tylr: A Tiny Tile-Based Structure Editor

David Moon
dmoo@umich.edu
University of Michigan
Ann Arbor, MI, USA

Andrew Blinn
blinnand@umich.edu
University of Michigan
Ann Arbor, MI, USA

Cyrus Omar
comar@umich.edu
University of Michigan
Ann Arbor, MI, USA

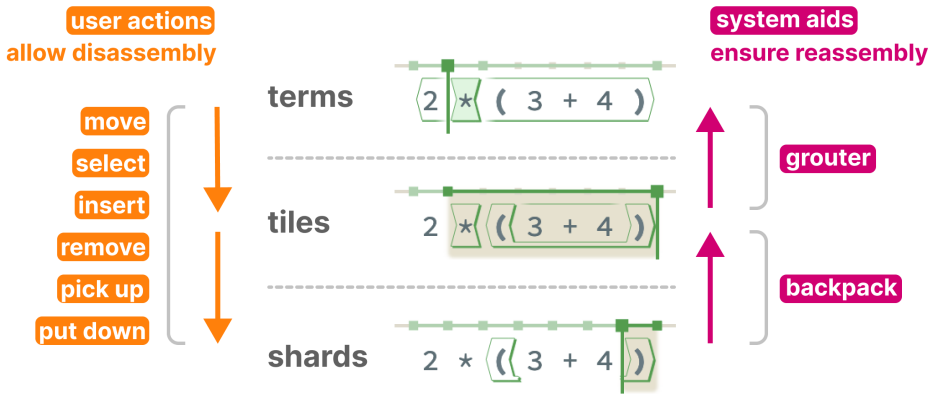


Figure 1. A high-level schematic of the concepts of tile-based editing. A tile-based editor operates on three levels of structure: terms, which follow the abstract syntax of the language; tiles, which correspond to groups of matching delimiters; and shards, which correspond to individual tokens and delimiters. Terms disassemble into tiles, tiles into shards as needed to accommodate user actions; meanwhile, system aids assist and guide user actions to ensure shards reassemble back to tiles, tiles back to terms.

Abstract

Structure editors designed for keyboard input often struggle to resolve the tension between maintaining hierarchical term structure and offering efficient linear editing affordances. Contemporary designs either compromise structure by deferring to text near the leaves or else maintain structure by permitting only edits that transform the selected term. However, visually adjacent sequences (e.g. of operators, operands, and individual delimiters) do not always cleave cleanly to term boundaries, so even experienced users report difficulties with selection and code restructuring tasks. We propose a novel approach to structure editing, *tile-based editing*, that maintains term structure while offering linear selection and modification affordances. The idea is to allow *disassembly* of terms into linearly sequenced tiles and shards around user selections, while guiding the user through restructuring actions and automatically inserting holes in a manner that ensures *reassembly* into a term.

This paper introduces `tylr`, a tiny tile-based editor designed primarily to highlight this uniquely flexible set of affordances. We evaluated `tylr` with a lab study where participants performed simple code transcription and modification tasks using `tylr` as well as a text editor and a structure editor built on JetBrains MPS, a state-of-the-art keyboard-driven structure editor generator. Our results indicate that participants frequently made use of `tylr`'s selection expressivity, and that this flexibility helped them complete some modification tasks significantly more quickly than with the MPS editor. We further observed that a few participants completed some tasks more quickly using `tylr` than with text, but were in general slowed by a number of limitations in our current design and implementation. We discuss these limitations and suggest future research and design directions aiming toward more flexible structure editing interfaces.

CCS Concepts: • **Software and its engineering → Integrated and visual development environments**.

Keywords: structure editing, projectional editing, usability

ACM Reference Format:
David Moon, Andrew Blinn, and Cyrus Omar. 2022. `tylr`: A Tiny Tile-Based Structure Editor. In *Proceedings of the 7th ACM SIGPLAN International Workshop on Type-Driven Development (TyDe '22), September 11, 2022, Ljubljana, Slovenia.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3546196.3550164

TyDe '22, September 11, 2022, Ljubljana, Slovenia

ACM ISBN 978-1-4503-9439-0/22/09…$15.00
https://doi.org/10.1145/3546196.3550164

1 Introduction

Programmers typically write their programs as text that is subsequently parsed into abstract terms of the underlying language. While text offers an ubiquitous and flexible interface, it suffers in the programming context because most textual edit states cannot be parsed successfully. Consequently, in such edit states, text-based programming environments must either disable language-specific tooling or otherwise rely on ad hoc heuristics to guess the user's intended term structure.

Structure editors resolve this issue by having the programmer directly modify the program's term structure, thereby ensuring well-formed input to language-specific analyses. Recent work on Hazelnut [18], a type-aware structure editor calculus, has shown that it is possible for every edit state to be meaningfully typed, giving the foundations for a continuously available type-directed development experience. Unfortunately, as many have observed [1, 2, 7, 9, 11, 13–15, 17, 20, 22], structure editors are often too slow or difficult to use. For example, while block-based editors like Scratch [12] have recently excelled at introducing programming to novices, their mouse-driven input and low visual information density make them unwieldy as users gain experience and create larger programs [4]. Other structure editors avoid those issues with keyboard-driven text-like interfaces, but either compromise structure by deferring to text at the leaves [10], or suffer from steep learning curves and difficult-to-predict editing behavior [3].

Whatever the input modality, structure editors struggle with a tension between the hierarchical structure demanded by the program's abstract syntax, and the editing affordances suggested by the program's two-dimensional projection onto the user's screen. Clusters of visually adjacent projectional components frequently do not hew to terms in the abstract syntax (e.g. `2 + 3` in the larger expression `2 + 3 * 4`), leading to a severely restricted selection capability from the user's point of view. Meanwhile, projected components of a term (e.g. the left and right braces delimiting an expression block), occupying potentially distant positions on the screen, remain rigidly locked together—by analogy, consider a vector graphics editor in which it is not possible to manipulate the individual corners or edges of a box. Both of these issues stem from the fact that structure editors traditionally operate solely on complete language terms.

We propose a novel approach to structure editing, called *tile-based editing*, that offers these missing partial structure editing affordances while still maintaining term structure. A tile-based editor visually organizes projected tokens into hierarchical structures of three distinct strata depicted in Figure 1: *terms*, *tiles*, and *shards*, ordered high to low. Higher structures may be *disassembled* (i.e. serialized) into lower structures as needed when the user's selection boundaries cut across the higher structure's token range. Meanwhile, lower structures are opportunistically *reassembled* (i.e. parsed) into higher structures in and around the user's selection as it grows and shrinks.

This paper contributes `tylr`, a tiny tile-based editor that concretely demonstrates these unique affordances. After an overview of `tylr`'s design, we present the results of a lab study where participants performed simple code transcription and modification tasks using `tylr` as well as a text editor and JetBrains MPS, the state-of-the-art in keyboard-driven structure editing. We found that participants using `tylr` completed some modification tasks significantly more quickly than when using MPS, particularly on those tasks where they made use of `tylr`'s selection expressivity. We further observed that participants using `tylr` occasionally outperformed themselves on similar tasks using a text editor, but were in general slowed by a number of limitations in our current design and implementation. We discuss these limitations and conclude with future research and design directions for further improving the usability of tile-based editing.

2 Background & Motivation

`tylr` contributes to a long history of structure editor design, dating back to the introduction of the Cornell Program Synthesizer [19] in 1981. Unlike prior work, which generally limits user edits to simple operations on complete program terms, `tylr` broadens the class of structures that the user may select and manipulate, while ensuring the result has valid term structure. In this section we illustrate the user experience limitations of strictly term-based structure editing, focusing on contemporary designs, to motivate `tylr`'s expressive selection capabilities.

The most popular structure editors today are block-based editors like Scratch [12]. In these editors, the user authors a program like the Scratch program shown to the right (adapted from [24]) by drag-and-dropping blocks together on a canvas. Each block corresponds to a syntactic form of the underlying language and is shaped, based on its sort and type, to visually indicate how it should be placed relative to other blocks. `tylr` employs a similar metaphor of syntactic-forms-as-puzzle-pieces, but uses a uniform shape system across all sorts, eliminating the visual design burden of language customization.

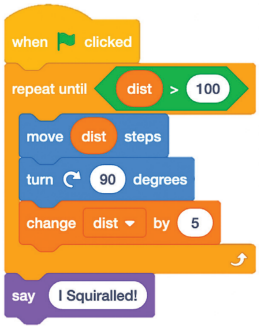


Figure 2. Scratch blocks

While block-based editors have seen great success in recent years at teaching programming to novices, they soon become unwieldy once users start creating and maintaining larger or more expression-oriented programs. For example—adapting an observation by Brown et al. [4]—constructing

the small calculation shown on the left involves assembling seven blocks, each requiring a sequence of mouse gestures to find the appropriate form and drag-and-drop it into the right spot on the canvas; the equivalent construction in a text editor or `tylr` would take seven keypresses. The block-based approach is further slowed if the user chooses to construct the expression left-to-right or bottom-up rather than top-down, since wrapping an existing block in a new one requires two drag-and-drop sequences. Meanwhile, the vertical height of the expression block grows with its tree height, leading to low visual information density when working with deeply nested expressions [9].

Other structure editors feature keyboard-driven text-like interfaces, which avoids the particular pitfalls of block-based editing but forces new trade-offs. Some such editors [10, 19] employ hybrid editing models, using structure editing for large syntactic forms while deferring to text editing at the leaves. This approach loses the benefits of structure editing at those levels, e.g., unrestricted language composition at the expression level.

Other editors, like those built with the language workbench JetBrains MPS [21], take a strictly structured approach and use a number of techniques to translate text-like editing flows into program term transformations. For example, MPS exposes hooks to the language engineer by which they may specify how to transform a program term and its context when the language user types text to the left or right of the term; common such patterns, e.g. left-to-right insertion of operator sequences, are codified and specified declaratively [23]. However, these techniques are limited to insertion and do not offer similar affordances for selection and deletion. Selections remain restricted to complete program terms, which can lead to cumbersome multi-step interactions to perform what amount to simple swaps of token ranges in the user-facing text-like projection, as shown for example in Figure 3. Meanwhile, while an MPS user may construct the expression `x * x + y * y` just like in a text editor, subsequently deleting the `+` token would delete `y * y` along with it, leaving only `x * x`; owing to its strictly term-based edit state, MPS editors cannot retain more than one child of a deleted term. Indeed, in a controlled user study, Berger et al. [3] observed that MPS novice users felt that selection was slow and inaccurate relative to text, despite a 45-minute training session and another 30-45 minutes worth of study tasks; and that both MPS novices and *experts* alike struggled to predict the effects of deletion.

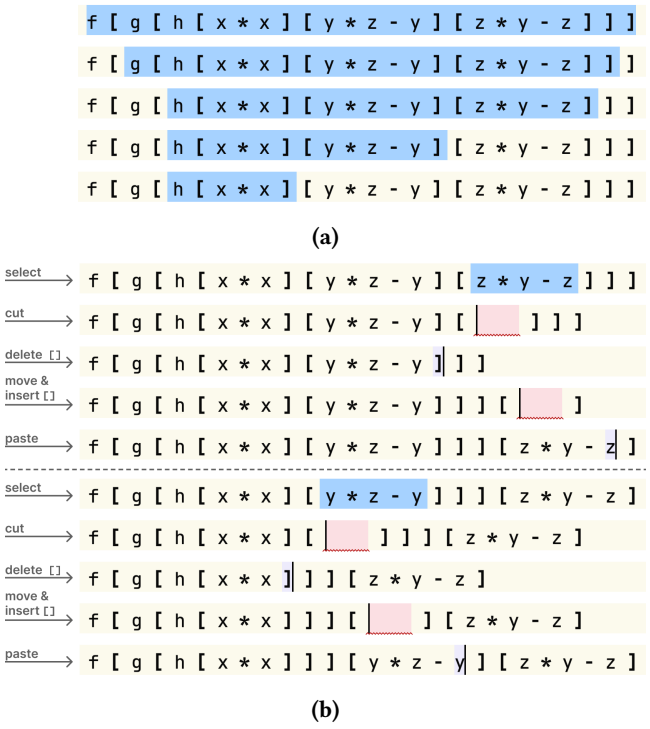


Figure 3. Screenshots of a JetBrains MPS editor being used to edit a program expression of nested function applications, written in a small artificial language called Lamb that we used in our user study (Figure 7). (a) shows all possible selections the user can make that contains a bracket, given MPS's restriction of selections to complete program terms. (b) shows the optimal edit sequence for completing one of our study tasks. The ultimate effect in the user-facing projection is swap the token ranges `[y * z - y]` and `[z * y -z]` and `]]`, but selection restriction means the user must go through two separate procedures of cutting an argument, deleting its enclosing brackets, reconstructing the brackets elsewhere, and pasting.

Such selection and deletion behavior may be less surprising with better visualizations indicating the term structure, such as in block-based editors, but even then users experience frustrations. In a user study of block-based editing involving large refactoring tasks [9], Holwerda and Hermans elicited post-task user responses on the cognitive dimensions [8] of block-based editing and found that *viscosity* was the most commented-on dimension with 24 remarks. Half (12) were positive, a majority of which were about the ease of refactoring when the selected elements corresponded to complete syntactic terms. Of the negative half, half (6) were about the difficulty of refactoring when the desired selection does not correspond to a complete term; for example, in Figure 2, dragging the `move` block out of the `repeat` block drags all

the statements below along with it, thereby requiring multiple gestures in sum to select and remove the single `move` block. These results help motivate the expressive selection capabilities of tile-based editing, independent of the specific visualization scheme or input modality.

3 Design Overview

`tylr` is a minimal prototype of tile-based editing, optimized at this stage for exposition rather than usability as a practical authoring tool. Its most salient limitations currently include a single-line edit state, single-character variables and numbers, and single-key input for constructing new forms. Nevertheless, it demonstrates uniquely flexible selection affordances compared to term-based structure editing while still preventing structural violations.

To a first approximation, `tylr` acts on lexical token sequences much like a text editor acts on character sequences. Using keyboard input, the user moves a cursor to positions between tokens, where they may insert and remove tokens, mark selection boundaries of arbitrary ranges, and 'cut' selections to 'paste' them elsewhere. Unlike a text editor, `tylr` assists and guides these interactions to ensure that every edit state, upon pasting, can be reassembled into a well-formed term.

This assistance is divided into two independent subsystems that may be understood as operating at distinct levels of `tylr`'s structural strata, as shown in Figure 1: the *grouter*, which aids the reassembly of tiles into terms (Section 3.1); and the *backpack*, `tylr`'s spiritual successor to the text editor's clipboard, which guides user movement to ensure proper reassembly of shards into tiles (Section 3.2).

3.1 Terms ⇌ Tiles: The Grouter

Panning the cursor over a program in `tylr` reveals its term structure, as depicted in Figure 4a, which follows the abstract syntax of a simple functional language. `tylr` indicates each term with a convex hexagonal outline, within which it highlights the term's constituent tokens. The visual nesting between terms reflects their strictly hierarchical organization.

Selecting the range encompassing the first term in Figure 4a reveals the term's disassembly into a sequence of tiles, as shown in Figure 4b. Each tile consists of a complete set of matching tokens (e.g. the tokens `let`, `=`, and `in` of the first tile) coupled with the terms those tokens delimit on both sides (e.g. the bound variable `f` and its definition as an anonymous function that returns the sum of its arguments). Unlike the strictly convex terms, the tips of a tile may each be convex or concave (e.g. the first tile has a convex left tip and a concave right tip). The different configurations of a tile's left and right tips indicate its syntactic role as an ⟨operand⟩, ⟨prefix operator⟨, ⟩postfix operator⟩, or ⟩infix operator⟨.

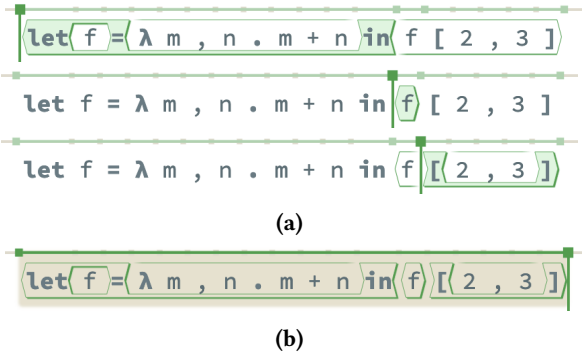


Figure 4. Screenshots of `tylr` showing a program's (a) term and (b) tile structure.

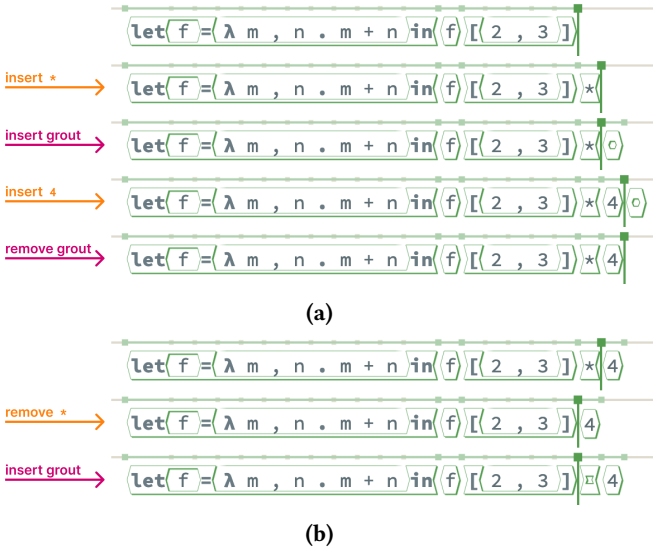
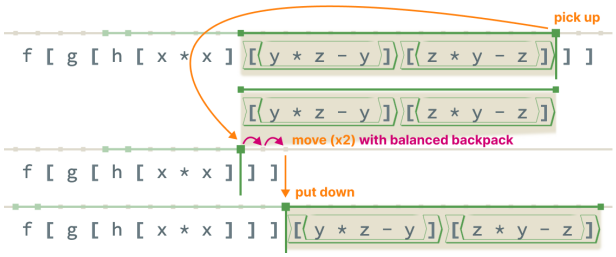


Figure 5. The grouter in action, invoked (in magenta) by `tylr` after every user modification (in orange). We show the underlying tile structure rather than the default term structure for expositional clarity.

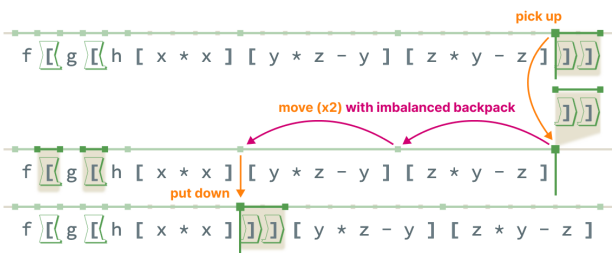
Via operator-precedence parsing, a sequence of tiles reassembles into a valid term if and only if the tiles fit together into a convex hexagon; that is:

(1) consecutive tiles fit together, i.e. one tile's convex tip meets the concave tip of the other; and
(2) tiles at the ends have convex outer tips.

In order to maintain these conditions of fit and ensure proper term reassembly, `tylr` is equipped with a subsystem we dub the grouter. Invoked immediately after each user modification, the grouter inspects the modification site and inserts or removes system-privileged structures, collectively called *grout*, that act as connecting glue between otherwise ill-fitting tiles.



(a) If the backpack is balanced, i.e. carries only complete tiles, then the user may move freely.



(b) If the backpack is imbalanced, i.e. carries unmatched shards, then the user may only traverse past balanced ranges.

Figure 6. The backpack in action, guiding user movement based on its contents.

For example, consider the sequence of edit states shown in Figure 5a, where the user modifies the program in Figure 4 by multiplying the function application `f[2, 3]` by 4. Upon the user inserting the operator `*`, the grouter inspects the affected tile sequence; identifies that the last tile has a concave right tip, which violates condition (2); and repairs the edit state by inserting grout to its right. Subsequently, when the user inserts 4, the grouter identifies that the affected sequence once more satisfies the conditions of fit, and removes the now excess grout.

Grout elements come in two varieties: convex and concave. Convex grout, such as that inserted and removed in Figure 5a, succeed the familiar concept of *holes* [18] in term-based structure editors. Meanwhile, concave grout model infix operator placeholders between yet-to-be-adopted operands. For example, consider the edit state sequence shown in Figure 5b, where the user press Backspace to the multiplication operator `*` inserted in Figure 5a. Upon deletion, the grouter identifies a violation of condition (1) and repairs the edit state by inserting concave grout between the orphaned operands. As discussed in Section 2, if we were to perform the same edit in a term-based editor with only the usual notion of holes, the editor would need to choose one of the orphaned children, `f[2, 3]` or 4, to remove along with the parent; with concave grout, `tylr` can save both.

3.2 Tiles ⇌ Shards: The Backpack

`tylr` features a second subsystem, called the backpack, that operates independently from the grouter. Upon making a selection, the user may pick it up into the backpack and put it down elsewhere, much like a text editor user uses the clipboard to cut and paste. Unlike the clipboard, the backpack is a visible component attached to the cursor. Moreover, it is *structure-aware* and guides user movement based on its contents to ensure they are put down in reasonable positions.

Figure 6 shows how the backpack could be used to complete one of the tasks we assigned our study participants. 4 out of 11 participants completed the task using an edit sequence like the one shown in Figure 6a: upon selecting the applied function arguments, they picked up the selection, moved right twice, and put it down. While term-based structure editors often provide cut-and-paste affordances, such a workflow would be impossible in that setting, since the selected tiles do not alone form a complete term—indeed, we observed participants particularly struggle to complete the same task with a term-based editor because of this limitation (Section 4.2).

Sometimes even more selection granularity may be desirable. Consider an alternative approach to completing the same task, shown in Figure 6b, taken by another 4 of our study participants. They began this edit sequence by selecting the closing brackets, thereby disassembling the function application tiles into shards: the individual matching tokens that comprise a tile. They then picked up this selection, moved left twice, and put it down. Whereas in Figure 6a the backpack's contents were *balanced* i.e. had no unmatched shards, in this case the backpack's contents were *imbalanced*.

Text editing programmers may be familiar with a feeling of tension that comes with manipulating such selections, given the possibility of miscounting delimiters and putting them somewhere that breaks the well-nested structure of their program. In a tile-based setting, the backpack relieves this burden by steering the user's cursor movement such that it can only move past balanced ranges if the backpack is imbalanced, ensuring that the result of unloading backpack can be reassembled into well-nested tiles. Moreover this may lead to efficiency gains: while both edit sequences in Figure 6 take 4 user actions, (b) requires only 2 steps of movement to make the initial selection, whereas (a) requires 16.

4 Evaluation

To investigate `tylr`'s usability, we ran a within-subjects lab study in which participants completed a series of short program editing tasks using VS Code, a text editor; a baseline term-based editor we built with JetBrains MPS; and `tylr`. We sought to answer the following questions:

- Does `tylr` help first-time users complete program editing tasks more quickly than with another keyboard-driven but term-based structure editor? How does

$$
\begin{array}{rrcl}
\text{expr} & e & ::= & n \mid x \mid (e) \\
& & \mid & \backslash\, x \,\{\, e \,\} \mid e[e] \\
& & \mid & \texttt{let}\ x = e\ \texttt{in}\ e \\
& & \mid & e * e \mid e \,/\, e \\
& & \mid & e + e \mid e - e \\
& & \mid & e, e \\
\text{num} & n & \in & \{0 - 9\} \\
\text{var} & x & \in & \{\mathsf{a} - \mathsf{z}\}
\end{array}
$$

(a) The binary operators are arranged into rows ordered by their operator precedence.

```
          (* A: bind to a variable and use *)
(* A-t *) \ x { let i = n * n - 8 in x/i }
(* A-m *) let f = \ x { let i = n * n - 8 in x/i } in f[n]

          (* B: internalize bindings *)
(* B-t *) let f = \ x { 5/x } in let m = n + 1 in let y = ( f[m] ) in y/n
(* B-m *) let y = ( let f = \ x { 5/x } in let m = n + 1 in f[m] ) in y/n

          (* C: extract a helper function *)
(* C-t *) (g[a] * h[b] + c * r[x * x], g[a] * h[b] + c * s[y * y])
(* C-m *) let f = \ n { g[a] * h[b] + c * n } in (f[r[x * x]], f[s[y * y]])

          (* D: transfer arguments *)
(* D-t *) f[g[h[x * x][y * z - y][z * y - z]]]
(* D-m *) f[g[h[x * x]]][y * z - y][z * y - z]
```

(b) The eight tasks are grouped into four pairs labeled A, B, C, D. Each pair consists of a transcription task (A-t, B-t, C-t, D-t) followed by a modification task (A-m, B-m, C-m, D-m).

Figure 7. The textual syntax of Lamb (a) and the editing tasks in Lamb we assigned our participants (b).

this performance compare to their text editor performance?
- To what extent do users make use of `tylr`'s selection expressivity?

4.1 Method

We recruited 11 participants (P1-P11, 5 female and 6 male, $\mu = 22.2$ years old, $\sigma = 2.9$ years) from students at the University of Michigan by posting in the university subreddit (r/uofm) and in chat forums shared by computer science graduate students, as well as by emailing students enrolled in the undergraduate course on programming languages. Because our tasks involved editing programs in an expression-oriented language (e.g. OCaml, Rust, Scala, etc), we selected for those with some prior exposure. Most participants reported less than a year of experience with such languages but had otherwise substantial programming backgrounds ($\mu = 6.8$ years, $\sigma = 3.3$ years). Each participant was compensated $30 dollars for a 75-minute session.

Each study session consisted of three components, one for each editor. Each component consisted of a 10-minute tutorial portion followed by a task portion, in which the participant completed small editing tasks with the given editor in an artificial expression-oriented language called Lamb. We designed Lamb's syntax, shown in Figure 7a, to accord with `tylr`'s prototypal limitations.

Figure 7b shows the eight editing tasks participants completed in each editor component. The tasks were presented in four pairs in a randomized order for each participant-component. Each pair (e.g. A) consisted a transcription task (e.g. A-t), where the participant transcribed a Lamb program from scratch (after taking up to 30 seconds to read it); followed by a modification task (e.g. A-m), where the participant modified their transcribed program (after taking up to a minute to read the modified program). Within the limits of Lamb and `tylr`, we designed our modification tasks to represent general code restructuring patterns one may encounter in larger-scale settings. We intentionally chose non-minimal starting programs so as to disincentivize wholesale deletion and re-transcription in modification tasks.

Every participant started with the VS Code component. We used its tutorial portion to introduce participants to Lamb and to verify they understood its term structure before proceeding to the structure editing components. Specifically, as we introduced Lamb's syntax, we asked participants to parenthesize all subterms in a few sample programs. We configured VS Code to syntax-highlight Lamb expressions [5] and color matching brackets [6].

Participants were randomly assigned to an order for the subsequent MPS and `tylr` components. Both tutorials covered the basics of expression construction; automatic hole/-grout insertion and removal; and selection and cut-and-paste capabilities. The MPS tutorial additionally covered MPS's "Surround With" menu [16], a user-invoked dropdown menu that provides options for wrapping the currently selected program term in a new form. Using MPS's grammar cells system [23], we configured our MPS editor to support left-to-right insertion of operator sequences, including wrapping a term as the conclusion of a new let expression and as the function of a new function application. In order to maintain parity with `tylr`'s limitations, our MPS editor used single-key inputs for constructing expression forms (e.g. `=` for a let expression, `\` for a lambda expression); variables were

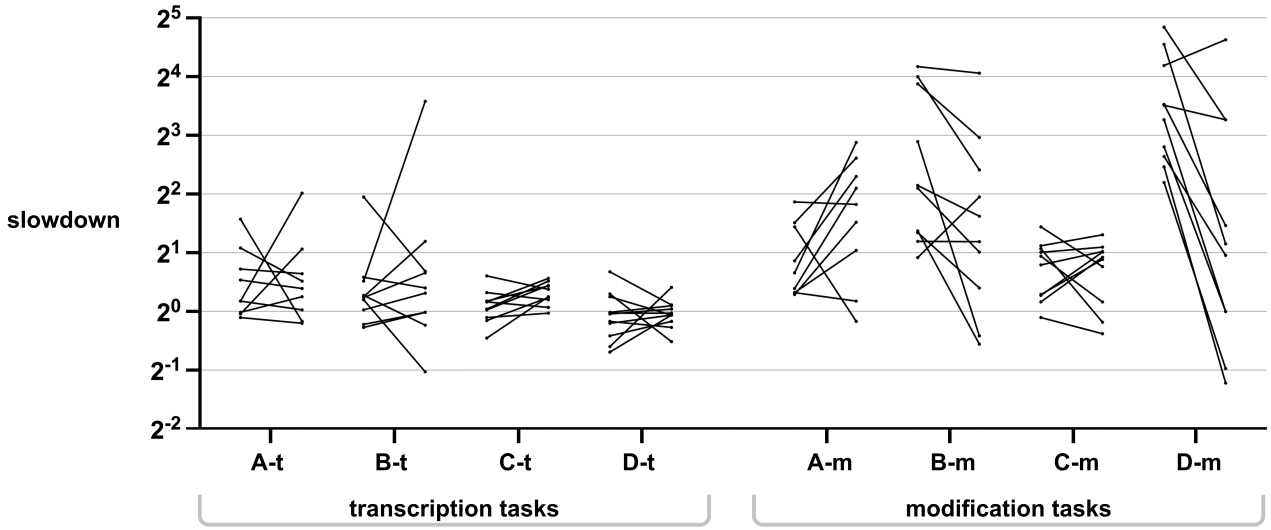


Figure 8. Summary of the slowdowns participants experienced in each task when using a structure editor as opposed to a text editor, where the slowdown is calculated as a participant's structure editor completion time divided by their text editor completion time. Each line segment corresponds to a participant; the left and right endpoints indicate the participant's MPS-vs-text slowdown and `tylr`-vs-text slowdown, respectively, on the x-axis-labeled task.

restricted to single characters; and the edit state was always laid out in a single line.

We asked the participants to complete each task as quickly and accurately as they comfortably could and recorded their screen during the tasks. We did not impose any time limits; no task took more than 5 minutes. A few participants did not complete all tasks in their final component because we ran out of time, and we discarded the data for the couple occasions the participant accidentally refreshed the browser in the middle of a task. To keep our data well-matched, for any missing or discarded data for a task, we discarded the corresponding data for the same task in all components.

4.2 Results

Our evaluation suggests that participants did indeed make use `tylr`'s selection expressivity and that this helped them complete some modification tasks more quickly than with MPS. On other tasks, however, participants were slowed by a few limitations in `tylr`'s current design.

Figure 8 summarizes the task completion times we measured across all three editor components. We treated the participant's VS Code completion time for each task as a normalization factor and focused our analysis on the relative slowdowns (or speedups) the participant experienced on the same task when using one of the two structure editors, calculated as the ratio of their structure editor completion time to their text editor completion time. By and large, participants were slower with the structure editors than with text, which we expected given that the participants had no prior experience with the structure editors. We are encouraged, however, to see that several were faster on some transcription tasks, and a few were faster with `tylr` specifically on some modification tasks, though some of this may be due to learning effects from completing the VS Code component first.

For each task, we used a paired t-test to check for significant differences between the base-2 logarithms of the structure editor slowdowns. We observed no significant differences between slowdowns on the transcription tasks except for C-t, where we found that participants experienced greater slowdown using `tylr` than with MPS ($t = 2.37, p < 0.05$, Cohen's $d = 0.79$). We think this was largely due to an incidental limitation: Task C-t involved moving past 6 closing brackets; meanwhile `tylr` did not share with VS Code and MPS the ability to move past a closing bracket by typing it, forcing users instead to reach for the right arrow key instead, frequently after a pause to stifle their usual habit or undo their accidental insertion of a new pair of brackets.

In the modification tasks, we found that participants experienced dramatically less slowdown using `tylr` over MPS on Tasks B-m ($t = -2.51, p < 0.05$, Cohen's $d = 0.83$) and D-m ($t = -4.87, p < 0.001$, Cohen's $d = 1.62$). Notably these tasks correspond to those in which participants made the most use of `tylr`'s selection expressivity. Figure 9 summarizes counts of selections users picked up into the backpack during the modification tasks of the `tylr` component, broken down by task and structure of the selected content. Overall, more than half (36) of all selections (67) picked up by participants fell into the balanced and imbalanced categories, i.e. could not be specified in MPS. The same is true specifically of the

		selection structure			
		term	balanced	imbalanced	total
task	A-m	10	0	5	15
	B-m	5	1	8	14
	C-m	16	5	5	26
	D-m	0	4	8	12
	total	31	10	26	67

Figure 9. Counts of selections participants picked up into the backpack when using `tylr` to complete the modification tasks, broken down by task and the following structural categorization of the selected content: a term at selection time (e.g. the selection in Figure 4a), balanced but not a term at selection time (Figure 6a), and imbalanced (Figure 6b).

selections picked up in Tasks B-m (9 out of 14) and D-m (12 out of 12) respectively, which suggests that `tylr`'s selection expressivity was important for completing those tasks more quickly. After completing Task D-m using the edit sequence in Figure 6a, P10 remarked: "That's exactly what I wanted to try to do in the last one and then it didn't work. It's nice that we get both the structure but also like when you do selections, like it works the way you expect it to, like it's actually taking the characters that you're expecting... so this is great."

We observed no significant differences between the structure editor slowdowns on Tasks A-m and C-m; both columns in Figure 8 show several participants experienced worse slowdowns with `tylr` than with MPS. On these tasks, we observed many participants get slowed by prototypal limitations of `tylr`'s backpack system. A major limitation is that the user cannot insert and remove forms as usual when they have something in the backpack, as one can with the clipboard in VS Code and MPS. Several participants forgot about this limitation when completing Task A-m with `tylr`: they started by picking up the starting program and attempted to construct a let expression, only to be reminded by `tylr`'s interface that this is not possible.

Another, more subtle breakdown was caused by the backpack's movement behavior changing dramatically given small changes in the picked-up selection. For example, consider the two edit sequences shown in Figure 10. In the first sequence, the picked-up selection is balanced, so subsequently the user may move freely, in particular into the let definition where they intend to put down the selection. Now suppose the user accidentally overselects the opening parentheses as well, as at the start of the second edit sequence. In this case, because the backpack contents are imbalanced, the user finds they cannot enter the let tile as intended. We observed a few participants get confused after making the same mistake when completing Task C-m with `tylr`.

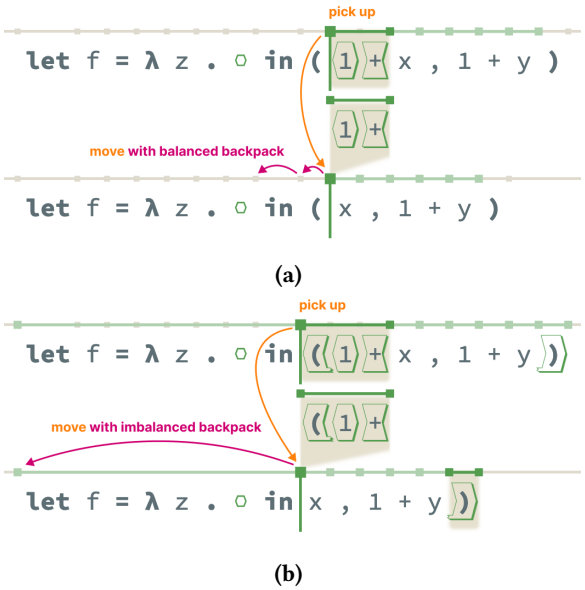


Figure 10. Two similar edit sequences showing the error-proneness of strictly backpack-guided movement. Intending to perform the first edit sequence in 10a, where the picked-up selection is balanced, the user may accidentally overselect and pick up an imbalanced selection, which dramatically changes the user's subsequent allowed movement.

4.3 Limitations

Our study had several limitations. Our task design was constrained by `tylr`'s prototypal nature; the editing tasks were small, synthetic, and given on single lines in an artificial language with unfamiliar syntax. The measured times record participants' first-time use of both structure editors and do not reflect optimal performance.

It is possible to engineer more ergonomic structure editors with MPS than the one we built and evaluated in this study. Part of the limitations of our editor were to maintain parity with `tylr`'s limitations, as described in Section 4.1. In general, it is possible to adjust the language grammar to improve an MPS editor's selection expressivity. Our editor directly implemented the expression structure of Lamb, as given in Figure 7a, which for example makes it impossible to select a let binding independent of its conclusion (e.g. `let x = 1 in` in `let x = 1 in x`); this would be possible if instead we introduced a distinct expression block sort consisting of a sequence of let bindings and expression lines, each individually selectable in this form. We view such grammatical adjustments as ad hoc approximations of the generic disassembly of terms into tiles in the tile-based setting, and sought to focus our comparison on pure term- and tile-based editing.

5 Future Work

Efficient and easy-to-use structure editing has been tantalizingly out of reach for many decades. This paper highlights and targets the central tension between consistently available hierarchical structure and flexible editing of its linearized representation. Our proposed solution, tile-based editing, navigates this tension by operating on a broader class of structures than traditional term-based editing, allowing disassembly of hierarchical structures while ensuring proper reassembly. Our user study of `tylr`, a tiny tile-based editor, showed that users made frequent use of this structural flexibility, and that this flexibility helped them complete some code restructuring tasks significantly more quickly than with a traditional term-based structure editor. We are encouraged by these results, although our study was limited due to `tylr`'s prototypal nature. In future work, we plan to scale up tile-based editing so that we may use and evaluate it in more realistic settings.

This involves two sets of challenges. The first centers around scaling up basic editing affordances, such as multi-key input, multi-character tokens, multi-line layout, as well as lifting the restrictions imposed by `tylr`'s current backpack system. The second centers around scaling up to more realistic languages featuring multiple sorts as well as tokens shared across different syntactic forms. We are currently exploring a new tile-based system aid that is capable of "remolding" tiles, as well as a sort system that allows for sort inconsistencies much like Hazelnut [18] allows for type inconsistencies. In addition, we hope to generalize our approach by investigating which grammar classes are suitable for tile-based editing, leading ultimately to a tile-based editor generator. If these challenges can be overcome, we hope to achieve a generic approach to structure editing that compromises virtually none of the fluidity and familiarity of text editing.

References

[1] R. Bahlke and G. Snelting. 1992. Design and structure of a semantics-based programming environment. *International Journal of Man-Machine Studies* 37, 4 (1992), 467–479. https://doi.org/10.1016/0020-7373(92)90005-6 Structure-based editors and environments.

[2] David Bau, Jeff Gray, Caitlin Kelleher, Josh Sheldon, and Franklyn Turbak. 2017. Learnable Programming: Blocks and Beyond. *Commun. ACM* 60, 6 (May 2017), 72–80. https://doi.org/10.1145/3015455

[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)*. ACM, New York, NY, USA, 763–774. https://doi.org/10.1145/2950290.2950315

[4] Neil C. C. Brown, Michael Kolling, and Amjad Altadmri. 2015. Position paper: Lack of keyboard support cripples block-based programming. In *2015 IEEE Blocks and Beyond Workshop (Blocks and Beyond)*. 59–61. https://doi.org/10.1109/BLOCKS.2015.7369003

[5] VS Code. 2022. Syntax Highlight Guide. https://code.visualstudio.com/api/language-extensions/syntax-highlight-guide. Accessed: 2022-05-30.

[6] Henning Dieterichs. 2021. Bracket pair colorization 10,000x faster. https://code.visualstudio.com/blogs/2021/09/29/bracket-pair-colorization. Accessed: 2022-05-30.

[7] Dennis R. Goldenson and Marjorie B. Lewis. 1988. Fine Tuning Selection Semantics in a Structure Editor Based Programming Environment: Some Experimental Results. *SIGCHI Bull.* 20, 2 (Oct. 1988), 38–43. https://doi.org/10.1145/54386.54400

[8] T.R.G. Green and M. Petre. 1996. Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework. *Journal of Visual Languages and Computing* 7, 2 (1996), 131–174. https://doi.org/10.1006/jvlc.1996.0009

[9] Robert Holwerda and Felienne Hermans. 2018. A Usability Analysis of Blocks-based Programming Editors using Cognitive Dimensions. In *2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 217–225. https://doi.org/10.1109/VLHCC.2018.8506483

[10] Michael Kölling. 2010. The Greenfoot Programming Environment. *ACM Trans. Comput. Educ.* 10, 4, Article 14 (Nov. 2010), 21 pages. https://doi.org/10.1145/1868358.1868361

[11] Bernard Lang. 1986. On the Usefulness of Syntax Directed Editors. In *Proceedings of an International Workshop on Advanced Programming Environments*. Springer-Verlag, Berlin, Heidelberg, 47–51.

[12] John Maloney, Mitchel Resnick, Natalie Rusk, Brian Silverman, and Evelyn Eastmond. 2010. The Scratch Programming Language and Environment. *ACM Trans. Comput. Educ.* 10, 4, Article 16 (Nov. 2010), 15 pages. https://doi.org/10.1145/1868358.1868363

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

[14] Sten Minör. 1992. Interacting with Structure-Oriented Editors. *Int. J. Man Mach. Stud.* 37, 4 (1992), 399–418. https://doi.org/10.1016/0020-7373(92)90002-3

[15] Jens Monig, Yoshiki Ohshima, and John Maloney. 2015. Blocks at Your Fingertips: Blurring the Line Between Blocks and Text in GP. In *Proceedings of the 2015 IEEE Blocks and Beyond Workshop (Blocks and Beyond) (BLOCKS AND BEYOND '15)*. IEEE Computer Society, Washington, DC, USA, 51–53. https://doi.org/10.1109/BLOCKS.2015.7369001

[16] JetBrains MPS. 2021. MPS Intentions. https://www.jetbrains.com/help/mps/mps-intentions.html. Accessed: 2022-05-30.

[17] Lisa Rubin Neal. 1986. Cognition-Sensitive Design and User Modeling for Syntax-Directed Editors. *SIGCHI Bull.* 18, 4 (May 1986), 99–102. https://doi.org/10.1145/1165387.30866

[18] Cyrus Omar, Ian Voysey, Michael Hilton, Jonathan Aldrich, and Matthew A. Hammer. 2017. Hazelnut: a bidirectionally typed structure editor calculus. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 86–99. https://doi.org/10.1145/3009837

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

[20] Michael L. Van De Vanter. 1995. Practical language-based editing for software engineers. In *Software Engineering and Human-Computer Interaction*, Richard N. Taylor and Joëlle Coutaz (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 251–267.

[21] Markus Voelter and Vaclav Pech. 2012. Language modularity with the MPS language workbench. In *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, Martin Glinz, Gail C. Murphy, and Mauro Pezzè (Eds.). IEEE Computer Society, 1449–1450. https://doi.org/10.1109/ICSE.2012.6227070

[22] Markus Voelter, 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.

[23] 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)*. ACM, New York, NY, USA, 28–40. https://doi.org/10.1145/2997364.2997365

[24] David Weintrop. 2019. Block-Based Programming in Computer Science Education. *Commun. ACM* 62, 8 (July 2019), 22–25. https://doi.org/10.1145/3341221