Accessible AST-Based Programming for Visually-Impaired Programmers

Emmanuel Schanzer
Bootstrap / Brown University
schanzer@BootstrapWorld.org

Sina Bahram
Prime Access Consulting
sina@sinabahram.com

Shriram Krishnamurthi
Brown University
sk@cs.brown.edu

ABSTRACT

Most programmers rely on visual tools (block-based editors, auto-indentation, bracket matching, syntax highlighting, etc.), which are inaccessible to visually-impaired programmers. While prior language-specific, downloadable tools have demonstrated benefits for the visually-impaired, we lack language-independent, cloud-based tools, both of which are critically needed.

We present a new toolkit for building fully-accessible, browser-based programming environments for *multiple* languages. Given a parser that meets certain specifications, this toolkit will generate a block editor familiar to sighted users that also communicates the structure of a program using spoken descriptions, and allows for navigation using standard (accessible) keyboard shortcuts.

This paper presents the toolkit and a first evaluation of it. While the toolkit allows for full editing of code, we chose to focus strictly on navigation for this evaluation, using the navigation-only study design of Baker, Milne and Ladner. Visually-impaired programmers completed several tasks with and without our tool, and we compared their results and experience. Users had improved accuracy when completing tasks, were significantly better able to orient when reading code, and felt better about completing the tasks when using the tool. Moreover, these improvements came with no significant change in task completion time over plain text, even for experienced programmers who navigate text using screen readers set to high words-per-minutes.

CCS CONCEPTS

• Human-centered computing~Empirical studies in accessibility • Human-centered computing~Accessibility technologies • Human-centered computing~Accessibility systems and tools • Social and professional topics~People with disabilities

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

KEYWORDS

Accessibility; Visually Impaired/Blind Programmers; Screen Reader; Code Navigation; Code Structure; Blocks

ACM Reference format:

SIGCSE '19, February 27-March 2, 2019, Minneapolis, MN, USA
© 2019 Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-5890-3/19/02…$15.00
https://doi.org/10.1145/3287324.3287499

1 Introduction

Reading the textual syntax of a program can be non-trivial. Novice and expert programmers use various visual cues, such as block languages, auto-indentation, syntax highlighting, bracket-matching, and more. However, these cues are useless for the roughly 65,000 blind and visually-impaired students in the US alone [6], who must rely primarily or solely on the textual syntax of the language, as spoken aloud by a screen reader or communicated through a Braille display.

Screen readers are adept at communicating structure, and conventions for navigating tree-like structures (e.g. mailboxes, directories, etc.) are well-defined [4]. Unfortunately, screen readers do not have access to a program's structure. Tokens are read one-at-a-time, and the program is broken up into nothing more than a series of lines. Navigation suffers accordingly, with programmers forced to use arrow keys to read each line of code. Losing the visual cues on which sighted programmers rely is a significant impairment: blind programmers have been shown to have more difficulties navigating and understanding the structure of code than their sighted counterparts [5, 9, 10].

Prior work has shown significant gains when screen-readers are given access to *structure* rather than the raw text. Smith et al. [5] created a language-specific tool to allow blind programmers to navigate the tree structure of *files* in the Eclipse IDE, and Baker et al. [2] created the StructJumper plugin for Eclipse that allows programmers to navigate a Java program's structure.

However, browser-based programming environments are becoming increasingly popular in education. Environments such as Code.org's AppLab, Bootstrap's WeScheme, MIT's Scratch, and others live in the browser [13]. For schools that have adopted Chromebooks, desktop applications are not even an option. These

factors limit the usefulness of prior work, and introduce an additional engineering constraint.

Our tool, CodeMirror-Blocks (CMB), expands on prior work in three significant ways. First, *it is designed to be extensible to other languages*. When provided with a parser that meets certain requirements (described in the documentation), CMB will create a fully-accessible Abstract Syntax Tree (AST) editor for that language, rendered as blocks. Second, *it is designed to run entirely in a web browser*. CMB is built atop the popular CodeMirror library, which is used by thousands of software tools worldwide [3]. Any programming environment that uses CodeMirror can be accessibility-enabled by attaching this tool to the appropriate parser. Finally, *it decouples the textual syntax from the spoken descriptive label for that text*, allowing for plain-language description of fragments of code.

While our tool allows for navigation and editing of code, this first phase of the evaluation is strictly limited to navigation.

2 Related Work

Difficulties for blind developers to explore code efficiently as well as lack of access to advanced IDE features were qualitatively explored by Mealin et al. [5]. Baker et al. provide additional evidence for the claim that blind developers are forced to read entire source code files repeatedly and rely on their short-term memory for complex pieces of information such as a nested conditional within a loop, while also remembering their current depth in said code [2].

2.1 Audio-Based Efforts

Stefik et al.'s work on SodBeans [10] provides both speech and audio cues to notify blind developers of errors, assist in debugging, and convey scope. It lays out three rules for providing lists of information about code: lists must be browsable, short, and place important things first. CMB attempts to strictly follow these rules. To address the concern that audio cues are hard to understand [2], CMB uses audio *and* speech cues, so that users can learn audio cues over time but are never forced to remember them. Our hope is that this hybrid approach will be accessible to novices and useful for experts.

2.2 Purpose-Built Programming Languages

Stefik et al.'s work on the Quorum [12] language shows that syntactic decisions can have a positive effect on accessibility. Unfortunately, many programmers (and students) cannot choose the language they use, and anyway such a language may not be a good fit for the task in other ways (such as its features or semantic choices). Many of the observations of that work may be replicable by custom descriptions in CMB (section 3.4 and 3.5).

2.3 Enhancements to Existing IDEs

Potluri et al. explored enhancing blind developers' efficiency through their work on CodeTalk [7]. CodeTalk makes extensive use of audio cues and aims to make improvements in four areas: Discoverability, Glanceability, Navigability, and Alertability. CodeTalk is a Visual Studio plugin and, as such, it can achieve exacting control over sound effects and much tighter control over said sound effects' timing in relation to speech cues. While we do not evaluate CMB along these four categories, we agree that they are appropriate for blind developers. Evaluation along these lines is an area for future work.

2.4 Structural Information

Screen readers use hierarchical language to convey heading level, and therefore position, in many contexts. Several already-discussed works [2, 5, 7, 10] include this feature. CMB also prioritizes structural information for the blind, and goes further to provide context beyond simple location (see section 3.4).

3 Design and Implementation

CMB had several design and implementation constraints:

1. *It should not be tied to any one programming language.* The editor should be flexible enough to work with different languages (assuming they can satisfy the parser constraints).
2. *It should be easy to integrate into existing cloud-based editing environments*. The editor should not require any browser plugins or extra programs to be installed, and should not require any server-side processing.
3. *It should communicate structure*. As with StructJumper, the structure of code should be navigable via keyboard, announcing relevant information via a screen reader.
4. *It should describe code*, instead of reading syntax. This addresses the same problem as Quorum, in a different way.
5. *It should be performant*. The tool should be responsive and memory-efficient enough to run on tablets, underpowered laptops, etc.

Our editor is built around a continuously-updated AST. The editor has an internal definition of an AST structure, as well as various *ASTNode* types (such as literals, function applications, conditionals, etc.), which can be rendered as text or as a DOM tree in the browser.

3.1 Language Flexibility

The first constraint is addressed through the AST interface. An ASTNode includes `from` and `to` positions (implemented as line-character pairs), as well as a `type` field that declares whether the node represents a conditional, a literal, etc. To use our accessible editor, a language designer must provide a parser that generates the appropriate AST nodes. Additionally, language designers can provide new ASTNode types in order to express semantic elements not defined within the library itself.

3.2 Browser-Only Implementation

To address the second constraint, our editor is implemented entirely in JavaScript, as a wrapper for the widely-used CodeMirror library [3]. By implementing much of the same API as CodeMirror, any project that uses CodeMirror can integrate our editor with minimal effort beyond parsing (which it presumably already has, or must anyway build).

CodeMirror runs on all major browsers, and provides text-handling features like syntax-highlighting, bracket-matching, auto-indenting, and more. While it provides a compelling

experience for sighted programmers, it is completely opaque to users who rely on screen readers. Sadly, the best web-based experience for programmers who use screen readers is essentially an unformatted `textarea`. Fortunately, CodeMirror has a notion of *widgets*, which are arbitrary DOM nodes that can replace a range of text. We exploit this mechanism in CMB, leveraging CodeMirror's robust support for undo/redo, cursor tracking, scrolling, etc.

3.3 Relationship to ARIA

ARIA [1] is a set of attributes designed to enhance accessibility, typically by providing semantic information about content. After parsing the contents of a CodeMirror editor into an AST, we render each root node as a DOM tree, embed a great deal of information via ARIA attributes, and use those trees as widgets to replace the corresponding text range in CodeMirror. These DOM trees allow us to replicate the functionality of StructJumper, expressing the underlying structure of the code entirely in the browser. When navigating a tree, a blind user orients in terms of *label* ("what am I looking at?"), *level* ("how deep am I?"), *size of the set* ("how many are there at this level?"), and *set locus* ("where am I at this level?"). CMB represents each of these — for every AST node — using `aria-label, aria-level`, `aria-setsize` and `aria-posinset`, respectively.

3.4 Describing Structure

One of the key insights of StructJumper was the recognition that a program can be thought of either as a list of tokens (after lexing) or as a tree structure (after parsing). Their paper demonstrates that visually-impaired programmers benefit from navigating the *structure* of the code, rather than hearing the tokens read aloud. CMB does exactly this.

Consider the following simple program:

```
(define (add a b) (+ a b)
(define (factorial n)
  (if (n < 2) 1 (* n (factorial (- n 1))))
```

A CMB user would see the first function rendered as a block:

Figure 1 – Function definition block (collapsed and expanded)

When the block is focused, a V.I. user would hear "`add: a function definition with two arguments: a and b. Level 1. 1 of 2.`" Immediately, they are given a useful, descriptive label, the level, the ordinality and the size. Repeatedly hitting `down-arrow` will read the rest of the function, one part at a time:

```
add
two arguments: a and b
a
b
plus expression, two inputs
plus
a
b
```

Alternately, they can *collapse* the function definition (`left-arrow`), and move on to the next top-level expression. `Shift-left-arrow` collapses *all* nodes, allowing the user to quickly skim even very large programs, expanding only the nodes they are interested in.

At any time, the user can also convert a node into its syntax, and navigate it using normal text controls. This dual-syntax functionality allows for a "syntax when you need it, structure when you don't" approach for both sighted and V.I. users. Prior work has shown this modality to be effective [15].

CMB provides search functionality, allowing the user to search for a term and page through all the matches in the document. Instead of jumping from line to line in the document, however, the user jumps from *matching node to matching node.* A sighted user, after jumping to a random cursor location, will quickly scan the adjacent code to see where they are. For a blind user, however, hearing a line and column number is not a useful way to orient. CMB provides a keyboard shortcut that will read the labels of the ancestors of the active node. For example, instead of hearing "line 812, column 9", they hear "`inside multiply expression, 3 inputs; inside if-expression, inside foo: a value definition.`"

3.5 Describing Code

When parsing a program to generate an ASTNode, the parser may also specify a *label* for that node, which is rendered to the DOM using the `aria-label` attribute. This effectively separates the way a node is *written* in the syntax from how it is *described.* Descriptions can be pedagogical in nature ("`foo: a function definition that is public, static and produces a double`"), and can be tailored for age-level or even spoken language ("`foo: una definición de función que es pública, estática y produce un doble`"). There are many interesting implications of this feature, but space limits our ability to discuss it here.

While CMB will automatically provide location information for all nodes in the tree, it is up to the parser to provide good labels for those nodes. In short, any use of CMB is only as good as the parser with which it is used.

3.6 Performance

By relying on CodeMirror, we achieve performance essentially for free: our widgets are only rendered when they are visible. The DOM nodes rendered and tracked are proportional to the size of the *visible content* instead of the size of the program, resulting in limited memory use and computation. Using the Chrome Task Manager, we found that displaying a large program used only 277MB using this approach, and that even-larger programs never used more than 290MB to display.

4 Study Design

To evaluate CMB, thirteen blind programmers completed three tasks using two browser-based environments: CMB as the experiment and a browser `textarea` element as the control. Readers may point out that more sophisticated methods of browser-based text delivery exist (using `contentEditable` on a styled element, for example), but their support for screen-readers is so poor as to be nearly unusable. We wanted to compare our tool to the most accessible web-based option available. After participants had completed the tasks, we asked them questions about their experience.

We are aware of the challenges faced by those looking to generalize from this sample. Similar studies (including StructJumper) with single-digit sample sizes are common in this space, highlighting the urgency of making programming accessible to more users.

4.1 Participants

Using mailing lists, social media posts, and personal contacts, we recruited 13 participants with an offer of a stipend of USD 150 in exchange for two hours of their time. The number of participants compares favorably to the sizes of other similar studies: the SIGCHI paper on StructJumper, for instance, had only seven. In addition, the community of visually-impaired programmers is small — which highlights the need for work like ours.

Of the 13, three were "novice programmers" (1-5 years of experience), seven had "moderate experience" (5-10 years), and three more were "experienced" (10 or more years). One self-reported as being "somewhat comfortable" with screen-readers, and all others as being "very comfortable". 12 participants were totally blind, while one had profound visual impairment.

4.2 Configuration

Following the format of the StructJumper study, we conducted interviews remotely using screen-sharing in Skype to watch and record as the participants worked through the tasks. Participants used either NVDA or JAWS (latest version as of May 2018) with a current version of Chrome (as a preferred platform) or Firefox (as a fallback). Participants used their preferred screen-reader settings for talking speed and verbosity.

Blind programmers are comfortable hearing the syntax of their preferred language(s) spoken aloud, and typically have their speech settings turned up to several hundred words per minute (one of this paper's authors, who is blind, listens at well over 750wpm!). Programmers who can parse Java syntax into ASTs in their heads at hundreds of words per minute will mask the effects of a tool designed to communicate AST information. To mitigate this effect, we specifically chose a language, Racket, with which few of the participants were familiar.

4.3 Procedure

Participants were asked to provide information about their visual impairment, programming experience, and screen reader use before the interviews were conducted. As with StructJumper, the study was divided into three parts:

1. A short "training session" in which participants learned to use CMB.
2. A series of tasks with and without our tool, using two different code bases.
3. A short, post-session interview.

In the training session, participants explored a small, "training" code base and learned the various key commands and shortcuts needed to navigate it. Once they felt familiar with CMB, participants were asked to follow a series of directions to check if they knew each of the key commands. After this period, the experimental portion of the study began.

Participants were given two sample programs (*Space Invaders* and *Aliens vs. Cows*), each of which had similar levels of structural complexity (maximum nesting depth ~10 levels) and length (~250 lines of code). Both programs represent interactive animations, similar to those use in the widely-used Bootstrap:Algebra [8] curriculum, representing a real-world test case for CMB. Both make use of data structures, recursion, multiple function and variable definitions, `switch`-like condition statements and deeply-nested `if`-expressions. Before completing the tasks, users were given up to 15 minutes to familiarize themselves with the program. To minimize interaction effects, we counterbalanced which program was used with which tool, and which code base was encountered first.

After 15 minutes, the participants were given three tasks modeled on those used by Baker, Milne and Ladner [2]. The first two involved navigating the code to answer questions. These questions were non-trivial, requiring substantial program comprehension and testing the capabilities of the tool as a navigation aid. One was designed to be easier if the user relied on search (the *With Search* task), and the other forced the user to manually-scan the entire program (the *Without Search* task). As with Baker et al., our goal was to determine whether search is an effective modality in the context of a structured code-reader. The third task, *Conditions*, asked the user to indicate which conditions would have to be true in order for a particular line of code to execute. In both programs, this line of code was nested within multiple `if`-expressions, buried within a function definition.

The three tasks for *Space Invaders* were:

1. *Locate With Search*: Find the location in the code where a cow is removed from the list of cows.
2. *Locate Without Search*: Find the location in the code where a cow's direction is updated because it hit a wall.
3. *Conditions:* What conditions have to be true in order for the UFO to be moved left?

The three tasks for *Aliens vs Cows* were:

1. *Locate With Search*: Find the location in the code where `ALIEN-SIZE` is used to determine if an alien hits a bullet.
2. *Locate Without Search*: Find the location in the code that is evaluated when the mouse button is down.
3. *Conditions:* In what situation is the input parameter `w` returned unchanged from the mouse-handler?

Participants were timed as they completed each task, and their answers and duration of the task were recorded. Following the

StructJumper protocol, the specific timing of each task's start and end were based on the moment the interviewer finished reading the question and the moment the participant stated their answer after looking at the code. Due to timing restrictions, we deviated from the StructJumper study in one significant way: participants were given a limit of 5 minutes to complete each task.

Answers were rating on a scale from 0-3 points. For the *Locate* tasks, 3 points were awarded if they found the precise location of the desired expression, 2 points for finding the location of a similar or related expression, 1 point for a loosely-related section of code, and no points if their answer was unrelated to the desired expression. For the *Conditions* tasks, 3 points were awarded for finding the precise conditions necessary for the desired expression to be evaluated, and a point was subtracted for every extraneous or missing condition (until reaching zero). If a participant did not provide an answer in the allotted time, they received a score of 0.

After completing all three tasks for the first code base, participants were asked to provide three ratings of their experience on the Likert scale established by Baker et al. The difficulty and frustration of task completion were rated 1 (not at all) to 5 (very). How well they knew where they were in the code while completing the tasks were rated 1 (no idea) to 5 (always knew where they were in the code).

Once participants completed all three tasks with one program and rated their experience, participants repeated the process with the second program. If they used CMB for the first program they were given a `textarea` for the second, and vice versa. After completing both sets of tasks and reflections, participants were asked to share their thoughts on the process, both with and without CMB.

4.4 Analysis

StructJumper's use of a desktop environment and the context of a single, fixed language make direct comparisons to CMB impossible. However, the similarities in research question allow us to borrow heavily from their analysis.

The two factors at work in our design are the program participants encountered first (*Aliens v. Cows* and *Space Invaders*) and whether or not they used CMB first. We used a 2x2 mixed factorial design, allowing us to model both within-subject and between subject variables. Participants completed a total of 6 tasks, for a total of 78 tasks completed altogether. When analyzing task completion time, we used a mixed-effects model ANOVA with Tool and Participant as model variables. For the semantically anchored scale, we used the descriptive statistics to identify the impact of the Tool. Differences between groups with and without the tool were assessed for significance using two-tailed t-tests.

5 Evaluation Results

We measured the impact of CMB using multiple dimensions, including time-to-complete, accuracy-of-answer, and the semantically-anchored self-reported scales for perceived difficulty, frustration, and orientation. For participants who did not finish the task in the time allotted, we capped their completion time at 5m and gave them an accuracy score of 0.

While the tool is intended for novice users, the difficulty in recruiting novice V.I. users led to most participants being "expert users" with years of experience reading code-as-text. As such, we might expect to see an *increase* in task time for this population.

While not significant, we found that average task completion time was slightly slower when using CMB, but also more accurate. In addition, participants' perception of task difficulty and sense of frustration when completing the task were all better when using CMB, and their sense of orientation within the code was significantly improved.

5.1 Task Completion Time

Participants were *more successful* completing the tasks in the 5m allotted when using CMB. If participants had not been capped at 5m, the average completion time would be greater for every unfinished task. This impact would be disproportionally greater for tasks done without CMB, of which far more were left unfinished (10) than with CMB (3).

	Without CMB		**With CMB**	
	Mean	***SD***	***Mean***	***SD***
Task 1 - Time	2m29s	2m3s	2m39s	1m44s
Task 2 - Time	1m55s	1m28s	2m27s	1m18s
Task 3 – Time	2m56s	1m20s	2m40s	1m30s
Avg. Time	2m28s	1m38s	2m35s	1m19s

As expected, this population was slightly (though not significantly) slower with CMB than without it. Participants completed the *Locate with Search* an average of 10 seconds slower with CMB, and *Locate without Search* wan average of 32 seconds slower. However, the *Conditions* task – the most cognitively demanding of the three - was actually completed an average of 16 seconds faster with CMB than without it.

5.2 Task Score

	Without CMB		**With CMB**	
	Mean	***SD***	***Mean***	***SD***
Task 1 – Score	2.31	1.11	2.62	0.51
Task 2 – Score	2.39	1.12	2.53	0.88
Task 3 – Score	1.85	1.34	2.23	1.17
Avg. Score	2.18	1.19	2.46	0.88

When using CMB, participants scored higher — and more consistently so — on every task. The largest difference in task score was found on the cognitively-demanding *Conditions* task. When using CMB, 9 (out of 39) tasks lost points due to inaccurate answers, compared to 6 without it. However, CMB resulted in less than one-third the number of incomplete tasks (10) than traditional text (3).

Participants lost points in the *Locate* tasks because they found a related part of the code but not the precise location. These specific mistakes involved participants searching for a particular term, finding it, and then reporting it as the answer without checking to see if this term was being used in the right place.

More participants lost points on the *Conditions* task than any other task. Of the 13 participants in the study, 8 received the full score when using CMB, compared to only 6 without. Of those who lost points but still managed to complete the task, every point lost was due to participants failing to consider the impact of nested `if`-statements.

5.3 Participant Experience

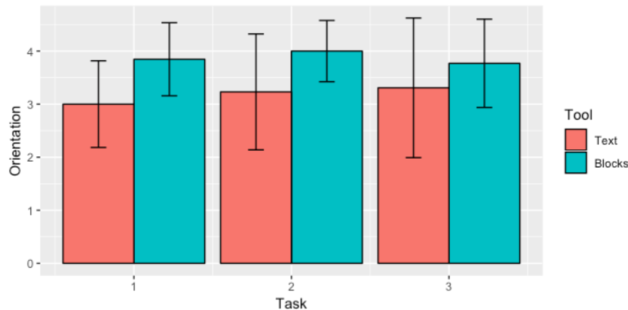

Figure 2 – Orientation was significantly improved

When describing their ability to *orient* themselves while completing the tasks, users felt that CMB was significantly better than reading raw text ($p < 0.005$). Scores of 3 or above were more far more frequent with the tool (38) than without it (29).

	Without CMB		With CMB	
	Mean	***SD***	***Mean***	***SD***
Difficulty	1.34	1.03	1.31	0.98
Frustration	1.31	1.28	0.77	1.09
Orientation**	3.18	1.07	3.85	1.07

Users also reported lower levels of *frustration* when completing tasks - as well as the perception that tasks were less difficult – when using the tool. These differences were not significant.

5.4 Qualitative Results

After completing the tasks, participants were asked a series of open-ended questions about their experiences, to get a sense for what ways (if any) they found the tool helpful, frustrating, or useful. Several common themes emerged.

5.4.1 Orienting Better. Nearly every participant commented that CMB helped them orient themselves when reading code. Some participants attributed this to the fact that browsing with CMB naturally enforced an understanding of structure ("*everything is arranged so you hear the structure just by going to the next node*"). Other participants made ample use of "orientation shortcuts" in CMB, which would read all of the ancestors of a particular node. Several pointed out that "location" in a text editor is often defined in terms of line and column numbers, and said they preferred CMB's orientation within the AST:

"*It's way more useful to hear that what I'm looking at is inside an if-expression, which is inside the definition of the hitting-wall function, rather than just hearing that I'm on line 215.*"

5.4.2 Focus on Structure, not Syntax. Reading the structure of a program is a different task than reading the syntax, and many participants remarked on how freeing it was to be able to focus on the structure: e.g., that "*The TreeView was really nice. I didn't have to think about indentation to form a tree on my own*" or "*I wish I had this tool for when I'm exploring new languages! I liked that it always gave me a consistent view of the code…I'm often a little distracted if I get different indentation, or if there are a lack of spaces I get funny line wrapping with my braille display.*"

This effect may have been enhanced by the fact that virtually none of the participants were familiar with the syntax of the language. One might expect, for example, that the syntax burden would be less of an issue for Java programmers reading Java code; on the other hand, it better reflects the experience of a novice. Indeed, allowing students to focus on structure instead of syntax is one of the goals of block languages like Scratch. Replicating that effect in in a way that is accessible for visually-impaired users is an important goal for this study.

5.4.3 Perceived Speed. Many participants indicated that things "felt faster" when using CMB. In particular, they liked the ability to collapse blocks and "skim": *"Loved the collapsed-all! Really handy to skip over to skip over stuff I don't care about. Very quickly, I knew that the then clause of the if-expression was something I could skip. Being able to just collapse it was awesome.*"

6 Future Work

While this evaluation focused exclusively on using CMB to *navigate* code, future user studies will focus on the *editing* functionality. And given CMB's ability to let users manage their cognitive load (choosing when to work directly with syntax and when to avoid it), it would be useful to evaluate the tool when examining languages with which the users are already familiar.

The user studies described here also provided extensive feedback about areas for future development. The simple search functionality implemented for this study was clearly a limitation, and multiple users communicated that a more robust search feature would have been helpful when completing these tasks. Additionally, several users asked for a "glances" stack, which would allow them to hit a key and have their current position saved. After further exploration, they could hit a different key a quickly return to the location at the top of the stack.

Finally, it would be valuable to explore the impact of using CMB as an IDE for *sighted* users. Having the computer read a description of a block in an age-appropriate language, or different natural language, could have major implications for all learners – not just those with visual impairments.

ACKNOWLEDGMENTS

We are grateful to AccessCSforAll for their enormous efforts to recruit participants, and to the participants themselves for their time and feedback. We also thank Vint Cerf, the ESA Foundation and the US National Science Foundation for supporting this work.

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
[7] Venkatesh Potluri, Priyan Vaithilingam, Suresh Iyengar, Y. Vidya, Manohar Swaminathan, and Gopal Srinivasa. 2018. CodeTalk: Improving Programming Environment Accessibility for Visually Impaired Developers. In *Conference on Human Factors in Computing Systems*.
[8] Emmanuel Schanzer, Kathi Fisler, and Shriram Krishnamurthi. 2018. Assessing Bootstrap: Algebra Students on Scaffolded and Unscaffolded Word Problems. In *Symposium on Computer Science Education*.
[9] Ann C. Smith, Justin S. Cook, Joan M. Francioni, Asif Hossain, Mohd Anwar, and M. Fayezur Rahman. 2003. Nonvisual Tool for Navigating Hierarchical Structures. In *SIGACCESS Accessibility and Computing*. no 77-78, (pp. 133-139). ACM.
[10]Andreas Stefik, Andrew Haywood, Shahzada Mansoor, Brock Dunda, and Daniel Garcia. 2009. "Sodbeans." In *International Conference on Program Comprehension*. pp. 293-294.
[11] Andreas Stefik, Christopher Hundhausen, and Robert Patterson. 2011. An empirical investigation into the design of auditory cues to enhance computer program comprehension. In *International Journal of Human-Computer Studies*, no 69 (pp. 820-838).
[12] Andreas Stefik,, Susanna Siebert, Melissa Stefik, and Kim Slattery. 2011. An empirical comparison of the accuracy rates of novices using the Quorum, Perl, and Randomo programming languages. In Workshop on Evaluation and Usability of Programming Languages and Tools (pp. 3-8).
[13] Wilson, C. (2014). Hour of Code: We can solve the diversity problem in computer science. ACM Inroads, 5(4), 22-22.
[14] Anja Thieme, Cecily Morrison, Nicolas Villar, Martin Grayson, and Siân Lindley. 2017. Enabling Collaboration in Learning Computer Programing Inclusive of Children with Vision Impairments. In Proceedings of the 2017 Conference on Designing Interactive Systems (DIS '17). ACM, 739-752.
[15] Weintrop, D., & Holbert, N. (2017). From Blocks to Text and Back: Programming Patterns in a Dual-Modality Environment. In Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education (pp. 633-638). New York, NY, USA: ACM.