Gyalpozhing College of Information Technology
Royal University of Bhutan

CAE305: COMPILERS

Case Study by Group 3

Review on "Incremental Packrat Parsing"

Authored by Patrick Dubroy and Alessandro Warth

Member:

Dechen Pelden (12190004)

Dorji Wangmo (12190006)

Sancha Bir Subba (12190020)

Tandin Wangchuk (12190027)

Sonam (12190024)

1. Introduction

Packrat parser is a backtracking, recursive-descent parser that supports unlimited lookahead. They save all the intermediate parsing results as they are computed ensuring that no result is evaluated more than once. It has its disadvantage which is well known by the memory footprint, the memory consumption grows linearly with the size of the input. To overcome this problem, they have used straightforward modification to the memoization mechanism to support incremental parsing. Some of the contributions of this paper are an algorithm for the packrat parsing (which requires no changes to the grammars to support incrementality), two modifications to the core data structure and the memo table which will allow an algorithm to perform efficiently on large input.

2. An Overview of Packrat Parsing

The main idea of packrat parsing is that by memoizing all intermediate parse results as they are computed. For example, let's take language of arithmetic expression;

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

num = digit +

digit = "0"..”9”

And the input strings are "869 -7".

The iteration begins with the num "+" num. In this, the first term num matches and succeeds after consuming the first three characters from the input stream("869"). In the next step when the parser tries to match "+" character, it fails as in the input string it has "-". And this is where backtracking to position 0 occurs. And it tries for the alternative num "-" num. Here the work will be duplicated which was done earlier because result of applying num at position 0 is memoized on the first attempt and it attempts to match the next part of the pattern ("-") and

finally applies to 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 the intermediate parser is memoized then the result is stored in the parser's memo table. Usually the memo table is modeled as an m x n matrix, with a column for each of n characters in the input, and one row for each rule in the grammar. For example the memory table of the above example could be:

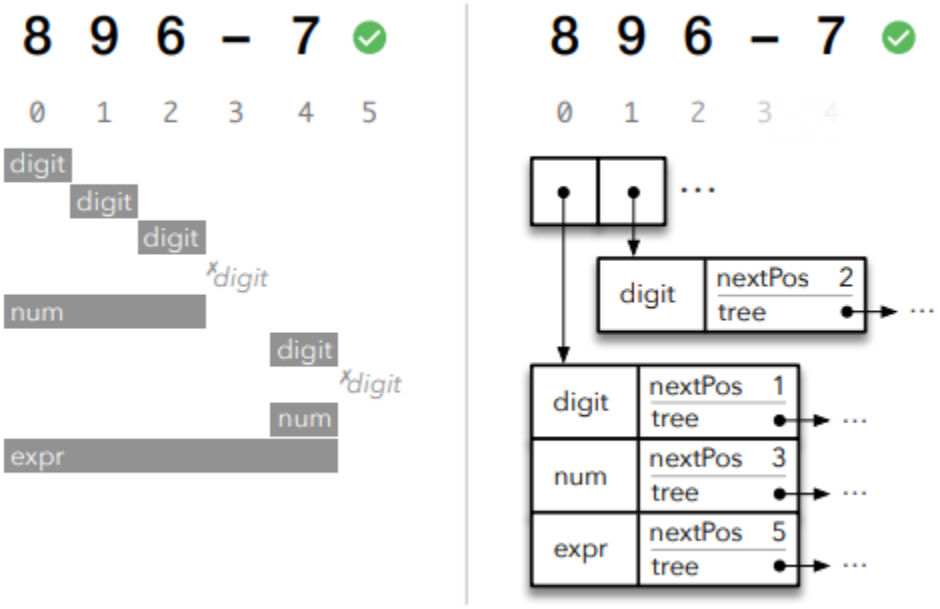

Figure 1: Contents of a packrat parser's memo table after successfully parsing " ł896-7". Left: showing the consumed interval for successful applications (failed applications in italics). Right: a detail view showing the contents of the first two columns.

The memory table has two fields:

- nextPos: indicates where the remaining input begins.
- tree: contains a parse tree if the application succeeded.

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

3. Incremental Packrat Parsing

The purpose of incremental parsing is to reparse a modified input text as quickly as feasible by reusing as much of the prior result as possible (s). A standard (non-incremental) packrat parser conveniently records all of its intermediate results in a memo table, which is then deleted when parsing is finished. The concept of incremental packrat parsing is straightforward: should keep the memo table — or as much of it as feasible — and make the intermediate parse results usable across successive parser calls.

3.1 Detecting Affected Memo Table Entries

An incremental parser's primary accuracy condition is that it must yield the same result as parsing from scratch. An edit affects a memo table entry if, after editing, the entry must be updated or destroyed to maintain validity.

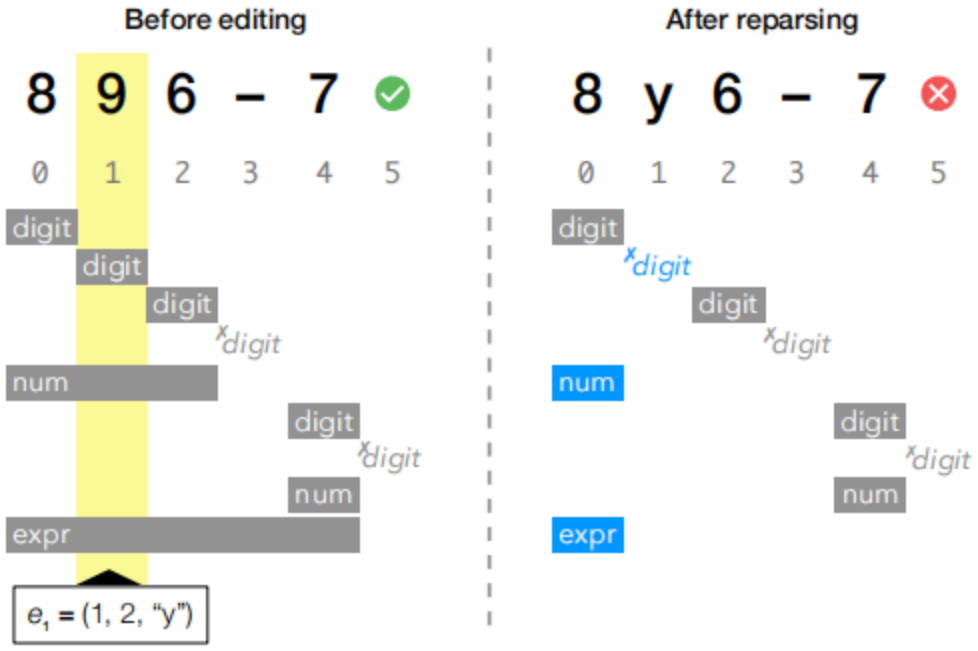

Figure 2 : Before and after using edit operation e1, left, take note of the contents of the table. ApplyEdit will invalidate the three memo table entries that are affected by the modification using the simple overlap rule. Right, the blue items are new, indicating that more effort was done in reparsing the updated data.

We'll start with a subset of possible edit operations: replacement operations that don't change the length of the input, to describe our technique for discovering affected memo table entries. The next step is to discover which (if any) of the entries are affected by the change after it has been applied to the input string.

Consider the input "896–7" from Section 1 and the edit operation "e1 = (1, 2, "y"), which swaps the "9" with a "y". The new input string after the procedure is 8y6–7", as illustrated in Figure 2.

The digit rule had previously succeeded at position 1, but it will now fail in a from-scratch parsing of the updated input (since y" is not a digit). We can remove the item for digit at position 1 from the memo table to ensure that the parser will attempt digit at that point again.

3.1.1 Basic “Overlap Rule”

In general, if the extent of a memo table entry coincides with the scope of an edit operation e, the entry is likely to be affected by e. It may not be affected depending on the modification (for example, if the replacement text is the same as the original text), but we take a conservative approach and consider any memo table entry that overlaps the edit to be invalid and removed from the memo table.

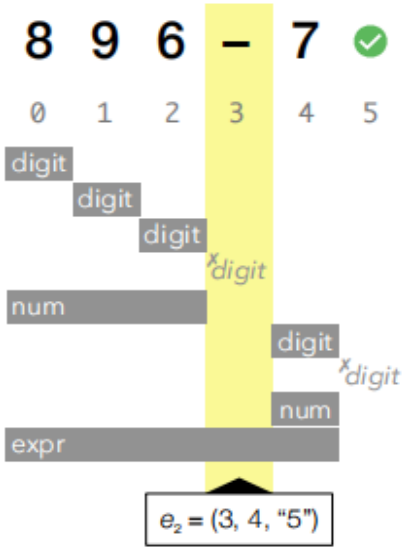

Figure 3: Memo table contents before applying edit operation e2. The simple ”overlap rule” does not detect that ‘num‘ at position 0 is affected by the change.

We can see that the entries for expr and num at position 0 must also be eliminated by applying this "overlap rule" to the full memo table (Figure 1, left). There are two new entries at position 0 (num and expr) and a new entry for digit at position 1 after re-parsing (Figure 1, right). At point 0, num still succeeds, but it now consumes only one character (8"). Although expr succeeds and consumes the "8," the parse as a whole fails since expr did not absorb the complete input.

3.1.2 Problem: Basic Overlap is not Enough

Consider the edit operation e2 = (3, 4, "5"), illustrated in Figure 3, which replaces the -" with a "5" in the original input. Only two entries would be disqualified using the same overlap rule as

before: digit at position 3 and expr at position 0. This isn't enough: an application of num at position 0 should now take the whole input, but the memo table entry only consumes 896," resulting in an inaccurate incremental parsing.

This example shows that the basic overlap rule isn't enough to catch all erroneous memo table entries. Although the character "6" at position 3 was not consumed by the application of num at position 0, the greedy repetition expression digit+, which is responsible for the application of digit at position 3, does. The num rule must check the next character via digit, even if it doesn't consume 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 łabcdef!”, this rule will examine the entire input for its first alternative, then backtrack and succeed on the second alternative, consuming nothing.

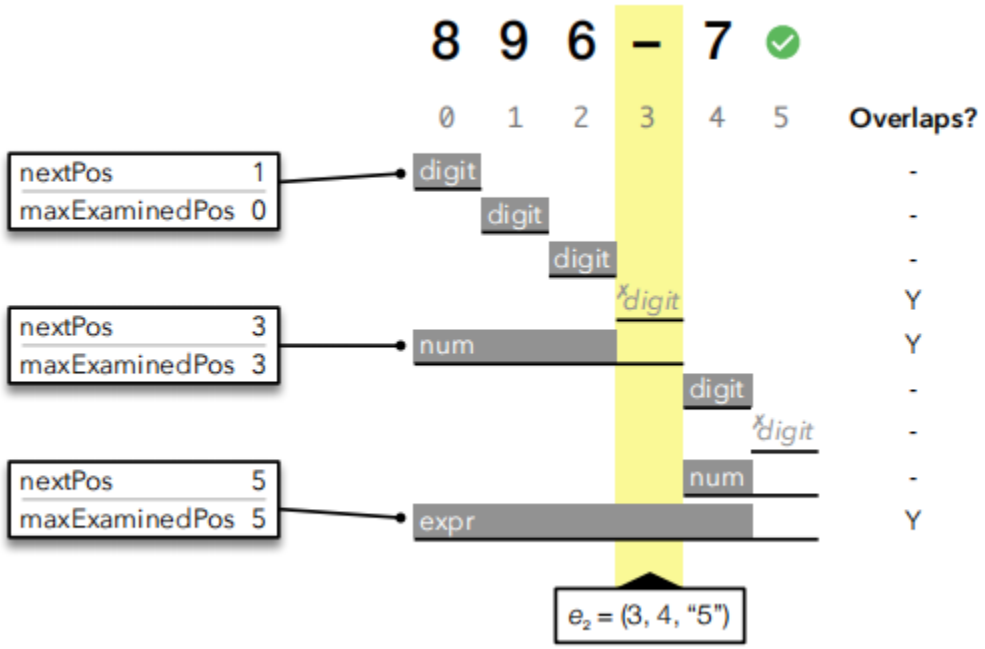

Figure 4: Memo table contents after parsing ł896-7”. The entry for num at position 0 is invalidated by e2, because its examined interval overlaps with the edit, though its consumed interval does not.

3.1.3 Solution: Maximum Examined Position

To remedy this, we add a new column to memo table entries called maximum examined position (maxExaminedPos), which records the input stream's furthest location examined throughout the parsing process. When either (a) the character at that position is eaten, or (b) the character's value is used to make a parsing choice, an input location is evaluated.

In practice, the examined interval of a rule application (specified as the closed interval [p, maxExaminedP os]) contains all of the characters that may have influenced the rule's parsing outcome. The studied interval of an expression, by definition, includes the analyzed intervals of all its subexpressions.

Figure 4 depicts a memo table with entries indicating both the next and maximum examined positions. It's worth noting that the num at position 0 examined interval overlaps with the edit operation e2. As a result, by invalidating items based on their reviewed interval rather than their consumed interval, we may maintain accuracy.

This makes intuitive sense: the studied interval of a rule application represents the complete fraction of the input that could have influenced the outcome. If and only if the edit action impacts that piece of the input, the memo table entry should be invalidated.

3.2 Relocating Memo Tables Entries

The invalidation mentioned above is sufficient as long as the length of the input does not vary. However, the problem arises when the length of the input changes, requiring the addition or deletion of elements as needed. As a result, the relocation of memo table entries exists.

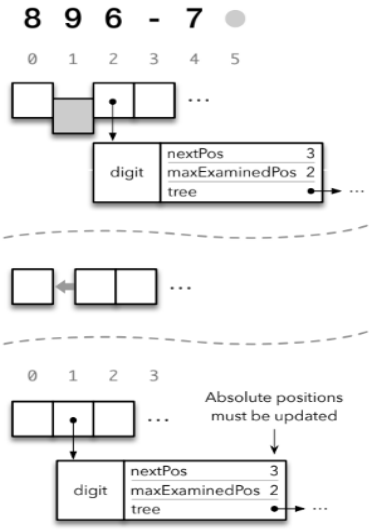

Figure 5 : Deleting the character at position 1 means deleting the corresponding column from the memo table. The entry for digit at position 3 is now at position 2, requiring nextPos and maxExaminedPos to be updated accordingly.

For example, consider the edit e3 = (1, 2, '"), which replaces the 9" in the original input with 86–7". The characters past position 1 (6–7) are pushed left by one position when the "9" is removed. After changing the input, we need to adjust the memo table, which implies eliminating column 1 and shifting the other columns left by one position, as illustrated in Figure 3.2.

The entry for digit that was previously at position 2 is now at position 1. Since *nextPos* and *maxExaminedPos* are absolute positions, they must be adjusted whenever a memo table entry is relocated.

The cost of such updates would eliminate most of the benefits of incremental parsing in a real-world application, such as putting characters at the beginning of a huge file. Our answer to this challenge is to make individual memo table entries position-independent, allowing them to be moved without incurring additional costs. To accomplish so, we use *matchLength* to replace the *nextPos* property of memo table entries with a relative offset. Similarly, an entry's examined interval is saved as *examinedLength*. This is a simple adjustment in most packrat parsing implementations, and it has no effect on parsing asymptotic complexity.

3.3 An Algorithm for A PPLYEDIT

This algorithm describes how the *examined interval to detect memo table entries* and how we *store memo table entries in a position-independent manner* technique can be used to implement together in the APPLYEDIT procedure.

APPLYEDIT : *(s, M, e) → (s', M')*

Where,

s = input string.

M = memo table.

e = edit operation.

s' = modified input string.

M' = new memo table.

The 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 within the edit interval, and if necessary, add or delete columns, as well as move any columns to the right of the interval.

Step 3: Scanning the memo table for any leftover entries whose examined interval overlaps the edit interval should be removed. Return s ′ and M′ at the end.

These three steps are the foundation of our incremental packrat parsing technique and a complete implementation of ApplyEdit.

3.4 Analysis and Optimization

The above implementation of ApplyEdit has a complexity of *O(m x n)*.

Where;

n = size of the input.

m = number of rules in the grammar.

Assuming that string concatenation (step 1) and array concatenation (step 2) are both *O(n)*, the memo table invalidation in step 3 consumes the majority of the running time. In the worst-case scenario, invalidation necessitates accessing *O(n)* columns in the memo table and scanning *O(m)*

memo table entries for overlaps with the edit interval in each column. While this has no effect on packrat parsing's asymptotic complexity (which is already *O(m x n)*), it can lead to poor performance in real-world applications.

3.4.1 Maximum Examined Length

We can improve the performance of the invalidation algorithm by keeping track of the largest examined interval of all its entries. We store this value in a per-column property called *maxExaminedLength.* This way we need not have to scan all the entries as *maxExaminedLength* shows none of its entries overlap with the edit.

During the parsing, the *maxExaminedLength* can only grow because the memo tables entries will never delete. Therefore the *maxExaminedLength* can be updated in a constant manner whenever memo table entry is added to the column.

4. Evaluation

To evaluate the performance, we implement 2 different packrat parsers.
The incremental packrat parsing algorithm was originally developed for Ohm [20, 21],

1. 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”.
2. The second parser (łEs5-Incremental”) is an incremental parser.

4.1 Parsing Performance

To evaluate the performance of the parsers in a representative setting, the benchmark was created 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.

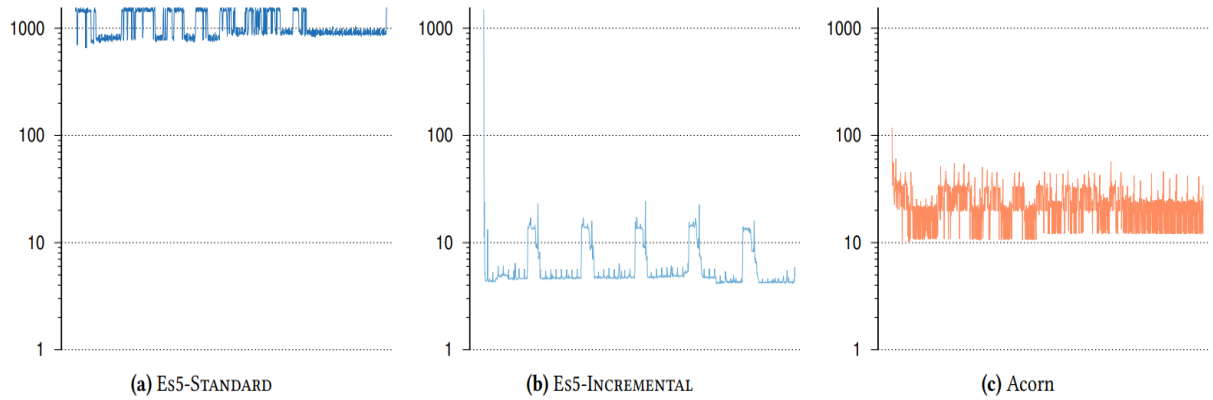

Figure 6: graphs the response times for each parser over the course of the editing session.

On the initial parse, Es5-Standard (Fig. 7a) and Es5- Incremental (Fig. 7a) are significantly slower than Acorn.On subsequent parses, the incremental parser is consistently faster than the non-incremental one — reparsing the modified source roughly two orders of magnitude faster. On average, it also outperforms Acorn.

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.

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.

However, the 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.

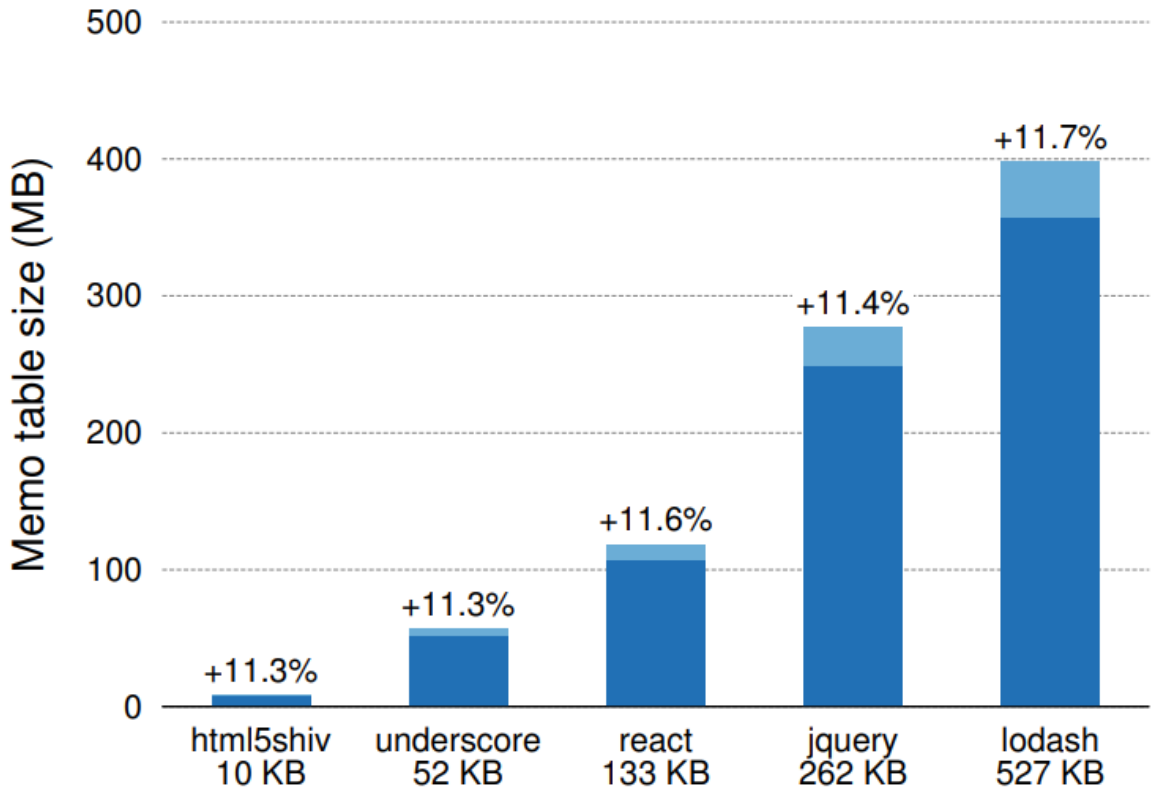

Figure 7: Memory usage for the memo table in Es5- Standard (dark blue) and Es5-Incremental (light blue) after parsing several popular JavaScript libraries

Figure 7 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.

4.3 Discussion

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.

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.

5. Related Work

5.1 Incremental Parsing

Ghezzi and Mandrioli were the first to introduce the concept of incremental parsing in their research report Incremental parsing which was first published in 1979. In this paper the incremental parser presented extends the conventional LR parsing algorithm and its performance is compared with that of a conventional parser.

Much research followed after that such as Optimal incremental parsing by J.-M. Larchevêque and Efficient and flexible incremental parsing by Graham and Wagner which focused on improving the performance of incremental shift-reduce parsers, and on attaining optimal node reuse.

Techniques for incremental top-down parsing have mostly focused on LL(1) grammars, which is a more restricted class of languages than the class of languages supported by packrat parsing. These solutions have primarily emphasized single-site editing that is tightly integrated with an editor. Their algorithm supports any number of edit sites (via repeated calls to ApplyEdit between calls to Parse) and does not require any special editor integration other than the ability to detect and respond to the user's edit operations.

Papa Carlo is a Scala parsing library that may be used to create incremental parsers and is based on Parsing Expression Grammars (PEGs). However, it is not a real packrat parser because it only uses memoization in a limited way and hence does not guarantee linear parse times.In Papa Carlo the author of a grammar must manually define the syntax of its code fragments. In contrast, the approach used in this paper 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, the technique makes all partial parsing results available for reuse, not just the results of selected rules.

5.2 Optimization of Packrat Parsers

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. While many of these optimizations improve batch performance, their reduced use of memoization might have a negative impact on incremental response time. Any memoized result in incremental

packrat parsing could be useful in the future, therefore it's important to avoid requiring "too much" work in response to each edit operation.

6. Conclusions and Future Work

In this paper they have presented an algorithm for incremental packrat parsing which is probably the first such algorithm according to them. They have described two key optimizations which are 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.

For the experimental evaluation, they have compared the performance between an incremental parser based on their algorithm and standard packrat parser. Their technique provides large performance gains at the cost of 12% more memory usage and slightly worse batch parsing performancing.

In the future they want to search for ways to combine optimization with their algorithm in order to balance memory usage and batch parsing performance with low incremental response times. They also want to combine their algorithm with incremental semantic analysis.

They have successfully implemented their techniques in Ohm, a popular PEG-based parsing toolkit for JavaScript.

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] Dubroy, P., & Warth, A. (2017, October). Incremental packrat parsing. In *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering* (pp. 14-25).