Block-Based Editing in a Textual World

Tom Beckmann
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
tom.beckmann@hpi.uni-potsdam.de

Lukas Böhme
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
lukas.boehme@hpi.uni-potsdam.de

Marcel Taeumel
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
marcel.taeumel@hpi.uni-potsdam.de

Robert Hirschfeld
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
hirschfeld@hpi.uni-potsdam.de

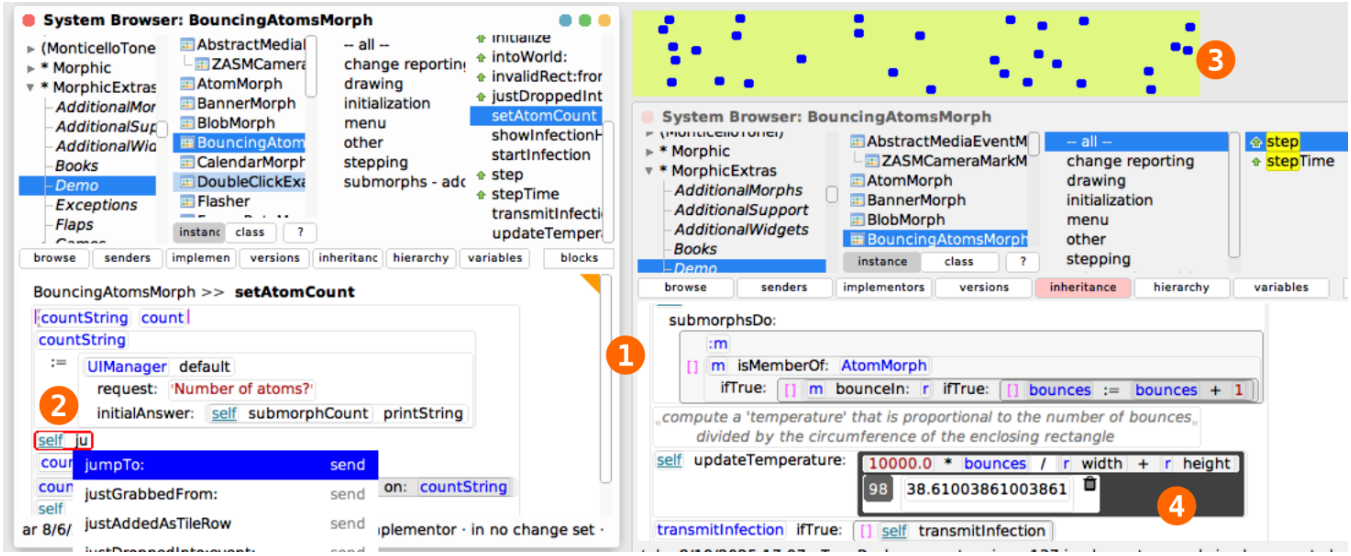

Figure 1. A screenshot of our block editor embedded in the Squeak/Smalltalk code browser. Shown are (1) our block editor, on the left and right, (2) the block cursor, (3) a running program, and (4) a probe showing the program's runtime values.

Abstract

Block-based programming environments offer unique benefits for editing or integration of visual tools that could be useful across programming environments. However, most general-purpose programming environments are designed around textual representations of programs.

We explore the design of a block-based editor that integrates with existing textual environments, using the example of the Squeak/Smalltalk development environment. Through a user study, we show that users felt generally comfortable with our editor's edit interactions when compared to textual editing. We discuss the design's difficulties and opportunities we observed during our user study and our own use to help propose block-based editor designs that integrate well with text-based environments.

CCS Concepts: • **Software and its engineering** → Formal language definitions; *Visual languages*; **Integrated and visual development environments**.

Keywords: Block-based Editing, Structured Editing, Visual Programming, General-purpose Programming

ACM Reference Format:
Tom Beckmann, Lukas Böhme, Marcel Taeumel, and Robert Hirschfeld. 2025. Block-Based Editing in a Textual World. In *Proceedings of the 4th ACM SIGPLAN International Workshop on Programming Abstractions and Interactive Notations, Tools, and Environments (PAINT '25), October 12–18, 2025, Singapore, Singapore.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3759534.3762681

PAINT '25, Singapore, Singapore

ACM ISBN 979-8-4007-2160-1/25/10
https://doi.org/10.1145/3759534.3762681

1 Introduction

Block-based programming languages are successfully used for teaching [21]. Their visual and direct nature lends itself well for encouraging experimentation with fast feedback loops and fewer errors, compared to their textual counterparts. Integrating visual tools within a block-based editor also appears easier compared to text editors: where text editors have to accommodate syntax errors, arbitrary deletion commands, or line-based text layout, integrating visual tools in a block-based editor is as simple as placing the relevant visual tool within a block. As such, we see potential benefits to gain from using of block-based editing not just in education but also in professional or general-purpose programming for easier integration of visual tools within general-purpose programming languages.

Indeed, projects such as GP [14] and Snap*!* [9] demonstrate that general-purpose programming is possible within block-based editors. At the same time, the majority of professional programming environments and their ecosystems are centered on text and keyboards for input. Textual programming languages tend to be structured in ways that appear not to translate well to block-based editors [23]: textual languages appear to favor many small building blocks that compose to flexible wholes simply by adding or omitting specifiers, such as "const" or "final", or expressions, such as a chain of setter calls, whereas block-based editors must either produce a separate block for each permutation, include slots for each specifier, or carry extra user interface elements that allow users to configure the block in-place. Other small differences add minor points of friction. For instance, block-based editors typically distinguish between expressions and statements, whereas in many textual languages top-level expressions automatically become statements.

In this paper, we explore a design for a block-based editor that seeks compatibility with the existing environments of general-purpose textual programming languages, using the example of a language with a small grammar, Smalltalk. To begin generalizing our insights, we also describe early prototypes of our concept for a subset of the JavaScript and Scheme programming languages. We address space efficiency to allow comfortable reading of larger programs (Section 2), editing interactions in the absence of well-designed block palettes and the presence of large API surfaces through a *block cursor* (Section 3), describe an evaluation of its usability (Section 4), and discuss bi-directional compatibility with the surrounding textual environment (Section 5).

The presented editor concepts moved through three design stages. In the first stage, which resembled practice-led craft research [5], feedback from one of the authors and informal user testing with other programmers allowed us to find a baseline design (shown in Figure 2). In a second stage, we ran a user study (n=8) using the baseline design to analyze how our concept compares to text editing and identify further areas for the concept to improve. In a third stage, one of the authors continued to use our editor concept over the course of multiple months for all Smalltalk development activities. Through this third stage, we identified and also fixed many points of friction for co-existence with a textual development environment to arrive at a concept that is usable as main editing tool (shown in Figure 1).

2 Mapping from Text to Blocks: Clarity vs. Space Efficiency

In this section, we will describe the visual layout of our editor and its design rationale. At the design's core is a need to resolve the tension between visual clarity and space efficiency. Screen space usage has been identified as a major issue for visual languages [7, 16, 18]. Block-based editors typically use large insets, strong borders, and distinct colors for their blocks. As a result, the blocks are clearly distinguishable but even small programs tend to occupy large portions of a screen. Since we want to be able to display existing textual programs in the context of the tools users are familiar with, we seek a design that gets as close to the space use of textual Smalltalk as possible while maintaining clear boundaries between blocks.

As a first step for the mapping, we need to identify the language's elements that turn into distinct blocks. The Smalltalk programming language has a small grammar compared to other popular textual programming languages like Python or Java. Its major components are three types of message sends, assignments, return statements, blocks, arrays, and literals. Control flow primitives and other structures that often extend a language's grammar are syntactically realized in Smalltalk through message sends and blocks.

Message sends in Smalltalk are either unary, binary, or keyword messages. Unary messages appear as `2 squared`, binary messages as `2 + 3`, and keyword messages as `2 raisedTo: 3` or `5 clampLow: 0 high: 10`. Here, the message selectors are respectively `#squared`, `#+`, `#raisedTo:`, and `#clampLow:high:`.

Our block-based editing interface for Smalltalk is designed to be bi-directionally compatible, so we want to ensure that programs remain legible to people familiar with textual Smalltalk. At the same time, the block-based structure of our editor imposes some constraints on the possible edit operations that would otherwise be possible in text, as we will describe later on in more detail. To clearly communicate these constraints, we change some of the syntactic conventions of textual Smalltalk to avoid inadvertently making users believe that the editing operations familiar to them from text were possible.

Our visual layout is adapted from a design described in prior work [3]. We map the parse tree of the default Squeak/Smalltalk [11] parser one-to-one to blocks. Its granularity matches the vocabulary that users familiar with the language would expect, comprising, e.g., assignments, message sends,

or arrays, and no elements that exist purely for technical reasons, such as "parenthesized elements" or "keyword part". Each node of the parse tree is mapped as a block in the form of a nested rectangle. The rectangles' visuals aim to support the communication of the nested structure, as during editing, valid operations are constrained by the nesting.

We use color in two ways: first, brightness of the blocks' background color increases with deeper nesting. To make the levels clearly distinguishable for common use cases, we settled on nine levels. When blocks reach plain white, a subtle border still allows users to discern boundaries. Second, for each Smalltalk block closure, we shift the hue of the background color or give a stronger border. As a result, groups of linear control flow are visually separated.

Editing in our editor occurs on a block level, which corresponds to full parse nodes of a program's parse result. Consequently, it is not possible to delete a single parenthesis or move it separately. During early prototyping, users were commonly attempting to delete parentheses when they appeared in the locations they are familiar with from text. To better communicate the valid editing operations, we use the syntactic characters as a form of icon by prefixing blocks with them, as visible in Figure 2 (note the square brackets `[]` that are surrounding the block's content in the textual Smalltalk grammar). Users can still recognize the element as they are familiar with the syntactic characters, but they no longer appear to offer an affordance for deletion, as they are removed from the text flow within the blocks.

Round parentheses in Smalltalk serve to override precedence of operations, as in most other programming languages. As the nested tree structure already encodes the precedence, we opted to remove round parentheses, removing another source of confusion when it comes to supported editing operations. Further, we omitted most whitespace from the mapping and employed an automatic layouting system that attempts to find a near-optimal layout within linear time [27]: each type of block, such as message sends or assignments, communicates positions for soft- and hard-line breaks. The layout algorithm then traverses the tree structure, attempts to lay out blocks in a single row and falls back to introducing line breaks where soft line break markers have been placed, starting from outer-most elements. As a result, the block layout is subject to only few jumps while users are typing, as line breaks occur near the root of the tree, and closely resembles a layout that authors of Smalltalk code manually create.

An initial version of the editor omitted all whitespace. However, as a form of secondary notation, whitespace is commonly used to, for example, visually group statements by introducing empty lines. As the importance of this function became evident in our own use of the editor, we introduced the ability to add empty lines within any statement block.

The insets around the contents of each block are the deciding factor for the editor's space efficiency, or lack thereof.

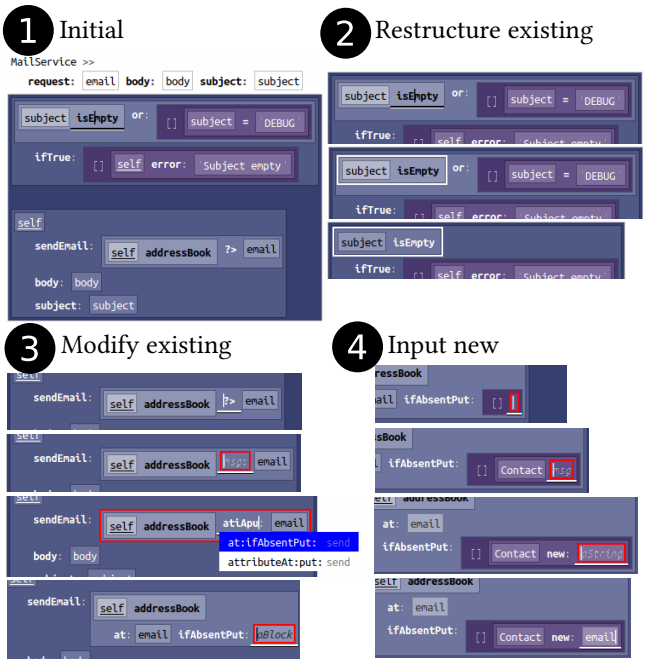

Figure 2. An edit history of a Smalltalk method for sending mails. Refer to Section 3 for a detailed description. (1) shows the initial method. In (2), we restructure the condition at the top to remove the "DEBUG" clause. In (3), we modify the existing access to the address book and adapt it to save a new entry by fuzzy-matching against the autocompletion. In (4), we type out an expression to create a new contact. These interactions are carried out through the keyboard and our block cursor without need for a mouse or a block palette.

By removing the insets, our use of space and visual layout would be nearly identical to text. However, the differences in node color would be invisible except in the leaf nodes.

A dense layout is generally preferable, as it allows users to reference more information without the need for navigation actions, as long as it remains clearly legible. The final dimensions of the method result from the sum of the height of all lines but only the maximum of the width of all lines. Thus, as a compromise between clarity of nesting and space efficiency, we always allocate horizontal insets. Vertical insets are applied only to keyword message sends with more than one message part, where the block structure is otherwise not readable: the keyword message send must be clearly distinguishable as a continuous block nesting several other blocks.

An alternative design could have kept the round parentheses, which disambiguate the nesting of keyword message sends. We decided against this since the parentheses would then, once again, appear like an editable part of the text flow.

3 Editing Through a Block Cursor

In terms of editing, we chose to investigate keyboard-centric interactions that resemble those of textual editors. Since the environment of Smalltalk is designed around textual editing, its libraries also feature large API surfaces and are often designed for composition of many small message sends as opposed to few large blocks.

Our editor design allows the use of drag-and-drop for editing and also features block palettes. For keyboard-centric interactions, we propose an approach to editing we call the *block cursor*. The block cursor allows users to create or modify blocks through their keyboard. During edit interactions, the block cursor ensures that the program remains in a valid state. In our design, a program is considered valid if it is syntactically correct in its textual equivalent with the sole exception that we allow empty leaf nodes, often named *holes* in structured editors.

Figure 2 shows a scenario where a programmer uses the block cursor to edit a Smalltalk method that is sending an email. The programmer wants to remove a debug clause from the top of the method and ensure that previously unknown email addresses are added as contacts to our address book.

1. First, the programmer moves the block cursor to the `subject isEmpty` expression.
2. The programmer then uses the cut-around shortcut to cut out the `DEBUG` block surrounding the block cursor.
3. The programmer moves the block cursor to the `?>` operator that is querying the address book. They start typing a fuzzy match of the desired symbol `at:ifAbsentPut:` to filter the autocompletion menu. Upon accepting the suggestion using the tab key, the block cursor is moved to the hole of the new missing argument.
4. The programmer starts typing a square bracket to create a closure. They proceed to type out the textual equivalent of the desired expression, `Contact new: email`, and the block cursor automatically creates the required block structure as each token is completed.

Conceptually, the block cursor can be considered as a three-element tuple consisting of

- the currently selected block,
- the currently active mode, which is either the text input mode or the block selection mode, and
- a clipboard containing zero or more blocks.

This tuple forms the context that controls the availability and outcome of actions.

In the following, we first describe navigation using the block cursor, then block input, restructuring subtrees of blocks, and finally modification of block leafs.

3.1 Visual Navigation Using The Block Cursor

Navigation using the block cursor is designed with two constraints in mind: first, the block cursor's movement follows the visual layout of the blocks rather than structural properties of the AST, and second, the block cursor enables direct keyboard input whenever possible.

To ensure the constraints, the cursor traverses the blocks from leaf node to adjacent leaf node. As such, it mimics the behavior of a text cursor that traverses between visually adjacent elements. We initially experimented with cursor movement that follows the tree's pre-order enumeration but almost all test persons we asked during formative user tests preferred movement along the visual structure.

As another consequence, the block cursor skips blocks that only act as containers for other blocks, such as the closures in Figure 2. The block cursor will thus move only between places that accept direct input, or leaves.

When the programmer needs to address container structures, such as the aforementioned closures, the block cursor's selection mode allows to enlarge selections. Through the Shift+Up shortcut, the block cursor selects the next block along the currently selected block's chain of parent blocks. In the example in step 2 of Figure 2, the programmer uses this function to select the full `subject isEmpty` expression starting from the `subject` leaf node in the tree.

Vertical cursor movement is equivalent to the behavior of cursor movement word processor: we locate the horizontally closest cursor position that appears below or above the current cursor position. The cursor always selects a specific block and within that block a specific position in the block's text, if any. When the cursor is moved over the boundary of a nested text field, it automatically jumps to an adjacent block.

3.2 Grammar-assisted Input

Since input using the block cursor is designed to closely resemble textual input, we derive rules for interactions from the language grammar. These rules form a system we call *grammar-assisted input*.

In the simplest case, the block cursor is currently in a hole, such as the empty statement in step 4 of Figure 2. Holes exist in different contexts: the empty statement accepts any Smalltalk expression, while the hole for the message send part in step 3 accepts only identifiers with a colon postfix.

Input handling can be modeled as a state machine that transitions between different grammar rules. Using the example of a hole for the expression statement in Smalltalk, we can transition to all commonly used grammar rules using just the first typed character. The complete set of rules is as follows:

- given a digit, create a number literal,
- given a letter or an underscore, create an identifier,
- given a single quote, create a string,
- given a double quote, create a comment,
- given a hashtag, create a symbol,
- given a dollar sign, create a character,

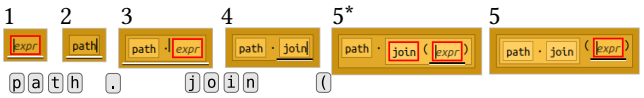

Figure 3. (1) The programmer starts with the block cursor located in a hole. (2) upon typing the first "p" of "path", the block turns from a hole to an identifier. (3) the dot character may not appear as part of an identifier, the block cursor will thus wrap the current block in a binary operator structure (specifically an attribute access), which can contain the dot character. (4) when typing "join", the input is again rejected for the currently active binary operator structure and instead gets moved to the next hole. (5*) the programmer pressed the opening parenthesis character, so the block cursor wrapped the current block in a method call. This is an intermediate step that the user will not get to see. (5) as the precedence of the dot operator is higher than that of the method call operator in JavaScript, the block cursor now instructs the method call to perform a left-rotation, restructuring the tree according to the expected structure.

- given a curly brace, create an array,
- given a square bracket, create a block closure,
- and, if we are in the first expression of a block closure and we receive a colon, create a block binding.

Notably, this mapping prevents two comparatively rare constructs in Smalltalk from being entered: byte arrays use `#[0 1 2]` as syntax and literal arrays use `#(0 1 2)`. Since the above listing commits to a symbol upon seeing a hashtag, neither of these constructs can be created.

For languages with more complex grammars, such as C, this same problem occurs much more frequently. For instance, typing an f in a place for a statement in C could, among others, indicate a reference to a variable, a type for a declaration, a type for a definition, or a for-loop. Our system would have to wait for a non-alphanumeric character to appear to begin disambiguating, or prompt the user for manual disambiguation. In this editor design, we opted for manual disambiguation for the Smalltalk array as the other two types of arrays are used infrequently: after creating a dynamic array, the user can select it and issue a conversion command.

Similar to determining the type of a block through input, the block cursor interprets single invalid keystrokes according to the programmer's most likely intent and automatically performs a tree restructuring. For example, if the programmer types a + while in an identifier block, the block cursor rejects the input for the identifier block as it is not a valid character in an identifier as per the language's grammar. Instead, the block cursor wraps the number block in a binary operator with the identifier block on the left-hand side and a new hole on the right-hand side of the operator. If the programmer now types another letter while the block cursor is still in the operator block, the block cursor again rejects the input but forwards it to the hole in the operator's right-hand side. This interaction can be seen in steps (1) to (4) of Figure 3 for the attribute access operator in JavaScript.

During early user testing, three important properties of grammar-assisted input became apparent.

Do not interfere with muscle memory. During our study pilot, programmers routinely pressed the space key to separate the plus operator from its operands, which is, as previously described, not necessary since the grammar-assisted input automatically restructures or forwards input. In earlier versions of the block cursor a press of the space key created a new message send block around the selected block, such that users were left with two additional message send blocks they did not mean to create when typing a binary addition the way they would in a text editor. Correspondingly, we added further rules to the handling of the space key that instead move the cursor to adjacent blocks in the way users would expect from a text editor.

Follow operator precedence. Languages that have complex precedence rules require the block cursor to automatically rotate the tree as the programmer is typing a longer expression. For example, in JavaScript, if the programmer types `path.join(`, the structure shown in step 5 of Figure 3 should be the result. Without awareness of precedence, the `(` character would wrap the `join` block in a method call, rather than the `path.join` compound, as seen in step 5 of Figure 3. The block cursor produces the expected results by first performing the simple wrap operation and then asking the newly added outer block whether its precedence dictates that it should have wrapped around the next parent as well. If so, we perform a tree left-rotation with the outer block as pivot, restructuring the tree to fit the programmer's expectation.

For Smalltalk, we break this rule for message sends as formative user tests showed that the simple rule of always sending to the currently selected block led to more predictable behavior. As an example, consider the statement `2 = 3 ifTrue : []`. When typed in our editor the same way as in a text editor, the textual equivalent of the resulting program would instead be `2 = (3 ifTrue: [])`, as the `3` is the selected block when we begin creating a message send using the space key. This finding was somehow surprising to us but may be due to Smalltalk having a very simple precedence system that already breaks with users' expectations, for example for mathematical binary operations of the form `2 + 3 * 4`, which in C would have to be expressed as `(2 + 3) * 4`.

Support backspace to undo. Previous studies [12] have shown that programmers often use backspace to undo accidental changes, which also matches with observations from our study. To support this as part of grammar-assisted input, the block cursor ensures that any automatically created structures can be removed again by hitting the backspace

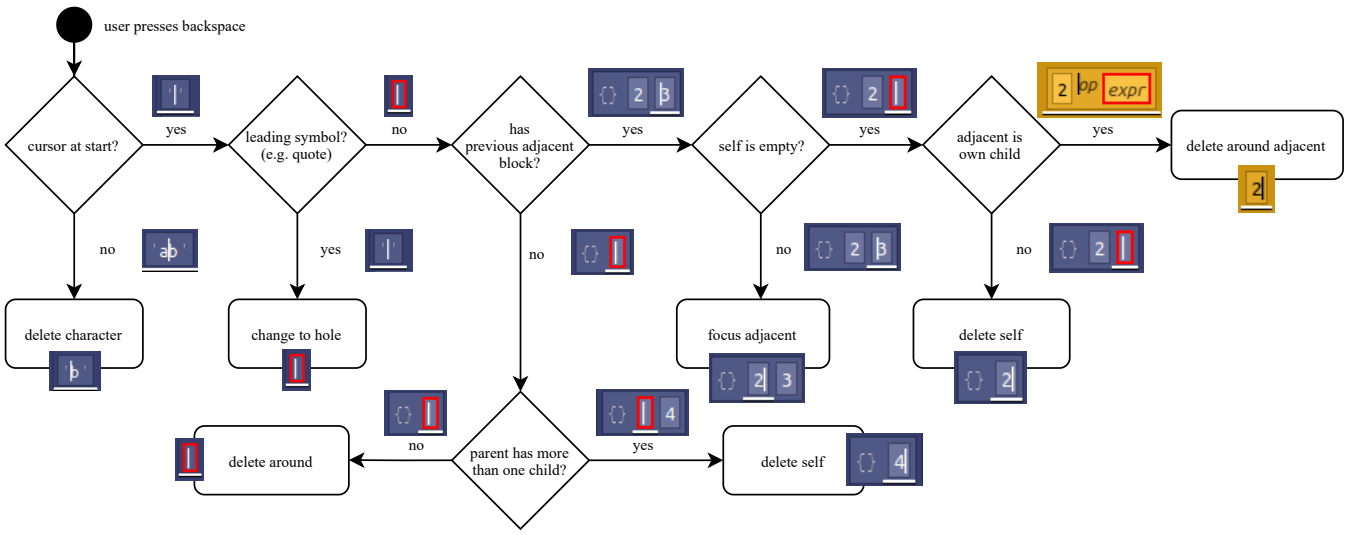

Figure 4. Flowchart illustrating the decision tree for matching against the block structure to handle backspace in a way that allows undoing the creation of blocks that resulted from erroneous user input. Each decision is illustrated by an example block configuration. Blocks appearing in blue are from our Smalltalk implementation, blocks appearing in yellow are from our JavaScript implementation.

key. The rules we derived for this have so far been able to generalize to all block constructs that our grammar-assisted input produces. Figure 4 illustrates this system.

3.3 Restructuring Trees Using Expressive Actions

As described, grammar-assisted input will automatically restructure trees when the user's input conveys an intention. To allow users to manually restructure blocks where there is no direct equivalent through normal text input, as for example in step 2 of Figure 2, the block cursor supports a set of expressive restructuring actions.

Most restructuring interactions with the block cursor make use of its clipboard. To provide more control, the block cursor offers users specializations of the usual cut, copy, and paste actions as shown in Table 1, such as pasting before or after the selection.

Table 1. Overview of restructuring actions.

Action	Specializations		
insert	before	after	
insert statement	before	after	
paste	before	after	replace
copy	self	around	
cut	self	around	
swap	left	right	
rotate subtree			

Most programming languages have only few distinct shapes that valid subtrees assume. Binary operators, for example, naturally create binary trees. Another important structure are sequences: these appear as a, typically arbitrary, number of siblings in a subtree. Languages use sequences for lists of arguments, statements in a block, or elements of an array literal. We refer to these blocks that map to sequences of language constructs as *sequence blocks*.

Sequence blocks share a set of dedicated actions for manipulating their contents. Programmers insert new elements before or after their current selection in a sequence using an insert shortcut from Table 1. The sequence block will decide what form the new element should take, usually taking the form of a hole waiting for input. Additionally, sequence blocks allow pasting the block cursor's clipboard before and after the selected block. As sequences of statements are likely the most often used type of sequence in programs, we introduce dedicated actions for inserting new statements above and below the statement containing our block cursor. These actions will always create a new statement, independent of the block cursor's nesting depth in an expression.

For more complex restructuring, the cut-around action is an often helpful tool. It is used in step 2 of Figure 2 to remove the second condition clause: the action replaces the selected block with its parent, giving the impression of deleting the block around the selection. It will additionally place the now removed parent in the block cursor's clipboard and leave a hole where the selected block used to be. Upon pasting, the clipped block wraps around the selection, rather than replacing it. This allows programmers to quickly move outer expressions around.

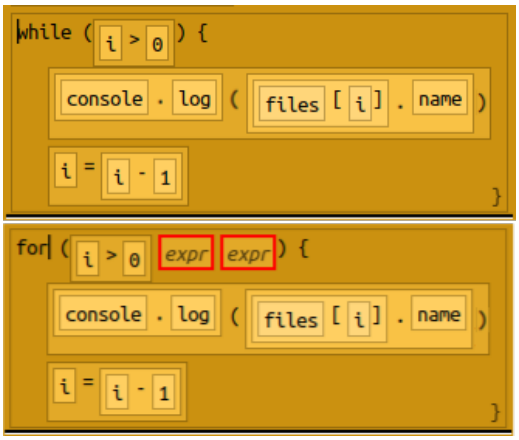

Figure 5. A JavaScript `while` loop iterating over a list of files. The programmer can change it to be a `for` loop by editing the keyword's label. The arity of the block will automatically adapt, adding holes where JavaScript's semantics of for-loops require the loop condition and step. The existing arguments and statements remain unchanged.

3.4 Modifying Blocks In-place

Not allowing the user to change parts of a block that would cause a rippling effect to its arguments avoids syntax errors and allows the environment to guide programmers when assembling their block program [23]. To still support common cases, especially those where users are used to be able to perform these changes very easily in text, the block cursor allows programmers to also change parts of blocks that imply changes to its structure in-place while making sure the result remains well-formed. Consider editing a while loop to instead use a for loop, as illustrated in Figure 5. The programmer moves their cursor into the keyword and changes it to `for`. Since we know from JavaScript's grammar that a for loop has three slots for elements in its clause, we adapt the clause by adding new holes but keep the first element. The programmer could now proceed to move the condition to the second slot using the swap command. For the reverse, we drop the now superfluous blocks in our editor. Alternatively, the blocks could have been "ejected" and placed next to the method, such that users can still refer to them if needed.

Similarly, blocks can be reinterpreted, allowing programmers to keep a block's content but change its semantics, rather than having to create a new block and move the content. For example, a variable or string block can be automatically reinterpreted as any type of block that requires one textual label when pasted.

In our implementation, these types of edits require custom handling logic in the classes implementing the respective structures. Either they convert from one to structure to another, or, as with the for-to-while example, we abstracted these language components into a single class based on their structure. In this case, we have a single "KeywordWithBlock" class that supports if, while, and for constructs in JavaScript.

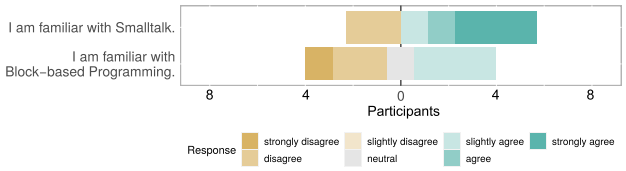

Figure 6. Most participants rated their familiarity with Smalltalk above average and indicated varying experience with block-based programming systems.

4 Evaluation

We conducted a within-subjects user study to get qualitative insights on how programmers edit programs using the block cursor.

4.1 Study Design

Participants were given two editing tasks and performed each once in Visual Studio Code (VS Code) and once in our editor implementing the block cursor concept, for a total of four conditions. Our goal was not to simulate a realistic programming session but to understand how users' approach to edit operations that commonly occur during programming sessions may differ in our editor. To that end, we designed our tasks around a previously observed distribution of common edit actions [12]. To ensure participants would perform similar sequences of actions, allowing us to draw comparisons, we showed the participants screenshots of each step's final result and asked them to reproduce the shown code. While participants performed the changes, we took note of erroneous inputs and means of performing the changes.

In the first task, participants extended an existing Smalltalk class implementing the Observer pattern with the functionality to subscribe to specific topics. The source file comprised 50 lines of code, with 20 lines requiring changes as part of the task. In the second task, participants were asked to type an implementation of the quicksort algorithm in Smalltalk from scratch, spanning 20 lines of code. We alternated the editor that participants start with to mitigate effects of familiarity with the code samples.

Participants were allowed to use all functionalities they were aware of in the default configuration of both editors, including autocompletion or multi-cursor editing. Our editor integrates with Squeak/Smalltalk's [11] built-in autocompletion system and we made sure to seed VS Code's autocompletion with all needed identifiers before the task.

Participants. We recruited 7 graduate students and 1 professional programmer for our study (5 male, 3 female). Three participants had seen our prototype prior to the study in an

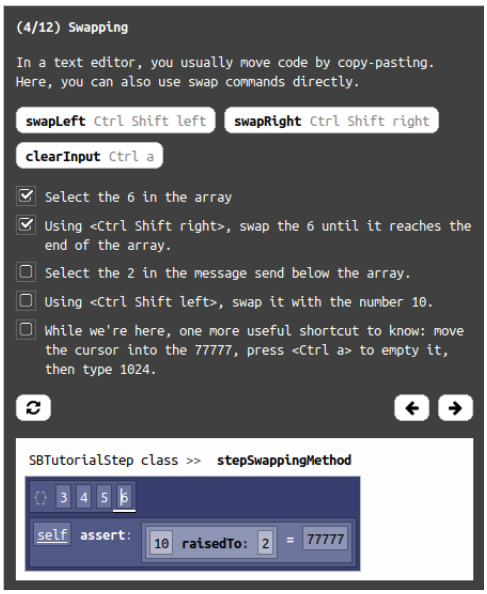

Figure 7. A screenshot of the tutorial that participants in the study were able to walk through before working in our editor. The relevant shortcuts are shown as buttons near the top and a number of tasks are presented and automatically checked off once completed in the code snippet below.

earlier version with a different set of shortcuts. The participants reported 3 to 5 years of professional programming experience and 5 to 15 years of programming experience in general. They rated their familiarity with Smalltalk as 2 - 7 out of 7 and their familiarity with block-based programming as 2 - 5 out of 7, as seen in Figure 6.

Procedure. We used TeamViewer[1] to conduct the study remotely: participants controlled a remote machine that had the editors set up and had a key tracker running in the background (which the participants were informed of). The screen was setup such that the left two-thirds were taken up by the participant's editor and the right third showed the next set of changes to be performed. The instructors were able to see the participants' screens to take notes on their usage behavior. We asked participants to think aloud while performing the tasks, to get insights into their usage and identify interactions where the cognitive load appeared higher.

Directly before starting the first task in our editor, we asked participants to walk through an interactive tutorial of the shortcuts of our editor prototype shown in Figure 7. During the tutorial, participants were allowed to ask the instructor for clarification. The time required to complete the tutorial varied heavily between participants, as some participants had already begun commenting on the editor design during the tutorial. After finishing all conditions,

[1] https://www.teamviewer.com/en-us/, accessed: 2025-06-27

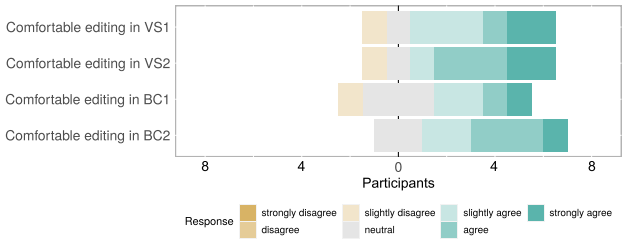

Figure 8. Participants reported whether they felt comfortable with performing the editing tasks in our editor (BC) and in the text editor (VS).

participants filled out a survey with 7-point Likert scale questions on their experience and impressions, followed by an unstructured interview with the instructor.

4.2 Results

In the following, we report and interpret the survey results, think-aloud comments, interview responses, and observations. We refer to participants as P1 through P8. Quotes are translated from German.

Editing Experience. Participants reported whether they felt comfortable performing the edit tasks through Likert scales. Most participants either slightly or strongly agreed, as indicated in Figure 8. Those who disagreed or were neutral noted that they were still unsure of some of the interactions and that they would need some more time to get used to the interaction scheme. We discuss some of the challenges participants reported or we observed below.

Erroneous Input. The participants made errors in both editors. In VS Code, participants incorrectly entered essential pieces of syntax (colons to delineate message sends, dots to terminate statements, incorrect nesting of parenthesis) in 12 out of the total 16 runs. In our editor, participants made structural mistakes in 6 out of the total 16 runs, with some recognizing and fixing them later. We will further discuss erroneous inputs participants made in the specific sections below.

Navigation. All participants except for P8 were able to intuitively move the block cursor around in all four directions with some mentioning that "it behaves just like a text cursor". P8 expected the navigation to follow the tree structure, rather than the visual structure of the blocks, and struggled throughout the study. Besides movement in the cardinal directions, some participants appeared to initially struggle with identifying when to use the shortcut to enlarge selections "up the tree". Others actively used the enlarge selection shortcut to explore the exact nesting and association of blocks to their surroundings.

Grammar-assisted Input Across Languages. To find to what extent the block cursor's grammar-assisted input

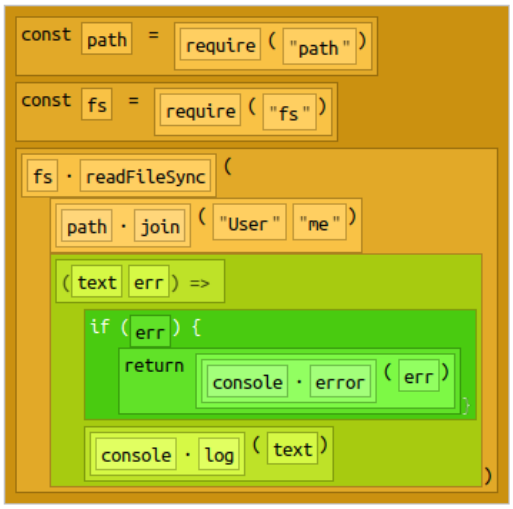

Figure 9. A snippet of JavaScript that we asked participants to try and recreate without further explanation of the JavaScript block syntax and interactions.

generalizes across languages, we also asked participants to enter the JavaScript block structure shown in Figure 9 as part of the study. We did not ask participants' to rate their experience with JavaScript but only ensured that they had used the language before and were familiar with the syntax. All participants were able to reproduce the snippet, requiring no further instructions beyond those received for Smalltalk, except for two cases of confusion: first, to add an argument, users should press the generic insert actions. Instead, participants tried to use the comma key. None initially used the insert action but most tried the shortcut just after.

Second, we designed keyword control structures, which do not exist in Smalltalk, such that pressing the space key after the keyword would expand the structure, e.g., typing `if` and pressing space would result in the expected block structure. We have since adapted it to also expand when the opening parenthesis is pressed, which was what most participants expected, likely because pressing space explicitly was not required in other parts of the interaction design, unless letters of two consecutive tokens appeared. Beyond those two cases, participants voiced that they felt well supported by the grammar-assisted input system, with some mentioning that it behaved according to their expectations.

Restructuring. Multiple participants showed excitement when discovering the swap and wrap restructuring actions during the tutorial, noting that these operations often feel cumbersome in text editors. All participants were initially confused about the cut-around action but most stated that they understood its intent after using it on more examples. The fact that the clipboard is automatically filled after all delete (cut) operations required some care, leading to some

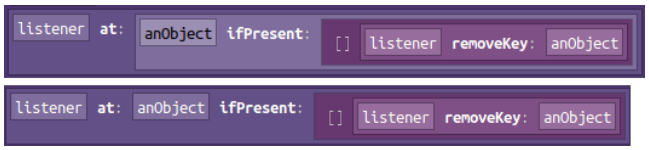

Figure 10. Participants frequently split the `at:ifPresent:` message send in two separate blocks (top), rather than extending the existing `at:` message send with a second `ifPresent:` message part (bottom). The textual equivalents would be `listener at: (anObject ifPresent: [listener removeKey: anObject])` vs. `listener at: anObject ifPresent: [listener removeKey: anObject]`.

participants overriding their clipboard contents when allocating space for the paste operation. This is an issue that also occurs in the Vim[2] editor, where delete actions fill the default clipboard register as a side effect. As a solution in Vim, any delete action can be prefixed with a special shortcut to address a specific clipboard register.

Participants often started making selections when they did not necessarily need to do so. This was in particular noticeable with the "insert statements" action, which works independent of block nesting as explained in the tutorial, but participants stated that they often did not feel certain where the new statement would go unless they first move the cursor to a block adjacent to where they want the new statement to appear. We also noticed cases where selections caused mode errors [19]: our design constrained actions based on whether or not a selection was made. We have since adapted the corresponding actions to provide the behavior that participants expected.

Modifying Blocks. Participants frequently made use of the option to reuse existing blocks and adapt their labels, including the option to adapt the arity of a message send. A common source of errors was the use of the "wrap in message send" action as opposed to the insert action to extend an existing message send, as shown in Figure 10. Participants noted that they would have likely not run into this problem had they formulated the code themselves, as they most often simply misread the nesting as shown in the screenshot they were asked to recreate. This points to a need for improving our visualization of blocks.

In the version of the prototype used during the study, our Smalltalk implementation used distinct types of blocks for assignments and message sends. This was another frequent source of errors, as participants did not realize that converting between the two was not straightforward, see Figure 11. The participants attempted to enter the tokens that make up assignments in Smalltalk in the text field for the message send, indicating that merging structurally similar AST nodes into one type of block supports users' intuition: specifically,

[2] https://www.vim.org/, accessed: 2025-08-18

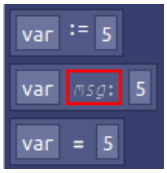

Figure 11. At the top, the dedicated assignment block of an earlier version of our prototype is shown. Below, is the empty message send block and a message send with the = operator, which participants frequently confused with the assignment operator.

an assignment operator behaves structurally identical to a binary message send.

4.3 Threats to Validity

Internal. As participants were asked to think-aloud, their behavior and impressions may have been altered. The presentation of the tasks was optimized to facilitate comparison of sequences of interactions across participants and do not necessarily reflect the typical way that a specific programmer would tackle the presented tasks. Some participants stated that they expected to be able to perform both tasks more quickly if they had planned out the actions themselves, rather than copying them from our instructions. Since the authors of the editor also conducted the study there is a risk of participant response bias [8].

External. It is likely that participants who received more training will perform differently in our editor. In our observations, reported experience with Smalltalk also appeared to influence efficiency and confidence with editing in both editors. For example, participants with more reported experience with Smalltalk appeared to struggle less with entering the more complicated multi-part keyword messages (such as `5 clampLow: 0 high: 10`), which is a construct rarely seen in other textual programming languages. Since we did not collect quantitative measures we cannot say if this is a significant factor. Repeating the experiment with a different, maybe syntactically more complex, language could yield different results.

5 Integrating into an Existing Environment

To find how our proposed design fits into the existing environment of Squeak/Smalltalk, one of the authors used the editor almost exclusively for all Smalltalk-related programming tasks over the course of several months. In the following, we describe and discuss the resulting experience.

5.1 Integration with Squeak and the Operating System

Initial versions of our editor opened methods on a 2D infinite canvas, comparable to CodeBubbles [6]. Later versions then integrated with the default Smalltalk browser, as seen

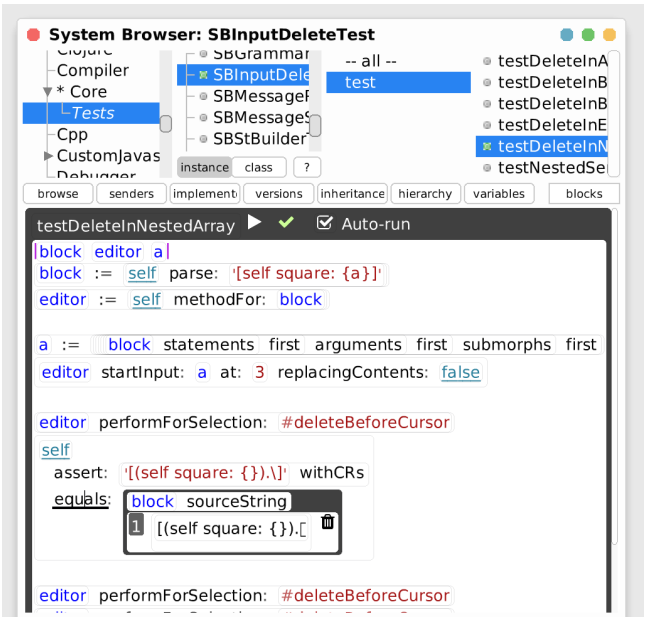

Figure 12. Screenshot of the default Squeak/Smalltalk browser with our editor as the editing widget. Shown is a test method with a wrapper that automatically executes the test on save, and a watch showing the value observed during execution for the wrapped expression.

in Figure 12. While the author still used the canvas when prototyping, especially for live programming loops, the integration with the browser proved important for its ease of access to the APIs of classes.

Early on, we invested considerable time in improving automatic formatting of textual code, as the blocks omit any formatting. Consequently, the formatting of the textual code our editor produced had to look at a minimum legible and ideally idiomatic, for use in textual contexts such as code reviews in online code repositories.

Another important aspect turned out to be bi-directional synchronization with the default system clipboard. Notably, the block cursor maintains a block clipboard that stores a reference to the copied block object. When copying a block, we also write a textual version of the block into the system clipboard. The user can thus paste the textual version of the block externally. Conversely, when pasting, we first inspect the system clipboard. If the same string we wrote for the last-copied block is still in the system clipboard, we paste the block reference. Otherwise, we attempt to parse the textual clipboard as blocks and insert them. On parse failure, the user currently has to manually correct the source first. In practice, this happened rarely, likely in large part because of the simple grammar of Smalltalk.

For integrating with the large API surfaces of the Smalltalk classes, we used an autocompletion popup as it is common in most IDEs. This proved to be an effective tool to create blocks.

When completing, we left holes for any arguments, such that the user can use the tab key to jump to the next pending insertion points. More generally, the explicit hole concept coupled with tab and shift+tab for navigation between holes became an important part of the author's workflow. At times, the author even created new holes using the newline shortcut as jump marks to return to in a bit.

5.2 Tools inside the Block Editor

As the blocks expose the structure of the program node tree directly, creating domain-specific tools that integrate tightly with the source code was simplified. For example, we created a wrapper for test case methods that shows a run button, its pass/fail state, and even has a toggle for automatically re-running it on save. The editor broadcasts events of general interest to all tools, such as our test case wrapper, so that they can react when methods are saved or similar.

The tools we created either attached to nodes or, more commonly, replaced them. For example, to show a watch tool around an expression that displays the last value that the expression evaluated to, we changed the actual method source code to contain our instrumentation and then wrote a tool that matches against the instrumented program nodes and shows the widget in their place. This approach yielded two notable advantages: first, we did not have to maintain a source map or similar approach to translate between the source code as shown to the user and the source code as executed by the runtime including instrumentation. Second, this approach made copy-paste of tools near effortless, as the copy interaction would copy the underlying instrumented code and, upon pasting, the instrumented code would once again be replaced by the tool.

5.3 Generalizability

As Squeak/Smalltalk already uses interface elements instead of plain text for all structures except method source code and has a relatively simple grammar, integrating into the Squeak/Smalltalk may be easier than in other environments. To take first steps toward understanding generalizability to other languages, we created editors for subsets of JavaScript and Scheme using the block cursor.

As Scheme has even fewer structurally different constructs than Smalltalk, its implementation for the block cursor ended up smaller and simpler. In our implementation, we ended up with specific implementations for the different common literals, boolean, string, symbol, number, and a general expression element that could optionally be quoted. By omitting specific elements for special forms like `if` or `define`, users are able to change the semantics of the special form while keeping any arguments. We still communicate well-formedness of special forms by adding an error highlight if their structure is invalid. In the palette, we offer pre-built compositions of special forms that only omit the non-fixed parts. Beyond the flexibility that we gain from the simple structural model, we also noticed through our own experiments using the editor the usefulness of the well-established shortcuts found in editors designed for S-Expressions, such as the "slurp" and "barf" operators as found in ParEdit [3] that are used to take in or eject elements from S-Expressions.

For languages with larger grammars, a significant challenge is to enumerate and derive suitable interactions for all blocks. For example, the JavaScript parser babel.js lists more than 80 types of AST nodes in its implementation [2]. Kogi, a tool to derive block-based languages for Google Blockly from context-free grammars demonstrates an automated approach to map an existing language to blocks [24]. While this reduces the effort significantly, it currently results in high numbers of distinct blocks for large languages.

For our prototype, we chose to manually implement the language decomposition into blocks. The guiding principle was to merge as many blocks as possible based on structural similarity: rather than having separate blocks for "while" and "for" loops in JavaScript, there is only a single type of keyword block that has a text field to specify its exact semantics, as seen in Figure 5. Coupled with the ad-hoc transformations of the block cursor's grammar-assisted input, this allows making large parts of a language available with just a few block types, reducing the cognitive load on the programmer using the block-based programming system. As an important side-effect, this supports the ability to modify existing blocks as changing a block to a structurally compatible one only requires changing a keyword.

5.4 Correspondence to Block-based Programming

As outlined in Section 1, we wanted to find a design for block-based editors to fit in the context of an existing textual general-purpose programming language. The design we settled on might almost appear closer to textual editing than to block-based editors in terms of visuals.

To ease a transition for users coming from block-based editors, we integrated a custom theme that makes increases block insets for easier drag-and-drop, and chooses colors based on method categories, as most block-based editors, as opposed to nesting depth. A screenshot can be seen in Figure 13.

Similarly, we integrated a block palette such that users can interact with the editor both through text input and autocompletion, as well as purely through drag-and-drop.

6 Related Work

The block cursor concept builds on ideas of existing block-based environments, syntax-directed or syntax-supported editing, and implementations of projectional editors.

Block-based programming editors are often designed for programming education or experimentation, for example, Scratch [21] or Snap [9]. GP is an editor that resembles

[3] https://www.emacswiki.org/emacs/ParEdit, accessed: 2025-08-18

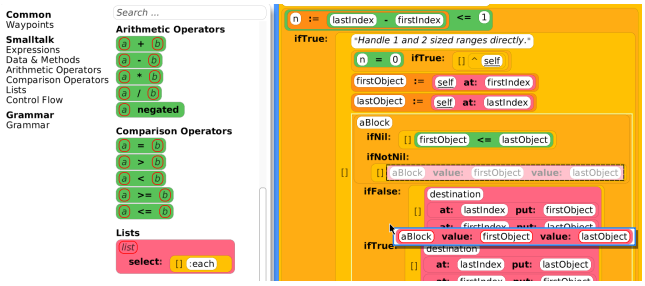

Figure 13. Screenshot of block palette and the excerpt of a method performing merge sort in a theme with larger insets and block colors chosen by semantic category of a method.

Scratch closely in terms of its interactions but extends the language model to be suitable for more complex use cases [14]. For example, it allows the user to define custom methods through blocks. Some concepts have been proposed to make editing in GP more efficient, such as a toggle that switches between the block and a text display of the source code [17]. Notably, the concept of a "block editing cursor" has also been demonstrated [17], which allows users to move between blocks with tab and the arrow keys, to delete blocks before the cursor using backspace, and to invoke an autocompletion menu when typing characters. A similar keyboard mode exists in Snap. In this paper, we provide a description of a block cursor concept that maps to an existing textual programming language, with its associated challenges of the grammar's complexity and users' prior familiarity, and evaluate the concept's effectiveness.

Syntax-directed interactions have been considered in various other projects. For example, a syntax-directed keyboard for touch devices [1] has been proposed that places buttons to create common syntactic structures specific to the language on the keyboard. TouchDevelop [22] is an application development framework with a programming interface optimized for touch devices that mixes text-based editing for expressions with a structured editor for statements to reduce the likelihood of mistakes. In Tylr [15], the user is able to modify structures orthogonal to the tree structure, granting more flexibility. Sandblocks [4] demonstrates an approach to map textual languages that generalizes across languages by targeting the grammar formalism instead of a specific language. As a result, quality of the editor is tied to the quality or decomposition of the grammar, whereas the approach described in this paper allows the editor creators to consider the most appropriate mapping from text to blocks from scratch.

In frame-based editing [10], each program scope receives a graphical box frame, while code at the expression level uses text. The context is designed to enable a smoother transition from block-based languages to textual languages. Editing through direct manipulation is still possible but not at the same granularity as in block-based editors. A study showed that an implementation of this concept called Stride had a positive effect on task completion time for programming novices when compared to a purely textual programming environment [20]. In the language editor implementation framework Barista [13], when a new character is inserted in a structure, the editor will convert the affected structure back to its original textual tokens, incorporate the new character, and reparse the tokens into structures. Frame-based editing and Barista both move the core interface model further away from blocks and towards text, while our block cursor concept aims to maintain the visual appearance and interactions of blocks where possible.

MPS is a language workbench for defining projectional languages. In the context of MPS, aspects that enable user-friendly projectional editors have been investigated [25]. The authors demonstrated that editing efficiency in MPS-based languages can surpass that of text editors using a concept called Grammar Cells [26]. Grammar Cells define restructuring rules that are triggered as users input characters of a language's textual syntax. Grammar Cells were an inspiration for the grammar-assisted input in our block cursor concept.

7 Conclusion

In this paper, we investigate the design of a block-based editor that can be integrated into and used as part of a text-based programming environment. For that, we presented the concept of a block cursor that enables interactions with blocks similar to interactions in a traditional text editor, offering experienced programmers the familiarity they are used to from their traditional environments. Participants of a user study (n=8) where able to comfortably use the editor.

Through our own extended use of our block-based editor, we identified important insights for its integration, such as autoformatting, clipboard support, or tooling. Our approach demonstrates a way for block-based editors to adapt to the needs of textual environments when needed.

Acknowledgments

We sincerely thank the anonymous reviewers for their detailed and valuable feedback. This work was supported by SAP and the HPI–MIT "Designing for Sustainability" research program[4].

References

[1] I. Almusaly, R. Metoyer, and C. Jensen. 2017. Syntax-directed keyboard extension: Evolution and evaluation. In *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 285–289. doi:10.1109/VLHCC.2017.8103480

[2] Babel. 2020. *@babel/parser AST node types.* https://web.archive.org/web/20210404141645/https://github.com/babel/babel/blob/master/packages/babel-parser/ast/spec.md

[4] https://hpi.de/en/research/cooperations-partners/research-program-designing-for-sustainability.html

[3] Tom Beckmann, Stefan Ramson, Patrick Rein, and Robert Hirschfeld. 2020. Visual design for a tree-oriented projectional editor. In *Companion Proceedings of the 4th International Conference on Art, Science, and Engineering of Programming (Programming '20)*. Association for Computing Machinery, New York, NY, USA, 113–119. doi:10.1145/3397537.3397560

[4] Tom Beckmann, Patrick Rein, Stefan Ramson, Joana Bergsiek, and Robert Hirschfeld. 2023. Structured Editing for All: Deriving Usable Structured Editors from Grammars. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems* (Hamburg, Germany) *(CHI '23)*. Association for Computing Machinery, New York, NY, USA, Article 595, 16 pages. doi:10.1145/3544548.3580785

[5] Alan F. Blackwell and Sam Aaron. 2015. Craft Practices of Live Coding Language Design, Alex McLean, Thor Magnusson, Kia Ng, Shelly Knotts, and Joanne Armitage (Eds.). *Proceedings of the First International Conference on Live Coding*.

[6] Andrew Bragdon, Robert Zeleznik, Steven P. Reiss, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, and Joseph J. LaViola. 2010. Code bubbles: a working set-based interface for code understanding and maintenance. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, Atlanta Georgia USA, 2503–2512. doi:10.1145/1753326.1753706

[7] Wayne Citrin, Richard S. Hall, and Benjamin G. Zorn. 1995. Addressing the Scalability Problem in Visual Programming ; CU-CS-768-95.

[8] Nicola Dell, Vidya Vaidyanathan, Indrani Medhi, Edward Cutrell, and William Thies. 2012. "Yours is better!": participant response bias in HCI. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Austin, Texas, USA) *(CHI '12)*. Association for Computing Machinery, New York, NY, USA, 1321–1330. doi:10.1145/2207676.2208589

[9] Brian Harvey and Jens Mönig. 2015. Lambda in blocks languages: Lessons learned. In *2015 IEEE Blocks and Beyond Workshop (Blocks and Beyond)*. 35–38. doi:10.1109/BLOCKS.2015.7368997

[10] Michelle Ichinco, Kyle Harms, and Caitlin Kelleher. 2017. Towards Understanding Successful Novice Example Use in Blocks-Based Programming. *Journal of Visual Languages and Sentient Systems* 3, 1 (July 2017), 101–118. doi:10.18293/vlss2017-012

[11] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. 1997. Back to the Future: The Story of Squeak, a Practical Smalltalk Written in Itself. *SIGPLAN Not.* 32, 10 (Oct. 1997), 318–326. doi:10.1145/263700.263754

[12] Amy J. Ko, Htet Htet Aung, and Brad A. Myers. 2005. Design requirements for more flexible structured editors from a study of programmers' text editing. In *Extended Abstracts Proceedings of the 2005 Conference on Human Factors in Computing Systems, CHI 2005, Portland, Oregon, USA, April 2-7, 2005*, Gerrit C. van der Veer and Carolyn Gale (Eds.). ACM, 1557–1560. doi:10.1145/1056808.1056965

[13] Amy J. Ko and Brad A. Myers. 2006. Barista: An implementation framework for enabling new tools, interaction techniques and views in code editors. In *Proceedings of the 2006 Conference on Human Factors in Computing Systems, CHI 2006, Montréal, Québec, Canada, April 22-27, 2006*, Rebecca E. Grinter, Tom Rodden, Paul M. Aoki, Edward Cutrell, Robin Jeffries, and Gary M. Olson (Eds.). ACM, 387–396. doi:10.1145/1124772.1124831

[14] John Maloney, Michael Nagle, and Jens Mönig. 2017. GP: A General Purpose Blocks-Based Language (Abstract Only). In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education, Seattle, WA, USA, March 8-11, 2017*, Michael E. Caspersen, Stephen H. Edwards, Tiffany Barnes, and Daniel D. Garcia (Eds.). ACM, 739. doi:10.1145/3017680.3017825

[15] David Moon, Andrew Blinn, and Cyrus Omar. 2023. Gradual Structure Editing with Obligations. In *2023 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, Washington, DC, USA, 71–81. doi:10.1109/vl-hcc57772.2023.00016

[16] Brad A. Myers. 1990. Taxonomies of visual programming and program visualization. *J. Vis. Lang. Comput.* 1, 1 (1990), 97–123. doi:10.1016/S1045-926X(05)80036-9

[17] Jens Mönig, Yoshiki Ohshima, and John Maloney. 2015. Blocks at your fingertips: Blurring the line between blocks and text in GP. In *2015 IEEE Blocks and Beyond Workshop (Blocks and Beyond)*. 51–53.

[18] Bonnie A. Nardi. 1993. *A Small Matter of Programming: Perspectives on End User Computing*. MIT Press, Cambridge, MA, USA.

[19] Donald A. Norman. 1983. Design Rules Based on Analyses of Human Error. *Commun. ACM* 26, 4 (1983), 254–258. doi:10.1145/2163.358092

[20] Thomas W. Price, Neil C. C. Brown, Dragan Lipovac, Tiffany Barnes, and Michael Kölling. 2016. Evaluation of a Frame-based Programming Editor. In *Proceedings of the 2016 ACM Conference on International Computing Education Research, ICER 2016, Melbourne, VIC, Australia, September 8-12, 2016*, Judy Sheard, Josh Tenenberg, Donald Chinn, and Brian Dorn (Eds.). ACM, 33–42. doi:10.1145/2960310.2960319

[21] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay S. Silver, Brian Silverman, and Yasmin B. Kafai. 2009. Scratch: programming for all. *Commun. ACM* 52, 11 (2009), 60–67. doi:10.1145/1592761.1592779

[22] Nikolai Tillmann, Michal Moskal, Jonathan de Halleux, and Manuel Fahndrich. 2011. TouchDevelop: Programming Cloud-Connected Mobile Devices via Touchscreen. In *Proceedings of the 10th SIGPLAN Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (Portland, Oregon, USA) *(Onward! 2011)*. Association for Computing Machinery, New York, NY, USA, 49–60. doi:10.1145/2048237.2048245

[23] Mauricio Verano Merino, Tom Beckmann, Tijs Van Der Storm, Robert Hirschfeld, and Jurgen J. Vinju. 2021. Getting grammars into shape for block-based editors. In *Proceedings of the 14th ACM SIGPLAN International Conference on Software Language Engineering*. ACM, Chicago IL USA, 83–98. doi:10.1145/3486608.3486908

[24] Mauricio Verano Merino and Tijs van der Storm. 2020. Block-Based Syntax from Context-Free Grammars. In *SLE 2020 - Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering, Co-located with SPLASH 2020*, Ralf Lammel, Laurence Tratt, and Juan de Lara (Eds.). ACM/IEEE, 283–295. doi:10.1145/3426425.3426948

[25] Markus Voelter, Janet Siegmund, Thorsten Berger, and Bernd Kolb. 2014. Towards User-Friendly Projectional Editors. In *Software Language Engineering - 7th International Conference, SLE 2014, Västerås, Sweden, September 15-16, 2014. Proceedings (Lecture Notes in Computer Science, Vol. 8706)*, Benoît Combemale, David J. Pearce, Olivier Barais, and Jurgen J. Vinju (Eds.). Springer, 41–61. doi:10.1007/978-3-319-11245-9_3

[26] Markus Voelter, Tamás Szabó, Sascha Lisson, Bernd Kolb, Sebastian Erdweg, and Thorsten Berger. 2016. Efficient development of consistent projectional editors using grammar cells. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering, Amsterdam, The Netherlands, October 31 - November 1, 2016*, Tijs van der Storm, Emilie Balland, and Dániel Varró (Eds.). ACM, 28–40. http://dl.acm.org/citation.cfm?id=2997365

[27] Philip Wadler. 2003. A prettier printer. *The Fun of Programming, Cornerstones of Computing* (2003), 223–243.

Received 2025-07-09; accepted 2025-08-11