Incremental Parsing of Common Lisp Code

Irène Durand
Robert Strandh
irene.durand@u-bordeaux.fr
robert.strandh@u-bordeaux.fr
LaBRI, University of Bordeaux
Talence, France

ACM Reference Format:
Irène Durand and Robert Strandh. 2018. Incremental Parsing of Common Lisp Code. In *Proceedings of the 11th European Lisp Symposium (ELS'18).* ACM, New York, NY, USA, 7 pages.

1 INTRODUCTION

Whether autonomous or part of an integrated development environment, an editor that caters to Common Lisp programmers must analyze the buffer contents in order to help the programmer understand how this contents would be analyzed when submitted to the Common Lisp compiler or interpreter. Furthermore, the editor analysis must be *fast* so that it is up to date shortly after each keystroke generated by the programmer. Miller [4] indicates that an upper bound on the delay between a keystroke and the updated result is around 0.1 seconds.

In order to obtain such speed for the analysis, it must be *incremental*. A complete analysis of the entire buffer for each keystroke is generally not feasible, especially for buffers with a significant amount of code.

Furthermore, the analysis is necessarily *approximate*. The reader macro `#.` (hash dot) and the macro `eval-when` allow for arbitrary computations at read time and at compile time, and these computations may influence the environment in arbitrary ways that may invalidate subsequent, or even preceding analyses, making an analysis that is both precise and incremental impossible in general.

The question, then, is how approximate the analysis has to be, and how much of it we can allow ourselves to recompute, given the performance of modern hardware and modern Common Lisp implementations.

In this paper, we describe an analysis technique that represents an improvement compared with the ones used by the most widespread editors for Common Lisp code used today. The technique is more precise than existing ones, because it uses the Common Lisp `read` function, which is a better approximation than the regular-expression techniques most frequently used. We show that our analysis is sufficiently fast because it is done incrementally, and it requires very little incremental work for most simple editing operations.

ELS'18, April 16–17 2018, Marbella, Spain

ACM ISBN 978-2-9557474-2-1.

The work in this paper is specific to the Common Lisp language. This language has a number of specific features in terms of its syntax, some of which make it harder to write a parser for it, and some of which make it easier:

- The reader algorithm is defined in terms of a non-tokenizing recursive descent parser. This fact makes our task easier, because the result of calling `read` at any location in the source code is well defined and yields a unique result. For other languages, the meaning of some sequence of characters may depend on what comes much later.
- The Common Lisp reader is *programmable* in that the application programmer can define *reader macros* that invoke arbitrary code. This feature makes our task harder, because it makes it impossible to establish fixed rules for the meaning of a sequence of characters in the buffer. The technique described in this paper can handle such arbitrary syntax extensions.
- As previously mentioned, arbitrary Common Lisp code may be invoked as part of a call to `read`, and that code may modify the readtable and/or the global environment. This possibility makes our task harder, and we are only able to address some of the problems it creates.

2 PREVIOUS WORK

In this section, we present a selection of existing editors, and in particular, we discuss the technique that each selected editor uses in order to analyze a text buffer containing Common Lisp code.

We do not cover languages other than Common Lisp, simply because our technique crucially depends on ability to analyse the buffer contents with a non-tokenizing (i.e, based on reading a character at a time) recursive descent parser. The Common Lisp `read` function is defined this way, but most other languages require more sophisticated parsing techniques for a correct analysis.

2.1 Emacs

GNU Emacs [1, 2] is a general-purpose text editor written partly in C but mostly in a special-purpose dialect of Lisp [3].

In the editing mode used for writing Common Lisp source code, highlighting is based on string matching, and no attempt is made to determine the symbols that are present in the current package. Even when the current package does not use the `common-lisp` package, strings that match Common Lisp symbols are highlighted nevertheless.

In addition, no attempt is made to distinguish between the role of different occurrences of a symbol. In Common Lisp where a symbol can simultaneously be used to name a function, a variable, etc., it

would be desirable to present occurrences with different roles in a different way.

Indentation is also based on string matching, resulting in the text being indented as Common Lisp code even when it is not. Furthermore, indentation does not take into account the role of a symbol in the code. So, for example, if a lexical variable named (say) `prog1` is introduced in a `let` binding and it is followed by a newline, the following line is indented as if the symbol `prog1` were the name of a Common Lisp function as opposed to the name of a lexical variable.

2.2 Climacs

Climacs[1] is an Emacs-like editor written entirely in Common Lisp. It uses McCLIM [8] for its user interface, and specifically, an additional library called ESA [9].

The framework for syntax analysis in Climacs [5] is very general. The parser for Common Lisp syntax is based on table-driven parsing techniques such as LALR parsing, except that the parsing table was derived manually. Like Emacs, Climacs does not take the current package into account. The parser is incremental in that the state of the parser is saved for certain points in the buffer, so that parsing can be restarted from such a point without requiring the entire buffer to be parsed from the beginning.

Unlike Emacs, the Climacs parser is more accurate when it comes to the role of symbols in the program code. In many cases, it is able to distinguish between a symbol used as the name of a function and the same symbol used as a lexical variable.

2.3 Lem

Lem[2] is a relatively recent addition to the family of Emacs clones. It is written in Common Lisp and it uses `curses` to display buffer contents.

Like Emacs (See Section 2.1.), it uses regular expressions for analyzing Common Lisp code, with the same disadvantages in terms of precision of the analysis.

3 OUR TECHNIQUE

3.1 Buffer update protocol

Our incremental parser parses the contents of a buffer, specified in the form of a CLOS protocol[7]. We include a brief summary of that protocol here.

The protocol contains two sub-protocols:

(1) The *edit protocol* is used whenever items are inserted or deleted from the buffer. An edit operation is typically invoked as a result of a keystroke, but an arbitrary number of edit operations can happen as a result of a keystroke, for example when a region is inserted or deleted, or when a keyboard macro is executed.
(2) The *update protocol* is used when the result of one or more edit operations must be displayed to the user. This protocol is typically invoked for each keystroke, but it can be invoked less frequently if some view of the buffer is temporarily hidden. Only when the view becomes visible is the update protocol invoked.

This organization solves several problems with the design of similar protocols:

- The edit protocol does not trigger any updates of the views. The edit operations simply modify the buffer contents, and marks modified lines with a *time stamp*. Therefore the operations of the edit protocol are fast. As a result, complex operations such as inserting or deleting a region, or executing a complicated keyboard macro, can be implemented as the repeated invocation of simpler operations in the edit protocol. No special treatment is required for such complex operations, which simplifies their overall design.
- There is no need for the equivalent of *observers* as many object-oriented design methods require. Visible views are automatically updated after every keystroke. Each view contains a time stamp corresponding to the previous time it was updated, and this time stamp is transmitted to the buffer when the update protocol is invoked.
- Views that are invisible are not updated as a result of a keystroke. Such views are updated if and when they again become visible.

When a view invokes the update protocol, in addition to transmitting its time stamp to the buffer, it also transmits four *callback functions*. Conceptually, the view contains some mirror representation of the lines of the buffer. Before the update protocol is invoked, the view sets an index into that representation to zero, meaning the first line. As a result of invoking the update protocol, the buffer informs the view of changes that happened after the time indicated by the time stamp by calling these callback functions as follows:

- The callback function *skip* indicates to the view that the index should be incremented by a number given as argument to the function.
- The callback function *modify* indicates a line that has been modified since the last update. The line is passed as an argument. The view must delete lines at the current index until the correct line is the one at the index. It must then take appropriate action to reflect the modification, typically by copying the new line contents into its own mirror data structure.
- The callback function *insert* indicates that a line has been inserted at the current index since the last update. Again, the line is passed as an argument. The view updates its mirror data structure to reflect the new line.
- The callback function *sync* is called with a line passed as an argument. The view must delete lines at the current index until the correct line is the one at the index.

Notice that there is no *delete* callback function. The buffer does not hold on to lines that have been deleted, so it is incapable of supplying this information. Instead, the *modify* and *sync* operations provide this information implicitly by supplying the next line to be operated on. Any lines preceding it in the mirror data structure are no longer present in the buffer and should be deleted by the view.

The buffer protocol is *line-oriented* in two different ways:

[1] See: https://common-lisp.net/project/climacs/
[2] See https://github.com/cxxxr/lem.

(1) The editing operations specified by the protocol define a *line* abstraction, in contrast to a buffer of GNU Emacs [1] which exposes a single sequence containing newline characters to indicate line separation.
(2) The update protocol works on the granularity of a line. An entire line can be reported as being modified or inserted.

In the implementation of the buffer protocol, a line being edited is represented as a gap buffer. Therefore, editing operations are very fast, even for very long lines. However, the update protocol works on the granularity of an entire line. This granularity is acceptable for Common Lisp code, because lines are typically short. For other languages it might be necessary to use a different buffer library.

For the purpose of this paper, we are only interested in the update protocol, because we re-parse the buffer as a result of the update protocol having been invoked. We can think of such an invocation as resulting in a succession of operations, sorted by lines in increasing order. There can be three different update operations:

- Modify. The line has been modified.
- Insert. A new line has been inserted.
- Delete. An existing line has been deleted.

Although the presence of a *delete* operation may seem to contradict the fact that no such operation is possible, it is fairly trivial to derive this operation from the ones that are actually supported by the update protocol. Furthermore, this derived set of operations simplifies the presentation of our technique in the rest of the paper.

In order to parse the buffer contents, we use a custom `read` function. This version of the `read` function differs from the standard one in the following ways:

- Instead of returning S-expressions, it returns a nested structure of instances of a standard class named `parse-result`. These instances contain the corresponding S-expression and the start and end position (line, column) in the buffer of the parse result.
- The parse results returned by the reader also include entities that would normally not be returned by `read` such as comments and, more generally, results of applying reader macros that return no values.
- Instead of attempting to call `intern` in order to turn a token into a symbol, the custom reader returns an instance of a standard class named `token`.

The reader from the SICL project[3] was slightly modified to allow this kind of customization, thereby avoiding the necessity of maintaining the code for a completely separate reader.

No changes to the mechanism for handling reader macros is necessary. Therefore, we handle custom reader macros as well. Whenever a reader macro calls `read` recursively, a nested parse result is created in the same way as with the standard reader macros. More information about the required modifications to the reader are provided in Appendix B.

For a visible view, the buffer update protocol is invoked after each keystroke generated by the end user, and the number of modifications to the buffer since the previous invocation is typically very modest, in that usually a single line has been modified. It would be wasteful, and too slow for large buffers, to re-parse the entire buffer character by character, each time the update protocol is invoked. For that reason, we keep a *cache* of parse results returned by the customized reader.

[3]See: https://github.com/robert-strandh/SICL.

3.2 Cache organization

The cache is organized as a sequence[4] of top-level parse results. Each top-level parse result contains the parse results returned by nested calls to the reader. Here, we are not concerned with the details of the representation of the cache. Such details are crucial in order to obtain acceptable performance, but they are unimportant for understanding the general technique of incremental parsing. Refer to appendix A for an in-depth description of these details.

When the buffer is updated, we try to maintain as many parse results as possible in the cache. Updating the cache according to a particular succession of update operations consists of two distinct phases:

(1) Invalidation of parse results that span a line that has been modified, inserted, or deleted.
(2) Rehabilitation of the cache according to the updated buffer contents.

3.3 Invalidation phase

As mentioned in Section 3.1, the invocation of the buffer-update protocol results in a sequence of operations that describe how the buffer has changed from the previous invocation.

As far as the invalidation phase is concerned, there are only minor variations in how the different types of operations are handled. In all cases (line modification, line insertion, line deletion), the existing parse results that straddle a place that has been altered must be invalidated. Notice that when a top-level parse result straddles such a modification, that parse result is invalidated, but it is very likely that several of its children do not straddle the point of modification. Therefore such children are not invalidated, and are kept in the cache in case they are needed during the rehabilitation phase.

In addition to the parse results being invalidated as described in the previous paragraph, when the operation represents the insertion or the deletion of a line, remaining valid parse results following the point of the operation must be modified to reflect the fact that they now have a new start-line position.

As described in Appendix A, we designed the data structure carefully so that both invalidating parse results as a result of these operations, and modifying the start-line position of remaining valid parse results can be done at very little cost.

3.4 Rehabilitation phase

Conceptually, the rehabilitation phase consists of parsing the entire buffer from the beginning by calling `read` until the end of the buffer is reached. However, three crucial design elements avoid the necessity of a full re-analysis:

- Top level parse results that precede the first modification to the buffer do not have to be re-analyzed, because they must return the same result as before any modification.

[4]Here, we use the word *sequence* in the meaning of a set of items organized consecutively, and not in the more restrictive meaning defined by the Common Lisp standard.

- When `read` is called at a buffer position corresponding to a parse result that is in the cache, we can simply return the cache entry rather than re-analyzing the buffer contents at that point.
- If a top-level call to `read` is made beyond the last modification to the buffer, and there is a top-level parse result in the cache at that point, then every remaining top-level parse result in the cache can be re-used without any further analysis required.

4 PERFORMANCE OF OUR TECHNIQUE

The performance of our technique can not be stated as a single figure, nor even as a function of the size of the buffer, simply because it depends on several factors such as the exact structure of the buffer contents and the way the user interacts with that contents.

Despite these difficulties, we can give some indications for certain important special cases. We ran these tests on a 4-core Intel Core processor clocked at 3.3GHz, running SBCL version 1.3.11.

4.1 Parsing with an empty cache

When a buffer is first created, the cache is empty. The buffer contents must then be read, character by character, and the cache must be created from the contents.

We timed this situation with a buffer containing 10000 lines of representative Common Lisp code. The total time to parse was around 1.5 seconds. This result deserves some clarifications:

- It is very unusual to have a file of Common Lisp code with this many lines. Most files contain less than 2000 lines, which is only 1/5 of the one in our test case.
- This result was obtained from a very preliminary version of our parser. In particular, to read a character, several generic functions where called, including the `stream-read-char` function of the Gray streams library, and then several others in order to access the character in the buffer. Further optimizations are likely to decrease the time spent to read a single character.
- This situation will happen only when a buffer is initially read into the editor. Even very significant subsequent changes to the contents will still preserve large portions of the cache, so that the number of characters actually read will only be a tiny fraction of the total number of characters in the buffer.
- Parsing an entire buffer does not exercise the incremental aspect of our parser. Instead, the execution time is a complex function of the exact structure of the code, the performance of the reader in various situations, the algorithm for generic-function dispatch of the implementation, the cost of allocating standard objects, etc. For all these reasons, a more thorough analysis of this case is outside the scope of this paper, and the timing is given only to give the reader a rough idea of the performance in this initial situation.
- This particular case can be handled by having the parser process the original stream from which the buffer contents was created, rather than giving it the buffer protocol wrapped in a stream protocol after the buffer has been filled. That way, the entire overhead of the Gray-stream protocol is avoided altogether.

4.2 Parsing after small modifications

We measured the time to update the cache of a buffer with 1200 lines of Common Lisp code. We used several variations on the number of top-level forms and the size of each top-level form. Three types of representative modifications were used, namely inserting/deleting a constituent character, inserting/deleting left parenthesis, and inserting/deleting a double quote. All modifications were made at the very beginning of the file, which is the worst-case scenario for our technique.

For inserting and deleting a constituent character, we obtained the results shown in Table 1. For this benchmark, the performance is independent of the distribution of forms and sub-forms, and also of the number of characters in a line. The execution time is roughly proportional to the number of lines in the buffer. For that reason, the form size is given only in number of lines. The table shows that the parser is indeed very fast for this kind of incremental modification to the buffer.

nb forms	form size	time
120	10	0.14ms
80	15	0.14ms
60	20	0.14ms
24	100	0.23ms
36	100	0.32ms

Table 1: Inserting and deleting a constituent character.

For inserting and deleting a left parenthesis, we obtained the results shown in Table 2. For this benchmark, the performance is independent of the size of the sub-forms of the top-level forms. For that reason, the form size is given only in number of lines. As shown in the table, the performance is worse for many small top-level forms, and then the execution time is roughly proportional to the number of forms. When the number of top-level forms is small, the execution time decreases asymptotically to around 0.5ms. However, even the slowest case is very fast and has no impact on the perceived overall performance of the editor.

nb forms	form size	time
120	10	1.3ms
80	15	1.0ms
60	20	0.5ms
40	30	0.7ms
30	40	0.6ms
24	50	0.5ms
12	100	0.5ms

Table 2: Inserting and deleting a left parenthesis.

Finally, for inserting and deleting a double quote, we obtained the results shown in Table 3. For this benchmark, the performance is roughly proportional to the number of characters in the buffer when the double quote is inserted, and completely dominated by the execution time of the reader when the double quote is deleted. The execution time thus depends not only on the number of characters in the buffer, but also on how those characters determine what

the reader does. As shown by the table, these execution times are borderline acceptable. In the next section, we discuss possible ways of improving the performance for this case.

nb forms	form size	characters per line	time
120	10	1	18ms
80	15	1	15ms
60	20	1	17ms
24	100	1	33ms
36	100	1	50ms
120	10	30	70ms

Table 3: Inserting and deleting a double quote.

5 CONCLUSIONS AND FUTURE WORK

Currently, parse results that are not part of the final structure of the buffer are discarded. When the user is not using an editor mode that automatically balances characters such as parentheses and double quotes, inserting such a character often results in a large number of parse results being discarded, only to have to be created again soon afterward, when the user inserts the balancing character of the pair. We can avoid this situation by keeping parse results that are not part of the final structure in the cache, in the hopes that they will again be required later. We then also need a strategy for removing such parse results from the cache after some time, so as to avoid that the cache grows without limits.

Parsing Common Lisp source code is only the first step in the analysis of its structure. In order to determine the role of each symbol and other information such as indentation, further analysis is required. Such analysis requires a *code walker*, because the role of a symbol may depend on the definitions of macros to which it is an argument. Similarly, computing standard *indentation*, also requires further analysis. To implement this code walker, we consider using the first phase of the Cleavir compiler framework.[5]

We plan to investigate the use of a new implementation of *first-class global environments* [6]. This new implementation of the existing CLOS protocol would use *incremental differences* to the *startup environment*[6] so as to define a *compilation environment*[7] that is different for each top-level form in the editor buffer. This technique would allow us to restart the compiler in an appropriate environment without having to process the entire buffer from the beginning.

The combination of the use of the first pass of the Cleavir compiler framework and the use of incremental first-class global environments will allow us to handle compile-time evaluation of certain top-level forms in a way that corresponds to the semantics of the file compiler. In particular, imperative environment operations such as changing the current package or modifying the readtable in the middle of a buffer will have the expected consequences, but only to subsequent forms in the buffer.

[5]Cleavir is currently part of SICL. See the directory named `Code/Cleavir` in the SICL repository.

[6]Recall that the startup environment is the environment in which the compiler was invoked.

[7]Recall that the compilation environment is the environment used by the compiler for definitions and side effects of the compilation process.

A more precise analysis of Common Lisp code opens the possibility for additional functionality that requires knowledge about the role of each expression. In particular, such an analysis could be the basis for sophisticated code transformations such as variable renaming and code refactoring.

6 ACKNOWLEDGMENTS

We would like to thank Philipp Marek and Cyrus Harmon for providing valuable feedback on early versions of this paper.

REFERENCES

[1] Craig A. Finseth. Theory and practice of text editors, or, A cookbook for an Emacs. Thesis (B.S.), M.I.T., Department of Electrical Engineering and Computer Science, Cambridge, MA, USA, 1980. Supervised by David P. Reed.

[2] Craig A. Finseth. *The Craft of Text Editing – Emacs for the Modern World.* Springer-Verlag, 1991. ISBN 0-387-97616-7 (New York), 3-540-97616-7 (Berlin).

[3] Bill Lewis, Dan LaLiberte, and Richard Stallman. *GNU Emacs Lisp Reference Manual.* Free Software Foundation, Boston, MA, USA, 2014. ISBN 1-882114-74-4.

[4] Robert B. Miller. Response time in man-computer conversational transactions. In *Proceedings of the December 9-11, 1968, Fall Joint Computer Conference, Part I*, AFIPS '68 (Fall, part I), pages 267–277, New York, NY, USA, 1968. ACM. doi: 10.1145/1476589.1476628. URL http://doi.acm.org/10.1145/1476589.1476628.

[5] Christophe Rhodes, Robert Strandh, and Brian Mastenbrook. Syntax Analysis in the Climacs Text Editor. In *Proceedings of the International Lisp Conference*, ILC 2005, June 2005.

[6] Robert Strandh. First-class Global Environments in Common Lisp. In *Proceedings of the 8th European Lisp Symposium*, ELS 2015, pages 79 – 86, April 2015. URL http://www.european-lisp-symposium.org/editions/2015/ELS2015.pdf.

[7] Robert Strandh. A CLOS Protocol for Editor Buffers. In *Proceedings of the 9th European Lisp Symposium*, ELS 2016, pages 3:21–3:28. European Lisp Scientific Activities Association, 2016. ISBN 978-2-9557474-0-7. URL http://dl.acm.org/citation.cfm?id=3005729.3005732.

[8] Robert Strandh and Timothy Moore. A Free Implementation of CLIM. In *Proceedings of the International Lisp Conference*, ILC 2002, October 2002.

[9] Robert Strandh, David Murray, Troels Henriksen, and Christophe Rhodes. ESA: A CLIM Library for Writing Emacs-Style Applications. In *Proceedings of the 2007 International Lisp Conference*, ILC '07, pages 24:1–24:10, New York, NY, USA, 2009. ACM. ISBN 978-1-59593-618-9. doi: 10.1145/1622123.1622150. URL http://doi.acm.org/10.1145/1622123.1622150.

A CACHE REPRESENTATION

Figure 1 illustrates the representation of the cache for parse results. The buffer contents that corresponds to that cache contents might for instance be:

```
(a
 (b c))
(d)
(e
 f)
(g
 (h))
```

The sequence of top-level parse results is split into a *prefix* and a *suffix*, typically reflecting the current position in the buffer being edited by the end user. The suffix contains parse results in the order they appear in the buffer, whereas the prefix contains parse results in reverse order, making it easy to move parse results between the prefix and the suffix.

Depending on the location of the parse result in the cache data structure, its position may be *absolute* or *relative*. The prefix contains parse results that precede updates to the buffer. For that reason, these parse results have absolute positions. Parse results in the suffix, on the other hand, follow updates to the buffer. In particular, if a line is inserted or deleted, the parse results in the suffix will

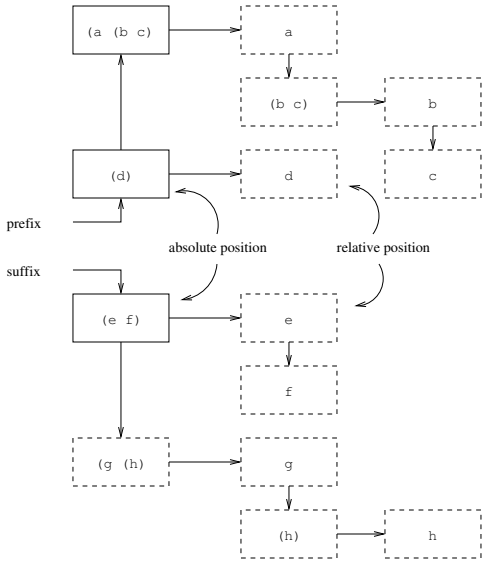

Figure 1: Representation of the cache.

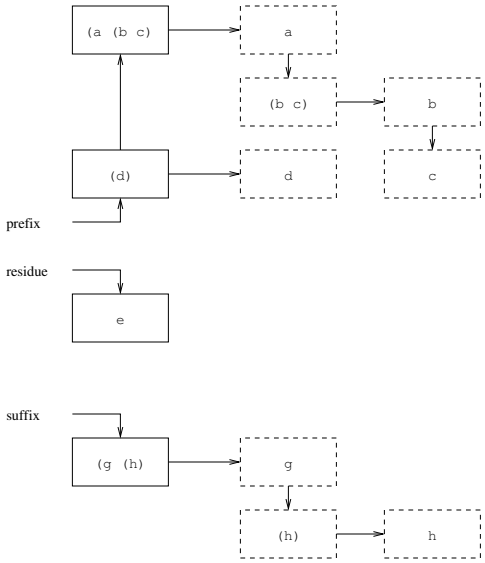

Figure 2: Cache contents after invalidation.

have their positions changed. For that reason, only the first parse result of the suffix has an absolute position. Each of the others has a position relative to its predecessor. When a line is inserted or deleted, only the first parse result of the suffix has to have its position updated. When a parse result is moved from the prefix to the suffix, or from the suffix to the prefix, the positions concerned are updated to maintain this invariant.

To avoid having to traverse all the descendants of a parse result when its position changes, we make the position of the first child of some parse result P relative to that of P, and the children, other than the first, of some parse result P, have positions relative to the previous child in the list.

As a result of executing the invalidation phase, a third sequence of parse results is created. This sequence is called the *residue*, and it contains valid parse results that were previously children of some top-level parse result that is no longer valid. So, for example, if the line containing the symbol `f` in the buffer corresponding to the cache in Figure 1 were to be modified, the result of the invalidation phase would be the cache shown in Figure 2.

As Figure 2 shows, the top-level parse result corresponding to the expression `(e f)` has been invalidated, in addition the child parse result corresponding to the expression `f`. However, the child parse result corresponding to the expression `e` is still valid, so it is now in the residue sequence. Furthermore, the suffix sequence now contains only the parse result corresponding to the expression `(g (h))`.

For the rehabilitation phase, we can imagine that a single character was inserted after the `f`, so that the line now reads as `fi)`.

At the start of the rehabilitation phase, the position for reading is set to the end of the last valid top-level parse result in the prefix, in this case at the end of the line containing the expression `(d)`. When the reader is called, it skips whitespace characters until it is positioned on the left parenthesis of the line containing `(e`. There is no cache entry, neither in the residue nor in the suffix, corresponding to this position, so normal reader operation is executed. Thus, the reader macro associated with the left parenthesis is invoked, and the reader is called recursively on the elements of the list. When the reader is called with the position corresponding to the expression `e`, we find that there is an entry for that position in the residue, so instead of this expression being read by normal reader operation, the contents of the cache is used instead. As a result, the position in the buffer is set to the end of the cached parse result, i.e. at the end of the expression `e`. The remaining top-level expression is read using then normal reader operation resulting in the expression `(e fi)`. This parse result is added to the prefix resulting in the cache contents shown in figure 3.

The reader is then invoked again in order to read another top-level expression. In this invocation, whitespace characters are first skipped until the reader is positioned immediately before the expression `(g (h))`. Not only is there a parse result in the cache corresponding to this position, but that parse result is the first in the suffix sequence. We therefore know that all parse results on in the suffix are still valid, so the we can terminate the rehabilitation phase.

B READER CUSTOMIZATION

In order for it to be possible for the Common Lisp `read` function to serve as a basis for the incremental parser described in this paper, it must be adapted in the ways described below.

B.1 Returning parse results

In addition to the nested expressions returned by an unmodified `read` function, it must also return a nested structure of *parse results*, i.e. expressions wrapped in standard instances that also contain information about the location in the source code of the wrapped expressions.

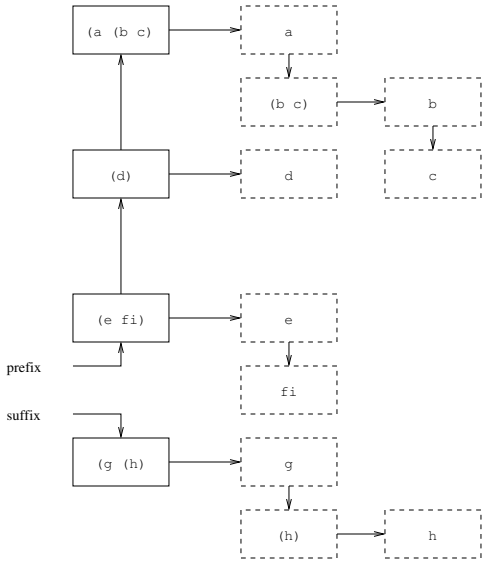

Figure 3: Cache contents after read.

To accomplish this additional functionality, it is not possible to create a custom `read` function that returns parse results *instead of* expressions, simply because the function must handle custom reader macros, and those reader macros return expressions, and not parse results. Also, it would create unnecessary maintenance work if all standard reader macros had to be modified in order to return parse results instead of expressions.

It is also not possible to modify the `read` function to return the parse result as a second value, in addition to the normal expression. One reason is that we would like for the modified `read` function to be compatible with the standard version, and it is not permitted by the Common Lisp standard to return additional values.

Instead, the modified `read` function maintains an explicit stack of parse results in parallel with the expressions that are normally returned. This explicit stack is kept as the value of a special variable that our parser accesses after a call to `read`.

B.2 Returning parse results for comments

The modified `read` function must return parse results that correspond to source code that the standard `read` function does not return, such as comments and expressions that are not selected by a read-time conditional. We solve this problem by checking when a reader macro returns no values, and in that case, a corresponding parse result is pushed onto the explicit stack mentioned in the previous section.

B.3 Intercepting symbol creation

The modified `read` function must not call `intern` in all situations that the ordinary `read` function would, and it must not signal an error when a symbol with an explicit package prefix does not exist. For that reason, the modified reader calls a generic function with the characters of a potential token instead. The unmodified `read` function just calls `intern`, whereas the custom `read` function creates a particular parse result that represents the token, and that can be exploited by the editor.