Drei Manual

Table of Contents

Drei - an acronym for *Drei Replaces EINE's Inheritor* - is one of the editor substrates provided by McCLIM. Drei is activated by default.

1 Drei Concepts

The reason for many of Drei's design decisions, and the complexity of some of the code, is due to the flexibility that Drei is meant to expose. Drei has to work as, at least, an input-editor, a text editor gadget and a simple pane. These three different uses have widely different semantics for reading input and performing redisplay - from passively being fed gestures in the input editor, to having to do event handling and redisplay timing manually in the gadget version. Furthermore, Drei is extensible software, so we wished to make the differences between these three modi operandi transparent to the extender (as much as possible at least, unfortunately the Law of Leaky Abstractions prevents us from reaching perfection). These two demands require the core Drei protocols, especially those pertaining to redisplay, gesture handling and accepting input from the user, to be customizable by the different specialized Drei classes.

We call a specific instance of the Drei editor substrate a *Drei instance*. A *Drei variant* is a specific subclass of `drei` that implements a specific kind of editor, such as an input-editor or a gadget. A given Drei instance has a single view associated with it, this view must be unique to the Drei instance (though this is not enforced), but may be changed at any time. The most typical view is one that has a buffer and maintains syntax information about the buffer contents. A buffer need not be unique to a buffer-view, and may be changed at any time. The view instance has two marks into the buffer, called the *top* and *bottom* mark. These marks delimit the visible region of the buffer - for some Drei variants, this is always the entire buffer, while others may only have a smaller visible region. Note that not all of the visible region necessarily is on display on the screen (parts, or all, of it may be hidden due to scrolling, for example), but nothing outside the visible region is on display, though remember that the same buffer may be used in several vires, and that each of these views may have their own idea about what the visible region is. Most views also maintain marks for the current *point* and *mark*. This means that different views sharing the same buffer may have different points and marks. Every Drei instance also has a *kill ring* object which contains object sequences that have been killed from the buffer, and can be yanked back in at the users behest. These are generally not shared.

Every Drei instance is associated with an editor pane - this must be a CLIM stream pane that is used for redisplay (see Section 4.7 [Redisplay Protocol], page 23). This is not necessarily the same object as the Drei instance itself, but it can be. (With a little work, the editor pane can be `NIL`, which is useful for resting.)

For each Drei instance, Drei attempts to simulate an application top-level loop with something called a *pseudo command loop*, and binds a number of special variables appropriately. This is to make command writing more convenient and similar across all Drei variants, but it also means that any program that uses one of the low-level Drei variants that do not to this, such as `drei-pane`, need to bind these special variables themselves, or Drei commands are likely to malfunction.

1.1 Access Functions

The access functions are the primary interface to Drei state, and should be used to access the various parts. It is not recommended to save the return value of these functions, as they are by nature ephemeral, and may change over the course of a command.

`drei:drei-instance` **&optional** *object* [Function]
Return the Drei instance of `object`. If `object` is not provided, the currently running Drei instance will be returned.

`drei:current-view` **&optional** *object* [Function]
Return the view of the provided object. If no object is provided, the currently running Drei instance will be used.

`esa:current-buffer` [Function]
Return the currently active buffer of the running `esa`.

`drei:point` **&optional** *object* [Function]
Return the point of the provided object. If no object is provided, the current view will be used.

`drei:mark` **&optional** *object* [Function]
Return the mark of the provided object. If no object is provided, the current view will be used.

`drei:current-syntax` [Function]
Return the syntax of the current buffer.

1.2 Special Variables

Drei uses only a few special variables to provide access to data structures.

`drei-kill-ring:*kill-ring*` [Variable]
This special variable is bound to the kill ring of the running application or Drei instance whenever a command is executed.

Additionally, a number of ESA special variables are used in Drei.

`esa:*minibuffer*` [Variable]
The minibuffer pane of the running application.

`esa:*previous-command*` [Variable]
When a command is being executed, the command previously executed by the application.

2 External API

`drei:drei` [Class]

Class precedence list: `drei, standard-object, slot-object, t`

Slots:

- `%view` — initargs: `:view`

  The CLIM view that will be used whenever this Drei is being displayed. During redisplay, the `stream-default-view` of the output stream will be temporarily bound to this value.

- `%previous-command`

  The previous CLIM command executed by this Drei instance. May be NIL if no command has been executed.

- `%editor-pane` — initargs: `:editor-pane`

  The stream or pane that the Drei instance will perform output to.

- `%minibuffer` — initargs: `:minibuffer`

  The minibuffer pane (or null) associated with the Drei instance. This may be NIL.

- `%command-table` — initargs: `:command-table`

  The command table used for looking up commands for the Drei instance. Has a sensible default, don't override it unless you know what you are doing.

- `%cursors`

  A list of which cursors are associated with the Drei instance. During redisplay, `display-drei-view-cursor` is called on each element of this list.

- `%point-cursor`

  The cursor object that is considered the primary user-oriented cursor, most probably the cursor for the editor point. Note that this cursor is also in the cursors-list.

- `%cursors-visible` — initargs: `:cursors-visible`

  If true, the cursors of this Drei instance will be visible. If false, they will not.

The abstract Drei class that maintains standard Drei editor state. It should not be directly instantiated, a subclass implementing specific behavior (a Drei variant) should be used instead.

`:editable-p` [`drei` Initarg]

Whether or not the Drei instance will be editable. If `NIL`, the buffer will be set to read-only (this also affects programmatic access). The default is `T`.

`:single-line` [`drei` Initarg]

If `T`, the buffer created for the Drei instance will be single line, and a condition of type `buffer-single-line` will be signalled if an attempt is made to insert a newline character.

`drei:handling-drei-conditions` **&body** *body* [Macro]

Evaluate `body` while handling Drei user notification signals. The handling consists of displaying their meaning to the user in the minibuffer. This is the macro that ensures conditions such as motion-before-end does not land the user in the debugger.

`drei:with-bound-drei-special-variables` (*drei-instance* **&key** *kill-ring minibuffer command-parser partial-command-parser previous-command prompt*) **&body** *body* [Macro]

Evaluate `body` with a set of Drei special variables ((drei-instance), `*kill-ring*`, `*minibuffer*`, `*command-parser*`, `*partial-command-parser*`, `*previous-command*`, `*extended-command-prompt*`) bound to their proper values, taken from `drei-instance`. The keyword arguments can be used to provide forms that will be used to obtain values for the respective special variables, instead of finding their value in `drei-instance` or their existing binding. This macro binds all of the usual Drei special variables, but also some CLIM special variables needed for ESA-style command parsing.

`drei:performing-drei-operations` (*drei* **&rest** *args* **&key** *with-undo redisplay*) **&body** *body* [Macro]

Provide various Drei maintenance services around the evaluation of `body`. This macro provides a convenient way to perform some operations on a Drei, and make sure that they are properly reflected in the undo tree, that the Drei is redisplayed, the syntax updated, etc. Exactly what is done can be controlled via the keyword arguments. Note that if `with-undo` is false, the *entire* undo history will be cleared after `body` has been evaluated. This macro expands into a call to `invoke-performing-drei-operations`.

`drei:invoke-performing-drei-operations` *drei continuation* **&key** *with-undo redisplay* [Generic Function]

Invoke `continuation`, setting up and performing the operations specified by the keyword arguments for the given Drei instance.

`drei:accepting-from-user` (*drei*) **&body** *body* [Macro]

Modidfy `drei` and the environment so that calls to `accept` can be done to arbitrary streams from within `body`. Or, at least, make sure the Drei instance will not be a problem. When Drei calls a command, it will be wrapped in this macro, so it should be safe to use `accept` within Drei commands. This macro expands into a call to `invoke-accepting-from-user`.

`drei:invoke-accepting-from-user` *drei continuation* [Generic Function]

Set up `drei` and the environment so that calls to `accept` will behave properly. Then call `continuation`.

`drei:execute-drei-command` *drei-instance command* [Generic Function]

Execute `command` for `drei`. This is the standard function for executing Drei commands – it will take care of reporting to the user if a condition is signalled, updating the syntax, setting the `previous-command` of `drei` and recording the operations performed by `command` for undo.

3 Standard Drei Variants

Because the standard `drei` class doesn't implement immediately-usable editor behavior, three subclasses have been defined to provide a concrete implementaton of the editor substrate. These are the input-editor-oriented Drei variant, the pane-oriented Drei variant and the gadget-oriented Drei variant.

4 Protocols

Much of Drei's functionality is based on generic function protocols. This section lists some of them.

4.1 Buffer Protocol

The Drei buffer is what holds textual and other objects to be edited and displayed. Conceptually, the buffer is a potentially large sequence of objects, most of which are expected to be characters (the full Unicode character set is supported). However, Drei buffers can contain any Common Lisp objects, as long as the redisplay engine knows how to render them.

The Drei buffer implementation differs from that of a vector, because it allows for very efficient editing operations, such as inserting and removing objects at arbitrary offsets.

In addition, the Drei buffer protocols defines that concept of a mark.

4.1.1 General Buffer Protocol Parts

`drei-buffer:buffer` [Class]

Class precedence list: `buffer, standard-object, slot-object, t`

The base class for all buffers. A buffer conceptually contains a large array of arbitrary objects. Lines of objects are separated by newline characters. The last object of the buffer is not necessarily a newline character.

`drei-buffer:standard-buffer` [Class]

Class precedence list: `standard-buffer, buffer, standard-object, slot-object, t`

The standard instantiable class for buffers.

`drei-buffer:mark` [Class]

Class precedence list: `mark, standard-object, slot-object, t`

The base class for all marks.

`drei-buffer:left-sticky-mark` [Class]

Class precedence list: `left-sticky-mark, mark, standard-object, slot-object, t`

A subclass of mark. A mark of this type will "stick" to the left of an object, i.e. when an object is inserted at this mark, the mark will be positioned to the left of the object.

`drei-buffer:right-sticky-mark` [Class]

Class precedence list: `right-sticky-mark, mark, standard-object, slot-object, t`

A subclass of mark. A mark of this type will "stick" to the right of an object, i.e. when an object is inserted at this mark, the mark will be positioned to the right of the object.

`drei-buffer:offset` *mark* [Generic Function]
Return the offset of the mark into the buffer.

`(setf drei-buffer:offset)` *new-offset mark* [Generic Function]
Set the offset of the mark into the buffer. A motion-before-beginning condition is signaled if the offset is less than zero. A motion-after-end condition is signaled if the offset is greater than the size of the buffer.

`drei-buffer:clone-mark` *mark* **&optional** *stick-to* [Generic Function]
Clone a mark. By default (when stick-to is NIL) the same type of mark is returned. Otherwise stick-to is either :left or :right indicating whether a left-sticky or a right-sticky mark should be created.

`drei-buffer:buffer` *mark* [Generic Function]
Return the buffer that the mark is positioned in.

`drei-buffer:no-such-offset` [Condition]
Class precedence list: `no-such-offset, error, serious-condition, condition, slot-object, t`
This condition is signaled whenever an attempt is made to access buffer contents that is before the beginning or after the end of the buffer.

`drei-buffer:offset-before-beginning` [Condition]
Class precedence list: `offset-before-beginning, no-such-offset, error, serious-condition, condition, slot-object, t`
This condition is signaled whenever an attempt is made to access buffer contents that is before the beginning of the buffer.

`drei-buffer:offset-after-end` [Condition]
Class precedence list: `offset-after-end, no-such-offset, error, serious-condition, condition, slot-object, t`
This condition is signaled whenever an attempt is made to access buffer contents that is after the end of the buffer.

`drei-buffer:invalid-motion` [Condition]
Class precedence list: `invalid-motion, error, serious-condition, condition, slot-object, t`
This condition is signaled whenever an attempt is made to move a mark before the beginning or after the end of the buffer.

`drei-buffer:motion-before-beginning` [Condition]
Class precedence list: `motion-before-beginning, invalid-motion, error, serious-condition, condition, slot-object, t`
This condition is signaled whenever an attempt is made to move a mark before the beginning of the buffer.

`drei-buffer:motion-after-end` [Condition]
Class precedence list: `motion-after-end, invalid-motion, error, serious-condition, condition, slot-object, t`
This condition is signaled whenever an attempt is made to move a mark after the end of the buffer.

`drei-buffer:size` *buffer* [Generic Function]
Return the number of objects in the buffer.

`drei-buffer:number-of-lines` *buffer* [Generic Function]
Return the number of lines of the buffer, or really the number of newline characters.

4.1.2 Operations Related To The Offset Of Marks

`drei-buffer:forward-object` *mark* **&optional** *count* [Generic Function]
Move the mark forward the number of positions indicated by count. This function could be implemented by an `incf` on the offset of the mark, but many buffer implementations can implement this function much more efficiently in a different way. A `motion-before-beginning` condition is signaled if the resulting offset of the mark is less than zero. A `motion-after-end` condition is signaled if the resulting offset of the mark is greater than the size of the buffer. Returns `mark`.

`drei-buffer:backward-object` *mark* **&optional** *count* [Generic Function]
Move the mark backward the number of positions indicated by count. This function could be implemented by a `decf` on the offset of the mark, but many buffer implementations can implement this function much more efficiently in a different way. A `motion-before-beginning` condition is signaled if the resulting offset of the mark is less than zero. A `motion-after-end` condition is signaled if the resulting offset of the mark is greater than the size of the buffer. Returns `mark`.

`drei-buffer:mark=` *mark1 mark2* [Generic Function]
Return `t` if the offset of `mark1` is equal to that of `mark2`. An error is signaled if the two marks are not positioned in the same buffer. It is acceptable to pass an offset in place of one of the marks.

`drei-buffer:mark<` *mark1 mark2* [Generic Function]
Return `t` if the offset of `mark1` is strictly less than that of `mark2`. An error is signaled if the two marks are not positioned in the same buffer. It is acceptable to pass an offset in place of one of the marks.

`drei-buffer:mark<=` *mark1 mark2* [Generic Function]
Return `t` if the offset of `mark1` is less than or equal to that of `mark2`. An error is signaled if the two marks are not positioned in the same buffer. It is acceptable to pass an offset in place of one of the marks.

`drei-buffer:mark>` *mark1 mark2* [Generic Function]
Return `t` if the offset of `mark1` is strictly greater than that of `mark2`. An error is signaled if the two marks are not positioned in the same buffer. It is acceptable to pass an offset in place of one of the marks.

`drei-buffer:mark>=` *mark1 mark2* [Generic Function]
Return `t` if the offset of `mark1` is greater than or equal to that of `mark2`. An error is signaled if the two marks are not positioned in the same buffer. It is acceptable to pass an offset in place of one of the marks.

`drei-buffer:beginning-of-buffer` *mark* [Generic Function]
Move the mark to the beginning of the buffer. This is equivalent to (setf (offset mark) 0), but returns mark.

`drei-buffer:end-of-buffer` *mark* [Generic Function]
Move the mark to the end of the buffer and return mark.

`drei-buffer:beginning-of-buffer-p` *mark* [Generic Function]
Return `t` if the mark is at the beginning of the buffer, nil otherwise.

`drei-buffer:end-of-buffer-p` *mark* [Generic Function]
Return `t` if the mark is at the end of the buffer, NIL otherwise.

`drei-buffer:beginning-of-line` *mark* [Generic Function]
Move the mark to the beginning of the line. The mark will be positioned either immediately after the closest receding newline character, or at the beginning of the buffer if no preceding newline character exists. Returns `mark`.

`drei-buffer:end-of-line` *mark* [Generic Function]
Move the mark to the end of the line. The mark will be positioned either immediately before the closest following newline character, or at the end of the buffer if no following newline character exists. Returns `mark`.

`drei-buffer:beginning-of-line-p` *mark* [Generic Function]
Return `t` if the mark is at the beginning of the line (i.e., if the character preceding the mark is a newline character or if the mark is at the beginning of the buffer), NIL otherwise.

`drei-buffer:end-of-line-p` *mark* [Generic Function]
Return `t` if the mark is at the end of the line (i.e., if the character following the mark is a newline character, or if the mark is at the end of the buffer), NIL otherwise.

`drei-buffer:buffer-line-number` *buffer offset* [Generic Function]
Return the line number of the offset. Lines are numbered from zero.

`drei-buffer:buffer-column-number` *buffer offset* [Generic Function]
Return the column number of the offset. The column number of an offset is the number of objects between it and the preceding newline, or between it and the beginning of the buffer if the offset is on the first line of the buffer.

`drei-buffer:line-number` *mark* [Generic Function]
Return the line number of the mark. Lines are numbered from zero.

`drei-buffer:column-number` *mark* [Generic Function]
Return the column number of the mark. The column number of a mark is the number of objects between it and the preceding newline, or between it and the beginning of the buffer if the mark is on the first line of the buffer.

4.1.3 Inserting And Deleting Objects

`drei-buffer:insert-buffer-object` *buffer offset object* [Generic Function]
Insert the object at the offset in the buffer. Any left-sticky marks that are placed at the offset will remain positioned before the inserted object. Any right-sticky marks that are placed at the offset will be positioned after the inserted object.

`drei-buffer:insert-buffer-sequence` *buffer offset sequence* [Generic Function]
Like calling insert-buffer-object on each of the objects in the sequence.

`drei-buffer:insert-object` *mark object* [Generic Function]
Insert the object at the mark. This function simply calls insert-buffer-object with the buffer and the position of the mark.

`drei-buffer:insert-sequence` *mark sequence* [Generic Function]
Insert the objects in the sequence at the mark. This function simply calls insert-buffer-sequence with the buffer and the position of the mark.

`drei-buffer:delete-buffer-range` *buffer offset n* [Generic Function]
Delete n objects from the buffer starting at the offset. If `offset` is negative or `offset+n` is greater than the size of the buffer, a `no-such-offset` condition is signaled.

`drei-buffer:delete-range` *mark* **&optional** *n* [Generic Function]
Delete `n` objects after (if n > 0) or before (if n < 0) the mark. This function eventually calls delete-buffer-range, provided that `n` is not zero.

`drei-buffer:delete-region` *mark1 mark2* [Generic Function]
Delete the objects in the buffer that are between `mark1` and `mark2`. An error is signaled if the two marks are positioned in different buffers. It is acceptable to pass an offset in place of one of the marks. This function calls `delete-buffer-range` with the appropriate arguments.

4.1.4 Getting Objects Out Of The Buffer

`drei-buffer:buffer-object` *buffer offset* [Generic Function]
Return the object at the offset in the buffer. The first object has offset 0. If `offset` is less than zero or greater than or equal to the size of the buffer, a `no-such-offset` condition is signaled.

`(setf drei-buffer:buffer-object)` *object buffer offset* [Generic Function]
Set the object at the offset in the buffer. The first object has offset 0. If `offset` is less than zero or greater than or equal to the size of the buffer, a `no-such-offset` condition is signaled.

`drei-buffer:buffer-sequence` *buffer offset1 offset2* [Generic Function]
Return the contents of the buffer starting at `offset1` and ending at offset2-1 as a sequence. If either of the offsets is less than zero or greater than or equal to the size of the buffer, a `no-such-offset` condition is signaled. If `offset2` is smaller than or equal to `offset1`, an empty sequence will be returned.

`drei-buffer:object-before` *mark* [Generic Function]
Return the object that is immediately before the mark. If mark is at the beginning of the buffer, a `no-such-offset` condition is signaled. If the mark is at the beginning of a line, but not at the beginning of the buffer, a newline character is returned.

`drei-buffer:object-after` *mark* [Generic Function]
Return the object that is immediately after the mark. If mark is at the end of the buffer, a `no-such-offset` condition is signaled. If the mark is at the end of a line, but not at the end of the buffer, a newline character is returned.

`drei-buffer:region-to-sequence` *mark1 mark2* [Generic Function]
Return a freshly allocated sequence of the objects after `mark1` and before `mark2`. An error is signaled if the two marks are positioned in different buffers. If mark1 is positioned at an offset equal to or greater than that of `mark2`, an empty sequence is returned. It is acceptable to pass an offset in place of one of the marks. This function calls `buffer-sequence` with the appropriate arguments.

4.1.5 Implementation Hints

The buffer is implemented as lines organized in a 2-3-tree. The leaves of the tree contain the lines, and the internal nodes contain additional information of the left subtree (if it is a 2-node) or the left and the middle subtree (if it is a 3-node). Two pieces of information are stored: The number of lines in up to and including the subtree and the total number of objects up to an including the subtree. This organization allows us to determine, the line number and object position of any mark in O(log N) where N is the number of lines.

A line is an instance of the 'buffer-line' class. A line can either be open or closed. A closed line is represented as a sequence. The exact type of the sequence depends on the objects contained in the line. If the line contains only characters of type base-char, then the sequence is of type base-string. If the line contains only characters, but not of type base-char, the sequence is a string. Otherwise it is a vector of arbitrary objects. This way, closed lines containing characters with code points below 256 have a compact representation with 8 bits per character while still allowing for arbitrary objects when necessary. An open line is represented as a cursorchain of objects.

Marks in a closed line are represented as an integer offset into the sequence. Marks in an open line are represented as flexicursors.

When a line is opened, it is converted to a cursorchain. When a line is closed, it is examined to determine whether it contains non-character objects, in which case it is converted to a vector of objects. If contains only characters, but it contains characters with code points above what can be represented in a base-char, it is converted to a string. If it contains only base-chars, it is converted to a base-string.

A mark contains two slots: a flexicursor that determines which line it is on, and either an integer (if the line is closed) that determines the offset within the line or another flexicursor (if the line is open). For each line, open or closed, a list of weak references to marks into that line is kept.

Lines are closed according to a LRU scheme. Whenever objects are inserted to or deleted from a line, it becomes the most recently used line. We keep a fixed number of open lines so that when a line is opened and the threshold is reached, the least recently used line is closed.

4.2 Buffer Modification Protocol

The buffer modification protocol is based on the ESA observer/observable facility, which is in return a fairly ordinary Model-View implementation.

`drei-buffer:observable-buffer-mixin` [Class]

Class precedence list: `observable-buffer-mixin, observable-mixin, standard-object, slot-object, t`

A mixin class that will make a subclass buffer notify observers when it is changed through the buffer protocol. When an observer of the buffer is notified of changes, the provided data will be a cons of two values, offsets into the buffer denoting the region that has been modified.

Syntax-views use this information to determine what part of the buffer needs to be reparsed. This automatically happens whenever a request is made for information that might depend on outdated parsing data.

4.3 DREI-BASE Package

The buffer protocol has been designed to be reasonably efficient with a variety of different implementation strategies (single gap buffer or sequence of independent lines). It contains (and should only contain) the absolute minimum of functionality that can be implemented efficiently independently of strategy. However, this minimum of functionality is not always convenient.

The purpose of the `DREI-BASE` package is to implement additional functionality on top of the buffer protocol, in a way that does not depend on how the buffer protocol was implemented. Thus, the `DREI-BASE` package should remain intact across different implementation strategies of the buffer protocol.

Achieving portability of the `DREI-BASE` package is not terribly hard as long as only buffer protocol functions are used. What is slightly harder is to be sure to maximize efficiency across several implementation strategies. The next section discusses such considerations and gives guidelines to implementers of additional functionality.

Implementers of the buffer protocol may use the contents of the next section to make sure they respect the efficiency considerations that are expected by the `DREI-BASE` package.

4.3.1 Efficiency considerations

In this section, we give a list of rules that implementors of additional functionality should follow in order to make sure that such functionality remains efficient (in addition to being portable) across a variety of implementation strategies of the buffer protocol.

Rule: Comparing the position of two marks is efficient, i.e. at most O(log n) where n is the number of marks in the buffer (which is expected to be very small compared to the number of objects) in all implementations. This is true for all types of comparisons.

It is expected that marks are managed very efficiently. Some balanced tree management might be necessary, which will make operations have logarithmic complexity, but only in the number of marks that are actually used.

Rule: While computing and setting the offset of a mark is fairly efficient, it is not guaranteed to be O(1) even though it might be in an implementation using a single gap buffer. It might have a complexity of O(log n) where n is the number of lines in the buffer. This is true for using incf on the offset of a mark as well, as incf expands to a setf of the offset.

Do not hesitate computing or setting the offset of a mark, but avoid doing it in a tight loop over many objects of the buffer.

Rule: Determining whether a mark is at the beginning or at the end of the buffer is efficient, i.e. O(1), in all implementations.

Rule: Determining whether a mark is at the beginning or at the end of a line is efficient, i.e. O(1), in all implementations.

Rule: Going to the beginning or to the end of a line might have linear-time complexity in the number of characters of the line, though it is constant-time complexity if the implementation is line oriented.

It is sometimes inevitable to use this functionality, and since lines are expected to be short, it should not be avoided at all cost, especially since it might be very efficient in some implementations. We do recommend, however to avoid it in tight loops.

Always use this functionality rather than manually incrementing the offset of a mark in a loop until a Newline character has been found, especially since each iteration might take logarithmic time then.

Rule: Computing the size of the buffer is always efficient, i.e., O(1).

Rule: Computing the number of lines of the buffer is always efficient, i.e., O(1).

Implementations of the buffer protocol could always track the number of insertions and deletions of objects, so there is no reason why this operation should be inefficient.

Rule: Computing the line number of a mark or of an offset can be very costly, i.e. O(n) where n is size of the buffer.

This operation is part of the buffer protocol because some implementations may implement it fairly efficiently, say O(log n) where n is the number of lines in the buffer.

4.4 Syntax Protocol

A syntax module is an object that can be associated with a buffer. The syntax module usually consists of an incremental parser that analyzes the contents of the buffer and creates some kind of parse tree or other representation of the contents in order that it can be exploited by the redisplay module and by user commands.

4.4.1 General Syntax Protocol

`drei-syntax:syntax` [Class]

Class precedence list: `syntax`, `name-mixin`, `observable-mixin`, `standard-object`, `slot-object`, `t`

Slots:

- `%updater-fns` — initargs: `:updater-fns`

  A list of functions that are called whenever a syntax function needs up-to-date syntax information. `update-syntax` is never called directly by syntax commands.

Each function should take two arguments, integer offsets into the buffer of the syntax delimiting the region that must have an up-to-date parse. These arguments should be passed on to a call to `update-syntax`.

The base class for all syntaxes.

The redisplay module exploits the syntax module for several things:

- highlighting of various syntactic entities of the buffer
- highlighting of matching parenthesis,
- turning syntactic entities into clickable presentations,
- marking lines with inconsistent indentation,
- etc.

User commands can use the syntax module for:

- moving point by units that are specific to a particular buffer syntax, such as expressions, statements, or paragraphs,
- transposing syntactic units,
- sending the text of a syntactic unit to a language processor,
- indenting lines according to the syntax,
- etc.

The ideal is that the view that the syntax module has of the buffer is updated only when needed, and then only for the parts of the buffer that are needed, though implementing this in practise is decidedly nontrivial. Most syntax modules (such as for programming languages) need to compute their representations from the beginning of the buffer up to a particular point beyond which the structure of the buffer does not need to be known.

There are two primary situations where updating might be needed:

- Before redisplay is about to show the contents of part of the buffer in a pane, to inform the syntax module that its syntax must be valid in the particular region on display,
- as a result of a command that exploits the syntactic entities of the buffer contents.

These two cases do boil down to "whenever there is need for the syntax information to be correct", however.

The first case is handled by the invocation of a single generic function:

`drei-syntax:update-syntax` *syntax unchanged-prefix unchanged-suffix* **&optional** *begin end* [Generic Function]

Method combination: `VALUES-MAX-MIN` (most-specific-last)

Inform the syntax module that it must update its view of the buffer. `unchanged-prefix unchanged-suffix` indicate what parts of the buffer has not been changed. `begin` and `end` are offsets specifying the minimum region of the buffer that must have an up-to-date parse, defaulting to 0 and the size of the buffer respectively. It is perfectly valid for a syntax to ignore these hints and just make sure the entire syntax tree is up to date, but it *must* make sure at at least the region delimited by `begin` and `end` has an up to date parse. Returns two values, offsets into the buffer of the syntax, denoting the buffer region thas has an up to date parse.

It is important to realize that the syntax module is not directly involved in displaying buffer contents in a pane. In fact, the syntax module should work even if there is no graphic user interface present, and it should be exploitable by several, potentially totally different, display units.

The second case is slightly trickier, as any views of the syntax should be informed that it has reparsed some part of the buffer. Since `update-syntax` is only called by views, the view can easily record the fact that some part of the buffer has an up-to-date parse. Thus, functions accessing syntax information must go to some length to make sure that the view of the syntax is notified of any reparses.

`drei-syntax:update-parse` *syntax* **&optional** *begin end* [Function]
Make sure the parse for `syntax` from offset `begin` to `end` is up to date. `begin` and `end` default to 0 and the size of the buffer of `syntax`, respectively.

4.4.2 Incremental Parsing Framework

`drei-syntax:parse-tree` [Class]
Class precedence list: `parse-tree, standard-object, slot-object, t`

The base class for all parse trees.

We use the term parse tree in a wider sense than what is common in the parsing literature, in that a lexeme is a (trivial) parse tree. The parser does not distinguish between lexemes and other parse trees, and a grammar rule can produce a lexeme if that should be desired.

`drei-syntax:start-offset` *parse-tree* [Generic Function]
The offset in the buffer of the first character of a parse tree.

`drei-syntax:end-offset` *parse-tree* [Generic Function]
The offset in the buffer of the character following the last one of a parse tree.

The length of a parse-tree is thus the difference of its end offset and its start offset.

The start offset and the end offset may be NIL which is typically the case when a parse tree is derived from the empty sequence of lexemes.

4.4.3 Lexical analysis

`drei-syntax:lexer` [Class]
Class precedence list: `lexer, standard-object, slot-object, t`

Slots:

- `buffer` — initargs: `:buffer`

  The buffer associated with the lexer.

The base class for all lexers.

`drei-syntax:incremental-lexer` [Class]
Class precedence list: `incremental-lexer, lexer, standard-object, slot-object, t`

A subclass of lexer which maintains the buffer in the form of a sequence of lexemes that is updated incrementally.

In the sequence of lexemes maintained by the incremental lexer, the lexemes are indexed by a position starting from zero.

`drei-syntax:nb-lexemes` *lexer* [Generic Function]
Return the number of lexemes in the lexer.

`drei-syntax:lexeme` *lexer pos* [Generic Function]
Given a lexer and a position, return the lexeme in that position in the lexer.

`drei-syntax:insert-lexeme` *lexer pos lexeme* [Generic Function]
Insert a lexeme at the position in the lexer. All lexemes following `pos` are moved to one position higher.

`drei-syntax:delete-invalid-lexemes` *lexer from to* [Generic Function]
Invalidate all lexemes that could have changed as a result of modifications to the buffer

`drei-syntax:inter-lexeme-object-p` *lexer object* [Generic Function]
This generic function is called by the incremental lexer to determine whether a buffer object is an inter-lexeme object, typically whitespace. Client code must supply a method for this generic function.

`drei-syntax:skip-inter-lexeme-objects` *lexer scan* [Generic Function]
This generic function is called by the incremental lexer to skip inter-lexeme buffer objects. The default method for this generic function increments the scan mark until the object after the mark is not an inter-lexeme object, or until the end of the buffer has been reached.

`drei-syntax:update-lex` *lexer start-pos end* [Generic Function]
This function is called by client code as part of the buffer-update protocol to inform the lexer that it needs to analyze the contents of the buffer at least up to the `end` mark of the buffer. `start-pos` is the position in the lexeme sequence at which new lexemes should be inserted.

`drei-syntax:next-lexeme` *lexer scan* [Generic Function]
This generic function is called by the incremental lexer to get a new lexeme from the buffer. Client code must supply a method for this function that specializes on the lexer class. It is guaranteed that scan is not at the end of the buffer, and that the first object after scan is not an inter-lexeme object. Thus, a lexeme should always be returned by this function.

4.4.4 Earley Parser

Drei contains an incremental parser that uses the Earley algorithm. This algorithm accepts the full set of context-free grammars, allowing greater freedom for the developer to define natural grammars without having to think about restrictions such as LL(k) or LALR(k).

Beware, though, that the Earley algorithm can be quite inefficient if the grammar is sufficiently complicated, in particular if the grammar is ambiguous.

4.4.5 Specifying A Grammar

An incremental parser is created from a grammar.

`drei-syntax:grammar` **&body** *body* [Macro]
Create a grammar object from a set of rules.

symbol `->` (*&rest arguments*) *&optional body* [Rule]
Each rule is a list of this form.

Here *symbol* is the target symbol of the rule, and should be the name of a CLOS class.

(*var type test*) [Rule argument]
The most general form of a rule argument.

Here *var* is the name of a lexical variable. The scope of the variable contains the test, all the following arguments and the body of the rule. The *type* is a Common Lisp type specification. The rule applies only of the *type* of the object contain in var is of that type. The *test* contains arbitrary Common Lisp code for additional checks as to the applicability of the rule.

(*var type*) [Rule argument]
Abbreviated form of a rule argument.

Here, *type* must be a symbol typically the name of a CLOS class. This form is an abbreviation for `(var type t)`.

(*var test*) [Rule argument]
Abbreviated form of a rule argument.

Here, *test* must not be a symbol. This form is an abbreviation of `(var var test)`, i.e., the name of the variable is also the name of a type, typically a CLOS class.

var [Rule argument]
Abbreviated form of a rule argument.

This form is an abbreviation of `(var var t)`.

The *body* of a rule, if present, contains an expression that should have an instance (not necessarily direct) of the class named by the symbol (the left-hand-side) of the rule. It is important that this restriction be respected, since the Earley algorithm will not work otherwise.

If the *body* is absent, it is the same as if a body of the form `(make-instance 'symbol)` had been given.

The body can also be a sequence of forms, the first one of which must be a symbol. These forms typically contain initargs, and will be passed as additional arguments to `(make-instance 'symbol)`.

4.5 View Protocol

Drei extends CLIMs concept of "views" to be more than just a manner for determining the user interface for accepting values from the user. Instead, the view is what controls the user interface of the Drei instance the user is interacting with. To simplify the discussion, this section assumes that the view is always associated with a single buffer. A buffer does not have to be associated with a view, and may be associated with many views, though each view may only have a single buffer. The view controls how the buffer is displayed to the user, and which commands are available to the user for modifying the buffer. A view may use a syntax module to maintain syntactical information about the buffer contents, and use the resulting information to highlight parts of the buffer based on its syntactical value ("syntax highlighting").

`drei:drei-view` [Class]

Class precedence list: `drei-view, tabify-mixin, subscriptable-name-mixin, name-mixin, standard-object, slot-object, t`

Slots:

- `%active` — initargs: `:active`

  A boolean value indicating whether the view is "active". This should control highlighting when redisplaying.

- `%modified-p` — initargs: `:modified-p`

  This value is true if the view contents have been modified since the last time this value was set to false.

- `%no-cursors` — initargs: `:no-cursors`

  True if the view does not display cursors.

- `%full-redisplay-p`

  True if the view should be fully redisplayed the next time it is redisplayed.

- `%use-editor-commands` — initargs: `:use-editor-commands`

  If the view is supposed to support standard editor commands (for inserting objects, moving cursor, etc), this will be true. If you want your view to support standard editor commands, you should *not* inherit from `editor-table` - the command tables containing the editor commands will be added automatically, as long as this value is true.

- `%extend-pane-bottom` — initargs: `:extend-pane-bottom`

  Resize the output pane vertically during redisplay (using `change-space-requirements`), in order to fit the whole buffer. If this value is false, redisplay will stop when the bottom of the pane is reached.

The base class for all Drei views. A view observes some other object and provides a visual representation for Drei.

`drei:drei-buffer-view` [Class]

Class precedence list: `drei-buffer-view, drei-view, tabify-mixin, subscriptable-name-mixin, name-mixin, standard-object, slot-object, t`

Slots:

- `%buffer` — initargs: `:buffer`
  The buffer that is observed by this buffer view.
- `%top`
  The top of the displayed buffer, that is, the mark indicating the first visible object in the buffer.
- `%bot`
  The bottom of the displayed buffer, that is, the mark indicating the last visible object in the buffer.
- `%cache-string`
  A string used during redisplay to reduce consing. Instead of consing up a new string every time we need to pull out a buffer region, we put it in this string. The fill pointer is automatically set to zero whenever the string is accessed through the reader.
- `%displayed-lines`
  An array of the `displayed-line` objects displayed by the view. Not all of these are live.
- `%displayed-lines-count`
  The number of lines in the views `displayed-lines` array that are actually live, that is, used for display right now.
- `%max-line-width`
  The width of the longest displayed line in device units.
- `%lines`
  The lines of the buffer, stored in a format that makes it easy to retrieve information about them.
- `%lines-prefix`
  The number of unchanged objects at the start of the buffer since the list of lines was last updated.
- `%lines-suffix`
  The number of unchanged objects at the end of the buffer since since the list of lines was last updated.
- `%last-seen-buffer-size`
  The buffer size the last time a change to the buffer was registered.

A view that contains a `drei-buffer` object. The buffer is displayed on a simple line-by-line basis, with top and bot marks delimiting the visible region. These marks are automatically set if applicable.

`drei-buffer:buffer` (*drei-buffer-view drei-buffer-view*) [Method]
The buffer that is observed by this buffer view.

`drei:drei-syntax-view` [Class]
Class precedence list: `drei-syntax-view`, `drei-buffer-view`, `drei-view`, `tabify-mixin`, `subscriptable-name-mixin`, `name-mixin`, `standard-object`, `slot-object`, `t`

Slots:

- `%syntax`

  An instance of the syntax class used for this syntax view.

- `%prefix-size`

  The number of unchanged objects at the beginning of the buffer.

- `%suffix-size`

  The number of unchanged objects at the end of the buffer.

- `%recorded-buffer-size`

  The size of the buffer the last time the view was synchronized.

A buffer-view that maintains a parse tree of the buffer, or otherwise pays attention to the syntax of the buffer.

`drei:point-mark-view` [Class]

Class precedence list: `point-mark-view, drei-buffer-view, drei-view, tabify-mixin, subscriptable-name-mixin, name-mixin, standard-object, slot-object, t`

Slots:

- `%goal-column`

  The column that point will be attempted to be positioned in when moving by line.

A view class containing a point and a mark into its buffer.

The `synchronize-view` generic function is the heart of all view functionality.

`drei:synchronize-view` *view* **&key** *begin end force-p* **&allow-other-keys** [Generic Function]

Synchronize the view with the object under observation - what exactly this entails, and what keyword arguments are supported, is up to the individual view subclass.

4.6 Unit Protocol

Many of the actions performed by an editor is described in terms of the syntactically unit(s) they affect. The syntax module is responsible for actually dividing the buffer into syntactical units, but the *unit protocol* is the basic interface for acting on these units. A *unit* is some single syntactical construct - for example a word, a sentence, an expression or a definition. The unit protocol defines a number of generic functions for the various unit types that permit a uniform interface to moving a mark a given number of units, deleting a unit, killing a unit, transposing two units and so forth. A number of macros are also provided for automatically generating all these functions, given the definition of two simple movement functions. All generic functions of the unit protocol dispatch on a syntax, so that every syntax can implement its own idea of what exactly, for example, an “expression” is. Defaults are provided for some units - if nothing else has been specified by the syntax, a word is considered any sequence of alphanumeric characters delimited by non-alphanumeric characters.

The type of unit that a protocol function affects is represented directly in the name of the function - this means that a new set of functions must be generated for every new unit you want the protocol to support. In most cases, the code for these functions is very repetitive and similar across the unit types, which is why the motion protocol offers a set of macros that can generate function definitions for you. These generator macros define their generated functions in terms of basic motion functions.

A basic motion function is a function named `FORWARD-ONE-`*`unit`* or `backward-one-`*`unit`* of the signature `(`*`mark syntax`*`)` that returns true if any motion happened or false if a limit was reached.

There isn't really a single all-encompassing unit protocol, instead, it is divided into two major parts - a motion protocol defining functions for moving point in terms of units, and an editing protocol for changing the buffer in terms of units. Both use a similar interface and a general mechanism for specifying the action to take if the intended operation cannot be carried out.

Note that `forward-object` and `backward-object`, by virtue of their low-level status and placement in the buffer protocol (see buffer.lisp) do not obey this protocol, in that they have no syntax argument. Therefore, all *`frob`*`-object` functions and commands (see Section 4.6.3 [Editing Protocol], page 23) lack this argument as well. There are no `forward-one-object` or `backward-one-object` functions.

4.6.1 Motors And Limit Actions

A limit action is a function usually named *`mumble`*`-limit-action` of the signature `(`*`mark original-offset remaining-units unit syntax`*`)` that is called whenever a general motion function cannot complete the motion. *Mark* is the mark the object in motion; *original-offset* is the original offset of the mark, before any motion; *remaining-units* is the number of units left until the motion would be complete; *unit* is a string naming the unit; and *syntax* is the syntax instance passed to the motion function. There is a number of predefined limit actions:

`drei-motion:beep-limit-action` *mark original-offset remaining unit syntax* [Function]

This limit action will beep at the user.

`drei-motion:revert-limit-action` *mark original-offset remaining unit syntax* [Function]

This limit action will try to restore the mark state from before the attempted action. Note that this will not restore any destructive actions that have been performed, it will only restore the position of `mark`.

`drei-motion:motion-limit-error` [Condition]

Class precedence list: `motion-limit-error`, `error`, `serious-condition`, `condition`, `slot-object`, `t`

This error condition signifies that a motion cannot be performed.

`drei-motion:error-limit-action` *mark original-offset remaining unit syntax* [Function]

This limit action will signal an error of type `motion-limit-error`.

A *diligent motor* is a combination of two motion functions that has the same signature as a standard motion function (see Section 4.6.2 [Motion Protocol], page 23). The primary motion function is called the *motor* and the secondary motion function is called the *fiddler*. When the diligent motor is called, it will start by calling its motor - if the motor cannot carry out its motion, the fiddler will be called, and if the fiddler is capable of performing its motion, the motor will be called again, and if this second motor call also fails, the fiddler will be called yet again, etc. If at any time the call to the fiddler fails, the limit action provided in the call to the diligent motor will be activated. A typical diligent motor is the one used to implement a `Backward Lisp Expression` command - it attempts to move backwards by a single expression, and if that fails, it tries to move up a level in the expression tree and tries again.

`drei-motion:make-diligent-motor` *motor fiddler* [Function]
: Create and return a diligent motor with a default limit action of `beep-limit-action`. `motor` and `fiddler` will take turns being called until either `motor` succeeds or `fiddler` fails.

4.6.2 Motion Protocol

The concept of a *basic motion function* was introduced in Section 4.6 [Unit Protocol], page 21. A general motion function is a function named `forward-`*`unit`* or `backward-`*`unit`* of the signature `(`*`mark syntax`* `&optional (`*`count`* `1) (`*`limit-action`* `#'ERROR-LIMIT-ACTION))` that returns true if it could move forward or backward over the requested number of units, *count*, which may be positive or negative; and calls the limit action if it could not, or returns `NIL` if the limit action is `NIL`.

4.6.3 Editing Protocol

An editing function is a function named `forward-`*`frob-unit`* or `backward-`*`frob-unit`*, or just *`frob-unit`* in the case where discering between forward and backward commands does not make sense (an example is `transpose-`*`unit`*).

A proper unit is a unit for which all the functions required by the motion protocol has been implemented, this can be trivially done by using the macro `define-motion-commands` (see Section 4.6.4 [Generator Macros], page 23).

4.6.4 Generator Macros

4.7 Redisplay Protocol

A buffer can be on display in several panes, possibly by being located in several Drei instances. Thus, the buffer does not concern itself with redisplay, but assumes that whatever is using it will redisplay when appropriate. There is no predictable definitive rule for when a Drei instance will be redisplayed, but when it is, it will be done by calling the following generic function.

`drei:display-drei` *drei* **&key** *redisplay-minibuffer* [Generic Function]
: `drei` must be an object of type `drei` and `frame` must be a CLIM frame containing the editor pane of `drei`. If you define a new subclass of `drei`, you must define a method for this generic function. In most cases, methods defined on this function will merely be a trampoline to a function specific to the given Drei variant.

If `redisplay-minibuffer` is true, also redisplay `*minibuffer*` if it is non-NIL.

The redisplay engine supports view-specific customization of the display in order to facilitate such functionality as syntax highlighting. This is done through the following two generic functions, both of which have sensible default methods defined by `drei-buffer-view` and `drei-syntax-view`, so if your view is a subclass of either of these, you do not need to define them yourself.

`drei:display-drei-view-contents` *stream view* [Generic Function]
The purpose of this function is to display the contents of a Drei view to some output surface. `stream` is the CLIM output stream that redisplay should be performed on, `view` is the Drei view instance that is being displayed. Methods defined for this generic function can draw whatever they want, but they should not assume that they are the only user of `stream`, unless the `stream` argument has been specialized to some application-specific pane class that can guarantee this. For example, when accepting multiple values using the `accepting-values` macro, several Drei instances will be displayed simultaneously on the same stream. It is permitted to only specialise `stream` on `clim-stream-pane` and not `extended-output-stream`. When writing methods for this function, be aware that you cannot assume that the buffer will contain only characters, and that any subsequence of the buffer is coercable to a string. Drei buffers can contain arbitrary objects, and redisplay methods are required to handle this (though they are not required to handle it nicely, they can just ignore the object, or display the `princ`ed representation.)

`drei:display-drei-view-cursor` *stream view cursor* [Generic Function]
The purpose of this function is to display a visible indication of a cursor of a Drei view to some output surface. `stream` is the CLIM output stream that drawing should be performed on, `view` is the Drei view object that is being redisplayed, `cursor` is the cursor object to be displayed (a subclass of `drei-cursor`) and `syntax` is the syntax object of `view`. Methods on this generic function can draw whatever they want, but they should not assume that they are the only user of `stream`, unless the `stream` argument has been specialized to some application-specific pane class that can guarantee this. It is permitted to only specialise `stream` on `clim-stream-pane` and not `extended-output-stream`. It is recommended to use the function `offset-to-screen-position` to determine where to draw the visual representation for the cursor. It is also recommended to use the ink specified by `cursor` to perform the drawing, if applicable. This method will only be called by the Drei redisplay engine when the cursor is active and the buffer position it refers to is on display - therefore, `offset-to-screen-position` is *guaranteed* to not return NIL or `t`.

4.8 Undo Protocol

Undo is the facility by which previous modifications to the buffer can be undone, returning the buffer state to what it was prior to some modification.

Undo is organized into a separate module. This module conceptually maintains a tree where the nodes represent application states and the arcs represent transitions between these states. The root of the tree represents the initial state of the application. The undo module also maintains a current state. During normal application operation, the current

state is a leaf of a fairly long branch of the tree. Normal application operations add new nodes to the end of this branch. Moving the current state up the tree corresponds to an undo operation and moving it down some branch corresponds to some redo operation.

Arcs in the tree are ordered so that they always point FROM the current state. When the current state moves from one state to the other, the arc it traversed is reversed. The undo module does this by calling a generic function that client code must supply a method for.

4.8.1 Protocol Specification

`drei-undo:no-more-undo` [Condition]

Class precedence list: `no-more-undo`, `error`, `serious-condition`, `condition`, `slot-object`, `t`

A condition of this type is signaled whenever an attempt is made to call undo when the application is in its initial state.

`drei-undo:undo-tree` [Class]

Class precedence list: `undo-tree`, `standard-object`, `slot-object`, `t`

The base class for all undo trees.

`drei-undo:undo-record` [Class]

Class precedence list: `undo-record`, `standard-object`, `slot-object`, `t`

The base class for all undo records.

`drei-undo:standard-undo-record` [Class]

Class precedence list: `standard-undo-record`, `undo-record`, `standard-object`, `slot-object`, `t`

Slots:

- `tree`

  The undo tree to which the undo record belongs.

Standard instantiable class for undo records.

`drei-undo:add-undo` *undo-record undo-tree* [Generic Function]

Add an undo record to the undo tree below the current state, and set the current state to be below the transition represented by the undo record.

`drei-undo:flip-undo-record` *undo-record* [Generic Function]

This function is called by the undo module whenever the current state is changed from its current value to that of the parent state (presumably as a result of a call to undo) or to that of one of its child states.

Client code is required to supply methods for this function on client-specific subclasses of `undo-record`.

`drei-undo:undo` *undo-tree* **&optional** *n* [Generic Function]

Move the current state `n` steps up the undo tree and call `flip-undo-record` on each step. If the current state is at a level less than `n`, a `no-more-undo` condition is signaled and the current state is not moved (and no calls to `flip-undo-record` are made).

As long as no new record are added to the tree, the undo module remembers which branch it was in before a sequence of calls to undo.

`drei-undo:redo` *undo-tree* **&optional** *n* [Generic Function]
Move the current state `n` steps down the remembered branch of the undo tree and call `flip-undo-record` on each step. If the remembered branch is shorter than `n`, a `no-more-undo` condition is signaled and the current state is not moved (and no calls to `flip-undo-record` are made).

4.8.2 Implementation

Application states have no explicit representation, only undo records do. The current state is a pointer to an undo record (meaning, the current state is BELOW the transition represented by the record) or to the undo tree itself if the current state is the initial state of the application.

4.8.3 How The Buffer Handles Undo

`drei:undo-mixin` [Class]
Class precedence list: `undo-mixin, standard-object, slot-object, t`

Slots:

- `tree`
  Returns the undo-tree of the buffer.
- `undo-accumulate`
  The undo records created since the start of the undo context.
- `performing-undo`
  True if we are currently performing undo, false otherwise.

This is a mixin class that buffer classes can inherit from. It contains an undo tree, an undo accumulator and a flag specifyng whether or not it is currently performing undo. The undo tree and undo accumulators are initially empty.

`drei:undo-tree` *buffer* [Generic Function]
The undo-tree object associated with the buffer. This usually contains a record of every change that has been made to the buffer since it was created.

Undo is implemented as :before methods on, insert-buffer-object, insert-buffer-sequence and delete-buffer-range specialized on undo-mixin.

`drei:undo-accumulate` *buffer* [Generic Function]
A list of the changes that have been made to `buffer` since the last time undo was added to the undo tree for the buffer. The list returned by this function is initially NIL (the empty list). The :before methods on `insert-buffer-object`, `insert-buffer-sequence`, and `delete-buffer-range` push undo records on to this list.

`drei:performing-undo` *buffer* [Generic Function]
If true, the buffer is currently performing an undo operation. The :before methods on `insert-buffer-object`, `insert-buffer-sequence`, and `delete-buffer-range` push undo records onto the undo accumulator only if `performing-undo` is false, so that no undo information is added as a result of an undo operation.

Three subclasses `insert-record`, `delete-record`, and `compound-record` of undo-record are used. An insert record stores a position and some sequence of objects to be inserted, a delete record stores a position and the length of the sequence to be deleted, and a compound record stores a list of other undo records.

The :before methods on `insert-buffer-object` and `insert-buffer-sequence` push a record of type delete-record onto the undo accumulator for the buffer, and the :before method on `delete-buffer-range` pushes a record of type `insert-record` onto the undo accumulator.

`drei:with-undo` (*get-buffers-exp*) **&body** *body* [Macro]

This macro executes the forms of `body`, registering changes made to the list of buffers retrieved by evaluating `get-buffers-exp`. When `body` has run, for each buffer it will call `add-undo` with an undo record and the undo tree of the buffer. If the changes done by `body` to the buffer has resulted in only a single undo record, it is passed as is to `add-undo`. If it contains several undo records, a compound undo record is constructed out of the list and passed to `add-undo`. Finally, if the buffer has no undo records, `add-undo` is not called at all.

To avoid storing an undo record for each object that is inserted, the with-undo macro may in some cases just increment the length of the sequence in the last delete-record.

The method on `flip-undo-record` specialized on `insert-record` binds `performing-undo` for the buffer to `T`, inserts the sequence of objects in the buffer, and calls `change-class` to convert the `insert-record` to a `delete-record`, giving it a the length of the stored sequence.

The method on `flip-undo-record` specialized on `delete-record` binds `performing-undo` for the buffer to `T`, deletes the range from the buffer, and calls `change-class` to convert the `delete-record` to an `insert-record`, giving it the sequence at the stored offset in the buffer with the specified length.

The method on `flip-undo-record` specialized on `compound-record` binds `performing-undo` for the buffer to `T`, recursively calls `flip-undo-record` on each element of the list of undo records, and finally destructively reverses the list.

`drei:drei-undo-record` [Class]

Class precedence list: `drei-undo-record`, `standard-undo-record`, `undo-record`, `standard-object`, `slot-object`, `t`

Slots:

- `buffer` — initargs: `:buffer`

  The buffer to which the record belongs.

A base class for all output records in Drei.

`drei:simple-undo-record` [Class]

Class precedence list: `simple-undo-record`, `drei-undo-record`, `standard-undo-record`, `undo-record`, `standard-object`, `slot-object`, `t`

Slots:

- `offset` — initargs: `:offset`

  The offset that determines the position at which the undo operation is to be executed.

A base class for output records that modify buffer contents at a specific offset.

`drei:insert-record` [Class]

Class precedence list: `insert-record, simple-undo-record, drei-undo-record, standard-undo-record, undo-record, standard-object, slot-object, t`

Slots:

- `objects` — initargs: `:objects`

  The sequence of objects that are to be inserted whenever flip-undo-record is called on an instance of insert-record.

Whenever objects are deleted, the sequence of objects is stored in an insert record containing a mark.

`drei:delete-record` [Class]

Class precedence list: `delete-record, simple-undo-record, drei-undo-record, standard-undo-record, undo-record, standard-object, slot-object, t`

Slots:

- `length` — initargs: `:length`

  The length of the sequence of objects to be deleted whenever `flip-undo-record` is called on an instance of `delete-record`.

Whenever objects are inserted, a `delete-record` containing a mark is created and added to the undo tree.

`drei:compound-record` [Class]

Class precedence list: `compound-record, drei-undo-record, standard-undo-record, undo-record, standard-object, slot-object, t`

Slots:

- `records` — initargs: `:records`

  The undo records contained by this compound record.

This record simply contains a list of other records.

4.9 Kill Ring Protocol

During the process of text editing it may become necessary for regions of text to be manipulated non-sequentially. The kill ring and its surrounding protocol offers both a temporary location for data to be stored, as well as methods for stored data to be accessed.

Conceptually, the kill ring is a stack of bounded depth, so that when elements are pushed beyond that depth, the oldest element is removed. All newly added data is attached to a single point at the "start of ring position" or SORP.

This protocol provides two methods which govern how data is to be attached to the SORP. The first method moves the current SORP to a new position, on to which a new

object is attached. The second conserves the current position and replaces its contents with a sequence constructed of new and pre-existing SORP objects. This latter method is refered to as a "concatenating push".

For data retrievial the kill ring class provides a "yank point" which allows focus to be shifted from the SORP to other positions within the kill ring. The yank point is limited to two types of motition, one being a rotation away from the SORP and the other being an immediate return or "reset" to the start position. When the kill ring is modified, for example by a push, the yank point will be reset to the start position.

4.9.1 Kill Ring Protocol Specification

`drei-kill-ring:kill-ring` [Class]

Class precedence list: `kill-ring, standard-object, slot-object, t`

Slots:

- `max-size` — initargs: `:max-size`

  The limitation placed upon the number of elements held by the kill ring. Once the maximum size has been reached, older entries must first be removed before new ones can be added. When altered, any surplus elements will be silently dropped.

- `cursorchain`

  The cursorchain associated with the kill ring.

- `yankpoint`

  The flexicursor associated with the kill ring.

A class for all kill rings

`drei-kill-ring:kill-ring-max-size` *kr* [Generic Function]

Returns the value of the kill ring's maximum size

`drei-kill-ring:kill-ring-length` *kr* [Generic Function]

Returns the current length of the kill-ring. Note this is different than `kill-ring-max-size`.

`drei-kill-ring:kill-ring-standard-push` *kr vector* [Generic Function]

Pushes a vector of objects onto the kill ring creating a new start of ring position. This function is much like an everyday Lisp push with size considerations. If the length of the kill ring is greater than the maximum size, then "older" elements will be removed from the ring until the maximum size is reached.

`drei-kill-ring:kill-ring-concatenating-push` *kr vector* [Generic Function]

Concatenates the contents of vector onto the end of the current contents of the top of the kill ring. If the kill ring is empty the a new entry is pushed.

`drei-kill-ring:kill-ring-reverse-concatenating-push` *kr vector* [Generic Function]

Concatenates the contents of vector onto the front of the current contents of the top of the kill ring. If the kill ring is empty a new entry is pushed.

`drei-kill-ring:rotate-yank-position` *kr* **&optional** *times* [Generic Function]
Moves the yank point associated with a kill-ring one or times many positions away from the start of ring position. If times is greater than the current length then the cursor will wrap to the start of ring position and continue rotating.

`drei-kill-ring:reset-yank-position` *kr* [Generic Function]
Moves the current yank point back to the start of of kill ring position

`drei-kill-ring:kill-ring-yank` *kr* **&optional** *reset* [Generic Function]
Returns the vector of objects currently pointed to by the cursor. If `reset` is `t`, a call to `reset-yank-position` is called before the object is yanked. The default for reset is NIL. If the kill ring is empty, a condition of type `empty-kill-ring` is signalled.

4.9.2 Kill Ring Implementation

The kill ring structure is built mainly of two parts: the stack like ring portion, which is a cursorchain, and the yank point, which is a left-sticky-flexicursor. To initialize a kill ring, the :max-size slot initarg is simply used to set the max size. The remaining slots constisting of the cursorchain and the left-sticky-flexicursor are instantized upon creation of the kill ring.

Stored onto the cursorchain are simple-vectors of objects, mainly characters from a Drei buffer. In order to facilitate this, the kill ring implementation borrows heavily from the flexichain library of functions. The following functions lie outside the kill ring and flexichain protocols, but are pertinent to the kill ring implementation.

`drei-kill-ring:kill-ring-chain` *ring* [Generic Function]
Return the cursorchain associated with the kill ring `ring`.

`drei-kill-ring:kill-ring-cursor` *ring* [Generic Function]
Return the flexicursor associated with the kill ring.

5 Defining Drei Commands

Drei commands are standard CLIM commands that are stored in standard CLIM command tables. Drei uses a number of distinct command tables, some of which are merely used to group commands by category, and some whose contents may only be applicable under specific circumstances. When the contents of a command table is applicable, that command table is said to be active. Some syntaxes may define specific command tables that will only be active for buffers using that syntax. Commands in such tables are called syntax-specific commands.

5.1 Drei Command Tables

Here is a list of the command tables that are always active, along with a note describing what they are used for:

`comment-table` [Command Table]
: Commands for dealing with comments in, for example, source code. For syntaxes that do not have the concept of a comment, many of the commands of this table will not do anything.

`deletion-table` [Command Table]
: Commands that destructively modify buffer contents.

`editing-table` [Command Table]
: Commands that transform the buffer contents somehow (such as transposing two words).

`fill-table` [Command Table]
: Commands that fill (wrap) text.

`case-table` [Command Table]
: Commands that modify the case of characters.

`indent-table` [Command Table]
: Commands that indent text based on the current syntax.

`marking-table` [Command Table]
: Commands that deal with managing the mark or nondestructively copying buffer contents.

`movement-table` [Command Table]
: Commands that move point.

`search-table` [Command Table]
: Commands that can search the buffer.

`info-table` [Command Table]
: Commands that display information about the state of the buffer.

`self-insert-table` [Command Table]
: Commands that insert the gesture used to invoke them into the buffer. You probably won't need to add commands to this table.

`editor-table` [Command Table]
: A command table that inherits from the previously mentioned tables (plus some more). This command table is the "basic" table for accessing Drei commands, and is a good place to put your own user-defined commands if they do not fit in another table.

There are also two conditionally-active command tables:

`exclusive-gadget-table` [Command Table]
: This command table is only active in the gadget version of Drei.

`exclusive-input-editor-table` [Command Table]
: This command table is only active when Drei is used as an input-editor.

When you define keybindings for your commands, you should put the keybindings in the same command table as the command itself.

5.2 Numeric Argument In Drei

The numeric argument state is currently not directly accessible from within commands. However, Drei uses ESA's numeric argument processing code, Drei commands can thus be provided with numeric arguments in the same way as ESA commands can. When using `set-key` to setup keybindings, provide the value of `*numeric-argument-marker*` as an argument to have the command processing code automatically insert the value of the numeric argument whenever the keybinding is invoked. You can also use `*numeric-argument-p*` to have a boolean value, stating whether or not a numeric argument has been provided by the user, inserted. Note that you must write your commands to accept arguments before you can do this (see Section 5.3 [Examples Of Defining Drei Commands], page 32).

5.3 Examples Of Defining Drei Commands

A common text editing task is to repeat the word at point, but for some reason, Drei does not come with a command to do this, so we need to write our own. Fortunately, Drei is extensible software, and to that end, a `DREI-USER` package is provided that is intended for user customizations. We're going to create a standard CLIM command named `com-repeat-word` in the command table `editing-table`. The implementation consists of cloning the current point, move it a word backward, and insert into the buffer the sequence delimited by point and our moved mark. Our command takes no arguments.

```
(define-command (com-repeat-word :name t
                                 :command-table editing-table)
    ()
  (let ((mark (clone-mark (point)))
    (backward-word mark (current-syntax 1)
    (insert-sequence mark (region-to-sequence mark (point))))
```

For `(point)` and `(current-syntax)`, see Section 1.1 [Access Functions], page 2.

This command facilitates the single repeat of a word, but that's it. This is not very useful - instead, we would like a command that could repeat a word an arbitrary (user-supplied) number of times. The primary way for a CLIM command to ask for user-supplied values

is to use command arguments. We define a new command that takes an integer argument specifying the number of times to repeat the word at point.

```
(define-command (com-repeat-word :name t
                                  :command-table editing-table)
    ((count 'integer :prompt "Number of repeats"))
  (let ((mark (clone-mark (point)))
    (backward-word mark (current-syntax 1)
    (let ((word (region-to-sequence mark (point)))
      (dotimes (i count)
        (insert-sequence mark word)))))
```

Great - our command is now pretty full-featured. But with an editing operation as common as this, we really want it to be quickly accessible via some intuitive keystroke. We choose *M-C-r*. Also, it'd be nice if, instead of interactively quering us for commands, the command would just use the value of the numeric argument as the number of times to repeat. There's no way to do this with a named command (ie. when you run the command with *M-x*), but it's quite easy to do in a keybinding. We use the ESA `set-key` function:

```
(set-key `(com-repeat-word ,*numeric-argument-marker*)
         'editing-table
         '((#\r :control :meta)))
```

Now, pressing *M-C-r* will result in the `com-repeat-word` command being run with the first argument substituted for the value of the numeric argument. Since the numeric argument will be 1 if nothing else has been specified by the user, we are guaranteed that the first argument is always an integer, and we are guaranteed that the *count* argument will have a sensible default, even if the user does not explicitly provide a numeric argument.

5.4 Drei's Syntax Command Table Protocol

In order to provide conditionally active command tables, Drei defines the `syntax-command-table` class. While this class is meant to facilitate the addition of commands to syntaxes when they are run in a specific context (for example, a large editor application adding a `Show Macroexpansion` command to Lisp syntax), their modus operandi is general enough to be used for all conditional activity of command tables. This is useful for making commands available that could not be generally implemented for all Drei instances — returning to the `Show Macroexpansion` example, such a command can only be implemented if there is a sufficiently large place to show the expansion, and this might not be available for a generic Drei input-editor instance, but could be provided by an application designed for it.

Syntax command tables work by conditionally inheriting from other command tables, so it is necessary to define one (or more) command tables for the commands you wish to make conditionally available.

When providing a *:command-table* argument to **define-syntax** that names a syntax command table, an instance of the syntax command table will be used for the syntax.

`drei-syntax:syntax-command-table` [Class]

Class precedence list: `syntax-command-table, standard-command-table, command-table, standard-object, slot-object, t`

A syntax command table provides facilities for having frame-specific commands that do not show up when the syntax is used in other applications than the one it is supposed to. For example, the Return From Definition command should be available when Lisp syntax is used in Climacs (or another editor), but not anywhere else.

`drei-syntax:additional-command-tables` *editor command-table* [Generic Function]

Method combination: `APPEND` (most-specific-first)

Return a list of additional command tables that should be checked for commands in addition to those `command-table` inherits from. The idea is that methods are specialised to `editor` (which is at first a Drei instance), and that those methods may call the function again recursively with a new `editor` argument to provide arbitrary granularity for command-table-selection. For instance, some commands may be applicable in a situation where the editor is a pane or gadget in its own right, but not when it functions as an input-editor. In this case, a method could be defined for `application-frame` as the `editor` argument, that calls `additional-command-tables` again with whatever the "current" editor instance is. The default method on this generic function just returns the empty list.

`drei-syntax:define-syntax-command-table` *name* **&rest** *args* **&key** **&allow-other-keys** [Macro]

Define a syntax command table class with the provided name, as well as defining a CLIM command table of the same name. `args` will be passed on to `make-command-table`. An :around method on `command-table-inherit-from` for the defined class will also be defined. This method will make sure that when an instance of the syntax command table is asked for its inherited command tables, it will return those of the defined CLIM command table, as well as those provided by methods on `additional-command-tables`. Command tables provided through `additional-command-tables` will take precence over those specified in the usual way with :inherit-from.

Concept Index

Variable Index

D

E

Function And Macro Index

(

D

E