



Syntactic Completions with Material Obligations*

DAVID MOON, University of Michigan, USA

ANDREW BLINN, University of Michigan, USA

THOMAS J. PORTER, University of Michigan, USA

CYRUS OMAR, University of Michigan, USA

Code editors provide essential services that help developers understand, navigate, and modify programs. However, these services often fail in the presence of syntax errors. Existing syntax error recovery techniques, like panic mode and multi-option repairs, are either too coarse, e.g. in deleting large swathes of code, or lead to a proliferation of possible completions. This paper introduces *tall tylr*, an error-handling parser and editor generator that completes malformed code with *syntactic obligations* that abstract over many possible completions. These obligations generalize the familiar notion of holes in structure editors to cover missing operands, operators, delimiters, and sort transitions.

tall tylr is backed by a novel theory of tile-based parsing, conceptually organized around a *molder* that turns tokens into tiles and a *melder* that completes and parses tiles into terms using an error-handling generalization of operator-precedence parsing. We formalize melding as a parsing calculus, *meldr*, that completes input tiles with additional obligations such that it can be parsed into a well-formed term, with success guaranteed over all inputs. We further describe how *tall tylr* implements molding and completion-ranking using the principle of *minimizing obligations*.

Obligations offer a useful way to scaffold internal program representations, but in *tall tylr* we go further to investigate the potential of *materializing* these obligations visually to the programmer. We conduct a user study to evaluate the extent to which an editor like *tall tylr* that materializes syntactic obligations might be usable and useful, finding both points of positivity and interesting new avenues for future work.

CCS Concepts: • **Software and its engineering** → *General programming languages*.

Additional Key Words and Phrases: structure editing, error-handling parsing, operator precedence

ACM Reference Format:

David Moon, Andrew Blinn, Thomas J. Porter, and Cyrus Omar. 2025. Syntactic Completions with Material Obligations. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 404 (October 2025), 27 pages. <https://doi.org/10.1145/3763182>

1 Introduction

Programmers rely on editor services like syntax highlighting, code completion, and go-to-definition for help with comprehending, modifying, and navigating programs in various stages of completion. These services require analyzing the syntactic structure of the program being edited. The problem is that, for typical grammars, most textual edit states do not successfully parse [26, 28]. For example, consider the malformed program in Fig. 1A, written in an OCaml-like expression language (that uses parentheses for function application, rather than spaces). A naively implemented parser would simply stop upon encountering the first unexpected token, *p2*, at the end of the first line, leaving

*A version of this paper with errata and a complete appendix is available at <https://arxiv.org/abs/2508.16848>.

Authors' Contact Information: David Moon, University of Michigan, Ann Arbor, USA, dmoo@umich.edu; Andrew Blinn, University of Michigan, Ann Arbor, USA, blinnand@umich.edu; Thomas J. Porter, University of Michigan, Ann Arbor, USA, thomasjp@umich.edu; Cyrus Omar, University of Michigan, Ann Arbor, USA, comar@umich.edu.



This work is licensed under a Creative Commons Attribution 4.0 International License.

© 2025 Copyright held by the owner/author(s).

ACM 2475-1421/2025/10-ART404

<https://doi.org/10.1145/3763182>

```

fun (p1 p2
  let (x1: Num, y1: Num = p1
  let ( : Num, y2: ) = in
  let -> = fun x => x * x in
  sqrt(sqr(x1 - ) + sqr(y1 - y2))

fun (p1 p2
  let (x1: Num, y1: Num = p1
  let ( : Num, y2: ) = in
  let -> = fun x => x * x in
  sqrt(sqr(x1 - ) + sqr(y1 - y2))

fun (p1 p2 =>
fun (p1 p2 =>
fun (p1 p2 =>
fun (p1, p2) =>
fun (p1::p2) =>
fun (p1 | p2) =>
fun (p1 as p2) =>
fun (p1, p2) =>
...

```

Fig. 1. (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.

the program unparsed and unanalyzable by downstream editor services. To address this problem, modern parsers attempt to recover from and continue parsing around malformed syntax, but often leave much to be desired. Let us consider how contemporary methods might recover from the unexpected token `p2`.

A simple recovery method known as “panic mode” [3, 16] drops tokens heuristically around the error until parsing can resume—in this case, as shown in Fig. 1B, 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, leaving the programmer with limited or incorrect assistance where they may need it most [10]. For example, the dropped lines in Fig. 1B 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 error recovery methods consider a range of possible repairs around the error location. Fig. 1C shows some possible repairs for the first line of code in Fig. 1A. The first three repairs show how this method reduces dropped input (i.e. deleted tokens) 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 a combinatorial explosion as additional insertions, deletions, and larger repair windows are considered [7, 10]. To select from these possible repairs, most parsers use *ad hoc* heuristics [11, 14]. The heuristically chosen repair is generally not communicated to the programmer, leaving them only indirect clues in the behavior of downstream editor services. Having the programmer disambiguate between possible repairs interactively can cause information overload due to the number of possible completions [20].

This paper 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 can be used to determine 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 paper we go further to investigate the potential of *materializing* these obligations visually to the programmer.



The screenshot in Fig. 2 shows how *tall tylr* repairs the program from Fig. 1A. 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

```

fun (p1 ■ p2) =>
  (let (x1: Num, y1: Num) = p1 in
    let (•: Num, y2: •) = • in
    let • • -> • = fun x => x * x in
    sqrt(sqr(x1 - •) + sqr(y1 - y2))

```

Fig. 2. The syntactic completion of the program from Fig. 1 in *tall tylr*, our tile-based editor.

placing the cursor on these faded 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 §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. 2 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. 2 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 [6, 22].

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 [13]. OP parsing enjoys the *bounded context* property [16] 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 §3, we precisely specify melding as a parsing calculus called *meldr*. In addition to precedence walks, we describe in §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 §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 §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, 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.

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 [27]. 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.

2.1 Operand Obligations



Fig. 3. Basic expression insertion in tall tylr, demonstrating operand obligations and term decorations.

We begin with an empty editor buffer in Fig. 3(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. 3(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. 3(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. 3(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 `3`, this automatically inserted space is “consumed” rather than causing the insertion of a second space, as shown in Fig. 3(e).

Finally, we continue typing as shown, resulting in Fig. 3(f). Operator sequences are parsed according to Hazel’s precedences and associativities. Notice in both Fig. 3(d) and Fig. 3(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.

2.2 Infix Obligations

Starting from the editor state in Fig. 4(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. 4(b). Infix obligations have the lowest precedence.

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.



Fig. 4. Adjacent operands are connected by infix obligations in tall tylr.

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 molds. For example, in Fig. 5(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.

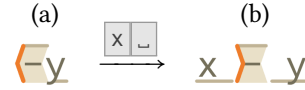


Fig. 5. The minus sign has multiple molds. The mold is chosen to minimize obligations.

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(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.

2.4 Ghost Obligations



Fig. 6. 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. 6(a), we can insert a `let` expression by typing `let`. This causes insertion of *ghost obligations*, shown in gray in Fig. 6(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. 6(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 based primarily on newline placement in the buffer, summarized by the example in Fig. 7. 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. 7(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 sort as shown in Fig. 7(c-d).

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. 8(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. 8(b). Note that tall tylr automatically manages indentation.

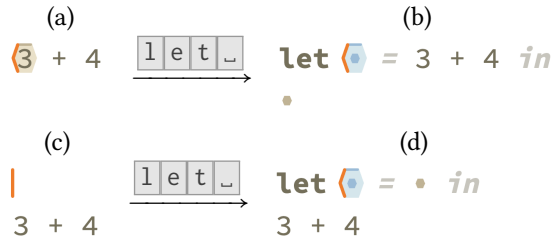


Fig. 7. Ghost obligation placement is chosen heuristically, here based on newline locations.

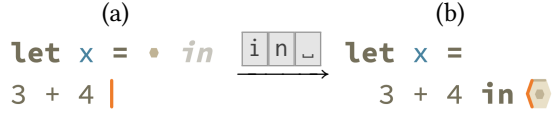


Fig. 8. Ghost obligations can be ignored and are cleaned up if entered elsewhere.

2.5 Sort Transition Obligations

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. 9(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. 9(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.

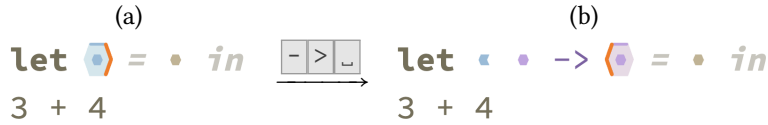


Fig. 9. Sort transition obligations are needed when entering forms that are not sort-correct.

2.6 Unmolded Tokens

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



Fig. 10. Unrecognized tokens are left unmolded, and therefore cannot fulfill obligations.

and ignores it as shown in Fig. 10. Notice that no matter where the ! token appears, the operand obligation remains unfilled.

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* (§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 3.4).

As a substrate for these ideas, we generalize and unify two prior accounts of operator precedence: Aasa’s semantics for precedence annotations in grammars [1] and Floyd’s seminal introduction of operator-precedence parsing (OP parsing) [13]. 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 §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 §3.1.4 that Floyd’s precedence relations derived from \mathcal{H} cohere as expected with the annotations in \mathcal{G} (Theorem 3.1).

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 §3.2, we present our error-handling variation in §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 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 §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 3.4).

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

option	$\alpha?$	$::=$	$\circ \mid \bullet\alpha$
sequence	$\bar{\alpha}$	$::=$	$\cdot \mid \alpha\bar{\alpha}$

3.1 Elaborating Precedence Annotations

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

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

Fig. 11. Syntax of precedence-bounded grammars

terminal τ	$::= \prec \mid t \mid \succ$
nonterminal ρ, σ	$::= p_s^q$
symbol χ	$::= \tau \mid \sigma$
CFG \mathcal{H}	$\in \{\sigma \Rightarrow \bar{\chi}\}$

Fig. 12. Syntax of elaborated context-free grammars

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.

$$\begin{aligned}
 \mathcal{T}_{\text{HZ}} = & \left\{ \begin{array}{l} \text{let} \langle \rangle \text{in} \dots \langle \rangle \langle \rangle \langle \rangle \\ \langle \rangle \text{var} \langle \rangle \langle \rangle \langle \rangle \\ \text{Num} \langle \rangle \langle \rangle \langle \rangle \end{array} \right\} \\
 \sqcup & \left\{ \begin{array}{l} \langle \rangle \text{var} \langle \rangle \langle \rangle \langle \rangle \\ \text{Num} \langle \rangle \langle \rangle \langle \rangle \end{array} \right\} \\
 \sqcup & \left\{ \begin{array}{l} \langle \rangle \text{var} \langle \rangle \langle \rangle \langle \rangle \\ \text{Num} \langle \rangle \langle \rangle \langle \rangle \end{array} \right\} \\
 \mathcal{S}_{\text{HZ}} = & \{\hat{E}, P, T\}
 \end{aligned}$$

$$\mathcal{G}_{\text{HZ}} = \left\{ \begin{array}{l} E \mapsto 0 \mapsto \langle \text{let} \cdot P \cdot \langle \rangle \cdot E \cdot \text{in} \cdot E \rangle \\ 1^> \mapsto E \cdot \langle \rangle \mid \langle \rangle \cdot E \\ 2^> \mapsto E \cdot \langle \rangle \mid \langle \rangle \cdot E \\ 3 \mapsto \langle \text{num} \rangle \mid \langle \text{var} \rangle \\ \quad \mid \langle \langle \cdot E \cdot \langle \rangle \cdot E \rangle^* \cdot \rangle \\ P \mapsto 0 \mapsto P \cdot \langle \rangle \cdot T \\ 1 \mapsto \langle \text{var} \rangle \mid \langle \langle \cdot P \cdot \langle \rangle \cdot P \rangle^* \cdot \rangle \\ T \mapsto 0 \mapsto \langle \text{Num} \rangle \mid \langle \langle \cdot T \cdot \langle \rangle \cdot T \rangle^* \cdot \rangle \end{array} \right\}$$

Fig. 13. A PBG \mathcal{G}_{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.

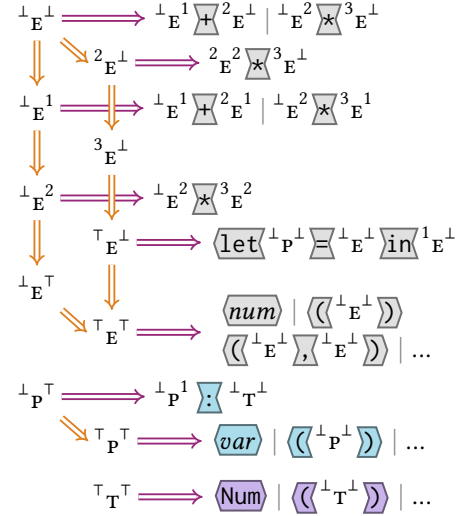


Fig. 14. An excerpt of the CFG \mathcal{H}_{HZ} elaborated (Fig. 17) from \mathcal{G}_{HZ} (Fig. 13). The production rules are arranged and color-coded by whether each is elaborated by **reduction** or by **tightening**.

The syntax of PBGs is given in Fig. 11. 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 $\mathcal{P} = \mathbb{N} \sqcup \{\perp, \top\}$ includes the natural numbers \mathbb{N} for precedence levels assigned in \mathcal{G} as well as minimum \perp and maximum \top precedence levels that are reserved for internal use. We further assume that \mathcal{P} is equipped with ordering relations $<_s, >_s$ that abstract the details of associativity for each sort $s \in \mathcal{S}$. For example, $5 <_E 5$ 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 $p <_s \top >_s p$ and $p >_s \perp <_s p$ for all other $p \in \mathbb{N}$.

Fig. 13 gives a concrete PBG \mathcal{G}_{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 $\otimes \in \{<_s, >_s\}$ that applies to the reflexive pair $p \otimes p$, if any—in this case, levels $1^>$ and $2^>$ of sort E are marked as left-associative, i.e. $1 >_E 1$

and $2 >_E 2$. 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 $\llbracket g \rrbracket$ of matching symbol strings \bar{x} is defined as follows:

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

A \mathcal{G} -form is a symbol string $\bar{x} \in \llbracket \mathcal{G}(s, p) \rrbracket$ 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* [13]:

ASSUMPTION 1 (OPERATOR FORM). *There exist no sorts $s_L, s_R \in \mathcal{S}$ and regex $\mathcal{G}(s, p)$ such that $\dots s_L s_R \dots \in \llbracket \mathcal{G}(s, p) \rrbracket$.*

In other words, every \mathcal{G} -form may be written in the form $s?_0 [t_i s?_{i+1}]_{0 \leq i \leq k}$. Greibach [15] showed that every CFG can be normalized to a strongly equivalent one in operator form.

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. 12. An elaborated CFG \mathcal{H} is a (possibly infinite) set of production rules $\sigma \Rightarrow \bar{\chi}$, each mapping a nonterminal σ to a finite sequence $\bar{\chi}$ of symbols—we will refer to σ and $\bar{\chi}$ 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. 14 shows an excerpt of the CFG \mathcal{H}_{HZ} elaborated from \mathcal{G}_{HZ} , the process of which we discuss in §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 $\sigma \Rightarrow \bar{\chi}$, we say that the symbol string $\bar{\chi}_L \sigma \bar{\chi}_R$ *produces* the string $\bar{\chi}_L \bar{\chi} \bar{\chi}_R$, written $\bar{\chi}_L \sigma \bar{\chi}_R \Rightarrow \bar{\chi}_L \bar{\chi} \bar{\chi}_R$ reusing the production rule syntax. A production sequence $\bar{\chi}_0 \Rightarrow \bar{\chi}_1 \Rightarrow \dots$ from the designated start string $\bar{\chi}_0 = \prec^\perp s^\perp \succ$ is called a *derivation*; the language of a CFG collects all of its derivable strings.

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 [13] does not surface the operator index $\rho?$, whose omission we will later show contributes to an unsound parsing method (§3.2.2), and whose use in our resolution we describe in §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 explicitly $\tau_L \odot_\circ \tau_R$.

The comparison operator $\odot \in \{\prec, \doteq, \succ\}$ indicates in what relative order the neighbors τ_L, τ_R were first produced in the derivation. Fig. 16 shows an excerpt of the precedence comparisons for \mathcal{H}_{HZ} (Fig. 14). The derivation ${}^1 E^\perp \Rightarrow {}^1 E^1 \boxplus^2 E^\perp \Rightarrow {}^1 E^1 \boxplus^2 E^2 \boxtimes^3 E^\perp$ tells us $\boxplus < \boxtimes$ (“ \boxplus binds less tightly than \boxtimes ”) since \boxplus is produced before its neighbor \boxtimes . Meanwhile, the derivation ${}^1 E^\perp \Rightarrow {}^1 E^\top \Rightarrow \llbracket {}^1 E^\perp \rrbracket$ tells us $\llbracket \doteq \rrbracket$ (“ \llbracket matches \rrbracket ”) since neighbors \llbracket and \rrbracket 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 $\tau_L < \tau_R$ to be equivalent to $\tau_R > \tau_L$, nor for \doteq to be symmetric.

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 3.1](#)).

THEOREM 3.1 (ANNOTATION-COMPARISON COHERENCE). *For all sorts s , precedence levels p_L, p_R , and tiles t_L, t_R such that $\dots t_L s \in \llbracket \mathcal{G}(s, p_L) \rrbracket$ and $st_R \dots \in \llbracket \mathcal{G}(s, p_R) \rrbracket$, the following equivalences hold:*

$$t_L < t_R \iff p_L <_s p_R \quad t_L > t_R \iff p_L >_s p_R$$

To motivate our design ([Fig. 17](#)), 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} :

$$\mathcal{H}_0 \triangleq \{s \Rightarrow \bar{x} \mid \bar{x} \in \llbracket \mathcal{G}(s, p) \rrbracket, p \in \mathcal{P}, s \in \mathcal{S}\}$$

\mathcal{H}_0 allows for problematic derivations like $E \Rightarrow \langle E \star E \rangle \Rightarrow \langle \langle E \star E \rangle \star E \rangle$, problematic because it witnesses the unwanted comparison $\star > \star$ (“ \star binds more tightly than \star ”).

A better approach—similar in effect to that of Danielsson and Norell [8] and of Klint and Visser [19]—would use singly-bounded nonterminals s^p and limit their productions to \mathcal{G} -forms of equal or stronger precedence:

$$\mathcal{H}_1 \triangleq \{s^p \Rightarrow [\bar{x}]^p \mid \bar{x} \in \llbracket \mathcal{G}(s, q) \rrbracket, p \leq_s q, s \in \mathcal{S}\}$$

where $[\bar{x}]^p$ lifts each sort symbol $s \in \bar{x}$ to some suitably bounded nonterminal.

- (a) $E^1 \Rightarrow \langle E^2 \star E^3 \rangle \Rightarrow \langle \langle E^1 \star E^2 \rangle \star E^3 \rangle$
- (b) $E^1 \Rightarrow \langle E^2 \star E^3 \rangle \Rightarrow \langle \langle \text{let } p^1 \text{ in } E^1 \text{ in } E^0 \rangle \star E^3 \rangle$
- (c) $E^1 \Rightarrow \langle E^2 \star E^3 \rangle \Rightarrow \langle E^2 \star \langle \text{let } p^1 \text{ in } E^1 \text{ in } E^0 \rangle \rangle$

This approach properly rules out unwanted derivations on the left like (a) (for witnessing $\star > \star$) and (b) ($\text{in} > \star$), since the left argument E^2 of \star cannot produce the \star - and let -forms of weaker precedence levels 1 and 0. However, \mathcal{H}_1 is overly conservative: it also rules out acceptable derivations like (c) ($\star < \text{let}$). Ultimately the purpose of precedence annotations is to resolve choices between

comparison $\odot ::= < \mid \doteq \mid >$

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

Prec-LT
 $\tau_L \sqsupset \sigma \quad \sigma \Rightarrow^* \rho? \tau_R \dots$
 $\tau_L <_{\rho?} \tau_R$

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

Prec-GT
 $\sigma \Rightarrow^* \dots \tau_L \rho? \quad \sigma \sqsupset \tau_R$
 $\tau_L >_{\rho?} \tau_R$

$\tau_L \backslash \tau_R$	let	in	\star	$\langle \rangle$	num	Num
let	\doteq	$<$	$<$	$<$	$<$	$<$
in	$<$	\doteq	$<$	$<$	$<$	$<$
\star	$<$	$<$	\doteq	$<$	$<$	$<$
$\langle \rangle$	$<$	$<$	$<$	\doteq	$<$	$<$
num	$<$	$<$	$<$	$<$	\doteq	$<$
Num	$<$	$<$	$<$	$<$	$<$	\doteq

Fig. 15. Precedence comparisons

Fig. 16. An excerpt of precedence comparisons $\tau_L \odot \tau_R$ for \mathcal{H}_{HZ} ([Fig. 14](#))

$\boxed{\chi \sim x}$	CFG symbol χ is consistent with PBG symbol x	$\boxed{\sigma \Rightarrow \bar{\chi}}$	Nonterminal σ produces symbols $\bar{\chi}$
$t \sim t$	$p_s^q \sim s$	Produce-Subsume $\frac{\sigma \Leftarrow \bar{\chi}}{\sigma \Rightarrow \bar{\chi}}$	Produce-Tighten $\frac{p \preceq_s m \quad n \succeq_s q}{p_s^q \Rightarrow m_s^n}$
$\boxed{\sigma \Leftarrow \bar{\chi}}$	Symbol sequence $\bar{\chi}$ reduces to nonterminal σ		
PElab-Operand ($k \geq 0$) $\frac{\begin{array}{c} [x_i]_{0 \leq i \leq k} \in \llbracket \mathcal{G}(s, \square) \rrbracket \quad [\chi_i \sim x_i]_{0 \leq i \leq k} \\ x_0 \neq s \neq x_k \end{array}}{\top s^\top \Leftarrow [\chi_i]_{0 \leq i \leq k}}$		PElab-Infix ($k \geq 0$) $\frac{\begin{array}{c} [x_i]_{0 \leq i \leq k} \in \llbracket \mathcal{G}(s, m) \rrbracket \quad [\chi_i \sim x_i]_{0 \leq i \leq k} \\ \chi_0 = p_s^{n_L} \quad n_R s^q = \chi_k \\ n_L >_s m <_s n_R \end{array}}{\min(p, m)_s^{\min(m, q)} \Leftarrow [\chi_i]_{0 \leq i \leq k}}$	
PElab-Prefix ($k \geq 1$) $\frac{\begin{array}{c} [x_i]_{0 \leq i \leq k} \in \llbracket \mathcal{G}(s, m) \rrbracket \quad [\chi_i \sim x_i]_{0 \leq i \leq k} \\ x_0 \neq s \quad n_R s^q = \chi_k \\ m <_s n_R \end{array}}{\top s^{\min(m, q)} \Leftarrow [\chi_i]_{0 \leq i \leq k}}$		PElab-Postfix ($k \geq 1$) $\frac{\begin{array}{c} [x_i]_{0 \leq i \leq k} \in \llbracket \mathcal{G}(s, m) \rrbracket \quad [\chi_i \sim x_i]_{0 \leq i \leq k} \\ \chi_0 = p_s^{n_L} \quad s \neq x_k \\ n_L >_s m \end{array}}{\min(p, m)_s^\top \Leftarrow [\chi_i]_{0 \leq i \leq k}}$	

Fig. 17. Bidirectional elaboration of production $\sigma \Rightarrow \bar{\chi}$ and reduction $\sigma \Leftarrow \bar{\chi}$ rules for CFG \mathcal{H} from PBG \mathcal{G}

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 (\boxplus and \boxminus) before the right (\boxtimes) over the reduced children \mathbf{e}^2 and \mathbf{e}^0 , respectively. On the other hand, (c) has no viable alternative reduction order, since $\overline{\text{let}}$ cannot parent a child to its left. In such cases, the precedence annotations need not be consulted.

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. 17 is organized accordingly. A production rule $\sigma \Rightarrow \bar{\chi}$ is introduced either by *tightening* the bounds on σ (Produce-Tighten) or by *subsuming* the corresponding reduction $\sigma \Leftarrow \bar{\chi}$ (Produce-Subsume), as illustrated for \mathcal{H}_{HZ} in Fig. 14.

$$\frac{\begin{array}{c} \overline{\text{let}} \boxplus \mathbf{e} \boxminus \mathbf{e} \in \llbracket \mathcal{G}(\mathbf{e}, 0) \rrbracket \\ \overline{\text{let}} \neq \mathbf{e} \quad 0 <_{\mathbf{e}} 1 \quad 1_{\mathbf{e}}^\perp \sim \mathbf{e} \end{array}}{\top_{\mathbf{e}}^{\min(0, \perp)} \Leftarrow \overline{\text{let}}^\perp \boxplus^\perp \boxminus^\perp \mathbf{e}^\perp}$$

Meanwhile, a reduction rule $\sigma \Leftarrow \bar{\chi}$ synthesizes the tightest possible bounds on σ that can accommodate $\bar{\chi}$. These correspond to Aasa’s notion of *precedence weights* [1] 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}_{HZ} , the prefix-shaped $\overline{\text{let}}$ -form synthesizes left weight \top and right weight $\min(0, \perp)$, where the latter folds in the annotated level 0 into the subweight \perp (underlined to distinguish it from other \perp values in the derivation) already computed for the rightmost child $1_{\mathbf{e}}^\perp$. Our bidirectional presentation

node $\mathbb{X} ::= \tau \mid \mathbb{S}$
 term $\mathbb{R}, \mathbb{S} ::= \{\overline{\mathbb{X}}\}$

Fig. 18. Syntax of terms

leq $\leq ::= < \mid \doteq < \mid \odot$
 stack $\mathbb{K} ::= \prec \mid \mathbb{K} \leq_{\mathbb{S}^?} \tau$

$\text{hd}(\prec) = \prec$
 $\text{hd}(\mathbb{K} \leq_{\mathbb{S}^?} \tau) = \tau$

Fig. 19. Syntax of stacks

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

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

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

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

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

WFStack-Nil $\frac{}{\prec \text{ wf}}$

Reduce-Term ($k \geq 0$)

$$\frac{\begin{array}{l} \sigma \Leftarrow [\chi_i]_{0 \leq i \leq k} \\ [\chi_i \Leftarrow \mathbb{X}_i]_{0 \leq i \leq k} \end{array}}{\sigma \Leftarrow \{[\mathbb{X}_i]_{0 \leq i \leq k}\}}$$

Produce-Term ($k \geq 0$)

$$\frac{\begin{array}{l} \sigma \Rightarrow [\chi_i]_{0 \leq i \leq k} \\ [\chi_i \Rightarrow \mathbb{X}_i]_{0 \leq i \leq k} \end{array}}{\sigma \Rightarrow \{[\mathbb{X}_i]_{0 \leq i \leq k}\}}$$

WFStack-Cons

$$\frac{\begin{array}{l} \mathbb{K} \text{ wf} \quad \sigma^? \Rightarrow \mathbb{S}^? \\ \text{hd}(\mathbb{K}) \leq_{\sigma^?} \tau \end{array}}{\mathbb{K} \leq_{\mathbb{S}^?} \tau \text{ wf}}$$

Fig. 20. A node is well-formed if it is reducible to or producible from a symbol. Fig. 21. Well-formed stacks

reorganizes and generalizes Aasa's to multi-sorted grammars of arbitrary mixfix forms, which we discuss further in §6.

3.2 OP Parsing Errors

In this section, we review Floyd's original method for operator-precedence (OP) parsing [13]. To motivate our error-handling generalization in §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. 18 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. 20.

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. 19 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 $\mathbb{K} \leq_{\mathbb{S}^?} \tau$, linking a token τ to the rest of the stack \mathbb{K} with two pieces of information: a comparison operator \leq 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. 21, when each of its links $\tau_L \leq_{\mathbb{S}^?} \tau_R$ reflects a valid precedence relation $\tau_L \leq_{\mathbb{S}^?} \tau_R$ such that $\sigma^? \Rightarrow \mathbb{S}^?$. For brevity, we will call optional nonterminals *slots* and optional terms *cells*.

The *height* of a stack is the number of \prec -operators it contains. We can decompose any stack of height h into a sequence of h height-1 stacks, each of the form $\tau \leq_{\mathbb{S}^?_0} \tau_0 [\doteq_{\mathbb{S}^?_i} \tau_i]_{0 < i \leq k} (k \geq 0)$. We may interpret each such stack as a term under construction $\leq_{\mathbb{S}^?_0} \tau_0 [\doteq_{\mathbb{S}^?_i} \tau_i]_{0 < i \leq k}$ 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. 22 shows Floyd's original algorithm, presented here as a push operation $\mathbb{K} \xleftarrow{\mathbb{R}^?} \tau = \mathbb{K}'$ 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. 23 and Fig. 24 illustrate concrete OP parsing traces for \mathcal{H}_{HZ} using the precedence table in Fig. 16—each colored box applies one of the rules in Fig. 22, 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 $\text{hd}(\mathbb{K})$ precedence-compares with the pushed token τ to decide whether to **Shift** or **Reduce**. If $\text{hd}(\mathbb{K}) \leq \tau$, then the parser shifts τ onto \mathbb{K} and “finalizes” the reduction $\mathbb{R}?$ between them. Else, if $\text{hd}(\mathbb{K}) > \tau$, and \mathbb{K} has height $h \geq 1$, then the parser has identified its next *handle* (i.e. reduction target) of the form $\text{hd}(\mathbb{K}) \leq_{\mathbb{R}^?_0} \tau_0 \left[\dot{\leq}_{\mathbb{R}^?_i} \tau_i \right]_{0 \leq i \leq k} \dot{\geq}_{\mathbb{R}^?_{k+1}} \tau$ where $\text{hd}(\mathbb{K})$ and τ delimit the handle $\{\mathbb{R}^?_0[\tau_i \mathbb{R}^?_{i+1}]_{0 \leq i \leq k}\}$ to be reduced and propagated up the stack.

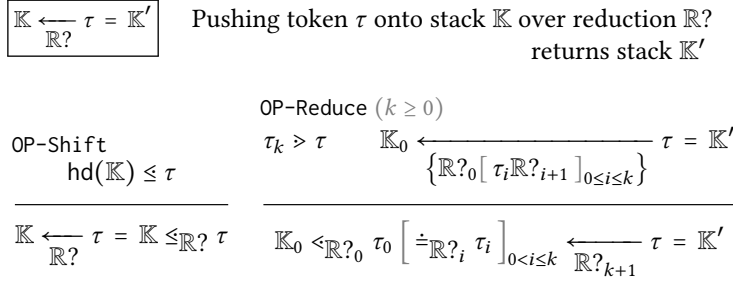


Fig. 22. OP parsing

Let us consider the ways this algorithm can fail.

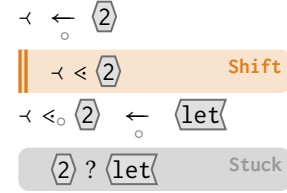
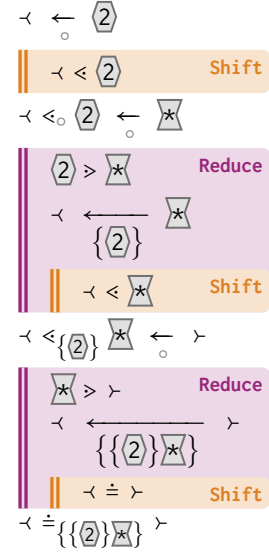
3.2.1 Stuck. Like with most (non-error-handling) methods, a typical OP parser will easily get stuck. This occurs when the stack head and pushed token share no precedence relation, like $\langle 2 \rangle$ and $\langle \text{let} \rangle$ in Fig. 23.

3.2.2 Invalid Reduction. OP parsing is unsound, meaning it can produce grammatically invalid reductions. Recall from §3.1.3 that each precedence comparison $\tau_L \odot_{\rho^?} \tau_R$ means there exists a derivable string of the form $\dots \tau_L \rho^? \tau_R \dots$. Floyd’s original definition of precedence 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. 24, the parser identifies the handle pattern $\prec \langle \star \rangle \succ$ and proceeds blindly to reduce $\{\{\langle 2 \rangle\} \star\}$ without checking that the initial reduction \circ is a valid right-argument to \star .

3.2.3 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.

$\hat{\tau} \rightarrow \$x\$ | t$
 $x \rightarrow \{t\} | x$

Ideally prefix-validity would be implied by stack well-formedness (Fig. 21), but this is not always the case for an OP parser depending on the grammar. Consider the small grammar on the left, which produces strings like $t, \$x\$, \$\{t\}\$, \$\{\$x\}\$,$ etc. Generated from this grammar are the precedence comparisons $\prec \langle \$ \rangle$ and $\dot{=} \$$, so Floyd’s parser would happily ingest the tokens $\$x\$x\$$ and organize them into the stack

Fig. 23. An OP parsing trace for \mathcal{H}_{HZ} (Fig. 16) that gets stuck trying to compare neighbors $\langle 2 \rangle$ and $\langle \text{let} \rangle$ Fig. 24. A valid OP parsing trace for \mathcal{H}_{HZ} that returns the invalid term $\{\{\langle 2 \rangle\} \star\}$

$\rightarrow \prec_0$. $\$ \doteq_{\bullet\{x\}} \$ \doteq_{\bullet\{x\}} \$$, which is well-formed but prefix-invalid. The issue here is the reuse of $\$$ as both opener and closer in the rule $\hat{\tau} \rightarrow \$x\$$. Distinguishing between these two uses would require stack-level analyses out of scope of the local pairwise precedence comparisons.

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 (§3.3.1) and transformations (§3.3.2) of the language grammar; others involve generalizing Floyd’s algorithm (§3.3.3) to incorporate completion-based repairs and to restore soundness by making use of our nonterminal-enriched precedence comparisons.

3.3.1 Molding Tiles. To secure the valid prefix property (§3.2.3), 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 $(\dots t \dots \in \llbracket \mathcal{G}(s, p) \rrbracket$ and $\dots t \dots \in \llbracket \mathcal{G}(r, q) \rrbracket$ 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 $\tau \prec_{\rho?_0} t_0 \stackrel{\circ}{=} \rho?_i t_i \Big]_{0 \leq i \leq k}$ 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 3.2 (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 $\tau \triangleleft_{\rho?_0} t_0 \stackrel{\triangle}{=} [\rho?_i \ t_i]_{0 < i \leq k} \sqcap \rho?_{k+1}$ 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 $\tau \sqcap \sigma_\tau \triangleleft^* \sigma \Rightarrow \rho?_0 [t_i \ \rho?_{i+1}]_{0 \leq i \leq \ell}$.*

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}_{HZ} (Fig. 13) 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 §4, where we describe how tall tylr chooses between multiple possible molds for a textual token.

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. 24, the reduction $\{\{\boxed{2}\} \star\}$ in the last **Reduce** step is ill-formed because there is no term where one is expected as the right multiplicand; in Fig. 23, the parser gets stuck on neighbors $\boxed{2}$ and $\boxed{\text{let}}$ 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. 9 between the $\boxed{\text{let}}$ -delimiter expecting a pattern and the $\boxed{\rightarrow}$ -term providing a type.

Fig. 25 shows how we materialize these inconsistencies as *grout* forms injected into the elaborated grammar, extending our definition in Fig. 17, while Fig. 26 shows an excerpt of the grout forms injected into \mathcal{H}_{HZ} . Every sort acquires the form $\top s^\top \Rightarrow \circ^s$, injected via rule GInj-Hole, consisting of a single convex grout terminal \circ^s that stands in for missing terms of sort s .

Grout terminals also come in prefix \mathfrak{A} , postfix \mathfrak{B} , and infix \mathfrak{X} 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 α^s or postfix \mathfrak{D}^s grout, respectively. The left (right) bookend is optional when the exposed nonterminal is a leftmost (rightmost) descendant of the unbounded sort ${}^\perp s^\perp$. For example, Fig. 26 includes the grout production ${}^\top P^\perp \Rightarrow \alpha^\perp T^\perp$ because of the P-sorted form ${}^\perp P^\perp \boxtimes {}^\perp T^\perp$ in \mathcal{H}_{HZ} (Fig. 14), where ${}^\perp T^\perp$ is the rightmost descendant. On the other hand, there is no E-sorted form with ${}^\perp P^\perp$ as its leftmost or rightmost descendant, so ${}^\perp P^\perp$ can only appear in the E-sorted grout forms that buffer it on both sides (e.g. $\alpha^\perp P^\perp \mathfrak{D}$).

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 $\gamma^s \prec t$ 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.

3.3.3 Parsing with meldr. Fig. 29 gives the rules for parsing with meldr, whose notable features we will illustrate through several examples.

Fig. 30 illustrates how meldr directly generalizes the standard non-error-handling algorithm (Fig. 22). The main difference is the new *fill* operation, defined in Fig. 27, invoked in Fig. 30 as $\cdot \curvearrowright \circ_{\text{slot}} = \circ_{\text{cell}}$ in **Reduce** and $\{\langle \mathbf{x} \rangle\} \curvearrowright {}^\perp P^\perp = \{\langle \mathbf{x} \rangle\}$ in **Shift**. Filling is responsible for assigning accumulated reductions to grammatically appropriate slots, now exposed as operator indices in

$$\begin{array}{c}
 \boxed{\sigma \Leftarrow \overline{\chi}} \quad \overline{\chi} \text{ reduces to } \sigma \text{ in grout-injected } \mathcal{G} \\
 \text{grout } \gamma ::= \circ \mid \alpha \mid \mathfrak{D} \mid \mathfrak{X} \\
 \text{terminal } \tau ::= \dots \mid \gamma^s
 \end{array}$$

$$\begin{array}{c}
 \text{GInj-Hole} \\
 \dots \quad \frac{}{{}^\top s^\top \Leftarrow \alpha^s} \quad \sigma ! s = \begin{cases} \top & \text{if } \sigma \not\sim s \\ \perp & \text{if } \sigma \sim s \end{cases} \quad \langle p r^q \rangle_s^m = \begin{cases} \max(m, p) r^{\max(q, m)} & \text{if } r = s \\ p r^q & \text{if } r \neq s \end{cases}
 \end{array}$$

$$\begin{array}{c}
 \text{GInj-Operand } (k \geq 0) \\
 \frac{[{}^\perp s^\perp \Rightarrow^* \dots \rho_i \dots]_{0 \leq i \leq k}}{{}^\top s^\top \Leftarrow \alpha^s \langle \rho_0 \rangle_s^0 [\mathfrak{X}^s \langle \rho_i \rangle_s^0]_{0 \leq i \leq k} \mathfrak{D}^s}
 \end{array}$$

$$\begin{array}{c}
 \text{GInj-Infix } (k \geq 0) \\
 \frac{{}^\perp s^\perp \Rightarrow^* \rho_0 \dots [{}^\perp s^\perp \Rightarrow^* \dots \rho_i \dots]_{0 \leq i \leq k} \quad {}^\perp s^\perp \Rightarrow^* \dots \rho_k}{\rho_0 ! s {}^\top s^\top \Leftarrow \langle \rho_0 \rangle_s^0 [\mathfrak{X}^s \langle \rho_i \rangle_s^0]_{0 \leq i \leq k}}
 \end{array}$$

$$\begin{array}{c}
 \text{GInj-Prefix } (k \geq 0) \\
 \frac{[{}^\perp s^\perp \Rightarrow^* \dots \rho_i \dots]_{0 \leq i \leq k} \quad {}^\perp s^\perp \Rightarrow^* \dots \rho_k}{{}^\top s^{\rho_k ! s} \Leftarrow \alpha^s \langle \rho_0 \rangle_s^0 [\mathfrak{X}^s \langle \rho_i \rangle_s^0]_{0 \leq i \leq k}}
 \end{array}$$

$$\begin{array}{c}
 \text{GInj-Postfix } (k \geq 0) \\
 \frac{{}^\perp s^\perp \Rightarrow^* \rho_k \dots [{}^\perp s^\perp \Rightarrow^* \dots \rho_i \dots]_{k \geq i \geq 0}}{\rho_k ! s {}^\top s^\top \Leftarrow [\langle \rho_i \rangle_s^0 \mathfrak{X}^s]_{k \geq i \geq 0} \langle \rho_0 \rangle_s^0 \mathfrak{D}^s}
 \end{array}$$

Fig. 25. Grout injection extending the definition of terminals τ (Fig. 12) and reduction $\sigma \Leftarrow \overline{\chi}$ (Fig. 17)

$$\begin{array}{ccccc}
 {}^\perp E^\perp \Rightarrow {}^0 E^0 \mathfrak{X}^0 E^0 \mid {}^0 E^0 \mathfrak{X}^0 E^0 \mathfrak{X}^0 E^0 \mid \dots & {}^\perp P^\perp \Rightarrow {}^0 P^0 \mathfrak{X}^0 P^0 \mid {}^0 P^0 \mathfrak{X}^0 T^\perp \mid \dots & {}^\perp T^\perp \Rightarrow {}^0 T^0 \mathfrak{X}^0 T^0 \mid \dots \\
 \Downarrow \curvearrowright \text{E}^\perp \Rightarrow \alpha^0 E^0 \mid \alpha^\perp P^\perp \mathfrak{X}^0 E^0 \mid \dots & \Downarrow \curvearrowright \text{P}^\perp \Rightarrow \alpha^0 P^0 \mid \alpha^\perp T^\perp \mid \dots & \Downarrow \curvearrowright \text{T}^\perp \Rightarrow \alpha^0 T^0 \mid \dots \\
 {}^\perp E^\top \Rightarrow {}^0 E^0 \mathfrak{D} \mid {}^0 E^0 \mathfrak{X}^0 T^\perp \mathfrak{D} \mid \dots & {}^\perp P^\top \Rightarrow {}^0 P^0 \mathfrak{D} \mid {}^0 P^0 \mathfrak{X}^0 T^\perp \mathfrak{D} \mid \dots & {}^\perp T^\top \Rightarrow {}^0 T^0 \mathfrak{D} \mid \dots \\
 \Downarrow \curvearrowright \text{E}^\top \Rightarrow \circ \mid \alpha^0 E^0 \mathfrak{D} \mid \alpha^\perp P^\perp \mathfrak{D} \mid \dots & \Downarrow \curvearrowright \text{P}^\top \Rightarrow \circ \mid \alpha^0 T^0 \mathfrak{D} \mid \dots & \Downarrow \curvearrowright \text{T}^\top \Rightarrow \circ \mid \dots
 \end{array}$$

Fig. 26. Excerpt of the grout rules injected (Fig. 25) into \mathcal{H}_{HZ} (Fig. 14). The production rules are arranged and color-coded by whether they emerge from subsuming **reduction** or by **tightening** (Fig. 17).

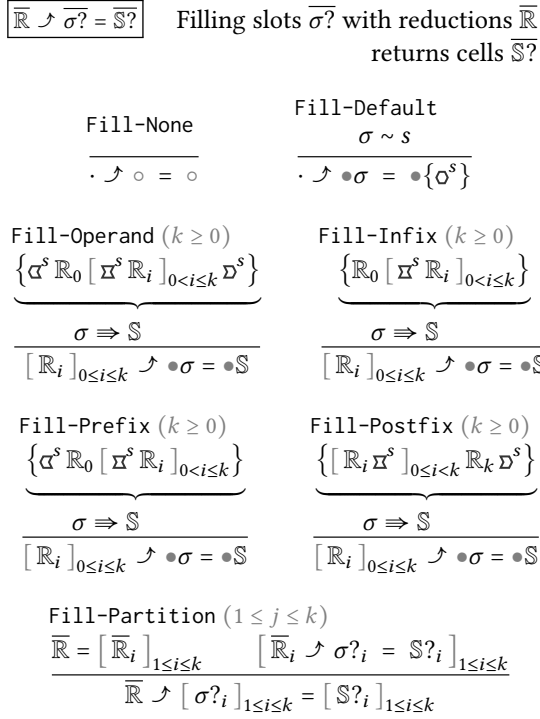


Fig. 27. Filling slots

$$\begin{aligned} \text{parse}(\mathbb{K}, \cdot) &= \mathbb{K} \\ \text{parse}(\mathbb{K}, \tau \bar{\tau}) &= \text{parse}(\mathbb{K} \leftarrow \tau, \bar{\tau}) \end{aligned}$$

Fig. 28. Parsing with meldr

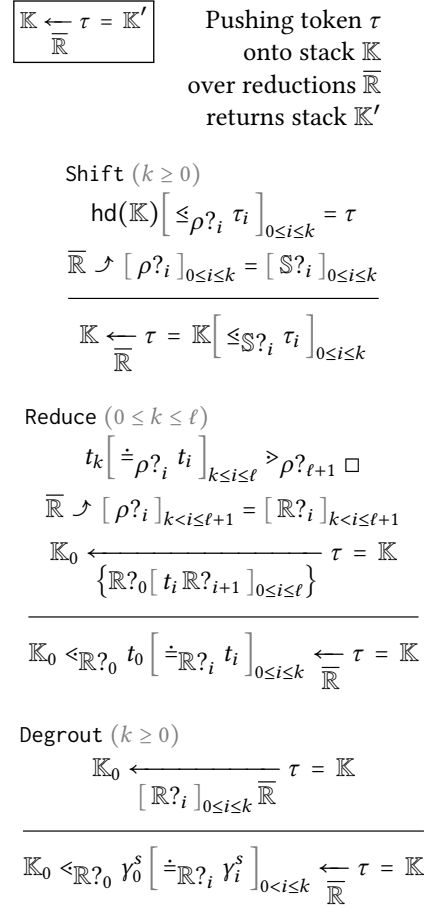


Fig. 29. Pushing with meldr

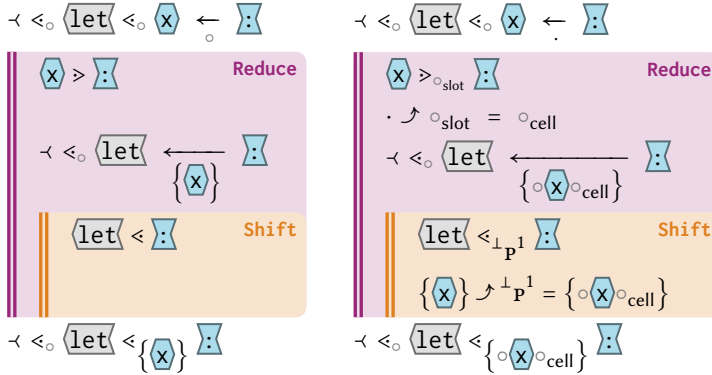
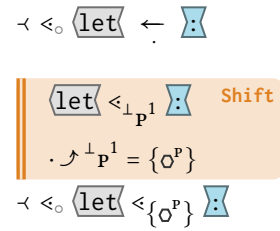


Fig. 30. Corresponding traces of OP parsing (left) and meldr (right) on the same inputs to highlight their differences

Fig. 31. meldr filling the slot $\perp P^1$ with the grout form $\{o^p\}$

the precedence comparisons. In **Reduce**, nothing \cdot is assigned to the unfillable slot \circ_{slot} ; in **Shift**, the reduction $\{\mathbf{x}\}$ is assigned to the fillable slot $\bullet^{\perp} p^1$. 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. 31 shows how pushing \mathbf{x} against the stack $\prec \prec_{\circ} \langle \text{let} \rangle$ leads to the slot $\bullet^{\perp} p^1$ getting filled instead with convex grout $\{\mathbf{o}^p\}$.

Fig. 32 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 $\langle \text{Num} \rangle$ or \succ against the stack $\prec \prec_{\circ} \langle \text{let} \rangle$ because $\langle \text{let} \rangle$ is precedence-comparable with either, meldr can proceed because it finds extended walks like $\langle \text{let} \rangle \prec_{\circ} \alpha^p \prec_{\circ} \langle \text{Num} \rangle$ and $\langle \text{let} \rangle \dot{\prec}_{\perp p^1} \Xi \dot{\prec}_{\perp E^1} \langle \text{in} \rangle \succ_{\perp E^1} \succ$. In both of these cases, the fill operation has multiple ways of assigning the initial reduction $\{\mathbf{x}\}$ to the traversed slots, as determined by the rule Fill-Partition. Whichever walk is chosen and however its slots are filled (§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, 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. 33b that handles pushing $\langle \text{let} \rangle$ against the stack $\prec \prec_{\circ} \langle 2 \rangle$. 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.

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 3.3 (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 3.4 (PARSING IS SOUND AND TOTAL). *For all well-formed stacks \mathbb{K} wf and tile sequences \bar{t} , there exists well-formed stack $\prec \dot{\prec}_{\mathbb{S}} \succ$ wf such that $\text{parse}(\mathbb{K}, \bar{t} \succ) = \prec \dot{\prec}_{\mathbb{S}} \succ$.*

The traces in Fig. 33 illustrate this guarantee for the failed examples in Fig. 23 and Fig. 24.

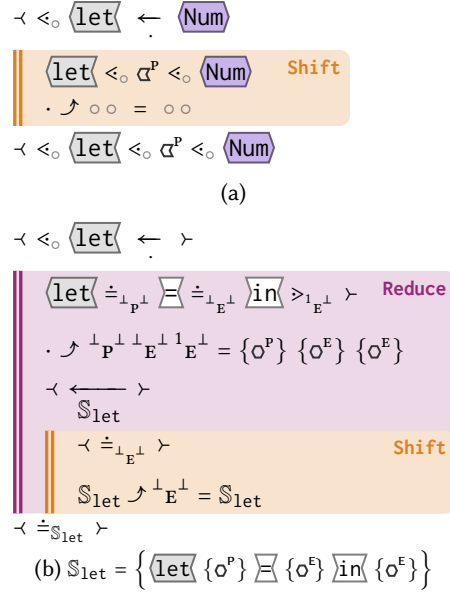


Fig. 32. meldr traces with multi-step precedence walks

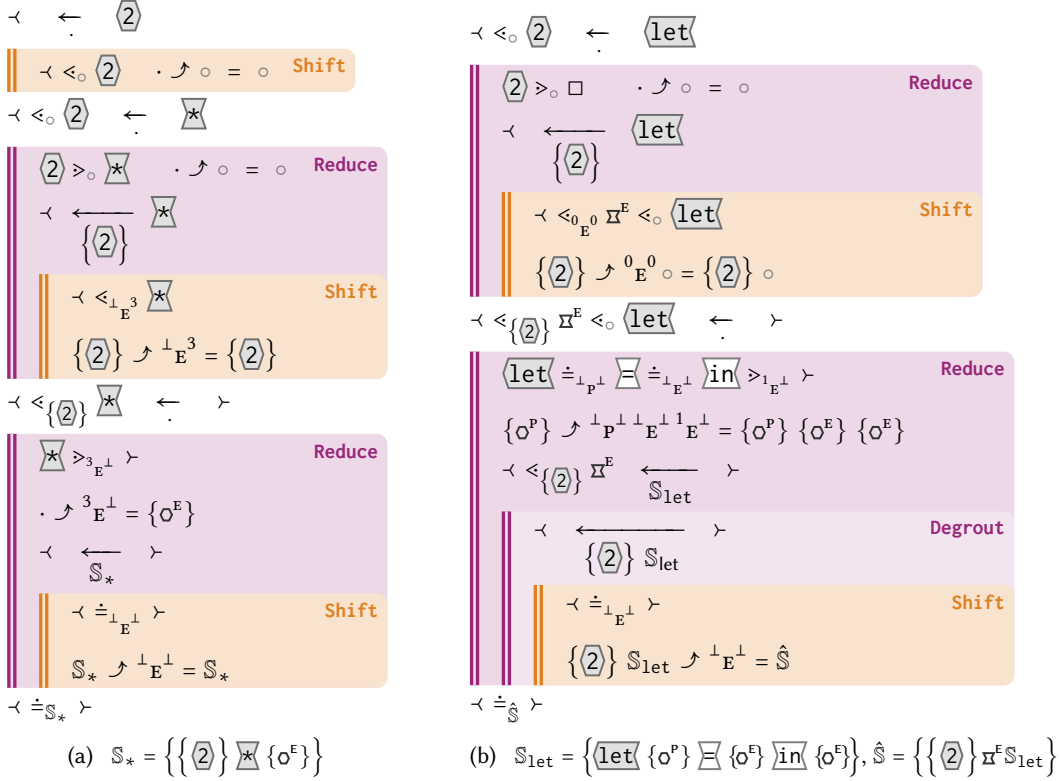


Fig. 33. Complete parsing traces using the rules in Fig. 29 to illustrate how meldr (a) avoids producing ill-formed terms like in Fig. 24 and (b) avoids getting stuck like in Fig. 23

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 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, Section 5 presents the user study we conducted to evaluate these decisions.

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 §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 \circ 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 \triangleleft and postfix \triangleright grout indicate there is a term as expected ($\bullet = 1$), but of the wrong sort.
- Infix grout \boxtimes 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 ty1r 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 edit-

$$\begin{aligned} \prec \triangleleft \circ \boxed{\text{let}} \leftarrow \boxed{\langle \rangle} &= \prec \triangleleft \circ \boxed{\text{let}} \triangleleft_{\{o^p\}} \boxed{\triangleright} \triangleleft \circ \boxed{\langle \rangle} \\ \prec \triangleleft \circ \boxed{\text{let}} \leftarrow \boxed{\langle \rangle} &= \prec \triangleleft \circ \boxed{\text{let}} \triangleleft \circ \boxed{\langle \rangle} \\ \prec \triangleleft \circ \boxed{\text{let}} \leftarrow \boxed{\langle \rangle} &= \prec \triangleleft \circ \boxed{\text{let}} \triangleleft \circ \triangleleft^p \triangleleft \circ \boxed{\langle \rangle} \end{aligned}$$

ing Hazel (Fig. 13), suppose the token \langle is inserted against the stack $\prec \triangleleft \circ \boxed{\text{let}}$. 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 $\prec \triangleleft \circ \boxed{\langle \rangle} \leftarrow \boxed{\triangleright}$:

$$\begin{aligned} (A) \quad & \boxed{\langle \rangle} \triangleleft_{\{o^e\}} \boxed{\triangleright} & (D) \quad & \boxed{\langle \rangle} \triangleleft \circ \boxed{\langle \rangle} \triangleleft_{\{o^e\}} \boxed{\triangleright} \\ (B) \quad & \boxed{\langle \rangle} \triangleleft_{\{o^e\}} \boxed{\triangleright} \triangleleft_{\{o^e\}} \boxed{\triangleright} & (E) \quad & \boxed{\langle \rangle} \triangleleft \circ \boxed{\text{let}} \triangleleft_{\{o^p\}} \boxed{\triangleright} \triangleleft \circ \boxed{\langle \rangle} \triangleleft_{\{o^e\}} \boxed{\triangleright} \\ (C) \quad & \boxed{\langle \rangle} \triangleleft_{\{o^e\}} \boxed{\triangleright} \triangleleft_{\{o^e\}} \boxed{\triangleright} \triangleleft_{\{o^e\}} \boxed{\triangleright} & (F) \quad & \boxed{\langle \rangle} \triangleleft \circ \triangleleft^e \triangleleft \circ \boxed{\langle \rangle} \triangleleft_{\{o^e\}} \boxed{\triangleright} \end{aligned}$$

tall ty1r limits the walks considered to those of shortest length found via breadth-first search, ruling out options like (B) and (C). tall ty1r also filters out walks with strictly \triangleleft -intermediate tile levels, such as $\boxed{\text{let}} \triangleleft_{\{o^p\}} \boxed{\triangleright}$ 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.

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 [24, 25]. 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, maintaining, 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 §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

$$\prec \prec_{\circ} \llcorner \prec_{\circ} \boxed{2} \mid \boxplus \succ_{\circ} \{\boxed{3}\} \rrcorner \succ_{\circ}$$

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

$$\prec \prec_{\circ} \boxed{\text{let}} \doteq_{\{o^p\}} \boxminus \doteq_{\{o^e\}} \boxed{\text{in}} \prec_{\circ} \boxed{4} \mid \succ_{\circ}$$

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.

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. 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 [4, 5]. Meanwhile, prior work on enumerating local repairs [7, 10] suggests this can be done efficiently. We leave detailed optimizations along these lines to future work.

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?

Table 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

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

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 §5.2.

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 ($\mu = 28$); 5-25 years of programming experience ($\mu = 13$), and 1-15 years of functional programming experience ($\mu = 6$).

5.1.2 Tasks. We chose nine code editing tasks (Table 1) intended to reflect real-world use patterns, six of which are adapted from a previous study [25]. 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. 34. 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.

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.

5.2 Results

Participant assessments of overall usability are summarized in Fig. 35, 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

```

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

```

→

```

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

```

Fig. 34. 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.

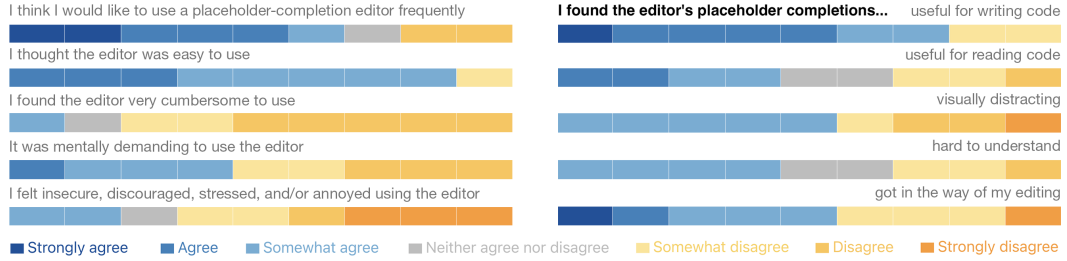


Fig. 35. Participant opinions on tall tylr's general usability (left) and reactions to placeholders (right)

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. 36 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.

A `(A -> Acc -> Acc) -> List(A) -> Acc -> Acc`
B `(A -> Acc -> Acc, *) * (List(A) -> Acc -> Acc`
C `(A -> Acc -> Acc, List(A)) -> Acc -> Acc`

Fig. 36. 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 an 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”.

5.3 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.

6 Related Work

OP Parsing. Operator-precedence (OP) parsing [13] is an early form of bottom-up parsing, which proceeds by iteratively reducing the input sequence of terminal symbols to a single start nonterminal. The goal at each iteration is to find the next *handle*, a subsequence of symbols matching the righthand side of a grammar production rule, and replace it with the lefthand side nonterminal. Unlike other parsing methods, OP parsing elides the reduced nonterminals. Handles are identified as chains of precedence-related terminals of the form $t_L < t_0 \doteq \dots \doteq t_k > t_R$, where t_L and t_R delimit the handle's terminals $t_0 \dots t_k$. Where typically the handle would be replaced by its reduced nonterminal, OP parsing instead replaces it with a precedence comparison $\odot \in \{<, \doteq, >\}$ describing the relation between the delimiters t_L and t_R .

Levy [21] observed that OP parsing is consequently unsound,¹ as we illustrated in Fig. 24. In a resolution similar to ours, Henderson and Levy [17] 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. Our approach generalizes this idea by indexing each relation by the optional nonterminal itself—the slot-filling operation in mldr (Fig. 27) uses this extra information to validate that the bottom-up accumulated reduction meets the top-down slot's requirements.

Precedence Annotations. In OP parsing, the precedence comparisons are derived from the derivation patterns of an unannotated CFG. 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 proliferation of nonterminals, one for each precedence level, that obscure the language's natural organization into semantically meaningful sorts. The rule Produce-Tighten in our PBG-to-CFG elaboration (Fig. 17) automatically extracts these dependency structures from a sort-organized PBG. 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.

Predominant interpretations of precedence annotations are specific to the parsing method—in LR parsers generators, for example, the annotations are used to resolve shift/reduce conflicts in the generated action table [2]. Less common are parser-independent semantics, such as our PBG-to-CFG elaboration (Fig. 17). §3.1.4 described how prior semantics by de Souza Amorim and Visser [9] (for the language workbench Spoofox [18]) and Danielsson and Norell [8] (for mixfix operators in Agda) are unnecessarily restrictive in how they handle prefix and postfix operators. Making similar observations, Aasa [1] defined the precedence weights that we recapitulate in our elaborated reduction rules (Fig. 17). Aasa used these measure 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 [8]. Our elaboration re-centers Aasa's precedence weights via a novel bidirectional organization.

Error Handling. Modern parsers are expected to be able to *recover* from errors (i.e. unexpected tokens) and continue parsing around the error site. 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 [3, 16] 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 illustrated in Fig. 1B, leaving the programmer without downstream semantic analysis. mldr takes an opposite approach, where the tokens dropped by a

¹Levy 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.

panicking parser are instead completed, reduced, and propagated up the stack with the assurance that, eventually, grout can be used to join together extraneous terms.

To minimize skipped input, more sophisticated methods [7, 11, 14] consider the full range of possible repairs around an error, 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 to our work is the FMQ method [12], which performs repairs using only insertions. This work defines the *insert-correctable* class of grammars against which any input text can be repaired by insertions to grammatical form—in our work, grout injection (Fig. 25) systematically relaxes any grammar to be insert-correctable.

Across their variations, these prior repair-based recovery methods limit themselves to purely textual repairs. This can lead to combinatorial explosion in the space of possible insertion repairs, as illustrated in Fig. 1C. While these repairs can be enumerated efficiently in practice [7, 10], prior work does not consider the question of how to effectively surface these repairs to the programmer. Our approach is novel in its use of abstract syntactic obligations to compress, communicate, and rank the space of possible repairs.

Structure Editing. tall tylr continues a series of design experiments in increasingly flexible and text-like structure editing. Its predecessors, tiny tylr [24] and teen tylr [25], proposed and refined a model of structure editing in which the primary units of the edit state are nested spans of matching tokens, there called *tiles* of matching *shards*. A design consequence of this particular physical metaphor was that shards remained matched for life in those editors, which Moon et al. [25] observed compromised overall usability. Relaxing this restriction, and repurposing the term *tile* for molded tokens, led to the present design.

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 token-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.

7 Conclusion

This paper 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 paper.

Acknowledgements

This work was partially funded by the National Science Foundation under Grant No. 2238744. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

Data Availability Statement

This paper includes an artifact [23] consisting of the study materials given to participants in the user study. These include a slideshow, introductory videos, an exit survey, and a copy of the tall tyler editor prototype, which can be run in a modern web browser. A detailed table of contents and instructions are provided in `README.md`.

References

- [1] 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
- [2] 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
- [3] 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.
- [4] Alessandro Barengi, 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
- [5] Alessandro Barengi, 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
- [6] Sam Cohen and Ravi Chugh. 2025. Code Style Sheets: CSS for Code. arXiv:2502.09386 [cs] doi:10.1145/3720421
- [7] Breandan Considine, Jin Guo, and Xujie Si. [n. d.]. Syntax Repair as Idempotent Tensor Completion. ([n. d.]).
- [8] 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
- [9] 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
- [10] 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
- [11] 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.
- [12] 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
- [13] Robert W. Floyd. 1963. Syntactic Analysis and Operator Precedence. *J. ACM* 10, 3 (July 1963), 316–333. doi:10.1145/321172.321179
- [14] 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
- [15] 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
- [16] Dick Grune and Criel J.H. Jacobs. 2008. *Parsing Techniques: A Practical Guide* (2 ed.). Springer, New York, NY, USA.
- [17] 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
- [18] Lennart C.L. Kats and Eelco Visser. 2010. The Spoox 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
- [19] Paul Klint and Eelco Visser. [n. d.]. Using Filters for the Disambiguation of Context-free Grammars. ([n. d.]).
- [20] Amy J. Ko and Brad A. Myers. 2005. A Framework and Methodology for Studying the Causes of Software Errors in Programming Systems. *Journal of Visual Languages & Computing* 16, 1 (Feb. 2005), 41–84. doi:10.1016/j.jvlc.2004.08.003
- [21] M. R. Levy. 1975. Complete Operator Precedence. *Inform. Process. Lett.* 4, 2 (Nov. 1975), 38–40. doi:10.1016/0020-0190(75)90010-1

- [22] 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
- [23] David Moon. 2025. Artifact for Syntactic Completions with Material Obligations. Zenodo. doi:10.5281/zenodo.17007910
- [24] 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
- [25] 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
- [26] 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
- [27] 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]
- [28] Young Seok Yoon and Brad A. Myers. 2014. A Longitudinal Study of Programmers' Backtracking. In *2014 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 101–108. doi:10.1109/VLHCC.2014.6883030

Received 2025-03-26; accepted 2025-08-12