A CLOS Protocol for Editor Buffers

Robert Strandh
University of Bordeaux
351, Cours de la Libération
Talence, France
robert.strandh@u-bordeaux1.fr

ABSTRACT

Many applications and libraries contain a data structure for storing and editing text. Frequently, this data structure is chosen in a somewhat arbitrary way, without taking into account typical use cases and their consequence to performance. In this paper, we present a data structure in the form of a CLOS protocol that addresses these issues. In particular, the protocol is divided into an *edit* protocol and an *update* protocol, designed to be executed at different frequencies. The update protocol is based on the concept of *time stamps* allowing multiple *views* without any need for *observers* or similar techniques for informing the views of changes to the model (i.e., the text buffer).

In addition to the protocol definition, we also present two different implementations of the definition. The main implementation uses a splay tree of lines, where each line is represented either as an ordinary vector or as a gap buffer, depending on whether the line is being edited or not. The other implementation is very simple and supplied only for the purpose of testing the main implementation.

CCS Concepts

•Applied computing → Text editing;

Keywords

CLOS, Common Lisp, Text editor

1. INTRODUCTION

Many applications and libraries contain a data structure for storing and editing text. In a simple input editor, the content can be a single, relatively short, line of text, whereas in a complete text editor, texts with thousands of lines must be supported.

In terms of abstract data types, one can think of an editor buffer as an *editable sequence*. The problem of finding a good data structure for such a data type is made more interesting because a data structure with optimal asymptotic worst-case complexity would be considered as having too much overhead, both in terms of execution time, and in terms of memory requirements.

For a text editor with advanced features such as keyboard macros, it is crucial to distinguish between two different control loops:

- The innermost loop consists of inserting and deleting individual items[1] in the buffer, and of moving one or more *cursors* from one position to an adjacent position.
- The outer loop consists of updating the *views* into the buffer. Each view is typically an interval of less than a hundred lines of the buffer.

When the user inserts or deletes individual items, the inner loop performs a single iteration for each iteration of the outer loop, i.e., the views are updated for each elementary operation issued by the user.

When operations on multiple items are issued, such as the insertion or deletion of *regions* of text, the inner loop can be executed a large number of iterations for a single iteration of the outermost loop. While such multiple iterations could be avoided in the case of regions by providing operations on intervals of items, doing so does not solve the problem of *keyboard macros* where a large number of small editing operations can be issued for a single execution of a macro. Furthermore, to avoid large amounts of special-case code, it is preferable that operations on regions be possible to implement as repeated application of elementary editing operations.

Roughly speaking, we can say that each iteration of the outer loop is performed for each character typed by the user. Given the relatively modest typing speed of even a very fast typist, as long as an iteration can be accomplished in a few tens of milliseconds, performance will be acceptable. This is sufficient time to perform a large number of fairly sophisticated operations.

An iteration of the inner loop, on the other hand, must be several orders of magnitude faster than an iteration of the outer loop.

[1]In a typical editor buffer, the items it contains are individual characters. Since our protocols and our implementations are not restricted to characters, we refer to the objects contained in it as "items" rather than characters. An item is simply an object that occupies a single place in the editable sequence that the buffer defines.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

© 2016 Copyright held by the owner/author(s).

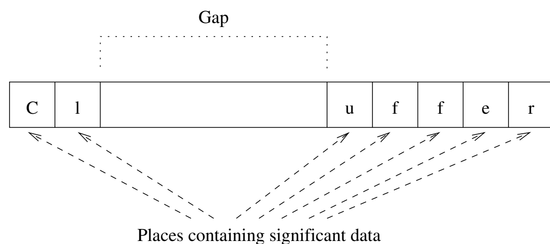

Figure 1: Gap buffer.

In this paper, we propose a data structure that has fairly low overhead, both in terms of execution time and in terms of storage requirements. More importantly, our data structure is defined as a collection of CLOS *protocols* each one aimed either at the inner or the outer control loop.

In Section 2, we provide an overview of existing representations of editor buffers, along with the characteristics of each representation. We give examples of existing editors with respect to which representation each one uses.

2. PREVIOUS WORK

2.1 Representing items in a buffer

There are two basic techniques for representing the items in an editor buffer, namely *gap buffer* and *line oriented*.

2.1.1 Gap buffer

A gap buffer can be thought of as a vector holding the items of the buffer but with some additional free space. In a typical gap-buffer implementation, a possibly empty *prefix* of the buffer content is stored at the beginning of the vector, and a possibly empty *suffix* of the content is stored at the end of the vector, leaving a possibly empty *gap* between the prefix and the suffix. This representation is illustrated in Figure 1.

Buffer items are moved from the end of the prefix to the beginning of the suffix, and vice-versa, in order to position the gap where an item is about to be inserted or deleted. The typical use case for text editing has a very high probability that two subsequent editing operations will be *close* to each other (in terms of the number of items between the two). Therefore, in most cases, few items will have to be moved, making this data structure very efficient for editing operations corresponding to this use case.

Clearly, in the worst case, all buffer items must be moved for every editing operation. This case happens when editing operations alternate between the beginning of the buffer and the end of the buffer. Even so, moving all the items even in a very large buffer does not represent a serious performance problem. Furthermore, the pathological case can be largely avoided by considering the vector holding the items as being *circular* (as Flexichain [5] does).

Perhaps the main disadvantage of representing the entire buffer as a single gap buffer is that it is difficult to associate additional information with specific points in the buffer. One might, for instance, want to associate some state of an *incremental parser* that keeps track of the buffer content in a more structured form. One possible solution to this problem is to introduce a *cursor*[2] at the points where it is desirable to attach information.

Another difficulty with the gap-buffer representation has to do with updating possibly multiple *views*. As we discussed in Section 1, views are updated at the frequency of the event loop, whereas the manipulation of *regions* of items and especially the use of *keyboard macros* may make the frequency of editing operations orders of magnitude higher.

2.1.2 Line oriented

Another common way of representing the editor buffer recognizes that text is usually divided into *lines*, where each line typically has a very moderate number of items in it.

In a line-oriented representation, we are dealing with a *two-level sequence*. At the outer level, we have a sequence of lines, and each element of that sequence is a sequence of items. Every possible combination of representations of these two sequences is possible. However, since the number of items in an individual line is usually small, most existing editors do not go to great lengths to optimize the representation of individual lines. Furthermore, while the number of lines in a buffer is typically significantly greater than the number of items in a line, a typical buffer may contain at most a few thousand lines, making the representation of the outer sequence fairly insignificant as well.

Perhaps the main disadvantage of a line-oriented representation compared to a gap-buffer representation is that transferring items to and from a file is slower. With a gap-buffer representation, the representation in memory and the representation in a file are very similar, making the transfer almost trivial. With a line-oriented representation, when a buffer is created from the content of a file, each line separator must be handled by the creation of a new representation of a line.

However, with modern processors, the time to load and store a buffer is likely to be dominated by the input/output operations. Furthermore, the number of lines in a typical buffer is usually very modest. For that reason, a line-oriented representation does not incur any serious performance penalty compared to a gap buffer.

2.2 Updating views

When interactive full-display text editors first started to appear, the main issue with updating a view was to minimize the number of bytes that had to be sent to a CRT terminal; this issue was due to the relative slowness of the communication line between the computer and the terminal. To accomplish this optimization, the *redisplay* function compared the previous view to the next one, and attempted to issue terminal-specific editing operations to turn the screen content into the updated version. Of course, most of the time, the task consisted of positioning the cursor and inserting a single character.

Today, there is no need to minimize the number of editing operations on a terminal; it is perfectly feasible to redraw the entire view for each iteration of the event loop. However, today we have many more requirements on a text editor. In the most advanced cases, we would like for an *incremental parser* in the view to keep a structured version of the buffer content, for various purposes, such as syntax highlighting, language-specific completion and parsing, etc. An incremen-

[2]What we call a *cursor* in this paper is called a *point* in GNU Emacs terminology.

tal parser may require considerable computing power. It is therefore of utmost importance that as little work as possible is done each time around the event loop. Representing the entire editor buffer as a gap buffer does not lend itself to such advanced incremental processing.

In fact, most existing editors have very primitive parsers, mainly because the buffer representation does not necessarily lend itself to efficient incremental parsing.

2.3 Existing editors

2.3.1 GNU Emacs

GNU Emacs [3] [1] uses a *gap buffer* for the entire buffer of text, as described in Section 2.1.1.

Creating sophisticated parsers for the content of a buffer in GNU Emacs is not trivial. For that reason, existing parsers are typically fairly simple. For example, the parser for Common Lisp source code is unable to recognize the role of symbols in different contexts, such as the use of a Common Lisp symbol as a lexical variable. As a result, syntax highlighting can become confusing, and indentation is sometimes incorrect.

2.3.2 Multics Emacs

Multics Emacs[3] [2] was the first Emacs implementation written in Lisp, specifically, Multics MacLisp. It therefore pre-dates GNU Emacs.

Multics Emacs used a doubly linked list of lines, with the line content itself separate from the linked structure. All but a single line were said to be *closed*, and the content of a closed line was represented as a compact character string.

For the current line, a new MacLisp data type was added to the Multics MacLisp implementation, and it was called a *rplacable string*. Such a string could be seen as an ordinary MacLisp string, but could also have characters inserted or deleted through the use of primitives written in assembler and using special instructions on the GE 645 processor.

2.3.3 Climacs

Like GNU Emacs, Climacs uses a gap buffer for the entire buffer. It avoids the bad case by using a circular buffer. In fact, it uses Flexichain [5].

Climacs is able to accommodate fairly sophisticated parsers for the buffer content. But in order to avoid a complete analysis of the entire buffer content for each view update, such parsers must be *incremental*.

Information about the state of such parsers at various positions in the buffer must be kept and compared between view updates. Unfortunately, the gap-buffer representation does not necessarily lend itself to storing such information. The workaround used in Climacs is to define a large number of *cursors* to hold parser state at various places in the buffer, but managing these cursors is a non-trivial task.

2.3.4 Others

Hemlock uses a doubly linked list of lines. Each line is a `struct` containing a reference to the previous line and a reference to the next line. No more than one line is *open* at any point in time, and then the content is stored separately in a gap buffer. The gap-buffer data is contained in special variables and not encapsulated in a class or a `struct`.

[3]The description in this section is a summary of the information found here: http://www.multicians.org/mepap.html

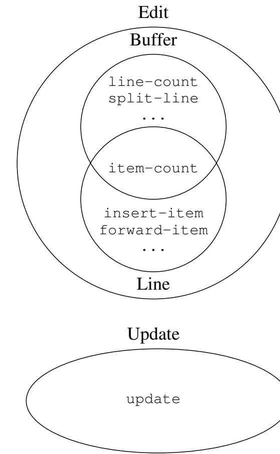

Figure 2: External protocols.

Goatee was written to be the input editor of McCLIM. Like Hemlock, it uses a doubly linked list of lines, with the difference that the line content itself is separate from the doubly linked structure. Lines are represented by a gap buffer. The gap buffer is encapsulated in a library called Flexivector, which was later extended to become the Flexichain library.

3. OUR TECHNIQUE

3.1 Protocols

Recall from Section 1 the existence of two nested control loops, the *inner* control loops in which each iteration is executing a single edit operation, and the *outer* control loop for the purpose of updating views.

The inner control loop is catered to by two different protocols; one containing operations on individual *lines* of items and one containing operations at the *buffer* level, concerning mainly the creation and deletion of lines. While we supply reasonable implementations of both these protocols, we also allow for sophisticated clients to substitute specific implementations of each one.

The outer control loop is catered to by the *update protocol*. This protocol is based on the concept of *time stamps*. In order to request an update, client code supplies the time stamp of the previous similar request in addition to four different functions (`sync`, `skip`, `modify`, and `create`). These functions can be thought of as representing editing operations on the lines of the buffer. Our protocol implementation calls these functions in an order that will update the buffer content from its previous to its current state. The implementations of these functions are supplied by client code according to its own representation of the buffer content.

Figure 2 illustrates the relationship between these protocols.

The protocols illustrated in Figure 2 are related to one another by the *protocol classes* that they operate on. The buffer-edit protocol operates on instances of the protocol class named `buffer`. The line-edit protocol operates on instances of the two protocol classes `line` and `cursor`. These protocols are tied together by an internal protocol class named `dock`. Figure 3 illustrates the participation of these

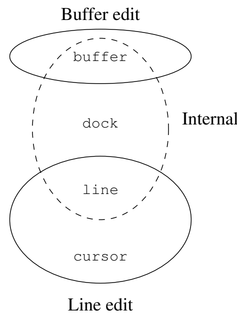

Figure 3: Participation of classes in protocols.

protocol classes in the different protocols, omitting the update protocol.

The internal protocol contains generic functions for which methods must be created that specialize to different *implementations* of the buffer-edit and the line-edit protocols. Client code using the library is not concerned with the existence of the internal protocol.

3.2 Supplied implementations

For the *line protocol*, we supply two different implementations, the *standard line* implementation and the *simple line* implementation. Similarly, for the *buffer protocol*, we supply two different implementations, the *standard buffer* implementation and the *simple buffer* implementation.

3.2.1 Standard line implementation

The standard line implementation is the one that a typical application would always use, unless an application-specific line implementation is desired.

To appreciate the design of the standard line, we need to distinguish between two different *categories* of operations on a line. We call these categories *editing operations* and *content queries*, respectively. An editing operation is one in which the content of the line is modified in some way, and is the result of the interaction of a user typing text, inserting or removing a *region* of text, or executing a *keyboard macro* that results in one or more editing operations. A content query happens as a result of an *event loop* or a *command loop* updating one or more *views* of the content.

A crucial observation related to these categories is that content queries are the result of *events* (typically, the user typing text or executing commands). The frequency of such events is fairly low, giving us ample time to satisfy such a query. Editing operations, on the other hand, can be arbitrarily more frequent, simply because a single keystroke on the part of the user can trigger a very large number of editing operations.[4]

This implementation supplies two different representations of the line that we call *open* and *closed* respectively. A line is *open* if the last operation on it was an editing operation. It is *closed* if the last operation was a content query in the form of a call to the generic function `items`. Accordingly, a line is changed from being open to being closed whenever there is a content query, and from closed to open when there is a call to an editing operation.

A closed line is represented as a Common Lisp simple vector. An open line is represented as a gap buffer. (See Section 2.) The protocol specifically does not allow for the caller of a content query to modify the vector returned by the query. This restriction allows us to return the same vector each time there is a content query without any intervening editing operation, thus making it efficient for views to query closed lines repeatedly. Similarly, repeated editing operations maintain the line open, making such a sequence of operations efficient as well.

Clearly, the typical use case when a user issues keystrokes, each one resulting in a simple editing operation such as inserting or deleting an item, followed by an update of one or more views of the buffer content is not terribly efficient. The reason for this inefficiency is that this use case results in a line being alternately opened (as a result of the editing operation) and closed (as a result of the view update) for each keystroke. However, this use case does not have to be very efficient, again because the costly operations are invoked at the frequency of the event loop. The use case for which the standard line design was optimized is the one where a single keystroke results in several simple editing operations, i.e., the exact situation in which performance is crucial.

3.2.2 Simple line implementation

We supply a second implementation, called the *simple line*, for the line editing protocol. The main purpose of this implementation is to serve as a reference for *random tests*. The idea here is that the implementation of the simple line is trivial, so that the correctness of the implementation is mostly obvious from inspecting the code, and in any case, it is unlikely that a defect in the simple line and another defect in the standard line will result in the same external behavior on a large body of randomly generated operations.

In addition to serving as a reference implementation for testing the standard line, this implementation can also serve as a reference for programmers who would like to create their own implementation of the line editing protocol.

The simple line implementation provides a single line abstraction, implemented as a Common Lisp simple vector. Each editing operation is implemented as reallocation of a new vector followed by calls to `replace` to copy items from the original line content to the one resulting from the editing operation. Clearly, this technique is very inefficient. For that reason, it is not recommended to use the simple implementation in client code.

3.2.3 Standard buffer implementation

The main performance challenge for the buffer implementation is to obtain acceptable performance in the presence of multiple views (into a single buffer) that are far apart, and that both issue editing operations in each interaction. The typical scenario would be a user having two views, one close to the beginning of the buffer and one close to the end of the buffer, while executing a keyboard macro that deletes from one of the views and inserts into the other.

This time, the performance challenge has to do with the *update protocol* rather than with the edit protocols. A naive

[4]It is of course possible to supply *aggregate* operations that alleviate the problem of frequent editing operations. In particular, it is possible to supply operations that insert a *sequence* of items, and that delete a *region* of items. However, such operations complicate the implementations of the protocol. Worse, there are still cases where many simple editing operations need to be executed, in particular as a result of executing keyboard macros.

buffer implementation would have to iterate over all the lines each time the update protocol is invoked.

To obtain reasonable performance in the presence of multiple views, the standard buffer implementation uses a *splay tree* [4] with a node for each line in the buffer. A splay tree is a *self-adjusting* binary tree, in that nodes that are frequently used migrate close to the root of the tree. Although the typical use of splay trees and other tree types is to serve as implementations of *dictionaries*, an often overlooked fact is that all trees can be used to implement *editable sequences*; that is how we use the splay tree here.

In addition to containing a reference to the associated line, each node in the splay tree contains time stamps corresponding to when the line was created and last modified. In addition, each node also contains summary information for the entire subtree rooted at this node. This summary information is what allows us to skip entire subtrees when a view requests update information and no node in the subtree has been modified since the last update request.

Finally, each node contains both a line count and an item count for the entire subtree, so that the offset of a particular line or a particular item can be computed efficiently, at least for nodes that are close to the root of the tree.

3.2.4 Simple buffer implementation

As with the implementations of the line-edit protocol, we supply a second implementation, called the *simple buffer*, for the buffer editing protocol as well. Again, the main purpose of this implementation is to serve as a reference for *random tests*. As with the simple line implementation, the implementation of the simple buffer is trivial, so that the correctness of the implementation is mostly obvious from inspecting the code.

The simple buffer implementation represents the buffer as a Common Lisp vector of nodes, where each node contains a line and time stamps indicating when a line was created and last modified.

4. BENEFITS OF OUR TECHNIQUE

There are several advantages to our technique compared to other existing solutions.

First, most techniques expose a more concrete representation of the buffer to client code, such as a doubly linked list of lines. Our technique is defined in terms of an abstract CLOS protocol that can have several potential implementations.

Furthermore, our *update protocol* based on time stamps provides an elegant solution to the problem of updating multiple views at different times and with different frequencies. In addition, the standard buffer implementation provided by our library provides an efficient implementation of the update protocol.

Our technique can be *customized* by the fact that the buffer editing protocol and the line editing protocol are independent. Client code with specific needs can therefore replace the implementation of one or the other or both according to its requirements. Thanks to the existence of the CLOS protocol, such customization can be done gradually, starting with the supplied implementations and replacing them as requirements change.

The standard line implementation supplied makes it possible to obtain reasonable performance for aggregate editing operations even when these operations are implemented as iterative calls to elementary editing operations. This quality makes it possible for client code to be simpler, for obvious benefits.

Finally, our technique is not specific to the abstractions of any particular existing editor, making our library useful in a variety of potential clients.

5. CONCLUSIONS AND FUTURE WORK

In this paper, we have defined a CLOS protocol for manipulating text editor buffers. The protocol is divided into several sub-protocols, so that sophisticated clients can provide specific implementations according to the requirements of the application.

The *update* sub-protocol was designed to be used by an arbitrary number of *views*. The implementation we supply is fast so that this protocol can be invoked for events such as keystrokes, exposures, and changes of window geometry.

In the remainder of this section, we outline future plans for the library.

5.1 Layer for Emacs compatibility

We plan to define a protocol layer on top of the edit protocols with operations that have the same semantics as the buffer protocol of GNU Emacs. Mainly, this work involves hiding the existence of individual lines, and treating the separation between lines as if it contained a *newline* character.

When a cursor is moved forward beyond the end of a line, or backward beyond the beginning of a line, this compatibility layer will have to detach the cursor from the line that it is currently attached to and re-attach it to the following or preceding line as appropriate.

Other minor operations need to be adapted, such as computing the item count of the entire buffer. This calculation will have to consider the separation of each pair of lines to contribute another item to the item count of the buffer.

5.2 Incremental Common Lisp parser

One of the essential reasons for the present work is to serve as an intermediate step towards the creation of a fully featured editor for Common Lisp code, entirely written in Common Lisp. Such an editor must be able to analyze the buffer content at least at the same frequency as that of the update of a view.

To that end, we plan to create a framework that allows the incremental parsing of the buffer content as Common Lisp code. Such a framework should allow for features such as syntax highlighting and automatic code indentation. Preferably, it should have a fairly accurate representation of the code such that the role of various code fragments can be determined. For example, it would be preferable to distinguish between a symbol in the `common-lisp` package when it is used to name a Common Lisp function and when (as the Common Lisp standard allows) it is used as a lexical variable with no relation to the standard function.

The first step of this incremental parser framework will be to adapt an implementation of the Common Lisp `read` function so that it can be used for incremental parsing, and so that the interpretation of *tokens* can take into account the specific situation of an editor buffer.

To take into account the different roles of symbols, the framework needs to include a *code walker* so that the occurrence of macro calls will not hamper the analysis.

5.3 Thread safety

The current implementation assumes that access is single-threaded. We plan to make multi-threaded access possible and safe. Implementing thread safety is not particularly difficult in itself. The interesting part would be to determine whether it is possible to achieve multi-threaded access without using a global lock for the entire buffer for every elementary operation.

Since each line is a separate object, it would appear that locking a single line would be sufficient for most operations such as inserting or deleting a single item. However, the current implementation also keeps the *item count* of the entire buffer up to date for each such operation.

However, the item count for the entire buffer is typically asked for only at the frequency of the update protocol, for instance, in order to display this information to the end user. Other situations exist when this information is needed, for example when an operation to go to a particular item offset is issued. But such operations are relatively rare.

This analysis suggests that it may be possible to update the global item count lazily. Each line would be allowed to have a different item count from what is currently stored in the buffer data structure, and the buffer itself would maintain a set of lines with modified item counts. When the global item count of the buffer is needed, this set is first processed so that the global item count is up to date.

6. ACKNOWLEDGMENTS

We would like to thank Daniel Kochmański, Bart Botta, and Matthew Alan Martin for providing valuable feedback on early versions of this paper.

APPENDIX

A. PROTOCOL

In this section, we describe the protocols that are implemented by our library.

For each class, generic function, and condition type, we include only a brief description. In particular, we do not include a complete description of the exceptional situations possible. For a complete description, see the `Documentation` subdirectory in the repository at GitHub.[5]

A.1 Classes

`buffer` [*Protocol Class*]
This class is the base class of all buffers. Each different buffer implementation defines specific implementation classes to be instantiated by client code.

`line` [*Protocol Class*]
This class is the base class of all lines. Each different line implementation defines specific implementation classes to be instantiated by client code.

`cursor` [*Protocol Class*]
This class is the base class of all cursors. Each different line implementation defines specific implementation classes to be instantiated by client code.

A.2 Generic functions

`item-count` *entity* [*GF*]
If *entity* is a line, then return the number of items in that line. If *entity* is a cursor, return the number of items in the line in which *cursor* is located. If *entity* is a buffer, then return the number of items in the buffer.

`item-at-position` *line position* [*GF*]
Return the item located at *position* in *line*.

`insert-item-at-position` *line item position* [*GF*]
Insert *item* into *line* at *position*.

After this operation completes, what happens to cursors located at *position* before the operation depends on the class of the cursor and of *line*.

`delete-item-at-position` *line position* [*GF*]
Delete the item at *position* in *line*.

`cursor-position` *cursor* [*GF*]
Return the position of *cursor* in the line to which it is attached.

`(setf cursor-position)` *new-position cursor* [*GF*]
Set the position of *cursor* to *new-position* in the line to which *cursor* is attached.

`insert-item` *cursor item* [*GF*]
Calling this function is equivalent to calling `insert-item-at-position` with the line to which *cursor* is attached, *item*, and the position of *cursor*.

`delete-item` *cursor* [*GF*]
Delete the item immediately after *cursor*.

Calling this function is equivalent to calling `delete-item-at-position` with the line to which *cursor* is attached and the position of *cursor*.

`erase-item` *cursor* [*GF*]
Delete the item immediately before *cursor*.

Calling this function is equivalent to calling `delete-item-at-position` with the line to which *cursor* is attached and the position of *cursor* minus one.

`cursor-attached-p` *cursor* [*GF*]
Return *true* if and only if *cursor* is currently attached to some line.

`detach-cursor` *cursor* [*GF*]
Detach *cursor* from the line to which it is attached.

`attach-cursor` *cursor line* `&optional` *(position 0)* [*GF*]
Attach *cursor* to *line* at *position*.

`beginning-of-line-p` *cursor* [*GF*]
Return *true* if and only if *cursor* is located at the beginning of the line to which *cursor* is attached.

`end-of-line-p` *cursor* [*GF*]
Return *true* if and only if *cursor* is located at the end of the line to which *cursor* is attached.

`beginning-of-line` *cursor* [*GF*]
Position *cursor* at the very beginning of the line to which it is attached.

`end-of-line` *cursor* [*GF*]
Position *cursor* at the very end of the line to which it is attached.

`forward-item` *cursor* [*GF*]
Move *cursor* forward one position.

`backward-item` *cursor* [*GF*]
Move *cursor* backward one position.

`update` *buffer time sync skip modify create* [*GF*]
This generic function is the essence of the update protocol. The *time* argument is a time stamp that can be `nil` (meaning the creation time of the buffer) or a value returned by previous invocations of `update`. The arguments *sync*, *skip*, *modify*, and *create*, are functions. The *sync* function is called with the first unmodified line following a sequence of modified lines. The *skip* function is called with a number indi-

[5] `https://github.com/robert-strandh/Cluffer`

cating the number of lines that have not been altered. The *modify* function is called with a line that has been modified. The *create* function is called with a line that has been created.

B. REFERENCES

[1] C. A. Finseth. *The Craft of Text Editing – Emacs for the Modern World*. Springer-Verlag, 1991.

[2] B. S. Greenberg. Multics emacs (prose and cons): A commercial text-processing system in lisp. In *Proceedings of the 1980 ACM Conference on LISP and Functional Programming*, LFP '80, pages 6–12, New York, NY, USA, 1980. ACM.

[3] B. Lewis, D. LaLiberte, and R. Stallman. *GNU Emacs Lisp Reference Manual*. Free Software Foundation, Boston, MA, USA, 2014.

[4] D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *J. ACM*, 32(3):652–686, July 1985.

[5] R. Strandh, M. Villeneuve, and T. Moore. Flexichain: An editable sequence and its gap-buffer implementation. In *Proceedings of the Lisp and Scheme Workshop*, 2004.