Programming Education for Blind and Low Vision Users: Beyond Reading the Screen

Joshua G. Lock
joshua.lock@kcl.ac.uk
King's College London
London UK

Neil C. C. Brown
neil.c.c.brown@kcl.ac.uk
King's College London
London UK

Michael Kölling
michael.kolling@kcl.ac.uk
King's College London
London UK

Abstract

Learning to program is often considered a difficult task, but this difficulty is multiplied for blind and low vision users, especially among young learners who may be learning about their assistive technologies (such as screen readers) alongside learning to program. In this position paper we argue that the entire paradigm of screen-reading is a flawed way to construct an auditory interface for users with limited or no functional vision. We contend that block-like paradigms – despite historically being inaccessible – actually point towards a superior design where the abstract syntax tree is used to produce an auditory interface directly, rather than relying on reading out a plain-text rendering of the syntax tree. We believe that this can have benefits for learners and professionals alike.

CCS Concepts

• **Human-centered computing** → **Accessibility**; • **Social and professional topics** → **Computing education**; • **Software and its engineering** → **Integrated and visual development environments**.

Keywords

Non-visual accessibility, Programming education, Program editing

ACM Reference Format:
Joshua G. Lock, Neil C. C. Brown, and Michael Kölling. 2025. Programming Education for Blind and Low Vision Users: Beyond Reading the Screen. In *25th Koli Calling International Conference on Computing Education Research (Koli Calling '25), November 11–16, 2025, Koli, Finland*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3769994.3770018

1 Introduction

It is a cliché that every programming education paper begins "Learning to program is hard" [5, 9, 15]. However, it is even harder for blind and low vision users, who must perform the same learning in programming interfaces designed for sighted users, typically via assistive technologies. This challenge is increased again among young learners, who may be simultaneously learning to master the assistive technologies, while also trying to learn to program [19, 20].

One of the most common assistive technologies, especially among users with minimal or no functional vision [16], are screen readers. Screen readers have particularly awkward design incompatibilities with program code: they tend to not read out punctuation (which can be vital for the meaning of program code), they deal awkwardly with white spaces for indentation, they read too much or too little of the code at once, and many programming tools (especially those designed for younger or novice learners) are inaccessible or have poor accessibility via screen readers. Furthermore, different screen readers (and different versions of the same screen reader) provide different presentations of the same code.

In this position paper we contend that the basic premise of reading a screen full of textual program code produces a poor usability outcome for blind and low vision programmers, and that this poor performance is not the result of sub-optimal design of the software at hand, but an intrinsic problem of the screen-reading model on which these assistive programs are based. We examine many ways in which screen readers fail to meet the needs of such programmers, and then we argue for an alternative approach, built on the structured programming ideas of paradigms such as block-based and frame-based editing.

2 Background

To fully explain our argument, we must first provide some technical background on program code, then explain different levels of vision, before explaining the most common way that blind and low vision users interact with program code: screen readers. From this we can argue – backed by literature – why this is the wrong design, and offer an opinion on a better design.

2.1 Program code

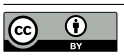
There are multiple different possible representations of computer programs, such as flowcharts or data-flow programming. In this paper we are specifically interested in the most common set of representations: those that use an abstract syntax tree (AST) to form the internal representation of the program.

For text-based languages, such as Python, Java, C, etc, almost all developer tools (such as compilers, editors and IDEs) use an AST to facilitate their operation. This AST is derived from the plain-text source code using a parser. For example, 1 shows a simplified view of how a toolchain might translate some plain-text Python source code into an AST.

Block-based languages and some other editors, such as structure editors and projectional editors, aim to edit the AST directly rather than relying on a parser to derive it from a plain-text representation.

2.2 Human levels of vision

People can have a variety of levels or types of vision. Many users can see the entire screen, and read all text on the screen. Some types of vision, such as limited colour discrimination (colour blindness), have a relatively limited impact on the ability to interpret graphical



Koli Calling '25, Koli, Finland

ACM ISBN 979-8-4007-1599-0/25/11
https://doi.org/10.1145/3769994.3770018

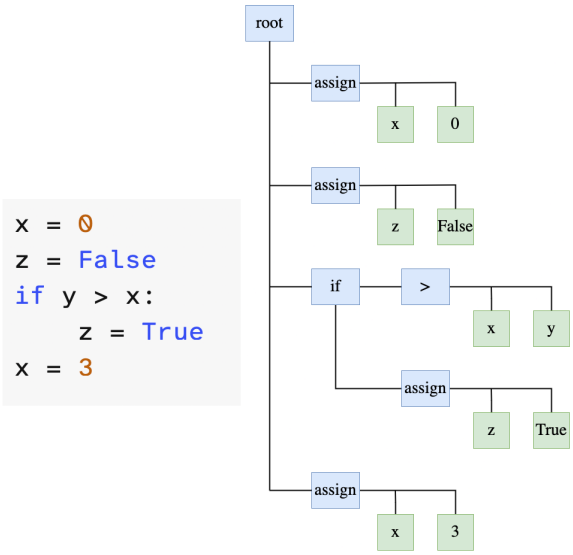


Figure 1: The left-hand side shows a simple snippet of source text. The right-hand side shows an AST representation of the same code.

user interfaces – mainly thanks to human interface design guidelines which suggest to avoid distinguishing attributes and states solely by colour to prevent these users experiencing difficulty[1].

Other types of vision may range from being completely blind, to having restricted fields of view, blurry, or partially obscured vision. In most countries, legal blindness often includes the latter; this is contrary to common popular understanding which associates the term "blind" with the absence of any functional vision. In this paper we will primarily refer to blind (i.e. no or minimal functional vision) and low vision (i.e. heavily obscured sight, or unable to see details except with significant magnification), and will use the common acronym BLV – Blind and Low Vision – to refer to both groups together.

2.3 Screen readers

A variety of assistive technologies are available to computer users. In this paper we focus on screen readers, which are the most popular assistive technology among blind users (used by 95+% of blind or low vision users [16]) and commonly used by programmers, often alongside a Braille display. Albusays and Ludi [1] surveyed 69 BLV programmers, and found that the majority use a screen reader and almost half also use a Braille display, which can help supplement screen readers for detailed reading of program text [17].

Screen readers produce audio interpretations of a user interface. For graphical user interfaces, these interpretations must map spatial, two-dimensional interfaces into linear audio streams. Raman [25] compares the way a screen reader interprets a graphical user interface to performing direct translations of native language phrases to a foreign language — not incorrect, but often somewhat awkward.

[1]For example, see the 'Inclusive Color' guidance in Apple's Human Interface Guidelines [12] and the 'Accessibility Considerations' in GNOME's Human Interface Guidelines [23].

For example, if a screen reader switches focus to Microsoft Outlook for Mac, the VoiceOver screen reader will narrate the following[2]:

> Outlook Inbox bullet username at domain dot tld window Message List table No selection.
> You are currently on a table. To enter this table, press Control-Option-Shift-Down Arrow.

At this point, the user has the following options:

- have the screen reader read all content from the message list table (using the read all command — Ctrl+Option+A). This causes the screen reader to read the table from top-to-bottom, including the message age indicators (today, this week, etc) and the full contents of each email, or
- have the screen reader start reading one email at a time from the message list. This is achieved by navigating into the message list (Ctrl+Option+Shift-Down Arrow) and navigating through messages (Ctrl+Option+Down or Ctrl+Option+Up).

These two options demonstrate how the screen reader translation of complex graphical interfaces often oscillates between presenting less information than would be available visually—as in when the application is first started—and presenting a stream of information that may be more than is desired—as in when trying to locate a recent email without wanting to listen to the full contents.

Screen readers can be particularly challenging for young learners, who may be learning simultaneously how to use assistive technologies—which are complex technologies with many keyboard sequences to memorise—and learning how to program, thus increasing the challenge compared to sighted learners. Some users of screen readers may receive no instruction on how to use their screen reader (especially in developing countries [26]), or are taught how to use them by non-expert sighted tutors, and thus may be unaware of the full range of features and may be using the screen reader in a suboptimal way.

The challenges of screen reader use can be further amplified by social aspects of the learning environment, for example in mixed visual-ability learning environments—where there are learners with different visual disabilities and those with no visual disabilities—users of assistive technologies may be self-conscious of being observed using their assistive technologies [29], particularly if the assistive technologies appear to make the task at hand harder than for their peers. Similarly, when those with sensory disabilities are provided separate tools with additional support for their sensory disability this can lead to feelings of othering and exclusion. Worse yet, learners with sensory disabilities may be completely excluded from learning due to their access needs being unsupported by the learning environment or technologies employed therein.

Screen readers look to provide a generic interface that is suitable for all applications. To achieve this, the APIs generally have knowledge of common graphical user interface components: text fields, buttons, labels, date controls and so on. However, this is typically a finite set, and can prevent the use of novel interface controls or displays as might be used in novel programming interfaces. For example, a flowchart editor is not easy to represent to a screen-reader [4] and, outside of programming, map displays have been

[2]Using New Outlook for Mac version 16.98 on macOS Sequoia 15.5.

found to be inaccessible by screen readers [10]. Modern GUI systems enable even custom interface controls to be made accessible – but this must be explicitly implemented by a programmer who both understands the need to make interface elements accessible and can provide useful accessibility cues [8].

2.4 Audio-first interfaces

Conversational interfaces, in which both input and output are entirely in speech, are now well known and commonly used. Examples include voice assistants, such as Amazon's Alexa[34] and Apple's Siri. In this paper we are interested in audio output interfaces, where the input is performed with keyboard interactions rather than spoken input, especially those which are suited to programming.

Raman [24] proposes that direct access to the application's context should be used to create an auditory first user interface on-par with the application's visual interface. Such an auditory user interface is implemented for Emacs in Emacspeak. Emacs is well suited as the basis for an auditory-first environment, as it is inherently extensible and has access to many "modes" which provide functionality within Emacs equivalent to typical graphical applications for tasks such as email, calendaring, messaging, multimedia and programming. While this approach to auditory user interfaces has not become common in mainstream programming software, our proposal matches the intent to provide an auditory first interface as an equal peer to the graphical interface.

2.5 Historical patterns

Our literature survey, notably Raman [25], and discussions with BLV programmers gave us the sense that non-visual accessibility of computer applications was more successful before the ubiquity of graphical user interfaces (GUI). When interfaces were limited to a modest character display screen-reader technology was able to interpret and present the screen's content to its user without any assistance from the application, operating system (OS), or interface toolkit being presented. With the introduction of graphical interfaces, the complexity of what a screen reader needs to translate and present increased significantly, resulting in a need for OSs and interface toolkits to build-in features to expose data for a screen reader to present. Consequently, there was a period where GUIs were *not* accessible to screen reader users while the requirements were understood and the OS and toolkit features were built. While things have improved in recent times, thanks to significant effort from OS and interface toolkit creators (see, for example, Fleizach and Bigham [8]), many applications still have poor or no accessibility to screen reader users.

Notably for our domain and informing our position, block-based programming systems perpetuated the common pattern of designing a system, then figuring out whether and how to make it accessible at a later time. The heavy reliance of block-based systems on mouse input harmed accessibility for screen reader users, and others, even though we believe the fundamental model of block-based programming systems could improve accessibility.

3 Reading the screen is the wrong approach

The fundamental problem with screen readers is inherent in their name: they read the screen. Users without vision do not care that the screen exists. They would benefit from an optimal audio representation of their program code — there is no need for it to relate to a screen. Using the screen representation as the structural basis for the audio output is very likely to lead to sub-optimal results.

For low vision or partial-vision users, who can see some aspects of the screen, it is more apparent why reading the screen might be useful, as it allows detail which cannot be visually accurately perceived to be filled in as needed. But even in these cases, it is only necessary to be able to easily relate the audio to the visual representation; it is not necessarily the case that they must follow the same order and give endless details on layout.

For program code specifically, the problem is shown on the left-hand side of 2. A textual representation is used as the canonical representation, and from this the program code is parsed; but the screen reader is also based on the program text.

3.1 Speech ordering

Consider this Java code snippet:

```
1  public void setWidth(int w) {
2      this.width = w;
3  }
4  public void setHeight(int h) {
5      this.height = h;
6  }
7  public void setX(int x) {
8      this.x = x;
9  }
10 public void setY(int y) {
11     this.y = y;
12 }
```

Imagine a screen reader user wants to move down to the `setY` method. The user begins by placing their text cursor at the start of the first line. The screen reader reads out "public void set width"; at the point the user hears the "width" part, they know they are on the wrong line and can press down to move to the next line, hear the method's body being narrated, and continue to press down until they hit the next method definition. They then hear "public void set height" and on hearing "height" they press down again. By the time they reach the right line, they have heard "public void" four times, even though it is unrelated to their task, and heard at least partial narration of several other lines of code.

There are other solutions to this specific issue (e.g. a keyboard shortcut to jump to a method by name or the 'find' functionality) but the principle applies much more widely. There is a design principle in accessibility guidelines called "front-loading"[3]: the most important or distinguishing text should be at the beginning of an item so that it is read first. This often does not correspond to the first token(s) on a particular line of program code.

Similar front-loading concepts in auditory representations of computer programs were evaluated for program comprehension during debugging [32][31]. Studies with sighted programmers suggest promising improvements in accuracy of understanding when semantic information about the code is prioritised over syntactic information during presentation.

[3]See https://www.w3.org/WAI/wcag-curric/sam110-0.htm

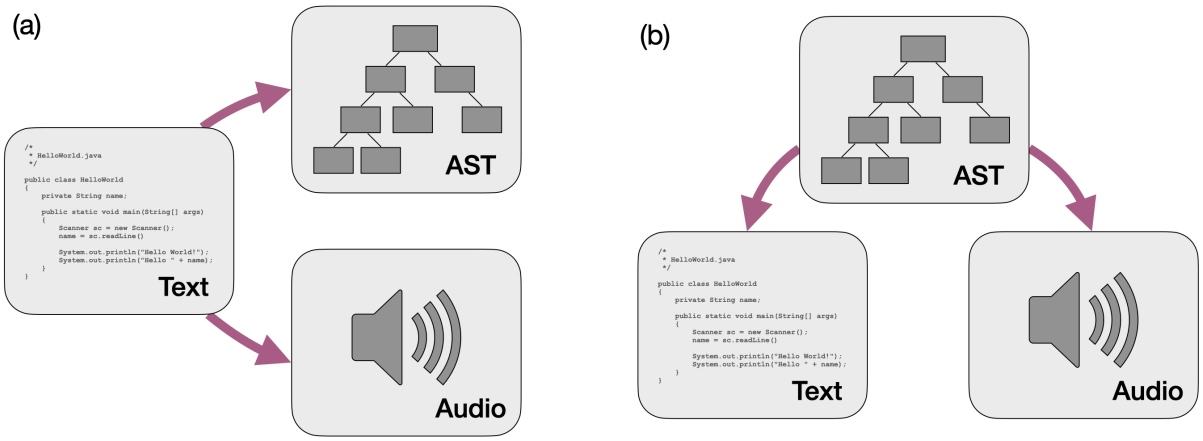


Figure 2: (a) The traditional model of programming: The user produces program text; the AST and audio are both derived from the program text. (b) A model for improved representation: The user manipulates the AST; both the textual and the audio representations are derived from the AST.

3.2 Invisible punctuation

There is also the issue of punctuation. Program text uses punctuation as vital tokens to assist the parsing of the code. For example, Python has colons at the end of many control statements. This is only to assist the parser, but (a) they have no semantic purpose and (b) punctuation is often not read out by screen readers. Tokens such as commas, periods, dashes, semi-colons and colons are read as pauses in speech, rather than explicitly read out. (Not to mention the issue of how they should be pronounced, which is a difficulty for novices [11] as well as screen readers [6].)

The Quorum programming language was specifically designed to alleviate this issue [33], by using words and layout for almost all syntax, instead of markers such as semi-colons and colons.

3.3 Indentation

The matter of invisible punctuation also applies to white-space, especially indentation. Indentation is either not read out by the screen reader or read out awkwardly, making it doubly problematic for screen-reader users. First, it is useless as a way to indicate program structure (either by convention as in Java, or as a strict meaning in Python) to a person reliant on a screen reader. Second, it makes it particularly awkward for a screen reader user to manage indentation and make it correct (again, either for convention in Java or as required in Python).

In the case where it is convention, tools such as "prettifiers" (auto-formatters) can solve this problem automatically, but (a) it seems odd to manage manually if it can be done automatically and (b) this is not possible in indentation-based languages like Python where the user must set the indent to determine the structure and indentation cannot be inferred (though auto-formatters can ensure *consistent* use of indentation). For indentation-based languages the program editor can help manage indentation, for example retaining the current level of indentation and increasing/decreasing indentation level intelligently when certain syntax (a colon) or semantic conventions (two or more carriage returns) are encountered, but even such intelligent editing is not always correct and must be managed by the programmer.

Albusays et al. [2] point out that when indentation is read out, it is done in a non-helpful way for programmers:

> When a screen reader user navigates through indentation based languages, [they] will hear [their] screen reader verbalizing whitespaces as a single space (e.g., "space, space, space") rather than a count ("three spaces")

Several programmers that Albusays et al. interviewed had independently customised their screen reader to produce the latter behaviour, while others used a Braille reader to help determine indentation levels.

3.4 Confusing values

An additional challenge of reading program text is that different program text may be narrated the same way. For example, different values can be read the same way by a screen reader, even when those values have different data types. In the following Java code[4]:

```
1 baa = 7;
2 bah = "7";
3 bar = "seven";
```

The values 7, "7", and "seven" will be narrated the same way — spoken as the word "seven" — by standard screen readers, and yet the values are syntactically and semantically different. An experienced programmer may be able to intuit the difference, at least enough to use character-by-character inspection to confirm their suspicion, but for novices this difference in presentation is likely to cause confusion.

Schanzer et al. [27] solve this problem by having their environment narrate the types of the non-number values and further disambiguates numerals in a string, as in `"7"`, and the written number, as in `"seven"`, by switching to a verbose mode and reading out the written form character-by-character. Building on the "semantic

[4]Based on a similar example given by Schanzer et al. [27].

prioritization" work of Stefik et al. [32][31], their system reads out the type *after* the value itself, for example for the value of `bah` the system would read "seven, a string". (This is also an example of front-loading, with the most semantically-distinguishing value first.)

3.5 Different modes of reading

In practice, sighted programmers employ a host of different reading strategies at different phases in their work. They may be skim-reading a program, focusing mainly on names of high-level structuring entities (such as classes or functions). They may be reading for program comprehension, trying to work out what a segment of code does or how it does it. Or they may be engaged in a debugging task, paying attention to every single character.

All these modes of reading are supported in various ways in the visual interfaces of modern editors. Syntax highlighting, using both font faces and colour, and layout conventions support skimming a program. Keywords are highlighted to support semantic reading of code. Yet all characters are available for inspection to support debugging.

Common screen readers struggle to support these different reading modes, instead providing a simplistic all-or-nothing linearisation of the text.

In a text-based program editor, a screen reader typically starts by reading the content linearly, starting on the first line (or at the current cursor position) and proceeding until the end, unless interrupted. Though there has been some research into improving skim-reading and navigation interfaces—through adapting the programming system to screen reader affordances [3] and through supplementing the screen reader with dedicated tactile hardware [7] or standard laptop trackpads [28]—this all-or-nothing linearisation remains the standard way that text content is narrated by screen readers.

In practice, many BLV programmers do not let their screen reader narrate in this way and instead rely on a combination of search functionality to locate key sections—such as class and function definitions and waypoint function names, such as main—and using detailed, line-by-line, narration of the program text.

3.6 Different code style preferences

The fundamental problem, broken down and described throughout this section, is emphasised by the often significantly differing code style preferences of BLV programmers compared to sighted programmers when working on text-based code, often requiring one visual-ability group of developers to compromise on their code style preferences in the name of collaboration. For example, Pandey et al. [22] found that, when working with Python, the formatting preferences of BLV programmers differ in 8 of 13 code styling practices from the styling practices favoured by sighted developers as enshrined in documentation[5] and with tooling[6] as the conventional practices for the Python programming language ecosystem.

[5]In PEP 8: https://peps.python.org/pep-0008/
[6]With Pycodestyle: https://pycodestyle.pycqa.org/en/latest/intro.html and also Black: https://black.readthedocs.io/en/stable/index.html

4 Audio should come from the AST, not text

Our suggested approach stems from the right-hand side of 2. The AST should be treated as the canonical representation (which is already true for editing paradigms such as block-based programming), and crucially, the audio presentation should be derived directly from the AST, without necessarily mapping to the visual presentation with 100% fidelity.

An AST-derived audio presentation need not present whitespace and punctuation, completely bypassing the "invisible punctuation" constraint of present day screen readers. When punctuation is meaningful in a text-based language, such as the dot operator when accessing members of a class or structure, it can be presented aurally in a more semantically meaningful way. For example, the `name` member of a `student` class could be presented as "name of student" or "student member name" instead of "student dot name".

For example, the `setWidth` method definition from the earlier code sample might be presented aurally as:

```
setWidth public method takes w, an int
width member assigned w
```

By deriving the audio presentation from the AST we avoid the problems of invisible punctuation and indentation, because these become aspects of the presentation rather than elements of the code's representation, while opening up a rich design space to explore and evaluate novel—auditory-first—approaches to the problems of linearisation, speech ordering, and confusing values.

All of the semantic information is presented in this example, yet it is not reading the program syntax. Different design decisions could be taken over exact wording, ordering and terminology, but the basic idea is: the speech should be derived from the AST, not from the textual representation of the AST.

4.1 Mixed visual-ability collaboration

While an audio presentation need not map 100% to the visual presentation, it is important that the two presentations are equivalent, for example sharing referents (such as line numbers, or similar), in order to facilitate collaboration amongst peers, and between pupil and teacher, and to foster a greater sense of solidarity and mutuality among peers of all visual-abilities.

We are not suggesting a complete break between the representations: obviously a visual and auditory representation of the same code will be similar. But our argument is that they do not need to be dogmatically identical if it interferes with understanding. Once familiar with one representation it would not be difficult to derive or understand the other, which would enable collaboration.

4.2 Accessible block-based editing

The initial versions of block-based editors were inaccessible for multiple different reasons. Ludi [14] pointed out ten years ago that the visual and mouse-centric design of block-based programming systems makes them largely inaccessible to BLV programmers. Milne and Ladner [18] further emphasised how block-based programming systems are inaccessible to screen reader users because the shapes and colours of the blocks are not typically spoken by the screen reader, and spatial relationships are verbose to describe and often meaningless to BLV programmers. Milne and Ladner's focus was

improving the accessibility of block-based environments on touchscreen devices—by developing touchscreen interactions to help identify blocks and their types, move blocks around, and convey program structure—however, many of the design guidelines in the paper are not specific to touchscreens or block-based editors and may be applicable to other accessible programming environments.

The challenges of making block-based editors accessible to screenreaders was two-fold: keyboard support was needed for all navigation and editor interactions, and it was necessary to find a way to represent the novel user interface paradigm to screen readers.

In order to enable the keyboard as a mouse alternative for block-based environments Mountapmbeme et al. [21] added a virtual cursor mode to Blockly, enabling users to navigate without changing the edit location. A corresponding screen reader module outputs speech to describe the focused element at the virtual cursor's location. Combined, these enhancements led to increased speed and accuracy of navigation and understanding tasks for BLV programmers.

Recently, Stefik et al. [30] created Quorum Blocks, with an accessible block-based editor (and impressive contributions beyond in the field of accessible graphics output which are beyond the scope of this paper). This paper produced a new block-based editor specifically designed to be accessible to the visually impaired via screen-readers, which took several ideas from previous research and also from frame-based editing which we will consider next.

4.3 Frame-based editing

Frame-based editing is designed to combine the strengths of block-based editing and text-based editing to provide improved error reduction and faster program manipulation in text-like programming languages[13]. We believe that frame-based editing provides a solid foundation upon which to build accessible program editors. The frame-based editing paradigm is text-like, enabling harmonious first-class integration with speech synthesis, and keyboard driven, meaning that manipulation interactions are more accessible for BLV users.

Because in frame-based editing the interface presents an AST-like representation of the program structure, not program text, a frame-based editor is better able to refine the presentation of program code for an auditory interface paradigm, even if those same refinements are not represented in the visual interface. Additionally, the "frame cursor" in the frame-based paradigm — used for insertion and manipulation of syntax nodes — provides a locus of focus for the user, somewhat analogous to the virtual cursor later added to Blockly [21], which helps users understand what type of manipulation actions are available, where their manipulation actions will manifest, and affords an opportunity to present the code differently than when using the text cursor to do detailed editing of program statements.

Furthermore, the frame-based paradigm supports a rich graphical interface which we believe can run in-tandem with a rich auditory first interface and enable equitable collaboration between BLV and sighted collaborators.

5 Conclusion

Screen readers read what is on the screen, but even when applications correctly permit this, the screen-based presentation may not suit the task at hand. For programming learners we believe that we can and should do better. Separating program structure from its representation—as in block-based, frame-based and structure editing—provides an opportunity to create first-class auditory interfaces with greater affordances for the ephemeral and linear nature of audio. Careful design consideration should be given to ordering and wording of the audio representation of code, rather than just reading out the textual syntax presentation.

Keyboard-based manipulation of the AST, without the need to type out all syntax, allows for a more accessible input. Keyboards are generally more accessible by users with all kinds of impairments than mice, which involve careful and precise aiming of the cursor that is not possible by many people. The keyboard entry in systems such as frame-based editors can also reduce the amount of syntax that must be memorised and the amount of typing required to translate intent to program code, which is both more accessible and potentially faster for learners.

In a frame-based editor, specifically, the user will not need to manage whitespace or punctuation and will need to use fewer keystrokes to insert code. In future work we will explore the suitability of frame-based editors as a foundation to build a "born-accessible" program editor with first-class audio presentation and keyboard driven interactions.

References

[1] Khaled Albusays and Stephanie Ludi. 2016. Eliciting programming challenges faced by developers with visual impairments: exploratory study. In *Proceedings of the 9th International Workshop on Cooperative and Human Aspects of Software Engineering*. ACM, 82–85. doi:10.1145/2897586.2897616

[2] Khaled Albusays, Stephanie Ludi, and Matt Huenerfauth. 2017. Interviews and Observation of Blind Software Developers at Work to Understand Code Navigation Challenges. In *Proceedings of the 19th International ACM SIGACCESS Conference on Computers and Accessibility*. 91–100. doi:10.1145/3132525.3132550

[3] Ameer Armaly, Paige Rodeghero, and Collin McMillan. 2018. AudioHighlight: Code Skimming for Blind Programmers. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 206–216. doi:10.1109/ICSME.2018.00030

[4] Suzanne P. Balik, Sean P. Mealin, Matthias F. Stallmann, Robert D. Rodman, Michelle L. Glatz, and Veronica J. Sigler. 2014. Including blind people in computing through access to graphs. In *Proceedings of the 16th International ACM SIGACCESS Conference on Computers & Accessibility* (Rochester, New York, USA) *(ASSETS '14)*. Association for Computing Machinery, New York, NY, USA, 91–98. doi:10.1145/2661334.2661364

[5] Brett A. Becker. 2021. What does saying that 'programming is hard' really say, and about whom? *Commun. ACM* 64, 8 (July 2021), 27–29. doi:10.1145/3469115

[6] A. Begel and S.L. Graham. 2005. Spoken programs. In *2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05)*. 99–106. doi:10.1109/VLHCC.2005.58

[7] Olutayo Falase, Alexa F. Siu, and Sean Follmer. 2019. Tactile Code Skimmer: A Tool to Help Blind Programmers Feel the Structure of Code. In *The 21st International ACM SIGACCESS Conference on Computers and Accessibility*. ACM, 536–538. doi:10.1145/3308561.3354616

[8] Chris Fleizach and Jeffrey P. Bigham. 2024. System-class Accessibility: The architectural support for making a whole system usable by people with disabilities. *Queue* 22, 5 (Nov. 2024), 28–39. doi:10.1145/3704627

[9] Mark Guzdial. 2010. Why is it so hard to learn to program. *Making Software: What Really Works, and Why We Believe It. O'Reilly Media* (2010), 111–124.

[10] Sayed Kamrul Hasan and Terje Gjøsæter. 2021. Screen reader accessibility study of interactive maps. In *International Conference on human-computer interaction*. Springer, 232–249.

[11] Felienne Hermans, Alaaeddin Swidan, and Efthimia Aivaloglou. 2018. Code phonology: an exploration into the vocalization of code. In *Proceedings of the 26th Conference on Program Comprehension* (Gothenburg, Sweden) *(ICPC '18)*. Association for Computing Machinery, New York, NY, USA, 308–311. doi:10.

1145/3196321.3196355
[12] Apple Inc. [n. d.]. *Apple Human Interface Guidelines*. https://web.archive.org/web/20250613135358/https://developer.apple.com/design/human-interface-guidelines/
[13] Michael Kölling, Neil CC Brown, and Amjad Altadmri. 2017. Frame-based editing. *Journal of Visual Languages and Sentient Systems* 3 (2017), 40–67.
[14] Stephanie Ludi. 2015. Position paper: Towards making block-based programming accessible for blind users. In *2015 IEEE Blocks and Beyond Workshop (Blocks and Beyond)*. 67–69. doi:10.1109/BLOCKS.2015.7369005
[15] Andrew Luxton-Reilly. 2016. Learning to Program is Easy. In *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education* (Arequipa, Peru) *(ITiCSE '16)*. Association for Computing Machinery, New York, NY, USA, 284–289. doi:10.1145/2899415.2899432
[16] Michele C. McDonnall, Anne Steverson, Rachael Sessler Trinkowsky, and Katerina Sergi and. 2024. Assistive technology use in the workplace by people with blindness and low vision: Perceived skill level, satisfaction, and challenges. *Assistive Technology* 36, 6 (2024), 429–436. doi:10.1080/10400435.2023.2213762 PMID: 37171786.
[17] Sean Mealin and Emerson Murphy-Hill. 2012. An exploratory study of blind software developers. In *2012 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 71–74. doi:10.1109/VLHCC.2012.6344485
[18] Lauren R. Milne and Richard E. Ladner. 2018. Blocks4All: Overcoming Accessibility Barriers to Blocks Programming for Children with Visual Impairments. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. 1–10. doi:10.1145/3173574.3173643
[19] Aboubakar Mountapmbeme and Stephanie Ludi. 2020. Investigating Challenges Faced by Learners with Visual Impairments using Block-Based Programming/Hybrid Environments. In *Proceedings of the 22nd International ACM SIGACCESS Conference on Computers and Accessibility* (Virtual Event, Greece) *(ASSETS '20)*. Association for Computing Machinery, New York, NY, USA, Article 73, 4 pages. doi:10.1145/3373625.3417998
[20] Aboubakar Mountapmbeme and Stephanie Ludi. 2021. How Teachers of the Visually Impaired Compensate with the Absence of Accessible Block-Based Languages. In *Proceedings of the 23rd International ACM SIGACCESS Conference on Computers and Accessibility* (Virtual Event, USA) *(ASSETS '21)*. Association for Computing Machinery, New York, NY, USA, Article 4, 10 pages. doi:10.1145/3441852.3471221
[21] Aboubakar Mountapmbeme, Obianuju Okafor, and Stephanie Ludi. 2022. Accessible Blockly: An Accessible Block-Based Programming Library for People with Visual Impairments. In *Proceedings of the 24th International ACM SIGACCESS Conference on Computers and Accessibility* (Athens, Greece) *(ASSETS '22)*. Association for Computing Machinery, New York, NY, USA, Article 19, 15 pages. doi:10.1145/3517428.3544806
[22] Maulishree Pandey, Steve Oney, and Andrew Begel. 2024. Towards Inclusive Source Code Readability Based on the Preferences of Programmers with Visual Impairments. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*. 1–18. doi:10.1145/3613904.3642512
[23] The GNOME Project. [n. d.]. *GNOME Human Interface Guidelines*. https://web.archive.org/web/20250613135249/https://developer.gnome.org/hig/
[24] TV Raman. 1996. Emacspeak—a speech interface. In *Proceedings of the SIGCHI conference on Human factors in computing systems*. 66–71.
[25] T. V. Raman. 1997. *Auditory User Interfaces*. Springer US. doi:10.1007/978-1-4615-6225-2
[26] Prakash Sankhi and Frode Eika Sandnes. 2022. A glimpse into smartphone screen reader use among blind teenagers in rural Nepal. *Disability and Rehabilitation: Assistive Technology* 17, 8 (2022), 875–881.
[27] Emmanuel Schanzer, Sina Bahram, and Shriram Krishnamurthi. 2020. Adapting Student IDEs for Blind Programmers. In *Proceedings of the 20th Koli Calling International Conference on Computing Education Research* (Koli, Finland) *(Koli Calling '20)*. Association for Computing Machinery, New York, NY, USA, Article 23, 5 pages. doi:10.1145/3428029.3428051
[28] Ather Sharif, Venkatesh Potluri, Jazz R. X. Ang, Jacob O. Wobbrock, and Jennifer Mankoff. 2024. Touchpad Mapper: Exploring Non-Visual Touchpad Interactions for Screen-Reader Users. In *Proceedings of the 21st International Web for All Conference* (Singapore, Singapore) *(W4A '24)*. Association for Computing Machinery, New York, NY, USA, 42–44. doi:10.1145/3677846.3677867
[29] Kristen Shinohara and Jacob O. Wobbrock. 2016. Self-Conscious or Self-Confident? A Diary Study Conceptualizing the Social Accessibility of Assistive Technology. *ACM Trans. Access. Comput.* 8, 2, Article 5 (Jan. 2016), 31 pages. doi:10.1145/2827857
[30] Andreas Stefik, Willliam Allee, Gabriel Contreras, Timothy Kluthe, Alex Hoffman, Brianna Blaser, and Richard Ladner. 2024. Accessible to Whom? Bringing Accessibility to Blocks. In *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1* (Portland, OR, USA) *(SIGCSE 2024)*. Association for Computing Machinery, New York, NY, USA, 1286–1292. doi:10.1145/3626252.3630770
[31] Andreas Stefik and Ed Gellenbeck. 2009. Using spoken text to aid debugging: An empirical study. In *2009 IEEE 17th International Conference on Program Comprehension*. 110–119. doi:10.1109/ICPC.2009.5090034
[32] Andreas Stefik, Andrew Haywood, Shahzada Mansoor, Brock Dunda, and Daniel Garcia. 2009. SODBeans. In *2009 IEEE 17th International Conference on Program Comprehension*. 293–294. doi:10.1109/ICPC.2009.5090064
[33] Andreas Stefik and Susanna Siebert. 2013. An empirical investigation into programming language syntax. *ACM Transactions on Computing Education (TOCE)* 13, 4 (2013), 1–40.
[34] Dilawar Shah Zwakman, Debajyoti Pal, and Chonlameth Arpnikanondt. 2021. Usability evaluation of artificial intelligence-based voice assistants: The case of Amazon Alexa. *SN Computer Science* 2, 1 (2021), 28.