Syntactic Completions with Material Obligations

by

David Moon

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in the University of Michigan
2025

Doctoral Committee:

Assistant Professor Cyrus Omar, Chair
Assistant Professor Max S. New
Associate Professor Steve Oney
Professor Laurence Tratt, King's College London

David Moon
dmoo@umich.edu
ORCID iD: 0000-0002-1081-2235

ACKNOWLEDGEMENTS

I started working with Cyrus in 2018, when he was still a postdoc at the University of Chicago and I starting my PhD at CU Boulder. It was a stormy, tumultuous, exhilarating year. I saw the best clouds of my life, and I decided to leave Boulder and join Cyrus wherever he got a job. Much thanks to Ben Shapiro for shepherding my transition from Boulder to Michigan. Cyrus, thank you for giving me the chance to work on ambitious problems and the guidance and space to develop ambitious solutions. It has been a deeply fulfilling pleasure.

Andrew Blinn joined me at Michigan in 2020, and we lived and breathed structure editing together for the next five years. Living and working with him was a formative and enriching experience. This work owes much to his influence. I am grateful also to have met and worked with the many members of the Future of Programming lab—Eric Griffis, Hannah Potter, Yongwei Yuan, Milan Lustig, Matthew Keenan, Thomas Porter, Alexander Bandukwala, Gregory Croisdale, and more—whose insights and enthusiasm have been a steady source of inspiration and motivation.

TABLE OF CONTENTS

LIST OF FIGURES

FIGURE

LIST OF APPENDICES

ABSTRACT

Language-aware program editors offer essential services to help programmers understand, navigate, and modify their code in various stages of completion. These services require analyzing the syntactic structure of the code, which is typically inferred by a parser from textual input. The problem motivating this work is that, for typical grammars, most textual edit states do not successfully parse, compromising the behavior of downstream editor services.

Contemporary editors address this issue in one of two ways: either it is a text editor equipped with an *error-handling parser*, or it is a *structure editor* whose edit operations maintain syntactic structure throughout development. Error-handling parsing methods, which often repair the text around unexpected tokens, leave unanswered questions about how to surface these repairs effectively to the programmer. Meanwhile, structure editors are notoriously cumbersome to use, particularly when it comes to modifying existing code, due to mismatch between the tree structure of the internal edit state and the linearity of the externally projected view.

This dissertation contributes the `tylr` series of program editors, a series of increasingly flexible structure editors that converge on a novel method of error-handling parsing. The central innovation is a language-parametrized system of *syntactic obligations* used to scaffold and guide completion of partial structures. From the view of structure editing, obligations generalize holes (placeholders for missing trees) and enable flexible interactions with arbitrary token ranges while preserving structural guarantees. From the view of error-handling parsing, they abstract over similarly structured textual repairs that prior methods would consider individually, as well as offer new possibilities for user-interaction.

I propose and evaluate three such possibilities that constitute the `tylr` series: `tiny tylr`, `teen tylr`, and `tall tylr`. Informed by user studies of its predecessors, each iteration loosens restrictions on the interactive behavior of obligations to support more flexible, text-like workflows. These changes lead to an error-handling generalization of operator-precedence parsing, formalized as a parsing calculus called meldr, that recovers from errors by inserting obligations and is guaranteed to produce a grammatical term on all inputs. I conclude with a discussion of takeaways for future program editor designs.

CHAPTER 1

Introduction

Language-aware program editors offer a range of useful services that help programmers understand, navigate, and modify their code in various stages of completion. Common examples, illustrated in Fig. 1.1, include (A) syntax highlighting, (B) variable renaming and type inspection, (C) go-to definition and documentation preview, and (D) live value inspection. These services require analyzing the syntactic structure of the program being edited, which is typically inferred by a parser from textual input. The problem motivating this work is that, for typical grammars, most textual edit states do not successfully parse, compromising the behavior of downstream editor services.

Consider Fig. 1.2(A), for example, which shows a malformed version of the program in Fig. 1.1. Processing the text left-to-right, a naively implemented parser would simply stop upon encountering the first unexpected token, `p2`, at the end of the first line, leaving the program unparsed and therefore unanalyzable. All of the downstream services shown in Fig. 1.1 would then have to be disabled, leaving the programmer only with "dead" text. This is unfortunate, especially since it is in these erroneous states that the programmer may need the most help.

Existing program editors address this problem in one of two ways. Either it is a text editor equipped with an *error-handling parser*, which produces analyzable structure from the text on a best-effort basis, or it is a *structure editor* whose edit operations maintain syntactic structure throughout development. The two approaches differ in the trade-offs they make between service availability and editor usability, which we will discuss next in §1.1. This work considers how we might combine their strengths.

1.1 Motivation

Error-handling parsing. An error-handling parser attempts to recover from unexpected textual input, typically by repairing the input and adjusting its internal state, so that it can resume parsing as usual. This way, editor services may continue to operate on the rest of the program around the error site. However, existing methods suffer from various limitations. Let us consider

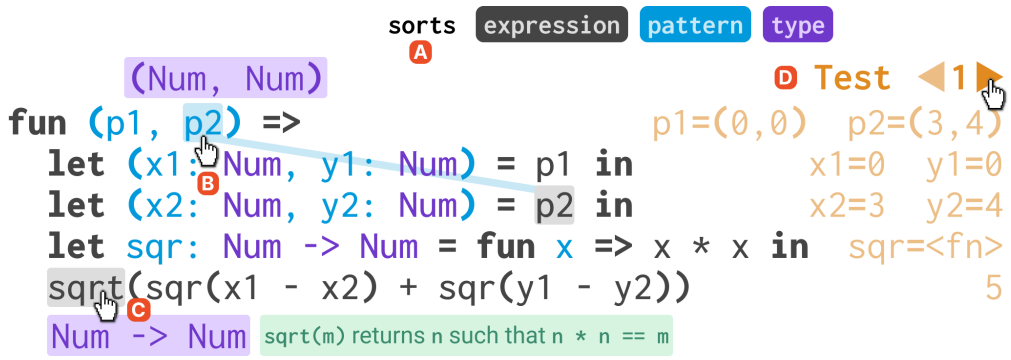


Figure 1.1: Illustration of common editor services: (A) Syntax highlighting helps the programmer distinguish at a glance between different syntactic sorts. (B) Hovering over a variable binding points to its uses; clicking with a modifier lets the programmer rename the binding and its uses all at once. (C) Tooltips on variable uses reveal their types and documentation; clicking with a modifier jumps to the binding site (not shown). (D) A test runner shows the concrete values being passed into the function over several test cases.

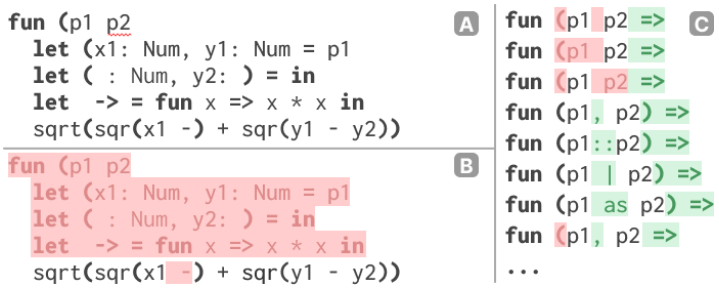
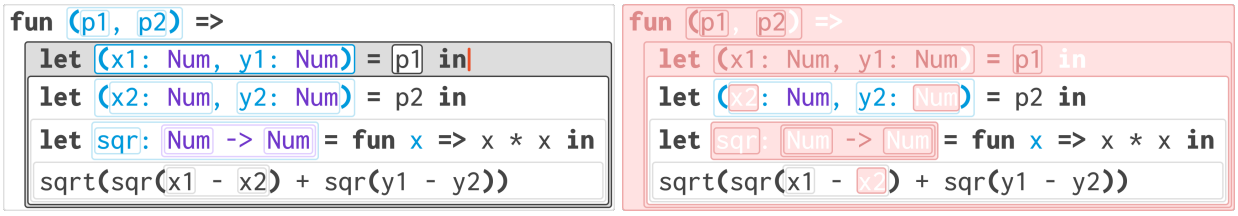


Figure 1.2: (A) A malformed editor state, with the first unexpected token underlined in red. (B) Regions skipped by a simple panicking parser, highlighted in red. (C) Some possible textual repairs for the first line generated by a conventional error-correcting parser.

how they might recover from the unexpected token p2 in Fig. 1.2(A).

A simple recovery method known as "panic mode" [5, 33] drops tokens heuristically around the error until parsing can resume—in this case, as shown in Fig. 1.2B, a simple panicking parser might drop the first four lines of code because of the various parse errors on those lines, then perhaps recover more granularly on the final line by ignoring the dangling minus sign. While better than nothing and relatively easy to implement, this approach is liable to ignore large windows around error locations [25], leaving the programmer with limited or incorrect assistance in those regions. For example, the dropped lines in Fig. 1.2B would lead a type error reporting service to mark the uses of x1, y1, and y2 unbound (in contrast to what a human would likely conclude).

More sophisticated recovery methods consider a broader range of possible repairs around the error location. Fig. 1.2C shows some possible repairs for the first line of code in Fig. 1.2A. The first three repairs show how, by considering possible insertions as well as deletions, this method reduces skipped input compared to a panicking parser. On the other hand, the next four completion-only repairs show how this method must enumerate all tokens that play similar structural roles—in this case, infix operators on patterns—which can lead to combinatorial explosion as additional insertions and deletions, larger repair windows, and larger languages are considered. There is prior work that suggests these repairs can be enumerated efficiently in practice [16, 25],



(a) (b)

Figure 1.3: (a) A simple structure editor that projects its edit state as nested blocks (some are elided) and features a text-like cursor. (b) When pressing Backspace on all tokens in white (the same tokens missing in Fig. 1.2(A)), the tokens in red also get removed.

but limit their consideration of how to surface these repairs to the programmer to ranking the repairs in an interactive dropdown menu. Such interfaces are uncommon in practice, perhaps because the visual information density of a list of various similar textual repairs is too low to be worth the effort of disambiguation. More commonly, the parser chooses for them, typically using some mixture of text-based and language-tuned heuristics [27, 30]. In this case, however, the choice is rarely surfaced to the programmer, leaving them only indirect clues in the behavior of downstream editor services.

In summary, existing error-handling parsing methods either limit themselves to deletion-based repairs, leading to large ignored regions; or else they incorporate insertion-based repairs but set aside questions about how to surface these repairs to the programmer, leading either to too many choices or a lack of transparency. Is there a better user interface out there for syntactic repairs?

Structure editing. Given the challenges of parsing and error handling, there has been a long and storied line of research on structure editing (a.k.a. structured editing or projectional editing) [6, 34, 44, 45, 50, 53, 59, 62]. A structure editor maintains a tree-structured edit state rather than a textual one, with *holes* standing for missing subtrees, and projects it to the programmer's display in a suitably interactive form. Interactions with this projection are then mapped back to the edit state as tree-to-tree transformations, which ensures that the edit state is continuously well-structured and amenable to analysis.

For example, Fig. 1.3a illustrates how a simple structure editor might project the program in Fig. 1.1 as nested labeled blocks. A text-like cursor navigates between tokens in the visualized order and highlights the smallest block containing the current token, indicating it is the target of any subsequent modifications (e.g. removing it, wrapping it in a new block). Because changes occur at the granularity of whole tree nodes, malformed edit states like the one in Fig. 1.2 would simply be unreachable.

Unfortunately, this continuous structure comes at the cost of a highly viscous [31] editing

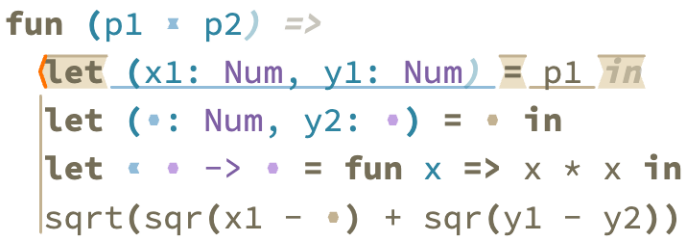


Figure 1.4: The syntactic completion of the program from Fig. 1.2 in `tylr`, our tile-based editor.

experience, i.e. it can be cumbersome to modify existing code. Pretend you are the designer of the structure editor in Fig. 1.3: what should the result be of pressing Backspace at the depicted edit state in (a)? Should this editor behave like Scratch [44] and remove the entire block, descendants included, leaving a hole in its place? If not all descendants, which should be left behind instead? (Is your choice based on a generic or form-specific policy? How will you communicate this policy to the programmer?) However you choose, your choice is limited to at most one block to replace the removed one.

Consequently, deletion in strict structure editors can feel a bit like a sledgehammer. Fig. 1.3b shows one possible outcome of pressing Backspace on all the tokens in white, the same as the ones missing in Fig. 1.2(A). A majority of the total perceived change is collateral damage, shown as tokens in red, that can be challenging to predict even for experienced users (see Fig. 2.2, adapted from a study of the MPS structure editor [11]). Might it be possible to support more precise deletion behavior without sacrificing all structural guarantees?

It is further interesting to contrast the collateral damage in Fig. 1.3b to the tokens skipped by the panicking parser in Fig. 1.2(B), both caused by the same tree-structural constraints. The structure editor manages to get away with relatively less damage, thanks to its ability to insert holes in place of missing trees, which lets it preserve the two innermost `let`-expressions. Could there be other forms of "holes" used to further reduce the damage?

1.2 Contributions

This dissertation contributes the `tylr` series of program editors, a series of increasingly flexible structure editors that converge on a novel framework for error-handling parsing. The central innovation is a language-parametrized system of *syntactic obligations* used to scaffold and guide completion of partial structures. From the view of structure editing, obligations generalize holes and enable flexible interactions with arbitrary token ranges while preserving structural guarantees. From the view of error-handling parsing, they abstract over similarly structured textual repairs that prior methods would consider individually, as well as offer new possibilities for user-interaction.

Fig. 1.4 shows how the latest `tylr` editor repairs our malformed program text in Fig. 1.2A by completing it with various obligations. These obligations categorize the various *multiplicity* and *sort* inconsistencies that, when parsing, may arise between top-down expectations of the language grammar and bottom-up reductions of the input tokens:

- Lines 3-5 are missing operands of various syntactic sorts: one pattern, three types, and two expressions. In their place are *operand obligations* ⬡ that, like structure editor holes, indicate there is no subtree where one is expected ($0 = \bullet < 1$).
- Lines 1-2 are missing delimiters required by various multi-delimiter forms. In their place are *delimiter obligations*—⟩), ⟩=>⟨, and ⟩in⟨ in transparent italicized font—that indicate there is a partial form where a complete one is expected ($0 < \bullet < 1$).
- Line 4 is missing a pattern form expected to wrap the following type. In its place is a *sort transition obligation* ⧏ that indicates there is a subtree as expected ($\bullet = 1$), but of the wrong sort.
- Line 1 is missing a connective between the two pattern variables. In its place is an *infix obligation* ⧈ that indicates there are multiple unjoined subtrees where one is expected ($1 < \bullet$).

In short, obligations classify and repair all the ways a bottom-up parse may "go wrong".

I did not start this work with this parsing-centric understanding of obligations. Early on, my goal was to design an easy-to-use text-like structure editor for the Hazel programming environment [55]. Prior art at the time supported some text-like insertion patterns, e.g. writing infix operator sequences left-to-right, but offered no similar guidance for selection or deletion. I review this work among others in Chapter 2, focusing on designs with keyboard-driven textual projections, and identify three distinct facets of their viscosity—selection expressivity, delimiter matching, and term multiplicity—that help explain why even experts of an existing tool reported finding it difficult to predict the result of deletions, and directly motivate our obligation design.

Central in this work is the question of how the programmer should *interact* with these obligations—in particular, how to discharge them. I propose and evaluate three possible answers that constitute the `tylr` series: `tiny tylr` (Chapter 3); `teen tylr` (Chapter 4); and `tall tylr` (Chapter 5), the latest version of `tylr` as of this writing. Using `tall tylr` in Fig. 1.4, for example, the programmer may accept the suggested location of the delimiter obligation ⟩in⟨ by placing the cursor on it and pressing the Tab key or typing over it explicitly; alternatively, they may type `in` in a different intended location, and the obligation is removed when no longer necessary. Informed by user studies of its predecessors, each iteration loosens restrictions on the interactive behavior of obligations to support more flexible, text-like workflows.

Chapter 3 on `tiny tylr` (adapted from [46]) and Chapter 4 on `teen tylr` (adapted from [47]) show how we first developed, then refined the use of syntactic obligations for improving struc-

ture editor usability. The main idea governing those designs was to allow selection and modification of arbitrary token ranges, while scaffolding the results with obligations to guide proper reassembly back to hierarchical structure. Our user studies focused on structurally complex modification tasks and showed significant improvements in modification performance and reductions in mental load when participants used `tiny tylr` and `teen tylr` over JetBrains MPS editors, the state-of-the-art in keyboard-driven structure editing.

Our studies also suggested that there remained usability limitations in our designs compared to text editors. Refining our conceptual design of obligations in response to this feedback led us to the error-handling parsing approach taken by `tall tylr` in Chapter 5 (adapted from [48]). Unlike the preceding chapters, which focused entirely on the user-interaction design of obligations, this chapter gives a formal, language-agnostic description of obligations and their use in error-handling parsing. Specifically, this chapter contributes meldr, a parsing calculus consisting of two parts: a grammar transformation that extends the language grammar with a small number of obligation forms per sort, and an error-handling generalization of an operator-precedence parser that is proven to be capable of completing any sequence of terminal symbols with obligations and parsing it into an extended-grammatical term. Along the way, this calculus contributes a novel parser-independent semantics for precedence annotations on context-free grammars.

meldr goes beyond mere recognition of grammatical input, as is typical of existing parsing formalisms, and models the resulting concrete syntax trees. While our current theory only describes left-to-right parsing from scratch, these definitions establish a foundation for future work on modeling structured edit states and incremental parsers. I conclude in Chapter 6 with a discussion of ongoing work and takeaways for the design of future program editors.

Among other things, this work defends the following statement:

> *Syntactic obligations provide the basis for flexible structure editing and total error-handling parsing that skips no input.*

CHAPTER 2

Background and Related Work

2.1 Structure Editing

`tylr` contributes to a long history of structure editor design, dating as far back as the Emily editor introduced in 1971 [34].

2.1.1 Block-Based Editing

The most popular structure editors today are block-based editors like Scratch [44]. In these editors, the programmer constructs programs like those shown in Fig. 2.1 (adapted from Weintrop [67]) using their mouse to drag-and-drop 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. The `tylr` editors employ a similar metaphor of syntactic-forms-as-puzzle-pieces, but use a uniform shape system across all sorts, eliminating the visual design burden of language customization.

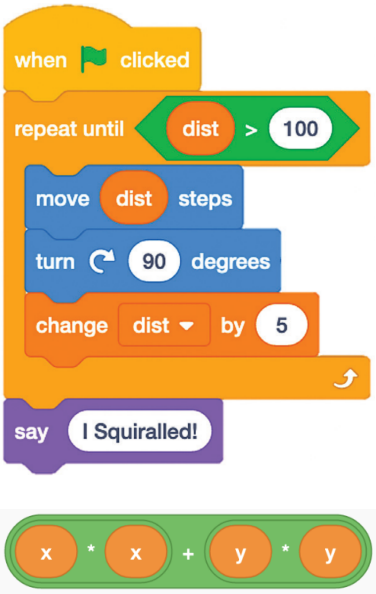


Figure 2.1: Scratch blocks

While block-based editors have seen great success at teaching programming to novices, they soon become unwieldy once programmers start creating and maintaining larger or more expression-based programs. For example—adapting an observation by Brown et al. [12]—constructing the small calculation at the bottom of Fig. 2.1 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; meanwhile, the equivalent construction in a text editor 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 (drag in the

new block, then drag into it the existing block). 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 [36]—our designs avoid this problem by only decorating the structures under the cursor, rather than decorating all structures at all times.

In a user study of block-based editing involving large refactoring tasks, Holwerda and Hermans [36] elicited post-task user responses on the cognitive dimensions [31] 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 Fig. 2.1, 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.

Verano Merino and van der Storm [61] present a method for generating block editors from grammars, refined by Verano Merino et al. [60] by pre-processing grammars to remove parsing-specific production rules not relevant in a block context. More recent work by Steimann and Stunic [58] re-contextualizes block editor derivation via dependency grammars, a framework which more naturally supports the derivation of block editors for context sensitive languages.

Homer and Noble [37] present a hybrid text and block-based programming language called Tiled Grace, which they term 'tile-based', though their tiles are closer to blocks in the Scratch/Blocky sense than the `tylr` sense. Tiled Grace can shift between draggable blocks and editable textual syntax, though this shift is modal. No attempt is made to preserve structure in the textual mode, and the text must be syntactically correct to switch back to tile form.

2.1.2 Text-Like Editing

In this work, I focus on keyboard-driven, textually projecting structure editing in order to capitalize on the literacy and keyboard skills of experienced programmers. This avoids the particular pitfalls of block-based editing but faces new challenges.

Some keyboard-driven editors like Greenfoot [42] and the Cornell Program Synthesizer [59] employ hybrid editing models, using structure editing for large syntactic forms while deferring to text editing at the leaves, e.g. when editing expressions. This approach loses the benefits of structure editing at those levels.

Other editors, like those built with the language workbench JetBrains MPS [62], take a strictly structured approach and use a number of techniques to translate text-like editing flows into tree transformations. MPS editors project the program tree to a caret-navigated textual form. Editor behavior is customizable, both by modifying the language grammar and by implementing hooks

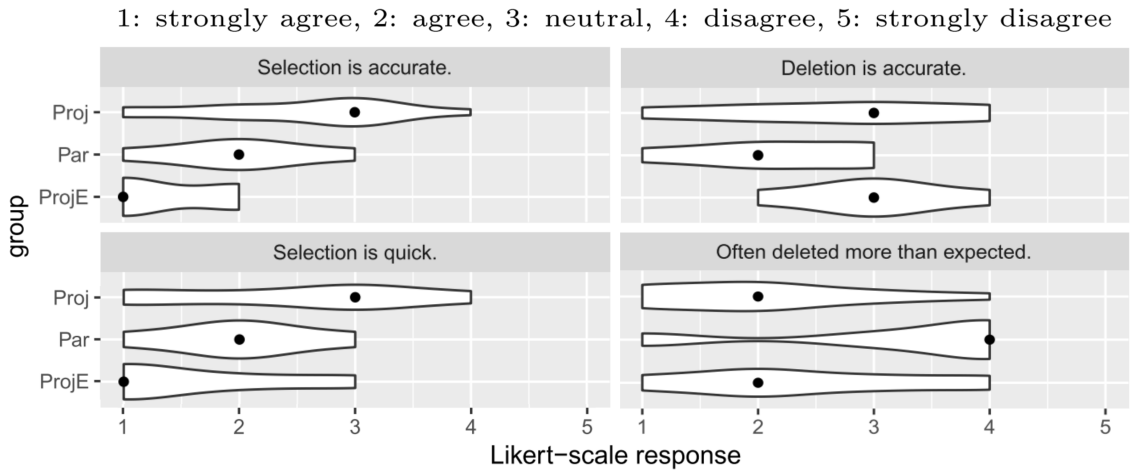


Figure 2.2: Violin plots of post-task questionnaire responses from a controlled user study of MPS, adapted from [11]. Each plot partitions the responses across the three study groups: MPS novices (Proj), MPS experts (ProjE), and text editor users (Par).

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 [63].

Chapter 4 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 like grammar cells exist, they are limited to an ad hoc collection of editing patterns—e.g. transcription of single-token infix operator sequences—that are concentrated on insertion and offer much less for selection, deletion, and modifying more complicated expression structures.

The consequences are shown in the violin plots in Fig. 2.2 reproduced from a user study comparing text editing to MPS. The left two plots in Fig. 2.2 show that, after the 90-minute study, MPS novices felt that selection was relatively slow and inaccurate. The viscosity here is caused by what I 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`. This 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 Fig. 2.3. Recent work by Prinz et al. [56] proposes mitigating the selection expressivity problem by supporting selections of one-hole contexts, i.e. trees with a single hole—our work aims instead

for text-like selections of arbitrary token ranges.

Meanwhile, the right two plots in Fig. 2.2 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. I construe this overall 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 (referred to as the term's *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 I 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 strict, single-tree 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. Block-based editors sidestep the multiplicity problem because any number of disjoint structures may co-exist in their canvas-based interfaces, but at the cost of slower mouse-driven input. In Chapter 4, I describe how the matching and multiplicity problems led to competing demands for limited clipboard space when our study participants used MPS, leading to higher reported mental load.

Beckmann et al. [10] proposed a system called Sandblocks for supporting some text-like editing flows using a form of incremental parsing called partial parsing [9]. Sandblocks uses partial parsing to complete prefixes of syntactic forms and lets the user pick between different possible completions using a dropdown menu, where completions may features holes for missing terms. Like MPS, these techniques are limited to simple wholesale insertion and deletion flows and does not address the viscosity problems associated with more complex modifications. In particular, Sandblocks does not support structurally cross-cutting selections.

The present work grew out of work started in Hazel [55], a live functional programming environment. Hazel is built on a theory of incomplete programs that assigns static [53, 70] and dynamic [52, 69] meaning to every well-structured edit state. Omar et al. [51] developed on top of these foundations a system of live graphical macros, called livelits, that enable new combinations of graphical and symbolic programming in the same editor. The promise of rich editing experiences like these continues to motivate the present work.

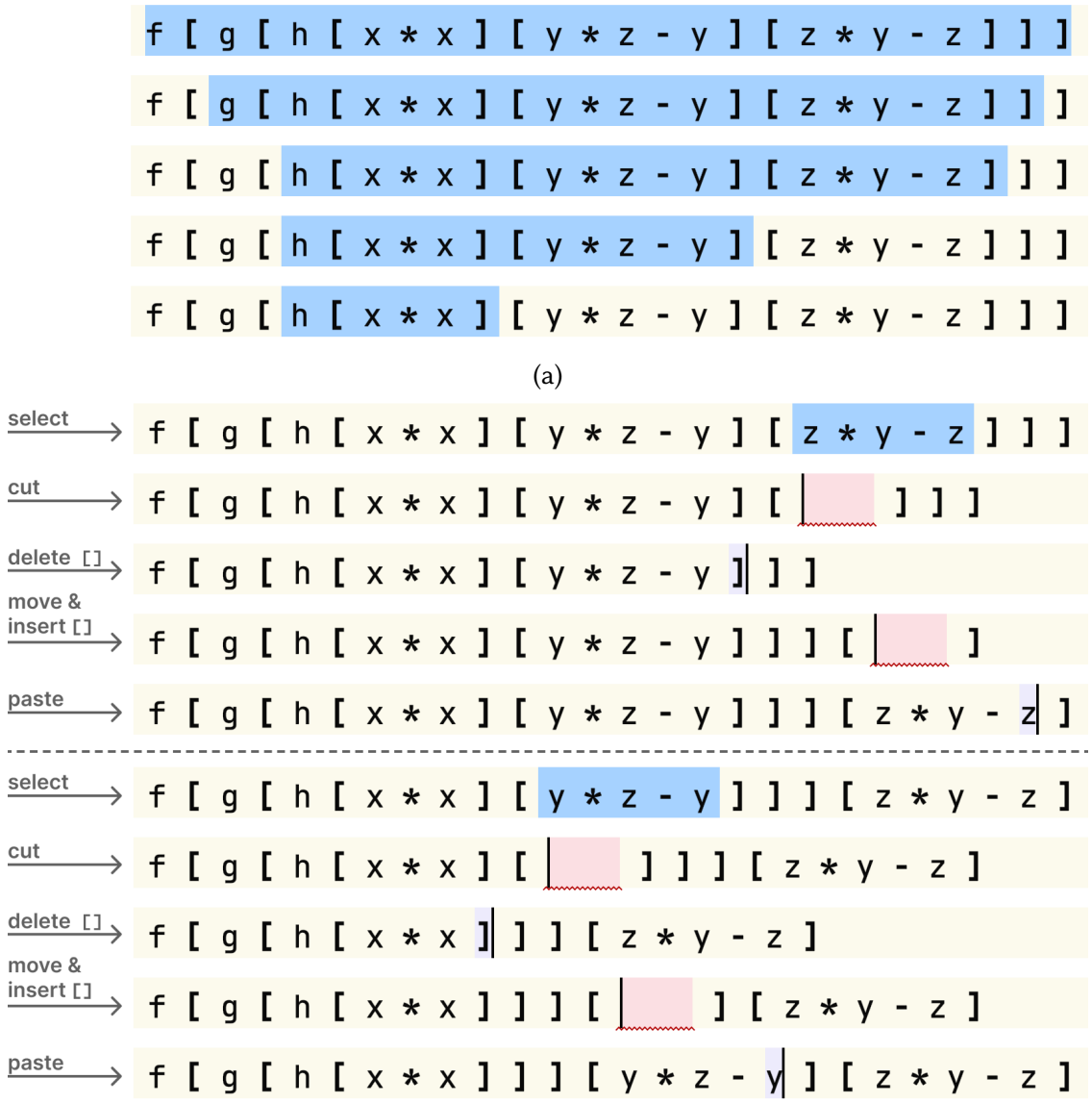


Figure 2.3: Screenshots of a JetBrains MPS editor being used to edit an expression of nested function applications, one of the study tasks we used to evaluate MPS and `tiny tylr` (§3.3). (a) shows all possible selections the user can make that contains a bracket, given MPS's restriction of selections to complete program terms. (b) shows the optimal edit sequence for completing the task. 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.

2.2 Parsing

First, some preliminary definitions. A *context-free grammar* (CFG) consists of:

- a set $X = T \sqcup N$ of symbols, partitioned into sets T and N of *terminals* and *nonterminals*, respectively;
- a set of *production rules*, each a mapping $n \mapsto \overline{x}$ from a nonterminal $n \in N$ to a sequence of symbols $\overline{x} \in X^*$; and
- a designated *start symbol* $n_0 \in N$.

If $n \mapsto \overline{x}$ is a production rule, then for all symbol strings $\overline{x}_L, \overline{x}_R \in X^*$, we say that the string $\overline{x}_L n \overline{x}_R$ *yields* the string $\overline{x}_L \overline{x} \overline{x}_R$, written $\overline{x}_L n \overline{x}_R \Downarrow \overline{x}_L \overline{x} \overline{x}_R$. If $\overline{x}_0 \Downarrow^* \overline{x}$, where $\Downarrow^*$ denotes the reflexive transitive closure of $\Downarrow$, we say that $\overline{x}_0$ *derives* $\overline{x}$. Given a string of terminals $\overline{t} \in T^*$, the goal of parsing is to show there exists a derivation $n_0 \Downarrow^* \overline{t}$. Any string of symbols $\overline{x} \in X^*$ derivable from n_0 is called a *sentential form*.

Witnessing every derivation $n_0 \Downarrow^* \overline{t}$ is a sequence $n_0 \Downarrow \overline{x}_1 \Downarrow \cdots \Downarrow \overline{x}_k = \overline{t}$ ($k \geq 0$) of yields, each invoking a production rule. We may organize this sequence of yields into a *derivation tree*: a tree whose leaves are labeled with the terminals in $\overline{t}$, and whose internal nodes are labeled with nonterminals, the root labeled n_0, such that every internal node n and its children $\overline{x}$ corresponds one-to-one to a yield invoking the rule $n \mapsto \overline{x}$ in the overall sequence. Given a string of terminals $\overline{t}$, a *recognizer* checks whether there exists a derivation $n_0 \Downarrow^* \overline{t}$, while a *parser* additionally produces the witnessing derivation tree, called in this context the *parse tree*. It is possible in general for there to be multiple parse trees for the same input, in which case the underlying grammar is called *ambiguous*.

2.2.1 Bottom-Up Parsing

A *bottom-up parser* starts with a sequence of terminal leaves $\overline{t}$ and repeatedly merges together neighboring trees as permitted by production rules, until there remains a single parse tree helmed by the start symbol (provided nothing goes wrong).

Fig. 2.4 illustrates this process when parsing left-to-right. The parser ingests terminal leaves one-by-one, organizing them in a stack and merging them in-place when appropriate. A left-to-right bottom-up parser is also called a *shift-reduce parser* because at each step, it may either *shift* the next terminal leaf onto the stack or else *reduce* (merge) together a span of neighboring trees on the stack. How the parser identifies the next reduction target, called the *handle*, is its distinguishing feature amongst other shift-reduce parsers.

For example, LR(k) parsers [41] identify handles using a grammar-derived automaton that is run alongside the stack in Fig. 2.4 and fed the same terminals. Fig. 2.5 shows at the top a

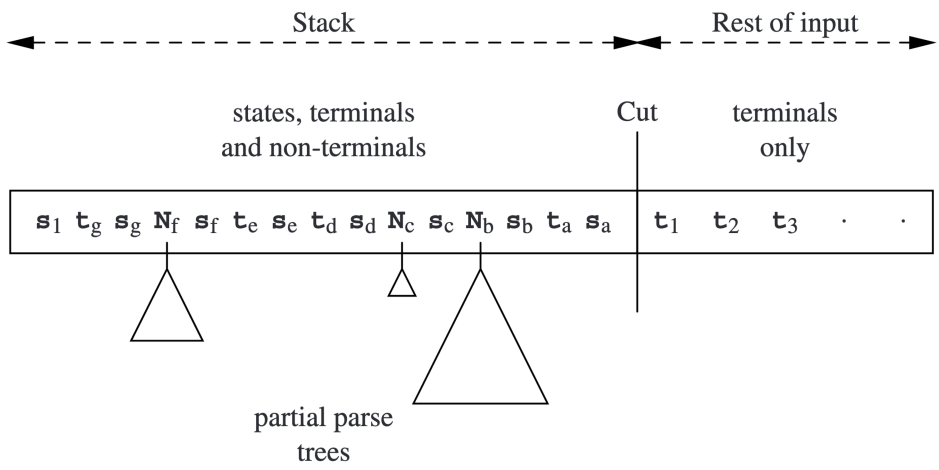


Figure 2.4: A schematic of left-to-right bottom-up parsing, adapted from Grune and Jacobs [33]

simple grammar for the differences of numbers alongside its LR(0) handle-finding automaton. Each state is labeled with an *item* consisting of a production rule and a marked point in its yield—this signifies the automaton's current progress toward identifying a handle reducible by that rule. At the bottom is a sample trace of the automaton processing a sentential form when it finds its first handle, i.e. reached an item $\mathsf{T} \to n\bullet$ marked at its right end. To perform the reduction, the automaton is rewound to the state immediately before the handle, i.e. the last state with item $\mathsf{E} \to \bullet\mathsf{T}$; the handle n is reduced to T; then T is fed back to the automaton. (Exercise: what is the resulting state?)

2.2.2 Operator-Precedence Parsing

Operator-precedence (OP) parsing [29] is an early pioneer of shift-reduce parsing. Instead of a handle-finding automaton, an OP parser consults a *precedence table*, a CFG-derived table recording which *precedence relations* hold between all pairs of terminal symbols. Fig. 2.6 shows the derived precedence table for a simple arithmetic grammar following the standard order of operations.

Each cell in the table, at row t_L and column t_R, records whether the terminals t_L and t_R are precedence-related and, if so, by which operator(s) $\odot \in \{\lessdot, \doteq, \gtrdot\}$. Each precedence-related pair $t_L \odot t_R$ indicates the terminals appear consecutively in the given order, possibly separated by a nonterminal n, in some sentential form $\overline{x} = \cdots\ t_L\ \{n?\}\ t_R\ \cdots$. The operator $\odot \in \{\lessdot, \doteq, \gtrdot\}$ indicates in what order t_L and t_R were first yielded in the derivation of $\overline{x}$. For example, + getting yielded before × in the sequence

$$\mathsf{S}_S \Downarrow \mathsf{E} \Downarrow \mathsf{E} + \mathsf{T} \Downarrow \mathsf{E} + \mathsf{T} \times \mathsf{F}$$

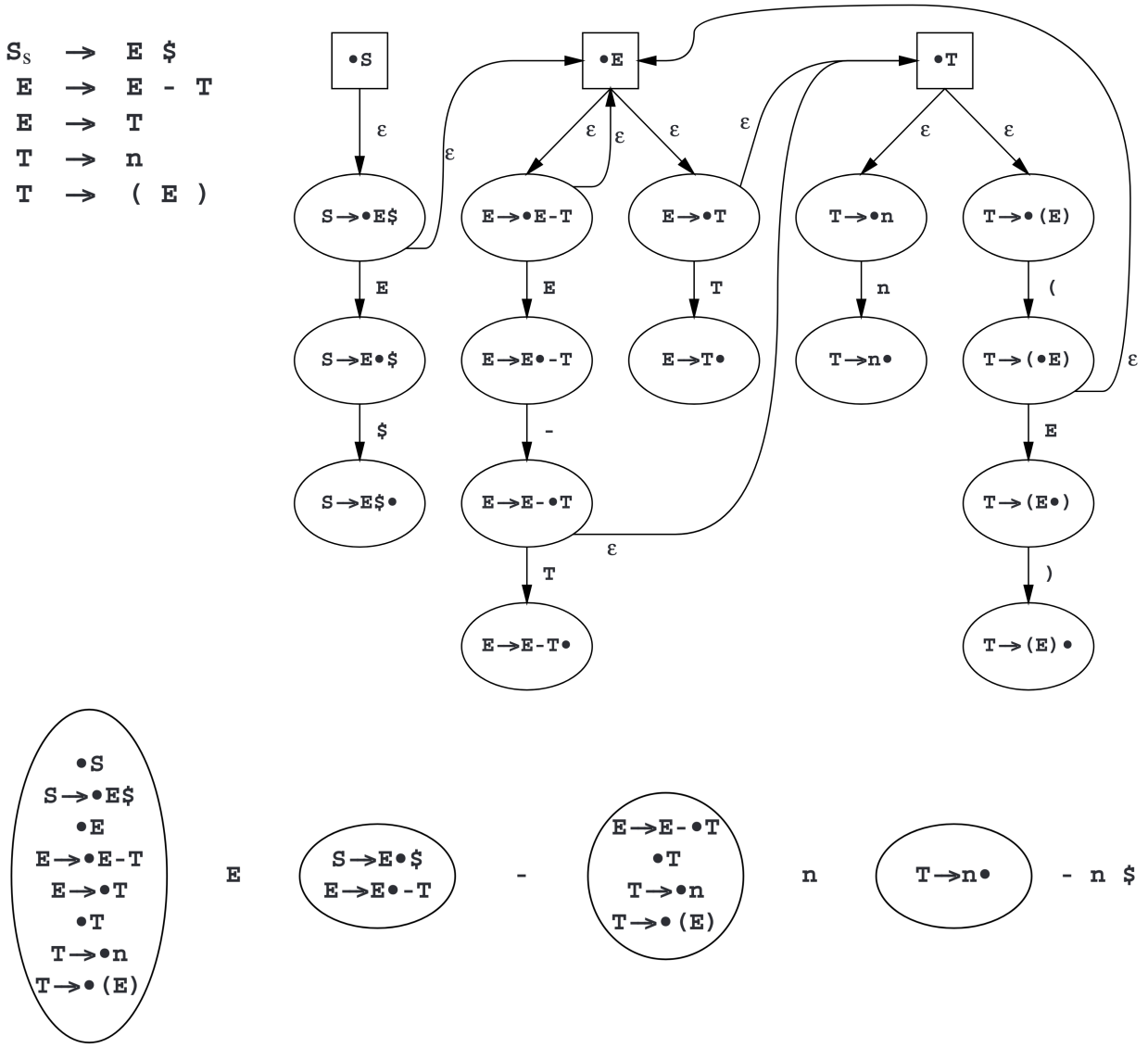


Figure 2.5: A simple grammar for the differences of numbers, its derived LR(0) nondeterministic automaton, and a sample trace identifying the first handle in the sentential form E-n-n$. Adapted from Fig. 9.14, 9.15, and 9.16 in Grune and Jacobs [33].

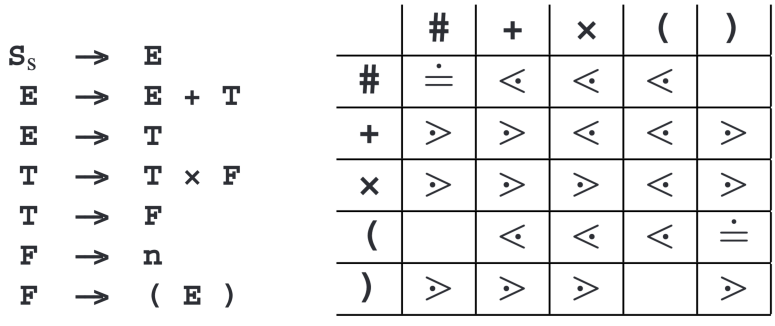


Figure 2.6: A simple arithmetic grammar and its derived precedence table. # denotes explicit start and end markers of an expression—the first rule should read $S_S \rightarrow$ # E #. Adapted from Fig. 9.2 and 9.4 in Grune and Jacobs [33].

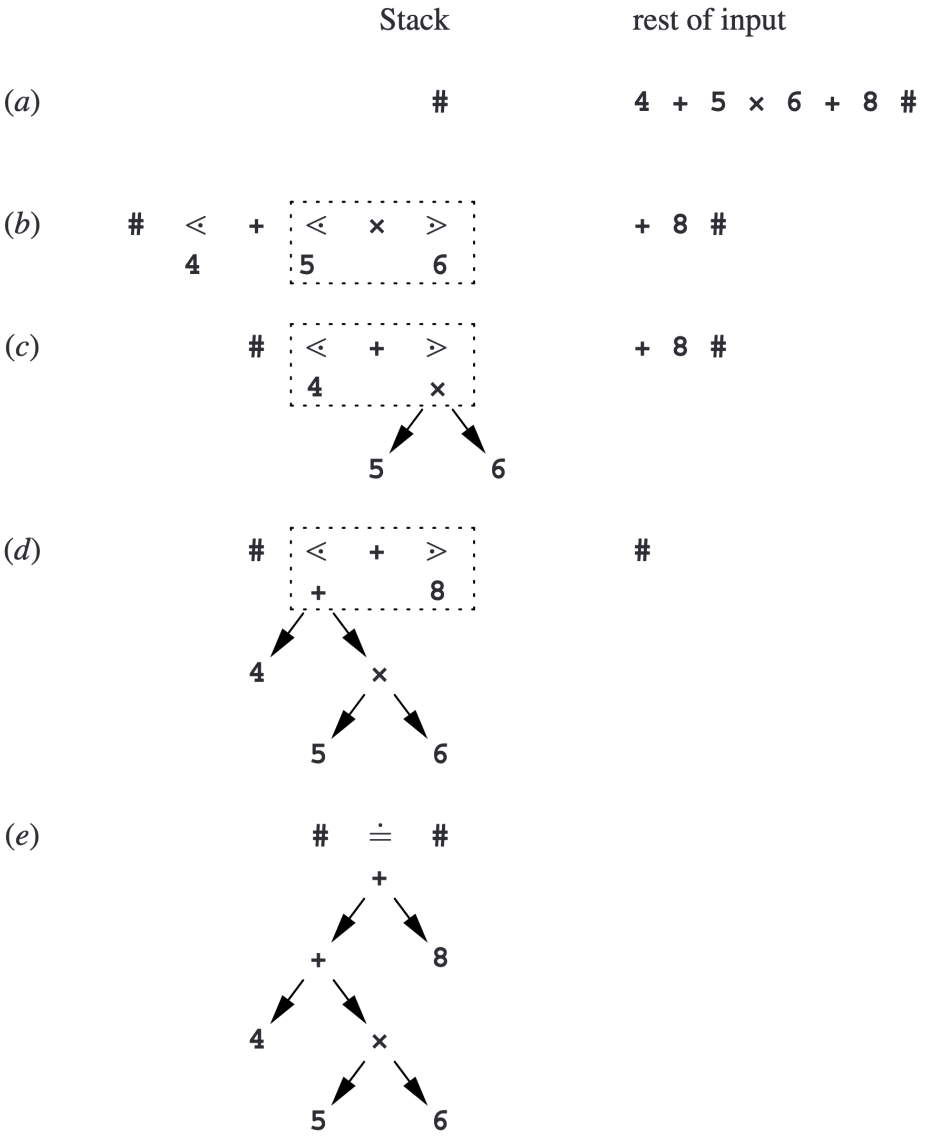


Figure 2.7: A trace of an operator-precedence parser for the grammar in Fig. 2.6, highlighting the moments when it finds a handle containing an operator. The parser proceeds from each such moment by reducing the found handle and tucking it under a precedence comparison operator relating the handle's delimiting tokens. Adapted from Fig. 9.5 in Grune and Jacobs [33].

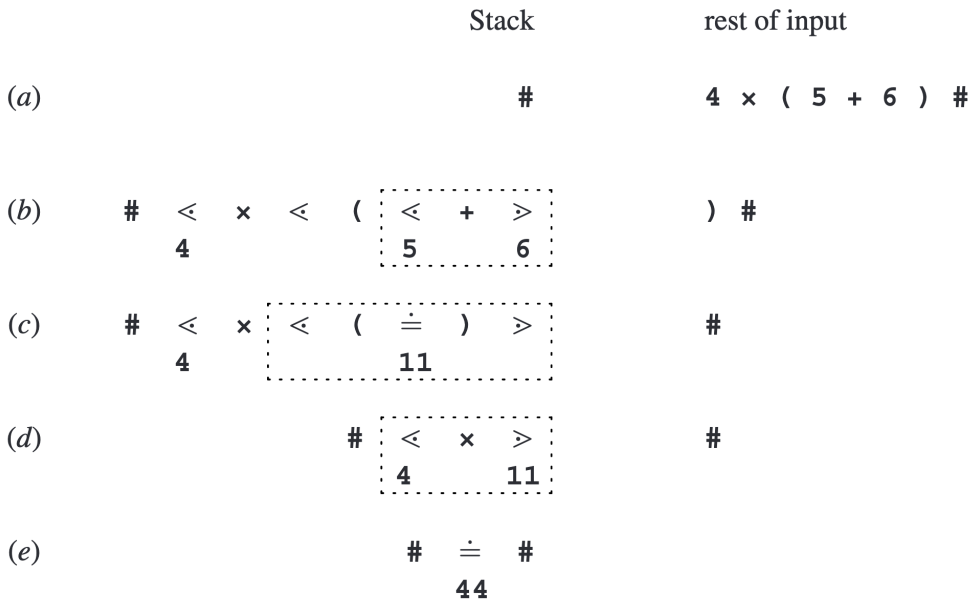


Figure 2.8: A trace of an operator-precedence parser for the grammar in Fig. 2.6, highlighting the moments when it finds a handle containing an operator or parentheses. The parser proceeds from each such moment by reducing the found handle and tucking it under a precedence comparison operator relating the handle's delimiting tokens. Adapted from Fig. 9.6 in Grune and Jacobs [33].

tells us that + ⋖ ×. Similarly, (and) getting yielded together in the sequence

$$\mathsf{S}_S \Downarrow \mathsf{E} \Downarrow \mathsf{T} \Downarrow \mathsf{F} \Downarrow \texttt{(}\ \mathsf{E}\ \texttt{)}$$

tells us (≐).

Given the notation, you might be tempted to think that $t_l \lessdot t_R$ is equivalent to $t_R \gtrdot t_L$, or that ≐ is symmetric—neither is the case. Keep in mind that the written order of each related pair $t_L \odot t_R$ of terminals reflects the order in which they appear in the sentential form $\overline{x} = \cdots\ t_L\ \{n?\}\ t_R \cdots$ witnessing their relation. Relatedly, the derivation

$$\mathsf{S}_S \Downarrow \mathsf{E} \Downarrow \mathsf{E}\ +_R\ \mathsf{T} \Downarrow \mathsf{E}\ +_L\ \mathsf{T}\ +_R\ \mathsf{T}$$

tells us that $+_L \gtrdot +_R$, i.e. + is left-associative.

It is possible in general for the same pair of terminals to be precedence-related via multiple operators, or none at all. To make use of these relations without backtracking, an OP parser stipulates that every pair of terminals be precedence-related by at most one operator—i.e. they are *conflict-free*—as is the case for the arithmetic grammar in Fig. 2.6.

Fig. 2.7 and Fig. 2.8 show traces of the arithmetic parser parsing sentential forms using the precedence table in Fig. 2.6. What is unusual about OP parsing compared to other methods is that it is blind to nonterminals. The stack is a chain $\# \left[\leqslant\ t_i \right]_{0 \leq i \leq n}$ of precedence-related terminals,

where each precedence operator may additionally have a reduced un-nonterminal-labeled tree tucked under it. Every step of OP parsing begins with a precedence comparison between the nearest terminal t_L in the stack and the next input terminal t_R. If $t_L \lessdot t_R$ for either $\lessdot \in \{\lessdot, \doteq\}$, then the parser shifts $\lessdot t_R$ onto the stack. Otherwise, if $t_L \gtrdot t_R$, this marks the end of a handle and it is time to reduce. Rather than by citing a production rule, handles are identified as chains of precedence-related terminals of the form $t_0 \lessdot t_1 \doteq \cdots \doteq t_n \gtrdot t_{n+1}$, where t_0 and t_{n+1} are delimiters to the actual handle segment between them—in this case, $t_n = t_L$ and $t_{n+1} = t_R$. The handle is reduced under a new unlabeled node, then passed on to the next OP parsing step, to be tucked under the next precedence comparison.

Because of this nonterminal-blindness, OP parsing is unsound [43], meaning it may accept non-sentential forms.[1] See Fig. 5.22 in Chapter 5 for an example of an unsound trace. Simple precedence parsing [68] presents one way of re-introducing nonterminals into OP-style parsing, which is to extend precedence comparisons to all symbols, not just terminals—unfortunately this comes at the cost of a severely restricted grammar class [33]. In a resolution similar to ours, Henderson and Levy [35] split each precedence relation $\odot$ into two relations $\odot_1$ and $\odot_2$, the difference being whether a reduced nonterminal is expected between the related terminals. meldr generalizes this idea by indexing each relation by the optional nonterminal itself—the slot-filling operation in meldr (Fig. 5.25) uses this extra information to validate that the bottom-up accumulated reduction meets the top-down slot's requirements.

OP parsers are easily specified and implemented, scale linearly with input, and enjoy the *bounded-context* property: handles can be identified knowing only the terminals that delimit them on the left and right.[2] Barenghi et al. [7, 8] exploited this property to develop efficient parallel parsers. In that work, they show how the bounded-context property implies that any delimited range of symbols within a sentential form can be reduced to a uniform double-ended stack-like structure—more precisely, for any delimited range $t_L \overline{x} t_R$ of symbols within a sentential form, there exists a unique string $\overline{y} = \overline{y}_L \overline{y}_R$ such that $\overline{y}_L$ contains only $\gtrdot$-operators, $\overline{y}_R$ contains only $\lessdot$-operators, and $t_L \overline{y} t_R \Downarrow^* t_L \overline{x} t_R$. This is an appealing property for building program editors and services, because it provides a uniform, maximally structured, and *modular* interface to any user-selected (or otherwise relevant) token range—where by modular I mean that the range need not also lug around its context to be structured and analyzed.

Despite these various advantages, OP parsing is typically passed over for more sophisticated methods (e.g. LR) because of its limited grammar expressivity. The requirement that the precedence table be conflict-free makes it difficult to reuse the same terminal symbol in different syn-

[1] Levy [43] took the goal of parsing to be detecting invalid sentences rather than valid ones, and hence called "complete" (detect all invalid sentences) what we call "sound" (detect only valid sentences) in this work.

[2] Specifically, this is $BC(1,1)$ property, where $BC(m,n)$ means handles can be identified knowing m symbols on the left, n symbols on the right.

tactic contexts, e.g. the minus symbol for both prefix negation and infix subtraction. On the other hand, Grune and Jacobs [33] remark that it is "surprising how many grammars are (almost) operator-precedence." Later in Chapter 5, I consider how we might retain the editor modeling advantages of OP parsing while also permitting reuse of tokens in different grammatical contexts.

2.2.3 Precedence Annotations

In OP parsing, the handle-identifying precedence relations are derived automatically from the grammar and its derivation patterns. In other words, these methods expect operator precedence conventions to be encoded in the CFG's nonterminal dependency structure. This is tedious to do by hand and leads to a profileration of nonterminals, one for each precedence level, that obscure the language's natural organization into semantic sorts—for example, consider the difference between the grammar in Fig. 2.6 and the grammar $\mathtt{E} \to \mathtt{n} \mid \mathtt{(E)} \mid \mathtt{E + E} \mid \mathtt{E} \times \mathtt{E}$. Our grammar elaboration (Fig. 5.15) automatically extracts these dependency structures from a precedence-annotated grammar. Not only does this organization benefit grammar authoring and documentation, it also helps our system repair errors using a concentration of grout forms that are semantically meaningful and thereby more easily user-communicable.

A different way to specify operator precedence is to annotate the CFG's production rules with explicit precedence levels or comparisons, like in our precedence-bounded grammars (Fig. 5.9) and as commonly supported by modern parser generators. This approach allows for a close mapping between the language's semantic and syntactic organization—not only does this make it easier to author grammars, it also lets our system repair errors using a concentration of grout forms that are semantically meaningful and thereby more easily user-communicable.

Predominant interpretations of precedence annotations, however, are specific to the parsing method—in LR parser generators, for example, the annotations are used to resolve shift/reduce conflicts in the generated action table [4]. Less common are parser-independent semantics, such as our PBG-to-CFG elaboration (Fig. 5.15). §5.3.1.4 described how prior semantics by de Souza Amorim and Visser [21] (for the language workbench Spoofax [39]) and Danielsson and Norell [20] (for mixfix operators in Agda) are unnecessarily restrictive in how they handle prefix and postfix operators. Making similar observations, Aasa [3] defined the precedence weights that we recapitulate in our elaborated reduction rules (Fig. 5.15). Aasa used these measures to define when a derivation tree of the underlying unannotated grammar is *precedence-correct* according to the annotations. Separately, Aasa also defined a translation from annotated to unannotated grammars, but this translation follows a different, more complicated design, an opinion we share with Danielsson and Norell [20]. Our grammar elaboration re-centers Aasa's precedence weights via a novel bidirectional organization.

2.2.4 Error Handling

Besides recognizing and parsing correctly written programs, modern parsers are expected to help developers track down and fix syntax errors. Here, syntax errors are operationalized as source locations where the parser gets stuck. Most left-to-right parsers can detect and locate the first error without modification (though, as discussed in §2.2, this is not always the case for OP parsing). The more challenging problem is to *recover* from each encountered error and continue parsing to detect subsequent errors.

Most recovery methods attempt to *repair* the input text around the error, differing in what repairs they consider and how they choose among them. The simple "panic mode" method [5, 33] limits itself to repair by deletion, dropping tokens around the error until parsing can resume from some prior state. While simple to implement, this method is liable to skip large regions of code, as was illustrated in Fig. 1.2B, leaving the programmer without downstream semantic analysis in those regions. To avoid this issue, more sophisticated methods [16, 27, 30] consider the full range of possible repairs, including insertions as well as deletions, and pick one of least cost according to a language-specific cost vector of token modifications or else textual edit distance. Most similar in spirit to our work is the FMQ method [28]—defined for a different class of top-down parsing methods not discussed here—which performs repairs using only insertions. Fischer et al. [28] defined the *insert-correctable* class of grammars against which any input text can be repaired by insertions to a grammatical form. meldr's injection of additional obligation forms (Fig. 5.23) automatically promotes every grammar to be insert-correctable.

Across their variations, these prior repair-based error-handling methods limit themselves to purely textual repairs. This can lead to a combinatorial explosion in the space of possible insertion repairs, as illustrated in Fig. 1.2C. While these repairs can be enumerated efficiently in practice [16, 25], most prior work does not consider the question of how to effectively surface these repairs to the programmer beyond ranked enumeration in a dropdown menu—some exceptions are discussed in §2.2.5. Our approach is novel in its use of abstract syntactic obligations to compress, rank, and communicate possible repairs.

Some recovery techniques [17, 57] avoid modifying the input text entirely. Instead, they partition the input into valid substrings of the language of maximal length and report the partition points as the locations of syntax errors. This approach avoids the problem of choosing the correct repair and the possibility of a wrong choice leading to cascading spurious errors, but at the cost of not producing a parse tree in the presence of errors and thereby disabling downstream analyses. Perhaps for this reason, noncorrecting methods like these have not reached mainstream adoption.

Error-handling parsing and structure editing are kin in their goals of maximizing structure, but emphasize quite different aspects of the design/technical problem space. Our prior emphasis on

structure editor design led here to a unique approach to error-handling parsing, one that builds on the underexplored symbol-based perspective of OP parsing (driven by symbol-to-symbol precedence comparisons), as opposed to the predominant item-based perspective of methods like LL/LR (where items refer to points *in between* the symbols of a production rule, used to define the states of handle-finding automata). The symbol-based perspective comes with design advantages, simply because tokens provide more visual surface area to decorate than zero-width items—it is, in our opinion, much easier to display and describe molds to the programmer than it is to display and describe items and automaton states.

2.2.5 Incremental Parsing

So far, I have discussed *batch* parsing methods, where the parser starts from scratch on a sequence of terminal leaves. An *incremental parser* [23, 65] attempts to minimize repeated work between successive edit states by reusing as much of the previous parse tree as possible. The schematic for its operation is almost the same as the one for batch parsing in Fig. 2.4—the only change is that the yet-unshifted terminal leaves may now be arbitrary parse trees, deemed potentially reusable from the previous edit state, and possibly containing user modifications that need to be revisited and reparsed. Additional machinery determines when a parse tree can be reused (i.e. shifted as is) and, if not, breaks it down into a sequence of constituent subtrees for subsequent processing. Wagner [64] developed incremental algorithms for LR and GLR parsing—later fixed and extended by Diekmann [23]—that live on today in Tree-sitter, a popular incremental parsing library [1].

Besides minimizing repeated work, an incremental parser also supports *history-sensitive* error-handling [23, 66]. The idea is simple: not every user modification needs to be structurally incorporated into the parse tree, and unincorporated modifications are decorated as errors. Once modifications may persist unincorporated, questions arise as to when it becomes appropriate to incorporate them and how to efficiently determine the relevance of other modifications. Wagner and Graham [66] and Diekmann [23] describe methods for isolating errors within containing parse trees (identified using the previous parse tree), such that the incremental parser may ignore the error when processing modifications outside of the isolated tree. Additional techniques are needed to incorporate legal modifications performed within an error's isolation tree.

The editing models of the `tylr` series also converge on incremental parsing with history-sensitive error-handling. The main difference lies in our recovery strategy and use of obligations. Where Wagner and Graham [66] uses a purely deletion-based recovery strategy that ignores some user input, we use a purely insertion-based recovery strategy that incorporates all user input and completes it using obligations.

CHAPTER 3

`tiny tylr`: A Tiny Tile-Based Editor

Preface

The first incarnation of `tylr` emerged from my earlier efforts to improve the structure editing experience in Hazel. When I started that work in 2019, Hazel had been designed as a simple and fairly standard tree-based structure editor. One deviation was how it directly modeled infix operator sequences in the syntax tree in order to simplify left-to-right insertion, while an operator-precedence parser produced the fully structured tree as needed. Left-to-right entry of simple operator sequences felt good, but I would start to feel trapped as soon as I introduced any multi-delimiter forms, especially when trying to refactor them. I wanted to extend to refactoring multi-delimiter forms some of the linear editing flows available for transcribing operator sequences. The challenge was doing this while maintaining a continuously well-structured edit state as required for Hazel's corresponding semantic guarantees (cf. end of §2.1.2).

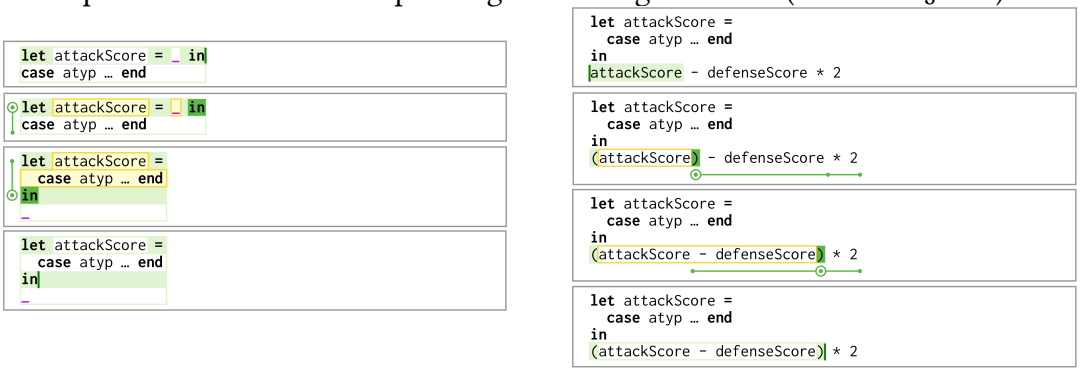


Figure 3.1: Early mockups of restructuring mode in Hazel

This motivated an early concept I called ~~node staging~~—wait, no—restructuring mode. Illustrated in Fig. 3.1, this mode would let the programmer move a single delimiter of a form at a time, with movement restricted to syntactically valid positions for the delimiter. Generalizing this feature over the growing Hazel language, then to moving arbitrary ranges of tokens, led to the concept of *tiles* as a generic way to model linearly arranged syntactic structures, determine valid movements/relocations among them, and inform subsequent hole-based repair. Meanwhile,

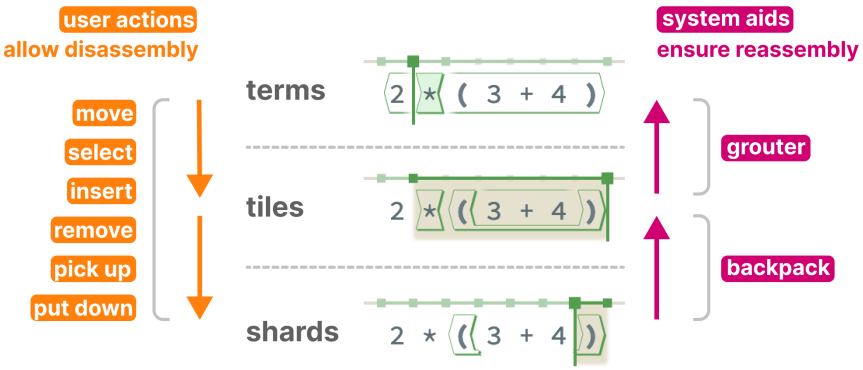


Figure 3.2: A high-level schematic of the concepts of tile-based editing. A tile-based editor operates on three levels of structure: terms, which follow the abstract syntax of the language; tiles, which correspond to groups of matching delimiters; and shards, which correspond to individual tokens and 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.

holes were generalized to *grout* to account for the various kinds of sequential inconsistencies that can arise between neighboring tiles.

Interested readers may play with `tiny tylr` at `https://tylr.fun/tiny`, the source of all screenshots in this chapter.

3.1 Contributions

A tile-based editor visually organizes projected tokens into hierarchical structures of three distinct strata depicted in Fig. 3.2: *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 chapter contributes `tiny tylr`, a tiny tile-based editor that concretely demonstrates these unique affordances. After an overview of `tiny tylr`'s design, I present the results of a lab study where participants performed simple code transcription and modification tasks using `tiny tylr` as well as a text editor and JetBrains MPS, state-of-the-art in keyboard-driven structure editing. I found that participants using `tiny tylr` completed some modification tasks significantly more quickly than when using MPS, particularly on those tasks where they made use of `tiny tylr`'s selection expressivity. I further observed that participants using `tiny 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. I discuss these limitations and conclude with future research and design directions for further improving the usability of tile-based

editing.

3.2 Design Overview

`tiny 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, `tiny 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, `tiny tylr` assists and guides these interactions to ensure that every edit state, upon pasting, can be reassembled into a well-formed term.

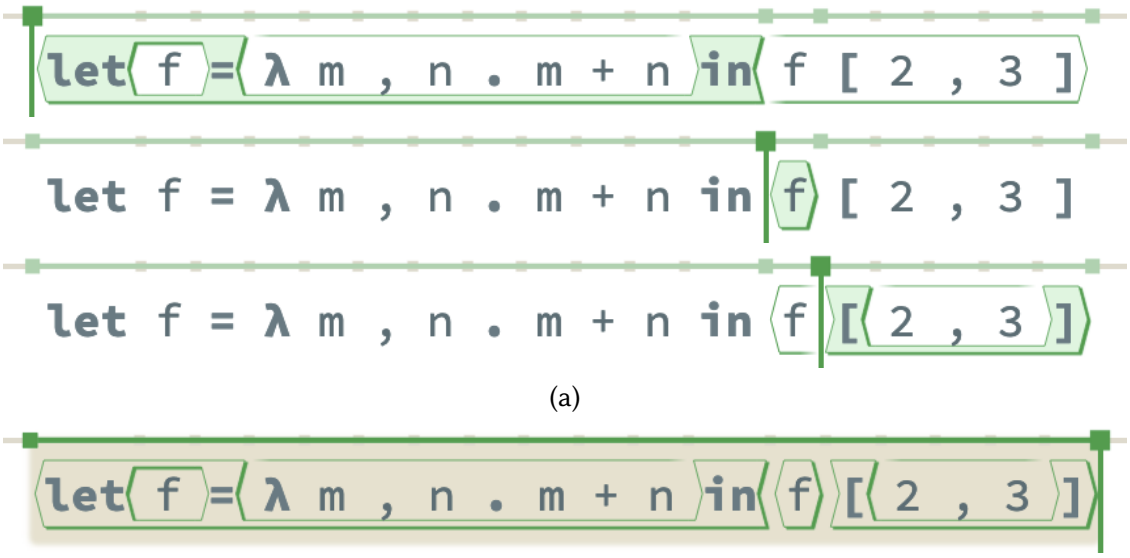
This assistance is divided into two independent subsystems that may be understood as operating at distinct levels of `tiny tylr`'s structural strata, as shown in Fig. 3.2: the *grouter*, which aids the reassembly of tiles into terms (§3.2.1); and the *backpack*, `tiny tylr`'s spiritual successor to the text editor's clipboard, which guides user movement to ensure proper reassembly of shards into tiles (§3.2.2).

3.2.1 Terms ⇌ Tiles: The Grouter

Panning the cursor over a program in `tiny tylr` reveals its term structure, as depicted in Fig. 3.3a, which follows the abstract syntax of a simple functional language. `tiny 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 Fig. 3.3a reveals the term's disassembly into a sequence of tiles, as shown in Fig. 3.3b. 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⟨.

Via operator-precedence parsing, a sequence of tiles reassembles into a valid term if and only



(a)

(b)

Figure 3.3: Screenshots of `tylr` showing a program's (a) term and (b) tile structure.

if the tiles fit together into a convex hexagon; that is:

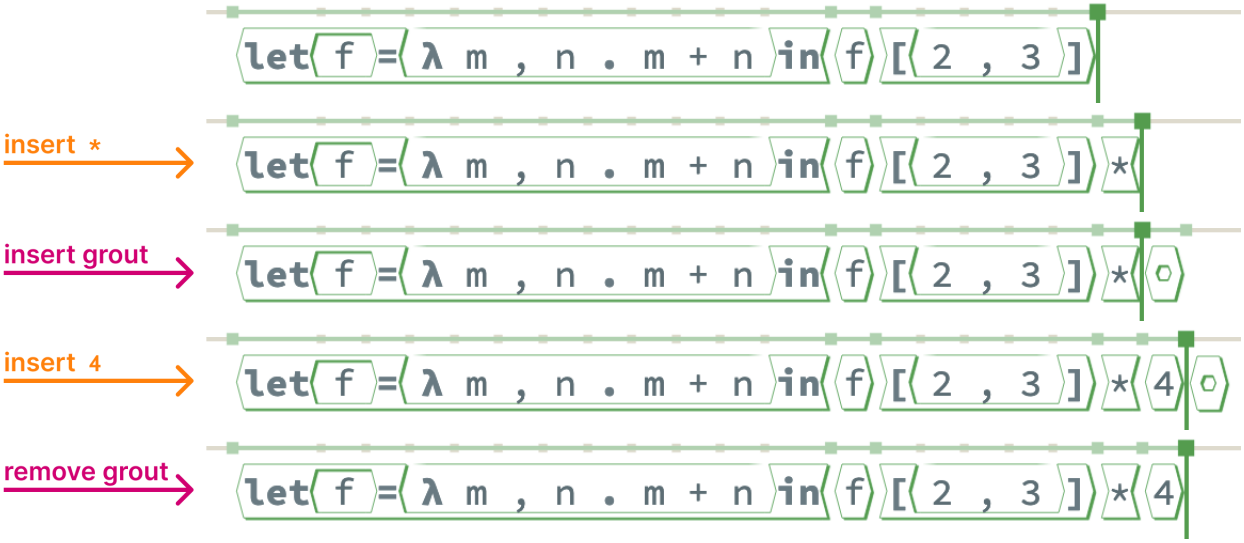
(1) consecutive tiles fit together, i.e. one tile's convex tip meets the concave tip of the other; and

(2) tiles at the ends have convex outer tips.

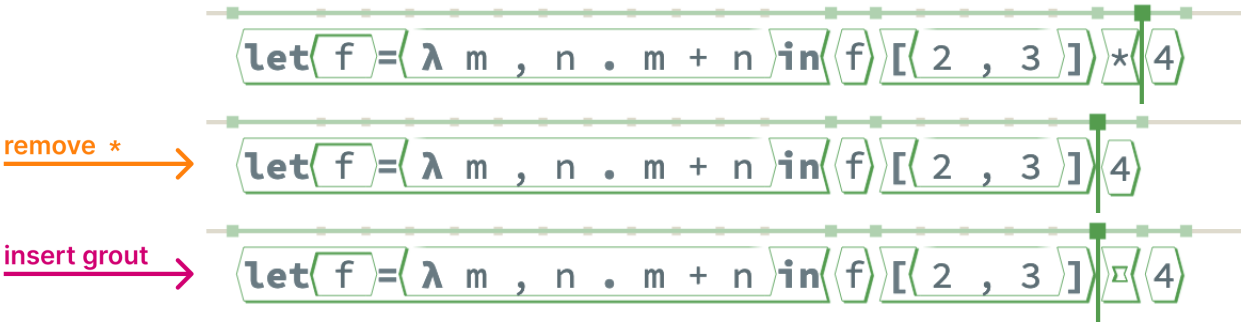
In order to maintain these conditions of fit and ensure proper term reassembly, `tiny tylr` is equipped with a subsystem we dub the grouter. Invoked immediately after each user modification, the grouter inspects the modification site and inserts or removes system-privileged structures, collectively called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

For example, consider the sequence of edit states shown in Fig. 3.4a, where the user modifies the program in Fig. 3.3 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 Fig. 3.4a, succeed the familiar concept of *holes* [54] 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 Fig. 3.4b, where the user press Backspace to the multiplication operator `*` inserted in Fig. 3.4a. 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 §2.1.2, if we were to perform the same edit in a term-based

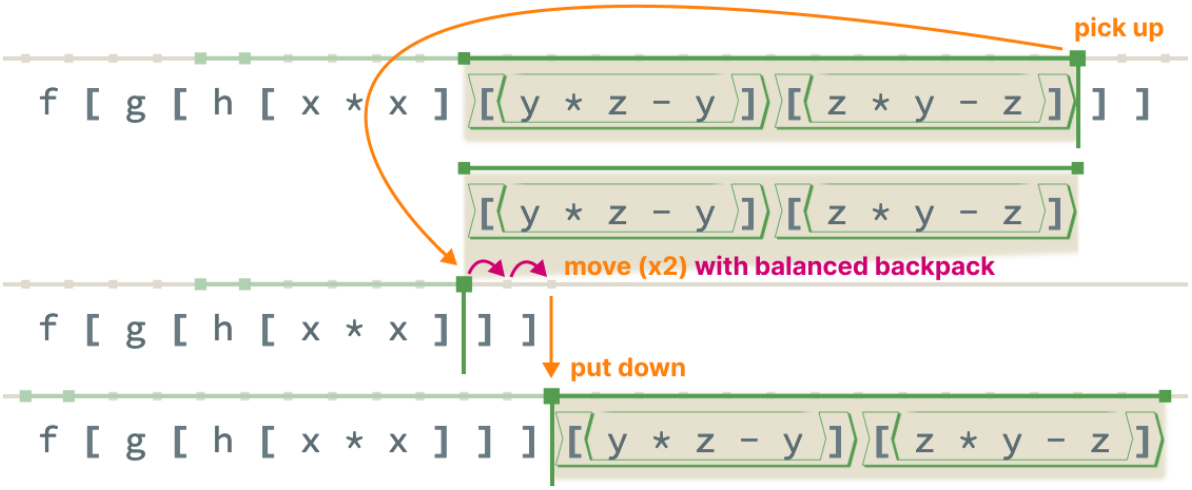


(a)

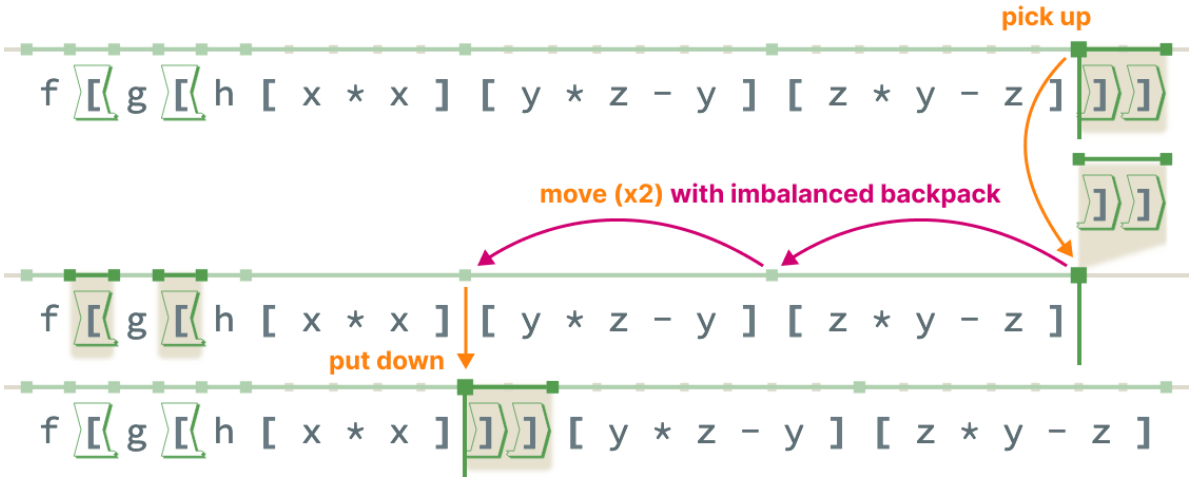


(b)

Figure 3.4: 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.



(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 3.5: The backpack in action, guiding user movement based on its contents.

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, `tiny tylr` can save both.

3.2.2 Tiles ⇌ Shards: The Backpack

`tiny 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.

Fig. 3.5 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 Fig. 3.5a: 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 (§3.3.2).

Sometimes even more selection granularity may be desirable. Consider an alternative approach to completing the same task, shown in Fig. 3.5b, 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 Fig. 3.5a 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 Fig. 3.5 take 4 user actions, (b) requires only 2 steps of movement to make the initial selection, whereas (a) requires 16.

3.3 Evaluation

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

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

To investigate `tiny 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 `tiny tylr`. We sought to answer the following questions:

- Does `tiny tylr` help first-time users complete program editing tasks more quickly than with another keyboard-driven but term-based structure editor? How does this performance compare to their text editor performance?
- To what extent do users make use of `tiny tylr`'s selection expressivity?

3.3.1 Method

We recruited 11 participants (P1-P11, 5 female and 6 male, μ = 22.2 years old, σ = 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-based 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 (μ = 6.8 years, σ = 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-based language called Lamb. We designed Lamb's syntax, shown in Fig. 3.6a, to accord with `tiny tylr`'s prototypal limitations.

Fig. 3.6b 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 `tiny 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

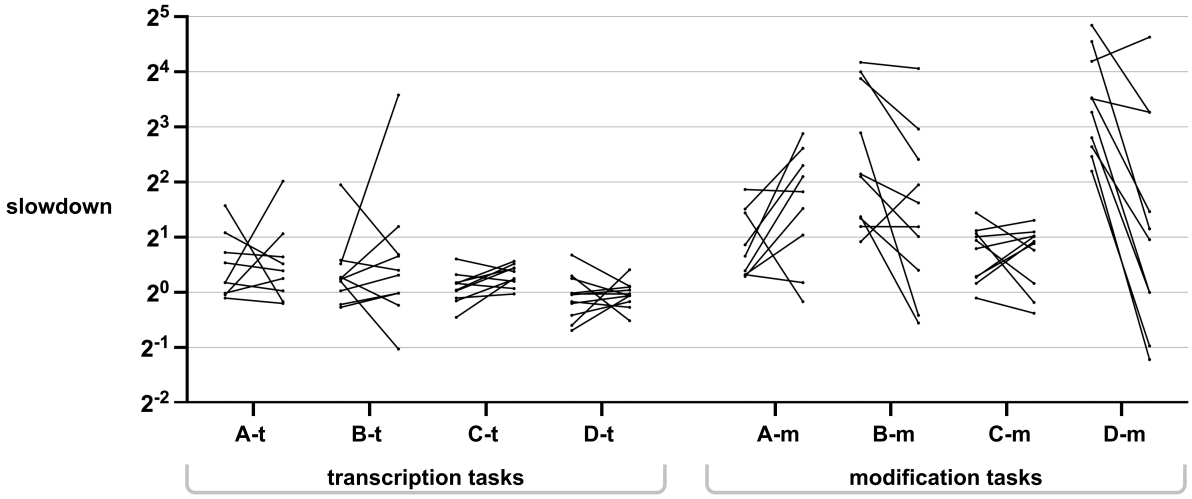


Figure 3.7: Summary of the slowdowns participants experienced in each task when using a structure editor as opposed to a text editor, where the slowdown is calculated as a participant's structure editor completion time divided by their text editor completion time. Each line segment corresponds to a participant; 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.

participants to parenthesize all subterms in a few sample programs. We configured VS Code to syntax-highlight Lamb expressions [14] and color matching brackets [26].

Participants were randomly assigned to an order for the subsequent MPS and `tiny 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 [49], 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 [63], 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 `tiny 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 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.

		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 3.8: 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 Fig. 3.3a), balanced but not a term at selection time (Fig. 3.5a), and imbalanced (Fig. 3.5b).

3.3.2 Results

Our evaluation suggests that participants did indeed make use `tiny 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 `tiny tylr`'s current design.

Fig. 3.7 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 `tiny 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 `tiny 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 `tiny 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

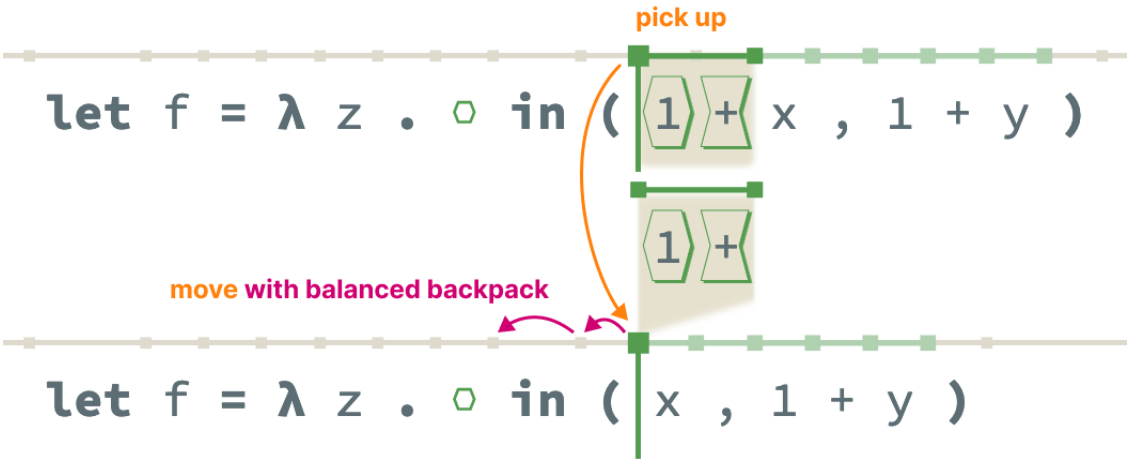
using `tiny tylr` over MPS on Tasks B-m ($t = -2.51, p < 0.05$, Cohen's $d = 0.83$) and D-m ($t = -4.87, p < 0.001$, Cohen's $d = 1.62$). Notably these tasks correspond to those in which participants made the most use of `tiny tylr`'s selection expressivity. Fig. 3.8 summarizes counts of selections users picked up into the backpack during the modification tasks of the `tiny 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 selections picked up in Tasks B-m (9 out of 14) and D-m (12 out of 12) respectively, which suggests that `tiny tylr`'s selection expressivity was important for completing those tasks more quickly. After completing Task D-m using the edit sequence in Fig. 3.5a, 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 Fig. 3.7 show several participants experienced worse slowdowns with `tiny tylr` than with MPS. On these tasks, we observed many participants get slowed by prototypal limitations of `tiny 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 `tiny tylr`: they started by picking up the starting program and attempted to construct a let expression, only to be reminded by `tiny tylr`'s interface that this is not possible.

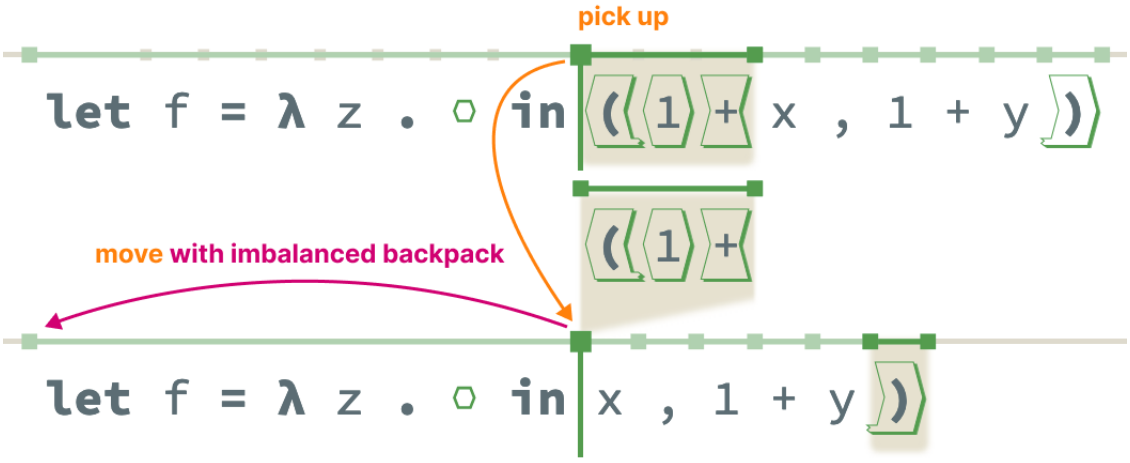
Another, more subtle breakdown was caused by the backpack's movement behavior changing dramatically given small changes in the picked-up selection. For example, consider the two edit sequences shown in Fig. 3.9. In the first sequence, the picked-up selection is balanced, so subsequently the user may 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 `tiny tylr`.

3.3.3 Limitations

Our study had several limitations. Our task design was constrained by `tiny 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



(a)



(b)

Figure 3.9: Two similar edit sequences showing the error-proneness of strictly backpack-guided movement. Intending to perform the first edit sequence in 3.9a, 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.

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 `tiny tylr`'s limitations, as described in §3.3.1. 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 Fig. 3.6a, 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.

3.4 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 `tiny 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 `tiny 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 `tiny 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 [53] 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.

CHAPTER 4

teen tylr: Gradual Structure Editing with Obligations

Preface

tiny tylr demonstrated a unique approach to structure editing, but its minimal, prototypal nature significantly limited its use and evaluation. Andrew Blinn and I sought to change that as we developed the next iteration, teen tylr. teen tylr largely adheres to the original term-tile-shard hierarchy introduced by tiny tylr, but situates it in a much more practical authoring tool. As of this writing, teen tylr continues to power the Hazel editor and its wide range of language and editor experiments.

Besides the obvious upgrades in scale (e.g. to multi-character tokens, multi-line layout), teen tylr boasts a more flexible, non-modal backpack system. Rather than restricting movement to syntactically valid targets to put down its contents like tiny's, teen's backpack simply monitors whether the current position is a valid target and prevents unloading if not. Where tiny's backpack could only handle one set of matching delimiters/selections at a time, disabling selection until it was empty, teen's backpack can grow a stack of any number of sets of matching delimiters/selections, either put there by the programmer or populated automatically as new matching requirements occur. Thanks to these simplifications, and the more practical scale, our study of teen is significantly more informative than the one for tiny.

Interested readers may play with teen tylr at https://tylr.fun/teen, the source of all screenshots in this chapter.

4.1 Contributions

We propose a new paradigm for structured code editing, called *gradual structure editing*, that aims to resolves the three problems of selection expressivity, delimiter matching, and term multiplicity

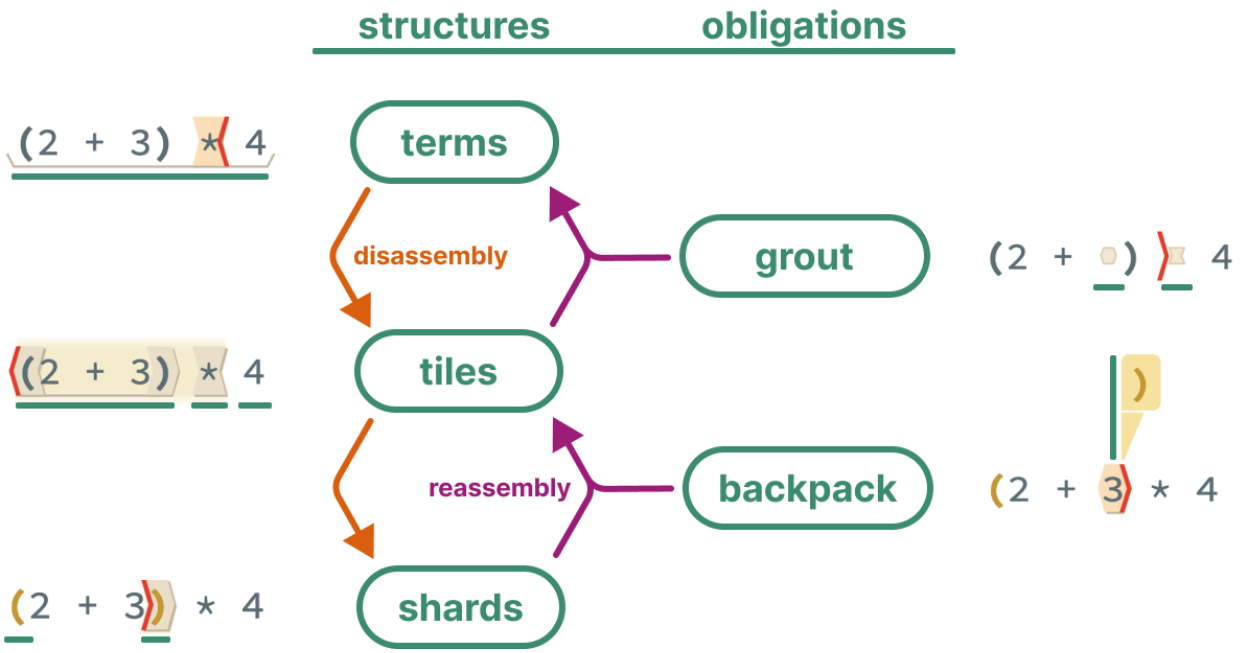


Figure 4.1: A high-level schematic of the concepts of tile-based editing, this chapter's realization of gradual structure editing.

(§2.1.2). 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 chapter'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. 4.1 and detailed in §4.2. 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. 4.1.

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. 4.1 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`, which is the source of all screenshots in this paper. After introducing the design of `teen tylr` in more detail (§4.2), we present the results (§4.4) of a lab study (§4.3) 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.

4.2 Design Overview

We now give an example-driven overview of tile-based editing using `teen tylr`. Fig. 4.2 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.

4.2.1 Terms, Tiles, Shards

Fig. 4.3a 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. For example, the outermost indicated function term in Fig. 4.3a 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. 4.3b. 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

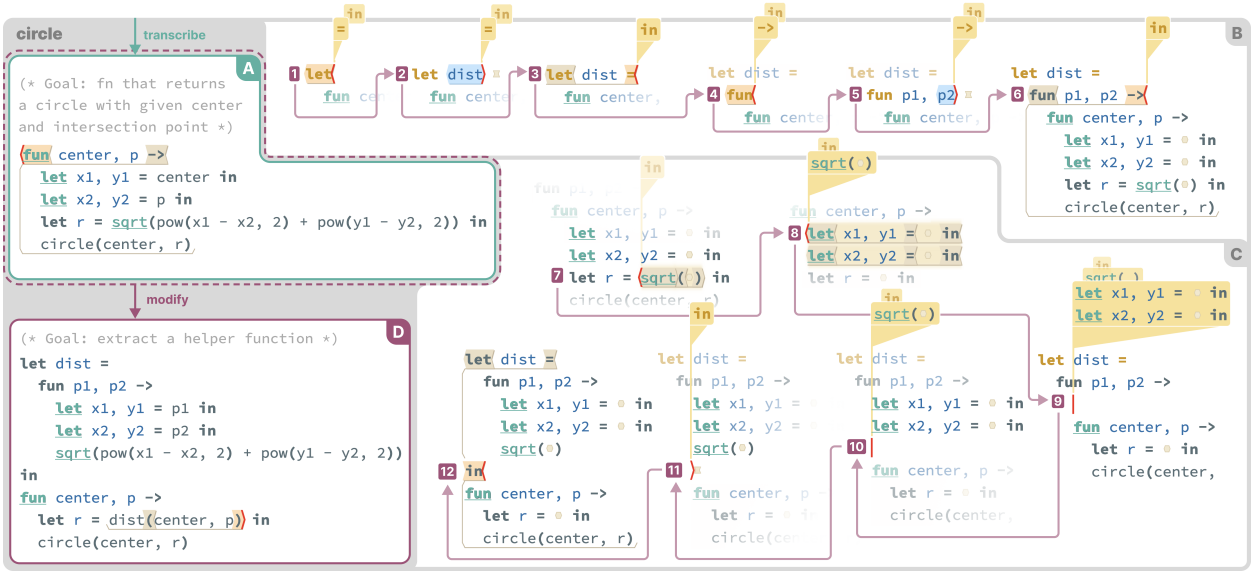


Figure 4.2: A pair of editing tasks we assigned our lab study participants, consisting of a **transcription** task (Panel A) followed by a **modification** task (Panel D), and the edit sequence by which participant P9 completed the modification task using `tylr` (Panels B & C). Due to space constraints, 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 `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.

Figure 4.3: Terms (a) and tiles (b) annotated with green borders.

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 $\langle\overline{\text{operand}}\rangle$, $\langle\overline{\text{prefix}}\langle$ operator, $\rangle\overline{\text{postfix}}\rangle$ operator, or $\rangle\overline{\text{infix}}\langle$ 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. 4.2 shows P9 using this capability to select the two `let`-bindings without their concluding body.

4.2.2 Terms ⇌ 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. 4.2, 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. 4.1, `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.

4.2.3 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. 4.2 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. 4.2. 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 (§4.4.2), and we discuss lifting this limitation in §4.5.

4.3 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?

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

Figure 4.4: The concrete syntax of Camel, a simple expression-based language we designed for our lab study. Camel is a near-subset of OCaml expressions and patterns—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 $\gg$ indicates forms to its left have greater precedence than those to its right.

4.3.1 Participants

Because our study tasks involved editing programs written in an expression-based 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-based languages (0.3-10 years, median 3.5 years). Eight participants had some prior experience with structured editing interfaces, such as Scratch [44] and ParEdit [13]; 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.

4.3.2 Tasks & Editors

The study tasks involved small program editing tasks in Camel, an OCaml-like expression-based language whose syntax is given in Fig. 4.4. Fig. 4.2 and Fig. 4.5 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 its grammar cells system [63] to operate

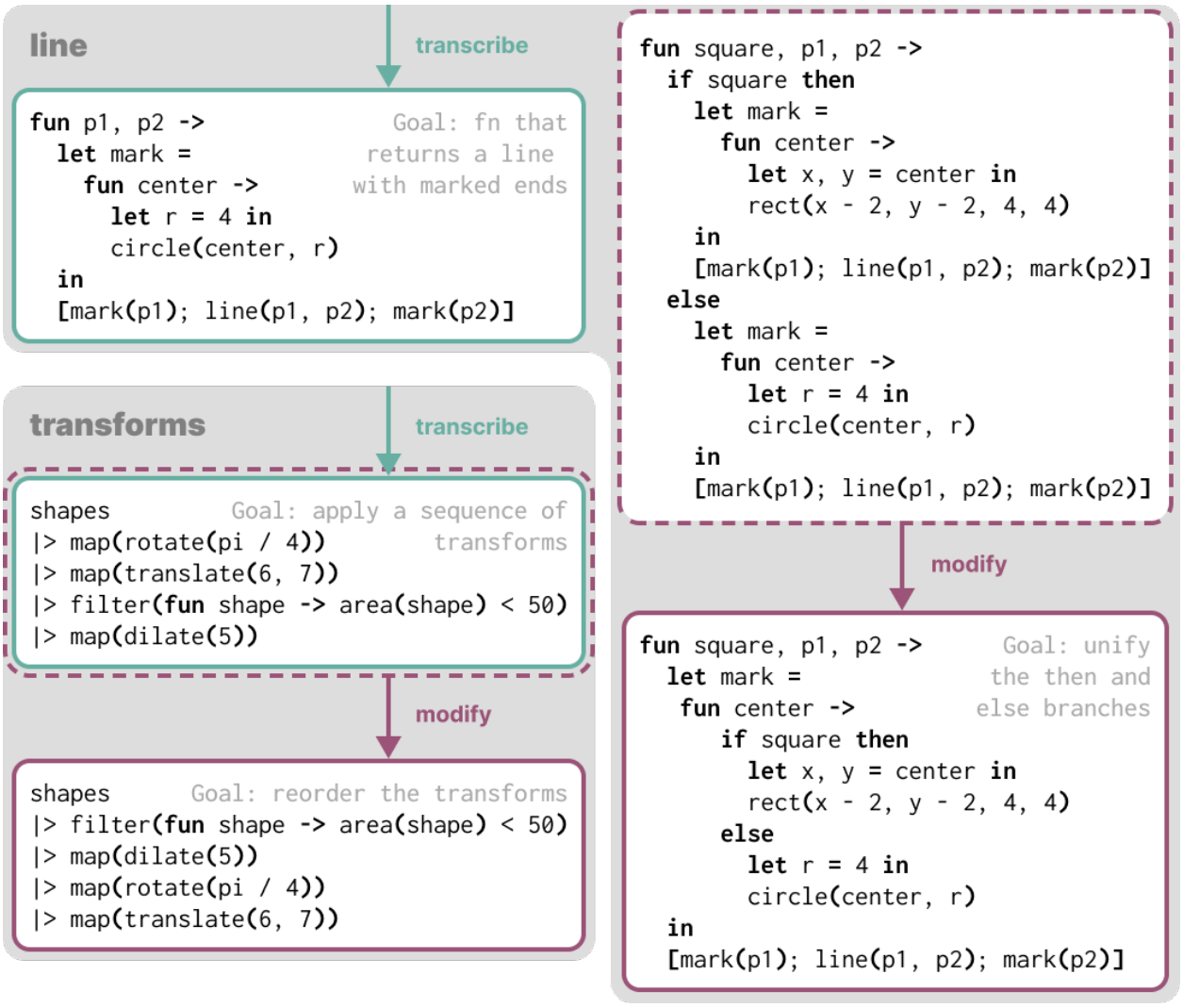
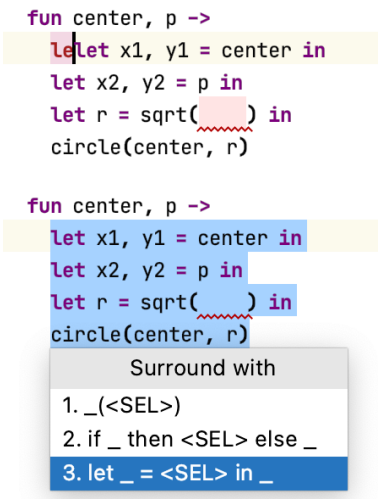


Figure 4.5: **Transcription-modification** task pairs `line` and `transforms`. See Fig. 4.2 for the third pair `circle`.



on Camel programs, as shown in screenshots on the left. 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 §4.2, 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 §4.4.4.

4.3.3 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 prepa-

ration 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.

4.4 Results

The lab study, administered by the second author of [47], 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.

§4.4.1 summarizes our quantitative results. After noting likely effects and patterns, we elaborate on specific causes in §4.4.2 and 4.4.3.

4.4.1 Completion Times, Mental Load, Code Reuse (Q1)

Given known limitations of null hypothesis significance testing [18], we base our analyses of time measures on estimated effect sizes with confidence intervals [19]. Fig. 4.6 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

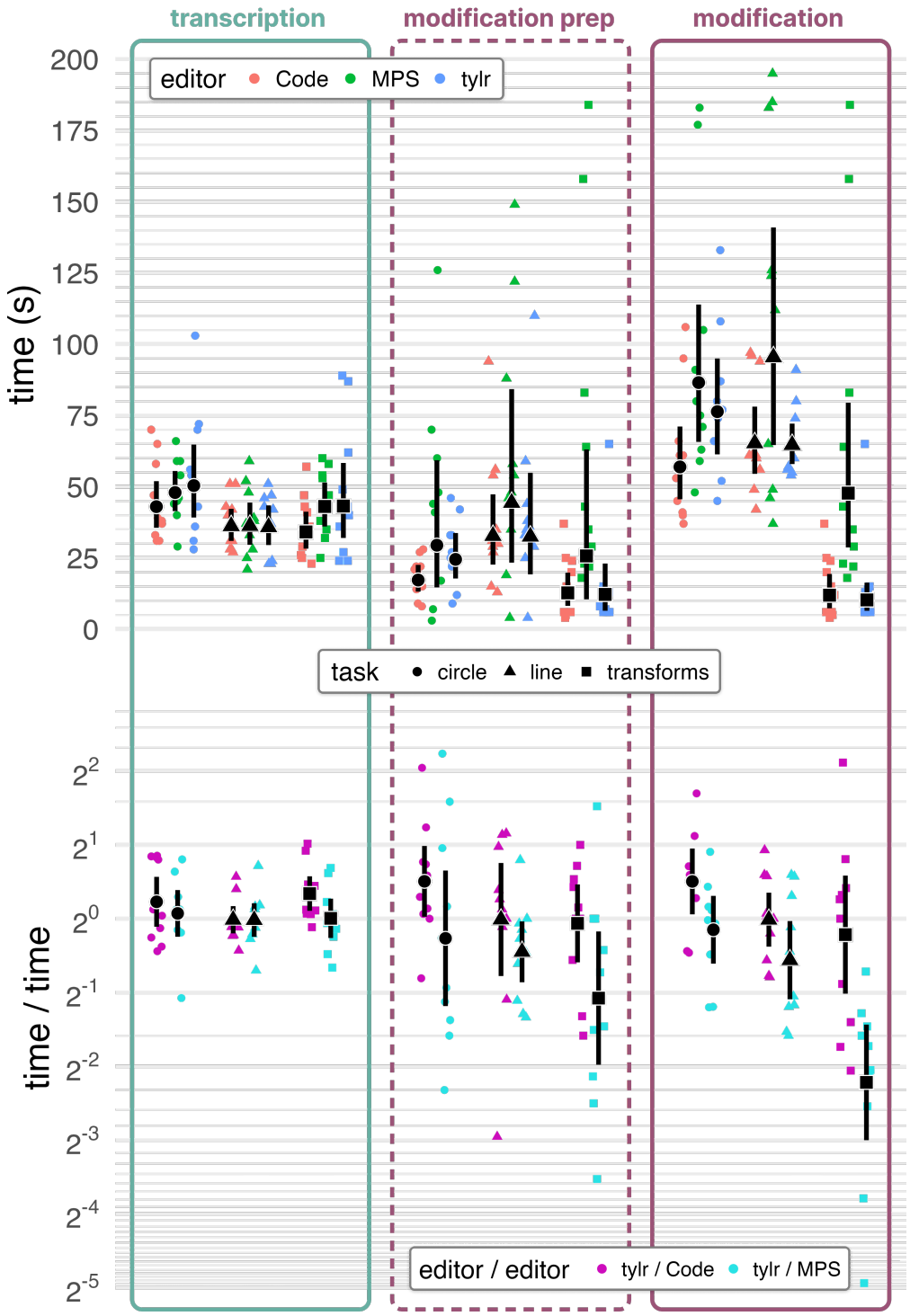


Figure 4.6: Dot plots overlaid with 95% confidence intervals summarizing how long participants took to prepare for and complete tasks with each editor. Each dot represents an individual participant measure. The top half shows the raw task times; the bottom half shows the relative slowdowns/speedups participants exhibited on each task using `tylr` compared to the other two editors. Confidence was calculated with the log of both measures to correct for positive skew.

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. 4.7. 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), 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. 4.8 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 §4.2.3). 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.

4.4.2 Mistakes, Inefficiencies, Frustrations (Q2)

The results of §4.4.1 indicate that participants struggled with MPS because of how mentally demanding it was to perform complex modifications, with notable impacts on both task perfor-

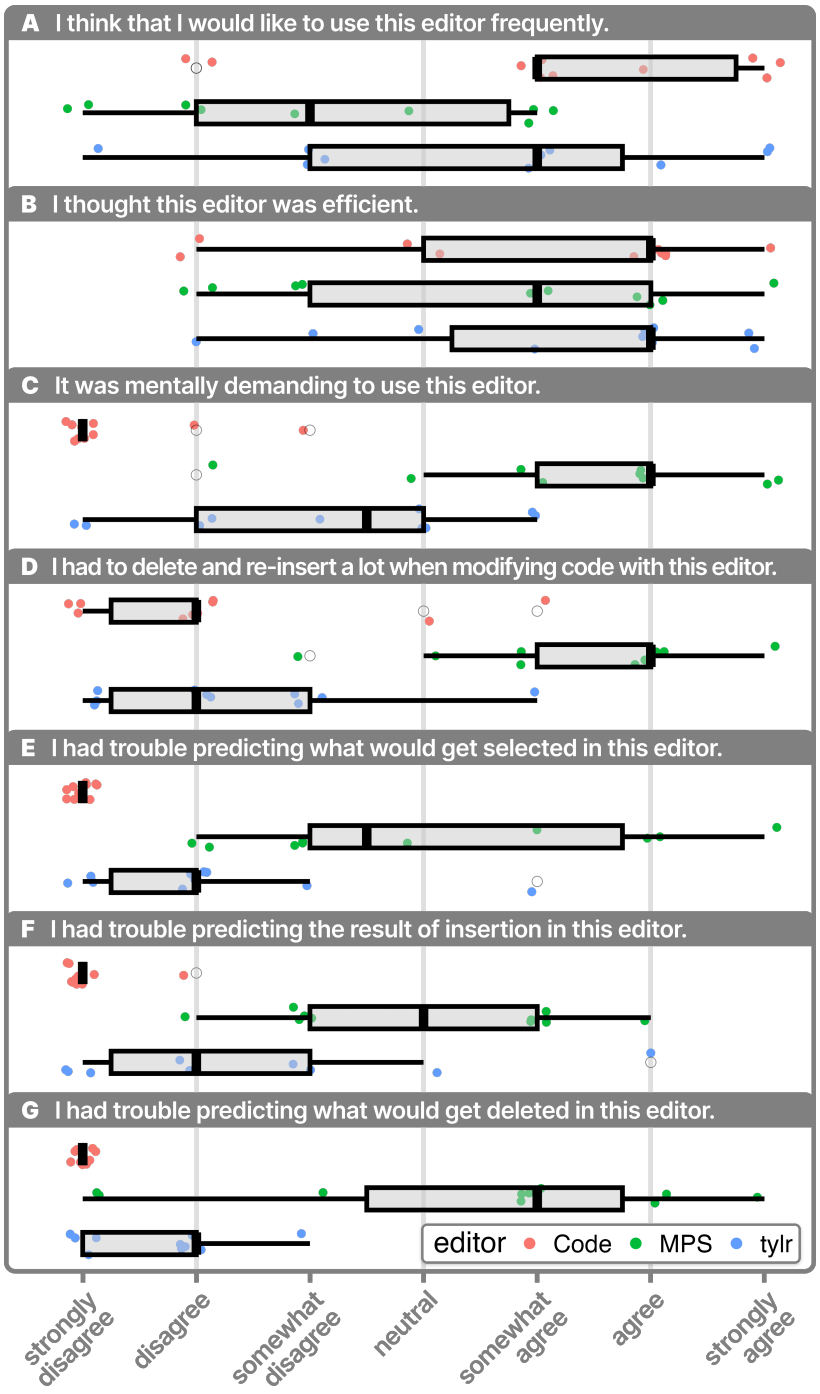


Figure 4.7: Box plots summarizing post-task survey responses.

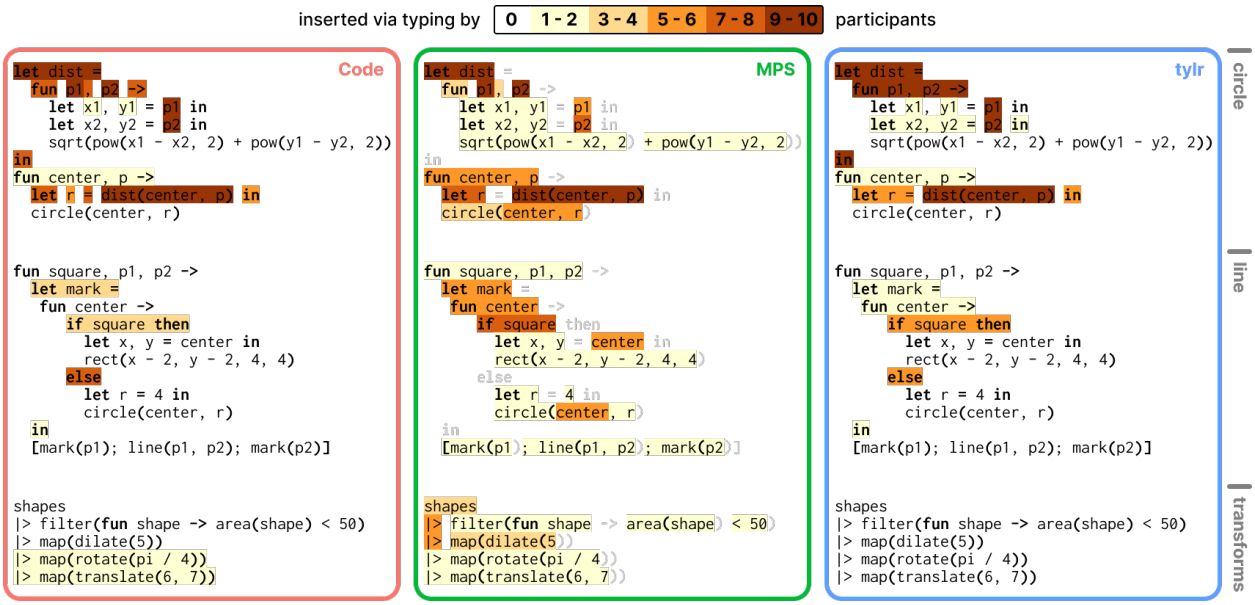


Figure 4.8: Heat maps summarizing code reuse in the modification tasks, measured by the number of participants that inserted via typing, rather than cutting and pasting, each token in the goal state. Delimiters auto-inserted by MPS are excluded.

mance 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 §2.1.2.

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 §4.2.3. 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. 4.8. 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."

4.4.3 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.

4.4.4 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 [22].

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.

4.5 Discussion

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-based 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.

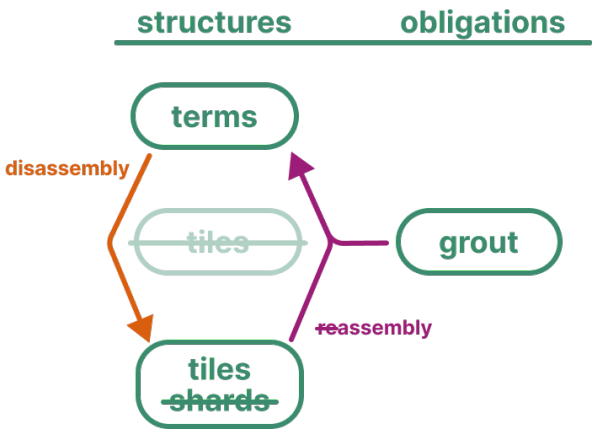


Figure 4.9: A simpler ontology for tile-based editing

Lifting these limitations in ongoing work has led us to simplify the tile-based editing ontology, from Fig. 4.1 to something closer to Fig. 4.9. 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 [16, 25], or otherwise one is chosen using ad hoc heuristics [30]. 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 [65], 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.

CHAPTER 5

`tall tylr`: Syntactic Completions with Material Obligations

Preface

After accepting that I was in fact designing an error-handling parser, I discovered Floyd's original description of operator-precedence (OP) parsing [29] and realized the three precedence comparisons (§2.2.2) corresponded precisely to different parts of the shard-tile-term hierarchy (Fig. 4.1): ≐-comparisons governed shard-tile relationships; ⋖/⋗-comparisons governed tile-term relationships. This observation encouraged me to shift to the simplified ontology in Fig. 4.9, with OP parsing sending tiles up to terms.

I learned that OP parsing enjoys the bounded context property, meaning substrings can be maximally reduced without backtracking, knowing only the delimiting tokens on either end. Others refer to this property as "local parsability" and have used it to parallelize OP parsing [7, 8]. I was interested in this property for editing and repair purposes, since this meant that any range of tokens, independent of its larger context, could be maximally parsed as well as repaired by completion.

I also learned that OP parsing imposes awkward constraints on its accepted grammars. The requirement that the precedence table be conflict-free makes it difficult to reuse the same token in different syntactic contexts, e.g. the minus symbol for both prefix negation and infix subtraction. Meanwhile, I was grappling with the design of `teen tylr`'s molder (a system not discussed in Chapter 4 because it was under development), used to promote textual input tokens to tiles by assigning each a "mold" specifying its sort and operator shape—the challenge was picking among multiple molds assignable to a token (e.g. prefix  vs infix  , expression  vs pattern  , etc). A solution to the molding problem, I realized, was a way to sidestep the grammar restrictions of OP parsing—by having the OP parser operate on (i.e. index its precedence table by) mold-distinguished tiles rather than raw tokens, we avoid the conflicts that might otherwise arise if we had to collapse together table rows/columns for tiles with the same token.

Altogether, we arrive at a unique conceptual division of the parsing process, dubbed *tile-based parsing*: (1) top-down, context-dependent *molding* that turns raw tokens into tiles; (2) bottom-up, context-independent *melding*—what I call our error-handling generalization of OP parsing—that repairs and reduces tiles to terms. This chapter contributes both a formalization of melding in §5.3 and a discussion of how we implement in molding in `tall tylr`, in particular by minimizing obligations.

Separately from its internal design around molding and melding tiles, `tall tylr` lets go of `teen tylr`'s backpack in favor of *ghosts* of missing tokens that appear inline in the editor. The benefit of this design is that there is no ambiguity, either to the system or the user, as to the current program structure, unlike the noncommital backpack. The challenge is choosing reasonable, predictable defaults, while also allowing the programmer to easily specify alternative choices.

Interested readers may play with `tall tylr` at `https://tylr.fun/tall`, the source of all screenshots in this chapter.

5.1 Contributions

This chapter introduces `tall tylr`, a parser and editor generator that performs syntax repair by *syntactic completion* (but not deletion) in a grammar enriched with *obligations*. This approach could be used for determining the internal program representation within an editor or language server (and indeed we expect this to be a common application of these ideas), but in this chapter we go further to investigate the potential of *materializing* these obligations visually to the programmer.

The screenshot in Fig. 1.4 shows how `tall tylr` repairs the program from Fig. 1.2A. On Line 1, `tall tylr` completes the user-inserted tokens `fun (p1 p2` by materializing three obligations. The first obligation, ▮, is an *infix obligation*, i.e. it ranges over many possible infix operators in pattern position. The remaining two obligations, `)` and `=>`, as well as those on Line 2, are *ghost obligations* that serve to complete the partially written syntactic forms. The user can accept these suggested locations by placing the cursor on these grayed out tokens and pressing the `Tab` key or typing over them explicitly. If they wish to place them elsewhere, the ghost obligations can be ignored and the user can type these obligations elsewhere; the ghost obligations are removed when no longer necessary. On Lines 3 and 5, the user has omitted operands of various syntactic sorts: one pattern, one type, and two expressions. `tall tylr` materializes *operand obligations* (a.k.a. *holes*), written ⬢ and colored by syntactic sort, to stand for the missing operands. Finally, on Line 4, ▮, is a *sort transition obligation* that indicates that there is a missing transition from the pattern sort (in blue) to the type sort (in purple), because in this language the token `->` can appear only in types. This collection of obligations captures the different ways a program may

be incomplete. We expand on this visual taxonomy with additional examples from the user's perspective in §5.2.

Each token and obligation in `tall tylr` has a color and a shape, collectively a *mold*. We refer to a token or obligation equipped with a mold as a *tile*. In particular, the color indicates the syntactic sort of term being considered. The shape indicates the hierarchical relationship between a token and its neighbors. For example, the convex tip on the left of the `let` token in Fig. 1.4 indicates that it is the beginning of a term, and the concave tip on its right indicates that a child term is expected to its right. Tips are visualized only for the term at the cursor (which is shown as a red angle in Fig. 1.4 to conform to the shape of its adjacent token) to avoid the visual clutter associated with nested block-based visualizations like those in systems like Scratch [15, 44].

Underlying this visual taxonomy is a novel theory of parsing that we dub *tile-based parsing*. Tile-based parsing departs from the predominant item-based approach of the LL/LR methods and instead builds on the token-based perspective of operator-precedence (OP) parsing as first described by Floyd [29]. OP parsing enjoys the *bounded context* property [33] that makes it possible to maximally parse any subrange of input knowing only its single-token delimiters, an attractive property for modeling and analyzing program edit states. On the other hand, OP parsing is also known for its limited grammar class, owing to difficulties reusing the same token in different structural roles (e.g. `-` for both infix subtraction and unary negation). With tile-based parsing, we propose splitting the overall problem of parsing into a top-down, context-dependent *molder* that molds tokens into tiles, thereby distinguishing one structural use of a token from another; and a bottom-up, bounded-context *melder* of tiles that extends OP parsing with obligation-based syntactic completions.

Where error handling is typically an afterthought in existing parsing methods, it emerges in tile-based parsing as a natural generalization of the core OP parsing method. In particular, we generalize the single-step precedence comparisons in OP parsing to multi-step precedence *walks* in melding, where the intermediate steps between the comparands constitute possible completions between them. In §5.3, we precisely specify melding as a parsing calculus called meldr. In addition to precedence walks, we describe in §5.3.3.2 how meldr "injects" the given grammar with additional grout forms that buffer the various inconsistencies that may arise between bottom-up reductions and top-down expectations. Along the way, we present in §5.3.1 a new parser-independent semantics for precedence annotations that generalizes and unifies prior accounts.

meldr describes a nondeterministic parser of tiles, leaving many decisions up to the implementation regarding how tiles are molded and completions are chosen. In §5.4, we describe the principle of *minimizing obligations* and additional heuristics that guide these decisions in `tall tylr`. Finally, we evaluate our overall design with a user study in §5.5, investigating both code

insertion and code modification tasks. We discover our design of materialized obligations has both promise and demand, but more design work is needed to give the programmer more control over their placement and removal, especially when modifying existing code.

5.2 Design Overview

We begin with a user-facing summary of how `tall tylr` operates in various common editing scenarios that demonstrate each form of obligation and how it is materialized to the user.

`tall tylr` is a parser and editor generator, i.e. it can be instantiated with various grammars. In this section, we will write programs in a simple expression-oriented programming language, Hazel [55]. Hazel is a near-subset of OCaml. One notable deviation is the use of postfix parentheses, $e(e)$, instead of infix space, $e\ e$, for function application. This allows us to demonstrate how `tall tylr` handles adjacency when whitespace is not accepted by the grammar as an infix operator.

5.2.1 Operand Obligations

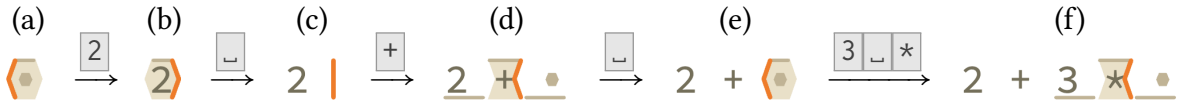


Figure 5.1: Basic expression insertion in `tall tylr`, demonstrating operand obligations and term decorations.

We begin with an empty editor buffer in Fig. 5.1(a). The root sort of Hazel is expression, and no expression has been entered, so `tall tylr` repairs the empty buffer to a single *operand obligation*, or simply *hole*, of that sort. Holes have convex tips on both sides, and the user's caret (in red) appears angled when on either side of the hole to emphasize its shape.

We next type the character 2, which causes the hole to be "filled" with the number literal 2 in Fig. 5.1(b). Atomic operands also have convex tips on both sides. The sort (here, expression) and the shape, i.e. the convexity of the tips on either side, are collectively called a *mold* and a token or obligation equipped with a mold is called a *tile*. Visually, the editor indicates the sort of a tile using color (here, expressions are grey) and the shape as shown above when the caret is on the tile. Next, we type a space, ␣, causing a space character to be inserted and the caret to shift right in Fig. 5.1(c). The caret is no longer on a tile, so no visual indications appear and the caret straightens out. Note that `tall tylr` only supports whitespace-insensitive grammars as of this writing.

Next, we type +. In the Hazel grammar, the + token is only used as an infix operator, so

the molder assigns it a shape with concave tips on both sides, as shown visually in Fig. 5.1(d). `tall tylr` must then perform syntax repair, because `2 +` is not accepted by the grammar. To do so, `tall tylr` performs a *precedence walk*. We will describe precedence walks precisely later in the paper, but for now, let us develop some intuition. The idea is that we need to walk from the current token, `+`, to the following token, which in this case is an implicitly included end-of-buffer token. The only walk which allows this is one that traverses the right operand of the form e `+` e. Consequently, `tall tylr` repairs the syntax by inserting an expression-sorted operand obligation, i.e. hole, as shown. For convenience, `tall tylr` also automatically inserts the space between the operator and the hole. When we subsequently type ␣, this automatically inserted space is "consumed" rather than causing the insertion of a second space, as shown in Fig. 5.1(e).

Finally, we continue typing as shown, resulting in Fig. 5.1(f). Operator sequences are parsed according to Hazel's precedences and associativities. Notice in both Fig. 5.1(d) and Fig. 5.1(f) that `tall tylr` underlines the associated operands when the caret is on a tile to visually communicate the structure of the overall term. Notice also that completed terms are always convex on both sides. Indeed, a user's mental model can simply be that `tall tylr` inserts obligations to maintain visual convexity.

5.2.2 Infix Obligations

Starting from the editor state in Fig. 5.2(a), we press backspace, ⌫, deleting the `+` tile. Textually, this would result in the operands `2` and `3` appearing adjacent to one another, which is not accepted by the Hazel grammar. There are many possible walks between adjacent terms—one for each of the infix operators—so `tall tylr` abstracts over them by inserting an *infix obligation*, a.k.a. an *operator hole*, as shown in Fig. 5.2(b). Infix obligations have the lowest precedence.

Figure 5.2: Adjacent operands are connected by infix obligations in `tall tylr`.

One way to think about this mechanism is that operand obligations arise when one term is expected but zero terms appear, whereas infix obligations arise when one term is expected but many adjacent terms appear, as is often the case transiently during edits.

5.2.3 Molding Ambiguity

The situation becomes more interesting if we use the `-` token, because in the Hazel grammar this token can appear both as an infix operator (subtraction) and as a prefix operator (negation). These correspond to different

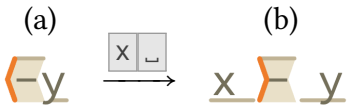


Figure 5.3: The minus sign has multiple molds. The mold is chosen to minimize obligations.

molds. For example, in Fig. 5.3(a), the - token before y is molded as a prefix operator, visualized with a convex tip on the left and a concave tip on the right as shown.

If in this position, we type x followed by ␣ (to move the caret over the automatically inserted space), `tall tylr` remolds the token into the corresponding infix operator as shown in Fig. 5.3(b). The reason is `tall tylr`'s novel approach to disambiguation: `tall tylr` always selects the mold which locally minimizes the number of obligations that must be inserted. Retaining the prefix mold would have required also inserting an infix obligation, whereas the infix mold requires no obligations.

Although this approach requires considering alternative token moldings as tokens are encountered, we note that there are generally only a few tokens in a typical grammar which can have multiple possible moldings. In the Hazel grammar, only - and (have this property. Traditional operator precedence parsing cannot handle such grammars, but using a molder separate from the core parsing algorithm that makes decisions based on repair costs allows us to overcome this expressiveness limitation while retaining a relatively simple core parsing algorithm.

Note that formally, a mold is not simply a shape and sort, but rather a zipper into the grammar, so the molder is also responsible for resolving other parsing ambiguities that might arise as well, e.g. the famous "dangling else" problem in imperative languages. This is in contrast to approaches where the parser resolves these ambiguities, e.g. by favoring shifts over reduces. We leave to future work the problem of declaratively specifying disambiguation policies in this setting. We only work with unambiguous grammars in the remainder of the paper.

5.2.4 Ghost Obligations

Figure 5.4: Ghost obligations are inserted for mixfix forms in `tall tylr`.

So far, our examples have only used infix operators. Introducing *mixfix operators* requires enriching our language of obligations to handle mixfix delimiters that have not yet been inserted.

For example, starting from an empty buffer in Fig. 5.4(a), we can insert a let expression by typing l e t ␣. This causes insertion of *ghost obligations*, shown in gray in Fig. 5.4(b). These obligations are again determined by computing a precedence walk from the inserted token to the next token. When walking over a token that is not explicitly in the editor state, we can include it as a ghost obligation in the corresponding completion. We again choose the completion that minimizes obligations. The user can continue by entering a variable to fill the pattern hole at the

caret and, when they reach the =, they can either press tab, ⟶|, or type over the ghost character(s), here by entering =. Either choice will result in the state shown in Fig. 5.4(c). Note that the term structure is visualized despite the missing delimiter.

When inserting a let expression in the middle of an existing program, `tall tylr` needs to heuristically decide where to place the ghost obligations. The heuristic we use is is based primarily on newline placement in the buffer, summarized by the example in Fig. 5.5. If we insert the let expression immediately before existing code on the same line, that code is placed in the first child position of the same sort as shown in Fig. 5.5(a-b). If instead we enter the let expression on a blank line, subsequent lines are placed in the last child position of the same short as shown in Fig. 5.5(c-d).

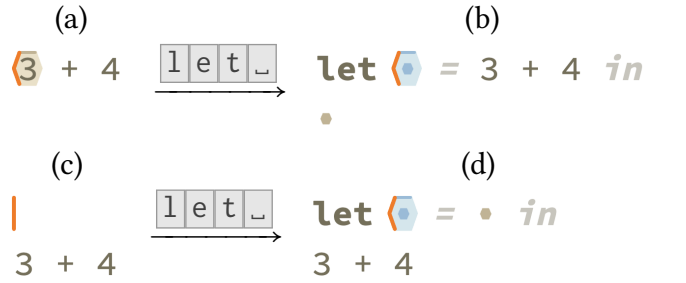


Figure 5.5: Ghost obligation placement is chosen heuristically, here based on newline locations.

If the user's intent differs from this heuristic placement, they can ignore the ghost obligations and insert the obligation explicitly where they intend. Given the state in Fig. 5.6(a), if the user enters `in` at the end of the buffer, `tall tylr` would clean up the ghost `in` and restructure the code as shown in Fig. 5.6(b). Note that `tall tylr` automatically manages indentation.

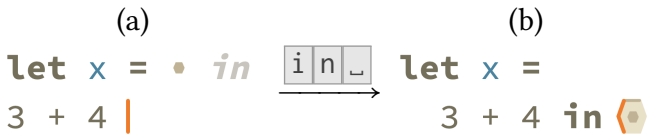


Figure 5.6: Ghost obligations can be ignored and are cleaned up if entered elsewhere.

5.2.5 Sort Transition Obligations

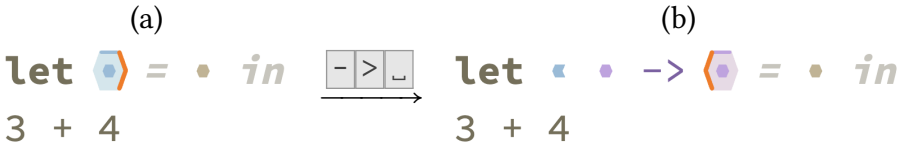


Figure 5.7: Sort transition obligations are needed when entering forms that are not sort-correct.

Some syntactic forms are legal only when entering terms of a particular sort. For example, in Hazel, the arrow operator, ->, can only appear in types. If we enter the arrow in pattern position, as shown in Fig. 5.7(a), `tall tylr` inserts obligations indicating that there needs to be a sort transition from the pattern sort to the type sort, as shown in Fig. 5.7(b). If there were additional text on the right that could be parsed as a pattern, a sort transition "back" on the right side would also appear.

5.2.6 Unmolded Tokens

Figure 5.8: Unrecognized tokens are left unmolded, and therefore cannot fulfill obligations.

Finally, some tokens are not recognized by the grammar at all. To handle these, the molder marks them as unmolded tokens and treats them like whitespace or comments, i.e. they do not have a shape and so do not participate in obligation insertion. For example, in Hazel, there is no ! token, so `tall tylr` simply marks it in red and ignores it as shown in Fig. 5.8. Notice that no matter where the ! token appears, the operand obligation remains unfilled.

5.3 meldr

We now present an error-handling parser calculus, called meldr, that describes how to complete token sequences with additional tokens such that they can be parsed into grammatical terms. Given a language grammar, whose terminal symbols we call *tiles*, we "inject" it with additional *grout* forms that either stand in for missing terms or else wrap sort-inconsistent and extraneous terms. From this grout-injected grammar, we generate an error-handling parser of tile sequences that completes its input with grout and additional requisite tiles (manifesting as ghosts in `tall tylr` (§5.2.4)) such that it can be parsed. By first relaxing grammaticality with grout, we ensure that the generated tile parser is total over all inputs (Theorem 2).

As a substrate for these ideas, we generalize and unify two prior accounts of operator precedence: Aasa's semantics for precedence annotations in grammars [3] and Floyd's seminal introduction of operator-precedence parsing (OP parsing) [29]. The two works have related but complementary scopes: Aasa describes how *precedence annotations* act as filters on the set of valid derivation trees of the underlying grammar, as well as how to elaborate the annotated grammar into an unannotated one; meanwhile, Floyd begins with an unannotated grammar and describes how to derive a set of *precedence relations* between terminal symbols that can be used to steer a bottom-up parser. In §5.3.1, we specify a new elaboration from annotated grammars $\mathcal{G}$ to unannotated grammars $\mathcal{H}$ that simplifies Aasa's version and generalizes it to allow for arbitrary mixfix forms of varying sorts in $\mathcal{G}$. To demonstrate correctness, we show in §5.3.1.4 that Floyd's precedence relations derived from $\mathcal{H}$ cohere as expected with the annotations in $\mathcal{G}$ (Theorem 1).

Next, we generalize OP parsing to handle errors using completion-only repair. After reviewing Floyd's original parsing method and noting the various ways that it can "go wrong" in §5.3.2, we present our error-handling variation in §5.3.3. Among other things, our approach generalizes the single-step precedence comparisons between neighboring input tokens that steer an OP parser to multi-step precedence *walks*, where the intermediate steps constitute a possible completion

$$
\begin{array}{rrcl}
\text{tile} & t & \in & \mathcal{T} \\
\text{sort} & r, s & \in & \mathcal{S} \supseteq \{\hat{s}\} \\
\text{symbol} & x, y & ::= & t \mid s \\
\text{regex} & g & ::= & \epsilon \mid x \mid g \mid g \mid g \cdot g \mid g^* \\
\text{precedence} & m, n, p, q & \in & \mathcal{P} = \mathbb{N} \sqcup \{\bot, \top\} \\
\text{PBG} & \mathcal{G} & \in & \mathcal{S} \to \mathcal{P} \to g
\end{array}
$$

Figure 5.9: Syntax of precedence-bounded grammars

$$
\begin{array}{rrcl}
\text{terminal} & \tau & ::= & \prec \mid t \mid \succ \\
\text{nonterminal} & \rho, \sigma & ::= & {}^{p}s^{q} \\
\text{symbol} & \chi & ::= & \tau \mid \sigma \\
\text{CFG} & \mathcal{H} & \in & \{\sigma \Rightarrow \overline{\chi}\}
\end{array}
$$

Figure 5.10: Syntax of elaborated context-free grammars

between the tokens.

This approach alone is not quite sufficient to guarantee a successful parse across all grammars and inputs—moreover, in practice, it would require the parser to make many heuristic choices between structurally identical tile completions. To remedy these issues, we describe in §5.3.3.2 how to inject grout forms into the translated grammar $\mathcal{H}$, which serve as fallbacks when no tile-only completion exists, and also as natural defaults when there are many possible choices. Given these fallbacks, we show that the generated parser is sound and total over all inputs (Theorem 2).

Notation Throughout this section, we will use the following notation for options and sequences given an element type α.

$$
\begin{array}{rrcl}
\text{option} & \alpha? & ::= & \circ \mid \bullet\alpha \\
\text{sequence} & \overline{\alpha} & ::= & \cdot \mid \alpha\overline{\alpha}
\end{array}
$$

Given a judgment form $J\ \alpha$, we will write $J\ \alpha?$ to mean either $\alpha? = \circ$ or else $\alpha? = \bullet\alpha$ such that $J\ \alpha$ holds. Similarly, given $J\ \alpha\ \beta$, we will write $J\ \alpha?\ \beta?$ to mean either $\alpha? = \circ$ and $\beta? = \circ$ or else $\alpha? = \bullet\alpha$ and $\beta? = \bullet\beta$ such that $J\ \alpha\ \beta$ holds.

5.3.1 Elaborating Precedence Annotations

5.3.1.1 Precedence-Bounded Grammars

Our calculus is parametrized by a language grammar $\mathcal{G}$ in EBNF form with precedence annotations, what we call in this work a *precedence-bounded grammar (PBG)*. Compared to ordinary context-free grammars, where precedence must be encoded in tedious towers of dependent production rules, PBGs allow language forms of the same semantic *sort* (e.g. expressions vs patterns vs types) to be organized under a single named entity, leading to more natural and concise grammar definitions. By having the author explicitly specify the language's sorts, PBGs also help us generate a minimal set of semantically meaningful grout forms.

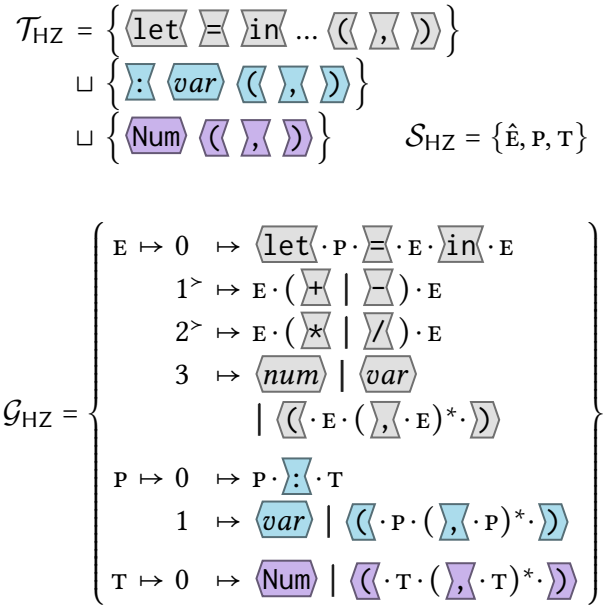


Figure 5.11: A PBG $\mathcal{G}_{\text{HZ}}$ for a small expression-oriented language. Sorts consist of expressions (E) in grey, patterns (P) in blue, and types (T) in purple. Tiles are distinguished by text, shape, and color-coded sort.

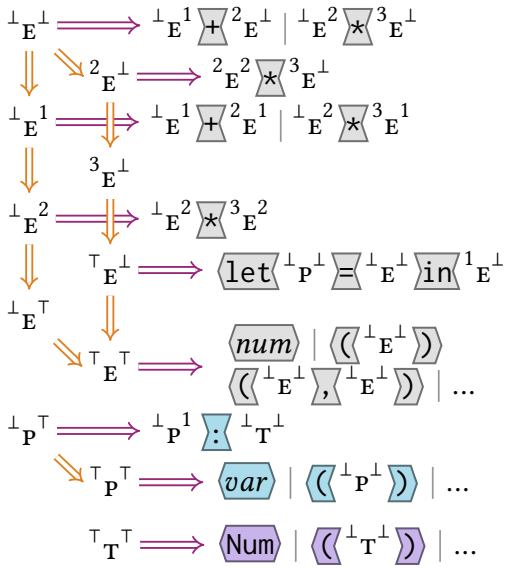


Figure 5.12: An excerpt of the CFG $\mathcal{H}_{\text{HZ}}$ elaborated (Fig. 5.15) from $\mathcal{G}_{\text{HZ}}$ (Fig. 5.11). The production rules are arranged and color-coded by whether each is elaborated by subsuming reduction or by tightening.

The syntax of PBGs is given in Fig. 5.9. A PBG $\mathcal{G}$ is a partial function mapping a sort $s \in \mathcal{S}$ and precedence $p \in \mathcal{P}$ to a regex g over symbols x, each either a tile $t \in \mathcal{T}$ or a sort $s \in \mathcal{S}$. We assume that $\mathcal{S}$ includes a designated start sort $\hat{s}$. We assume $\mathcal{P} = \mathbb{N} \sqcup \{\bot, \top\}$ includes the natural numbers $\mathbb{N}$ for precedence levels assigned in $\mathcal{G}$ as well as minimum $\bot$ and maximum $\top$ precedence levels that are reserved for internal use. We further assume that $\mathcal{P}$ is equipped with ordering relations $\prec_s, \succ_s$ that abstract the details of associativity for each sort $s \in \mathcal{S}$. For example, $5 \prec_{\text{E}} 5$ would encode that infix operators at precedence level 5 of sort E are right-associative—otherwise, outside of reflexive pairs, these relations coincide with the usual ordering relations on natural numbers. For all sorts $s \in \mathcal{S}$, we assume $p \prec_s \top \succ_s p$ and $p \succ_s \bot \prec_s p$ for all other $p \in \mathbb{N}$.

Fig. 5.11 gives a concrete PBG $\mathcal{G}_{\text{HZ}}$ encoding an excerpt of Hazel expressions, patterns, and types, which we will use as a running example throughout the rest of the paper. Here, each precedence level $p \in \mathbb{N}$ of sort s is optionally tagged with the relation $\otimes \in \{\prec_s, \succ_s\}$ that applies to the reflexive pair $p \otimes p$, if any—in this case, levels $1^{\succ}$ and $2^{\succ}$ of sort E are marked as left-associative, i.e. $1 \succ_{\text{E}} 1$ and $2 \succ_{\text{E}} 2$. Meanwhile, the tiles are distinguished by their shape and color (gray for expressions, blue for patterns, purple for types) in addition to their text.

A regex g is either ϵ, matching the empty string; a symbol x; a choice $g_L \mid g_R$; a concatenation $g_L \cdot g_R$; or a Kleene star g^*. Its language $[\![g]\!]$ of matching symbol strings $\overline{x}$ is defined as follows:

$$\llbracket \epsilon \rrbracket = \{\cdot\} \qquad \llbracket g_L \mid g_R \rrbracket = \llbracket g_L \rrbracket \cup \llbracket g_R \rrbracket \qquad \llbracket g^* \rrbracket = \bigcup_{k \in \mathbb{N}} \llbracket g^k \rrbracket \quad \text{where } g^0 = \epsilon \text{ and } g^{k+1} = g \cdot g^k$$
$$\llbracket x \rrbracket = \{x\} \qquad \llbracket g_L \cdot g_R \rrbracket = \{\overline{x}_L \overline{x}_R \mid \overline{x}_L \in \llbracket g_L \rrbracket, \overline{x}_R \in \llbracket g_R \rrbracket\}$$

A $\mathcal{G}$*-form* is a symbol string $\overline{x} \in \llbracket \mathcal{G}(s,p) \rrbracket$ for any $s \in \mathcal{S}, p \in \mathcal{P}$. To make use of operator-precedence parsing techniques, we assume that every $\mathcal{G}$-form is in *operator form* [29]:

Assumption 1 (Operator Form)**.** *There exist no sorts $s_L, s_R \in \mathcal{S}$ and regex $\mathcal{G}(s,p)$ such that $\ldots s_L s_R \ldots \in \llbracket \mathcal{G}(s,p) \rrbracket$.*

In other words, every $\mathcal{G}$-form may be written in the form $s?_0 [\, t_i s?_{i+1} \,]_{0 \le i \le k}$. Greibach [32] showed that every CFG can be normalized to a strongly equivalent one in operator form.

5.3.1.2 Context-Free Grammars

We assign meaning to the precedence-annotated grammar $\mathcal{G}$ by elaborating it to an unannotated context-free grammar (CFG) $\mathcal{H}$, whose syntax is given in Fig. 5.10. An elaborated CFG $\mathcal{H}$ is a (possibly infinite) set of production rules $\sigma \Rightarrow \overline{\chi}$, each mapping a nonterminal σ to a finite sequence $\overline{\chi}$ of symbols—we will refer to σ and $\overline{\chi}$ as the rule's *producer* and *product*. Each symbol χ is either a terminal τ or a nonterminal σ. A terminal symbol τ is either a tile t or a root delimiter, $\prec$ or $\succ$, marking the start or end of input. Meanwhile, a nonterminal is a *precedence-bounded sort* ${}^{p}s^{q}$, where $p, q \in \mathcal{P}$ will serve as constraints from the left and right sides of the nonterminal in the overall production tree. Fig. 5.12 shows an excerpt of the CFG $\mathcal{H}_{\mathsf{HZ}}$ elaborated from $\mathcal{G}_{\mathsf{HZ}}$, the process of which we discuss in §5.3.1.4.

We have specialized the symbols here to serve as our elaboration outputs, but their rewriting semantics are standard: given a production rule $\sigma \Rightarrow \overline{\chi}$, we say that the symbol string $\overline{\chi}_L \sigma \overline{\chi}_R$ *produces* the string $\overline{\chi}_L \overline{\chi} \overline{\chi}_R$, written $\overline{\chi}_L \sigma \overline{\chi}_R \Rightarrow \overline{\chi}_L \overline{\chi} \overline{\chi}_R$ reusing the production rule syntax. A production sequence $\overline{\chi}_0 \Rightarrow \overline{\chi}_1 \Rightarrow \ldots$ from the designated start string $\overline{\chi}_0 = \prec^{\perp} \hat{s}^{\perp} \succ$ is called a *derivation*; the language of a CFG collects all of its derivable strings.

5.3.1.3 Precedence Comparisons

Given a CFG, we may generate a collection of *precedence comparisons* that classify derivation patterns between neighboring terminals. Each comparison $\tau_L \odot_{\rho?} \tau_R$ means there exists a derivable string with the substring $\tau_L \rho? \tau_R$, which consists of neighbors τ_L, τ_R that are either adjacent ($\rho? = \circ$) or separated ($\rho? = \bullet\rho$) by a nonterminal ρ. Floyd's original definition [29] does not surface the operator index $\rho?$, whose omission we will later show contributes to an unsound parsing method

comparison $\odot ::= \lessdot \mid \doteq \mid \gtrdot$

$\boxed{\tau_L \odot_{\rho?} \tau_R}$ τ_L compares with τ_R (over $\rho?$)

Prec-LT
$$\frac{\tau_L \lhd \sigma \qquad \sigma \Rightarrow^* \rho? \tau_R \ldots}{\tau_L \lessdot_{\rho?} \tau_R}$$

Prec-EQ
$$\frac{\sigma \Rightarrow \ldots \tau_L \rho? \tau_R \ldots}{\tau_L \doteq_{\rho?} \tau_R}$$

Prec-GT
$$\frac{\sigma \Rightarrow^* \ldots \tau_L \rho? \qquad \sigma \rhd \tau_R}{\tau_L \gtrdot_{\rho?} \tau_R}$$

Figure 5.13: Precedence comparisons

τ_L \ τ_R	≻	let	=	in	+	*	(	)	num	:	(	)	Num
≺	≐	⋖			⋖	⋖	⋖		⋖				
let			≐							⋖	⋖		
=		⋖		≐	⋖	⋖	⋖		⋖				
in	⋗	⋖		⋗	⋖	⋖	⋖	⋗	⋖				
+	⋗	⋖		⋗	⋗	⋖	⋖	⋗	⋖				
*	⋗	⋖		⋗	⋗	⋗	⋖	⋗	⋖				
(		⋖			⋖	⋖	⋖	≐	⋖				
)	⋗			⋗	⋗	⋗		⋗					
num	⋗			⋗	⋗	⋗		⋗					
:			⋗							⋗		⋗	⋖
(										⋖	⋖	≐	
)			⋗							⋗		⋗	
Num			⋗									⋗	

Figure 5.14: An excerpt of precedence comparisons $\tau_L \odot \tau_R$ for $\mathcal{H}_{\mathsf{HZ}}$ (Fig. 5.12)

(§5.3.2), and whose use in our resolution we describe in §5.3.3. Until then, we will similarly omit it from the notation $\tau_L \odot \tau_R$—note this is different from assuming $\rho? = \circ$, which we will always notate explictly $\tau_L \odot_\circ \tau_R$.

The comparison operator $\odot \in \{\lessdot, \doteq, \gtrdot\}$ indicates in what relative order the neighbors τ_L, τ_R were first produced in the derivation. Fig. 5.14 shows an excerpt of the precedence comparisons for $\mathcal{H}_{\mathsf{HZ}}$ (Fig. 5.12). The derivation ${}^{\bot}\mathtt{E}^{\bot} \Rightarrow {}^{\bot}\mathtt{E}^{1} \langle + {}^{2}\mathtt{E}^{\bot} \Rightarrow {}^{\bot}\mathtt{E}^{1} \langle + {}^{2}\mathtt{E}^{2} \langle * {}^{3}\mathtt{E}^{\bot}$ tells us $\langle + \lessdot \langle *$ (“$\langle +$ binds less tightly than $\langle *$”) since $\langle +$ is produced before its neighbor $\langle *$. Meanwhile, the derivation ${}^{\bot}\mathtt{E}^{\bot} \Rightarrow {}^{\top}\mathtt{E}^{\top} \Rightarrow \langle ({}^{\bot}\mathtt{E}^{\bot}) \rangle$ tells us $\langle (\doteq) \rangle$ (“$\langle ($ matches $) \rangle$”) since neighbors $\langle ($ and $) \rangle$ are produced together. Keep in mind that the written order of arguments τ_L, τ_R in each comparison $\tau_L \odot \tau_R$ reflects their sequential order in the derived string, so we should not generally expect $\tau_L \lessdot \tau_R$ to be equivalent to $\tau_R \gtrdot \tau_L$, nor for $\doteq$ to be symmetric.

5.3.1.4 Precedence Elaboration

Elaboration turns an annotated PBG $\mathcal{G}$ into an unannotated CFG $\mathcal{H}$ whose nonterminals internalize relevant bounding annotations. Governing its design is the expectation that precedence comparisons $t_L \odot t_R$ between tiles in $\mathcal{H}$ mirror, when relevant, numeric comparisons between the tiles' backing annotations in $\mathcal{G}$ (Theorem 1).

Theorem 1 (Annotation-Comparison Coherence). *For all sorts s, precedence levels p_L, p_R, and tiles*

t_L, t_R such that $\ldots t_L s \in [\![\mathcal{G}(s, p_L)]\!]$ and $s t_R \ldots \in [\![\mathcal{G}(s, p_R)]\!]$, the following equivalences hold:

$$t_L \lessdot t_R \iff p_L <_s p_R \qquad t_L \gtrdot t_R \iff p_L >_s p_R$$

To motivate our design (Fig. 5.15), it is instructive first to consider the issues with simpler alternatives—in particular, elaborating to nonterminals with fewer than two bounds. If we elaborated the PBG $\mathcal{G}$ to a CFG $\mathcal{H}_0$ in which the nonterminals were plain unbounded sorts s, the best we could do is a trivial elaboration that simply ignores the precedence annotations in $\mathcal{G}$:

$$\mathcal{H}_0 \triangleq \{s \Rightarrow \overline{x} \mid \overline{x} \in [\![\mathcal{G}(s, p)]\!], p \in \mathcal{P}, s \in \mathcal{S}\}$$

$\mathcal{H}_0$ allows for problematic derivations like $\mathrm{E} \Rightarrow \langle \mathrm{E}\ \boxed{*}\ \mathrm{E} \rangle \Rightarrow \langle \langle \mathrm{E}\ \boxed{+}\ \mathrm{E} \rangle\ \boxed{*}\ \mathrm{E} \rangle$, problematic because it witnesses the unwanted comparison $\boxed{+} \gtrdot \boxed{*}$ ("$\boxed{+}$ binds more tightly than $\boxed{*}$").

A better approach—similar in effect to that of Danielsson and Norell [20] and of Klint and Visser [40]—would use singly-bounded nonterminals s^p and limit their productions to $\mathcal{G}$-forms of equal or stronger precedence: $\mathcal{H}_1 \triangleq \{s^p \Rightarrow \lceil \overline{x} \rceil^p \mid \overline{x} \in [\![\mathcal{G}(s, q)]\!], p \preceq_s q, s \in \mathcal{S}\}$ where $\lceil \overline{x} \rceil^p$ lifts each sort symbol $s \in \overline{x}$ to some suitably bounded nonterminal. This approach properly rules out unwanted derivations on the left like (a) (for witnessing $\boxed{+} \gtrdot \boxed{*}$) and (b) ($\boxed{\texttt{in}} \gtrdot \boxed{*}$), since the left argument E^2 of $\boxed{*}$ cannot produce the $\boxed{+}$- and $\langle\boxed{\texttt{let}}$-forms of weaker precedence levels 1 and 0. However, $\mathcal{H}_1$ is overly conservative: it also rules out acceptable derivations like (c) ($\boxed{*} \lessdot \langle\boxed{\texttt{let}}$). Ultimately the purpose of precedence annotations is to resolve choices between different possible reduction orders: given a reduced child, which of the operators on either side of it should be reduced next as part of its parent? Derivations (a) and (b) represent disfavored choices of reducing the left parent ($\boxed{+}$ and $\boxed{\texttt{in}}$) before the right ($\boxed{*}$) over the reduced children E^2 and E^0, respectively. On the other hand, (c) has no viable alternative reduction order, since $\langle\boxed{\texttt{let}}$ cannot parent a child to its left. In such cases, the precedence annotations need not be consulted.

(a) $\mathrm{E}^\perp \Rightarrow \langle \mathrm{E}^2\ \boxed{*}\ \mathrm{E}^3 \rangle \Rightarrow \langle \langle \mathrm{E}^1\ \boxed{+}\ \mathrm{E}^2 \rangle\ \boxed{*}\ \mathrm{E}^3 \rangle$
(b) $\mathrm{E}^\perp \Rightarrow \langle \mathrm{E}^2\ \boxed{*}\ \mathrm{E}^3 \rangle \Rightarrow \langle \langle \langle\boxed{\texttt{let}}\ \mathrm{P}^\perp\ \boxed{=}\ \mathrm{E}^\perp\ \boxed{\texttt{in}}\ \mathrm{E}^0 \rangle\ \boxed{*}\ \mathrm{E}^3 \rangle$
(c) $\mathrm{E}^\perp \Rightarrow \langle \mathrm{E}^2\ \boxed{*}\ \mathrm{E}^3 \rangle \Rightarrow \langle \mathrm{E}^2\ \boxed{*}\ \langle \langle\boxed{\texttt{let}}\ \mathrm{P}^\perp\ \boxed{=}\ \mathrm{E}^\perp\ \boxed{\texttt{in}}\ \mathrm{E}^0 \rangle \rangle$

To account properly for these left- and right-sided concerns, our elaborated grammar $\mathcal{H}$ features nonterminals $\sigma = {}^p s^q$ with separate precedence bounds p and q on either side. Uniquely to this work, we interpret these bounds in a *bidirectional* fashion: either p and q are bounds imposed by the surrounding derivation tree producing σ, limiting the terms σ *produces*; or they are bound-requirements synthesized from the term that *reduces* to σ. Our definition of elaboration in Fig. 5.15 is organized accordingly. A production rule $\sigma \Rightarrow \overline{\chi}$ is introduced either by *tightening* the bounds on σ (`Produce-Tighten`) or by *subsuming* the corresponding reduction $\sigma \Leftarrow \overline{\chi}$ (`Produce-Subsume`), as illustrated for $\mathcal{H}_{\mathsf{HZ}}$ in Fig. 5.12.

$\boxed{\chi \sim x}$ CFG symbol χ is consistent with PBG symbol x

$$t \sim t \qquad {}^{p}s^{q} \sim s$$

$\boxed{\sigma \Rightarrow \overline{\chi}}$ Nonterminal σ produces symbols $\overline{\chi}$

$$\texttt{Produce-Subsume}\ \frac{\sigma \Leftarrow \overline{\chi}}{\sigma \Rightarrow \overline{\chi}} \qquad \texttt{Produce-Tighten}\ \frac{p \preccurlyeq_s m \qquad n \succcurlyeq_s q}{{}^{p}s^{q} \Rightarrow {}^{m}s^{n}}$$

$\boxed{\sigma \Leftarrow \overline{\chi}}$ Symbol sequence $\overline{\chi}$ reduces to nonterminal σ

$$\texttt{PElab-Operand}\ (k \geq 0)\ \frac{[\, x_i \,]_{0 \leq i \leq k} \in [\![\mathcal{G}(s, \square)]\!] \qquad [\, \chi_i \sim x_i \,]_{0 \leq i \leq k} \qquad x_0 \neq s \neq x_k}{{}^{\top}s^{\top} \Leftarrow [\, \chi_i \,]_{0 \leq i \leq k}}$$

$$\texttt{PElab-Infix}\ (k \geq 0)\ \frac{[\, x_i \,]_{0 \leq i \leq k} \in [\![\mathcal{G}(s, m)]\!] \qquad [\, \chi_i \sim x_i \,]_{0 \leq i \leq k} \qquad \chi_0 = {}^{p}s^{n_L} \qquad {}^{n_R}s^{q} = \chi_k \qquad n_L \succ_s m \prec_s n_R}{{}^{\min(p,m)}s^{\min(m,q)} \Leftarrow [\, \chi_i \,]_{0 \leq i \leq k}}$$

$$\texttt{PElab-Prefix}\ (k \geq 1)\ \frac{[\, x_i \,]_{0 \leq i \leq k} \in [\![\mathcal{G}(s, m)]\!] \qquad [\, \chi_i \sim x_i \,]_{0 \leq i \leq k} \qquad x_0 \neq s \qquad {}^{n_R}s^{q} = \chi_k \qquad m \prec_s n_R}{{}^{\top}s^{\min(m,q)} \Leftarrow [\, \chi_i \,]_{0 \leq i \leq k}}$$

$$\texttt{PElab-Postfix}\ (k \geq 1)\ \frac{[\, x_i \,]_{0 \leq i \leq k} \in [\![\mathcal{G}(s, m)]\!] \qquad [\, \chi_i \sim x_i \,]_{0 \leq i \leq k} \qquad \chi_0 = {}^{p}s^{n_L} \qquad s \neq x_k \qquad n_L \succ_s m}{{}^{\min(p,m)}s^{\top} \Leftarrow [\, \chi_i \,]_{0 \leq i \leq k}}$$

Figure 5.15: Bidirectional elaboration of production $\sigma \Rightarrow \overline{\chi}$ and reduction $\sigma \Leftarrow \overline{\chi}$ rules for CFG $\mathcal{H}$ from PBG $\mathcal{G}$

$$\begin{array}{rcl} \text{node } \mathbb{X} & ::= & \tau \mid \mathbb{S} \\ \text{term } \mathbb{R}, \mathbb{S} & ::= & \{\overline{\mathbb{X}}\} \end{array}$$

Figure 5.16: Syntax of terms

$$\begin{array}{rcl} \text{leq } \leq & ::= & \lessdot \mid \doteq \mid <: \mid \odot \\ \text{stack } \mathbb{K} & ::= & \prec \mid \mathbb{K} \leq_{\mathbb{S}?} \tau \end{array} \qquad \begin{array}{rcl} \mathsf{hd}(\prec) & = & \prec \\ \mathsf{hd}(\mathbb{K} \leq_{\mathbb{S}?} \tau) & = & \tau \end{array}$$

Figure 5.17: Syntax of stacks

$\boxed{\chi \Leftarrow \mathbb{X}}$ Node $\mathbb{X}$ reduces to symbol χ

$$\texttt{Reduce-Token}\ \frac{}{\tau \Leftarrow \tau}$$

$$\texttt{Reduce-Term}\ (k \geq 0)\quad \frac{\sigma \Leftarrow [\,\chi_i\,]_{0 \leq i \leq k} \qquad [\,\chi_i \Leftarrow \mathbb{X}_i\,]_{0 \leq i \leq k}}{\sigma \Leftarrow \{[\,\mathbb{X}_i\,]_{0 \leq i \leq k}\}}$$

$\boxed{\chi \Rightarrow \mathbb{X}}$ Symbol χ produces node $\mathbb{X}$

$$\texttt{Produce-Token}\ \frac{}{\tau \Rightarrow \tau}$$

$$\texttt{Produce-Term}\ (k \geq 0)\quad \frac{\sigma \Rightarrow [\,\chi_i\,]_{0 \leq i \leq k} \qquad [\,\chi_i \Rightarrow \mathbb{X}_i\,]_{0 \leq i \leq k}}{\sigma \Rightarrow \{[\,\mathbb{X}_i\,]_{0 \leq i \leq k}\}}$$

Figure 5.18: A node is well-formed if it reduces to or is produced by a symbol.

$\boxed{\mathbb{K}\ \mathsf{wf}}$ Stack $\mathbb{K}$ is well-formed

$$\texttt{WFStack-Nil}\ \frac{}{\prec\ \mathsf{wf}}$$

$$\texttt{WFStack-Cons}\quad \frac{\mathbb{K}\ \mathsf{wf} \qquad \sigma? \Rightarrow \mathbb{S}? \qquad \mathsf{hd}(\mathbb{K}) \leq_{\sigma?} \tau}{\mathbb{K} \leq_{\mathbb{S}?} \tau\ \mathsf{wf}}$$

Figure 5.19: Well-formed stacks

$$\frac{\langle \texttt{let} \rangle\ \mathrm{P}\ \langle = \rangle\ \mathrm{E}\ \langle \texttt{in} \rangle\ \underline{\mathrm{E}} \in [\![\mathcal{G}(\mathrm{E}, 0)]\!] \qquad \langle \texttt{let} \rangle \neq \mathrm{E} \qquad 0 \prec_{\mathrm{E}} 1 \qquad {}^{1}\mathrm{E}^{\underline{\perp}} \sim \underline{\mathrm{E}}}{{}^{\top}\mathrm{E}^{\min(0,\underline{\perp})} \Leftarrow \langle \texttt{let} \rangle\ {}^{\perp}\mathrm{P}^{\perp}\ \langle = \rangle\ {}^{\perp}\mathrm{E}^{\perp}\ \langle \texttt{in} \rangle\ {}^{1}\mathrm{E}^{\underline{\perp}}}$$

Meanwhile, a reduction rule $\sigma \Leftarrow \overline{\chi}$ synthesizes the tightest possible bounds on σ that can accommodate $\overline{\chi}$. These correspond to Aasa's notion of *precedence weights* [3] that aggregate the precedence levels of operators exposed along the left and right spines of a syntax tree. Whether an operator contributes its annotated precedence level to its left and right weights depends on its shape—either operand, prefix, postfix, or infix. For example, in the derivation on the left using rule `PElab-Prefix` for $\mathcal{H}_{\mathsf{HZ}}$, the prefix-shaped $\langle \texttt{let} \rangle$-form synthesizes left weight $\top$ and right weight $\min(0, \underline{\perp})$, where the latter folds in the annotated level 0 into the subweight $\underline{\perp}$ (underlined to distinguish it from other $\perp$ values in the derivation) already computed for the rightmost child ${}^{1}\mathrm{E}^{\underline{\perp}}$. Our bidirectional presentation reorganizes and generalizes Aasa's to multi-sorted grammars of arbitrary mixfix forms, which we discuss further in §2.2.2.

5.3.2 OP Parsing Errors

In this section, we review Floyd's original method for operator-precedence (OP) parsing [29]. To motivate our error-handling generalization in §5.3.3, we consider in particular the different ways an OP parser can fail.

Parsing is the task of organizing token sequences into grammatically well-formed terms. Fig. 5.16 gives the syntax of terms: a term $\mathbb{S} = \{\overline{\mathbb{X}}\}$ demarcates a sequence $\overline{\mathbb{X}}$ of child nodes, each either a token τ or a subterm. We consider $\mathbb{S}$ to be well-formed if it is reducible to or producible from a nonterminal σ, i.e. $\sigma \Leftarrow \mathbb{S}$ or $\sigma \Rightarrow \mathbb{S}$ as defined in Fig. 5.18.

OP parsing is a simple form of shift-reduce parsing: input tokens are ingested one at a time,

left-to-right, and kept organized in a maximally reduced stack $\mathbb{K}$ whose contents form prefixes of terms under construction. Fig. 5.17 gives the syntax of OP parsing stacks: a stack $\mathbb{K}$ is either empty, the start delimiter $\prec$ affixed at its base; or it is nonempty $\mathbb{K} \lesssim_{\mathbb{S}?} \tau$, linking a token τ to the rest of the stack $\mathbb{K}$ with two pieces of information: a comparison operator $\lesssim$ recording how the head of $\mathbb{K}$ precedence-relates to τ, and an optional term $\mathbb{S}?$ recording what was first reduced between them. A stack $\mathbb{K}$ is considered well-formed, as specified in Fig. 5.19, when each of its links $\tau_L \lesssim_{\mathbb{S}?} \tau_R$ reflects a valid precedence relation $\tau_L \lesssim_{\sigma?} \tau_R$ such that $\sigma? \Rrightarrow \mathbb{S}?$. For brevity, we will call optional nonterminals *slots* and optional terms *cells*.

The *height* of a stack is the number of $\lessdot$-operators it contains. We can decompose any stack of height h into a sequence of h height-1 stacks, each of the form $\tau \lessdot_{\mathbb{S}?_0} \tau_0 \left[\doteq_{\mathbb{S}?_i} \tau_i \right]_{0<i\leq k}$ $(k \geq 0)$ We may interpret each such stack as a term under construction $\lessdot_{\mathbb{S}?_0} \tau_0 \left[\doteq_{\mathbb{S}?_i} \tau_i \right]_{0<i\leq k}$ delimited on its left by τ, which is either the start of input $\prec$ or the head of the preceding stack in the decomposition.

Fig. 5.20 shows Floyd's original algorithm, presented here as a push operation $\mathbb{K} \underset{\mathbb{R}?}{\longleftarrow} \tau = \mathbb{K}'$ that pushes the next input token τ onto stack $\mathbb{K}$ over the current reduction-in-progress $\mathbb{R}?$ to yield a new stack $\mathbb{K}'$. Fig. 5.21 and Fig. 5.22 illustrate concrete OP parsing traces for $\mathcal{H}_{\mathsf{HZ}}$ using the precedence table in Fig. 5.14—each colored box applies one of the rules in Fig. 5.20, enumerating within it the satisfied premises, and sends the push-inputs above it to the output stack below it. Every push begins by consulting how the stack head $\mathsf{hd}(\mathbb{K})$ precedence-compares with the pushed token τ to decide whether to `Shift` or `Reduce`. If $\mathsf{hd}(\mathbb{K}) \lesssim \tau$, then the parser shifts τ onto $\mathbb{K}$ and "finalizes" the reduction $\mathbb{R}?$ between them. Else, if $\mathsf{hd}(\mathbb{K}) \gtrdot \tau$, and $\mathbb{K}$ has height $h \geq 1$, then the parser has identified its next *handle* (i.e. reduction target) of the form

$$\mathsf{hd}(\mathbb{K}) \lessdot_{\mathbb{R}?_0} \tau_0 \left[\doteq_{\mathbb{R}?_i} \tau_i \right]_{0<i\leq k} \gtrdot_{\mathbb{R}?_{k+1}} \tau$$

where $\mathsf{hd}(\mathbb{K})$ and τ delimit the handle $\left\{ \mathbb{R}?_0 \left[\tau_i \mathbb{R}?_{i+1} \right]_{0\leq i\leq k} \right\}$ to be reduced and propagated up the stack.

Let us consider the ways this algorithm can fail.

Stuck. Like with most (non-error-handling) methods, a typical OP parser will easily get stuck. This occurs when the stack head and pushed token share no precedence relation, like ⟨2⟩ and ⟨`let`⟩ in Fig. 5.21.

Invalid Reduction. OP parsing is unsound, meaning it can produce grammatically invalid reductions. Recall from § 5.3.1.3 that each precedence comparison $\tau_L \odot_{\rho?} \tau_R$ means there exists a derivable string of the form $\ldots \tau_L \rho? \tau_R \ldots$. Floyd's original definition of precedence

$\boxed{\mathbb{K} \underset{\mathbb{R}?}{\longleftarrow} \tau = \mathbb{K}'}$ Pushing token τ onto stack $\mathbb{K}$ over reduction $\mathbb{R}?$ returns stack $\mathbb{K}'$

$$\textsc{OP-Shift}\ \frac{\mathsf{hd}(\mathbb{K}) \lessdot \tau}{\mathbb{K} \underset{\mathbb{R}?}{\longleftarrow} \tau = \mathbb{K} \lessdot_{\mathbb{R}?} \tau}$$

$$\textsc{OP-Reduce}\ (k \geq 0)\quad \frac{\tau_k \gtrdot \tau \qquad \mathbb{K}_0 \underset{\left\{\mathbb{R}?_0 \left[\, \tau_i \mathbb{R}?_{i+1} \,\right]_{0 \leq i \leq k}\right\}}{\longleftarrow\!\!\longleftarrow} \tau = \mathbb{K}'}{\mathbb{K}_0 \lessdot_{\mathbb{R}?_0} \tau_0 \left[\, \doteq_{\mathbb{R}?_i} \tau_i \,\right]_{0 < i \leq k} \underset{\mathbb{R}?_{k+1}}{\longleftarrow\!\!\longleftarrow} \tau = \mathbb{K}'}$$

Figure 5.20: OP parsing

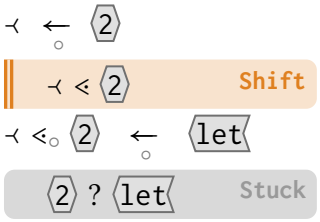


Figure 5.21: An OP parsing trace for $\mathcal{H}_{\mathsf{HZ}}$ (Fig. 5.14) that gets stuck trying to compare neighbors ⟨2⟩ and ⟨let⟩

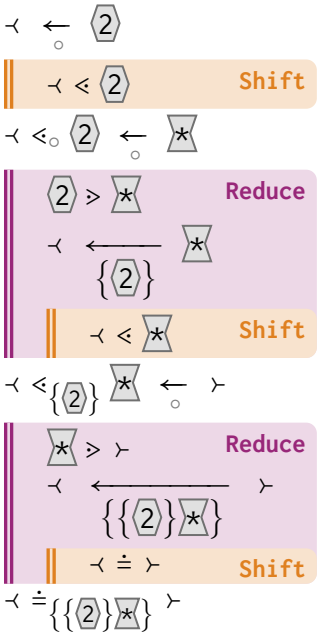


Figure 5.22: A valid OP parsing trace for $\mathcal{H}_{\mathsf{HZ}}$ that returns the invalid term $\{\{⟨2⟩\}⟨*⟩\}$

comparisons omits the index $\rho?$. This "nonterminal blindness" means that an OP parser, given a reduction $\mathbb{R}?$ between delimiters τ_L and τ_R, can determine which parent delimiter(s) to reduce next, but not whether $\mathbb{R}?$ is a valid child of the chosen parent. In the final **Reduce** step in Fig. 5.22, the parser identifies the handle pattern $\prec \lessdot ⟨*⟩ \gtrdot \succ$ and proceeds blindly to reduce $\{\{⟨2⟩\}⟨*⟩\}$ without checking that the initial reduction $\circ$ is a valid right-argument to $⟨*⟩$.

Invalid Prefix. A core tenet of shift-reduce parsers is the *valid prefix property*, which maintains that the parse stack forms the prefix of some grammar-derivable symbol string. This property ensures that the parser is sound, i.e. every parsed term is well-formed.

Ideally prefix-validity would be implied by stack well-formedness (Fig. 5.19), but this is not always the case for an OP parser depending on the grammar. Consider the following small grammar

$$\begin{array}{lcl} \hat{\mathtt{T}} & \to & \mathtt{\$\,x\,\$} \mid \mathtt{t} \\ \mathtt{x} & \to & \mathtt{\{\,T\,\}} \mid \mathtt{x} \end{array}$$

which produces strings like `t`, `$x$`, `${t}$`, `${$x$}$`, etc. Generated from this grammar are the precedence comparisons $\prec \lessdot \$$ and $\$ \doteq \$$, so Floyd's parser would happily ingest the tokens `$x$x$` and organize them into the stack $\prec \lessdot_\circ \$ \doteq_{\bullet\{x\}} \$ \doteq_{\bullet\{x\}} \$$, which is well-formed but prefix-invalid. The issue here is the reuse of `$` as both opener and closer in the rule $\hat{\mathtt{T}} \to \mathtt{\$\,x\,\$}$. Distinguishing between these two uses would require stack-level analyses out of scope of the local pairwise precedence comparisons.

5.3.3 OP Parsing with Error Handling

We now define our error-handling extension of OP parsing that avoids or recovers from the various failure modes seen in the last section. Some of our changes involve requirements (§5.3.3.1) and transformations (§5.3.3.2) of the language grammar; others involve generalizing Floyd's algorithm (§5.3.3.3) to incorporate completion-based repairs and to restore soundness by making use of our nonterminal-enriched precedence comparisons.

5.3.3.1 Molding Tiles

To secure the valid prefix property, we take the blunt approach of requiring every tile $t \in \mathcal{T}$ to appear uniquely in the PBG $\mathcal{G}$:

Assumption 2 (Unique Tiles)**.** *A tile $t \in \mathcal{T}$ is called* unique *if* $(\ldots t \ldots \in [\![\mathcal{G}(s,p)]\!]$ *and* $\ldots t \ldots \in [\![\mathcal{G}(r,q)]\!])$ *imply* $(s = r$ *and* $p = q$ *and* t *appears uniquely in* $\mathcal{G}(s,p))$*. All tiles $t \in \mathcal{T}$ are unique.*

With Assumption 2, we can guarantee for any height-1 precedence chain of the form $\tau \lessdot_{\rho?_0} t_0 \Big[\doteq_{\rho?_i} t_i \Big]_{0<i\leq k}$ that the string $[\, \rho?_i t_i \,]_{0\leq i\leq k}$ forms a prefix of the yield of some nonterminal σ adjacent to τ:

Lemma 1 (Valid Prefixes)**.** *For all terminals τ, tiles $[\, t_i \,]_{0\leq i\leq k}$, and slots $[\, \rho?_i \,]_{0\leq i\leq k+1}$ $(k \geq 0)$ such that $\tau \lessdot_{\rho?_0} t_0 \Big[\doteq_{\rho?_i} t_i \Big]_{0<i\leq k} \lhd \rho?_{k+1}$ there exist nonterminals σ_τ, σ, tiles $[\, t_i \,]_{k<i\leq \ell}$, and slots $[\, \rho?_i \,]_{k+1<i\leq \ell+1}$ $(\ell \geq k)$ such that $\tau \lhd \sigma_\tau \lessdot^* \sigma \Rightarrow \rho?_0 [\, t_i \rho?_{i+1} \,]_{0\leq i\leq \ell}$.*

Assumption 2 would be severely constraining if $\mathcal{G}$ were a grammar of purely textual tokens—for example, we would not be able to reuse parentheses `( )` across different sorts. In this work, we take $\mathcal{G}$ to be a grammar of tiles, which we conceptualized in our Hazel grammar $\mathcal{G}_{\mathsf{HZ}}$ (Fig. 5.11) as being textual tokens paired with "molds", visually distinguished using color and shape. Rather than requiring that the grammar author manually design and distinguish their terminal symbols, however, we can generically convert any ordinary textual grammar $\mathcal{F}$ into a grammar $\mathcal{G}$ of unique tiles, simply by augmenting each terminal symbol in $\mathcal{F}$ with its *one-hole context*, i.e. its mold. We continue this discussion in §5.4, where we describe how `tylr` chooses between multiple possible molds for a textual token.

5.3.3.2 Injecting Grout

When a shift-reduce parser "goes wrong", it is because of an unresolved mismatch between the bottom-up reductions accumulated so far and the remaining top-down expectations of the grammar. Many of these mismatches are inconsistencies of *multiplicity*: in Fig. 5.22, the reduction $\{\{\langle 2 \rangle\}\, \langle * \rangle\}$ in the last `Reduce` step is ill-formed because there is no term where one is expected

$\boxed{\sigma \Leftarrow \overline{\chi}}$ $\overline{\chi}$ reduces to σ in grout-injected $\mathcal{G}$

grout $\gamma ::= \bigcirc \mid \prec \mid \succ \mid \times$
terminal $\tau ::= \ldots \mid \gamma^s$

$$\ldots \qquad \frac{}{{}^{\top}s^{\top} \Leftarrow \bigcirc^s}\ \textsc{GInj-Hole} \qquad \sigma \,!\, s = \begin{cases} \top & \text{if } \sigma \not\sim s \\ \bot & \text{if } \sigma \sim s \end{cases} \qquad \langle {}^{p}r^{q} \rangle_s^m = \begin{cases} {}^{\max(m,p)}r^{\max(q,m)} & \text{if } r = s \\ {}^{p}r^{q} & \text{if } r \neq s \end{cases}$$

$$\textsc{GInj-Operand}\ (k \geq 0)\quad \frac{\left[\, {}^{\bot}s^{\bot} \Rightarrow^* \ldots \rho_i \ldots \,\right]_{0 \leq i \leq k}}{{}^{\top}s^{\top} \Leftarrow \prec^s \langle \rho_0 \rangle_s^0 \left[\, \times^s \langle \rho_i \rangle_s^0 \,\right]_{0<i\leq k} \succ^s}$$

$$\textsc{GInj-Infix}\ (k \geq 0)\quad \frac{{}^{\bot}s^{\bot} \Rightarrow^* \rho_0 \ldots \qquad \left[\, {}^{\bot}s^{\bot} \Rightarrow^* \ldots \rho_i \ldots \,\right]_{0<i<k} \qquad {}^{\bot}s^{\bot} \Rightarrow^* \ldots \rho_k}{{}^{\rho_0 \,!\, s}s^{\rho_k \,!\, s} \Leftarrow \langle \rho_0 \rangle_s^0 \left[\, \times^s \langle \rho_i \rangle_s^0 \,\right]_{0<i\leq k}}$$

$$\textsc{GInj-Prefix}\ (k \geq 0)\quad \frac{\left[\, {}^{\bot}s^{\bot} \Rightarrow^* \ldots \rho_i \ldots \,\right]_{0 \leq i < k} \qquad {}^{\bot}s^{\bot} \Rightarrow^* \ldots \rho_k}{{}^{\top}s^{\rho_k \,!\, s} \Leftarrow \prec^s \langle \rho_0 \rangle_s^0 \left[\, \times^s \langle \rho_i \rangle_s^0 \,\right]_{0<i\leq k}}$$

$$\textsc{GInj-Postfix}\ (k \geq 0)\quad \frac{{}^{\bot}s^{\bot} \Rightarrow^* \rho_k \ldots \qquad \left[\, {}^{\bot}s^{\bot} \Rightarrow^* \ldots \rho_i \ldots \,\right]_{k>i\geq 0}}{{}^{\rho_k \,!\, s}s^{\top} \Leftarrow \left[\, \langle \rho_i \rangle_s^0 \times^s \,\right]_{k \geq i > 0} \langle \rho_0 \rangle_s^0 \succ^s}$$

Figure 5.23: Grout injection extending the definition of terminals τ (Fig. 5.10) and reduction $\sigma \Leftarrow \overline{\chi}$ (Fig. 5.15)

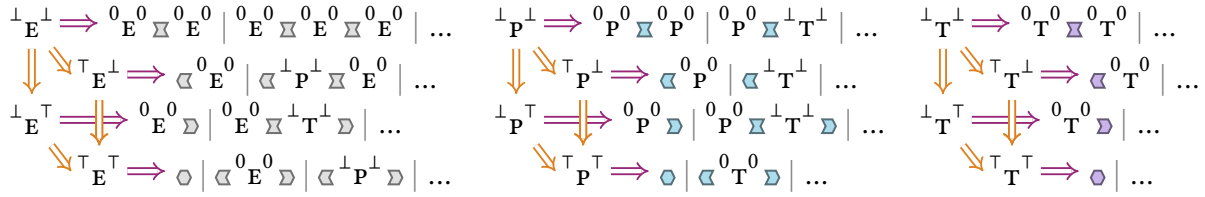


Figure 5.24: Excerpt of the grout rules injected (Fig. 5.23) into $\mathcal{H}_{\mathsf{HZ}}$ (Fig. 5.12). The production rules are arranged and color-coded by whether they emerge from subsuming reduction or by tightening (Fig. 5.15).

as the right multiplicand; in Fig. 5.21, the parser gets stuck on neighbors ⟨2⟩ and ⟨let⟩ because it does not know how to combine these parts of two unrelated terms into one as required. When multiplicities align, there remains further the possibility of *sort* inconsistencies, such as the one in Fig. 5.7 between the ⟨let⟩-delimiter expecting a pattern and the ⟨->⟩-term providing a type.

Fig. 5.23 shows how we materialize these inconsistencies as *grout* forms injected into the elaborated grammar, extending our definition in Fig. 5.15, while Fig. 5.24 shows an excerpt of the grout forms injected into $\mathcal{H}_{\mathsf{HZ}}$. Every sort acquires the form ${}^{\top}s^{\top} \Rightarrow \bigcirc^s$, injected via rule `GInj-Hole`, consisting of a single convex grout terminal $\bigcirc^s$ that stands in for missing terms of sort s.

Grout terminals also come in prefix $\prec$, postfix $\succ$, and infix $\times$ shapes that are used to wrap sort-inconsistent and extraneous terms, injected via the rules `GInj-Operand`, `GInj-Infix`, `GInj-Prefix`, and `GInj-Postfix`. There are four of these rules to enumerate over whether the left and right ends of the form are bookended with a prefix $\prec^s$ or postfix $\succ^s$ grout, respectively. The left (right) bookend is optional when the exposed nonterminal is a leftmost (rightmost) descendant of the unbounded sort ${}^{\bot}s^{\bot}$. For example, Fig. 5.24 includes the grout production ${}^{\top}\mathrm{P}^{\bot} \Rightarrow \prec^{\bot}\mathrm{T}^{\bot}$

because of the P-sorted form ${}^{\bot}\mathrm{P}^{1}$ ⟩:⟨ ${}^{\bot}\mathrm{T}^{\bot}$ in $\mathcal{H}_{\mathsf{HZ}}$ (Fig. 5.12), where ${}^{\bot}\mathrm{T}^{\bot}$ is the rightmost descendant. On the other hand, there is no E-sorted form with ${}^{\bot}\mathrm{P}^{\bot}$ as its leftmost or rightmost descendant, so ${}^{\bot}\mathrm{P}^{\bot}$ can only appear in the E-sorted grout forms that buffer it on both sides (e.g. ◸${}^{\bot}\mathrm{P}^{\bot}$◹).

Grout terminals behave like associative operators of loosest precedence within each sort, where their left and right tip decorations follow the pattern of tiles. More precisely, $\gamma_L^s \doteq \gamma_R^s$ if γ_L is right-concave and γ_R is left-concave, and $\gamma^s \lessdot t$ for any tile t of sort s if γ is right-concave. The nonterminal descendants ρ_i are precedence-bounded in their injected forms $\langle \rho_i \rangle_s^0$, depending on their sort, to prevent conflicting precedence comparisons between grout terminals of the same sort.

5.3.3.3 Parsing with meldr

Fig. 5.27 gives the rules for parsing with meldr, whose notable features we will illustrate through several examples.

Fig. 5.28 illustrates how meldr directly generalizes the standard non-error-handling algorithm (Fig. 5.20). The main difference is the new *fill* operation, defined in Fig. 5.25, invoked in Fig. 5.28 as $\cdot \curvearrowright \circ_{\mathrm{slot}} = \circ_{\mathrm{cell}}$ in **Reduce** and $\{\langle \mathtt{x} \rangle\} \curvearrowright {}^{\bot}\mathrm{P}^{1} = \{\langle \mathtt{x} \rangle\}$ in **Shift**. Filling is responsible for assigning accumulated reductions to grammatically appropriate slots, now exposed as operator indices in the precedence comparisons. In **Reduce**, nothing $\cdot$ is assigned to the unfillable slot $\circ_{\mathrm{slot}}$; in **Shift**, the reduction $\{\langle \mathtt{x} \rangle\}$ is assigned to the fillable slot $\bullet^{\bot}\mathrm{P}^{1}$. In these cases, the input reduction is returned as is because it is consistent with its assigned slot. In other cases, filling may additionally repair the given reduction with additional grout to bridge any multiplicity or sort inconsistencies. Fig. 5.29 shows how pushing ⟩:⟨ against the stack $\prec \lessdot_\circ$ ⟨let⟩ leads to the slot $\bullet^{\bot}\mathrm{P}^{1}$ getting filled instead with convex grout $\{\circlearrowright^{\mathrm{P}}\}$.

Fig. 5.30 shows how meldr generalizes the single-step precedence comparisons of the original algorithm to multi-step precedence *walks*. Where the original method would get stuck trying to push ⟨Num⟩ or $\succ$ against the stack $\prec \lessdot_\circ$ ⟨let⟩ because ⟨let⟩ is precedence-comparable with either, meldr can proceed because it finds extended walks like ⟨let⟩ $\lessdot_\circ$ ◸$^{\mathrm{P}}$ $\lessdot_\circ$ ⟨Num⟩ and ⟨let⟩ $\doteq_{{}^{\bot}\mathrm{P}^{\bot}}$ ⟩=⟨ $\doteq_{{}^{\bot}\mathrm{E}^{\bot}}$ ⟩in⟨ $\gtrdot_{1\,{}^{\bot}\mathrm{E}^{\bot}}$ $\succ$. In both of these cases, the fill operation has multiple ways of assigning the initial reduction $\{\langle \mathtt{x} \rangle\}$ to the traversed slots, as determined by the rule `Fill-Partition`. Whichever walk is chosen and however its slots are filled (§5.4.1), the intermediate terminals and filled slots traversed between the comparands form the completion meldr uses to repair the input.

A subtle but consequential difference between meldr and OP parsing lies in our definition of **Reduce**: meldr does not require that the comparison walk conclude with the pushed terminal τ—any concluding terminal (notated $\square$) is sufficient. This relaxation allows meldr to fall back to **Reduce** when the stack head and pushed terminal are not monotonically precedence-walkable,

$\boxed{\overline{\mathbb{R}} \curvearrowright \overline{\sigma?} = \overline{\mathbb{S}?}}$ Filling slots $\overline{\sigma?}$ with reductions $\overline{\mathbb{R}}$ returns cells $\overline{\mathbb{S}?}$

$$\textsc{Fill-None}\quad \frac{}{\cdot \curvearrowright \circ = \circ} \qquad \textsc{Fill-Default}\quad \frac{\sigma \sim s}{\cdot \curvearrowright \bullet\sigma = \bullet\{\square^s\}}$$

$$\textsc{Fill-Operand}\ (k \geq 0)\quad \frac{\underbrace{\{\square^s\, \mathbb{R}_0\, [\, \square^s\, \mathbb{R}_i\,]_{0<i\leq k}\, \square^s\}}_{\sigma \Rrightarrow \mathbb{S}}}{[\, \mathbb{R}_i\,]_{0\leq i\leq k} \curvearrowright \bullet\sigma = \bullet\mathbb{S}} \qquad \textsc{Fill-Infix}\ (k \geq 0)\quad \frac{\underbrace{\{\mathbb{R}_0\, [\, \square^s\, \mathbb{R}_i\,]_{0<i\leq k}\}}_{\sigma \Rrightarrow \mathbb{S}}}{[\, \mathbb{R}_i\,]_{0\leq i\leq k} \curvearrowright \bullet\sigma = \bullet\mathbb{S}}$$

$$\textsc{Fill-Prefix}\ (k \geq 0)\quad \frac{\underbrace{\{\square^s\, \mathbb{R}_0\, [\, \square^s\, \mathbb{R}_i\,]_{0<i\leq k}\}}_{\sigma \Rrightarrow \mathbb{S}}}{[\, \mathbb{R}_i\,]_{0\leq i\leq k} \curvearrowright \bullet\sigma = \bullet\mathbb{S}} \qquad \textsc{Fill-Postfix}\ (k \geq 0)\quad \frac{\underbrace{\{[\, \mathbb{R}_i\, \square^s\,]_{0\leq i<k}\, \mathbb{R}_k\, \square^s\}}_{\sigma \Rrightarrow \mathbb{S}}}{[\, \mathbb{R}_i\,]_{0\leq i\leq k} \curvearrowright \bullet\sigma = \bullet\mathbb{S}}$$

$$\textsc{Fill-Partition}\ (1 \leq j \leq k)\quad \frac{\overline{\mathbb{R}} = [\, \overline{\mathbb{R}}_i\,]_{1\leq i\leq k} \qquad [\, \overline{\mathbb{R}}_i \curvearrowright \sigma?_i = \mathbb{S}?_i\,]_{1\leq i\leq k}}{\overline{\mathbb{R}} \curvearrowright [\, \sigma?_i\,]_{1\leq i\leq k} = [\, \mathbb{S}?_i\,]_{1\leq i\leq k}}$$

Figure 5.25: Filling slots

$$\begin{aligned} \mathsf{parse}\,(\mathbb{K}, \cdot) &= \mathbb{K} \\ \mathsf{parse}\,(\mathbb{K}, \tau\,\overline{\tau}) &= \mathsf{parse}\left(\mathbb{K} \underset{\cdot}{\leftarrow} \tau, \overline{\tau}\right) \end{aligned}$$

Figure 5.26: Parsing with meldr

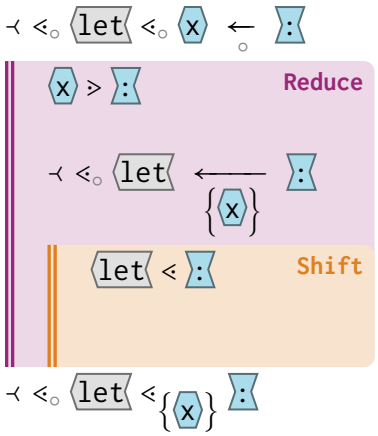
$\boxed{\mathbb{K} \underset{\overline{\mathbb{R}}}{\leftarrow} \tau = \mathbb{K}'}$ Pushing token τ onto stack $\mathbb{K}$ over reductions $\overline{\mathbb{R}}$ returns stack $\mathbb{K}'$

$$\textsc{Shift}\ (k \geq 0)\quad \frac{\mathsf{hd}(\mathbb{K})\left[\, \lessdot_{\rho?_i}\, \tau_i\, \right]_{0\leq i\leq k} = \tau \qquad \overline{\mathbb{R}} \curvearrowright [\, \rho?_i\,]_{0\leq i\leq k} = [\, \mathbb{S}?_i\,]_{0\leq i\leq k}}{\mathbb{K} \underset{\overline{\mathbb{R}}}{\leftarrow} \tau = \mathbb{K}\left[\, \lessdot_{\mathbb{S}?_i}\, \tau_i\, \right]_{0\leq i\leq k}}$$

$$\textsc{Reduce}\ (0 \leq k \leq \ell)\quad \frac{t_k\left[\, \doteq_{\rho?_i}\, t_i\, \right]_{k\leq i\leq \ell} \gtrdot_{\rho?_{\ell+1}} \square \qquad \overline{\mathbb{R}} \curvearrowright [\, \rho?_i\,]_{k<i\leq \ell+1} = [\, \mathbb{R}?_i\,]_{k<i\leq \ell+1} \qquad \mathbb{K}_0 \underset{\{\mathbb{R}?_0[\, t_i\, \mathbb{R}?_{i+1}\,]_{0\leq i\leq \ell}\}}{\leftarrow} \tau = \mathbb{K}}{\mathbb{K}_0 \lessdot_{\mathbb{R}?_0} t_0\left[\, \doteq_{\mathbb{R}?_i}\, t_i\, \right]_{0\leq i\leq k} \underset{\overline{\mathbb{R}}}{\leftarrow} \tau = \mathbb{K}}$$

$$\textsc{Degrout}\ (k \geq 0)\quad \frac{\mathbb{K}_0 \underset{[\, \mathbb{R}?_i\,]_{0\leq i\leq k}\, \overline{\mathbb{R}}}{\leftarrow} \tau = \mathbb{K}}{\mathbb{K}_0 \lessdot_{\mathbb{R}?_0} \gamma_0^s\left[\, \doteq_{\mathbb{R}?_i}\, \gamma_i^s\, \right]_{0<i\leq k} \underset{\overline{\mathbb{R}}}{\leftarrow} \tau = \mathbb{K}}$$

Figure 5.27: Pushing with meldr



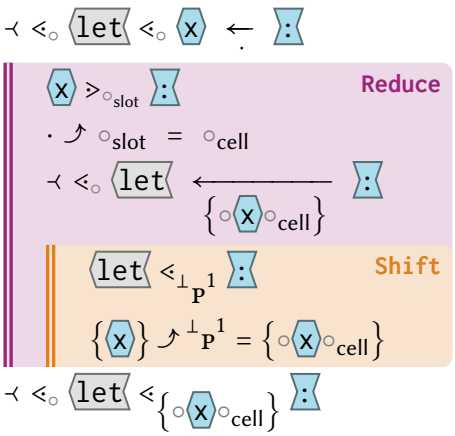


Figure 5.28: Corresponding traces of OP parsing (left) and meldr (right) on the same inputs to highlight their differences

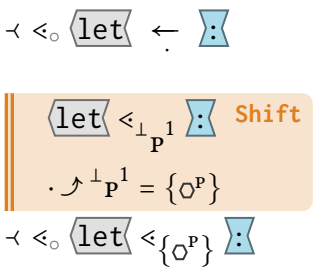


Figure 5.29: meldr filling slot ${}^{\perp}\mathrm{P}^{1}$ with grout form $\{\square^{\mathrm{P}}\}$

completing and reducing the head stack level and deferring the comparison to something further up the stack. An example of this is shown in the first Reduce step in Fig. 5.31b that handles pushing ⟨let⟩ against the stack ≺ ⋖∘ ⟨2⟩. As shown in the example and in our metatheory, this recursive deferral is guaranteed to conclude eventually with the base rule Shift, thanks to the various grout forms that can accommodate both the accumulated reduction and the pushed terminal. This fallback to completion and reduction is a sort of opposite of "panic mode", which is forced instead to drop parts of the stack without the multiplicity-handling guarantees of grout.

5.3.3.4 Sound and Total

Altogether, molded tiles, injected grout, and our filling and walking extensions of OP parsing guarantee that meldr can complete and reduce any sequence of input tiles into a well-formed term.

Lemma 2 (Pushing is Sound and Total)**.** *For all well-formed stacks* $\mathbb{K}$ wf *and tiles* t*, there exists well-formed stack* $\mathbb{K}'$ wf *such that* $\mathbb{K} \leftarrow t = \mathbb{K}'$.

Theorem 2 (Parsing is Sound and Total)**.** *For all well-formed stacks* $\mathbb{K}$ wf *and tile sequences* $\bar{t}$*, there exists well-formed stack* $\prec \doteq_{\mathbb{S}} \succ$ wf *such that* $\mathsf{parse}(\mathbb{K}, \bar{t}\,\succ) = \prec \doteq_{\mathbb{S}} \succ$.

The traces in Fig. 5.31 illustrate this guarantee for the failed examples in Fig. 5.21 and Fig. 5.22.

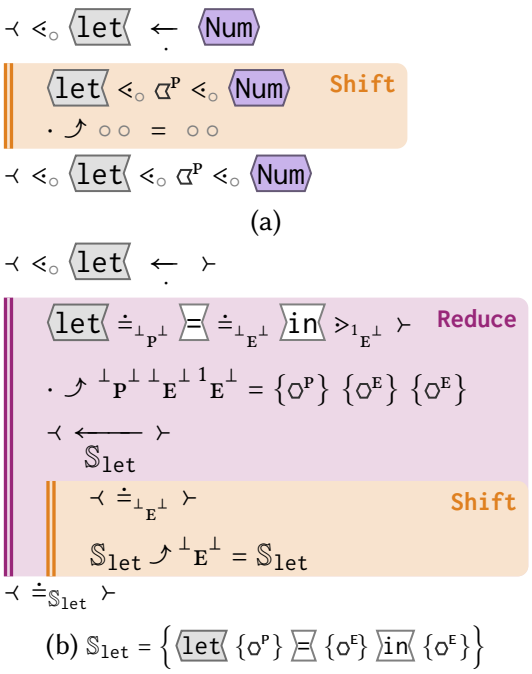
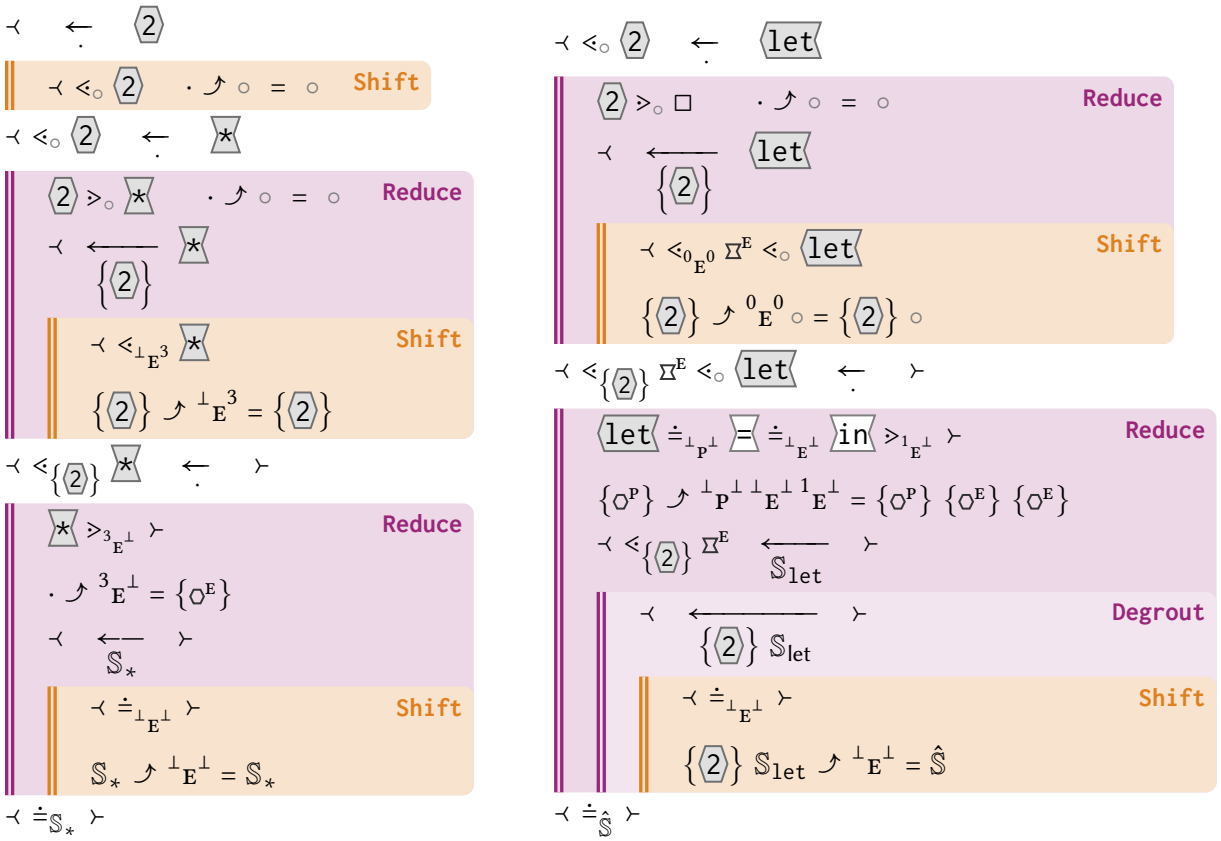


Figure 5.30: meldr traces with multi-step precedence walks

5.4 From meldr to `tall tylr`

meldr formalizes a nondeterministic parser of tile sequences, possibly completing them with some choice of grout and additional tiles, and we showed that the resulting term is grammatical and guaranteed to exist. To turn this into a deterministic parser of textual input, we must answer the following questions. (A) How does the parser "mold" raw text into the tiles to be parsed, in particular when numerous grammatically unique tiles share a common textual form? (B) How does the parser rank and choose among different possible completions?

Moreover, meldr assumes a batch processing context, where the entire input is parsed left-to-right from scratch. Further questions arise when incorporating meldr into an interactive editor



(a) $\mathbb{S}_* = \left\{\left\{\boxed{2}\right\} \boxed{*} \left\{\circ^{\mathrm{E}}\right\}\right\}$

(b) $\mathbb{S}_{\mathsf{let}} = \left\{\boxed{\mathsf{let}} \left\{\circ^{\mathrm{P}}\right\} \boxed{=} \left\{\circ^{\mathrm{E}}\right\} \boxed{\mathsf{in}} \left\{\circ^{\mathrm{E}}\right\}\right\}$, $\hat{\mathbb{S}} = \left\{\left\{\boxed{2}\right\} \varpi^{\mathrm{E}} \mathbb{S}_{\mathsf{let}}\right\}$

Figure 5.31: Complete parsing traces using the rules in Fig. 5.27 to illustrate how meldr (a) avoids producing ill-formed terms like in Fig. 5.22 and (b) avoids getting stuck like in Fig. 5.21

like `tall tylr`. (C) How might existing structures and completions guide or constrain subsequent molding and completion choices? (D) How does the user interact with the system-chosen completions, in particular when it differs from their intent?

This section describes how we addressed these questions in our implementation of `tall tylr`. Subsequently, §5.5 presents the user study we conducted to evaluate these decisions.

5.4.1 Minimizing Obligations

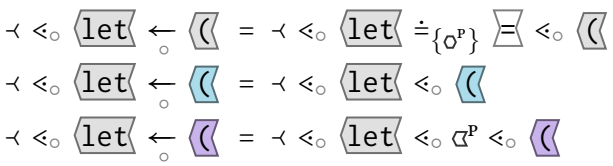
Guiding `tall tylr`'s various decisions is a simple principle: *minimize obligations*. Obligations serve not only to scaffold and complete partial structures, but also as a useful metric for resolving ambiguities. Because meldr is total, we may adopt the simple strategy of trying every choice at each juncture—setting aside efficiency concerns for the moment—and taking the one that inserts the fewest (and removes the most) obligations.

Each type of obligation is weighted differently. Recall from §5.2 that the various forms of obligations can be viewed as indicators of *multiplicity* and *sort* inconsistencies between the top-down expectations of the grammar and the bottom-up reductions of the input:

- Operand grout ○ indicate there is no term where one is expected ($0 = \bullet < 1$).
- Ghosts indicate there is a partial term where one is expected ($0 < \bullet < 1$).
- Prefix ◁ and postfix ▷ grout indicate there is a term as expected ($\bullet = 1$), but of the wrong sort.
- Infix grout ⧈ indicate there are multiple terms where one is expected ($1 < \bullet$).

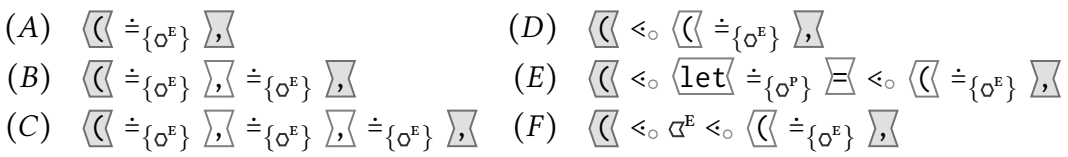
The obligations are listed above in order of increasing weight class. Given two sets of changes in obligations, we compare them lexicographically from highest to lowest weight class. The principle underlying this ordering is *context preservation*: lower-weighted obligations like operand grout and ghosts are introduced to complete a form independent of its context, whereas higher-weighted obligations like pre-, post-, and infix grout appear when the current context cannot accommodate a form and must change.

Molding Tiles When a token is inserted, `tall tylr` looks up which tiles in the grammar share the same textual form (typically only a few) and considers the consequences of parsing each one. For example, when editing Hazel (Fig. 5.11), suppose the token `(` is inserted against the stack $\prec \lessdot_\circ$ `let`. There are three distinct tiles with the same textual label, each of a different sort. Pushing each tile against the stack leads to the following minimal outcomes, where ghosts are indicated with a white background:



The first option introduces an operand hole and a ghost, while the third introduces a prefix grout. The clear winner is the second option, an opening parenthesis of pattern sort, which introduces no new obligations.

Choosing Completions The parsing rules allow for arbitrary walks through the precedence relation graph, with each step from the head of the stack inserting one or more new obligations. For example, the following are all valid precedence walks when applying the `Shift` rule to derive $\prec \lessdot_\circ \langle (\underset{\circ}{\leftarrow} ,$:



`tall tylr` limits the walks considered to those of shortest length found via breadth-first search, ruling out options like (B) and (C). `tall tylr` also filters out walks with strictly $\lessdot$-intermediate tile levels, such as $\langle \texttt{let} \doteq_{\{o^P\}}$ in (E), preferring instead to abstract such possibilities with grout like in (F). The remaining walks are subsequently sorted by height and length to break ties in obligation deltas when filling in any accumulated terms.

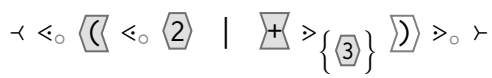
5.4.2 Maintaining Obligations

Total error-correcting parsing lends itself to a continuously structured editing experience. Indeed, our obligation design is inspired directly by numerous structure editor designs [46, 47]. In this setting, questions arise as to how to maintain and remove existing obligations to produce a smooth editing experience, and how to insert new obligations around existing structures.

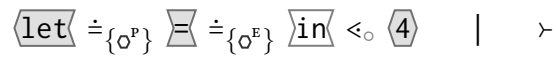
The main concern regards inserting, maintining, and removing ghosts, as the minimal requisite grout needed to complete an edit state is fully determined if all requisite tiles are in place. Ghost maintenance concerns roughly divide into three areas. The first concerns inserting ghost replacements after deleting requisite tiles—in this case, to maximize continuity, `tall tylr` replaces deleted requisite tiles with ghosts in the same position. The second concerns inserting fresh ghosts around existing structures on insertion. As mentioned in §5.2, `tall tylr` uses a simple policy of inserting any pending ghosts at the first newline following the insertion—all other positioning concerns are deferred to obligation minimization and completion choices.

The third area concerns removing existing ghosts when they are no longer needed. `tall tylr` models its edit state as a pair of prefix and suffix stacks, where the suffix is reparsed after each change. There are two cases to consider. The first is when a ghost in the suffix becomes

redundant—for example, when inserting `)` between the stacks



When a ghost is encountered in the suffix, `tall tylr` pushes it onto the prefix stack as if it were a solid tile and removes it if it cannot find an $\doteq$-match. The second case is when a ghost in the prefix becomes redundant—for example, when inserting `in` between the stacks



When `tall tylr` pushes `in` onto the prefix and encounters the ghost `in`, `tall tylr` tries removing it and commits to the removal if the pushed `in` finds an $\doteq$-match. Our current design is limited in that it provides no way to removing ghosts directly, instead requiring the user to insert a solid tile replacement elsewhere, the consequences of which we discuss in more detail in §5.5.

5.4.3 Performance

The focus of this paper is on the conceptual, theoretical, and interaction design of `tall tylr`. We did little to optimize its performance beyond what was needed for responsiveness on relatively small programs (less than 100 lines) and make no strong claims, though we report basic performance numbers in the supplemental appendix for the sake of completeness (Appendix B). There are high-level reasons to believe that this approach would scale performantly. Standard OP parsing scales linearly with the input and moreover enjoys the property of local parsability which greatly simplifies incrementalization and parallelization [7, 8]. Meanwhile, prior work on enumerating local repairs [16, 25] suggests this can be done efficiently. We leave detailed optimizations along these lines to future work.

5.5 User Study

Prior work on error-handling parsing does not explicitly consider user interfaces for representing and interacting with parse errors. In `tall tylr`, we explore a novel UI that materializes obligation-based repairs as inline completions. This requires making choices about where to insert obligations in situations underdetermined in our formal model. Providing a good user experience thus requires choosing heuristics which adequately anticipate user intent across real-world coding tasks, as well as providing affordances to correct obligation placement in cases where these heuristics fail.

We took a maximally structured approach, inserting or removing obligations on every code edit so that the edit state remains structured at all times. While this strategy is desirable in that it allows the possibility of continuous language server feedback, it is relatively aggressive, raising questions about the impact of frequent insertion and removal of elements within the text flow.

We considered the following questions:

Q1 Do users generally find `tall tylr` usable and useful across a range of naturalistic code insertion and modification tasks?

Q2 During which kinds of editing operations do users find specific `tall tylr` mechanisms useful, confusing, or cumbersome?

5.5.1 Study Design

We ran a remote user study, recording participants' screens as they performed nine code transcription and modification tasks. Each sixty minute session began with a series of pre-recorded videos outlining the motivation for `tall tylr` and its essential editor mechanisms. To reduce jargon, we referred to syntactic obligations as 'placeholders' in the study materials.

After the introduction, users performed a practice task to familiarize themselves with the study setup. For each task, they were asked to read and internalize their goal, ask any clarifying questions, and then proceed, pausing after each task to relay any reflections, possibly replaying their actions. At the end of the study, participants were sent a link to an exit survey.

We piloted a shorter version of this study with an earlier prototype; we have included quotes from one previous participant (labeled **P0**) in §5.5.2.

5.5.1.1 Participants

We recruited participants with self-reported experience in expression-based languages by posting on Bluesky, Mastodon, and X offering compensation of $25 USD for a 1-hour session. Our study had 9 participants (8 male, 1 non-binary); ages 19-38 (μ = 28); 5-25 years of programming experience (μ = 13), and 1-15 years of functional programming experience (μ = 6).

5.5.1.2 Tasks

We chose nine code editing tasks (Table 5.1) intended to reflect real-world use patterns, six of which are adapted from a previous study [47]. As well as simple entry and spot-editing tasks, we included more complex goals most economically accomplished by multiple edits which temporarily break term structure; an example is shown in Fig. 5.32. Since the language syntax is new to study participants, we asked them to carefully read the desired end state, and to ask the study administrator any questions about the semantics of the requested transformation.

Table 5.1: Study tasks including line count change between initial and target states

Task	Type	Description	Lines
1	Transcription	Linear entry of data pipeline	+5
2	Modification	Rearrange the elements of a data pipeline	+2 -2
3	Transcription	Linear entry of geometry processing function	+5
4	Modification	Extract helper function	+6 -3
5	Transcription	Linear entry of graphics function definition	+7
6	Modification	Refactor a function to remove redundancy	+7 -7
7	Modification	Add a sum type and add branching to linear code	+9 -3
8	Modification	Uncurry function definition and type annotation	+2 -2
9	Modification	Fuse a series of transformations	+4 -4

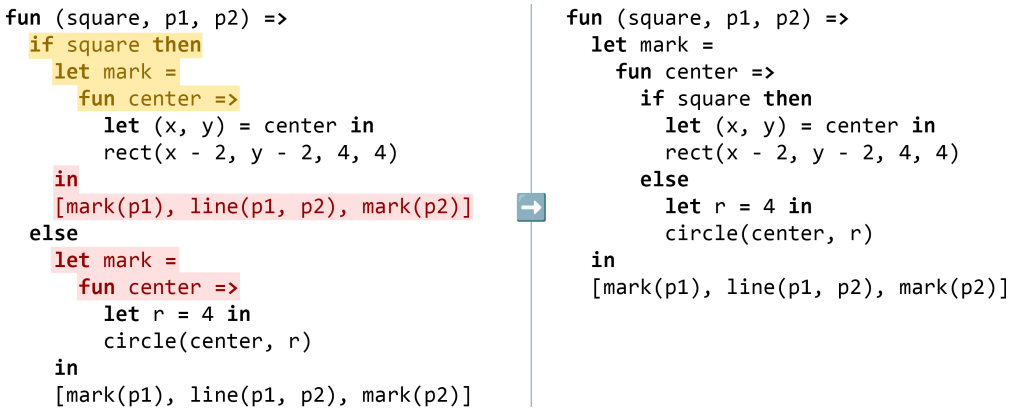


Figure 5.32: Task 6 asked participants to refactor a function from the start state on the left to the target state on the right. Highlighting is added here for readability and was not present in the study.

After each task, participants were asked to reflect on any unexpected or interesting behaviors they encountered. Since we knew participants would approach tasks via different editing strategies, for some tasks we provided a follow-up reflection slide illustrating a specific edit and ensuing placeholder insertion in order to more directly solicit opinions on particular heuristics.

5.5.2 Results

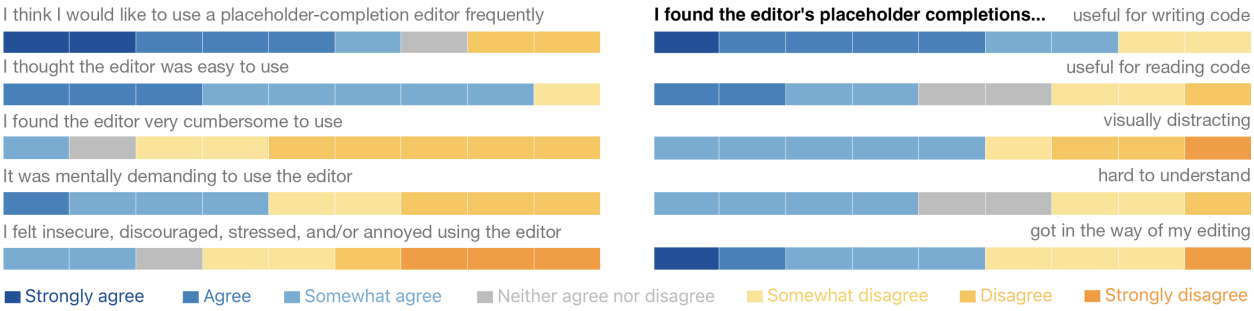


Figure 5.33: Participant opinions on `tall tylr`'s general usability (left) and reactions to placeholders (right)

Participant assessments of overall usability are summarized in Fig. 5.33 , with eight of nine participants at least somewhat agreeing that `tall tylr` was easy to use. However four participants found the editor at least somewhat mentally demanding, with two experiencing stress or annoyance. This may have been impacted by bugs in the prototype. Promisingly, six participants reported desire to frequently use an editor supporting placeholder completions.

Of those who found `tall tylr` easy to use, **P8** said "the typing experience felt premium, bespoke, closer to video game than text editor". With respect to obligations, **P0** remarked "I don't think I expect an editor to exactly pinpoint what fix I need to make. How would it know what I intended? But I like that this allows me to instantly see what it is that you're assuming I meant."

Seven participants at least somewhat agreed obligations helped while writing code. Participants found placeholders particularly helpful during left-to-right entry, with **P8** saying "I always would like placeholder completions until I have a complete expression!". Some attributed this to lowered mental load - **P0** liked that they could "turn my brain off a bit while typing them out". **P2** appreciated that they "just have to remember how to write the first token in a term" due to ghost insertion.

However, five participants felt obligations at least somewhat got in the way while modifying code, and half of participants found placeholders at least somewhat visually distracting and hard to understand. **P9** said that "When I've created an invalid state [during refactoring], the placeholders often didn't feel helpful". **P7** agreed, saying "it seemed to just break and also just jumble up the screen which is when I probably would've preferred a normal editor with red text".

We identified a number of specific scenarios where placeholders proved problematic. Three are

included below, and others (along with more participant reactions) are located in the appendix.

Failed attempts to bust ghosts directly. Although our intended workflow to address ghosts in unwanted positions is for the user to insert the delimiter where they wanted, leaving `tall tylr` to clean up the misplaced ghost, many participants found themselves wanting to interact with ghosts more directly. At least five participants attempted to directly delete ghosts in one or more tasks, despite a caution against this in the introductory video. **P4** felt "the placeholder completions felt like they were there to help the computer, not me", saying that "where I was unable to delete the ghosts and grout, it took a while to figure out how to get rid of them." This was particularly felt when the obligations were inserted in the middle of a complex edit, with **P5** saying "the editor sometimes added a lot of holes while I was in the middle of editing an expression, which I instinctively tried to delete." This issue was exacerbated by a bug in the `tall tylr` prototype that sometimes prevented ghost cleanup in the presence of nested ghost delimiters.

Ghosts are sometimes too tied to the place where they were deleted. Although participants generally liked that ghost delimiters remembered their original positions, this did lead to some confusing in-between ("tween") states. Fig. 5.34 shows a scenario encountered by three participants during Task 8. Here users found the tween state distracting, sometimes attempting unsuccessfully to delete the obligations directly, although all eventually moved on to complete the task successfully.

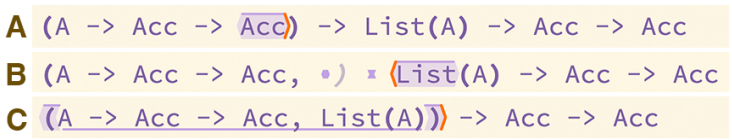


Figure 5.34: During Task 8, participants must modify type annotation (A) to uncurried form. If this is approached in a left-to-right fashion, the user will insert a comma (creating an operand obligation), delete the parenthesis (leaving a ghost), and delete the type arrow (creating a infix obligation) as shown in (B). If the ghost parenthesis did not retain its location, the grout could be combined and cleaned up. This cleanup only occurs when the user re-inserts the closing parenthesis (C).

Uncertainty around triggering token remolding. In `tall tylr` users must press space after entering a leading delimiter like `let` before the associated grout and trailing delimiter ghosts are inserted. This special treatment of space is primarily to permit entry of tokens beginning with `let`, and secondarily to mitigate jarring changes by limiting them to occur only when certain 'action keys' are pressed. In our study this behavior was unproblematic when writing code, but for editing it caused issues, particularly during typo correction. For example, during Task 1 participant **P3** mistyped an operator requiring space and continued on to the end of the line. They later went back to correct it, but since there was already a space afterwards, they didn't bother

to press space after the correction, resulting in remaining infix obligation and the operator left unmolded. Similar issues confusion for at least 3 participants, including **P7** who noted that "space has a learning curve".

5.5.3 Threats to Validity

Our participants are few and drawn from social media networks already self-selected for affinity towards novel programming tools and concepts.

Our study is a synthetic representation of real coding tasks in the sense that participants' attention is artificially divided. Where programmers might otherwise be focused on writing, they now must go back and forth between the slides and the editor. This might make `tall tylr` look both worse and better, in that participants cannot devote their full attention to editor mechanics, but also may avoid being distracted by confusing obligations during awkward tween states.

For the purpose of reducing jargon, we referred to syntactic obligations as 'placeholders' in the study materials, a choice which may have backfired as some participants seemed to expect that these placeholders should be less insistent and easier to dismiss.

There are also a number of factors which complicate clearly ascribing participant difficulties to `tall tylr` mechanics, including: (1) unfamiliar syntax leading to task confusion and higher rates of typos; (2) bugs in the editor interfering with participants' accurately internalizing editor mechanics; (3) learning curve and difficulty internalizing novel concepts within 60 minutes.

5.6 Conclusion

This chapter presented `tall tylr`, a tile-based parser and editor generator that handles errors by completing its input with syntactic obligations. We developed these ideas precisely in our parsing calculus meldr, which extends OP parsing with error handling and guarantees a well-formed result on all inputs—along the way, it offers a new unified account of operator precedence. Key components of meldr's assured totality include relaxing grammaticality with grout, used to buffer inconsistencies of multiplicity and sort, and generalizing the single-step comparisons of OP parsing to multi-step walks that serve as completion-repairs. We proposed the principle of minimizing obligations that governs how `tall tylr` discharges the various choices required for parsing and handling errors. Our user study suggested that syntactic obligations generated this way have both demand and promise, but more design work is needed to give the programmer more control over their placement and removal, especially when modifying existing code. Altogether, this work opens up a significant new design space and we look forward to future design experiments driven by the core ideas introduced in this chapter.

CHAPTER 6

Concluding Remarks

This dissertation presented the `tylr` series of design experiments in increasingly text-like structure editing. Where prior art has focused on streamlining left-to-right transcription, this work emphasized the ergonomics of modifying existing code. A key unifying idea across these designs was the use of *obligations* to buffer structural inconsistencies and materialize to the programmer what remains to reach completion.

A persistent design challenge lay in the interaction mechanics for delimiter obligations, which saw the biggest changes across the `tylr` series. Our user study of `tall tylr` and its use of ghosts suggested some promise but also that further design work remains, both in unifying the interaction mechanics for different types of obligations and providing the user better ways of rejecting default completions. If surfacing obligations continues to pose problems, then we may consider hybrid approaches that continue to use obligations internally if not always externally.

These design experiments converged on a novel method of parsing and error handling. Starting with the perspective of structure editing led us naturally to operator-precedence parsing, given its unique token-centric approach to parsing. OP parsing proved a useful pivot point by which to decompose parsing into top-down molding of tokens to tiles, and bottom-up melding of tiles to terms. The melder encapsulates the basic concerns of reduction and is an unchanging kernel underlying many different possible molding strategies. In our design of `tall tylr`, we propose a simple method based on minimizing obligations. This seemed to work well for Hazel, but it is unclear how well it would work in general. It would be useful in future work to show parsing and completion behavioral guarantees for various molding strategies.

Our formal development of meldr unified and generalized two prior works on operator precedence: Aasa's semantics for precedence-annotated grammars and Floyd's operator-precedence parsing for unannotated grammars. We contribute in particular a novel elaboration from annotated to unannotated grammars that addresses limitations in expressivity or complexity of previous approaches. Error-handling by obligation-based repair emerges as a natural extension of Floyd's original method, where completions arise as the intermediates components of precedence walks between compared tokens.

This work has several directions for future work, including addressing remaining design and theoretical limitations of tile-based editing and parsing, as well as capitalizing on the tile-based approach to develop new features for program editing and analysis.

Rejecting Obligations. In our user study of `tall tylr` (§5.5), participants demanded more direct ways of rejecting obligation-based completions. This challenges our structure-editor-born design assumption thus far that obligations are always materialized and indeed obligatory to discharge. It may be possible to uphold this assumption while meeting the demand for rejection affordances by regenerating obligations elsewhere when they are deleted by the programmer—e.g. in Fig. 5.34(B), when deleting the ghost closing parenthesis, it is "scooched" down to the next viable position to its right (similar in effect to slurping in Paredit [13]).

Otherwise, if continuous materialization proves infeasible, it may still be useful to materialize obligations on demand. Live programming environments often incorporate an execute command that the programmer can invoke to update their live results after some set of modifications—examples include tactic execution in proof assistants like Rocq and cell re-execution in notebooks like Jupyter. This would be an appropriate moment for obligation materialization, such that the programmer is guaranteed a live result while, in the case of a source error, they are shown how the system chose to repair it. In this setting, obligations could be removed manually by the programmer, or automatically cleaned up when appropriately discharged, but would only be generated anew when explicitly requested.

Performance. There remain basic engineering challenges to scaling the performance of tile-based parsing to larger programs. meldr has many points of nondeterminism, including choosing molds, choosing completions, and choosing fills. `tall tylr` discharges these decisions simply by trying them all at each juncture and choosing the result that minimizes obligations. As a result, `tall tylr` can only handle programs of similar magnitude to our study tasks before responsiveness noticeably degrades. I suspect that many of these decisions can be precomputed at parser generation time.

Eliminating Precedence Conflicts. The restricted grammar expressivity of OP parsing stems from the requirement that the derived precedence comparisons be conflict-free: for any pair of terminals τ_L, τ_R, there exists at most one operator $\odot \in \{\lessdot, \doteq, \gtrdot\}$ such that $\tau_L \odot \tau_R$. This allows an OP parser to proceed deterministically. A current gap in our theory of precedence elaboration (§5.3.1) is its lack of guarantees for conflict-freedom.

I do not think the goal should be to eliminate all precedence conflicts. Suppose we extended

our Hazel PBG $\mathcal{G}_{\text{HZ}}$ (Fig. 5.11) with `if`-expressions with optional `else`-branches:

$$\mathcal{G}_{\text{HZ}}(\text{E}, 0) \;:=\; \mathcal{G}_{\text{HZ}}(\text{E}, 0) \mid \langle \texttt{if} \rangle \cdot \text{E} \cdot \langle \texttt{then} \rangle \cdot \text{E} \cdot (\epsilon \mid \langle \texttt{else} \rangle \cdot \text{E})$$

Elaborating this grammar and generating its precedence comparisons would lead to the conflict $\langle \texttt{then} \rangle \doteq \langle \texttt{else} \rangle$ and $\langle \texttt{then} \rangle \gtrdot \langle \texttt{else} \rangle$—that is, when $\langle \texttt{then} \rangle$ and $\langle \texttt{else} \rangle$ are reduced neighbors, it is not clear whether the two should match or if, possibly, the $\langle \texttt{else} \rangle$ should match with a different $\langle \texttt{then} \rangle$ further to the left. This ambiguity, known as the dangling else problem [2], is commonly resolved by preferring the $\doteq$-comparison—that is, $\langle \texttt{else} \rangle$ always matches with the nearest $\langle \texttt{then} \rangle$ to its left. This default disambiguation is easy to communicate to the programmer, an essential reason being that a conflict involving an $\doteq$-comparison necessarily involves terminals of a single form (i.e. they appear in the same regex $\mathcal{G}(s, p)$ for some sort s and precedence p).

The problematic conflicts are $\lessdot/\gtrdot$ conflicts. Such conflicts typically involve terminals of different forms, in which case there is rarely a reasonable default disambiguation, reasonable meaning easily explained to and remembered by the programmer. I conjecture that our precedence elaboration already rules out $\lessdot/\gtrdot$ conflicts between tiles of the same sort. The possibility for conflict remains between tiles of different sorts. Suppose we extended $\mathcal{G}_{\text{HZ}}$ with another pattern-matching form along with `when`-clauses in patterns, for some suitable precedence levels m, n:

$$\mathcal{G}_{\text{HZ}}(\text{E}, m) \;:=\; \mathcal{G}_{\text{HZ}}(\text{E}, m) \mid \text{P} \langle \texttt{:=} \rangle \text{E} \langle \texttt{;} \rangle \text{E}$$
$$\mathcal{G}_{\text{HZ}}(\text{P}, n) \;:=\; \mathcal{G}_{\text{HZ}}(\text{P}, n) \mid \text{P} \langle \texttt{when} \rangle \text{E}$$

Elaborating and generating precedence comparisons leads to the conflict $\langle \texttt{when} \rangle \lessdot \langle \texttt{:=} \rangle$ and $\langle \texttt{when} \rangle \gtrdot \langle \texttt{:=} \rangle$. I conjecture that precedence elaboration rules out all such $\lessdot/\gtrdot$ conflicts, including between tiles of different sorts, provided the input PBG $\mathcal{G}$ additionally satisfies the following assumption:

Assumption 3. *There exists no pair of distinct sorts $r, s \in \mathcal{S}$ and precedence levels $p, q \in \mathcal{P}$ such that $s\ldots \in [\![\mathcal{G}(r, p)]\!]$ and $\ldots r \in [\![\mathcal{G}(s, q)]\!]$.*

Molding and Completion Strategies. `tall tylr` uses obligation minimization to determine how to mold tokens and choose completions (§5.4.1). Other measures and analyses could be used also to guide these decisions. Completion choice would likely benefit from additional input from type-checking, language models, explicit user preferences, etc.

`tall tylr` employs what we might call an "LL" molding strategy: going left-to-right, it assigns a single mold to each encountered token and never backtracks on an assignment. For this strategy to work well (i.e. avoid ambiguous moldings that could later lead to excessive obligations), we

must similarly constrain the class of acceptable grammars. LL grammars must be left-factored (no two production rules have yields with a common prefix) and avoid left recursion (a nonterminal cannot produce a yield that starts with itself). PBGs and precedence elaboration lift the restriction on left recursion, but the need for left-factoring remains. It would be useful to formalize these constraints and guarantees.

Another direction for future inquiry would be to consider "LR" molding strategies. Where LL parsers decide which form a token will reduce to as soon as it is encountered, LR parsers delay this decision until they see the form in its entirety. An LR molding strategy would likely require assigning multiple molds to a token and filtering prior assignments as additional context is ingested.

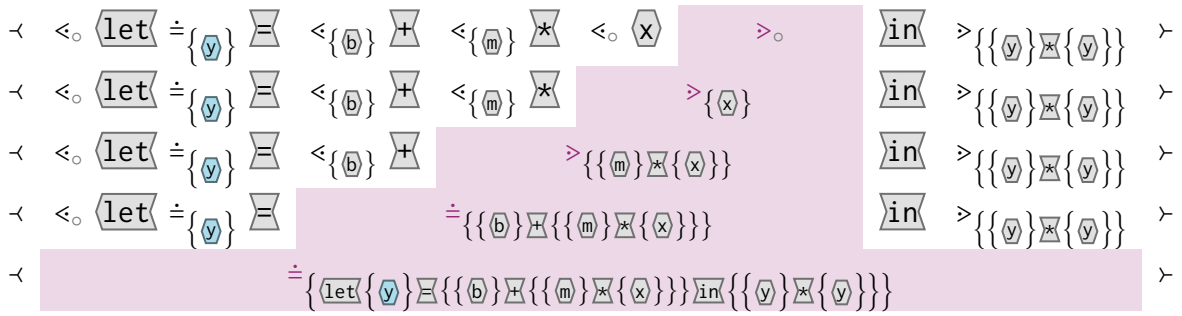
Diekmann [23] considered the problem of composing arbitrary languages together into a single editor, which in general introduces parsing ambiguities when the composed languages feature overlapping syntactic forms. Their proposed system Eco uses *language boxes* to delineate transitions between different languages and resolve these ambiguities. These language boxes are a meta-textual feature that, on their own, the programmer must insert and remove manually by invoking editor commands. Recognizing that this can interrupt the programmer's text-editing flow, Diekmann and Tratt [24] proposed an algorithm for automatically inserting and removing language boxes as the programmer types. Such methods may also be useful in the tile-based setting for resolving molding ambiguities.

Modeling Editors and Incremental Parsing. meldr describes left-to-right batch parsing, but extends trivially to modeling structured edit states and their navigation in `tall tylr`. `tall tylr` models its edit state as a pair of inward-facing stacks, the point between them representing the cursor. For example, using `|` to signify the cursor, `tall tylr` would parse and model the Hazel edit state `let y = b + m * x | in y * y` as the pair of prefix and suffix stacks

(6.1)

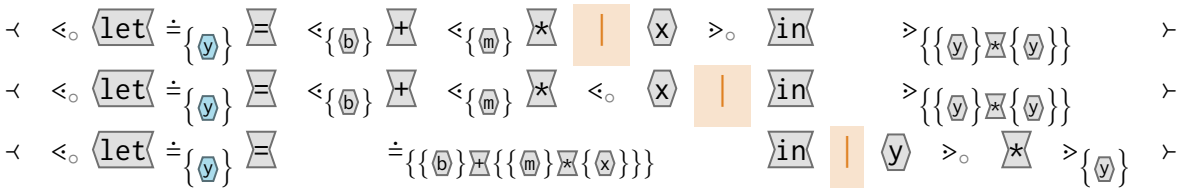
where the suffix stack uses ⩾-relations instead of ⩽-relations.

Note that this representation does not explicitly record the full term structure, instead deferring the hierarchical relationships that may change on subsequent modification. To recover those deferred relationships, we can simply *zip* (à la zippers [38]) the stacks together into a reduced term, using precedence comparisons between the stack heads (the zipper "teeth") to determine which to reduce next, as shown in the following trace.



Further note that there is no additional parser state that needs to be persisted and interleaved within this edit state model, thanks to the symbol-based organization of OP parsing. The predominant item-based approach to parsing (see §2.2.1) means that incremental parsers must additionally persist (or regenerate as needed [65]) parser states representing the points *in between* the symbols on the stack. Not only does the symbol-based approach of OP parsing have the design advantage that tokens have more visual surface area to decorate, thereby making it easier to communicate parser state to the programmer, it also means that our models for edit states and incremental parsers directly align without additional organizational effort. This is a simplifying win for the design, theory, and implementation of incremental parsing.

Meanwhile, the bounded context property of OP parsing makes it trivial to define linear navigation and selection of the edit state. Movement is simply defined as pulling off the head token of the stack in the direction of movement and pushing it onto the opposite stack. Below are the results of moving the cursor left and right around the edit state in (6.1), repeated here in the middle row.

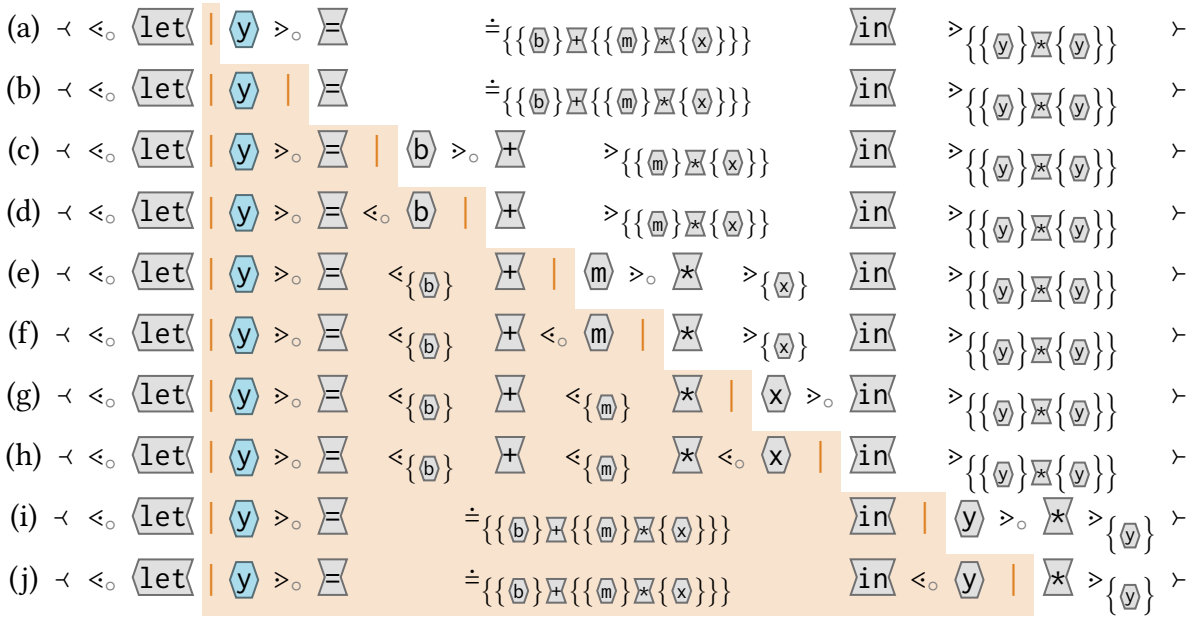


Note that the terms $\{\{y\} * \{y\}\}$ and $\{\{b\} + \{\{m\} * \{x\}\}\}$ are unrolled into cursor-facing stacks once exposed.[1] We may read this sequence of edit states top-to-bottom to see the cursor moving right or bottom-to-top to see the cursor moving left. In the latter case, when we pull `in` from the prefix stack and push it onto the suffix stack, we can perform this operation independently from the remaining prefix tail, thanks to the bounded context property. This is not the case for incremental forms of unbounded-prefix-dependent parsing methods like LL and LR,

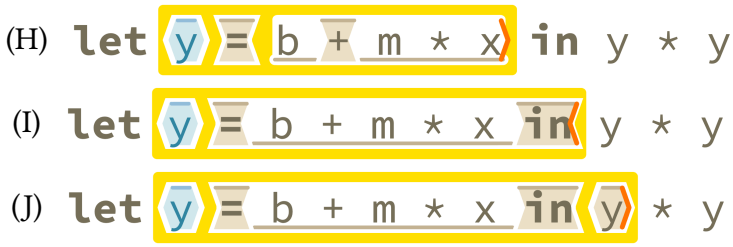
[1] These unrollings correspond to the left and right breakdown operations employed by Wagner [64] and Diekmann [23], here performed during movement rather than there during incremental reparsing after a modification.

meaning editors based on these methods must define separate machinery for linear navigation. The ability to directly repurpose incremental parsing operations this way further underscores the advantage of organizing parsing around locally parsable tiles.

Dynamic Code Folding. Selection is defined similarly simply. A selected range is modeled as a *double-ended stack* onto and from which we may push and pull tiles just as we did for movement on the prefix and suffix stacks above. The following edit states depict various selected ranges that start with the pattern tile ⟨y⟩ on the left.



The same selections are rendered in `tall tylr` as follows:



`tall tylr` renders each stack level as a contiguous "cell" in the selected "honeycomb", e.g. $<_{\{\text{b}\}}$ + in (e) as b + in (E). `tall tylr` additionally incorporates the delimiters surrounding the selection to merge cells that form a complete term, e.g. the multi-level stack $<_{\{\text{b}\}}$ + $<_{\{\text{m}\}}$ * $<_{\circ}$ x in (h) rendered as the single merged cell b + m * x in (H).

The significance of this range selection model is: (1) it is maximally structured (maximal meaning that additional reductions, i.e. cell merging, would require additional unselected tokens), and (2) it is independent of its context. Editors based on unbounded-context parsing do not have straightforward ways of achieving (2). One way to model range selections in this setting would be to pair the overall program tree with the tree paths leading to the cursor endpoints of the selection, but this is an overparametrized definition, since the same range selection can be represented in an infinite number of ways by varying its prefix and suffix in the overall tree. Meanwhile, in the double-ended stack model, every range selection has a single canonical and context-independent representation.

I am interested in exploiting this structured range model in the design of *ubiquitous* and *dynamic* code folding interfaces. Code folding lets the programmer collapse complete delimited structures, such that they can achieve a bird's-eye view of the surrounding structure. Contemporary interfaces require manually toggling the collapsed/expanded state of each structure and often only surfaced for specific sets of delimiters (e.g. matching curly braces) in specific settings like the editor pane containing an entire module. I avoid using them—despite often wanting such abstracted views to minimize the constant scrolling and context-switching of large-scale programming—because I find them unreliable and it too tedious to manage the many toggle states.

The ability to structure any range selection means that we may start to incorporate code folding into more diverse settings. Imagine an interface of language model completions with progressive disclosure of large or deeply nested completions. Imagine a selection interface that dynamically folds its contents and/or the complete structural units nearby, such that large structures could be easily selected in place without the context-switch required in contemporary editors to navigate to the far-off endpoint of a large, purely textual selection. As programmers, we must interact with ever growing syntactic structures and artifacts, an accelerating issue in the era of generative artificial intelligence. It would behoove us to develop more ubiquitous and fluid interfaces for navigating these large structures at varying levels of abstraction.

BIBLIOGRAPHY

[1] 2018. Tree-Sitter. https://tree-sitter.github.io/. (Accessed: 2025-03-26).

[2] 2025. Dangling Else. *Wikipedia* (June 2025).

[3] Annika Aasa. 1995. Precedences in Specifications and Implementations of Programming Languages. *Theoretical Computer Science* 142, 1 (May 1995), 3–26. doi:10.1016/0304-3975(95)90680-J

[4] A. V. Aho, S. C. Johnson, and J. D. Ullman. 1975. Deterministic Parsing of Ambiguous Grammars. *Commun. ACM* 18, 8 (Aug. 1975), 441–452. doi:10.1145/360933.360969

[5] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman (Eds.). 2007. *Compilers: Principles, Techniques, & Tools* (2. ed., pearson internat. ed ed.). Pearson Addison-Wesley, Boston Munich.

[6] R. Bahlke and G. Snelting. 1992. Design and Structure of a Semantics-Based Programming Environment. *International Journal of Man-Machine Studies* 37, 4 (Oct. 1992), 467–479. doi:10.1016/0020-7373(92)90005-6

[7] Alessandro Barenghi, Stefano Crespi Reghizzi, Dino Mandrioli, Federica Panella, and Matteo Pradella. 2015. Parallel Parsing Made Practical. *Science of Computer Programming* 112 (Nov. 2015), 195–226. doi:10.1016/j.scico.2015.09.002

[8] Alessandro Barenghi, Stefano Crespi Reghizzi, Dino Mandrioli, and Matteo Pradella. 2013. Parallel Parsing of Operator Precedence Grammars. *Inform. Process. Lett.* 113, 7 (April 2013), 245–249. doi:10.1016/j.ipl.2013.01.008

[9] Tom Beckmann, Patrick Rein, Toni Mattis, and Robert Hirschfeld. 2022. Partial Parsing for Structured Editors. In *Proceedings of the 15th ACM SIGPLAN International Conference on Software Language Engineering*. ACM, Auckland New Zealand, 110–120. doi:10.1145/3567512.3567522

[10] Tom Beckmann, Patrick Rein, Stefan Ramson, Joana Bergsiek, and Robert Hirschfeld. 2023. Structured Editing for All: Deriving Usable Structured Editors from Grammars. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*. ACM, Hamburg Germany, 1–16. doi:10.1145/3544548.3580785

[11] 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 (FSE 2016). Association for Computing Machinery, New York, NY, USA, 763–774. doi:10.1145/2950290.2950315

[12] 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. doi:10.1109/BLOCKS.2015.7369003

[13] Taylor R Campbell. 2022. Paredit — Parenthetical Editing in Emacs. https://paredit.org/. (Accessed: 2025-09-15).

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

[15] Sam Cohen and Ravi Chugh. 2025. Code Style Sheets: CSS for Code. arXiv:2502.09386 [cs] doi:10.1145/3720421

[16] Breandan Considine, Jin Guo, and Xujie Si. [n. d.]. Syntax Repair as Idempotent Tensor Completion. ([n. d.]).

[17] G. V. Cormack. 1989. An LR Substring Parser for Noncorrecting Syntax Error Recovery. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation*. ACM, Portland Oregon USA, 161–169. doi:10.1145/73141.74832

[18] Geoff Cumming. 2014. The New Statistics: Why and How. *Psychological Science* 25, 1 (2014), 7–29. doi:10.1177/0956797613504966

[19] Geoff Cumming and Sue Finch. 2005. Inference by Eye: Confidence Intervals and How to Read Pictures of Data. *The American Psychologist* 60 (Feb. 2005), 170–80. doi:10.1037/0003-066X.60.2.170

[20] Nils Anders Danielsson and Ulf Norell. 2011. Parsing Mixfix Operators. In *Implementation and Application of Functional Languages (Lecture Notes in Computer Science)*, Sven-Bodo Scholz and Olaf Chitil (Eds.). Springer, Berlin, Heidelberg, 80–99. doi:10.1007/978-3-642-24452-0_5

[21] Luís Eduardo de Souza Amorim and Eelco Visser. 2020. Multi-Purpose Syntax Definition with SDF3. In *Software Engineering and Formal Methods (Lecture Notes in Computer Science)*, Frank de Boer and Antonio Cerone (Eds.). Springer International Publishing, Cham, 1–23. doi:10.1007/978-3-030-58768-0_1

[22] Nicola Dell, Vidya Vaidyanathan, Indrani Medhi, Edward Cutrell, and William Thies. 2012. '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*, Joseph A. Konstan, Ed H. Chi, and Kristina Höök (Eds.). ACM, 1321–1330. doi:10.1145/2207676.2208589

[23] Lukas Diekmann. 2019. *Editing Composed Languages*. Ph. D. Dissertation. King's College London.

[24] Lukas Diekmann and Laurence Tratt. 2020. Default Disambiguation for Online Parsers. arXiv:1909.08557 [cs] doi:10.48550/arXiv.1909.08557

[25] Lukas Diekmann and Laurence Tratt. 2020. Don't Panic! Better, Fewer, Syntax Errors for LR Parsers. In *DROPS-IDN/v2/Document/10.4230/LIPIcs.ECOOP.2020.6*. Schloss-Dagstuhl - Leibniz Zentrum für Informatik. doi:10.4230/LIPIcs.ECOOP.2020.6

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

[27] Charles Fischer, Bernard Dion, and Jon Mauney. 1979. *A Locally Least-Cost LR-Error Corrector*. Technical Report. University of Wisconsin-Madison Department of Computer Sciences.

[28] C. N. Fischer, D. R. Milton, and S. B. Quiring. 1980. Efficient LL(1) Error Correction and Recovery Using Only Insertions. *Acta Informatica* 13, 2 (Feb. 1980), 141–154. doi:10.1007/BF00263990

[29] Robert W. Floyd. 1963. Syntactic Analysis and Operator Precedence. *J. ACM* 10, 3 (July 1963), 316–333. doi:10.1145/321172.321179

[30] Susan L. Graham and Steven P. Rhodes. 1975. Practical Syntactic Error Recovery. *Commun. ACM* 18, 11 (Nov. 1975), 639–650. doi:10.1145/361219.361223

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

[32] Sheila A. Greibach. 1965. A New Normal-Form Theorem for Context-Free Phrase Structure Grammars. *J. ACM* 12, 1 (Jan. 1965), 42–52. doi:10.1145/321250.321254

[33] Dick Grune and Ceriel J.H. Jacobs. 2008. *Parsing Techniques: A Practical Guide* (2 ed.). Springer, New York, NY, USA.

[34] Wilfred J. Hansen. 1971. *Creation of Hierarchic Text with a Computer Display*. Ph. D. Dissertation. Stanford University.

[35] D. S. Henderson and M. R. Levy. 1976. An Extended Operator Precedence Parsing Algorithm. *Comput. J.* 19, 3 (Jan. 1976), 229–233. doi:10.1093/comjnl/19.3.229

[36] 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. doi:10.1109/VLHCC.2018.8506483

[37] Michael Homer and James Noble. 2013. A Tile-Based Editor for a Textual Programming Language. In *2013 First IEEE Working Conference on Software Visualization (VISSOFT)*. 1–4. doi:10.1109/VISSOFT.2013.6650546

[38] Gérard Huet. 1997. The Zipper. *Journal of Functional Programming* 7, 5 (Sept. 1997), 549–554. doi:10.1017/S0956796897002864

[39] Lennart C.L. Kats and Eelco Visser. 2010. The Spoofax Language Workbench: Rules for Declarative Specification of Languages and IDEs. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*. ACM, Reno/Tahoe Nevada USA, 444–463. doi:10.1145/1869459.1869497

[40] Paul Klint and Eelco Visser. 1994. Using Filters for the Disambiguation of Context-free Grammars. (1994).

[41] Donald E. Knuth. 1965. On the Translation of Languages from Left to Right. *Information and Control* 8, 6 (Dec. 1965), 607–639. doi:10.1016/S0019-9958(65)90426-2

[42] Michael Kölling. 2010. The Greenfoot Programming Environment. *ACM Trans. Comput. Educ.* 10, 4 (Nov. 2010), 14:1–14:21. doi:10.1145/1868358.1868361

[43] M. R. Levy. 1975. Complete Operator Precedence. *Inform. Process. Lett.* 4, 2 (Nov. 1975), 38–40. doi:10.1016/0020-0190(75)90010-1

[44] John Maloney, Mitchel Resnick, Natalie Rusk, Brian Silverman, and Evelyn Eastmond. 2010. The Scratch Programming Language and Environment. *ACM Transactions on Computing Education* 10, 4 (Nov. 2010), 1–15. doi:10.1145/1868358.1868363

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

[46] 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*. ACM, Ljubljana Slovenia, 28–37. doi:10.1145/3546196.3550164

[47] David Moon, Andrew Blinn, and Cyrus Omar. 2023. Gradual Structure Editing with Obligations. In *2023 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, Washington, DC, USA, 71–81. doi:10.1109/VL-HCC57772.2023.00016

[48] David Moon, Andrew Blinn, Thomas J. Porter, and Cyrus Omar. 2025. Syntactic Completions with Material Obligations. arXiv:2508.16848 [cs] doi:10.48550/arXiv.2508.16848

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

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

[51] Cyrus Omar, David Moon, Andrew Blinn, Ian Voysey, Nick Collins, and Ravi Chugh. 2021. Filling Typed Holes with Live GUIs. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. ACM, Virtual Canada, 511–525. doi:10.1145/3453483.3454059

[52] Cyrus Omar, Ian Voysey, Ravi Chugh, and Matthew A. Hammer. 2019. Live Functional Programming with Typed Holes. *Proceedings of the ACM on Programming Languages* 3, POPL (Jan. 2019), 1–32. doi:10.1145/3290327

[53] 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 '17)*. Association for Computing Machinery, New York, NY, USA, 86–99. doi:10.1145/3009837.3009900

[54] 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. doi:10.1145/3009837

[55] Cyrus Omar, Ian Voysey, Michael Hilton, Joshua Sunshine, Claire Le Goues, Jonathan Aldrich, and Matthew A. Hammer. 2017. Toward Semantic Foundations for Program Editors. arXiv:1703.08694 [cs]

[56] Jacob Prinz, Henry Blanchette, and Leonidas Lampropoulos. 2025. Pantograph: A Fluid and Typed Structure Editor. *Pantograph Implementation* 9, POPL (Jan. 2025), 28:802–28:831. doi:10.1145/3704864

[57] Helmut Richter. 1985. Noncorrecting Syntax Error Recovery. *ACM Transactions on Programming Languages and Systems* 7, 3 (July 1985), 478–489. doi:10.1145/3916.4019

[58] Friedrich Steimann and Robin Stunic. 2024. The Linguistic Theory behind Blockly Languages. In *Proceedings of the 17th ACM SIGPLAN International Conference on Software Language Engineering (SLE '24)*. Association for Computing Machinery, New York, NY, USA, 113–129. doi:10.1145/3687997.3695636

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

[60] Mauricio Verano Merino, Tom Beckmann, Tijs van der Storm, Robert Hirschfeld, and Jurgen J. Vinju. 2021. Getting Grammars into Shape for Block-Based Editors. In *Proceedings of the 14th ACM SIGPLAN International Conference on Software Language Engineering (SLE 2021)*. Association for Computing Machinery, New York, NY, USA, 83–98. doi:10.1145/3486608.3486908

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

[62] Markus Voelter and Vaclav Pech. 2012. Language Modularity with the MPS Language Workbench. In *2012 34th International Conference on Software Engineering (ICSE)*. 1449–1450. doi:10.1109/ICSE.2012.6227070

[63] 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 (SLE 2016)*. Association for Computing Machinery, New York, NY, USA, 28–40. doi:10.1145/2997364.2997365

[64] Tim A Wagner. 1998. *Practical Algorithms for Incremental Software Development Environments*. Ph. D. Dissertation.

[65] Tim A. Wagner and Susan L. Graham. 1998. Efficient and Flexible Incremental Parsing. *ACM Transactions on Programming Languages and Systems* 20, 5 (Sept. 1998), 980–1013. doi:10.1145/293677.293678

[66] Tim A Wagner and Susan L Graham. 1999. History-Sensitive Error Recovery. (1999).

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

[68] Niklaus Wirth and Helmut Weber. 1966. EULER: A Generalization of ALGOL and Its Formal Definition: Part 1. *Commun. ACM* 9, 1 (Jan. 1966), 13–25. doi:10.1145/365153.365162

[69] Yongwei Yuan, Scott Guest, Eric Griffis, Hannah Potter, David Moon, and Cyrus Omar. 2023. Live Pattern Matching with Typed Holes. *Proceedings of the ACM on Programming Languages* 7, OOPSLA1 (April 2023), 609–635. doi:10.1145/3586048

[70] Eric Zhao, Raef Maroof, Anand Dukkipati, Andrew Blinn, Zhiyi Pan, and Cyrus Omar. 2024. Total Type Error Localization and Recovery with Holes. *Proceedings of the ACM on Programming Languages* 8, POPL (Jan. 2024), 2041–2068. doi:10.1145/3632910

Appendix A

Proofs for §5.3

In this appendix, we will use the following shorthand notations:

$$\begin{aligned}
\sigma \triangleleft \chi &\triangleq \sigma \Rightarrow \chi \ldots \\
\chi_L \square \chi_R &\triangleq \exists \sigma.\ \sigma \Rightarrow \ldots \chi_L \chi_R \ldots \\
\chi \triangleright \sigma &\triangleq \sigma \Rightarrow \ldots \chi \\
\sigma \triangle \chi &\triangleq \sigma \Rightarrow \ldots \chi \ldots
\end{aligned}$$

A.1 Precedence Comparisons

Lemmas 3-7 follow by inspection of the elaboration (Fig. 5.15) and injection (Fig. 5.23) rules.

Lemma 3 (Homogeneity). *If $\tau_L \doteq_\square \tau_R$ then either*

- *$\tau_L = \prec$ and $\tau_R = \succ$;*
- *$\tau_L = t_L$ and $\tau_R = t_R$ for some tiles t_L, t_R; or*
- *$\tau_L = \gamma_L^s$ and $\tau_R = \gamma_R^s$ for some grout γ_L, γ_R and sort s.*

Lemma 4 (Grout Precedence). *The following statements hold:*

- *If γ is left-convex, then $\tau \odot_{\sigma?} \gamma^s$ if and only if $\sigma? = \circ$ and $\odot = \lessdot$.*
  If γ is right-convex, then $\gamma^s \odot_{\sigma?} \tau$ if and only if $\sigma? = \circ$ and $\odot = \gtrdot$.
- *If γ_R is left-concave, then $\tau \doteq_{\sigma?} \gamma_R^s$ if and only if $\sigma? = \bullet^0 s^0$ and $\tau = \gamma_L^s$ is right-concave.*
  If γ_L is right-concave, then $\gamma_L^s \doteq_{\sigma?} \tau$ if and only if $\sigma? = \bullet^0 s^0$ and $\tau = \gamma_R^s$ is left-concave.

Lemma 5 (Start Matches End). *$\prec \doteq_{\sigma?} \tau$ if and only if $\sigma? = \bullet^\bot \hat{s}^\bot$ and $\tau = \succ$.*

Lemma 6 (No Escaping). *There exists no τ such that $\tau \lessdot \succ$ or $\prec \gtrdot \tau$.*

Lemma 7 (No Trespassing). *There exists no τ and $\odot$ such that $\tau \odot \prec$ or $\succ \odot \tau$.*

Lemma 8 (Root-Grout Delimits Tiles). *If γ is right-concave, then for all tiles t, there exist slots $[\ \sigma?_i\ ]_{0 \le i \le k}$ and tiles $[\ t_i\ ]_{0 \le i \le k}$ such that $\gamma^{\hat{s}} \lessdot_{\sigma?_0} t_0 [\ \doteq_{\sigma?_i} t_i\ ]_{1 \le i \le k} = t$.*

Proof. By Assumption 2 (Unique Tiles), there exists sort s and precedence p such that $\ldots t \ldots \in [\![\mathcal{G}(s,p)]\!]$. By Assumption 1 (Operator Form), there exist optional sorts $\lceil s?_i \rceil_{0 \le i \le n+1}$ and tiles $\lceil t_i \rceil_{0 \le i \le n}$ such that $t_k = t$ for some $k \le n$ and

$$s?_0 \lceil t_i s?_{i+1} \rceil_{0 \le i \le n} \in [\![\mathcal{G}(s,p)]\!] \tag{A.1}$$

Depending on whether $s?_0 = s$ and $s?_{n+1} = s$, apply one of the elaboration rules in Fig. 5.15 to (A.1) to construct a production rule for ${}^0s^0$, knowing that $0 \prec_s p$ and $p \succ_s 0$ as needed. Across all cases, we can show

$${}^0s^0 \Rightarrow \ldots t_0 \lceil \lceil s?_i \rceil t_i \rceil_{1 \le i \le n} \ldots \tag{A.2}$$

Define $\lceil \sigma?_i = \lceil s?_i \rceil \rceil_{0 \le i \le n}$. `Prec-EQ` applied to (A.2) gives us

$$\lceil t_i \doteq_{\sigma?_i} t_{i+1} \rceil_{0 \le i < n} \tag{A.3}$$

It remains to show $\gamma^{\hat{s}} \prec_{\sigma?_0} t_0$. It can be shown using one of the injection rules in Fig. 5.23 that $\gamma^{\hat{s}} \,\square\, {}^0\hat{s}^0$. □

Lemma 9 (Reachability)**.** *For every tile t, there exist slots $\lceil \sigma?_i \rceil_{1 \le i \le k}$ and tiles $\lceil t_i \rceil_{1 \le i \le k}$ such that $\prec \lceil \preceq_{\sigma?_i} t_i \rceil_{1 \le i \le k} = t$.*

Proof. Follows from $\prec \lessdot_\circ \lhd^{\hat{s}}$ (since $\prec \,\square\, {}^\perp\hat{s}^\perp \lhd \lhd^{\hat{s}}$) and Lemma 8 (Root-Grout Delimits Tiles). □

A.2 Precedence Bounds

Lemmas 10-13 follow by inspection of the elaboration (Fig. 5.15) and injection (Fig. 5.23) rules.

Lemma 10. *If ${}^p s^\square \Rightarrow {}^q s^m t \ldots$ then $p \preccurlyeq_s q \prec_s m$.*

Lemma 11. *If ${}^p s^q \lhd^* {}^m r^n$ and $s \neq r$ then ${}^p s^q \lhd^* {}^\perp r^\perp$.*

Lemma 12. *If $t \,\square\, {}^p s^q$ then $t \,\square\, {}^p s^\perp$.*

Lemma 13. *If $\sigma \lhd^* \rho$ and $\sigma \sim s$ then either $\rho \sim s$ or $\sigma \Rightarrow \rho\, \Sigma^s$.*

Lemma 14. *If $\rho \Rightarrow^* \sigma?\tau \ldots$ then there exists a nonterminal σ such that*

$$\rho \Rightarrow^* \sigma \Rightarrow \sigma?\tau \ldots\ldots$$

Proof. We will show a slight generalization: if $\rho \Rightarrow^+ \sigma?\tau \ldots$ then there exists a nonterminal σ such that $\rho \Rightarrow^* \sigma \Rightarrow \sigma?\tau \ldots\ldots$. This is a proper generalization, despite the use of $\Rightarrow^+$ instead of $\Rightarrow^*$, because $\rho \neq \sigma?\tau \ldots$ and therefore $\rho \Rightarrow^* \sigma?\tau \ldots$ implies $\rho \Rightarrow^+ \sigma?\tau \ldots$, which implies $\rho \ldots \Rightarrow^+ \sigma?\tau \ldots$. We will assume here that every yield step $\overline{\chi}_0 \Rightarrow \overline{\chi}_1$ is a leftmost yield, meaning it rewrites the leftmost nonterminal in $\overline{\chi}_0$.

Induct on the premise $\rho \ldots \Rightarrow^+ \sigma?\tau \ldots$:

- Suppose $\rho\ldots \Rightarrow \sigma?\tau\ldots$. Our assumption of leftmost yields implies $\rho \Rightarrow \sigma?\tau\ldots$, so returning $\sigma = \rho$ gives $\rho \Rightarrow^* \sigma?\tau\ldots\ldots$ as desired.
- Suppose $\rho\ldots \Rightarrow \overline{\chi} \Rightarrow^+ \sigma?\tau\ldots$.
  Further suppose $\overline{\chi} = \rho_1\ldots$ for some nonterminal ρ_1. Then our assumption of leftmost yields implies $\rho \lhd \rho_1$, and our inductive hypothesis gives us σ such that $\rho_1 \Rightarrow^* \sigma \Rightarrow \sigma?\tau\ldots\ldots$. Putting it altogether, we have $\rho \Rightarrow^* \sigma \Rightarrow \sigma?\tau\ldots\ldots$ as desired.
  Otherwise, assume $\overline{\chi} = \tau_1\ldots$ for some terminal τ_1. Given $\rho\ldots \Rightarrow \tau_1\ldots$, our assumption of leftmost yields implies $\rho \Rightarrow \tau_1\ldots$. Given $\tau_1\ldots \Rightarrow^+ \sigma?\tau\ldots$, it must be that $\sigma? = \circ$ and $\tau_1 = \tau$. It follows that setting $\sigma = \rho$ gives $\rho \Rightarrow^* \sigma \Rightarrow \tau_1\ldots = \sigma?\tau\ldots\ldots$ as desired.

□

Lemma 15. *If $\tau_L \lessdot_{\rho?} \tau_R$, then there exists a nonterminal $\sigma = {}^{\square}s^{\perp}$ such that $\tau_L \,\square\, \sigma$ and $\sigma \Rightarrow^* \rho?\tau_R\ldots$.*

Proof. Invert the premise $\tau_L \lessdot_{\rho?} \tau_R$ to get nonterminal $\rho = {}^{p}s^{q}$ such that

$$\tau_L \,\square\, \rho \tag{A.4}$$
$$\rho \Rightarrow^* \rho?\tau_R\ldots \tag{A.5}$$

Let $\sigma = {}^{p}s^{\perp}$. Apply Lemma 12 to (A.4) to get $\tau_L \,\square\, \sigma$. Since $q \geqslant_s \perp$ by definition, we have by rule `Produce-Tighten` that $\sigma \Rightarrow \rho$, which together with (A.5) gives $\sigma \Rightarrow^* \rho?\tau_R\ldots$ as desired. □

A.3 Proof of Theorem 1

For all sorts s, precedence levels p_L, p_R, and tiles t_L, t_R such that $\ldots t_L s \in [\![\mathcal{G}(s, p_L)]\!]$ and $s t_R\ldots \in [\![\mathcal{G}(s, p_R)]\!]$, the following equivalences hold:

$$t_L \lessdot t_R \iff p_L < p_R$$
$$t_L \gtrdot t_R \iff p_L > p_R$$

Proof. We assume the premises. Beginning with the left-to-right direction of the first equivalence, we also assume that $t_L \lessdot t_R$. By repeated rule inversion, we obtain ${}^{a}s^{b} \Leftarrow \ldots {t_L}^{c}s^{d}$ and ${}^{d^+}s^{d^+} \Leftarrow {}^{e}s^{f}t_R\ldots$, for some precedences a, b, c, d, e, f such that $c \leq c^+$ and $d^+ \geq d$ (the result of `Produce-Tighten`). Applying rule inversion once more to each yields the inequalities $p_L < c$ and $c^+ = \min(e, p_R)$. Together we have $p_L < c \leq c^+ \leq p_R$, completing the implication.

For the other direction, we assume that $p_L < p_R$. We will first derive ${}^{a}s^{p_R} \Leftarrow \ldots {t_L}^{p_R}s^{p_R}$ for some extension $\ldots$ and some precedence a. In the case that the prefix of $\ldots t_L s$ begins with s, `PElab-Infix` is used, with the bounds of each χ_i being $\top$ except for $\chi_k = {}^{p_R}s^{p_R}$. The premises are satisfied with $\ldots t_L s \in [\![\mathcal{G}(s, p_L)]\!]$, $\top > p_L < p_R$, and $p_L = \min(p_L, p_R)$. The case when the prefix does not begin with s is analogous, with `PElab-Prefix` used instead.

From this reduction judgment we obtain $t_L \,\square\, {}^{p_R}s^{p_R}$, the first premise to our desired conclusion. For the second premise, it suffices to show that ${}^{a}s^{p_R} \Leftarrow {}^{\top}s^{\top}t_R\ldots$, which we obtain from $s t_R\ldots \in [\![\mathcal{G}(s, p_R)]\!]$ via `PElab-Infix` or `PElab-Postfix` as needed, with $\top > p_R < \top$ and $p_R = \min(p_R, \top)$.

The second equivalence is proven symmetrically. □

A.4 Proof of Lemma 1 (Valid Prefixes)

Lemma 16 (Splice). *Suppose symbol x appears uniquely in regex g. If $\overline{x}_L x \ldots \in [\![g]\!]$ and $\ldots x \overline{x}_R \in [\![g]\!]$, then $\overline{x}_L x \overline{x}_R \in [\![g]\!]$.*

Proof. Proceed by induction on g. Start each case by assuming the premises:

$$\overline{x}_L x \overline{y}_R \in [\![g]\!] \tag{A.6}$$
$$\overline{y}_L x \overline{x}_R \in [\![g]\!] \tag{A.7}$$

- $g = \epsilon$: Impossible, cannot derive a nonempty string.
- $g = x_0$: Then $x_0 = x$ and $\overline{x}_L = \overline{y}_L = \overline{x}_R = \overline{y}_R = \circ$ and we have the goal by assumption.
- $g = g_L \mid g_R$: x appears in either g_L or g_R—assume g_L without loss of generality. Then

  $$\overline{x}_L x \overline{y}_R \in [\![g_L]\!] \tag{A.8}$$
  $$\overline{y}_L x \overline{x}_R \in [\![g_L]\!] \tag{A.9}$$

  By the inductive hypothesis, we have $\overline{x}_L x \overline{x}_R \in [\![g_L]\!]$ and therefore $\overline{x}_L x \overline{x}_R \in [\![g_L]\!] \cup [\![g_R]\!] = [\![g]\!]$.
- $g = g_L \cdot g_R$: x appears in either g_L or g_R—assume g_L without loss of generality.
  There exists partitions $\overline{y}_R = \overline{y}_\ell \overline{y}_r$ and $\overline{x}_R = \overline{x}_\ell \overline{x}_r$ such that

  $$\overline{x}_L x \overline{y}_\ell \in [\![g_L]\!] \tag{A.10}$$
  $$\overline{y}_r \in [\![g_R]\!] \tag{A.11}$$
  $$\overline{y}_L x \overline{x}_\ell \in [\![g_L]\!] \tag{A.12}$$
  $$\overline{x}_r \in [\![g_R]\!] \tag{A.13}$$

  Apply the inductive hypothesis to (A.10) and (A.12) to get $\overline{x}_L x \overline{x}_\ell \in [\![g_L]\!]$, which combined with (A.13) gives $\overline{x}_L x \overline{x}_R = \overline{x}_L x \overline{x}_\ell \overline{x}_r \in [\![g_L \cdot g_R]\!]$.
- $g = g_0^*$:
  There exist $m, n \in \mathbb{N}$ such that

  $$\overline{x}_L x \overline{y}_R \in [\![g_0^m]\!] \tag{A.14}$$
  $$\overline{y}_L x \overline{x}_R \in [\![g_0^n]\!] \tag{A.15}$$

  We know $m > 0$ and $n > 0$ because $g_0^0 = \epsilon$ derives only the empty string, meaning

  $$\overline{x}_L x \overline{y}_R \in [\![g_0 \cdot g_0^{m-1}]\!] \tag{A.16}$$
  $$\overline{y}_L x \overline{x}_R \in [\![g_0 \cdot g_0^{n-1}]\!] \tag{A.17}$$

  Case analysis on whether x appears in g_0^{m-1} and in g_0^{n-1}:
  - Suppose x appears in g_0^{m-1} i.e. there exists partition $\overline{x}_L = \overline{x}_\ell \overline{x}_r$ such that

    $$\overline{x}_\ell \in [\![g_0]\!] \tag{A.18}$$
    $$\overline{x}_r x \overline{y}_R \in [\![g_0^{m-1}]\!] \tag{A.19}$$

Apply the inductive hypothesis to (A.19) and (A.15) to get $\overline{x}_r x \overline{x}_R \in [\![g_0^*]\!]$, which combined with (A.18) gives

$$\overline{x}_L x \overline{x}_R = \overline{x}_\ell \cdot \overline{x}_r x \overline{x}_R \in [\![g_0 \cdot g_0^*]\!] = [\![g_0^*]\!] \tag{A.20}$$

– Suppose x appears in g_0^{n-1} i.e. there exists partition $\overline{y}_L = \overline{y}_\ell \overline{y}_r$ such that

$$\overline{y}_\ell \in [\![g_0]\!] \tag{A.21}$$
$$\overline{y}_r x \overline{x}_R \in [\![g_0^{n-1}]\!] \tag{A.22}$$

Apply the inductive hypothesis to (A.14) and (A.22) to get

$$\overline{x}_L x \overline{x}_R \in [\![g_0^m \cdot g_0^{n-1}]\!] \subseteq [\![g_0^*]\!] \tag{A.23}$$

– Suppose x appears in neither g_0^{m-1} nor g_0^{n-1}. Then there exist partitions $\overline{y}_R = \overline{y}_\ell \overline{y}_r$ and $\overline{x}_R = \overline{x}_\ell \overline{x}_r$ such that

$$\overline{x}_L x \overline{y}_r \in [\![g_0]\!] \tag{A.24}$$
$$\overline{y}_L x \overline{x}_r \in [\![g_0]\!] \tag{A.25}$$
$$\overline{y}_r \in [\![g_0^{m-1}]\!] \tag{A.26}$$
$$\overline{y}_r \in [\![g_0^{n-1}]\!] \tag{A.27}$$

Apply the same argument as in the case $g = g_L \cdot g_R$ to reach the goal.

□

Proof of Lemma 1 (Valid Prefixes). Assume the premises

$$\tau \lessdot_{\rho ?_0} t_0 \tag{A.28}$$
$$\left[t_{i-1} \doteq_{\rho ?_i} t_i \right]_{0<i\leq k} \tag{A.29}$$
$$t_k \sqsupset \rho ?_{k+1} \tag{A.30}$$

Apply Lemma 15 to (A.28) to get nonterminal $\sigma_\tau = {}^m s_\tau{}^\perp$ such that

$$\tau \sqsupset \sigma_\tau \tag{A.31}$$
$$\sigma_\tau \Rightarrow^* \rho ?_0 t_0 \ldots \tag{A.32}$$

Apply Lemma 14 to (A.32) to get nonterminal σ_0 such that

$$\sigma_\tau \Rightarrow^* \sigma_0 \ldots \tag{A.33}$$
$$\sigma_0 \Rightarrow \rho ?_0 t_0 \ldots \tag{A.34}$$

Invert (A.29) to get nonterminals $\left[\sigma_i \right]_{0<i\leq k}$ such that

$$\left[\sigma_i \Rightarrow \ldots t_{i-1} \rho ?_i t_i \ldots \right]_{0<i\leq k} \tag{A.35}$$

Finally, unabbreviate (A.30) to get nonterminal σ_{k+1} such that

$$\sigma_{k+1} \Rightarrow \ldots t_k \rho?_{k+1} \ldots \tag{A.36}$$

First, we will show that the productions in (A.34)-(A.36) are backed by a shared derivation from some regex in $\mathcal{G}$, given Assumption 2 (Unique Tiles). Letting $\left[\ \sigma_i = {}^{p_i}s_i{}^{q_i}\ \right]_{0 \le i \le k}$, invert (A.34)-(A.36) by rule `Produce-Subsume` to get

$${}^{p_0}s_0{}^{q_0} \Leftarrow \rho?_0 t_0 \ldots \tag{A.37}$$

$$\left[\ {}^{p_i}s_i{}^{q_i} \Leftarrow \ldots t_{i-1} \rho?_i t_i \ldots\ \right]_{0 < i \le k} \tag{A.38}$$

$${}^{p_{k+1}}s_{k+1}{}^{q_{k+1}} \Leftarrow \ldots t_k \rho?_{k+1} \ldots \tag{A.39}$$

Invert again to get precedence levels $[\ n_i\ ]_{0 \le i \le k+1}$ and optional sorts $[\ r?_i\ ]_{0 \le i \le k+1x}$ such that

$$[\ \rho?_i \sim r?_i\ ]_{0 \le i \le k+1} \tag{A.40}$$

$$r?_0 t_0 \ldots \in [\![\mathcal{G}(s_0, n_0)]\!] \tag{A.41}$$

$$[\ \ldots t_{i-1} r?_i t_i \ldots \in [\![\mathcal{G}(s_i, n_i)]\!]\ ]_{0 < i \le k} \tag{A.42}$$

$$\ldots t_k r?_{k+1} \ldots \in [\![\mathcal{G}(s_{k+1}, n_{k+1})]\!] \tag{A.43}$$

Assumption 2 (Unique Tiles) applied to (A.41)-(A.43) gives us sort s and precedence n such that

$$[\ s = s_i\ ]_{0 \le i \le k+1} \tag{A.44}$$

$$[\ n = n_i\ ]_{0 \le i \le k+1} \tag{A.45}$$

Given (A.44) and (A.45), we can now apply Lemma 16 (Splice) to (A.41)-(A.43) to get

$$[\ r?_i t_i\ ]_{0 \le i \le k}\ r?_{k+1}\ \overline{x} \in [\![\mathcal{G}(s, n)]\!] \tag{A.46}$$

for some symbol sequence $\overline{x}$, as desired. By Assumption 1 (Operator Form), we can rewrite $\overline{x}$ with optional sorts $[\ r?_i\ ]_{k+1 < i \le \ell+1}$ and tiles $[\ t_i\ ]_{k < i \le \ell}$ such that

$$r?_0 [\ t_i r?_{i+1}\ ]_{0 \le i \le \ell} \in [\![\mathcal{G}(s, n)]\!] \tag{A.47}$$

Now it remains to determine bounds p, q, tiles $[\ t_i\ ]_{k < i \le \ell}$, and slots $[\ \rho?_i\ ]_{k+1 < i \le \ell+1}$ such that

$$\sigma_\tau \Rightarrow^* {}^{p}s^{q} \ldots \tag{A.48}$$

$${}^{p}s^{q} \Leftarrow \rho?_0 [\ t_i\ \rho?_{i+1}\ ]_{0 \le i \le \ell} \tag{A.49}$$

We will show (A.49) using one of the elaboration rules in Fig. 5.15. Determining which elaboration rule to use requires case analysis on whether $r?_0 = \bullet s$ and $r?_{\ell+1} = \bullet s$. Without loss of generality, assume $r?_0 = \bullet s$ and $r?_{\ell+1} \neq \bullet s$.

Proceed by case analysis on whether $s_\tau = s$:

- Suppose $s_\tau \neq s$. Rewrite (A.33) with (A.44) to get $\sigma_\tau \triangleleft^* {}^{\square}s^{\square}$, then apply Lemma 11 to get

$$\sigma_\tau \triangleleft^* {}^{\perp}s^{\perp} \tag{A.50}$$

  Pick $p = \perp$ and $q = \perp$ to reach (A.48).
  It remains to construct slots $[\ \rho?_i\ ]_{k+1<i\leq\ell+1}$ satisfying (A.49). Apply `PElab-Postfix` to (A.47) to get

$${}^{p}s^{q} \Leftarrow \rho?_0 t_0 [\ \lceil r?_i \rceil\, t_i\ ]_{0<i\leq\ell} \tag{A.51}$$

  where we write

$$\lceil r?_i \rceil \triangleq \begin{cases} \rho?_i & \text{if } 0 < i \leq k \\ \bullet^{\perp} r_i {}^{\perp} & \text{if } k < i \text{ and } r?_i = \bullet r_i \\ \circ & \text{if } r?_i = \circ \end{cases} \tag{A.52}$$

  Picking $[\ \rho?_i = \lceil r?_i \rceil\ ]_{k<i\leq\ell+1}$ gives us our goal (A.49).

- Suppose $s_\tau = s$. Pick $p = m$ and $q = \perp$, i.e. $\sigma_\tau = {}^{m}s^{\perp} = {}^{p}s^{q}$, which satisfies (A.48).
  It remains to construct slots $[\ \rho?_i\ ]_{k<i\leq\ell+1}$ satisfying (A.49). We have ${}^{m}s^{\perp} = \sigma_\tau \triangleleft^* \rho_0 = {}^{p_0}s^{q_0}$. By Lemma 10, we have $m \preccurlyeq_s p_0$. We have $r?_0 = \bullet s$ which implies $\rho?_0 = \bullet^{n_L} s^{n_R}$ for some n_L, n_R. We have ${}^{p_0}s^{q_0} \Rightarrow {}^{n_L}s^{n_R} t_0 \ldots$ by (A.34)—case analysis on the underlying reduction (either `PElab-Postfix` or `PElab-Infix`) tells us that $n_R = n$. By Lemma 10, we have $p_0 \preccurlyeq_s n_L \prec_s n_R$. Inverting (A.34) gives us $n_R = n$. Therefore $m \preccurlyeq_s p_0 \preccurlyeq_s n_L \prec_s n_R = n$.
  We have $m \prec_s n$ and that the rightmost symbol of the derived string in (A.47) is t_ℓ, so apply `PElab-Postfix` to (A.47) to get

$${}^{p}s^{q} \Leftarrow {}^{p}s^{n} t_0 [\ \lceil r?_i \rceil\, t_i\ ]_{0<i\leq\ell} \tag{A.53}$$

  where $\lceil r?_i \rceil$ is defined as in (A.52). The same reasoning applied in the case $s_\tau \neq s$ from (A.51) onward gives us our goal (A.49).

□

A.5 Proof of Lemma 2

Lemma 17 (Fill Produces Well-Formed Terms)**.** *If* $\mathbb{R}?\ \curvearrowright\ [\ \sigma?_i\ ]_{1\leq i\leq k} = [\ \mathbb{S}?_i\ ]_{1\leq i\leq k}$ *then* $[\ \sigma?_i \Rrightarrow \mathbb{S}?_i\ ]_{1\leq i\leq k}$.

Proof of Lemma 2 (Pushing is Sound and Total). Corollary of Lemma 18 (Generalized Push Totality), generalized to support proof by induction over recursive `Reduce` steps. □

Lemma 18 (Generalized Push Totality)**.** *For every stack* $\mathbb{K}$*, reduction sequence* $\overline{\mathbb{R}}$*, and token* τ *such that* $\mathbb{K} \underset{\overline{\mathbb{R}}}{\leftarrow} \square$ wf*, there exists* $\mathbb{K}'$ *such that* $\mathbb{K} \underset{\overline{\mathbb{R}}}{\leftarrow} \tau = \mathbb{K}'$ *and* $\mathbb{K}'$ wf*.*

$\boxed{\mathbb{S}\ \mathsf{nat}}$ Term $\mathbb{S}$ is natural

$$\frac{\texttt{Natural} \qquad 0 \preccurlyeq_s p \qquad {}^{p}s^{q} \Leftarrow \mathbb{S} \qquad q \succcurlyeq_s 0}{\mathbb{S}\ \mathsf{nat}}$$

$\boxed{\mathbb{K} \underset{\overline{\mathbb{R}}}{\leftarrow} \square\ \mathsf{wf}}$ Stack configuration $\mathbb{K} \underset{\overline{\mathbb{R}}}{\leftarrow} \square$ is well-formed

$$\frac{\texttt{WFConfig}\ (k \geq 0) \qquad \mathbb{K}\ \mathsf{wf} \qquad \mathsf{hd}(\mathbb{K}) \triangleright \sigma? \qquad [\,\mathbb{R}_i\,]_{0\leq i\leq k} \rightharpoonup \sigma? = \overline{\mathbb{S}?} \qquad [\,\mathbb{R}_i\ \mathsf{nat}\,]_{0\leq i\leq k}}{\mathbb{K} \underset{[\,\mathbb{R}_i\,]_{0\leq i\leq k}}{\longleftarrow\!\!\!\!\!\!\!\!\!\!\!\!} \square\ \mathsf{wf}}$$

Figure A.1: Push invariant

Proof. Proceed by strong induction on the stack $\mathbb{K}$.

- Suppose $\mathbb{K}$ is empty, i.e. $\mathbb{K} = \prec$.
  First, we will show that we can apply `Shift`, i.e. there exist nonterminals $[\,\rho_i\,]_{0\leq i\leq k}$, tokens $[\,\tau_i\,]_{0\leq i\leq k}$, and cells $[\,\mathbb{S}?_i\,]_{0\leq i\leq k}$ such that

$$\prec \left[\, \preccurlyeq \rho?_i\ \tau_i \,\right]_{0\leq i\leq k} = \tau \tag{A.54}$$

$$\overline{\mathbb{R}} \rightharpoonup [\,\rho?_i\,]_{0\leq i\leq k} = [\,\mathbb{S}?_i\,]_{0\leq i\leq k} \tag{A.55}$$

  Proceed by case analysis on τ:

  - First suppose $\tau = \succ$. Goal (A.54) is satisfied by the fact that $\prec \doteq_{\bullet^{\perp}\hat{s}^{\perp}} \succ$. It remains to show there exist cells $\overline{\mathbb{S}?}$ such that $\overline{\mathbb{R}} \rightharpoonup \bullet^{\perp}\hat{s}^{\perp} = \overline{\mathbb{S}?}$. Inverting the premise $\prec \underset{\overline{\mathbb{R}}}{\leftarrow} \square\ \mathsf{wf}$, there exists slot $\rho?$ and cells $\overline{\mathbb{S}?}$ such that

$$\prec \triangleright \rho? \tag{A.56}$$

$$\overline{\mathbb{R}} \rightharpoonup \rho? = \overline{\mathbb{S}?} \tag{A.57}$$

    Given (A.56), we further have

$$\rho? = \bullet^{\perp}\hat{s}^{\perp} \tag{A.58}$$

    with which rewriting (A.57) gives us (A.55).
  - Otherwise, suppose $\tau = t$ for some tile t. Let $\overline{\mathbb{R}} = [\,\mathbb{R}_i\,]_{0<i\leq k}$ for some $k \geq 0$.
    Lemma 8 gives us tiles $[\,t_i\,]_{0\leq i\leq \ell}$ and slots $[\,\sigma?_i\,]_{0\leq i\leq \ell}$ such that

$$\natural^{\hat{s}} \prec_{\sigma?_0} t_0 \left[\, \doteq_{\sigma?_i} t_i \,\right]_{1\leq i\leq \ell} = t \tag{A.59}$$

We can further show that

$$\prec \lessdot_{\circ} \varpi^{\hat{s}} \left[\doteq_{\bullet}{}^{0}\hat{s}^{0} \varpi^{\hat{s}} \right]_{0 \le i \le k} \tag{A.60}$$

given that $\prec \triangleleft {}^{\perp}\hat{s}^{\perp}$ and ${}^{\perp}\hat{s}^{\perp} \Leftarrow \varpi^{\hat{s}} \left[{}^{0}\hat{s}^{0} \varpi^{\hat{s}} \right]_{0 \le i \le k}$ by rule `GInj-Prefix`. Composing (A.60) with (A.59) gives us (A.54).

To show (A.55), it suffices to show there exist cells $[\, \mathbb{S}?_i \,]_{0 < i \le k}$ such that

$$\left[\mathbb{R}_i \curvearrowright \bullet^{0}\hat{s}^{0} = \mathbb{S}?_i \right]_{0 < i \le k} \tag{A.61}$$

which can be shown using reduction naturality.

Given (A.54) and (A.55), rule `Shift` gives us

$$\prec \underset{\overline{\mathbb{R}}}{\longleftarrow} \tau = \prec \left[\trianglelefteq \mathbb{S}?_i \; \tau_i \right]_{0 \le i \le k} \tag{A.62}$$

It remains to show

$$\prec \left[\trianglelefteq \mathbb{S}?_i \; \tau_i \right]_{0 \le i \le k} \; \mathsf{wf} \tag{A.63}$$

Lemma 17 (Fill Produces Well-Formed Terms) applied to (A.55) gives us

$$[\, \sigma?_i \Rrightarrow \mathbb{S}?_i \,]_{0 \le i \le k} \tag{A.64}$$

Using (A.54) and (A.64) and $\prec$ wf (given by `WFStack-Empty`), apply rule `WFStack-Cons` $k + 1$ times in succession to get (A.63).

- Otherwise, assume $\mathbb{K}$ is nonempty and the inductive hypothesis that, for any substack $\mathbb{K}_0$ of $\mathbb{K}$ and reductions $\overline{\mathbb{S}}_0$ such that $\mathbb{K}_0 \underset{\overline{\mathbb{R}}_0}{\longleftarrow} \square$ wf there exists $\mathbb{K}'$ such that $\mathbb{K}_0 \underset{\overline{\mathbb{S}}_0}{\longleftarrow} \tau = \mathbb{K}'$ and $\mathbb{K}'$ wf.

  Inverting the premise $\mathbb{K} \underset{\overline{\mathbb{R}}}{\longleftarrow} \square$ wf, there exists slot $\sigma?$ and cells $\overline{\mathbb{S}?}$ such that

$$\mathbb{K}\ \mathsf{wf} \tag{A.65}$$

$$\mathsf{hd}(\mathbb{K}) \triangleleft \rho? \tag{A.66}$$

$$\overline{\mathbb{R}} \curvearrowright \rho? = \overline{\mathbb{S}?} \tag{A.67}$$

  Since $\mathbb{K}$ is nonempty, Lemma 5 (Start Matches End) implies there exist tokens $[\, \tau_i \,]_{0 \le i \le k}$ and cells $[\, \mathbb{R}?_i \,]_{0 \le i \le k}$ and stack $\mathbb{K}_0$ such that

$$\mathbb{K} = \mathbb{K}_0 \lessdot \mathbb{R}?_0 \; \tau_0 \left[\doteq \mathbb{R}?_i \; \tau_i \right]_{1 \le i \le k}$$

for some $k \geq 0$. Starting with (A.65), invert rule `WFStack-Cons` $k+1$ times to get

$$\mathbb{K}_0 \text{ wf} \tag{A.68}$$

$$\mathsf{hd}(\mathbb{K}_0) \lessdot_{\rho?_0} \tau_0 \Big[\doteq_{\rho?_i} \tau_i \Big]_{1 \leq i \leq k} \tag{A.69}$$

$$\big[\, \rho?_i \Rrightarrow \mathbb{R}?_i \,\big]_{0 \leq i \leq k} \tag{A.70}$$

Lemma 3 (Homogeneity) applied to (A.69) implies either

$$\big[\, \tau_i = t_i \,\big]_{0 \leq i \leq k} \tag{A.71}$$

for some tiles $\big[\, t_i \,\big]_{0 \leq i \leq k}$ or

$$\big[\, \tau_i = \gamma_i^s \,\big]_{0 \leq i \leq k} \tag{A.72}$$

for some grout $\big[\, \gamma_i \,\big]_{0 \leq i \leq k}$ and sort s.

– Suppose (A.71) holds. We will show that it is possible to apply rule `Reduce`.
By Lemma 1 (Valid Prefixes) applied to (A.69) and (A.66), there exist nonterminals σ_0, σ, slots $\big[\, \rho?_i \,\big]_{k < i \leq \ell+1}$, and tiles $\big[\, t_i \,\big]_{k < i \leq \ell}$ for some $\ell \geq k$ such that

$$\rho?_{k+1} = \rho? \tag{A.73}$$

$$\mathsf{hd}(\mathbb{K}_0) \triangleleft \sigma_0 \Rightarrow^* \sigma \ldots \tag{A.74}$$

$$\sigma \Leftarrow \rho?_0 \big[\, t_i \; \rho?_{i+1} \,\big]_{0 \leq i \leq \ell} \tag{A.75}$$

To conclude with rule `Reduce`, given (A.75) and (A.70), it remains to show there exist cells $\big[\, \mathbb{R}?_i \,\big]_{k < i \leq \ell+1}$ and stack $\mathbb{K}'$ such that

$$\overline{\mathbb{R}} \leftharpoonup \big[\, \rho?_i \,\big]_{k < i \leq \ell+1} = \big[\, \mathbb{R}?_i \,\big]_{k < i \leq \ell+1} \tag{A.76}$$

$$\mathbb{K}_0 \xleftarrow[\left\{\mathbb{R}?_0 \big[\, t_i \, \mathbb{R}?_{i+1} \,\big]_{0 \leq i \leq \ell}\right\}]{} \tau = \mathbb{K}' \tag{A.77}$$

* To show (A.76), it suffices by rule `Fill-Partition` to show there exists a partition $\overline{\mathbb{R}} = \big[\, \overline{\mathbb{R}}_i \,\big]_{k < i \leq \ell+1}$ and cells $\big[\, \mathbb{R}?_i \,\big]_{k < i \leq \ell+1}$ such that

$$\big[\, \overline{\mathbb{R}}_i \leftharpoonup \rho?_i = \mathbb{R}?_i \,\big]_{k < i \leq \ell+1} \tag{A.78}$$

Pick the partition $\overline{\mathbb{R}} = \overline{\mathbb{R}}\big[\,\cdot\,\big]_{k+1 < i \leq \ell+1}$ and use (A.67) for index $k+1$ and either rule `Fill-None` or `Fill-Default` for the remaining indices $k+1 < i \leq \ell+1$.

* To show (A.77), it suffices by the inductive hypothesis to show

$$\mathbb{K}_0 \xleftarrow[\left\{\mathbb{R}?_0 \big[\, t_i \, \mathbb{R}?_{i+1} \,\big]_{0 \leq i \leq \ell}\right\}]{} \square \text{ wf} \tag{A.79}$$

Let $\mathbb{S} = \left\{\mathbb{R}?_0 \big[\, t_i \, \mathbb{R}?_{i+1} \,\big]_{0 \leq i \leq \ell}\right\}$. Given (A.68) and (A.74), it suffices to show there exists

term $\mathbb{S}_0$ such that

$$\mathbb{S} \rightharpoonup \bullet\sigma_0 = \bullet\mathbb{S}_0 \tag{A.80}$$

It suffices by rule `Fill-Postfix` to show

$$\sigma_0 \Rrightarrow \{\mathbb{S}\,\mathbb{D}^s\} \tag{A.81}$$

It suffices by rule `Produce-Term` to show

$$\sigma_0 \Rightarrow \sigma\,\mathbb{D}^s \tag{A.82}$$

It suffices by rule `Produce-Subsume` to show

$$\sigma_0 \Leftarrow \sigma\,\mathbb{D}^s \tag{A.83}$$

Let $\sigma_0 = {}^{p_0}s_0{}^{q_0}$ and $\sigma = {}^{p}s^{q}$. It suffices by rule `GInj-Postfix` to show

$${}^{0}s_0{}^{0} \triangleleft^* \sigma \tag{A.84}$$

By Lemma 13 applied to (A.74), either $\sigma \sim s_0$ or $\sigma_0 \Rightarrow \sigma\,\mathbb{D}^{s_0}$.

- Case $\sigma \sim s_0$, i.e. $s = s_0$: By reduction naturality applied to (A.75), we know $0 \preccurlyeq_s p$ and $q \succcurlyeq_s 0$, so we are done by `Produce-Tighten`.
- Case $\sigma_0 \Rightarrow \sigma\,\mathbb{D}^{s_0}$: Invert rule `Produce-Subsume` to get $\sigma_0 \Leftarrow \sigma\,\mathbb{D}^{s_0}$. Invert rule `GInj-Postfix` in turn to conclude.

– Suppose (A.72) holds. It suffices by rule `Degrout` to show there exists $\mathbb{K}'$ such that

$$\mathbb{K}_0 \xleftarrow[\left[\,\mathbb{R}?_i\,\right]_{0\le i\le k}\overline{\mathbb{R}}]{} \tau = \mathbb{K}' \tag{A.85}$$

It suffices by our inductive hypothesis to show

$$\mathbb{K}_0 \xleftarrow[\left[\,\mathbb{R}?_i\,\right]_{0\le i\le k}\overline{\mathbb{R}}]{} \square\ \mathsf{wf} \tag{A.86}$$

That is, by `WFConfig`, to show

$$\mathbb{K}_0\ \mathsf{wf} \tag{A.87}$$

$$\mathsf{hd}(\mathbb{K}_0) \triangleright \sigma_0? \tag{A.88}$$

$$\left[\,\mathbb{R}?_i\,\right]_{0\le i\le k}\overline{\mathbb{R}} \rightharpoonup \sigma?_0 = \overline{\mathbb{S}?_0} \tag{A.89}$$

$$\left[\,\mathbb{R}?_i\,\right]_{0\le i\le k}\overline{\mathbb{R}}\ \mathsf{nat} \tag{A.90}$$

For some $\sigma?_0$ and $\overline{\mathbb{S}?_0}$. We already have (A.87) from (A.68). We obtain (A.88) for $\sigma?_0 = \bullet\sigma_0$ by rule inversion of `Prec-LT` on $\mathsf{hd}(\mathbb{K}_0) \lessdot_{\rho?_0} \tau_0$ in (A.69). For (A.89), we apply either `Fill-Default` or `Fill-Operand`, depending on whether $\left[\,\mathbb{R}?_i\,\right]_{0\le i\le k}\overline{\mathbb{R}}$ is empty. The for-

mer case is trivial. For the latter case, we use `Produce-Tighten`. This means we have to show that σ_0 produces a sequence of alternating grout and nonterminals, such that the nonterminals produce the nonempty entries in $[\,\mathbb{R}?_i\,]_{0\le i\le k}\,\overline{\mathbb{R}}$. We already have the nonterminals for $[\,\mathbb{R}?_i\,]_{0\le i\le k}$; they are $[\,\rho?_i\,]_{0\le i\le k}$ by (A.70). The sorts for $\overline{\mathbb{R}}$ are the sorts they synthesize, with $0, 0$ bounds, they analyzed against these because they are natural.

Now we need to show that σ_0 actually does produce this form. We obtain this by widening to $\top$-$\top$ bounds, applying subsumption, then applying `GInj-Operand`. Now we have to show that each of the nonterminals mentioned above is accessible from the sort of σ_0, and has non-$\bot$ bounds. We obtain non-$\bot$ bounds for the first set, $[\,\rho?_i\,]_{0\le i\le k}$, by rule inversion on the fact that they appear next to grout in (A.69). The second set was constructed to have 0 bounds, which are not $\bot$. We have accessibility from the sort of σ_0 for $[\,\rho?_i\,]_{0\le i\le k}$ by *splicing* (A.69). For accessibility of the root sorts of $\overline{\mathbb{R}}$, we observe that each one is accessible from $\rho?$ by (A.67), which is in turn accessible from the sort of σ_0 by the aforementioned splice.

This completes the slot filling obligation. The final obligation is to show naturality, (A.90). We already have $\overline{\mathbb{R}}$ nat by assuming the original stack configuration is well-formed. For $[\,\rho?_i\,]_{0\le i\le k}$, we rely on the fact that the bounds for each $[\,\rho?_i\,]_{0\le i\le k}$ is non-$\bot$, since they appear next to grout.

□

Appendix B

`tall tylr` Performance

We performed some simple benchmarks to test `tall tylr`'s parsing and insertion edit performance.

B.0.1 Left-to-right parsing performance setup

For the purposes of judging parsing performance with respect to program length, we approximated a naturalistic code example by assembling a base program of 100 lines consisting of all examples from our user study (§5.5), expressed as definitions with a trailing hole. For each trial, we concatenated as many copies of that program as necessary to hit the line total, and then truncated the result to the desired line length. This creates a program which is syntactically correct and complete modulo trailing obligations. This program is then parsed, by splitting into tokens and performing insertion actions for each token, measuring the total time for all insertions.

(The base program consists of 100 lines, 2233 chars, and 1270 tokens, resulting in an average of 12.7 tokens and 22.3 characters per line).

B.0.2 Left-to-right parsing performance results

Parsing results are shown in Fig. B.1. Performance is currently quadratic in (realistic) program length owing to the fact that each let definition imposes an additional level of nesting. We are considering a specific optimization for top-level definition forms in the future to make this linear.

B.0.3 Insertion action performance setup

Taking the base 100-line program from the above task, we deleted 20 single-token terms distributed randomly over the program, leaving 20 operand obligations. For each resulting obligation, we measured the time taken to insert a single character token into the hole, repeating each such insertion 200 times to increase precision.

For each operand obligation, we derived the syntactic nesting depth (number of containing forms) and the total length of the operand sequence (prefix plus suffix) in which the hole is contained, and plotted these versus the time taken per insertion.

B.0.4 Insertion action performance results

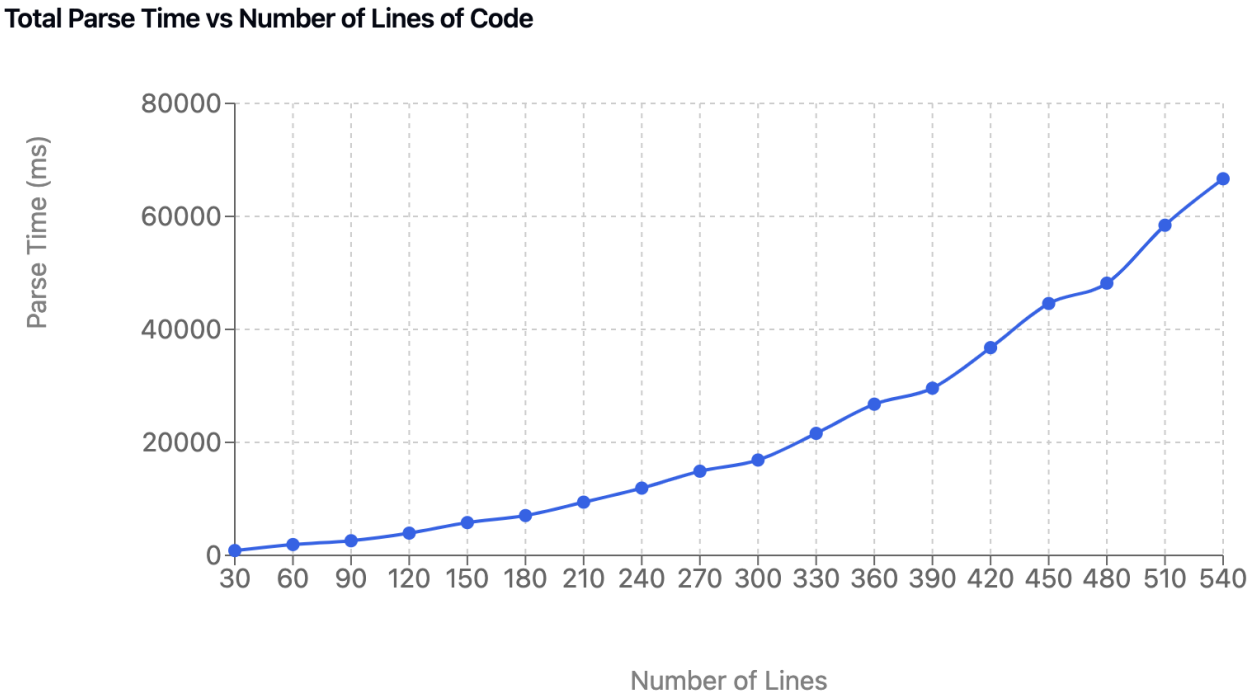


Figure B.1: Time taken to parse syntactically correct and prefix-complete programs across a range of program lengths

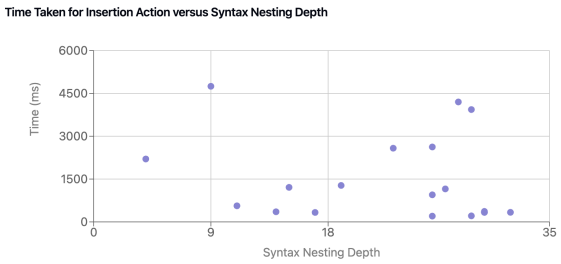


Figure B.2: Time taken to perform 200 single-character token insertions in an operand hole of the specified depth

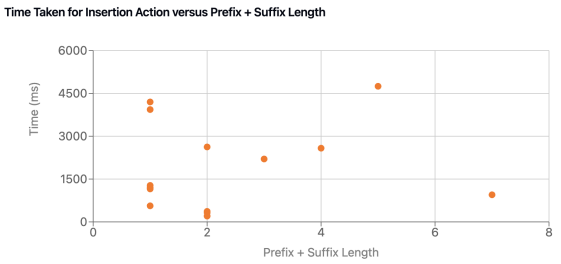


Figure B.3: Time taken to perform 200 single-character token insertions in an operand hole of the specified operand sequence length

Appendix C

Additional Data for §5.5

C.1 More Scenarios

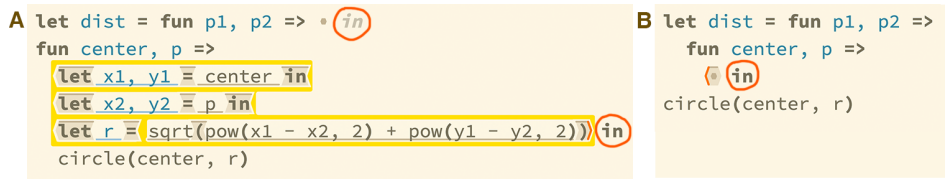


Figure C.1: A participant has stubbed the header for a helper function, and is about to cut some relevant code to paste in the helper. However, they left the `in` delimiter belonging to the helper stub as a ghost, and incidentally omitted an `in` from their selection. On cut, that latter `in` becomes an orphan, which is then matched to the ghost `in`. This has the effect of shunting the existing function literal into the body of the helper.

C.1.1 Spooky action-at-a-distance due to unattended ghosts

Ghost cleanup logic can trigger non-local effects which can be hard to anticipate when the ghost is not directly involved in the current edit. Fig. C.1 illustrates an example from Task 4, where the participant was confused that their cut action seemingly sucked the function literal into the body of the definition, taking them further from their goal.

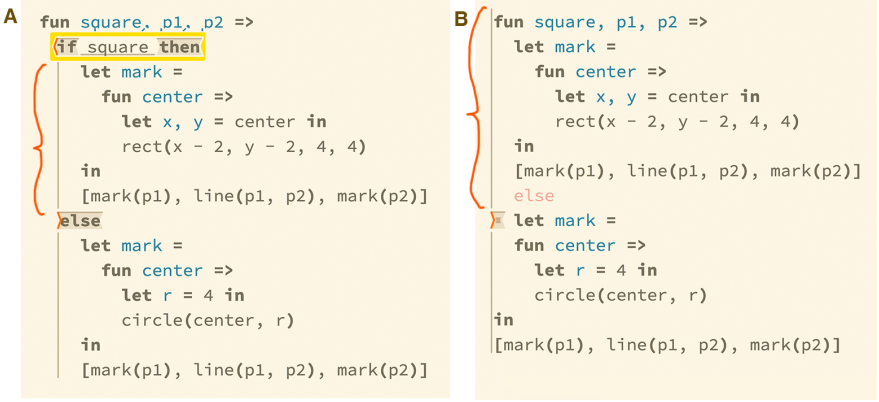


Figure C.2: During Task 6, participants had to push an `if` expression deeper into a function, which they typically approached by cutting the segment highlighted in yellow. This cut leaves behind an unmolded red 'else' delimiter and an infix obligation. Since infix obligations are assigned the loosest precedence, the function literal taking square as an argument is now entirely on the left side of the grout, and the expression on the last line of the program is no longer inside that function literal, resulting in a subtle but substantial change to the program structure.

C.1.2 Infix obligations and unexpected shifts in structure

While participants seemed to accept the notion of infix obligations as missing infix operators, their sudden appearance during the deletion of compound syntactic forms like definitions and conditionals lead to subtle and surprising situations. One of these is illustrated in Fig. C.2. Many participants did not notice the infix obligation in such situations, pushing on with their planned edits, but for those that did this was a source of confusion. Participant **P6** notes: "What I expected to be the case is that this piece of grout is a binary operator on this rect and this list in the body of this let. But that's not even true."

C.2 More Participant Reactions

> **P2:** "It's feeling pretty good. [...] I don't know if I'm just like not thinking of a lot of stuff right now, or if that was just so smooth that it doesn't give me a lot of thoughts"
>
> **P3:** "It takes probably takes more time to develop an intuition behind when to use tylr powers [...] in particular, when use space and start to type something"
>
> **P9:** "It was always clear visually if the editor was on the same page as me."
>
> **P2:** "It's feeling pretty good. [...] I don't know if I'm just like not thinking of a lot of stuff right now, or if that was just so smooth that it doesn't give me a lot of thoughts"
>
> P4: "The grout and ghosts, when they worked, felt pretty seamless"
>
> **P8:** "I always would like placeholder completions until I have a complete expression! They allow me to scaffold until I have a complete expression of code."
>
> **P2:** "I don't have to remember the syntax of the language as much, I just have to remember how to write the first token in a term."

This was particularly felt when the obligations were inserted in the middle of a complex edit:

> **P5:** "The editor sometimes added a lot of holes [...] while I was in the middle of editing an expression, which I instinctively tried to delete."
>
> **P7:** "I was actually surprised. I thought it was going to break. But then it worked. So, I'm pleasantly surprised, I'll say for that one."
>
> **P4:** "So the the fact that these ghosts and grout are in my way and I can't get rid of them is super annoying."
>
> **P3:** "It takes probably takes more time to develop an intuition behind when to use tylr powers [...] in particular, when use space and start to type something"