Building an Accessible Block Environment

Multi-Language, Fully-Accessible AST-based Editing in the Browser

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

Sina Bahram
Prime Access Consulting
New York, NY USA
sina@sinabahram.com

Shriram Krishnamurthi
Department of Computer Science
Brown University
Providence, RI USA
sk@cs.brown.edu

ABSTRACT

UncleGoose is a toolkit that provides a fully-accessible block environment, for multiple languages. The toolkit generates (1) a block editor that uses standard drag-and-drop conventions familiar to sighted users while also (2) using keyboard navigation and spoken feedback that is familiar to visually-impaired users. The mechanism used creates unique opportunities for (3) separating the description of a block from the visual or textual syntax of that block. This effectively provides a *third representation* (beyond text and blocks), which is spoken aloud and can be tailored to a specific audience. The toolkit lives entirely in the browser and relies on web standards, needing no plugins or server support.

Finally, UnceGoose is implemented as a wrapper for the widely-used CodeMirror library, which is used to display source code using syntax highlighting, bracket matching, and indentation. Any project that already uses CodeMirror can use our toolkit with minimal effort. *By providing a parser that implements our API*, these projects can quickly implement a block editor on top of their text editor, while also getting a fully accessible programming environment that goes far beyond the reading of code.

KEYWORDS

Accessibility, Visually Impaired/Blind Programmers; Screenreader; Navigation; Code Structure, Blocks

ACM Reference format:

Emmanuel Schanzer and Shriram Krishnamurthi. 2019. Building an Accessible Block Environment. In *BLOCKS+, ACM SPLASH BLOCKS+ 2018 Workshop. ACM, New York, NY, USA.*

1 The Promise of Block Environments

From the perspective of most compilers, well-formed programs are trees: the Abstract Syntax Tree (AST) describes the underlying structure of a program. This AST is typically *represented* as textual syntax, due to the ease of rendering and the density of information it affords. However, not all valid text-edits correspond to tree-edits. Deleting a word makes perfect sense when thinking of a program as a long list of words, but doing so can make it impossible to create the AST. This forces programmers to effectively keep track of the tree – in their heads – while working with the text in front of their eyes. This is a challenge even for sighted programmers, who use an array of visual cues such as syntax highlighting, bracket matching, and auto-indenting to help keep track of the AST.

But for visually-impaired (VI) users, these cues are of little help. They are hit particularly hard by the loss of structure [5, 8], and prior work has shown improved comprehension when they are able to navigate the *structure* of the program rather than the syntax [2]. Block environments would seem to be a solution, as they also represent a tree structure. Ironically, blocks rely even more heavily on visual metaphors, making many block tools a step *backwards* for the 65,000 VI students in the US alone [6].

2 The Challenge of Accessible Block Environments

Many of the metaphors on which block programming environments rely are inaccessible or directly in conflict with metaphors for navigating trees with a screen reader. In this section, we discuss a small collection of these metaphors.

2.1 Shifting Focus

When navigating, VI users should always have access to certain information: the *label* ("what am I looking at?"), the *level* ("how deep am I?"), the *size* of the set ("how many are there at this depth?") and the *position* within the set ("which one am I on?"). Blind users have well-established conventions for navigating tree views, with well-known keyboard shortcuts, vocalizations, and behaviors that provide this information [4]. Unfortunately, the few block environments that attempt to be accessible often implement their own, incompatible conventions for navigation.

2.2 Moving Code

Most block environments rely exclusively on drag-and-drop, where a block is moved from one parent to another. This is completely inaccessible to VI users who cannot see the screen.

2.3 Selection v. Focusing

Blind users must distinguish between *what they have selected* and the *location of their focus*. In text editors, this is accomplished via a distinction between "selection" and "cursor". However, due to the concrete metaphor of grabbing a block, this distinction simply does not exist in block editors. A node is only selected when it is grabbed, and attempts to implement "grabbing" for VI users fail to provide a separate metaphor for selection.

2.4 Search is Expensive

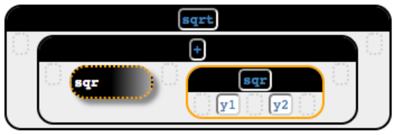
Few block editors provide search tools, instead assuming the user will scan the screen to find what they're looking for. This is a reasonable assumption for most sighted users, but not for VI users who must manually search through potentially hundreds of blocks to find what they are looking for.

3. Implementing an accessible block environment

Our approach begins with an accessible tree view, rendered using the browser DOM. When a change is made to the tree, we make that change to the underlying text, re-parse the program into an AST, and redraw the tree.

3.1 Web Standards

By using on W3C ARIA conventions [1], we communicate the label, level, size and position to the installed screen reader, and UncleGoose behaves just like any other tree view. Users will even hear their custom vocalization settings when using UncleGoose!
We also make ample use of other ARIA and CSS attributes, providing a distinction between focused (`:focus`) and selected (`aria-selected=true`), and even a robust form of *code-folding* (`aria-expanded=false`). This allows users to collapse whole ranges of code, and quickly "skim" without having to navigate through every block or line of code.



One block collapsed and selected, with focus on another

3.2 Describing Nodes with Plain Language

```
function foo(x,y) { return x + y; }
```

How should this be read? Reading every symbol on the line – the status quo for most VI users – seems cumbersome. Our toolkit allows for plain-language labels that put important information first: "foo, a function definition with two inputs". The user can navigate inside `foo` to explore, or move to a sibling. The label is up to the parser, and could be worded instead in Spanish, or using different vocabulary selected for a target age group. Prior work has shown that this form of "semantic prioritization" results in better comprehension for screen-reader users [8], allowing UncleGoose to be a pedagogical *support*, rather than a constraint.

3.3 Search

The AST editor provides a robust search tool, which allows users to search for strings, regular expressions, and even node types (e.g.: "find me all the class definitions").

3.4 But What About Sighted Users?

Our AST nodes exist in the browser DOM, and can be styled using CSS to appear similar to blocks in *Scratch* or *Snap!*. More importantly, we support the drag-and-drop modality. Sighted users who have used other block editors will feel right at home.

3.5 Performance

CodeMirror [3] already provides high performance when working with large documents, by ensuring that only nodes that are *visible* are rendered to the DOM. By rendering the AST root nodes inside of CodeMirror, we have been able to provide good performance and memory use even for large programs.

4. User Studies

Alpha testing began with an April 2016 pilot with blind students in Alabama. We then partnered with AccessCSforAll to conduct a formal user study, in which users were given a set of tasks to complete both with and without the editor. Participants had better accuracy when completing the tasks, were better able to orient when reading through code, and felt better about completing the tasks when using the tool instead navigating plain text. Importantly, these improvements came with no significant change in *completion time* over plain text, even for experienced programmers who use screen readers set to more than 900 wpm.

5 Future Work

This toolkit is available now in a beta version of the Racket IDE used by Bootstrap, and the source code is available on GitHub. We are currently building a language mode for the Pyret programming language, and talking to other teams about doing the same. While editing is already supported, we have also received valuable feedback from VI users about additional features that would support the process. For example, we could use the *static analyzer* from the compiler to allow a variable to quickly point to its definition. This and other features are all slated for future work, as well as research into how the language of node description can better support learners.

ACKNOWLEDGMENTS

We are grateful to AccessCSforAll for collaborating and helping to fund the user studies. We also wish to thank the ESA Foundation, US National Science Foundation, and Vint Cerf for their financial support, without which this project would not exist.

REFERENCES

[1] World Wide Web Consortium. Accessible Rich Internet Applications (WAI-ARIA) 1.1 W3C Recommendation 14 December 2017. Retrieved August 29th, 2018 from https://www.w3.org/TR/wai-aria-1.1/
[2] 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 *Conference on Human Factors in Computing Systems*.
[3] CodeMirror. Retrieved August 29th, 2018 from https://codemirror.net/
[4] Becky Gibson. 2007. Enabling an accessible web 2.0. In *International Cross-Disciplinary Conference on Web Accessibility.*
[5] Sean Mealin, Emerson Murphy-Hill. 2012. An exploratory study of blind software developers. In *Visual Languages and Human-Centric Computing*.
[6] National Federation for the Blind, Retrieved August 29th, 2018 from https://nfb.org/blindness-statistics
[8] Andreas Stefik, Andrew Haywood, Shahzada Mansoor, Brock Dunda, and Daniel Garcia. 2009. "Sodbeans." In *International Conference on Program Comprehension*. pp. 293-294.