Grid-Coding: An Accessible, Efficient, and Structured Coding Paradigm for Blind and Low-Vision Programmers

Md Ehtesham-Ul-Haque
Pennsylvania State University
University Park, Pennsylvania, USA
mfe5232@psu.edu

Syed Mostofa Monsur
Bangladesh University of Engineering and Technology
Dhaka, Bangladesh
0419052034@grad.cse.buet.ac.bd

Syed Masum Billah
Pennsylvania State University
University Park, Pennsylvania, USA
sbillah@psu.edu

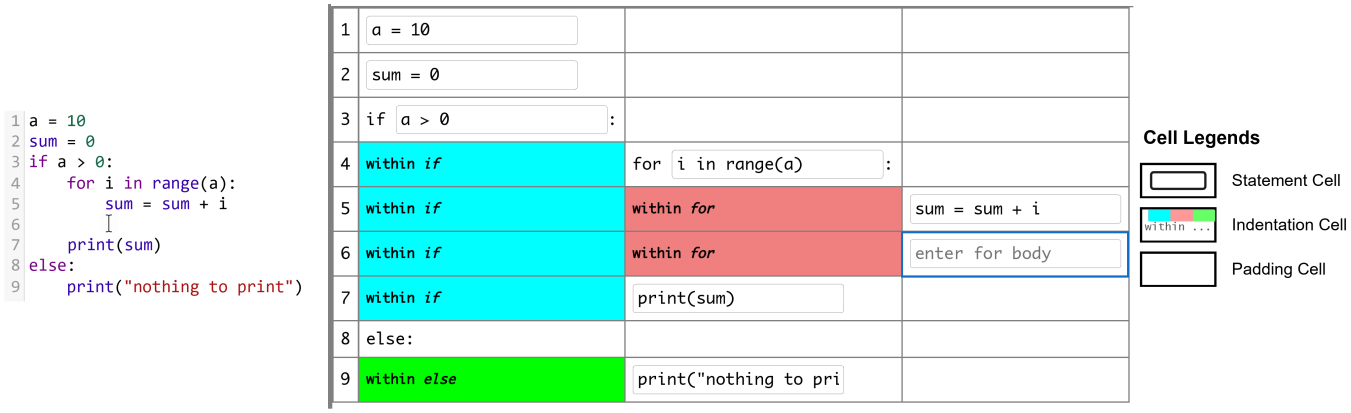


Figure 1: Code reading and writing in a traditional code editor vs. GRID-CODING**. (Left) shows a piece of Python code (9 lines) in a popular IDE, Visual Studio Code (VS Code). Note that VS Code highlights keywords and function names with colors and represents indentations with spaces. Unfortunately, these visual cues do not benefit blind programmers. (Right) shows an accessible representation of the same piece of code in *Grid Editor*, our implementation of Grid-Coding. Each grid component (e.g., rows, columns, and cells) has consistent semantics. For example, a row represents a single line; a column represents a scope (or an indentation level), except for the leftmost column, which represents line numbers; and a cell can represent either a statement, an indentation, or a padding. Note that indentation cells convey semantically meaningful info (e.g., *within if* instead of *space, space, . . . , space*), which helps blind programmers to stay situated at any cell. In addition, blind programmers can easily navigate the grid structure using arrow keys, edit a cell by pressing ENTER if allowed (e.g., at row 6, the rightmost column), and go back to navigating the grid in read-only mode by pressing ESC. Bright cell colors provide visual aids to sighted and low-vision programmers.**

ABSTRACT

Sighted programmers often rely on visual cues (e.g., syntax coloring, keyword highlighting, code formatting) to perform common coding activities in text-based languages (e.g., Python). Unfortunately, blind and low-vision (BLV) programmers hardly benefit from these visual cues because they interact with computers via assistive technologies (e.g., screen readers), which fail to communicate visual semantics meaningfully. Prior work on making text-based programming languages and environments accessible mostly focused on code navigation and, to some extent, code debugging, but not much toward code editing, which is an essential coding activity.

We present Grid-Coding to fill this gap. Grid-Coding renders source code in a structured 2D grid, where each row, column, and cell have consistent, meaningful semantics. Its design is grounded on prior work and refined by 28 BLV programmers through online participatory sessions for 2 months. We implemented the Grid-Coding prototype as a spreadsheet-like web application for Python and evaluated it with a study with 12 BLV programmers. This study revealed that, compared to a text editor (i.e., the go-to editor for BLV programmers), our prototype enabled BLV programmers to navigate source code quickly, find the context of a statement easily, detect syntax errors in existing code effectively, and write new code with fewer syntax errors. The study also revealed how BLV programmers adopted Grid-Coding and demonstrated novel interaction patterns conducive to increased programming productivity.

UIST '22, October 29-November 2, 2022, Bend, OR, USA

ACM ISBN 978-1-4503-9320-1/22/10…$15.00
https://doi.org/10.1145/3526113.3545620

CCS CONCEPTS

• **Human-centered computing** → **Accessibility systems and tools**.

KEYWORDS

Accessibility, assistive technology, screen readers; programming languages, text-based programming languages, Python, code reading, code writing; grid-coding; blind and low-vision, programmers.

ACM Reference Format:
Md Ehtesham-Ul-Haque, Syed Mostofa Monsur, and Syed Masum Billah. 2022. Grid-Coding: An Accessible, Efficient, and Structured Coding Paradigm for Blind and Low-Vision Programmers. In *The 35th Annual ACM Symposium on User Interface Software and Technology (UIST '22), October 29-November 2, 2022, Bend, OR, USA.* ACM, New York, NY, USA, 21 pages. https://doi.org/10.1145/3526113.3545620

1 INTRODUCTION

Many aspects of text-based programming are visual. For example, sighted programmers can visually inspect the matching parentheses or the alignment of spaces in a code snippet. For additional support, they often resort to an integrated programming environment (IDE), such as Visual Studio Code, IntelliJ, and Eclipse, which contain advanced code editors that offer numerous visual and syntactical cues (e.g., keyword highlighting and syntax coloring, bracket-matching, and realtime error indication with squiggly lines). Unfortunately, these visual cues are not readily available to blind and low-vision (BLV) programmers who must use assistive technologies like screen readers (e.g., NVDA [3]). This is because a screen reader can only narrate textual descriptions of content given by an app or an IDE, and for most visual cues, IDEs do not provide meaningful descriptions [15, 38, 49, 52, 62]. Thus, most BLV programmers do not benefit from modern IDEs; instead, they are forced to use *plain text editors* (e.g., Notepad, TextEdit, Notepad++) [15, 45], which affects their coding activities, such as code *navigation*, *comprehension*, *skimming*, *debugging*, and *editing* [15, 34, 38, 45, 49, 62].

Out of all challenges, code navigation challenges are extensively studied. Other challenges, particularly code comprehension and skimming, and to some extent, debugging [35], are considered as a consequence of code navigation challenges [49]. Put differently, code navigation and associated challenges stem from the inherent difficulty in *reading* the code, whereas challenges in code editing involve *writing*, which are more consequential. BLV programmers, for instance, often write code at an incorrect or an unintended location [23, 38, 44, 45], inadvertently introducing syntax errors and causing frustrations thereof [17]. Although consequential, prior work hardly addresses the code editing challenges for them.

In this paper, we introduce **Grid-Coding** paradigm, a fresh perspective on non-visual programming that addresses the code navigation, editing, and associated challenges with text-based programming languages for BLV programmers. To demonstrate the novelty in Grid-Coding, we briefly elaborate on the challenges BLV programmers face in code navigation and editing. Both challenges stem from the grammar of a programming language, limited support from plain text editors, and screen readers reading code sequentially, line by line, sometimes character by character [14–16, 45].

First, BLV programmers listen to the entire source code repeatedly to create a mental map of the code hierarchy, which can burden their short-term memory [21, 55]. **Second**, they frequently move around their screen reader cursor in a lengthy codebase, often losing track of the last known location of their cursor and struggling to comprehend the overall code structure [14, 15, 45]. **Third**, in programming languages like Python, they need to mentally count and listen to whitespace characters (e.g., `NEWLINE`, `SPACE`, `TAB`) to determine the current scope or level in the code [35]. For example, in Figure 1, line 5 on the left side of the code will be read out as ''`Space space space ... (8 times), sum equals sum plus i`''. This process is slow, tedious, and error-prone [15, 45, 52]. **Fourth**, because of the uncertainty in determining the scope and the current cursor focus, they are prone to editing code at an incorrect location.

Grid-Coding overcomes the above challenges as follows. It flattens the implicit hierarchy in the code into a 2D grid and renders the grid in a spreadsheet-like structure with a finite number of rows and columns (shown on the right side of Figure 1). Each row in the grid represents a single line in the original code snippet, and each column represents an indentation level, except for the first column, which represents line numbers. Each cell contains either a statement, an indentation, or padding. `Indentation cells` carry semantically meaningful information about an indentation (e.g., `within for` if the current statement is within a *for* loop, instead of reading out as ''`Space space space ...`"). `Padding cells` are non-editable and always appear on the right of a statement cell to preserve the uniform grid structure, i.e., ensure the number of columns equals the highest level in any rows. A programmer can navigate the grid using common spreadsheet shortcuts, such as directional `Arrows` to move to a neighboring cell and `Ctrl + Arrows` to jump over `Padding cells` or `Indentation cells` in a row or column.

This 2D, spreadsheet-like structure eliminates the challenges of navigating hierarchies—one can easily navigate grid cells with arrow keys and remain positioned in the grid by listening to the row and column numbers. The maximum number of rows and columns inform programmers of source codes' inherent complexity (e.g., lines of code, the maximum number of nested scopes), which can help them create a mental map easily [42, 66]. Furthermore, `Indentation cells` provide the context of each statement at all times, thus eliminating the confusion about indentation and scopes, as well as the need to memorize the scope or level of a statement. This can also lessen the burden on programmers' short-term memory and increase their confidence in locating the screen reader cursor in a larger codebase. Moreover, since the grid layout is well-defined (e.g., $m \times n$), it ensures uniformity of interaction experience, a cornerstone for effective non-visual interaction [25].

Grid-Coding utilizes spearcons and earcons to communicate important contextual information, such as reaching the boundary in the grid, reporting syntax errors, and distinguishing `Indentation cells` from `Padding cells` [36]. It also uses bright cell colors to provide a visual aid to low-vision programmers. In all, the dynamic grid structure, a small set of simple shortcuts, consistent semantics, and meaningful auditory cues of Grid-Coding address the first three challenges involving coding reading.

To address the fourth challenge involving risk-free code writing, Grid-Coding makes all grid cells read-only by default. One can press `Enter` to edit a `Statement cell` and `Esc` to go back to navigating the grid in read-only mode. While editing, Grid-Coding disables breaking a statement into multiple lines, which is consistent with BLV programmers' current practice of reading code line by line or character by character. Furthermore, Grid-Coding only allows editing an `Indentation cell` if doing so does not cause a syntax error. For example, the `within for Indentation cell` in row 6 in Figure 1.right is editable but the `within for` cell in row 5 is not. By incorporating these built-in safety mechanisms, Grid-Coding avoids introducing common syntax errors that BLV programmers

often make with a plain text editor, where editing is unconstrained, i.e., one can edit at any location.

We implemented Grid-Coding paradigm by developing an online code editor, namely, **Grid Editor**, from scratch (§3). Grid Editor leverages Abstract Syntax Tree (AST) of a code, similar to prior work [21, 54]. However, unlike prior work, it augments the AST to create Abstract Syntax Table (ASTab). This ASTab (§4.3) maintains the proposed grid structure and allows programmers to edit a `Statement cell` or create a new one (i.e., mutate the AST).

Leveraging AST has notable benefits: it allows Grid-Coding to generalize for any programming language because open-source packages to generate AST for most languages are available [7]. We demonstrated Grid-Coding for `Python` because it is the most popular language to date [9] and is used to develop a popular screen reader, NVDA [50], plus its plug-ins [47], thus has of significant importance to BLV community. We demonstrated how Grid-Coding can be implemented for another language (e.g., Java) in Section 8.

We adopted an online participatory design with 28 BLV programmers for 2 months to refine Grid-Coding (§3.1). During development, we followed the ten implementation guidelines by Philip Guo [33] to scale and sustain our prototype[1] in the longer run. A user study with 12 BLV programmers reveals that, compared to a plain text editor (i.e., the go-to editor for most BLV programmers), Grid Editor enables participants to quickly navigate source code, find the context hierarchy of a statement, understand the structure, and detect and fix syntax errors. In addition, they can comfortably edit source code in the grid cells, zeroing in on indentation errors and minimizing the number of non-indentation errors in the code. The study also revealed how BLV programmers adopted Grid-Coding and demonstrated novel interaction patterns conducive to increased programming productivity.

In sum, our contribution is three-fold:

- Proposing Grid-Coding: a new paradigm for non-visual programming on grid structure instead of text editors.
- Implementing Grid Editor, a web-based, spreadsheet-like editor to read and write Python code that augments AST.
- Evaluating Grid Editor with 12 BLV programmers and reporting unique usage patterns emerged in this new paradigm.

2 BACKGROUND AND RELATED WORK

The common accessibility issues for BLV programmers can be categorized as *assistive technology-related issues*, *programming environments-related issues*, and *programming language-related issues*. We briefly describe these issues and current workarounds and how these issues informed the design of Grid-Coding. We also describe Grid-Coding' relationships with block coding, tabular data representation, and AST, as well as the use of auditory cues in non-visual programming.

2.1 Assistive Technology-Related Issues

Assistive technologies, such as screen readers, narrate the contents of the screen via audio. A screen reader (e.g., NVDA [3], JAWS [1], and VoiceOver [19]) compensates for the inability of users to see a graphical user interface, and implements keyboard shortcuts for users as an alternative to using a mouse to point-and-click graphical elements. Due to the transient nature of auditory perception, BLV programmers usually listen to code line-by-line, from the start-to-the end, multiple times. These make it hard to navigate a large codebase and comprehend the context of a line (e.g., current scope) [15, 21, 45, 52]. Screen readers also read out source code in a way that is hard to understand because some keywords and symbols in the source code may not be dictionary words.

Workarounds. Instead of going line-by-line, BLV programmers often use the search feature to look up information. For effective searching, they put private comments and keywords in the code. However, before sending this code to others, they must remove all of these comments, which is inconvenient [15, 45].

Another workaround is to force the screen readers to read out the gist of a statement. For example, Stefik et al. [59, 62] recommended describing source code as naturally as possible (e.g., "m = 5;" can be read out as "m equals 5" or "m to 5"). Baker et al. [21] presented the name, followed by the input, return type, and modifiers or annotations to read out functions; and the name, followed by what it extends and implements, followed by the modifiers for class declarations. Schanzer et al. [54] also included a descriptive label of the function with input arguments. Potluri et al. [52] provided a list view of available functions, similar to virtual link lists offered by screen readers on web pages [37]. They additionally provided a code summary view containing the variables and functions inside a class along with their line numbers. We experimented with different textual representations of code statements during our participatory design phase (§3.1).

2.2 Programming Environments-Related Issues

Programming Environments or IDEs, such as Visual Studio Code, NetBeans, and IntelliJ, pack myriad features (e.g., auto-completion, auto-indentation, code-folding, error alerts, and visual debugging) and visual cues (e.g., syntax coloring, keyword highlighting, parenthesis matching, auto-indentation, and red squiggles). Sighted programmers conveniently use these features and visual cues to navigate, comprehend, skim, debug, and edit code. Unfortunately, most BLV programmers are unaware of these features or rarely use complex IDE features [45, 52]. Worse, many of these features are not exposed to screen readers [15, 45, 52]. For example, auto-completion dialog, code-folding, and debugging panels are not fully accessible to screen readers [15, 38, 49, 52, 55, 62] Similarly, screen readers do not read out red squiggles in Visual Studio that alert sighted users about a syntax error. This issue can easily be addressed by playing an error tone in the presence of a syntax error [52], which we incorporated into our design.

Workarounds. Common workarounds include (a) using plain text editors (e.g., Notepad, Notepad++) with basic features [15, 45] that are more accessible with screen readers and (b) searching for external plug-ins that can make an IDE more accessible [47]. For example, Notepad++ supports plug-ins that read out the number of indentation characters to assist BLV programmers during code editing. Sometimes, BLV programmers use plain text editors as a secondary buffer to record errors, bugs, and variable locations for

[1] https://ally-ide.herokuapp.com/

easier navigation and bookmarking [15]. This strategy also allows them to look up information without losing their location in the code and enables out-of-context editing [14, 15, 45]. It inspires us to design multiple coordinated views: a Text Editor and a Grid Editor side-by-side, where one can seamlessly transition between them.

2.3 Programming Language-Related Issues

Text-based programming languages are governed by grammars, which include a set of textual rules to describe all possible strings (e.g., variables, statements, scopes, conditions, loops, delimiters) in a specific programming language [13, pp. 42–52]. To adhere to the underlying grammar, programmers often need visual perception. For example, programming languages like C, Lisp, and Java, require matching parenthesis to create a scope, whereas some new languages, like Python, Ruby, or Quorum [60], require `SPACE` or `TAB`, as shown on the left side of Figure 1. In both cases, sighted programmers can visually inspect the matching parentheses or the alignment of spaces. For additional support, they often resort to an integrated programming environment (IDE).

Some programming languages have complicated syntax (e.g., Java) [61], and some rely on visual cues (e.g., space for indentation in Python). These languages pose a learning challenge for BLV programmers. Further, in most programming languages, source code can contain implicit hierarchies (e.g., global scope, nested scopes) and recursion. These can make it challenging for BLV programmers to navigate code, as they can easily lose track of which level they were at [15, 52]. Besides navigation, Albusays et al. [15] reported that blind developers constantly struggle to contextualize the code. Other exploratory studies [40, 45] reported similar findings.

Workarounds. Researchers have proposed specialized, BLV-friendly programming languages and tools. For example, Sanchez et al. [63] developed an Audio Programming Language (APL). However, their language supports only a limited set of functionalities, hence not scalable. Similarly, Stefik et al. [58, 62] developed a programming language, Quorum [60], for BLV students, and an audio-based programming environment, Sodbeans [58]. Sodbeans provides a sonified omniscient debugger (SOD) that integrates audio cues to help BLV users debug their code. In our design, we parse the compiler error output (e.g., error line number, error description) and present it in a consistent format that allows the programmers to find the relevant error information and jump to the originating line.

A second workaround is explicitly constructing the code hierarchy using the AST and presenting it to BLV programmers as a tree [14, 21]. For example, Smith et.al. [55] developed an Eclipse plugin to navigate hierarchical structures of the files in a codebase. Drawing on this work, Baker et al. [21] presented StructJumper, an Eclipse plugin, to create a hierarchical tree of the code inside a Java class. Schanzer et al. [54] promoted hierarchical code navigation in a browser-based environment that leverages AST to support multi-language. However, this workaround is for navigating code where the tree remains read-only. Our goal in this paper is to make reading and writing of mainstream programming languages easier for BLV programmers.

2.4 Relationship with Block-Based Coding

Block-based programming languages (e.g., Scratch [10], Blockly [8], and Snap! [11]) provide a set of predefined visual blocks that one can connect by dragging and dropping to create code. Block languages use the puzzle-piece metaphor to indicate the compatibility of blocks and ensure that the generated code is always syntactically correct [65]. Thus, block languages are suitable for children because they do not have to worry about syntax errors while creating code with blocks [22].

However, the reliance on colors and shapes of visual blocks and the drag-and-drop interaction make block-based programming inaccessible for BLV individuals [46, 65]. Nonetheless, we were inspired by the error-prevention mechanism in block programming and supported auto-completion of block-like code templates (e.g., loops and if-else conditions) to lower the likelihood of common syntax errors.

2.5 Audio Cues for Accessible Programming

Auditory cues are an important construct in non-visual programming. Stefik et al. [59] suggested that audio cues should be short; browsable, i.e., one can extract key info while skimming; and deliver important information first [62]. Other work [21, 54] also found these principles useful. Among different types of audio cues, speech-based, spearcons, earcons, and audio icons are commonly used. Researchers have found that spearcons and earcons are effective in hierarchical menu navigations [27, 64]; and speech-based cues and earcons are effective in understanding compiler errors, invalid statements, cursor location, background process, value changes of debug variables, and flow of program execution while debugging [15, 57, 59]. Albusays et al. reported that BLV programmers think of auditory feedback as a core component of programming interaction [15]. Memorizing multiple audio cues can be a problem. In this regard, Schanzer et al. [54] suggested playing earcons and speech-based cues together so that users can associate earcons with speech over time but are never forced to remember them. To increase the information content in auditory output, researchers have suggested playing a secondary audio channel distinct from the screen reader speech at the same time [52, 53]. Drawing on these large bodies of work, we chose to use speech-based (e.g., current row number), familiar (e.g., error beep for syntax error representation), and semantically meaningful (e.g., a typewriter sound while editing code) audio cues in our implementation.

2.6 Tabular Representation of Source Code

There is growing evidence that BLV users find tabular structure (i.e., spreadsheets, grids) more useful than raw data (e.g., text files, web pages); they can make sense of the information and seek data faster in tables because the tabular format is organized, navigable by directional arrow keys, and the interaction is deterministic and reversible [18, 20, 28, 32, 39, 41, 42, 48, 51, 56, 66, 67]. Further, prior research [24–26] has shown that BLV users prefer a small set of shortcuts (e.g., arrow keys), and uniformity in interaction experience. A grid or tabular structure thus can fulfill these requirements and is the key to our design.

2.7 Uniform Abstract Syntax Tree (AST)

An AST represents the syntax of a source code in a hierarchical (i.e., tree) structure, as defined by the grammar of a programming language. Generating an AST from a source code indicates whether the code is syntactically correct or not. Schanzer et al. [54] suggested that IDEs should support multiple languages rather than be tied to one. They recommended constructing a language agnostic, *uniform AST* to represent any source code. This is particularly appealing because converting a language-specific AST to a uniform AST is doable. Our system works with the AST of Python code; it first converts the tree representation of AST into a 2D tabular structure and represents it in the grid.

3 DESIGN AND OVERVIEW OF GRID-CODING

First, we present an initial prototype of Grid-Coding that was used as a probe in our participatory design. We then provide an overview of our final design and highlight how the participatory design refined our final prototype. A list of supported shortcuts and audio cues are presented in Table 1 and Table 2.

3.1 Participatory Design

Initial prototype. We chose to develop our programming environment as a web-based system because (a) web-based platforms are getting popular due to the increasing availability of Chromebooks [54]; (b) users can use it without additional setup [33]; and (c) we can distribute its URL to online forums for early evaluations.

We analyzed relevant prior work on accessibility issues in mainstream, text-based programming languages and IDEs. As described in Section 2, we chose to represent the AST of Python in a grid-based structure [42, 66]. Following the ARIA guidelines [6], the first row of this grid contained column headers (e.g., line number, level 1, level 2), and the first column contained line numbers. The subsequent rows represented a line, and the columns represented a level (or scope) of `Python` code. The number of columns increased as the programmers added nested levels. We made the structure dynamic with `Padding cells` to provide a uniform interaction experience [25]. To support easier context understanding [15, 45], we replaced whitespaces with `Indentation cells` containing contextual cues (e.g., `within if`). The prototype supported writing block `Statement cells` with accessible auto-completion. Users could use `Tab` and `Shift + Tab` to change indentation levels. The prototype provided error beeps as soon as a syntax error occurs (similar to red squiggles), as suggested by [52, 59]. Another periodic error cue was available announcing the error line at a fixed interval to keep users informed about errors in the code [15, 59]. The prototype also provided meaningful earcons during each interaction with the system using a secondary audio channel [52] over the screen readers' audio [53]. It contained a plain text editor on the left and a grid editor on the right. Both editors were highly coordinated, i.e., the text cursor in one view was mapped with the corresponding location in the other.

Initial prototype as a probe. After IRB approval, we reached out to two online groups for BLV users (`nvda@nvda.groups.io` and `program-l@freelists.org`), inviting them to evaluate our initial prototype. Within a week, we shared the URL of this prototype with 28

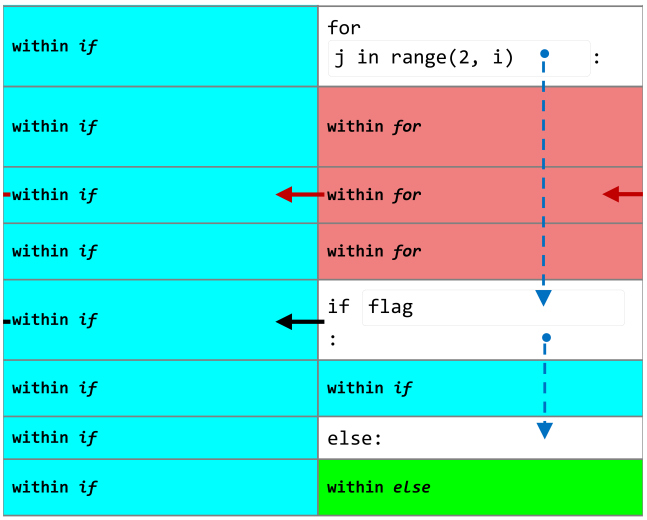


Figure 2: Navigating in Grid Editor. Black and Red Left arrows illustrate using `LEFT Arrow` to go to the adjacent cell on the left of a cell (e.g., from 'if flag:' to 'within if'). Blue arrows depict the navigation of all the statements within a level and skipping the `Indentation cells` with `Ctrl + DOWN Arrow` (e.g., from 'for j in range (2, i):' to 'if flag:').

Shortcut	Description
→/←/↑/↓	Go to the adjacent cell in the direction of the arrow key if available; otherwise, play boundary notification
`Ctrl` + →/←/↑/↓	Jump to the `Statement cell` or the cell at the edge of the row/column in the direction of the arrow key skipping `Indentation cell` and `Padding cell` if available; otherwise, play boundary notification
`Enter`	Make the current cell editable from `Navigation Mode` (if allowed) or create a new row if the cell is already in `Edit Mode`
`Esc`	Make the current `Statement cell` read-only from `Edit Mode` and go to `Navigation Mode`
`Alt + Enter`	Execute the source code and go to Code Output
`Crtl + G`	Go to a line in Grid Editor
`Crtl + L`	Announce the line number, level, and scope of the current `Statement cell`

Table 1: List of shortcuts for Grid Editor.

Notification	Meaning of auditory cue
Boundary	A cell is in the top/bottom/left/right boundary and trying to go beyond
Error	A beep alert if cell contains a syntax error
Compiling	The code is being compiled
Enter Edit Mode	Pressed `Enter` on a `Statement` cell in `Navigation Mode`
Exit Edit Mode	Pressed `Esc` in `Edit Mode`
Editing	A typewriter sound in `Edit Mode`
Cell Navigation	Traversing the grid with arrow keys
Block Auto-Completion	Using suggestions for auto-completion of block statements

Table 2: List of auditory cues for Grid Editor.

BLV programmers who responded to our invitations. These participants used our prototype in their daily coding activities (e.g., doing homework, copying and pasting 3rd-party code to check the overall structure, and for fun). From time to time, they provided feedback, reported bugs, and requested new features via emails. Upon receiving feedback, a bug report, or a feature request, two authors of this paper discussed how to resolve the issue gracefully and quickly deployed a solution. This cycle continued for 2 months until the prototype became stable and received encouraging feedback from the participants. Below, we describe how various functionalities of Grid-Coding evolved from its initial version to the prototype presented in this paper (shown in Figure 3).

3.2 Code Presentation

Grid Editor represents a source code in a 2D grid or tabular format, which localizes the programming area by dividing it into cells (see Figure 3.Right). Each row in the grid represents a line of code, and each column represents a level or scope. The structure is dynamic and does not contain any redundant rows or columns.

The first column of the grid contains the line numbers. The subsequent columns are part of the code. For example, The second column contains the first level of code, the third column contains the second level, and so on. Our initial design had a header row. However, we had to remove it because most participants reported that it creates a mismatch between the current row number and line number in the code. For example, screen readers consider the header row as *row 1*. So, if the current line number is 5, the screen reader would say row 6 (line number + 1). It was confusing, and participants suggested removing the header row.

Grid Editor has three types of cells: `Statement cell`, `Indentation cell`, and `Padding cell`. `Statement cells` are editable that contain the code that users write in each row. `Indentation cells` replace whitespaces and convey semantically meaningful information (e.g., `within if`) about block statements. Finally, `Padding cells` maintain a uniform grid structure. Each row in the grid can contain only a single `Statement cell` and multiple `Indentation cells` and `Padding cells`. `Indentation cells` always appear before the `Statement cell` in a row, and `Padding cells` always follow the `Statement cell`.

3.2.1 Modes of Operation. Grid Editor operates in two modes: `Navigation Mode` and `Edit Mode`. `Navigation Mode` is the non-editable state of the grid for risk-free navigation of source code. In `Edit Mode`, users can write source code in the `Statement cells` of the grid. Users can go from `Navigation Mode` to `Edit Mode` by pressing `Enter` on a valid cell and go back to the `Navigation Mode` with `Esc`. Participants recommended including familiar and distinctive audio cues to communicate states. Therefore, Grid Editor conveys a low-to-high note when entering from `Navigation Mode` to `Edit Mode`, a typewriter sound during `Edit Mode`; a high-to-low note when exiting `Edit Mode`.

3.3 Code Navigation

3.3.1 Navigating Source Code. Grid Editor provides risk-free navigation of source code in `Navigation Mode`, protecting against accidental modification. Users can use the `Arrow` keys to go from one cell to another adjacent cell in the grid (see Black and Red arrows in Figure 2), including line numbers. Grid Editor provides audio feedback during each `Arrow` key press, including a boundary indicator sound when trying to go out from a cell at the grid boundary.

In addition to cell-by-cell navigation, users can jump over the `Padding cells` and `Indentation cells` with `Ctrl + UP/DOWN Arrow` to navigate the code block-by-block (see broken Blue arrows in Figure 2). Similarly, `Ctrl + LEFT/RIGHT Arrow` allows users to skip `Padding cells` or `Indentation cells` in a row. The feedback from the participatory sessions informed the design of these shortcuts to skip cells. Users can invoke the `Go To` option with `Ctrl + G` shortcut, type the row number, and press `Enter` to quickly jump to a particular row.

3.3.2 Finding Contexts. In Grid Editor, users can get information about the line, level, and scope of the code (e.g., *at line 8, level 2, within if scope*) with the `Ctrl + L` shortcut. To understand the complete nesting structure of a statement, users can check all the `Indentation cells` in that row and listen to contextual information such as `within if` and `within for` (see Figure 4). This informs the users about all the block statements written at each level without leaving the row and searching for other nested statements. Later, some participants pointed out that including the actual conditions in the `Indentation cells` (e.g., `within if a > 0`) can help understand the context hierarchy of a deeply nested statement. Therefore, we included the option to enable conditions but disabled it by default as some participants found it too verbose, particularly in a small source code.

3.4 Code Editing

3.4.1 Writing Statements. Users can edit in Grid Editor by finding a `Statement cell` in `Navigation Mode` and pressing `Enter` to go to `Edit Mode`. To remind users that they are in `Edit Mode`, Grid Editor periodically provides a typewriter sound cue. Without this cue in the initial design, participants reported being unsure of the mode of operation. Each editable cell contains a placeholder depending on the position in that grid to provide contextual cues. For example, a `Statement cell` at level 1 announces *'enter a statement'* whereas a `Statement cell` inside a `for block` announces *'enter for body'*.

In `Edit Mode`, users can type a statement such as *a = 5* inside a `Statement cell` and press `Enter` to create a new row just below the current row. The new row will contain a `Statement cell` at the same level unless users create a block statement (e.g., conditions, loops). In that case, an `Indentation cell` will be created first, and the `Statement cell` will be at the next level (block statements are described in §3.4.3). Users will stay in `Edit Mode` until they press `Esc` to go to `Navigation Mode`.

3.4.2 Suggestions for Variables and Blocks. Grid Editor provides a list of suggestions in realtime based on the user input using an accessible drop-down menu. Users can traverse the menu with `UP/DOWN Arrow` and auto-complete variables (e.g., *value*, *sumOfNumbers*) and block statements (e.g., `if block`, `try block`) by pressing `Enter`. The suggestions for auto-completion are dynamically generated to ensure the creation of valid block statements in the code (e.g., an `else block` is not suggested without an immediate `if block`).

3.4.3 Writing Block Statements. Grid Editor supports the completion of all block statements in `Python`. Consider writing an `if block`

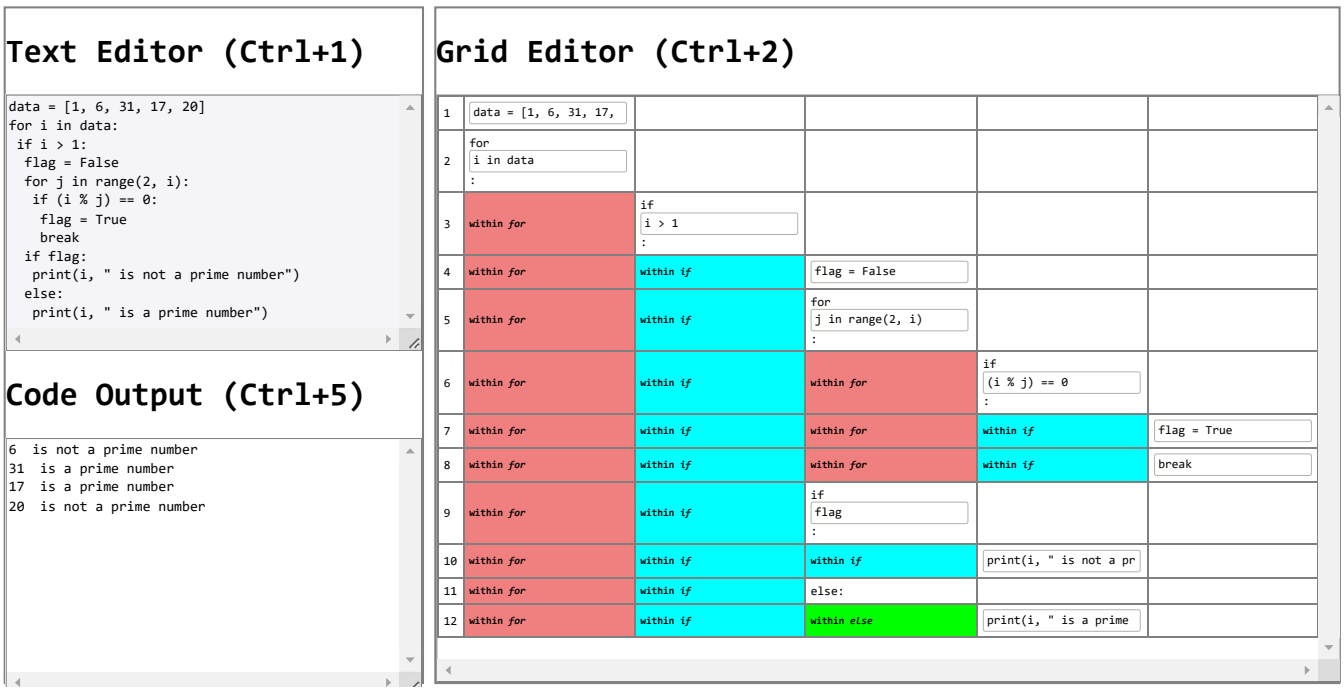


Figure 3: A screenshot of our final prototype. (Upper Left) a general-purpose Text Editor shows the textual representation of a code. (Right) Grid Editor showing the grid representation of the code. (Lower Left) Code Output shows the output of the code. Note that Text Editor and Grid Editor are coordinated.

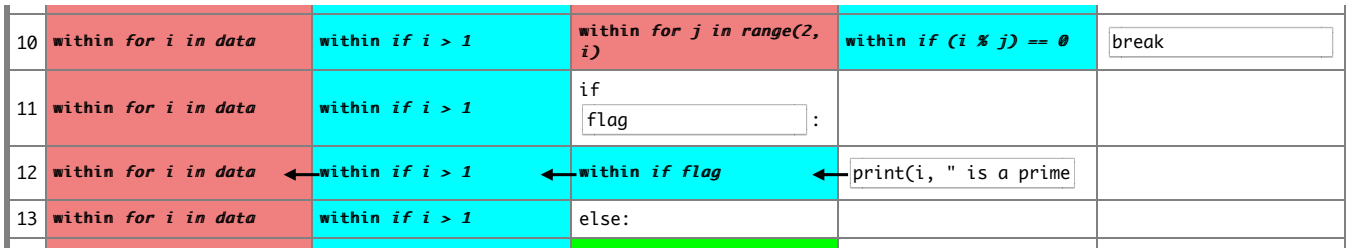


Figure 4: Finding the context hierarchy of a statement in the grid. A user can start from the `Statement cell` and check all the `Indentation cell` with `LEFT Arrow` (shown in Black arrows). Note that `Indentation cells` show the actual condition (which is disabled by default).

at level 1 with auto-completion. Users type *'i'* or *'if'* and select the `if block` option from the drop-down menu (see Figure 5). Grid Editor will put a non-editable `if` keyword, the required colon (:), and an editable placeholder in-between to write the condition. A new row is created just below the current row containing the `within if Indentation cell` at level 1 and a `Statement cell` at level 2 for writing the body. Users write the condition in the placeholder, press `Enter` to go to the `Statement cell` at level 2 in the next row, and finally, write the body of `if block`.

Our initial design only supported writing blocks using auto-completion to reduce syntax errors. However, participants strongly suggested that Grid Editor should allow editing as naturally as possible. Later, we made the auto-completion of the block statements optional. Therefore, users can also write an `if block` manually by typing *'if'*, the condition, followed by a colon(:), and pressing `Enter`. The grid automatically adds the `Indentation cell` and takes users directly to the block's body at the next level in the next row (see Figure 6).

3.4.4 Going Out of Indentation. To go out of a block, users press `Esc` to go to the `Navigation Mode`, use `LEFT Arrow` to determine the `Indentation cell` of the block statement users want to end, and press `Enter` to convert the `Indentation cell` to a `Statement cell` (see Figure 7). Mandatory `Indentation cells` (e.g., the first `Indentation cell` of a block) are prevented from conversion to avoid indentation errors in the code. We initially used `Tab` and `Shift + Tab` to modify indentation levels. However, participants strongly recommended using shortcuts consistent with popular spreadsheet software (e.g., Microsoft Excel). Their feedback also

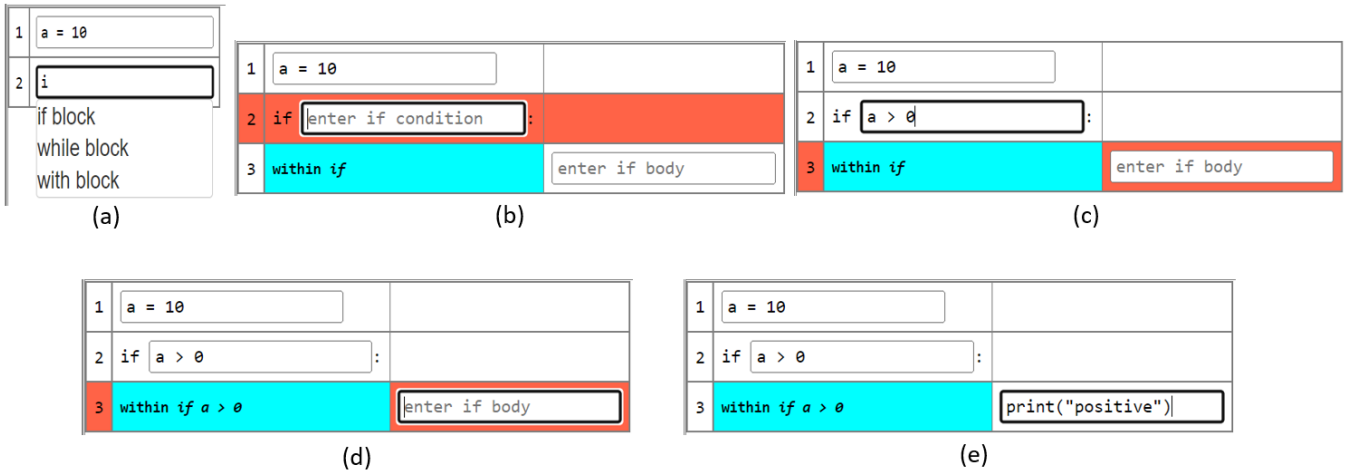


Figure 5: Writing an `if block` with auto-completion feature. (a) Typing *'i'* at row 2 provides a list of suggestions for auto-completion including the option, `if block`. (b) Selecting the `if block` from the list and pressing `Enter` creates a skeleton for `if block`. The input box at row 2 for writing the condition of the `if block` is focused. (c) Writing the condition of the `if block` at row 2. (d) After pressing `Enter`, the focus is switched to the `Statement cell` for writing the body at row 3. The `Indentation cell` is modified to contain the condition. (e) Writing inside the body of the `if block` at row 3.

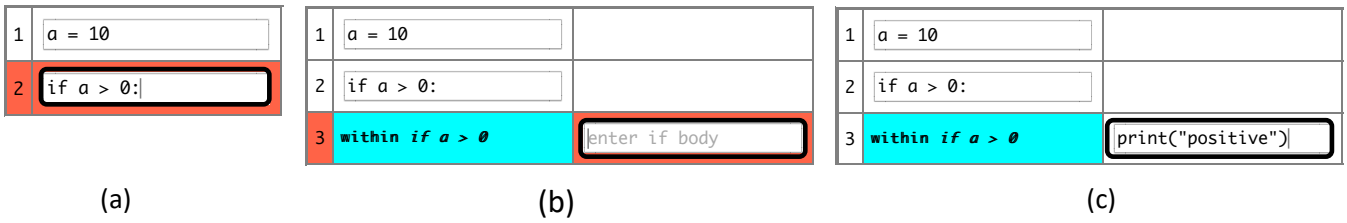


Figure 6: Writing an `if block` manually. (a) Writing the condition of the `if block` at row 2. (b) After pressing `Enter`, the focus is switched to the `Statement cell` for writing the body at row 3. The `Indentation cell` is modified to contain the condition. (c) Writing inside the body of the `if block` at row 3.

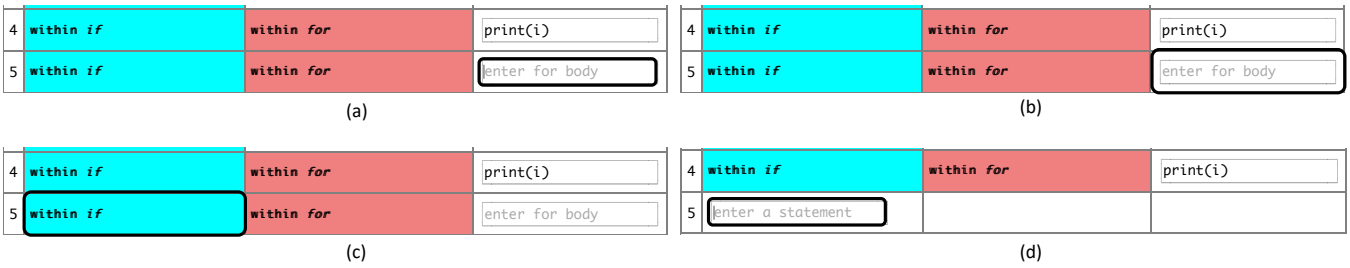


Figure 7: Going out of a block. (a) Creating a new row inside the block. (b) Pressing `Esc to go to Navigation Mode`. (c) Finding the `Indentation cell` where the new level will start. (d) Pressing `Enter` to replace the `Indentation cell` with a `Statement cell`.

suggests that exploring `Indentation cells` before going out of a block helped them avoid writing code at unintentional levels.

3.5 Error Detection in Grid Editor

3.5.1 Error Cues in Grid Editor. Grid Editor plays a beep sound as an error cue to quickly detect syntax errors in the code. Analogous to the squiggly error lines of IDEs, our initial design beeped as soon as a syntax error was detected. However, participants found too many beeps before completing a statement annoying and counter-productive. Instead, they suggested providing cues when users try to leave a `Statement cell` containing an error (e.g., going to `Navigation Mode` or creating a new row). Users also hear the same

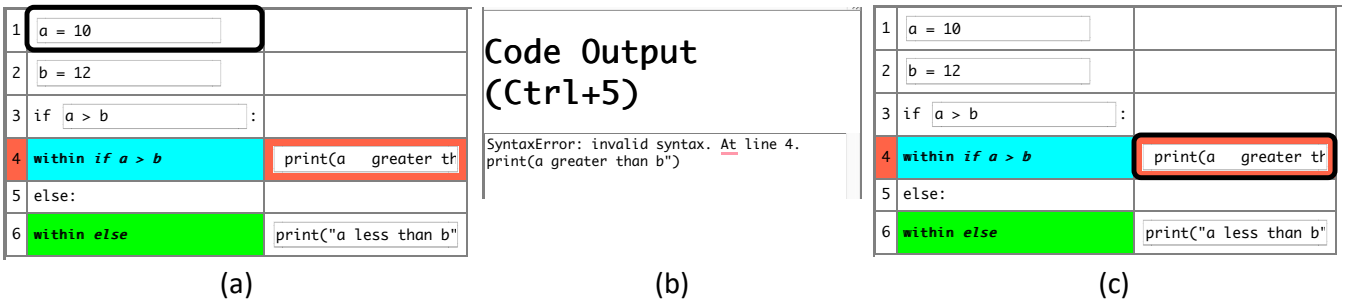


Figure 8: Detecting an error in Grid Editor. (a) Executing the code from row 1 and level 1 by pressing `Alt + Enter`. (b) Keyboard cursor is at Code Output, which can be traversed with `Arrow` keys. (c) Going back to Grid Editor automatically focuses the error `Statement cell` at row 4, level 2.

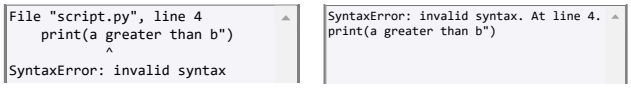


Figure 9: (a) Original error output from a `Python` interpreter. Output contains redundant information such as a visual indicator at line 3. (b) Modified error output in Grid Editor. Only useful information are provided to users in a consistent format, such as the error message, the line number of error, and the error content.

error beep sound in `Navigation Mode` while traversing a row containing an error `Statement cell`. In addition, Grid Editor also periodically announces the location of syntax errors (e.g., *error at line 5*). Initially, we kept the interval of the periodic cue fixed. However, to keep the verbosity at a preferred level, participants suggested providing options to toggle error cues and set interval periods on demand.

3.5.2 Code Execution and Modified Error Message. Users can execute the code anytime from Grid Editor using `Alt + Enter`, and read the output from the Code Output window (see Lower Left of Figure 3). In case of an error, we modified the output to present useful error information to the users. Figure 9a shows the actual error output from a `Python` interpreter, which contains a visual indicator at line 3 indicating the position of the error to sighted users and separating the error message (line 4) from the error content (line 2). Most participants informed that they were confused by the line with the indicator as their screen readers did not announce anything, and they thought the error output ended there. Therefore, we rearranged the syntax error output into 2 lines - the error type, message, and line number in the first line, followed by the error statement in the second line (see Figure 9b).

3.5.3 Quickly Jumping to the Error Statement from Code Output. When users go back to Grid Editor from the Output Window with a syntax error, the grid automatically takes them to the error cell, allowing them to fix the error quickly without navigating the entire code. Figure 8 illustrates this feature.

4 GRID EDITOR: AN INSTANTIATION OF GRID-CODING PARADIGM

In this section, we describe the technology and the implementation strategies of Grid-Coding.

4.1 Implementation Guidelines

We followed the ten guidelines by Philip Guo [33] to make our research prototype scalable and sustainable. Table 3 summarizes the guidelines and how we instantiated each guideline during the development of Grid Editor.

In summary, our prototype is developed with old web technology (e.g., plain HTML and JavaScript) and does not need any installation or login to write and execute code. We do not host any user data on our website. Our prototype is static, stateless, and easy to deploy at any server. Only two developers (authors of this paper) developed the prototype and managed the system (e.g., deploying it on the server). We aim to keep our development team small and integrate only user experience-related feedback to make future developments and maintenance easier.

4.2 Technical Components

To demonstrate Grid-Coding, we implemented a web-based prototype, namely Grid Editor. Figure 3 provides a snapshot of this prototype and its components. In our implementation, we followed the best practices suggested by WAI-ARIA [6] to ensure our prototype is fully accessible with screen readers. The final prototype contains three views: Grid Editor, an implementation of our Grid-Coding paradigm; Text Editor, the go-to editor for BLV programmers; and Code Output, the output window.

We utilized several open-source libraries to develop our system. For example, to parse `Python` code and generate an Abstract Syntax Tree (AST), we used an incremental parser called `Lezer` [5], that can generate an AST even though the code has a syntax error. This feature is particularly useful for implementing our editor because programmers are likely to make syntax errors (unintentional) during code editing, and the parser ensures that a partial AST is generated. To compile the code, we used an external API from `judge0` [4], an online code execution system. Our editor dynamically parses `Python` code while users are editing, generates a list (or table) representation from the `AST` called `ASTab`, uses the

Design Guidelines	Examples from Grid Editor
User Experience	
Walk-up-and-Use	No installation or login required. Users can write code in Grid Editor and execute the code
Should 'Just Work'	Users can work with any Python code with Grid Editor without any setup
Sharing and not Hosting	Grid Editor does not host any user data
Minimize User Option	Grid Editor provides only a minimal number of settings to the users to customize
Software Architecture	
Be Stateless	Grid Editor is stateless, easy to deploy on servers
Use Old Technologies	Grid Editor is designed with `HTML`, `CSS`, and `JavaScript` and following ARIA guidelines
Minimize Dependencies	External dependencies for Grid Editor is minimal. Python parser to create AST is bundled inside the code. Only external dependency is a third-party compiler as Grid Editor is a static website
Development Workflow	
Minimum Developers	Only a couple of developers (authors) implemented Grid-Coding and managed the system
Start Specific	Grid Editor started as an accessible spreadsheet representation of a code that only supports Python
Ignore Most Users	Only a handful of feedback from BLV users has been used to improve Grid Editor that we believed will improve accessibility and user experience of BLV programmers

Table 3: A list of technical design guidelines from [33] that impacted the development of Grid Editor

`ASTab` to construct and synchronize Text Editor and Grid Editor, and notifies syntax errors to users with cues.

4.3 Augmenting Abstract Syntax Tree

Figure 11 shows a simplified grammar for `Python` for the code snippet displayed on Figure 12(Left). Using the grammar, the `Python` code is converted into AST (Figure 12(Right)). The `Python` code contains only 3 lines, whereas the AST representation is complex and contains a higher number of nodes. Note that only the leaf nodes in AST contain a portion of the `Python` code (e.g., the AST node `variable name: a` represents the variable on the first line). The internal nodes of the tree (non-leaf nodes) define the structure for a syntax error-free code. To represent a code in a grid-like structure in realtime, we were interested in extracting useful information from AST and representing it in a tabular structure, which we refer to as `ASTab`.

4.3.1 Structure of `ASTab`. The corresponding `ASTab` representation of the `Python` code in Figure 12 (Left) is shown in Figure 12(Bottom). In `ASTab`, each code line is represented as a row. The *row* and *col* columns contain the line and level information for the Grid Editor, and the *start* and *end* columns represent the start and end character position of a statement in the raw Text Editor. We also store the *parent* line number of each line (`null` for the lines at level 1) and a *scope* list for that line. Finally, we include a marker to indicate whether a line contains an *error*, which is found from a malformed AST.

4.3.2 Generating `ASTab` *from AST.* To generate `ASTab` from an AST, we parse the nodes of the AST from top to bottom using a Depth First Traversal. The intermediate nodes of the tree are used to understand the structure and scope of different statements of the

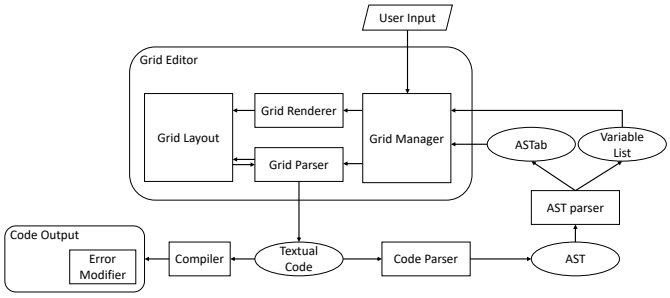


Figure 10: Components and workflow of Grid Editor.

code. Each internal node named `body` is used to keep a running `stack` of all the block statements nested within each other to determine the *scope* and the *parent*. From the leaf nodes, the actual code content is extracted and placed into *stmt* column of `ASTab`. The `variable name` leaf nodes provide the name of all the variables in the code. If the code has any syntax errors, the AST contains an `Error Node`, which is used to determine all the error lines.

A key challenge of working with AST is that whitespaces and blank lines are ignored by AST. However, to generate the tabular structure, we need to handle the empty lines that programmers create by pressing `Enter` from a line. If a new blank line has the same indentation as the previous line, `ASTab` should contain a row with empty content and the same scope as the previous line. However, such information cannot be found in the AST. To handle this special case, we insert a temporary lookahead statement in that blank line, which makes AST incorporate the indentation in front, making the

```
statements: statement+
statement: compound_stmt | simple_stmt

compound_stmt: if_stmt | for_stmt

if_stmt:
    | 'if' named_expression ':' block elif_stmt
    | 'if' named_expression ':' block [else_block]

block:
    | NEWLINE INDENT statements DEDENT
    | simple_stmt
```

Figure 11: A fragment of Python's context-free grammar (CFG) showing the syntax of statements, if_statement, and block [2]. Note that INDENT indicates the presence of a TAB or SPACE character in a block, whereas DEDENT indicates their removal.

AST traversal easier and the ASTab consistent. The content of the lookahead is removed from ASTab.

4.3.3 Extending Grid-Coding for Multiple Language with ASTab. One significant advantage of ASTab is that it does not contain any language-specific information. Instead, it is built on the information available in the AST. Therefore, extending the Grid Editor for any other language is more straightforward with ASTab. Given an AST of another language, we can easily extend our system for that language.

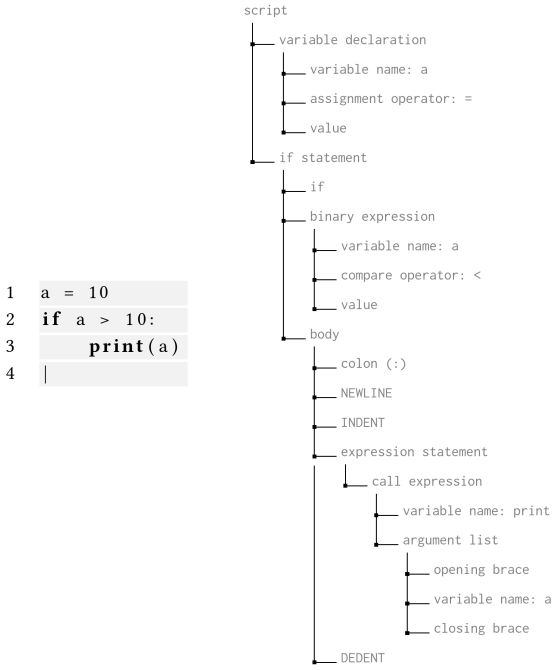
4.4 Components of Grid Editor

Grid Editor contains four components: Grid Layout, Grid Manager, Grid Renderer, and Grid Parser. Figure 10 provides a higher-level overview and workflow of these four components along with external entities, such as the parser and ASTab. We describe the implementation and the functionality of each component below.

4.4.1 Grid Layout. Grid Layout is implemented with an HTML `table` component. Each row is implemented with `tr`, containing a set of `td` elements representing different cells. `Statement cells` contain an `input` element inside to write code. In addition, the `td` elements for `Indentation cells` have attributes for representing unique colors for each block statement.

4.4.2 Grid Manager. Grid Manager is an essential component of Grid Editor, which interacts with all other internal and external components. All user interactions in Grid Editor are handled by Grid Manager. This component maintains the structure of the grid, manages the dynamic auto-completion, and provides interaction cues to users. Grid Manager constantly monitors user input in the grid. As soon as users invoke a keypress, Grid Manager intercepts it for processing. Using the Grid Parser, a textual representation of the code is generated, which is converted to AST and ASTab. Finally, Grid Manager uses the ASTab to reflect changes in the Grid Layout with the help of Grid Renderer.

Grid Manager also prevents users from going out of the grid structure and modifying invalid cells. When a new row is created in the grid, Grid Manager initializes a set of available suggestions



stmt	scope	row	col	start	end	parent	error
$a = 10$	[]	1	1	0	5	*null*	*false*
$if\, a > 5:$	[]	2	1	7	15	*null*	*false*
$print(x)$	[if]	3	2	17	24	2	*false*

Figure 12: (Left) A code snippet in Python instantiating the grammar of a simple_statement and if_statement. (Right) The corresponding *abstract syntax tree* (AST) of the code snippet. Note that INDENT and DEDENT appear as nodes in AST, unlike as an empty character (or removal thereof) in the code snippet). (Bottom) The *augmented abstract syntax table* (ASTab) used in Grid-Coding. Note that columns in ASTab represent the nested nodes in AST, whereas rows represent line numbers of the code.

for the `Statement cell` using the variables list (see Figure 10) depending on the position of the cell. To accomplish that, each cell is assigned cell types - `statement`, `block definition`, and `block body`. A `Statement cell` can have more than one cell types (e.g., a definition of an `if block` inside another block has two cell types - `block definition` and `block body`). Using the cell types and the contents of the cells in the row above, Grid Manager calculates appropriate suggestions for that cell (e.g., no `else block` suggestion without an adjacent `if block`).

4.4.3 Grid Renderer. Grid Renderer changes the DOM elements of the Grid Layout to represent the code in the grid. When a new row is created, Grid Renderer creates a new `tr` element in the DOM. Then, it generates the appropriate `td` elements for `Indentation cells`, `Statement cell`, and `Padding cells`. If this new row creates a new level in the grid, Grid Renderer creates a new column, adds `Padding cells` to all the previous columns to maintain the uniform structure, and takes users to the new level. Similarly, if a new row is created

in the middle of the grid, all the rows after it will be pushed down, and the line number of each row gets modified.

4.4.4 Grid Parser. Grid Parser converts the grid representation to a textual format for the parser to generate AST. In our implementation, first, we scan each row of the Grid Editor in a cell-by-cell manner. The `Statement cell` in the Grid Editor can both be editable inputs or non-editable code contents (e.g., keywords, colons). If a `Statement cell` is detected in the Grid Editor, we find all the children of that particular DOM element and aggregate the text content to generate the textual representation. `Indentation cells` are replaced by a predefined number of whitespaces and `Padding cells` are replaced by `Line Feeds` to separate the lines.

4.5 Text Editor

We included a Notepad++-like plain text editor to write raw code in our prototype. To implement this editor, we used the HTML `textarea`, as the unformatted `textarea` provides the best user accessibility experience to BLV programmers [54]. It supports all text manipulation features that an HTML `textarea` supports. We also included an auto-indention feature for block statements, which users can turn on and off. Users can access Text Editor using the shortcut `Ctrl + 1`.

4.6 Syncing Grid Editor and Text Editor

4.6.1 From Grid Editor to Text Editor. We need to reflect the changes in Text Editor whenever users change the code in Grid Editor. Grid Manager tracks the users' key actions when they write code in the Grid Editor. Then, it invokes Grid Parser to generate a textual representation of the code and places it in Text Editor to sync both the editors.

4.6.2 From Text Editor to Grid Editor. Conversely, Grid Editor must be consistent with the changes made to Text Editor. For this purpose, the contents from Text Editor are extracted first to generate the AST and then the `ASTab`. Finally, `ASTab` is passed to Grid Manager, which invokes the Grid Renderer to represent the code in Grid Layout.

4.7 Code Output

`Code Output` is a read-only HTML `textarea`, which is fully accessible. To run the Python code and view the output, we use an online code execution system called `judge0` [4]. Users can run the code with the shortcut `Alt + Enter`. The content of Code Output can also be accessed without executing the code using the shortcut `Ctrl + 5`. In case of a syntax error, we parsed the raw error output from `judge0` and rearranged it to the format shown in Figure 9b.

5 EVALUATION OF GRID-CODING

We conducted an IRB-approved study to understand the effectiveness of Grid-Coding for performing programming tasks non-visually. The following sections describe this study.

5.1 Participants

We recruited 12 BLV participants (11 males, 1 female) through local mailing lists, university mailing lists, and public posts on online groups for BLV users. Our inclusion criteria included adult BLV individuals with basic Python skills who are fluent in English.

ID	Age/ Sex	Expertise	IDE Used	Profession
P1	39/M	Beginner	Notepad, Vim	Entrepreneur
P2	35/M	Expert	Notepad, Notepad++	IT Instructor
P3	70/M	Expert	Notepad, Notepad++, EdSharp	Music Teacher
P4	47/M	Expert	Notepad++	Network Admin
P5	34/F	Expert	Notepad, Notepad++	Distance Learner
P6	26/M	Expert	Notepad, VS Code	Distance Learner
P7	77/M	Expert	EdSharp	Self-employed
P8*	58/M	Expert	EdSharp, Notepad	QA Engineer
P9*	67/M	Expert	Notepad++, VS Code	Engineer
P10	40/M	Expert	Notepad	AT Instructor
P11	68/M	Expert	Notepad, EdSharp	AT Instructor
P12	54/M	Expert	Notepad++, EdSharp, VS Code	Software Developer

Table 4: Participant demographics (programming expertise are self-reported, * indicates participants with low-vision).

Participant varied in age from 26 to 77 ($M = 51.25$, $SD = 16.76$) and professions: *Engineer* = 2, *Distance Learner* = 2, *Assistive Technology (AT) Instructor* = 2, *Software Developer* = 1, *Entrepreneur* = 1, *Network Administrator* = 1, *IT Instructor* = 1, *Music Teacher* = 1, and *Self-employed* = 1. They self-reported their programming expertise, which we could verify by observing their task performances. None of these participants took part in our earlier participatory design phase. Table 4 presents their demographics.

Note that gender inequality among our participants is a common phenomenon in computer science education in general [12]. Also, note that most participants were experts (self-reported). This is unsurprising because programming, in and of itself, is an advanced technical skill, and only those who are strongly motivated decide to overcome the accessibility barriers to learn how to code.

5.2 Study Design

We used a within-subject design—all participants performed three types of tasks using two study conditions.

5.2.1 Task Design. We chose tasks that represent common programming practices. For example, programmers often utilize code snippets from online forums[2]. Task T1 captures this aspect. Similarly, identifying syntax errors in source code non-visually is challenging but an essential programming skill, which task T2 captures. Likewise, programmers need to implement algorithms or translate high-level design ideas into code. Task T3 captures this routine. Note that tasks are organized in order of their complexity.

The tasks are described as follows. The fourth Task (T4) was exploratory and not measured.

- **T1** *Context Understanding*: For a given code snippet, go to a given line and describe its context (e.g., its current level, the maximum level of the code, and nested information).
- **T2** *Error Correction*: For a given code snippet containing a syntax error, locate the erroneous line and correct it.

[2] https://stackoverflow.com/

T3 *Code Writing*: Implement a given pseudocode in `Python` and produce output by running the code.

T4 *Open-ended (optional)*: *This task was not measured.* Participants explore different features and views of the programming environment.

5.2.2 Study Conditions. Each task has the following two study conditions: Grid Editor and Text Editor, which is the go-to editor for most BLV programmers [15, 45]. Since our task involved code editing, we looked for a research prototype that is accessible, supports `Python`, and enables code editing. Unfortunately, we did not find any such prototype to use as another condition.

C1 *Text Editor*: Participants only use Text Editor with a screen reader. They were allowed to run their code and check the code output. This was our baseline.

C2 *Grid Editor*: Participants must use Grid Editor with a screen reader. They were also allowed to run their code and check the code output. However, they were not allowed to use Text Editor.

5.3 Study Procedure

We counter-balanced the study conditions - half of the participants used Text Editor first, followed by Grid Editor; the other half used the reverse order. The tasks were presented sequentially: T1 followed by T2, followed by T3, by their increasing order of complexity. We considered separate code snippets within a task as trials. Task T1 had 2 trials: find context of a given line in a code that (i) prints whether a given number is prime or not; and (ii) for all numbers 'x' in a list, if 'x' is within 1 and 100, prints whether 'x' is prime or not, otherwise, calculates the sum of all numbers from 0 up to 'x'. T2 had 1 trial: find the error in a code where an addition operator ('+') had a missing operand in the code. T3 had 2 trials: (i) find the maximum of two numbers and print it; and (ii) find the sum of all the positive numbers in a given list. In total, we recorded 120 data-points (= 12 participants * 2 conditions * 5 trials/condition).

Two experimenters conducted the study remotely over Zoom or Google Meet depending on the preferences of the participants. After collecting consent, we first asked participants to introduce themselves, their educational and professional background, their programming experience in `Python`, and their preferred programming environment and tools to write `Python` code.

We then shared the URL of our study prototype (see Figure 3) via email or chat window. All participants used the Chrome browser to run our editor except P7, who preferred a Brave browser. Participants used their preferred screen readers (e.g., NVDA, JAWS, and VoiceOver) with their preferred verbosity and speech rate. Next, we asked participants to share their browser tab running our editor as well as their screen readers' audio. We started recording the session using Zoom or Google Meet software as per their consent.

All participants received a 30-minute training session introducing different features of Grid Editor, shortcuts, and audio cues. Participants wrote simple `Python` code in the Grid Editor, such as declaring and printing variables, performing arithmetic operations, and writing `if statement` to get familiar with the programming environment. Once participants became comfortable with shortcuts and other training materials, we started our experiment by asking them to press a special shortcut (`Ctrl + P`) to load a code snippet representing a task/trial. Then, we described the task and started a timer. We instructed them to perform a task at their regular pace and as accurately as possible. We also instructed them not to ask for help during a task. However, they were allowed to talk aloud to verbalize their strategy and point out editor issues. A trial of a task was completed when participants stated that they were done, or the experimenter noted that it was done, or a timeout (after 10 minutes) occurred. Before proceeding to the next trial/task, we asked them if they had any questions. Experimenters took notes and observed how participants interacted with the system or approached a trial on the shared screen.

After completing all tasks in each condition, we asked the participants to rate that editor on a Likert scale of 1 to 5, where 1 being the least useful and 5 being the most useful. The rating questions were designed to collect their feedback on understating the levels and depth of a code snippet, correcting errors, writing experience, and their confidence during coding on each editor.

In the end, we engaged in open-ended discussion, including their preferred editor, potential use case for each editor, usage patterns, and task performances. Each session lasted for 2 hours. We compensated participants with an hourly rate of USD $25.

5.4 Data Collection and Analysis

We transcribed and analyzed the video recordings of participants to investigate their programming behavior. We observed their approaches and strategies to solve each task. We also noticed their keyboard use by following their cursor movements and listening to screen readers' audio to determine their navigation strategies. We analyzed participants' comments while performing the tasks to understand their reaction towards the editors. We performed a thematic analysis with initial coding to analyze the qualitative data. While performing certain tasks, we observed remarkably similar and novel navigation patterns in Grid-Coding. These patterns are described in §7.

We measured the completion time (in seconds) for all tasks. T1 and T3 had multiple trials, and we averaged the completion time for each trial. In addition, we calculated the accuracy of T1 to determine the understanding of the given source code. Participants received full points by providing the correct indentation level of a given line and the name and order of all the statements in the context hierarchy. We subtracted a point for adding an incorrect or missing a correct statement. Again, for T3, we calculated the number of indentation and non-indentation errors (normalized by the number of block statements) a participant made until getting the desired output.

For quantitative analysis, following recent guidelines for statistical analysis in HCI [30], we intentionally avoided traditional null-hypothesis-based statistical testing in favor of estimation methods to derive 95% confidence intervals (CIs) for all measures. We employed non-parametric bootstrapping [31] with $R = 1000$ iterations. We also reported the mean difference as a sample effect size and Cohen's d as a standardized measure of effect size [29]. Results from quantitative analysis are presented in §6.

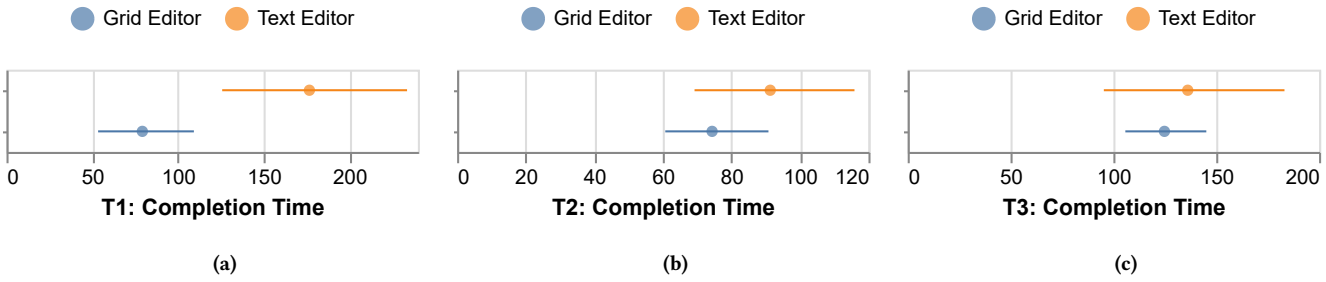


Figure 13: Completion time (sec.) for tasks T1, T2, and T3 in two study conditions (less is better). Error bars show 95% confidence intervals (CIs).

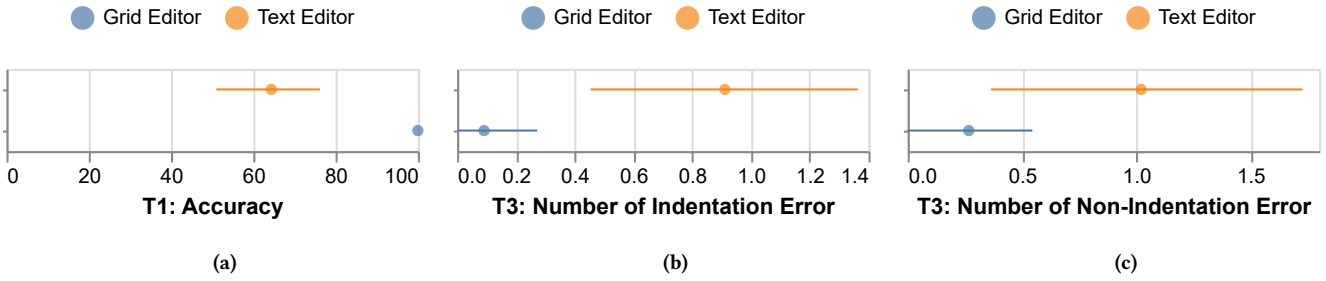


Figure 14: (a) Participants' accuracy (%) in completing task T1 in two conditions (more is better); (b) Number of *indentation* errors per block statement occurred during task T3 in two conditions (less is better); (c) Number of *non-indentation* errors per block statement occurred during task T3 in two conditions (less is better). Error bars show 95% confidence intervals (CIs).

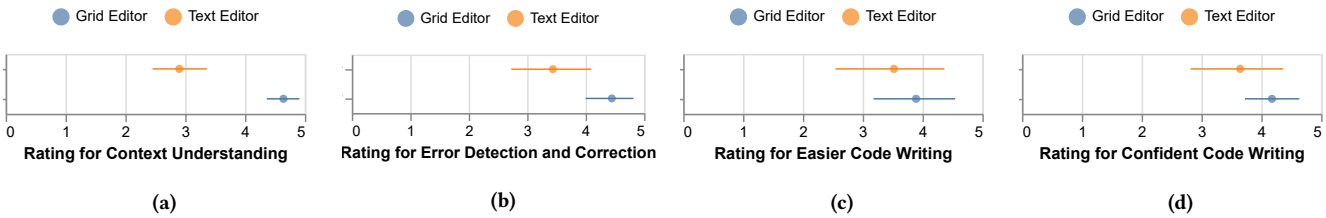


Figure 15: Participants' ratings for two study conditions on a scale of 1 (least useful) to 5 (most useful). (a) Rating for easier context understanding. (b) Rating for easier error detection and correction. (c) Rating for easier code writing. (d) Rating for confidence in not making syntax errors. Error bars show 95% confidence intervals (CIs).

6 QUANTITATIVE ANALYSIS

6.1 Completion Time

We compared the completion time of each task in both conditions. In T1 (*Understanding Context*), participants had to find the context of a statement in a nested block. Participants were faster on average by 94.95*s* (d:1.29) using Grid Editor (mean: 80.13*s*) than Text Editor (mean: 175.08*s*), as shown in Figure 13.a. The high *d* value indicates a large effect. Therefore, Grid Editor can make a large practical difference in understanding context.

For T2 (*Error Correction*), participants had to detect a syntax error in the code and correct it. Participants were faster on average by 17.68*s* (d: 0.5) using Grid Editor (mean: 74.05*s*) than Text Editor (mean: 91.72*s*), as shown in Figure 13.b. The *d* value indicates a medium effect size. Therefore, Grid Editor can help programmers detect and correct errors faster than Text Editor in practice.

For T3 (*Writing Code*), participants had to write a source code. They were faster on average by 11.34*s* (d: 0.16) using Grid Editor (mean: 125.10*s* vs. 136.44*s*), shown in Figure 13.c. The *d* value indicates a small effect size. Therefore, participants had similar performances in both editors.

6.2 Accuracy

For T1 (*Understanding Context*), we measured how accurately participants could find the level in a deeply nested code and all the nested blocks. Trial (i) had the given statement at level 4, and trial (ii) at level 5. Participants were on average 35.42% more accurate (d: 2.26) in Grid Editor (mean: 100%) than in Text Editor (mean: 64.58%)

(see Figure 14.a). The high *d* value indicates a large effect size, indicating that Grid Editor can significantly improve the accuracy of understanding the context.

6.3 Number of Errors

6.3.1 Number of Indentation Errors. For T3 (*Writing Code*), we measured the number of indentation errors per block statement (i.e., total number of errors/total number of block statements) that the participants made during T3. Only a single participant made indentation errors in the Grid Editor while writing block statements manually. Participants made on average 0.83 fewer indentation error (d: 1.28) with Grid Editor (mean: 0.09) than with Text Editor (mean: 0.92), as shown in Figure 14.b. The high *d* value indicates a large effect. Therefore, Grid Editor allows users to manage their indentation in code writing and avoid indentation errors.

6.3.2 Number of Non-Indentation Errors. We also measured the number of syntax errors per block statement other than indentation errors that the participants made during T3. Participants made on average 0.69 fewer indentation error (d: 0.81) with Grid Editor (mean: 0.29) than with Text Editor (mean: 0.97), as shown in Figure 14.c. The *d* value indicates a large effect. Therefore, Grid Editor helps writing syntax error-free code.

6.4 Subjective Feedback

We asked participants to rate both editors in 4 aspects on a Likert scale to understand their experience in using the editors. Participants rated that Grid Editor is easier to get the context and structure of a code (mean: 4.64, SD: 0.50) than Text Editor (mean: 2.91, SD: 0.83), which is shown in Figure 15.a. In terms of finding and correcting errors, participants found Grid Editor to be more useful (mean: 4.45, SD: 0.69) compared to Text Editor (mean: 3.45, SD: 1.21) (see Figure 15.b). Participants gave higher rating to Grid Editor (mean: 3.91, SD: 1.22) than Text Editor in terms of easier code editing (mean: 3.54, SD: 1.69) (see Figure 15.c). As shown in Figure 15.d, participants felt more confident about writing syntax error-free code in Grid Editor (mean: 4.18, SD: 0.75) than Text Editor (mean: 3.63, SD: 1.43).

7 OBSERVATIONS AND USAGE PATTERNS

7.1 Perception of Grid-Coding

When participants were introduced to Grid-Coding for the first time, we could hear the excitement in their voices: *"Oh! I see what is happening. That makes more sense!"*. They could easily understand the structure and the meaning of the cells in the grid. P11 expressed that the `Indentation cell` *"is telling me which block I am in, and I have to go to the next level. That is a cool idea!"*. P10 found the `Padding cell` *"indicates a cell beyond the statement in this row"* and moved to a lower level to find the `Statement cell`. All participants confirmed that the grid was more structured than Text Editor. P5 found that *"the grid is very clear, and the structure is great!"*. P6 preferred the idea of getting more information about a cell by going through the row or the column.

9 participants (P2-P6, P8-P11) reported that the dynamic structure of Grid Editor gave them an outline of the overall source code. From the total number of rows and columns in the grid, they could anticipate the length and complexity of a new code snippet. All participants (except for P9) remarked that the line numbers in the first column were beneficial and informative. P10 thought the line numbers could help him quickly identify a statement in case of an error. P2 provided an insightful analogy about how he perceived Grid-Coding:

> *"Think about coding as walking a road. For a sighted person, he can see the road, but for a blind person, the road is dark. As a result, he may fall. Errors are also like falling down. But if I have something to hold on to while walking, it will help me walk the road more easily. The way I see it, Grid Editor gives me that structure I can hold on to. I always know which line and level I am. Not only that, I have the nesting information like within if, within for in the same line. So, I am always aware of the structure and my position in it. This awareness is the most important source of not making any errors."*

7.2 Usage Patterns in Code Navigation

7.2.1 Usage Patterns in Code Overview. We observed how participants explored a given source code in the grid structure and found that all three cell types supported participants to traverse the code (see Figures 16a and 16b). `Statement Cells` were an indicator for participants to go to the row above or below without changing the level. `Indentation cells` triggered moving at a higher level with the `RIGHT arrow`, and `Padding cells` triggered the opposite within the same row. Although the trajectory of top-down navigation is different from bottom-up navigation, the meaning of the cells was retained for both approaches.

As participants became more comfortable with Grid-Coding, they treated `UP/DOWN + RIGHT Arrow` as a unit operation to go to a higher level of the code, quickly skipping an `Indentation cell` (bending arrow from row 2, column 2 to row 3, column 3 in Figure 16a). Similarly, `UP/DOWN + LEFT Arrow` became a unit to skip a `Padding cell` (bending arrow from row 4, column 4 to row 3, column 3 in Figure 16b). P5 reported that the non-editable `Navigation Mode` was handy for risk-free code navigation. P6 commented: *"If I have to read a more extensive code provided by someone, and if there are complex structures in the code, I can make easier connections among the statements of the code with Grid Editor than Text Editor"*.

7.2.2 Usage Patterns in Finding the Context of a Statement. All 12 participants found the idea of incorporating the maximum amount of information within a row using `Indentation cells` useful and innovative. They repeatedly complained about the difficulty of context understanding in text editors, which requires finding all block statements above that statement and counting the preceding `Spaces/Tabs`. P1 reported two ways to find indentation in a text editor: installing plugins that provide indentation cues with musical notes and counting the `Spaces/Tabs` before each line. However, he found both approaches to be error-prone, time-consuming, and tedious. In Grid Editor, he could *"check the* `Indentation cells` *in a row and know about the indentation and the blocks creating the nested structure"*. Therefore, the grid made indentation easy to understand (P1) and finding indentation a trivial task (P9). We also observed that all participants frequently used the `Ctrl + L` shortcut to listen to

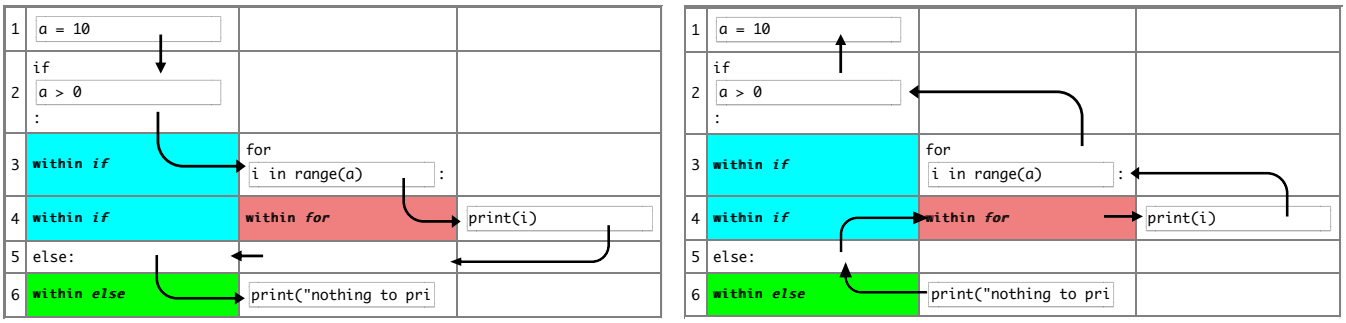


Figure 16: Exploring a Python source code in the Grid Editor. (a) Exploring in a top-down approach. (b) Exploring in a bottom-up approach.

the line number, level, and scope for context understanding during code editing without going to Navigation Mode.

Two participants (P3 and P6) used screen reader plugins to get indentation cues in text editors. P3, a music teacher, used musical notes with varying pitch, and P6 used an aggregated number of Spaces (e.g., *4 Spaces*, *12 Spaces*) to convey indentations. However, when two same-level statements were far apart and separated by statements of other indentation levels, both participants struggled to compare the cues because of the interference of the other statements. As a result, neither of them could correctly determine the context of a statement in Text Editor, but they were successful in Grid Editor.

7.2.3 Usage Patterns in Code Skipping. All participants reported that Grid Editor allowed more granular and flexible source code navigation compared to the linear navigation of Text Editor. P11 thought navigation by levels helps him focus on the part of the code rather than the whole, which he could not do in Text Editor. Moreover, navigating all the statements at a particular level provided him a way to check their alignments in the code to understand the structure and semantics. P2 found the grid useful to get a higher-level summary of the code by navigating only the top level (global scope). The ability to skip the Padding cells and Indentation cells and jump block-by-block with Ctrl + UP/DOWN Arrow made code navigation faster according to 6 participants (P2, P3 P6, P8, P9, and P11) (see blue arrows in Figure 17).

4 participants (P1, P2, P3, and P6) mentioned that the dynamic structure allowed them to investigate the code from the maximum level and quickly locate the portion of the code with a complex nesting structure (see red arrows in Figure 17). According to P6, *"visually, every code goes to the right side because we are putting spaces on the left. I think having the option to access the code from the right side represents similar information that a sighted person gets when they scan through the code and look at the lines that are on the right side because of the deep nesting of those lines. It also gives an idea of how complex the code is."*

6 participants (P1, P3, P4, P5, P8, and P11) found the line number column to be useful to navigate a small source code or quickly move to another line close to the current line (see black arrows in Figure 17). For an extended source code navigation, most participants preferred the Go To option.

7.2.4 Usage Patterns in Code Comprehension. We observed that the two modes of Grid Editor—Navigation Mode and Edit Mode—created a clear distinction between listening to a statement or investing it character-by-character. In Grid Editor, participants usually comprehended the code in Navigation Mode. They occasionally went to Edit Mode to investigate characters if an statement required extra attention to comprehend (e.g., list of numbers, conditions) or could introduce potential syntax errors (e.g., quotes, parentheses). In contrast, code comprehension in Text Editor was mostly accomplished by reading character-by-character to count the number of spaces. In doing so, participants had to find the terminal position of a line by checking the Spaces until they could hear LINE FEED, which indicates the last character of the previous line. This checking was more frequent and random in Text Editor.

7.3 Usage Patterns in Code Editing

7.3.1 Separation of Whitespaces from Source Code. All participants mentioned that using whitespaces to create indentations in Text Editor is annoying and error-prone. For example, P11 preferred a single Space to indent his code in Text Editors to avoid counting a large number of Spaces. However, he had to deal with sighted programmers' code occasionally, which typically contained 4 or 8 Spaces for indentation. Moreover, different editors represented Spaces differently, which was hard for him to understand. All 12 participants were excited that Indentation cells separated whitespaces from the actual code, and they did not need to deal with whitespaces in Grid Editor. P5 expressed -

> *"I like the idea! Because whitespaces depend on how it is set up. Some programs use 4 spaces; some use 5. Other programs may use only 1 whitespace. With Text Editors, you have to know all of that. You have to be able to decipher that. But with the Grid Editor, all of that is right at your disposal."*

7.3.2 Writing Block Statements in Grid Editor. All participants found merit in both manual and auto-completion options to write block statements in Grid Editor. Eight participants (P1, P2, P4, P7, P8, P10, and P11) agreed that the auto-completion was the error-proof option. P1, a novice Python programmer, always preferred using this option because he found fixing indentation errors cumbersome. P10 thought that having both options was a good idea, as it give

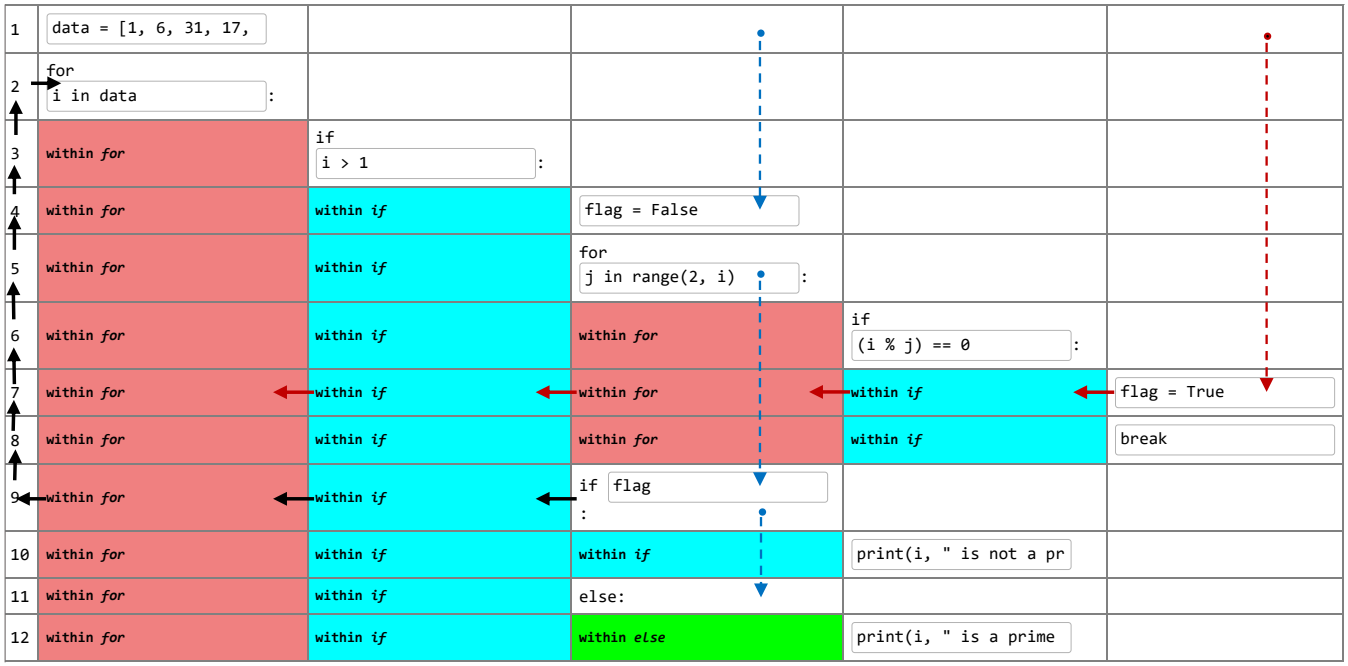


Figure 17: Blue arrows show navigating the `Statement cells` at level 3 while skipping `Indentation cells` and `Padding cells`. Black arrows show the process of finding the context of the statement '`if flag:`' at line 9, and then, using the line number column to go to line 2. Red arrows demonstrate jumping to the statement '`flag = True`' at the maximum level and finding the context of line 7.

users flexibility to write at their own pace. He remarked, *"... the auto-complete ... is good to have for people who are trying to learn and haven't figured out everything yet".*

Expert participants preferred writing the blocks manually as they were more confident about writing error-free code. However, 3 participants (P8, P9, and P11) mentioned that they would use the auto-completion when writing a block that they did not use frequently. For example, P9 did not write `with block` or `try/catch block` as frequently as `if block` or `for block`. Therefore, he could utilize auto-completion to avoid syntax errors in this case. P11 had this to say:

> *"You had me go down the menus when I wrote 'i', and there was an if, a with. That's good too, you know! Because lots of time you forget what to type and you don't want to look up in a book."*

7.3.3 Maintaining Indentation in a Source Code. Maintaining indentation in Grid Editor was quite easy for all participants compared to Text Editor, as levels are represented in columns. Maintaining consistent indentation was particularly challenging in a large codebase with highly nested statements. P6 commented that `Indentation cells` could help him maintain indentation level in a longer source code as he was always aware of the levels and nesting. P11 commented:

> *"When I am writing a larger program [in a text editor], I constantly need to go back and check if everything is aligned. With this [Grid Editor], I never need to do that because I know where everything is. I like that!... I think this will really help us out because, in my case, you know, when I am doing a more complex program, this will save me from having to go back and align everything. Typically, I use F5 to execute the code, and python will come up and tell me 'syntax error'. And that really tells me nothing because I cannot see the alignment. And this [Grid Editor] is perfect!"*

7.3.4 Verifying Structure. In the Grid Editor, we noticed that participants concentrated more on the structure than the contents for rechecking their source code. To check whether they were writing at the correct place, participants used `UP Arrow` to check the previous 1 or 2 `Statement cells`. We observed this behavior frequently when participants were modifying in the middle of the grid. They also checked the `Indentation cells` to ensure they were writing at the correct level. Similar checking in Text Editor included counting the preceding `Spaces` of all statements inside the body as well as the block definitions.

7.3.5 Error Prevention. All participants liked that Grid Editor manages the editable and non-editable cells, allowing them to write only at the valid level in Grid Editor. 7 participants (P1, P2, P4, P5, P7, P8, and P11) found that preventing the modification of mandatory `Indentation cells` could prevent accidental errors. 4 participants (P1, P3, P4, and P8) tried to write an `else block` that did not align with an `if block` but could not find the suggestion in the dropdown menu. After rechecking the code, they found the mistake and appreciated the dynamic auto-completion.

Grid Editor also prevented users from accidentally splitting their code into two separate lines. We noticed 2 participants (P1 and P4) did such splits in Text Editor and later struggled to understand the

mistake as the contents got separated. Another common error in Text Editor was accidentally deleting the wrong number of whitespaces while going out of indentation (P1, P3, P4, P5, P7, P10). P1 and P5 mistakenly took their cursor at the end of the previous line while trying to start from level 1 (global scope), and wrote two statements in the same line. Grid Editor completely removed the possibility of such inconsistent indentations.

7.4 Usage Patterns in Code Debugging

7.4.1 Modified Error Output. The option to execute the code and read the output within Grid Editor was appreciated by all participants. 10 participants (P1-P5, P7-P9, P11, and P12) reported that the modified error output was precise and easy to understand. For example, after reading the error message from the first line, P11 knew what type of error to look for in the next line containing the content. Consequently, before returning to Grid Editor, he had an idea of the position and the reason for the error, which helped him quickly fix it. P5 had this to say after using Code Output -

> *"The output is what I like the most! I can just write the code and press Alt + Enter, and it takes me to the output. In Notepad++, I have to save the file, go to the command line, write the command to run the file, and see if this works. But your output is so simple! Also, when there is an error, the output gives you the line number and tells you exactly what it is!"*

7.4.2 Quickly Jumping to the Error Line. All participants found the idea of jumping to the error row from the Code Output helped them pinpoint and fix syntax errors in the code. P8 found Code Output *"quite convenient"* and liked that *it automatically took him to the error line.* P11 thought such a feature made him a quicker programmer and enabled him to write correct code and deliver faster while working in a team. P4 commented:

> *"I will debug my errors in the grid. I think it will do a very good job, especially when I have a large code to debug."*

7.4.3 Error Cue. All participants agreed that the error cues provided by the Grid Editor helped detect and correct errors more quickly without interrupting them during coding. P6 liked that Grid Editor only reminded him when he left a `Statement cell` with an error. P1 felt more confident about writing error-free code when he did not hear any error cue from Grid Editor.

Similarly, 7 participants (P1, P2, P4, P7, P8, P9, P12) liked the idea of a periodical error cue. P9 liked that Grid Editor beeped or alerted him when he went to a line with an error in the Grid Editor in `Navigation Mode`, which he found helpful to identify any error when working on a code provided by others or copied from the internet. P7 expressed -

> *"The error cue periodically tells me if there is an error in the line and also the line number. So, it is easier for me to go to that line and check the error. It is similar to the visual error cue I would get if I were sighted. As a result, I am always informed, and there is no chance to forget about the error."*

The flexibility of turning off the error cue and configuring the interval period was well-taken by all 12 participants. They expressed the lack of such cues in other text editors and asked whether they could enable the error cues while performing the tasks in Text Editor (*C1* of the study).

8 DISCUSSION AND FUTURE WORK

Support for other languages. Although our current implementation works for Python, we can extend Grid-Coding for any language, given that its AST parser is available. For example, parentheses are visual markers in C/Lisp-style languages like Java. Grid-Coding can represent these markers with two additional `Indentation cells`, containing *open paren* and *close paren* semantics (see Figure A1 in Appendix). In the grid representation, the *open paren* appears in the same line as the text representation. In addition, the *close paren* marker is at the same level as the *open paren* so that BLV users can identify the beginning and end of the scope while traversing a level in the code. To distinguish the block statements from the class definitions and the functions, the `Indentation cells` can similarly contain two additional semantics: *within * class* and *within * function.*

Enable mixed-ability collaborations. We found that the freedom of copying-and-pasting code snippets is a big advantage for text editors that is lost in more structured, restrictive editors like ours. Therefore, we believe coordinating grid and text editors and keeping them side-by-side will be most beneficial. We also believe that such a configuration will enable mixed-ability collaboration (e.g., sighted and blind programmers working together), mixed-skill collaboration (e.g., a novice programmer learning the code syntax [43] from an expert), or help students transition from block-based coding to text-based programming.

All BLV participants in our study used screen readers as their primary assistive technology. But many low-vision programmers with slightly higher visual acuity prefer screen magnifiers over screen readers. Recent work has shown that a grid-like tabular structure can benefit low-vision screen magnifier users in accessing data from the grid [41]. This is certainly encouraging. As such, we will investigate whether or how Grid-Coding can benefit low-vision programmers who use screen magnifiers during programming.

Integration with other IDEs. It is not our goal to compete with existing text editors. Instead, we envision that IDE developers will recognize Grid-Coding as an accessible editor for BLV programmers and adapt it to their product suite. In the future, we plan to integrate Grid Editor with Jupiter Notebook (https://jupyter.org/) to allow BLV programmers to build machine learning models in Python.

Limitations. Most participants in our study were expert Python programmers. Therefore, their opinion and usage patterns may not reflect the expectations of novice programmers. Also, our error detection task did not include semantic errors in programming languages, such as type mismatch or null pointers exception. So the performance of Grid-Coding to support these error types is unknown. Finally, Grid Editor currently depends on a third-party code execution system that does not allow importing external libraries.

9 CONCLUSION

This work proposed a novel programming paradigm named Grid-Coding for BLV programmers to address code navigation, editing,

and related challenges in text-based programming languages. Grid-Coding represents a source code using a 2D grid, where each row represents a line, and each column represents a scope in the code. To support the proposed paradigm, we implemented Grid Editor for `Python` using online participatory design with 28 BLV programmers. To evaluate the effectiveness of Grid Editor, we conducted a user study with 12 BLV programmers who performed code reading, writing, and error correction tasks in Grid Editor and Text Editor. Our quantitative analysis indicated that Grid Editor significantly improved the required time for code understanding, navigation, and error detection. In addition, participants were 100% accurate in understanding context, made almost no indentation errors, and fewer non-indentation errors. They could also effectively navigate and understand source code, manage indentation during code editing, and write syntax-error-free source code in Grid Editor. Finally, we discussed strategies for future implementation of Grid-Coding for other programming languages.

ACKNOWLEDGMENTS

We thank anonymous reviewers for their insightful feedback. This work was supported in part by NIH subaward 87527/2/1159967. The content is solely the responsibility of the authors and does not necessarily represent the official views of the National Institutes of Health.

REFERENCES

[1] 2018. What's New in JAWS 2018 Screen Reading Software. Retrieved September 19, 2018 from https://www.freedomscientific.com/downloads/JAWS/JAWSWhatsNew
[2] 2020. Full Grammar specification. https://docs.python.org/3/reference/grammar.html
[3] 2020. NV Access. https://www.nvaccess.org/. (Accessed on 09/20/2018).
[4] 2021. Judge0 CE - API Docs. https://ce.judge0.com/ (Accessed on 07/26/2022).
[5] 2021. Lezer - Incremental Parser System. https://github.com/lezer-parser (Accessed on 07/26/2022).
[6] 2022. Accessible Rich Internet Applications suite of web standards. https://www.w3.org/WAI/standards-guidelines/aria/ (Accessed on 07/26/2022).
[7] 2022. AST Explorer. https://astexplorer.net/
[8] 2022. Blockly | Google Developers. https://developers.google.com/blockly (Accessed on 07/26/2022).
[9] 2022. PYPL PopularitY of Programming Language. https://pypl.github.io/PYPL.html
[10] 2022. Scratch - Imagine, Program, Share. https://scratch.mit.edu/ (Accessed on 07/26/2022).
[11] 2022. Welcome to Snap! https://snap.berkeley.edu/ (Accessed on 07/26/2022).
[12] Swati Agarwal, Nitish Mittal, Rohan Katyal, Ashish Sureka, and Denzil Correa. 2016. Women in computer science research: What is the bibliography data telling us? *Acm Sigcas Computers and Society* 46, 1 (2016), 7–19.
[13] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. 1986. Compilers, principles, techniques. *Addison wesley* 7, 8 (1986), 9.
[14] 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* (Austin, Texas) *(CHASE '16)*. Association for Computing Machinery, New York, NY, USA, 82–85. https://doi.org/10.1145/2897586.2897616
[15] Khaled Albusays, Stephanie Ludi, and Matt Huenerfauth. 2017. Interviews and Observation of Blind Software Developers at Work to Understand Code Navigation Challenges *(ASSETS '17)*. Association for Computing Machinery, New York, NY, USA, 91–100. https://doi.org/10.1145/3132525.3132550
[16] Khaled L Albusays. 2020. *The Role of Sonification as a Code Navigation Aid: Improving Programming Structure Readability and Understandability For Non-Visual Users*. Rochester Institute of Technology.
[17] Hind Alotaibi, Hend S Al-Khalifa, and Duaa AlSaeed. 2020. Teaching Programming to students with vision impairment: impact of tactile teaching strategies on student's achievements and perceptions. *Sustainability* 12, 13 (2020), 5320.
[18] Dagmar Amtmann, Kurt Johnson, and Debbie Cook. 2002. Making web-based tables accessible for users of screen readers. *Library Hi Tech* (2002).
[19] Apple Inc. 2020. VoiceOver. https://www.apple.com/accessibility/osx/voiceover/.
[20] Chieko Asakawa and Takashi Itoh. 1999. User interface of a nonvisual table navigation method. In *CHI'99 Extended Abstracts on Human Factors in Computing Systems*. 214–215.
[21] 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 *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems* (Seoul, Republic of Korea) *(CHI '15)*. Association for Computing Machinery, New York, NY, USA, 3043–3052. https://doi.org/10.1145/2702123.2702589
[22] David Bau, Jeff Gray, Caitlin Kelleher, Josh Sheldon, and Franklyn Turbak. 2017. Learnable programming: blocks and beyond. *Commun. ACM* 60, 6 (2017), 72–80.
[23] Jeffrey P Bigham, Maxwell B Aller, Jeremy T Brudvik, Jessica O Leung, Lindsay A Yazzolino, and Richard E Ladner. 2008. Inspiring blind high school students to pursue computer science with instant messaging chatbots. In *Proceedings of the 39th SIGCSE technical symposium on Computer science education*. 449–453.
[24] Syed Masum Billah, Vikas Ashok, Donald E. Porter, and I.V. Ramakrishnan. 2017. Speed-Dial: A Surrogate Mouse for Non-Visual Web Browsing. In *Proceedings of the 19th International ACM SIGACCESS Conference on Computers and Accessibility*. ACM, 3132531, 110–119. https://doi.org/10.1145/3132525.3132531
[25] Syed Masum Billah, Vikas Ashok, Donald E. Porter, and I.V. Ramakrishnan. 2017. Ubiquitous Accessibility for People with Visual Impairments: Are We There Yet?. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*. ACM, 5862–5868. https://doi.org/10.1145/3025453.3025731
[26] Syed Masum Billah, Vikas Ashok, Donald E. Porter, and I.V. Ramakrishnan. 2018. SteeringWheel: A Locality-Preserving Magnification Interface for Low Vision Web Browsing. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. ACM, 3173594, 1–13. https://doi.org/10.1145/3173574.3173594
[27] Stephen A. Brewster. 1998. Using Nonspeech Sounds to Provide Navigation Cues. *ACM Trans. Comput.-Hum. Interact.* 5, 3 (Sept. 1998), 224–259. https://doi.org/10.1145/292834.292839
[28] Mechmet Chiousemoglou and Helmut Jürgensen. 2011. Setting the table for the blind. In *Proceedings of the 4th International Conference on PErvasive Technologies Related to Assistive Environments*. 1–8.
[29] Jacob Cohen. 1988. Statistical power analysis for the social sciences. (1988).
[30] Pierre Dragicevic. 2016. Fair statistical communication in HCI. In *Modern statistical methods for HCI*. Springer, 291–330.
[31] Bradley Efron. 1992. Bootstrap methods: another look at the jackknife. In *Breakthroughs in statistics*. Springer, 569–593.
[32] António Ramires Fernandes, Alexandre Carvalho, José João Almeida, and Alberto Simoes. 2006. Transcoding for Web Accessibility for the Blind: Semantics from Structure. (2006).
[33] Philip Guo. 2021. Ten Million Users and Ten Years Later: Python Tutor's Design Guidelines for Building Scalable and Sustainable Research Software in Academia. In *The 34th Annual ACM Symposium on User Interface Software and Technology*. 1235–1251.
[34] Alex Hadwen-Bennett, Sue Sentance, and Cecily Morrison. 2018. Making programming accessible to learners with visual impairments: a literature review. *International Journal of Computer Science Education in Schools* 2, 2 (2018), 3–13.
[35] Earl W Huff, Kwajo Boateng, Makayla Moster, Paige Rodeghero, and Julian Brinkley. 2020. Examining the work experience of programmers with visual impairments. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 707–711.
[36] Joe Hutchinson and Oussama Metatla. 2018. An initial investigation into non-visual code structure overview through speech, non-speech and spearcons. In *Extended Abstracts of the 2018 CHI Conference on Human Factors in Computing Systems*. 1–6.
[37] JAWS – screen reading software. 2022. Navigating Web Pages.
[38] Shaun K Kane and Jeffrey P Bigham. 2014. Tracking @stemxcomet: teaching programming to blind students via 3D printing, crisis management, and twitter. In *Proceedings of the 45th ACM technical symposium on Computer science education*. 247–252.
[39] Rushil Khurana, Duncan McIsaac, Elliot Lockerman, and Jennifer Mankoff. 2018. Nonvisual interaction techniques at the keyboard surface. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. 1–12.
[40] Mario Konecki, Alen Lovrenčić, and Robert Kudelić. 2011. Making programming accessible to the blinds. In *2011 Proceedings of the 34th International Convention MIPRO*. IEEE, 820–824.
[41] Hae-Na Lee and Vikas Ashok. 2022. Customizable Tabular Access to Web Data Records for Convenient Low-vision Screen Magnifier Interaction. *ACM Transactions on Accessible Computing (TACCESS)* 15, 2 (2022), 1–22.
[42] Hae-Na Lee, Sami Uddin, and Vikas Ashok. 2020. TableView: Enabling Efficient Access to Web Data Records for Screen-Magnifier Users. In *The 22nd International ACM SIGACCESS Conference on Computers and Accessibility* (Virtual Event, Greece) *(ASSETS '20)*. Association for Computing Machinery, New York, NY, USA, Article 23, 12 pages. https://doi.org/10.1145/3373625.3417030
[43] Yuhan Lin and David Weintrop. 2021. The landscape of Block-based programming: Characteristics of block-based environments and how they support the transition to text-based programming. *Journal of Computer Languages* 67 (2021), 101075.

[44] Stephanie Ludi, Lindsey Ellis, and Scott Jordan. 2014. An accessible robotics programming environment for visually impaired users. In *Proceedings of the 16th international ACM SIGACCESS conference on Computers & accessibility*. 237–238.
[45] Sean Mealin and Emerson Murphy-Hill. 2012. An exploratory study of blind software developers, In Visual Languages and Human-Centric Computing (VL/HCC), 2012 IEEE Symposium on. *Proceedings of IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC*, 71–74. https://doi.org/10.1109/VLHCC.2012.6344485
[46] 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.
[47] Farhani Momotaz, Md Touhidul Islam, Md Ehtesham-Ul-Haque, and Syed Masum Billah. 2021. Understanding Screen Readers' Plugins. In *The 23rd International ACM SIGACCESS Conference on Computers and Accessibility*. ACM, 1–10. https://doi.org/10.1145/3441852.3471205
[48] Lourdes Moreno, Xabier Valencia, J Eduardo Pérez, and Myriam Arrue. 2018. Exploring the Web navigation strategies of people with low vision. In *Proceedings of the XIX International Conference on Human Computer Interaction*. 1–8.
[49] Aboubakar Mountapmbeme, Obianuju Okafor, and Stephanie Ludi. 2022. Addressing Accessibility Barriers in Programming for People with Visual Impairments: A Literature Review. *ACM Transactions on Accessible Computing (TACCESS)* 15, 1 (2022), 1–26.
[50] NVDA-Project. 2020. GitHub - nvaccess/nvda: NVDA, the free and open source Screen Reader for Microsoft Windows. https://github.com/nvaccess/nvda. Accessed: 2020-06-29.
[51] Afra Pascual, Mireia Ribera, Toni Granollers, and Jordi L Coiduras. 2014. Impact of accessibility barriers on the mood of blind, low-vision and sighted users. *Procedia Computer Science* 27 (2014), 431–440.
[52] Venkatesh Potluri, Priyan Vaithilingam, Suresh Iyengar, Y. Vidya, Manohar Swaminathan, and Gopal Srinivasa. 2018. CodeTalk: Improving Programming Environment Accessibility for Visually Impaired Developers. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems* (Montreal QC, Canada) *(CHI '18)*. Association for Computing Machinery, New York, NY, USA, 1–11. https://doi.org/10.1145/3173574.3174192
[53] Dominic Roberts and Karlton Weaver. 2011. Audio Aids in Source Code. *Retrieved September* 19 (2011), 2017.
[54] Emmanuel Schanzer, Sina Bahram, and Shriram Krishnamurthi. 2019. Accessible AST-Based Programming for Visually-Impaired Programmers. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education* (Minneapolis, MN, USA) *(SIGCSE '19)*. Association for Computing Machinery, New York, NY, USA, 773–779. https://doi.org/10.1145/3287324.3287499
[55] Ann C. Smith, Justin S. Cook, Joan M. Francioni, Asif Hossain, Mohd Anwar, and M. Fayezur Rahman. 2003. Nonvisual Tool for Navigating Hierarchical Structures. *SIGACCESS Access. Comput.* 77–78 (Sept. 2003), 133–139. https://doi.org/10.1145/1029014.1028654
[56] Dimitris Spiliotopoulos, Gerasimos Xydas, Georgios Kouroupetroglou, Vasilios Argyropoulos, and Kalliopi Ikospentaki. 2010. Auditory universal accessibility of data tables using naturally derived prosody specification. *Universal Access in the Information Society* 9, 2 (2010), 169–183.
[57] Andreas Stefik, Roger Alexander, Robert Patterson, and Jonathan Brown. 2007. WAD: A feasibility study using the wicked audio debugger. In *15th IEEE International Conference on Program Comprehension (ICPC'07)*. IEEE, 69–80.
[58] Andreas Stefik, Andrew Haywood, Shahzada Mansoor, Brock Dunda, and Daniel Garcia. 2009. Sodbeans. In *2009 IEEE 17th International Conference on Program Comprehension*. IEEE, 293–294.
[59] Andreas Stefik, Christopher Hundhausen, and Robert Patterson. 2011. An Empirical Investigation into the Design of Auditory Cues to Enhance Computer Program Comprehension. *Int. J. Hum.-Comput. Stud.* 69, 12 (Dec. 2011), 820–838. https://doi.org/10.1016/j.ijhcs.2011.07.002
[60] Andreas Stefik and Richard Ladner. 2017. The quorum programming language. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*. 641–641.
[61] Andreas Stefik and Susanna Siebert. 2013. An empirical investigation into programming language syntax. *ACM Transactions on Computing Education (TOCE)* 13, 4 (2013), 1–40.
[62] Andreas M Stefik, Christopher Hundhausen, and Derrick Smith. 2011. On the design of an educational infrastructure for the blind and visually impaired in computer science. In *Proceedings of the 42nd ACM technical symposium on Computer science education*. 571–576.
[63] Jaime Sánchez and Fernando Aguayo. 2006. APL: Audio Programming Language for Blind Learners. https://doi.org/10.1007/11788713_192
[64] Bruce N Walker, Amanda Nance, and Jeffrey Lindsay. 2006. Spearcons: Speech-based earcons improve navigation performance in auditory menus. Georgia Institute of Technology.
[65] David Weintrop. 2019. Block-based programming in computer science education. *Commun. ACM* 62, 8 (2019), 22–25.
[66] Kristin Williams, Taylor Clarke, Steve Gardiner, John Zimmerman, and Anthony Tomasic. 2019. Find and Seek: Assessing the Impact of Table Navigation on Information Look-up with a Screen Reader. *ACM Trans. Access. Comput.* 12, 3, Article 11 (Aug. 2019), 23 pages. https://doi.org/10.1145/3342282
[67] Kristin Williams, Taylor Clarke, Steve Gardiner, John Zimmerman, and Anthony Tomasic. 2019. Find and seek: Assessing the impact of table navigation on information look-up with a screen reader. *ACM Transactions on Accessible Computing (TACCESS)* 12, 3 (2019), 1–23.

A APPENDIX

A.1 Grid Coding for C-style Languages

Since Grid-Coding utilizes the abstract syntax tree (AST) of a programming language, we can easily extend it for any programming language. For example, Figure A1 presents how Grid-Coding can render a piece of Java code. Note that it introduces several new `Indentation cells` (e.g., *open paren*, *close paren*, *within * class*, and *within * function*) to convey Java-specific semantics. The grid representation of a sample Java code is shown in Figure A1 (';' is optional in Grid-Coding). Notice the different positions of opening curly braces in Figure A1(a) for the `if` statement (line 3) and the `else` statement (line 8).

```
1  public class Example {
2      public static void main(String args[
3          int a = 10;
4          if (a > 0) {
5              doSomething();
6          }
7          else
8          {
9              doSomethingElse();
10         }
11     }
12 }
```

1	public class Example	open paren		
2	within Example class	public static void main..	open paren	
3	within Example class	within Main function	int a = 10;	
4	within Example class	within Main function	if (a > 0)	open paren
5	within Example class	within Main function	within if	doSomething();
6	within Example class	within Main function	within if	close paren
7	within Example class	within Main function	else	
8	within Example class	within Main function	within else	open paren
9	within Example class	within Main function	within else	doSomethingElse();
10	within Example class	within Main function	within else	close paren
11	within Example class	within Main function	close paren	
12	within Example class	close paren		

Figure A1: Extension of Grid-Coding for Java (C/Lisp-style) programming language. (Left) A sample Java code written in Visual Studio Code. (Right) Representation of the Java code in a grid. All Grid-Coding principles, such as each row representing a single line, each column representing a level (except for the first column representing line numbers), and all three cell types are also present. Two additional semantic cells - *open paren* and *close paren* - have been used to replace the curly braces that enclose a scope in Java. These markers always appear at the same level for BLV users to find them while traversing a level. Further, as Java always contains functions within a class, we added two additional semantics - *within * class* and *within * function* in the `Indentation cells` to distinguish them from other block statements. Block statements have the same `Indentation cells` as before for understanding context.