Syntactic Completion with Material Obligations

DAVID MOON, University of Michigan, USA

1 INTRODUCTION

Programming is cognitively demanding, so novices and experts alike rely on various editor services to help them understand, navigate, and modify programs. Often these services depend on the program's syntactic structure, specified by the language *grammar* and inferred by a *parser* from the authored program text. The problem is that, for typical grammars, most textual edit states do not parse directly into valid structure [Yoon and Myers 2014]. Consequently, to ensure continuous availability of essential services, parsers must be able to accommodate and recover from partial and erroneous input.

Despite this need, and corresponding research spanning several decades, parser error handling remains frustratingly inadequate today. A simple and common approach known as "panic mode" skips unexpected tokens until it finds a synchronization token—a likely end-of-term delimiter specified by the grammar author—from which parsing can resume [Grune and Jacobs 2008]. While easy to implement, this approach is liable to ignore large windows around error sites, leaving the programmer without assistance where they may need it most. For example, consider the syntactically malformed program shown in Fig. 1(A), written in an OCaml-like language with parenthesized, comma-separated (rather than space-separated) function arguments. The first line of Fig. 1(B) shows how a panicking parser might skip the unexpected token `p2`; reach the synchronizing token `let` on the next line; and consequently, to recalibrate with it, drop everything parsed so far.

A
```
fun (p1 p2
  let (x1: Num, y1: Num) = p1 in
  let (Num, y2: Num) = p2 in
  sqrt(pow(x1 -, 2) + pow(y1 - y2, 2))
```

B
```
fun (p1 p2
  let (x1: Num, y1: Num) = p1 in
  let (Num, y2: Num) = p2 in
  sqrt(pow(x1 -, 2) + pow(y1 - y2, 2))
```

C
```
fun (p1 p2 ->
fun (p1 p2 ->
fun (p1 p2 ->
fun (p1, p2) ->
fun (p1::p2) ->
fun (p1 | p2) ->
fun (p1 as p2) ->
...
```

Fig. 1. (A) A function computing the distance between points, partially written in an OCaml-like language with parenthesized function application. (B) Regions skipped by a panicking parser, highlighted in red. (C) Some possible textual repairs for the first line generated by an error-correcting parser.

More sophisticated methods consider the full range of textual repairs around the error site. Fig. 1(C) shows some possible repairs for the first line of code in Fig. 1(A). The first three repairs show how this method reduces ignored input (i.e. deleted tokens) compared to a panicking parser. On the other hand, the last four insertion-only repairs show how this method must enumerate all tokens that play similar structural roles—in this case, infix operators on patterns—which can lead to combinatorial explosion as additional insertions and larger repair windows are considered [Considine et al. [n. d.]; Diekmann and Tratt 2020]. The choice is then passed on to the programmer, causing potential information overload [Ko and Myers 2005], or else the parser picks a repair using ad hoc heuristics [Fischer et al. 1979; Graham and Rhodes 1975]—what's worse, the heuristically chosen repair is rarely surfaced directly to the programmer, leaving them only indirect clues in the behavior of downstream editor services.

The error handling methods described so far have the advantage of working for all grammars, but due to their various shortcomings, many production-grade parsers instead employ handwritten recovery procedures particular to their grammars. This approach enables higher-quality diagnostics, but at the cost of substantial additional effort during parser development [Diekmann and Tratt 2020], a notoriously effortful process on its own [Malloy et al. 2002].

Author's address: David Moon, dmoo@umich.edu, University of Michigan, Ann Arbor, MI, USA.

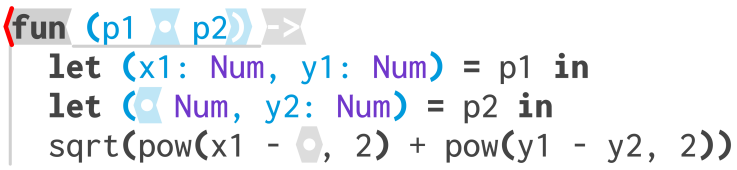


Fig. 2. A syntactic completion of the program in Fig. 1 using obligations in a tile-based editor. When parsed, each input token is assigned a *grammar mold* (formally speaking, a zipper [Huet 1997] into the grammar), visually signified by color for sort and shape for its hierarchical relationship with its neighbors. A token together with its mold is called a *tile*. User-inserted tiles are shown in dark font, system-inserted obligations in light font. Obligations come in two forms: specific missing tiles expected by their matching counterparts, such as `->` expected by `fun`; and *grout*, notated with hole-like icons, that stand in for multiple possible replacements.

I propose defending the following thesis: it is possible to design an error-handling parser that

- requires no additional specification beyond the grammar;
- ignores no input tokens and abstracts over similarly structured textual completions; and
- materializes these abstracted completions to the programmer in a visible and usable manner.

Instead of arbitrating between similarly structured textual completions, my proposed design incorporates a novel language of *syntactic obligations* that abstract over many possible textual replacements. Importantly, these obligations are *materialized* in the edit state alongside the text that they complement, making scaffolded structures explicit to the programmer—for example, the completed program in Fig. 2 shows the obligatory delimiter `->` matched with its user-inserted counterpart `fun`, keeping separate the function's arguments and body. This obligation system is housed in a novel theory of *tile-based parsing and editing*. In a tile-based editor, parsing turns tokens into tiles by assigning them *grammar molds* (formally speaking, zippers [Huet 1997] into the grammar) which are visually indicated using color and shape, as depicted for the `fun` keyword in Fig. 2. Complementing these tiles are two kinds of obligations: specific missing tiles (e.g. `->` on the first line) expected by their matching counterparts (e.g. the `fun` tile); and *grout*, notated with a hole-like icon on the first, third, and fourth lines, which stand in for many possible replacements.

2 PRIOR WORK

My proposed thesis is the culmination of a series of program editor design iterations [Moon et al. 2022, 2023]—screenshots of which are shown in Fig. 3—where the original aim was to improve the usability of structure editing.

Unlike parser-equipped text editors, *structure editors* maintain a tree-structured editor state, with holes standing for missing terms (i.e. subtrees), and project this internal model to various interactive forms, typically embedded in a two-dimensional display. The choice of projection varies widely—from the bright drag-and-droppable blocks of Scratch [Maloney et al. 2010] to the keyboard-driven text-like interface of JetBrains MPS [Voelter and Pech 2012]—but all projections restrict user interactions to simple operations on the underlying tree, which ensures that the edit state is continuously well-structured and amenable to analysis. However, these restrictions can be in tension with the program's visually flattened projection and its suggested affordances, leading many [Bahlke and Snelting 1992; Bau et al. 2017; Goldenson and Lewis 1988; Holwerda and Hermans 2018; Lang 1986; Miller et al. 1994; Minör 1992; Monig et al. 2015; Van De Vanter 1995; Voelter et al. 2014] to report a highly viscous [Green and Petre 1996] editing experience—that is, it is cumbersome to modify existing code.

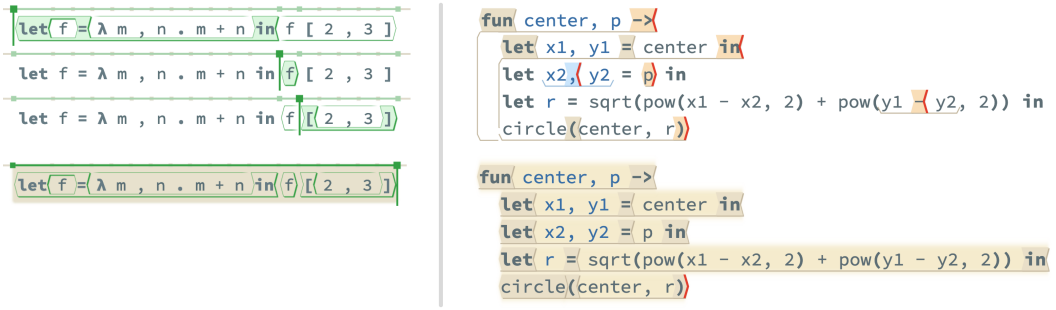


Fig. 3. Various cursor positions and selection states in `tiny tylr` [Moon et al. 2022] (left) and `teen tylr` [Moon et al. 2023] (right).

My prior design efforts identified and targeted three major facets of this viscosity in prior keyboard-driven textually projecting structure editors:

- **(selection expressivity)** Selections must cleave to complete terms. It is impossible to select portions of a term, or for selections to span across term boundaries, even when the intended tokens are visually adjacent. For example, in `2 * 3 + 4`, it is impossible to select `3 + 4` or `+ 4`.
- **(delimiter matching)** All keywords or symbols in the projection of a term must be inserted and deleted together. These matching delimiters may be visually distant from one another due to intervening children, leading to a sort of "spooky action at a distance" and making edits that amount to repositioning or changing an individual delimiter difficult. For example, deleting the closing parenthesis in `f(2 * 3) + 4`—with the intent, say, of re-inserting it after `4`—also deletes the matching open parenthesis (as well as the function argument, which brings us to the next facet).
- **(term multiplicity)** The edit state is modeled as an optionally present, individual tree, which rules out the possibility of having multiple adjacent trees, even temporarily. Consequently, when deleting a tree node with multiple children, e.g. function application, the editor must delete all but one of its children as well. This leads to deletion behavior that even experts find challenging to predict [Berger et al. 2016].

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

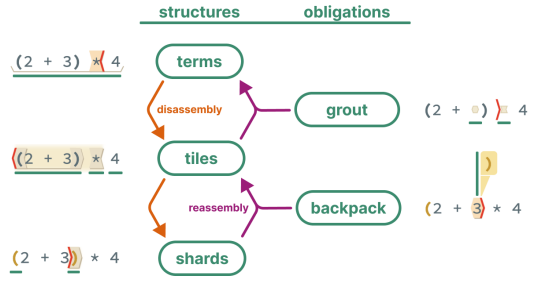


Fig. 4. A high-level schematic of the concepts of tile-based editing as described in [Moon et al. 2023]. Our terminology has since evolved in our ongoing work (cf. the end of this section).

I presented, as a concrete instantiation of gradual structure editing, the design, implementation, and evaluation of a *tile-based editor*, so called because disassembly proceeds through three distinct strata—*terms*, *tiles*, and *shards*, ordered high to low as depicted in Fig. 4. Disassembly to lower structures occurs when the user's selection boundaries cut across the linear span of the higher structure, thereby addressing the selection expressivity problem. For example, the depicted selection `(2 + 3) *` (middle left) reveals the containing term's disassembly into its constituent tiles; similarly, the selection `)` (lower left) reveals the containing tile's disassembly into its shards, i.e. its matching delimiters.

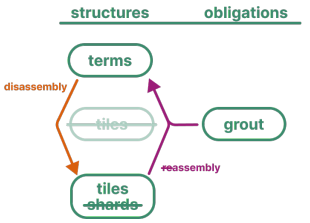
After insertion or deletion, syntactic obligations are generated to ensure eventual reassembly of any remaining lower structures. This bookkeeping is managed and presented by two independent subsystems operating at distinct levels of the structural strata.

(1) The *backpack* scaffolds reassembly from shards to tiles by managing matching delimiter obligations, presenting these obligations in a pop-up stack attached to the cursor, as depicted in the lower right of Fig. 4.
(2) The *grouter* scaffolds reassembly of tiles into terms, managing multiplicity obligations by inserting and removing *grout*. Grout are generated based on the requirements of neighboring tiles, which are shaped on either end with a concave or convex tip to indicate whether or not, respectively, delimits a child operand. Grout generalize holes to support both missing terms (<1) and adjacent terms (>1); for example, the upper right of Fig. 4 shows a convex piece of grout standing in for the missing operand 3, as well as a concave piece of grout connecting the former operands of the missing operator `*`.

We implemented a tile-based editor called `teen tylr` (the successor to `tiny tylr` [Moon et al. 2022]) and evaluated it in a small ($n = 10$) within-subjects lab study wherein we asked participants to complete short editing tasks using VS Code, a popular text editor; JetBrains MPS [Berger et al. 2016], a state-of-the-art textually projecting structure editor; and `teen tylr`. We designed our tasks to require somewhat complex structural modifications, an example of which is shown in Fig. 5. We observed that our decomposition of structure editor modification viscosity into selection expressivity, term multiplicity, and delimiter matching helped explain most of the breakdowns participants encountered with MPS—in particular, we found that multiplicity and matching constraints applied competing demands for limited clipboard space, leading to less code reuse and greater mental load compared to VS Code and `teen tylr`.

On the other hand, participants reported mixed opinions about the backpack system. Part of the issue was its dual and conflated roles as a clipboard and matching obligation tracker. A more severe limitation was that it did not support assembling shards from different tiles, which participants found constraining relative to their usual text editing habits.

In addressing these issues in my ongoing work, I have simplified the tile-based vocabulary and ontology from the one in Fig. 4 to the one on the right. Here, I rid the previous concept of tiles as intermediate structures of matching shards, and rename shards as tiles in order to dispel the physical metaphor that they are fractured components of a particular parent entity and should be reassembled as such. Meanwhile, the grouter subsumes the role previously handled by the backpack, such that grout elements represent both multiplicity and delimiter matching obligations—in the latter case, they are additionally decorated with transparent text of the missing delimiters.



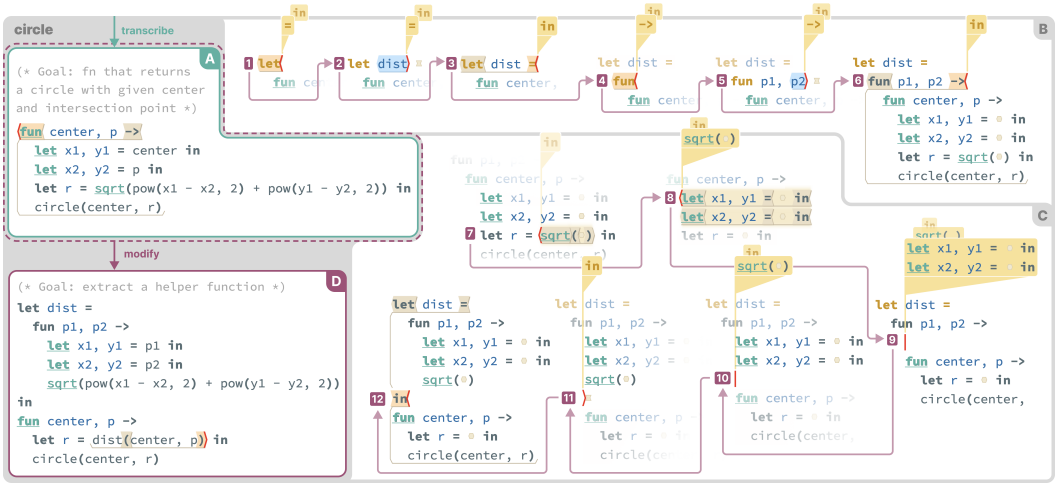


Fig. 5. A pair of editing tasks we assigned our lab study participants, consisting of a **transcription** task (Panel A) followed by a **modification** task (Panel D), and the edit sequence by which participant P9 completed the modification task using `teen tylr` (Panels B & C). Due to space constraints, the variable references `center` and `p` and the argument to `sqrt` in Panels A & D are elided in Panels B & C. In Panel B, P9 begins binding a new variable `dist` (B.1-3) to a newly inserted function taking arguments `p1` and `p2` (B.4-6). Subsequently, in Panel C, P9 selects and cuts the `sqrt` expression and the two preceding `let`-lines (C.7-9), pastes them above the original function (C.9-11), and completes the `let`-binding for `dist` with the concluding delimiter `in` (C.11-12). Finally, not shown, they modify the variable references `center` and `p` to `p1` and `p2` and inserted a call to the newly defined `dist` function to arrive at Panel D.

3 ONGOING WORK

Lifting the remaining usability limitations of `teen tylr` leads to the design of a novel error-correcting parser. Virtually all prior work on error-handling parsing stops short of considering the details of presentation and interaction—in contrast, these were the air and water of our design efforts in the structure editing realm. This design emphasis led naturally to a visual concentration of grammar-relevant signifiers (e.g. color for sort) on the tokens of the edit state—this is in contrast to the points in between tokens, illustrated in Fig. 6, that correspond to the assumed primitives (e.g. Earley items, LL/LR automaton states) of prior work on error handling, and that afford much less visual real estate. Thus, I depart from current parsing tradition and instead build on the token-centric foundations of *operator precedence* (OP) parsing.

$$E \rightarrow \bullet\texttt{let}\bullet P \bullet\texttt{=}\bullet E \bullet\texttt{in}\bullet E\bullet$$
$$E \rightarrow \langle\texttt{let}\rangle\, P \,\langle\texttt{=}\rangle\, E \,\langle\texttt{in}\rangle\, E$$

Fig. 6. Earley items (top) vs token molds (bottom)

As first described by Floyd [Floyd 1963], an OP parser generalizes the "order of operations" method commonly taught for mathematical notation. The parser's execution is guided by a table of *precedence relations* between all tokens of the language, an example of which is shown in Fig. 7. Roughly speaking, given tokens l and r that appear in the input sequence in that order and sufficiently "near" each other, $l \lessdot r$ means l will be parsed into an ancestor of the term containing r, and vice versa for $l \gtrdot r$. Meanwhile, $l \doteq r$ means l and r will be parsed into the same term, i.e. they are matching delimiters.

OP parsers are easily specified and implemented, scale linearly with input, and moreover enjoy the attractive property of *local parsability* [Barenghi et al. 2015]: every subrange of a parseable input

sequence can be parsed into a "ziggurat"-like structure[1], in a manner that proceeds monotonically as the subrange grows. I believe this property makes OP parsing the right choice to power future program editing interfaces, as it can efficiently generate a uniform, maximally structured interpretation of any user selection, a powerful untapped primitive in the design of editor services.

$$S \to S + A \mid A$$
$$A \to A \times B \mid B$$
$$B \to n \mid (S)$$

	$+$	$\times$	$($	$)$	n
$+$	$\gtrdot$	$\lessdot$	$\lessdot$	$\gtrdot$	$\lessdot$
$\times$	$\gtrdot$	$\gtrdot$	$\lessdot$	$\gtrdot$	$\lessdot$
$($	$\lessdot$	$\lessdot$	$\lessdot$	$\doteq$	$\lessdot$
$)$	$\gtrdot$	$\gtrdot$		$\gtrdot$	
n	$\gtrdot$	$\gtrdot$		$\gtrdot$	

Fig. 7. A arithmetic grammar and its derived precedence table, in which the relation in each cell takes its row header as its left argument, its column header as its right. Adapted from [Barenghi et al. 2015].

Despite these advantages, OP parsing was historically overshadowed by more sophisticated methods because of its limited grammar expressivity. OP parsing stipulates that there be a unique precedence relation between every ordered pair of language tokens, which severely impedes the re-use of tokens across different forms (e.g. the minus symbol for both prefix negation and infix subtraction). I propose overcoming this limitation first by defining precedence relations not between raw textual tokens, but rather between zippers [Huet 1997] into terminal symbols in the grammar. These zippers pair the surrounding context of the grammar with each terminal symbol, which distinguishes between appearances of the same terminal symbol in different parts of the grammar. In the user-facing tile-based parlance, I refer to these zippers as *molds*. Taking as given for the moment an appropriate *molding policy* that first assigns molds to each input token, we can subsequently OP-parse the mold-distinguished tokens without issue.

Molds also give rise to a natural method of local error correction. With respect to a given grammar, Floyd originally defined the grammar's precedence relations based on patterns of its string derivations. In the zipper-based perspective, we can reframe precedence relations as differently shaped *walks* between zippers. The advantage of this framing is that these walks are exactly the possible local syntactic completions between neighboring molded tokens. In such a completion, any molds and nonterminals that appear strictly in between the endpoints of a walk would be inserted, respectively, as missing tiles and grout in the editor.

What remains to be described is the aforementioned molding policy. This molding policy plays a spiritually similar role to the top-down handle-finding LR automaton that guides bottom-up reduction, but in this setting no additional machinery is needed. Our policy simply considers all possible molds assignable to a given token, attempts shifting each molded candidate onto the parse stack, and picks the mold that minimizes local obligations. This approach leads to a worst-case time complexity of $O(kn)$, where n is the number of input tokens, and k is the maximum number of times a token is reused throughout the grammar, which is small for typical grammars.

I plan to make the following contributions, organized into parallel tracks of theoretical and empirical evaluation.

- I plan to formally characterize my work as grammar-parametrized parsing and editor calculi. Specifically, I plan to complete the following by late June:
  - An implementation-independent semantics of context-free grammars (CFGs) whose rules are divided into precedence levels, which I refer to as *precedence-bounded grammars* (PBGs). PBGs are a common method of documenting precedence-organized syntax [OCaml developers [n. d.]], but the grammars specifying parsers usually cannot be expressed so succinctly in the language of pure context-free grammars due to parsing

[1] Formally speaking, there is a unique reduction z of the input subrange with a factorization $z = x\,y$ such that the precedence relations between consecutive terminals in x do not include $\lessdot$, and those in y do not include $\gtrdot$.

ambiguity. This semantics will take two forms: a notion of precedence-bounded derivation, a variation of derivations for CFGs that is restricted according to the specified precedence levels; and a denotational semantics for PBGs as unambiguous CFGs, which I will show coheres with the notion of precedence-bounded derivation.
  - A formal tile-based parsing framework that produces a linear-time syntactically-completing parser for any PBG. The parser consists of a bottom-up OP-parsing core, defined to operate on grammar molds, and a top-down molding policy that assigns molds to tokens. Syntactic completion incorporates abstract user-facing obligations that complement user text, scaffold partial structures, and provide a natural ranking metric for the molding policy. I will show that this leads to a linear-time parsing algorithm, assuming a constant bound on the number of times a token is reused throughout the grammar. I additionally conjecture and hope to show that our parsing method admits a class of grammars that strictly contains the LL(1) grammars, while being a strict subset of LR(1) grammars.
  - A formal tile-based editing and incremental parsing calculus that gives a structured representation of any user selection. I will show that this framework guarantees that every edit state consists of a complete and well-formed term (possibly with obligations).
- On the empirical side, I have implemented the proposed parsing and editing calculi ($\tilde{2}$kLoC ReasonML) and am currently integrating this implementation into the frontend of the Hazel programming environment [Omar et al. 2017], which I expect to complete by mid-March. I plan to perform the following evaluations:
  - a case-study evaluation of our formalism grammar expressivity by specifying various practical language grammars as PBGs;
  - a case-study evaluation of our syntactic completion quality by comparing its completions with handwritten error recovery rules in open-source production-grade compilers [Grošup 2023], along with a small user study to further verify (in addition to our prior evaluation in [Moon et al. 2023]) that our obligation-based completions are helpful and not overwhelming (late May);
  - performance benchmarks validating that our parsing method performs reasonably well for real-world grammars (late June).

4 RELATED WORK

I am not the first to incorporate grammar zippers into parsing. Indeed, the items of Fig. 6 commonly used to represent parser states may be understood as grammar zippers that focus on the points between symbols, albeit they were developed before the first well-known description of zippers [Huet 1997]. Recent work [Darragh and Adams 2020; Edelmann et al. 2020] use grammar zippers to memoize grammar traversals in order speed up parsing with derivatives [Brzozowski 1964; Might et al. 2011], an elegant but inefficient method when implemented naively. In this work, I contribute a visual design for grammar zippers (molds) focused on user-inserted tokens and system-inserted obligations, and use them to develop a novel formal basis for operator precedence relations.

Wagner and Graham's work on parser error recovery [Wagner and Graham 1999] has similar aims to my proposed thesis. Situated within an incremental parser in an IDE, their approach distinguishes between programmer changes that introduce valid structural changes and those that do not—the latter are kept in their "unanalyzed" form (i.e. they are ignored) until additional changes (e.g. inserting matching tokens) allow them to be structurally incorporated. Wagner and Graham characterize this "non-correcting strategy" as a strength because it avoids guessing the programmer's intent regarding yet-to-be-inserted tokens. I argue, however, that their approach

provides inadequate semantic feedback for partially complete terms—for example, upon the programmer typing `let x : Num =|` (where `|` represents the cursor) left-to-right, their strategy would not inform downstream services that a `let`-expression is being constructed, meaning the programmer could not, say, get type-constrained autocompletions of the definition they are about to construct next. By completing programmer insertions with obligations, my proposed method ensures semantic feedback is available even for partially complete terms—for example, `let x : Num =|` would get completed to `let x : Num =| <> >in< <>`, such that completions at the cursor can incorporate the expected type `Num`. These obligation-based completions do require some guessing—while the obligations `<>` abstract over many textual completions, there is not always a unique place to insert missing tiles like `>in<` (e.g. consider if `let x : Num =` were inserted above a large program block). On the other hand, these obligations are not binding—the programmer may always choose to complete the program in a different way, and redundant obligations are pruned.

REFERENCES

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

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

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

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

Janusz A. Brzozowski. 1964. Derivatives of Regular Expressions. *J. ACM* 11, 4 (Oct. 1964), 481–494. https://doi.org/10.1145/321239.321249

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

Pierce Darragh and Michael D. Adams. 2020. Parsing with Zippers (Functional Pearl). *Proceedings of the ACM on Programming Languages* 4, ICFP (Aug. 2020), 1–28. https://doi.org/10.1145/3408990

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. https://doi.org/10.4230/LIPIcs.ECOOP.2020.6

Romain Edelmann, Jad Hamza, and Viktor Kunčak. 2020. Zippy LL(1) Parsing with Derivatives. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, London UK, 1036–1051. https://doi.org/10.1145/3385412.3385992

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.

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

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

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

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

Tomáš Grošup. 2023. Announcing F# 8. https://devblogs.microsoft.com/dotnet/announcing-fsharp-8/.

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

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

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

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. https://doi.org/10.1016/j.jvlc.2004.08.003

Bernard Lang. 1986. On the Usefulness of Syntax Directed Editors. In *Advanced Programming Environments (Lecture Notes in Computer Science)*, Reidar Conradi, Tor M. Didriksen, and Dag H. Wanvik (Eds.). Springer, Berlin, Heidelberg, 47–51. https://doi.org/10.1007/3-540-17189-4_87

Brian A. Malloy, James F. Power, and John T. Waldron. 2002. Applying Software Engineering Techniques to Parser Design: The Development of a C# Parser. In *Proceedings of the 2002 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists on Enablement through Technology (SAICSIT '02)*. South African Institute for Computer Scientists and Information Technologists, ZAF, 75–82.

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. https://doi.org/10.1145/1868358.1868363

Matthew Might, David Darais, and Daniel Spiewak. 2011. Parsing with Derivatives: A Functional Pearl. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming (ICFP '11)*. Association for Computing Machinery, New York, NY, USA, 189–195. https://doi.org/10.1145/2034773.2034801

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

Sten Minör. 1992. Interacting with Structure-Oriented Editors. *International Journal of Man-Machine Studies* 37, 4 (Oct. 1992), 399–418. https://doi.org/10.1016/0020-7373(92)90002-3

Jens Monig, Yoshiki Ohshima, and John Maloney. 2015. Blocks at Your Fingertips: Blurring the Line between Blocks and Text in GP. In *2015 IEEE Blocks and Beyond Workshop (Blocks and Beyond)*. 51–53. https://doi.org/10.1109/BLOCKS.2015.7369001

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. https://doi.org/10.1145/3546196.3550164

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. https://doi.org/10.1109/VL-HCC57772.2023.00016

OCaml developers. [n. d.]. The OCaml Language: Expressions. https://v2.ocaml.org/manual/expr.html.

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]

Michael L. Van De Vanter. 1995. Practical Language-Based Editing for Software Engineers. In *Software Engineering and Human-Computer Interaction (Lecture Notes in Computer Science)*, Richard N. Taylor and Joëlle Coutaz (Eds.). Springer, Berlin, Heidelberg, 251–267. https://doi.org/10.1007/BFb0035821

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

Markus Voelter, Janet Siegmund, Thorsten Berger, and Bernd Kolb. 2014. Towards User-Friendly Projectional Editors. In *Software Language Engineering (Lecture Notes in Computer Science)*, Benoît Combemale, David J. Pearce, Olivier Barais, and Jurgen J. Vinju (Eds.). Springer International Publishing, Cham, 41–61. https://doi.org/10.1007/978-3-319-11245-9_3

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

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. https://doi.org/10.1109/VLHCC.2014.6883030