Gradual Structure Editing with Obligations

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

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

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

***Abstract*—Structure editors have long promised to facilitate a continuous dialogue between programmer and system—one uninterrupted by syntax errors, such that vital program analyses and editor services are always available. Unfortunately, structure editors are notoriously slow or difficult to use, particularly when it comes to modifying existing code. Prior designs often struggle to resolve the tension between maintaining a program's hierarchical structure and supporting the editing affordances expected of its linear projection. We propose the paradigm of *gradual structure editing*, which mitigates this tension by allowing for temporary disassembly of hierarchical structures as needed for text-like editing, while scaffolding these interactions by generating syntactic *obligations* that, once discharged, guarantee proper reassembly.**

This paper contributes the design and evaluation of a gradual structure editor called `teen tylr`. We conducted a lab study comparing `teen tylr` to a text editor and a traditional structure editor on structurally complex program editing tasks, and found that `teen tylr` helped resolve most usability problems we identified in prior work, though not all, while achieving competitive performance with text editing on most tasks. We conclude with a discussion of `teen tylr`'s remaining limitations and design implications for future code editors and parsers.

I. Introduction

Programming is cognitively demanding, so novices and experts alike rely on various editor services to help them understand, navigate, and modify programs. Often specific to the programming language, these services form a conversation between language and tool designers on the one hand, and programmers on the other, with each group preferring their own representation of the authored code. Language designers model programs as hierarchical tree structures, which enables compact specification of program analyses and transformations. Meanwhile, programmers typically read and write programs displayed as linearly sequenced text, decorated with secondary notation (e.g. colors, block outlines, type errors, etc), which lets them capitalize on their literacy and keyboard skills.

When an editor presents programs as decorated text, it is natural to expect it to offer standard text editing affordances. Text editing challenges editor service design, however, as the system is left to parse the text resulting from each edit to determine a corresponding tree structure for subsequent analysis. Many text editor states fail to parse, creating gaps in the availability of downstream services [1].

Due to this "gap problem", there has been a long and storied line of research on *structure editors* (a.k.a. *projectional editors*), which maintain a tree-structured, rather than textual, editor state, with *holes* standing for missing terms (i.e. sub-trees). The programmer interacts with a projection of this editor state, which specifies how the code is presented and offers appropriate editing affordances. The choice of projection varies widely—from the bright drag-and-droppable blocks of Scratch [2] to the keyboard-driven text-like interface of JetBrains MPS [3]—but all projections restrict user interactions to simple operations on the underlying tree, which ensures that the edit state is continuously well-structured and amenable to analysis. On the other hand, these restrictions can be in tension with the linearized projection and its suggested affordances, leading many [4–14] to report a highly viscous [15] editing experience, i.e. it can be cumbersome to modify existing code.

Concretely, let us consider the violin plots in Fig. 1 reproduced from a study comparing text editing to MPS, the state-of-the-art in keyboard-driven structure editing [16]. The left two plots in Fig. 1 show that, after the 90-minute study, MPS novices felt that selection was relatively slow and inaccurate. The viscosity here is caused by what we call the **selection expressivity problem**, namely that selections must cleave to whole terms. It is impossible to select portions of a term, or for selections to span across term boundaries, even when the intended tokens are visually adjacent. For example, in `2 * 3 + 4`, it is impossible to select `3 + 4` or `+ 4`. Novices, in particular, may find it difficult to "see" the invisible term boundaries that restrict selection.

Meanwhile, the right two plots in Fig. 1 show that both MPS novices and experts struggled to predict the effects of deletion operations, which can be suprisingly destructive in the name of maintaining strict tree structure. We construe this overall

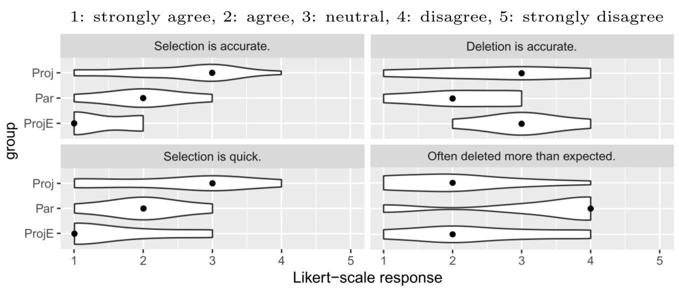


Fig. 1: Violin plots of post-task questionnaire responses from a controlled user study of MPS, adapted from [16]. Each plot partitions the responses across the three study groups: MPS novices (Proj), MPS experts (ProjE), and text editor users (Par).

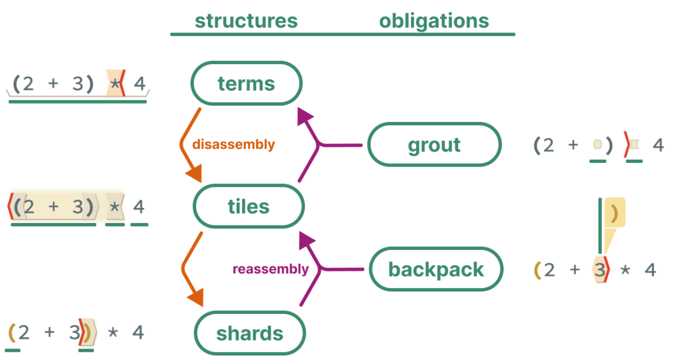


Fig. 2: A high-level schematic of the concepts of tile-based editing, this paper's realization of gradual structure editing.

phenomenon as the combination of two distinct problems.

First, the **delimiter matching problem** arises from the constraint that all keywords or symbols from the projection of a term (which we refer to as its *matching delimiters*) are inserted and deleted together. Matching delimiters may be visually distant from one another due to intervening children, leading to a sort of "spooky action at a distance" and making edits that amount to repositioning or changing an individual delimiter difficult. For example, in a text editor one can go from `f(2 * 3) + 4` to `f(2 * 3 + 4)` by deleting (or cutting) the closing parenthesis and inserting (or pasting) it after `4`, whereas in MPS, deleting the closing parenthesis also deletes the matching opening parenthesis (and the function argument, which brings us to the next problem).

The second source of over-deletion, which we refer to as the **multiplicity problem**, arises from the fact that in a text editor, terms can appear transiently adjacent to one another, e.g. deleting the `+` in `2 + 3` leaves `2 3`, whereas in a structure editor, deletion must leave behind either a hole, i.e. zero terms, or one term. Consequently, MPS deletes not just the `+` character but also all but at most one of its child operands, leading to behavior that is difficult even for experts to predict.

We propose a new paradigm for structured code editing, called *gradual structure editing*, that aims to resolves these three problems. The organizing principle is to permit local disassembly of hierarchically-structured terms to their projected components as needed to resolve the selection expressivity problem, as well as insert and delete these components individually. After each change, the system analyzes the locally linear structure to generate a set of syntactic *obligations* that, once discharged, guarantee reassembly to a complete term. Syntactic obligations generalize holes, which can be understood as obligating term insertion, to include matching- and multiplicity-related obligations.

We call this paper's particular realization of gradual structure editing *tile-based editing*, because disassembly proceeds through three distinct strata—*terms*, *tiles*, and *shards*, ordered high to low as depicted in Fig. 2 and detailed in Sec. III. Disassembly to lower structures occurs when the user's selection boundaries cut across the linear span of the higher structure, thereby addressing the selection expressivity problem. For example, the depicted selection `(2 + 3) *` (middle left) reveals the containing term's disassembly into its constituent tiles; similarly, the selection `)` (lower left) reveals the containing tile's disassembly into its shards, i.e. its matching delimiters.

After insertion or deletion, syntactic obligations are generated to ensure eventual reassembly of any remaining lower structures. This bookkeeping is managed and presented by two independent subsystems operating at distinct levels of the structural strata.

1) The *backpack* scaffolds reassembly from shards to tiles by managing matching delimiter obligations, presenting these obligations in a pop-up stack attached to the cursor, as depicted in the lower right of Fig. 2.
2) The *grouter* scaffolds reassembly of tiles into terms, managing multiplicity obligations by inserting and removing *grout*. Grout are generated based on the requirements of neighboring tiles, which are shaped on either end with a concave or convex tip to indicate whether or not, respectively, delimits a child operand. Grout generalize holes to support both missing terms (<1) and adjacent terms (>1); for example, the upper right of Fig. 2 shows a convex piece of grout standing in for the missing operand `3`, as well as a concave piece of grout connecting the former operands of the missing operator `*`.

We have implemented tile-based editing in `teen tylr`[1], which is the source of all screenshots in this paper. After describing related work (Sec. II) and introducing the design of `teen tylr` in more detail (Sec. III), we present the results (Sec. V) of a lab study (Sec. IV) comparing text editing, MPS, and `teen tylr` on structurally complex program editing tasks. We found that the problems of selection expressivity, delimiter matching, and multiplicity helped explain the most common breakdowns participants encountered with MPS. We further observed that `teen tylr` resolved the selection expressivity and multiplicity problems, while our design of the backpack mitigates but incompletely resolves the matching problem. Nevertheless, we found that participants achieved competitive performance using `teen tylr` compared to text on most tasks and expressed largely positive sentiments toward its feature set. We conclude with a discussion of design implications for future code editors and parsers.

II. Related Work

`teen tylr` continues a long history of structure editor design, dating back to the Cornell Program Synthesizer in 1981 [18].

Much of this prior work has been on strictly tree-structured editing, which as discussed in Sec. I suffers from several problems rooted in the tension between textual projections and tree-structured editing. This approach nevertheless persists in present-day designs, notably including JetBrains MPS [3]. MPS is a state-of-the-art structure editor generator whose generated instances project the program tree to a caret-navigated textual form. Editor behavior is customizable, both by

[1] `teen tylr` is the successor to `tiny tylr`, which was introduced in a preliminary workshop paper [17]. This paper substantially refines that early design and presents new empirical evaluation.

$$
\begin{array}{rrcl}
\text{expression} & e & ::= & n \mid x \mid \texttt{(}e\texttt{)} \mid \texttt{[}e(\texttt{;}\ e)^{*}\texttt{]} > e\texttt{(}e\texttt{)} \\
& & > & e \texttt{ * } e \mid e \texttt{ / } e > e \texttt{ + } e \mid e \texttt{ - } e \\
& & > & e \texttt{ |> } e > e(\texttt{,}\ e)^{+} > \texttt{fun } p \texttt{ -> } e \\
& & \mid & \texttt{let } p \texttt{ = } e \texttt{ in } e \mid \texttt{if } e \texttt{ then } e \texttt{ else } e \\
\\
\text{pattern} & p & ::= & x > p(\texttt{,}\ p)^{+}
\end{array}
$$

Fig. 3: The concrete syntax of Camel, a simple expression-oriented language we designed for our lab study. Camel is a near-subset of OCaml expressions [20] and patterns [21]—the single deviation, postfix parentheses instead of infix space for function application, was to accommodate comparison with MPS, which has limited support for whitespace-based syntax. The operator $>$ indicates forms to its left have greater precedence than those to its right.

modifying the language grammar and by implementing hooks that modify the program tree when triggered, often for the purpose of easing linear interactions with the textual projection. These customizations can get quite varied and complex when implemented directly, especially when dealing with issues of operator precedence and associativity, so the most common editing patterns are codified in a domain-specific language called grammar cells [19].

This work presents a detailed comparison of `teen tylr` with an MPS editor configured with grammar cells (as well as a text editor). Prior work on structure editing focuses its efforts on statement-based languages, where the sequential syntactic structure of statement blocks alleviates some of the awkwardness of strictly tree-structured editing. The problem remains, however, at the typically expression-structured leaves of the program tree. While additional mitigations such as grammar cells exist, they are limited to an ad hoc collection of editing patterns—e.g. left-to-right insertion of single-token infix operator sequences—leading to the usability problems with selection and deletion described in Sec. I, and providing no assistance when modifying more complicated expression structures. The present work aims to fill these gaps. Specifically, in our study (Sec. IV), we configured both `teen tylr` and MPS to operate on an OCaml-like expression-oriented syntax we dub Camel, whose concrete syntax is given in Fig. 3, and evaluated them on program editing tasks involving complex hierarchies and modifications.

In this work, we focus on keyboard-driven text-like structure editing in order to capitalize on the literacy and keyboard skills of experienced programmers. However, mouse-driven block-based editors like Scratch [2], which are used mainly by novices and end-users, suffer from variants of the same problems described for keyboard-driven editors. Block outlines may provide visual justification for limited selection expressivity, but the limitation remains. In a user study of block-based editing involving large refactoring tasks [13], Holwerda and Hermans elicited post-task user responses on the cognitive dimensions [15] 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 (12), half (6) were about the difficulty of refactoring when the desired selection does not correspond to a complete term. These results suggest that problems like selection expressivity and delimiter matching remain important usability issues for structure editors, independent of the visualization scheme or input modality.

On the other hand, block-based editors sidestep the multiplicity problem because any number of disjoint structures may co-exist in their canvas-based interfaces, at the cost of inefficient mouse-driven interaction [22]. Some researchers have proposed but not evaluated designs for augmenting block-based editors with keyboard interaction [12] and extending caret navigation to arbitrarily arranged structures [23]. Block-based editors also suffer from low visual information density [13] because all blocks are shown at once and grow in size as they nest. In contrast, `teen tylr`'s decorations appear locally to the cursor and do not take up additional space beyond the underlying text.

Some structure editors [18, 24] employ hybrid editing models, using structural editing for large syntactic forms while deferring to text editing entirely at the leaves. This approach loses the benefits of structure editing at those levels, e.g. the gap problem must again be confronted by downstream tools. Moreover, while arbitrary text selections are possible within a text leaf, they cannot extend beyond those bounds and partially select any strictly structural forms.

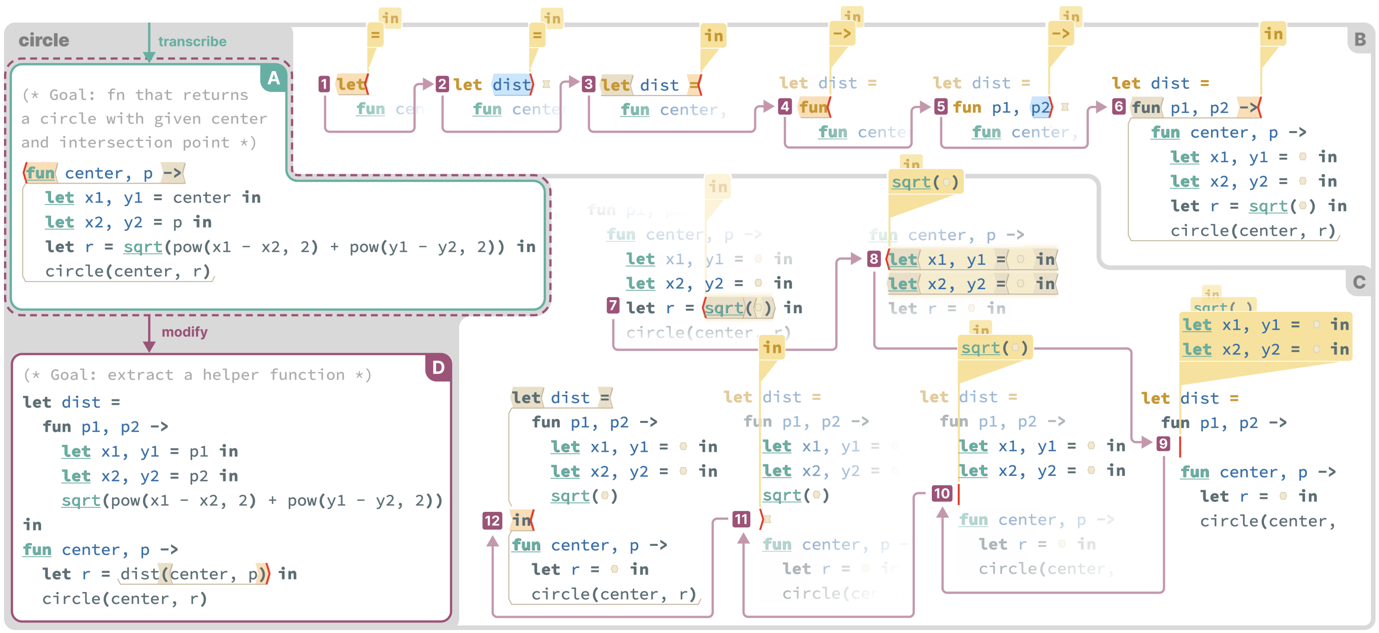
Gradual structure editing was conceptually inspired in part by gradual typing [25], which observes that users often want to leave a program partially well-typed, with holes in type position, in exchange for partial runtime feedback. Similarly, gradual structure editing leaves a program partially well-structured (e.g. when `teen tylr`'s backpack is non-empty or the program contains grout) to support partial semantic feedback. This work focuses on the usability aspects of gradual structure editing, leaving a full treatment of semantic aspects to future work. We continue discussing these ideas in Sec. VI, particularly in connection to error-handling parsers, which similarly navigate trade-offs between usability, structure, and semantics.

III. Design Overview

We now give an example-driven overview of tile-based editing using `teen tylr`. Fig. 4 shows a Camel program we asked our study participants to transcribe and subsequently modify, and depicts how one participant completed the modification. We will describe different parts of this edit sequence in more detail in this section as we introduce `teen tylr`'s features.

A. Terms, Tiles, Shards

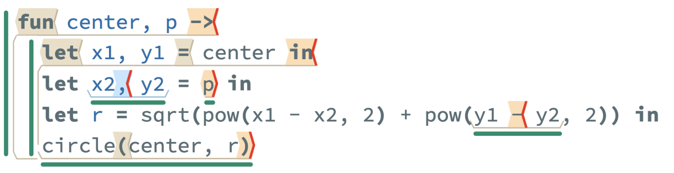
Fig. 5a depicts a composite of `teen tylr` edit states with its cursor in various positions. At each position, when there is nothing selected, `teen tylr` highlights the smallest containing term. Each term is made up of a set of matching *shards*, the term's hexagonally shaped delimiters, and the delimited child terms, outlined to the left or bottom depending on their layout.



```
(* Goal: fn that returns
a circle with given center
and intersection point *)
fun center, p ->
  let x1, y1 = center in
  let x2, y2 = p in
  let r = sqrt(pow(x1 - x2, 2) + pow(y1 - y2, 2)) in
  circle(center, r)
```

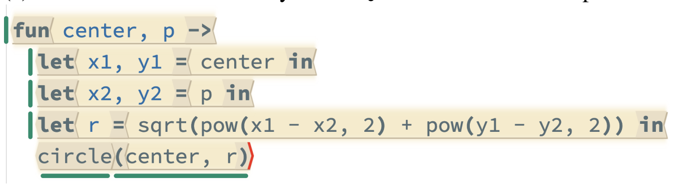
```
(* Goal: extract a helper function *)
let dist =
  fun p1, p2 ->
    let x1, y1 = p1 in
    let x2, y2 = p2 in
    sqrt(pow(x1 - x2, 2) + pow(y1 - y2, 2))
in
fun center, p ->
  let r = dist(center, p) in
  circle(center, r)
```

Fig. 4: A pair of editing tasks we assigned our lab study participants, consisting of a **transcription** task (Panel A) followed by a **modification** task (Panel D), and the edit sequence by which participant P9 completed the modification task using `teen tylr` (Panels B & C). Due to space constraints, the variable references `center` and `p` and the argument to `sqrt` in Panels A & D are elided in Panels B & C. In Panel B, P9 begins binding a new variable `dist` (B.1-3) to a newly inserted function taking arguments `p1` and `p2` (B.4-6). Subsequently, in Panel C, P9 selects and cuts the `sqrt` expression and the two preceding `let`-lines (C.7-9), pastes them above the original function (C.9-11), and completes the `let`-binding for `dist` with the concluding delimiter `in` (C.11-12). Finally, not shown, they modify the variable references `center` and `p` to `p1` and `p2` and inserted a call to the newly defined `dist` function to arrive at Panel D.



```
fun center, p ->
  let x1, y1 = center in
  let x2, y2 = p in
  let r = sqrt(pow(x1 - x2, 2) + pow(y1 - y2, 2)) in
  circle(center, r)
```

(a) Nested terms indicated by `teen tylr` at various cursor positions.



```
fun center, p ->
  let x1, y1 = center in
  let x2, y2 = p in
  let r = sqrt(pow(x1 - x2, 2) + pow(y1 - y2, 2)) in
  circle(center, r)
```

(b) A sequence of tiles produced by disassembling the function term.

Fig. 5: Terms (a) and tiles (b) annotated with green borders.

For example, the outermost indicated function term in Fig. 5a highlights its shards `fun` and `->` and outlines its argument pattern and body.

Selecting the function term reveals its disassembly into a sequence of *tiles*, shown in Fig. 5b. Tiles are, collectively, in one-to-one correspondence with terms: each tile consists of the term's shards and the children they bidelimit (delimit on both left and right). For example, the function tile includes the shards `fun` and `->` and the bidelimited pattern `center, p` but does not include the function body, because it is not delimited on the right. Instead, the body's tiles are simply adjacent to the function header in the tile sequence.

Tiles model unassociated operator sequences, where each tile is shaped at its tips to indicate whether its an ⟨operand⟩, ⟨prefix⟨ operator, ⟩postfix⟩ operator, or ⟩infix⟨ operator. While contemporary structure editors like MPS make use of this sequential structure to ease linear insertion, they do so emphemerally, such that the user cannot subsequently select arbitrary subsequences after insertion. `teen tylr` resolves this limitation by directly manifesting the sequential structure in the edit state as needed. Edit state C.8 in Fig. 4 shows P9 using this capability to select the two `let`-bindings without their concluding body.

B. *Terms* $\rightleftharpoons$ *Tiles + Grout*

Via operator-precedence parsing, a sequence of tiles reassembles into a valid term if and only if the tiles fit together sequentially 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, `teen tylr` is equipped with a scaffolding system we dub the grouter. 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.

Convex grout succeed the familiar concept of *holes* in traditional structure editing, i.e. they handle the situation where a term is expected but *none* is found. For example, when P9 selects and cuts the `sqrt` expression in C.7-8 of Fig. 4, the grouter leaves behind a convex grout piece in its place. Meanwhile, concave grout handle the situation when *more than one* is found, thereby addressing the multiplicity problem. For example, when P9 pastes the `sqrt` expression in C.10-11, the grouter inserts a concave grout piece to temporarily buffer the two terms on either side, which P9 subsequently replaces with the `in` shard in C.12. Concave grout also make it straightforward to define minimal, local deletions: recall how, in the upper right of Fig. 2, `teen tylr` is able to preserve both orphaned children `(2 + 3)` and `4` upon removing their parent `*`, unlike MPS which could save at most one.

C. Tiles ⇌ Shards + Backpack

Tiles may be further disassembled into their constituent shards, whose subsequent reassembly is guided by a second system called the *backpack*. The backpack succeeds the familiar clipboard, but extended in a few distinct ways. Two differences are most immediate. First, it is visible—edit state C.11 in Fig. 4 shows how it appears as a yellow "balloon" tied to the cursor. Second, it can carry multiple items, organized into a stack—for example, edit states B.1-6 show how the backpack grows and shrinks as P9 inserts shard-by-shard, while C.7-9 show how P9 used the backpack to pick up both the `sqrt` expression as well as the two preceding `let`-bindings.

The backpack is additionally co-managed by `teen tylr` to ensure that tiles are well-nested, and that freshly inserted shards are not left unmatched. Consider the edit sequence in Panel B of Fig. 4. When P9 inserts the first shard `let` (B.1), `teen tylr` recognizes it as a keyword and populates the backpack with its matching shards `=` and `in`. After typing `dist` (B.2), P9 puts down the head shard `=` (B.3) and goes on to insert the function tile (B.4-6), again supported by the backpack, emerging carrying the final obligatory shard `in`. When they subsequently move inside another tile to select the `sqrt` expression in C.7, the backpack turns transparent and inactive, indicating that it would be structurally invalid to put down the `in` there.

This last example in C.7 reveals an overly conservative limitation of our current tile-based editing ontology, which is that a tile's shards are permanently matched and cannot be exchanged with another tile's shards, even a tile of the same form. One could imagine a variation of our system that permits pasting the `in`-shard in C.7, matching it to `let r =`, while the `in`-shard originally matched to `let r =` is re-matched to `let dist =`. Our study revealed that this indeed posed a significant remaining usability problem (Sec. V-B), and we discuss lifting this limitation in Sec. VI.

IV. Lab Study

We sought to empirically investigate the effect of the selection, multiplicity, and matching problems in a term-based editor, as well as the impact on user experience when those problems are mitigated in a tile-based editor. To do so, 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 configured with MPS; and `teen tylr`. We had the following questions:

Q1 When first-time users attempt structurally complex modifications with a term- or tile-based editor, how does editor choice impact completion time, mental load, and code reuse, relative to a text editor?

Q2 What mistakes and inefficiencies do first-time users experience with a term- or tile-based editor? What aspects do they find most frustrating?

Q3 What aspects of term- and tile-based editing do first-time users find most empowering or appealing?

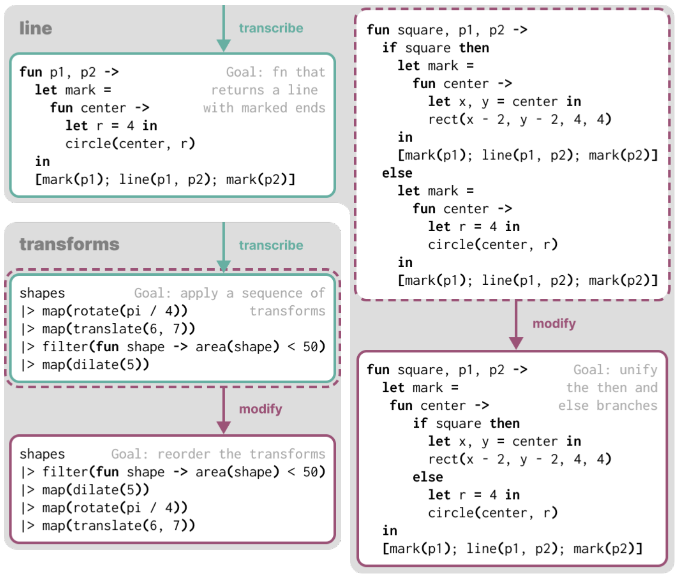
A. Participants

Because our study tasks involved editing programs written in an expression-oriented language, we sought participants with some prior experience with such languages. We recruited 10 participants P0-P9 (8 male, 1 female, 1 unstated; ages 19-31 years, median 23.5 years) by posting on Twitter and subreddits for OCaml (r/ocaml) and the authors' institution, as well as emailing students recently enrolled in the undergraduate programming languages course there. Most participants reported substantial experience with expression-oriented languages (0.3-10 years, median 3.5 years). Eight participants had some prior experience with structured editing interfaces, such as Scratch [2] and ParEdit [26]; two of the eight had designed and implemented their own structure editors. Each participant was compensated $40 for a 90-minute session of study tasks followed by a 10-minute exit survey.

B. Tasks & Editors

The study tasks involved small program editing tasks in Camel, the expression-oriented language introduced in Fig. 3. Fig. 4 and Fig. 6 together show the six editing tasks participants completed with each editor, organized into three pairs: `circle`, `line`, and `transforms`. Each pair consisted of a **transcription** task, where the participant transcribed a Camel program from scratch; followed by a **modification** task, where the participant modified their transcribed program (or variation thereof in the case of `line`) to a given goal program. We designed our modification tasks to involve complex 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.

We asked participants to complete the tasks with three different editors: VS Code, JetBrains MPS, and `teen tylr`. We configured VS Code to syntax-highlight Camel code and otherwise disabled other extensions. We configured MPS with



```
fun p1, p2 ->                    Goal: fn that
  let mark =                      returns a line
    fun center ->               with marked ends
      let r = 4 in
      circle(center, r)
  in
  [mark(p1); line(p1, p2); mark(p2)]
```

```
shapes            Goal: apply a sequence of
|> map(rotate(pi / 4))             transforms
|> map(translate(6, 7))
|> filter(fun shape -> area(shape) < 50)
|> map(dilate(5))
```

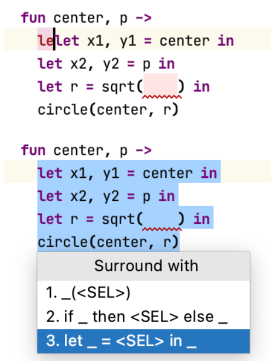
```
shapes          Goal: reorder the transforms
|> filter(fun shape -> area(shape) < 50)
|> map(dilate(5))
|> map(rotate(pi / 4))
|> map(translate(6, 7))
```

```
fun square, p1, p2 ->
  if square then
    let mark =
      fun center ->
        let x, y = center in
        rect(x - 2, y - 2, 4, 4)
    in
    [mark(p1); line(p1, p2); mark(p2)]
  else
    let mark =
      fun center ->
        let r = 4 in
        circle(center, r)
    in
    [mark(p1); line(p1, p2); mark(p2)]
```

```
fun square, p1, p2 ->         Goal: unify
  let mark =                  the then and
    fun center ->             else branches
      if square then
        let x, y = center in
        rect(x - 2, y - 2, 4, 4)
      else
        let r = 4 in
        circle(center, r)
  in
  [mark(p1); line(p1, p2); mark(p2)]
```

Fig. 6: Transcription-modification task pairs `line` and `transforms`. See Fig. 4 for the third pair `circle`.

its grammar cells system (described in Sec. II) to operate on Camel programs, as shown in screenshots on the left.



```
fun center, p ->
  lelet x1, y1 = center in
  let x2, y2 = p in
  let r = sqrt(   ) in
  circle(center, r)

fun center, p ->
  let x1, y1 = center in
  let x2, y2 = p in
  let r = sqrt(   ) in
  circle(center, r)
```

Grammar cells enable some familar linear editing patterns, e.g. one may type the keyword `let` as depicted in the top half to insert a new `let`-binding above the others. Other operations, such as binding an existing expression to a new variable, require selecting the expression and invoking MPS's "Surround with" menu as depicted in the bottom half; alternatively, one may cut the selection, insert the `let`-expression, and paste back the selection at the desired location. Our implementation of `teen tylr` at the time of the study largely followed our presentation in Sec. III, except that we had not fully implemented the visual design for selections: selected ranges were highlighted without showing their internal make-up of tiles and shards. Moreover, at the time, `teen tylr` did not support mouse input. We discuss the potential impact of these limitations in Sec. V-D.

C. Procedure

We conducted the study remotely and recorded participants' screens. Each session consisted of three components, one for each editor. Each component consisted of a 10-minute tutorial portion followed by the three task pairs in a randomized order.

Every participant started with the VS Code component. We used the tutorial portion of this component to introduce participants to Camel and to verify they understood its term structure before proceeding to the structure editing components. Specifically, as we introduced Camel's syntax, we asked participants to parenthesize all subterms in a sample program that included all of Camel's syntactic forms. While this imposed learning effects on task performance in the subsequent components, there is a strong counteracting effect in the much greater familiarity and skill participants had with text editing compared to MPS and `teen tylr`.

Participants were evenly split between different orders for the subsequent MPS and `teen 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 the different approaches to inserting and deleting expression forms as supported by grammar cells or otherwise using the "Surround With" menu. The `teen tylr` tutorial additionally noted the backpack's enforcement of permanent shard matching.

We sought to distinguish the time spent figuring out how to complete a task from the time spent performing the edits, as well as minimize mistakes caused by misunderstanding of program structure, so we asked participants to plan their edits before completing each task—this preparation time additionally served as an objective complement to their subjective reports of mental load. When preparing for modification tasks, participants were encouraged to make selections in the starting code to verify their understanding of selectable structures. We suggested participants take up to 1 minute to prepare for transcription tasks and up to 2 minutes for modification tasks, but did not enforce these limits. We asked participants to complete each task as quickly and accurately as they comfortably could.

After each component, participants were asked to reflect on their experience with the editor and to compare it with any previous editors. Finally, after completing all components, participants completed a 10-minute exit survey.

V. Results

The lab study, administered by the second author, produced roughly 15 total hours of screen-recorded video. The first author segmented and reviewed the preparation and task portions (4 hours) of these recordings in detail to study participants' editing patterns, infer high-level intent, and identify mistakes. Preparation time was measured as starting when participants started reading each task description and ending when they said they were ready to begin, minus any time spent on dialogue (e.g. to ask clarifying questions); task completion time was measured as starting when the administrator gave a cue to begin and ending when the participant said they were done. Intent was rarely ambiguous given both knowledge of the editor clipboard state, always preceded by a visible selection, as well as the highly constrained nature of the tasks. Mistakes were identified by subsequent backtracking via undo and voiced expressions of uncertainty or regret.

Section V-A summarizes our quantitative results. After noting likely effects and patterns, we elaborate on specific causes in Sec. V-B and V-C.

A. Completion Times, Mental Load, Code Reuse (Q1)

Given known limitations of null hypothesis significance testing [27], we base our analyses of time measures on

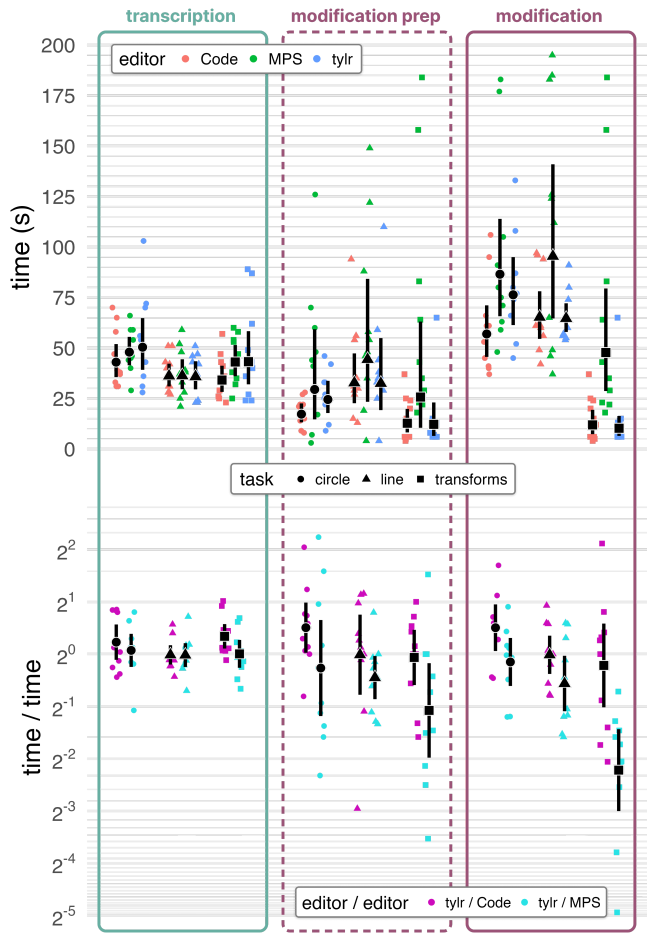


Fig. 7: Dot plots overlaid with 95% confidence intervals summarizing how long participants took to prepare for and complete tasks with each editor. Each dot represents an individual participant measure. The top half shows the raw task times; the bottom half shows the relative slowdowns/speedups participants exhibited on each task using `teen tylr` compared to the other two editors. Confidence was calculated with the log of both measures to correct for positive skew.

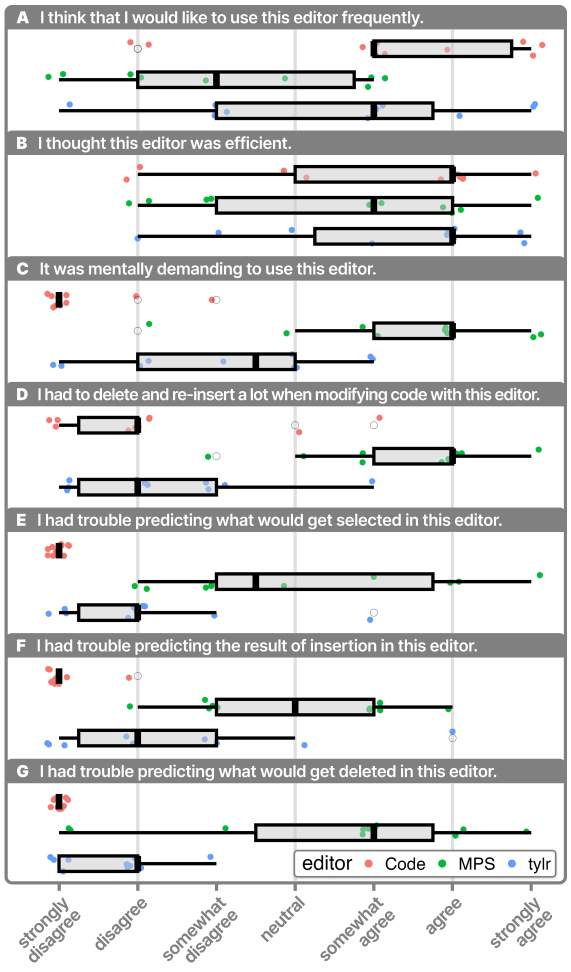


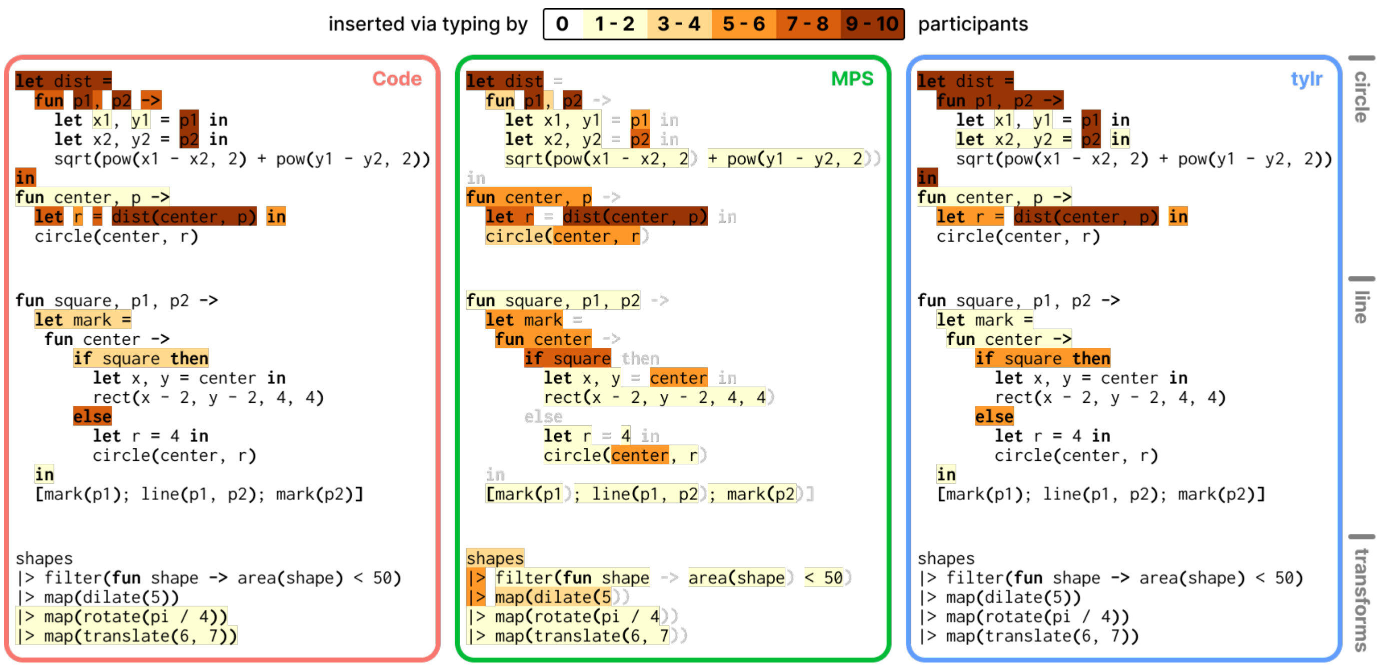
Fig. 8: Box plots summarizing post-task survey responses.

estimated effect sizes with confidence intervals [28]. Fig. 7 summarizes how long participants took to prepare for and complete the tasks with each editor, and how these repeated measures compare as ratios between `teen tylr` and the other two editors respectively. P1 encountered a crash when using `teen tylr` to modify the `circle` program, so we discarded this measure and its corresponding ratios. We also omitted transcription preparation times because participants generally took no additional time beyond reading the task description.

We observed quite similar transcription performance between editor pairs (`tylr`/Code and `tylr`/MPS) across the three tasks: all six estimates fall within relatively precise bounds ($[0.8, 1.5]$), with all but one (`tylr`/Code on `transforms`) overlapping with equal performance. These results are unsurprising given that all three editors facilitate familiar left-to-right transcription flows.

On the other hand, our results show that the choice of editor had clear impact on modification preparation and performance, despite more imprecise estimates. In the case of `tylr`/Code, our estimates suggest some possible slowdown in both preparation ($1.42, [1.02, 1.98]$) and performance ($1.42, [1.04, 1.93]$) for `circle`, while remaining inconclusive as to the effect direction for `line` and `transforms`. Meanwhile, our estimates for `tylr`/MPS suggest speedups in both preparation and performance on `line` ($0.73, [0.55, 0.97]$ and $0.68, [0.47, 0.98]$) and `transforms` ($0.47, [0.25, 0.89]$ and $0.21, [0.12, 0.37]$), especially the latter.

These patterns were mirrored in participants' subjective responses to our post-task survey, summarized in Fig. 8. Plot B shows that median participants felt that all three editors were at least somewhat efficient—perhaps due to the similar transcription performance across editors—though with more disagreement in the case of MPS. On the other hand, correlating to the observed effects on preparation, Plot C shows they felt that `teen tylr` was much less mentally demanding to use than MPS, while still more demanding than VS Code. These patterns persist across their opinions about the predictability of different edit operations (Plots E-G) and how much they felt they had to delete and re-insert code when modifying (Plot D),



```
let dist =
  fun p1, p2 ->
    let x1, y1 = p1 in
    let x2, y2 = p2 in
    sqrt(pow(x1 - x2, 2) + pow(y1 - y2, 2))
in
fun center, p ->
  let r = dist(center, p) in
  circle(center, r)


fun square, p1, p2 ->
  let mark =
   fun center ->
      if square then
        let x, y = center in
        rect(x - 2, y - 2, 4, 4)
      else
        let r = 4 in
        circle(center, r)
  in
  [mark(p1); line(p1, p2); mark(p2)]


shapes
|> filter(fun shape -> area(shape) < 50)
|> map(dilate(5))
|> map(rotate(pi / 4))
|> map(translate(6, 7))
```

```
let dist =
  fun p1, p2 ->
    let x1, y1 = p1 in
    let x2, y2 = p2 in
    sqrt(pow(x1 - x2, 2) + pow(y1 - y2, 2))
in
fun center, p ->
  let r = dist(center, p) in
  circle(center, r)


fun square, p1, p2 ->
  let mark =
   fun center ->
      if square then
        let x, y = center in
        rect(x - 2, y - 2, 4, 4)
      else
        let r = 4 in
        circle(center, r)
  in
  [mark(p1); line(p1, p2); mark(p2)]


shapes
|> filter(fun shape -> area(shape) < 50)
|> map(dilate(5))
|> map(rotate(pi / 4))
|> map(translate(6, 7))
```

```
let dist =
  fun p1, p2 ->
    let x1, y1 = p1 in
    let x2, y2 = p2 in
    sqrt(pow(x1 - x2, 2) + pow(y1 - y2, 2))
in
fun center, p ->
  let r = dist(center, p) in
  circle(center, r)


fun square, p1, p2 ->
  let mark =
   fun center ->
      if square then
        let x, y = center in
        rect(x - 2, y - 2, 4, 4)
      else
        let r = 4 in
        circle(center, r)
  in
  [mark(p1); line(p1, p2); mark(p2)]


shapes
|> filter(fun shape -> area(shape) < 50)
|> map(dilate(5))
|> map(rotate(pi / 4))
|> map(translate(6, 7))
```

Fig. 9: Heat maps summarizing code reuse in the modification tasks, measured by the number of participants that inserted via typing, rather than cutting and pasting, each token in the goal state. Delimiters auto-inserted by MPS are excluded.

i.e. how much difficulty they had reusing existing code. Plot A suggests that participants overall preferred using VS Code and `teen tylr` over MPS, but with wide divergence in opinions in the case of `teen tylr`.

Fig. 9 complements participants' subjective responses about code reuse with objective measures on the modification tasks. We observed quite similar patterns of reuse between `teen tylr` and VS Code, though with the subtle difference that reuse of matching shards in `teen tylr` are strictly correlated due to the backpack enforcing lifelong matching (as discussed in Sec. III-C). We also observed overall less reuse with MPS than the other two editors. P0 opted not to reuse any starting code for `line` with MPS, instead deleting it and transcribing the goal state; P8 did the same for `transforms`. Some participants were forced to rewrite variable references like `center` in `line` on account of MPS's strict binding requirements, which would lead to variable references disappearing from the clipboard when they deleted the original binding sites. Other notable differences include the concluding function in `circle`; and two of the pipe operators in `transforms`, as well as the first and third operands in the pipeline.

B. Mistakes, Inefficiencies, Frustrations (Q2)

The results of Sec. V-A indicate that participants struggled with MPS because of how mentally demanding it was to perform complex modifications, with notable impacts on both task performance and code reuse even after an initial planning phase. P2 wrote, "MPS was EXTREMELY cognitively demanding to me; it felt like I was solving tree-manipulating puzzles the entire time I used it." We found that the most common mistakes and frustrations with MPS related to the three usability problems we described in Sec. I.

The selection expressivity problem was mitigated by the preparation phases of our study procedure, but there remained some cases where participants began modification tasks with mistaken interpretations of the initial expression structure. For example, P0, P7, and P8 started modifying `transforms` thinking they could select individual lines, only to discover their selections rounded up to the nearest prefix of lines. Five participants (P1, P3, P5, P7, P9) mentioned inexpressive selections when asked about frustrating aspects of MPS.

The most common class of mistakes related to the delimiter matching problem, in particular when inserting and deleting multifix forms such as `let _ = _ in _` and `if _ then _ else _`. When inserting, participants frequently misremembered whether to rely on the side-wrapping behavior via grammar cells or to use the Surround With menu, forcing them subsequently to amend the mistake by either backtracking and applying the alternative action, or else cutting and pasting the miswrapped child into its proper slot. We observed three participants (P0-1, P9) make this mistake when modifying `circle`, six (P2-4, P5-7, P9) when modifying `line`. Similarly, when deleting, some participants misremembered whether to rely on the side-unwrapping behavior of MPS's grammar cells, leading to accidental overdeletion of a bidelimited child; we observed three participants (P5, P7, P9) make this mistake when modifying `line`. Four participants (P1, P4-5, P7) referred to these delimiter matching issues as most frustrating.

Some avoided these matching-related decisions by conservatively stashing the child to be wrapped/unwrapped in the clipboard before inserting/deleting, a more uniform but less efficient manuever when side-wrapping/unwrapping is possible. These maneuvers led three participants (P1, P4-5) to liken the experience to the Towers of Hanoi, a puzzle commonly used to exercise recursive problem solving.

Meanwhile, the multiplicity problem led to breakdowns when pasting content with MPS, where pasting to the left or right of a term overwrites it. While we explicitly noted this behavior in our tutorial, it still occasionally led to surprises. For example, when modifying `transforms`, P7 cut the expression spanning the first three lines of the start state, re-inserted `shapes` in its place, and pasted the cut expression at the end of the last transformation (`map(dilate(5))`), unintentionally overwriting the transformation and having to re-insert it. Overall we observed two participants (P0, P9) make this mistake when modifying `circle`, one (P7) when modifying `line`, and four (P0-1, P4, P7) when modifying `transforms`.

Others were more cautious about this overwriting behavior, but instead reported that taking this care was mentally taxing, especially given the combined pressures of the multiplicity and matching problems on the clipboard. P2 wrote of MPS: "the worst part by far is the lack of 'scratch' workspace; in VS Code I can keep syntactically invalid code in the file, and in tylr I can keep it in the backpack. In MPS I could only keep it in a single clipboard or some ad hoc location in the AST." As a result, the most successful participants were those who adopted a general strategy of inserting before deleting in order to expand their available scratch space for subsequent clipboard-dependent modifications.

Participants using `teen tylr` did not experience the same scarcity of scratch space as with MPS: its flexible selections reduced the number of selections necessary, concave grout made it possible to paste without overwriting, and the backpack accommodated any number of remaining selection needs.

On the other hand, several had trouble with the backpack enforcing well-nested and permanent shard matching, as described in Sec. III-C. For example, P0 started modifying `line` by selecting the third and fourth lines `let mark = \n fun center ->` and attempting to paste them above the second line `if square then`, but could not on account of the remaining `in` shard left behind in the `then`-branch—completing such a maneuver requires cutting the `in` shard as well before pasting. Despite this being mentioned in our tutorial, six participants (P0-1, P3, P5, P7, P9) encountered this issue when modifying `line`, one (P4) when modifying `circle`, and only one (P3 on `line`) successfully proceeded by picking up additional shards rather than backtracking and re-strategizing. Seven participants (P0-2, P4-5, P7-8) mentioned this issue when asked what was most frustrating about `teen tylr`.

This issue could get particularly confusing or frustrating when rearranging shards of tiles of the same form, especially given the lack of visual distinction (e.g. using color) between the different pairings. Some felt that this ran counter to their preferred workflows with text—for example, P2 mentioned, "I commonly 'reuse' if/else tokens from nested/sequential if expressions by deleting, say, the else branch of the first one and the then branch of the second one."—which we also observed in code reuse patterns for VS Code in Fig. 9. These issues, combined with `teen tylr`'s otherwise text-like experience, led P4 to describe it as feeling like "an uncanny valley between structured editing and text... `teen tylr` mostly felt like a text editor... but it also isn't quite text—fixing parenthesization errors required that I think in terms of structure."

C. Empowering or Appealing (Q3)

Despite these issues with the backpack, several participants expressed positive sentiment toward its other aspects when asked about empowering or appealing aspects of `teen tylr`. P0 wrote, "The backpack was pretty cool, and I definitely think it's something I would make use of." P2 enjoyed not needing to "worry about how to keep things in 'scratch' space like in MPS (which is huge, to me)". P9 appreciated the ability to "cut multiple things at once and then paste it later, saving me time and context switch of going back and forth to copy and paste." A few participants expressed wishing to have the backpack available in MPS.

Others liked `teen tylr`'s visual design and grouting system based on convex/concave tips. P7 enjoyed the "curved cursor", P1 the "arms on operators, concave and convex carets(!)". P2 appreciated having "the benefit of automatic hole insertion (which is also great)", while P4 enjoyed the "friendly feedback" of the grouting system: "Seeing unexpected placeholders sometimes pointed out a syntax mistake I had made, in a more pleasant way than a traditional squiggly red underline."

When asked about empowering or appealing aspects of MPS, four participants (P1-4) expressed appreciation for the efficiency of selection when it aligned with their goals. Another four (P1, P4-5, P9) cited the ability to jump to empty holes using the Tab key (which had not yet been implemented in `teen tylr` at the time of the study). Others (P1, P3-4, P6-7) appreciated the way MPS managed details like inserting matching delimiters and formatting whitespace (although others (P0, P8) disliked this lack of control). P1 appreciated how these features added up to an overall "clicky" experience.

D. Limitations

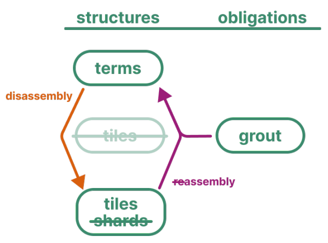
Our study had several limitations. Our results were affected by `teen tylr`'s prototypal nature: for example, unlike the other two editors, `teen tylr` did not support mouse input at the time so we asked participants to limit themselves to keyboard input, at which some expressed unfamiliarity in the case of selection. Participants had only 30 minutes to get introduced to and complete tasks with MPS and `teen tylr` respectively, so our results do not reflect optimal performance or behavior, but rather trends and obstacles in first-time use. The editing tasks were few, small, and synthetic, particularly the fact that participants were expected to reach a given goal state verbatim and asked to plan their edits before performing them—while these constraints made it possible for us to make detailed comparisons of the editors, they deviate from typical editing practices in the way they split user attention between the goal and the edit state, as well as prevent natural interleaving of problem solving, planning, and editing. Because all participants started with the VS Code component and completed the same tasks in every component, learning effects of the tasks impact our comparisons between VS Code and the other two editors—on the other hand, they are counteracted by participants' vastly greater experience with text editing. Finally, participants were aware that the authors had designed and implemented `teen tylr`, which is known to contribute to response bias [29].

It is possible to engineer more ergonomic structure editors with MPS than the one we built and evaluated in this study by adjusting the language grammar to improve selection expressivity. Our editor directly implemented the expression structure of Camel, 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.

VI. Discussion and Future Work

While our study was small and not necessarily reflective of more proficient use, we think it contributes new detail and insight into the general problem of structure editor usability, especially in the context of keyboard-driven editing of nested expression structures. Our decomposition of the problem into selection expressivity, multiplicity, and delimiter matching proved useful in explaining common breakdowns our participants encountered when performing complex modification tasks with MPS. Particularly interesting was the interaction between the multiplicity and matching problems, which in their competing demands for limited clipboard space led to less code reuse and greater mental load with MPS. This phenomenon suggests that traditional keyboard-driven structure editors, whether gradual or not, might substantially improve usability simply by increasing the number of available slots in the clipboard system. Our study additionally highlighted the limitations of MPS's grammar cells system, particularly in the way they bifurcated insertion and deletion flows for larger mixfix forms common in expression-oriented languages like OCaml.

Our study further suggested that our design of `teen tylr` successfully mitigated most but not all of these issues, leading to improved modification performance, greater code reuse, and lower reports of mental load compared to MPS. Our participants expressed appreciation for `teen tylr`'s greater selection flexibility, the scratch space made available by the backpack, and its overall visual indications of expected structure via convex/concave tips and grout. On the other hand, several were surprised and confused by the permanent matching of shards, which stood in the way of delimiter re-matching workflows they undertook with VS Code. Some also wished for a more structured feel to the editing experience, which we attribute to lacking features at the time of the study, such as system-managed whitespace and tabbing to holes, rather than any fundamental limitation of our design. `teen tylr` is so named because of these remaining limitations, which we construe as an awkward adolescent phase in its evolution from strict structure editing to increasingly text-like editing.



Lifting these limitations in ongoing work has led us to simplify the tile-based editing ontology, from the one in Fig. 2 to something closer to the one on the left. Here, we rid ourselves of the previous concept of tiles as intermediate structures of matching shards, and rename shards as tiles in order to dispel the physical metaphor that they are fractured components of a particular parent entity and should be reassembled as such. Meanwhile, the grouter subsumes the role previously handled by the backpack, such that grout elements represent both multiplicity and delimiter matching obligations—in the latter case, they would be additionally decorated with transparent text of the missing delimiters. The backpack may continue to serve as a visual clipboard stack given its positive feedback, but would no longer be system-managed or relevant to maintaining structural integrity. This new framing seems suspiciously close to regular text parsing, which raises the question: are we back where we started? Should we have saved our efforts on the notorious impracticalities of structure editing and instead started with the well-established methods of text parsing?

We think our design efforts in the structure editing realm provide unique guidance for future parser and editor designs, in ways not emphasized by current parsing literature. Error-handling parsers are kin to structure editors in their aim to maintain continuous maximal structure for downstream analyses and editor services, but typically define the problem starting from simple blackbox assumptions about the textual interfaces they inhabit, leading to major shortcomings in user experience. Error-handling methods consider strictly textual corrections, of which there can be many possibilities even of minimal size—the burden of choice is then passed on to the programmer [30], or otherwise one is chosen using ad hoc heuristics [31]. Morever, if a heuristic choice is made, it is typically invisible to the programmer, leaving them only indirect clues in the behavior of downstream editor services. The missing piece, we believe, is a complementary system of user-facing obligations, much like the one developed in this work, that can stand in for many possible completions while explicitly scaffolding nearby structures. Such a system would be easily integrated with the graphical capabilities of modern text-based IDEs.

Another aspect of `teen tylr`'s design not emphasized in current parsing literature is the maximal assembly and visualization of gradual structures. While there exists work on incremental parsing [32], it is only incremental in the sense that it isolates errors, while parsed structures exist only at the granularity of complete AST nodes. In ongoing and future work, we aim to develop general parsing methods for gradual structures that can be used to structure and visually organize arbitrary user selections and edit states.

References

[1] C. Omar, I. Voysey, M. Hilton, J. Sunshine, C. Le Goues, J. Aldrich, and M. A. Hammer, "Toward semantic foundations for program editors," in *Summit on Advances in Programming Languages (SNAPL)*, 2017.

[2] J. Maloney, M. Resnick, N. Rusk, B. Silverman, and E. Eastmond, "The Scratch programming language and environment," *ACM Trans. Comput. Educ.*, vol. 10, no. 4, Nov. 2010. [Online]. Available: https://doi.org/10.1145/1868358.1868363

[3] M. Voelter and V. Pech, "Language modularity with the MPS language workbench," in *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, M. Glinz, G. C. Murphy, and M. Pezzè, Eds. IEEE Computer Society, 2012, pp. 1449–1450. [Online]. Available: https://doi.org/10.1109/ICSE.2012.6227070

[4] L. R. Neal, "Cognition-sensitive design and user modeling for syntax-directed editors," *SIGCHI Bull.*, vol. 18, no. 4, p. 99–102, May 1986. [Online]. Available: https://doi-org.proxy.lib.umich.edu/10.1145/1165387.30866

[5] D. R. Goldenson and M. B. Lewis, "Fine tuning selection semantics in a structure editor based programming environment: Some experimental results," *SIGCHI Bull.*, vol. 20, no. 2, p. 38–43, Oct. 1988. [Online]. Available: https://doi-org.proxy.lib.umich.edu/10.1145/54386.54400

[6] M. L. Van De Vanter, "Practical language-based editing for software engineers," in *Software Engineering and Human-Computer Interaction*, R. N. Taylor and J. Coutaz, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 251–267.

[7] B. Lang, "On the usefulness of syntax directed editors," in *Proceedings of an International Workshop on Advanced Programming Environments*. Berlin, Heidelberg: Springer-Verlag, 1986, p. 47–51.

[8] R. Bahlke and G. Snelting, "Design and structure of a semantics-based programming environment," *International Journal of Man-Machine Studies*, vol. 37, no. 4, pp. 467–479, 1992, structure-based editors and environments. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0020737392900056

[9] S. Minör, "Interacting with structure-oriented editors," *Int. J. Man Mach. Stud.*, vol. 37, no. 4, pp. 399–418, 1992. [Online]. Available: https://doi.org/10.1016/0020-7373(92)90002-3

[10] M. Voelter, J. Siegmund, T. Berger, and B. Kolb, "Towards user-friendly projectional editors," in *Software Language Engineering*, B. Combemale, D. J. Pearce, O. Barais, and J. J. Vinju, Eds. Cham: Springer International Publishing, 2014, pp. 41–61.

[11] P. Miller, J. Pane, G. Meter, and S. A. Vorthmann, "Evolution of novice programming environments: The structure editors of Carnegie Mellon University," *Interact. Learn. Environ.*, vol. 4, no. 2, pp. 140–158, 1994. [Online]. Available: https://doi.org/10.1080/1049482940040202

[12] J. Monig, Y. Ohshima, and J. Maloney, "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)*, ser. BLOCKS AND BEYOND '15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 51–53. [Online]. Available: http://dx.doi.org/10.1109/BLOCKS.2015.7369001

[13] R. Holwerda and F. Hermans, "A usability analysis of blocks-based programming editors using cognitive dimensions," in *2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 2018, pp. 217–225.

[14] D. Bau, J. Gray, C. Kelleher, J. Sheldon, and F. Turbak, "Learnable programming: Blocks and beyond," *Commun. ACM*, vol. 60, no. 6, p. 72–80, May 2017. [Online]. Available: https://doi-org.proxy.lib.umich.edu/10.1145/3015455

[15] T. Green and M. Petre, "Usability analysis of visual programming environments: A 'cognitive dimensions' framework," *Journal of Visual Languages and Computing*, vol. 7, no. 2, pp. 131–174, 1996. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1045926X96900099

[16] T. Berger, M. Völter, H. P. Jensen, T. Dangprasert, and J. Siegmund, "Efficiency of projectional editing: A controlled experiment," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016. New York, NY, USA: ACM, 2016, pp. 763–774. [Online]. Available: http://doi.acm.org/10.1145/2950290.2950315

[17] D. Moon, A. Blinn, and C. Omar, "tylr: A tiny tile-based structure editor," in *Proceedings of the 7th ACM SIGPLAN International Workshop on Type-Driven Development*, ser. TyDe 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 28–37. [Online]. Available: https://doi.org/10.1145/3546196.3550164

[18] T. Teitelbaum and T. Reps, "The Cornell program synthesizer: A syntax-directed programming environment," *Commun. ACM*, vol. 24, no. 9, pp. 563–573, Sep. 1981. [Online]. Available: http://doi.acm.org/10.1145/358746.358755

[19] M. Voelter, T. Szabó, S. Lisson, B. Kolb, S. Erdweg, and T. Berger, "Efficient development of consistent projectional editors using grammar cells," in *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering*, ser. SLE 2016. New York, NY, USA: ACM, 2016, pp. 28–40. [Online]. Available: http://doi.acm.org/10.1145/2997364.2997365

[20] "The OCaml language: Expressions," https://v2.ocaml.org/manual/expr.html, accessed: 2023-07-01.

[21] "The OCaml language: Patterns," https://v2.ocaml.org/manual/patterns.html, accessed: 2023-07-01.

[22] N. C. C. Brown, M. Kolling, and A. Altadmri, "Position paper: Lack of keyboard support cripples block-based programming," in *2015 IEEE Blocks and Beyond Workshop (Blocks and Beyond)*, 2015, pp. 59–61.

[23] E. Evans, "A caret for your thoughts: Adapting caret navigation to visual editors," https://2023.programming-conference.org/details/px-2023-papers/1/A-Caret-for-Your-Thoughts-Adapting-Caret-Navigation-to-Visual-Editors, accessed: 2023-05-05.

[24] M. Kölling, "The Greenfoot programming environment," *ACM Trans. Comput. Educ.*, vol. 10, no. 4, Nov. 2010. [Online]. Available: https://doi.org/10.1145/1868358.1868361

[25] J. G. Siek, M. M. Vitousek, M. Cimini, and J. T. Boyland, "Refined criteria for gradual typing," in *1st Summit on Advances in Programming Languages, SNAPL 2015, May 3-6, 2015, Asilomar, California, USA*, ser. LIPIcs, T. Ball, R. Bodík, S. Krishnamurthi, B. S. Lerner, and G. Morrisett, Eds., vol. 32. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2015, pp. 274–293. [Online]. Available: https://doi.org/10.4230/LIPIcs.SNAPL.2015.274

[26] T. R. Campbell, "paredit — parenthetical editing in emacs," https://paredit.org/, accessed: 2023-05-08.

[27] G. Cumming, "The new statistics: Why and how," *Psychological Science*, vol. 25, no. 1, pp. 7–29, 2014, pMID: 24220629. [Online]. Available: https://doi.org/10.1177/0956797613504966

[28] G. Cumming and S. Finch, "Inference by eye: Confidence intervals and how to read pictures of data," *The American Psychologist*, vol. 60, pp. 170–80, 02 2005.

[29] N. Dell, V. Vaidyanathan, I. Medhi, E. Cutrell, and W. Thies, "'Yours is better!': participant response bias in HCI," in *CHI Conference on Human Factors in Computing Systems, CHI '12, Austin, TX, USA - May 05 - 10, 2012*, J. A. Konstan, E. H. Chi, and K. Höök, Eds. ACM, 2012, pp. 1321–1330. [Online]. Available: https://doi.org/10.1145/2207676.2208589

[30] L. Diekmann and L. Tratt, "Don't panic! better, fewer, syntax errors for LR parsers (artifact)," *Dagstuhl Artifacts Ser.*, vol. 6, no. 2, pp. 17:1–17:2, 2020. [Online]. Available: https://doi.org/10.4230/DARTS.6.2.17

[31] S. L. Graham and S. P. Rhodes, "Practical syntactic error recovery," *Commun. ACM*, vol. 18, no. 11, p. 639–650, nov 1975. [Online]. Available: https://doi.org/10.1145/361219.361223

[32] T. A. Wagner and S. L. Graham, "Efficient and flexible incremental parsing," *ACM Trans. Program. Lang. Syst.*, vol. 20, no. 5, p. 980–1013, sep 1998. [Online]. Available: https://doi.org/10.1145/293677.293678