Structured Editing for All: Deriving Usable Structured Editors from Grammars

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

Patrick Rein
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
patrick.rein@hpi.uni-potsdam.de

Stefan Ramson
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
stefan.ramson@hpi.uni-potsdam.de

Joana Bergsiek
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
joana.bergsiek@student.hpi.uni-potsdam.de

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

ABSTRACT

Structured editing can show benefits in learnability, tool building, and editing efficiency in programming. However, creating a usable structured editor is laborious and demanding, typically requiring tool builders to manually create or adjust editing interactions.

We present Sandblocks, a system that allows users to automatically generate structured editors for every language with a formal grammar available. Our system's *input reconciliation process* acts on arbitrary syntax trees to provides consistent interactions across our generated editors. Our editors' editing experience is designed to be familiar to users from textual editing but, compared to previous work, requires no manual annotation in the grammars.

We demonstrate our editors' usability across languages through a user study (N=18). Compared to conventional text editors, even with minimal training, participants only took on average 21% (JS), 34% (Clojure), and 95% (RegExp) longer and reported that editing felt natural with a score of 6/7.

CCS CONCEPTS

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

KEYWORDS

structured editing, grammars, text-like editing

ACM Reference Format:
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 (CHI '23), April 23–28, 2023, Hamburg, Germany*. ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3544548.3580785

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
CHI '23, April 23–28, 2023, Hamburg, Germany
© 2023 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9421-5/23/04.
https://doi.org/10.1145/3544548.3580785

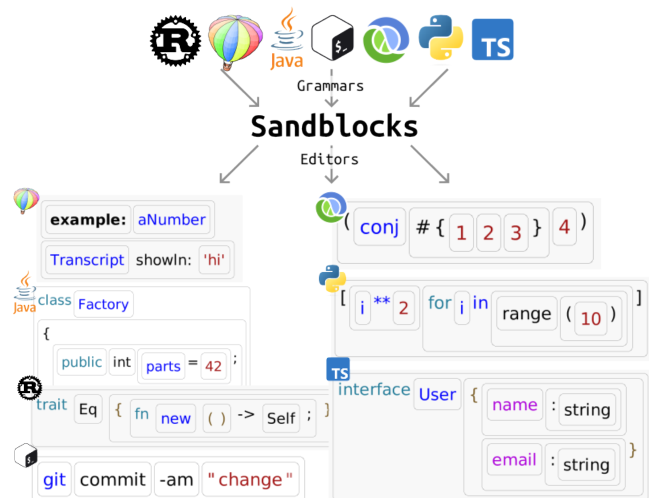

Figure 1: Sandblocks takes any grammars expressed using Tree-sitter and automatically generates usable structured editors. Shown are snippets from some languages we tested.

1 INTRODUCTION

For programming tools, structured editors offer unique advantages over text editors, as they maintain a valid tree of the program at all times and model editing operations explicitly as transformations over that tree. As a result, structured editors can offer users syntactic guidance while editing [14, 29], simplify the composition of languages [37], or enable novel tools to provide features that rely on structural guarantees and would thus be considerably more difficult to realize in text editors [7, 8, 20, 21, 28].

However, structured editors in prior work are caught in a conflict between their *usability* and their *availability* across languages,

which we believe hindered their widespread adoption. In contrast, text editing provides a single, sufficiently usable editing metaphor with consistent interactions across all (textual) languages.

Structured editors with great *usability* have not been as widely *available*. Structured editors with a familiar and efficient editing interface have been presented and are in active use [20, 38, 39]. However, prior work relies on manual annotations for each new language to get the structured editor to behave consistently and predictably. As a consequence, structured editors tend to be *available* only for domain-specific languages [9, 32], subsets of languages [30], or for languages that are particularly suitable for structured editing, such as Lisp-like languages.

On the flip side, structured editors that have been widely *available* did not have great *usability*. Generating structured editors starting from a grammar is possible, thus making in principle *all* textual languages *available* for use with a structured editor at no additional cost [23, 36]. However, the generated editors suffer from high viscosity [4] for common editing interactions [35] thus impacting *usability* in terms of their efficiency. Interactions in generated editors commonly include generic, menu-based tree manipulation operations or drag-and-drop interfaces, such as in block-based editors [14, 29]. While block-based editors are great in an educational context, their reliance on drag-and-drop means that interactions are considerably slower than with a keyboard.

However, this conflict between *usability* and *availability* is not a fundamental issue: In this paper, we present a practical approach to resolve the conflict by defining a set of heuristics for editing that act on the level of the grammar operators, rather than any specific language. As a result, our generated editors in our environment, *Sandblocks*, are available across languages and provide a single, consistent, and usable editing metaphor for all languages, similar to text editors. Because of the unified, familiar editing interface, users were able to edit not only domain-specific languages, but also complex general-purpose languages in our user study. More concretely, similar to prior work, we generate structured editors from grammars of textual programming languages. But, in addition, we propose the concept of an *input reconciliation process* that, continuously on every edit, consults the grammar to reconcile input in a manner that users would expect from a text editor.

In our evaluation, we demonstrate four properties of editors running our input reconciliation process: (1) interactions appear familiar to users, (2) learned interactions transfer between editors for different languages, (3) interactions are efficient, and (4) users are able to enter any desired language constructs. We find that editing feels natural to users coming from textual editing; that performing editing tasks in various languages was possible for our users with no additional instructions; and that editing efficiency decreases only slightly compared to our baseline of textual editing.

With this new possibility of generating usable structured editors for even complex languages such as Rust, TypeScript, or Regular Expressions (see Figure 1) at no additional cost, we hope to enable a wider audience to enjoy the benefits of structured editing.

Our reference implementation for the described concepts, Sandblocks, is available open-source on Github[1].

[1] https://github.com/hpi-swa-lab/sb-tree-sitter/

2 RELATED WORK

In this section, we give an overview of works that investigated the usability of structured editors, as well as frameworks designed for the (automatic) generation of structured editors.

2.1 Usability in Structured Editors

In early structured editors such as the Cornell Program Synthesizer [33] interactions tended to be menu-based, requiring users to select language constructs they wanted to enter via dedicated user interfaces. Of these, Gandalf [27] is one of the earliest projects to automatically generate structured editors. It defined its own language for specifying interactions and views within the editor.

The GNOME [11] and MacGnome [24] projects took inspiration from these early projects to design a structured editor for programming education. Where GNOME was menu-based, MacGnome extended interactions with mouse navigation and temporary conversion of edited nodes to text to support code transformations. MacGnome utilizes a parser for editing but awaits complete inputs before committing intermediary text to structures, unlike our system which continuously gives users feedback. For both GNOME and MacGnome the authors described that they manually added numerous special cases over the lifetime of the project to try and address usability concerns [24].

A number of structured editors, such as Lamdu [21] or Hazel [28], employ a set of hand-written heuristics to provide users with a text-like editing experience for their editor's language. More generally, authors have studied the constraints that influence usability concerns in structured editing [18, 39], identifying that users tend to expect or benefit from a text-like editing experience.

More closely related, GrammarCells for MPS [38] aims to address the issue of inconsistencies and implementation effort when integrating text-like editing with MPS languages, by providing high-level constructs that ease and unify this integration. The resulting editing experience allows users to perform most editing operations as if working in a text editor. In a user study, GrammarCells demonstrated that it allows experienced users to surpass efficiency of text editing in MPS [3]. For example, changing an operation from addition to assignment is made possible by linearizing and reparsing subtrees; or deleting a qualifier from a declaration by hitting backspace is possible where otherwise a menu action may have been used. As such, the resulting interactions are resembling those of textual editing while syntax errors are contained in subtrees, similar to our proposed system. Unlike our system where editing interactions are derived from the language definition, language authors manually specify editor hints to inform the system of the editing behavior thus requiring considerations for both the language definition and the editing definitions.

Barista [20] supports an editing experience similar to textual editing for languages expressed in a custom grammar; the example implementation demonstrates this for Java. When the user performs a change, Barista first tries to apply the change inside the selected node, similar to our input reconciliation process as described in detail in subsection 3.2. If the change cannot be applied it performs what we call stringification in our process: the subtree is linearized, the change inserted, and the linear stream of tokens attempted to be reparsed. If reparsing is not possible, the now invalid tokens are

contained in an error production. As a result, the editing experience in Barista as we experienced it, places the user frequently in an intermediate error state, whereas our input reconciliation process typically brings users to a valid state earlier since it attempts different, escalating strategies for applying changes. The impact of this difference is difficult to assess, as the paper does not describe a user study or evaluation. Notably, however, similar to GrammarCells, Barista uses a specifically designed language [19] to map from a parse tree to an interactive view but does not do so automatically.

An alternative approach that sacrifices most of the benefits expected by structured editing is to add structural editing interactions on top of a text editor. For example, Vim or Emacs, in particular Emacs' ParEdit mode[2], provide text objects, often akin to syntactic constructs, that users can address as part of shortcuts. Deuce [15] augments a text editor by providing structural selection and also integrates guided editing options via a menu that allows users to perform predefined operations on their selection. The goal here is typically efficiency or convenience. As the underlying system is still a text editor, benefits such as tool integration or syntactically-guided editing cannot be gained or only approximated through a parser.

In the space of education, structured editors that focus on drag-and-drop without considerations for textual editing are prevalent, often called block-based editors, for example Scratch [29] or Snap [14]. Here, users move and assemble composed language constructs using their mouse. The keyboard is typically only used to type numbers or names. To form a bridge between block-based editors and text editors, frame-based editing [16] displays code in draggable structures on the statement level but allows for text editing for expressions. While this compromise is helpful, in particular for imperative languages such as Java, languages that focus more heavily on expressions, such as Clojure, see less benefit.

2.2 Generating Structured Editors

A number of dedicated language creation frameworks or toolkits exist. In Proxima [31], Harmonia [5], or MPS [37] authors define languages that the framework then turns into structured editors. They differ in their approach but neither is necessarily designed to enable reuse of existing general-purpose language grammars.

Language workbenches, such as Rascal [34], support authors in creating domain-specific languages, for example by deriving often-used tools such as debuggers or autocompletion. Rascal2MPS [23] allows authors to take a language definition in the Rascal language workbench and translate it to a projectional editor in MPS. The approach requires some manual steps and the authors note that usability is likely limited, pending analysis through a user study.

Kogi [36] and S/Kogi [35] similarly take as input a Rascal or Ohm language definition and translate it to a Blocky [12] block-based editor. S/Kogi preprocesses the grammar to arrive at a version that generates block-based editors that the authors describe as more usable, as indicated by fewer blocks required to formulate a program. Their preprocessing steps inspired the steps we describe in subsection 4.1. However, unlike the editors generated for S/Kogi, which the authors stated were only feasible for small to medium-sized domain-specific languages, our system is designed to work with general-purpose programming languages.

3 EDITING IN AN AUTOMATICALLY GENERATED STRUCTURED EDITOR

The primary design goals of our system are to provide editing interactions that feel familiar and efficient to users coming from textual editing and to work consistently on syntax trees of arbitrary languages. As such, even though the user's edits are at all times constrained by the grammar, we need to support the sequences of input users are used to and would expect from entering expressions in a textual editor. Figure 2 demonstrates this behavior in a walkthrough for a refactoring in a TypeScript editor generated by our system, where the user wants to extract an expression to a new variable. We chose TypeScript as an example language with moderately complex syntax.

First, frames (a) through (b) show that the navigation and copying behavior in our editor is equivalent to textual editors, with the exception of structural selection. In frame (a), the user's cursor is on the `setQuaternionFrom` property. The user moves the cursor word-wise until it is inside the expression to be extracted, uses the `enlarge selection` shortcut to select the entire expression, and copies it to the clipboard (b).

Frames (c) through (l) demonstrate the editing behavior, again equivalent to textual editing, but guided by the structural properties of the tree: with the selection of the expression still active, the user begins typing the name of the desired variable (c), which replaces the selection as it would in a text editor. Using Shift+Enter, the user adds a new statement above the current statement (d). The user begins typing the keyword `let`, which initially is ambiguous according to the grammar, as `le` could also be an identifier. Correspondingly, our system displays a popup where the user can explicitly disambiguate the input. Instead, the user continues typing and finally hits space (f), at which time the input is unambiguous and thus expanded to a declaration. The user continues typing the name of the variable and the equal sign (g), then pastes the copied expression (h).

Our editor permits intermediate stages of incomplete syntax trees to allow the user to change even non-structural properties of subtrees. In our scenario, the user realizes that a constant declaration would be better suited here, thus jumps to the start of the statement, and clears the `let` keyword (i), which our editor allows but remembers the fact that this text field is meant to hold the `let` keyword. As soon as the user hits the `c` character in the now empty text field, the declaration is adapted, keeping the still compatible subtree to the right of the keyword. Finally, the user navigates to the identifier, presses the colon character (k), and adds a type declaration (l).

The walkthrough demonstrates that our system is capable of interpreting user input in a manner that should appear familiar to users coming from textual editing in popular editors such as Visual Studio or IntelliJ IDEA. The rules driving these interactions are defined by our *input reconciliation process* that interfaces with the Tree-sitter TypeScript grammar; the interactions are defined for arbitrary syntax trees and thus work consistently in editors

[2] https://www.emacswiki.org/emacs/ParEdit

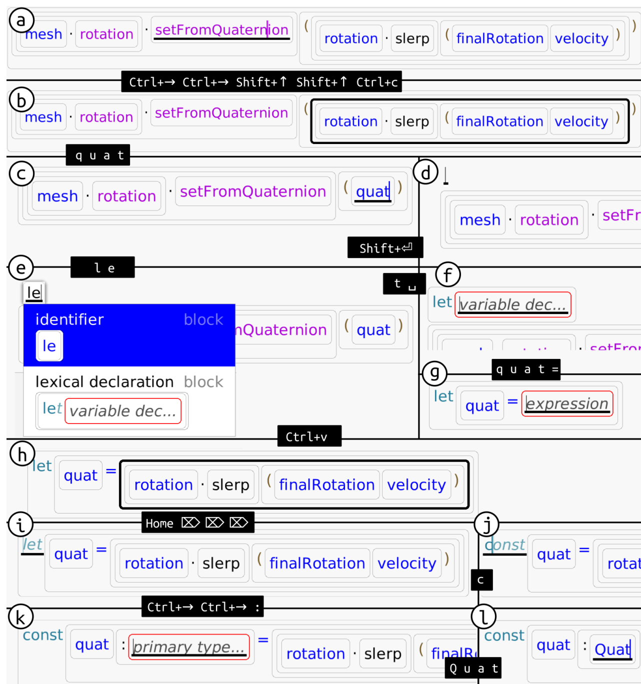

Figure 2: The user extracts an expression into a new variable by copying it, creating a new assignment statement, and pasting the expression (a-h), changes the declaration from `let` to `const` (i-j), and adds a type to the declaration (k-l). Shown in black boxes are the keypresses from the previous to the adjacent frame.

generated for any other language. Three central heuristics guide the design of the input reconciliation process to provide users with predictable, familiar editing interactions that resemble those of textual editing:

(1) *Navigation follows the visual layout of text fields and blocks, rather than the tree structure.*
(2) *Character entry by the user is treated as if injected in the string representation of the tree and buffers input until the user's intent appears unambiguous.*
(3) *Deletion acts conservatively, requiring explicit action before deleting structures even when empty.*

In the following, we will describe how these heuristics are realized in our system to act on arbitrary syntax trees.

3.1 Navigation

Our editor creates user interface (UI) elements from syntax trees, mapping each non-terminal node in the tree to a block with an outline and each terminal node to a text field. Text fields and blocks store references back to the grammar operator that created them.

To support keyboard-driven navigation and editing in the editor, text fields and blocks can be selected via a *block cursor*. This block cursor can be in one of three types of positions, as illustrated in Figure 3:

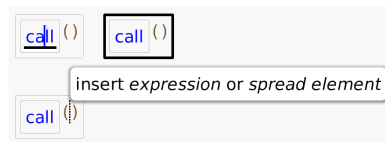

Figure 3: The three types of cursor positions. In the top-left, the cursor is inside a text field, in a text position. In the top-right, the cursor selects the entire visible subtree of the call expression. Finally, at the bottom, the cursor is in an insert position. A hint is showing what types of grammar rules can be instantiated at the selected position.

- In a *text* position, the cursor is inside a specific text field at a specific caret position within that text field's string contents.
- In a *select* position, the cursor selects an entire block. This allows the user to move the selection up and down the hierarchy of the syntax tree.
- In an *insert* position, the cursor is at a position between blocks where the grammar allows repetition or optional elements. Here, users may begin typing to create a corresponding element.

As per the previously established heuristic, the cursor moves along the cardinal directions visually. During pilots, we realized that the majority of users preferred this behavior rather than for example following the tree structure. This observation is consistent with previous work on structure editors [24]. Correspondingly, via the left and right arrows, the cursor moves through the leaf nodes of the tree of text fields and blocks, stopping in all caret positions of text fields and all insert positions between blocks. Via the Control modifier, the cursor moves block-wise, also skipping insert positions. To move to a select position, users can press the `enlarge selection` shortcut, Shift+Up arrow, mimicking the typical Shift+Arrow keys selection in text editors. The up and down arrows simulate the behavior of navigating vertically in a text editor. First, we collect all text fields that are below or above our selection, respectively. We then select the text field whose bounds rectangle has the smallest Euclidian distance to the top or bottom tip of our cursor from the closest point along any of its edges, thus typically jumping to a text field just above or below our current one. In addition, as the tree is constantly restructured as the user is typing as part of the input reconciliation process for character entry or deletion, we ensure that the positioning of the cursor after a restructuring allows the user to continue typing as they normally would in a text editor. We do so by finding the last character the user actually typed themselves, placing the cursor just after when a restructuring occurred.

3.2 Character Entry

A modified parser, which we call a *partial parser* [2] informs our reconciliation process of how users' changes can be made compatible with the language grammar. Tree-sitter's default parser supports incremental parsing with error recovery, which is different from partial parsing. Incremental parsing speeds up subsequent parses by reusing unchanged parts of the parse tree; error recovery contains errors in the input to the smallest area possible, sometimes at

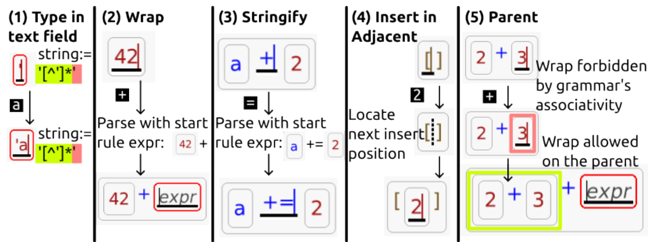

Figure 4: The five steps of our input reconciliation process for character entry. (1) The user has typed a letter in a string. While the string is not valid until the user has entered the quotation mark, the prefix of the regex still matches and the edit is thus allowed. (2) The plus is not allowed inside the number, thus the system places a query to replace the number with a wrapped expression. (3) An equal sign is entered; the binary operator cannot incorporate it, however by "stringifying" the operator's contents and re-parsing, we find that an augmented assignment operator can incorporate the full string. (4) A number is entered that cannot be added to the array delimiter. The system searches for subsequent positions that would take it and creates a number inside the array. (5) The user enters a plus sign at the right edge of the expression. We cannot wrap the number as in scenario (2) because addition was defined to be left-associative, so we try again on the parent and find that we can apply the wrap (2) operation.

the expense of correctness. While related, these features are not fit for our need of exhaustively listing options in which partial input could be turned into valid subtrees [2].

Partial parsing yields a superset of an ordinary CFG parser. A partial parser accepts input even when only a prefix of a rule matches. Prefix matches are autocompleted by creating placeholder nodes. In addition, we can *query* the partial parser giving not only a string to be parsed but a mix of characters and existing nodes. In case of a partial parse tree, we interpret the placeholder nodes as the locations where users can continue typing to create a valid syntactical structure. For example, if the user provides `2+` as input, a partial parser will produce a valid parse tree where the missing right-hand operand will be filled by a hole, a block where the user can continue typing. The partial parser is realized by reimplementing all grammar operators of the Tree-sitter grammar language (i.e. sequences, repeats, regexes, ...) and modifying the sequence and regex operators to not abort once the input stream is exhausted but instead autocompleting with placeholders [2].

As part of the input reconciliation process for character entry, the partial parser is typically queried multiple times to check if the given input is compatible with the place the query is probing. Following the heuristic that character entry should be handled as if inserted in a textual source code string, we seek the closest position to the cursor in the syntax tree that accepts the user's input. To do so, we attempt each of the following steps until the first succeeds, as illustrated in Figure 4:

(1) type in text field,
(2) wrap selection,
(3) stringify selection,
(4) insert in adjacent, and
(5) recurse on parent.

Type in text field. In the simplest case, we can insert the input into the currently focused text field at the cursor position and the resulting string still conforms to the text field's regular expression, as derived from the Tree-sitter grammar. More specifically, we require only a prefix match of the regex. For example, a regular expression specifying a string (ignoring escaping) may take the form `/'[\^']*'/`. An input such as `'a` does not conform to the full regex but should still be accepted by the text field as it would commonly occur when the user is in the process of initially typing the string.

Wrap selection. Otherwise, if the cursor was at the very start or end of the selected text field, we attempt to wrap the containing block (hereafter named selected block). For this, we place the selected block in the input stream that we pass to the partial parser and insert the user's input characters before or after the element in the stream, depending on the cursor position. As the parent grammar operator for our query to the partial parser, we specify the selected block's own parent grammar operator, as we expect the result of the partial parser to take the selected element's place. For example, this allows replacing the expression `2` with an expression of the form `2+_` by typing a plus sign at the end of the number expression.

Stringify selection. If wrapping did not succeed or the cursor was not at the start or end of the text field, we attempt to "stringify" the contents of the selected text field and insert the user's input. Unlike the previous two operations, this allows the selected text field or block to be reconstructed with an entirely new parent grammar operator and based on a different grammar rule. For example, in Python there are two different grammar rules for integers and floats. As such, to turn an integer into a float, we take the literal string contents of an integer block's text field, append the dot character and ask the partial parser for matching results, which will produce a block for the float rule.

Insert in adjacent. If none of the steps above yielded results, we check for adjacent cursor positions that would place the cursor either in an insert position or in a text position for a text field that is currently empty. Both cases are illustrated in Figure 5. The first case, insert positions, occurs for example for JavaScript variable declarations. A simplified version of the rule is expressed through the form:

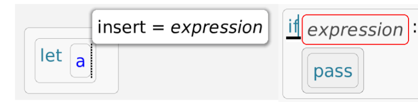

Figure 5: The two types of adjacent positions: in the left scenario, in JavaScript, typing an equal sign would expand the text fields and blocks for the optional assignment that is hinted at in the popup. In the right scenario, in Python, typing a letter would forward the input into the expression hole.

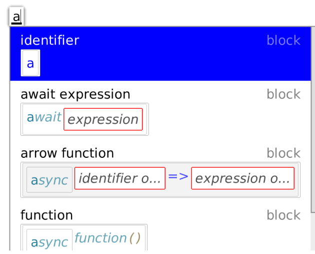

Figure 6: The user typed the character `a`, which in the context of a TypeScript expression is ambiguous. A popup offers the user to select an option and thus explicitly disambiguate. Alternatively, the user may continue typing until the desired construct is unambiguous.

```
declaration := "let" identifier ("=" expression)?
```

When first typing the `let` keyword, only the keyword and identifier are constructed as the assignment is optional. If the user then types an equal sign at the end of the identifier neither of the first three steps above will produce results. Instead, when checking its adjacent insert positions, we find the optional elements after the identifier. By passing the user's input into our partial parser and taking `("=" expression)` as the desired parent grammar operator, we find that this position indeed conforms to the user's input. The second case, an empty text field, most often occurs when we expand larger constructs. For example, if the user places their cursor in the keyword of an if-conditional in Python, where the condition has not been filled out yet, entering the letter `a` is again not a valid input for the `if` keyword, but the adjacent empty text field for the conditional can be turned into an identifier, consuming the input.

Recurse on parent. Finally, if neither of the above steps succeeded, we restart the entire process, this time taking the parent of the previously focussed text field or block as reference. So, when stringifying, wrapping, or looking for adjacent positions, we consider a larger subtree of the program. We abort unsuccessfully if this step is reached and the parent is the root of the program tree.

3.3 Modifying and Correcting Input

The above presents the linear flow of the process when users enter language constructs character-by-character, left-to-right, without error. Here, we describe how the input reconciliation process recovers from deviations from the happy path.

Ambiguity. For many inputs, there are multiple valid constructs that match according to our partial parser. For example, given just the input `a` with a parent grammar operator for an expression in TypeScript, we obtain among other options the identifier `a`, an `await _` expression, and an `async _ => _` function literal, as shown in Figure 6. In the presence of such ambiguity, we show a popup and offer the user to either directly select any of the given options, or to continue typing. In this example, if the user adds the letter `x` to the input, we immediately choose the identifier `ax` as option, as all other previous choices no longer consume all input.

The user can also trigger the popup manually through an explicit insert-before action (mapped to Shift+Space by default). This

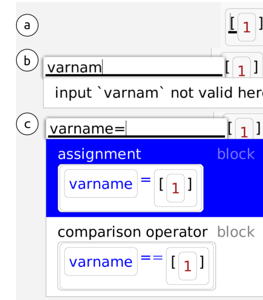

Figure 7: (a) The user selects the expression they want to wrap in an assignment. (b) They hit the insert-before-shortcut to open a popup and begin typing the declaration. (c) As they complete the expression, the partial parser offers two options to restructure the selected subtree, given the input.

Figure 8: The user typed the characters `x a` in a TypeScript editor using our system. In turn, our system autocompleted an as-expression for type casts. The user has not typed the `s` of the `as` keyword yet, as such it is displayed translucent. Users can now either type the missing `s` or navigate to the hole with the red outline directly.

somewhat unintuitive step is covered as part of the accompanying tutorial and is currently required to insert larger structures before the selection. As an example, the user may want to assign the array `[1]` to a variable, as demonstrated in Figure 7. In this case, they select the array, invoke the popup, and type `varname=`, at which point the partial parser can construct the assignment subtree for `varname=[1]`. The need for the popup arises as committing to an option as soon as possible can sometimes cause jarring transitions, as significant tokens to determine the desired language construct may only be entered late, as users are essentially filling a gap in the middle of a prefix and a suffix language construct. In the above example, we would immediately convert the structure into an attribute access of the form `v[1]`, only to later reinterpret it as assignment once the equal sign has been entered. In this instance, while jarring, the edit would work; if larger structures had been created in intermediary stages, the desired conversion may not have been possible anymore.

Autocompleted Tokens. One consideration for the aforementioned goal of supporting a typing flow that is identical to textual input concerns autocompleted tokens. For example, in TypeScript, users can perform an explicit cast using the infix operator `as`, e.g. `x as int`. However, since this is the only infix operator in TypeScript starting with the character `a`, our system immediately commits to this parse tree when the user presses the `a` key. As such, users in our pilots frequently ended up typing `x as sint` (note the double `s`), as the `s` was autocompleted but typing it was still in their muscle memory. As a solution, we still autocomplete the structures eagerly but leave all characters that the user did not explicitly type in a slightly translucent color and italic font, as shown in Figure 8. Users

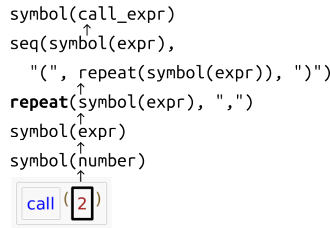

Figure 9: The block containing the number `2` and its parent grammar operators (other blocks shown for context). The grammar operators show that this block represents the `number` rule of the grammar. Specifically, it is a reference to that rule found inside a repeat operator (marked bold), which in turn is inside a sequence that is the body of a `call_expression`.

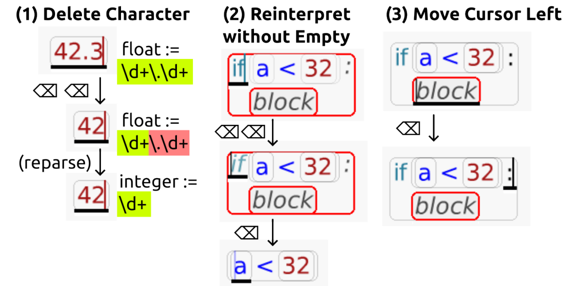

Figure 10: The three steps of our input reconciliation process for deletion. All three examples are based on the Python grammar. The later steps only occur if the previous were not applicable (refer to Figure 11 for a continuous example). In the first scenario, the user deletes the decimal part of a float via backspace. As a result, the regex for the float rule no longer matches and the block is turned into an integer. In the second scenario, the user deleted all but the comparison in the if-block (including the colon, which is set in italics to signal that it has been deleted but is syntactically still required here, as also seen in Figure 8). On the next backspace, the now-empty parts are removed and replaced by the only non-empty child, the comparison. In the third scenario, the user's cursor is in the empty block, which is required inside an if block per the grammar. As a result, when the user presses backspace, the cursor moves to the left and does not delete anything.

can now either just type the autocompleted letters or skip ahead by navigating, in which case we turn the characters into the regular font and style.

Whitespace. Most language grammars treat whitespace as filler tokens that may appear anywhere between tokens, a secondary notation designed to allow users to create logical groupings around the primary notation. In our design, we embraced the trend of pretty printers that streamline and auto-format all whitespace except deliberately placed empty lines. On language import, we modify the grammar's statement rule such that it may also allow empty lines and detect these when parsing files. Whitespace that is entered around tokens, for example when typing `a = 5`, is recognized but ignored by the partial parser, as it will not have an impact on the generated expression.

Errors. As users type expressions, they will inevitably produce typos. As these occur, two scenarios are possible: first, the input reconciliation process may not find a means to insert the given characters. In that case, the input will be buffered in the popup shown in Figure 7b and a notice is shown to the user. Second, the input reconciliation process will eagerly produce a language construct that was not desired. If that language construct autocompleted further tokens, such as an if-statement, users must use the undo shortcut to directly revert their change. Otherwise, they can just backspace to revert to a previous cursor position, as described in the next section.

3.4 Deletion

Explicitly invoked deletion occurs when the user presses either the delete or cut shortcuts. In this case, we traverse up the parents of the selected text field's or block's grammar operators to see if we find a repeat node, as illustrated in Figure 9. If so, and given that deletion will maintain the minimum number of elements for this repeat operator, we simply remove the text field or block from the tree. If no repeat operator is found, we instead turn the selected block into a hole, essentially clearing the contents of that subtree and allowing users to type new contents. Explicit deletion always acts on an entire block, if the cursor is in a text position or insert position, the containing block is targeted.

Implicit deletion occurs through backspacing and follows the heuristic stated in the introduction to this section to act conservatively. In our pilots the need to support backspacing to "undo" the construction of previously typed blocks became apparent: users were used to be able to correct mistakes by backspacing and retyping constructs from textual editing and were expecting the same functionality to work in our structured editor. At the same time, if users were correcting errors, jarring transitions from removal of elements as soon as they were empty caused users to question whether they did the right thing. As such, once elements are empty, we still wait for the user to press backspace one more time before proceeding to remove any elements.

Similar to the input reconciliation process for character entry described in subsection 3.2, the input reconciliation process for deletion successively tries to apply more local to more global changes, proceeding to the next step only if no previous step succeeded. Figure 10 illustrates the process described below:

(1) delete character,
(2) reinterpret without empty, and
(3) move cursor left.

Delete Character. If the selected text field is not empty, we just delete the character before the cursor. After deletion, we check if the string now contained in the text field matches the full regular expression of that text field, unlike in input reconciliation for character entry where we only check for the prefix of that regular expression. If it is not valid, we ask the partial parser to try and find a new subtree that would match the contents. Matching the full

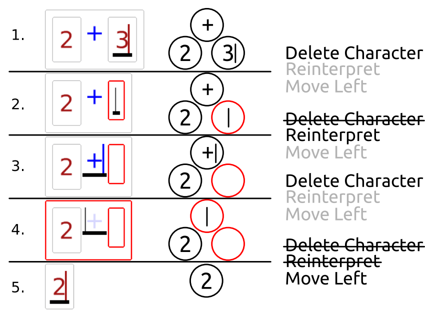

Figure 11: A step-by-step of an editing scenario. Shown are first the user interface, then the corresponding parse tree, and finally which step of the input reconciliation process applies to arrive at the next state. In scenario (1), the user would like to delete the binary operator and right operand, only keeping the `2` (5). Shown are the resulting text fields and blocks after pressing backspace, the syntax tree, and the steps of our input reconciliation process for deletion that apply. In (4), the plus sign has been deleted but the binary operator signals that it would have otherwise contained a plus sign. We do not immediately delete the operator at this point to allow users to easily change the plus into for example a minus.

regular expression is required to go from Python's float rule back to an integer rule, as an integer is a valid prefix of every float in Python and would thus not signal a need to re-parse the contents.

Reinterpret without Empty. Otherwise, if the text field is empty, we pass all non-empty children of the current selection's parent to the partial parser and see if a new element can be constructed using just those elements. In practice, this means that users can progressively empty larger elements until only the parts are left they want to keep.

Move Cursor left. If no deletion was possible in the previous steps, we move the cursor one position to the left. As a result, we skip over as-of-yet non-deletable parts of the program without users having to switch to navigation themselves.

As an example, assume the user placed the cursor at the very end of the expression `2+3` and wants to delete everything but the number `2`, as shown in Figure 11. As the user hits backspace, we first delete the character 3, as the cursor is in a non-empty text field. Reinterpreting the new contents as a different type can be omitted, as the text field is now empty. Next, when the user presses backspace again, there are no more characters to delete in this text field, so we skip to the second step. We then try to reinterpret the children of the binary addition but, in the TypeScript grammar, no other rules match the elements we still have. Thus, we skip to the third step, where we simply move the cursor left, into the plus character's text field. Again, this plus sign is deleted as per the first step. Finally, only after the next backspace in the already empty text field, we attempt to reinterpret the contents and find that the remaining number block can be used as a replacement for our binary addition.

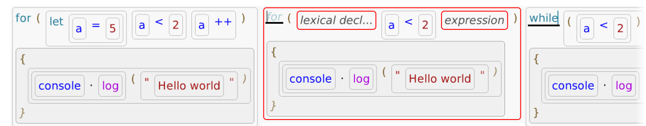

Figure 12: Changing a for-loop to be a while-loop. The user clears all fields they wish not to keep and changes the keyword. If instead, the user changes the keyword without first deleting the other fields, the text will appear in a popup informing the user that the input is not valid here, as shown in Figure 7. Refer to section 6 for a discussion of this behavior.

Modifying Elements. Modifications to elements follow the same heuristics as deletion. Assume that a user has typed a C-style for-loop and wants to change it to a C-style while-loop, as shown in Figure 12. We follow the heuristic to act conservatively: as the while-loop requires fewer blocks than the for-loop, just converting it would either yield an invalid tree or would require arbitrarily discarding elements that the user may want to keep. Instead, we impose on the user to perform conversions such as this one, where some blocks have to be discarded, to first empty or move those blocks they do not care about and only then trigger the conversion, in this case simply by changing the keyword. In this way, the structure of the tree is maintained at all times, the interactions resemble those of textual editing, with the exception of imposing an order to the operations. Modifications that do not discard blocks can be performed without additional consideration, for example changing an addition of the form `a+b` to an attribute access of the form `a.b` can be performed by replacing the plus sign with a dot.

4 FROM GRAMMARS TO STRUCTURED EDITORS

Our system takes arbitrary Tree-sitter grammars and derives user interface (UI) elements in the form of nested blocks. Figure 1 shows snippets in a number of different languages in our structured editor. Here, we briefly describe the import process and underlying data structure that informs our input reconciliation process.

4.1 Automatically Importing Grammars

The import process of the grammar into our system is fully automatic, with the exception of definitions of external rules and optional annotations, which we both describe further below. Users provide the URL to a Github repository containing the grammar definition and our system imports and generates the definitions as described below. We then preprocess the Tree-sitter grammar to condense it as much as possible, to reduce the number of blocks generated in the translation step while still maintaining flexibility during editing.

Tree-sitter grammars are composed of operators for labels, texts, choice, sequences, repeats, symbols, and precedence/associativity (and other, more specific ones, not relevant to this discussion). Tree-sitter grammars support left-associativity, which is commonly used for the expression rule in programming language grammars.

Note that for the purpose of producing subsequent partial parses this poses a particular challenge, as elements in the tree cannot be considered in isolation during parsing but rather have to be considered in the context of their parents in the tree. The partial parser may address this by reparsing multiple times, trying each left-associative parent of the current selection and listing all valid results [2].

The preprocessing steps are adapted from a similar method described in related work [35].

- Tree-sitter allows specifying that a rule should be inlined in a user-facing parse tree. We apply this inlining step as part of our preprocessing.
- We repeatedly apply a normalization step that, for example, merges nested repeats or turns `A+?` into `A*`, which occur as part of other transformations.
- As blocks are visually delineated through their appearance, we remove separators in lists, which renders editing less noisy. We employ three heuristics derived from Tree-sitter grammars we analyzed to detect separated lists.

As an example of such separated lists, given the rule

```
call_expr := sequence(expr, "(",
  repeat(sequence(expr, ",")), optional(expr), ")")
```

we detect the pattern that defines a delimited list and rewrite it as:

```
call_expr := sequence(expr, "(", repeat(expr, ","), ")")
```

In the object representation of the repeat operator, we remember that this repeat was delimited by the `","` character in the original grammar such that users can still type the delimiter if they want to.

External Rules. For cases where specifying a rule in Tree-sitter's DSL is either convoluted or simply not possible, Tree-sitter offers grammar authors to formulate an external scanner using C code. For example, bash's heredoc notation, which begins a literal string with user-defined delimiters, requires such an external scanner. If the imported grammar uses an external scanner, we require users to specify approximations in Tree-sitter's DSL of the rules that are specified in C. If the user does not provide them, the editor still works but the respective language constructs cannot be entered as our editor cannot infer the relevant tokens. Of the languages shown in Figure 1, Bash (heredoc, file descriptor, variable names, regexes), Python (string prefixes), and TypeScript (template string, automatic semicolon) each have at least one relevant external rule, while Smalltalk, Java, and Clojure have none.

Optional Annotations. There are three kinds of optional annotations we cannot derive automatically from the grammar that enhance the editor's functionality if specified during import. First, users can specify which repeat operators in the grammars contain what is considered statements in the imported language. With this information, we know what elements to insert when the Return key is pressed. Second, users can specify a statement terminator string, which is then removed from the grammar during preprocessing, similar to list separators. Third, users can specify which grammar operators signal soft or hard line breaks, as well as indentation, for the layouting engine. Without any hints, the layouting engine will place hard line breaks after statements, and soft line breaks in sequences and repeats, which tends to produce more soft line breaks than is desirable.

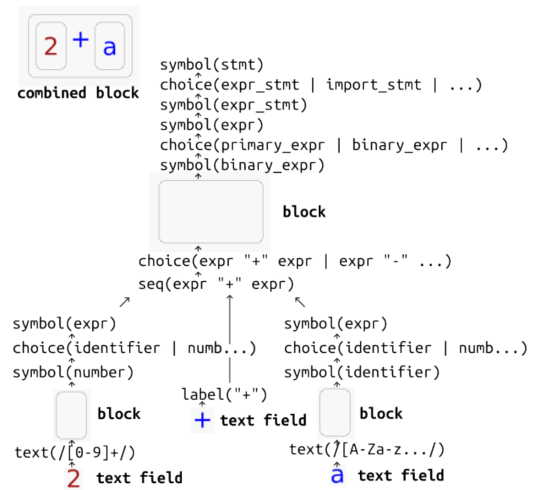

Figure 13: Blocks and text fields with their associated grammar operators for the TypeScript expression in the top-left, labeled "combined block". Every bold label marks a generated block or text field. All non-bold labels are operators in the grammar. Parent and child pointers are stored between all grammar operators, blocks, and text fields, as exemplified through the arrows. Text operators yield text fields, symbol operators yield blocks. Note that the `stmt`, `expr_stmt`, and `expr` symbols are marked as *supertypes* in this Tree-sitter grammar, meaning they act as container symbols for a number of other symbols and thus do not get their own block.

Applicability to Other Grammar Languages. Our approach and the described procedures are not limited to Tree-sitter. To our knowledge, Tree-sitter currently presents one of the largest, openly available repositories of language grammars within one language and was thus our choice for demonstrating the feasibility of the input reconciliation process. More generally, any grammar language with support for partial parsing can be used. As partial parsing is based on CFGs, any CFG-based language can be used [2]. The major consideration, besides coverage of languages, is the closeness of mapping between the expression of language constructs in the grammar language and the way users tend to think about them. If the grammar requires for example unrolling rules for precedence or introducing helper rules to express constructs, the resulting editors will display that incidental complexity to the user, too. Consequently, a grammar language that is similarly high-level as Tree-sitter's, such as the one used in Rascal [34], may be more suitable, than for example Parsing Expression Grammars [10], which, while easier to parse, tend to require grammar authors to depart further from the way languages are understood by users.

4.2 User Interface Data Structure

To go from a textual source file to our user interface, we invoke the Tree-sitter parser, which acts on the unmodified grammar. For each parse node in the parser's output, we traverse the grammar operators in their corresponding grammar rule and produce text

fields and blocks, making sure to map to our preprocessed version of the grammar in the process.

The layout and display of tree structure we chose to adopt is inspired by related work [1]. Symbol operators, which correspond to references to rules, get turned into blocks, as shown in Figure 13. Label and text operators get turned into text fields. The other operators only pass the request down to their child operators.

Text fields and blocks store a reference to the grammar operator that produced them. The grammar operators in turn store references to their parent and child operators as shown in Figure 13. We can thus traverse the stack of operators starting from each text field or block. Through this data structure, the partial parser can obtain information on what constraints from the grammar are imposed on the text fields and blocks that are passed in queries and text fields and blocks can determine whether they are in a valid state.

4.3 System Overview

To give a more complete impression of the system in which our implementation of the input reconciliation process is embedded in, we briefly discuss some aspects not relevant to the process itself.

Our system is implemented in Squeak/Smalltalk [17], based on the Morphic UI framework [22] and is publicly available on Github[3]. It is designed with a specific frontend in mind but can also act as a backend for other systems that wish to support textual editing inside a structured editor by providing our system with the new frontend's keyboard input and replaying the edits to the tree structure our system generates in the new frontend's user interface.

Next to textual interactions, which are at the heart of our input reconciliation process, our system supports mouse-based interactions. Users can click in text fields to place the block cursor. Users can also drag-and-drop blocks, thus acting as a block-based editor that also supports textual editing. Selection of multiple elements or cross-cutting selections only have rudimentary support at this point: users can multi-select via shift and left click but a partial selection of the textual contents is not currently possible. As with most editors, the appearance is configurable, both in terms of color but also in terms of the layout and size of blocks. The design presented here and evaluated in our user study favored larger blocks to increase clarity.

Our system integrates with the operating system's clipboard: when users copy a block, its textual representation is copied as well. If they paste a string and it successfully parses to a parse node at their cursor location, it is inserted; failures to parse are not currently handled. Users can open and edit textual files as they are used to from textual development environments. When saving, our editor follows a set of default heuristics that determines potentially required whitespace to output syntactically valid code (in particular: output whitespace if a letter or number is adjacent in two neighboring tokens, output newline for each statement) but users can also point our system to an external pretty printer for their language to be used.

If language authors specifically target our system as a frontend, they can extend the default set of shortcuts (cut/copy/paste) with custom shortcuts designed for their language. For example, an adaptation for the Clojure language may add the "slurp" and "barf" operations from ParEdit. In addition, our system can integrate with language servers to offer semantic editing operations, such as identifier autocompletion, refactoring actions, or resolving variable names. This is possible by storing and updating the text ranges of each parse node and using these in communication with the respective language servers.

[3]https://github.com/hpi-swa-lab/sb-tree-sitter/

5 EVALUATION

We evaluate our design along the following four research questions, which correspond to properties of our generated editors we described in section 1:

RQ1: Does working in our editor appear natural to users? A major goal of our design is to provide editing interactions that feel natural or familiar to users coming from textual editing. For this, we conducted a user study where we asked participants to perform a set of tasks and rate their experience. We aim to provide familiar editing interactions to reduce the cognitive load that users experience when using a structured editor for the first time, and thus reduce the entry barrier. Thus, we asked users to provide answers to the NASA TLX [13] to gauge the load experienced during the tasks.

RQ2: Do the interactions users learn transfer between editors for different languages? Our approach uses the same heuristics for all generated editors, independent of the language. Thereby, our editors should provide consistent editing interactions between all languages. The importance of consistency in interactions in structured editors has been pointed out by the GrammarCells project [38], noting that authors of structured editors typically have to define interactions each time they add support for a language anew. Contrarily, in text editors, interactions are always consistent between languages, as interactions do not need to be special-cased for entering specific language constructs; users always enter only characters or navigate the cursor.

To test this, all participants used Python as the same language during the tutorial and then worked with JavaScript, Clojure, and Regular Expressions in the study. In addition, the languages we selected for our study have varying syntactic characteristics.

RQ3: Is working in our editor efficient? In our editor, despite the heuristics, edits to the program have a higher complexity, as a change to the program does not only change the mere text but may influence the local structure of the program. Thus, we expect the editor to slow down programmers in their editing efficiency. A small slowdown would be acceptable, given the short training time the study setup permits and the projected advantages of structured editors in tool integration or support for learners. Nevertheless, our design aims to keep the slowdown small through familiar editing operations. To get an estimate of the actual impact of our design on the editing efficiency, we measured task completion times for each task in our editor and presented users with an equivalent task to be done in a text editor.

RQ4: Does our interaction design allow users to enter all desired structures? Finally, our input reconciliation system aims to interpret input sequences such that our system arrives at the same language constructs a textual parser would, given the same input. As an

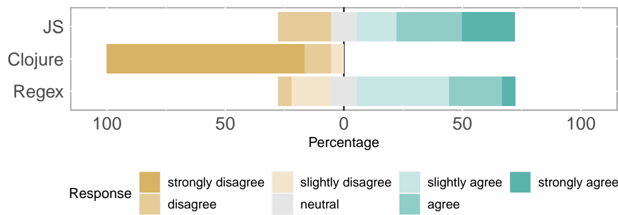

Figure 14: Participants reported whether they are familiar with the above languages. Median values are 5.5/7 for JavaScript, 1/7 for Clojure, and 5/7 for Regular Expressions.

approximation for evaluating this property, we sampled program subtrees of open-source projects in different languages, automatically re-typed them using our system, and evaluated whether our system reproduced the same program subtrees without mistake.

5.1 User Study

Our user study was designed to gauge the impact of the editing method (structured/textual) on the editing experience and performance of participants (RQ1, RQ3) and whether the learned interactions generalize across languages of different syntactic characteristics (RQ2).

Method. We designed our study as within-subjects. We recruited 18 participants, 17 male, 1 female; 2 professional programmers, and 5 PhD, 9 graduate, 2 undergraduate students between age 20 and 28; participants reported between 1 and 18 years of programming experience, of that between 0 and 6 in professional capacity. We first collected demographics, programming experience, and familiarity with the programming languages used in the experiment (shown in Figure 14) through a survey. Participants then spent around 5 minutes on an interactive tutorial for our editor, focused on teaching the mindset of typing code as one is used to and structural selection.

Next, participants performed 8 tasks in total; the first two were in Python and designed for warmup, to verify that the objective was well understood and is thus not reported in the results below. Of the other six tasks two were in JavaScript, two were in Clojure, and two were in ECMAScript Regular Expressions, embedded in JavaScript. This was to cover a wide spectrum of languages, from C-like, to Lisp-like, to an extremely concise, declarative language in the case of Regular Expressions. Both the tutorial and the two warmup tasks were using Python to not provide any learning effect. Participants did a task for all three languages in both our editor and a text editor. We counterbalanced the order of the tasks and editors through latin square to lessen learning effects; the order in which tasks and editor assignments would appear was thus different for each participant. For the text editor, we chose a subset of CodeMirror, a popular web-based code editor, that provides similar conveniences as our editor, matching the standard configuration of current popular editors such as Visual Studio Code: in particular, closing parentheses were automatically inserted, users could press parentheses on a selection to wrap the selection, standard shortcuts for cut, copy, paste and syntax highlighting were enabled for the respective languages, as shown in Figure 15. Identifier autocompletion was disabled in both

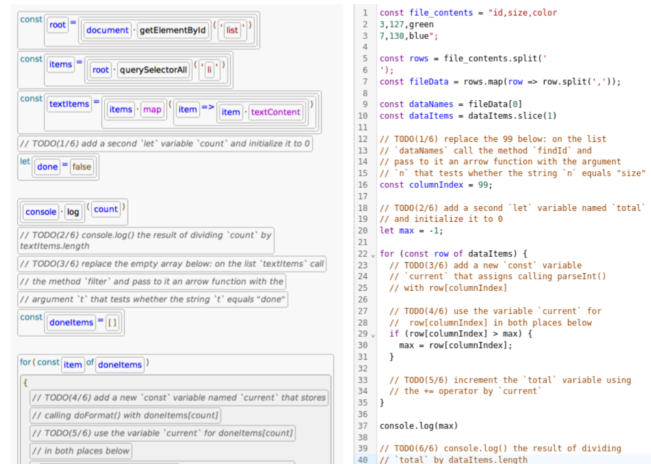

Figure 15: Screenshots of our editor (left) and the text editor (right) as presented to participants during the experiment for the JavaScript task. Note that we show excerpts of both permutations, so identifiers and order of tasks differ.

editors and support for multiple cursors was disabled in CodeMirror. Throughout the tasks, we recorded all keypresses and clicks.

After performing each task, we asked participants to fill out the six values of the NASA TLX [13]. After all tasks were completed, we asked participants to rate how natural text editing, deleting, and navigation felt in our system and performed a semi-structured interview to gain qualitative insights beyond our observations during participants' use of the system.

Tasks. For each task, we gave participants small programs of 13 (RegExp), 47 (JavaScript), and 21 (Clojure) lines of code. Each program had several TODOs inlined as comments that expressed a code change in natural language, as seen in Figure 15. Before participants started work on the task, we allowed them to read every desired change and to ask for clarification, e.g. if they did not know how to formulate a requested language construct. This was done to single out the time for editing, without the noise from strategic considerations while editing. To maintain some level of ecological validity, we nevertheless used composed editing tasks, consisting of adding multiple program elements across a number of code locations. Once participants went through all TODOs, they pressed a start button, performed the changes, and pressed a stop button. The time we report is the interval between the first and last keypress users performed while the task was active, thus excluding the time needed to move the cursor to the start/stop button. For each language, the two tasks in the two different editors required users to type the same language constructs using the same number of keystrokes but using different identifiers and in a different order, to reduce a learning effect between tasks, even if the tasks were otherwise identical in structure in order to be comparable.

RQ1: Editing Experience. Through observation by the instructors through screenshare, we determined that participants were able to complete all tasks in both our editor and the text editor. In some runs, participants accidentally omitted tokens such as closing

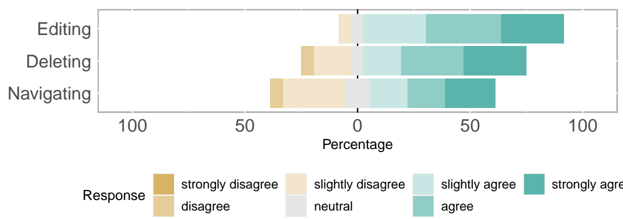

Figure 16: Participants rated whether they agree that text editing, deleting, and navigating felt natural in our system after completing all tasks. A majority tend to agree with each statement; deleting and navigating score lower, with two participants stating disagreement in either. Median values are 6 for editing and deleting and 5 for navigating.

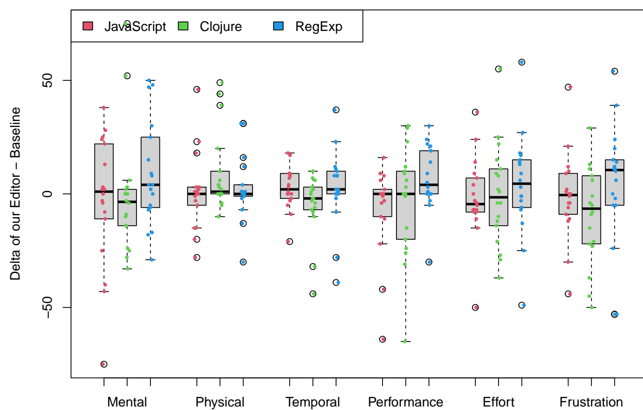

Figure 17: Boxplot of delta between raw values of TLX per language in our editor minus the text editor. Values below 0 indicate a load that was lower in our editor, values above 0 a load that was lower in the text editor.

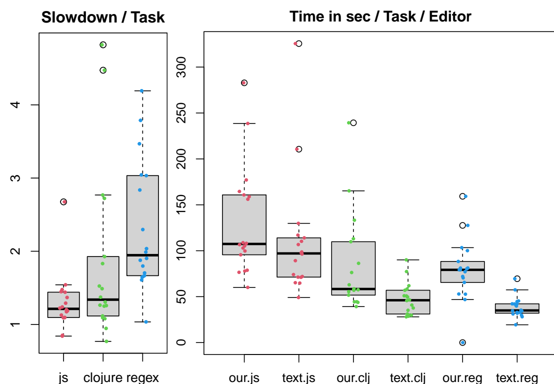

Figure 18: On the left: Boxplot of slowdown of task completion times per language. A factor of 1 means equal time in both editors, a factor of 2 would mean that users took twice as long in our editor as in the text editor. On the right: Boxplot of time in seconds per task (JS, Clojure, Regex) in our editor and in the text editor.

parentheses in the text editor. In our editor, participants in some cases accidentally entered an incorrect but visually similar language construct. Answers from the TLX (Figure 17) suggest a slightly lower task load in the text editor. In general, task load appears not to have varied strongly across the editors with all dimensions averaging around close to the same load level. As shown in Figure 16, a majority of participants tended to agree that text editing, deleting, and navigating in our system felt natural, with navigation scoring the lowest.

RQ2: Consistency Between Languages. Our experiment was specially set up to demand of our participants to extrapolate learned interactions between languages. Instructions were only given for the Python editor, to familiarize users with the editor in general. The tasks were then carried out in editors for different languages with no further instructions and no time for the users to familiarize themselves. Still, participants successfully completed all tasks. As such, we conclude that participants successfully formed an understanding of the editors' function during the tutorial that generalized across languages.

RQ3: Editing Efficiency. With regard to the editing efficiency, we found that, in general, participants were slower using our editor than using the text editor. The median of the sums of all task times of each participant is 181s for the text editor and 243s for our system. However, the results differ between languages as can be seen in the detailed breakdown in Figure 18. For the large tasks in JavaScript and Clojure, most participants were able to use the editor with only a small slowdown. For JavaScript the median slow-down factor of our editor is 1.2x (range 0.8x - 2.7x) and for Clojure it is 1.3x (range 0.8x - 4.8x). In both languages, a few participants even performed faster than or very much on par with the text editor. The small slowdown for JavaScript and Clojure is promising as participants were able to use the editor with such a small slowdown despite the fact that they initially were unfamiliar with the editor and only received minimal training.

The Regex task yielded different results: Participants exhibited a larger slowdown factor of 1.9x (range 1.0x - 4.2x). Note that these higher slowdown factors for the Regex task apply to task times that are much lower: the Regex task has a median task time of 50s across all conditions (range 19s - 159s), while the JavaScript task has a median task time of 103s (range 49s - 239s). Further, the Regex task turned out to have a special influence on how participants interacted with the editor. The subsequent detailed analysis of interactions shows that many participants used significantly more navigation actions in the Regex task when using our editor ($t(17) = 4.2$, $p < .001$), as seen in Figure 19. Overall, our presented setup does not allow us to decide whether the higher slowdown just coincides with the shorter tasks and the higher number of navigation events, or whether there is a causal relation between any of the three.

While we expect language proficiency to generally have an effect on participants' editing efficiency, in our study, other factors influenced the efficiency more strongly. Almost all participants reported to be somewhat familiar with JavaScript and Regex, while all reported to be unfamiliar with Clojure, as seen in Figure 14. At the same time, Clojure editing efficiency is not worse than Regex

editing efficiency. Also, while there are moderate, negative correlations between the time overhead and language proficiency, none of these correlations were significant for our results.

We also investigated the recorded interactions as shown in Figure 19 in more detail in order to understand the slowdown in editing performance in our editor and to gain insights for future iterations of structured editors. Participants used significantly more interactions in our editor ($t(53) = 2.8$, $p < 0.01$) and also took longer per interaction ($t(53) = 4.8$, $p < 0.001$), so both aspects need to be considered in the future. A notable exception of these results, which may lead to more interesting insights in the future is that the difference in the number of interactions was less pronounced in the JavaScript task and the only non-significant difference ($t(17) = -1.0$, $p = 0.33$).

Threats to Validity. Even though we tried to prevent a learning effect between tasks by letting users read and understand every change they want to perform before each task and by shuffling and exchanging identifiers, it is still possible that participants encountered unexpected challenges only after they started typing on the first task. As a result, some cases may favor the second task in each language. In addition, the small size of the tasks may overemphasize incidental differences, as the time taken to recover from accidental inputs in both editors may amount to a large portion of total edit time. Further, the comparison between a text editor that users have been working with for years and our novel interface that users only had around 5 minutes of training time with necessarily renders the task completion times as approximations and as an indication of the efficiency users can expect as they just get started with the interface.

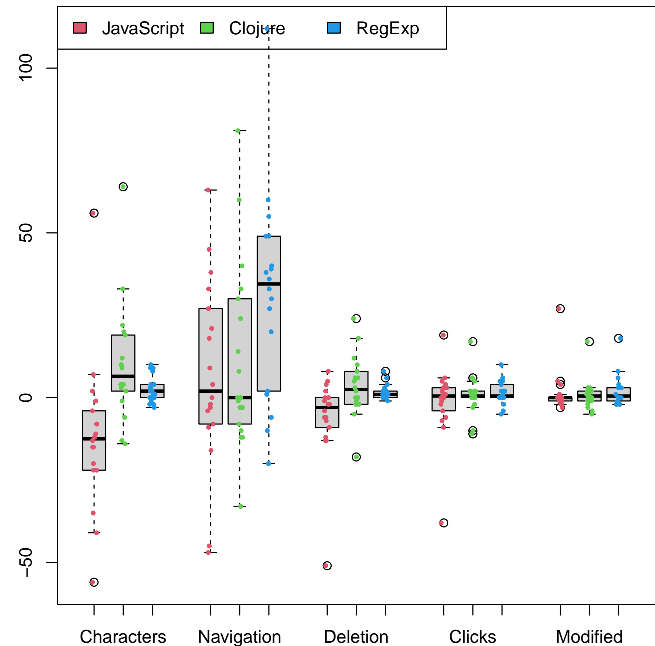

Figure 19: Boxplot of delta between number of interactions per language in our editor minus the text editor. Values below 0 indicate a count that was lower in our editor. Navigation includes arrow keys, home, end, and page up and down; deletion includes backspace and delete keys; clicks is the number of mouse clicks; modified includes all keys where ctrl/cmd were held simultaneously; characters include all other key presses.

5.2 Observations

In the following, we describe observations and insights we collected while the participants were using the editor and the subsequent interviews. Quotes from participants are translated from German by the authors and marked with P1 through P18.

Some participants expressed surprise at how quickly they had become familiar with the editor that visually seemed to suggest a more complicated method of use (P12: "using [the editor] felt very intuitive", P4: "I did not expect that I would be comfortable working in this editor so quickly."). Especially in the languages where participants were not as familiar, a majority of participants expressed that they appreciated the support given by the syntactic support of the structure editor. P9 said "from the layout, one could directly see when the code had a wrong structure, for example the way the [JavaScript] arrow function acts on expressions. In the text editor, I was less sure about that."

We observed three notable issues with our design that were recurring between runs. First, participants encountered issues with error recovery: users sometimes did not notice that a typo had led to the creation of a different language construct than desired and subsequently entered text was thus also placed in undesired spots.

Second, the popup that buffered input while it was still ambiguous caused some confusion, especially initially, as participants were still getting used to its function. Occasionally, participants would type a larger sequence in the popup but at an incorrect insert point, expecting it to eventually accept their input. For example, P14 said referring to the popup: "the hints were mostly confusing, maybe find a more friendly way to tell me what's happening". P3 said "the popup initially pulled me out of my [editing] flow, by now I have gotten used to it".

Third, we believe that the visual presentation we chose led some participants to get into a mode akin to filling out a form, as they tended to press the enter key when (read-only) hints popped up or after finishing to type in a text field (e.g, P15: "[...] pressed enter frequently, even though I don't usually do that when coding"), whereas just typing like in a text editor would have been the intended interaction. Similarly, navigation in our system was surprising to users in some ways, a potential explanation for the comparatively lower scores in a natural feeling in Figure 16 and the differences in interaction counts in Figure 19. While users reported that block-wise navigation behaved as they expected it to, they also reported that they had trouble predicting where the cursor would go in character- and line-wise navigation. P3 said "it wasn't clear how often I had to press [the arrow keys] to get to a specific spot. When I held control [for block-wise movement] it worked fine though".

On the positive side, some participants lauded the large padding for clearly showing structures, especially for the regular expression task (P12: "seeing the regex spread out like this helped understand its function"). Contrarily, the extra space taken up by the blocks led

Table 1: Projects from which we sampled the 50 expressions on single lines for our evaluation. For each language, we selected popular repositories of medium size that were active at the time of writing. The reported lines of code and number of files include only files of the language we evaluated.

Project	Language	Lines of Code	Files	Failures
flask[4]	Python	10372	75	1
express.js[5]	JavaScript	16381	153	0
Compojure[6]	Clojure	1202	12	0
Metacello[7]	Smalltalk	64785	4596	0
Vue.js[8]	TypeScript	61351	386	4

the auto-layout to introduce more line breaks than in the equivalent program in the text editor, increasing the vertical distance of program elements further and adding visual complexity (P18: "the display of the regular expressions felt noisy because there were so many boxes").

Deletion was mentioned positively during the interviews across participants and proved to be a reliable way for users to backtrack, for example P16 said "deleting the entire box like this feels satisfying", or P9 said "deleting felt the same as in a text editor".

According to the interviews, structural selection was the editor feature that was best received. P17, for example said: "I liked how easy it was to replace whole blocks. I could just shift-select the entire [expression] without having to precisely select the [text range of the] expression.", and P3 said: "using [structural selection] by selecting upward worked well. Copy-pasting whole blocks is usually what I intend to do anyways". Participants reported that they felt more productive and explicit in their actions, being able to perform their desired tasks with fewer keypresses than in a text editor. P18 said for example, "I got a bit the impression of working in Vim, it felt similarly efficient".

5.3 Automatically Retyping Program Subtrees

In the user study, we observed that users were able to formulate any language constructs the tasks required and were also able to recover from typos using backspace. To get an impression of the robustness of our system when entering language constructs we did not choose ourselves, we sampled lines of code of a number of popular open-source projects, deleted them, and retyped them using our system. Note that the input was thus the final code that a programmer checked into source control and did not contain slips like typos that would happen when first formulating the code. The experiment was thus used to evaluate compatibility with diverse language constructs and not a perfect simulation of user behavior. Error recovery and user behavior were covered by our user study. Thus, by checking a variety of language constructs in numerous situations, we can identify cases where the required input to enter a language construct deviates from the exact sequence used in a text editor; an absence of errors would give us greater certainty that our approach provides an exact match of input behavior to a text editor.

We selected five open-source projects, choosing from active projects with a comparatively high count of Github stars. The list of projects is shown in Table 1. For each project, we picked a random file and a random subtree in that file that was on a single line to approximate a change a user may do during a refactoring. We then take the selected program subtree as textual source code and send the corresponding string character by character as keyboard input to our system. Finally, we compare if the resulting subtree is identical in structure to the one we had deleted. For each project, we repeat this process 50 times, yielding a total of 250 re-typed subtrees.

Almost all subtrees were correctly recreated, as shown in the last column of Table 1. The exception in Python occurred on an import statement of the form `from a import b`. As commas are optional in our system, there is a valid insert position before the `import` token that accepts the first letters of the `import` keyword, leading to an expression of the form `from a, import import b` (as the import keyword had been autocompleted but also re-typed into the insert position). Here, our design should either reconsider the removal of commas or redirect the input as soon as it was clear that the user was typing a keyword, not a package name.

The TypeScript failures consist of 3 parsing errors in our system, and one template string of the form `` `${key}_${i}` ``. Our system currently requires users to explicitly state that they aim to create a template interpolation by moving into an insert position inside the string.

For a human user observing the editor state both issues are easily circumvented, potentially even more so as users become more familiar with a mindset of thinking in language constructs as opposed to textual tokens. However, to not pose any unexpected surprises, we plan to address both issues in future iterations.

6 DISCUSSION AND FUTURE WORK

Our described concept to achieve a text-like editing experience was confirmed by almost all participants, with some not even being able to state differences to editing in text editors when prompted during the interview.

In the following, we discuss means to address the three major points for improvement that resulted from our interviews and observations, as described subsection 5.2.

Error Recovery. As our system creates structures as soon as the input is unambiguous, it depended on users noticing faulty input quickly, e.g., when users might want to create a for loop but mistype a keyword and thus create a different language construct but continue to type as if the for loop construct had been created. We took great care to ensure that correcting errors is easy by allowing users to simply use backspace to undo the creation of language elements even of large constructs, which was also well-received in the interviews and in practice. Still, users tended to take some time to identify that a wrong language construct had been created and to which point they had to backtrack. Future work may investigate means to let users correct faulty input without need to delete structures.

[3]https://github.com/pallets/flask/tree/c34c84b69085e6bce67d0701b8f8ba3145f42ff2
[4]https://github.com/expressjs/express/tree/33e8dc303af9277f8a7e4f46abfdcb5e72f67
[5]https://github.com/weavejester/compojure/tree/cd9536e11f24c075ec670c2abc4b040
[6]https://github.com/metacello/metacello/tree/214c51948d36400251cf862009093765e
[7]https://github.com/vuejs/vue/tree/60d268c4261a0b9c5125f308468b31996a8145ad

Input Popup. The popup that buffers input while it is ambiguous was described as confusing by users. From our observations, we believe the main hurdle was a lack of clarity on where the buffered text would eventually be integrated into the program tree, as the popup element obscures potentially important context. In a future iteration, we will investigate removing the popup entirely and integrating the buffered text into the already-existing tree, to give users continuous feedback about the input's precise position in the tree. Further, this will allow input to temporarily exist in the tree, while the user deletes elements that were preventing a reinterpretation of a block to a block of another type, as it may occur in the scenario in Figure 12.

Visuals and Layout. The appearance and layout of blocks proved to be a point where users were divided: while some appreciated the increase in clarity through the large padding, others found the resulting difference from the line-like layout they are used to from textual code confusing. We believe the largest impact on usability, especially for predicting the outcome of navigation actions, arose from heterogeneous size of gaps in the block layout. In a text editor with a monospace font the cursor always moves equal distances. In our editor the vertical gaps between elements are of different sizes, leading the cursor to seemingly accelerate as it jumped greater vertical distances. This was compounded by the large padding we chose for blocks and occasional suboptimal decisions by our auto-layout. Similarly, prior work emphasized that screen space usage tends to be a major issue for visual programming interfaces [6, 25, 26]. Future versions of our layouting algorithm may thus tweak visuals to appear more strongly as in a text editor, for example by reducing padding of blocks, making hints for structure more subtle, and aligning elements to have equal distances.

7 CONCLUSION

We presented Sandblocks, a system that, given arbitrary Tree-sitter grammars, generates structured editors that have consistent interactions across languages and, according to a user study, tend to feel natural to users coming from a textual editing background. Guided by three central heuristics, (1) letting navigation follow the visual structure of the tree, (2) interpreting character entry as a textual parser would, and (3) conservatively requiring explicit action to delete structures, our input reconciliation system meets users expectations to make editing in our generated structure editors feel familiar. Our system thus forms a basis for making the vast number of current textual languages available in structured editing, hopefully enabling future work that better integrates tools with the syntax tree, or form a middleground between structured and textual editing to benefit both learners of particular general-purpose languages and enrich the editing experience of advanced users.

ACKNOWLEDGMENTS

We thank Jens Lincke for feedback on the structure of the paper. We thank Paul Methfessel and Corinna Jaschek for feedback on early versions of the generated structured editors.

This work was supported by Deutsche Forschungsgemeinschaft (DFG) grant #449591262. We also gratefully acknowledge the financial support of HPI's Research School[9] and the Hasso Plattner Design Thinking Research Program[10].

REFERENCES

[1] Tom Beckmann, Stefan Ramson, Patrick Rein, and Robert Hirschfeld. 2020. Visual Design for a Tree-Oriented Projectional Editor. In *Conference Companion of the 4th International Conference on Art, Science, and Engineering of Programming* (Porto, Portugal) *(<Programming> '20)*. Association for Computing Machinery, New York, NY, USA, 113–119. https://doi.org/10.1145/3397537.3397560

[2] Tom Beckmann, Patrick Rein, Toni Mattis, and Robert Hirschfeld. 2022. Partial Parsing for Structured Editors. In *Proceedings of the 15th ACM SIGPLAN International Conference on Software Language Engineering* (Auckland, New Zealand) *(SLE 2022)*. Association for Computing Machinery, New York, NY, USA, 110–120. https://doi.org/10.1145/3567512.3567522

[3] Thorsten Berger, Markus Völter, Hans Peter Jensen, Taweesap Dangprasert, and Janet Siegmund. 2016. Efficiency of Projectional Editing: A Controlled Experiment. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Seattle, WA, USA) *(FSE 2016)*. Association for Computing Machinery, New York, NY, USA, 763–774. https://doi.org/10.1145/2950290.2950315

[4] Alan F. Blackwell, Carol Britton, Anna Louise Cox, Thomas R. G. Green, Corin A. Gurr, Gada F. Kadoda, Maria Kutar, Martin Loomes, Chrystopher L. Nehaniv, Marian Petre, Chris Roast, Chris Roe, Allan Wong, and Richard M. Young. 2001. Cognitive Dimensions of Notations: Design Tools for Cognitive Technology. In *Proceedings of the 4th International Conference on Cognitive Technology: Instruments of Mind (CT '01)*. Springer-Verlag, Berlin, Heidelberg, 325–341.

[5] Marat Boshernitsan. 2001. *Harmonia: A Flexible Framework for Constructing Interactive Language-Based Programming Tools.* Technical Report CSD-01-1149. University of California at Berkeley, USA.

[6] Wayne Citrin, Richard Hall, Carlos Santiago, and Benjamin Zorn. 1998. Addressing the Scalability Problem in Visual Programming Through Containment, Zooming and Fisheyeing. In *1998 IEEE Aerospace Conference Proceedings (Cat. No.98TH8339)*, Vol. 4. IEEE, USA, 189–202. https://doi.org/10.1109/AERO.1998.682192

[7] Jonathan Edwards and Tomas Petricek. 2021. Typed Image-based Programming with Structure Editing. (2021). arXiv:2110.08993 [cs.PL] Online at http://tomasp.net/academic/papers/typed-image; Presented at Human Aspects of Types and Reasoning Assistants (HATRA), 2021.

[8] Jonathan Edwards and Tomas Petricek. 2022. Interaction vs. Abstraction: Managed Copy and Paste. In *Proceedings of the 1st ACM SIGPLAN International Workshop on Programming Abstractions and Interactive Notations, Tools, and Environments* (Auckland, New Zealand) *(PAINT 2022)*. Association for Computing Machinery, New York, NY, USA, 11–19. https://doi.org/10.1145/3563836.3568723

[9] Epic Games. 2012. *Unreal Engine Blueprint.* Epic Games. Retrieved 11 September 2022 from https://docs.unrealengine.com/4.26/en-US/BlueprintAPI/

[10] Bryan Ford. 2004. Parsing Expression Grammars: A Recognition-Based Syntactic Foundation. *SIGPLAN Not.* 39, 1 (jan 2004), 111–122. https://doi.org/10.1145/982962.964011

[11] David B. Garlan and Philip L. Miller. 1984. GNOME: An Introductory Programming Environment Based on a Family of Structure Editors. *SIGSOFT Softw. Eng. Notes* 9, 3 (apr 1984), 65–72. https://doi.org/10.1145/390010.808250

[12] Google. 2020. *Blockly.* Google. Retrieved 10 August 2022 from https://developers.google.com/blockly

[13] Sandra G. Hart. 2006. Nasa-Task Load Index (NASA-TLX); 20 Years Later. *Proceedings of the Human Factors and Ergonomics Society Annual Meeting* 50, 9 (2006), 904–908. https://doi.org/10.1177/154193120605000909

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

[15] Brian Hempel, Justin Lubin, Grace Lu, and Ravi Chugh. 2018. Deuce: A Lightweight User Interface for Structured Editing. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 654–664. https://doi.org/10.1145/3180155.3180165

[16] 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. https://doi.org/10.18293/vlss2017-012

[17] 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. https://doi.org/10.1145/263700.263754

[18] 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 *CHI '05 Extended Abstracts on Human Factors in Computing Systems* (Portland, OR, USA) *(CHI EA '05)*. Association for Computing Machinery, New York, NY, USA, 1557–1560. https://doi.org/10.1145/1056808.1056965

[9] https://hpi.de/en/research/research-school.html

[10] https://hpi.de/en/dtrp/

[19] Amy J. Ko and Brad A. Myers. 2005. Citrus: A Language and Toolkit for Simplifying the Creation of Structured Editors for Code and Data. In *Proceedings of the 18th Annual ACM Symposium on User Interface Software and Technology* (Seattle, WA, USA) *(UIST '05)*. Association for Computing Machinery, New York, NY, USA, 3–12. https://doi.org/10.1145/1095034.1095037

[20] 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 SIGCHI Conference on Human Factors in Computing Systems* (Montréal, Québec, Canada) *(CHI '06)*, Rebecca E. Grinter, Tom Rodden, Paul M. Aoki, Edward Cutrell, Robin Jeffries, and Gary M. Olson (Eds.). Association for Computing Machinery, New York, NY, USA, 387–396. https://doi.org/10.1145/1124772.1124831

[21] Eyal Lotem and Yair Chuchem. 2011. *Lamdu*. Lamdu Community. Retrieved 31 August 2022 from https://www.lamdu.org/

[22] John H. Maloney and Randall B. Smith. 1995. Directness and Liveness in the Morphic User Interface Construction Environment. In *Proceedings of the 8th Annual ACM Symposium on User Interface and Software Technology* (Pittsburgh, Pennsylvania, USA) *(UIST '95)*. Association for Computing Machinery, New York, NY, USA, 21–28. https://doi.org/10.1145/215585.215636

[23] Mauricio Verano Merino, Jur Bartels, Mark van den Brand, Tijs van der Storm, and Eugen Schindler. 2021. Projecting Textual Languages. In *Domain-Specific Languages in Practice: with JetBrains MPS*, Antonio Bucchiarone, Antonio Cicchetti, Federico Ciccozzi, and Alfonso Pierantonio (Eds.). Springer International Publishing, Cham, 197–225. https://doi.org/10.1007/978-3-030-73758-0_7

[24] Philip Miller, John Pane, Glenn Meter, and Scott Vorthmann. 1994. Evolution of Novice Programming Environments: The Structure Editors of Carnegie Mellon University. *Interactive Learning Environments* 4, 2 (1994), 140–158. https://doi.org/10.1080/1049482940040202

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

[26] Bonnie A. Nardi. 1993. *A Small Matter of Programming: Perspectives on End User Computing*. MIT Press, Cambridge, MA, USA. https://doi.org/10.7551/mitpress/1020.001.0001

[27] David Notkin. 1985. The GANDALF project. *Journal of Systems and Software* 5, 2 (1985), 91–105. https://doi.org/10.1016/0164-1212(85)90011-1

[28] Cyrus Omar, Ian Voysey, Michael Hilton, Joshua Sunshine, Claire Le Goues, Jonathan Aldrich, and Matthew A. Hammer. 2017. Toward Semantic Foundations for Program Editors. In *2nd Summit on Advances in Programming Languages (SNAPL 2017) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 71)*, Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi (Eds.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 11:1–11:12. https://doi.org/10.4230/LIPIcs.SNAPL.2017.11

[29] 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. https://doi.org/10.1145/1592761.1592779

[30] André L. Santos. 2020. Javardise: A Structured Code Editor for Programming Pedagogy in Java. In *Conference Companion of the 4th International Conference on Art, Science, and Engineering of Programming* (Porto, Portugal) *(<Programming> '20)*. Association for Computing Machinery, New York, NY, USA, 120–125. https://doi.org/10.1145/3397537.3397561

[31] Martijn M. Schrage. 2004. *Proxima - A presentation-oriented editor for structured documents*. Ph. D. Dissertation. Institute for Programming research and Algorithmics, Universiteit Utrecht. https://hdl.handle.net/1874/1074

[32] Tamás Szabó, Markus Voelter, Bernd Kolb, Daniel Ratiu, and Bernhard Schaetz. 2014. Mbeddr: Extensible Languages for Embedded Software Development. *Ada Lett.* 34, 3 (Oct. 2014), 13–16. https://doi.org/10.1145/2692956.2663186

[33] Tim Teitelbaum and Thomas Reps. 1981. The Cornell Program Synthesizer: A Syntax-Directed Programming Environment. *Commun. ACM* 24, 9 (sep 1981), 563–573. https://doi.org/10.1145/358746.358755

[34] Tijs van der Storm. 2011. *The Rascal Language Workbench.* Technical Report SEN-1111. Centrum Wiskunde & Informatica, Amsterdam. https://ir.cwi.nl/pub/18531

[35] 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* (Chicago, IL, USA) *(SLE 2021)*. Association for Computing Machinery, New York, NY, USA, 83–98. https://doi.org/10.1145/3486608.3486908

[36] Mauricio Verano Merino and Tijs van der Storm. 2020. Block-Based Syntax from Context-Free Grammars. In *Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering* (Virtual, USA) *(SLE 2020)*. Association for Computing Machinery, New York, NY, USA, 283–295. https://doi.org/10.1145/3426425.3426948

[37] Markus Voelter. 2011. Language and IDE Modularization, Extension and Composition with MPS, In Generative and Transformational Techniques in Software Engineering IV, GTTSE 2011, Ralf Lämmel, João Saraiva, and Joost Visser (Eds.). *GTTSE 2011* 7680, 383–430. https://doi.org/10.1007/978-3-642-35992-7_11

[38] 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, Netherlands) *(SLE 2016)*, Tijs van der Storm, Emilie Balland, and Dániel Varró (Eds.). Association for Computing Machinery, New York, NY, USA, 28–40. https://doi.org/10.1145/2997364.2997365

[39] Markus Völter, Janet Siegmund, Thorsten Berger, and Bernd Kolb. 2014. Towards User-Friendly Projectional Editors. In *Software Language Engineering*, Benoît Combemale, David J. Pearce, Olivier Barais, and Jurgen J. Vinju (Eds.). Springer International Publishing, Cham, 41–61.