Cluffer

A library for text-editor buffers.

Robert Strandh

2015

Contents

Chapter 1

Introduction

Cluffer is a library for representing the buffer of a text editor. As such, it defines a set of CLOS *protocols* for client code to interact with the buffer contents in various ways, and it supplies different *implementations* of those protocols for different purposes.

The buffer protocols have been chosen so that they can fit a variety of editors. As a consequence, they are not particularly Emacs-centric. For example, in Emacs, a newline character is just another character, so that moving past it using the `forward-char` command changes the line in which *point* is located, and using the `delete-char` command when *point* is to the left of a newline character joins the line to the next one.

In contrast, the buffer protocols documented here are *line oriented* and there is no newline character; only a sequence of lines. At some level, it is of course desirable to have Emacs-compatible commands, but these commands are written separately, using this buffer protocol to accomplish the effects. For example, the Emacs-compatible `forward-item` command checks whether it is at the end of a line, and if so, detaches the cursor from that line and attaches it to the next one. Similarly, the Emacs compatible `delete-item` command calls `join-line` in the buffer protocol to obtain the desired effect when it is at the end of a line.

By writing the editor commands in two levels like this, we hope it will be easier to use the buffer protocols to write emulators for other editors, such as `VIM`.

The buffer participates in two different buffer protocols:

1. The *edit protocol*, used by client editing and cursor-motion operations.
2. The *update protocol*, used by redisplay operations to determine what items are contained in the buffer.

The operations in the edit protocol were designed to be *fast* (typically around 10 μs) so that it is practical to use these operations in a loop, say to insert or delete a region, or to accomplish several operations inside a keyboard macro. The exceptions are the operations `split-line` and `join-line` that take time proportional to the number of items in the second line.[1]

The operations in the update protocol were designed to be called at the frequency of the *event loop* of an application, typically once for each character typed, but also when a window is resized or scrolled (in which case, these operations are very fast since no modifications to the buffer have occurred).

The buffer edit protocols expose two levels of abstraction to client code:

1. The *buffer* level represents the *sequence of lines* independently of how the individual lines are represented.
2. The *line* level represents individual lines.

As mentioned above, the buffer protocols do not pretend to manage any equivalence between line breaks and some sequence of characters. It is up to client code to model such an equivalence if desired. As a consequence, the buffer protocols do not allow for a cursor at the beginning of a line to move backward or a cursor at the end of a line to move forward. An attempt at doing so will result in an error being signaled. If client code wants to impose a model where the line break corresponds to (say) the *newline* character, then it must explicitly detach and reattach the cursor to a different line in these cases. It can manage that in two different ways: either by explicitly testing for `beginning-of-line` or `end-of-line` before calling the equivalent buffer function, or by handling the error that results from the attempt.

[1] We may improve on this performance in the future.

The buffer also does not interpret the meaning of any of items contained in it. For instance, whether an item is to be considered part of a *word* or not, is not decided at the buffer level, but at the level of the syntax. As a consequence, the buffer protocol does not offer any functions that require such interpretation, such as `forward-word`, `end-of-paragraph`, etc.

Chapter 2

External protocols

2.1 Package

All symbols in the external protocol are in the package named `cluffer`. We recommend against client code *using* this package in the sense of the `:use` option to `defpackage` or in the sense of calling `use-package`.

The reason for this recommendation is that we can not guarantee that future additions to this library will not define external symbols that conflict with symbols in the `common-lisp` package.

Instead, we recommend that client code use explicit package prefixes, which in addition will make the origin of the symbol obvious from the source code.

If, for some reason, it is not desirable to use explicit package prefixes, we suggest selectively importing the desired symbols.

2.2 Conditions

Cluffer defines a number of conditions that are signaled when Cluffer is unable to fulfill the contract stipulated by the protocol function being used.

⇒ `cluffer:cluffer-error` [*Condition*]

This condition type is the base of all error conditions signaled by Cluffer. Client code that wishes to handle all error conditions signaled by Cluffer may use this condition in its condition handlers.

⇒ `cluffer:end-of-line` [*Condition*]

This condition is signaled when an attempt is made to use a position that is too large, either by moving a cursor there, or by attempting to access an item in such a position. Notice that in some cases, "too large" means "strictly greater than the number of items in a line", and sometimes it means "greater than or equal to the number of items in a line". For example, it is perfectly acceptable to move a cursor to a position that is equal to the number of items in a line, but it is not acceptable to attempt to access an item in a line at that position.

⇒ `cluffer:beginning-of-buffer` [*Condition*]

⇒ `cluffer:end-of-buffer` [*Condition*]

⇒ `cluffer:cursor-attached` [*Condition*]

This condition is signaled when an attempt is made to use a cursor in an operation that requires that cursor to be detached, but the cursor used in the operation is attached to a line.

⇒ `cluffer:cursor-detached` [*Condition*]

This condition is signaled when an attempt is made to use a cursor in an operation that requires that cursor to be attached, but the cursor used in the operation is not attached to any line.

⇒ `cluffer:cursors-are-not-comparable` [*Condition*]

This condition is signaled when an attempt is made to compare two cursors which are each attached to a line but the lines do not belong to the same buffer. The readers `cursor1` and `cursor2` can be used to obtain the offending cursor objects.

⇒ `cluffer:line-detached` [*Condition*]

This condition is signaled when an attempt is made to use a line in an operation that requires the line to be attached to a buffer, but the line used in the operation is not attached to a buffer. An example of such an operation would be to attempt to get the line number of the line, given that the line number of

a line is determined by the buffer to which the line is attached.

⇒ `cluffer:object-must-be-line` [*Condition*]

This condition is signaled by protocol generic functions that take a line object as an argument, but something other than a line object was given.

⇒ `cluffer:object-must-be-buffer` [*Condition*]

This condition is signaled by protocol generic functions that take a buffer object as an argument, but something other than a buffer object was given.

2.3 Line edit protocol

2.3.1 Protocol classes

⇒ `line` [*Class*]

This class is the base class for all lines. It should not itself be instantiated. Instead, Cluffer contains two different modules each supplying a different subclass of `line` that can be instantiated.

⇒ `cursor` [*Class*]

This is the base class for all cursors.

⇒ `:line` [*Initarg*]

The class `cursor` accepts this initarg which is the line to which the new cursor is to be attached.

⇒ `:cursor-position` [*Initarg*]

The class `cursor` accepts this initarg which is the position to which the new cursor is initialized on the attached line.

2.3.2 Operations on lines and cursors

⇒ `cursor-position` *cursor* [*Generic Function*]

Return the position of *cursor* in the line to which it is attached.

If *cursor* is not currently attached to a line, a condition of type `cursor-detached` is signaled.

⇒ `(setf cursor-position)` *new-position cursor* [*Generic Function*]

Set the position of *cursor* to *new-position* in the line to which *cursor* is attached.

If *cursor* is not currently attached to a line, a condition of type `cursor-detached` is signaled.

If *new-position* is negative, then a condition of type `beginning-of-line` is signaled. If *new-position* is strictly greater than the number of items in the line to which `cursor` is attached (See the generic function `item-count`), then a condition of type `end-of-line` is signaled.

⇒ `beginning-of-line-p` *cursor* [*Generic Function*]

Return *true* if and only if *cursor* is located at the beginning of the line to which *cursor* is attached.

If *cursor* is not currently attached to a line, a condition of type `cursor-detached` is signaled.

Calling this function is equivalent to calling the function `cursor-position` with *cursor* as argument and comparing the return value to 0. However, this function might be implemented differently for reasons of performance.

⇒ `end-of-line-p` *cursor* [*Generic Function*]

Return *true* if and only if *cursor* is located at the end of the line to which *cursor* is attached.

If *cursor* is not currently attached to a line, a condition of type `cursor-detached` is signaled.

Calling this function is equivalent to calling the function `cursor-position` with *cursor* as argument and comparing the return value to the number of items in the line to which *cursor* is attached (See the generic function `item-count`). However, this function might be implemented differently for reasons of performance.

⇒ `beginning-of-line` *cursor* [*Generic Function*]

Position *cursor* at the very beginning of the line to which it is attached.

If *cursor* is not currently attached to a line, a condition of type `cursor-detached` is signaled.

Calling this function is equivalent to calling the function `(setf cursor-position)` with 0 and *cursor* as arguments. However, this function might be implemented differently for reasons of performance.

⇒ `end-of-line` *cursor* [*Generic Function*]

Position *cursor* at the very end of the line to which it is attached.

If *cursor* is not currently attached to a line, a condition of type `cursor-detached` is signaled.

Calling this function is equivalent to calling the function `(setf cursor-position)` with the number of items in the line to which *cursor* is attached (See generic function `item-count`) and *cursor* as arguments. However, this function might be implemented differently for reasons of performance.

⇒ `forward-item` *cursor* [*Generic Function*]

Move *cursor* forward one position.

If *cursor* is not currently attached to a line, a condition of type `cursor-detached` is signaled.

Calling this function is equivalent to incrementing the `cursor-position` of *cursor*. However, this function might be implemented differently for reasons of performance.

⇒ `backward-item` *cursor* [*Generic Function*]

Move *cursor* backward one position.

If *cursor* is not currently attached to a line, a condition of type `cursor-detached` is signaled.

Calling this function is equivalent to decrementing the `cursor-position` of *cursor*. However, this function might be implemented differently for reasons of performance.

⇒ `item-at-position` *line position* [*Generic Function*]

Return the item located at *position* in *line*.

If *position* is less than zero, a condition of type `beginning-of-line` is signaled. If *position* is greater than or equal to the number of items in *line* (See the definition of the generic function `item-count`.), a condition of type `end-of-line` is signaled.

⇒ `item-after-cursor` *cursor* [*Generic Function*]

Return the item located immediately after *cursor*.

If *cursor* is not currently attached to a line, a condition of type `cursor-detached` is signaled.

Calling this function is equivalent to calling `item-at-position` with the line to which the cursor is attached (see generic function `line`) and the position of *cursor* (see generic function `cursor-position`). However, this function might be implemented differently for reasons of performance.

⇒ `item-before-cursor` *cursor* [*Generic Function*]

Return the item located immediately before *cursor*.

If *cursor* is not currently attached to a line, a condition of type `cursor-detached` is signaled.

Calling this function is equivalent to calling `item-at-position` with the line to which the cursor is attached (see generic function `line`) and the position of *cursor* (see generic function `cursor-position`) minus one. However, this function might be implemented differently for reasons of performance.

⇒ `insert-item-at-position` *line item position* [*Generic Function*]

Insert *item* into *line* at *position*.

If *position* is less than zero, a condition of type `beginning-of-line` is signaled. If *position* is greater than the number of items in *line* (See the definition of the generic function `item-count`.), a condition of type `end-of-line` is signaled.

After this operation completes, what happens to cursors located at *position* before the operation depends on the class of the cursor and of *line*. The `standard-line` implementation provides two kinds of cursors, namely *left-sticky* cursors and *right-sticky* cursors. For such an implementation, after this

operation completes, any left-sticky cursor located at *position* will be located before *item*, and any right-sticky cursor located *position* will be located after *item*.

⇒ `delete-item-at-position` *line position* [*Generic Function*]

Delete the item at *position* in *line*.

If *position* is less than zero, a condition of type `beginning-of-line` is signaled. If *position* is greater than or equal to the number of items in *line* (See the definition of the generic function `item-count`.), a condition of type `end-of-line` is signaled.

⇒ `insert-item` *cursor item* [*Generic Function*]

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*. However, this function might be implemented differently for reasons of performance.

If *cursor* is not currently attached to a line, a condition of type `cursor-detached` is signaled.

⇒ `delete-item` *cursor* [*Generic Function*]

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*. However, this function might be implemented differently for reasons of performance.

If *cursor* is not currently attached to a line, a condition of type `cursor-detached` is signaled.

⇒ `erase-item` *cursor* [*Generic Function*]

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. However, this function might be implemented differently for reasons of performance.

If *cursor* is not currently attached to a line, a condition of type `cursor-detached`

is signaled.

⇒ `cursor-attached-p` *cursor* [*Generic Function*]

Return *true* if and only if *cursor* is currently attached to some line.

⇒ `detach-cursor` *cursor* [*Generic Function*]

Detach *cursor* from the line to which it is attached.

If *cursor* is already detached, a condition of type `cursor-detached` is signaled.

⇒ `attach-cursor` *cursor line* `&optional` *(position 0)* [*Generic Function*]

Attach *cursor* to *line* at *position*. If *position* is supplied and it is greater than the number of items in *line*, the error condition `end-of-line` is signaled. If *cursor* is already attached to a line, the error condition `cursor-attached` is signaled.

⇒ `item-count` *entity* [*Generic Function*]

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 cursor that is not currently attached to a line, a condition of type `cursor-detached` is signaled.

Note: the argument *entity* can also be a buffer, in which case the total number of items is returned. (See Section 2.4.)

⇒ `line` *cursor* [*Generic Function*]

Return the line in which *cursor* is located.

If *cursor* is not currently attached to a line, a condition of type `cursor-detached` is signaled.

⇒ `buffer` *entity* [*Generic Function*]

Return the buffer of *entity*. If *entity* is a line, then return the buffer to which the line is attached. If *entity* is a cursor, then return the buffer of the line to which the cursor is attached.

If *entity* is a cursor that is not currently attached to a line, a condition of type `cursor-detached` is signaled.

If *entity* is a line that is not currently attached to a buffer, a condition of type `line-detached` is signaled.

Comparing Cursors

Cursors which are attached to lines which belong to the same buffer can be lexicographically ordered based on their line numbers and within-line positions. The following functions allow comparing cursor objects according to this order:

⇒ `cursor<` *cursor* `&rest` *more-cursors* [*Function*]

Return true if for each adjacent pair of cursors (c_1, c_2) in the sequence of cursors consisting of *cursor* followed by *more-cursors*, c_1 is positioned before c_2 in the buffer. This function calls the generic function `cursor</2` for each such pair to check whether the property holds. As a consequence, return true if only *cursor* is supplied.

If any of the cursors is not currently attached to a line, a condition of type `cursor-detached` is signaled. Unless all cursors are attached to lines which belong to the same buffer, a condition of type `cursors-are-not-comparable` is signaled.

⇒ `cursor<=` *cursor* `&rest` *more-cursors* [*Function*]

Return true if for each adjacent pair of cursors (c_1, c_2) in the sequence of cursors consisting of *cursor* followed by *more-cursors*, c_1 is positioned before c_2 or at the same position as c_2 in the buffer. This function calls the generic function `cursor<=/2` for each such pair to check whether the property holds. As a consequence, return true if only *cursor* is supplied.

If any of the cursors is not currently attached to a line, a condition of type `cursor-detached` is signaled. Unless all cursors are attached to lines which belong to the same buffer, a condition of type `cursors-are-not-comparable` is signaled.

⇒ `cursor=` *cursor* `&rest` *more-cursors* [*Function*]

Return true if all cursors in the sequence of cursors consisting of *cursor* followed by *more-cursors* are positioned at the same position in the buffer. This function calls the generic function `cursor=/2` for pairs of cursors to check whether the

property holds. As a consequence, return true if only *cursor* is supplied.

If any of the cursors is not currently attached to a line, a condition of type `cursor-detached` is signaled. Unless all cursors are attached to lines which belong to the same buffer, a condition of type `cursors-are-not-comparable` is signaled.

⇒ `cursor/=` *cursor* `&rest` *more-cursors* [*Function*]

Return true if no two cursors in the sequence of cursors consisting of *cursor* followed by *more-cursors* are positioned at the same position in the buffer. This function calls the generic function `cursor=/2` for *all* pairs of cursors to check whether the property is violated. As a consequence, return true if only *cursor* is supplied.

If any of the cursors is not currently attached to a line, a condition of type `cursor-detached` is signaled. Unless all cursors are attached to lines which belong to the same buffer, a condition of type `cursors-are-not-comparable` is signaled.

⇒ `cursor>=` *cursor* `&rest` *more-cursors* [*Function*]

Return true if for each adjacent pair of cursors (c_1, c_2) in the sequence of cursors consisting of *cursor* followed by *more-cursors*, c_1 is positioned after or at the same position as c_2 in the buffer. This function calls the generic function `cursor</2` for each such pair to check whether the property is violated. As a consequence, return true if only *cursor* is supplied.

If any of the cursors is not currently attached to a line, a condition of type `cursor-detached` is signaled. Unless all cursors are attached to lines which belong to the same buffer, a condition of type `cursors-are-not-comparable` is signaled.

⇒ `cursor>` *cursor* `&rest` *more-cursors* [*Function*]

Return true if for each adjacent pair of cursors (c_1, c_2) in the sequence of cursors consisting of *cursor* followed by *more-cursors*, c_1 is positioned strictly after c_2 in the buffer. This function calls the generic function `cursor<=/2` for each such pair to check whether the property is violated. As a consequence, return true if only *cursor* is supplied.

⇒ `cursor</2` *cursor1 cursor2* [*Generic Function*]

Return true if *cursor1* is positioned strictly before *cursor2* in the buffer.

If any of the cursors is not currently attached to a line, a condition of type `cursor-detached` is signaled. Unless all cursors are attached to lines which belong to the same buffer, a condition of type `cursors-are-not-comparable` is signaled.

⇒ `cursor<=/2` *cursor1 cursor2* [*Generic Function*]

Return true if *cursor1* is positioned before or at the same position as *cursor2* in the buffer.

If any of the cursors is not currently attached to a line, a condition of type `cursor-detached` is signaled. Unless all cursors are attached to lines which belong to the same buffer, a condition of type `cursors-are-not-comparable` is signaled.

⇒ `cursor=/2` *cursor1 cursor2* [*Generic Function*]

Return true if *cursor1* is positioned at the same position as *cursor2* in the buffer.

If any of the cursors is not currently attached to a line, a condition of type `cursor-detached` is signaled. Unless all cursors are attached to lines which belong to the same buffer, a condition of type `cursors-are-not-comparable` is signaled.

2.4 Buffer edit protocol

2.4.1 Protocol classes

⇒ `buffer` [*Class*]

This is the base class for all buffers. It should not itself be instantiated. Instead, Cluffer contains different modules, each providing a different subclass of this class that can be instantiated.

By default, it is recommended that client code instantiate the class `buffer` in the package `cluffer-standard-buffer`. (See Section 3.3.)

2.4.2 Operations on buffers

⇒ `beginning-of-buffer-p` *cursor* [*Generic Function*]

Return *true* if and only if *cursor* is located at the beginning of a buffer.

⇒ `end-of-buffer-p` *cursor* [*Generic Function*]

Return *true* if and only if *cursor* is located at the end of a buffer.

⇒ `beginning-of-buffer` *cursor* [*Generic Function*]

Position *cursor* at the very beginning of the buffer.

⇒ `end-of-buffer` *cursor* [*Generic Function*]

Position *cursor* at the very end of the buffer.

⇒ `split-line-at-position` *line position* [*Generic Function*]

Split *line* into two lines, the first cone containing the items preceding *position* and the second one containing the items following *position*. After this operation, any left-sticky cursor located at *position* will be located at the end of the first line, and any right-sticky cursor located at *position* will be located at the beginning of the second line.

This operation is implemented in one of two ways:

1. Either the prefix is moved from the original line to a new line. In this case, the second of the two resulting lines is `eq` to the original one.
2. Or the suffix is moved from the original line to a new line. In this case, the first of the two resulting lines is `eq` to the original one.

However, it is not specified which of the two techniques is used, and it is not specified what value is returned from this operation.

⇒ `split-line` *cursor* [*Generic Function*]

Calling this function is equivalent to calling `split-line-at-position`, passing it the line to which *cursor* is attached, and the position of *cursor*.

If *cursor* is not currently attached to a line, a condition of type `cursor-detached` is signaled.

⇒ `join-line` *entity* [*Generic Function*]

The argument *entity* may be a cursor or a line.

If *entity* is a line, then join that line with the line following it in the buffer to which the line is attached. If *entity* is a cursor, join the line to which the cursor is attached with the line following it in the buffer to which the line is attached.

If *entity* is a cursor that is not currently attached to a line, a condition of type `cursor-detached` is signaled.

If *entity* is a line that is not currently attached to a buffer, a condition of type `line-detached` is signaled.

If *entity* is a cursor and it is attached to the last line of the buffer, the error condition `end-of-buffer` will be signaled.

⇒ `line-count` *buffer* [*Generic Function*]

Return the number of lines in *buffer*.

⇒ `line-number` *entity* [*Generic Function*]

The argument *entity* may be a cursor or a line.

If *entity* is a cursor and that cursor is not attached to any line, a condition of type `cursor-detached` is signaled.

If *entity* is a cursor and that cursor is attached to a line, then the generic function `line` (See Section 2.3.) is called with *entity* as an argument and the return value is used as argument in a recursive call to `line-number`.

If *entity* is a line and that line is not attached to a buffer, then `nil` is returned.

If *entity* is a line and that line is attached to a buffer, then the line number of *line* in that buffer is returned. The first line of the buffer has the number 0.

⇒ `find-line` *buffer line-number* [*Generic Function*]

Return the line in the buffer with the given *line-number*. If *line-number* is less than 0 then the error `beginning-of-buffer` is signaled. If *line-number* is greater than or equal to the number of lines in the buffer, then the error `end-of-buffer` is signaled.

Notice that the edit protocol does not contain any `delete-line` operation. This design decision was made on purpose. By only providing `join-line`, we guarantee that removing a line leaves a *trace* in the buffer in the form of a modification operation on the first of the two lines that were joined. This features is essential in order for the *update protocol* to work correctly.

2.5 Buffer update protocol

The purpose of the buffer update protocol is to allow for a number of edit operations to the buffer without updating the view of the buffer. This functionality is important because a single command may result in an arbitrary number of edit operations to the buffer, and we typically want the view to be updated only once, when all those edit operations have been executed.

At the center of the update protocol is the concept of a *time stamp*. The nature of this time stamp is not specified, other than the fact that its value is incremented for each operation that alters the contents of the buffer. The only operation allowed by client code on a time stamp is to store it and pass it as an argument to the protocol function `update`.

⇒ `items` *line* `&key` *start end* [*Generic Function*]

Return the items of *line* as a vector. The keyword parameters *start* and *end* have the same interpretation as for Common Lisp sequence functions.

⇒ `update` *buffer time sync skip modify create* [*Generic Function*]

The *buffer* parameter is a buffer that might have been modified since the last update operation. The *time* parameter is the time stamp of the last time the update operation was called, so that the `update` function will report modifications since that time. In addition to time stamps, the *time* argument can also be `nil`, which is interpreted as the beginning of time. Thus, when `nil` is given as a value of this argument, the operations generated correspond to the creation of the buffer.

The `update` function returns a new time stamp to be used as the *time* argument in the next call to `update`.

The time stamp is specific to each buffer, and more importantly, to each *buffer*

implementation. The consequences are unspecified if a time stamp returned by calling `update` on one buffer is used in a call to `update` with a different buffer.

The parameters *sync*, *skip*, *modify*, and *create*, are functions that are called by the `update` function. They are to be considered as *edit operations* on some representation of the buffer as it was after the previous call to `update`. The operations have the following meaning:

- *sync* indicates the first unmodified line after a sequence of new or modified lines. Accordingly, this function is called once, following one or more calls to *create* or *modify*. This function is called with a single argument: the unmodified line. Client code must compare the current line of its view to the argument, and delete the current line repeatedly until the two are `eq`. Finally, it must make the immediately following line the current one.

- *skip* indicates that a number of lines have not been subject to any modifications since the last update. The function takes a single argument, the number of lines to skip. This function is called *first* to indicate that a *prefix* of the buffer is unmodified, or after a *sync* operation to indicate that that many lines following the one given as argument to the *sync* operation are unmodified. This operation is also called when there are unmodified lines at the end of the buffer so that the total line count of the buffer corresponds to the total number of lines mentioned in the sequence of operations.

- *modify* indicates a line that has been modified. The function is called with that line as an argument. Client code must compare the current line of its view to the argument, and delete the current line repeatedly until the two are `eq`. It must then take whatever action is needed for the modified contents of the line, and finally it must make the immediately following line the current one.

- *create* indicates a line that has been created. The function is called with that line as an argument.

Chapter 3

Supplied implementations

3.1 Standard line

3.1.1 Characteristics

The *standard line* implementation provides a reasonably-efficient implementation of the line-edit protocol. Two different line classes are provided, one called *open* and the other one called *closed*.

A line is open if it has been modified after the last time client code asked for its contents (as provided by the protocol generic function `items`). Otherwise the line is closed. Open lines are typically efficient for a number of consecutive editing operations at positions that are close to one another, as is typically the case when a region of text is inserted or deleted. Closed lines, on the other hand, are efficient when client code asks for the items contained in them.

Notice that for the typical scenario where an end user types text as one character at a time, the line into which the text is typed is going to be opened and closed once for every keystroke. When the end user types the character, the line will be opened in order to prepare it for the editing operation. Immediately after the editing operation terminates, the visible part of the text buffer is going to be *displayed* in some window on the screen. This display operation requires the client code to ask for the contents of the line, which will require

the line to be closed.

While this alternate opening and closing of the line may seem unreasonably inefficient, notice that it is happening at typing speed, so that there is typically ample time for these operations between two keystrokes typed. More importantly, the scenario of an end user typing text is not the one for which this design was optimized. Instead, it was designed for cases where a single keystroke results in multiple editing operations. Typical such scenarios include inserting and deleting a region of text, or executing a keyboard macro that results in multiple editing operations. The design of the standard line allows such operations to be implemented as a sequence of elementary editing operations without affecting performance. As long as the contents of the line is not asked for (presumably for that contents to be displayed), the line remains open.

In addition to two different types of line, the `standard-line` implementation provides two types of *cursors*, namely *left-sticky* and *right-sticky* cursors. The two cursor types differ in behavior when an item is inserted at the very position of the cursor. In this case, the left-sticky cursor keeps its old position, whereas the position of any right-sticky cursor is incremented. A right-sticky cursor is typically used for the place where end-user editing operations take place (the *point* in the terminology of Emacs). A left-sticky cursor can be used to mark the end of a region when the desired effect is that an insertion of an item at the end of the region should not be included in the region.

3.1.2 Package

The package named `cluffer-standard-line` is used for all names that are specific to the implementation of standard lines.

3.1.3 Classes

⇒ `line` [*Class*]

This class is a subclass of the protocol class named `line` in the package named `cluffer`.

⇒ `open-line` [*Class*]

This class is a subclass of the class named `line`. It is used when the contents of the line is modified.

For this class, the contents is represented as a simple *gap buffer* so that adding or deleting items is done at the beginning or the end of the gap.

⇒ `closed-line` [*Class*]

This class is a subclass of the class named `line`. It is used when the contents of the line has not been modified after an invocation of the function `items`.

For this class, the contents is represented by a simple Common Lisp vector. Whenever `items` is invoked on a closed line and the entire contents is asked for, this simple vector is returned, and no copy is made.

⇒ `cursor` [*Class*]

This class is the base class for the cursor classes provide by the `standard-line` implementation.

⇒ `left-sticky-cursor` [*Class*]

This class is used for cursors that should have their positions unaltered when an item is inserted at the very position of the cursor. It is a subclass of the class `cursor`.

⇒ `right-sticky-cursor` [*Class*]

This class is used for cursors that should have their positions incremented when an item is inserted at the very position of the cursor. It is a subclass of the class `cursor`.

3.2 Simple line

3.2.1 Characteristics

The simple line implementation was written mainly for testing purposes. No attempt has been made to optimize the implementation of the operations of a simple line.

A line in this implementation is implemented as Common Lisp simple vector.

Every simple editing operation requires that this vector be re-allocated with a different size.

Another use for this line implementation is as an illustration of the semantics of the various protocol generic functions, so that it can be used as a model for new implementations of the line-edit protocol.

3.2.2 Package

The package named `cluffer-simple-line` is used for all names that are specific to the implementation of simple lines.

3.2.3 Classes

⇒ `line` [*Class*]

This class is a subclass of the protocol class named `line` in the package named `cluffer`.

Contrary to the standard line (See Section 3.1.) the simple line has no concept of open or closed lines. All lines are represented the same way with the items stored in a simple Common Lisp vector.

3.3 Standard buffer

3.3.1 Characteristics

The standard buffer representation organizes the lines in a *splay tree* [ST85]. This organization has several advantages:

- A line that is modified moves to the root of the tree, and recently used lines stay close to the root, making some editing operations more efficient.
- It is computationally cheap to know the line number of the current line at all times.

3.3.2 Package

The package named `cluffer-standard-buffer` is used for all names that are specific to the implementation of the standard buffer.

3.3.3 Classes

⇒ `buffer` [*Class*]

This class is a subclass of the protocol class named `buffer` in the package named `cluffer`.

⇒ `:initial-line` [*Initarg*]

This initarg must be supplied when the `buffer` class is instantiated. This initarg is the mechanism by which the otherwise independent implementations of the buffer and the line protocols are connected.

3.4 Simple buffer

3.4.1 Characteristics

The simple buffer implementation was written mainly for testing purposes. No attempt has been made to optimize the implementation of the operations of a simple buffer.

The lines are stored in a Common Lisp simple vector. Whenever a line is added or removed, the vector is reallocated. To compute the number of items in the buffer, the sum of the items of each line is computed.

3.4.2 Package

The package named `cluffer-simple-buffer` is used for all names that are specific to the implementation of the simple buffer.

3.4.3 Classes

⇒ `buffer` [*Class*]

This class is a subclass of the protocol class named `buffer` in the package named `cluffer`.

⇒ `:initial-line` [*Initarg*]

Just as with the standard line, this initarg must be supplied when the `buffer` class is instantiated.

Chapter 4

Writing new implementations

4.1 Introduction

In general, we advise against the `:use` of packages other than the `common-lisp` package. For that reason, in the remainder of this chapter, we use explicit package prefixes to make it clear what symbols are referred to.

4.2 Writing new line implementations

4.2.1 Package

It is generally a good idea to define a new package for a new implementation of the concept of a line. For the remainder of this section, we use the name `new-line` for this package.

4.2.2 Classes

A class that is a subclass of `cluffer:line` must be provided. In the remainder of this section we refer to this class as `new-line:line`.

It is not mandatory to provide any implementation of the *cursor* abstraction, but if such an abstraction is provided, it is recommended that the root class of all cursor classes be a subclass of `cluffer:cursor`. In the remainder of this section we refer to this class as `new-line:cursor`.

Both the *standard line* (See Section 3.1.) and the *simple line* (See Section 3.2.) provide two kinds of cursor abstractions, namely *left-sticky* cursors and *right-sticky* cursors. If the new implementation provides different kinds of cursors, the different behavior can be accomplished either by subclassing as is done in both provided implementations, or it can be accomplished by some other means such as storing additional information in slots of the root class.

4.2.3 Methods

In addition to the methods documented in this section, a new implementation may define methods on generic functions for which default methods are provided.

In particular, Cluffer provides default methods on many cursor operations such as `delete-item`, `forward-item`, etc. that call more basic generic functions to accomplish the task. It might be advantageous for performance reasons for a new implementation to define methods on such functions in addition to the ones document here.

⇒ `item-count` *(line* `new-line:line`*)* [*Method*]

This method should return the number of items in *line*.

If the new implementation defines subclasses of `new-line:line` for which the item count is computed differently, as is the case for the standard line (See Section 3.1.) then this method must be replaced by a method for each subclass with a specific way of computing the item count.

⇒ `item-at-position` *(line* `new-line:line`*) position* [*Method*]

This method should return the item at *position* in *line*.

There is no need to check that *position* is a valid item position in *line*. This check is already taken care of by an auxiliary method on `item-at-position`.

As with `item-count`, if the new implementation defines subclasses of the root class `new-line:line` for which the item at a particular position is computed differently, as is the case for the standard line (See Section 3.1.) then this method must be replaced by a method for each subclass with a specific way of computing the item count.

⇒ `insert-item-at-position` *(line* `new-line:line`*) item position* [*Method*]

This method should insert *item* at *position* in *line*.

There is no need to check that *position* is a valid insertion position in *line*. This check is already taken care of by an auxiliary method on the generic function `insert-item-at-position`.

⇒ `delete-item-at-position` *(line* `new-line:line`*) position* [*Method*]

This method should delete the item at *position* in *line*.

There is no need to check that *position* is a valid item position in *line*. This check is already taken care of by an auxiliary method on the generic function `delete-item-at-position`.

⇒ `items` *(line* `new-line:line`*)* [*Method*]

This method should return all the items in *line*.

The generic function `items` is typically not called by implementations of the buffer protocol; only by client code. Therefore, the data type used to return the items is a matter between the line implementation and the application. Both the simple line (See Section 3.2.) and the standard line (See Section 3.1.) return the items as a simple vector. For applications that allow only character items, it might be a good idea to return the contents as a string instead.

It is even possible to omit this method altogether, since it is always possible for client code to obtain the items of a line by calling the generic function `item-at-position` for each possible position of the line.

⇒ `cursor-attached-p` *(cursor new-line:cursor)* [*Method*]

This method should return *true* if and only if *cursor* is currently attached to a line.

⇒ `cursor-position` *(cursor new-line:cursor)* [*Method*]

This method should return the position of *cursor* in the line to which *cursor* is attached.

There is no need to check that *cursor* is attached to a line. This check is already taken care of by an auxiliary method on the generic function `cursor-position`.

⇒ `(setf cursor-position)` *new-position (cursor new-line:cursor)* [*Method*]

This method should set the position of *cursor* to *new-position* in the line to which *cursor* is attached

There is no need to check that *cursor* is attached to a line. This check is already taken care of by an auxiliary method on `(setf cursor-position)`.

There is also no need to check that *new-position* is a valid cursor position in the line to which *cursor* is attached. This check is already taken care of by another auxiliary method on the generic function `(setf cursor-position)`.

⇒ `cluffer-internal:line-split-line` *(line new-line:line) position* [*Method*]

This method should split *line* at *position* and return the newly-created line, i.e., the line that holds the items that are initially located after *position* in *line*.

Cursors that are initially located after *position* in *line* should be detached from *line* and attached to the newly-created line.

Cursors that are initially located exactly at *position* can either be left attached to *line* (and will then be at the end), or they can be detached from *line* and attached to the newly-created line (and will then be at the beginning). Both the simple line (See Section 3.2.) and the standard line (See Section 3.1.) provide two different kinds of cursors, namely *left-sticky* and *right-sticky* cursors that behave differently in this situations in that left-sticky cursors remain in the initial line and right-sticky cursors will be attached to the newly-created line.

⇒ `cluffer-internal:line-join-line` *(line1 new-line:line) line2* [*Method*]

This method should join *line1* and *line2* by adding the items of *line2* to the end of *line1*, detaching any cursors attached to *line2*, and attaching those cursors to *line1* in the appropriate position.

4.3 Writing new buffer implementations

4.3.1 Package

It is generally a good idea to define a new package for a new implementation of the concept of a buffer. For the remainder of this section, we use the name `new-buffer` for this package.

4.3.2 Classes

A class that is a subclass of `cluffer:buffer` must be provided. In the remainder of this section, we refer to this class as `new-buffer:buffer`.

A class that is a subclass of `cluffer-internal:dock` must be provided. The purpose of this class is to serve as an intermediate between a line and the buffer in which the line is contained. In the remainder of this section, we refer to this class as `new-buffer:dock`.

4.3.3 Methods

⇒ `line-count` *(buffer new-buffer:buffer)* [*Method*]

This method should return the number of lines in *buffer*.

⇒ `item-count` *(buffer new-buffer:buffer)* [*Method*]

This method should return the number of items in *buffer*. The number of items in the buffer is defined to be the sum of the number of items in each line of the buffer.

⇒ `find-line` *(buffer new-buffer:buffer) line-number* [*Method*]

This method should return the line with the given *line-number* in *buffer*.

There is no need to check that *line-number* is valid. This check is already taken care of by an auxiliary method on `line-number`.

⇒ `cluffer-internal:buffer-line-number`
(buffer new-buffer:buffer) (dock new-buffer:dock) line [*Method*]

This method is part of the internal protocol for communicating between the line implementation and the buffer implementation.

It should return the line number if *line* in *buffer*. The parameter *dock* is the dock to which the line is attached.

Implementations can choose to ignore either the *buffer* or the *dock* parameter, depending on how the buffer is represented.

⇒ `cluffer-internal:buffer-split-line`
(buffer new-buffer:buffer) (dock new-buffer:dock) (line cluffer:line) position [*Method*]

This method is part of the internal protocol for communicating between the line implementation and the buffer implementation.

It must call `cluffer-internal:line-split-line` with *line* and *position* in order to obtain a new line to insert into its buffer representation, and it must take into account that *line* now has fewer items in it as indicated by *position*.

Implementations can choose to ignore either the *buffer* or the *dock* parameter, depending on how the buffer is represented.

⇒ `cluffer-internal:buffer-join-line`
(buffer new-buffer:buffer) (dock new-buffer:dock) (line cluffer:line) [*Method*]

This method is part of the internal protocol for communicating between the line implementation and the buffer implementation.

It must call `cluffer-internal:line-join-line`, passing it *line* and the line following *line* in *buffer*, and it must eliminate the line following *line* and take into account that the items in it will be appended to *line*.

Implementations can choose to ignore either the *buffer* or the *dock* parameter, depending on how the buffer is represented.

⇒ `cluffer:update`
(buffer new-buffer:buffer) time sync skip modify create [*Method*]

This method is part of the update protocol. It should implement the update protocol as described in Section 2.5.

Bibliography

[ST85] Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *J. ACM*, 32(3):652–686, July 1985.