Incremental Packrat Parsing

Patrick Dubroy
Y Combinator Research, USA
pat.dubroy@ycr.org

Alessandro Warth
Y Combinator Research, USA
alex.warth@ycr.org

Abstract

Packrat parsing is a popular technique for implementing top-down, unlimited-lookahead parsers that operate in guaranteed linear time. In this paper, we describe a method for turning a standard packrat parser into an *incremental parser* through a simple modification to its memoization strategy. By "incremental", we mean that the parser can perform syntax analysis without completely reparsing the input after each edit operation. This makes packrat parsing suitable for interactive use in code editors and IDEs — even with large inputs. Our experiments show that with our technique, an incremental packrat parser for JavaScript can outperform even a hand-optimized, non-incremental parser.

CCS Concepts • **Software and its engineering → Parsers**;

Keywords packrat parsing, incremental parsing

ACM Reference Format:
Patrick Dubroy and Alessandro Warth. 2017. Incremental Packrat Parsing. In *Proceedings of 2017 ACM SIGPLAN International Conference on Software Language Engineering (SLE'17).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3136014.3136022

1 Introduction

Packrat parsers [3, 4] are backtracking, recursive-descent parsers that support unlimited lookahead while guaranteeing linear parse times. They do this "by saving all intermediate parsing results as they are computed and ensuring that no result is evaluated more than once." [4]

A well-known disadvantage of this technique is its large memory footprint: because a packrat parser "literally squirrels away everything it has ever computed about the input text" [4], its memory consumption also grows linearly with the size of the input. While this usually isn't a problem for moderately-sized inputs, to make packrat parsing practical for larger inputs, researchers have introduced a number of techniques for reducing the size of the parser's *memo table* [1, 8, 11, 16].

In this paper, we approach the packrat parser's memo table not as a problem, but as an opportunity. Namely, we present a straightforward modification to the memoization mechanism used by packrat parsers that enables them to support *incremental parsing* [6, 7]. This is particularly useful in an interactive setting such as a code editor or an Integrated Development Environment (IDE) where, regardless of the input size, near-instantaneous parse times are required in order to provide syntax highlighting, type checking, etc. as part of a responsive user experience.

The main contributions of this paper are:

(a) An algorithm for interactive packrat parsing, which is (to our knowledge) the first such algorithm. It requires **no** changes to the grammars to support incrementality.
(b) Two modifications to the core data structure of packrat parsing, the memo table, that allow our algorithm to perform efficiently on large inputs with real-world grammars.
(c) The JavaScript source code for a simple packrat parser, and for an incremental one based on our algorithm. Together, they show precisely what changes are required to make a standard packrat parser incremental.

Experiments with our prototype implementation in an interactive setting (see Section 4) show that the proposed modification introduces only a small memory overhead (approx. 12%) and results in a huge speedup (two orders of magnitude) compared to a standard packrat parser. When used interactively, our prototype is also faster than Acorn [9], a best-of-breed, non-incremental JavaScript parser.

The rest of this paper is structured as follows: Section 2 provides a brief overview of packrat parsing. Section 3 describes our modification to the memoization mechanism, and two optimizations that make the algorithm more efficient. Section 4 discusses the effects of our strategy on parse times, when used in batch mode as well as incrementally, and compares the performance of our prototype to that of Acorn, a popular non-incremental JavaScript parser. Section 5 discusses related work, and Section 6 concludes. The appendix presents the full JavaScript source code for an incremental parser based on our algorithm.

SLE'17, October 23–24, 2017, Vancouver, Canada

ACM ISBN 978-1-4503-5525-4/17/10…$15.00
https://doi.org/10.1145/3136014.3136022

2 An Overview of Packrat Parsing

The key idea of packrat parsing is that by memoizing *all* intermediate parse results as they are computed, it's possible to guarantee linear parse times even in the presence of backtracking and unlimited lookahead.[1] To understand how this works, consider the following grammar fragment from a simple language of arithmetic expressions[2]:

```
expr = num "+" num
     | num "-" num

num = digit+

digit = "0".."9"
```

When a recursive-descent parser attempts to match the input "869–7" with the `expr` rule shown above, it begins with the first alternative,

```
num "+" num
```

The first term, `num`, matches a sequence of one or more digits. Here, it succeeds after consuming the first three characters from the input stream ("869"). Next, the parser attempts to match a "+" character, which fails because the next character in the input stream is "-". This causes the parser to backtrack to position 0, which is where the current alternative started. Then, the parser tries the second alternative:

```
num "-" num
```

At this point, a conventional recursive-descent parser would apply the `num` rule again, duplicating work that was done for the first alternative — i.e., matching and consuming the digits at position 0, 1, and 2. In a packrat parser, however, the result of applying `num` at position 0 is *memoized* on the first attempt, so almost no work is required this time around. Because the parser already "knows" that `num` succeeds at position 0 after consuming three characters, it can simply update the position to 3 and attempt to match the next part of the pattern ("-"), which succeeds. Finally, the parser applies `num` at position 4, which consumes the final character ("7") and causes the entire parse of `expr` to succeed.

2.1 Memoization in Packrat Parsers

When an intermediate parsing result is memoized (e.g., the result of matching `num` at position 0), the result is stored in the parser's *memo table*. The memo table can be modeled as an $m \times n$ matrix, with a column for each of the n characters in the input, and one row for each rule in the grammar. We refer to each matrix element as a *memo table entry*.

Figure 1 shows the contents of the memo table (using a sparse matrix representation) after matching `expr` as described above. Each memo table entry has two fields:

[1] Packrat parsers are said to support "unlimited lookahead" [4] because in packrat parsing there is little practical difference between backtracking and conventional lookahead (i.e., examining but not consuming tokens).

[2] The concrete syntax is based on a variant of *parsing expression grammars* [5], a grammar formalism closely associated with packrat parsing.

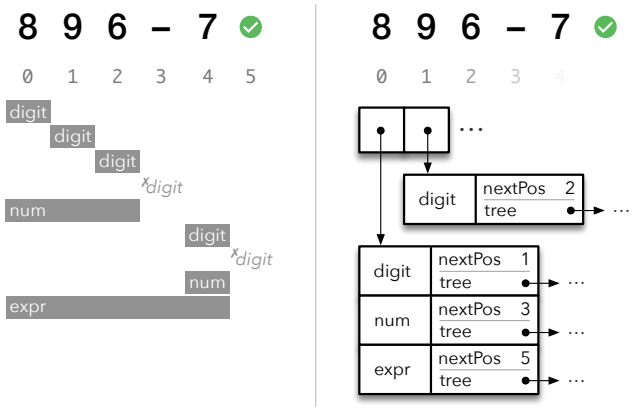


Figure 1. Contents of a packrat parser's memo table after successfully parsing "896-7". *Left*: a compact representation that is used throughout this paper, showing the consumed interval for successful applications (failed applications in italics). *Right*: a detail view showing the contents of the first two columns.

- *nextPos*, which is an offset into the input string indicating where the remaining input begins, and
- *tree*, which contains a parse tree if the application succeeded, or the special value Fail.

Each time a rule r is applied at position p, the parser checks to see if there is a memo table entry for r at that position. If an entry exists, the parser's current position is updated to *nextPos* and the value stored in *tree* is returned. If not, the parser evaluates r and records the results in a new memo table entry. In this way, a packrat parser ensures that no rule is ever evaluated more than once at a given position.

> An important property of packrat parsing and its memoization mechanism is that "the parsing function for each nonterminal depends only on the input string, and not on any other information accumulated during the parsing process." [3]
>
> In other words, **individual memo table entries neither capture nor depend upon the specific *state* of the parser** – "there is 'only one way' to parse a given nonterminal at any given input position." [3] This property is central to our incremental packrat parsing algorithm, which is described in the next section.

3 Incremental Packrat Parsing

The goal of incremental parsing is to efficiently reparse a modified input string by reusing as much as possible from the previous result(s). Conveniently, a regular (non-incremental) packrat parser records all of its intermediate results in a memo table — but that memo table is discarded when parsing completes. The idea of incremental packrat parsing is simple:

to retain the memo table — or as much of it as possible — and enable the intermediate parse results to be used across multiple invocations of the parser.

A standard packrat parser can be modeled by the function:

$$\textsc{Parse} : (G, s) \to T$$

where G is a grammar, s is an input string, and T is a parse tree (or the special value Fail). Similarly, an *incremental* packrat parser can be modeled by:

$$\textsc{Parse} : (G, s, M) \to (M', T)$$

where M and M' are memo tables that (ideally) share some structure. Together, G, s, and M comprise the state or *environment* of the parser, which is carried forward in an explicit environment-passing scheme.

When the input is modified, it will also be necessary to modify the memo table to account for the changes. For this, we introduce a new function:

$$\textsc{ApplyEdit} : (s, M, e) \to (s', M')$$

where e is an *edit operation* consisting of:

- a start position p_s,
- an end position p_e, and
- a replacement string r.

In the general case, applying e means replacing the characters in the interval $[p_s, p_e)$ with the characters in r, which may be of any length. Under this definition, insertion (when $p_s = p_e$) and deletion (when r is empty) are special cases of replacement.

After modifying s to produce s', ApplyEdit produces a new memo table M' which shares as many entries as possible with M. The details of this procedure form the core of our algorithm, which will be described below.

The remainder of this section describes the primary contribution of this paper, namely:

(a) a strategy for detecting which memo table entries are affected by an edit (Section 3.1), allowing the other entries to be reused on the next parse

(b) a modification to the representation of the memo table (Section 3.2) which allows ApplyEdit to efficiently compute M', and

(c) an optimization for our algorithm that allows it to operate more efficiently on large inputs with real-world grammars (Section 3.4).

3.1 Detecting Affected Memo Table Entries

The basic correctness criterion of an incremental parser is that it must produce the same result as parsing from scratch. We say that a memo table entry is *affected* by an edit if, after editing, the entry must be modified or deleted to preserve correctness.

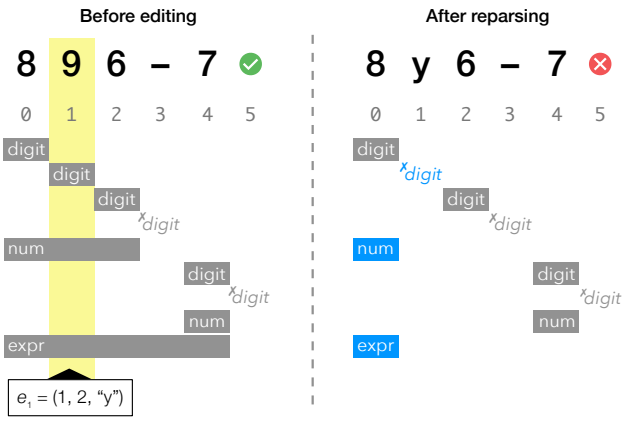


Figure 2. Memo table contents before and after applying edit operation e_1. *Left*, Using the basic overlap rule, ApplyEdit will invalidate the three memo table entries that are affected by the edit. *Right*, the entries shown in blue are new, representing additional work that was done in reparsing the modified input.

> To better explain the technique presented in this paper, we have implemented an interactive memo table visualization, available at https://ohmlang.github.io/sle17/. Readers are encouraged to use it to follow along with the examples in this section.

In explaining our strategy for detecting affected memo table entries, we will first discuss a subset of possible edit operations: replacement operations that do not change the length of the input. After applying the edit to the input string, the next step is to determine which (if any) of the entries are affected by the edit.

For example, consider the input "896–7" which was used in Section 1, and an edit operation $e_1 = (1, 2, \text{"}y\text{"})$ that replaces the "9" with a "y". After applying the operation, the new input string is "8y6–7", as shown in Figure 2.

Previously, the `digit` rule succeeded at position 1; but now, in a from-scratch parse of the modified input, it will fail (since "y" is not a digit). To preserve correctness, we can remove the entry for `digit` at position 1 from the memo table, ensuring that the parser will re-attempt `digit` at that position.

3.1.1 Basic "Overlap Rule"

In general, we can observe that if the extent of a memo table entry overlaps with an edit operation e, then the entry is possibly affected by e. Depending on the edit, it may *not* be affected (e.g., if the replacement text is the same as the original text), but we adopt a conservative approach: any memo table entry overlapping the edit is considered *invalid* and will be removed from the memo table.

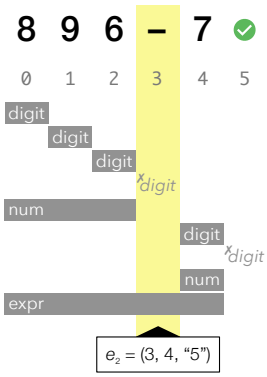


Figure 3. Memo table contents before applying edit operation e_2. The simple "overlap rule" does not detect that 'num' at position 0 is affected by the change.

By applying this "overlap rule" to the entire memo table, we see that the entries for `expr` and `num` at position 0 must also be removed (Figure 2, left). After re-parsing (Figure 2, right), there are two new entries at position 0 (`num` and `expr`) and a new entry for `digit` at position 1. `num` still succeeds at position 0, but now only consumes a single character ("8"). `expr` also succeeds and consumes the "8", but the entire parse now fails because `expr` did not consume the entire input.

3.1.2 Problem: Basic Overlap is not Enough

Now, consider a different edit operation $e_2 = (3, 4, \text{"5"})$, shown in Figure 3, which modifies the original input by replacing the "-" with "5". Using the same "overlap rule" as before, only two entries would be invalidated: `digit` at position 3 and `expr` at position 0. However, this is not sufficient: an application of `num` at position 0 should now consume the entire input, yet the memo table entry only consumes "896", which would lead to an incorrect incremental parse.

This example demonstrates that the basic "overlap rule" is not sufficient to detect all invalid memo table entries. Although the application of `num` at position 0 did not *consume* the character "6" at position 3, it does depend on it via the greedy repetition expression `digit+`, which is responsible for the application of `digit` at position 3. Via `digit`, the `num` rule must *examine* the next character, even if it doesn't end up consuming it — as is the case here.

Note that the unlimited lookahead capability of packrat parsing allows a rule to examine any number of characters, regardless of how many it consumes. For example:

```
allOrNothing = "abcdefg"
             | ""
```

When matched against "abcdef!", this rule will examine the entire input for its first alternative, then backtrack and succeed on the second alternative, consuming nothing.

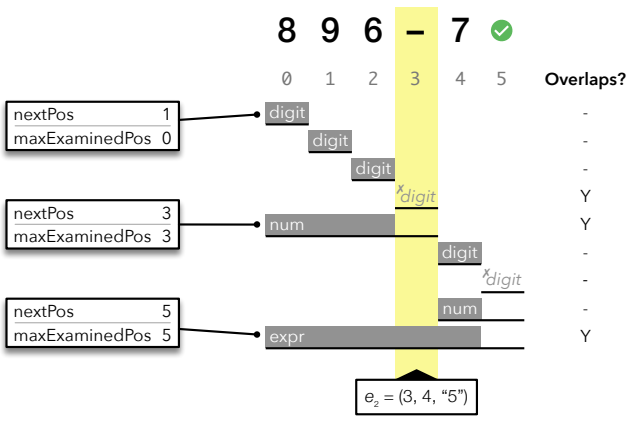


Figure 4. Memo table contents after parsing "896-7". The entry for `num` at position 0 is invalidated by e_2, because its *examined interval* overlaps with the edit, though its *consumed interval* does not.

3.1.3 Solution: Maximum Examined Position

To address this, we introduce another field to memo table entries, called *maximum examined position* or *maxExaminedPos*, which records the furthest position examined in the input stream over the entire course of parsing a rule. An input position is *examined* when either (a) the character at that position is consumed, or (b) the value of the character is used to make a parsing decision.

In practical terms, the *examined interval* (defined as the closed interval $[p, maxExaminedPos]$) of a rule application contains all of the characters that could have influenced the result of parsing that rule. By definition, the examined interval of an expression covers the examined intervals of all its subexpressions.

Figure 4 shows a memo table whose entries record both the next position and the maximum examined position. Note that the examined interval for `num` at position 0 does in fact overlap with the edit operation e_2. Thus, we can preserve correctness by invalidating entries based on their examined interval rather than their consumed interval.

Intuitively, this should make sense: the examined interval of a rule application represents the entire portion of the input that could possibly have influenced its result. The memo table entry should be invalidated if and only if the edit operation affects that portion of the input.

3.2 Relocating Memo Table Entries

As long as the length of the input does not change, the invalidation strategy described above is sufficient. After invalidating any overlapping entries, the remainder of the memo table can be reused to parse the new input. However, when the length of the input *does* change, such as when an insertion or deletion occurs, some of the memo table entries will require additional processing.

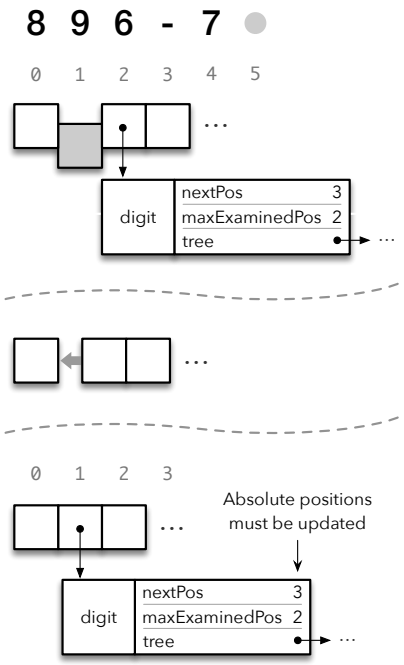


Figure 5. Deleting the character at position 1 means deleting the corresponding column from the memo table. The entry for `digit` at position 3 is now at position 2, requiring $nextPos$ and $maxExaminedPos$ to be updated accordingly.

For example, consider an edit $e_3 = (1, 2, \text{‘’})$ which deletes the "9" from the original input, resulting in "86–7". When the "9" is deleted, the characters past position 1 ("6–7") are shifted left by one position. After modifying the input, we must change the memo table accordingly, which means deleting column 1 and shifting the other columns left by one position, as shown in Figure 5.

The entry for `digit` that was previously at position 2 is now at position 1. However, its *nextPos* value is 3, which would indicate that it consumes *two* characters (which isn't right). Since *nextPos* and *maxExaminedPos* are absolute positions, their values need to be updated each time a memo table entry is relocated.

In a real-world scenario, such as inserting characters at the beginning of a large file, the cost of such updates would eliminate most of the benefits of incremental parsing. Our solution to this problem is to make the individual memo table entries position-independent, so that they be relocated at no extra cost. To do this, we replace the *nextPos* property of memo table entries with a relative offset, which we call *matchLength*. Similarly, the examined interval of an entry is stored as *examinedLength*. This is a straightforward change in most packrat parsing implementations, and does not affect the asymptotic complexity of parsing.

3.3 An Algorithm for ApplyEdit

In Section 3.1, we described how we use the *examined interval* to detect memo table entries which are invalidated by an edit operation. In Section 3.2, we explained how we store memo table entries in a position-independent manner so that they can be relocated efficiently. Now we describe how these two techniques can be used together to implement the ApplyEdit procedure.

Recall the definition of ApplyEdit which we introduced at the beginning of this section:

$$\textsc{ApplyEdit} : (s, M, e) \rightarrow (s', M')$$

Given an input string s, a memo table M, and an edit operation e, ApplyEdit produces the modified input string s', and a new memo table M' that can be used to incrementally parse s'. A basic implementation consists of the following steps:

Step 1 Apply the edit operation to s, producing s'.
Step 2 Adjust the memo table: remove all entries from the columns inside the edit interval, and, if necessary, add or delete columns and relocate any columns to the right of the interval.
Step 3 Scan the memo table, removing any remaining entries whose examined interval overlaps the edit interval. Finally, return s' and M'.

Note that in Step 3, only the portion to the left of the edit interval needs to be scanned: invalid entries that begin *inside* the edit interval are removed in Step 2, and any entries to the right (the relocated entries) cannot be affected by the edit. However, it is possible that those entries are no longer used in the current parse — e.g., if the edit caused those characters to become part of a comment.

Together, these three steps make up a complete implementation of ApplyEdit and the core of our technique for incremental packrat parsing. In the next section, we will discuss one way that it can be optimized.

3.4 Analysis and Optimization

The implementation of ApplyEdit presented above has a complexity of $O(m \times n)$, where n is the size of the input and m is the number of rules in the grammar. Assuming that string concatenation (step 1) and array concatenation (step 2) are both $O(n)$, the running time is dominated by the memo table invalidation in step 3. In the worst case, invalidation requires visiting $O(n)$ columns in the memo table, and at each column, scanning $O(m)$ memo table entries to check if they overlap the edit interval. While this does not affect the asymptotic complexity of packrat parsing (which is already $O(m \times n)$), it can result in poor performance in real-world use.

3.4.1 Maximum Examined Length

To improve the performance of our invalidation algorithm, we modify the memo table layout so that each column can keep track of the largest examined interval of all its entries. We store this value in a per-column property called *maxExaminedLength*. This way, when we are scanning the columns in step 3 of ApplyEdit, we can avoid scanning all the entries in a column if *maxExaminedLength* shows that none of its entries overlap with the edit.

During parsing, a column's *maxExaminedLength* can only grow, because memo table entries are never deleted during parsing. This means that *maxExaminedLength* can be updated in constant time whenever a new memo table entry is added to the column.

Individual memo table entries can only deleted when ApplyEdit is already scanning all of the entries in a given column. Thus, the new value of *maxExaminedLength* (in case it is smaller) can also be calculated with a constant amount of extra work. Therefore, maintaining the maximum examined length on a per-column basis does not affect the asymptotic complexity of either our invalidation algorithm, or packrat parsing in general.

3.5 Putting It All Together

Figure 6 shows the final version of our ApplyEdit algorithm, including the *maximum examined length* optimization. Given the input string s, the memo table M, and an edit operation e, ApplyEdit produces the modified input s' and a new memo table M' which shares with M any entries not affected by the edit.

To perform an incremental parse, the results of ApplyEdit are passed to Parse, which makes use of the pre-filled memo table entries, and returns a new memo table with potentially more entries. In a code editor or IDE, a typical use would be to reparse the input after applying every edit:

M = a new memo table
s = an empty string
for each edit operation e
  s, M = ApplyEdit(s, M, e)
  M = Parse(G, s, M)

Alternatively, edits can be batched together by calling ApplyEdit multiple times before passing the results to Parse.

The implementation of Parse is largely the same as in a standard packrat parser, with the following exceptions:

1. It maintains the *examinedLength* property of memo table entries and the *maxExaminedPos* property of memo table columns.
2. It returns its memo table at the end of the parse.

We do not present an algorithm for (1) here, as the details are implementation-dependent (though straightforward). For an example, see the `match` methods of `IncrementalMatcher` and `IncRuleApplication` in the appendix of this paper.

```
ApplyEdit(s, M, e)
    ▷ Step 1: Apply the edit to s
1   s' = Concat(s[0..e.pos_s], e.r, s[e.pos_e..])

    ▷ Step 2: Adjust the memo table
2   M' = Concat(M[0..e.pos_s],
                a list of e.r.length new columns,
                M[e.pos_e..])

    ▷ Step 3: Invalidate overlapping entries
3   for i = 0 to e.pos_s
4       col = M'[i]
5       // Does any entry in this column overlap e?
6       if i + col.maxExaminedLength ≤ e.pos_s
7           continue to next column

8       newMax = 0
9       for each entry in col
10          if i + entry.examinedLength > e.pos_s
11              Delete entry from M'
12          elseif entry.examinedLength > newMax
13              newMax = entry.examinedLength
14      col.maxExaminedLength = newMax
15  return s', M'
```

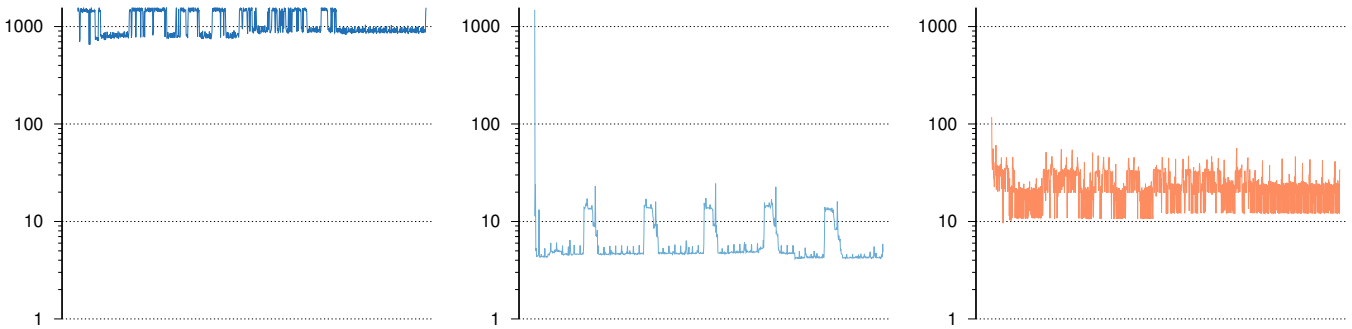
Figure 6. Pseudocode for ApplyEdit, the core of incremental packrat parsing. Lines 5-7, 8, and 12-14 implement the *maximum examined length* optimization (see Section 3.4.1).

4 Evaluation

To evaluate the performance of our algorithm in real-world use, we implemented two different packrat parsers for the ECMAScript 5.1 language (ES5), a widely-supported version of JavaScript (the parsers themselves are also written in JavaScript).

> Our incremental packrat parsing algorithm was originally developed for Ohm [20, 21], our open-source packrat parsing framework. In order to better evaluate the techniques presented in this paper, we built a minimal packrat parsing library in JavaScript (presented in the appendix), which is the basis for the ES5 parsers described in this section. The full source code for the library and the ES5 grammar can be found at https://ohmlang.github.io/sle17/.

The first parser is a standard (non-incremental) packrat parser, implemented using an object-oriented version of parser combinators. We refer to this as "Es5-Standard". The second parser ("Es5-Incremental") is an incremental parser using the techniques described in this paper. Finally,



(a) Es5-Standard **(b)** Es5-Incremental **(c)** Acorn

Figure 7. Response times in milliseconds (log scale) for a series of 891 simulated edits to a 279 KB JavaScript source file. Each measurement includes the time to apply the edit to the input string as well as the time to parse the modified input.

we also compare both our parsers against Acorn, a popular, high-performant, hand-optimized JavaScript parser.

All benchmarks were run on an Apple MacBook Pro (Retina, 13-inch, Early 2015) with a 2.9 GHz Intel Core i5 processor and 16 GB RAM, running OSX version 10.11.6 "El Capitan" and Node.js version 6.1.0. For Acorn, we used version 5.1.2 with the default options, except that `ecmaVersion` option was set to 5 (which selects the latest version of ES5, i.e., ECMAScript 5.1).

4.1 Parsing Performance

To evaluate the performance of the parsers in a representative setting, we created a benchmark that simulates a typical source code editing session. To do so, we recorded every keystroke (typos and all) as one of the authors manually retyped the contents of a recent commit to a single, large (279 KB, 4761 SLOC) file in Ohm [20, 21], our open-source parser generator.[3] The reified edit log consisted of 891 edit operations roughly clustered around the middle of the file.

Figure 7 graphs the response times for each parser over the course of the editing session. This measurement includes the time to apply the edit to the input string as well as the time to reparse the new string.

On the initial parse, Es5-Standard (Fig. 7a) and Es5-Incremental (Fig. 7a) are significantly slower than Acorn (Fig. 7c), requiring 1562ms and 1483ms respectively. Acorn is more than an order of magnitude faster, taking only 118ms.

On subsequent parses, the incremental parser is consistently faster than the non-incremental one — reparsing the modified source roughly two orders of magnitude faster (mean 6.2ms, median 4.7ms). On average, it also outperforms Acorn (mean 23.7ms, median 23.6ms) by a significant margin.

The reason for the differences should be clear: each time an edit happens, Es5-Standard and Acorn must parse the entire source code from scratch, while subsequent parses by Es5-Incremental require only a small amount of extra work, which is why the parse times are only around 5–6ms.

The bimodal performance of Es5-Standard (Fig. 7a) — which can also be seen in Acorn's results — is due to the fact that some edits leave the source code in a syntactically-invalid state, which results in a faster parse. The spikes in Figure 7b are related to garbage collection: they correspond to times where incremental marking is active, resulting in periods of reduced JavaScript performance.

4.1.1 Improving Initial Parse Time

The pure throughput of our naively-written packrat parsers is not at all competitive with a hand-optimized parser, as the initial parse times in Figure 7 clearly show. Fortunately, the literature on packrat parsing provides a number of optimizations that could be used to improve the throughput [8, 11, 16], which we discuss in Section 5.

Additionally, our incremental parser can be modified to support a soft cap on response times by returning from Parse before the parse is complete. This allows the UI to remain responsive while the initial parse makes progress in small increments. In some use cases (e.g. syntax highlighting), the partial results of the parse may even be immediately useful.

4.2 Space Efficiency

The major downside of packrat parsing is that its use of memoization results in high memory usage in typical workloads. The technique described in this paper does not directly address this issue; in fact, it slightly increases the memory requirements of packrat parsing, as we discuss below. However, we argue that incremental parsing is a way to get more value in the space-time tradeoff, as it greatly increases the amount of time saved per byte of storage.

In Section 3, we described the memo table layout required by our technique. The main difference from a standard packrat parser is the *examinedLength* property that is added to every memo table entry. Though it means storing at least

[3] Based on a recent empirical study of source code file sizes [10], this is in the 99th percentile.

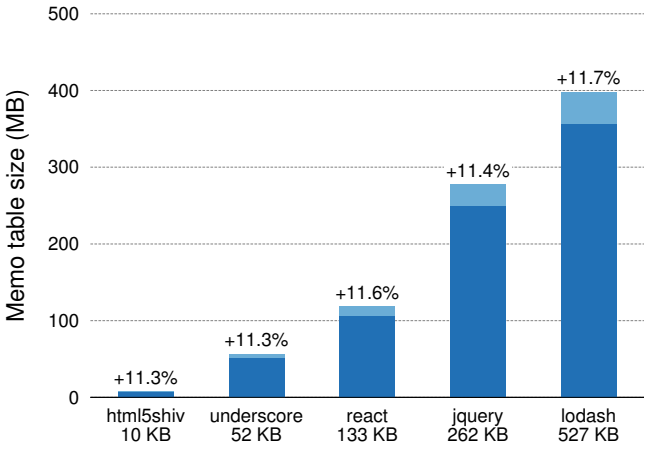


Figure 8. Memory usage for the memo table in Es5-Standard (*dark blue*) and Es5-Incremental (*light blue*) after parsing several popular JavaScript libraries.

three pieces of information per entry instead of just two, the actual amount of extra memory this requires is highly implementation dependent. Figure 8 shows the experimental results of the memory usage required by our standard and incremental ES5 parsers to parse five different JavaScript libraries of varying size.

The mean additional memory usage for the memo table is 11.5%. The majority of this is due to the extra field (*examinedLength*) stored in each memo table entry, and the remainder can be attributed to per-column *maxExaminedLength* property required by our technique. The small variation in the additional memory usage (11.3–11.7%) is likely due to the fact that different inputs will have a varying number of memo table entries per column.

4.3 Discussion

For user interface responsiveness, 100ms is recognized in the human-computer interaction literature as the upper limit at which a system is perceived to be reacting "instantaneously" [15][2]. However, a naively-implemented packrat parser can take much longer than this to parse even moderately-sized inputs.

A common solution is to perform parsing on a background thread. However, this adds significant implementation complexity — especially in JavaScript, which does not (yet) support threads with shared memory. And if the editor is relying on the parser to produce layout and styling information, then long response times will still result in a degraded user experience.

As we have demonstrated in this section, our incremental packrat parsing technique offers extremely low parse times in interactive use, at the cost of some extra memory usage for the memo table. We believe this to be a worthwhile trade-off for in cases where the inputs may be large (as in our experimental evaluation) and where instantaneous feedback is desired.

5 Related Work

5.1 Incremental Parsing

The idea of incremental parsing was first introduced by Ghezzi and Mandrioli [6, 7] as an extension of LR parsing. Much of the research that followed [14, 19] focused on improving the performance of incremental shift-reduce parsers, and on attaining optimal node reuse [19]. In contrast, our technique is based on packrat parsing, which differs from LR(k) in that is top-down and supports unlimited lookahead.

Techniques for incremental top-down parsing [17, 18] have mostly focused on LL(1) grammars, which is more restrictive than the class of languages support by packrat parsing. These solutions have mostly focused on single-site editing that is tightly integrated with an editor. Our algorithm supports any number of edit sites (via repeated calls to ApplyEdit between calls to Parse) and requires no special editor integration, except the ability to detect and react to the user's edit operations.

Papa Carlo [13] is a parsing library for Scala that can be used to build incremental parsers. While it is based on Parsing Expression Grammars (PEGs) [5], it is not a true packrat parser, as it only employs memoization in a limited way, and therefore does not guarantee linear parse times.

Support for incremental parsing in Papa Carlo is based on a notion of *source code fragments*, which are (possibly nested) substrings of the input for which parsing results are cached. The author of a grammar must manually define the syntax of its code fragments — e.g., for C/C++ one type of code fragment would be anything between "{" and "}" tokens — and ensure that "their syntactical meaning [is] invariant to [their] internal content" [13], i.e., the code inside a fragment must always be parsed by the same rule in the grammar [12].

In contrast, by leveraging the packrat parser's memoization mechanism, our approach does not introduce any new concepts that must be understood by grammar authors, or require them to do any additional work in order to enjoy the benefits of incremental parsing. Additionally, our technique makes all partial parsing results available for reuse, not just the results of selected rules.

5.2 Optimization of Packrat Parsers

As discussed in Section 4, several researchers have introduced techniques for reducing the size of a packrat parser's memo table, with the aim of both reducing memory usage and improving throughput. For example, some packrat parser implementations allow the grammar author to restrict the use of memoization to a subset of the rules in the grammar [1, 8]. Others have proposed similar, automated approaches based on static [8, 16] or dynamic [11] analysis of the grammars.

Most of these techniques are also applicable to incremental packrat parsers, though special care should be taken in order to maintain the parser's linear time guarantee (as is true for standard, non-incremental packrat parsers). Also, many of these optimizations improve batch performance, but their reduced use of memoization can negatively impact incremental response time. In incremental packrat parsing, *any* memoized result could prove useful in the future, and one must be careful to avoid requiring "too much" work to be done in response to each edit operation.

6 Conclusions and Future Work

In this paper, we presented an algorithm for incremental packrat parsing — to our knowledge, the first such algorithm. Our technique is based on a slight modification to the standard packrat memoization strategy, and it requires no grammar modification to achieve incrementality. Its simplicity is demonstrated by our inclusion, in the appendix of this paper, of the full JavaScript source code for both a standard packrat parser and an incremental variant.

We described two key optimizations — relocatable memo table entries and the per-column *maximum examined length* property — which ensure that our algorithm is efficient on large inputs with real-world grammars.

In an experimental evaluation, we compared an incremental parser based on our algorithm to a standard packrat parser. Our experiments show that our algorithm delivers large performance gains (two orders of magnitude) at the cost of approximately 12% more memory usage and only slightly worse batch parsing performance.

We also showed that with a large input in interactive use, a naively-implemented packrat parser that uses our algorithm can outperform a hand-optimized, non-incremental JavaScript parser.

In the future, we plan to investigate how the packrat parsing optimizations described in Section 5 can be combined with our algorithm, in order to balance memory usage and batch parsing performance with low incremental response times.

Also, we would like to explore how the algorithm presented in this paper can be combined with incremental semantic analysis. This could be used to support such features as incremental type checking and type-directed autocompletion.

We have already successfully implemented our technique in Ohm, a popular PEG-based parsing toolkit [20, 21] for JavaScript. As the Ohm community experiments with our implementation, we hope to learn more about its performance in a wider set of use cases.

Appendix: Source Code

This section presents the JavaScript (ES6) source code of a set of classes for building incremental and non-incremental packrat parsers. The incremental functionality is contained in the `IncrementalMatcher` and `IncRuleApplication` classes.

```
class Matcher {
  constructor(rules) {
    this.rules = rules;
  }

  match(input) {
    this.input = input;
    this.pos = 0;
    this.memoTable = [];
    var cst = new RuleApplication('start').eval(this);
    if (this.pos === this.input.length) {
      return cst;
    }
    return null;
  }

  hasMemoizedResult(ruleName) {
    var col = this.memoTable[this.pos];
    return col && col.has(ruleName);
  }

  memoizeResult(pos, ruleName, cst) {
    var col = this.memoTable[pos];
    if (!col) {
      col = this.memoTable[pos] = new Map();
    }
    if (cst !== null) {
      col.set(ruleName,
              {cst: cst, nextPos: this.pos});
    } else {
      col.set(ruleName, {cst: null});
    }
  }

  useMemoizedResult(ruleName) {
    var col = this.memoTable[this.pos];
    var result = col.get(ruleName);
    if (result.cst !== null) {
      this.pos = result.nextPos;
      return result.cst;
    }
    return null;
  }

  consume(c) {
    if (this.input[this.pos] === c) {
      this.pos++;
      return true;
    }
    return false;
  }
}
```

```
class IncrementalMatcher {
  constructor(rules) {
    this.rules = rules;
    this.memoTable = [];
    this.input = '';
  }

  match() {
    this.pos = 0;
    this.maxExaminedPos = -1;
    var cst =
        new IncRuleApplication('start').eval(this);

    if (this.pos === this.input.length) {
      return cst;
    } else {
      return null;
    }
  }

  hasMemoizedResult(ruleName) {
    var col = this.memoTable[this.pos];
    return col && col.memo.has(ruleName);
  }

  memoizeResult(pos, ruleName, cst) {
    var col = this.memoTable[pos];
    if (!col) {
      col = this.memoTable[pos] = {
        memo: new Map(),
        maxExaminedLength: -1
      };
    }
    var examinedLength =
        this.maxExaminedPos - pos + 1;
    if (cst !== null) {
      col.memo.set(ruleName, {
        cst: cst,
        matchLength: this.pos - pos,
        examinedLength: examinedLength
      });
    } else {
      col.memo.set(ruleName, {
        cst: null,
        examinedLength: examinedLength
      });
    }
    col.maxExaminedLength = Math.max(
        col.maxExaminedLength,
        examinedLength);
  }

  useMemoizedResult(ruleName) {
    var col = this.memoTable[this.pos];
    var result = col.memo.get(ruleName);
    this.maxExaminedPos = Math.max(
        this.maxExaminedPos,
        this.pos + result.examinedLength - 1);
    if (result.cst !== null) {
      this.pos += result.matchLength;
      return result.cst;
    }
    return null;
  }

  consume(c) {
    this.maxExaminedPos =
        Math.max(this.maxExaminedPos, this.pos);
    if (this.input[this.pos] === c) {
      this.pos++;
      return true;
    }
    return false;
  }

  applyEdit(startPos, endPos, r) {
    var s = this.input;
    var m = this.memoTable;

    // Step 1: Apply edit to the input
    this.input =
        s.slice(0, startPos) + r + s.slice(endPos);

    // Step 2: Adjust memo table
    this.memoTable = m.slice(0, startPos).concat(
        new Array(r.length).fill(null),
        m.slice(endPos));

    // Step 3: Invalidate overlapping entries
    for (var pos = 0; pos < startPos; pos++) {
      var col = m[pos];
      if (col != null &&
          pos + col.maxExaminedLength > startPos) {
        var newMax = 0;
        for (var [ruleName, entry] of col.memo) {
          var examinedLen = entry.examinedLength;
          if (pos + examinedLen > startPos) {
            col.memo.delete(ruleName);
          } else if (examinedLen > newMax) {
            newMax = examinedLen;
          }
        }
        col.maxExaminedLength = newMax;
      }
    }
  }
}
```

```
class RuleApplication {
  constructor(ruleName) {
    this.ruleName = ruleName;
  }

  eval(matcher) {
    var name = this.ruleName;
    if (matcher.hasMemoizedResult(name)) {
      return matcher.useMemoizedResult(name);
    } else {
      var origPos = matcher.pos;
      var cst = matcher.rules[name].eval(matcher);
      matcher.memoizeResult(origPos, name, cst);
      return cst;
    }
  }
}

class IncRuleApplication {
  constructor(ruleName) {
    this.ruleName = ruleName;
  }

  eval(matcher) {
    var name = this.ruleName;
    if (matcher.hasMemoizedResult(name)) {
      return matcher.useMemoizedResult(name);
    } else {
      var origPos = matcher.pos;
      var origMax = matcher.maxExaminedPos;
      matcher.maxExaminedPos = -1;
      var cst = matcher.rules[name].eval(matcher);
      matcher.memoizeResult(origPos, name, cst);
      matcher.maxExaminedPos = Math.max(
          matcher.maxExaminedPos,
          origMax);
      return cst;
    }
  }
}

class Terminal {
  constructor(str) {
    this.str = str;
  }

  eval(matcher) {
    for (var i = 0; i < this.str.length; i++) {
      if (!matcher.consume(this.str[i])) {
        return null;
      }
    }
    return this.str;
  }
}

class Choice {
  constructor(exps) {
    this.exps = exps;
  }

  eval(matcher) {
    var origPos = matcher.pos;
    for (var i = 0; i < this.exps.length; i++) {
      matcher.pos = origPos;
      var cst = this.exps[i].eval(matcher);
      if (cst !== null) {
        return cst;
      }
    }
    return null;
  }
}

class Sequence {
  constructor(exps) {
    this.exps = exps;
  }

  eval(matcher) {
    var ans = [];
    for (var i = 0; i < this.exps.length; i++) {
      var exp = this.exps[i];
      var cst = exp.eval(matcher);
      if (cst === null) {
        return null;
      }
      if (!(exp instanceof Not)) {
        ans.push(cst);
      }
    }
    return ans;
  }
}

class Not {
  constructor(exp) {
    this.exp = exp;
  }

  eval(matcher) {
    var origPos = matcher.pos;
    if (this.exp.eval(matcher) === null) {
      matcher.pos = origPos;
      return true;
    }
    return null;
  }
}
```

```
class Repetition {
  constructor(exp) {
    this.exp = exp;
  }

  eval(matcher) {
    var ans = [];
    while (true) {
      var origPos = matcher.pos;
      var cst = this.exp.eval(matcher);
      if (cst === null) {
        matcher.pos = origPos;
        break;
      } else {
        ans.push(cst);
      }
    }
    return ans;
  }
}
```

Acknowledgments

The authors would like to thank Jonathan Edwards, Marijn Haverbeke, Marko Röder, Nada Amin, Saketh Kasibatla, Sean McDirmid, Yoshiki Ohshima, and the anonymous reviewers for feedback on this work and earlier drafts of the paper.

References

[1] Ralph Becket and Zoltan Somogyi. 2008. DCGs + Memoing = Packrat Parsing but Is It Worth It?. In *Proc. of Practical Aspects of Declarative Languages: 10th International Symposium (PADL 2008)*. Springer, 182–196. https://doi.org/10.1007/978-3-540-77442-6_13

[2] Stuart K. Card, Thomas P. Moran, and Allen Newell. 1980. The Keystroke-Level Model for User Performance Time with Interactive Systems. *Commun. ACM* 23, 7 (1980), 396–410. https://doi.org/10.1145/358886.358895

[3] Bryan Ford. 2002. *Packrat Parsing: A Practical Linear-Time Algorithm with Backtracking*. Master's thesis. Massachusetts Institute of Technology.

[4] Bryan Ford. 2002. Packrat Parsing: Simple, Powerful, Lazy, Linear Time. In *Proc. of the Seventh ACM SIGPLAN International Conference on Functional Programming (ICFP '02)*, Vol. 37. ACM, 36–47. https://doi.org/10.1145/581478.581483

[5] Bryan Ford. 2004. Parsing Expression Grammars: A Recognition-Based Syntactic Foundation. In *Proc. of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '04)*. ACM, 111–122. https://doi.org/10.1145/964001.964011

[6] Carlo Ghezzi and Dino Mandrioli. 1979. Incremental Parsing. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 1, 1 (Jan. 1979), 58–70. https://doi.org/10.1145/357062.357066

[7] 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

[8] Robert Grimm. 2006. Better Extensibility Through Modular Syntax. In *Proc. of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '06)*. ACM, 38–51. https://doi.org/10.1145/1133981.1133987

[9] Marijn Haverbeke. 2017. Acorn webpage. (2017). https://github.com/ternjs/acorn

[10] Israel Herraiz, Daniel M. German, and Ahmed E. Hassan. 2011. On the Distribution of Source Code File Sizes.. In *ICSOFT (2)*. 5–14.

[11] Kimio Kuramitsu. 2015. Packrat Parsing with Elastic Sliding Window. *Journal of Information Processing* 23, 4 (7 2015), 505–512. https://doi.org/10.2197/ipsjjip.23.505

[12] Ilya Lakhin. 2013. Incremental Parser Based on Invariant Syntax Fragments (LtU post). (2013). http://lambda-the-ultimate.org/node/4840

[13] Ilya Lakhin. 2013. Papa Carlo webpage. (2013). http://lakhin.com/projects/papa-carlo/

[14] J.-M. Larchevêque. 1995. Optimal Incremental Parsing. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 17, 1 (Jan. 1995), 1–15. https://doi.org/10.1145/200994.200996

[15] Robert B. Miller. 1968. Response Time in Man-Computer Conversational Transactions. In *Proc. of the December 9–11, 1968, Fall Joint Computer Conference, Part I*. ACM, 267–277. https://doi.org/10.1145/1476589.1476628

[16] Kota Mizushima et al. 2010. Packrat Parsers Can Handle Practical Grammars in Mostly Constant Space. In *Proc. of PASTE '10*. ACM, 29–36. https://doi.org/10.1145/1806672.1806679

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

[18] John J. Shilling. 1993. Incremental LL (1) Parsing in Language-Based Editors. *IEEE Transactions on Software Engineering* 19, 9 (1993), 935–940. https://doi.org/10.1109/32.241775

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

[20] Alessandro Warth, Patrick Dubroy, et al. 2016. Ohm webpage. (2016). https://ohmlang.github.io/

[21] Alessandro Warth, Patrick Dubroy, and Tony Garnock-Jones. 2016. Modular Semantic Actions. In *Proc. of the 12th Symposium on Dynamic Languages (DLS 2016)*. ACM, 108–119. https://doi.org/10.1145/2989225.2989231