McCLIM User's Manual

The Users Guide

and

API Reference

Table of Contents

1 Introduction

CLIM is a large layered software system that allows the user to customize it at each level. The most simple ways of using CLIM is to directly use its top layer, which contains application frames, panes, and gadgets, very similar to those of traditional windowing system toolkits such as GTK, Tk, and Motif.

But there is much more to using CLIM. In CLIM, the upper layer with panes and gadgets is written on top of a basic layer containing more basic functionality in the form of sheets. Objects in the upper layer are typically instances of classes derived from those of the lower layer. Thus, nothing prevents a user from adding new gadgets and panes by writing code that uses the sheet layer.

Finally, since CLIM is written in Common Lisp, essentially all parts of it can be modified, replaced, or extended.

For that reason, a user's manual for CLIM must contain not only a description of the protocols of the upper layer, but also of all protocols, classes, functions, macros, etc. that are part of the specification.

1.1 Standards

This manual documents McCLIM 0.9.7-dev which is a mostly complete implementation of the CLIM 2.0 specification and its revision 2.2. To our knowledge version~2.2 of the CLIM specification is only documented in the "CLIM 2 User's Guide" by Franz. While that document is not a formal specification, it does contain many cleanups and is often clearer than the official specification; on the other hand, the original specification is a useful reference. This manual will note where McCLIM has followed the 2.2 API.

Also, some protocols mentioned in the 2.0 specification, such as parts of the incremental redisplay protocol, are clearly internal to CLIM and not well described. It will be noted here when they are partially implemented in McCLIM or not implemented at all.

1.2 How CLIM Is Different

Many new users of CLIM have a hard time trying to understand how it works and how to use it. A large part of the problem is that many such users are used to more traditional GUI toolkits, and they try to fit CLIM into their mental model of how GUI toolkits should work.

But CLIM is much more than just a GUI toolkit, as suggested by its name, it is an *interface manager*, i.e. it is a complete mediator between application "business logic" and the way the user interacts with objects of the application. In fact, CLIM doesn't have to be used with graphics output at all, as it contains a large collection of functionality to manage text.

Traditional GUI toolkits have an *event loop*. Events are delivered to GUI elements called *gadgets* (or *widgets*), and the programmer attaches *event handlers* to those gadgets in order to invoke the functionality of the application logic. While this way of structuring code is sometimes presented as a virtue ("Event-driven programming"), it has an unfortunate side effect, namely that event handlers are executed in a null context, so that it becomes hard

to even remember two consecutive events. The effect of event-driven programming is that applications written that way have very rudimentary interaction policies.

At the lowest level, CLIM also has an event loop, but most application programmers never have any reason to program at that level with CLIM. Instead, CLIM has a *command loop* at a much higher level than the event loop. At each iteration of the command loop:

1. A command is acquired. You might satisfy this demand by clicking on a menu item, by typing the name of a command, by hitting some kind of keystroke, by pressing a button, or by pressing some visible object with a command associated with it;
2. Arguments that are required by the command are acquired. Each argument is often associated with a *presentation type*, and visible objects of the right presentation type can be clicked on to satisfy this demand. You can also type a textual representation of the argument, using completion, or you can use a context menu;
3. The command is called on the arguments, usually resulting in some significant modification of the data structure representing your application logic;
4. A *display routine* is called to update the views of the application logic. The display routine may use features such as incremental redisplay.

Instead of attaching event handlers to gadgets, writing a CLIM application therefore consists of:

- writing CLIM commands that modify the application data structures independently of how those commands are invoked, and which may take application objects as arguments;
- writing display routines that turn the application data structures (and possibly some "view" object) into a collection of visible representations (having presentation types) of application objects;
- writing completion routines that allow you to type in application objects (of a certain presentation type) using completions;
- independently deciding how commands are to be invoked (menus, buttons, presentations, textual commands, etc).

By using CLIM as a mediator of command invocation and argument acquisition, you can obtain some very modular code. Application logic is completely separate from interaction policies, and the two can evolve separately and independently.

User Manual

2 CLIM Demos and Applications

2.1 Running the Demos

The McCLIM source distribution comes with a number of demos and applications. They are intended to showcase specific CLIM features, demonstrate programming techniques or provide useful tools.

These demos and applications are available in the `Examples` and `Apps` subdirectories of the source tree's root directory. Instructions for compiling, loading and running some of the demos are included in the files with the McCLIM installation instructions for your Common Lisp implementation.

Demos are meant to be run after loading the `clim-examples` system from the frame created with `(clim-demo:demodemo)`.

```
(asdf:load-system 'clim-examples)
(clim-demo:demodemo)
```

Available demos and tests are defined in the following files:

`Examples/demodemo.lisp`
: Demonstrates different pane types and other tests.

`Examples/clim-fig.lisp`
: Simple paint program.

`Examples/calculator.lisp`
: Simple desk calculator.

`Examples/method-browser.lisp`
: Example of how to write a CLIM application with a "normal" GUI, where "normal" is a completely event driven app built using gadgets and not using the command-oriented framework.

`Examples/address-book.lisp`
: Simple address book.

`Examples/puzzle.lisp`
: Simple puzzle game.

`Examples/colorslider.lisp`
: Interactive color editor.

`Examples/town-example.lisp`
: "Large Cities of Germany" application example by Max-Gerd Retzlaff.

`Examples/logic-cube.lisp`
: Software-rendered 3d logic cube game. Shows how the transformations work and how to implement custom handle-repaint methods.

`Examples/menutest.lisp`
: Displays a window with a simple menu bar.

`Examples/gadget-test.lisp`
: Displays a window with various gadgets.

`Examples/dragndrop.lisp`
: Example of "Drag and Drop" functionality.

`Examples/dragndrop-translator.lisp`
: Another example of "Drag and Drop" functionality (with colors!).

`Examples/draggable-graph.lisp`
: Demo of draggable graph nodes.

`Examples/image-viewer.lisp`
: A simple program for displaying images of formats known to McCLIM.

`Examples/font-selection.lisp`
: A font selection dialog.

`Examples/tabdemo.lisp`
: A tab layout demo (McCLIM extension).

`Examples/postscript-test.lisp`
: Displays text and graphics to a PostScript file. Run it with:

```
(clim-demo::postscript-test)
```

The resulting file `ps-test.ps` is generated in the current directory and can be displayed by a PostScript viewer such as `gv` on Unix-like systems.

`Examples/presentation-test.lisp`
: Displays an interactive window in which you type numbers that are successively added. When a number is expected as input, you can either type it at the keyboard, or click on a previously entered number. Labeled "Summation".

`Examples/sliderdemo.lisp`
: Apparently a calculator demo (see above). Labeled "Slider demo".

`Examples/stream-test.lisp`
: Interactive command processor that echoes its input in `*trace-output*`.

The following programs are currently **known not to work**:

- `Examples/gadget-test-kr.lisp`
- `Examples/traffic-lights.lisp`

2.2 Applications

`Apps/Listener`
: CLIM-enabled Lisp listener. System name is `clim-listener`. See instructions in `Apps/Listener/README` for more information.

`Apps/Inspector`
: CLIM-enabled Lisp inspector. System name is `clouseau`. See instructions in `Apps/Inspector/INSTALL` for more information..

`Apps/Debugger`
: Common Lisp debugger implemented in McCLIM. It uses the portable debugger interface developed for the Slime project. Application has some quirks and requires work. System name is `clim-debugger`.

`Apps/Functional-Geometry`

Peter Henderson idea, see `http://www.ecs.soton.ac.uk/~ph/funcgeo.pdf` and `http://www.ecs.soton.ac.uk/~ph/papers/funcgeo2.pdf` implemented in Lisp by Frank Buss. CLIM Listener interface by Rainer Joswig. System name is `functional-geometry`.

```
(functional-geometry:run-functional-geometry)
(clim-plot *fishes*) ; from a listener
```

3 The First Application

3.1 How CLIM applications produce output

CLIM stream panes use output recording. This means that such a pane maintains a display list, consisting of a sequence of output records, ordered chronologically, from the first output record to be drawn to the last.

This display list is used to fill in damaged areas of the pane, for instance as a result of the pane being partially or totally covered by other panes, and then having some or all of its area again becoming visible. The output records of the display list that have some parts in common with the exposed area are partially or totally replayed (in chronological order) to redraw the contents of the area.

An application can have a pane establish this display list in several fundamentally different ways.

Very simple applications have no internal data structure to keep track of application objects, and simply produce output to the pane from time to time as a result of running commands, occasionally perhaps erasing the pane and starting over. Such applications typically use text or graphics output as a result of running commands. CLIM maintains the display list for the pane, and adds to the end of it, each time also producing the pixels that result from drawing the new output record. If the pane uses scrolling (which it typically does), then CLIM must determine the extent of the pane so as to update the scroll bar after each new output.

More complicated applications use a display function. Before the display function is run, the existing display list is typically deleted, so that the purpose of the display function becomes to establish an entirely new display list. The display function might for instance produce some kind of form to be filled in, and application commands can use text or graphics operations to fill in the form. A game of tic-tac-toe could work this way, where the display function draws the board and commands draw shapes into the squares.

Even more complicated applications might have some internal data structure that has a direct mapping to output, and commands simply modify this internal data structure. In this case, the display function is run after each time around the command loop, because a command can have modified the internal data structure in some arbitrary ways. Some such applications might simply want to delete the existing display list and produce a new one each time (to minimize flicker, double buffering could be used). This is a very simple way of structuring an application, and entirely acceptable in many cases. Consider, for instance, a board game where pieces can be moved (as opposed to just added). A very simple way of structuring such an application is to have an internal representation of the board, and to make the display function traverse this data structure and produce the complete output each time in the command loop.

Some applications have very large internal data structures to be displayed, and it would cause a serious performance problem if the display list had to be computer from scratch each time around the command loop. To solve this problem, CLIM contains a feature called incremental redisplay. It allows many of the output records to be kept from one iteration of the command loop to the next. This can be done in two different ways. The simplest way is for the application to keep the simple structure which consists of traversing the entire data

structure each time, but at various points indicate to CLIM that the output has not changed since last time, so as to avoid actually invoking the application code for computing it. This is accomplished by the use of `updating-output`. The advantage of `updating-output` is that the application logic remains straightforward, and it is up to CLIM to do the hard work of recycling output records. The disadvantage is that for some very demanding applications, this method might not be fast enough.

The other way is more complicated and requires the programmer to structure the application differently. Essentially, the application has to keep track of the output records in the display list, and inform CLIM about modifications to it. The main disadvantage of this method is that the programmer must now write the application to keep track of the output records itself, as opposed to leaving it to CLIM.

3.2 Panes and Gadgets

A CLIM application is made up of a hierarchy of *panes* and *gadgets* (gadgets are special kinds of panes). These elements correspond to what other toolkits call *widgets*. Frequently used CLIM gadgets are `button`s, `slider`s, etc, and typical panes are the layout panes such as `hbox`, `vbox`, `hrack`, etc.

3.3 Defining Application Frames

Each CLIM application is defined by an *application frame*. An application frame is an instance of the class `application-frame`. As a CLIM user, you typically define a class that inherits from the class `application-frame`, and that contains additional slots needed by your application. It is considered good style to keep all your application-specific data in slots in the application frame (rather than, say, in global variables), and to define your application-specific application frame in its own package.

The usual way to define an application frame is to use the macro `define-application-frame`. This macro works much like `defclass`, but also allows you to specify the hierarchy of *panes* and *gadgets* to use.

3.4 A First Attempt

Let us define a very primitive CLIM application. For that, let us put the following code in a file:

```
(in-package :common-lisp-user)

(defpackage "APP"
  (:use :clim :clim-lisp)
  (:export "APP-MAIN"))

(in-package :app)

(define-application-frame superapp ()
  ()
  (:panes
    (int :interactor :height 400 :width 600))
```

```
    (:layouts
      (default int)))

(defun app-main ()
  (run-frame-top-level (make-application-frame 'superapp)))
```

As we can see in this example, we have put our application in a separate package, here a package named `APP`. While not required, putting the application in its own package is good practice.

The package for the application uses two packages: `CLIM` and `CLIM-LISP`. The `CLIM` package is the one that contains all the symbols needed for using CLIM. The `CLIM-LISP` package replaces the `COMMON-LISP` package for CLIM applications. It is essentially the same as the `COMMON-LISP` package as far as the user is concerned.

In our example, we export the symbol that corresponds to the main function to start our application, here called `APP-MAIN`.

The most important part of the code in our example is the definition of the application-frame. In our example, we have defined an application frame called `superapp`, which becomes a CLOS class that automatically inherits from some standard CLIM application frame class.

The second argument to `define-application-frame` is a list of additional superclasses from which you want your application frame to inherit. In our example, this list is empty, which means that our application frame only inherits from the standard CLIM application frame.

The third argument to `define-application-frame` is a list of CLOS slots to be added to any instance of this kind of application frame. These slots are typically used for holding all application-specific data. The current instance of the application frame will always be the value of the special variable `*application-frame*`, so that the values of these slots can be accessed. In our example, we do not initially have any further slots.

The rest of the definition of an application frame contains additional elements that CLIM will allow the user to define. In our example, we have two additional (mandatory) elements: `:panes` and `:layouts`.

The `:panes` element defines a collection of CLIM panes that each instance of your application may have. Each pane has a name, a type, and perhaps some options that are used to instantiate that particular type of pane. Here, we have a pane called `int` of type `:interactor` with a height of 400 units and a width of 600 units. In McCLIM, the units are initially physical units (number of pixels) of the native windowing system.

The `:layouts` element defines one or more ways of organizing the panes in a hierarchy. Each layout has a name and a description of a hierarchy. In our example, only one layout, named `default`, is defined. The layout called `default` is the one that is used by CLIM at startup. In our example, the corresponding hierarchy is trivial, since it contains only the one element `int`, which is the name of our only pane.

3.5 Executing the Application

In order to run a CLIM application, you must have a Lisp system that contains McCLIM. If you use CMUCL or SBCL, you either need a `core` file that already has McCLIM in it, or

else, you have to load the McCLIM compiled files that make up the McCLIM distribution. The fist solution is recommended so as to avoid having to load the McCLIM files each time you start your CLIM application.

To execute the application, load the file containing your code (possibly after compiling it) into your running Lisp system. Then start the application. Our example can be started by typing `(app:app-main)`.

3.6 Adding Functionality

In a serious application, you would probably want some area where your application objects are to be displayed. In CLIM, such an area is called an *application pane*, and would be an instance (direct or indirect) of the CLIM class `application-pane`. In fact, instances of this class are in reality also *streams* which can be used in calls both to ordinary input and output functions such as `format` and `read` and to CLIM-specific functions such as `draw-line`.

In this example we have such an application pane, the name of which is `app`. As you can see, we have defined it with an option `:display-time nil`. The default value for this option for an application pane is `:command-loop`, which means that the pane is cleared after each iteration in the command loop, and then redisplayed using a client-supplied *display function*. The default display function does nothing, and we have not supplied any, so if we had omitted the `:display-time nil` option, the `parity` command would have written to the pane. Then, at the end of the command loop, the pane would have been cleared, and nothing else would have been displayed. The net result is that we would have seen no visible output. With the option `:display-time nil`, the pane is never cleared, and output is accumulated every time we execute the `parity` command.

For this example, let us also add a few *commands*. Such commands are defined by the use of a macro called `define-`*`name`*`-command`, where *name* is the name of the application, in our case `superapp`. This macro is automatically defined by `define-application-frame`.

Let us also add a pane that automatically provides documentation for different actions on the pointer device.

Here is our improved example:

```
(in-package :common-lisp-user)

(defpackage "APP"
  (:use :clim :clim-lisp)
  (:export "APP-MAIN"))

(in-package :app)

(define-application-frame superapp ()
  ()
  (:pointer-documentation t)
  (:panes
    (app :application :display-time nil :height 400 :width 600)
    (int :interactor :height 200 :width 600))
  (:layouts
    (default (vertically () app int))))
```

```
(defun app-main ()
  (run-frame-top-level (make-application-frame 'superapp)))

(define-superapp-command (com-quit :name t) ()
  (frame-exit *application-frame*))

(define-superapp-command (com-parity :name t) ((number 'integer))
  (format t "~a is ~a~%" number
  (if (oddp number)
      "odd"
      "even")))
```

If you execute this example, you will find that you now have three different panes, the application pane, the interactor pane and the pointer documentation pane. In the pointer documentation pane, you will see the text `R possibilities` which indicates that if you click the right mouse button, you will automatically see a popup menu that lets you choose a command. In our case, you will have the default commands that are automatically proposed by McCLIM plus the commands that you defined yourself, in this case `quit` and `parity`.

Figure 3.1 shows what ought to be visible on the screen.

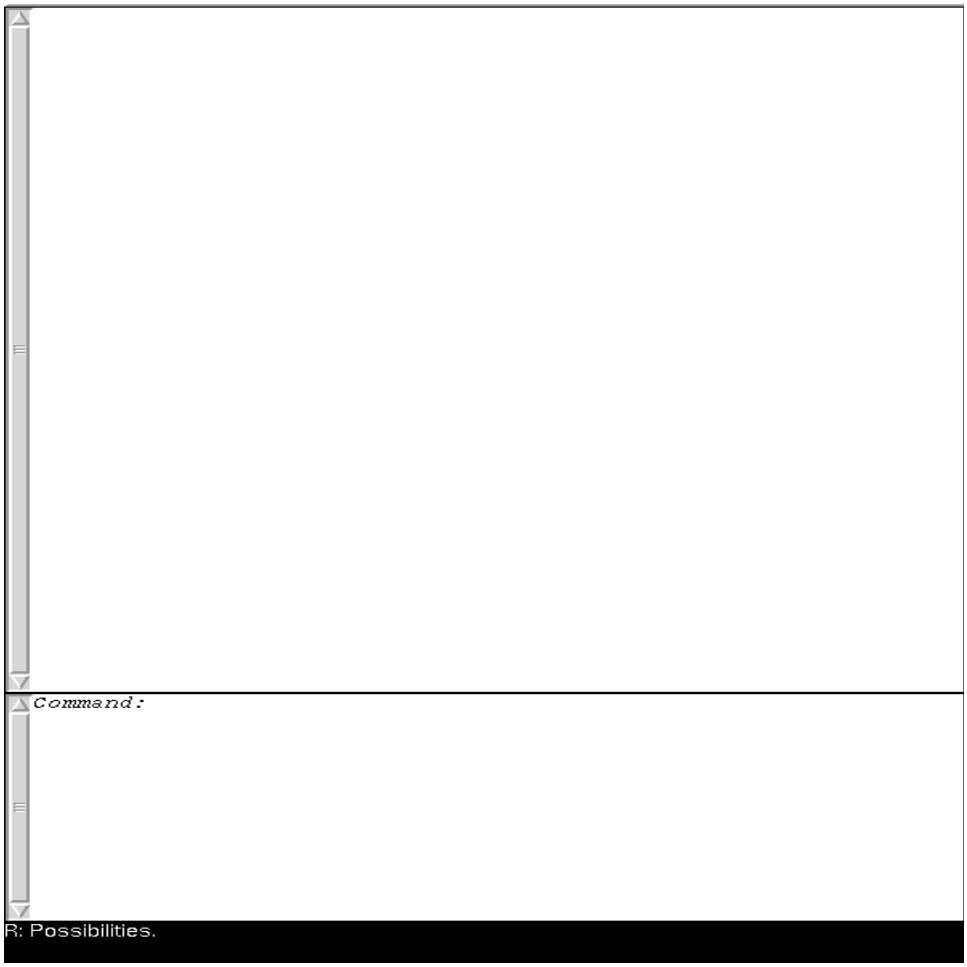


Figure 3.1

Notice that commands, in order to be available from the command line, must have an option of `:name t`. The reason is that some commands will be available only from menus or by some other mechanism.

You may notice that if the output of the application is hidden (say by the window of some other application) and then re-exposed, the output reappears normally, without any intervention necessary on the part of the programmer. This effect is accomplished by a CLIM mechanism called *output recording*. Essentially, every piece of output is not only displayed in the pane, but also captured in an *output record* associated with the pane. When a pane is re-exposed, its output records are consulted and if any of them overlap the re-exposed region, they are redisplayed. In fact, some others may be redisplayed as well, because CLIM guarantees that the effect will be the same as when the initial output was created. It does that by making sure that the order between (partially) overlapping output records is respected.

Not all panes support output recording, but certainly application panes do, so it is good to use some subclass of `application-pane` to display application-specific object, because output recording is then automatic.

3.7 An application displaying a data structure

Many applications use a central data structure that is to be on display at all times, and that is modified by the commands of the application. CLIM allows for a very easy way to write such an application. The main idea is to store the data structure in slots of the application frame, and to use a *display function* that after each iteration of the command loop displays the entire data structure to the application pane.

Here is a variation of the previous application that shows this possibility:

```
(in-package :common-lisp-user)

(defpackage "APP"
  (:use :clim :clim-lisp)
  (:export "APP-MAIN"))

(in-package :app)

(define-application-frame superapp ()
  ((currrent-number :initform nil :accessor current-number))
  (:pointer-documentation t)
  (:panes
    (app :application
 :height 400 :width 600
 :display-function 'display-app)
    (int :interactor :height 200 :width 600))
  (:layouts
    (default (vertically () app int))))

(defun display-app (frame pane)
  (let ((number (current-number frame)))
    (format pane "~a is ~a"
    number
    (cond ((null number) "not a number")
  ((oddp number) "odd")
  (t "even")))))

(defun app-main ()
  (run-frame-top-level (make-application-frame 'superapp)))

(define-superapp-command (com-quit :name t) ()
  (frame-exit *application-frame*))

(define-superapp-command (com-parity :name t) ((number 'integer))
  (setf (current-number *application-frame*) number))
```

Here, we have added a slot that is called `current-number` to the application frame. It is initialized to `NIL` and it has an accessor function that allow us to query and to modify the value.

Observe that in this example, we no longer have the option `:display-time nil` set in the application pane. By default, then, the `:display-time` is `:command-loop` which means that the pane is erased after each iteration of the command loop. Also observe the option `:display-function` which takes a symbol that names a function to be called to display the pane after it has been cleared. In this case, the name is `display-app`, the name of the function defined immediately after the application frame.

Instead of immediately displaying information about its argument, the command `com-parity` instead modifies the new slot of the application frame. Think of this function as being more general, for instance a command to add a new object to a set of graphical objects in a figure drawing program, or as a command to add a new name to an address book. Notice how this function accesses the current application frame by means of the special variable `*application-frame*`.

A display function is called with the frame and the pane as arguments. It is good style to use the pane as the stream in calls to functions that will result in output. This makes it possible for the same function to be used by several different frames, should that be called for. In our simple example, the display function only displays the value of a single number (or `NIL`), but you could think of this as displaying all the objects that have been drawn in some figure drawing program or displaying all the entries in an address book.

3.8 Incremental redisplay

While the example in the previous section is a very simple way of structuring an application (let commands arbitrarily modify the data structure, and simply erase the pane and redisplay the structure after each iteration of the command loop), the visual result is not so great when many objects are to be displayed. There is most often a noticeable flicker between the moment when the pane is cleared and the objects are drawn. Sometimes this is inevitable (as when nearly all objects change), but most of the time, only an incremental modification has been made, and most of the objects are still in the same place as before.

In simple toolkits, the application programmer would have to figure out what has changed since the previous display, and only display the differences. CLIM offers a mechanism called *incremental redisplay* that automates a large part of this task. As we mentioned earlier, CLIM captures output in the form of *output records*. The same mechanism is used to obtain incremental redisplay.

To use incremental redisplay, Client code remains structured in the simple way that was mention above: after each iteration of the command loop, the display function output the entire data structure as usual, except that it helps the incremental redisplay mechanism by telling CLIM which piece of output corresponds to which piece of output during the previous iteration of the command loop. It does this by giving some kind of *unique identity* to some piece of output, and some means of indicating whether the contents of this output is *the same* as it was last time. With this information, the CLIM incremental redisplay mechanism can figure out whether some output is new, has disappeared, or has been moved, compared to the previous iteration of the command loop. As with re-exposure, CLIM guarantees that

the result is identical to that which would have been obtained, had all the output records been output in order to a blank pane.

The next example illustrates this idea. It is a simple application that displays a fixed number (here 20) of lines, each line being a number. Here is the code:

```
(in-package :common-lisp-user)

(defpackage "APP"
  (:use :clim :clim-lisp)
  (:export "APP-MAIN"))

(in-package :app)

(define-application-frame superapp ()
  ((numbers :initform (loop repeat 20 collect (list (random 100000000)))
    :accessor numbers)
   (cursor :initform 0 :accessor cursor))
  (:pointer-documentation t)
  (:panes
    (app :application
 :height 400 :width 600
 :incremental-redisplay t
 :display-function 'display-app)
    (int :interactor :height 200 :width 600))
  (:layouts
    (default (vertically () app int))))

(defun display-app (frame pane)
  (loop for element in (numbers frame)
for line from 0
do (princ (if (= (cursor frame) line) "*" " ") pane)
do (updating-output (pane :unique-id element
  :id-test #'eq
  :cache-value (car element)
  :cache-test #'eql)
     (format pane "~a~%" (car element)))))

(defun app-main ()
  (run-frame-top-level (make-application-frame 'superapp)))

(define-superapp-command (com-quit :name t) ()
  (frame-exit *application-frame*))

(define-superapp-command (com-add :name t) ((number 'integer))
  (incf (car (elt (numbers *application-frame*)
  (cursor *application-frame*)))
number))
```

```
(define-superapp-command (com-next :name t) ()
  (incf (cursor *application-frame*))
  (when (= (cursor *application-frame*)
   (length (numbers *application-frame*)))
    (setf (cursor *application-frame*) 0)))

(define-superapp-command (com-prev :name t) ()
  (decf (cursor *application-frame*))
  (when (minusp (cursor *application-frame*))
    (setf (cursor *application-frame*)
  (1- (length (numbers *application-frame*))))))
```

We store the numbers in a slot called `numbers` of the application frame. However, we store each number in its own list. This is a simple way to provide a unique identity for each number. We could not use the number itself, because two numbers could be the same and the identities would not be unique. Instead, we use the cons cell that store the number as the unique identity. By using `:id-test #'eq` we inform CLIM that it can figure out whether an output record is the same as one that was issued previous time by using the function `eq` to compare them. But there is a second test that has to be verified, namely whether an output record that was issued last time has to be redisplayed or not. That is the purpose of the cache-value. Here we use the number itself as the cache value and `eql` as the test to determine whether the output is going to be the same as last time.

For convenience, we display a `*` at the beginning of the current line, and we provide two commands `next` and `previous` to navigate between the lines.

Notice that in the declaration of the pane in the application frame, we have given the option `:incremental-redisplay t`. This informs CLIM not to clear the pane after each command-loop iteration, but to keep the output records around and compare them to the new ones that are produced during the new iteration.

4 Using presentation types

4.1 What is a presentation type

The concept of *presentation types* is central to CLIM. Client code can choose to output graphical or textual representations of application objects either as just graphics or text, or to associate such output with an arbitrary Common Lisp object and a presentation type. The presentation type is not necessarily related to the idea Common Lisp might have of the underlying object.

When a CLIM command or some other client code requests an object (say as an argument) of a certain presentation type, the user of the application can satisfy the request by clicking on any visible output labeled with a compatible presentation type. The command then receives the underlying Common Lisp object as a response to the request.

CLIM presentation types are usually distinct from Common Lisp types. The reason is that the Common Lisp type system, although very powerful, is not quite powerful enough to represent the kind of relationships between types that are required by CLIM. However, every Common Lisp class (except the built-in classes) is automatically a presentation type.

A presentation type has a name, but can also have one or more *parameters*. Parameters of presentation types are typically used to restrict the type. For instance, the presentation type `integer` takes as parameters the low and the high values of an interval. Such parameters allow the application to restrict objects that become clickable in certain contexts, for instance if a date in the month of March is requested, only integers between 1 and 31 should be clickable.

4.2 A simple example

Consider the following example:

```
(in-package :common-lisp-user)

(defpackage :app
  (:use :clim :clim-lisp)
  (:export #:app-main))

(in-package :app)

(define-application-frame superapp ()
  ()
  (:pointer-documentation t)
  (:panes
    (app :application :display-time t :height 300 :width 600)
    (int :interactor :height 200 :width 600))
  (:layouts
    (default (vertically () app int))))

(defun app-main ()
  (run-frame-top-level (make-application-frame 'superapp)))
```

```
(define-superapp-command (com-quit :name t) ()
  (frame-exit *application-frame*))

(define-presentation-type name-of-month ()
  :inherit-from 'string)

(define-presentation-type day-of-month ()
  :inherit-from 'integer)

(define-superapp-command (com-out :name t) ()
  (with-output-as-presentation (t "The third month" 'name-of-month)
    (format t "March~%"))
  (with-output-as-presentation (t 15 'day-of-month)
    (format t "fifteen~%")))

(define-superapp-command (com-get-date :name t)
    ((name 'name-of-month) (date 'day-of-month))
  (format (frame-standard-input *application-frame*)
  "the ~a of ~a~%" date name))
```

In this application, we have two main panes, an application pane and an interactor pane. The application pane is given the option `:display-time t` which means that it will not be erased before every iteration of the command loop.

We have also defined two presentation types: `name-of-month` and `day-of-month`. The `out` command uses `with-output-as-presentation` in order to associate some output, a presentation type, and an underlying object. In this case, it will show the string "March" which is considered to be of presentation type `name-of-month` with the underlying object being the character string `"The third month"`. It will also show the string "fifteen" which is considered to be of presentation type `day-of-month` with the underlying object being the number 15. The argument `t` to `with-output-as-presentation` indicates that the stream to present on is `*standard-output*`.

Thus, if the `out` command has been executed, and then the user types "Get Date" in the interactor pane, the `get-date` command will try to acquire its arguments, the first of presentation type `name-of-month` and the second of type `day-of-month`. At the first prompt, the user can click on the string "March" but not on the string "fifteen" in the application pane. At the second prompt it is the string "fifteen" that is clickable, whereas "March" is not.

The `get-date` command will acquire the underlying objects. What is finally displayed (in the interactor pane, which is the standard input of the frame), is "the 15 of The third month".

5 Using views

The CLIM specification mentions a concept called a *view*, and also lists a number of predefined views to be used in various different contexts.

In this chapter we show how the *view* concept can be used in some concrete programming examples. In particular, we show how to use a single pane to show different views of the application data structure at different times. To switch between the different views, we supply a set of commands that alter the `stream-default-view` feature of all CLIM extended output streams.

The example shown here has been stripped to a bare minimum in order to illustrate the important concepts. A more complete version can be found in `Examples/views.lisp` in the McCLIM source tree.

Here is the example:

```
;;; part of application "business logic"
(defclass person ()
  ((%last-name :initarg :last-name :accessor last-name)
   (%first-name :initarg :first-name :accessor first-name)
   (%address :initarg :address :accessor address)
   (%membership-number :initarg :membership-number :reader membership-number)))

;;; constructor for the PERSON class.  Not strictly necessary.
(defun make-person (last-name first-name address membership-number)
  (make-instance 'person
                 :last-name last-name
                 :first-name first-name
                 :address address
                 :membership-number membership-number))

;;; initial list of members of the organization we imagine for this example
(defparameter *members*
  (list (make-person "Doe" "Jane" "123, Glencoe Terrace" 12345)
        (make-person "Dupont" "Jean" "111, Rue de la Republique" 54321)
        (make-person "Smith" "Eliza" "22, Trafalgar Square" 121212)
        (make-person "Nilsson" "Sven" "Uppsalagatan 33" 98765)))

;;; the CLIM view class that corresponds to a list of members, one member
;;; per line of text in a CLIM application pane.
(defclass members-view (view) ())

;;; since this view does not take any parameters in our simple example,
;;; we need only a single instance of it.
(defparameter *members-view* (make-instance 'members-view))

;;; the application frame.  It contains instance-specific data
;;; such as the members of our organization.
(define-application-frame views ()
```

```
  ((%members :initform *members* :accessor members))
  (:panes
   (main-pane :application :height 500 :width 500
              :display-function 'display-main-pane
              ;; notice the initialization of the default view of
              ;; the application pane.
              :default-view *members-view*)
   (interactor :interactor :height 100 :width 500))
  (:layouts
   (default (vertically ()
              main-pane
              interactor))))

;;; the trick here is to define a generic display function
;;; that is called on the frame, the pane AND the view,
;;; whereas the standard CLIM display functions are called
;;; only on the frame and the pane.
(defgeneric display-pane-with-view (frame pane view))

;;; this is the display function that is called in each iteration
;;; of the CLIM command loop.  We simply call our own, more elaborate
;;; display function with the default view of the pane.
(defun display-main-pane (frame pane)
  (display-pane-with-view frame pane (stream-default-view pane)))

;;; now we can start writing methods on our own display function
;;; for different views.  This one displays the data each member
;;; on a line of its own.
(defmethod display-pane-with-view (frame pane (view members-view))
  (loop for member in (members frame)
        do (with-output-as-presentation
               (pane member 'person)
             (format pane "~a, ~a, ~a, ~a~%"
                     (membership-number member)
                     (last-name member)
                     (first-name member)
                     (address member)))))

;;; this CLIM view is used to display the information about
;;; a single person.  It has a slot that indicates what person
;;; we want to view.
(defclass person-view (view)
  ((%person :initarg :person :reader person)))

;;; this method on our own display function shows the detailed
;;; information of a single member.
(defmethod display-pane-with-view (frame pane (view person-view))
```

```
  (let ((person (person view)))
    (format pane "Last name: ~a~%First Name: ~a~%Address: ~a~%Membership Number: ~a~%"
            (last-name person)
            (first-name person)
            (address person)
            (membership-number person))))

;;; entry point to start our applciation
(defun views-example ()
  (run-frame-top-level (make-application-frame 'views)))

;;; command to quit the application
(define-views-command (com-quit :name t) ()
  (frame-exit *application-frame*))

;;; command to switch the default view of the application pane
;;; (which is the value of *standard-output*) to the one that
;;; shows a member per line.
(define-views-command (com-show-all :name t) ()
  (setf (stream-default-view *standard-output*) *members-view*))

;;; command to switch to a view that displays a single member.
;;; this command takes as an argument the person to display.
;;; In this application, the only way to satisfy the demand for
;;; the argument is to click on a line of the members view.  In
;;; more elaborate application, you might be able to type a
;;; textual representation (using completion) of the person.
(define-views-command (com-show-person :name t) ((person 'person))
  (setf (stream-default-view *standard-output*)
        (make-instance 'person-view :person person)))
```

The example shows a stripped-down example of a simple database of members of some organization.

The main trick used in this example is the `display-main-pane` function that is declared to be the display function of the main pane in the application frame. The `display-main-pane` function trampolines to a generic function called `display-pane-with-view`, and which takes an additional argument compared to the display functions of CLIM panes. This additional argument is of type `view` which allows us to dispatch not only on the type of frame and the type of pane, but also on the type of the current default view. In this example the view argument is simply taken from the default view of the pane.

A possibility that is not obvious from reading the CLIM specification is to have views that contain additional slots. Our example defines two subclasses of the CLIM `view` class, namely `members-view` and `person-view`.

The first one of these does not contain any additional slots, and is used when a global view of the members of our organization is wanted. Since no instance-specific data is required

in this view, we follow the idea of the examples of the CLIM specification to instantiate a singleton of this class and store that singleton in the `stream-default-view` of our main pane whenever a global view of our organization is required.

The `person-view` class, on the other hand, is used when we want a closer view of a single member of the organization. This class therefore contains an additional slot which holds the particular person instance we are interested in. The method on `display-pane-with-view` that specializes on `person-view` displays the data of the particular person that is contained in the view.

To switch between the views, we provide two commands. The command `com-show-all` simply changes the default view of the main pane to be the singleton instance of the `members-view` class. The command `com-show-person` is more complicated. It takes an argument of type person, creates an instance of the `person-view` class initialized with the person that was passed as an argument, and stores the instance as the default view of the main pane.

6 Using command tables

A *command table* is an object that is used to determine what commands are available in a particular context and the ways in which commands can be executed.

Simple applications do not manage command tables explicitly. A default command table is created as a result of a call to the macro `define-application-frame` and that command table has the same name as the application frame.

Each command table has a *name* and that CLIM manages a global *namespace* for command tables.

`find-command-table` *name* (*errorp t*) [Function]

This function returns the command table with the name *name*. If there is no command table with that name, then what happens depends on the value of *errorp*. If *errorp* is *true*, then an error of type `command-table-not-found` is signaled. If *errorp* is *false*, otherwise `nil` is returned.

`define-command-table` *name* `i`*nherit-from menu inherit-menu* [Macro]

`make-command-table` *name* `i`*nherit-from menu inherit-menu* (*errorp t*) [Macro]

–

By default command tables inherit from `global-command-table`. According to the CLIM~2.0 specification, a command table inherits from no command table if `nil` is passed as an explicit argument to `inherit-from`. In revision~2.2 all command tables must inherit from `global-command-table`. McCLIM treats a `nil` value of *inherit-from* as specifying `'(global-command-table)`.

Reference Manual

7 Concepts

7.1 Coordinate systems

CLIM uses a number of different coordinate systems and transformations to transform coordinates between them.

The coordinate system used for the arguments of drawing functions is called the *user coordinate system*, and coordinate values expressed in the user coordinate system are known as *user coordinates*.

Each sheet has its own coordinate system called the *sheet coordinate system*, and positions expressed in this coordinate system are said to be expressed in *sheet coordinates*. User coordinates are translated to *sheet coordinates* by means of the *user transformation* also called the *medium transformation*. This transformation is stored in the *medium* used for drawing. The medium transformation can be composed temporarily with a transformation given as an explicit argument to a drawing function. In that case, the user transformation is temporarily modified for the duration of the drawing.

Before drawing can occur, coordinates in the sheet coordinate system must be transformed to *native coordinates*, which are coordinates of the coordinate system of the native windowing system. The transformation responsible for computing native coordinates from sheet coordinates is called the *native transformation*. Notice that each sheet potentially has its own native coordinate system, so that the native transformation is specific for each sheet. Another way of putting it is that each sheet has a mirror, which is a window in the underlying windowing system. If the sheet has its own mirror, it is the *direct mirror* of the sheet. Otherwise its mirror is the direct mirror of one of its ancestors. In any case, the native transformation of the sheet determines how sheet coordinates are to be translated to the coordinates of that mirror, and the native coordinate system of the sheet is that of its mirror.

The composition of the user transformation and the native transformation is called the *device transformation*. It allows drawing functions to transform coordinates only once before obtaining native coordinates.

Sometimes, it is useful to express coordinates of a sheet in the coordinate of its parent. The transformation responsible for that is called the *sheet transformation*.

7.2 Arguments to drawing functions

Drawing functions are typically called with a sheet as an argument.

A sheet often, but not always, corresponds to a window in the underlying windowing system.

8 Windowing system drawing functions

A typical windowing system provides a hierarchy of rectangular areas called windows. When a drawing functions is called to draw an object (such as a line or a circle) in a window of such a hierarchy, the arguments to the drawing function will include at least the window and a number of coordinates relative to (usually) the upper left corner of the window.

To translate such a request to the actual altering of pixel values in the video memory, the windowing system must translate the coordinates given as argument to the drawing functions into coordinates relative to the upper left corner of the entire screen. This is done by a composition of translation transformations applied to the initial coordinates. These transformations correspond to the position of each window in the coordinate system of its parent.

Thus a window in such a system is really just some values indicating its height, its width, and its position in the coordinate system of its parent, and of course information about background and foreground colors and such.

9 CLIM drawing functions

CLIM generalizes the concept of a hierarchy of window in a windowing system in several different ways. A window in a windowing system generalizes to a *sheet* in CLIM. More precisely, a window in a windowing system generalizes to the *sheet region* of a sheet. A CLIM sheet is an abstract concept with an infinite *drawing plane* and the *region* of the sheet is the potentially visible part of that drawing plane.

CLIM *sheet region*s don't have to be rectangular the way windows in most windowing systems have to be. Thus, the width and the height of a window in a windowing system generalizes to an arbitrary *region* in CLIM. A CLIM region is simply a set of mathematical points in a plane. CLIM allows this set to be described as a combination (union, intersection, difference) of elementary regions made up of rectangles, polygons and ellipses.

Even rectangular regions in CLIM are generalizations of the width+height concept of windows in most windowing systems. While the upper left corner of a window in a typical windowing system has coordinates (0,0), that is not necessarily the case of a CLIM region. CLIM uses that generalization to implement various ways of scrolling the contents of a sheet. To see that, imagine just a slight generalization of the width+height concept of a windowing system into a rectangular region with x+y+width+height. Don't confuse the x and y here with the position of a window within its parent, they are different. Instead, imagine that the rectangular region is a hole into the (infinite) drawing plane defined by all possible coordinates that can be given to drawing functions. If graphical objects appear in the window with respect to the origin of some coordinate system, and the upper-left corner of the window has coordinates (x,y) in that coordinate system, then changing x and y will have the effect of scrolling.

CLIM sheets also generalize windows in that a window typically has pixels with integer-value coordinates. CLIM sheets, on the other hand, have infinte resolution. Drawing functions accept non-integer coordinate values which are only translated into integers just before the physical rendering on the screen.

The x and y positions of a window in the coordinate system of its parent window in a typical windowing system is a translation transformation that takes coordinates in a window and transform them into coordinates in the parent window. CLIM generalizes this concepts to arbitrary affine transformations (combinations of translations, rotations, and scalings). This generalization makes it possible for points in a sheet to be not only translated compared to the parent sheet, but also rotated and scaled (including negative scaling, giving mirror images). A typical use for scaling would be for a sheet to be a zoomed version of its parent, or for a sheet to have its y-coordinate go the opposite direction from that of its parent.

When the shapes of, and relationship between sheets are as simple as those of a typical windowing system, each sheet typically has an associated window in the underlying windowing system. In that case, drawing on a sheet translates in a relativly straightforward way into drawing on the corresponding window. CLIM sheets that have associated windows in the underlying windowing system are called *mirrored sheets* and the system-dependent window object is called the *mirror*. When shapes and relationships are more complicated, CLIM uses its own transformations to transform coordinates from a sheet to its parent and to its grandparent, etc., until a *mirrored sheet* is found. To the user of CLIM, the net effect is to have a windowing system with more general shapes of, and relationships between windows.

10 Panes

Panes are subclasses of sheets. Some panes are *layout panes* that determine the size and position of its children according to rules specific to each particular type of layout pane. Examples of layout panes are vertical and horizontal boxes, tables etc.

According to the CLIM specification, all CLIM panes are *rectangular objects*. For McCLIM, we interpret that phrase to mean that:

- CLIM panes appear rectangular in the native windowing system;
- CLIM panes have a native transformation that does not have a rotation component, only translation and scaling.

Of course, the specification is unclear here. Panes are subclasses of sheets, and sheets don't have a shape per-se. Their *regions* may have a shape, but the sheet itself certainly does not.

The phrase in the specification *could* mean that the *sheet-region* of a pane is a subclass of the region class *rectangle*. But that would not exclude the possibility that the region of a pane would be some non-rectangular shape in the *native coordinate system*. For that to happen, it would be enough that the *sheet-transformation* of some ancestor of the pane contain a rotation component. In that case, the layout protocol would be insufficient in its current version.

McCLIM panes have the following additional restrictions:

- McCLIM panes have a coordinate system that is only a translation compared to that of the frame manager;
- The parent of a pane is either nil or another pane.

Thus, the panes form a *prefix* in the hierarchy of sheets. It is an error for a non-pane to adopt a pane.

Notice that the native transformation of a pane need not be the identity transformation. If the pane is not mirrored, then its native transformation is probably a translation of that of its parent.

Notice also that the native transformation of a pane need not be the composition of the identity transformation and a translation. That would be the case only of the native transformation of the top level sheet is the identity transformation, but that need not be the case. It is possible for the frame manager to impose a coordinate system in (say) millimeters as opposed to pixels. The native transformation of the top level sheet of such a frame manager is a scaling with coefficients other than 1.

10.1 Creating panes

There is some confusion about the options that are allowed when a pane is created with `make-pane`. Some parts of the specification suggest that stream panes such as application panes and interactor panes can be created using `make-pane` and an option `:scroll-bars`. Since these application panes do not in themselves contain any scroll bars, using that option results in a pane hierarchy being created with the topmost pane being a pane of type `scroller-pane`.

As far as McCLIM is concerned, this option to `make-pane` is obsolete.[1] The same goes for using this option together with the equivalent keyword, i.e., `:application` or `interactor`, in the `:panes` section of `define-application-frame`.

Instead, we recommend following the examples of the specification, where scroll bars are added in the `layouts` section of `define-application-frame`.

When specification talks about panes in a fashion implying some order (i.e "first application-pane") McCLIM assumes order of definition, not order of appearing in layout. Particularly that means, that if one pane is put before another in `:panes` option, then it precedes it. It is relevant to `frame-standard-output` (therefore binding of `*standard-output*`) and other similar functions.

10.2 Pane names

Every pane class accepts the initialization argument `:name` the value of which is typically a symbol in the package defined by the application. The generic function `pane-name` returns the value of this initialization argument. There is no standard way of changing the name of an existing pane. Using the function `reinitialize-instance` may not have the desired effect, since the application frame may create a dictionary mapping names to panes, and there is no way to invalidate the contents of such a potential dictionary.

The function `find-pane-named` searches the pane hierarchy of the application frame, consulting the names of each pane until a matching name is found. The CLIM specification does not say what happens if a name is given that does not correspond to any pane. McCLIM returns `nil` in that case. If pane names are not unique, it is unspecified which of several panes is returned by a call to this function.

If the advice of Section 10.1 [Creating panes], page 30, is followed, then the name given in the `:panes` option of the macro `define-application-frame` will always be the name of the top-level pane returned by the `body` following the pane name.

If that advice is not followed, then the name given to a pane in the `:panes` option of the macro `define-application-frame` may or may not become the name of the pane that is constructed by the `body` that follows the name. Recall that the syntax of the expression that defines a pane in the `:panes` option is `(name . body)`. Currently, McCLIM does the following:

- If the `body` creates a pane by using a keyword, or by using an explicitly mentioned call to `make-pane`, then the name is given to the pane of the type explicitly mentioned, even when the option `:scroll-bars` is given.
- If the `body` creates a pane by calling some arbitrary form other than a call to `make-pane`, then the name is given to the topmost pane returned by the evaluation of that form.

We reserve the right to modify this behavior in the future. Application code should respect the advice given in Section 10.1 [Creating panes], page 30.

[1] In the specification, there is no example of the use of this option to `make-pane` or to the equivalent keywords in the `:panes` section of `define-application-frame`. There is however one instance where the `:scroll-bars` option is mention for pane creation. We consider this to be an error in the specification.

10.3 Layout protocol

There is a set of fundamental rules of CLIM dividing responsibility between a parent pane and a child pane, with respect to the size and position of the region of the child and the *sheet transformation* of the child. This set of rules is called the *layout protocol*.

The layout protocol is executed in two phases. The first phase is called the *space compostion* phase, and the second phase is called the *space allocation* phase.

10.3.1 Space composition

The space composition is accomplished by the generic function `compose-space`. When applied to a pane, `compose-space` returns an object of type *space-requirement* indicating the needs of the pane in terms of preferred size, minimum size and maximum size. The phase starts when compose-space is applied to the top-level pane of the application frame. That pane in turn may ask its children for their space requirements, and so on until the leaves are reached. When the top-level pane has computed its space requirments, it asks the system for that much space. A conforming window manager should respect the request (space wanted, min space, max space) and allocate a top-level window of an acceptable size. The space given by the system must then be distributed among the panes in the hierarchy `space-allocation`.

Each type of pane is responsible for a different method on `compose-space`. Leaf panes such as *labelled gadgets* may compute space requirements based on the size and the text-style of the label. Other panes such as the vbox layout pane compute the space as a combination of the space requirements of their children. The result of such a query (in the form of a space-requirement object) is stored in the pane for later use, and is only changed as a result of a call to `note-space-requirement-changed`.

Most *composite panes* can be given explicit values for the values of `:width`, `:min-width`, `:max-width`, `:height`, `:min-height`, and `:max-height` options. If such arguments are not given (effectively making these values nil), a general method is used, such as computing from children or, for leaf panes with no such reasonable default rule, a fixed value is given. If such arguments are given, their values are used instead. Notice that one of `:height` and `:width` might be given, applying the rule only in one of the dimensions.

Subsequent calls to `compose-space` with the same arguments are assumed to return the same space-requirement object, unless a call to note-space-requirement-changed has been called in between.

10.3.2 Space allocation

When `allocate-space` is called on a pane P, it must compare the space-requirement of the children of P to the available space, in order to distribute it in the most preferable way. In order to avoid a second recursive invokation of `compose-space` at this point, we store the result of the previous call to `compose-space` in each pane.

To handle this situtation and also explicitly given size options, we use an `:around` method on `compose-space`. The `:around` method will call the primary method only if necessary (i.e., `(eq (slot-value pane 'space-requirement) nil)`), and store the result of the call to the primary method in the `space-requirement` slot.

We then compute the space requirement of the pane as follows:

```
(setf (space-requirement-width ...)  (or explicit-width
```

```
(space-requirement-width request)) ...
(space-requirement-max-width ...)  (or explicit-max-width
explicit-width (space-requirement-max-width request)) ...)
```

When the call to the primary method is not necessary we simply return the stored value.

The `spacer-pane` is an exception to the rule indicated above. The explicit size you can give for this pane should represent the margin size. So its primary method should only call compose on the child. And the around method will compute the explicit sizes for it from the space requirement of the child and for the values given for the surrounding space.

10.3.3 Change-space Notification Protocol

The purpose of the change-space notification protocol is to force a recalculation of the space occupied by potentially each pane in the *pane hierarchy*. The protocol is triggerred by a call to `note-space-requirement-changed` on a pane P. In McCLIM, we must therefore invalidate the stored space-requirement value and re-invoke `compose-space` on P. Finally, the *parent* of P must be notified recursively.

This process would be repeated for all the panes on a path from P to the top-level pane, if it weren't for the fact that some panes compute their space requirements independently of those of their children. Thus, we stop calling `note-space-requirement-changed` in the following cases:

- when P is a `restraining-pane`,
- when P is a `top-level-sheet-pane`, or
- when P has been given explicit values for `:width` and `:height`

In either of those cases, `allocate-space` is called.

11 Output Protocol

11.1 Protocol Changes

`clim-extensions:line-style-effective-thickness` [Generic Function]
line-style medium

Returns the thickness in device units of a line, rendered on MEDIUM with the style LINE-STYLE.

`(setf output-record-parent)` *parent record* [Generic Function]

Additional protocol generic function. *parent* may be an output record or `nil`.

`clim:replay-output-record` *record stream* **&optional** *region x-offset y-offset* [Generic Function]

Displays the output captured by RECORD on the STREAM, exactly as it was originally captured. The current user transformation, line style, text style, ink and clipping region of STREAM are all ignored. Instead, these are gotten from the output record.

Only those records that overlap `region` are displayed.

`clim:map-over-output-records-containing-position` [Generic Function]
function record x y **&optional** *x-offset y-offset* **&rest** *function-args*

Maps over all of the children of RECORD that contain the point at (X,Y), calling FUNCTION on each one. FUNCTION is a function of one or more arguments, the first argument being the record containing the point. FUNCTION is also called with all of FUNCTION-ARGS as APPLY arguments.

If there are multiple records that contain the point, MAP-OVER-OUTPUT-RECORDS-CONTAINING-POSITION hits the most recently inserted record first and the least recently inserted record last. Otherwise, the order in which the records are traversed is unspecified.

`clim:map-over-output-records-overlapping-region` [Generic Function]
function record region **&optional** *x-offset y-offset* **&rest** *function-args*

Maps over all of the children of the RECORD that overlap the `region`, calling FUNCTION on each one. FUNCTION is a function of one or more arguments, the first argument being the record overlapping the region. FUNCTION is also called with all of FUNCTION-ARGS as APPLY arguments.

If there are multiple records that overlap the region and that overlap each other, `map-over-output-records-overlapping-region` hits the least recently inserted record first and the most recently inserted record last. Otherwise, the order in which the records are traversed is unspecified.

`add-output-record` *child record* [Generic Function]

Sets *record* to be the parent of *child*.

`delete-output-record` *child record* **&optional** (*errorp* **t**) [Generic Function]

If *child* is a child of *record*, sets the parent of *child* to `nil`.

`clear-output-record` *record* [Generic Function]

Sets the parent of all children of *record* to `nil`.

`clim:with-new-output-record` (*stream* **&optional** *record-type record* **&rest** *initargs*) **&body** *body* [Macro]

Creates a new output record of type RECORD-TYPE and then captures the output of BODY into the new output record, and inserts the new record into the current "open" output record assotiated with STREAM. If RECORD is supplied, it is the name of a variable that will be lexically bound to the new output record inside the body. INITARGS are CLOS initargs that are passed to MAKE-INSTANCE when the new output record is created. It returns the created output record. The STREAM argument is a symbol that is bound to an output recording stream. If it is `t`, *STANDARD-OUTPUT* is used.

`clim:with-output-to-output-record` (*stream* **&optional** *record-type record* **&rest** *initargs*) **&body** *body* [Macro]

Creates a new output record of type RECORD-TYPE and then captures the output of BODY into the new output record. The cursor position of STREAM is initially bound to (0,0) If RECORD is supplied, it is the name of a variable that will be lexically bound to the new output record inside the body. INITARGS are CLOS initargs that are passed to MAKE-INSTANCE when the new output record is created. It returns the created output record. The STREAM argument is a symbol that is bound to an output recording stream. If it is `t`, *STANDARD-OUTPUT* is used.

12 Command Processing

`define-command-table` *name* **&key** *inherit-from menu inherit-menu* [Macro]

`make-command-table` *name* **&key** *inherit-from inherit-menu* (*errorp* **t**) [Function]

By default command tables inherit from `global-command-table`. According to the CLIM~2.0 specification, a command table inherits from no command table if \nil\ is passed as an explicit argument to *inherit-from*. In revision~2.2 all command tables must inherit from `global-command-table`. McCLIM treats a \nil\ value of *inherit-from* as specifying `'(global-command-table)`.

Extensions

13 Output Protocol Extensions

`clim-extensions:medium-miter-limit` *medium* [Generic Function]

If LINE-STYLE-JOINT-SHAPE is :MITER and the angle between two consequent lines is less than the values return by `medium-miter-limit`, :BEVEL is used instead.

14 Output Recording Extensions

14.1 Standard classes

`standard-output-recording-stream` [Class]

This class is mixed into some other stream class to add output recording facilities. It is not instantiable.

15 Raster Image backend

Raster image backend includes a medium that supports:

- CLIM's medium protocol,
- CLIM's output stream protocol, and
- CLIM's Pixmap protocol.

Package `mcclim-raster-image` exports the following macros:

`mcclim-render:with-output-to-raster-image-stream` (*stream-var file-stream format* **&rest** *options*) **&body** *body* [Macro]

`mcclim-render:with-output-to-rgb-pattern` (*stream-var image* **&rest** *options*) **&body** *body* [Macro]

Within `body`, `stream-var` is bound to a stream that produces a raster image. This stream is suitable as a stream or medium argument to any CLIM output utility, such as `draw-line*` or `write-string`.

The value of `options` is a list consisting of alternating keyword and value pairs. These are the supported keywords:

- `:width` — specifies the width of the image. Its default value is 1000.
- `:height` — specifies the height of the image. Its default value is 1000.

`mcclim-render:with-output-to-raster-image-stream` (*stream-var file-stream format* **&rest** *options*) **&body** *body* [Macro]

An image describing the output to the `stream-var` stream will be written to the stream `file-stream` using the format `format`. `format` is a symbol that names the type of the image. Valid values are `:png`, `:jpg`, `:jpeg`, `tiff`, `tif`, `gif`, `pbm`, `pgm`, and `ppm`. Its default value is `:png`.

`mcclim-render:with-output-to-rgb-pattern` (*stream-var image* **&rest** *options*) **&body** *body* [Macro]

An image describing the output to the `stream-var` stream will be returned as an rgb-pattern (of class `climi::rgb-pattern`).

To run an example:

```
(ql:quickload :clim-examples)
(load "Examples/drawing-tests")
(clim-demo::run-drawing-tests)
```

16 PostScript Backend

16.1 Postscript Fonts

Font mapping is a cons, the car of which is the name of the font (FontName field in the AFM file), and the cdr is the size in points. Before establishing the mapping, an information about this font should be loaded with the function `load-afm-file`.

16.2 Additional functions

Package `clim-postscript` exports the following functions:

`load-afm-file` *afm-filename* [Function]

Loads a description of a font from the specified AFM file.

17 Drawing Two-Dimensional Images

17.1 Image Architecture

17.1.1 Images

Images are all rectangular arrangements of pixels. The type of a pixel depends on the exact type of the image. In addition, a pixel has a color which also depends on the exact type of the image. You can think of the color as an interpretation of the pixel value by the type of image.

The coordinate system of an image has (0,0) in its upper-left corner. The x coordinate grows to the right and the y coordinate downwards.

`image` [Protocol Class]

This class is the base class for all images.

`image-width` *image* [Generic Function]

`image-height` *image* [Generic Function]

This function returns the width and the height of the image respectively.

`image-pixels` *image* [Generic Function]

This function returns a two-dimensional array of pixels, whose element type depends on the exact subtype of the image.

`image-pixel` *image x y* [Generic Function]

This function returns the pixel at the coordinate indicated by the values of x and y. The type of the return value depends on the exact image type.

`(setf image-pixel)` *x y pixel image* [Generic Function]

Set the value of the pixel at the coordinate indicated by the values of x and y. The exact type acceptable for the pixel argument depends on the exact subtype of the image. If x or y are not within the values of the width and height of the image, an error is signaled.

`image-color` *image x y* [Generic Function]

This function returns the color value of the pixel indicated by the values of x and y. The exact type of the return value depends on the specific subtype of the image.

`(setf image-color)` *x y color image* [Generic Function]

Set the color value of the pixel at the coordinate indicated by the values of x and y. The exact type acceptable for the color argument depends on the exact subtype of the image. In addition, the exact color given to the pixel may be an approximation of the value of the color argument. For instance, if the image is a gray-level image, then the color given will correspond to the intensity value of the color argument. If x or y are not within the values of the width and height of the image, an error is signaled.

`spectral-image` [Protocol Class]

This class is a subclass of the image class. It is the root of a subhiearchy for manipulating images represented in various spectral formats, other than RGB. [This

subhierarchy will be elaborated later in the context of the color model of Strandh and Braquelaire].

`rgb-image` [Protocol Class]
This class is a subclass of the image class. It is the root of a subhierarchy for manipulating images whose pixel colors are represented as RGB coordinates. The function image-color always returns a value of type (unsigned-byte 24) for images of this type, representing three different intensity values of 0-255.

`truecolor-image` [Protocol Class]
This class is a subclass of the rgb-image class. Images of this class have pixel values of type (unsigned-byte 24). The pixel values directly represent RGB values.

`colormap-image` [Protocol Class]
This class is a subclass of the rgb-image class. Images of this class have pixel values that don't directly indicate the color of the pixel. The translation between pixel value and color may be implicit (as is the case of gray-level images) or explicit with a colormap stored in the image object.

`gray-level-image` [Protocol Class]
This class is a subclass of the colormap-image class. Images of this type have pixel values that implicitely represent a gray-level. The function pixel-color always returns an RGB value that corresponds to the identical intensities for red, green, and blue, according to the pixel value.

`gray-image-max-levels` *gray-level-image* [Generic Function]
This function returns the maximum number of levels of gray that can be represented by the image. The value returned by this function minus one would yield a color value of 255,255,255 if it were the value of a pixel.

`gray-image-max-level` *gray-level-image* [Generic Function]
This function returns the maximum level currently present in the image. This function may be very costly to compute, as it might have to scan the entire image.

`gray-image-min-level` *gray-level-image* [Generic Function]
This function returns the minimum level currently present in the image. This function may be very costly to compute, as it might have to scan the entire image.

`256-gray-level-image` [Class]
This class is a subclass of the gray-level-image class. Images of this type have pixels represented as 8-bit unsigned pixels. The function image-pixel always returns a value of type (unsigned-byte 8) for images of this type. The function gray-image-max-levels returns 256 for all instances of this class.

`binary-image` [Class]
This class is a subclass of the gray-level-image class. Images of this type have pixel values of type bit. The function image-pixel returns values of type bit when applied to an image of this type. The function pixel-color returns 0,0,0 for zero-valued bits and 255,255,255 for one-valued bits.

17.1.2 Utility Functions

`rotate-image` *image angle* **&key** (*antialias* **t**) [Generic Function]

`flip-image` *image ...* [Generic Function]

`translate-image` *image ...* [Generic Function]

`scale-image` *image ...* [Generic Function]
...

17.2 Reading Image Files

`read-image` *source* **&key** *type width height* [Generic Function]
Read an image from the source. The source can be a pathname designator (a string or a path), or a stream. The caller can supply a value for type, width, and height for sources that don't indicate these values. A value of nil for type means recognize the type automatically. Other values for type are :truecolor (an array of 3-byte color values) :256-gray-level (an array of 1-byte gray-level values) :binary (an array of bits).

`write-image` *image destination* **&key** (*type* **:pnm**) (*quality* **1**) [Generic Function]
Write the image to the destination. The destination can be a pathname designator (a string or a path), or a stream. Valid values of type are :pnm (pbm, pgm, or ppm according to the type of image), :png, :jpeg, (more...). The quality argument is a value from 0 to 1 and indicates desired image quality (for formats with lossy compression).

18 Fonts and Extended Text Styles

18.1 Extended Text Styles

McCLIM extends the legal values for the `family` and `face` arguments to `make-text-style` to include strings (in additional to the portable keyword symbols), as permitted by the CLIM spec, section 11.1.

Each backend defines its own specific syntax for these family and face names.

The CLX backend maps the text style family to the X font's *foundry* and *family* values, separated by a dash. The face is mapped to *weight* and *slant* in the same way. For example, the following form creates a text style for *-misc-fixed-bold-r-*-*-18-*-*-*-*-*-*-**:

```
(make-text-style "misc-fixed" "bold-r" 18)
```

In the GTK backend, the text style family and face are used directly as the Pango font family and face name. Please refer to Pango documentation for details on the syntax of face names. Example:

```
(make-text-style "Bitstream Vera Sans" "Bold Oblique" 54)
```

18.2 Listing Fonts

McCLIM's font listing functions allow applications to list all available fonts available on a `port` and create text style instances for them.

Example:

```
* (find "Bitstream Vera Sans Mono"
        (clim-extensions:port-all-font-families (clim:find-port))
        :key #'clim-extensions:font-family-name
        :test #'equal)
#<CLIM-GTKAIRO::PANGO-FONT-FAMILY Bitstream Vera Sans Mono>

* (clim-extensions:font-family-all-faces *)
(#<CLIM-GTKAIRO::PANGO-FONT-FACE Bitstream Vera Sans Mono, Bold>
 #<CLIM-GTKAIRO::PANGO-FONT-FACE Bitstream Vera Sans Mono, Bold Oblique>
 #<CLIM-GTKAIRO::PANGO-FONT-FACE Bitstream Vera Sans Mono, Oblique>
 #<CLIM-GTKAIRO::PANGO-FONT-FACE Bitstream Vera Sans Mono, Roman>)

* (clim-extensions:font-face-scalable-p (car *))
T

* (clim-extensions:font-face-text-style (car **) 50)
#<CLIM:STANDARD-TEXT-STYLE "Bitstream Vera Sans Mono" "Bold" 50>
```

`clim-extensions:font-family` [Class]

Class precedence list: `font-family`, `standard-object`, `slot-object`, `t`

The protocol class for font families. Each backend defines a subclass of font-family and implements its accessors. Font family instances are never created by user code. Use port-all-font-families to list all instances available on a port.

`clim-extensions:font-face` [Class]

Class precedence list: `font-face, standard-object, slot-object, t`

The protocol class for font faces Each backend defines a subclass of font-face and implements its accessors. Font face instances are never created by user code. Use font-family-all-faces to list all faces of a font family.

`clim-extensions:port-all-font-families` *port* **&key** *invalidate-cache* **&allow-other-keys** [Generic Function]

Returns the list of all `font-family` instances known by PORT. With INVALIDATE-CACHE, cached font family information is discarded, if any.

`clim-extensions:font-family-name` *font-family* [Generic Function]

Return the font family's name. This name is meant for user display, and does not, at the time of this writing, necessarily the same string used as the text style family for this port.

`clim-extensions:font-family-port` *font-family* [Generic Function]

Return the port this font family belongs to.

`clim-extensions:font-family-all-faces` *font-family* [Generic Function]

Return the list of all font-face instances for this family.

`clim-extensions:font-face-name` *font-face* [Generic Function]

Return the font face's name. This name is meant for user display, and does not, at the time of this writing, necessarily the same string used as the text style face for this port.

`clim-extensions:font-face-family` *font-face* [Generic Function]

Return the font family this face belongs to.

`clim-extensions:font-face-all-sizes` *font-face* [Generic Function]

Return the list of all font sizes known to be valid for this font, if the font is restricted to particular sizes. For scalable fonts, arbitrary sizes will work, and this list represents only a subset of the valid sizes. See font-face-scalable-p.

`clim-extensions:font-face-text-style` *font-face* **&optional** *size* [Generic Function]

Return an extended text style describing this font face in the specified size. If size is nil, the resulting text style does not specify a size.

19 Tab Layout

The tab layout is a composite pane arranging its children so that exactly one child is visible at any time, with a row of buttons allowing the user to choose between them.

See also the tabdemo.lisp example code located under Examples/ in the McCLIM distribution. It can be started using demodemo.

`clim-tab-layout:tab-layout` [Class]

Class precedence list: `tab-layout`, `sheet-multiple-child-mixin`, `basic-pane`, `sheet-parent-mixin`, `pane`, `standard-repainting-mixin`, `standard-sheet-input-mixin`, `sheet-transformation-mixin`, `basic-sheet`, `sheet`, `bounding-rectangle`, `standard-object`, `slot-object`, `t`

The abstract tab layout pane is a composite pane arranging its children so that exactly one child is visible at any time, with a row of buttons allowing the user to choose between them. Use `with-tab-layout` to define a tab layout and its children, or use the :`pages` argument to specify its contents when creating it dynamically using `make-pane`.

`clim-tab-layout:tab-layout-pane` [Class]

Class precedence list: `tab-layout-pane`, `tab-layout`, `sheet-multiple-child-mixin`, `basic-pane`, `sheet-parent-mixin`, `pane`, `standard-repainting-mixin`, `standard-sheet-input-mixin`, `sheet-transformation-mixin`, `basic-sheet`, `sheet`, `bounding-rectangle`, `standard-object`, `slot-object`, `t`

A pure-lisp implementation of the tab-layout, this is the generic implementation chosen by the CLX frame manager automatically. Users should create panes for type `tab-layout`, not `tab-layout-pane`, so that the frame manager can customize the implementation.

`clim-tab-layout:tab-page` [Class]

Class precedence list: `tab-page`, `standard-object`, `slot-object`, `t`

Instances of `tab-page` represent the pages in a `tab-layout`. For each child pane, there is a `tab-page` providing the page's title and additional information about the child. Valid initialization arguments are :`title`, :`pane` (required) and :`presentation-type`,:DRAWING-OPTIONS (optional).

`clim-tab-layout:with-tab-layout` (*default-presentation-type* **&rest** *initargs* **&key** *name* **&allow-other-keys**) **&body** *body* [Macro]

Return a `tab-layout`. Any keyword arguments, including its name, will be passed to `make-pane`. Child pages of the `tab-layout` can be specified using BODY, using lists of the form (`title` PANE &KEY PRESENTATION-TYPE DRAWING-OPTIONS `enabled-callback`). `default-presentation-type` will be passed as :`presentation-type` to pane creation forms that specify no type themselves.

`clim-tab-layout:tab-layout-pages` *tab-layout* [Generic Function]

Return all TAB-PAGEs in this tab layout, in order from left to right. Do not modify the resulting list destructively. Use the `setf` function of the same name to assign a new list of pages. The `setf` function will automatically add tabs for new page objects, remove old pages, and reorder the pages to conform to the new list.

`clim-tab-layout:tab-page-tab-layout` *tab-page* [Generic Function]
Return the `tab-layout` this page belongs to.

`clim-tab-layout:tab-page-title` *tab-page* [Generic Function]
Return the title displayed in the tab for this `page`. Use the `setf` function of the same name to set the title dynamically.

`clim-tab-layout:tab-page-pane` *tab-page* [Generic Function]
Return the CLIM pane this page displays. See also SHEET-TO-PAGE, the reverse operation.

`clim-tab-layout:tab-page-presentation-type` *tab-page* [Generic Function]
Return the type of the presentation used when this page's header gets clicked. Use the `setf` function of the same name to set the presentation type dynamically. The default is `tab-page`.

`clim-tab-layout:tab-page-drawing-options` *tab-page* [Generic Function]
Return the drawing options of this page's header. Use the `setf` function of the same name to set the drawing options dynamically. Note: Not all implementations of the tab layout will understand all drawing options. In particular, the Gtkairo backends understands only the :INK option at this time.

`clim-tab-layout:add-page` *page tab-layout* **&optional** *enable* [Function]
Add `page` at the left side of `tab-layout`. When `enable` is true, move focus to the new page. This function is a convenience wrapper; you can also push page objects directly into `tab-layout-pages` and enable them using (`setf` TAB-LAYOUT-ENABLED-PAGE).

`clim-tab-layout:remove-page` *page* [Function]
Remove `page` from its tab layout. This is a convenience wrapper around SHEET-DISOWN-CHILD, which can also be used directly to remove the page's pane with the same effect.

`clim-tab-layout:tab-layout-enabled-page` *tab-layout* [Generic Function]
The currently visible tab page of this tab-layout, or NIL if the tab layout does not have any pages currently. Use the `setf` function of the name to change focus to another tab page.

`clim-tab-layout:sheet-to-page` *sheet* [Function]
For a `sheet` that is a child of a tab layout, return the page corresponding to this sheet. See also `tab-page-pane`, the reverse operation.

`clim-tab-layout:find-tab-page-named` *name tab-layout* [Function]
Find the tab page with the specified `title` in `tab-layout`. Note that uniqueness of titles is not enforced; the first page found will be returned.

`clim-tab-layout:switch-to-page` *page* [Function]
Move the focus in page's tab layout to this page. This function is a one-argument convenience version of (`setf` TAB-LAYOUT-ENABLED-PAGE), which can also be called directly.

`clim-tab-layout:remove-page-named` *title tab-layout* [Function]
Remove the tab page with the specified `title` from `tab-layout`. Note that uniqueness of titles is not enforced; the first page found will be removed. This is a convenience wrapper, you can also use FIND-TAB-PAGE-NAMED to find and the remove a page yourself.

`clim-tab-layout:note-tab-page-changed` *layout page* [Generic Function]
This internal function is called by the `setf` methods for `tab-page-title` and -DRAWING-OPTIONS to inform the page's tab-layout about the changes, allowing it to update its display. Only called by the `tab-layout` implementation and specialized by its subclasses.

20 Drei

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

20.1 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 20.4.7 [Redisplay Protocol], page 71). 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.

20.1.1 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.

20.1.2 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.

20.2 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.

20.3 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.

20.4 Protocols

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

20.4.1 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.

20.4.1.1 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.

20.4.1.2 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.

20.4.1.3 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.

20.4.1.4 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.

20.4.1.5 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.

20.4.2 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.

20.4.3 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.

20.4.3.1 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.

20.4.4 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.

20.4.4.1 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.

20.4.4.2 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.

20.4.4.3 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.

20.4.4.4 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.

20.4.4.5 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)`.

20.4.5 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.

20.4.6 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 20.4.6.3 [Editing Protocol], page 71) lack this argument as well. There are no `forward-one-object` or `backward-one-object` functions.

20.4.6.1 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 20.4.6.2 [Motion Protocol], page 71). 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.

20.4.6.2 Motion Protocol

The concept of a *basic motion function* was introduced in Section 20.4.6 [Unit Protocol], page 69. 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`.

20.4.6.3 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 20.4.6.4 [Generator Macros], page 71).

20.4.6.4 Generator Macros

20.4.7 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`.

20.4.8 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.

20.4.8.1 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).

20.4.8.2 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.

20.4.8.3 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.

20.4.9 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.

20.4.9.1 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.

20.4.9.2 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.

20.5 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.

20.5.1 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.

20.5.2 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 20.5.3 [Examples Of Defining Drei Commands], page 80).

20.5.3 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 20.1.1 [Access Functions], page 52.

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.

20.5.4 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.

Utility Programs

21 Listener

22 Inspector

The inspector, called "Clouseau", is used for interactively inspecting objects. It lets you look inside objects, inspect slots, disassemble and trace functions, view keys and values in hash tables, and quite a few other things as well. It can be extended to aid in debugging of specific programs, similar to the way the Lisp printer can be extended with `print-object`.

22.1 Usage

22.1.1 Quick Start

To get up and running quickly with Clouseau:

1. With ASDF and McCLIM loaded, load the file
   `mcclim/Apps/Inspector/inspector.asd`.
2. Load Clouseau with:
   `(asdf:operate 'asdf:load-op :clouseau)`
3. Inspect an object with `(clouseau:inspector` *object*`)`. If you use a multithreaded Lisp implementation, you can also include the `:new-process` keyword argument. If it is `t`, then Clouseau is started in a seperate process. This should be relatively safe; it is even possible to have an inspector inspecting another running inspector.

22.1.2 The Basics

Once you inspect something, you will see a full representation of the object you are inspecting and short representations of objects contained within it. This short representation may be something like `#<STANDARD-CLASS SALAD-MIXIN>` or something as short as ". . .". To see these objects inspected more fully, left-click on them and they will be expanded. To shrink expanded objects, left-click on them again and they will go back to a brief form.

That's really all you need to know to get started. The best way to learn how to use Clouseau is to start inspecting your own objects.

22.1.3 Handling of Specific Data Types

Clouseau can handle numerous data types in different ways. Here are some handy features you might miss if you don't know to look for them:

22.1.3.1 Standard Objects

Standard objects have their slots shown, and by left-clicking on the name of a slot you can change the slot's value. You can see various slot attributes by middle clicking on a slot name.

22.1.3.2 Structures

Structures are inspected the same way as standard objects.

22.1.3.3 Generic Functions

You can remove methods from generic functions with the `Remove Method` command.

22.1.3.4 Functions

You can disassemble functions with the `Toggle Disassembly` command. If the disassembly is already shown, this command hides it.

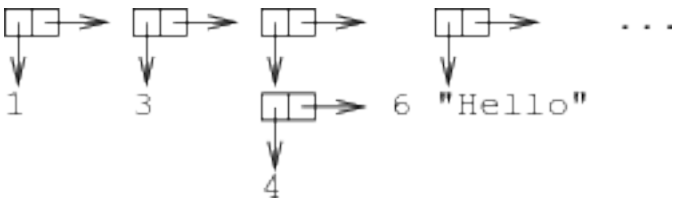
22.1.3.5 Symbols

If a symbol is fbound, you can use the `Trace` and `Untrace` commands to trace and untrace the function bound to it.

22.1.3.6 Lists and Conses

Lists and conses can be displayed in either the classic format (such as `(1 3 (4 . 6) "Hello" 42)`) or a more graphical cons-cell diagram format. The default is the classic format, but this can be toggled with the `Toggle Show List Cells` command.

The new cons cell diagram format looks like this:



22.2 Extending Clouseau

Sometimes Clouseau's built-in inspection abilities aren't enough, and you want to be able to extend it to inspect one of your own classes in a special way. Clouseau supports this, and it's fairly simple and straightforward.

Suppose that you're writing a statistics program and you want to specialize the inspector for your application. When you're looking at a sample of some characteristic of a population, you want to be able to inspect it and see some statistics about it, like the average. This is easy to do.

We define a class for a statistical sample. We're keeping this very basic, so it'll just contain a list of numbers:

```
(in-package :clim-user)
(use-package :clouseau)

(defclass sample ()
  ((data :initarg :data
         :accessor data
         :type list :initform '()))
  (:documentation "A statistical sample"))

(defgeneric sample-size (sample)
  (:documentation "Return the size of a statistical sample"))

(defmethod sample-size ((sample sample))
  (length (data sample)))
```

The `print-object` function we define will print samples unreadably, just showing their sample size. For example, a sample with nine numbers will print as `#<SAMPLE n=9>` We create such a sample and call it `*my-sample*`.

```
(defmethod print-object ((object sample) stream)
  (print-unreadable-object (object stream :type t)
    (format stream "n=~D" (sample-size object))))

(defparameter *my-sample*
  (make-instance 'sample
                 :data '(12.8 3.7 14.9 15.2 13.66
                         8.97 9.81 7.0 23.092)))
```

We need some basic statistics functions. First, we'll do sum:

```
(defgeneric sum (sample)
  (:documentation "The sum of all numbers in a statistical
sample"))

(defmethod sum ((sample sample))
  (reduce #'+ (data sample)))
```

Next, we want to be able to compute the mean. This is just the standard average that everyone learns: add up all the numbers and divide by how many of them there are. It's written $\overline{x}$

```
(defgeneric mean (sample)
  (:documentation "The mean of the numbers in a statistical
sample"))

(defmethod mean ((sample sample))
  (/ (sum sample)
     (sample-size sample)))
```

Finally, to be really fancy, we'll throw in a function to compute the standard deviation. You don't need to understand this, but the standard deviation is a measurement of how spread out or bunched together the numbers in the sample are. It's called s, and it's computed like this: $s = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(x_i - \overline{x})^2}$

```
(defgeneric standard-deviation (sample)
  (:documentation "Find the standard deviation of the numbers
in a sample. This measures how spread out they are."))

(defmethod standard-deviation ((sample sample))
  (let ((mean (mean sample)))
    (sqrt (/ (loop for x in (data sample)
                   sum (expt (- x mean) 2))
             (1- (sample-size sample))))))
```

This is all very nice, but when we inspect `*my-sample*` all we see is a distinctly inconvenient display of the class, its superclass, and its single slot, which we actually need to *click on* to see. In other words, there's a lot of potential being missed here. How do we take advantage of it?

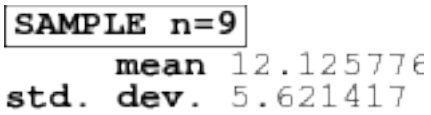
We can define our own inspection functions. To do this, we have two methods that we can define. To change how sample objects are inspected compactly, before they are clicked on, we can define an `inspect-object-briefly` method for our `sample` class. To change the full, detailed inspection of samples, we define `inspect-object` for the class. Both of these methods take two arguments: the object to inspect and a CLIM output stream. They are expected to print a representation of the object to the stream.

Because we defined `print-object` for the `sample` class to be as informative as we want the simple representation to be, we don't need to define a special `inspect-object-briefly` method. We should, however, define `inspect-object`.

```
(defmethod inspect-object ((object sample) pane)
  (inspector-table (object pane)
      ;; This is the header
      (format pane "SAMPLE n=~D" (sample-size object))
    ;; Now the body
    (inspector-table-row (pane)
      (princ "mean" pane)
      (princ (mean object) pane))
    (inspector-table-row (pane)
      (princ "std. dev." pane)
      (princ (standard-deviation object) pane))))
```

Here, we introduce two new macros. `inspector-table` sets up a box in which we can display our representation of the sample. It handles quite a bit of CLIM work for us. When possible, you should use it instead of making your own, since using the standard facilities helps ensure consistency.

The second macro, `inspector-table-row`, creates a row with the output of one form bolded on the left and the output of the other on the right. This gives us some reasonably nice-looking output:



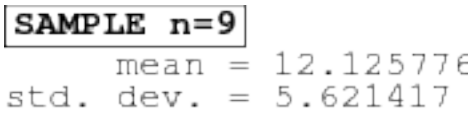
But what we really want is something more closely adapted to our needs. It would be nice if we could just have a table of things like $\overline{x} = 12.125776$ and have them come out formatted nicely. Before we attempt mathematical symbols, let's focus on getting the basic layout right. For this, we can use CLIM's table formatting.

```
(defmethod inspect-object ((object sample) pane)
  (inspector-table (object pane)
      ;; This is the header
      (format pane "SAMPLE n=~D" (sample-size object))
    ;; Now the body
    (inspector-table-row (pane)
      (princ "mean" pane)
      (princ (mean object) pane))
    (inspector-table-row (pane)
      (princ "std. dev." pane)
```

```
        (princ (standard-deviation object) pane))))
```

In this version, we define a local function `x=y` which outputs a row showing something in the form "label = value". If you look closely, you'll notice that we print the label with `princ` but we print the value with `inspect-object`. This makes the value inspectable, as it should be.

Then, in the `inspector-table` body, we insert a couple of calls to `x=y` and we're done. It looks like this:



Finally, for our amusement and further practice, we'll try to get some mathematical symbols—in this case we'll just need $\overline{x}$. We can get this by printing an italic x and drawing a line over it:

```
(defun xbar (stream)
  "Draw an x with a bar over it"
  (with-room-for-graphics (stream)
    (with-text-face (stream :italic)
      (princ #\x stream)
      (draw-line* stream 0 0
                  (text-style-width *default-text-style*
                                    stream) 0))))

(defmethod inspect-object ((object sample) pane)
  (flet ((x=y (x y)
           (formatting-row (pane)
             (formatting-cell (pane :align-x :right)
               ;; Call functions, print everything else in italic
               (if (functionp x)
                   (funcall x pane)
                   (with-text-face (pane :italic)
                     (princ x pane))))
             (formatting-cell (pane) (princ "=" pane))
             (formatting-cell (pane)
               (inspect-object y pane)))))
    (inspector-table (object pane)
        ;; This is the header
        (format pane "SAMPLE n=~D" (sample-size object))
      ;; Now the body
      (x=y #'xbar (mean object))
      (x=y #\S (standard-deviation object)))))
```

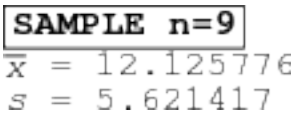
Finally, to illustrate the proper use of `inspect-object-briefly`, suppose that we want the "n=9" (or whatever the sample size n equals) part to have an itlicised n. We can fix this easily:

```
(defmethod inspect-object-briefly ((object sample) pane)
```

```
(with-output-as-presentation (pane object 'sample)
  (with-text-family (pane :fix)
    (print-unreadable-object (object pane :type t)
      (with-text-family (pane :serif)
        (with-text-face (pane :italic)
          (princ "n" pane)))
      (format pane "=~D" (sample-size object))))))
```

Notice that the body of `inspect-object-briefly` just prints a representation to a stream, like `inspect-object` but shorter. It should wrap its output in `with-output-as-presentation`. `inspect-object` does this too, but it's hidden in the `inspector-table` macro.

Our final version looks like this:



For more examples of how to extend the inspector, you can look at `inspector.lisp`.

22.3 API

The following symbols are exported from the `clouseau` package:

`inspector` *object &key new-process* [Function]
Inspect `object`. If `new-process` is `t`, Clouseau will be run in a new process.

`inspect-object` *object pane* [Generic Function]
Display inspected representation of `object` to the extended output stream `pane`. This requires that `*application-frame*` be bound to an inspector application frame, so it isn't safe to use in other applications.

`inspect-object-briefly` *object pane* [Generic Function]
A brief version of `inspect-object`. The output should be short, and should try to fit on one line.

`define-inspector-command` *name args* **&rest** *body* [Generic Function]
This is just an inspector-specific version of `define-command`. If you want to define an inspector command for some reason, use this.

`inspector-table` (*object pane*) *header* *body body* [Macro]
Present `object` in tabular form on `pane`, with `header` evaluated to print a label in a box at the top. `body` should output the rows of the table, possibly using `inspector-table-row`.

`inspector-table-row` (*pane*) *left right* [Macro]
Output a table row with two items, produced by evaluating `left` and `right`, on `pane`. This should be used only within `inspector-table`.

When possible, you should try to use this and `inspector-table` for consistency, and because they handle quite a bit of effort for you.

23 Debugger

The debugger is used for interactively inspecting stack frame when the unhandled conditions are signalled. Given high enough debug settings it lets you inspecting frame local variables, evaluating code in it, examining backtrace and choosing available restarts.

23.1 Debugger usage

To get up and running quickly with Debugger:

1. With Quicklisp loaded, invoke in repl:
   ```
   (ql:quickload 'clim-debugger)
   ```
2. Run simple test condition:
   ```
   (clim-debugger:with-debugger (error "test"))
   ```

Debugger is highly inspired by Slime and uses Swank to gain portability across implementations. Module is still under development and some details may change in the future.

Selecting frame with a pointer switches its details and marks it active. Each locale value may be inspected by selection with mouse pointer. Active frame is distinguished from others with red color. `Eval in frame` command evaluates expression in the active frame.

23.2 Keyboard shortcuts

Warning: these key accelerators may change in the future.

'`M-m`'	Show more frames
'`M-p`'	Mark previous frame active
'`M-n`'	Mark next frame active
'`M-e`'	Eval in active frame
'`TAB`'	Toggle active frame details
'`M-[0-9]`'	Invoke nth restart
'`M-q`'	Quit debugger

23.3 Debugger API

`debugger` *condition me-or-my-encapsulation* [Function]

Starts debugger with supplied condition. Second argument should be supplied by an underlying implementation allowing to encapsulate or supply different debugger for recursive debugger calls.

`with-debugger` **&body** *body* [macro]

Wraps the code in `body` to invoke clim debugger when condition is signalled (binds `*debugger-hook*` to `#'debugger`).

Auxiliary Material

24 Glossary

Direct mirror

A *mirror* of a sheet which is not shared with any of the ancestors of the sheet. All grafted McCLIM sheets have mirrors, but not all have direct mirrors. A McCLIM sheet that does not have a direct mirror uses the direct mirror of its first ancestor having a direct mirror for graphics output. Asking for the direct mirror of a sheet that does not have a direct mirror returns nil.

Whether a McCLIM sheet has a direct mirror or not, is decided by the frame manager. Some frame managers may only allow for the graft to be a mirrored sheet. Even frame managers that *allow* hierarchical mirrors may decide not to allocate a direct mirror for a particular sheet. Although sheets with a direct mirror must be instances of the class mirrored-sheet-mixin, whether a McCLIM sheet has a direct mirror or not is not determined statically by the class of a sheet, but dynamically by the frame manager.

Mirror

A device window such as an X11 window that parallels a *sheet* in the CLIM *sheet hierarchy*. A *sheet* having such a *direct* mirror is called a *mirrored sheet*. When *drawing functions* are called on a *mirrored sheet*, they are forwarded to the host windowing system as drawing commands on the *mirror*.

CLIM *sheet*s that are not mirrored must be *descendents* (direct or indirect) of a *mirrored sheet*, which will then be the *sheet* that receives the drawing commands.

Mirrored sheet

A *sheet* in the CLIM *sheet hiearchy* that has a direct parallel (called the *direct mirror*) in the host windowing system. A mirrored sheet is always an instance of the class `mirrored-sheet-mixin`, but instances of that class are not necessarily mirrored sheets. The sheet is called a mirrored sheet only if it currently has a direct mirror. There may be several reasons for an instance of that class not to currently have a direct mirror. One is that the sheet is not *grafted*. Only grafted sheets can have mirrors. Another one is that the *frame manager* responsible for the look and feel of the sheet hierarchy may decide that it is inappropriate for the sheet to have a direct mirror, for instance if the underlying windowing system does not allow nested windows inside an application, or that it would simply be a better use of resources not to create a direct mirror for the sheet. An example of the last example would be a stream pane inside a the *viewport* of a *scroller pane*. The graphics objects (usually text) that appear in a stream pane can have very large coordinate values, simply because there are many lines of text. Should the stream pane be mirrored, the coordinate values used on the mirror may easily go beyond what the underlying windowing system accepts. X11, for instance, can not handle coordinates greater than 64k (16 bit unsigned integer). By not having a direct mirror for the stream pane, the coordinates will be translated to those of the (not necessarily direct) mirror of the *viewport* before being submitted to the windowing system, which gives more reasonable coordinate values.

It is important to realize the implications of this terminology. A mirrored sheet is therefore not a sheet that has a mirror. All grafted sheets have mirrors. For the sheet to be a mirrored sheet it has to have a *direct* mirror. Also, a call to `sheet-mirror` returns a mirror for all grafted sheets, whether the sheet is a mirrored sheet or not. A call to `sheet-direct-mirror`, on the other hand, returns nil if the sheet is not a mirrored sheet.

Mirror transformation

The transformation that transforms coordinates in the coordinate system of a mirror (i.e. the native coordinates of the mirror) to native coordinates of its parent in the underlying windowing system. On most systems, including X, this transformation will be a simple translation.

Native coordinates

Each mirror has a coordinate system called the native coordinate system. Usually, the native coordinate system of a mirror has its origin in the upper-left corner of the mirror, the x-axis grows to the right and the y-axis downwards. The unit is usually pixels, but the frame manager can impose a native coordinate system with other units, such as millimeters.

The native coordinate system of a sheet is the native coordinate system of its mirror (direct or not). Thus, a sheet without a direct mirror has the same native coordinate system as its parent. To obtain native coordinates of the parent of a mirror, use the *mirror transformation*.

Native region

The native region of a sheet is the intersection of its region and the sheet region of all of its parents, expressed in the *native coordinates* of the sheet.

Potentially visible area

A bounded area of an otherwise infinte drawing plane that is visible unless it is covered by other visible areas.

Sheet coordinates

The coordinate system of coordinates obtained by application of the *user transformation*.

Sheet region

The *region* of a sheet determines the visible part of the drawing plane. The dimensions of the sheet region are given in *sheet coordinates*. The location of the visible part of a sheet within its *parent sheet* is determined by a combination of the *sheet transformation* and the position of the sheet region.

For instance, assuming that the sheet region is a rectangle with its upper-left corner at (2, 1) and that the sheet transformation is a simple translation (3, 2). Then the origin of the *sheet coordinate system* is at the point (3, 2) within the *sheet coordinate system* of its *parent sheet*. The origin of its the coordinate system is not visible, however, because the visible region has its upper-left corner at (2, 1) in the *sheet coordinate system*. Thus, the visible part will be a rectangle whose upper-left corner is at (5, 3) in the *sheet coordinate system* of the *parent sheet*.

Panes and gadgets alter the region and *sheet transformation* of the underlying sheets (panes and gadgets are special kinds of sheets) to obtain effects such as scrolling, zooming, coordinate system transformations, etc.

Sheet transformation

The transformation used to transform *sheet coordinates* of a sheet to *sheet coordinates* of its *parent sheet*. The sheet transformation determine the position, shape, etc. of a sheet within the coordinate system of its parent.

Panes and gadgets alter the transformation and *sheet region* of the underlying sheets (panes and gadgets are special kinds of sheets) to obtain effects such as scrolling, zooming, coordinate system transformations, etc.

User Clipping region

A *clipping region* used to limit the effect of *drawing functions*. The user *clipping region* is stored in the *medium*. It can be altered either by updating the *medium*, or by passing a value for the :clipping-region *drawing option* to a *drawing function*.

User Coordinates

The coordinate system of coordinates passed to the *drawing functions*.

User Transformation

A transformation used to transform *user coordinates* into *sheet coordinates*. The user transformation is stored in the *medium*. It can be altered either by updating the *medium*, or by passing a value for the :transformation *drawing option* to a *drawing function*.

Visible area

25 Development History

Mike McDonald started developing McCLIM in 1998. His initial objective was to be able to run the famous "address book" demo, and to distribute the first version when this demo ran. With this in mind, he worked "horizontally", i.e., writing enough of the code for many of the chapters of the specification to be able to run the address book example. In particular, Mike wrote the code for chapters 15 (Extended Stream Output), 16 (Output Recording), and 28 (Application Frames), as well as the code for interactor panes. At the end of 1999, Mike got too busy with other projects, and nothing really moved.

Also in 1998, Gilbert Baumann started working "vertically", writing a mostly-complete implementation of the chapters 3 (Regions) and 5 (Affine Transformations). At the end of 1999, he realized that he was not going to be able to finish the project by himself. He therfore posted his code to the free-CLIM mailing list. Gilbert's code was distributed according to the GNU Lesser General Public Licence (LGPL).

Robert Strandh picked up the project in 2000, starting from Gilbert's code and writing large parts of chapters 7 (Properties of Sheets) and 8 (Sheet Protocols) as well as parts of chapters 9 (Ports, Grafts, and Mirrored Sheets), 10 (Drawing Options), 11 (Text Styles), 12 (Graphics), and 13 (Drawing in Color).

In early 2000, Robert got in touch with Mike and eventually convinced him to distribute his code, also according to the LGPL. This was a major turning point for the project, as the code base was now sufficiently large that a number of small demos were actually running. Robert then spent a few months merging his code into that produced by Mike.

Arthur Lemmens wrote the initial version of the code for the gadgets in june of 2000.

Bordeaux students Iban Hatchondo and Julien Boninfante were hired by Robert for a 3-month summer project during the summer of 2000. Their objective was to get most of the pane protocols written (in particular space composition and space allocation) as well as some of the gadgets not already written by Arthur, in particular push buttons. The calculator demo was written to show the capabilities of their code.

In July of 2000, Robert invited Gilbert to the LSM-2000 metting in Bordeaux (libre software meeting). This meeting is a gathering of developers of free software with the purpose of discussing strategy, planning future projects, starting new ones, and working on existing ones. The main result of this meeting was that Gilbert managed to merge his code for regions and transformations into the main code base written by Mike, Robert, Iban, and Julien. This was also a major step towards a final system. We now had one common code base, with a near-complete implementation of regions, transformations, sheet protocols, ports, grafts, graphics, mediums, panes, and gadgets.

Meanwhile, Mike was again able to work on the project, and during 2000 added much of the missing code for handling text interaction and scrolling. In particular, output recording could now be used to redisplay the contents of an interactor pane. Mike and Robert also worked together to make sure the manipulation of sheet transformations and sheet regions as part of scrolling and space-allocation respected the specification.

Robert had initially planned for Iban and Julien to work on McCLIM for their fifth-year student project starting late 2000 and continuing until end of march 2001. For reasons beyond his control, however, he was forced to suggest a different project. Thus, Iban and Julien, together with two other students, were assigned to work on Gsharp, an interactive

score editor. Gsharp was the original reason for Robert to start working on CLIM as he needed a toolkit for writing a graphical user interface for Ghsarp. The lack of a freely-available version of a widely-accepted toolkit such as CLIM made him decide to give it a shot. Robert's idea was to define the student project so that a maximum of code could be written as part of McCLIM. The result was a complete rewrite of the space-allocation and space-composition protocols, and many minor code snippets.

As part of the Gsharp project, Robert wrote the code for menu bars and for a large part of chapter 27 (Command Processing).

Julien was hired for six months (April to September of 2001) by Robert to make major progress on McCLIM. Julien's first task was to create a large demo that showed many of the existing features of McCLIM (a "killer app"). It was decided to use Gsharp since Julien was already familiar with the application and since it was a sufficiently complicated application that most of the features would be tested. An additional advantage of a large application was to serve as a "smoke test" to run whenever substantial modifications to the code base had been made. As part of the Gsharp project, Julien first worked on adding the possibility of using images as button labels.

Early 2001, Robert had already written the beginning of a library for manipulating 2-dimensional images as part of McCLIM. A group of four fourth-year students (Gregory Bossard, Michel Cabot, Cyrille Dindart, Lionel Vergé) at the university of Bordeaux was assigned the task of writing efficient code for displaying such images subject to arbitrary affine transformations. This code would be the base for drawing all kinds of images such as icons and button labels, but also for an application for manipulating document images. The project lasted from January to May of 2001.

Another group of four fourth-year students (Loïc Lacomme, Nicolas Louis, Arnaud Rouanet, Lionel Salabartan) at the university of Bordeaux was assigned the task of writing a file-selector gadget presented as a tree of directories and files, and with the ability to open and close directories, to select files, etc. The project lasted from January to May of 2001.

One student in particular, Arnaud Rouanet started becoming interested in the rest of CLIM as well. During early 2001, he fixed several bugs and also added new code, in particular in the code for regions, graphics, and clx-mediums.

Arnaud and Lionel were hired by Robert for the summer of 2001 to work on several things. In particular, they worked on getting output recording to work and wrote CLIM-fig, a demo that shows how output recording is used. They also worked on various sheet protocols, and wrote the first version of the PostScript backend.

Alexey Dejneka joined the project in the summer of 2001. He wrote the code for table formatting, bordered output and continued to develop the PostScript output facility.

In the fall of 2001 Tim Moore became interested in the presentation type system. He implemented presentation type definition and presentation method dispatch. Wanting to see that work do something useful, he went on to implement present and accept methods, extended input streams, encapsulating streams, and the beginnings of input editing streams. In the spring of 2002 he wrote the core of Goatee, an Emacs-like editor. This is used to implement CLIM input editing.

Brian Spilsbury became involved towards the beginning of 2001. His motivation for getting involved was in order to have internationalization support. He quickly realized that the first step was to make SBCL and CMUCL support Unicode. He therefore worked to

make that happen. So far (summer 2001) he has contributed a number of cosmetic fixes to McCLIM and also worked on a GTK-like gadget set. He finally started work to get the OpenGL backend operational.

Concept Index

Panes order

A

B

C

D

E

G

I

L

M

N

O

P

S

T

U

V

W

Variable Index

*

D

E

Function And Macro Index

(

A

C

D

E

F

G

I

L

M

R

S

T

W