McCLIM User's Manual

The Users Guide

and

API Reference

Table of Contents

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.

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.

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.

1 User manual

1.1 Building McCLIM

1.1.1 Examples and 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)`.

The easiest way to try this is to use *Quicklisp* library manager. Assuming that this is already setup, trying out the demos is straightforward:

```
(ql:quickload 'clim-examples)
(clim-demo:demodemo)
```

Alternatively, for the more courageous (which requires separately downloading dependencies and setting a local repository), `asdf` also works well starting from the McCLIM source code directory.

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

Available demos may be explored by a programmer in `Examples` directory.

1.1.2 Applications

Additionally McCLIM has a few bundled applications:

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

```
(asdf:load-system 'clim-listener)
(clim-listener:run-listener)
```

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

```
(asdf:load-system 'clouseau)
(clouseau:inspector clim:+indian-red+)
```

`Apps/Debugger`
: Common Lisp debugger implemented in McCLIM. It uses a portable debugger interface for `sldb` (part of Slime project). System name is `clim-debugger`.

```
(asdf:load-system 'clim-debugger)
(clim-debugger:with-debugger
 (break "simple-break"))
```

`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`.

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

1.2 The first application

1.2.1 A bit of terminology

CLIM was developed before the GUI toolkits widely used at the moment. Qt, GTK and others appeared much later than CLIM and the difference of terminology reflects this.

A CLIM application is made up of a hierarchy of an *application frame*, *panes* and *gadgets* (gadgets are special kinds of panes):

'`application frame`'
: An *application frame* is what would usually be called an application.

'`panes`'
: At a very high level, panes describe an application frame's visual building blocks: a side bar, a menu bar, a table displaying a list of items, a text input are all panes. They can be used by application programmers to compose the top-level user interface of their applications, as well as auxiliary components such as menus and dialogs. In addition, panes can be more abstract such as layout panes such as `hbox`, `vbox` to arrange other panes horizontally or vertically, etc.

'`gadgets`'
: *gadgets* correspond to what other toolkits call *widgets* and *control*. Frequently used CLIM gadgets are `button`s, `slider`s, etc.

1.2.2 How CLIM applications produce output

Although it is easy to imagine panes in term of their appearance on screen, they are much richer: they are actually the series of operations that produces that appearance. They are not only the end product visible on a screen, but they contain all the step-by-step information that led to that representation.

More precisely, CLIM panes record the series of operations that produces that generates an output. 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, each more sophisticated:

‘`Simple application`’

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.

‘`Application with a static display function`’

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.

‘`Application with a dynamic display function`’

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.

‘`Application with an incremental static display function`’

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.

'`Programmer does it all`'

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.

1.2.3 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.

1.2.4 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.

1.2.5 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 :my-first-app
  ;; Imports the appropriate CLIM library
  (:use :clim :clim-lisp)

  ;; The package will only export a function to run the app
  (:export run-my-first-app))

;; Good practice
(in-package :my-first-app)

;; Definition of the structure of a minimum app
(define-application-frame my-first-clim-app ()
  ()

  ;; This app only has 1 pane
  (:panes
```

```
    (my-interactor :interactor
                   :height 400
                   :width 600))

  ;; :layouts section describes how the pane is positioned inside
  ;; the application frame.
  ;; With 1 pane, no point getting complicated, Default is fine...
  (:layouts
    (my-default my-interactor)))

;; Now that the structure of the app is defined, need a function
;; to launch an instance of this app. (The user could run
;; several instances of the same app.)
(defun run-my-first-app ()
  (run-frame-top-level (make-application-frame 'my-first-clim-app)))
```

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 `run-my-first-app`.

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 `my-first-clim-app`, 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.

1.2.6 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)`.

1.2.7 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`.

Let's consider an improved example, where for sake of efficiency the *my-* names have been replaced by shorter versions:

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

(defpackage :app
  (:use :clim :clim-lisp)
  (:export run-app))

(in-package :app)

(define-application-frame superapp ()
  ()
  (:pointer-documentation t)
  (:panes

   ;; Let's add an additional pane
   (app :application

        ;; When should this pane be displayed in the command loop.
        ;; Note that the refresh is pane-specific, not
        ;; application-wide.
        :display-time nil
        :height 400
        :width 600)
```

```
   (int :interactor
        :height 200
        :width 600))

  (:layouts

   ;; This time we explicitly specify that the 2 defined panes
   ;; should be stacked vertically.
   (default (vertically ()
              app int))))

;;
;; Let's also define commands that will act on the application.
;;

;; How to leave the application.
;; Note the '-superapp-' part of the command definition, coming from
;; the name of the application frame.
(define-superapp-command (com-quit :name t) ()
  (frame-exit *application-frame*))


;; This is an additional command that will be used in the next
;; example, so it's content is not important. However, it is useful
;; to describe some aspect of the command loop. See below.
(define-superapp-command (com-parity :name t) ((number 'integer))
  (format t "~a is ~a~%" number
          (if (oddp number)
              "odd"
              "even")))


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

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.

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 1.1 shows what ought to be visible on the screen.

Figure 1.1

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.

1.2.8 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 ()

  ;; New addition of a slot to the application frame which
  ;; defines a application-specific slot.

  ;; The slot is simply a number.
  ((currrent-number :initform nil
                    :accessor current-number))

  ;; The rest of the application frame is unchanged.
  (: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))))

;; This is the function that will display the pane app.
;; Simply prints the number of the application frame slot
;; and whether it is odd or even.
;; Note that the print stream of format is pane.
(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")))))

(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))


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

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.

1.3 Using 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))))

;; As usual, the displaying code relates to a pane, not the application frame.
(defun display-app (frame pane)

  (loop
     ;; taking items one-by-one from the frame slot 'numbers'
     for current-element in (numbers frame)

     ;; and increasing line-by-line
     for line from 0

     ;; prints a star if the cursor is on that line
     ;; (Note that here, there is no incremental redisplay. The output
     ;; record of the star will be printed at each call of the display
     ;; function -- that is at each iteration of the command loop.)
     do (princ (if (= (cursor frame) line) "*" " ") pane)

     ;; and incrementally updates the rendering instructions of the
     ;; number on that line
     ;; (Note that 'numbers' was defined as a list of lists, each
     ;; sublist holding an individual number. The reason for that is
     ;; explained below, but this is why (car current-element) is
     ;; needed.)
     do (updating-output (pane :unique-id   current-element
                               :id-test     #'eq
                               :cache-value (car current-element)
                               :cache-test  #'eql)
          (format pane "~a~%" (car current-element)))))


;;
;; Command definitions
;;

;; increase the value of the number on the current line
(define-superapp-command (com-add :name t) ((number 'integer))
  (incf (car (elt (numbers *application-frame*)
                  (cursor *application-frame*)))
        number))

;; move the cursor one line down (increasing the cursor position),
;; looping back to the beginning if going too far
```

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

;; move the cursor one line up
(define-superapp-command (com-previous :name t) ()
  (decf (cursor *application-frame*))
  (when (minusp (cursor *application-frame*))
    (setf (cursor *application-frame*)
          (1- (length (numbers *application-frame*))))))

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


;; Exported function to launch an instance of the application frame
(defun app-main ()
  (run-frame-top-level (make-application-frame 'superapp)))
```

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.

1.4 Using presentation types

1.4.1 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.

1.4.2 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".

1.5 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.

1.6 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.

`clim:find-command-table` *name* **&key** (*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.

1.7 Using menu bar

Menu bar has become essential part of every GUI system, including McClim. Ideally, McClim should try to use the menu bar provided by host window system via McClim backends, but the current `clx-backend` doesn't supports native menu bars. That's why It has some quirks of its own, like you need to keep mouse button pressed while accessing the sub-menus.

1.7.1 Creating Menu bar

McCLIM makes creating menu bar quite easy.

```
(clim:define-application-frame foo ()
  ;; ...
  (:menu-bar t)
  ;; ...
  )
```

The only argument for `:menu-bar` can be:

'`T (default)`'
: McCLIM will provide the menu bar. Later, when you start defining commands, you can provide a `(:menu t)` argument to command definition that will add this command to menu bar.

'`NIL`'
: McCLIM won't provide the menu bar.

'`command-table`'
: If you provide a named command table as argument, that command table is used to provide the menu bar (See Section 1.6 [Using command tables], page 21).

To add a sub-menu to menu bar, you need to change the type of menu-item from `:command` to `:menu` (which requires another `command-table` as argument) what is described in the next section.

1.7.2 Modifying Menu bar

Menu bar can be changed anytime by changing `command-table` associated with the current `frame`.

```
(setf (frame-command-table *application-frame*)
      new-command-table)
```

Example above changes menu bar of `*application-frame*` by replacing current `command-table` (accessible with `frame-command-table` function) with `new-command-table`.

Modifying menu items of command table

`clim:add-menu-item-to-command-table` *command-table string type value* **&rest** *args* **&key** *documentation after keystroke text-style errorp* [Function]

Adds menu item to the command table.

Function arguments:

'`command-table`'
: Command table to which we want to add the menu item.

'`string`'
: Name of the menu item as it will appear on the menu bar. Its character case is ignored e.g. you may give it `file` or `FILE` but it will appear as `File`.

'`type and value`'
: type can be one of `:command`, `:function`, `:menu` and `:divider`. Value of `value` depends on `type`:

  '`:command`'
  : `value` must be a command or a cons of command name and it's arguments. If you omit the arguments McCLIM will prompt for them.

  '`:function`'
  : `value` must be a function having indefinite extent that, when called, returns a command. Function must accept two arguments, the gesture (keyboard or mouse press event) and a `numeric argument`.

  '`:menu`'
  : `value` must be another command table. This type is used to add sub-menus to the menu.

  '`:divider`'
  : `value` is ignored and `string` is used as a divider string. Using `|` as string will make it obvious to users that it is a divider.

'`documentation`'
: You can provide the documentation (for non-obvious menu items) which will be displayed on pointer-documentation pane (if you have one).

'`after (default :end)`'
: This determines where item will be inserted in the menu. The default is to add it to the end. Other values could be `:start`, `:sort` (add in alphabetical order) or `string` which is name of existing menu-item to add after it.

'`keystroke`'
: If keystroke is supplied, it will be added to comand tables keystroke accelerator table. Value must be a keyboard gesture name e.g. `(:s :control)` for `Control + s`.

'`text-style`'
: Either a text style spec or `NIL`. It is used to indicate that the command menu item should be drawn with the supplied text style in command menus.

'`error-p`'
: If `T`, adding the existing item to the menu will signal error. If `NIL`, it will overwrite the existing item in the command table.

To remove items from command table, following function is used:

`clim:remove-menu-item-from-command-table` *command-table string* **&key** *errorp* [Function]

Removes item from the `command-table`.

Where `command-table` is command-table-designator and `string` is menu item's name (it is case-insensitive). You can provide `:error-p nil` to suppress the error if item is not in the command-table.

Note that both of above functions *does not* automatically update the menu bar. For that you need to replace existing `frame-command-table` with modified command table using `setf`. One way to do this is use `let` to create the copy of `frame-command-table`, modify it and at the end call `setf` to replace the original.

2 Reference manual

2.1 Concepts

2.1.1 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*.

2.1.2 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.

2.2 Sheet hierarchies

CLIM sheets are organized into a hierarchy. Each sheet has a sheet transformation and a sheet region. The sheet tranformation determines how coordinates in the sheet's own coordinate system get translated into coordinates in the coordinate system of its parent.

The sheet region determines the *potentially visible area* of the otherwise infinite drawing plane of the sheet. The sheet region is given in the coordinate system of the sheet.

In McCLIM, every grafted sheet has a *native transformation*. The native transformation is used by drawing functions to translate sheet coordinates to *native coordinates*, so that drawing can occur on the (not necessarily immediate) mirror of the sheet. It would therefore be enough for sheets that support the *output protocol* to have a native transformation. However, it is easier to generalize it to all sheets, in order to simplify the programming of the computation of the native transformation. Thus, in McCLIM, even sheets that are mute for output have a native transformation.

In McCLIM, every grafted sheet also has a *native region*. The native region is intersection the sheet region and the region of all of its ancestors, except that the native region is given in *native coordinates*, i.e. the coordinates obtained after the application of the *native transformation* of the sheet.

2.2.1 Computing the native transformation

2.2.2 Computing the native region

2.2.3 Moving and resizing sheets and regions

2.2.4 Scrolling

2.3 Drawing functions

2.3.1 Windowing system drawing

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.

2.3.2 CLIM drawing

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.

2.4 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.

2.4.1 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`.

[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.

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.

2.4.2 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 2.4.1 [Creating panes], page 27, 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 2.4.1 [Creating panes], page 27.

2.4.3 Redisplaying panes

Recall that *redisplay* refers to the creation of the output history of a pane. There are two typical ways of creating this output history:

- The application maintains some data structure that needs to be reflected in the text and graphics of the pane. In this case, a pane of type `application-pane` is typically used, and the default value of the `:display-time` option is used, which means that some kind of application-supplied *display function* is executed at the end of each iteration of the command loop. In this situation, the output history is either recomputed from

scratch in each iteration, or the programmer can use the *incremental redisplay* facility to reuse some of the existing output records in the history.

- The application does not keep any data structure, and instead generates output incrementally, either as a result of some user action, or of some data arriving from an external source. In this case, the `:display-time` option is either going to be `t` or `nil`. With both of these options, the output history is maintained intact after each iteration of the command loop. Instead, when user actions are issued, more output records are simply added to the existing output history.

For the second possibility, the pane is never redisplayed. Instead, the action of updating the pane contents is referred to as *replaying* the output history. The remainder of this section is entirely dedicated to the *redisplay* action.

It is occasionally necessary for the application to redisplay a pane explicitly, as opposed to letting the command loop handle it. For example, if the application data structure is updated in some way, but this update is not the result of a command, then after such an update, the redisplay function needs to be executed explicitly. Such an update could be the result of a timer event, or of communication with an external process.

`clim:redisplay-frame-pane` *frame pane* **&key** *force-p* [Generic Function]

Calling this generic function causes an immediate redisplay of *pane*. When *force-p* is false and the incremental redisplay facility is in use for *pane*, then output records are reused as appropriate. Supplying a true value for *force-p* causes the entire output history to be recomputed from scratch.

Notice that this function does not check whether the pane has been marked to need redisplay, as indicated by a call to the generic function `pane-needs-redisplay`. It results in an unconditional redisplay of *pane*.

`clim:redisplay-frame-panes` *frame* **&key** *force-p* [Generic Function]

Calling this generic function causes an immediate redisplay of all the panes of *frame* that are visible in the current layout. This function simply calls `redisplay-frame-pane` for each visible pane of *frame*.

Again, notice that no check is being made as to whether the visible panes have been marked as needing redisplay. This function calls `redisplay-frame-pane` unconditionally for each visible pane, and since `redisplay-frame-pane` redisplays the pane unconditionally, it follows that all visible panes are unconditionally redisplayed.

Also notice that the implication of this unconditional behavior on the part of `redisplay-frame-panes` means that this is not the function called by the standard command loop. The standard command loop only redisplays panes that have been marked as needing redisplay, though when the value of the `:display-time` option is `:command-loop` for some pane, then it is always marked as needing redisplay in each iteration of the command loop.

2.4.4 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.

2.4.4.1 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.

2.4.4.2 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.

2.4.4.3 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.

2.5 Output Protocol

`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.

`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 clim: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* [Generic Function]
x-offset y-offset
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` *function record* **&optional** *x-offset* [Function]
y-offset **&rest** *function-args*
Maps over all of the children of the RECORD, calling FUNCTION on each one. It is a function of one or more arguments and called with all of FUNCTION-ARGS as APPLY arguments.

`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.

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

Class precedence list: `standard-output-recording-stream, output-recording-stream, standard-object, slot-object, t`

Slots:

- `local-record-p`

  This flag is used for dealing with streams outputting strings char-by-char.

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

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

Sets RECORD to be the parent of CHILD.

`clim:delete-output-record` *child record* **&optional** *errorp* [Generic Function]

If CHILD is a child of RECORD, sets the parent of CHILD to NIL.

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

Sets the parent of all children of RECORD to NIL.

`clim:invoke-with-new-output-record` *stream continuation* [Generic Function]
record-type constructor **&key** *parent* **&allow-other-keys**

Same as in CLIM 2.2 (missing CONSTRUCTOR added).

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

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.

2.6 Command Processing

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

`clim: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)`.

2.7 Incremental redisplay

CLIM applications are most often structured around the *command loop*. The various steps that such an application follow are:

- A *command* is acquired because the user, either typed the name of the command to an interactive prompt, selected a menu item representing a command, or clicked on an active object that translates to a command.
- The *arguments* to that command are acquired. As with the acquisition of the command itself, various gestures can be used to supply the arguments.
- The command is *executed* with the acquired arguments. Typically, the command modifies some part of the *model*

  [2] contained in one or more slots in the application frame.

[2] Some authors use the term *business logic* instead of *model*. Both words refer to the representation of the intrinsic purpose of the application, as opposed to superficial characteristics such as how objects are physically presented to the user.

- The *redisplay functions* associated with the visible panes of the application are executed. Typically, the redisplay function erases all the output and traverses the entire model in order to produce a new version of that output. Since output exists in the form of *output records*, this process involves deleting the existing output records and computing an entirely new set of them.

This way of structuring an application is very simple. The resulting code is very easy to understand, and the relationship between the code of a redisplay function and the output it produces is usually obvious. The concept of output records storing the output in the application pane is completely hidden, and instead output is produced using textual or graphic drawing functions, or more often produced indirectly through the use of `present` or `with-output-as-presentation`.

However, if the model contains a large number of objects, then this simple way of structuring an application may penalize performance. In most libraries for creating graphic user interfaces, the application programmer must then rewrite the code for manipulating the model, and especially for incrementally altering the output according to the modification of the model resulting from the execution of a command.

In CLIM, a different mechanism is provided called *incremental redisplay*. This mechanism allows the user to preserve the simple logic of the display function with only minor modifications while still being able to benefit in terms of performance.

3 Developer manual

3.1 Writing backends

3.1.1 Different types of backends

Backend provides platform specific API for low level drawing operations, getting events, managing window geometry properties and providing native look-and-feel to the application.

There are three types of backends:

Draw-only backend
: This type doesn't implement any kind of events and allows only drawing on it. A good example of it is the See Section 3.2 [PostScript Backend], page 39, which is part of *CLIM II* specification.

Basic backend
: OpenGL, X, or HTML 5 canvas are resources which provide only drawing and event handling primitives. In this case we need to wrap their APIs for McCLIM to use. McCLIM will then use these drawing and windowing primitives to implement portable widgets.

Native backend
: Native backend is based on already complete GUI library which provides a rich set of widgets (for example Cocoa or Win32 API). Additionally to the things needed to be implement in the first two cases, we can also map these native look and feel widgets in McCLIM.

The `clim-null` backend can be used as a template to start with a new backend. If the underlying library you write backend for manages window hierarchy, positioning and events, it is possible to base new pane types on `clim-standard:standard-full-mirrored-sheet-mixin`. Otherwise `clim-standard:standard-single-mirrored-sheet-mixin` provides calculation to support that hierarchy in Lisp side.

Backend protocol

```
NEW CLASS FOR BACKEND 'FOO'
---------------------------
foo-frame-manager
foo-native-frame-manager (optional)
foo-graft
foo-port
foo-medium
foo-pointer
```

Event handling

```
EVENT HANDLING (in port.lisp)
-----------------------------
;;; Use clim-standard:standard-event-port-mixin with
;;; clim-standard:standard-full-mirrored-sheet-mixin
```

```
;;;
;;; Use clim-standard:standard-handled-event-port-mixin with
;;; clim-standard:standard-single-mirrored-sheet-mixin
;;;
;;; and make this event-port-mixin as a subclass of foo-port

;;; Originally in CLIM-INTERNALS
get-next-event
port-frame-keyboard-input-focus
port-grab-pointer
port-ungrab-pointer
synthesize-pointer-motion-event
```

Graft protocol

```
GRAFT (in grafts.lisp)
----------------------------------
;;; Use clim-standard:standard-graft as superclass

;;; Originally in CLIM
graft        ; root window/screen
graft-height ; screen height
graft-width  ; screen width
```

Medium drawing

```
MEDIUM DRAWING (in medium.lisp)
-------------------------------
;;; Originally in CLIM
medium-draw-circle*
medium-draw-ellipse*
medium-draw-line*
medium-draw-lines*
medium-draw-point*
medium-draw-points*
medium-draw-polygon*
medium-draw-rectangle*
medium-draw-rectangles*
medium-draw-text*
```

Medium operationMiscellanies protocol

```
MEDIUM OPERATIONS (in medium.lisp)
----------------------------------
;;; Originally in CLIM
make-medium ; make medium for a given sheet
medium-beep
medium-buffering-output-p
medium-clear-area
```

```
medium-copy-area
medium-finish-output
medium-force-output
medium-line-style
medium-text-style
```

Port protocol

```
PORT (BRIDGE) TO GUI (A SERVER LIKE)
------------------------------------
;;; Originally in CLIM
destroy-port

;;; Originally in CLIM-INTERNALS
port-allocate-pixmap
port-deallocate-pixmap
port-disable-sheet
port-enable-sheet
port-force-output
port-mirror-height
port-mirror-width
port-set-mirror-region
port-set-mirror-transformation
set-sheet-pointer-cursor
```

Frame manager, panes and gadgets

```
FRAME MANAGER, PANES AND GADGETS
--------------------------------
;;; Originally in CLIM
;; in frame-manager.lisp
make-pane-1
note-space-requirements-changed
adopt-frame

;; in port.lisp or pane.lisp/gadget.lisp
allocate-space
destroy-mirror
handle-repaint
realize-mirror
```

Pointer protocol (events?)

```
POINTER (port.lisp or pointer.lisp)
-----------------------------------
;;; Originally in CLIM
pointer-button-state
pointer-modifier-state
pointer-position
```

Text size

```
TEXT SIZE (medium.lisp)
-----------------------
;;; Originally in CLIM-INTERNALS
text-style-character-width
;;; Originally in CLIM
text-size
text-style-ascent
text-style-descent
text-style-height
text-style-mapping
text-style-width
```

Text selection

```
TEXT SELECTION (port.lisp)
--------------------------
;;; Originally in CLIM
selection-owner
selection-timestamp
selection-event
selection-clear-event
selection-notify-event
selection-request-event
selection-event-requestor
request-selection
release-selection
bind-selection
send-selection
get-selection-from-event
```

Miscellaneous

```
MISC
----
;;; Originally in CLIM-INTERNALS
invoke-with-special-choices ; not sure, seems just a funcall

;;; Originally in CLIM-EXTENSIONS
medium-miter-limit          ; determine a draw for miter < sina/2
```

Obsolete

```
NO LONGER NEEDED IN BACKEND
---------------------------
medium-draw-glyph               ; X-specific concept
port-motion-hints               ; X-specific concept
queue-callback                  ; moved to clim-core
medium-clipping-                ; moved to clim-basic
```

```
port-set-sheet-region           ; never used
port-set-sheet-transformation   ; never used
mirror-transformation           ; never used
```

3.2 PostScript Backend

3.2.1 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`.

3.2.2 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.

3.3 Raster Image backend

Raster image backend includes a medium that implements:

- 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` [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` [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)
```

4 Extensions

4.1 Text editor substrate

For brievity only basic topics common to all substrates are covered in the manual. Drei documentation is provided as a separate document.

4.2 Drawing Two-Dimensional Images

4.2.1 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.

4.2.2 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]
...

4.2.3 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).

4.3 Fonts and Extended Text Styles

4.3.1 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)
```

4.3.2 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* [Generic Function]
&optional *size*
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.

4.4 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.

4.5 Render Images

This extension has the goal to provide a fast and flexible way to display images in the screen. It is not a general purpose image processing library (see opticl).

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 the 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.

An image can have an additional alpha channel.

`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-alpha-p` *image* [Generic Function]
This function returns true if the image has an aplha channel.

4.5.1 Image Mixins

`rgb-image-mixin` [Protocol Class]

`stencil-image-mixin` [Protocol Class]

4.5.2 Basic Image

`basic-image` [Protocol Class]
This class is a subclass of the image class.

`image-pixels` *basic-image* [Generic Function]
This function returns the internal representation of the array of pixels.

4.5.3 Drawable Images

In order to be drawn in a screen, each pixel must be mapped into its rgb components. Each component is represented by an octet.

In McCLIM a rgb color is represented as a triple of real number between 0 and 1. Differently, in the image library, when we want drawn an image, a rgb color must be represented as a triple of octet (integer number between 0 and 255). In addition, the opacity is represented by an octet (0 - full transparent, 255 - opaque).

`octect` [Type]
`'(unsigned-byte 8)`

`color-octet->value` *v* [Function]
This function returns `(/ v 255)`.

`color-value->octet` *color* [Function]
This function returns `(coerce (round (* 255 color)) 'octet))`.

`octet-mult` (*o1 o2*) [Function]
This function returns a rusult equals to `(round (* (/ o1 255) (/ o2 255) 255))`.

`octet-blend-function` (*r1 g1 b1 o1 r2 g2 b2 o2*) [Function]
This function is analogous to the `blend-function` for octet.

`drawable-image` [Protocol Class]
This class is a subclass of the image class. All subclasses must implement `map-rgb-color`.

By default, in order to drawn an image into a screen, the backend calls `map-rgb-color`.

`map-rgb-color` *drawable-image fn* [Generic Function]
This function calls fn for each pixels of the image. Function fn must take 5 arguments: x, y, red, green, and blue.

To draw an image you can use `draw-image`.

`draw-image` *sheet image* **&rest** *args* **&key** *clipping-region transformation* [Function]
This function draws an image on a sheet.

4.5.4 McCLIM integration

An image can be used as design or pattern. To draw an image you can also use `draw-pattern*` or `draw-design`.

`image-design` [Class]
This class is a subclass of the design class.

`make-image-design` *image* [Function]
This function returns an image-design of the image `image`.

`image-pattern` (*pattern image-design*) [Class]
This class is a subclass of the pattern and image-design classes.

`make-image-pattern` *image* [Function]
This function returns an image-pattern of the image `image`.

Every design can be converted into pixeled-design.

`pixeled-design` [Protocol Class]

`pixeled-design-region` *pixeled-design* [Generic Function]

`make-pixeled-rgba-octets-fn` *pixeled-design* [Generic Function]

`make-pixeled-rgba-octets-unsafe-fn` *pixeled-design* [Generic Function]

`pixeled-uniform-design` [Protocol Class]

`pixeled-functional-design` [Protocol Class]

`pixeled-image-design` [Protocol Class]

`make-pixeled-design` *design* **&key** *foreground background* [Feneric Function]

4.5.5 Opticl Images

`opticl-image` [Protocol Class]
This class is a subclass of the basic-image and drawable-image classes. The generic function image-pixels returns an opticl image.

`opticl-rgb-image` [Class]
This class is a subclass of the basic-image and opticl-image classes. A pixel is a triple of octets that represents the red, green and blue component, respectively. A pixel can have an optional octet that represents its alpha value.

In the current implementation, `image-pixels` returns an array of type `opticl:8-bit-rgba-image`.

`ocpicl-gray-level-image` [Class]
This class is a subclass of the basic-image and opticl-image classes. In the current implementation, `image-pixels` returns an array of type `8-bit-gray-image`.

`opticl-stencil-image` [Class]
This class is a subclass of the basic-image and opticl-image classes. This image contains only al alpha channel. In the current implementation, `image-pixels` returns an array of type `8-bit-gray-image`.

4.5.6 2d Images

`2d-image` [Protocol Class]
This class is a subclass of the basic-image and drawable-image classes. The generic function image-pixels returns a two dimensioanl array of pixels.

`2d-rgb-image` [Class]
This class is a subclass of the basic-image and 2d-image classes. A pixel is a triple of octets that represents the red, green and blue component, respectively. A pixel can have an optional octet that represents its alpha value.

`image-pixels` returns a (simple-array (unsigned-byte 32) (* *)). The pixel values directly represent ABRG octet values.

`2d-gray-level-image` [Class]
This class is a subclass of the basic-image and 2d-image classes. `image-pixels` returns a (simple-array (unsigned-byte 8) (* *)).

`2d-stencil-image` [Class]
This class is a subclass of the basic-image and 2d-image classes. This image contains only al alpha channel.

`image-pixels` returns a (simple-array (unsigned-byte 8) (* *)).

4.5.7 Operations

`fill-image` *image pixeled-design stencil* **&key** *x y width height stencil-dx stencil-dy* [Generic Function]

`copy-image` *src-image sx sy width height dst-image x y* [Generic Function]

`coerce-image` *image image-class* [Generic Function]

4.5.8 I/O Images

`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).

4.5.9 CL-Vectors Integration

`aa-cells-sweep/rectangle` *image pixeled-design state clip-region* [Generic Function]

`aa-update-state` *state paths transformation* [Function]

`aa-fill-paths` *image pixeled-design paths state transformation clip-region* [Function]

`aa-stroke-paths` *image pixeled-design paths line-style state transformation clip-region* [Function]

`make-path` *x y* [Function]

`line-to` *path x y* [Function]

`close-path` *path* [Function]

`stroke-path` *path line-style* [Function]

5 Applications

5.1 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.

5.1.1 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.

5.1.2 Keyboard shortcuts

Warning: these key accelerators may change in the future.

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

5.1.3 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` *options* **&body** *body* [macro]

Wraps the code in `body` to invoke clim debugger when condition is signalled (binds `*debugger-hook*` to `#'debugger`). Bindings are inherited by new threads. `options` are not used at the moment.

`install-debugger` [function]

Installs `clim-debugger` globally (no need to wrap body in `with-debugger`.

5.2 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**.

5.2.1 Usage

5.2.1.1 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.

5.2.1.2 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.

5.2.1.3 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:

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.

Structures
: Structures are inspected the same way as standard objects.

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

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

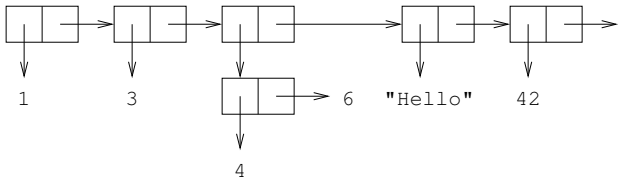
Symbols

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

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:



5.2.2 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:

SAMPLE n=9

mean	12.125776
std. dev.	5.621417

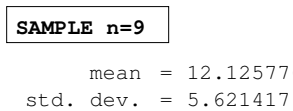
But what we really want is something more closely adapted to our needs. It would be nice if we could just have a table of things like $\overline{x} = 12.125776$ and have them come out formatted nicely. Before we attempt mathematical symbols, let's focus on getting the basic layout right. For this, we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

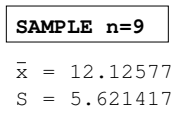
Finally, to illustrate the proper use of **inspect-object-briefly**, suppose that we want the "n=9" (or whatever the sample size n equals) part to have an itlicised n. We can fix this easily:

```
(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`.

5.2.3 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.

5.3 Listener

Auxiliary material

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

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

Variable index

Function and macro index

(

A

C

D

F

G

I

L

M

O

P

R

S

T

W