Fast Incremental PEG Parsing

Zachary Yedidia
Harvard University
USA

Stephen Chong
Harvard University
USA

Abstract

Incremental parsing is an integral part of code analysis performed by text editors and integrated development environments. This paper presents new methods to significantly improve the efficiency of incremental parsing for Parsing Expression Grammars (PEGs). We build on *Incremental Packrat Parsing*, an algorithm that adapts packrat parsing to an incremental setting, by implementing the memoization table as an interval tree with special support for shifting intervals, and modifying the memoization strategy to create tree structures in the table. Our approach enables reparsing in time logarithmic in the size of the input for typical edits, compared with linear-time reparsing for Incremental Packrat Parsing. We implement our methods in a prototype called GPeg, a parsing machine for PEGs with support for dynamic parsers (an important feature for extensibility in editors). Experiments show that GPeg has strong performance (sub-5ms reparse times) across a variety of input sizes (tens to hundreds of megabytes) and grammar types (from full language grammars to minimal grammars), and compares well with existing incremental parsers. As a complete example, we implement a syntax highlighting library and prototype editor using GPeg, with optimizations for these applications.

CCS Concepts: • **Software and its engineering → Parsers**.

Keywords: incremental parsing, PEG, packrat parsing

ACM Reference Format:
Zachary Yedidia and Stephen Chong. 2021. Fast Incremental PEG Parsing. In *Proceedings of the 14th ACM SIGPLAN International Conference on Software Language Engineering (SLE '21), October 17–18, 2021, Chicago, IL, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3486608.3486900

1 Introduction

Automated tooling for managing and analyzing source code is important to developer productivity, and fundamentally relies on parsing. Text editors and integrated development environments tend to include source code analysis tooling so that programmers can see feedback immediately as they edit code. The problem of parsing therefore becomes more difficult because it is not efficient to parse the document from scratch after every edit. In many cases, it is not necessary either, as an edit changes only a small part of the overall parse tree. Incremental parsing algorithms provide methods for reparsing only the necessary subsets of the document after an edit, while still providing the same resulting parse tree.

Incremental Packrat Parsing [3] is an existing algorithm for incremental parsing built on packrat parsing – a parsing strategy that involves saving parse results for reuse. Incremental Packrat Parsing makes a small modification to packrat parsing and to make it suitable for incremental settings. As a result it is easy to modify existing packrat parsers to be incremental. However, implementations in prior work use traditional data structures and strategies for memoization. These techniques are not ideal for incremental parsing and result in only a constant improvement to reparse time (rather than asymptotic improvement with respect to the input size). Figure 1 shows that the reparse performance of incremental packrat parsing quickly degrades as the input size grows.

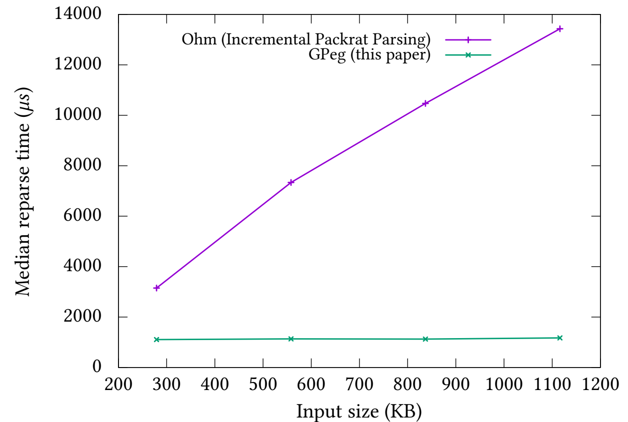


Figure 1. Replicated results from *Incremental Packrat Parsing*, using the original ES5 grammar and 891 edits from their evaluation. This graph shows the linear scaling of the original algorithm. In this paper, we show how to improve the algorithm to give reparse times that are logarithmic in the file size, showing sub-5ms reparse times for much larger files with significantly less memory overhead.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SLE '21, October 17–18, 2021, Chicago, IL, USA
© 2021 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-9111-5/21/10…$15.00
https://doi.org/10.1145/3486608.3486900

In this paper, we rethink the fundamental implementation of the incremental packrat parsing algorithm in order to significantly improve reparse time. We use a new data structure for the memoization table, and present a new memoization strategy that can efficiently handle large amounts of linear repetition – for example, caused by a Kleene star expression `p*`. The result is an implementation of incremental packrat parsing that provides improvements to the asymptotic runtime of the reparse in the common case. With our changes, reparse time is logarithmic rather than linear, in the size of the input for common edits[1] and grammars. Using these new strategies, we are also able to aggressively prune the memoization table, resulting in significantly less memory overhead. Experiments with our prototype show that reparse time is not substantially affected by the size of the input, and can efficiently handle a wide variety of grammar types. We show that these changes to incremental packrat parsing make it applicable to workloads such as syntax highlighting and full language parsing for very large files.

We make three primary modifications to incremental packrat parsing:

1. The memoization table structure is an interval tree so that it can efficiently handle overlap queries. We also describe the modifications that were necessary to implement our incremental parser.
2. The memoization table handles shifts in the position of the text efficiently by applying them lazily.
3. The memoization strategy memoizes as a tree structure the linear repetition caused by the Kleene star operator. This modification also makes further optimizations in space usage practical, such as a memoization threshold to prevent small entries being memoized (since the tree structure automatically creates large entries).

In addition to these algorithm modifications, we provide a complete implementation with three artifacts:

1. GPeg[2]: a PEG parsing machine that has been augmented with support for incremental parsing.
2. Flare[3]: a syntax highlighting engine with a simple grammar language that can easily define new languages. Because GPeg is not a static parser generator, these languages can be added and loaded at runtime.
3. Demo editor: we use Flare to implement a prototype editor and in doing so make some additional optimizations, primarily to ensure that memory usage stays reasonable, even for large files (tens of megabytes).

2 Related Work

Incremental Packrat Parsing [3] is the algorithm that we build on in this work, and can be used to easily adapt PEG packrat parsers to be incremental. The algorithm itself is quite simple and effective, and does not require any explicit support in the grammar or editor/integrated application. However, the original implementation uses strategies and data structures that mean the reparse time is linear in the size of the document, and memory usage cannot easily be restricted. Other top-down incremental parsing algorithms have focused on LL(1) grammars [16, 17], and are restrictive compared to Incremental Packrat Parsing.

Alternative algorithms in incremental parsing mostly focus on implementations alongside LR parsers, first introduced by Ghezzi and Mendrioli [9, 10]. Building on that work, Wagner presents methods for optimally applying edits to a parse tree generated by an LR parser [19, 20]. Wagner's algorithms are in some ways similar to the ones we propose (with a heavy reliance on trees), but packrat parsing uses an explicit cache (the memoization table), while incremental LR parsing detects changes directly in the parse tree. The explicit cache makes some space optimizations easier to implement, particularly in editors or any case where only the parse tree for a certain window of the input is needed. Packrat parsing is also often used for parsing PEGs rather than CFGs, which makes our algorithm better suited for incremental PEG parsing. Though Wagner provides no open-source implementation, Tree-Sitter [4] is a recent project built on his research that has been used for syntax highlighting and more in multiple mainstream editors.

Papa Carlo [14] is a project that implements incremental parsing for PEGs. It uses an approach where "fragments" in the grammar are manually marked for caching, and the grammar writer must ensure that the chosen fragments adhere to some constraints. Papa Carlo's fragments allow the reparse stage to perform independent of the file size, but it appears to use a linear-time algorithm to determine the portion of the document that has changed and which fragments have been modified.

We implement our incremental parser as a parsing machine. The concept of a parsing machine was first introduced by Knuth [13]. More recently, parsing machines have been used for parsing PEGs [11, 15], with an implementation in the LPeg library [12]. Our GPeg is heavily based on this research and uses the same core instructions and machine organization, though we have had to make modifications to captures and add support for memoization. NPeg [2] is another PEG parsing machine written for the Nim programming language.

3 Background

Incremental Packrat Parsing is best applied to parsing PEGs because of their determinism. While packrat parsing can be applied to non-PEG grammars, throughout the paper we present the algorithm in the context of PEGs only.

[1] Specifically, edits that modify the parse result only in a localized area.
[2] Available at github.com/zyedidia/gpeg.
[3] Available at github.com/zyedidia/flare.

```
Top    <- Expr !.
Expr   <- Term ([+\-] Term)*
Term   <- Factor ([*/] Factor)*
Factor <- Num / '(' Expr ')'
Num    <- [0-9]+
```

Figure 2. Arithmetic expression PEG. The `Top` non-terminal ensures that the entire input is a single arithmetic expression by using the `!.` pattern.

3.1 Parsing Expression Grammars

A Parsing Expression Grammar (PEG) is a formalism for specifying deterministic string recognizers [7]. A PEG defines a top-down recursive descent parser by specifying a set of non-terminals and their corresponding patterns. PEG definitions are similar to context-free grammars, using similar notation for repetition, ranges, and literals, except for some key differences: the choice operator, and predicates for unbounded lookahead.

In a PEG, the choice operator, written '`/`', is a prioritized choice. The pattern `a / b` must attempt to match `a` before it attempts `b`. Thus `b` can match only if `a` does not match. As a result, the choice operator in PEGs does not introduce any ambiguity, whereas it can in context-free grammars. One consequence of prioritized choice is that left-recursion cannot be expressed in a PEG. An expression such as `a <- a / b` would create an infinite loop where the parser always attempts to match `a` without any way to exit.

PEGs also support negative and positive lookahead predicates, written using the '`!`' and '`&`' operators. The expression `&p` succeeds if `p` matches at the current location and fails otherwise, consuming no input regardless. It attempts to match the pattern `p`, then backtracks to the original point where the match attempt began while preserving only the knowledge of whether `p` matched or not. The expression `!p` is the converse of `&p`: it fails if `p` matches and succeeds otherwise. Note that `&p` is equivalent to `!!p`.

One common use of the Not predicate is to force the grammar to consume characters until the end of the document. The pattern `!.` succeeds if it is not possible to accept another character, so placing this pattern at the end of the top-level non-terminal ensures that the parse succeeds only if the entire document is consumed by the non-terminal.

As an example, a PEG for an arithmetic expression language is shown in Figure 2.

3.2 Packrat Parsing

Along with PEGs, Ford presented packrat parsing [6], which can be used for efficiently parsing PEGs. Packrat parsing is commonly used for parsing PEGs because it guarantees linear time parses even though PEGs support unlimited lookahead [6]. It works by keeping a memoization table – a data structure which allows the parser to remember the results of attempting to parse a certain pattern starting at a certain location. If the parser ever tries to reparse the same pattern at that location, it can first check the memoization table for an entry and if there is one simply use the information in the entry instead of parsing the input. Parsing time is linear because work is never duplicated and the grammar only has a fixed number of patterns to try before it fails.

The memoization table is a key-value store where the key is a pair (id, pos) which corresponds to a pattern or non-terminal (uniquely identified by id) starting at a given location pos in the input. The value is a memoization entry that stores all the information produced by parsing from the pair (id, pos):

1. The length of the match (or a special value $\bot$ if the pattern did not match).
2. A resulting parse tree (optional).

Patterns may be marked for memoization[4]. When the parser attempts to match such a pattern at a certain position, it will either succeed or fail. In both cases, the parser will then insert an entry into the memoization table at (id, pos), logging the result. Packrat parsing algorithms memoize every non-terminal in the grammar. However, it is possible and even desirable to use a different memoization strategy because the packrat strategy has significant memory overhead. Note that using a different memoization strategy may cause the parsing algorithm to become super-linear for pathological cases [7].

An example of a packrat parser's memoization table is shown in Figure 3. The memo table in the figure is also augmented with additional information for incremental parsing as explained in the next section.

3.3 Incremental Packrat Parsing

Packrat parsing is appealing for the case of incremental parsing because it keeps a memoization table of partial results. Incremental Packrat Parsing is a simple and effective algorithm that takes advantage of this.

As explained by Dubroy [3], a packrat parser can be modeled by a function:

$$\textsc{Parse} : (G, s) \rightarrow R$$

where G is a grammar, s is an input string, and R is the parse result (possibly a parse tree, or indication of success/failure).

An incremental packrat parser can be modeled similarly by a function:

$$\textsc{Parse} : (G, s, M) \rightarrow (M', R)$$

where M and M' are memoization tables. When M is empty, the incremental packrat parser is the same as the packrat parser, except it exposes the resulting memoization table to the user. If M is not empty, then the parser will execute faster

[4]Our notation uses `{{ p }}` to mark pattern `p` for memoization.

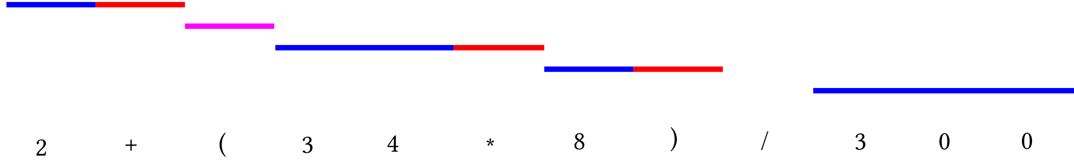


Figure 3. Memoization table for an arithmetic expression, where only the Num non-terminal has been memoized (for the purposes of the example). Blue indicates the characters that belong to the match, and red indicates additional examined characters. A magenta entry indicates that the match was attempted at that location but failed, and examined the shown character(s). Note there is no entry above the division symbol because the parser did not attempt to match Num at that location.

because it will be able to skip entries that were filled before the parse even began.

Algorithm 1 Incremental Packrat Parse

```
1: M ← a new memoization table
2: s ← the initial input string
3: G ← the grammar
4: When an edit operation e occurs
5:     s, M ← ApplyEdit(s, M, e)
6:     M, res ← Parse(G, s, M)
```

After an initial parse, when an edit to the input string occurs, parts of the memoization table become invalid. Evicting these entries results in a valid memoization table which can then be used as an input for reparsing. Dubroy introduces a function for this:

$$\text{ApplyEdit} : (s, M, e) \longrightarrow (s', M')$$

where e is an edit consisting of two parts: an interval $[e_{start}, e_{end})$, specifying the part of the document that is removed, and a string of bytes e_{text} which is then inserted at e_{start}.[5]

Applying the edit consists of evicting all newly invalidated memoization entries, and making sure the start positions of all entries are properly shifted according to the edit (deletion and/or insertion).

The incremental packrat parsing algorithm (Algorithm 1) can thus be summarized as the following three steps that must be performed when an edit is made:

1. Determine all memoization entries that are invalidated by the edit and evict them from the memoization table (performed by ApplyEdit).
2. Shift the start position of all memoization entries that start after the edit (performed by ApplyEdit).
3. Reparse the document from the start using the modified memoization table (performed by Parse).

A memoization entry is invalidated by an edit if any of the characters that were examined to make the match are changed by the edit. Thus, in a memoization entry we must not only track how many characters were matched, but also how many characters were examined to make the match. Our memoization entry now stores:

1. The length of the match (or ⊥ if the pattern did not match).
2. The number of characters examined to make the match.
3. The parse tree generated by matching by the pattern (only if the pattern matched).

Since PEGs support unlimited lookahead, the number of examined characters may be much larger than the length of the match. A memoization entry at position p with n_e characters examined to make the match is invalidated by an edit over the interval $[e_{start}, e_{end})$ if that interval overlaps with $[p, p + n_e)$.

Figure 3 shows an example memoization table after parsing an arithmetic expression. The table stores entries for the Num non-terminal, with each entry tracking the starting position, length, and examined length. Only entries that overlap with an edit are evicted, allowing reparses to reuse the remaining entries without examining the unchanged text.

4 Improving Incremental Packrat Parsing

The incremental packrat parsing algorithm is effective for improving the performance of reparses, but the original implementation does not improve the asymptotic complexity of reparsing compared to the initial parse. This is because it uses a traditional memoization table structure, which provides efficient indexing of (id, pos) keys, but has linear complexity for interval overlap queries (step 1) and applying shifts (step 2). Additionally, traditional memoization strategies (memoize every non-terminal) result in high memory usage, and linear time for step 3 for flat grammars (grammars without much nesting of non-terminals).

Using non-traditional memoization table data structures and a packrat parsing strategy tailored for reparsing, we can achieve logarithmic performance for reparsing in the common case.[6]

The problem with evicting entries that overlap with the edit (step 1) is that an array or hashtable memoization table is not well-suited to computing interval-overlap queries.

[5] Insertion and deletion are special cases where $e_{start} = e_{end}$, and $|e_{text}| = 0$ respectively.

[6] Since an edit can completely destroy the parse tree (e.g., opening a multiline comment at the top of the document), worst case complexity is still linear.

Solving this requires a change of data structure. In particular, we can implement the memoization table as an interval tree, which supports overlap queries in logarithmic time. Applying shifts (step 2) is still a problem, but with some augmentations to the interval tree we can also significantly improve performance there.

4.1 Interval Tree

An interval tree is an augmented binary search tree that stores a set of n intervals. It can perform the following operations:

- Insert a new interval: $O(\log n)$.
- Delete an interval: $O(\log n)$.
- Find the interval starting at a location/key: $O(\log n)$.
- Query for all intervals that overlap with a specified interval: $O(m + \log n)$, where m is the number of overlapping intervals (size of the result).

The interval tree provides an ideal solution to evicting invalidated entries because the intervals that overlap with the edit can be found in logarithmic time.

Implementing an interval tree involves augmenting a binary search tree. Each node in the tree corresponds to an interval, and that node is sorted based on the start position of the interval. In addition, each node in the tree stores the maximum position of any interval in its children. This allows the overlap search to skip entire subtrees, resulting in logarithmic time queries.

While the interval tree may be slower than an array or map for insertion and lookup, the difference between constant time and logarithmic time (lookup/insertion) is not nearly as significant as the difference between logarithmic time and linear time (overlap query).

The OVERLAPS procedure (Algorithm 2) for interval trees recursively finds all intervals in the tree that overlap with the query interval. Each node makes sure to avoid subtrees if it can guarantee the interval will not overlap with any intervals in those subtrees. These guarantees can be made because the maximum endpoints of every subtree are known, and the right subtree is guaranteed to be only intervals to the right of its parent node. Note that in an implementation for incremental parsing, we would want to remove all overlapping entries during OVERLAPS rather than return them for a slight performance improvement.

Our prototype implementation builds the interval tree using an AVL tree as the underlying binary search tree. This ensures that the tree remains balanced.

4.2 Lazy Shifts

Most edits to the document will involve the insertion or deletion of characters. If an edit happens at position n, then all memoization entries that begin at an index $\geq n$ will need to shift their position by the number of characters inserted/deleted (the shift information is a pair made from the index and size of the edit). In the worst case, where the edit happens at the start of the document, every memoization entry in the table will need to be shifted. If shifts are immediately applied to every entry, this will clearly result in shift application being an expensive linear-time operation.

Algorithm 2 Interval-Overlaps

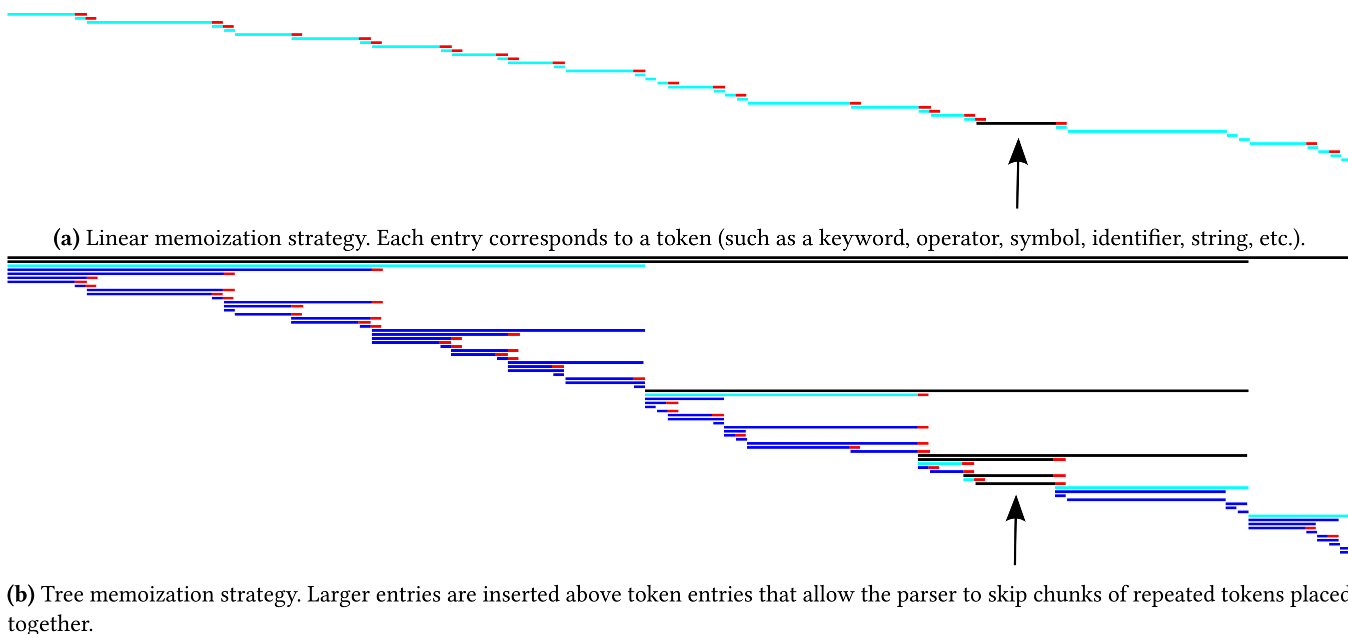
```
1:  procedure OVERLAPS(n, [l, h), r)
2:      if l > n_max then
3:          return r
4:      r ← OVERLAPS(n_left, [l, h), r)
5:      if HASOVERLAP([l, h), n_interval) then
6:          r ← r :: n_interval
7:      if h < n_start then
8:          return r
9:      r ← OVERLAPS(n_right, [l, h), r)
10:     return r
11: procedure HASOVERLAP([l_1, h_1), [l_2, h_2))
12:     return l_1 < h_2 ∧ h_1 > l_2
```

Our solution is to apply shifts lazily, only as they are needed. When a node in the tree (i.e., an interval) is read or observed, it must make sure all shifts so far have been properly applied to it. When an edit occurs, the corresponding shift is recorded in a global shift list, but does not have to be applied to any nodes until specific nodes are observed (e.g., during a lookup), and is never required to be applied to all nodes at once. To enable this, each node in the interval tree stores a timestamp that tracks its most recently applied shift. When any data is requested from the node (such as its interval end-points), it first applies any more recent shifts from the global list.

The size of the global shift list is proportional to the number of edits that have been made so far. In order to prevent unbounded growth, a reference counting approach can be used to remove old shifts, or the entire list can be applied after a fixed number of edits. However, we find that in practice it is reasonable to let the list grow indefinitely. The primary issue is memory usage, which only becomes significant after millions or billions of edits.

There are other approaches to handling shifts, for example by using a fully lazy data structure where shifts are propagated to individual nodes. Another possibility might include a rope-like data structure where interval positions can be calculated from relative information, avoiding the need to store absolute positions. However, we find that the global list approach is a good balance of simplicity and performance.

Relocatable Parse Results. If parse results (nodes in the syntax tree produced by the parser) are to be stored in the memoization table, they must be relocatable, meaning that when a shift occurs the data stored by the parse result (such as a starting location) is still valid. Dubroy et al. [3] solve this problem by storing only the captured string in the parse tree



(a) Linear memoization strategy. Each entry corresponds to a token (such as a keyword, operator, symbol, identifier, string, etc.).

(b) Tree memoization strategy. Larger entries are inserted above token entries that allow the parser to skip chunks of repeated tokens placed together.

Figure 4. Resulting memoization table from the same input using two different strategies. The black arrow denotes an edit. Black entries are evicted by the edit, and light blue entries are those looked up in the table during the subsequent reparse. Red indicates characters examined by the corresponding entry. This example used a syntax highlighting grammar, so each bottom-level entry corresponds to a single token, and the grammar is just a Kleene star repetition of the token pattern, allowing for the tree memoization optimization to take place.

nodes. However, for many use-cases it is important to store the position of the node, either for use by the source code analysis or for error reporting. We can solve this issue by also applying lazy shifts to the parse tree nodes. If nodes store a reference and position offset to the memoization entry they are contained in, the start position can be calculated lazily. One consequence is that the memory size of captures is increased by the need to store this reference and offset.

4.3 Tree Memoization

With our new memoization table data structure, we can invalidate and evict entries from the table much faster. The last step in the incremental packrat parsing algorithm is to reparse from the start of the document, using the memoization table to skip unchanged parts. The layout of entries in the table will determine the efficiency of this reparse. If the memoization table exhibits a linear structure, as shown in Figure 4a, the reparse will be slow for large inputs, since the parser will have to perform a number of memoization table lookups that is proportional to the size of the file. Linear structures can arise commonly in very large files (usually the majority of a large file is a single repeated pattern). In general, the repetition operator (Kleene star) is the root of the problem because it leads to memoization structures that have a linear rather than logarithmic structure.

Our solution is to change the memoization strategy for the repetition operator. By using a special case for the Kleene star operator we can force a tree structure rather than linear structure for the memoization of patterns of the form `{{ p }}*`[7]. The goal is for the resulting memoization table to resemble that of Figure 4b. With a tree structure in the memoization table, any location in the document can be reached by skipping through a logarithmic number of memoization entries. This is shown visually in Figure 4: when an edit occurs at the black arrow, the black entries are evicted and the light blue entries are looked up during the subsequent reparse. In the linear structure every entry is looked up, while in the tree structure large parent entries result in only a logarithmic number of lookups.

Implementation. We now describe the changes to the memoization table necessary to implement tree memoization, and our algorithm for creating the tree structure while parsing.

In order to implement tree memoization, two modifications to the memoization table are necessary. First, the memoization table must be able to accommodate multiple entries starting at the same location and ID, and thus with the same key. This means that the interval tree must store an array of entries per key. During a find operation, the tree should return the value associated with the largest interval matching the requested key. This ensures the largest possible section of the input is skipped.

[7]Recall that `{{ p }}` denotes that p should be memoized.

The second modification is necessary for the actual construction of the tree structure during the parse. Memoization entries must store a "count" that refers to the number of repeated patterns stored in the entry. For example, if the memoized pattern corresponds to the pattern `p*`, the count will store the number of occurrences of `p` in the memoized section of the input. Only entries that are created as a result of tree memoization will have a count greater than 1.

While parsing a memoized Kleene star, `{{ p }}*`, the parser executes a loop matching and memoizing `p`. The general algorithm for tree memoization is to match `p` at each iteration, push a new stack item corresponding to the match, and then run a routine that traverses the stack and "merges" items if they are large enough. When two stack items are merged, a new memoization entry that covers both stack items' matches is inserted into the table. Note that when the parser attempts to match `p`, it may either find an entry already in the memoization table, possibly storing multiple back-to-back occurrences of `p` (with a count greater than 1), or it may attempt a direct match if nothing in the table is found.

The stack items that are pushed during tree memoization must store the following information:

- The position of the match.
- The count. This will be 1 for a direct match. If an entry is found in the memoization table, this will be that entry's count.
- Depending on the parser implementation, the entry may also contain a list of parse results that were made while matching `p` or stored in the recovered memoization entry.

At the end of each iteration, the parser traverses the stack and inserts new "parent" memoization entries into the table. As the parser scans down the stack, it sums the counts of the stack items, and if it finds an item whose count is less than or equal to the running sum, all the scanned items are popped and a new stack item is pushed encompassing them all (starting from the earliest start, with length to cover all items, a count that is the sum of all counts, and all parse results). This creates a tree structure that is also resistant to new inserts/deletes in the input text.

Figure 5 shows a small example for the pattern `{{ . }}*` matching the text "1234" to demonstrate how the tree is created, and how it reacts when a new character is inserted. As each occurrence of the repeated pattern (a single character in this case) is parsed, new memoization entries are created along with new stack items. The post-match routine then traverses the stack and attempts to merge stack items to create large parent entries. Note that in practice, a memoization threshold (discussed in Section 4.4) will prevent single-character or other small entries like those in this example.

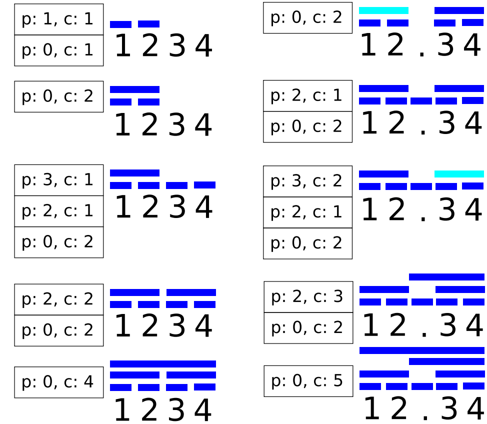


Figure 5. Tree memoization construction example showing the input text, stack, and memoization table (some iterations are condensed and some are expanded). Each stack item stores a position, marked 'p', and a count, marked 'c'. The pattern being matched is `{{ . }}*`. The left column shows the initial construction of the tree structure, and the right column shows reconstruction during a reparse after a character is inserted.

4.4 Space Optimizations

Space usage is a common problem with packrat parsers because the memoization table can become very large. A simple optimization is to introduce a threshold to avoid memoizing entries that are below a certain length. In a standard incremental packrat parser this can cause lots of useful information to be discarded, specifically in cases where many small tokens are repeated one after another. However, with the tree-based memoization strategy presented in Section 4.3 we can be much more aggressive with this optimization, since repeated small tokens are guaranteed to be memoized under larger parent entries. In fact, this optimization becomes very powerful, allowing thresholds on the order of hundreds to thousands of bytes. With a large enough threshold, the space usage of the memoization table can be significantly reduced from the size it would be with standard packrat parsing.

The more significant consumer of space (other than the input itself) is then the resulting parse tree. We cannot take the same approach as with the memoization table because small results are needed in the parse tree. However, in many cases, especially when using an editor, only a certain piece of the parse tree is needed for processing. For example, only the parse tree corresponding to the tokens in the currently viewed window is needed for syntax highlighting or indentation analysis. Therefore we can avoid constructing parse results for tokens that do not overlap with the viewed interval, saving greatly on space usage. This makes the space cost of the parse result proportional to the editor window size rather than the file size.

We validate these two space optimizations in the evaluation section, and find that the end-to-end space cost of syntax highlighting is quite reasonable even for very large files.

4.5 Corner Cases

For typical edits, the reparse time will be logarithmic in the size of the document. However, even small edits can require reparsing of the entire document in certain cases. For example, in a syntax highlighting grammar, inserting the start of a multiline comment at the top of a document will cause the entire document to become a comment, which does not use any pre-existing information from the memoization table. Another example occurs in languages which use multiline strings. Deleting a quotation mark will flip following text in the document – text previously in a string will no longer be in a string and vice versa. Again, no saved information will be used, resulting in non-logarithmic reparse time. However, in both of these cases, the old information is still stored in the memoization table, meaning if the edit is reverted then reparsing will be fast. Reparsing will also be fast if the edit is then re-applied.

5 Parsing Machine

Our improvements are implemented in a PEG parsing machine called GPeg. The full parsing machine is outside the scope of this paper, so we try to distill the important parts of the implementation in this section. If readers wish to learn more about parsing machines, please see the LPeg parsing machine [11, 15], and our GPeg implementation for the full details.

When using a parsing machine, a grammar is compiled into a set of instructions and then executed by the parsing machine interpreter. This has a number of benefits, including that grammars can be compiled and executed at runtime, and do not need to be statically compiled and bundled with the application.

In order to adapt the parsing machine approach to incremental parsing, we had to make two modifications:

1. Parse results (called *captures* in the parsing machine), such as syntax tree nodes, must be generated directly during the parse rather than logged and generated by a post-parse pass as in LPeg.
2. New memoization instructions must be introduced, including special instructions for performing the tree memoization strategy.

The parsing machine state consists of a set of registers and a stack. The registers store information such as the current instruction pointer (called *ip*) and the current position in the input to examine (called *sp*). The stack can be used to store backtracking and memoization information. Backtracking works by pushing a special backtrack stack entry storing an (ip, sp) pair to return to if necessary. The parsing machine supports a number of instructions that modify its state and examine the input string. Each instruction moves the instruction pointer, or sets it to a special *fail* state, during which the machine repeatedly pops entries from the stack and backtracks to their locations.

A grammar can be compiled into a list of instructions to be run on the machine.

Some core example instructions are given below:

- `Char` *b*: advances *ip* by one instruction and consumes one byte from the subject if it matches *b* and goes to the fail state otherwise.
- `Choice` *l*: pushes a backtrack entry storing *l* and *sp* so that the parser can return to this position in the document later and parse a different pattern (stored at *l*).
- `Commit` *l*: pops the top entry off the stack and jumps to *l* (setting $ip = l$). This allows the machine to commit to a state and discard a backtrack entry.
- `End`: if the machine reaches this instruction the match is declared a success, and the current *sp* corresponds to the final matched character.

With these instructions, the pattern `'ab'/ 'yz'` can be compiled into a simple program:

```
    Choice L1
    Char 'a'
    Char 'b'
    Commit L2
L1: Char 'y'
    Char 'z'
L2: End
```

This program first pushes a backtrack entry using the `Choice` instruction. It then tries to match 'a' followed by 'b.' If either match fails, it will go to the fail routine, which pops the backtrack entry and sets *ip* to `L1` and *sp* back to `0`. Then it will attempt to match 'yz.' If matching 'ab' succeeds, the commit instruction will discard the backtrack entry and jump to the `End` instruction.

The full parsing machine supports additional instructions, including function calls, capture and memoization instructions, and a large number of instructions for optimization.

5.1 Capture Mechanism

A capture is a parse result of the form:

$$(id, content, children) \in \mathbb{N} \times \textbf{Content} \times \langle \textbf{Capture} \rangle.$$

It stores an ID, the "content," and a list of child captures. The structure of the content is left unspecified, but for example may consist of a portion of the source text.

Incremental parsing requires captures to be computed during the parse so that intermediate results can be stored in memoization entries. This means we cannot use LPeg's

capture approach because it involves storing capture information during the parse and constructing the actual captures during a post-parse pass. Instead we must track captures during the parse, via a top-level capture list.

Two instructions are added to support captures: `CaptureBegin ID`, which creates a special capture stack entry, and `CaptureEnd`, which pops the stack entry and constructs the capture object. A pattern can then be captured by surrounding its compilation result with those two instructions:

```
CaptureBegin ID
<p>
CaptureEnd
```

The parser keeps a list of top-level captures, which are returned when the parse completes. However when `CaptureEnd` constructs a capture, it cannot add it directly to the list because it is possible that the text after the capture will cause the parser to fail and backtrack, invalidating the capture. As a result, we must store a list of captures in each stack entry, and when the entry is popped it appends its list to the entry above it. If there is no entry above it, the captures are appended to the top-level list. When a stack entry is popped in the fail routine, the captures are destroyed rather than appended.

5.2 Memoization

Memoization in the parsing machine is implemented similarly to captures, with a `MemoOpen Label ID` instruction and `MemoClose`. The `MemoOpen` instruction searches the memoization table for a pattern with `ID` at the current position. If one is found, it jumps to `Label`, but if not it pushes a stack item with the current position and `ID`. When `MemoClose` runs, it looks for a stack item pushed by `MemoOpen` and constructs a memoization entry to insert into the table.

A pattern `p` can be memoized by compiling it as:

```
    MemoOpen L1 ID
    <p>
    MemoClose
L1: ...
```

Tree memoization is a special case, where a pattern of the form `{{ p }}*` is compiled using specialized tree instructions.

```
L1: MemoTreeOpen L3 ID
    Choice L2
    <p>
LN: MemoTreeInsert
L3: MemoTree
    Jump L1
L2: MemoTreeClose
```

The `MemoTreeOpen` performs the first part of the tree memoization algorithm by checking the memoization table and pushing a stack item. If an entry is found, it jumps directly to `L3`, which performs the stack traversal using the `MemoTree` instruction. If no entry is found it attempts to match `p` and insert an entry to the table before performing the stack traversal with `MemoTree`. If matching `<p>` ever fails, the parser exits the loop via the backtrack entry pushed by the `Choice` instruction, and runs `MemoTreeClose`, which cleans up the stack. The exact details of these instructions are outside the scope of the paper, but this code provides an outline for how the parsing machine implements tree memoization.

6 Syntax Highlighting

One of the primary applications of GPeg is syntax highlighting. While GPeg supports general incremental parsing, syntax highlighting is one of the most common workloads where incremental parsing is needed. We examine *token highlighting*, a specific method of performing syntax highlighting where the language is analyzed at only the token level (as opposed to using a full language grammar). Token highlighting is a good example for testing GPeg because these grammars produce very linear parse trees, which are quite different from those produced by a full language grammar. In addition, since a token highlighting grammar accepts any input, no edit can cause a parse failure.

Creating a token highlighting grammar with GPeg is simple. We define patterns for each lexical element of the language, such as keywords, comments, strings, etc. For example, in Java[8], the `comment` pattern may be defined as[9]

```
comment       <- line_comment / block_comment
line_comment  <- '//' (!'\n' .)*
block_comment <- '/*' (!'*/' .)* '*/'?
```

Once we have similar patterns for keywords, strings and other language elements, we define a `token` non-terminal that attempts to match one of them:

```
token <- whitespace / keyword / comment / ...
```

The highlighter should then repeatedly attempt to consume a `token`, if the pattern does not match, consume characters until a match is found. This repetition pattern is the top-level non-terminal of the grammar.

```
{{ token / . (!token .)* }}*
```

Since this is a repetition of a memoized pattern, we will be able to take advantage of tree memoization for fast incremental parse times. In the case where `token` does not match, we continue to consume characters using `(!token .)*` until a `token` matches. This ensures that a contiguous block of characters that do not belong to a token are saved as a single memoization entry.

As a demonstration, we built Flare[10], a syntax highlighting engine that uses GPeg with grammars of this form, with support for 10 languages at the time of writing.

[8]See Appendix A for the full Java grammar.
[9]We could also define the `block_comment` using memoization as `'/*' {{ (!'*/' .) }}* '*/'?` to efficiently handle cases of extremely large block comments.
[10]Available at github.com/zyedidia/flare.

7 Evaluation

The experimental validation of GPeg is presented in three sections:

1. Asymptotic scaling: we test GPeg's performance on increasingly large files to show that the reparse time remains fast.
2. Per-edit performance: we show the reparse time required for each individual edit over the course of many synthetic edits.
3. End-to-end editor: we integrate GPeg-powered syntax highlighting into a prototype editor, and validate its performance when integrated with all the additional machinery that an editor requires. We also validate the space usage optimization presented in Section 4.4.

Throughout the experiments we have tested against existing parsers, both incremental and traditional. We also perform our experiments using a variety of grammars: Java/JSON for highly nested full-language grammars, and syntax highlighters for very flat grammars.

Throughout the experimental validation, we show:

- The reparse time is independent of the input size.
- Reparse performance scales well with many edits.
- The memory overhead of the memoization table and relocatable captures can be high, but with certain optimizations it is manageable.

Reparse time and memory usage are the primary measurements we are concerned with, and initial parse time is secondary. In addition, we use the window space optimization for captures since this most accurately models the editor use-case. For workloads where the full syntax tree is desired, parse nodes could be optimized for better space performance compared to our implementation. We keep our implementation since the parse tree does not consume significant memory when the window optimization is applied in an editor.

All experiments are performed on a machine running Linux 5.4.0 with an AMD Ryzen 5 1600 CPU (mid-range desktop CPU from 2017). GPeg is written in Go and is compiled with the Go 1.16 compiler.

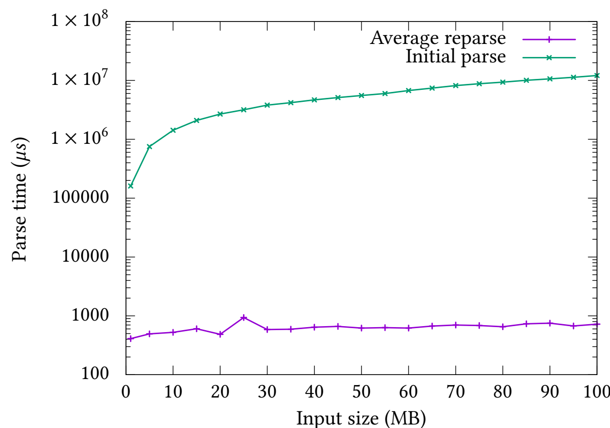


Figure 6. Asymptotic scaling of the Java syntax highlighter. The average reparse time over 1000 edits stays constant at 1ms while the initial parse time rises proportionally with input size.

7.1 Asymptotic Scaling

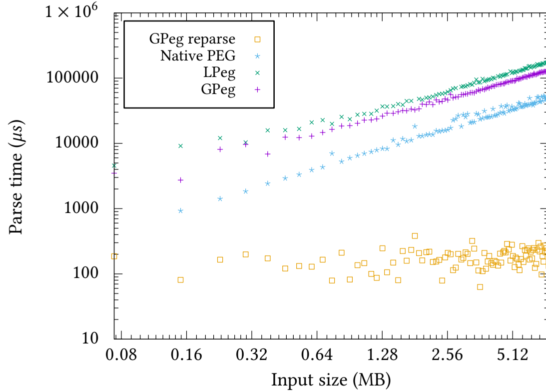
In the first experiment, we show how GPeg improves on Incremental Packrat Parsing by replicating their experiment (using their implementation, called Ohm). We perform 891 simulated edits on the original 279KB JavaScript file, and duplicated versions of it, and compare the performance of GPeg and Ohm in Figure 1 (presented during the introduction). As expected, Ohm exhibits linear scaling, in this case primarily caused by linear behavior in ApplyEdit (invalidating memoization entries and shifting them). This is because the memoization table is quite large (roughly 2,000,000 entries). GPeg's memoization table is the same size, but the lazy interval tree enables much more efficient edit application, and the performance stays flat. JavaScript is a highly nested grammar and the file in question already exhibits a tree-like structure, so our tree memoization optimization does not provide a significant speedup in this case. In converting the ES5 grammar to GPeg, we had to mark certain non-terminals with explicit memoization because the grammar relied on this behavior for handling recursion.

The next experiment shows asymptotic performance using a Java syntax highlighting grammar. In this case we test on larger files, ranging from 1 to 100 megabytes, constructed by concatenating together source files from the OpenJDK7 project. Parse time scaling is shown in Figure 6. This shows the initial parse time increasing roughly linearly ($O(n \log n)$) while the reparse time stays constant at around 1ms.

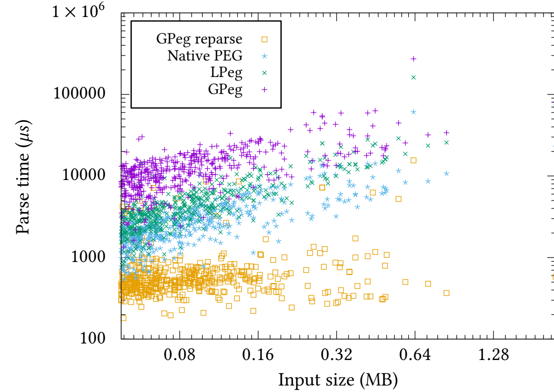
Finally, we test reparse time with complete Java and JSON grammars. Figure 7 shows that the reparse performance stays flat as file size increases, and also compares GPeg's initial parse time with other PEG parsers: LPeg [12] (a similar VM approach implemented in C) and a Go PEG parser generator [18]. GPeg's initial parse is similar to LPeg's, but the native PEG parser is faster, as expected.

7.2 Per-Edit Performance

We would like to ensure that the reparse time does not degrade as more edits are made. We apply a series of synthetic edits to a Java file, and parse the result with a Java parser and highlighter. Generating synthetic edits for the Java parser requires some additional work, since random edits will cause the input to become invalid Java almost immediately. We generate edits that maintain the validity of the Java document by using a small parser to extract elements such as comments, strings, function names, etc. and then changing,



(a) JSON parser performance on files ranging in size from 100KB to 8MB, synthetically created.



(b) Java parser performance on files ranging in size from 50KB to 2.5MB, from the Ceylon [8] and OpenJDK7 [1] projects.

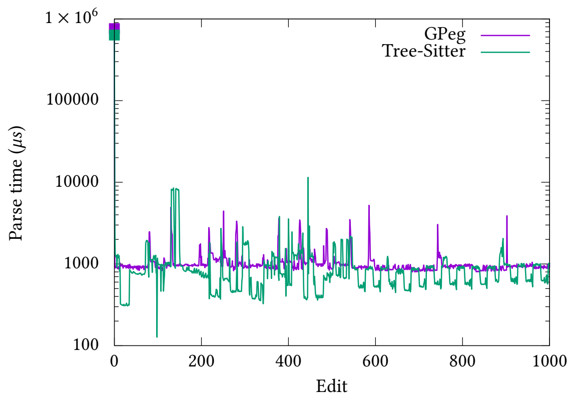
Figure 7. Parser performance for JSON and Java datasets.

those elements in appropriate places. Edits for the Java highlighter are randomly generated, since no edit can cause the input to become invalid.

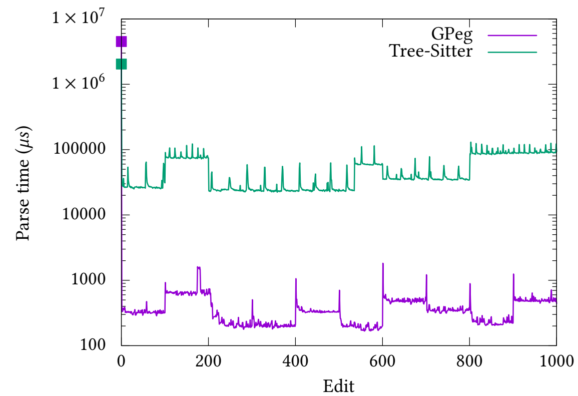
Results are shown in Figure 8, where GPeg is also compared with Tree-Sitter, a state-of-the-art non-PEG incremental parser. Tree-Sitter generates static native C parsers, compared to GPeg's dynamic parsers run in a VM written in Go. GPeg's performance is similar to Tree-Sitter's, though GPeg has a slower initial parse time. In Figure 8b, the random edits stress Tree-Sitter's error recovery mechanism since it is still trying to apply a full Java grammar to perform highlighting, rather than using a simpler token-based highlighter as with GPeg. The two figures show rough comparisons, and help to frame GPeg's performance in the context of another incremental parser from the perspective of scalability with large files and numbers of edits, rather than in absolute time, due to differences in grammars (CFG vs. PEG), capture generation, and highlighting method.

7.3 Memoization Threshold

So far we have been using a memoization threshold of 128 bytes, which prevents any entries below the threshold from being memoized. This threshold introduces a trade-off between memory usage and reparse time. If the threshold is



(a) Full Java parsing of a 5 MB file. Edits are synthetically generated to maintain the validity of the Java program and are applied as single-character changes. Tree-Sitter additionally generates a syntax tree while this is disabled in GPeg.



(b) Java highlighting of a 17 MB file. Edits are completely random and clustered in chunks of 100 edits in the same location. At each change we can see a spike in GPeg as shifts in that portion of the interval tree are lazily applied. Tree-Sitter's performance degrades because it uses a full grammar for highlighting and the random edits stress the error recovery.

Figure 8. Per-edit performance using a Java parser and highlighter. These benchmarks show that GPeg's reparse time does not degrade with the number of edits, and has similar scaling behavior to Tree-Sitter, a well-known incremental parser and highlighter.

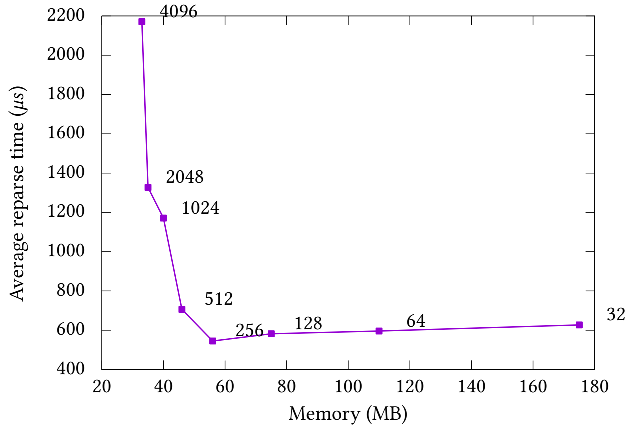


Figure 9. Tradeoff between speed and space while highlighting a 26MB file. Each point marks a different choice of memoization threshold. Memory usage consists only of the input data and the memoization table.

low the memoization table will have high granularity, allowing more re-use of parse results, but the table will have more entries, and the reverse is true for a high threshold. We test a variety of thresholds, and plot the Pareto optimal curve in Figure 9, finding that 128-512 bytes is a good balance in our experimental environment.

7.4 End-to-End Editor

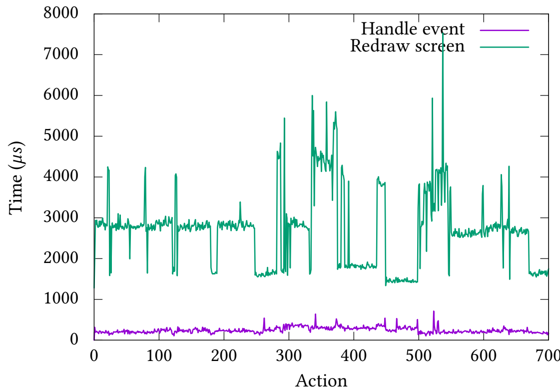
As a complete validation of our incremental parser and syntax highlighter, we implement a text editor with them. The editor uses the window space optimization from Section 4.4, and stores highlighting results in a table before applying a configurable color theme and rendering the viewed window to the screen. With everything put together, we test the editor performance by editing a 50 MB Java file. Actions performed include inserting/deleting characters, seeking to the end of the file, and scrolling. Throughout the actions we observe mostly sub-5ms update time (Figure 10a) and reasonable (roughly 2x) memory overhead (Figure 10b). Redraw time is split between rehighlighting and actually rendering the text to the screen.

8 Conclusion

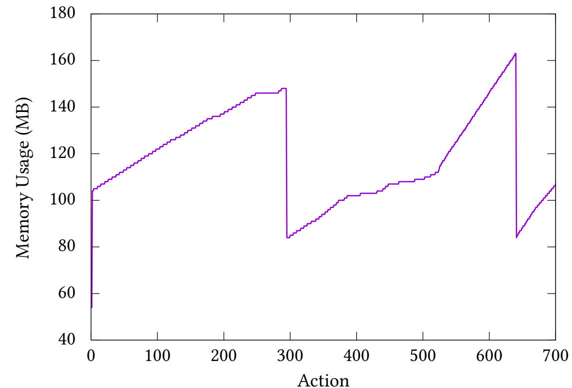
In this paper we present new methods for incremental PEG parsing. We build on *Incremental Packrat Parsing* by modifying the memoization table data structure and parsing strategy to tune the algorithm for incremental parsing. In particular, we present three primary optimizations: the use of an interval tree as a memoization table, lazy shifting in the interval tree, and a parsing strategy that ensures a tree structure for memoization entries in the table. Our changes allow reparsing in time logarithmic in the size of the input for typical edits, compared to linear-time parsing with incremental packrat parsing.

We implement a syntax highlighting library and text editor as demonstrations of our incremental parser and observe strong performance, and show several optimizations to memory usage that are made possible by our incremental parsing strategy.

We believe our results show that GPeg could serve as a performant incremental parser and syntax highlighter in a text editor, and we will be working to integrate it into the next version of the Micro text editor.



(a) Editor performance on a 51 MB Java file.



(b) Editor memory usage on a 51 MB Java file. When the editor initially loads, the highlighter has not yet finished, resulting in low memory usage. Memory usage grows until the Go GC runs, usually when half of allocated memory is garbage.

Figure 10. End-to-end editor performance and memory usage. "Handle event" includes the time to route the event to the correct buffer and binding, apply the edit, and scroll the view. "Redraw screen" includes the time to perform rehighlighting, render the screen to an internal cell buffer, and write the buffer to the terminal emulator.

A Java Highlighter

The Java syntax highlighter used by Flare, based on a Scintillua [5] lexer, is shown below. Grammars include some built-in non-terminals such as `float`, `integer`, `word`, and `space`, as well as functions for capturing (`cap`) and matching a choice of individual words (`words`, which uses a specialized feature to efficiently match a large alternation of words). The top-level repetition pattern is automatically generated using the `token` non-terminal as `{{ token / . (!token .)* }}*`.

```
ws <- space+

line_comment  <- '//' (!'\n' .)*
block_comment <- '/*' (!'*/' .)* '*/'?

comment <- cap{
    line_comment / block_comment,
    "comment"
}

sq_str <- "'" (escape / (!['\n] .))* "'"?
dq_str <- '"' (escape / (!["\n] .))* '"'?
escape <- cap{
    '\\' ['"tnbfr\\],
    "constant.string.escape"
}
string <- cap{sq_str / dq_str, "constant.string"}

number <- cap{
    (float / integer) [LlFfDd]?,
    "constant.number"
}

keyword <- cap{
    words{
        "abstract", "assert", "break", "case",
        "catch", "class", "const", "continue",
        "default", "do", "else", "enum",
        "extends", "final", "for", "goto", "if",
        "implements", "import", "instanceof",
        "interface", "native", "new", "package",
        "private", "protected", "public",
        "return", "static", "strictfp", "super",
        "switch", "synchronized", "this", "throw",
        "throws", "transient", "try", "while",
        "volatile"
    },
    "keyword"
}

bool <- cap{
    words{
        "true", "false", "null"
    },
    "constant.bool"
}

type <- cap{
    words{
        "boolean", "byte", "char", "double",
        "float", "int", "long", "short", "void",
        "Boolean", "Byte", "Character", "Double",
        "Float", "Integer", "Long", "Short",
        "String"
    },
    "type"
}

identifier <- cap{word, "identifier"}
operator   <- cap{
    [+\-/*%<>!=^&|?~:;.()\[\]{}],
    "symbol.operator"
}
annotation <- cap{'@' word, "type.annotation"}
func       <- cap{word, "function"} '('
class      <- cap{'class', "keyword"}
              space+ cap{word, "type.class"}

token <- ws / class / keyword / bool / type / func
    / identifier / string / comment / number
    / annotation / operator
```

References

[1] 2021. JDK 7 webpage. https://openjdk.java.net/projects/jdk7/

[2] Ico Doornekamp. 2021. NPeg webpage. https://github.com/zevv/npeg

[3] Patrick Dubroy and Alessandro Warth. 2017. Incremental Packrat Parsing. In *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering (SLE 2017)*. Association for Computing Machinery, New York, NY, USA, 14–25. https://doi.org/10.1145/3136014.3136022

[4] Max Brunsfeld et al. 2021. Tree Sitter webpage. https://tree-sitter.github.io/tree-sitter/

[5] Mitchell Foral. [n. d.]. Scintillua. https://orbitalquark.github.io/scintillua/index.html

[6] Bryan Ford. 2002. Packrat Parsing: Simple, Powerful, Lazy, Linear Time, Functional Pearl. *SIGPLAN Not.* 37, 9 (Sept. 2002), 36–47. https://doi.org/10.1145/583852.581483

[7] Bryan Ford. 2004. Parsing Expression Grammars: A Recognition-Based Syntactic Foundation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '04)*. Association for Computing Machinery, New York, NY, USA, 111–122. https://doi.org/10.1145/964001.964011

[8] Eclipse Foundation. 2020. Ceylon webpage. https://github.com/eclipse/ceylon

[9] Carlo Ghezzi and Dino Mandrioli. 1979. Incremental Parsing. *ACM Trans. Program. Lang. Syst.* 1, 1 (Jan. 1979), 58–70. https://doi.org/10.1145/357062.357066

[10] Carlo Ghezzi and Dino Mandrioli. 1980. Augmenting Parsers to Support Incrementality. *J. ACM* 27, 3 (July 1980), 564–579. https://doi.org/10.1145/322203.322215

[11] Roberto Ierusalimschy. 2009. A Text Pattern-Matching Tool Based on Parsing Expression Grammars. *Softw. Pract. Exper.* 39, 3 (March 2009), 221–258.

[12] Roberto Ierusalimschy. 2019. LPeg: Parsing Expression Grammars for Lua. http://www.inf.puc-rio.br/~roberto/lpeg

[13] Donald E. Knuth. 1971. Top-down Syntax Analysis. *Acta Inf.* 1, 2 (June 1971), 79–110. https://doi.org/10.1007/BF00289517

[14] Ilya Lakhin. 2013. Papa Carlo webpage. https://lakhin.com/projects/papa-carlo/

[15] Sérgio Medeiros and Roberto Ierusalimschy. 2008. A Parsing Machine for PEGs. In *Proceedings of the 2008 Symposium on Dynamic Languages (DLS '08)*. Association for Computing Machinery, New York, NY, USA, Article 2, 12 pages. https://doi.org/10.1145/1408681.1408683

[16] Arvid M. Murching, Y. V. Prasad, and Y. N. Srikant. 1990. Incremental Recursive Descent Parsing. *Comput. Lang.* 15, 4 (Oct. 1990), 193–204. https://doi.org/10.1016/0096-0551(90)90020-P

[17] J. J. Shilling. 1993. Incremental LL(1) Parsing in Language-Based Editors. *IEEE Trans. Softw. Eng.* 19, 9 (Sept. 1993), 935–940. https://doi.org/10.1109/32.241775

[18] Andrew Snodgrass. 2021. Peg webpage. https://github.com/pointlander/peg

[19] Tim Allen Wagner. 1998. *Practical Algorithms for Incremental Software Development Environments*. Ph.D. Dissertation. USA. UMI Order No. GAX98-03388.

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