula that are focused on visual media are largely silent on this problem. Stefik indicates in a recent talk that he is working on something analogous to what we describe above for images, but it is not published.

Finally, readers interested in these topics should look at the materials assembled under the aegis of the Alliance for Access to Computing Careers and the AccessCSForAll [6] projects.

3 IMAGES

We start by focusing on presenting visual output, and return to the other elements of the IDE later. In BS:A, students learn functions that draw various images as well as functions that combine them. A student can, for example, display a triangle rotated 45 degrees, inscribed inside a circle. However, images are opaque datatypes, so a screen reader can't inherently describe them. Annotating them with generic alternate text (e.g., "This is an image") is common but useless. If a picture is worth a thousand words, that's a lot of missing information!

Fortunately, WeScheme includes the entire language implementation stack. We therefore altered the image libraries to build *scene graphs* [4, §6.6], i.e., a "sentence diagram" for pictures. These are flattened into a single image when it's time to render them to the screen. Similarly, we can annotate the scene graph components. The data structure for triangles, for example, is annotated "a triangle of side-length <*side length*>". Rotated images are annotated "an image rotated <*degrees*> degrees: <*image description*>", where the latter is a recursive description of the image. The annotations are flattened into a combined description in the same way that the images are flattened into a combined picture. If the triangle is passed into the rotate function, for example, the description becomes "an image rotated 45 degrees: a triangle of side-length 50".

For instance, the picture in fig. 1 is vocalized as "an overlay: a solid red circle of radius 50 centered above a solid blue circle of radius 100". Notice that the code has functions and variables, none of which are present in the visual (and hence verbal) output. The difference between code and output can grow vastly greater with more composition, iteration or recursion, etc.

This gets us part of the way there, but does not cover all parts and forms of visual output.

3.1 Describing Colors

Many colors have names associated with them, which can be referenced programmatically as strings. "Purple", for example, might map to the RGB value `rgb(128, 0, 128)`. We modify the language implementation to preserve these strings to describe an image, making the result of `(circle 40 "solid" "green")` read as "a solid green circle of radius 40".

However, of course, not all colors have (meaningful) names. A program might construct a color programmatically directly using RGB values, such as `(make-color 66 33 99)`. We can attempt to look this up in a color table, but if the browser doesn't have a built-in name for this color, what do we do?

One option would be to give up, and report all colors as RGB numbers. This is, naturally, unsatisfactory, because it transforms even information that could be rendered meaningfully into abstract information: "255, 0, 0" takes much longer to recognize (and may not be recognized at all) compared to "red". It also makes it difficult for a blind student to answer a word problem that uses color names. Imagine an assignment that asks students to "write a function that takes in a number and draws green triangles of that size"; if the output doesn't say "green", a blind student will have a much more difficult time knowing whether they've solved it. Therefore, we need a way to come up with a name for any color.

Ideally, we want to search in color-space for the nearest color. Fortunately, there is significant research in color theory and color spaces, offering dozens of models. In one model two shades might be very close together (red is close to dark red), while in another they are not (black is closer to dark red). We chose the LAB model [4, §28.9], for two main reasons: First, LAB is device-independent, meaning it does not rely on the particular screen a student is using. Second, LAB represents the space of all colors perceivable by sighted humans, reducing any gap between the populations.

We have therefore modified WeScheme to record LAB values of every named color that has a string equivalent. When the Scene Graph tries to describe a color that it doesn't recognize, it converts the RGB value to an LAB one, and then searches the list of known LAB values to find the nearest-neighbor. Fortunately, we only have a small number of named colors, so this search is fast even on mobile phones. (If we had thousands of named colors, we could speed up search using structures like k-d trees, but we have not felt the need for this in practice.) Our implementation is sufficiently modular that if there were good reason to use a different model of colors, it should be easy to adapt the software to it.

3.2 Describing Imported Images

So far, we have talked about *generated* images. WeScheme, however, also allows students to use images from the Web. For instance,

```
(bitmap/url
  "https://www.bootstrapworld.org/images/icon.png")
```

fetches the BS:A logo. These pre-created images are particularly valuable in BS:A. Students use such images to personalize the curriculum, and in the process take ownership of their product [9]. In many cases, parts of images they want—whose fidelity may be personally meaningful—are literally, or at least (with their knowledge) effectively, impossible to generate computationally (e.g., the globe at the center of the Brazilian flag).

Unfortunately, these Web images do not come with textual descriptions. Remembering the process of obtaining it does not help if the image is in a file from another student (e.g., when pair programming). Thus, directly supporting these images is critical.

Here, AI comes to the rescue. Tools like Google's Vision API can consume an image and produce a textual description of it. While these are not always perfect, they are often extremely useful, and can also exploit information from other sources than just the raw pixels (e.g., search results) to improve over time. We therefore incorporated this API to generate useful image descriptions.[1]

3.3 From Static Images to Moving Pictures

The BS:A curriculum does not stop with images: in the end, students produce a small video game. This requires the description of not only images but also sequences of them.

In principle, we can simply describe each one in order, which we have modified WeScheme to do (to our knowledge, the first educational environment to do this). However, this solution is naturally naïve and unlikely to work well in practice. A smarter solution would keep track of just what has *changed* between frames, and read out just that description—with a vocalization that distinguishes full-image descriptions from descriptions of changes. We leave this as a challenge for future work.

4 NAVIGATION

As mentioned in §1, the entire IDE must function accessibly, not only output images.

The key to making the UI accessible is to annotate the different sections of WeScheme so that screen readers can identify and describe them appropriately. The annotation system for this is called ARIA, which is a detailed specification for how Web developers can express their interface for use with screen readers. In addition to the Interactions and Definitions area, the reader will notice that fig. 1 also shows a toolbar. Not only did we label each of these regions, we also ensured that each button or field in the toolbar was appropriately labeled: each control announces itself with a descriptive name, and an associated keyboard shortcut.

Supporting students who use screen readers also requires that we prioritize different modalities for interaction. While many sighted users use keybindings, these are far more critical for blind users to be able to navigate a system efficiently. Blind users also have some expectations: that objects like histories (like the REPL) and lists (like output) can be traversed. Thus, designing an interface for the blind requires enumerating all the operations that might need keyboard support, prioritizing them, and then allocating them to keys around the existing constraints of the operating system and platform (for instance, WeScheme runs in the browser and F5 reloads the page in all browsers, so we leave this key alone).

To enable quick navigation, we borrow ChatZilla's F6-Carousel approach, allowing the user to hit F6 (or Shift-F6) to rotate focus between the Toolbar, Definitions, and Interactions regions. However, students employing screen readers also use navigation aids that go far beyond the carousel. In addition to linear ("next element") navigation, screen-readers provide a notion of a "search cursor", which is controlled with arrow keys and can be used to navigate the interface spatially. They can also present the user with a list of "landmarks", which exist in both tab-order and spatial-order modalities. These landmarks must be explicitly declared by the interface to be used by the screen reader.

[1] This API is not free. We were fortunate to get credits from Google for limited use, but general use will require some way to pay for it.

The Interactions Area

The Interactions area is a standard REPL: the user enters an expression and the computer replies with an evaluated result (or an error message). Programs typed in the REPL are typically quite short, and the Interactions area is used for student experimentation and exploration. For short programs, having a screen reader announce each symbol, keyword or string as the user types is acceptable.

The REPL "conversation" needs to be annotated as well, in order to be navigable. Again, we borrow from the conventions established by ChatZilla: we use history shortcuts like Alt-1, Alt-2, etc. to have the browser read back previous program-evaluation pairs, and Alt-Up and Alt-Down to page back and forth through previously-evaluated programs. All of this simplifies entering code into the Interactions Area. Even non-image output, however, is complicated.

First, the easy part: we can use ARIA alerts to announce the new content. This even works for error messages, allowing the computer to read the error aloud to the student. But consider the following three terms:

Value	What it Means
`3`	the number 3
`"3"`	the string "3"
`"three"`	the string "three"

A standard screen-reader will read all of those the same way, despite the fact that they are *very* different values. In contrast, our modified WeScheme reads these out respectively as "three", "three, a string", and "three, a string". The latter two are disambiguated by switching to a verbose mode in the screen-reader. This is a matter of efficiency: code would become intolerable if *every* string were read out character-by-character. Programmers must therefore exercise some care to avoid these kinds of situations where possible, though this is true even for sighted programmers: in our experience, almost all students are confused by values like `"3"` (string or number?).

As shown above, numbers are not explicitly tagged but all other types are disambiguated. The types are rendered after the values based on the principle of *semantic prioritization* [10], which is critical due to the speeds at which blind users run vocalization. Some programmers, however, use different *voices* for different types, an aural equivalent of code-coloring; we do not yet support this but can do so very easily.

Programs, of course, produce even more complex values like lists, vectors, functions, and images (which we have already covered in §3). Solving these issues required that we build in accessibility features at the runtime level, so that *every possible datatype* is tagged with the appropriate ARIA label, and those labels bubble up to the IDE to be read by the screen reader. Thus, the value `#3(true "hello" 4)`—a three-element vector whose first element is the Boolean value `true`, the second is the string `"hello"`, and the third is the number 4—is verbalized by WeScheme as

> A vector of size three. True, a boolean. Hello, a string. Four.

The difficulty of implementing this depends on the representation choices in the language implementation. For us it was possible because WeScheme includes the full language implementation, which we were free to modify. Indeed, for very large values, we can even (but have not yet) replace *strings* with *generators*: e.g., the output can read "A list with thousand elements." and take verbal input for how to navigate that large structure. This would be very hard or at least awkward to do with a purely textual system, e.g., if one were to build a `toVoiceOver` analogous to `toString`.

5 WRITING CODE

In the REPL, programs are sufficiently short that they can be entered by standard voice dictation. Furthermore, these "programs" are not meant to persist. In contrast, programs in the Definitions area are edited incrementally over time, and can get quite long, so programmers need to be able to navigate them efficiently.

As we have noted in §4, a critical step is to annotate the interface for the screen-reader. This annotation should ideally be *structural* (as we saw with REPL output, above). Structure is useful for usability, as evidenced by our various computing metaphors (like nested folders, navigable trees, etc.), and even non-computer text (sentences, paragraphs, and the like).

How do we structure code? A natural, generic structure is available in the form of line-numbers. This approach is, however, deeply unsatisfying. It ignores the rich structure that programs have, forcing programmers to keep it in their head; and it is very brittle, because any line-level addition earlier in the program changes all the later positions, rendering a programmer's memory useless.

A better option is to use the program's abstract syntax tree as the basis for navigation. This automatically gives the programmer a *tree*-shaped rendering of their program, easily letting them recursively descend and ascend the program in a *semantically* meaningful way (which lines usually are not). This idea was explored with great promise by Baker, Milne, and Ladner in their StructJumper project [1]. They enable blind programmers to navigate well-formed Java programs, and their study found that several common tasks were made easier with a tree view. In principle, we do the same.

Sadly, this approach has its own significant brittleness: if the program has even a tiny syntax or other parsing error, then it is not well-formed, and therefore does not have an abstract syntax tree. At that point, this reasonable approach to navigation breaks down entirely and the programmer is left with just a block of unstructured text. We must therefore do everything possible to construct a parse tree at all times. However, the history of structured editors shows that this can be overly restrictive.

Our solution is the one part of this paper that does not easily generalize to another IDE: we exploit the syntactic structure of the underlying Racket language. Drawing on its Lisp legacy, there are *two* levels of well-formedness (sometimes called a *bicameral* syntax [5]). At the low-level of well-formedness, all we know is that all the parentheses in the program line up. At the high-level, the program is also syntactically correct, meeting the rules of the language. This distinction is shared with other bicameral languages like those made with XML: *well-formedness* means adherence to the generic XML syntax (e.g., the tags match up), while *validity* means adherence to a *specific* language's rules (which tags are allowed, how they can nest, their multiplicity, etc.).

In Racket, well-formed expressions are either atomic (like `3`, `x`, or `"hello"`) or parenthesized, only some of which are valid: `(if true 1 2)` is both well-formed *and* valid, while `(if true 1)` and `(if true)` are well-formed but *not* valid because an `if` expression must always have a conditional and both then- and else-parts.

Our editor takes advantage of this distinction. It attempts to parse each expression. If it parses, then the expression is verbalized as the language construct— e.g., "2-circs: function definition with one argument: size"—and offers high-level, construct-specific navigation. Otherwise, it verbalizes the term as "unknown"… but can continue parsing the rest of the program. The programmer can switch to textual editing to modify the unknown expression. Therefore, they only need to resort to text editing for the sub-expressions that do not parse.

The only thing this depends on is that the parentheses remain matched. This is easy to achieve because every parenthesis inserted also inserts a corresponding closing parenthesis, and in general we "hide" the parenthesis from the user entirely. Only if they go into "power user" mode to edit individual characters can they modify the parentheses, and when they do so, the editor will not let them commit their change if they leave the term mis-parenthesized.

6 LESSONS LEARNED FROM USERS

As Bootstrap team members, we were once told, "Blind kids just aren't going to make their own videogames if they can't see them". The visual-centricity of the BS:A curriculum therefore creates an obvious challenge for universal use.

In April 2017, we were able to test our modified WeScheme with a group of blind students. Our project was invited to participate in the Alabama STEM Wars event, hosted by Daniela Marghitu's team at Auburn University. STEM Wars is a cross-discipline event aimed at exposing differently-abled students to various STEM activities, and Dr. Marghitu asked us to run the Computer Science track at the event. Most of the students at this particular STEM Wars had low or no vision, which made it a perfect opportunity to test our modifications.

After a welcome presentation, the students split up into three groups, which rotated through brief events in Computer Science, Engineering, and Biology. Most students were in the age range 12–16, and none of them had ever programmed a computer before.

They only had a short time to learn any programming, but every student learned how to use our editor and learned enough Racket to write simple image programs on the computer. Many of the students were very excited by it, and quite a few said that programming was their favorite event of the day. At the same time, we learned several things from this experience, some of which surprised us:

- Students are not familiar with screen-readers. We incorrectly assumed that many students would be comfortable with the basic keyboard shortcuts and spoken feedback used by screen-readers. Computer science teachers working with students with low or no vision should not make this same mistake! This has curricular implications, since it suggests at least some classroom time should be spent building up this knowledge for students.
- There's a threshold code length below which typing is better than overly-structured editing. Students overwhelmingly preferred text-based programming for short expressions. This may be due to lack of familiarity, or it could be because text is just a convenient interface for simple expressions. Students with low or no vision found the overhead of navigation unnecessary for small programs. There's reason to believe that this threshold is different for each student and alters with experience, which underscores a need for an environment that can switch between both modes.
- Students with low or no vision enjoyed making images. We hypothesized that making images would be either uninteresting or downright frustrating for students who couldn't see them. This hypothesis turned out to be completely wrong. A teacher who'd spent his career supporting such students said he wasn't surprised at all; specifically, "*Even when [such] kids are included in a mainstream class, they're often given 'blind work' that is different from the work given to sighted kids, and they feel that*" (our emphasis). The opportunity to write programs that would typically be intended for sighted users was exhilarating.

In short, we learned three lessons. First, structured editors aren't always a win! Second, curriculum writers should not take usability aids for granted and must include materials and scaffolds that introduce these devices—just as they already include materials for familiarizing sighted students with the UIs of Eclipse, Scratch, or WeScheme. And finally, curriculum developers should be cognizant of the tradeoffs between making special accommodations for differently-abled students. There's a balance between completely other-izing and making no accommodations at all, and that balance point may be surprising and subtle.

Acknowledgments. For funding and/or other support, we thank the US NSF, the ESA Foundation, Google, Vinton Cerf, Daniela Marghitu, AccessCSforAll, Richard Ladner, and Andreas Stefik.

REFERENCES

[1] Catherine M. Baker, Lauren R. Milne, and Richard E. Ladner. 2015. StructJumper: A Tool to Help Blind Programmers Navigate and Understand the Structure of Code. In *SIGCHI Conference on Human Factors in Computing Systems*.
[2] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. 2018. *How to Design Programs* (second ed.). MIT Press. http://www.htdp.org/
[3] Mark Guzdial and Barbara Ericson. 2016. *Introduction to Computing and Programming in Python.* Pearson.
[4] John F. Hughes, Andries van Dam, Morgan McGuire, David F. Sklar, James D. Foley, Steven K. Feiner, and Kurt Akeley. 2013. *Computer Graphics: Principles and Practice* (third ed.). Addison-Wesley.
[5] Shriram Krishnamurthi. 2006. *Programming Languages: Application and Interpretation.*
[6] Richard E. Ladner and Andreas Stefik. 2017. AccessCSforall: making computer science accessible to K-12 students in the United States. *ACM SIGACCESS Accessibility and Computing* 118 (June 2017), 3–8.
[7] Emmanuel Schanzer, Sina Bahram, and Shriram Krishnamurthi. 2019. Accessible AST-Based Programming for Visually-Impaired Programmers. In *ACM Technical Symposium on Computer Science Education.*
[8] Emmanuel Schanzer, Kathi Fisler, Shriram Krishnamurthi, and Matthias Felleisen. 2015. Transferring Skills at Solving Word Problems from Computing to Algebra Through Bootstrap. In *ACM Technical Symposium on Computer Science Education.*
[9] Emmanuel Schanzer, Shriram Krishnamurthi, and Kathi Fisler. 2018. Creativity, Customization, and Ownership: Game Design in Bootstrap:Algebra. In *ACM Technical Symposium on Computer Science Education.*
[10] Andreas Stefik, Andrew Haywood, Shahzada Mansoor, Brock Dunda, and Daniel Garcia. 2009. SODBeans. In *IEEE International Conference on Program Comprehension.*
[11] Andreas Stefik and Richard E. Ladner. 2017. The Quorum Programming Language. In *ACM Technical Symposium on Computer Science Education.*
[12] Danny Yoo, Emmanuel Schanzer, Shriram Krishnamurthi, and Kathi Fisler. 2011. WeScheme: The Browser is Your Programming Environment. In *Conference on Innovation and Technology in Computer Science Education.*