Partial Parsing for Structured Editors

Tom Beckmann
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
tom.beckmann@hpi.uni-potsdam.de

Patrick Rein
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
patrick.rein@hpi.uni-potsdam.de

Toni Mattis
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
toni.mattis@hpi.uni-potsdam.de

Robert Hirschfeld
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
robert.hirschfeld@uni-potsdam.de

Abstract

Creating structured editors, which maintain a valid syntax tree at all times rather than allowing to edit program text, is typically a time consuming task. Recent work has investigated the use of existing general-purpose language grammars as a basis for automatically generating structured editors, thus considerably reducing the effort required. However, in these generated editors, input occurs through menu and mouse-based interaction, rather than via keyboard entry that is familiar to most users.

In this paper we introduce modifications to a parser of general-purpose programming language grammars to support keyboard-centric interactions with generated structured editors. Specifically, we describe a system we call *partial parsing* to autocomplete language structures, removing the need for a menu of language constructs in favor of keyboard-based disambiguation. We demonstrate our system's applicability and performance for use in interactive, generated structured editors. Our system thus constitutes a step towards making structured editors generated from language grammars usable with more efficient and familiar keyboard-centric interactions.

CCS Concepts: • **Software and its engineering** → *Formal language definitions.*

Keywords: partial parsing, structured editing, tree-sitter

ACM Reference Format:
Tom Beckmann, Patrick Rein, Toni Mattis, and Robert Hirschfeld. 2022. Partial Parsing for Structured Editors. In *Proceedings of the 15th ACM SIGPLAN International Conference on Software Language Engineering (SLE '22), December 06–07, 2022, Auckland, New Zealand.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3567512.3567522

SLE '22, December 06–07, 2022, Auckland, New Zealand

ACM ISBN 978-1-4503-9919-7/22/12.
https://doi.org/10.1145/3567512.3567522

1 Introduction

Structured editors allow editing abstract syntax trees (ASTs) of programs directly, rather than working on a textual representation that requires parsing to be turned into an AST. For novices, structured editors provide an interface that does not cause syntax errors and may guide them more clearly to formulate programs than a text-based editor otherwise would [2] and are thus often used in education [14, 18]. More generally, structured editors simplify language composition [23]. Since boundaries of AST nodes are clearly delineated in the model of the structured editor there is no ambiguity when grammars of two languages are composed. Further, the composed languages do not have to resemble textual languages but can be visual and domain-specific, for example allowing users to edit a state machine embedded in their code [20].

Typical structured editors are either hand-crafted, meaning their authors manually define appearance, rules for interactions, and serialization. Hand-crafting may result in interactions that are highly adapted to the specific constraints of the underlying language but necessarily require considerable engineering work. Otherwise, structured editors may also be generated from a specification, for example as expressed in a language workbench [9, 23]. Some generators also provide support for taking in the grammar of an already existing textual language and deriving a structured editor, such as Rascal2MPS or Kogi [13, 21, 22]. Correspondingly, reuse of existing grammars results in considerably less engineering work for deriving a structured editor for an existing language such as JavaScript or Python.

Concerning input, prior work in MPS has demonstrated that following user's expectations from textual editing, even in a structured editor, benefits efficiency [4]. The required interactions are added by the language's authors to inform

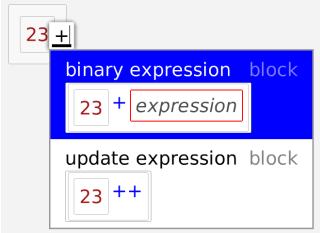

Figure 1. The user entered a plus token on an existing number node in a keyboard-driven structured editor. We derive two partial parse trees by taking the number node and the user input and present those for disambiguation through the user. The editor and its interactions are generated from a JavaScript grammar.

the editor how text-like editing should affect the tree structures [24]. The importance of following user expectations' from textual editing in structured editors has been identified in multiple places [11, 25]. Prior work [21, 22] demonstrated the possibility of fully automatically generating projectional editors in Blockly [8], a block-based framework for structured editors. Here, language constructs are created via drag-and-drop from a palette without keyboard interaction.

In this paper, we describe a combination of these two approaches: a structured editor that is generated from a language grammar but supports entry of language constructs through keyboard through use of a modified parser for that grammar. For example, assume a syntax tree in JavaScript for the number 23, and the plus sign as keyboard input by the user, as shown in Figure 1. Since the JavaScript grammar knows two constructs that start with a number followed by a plus sign, our parser produces two parse trees from the partial input. The user may resume typing, for example the number 5, at which point the input would be unambiguous and our parser would produce a single parse tree for the expression 23+5. Thus, our modified parser allows the construction of a binary addition via keyboard, as opposed to drag-and-drop.

In the following, we will first discuss means of interactions in structured editors in general. We will then present our modified parser and an evaluation of its suitability for structured editing. Finally, we present related work and conclude the paper.

2 Interactions in Grammar-derived Structured Editors

In this section, we first describe the structure of Tree-sitter grammars, which our approach uses to auto-generate mappings from general-purpose programming language grammars to structured editors. We then present different means to interact with structured editors to illustrate the design space relevant to providing input via our partial parser.

2.1 From Tree-sitter Grammars to Structured Editing

For our example implementation of the described approach, we chose to use grammars as expressed in the Tree-sitter parser generator's grammar language [5]. Tree-sitter is an suitable target for automatic generation of editors as grammars for a large set of commonly used programming languages exist, including C, C++, Python, or JavaScript [5]. The Tree-sitter parser generator encourages authors to formulate grammars with a close correspondence to the language constructs users have in mind, supporting the resulting parse tree's use in other tools. For this, it offers, for example, constructs to model associativity and precedence, such that these do not have to be encoded in the grammar's structure.

Listing 1 shows an example of a grammar in Tree-sitter's DSL. Tree-sitter's DSL is built on top of JavaScript. By specifying `prec.left` around a rule, we can mark this rule as left-associative and optionally supply a numerical precedence. The rule names that appear in the supertypes list will not generate visible nodes in the resulting AST. For a full description of Tree-sitter's DSL refer to its documentation[5]. We will explain all elements relevant to our approach and relate to other common specifications.

```
/* Tree-Sitter DSL (written in JavaScript): */
module.exports = grammar({
 name: 'example',
 supertypes: $ => [$.expr],
 rules: {
  expr: $ =>
   choice($.add,$.mul,$.id,$.closure,$.new),
  id: $ => /\w+/,
  new: $ => prec.left(seq("new",$.expr)),
  add: $ => prec.left(1,seq($.expr,"+",$.expr)),
  mul: $ => prec.left(2,seq($.expr,"*",$.expr)),
  closure: $ => prec.left(seq($.id,"=>",$.expr)),
 }
});

/* Approximate pendant in more common syntax:
expr := add | mul | id | closure | new
id := \w+
new := "new" expr
add := expr "+" expr
mul := expr "*" expr
closure := id => expr
*/
```

Listing 1. An example grammar written in the Tree-sitter DSL and the same grammar expressed in a more conventional style. `prec.left()` marks a left-associative rule, optionally with a numeric precedence.

To map from a grammar to a structured editor's user interface, a straightforward mapping may for example turn non-terminals to block compounds and terminals to text fields [22]. In our example implementation, we show tree structures through nesting of blocks, similar to prior work on a structured editor for Smalltalk [3]. Figure 2 shows an example from a Tree-sitter parse tree and its derived user interface. Every user interface element stores a reference to the rule in the grammar that is was derived from. Thus, a

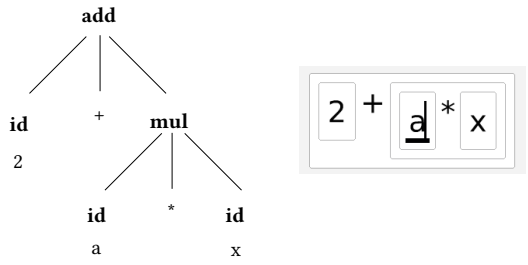

Figure 2. Mapping from a Tree-sitter parse tree to our structured editor's user interface. Each node in the tree (marked bold) is turned into a block. Terminals (non-bold text) are turned into textfields that are also embedded in the corresponding blocks. The user's cursor is shown on the right in the a block signified by a bold underline to show the selected node and a caret to show the precise text cursor's position.

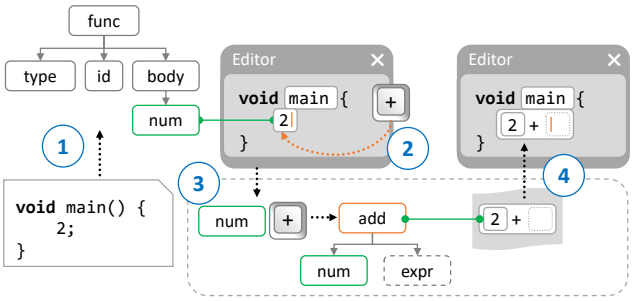

Figure 3. Overview of our structured editor system. In (1), the Tree-sitter parser maps from a source file to a syntax tree. In (2), the user adds a plus sign to the number node. In (3), our partial parsing system reconciles this input with the surrounding tree structure and produces a new subtree. In (4), we replace the old subtree with the updated subtree.

text field can tell if new input provided by the user conforms to its allowed characters.

As a result, the blocks form a bijective mapping from the AST of a textual program to a user interface. By walking the tree and collecting values from text fields, the blocks can be serialized back into a textual form for storage purposes. Some aspects, such as comments or deliberate formatting decisions require additional considerations to "survive" the mapping process but are not relevant for a discussion of the input system.

2.2 Interactions in Structured Editors

Structured editors with an appearance close to text editors, including Barista [12] or editors using GrammarCells [24], as well as our own system, tend to ensure a close mapping to the interactions that users are already familiar with from

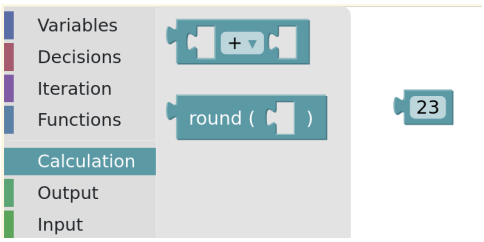

Figure 4. Palette and number block in BlockPy [1], a structured editor built using Blockly for Python. Users drag-and-drop blocks from the palette on the left onto the code canvas on the right to construct programs.

textual editing. For example, typing an expression such as `2 + 3 * num` should be possible with the same keystrokes as in a text editor but automatically construct the corresponding tree.

In contrast to this, block-based editors typically feature a block palette that allows users to create language constructs via drag-and-drop, see Figure 4. This forms a trade-off between intuitiveness and editing efficiency, when compared to text-like editing.

A structured editor with a focus on text-like editing may instead ask users to start typing out the textual equivalent of the language construct they want to create, for example the letters `for` for a loop. At this point, the input is still ambiguous: users may want to continue typing to create an identifier such `foreign` or type a space character, at which point the system can conclude that the keyword `for` is meant.

Perhaps intuitively we enable these interactions through analysis of the language's grammar. We identify all language constructs in the grammar that may start with the given input and ask users to continue typing until only one construct is left or they explicitly disambiguate between choices through a user interface.

In a parser that acts on complete files, this type of lexical conflict would typically be resolved by looking ahead in the input stream. As our system acts on live user input, this is not possible and thus our system presents all valid options up to this point, until the user provides further input or explicitly selects an option. This differs from autocompletion as seen in IDEs, as no information on the semantics of the program are involved. Instead, it more closely resembles snippets that automatically complete the boilerplate required for, e.g., an if-statement when the user invokes a shortcut, but generalized across the entire grammar.

Figure 3 provides a view on the complete system from source file, over a change by the user, to the updated tree. In (1), the standard Tree-sitter parser takes the input file and creates a corresponding syntax tree. This tree is bijectively mapped to a user interface. In (2), we illustrate a change: the user has the number 2 selected and presses the plus button. In response, our partial parsing system is activated

in (3). In the example, our system finds that the add rule incorporates both the existing number node and the new plus input character and proposes this as the new subtree to be used. Since no other subtrees matched, we can directly update the trees, as shown in (4), resulting in the updated program.

3 Partial Parsing

In the following, we will first describe our partial parser as a set of modifications to a recursive descent parser. We will then detail considerations and heuristics we added to the parsing process and invocations of the parser to support a familiar, keyboard-centric editing flow.

For ease of implementation of the modifications described in the following, we designed the baseline parser using recursive descent. The parser we are starting from before applying our modifications accepts most context-free grammars as described by Tree-sitter's grammar notation. Note that our contribution is a set of practical adaptations to a recursive descent CFG parser to support interactive editing in structured editors, rather than the demonstration of a formally equivalent parser that accepts all inputs as the original Tree-sitter parser.

We formulate our modifications in pseudo code. For example, take the below definition of the implementation of the choice operator:

```
def choice.apply(parser):
  results := new List()
  for alt in this.alternatives:
    results.addAll(alt.apply(parser.copy()))
  return results
```

We take the parser state as argument, apply each operator in the choice's alternatives to a copy of the state, and return the new list of resulting parser states. The parser state consists of

1. the input `stream` that can be accessed via its `peek` and `next` methods,
2. the current subtree, to which we can add nodes via the `addParseNode` or `addPlaceholder` (to construct a hole with the given grammar rule) methods, and
3. fields required to manage recursion and memoization.

To support left-recursion as used in Tree-sitter grammars in our recursive descent parser, we make use of seed growing [27]. Seed growing exploits the memoization table that stores parse tree results per rule and per index of the input stream. When a rule is first encountered, a memoization entry is created as a marker. If the same rule is invoked, the marker gets flagged as a left recursion and returns no results and as a consequence skips to the next alternative that includes no left recursion. Finally, when the recursive descent returns to the first invocation of the rule and a left recursion was flagged, we evaluate the same rule again. This time, when the recursion would occur, there is already an entry in the memo table, for the smallest possible parse of the recursive rule, the seed. By repeatedly evaluating the recursive rule while the parser still advances, we thus iteratively grow the seed until the input that constitutes the left-recursive rule is consumed.

3.1 Partial Input

Our first modification should enable our parser to receive a partial input and return all parse trees that are valid considering just this input. For example, given the grammar shown in Listing 1 and the input `n`, the following are valid results in this first version of our recursive descent parser:

```
n         (id)
n + _     (add)
n * _     (mul)
n => _    (closure)
new _     (new)
```

Listing 2. An exhaustive list of results for the input `n`. The underscore symbolizes a hole that awaits further input through the user.

Observe that the letter `n` takes on the role of an `id` node in the first four parse trees, in the first three it is also an expression, whereas in the last it is part of a larger token.

To illustrate our understanding of partial input and partial results, consider the input stream `2+` for our example grammar in Listing 1. Normally, a parser would attempt to form an `add` subtree but reject the input as the right-hand side of the binary operation is missing. However, at least for our example grammar, this input already unambigiously signals the user's intent to construct an `add` node with an `id` of `2` in the left-hand side. To provide the benefits of structural editing, we want to commit to this tree as early as possible and not wait for a complete input.

The below code illustrates our recursive descent parser's current procedure for evaluating sequences. The algorithm maintains a queue of parser states and the index of the sequence's elements this state reached. In each iteration, we advance a parser state from the queue and add all resulting, unfinished parser states to our queue, until the queue is empty and we return all states that reached the end of the sequence's elements.

```
def sequence.apply(parser):
  results := new List()
  queue := new Queue()
  # queue a tuple combining the parser state and
  # the current index in our sequence
  queue.push((parser, 0))
  while queue.notEmpty():
    (parser, i) := queue.pop()
    # this.elements is the list of elements this
    # sequence is made up of; here we get all
    # parse trees for its i'th element
    candidates := this.elements[i].apply(parser)
    for parser in candidates:
      if not parser.atEnd():
        queue.push((parser, i + 1))
      else:
        if i == this.elements.size:
          results.add(parser)
```

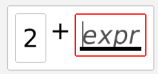

Figure 5. The result of entering 2+, completed `id` and add nodes, and an incomplete placeholder waiting for an `expr` node to be entered.

```
19    return results
```

Listing 3. Unmodified procedure for evaluating sequence operators.

To allow our parser to accept incomplete input, we need to adapt its handling of sequences. At the moment, we require to have found results for each element of the sequence in Line 14. Instead, we modify the if-conditional to keep track of all parser states that exhausted their input stream as checked in Line 13 but did not reach the end of the sequence in a separate list (Line 16 below).

```
12        ...
13        if not parser.atEnd():
14          queue.push((parser, i + 1))
15        else:
16          if i == this.elements.size:
17            results.add(parser)
18          else:
19            incompleteResults.add((parser, i))
20        ...
```

Finally, before returning the results, we complete the incomplete results with placeholder blocks as shown in Figure 5, starting from the first index where our parser had reached its end but there were still pending elements of the sequence. Once completed, we add them to our results list.

```
19    ...
20    for (parser, lastIndex) in incompleteResults:
21      for index in lastIndex+1..this.elements.size:
22        parser.addPlaceholder(this.elements[index])
23      results.add(parser)
24    return results
```

With these changes to our sequence code, we accept the first four results shown in Listing 2. To accept the fifth, we also need to adapt our implementation of the text operator shown below. In the unmodified implementation, we match the text operator's regular expression against the remaining stream in the parser state and construct a leaf parse node for this match if it succeeded.

```
1 def text.apply(parser):
2   if match = regex.matches(parser.stream):
3     parser.addParseNode(new ParseNode(match))
4     return parser
5   return new List()
```

Instead of matching the text operator's entire regular expression, we allow only matching a prefix if the match would bring us to the end of the input stream:

```
1 def text.apply(parser):
2   if match = regex.matches(parser.stream) or
3       (match = regex.matchesPrefix(parser.stream)
4         and parser.atEnd()):
5     text = match + this.remainingText(match.size)
6     parser.addParseNode(new ParseNode(text))
7     ...
```

With this change, given a label such as `new` and the input `ne`, we match the operator and autocomplete the text to `new` in Line 5.

3.2 Minimizing the Results Set

Once our parser has produced the exhaustive list of (partial) parse trees that are valid for a given input, we want to trim that list to only those that we deem relevant to the user. Considering the example in Listing 2, we argue that only two results are of relevance:

```
1 n         (id)
2 new _     (new)
```

To the user, it is clear that all other parse trees can be obtained by extending the `n` that gets parsed as an `id` by typing for example a plus sign afterwards. Contrarily, the `new` is its own language construct.

Note that it would be equally valid to also eliminate the `new` and allow users to enter the `new` rule by fully typing out the keyword. When testing, we found that it was less jarring to now jump between an accepted identifier and a larger construct started by a keyword. For example, consider the case where the user wants to type the identifier `newHouse`. When entering `n`, the parser would commit to the `id` type. Once the user has typed `new`, however, it would then enter an ambiguous state again, not knowing whether the user wants to continue typing or designate the keyword, until they complete the identifier and remove the remaining ambiguity.

As such, the heuristic we apply for this minization step can be expressed as: Eliminate all parse trees that can be obtained by extending the chosen parse tree when the user adds more input. With this rule, all resulting parse trees where the last input produced an `id` node are grouped.

To implement the grouping step, we first add a marker for the last parse node that was constructed, i.e., the parse node that consumed the last of the user's input string. From the marker, we obtain the list of its parent nodes in the parse tree.

For the input `n`, the resulting set is as follows:

```
1 id                    (n)
2 id -> add             (n + _)
3 id -> mul             (n * _)
4 id -> closure         (n => _)
5 "new" -> new          (new _)
```

Next, we discard all results that have another chain as prefix. Thus, results 2, 3, and 4 are discarded since the chain of result 1 is their prefix.

Finally, consider the input string `new`. Both the `id` rule and the `new` rule will match. However, since the latter would be interpreted as a keyword in a textual language, we want to discard the id option. As such, we check if our last constructed node is a keyword in the grammar. If that is the

case, we only permit results where the last constructed node is also a keyword.

Keywords. Earlier, we established as a heuristic to only merge if the same language construct was derived from the input. For example, we do not want to merge `new _` and `n` but we do want to merge `n` and `n+_`. During our own usage with the system we noticed some exceptions to this rule. In the Python grammar, for example, there are separate rules for integers and floats. As such, when the user begins typing a number, the editor will never be able to unambigiously commit to an integer, as integer literals always form a prefix for the float rule. For lack of a generalizable pattern with these exceptions, we are currently annotating pairs of rules that should be merged even though they map to separate language constructs manually.

3.3 Maintaining Existing Subtrees

For purposes of tool building it can be desirable to keep as much of the existing subtrees as possible. For example, it allows storing references to the nodes directly, rather than working with identifiers. Further, by reusing entire subtrees, we can improve performance while parsing, similar to the method incremental parsers employ.

To maintain existing subtrees in our partial parser, we extend the parser to accept an input stream that may contain both characters as well as existing parse nodes. The extension simply requires our non-terminal to not just apply its body but to first peek the input stream to check whether the next element is a parse node. If so, we compare the non-terminal's grammar rule with the parse node's grammar rule and return a parser state that incorporates the existing parse tree on a match.

For example, consider an existing tree `2*3+4` and a change by the editor's user to replace the `+` to be a `*`. In this scenario, we provide our parser with an input stream consisting of a `mul` node (`2*3`), the character `*` that the user just typed, and an `id` node for the `4`. The resulting tree `2*3*4` will reuse the two nodes that were passed to the parser in the entirety and only replaces the root node, as it changed its semantics from `add` to `mul`.

In other scenarios a reinterpretation on the character level is inevitable: given the `id` node `321456`, the user may want to split the number into the expression `321+456`. Here, the query to the parser would be a plain character stream that replaces the current root node with three entirely new nodes.

3.4 Querying the Parser from the Editor

As outlined in the previous subsection, the parser will receive a mix of characters and existing subtrees as inputs. Given a character typed by the user, we have to decide what parts of the tree we want to keep and what parts may need to be "stringified" before being passed to the parser. This concern is not directly related to our partial parsing system but poses challenges related to the grammar concerning precedence and subtree reuse. As such, we briefly outline how the parser is triggered from a user's point of view, referred to as *query to the parser*, before discussing precedence and subtree reuse.

Queries to the Parser. Generally, when the user types a character, a number of queries are placed to the parser starting from the deepest node and traveling up the parent hierarchy until a query returns at least one result. If it is exactly one result, we apply the restructured parse tree. In case of multiple results, the user may continue typing until the input becomes unambiguous or interactively chooses one of the presented options, as shown for example in Figure 1. The process of incorporating user input is as follows, proceeding to the next step only if the previous step did not already produce results:

1. When a new input character is received, the focused node will try to take the input as part of its **text field**, if it would then still conform to the grammar.
2. Next, if the input occurred at the very start or end of the text field, we take the node and insert the input before or after the entire node in the stream. This enables typing a `+` sign on an existing number, transforming a `2` node into the node `2+_` that **wraps** the existing number node.
3. Next, the focused node will **stringify** its contents, insert the character in the stream at the cursor position and reparse the node. For example, users can enter a `:` after a Smalltalk unary message to change it to a keyword: `3 raised` to `3 raised: _`.
4. Finally, if none of the above steps produced a result, we repeat all steps with the **parent** node of our selection.

Precedence and Left-recursion. A challenge with this approach arises in the presence of precedence, especially in left-recursive constructs. As an example, take a grammar that contains both the C postfix increment operator and binary addition.

```
1 expr := add | inc
2 add := expr "+" expr
3 inc := expr "++"
```

We define that the postfix increment binds more strongly to its expression than the binary addition. Thus, when the user types a plus sign on the `3` of an expression such as `1+2+3`, the system will immediately commit to `1+2+3++` since we find a single valid parse tree that consumes the entire input. Instead, we would have expected a conflict between the described choice and `1+2+3+_`, which did not occur because, to reach this second parse tree, the system would have needed to traverse to the parent twice. See Figure 6 for an illustration of the issue.

To address this, we modify the third step in the previously described process: if we find that we are on the right-hand side of a left-recursive rule, we query results for both the

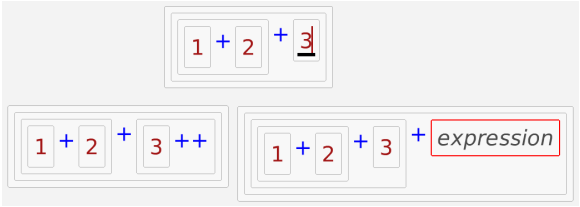

Figure 6. Our initial expression on the first line. Given the input +, we want to be able to obtain both of the below parse trees. Note the visual appearance of the editor, which may suggest editing interactions akin to block-based editors. However, through use of the partial parser, editing interactions are designed to mimic text editing, while the visual appearance aims for clear boundaries between syntactic elements.

Table 1. Projects from which we sampled the 30 most recent single line changes for our evaluation. We selected popular repositories of medium size that were active at the time of writing, for each language. The reported lines of code and number of files include only files of the language we evaluated.

Project	Language	Lines of Code	Files
flask[3]	Python	10372	75
express.js[4]	JavaScript	16381	153
Overtone[5]	Clojure	30986	282

active node and its parent. Thus, rather than stopping with the results from just the active node, we ensure that we get both options as displayed in Figure 6.

4 Evaluation

Since our goal was to improve parsing in an interactive context, we study the end-to-end performance of typing code fragments that could have been written realistically by programmers. We created a dataset of such edits by extracting small code changes from the public version history of the software projects listed in Table 1. More precisely, we extracted the most recent 30 source file[1] changes that inserted or edited one line of code. The changes are supposed to resemble those a user may have performed step-by-step in an interactive programming session. After reverting the change, we let our system automatically repeat the change and verify that the resulting program source is equivalent. Both the underlying editor, as well as the implementation of the partial parser are available open-source on Github [2].

4.1 Re-performing Changes

As an example, consider the following single line change from the express.js repo where the below statement was inserted:

```
mock.uri.params = mock.uri.params || {}
```

Figure 7 illustrates one such change as an example. To prepare the edit, we first locate the corresponding subtree in the final file's source tree. We then replace this subtree with a hole and place the cursor inside, as shown in (1) of Figure 7. Now, we begin replaying the source line character by character as inputs to our system. In (2), we simulated typing the first identifier, which replaces the hole initially. When the dot is typed, in (3), we wrap the identifier in a member access and the cursor awaits further input for the property. We continue typing until in (4) we hit the first ambiguous input: the equal sign could either be an assignment or a comparison. Once the user presses the next letter in (5), we know that it has to be an assignment and automatically apply the change. Finally, in (6), we are left with another ambiguity that does not resolve as both options share the same prefix. The input could either be a JavaScript object, or a JavaScript object destructuring assignment. To resolve this case, we automatically press return once if we reached the end of the input and the parser still produces multiple valid options.

While going over the first 30 changes in each repository, we skipped changes if they concerned only comments or Python comment strings. Further, 11 subtrees in the first 30 of the Python/flask data set could not be automatically reproduced with our current system and were thus skipped. All 11 concerned `import` statements of the form `import name from module`, where the `from` keyword was autocompleted and thus re-typed by the automatic replay. Note that manually reproducing all 11 statements using our system was possible, by simply skipping over the autocompleted `from` token.

4.2 Performance

We measured the end-to-end time each processing after a keystroke took. Note that for a single response, there may have been multiple queries to the partial parser, as described in subsection 3.4. In total, during the experiment, we processed 1233 keystrokes. We report the time from the first query to the response of the last query in Figure 8.

The times were measured on a laptop with a 12th Gen Intel i7-1255U on Ubuntu 22.04 LTS. The implementation is written in Squeak/Smalltalk [10], and exploratory-style live programming environment [16], and thus not heavily optimized for performance.

[1] Identified by file extension, excluding configuration files.
[2] https://github.com/hpi-swa-lab/sb-tree-sitter/
[3] https://github.com/pallets/flask/tree/36af821ed
[4] https://github.com/expressjs/express/tree/2c4782705
[5] https://github.com/overtone/overtone/tree/ddb4046c3

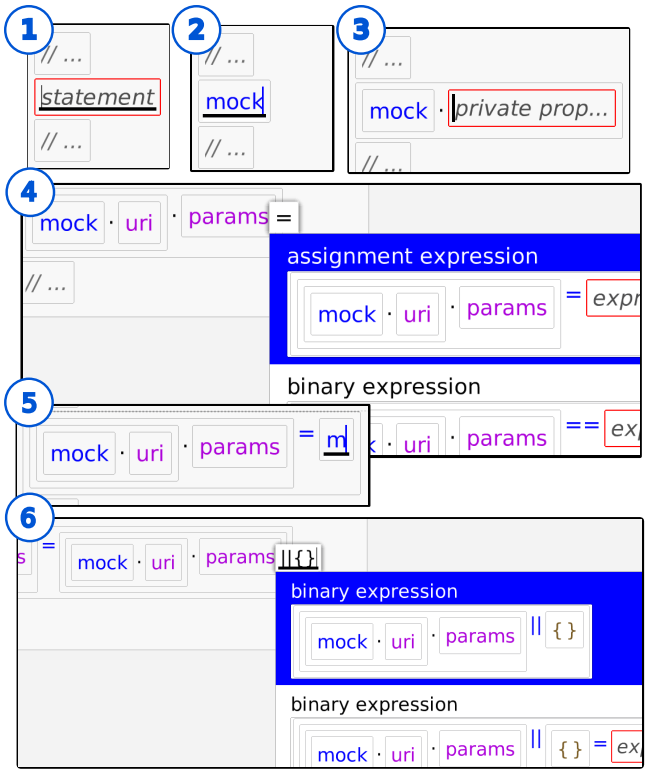

Figure 7. Walkthrough of an example change as performed for our evaluation. The user is typing the expression `mock.uri.params = mock.uri.params || {}`. The popup in steps four and six occurs when the input is ambiguous.

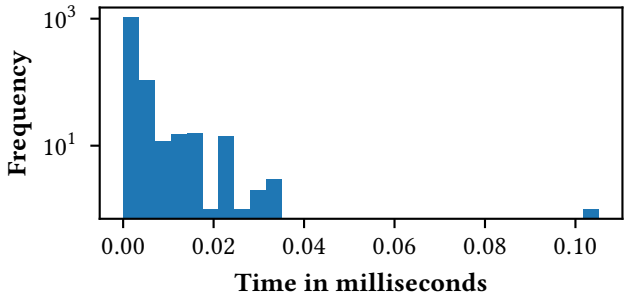

Figure 8. Histogram of end-to-end response times for all 1233 parser queries in response to keystrokes in our evaluation. The x-axis reports time in milliseconds, the y-axis frequency per bin on a logarithmic scale. The highest displayed value is at around 0.105 milliseconds.

For an interactive use case, processing keystrokes should ideally not take more time than the user leaves between two physical keystrokes. Assuming a user typing at 120 words per minute and an average word length of five, an upper bound may thus be 0.06 seconds or 60 milliseconds per character[6]. The highest measured duration in our sample set 0.105 milliseconds and thus far below this limit.

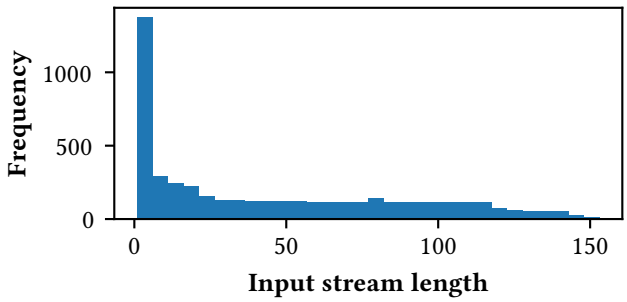

Figure 9. Histogram of the lengths of input streams in our evaluation. In total, this includes 4745 values, corresponding to all parsing operations that were started.

The fast response times are of course to a large part due to committing to a subtree as early as possible and reusing those subtrees in their entirety in subsequent queries. We report the input stream lengths to our parser used in our evaluation in Figure 9, the distribution's median is 30 characters or nodes in the input stream. As such, most inputs are comparatively small compared to the lengths of typical source code files.

4.3 Threats to Validity

By using historical code changes, we increase the ecological validity of our study at the expense of a controlled setting, i.e., not all language features might have been used and their frequency reflects the particular coding style of the projects' authors. Using single-line changes omits composite changes and refactorings, but reconstructing a realistic sequence of editing operations from a large diff is out of scope.

4.4 Generalizability

In subsection 3.2, we briefly mentioned that we manually tweaked the Python grammar to resolve ambiguity between the integer and float rule. It is to note that these changes improve the user experience but are not required to produce a usable editor.

Our approach is implemented on top of Tree-sitter grammars. More generally, the described essential idea of modifying parsers to allow partial input to support structured editing is compatible with arbitrary grammar formalisms that describe textual programming languages. In Tree-sitter, only the sequence and text operators may produce boundaries at which partial input can terminate and require completion

[6] 120 words/min * 5 characters/word = 1000 characters/minute, 60sec / 1000 characters/minute = 0.06sec

with empty elements. A different grammar formalism may introduce additional such boundaries.

Importantly, the quality of the generated editors will vary with the closeness of mapping between the way users think about a programming language and the way rules are expressed in the grammar. For example, if the grammar formalism does not support an explicit construct for precedence, a preprocessing step such as the one described in prior work [21] to collapse the grammar rules may improve usability, as details of the language's implementation in the grammar are hidden.

As such, the quality of the resulting editor will depend less on the type of formalism used but rather how well the formalism can express the language it is describing.

5 Related Work

Several previous projects have been built around the idea of using a custom parsing technique to improve how users interact with a structured editor.

One approach called *substring parsing* [17] uses a similar parsing technique based on exploring all possible parse trees given a partial input. As with our approach, the approach produces parse trees that contain non-terminals that have no children and represent holes that users may fill. However, the algorithm is only described in general terms, making it difficult to apply it directly or evaluate its properties. Also, while the substring parsing approach was primarily motivated by structured editors, the resulting challenges are not explored. For instance, there is no discussion about whether the approach can exhaustively cover potential parse trees. Further, the approach does not integrate the existing AST shown in the editor but parses the input in isolation, "stringifying" nodes eagerly.

The technique of substring parsing is also used in another project, implementing partial parsing for LR parsers [19]. In contrast to our approach, this approach does also not incorporate the existing AST shown in the editor. Further, there is not characterization of the properties of the resulting parsers, neither with regard to exhaustiveness of parse trees nor with regard to performance characteristics.

Beyond these individual works, the three parsing techniques error recovery, island parsing, and incremental parsing are all similar to our approach with regard to their goals or mechanisms.

Our approach is similar to error recovery techniques in parsing, as both enable parsers to continue in the presence of incomplete user input [6]. However, the resulting parse trees differ. Error recovery techniques aim to produce parse trees with placeholders for errors or repairs, while our approach produces parse trees with non-terminals in places of incomplete input. Further, while error recovery typically aims to contain the error and proceed parsing, we aim to incorporate the incomplete input as good as possible into potential parse trees.

Similarly, our approach is similar to island parsing in that it produces incomplete parse trees [15]. The goal of island parsing is to speed up parsing in scenarios in which users do not need a parse tree that corresponds to the original language, for example when only extracting variable writes from a large corpus of source code. Thus, island parsing ignores parse trees with nodes representing ignored sections of the input, while our approach produces parse trees with valid non-terminals representing potential future extensions of the input.

Our approach and incremental parsing techniques share the idea of re-using information from a previous parse. Typical incremental parsing techniques aim to speed up parsing by re-using information, such as the memoization table [7] or the parse tree [26]. For example, the Tree-sitter incremental parsing algorithm takes the previous parse tree and the code change. Based on the code change the algorithm determines non-affected parse nodes in the tree and skips them, thus only parsing sections relevant to affected nodes [26]. Similar to incremental parsing, our approach re-uses information from an existing parse tree. However, incremental parsing aims to make parsers determine the correct parse tree faster, while our approach aims to produce all possible parse trees given incomplete user input.

In Rascal2MPS [13] a Rascal language grammar is used to generate concepts for the projectional language workbench MPS. Rather than employing the rules of the grammar for interactions directly, Rascal2MPS generates definitions for MPS once. The resulting interactions follow the standard for MPS editors, with a mixture of autocompletion and menus, which may not always conform to the expectations users have from text-based editing.

Grammar Cells [24] present an approach to provide hints to a language specification in MPS on how keyboard interactions should restructure the program tree, to make interactions resemble textual editing more closely. These hints have to be manually provided by the language author. Similarly, Barista [12] is an implementation framework for structured editors, where interactions are derived from hand-written classes for each language construct. Both Grammar Cells and Barista employ a comparable strategy of linearizing tokens in a subtree when textual input occurs in a node and performing a re-parse but rely on hand-written specifications.

Our work can be seen as an extension of the work on Kogi [21, 22], which generates structured editors from existing language grammars. In the context of Kogi, issues in terms of usability were identified when generating editors from general-purpose programming languages, as the number of language constructs posed challenges with the menu and mouse-based interactions. Our approach thus makes usage of generated structured editors for general-purpose programming languages feasible, given that the user already

knows the language's syntax and can thus create the elements via the keyboard.

6 Conclusion and Future Work

We demonstrated that through our modifications to a recursive descent parser, our partial parser was able to type all language constructs we have examined in our evaluation. We further showed that a significant portion of nodes can be reused during the parsing stage, allowing tools to maintain references to the existing structures. Through a performance evaluation we demonstrated that during typical use our parser is more than fast enough to support interactive use.

As such, we presented a parsing system that is capable of taking in existing language grammars without requiring manual annotations and producing an input system that allows users to interactively formulate any language construct in a manner familiar to users from textual editing. This forms an important aspect for making structured editing compatible with existing language ecosystems, thanks to large language grammar repositories such as those provided by Tree-sitter.

In future work, we plan to evaluate the heuristics described to reach a familiar editing experience in a user study. Importantly, to reach a fully functioning structured editor, further work is required that is not integral to the partial parser concept. For example, backspacing to delete structures step-by-step is a commonly used means for deletion in textual editing.

In its current form, our partial results are constrained to the right edge of the input stream. In some circumstances it may also be desirable to construct incomplete structures in the middle or left edge of the input stream. For example, given the JavaScript expression `42` and a prefix of `const a`, the system may automatically derive the user's intent to formulate a declaration of the form `const a=42`, autocompleting the equal sign in the middle of the input stream.

Acknowledgments

This work was supported by Deutsche Forschungsgemeinschaft (DFG) grant #449591262. We also gratefully acknowledge the financial support of HPI's Research School[7] and the Hasso Plattner Design Thinking Research Program[8].

References

[1] Austin Cory Bart, Javier Tibau, Eli Tilevich, Clifford A. Shaffer, and Dennis Kafura. 2016. Implementing an Open-Access, Data Science Programming Environment for Learners. In *2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*, Vol. 1. 728–737. https://doi.org/10.1109/COMPSAC.2016.132

[7] https://hpi.de/en/research/research-school.html

[8] https://hpi.de/en/dtrp/

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

[3] Tom Beckmann, Stefan Ramson, Patrick Rein, and Robert Hirschfeld. 2020. Visual Design for a Tree-Oriented Projectional Editor. In *Conference Companion of the 4th International Conference on Art, Science, and Engineering of Programming* (Porto, Portugal) *(<Programming> '20)*. Association for Computing Machinery, New York, NY, USA, 113–119. https://doi.org/10.1145/3397537.3397560

[4] Thorsten Berger, Markus Völter, Hans Jensen, Taweesap Dangprasert, and Janet Siegmund. 2016. Efficiency of projectional editing: a controlled experiment. In *FSE 2016*. 763–774. https://doi.org/10.1145/2950290.2950315

[5] Max Brunsfeld. 2020. Tree-sitter. https://tree-sitter.github.io/tree-sitter/. https://tree-sitter.github.io/tree-sitter/ [Online, accessed 10 August 2022].

[6] Lukas Diekmann and Laurence Tratt. 2020. Don't Panic! Better, Fewer, Syntax Errors for LR Parsers. In *34th European Conference on Object-Oriented Programming, ECOOP 2020, November 15-17, 2020, Berlin, Germany (Virtual Conference) (LIPIcs, Vol. 166)*, Robert Hirschfeld and Tobias Pape (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 6:1–6:32. https://doi.org/10.4230/LIPIcs.ECOOP.2020.6

[7] Patrick Dubroy and Alessandro Warth. 2017. Incremental packrat parsing. In *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2017, Vancouver, BC, Canada, October 23-24, 2017*, Benoît Combemale, Marjan Mernik, and Bernhard Rumpe (Eds.). ACM, 14–25. https://doi.org/10.1145/3136014.3136022

[8] Google. 2020. Blockly. https://developers.google.com/blockly. https://developers.google.com/blockly [Online, accessed 10 August 2022].

[9] Pedro Rangel Henriques, Maria João Varanda Pereira, Marjan Mernik, Mitja Lenic, Jeff Gray, and Hui Wu. 2005. Automatic generation of language-based tools using the LISA system. *IEE Proc. Softw.* 152, 2 (2005), 54–69. https://doi.org/10.1049/ip-sen:20041317

[10] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. 1997. Back to the Future: The Story of Squeak, a Practical Smalltalk Written in Itself. In *Proceedings of the 12th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Atlanta, Georgia, USA) *(OOPSLA '97)*. Association for Computing Machinery, New York, NY, USA, 318–326. https://doi.org/10.1145/263698.263754

[11] Amy Ko, Htet Aung, and Brad Myers. 2005. Design requirements for more flexible structured editors from a study of programmers' text editing. 1557–1560. https://doi.org/10.1145/1056808.1056965

[12] Amy J. Ko and Brad A. Myers. 2006. Barista: An implementation framework for enabling new tools, interaction techniques and views in code editors. In *Proceedings of the 2006 Conference on Human Factors in Computing Systems, CHI 2006, Montréal, Québec, Canada, April 22-27, 2006*, Rebecca E. Grinter, Tom Rodden, Paul M. Aoki, Edward Cutrell, Robin Jeffries, and Gary M. Olson (Eds.). ACM, 387–396. https://doi.org/10.1145/1124772.1124831

[13] Mauricio Verano Merino, Jur Bartels, Mark van den Brand, Tijs van der Storm, and Eugen Schindler. 2021. *Projecting Textual Languages*. Springer International Publishing, Cham, 197–225. https://doi.org/10.1007/978-3-030-73758-0_7

[14] Mauricio Verano Merino, Jurgen Vinju, and Mark van den Brand. 2021. DRAFT-What you always wanted to know but could not find about block-based environments. (2021). arXiv:2110.03073 [cs.SE] https://arxiv.org/abs/2110.03073 [Under review at ACM Computing Surveys].

[15] Leon Moonen. 2001. Generating Robust Parsers Using Island Grammars. In *Proceedings of the Eighth Working Conference on Reverse Engineering, WCRE'01, Stuttgart, Germany, October 2-5, 2001*, Elizabeth Burd, Peter Aiken, and Rainer Koschke (Eds.). IEEE Computer Society, 13. https://doi.org/10.1109/WCRE.2001.957806

[16] Patrick Rein, Stefan Ramson, Jens Lincke, Robert Hirschfeld, and Tobias Pape. 2019. Exploratory and Live, Programming and Coding - A Literature Study Comparing Perspectives on Liveness. *The Art, Science,*

and Engineering of Programming 3, 1 (2019), 1. https://doi.org/10.22152/programming-journal.org/2019/3/1

[17] Jan Rekers and Wilco Koorn. 1991. Substring parsing for arbitrary context-free grammars. *ACM SIGPLAN Notices* 26, 5 (may 1991), 59–66. https://doi.org/10.1145/122501.122505

[18] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, and et al. 2009. Scratch: Programming for All. *Commun. ACM* 52, 11 (Nov. 2009), 60–67. https://doi.org/10.1145/1592761.1592779

[19] Gregor Snelting. 1990. How to build LR parsers which accept incomplete input. *ACM SIGPLAN Notices* 25, 4 (apr 1990), 51–58. https://doi.org/10.1145/987481.987485

[20] Tamás Szabó, Markus Voelter, Bernd Kolb, Daniel Ratiu, and Bernhard Schaetz. 2014. Mbeddr: Extensible Languages for Embedded Software Development. *Ada Lett.* 34, 3 (Oct. 2014), 13–16. https://doi.org/10.1145/2692956.2663186

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

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

[23] Markus Voelter. 2011. Language and IDE Modularization, Extension and Composition with MPS. *GTTSE 2011* 7680 (07 2011). https://doi.org/10.1007/978-3-642-35992-7_11

[24] Markus Voelter, Tamás Szabó, Sascha Lisson, Bernd Kolb, Sebastian Erdweg, and Thorsten Berger. 2016. Efficient Development of Consistent Projectional Editors Using Grammar Cells. In *Proc. SLE* (Amsterdam, Netherlands) *(SLE 2016)*. ACM, New York, NY, USA, 28–40. https://doi.org/10.1145/2997364.2997365

[25] Markus Völter, Janet Siegmund, Thorsten Berger, and Bernd Kolb. 2014. Towards User-Friendly Projectional Editors. In *SLE 2014*. 41–61. https://doi.org/10.1007/978-3-319-11245-9_3

[26] Tim A. Wagner and Susan L. Graham. 1998. Efficient and Flexible Incremental Parsing. *ACM Trans. Program. Lang. Syst.* 20, 5 (1998), 980–1013. https://doi.org/10.1145/293677.293678

[27] Alessandro Warth, James Douglass, and Todd Millstein. 2008. Packrat Parsers Can Support Left Recursion. *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*. https://doi.org/10.1145/1328408.1328424

Received 2022-08-08; accepted 2022-09-30