CLIM 2 User Guide

version 2.2

June, 2000

Copyright and other notices:

Contents

[This page intentionally left blank.]

Chapter 1 Introduction and notation

CLIM stands for Common Lisp Interface Manager. It is a portable, powerful, high-level user interface management system intended for Common Lisp software developers.

This manual is written for CLIM 2.0 by developers of CLIM 2.0. It contains parts (when relevant) of the CLIM 1.0 manual (published by Symbolics) and it contains new material describing features specific to CLIM 2.0.

You comments on this manual will be most appreciated. Please tell us both what parts you like (so we can replicate the style in other areas) and what you feel is missing (so we can add it). Information on where to send comments can be found on the data sheet at the end of this manual.

1.1 Notation used in this manual

The notation used in this document differs somewhat from the notation used in earlier Allegro CL documents, although the changes are mostly evolutionary.

Formal definitions are displayed. The first line names the object being defined and tells what kind of object it is. (The name is on the left, the type in brackets on the right). here are some examples:

name1 **[Function]**

name2 **[Generic function]**

name3 **[Variable]**

If the object accepts arguments (if it is a function, macro, generic function, or even a keyword argument), the arguments are specified on their own displayed line titled '**Arguments:**'. If an operator take no arguments, the **Arguments:** line is present but nothing follows **Arguments:**. Following the arguments line is the description of the object, again in indented paragraphs, some started with a filled in black square. Here, for example, is the complete description of the generic function **transform-rectangle***:

transform-rectangle* **[Generic function]**

Arguments: *transform x1 y1 x2 y2*

- Applies the transformation *transform* to the rectangle specified by the four coordinate arguments, which are real numbers. The arguments *x1*, *y1*, *x2*, and *y1* are canonicalized in the same way as for **make-bounding-rectangle**. Returns four values that specify the minimum and maximum points of the transformed rectangle in the order *min-x*, *min-y*, *max-x*, and *max-y*.
- It is an error is *transform* does not satisfy **rectilinear-transformation-p**.
- **transform-rectangle*** is the spread version of **transform-region** in the case where the transformation is rectilinear and the region is a rectangle.

Generally, a black square is used to indicate a new topic in the description. Note the following about these definitions:

Packages

Unless otherwise specified, all symbols naming objects are in the `clim` package. Most of the example code have a `clim:` package qualifier but some do not. If you enter example code, be sure that the `clim` package is used or that the `clim:` qualifier is present. The `clim-user` package uses the `clim` package and also the `common-lisp` package, so all `common-lisp` and `clim` symbols can be referenced without qualifiers when the current package is `clim-user`. `clim-user` does not use `excl` (the package containing many of the Allegro CL extensions). The `excl` package has no symbol conflicts with `clim-user` so you can use `excl` when in `clim-user` if you wish.

Keyword arguments

In other Allegro CL manuals, keyword arguments are named with a keyword in formal argument lists and are referred to as 'the value of the `:xxx` keyword argument' in the text. In this manual, we use a different method. Keyword arguments do not have a preceding : in the argument list (but they do follow `&key`); and the argument is referred to in the text in *`argument`* notation (again, without a :). Thus for example:

`accept-from-string` **[Function]**

Arguments: *`type string`* `&key view default default-type activation-gestures additional-activation-gestures delimiter-gestures additional-delimiter-gestures (start 0) end`

- Reads the printed representation of an object of type *`type`* from *`string`*. This function is like **`accept`**, except that the input is taken from *`string`*, starting at the position *`start`* and ending at *`end`*. *`view`*, *`default`*, and *`default-type`* are as in **`accept`**.
- [...]

In this example, there are many keyword arguments. One is *`start`*, which defaults to 0. If you want to specify a value for *`start`* in a call to **`accept-from-string`**, you would do it as follows:

```
(accept-from-string symbol "ab var1" string :start 3)
```

Type faces

Type faces are used to distinguish between symbols naming functions and other operators, symbols naming other things (like constants, variables, etc.), printed forms, and examples.

- Function and other operator names are printed in **`bold Courier`**.
- Arguments (and other placeholders) are in *`slant Courier`*.
- Symbols naming other things are printed in `plain Courier`. That face is thus used for `constants` (such as `#\A` and `nil`) and special symbols (such as `*package*`) and keywords and lambda-list keywords (such as `:test` and `&optional`, respectively).
- Printed forms and examples are printed in `Courier`, typically with user input in `plain` and what the system prints in **`bold`**.
- Longer examples sometimes are printed in a reduced size, so that all 80 characters in a line are printed on the same line.

We have tried to be careful to break symbol names at a hyphen in the symbol name. Thus, **`draw-rectangle`** will be broken at the hyphen or not at all.

→ symbol means 'evaluates to'

Occasionally, we wish to show a form and the result of evaluating the form. We use the symbol → to mean 'evaluates to'. Thus:

```
(+ 1 2) → 3
(car (list 'a 'b)) → a
```

Special notation for unimplemented features

Occasionally, you will see displayed paragraphs that look like:

> **IMPLEMENTATION LIMITATION:** In this release, this does not....

These describe known limitations of the release of CLIM 2.0. Note that not all known limitations are described in this way. Please also look at the Release Notes for CLIM, where other known bugs and problems are listed.

1.2 Comments and suggestions

We are pleased to hear from our users in order to improve our products. We invite your comments and suggestions. The address to which to write, either by post or by electronic mail, is on the information sheet at the end of this document.

1.3 Some CLIM terms

This section defines a number of terms used in CLIM. Note that some terms may have common meanings which may differ somewhat from the specific CLIM usage.

Bounding rectangle

Every region in CL:IM has a derived bounding rectangle, which is the smallest rectangle that contains every point in the region, and may contain additional points as well. Bounding rectangles are different from the ordinary geometric rectangles.

Frame

Also called an application frame. A frame appears on the screen as a window. It is subdivided into panes and these display the information and accept the input associated with your application. See chapter 9 **Defining application frames in CLIM** for more information.

Gesture

Actions such as pressing mouse buttons or keyboard keys, perhaps in combination, are called gestures. Because keyboards and mice differ, there is a level of abstraction between a gesture name and the physical action associated with a gesture. this allows applications to be designed to respond to gestures (by displaying a menu, selecting an item, etc.) which can be associated with physical actions when the application is run, rather than when it is written. See section 8.7.6 **Gestures in CLIM** for more information on defining gestures.

Output record

Unless told to do otherwise, all output to a window in CLIM is captured and saved by the output recording mechanism and stored in an output record. This record can be 'replayed', reproducing exactly the same output as that which generated the record. Output recording is used for scrolling and for presentations. See chapter 8 **Output recording in CLIM** for more information.

Presentation

CLIM associates Lisp objects with their visual representations on the screen. Thus when you point (with the mouse) to a visual representation on the screen, CLIM can associate what your are pointing to with a specific Lisp object. The visual representation of the object is called the presentation. See chapter 8 **Presentation types in CLIM** for more information.

Viewport

A window stream viewport is the region of the drawing plane that is visible through the window. You can change the viewport by scrolling or by reshaping the window. The viewport does not change if the window is covered by another window (that is, the viewport is the region of the drawing plane that *would* be visible if the window were stacked on top). When the cursor position moves out of the viewport, what happens is determined by the end-of-line action or the end-of-page action of the window stream.

1.4 Reporting bugs

We are committed to the highest standards of software engineering. Releases of Allegro CL and CLIM are extensively tested both internally and in the field before wide dissemination. Nevertheless, as with all computer programs, it is possible that you will find bugs or encounter behavior that you do not expect. In that event, we will do our utmost to resolve the problem. But, resolving bugs is a cooperative venture, and we need your help.

Before reporting a bug, please study this document to be sure that what you experienced is indeed a bug. If the documentation is not clear, this is a bug in the documentation: CLIM may not have done what you expected, but it may have done what it was supposed to do.

A report that such and such happened is generally of limited value in determining the cause of a problem. It is very important for us to know what happened before the error occurred: what you typed in, what Allegro CL typed out. A verbatim log, preferably hard copy, may be needed. If you are able to localize the bug and reliably duplicate it with a minimal amount of code, it will greatly expedite repairs.

It is much easier to find a bug that is generated when a single isolated function is applied than a bug that is generated somewhere when an enormous application is loaded. Although we are intimately familiar with Allegro CL, you are familiar with your application and the context in which the bug was observed. Context is also important in determining whether the bug is really in Allegro CL or in something that it depends on, such as the operating system.

To this end, we request that your reports to us of bugs or of suspected bugs include the following information. If any of the information is missing, it is likely to delay or complicate our response.

- **CLIM details**. Please tell us the version of CLIM, the machine on which you are running.
- **Lisp implementation details**. Tell us the implementation of Allegro CL that you are using, including at least the release number and date of release of Allegro CL, the manufacturer, model and version of the hardware on which you are running Allegro CL, and the operating system and its release number. The function **`excl:dribble-bug`** will automatically write all the needed information to a file.
- **Information about you**. Tell us who you are, where you are and how you can be reached (an electronic mail address if you are reachable via Internet or uucp, a postal address, and your telephone number), your Allegro CL license number, and in whose name the license is held.
- **A description of the bug**. Describe clearly and concisely the behavior that you observe.

- **Exhibits**. Provide us with the *smallest, self-contained* Lisp source fragment that will duplicate the problem, and a log (e.g. produced with **dribble** or **dribble-bug**) of a *complete* session with Allegro CL that illustrates the bug.

A convenient way of generating at least part of a bug report is to use the **excl:dribble-bug** function mentioned above. Typing

```
(excl:dribble-bug filename)
```

causes implementation and version information to be written to the file specified by *filename*, and then records the Lisp session in the same file. Typing

```
(dribble)
```

will close the file after the bug has been exhibited. **excl:dribble-bug** is defined in section 3.7. Note that if what you type to duplicate the bug loads in files of yours either directly or indirectly, attach a complete listing of the source version of these files to your session log. The following dialogue provides a rudimentary template for the kernel of a bug report.

```
USER(5) (dribble-bug "bug.dribble")
USER(6) ;; Now duplicate your bug . . .
USER(7) (dribble)
```

Send bug reports to either the electronic mail or postal address given on the information sheet that is enclosed with this document. We will investigate the report and inform you of its resolution in a timely manner.

We will meet you more than half way to get your project moving again when a bug stalls you. We only ask that you take a few steps in our direction.

Where to report bugs and send questions

The information sheet at the very end of this manual gives e-mail and street addresses for Franz Inc., as well as our telephone number (in short, **bugs@franz.com** for all bug reports and questions -- *any* question, despite the names 'bugs'; Franz Inc., 1995 University Ave., Berkeley CA 94704 USA; +510-548-3600).

1.5 Patches

Patches are available for download using the World Wide Web. Please see the discussion in the online document *doc/cl/introduction.htm* for information on patches.

The Allegro CL FAQ

We have prepared, and we regularly update, A FAQ (Frequently Asked Questions) document for Allegro CL. This document contains questions and answers about Allegro CL, examples of interest, and information that did not make it into this manual. Accessing the FAQ is described in *doc/cl/introduction.htm*.

[This page intentionally left blank.]

Chapter 2 Getting started with CLIM

In this chapter, we describe how to set up your system for using CLIM, how to access CLIM within Allegro CL, how to access the CLIM demo files, and how to write a simple demo of your own.

The simple demo involves displaying an application frame and identifying a pane in that frame. After we create the application frame, the pane will be made the value of the special variable `*test-pane*`. We use `*test-pane*` in many of the small examples in this manual.

2.1 General information

Features for CLIM

A Lisp image with CLIM 2.0 loaded will have both `:clim-2` and `:clim-2.0` on the `*features*` list. It will also have the feature `:clim-motif`.

Motif on Linux and FreeBSD

You must obtain version 2.1 of Motif to use CLIM on Linux and FreeBSD platforms. A free version is available from **www.openmotif.org**. With earlier release, it was necxessary to purchase a commercial version of Motif on these platforms. That is not longer necessary with release 6.1 of Allegro CL

Loading CLIM into a Lisp image built without CLIM

We urge users to build an image containing CLIM using buildclim.cl (simply load it into a running Lisp). However, you may load CLIM into a running Lisp by evaluating

```
(require :climxm)
```

on UNIX platforms and

```
(require :climnt)
```

on Windows.

The clim-user package

We recommend that users of CLIM work in the `clim-user` package instead of the `common-lisp-user` package. `clim-user` uses all the appropriate packages necessary for CLIM; as a result, you do not have to qualify symbols associated with CLIM. It also uses the `clim-lisp` package, which exports (among some other things) all the symbols in the `common-lisp` package. You make the clim-user package current by evaluating the form:

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

You may wish at this point to use the `excl` package. This package contains standard extensions to Common Lisp in Allegro CL. Use the excl package by evaluating:

```
(use-package :excl)
```

Setting the server path

The variable `*default-server-path*` tells CLIM specifies the display where CLIM windows will be shown. This variable should be set appropriately before doing anything else in CLIM.

The value of this variable should be a list of one or three or more elements. The first element must be `:motif`. The second and third elements (if present) are the keyword `:display` and a string naming the display.

The additional elements can specify the application name and application class of the CLIM application (viewed as an X client). See section 2.3 **X resources** for more information on setting the application name and class.

You name a display as follows:

```
"<machine-name>:display#:screen#"
```

The display# is typically 0. The screen# is typically left out (unless your display has more than one associated screen). So, if your machine is named ‘tavy’, the string would typically be "tavy:0". If you wanted to display Motif CLIM on tavy, you would evaluate the following form:

```
(setf clim:*default-server-path* '(:motif :display "tavy:0"))
```

If you do not specify a `:display`, CLIM gets the display from the DISPLAY environment variable. To see what Lisp sees as the value of this variable, evaluate:

```
(sys:getenv "DISPLAY")
```

The initial value of `*default-server-path*` is `(:motif)`.

The CLIM demos

The *src/clim/demo/* directory on the distribution tape contains a set of demo programs. Compiled versions of the demos also exist and can be loaded and run as described below.

To load the demos into an image with CLIM already loaded (see **Loading CLIM into a Lisp image built without CLIM** above), evaluate:

```
(require :climdemo)
```

Once that form has completed, you may start the demos. The simplest way is to evaluate:

```
(clim-demo:start-demo)
```

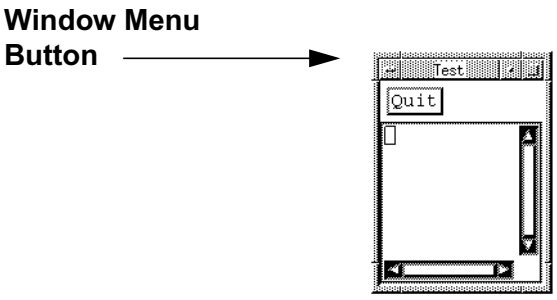
A simple example

Here is another simple example. We use this example throughout this manual. The following code displays a simple application frame with a display pane. This display pane can be used to test drawing and writing functions along with other things.

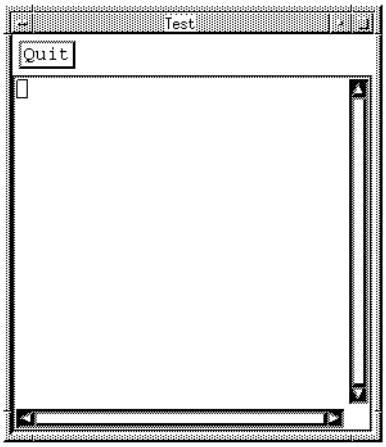
```
(in-package :clim-user)

(define-application-frame test ()
  ()
  (:panes
   (display :application))
  (:layouts
   (default display)))

(define-test-command (com-quit :menu t) ()
  (frame-exit *application-frame*))

(defvar *test-frame* nil)

(defun test ()
  (let ((frame (or *test-frame*
   (setf *test-frame*
        (make-application-frame 'test)))))
    (mp:process-run-function "test" #'run-frame-top-level frame)))

;;; evaluate the following to create and display the frame:
(test)

;;; after calling (test) and the frame has appeared evaluate...
(setq *test-pane* (get-frame-pane *test-frame* 'display))
```

Be sure to evaluate the last form (setting the value of `*test-pane*`) after you have evaluated `(test)` and the frame has appeared. Until that time, get-frame-pane will not work properly.

The frame, when it appears, it looks like this. .



Notice the Window Menu Button in the upper right corner. Pressing the left mouse button over it brings up a menu of choices. Choosing Size allows you to change the size of the window. We have done so, making the window bigger:



Clicking on the Quit button will cause the frame to be closed. You can clear the display pane portion by evaluating

```
(window-clear *test-pane*)
```

test-frame and *test-pane*

The frame we have just created is the value of `*test-frame*`. The display pane inside the frame is the value of `*test-pane*`. many of the examples in this manual use `*test-frame*` or `*test-pane*`. If you wish to try those examples, you should create this frame and pane as indicated above.

Many CLIM operations need a context to work

Many CLIM operations need to be executed from within the context of a frame's top level function. In particular this ensures that `*application-frame*` is correctly bound. If you get unexpected results by calling CLIM functions outside of this context (eg simply evaluating the forms within a listener) the first thing you should try is to call the functions from within the frame's top level. You can do this by defining frame commands with `:menu t` and then activating the command from the running frame. e.g.

```
(define-test-command (com-clear :menu t) ()
  (let ((stream (get-frame-pane *application-frame* 'display)))
    (window-clear stream)))
```

Another alternative is to use the CLIM Lisp Listener demo. This provides a read-eval-print loop running within the frame's top level. `*standard-output*` can be used whenever a CLIM stream pane is required. eg.

```
(window-clear *standard-output*)
```

2.2 Window-manager-specific information

Some window managers have peculiarities which makes CLIM behave differently than you might expect. We discuss those peculiarities know at the time of the printing of this document in this section. Note that there are likely other things which were unknown at the time this manual was printed. Check the release notes or contact customer support for more information.

Motif peculiarities

Using Motif on keyboards without a F10 key

When running a Motif application (including CLIM) you may see the following warning from time to time:

```
Warning: Xt:
    Name:
    Class: XmRowColumn
   Illegal mnemonic character; Could not convert X KEYSYM to a keycode
```

This occurs when you are using a keyboard that does not have an F10 key, which Motif uses by default to pop up a menu.

The keys used by Motif are set in a file named *.motifbind* in your home directory. We supply such a file in the distribution, *misc/dot-motifbind*. You can use this file if you do not use one of your own. Whichever *.motifbind* you use, it should not contain a reference to the F10 key Any such reference should be changed to refer to a key that does exist. Note that as supplied, *dot-motifbind* does refer to F10 so it has to be edited.

2.3 X resources

The visual appearance of CLIM applications can be controlled explicitly in the pane specifications of application frame definitions or implicitly by the use of X resources. In all cases the pane specification overrides any X resource defaults.

A complete resource specification uses the X toolkit style of resource specifications. This has the advantage that resource specifications controlling CLIM's behavior can be combined with specifications controlling the behavior of the underlying Motif toolkit in a uniform and consistent manner. The basic syntax (without wildcards) of a resource specification is:

```
application.widget.widget...resource:          value
```

Each of the application, widget and resource components are described in turn below:

Application

The application name and class for all CLIM windows is by default

Name	Class
`clim`	`Clim`

The defaults can be overridden by use of the options *`application-name`* and *`application-class`* in the port server-path. For example:

```
(find-port :server-path '(:motif :display "louie:0"
                                 :application-name "myApp"
                                 :application-class "MyApp"))
```

Widget

The widget names and classes of the underlying toolkit can be used to identify to which particular widget the resource applies.

CLIM names those widgets that are mirrors of CLIM panes with the name of the pane. These names can then be used to form portable (i.e. not toolkit dependent) resource specifications. The pane names are modified to conform with the standard X widget naming conventions. Thus, for example:

CLIM pane name	X widget name
`display`	`display`
`main-display`	`mainDisplay`

Resource

The resource names and classes that CLIM understands are:

Name	Class	Example
`foreground`	`Foreground`	`black`
`background`	`Background`	`gray70`
`textStyle`	`TestStyle`	`(:fix :roman :small)`

`foreground` and `background` have the standard meanings.

`textStyle` controls the font. It can be specified either as the name of an X font using the standard naming conventions or alternatively a CLIM text style specification can be given. If you use standard X font name to specify a text style then the individual components are not defined and cannot be individually modified. For example the following will not work as expected.

```
(with-text-size (stream :larger) (format stream ...)
```

Additionally resource name and classes specific to the underlying toolkit can be used.

Wildcards

The standard resource wildcards `*` and `?` can be used in resource specifications

Examples

```
Clim*Background:                    #B900B900B900
Clim*Foreground:                    black
Clim*TextStyle:                     (:fix :roman :small)
clim*menuBar*background:             khaki
clim*menuBar.textStyle:              -*-swiss 742-bold-r-normal-*-140-*-p-100-*
clim*XmPushButton.shadowThickness:   8
```

Reinitializing resources

This is hard to do while CLIM is running. It may work to destroy the current port (with **`destroy-port`**), make the changes to X, and then re-establish the port with **`find-port`**. However, the recommended thing to do is stop CLIM and then restart it.

2.4 Some miscellaneous quirks and tricks

This section points out things that do not seem to fit elsewhere and also contains bit of code for accomplishing various things that we believe a number of users will want to do.

Many CLIM macros turn bodies into closures

The following code defines a method to draw some buttons.

```
(defmethod draw-csf-buttons ((frame permanent-window) stream)
  (let ((csf-list '(S C H P Z I))
        (text-list '(" S " " C " " H " " P " " Z " "  I  "))
        (window-width (window-inside-width stream)))
    (draw-text* stream "Critical Safety Functions"
                (/ window-width 2) 15 :align-x :center :align-y :top
                :text-size :large)
    ;; draw the buttons
    (stream-set-cursor-position* stream 36 60)
    (dotimes (i (length text-list))
      (accept-values-command-button
       (stream)
       (princ (nth i text-list) stream)
       (display-csf-graph (nth i csf-list)))
      (stream-increment-cursor-position* stream 47 nil)
      )))
```

Now, the buttons work well but the function **`display-csf-graph`** is always called with `nil` as argument instead of the right one. This happens because **`accept-values-command-button`** creates a closure that references the variable `i`. This is **`setq`**'ed by the **`dotimes`** and so at the point the form `(display-csf-graph (nth i csf-list))` is evaluated the value of the variable is the length of `text-list`, i.e. the value at the point the loop terminates.

To get the desired behavior, the code should bind a variable to `i` and reference that variable within the body of the **`accept-values-command-button`**, as is done in the following code:

```
(dotimes (i (length text-list))
   (let ((index i))
    (accept-values-command-button
     (stream)
     (princ (nth index text-list) stream)
     (display-csf-graph (nth index csf-list)))
    (stream-increment-cursor-position* stream 47 nil)))
```

Reading a password

When an application reads a password from a user, it typically does not want the password echoed. That is easy enough to achieve but somewhat harder is echoing some characters (‘X’s or ‘?’s, e.g.) so the user can keep track of how many characters have been typed. The following code can be used for this purpose:

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

(define-presentation-type password () :inherit-from '((string)
                        :description "password"))

;;; Presenting a password prints a string of ?'s.

(define-presentation-method present (password (type password) stream
                                              (view textual-view)
                                               &key acceptably)
  (when acceptably (error "Not acceptably"))
  (write-string (make-string (length password) :initial-element #\?) stream))


;;; Accepting a password turns off output drawing and recording,
;;; then reads a string using READ-TOKEN.

(define-presentation-method accept
                          ((type password) stream (view textual-view) &key)
  (let* ((start (stream-scan-pointer stream))
         (passwd (with-output-recording-options (stream :draw nil :record nil)
                   (read-token stream))))
    (cond ((< (length passwd) 6)
           (simple-parse-error "Need a password with at least 6 characters!"))
          (t
           (presentation-replace-input
            stream passwd 'password view
            :buffer-start start)
           (return-from accept passwd)))))
```

Getting a gc cursor

CLIM runs on Lisp, which does its own memory management. From time to time, Lisp will seem to stop or freeze while it is in fact performing a garbage collection, that is cleaning up the memory it uses.

Allegro CL has two types of garbage collections: scavenges and global gc's. These are described in chapter 15 of the *Allegro CL User Guide*, but in brief, scavenges are usually quite fast, since only newspace is gc'ed, while global gc's, which clean up the entire heap, usually take a noticeable amount of time, sometimes several (and perhaps many) minutes in large complicated applications.

Because Lisp may not be able to allocate a Lisp object until the gc (of whichever type) is complete, getting a gc cursor (or some other indication that a gc is occurring) is difficult. CLIM 2, however, does provide such a facility. If you evaluate the following form, the cursor will turn into the waiting cursor (typically a watch or an hourglass) when the cursor is over a CLIM window during a gc.

```
(setq xm-silica::*use-clim-gc-cursor* t)
```

Note that Allegro CL provides a great deal of control over global gc's. You can arrange things so that a global gc never occurs automatically, but instead occurs only when your code thinks a global gc is appropriate. Since global gc's take much longer than scavenges, you may wish to use this additional control to schedule global gc's appropriately or to provide a warning to the user that a global gc is about to happen in addition to using the gc cursor. See chapter 15 of the *Allegro CL User Guide* for more information on controlling global gc's.

Getting hyper and super keys

Most keyboards have a Meta key of an equivalent (that is a modifier key which you can press that will be interpreted as Meta by CLIM without any action on your part). CLIM also supports Hyper and Super keys, but often a keyboard does not have these. X has mod1, mod2, and mod3 keys, and these correspond to Meta, Super, and Hyper but often only mod1 is set properly.

Here is how to use xmodmap to cause certain keys to generate mod2 and mod3 (super and hyper) on Suns (which we take as a standard example). Note that if you make use of this facility, users of your application will have to set up the keys themselves prior to running your application. The setup cannot be done programmatically from within CLIM.

Do the following to rebind the left meta key to Meta, the right meta key to Super and the left alt key to Hyper:

```
xmodmap -e 'clear mod1'
xmodmap -e 'clear mod2'
xmodmap -e 'clear mod3'
xmodmap -e 'add mod1 = Meta_L'
xmodmap -e 'add mod2 = Meta_R'
xmodmap -e 'add mod3 = Alt_L'
```

Rectangles and bounding-rectangles are different

`rectangle` objects (created with **`make-rectangle`**) and `bounding-rectangle` objects (created with **`make-bounding-rectangle`**) are quite different types of objects. Sometimes, users confuse them and use a rectangle where a bounding-rectangle is called for (for example, a clipping region should be a bounding-rectangle, not a rectangle). Please be aware of the difference as a potential cause of errors.

[This page intentionally left blank.]

Chapter 3 Drawing graphics in CLIM

3.1 Concepts of drawing graphics in CLIM

CLIM offers a set of *drawing functions* that enable you to draw points, lines, polygons, rectangles, ellipses, circles, and text. You can affect the way the geometric objects are drawn by supplying *drawing options* to the drawing functions. The drawing options support clipping, transformation, line style, text style, ink, and other aspects of the graphic to be drawn (see the section 4.2 **Using CLIM drawing options**).

In many cases, it is convenient to use **`with-drawing-options`** to surround several calls to drawing functions, each using the same options. You can override one or more drawing options in any given call by supplying keywords to the drawing functions themselves. This can also be more efficient, as passing drawing options to each drawing function can cons new short-lived objects; using **`with-drawing-options`** around several drawing functions will cons these objects only once.

3.1.1 The drawing plane

When drawing graphics in CLIM, you imagine that they appear on a *drawing plane*. The drawing plane extends infinitely in four directions and has infinite resolution (no pixels). A line that you draw on the drawing plane is infinitely thin. The drawing plane provides an idealized version of the graphics you draw. The drawing plane has no material existence and cannot be viewed directly.

Of course, you intend that the graphics should be visible to the user, and must be presented on a real display device. CLIM transfers the graphics from the drawing plane to the window via the *rendering process*. Because the window lives on hardware that has physical constraints, the rendering process is forced to compromise when it draws the graphics on the window. The actual visual appearance of the window is only an approximation of the idealized drawing plane.

Figure 3.1 shows the conceptual model of the drawing functions sending graphical output to the drawing plane, and the graphics being transferred to a screen by rendering.

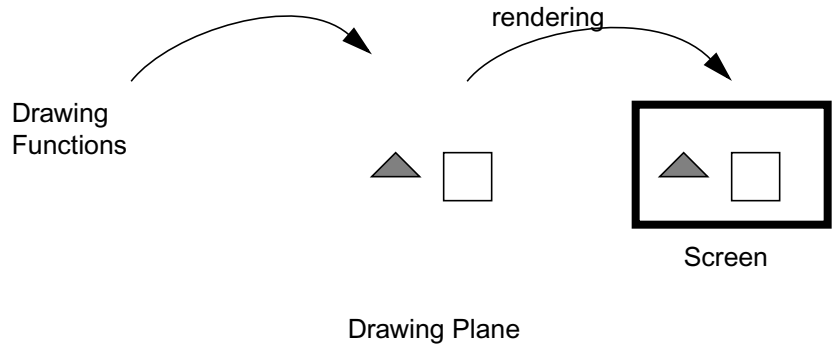


Figure 3.1. Rendering from Drawing Plane to Window

The distinction between the idealized drawing plane and the real window enables you to develop programs without considering the constraints of a real window or other specific output device. This distinction makes CLIM's drawing model highly portable.

3.1.2 Coordinates

When producing graphic output on the drawing plane, you indicate where to place the output with *coordinates*. Coordinates are a pair of numbers that specify the x and y placement of a point. When a window is first created, the origin (that is, x=0, y=0) of the drawing plane is positioned at the top-left corner of the window. Figure 3.2 shows the orientation of the drawing plane, X extends toward the right, and Y extends downward.

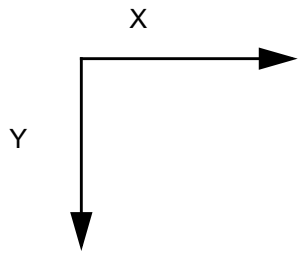


Figure 3.2. X and Y Axes of Drawing Plane

As the window scrolls downward, the origin of the drawing plane moves above the top edge of the window. Because windows maintain an output history, the Y axis can extend to a great length. In many cases, it is burdensome to keep track of the coordinates of the drawing plane, and it can be easier to think in terms of a *local coordinate system*.

For example, you might want to draw some business graphics as shown in Figure 3.3. For these graphics, it is more natural to think in terms of the Y axis growing upwards, and to have an origin other than the origin of the drawing plane, which might be very far from where you want the graphics to appear. You can create a local coordinate system in which to produce your graphics. The way you do this is to define a *transfor-*

mation which informs CLIM how to map from the local coordinate system to the coordinates of the drawing plane. For more information, see **`with-room-for-graphics`**.

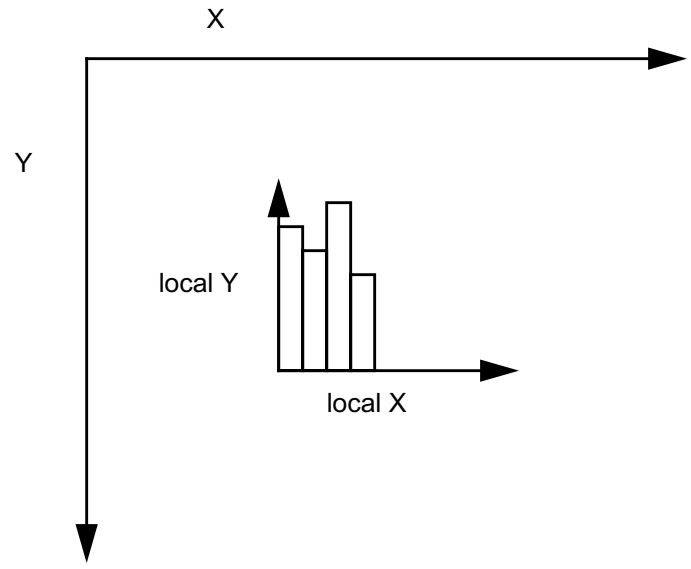


Figure 3.3. Using a Local Coordinate System

3.1.3 Sheets and Streams, and Mediums

A *sheet* is the most basic window-like object supported by CLIM. It has two primary properties: a region, and a transformation that relates its coordinate system to the coordinate system of its parent. A streams is a special kind of sheet that implements the stream protocol; streams include additional state such as the current cursor position (which is some point in the drawing plane).

A *medium* is an object on which drawing takes place. A medium has as attributes: a drawing plane, the medium's foreground and background, a drawing ink, a transformation, a clipping region, a line style, a text style, and a default text style. Sheets and streams that support output have a medium as one of their attributes.

The drawing functions take a *`medium`* argument that specifies the destination for output. The drawing functions are specified to be called on mediums, but they can be called on most panes, sheets, and stream as well.

The medium keeps track of default drawing options, so if drawing functions are called and some options are unspecified, they default to the values maintained by the medium.

Different medium classes are provided to allow users to draw on different sorts of devices, such as displays and printers.

3.2 Examples of Using CLIM Drawing Functions

Figure 3.4 shows the result of evaluating the following forms:

```
(draw-rectangle* *test-pane* 10 10 200 150 :filled nil :line-thickness 2)
(draw-line* *test-pane* 200 10 10 150)
```

```
(draw-point* *test-pane* 180 25)
(draw-circle* *test-pane* 100 75 40 :filled nil)
(draw-ellipse* *test-pane* 160 110 30 0 0 10 :filled nil)
(draw-ellipse* *test-pane* 160 110 10 0 0 30)
(draw-polygon* *test-pane* '(20 20 50 80 40 20) :filled nil)
(draw-polygon* *test-pane* '(30 90 40 110 20 110))
```

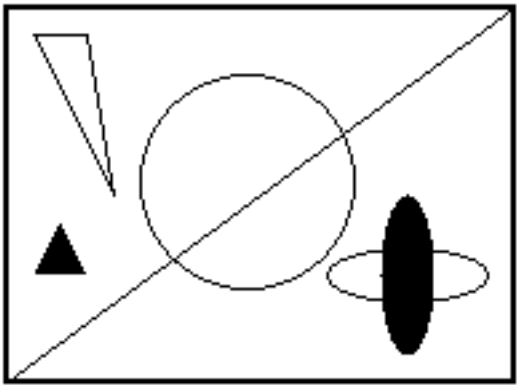


Figure 3.4. Simple Use of the Drawing Functions

3.3 CLIM drawing functions

Most of CLIM's drawing functions come in pairs. One function takes two arguments to specify a point by its x and y coordinates; the corresponding function takes one argument, a point object. The function accepting a point object has a name without an asterisk (*****), and the function accepting coordinates of the point has the same name with an asterisk appended to it. For example, **`draw-point`** accepts a point object, and **`draw-point*`** accepts coordinates of a point. We expect that using the starred functions and specifying points by their coordinates will be more convenient in most cases.

The drawing functions take keyword arguments specifying drawing options. For information on the drawing options, see the section 4.2 **Using CLIM drawing options**.

If you prefer to create and use point objects, see the section 3.6.3 **CLIM Point Objects**.

`draw-point` [Function]

Arguments: *`medium point-1`* `&key line-style line-thickness line-unit ink clipping-region transformation`

- Draws a point on *`medium`* at the position indicated by *`point`*.

  Note that a point is a one-dimensional object. In order to be visible, the rendering of a point must occupy some non-zero area on the display hardware. A line style object represents the advice of CLIM to the rendering substrate on how to perform the rendering.

- The *`line-unit`* and *`line-thickness`* arguments control the size on the display device of the blob used to render the point.

draw-point* [Function]

Arguments: *medium x y* &key line-style line-thickness line-unit ink clipping-region transformation

■ Draws a point on *medium* at the position indicated by *x* and *y*.

Note that a point is a one-dimensional object. In order to be visible, the rendering of a point must occupy some non-zero area on the display hardware. A line style object represents the advice of CLIM to the rendering substrate on how to perform the rendering.

■ The *line-unit* and *line-thickness* arguments control the size on the display device of the blob used to render the point.

Both **draw-point*** and **draw-point** call **medium-draw-point*** to do the actual drawing.

draw-points [Function]

Arguments: *medium point-seq* &key line-style line-thickness line-unit ink clipping-region transformation

■ Draws a set of points on *medium*. *point-seq* is a sequence of point objects specifying where a point is to be drawn. This function is equivalent to calling **draw-point** repeatedly, but it can be more convenient and efficient, when drawing more than one point.

draw-points* [Function]

Arguments: *medium coord-seq* &key line-style line-thickness line-unit ink clipping-region transformation

■ Draws a set of points on *medium*. *coord-seq* is a sequence of pairs of *x* and *y* positions (that is, a sequence of alternating x coordinates and y coordinates which when taken pairwise specify the points to be drawn).

■ This function is equivalent to calling **draw-point*** repeatedly, but it can be more convenient and efficient, when drawing more than one point.

Both **draw-points*** and **draw-points** call **medium-draw-points*** to do the actual drawing.

draw-line [Function]

Arguments: *medium point-1 point-2* &key line-style line-thickness line-unit line-dashes line-cap-shape ink clipping-region transformation

■ Draws a line segment on *medium*. The line starts at the position specified by *point-1* and ends at the position specified by *point-2*, two point objects.

■ This function is the same as **draw-line***, except that the positions are specified by points, not by x and y positions.

draw-line* [Function]

Arguments: *medium x1 y1 x2 y2* &key line-style line-thickness line-unit line-dashes line-cap-shape ink clipping-region transformation

■ Draws a line segment on *medium*. The line starts at the position specified by (*x1*, *y1*), and ends at the position specified by (*x2*, *y2*).

Both **draw-line*** and **draw-line** call **medium-draw-line*** to do the actual drawing.

draw-lines [Function]

Arguments: *medium point-seq* &key line-style line-thickness line-unit line-dashes line-cap-shape ink clipping-region transformation

■ Draws a set of disconnected line segments onto *medium*. *point-seq* is a sequence of pairs of points. Each point specifies the starting and ending point of a line.

This function is semantically equivalent to calling **draw-line** repeatedly, but it can be more convenient and more efficient when drawing more than one line segment. See the function **draw-line**.

draw-lines* [Function]

Arguments: *medium coord-seq* &key line-style line-thickness line-unit line-dashes line-cap-shape ink clipping-region transformation

■ Draws a set of disconnected line segments onto *medium*. *coord-seq* is a sequence of pairs of *x* and *y* positions in a list (or vector). Each pair specifies the starting and ending point of a line.

This function is equivalent to calling **draw-line*** repeatedly, but it can be more convenient and more efficient when drawing more than one line segment. See the function **draw-line***.

Both **draw-lines*** and **draw-lines** call **medium-draw-lines*** to do the actual drawing.

draw-arrow [Function]

Arguments: *stream point-1 point-2* &key from-head (to-head t) (head-length 10) (head-width 5) line-style line-thickness line-unit line-dashes line-cap-shape ink clipping-region transformation

■ Draws an arrow on *stream*. The arrow starts at the position specified by *point-1* and ends with the arrowhead at the position specified by *point-2*, two point objects.

This function is the same as **draw-arrow***, except that the positions are specified by points, not by x and y positions.

draw-arrow* [Function]

Arguments: *stream x1 y1 x2 y2* &key from-head (to-head t) (head-length 10) (head-width 5) line-style line-thickness line-unit line-dashes line-cap-shape ink clipping-region transformation

■ Draws an arrow on *stream*. The arrow starts at the position specified by *x1*,*y1* and ends with the arrowhead at the position specified by *x2*,*y2*.

draw-polygon [Function]

Arguments: *medium point-sequence-1* &key (closed t) (filled t) line-style line-thickness line-unit line-dashes line-joint-shape line-cap-shape ink clipping-region transformation

■ Draws a polygon, or sequence of connected lines, on *medium*. The keyword arguments control whether the polygon is closed (each segment is connected to two other segments) and filled. *point-sequence-1* is a list of points which indicate the start of a new line segment.

This function is the same as **draw-polygon***, except that the segments are specified by points, not x and y positions.

draw-polygon* [Function]

Arguments: *medium list-of-x-and-ys* &key (closed t) (filled t) line-style line-thickness line-unit line-dashes line-joint-shape line-cap-shape ink clipping-region transformation

■ Draws a polygon, or sequence of connected lines, on *medium*. The keyword arguments control whether the polygon is closed (each segment is connected to two other segments) and filled. *list-of-x-and-ys* is a list of alternating *x* and *y* positions which indicate the start of a new line segment.

filled Specifies whether the polygon should be filled, a boolean value. If t, a closed polygon is drawn and filled in. In this case, *closed* is assumed to be t.

closed When t, specifies that a segment is drawn connecting the ending point of the last segment to the starting point of the first segment.

Both **draw-polygon*** and **draw-polygon** call **medium-draw-polygon*** to do the actual drawing.

draw-rectangle [Function]

Arguments: *medium point1 point2* &key (filled t) line-style line-thickness line-unit line-dashes line-joint-shape ink clipping-region transformation

■ Draws an axis-aligned rectangle on *medium*. The boundaries of the rectangle are specified by the two points *point1* and *point2*.

This function is the same as **draw-rectangle***, except that the positions are specified by points, not by x and y positions.

draw-rectangle* [Function]

Arguments: *medium x1 y1 x2 y2* &key (filled t) line-style line-thickness line-unit line-dashes line-joint-shape ink clipping-region transformation

■ Draws an axis-aligned rectangle on *medium*. The boundaries of the rectangle are specified by *x1*, *y1*, *x2*, and *y2*, with (*x1*,*y1*) at the upper left and (*x2*,*y2*) at the lower right in the standard +Y-downward coordinate system.

Both **draw-rectangle*** and **draw-rectangle** call **medium-draw-rectangle*** to do the actual drawing.

draw-rectangles [Function]

Arguments: *medium point-seq* &key (filled t) line-style line-thickness line-unit line-dashes line-joint-shape ink clipping-region transformation

■ Draws a set of axis-aligned rectangles on *medium*. *point-seq* is a sequence of pairs of points. Each point specifies the upper left and lower right corner of the rectangle in the standard +Y-downward coordinate system.

This function is equivalent to calling **draw-rectangle** repeatedly, but it can be more convenient and more efficient when drawing more than one rectangle. See the function **draw-rectangle**.

draw-rectangles* [Function]

Arguments: *medium coord-seq* &key (filled t) line-style line-thickness line-unit line-dashes line-joint-shape ink clipping-region transformation

■ Draws a set of axis-aligned rectangles on *medium*. *coord-seq* is a sequence of 4-tuples *x1*, *y1*, *x2*, and *y2*, with (x1,y1) at the upper left and (x2,y2) at the lower right in the standard +Y-downward coordinate system.

This function is equivalent to calling **draw-rectangle*** repeatedly, but it can be more convenient and more efficient when drawing more than one rectangle. See the function **draw-rectangle***.

Both **draw-rectangles*** and **draw-rectangles** call **medium-draw-rectangles*** to do the actual drawing.

draw-ellipse [Function]

Arguments: *medium point-1 radius-1-dx radius-1-dy radius-2-dx radius-2-dy* &key start-angle end-angle (filled t) line-style line-thickness line-unit line-dashes line-cap-shape ink clipping-region transformation

IMPLEMENTATION LIMITATION: Because of limitations in X, only ellipses with axes aligned with the X and Y axes will be drawn. If you specify an ellipse with other axes, X (and thus CLIM) will draw an ellipse with aligned axes anyway.

■ Draws an ellipse or elliptical arc on *medium*. The center of the ellipse is specified by *point*.

This function is the same as **draw-ellipse***, except that the center position is expressed as a point instead of x and y. See the function **draw-ellipse***.

■ Two vectors, *radius-1-dx*, *radius-1-dy*, and *radius-2-dx*, *radius-2-dy* specify the bounding parallelogram of the ellipse. Those two vectors must not be collinear in order for the ellipse to be well defined. The special case of an ellipse with its major axes aligned with the coordinate axes can be obtained by setting both *radius-1-dy* and *radius-2-dx* to 0. For more information about the bounding parallelogram of an ellipse, see the section 3.6.8 **Ellipses and elliptical arcs in CLIM**.

start-angle and *end-angle* enable you to draw an arc rather than a complete ellipse. Angles are measured with respect to the positive *x* axis. The elliptical arc runs positively from *start-angle* to *end-angle*. The angles are measured from the positive *x* axis toward the positive *y* axis. In a right-handed coordinate system this direction is counter-clockwise.

The defaults for *start-angle* and *end-angle* are `nil` (that is, there is no default). If you supply *start-angle*, then *end-angle* defaults to 2π. If you supply *end-angle*, then *start-angle* defaults to 0.

filled specifies whether the ellipse should be filled, a boolean value.

In the case of a filled arc, the figure drawn is the pie slice area swept out by a line from the center of the ellipse to a point on the boundary as the boundary point moves from *start-angle* to *end-angle*.

When drawing unfilled ellipses, the current line style affects the drawing as usual, except that the joint shape has no effect. The dashing of an elliptical arc starts at *start-angle*.

draw-ellipse* [Function]

Arguments: *medium center-x center-y radius-1-dx radius-1-dy radius-2-dx radius-2-dy* &key (filled t) start-angle end-angle line-style line-thickness line-unit line-dashes line-cap-shape ink clipping-region transformation

IMPLEMENTATION LIMITATION: Because of limitations in X, only ellipses with axes aligned with the X and Y axes will be drawn. If you specify an ellipse with other axes, X (and thus CLIM) will draw an ellipse with aligned axes anyway.

■ Draws an ellipse or elliptical arc on *medium*. The center of the ellipse is specified by *center-x* and *center-y*.

This function is the same as **draw-ellipse**, except that the center position is expressed as its x and y coordinates, instead of as a point. See the function **draw-ellipse**.

■ Two vectors, *radius-1-dx*, *radius-1-dy*, and *radius-2-dx*, *radius-2-dy* specify the bounding parallelogram of the ellipse. Those two vectors must not be collinear in order for the ellipse to be well defined. The special case of an ellipse with its major axes aligned with the coordinate axes can be obtained by setting both *radius-1-dy* and *radius-2-dx* to 0. For more information about the bounding parallelogram of an ellipse, see the section 3.6.8 **Ellipses and elliptical arcs in CLIM**.

start-angle and *end-angle* enable you to draw an arc rather than a complete ellipse. Angles are measured with respect to the positive *x* axis. The elliptical arc runs positively from *start-angle* to *end-angle*. The angles are measured from the positive *x* axis toward the positive *y* axis. In a right-handed coordinate system this direction is counter-clockwise.

The defaults for *start-angle* and *end-angle* are `nil` (that is, there is no default). If you supply *start-angle*, then *end-angle* defaults to 2π. If you supply *end-angle*, then *start-angle* defaults to 0.

filled specifies whether the ellipse should be filled, a boolean value.

In the case of a filled arc, the figure drawn is the pie slice area swept out by a line from the center of the ellipse to a point on the boundary as the boundary point moves from *start-angle* to *end-angle*.

When drawing unfilled ellipses, the current line style affects the drawing as usual, except that the joint shape has no effect. The dashing of an elliptical arc starts at *start-angle*.

Both **draw-ellipse*** and **draw-ellipse** call **medium-draw-ellipse*** to do the actual drawing.

draw-circle [Function]

Arguments: *medium center radius* &key (filled t) start-angle end-angle line-style line-thickness line-unit line-dashes line-cap-shape ink clipping-region transformation

■ Draws a circle or arc on *medium*. The center of the circle is specified by the point *center*, and the radius is specified by *radius*.

This function is the same as **draw-circle***, except that the center position is expressed as a point instead of x and y. See the function **draw-circle***.

start-angle and *end-angle* enable you to draw an arc rather than a complete circle in the same manner as that of the ellipse functions. See the function **draw-ellipse***. The defaults for *start-angle* and *end-angle* are `nil` (that is, there is no default).

filled specifies whether the circle should be filled, a boolean value.

draw-circle* [Function]

Arguments: *medium center-x center-y radius* &key (filled t) start-angle end-angle line-style line-thickness line-unit line-dashes line-cap-shape ink clipping-region transformation

■ Draws a circle or arc on *medium*. The center of the circle is specified by *center-x* and *center-y*, and the radius is specified by *radius*.

start-angle and *end-angle* enable you to draw an arc rather than a complete circle in the same manner as that of the ellipse functions. See the function **draw-ellipse***.

The defaults for *start-angle* and *end-angle* are nil (that is, there is no default).

filled specifies whether the circle should be filled, a boolean value.

draw-oval [Function]

Arguments: *stream point-1 x-radius y-radius* &key (filled t) line-style line-thickness line-unit line-dashes line-cap-shape ink clipping-region transformation

■ Draws an oval, that is, a race-track shape, centered on *point-1*, a point object. If *x-radius* or *y-radius* is 0, draws a circle with the specified non-zero radius; otherwise, draws the figure that results from drawing a rectangle with dimensions *x-radius* and *y-radius* and then replacing the two short sides with semicircular arc of appropriate size.

draw-oval* [Function]

Arguments: *stream center-x center-y x-radius y-radius* &key (filled t) line-style line-thickness line-unit line-dashes line-cap-shape ink clipping-region transformation

■ Draws an oval, that is, a race-track shape, centered on (*center-x* center-y): if *x-radius* or *y-radius* is 0, draws a circle with the specified non-zero radius; otherwise, draws the figure that results from drawing a rectangle with dimensions *x-radius* and *y-radius* and then replacing the two short sides with semicircular arc of appropriate size.

draw-bezier-curve [Function]

Arguments: *medium points* &key (filled nil) line-style line-thickness line-unit line-dashes line-cap-shape ink clipping-region transformation

■ Fits and draws a Bezier curve using the points specified by *points*.

draw-bezier-curve* [Function]

Arguments: *medium position-seq* &key (filled nil) line-style line-thickness line-unit line-dashes line-cap-shape ink clipping-region transformation

■ Fits and draws a Bezier curve using the points specified by *position-seq*.

draw-text [Function]

Arguments: *medium text point* &key (start 0) end (align-x :left) (align-y :baseline) text-style text-family text-face text-size ink clipping-region transformation towards-point transform-glyphs

■ Draws *text* onto *medium* starting at the position specified by *point*.

This function is the same as **draw-text***, except that the position is expressed as a point instead of as x and y coordinate values. See the function **draw-text*** defined next.

IMPLEMENTATION LIMITATION: In the current release, text can only be vertical or horizontal. The system will determine which or vertical or horizontal most closely corresponds with the :towards-point argument. If text is printed vertically, each glyph is rotated appropriately regardless of the value of :transform-glyphs.

draw-text* [Function]

Arguments: *medium text x y* &key (start 0) end (align-x :left) (align-y :baseline) text-style text-family text-face text-size ink clipping-region transformation towards-x towards-y transform-glyphs

■ Draws *text* onto *medium* starting at the position specified by *x* and *y*. The exact definition of *starting at* is dependent on *align-x* and *align-y*; by default, the first glyph is drawn with its left edge and its baseline at the position specified by *x* and *y*.

■ The arguments are designed to behave as follows:

align-x

Specifies the horizontal placement of the text string. It can be one of :left (the default), :right, or :center.

:left means that the left edge of the first character of the string is at the specified x coordinate. :right means that the right edge of the last character of the string is at the specified x coordinate. :center means that the string is horizontally centered over the specified x coordinate.

align-y

Specifies the vertical placement of the string. It can be one of :baseline (the default), :top, :bottom, or :center.

:baseline means that the baseline of the string is placed at the specified y coordinate. :top means that the top of the string is at the specified y coordinate. :bottom means that the bottom of the string is at the specified y coordinate. :center means that the string is vertically centered over the specified y coordinate.

start

end

Specify what part of *text* to draw. *start* defaults to 0 and *end* defaults to the end of the string.

towards-x

towards-y

transform-glyphs

The line drawn between (*x*,*y*) and (*towards-x*,*towards-y*) give the baseline along which the glyphs should be placed. The glyphs are rotated so that their baseline is parallel to this baseline if and only if *transform-glyphs* is t.

IMPLEMENTATION LIMITATION: In this release, text can only be vertical or horizontal. The system will determine which or vertical or horizontal most closely corresponds with the :towards-x and :towards-y arguments. If text is printed vertically, each glyph is rotated appropriately regardless of the value of :transform-glyphs.

Both **draw-text*** and **draw-text** call **medium-draw-text*** to do the actual drawing.

3.4 Medium-level drawing functions in CLIM

The medium-level drawing functions are the lowest level, portable functions for doing graphical output. They bypass all other high-level facilities, including output recording. They take no drawing options because these are extracted from the medium. You should use these when performance is the most important.

`medium-draw-point*` [Generic function]

Arguments: *`medium x y`*

■ Draws a point on the medium *`medium`*. The point is drawn at (*`x`*,y), transformed by the medium's current transformation. The ink, clipping region, and line style are gotten from the medium.

`medium-draw-points*` [Generic function]

Arguments: *`medium position-seq`*

Draws a set of points on the medium *`medium`*. *`position-seq`* is a sequence of coordinate pairs, which are real numbers. It is an error if *`position-seq`* does not contain an even number of elements. The coordinates in *`position-seq`* are transformed by the medium's current transformation. The ink, clipping region, and line style are gotten from the medium.

`medium-draw-line*` [Generic function]

Arguments: *`medium x1 y1 x2 y2`*

■ Draws a line on the medium *`medium`*. The line is drawn from (*`x1`*,y1) to (*`x2`*,y2), with the start and end positions transformed by the medium's current transformation. The ink, clipping region, and line style are gotten from the medium.

`medium-draw-lines*` [Generic function]

Arguments: *`medium position-seq`*

■ Draws a set of disconnected lines on the medium *`medium`*. *`position-seq`* is a sequence of coordinate pairs, which are real numbers. It is an error if *`position-seq`* does not contain an even number of elements. The coordinates in *`position-seq`* are transformed by the medium's current transformation. The ink, clipping region, and line style are gotten from the medium.

`medium-draw-rectangle*` [Generic function]

Arguments: *`medium x1 y1 x2 y2 filled`*

■ Draws a rectangle on the medium *`medium`*. The corners of the rectangle are at (*`x1`*,y1) and (*`x2`*,y2), with the corner positions transformed by the medium's current transformation. If *`filled`* is `t`, the rectangle is filled, otherwise it is not. The ink, clipping region, and line style are gotten from the medium.

`medium-copy-area` [Generic function]

Arguments: *`from-medium from-x from-y width height to-medium to-x to-y`* `&optional (`*`boole-fun`* `boole-1)`

■ Copies the pixels from *`from-medium`* starting at the position specified by (*`from-x`*,*`from-y`*) to the position (*`to-x`*,*`to-y`*) on *`to-medium`*. A rectangle whose width and height is specified by *`width`* and *`height`* is copied. *`from-x`*, *`from-y`*, *`to-x`*, and *`to-y`* are specified in user coordinates. (If *`medium`* is a sheet or a stream, then *`from-x`* and *`from-y`* are transformed by the user transformation.)

■ The pixels are copied as if with the boolean operation specified by *boole-fun*. The value of that argument should one of the boole-*xxx* constants and defaults to boole-1, meaning that the source bits are set into the destination.

Arguments:

medium-draw-rectangles* [Generic function]

Arguments: *medium position-seq filled*

■ Draws a set of rectangles on the medium *medium*. *position-seq* is a sequence of coordinate pairs, which are real numbers. It is an error if *position-seq* does not contain an even number of elements. The coordinates in *position-seq* are transformed by the medium's current transformation. If *filled* is t, the rectangle is filled, otherwise it is not. The ink, clipping region, and line style are gotten from the medium.

medium-draw-polygon* [Generic function]

Arguments: *medium position-seq closed filled*

■ Draws a polygon or polyline on the medium *medium*. *position-seq* is a sequence of coordinate pairs, which are real numbers. It is an error if *position-seq* does not contain an even number of elements. The coordinates in *position-seq* are transformed by the medium's current transformation.

If *filled* is t, the polygon is filled, otherwise it is not. If *closed* is t, the coordinates in *position-seq* are considered to define a close polygon, otherwise the polygon will not be closed. The ink, clipping region, and line style are gotten from the medium.

medium-draw-ellipse* [Generic function]

Arguments: *medium center-x center-y radius-1-dx radius-1-dy radius-2-dx radius-2-dy start-angle end-angle filled*

IMPLEMENTATION LIMITATION: Because of limitations in X, only ellipses with axes aligned with the X and Y axes will be drawn. If you specify an ellipse with other axes, X (and thus CLIM) will draw an ellipse with aligned axes anyway.

■ Draws an ellipse on the medium *medium*. The center of the ellipse is at (*x*,y), and the radii are specified by the two vectors (*radius-1-dx*, *radius-1-dy*) and (*radius-2-dx*,*radius-2-dy*). The center point and radii are transformed by the medium's current transformation.

start-angle and *end-angle* are real numbers that specify an arc rather than a complete ellipse. The medium transformation must be applied to the angles as well. If *filled* is t, the ellipse is filled, otherwise it is not. The ink, clipping region, and line style are gotten from the medium.

medium-draw-text* [Generic function]

Arguments: *medium string-or-char x y start end align-x align-y towards-x towards-y transform-glyphs*

■ Draws a character or a string (specified by the *string-or-char* argument) on the medium *medium*. The text is drawn starting at (*x*,*y*), and towards (*toward-x*,*toward-y*); these positions are transformed by the medium's current transformation. The individual; glyphs are rotated to align with the line of text as *transform-glyphs* is true of false. The ink, clipping region, and line style are gotten from the medium.

IMPLEMENTATION LIMITATION: In the current release, text can only be vertical or horizontal. The system will determine which or vertical or horizontal most closely corresponds with the *towards-x* and *towards-y* arguments. If text is printed vertically, each glyph is rotated appropriately regardless of the value of *transform-glyphs*.

3.5 Pixmaps in CLIM

A *pixmap* can be thought of as an off-screen window, that is, a medium that can be used for graphical output, but is not visible on any display device. Pixmaps are provided to allow a programmer to generate a piece of output associated with some display device that can then later be rapidly drawn on a real display device. For example, an electrical CAD system might generate a pixmap that corresponds to a complex, frequently used part in a VLSI schematic, and then use **draw-pixmap** or **copy-from-pixmap** to draw the part as needed.

The exact representation of a pixmap is explicitly unspecified.

> **Note that there is no interaction between the pixmap operations and output recording**, that is, displaying a pixmap on a sheet is a pure drawing operation that affects only the display, not the output history. Some mediums may not support pixmaps (such as PostScript mediums); in this case, an error will be signaled.

copy-to-pixmap [Function]

Arguments: *medium medium-x medium-y width height* &optional *pixmap* (*pixmap-x* 0) (*pixmap-y* 0) (*boole-fun* boole-1)

■ Copies the pixels from the medium *medium* starting at the position specified by (*medium-x*,*medium-y*) into the pixmap *pixmap* at the position specified by (*pixmap-x*,*pixmap-y*). A rectangle whose width and height is specified by *width* and *height* is copied. *medium-x* and *medium-y* are specified in user coordinates. (If *medium* is a sheet or a stream, then *medium-x* and *medium-y* are transformed by the user transformation.)

If *pixmap* is not supplied, a new pixmap will be allocated. Otherwise, *pixmap* must be an object returned by **allocate-pixmap** that has the appropriate characteristics for *medium*.

The pixels are copied as if with the boolean operation specified by *boole-fun*. The value of that argument should one of the boole-*xxx* constants and defaults to boole-1, meaning that the source bits are set into the destination.

Note that **copy-to-pixmap** does not record any information on an output record. **draw-pixmap** and **draw-pixmap*** both do add to the output record.

■ The returned value is the pixmap.

copy-from-pixmap [Function]

Arguments: *pixmap pixmap-x pixmap-y width height medium medium-x medium-y* &optional (*boole-fun* boole-1)

■ Copies the pixels from the pixmap *pixmap* starting at the position specified by (*pixmap-x*,*pixmap-y*) into the medium *medium* at the position (*medium-x*,*medium-y*). A rectangle whose width and height is specified by *width* and *height* is copied. *medium-x* and *medium-y* are specified in user coordinates. (If *medium* is a sheet or a stream, then *medium-x* and *medium-y* are transformed by the user transformation.)

pixmap must be an object returned by **allocate-pixmap** that has the appropriate characteristics for *medium*.

As for **copy-to-pixmap**, the pixels are copied as if with the boolean operation specified by *boole-fun*. The value of that argument should one of the boole-*xxx* constants and defaults to boole-1, meaning that the source bits are set into the destination.

■ The returned value is the pixmap.

draw-pixmap* [Function]

Arguments: *medium pixmap x y* &rest *args* &key :ink :clipping-region :transformation (:function boole-1)

■ Draws the pixmap *pixmap* on *medium* at the position (*x*,*y*). *pixmap* is a pixmap created by using **copy-area** or **with-output-to-pixmap**. Unlike **copy-area**, **draw-pixmap*** will create a ''pixmap output record'' when called on an output recording stream.

■ :ink, :clipping-region, :transformation are the usual sort of drawing options. :function is a boolean operator that specifies how *pixmap* should be combined with the destination; it is the same as for **copy-area**.

draw-pixmap [Function]

Arguments: *medium pixmap point* &rest *args* &key :ink :clipping-region :transformation (:function boole-1)

■ Draws the pixmap *pixmap* on *medium* at the position *point*. This function is the same as **draw-pixmap***, except that the *position* is specified by a point object, not by an X/Y position.

copy-area [Generic function]

Arguments: *medium from-x from-y width height to-x to-y* &optional (*boole-fun* boole-1)

■ Copies the pixels from the medium *medium* starting at the position specified by (*from-x*,*from-y*) to the position (*to-x*,*to-y*) on the same medium. A rectangle whose width and height is specified by *width* and *height* is copied. *from-x*, *from-y*, *to-x*, and *to-y* are specified in user coordinates. (If *medium* is a sheet or a stream, then *from-x* and *from-y* are transformed by the user transformation.)

■ As for **copy-to-pixmap**, the pixels are copied as if with the boolean operation specified by *boole-fun*. The value of that argument should one of the boole-*xxx* constants and defaults to boole-1, meaning that the source bits are set into the destination.

allocate-pixmap [Function]

Arguments: *medium width height*

■ Allocates and returns a pixmap object that can be used on any medium that shares the same characteristics as *medium*. (The exact definition of 'shared characteristics' will vary from host to host.) *medium* can be a medium, a sheet, or a stream.

The resulting pixmap will be *width* units wide, *height* units high, and as deep as is necessary to store the information for the medium.

■ The returned value is the pixmap.

deallocate-pixmap [Function]

Arguments: *pixmap*

■ Deallocates the pixmap *pixmap*.

pixmap-width [Generic function]

Arguments: *pixmap*

■ Returns the width of the pixmap *pixmap*.

pixmap-height [Generic function]

Arguments: *pixmap*

■ Returns the height of the pixmap *pixmap*.

pixmap-depth **[Generic function]**

Arguments: *pixmap*

- Returns the depth of the pixmap *pixmap*.

with-output-to-pixmap **[Macro]**

Arguments: (*medium-var medium* &key width height) &body *body*

- Binds *medium-var* to a pixmap medium, that is, a medium that does output to a pixmap with the characteristics appropriate to the medium *medium*, and then evaluates *body* in that context. All the output done to the medium designated by *medium-var* inside of *body* is drawn on the pixmap stream. CLIM implementations are permitted, but not required, to have pixmap mediums support the stream output protocol (**write-char** and **write-string**).

  medium-var must be a symbol; it is not evaluated.

- *width* and *height* are integers that give the width and height of the pixmap. If they are unsupplied, the **with-output-to-output-record** is called to determine the size and then the output record is replayed on the pixmap stream. As a consequence, operations such as **window-clear** will operate on the wrong stream. Specify *width* and *height* if this is a problem.
- The returned value is a pixmap that can be drawn onto *medium* using **copy-from-pixmap**.

3.5.1 Example of Using CLIM Pixmaps

If you run the following code, it will wait for you to input a rectangular region, then it will copy the region into a pixmap. It will then wait for you to indicate another spot on the window stream, and will copy the pixmap out into the new place.

```
(defun test-copy-area (&optional (function boole-1) (stream *standard-output*))
 (let ((medium (clim:sheet-medium stream)))
  (multiple-value-bind (left top right bottom)
    (clim:pointer-input-rectangle* :stream stream)
   (let ((pixmap (clim:copy-to-pixmap
           medium left top (- right left) (- bottom top))))
    (multiple-value-bind (x y)
      (block get-position
       (clim:tracking-pointer (stream)
        (:pointer-button-press (x y)
         (return-from get-position (values x y)))))
     (clim:copy-from-pixmap pixmap 0 0 (- right left) (- bottom top)
                medium x y function))))))
```

The next example creates a pixmap using **with-output-to-pixmap**, and then displays the result at the place you click on.

```
(defun test-pixmaps (&optional (stream *standard-output*))
 (let ((medium (clim:sheet-medium stream)))
  (let ((pixmap (clim:with-output-to-pixmap (mv medium)
          (clim:formatting-table (mv)
           (dotimes (i 5)
            (clim:formatting-row (mv)
             (clim:formatting-cell (mv) (princ i mv))
             (clim:formatting-cell (mv) (princ (* i 2) mv)))))
          (clim:draw-circle* mv 50 50 20 :filled t)
```

```
        (clim:draw-rectangle* mv 0 0 90 90 :filled nil))))
  (multiple-value-bind (x y)
    (block get-position
     (clim:tracking-pointer (stream)
      (:pointer-button-press (x y)
       (return-from get-position (values x y)))))
   (clim:copy-from-pixmap
    pixmap 0 0 (clim:pixmap-width pixmap) (clim:pixmap-height pixmap)
    medium x y)))))
```

3.6 General geometric objects and regions in CLIM

A *region* is an object that denotes a set of points in the plane. Regions include their boundaries, that is, they are closed. Regions have infinite resolution.

A *bounded region* is a region that contains at least one point and for which there exists a number, d, called the region's diameter, such that if p1 and p2 are points in the region, the distance between p1 and p2 is always less than or equal to d.

An *unbounded region* either contains no points or contains points arbitrarily far apart.

Another way to describe a region is that it maps every (x,y) pair into either true or false (meaning member, or not a member, respectively, of the region).

The following classes are what CLIM uses to classify the various types of regions. All regions are a subclass of `region`, and all bounded regions are also a subclass of either `point`, `path`, or `area`. You may wish to subclass these classes to implement such things as graphical editors.

`region` **[Class]**

■ The protocol class that corresponds to a closed set of points. If you want to create a new class that obeys the region protocol, it must be a subclass of `region`.

`point` **[Class]**

■ The protocol class that corresponds to a mathematical point. If you want to create a new class that obeys the point protocol, it must be a subclass of `point`.

`path` **[Class]**

■ This is a subclass of `region` that denotes regions that have dimensionality 1. If you want to create a new class that obeys the path protocol, it must be a subclass of `path`.

Making a `path` object with no length canonicalizes it to `+nowhere+`. When paths are used to construct an area by specifying its outline, they need to have a direction associated with them.

`area` **[Class]**

■ This is a subclass of `region` that denotes regions that have dimensionality 2 (that is, have area). If you want to create a new class that obeys the area protocol, it must be a subclass of `area`.

Making an `area` object with no area canonicalizes it to `+nowhere+`.

The following two constants represent the regions that correspond, respectively, to all of the points on the drawing plane and none of the points on the drawing plane.

`+everywhere+` **[Constant]**

■ The region that includes all the points on the infinite drawing plane. This is the opposite of `+nowhere+`.

`+nowhere+` **[Constant]**

- The empty region (the opposite of `+everywhere+`).

3.6.1 Region predicates in CLIM

The following functions can be used to examine certain aspects of regions, such as whether two regions are equal or if they overlap.

`region-equal` [Generic function]

Arguments: *`region1 region2`*

- Returns `t` if *`region1`* and *`region2`* contain exactly the same set of points, otherwise returns `nil`.

`region-contains-region-p` [Generic function]

Arguments: *`region1 region2`*

- Returns `t` if all points in *`region2`* are members of *`region1`*, otherwise returns `nil`.

`region-contains-position-p` [Generic function]

Arguments: *`region x y`*

- Returns `t` if the point (*x*,*y*) is contained in *`region`*, otherwise returns `nil`. Since regions in CLIM are closed, this will return `t` if (*x*,*y*) is on the region's boundary. This is a special case of **`region-contains-region-p`**.

`region-intersects-region-p` [Generic function]

Arguments: *`region1 region2`*

- Returns `nil` if **`region-intersection`** of the two regions would be `+nowhere+`, otherwise returns `t`.

3.6.2 Composition of CLIM regions

Region composition in CLIM is the process in which two regions are combined in some way (such as union or intersection) to produce a third region.

Since all regions in CLIM are closed, region composition is not always equivalent to simple set operations. Instead, composition attempts to return an object that has the same dimensionality as one of its arguments. If this is not possible, then the result is defined to be an empty region, which is canonicalized to `+nowhere+`. (The exact details of this are specified with each function.)

Sometimes, composition of regions can produce a result that is not a simple contiguous region. For example, **`region-union`** of two rectangular regions might not be rectangular. In order to support cases like this, CLIM has the concept of a *region set*, which is an object that represents one or more region objects related by some region operation, usually a union.

`region-union` [Generic function]

Arguments: *`region1 region2`*

- Returns a region that contains all points that are in either *`region1`* or *`region2`* (possibly with some points removed to satisfy the dimensionality rule).

■ The result of **region-union** always has dimensionality that is the maximum dimensionality of *region1* and *region2*. For example, the union of a path and an area produces an area; the union of two paths is a path.

region-intersection [Generic function]

Arguments: *region1 region2*

■ Returns a region that contains all points that are in both *region1* and *region2* (possibly with some points removed to satisfy the dimensionality rule). The result of **region-intersection** has dimensionality that is the minimum dimensionality of *region1* and *region2*, or is +nowhere+. For example, the intersection of two areas is either another area or +nowhere+; the intersection of two paths is either another path or +nowhere+; the intersection of a path and an area produces the path clipped to stay inside of the area.

region-difference [Generic function]

Arguments: *region1 region2*

■ Returns a region that contains all points in *region1* that are not in *region2* (plus additional boundary points to make the result closed). The result of **region-difference** has the same dimensionality as *region1*, or is +nowhere+. For example, the difference of an area and a path produces the same area; the difference of a path and an area produces the path clipped to stay outside of the area.

region-set [Class]

■ The class that represents region sets; a subclass of region.

region-set-function [Generic function]

Arguments: *region*

■ Returns a symbol representing the operation that relates the regions in *region*. This will be one of the Common Lisp symbols union, intersection, or set-difference. For the case of region sets that are composed entirely of rectangular regions, CLIM canonicalizes the set so that the symbol will always be union. If *region* is a region that is not a region-set, the result is always union.

region-set-regions [Generic function]

Arguments: *region* &key normalize

■ Returns a sequence of the regions in *region*. *region* can be either a region-set or any member of region, in which case the result is simply a sequence of one element: *region*. For the case of region sets that are unions of rectangular regions, CLIM canonicalizes the set so that the rectangles returned by **region-set-regions** are guaranteed not to overlap.

If *normalize* is supplied, it may be :x-banding or :y-banding. If it is :x-banding and all the regions in *region* are rectangles, the result is normalized by merging adjacent rectangles with banding done in the x direction. If it is :y-banding and all the regions in *region* are rectangles, the result is normalized with banding done in the y direction.

■ Normalizing a region set that is not composed entirely of rectangles using x- or y-banding causes CLIM to signal the region-set-not-rectangular error.

map-over-region-set-regions [Generic function]

Arguments: *function region* &key normalize

■ Calls *function* on each region in *region*. This is often more efficient than calling **region-set-regions**. *region* can be either a region-set or any member of region, in which case *function* is called once on *region* itself. *normalize* is as in **region-set-regions**.

3.6.3 CLIM point objects

A *point* is a mathematical point in the drawing plane, which is identified by its coordinates, a pair of real numbers. Points have neither area nor length. Note that a point is not the same thing as a pixel; CLIM's model of the drawing plane has continuous coordinates.

You can create point objects and use them as arguments to the drawing functions. Alternatively, you can use the *spread* versions of the drawing functions, that is the drawing functions with stars appended to their names. For example, instead of **`draw-point`**, use **`draw-point*`** which take two arguments specifying a point by its coordinates. (Note that we generally recommend the use of the spread versions, since the CLIM implementation is optimized for those functions.)

The operations for creating and dealing with points are:

`point` **[Class]**

■ The protocol class that corresponds to a mathematical point. If you want to create a new class that obeys the point protocol, it must be a subclass of `point`.

`standard-point` **[Class]**

■ A class that implements a point. This is the class that **`make-point`** instantiates.

`make-point` **[Function]**

Arguments: *`x y`*

■ Creates and returns a point object whose coordinates are *`x`* and *`y`*. The point object is an instance of `standard`-point.

`point-position` **[Generic function]**

Arguments: *`point`*

■ Returns two values, the x and y coordinates of *`point`*.

`point-x` **[Generic function]**

Arguments: *`point`*

■ Returns the x coordinate of *`point`*.

`point-y` **[Generic function]**

Arguments: *`point`*

■ Returns the y coordinate of *`point`*.

3.6.4 Polygons and polylines in CLIM

A *polyline* is a path that consists of one or more line segments joined consecutively at their end-points. A *line* is a polyline that has only a single segment.

Polylines that have the end-point of their last line segment coincident with the start-point of their first line segment are called *closed*; this use of the term ‘closed’ should not be confused with closed sets of points.

A *polygon* is an area bounded by a closed polyline.

If the boundary of a polygon intersects itself, the odd-even winding-rule defines the polygon: a point is inside the polygon if a ray from the point to infinity crosses the boundary an odd number of times.

The classes that correspond to polylines and polygons are:

`polyline` **[Class]**

■ The protocol class that corresponds to a polyline. This is a subclass of `path`. If you want to create a new class that obeys the polyline protocol, it must be a subclass of `polyline`.

`polygon` **[Class]**

■ The protocol class that corresponds to a mathematical polygon. This is a subclass of `area`. If you want to create a new class that obeys the polygon protocol, it must be a subclass of `polygon`.

`standard-polyline` **[Class]**

■ A class that implements a polyline. This is a subclass of `polyline`. This is the class that **`make-polyline`** and **`make-polyline*`** instantiate.

`standard-polygon` **[Class]**

■ A class that implements a polygon. This is a subclass of `polygon`.

■ This is the class that **`make-polygon`** and **`make-polygon*`** instantiate.

`make-polygon` [Function]

Arguments: *`point-seq`*

■ Makes an object of class `standard-polygon` consisting of the area contained in the boundary that is specified by the segments connecting each of the points in *`point-seq`*.

`make-polygon*` [Function]

Arguments: *`coord-seq`*

■ Makes an object of class `standard-polygon` consisting of the area contained in the boundary that is specified by the segments connecting each of the points represented by the coordinate pairs in *`coord-seq`*.

`make-polyline` [Function]

Arguments: *`point-seq`* `&key closed`

■ Makes an object of class `standard-polyline` consisting of the segments connecting each of the points in *`point-seq`*.

■ If *`closed`* is `t`, the segment connecting the first point and the last point is included in the polyline.

`make-polyline*` [Function]

Arguments: *`coord-seq`* `&key closed`

■ Makes an object of class `standard-polyline` consisting of the segments connecting each of the points represented by the coordinate pairs in *`coord-seq`*.

■ If *`closed`* is `t`, the segment connecting the first point and the last point is included in the polyline.

`polyline-closed` [Generic function]

Arguments: *`polyline`*

■ Returns `t` if *`polyline`* is closed, otherwise returns `nil`.

`polygon-points` [Generic function]

Arguments: *polygon*

■ Returns a sequence of points that specify the segments in *polygon*.

`map-over-polygon-coordinates` [Generic function]

Arguments: *function polygon*

■ Applies *function* to all of the coordinates of the vertices of *polygon*. The *function* takes two arguments, the x and y coordinates.

`map-over-polygon-segments` [Generic function]

Arguments: *function polygon*

■ Applies *function* to the line segments that compose *polygon*. The *function* takes four arguments, the x and y coordinates of the start of the line segment, and the x and y coordinates of the end of the line segment.

■ When **`map-over-polygon-segments`** is called on a closed polyline, it will call *function* on the line segment that connects the last point back to the first point.

3.6.5 Lines in CLIM

A *line* is a special case of a polyline that has only a single segment. The functions for making and dealing with line are the following:

`line` [Class]

■ The protocol class that corresponds to a mathematical line-segment, that is, a polyline with only a single segment. This is a subclass of `polyline`. If you want to create a new class that obeys the line protocol, it must be a subclass of `line`.

`standard-line` [Class]

■ A class that implements a line. This is a subclass of `line`. This is the class that **`make-line`** and **`make-line*`** instantiate.

`make-line` [Function]

Arguments: *start-point end-point*

■ Makes an object of class `standard-line` that connects *start-point* to *end-point*.

`make-line*` [Function]

Arguments: *start-x start-y end-x end-y*

■ Makes an object of class `standard-line` that connects (*start-x*, *start-y*) to (*end-x*, *end-y*).

`line-start-point` [Generic function]

Arguments: *line*

■ Returns the starting point of *line*.

`line-end-point` [Generic function]

Arguments: *line*

■ Returns the ending point of *line*.

`line-start-point*` [Generic function]

Arguments: *line*

- Returns the starting point of *line* as two values representing the coordinate pair.

`line-end-point*` [Generic function]

Arguments: *line*

- Returns the ending point of *line* as two values representing the coordinate pair.

3.6.6 Rectangles in CLIM

A *rectangle* is a special case of a four-sided polygon whose edges are parallel to the coordinate axes. A rectangle can be specified completely by four real numbers (*min-x*, *min-y*, *max-x*, *max-y*). They are not closed under affine transformations.

The functions for creating and dealing with rectangles are the following:

`rectangle` [Class]

- The protocol class that corresponds to an axis-aligned mathematical rectangle, that is, rectangular polygons whose sides are parallel to the coordinate axes. This is a subclass of `polygon`. If you want to create a new class that obeys the rectangle protocol, it must be a subclass of `rectangle`.

`standard-rectangle` [Class]

- A class that implements a rectangle. This is a subclass of `rectangle`. This is the class that **`make-rectangle`** and **`make-rectangle*`** instantiate.

`make-rectangle` [Function]

Arguments: *min-point max-point*

- Makes an object of class `standard-rectangle` whose edges are parallel to the coordinate axes. One corner is at *min-point* and the opposite corner is at *max-point*.

`make-rectangle*` [Function]

Arguments: *min-x min-y max-x max-y*

- Makes an object of class `standard-rectangle` whose edges are parallel to the coordinate axes. One corner is at (*min-x*, *min-y*) and the opposite corner is at (*max-x*,*max-y*).

The representation of rectangles in CLIM is chosen to be efficient. CLIM represents rectangles by storing the coordinates of two opposing corners of the rectangle. Because this representation is not sufficient to represent the result of arbitrary transformations of arbitrary rectangles, CLIM is allowed to return a polygon as the result of such a transformation. (The most general class of transformations that is guaranteed to always turn a rectangle object into another rectangle object is the class of transformations that satisfy **`rectilinear-transformation-p`**.)

`rectangle-min-point` [Generic function]

Arguments: *rectangle*

- Returns the minimum point of *rectangle*. The position of a rectangle is specified by its minimum point.

rectangle-max-point [Generic function]

Arguments: *rectangle*

■ Returns the maximum point of *rectangle*. (The position of a rectangle is specified by its minimum point).

rectangle-edges* [Generic function]

Arguments: *rectangle*

■ Returns the coordinate of the minimum x and y and maximum x and y of *rectangle* as four values.

rectangle-min-x [Function]

Arguments: *rectangle*

■ Returns the coordinate of the minimum x of *rectangle*.

rectangle-min-y [Function]

Arguments: *rectangle*

■ Returns the coordinate of the minimum y of *rectangle*.

rectangle-max-x [Function]

Arguments: *rectangle*

■ Returns the coordinate of the maximum x of *rectangle*.

rectangle-max-y [Function]

Arguments: *rectangle*

■ Returns the coordinate of the maximum y of *rectangle*.

rectangle-width [Function]

Arguments: *rectangle*

■ Returns the width of *rectangle*. The width of a rectangle is the difference between the maximum x and the minimum x. (The height of a rectangle is difference between the maximum y and the minimum y.)

rectangle-height [Function]

Arguments: *rectangle*

■ Returns the height of *rectangle*. The height is the difference between the maximum y and the minimum y. (The width of a rectangle is the difference between the maximum x and the minimum x.)

rectangle-size [Function]

Arguments: *rectangle*

■ Returns two values, the width and the height of *rectangle*.

3.6.7 Bounding Rectangles in CLIM

Every bounded region in CLIM has a derived *bounding rectangle*, which is the smallest rectangle that contains every point in the region, and may contain additional points as well. Unbounded regions do not have any bounding rectangle. For example, all windows and output records have bounding rectangles whose coordinates are relative to the bounding rectangle of the parent of the window or output record.

Note that bounding-rectangles are distinct from rectangles. Bounding-rectangles are used internally in CLIM for a number of purposes. Note that when you are asked to provide a bounding-rectangle, supplying a rectangle instead may not cause an immediate error but will likely cause a subsequent failure, often one that is hard to diagnose.

`bounding-rectangle` **[Class]**

■ The protocol class that represents a bounding rectangle. Note that bounding rectangles are not a subclass of rectangle, nor even a subclass of region. This is because in general, bounding rectangles do not obey the region protocols. However, all bounded rectangles and sheets that obey the bounding rectangle protocol are subclasses of bounding-rectangle.

`standard-bounding-rectangle` **[Class]**

■ An instantiable class that implements a bounding rectangle. this is a subclass of both rectangle and bounding-rectangle, that is standard bounding rectangles obey the rectangle protocol. make-bounding-rectangle returns an object of this class.

`make-bounding-rectangle` [Function]

Arguments: *x1 y1 x2 y2*

■ Returns an object of the class `standard-bounding-rectangle` with the edges specified by the arguments, which must be real numbers. Two opposite corners of the resulting bounding rectangle have coordinates (`(min` *x1 x2*`)`, `(min` *y1 y2*`)`) and (`(max` *x1 x2*`)`, `(max` *y1 y2*`)`), with the other two corners determined in the obvious way.

The following functions can be used to access the bounding rectangle of a region.

`bounding-rectangle*` [Generic function]

Arguments: *region*

■ Returns the bounding rectangle of *`region`* as four real numbers specifying the left, top, right, and bottom edges of the bounding rectangle. *`region`* must be a bounded region, such as an output record, a window, or a geometric object such as a line or an ellipse.

■ The coordinates of the bounding rectangle of windows and output records are maintained relative to the parent of the window or output record.

`bounding-rectangle` [Generic function]

Arguments: *region* `&optional` *reuse-rectangle*

■ Returns the bounding rectangle of *`region`* as a `rectangle` object. *`region`* is as for **`bounding-rectangle*`**.

bounding-rectangle-min-x [Function]

Arguments: *region*

bounding-rectangle-left [Function]

Arguments: *region*

■ Returns the coordinate of the left edge of the bounding rectangle of *region*. *region* is as for **bounding-rectangle***.

bounding-rectangle-min-y [Function]

Arguments: *region*

bounding-rectangle-top [Function]

Arguments: *region*

■ Returns the coordinate of the top edge of the bounding rectangle of *region*. *region* is as for **bounding-rectangle***.

bounding-rectangle-max-x [Function]

Arguments: *region*

bounding-rectangle-right [Function]

Arguments: *region*

■ Returns the coordinate of the right edge of the bounding rectangle of *region*. *region* is as for **bounding-rectangle***.

bounding-rectangle-max-y [Function]

Arguments: *region*

bounding-rectangle-bottom [Function]

Arguments: *region*

■ Returns the coordinate of the bottom edge of the bounding rectangle of *region*. *region* is as for **bounding-rectangle***.

bounding-rectangle-size [Function]

Arguments: *region*

■ Returns the size (as two values, width and height) of the bounding rectangle of *region*. *region* is as for **bounding-rectangle***.

bounding-rectangle-width [Function]

Arguments: *region*

■ Returns the width of the bounding rectangle of *region*. *region* is as for **bounding-rectangle***.

bounding-rectangle-height [Function]

Arguments: *region*

■ Returns the height of the bounding rectangle of *region*. *region* is as for **bounding-rectangle***.

For example, the size of a piece of output (produced by a body of code specified by *body*) can be determined by calling **bounding-rectangle-size** on the output record:

```
(let ((record (clim:with-output-to-output-record (s) body)))
 (multiple-value-bind (width height)
   (clim:bounding-rectangle-size record)
  (format t "~&Width is ~D, height is ~D" width height)))
```

3.6.8 Ellipses and Elliptical Arcs in CLIM

An *ellipse* is an area that is the outline and interior of an ellipse. Circles are special cases of ellipses.

An *elliptical arc* is a path consisting of all or a portion of the outline of an ellipse. Circular arcs are special cases of elliptical arcs.

An ellipse is specified in a manner that is easy to transform, and treats all ellipses on an equal basis. An ellipse is specified by its center point and two vectors that describe a bounding parallelogram of the ellipse. The bounding parallelogram is made by adding and subtracting the vectors from the center point in the following manner:

	x coordinate	y coordinate
Center of Ellipse	x_c	y_c
Vectors	dx_1 dx_2	dy_1 dy_2
Corners of parallelogram	$x_c + dx_1 + dx_2$ $x_c + dx_1 - dx_2$ $x_c - dx_1 - dx_2$ $x_c - dx_1 + dx_2$	$y_c + dy_1 + dy_2$ $y_c + dy_1 - dy_2$ $y_c - dy_1 - dy_2$ $y_c - dy_1 + dy_2$

The special case of an ellipse with its axes aligned with the coordinate axes can be obtained by setting $dx_2 = dy_1 = 0$ or $dx_1 = dy_2 = 0$. Note that several different parallelograms specify the same ellipse.

The following classes and functions are used to represent and operate on ellipses and elliptical arcs.

`ellipse` **[Class]**

■ The protocol class that corresponds to a mathematical ellipse. This is a subclass of `area`. If you want to create a new class that obeys the ellipse protocol, it must be a subclass of `ellipse`.

`elliptical-arc` **[Class]**

■ The protocol class that corresponds to a mathematical elliptical arc. This is a subclass of `path`. If you want to create a new class that obeys the elliptical arc protocol, it must be a subclass of `elliptical-arc`.

`standard-ellipse` **[Class]**

■ A class that implements an ellipse. This is a subclass of `ellipse`. This is the class that **make-ellipse** and **make-ellipse*** instantiate.

`standard-elliptical-arc` **[Class]**

■ A class that implements an elliptical arc. This is a subclass of `elliptical-arc`. This is the class that **make-elliptical-arc** and **make-elliptical-arc*** instantiate.

make-ellipse [Function]

Arguments: *center-point radius-1-dx radius-1-dy radius-2-dx radius-2-dy* &key start-angle end-angle

■ Makes an object of class standard-ellipse. The center of the ellipse is *center-point*.

This function is the same as **make-ellipse*** except that the location of the center of the ellipse is specified as a point rather than as X and Y coordinates.

make-ellipse* [Function]

Arguments: *center-x center-y radius-1-dx radius-1-dy radius-2-dx radius-2-dy* &key start-angle end-angle

■ Makes an object of class standard-ellipse. The center of the ellipse is (*center-x*, *center-y*).

■ Two vectors, (*radius-1-dx*, *radius-1-dy*) and (*radius-2-dx*, *radius-2-dy*) specify the bounding parallelogram of the ellipse as explained above. It is an error for those two vectors to be collinear (in order for the ellipse to be well-defined). The special case of an ellipse with its axes aligned with the coordinate axes can be obtained by setting both *radius-1-dy* and *radius-2-dx* to 0.

If *start-angle* or *end-angle* are supplied, the ellipse is the pie slice area swept out by a line from the center of the ellipse to a point on the boundary as the boundary point moves from *start-angle* to *end-angle*. Angles are measured counter-clockwise with respect to the positive x axis. If *end-angle* is supplied, the default for *start-angle* is 0; if *start-angle* is supplied, the default for *end-angle* is 2π; if neither is supplied then the region is a full ellipse and the angles are meaningless.

make-elliptical-arc [Function]

Arguments: *center-point radius-1-dx radius-1-dy radius-2-dx radius-2-dy* &key start-angle end-angle

■ Makes an object of class standard-elliptical-arc. The center of the ellipse is *center-point*.

■ This function is the same as **make-elliptical-arc*** except that the location of the center of the arc is specified as a point rather than as X and Y coordinates.

make-elliptical-arc* [Function]

Arguments: *center-x center-y radius-1-dx radius-1-dy radius-2-dx radius-2-dy* &key start-angle end-angle

■ Makes an object of class standard-elliptical-arc. The center of the ellipse is (*center-x*,*center-y*).

■ Two vectors, (*radius-1-dx*, *radius-1-dy*) and (*radius-2-dx*, *radius-2-dy*), specify the bounding parallelogram of the ellipse as explained above. It is an error for those two vectors to be collinear (in order for the ellipse to be well-defined). The special case of an elliptical arc with its axes aligned with the coordinate axes can be obtained by setting both *radius-1-dy* and *radius-2-dx* to 0.

■ The arc is swept from *start-angle* to *end-angle*. Angles are measured counter-clockwise with respect to the positive x axis. If *end-angle* is supplied, the default for *start-angle* is 0; if *start-angle* is supplied, the default for *end-angle* is 2π; if neither is supplied then the region is a closed elliptical path and the angles are meaningless.

ellipse-center-point [Generic function]

Arguments: *ellipse*

- Returns the center point of *ellipse*.

ellipse-center-point* [Generic function]

Arguments: *ellipse*

- Returns the center point of *ellipse* as two values representing the coordinate pair.

ellipse-radii [Generic function]

Arguments: *ellipse*

- Returns four values corresponding to the two radius vectors of *ellipse*. These values may be canonicalized in some way, and so may not be the same as the values passed to the constructor function.

ellipse-start-angle [Generic function]

Arguments: *ellipse*

- Returns the start angle of *ellipse*. If *elliptical-object* is a full ellipse or closed path then **ellipse-start-angle** will return `nil`; otherwise the value will be a number greater than or equal to zero, and less than 2π.

ellipse-end-angle [Generic function]

Arguments: *ellipse*

- Returns the end angle of *ellipse*. If *elliptical-object* is a full ellipse or closed path then **ellipse-end-angle** will return `nil`; otherwise the value will be a number greater than zero, and less than or equal to 2π.

[This page intentionally left blank.]

Chapter 4 The CLIM drawing environment

4.1 Introduction to CLIM drawing environments

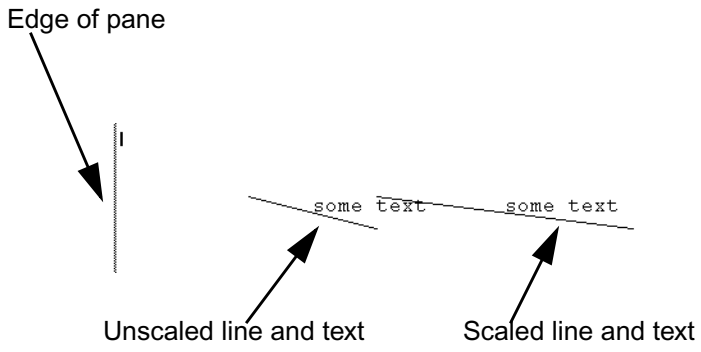
There are a number of factors which affect drawing in CLIM. Drawing might be affected by transformations, line style and text style options, clipping, and by the ink that is used. All of these are controlled by the drawing environment.

When you draw in CLIM, you do so on a medium. A medium can be thought of as a drawing surface plus a *graphics context* in which the medium keeps track of its drawing environment, which consists of the current transformation, text style, foreground and background inks, and so forth.

The drawing environment is dynamic. The CLIM facilities for affecting the drawing environment do so within their dynamic extent. For example, drawing done by any CLIM drawing function (such as **`draw-line`** or **`draw-text`**) below will be affected by the scaling transformation. First we draw a line and text outside the **`with-scaling`**, then inside:

```
(draw-line* *test-pane 100 50 200 75)
(draw-text* *test-pane* "some text" (floor (+ 100 200) 2) (floor (+ 50 75) 2))

(with-scaling (*test-pane* 2 1)
 (draw-line* *test-pane 100 50 200 75)
 (draw-text* *test-pane* "some text" (floor (+ 100 200) 2) (floor (+ 50 75) 2)))
```

The following figure shows the results of both operations. The line and text on the left are unscaled. Those on the right are scaled. Note that the size of the text is unaffected by scaling.



4.1.1 Components of CLIM Mediums

Each CLIM medium contains components that correspond to the drawing options. These components provide the default values for the drawing options. When drawing functions are called and some options are unspecified, the options default to the values maintained by the medium.

CLIM provides accessors which enable you to read and write the values of these components. Also, these components are temporarily bound within a dynamic context by using **`with-drawing-options`**, **`with-text-style`**, and related forms. Using **`setf`** of a component while it is temporarily bound takes effect immediately but is undone when the dynamic context is exited.

`medium-foreground` [Generic function]

Arguments: *`medium`*

■ Returns the current foreground design of the *`medium`*. You can use **`setf`** on **`medium-foreground`** to change the foreground design. You must not set the foreground ink to an indirect ink.

`medium-background` [Generic function]

Arguments: *`medium`*

■ Returns the current background design of the *`medium`*. You can use **`setf`** on **`medium-background`** to change the background design. You must not set the background ink to an indirect ink.

`medium-ink` [Generic function]

Arguments: *`medium`*

■ Returns the current drawing ink of the *`medium`*. You can use **`setf`** on **`medium-ink`** to change the current ink, or you can use the `:ink` drawing option to any of the drawing functions or to **`with-drawing-options`**.

`medium-transformation` [Generic function]

Arguments: *`medium`*

■ Returns the current user transformation of the *`medium`*. You can use **`setf`** on **`medium-transformation`** to change the current transformation, or you can use the `:transformation` drawing option to any of the drawing functions or to **`with-drawing-options`**.

■ When CLIM's actually draws at the device level, the user transformation is composed with the sheet's device transformation.

`medium-clipping-region` [Generic function]

Arguments: *`medium`*

■ Returns the current user clipping region of the *`medium`*. You can use **`setf`** on **`medium-clipping-region`** to change the clipping region, or you can use the `:clipping-region` drawing option to any of the drawing functions or to **`with-drawing-options`**.

■ When CLIM's actually draws at the device level, the user clipping region is intersected with the sheet's device region.

■ In the current implementation of CLIM, the clipping region must be either a bounding rectangle or a rectangle set of bounding rectangles.

`medium-line-style` [Generic function]

Arguments: *medium*

■ Returns the current line style of the *medium*. You can use **`setf`** on **`medium-line-style`** to change the line style, or you can use the `:line-style` drawing option to any of the drawing functions or to **`with-drawing-options`**.

`medium-text-style` [Generic function]

Arguments: *medium*

■ Returns the current text style of the *medium*. You can use **`setf`** on **`medium-text-style`** to change the current text style, or you can use the `:text-style` drawing option to any of the drawing functions or to **`with-drawing-options`**.

`medium-default-text-style` [Generic function]

Arguments: *medium*

■ The default text style for *medium*. **`medium-default-text-style`** must be a fully specified text style, unlike **`medium-text-style`** which can have null components. Any text styles that are not fully specified by the time they are used for rendering are merged against **`medium-default-text-style`** using **`merge-text-styles`**.

■ You can use **`setf`** on **`medium-default-text-style`** to change the default text style, but the default text style must be a fully specified text style.

4.2 Using CLIM drawing options

Drawing options control various aspects of the drawing process. You can supply drawing options in a number of ways:

- The medium (the destination for graphic output) itself has default drawing options. If a drawing option is not supplied elsewhere, the medium supplies the value.
- You can use **`with-drawing-options`** and **`with-text-style`** to temporarily bind the drawing options of the medium. In many cases, it is more convenient and more efficient to use **`with-drawing-options`** to surround several calls to drawing functions, each using the same options.
- You can supply the drawing options as keyword arguments to the drawing functions. These override the drawing options specified by **`with-drawing-options`**.

In some cases, it is important to distinguish between drawing *options* and drawing *suboptions*. Both text and lines have an option which controls the complete specification of the text and line style, and there are suboptions which can affect one aspect of the text or line style.

For example, the value of the `:text-style` option is a text style object, which describes a complete text style consisting of family, face, and size. There are also suboptions called `:text-family`, `:text-face`, and `:text-size`. Each suboption specifies a single aspect of the text style, while the option specifies the entire text style. Line styles are analogous to text styles; there is a `:line-style` option and some suboptions, such as `:line-thickness` and `:line-dashes`.

In a given call to **`with-drawing-options`** or a drawing function, normally you supply either the `:text-style` option or a text style suboption (or more than one suboption), but you would not supply both. If you do supply both, then the text style comes from the result of merging the suboptions with the `:text-style` option, and then merging that with the prevailing text style.

with-drawing-options **[Macro]**

Arguments: (*medium* &key ink clipping-region transformation line-style line-unit line-thickness line-dashes line-joint-shape line-cap-shape text-style text-family text-face text-size) &body *body*

■ Binds the state of *medium* to correspond to the supplied drawing *options*, and executes the *body* with the new drawing options in effect. Each option causes binding of the corresponding component of the medium for the dynamic extent of the body.

■ *medium* can be either a window stream or a Postscript stream.

■ Any call to a drawing function can supply a drawing option to override the prevailing one. In other words, the drawing functions effectively do a **with-drawing-options** when drawing option arguments are supplied to them.

■ The default value specified for a drawing option is the value to which the corresponding component of a medium is normally initialized.

invoke-with-drawing-options **[Generic function]**

Arguments: *stream function* &key ink clipping-region transformation line-style line-unit line-thickness line-dashes line-joint-shape line-cap-shape text-style text-family text-face text-size

■ This is the functional version of **with-drawing-options**. All of the arguments, except *function*, are the same as for **with-drawing-options**.

function is a function of one argument, a stream; it is called once all of the drawing options are in effect on *stream*.

4.2.1 Set of CLIM drawing options

The drawing options can be any of the following, plus any of the suboptions for line styles and text styles.

:clipping-region **[Drawing option]**

■ Specifies the region of the drawing plane on which the drawing functions can draw.

The clipping region must be an area and furthermore an error might be signaled if the clipping region is not a rectangle or a region-set composed of rectangles. Rendering is clipped both by this clipping region and by other clipping regions associated with the mapping from the target drawing plane to the viewport that displays a portion of the drawing plane. The default is +everywhere+, which means that no clipping occurs in the drawing plane, only in the viewport.

The :clipping-region drawing option temporarily changes the value of **medium-clipping-region** to **region-intersection** of the argument and the previous value. If both a clipping region and a transformation are supplied in the same set of drawing options, the clipping region is transformed by the newly composed transformation.

:ink **[Drawing option]**

■ A design used as the ink for drawing operations. The drawing functions draw with the color and pattern specified by the :ink option. The default value is +foreground-ink+.

The :ink drawing option temporarily changes the value of **medium-ink** and replaces the previous ink; the new and old inks are not combined in any way.

The value of :ink can be:

- a color
- a dynamic color
- a layered color
- the constant `+foreground-color+`
- the constant `+background-ink+`
- a flipping ink
- an opacity or the constant `+transparent-ink+` (opacities are not fully supported)
- a design

For more information on how to use the `:ink` drawing option, see chapter 6 **Drawing in color in CLIM**.

`:transformation` **[Drawing option]**

■ Transforms the coordinates used as arguments to drawing functions to the coordinate system of the drawing plane. The default value is `+identity-transformation+`.

■ The `:transformation` drawing option temporarily changes the value of **`medium-transformation`** to **`compose-transformations`** of the argument and the previous value.

`:text-style` **[Drawing option]**

■ Controls how text is rendered, for the graphic drawing functions and ordinary stream output. The value of the `:text-style` option is a text style object.

This drawing option temporarily changes the value of **`medium-text-style`** to the result of merging the value of `:text-style` with the prevailing text style.

If text style suboptions are also supplied, they temporarily change the value of **`medium-text-style`** to the result of merging the supplied suboptions with the `:text-style` drawing options, which is then merged with the previous value of **`medium-text-style`**.

■ See the section 5.3 **CLIM text style suboptions**.

`:line-style` **[Drawing option]**

■ Controls how lines and arcs are rendered. The value of the `:line-style` option is a line style object.

This drawing option temporarily changes the value of **`medium-line-style`**.

■ See the section 4.3.2 **CLIM line style suboptions**.

The following macro causes all of the output of the body to be done with fat lines that butt with a rounded joint.

```
(defmacro with-fat-lines ((stream &optional (thickness 5)) &body body)
  `(clim:with-drawing-options (,stream :line-thickness ,thickness
                                       :line-joint-shape :round)
     ,@body))
```

4.2.2 Using the :filled option to certain CLIM drawing functions

Certain drawing functions can draw either an area or the outline of that area. This is controlled by the `:filled` keyword argument to these functions. If the value is `t` (the default), then the function paints the entire area. If the value is `nil`, then the function strokes the outline of the area under the control of the line-style drawing option.

The `:filled` keyword argument is *not* a drawing option and cannot be supplied to **`with-drawing-options`**.

These are functions that have a `:filled` keyword argument:

`draw-circle`
`draw-circle*`
`draw-ellipse`
`draw-ellipse*`
`draw-polygon`
`draw-polygon*`
`draw-rectangle`
`draw-rectangle*`
`draw-rectangles`
`draw-rectangles*`

4.3 CLIM line styles

A line or other path (such as an unfilled ellipse or polygon) is a one-dimensional object. In order to be visible, the rendering of a line must, however, occupy some non-zero area on the display hardware. A *line style* object is used to represent the advice that CLIM supplies to the rendering substrate on how to perform the rendering. See the section 4.3.2 **CLIM line style suboptions** for detailed descriptions of the material below.

4.3.1 CLIM line style objects

It is often useful to create a line style object that represents a style you wish to use frequently, rather than continually supplying the corresponding line style suboptions.

The class of a line style object is `line-style`. You create a line style object with **`make-line-style`**. Line styles are immutable -- you cannot modify one once it has been created.

`make-line-style` **[Function]**

Arguments: `&key (unit :normal) (thickness 1) dashes (joint-shape :miter) (cap-shape :butt)`

- Creates a line style object with the supplied characteristics.

  Note that these line style keywords correspond to the CLIM line style suboptions that begin with `:line-` (for example, `:unit` corresponds to the line style suboption `:line-unit`).

  Since **`make-line-style`** conses a new object when it is called, you may want to create only a single instance of a line style and store it somewhere.

The following readers are provided for the components of line styles (remember line style objects cannot be modified so there are no setters):

`line-style-thickness` [Generic function]

Arguments: *line-style*

- Returns the thickness component of a line style object, which is an integer.

`line-style-joint-shape` [Generic function]

Arguments: *line-style*

- Returns the joint shape component of a line style object. This will be either `:miter`, `:bevel`, `:round`, or `:none`.

`line-style-cap-shape` [Generic function]

Arguments: *line-style*

- Returns the cap shape component of a line style object. This will be either `:butt`, `:square`, `:round`, or `:no-end-point`.

`line-style-dashes` [Generic function]

Arguments: *line-style*

- Returns the dashes component of a line style object. This will be `nil` to indicate a solid line, `t` to indicate a dashed line whose dash pattern is unspecified, or will be a sequence specifying some sort of a dash pattern.

`line-style-unit` [Generic function]

Arguments: *standard-line-style*

- Returns the units in which the line will be drawn, one of `:normal` or `:point`.

4.3.2 CLIM line style suboptions

Each line style suboption has a reader function which returns the value of that component from a line style object.

The line style suboptions are:

`:line-unit` **[Drawing option]**

- The units in which the thickness, dash pattern, and dash phase are measured. Possible values are `:normal` and `:point`, which have the following meanings:

  `:normal`

  A relative measure in terms of the usual or normal line thickness. The normal line thickness is the thickness of the comfortably visible thin line, which is a property of the underlying rendering substrate. This is the default.

  `:point`

  An absolute measure in terms of printer's points (approximately 1/72 of an inch).

- You can call **`line-style-unit`** on a line style object to get the value of the `:line-unit` or `:unit` component.

`:line-thickness` **[Drawing option]**

■ The thickness (an integer in the units described by **`line-style-unit`**) of the lines or arcs drawn by a drawing function. The default is 1, which combined with the default unit of `:normal`, means that the default line drawn is the comfortably visible thin line.

You can call **`line-style-thickness`** on a line style object to get the value of the `:line-thickness`, or `:thickness` component.

`:line-dashes` **[Drawing option]**

■ Controls whether lines or arcs are drawn as dashed figures, and if so, what the dashing pattern is. Possible values are:

`nil`

Lines are drawn solid, with no dashing. This is the default.

`t`

Lines are drawn dashed, with a dash pattern that is unspecified and may vary with the rendering substrate. This allows the underlying display substrate to provide a default dashed line for the user whose only requirement is to draw a line that is visually distinguished from the default solid line. Using the default dashed line can be more efficient than specifying customized dashes.

`sequence`

Specifies a sequence of integers, usually a vector, controlling the dash pattern of a drawing function. It is an error if the sequence does not contain an even number of elements. The elements of the sequence are lengths of individual components of the dashed line or arc. The odd elements specify the length of inked components, the even elements specify the gaps. All lengths are expressed in the units described by **`line-style-unit`**. You can use **`make-contrasting-dash-patterns`** to create a a sequence for the `:dashes` option.

See the function **`make-contrasting-dash-patterns`**.

■ You can call **`line-style-dashes`** on a line style object to get the value of the `:line-dashes`, or `:dashes` component.

`:line-joint-shape` **[Drawing option]**

■ Specifies the shape of joints between line segments of closed, unfilled figures, when the `:line-thickness` or `:thickness` option to a drawing function is greater than 1. The possible shapes are `:miter`, `:bevel`, `:round`, and `:none`; the default is `:miter`.

Note that the joint shape is implemented by the host window system, so not all platforms will necessarily support it equally.

■ You can call **`line-style-joint-shape`** on a line style object to get the value of the `:line-joint-shape`, or `:joint-shape` component.

`:line-cap-shape` **[Drawing option]**

■ Specifies the shape for the ends of lines and arcs drawn by a drawing function, one of `:butt`, `:square`, `:round`, or `:no-end-point`. The default is `:butt`. Note that the cap shape is implemented by the host window system, so not all platforms will necessarily support it equally.

■ You can call **`line-style-cap-shape`** on a line style object to get the value of the `:line-cap-shape`, or `:cap-shape component`.

This function can be used to generate a value for the `:dashes` line suboption.

make-contrasting-dash-patterns [Function]

Arguments: *n* &optional *k*

■ Makes a vector of *n* dash patterns with recognizably different appearances. If *k* (an integer between 0 and *n*-1) is supplied, **make-contrasting-dash-patterns** returns the *k*'th dash pattern. If *n* is greater than the value returned by **contrasting-dash-patterns-limit** (defined next), **make-contrasting-dash-patterns** signals an error.

contrasting-dash-patterns-limit [Function]

Arguments: *port*

■ Returns the number of contrasting dash patterns that the port *port* can generate. In Allegro CLIM, the number is currently 16, but this could change.

4.4 Transformations in CLIM

One of the features of CLIM's graphical capabilities is the use of coordinate system transformations. By using transformations you can often write simpler graphics code, because you can choose a coordinate system in which to express the graphics that simplifies the description of the drawing.

A transformation is an object that describes how one coordinate system is related to another. A graphic function performs its drawing in the current coordinate system of the stream. A new coordinate system is defined by describing its relationship to the old one (the transformation). The drawing can now take place in the new coordinate system. The basic concept of graphic transformations is illustrated in Figure 4.1.

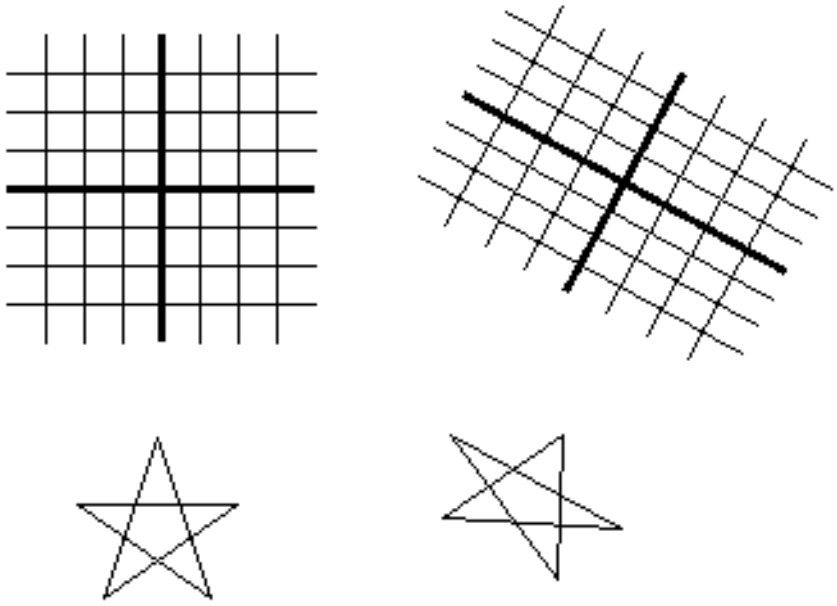


Figure 4.1. Graphic Transformation

For example, you might define the coordinates of a five-pointed star, and a function to draw it.

```
(defvar *star* '(0 3 2 -3 -3 1/2 3 1/2 -2 -3))
(defun draw-star (stream)
  (clim:draw-polygon* stream *star* :closed t :filled nil))
```

Without any transformation, the function draws a small star centered around the origin. By applying a transformation, the same function can be used to draw a star of any size, anywhere. For example, the following code will draw a picture somewhat like the lower half of Figure 4.1 on *stream*.:

```
(clim:with-room-for-graphics (stream)
 (clim:with-translation (stream 100 100)
  (clim:with-scaling (stream 10)
   (draw-star stream)))
 (clim:with-translation (stream 240 110)
  (clim:with-rotation (stream -0.5)
   (clim:with-scaling (stream 12 8)
    (draw-star stream)))))
```

4.4.1 The transformations used by CLIM

The type of transformations that CLIM uses are called *affine transformations*. An affine transformation is a transformation that preserves straight lines. In other words, if you take a number of points that fall on a straight line and apply an affine transformation to their coordinates, the transformed coordinates will fall on a straight line in the new coordinate system. Affine transformations include translations, scalings, rotations, and reflections.

- A *translation* is a transformation that preserves length, angle, and orientation of all geometric entities.
- A *rotation* is a transformation that preserves length and angles of all geometric entities. Rotations also preserve one point and the distance of all entities from that point. You can think of that point as the center of rotation, it is the point around which everything rotates.
- There is no single definition of a *scaling transformation*. Transformations that preserve all angles and multiply all lengths by the same factor (preserving the shape of all entities) are certainly scaling transformations. However, scaling is also used to refer to transformations that scale distances in the x direction by one amount and distances in the y direction by another amount.
- A *reflection* is a transformation that preserves lengths and magnitudes of angles, but changes the sign (or handedness) of angles. If you think of the drawing plane on a transparent sheet of paper, a reflection is a transformation that turns the paper over.

If we transform from one coordinate system to another, then from the second to a third coordinate system, we can regard the resulting transformation as a single transformation resulting from *composing* the two component transformations. It is an important and useful property of affine transformations that they are closed under composition.

Note that composition is not commutative; in general, the result of applying transformation A and then applying transformation B is not the same as applying B first, then A.

Any arbitrary transformation can be built up by composing a number of simpler transformations, but that same transformation can often be constructed by a different composition of different transformations.

Transforming a region applies a coordinate transformation to that region, thus moving its position on the drawing plane, rotating it, or scaling it. Note that this creates a new region, it does not side-effect the *region* argument.

The user interface to transformations is the `:transformation` option to the drawing functions. Users can create transformations with constructors; see the section 4.4.2 **CLIM transformation constructors**. The other operators documented in this section are used by CLIM itself, and are not often needed by users.

4.4.2 CLIM transformation constructors

The following functions can be used to create a transformation object that can be used, for instance, in a call to **`compose-transformations`**.

`make-translation-transformation` [Function]

Arguments: *`delta-x delta-y`*

- Makes a transformation that translates all points by *`delta-x`* in the x-direction and *`delta-y`* in the y-direction.

`make-rotation-transformation` [Function]

Arguments: *`angle`* `&optional` *`origin`*

- Makes a transformation that rotates all points by *`angle`* around the point *`origin`*. The angle is specified in radians. If *`origin`* is supplied it must be a point; if not supplied it defaults to (0,0).

`make-rotation-transformation*` [Function]

Arguments: *`angle origin-x origin-y`*

- Makes a transformation that rotates all points by *`angle`* around the point, (*`origin-x`*, origin-y). The angle is specified in radians.
- The following transformation rotates the coordinate system around the point (10,10):

```
(make-rotation-transformation* (/ pi 8) 10 10)
```

`make-scaling-transformation` [Function]

Arguments: *`mx my`* `&optional` *`origin`*

- Makes a transformation that multiplies the x-coordinate distance of every point from *`origin`* by *`mx`* and the y-coordinate distance of every point from *`origin`* by *`my`*. If *`origin`* is supplied it must be a point; if not supplied it defaults to (0,0).

`make-scaling-transformation*` [Function]

Arguments: *`mx my origin-x origin-y`*

- Makes a transformation that multiplies the x-coordinate distance of every point from *`origin-x`* by *`mx`* and the y-coordinate distance of every point from *`origin-y`* by *`my`*.
- For example, using the following transformation on a stream would cause all output on that stream to be only half as big:

```
(make-scaling-transformation* 1/2 1/2)
```

`make-reflection-transformation` [Function]

Arguments: *`point-1 point-2`*

- Makes a transformation that reflects every point through the line passing through the points *`point-1`* and *`point-2`*.

`make-reflection-transformation*` [Function]

Arguments: *`x1 y1 x2 y2`*

- Makes a transformation that reflects every point through the line passing through the points (*`x1`*, *`y1`*) and (*`x2`*, *`y2`*).

`make-transformation` [Function]

Arguments: *mxx mxy myx myy tx ty*

■ Makes a general transformation whose effect is,

$x' = m_{xx}x+m_{xy}y+t_x$

$y' = m_{yx}x+m_{yy}y+t_y$

where x and y are the coordinates of a point before the transformation and x' and y' are the coordinates of the corresponding point after.

`make-3-point-transformation` [Function]

Arguments: *point-1 point-2 point 3point-1-image point-2-image point-3-image*

■ Makes a transformation that takes *point-1* into *point-1-image*, *point-2* into *point-2-image* and *point-3* into *point-3-image*. (Three non-collinear points and their images under the transformation are enough to specify any affine transformation.)

■ It is an error for *point-1*, *point-2*, and *point-3* to be collinear; if they are collinear, the `transformation-underspecified` condition is signaled. If *point-1-image*, *point-2-image*, and *point-3-image* are collinear, the resulting transformation will be singular but this is not an error.

`make-3-point-transformation*` [Function]

Arguments: *x1 y1 x2 y2 x3 y3 x1-image y1-image x2-image y2-image x3-image y3-image*

■ Makes a transformation that takes (*x1*, *y1*) into (*x1-image*, *y1-image*), (*x2*, *y2*) into (*x2-image*, *y2-image*) and (*x3*, *y3*) into (*x3-image*, *y3-image*). (Three non-collinear points and their images under the transformation are enough to specify any affine transformation.)

■ It is an error for (*x1*, *y1*), (*x2*, *y2*) and (*x3*, *y3*) to be collinear; if they are collinear, the `transformation-underspecified` condition is signaled. If (*x1-image*, *y1-image*), (*x2-image*, *y2-image*), and (*x3-image*, *y3-image*) are collinear, the resulting transformation will be singular but this is not an error.

4.4.3 Operations on CLIM transformations

This section describes the various operations you can perform on CLIM transformations. Most of the operations are predicates that you can use to figure out what properties a transformation has.

`transformation` [Class]

■ The protocol class for all transformations. There are one or more subclasses of `transformation` with implementation-dependent names that implement transformations. If you want to create a new class that obeys the transformation protocol, it must be a subclass of `transformation`.

`+identity-transformation+` [Constant]

■ An instance of a transformation that is guaranteed to be an identity transformation, that is, the transformation that does nothing.

The following predicates are provided in order to be able to determine whether or not a transformation has a particular characteristic.

`transformation-equal` [Generic function]

Arguments: *transform1 transform2*

■ Returns `t` if the two transformations have equivalent effects (that is, are mathematically equal), otherwise returns `nil`.

`identity-transformation-p` [Generic function]

Arguments: *transform*

■ Returns `t` if *transform* is equal (in the sense of **`transformation-equal`**) to the identity transformation, otherwise returns `nil`.

`translation-transformation-p` [Generic function]

Arguments: *transform*

■ Returns `t` if *transform* is a pure translation, that is a transformation that moves every point by the same distance in x and the same distance in y, otherwise returns `nil`.

`invertible-transformation-p` [Generic function]

Arguments: *transform*

■ Returns `t` if *transform* has an inverse, otherwise returns `nil`.

`reflection-transformation-p` [Generic function]

Arguments: *transform*

■ Returns `t` if *transform* inverts the handedness of the coordinate system, otherwise returns `nil`. Note that this is a very inclusive category. Transformations are considered reflections even if they distort, scale, or skew the coordinate system, as long as they invert the handedness.

`rigid-transformation-p` [Generic function]

Arguments: *transform*

■ Returns `t` if *transform* transforms the coordinate system as a rigid object, that is, as a combination of translations, rotations, and pure reflections. Otherwise, it returns `nil`.

Rigid transformations are the most general category of transformations that preserve magnitudes of all lengths and angles.

`even-scaling-transformation-p` [Generic function]

Arguments: *transform*

■ Returns `t` if *transform* multiplies all x-lengths and y-lengths by the same magnitude, otherwise returns `nil`. This includes pure reflections through vertical and horizontal lines.

`scaling-transformation-p` [Generic function]

Arguments: *transform*

■ Returns `t` if *transform* multiplies all x-lengths by one magnitude and all y-lengths by another magnitude, otherwise returns `nil`. This category includes even scalings as a subset.

`rectilinear-transformation-p` [Generic function]

Arguments: *transform*

■ Returns `t` if *transform* will always transform any axis-aligned rectangle into another axis-aligned rectangle, otherwise returns `nil`. This category includes scalings as a subset, and also includes 90 degree rotations.

Rectilinear transformations are the most general category of transformations for which the bounding rectangle of a transformed object can be found by transforming the bounding rectangle of the original object.

4.4.4 Composition of CLIM transformations

Composing one transformation with another is the way to create a new transformation that has the same effect as applying both of the others.

The most general function is **compose-transformations**, but the following six functions are special-cases of **compose-transformations** that are more efficient:

compose-translation-with-transformation

compose-rotation-with-transformation

compose-scaling-with-transformation

compose-transformation-with-translation

compose-transformation-with-rotation

compose-transformation-with-scaling

compose-transformations [Generic function]

Arguments: *transform1 transform2*

■ Returns a transformation that is the composition of its arguments. Composition is in right-to-left order, that is, the resulting transformation represents the effects of applying *transform2* followed by *transform1*. This is consistent with the order in which **with-translation**, **with-rotation**, and **with-scaling** compose.

For example, the following two forms result in the same transformation, presuming that the stream's transformation is the identity transformation:

```
(clim:compose-transformations
 (clim:make-translation-transformation dx dy)
 (clim:make-rotation-transformation angle))

(clim:with-translation (stream dx dy)
 (clim:with-rotation (stream angle)
  (clim:medium-transformation stream)))
```

■ Note that any arbitrary transformation can be built up by composing a number of simpler transformations, but that composition is not unique.

compose-translation-with-transformation [Generic function]

Arguments: *transform dx dy*

■ Creates a new transformation by composing *transform* with a given translation, as specified by *dx* and *dy*. The order of composition is that the translation transformation is first, followed by *transform*.

■ This function be been implemented as follows:

```
(defun compose-translation-with-transformation (transform dx dy)
 (clim:compose-transformations
  transform
```

```
    (clim:make-translation-transformation dx dy)))
```

compose-scaling-with-transformation [Generic function]

Arguments: *transform mx my* &optional *origin*

- Creates a new transformation by composing *transform* with a given scaling, as specified by *mx*, *my*, and *origin*. If *origin* is supplied it must be a point; if not supplied it defaults to (0,0). The order of composition is that the scaling transformation is first, followed by *transform*.
- This function could be implemented as follows:

```
(defun compose-scaling-with-transformation
    (transform mx my &optional origin)
 (clim:compose-transformations
   transform
   (clim:make-scaling-transformation mx my origin)))
```

compose-rotation-with-transformation [Generic function]

Arguments: *transform angle* &optional *origin*

- Creates a new transformation by composing *transform* with a given rotation, as specified by *angle* and *origin*. *angle* is in radians. If *origin* is supplied it must be a point; if not supplied it defaults to (0,0). The order of composition is that the rotation transformation is first, followed by *transform*.
- This function could be implemented as follows:

```
(defun compose-rotation-with-transformation
    (transform angle &optional origin)
 (clim:compose-transformations
   transform
   (clim:make-rotation-transformation angle origin)))
```

compose-transformation-with-translation [Generic function]

Arguments: *transform dx dy*

- Creates a new transformation by composing the given translation, with the *transformation*. The order of composition is *transformation* is first, followed by the translation transformation. *dx* and *dy* are as for **make-translation-transformation**.

compose-transformation-with-scaling [Generic function]

Arguments: *transform mx my &optional origin*

- Creates a new transformation by composing the given scaling, with the *transformation*. The order of composition is *transformation* is first, followed by the scaling transformation. *mx*, *my*, and *origin* are as for **make-scaling-transformation**.

compose-transformation-with-rotation [Generic function]

Arguments: *transform angle &optional origin*

- Creates a new transformation by composing the given rotation, with the *transformation*. The order of composition is *transformation* is first, followed by the rotation transformation. *angle* and *origin* are as for **make-rotation-transformation**.

`invert-transformation` [Generic function]

Arguments: *transform*

- Returns a transformation that is the inverse of *transform*. The result of composing a transformation with its inverse is the identity transformation.
- If *transform* is singular, **invert-transformation** signals the `singular-transformation` condition, with a named restart that is invoked with a transformation and makes **invert-transformation** return that transformation. This is to allow a drawing application, for example, to use a generalized inverse to transform a region through a singular transformation.
- Note that finite-precision arithmetic there are several low-level conditions which might occur during the attempt to invert a singular or almost singular transformation. (These include computation of a zero determinant, floating-point underflow during computation of the determinant, or floating-point overflow during subsequent multiplication.) **invert-transformation** must signal the `singular-transformation` condition for all of these cases.

Any arbitrary transformation can be built up by composing a number of simpler transformations, but that composition is not unique.

The following three forms can be used to compose a transformation into the current transformation of a stream. They are intended as abbreviations for calling **compose-transformations** and **with-drawing-options** directly.

`with-rotation` [Macro]

Arguments: *(medium angle* &optional *origin)* &body *body*

- Establishes a rotation on medium that rotates by *angle* (in radians), and then executes *body* with that transformation in effect. If *origin* is supplied, the rotation is about that point. The default for origin is (0,0).
- This is equivalent to using **with-drawing-options** with the `:transformation` keyword argument supplied:

```
(clim:with-drawing-options
    (medium
     :transformation (clim:make-rotation-transformation
                      angle origin))
  body)
```

`with-translation` [Macro]

Arguments: *(medium dx dy)* &body *body*

- Establishes a scaling transformation on *medium* that scales by *dx* in the x direction and *dy* in the y direction, and then executes *body* with that transformation in effect.
- This is equivalent to using **with-drawing-options** with the `:transformation` keyword argument supplied:

```
(clim:with-drawing-options
    (medium
     :transformation (clim:make-trnslation-transformation
                      dx dy))
  body)
```

with-scaling **[Macro]**

Arguments: *(medium sx* &optional *sy)* &body *body*

■ Establishes a scaling transformation on *medium* that scales by *sx* in the x direction and *sy* in the y direction, and then executes *body* with that transformation in effect. If *sy* is not supplied, it defaults to *sx*.

■ This is equivalent to using **with-drawing-options** with the :transformation keyword argument supplied:

```
(clim:with-drawing-options
    (medium
     :transformation (clim:make-scaling-transformation
                      sx sy))
  body)
```

The following three macros also compose a transformation into the current transformation of a stream, but have more complex behavior.

with-room-for-graphics **[Macro]**

Arguments: (&optional *stream* &key height (first-quadrant t) (move-cursor t) record-type) &body *body*

■ Binds the dynamic environment to establish a local Cartesian coordinate system for doing graphics output onto stream. If first-quadrant is t (the default), a local Cartesian coordinate system is established with the origin (0,0) of the local coordinate system placed at the current cursor position; (0,0) is in the lower left corner of the area created. If the boolean *move-cursor* is t (the default), then after the graphic output is completed, the cursor is positioned past (immediately below) this origin. The bottom of the vertical block allocated is at this location (that is, just below point (0,0), not necessarily at the bottom of the output done).

If *height* is supplied, it should be a number that specifies the amount of vertical space to allocate for the output, in device units. If it is not supplied, the height is computed from the output.

record-type specifies the class of output record to create to hold the graphical output. The default is standard-sequence-output-record.

■ The following rather complicated example draws the points of a compass in a menu, and allow the user to choose one of the compass points. **with-room-for-graphics** is used to put the entire menu in a coordinate space whose upper-left corner is (0,0). because it depends on the correct context being present, this function must be run in a CLIM Lisp Listener (such as displayed by the Lisp Listener demo). Calling this function from Lisp top-level in, say, an Emacs *common-lisp* buffer signals an error because the correct context is not present.

```
(defun choose-compass-point (stream)
 (labels ((draw-compass-point (stream type symbol x y)
       (clim:with-output-as-presentation (stream symbol type)
        (clim:draw-text* stream (symbol-name symbol) x y
                :align-x :center :align-y :center
                :text-style '(:sans-serif :roman :large))))
      (draw-compass (stream type)
       (clim:with-room-for-graphics (stream :first-quadrant nil)
        (clim:draw-line* stream 0 25 0 -25 :line-thickness 2)
        (clim:draw-line* stream 25 0 -25 0 :line-thickness 2)
        (dolist (point '((n 0 -30) (s 0 30) (e 30 0) (w -30 0)))
         (apply #'draw-compass-point stream type point)))))
```

```
(clim:with-menu (menu stream :scroll-bars nil)
 (clim:menu-choose-from-drawer menu 'clim:menu-item #'draw-compass))))
```

with-local-coordinates **[Macro]**

Arguments: (&optional *stream x y*) &body *body*

■ Binds the dynamic environment to establish a local coordinate system with the positive X-axis extending to the right and the positive Y-axis extending downward, with (0,0) at (*x*,*y*). If *x* and *y* are not specified (or even if *x* is and *y* is not), the current cursor position of *stream* is used as (0,0).

with-first-quadrant-coordinates **[Macro]**

Arguments: (&optional *stream x y*) &body *body*

■ Binds the dynamic environment to establish a local coordinate system with the positive X-axis extending to the right and the positive Y-axis extending upward, with (0,0) at (*x*,*y*). If *x* and *y* are not specified (or even if *x* is and *y* is not), the current cursor position of *stream* is used as (0,0).

Here is an example using with-local-coordinates and with-first-quadrant-coordinates. We take `*test-pane*` and set the cursor position to (100, 50):

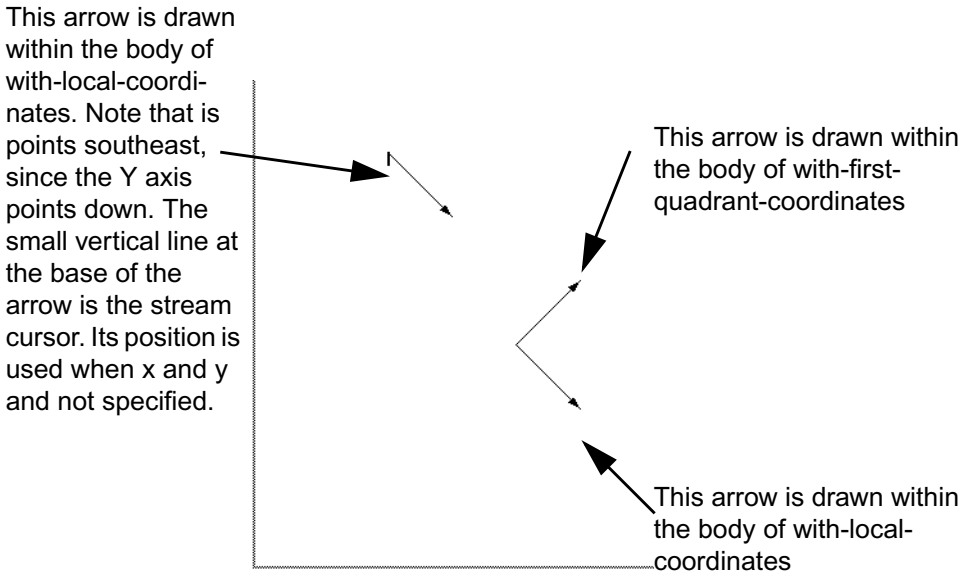
```
(stream-set-cursor-position *test-pane* 100 50)
```

Then we draw and arrow to the point (50,50) using with-local-coordinates without specifying *x* and *y*.

```
(with-local-coordinates (*test-pane*) (draw-arrow* *test-pane* 0 0 50 50)
```

Next we use both macros to draw arrows from the (local) origin to (50,50). Note that the arrow points down to the right (to the southeast) when drawn within the body of **with-local-coordinates** and up to the right (to the northeast) when drawn within the body of **with-first-quadrant-coordinates**. Note further that the cursor location is not modified by these macros.

```
(with-local-coordinates (*test-pane* 200 200) (draw-arrow* 0 0 50 50))
(with-first-quadrant-oordinaes (*test-pane 200 200) (draw-arrow* 0 0 50 50))
```



4.4.5 Applying CLIM transformations

The following functions can be used to apply a transformation to some sort of a geometric object, such as a region or a distance. Calling **`transform-position`** or **`untransform-position`** on a spread point is generally more efficient than calling **`transform-region`** or **`untransform-region`** on the unspread point object.

`transform-region` [Generic function]

Arguments: *`transformation region`*

■ Applies *`transformation`* to *`region`*, and returns a new transformed region.

Transforming a region applies a coordinate transformation to that region, thus moving its position on the drawing plane, rotating it, or scaling it. Note that this creates a new region, it does not side-effect the *`region`* argument.

`untransform-region` [Generic function]

Arguments: *`transformation region`*

■ Applies the inverse of *`transformation`* to *`region`* and returns a new transformed region. This is equivalent to:

```
(clim:transform-region
 (clim:invert-transformation transform) region)
```

`transform-position` [Generic function]

Arguments: *`transform x y`*

■ Applies *`transform`* to the point whose coordinates are *`x`* and *`y`*, and returns two values, the transformed x-coordinate and the transformed y-coordinate.

`transform-position` is the spread version of **`transform-region`** in the case where the region is a point.

`untransform-position` [Generic function]

Arguments: *`transform x y`*

■ Applies the inverse of *`transform`* to the point whose coordinates are *`x`* and *`y`*, and returns two values, the transformed x-coordinate and the transformed y-coordinate.

`transform-position` is the spread version of **`transform-region`** in the case where the region is a point.

`transform-distance` [Generic function]

Arguments: *`transform dx dy`*

■ Applies *`transform`* to the distance represented by *`dx`* and *`dy`*, and returns two values, the transformed *`dx`* and the transformed *`dy`*. A distance represents the difference between two points. It does *not* transform like a point.

`untransform-distance` [Generic function]

Arguments: *`transform dx dy`*

■ Applies the inverse of *`transform`* to the distance represented by *`dx`* and *`dy`*, and returns two values, the transformed *`dx`* and the transformed *`dy`*. A distance represents the difference between two points. It does *not* transform like a point.

transform-rectangle* [Generic function]

Arguments: *transform x1 y1 x2 y2*

- Applies the transformation *transform* to the rectangle specified by the four coordinate arguments, which are real numbers. The arguments *x1*, *y1*, *x2*, and *y1* are canonicalized in the same way as for **make-bounding-rectangle**. Returns four values that specify the minimum and maximum points of the transformed rectangle in the order *min-x*, *min-y*, *max-x*, and *max-y*.
- It is an error is *transform* does not satisfy **rectilinear-transformation-p**.
- **transform-rectangle*** is the spread version of **transform-region** in the case where the transformation is rectilinear and the region is a rectangle.

untransform-rectangle* [Generic function]

Arguments: *transform x1 y1 x2 y2*

- This is exactly equivalent to calling **transform-rectangle*** on the inverse of *transform*.

Chapter 5 Text styles in CLIM

5.1 Concepts of CLIM text styles

CLIM's model for the appearance of text follows the same principle as the model for creating formatted output. This principle holds that the application program should describe how the text should appear in high-level terms, and that CLIM will take care of the details of choosing a specific device font. This approach emphasizes portability.

You specify the appearance of text by giving it an abstract *text style*. Each CLIM medium defines a mapping between these abstract style specifications and particular device-specific fonts. At runtime, CLIM chooses an appropriate device font to represent the characters.

A *text style* is a combination of three characteristics that describe how characters appear. Text style objects have components for *family*, *face*, and *size*.

family

Characters of the same family have a typographic integrity, so that all characters of the same family resemble one another. CLIM supports the families `:fix`, `:serif`, `:sans-serif`, or `nil`.

face

A modification of the family, such as bold or italic. CLIM supports the faces `:roman` (meaning normal), `:bold`, `:italic`, `(:bold :italic)`, or `nil`.

size

The size of the character. One of the logical sizes (`:tiny`, `:very-small`, `:small`, `:normal`, `:large`, `:very-large`, `:huge`, `:smaller`, `:larger`), or a real number representing the size in printer's points, or `nil`.

Not all of these attributes need be supplied for a given text style object. Text styles can be merged in much the same way as pathnames are merged; unspecified components in the style object (that is, components which have `nil` in them) may be filled in by the components of a default style object.

A text style object is called *fully* specified if none of its components is `nil`, and the size component is not a relative size (that is, is neither `:smaller` nor `:larger`).

`*default-text-style*` **[Variable]**

■ This is the default text style used by all streams. When doing output to a stream, if the text style is not fully specified, it is merged against `*default-text-style*` using **`merge-text-styles`**.

If you change the value of `*default-text-style*`, the new value must be a fully specified text style.

Note that the sizes `:smaller` and `:larger` are treated specially in that they are merged with the default text style size to result in a size that is discernibly smaller or larger. For example, a text style size of `:larger` would merge with a default text size of `:small` to produce the resulting size `:normal`.

When text is rendered on a medium, the text style is mapped to some medium specific description of the glyphs for each character. This description is usually that medium's concept of a font object. This mapping is mostly transparent to the application developer, but it is worth noting that not all text styles have mappings associated with them on all mediums. If the text style used does not have a mapping associated with it on the given medium, a special text style reserved for this case will be used.

5.2 CLIM Text Style Objects

It is often useful to create a text style object that represents a style you wish to use frequently, rather than continually specifying the corresponding text style suboptions.

For example, you might want to have a completely different family, face and size for menus. You could make a text style object and make it be the value of `*menu-text-style*`.

You create text style objects using **`make-text-style`**.

```
(clim:with-text-style
  (my-stream (clim:make-text-style :fix :bold :large))
 (write-string "Here is a text-style example." my-stream))
```

Note that text style objects are interned. That is, two different invocations of **`make-text-style`** with the same combination of family, face and size will result in the same (in the sense of **`eq`**) text style object. For this reason, you must not modify text style objects.

5.3 CLIM Text Style Suboptions

You can use text style suboptions to specify characteristics of a text style object. Each text style suboption has a reader function which returns the current value of that component from a text style object.

The text style suboptions are:

`:text-family` **[Text style option]**

- Specifies the family of the text style. The reader function is **`text-style-family`**.

`:text-face` **[Text style option]**

- Specifies the face of the text style. The reader function is **`text-style-face`**.

`:text-size` **[Text style option]**

- Specifies the size of the text style. The reader function is **`text-style-size`**.

5.4 CLIM Text Style Functions

The following functions can be used to parse, merge, and create text style objects, and read the components of the objects.

parse-text-style [Generic function]

Arguments: *text-style*

- *text-style* is either a text style object or a device font, in which case **parse-text-style** returns *text-style*. Otherwise, *text-style* must be a list of three elements, the text style family, face, and size. In this case, *text-style* is parsed and a text style object is returned.

For example, `(clim:parse-text-style '(:fix :bold 12))` might return the object

```
#<STANDARD-TEXT-STYLE :FIX.:BOLD.12 1116707341>
```

merge-text-styles [Generic function]

Arguments: *style1 style2*

- Merges *style1* against the defaults provided by *style2*. That is, any `nil` components in *style1* are filled in from *style2*.

  If the size component of *style1* is a relative size, the resulting size will be the size component of *style2* as modified by the relative size.

  If the face component of *style1* is `:bold` and the face component of *style2* is `:italic` (or vice-versa), the resulting face will be `(:bold :italic)`.

Here are some examples:

```
(clim:merge-text-styles '(:fix :bold 12) '(nil :roman nil))
 → #<STANDARD-TEXT-STYLE :FIX.:BOLD.12 @ #xfac762>
(clim:merge-text-styles '(:fix :bold nil) '(nil :roman 10))
 → #<STANDARD-TEXT-STYLE :FIX.:BOLD.10 @ #xabe634>
(clim:merge-text-styles '(:fix :bold 12) '(nil :italic 10))
 → #<STANDARD-TEXT-STYLE :FIX.(:BOLD :ITALIC).12 @ #xf23be6>
```

text-style-components [Generic function]

Arguments: *text-style medium*

- Returns the components of *text-style* as three values (family, face, and size).

text-style-family [Generic function]

Arguments: *text-style*

- Returns the family component of the *text-style*.

text-style-face [Generic function]

Arguments: *text-style*

- Returns the face component of the *text-style*.

text-style-size [Generic function]

Arguments: *text-style*

- Returns the size component of the *text-style*.

text-style-ascent [Generic function]

Arguments: *text-style medium*

- The ascent (an integer) of *text-style* as it would be rendered on *medium*.

The ascent of a text style is the ascent of the medium's font corresponding to text-style. The ascent of a font is the distance between the top of the tallest character in that font and the baseline.

`text-style-descent` [Generic function]

Arguments: *`text-style medium`*

■ The descent (an integer) of *`text-style`* as it would be rendered on *`medium`*.

The descent of a text style is the descent of the medium's font corresponding to text-style. The descent of a font is the distance between the baseline and the bottom of the lowest descending character (usually "y", "q", "p", or "g").

`text-style-height` [Generic function]

Arguments: *`text-style medium`*

■ Returns the height (an integer) of the usual character in *`text-style`* on *`medium`*. The height of a text style is the sum of its ascent and descent.

`text-style-width` [Generic function]

Arguments: *`text-style medium`*

■ Returns the width (an integer) of the usual character in *`text-style`* on *`medium`*.

`text-style-fixed-width-p` [Generic function]

Arguments: *`text-style medium`*

■ Returns `t` if *`text-style`* will map to a fixed-width font on *`medium`*, otherwise returns `nil`.

`text-style-mapping` [Generic function]

Arguments: *`port style &optional character-set`*

■ Returns the font object that will be used if characters in *`character-set`* in the text style *`style`* are drawn on any medium on the port *`port`*. *`character-set`* defaults to the standard character set. Under Allegro CLIM, the object returned by **`text-style-mapping`** will be an X Windows font object.

■ If the port is using exact text style mapping, CLIM will choose a font whose size exactly matches the size specified in the text style. Otherwise if the port is using loose text style mappings, CLIM will choose the font whose size is closest to the desired size.

`(setf text-style-mapping)` [Generic function]

Arguments: *`mapping port text-style`* `&optional` *`character-set`*

■ Sets the text style mapping for *`port`*, *`character-set`*, and *`text-style`* to *`mapping`*. *`port`*, *`character-set`*, and *`text-style`* are as for **`text-style-mapping`**. *`mapping`* is either a font name or a list of the form `(:style` *`family face size`*`)`; in the latter case, the given style is translated at runtime into the font represented by the specified style. *`character-set`* defaults to the standard character set.

`make-text-style` [Function]

Arguments: *`family face size`*

■ Creates a text style object with the supplied characteristics. Generally, there is no need to call **`make-text-style`**; you should use **`parse-text-style`** or **`merge-text-styles`** instead.

■ The arguments can have the following values:

family

One of `:fix`, `:serif`, `:sans-serif`, or `nil`.

face

One of `:roman`, `:bold`, `:italic`, `(:bold :italic)`, or `nil`.

size

One of the logical sizes (`:tiny`, `:very-small`, `:small`, `:normal`, `:large`, `:very-large`, `:huge`, `:smaller`, `:larger`), or a real number representing the size in printer's points, or `nil`.

The following macros can be used to change the current text style for a stream by merging the specified style with the stream's current text style. They are intended as abbreviations for calling **`with-drawing-options`** directly.

`with-text-style` **[Macro]**

Arguments: *`(medium style)`* `&body` *`body`*

- Binds the current text style of *`medium`* to correspond to the new text *`style`*, within the *`body`*. *`style`* is either a text style specifier or a text style object. The default for *`medium`* is `*standard-output*`.
- This is the same as:

```
(clim:with-drawing-options (medium :text-style style) body)
```

- Note that **`with-text-style`** affects **`medium-text-style`**.

`with-text-face` **[Macro]**

Arguments: `(`*`medium face`*`)` `&body` *`body`*

- Binds the current text face of *`medium`* to correspond to the new text *`face`*, within the *`body`*. *`face`* is one of `:roman`, `:bold`, `:italic`, `(:bold :italic)`, or `nil`. The default for *`medium`* is `*standard-output*`.
- This is the same as:

```
(clim:with-drawing-options (medium :text-face face) body)
```

- Note that **`with-text-face`** affects **`medium-text-style`**.

`with-text-family` **[Macro]**

Arguments: `(`*`medium family`*`)` `&body` *`body`*

- Binds the current text family of medium to correspond to the new text family, within the body. family is one of `:fix`, `:serif`, `:sans-serif`, or `nil`. The default for medium is `*standard-output*`.
- This is the same as:

```
(clim:with-drawing-options (medium :text-family family) body)
```

- Note that **`with-text-family`** affects **`medium-text-style`**.

with-text-size **[Macro]**

Arguments: (*medium size*) &body *body*

- Binds the current text size of *medium* to correspond to the new text *size*, within the *body*. *size* is one of the logical sizes (`:normal`, `:small`, `:large`, `:very-small`, `:very-large`, `:smaller`, `:larger`), or a real number representing the size in printer's points, or `nil`. The default for *medium* is `*standard-output*`.
- This is the same as:

```
(clim:with-drawing-options (medium :text-size size) body)
```

- Note that **with-text-size** affects **medium-text-style**.

Chapter 6 Drawing in color in CLIM

6.1 Concepts of drawing in color in CLIM

To draw in color, you supply the `:ink` drawing option to CLIM's drawing functions when using streams opened under a color console.

Abstractly, the drawing functions work by selecting a region of the drawing plane and painting it with color. At the display device level, there are usually functions that draw a particular shape with the specified color.

The region to be painted is the intersection of the shape specified by the drawing function and the `:clipping-region` drawing option, which is then transformed by the `:transformation` drawing option. The shape can be a graphical area (such as a rectangle or an ellipse), a path (such as a line segment or the outline of an ellipse), or the letterforms of text.

Use the `:ink` drawing option to specify how to color this region of the can be a drawing plane. The value for `:ink` is often a color, but you can also specify a more general design for `:ink`. When you use a design for `:ink`, you can control the coloring-in process by specifying a new color of the drawing plane for each ideal point in the shape being drawn. (Note that this can depend on the coordinates of the point, and on the current color at that point in the drawing plane). For more information, see chapter 7 **Drawing with Designs in CLIM**.

Along with its drawing plane, a medium has a foreground and a background. The foreground is the default ink when the `:ink` drawing option is not specified. The background is drawn all over the drawing plane before any output is drawn. You can erase by drawing the background over the region to be erased. You can change the foreground or background at any time.

When you change the background, the contents of the drawing plane is redrawn. The effect is as if everything on the drawing plane is erased, the background is drawn on the entire drawing plane, and then everything that was ever drawn (provided it was saved in the output history) is redrawn using the new background.

6.1.1 CLIM color objects

A color in CLIM is an object representing the intuitive definition of color: white, black, red, pale yellow, and so forth. The visual appearance of a single point is completely described by its color.

A color can be specified by three real numbers between 0 and 1 inclusive, giving the amounts of red, green, and blue. Three 0's mean black; three 1's mean white. A color can also be specified by three numbers giving the intensity, hue, and saturation. A totally unsaturated color (a shade of gray) can be specified by a single real number between 0 and 1, giving the amount of white.

You can obtain a color object by calling one of **`make-rgb-color`**, **`make-ihs-color`**, **`make-gray-color`**, or **`find-named-color`** or by using one of the predefined colors listed in Predefined Color Names in CLIM. Specifying a color object as the `:ink` drawing option, the foreground, or the background causes CLIM to use that color in the appropriate drawing operations.

Rendering of colors

When CLIM renders the graphics and text in the drawing plane onto a real display device, physical limitations of the display device force the visual appearance to be an approximation of the drawing plane. Colors that the hardware doesn't support might be approximated by using a different color, or by using a stipple pattern. Even primary colors such as red and green can't be guaranteed to have distinct visual appearance on all devices, so if device independence is desired it is best to use **`make-contrasting-inks`** rather than a fixed palette of colors.

The region of the display device that gets colored when rendering a path or text is controlled by the line-style or text-style, respectively.

Palettes

All drawing is done via a palette. A palette is an opaque data structure that contains mappings of CLIM colors to port specific pixel values. A palette is associated with a port and can only be used when drawing on that port. It is not necessary to specify a palette when drawing as each sheet has an associated palette which is automatically used when drawing on that sheet. Each port has a default palette and by default when a sheet is grafted it takes the default palette of the port of the root window onto which the sheet is grafted.

Application frames can have their own separate palettes. This is useful if a particular frame needs a large number of color resources as it does not need to share palettes with other applications on the screen. Depending on the hardware of the host windowing system, this palette will be installed all the time for the frames top level window or only when for example the pointer is in that window. An application frame (and all of its panes) is associated with a palette when the frame is realized by the frame manager. Frame mangers are created by default with the default palette of their port but can be created with a new palette in order to provide this palette for any frames managed by that manager.

In addition to providing internal mappings of CLIM colors, a palette can also be queried to determine the visual type of the port. In particular it can be used to determine whether a port is monochrome or color and whether or not the display hardware supports writable color maps.

`palette` **[Class]**

■ The `palette` class is the protocol class for a palette. If you want to create a new class that behaves like a palette it should be a subclass of `palette`. Subclasses of `palette` must obey the palette protocol.

`palettep` [Function]

Arguments: *`object`*

■ Returns true if *`object`* is a palette, otherwise returns false.

The following two functions comprise the color protocol. Both of them return properties of the palette.

`palette-color-p` [Generic function]

Arguments: *`palette`*

■ Returns true if *`palette`* is associated with a port which supports color, or false if the port only supports monochrome.

`palette-mutable-p` [Generic function]

Arguments: *`palette`*

■ Returns true if *`palette`* is associated with a port which supports dynamic colors (i.e. has a writable hardware color map), otherwise returns false.

make-palette [Generic function]

Arguments: *port*

■ Return a member of the class palette. The *port* argument specifies which port the palette is associated with.

frame-palette [Generic function]

Arguments: *frame*

■ Returns the palette currently being used by frame. For managed frames the following forms evaluate to the same thing:

```
(frame-palette frame)
(frame-manager-palette (frame-manager frame))
```

For unmanaged frames, frame-palette returns the default palette for the port associated with the frame.

■ Note. Earlier documentation stated that you could use setf on this to change the frame's palette. This is not correct. If you wish to change the palette associated with the frame you should instead change the palette associated with frame's frame-manager.

frame-manager-palette [Generic function]

Arguments: *frame-manager*

■ Returns the palette that will be used by all the frames managed by *frame-manager*. You can use **setf** on this to change the frame-manager's palette. This will change the palette used by all frames adopted by the frame-manager and will force them to be repainted using the new palette.

■ Note. The *CLIM 2.0 User Guide* description of function is incorrect. Also in CLIM 2.0 **setf**'ing **frame-manager-palette** did not work correctly.

palette-full [Condition]

■ The condition signaled when an attempt is made to allocate a color in a palette which is full.

palette-full-palette [Generic function]

Arguments: *palette-full-condition*

■ Returns the palette associated with *palette-full-condition*.

palette-full-color [Generic function]

Arguments: *palette-full-condition*

■ Returns the color associated with *palette-full-condition*.

use-closest-color [Variable]

■ When non-nil, the closest available color will be used if the palette fills up and the use-other-color restart is found. When the value is :warn a warning is also given stating the desired color and the actual color used.

When nil the palette-full condition is always signaled if the palette fills up.

The default value is :warn.

find-closest-matching-color [Generic function]

Arguments: *palette desired-color*

■ Returns the closest matching color to *desired-color* available in *palette* and the distance between the returned color and the desired color. The distance is defined as the sum of the squares of the differences between the rgb values of the two colors. The function will return `nil` if *palette* is empty.

use-other-color [Named restart]

Arguments: *other-color*

■ When invoked this restart causes *other-color* to be used when a `palette-full` condition is signaled. This restart is not always available when the `palette-full` condition is signaled; it is unavailable, for example, when allocating a dynamic or layered color or if **add-colors-to-palette** fills the palette.

cadd-colors-to-palette [Generic function]

Arguments: *palette* &rest *colors*

■ This function allocates all of *colors* into *palette*. If the palette fills up during allocation of any of the colors, no colors are added to the palette and the *palette-full* condition is signaled.

This function can be used to determine in advance of running an application if there is enough room in the default palette allowing the application to create its own palette if there is not.

remove-colors-from-palette [Generic function]

Arguments: *palette &rest colors*

■ This function de-allocates all of colors from palette. It is an error to remove colors which are currently in use by a frame using the palette.

port-default-palette [Generic function]

Arguments: *port*

■ The default palette for the port. CLIM arranges to set this up based on the type of display server the port is connected to. The palette for a monochrome display will differ from the palette for a gray-scale display, which will differ from the palette for a full color display.

6.2 CLIM Operators for Drawing in Color

make-ihs-color [Function]

Arguments: *intensity hue saturation*

■ Creates a color object. *intensity* is a real number between 0 and the square root of 3 inclusive. *hue* and *saturation* are real numbers between 0 and 1 inclusive.

make-rgb-color [Function]

Arguments: *red green blue*

■ Creates a color object. *red*, *green*, and *blue* are real numbers between 0 and 1 inclusive that specify the values of the corresponding color components.

make-gray-color [Function]

Arguments: *luminance*

■ Creates a color object. *luminance* is a real number between 0 and 1 inclusive. 0 means black and 1 means white.

color-ihs [Generic function]

Arguments: *color*

■ Returns three values, the *intensity*, *hue*, and *saturation* components of *color*. The first value is a real number between 0 and the square root of 3 (inclusive). The second and third values are real numbers between 0 and 1 (inclusive).

color-rgb [Generic function]

Arguments: *color*

■ Returns three values, the *red*, *green*, and *blue* components of *color*. The values are real numbers between 0 and 1 inclusive.

make-contrasting-inks [Function]

Arguments: *n* &optional *k*

■ Makes a vector of *n* inks with different appearances.

If *k* (an integer between 0 and *n-1*) is supplied, **make-contrasting-inks** returns the *k*'th design.

The value returned by contrasting-inks-limit (defined next) is the maximum number of contrasting inks supported and so the maximum allowable value for *n*. If *n* is larger than the limit, **make-contrasting-inks** signals an error. The limit is the value of `contrasting-inks-limit`, defined next. This value should be at least 8 in any implementation.

The rendering of the design may be a color or a stippled pattern, depending on whether the output medium supports color.

contrasting-inks-limit [Function]

Arguments: *port*

■ Returns the number of contrasting inks that the port *port* can generate. In Allegro CLIM, the value on all platforms is at least 8.

Device colors

`clim-utils:device-color` [Class]

■ The class of colors that are specific to a particular display. Instances of this class represent particular entries in the colormap associated with a palette by referring to the particular pixel value.

`clim-utils:device-color` is a subclass of `color`.

clim-utils:device-color-pixel [Generic function]

Arguments: *device-color*

■ Returns the pixel value of device-color. For all ports pixel values are integers.

clim-utils:device-color-palette [Generic function]

Arguments: *device-color*

■ Returns the palette of device-color.

clim-utils:device-color-color [Generic function]

Arguments: *device-color*

■ Returns the actual color associated with device-color. This function looks up the particular entry in the colormap associated with device-color.

clim-utils:make-device-color [Generic function]

Arguments: *palette pixel*

■ Creates and returns a device-color representing the *pixel*'th entry in the colormap associated with palette. For all current ports pixel should be an integer.

Color conversion functionality

clim-utils:convert-rgb-to-ihs [Function]

Arguments: *red green blue*

■ Returns three values: intensity, hue and saturation. Converts an rgb color specification to ihs. Both forms below evaluate to the same thing:

```
(clim-utils:convert-rgb-to-ihs r g b)
(color-ihs (make-rgb-color r g b))
```

clim-utils:convert-ihs-to-rgb [Function]

Arguments: *intensity hue saturation*

■ Returns three values: red, blue and green. Converts an ihs color specification to rgb. Both forms below evaluate to the same thing:

```
(clim-utils:convert-ihs-to-rgb i h s)
(color-rgb (make-ihs-color i h s))
```

6.2.1 Dynamic colors and layered colors

Dynamic colors

A dynamic color can be the value of the `:ink` argument. Drawing with a dynamic color has the same effect as drawing with a member of the class color except that the actual color displayed on the screen can be quickly changed. Dynamic colors rely on the display hardware providing writable color maps. At any one time a dynamic color is associated with an actual color and that is the color which appears on the display.

make-dynamic-color [Generic function]

Arguments: `&key (:color +black+)`

■ Returns a design which is displayed as the solid design given as the color argument.

dynamic-color-color [Generic function]

Arguments: *dynamic-color*

■ Returns the actual color associated with the given dynamic color.

`(setf dynamic-color-color)` [Generic function]

Arguments: *color dynamic-color*

■ Changes the actual color associated with *dynamic-color* to *color* and returns *color*. If any drawing has been done with *dynamic-color*, changes to the displayed color will be immediate. Note that this has the same effect as **`recolor-dynamic-color`**.

`recolor-dynamic-color` [Generic function]

Arguments: *dynamic-color color*

■ Changes the actual color associated with *dynamic-color* to *color* and returns *color*. If any drawing has been done with *dynamic-color*, changes to the displayed color will be immediate. Note that this has the effect as **`(setf dynamic-color-color)`**. However, **`recolor-dynamic-color`** can also be used with layered (see below).

`with-delayed-recoloring` [Macro]

Arguments: &body *body*

■ During the execution of body any calls to **`recolor-dynamic-color`** or **`(setf dynamic-color-color)`** do not take any effect and are instead cached until exiting the extent of **`with-delayed-recoloring`** at which time they all take effect. In the case of nested **`with-delayed-recoloring`**, no color changes will take place until exiting the extent of the outermost **`with-delayed-recoloring`**.

■ This macro is provided to take advantage of the fact that certain window systems allow multiple color map entries to be written in one go.

Layered colors

A layered color may be the value of the `:ink` argument. Layered colors provide multiple layers of independent colors such that drawing can be performed which affects some layers and not others. This facility can be used to perform animation and fast overlays.

Conceptually a layered color is an *n*-dimensional array of dynamic colors where *n* is the number of layers. Initially, on creation of the layered color, each of the dynamic colors is set to `+black+`. Each layer is a positive integer specifying one dimension of the array. The total number of dynamic colors defined is given as the product of all the layers.

A group color is specified by a set of layers in much the same way as the dimensions of a layered color are defined. The layers of a group color act as indices into the array of dynamic colors associated with the layered color. Group colors can be specified with one or more layers as `nil`. The affect of this is to define an incomplete group color which is associated with more than one dynamic color. If none of the layers are specified as nil then the group color is complete.

A complete layered color specifies exactly one dynamic color.

Note that the dynamic colors associated with each of all the possible complete layered colors forms a non overlapping and exhaustive set of all the dynamic colors associated with the layered color set.

Drawing with a complete layered color is the same as drawing with the single dynamic color.

A complete layered color can be mutated with **`recolor-dynamic-color`** giving the layered color as the first argument. Therefore, though the color drawn with is fully specified, the actual color displayed depends on what real color the dynamic color is associated with.

An incomplete layered color specifies several dynamic colors. The number of dynamic colors is given by the product of those layers in the layered color for which the corresponding layer in the layered color is `nil`. The actual set of dynamic colors specified is given as those dynamic colors whose indices in the layer's associated array match the corresponding layers of the layered color with any `nil`s acting as wildcards.

Note that the dynamic colors associated with each of all the possible incomplete layered colors forms an overlapping and exhaustive set of all the dynamic colors associated with the layer.

Drawing with an incomplete layered color is in effect a two stage process. Firstly a complete layered color is made from the incomplete layered color and then that complete layered color is drawn with as described in the previous paragraph. To make the complete layered color, each `nil` in the incomplete layered color is replaced by the corresponding layer of the complete layered color which is being drawn over. Conceptually this process is performed on each pixel on which the incomplete layered color is drawn. The effects of drawing with an incomplete color on any pixels which were not previously drawn with a layered color is undefined.

An incomplete layered color can be mutated with **`recolor-dynamic-color`** giving the layered color as the first argument. Each of the set of dynamic colors associated with the layered color is mutated. Since the dynamic colors associated with two different incomplete layered colors can overlap, it is important to take care in the order in which calls to **`recolor-dynamic-color`** are made.

`make-layered-color-set` [Function]

Arguments: `&rest` *`layers`*

■ Returns a layered color set with the specified *`layers`*. Each layer must be a positive integer. Note that using many or large layers can result in a large array of dynamic colors all of which need to be allocated as writable entries in the host display's color map. This can lead to rapid consumption of the hosts color resources.

`layered-color` [Generic function]

Arguments: *`set`* `&rest` *`layers`*

■ Returns a layered color with the specified *`layers`*. The number of layers given must be the same as the number of layers with which *`set`* was created and each layer must be either `nil` or a non-negative integer which is less than the corresponding layer in the layer. If any of the layers are `nil` then the layered color is incomplete, otherwise it is complete.

6.3 Predefined color names in CLIM

Where the host window system has a database of named colors it is possible to query the database to find the rgb color corresponding to a particular name. This can be useful in helping the sharing of limited color resources as any other application running on that display may be able to share colors if they also are selecting colors from the database. It also is useful in maintaining color consistency across platforms because the rgb values in a particular host's database should be adjusted to handle any variations in screen hardware resulting in device dependent rgb values.

Therefore, there are only a few pre-defined colors in CLIM. The following symbols name these predefined colors. All are in the `clim` package.

```
+black+    +white+    +red+      +green+
+blue+     +magenta+  +cyan+     +yellow+
```

Colors available on your machine are typically named in some system file. For X11R4, the file naming available predefined colors is typically something like */X11/R4/mit/rgb/rgb.txt* (check with your system administrator for the equivalent file on your machine). The color corresponding to the color names in that file can be found by calling **`find-named-color`**.

`find-named-color` [Function]

Arguments: *name palette* &key errorp

■ Finds the color named name in the palette *palette*. The palette associated with the current application frame can be found by calling **`frame-palette`**.

If the color is not found and *errorp* is `t`, the `color-not-found` error is signaled. Otherwise if the color is not found, this function returns `nil`.

Here are a couple of examples. We use the palette for `*test-frame*`.

```
(find-named-color "light grey" (frame-palette *test-frame*))
→ #<CLIM-UTILS:GRAY-COLOR 66% Gray @ #x107d932>
(find-named-color "navy blue" (frame-palette *test-frame*))
→ #<CLIM-UTILS:RG-COLOR R=0.13672084 G=0.13672084 B=0.55469596 @ #x107de22>
```

[This page intentionally left blank.]

Chapter 7 Drawing with designs in CLIM

7.1 Concepts of Designs in CLIM

A design is an object that represents a way of arranging colors and opacities in the drawing plane. The simplest kind of design is a color, which simply places a constant color at every point in the drawing plane. See the chapter 6 **Drawing in color in CLIM**.

This chapter describes more complex kinds of design, which place different colors at different points in the drawing plane or compute the color from other information, such as the color previously at that point in the drawing plane. Not all of the features described in this chapter are supported in the present implementation.

Recall that the drawing functions work by selecting a region of the drawing plane and painting it with color, and that the `:ink` drawing option specifies how to color this region. The value of the `:ink` drawing option can be any kind of design, any member of the class `design`. The values of **`medium-foreground`**, **`medium-background`**, and **`medium-ink`** are also designs. Not all designs are supported as the arguments to the `:ink` drawing option, or as a foreground or background in the present implementation.

A design can be characterized in several different ways:

- All designs are either *bounded* or *unbounded*. Bounded designs are transparent everywhere beyond a certain distance from a certain point. Drawing a bounded design has no effect on the drawing plane outside that distance. Unbounded designs have points of non-zero opacity arbitrarily far from the origin. Drawing an unbounded design affects the entire drawing plane.
- All designs are either *uniform* or *non-uniform*. Uniform designs have the same color and opacity at every point in the drawing plane. Uniform designs are always unbounded, unless they are completely transparent.
- All designs are either *solid* or *translucent*. At each point a solid design is either completely opaque or completely transparent. A solid design can be opaque at some points and transparent at others. In translucent designs, at least one point has an opacity that is intermediate between completely opaque and completely transparent.
- All designs are either *colorless* or *colored*. Drawing a colorless design uses a default color specified by the medium's foreground design. This is done by drawing with

  ```
  (compose-in +foreground-ink+ the-colorless-design).
  ```

A variety of designs are available. See the following sections:

6.1 **Concepts of drawing in color in CLIM**

7.2 **Indirect ink in CLIM**

7.3 **Flipping ink in CLIM**

7.2 Indirect Ink in CLIM

Drawing with an *indirect ink* looks the same as drawing another design named directly. For example, `+foreground-ink+` is a design that draws the medium's foreground design. Indirect inks exist as an abbreviation for using **`medium-foreground`** or **`medium-background`**, and for the benefit of output recording. For example, one can draw with `+foreground-ink+`, change to a different **`medium-foreground`**, and replay the output record; the replayed output will come out in the new color.

If the current foreground is the color red, drawing with `+foreground-ink+` means to draw with the foreground, whatever it is. On the other hand, drawing with `+red+` means to draw with the color red, even if the foreground is changed to green.

You can change the foreground or background design at any time. This changes the contents of the drawing plane. Changing the background has the effect of erasing the drawing plane, drawing the new background design all over the drawing plane, and then replacing everything that was ever drawn (provided it was saved in the output history) is redrawn using the new foreground and background.

`+foreground-ink+` is the default value of the `:ink` drawing option.

If an infinite recursion is created using an indirect ink, an error is signaled when the recursion is created, when the design is used for drawing, or both.

In the present implementation, the foreground and background must be color objects.

Two indirect inks are defined:

`+foreground-ink+` **[Constant]**

- An indirect ink that uses the medium's foreground design.

`+background-ink+` **[Constant]**

- An indirect ink that uses the medium's background design.

7.3 Flipping Ink in CLIM

You can use a flipping ink to interchange occurrences of two colors. The purpose of flipping is to allow the use of XOR hacks for temporary changes to the display. On X Windows, only `+flipping-ink+` is supported at present

`make-flipping-ink` **[Function]**

Arguments: *`design1 design2`*

- Returns a design that interchanges occurrences of two designs. Drawing this design over a background changes the color in the background that would have been drawn by *`design1`* at that point into the color that would have been drawn by *`design2`* at that point, and vice versa.
- In the present implementation, both designs must be colors.

`+flipping-ink+` **[Constant]**

■ A flipping ink that flips `+foreground-ink+` and `+background-ink+`. You can think of this as an xor on monochrome displays.

7.4 Concepts of patterned designs in CLIM

Patterned designs are non-uniform designs that have a certain regularity. These include patterns, stencils, tiled designs, and transformed designs.

In the present implementation, patterned designs are not supported as a foreground or background, and the only patterned designs supported as the `:ink` drawing option are tilings of patterns of `+background-ink+` (or `+transparent-ink+`) and `+foreground-ink+`.

Patterns and Stencils

Patterning creates a bounded rectangular arrangement of designs. Drawing a pattern draws a different design in each rectangular cell of the pattern. To create a pattern, use **`make-pattern`**. To repeat a pattern so it fills the drawing plane, apply **`make-rectangular-tile`** to a pattern.

A stencil is a special kind of pattern that contains only opacities. The name stencil refers to their use with **`compose-in`** and **`compose-over`**.

Tiling

Tiling repeats a rectangular portion of a design throughout the drawing plane. This is most commonly used with patterns. Use **`make-rectangular`**-tile to make a tiled design.

Transforming Designs

The functions **`transform-region`** and **`untransform-region`** accept any design as their second argument and apply a coordinate transformation to the design. The result is a design that might be freshly constructed or might be an existing object.

Transforming a uniform design simply returns the argument. Transforming a composite, flipping, or indirect design applies the transformation to the component design(s). Transforming a pattern, tile, or output record design is described in the sections on those designs.

7.4.1 Operators for patterned designs in CLIM

`make-pattern` **[Function]**

Arguments: *`array designs`*

■ Creates a pattern design that has `(array-dimension` *`2d-array`* `0)` cells in the vertical direction and `(array-dimension` *`2d-array`* `1)` cells in the horizontal direction.

`array` must be a two-dimensional array of non-negative integers, each of which is less than the length of *`designs`*. *`designs`* must be a sequence of designs. The design in cell (*`i`*,*`j`*) of the resulting pattern is the *`n`*'th element of *`designs`*, if *`n`* is the value of `(aref` *`array i j`*`)`. For example, *`array`* can be a bit-array and *`designs`* can be a list of two designs, the design drawn for 0 and the one drawn for 1.

Each cell of a pattern can be regarded as a hole that allows the design in it to show through. Each cell might have a different design in it. The portion of the design that shows through a hole is the portion on the part of the drawing plane where the hole is located. In other words, incorporating a design

into a pattern does not change its alignment to the drawing plane, and does not apply a coordinate transformation to the design. Drawing a pattern collects the pieces of designs that show through all the holes and draws the pieces where the holes lie on the drawing plane. The pattern is completely transparent outside the area defined by the array.

Each cell of a pattern occupies a 1 by 1 square. You can use **transform-region** to scale the pattern to a different cell size and shape, or to rotate the pattern so that the rectangular cells become diamond-shaped. Applying a coordinate transformation to a pattern does not affect the designs that make up the pattern. It only changes the position, size, and shape of the cells' holes, allowing different portions of the designs in the cells to show through. Consequently, applying **make-rectangular-tile** to a pattern of nonuniform designs can produce a different appearance in each tile. The pattern cells' holes are tiled, but the designs in the cells are not tiled and a different portion of each of those designs shows through in each tile.

■ If *array* or *designs* is modified after calling **make-pattern**, the consequences are unspecified.

IMPLEMENTATION LIMITATION: In the present implementation, patterned designs are not supported as a foreground or background.

pattern-width [Function]

Arguments: *pattern*

■ Returns the width of the pattern *pattern* (that is, `(array-dimension` *2d-array* `1)` of the *2d-array* used to create the pattern).

pattern-height [Function]

Arguments: *pattern*

■ Returns the height of the pattern *pattern* (that is, `(array-dimension` *2d-array* `0)` of the *2d-array* used to create the pattern).

pattern-array [Generic function]

Arguments: *pattern*

Returns the array associated with pattern. The following holds:

```
(pattern-array (make-pattern array designs)) -> array
```

pattern-designs [Generic function]

Arguments: *pattern*

■ Returns the designs associated with *pattern*. The following holds:

```
(pattern-designs (make-pattern array designs)) -> designs
```

with the exception that **make-pattern** may coerce `designs` to an equivalent sequence of a different type.

make-pattern-from-pixmap [Generic function]

Arguments: *pixmap &key x y width height*

■ Creates and returns a pattern from a pixmap. *x* and y and *width* and *height* define the area of the pixmap that is used. *x* and y default to 0. *width* and *height* default to the respective dimensions of *pixmap*.

make-stencil [Function]

Arguments: *array*

■ Creates a pattern design that has (array-dimension *array* 0) cells vertically and (array-dimension *array* 1) cells horizontally. *array* must be a two-dimensional array of real numbers between 0 and 1. The design in cell (i,j) of the resulting pattern is the value of the following:

```
(clim:make-opacity (aref array i j))
```

The stencil opacity of the result at a given point in the drawing plane depends on which cell that point falls in. If the point is in cell (i,j), the stencil opacity is (aref *array* i j). The stencil opacity is 0 outside the region defined by the array.

Each cell of a pattern occupies a 1 x 1 square. The entity protocol can be used to scale the pattern to a different cell size and shape, or to rotate the pattern so that the rectangular cells become diamond-shaped.

■ If *array* is modified after calling **make-stencil**, the consequences are unspecified.

make-rectangular-tile [Function]

Arguments: *design width height*

■ Creates a design that tiles the specified rectangular portion of *design* across the entire drawing plane. The resulting design repeats with a period of *width* horizontally and *height* vertically. The portion of the argument *design* that appears in each tile is a rectangle whose top-left corner is at (0,0) and whose bottom-right corner is at (*width*,*height*).

The repetition of *design* is accomplished by applying a coordinate transformation to shift *design* into position for each tile, and then extracting an *width* by *height* portion of that design.

Applying a coordinate transformation to a rectangular tile does not change the portion of the argument *design* that appears in each tile. It can change the period, phase, and orientation of the repeated pattern of tiles.

draw-pattern* [Function]

Arguments: *stream pattern x y* &key clipping-region transformation

■ Draws the pattern *pattern* on *stream* at the position (*x*,*y*). *pattern* is a design created by calling **make-pattern**.

For example, the following creates a pattern whose zero values are colored with the background of *stream* and whose one values are colored with the foreground of *stream*

```
(clim:make-pattern #2A((0 0 0 1 1 0 0 0)
                       (0 0 1 1 1 1 0 0)
                       (0 1 1 1 1 1 1 0)
                       (1 1 1 0 0 1 1 1)
                       (1 1 1 0 0 1 1 1)
                       (0 1 1 1 1 1 1 0)
                       (0 0 1 1 1 1 0 0)
                       (0 0 0 1 1 0 0 0))
                   (list clim:+background-ink+ clim:+foreground-ink+))
```

You could also make the above pattern translucent by using +transparent-ink+ instead of +background-ink+. In that case, the zero values would allow the previous output to show through.

7.4.2 Reading patterns from X11 image files

Allegro CLIM supplies some functions that read standard X11 bitmap and pixmaps files.

`read-bitmap-file` [Function]

Arguments: *`pathname`* `&key (format :bitmap) (port (find-port))`

■ Reads a bitmap file named by *`pathname`*. *`port`* specifies the port; it defaults to `(find-port)`. *`format`* can be one of: the keywords in table 7.1. The default is `:bitmap`.

Table 7.1: Possible values for :format argument:

Value	Meaning
`:bitmap`	X bitmap format
`:pixmap`	X pixmap format version 1 (also know as xpm)
`:pixmap-3`	X pixmap format version 3 (also know as xpm)

■ **`read-bitmap-file`** returns the 2-dimensional array of pixel values and returns a second value, a sequence of CLIM colors or color names, when *`format`* is `:pixmap` or `:pixmap-3`. Colors are returned unless *`port`* is `nil`, in which case color names are returned.

`make-pattern-from-bitmap-file` [Function]

Arguments: *`file`* `&key designs (format :bitmap) (port (find-port))`

■ The function reads the contents of the bitmap or pixmap file *`file`* and creates a `pattern` object that represents the file.

`designs` is a sequence of CLIM designs (typically color objects) that will be used as the second argument in a call to **`make-pattern`**. *`designs`* must be supplied if no second value will be returned from **`read-bitmap-file`**.

`format` is as for **`read-bitmap-file`** above

`port` specifies the port. It defaults to `(find-port)`.

7.5 Concepts of compositing and translucent ink in CLIM

Translucent ink supports the following drawing techniques:

- Controlling Opacity
- Blending Colors
- Compositing

Controlling Opacity

Opacity controls how new output covers previous output. Intermediate opacity values result in color blending so that the earlier picture shows through what is drawn on top of it.

An opacity is a real number between 0 and 1; 0 is completely transparent, 1 is completely opaque, and fractions are translucent. The opacity of a design is the degree to which it hides the previous contents of the drawing plane when it is drawn. Opacity can vary from totally opaque to totally transparent.

Use **make-opacity** or **make-stencil** to specify opacity.

IMPLEMENTATION LIMITATION: Opacity values that are not either fully transparent or fully opaque are not fully supported. Translucent opacities are simulated by using stipples.

Color Blending

Drawing a design that is not completely opaque at all points allows the previous contents of the drawing plane to show through. The simplest case is drawing a *solid design*. Where the design is opaque, it replaces the previous contents of the drawing plane. Where the design is transparent, it leaves the drawing plane unchanged.

In the more general case of drawing a translucent design, the resulting color is a blend of the design's color and the previous color of the drawing plane. For purposes of color blending, the drawn design is called the foreground and the drawing plane is called the background.

The function **compose-over** performs a similar operation. It combines two designs to produce a design, rather than combining a design and the contents of the drawing plane to produce the new contents of the drawing plane. For purposes of color blending, the first argument to **compose-over** is called the foreground and the second argument is called the background.

Color blending is defined by an ideal function that operates on the color and opacity at a single point. (r_1,g_1,b_1,o_1) are the foreground color and opacity. (r_2,g_2,b_2,o_2) are the background color and opacity. (r_3,g_3,b_3,o_3) are the resulting color and opacity:

$$F:(r_1,g_1,b_1,o_1,r_2,g_2,b_2,o_2) \rightarrow (r_3,g_3,b_3,o_3)$$

The color blending function F is conceptually applied at every point in the drawing plane.

The function F performs linear interpolation on all four components:

$$o_3 = o_1+(1-o_1)*o_2$$
$$r_3 = (o_1*r_1+(1-o_1)*o_2*r_2)/o_3$$
$$g_3 = (o_1*g_1+(1-o_1)*o_2*g_2)/o_3$$
$$b_3 = (o_1*b_1+(1-o_1)*o_2*b_2)/o_3$$

In Allegro CLIM, F is implemented exactly only if o_1 is zero or one or if o_2 is zero. If o_1 is zero, the result is the background. If o_1 is one or o_2 is zero, the result is the foreground. For fractional opacity values, CLIM will deviate from the ideal color blending function either because the current hardware has limited opacity resolution and CLIM can compute a different color blending function much more quickly.

If a medium's background design is not completely opaque at all points, the consequences are unspecified. Consequently, a drawing plane is always opaque and drawing can use simplified color blending that assumes $o_2 = 1$ and $o_3 = 1$. However, **compose-over** must handle a non-opaque background correctly.

Compositing

IMPLEMENTATION LIMITATION: Compositing is not supported in this release.

Compositing creates a design whose appearance at each point is a composite of the appearances of two other designs at that point. Three varieties of compositing are provided: composing *over*, composing *in*, and composing *out*.

You can use **compose-over**, **compose-in**, or **compose-out** to create CLIM composite designs.

7.5.1 Operators for Translucent Ink and Compositing in CLIM

The following functions can be used to create an opacity object, and to compose a new ink from a color and an opacity. (The three composition operators can also be used to compose more complex designs.)

IMPLEMENTATION LIMITATION: Note that the present implementation of CLIM only supports opacities that are either fully opaque (opacity 1) or fully transparent (opacity 0).

`make-opacity` [Function]

Arguments: *value*

■ Creates a member of class `opacity` whose opacity is *value*, which is a real number in the range from 0 to 1 (inclusive), where 0 is fully transparent and 1 is fully opaque, but note implementation warning above: only 1 and 0 are supported.

`opacity-value` [Generic function]

Arguments: *opacity*

■ Returns the *value* of *opacity*, which is a real number in the range from 0 to 1 (inclusive).

`+transparent-ink+` [Constant]

■ When you draw a design that has areas of `+transparent-ink+`, the former background shows through in those areas. Typically, `+transparent-ink+` is used as one of the inks in a pattern so that parts of the pattern are transparent.

For example, the following creates a pattern whose zero values allow the previous draw areas of the stream to show through:

```
(clim:make-pattern #2A((0 0 0 1 1 0 0 0)
           (0 0 1 1 1 1 0 0)
           (0 1 1 1 1 1 1 0)
           (1 1 1 0 0 1 1 1)
           (1 1 1 0 0 1 1 1)
           (0 1 1 1 1 1 1 0)
           (0 0 1 1 1 1 0 0)
           (0 0 0 1 1 0 0 0))
         (list clim:+transparent-ink+ clim:+foreground-ink+))
```

`compose-over` [Generic function]

Arguments: *design1 design2*

IMPLEMENTATION LIMITATION: **`compose-over`** is not currently supported.

■ Composes a design that is equivalent to *design1* on top of *design2*. Drawing the resulting design produces the same visual appearance as drawing *design2* and then drawing *design1*, but might be faster and might not allow the intermediate state to be visible on the screen.

If both arguments are regions, **`compose-over`** is the same as **`region-union`**.

The result returned by **`compose-over`** might be freshly constructed or might be an existing object.

`compose-in` [Generic function]

Arguments: *design1 design2*

IMPLEMENTATION LIMITATION: **`compose-in`** is not currently supported.

■ Composes a design by using the color (or ink) of *design1* and clipping to the inside of *design2*. That is, *design2* specifies the mask to use for changing the shape of the design.

More precisely, at each point in the drawing plane the resulting design specifies a color and an opacity as follows: the color is the ink of *design1*. The opacity is the opacity of *design1*, multiplied by the *stencil opacity* of *design2*.

The *stencil opacity* of a design at a point is defined as the opacity that would result from drawing the design onto a fictitious medium whose drawing plane is initially completely transparent black (opacity and all color components are zero), and whose foreground and background are both opaque black. (With this definition, the stencil opacity of a member of class `opacity` is simply its value.)

■ If *design2* is a solid design, the effect of **`compose-in`** is to clip *design1* to *design2*. If *design2* is translucent, the effect is a soft matte.

■ If both arguments are regions, **`compose-in`** is the same as **`region-intersection`**.

■ The result returned by **`compose-in`** might be freshly constructed or might be an existing object.

`compose-out` [Generic function]

Arguments: *design1 design2*

IMPLEMENTATION LIMITATION: **`compose-out`** is not currently supported.

■ Composes a design by using the color (or ink) of *design1* and clipping to the outside of *design2*. That is, *design2* specifies the mask to use for changing the shape of the design.

More precisely, at each point in the drawing plane the resulting design specifies a color and an opacity as follows: the color is the ink of *design1*. The opacity is the opacity of *design1*, multiplied by 1 minus the stencil opacity of *design2*.

If *design2* is a solid design, the effect of **`compose-out`** is to clip *design1* to the complement of *design2*. If *design2* is translucent, the effect is a soft matte.

■ If both arguments are regions, **`compose-out`** is the same as **`region-difference`**.

■ The result returned by **`compose-out`** might be freshly constructed or might be an existing object.

7.6 Complex Designs in CLIM

IMPLEMENTATION LIMITATION: The designs described in this section are not supported as the `:ink` drawing option in the present implementation.

You can use **`make-design-from-output-record`** to make a design that replays *output-record* when drawn using **`draw-design`**.

`draw-design` [Generic function]

Arguments: *design stream* `&key ink clipping-region transformation line-style unit thickness joint-shape cap-shape dashes text-style text-family text-face text-size`

IMPLEMENTATION LIMITATION: **`draw-design`** is not currently supported.

■ Draws *design* on *stream*. *args* are additional keyword arguments that depend on the type of the *design*. For example, for designs that are paths (such as lines and unfilled circles), you may include the `:line-style` keyword.

■ The *design* types are:

`area`

Paints the specified region of the drawing plane with *`stream`*'s current ink.

`path`

Strokes the path with *`stream`*'s current ink under control of the line-style.

`point`

The same as **`draw-point`**.

a `color` or an `opacity`

Paints the entire drawing plane (subject to the clipping region).

`+nowhere+`

This has no effect.

- If *`design`* is a non-uniform design this paints the design positioned at coordinates (x=0, y=0).
- **`draw-design`** is currently supported for the following designs:
  - designs created by the geometric object constructors (such as **`make-line`** and **`make-ellipse`**)
  - designs created by **`compose-in`**, where the first argument is an ink and the second argument is a design
  - **`compose-over`** of designs created by **`compose-in`**
  - designs returned by **`make-design-from-output-record`**

`make-design-from-output-record` [Function]

Arguments: *`record`*

IMPLEMENTATION LIMITATION: **`make-design-from-output-record`** is not currently supported.

- Makes a design that replays the output record *`record`* when drawn by **`draw-design`**.
- Presently, only output records all of whose leaves are graphics displayed output records (such as the output records created by **`draw-line*`** and **`draw-ellipse*`**) can be turned into designs by **`make-design-from-output-record`**.
- You can use **`transform-region`** on the result of **`make-design-from-output-record`** in order to apply a transformation to it.

Any member of the class `region` is a solid, colorless design. The design is opaque at points in the region and transparent elsewhere.

See the section 3.6 **General Geometric Objects and Regions in CLIM**.

7.7 Achieving different drawing effects in CLIM

Here are some examples of how to achieve a variety of commonly used drawing effects:

Drawing in the foreground color

Use the default, or specify `:ink +foreground-ink+`
or `:ink (medium-foreground` *`medium`*`)`

Erasing

Specify `:ink +background-ink+`
or `:ink (medium-background` *`medium`*`)`

Drawing in color

Specify `:ink +green+`, or `:ink (make-color-rgb 0.6 0.0 0.4)`, or `:ink (find-named-color "green" (frame-palette` *`frame`*`))`

Painting a gray or colored wash over a display

IMPLEMENTATION LIMITATION: opacities are not implemented so these examples will not work.

Specify a translucent design as the ink, such as

```
:ink (clim:compose-in +black+ (clim:make-opacity 0.25))
:ink (clim:compose-in +red+ (clim:make-opacity 0.1))
:ink (clim:compose-in +foreground-ink+ (clim:make-opacity 0.75))
```

The last example can be abbreviated as `:ink (make-opacity 0.75)`. On a non-color, non-grayscale display this will turn into a stipple.

Drawing an opaque gray

Specify `:ink (make-gray-color 0.25)` to draw in a shade of gray independent of the window's foreground color. On a non-color, non-grayscale display this will turn into a stipple.

Drawing a faded but opaque version of the foreground color

IMPLEMENTATION LIMITATION: opacities are not currently implemented so this example will not work.

To draw at 25% of the normal contrast, specify:

```
:ink (clim:compose-over
       (clim:compose-in clim:+foreground-ink+
         (clim:make-opacity 0.25))
       clim:+background-ink+)
```

On a non-color, non-grayscale display this will probably turn into a stipple.

Drawing a stipple of little bricks

Specify `:ink bricks`, where `bricks` is defined as:

```
(clim:make-rectangular-tile
  (clim:make-pattern #2a((0 0 0 1 0 0 0 0)
                         (0 0 0 1 0 0 0 0)
                         (0 0 0 1 0 0 0 0)
                         (1 1 1 1 1 1 1 1)
                         (0 0 0 0 0 0 0 1)
                         (0 0 0 0 0 0 0 1)
                         (0 0 0 0 0 0 0 1)
                         (1 1 1 1 1 1 1 1))
  (list clim:+background-ink+
        clim:+foreground-ink+))
  8 8)
```

Drawing a tiled pattern

Specify

```
:ink (clim:make-rectangular-tile (clim:make-pattern array colors))
```

Drawing a pattern

Use

```
(clim:draw-pattern* medium (clim:make-pattern array colors) x y)
```

Chapter 8 Presentation types in CLIM

8.1 Concepts of CLIM presentation types

In object-oriented programming systems, applications are built around internal objects that model something in the real world. For example, an application that models a university has objects representing students, professors, and courses. A CAD system for designing circuits has objects representing gates, resistors, and so on. A desktop publishing system has objects representing paragraphs, headings, and drawings.

Users need to interact with the application objects. A CLIM user interface enables users to see a visual representation of the application objects, and to operate on them. The objects that appear on the screen are not the application objects themselves; they are objects called presentations that are one step removed. The visual representation of an object is a stand-in for the application object itself, in the same sense that the word 'cat' (or a picture of a cat) is a stand-in for a real cat.

In most user interface systems, the interface is constructed in terms of the objects in the toolkit; these objects must be converted, by the programmer, into the objects of the application. For example, choosing one of a set of objects might require you to build a radio box that has in it a set of buttons. Each of these buttons stands for one of the objects being chosen among. You must also write code that converts each of these buttons back into an application object, so that when the end-user picks on of the buttons, the correct application object is chosen.

If you later decide that the radio box is too cumbersome (perhaps because there are many selections in it), and you wish to change the user interface to use a list pane or option pane, you must then rewrite all of this code to use the new toolkit objects for list or option panes.

In CLIM, the user interface is constructed is terms of the application objects, and CLIM worries about the toolkit objects. Taking the same example, all you need to do is to specify that you wish to select one of the application objects by using the `member` type. CLIM deduces that this can be done via a radio box, and creates the radio box for you. If you later decide to use a list pane or option pane, you can advise CLIM to do this by explicitly specifying a *view.* At no time do you need to worry about the toolkit objects.

The most basic part of designing a CLIM user interface is to specify how users will interact with the application objects. There are two directions of interaction: you must present application objects to the user as output, and you must accept input from the user that indicates operations on the application objects. This is done with two basic functions, **`present`** and **`accept`**, and some related functions.

8.1.1 Presentations

CLIM keeps track of the association between a visual representation of an object and the object itself. CLIM maintains this association in a data structure called a *presentation*. A presentation embodies three things:

- The underlying application object
- Its presentation type
- Its visual representation

8.1.2 Output with its semantics attached

For example, a university application has a ‘student’ application object. The user sees a visual representation of a student, which might be a textual representation, or a graphical representation (such as a form with name, address, student id number), or even an image of the face of the student. The presentation type of the student is ‘student’; that is, the semantic type of the object that appears on the screen is ‘student’. Since the type of a displayed object is known, CLIM knows which operations are appropriate to perform on the displayed object, *irrespective* of what visual representation is being used for the object. For example, when a student is displayed, it is possible to perform operations such as ‘send tuition bill’ or ‘show transcript’.

8.1.3 Input context

Presentations are the basis of many of the higher-level application-building tools, which use **`accept`** to get input and **`present`** to do output. A command that takes arguments as input states the presentation type of each argument. This sets up an *input context*, in which presentations of that type are sensitive (they are highlighted when the pointer passes over them). When the user gives the ‘send tuition bill’ command, the input context is looking for a student, so any displayed students are sensitive. Presentations that have been output in previous user interactions retain their semantics. In other words, CLIM has recorded the fact that a student has been displayed, and has saved this information so that whenever the input context expects a student, all displayed students are sensitive.

8.1.4 Inheritance

CLIM presentation types can be designed to use inheritance, just as CLOS classes do. For example, a university might need to model `night-student`, which is a subclass of `student`. When the input context is looking for a student, night students are sensitive because they are represented as a subtype of `student`.

The set of presentation types forms a type lattice, an extension of the Common Lisp CLOS type lattice. When a new presentation type is defined as a subtype of another presentation type, it inherits all the attributes of the supertype except those explicitly overridden in the definition.

8.1.5 Presentation translators

You can define *presentation translators* to make the user interface of your application more flexible. For example, suppose the input context is expecting a command. Since CLIM commands are first class application objects, in this input context, all displayed commands are sensitive, so the user can point to one to execute it. However, suppose the user points to another kind of displayed object, such as a student. In the absence of a presentation translator, the student is not sensitive because the user must enter a command and cannot enter anything else to this input context.

In the presence of a presentation translator that translates from students to commands, however, the student would be sensitive. In one scenario, the student is highlighted, and the middle pointer button does ‘show transcript’ of the student.

8.1.6 What the application programmer does

By the time you get to the point of designing the user interface, you have probably designed the rest of the application and know what the application objects are. At this point, you need to do the following:

- First decide which types of application objects will be presented to the user as output and accepted from the user as input.
- For each type of application object that the user will see, assign a corresponding presentation type. You will need to define **`accept`** and **`present`** methods for these objects, unless these methods can be inherited from a superclass. In many cases, this means simply using a predefined presentation type. In other cases, you need to define a new presentation type. Usually the presentation type is the same as the class of the application object.
- Decide which of the application's operations should be available from the user interface, and define commands for each of these operations. These commands can be made available from a variety of interfaces (such as menus or dialogs, command lines, and so on). This is a detail that can be decided as you continue to develop the program.
- Use the application-building tools to specify the windows, menus, commands, and other elements of the user interface. Most of these elements will use the presentation types of your objects.

8.2 How to specify a CLIM presentation type

This section describes how to specify a CLIM presentation type. For a complete description of CLIM presentation types, options, and parameters, see the section 8.5 **Predefined presentation types in CLIM**.

Several CLIM operators take presentation types as arguments. You specify them using a *presentation type specifier*.

Most presentation type specifiers are also Common Lisp type specifiers, for example, the `boolean` presentation type is a Common Lisp type specifier. Not all presentation types are Common Lisp types, and not all Common Lisp types are presentation types, but there is a lot of overlap.

A presentation type specifier appears in one of the following three patterns:

```
name
(name parameters...)
((name parameters...) options...)
```

Each presentation type has a name, which is usually a symbol naming the presentation type. The name can also be a CLOS class object; this usage provides the support for anonymous CLOS classes.

The first pattern, *`name`*, indicates a simple presentation type, which can be one of the predefined presentation types or a user-defined presentation type.

Examples of the first pattern are:

`integer`
: A predefined presentation type

`pathname`
A predefined presentation type

`boolean`
A predefined presentation type

`student`
A user-defined presentation type

The second pattern, (*`name parameters`*...), supports parameterized presentation types, which are analogous to parameterized Common Lisp types. (In effect, CLIM extends CLOS to allow parameterized classes.) The parameters state a restriction on the presentation type, so a parameterized presentation type is a specialization, or a subset, of the presentation type of that name with no parameters.

Examples of the second pattern are:

`(integer 0 10)`
A parameterized type indicating an integer in the range of zero through ten.

`(string 25)`
A parameterized type indicating a string whose length is 25.

`(member :yes :no :maybe)`
A parameterized type which can be one of the three given values, `:yes`, `:no`, and `:maybe`.

The third pattern, ((*`name parameters`*...) *`options`*...), enables you to additionally supply options that affect the use or appearance of the presentation, but not its semantic meaning. The *`options`* are keyword/value pairs. The options are defined by the presentation type. All presentation types accept the `:description` option, which enables you to provide a string describing the presentation type. If provided, this option overrides the description specified in the **`define-presentation-type`** form, and also overrides the **`describe-presentation-type`** presentation method.

For example, you can use this form to specify an octal integer from 0 to 10:

```
((integer 0 10) :base 8)
```

Some presentation type options may appear as an option in any presentation type specifier. Currently, the only such option is `:description`.

8.3 Using CLIM presentation types for output

The reason for using presentations for program output is so that the objects presented will be acceptable to input functions. When you use presentations, CLIM manages all of the bookkeeping that remembers the presentations -- you needn't do any of this. What this means is that interfaces built using CLIM are live -- often, everything a user sees on the screen is active and available for input.

Suppose, for example, you present an object, such as 5, as a TV channel. When a command that takes a TV channel as an argument is issued or when a presentation translation function is looking for such a thing, the system will make that object sensitive. Also, when a command that is looking for a different kind of object (such as a highway number), the object 5 is not sensitive, because that object represents a TV channel, not a highway number.

A presentation includes not only the displayed representation itself, but also the object presented and its presentation type. When a presentation is output to a CLIM window, the object and presentation type are remembered -- that is, the object and type of the display at a particular set of window coordinates are

recorded in the window's *output history*. Because this information remains available, previously presented objects are themselves available for input to functions for accepting objects.

All of the above functionality is managed automatically by CLIM, but you can extend and modify the behavior in application-specific ways. For instance, this can be used to improve performance.

8.3.1 CLIM operators for presenting typed output

An application can use the following operators to produce output that will be associated with a given Lisp object and be declared to be of a specified presentation type. This output is saved in the window's output history as a presentation. Specifically, the presentation remembers the output that was performed (by saving the associated output record), the Lisp object associated with the output, and the presentation type specified at output time. The object can be any Lisp object.

CLOS provides these top-level facilities for presenting output. **`with-output-as-presentation`** is the most general operator, and **`present`** and **`present-to-string`** support common idioms.

`with-output-as-presentation` **[Macro]**

Arguments: `(`*`stream object type`* `&key single-box allow-sensitive-inferiors modifier parent record-type) &body` *`body`*

■ This macro generates a presentation from the output done in the *`body`* to the *`stream`*. In effect, it gives separate access to the two aspects of **`present`** -- recording the presentation and drawing the visual representation. The presentation's underlying object is *`object`*, and its presentation type is *`type`*.

For information on the syntax of specifying a presentation type, see the section 8.2 **How to specify a CLIM presentation type**.

■ All arguments of this macro are evaluated. **`with-output-as-presentation`** returns a presentation. Note that CLIM captures the presentation type for its own use, and you should not modify it once you have handed it to CLIM.

■ Each invocation of this macro results in the creation of a presentation object in the stream's output history unless output recording has been disabled or `:allow-sensitive-inferiors` is specified `nil` at a higher level, in which case the presentation object is not inserted into the history.

■ The arguments behave as follows:

`stream`

The stream to which output should be sent. The default is `*standard-output*`.

`single-box`

Controls how CLIM determines whether the pointer is pointing at this presentation and controls how this presentation is highlighted when it is sensitive. The possible values are:

`nil`

If the pointer is pointing at a visible piece of output (text or graphics) drawn as part of the visual representation of this presentation, it is considered to be pointing at this presentation. This presentation is highlighted by highlighting every visible piece of output that is drawn as part of its visual representation. This is the default.

`t`

If the pointer's position is inside the bounding rectangle of this presentation, it is considered to be pointing at this presentation. The presentation is highlighted by highlighting its bounding rectangle.

`:position`

Like `t` for determining whether the pointer is pointing at this presentation, like `nil` for highlighting. Supplying `:single-box :position` is useful when the visual representation of a presentation consists of one or more small graphical objects with a lot of space between them. In this case the default behavior offers only small targets that the user might find difficult to position the pointer over.

`:highlighting`

Like `nil` for determining whether the pointer is pointing at this presentation, like `t` for highlighting. Supplying `:single-box :highlighting` is useful when the default behavior produces an ugly appearance.

`allow-sensitive-inferiors`

When `nil`, specifies that nested calls to **`present`** or **`with-output-as-presentation`** inside this one should not generate presentations. The initial default is `t`; when there are nested calls to **`with-output-as-presentation`**, the default is sticky in that it is gotten from the outer call to **`with-output-as-presentation`**.

`modifier`

Not implemented in this release. Supplies a function of one argument (the new value) that can be called in order to store a new value for *`object`* after the user edits the presentation. The default is `nil`.

`record-type`

This option is useful when you have defined a customized record type to replace CLIM's default presentation output record type. It specifies the class of the output record to be created.

`present` [Function]

Arguments: *`object`* `&optional` *`presentation-type`*
`&key (stream *standard-output*) view modifier acceptably`
`for-context-type single-box allow-sensitive-inferiors`
`sensitive query-identifier prompt record-type`

■ Presents the object *`object`* whose presentation type is *`presentation-type`* to *`stream`*. The manner in which the object is displayed depends on the presentation type of the object; the display is done by the type's **`present`** method for the given *`view`*.

■ The arguments behave as follows:

`object`

The object to be presented.

`presentation-type`

A presentation type specifier, which may be a presentation type abbreviation. This defaults to `(presentation-type-of` *`object`*`)`.

`stream`

The stream to which output should be sent. The default is `*standard-output*`.

`view`

An object representing a view. The default is `(stream-default-view stream)`. For most streams, the default view is the textual view, `+textual-view+`.

`modifier`

`single-box`

`allow-sensitive-inferiors`

These are as for **`with-output-as-presentation`**.

sensitive

If `nil`, no presentation is produced. The default is `t`.

acceptably

Defaults to `nil`, which requests the **`present`** method to produce output designed to be read by the user. If `t`, this option requests the **`present`** method to produce output that can be parsed by the **`accept`** method. You will rarely need to use this, since this option makes no difference for most presentation types.

for-context-type

A presentation type indicating an input context. The present method can look at this to determine if the object should be presented differently. For example, the **`present`** method for the `command` presentation type uses this in order to determine whether to display a ":" before commands in `command-or-form` contexts. The *`for-context-type`* argument defaults to *`presentation-type`*. You will rarely need to use this, since this option makes no difference for most presentation types.

record-type

This option is useful when you have defined a customized record type to replace CLIM's default record type. It specifies the class of the output record to be created.

`present-to-string` [Function]

Arguments: *`object`* `&optional` *`presentation-type`* `&key view acceptably for-context-type string index`

■ Presents an object into a string in such a way that it can subsequently be accepted as input by **`accept-from-string`**.

■ **`present-to-string`** is the same as **`present`** within **`with-output-to-string`**. The other arguments are the same as for **`present`** and **`with-output-as-presentation`**.

8.3.2 Additional functions for operating on presentations in CLIM

The following functions can be used to examine or modify presentations.

`presentation` **[Class]**

■ The protocol class that corresponds to a presentation. If you want to create a new class that obeys the presentation protocol, it must be a subclass of `presentation`.

`standard-presentation` **[Class]**

■ The class that CLIM uses to represent presentations.

`presentationp` [Function]

Arguments: *`object`*

■ Returns `t` if and only if *`object`* is of type `presentation`.

`presentation-object` [Generic function]

Arguments: *`presentation`*

■ Returns the object represented by the presentation *`presentation`*. You can use **`setf`** on **`presentation-object`** to change the object associated with the presentation.

■ If you write your own class of presentation, it must implement or inherit a method for this generic function.

presentation-type **[Generic function]**

Arguments: *presentation*

- Returns the presentation type of the presentation *presentation*. You can use **setf** on **presentation-type** to change the presentation type associated with the presentation.
- If you write your own class of presentation, it must implement or inherit a method for this generic function.

8.4 Using CLIM presentation types for input

The primary means for getting input from the end user is **accept**. (Note that CLIM's command processor is built on top of **accept**, so the following material applies to user input to the command processor as well as to explicit calls to **accept**.) Characters typed in at the keyboard in response to a call to **accept** are parsed, and the application object they represent is returned to the calling function. (The parsing is done by the **accept** method for the presentation type.) Alternatively, if a presentation of the type specified by the **accept** call has previously been displayed, the user can click on it with the pointer and **accept** returns it directly (that is, no parsing is required).

Examples:

```
(clim:accept 'string) →
Enter a string: abracadabra
"abracadabra"
(clim:accept 'string) →
Enter a string [default abracadabra]: abracadabra
"abracadabra"
```

In the first call to **accept**, "abracadabra" was typed at the keyboard. In the second call to **accept**, the user clicked on the keyboard-entered string of the first function. In both cases, the string object "abracadabra" was returned.

In most circumstance, not every type of object is acceptable as input. Only an object of the presentation type specified in the current call to **accept** function (or one of its subtypes) can be input. In other words, the **accept** function establishes the current *input context*. For example, if the call to **accept** specifies an integer presentation type, only a typed-in or a displayed integer is acceptable. Numbers displayed as integer presentations would, in this input context, be sensitive, but those displayed as part of some other kind of presentation, such as a file pathname, would not. Thus, **accept** controls the input context and thereby the sensitivity of displayed presentations.

Clicking on a presentation of a type different from the input context may cause translation to an acceptable object. For example, you could make a presentation of a file pathname translate to an integer -- say, its length -- if you want. It is very common to translate to a command that operates on a presented object. For more information on presentation translators, see the section 8.7 **Presentation translators in CLIM**.

We say above that the range of acceptable input is, typically, restricted. How restricted is strictly up to you, the programmer. Using compound presentation types like `and` and `or`, and other predefined or specially devised presentation types gives you a high degree of flexibility and control over the input context.

8.4.1 CLIM operators for accepting input

CLIM provides the following top-level operators for accepting typed input. **with-input-context** is the most general operator, and **accept** and **accept-from-string** support common idioms.

`*input-context*` **[Variable]**

■ The current input context. This will be a list, each element of which corresponds to a single call to **with-input-context**. The first element of the list represents the context established by the most recent call to **with-input-context**, and the last element represents the context established by the least recent call to **with-input-context**.

■ The exact format of the elements in the list is unspecified, but the elements have dynamic extent.

input-context-type **[Function]**

Arguments: *context-entry*

■ Given one element from `*input-context*`, *context-entry*, this returns the presentation type of the context entry.

with-input-context **[Macro]**

Arguments: (*type* &key override) (&optional *object-var type-var event-var options-var*) *form* &body *clauses*

■ Establishes an input context of type *type*. You will rarely need to use this, since **accept** uses this to establish an input context for you.

When *override* is `nil` (the default), this invocation of **with-input-context** adds its context presentation type to the extant context. In this way an application can solicit more than one type of input at the same time.

When *override* is `t`, it overrides the current input context rather than nesting inside the current input context.

After establishing the new input context, *form* is evaluated. If no pointer gestures are made by the end user during the execution of *form*, all of the values of *form* are returned.

Otherwise, if the user invoked a translator by clicking on an object, one of the *clauses* is executed, based on the presentation type of the object returned by the translator. All of the values of that clause are returned as the values of **with-input-context**.

During the execution of one of the *clauses*, *object-var* is bound to the object returned by the translator, *type-var* is bound to the presentation type returned by the translator, and *event-var* is bound to the event corresponding to the user's gesture. *options-var* is bound to any options that the translator might have returned, and will be either `nil` or a list of keyword-value pairs. *clauses* is constructed like a **typecase** statement clause list whose keys are presentation types.

Note that, when one of the *clauses* is executed, nothing is inserted into the input buffer. If you want to insert input corresponding to the object the user clicked on, you must call **replace-input** or **presentation-replace-input**.

■ Only the arguments *type* and *override* are evaluated.

```
(clim:with-input-context ('pathname)
            (path)
   (read)
 (pathname
  (format t "The pathname ~A was clicked on." path)))
```

accept [Function]

Arguments: *type* &rest *accept-args* &key stream view default default-type history provide-default prompt prompt-mode display-default query-identifier activation-gestures additional-activation-gestures delimiter-gestures additional-delimiter-gestures insert-default (replace-input t) present-p (active-p t)

■ Requests input of the *type* from the *stream*. **accept** returns two values (or three values when called inside of **accepting-values**), the object and its presentation type.

accept works by displaying a prompt, establishing an input context via **with-input-context**, and then calling the **accept** presentation method for *type* and *view*.

Note that **accept** does not insert newlines. If you want to put prompts on separate lines, use **terpri**.

■ The arguments behave as follows:

type

A presentation type specifier that indicates what type to read. *type* may be an presentation type abbreviation. See the section 8.2 **How to specify a CLIM presentation type**.

stream

Specifies the input stream. The defaults is *standard-input*.

view

An object representing a view. The default is (stream-default-view stream). For most streams, the default view is the textual view, +textual-view+. Under Allegro CLIM, the default view inside of a dialog is the indirect view +gadget-dialog-view+.

default

Specifies the object to be used as the default value for this call to **accept**. If this keyword is not supplied and *provide-default* is t, then the default is determined by taking the most recent item from the presentation type history specified by the *history* argument. If *default* is supplied and the input provided by the user is empty, then *default* and *default-type* are returned as the two values from **accept**.

default-type

If *default* is supplied and the input provided by the user is empty, then *default* and *default-type* are returned as the two values from **accept**. This defaults to *type*.

history

Specifies which presentation type's history to use for the input editor's yanking commands. The default is to use the history for *type*. If *history* is nil, no history is used. It is almost never necessary to provide this.

provide-default

Specifies whether or not to provide a default value for this call to **accept** if *default* is not supplied. The default is nil.

prompt

Controls how **accept** will prompt the user. If *prompt* is t (the default), the prompt is a description of the type. If *prompt* is nil, prompting is suppressed. If it is a string, the string is displayed as the prompt. The default is t, which produces "Enter a *type*:" in a top-level **accept** or "(*type*)" in a nested **accept**.

prompt-mode

Can be `:normal` (which is the default) or `:raw`, which suppresses putting a colon after the prompt and/or default in a top-level **`accept`** and suppresses putting parentheses around the prompt and/or default in a nested **`accept`**. It is sometimes useful to use `:prompt-mode :raw` when writing complicated **`accept`** methods that recursively call **`accept`**.

display-default

When true, displays the default as part of the prompt, if one was supplied. When `nil`, the default is not displayed. *`display-default`* defaults to `t` if there was a prompt, otherwise it defaults to `nil`.

query-identifier

This option is used to supply a unique identifier for each call to **`accept`** inside **`accepting-values`**. If it is not supplied, it defaults to and value generated from *`type`* and *`prompt`*. It is generally best to explicitly use a `:query-identifier` inside of **`accepting-values`**, because doing so can prevent bugs of omission, particularly when type may change between calls to **`accept`** within **`accepting-values`**.

activation-gestures

A list of gestures that overrides the current activation gestures, which terminate input. See the section 17.1 **Input editing and built-in keystroke commands in CLIM**. Generally, `:activation-gestures` and `:additional-activation-gestures` are only used inside complex **`accept`** methods for textual views that read multiple fields separated by some delimiter.

additional-activation-gestures

A list of gestures that add to the activation gestures without overriding the current ones.

delimiter-gestures

A list of gestures that overrides the current delimiter gestures, which terminate an individual token but not the entire input sentence. See the section 17.1 **Input editing and built-in keystroke commands in CLIM**. You will rarely need to use this.

additional-delimiter-gestures

A list of gestures that add to the delimiter gestures without overriding the current ones.

insert-default

When true, inserts the default into the input buffer before getting input from the user. The default for *`insert-default`* is `nil`.

replace-input

Controls whether input gotten by clicking the pointer should be inserted back into the input buffer. The default is `t`. You will rarely need to use this.

active-p

Controls whether a call to **`accept`** within a dialog should produce an active field (that is, one available for input). The default is `t`. Use this when you want to deactivate some field in a dialog. Usually, the deactivated field will be grayed over.

present-p

is reserved for internal use by CLIM.

`accept-from-string` [Function]

Arguments: *`type string`* `&key view default default-type activation-gestures additional-activation-gestures`

```
delimiter-gestures additional-delimiter-gestures
(start 0) end
```

■ Reads the printed representation of an object of type *type* from *string*. This function is like **accept**, except that the input is taken from *string*, starting at the position *start* and ending at *end*. *view*, *default*, and *default-type* are as in **accept**.

■ **accept-from-string** returns three values: the object, its presentation type, and the index in *string* of the next character after the input.

■ If *default* is supplied, then the *default* and the *default-type* are returned if the input string is empty.

The remaining arguments, *activation-gestures*, *additional-activation-gestures*, *delimiter-gestures*, and *additional-delimiter-gestures*, are as for **accept**.

8.5 Predefined presentation types in CLIM

This section documents predefined CLIM presentation types, presentation type options, and parameters. For more information on how to use these presentation types, see the section 8.2 **How to specify a CLIM presentation type**.

Note that any presentation type with the same name as a Common Lisp type accepts the same parameters as the Common Lisp type (and additional parameters in a few cases).

8.5.1 Basic presentation types in CLIM

Here are basic presentation types that correspond to the Common Lisp built-in types having the same name.

`t` **[Presentation type]**

■ The supertype of all other presentation types. This type a default method for **accept** that allows only input via the pointer, and a default method for **present** that uses **write** to display the object.

`null` **[Presentation type]**

■ The presentation type that represents nothing. The single object associated with this type is `nil`, and its printed representation is "None".

`boolean` **[Presentation type]**

■ The presentation type that represents `t` or `nil`. The textual representation is "Yes" and "No", respectively.

`symbol` **[Presentation type]**

■ The presentation type that represents a symbol. Its **accept** method reads the name of a symbol, and its **present** method displays the symbol's name.

`keyword` **[Presentation type]**

■ The presentation type that represents a symbol in the keyword package. It is a subtype of `symbol`.

8.5.2 Numeric presentation types in CLIM

The following presentation types represent the Common Lisp numeric types having the same names.

`number` **[Presentation type]**

- The presentation type that represents a general number. It is the supertype of all the number types.

`complex` **[Presentation type]**

Arguments: `&optional` *`type`*

- The presentation type that represents a complex number. It is a subtype of `number`.

`type` is the type to be used for each component of the real number. If unspecified, it defaults to `real`.

`real` **[Presentation type]**

Arguments: `&optional` *`low high`*

- The presentation type that represents either a ratio, an integer, or a floating point number between *`low`* and *`high`*. *`low`* and *`high`* can be inclusive or exclusive; they are specified the same way as in the Common Lisp type specifiers for `real`.
- Options to this type are *`base`* (default is 10) and *`radix`* (default is `nil`), which control the value of `*print-base*` and `*print-radix*` when the number is printed (or `*read-base*` when the number is read).

`rational` **[Presentation type]**

- The presentation type that represents either a ratio or an integer between *`low`* and *`high`*. Options to this type are *`base`* and *`radix`*, which are the same as for the `real` type. It is a subtype of `real`.

`ratio` **[Presentation type]**

Arguments: `&optional` *`low high`*

- The presentation type that represents a ratio between *`low`* and *`high`*. Options to this type are *`base`* and *`radix`*, which are the same as for the `integer` type. It is a subtype of `rational`.

`integer` **[Presentation type]**

Arguments: `&optional` *`low high`*

- The presentation type that represents an integer between *`low`* and *`high`*. Options to this type are *`base`* and *`radix`*, which are the same as for the `real` type. It is a subtype of `rational`.

`float` **[Presentation type]**

Arguments: `&optional` *`low high`*

- The presentation type that represents a floating point number between low and high. This type is a subtype of `real`.

8.5.3 Character and string presentation types in CLIM

These two presentation types can be used for reading and writing character and strings.

`character` **[Presentation type]**

- The presentation type that represents a Common Lisp character object.

`string` **[Presentation type]**

Arguments: `&optional` *`length`*

■ The presentation type that represents a string. If *`length`* is supplied, the string must have exactly that many characters.

8.5.4 Pathname presentation type in CLIM

`pathname` **[Presentation type]**

■ The presentation type that represents a pathname.

■ The options are *`default-type`*, which defaults to `nil`, *`default-version`*, which defaults to `:newest`, and *`merge-default`*, which defaults to `t`. If *`merge-default`* is `nil`, **`accept`** returns the exact pathname that was entered, otherwise **`accept`** merges against the default provided to **`accept`** and *`default-type`* and *`default-version`*, using **`merge-pathnames`**. If no default is supplied, it defaults to `*default-pathname-defaults*`.

8.5.5 One-of and some-of presentation types in CLIM

The one-of and some-of presentation types can be used to accept and present one or more items from a set of items. The set of items can be specified as a rest argument, a sequence, or an alist.

This table summarizes single (one-of) and multiple (some-of) selection presentation types. Each row represents a type of presentation. Columns contain the associated single and multiple selection presentation types.

Args	Single	Multiple
most general	`clim:completion`	`clim:subset-completion`
`&rest elements`	`member`	`clim:subset`
`sequence`	`clim:member-sequence`	`clim:subset-sequence`
`alist`	`clim:member-alist`	`clim:subset-alist`

`completion` **[Presentation type]**

Arguments: *`sequence`* `&key test value-key`

■ The presentation type that selects one from a finite set of possibilities, with completion of partial inputs. Several types are implemented in terms of the `completion` type, including `token-or-type`, `null-or-type`, `member`, `member-sequence`, and `member-alist`.

■ The presentation type parameters are:

`sequence`

A list or vector whose elements are the possibilities. Each possibility has a printed representation (given by *`name-key`*), called its name, and an internal representation (given by *`value-key`*), called its value. **`accept`** reads a name and returns a value. **`present`** is given a value and outputs a name.

`test`

A function that compares two values for equality. The default is **`eql`**.

value-key

A function that returns a value given an element of *sequence*. The default is **identity**.

■ The following presentation type options are available:

name-key

A function that returns a name, as a string, given an element of *sequence*. The default is a function that behaves as follows:

Argument	Returned Value
string	the string
null	nil
cons	**string** of the **car**
symbol	**string-capitalize** of its name
otherwise	**princ-to-string** of it

documentation-key

A function that returns nil or a descriptive string, given an element of *sequence*. The default always returns nil.

partial-completers

A possibly-empty list of characters that delimit portions of a name that can be completed separately. The default is a list of one character, Space.

printer

A function of two arguments that is used to display the name of the item. The default is **write-token**.

highlighter

A function of three arguments that is used to highlight the currently selected item in a dialog. The first argument is a continuation function that should be called on the other two arguments, the object and the stream. The default method for textual dialogs simply displays the currently selected item in boldface.

member-sequence **[Presentation type abbreviation]**

Arguments: *sequence* &key test

■ Like member, except that the set of possibilities is the sequence *sequence*. The parameter *test* and the options are the same as for completion.

member-alist **[Presentation type abbreviation]**

Arguments: *alist* &key test

■ Like member, except that the set of possibilities is the alist *alist*. Each element of *alist* is either an atom as in member-sequence or a list whose **car** is the name of that possibility and whose **cdr** is one of the following:

- The value (which must not be a cons)

- A list of one element, the value
- A property list containing one or more of the following properties:

  `:value` -- the value

  `:documentation` -- a descriptive string

■ The *`test`* parameter and the options are the same as for `completion` except that *`value-key`* and *`documentation-key`* default to functions that support the specified alist format.

`subset-completion` **[Presentation type]**

Arguments: *`sequence`* `&key test value-key`

■ The presentation type that selects one or more from a finite set of possibilities, with completion of partial inputs. The parameters and options are the same as for `completion` with the following additional options:

`separator`

The character that separates members of the set of possibilities in the printed representation when there is more than one. The default is comma.

`echo-space`

(`t` or `nil`) Whether to insert a space automatically after the separator. The default is `t`.

The other subset types (`subset`, `subset-sequence`, and `subset-alist`) are implemented in terms of the `subset-completion` type.

`subset` **[Presentation type abbreviation]**

Arguments: `&rest` *`elements`*

■ The presentation type that specifies a subset of *`elements`*. Values of this type are lists of zero or more values chosen from the possibilities in *`elements`*. The printed representation is the names of the elements separated by the separator character. The options are the same as for `subset-completion`.

`subset-sequence` **[Presentation type abbreviation]**

Arguments: *`sequence`* `&key test`

■ Like `subset`, except that the set of possibilities is the sequence *`sequence`*. The parameter *`test`* and the options are the same as for `subset-completion`.

`subset-alist` **[Presentation type abbreviation]**

Arguments: *`alist`* `&key test`

■ Like `subset`, except that the set of possibilities is the alist *`alist`*. The parameter *`test`* and the options are the same as for `subset-completion`. The parameter *`alist`* has the same format as `member-alist`.

8.5.6 Sequence presentation types in CLIM

The following two presentation types can be used to accept and present a sequence of objects.

`sequence` **[Presentation type]**

Arguments: *element-type*

■ The presentation type that represents a sequence of elements of type *element-type*. The printed representation of a `sequence` type is the elements separated by the separator character. It is unspecified whether **accept** returns a list or a vector. You can supply the following options:

separator

The character that separates members of the set of possibilities in the printed representation when there is more than one. The default is comma, `#\,`.

echo-space

If this is `t`, then CLIM will insert a space automatically after the separator, otherwise it will not. The default is `t`.

■ *element-type* can be a presentation type abbreviation.

■ For example, you can read a collection of real numbers by calling **accept** on the type `(sequence real)`.

`sequence-enumerated` **[Presentation type]**

Arguments: `&rest` *element-types*

■ `sequence-enumerated` is like `sequence`, except that the type of each element in the sequence is individually specified. It is unspecified whether **accept** returns a list or a vector. You can supply the following options:

separator

The character that separates members of the set of possibilities in the printed representation when there is more than one. The default is comma, `#\,`.

echo-space

If this is `t`, then CLIM will insert a space automatically after the separator, otherwise it will not. The default is `t`.

■ The elements of *element-types* can be presentation type abbreviations.

■ For example, you might read a 2-dimensional coordinate by calling **accept** on the type `(sequence-enumerated real real)`.

8.5.7 Meta presentation types in CLIM

CLIM provides the `or` and `and` presentation types in order to combine presentation types.

`or` **[Presentation type]**

Arguments: `&rest` *types*

■ The presentation type that is used to specify one of several types, for example, `(or (member :all :none) integer)`. **accept** returns one of the possible types as its second value, not the original `or` presentation type specifier.

The elements of *types* can be presentation type abbreviations.

`and` **[Presentation type]**

Arguments: `&rest` *types*

■ The presentation type that is used for multiple inheritance. `and` is usually used in conjunction with `satisfies`. For example,

```
(and integer (satisfies oddp))
```

■ The elements of *types* can be presentation type abbreviations.

Note that the first type in *types* is in charge of accepting and presenting. The remaining elements of *types* are used for type checking (for example, filtering applicability of presentation translators).

■ The `and` type has special syntax that supports the two predicates, `satisfies` and `not`. `satisfies` and `not` cannot stand alone as presentation types and cannot be first in *types*. `not` can surround either `satisfies` or a presentation type.

8.5.8 Compound presentation types in CLIM

The following compound presentation types are provided because they implement some common idioms.

`token-or-type` **[Presentation type abbreviation]**

Arguments: *tokens type*

■ A compound type that is used to select one of a set of special tokens, or an object of type *type*. *tokens* is anything that can be used as the *alist* parameter to `member-alist`; typically it is a list of keyword symbols.

type can be a presentation type abbreviation.

`null-or-type` **[Presentation type abbreviation]**

Arguments: *type*

■ A compound type that is used to select `nil`, whose printed representation is the special token "None", or an object of type *type*.

type can be a presentation type abbreviation.

`type-or-string` **[Presentation type abbreviation]**

Arguments: *type*

■ A compound type that is used to select an object of type *type* or an arbitrary string, for example, `(type-or-string integer)`. Any input that **accept** cannot parse as the representation of an object of type *type* is returned as a string.

type can be a presentation type abbreviation.

8.5.9 Lisp form presentation types in CLIM

The Lisp form presentation types are complex types provided primarily for use by the top level interactor of an application. There are also presentation types defined for commands, which are discussed in section 10.7 **Command-related presentation types**.

`expression` **[Presentation type]**

■ The presentation type used to represent any Lisp object. The textual view of this type looks like what the standard **prin1** and **read** functions produce and accept.

This type has one option, *auto-activate*, which controls whether the expression terminates on a delimiter gestures, or when the Lisp expression balances (for example, you type enough right parentheses to complete the expression). The default for *auto-activate* is `nil`.

`form` **[Presentation type]**

■ The presentation type used to represent a Lisp form. This type is a subtype of `expression`. It also has an *`auto-activate`* option.

8.6 Defining a new presentation type in CLIM

In many cases, CLIM's built-in presentation types are not sufficient for your applications. This section describes how to define a new presentation type and the methods for that type.

8.6.1 Concepts of defining a new presentation type in CLIM

CLIM's standard set of presentation types will be useful in many cases, but most applications will need customized presentation types to represent the objects modeled in the application.

By defining a presentation type, you define all of the user interface components of the entity:

- A displayed representation, for example, textual or graphical
- An optional textual representation, for user input via the keyboard (this is required if the application uses command-line input for this type)
- Pointer sensitivity, for user input via the pointer

In other words, by defining a presentation type, you describe in one place all the information about an object necessary to display it to the user and interact with the user for getting input.

The set of presentation types forms a type lattice, an extension of the Common Lisp CLOS type lattice. When a new presentation type is defined as a subtype of another presentation type, it inherits all the attributes of the supertype except those explicitly overridden in the definition.

To define a new presentation type, you follow these steps:

1. Use the **`define-presentation-type`** macro.
   - Name the new presentation type.
   - Supply parameters that further restrict the type (if appropriate).
   - Supply options that affect the appearance of the type (if appropriate).
   - State the supertypes of this type, to make use of inheritance (if appropriate).
2. Define CLIM presentation methods.
   - Specify how objects are displayed with a **`present`** presentation method. (You must define a **`present`** method, unless the new presentation type inherits a method that is appropriate for it.)
   - Specify how objects are parsed with a **`accept`** presentation method. (In most cases, you must define a **`accept`** method, unless the new presentation type inherits a method that is appropriate for it. If it is never necessary to enter the object by typing its representation on the keyboard, you don't need to provide this method.)
   - Specify the type/subtype relationships of this type and its related types, if necessary, with **`presentation-typep`** and **`presentation-subtypep`** presentation methods. (You must define or inherit these methods if either the presentation type is not a CLOS class, or the type has parameters.)

8.6.2 CLIM presentation type Inheritance

Every presentation type is associated with a CLOS class. In the common case, the *name* of the presentation type is a class object or the name of a class, and that class is not a `built-in-class`. In this case, the presentation type is associated with that CLOS class.

Otherwise, **`define-presentation-type`** defines a class with metaclass `presentation-type-class` and superclasses determined by the presentation type definition. This class is not named *name*, since that could interfere with built-in Common Lisp types such as `and`, `member`, and `integer`. **`class-name`** of this class returns a list `(presentation-type` *name*`)`. `presentation-type-class` is a subclass of `standard-class`.

> **Note**: If the same name is defined with both **`defclass`** (or **`defstruct`**) and **`define-presentation-type`**, the **`defclass`** (or **`defstruct`**) must be evaluated first.

Every CLOS class (except for built-in classes) is a presentation type, as is its name. If it has not been defined with **`define-presentation-type`**, it allows no parameters and no options. As in CLOS, inheriting from a built-in class does not work, unless you specify the same inheritance that the built-in class already has; you may want to do this in order to add presentation type parameters to a built-in class.

If you define a presentation type that does not have the same name as a CLOS class, you must define a **`presentation-typep`** presentation method for it. If you define a presentation type that has parameters, you must define a **`presentation-subtypep`** for it.

If your presentation type has the same name as a class, doesn't have any parameters or options, doesn't have a history, and doesn't need a special description, you do not need to call **`define-presentation-type`**.

During method combination, presentation type inheritance is used both to inherit methods (‘what parser should be used for this type?’), and to establish the semantics for the type (‘what objects are sensitive in this context?’). Inheritance of methods is the same as in CLOS and thus depends only on the type name, not on the parameters and options.

Presentation type inheritance translates the parameters of the subtype into a new set of parameters for the supertype, and translates the options of the subtype into a new set of options for the supertype.

8.6.3 Examples of defining a new CLIM presentation type

This section has some examples of defining new presentation type. The first subsection contains a lengthy example which works through the details of part of a small application. The second subsection contains other, miscellaneous examples that cover some more advanced topics.

8.6.4 Example of modelling courses at a university

This example shows how to define a new presentation type, and how to define the presentation methods for the new type. First we define the application objects themselves and create some test data. Then we define a simple presentation type, and gradually add enhancements to it to show different CLIM techniques.

This example models a university. The application objects are students, courses, and departments. This is such a simple example that there is no need to use inheritance.

Note that this example must be run in a package, such as `clim-user`, that has access to symbols from the `clim` and `clos` packages.

These are the definitions of the application objects:

```
(defclass student ()
  ((name :reader student-name :initarg :name)
   (courses :accessor student-courses :initform nil)))

(defclass course ()
  ((name :reader course-title :initarg :title)
   (department :reader course-department :initarg :department)))

(defclass department ()
  ((name :reader department-name :initarg :name)))
```

The following code provides support for looking up objects by name.

```
(defvar *student-table* (make-hash-table :test #'equal))
(defvar *course-table* (make-hash-table :test #'equal))
(defvar *department-table* (make-hash-table :test #'equal))
(defun find-student (name &optional (errorp t))
 (or (gethash name *student-table*)
   (and errorp (error "There is no student named ~S" name))))
(defun find-course (name &optional (errorp t))
 (or (gethash name *course-table*)
   (and errorp (error "There is no course named ~S" name))))

(defun find-department (name &optional (errorp t))
 (or (gethash name *department-table*)
   (and errorp (error "There is no department named ~S" name))))

(defmethod initialize-instance :after ((student student) &key)
 (setf (gethash (student-name student) *student-table*) student))

(defmethod initialize-instance :after ((course course) &key)
 (setf (gethash (course-title course) *course-table*) course))

(defmethod initialize-instance :after ((department department) &key)
 (setf (gethash (department-name department) *department-table*) department))

(defmethod print-object ((student student) stream)
 (print-unreadable-object (student stream :type t)
  (write-string (student-name student) stream)))

(defmethod print-object ((course course) stream)
 (print-unreadable-object (course stream :type t)
  (write-string (course-title course) stream))
 (format stream " (~A)" (department-name (course-department course))))

(defmethod print-object ((department department) stream)
 (print-unreadable-object (department stream :type t)
  (write-string (department-name department) stream)))
```

Here we create some test data:

```
(flet ((make-student (name &rest courses)
  (setf (student-courses (make-instance 'student :name name))
```

```
        (copy-list courses)))
    (make-course (title department)
     (make-instance 'course :title title :department department))
    (make-department (name)
     (make-instance 'department :name name)))
 (let* ((english (make-department "English"))
     (physics (make-department "Physics"))
     (agriculture (make-department "Agriculture"))
     (englit (make-course "English Literature" english))
     (mabinogion (make-course "Deconstructing the Mabinogion" english))
     (e+m (make-course "Electricity and Magnetism II" physics))
     (beans (make-course "The Cultivation and Uses of Beans" agriculture))
     (horses (make-course "Horse Breeding for Track and Field" agriculture))
     (corn (make-course "Introduction to Hybrid Corn" agriculture)))
  (make-student "Susan Charnas" englit e+m)
  (make-student "Orson Card" englit beans)
  (make-student "Roberta MacAvoy" horses mabinogion)
  (make-student "Philip Farmer" corn beans horses)))
```

You can evaluate the following forms to test what you have done so far. A printed representation of each object will be displayed.

```
(find-student "Philip Farmer")
→ #<STUDENT Philip Farmer>
 (find-course "The Cultivation and Uses of Beans")
→ #<COURSE The Cultivation and Uses of Beans (Agriculture)>
(find-department "Agriculture")
→ #<DEPARTMENT Agriculture
```

If you try to evaluate a form that has not yet been defined (for example, if you try to look up a student that doesn't exist), you might see something like this:

```
CLIM-USER(12): (find-student "Jill Parker")
Error: There is no student named "Jill Parker"
CLIM-USER(13):
```

Now we are ready to develop a user interface. This first example defines presentations of students, represented by their names. This simple presentation type does not provide parameters or options. A real program would also provide presentation types for courses and departments, but this example shows students only.

```
(clim:define-presentation-type student ())
(clim:define-presentation-method clim:present
                 (student (type student) stream
                 (view clim:textual-view) &key)
 (write-string (student-name student) stream))
(clim:define-presentation-method clim:accept
                 ((type student) stream
                 (view clim:textual-view) &key)
 (let* ((token (clim:read-token stream))
     (student (find-student token nil)))
  (when student
   (return-from clim:accept student))
  (clim:input-not-of-required-type token type)))
```

Test this by evaluating the following forms in a CLIM Lisp Listener (which is part of the CLIM Demos system). Note that there is no completion and **`find-student`** is case-sensitive, so the student's name must be entered exactly to be accepted.

```
(clim:describe-presentation-type 'student t)
(clim:describe-presentation-type 'student t 5)
(clim:present (find-student "Philip Farmer") 'student)
(clim:accept 'student :default (find-student "Philip Farmer"))
```

We can improve the input interface by using completion over elements of `*student-table*`.

```
(clim:define-presentation-method clim:accept
                 ((type student) stream
                 (view textual-view) &key)
 (values        ;suppress values after the first
  (clim:completing-from-suggestions
    (stream :partial-completers '(#\Space))
   ;; SUGGEST takes arg of name, object
   (maphash #'clim:suggest *student-table*))))
```

Test this by evaluating the following form in a CLIM Lisp Listener. (Use the CLIM Lisp Listener demo for this purpose.)

```
(clim:accept 'student :default (find-student "Philip Farmer"))
```

Try the Control-? key, and try entering just the initials of a student, separated by a space; they complete to the full name.

It would be useful to be able to select students in a particular department. We can revise the presentation type for `student` by adding a parameter for the department. A student is in a department if the student is taking any course in that department.

```
(defun student-in-department-p (student department)
 (find department (student-courses student) :key #'course-department))
(clim:define-presentation-type student (&optional department))
```

When a presentation type has parameters, the defaults for the **`presentation-typep`** and **`presentation-subtypep`** presentation methods are not sufficient. Therefore, we need to define these presentation methods. We also define a new **`describe-presentation-type`** method.

```
(clim:define-presentation-method clim:presentation-typep
                 (object (type student))
 (or (eq department '*)
   (student-in-department-p object department)))

(clim:define-presentation-method clim:presentation-subtypep
                 ((type1 student) type2)
 (let ((department1 (clim:with-presentation-type-parameters
            (student type1) department))
    (department2 (clim:with-presentation-type-parameters
            (student type2) department)))
  (values (or (eq department1 department2)
        (eq department2 '*))
      t)))

(clim:define-presentation-method clim:describe-presentation-type
```

```
                ((type student) stream plural-count)
 (when (eql plural-count 1)
  (write-string (if (or (eq department '*)
             (not (find (char (department-name department) 0) "aeiou"
                   :test #'char-equal)))
           "a "
           "an ")
         stream))
 (format stream
   (if (and (integerp plural-count) (\> plural-count 1))
     (if (eq department '*) "~R student~:P" "~R ~A student~:*~:P")
     (if (eq department '*) "student~P" "~*~A student~:*~:P"))
  (typecase plural-count
   (integer plural-count)
   (null 1)
   (otherwise 2))
  (unless (eq department '*)
   (department-name department))))
```

Evaluate the following forms to test these methods:

```
(clim:presentation-typep (find-student "Philip Farmer")
             `(student ,(find-department "Agriculture")))

(clim:presentation-typep (find-student "Philip Farmer")
             `(student ,(find-department "English")))

(clim:presentation-typep "Philip Farmer"
             `(student ,(find-department "Agriculture")))

(clim:presentation-subtypep `(student ,(find-department "Agriculture"))
              `(student *))

(clim:presentation-subtypep `(student ,(find-department "Agriculture"))
              `(student ,(find-department "English")))

(clim:describe-presentation-type `(student ,(find-department "Physics")))

(clim:describe-presentation-type `(student *))
```

The existing method for **`accept`** suggests all the students, even the ones in the wrong department, so we provide the following `:around` method to check that we are returning a student in the right department.

```
(clim:define-presentation-method clim:accept :around
                ((type student) stream view &key)
 (declare (ignore stream view))
 (multiple-value-bind (object actual-type)
   (call-next-method)
  (unless (clim:presentation-typep object type)
   (clim:input-not-of-required-type object type))
  (values object actual-type)))
```

Evaluate the following form in a CLIM Listener before and after defining the above method.

```
(clim:accept `(student ,(find-department "Agriculture")))
```

Type the following at the prompt:

```
susan RETURN
```

Before defining the above method, **accept** returns a student that is not in the specified department. After defining the above method, CLIM asks the user to try again. But if you press Control-?, Susan Charnas is still listed as one of the possibilities.

Another way to do this would be to filter out students in other departments before calling **suggest**. To do that, define this method instead of the preceding method. This way works better because the completion possibilities won't include any extra students.

```
(clim:define-presentation-method clim:accept
                 ((type student) stream
                 (view clim:textual-view) &key)
 (values           ;suppress values after the first
  (clim:completing-from-suggestions
    (stream :partial-completers '(#\Space))
   (maphash (if (eq department '*)
          #'clim:suggest
          #'(lambda (name student)
            (when (student-in-department-p student department)
             (clim:suggest name student))))
        *student-table*))))

(fmakunbound '(method clim:accept-method :around (student t t t t t)))
```

Evaluate these forms in the CLIM Listener again. Try entering the names of agricultural and non-agricultural students.

```
(clim:accept `(student ,(find-department "Agriculture")))
(clim:accept `student)
```

You can also try the Control-? key.

It is easy to define an abbreviation for a presentation type. Here we define `aggie` as an abbreviation for a student in the Agriculture department:

```
(clim:define-presentation-type-abbreviation aggie ()
 `(student ,(find-department "Agriculture")))
```

Evaluate these forms to test it.

```
(clim:describe-presentation-type 'aggie)
(clim:accept 'aggie)
```

Now we refine our example by providing an option that controls the printing of the student's name.

```
(clim:define-presentation-type student (&optional department)
 :options (last-name-first))
(clim:define-presentation-method clim:present
                 (student (type student) stream
                 (view clim:textual-view) &key)
 (let* ((name (student-name student))
     (index (and last-name-first (position #\Space name :from-end t))))
  (cond ((null index)
      (write-string name stream))
```

```
     (t
      (write-string name stream :start (1+ index))
      (write-string ", " stream)
      (write-string name stream :end index)))))

(clim:define-presentation-method clim:accept
                 ((type student) stream
                 (view clim:textual-view) &key)
 (values            ;suppress values after the first
  (clim:completing-from-suggestions
    (stream :partial-completers '(#\Space #,))
   (maphash #'(lambda (name student)
          (when (clim:presentation-typep student type)
           (clim:suggest
            (or (and last-name-first
                (let ((index (position #\Space name
                           :from-end t)))
                 (and index
                    (concatenate 'string
                     (subseq name (1+ index))
                     ", "
                     (subseq name 0 index)))))
              name)
             student)))
        *student-table*))))
```

Evaluate these forms to test it.

```
(clim:present (find-student "Philip Farmer") 'student)
(clim:present (find-student "Philip Farmer") '((student) :last-name-first t))
(clim:accept `((student) :last-name-first t))
(clim:accept `((student ,(find-department "Physics")) :last-name-first t))
```

Since presentation type options are not automatically inherited by subtypes and abbreviations, the following example doesn't work.

```
(clim:accept '((aggie) :last-name-first t))
```

This example works if you redefine `aggie` to accept the `:last-name-first` option:

```
(clim:define-presentation-type-abbreviation aggie ()
 `((student ,(find-department "Agriculture"))
  :last-name-first ,last-name-first)
 :options (last-name-first))
```

Note that you can override the presentation type's description:

```
(clim:accept `((student ,(find-department "English"))
       :description "English major"))
```

8.6.5 Examples of more complex presentation types

Here is an example of a complex presentation type that consists of one field optionally followed by another. You should pay particular attention to the used of **`with-delimiter-gestures`**, and the way this type reads the delimiter gesture out of the buffer. This example can be used as the basis for many presentation types that read multiple fields.

```
;; In a real application, this would probably be something else...
(defvar *printer-name-alist*
    '(("Laserwriter" laserwriter)
     ("Lautscribner" lautscribner)))

(defvar *destination-type-alist*
    '((:window :value (:window nil nil nil))
     (:pathname :value (:pathname pathname "pathname" "Enter a pathname"))
     (:printer :value (:printer (clim:member-alist ,*printer-name-alist*)
              "printer" "Enter the name of a printer"))))

;; OUTPUT-DESTINATION is a presentation type whose printed representation
;; consists of the type of output destination (Window, File, Printer)
;; followed by another argument (a pathname or printer name). The two
;; fields are separated by a space.
(clim:define-presentation-type output-destination ())

(clim:define-presentation-method clim:accept
  ((type output-destination) stream (view clim:textual-view) &key)
 ;; Since #\Space separates the fields, make it a delimiter gesture
 ;; so that the calls to ACCEPT will terminate when the user types
 ;; a space character.
 (clim:with-delimiter-gestures (#\Space)
  (let (dtype place delimiter)
   ;; Read the destination type using ACCEPT. Establish a "help"
   ;; context to prompt the user to enter a destination type.
   (clim:with-accept-help
     ((:subhelp #'(lambda (stream action string)
             (declare (ignore action string))
             (write-string "Enter the destination type." stream))))
    (setq dtype (clim:accept '(clim:member-alist ,*destination-type-alist*)
                 :stream stream :view view :prompt "type")))
   (destructuring-bind (dtype type prompt help) dtype
    (when (eql type nil)
     (return-from clim:accept (list dtype nil)))
    ;; Read the delimiter -- it should be a space, but if it is not,
    ;; signal a parse-error.
    (setq delimiter (clim:stream-peek-char stream))
    (cond ((char-equal delimiter #\Space)
        ;; The delimiter was a space, so remove it from the input
        ;; buffer and read the next integer.
        (clim:stream-read-char stream)
        (clim:with-accept-help
          ((:subhelp #'(lambda (stream action string)
                 (declare (ignore action string))
                 (write-string help stream))))
```

```
        (setq place (clim:accept type
                     :stream stream :view view :prompt prompt))))
       (t (simple-parse-error "Invalid delimiter: ~S" delimiter)))
    ;; Return the result, leaving the final delimiter in place.
    (list dtype place)))))

(define-presentation-method clim:present
  (object (type output-destination) stream (view clim:textual-view) &key)
 ;; Just print the two parts of the object separated by a space.
 (destructuring-bind (dtype place) object
  (if place
    (format stream "~:(~A~) ~A" dtype place)
    (format stream "~:(~A~)" dtype))))

;; Only lists whose two elements are right
(define-presentation-method clim:presentation-typep (object (type output-
destination))
 (and (listp object)
    (= (length object) 2)
    (not (null (assoc (first object) *destination-type-alist*)))))
```

8.6.6 CLIM operators for defining new presentation types

This section describes the forms used for defining a new presentation type and its methods.

`define-presentation-type` **[Macro]**

Arguments: *`name parameters`* `&key options inherit-from description history parameters-are-types`

- Defines a CLIM presentation type. The arguments are as follows:

  `name`

  The name of the presentation type. *`name`* must be a symbol or a CLOS class object.

  `parameters`

  Parameters of the presentation type. These parameters are lexically visible within *`inherit-from`* and within the methods created with **`define-presentation-method`**. For example, the parameters are used by **`presentation-typep`** to refine its tests for type inclusion.

  `parameters` are specified the same way as they are for **`deftype`**.

  `options`

  A list of option specifiers, which defaults to `nil`. An option specifier is either a symbol or a list of the form (*`symbol`* `&optional` *`default supplied-p presentation-type accept-options`*).

  `symbol`, *`default`*, and *`supplied-p`* are as in a normal lambda-list. If *`presentation-type`* and *`accept-options`* are present, they specify how to accept a new value for this option from the user. *`symbol`* can also be specified in the (*`keyword variable`*) form allowed for Common Lisp lambda lists. *`symbol`* is a variable that is visible within *`inherit-from`* and within most of the methods created with **`define-presentation-method`**. The keyword corresponding to *`symbol`* can be used as an option in the third form of a presentation type specifier. An option specifier for the standard option *`description`* is automatically added to *`options`* if an option with that keyword is not present.

inherit-from

A form that evaluates to a presentation type specifier for another type from which the new type inherits. *inherit-from* can access the parameter variables bound by the *parameters* lambda list and the option variables specified by *options*. If *name* is or names a CLOS class, then *inherit-from* must specify the class's direct superclasses (using `and` to specify multiple inheritance). It is useful to do this when you want to parameterize previously defined CLOS classes.

If *inherit-from* is unsupplied, it defaults as follows: if *name* is or names a CLOS class, then the type inherits from the presentation type corresponding to the direct superclasses of that CLOS class (using `and` to specify multiple inheritance). Otherwise, the type inherits from `standard-object`.

Note: you cannot use **`define-presentation-type`** to create a new subclass any of the built-in types, such as `integer` or `symbol`.

history

Specifies what history to use for the presentation type. A presentation type's history is what **`accept`** uses to generate a default when none is supplied. *history* can be one of the following:

`nil`

(the default) Uses no history.

`t`

Uses its own history.

type-name

Uses *type-name*'s history.

If you want more flexibility, you can define a **`presentation-type-history`** presentation method.

description

A string or `nil`. If `nil` or unsupplied, a description is automatically generated; it will be a prettied up version of the type name. For example, `small-integer` would become "small integer". You can also write a **`describe-presentation-type`** presentation method.

parameters-are-types

If all of the parameters to the presentation type are themselves types (as is the case for `and` and `or`), you should supply `:parameters-are-types` t.

■ Unsupplied optional or keyword parameters default to `*` (as in **`deftype`**) if no default is supplied in *parameters*. Unsupplied options default to `nil` if no default is supplied in *options*.

■ There are certain restrictions on the *inherit-from* form, to allow it to be analyzed at compile time. The form must be a simple substitution of parameters and options into positions in a fixed framework. It cannot involve conditionals or computations that depend on valid values for the parameters or options; for example, it cannot require parameter values to be numbers. It cannot depend on the dynamic or lexical environment. The form will be evaluated at compile time with uninterned symbols used as dummy values for the parameters and options. In the type specifier produced by evaluating the form, the type name must be a constant that names a type, the type parameters cannot derive from options of the type being defined, and the type options cannot derive from parameters of the type being defined. All presentation types mentioned must be already defined. `and` can be used for multiple inheritance, but `or`, `not`, and `satisfies` cannot be used.

define-presentation-method **[Macro]**

Arguments: *presentation-function-name qualifiers* specialized-lambda-list* &body *body*

■ Defines a presentation method for the function named *name* on the presentation type named in *specialized-lambda-list*.

specialized-lambda-list is a CLOS specialized lambda list for the method, and its contents varies depending on what *name* is. *qualifier** is zero or more of the usual CLOS method qualifiers. *body* defines the body of the method.

None of the arguments is evaluated.

8.6.7 Defining new presentation methods

Under rare circumstances, you may wish to define or call a new presentation generic function. The following forms may be used to accomplish this.

define-presentation-generic-function **[Macro]**

Arguments: *generic-function-name presentation-function-name lambda-list* &rest *options*

■ Defines a new presentation named *presentation-function-name* whose methods are named by *generic-function-name*. *lambda-list* and *options* are as in **defgeneric**.

■ The first few arguments in *lambda-list* are treated specially. The first argument must be either type-key or type-class. If you wish to be able to access type parameters or options in the method, the next arguments must be either or both of parameters and options. Finally, a required argument called type must also be included in *lambda-list*.

■ For example, **describe-presentation-type** might be have been defined by the following:

```
(clim:define-presentation-generic-function
 describe-presentation-type-method clim:describe-presentation-type
 (type-key parameters options type stream plural-count))
```

■ None of the arguments is evaluated.

define-default-presentation-method **[Macro]**

Arguments: *presentation-function-name qualifiers* specialized-lambda-list* &body *body*

■ This is like **define-presentation-method**, except that it is used to define a default method that will be used if there are no more specific methods.

■ None of the arguments is evaluated.

funcall-presentation-generic-function **[Macro]**

Arguments: *presentation-function-name* &rest *arguments*

■ Funcalls the presentation generic function *presentation-function-name* with arguments *arguments*, using **funcall**.

■ The *presentation-function-name* arguments is not evaluated.

apply-presentation-generic-function **[Macro]**

Arguments: *presentation-function-name* &rest *arguments*

- Applies the presentation generic function *presentation-function-name* to arguments *arguments*, using **apply**.
- The *presentation-function-name* arguments is not evaluated.

8.6.8 CLIM operators for defining presentation type abbreviations

You can define an abbreviation for a presentation type for the purpose of naming a commonly used cliche. The abbreviation is simply another name for a presentation type specifier. You cannot define presentation methods on presentation type abbreviations.

define-presentation-type-abbreviation **[Macro]**

Arguments: *name parameters expansion* &key options

- Defines a presentation type that is an abbreviation for the presentation type specifier that is the value of *expansion*. Note that you cannot define any presentation methods on a presentation type abbreviation. If you need to define methods, use **define-presentation-type** instead.

  name must be a symbol and must not be the name of a CLOS class. *parameter*, and *options* are the same as in **define-presentation-type**.

  The type specifier produced by evaluating *expansion* can be a real presentation type or another abbreviation.
- This example defines a presentation type to read an octal integer:

```
(clim:define-presentation-type-abbreviation octal-integer (&optional low high)
 `((integer ,low ,high) :base 8 :description "octal integer"))
```

When writing presentation type abbreviations, it is sometimes useful to let CLIM include or exclude defaults for parameters and options. In some cases, you may also find it necessary to expand a presentation type abbreviation. The following three functions are useful in these circumstances.

expand-presentation-type-abbreviation **[Function]**

Arguments: *type* &optional *environment*

- **expand-presentation-type-abbreviation** is like **expand-presentation-type-abbreviation-1**, except that *type* is repeatedly expanded until all presentation type abbreviations have been removed.

expand-presentation-type-abbreviation-1 **[Function]**

Arguments: *type* &optional *environment*

- If the presentation type specifier *type* is a presentation type abbreviation, or is an `and`, `or`, `sequence`, or `sequence-enumerated` that contains a presentation type abbreviation, then **expand-presentation-type-abbreviation-1** expands the type abbreviation once, and returns two values, the expansion and `t`. If *type* is not a presentation type abbreviation, then the values *type* and `nil` are returned.

make-presentation-type-specifier **[Function]**

Arguments: *type-name-and-parameters* &rest *options*

■ Given a presentation type name and its parameters *type-name-and-parameters* and some presentation type options, make a new presentation type specifier that includes all of the type parameters and options. This is useful for assembling a presentation type specifier with options equal to their default values omitted. This is only useful for **define-presentation-type-abbreviation**, but not for the *inherit-from* clause of **define-presentation-type**.

■ For example,

```
(clim:make-presentation-type-specifier '(integer 1 10) :base 10)
→ (integer 1 10)

(clim:make-presentation-type-specifier '(integer 1 10) :base 8)
→ ((integer 1 10) :base 8)
```

8.6.9 More about presentation methods in CLIM

All presentation methods have an argument named *type* that must be specialized with the name of a presentation type. The value of *type* is a presentation type specifier, which can be for a subtype that inherited the method.

All presentation methods except those for **presentation-subtypep** have lexical access to the parameters from the presentation type specifier. Presentation methods for the following operators also have lexical access to the options from the presentation type specifier.

accept

present

describe-presentation-type

presentation-type-specifier-p

accept-present-default

Presentation methods inherit and combine in the same way as ordinary CLOS methods. The reason presentation methods are not exactly the same as ordinary CLOS methods revolves around the *type* argument. The parameter specializer for *type* is handled in a special way and presentation method inheritance arranges the type parameters and options seen by each method.

Here are the names of the various presentation methods defined by **define-presentation-method**, along with the lambda-list for each method.

present **[Presentation method]**

Arguments: *object* &optional *type stream view* &key acceptably for-context-type

■ This presentation method is responsible for displaying the representation of *object* having type *type* for a particular view *view*. The method's caller takes care of creating the presentation, so the method need only display the contents of the presentation.

The method must specify &key, but need only receive the keyword arguments that it is interested in. The remaining keyword arguments will be ignored automatically since the generic function specifies &allow-other-keys.

The **present** method can specialize on the *view* argument in order to define more than one view of the data. For example, a spreadsheet program might define a presentation type for revenue, which can be displayed either as a number or a bar of a certain length in a bar graph. Typically, at least one canonical view should be defined for a presentation type; for example, a **present** method specializing on the class `textual-view` should be defined if you want to allow the type to be displayed textually.

Note that CLIM captures the presentation type for its own use, and you should not modify it once you have handed it to CLIM.

Note that, for a particular view, the **present** and **accept** methods must be duals, that is, the **accept** method must be able to parse what the **present** prints.

accept **[Presentation method]**

Arguments: *type stream view* &key default default-type

■ This presentation method is responsible for parsing the representation of *type* for a particular view *view* on the stream *stream*. The **accept** method should return a single value, the object that was parsed, or two values, the object and its type (a presentation type specifier). The method's caller takes care of establishing the input context, defaulting, prompting, and input editing.

default and *default-type* are as in **accept**.

The method must specify &key, but need only receive the keyword arguments that it is interested in. The remaining keyword arguments will be ignored automatically since the generic function specifies &allow-other-keys.

The **accept** method can specialize on the *view* argument in order to define more than one input view for the data. In particular, the **accept** method specializing on the class `textual-view` must be defined if the programmer wants to allow the type to be used via the keyboard.

accept presentation methods can also call **accept** recursively. Such methods should be careful to call **accept** with `nil` for *prompt* and *display-default* unless prompting is really desired.

Note that, for a particular view, the **present** and **accept** methods must be duals, that is, the **accept** method must be able to parse what the **present** prints.

describe-presentation-type **[Presentation method]**

Arguments: *type stream plural-count*

■ This presentation method is responsible for textually describing the type *type*. *stream* will be a stream of some sort, never `nil`. *plural-count* is as in the **describe-presentation-type** function.

■ You will rarely need to define a method for this, since the `:description` option to **define-presentation-type** is usually sufficient.

default-describe-presentation-type [Function]

Arguments: *description stream plural-count*

■ Given a string *description* that describes a presentation type (such as "integer") and *plural-count* (either `nil` or an integer), this function pluralizes the string if necessary, prepends an indefinite article if appropriate, and outputs the result onto *stream*.

■ This function is useful when you are writing your own **describe-presentation-type** method, but want to get most of CLIM's default behavior.

presentation-typep **[Presentation method]**

Arguments: *object type*

■ This presentation method is called when the **presentation-typep** function requires type-specific knowledge. If the type name in *type* is or names a CLOS class, the method is called only if *object* is a member of the class and *type* contains parameters, and the method simply tests whether *object* is a member of the subtype specified by the parameters. For non-class types, the method is always called.

■ You must define a **presentation-typep** method if the presentation type does not have the same name as a CLOS class.

presentation-subtypep **[Presentation method]**

Arguments: *type putative-supertype*

■ This presentation method is called when the **presentation-subtypep** function requires type-specific knowledge.

The function **presentation-subtypep** walks the type lattice to determine that *type* is a subtype of *putative-supertype*, without looking at the type parameters. When a supertype of *type* has been found whose name is the same as the name of *putative-supertype*, then the **presentation-subtypep** method for that type is called in order to resolve the question by looking at the type parameters (that is, if the **presentation-subtypep** method is called, *type* and *putative-supertype* are guaranteed to be the same type, differing only in their parameters).

Unlike all other presentation methods, **presentation-subtypep** receives a *type* argument that has been translated to the presentation type for which the method is specialized; *type* is never a subtype. The method is only called if *putative-supertype* has parameters and the two presentation type specifiers do not have equal parameters.

presentation-subtypep returns two values, *subtypep* and *known-p*. *subtypep* can be `t` (meaning that *type* is definitely a subtype of *putative-supertype*) or `nil` (meaning that *type* is definitely not a subtype of *putative-supertype* when *known-p* is `t`, or that the answer cannot be determined if *known-p* is `nil`).

Since **presentation-subtypep** takes two arguments that are presentation types, the parameters are not lexically available as variables in the body of a presentation method. Use **with-presentation-type-parameters** if you want to access the parameters of the presentation types.

You must define a **presentation-subtypep** method if the presentation type has parameters.

accept-present-default **[Presentation method]**

Arguments: *type stream view default default-supplied-p present-p query-identifier* `&key prompt active-p`

■ This presentation method is called when **accept** turns into **present** inside of **accepting-values**. The default method calls **present** or **describe-presentation-type** depending on whether *default-supplied-p* is `t` or `nil`.

type, *stream*, *view*, *default*, and *query-identifier* are as for **accept**. *default-supplied-p* is `t` if and only if *default* was explicitly supplied to the call to **accept**.

■ You only need to define a method for **accept-present-default** when you wish to create interesting dialog behavior for the type.

presentation-refined-position-test **[Presentation method]**

Arguments: *record type x y*

■ This method used to definitively answer hit detection queries for a presentation, that is, determining that the point (*x*,*y*) is contained within the output record *record*. Its contract is exactly the same as for **output-record-refined-position-test**, except that it is intended to specialize on the presentation type *type*.

■ It is useful to define a **presentation-refined-position-test** method when the displayed output records that represent the presentation do not themselves implement the desired hit detection behavior. In practice, this comes up only rarely, since using the `:single-box` option to **present** and **with-output-as-presentation** will often produce the desired behavior.

highlight-presentation **[Presentation method]**

Arguments: *record type stream state*

■ This method is responsible for drawing a highlighting box around the *presentation* *record* on the *output recording stream* *stream*. *state* will be either `:highlight` or `:unhighlight`, meaning that the highlighting box should either be drawn or erased.

■ It is useful to define a **highlight-presentation** method when you wish to have special highlighting behavior, such as inverse video, for a presentation type.

presentation-type-specifier-p **[Presentation method]**

Arguments: *type*

■ The **presentation-type-specifier-p** method is responsible for checking the validity of the parameters and options for *type*. The default method returns `t`.

■ You will almost never need to define a **presentation-type-specifier-p** method.

8.6.10 Utilities for clim:accept presentation methods

The utilities documented in this section are typically useful with **accept** (and sometimes **present**) presentation methods.

The following two functions are used to read or write a token (that is, a string):

read-token **[Function]**

Arguments: *stream* &key `input-wait-handler pointer-button-press-handler click-only`

■ Reads characters from *stream* until it encounters an activation gesture, a delimiter gesture, or a pointer button event. All printing Standard Characters are acceptable (see CLtL p. 336, or CLtL2 p. 512). **read-token** returns the accumulated string that was delimited by an activation or delimiter gesture, leaving the delimiter unread, that is, still in the stream's input buffer.

input-wait-handler

Passed along to **read-gesture**. The default is a function that supports highlighting for **with-input-context**. You will rarely need to supply this.

pointer-button-press-handler

Passed along to **read-gesture**. The default is a function that supports presentation translators for **with-input-context**. You will rarely need to supply this.

click-only

If true, only pointer gestures are expected and anything else will result in a beep. The default is `nil`.

■ It is preferable to use **read-token** instead of **read-string** inside of **accept** methods.

write-token [Function]

Arguments: *token stream* &key acceptably

■ **write-token** is the opposite of **read-token**: given the string *token*, it writes it to the stream *stream*.

If *acceptably* is t and there are any characters in *token* that are delimiter gestures (see the macro **with-delimiter-gestures**), then **write-token** will surround the token with quotation marks, #\".

It is advisable to use **write-token** instead of **write-string** inside of **present** methods.

Sometimes, an **accept** method may wish to signal an error while it is parsing the user's input, or a nested call to **accept** may signal such an error itself. The following functions and conditions may be used:

simple-parse-error [Condition]

■ This condition is signaled when CLIM does not know how to parse some sort of user input while inside of **accept**. It is built on parse-error.

simple-parse-error [Function]

Arguments: *format-string* &rest *format-arguments*

■ Signals an error of type simple-parse-error. This can be called while parsing an input token, for example, by a method on **accept**. This function does not return.

input-not-of-required-type [Condition]

■ This condition is signaled when CLIM gets input that does not satisfy the specified type while inside of **accept**. It is built on parse-error.

input-not-of-required-type [Function]

Arguments: *object type*

■ Reports that input does not satisfy the specified type. *object* is a parsed object or an unparsed token (a string). *type* is a presentation type specifier. This function does not return.

Some **accept** methods will want to allow for the completion of partial input strings by the user. The following functions are useful for doing that. Please note that these functions return multiple values that are not appropriate for the **accept** presentation method to return (it should return one or two values, the object or the object and the presentation type, as described in section 8.6.9 above). Therefore, the following code is wrong and will cause errors which are hard to diagnose:

```
(define-presentation-method accept
    ((type my-tupe) stream (view textual-view) &key)
  (completing-from-suggestions (stream)
                               (dolist (name list-of-names)
                                 (suggest (format nil "~a" name) name))))
```

This seems a perfectly reasonable thing to do but it breaks things badly because **completing-from-suggestions** returns three values where the second value is typically t (indicating success). However, returning t as the presentation type can subsequently break things in a confusing manner. The correct thing to do, of course, it to wrap the call to **completing-from-suggestions** appropriately so that only the first value is returned:

```
(define-presentation-method accept
    ((type my-tupe) stream (view textual-view) &key)
  (let ((obj (completing-from-suggestions (stream)
                                          (dolist (name list-of-names)
                                        (suggest (format nil "~a" name) name))))
      obj))
```

complete-input [Function]

Arguments: *stream function* &key partial-completers allow-any-input possibility-printer help-displays-possibilities

■ Reads input from *stream*, completing over a set of possibilities. Typically, you will not need to call **complete-input** directly, but will instead use **complete-from-generator**, **complete-from-possibilities**, or **completing-from-suggestions**,

function

is a function of two arguments which is called to generate the possibilities. Its first argument is a string containing the input so far. Its second argument is the completion mode, one of the following:

:complete

Completes the input as much as possible, except that if the user's input exactly matches one of the possibilities, even if it is a left substring of another possibility, the shorter possibility is returned as the result.

:complete-limited

Completes the input up to the next partial delimiter.

:complete-maximal

Completes the input as much as possible.

:possibilities

Causes **complete-input** to return a list of the possible completions.

function must return five values:

string

The completed input string.

success

t if completion was successful (otherwise nil).

object

The accepted object (nil if unsuccessful).

nmatches

The number of possible completions of the input.

possibilities

An alist of completions ((*string object*) ...), returned only when the completion mode is :possibilities.

■ **complete-input** returns three values: *object*, *success*, and *string*.

■ *partial-completers* is a (possibly empty) list of characters that delimit portions of a name that can be completed separately. The default is an empty list. Often this will the partial completers will consist of spaces and dashes.

If *allow-any*-input is t, **complete-input** will return as soon as the user types an activation gesture, even if the input is not any of the possibilities. This is used when you want to complete

from a set of existing items, and still allow the user to type in the name of a new item, for example, CLIM's `pathname` type uses this. The default is `nil`.

If *`possibility-printer`* is supplied, it should be a function of three arguments, a possibility, a presentation type, and a stream. The function should display the possibility on the stream. The possibility will be a list of two elements, the first being a string and the second being the object corresponding to the string.

If *`help-displays`*-possibilities is `t` (the default), then when the user types a help character (one of the characters in `*help-gestures*`), CLIM will display all the matching possibilities. If `nil`, then CLIM will not display the possibilities unless the user types a possibility character (one of the characters in `*possibilities-gestures*`).

`complete-from-generator` [Function]

Arguments: *`string generator delimiters`* `&key (action :complete) predicate`

■ Given an input string *`string`* and a list of delimiter characters *`delimiters`* that act as partial completion characters, **`complete-from-generator`** completes against the possibilities that are generated by the function *`generator`*. *`generator`* is a function of two arguments, the string *`string`* and another function that it calls in order to process the possibility.

`action` will be one of `:complete`, `:complete-maximal`, `:complete-limited`, or `:possibilities`. See the function **`complete-input`**.

`predicate` should be a function of one argument, an object. If the predicate returns `t`, the possibility corresponding to the object is processed, otherwise it is not.

`complete-from-generator` returns five values, the completed input string, the success value (`t` if the completion was successful, otherwise `nil`), the object matching the completion (or `nil` if unsuccessful), the number of matches, and a list of possible completions if *`action`* was `:possibilities`.

You might use **`complete-from-generator`** inside the **`accept`** method for a cardinal number presentation type as follows:

```
(let ((possibilities '(("One" 1) ("Two" 2) ("Three" 3))))
 (flet ((generator (string suggester)
      (declare (ignore string))
      (dolist (possibility possibilities)
       (funcall suggester (first possibility) (second possibility)))))
  (clim:complete-input
   stream
   #'(lambda (string action)
     (clim:complete-from-generator
      string #'generator nil
      :action action)))))
```

`complete-from-possibilities` [Function]

Arguments: *`string completions delimiters`* `&key (action :complete) predicate name-key value-key`

■ Given an input string *`string`* and a list of delimiter characters *`delimiters`* that act as partial completion characters, **`complete-from-generator`** completes against the possibilities in the sequence (a list or a vector) *`completions`*.

The completion string is extracted from the possibilities in *`completions`* by applying *`name-key`*. The object is extracted by applying *`value-key`*. The default for name key is `first`, and the

default for value key is `second` (that is, the default format for each element in completions is a list of length 2).

action will be one of `:complete`, `:complete-maximal`, `:complete-limited`, or `:possibilities`. See the function **complete-input**.

predicate should be a function of one argument, an object. If the predicate returns `t`, the possibility corresponding to the object is processed, otherwise it is not.

complete-from-possibilities returns five values, the completed input string, the success value (`t` if the completion was successful, otherwise `nil`), the object matching the completion (or `nil` if unsuccessful), the number of matches, and a list of possible completions if *action* was `:possibilities`.

You might use **complete-from-possibilities** inside the **accept** method for a cardinal number presentation type as follows:

```
(let ((possibilities '(("One" 1) ("Two" 2) ("Three" 3))))
 (clim:complete-input
  stream
  #'(lambda (string action)
    (clim:complete-from-possibilities
     string possibilities nil
     :action action))))
```

completing-from-suggestions [Macro]

Arguments: `(`*stream* `&key partial-completers allow-any-input possibility-printer help-displays-possibilities) &body` *body*

■ Reads input from *stream*, completing over a set of possibilities generated by calls to **suggest** in *body*. Returns three values: *object*, *success*, and *string*.

partial-completers, *allow-any*-input, *possibility-printer*, and *help-displays*-possibilities are as for **complete-input**.

■ Here is an example of its use:

```
(clim:completing-from-suggestions (stream)
 (map nil
    #'(lambda(x)
      (clim:suggest (car x) (cdr x)))
    '(("One" . 1)
     ("Two" . 2)
     ("Three" . 3))))
```

suggest [Function]

Arguments: *name* `&rest` *objects*

■ Specifies one possibility for **completing-from-suggestions**. *completion* is a string, the printed representation. *object* is the internal representation.

This function has lexical scope and is defined only inside the body of **completing-from-suggestions**.

completion-gestures [Variable]

■ A list of gesture names that cause **complete-input** to complete the input as fully as possible. This includes the gesture corresponding to the Tab character.

`*possibilities-gestures*` **[Variable]**

■ A list of gesture names that cause **`complete-input`** to display a help message and the list of possibilities. This includes the gesture corresponding to the Control-? character.

`*help-gestures*` **[Variable]**

■ A list of gesture names that cause **`accept`** and **`complete-input`** to display a help message, and, for some presentation types, the list of possibilities. This includes the gesture corresponding to the Control-? character.

8.6.11 clim:accept and the input editor

Sometimes after an **`accept`** method has read some input from the user, it may be necessary to insert a modified version of that input back into the input buffer. The following two functions can be used to modify the input buffer:

`replace-input` [Generic function]

Arguments: *`stream new-input`* `&key start end rescan buffer-start`

■ Replaces the *`stream`*'s input buffer with the string *`new-input`*. *`start`* and *`end`* specify what part of the *`new-input`* will be inserted into the buffer, and default to 0 and the end of the string.

`buffer-start` specifies where *`new-input`* should be inserted, and defaults to the current position in the input line. If *`rescan`* is `t`, a rescan operation will be queued; the default is `nil`. Usually, you should use the default values for *`buffer-start`* and *`rescan`*, since the input editor automatically arranges for the correct behavior to occur under those circumstances.

You can use this in an **`accept`** method that needs to replace some of the user's input by something else. For example, **`complete-input`** uses it to replace partial input with the completed input.

■ The returned value is the position in the input buffer.

`presentation-replace-input` [Generic function]

Arguments: *`stream object type view`* `&key rescan buffer-start`

■ This is like **`replace-input`**, except that the new input to insert into the input buffer is gotten by presenting the object *`object`* with the presentation type *`type`* and view *`view`*.

`rescan` and *`buffer-start`* are as for **`replace-input`**.

For example, the following **`accept`** method reads a token followed by a system or a pathname, but if the user clicks on either a system or a pathname, it inserts that object into the input buffer and returns:

```
(clim:define-presentation-method clim:accept
                 ((type library) stream (view textual-view)
                 &key default)
 (clim:with-input-context '(or system pathname)) (object type)
    (let ((system (clim:accept '(clim:token-or-type (:private) system)
                 :stream stream :view view
                 :prompt nil :display-default nil
                 :default default
                 :additional-delimiter-gestures '(#\space)))
       file)
     (let ((char (clim:read-gesture :stream stream)))
```

```
    (unless (eql char #\space)
     (clim:unread-gesture char :stream stream))
    (when (eql system ':private)
     (setq file (clim:accept 'pathname
                 :stream stream :view view
                 :prompt "library pathname"
                 :display-default t)))
    (if (eql system ':private) file system)))
  (t (clim:presentation-replace-input stream object type view)
   (values object type))))
```

Occasionally, **accept** methods will want to change the conditions under which input fields (or the entire input line) should be terminated. The following macros are useful for this:

with-activation-gestures **[Macro]**

Arguments: (*additional-gestures* &key override) &body *body*

■ Specifies characters that terminate input during the execution of *body*. *additional-gestures* is a character or a form that evaluates to a list of characters. Since **accept** establishes a set of activation gestures, it is only rarely useful to establish you own set.

If *override* is t, then the *additional-gestures* will override the existing activation characters. If it is nil (the default), then *additional-gestures* will be added to the existing set of activation characters.

See the :activation-gestures option to **accept**. See also see the variable *standard-activation-gestures*.

with-delimiter-gestures **[Macro]**

Arguments: (*additional-gestures* &key override) &body *body*

■ Specifies characters that terminate an individual token but not the entire input sentence during the execution of *body*. *additional-gestures* is a character or a form that evaluates to a list of characters. Establishing your own set of delimiter gestures is most useful when you write an **accept** method that reads multiple fields separated by some delimiter.

If *override* is t, then the *additional-gestures* will override the existing blip characters. If it is nil (the default), then *additional-gestures* will be added to the existing set of blip characters.

See the :delimiter-gestures option to **accept**.

standard-activation-gestures **[Variable]**

■ A list of gesture names that cause the current input to be activated. This includes the gestures corresponding to the Return and Newline characters.

activation-gestures **[Variable]**

■ A list containing the gesture names of the currently active activation gestures.

activation-gesture-p **[Function]**

Arguments: *gesture*

■ Returns t if *gesture* is a currently active activation gesture.

delimiter-gestures **[Variable]**

■ A list containing the gesture names of the currently active delimiter gestures.

delimiter-gesture-p **[Function]**

Arguments: *gesture*

- Returns `t` if *gesture* is a currently active delimiter gesture.

8.6.12 Help facilities for clim:accept

accept tries to generate meaningful help messages based on the name of the presentation type, but sometimes this is not adequate. You can use **with-accept-help** to create more complex help messages.

with-accept-help **[Macro]**

Arguments: *options* &body *body*

- Binds the local environment to control Help and Control-? documentation for input to **accept**. **accept** sets up a help context each time it is called, so it is generally only useful to use **with-accept-help** when you are writing complex **accept** methods that read multiple fields by recursively calling **accept**.

  options is a list of option specifications. Each specification is itself a list of the form (*help-option help-string*). *help-option* is either a symbol that is a *help-type* or a list of the form (*help-type mode-flag*).

  help-type must be one of:

  `:top-level-help`
  Specifies that *help-string* be used instead of the default help documentation provided by **accept**.

  `:subhelp`
  Specifies that *help-string* be used in addition to the default help documentation provided by **accept**.

  mode-flag must be one of:

  `:append`
  Specifies that the current help string be appended to any previous help strings of the same help type. This is the default mode.

  `:override`
  Specifies that the current help string is the help for this help type; no lower-level calls to **with-accept-help** can override this. (`:override` works from the outside in.)

  `:establish-unless-overridden`
  Specifies that the current help string be the help for this help type unless a higher-level call to **with-accept-help** has already established a help string for this help type in the `:override` mode. This is what **accept** uses to establish the default help.

  help-string is a string or a function that returns a string. If it is a function, it receives three arguments, the stream, an action (either `:help` or `:possibilities`) and the help string generated so far.

- None of the arguments is evaluated.

Here are some examples of the use of **with-accept-help**. Evaluate the following forms and type Help or Control-?.

```
(clim:with-accept-help ((:subhelp "This is a test."))
 (clim:accept 'pathname))

==> You are being asked to enter a pathname. [ACCEPT did this for you]
  This is a test.              [You did this via :SUBHELP]

(clim:with-accept-help ((:top-level-help "This is a test."))
 (clim:accept 'pathname))

==> This is a test.              [You did this via :TOP-LEVEL-HELP]

(clim:with-accept-help (((:subhelp :override) "This is a test."))
 (clim:accept 'pathname))

==> You are being asked to enter a pathname. [ACCEPT did this]
  This is a test.              [You did this via :SUBHELP]

(clim:define-presentation-type test ())
(clim:define-presentation-method clim:accept ((type test) stream view &key)
 (values (clim:with-accept-help
       ((:subhelp "A test is made up of three things:"))
      (clim:completing-from-suggestions (...) ...))))

(clim:accept 'test) ==> You are being asked to enter a test.
            A test is made up of three things:
(clim:with-accept-help ((:subhelp "This is a test."))
```

accept uses the input editor to read textual input from the user. If you want an **accept** method to do any sort of typeout, you must coordinate it with the input editor via **with-input-editor-typeout** or **input-editor-format**.

8.6.13 Using views with CLIM presentation types

Views in CLIM provide a general mechanism by which data can be accepted and presented in different ways depending on context. While application programmers can define their own view classes and specialize presentation methods on these classes, the primary use of views in CLIM is in controlling the appearance of **accepting-values** dialogs.

This section gives a description of the built-in view classes available in CLIM and gives various examples of how, using views, the application programmer can control the choice of gadgets used in **accepting-values** dialogs.

The *view* argument, as passed to **accept**, can either be an instance of a view class, a symbol naming the class, or a list of values which are passed to **make-instance** to create the view object.

CLIM provides a number of view classes:

```
textual-view
textual-dialog-view
textual-menu-view
gadget-view
gadget-dialog-view
```

`gadget-menu-view`

`pointer-documentation-view`

and corresponding instances. For example `+textual-view+` is an instance of the class `textual-view`.

CLIM also provides a number of gadget views:

`toggle-button-view`

`push-button-view`

`radio-box-view`

`check-box-view`

`slider-view`

`text-field-view`

`text-editor-view`

`list-pane-view`

`option-pane-view`

Gadget views take the same initargs are the corresponding gadget class. For example the following view can be used to specify a vertical radio box:

```
'(radio-box-view :orientation :vertical)
```

CLIM defines `accept-present-default` presentation methods for these views and appropriate presentation types so they can be used in **`accepting-values`** dialogs. For example:

```
(accepting-values (stream)
  (accept '(member a b c) :view 'list-pane-view :stream stream))
```

Note that using a gadget view outside of an **`accepting-values`** won't work because no accept presentation methods (just `accept-present-default`) are defined on these views - so doing the following in the CLIM Listener will not have the desired effect:

```
(accept '(member a b c) :view 'list-pane-view :stream *standard-
input*)
```

Also `accept-present-default` methods are only defined for the following combinations of presentation type and views. (Note that `text-field-view` works with all types listed even though it is not repeated in each line).

Presentation type	Possible views

`completion`	`radio-box-view`
	`list-pane-view`
	`option-pane-view`

`subset-completion`	`check-box-view` `list-pane-view`
`boolean`	`toggle-button-view`
`real`	`slider-view`

Presentation type	Possible views
`float`	`slider-view`
`integer`	`slider-view`
`string`	`text-editor-view`
`t`	`text-field-view`

Note that `member`, `member-sequence` and `member-alist` are presentation type abbreviations for `completion`; and `subset`, `subset-sequence` and `subset-alist` are abbreviations for `subset-completion`.

Normally you don't need to specify a gadget view explicitly within an **`accepting-values`**. This is because `gadget-dialog-view` is an indirect view. CLIM decodes these views into a more specific view automatically depending on the presentation type. For example: for the completion presentation type, `gadget-dialog-view` is decoded to `radio-box-view`. You would only need to explicitly specify a view if you wanted to override the default, for example, to use a `list-pane-view` or to control the orientation of the radio box (see examples above).

The view that CLIM uses for an **`accepting-values`** is taken from the frame-manager. By default this is `gadget-dialog-view`. You can specify other views by binding `stream-default-view` of the **`accepting-values`** stream. For example:

```
(accepting-values (stream :view +textual-dialog-view+)
  (clim-utils:letf-globally (((stream-default-view stream)
                              +textual-dialog-view+))
    (accept '(member a b c) :stream stream)))
```

will give you a dialog with old CLIM 1 style fields rather than CLIM 2 gadgets.

`stream-default-view` [Generic function]

Arguments: *`stream`*

■ Returns the default view for the stream *`stream`*. Calls to **`accept`** default the *`view`* argument from **`stream-default-view`**.

Many CLIM streams will have the textual view, `+textual-view+`, as their default view. Inside of **`menu-choose`**, the default view will be `+textual-menu-view+` or `+gadget-menu-view+`. Inside of **`accepting-values`**, the default view will be what is returned by **`frame-manager-dialog-view`**, defined next.

You can change the default view for a stream by using **`setf`** on **`stream-default-view`**.

`frame-manager-dialog-view` [Generic function]

Arguments: *`frame-manager`*

■ Returns the view object that should be used to control the look-and-feel of **`accepting-values`** dialogs. In Allegro CLIM, this will be `+gadget-dialog-view+`, but it also makes sense for you to change the value to be `+textual-dialog-view+`.

■ You can change the dialog view for the frame managers by calling **`setf`** on **`frame-manager-dialog-view`**.

`textual-view` **[Class]**

■ The class that represents textual views. Textual views are used in most command-line oriented applications.

`textual-menu-view` **[Class]**

■ The class that represents the view that is used inside textual menus.

`textual-dialog-view` **[Class]**

■ The class that represents the view that is used inside textual **`accepting-values`** dialogs.

`+textual-view+` **[Constant]**

■ An instance of the class `textual-view`.

`+textual-menu-view+` **[Constant]**

■ An instance of the class `textual-menu-view`. Inside **`menu-choose`**, the default view for the menu stream is bound to `+textual-menu-view+`.

`+textual-dialog-view+` **[Constant]**

■ An instance of the class `textual-dialog-view`. Inside **`accepting-values`**, the default view for the dialog stream is bound to `+textual-dialog-view+`.

`gadget-view` **[Class]**

■ The class that represents gadget views. Gadgets views are used for toolkit-oriented applications.

`gadget-menu-view` **[Class]**

■ The class that represents the view that is used inside toolkit-style menus.

`gadget-dialog-view` **[Class]**

■ The class that represents the view that is used inside toolkit-style **`accepting-values`** dialogs.

■ The gadget dialog view is one example of an indirect view. When you use this view when calling **`accepting-values`**, CLIM decodes the view into a more specific view based on the presentation type. These more specific views include `+radio-box-view+`, `+check-box-view+`, `+toggle-button-view+`, `+slider-view+`, `+text-field-view+`, `+text-editor-view+`, `+list-pane-view+`, and `+option-pane-view+`.

The following is a table of presentation types and the actual view they map to when you use the `gadget-dialog-view` view.

Presentation type	Actual view
`clim:completion`	`clim:+radio-box-view+`
`clim:subset-completion`	`clim:+check-box-view+`
`clim:boolean`	`clim:+toggle-button-view+`
others	`clim:+text-field-view+`

`+gadget-view+` **[Constant]**

■ An instance of the class `gadget-view`.

`+gadget-menu-view+` **[Constant]**

■ An instance of the class `gadget-menu-view`. Inside **`menu-choose`**, the default view for the menu stream is bound to `+gadget-menu-view+`.

`+gadget-dialog-view+` **[Constant]**

■ An instance of the class `gadget-dialog-view`. Inside **`accepting-values`**, the default view for the dialog stream is bound to `+gadget-dialog-view+`.

8.6.14 Functions that operate on CLIM presentation types

These are some general-purpose functions that operate on CLIM presentation types.

`describe-presentation-type` **[Function]**

Arguments: *`presentation-type`* `&optional (`*`stream`* `*standard-output*)` `(`*`plural-count`* `1)`

■ Describes the *`presentation-type`* on the *`stream`*.

If *`stream`* is `nil`, a string containing the description is returned. *`plural-count`* is either `nil` (meaning that the description should be the singular form of the name), `t` (meaning that the description should the plural form of the name), or an integer greater than zero (the number of items to be described).

The *`presentation-type`* can be a presentation type abbreviation.

CLIM can generally figure out a good default description for a presentation type, but you can specialize this function to get a better description, if necessary. For example, CLIM's `complex` type has an `:after` method like this:

```
(define-presentation-method describe-presentation-type :after
             ((type complex) stream plural-count)
 (declare (ignore type plural-count))
 (unless (eq type '*)
  (format stream " whose components are ")
  (describe-presentation-type type stream t)))
```

`presentation-typep` **[Function]**

Arguments: *`object type`*

■ Returns `t` if *`object`* is of the type specified by *`type`*, otherwise returns `nil`. *`type`* may not be a presentation type abbreviation.

■ This function is analogous to **`typep`**.

`presentation-type-of` **[Function]**

Arguments: *`object`*

■ Returns a presentation type of which *`object`* is a member. **`presentation-type-of`** returns the most specific presentation type that can be conveniently computed and is likely to be useful to the programmer. This is often, but not always, the class name of the class of the object.

■ If **`presentation-type-of`** cannot determine the presentation type of the object, it may return either `expression` or `t`.

■ This is analogous to the Common Lisp **`type-of`** function.

`presentation-subtypep` [Function]

Arguments: *type putative-supertype*

- Answers the question 'Is the type specified by *type* a subtype of the type specified by *putative-supertype*?'. Neither *type* nor *putative-subtype* may be presentation type abbreviations.
- This function is analogous to **subtypep**.
- **presentation-subtypep** returns two values, *subtypep* and *known-p*. *subtypep* can be `t` (meaning that *type* is definitely a subtype of *putative-supertype*) or `nil` (meaning that *type* is definitely not a subtype of *putative-supertype* when *known-p* is `t`, or that the answer cannot be determined if *known-p* is `nil`).
- See the clim presentation method **presentation-subtypep** for a detailed description of how this works.

`with-presentation-type-decoded` [Macro]

Arguments: (*name-var* &optional *parameters-var options-var*) *type* &body *body*

- The specified variables are bound to the components of the presentation type specifier, the forms in *body* are executed, and the values of the last form are returned. The value of the *type* must be a presentation type specifier. *name-var*, if non-`nil`, is bound to the presentation type name. *parameters-var*, if present and non-`nil`, is bound to a list of the parameters. *options-var*, if present and non-`nil`, is bound to a list of the options.
- This macro is particularly useful inside of presentation translators, when you might wish to more closely examine the presentation type of the presentation on which the translator was called on.

`with-presentation-type-options` [Macro]

Arguments: (*type-name type*) &body *body*

- Variables with the same name as each option in the definition of the presentation type are bound to the option values in *type*, if present, or else to the defaults specified in the definition of the presentation type. The forms in *body* are executed in the scope of these variables and the values of the last form are returned.
- The value of the form *type* must be a presentation type specifier whose name is *type-name*. *type-name* is not evaluated.

`with-presentation-type-parameters` [Macro]

Arguments: (*type-name type*) &body *body*

- Variables with the same name as each parameter in the definition of the presentation type are bound to the parameter values in *type*, if present, or else to the defaults specified in the definition of the presentation type. The value of the form *type* must be a presentation type specifier whose name is *type-name*. *type-name* is not evaluated. The forms in *body* are executed in the scope of these variables and the values of the last form are returned.

8.7 Presentation translators in CLIM

CLIM provides a mechanism for translating between types. In other words, within an input context for presentation type A, the translator mechanism allows a programmer to define a translation from presentations of some other type B to objects that are of type A.

You can define *presentation translators* to make the user interface of your application more flexible. For example, suppose the input context is expecting a command. In this input context, all displayed commands

are sensitive, so the user can point to one to execute it. However, suppose the user points to another kind of displayed object, such as a student. In the absence of a presentation translator, the student is not sensitive because the user must enter a command and cannot enter anything else to this input context.

In the presence of a presentation translator that translates from students to commands, however, the student would be sensitive. In one scenario, the student is highlighted, and the middle pointer button does "Show Transcript" of the student.

A presentation translator defines how to translate from one presentation type to another. In the scenario above, the input context is `command`. A user-defined presentation translator states how to translate from the `student` presentation type to the `command` presentation type.

The concept of translating from an arbitrary presentation type to a command is so useful that CLIM provides a special macro for this purpose, **`define-presentation-to-command-translator`**. You can think of these presentation-to-command translators as a convenience for the users; users can select the command and give the argument at the same time. In fact, this is the fundamental tool for providing highly interactive user interfaces in CLIM.

Note that presentation-to-command translators make it easier to write applications that give a direct manipulation feel to the user.

8.7.1 What controls sensitivity in CLIM?

A presentation that appears on the screen can be *sensitive*. This means that the presentation can be operated on directly by using the pointer. In other words, the presentation is relevant to the current context. When the user moves the pointer over a sensitive presentation, the presentation is highlighted to indicate that it is sensitive. (In rare cases, the highlighting of some sensitive presentations is turned off.)

Sensitivity is controlled by three factors: the current input context, the location of the pointer, and the chord of modifier keys being pressed.

- Input context type -- a presentation type describing the type of input currently being accepted.
- Pointer location -- the pointer is pointing at a presentation or a blank area on the screen.
- Modifier keys -- these are control, meta, super, hyper, and shift. These keys expand the space of available gestures beyond what is available from the pointer buttons.

Presentation translators are the link among these three factors.

A presentation translator specifies the conditions under which it is applicable, a description to be displayed, and what to do when it is invoked by clicking a button on the pointer.

A presentation is sensitive if there is at least one applicable translator that could be invoked by clicking a button with the pointer at its current location and the modifier keys in their current state. If there is no applicable translator, there is no sensitivity, and no highlighting.

Each presentation translator has two associated presentation types, its *`from-presentation`*-type and *`to-presentation`*-type, which are the primary factors in its applicability. The basic idea is that a presentation translator translates an output presentation into an input presentation. Thus a presentation translator is applicable if the type of the presentation at the pointer matches *`from-presentation`*-type and the input context type 'matches' *`to-presentation`*-type. (We define what 'match' means below.) Each presentation translator is attached to a particular pointer gesture, which is a combination of a pointer button and a set of modifier keys. Clicking the pointer button while holding down the modifier keys invokes the translator.

A translator produces an input presentation consisting of an object and a presentation type, to satisfy the program accepting input. The result of a translator might be returned from **`accept`**, or might be absorbed

by a parser and provide only part of the input. An input presentation is not actually represented as an object. Instead, a translator's body returns two values. The object is the first value. The presentation type is the second value; it defaults to *to-presentation*-type if the body returns only one value.

8.7.2 CLIM operators for defining presentation translators

You can write presentation translators that apply to blank areas of the window, that is, areas where there are no presentations. Use `blank-area` as the *from-presentation-type*. There is no highlighting when such a translator is applicable, since there is no presentation to highlight. You can write presentation translators that apply in any context by supplying `nil` as the *to-presentation*-type.

define-presentation-translator supports the general case, and **define-presentation-to-command-translator** supports a common idiom.

define-presentation-translator **[Macro]**

Arguments: *name* (*from-type to-type command-table* &key (gesture ':select) tester tester-definitive documentation pointer-documentation (menu t) priority) arglist &body *body*

■ Defines a presentation translator named *name* which translates from objects of type *from-type* to objects of type *to-type*. *from-type* and *to-type* are presentation type specifiers, but must not include presentation type options. *from-type* and *to-type* may also be presentation type abbreviations. *to-type* can also be `nil`, in which case the translator applies in any input context since `nil` is a subtype of all presentation types.

■ None of the arguments to **define-presentation-translator** is evaluated. Here is what the arguments do:

name

The name of the presentation translator.

from-type

The presentation type of the presentation in a window. Presentation type options are not allowed in *from-type*.

to-type

The presentation type of the returned object. Presentation type options are not allowed in *to-type*. When *to-type* is `nil`, this translator is applicable in all input contexts.

command-table

This specifies which command table the translators should be stored in. It should be either a command table or the name of a command table. This translator will be applicable only when this command table is one of the command tables from which the current application frame's command table inherits.

gesture

A *gesture-name* (see the section 8.7.6 **Pointer Gestures in CLIM**). The body of the translator will be run only if the translator is applicable and the pointer event corresponding to the user's gesture matches the gesture name in the translator. For more information, see the section 8.7.3 **Applicability of CLIM presentation translators**. *gesture* defaults to `:select`.

Note that `:gesture` t means that any user gestures will match this translator, and `:gesture nil`, means that no user gesture will match this translator. `:gesture nil` is commonly used when the translator should appear only in a menu.

tester

Either a function or a list of the form (*tester-arglist* . *tester-body*), where *tester-arglist* takes the same form as *arglist* (see below), and *tester-body* is the body of the tester. The tester should return either `t` or `nil`. If it returns `nil`, then the translator is definitely not applicable. If it returns `t`, then the translator might be applicable, and the body of the translator may be run in order to definitively decide if the translator is applicable (for more information, see the section 8.7.3 **Applicability of CLIM presentation translators**). If no tester is supplied, CLIM arranges for a tester that always returns `t`.

You should use a tester when discriminating only on the input context and presentation type are not enough. For example, if you were implementing a gadget set by using presentations, your translators might have a tester that ensures that a gadget has not been disabled.

tester-definitive

When this is `t` and the tester returns `t`, this translator is definitely applicable. When this is `nil` and the tester returns `t`, this translator might be applicable; in order to find out for sure, the body of the translator is run, and, if it returns an object that matches the input context type (using **`presentation-typep`**), this translator is applicable. The default depends on whether *tester* is specified: `nil` if it is and *t* if it is not (and thus a translator with neither *tester* nor *tester-definitive* specified is definitively applicable). You will seldom need to use this.

documentation

An object that will be used to document the translator. For example, in menus: if the object is a string, the string itself will be used as the documentation. Otherwise, it should be either a function or a list of the form (*doc-arglist* . *doc-body*), where *doc-arglist* takes the same form as *arglist*, but includes a *stream* argument as well (see below), and *doc-body* is the body of the documentation function. The body of the documentation function should write the documentation to *stream*. The default is `nil`, meaning that there is no documentation.

pointer-documentation

Like *documentation* except that *pointer-documentation* is used in the pointer documentation line. This documentation is usually more succinct than *documentation*. If *pointer-documentation* is not supplied, it defaults to *documentation*.

You should use this if normal documentation is expensive to compute. If the normal documentation is expensive to compute, and you do not use `:pointer-documentation`, this can slow down input to your application, because CLIM will spend too much time computing pointer documentation.

menu

The value should be `t` or `nil`. The default is `t`, meaning the translator is to be included in the menu popped-up by the `:menu` gesture (click Right on a three-button mouse). Use `:menu` t `:gesture` nil to make the translator accessible only through the menu. `:menu` `nil` means that the translator should not appear in the menu.

priority

An integer that represents the priority of the translator. The default is 0. When there are several translators that match for the same gesture, the priority is used to determine which is to be used. The algorithm for comparing priorities is complicated. It is described under the heading **Determining the priority of translators** just below. Use *priority* when there is a collision between translators, and you want to specify that one take precedence over another.

arglist
tester-arglist
doc-arglist

An argument list that must be a subset (using **string-equal** as the comparison) of the canonical argument list:

```
(object presentation context-type frame event window x y)
```

In the body of the translator (or the tester), *object* will be bound to the presentation's object, *presentation* will be bound to the presentation that was clicked on, *context-type* will be bound to the presentation type of the context that actually matched, *frame* will be bound to the application frame that is currently active (usually *application-frame*), *event* will be bound to the object representing the gesture that the user used, *window* will be bound to the window stream from which the *event* came, and *x* and *y* will be bound to the X and Y positions within *window* where the pointer was when the user issued the gesture. The special variable *input-context* will be bound to the current input context.

body

is the body of the translator, and may return one, two, or three values. The first returned value is an object that must be **presentation-typep** of *to-type*. The second value is either nil or a presentation type that must be **presentation-subtypep** of *to-type*.

■ The third returned value is either nil or a list of options (as keyword-value pairs) that will be interpreted by **accept**. The only option currently used by **accept** is :echo. If the :echo option is t (the default), the object returned by the translator will be echoed by inserting its textual representing into the input buffer. If the :echo option is nil, the object will not be echoed.

■ *body* is run in the context of the application. The first two values returned by *body* are used, in effect, as the returned values for the call to **accept** that established the matching input context.

Determining the priority of translators

If there are more than one translator found by **find-applicable-translators**, the priority of the translators is used along with other information to decide which to use. The priority is an integer specified by the *priority* keyword argument to **define-presentation-translator**. The integer encodes the high-order and the low-order priorities.

The high-order priority is the first value returned by (floor *priority* 10), that is, the number of times 10 goes into *priority*. Thus, if priority is 8, the high-order priority is 0, priority 36 has high-order priority 3, and priority 1725 has high-order priority 172. The low-order priority is the second value returned by (floor *priority* 10), that is the remainder of dividing priority by 10. Thus, if priority is 8, the low-order priority is 8, priority 36 has low-order priority 6, and priority 1725 has low-order priority 5. There is a distinction between high- and low-order priorities because comparing *from-type*s of two translators of equal high-order priority takes precedence over comparing their low-order priorities.

Here are the specific rules. Translators are compared by each rule in order.

1. Translators with a higher high-order priority precede translators with a lower high-order priority. This allows programmers to set the priority of a translator in such a way that it always precedes all other translators.
2. Translators with a more specific *from-type* precede translators with a less specific *from-type*.
3. Translators with a higher low-order priority precede translators with a lower low-order priority. This allows programmers to break ties between translators that translate from the same type.

4. Translators from the current command table precede translators inherited from superior command tables.

Examples of presentation translators

Here is an example that defines a presentation translator to extract the real number representing the resistance of a resistor from a `resistor` presentation. Users have the options of typing in a resistance to the input prompt or clicking on a `resistor` presentation.

```
(clim:define-presentation-translator resistor-resistance
  (resistor real ECAD-command-table
   :documentation "Resistance of this resistor"
   :gesture :select )
  (object)
 (resistor-resistance object))
```

CLIM supplies an identity translator that maps an object of any presentation type to itself. This translator generates pointer documentation that is just the printed representation of the object. The following translator is what CLIM uses for items in menus. Its purpose is to generate better pointer documentation based on the menu item. It uses the `:priority` option to ensure that it takes precedence over the usual identity translator.

```
(clim:define-presentation-translator menu-item-identity
  (menu-item menu-item global-command-table
   :priority 1        ;prefer this to IDENTITY
   :tester-definitive t
   :documentation
    ((object stream)
    (let ((documentation (or (menu-item-documentation object)
                (menu-item-display object))))
     (write documentation :stream stream :escape nil)))
   :gesture :select)
  (object presentation)
 (values object (clim:presentation-type presentation)))
```

`define-presentation-to-command-translator` **[Macro]**

Arguments: *`name (from-type command-name command-table`* `&key (gesture ':select) tester documentation pointer-documentation (menu t) priority (echo t)) arglist &body` *`body`*

- Defines a presentation translator that translates a displayed presentation into a command.

  This is similar to **`define-presentation-translator`**, except that the *`to-type`* will be derived to be the command named by *`command-name`* in the command table *`command-table`*. *`command-name`* is the name of the command that this translator will translate to. Note that *`command-name`* and *`command-table`* are required arguments.

  The *`echo`* argument controls whether or not the command should be echoed in the command line when a user invokes this translator. The default for *`echo`* is `t`.

  The other arguments to **`define-presentation-to-command-translator`** are the same as for **`define-presentation-translator`**. For information on the arguments, see the macro **`define-presentation-translator`**.

  The body of the translator should return a list of the arguments to the command named by *`command-name`*. *`body`* is run in the context of the application. The returned value of the body, appended to the command name, is eventually passed to **`execute-frame-command`**.

■ None of the arguments to **`define-presentation-to-command-translator`** is evaluated.

Examples of Presentation to Command Translators

The following example defines a pair of translators. The first deletes a file. The second undeletes a file, but has a tester that causes the translator to be applicable only if the file has been deleted.

```
(clim:define-presentation-to-command-translator delete-file
  (pathname com-delete-file fsedit-command-table
   :tester ((object) (not (file-deleted-p object)))
   :documentation "Delete this file"
   :gesture :delete)
  (object)
 (list object))

(clim:define-presentation-to-command-translator undelete-file
  (pathname com-undelete-file fsedit-command-table
   :tester ((object) (file-deleted-p object))
   :documentation "Undelete this file"
   :gesture :delete)
  (object)
 (list object))
```

Note that, because of the highly restricted syntax of **`define-presentation-to-command-translator`**, these two command translators cannot be folded into a single command translator. You could write a normal translator that does this, as follows.

```
(clim:define-presentation-translator delete-or-undelete-file
  (pathname command fsedit-command-table
   :gesture :delete
   :documentation
    ((object stream)
    (if (file-deleted-p object)
      (write-string "Undelete this file" stream)
      (write-string "Delete this file" stream)))
   :tester-definitive t)
  (object)
 (if (file-deleted-p object)
   `(com-undelete-file ,object)
   `(com-delete-file ,object)))
```

`define-presentation-action` **[Macro]**

Arguments: *`name (from-type to-type command-table`*
`&key (gesture :select) tester documentation`
`pointer-documentation (menu t) priority)`
`arglist &body` *`body`*

■ This is similar to **`define-presentation-translator`**, except that the body of the action is not intended to return a value, but should instead side-effect some sort of application state.

`name`

The name of the presentation action.

from-type

The presentation type of the presentation in a window. Presentation type options are not allowed in *from-type*.

to-type

The presentation type of the current input context. Presentation type options are not allowed in *to-type*. When *to-type* is `nil`, this action is applicable in all input contexts.

command-table

Controls the applicability of this handler. *command-table* should be either a command table or the name of a command table. Actions are stored in the command table *command-table*.

- The other arguments to **define-presentation-action** are the same as for **define-presentation-translator**. For information on the arguments, see the macro **define-presentation-translator** above.
- None of the arguments to **define-presentation-action** is evaluated.

Note that an action does not satisfy requests for input as translators do. An action is something that happens while waiting for input. After executing an action, the program continues to wait for the same input it was waiting for prior to executing the action.

From time to time, it is appropriate to write application-specific presentation actions. The key test for whether something should be an action is that it makes sense for the action to take place while the user is entering a command sentence, and performing the action will not interfere with the input of the command sentence. For example, an application framework might have an action that changes what information is displayed in one of its panes. It makes sense to do this in the middle of entering a command because information displayed in that pane might be used in formulating the arguments to the command. This needn't interfere with the input of the command since a pane can be redisplayed without discarding the pending partial command. It is for these cases that the presentation action mechanism is provided. A simple rule of thumb is that actions may be used to alter how application objects are presented or displayed, but anything having to do with modification of application objects should be embodied in a command, with an appropriate set of translators.

In general, if you are using **define-presentation-action** to execute any kind of an application command, you should be using **define-presentation-translator** or **define-presentation-to-command-translator** instead.

Defining a Presentation Action

Presentation actions are only rarely needed. Often a presentation-to-command translator is more appropriate. One example where actions are appropriate is when you wish to pop up a menu during command input. Here is how CLIM's general menu action could be implemented:

```
(clim:define-presentation-action presentation-menu
  (t nil clim:global-command-table
   :tester-definitive t
   :documentation "Menu"
   :menu nil
   :gesture :menu)
  (presentation frame window x y)
 (clim:call-presentation-menu presentation clim:*input-context*
                frame window x y
                :for-menu t))
```

define-drag-and-drop-translator **[Macro]**

Arguments: *name* (*from-type to-type destination-type command-table* &key (gesture ':select) tester documentation (menu t) priority feedback highlighting pointer-cursor) arglist &body *body*

■ Defines a drag and drop (or direct manipulation) translator named *name* that translates from objects of type *from-type* to objects of type *to-type* when a from-presentation is picked up, dragged over, and dropped on to a to-presentation having type *destination-type*. *from-type*, *to-type*, and *destination-type* are presentation type specifiers, but must not include any presentation type options. *from-type*, *to-type* and *destination-type* may be presentation type abbreviations.

The interaction style used by these translators is that a user points to a from-presentation with the pointer, picks it up by pressing a pointer button matching *gesture*, drags the from-presentation to a to-presentation by moving the pointer, and then drops the from-presentation onto the to-presentation. The dropping might be accomplished by either releasing the pointer button or clicking again, depending on the frame manager. When the pointer button is released, the translator whose *destination-type* matches the presentation type of the to-presentation is chosen. For example, dragging a file to the TrashCan on a Macintosh could be implemented by a drag and drop translator.

While the pointer is being dragged, the function specified by *feedback* is invoked to provide feedback to the user. The function is called with eight arguments: the application frame object, the from-presentation, the stream, the initial *x* and *y* positions of the pointer, the current *x* and *y* positions of the pointer, and a feedback state (either `:highlight` to draw feedback, or `:unhighlight` to erase it). The feedback function is called to draw some feedback the first time pointer moves, and is then called twice each time the pointer moves thereafter (once to erase the previous feedback, and then to draw the new feedback). It is called a final time to erase the last feedback when the pointer button is released. *feedback* defaults to **frame-drag-and-drop-feedback**, whose default method simply draws the bounding rectangle of the object being dragged.

When the from-presentation is dragged over any other presentation that has a direct manipulation translator, the function specified by *highlighting* is invoked to highlight that object. The function is called with four arguments: the application frame object, the to-presentation to be highlighted or unhighlighted, the stream, and a highlighting state (either `:highlight` or `:unhighlight`). *highlighting* defaults to **frame-drag-and-drop-highlighting**, whose default method simply draws a box around the object over which the dragged object may be dropped.

Note that it is possible for there to be more than one drag and drop translator that applies to the same from-type, to-type, and gesture. In this case, the exact translator that is chosen for use during the dragging phase is unspecified. If these translators have different feedback, highlighting, documentation, or pointer documentation, the exact behavior is unspecified.

■ The other arguments to **define-drag-and-drop-translator** are the same as for **define-presentation-translator**.

Examples of Drag and Drop Translators

Suppose you are implementing some sort of desktop interface to a file system editor. You have already written commands for Hardcopy File, Delete File, and so forth, and you want a drag-and-drop interface. Assuming you have some icons that represent a hardcopy device, a trashcan, and so forth, and presentation types that correspond to those icons, you could do the following:

```
(clim:define-drag-and-drop-translator dm-hardcopy-file
  (pathname command printer fsedit-comtab
   :documentation "Hardcopy this file")
  (object destination-object)
```

```
 '(com-hardcopy-file ,object ,destination-object))

(clim:define-drag-and-drop-translator dm-delete-file
  (pathname command trashcan fsedit-comtab
   :documentation "Delete this file")
  (object)
 '(com-delete-file ,object))

(clim:define-drag-and-drop-translator dm-copy-file
  (pathname command folder fsedit-comtab
   :documentation "Copy this file")
  (object destination-object)
 '(com-copy-file ,object ,(make-pathname :name (pathname-name object)
                    :type (pathname-type object)
                    :defaults destination-object)))
```

`blank-area` **[Presentation type]**

■ The type that represents all the places in a window where there is no currently active presentation. CLIM provides a single null presentation (represented by the value of `*null-presentation*`) of this type.

`*null-presentation*` **[Constant]**

■ The null presentation, which occupies all parts of a window where there are no presentations matching the current input context.

Defining a Presentation Translator from the Blank Area

When you are writing an interactive graphics routine, you will probably encounter the need to have commands available when the mouse is not over any object. To do this, you can write a translator from the `blank-area` type. You will probably want the *x* and *y* arguments to the translator.

For example:

```
(clim:define-presentation-to-command-translator add-circle-here
  (clim:blank-area com-add-circle my-command-table
   :documentation "Add a circle here.")
  (x y)
 '(,x ,y))
```

8.7.3 Applicability of CLIM presentation translators

When CLIM is waiting for input (that is, inside a **`with-input-context`**), it is responsible for determining what translators are applicable to which presentations in a given input context. This loop both provides feedback in the form of highlighting sensitive presentations, and is responsible for calling the applicable translator when the user presses a pointer button.

`with-input-context` uses **`frame-find-innermost-applicable-presentation`** (via **`highlight-applicable-presentation`**) as its input wait handler, and **`frame-input-context-button-press-handler`** as its button press event handler.

Given a presentation, an input context established by **`with-input-context`**, and a user gesture, translator matching proceeds as follows.

The set of candidate translators is initially those translators accessible in the command table in use by the current application. For more information, see the section 10.3 **Command objects in CLIM**.

A translator *matches* if all of the following are true. Note that these tests are performed in the order listed.

1. The presentation's type is **`presentation-subtypep`** of the translator's *`from-type`*, ignoring type parameters (for example, if *`from-type`* is `number` and the presentation's type is `integer` or `float`, or if *`from-type`* is `(or integer string)` and presentation's type is `integer`).
2. The translator's *`to-type`* is **`presentation-subtypep`** of the input context type, ignoring type parameters.
3. The translator's gesture either is `t`, or is the same as the gesture that the user could perform with the current chord of modifier keys.
4. The presentation's object is **`presentation-typep`** of the translator's *`from-type`*, if the *`from-type`* has parameters.
5. The translator's tester returned a non-`nil` value. If there is no tester, the translator behaves as though the tester always returns `t`.
6. If there are parameters in the input context type and the `:tester-definitive` option is not `t` in the translator, the value returned by the body of the translator must be **`presentation-typep`** of the input context type. In **`define-presentation-to-command-translator`** and **`define-presentation-action`** the tester is always definitive.

The algorithm is somewhat more complicated in the face of nested presentations and nested input contexts. In this case, the sensitive presentation is the smallest presentation that matches the *innermost* input context. When the nested presentations are all of the same size, all translators for the presentations having the same size are computed, and the applicable translator is chosen from this set.

When there are several translators that match for the same gesture, the one with the highest priority is chosen (see the definition of **`define-presentation-translator`** and the information under the heading **Determining the priority of presentation translators** immediately following the definition for information on priority).

8.7.4 Input contexts in CLIM

Roughly speaking, the current *input context* indicates what type of input CLIM is currently asking the user for. These are the ways you can establish an input context in CLIM:

- **`accept`**
- **`accept-from-string`**
- The command loop of an application

Nested input contexts in CLIM

The input context designates a presentation type. However, the way to accept one type of object may involve accepting other types of objects as part of the procedure. (Consider the request to accept a complex number. It is likely to involve accepting two real numbers.) Such input contexts are called nested. In the case of a nested input context, several different context presentation types can be available to match the *`to-presentation`*-types of presentation translators.

Each level of input context is established by a call to **`accept`**. The macro **`with-input-context`** also establishes a level of input context.

The most common cause of input context nesting is accepting compound objects. For example, you might define a command called Show File, which reads a sequence of pathnames. When reading the argument to the Show File command, the input context contains `pathname` nested inside of `(sequence path-`

name). Acceptable keyboard input is a sequence of pathnames separated by commas. A presentation translator that translates to a (sequence pathname) supplies the entire argument to the command, and the command processor moves on to the next argument. A presentation translator that translates to a pathname is also applicable. It supplies a single element of the sequence being built up, and the command processor awaits additional input for this argument, or entry of a Space, Newline, or Return to terminate the argument.

When the input context is nested, sensitivity considers only the innermost context type that has any applicable presentation translators for the currently pressed chord of modifier keys.

8.7.5 Nested presentations in CLIM

Presentations can overlap on the screen, so there can be more than one presentation at the pointer location. Often when two presentations overlap, one is nested inside the other.

One cause of nesting is presentations of compound objects. For example, a sequence of pathnames has one presentation for the sequence, and another for each pathname.

When there is more than one candidate presentation at the pointer location, CLIM must decide which presentation is the sensitive one. It starts with the innermost presentation at the pointer location and works outwards through levels of nesting until a sensitive presentation is discovered. This is the innermost presentation that has any applicable presentation translators, to any of the nested input context types, for the currently pressed chord of modifier keys. Searching in this way ensures that a more specific presentation is sensitive. Note that nested input contexts are searched first, before nested presentations. For presentations that overlap, the most recently presented is searched first.

8.7.6 Gestures in CLIM

A *gesture* in CLIM is an input action by the user, such as typing a character or clicking a pointer button. A *pointer gesture* refers to those gestures that involve using the pointer.

An *event* is a CLIM object that represents a gesture by the user. (The most important pointer events are those of class pointer-button-event.)

A *gesture name* is a symbol that names a gesture.

You can use **define-gesture-name** to define your own gesture name.

Note that with the exception of the **define-gesture-name** forms (which you can use to map gesture names to keys and buttons), the application is independent of which platform it runs on. It uses keywords to give names to gestures, rather than making references to specific pointer buttons and keyboard keys.

Pointer gestures

CLIM defines the following pointer gesture names:

`:select`

For the most commonly used translator on an object. For example, use the `:select` gesture while reading an argument to a command to use the indicated object as the argument.

`:describe`

For translators that produce a description of an object (such as showing the current state of an object). For example, use the `:describe` gesture on an object in a CAD program to display the parameters of that object.

`:double-click`

The double-click mechanism works the same on both UNIX and Windows. You need to define the double-click gesture before you can use it:

```
(clim:define-gesture-name :double-click :pointer-button (:left :double))
(clim:define-gesture-name :double-right :pointer-button (:right :double))
(clim:define-presentation-to-command-translator com-center-screen
    (clim:blank-area com-center-screen came
     :gesture :double-click :menu nil)
    (window)
    (list window))
```

On both UNIX and Windows, a double-click generates four events, *press*, *release*, *press*, *release*. The first click will activate the :select gesture, the second click will activate the :double-click gesture. So if you have an applicable translator on both gestures, both will run.

On Windows, the operating system recognizes double-click events explicitly, using the double-click settings in your "Mouse" control panel. On UNIX, this mechanism was simulated within CLIM.

`:delete`

For translators that delete an object.

`:edit`

For translators that edit an object.

`:menu`

For translators that pop up a menu.

These correspond to the following events:

Gesture name	Event
`:select`	click Left
`:describe`	click Middle
`:double-click`	double-click on side specified by :pointer-button argument.
`:menu`	click Right
`:delete`	click Shift-Middle
`:edit`	click Meta-Left

The special gesture name `nil` is used in translators that are not directly invocable by a pointer gesture. Such a translator can be invoked only from a menu.

The special gesture name `t` means that the translator is available on every gesture.

Keyboard gestures

Most keyboard gestures simply map the obvious named key to the obvious action (e.g. the `:return` gesture is mapped to the Return key). Here are the defined gesture names:

Gesture name	Keystroke
`:return`	Return
`:newline`	Newline
`:tab`	Tab
`:rubout`	Rubout
`:backspace`	Backspace
`:page`	Page
`:line`	Line
`:escape`	Escape
`:end`	End
`:abort`	Control-Z
`:help`	help

Note that `:complete`, `:scroll`, `:refresh`, and `:clear-input` are not defined to correspond to keyboard keys in X. You can associate a key with these gesture by defining your own gesture name: In this case, we associate the `:complete` gesture with the R1 key:

```
(clim:define-gesture-name :complete :r1 (:complete))
```

8.7.7 Operators for gestures in CLIM

The following operators can be used to add or remove new gesture names, to examine CLIM event objects, or match CLIM event objects against gesture names.

`add-gesture-name` [Function]

Arguments: *`name type gesture-spec`* `&key (unique t)`

■ Adds a gesture named by the symbol *`name`* to the set of gesture names. *`type`* is the type of gesture being created, and must be one of the symbols described below. *`gesture-spec`* specifies the physical gesture that corresponds to the named gesture; its syntax depends on the value of *`type`*.

If *`unique`* is `t`, an error is signaled if there is already a gesture named *`gesture-name`*. The default is `nil`.

When *`type`* is `:keyboard`, *`gesture-spec`* is a list of the form (*`key-name`* . *`modifier-key-names`*). *`key-name`* is the name of a non-modifier key on the keyboard (see below). *`modifier-key-names`* is a (possibly empty) list of modifier key names, which are `:shift`, `:control`, `:meta`, `:super`, `:hyper`.

For the standard Common Lisp characters (the 95 ASCII printing characters including Space), *key-name* is the character object itself. For the other semi-standard characters, *key-name* is a keyword symbol naming the character (`:newline`, `:linefeed`, `:return`, `:tab`, `:backspace`, `:page`, and `:rubout`).

When *type* is `:pointer-button`, *gesture-spec* is a list of the form (*button-name* . *modifier-key-names*). *button-name* is the name of a pointer button (`:left`, `:middle`, or `:right`), and *modifier-key-names* is as above.

delete-gesture-name [Function]

Arguments: *gesture-name*

■ Removes the pointer gesture named *gesture-name*.

define-gesture-name [Macro]

Arguments: *name type gesture-spec* `&key (unique t)`

■ Defines a gesture named *name* by calling **add-gesture-name**. None of the arguments is evaluated.

■ For example:

```
(define-gesture-name :select :pointer-button (:left))
```

make-modifier-state [Function]

Arguments: `&rest` *modifiers*

■ Given a list of modifier state names, this creates an integer that serves as a modifier key state. The legal modifier state names are `:shift`, `:control`, `:meta`, `:super`, and `:hyper`.

event-matches-gesture-name-p [Function]

Arguments: *event gesture-name* `&optional` *port*

■ Returns `t` if the device event *event* matches the gesture named by *gesture-name*.

For pointer button events, the event matches the gesture name when the pointer button from the event matches the name of the pointer button one of the gesture specifications named by *gesture-name*, and the modifier key state from the event matches the names of the modifier keys in that same gesture specification.

For keyboard events, the event matches the gesture name when the key name from the event matches the key name of one of the gesture specifications named by *gesture-name*, and the modifier key state from the event matches the names of the modifier keys in that same gesture specification.

modifier-state-matches-gesture-name-p [Function]

Arguments: *state gesture-name*

■ Returns `t` if the modifier state *state* matches the modifier state of the gesture named by *gesture-name*. *state* is an integer returned by **make-modifier-state**.

8.7.8 Events in CLIM

An *event* is a CLIM object that represents some sort of user gesture (such as moving the pointer or pressing a key on the keyboard) or that corresponds to some sort of notification from the display server. Event objects store such things as the sheet associated with the event, the x and y position of the pointer within that sheet, the key name or character corresponding to a key on the keyboard, and so forth.

When you want to write your own gadgets, you will often specialize the **handle-event** generic function on some of these event classes.

`event` **[Class]**

- The protocol class that corresponds to any sort of CLIM event.
- Note that all of the event classes are immutable. You cannot change any of the slots of an event.

eventp [Function]

Arguments: *object*

- Returns `t` if *object* is an event, otherwise returns `nil`.

event-timestamp [Generic function]

Arguments: *event*

- Returns an integer that is a monotonically increasing timestamp for the event *event*.

event-type [Generic function]

Arguments: *event*

- For the event *event*, returns a keyword with the same name as the class name, except stripped of the '-event' ending. For example, the keyword `:key-press` is returned by **event-type** for an event whose class is `key-press-event`.

`device-event` **[Class]**

- The instantiable class that corresponds to any sort of device event. It is a subclass of `event`.

event-sheet [Generic function]

Arguments: *event*

- Returns the sheet on which the device event *event* occurred.

event-modifier-state [Generic function]

Arguments: *event*

- Returns an integer value that encodes the state of all the modifier keys on the keyboard. The modifier is returned as an integer with bits corresponding to the constants `+shift-key+`, `+control-key+`, `+meta-key+`, `+super-key+`, and `+hyper-key+`. A bit value of 1 means the corresponding shift key was pressed.

`keyboard-event` **[Class]**

- The instantiable class that corresponds to any sort of keyboard event. This is a subclass of `device-event`.

keyboard-event-key-name [Generic function]

Arguments: *keyboard-event*

■ Returns the name of the key that was pressed or released in a keyboard event. This will be a symbol whose value is port-specific. Key names corresponding to the set of standard characters (such as the alphanumerics) will be a symbol in the keyword package.

keyboard-event-character [Generic function]

Arguments: *keyboard-event*

■ Returns the character associated with the event *keyboard-event*, if there is any corresponding character. (For example, shift keys will not have a corresponding character.)

key-press-event [Class]

■ The instantiable class that corresponds to a key press event. This is a subclass of keyboard-event.

key-release-event [Class]

■ The instantiable class that corresponds to a key release event. This is a subclass of keyboard-event.

pointer-event [Class]

■ The instantiable class that corresponds to any sort of pointer event. This is a subclass of device-event.

pointer-event-x [Generic function]

Arguments: *pointer-event*

■ Returns the x position of the pointer at the time the event occurred, in the coordinate system of the sheet that received the event.

pointer-event-y [Generic function]

Arguments: *pointer-event*

■ Returns the y position of the pointer at the time the event occurred, in the coordinate system of the sheet that received the event.

pointer-button-event [Class]

■ The instantiable class that corresponds to any sort of pointer button event. This is a subclass of pointer-event.

pointer-event-button [Generic function]

Arguments: *pointer-button-event*

■ Returns an integer identifying the button that was pressed or released when the pointer button event *pointer-button-event* occurred. The value identifying the left button is the value of +pointer-left-button+, that identifying the middle button is the value of +pointer-middle-button+, and that identifying the right button is the value of +pointer-right-button+.

pointer-button-press-event [Class]

■ The instantiable class that corresponds to a pointer button press event. This is a subclass of pointer-button-event.

`pointer-button-release-event` **[Class]**

■ The instantiable class that corresponds to a pointer button release event. This is a subclass of `pointer-button-event`.

`pointer-motion-event` **[Class]**

■ The instantiable class that corresponds to any sort of pointer motion event. This is a subclass of `pointer-event`.

`pointer-boundary-event` **[Class]**

■ The instantiable class that corresponds to a pointer motion event that crosses some sort of sheet boundary. This is a subclass of `pointer-motion-event`.

`pointer-boundary-event-kind` [Generic function]

Arguments: *`pointer-boundary-event`*

■ Returns the kind of boundary event, which will be one of `:ancestor`, `:virtual`, `:inferior`, `:nonlinear`, `:nonlinear-virtual`, or `nil`. These event kinds correspond to the detail members for X11 enter and exit events.

`pointer-enter-event` **[Class]**

■ The instantiable class that corresponds to the pointer entering a sheet's region. This is a subclass of `pointer-boundary-event`.

`pointer-exit-event` **[Class]**

■ The instantiable class that corresponds to the pointer exiting a sheet's region. This is a subclass of `pointer-boundary-event`.

8.7.9 Low level functions for CLIM presentation translators

Some applications may wish to deal directly with presentation translators, for example, if you are tracking the pointer yourself and wish to locate sensitive presentations, or want to generate a list of applicable translators for a menu. The following functions are useful for finding and calling presentation translators directly.

`find-presentation-translators` [Function]

Arguments: *`from-type to-type command-table`*

■ Returns a list of all the translators in the command table *`command-table`* that translate from *`from-type`* to *`to-type`*, without taking into account any type parameters or testers. *`frame`* defaults to `*application-frame*`. *`from-type`* and *`to-type`* must not be presentation type abbreviations.

■ Do not modify the list returned by **`find-presentation-translators`**.

`find-applicable-translators` [Function]

Arguments: *`presentation input-context frame window x y`* `&key event modifier-state for-menu fastp`

■ Returns a list of translators that apply to *`presentation`* in the input context *`input-context`*. Since input contexts can be nested, **`find-applicable-translators`** iterates over all of contexts in *`input-context`*.

`frame` is the application frame. *`window`*, *`x`*, and *`y`* are the window the presentation is on, and the x and y position of the pointer (respectively).

event (which defaults to `nil`) is a pointer button event, and may be supplied to further restrict the set of applicable translators to only those whose gesture matches the pointer button event.

modifier-state (which defaults to the current modifier state for the window) may also be supplied to restrict the set of applicable translators to only those who gesture matches the shift mask. Only one of *event* or *modifier-state* may be supplied.

When *for-menu* is `t` (the default), this returns every possible translator, disregarding *event* and *modifier-state*. When *fastp* is `t`, this will simply return `t` if there are any translators. *fastp* defaults to `nil`.

presentation-matches-context-type [Function]

Arguments: *presentation context-type frame window x y* &key event (modifier-state 0)

■ Returns a non-`nil` value if there are any translators that translate from *presentation*'s type to *context-type*. (There is no *from-type* argument because it is derived from *presentation*.) *frame*, *window*, *x*, *y*, *event*, and *modifier-state* are as for **find-applicable-translators**.

If there are no applicable translators, **presentation-matches-context-type** will return `nil`.

test-presentation-translator [Function]

Arguments: *translator presentation context-type frame window x y* &key event (modifier-state 0) for-menu

■ Returns `t` if the translator *translator* applies to the presentation *presentation* in input context type *context-type*. (There is no *from-type* argument because it is derived from *presentation*.)

frame is the application frame. *window*, *x*, and *y* are the window the presentation is on, and the x and y position of the pointer (respectively).

event and *modifier-state* are, respectively, a pointer button event and a modifier state. These are compared against the translator's gesture-name. *event* defaults to `nil`, and *shift-mask* defaults to 0, meaning that no shift keys are held down. Only one of *event* or *modifier-state* may be supplied.

If *for-menu* is `t`, the comparison against *event* and *modifier-state* is not done.

presentation, *context-type*, *frame*, *window*, *x*, *y*, and *event* are passed along to the translator's tester if and when the tester is called.

If the translator is not applicable, **test-presentation-translator** will return `nil`.

test-presentation-translator is responsible for matching type parameters and calling the translator's tester. Under some circumstances, **test-presentation-translator** may also call the body of the translator to ensure that its value matches *to-type*.

call-presentation-translator [Function]

Arguments: *translator presentation context-type frame event window x y*

■ Calls the function that implements the body of *translator* on *presentation*'s object, and passes *presentation*, *context-type*, *frame*, *event*, *window*, *x*, and *y* to the body of the translator as well.

frame, window, *x*, and *y* are as for **find-applicable-translators**. *context-type* is the presentation type for the context that matched. *event* is the event corresponding to the user's gesture.

The returned values are the same as the values returned by the body of the translator, which should be the translated object and the translated type.

`document-presentation-translator` [Function]

Arguments: *translator presentation context-type frame event window x y* `&key (stream *standard-output*) documentation-type`

■ Computes the documentation string for *translator* and outputs it to *stream*. *presentation*, *context-type*, *frame*, *gesture*, *window*, *x*, and *y* are as for **find-applicable-translators**.

documentation-type should be either `:normal` or `:pointer`. When it is `:pointer`, **document-presentation-translator** tries to generate the short form of pointer documentation, otherwise it generates the normal form.

`call-presentation-menu` [Function]

Arguments: *presentation input-context frame window x y* `&key (for-menu t) label`

■ Finds all the applicable translators for *presentation* in the input context *input-context*, creates a menu that contains all of the translators, and pops up a menu from which the user can choose a translator. After the translator is chosen, it is called and the values are returned to the appropriate call to **with-input-context**.

frame, *window*, *x*, and *y* are as for **find-applicable-translators**. *for-menu*, which defaults to `t`, is used to decide which presentation translators go in the menu (their `:menu` option must match *for-menu*). *label* is used as a label for the menu, and defaults to `nil`, meaning the menu will not be labelled.

The following functions are useful for finding an application presentation in an output history.

`find-innermost-applicable-presentation` [Function]

Arguments: *input-context stream x y* `&key (frame *application-frame*) modifier-state event`

■ Given an input context *input-context*, an output recording window stream *window*, and X and Y positions *x* and *y*, **find-innermost-applicable-presentation** returns the innermost presentation that matches the innermost input context.

frame is the application frame. *modifier-state* is a mask that describes what shift keys are held down on the keyboard, and defaults to the window's current modifier state. *event* is a pointer button event.

`throw-highlighted-presentation` [Function]

Arguments: *presentation input-context button-press-event*

■ Calls the applicable translator for the *presentation*, *input-context*, and *button-press*-event (that is, the one corresponding to the user clicking a pointer button while over the presentation). If an applicable translator is found, this binds variables to the object and the presentation type returned from the translator and throws back to the call to **with-input-context** that establish the matching input context. `*application-frame*` should be bound to the current application frame when you call **throw-highlighted-presentation**.

highlight-applicable-presentation [Function]

Arguments: *frame stream input-context* &optional (*prefer-pointer-window* t)

■ This is the input wait handler used by **with-input-context**. It is responsible for locating the innermost applicable presentation, unhighlighting presentations that are not applicable, and highlighting the presentation that is applicable.

frame, *stream*, and *input-context* are as for **find-innermost-applicable-presentation**.

set-highlighted-presentation [Function]

Arguments: *stream presentation* &optional (*prefer-pointer-window* t)

■ Highlight the presentation *presentation* on *stream*. If *presentation* is nil, any highlighted presentations are unhighlighted.

If *prefer-pointer*-window is t (the default), this sets the highlighted presentation for the window that is located under the pointer. Otherwise it sets the highlighted presentation for the window *stream*.

unhighlight-highlighted-presentation [Function]

Arguments: *stream* &optional (*prefer-pointer-window* t)

■ Unhighlight any highlighted presentations on *stream*.

■ Most applications will never need to use any of these functions.

Chapter 9 Defining application frames in CLIM

9.1 Concepts of CLIM application frames

Application frames (or simply, *frames*) are the central abstraction defined by CLIM for presenting an application's user interface. A frame contains a hierarchy of *panes*, which can include CLIM stream panes and gadgets.

The look and feel of an application frame is managed by a *frame manager*. The frame manager is responsible for realizing the concrete, window system dependent gadget that corresponds to each abstract gadget (the abstract gadget might be a CLIM slider, for example, and its concrete gadget is a real Motif slider). It is also responsible for the look and feel of menus, dialogs, pointer documentation, and so forth. The use of a frame manager allows CLIM applications to support multiple looks and feels, which is very important when porting an application from one environment to another.

Application frames provide support for a standard interaction processing loop, like the Lisp read-eval-print loop. You are required to write only the code that implements the frame-specific commands and output display functions. A key aspect of this interaction processing loop is the separation of the specification of the frame's commands from the specification of the end-user interaction style.

The standard interaction loop consists of reading an input sentence (the command and all of its operands), executing the command, and updating the displayed information as appropriate. Command execution and display will not occur simultaneously, so user-defined functions need not cope with multiprocessing.

Note that this definition of the standard interaction loop does not constrain the interaction style to command-line interfaces. The input sentence may be entered via single keystrokes, pointer input, menu selection, or by typing full command lines. CLIM allows the application implementor to choose what subset of approaches will be applicable for each individual command.

9.2 Defining CLIM application frames

`define-application-frame` defines CLIM application frames. Application frames are represented by CLOS classes which inherit from **`standard-application-frame`**. You can specify a name for the application class, the superclasses (if there are any beyond **`standard-application-frame`**), the slots of the application class, and options.

The following operators are used to define and instantiate CLIM application frames.

`define-application-frame` **[Macro]**

Arguments: *`name superclasses slots`* `&rest` *`options`*

■ Defines an application frame. You can specify a *`name`* for the application class, the *`superclasses`* (if any), the *`slots`* of the application class, and *`options`*.

■ **define-application-frame** defines a class with the following characteristics:

- inherits some behavior and slots from the class `standard-application-frame`;
- inherits other behavior and slots from any other *superclasses* which you specify explicitly;
- has other slots, as explicitly specified by *slots*;
- none of the arguments is evaluated.

The arguments are:

name

A symbol naming the new frame and class.

superclasses

A list of superclasses from which the new class should inherit, as in **defclass**. When *superclasses* is `nil`, it behaves as though a superclass of `standard-application-frame` was supplied. If you do specify superclasses to inherit from, you must include `standard-application-frame` explicitly if none of the superclasses inherits from `standard-application-frame`.

slots

is a list of slot specifiers, as in **defclass**. Each instance of the frame will have slots as specified by these slot specifiers. These slots will typically hold any per-instance frame state.

options

allows you to customize the initial values of slots in either your specified *superclasses*, or in the application frame. The options are as follows. Note that you must supply `:panes` (and optionally `:layouts`) or `:pane`. There is no default for these options. All the remaining options have defaults.

`:panes` *pane-descriptions*

Specifies the application's panes. There is no default for this option. The syntax of *pane-descriptions* is given below in section 9.2.3. If you use this option, you might use the `:layouts` option as well. If you do not specify `:layouts`, a default is used. If you use `:panes`, you should not specify a value for `:pane`.

`:layouts` *layout*

Specifies the layout of the panes specified by the `:panes` option. The syntax of *layout* is given below in section 9.2.5.

`:pane` *form*

Specifies the application's panes. *form* is a Lisp form that creates all of the panes of the application. The syntax of *form* is the same as the syntax of a `:layouts` description. You should not use this option and the `:panes` option. Instead, use one or the other.

`:top-level` *top-level*

Allows you to specify the main loop for your application. The top level function defaults to **default-frame-top-level**, which is adequate for most applications. *top-level* is a list whose first element is the name of a function to be called to execute the top level loop. The function should take at least one argument, the frame. The rest of the list consists of additional arguments to be passed to the function. The default function is **default-frame-top-level**. (Note that you can use the `:prompt` keyword of **default-frame-top-level** to control application frame prompts in the interactor.)

`:command-table` *name-and-options*

Allows you to specify a particular command table for the application.

name-and-options is a list consisting of the name of the application's command table followed by some keyword-value pairs. The keywords can be `:inherit-from` or `:menu` (which are as in **`define-command-table`**). The default is to create a command table with the same name as the application.

`:disabled-commands` *commands*

Allows you to specify a particular set of initially disabled *commands* for the application. The default is `nil`.

`:command-definer` *value*

Where *value* is either `nil`, `t`, or another symbol. When it is `nil`, no command-defining macro is defined. When it is `t`, a command-defining macro is defined, whose name is of the form **`define-`***name*`-command`**. When it is another symbol, the symbol names the command-defining macro. The default is `t`.

`:menu-bar` *boolean*

Specifies whether or not CLIM should maintain a menu bar for the application. The default is `t`.

`:icon &key` *name pixmap clipping-mask*

name should be a string, *pixmap* should be a pattern (not, despite its name, a pixmap object), and *clipping-mask* should be a pattern.

`:geometry` *geometry*

If supplied, *geometry* specifies the default geometry of the frame (that is, its position and size). *geometry* is a property list whose properties may be `:left`, `:top`, `:right`, `:bottom`, `:width`, and `:height`. These properties will be used unless explicitly overridden in the call to **`make-application-frame`**.

`:default-initargs` *initargs*

Identical to the `:default-initargs` for **`defclass`**.

Some examples

Recall we define `*test-frame*` at the beginning of this manual (section 2.1) to use in the various examples. Here again is the define-application-frame form from that definition:

```
(define-application-frame test ()
  ()
  (:panes
   (display :application))
  (:layouts
   (default display)))
```

This is about as simple as it can get -- no superclasses (so the new frame inherits from `standard-application-frame` only) and no slots, a `:panes`/`:layouts` combination and not much else. Note that after defining the frame, we do:

```
(define-test-command (com-quit :menu t) ()
  (frame-exit *application-frame*))
```

`define-test-command` names the command definer for test since we did not specify a value for the `:command-definer` option. By default, therefore, **`define-`***name*`-command`** is the command definer.

Another example application defined in chapter 2 is for the `calendar` application:

```
(define-application-frame calendar ()
  ((selected-date :accessor calendar-selected-date)
   (region-end :accessor calendar-region-end)
   (months :accessor calendar-months))
  (:command-table (calendar
                   :inherit-from (calendar-file-commands
                                  calendar-edit-commands)
                   :menu (("File" :menu calendar-file-commands
                                  :mnemonic #f :documentation "File Commands")
                          ("Edit" :menu calendar-edit-commands
                              :mnemonic #\E :documentation "Edit Commands"))))
  (:pointer-documentation t)
  (:panes
   (display :application
            :incremental-redisplay '(t :check-overlapping nil)
            :display-function 'draw-calendar
            :text-cursor nil
            :width :compute :height :compute
            :end-of-page-action :allow
            :end-of-line-action :allow)
   (dialog :accept-values
           :display-function
           '(accept-values-pane-displayer :displayer ,#'display-dialog)))
  (:layouts
   (:default
       (vertically ()
         dialog
         display))))
```

Again, no superclasses, but lots of slots, more complicated panes, and then a layout. Both these examples are discussed in chapter 2, where they are first defined. We mention them here to show what such things look like.

More application-frame functions and utilities

`standard-application-frame` **[Class]**

- The standard CLIM application frame class.
- **`define-application-frame`** arranges to inherit from this class if you do not supply an superclasses. If you do specify superclasses to inherit from, you must include `standard-application-frame` explicitly if none of the superclasses inherits from `standard-application-frame`.

`application-frame-p` [Function]

Arguments: *`object`*

- Returns `t` if and only if *`object`* is of type `application-frame`.

make-application-frame [Function]

Arguments: *frame-name* &key frame-class pretty-name frame-manager calling-frame left top right bottom height width user-specified-position-p user-specified-size-p &allow-other-keys

■ Makes an instance of the application frame of type *frame-class*. In addition to the keyword arguments listed, you can also supply CLOS initargs for :frame-class. The keyword arguments not handled by **make-application-frame** are passed as additional arguments to **make-instance**.

frame-name

A symbol which is the *name* argument to **define-application-frame**.

frame-class

The class to instantiate, defaults to *frame-name*. For special purposes you can supply a subclass of *frame-name*.

pretty-name

A string that is used as a title. It defaults to a prettified version of *frame-name*.

frame-manager

The frame manager for the frame. See the function **find-frame-manager**. The default is to use the default frame manager on the default port. A typical value is:

```
:frame-manager (find-frame-manager
                :server-path '(:motif :display "<disp>:0")
```

<disp> is the name of the desired display.

calling-frame

In window systems supporting a hierarchy of top level windows, the new frame is created as a child of *calling-frame*. This is important in the case of modal dialogs to ensure that the new frame can accept input. For example, if within in an **accepting-values** dialog a new frame is launched, that frame should be created with the **accepting-values** frame as *calling-frame*. This is achieved by specifying *application-frame* as *calling-frame*.

left

top

right

bottom

The coordinates of the left, top, right, and bottom edges of the frame, in device units. These default to the full size of the root window.

height

width

The size of the frame in device units. You can also use coordinates to specify the size of the frame.

user-specified-position-p

user-specified-size-p

These arguments tell the Window Manager to use the position or size specified by the arguments rather than determining a position or size itself. *user-specified-position-p* defaults to nil unless *top* and *left* are specified, in which case it defaults to t. *user-specified-size-p* defaults to nil unless *height* and *width* are specified in which case it defaults to t.

find-application-frame **[Function]**

Arguments: *frame-name* &rest *initargs* &key (create t) (activate t) (own-process *multiprocessing-p*) port frame-manager frame-class &allow-other-keys

■ Calling this function is similar to calling **make-application-frame**, then calling **run-frame-top-level** on the result.

If `create` is `t`, a new frame will be created if one does not already exist. If `create` is `:force`, a new frame will be created regardless of whether there is one already.

If `activate` is `t`, the frame's top level function will be invoked. If `own-process` is `t` (the default in Allegro CLIM), the frame will be invoked in its own process.

`port` and `frame-manager` can be used to name the parent of the frame. *`frame-class`* is as for **make-application-frame**. The rest of the *`initargs`* are passed on to **make-application-frame**.

9.2.1 Panes in CLIM

CLIM panes are stream panes similar to the gadgets or widgets of other toolkits. 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. The application programmer provides an abstract specification of the pane hierarchy, which CLIM uses in conjunction with user preferences and other factors to select a specific look and feel for the application. In many environments a CLIM application can use the facilities of the host window system toolkit via a set of *adaptive panes*, allowing a portable CLIM application to take on the look and feel of a native application user interface.

Panes are rectangular objects that are implemented as special sheet classes. An application will typically construct a tree of panes that divide up the screen space allocated to the application frame. The various CLIM pane types can be characterized by whether they have children panes or not: panes that can have other panes as children are called *composite panes*, and those that don't are called *leaf panes*. Composite panes are used to provide a mechanism for spatially organizing (laying out) other panes. Leaf panes implement gadgets that have a particular appearance and react to user input by invoking application code. Another kind of leaf pane provides an area of the application's screen real estate that can be used by the application to present application specific information. CLIM provides a number of these application pane types that allow the application to use CLIM's graphics and extended stream facilities.

Abstract panes are panes that are defined only in terms of their programmer interface, or behavior. The protocol for an abstract pane (that is, the specified set of initialization options, accessors, and callbacks) is designed to specify the pane in terms of its overall purpose, rather then in terms of its specific appearance or particular interactive details. The purpose of this abstract definition is to allow multiple implementations of the abstract pane, each defining its own specific look and feel. CLIM can then select the appropriate pane implementation based on factors outside the control of the application, such as user preferences or the look and feel of the host operating environment. A subset of the abstract panes, the *adaptive panes*, have been defined to integrate well across all CLIM operating platforms. These include buttons, sliders, scroll bars, and so forth.

9.2.2 Basic pane construction

Applications typically define the hierarchy of panes used in their frames using the `:pane` or `:panes` options of **`define-application-frame`**. These options generate the body of methods on functions that are invoked when the frame is being adopted into a particular frame manager, so the frame manager can select the specific implementations of the abstract panes.

There are two basic interfaces to constructing a pane: **`make-pane`** of an abstract pane class name, or **`make-instance`** of a concrete pane class. The former approach is generally preferable, since it results in more portable code. However, in some cases the programmer may wish to instantiate panes of a specific class (such as an `hbox-pane` or a `vbox-pane`). In this case, using **`make-instance`** directly circumvents the abstract pane selection mechanism. However, the **`make-instance`** approach requires the application programmer to know the name of the specific pane implementation class that is desired, and so is inherently less portable. By convention, all of the concrete pane class names, including those of the implementations of abstract pane protocol specifications, end in '`-pane`'.

Using **`make-pane`** instead of **`make-instance`** invokes the look and feel realization process to select and construct a pane. Normally this process is implemented by the frame manager, but it is possible for other realizers to implement this process. **`make-pane`** is typically invoked using an abstract pane class name, which by convention is a symbol in the CLIM package that doesn't include the '-pane' suffix. (This naming convention distinguishes the names of the abstract pane protocols from the names of classes that implement them.) Programmers, however, are allowed to pass any pane class name to **`make-pane`** in which case the frame manager will generally instantiate that specific class.

See the functions **`make-pane`** and **`make-clim-stream-pane`**.

9.2.3 Using the :panes option to clim:define-application-frame

The `:panes` option to **`define-application-frame`** is used to describe the panes used by the application frame. It takes a list of *`pane-descriptions`*. Each *`pane-description`* can be one of two possible formats:

- A list consisting of a *`pane-name`* (which is a symbol), a *`pane-type`*, and *`pane-options`*, which are keyword-value pairs. *`pane-options`* is evaluated at run time.
- A list consisting of a *`pane-name`* (which is a symbol), followed by a form that is evaluated at run time to create the pane. See **`make-clim-stream-pane`** and **`make-pane`**.

The *`pane-types`* are:

`:application`
: Application panes are for the display of application-generated output. See the class `application-pane` and the macro **`make-clim-application-pane`**.

`:interactor`
: Interactor panes provide a place for the user to do interactive input and output. See the class `interactor-pane` and the macro **`make-clim-interactor-pane`**.

`:accept-values`
: Pane for the display of an **`accepting-values`** dialog. See the class `accept-values-pane` and section 13.4 **Using an :accept-values pane in a CLIM application frame**.

`:pointer-documentation`

Pane for pointer documentation. If such a pane is specified, then when the pointer moves over different areas of the frame, this pane displays documentation of the effect of clicking the pointer buttons.

See the class `pointer-documentation-pane`.

`:title`

Title panes are used for displaying the title of the application. The default title is a prettified version of the name of the application frame. This will be handle by the window manager, so you need not include a title pane.

See the class `title-pane`.

`:command-menu`

Command menu panes are used to hold a menu of application commands. See the class `command-menu-pane`. This will be handle by the menu bar, so you need not include a command menu pane unless you require other command menus.

`:menu-bar`

Menu-bar panes use the toolkit menu bar to display a command menu. The command-table option defaults to the command-table of the frame. See the class `menu-bar`.

The *`pane-options`* usable by all pane types are:

The space requirement specs, `:width`, `:height`, `:min-width`, `:min-height`, `:max-width`, and `:max-height`.

These options control the space requirement parameters for laying out the pane. The `:width` and `:height` options specify the preferred horizontal and vertical sizes. The `:max-width` and `:max-height` options specify the maximum amount of space that may be consumed by the pane, and give CLIM's pane layout engine permission to grow the pane beyond the preferred size. The `:min-width` and `:min-height` options specify the minimum amount of space that may be consumed by the pane, and give CLIM's pane layout engine permission to shrink the pane below the preferred size.

If unspecified, `:max-width` and `:max-height` default to `+fill+` and `:min-width` and `:min-height` default to 0.

`:max-width`, `:min-width`, `:max-height`, and `:min-height` can also be specified as a relative size by supplying a list of the form (*`number`* `:relative`). In this case, the number indicates the number of device units that the pane is willing to stretch or shrink.

The values of these options are specified in the same way as the `:x-spacing` and `:y-spacing` options to **`formatting-table`**. (Note that `:character` and `:line` may only be used on those panes that display text, such as a `clim-stream-pane` or a `label-pane`.)

`:background`, `:foreground`

Initial values for **`medium-foreground`** and **`medium-background`** for the pane.

`:text-style` *`text-style`*

Specifies a text style to use in the pane. The default depends on the pane type.

`:borders`

Controls whether borders are drawn around the pane (`t` or `nil`). The default is `t`.

`:scroll-bars` *`scroll-bar-spec`*

A *`scroll-bar`*-spec can be `:both` (the default for `:application` panes), `:horizontal`, `:vertical`, or `nil`. The pane will have only those scroll bars that were specified. In addition

the :scroll-bars argument can be a cons, the car of which is treated at the non-cons argument and the cdr of which is a list of keyword argument pairs to be used as options to the scroller-pane (see the **scrolling** macro).

:display-after-commands

One of t, nil, or :no-clear. If t, the print part of the read-eval-print loop runs the display function; this is the default for most pane types. If nil, you are responsible for implementing the display after commands.

:no-clear behaves the same as t, with the following change. If you have not specified :incremental-redisplay t, then the pane is normally cleared before the display function is called. However, if you specify :display-after-commands :no-clear, then the pane is not cleared before the display function is called.

Note that :display-after-commands is retained primarily for compatibility with CLIM 1.1. It has the same functionality as using the :display-time option to **make-clim-stream-pane**.

:display-function *display-spec*

Where *display-spec* is either the name of a function or a list whose first element is the name of a function. The function is to be applied to the application frame, the pane, and the rest of *display-spec* if it was a list when the pane is to be redisplayed.

One example of a predefined display function is **display-command-menu**.

:display-string *string*

(for :title panes only) The string to display. The default is the frame's *pretty-name*. :incremental-redisplay *boolean* If t, CLIM initially runs the display function inside a **updating-output** form, and subsequent calls to **redisplay-frame-pane** will simply use **redisplay**. The default is nil.

:incremental-redisplay

This option, which defaults to nil, can either be a boolean or a list consisting of a boolean followed by keyword option pairs. The only option currently supported is :check-overlapping which takes a boolean value and defaults to t. Specifying this option as nil improves performance but should only be used where the output produced by the display function does not contain overlapping output. The :incremental-redisplay option is always used in conjunction with a pane display function (specified with :display-function).

:label

A string to be used as a label for the pane, or nil (the default).

:end-of-line-action, :end-of-page-action

These specify the initial values of the corresponding attributes.

:initial-cursor-visibility

:off means make the cursor visible if the window is waiting for input. :on means make it visible now. The default is :inactive which means the cursor is never visible. The default is :off for :interactor and :accept-values panes.

:output-record

Supply this if you want a different output history mechanism than the default.

:draw and :record

Specifies the initial state of drawing and output recording.

:default-view

Specifies the view object to use for the stream's default view.

`:text-margin`
Text margin to use if **`stream-text-margin`** isn't set. This defaults to the width of the viewport.

`:vertical-spacing`
Amount of extra space between text lines.

9.2.4 CLIM stream panes

This section describes the basic CLIM panes classes, and, in particular, the concept of a CLIM stream pane.

`pane` **[Class]**

■ The protocol class that corresponds to a pane, a subclass of `sheet`. A pane is a special kind of sheet that implements the pane protocols, including the layout protocols.

panep **[Function]**

Arguments: *`object`*

■ Returns `t` if *`object`* is a pane, otherwise returns `nil`.

`basic-pane` **[Class]**

■ The basic class on which all CLIM panes are built, a subclass of `pane`.

pane-frame **[Generic function]**

Arguments: *`pane`*

■ Returns the frame that owns the pane. You can call **`pane-frame`** on any pane in a frame's pane hierarchy, but it can only be invoked on active panes, that is, those panes that are currently adopted into the frame's pane hierarchy.

`clim-stream-pane` **[Class]**

■ This class implements a pane that supports the CLIM graphics, extended input and output, and output recording protocols. Most CLIM stream panes will be subclasses of this class.

`interactor-pane` **[Class]**

■ The pane class that is used to implement interactor panes (the `:interactor` type above). The default method for **`frame-standard-input`** will return the first pane of this type.

For **`interactor`**`-pane`, the default for `:display-time` is `nil` and the default for `:scroll-bars` is `:vertical`.

`application-pane` **[Class]**

■ The pane class that is used to implement application panes (the `:application` type above). The default method for **`frame-standard-output`** will return the first pane of this type.

For `application-pane`, the default for `:display-time` is `:command-loop` and the default for `:scroll-bars` is `t`.

`command-menu-pane` **[Class]**

■ The pane class that is used to implement command menu panes that are not menu bars (the `:command-menu` type above). The default display function for panes of this type is **`display-command-menu`**.

For `command-menu-pane`, the default for `:display-time` is `:command-loop`, the default for `:incremental-redisplay` is `t`, and the default for `:scroll-bars` is `t`.

`title-pane` **[Class]**

■ The pane class that is used to implement a title pane (the `:title` type above). The default display function for panes of this type is **`display-title`**.

For `title-pane`, the default for `:display-time` is `t` and the default for `:scroll-bars` is `nil`.

`pointer-documentation-pane` **[Class]**

■ The pane class that is used to implement the pointer documentation pane.

For `pointer-documentation-pane`, the default for `:display-time` is `nil` and the default for `:scroll-bars` is `nil`.

Making CLIM Stream Panes

Most CLIM stream panes will contain more information than can be displayed in the allocated space, so scroll bars are nearly always desirable. CLIM therefore provides a convenient form for creating composite panes that include the CLIM stream pane, scroll bars, labels, and so forth.

`make-clim-stream-pane` [Function]

Arguments: *`&rest options`* `&key type label label-alignment scroll-bars borders display-after-commands display-time &allow-other-keys`

■ Creates a pane of type *`type`*, which defaults to `clim-stream-pane`.

If *`label`* is supplied, it is a string used to label the pane. *`label-alignment`* is as for the **`labelling`** macro.

`scroll-bars` may be `t` to indicate that both vertical and horizontal scroll bars should be included, `:vertical` (the default) to indicate that vertical scroll bars should be included, or `:horizontal` to indicate that horizontal scroll bars should be included. In addition the `:scroll-bars` argument can be a cons, the car of which is treated at the non-cons argument and the cdr of which is a list of keyword argument pairs to be used as options to the scroller-pane (see the **`scrolling`** macro).

If *`borders`* is true, the default, a border is drawn around the pane.

`display-after`-commands is used to initialize the `:display-time` property of the pane. It may be `t` (for `:display-time :command-loop`), `:no-clear` (for `:display-time :no-clear`), or `nil` (for `:display-time` nil). See 9.2.3 **Using the :panes option to clim:define-application-frame**. You may only supply on of `:display-time` or `:display-after-commands`.

`display-time` is one of `:command-loop` (equivalent to `:display-after-commands` t), `:no-clear` (equivalent to `:display-after-commands :no-clear`) or `nil` (equivalent to `:display-after-commands nil`).

The other options may include all of the valid CLIM stream pane options.

`make-clim-interactor-pane` [Function]

Arguments: *`&rest options`*

■ Like **`make-clim-stream-pane`**, except that the type is forced to be `interactor-pane`.

`make-clim-application-pane` [Function]

Arguments: *&rest options*

- Like **`make-clim-stream-pane`**, except that the type is forced to be `application-pane`.

9.2.5 Using the :layouts Option to clim:define-application-frame

A *layout* is an arrangement of panes within the application-frame's top-level window. An application may have many layouts or it may have only one layout that remains constant for the life of the program. If you do not supply any layouts, CLIM will construct a default layout for the application.

CLIM's layout protocol is triggered by calling **`layout-frame`**, which is called when a frame is adopted by or resized by a frame manager.

CLIM uses a two pass algorithm to lay out a pane hierarchy. In the first pass (called *space composition*), the top level pane is asked how much space it requires. This may in turn lead to same the question being asked recursively of all the panes in the hierarchy, with the answers being composed to produce the top-level pane's answer. Each pane answers the query by returning a *space requirement* (or `space-requirement`) object, which specifies the pane's desired width and height as well as its willingness to shrink or grow along its width and height.

In the second pass (called *space allocation*), the frame manager attempts to obtain the required amount of space from the host window system. The top-level pane is allocated the space that is actually available. Each pane, in turn, allocates space recursively to each of its descendants in the hierarchy according to the pane's rules of composition.

The space requirement

For most types of panes, you can indicate the space requirements of the pane at creation time by using the space requirement options (described above). For example, application panes are used to display application-specific information, so the application can determine how much space should normally be given to them.

Other pane types automatically calculate how much space they need based on the information they need to display. For example, label panes automatically calculate their space requirement based on the text they need to display.

A composite pane calculates its space requirement based on the requirements of its children and its own particular rule for arranging them. For example, a pane that arranges its children in a vertical stack would return as its desired height the sum of the heights of its children. Note however that a composite is not required by the layout protocol to respect the space requests of its children; in fact, composite panes aren't even required to ask their children.

Space requirements are expressed for each of the two dimensions as a preferred size, a minimum size below which the pane cannot be shrunk, and a maximum size above which the pane cannot be grown. (The minimum and maximum sizes can also be specified as relative amounts.) All sizes are specified as a real number indicating the number of device units (such as pixels).

`make-space-requirement` [Function]

Arguments: `&key (width 0) (max-width 0) (min-width 0) (height 0) (max-height 0) (min-height 0)`

- Constructs a space requirement object with the given characteristics, `:width`, `:height`, and so on.

space-requirement-width [Generic function]

Arguments: *space-req*

space-requirement-min-width [Generic function]

Arguments: *space-req*

space-requirement-max-width [Generic function]

Arguments: *space-req*

space-requirement-height [Generic function]

Arguments: *space-req*

space-requirement-min-height [Generic function]

Arguments: *space-req*

space-requirement-max-height [Generic function]

Arguments: *space-req*

- These functions read the components of the space requirement *space-req*.

space-requirement-components [Generic function]

Arguments: *space-req*

- Returns the components of the space requirement *space-req* as six values, the width, minimum width, maximum width, height, minimum height, and maximum height.

space-requirement-combine [Function]

Arguments: *function sr1 sr2*

- Returns a new space requirement each of whose components is the result of applying the function *function* to each the components of the two space requirements *sr1* and *sr2*.
- *function* is a function of two arguments, both of which are real numbers. It has dynamic extent.

space-requirement+ [Function]

Arguments: *sr1 sr2*

- Returns a new space whose components are the sum of each of the components of *sr1* and *sr2*. This could be implemented as follows:

```
(defun space-requirement+ (sr1 sr2)
  (clim:space-requirement-combine #'+ sr1 sr2))
```

space-requirement+* [Function]

Arguments: *space-req* &key width min-width max-width height min-height max-height

- Returns a new space requirement whose components are the sum of each of the components of *space-req* added to the appropriate keyword argument (for example, the width component of *space-req* is added to *width*).
- This is a more efficient, spread version of **space-requirement+**.

compose-space [Generic function]

Arguments: *pane* &key width height

■ During the space composition pass, a composite pane will typically ask each of its children how much space it requires by calling **compose-space**. They answer by returning space-requirement objects. The composite will then form its own space requirement by composing the space requirements of its children according to its own rules for laying out its children.

The value returned by **compose-space** is a space requirement object that represents how much space the pane *pane* requires.

width and *height* are real numbers that the **compose-space** method for a pane may use as recommended values for the width and height of the pane. These are used to drive top-down layout.

allocate-space [Generic function]

Arguments: *pane width height*

■ During the space allocation pass, a composite pane will arrange its children within the available space and allocate space to them according to their space requirements and its own composition rules by calling **allocate-space** on each of the child panes. *width* and *height* are the width and height of *pane* in device units.

change-space-requirements [Generic function]

Arguments: *pane* &key resize-frame &rest space-req-keys

■ This function can be invoked to indicate that the space requirements for *pane* have changed. Any of the options that applied to the pane at creation time can be passed into this function as well.

resize-frame determines whether the frame should be resized to accommodate the new space requirement of the hierarchy. If *resize-frame* is t, then **layout-frame** will be invoked on the frame. If *resize-frame* is nil, then the frame may or may not get resized depending on the pane hierarchy and the :resize-frame option that was supplied to **define-application-frame**.

The layout

As the application is running, the current layout may be changed to any of the layouts described in the :layouts option of the frame definition. See **(setf frame-current-layout)** and **frame-current-layout**.

The :layouts option specifies and names the layouts of the application. A layout typically consists of rows, columns, and tables of panes, or more complicated nestings of rows, columns and tables. The value of the :layouts option is a list of layout descriptions. Each layout description is a two element list consisting of a symbol, which names the layout, and a corresponding *layout-spec*.

A *layout-spec* is a simply a form consisting of the various layout macros that constructs a pane.

A *size-spec* can be :fill, :compute, or a real number between zero and one. :fill means to use the remaining available space.

:compute means to run the :display-function to compute the size. Note that the display function is run at frame-creation time, so it must be able to compute the size correctly at that time.

A real number (between zero and one) is the fraction of the available space to use along the major axis.

The following macros provide layout for other panes in CLIM.

`+fill+` **[Constant]**

■ This constant can be used as a value to any of the relative size options in the following macros. It indicates that pane's willingness to adjust an arbitrary amount in the specified direction.

`horizontally` **[Macro]**

Arguments: `(&rest` *`options`* `&key spacing &allow-other-keys)` `&body` *`contents`*

■ The **`horizontally`** macro lays out one or more child panes horizontally, from left to right. The **`horizontally`** macro serves as the usual interface for creating an `hbox-pane`.

`spacing` specifies how much space should be left between each child pane. *`options`* may include other pane initargs, such as space requirement options.

This macro creates a pane of class `hbox-pane`. It does not create any corresponding concrete pane on the display server; CLIM handles this sort of layout itself.

`contents` is one or more forms that produce the child panes. Each form in *`contents`* is of the form:

- A pane. The pane is inserted at this point and its space requirements are used to compute the size.
- A number. The specified number of device units should be allocated at this point.
- The symbol `:fill`. This means that an arbitrary amount of space can be absorbed at this point in the layout.
- A list whose first element is a number and whose second element evaluates to a pane. If the number is less than 1, then it means that that fraction of excess space or deficit should be allocated to the pane. If the number is greater than or equal to 1, then that many device units are allocated to the pane. For example:

```
(clim:horizontally ()
 (1/3 (clim:make-pane 'label-button-pane))
 (2/3 (clim:make-pane 'label-button-pane)))
```

would create a horizontal stack of two label button panes. The first pane takes one-third of the space, then second takes two-thirds of the space.

`vertically` **[Macro]**

Arguments: `(&rest` *`options`* `&key spacing &allow-other-keys) &body` *`contents`*

■ The **`vertically`** macro lays out one or more child panes vertically, from top to bottom. The **`vertically`** macro serves as the usual interface for creating a `vbox-pane`.

The arguments to **`vertically`** are exactly the same as for the **`horizontally`** macro.

This macro creates a pane of class `vbox-pane`. It does not create any corresponding concrete pane on the display server; CLIM handles this sort of layout itself.

For example, the following will lay out its three children in a vertical stack, and the size of the stack will be determined from the children.

```
(clim:vertically ()
 (clim:make-pane 'push-button)
 (clim:make-pane 'push-button)
 (clim:make-pane 'toggle-button))
```

tabling **[Macro]**

Arguments: `(&rest` *`options`*`) &body` *`contents`*

■ The **tabling** macro lays out its child panes in a two-dimensional table arrangement. *contents* is a collection of elements, specified serially. Each element is itself specified by a list. For example,

```
(clim:tabling ()
 ((clim:make-pane 'push-button :text "Red")
  (clim:make-pane 'push-button :text "Green")
  (clim:make-pane 'push-button :text "Blue"))
 ((clim:make-pane 'push-button :text "Intensity")
  (clim:make-pane 'push-button :text "Hue")
  (clim:make-pane 'push-button :text "Saturation")))
```

■ This macro creates a pane of class `table-pane`. It does not create any corresponding concrete pane on the display server; CLIM handles this sort of layout itself.

outlining **[Macro]**

Arguments: `(&rest` *`options`* `&key thickness &allow-other-keys) &body` *`contents`*

■ The **outlining** layout macro puts an outlined border of the specified thickness around a single child pane. *contents* is a form that produces a single pane.

options may include other pane initargs, such as space requirement options, medium options (`:foreground`, and so on), and so forth.

spacing **[Macro]**

Arguments: `(&rest` *`options`* `&key thickness background &allow-other-keys) &body` *`contents`*

■ The **spacing** reserves some margin space around a single child pane. *thickness* specifies the amount of space, and *background* specifies the ink to be used as the panes background (that is, the color of the margin space). *contents* is a form that produces a single pane.

options may include other pane initargs, such as space requirement options, medium options (`:foreground`, and so on), and so forth.

labelling **[Macro]**

Arguments: `(&rest` *`options`* `&key label (label-alignment :top) &allow-other-keys) &body` *`contents`*

■ Creates a vertical stack consisting of two panes. One pane contains the specified label, which is a string. The other pane is specified by *contents*.

options may include other pane initargs, such as space requirement options, medium options (`:foreground`, and so on), and so forth.

scrolling **[Macro]**

Arguments: `(&rest` *`options`*`) &body` *`contents`*

■ Creates a composite pane that allows the single child specified by *contents* to be scrolled. *options* may include a `:scroll-bar` option. The value of this option may be `t` (the default), which indicates that both horizontal and vertical scroll bars should be created; `:vertical`, which indicates that only a vertical scroll bar should be created; or `:horizontal`, which indicates that only a horizontal scroll bar should be created. The following options are also supported:

`:drag-scroll`

If non-`nil` scrolling of the pane takes place as the scroll-bar is moved by the user. If `nil`, the pane is scrolled only after the scroll-bar is released by the user.

`:vertical-line-scroll-amount`

`:horizontal-line-scroll-amount`

Controls the distance the contents will scroll when the user activates the arrow at the end of the scroll bar.

`:vertical-page-scroll-amount`

`:horizontal-page-scroll-amount`

Controls the distance the contents will scroll when the user activates the scroll bar adjacent to the scroll bar slug. If the argument is an integer greater than 1 it represents the number of pixels to be scrolled. If it is a number less than or equal to 1 it represents the fraction of the viewport size to be scrolled.

The pane created by the **`scrolling`** macro will include a scroller pane that has as children the scroll bars and a viewport pane. The viewport of a pane is the portion of the window's drawing plane that is currently visible to the user. The viewport has as its child the specified contents.

This macro creates a scroll bar.

For example, the following creates a CLIM interactor pane with both vertical and horizontal scroll bars; the entire composite pane is surrounded by a thin amount of whitespace and a thin border.

```
(clim:outlining (:thickness 1)
 (clim:spacing (:thickness 1)
  (clim:scrolling (:scroll-bars :both)
   (clim:make-pane 'clim:interactor-pane
    :height 500 :width 600))))
```

The following functions return the viewport of a pane, the viewport's region, and scroll the pane. These are low-level functions; usually, you should use **`window-set-viewport-position`** to do programmatic scrolling.

`pane-viewport` [Generic function]

Arguments: *pane*

■ If the pane *pane* is part of a scroller pane, this returns the viewport pane for *pane*. Otherwise it returns `nil`.

`pane-viewport-region` [Generic function]

Arguments: *pane*

■ If the pane *pane* is part of a scroller pane, this returns the region of the pane's viewport. Otherwise it returns `nil`.

`scroll-extent` [Generic function]

Arguments: *pane x y*

■ If the pane *pane* is part of a scroller pane, this scrolls the pane in its viewport so that the position (*x*,*y*) of *pane* is at the upper-left corner of the viewport. Otherwise, it does nothing.

scroll-bar-size [Generic function]

Arguments: *scroll-bar*

■ Returns the size in pixels of the *scroll-bar*'s slug. This can be **setf**'d to change the size of a scroll-bar's slug

note-viewport-position-changed [Generic function]

Arguments: *frame pane*

■ This function is called when the position of *pane*'s viewport is changed. *frame* is

```
(pane-frame pane)
```

■ Applications can specialize on the *frame* argument to implement application-specific scrolling behavior.

9.2.6 Examples of the :panes and :layouts options to clim:define-application-frame

Here are some examples of how to use the `:panes` and `:layouts` options of **define-application-frame** to describe the appearance of your application.

We begin by showing an example of a CLIM frame with a simple layout, a single column of panes. Note that we use a `:command-menu` pane in this application instead of the more common menu bar. The command menus pane is allocated only enough space to display its contents, while the remaining space is divided among the other types of panes equally.

```
(clim:define-application-frame graphics-demo
            ()
  ()
 (:menu-bar nil)
 (:panes
  (commands :command-menu)
  (demo :application)
  (explanation :application :scroll-bars nil))
 (:layouts
  (main (clim:vertically () commands demo explanation))))
```

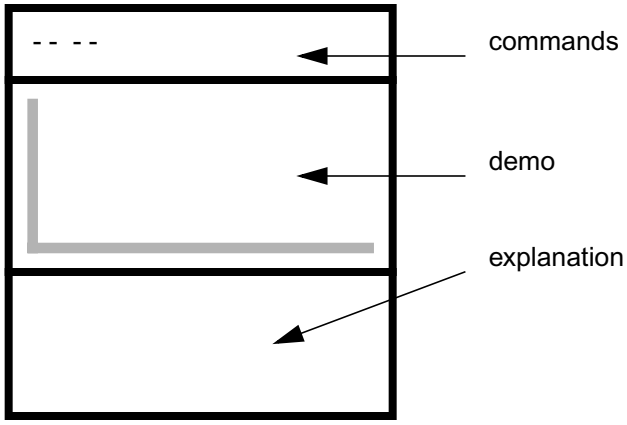


Figure 9.2. The default layout for the graphic-demo example when no explicit :layout is specified.

Now we add an explicit `:layouts` option to the frame definition from the previous example. The pane named `explanation` occupies the bottom sixth of the screen. The remaining five-sixths are occupied by the `demo` and `commands` panes, which lie side by side with the command pane to the right. The `commands` pane is only as wide as necessary to display the command menu.

```
(clim:define-application-frame graphics-demo
            ()
  ()
 (:panes
  (commands :command-menu)
  (demo :application)
  (explanation :application :scroll-bars nil))
 (:layouts
  (default
   (clim:vertically ()
    (5/6 (clim:horizontally demo commands))
    (1/5 explanation)))))
```

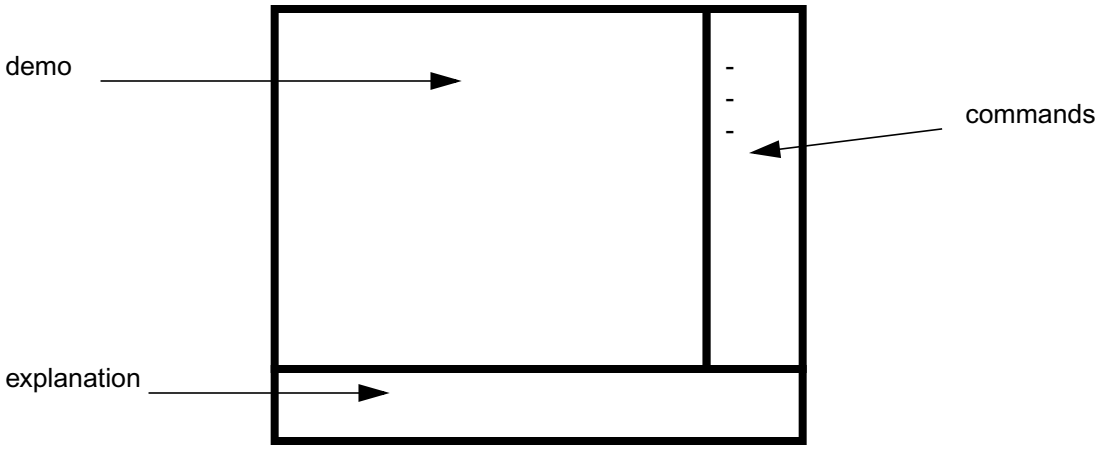


Figure 9.3. The layout for the graphic-demo example with an explicit :layout.

Finally, here is a stripped-down version of the application frame definition for the CAD demo (in the file *cad-demo.lisp* in the CLIM demos directory) that implements an extremely simplistic computer-aided logic circuit design tool.

There are four panes defined for the application. The pane named `title` displays the string "Mini-CAD" and serves to remind the user which application he is using. For the purpose of this example, we use a command menu pane instead of a menu bar. The pane named `design-area` is the actual work surface of the application on which various objects (logic gates and wires) can be manipulated. A pane named `documentation` is provided to inform the user about what actions can be performed using the pointing device (typically the mouse) and is updated based on what object is pointed to.

The application has two layouts, one named `main` and one named `other`. Both layouts have their panes arranged in vertical columns. At the top of both layouts is the `title` pane, which is of the smallest height necessary to display the title string "Mini-CAD". Both layouts have the `documentation` pane at the bottom.

The two layouts differ in the arrangement of the `menu` and `design-area` panes. In the layout named `main`, the `menu` pane appears just below the `title` pane and extends for the width of the screen. Its height will be computed so as to be sufficient to hold all the items in the menu. The `design-area` pane occupies the remaining screen real estate, extending from the bottom of the `menu` pane to the top of the `documentation` pane, and is as wide as the screen.

The layout named `other` differs from the `main` layout in the shape of the `design-area` pane. Here the implementor of the CAD demo realized that depending on what was being designed, either a short, wide area or a narrower but taller area might be more appropriate. The `other` layout provides the narrower, taller alternative by rearranging the `menu` and `design-area` panes to be side by side (forming a row of the two panes). The `menu` and `design-area` panes occupy the space between the bottom of the `title` pane and the top of the `documentation` pane, with the `menu` pane to the left and occupying as much width as is necessary to display all the items of the menu and the `design-area` occupying the remaining width.

```
(clim:define-application-frame cad-demo
   (clim:standard-application-frame clim:output-record)
  ((object-list :initform nil))
 (:menu-bar nil)
```

```
(:pointer-documentation t)
(:panes
 (title :title :display-string "Mini-CAD")
 (menu :command-menu)
 (design-area :application))
(:layouts
 (default
  (clim:vertically ()
   title menu design-area))
 (other
  (clim:vertically ()
   title
   (clim:horizontally () menu design-area)))))
```

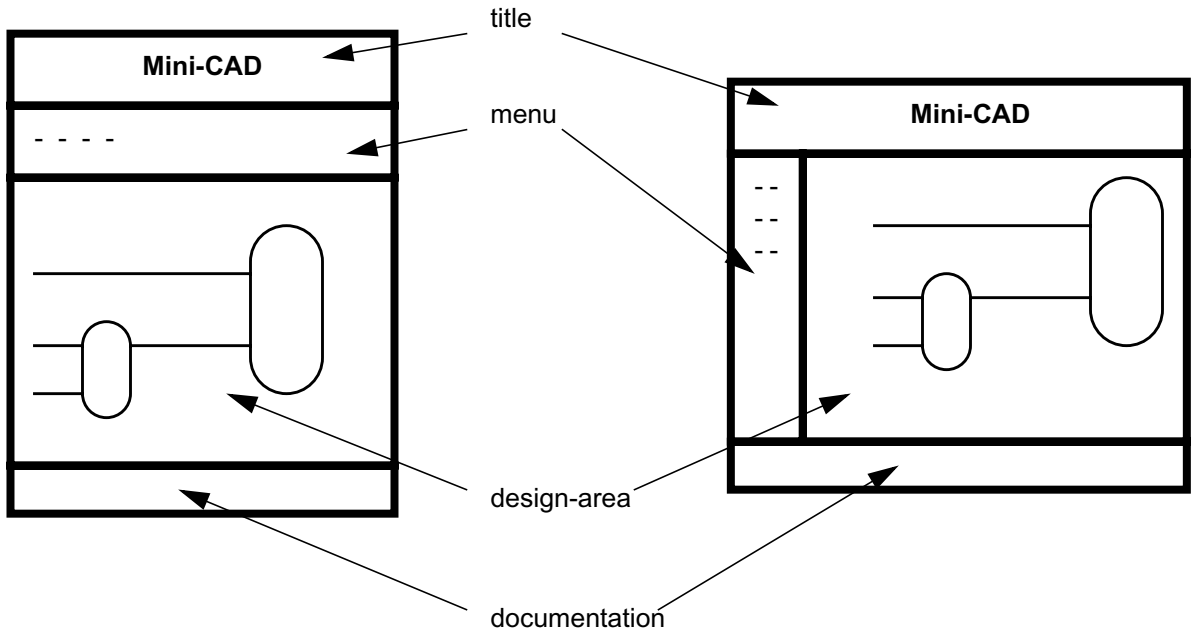


Figure 9.4. The two layouts of the Mini-CAD demo. Layout `main` is on the left, layout `other` is on the right.

9.3 CLIM application frames vs. CLOS

When you define an application frame, you also implicitly define a CLOS class that implements the frame. This section describes the interaction between you application frame classes and CLOS.

9.3.1 Initializing application frames

There are several ways to initialize an application frame:

- The value of an application frame's slot can be initialized using the `:initform` slot option (or `:default-initargs`) in the slot's specifier in the **`define-application-frame`** form. This technique is suitable if the slot's initial value does not depend on the values of other slots, calculations based on the values of initialization arguments, or other information that cannot be determined until after the application frame is created. See the macro **`defclass`** to learn about slot specifiers.
- For initializations that depend on information that may not be available until the application frame has been created, an `:after` method can be defined for **`initialize-instance`** on the application frame's class. Note that the special variable `*application-frame*` is not bound to the application since the application is not yet running. You may use **`with-slots`**, **`with-accessors`**, or any slot readers or accessors you have defined.
- A `:before` or `:around` method for **`run-frame-top-level`** on the application's frame is probably the most versatile place to perform application frame initialization. This method will not be executed until the application starts running. `*application-frame*` will be bound to the application frame. If the application frame employs its own top-level function, then this function can perform initialization tasks at the beginning of its body. This top-level function may call **`default-frame-top-level`** to achieve the standard behavior for application frames.

Of course, these are only suggestions. Other techniques might be more appropriate for your application. Of those listed, the `:before` or `:around` method on **`run-frame-top-level`** is probably the best for most circumstances.

Although application frames are CLOS classes, do not use **`make-instance`** to create them. To instantiate an application frame, always use **`make-application-frame`**. This function provides important initialization arguments specific to application frames which **`make-instance`** does not. **`make-application-frame`** passes any keyword value pairs which it does not handle itself on to **`make-instance`**. Thus, it will respect any initialization options which you give it, just as **`make-instance`** would.

9.3.2 Inheritance of application frames

Here is an example of how an application frame's behavior might be modified by inheritance from a superclass.

Suppose we wanted our application to record all of the commands which were performed while it was executing. This might be useful in the context of a program for the financial industry where it is important to keep audit trails for all transactions. As this is a useful functionality that might be added to any of a number of different applications, we will separate it out into a special class which implements the desired behavior. This class can then be used as a superclass for any application that should keep a log of its actions.

The class has a `:pathname` initarg which specifies the name of the log file. It has a slot named `transaction-stream` whose value is a stream opened to the log file when the application is running.

```
(defclass transaction-recording-mixin ()
    ((transaction-pathname :type pathname
                :initarg :pathname
                :reader transaction-pathname)
     (transaction-stream :accessor transaction-stream)))
```

We use an `:around` method on **`run-frame-top-level`** which opens a stream to the log file, and stores it in the `transaction-stream` slot. **`unwind-protect`** is used to clear the value of the slot when the stream is closed.

```
(defmethod clim:run-frame-top-level :around
     ((frame transaction-recording-mixin))
 (with-slots (transaction-pathname transaction-stream) frame
  (with-open-file (stream transaction-pathname
                  :direction :output)
   (unwind-protect
     (progn
      (setq transaction-stream stream)
      (call-next-method))
    (setq transaction-stream nil)))))
```

This is where the actual logging takes place. The command loop in **`default-frame-top-level`** calls **`execute-frame-command`** to execute a command. Here we add a `:before` method that will log the command.

```
(defmethod clim:execute-frame-command :before
     ((frame transaction-recording-mixin) command)
 (format (transaction-stream frame)
     "~&Command: ~A" command))
```

It is now an easy matter to alter the definition of an application to add the command logging behavior. Here is the definition of the puzzle application frame from the CLIM demos suite (from the file *clim-2.0/demos/puzzle.lisp*). Our modifications are shown in *`italics`*. We use the superclasses argument to specify that the `puzzle` application frame should inherit from `transaction-recording-mixin`. Because we are using the superclass argument, we must also explicitly include `standard-application-frame` which was included by default when the superclasses argument was empty.

```
(clim:define-application-frame puzzle
   (transaction-recording-mixin clim:standard-application-frame)
  ((puzzle :initform (make-array '(4 4))
       :accessor puzzle-puzzle))
 (:default-initargs :pathname "puzzle-log.text")
 (:panes
  (display :application
       :display-function 'draw-puzzle
       :text-style '(:fix :bold :very-large)
       :incremental-redisplay t
       :text-cursor nil
       :width :compute :height :compute
       :end-of-page-action :allow
       :end-of-line-action :allow))
 (:layouts
  (:default display)))
```

Also note the use of the following to provide a default value for the log file name if the user doesn't specify one.

```
(:default-initargs :pathname "puzzle-log.text")
```

The user might run the application by executing

```
(clim:run-frame-top-level
 (clim:make-application-frame 'puzzle
  :parent (find-port)
  :pathname "my-puzzle-log.text"))
```

Here the `:pathname` initarg was used to override the default name for the log file (as was specified by the `:default-initargs` clause in the above application frame definition) and to use the name "my-puzzle-log.text" instead.

9.3.3 Accessing slots and components of CLIM application frames

CLIM application frames are instances of the defined subclass of the `standard-application-frame` class. You explicitly specify accessors for the slots you have specified for storing application-specific state information, and use those accessors as you would for any other CLOS instance. Other CLIM defined components of application frame instances are accessed via documented functions. Such components include frame panes, command tables, top level sheet, and layouts.

9.4 Running a CLIM application

You can run a CLIM application using the functions **`make-application-frame`** and **`run-frame-top-level`**. First use **`find-port`** to create a port to pass as the `:parent` argument to **`make-application-frame`**. Here is a code fragment which illustrates this technique under Allegro:

```
(clim:run-frame-top-level
  (clim:make-application-frame 'frame-name
   :parent (clim:find-port :server-path '(:motif :host "localhost"))))
```

`run-frame-top-level` will not return until the application exits.

Note that `*application-frame*` is not bound until **`run-frame-top-level`** is invoked.

9.5 Examples of CLIM application frames

These are examples of how to use CLIM application frames.

9.5.1 Example of defining a CLIM application frame

Here is an example of an application frame. This frame has three slots, named `pathname`, `integer` and `member`. It has two panes, an `:accept-values` pane named `avv` and an `:application` pane named `display`. It uses a command table named `dingus`, which will automatically be defined for it (see **`define-command-table`**) and which inherits from the `accept-values-pane` command table so that the accept-values pane will function properly.

```
(clim:define-application-frame dingus ()
  ((pathname :initform #p"foo")
   (integer :initform 10)
   (member :initform :one))
 (:panes
  (avv :accept-values
     :display-function '(clim:accept-values-pane-displayer
                :displayer display-avv))
  (display :application
```

```
      :display-function 'draw-display
      :display-after-commands :no-clear))
 (:command-table (dingus :inherit-from (clim:accept-values-pane))))
```

This is the display function for the `display` pane of the "dingus" application. It just prints out the values of the three slots defined for the application.

```
(defmethod draw-display ((frame dingus) stream)
 (with-slots (pathname integer member) frame
  (fresh-line stream)
  (clim:present pathname 'pathname :stream stream)
  (write-string ", " stream)
  (clim:present integer 'integer :stream stream)
  (write-string ", " stream)
  (clim:present member '(member :one :two :three) :stream stream)
  (write-string "." stream)))
```

This is the display function for the `avv` pane. It invokes **`accept`** for each of the application's slots so that the user can alter their values in the `avv` pane.

```
(defmethod display-avv ((frame dingus) stream)
 (with-slots (pathname integer member) frame
  (fresh-line stream)
  (setq pathname (clim:accept 'pathname
                :prompt "A pathname" :default pathname
                :stream stream))
  (fresh-line stream)
  (setq integer (clim:accept 'integer
                :prompt "An integer" :default integer
                :stream stream))
  (fresh-line stream)
  (setq member (clim:accept '(member :one :two :three)
               :prompt "One, Two, or Three" :default member
               :stream stream))
  (fresh-line stream)
  (clim:accept-values-command-button (stream :documentation "You wolf")
    (write-string "Wolf whistle" stream)
   (beep))))
```

This function will start up a new "dingus" application. The argument is a port, as might be returned by **`find-port`**.

```
(defun run-dingus (port)
 (let ((dingus (clim:make-application-frame 'dingus
         :parent port :width 400 :height 400)))
  (clim:run-frame-top-level dingus)))
```

All this application does is allow the user to alter the values of the three application slots `pathname`, `integer` and `member` using the `avv` pane. The new values will automatically be reflected in the `display` pane.

9.5.2 Example of constructing a function as part of running an application

You can supply an alternate top level (which initializes some things and then calls the regular top level) to construct a function as part of running the application. Note that when you use this technique, you can close the function over other pieces of the Lisp state that might not exist until application runtime.

```
(clim:define-application-frame different-prompts ()
  ((prompt-state ...) ...)
 (:top-level (different-prompts-top-level))
 ...)

(defmethod different-prompts-top-level
      ((frame different-prompts) &rest options)
 (flet ((prompt (stream frame)
      (with-slots (prompt-state) frame
       ...)))
  (apply #'clim:default-frame-top-level
      frame :prompt #'prompt options)))
```

9.6 CLIM application frame accessors

The following functions may be used to access or modify the state of the application frame itself. Information available includes what the currently exposed panes are, what the current layout is, what command table is being used, and so forth. Functions are provided for moving top-level frames (**`position-sheet-carefully`**) as well as raising, burying, destroying, etc. frames.

`*application-frame*` **[Variable]**

■ The current application frame. The value is CLIM's default application. This variable is typically used in the bodies of commands and translators to gain access to the state variables of the application, usually in conjunction with **`with-slots`** or **`slot-value`**.

■ This variable is bound by an `:around` method of **`run-frame-top-level`** on **`application-frame`**. You should not rebind it, since CLIM depends on its value.

`with-application-frame` **[Macro]**

Arguments: `(`*`frame`*`) &body` *`body`*

■ This macro provides lexical access to the current frame for use with commands and the `:pane`, `:panes`, and `:layouts` options. *`frame`* is bound to the current frame within the context of one of those options.

■ *`frame`* is a symbol; it is not evaluated.

`map-over-frames` **[Function]**

Arguments: *`function &key port frame-manager`*

■ Applies the function *`function`* to all of the application frames that match *`port`* and *`frame-manager`*. If neither *`port`* nor *`frame-manager`* is supplied, all frames are considered to match. If *`frame-manager`* is supplied, only those frames that use that frame manager match. If *`port`* is supplied, only those frames that use that port match.

■ *`function`* is a function of one argument, the frame. It has dynamic extent.

destroy-frame [Generic function]

Arguments: *frame*

■ Destroys the application frame *frame*.

raise-frame [Generic function]

Arguments: *frame*

■ Raises the application frame *frame* to be on top of all of the other host windows by invoking **raise-sheet** on the frame's top-level sheet.

bury-frame [Generic function]

Arguments: *frame*

■ Buries the application frame *frame* to be underneath all of the other host windows by invoking **bury-sheet** on the frame's top-level sheet.

position-sheet-carefully [Function]

Arguments: *sheet x y*

■ *sheet* should be a top-level frame (i.e. a window). *sheet* is positioned so that its top left corner is located at (*x*,*y*), which are in screen coordinates.

frame-name [Generic function]

Arguments: *frame*

■ Returns the name, which is a symbol, of the application *frame*. You can change the name of an application frame by using **setf** on **frame-name**.

frame-pretty-name [Generic function]

Arguments: *frame*

■ Returns the pretty name, which is a string, of the application *frame*. You can change the pretty name of an application frame by using **setf** on **frame-pretty-name**.

frame-state [Generic function]

Arguments: *frame*

■ Returns one of `:disowned`, `:enabled`, `:disabled`, or `:shrunk`, indicating the current state of frame. `:disowned` means that no frame manager owns the frame. `:enabled` means the frame is currently enabled and visible on some port. `:disabled` means that the frame is owned by a frame manager, but is not visible anywhere. `:shrunk` means that the frame has been iconified.

frame-standard-input [Generic function]

Arguments: *frame*

■ Returns the value that should be used for `*standard-input*` for *frame*.

■ The default method (defined on **application-frame**) uses the first pane of type `:interactor`. If there are no interactor panes, the value returned by **frame-standard-output** is used.

default-frame-top-level calls this to determine what to bind `*standard-input*` to.

You will often implement a method for this generic function for an application frame, since CLIM cannot always reliably determine which pane to use for `*standard-input*`.

`frame-standard-output` [Generic function]

Arguments: *frame*

■ Returns the value that should be used for `*standard-output*` for *frame*.

The default method (defined on **`application-frame`**) uses the first pane of type `:application` in the current layout.

`default-frame-top-level` calls this to determine what to bind `*standard-output*` to.

You will often implement a method for this generic function for an application frame, since CLIM cannot always reliably determine which pane to use for `*standard-output*`.

`frame-error-output` [Generic function]

Arguments: *frame*

■ Returns the value that should be used for `*error-output*` for *frame*.

The default method (defined on **`application-frame`**) uses the first pane of type `:application` in the current layout.

`default-frame-top-level` calls this to determine what to bind `*error-output*` to.

`frame-query-io` [Generic function]

Arguments: *frame*

■ Returns the value that should be used for `*query-io*` for *frame*.

The default method (defined on **`application-frame`**) first tries to use the value returned by **`frame-standard-input`**, and if it is `nil`, it uses the value returned by **`frame-standard-output`**.

`default-frame-top-level` calls this to determine what to bind `*query-io*` to.

`*pointer-documentation-output*` [Variable]

■ This will be bound either to `nil`, or to a stream on which pointer documentation should be displayed.

`frame-pointer-documentation-output` [Generic function]

Arguments: *frame*

■ Returns the value that should be used for `*pointer-documentation-output*` for *frame*.

The default method (defined on **`application-frame`**) uses the first pane of type `:pointer-documentation` in the current layout.

`default-frame-top-level` calls this to determine what to bind `*pointer-documentation-output*` to.

`frame-current-layout` [Generic function]

Arguments: *frame*

■ Returns the name of the current layout for *frame*. Use `(setf frame-current-layout)` to change the current layout.

frame-current-panes [Generic function]

Arguments: *frame*

■ Returns a list of all of the named panes that are contained in the current layout for the frame *frame*. The elements of the list will be pane objects.

frame-panes [Generic function]

Arguments: *frame*

■ Returns a pane that represents that top level pane in the current layout.

get-frame-pane [Generic function]

Arguments: *frame pane-name* &key (errorp t)

■ Returns the pane named by *pane-name* in the current layout for *frame*. This is the recommended way to find the stream associated with a pane.

If the pane is not found in the current layout and *errorp* is `t`, and error is signaled. Otherwise if *errorp* is `nil`, the returned value will be `nil`.

find-pane-named [Generic function]

Arguments: *frame pane-name* &key (errorp t)

■ Returns the pane named by *pane-name* in the current layout for *frame*.

If the pane is not found in the current layout and *errorp* is `t`, and error is signaled. Otherwise if *errorp* is `nil`, the returned value will be `nil`.

frame-command-table [Generic function]

Arguments: *frame*

■ Returns the name of the command table currently being used by the frame *frame*. You can use this function with **setf** to change the command table to be used.

frame-find-innermost-applicable-presentation [Generic function]

Arguments: *frame input-context stream x y*

■ Locates and returns the innermost applicable presentation on the window *stream* at the pointer position indicated by *x* and *y*, in the input context *input-context*, on behalf of the application frame *frame*.

You can specialize this generic function for your own application frames. The default method calls **find-innermost-applicable-presentation**.

frame-input-context-button-press-handler [Generic function]

Arguments: *frame stream button-press-event*

■ This function is responsible for handling user pointer gestures on behalf of *frame*. *stream* is the window on which *button-press*-event took place.

The default method calls **frame-find-innermost-applicable-presentation** to find the innermost applicable presentation, and then calls **throw-highlighted-presentation** to execute the translator that corresponds to the user's gesture.

frame-maintain-presentation-histories [Generic function]

Arguments: *frame*

■ Returns `t` if the *frame* maintains histories for its presentations, otherwise returns `nil`. The default method on the class `standard-application-frame` returns `t` if and only if the frame has an interactor pane.

You can specialize this generic function for your own application frames.

frame-top-level-sheet [Generic function]

Arguments: *frame*

■ Returns the window that corresponds to the top level window for the frame *frame*. This is the window that has as its children all of the panes of the frame.

frame-document-highlighted-presentation [Generic function]

Arguments: *frame presentation input-context window x y stream*

■ This generic function is called to output the pointer documentation to *stream* for a presentation *presentation* in the application-frame *frame*. *x* and *y* are the mouse co-ordinates in *window*. *input-context* is the current input-context of the particular frame.

Frame iconification/deiconification

note-frame-deiconified [Generic function]

Arguments: *frame-manager frame*

■ This generic function is called whenever the frame-state of frame changes from :shrunk. Calling this function will force the state to change from :shrunk to :enabled. This function should only be called on frame's whose state is either :shrunk or :enabled.

note-frame-iconified [Generic function]

Arguments: *frame-manager frame*

■ This generic function is called whenever the frame-state of *frame* changes to `:shrunk`. Calling this function will force the state to change to :shrunk. This function should only be called on frame's whose state is either `:shrunk` or `:enabled`.

9.7 Operators for running CLIM applications

The following functions are used to start up an application frame, exit from it, and control the read-eval-print loop of the frame (for example, redisplay the panes of the frame, and read, execute, enable, and disable commands).

run-frame-top-level [Generic function]

Arguments: *frame* `&key &allow-other-keys`

■ Runs the top-level function for *frame*. The default method merely runs the top-level function of *frame* as specified by the `:top-level` option of **define-application-frame**, passing along any keyword arguments to the top level function. If `:top-level` was not supplied, **default-frame-top-level** is used.

application-frame provides an `:around` method which binds `*application-frame*` to *frame*.

`default-frame-top-level` [Generic function]

Arguments: *frame* `&key command-parser command-unparser partial-command-parser (prompt "Command: ")`

■ The default top-level function for application frames. This function implements a read-eval-print loop that calls **`read-frame-command`**, then calls **`execute-frame-command`**, and finally redisplays all of the panes that need to be redisplayed.

Note that the source for this function is in the file *default-frame-top-level.lisp* in the CLIM demos, which are in the *src/clim/demo/* directory in the distribution.

`default-frame-top-level` establishes a simple restart for `abort`, so that anything that invokes an `abort` restart will by default throw to the top level command loop of the application frame. (Of course, the programmer can specify a `restart-case` for the `abort` restart.)

`default-frame-top-level` binds several of Lisp's standard stream variables. `*standard-output*` is bound to the value returned by **`frame-standard-output`**. `*standard-input*` is bound to the value returned by **`frame-standard-input`**. `*query-io*` is bound to the value returned by **`frame-query-io`**.

`prompt` controls the prompt. You can supply either a string or a function of two arguments (*`stream`* and *`frame`*) that outputs the prompt on the stream. The default for *`prompt`* is the string "Command: ". To set your own prompt string supply `:prompt` to the `:top-level` option of **`define-application-frame`**:

```
(clim:define-application-frame different-prompt ()
  (...)
 (:top-level (clim:default-frame-top-level
        :prompt "What next, mate? "))
```

If you want the prompt to change as a function of the state of the application, you can supply a function (instead of a string):

```
(defun promptfun (stream frame)
 (with-slots (prompt-state) frame
  (format stream "Prompt ~D: " prompt-state)))

(clim:define-application-frame different-prompts ()
  ((prompt-state ...) ...)
 (:top-level (clim:default-frame-top-level
        :prompt promptfun))
 ...)
```

If there is an interactor pane in the frame, *`command-parser`* defaults to **`command-line-command-parser`**, *`command-unparser`* defaults to **`command-line-command-unparser`**, and *`partial-command-parser`* defaults to **`command-line-read-remaining-arguments-for-partial-command`**. If there is no interactor pane, *`command-parser`* defaults to **`menu-command-parser`** and *`partial-command-parser`* defaults to **`menu-read-remaining-arguments-for-partial-command`**; there is no need for an unparser when there is no interactor. The frame's top level loop binds `*command-parser*`, `*command-unparser*`, and `*partial-command-parser*` to the values of *`command-parser`*, *`command-unparser`*, and *`partial-command-parser`*.

frame-exit [Generic function]

Arguments: *frame*

■ Exits from the application frame *frame* by signalling a `frame-exit` condition. The condition's frame slot will have *frame* in it.

■ In the current implementation, you must call **frame-exit** on *frame* from the process running the frame-top-level of frame. Calling **frame-exit** from a different process behaves as a no-op.

■ Window managers may provide some way to 'close' or 'exit' a window, often from a window-manager-supplied menu. When this window-manager option is chosen, **frame-exit** is called and so the window-manager choice is typically equivalent to calling **frame-exit** directly.

■ Note that the action of **frame-exit** is similar to a **throw**. One result of this fact is that you cannot do anything to the frame after **frame-exit** is called, either after the call to **frame-exit** or in an `:after` method for **frame-exit**. Everything you want to do to the frame must be done before the call to **frame-exit**. Also, if you are tracing **frame-exit**, you will not see it return. This is expected behavior.

`frame-exit` [Condition]

■ The condition signaled by **frame-exit**.

frame-exit-frame [Generic function]

Arguments: *frame-exit*

■ Returns the application frame object that signaled the `frame-exit` condition.

redisplay-frame-pane [Generic function]

Arguments: *frame pane-name* `&key force-p`

■ Causes the pane *pane-name* of *frame* to be redisplayed immediately. If *force-p* is `t`, then the pane is forcibly redisplayed even if it is an incrementally redisplayed pane that would not otherwise require redisplay.

redisplay-frame-panes [Generic function]

Arguments: *frame* `&key force-p`

■ Causes all of the panes of *frame* to be redisplayed immediately. If *force-p* is `t`, then the panes are forcibly redisplayed even if they are incrementally redisplayed panes that would not otherwise require redisplay.

frame-replay [Generic function]

Arguments: *frame stream* `&optional` *region*

■ Replays all of the output records in *stream*'s output history on behalf of the application frame *frame* that overlap the region *region*. If *region* is not supplied, all of the output records that overlap the viewport are replayed.

■ You can specialize this generic function for your own application frames. The default method for this calls **stream-replay**.

frame-current-layout [Generic function]

Arguments: *frame*

■ Returns a symbol naming the current layout of *frame*.

`(setf frame-current-layout)` [Generic function]

Arguments: *frame new-layout*

■ Sets the layout of *frame* to be the new layout named *new-layout*. This is thy say that other layouts are selected and displayed.

■ Note: **`(setf frame-current-layout)`** throws out of the application's command loop, all the way back to **`run-frame-top-level`**. This is done so that CLIM can perform some window management functions, such as rebinding I/O streams that correspond to the windows in the new layout. Therefore, when you call **`(setf frame-current-layout)`**, you should only do so *after* you have done everything else in the sequence of operations.

`frame-all-layouts` [Generic function]

Arguments: *frame*

■ Returns a list of all the layout names for *frame*.

[This page intentionally left blank.]

Chapter 10 Commands in CLIM

10.1 Introduction to CLIM commands

In CLIM, users interact with applications through the use of commands. A command is an object that represents one interaction with a user that results in some operation being performed in an application.

Commands are read and executed by the command loop. CLIM's command loop accepts input of presentation type `command` and then executes the accepted command. This chapter discusses how commands are represented.

CLIM supports four main styles of command interaction. It is important to note that the choice of interaction styles is *independent* of the command loop or the set of commands. The relationship between a user's interactions and the commands to be executed is governed by command tables.

- Mouse interaction via command menus or dialogs. A command is invoked by clicking on an item in a menu, or by filling in the fields of a dialog.
- Keyboard interaction using keystroke accelerators. A single keystroke invokes the associated command.
- Mouse interaction via command translators. A command can be invoked by clicking on any object displayed by the interface. The particular combination of mouse-buttons and modifier keys (e.g. shift, control) is called a gesture. As part of the presentation system, a command translator turns a gesture on an object into a command. Drag and drop translators are a special case of more general translators, and can be used to implement Macintosh-desktop-like interfaces.
- Keyboard interaction using a command-line processor. The user types a complete textual representation of command names and arguments. The text is parsed by the command-line processor to form a command. A special character (usually newline) indicates to the command-line processor that the text is ready to be parsed.

A *command table* is an object that serves to mediate between a command input context (e.g., the top level of an application frame), a set of commands, and these interaction styles.

Commands may take arguments, which are specified by their presentation types.

For simple CLIM applications, **`define-application-frame`** will automatically create a command table, a top level command input context, and define a command defining macro for you.

Following a discussion of the simple approach, this chapter discusses command tables and the command processor in detail. This information is provided for the curious and for those who feel they require further control over their application's interactions. These are some circumstances which might suggest something beyond the simple approach:

- Your application requires more than one command table, for example, if it has multiple modes with different sets of commands available in each mode.
- If you have sets of commands which are common among several modes or even among several applications, you could use several command tables and inheritance to help organize your command sets.

- Your application may be complex enough that you may want to develop more powerful tools for examining and manipulating command tables.

If you do not require this level of detail, then you can just read section 10.2 **Defining Commands the Easy Way** and skip the remainder of this chapter.

10.2 Defining commands the easy way

CLIM provides utilities to make it easy to define commands for most applications. **`define-application-frame`** will automatically create a command table for your application. This behavior is controlled by the `:command-table` option. It will also define a command defining macro which you will use to define the commands for your application. This is controlled by the `:command-definer` option.

This command definer macro will behave similarly to **`define-command`**, but will automatically use your application's command table so you needn't supply one.

Here is an example code fragment illustrating the usage of **`define-application-frame`** which defines an application named `editor`. A command table named `editor-command-table` is defined to mediate the user's interactions with the `editor` application. It also defines a macro named **`define-editor-command`** which the application programmer will use to define commands for the `editor` application and install them in the command table `editor-command-table`.

```
(clim:define-application-frame editor ()
  ()
 (:command-table editor-command-table)
 (:command-definer define-editor-command)
 ...)
```

Note that for this particular example, the `:command-table` and `:command-definer` options need not be supplied since the names that they specify would be the ones which would be generated by default. These options normally are provided only when you want different names other than the default ones, you don't want a command definer or you want to specify which command tables the application's command table inherits from. See the chapter 9 **Defining application frames in CLIM** and see the macro **`define-application-frame`** for a description of these options.

10.2.1 Command names and command line names

Every command has a *command name*, which is a symbol. The symbol names the function which implements the command. The body of the command is the function definition of that symbol.

By convention, commands are named with a "com-" prefix, although CLIM does not enforce this convention.

To avoid collisions among command names, each application should live in its own package; for example, there might be several commands named `com-show-chart` defined for each of a spreadsheet, a navigation program, and a medical application.

CLIM supports a *command line name* which is separate from the command's actual name. For command line interactions, the end user sees and uses the command line name. For example, the command `com-show-chart` would have a command line name of "Show Chart". When defining a command using **`define-command`** (or the application's command defining macro), you can have a command line name generated automatically.

The automatically generated command line name consists of the command's name with the hyphens replaced by spaces, and the words capitalized; furthermore, if there is a prefix of "com-", the prefix is removed. For example, if the command name is `com-show-file`, the command-line name will be "Show File".

The **`define-editor-command`** macro, which would automatically be generated by the above example fragment, is used to define a command for the `editor` application. **`define-editor-command`** is used in the same way as **`define-command`**. However, rather than requiring that the programmer supply `editor-command-table` as the command table in which to define the command, **`define-editor-command`** will automatically use `editor-command-table`.

Through the appropriate use of the options to **`define-editor-command`** (the same options as for **`define-command`**), the programmer can provide the command via any number of the above mentioned interaction styles. For example, he could install the command in the `editor` application's menu as well as specify a single keystroke command accelerator character for it.

This example defines a command whose command name is `com-save-file`. The `com-save-file` command will appear in the application's command menu, by the name "Save File" (which is automatically generated from the command name based on the same method as for command line names). The single keystroke control-S will also invoke the command.

```
(define-editor-command (com-save-file :menu t
                    :keystroke (:s :control))
  ()
 ...)
```

Here, a command line name of "Save File" is associated with the `com-save-file` command. The user can then type ‘Save File’ to the application's interaction pane to invoke the command.

```
(define-editor-command (com-save-file :name "Save File")
  ()
 ...)
```

Since the command processor works by establishing an input context of presentation type `command` and executing the resulting input, any displayed presentation can invoke a command so long as there is a translator defined which translates from the presentation type of the presentation to the presentation type `command`. By this mechanism, the programmer can associate a command with a pointer gesture when applied to a displayed presentation. **`define-presentation-to-command-translator`** will do this; see the documentation for this in 8.7.2 **CLIM Operators for defining presentation translators**.

10.3 Command objects in CLIM

A *command* is an object that represents a single user interaction. Each command has a *command name*, which is a symbol. A command can also have arguments, both positional and keyword arguments.

CLIM represents commands as *command objects*. The internal representation of a command object is a cons of the command name and a list of the command's arguments and is therefore analogous to a Lisp expression. Functions are provided for extracting the command name and the arguments list from a command object:

`command-name` [Function]

Arguments: *`command`*

- Given a command object *`command`*, returns the command name.

command-arguments [Function]

Arguments: *command*

- Given a command object *command*, returns the command's arguments.

partial-command-p [Function]

Arguments: *command*

- It is possible to represent a command for which some of the arguments have not yet been supplied. The value of the symbol `*unsupplied-argument-marker*` is used in place of any argument which has not yet been supplied.
- **partial-command-p** returns `t` if *command* is a partial command.

One can think of **define-command** as defining templates for command objects. It defines a symbol as a command name and associates with it the presentation types corresponding to each of the command's arguments.

define-command [Macro]

Arguments: *name arguments* &body *body*

- Defines a command and characteristics of the command, including its name, its arguments, and, as options: the command table in which it should appear, its keystroke accelerator, its command-line name, and whether or not (and how) to add this command to the menu associated with the command table.

```
(clim:define-command (com-my-favorite-command
            :name "My Favorite"
            :keystroke (:f)
            :menu "My Fave"
            :command-table my-command-table)
 ((arg1 (or integer string)
     :default "none"
     :display-default t))
  body)
```

This is the most basic command-defining form. Usually, the programmer will not use **define-command** directly, but will instead use a **define**-*application*-**command** form that is automatically generated by **define-application-frame**. **define**-*application*-**command** adds the command to the application's command table. By default, **define-command** does not add the command to any command table.

- **define-command** defines two functions. The first function has the same name as the command name, and implements the body of the command. It takes as arguments the arguments to the command as specified by the **define-command** form, as required and keyword arguments.

  The second function defined by **define-command** implements the code responsible for parsing and returning the command's arguments.

name

Is either a command name, or a cons of the command name and a list of keyword-value pairs. The keyword-value pairs in *name* can be:

`:command-table` *command-table-name*

Specifies that the command should be added to a command table. *command-table-name* either names a command table to which the command should be added, or is `nil`

(the default) to indicate that the command should not be added to any command table. This keyword is only accepted by **define-command**, not by **define-***application*-**command** functions.

`:name` *`string`*

Provides a name to be used as the command-line name for the command for keyboard interactions in the command table specified by the `:command-table` option. *`string`* is a string to be used; or `nil` (the default) meaning that the command will not be available via command-line interactions; or `t`, which means the command-line name will be generated automatically. See the function **add-command-to-command-table**.

`:menu` *`menu-item`*

Specifies that this command will be an item in the menu of the command table specified by the `:command-table` option. The default is `nil`, meaning that the command will not be available via menu interactions. If *`menu-item`* is a string, then that string will be used as the menu name. If *`menu-item`* is `t`, then the menu name will be generated automatically. See the function **add-command-to-command-table**. Otherwise, *`menu-item`* should be a cons of the form (*`string`* . *`menu-options`*), where *`string`* is the menu name and *`menu-options`* consists of keyword-value pairs. The valid keywords are `:after` and `:documentation`, which are interpreted as for **add-menu-item-to-command-table**.

`:keystroke` *`gesture`*

Specifies a gesture to be used as a keystroke accelerator in the command table specified by the `:command-table` option. For applications with interactor panes, these gestures typically correspond to non-printing characters such as Control-D. The default is `nil`, meaning that there is no keystroke accelerator.

The `:name`, `:menu`, and `:keystroke` options are allowed only if the `:command-table` option was supplied explicitly or implicitly, as in **define-***application-command*.

If the command takes any non-keyword arguments and you have supplied either `:menu` or `:keystroke`, then when you select this command via a command menu or keystroke accelerator, a partial command parser will be invoked in order to read the unsupplied arguments; the defaults will not be filled in automatically. If this behavior is not desired, then you must call **add-menu-item-to-command-table** or **add-keystroke-to-command-table** yourself and fully specify the command. For example, use the following instead of supplying `:keystroke` for the `com-next-frame` command:

```
(define-debugger-command (com-next-frame :name t)
  ((nframes 'integer
    :default 1
    :prompt "number of frames"))
 (next-frame :nframes nframes))

(clim:add-keystroke-to-command-table
 'debugger '(:n :control) :command '(com-next-frame 1))
```

`arguments`

A list consisting of argument descriptions. A single occurrence of the symbol `&key` may appear in *`arguments`* to separate required command arguments from keyword arguments. Each argument description consists of a list containing a parameter variable, followed by a presentation type specifier, followed by keyword-value pairs. The keywords can be:

`:default` *value*

Provides a *value* which is the default that should be used for the argument, as for **accept**.

`:default-type` *type*

The same as for **accept**: If `:default` is supplied, then the `:default` and the `:default-type` are returned if the input is empty.

`:mentioned-default` *value*

Provides a *value* which is the default that should be used for the argument when a keyword is explicitly supplied via the command-line processor, but no value is supplied for it. `:mentioned-default` is allowed only for keyword arguments. This is most commonly provided for boolean keyword arguments; the typical use if `:default nil :mentioned-default t`, which means that the 'default default' for the boolean argument is `nil`, but the default becomes `t` when the user types the name of the keyword argument.

`:display-default` *boolean*

The same as for **accept**: When true, displays the default if one was supplied. When `nil`, the default is not displayed.

`:prompt` *string*

Provides a *string* which is a prompt to print out during command-line parsing, as for **accept**.

`:documentation` *string*

Provides a documentation string that describes what the argument is.

`:when` *form*

Provides a *form* that indicates whether this keyword argument is available. The *form* is evaluated in a scope where the parameter variables for the required parameters are bound, and if the result is `nil`, the keyword argument is not available. `:when` is allowed only on keyword arguments, and *form* can use only the values of required arguments (that is, it cannot use the values of any other keyword arguments).

`:gesture` *gesture-name*

Provides a *gesture-name* that will be used for a translator that translates from the argument to a command. The default is `nil`, meaning no translator will be written. `:gesture` is allowed only when the `:command-table` option was supplied to the command-defining form.

body

Provides the body of the command. It has lexical access to all of the command's arguments. If the body of the command needs access to the application frame, it should use `*application-frame*`. The returned values of body are ignored.

define-command arranges for the function that implements the body of the command to get the proper values for unsupplied keyword arguments.

name-and-options and *body* are not evaluated. In the argument descriptions, the parameter variable name is not evaluated, and everything else is evaluated at run-time when argument parsing reaches that argument, except that the value for `:when` is evaluated when parsing reaches the keyword arguments, and `:gesture` is not evaluated at all.

command-name-from-symbol [Function]

Arguments: *symbol*

■ Generates a string suitable for use as a command-line name from the symbol *symbol*. The string consists the symbol name with the hyphens replaced by spaces, and the words capitalized. If the symbol name is prefixed by 'com-', the prefix is removed. For example, if the symbol is `com-show-file`, the resulting string will be "Show File".

10.4 CLIM Command Tables

CLIM command tables are represented by instances of the CLOS class `command-table`. A *command table* serves to mediate between a command input context, a set of commands and the interactions of the application's user.

Command tables associate command names with command line names. Command line names are used in the command line interaction style. They are the textual representation of the command name when presented and accepted.

A command table can describe a menu from which users can choose commands. A command table can support keystroke accelerators for invoking commands.

A command table can have a set of presentation translators and actions, defined by **`define-presentation-translator`**, **`define-presentation-to-command-translator`**, and **`define-presentation-action`**. This allows the pointer to be used to input commands, including command arguments.

We say that a command is *present* in a command table when it has been added to that command table by being associated with some form of interaction. We say that a command is *accessible* in a command table when it is present in that command table or is present in any of the command tables from which that command table inherits.

`command-table` [Class]

■ The class that represents command tables.

command-table-name [Generic function]

Arguments: *command-table*

■ Returns the name of the command table *command-table*.

command-table-inherit-from [Generic function]

Arguments: *command-table*

■ Returns a list of all of the command tables from which *command-table* inherits. You can **`setf`** this in order to change the inheritance of *command-table*.

find-command-table [Function]

Arguments: *name* `&key (errorp t)`

■ Return the command table named by *name*. If *name* is itself a command table, it is returned. If the command table is not found and *errorp* is `t`, the `command-table-not-found` condition will be signaled.

define-command-table **[Macro]**

Arguments: *name* &key inherit-from menu inherit-menu

- Defines a command table whose name is the symbol *name*. The keyword arguments are:

inherit-from

A list of either command tables or command table names. The new command table inherits from all of the command tables specified by *inherit-from*. The inheritance is done by union with shadowing. In addition to inheriting from the explicitly specified command tables, every command table defined with **define-command-table** also inherits from CLIM's system command table. (This command table, global-command-table, contains such things as the "menu" translator that is associated with the right-hand button on pointers.)

menu

Specifies a menu for the command table. The value of *menu* is a list of clauses. Each clause is a list with the syntax (*string type value* &key *documentation keystroke*), where *string*, *type*, *value*, *documentation*, and *keystroke* are as in **add-menu-item-to-command-table** (defined in section 10.5.1 below)

inherit-menu

Normally, a menu does not inherit any menu items from its parents, but it can inherit menu items, keystrokes, or both. The possible values for this argument are :menu, :keystrokes, and t. When *inherit-menu* is :menu, menu items will be inherited; when it is :keystrokes, keystrokes are inherited; when it is t, both are inherited.

- If the command table named by *name* already exists, **define-command-table** will modify the existing command table to have the new value for *inherit-from* and *menu*, but will otherwise leave the other attributes for the existing table alone.
- None of the arguments to **define-command-table** arguments is evaluated.

make-command-table **[Function]**

Arguments: *name* &key inherit-from menu inherit-menu (errorp t)

- Creates a command table named *name* that inherits from *inherit-from* and has a menu specified by *menu*. *inherit-from* and *menu* are as in **define-command-table**. If the command table already exists and *error-p* is t, then a command-table-already-exists condition will be signaled.

A command table can inherit from other command tables. This allows larger sets of commands to be built up through the combination of smaller sets. In this way, a tree of command tables can be constructed. During command lookup, if a command is not found in the application's command table, then the command tables from which that command table inherits are searched also. It is only when the entire tree is exhausted that an error is signaled.

do-command-table-inheritance **[Macro]**

Arguments: (*command-table-var command-table*) &body *body*

- The macro **do-command-table-inheritance** is provided as a facility for programmers to walk over a command table and the command tables it inherits from in the proper precedence order.

  Successively executes *body* with *command-table*-var bound first to the command table *command-table*, and then to all of the command tables from which *command-table* inherits. The recursion follows a depth-first path, considering the inheritees of the first inheritee before considering the second inheritee. This is the precedence order for command table inheritance.

The following functions are provided for examining and altering the commands in a command table:

add-command-to-command-table [Function]

Arguments: *command-name command-table* &key name menu keystroke (errorp t)

■ Adds the command named by *command-name* to the command table *command-table*. *command-table* may be either a command table or a symbol that names a command table. The keyword arguments are:

name

The command-line name for the command, which can be `nil`, `t`, or a string. When it is `nil`, the command will not be available via command-line interactions. When it is a string, that string is the command-line name for the command. When it is `t`, the command-line name is generated automatically. The automatically generated command line name consists of the command's name with the hyphens replaced by spaces, and the words capitalized; furthermore, if there is a prefix of "com-", the prefix is removed. For example, if the command name is `com-show-file`, the command-line name will be "Show File".

For the purposes of command-line name lookup, the character case and style of *name* are ignored.

menu

A command menu item for the command, which can be `nil`, `t`, a string, or a cons. When it is `nil`, the command will not be available via menus. When it is a string, the string will be used as the menu name. When it is `t`, an automatically generated menu name will be used. When it is a cons of the form (*string* . *menu-options*), then *string* is the menu name and *menu-options* consists of keyword-value pairs. The valid keywords are `:after` and `:documentation`, which are interpreted as for **add-menu-item-to-command-table**.

keystroke

The value for *keystroke* is either a standard character, a gesture specification, or `nil`. When it is a standard character or gesture spec, that gesture is the keystroke accelerator for the command; otherwise the command will not be available via keystroke accelerators.

errorp

If the command is already present in the command table and *errorp* is `t`, the `command-already-present` condition will be signaled. When the command is already present in the command table and *errorp* is `nil`, then the old command will first be removed from the command table.

remove-command-from-command-table [Function]

Arguments: *command-name command-table* &key (errorp t)

■ Removes the command named by *command-name* from the command table *command-table*. *command-table* may be either a command table or a symbol that names a command table.

If the command is not present in the command table and *errorp* is `t`, the `command-not-present` condition will be signaled.

command-present-in-command-table-p **[Function]**

Arguments: *command-name command-table*

■ Returns `t` if *command-name* is present in *command-table*. A command is *present* in a command table when it has been added to that command table. A command is *accessible* in a command table when it is present in that command table or is present in any of the command tables from which that command table inherits.

command-accessible-in-command-table-p **[Function]**

Arguments: *command-name command-table*

■ If the command named by *command-name* is not accessible in *command-table*, then this function returns `nil`. Otherwise, it returns the command table in which the command was found. *command-table* may be either a command table or a symbol that names a command table.

map-over-command-table-commands **[Function]**

Arguments: *function*

Arguments: *command-table* `&key (inherited t)`

■ Applies *function* to all of the commands accessible in *command-table*. *function* should be a function that takes a single argument, the command name.

If *inherited* is `nil` instead of `t`, this applies *function* only to those commands present in *command-table*, that is, it does not map over any inherited command tables.

10.4.1 CLIM's predefined command tables

CLIM provides several command tables from which it is recommended that your application's command table inherit. These are the predefined command tables:

`global-command-table` **[Command table]**

■ The global command table from which all command tables inherit. For the most part, this command table contains only presentation translators needed by all CLIM applications, such as the identity translator (the translator that maps objects of any presentation type to themselves).

`user-command-table` **[Command table]**

■ A command table reserved for user-defined commands. This is the command table in which casual extensions should be inserted.

10.4.2 Conditions relating to CLIM command tables

Command table operations can signal these conditions:

`command-table-already-exists` **[Condition]**

■ This condition is signaled by **make-command-table** when you try to create a command table that already exists.

`command-table-not-found` **[Condition]**

■ This condition is signaled by functions such as **find-command-table** when the named command table cannot be found.

`command-not-present` **[Condition]**

■ A condition that is signaled when the command you are looking for is not present in the command table.

`command-not-accessible` **[Condition]**

■ A condition that is signaled when the command you are looking for is not accessible in the command table, for example, **`find-command-from-command-line-name`**.

10.5 Styles of interaction supported by CLIM

CLIM supports four main styles of interaction:

- Mouse interaction via command menus
- Mouse interaction via translators.
- Keyboard interaction using a command-line processor
- Keyboard interaction using keystroke accelerators

See the section 10.2 **Defining commands the easy way** for a simple description of how to use **`define-command`** to associate a command with any of these interaction styles.

The following sections provide descriptions of these interaction styles.

10.5.1 CLIM's Command Menu Interaction Style

Each command table may describe a menu consisting of an ordered sequence of command menu items. The menu specifies a mapping from a menu name (the name displayed in the menu) to either a command object or a submenu. The menu of an application's top-level command table may be presented in a window-system specific way, for example, as a menu bar, or in a `:menu` application frame pane.

These menu items are typically defined using the `:menu` option to **`define-command`** (or the application's command defining macro).

The following functions can be used to display a command menu in one of the panes of an application frame, or to choose a command from a menu.

`display-command-table-menu` **[Function]**

Arguments: *`command-table stream`* `&key max-width max-height n-rows n-columns x-spacing y-spacing (cell-align-x ':left) (cell-align-y ':top) (initial-spacing t) row-wise move-cursor`

■ Displays the menu for *`command-table`* on *`stream`*. This is not normally used in Motif based applications, since command table menus are generally displayed in a menu bar.

`max-width`

`max-height`

Specifies the maximum width, in device units, of the table display.

`n-rows`

Specifies the number of rows of the table. Specifying this overrides *`max-width`*.

`n-columns`

Specifies the number of columns of the table. Specifying this overrides *`max-height`*.

x-spacing

Determines the amount of space inserted between columns of the table; the default is the width of a space character. *x-spacing* can be specified in one of the following ways:

Integer

A size in the current units to be used for spacing.

String or character

The spacing is the width or height of the string or character in the current text style.

Function

The spacing is the amount of horizontal or vertical space the function would consume when called on the stream.

List of form (*number unit*)

The *unit* is `:point`, `:pixel`, or `:character`.

y-spacing

Specifies the amount of blank space inserted between rows of the table; the default is the vertical spacing for the stream. The possible values for this option are the same as for the *x-spacing* option.

cell-align-x

Specifies the horizontal placement of each of the cells in the command menu. This is like the `:align-x` option to **`formatting-cell`**.

cell-align-y

Specifies the horizontal placement of each of the cells in the command menu. This is like the `:align-y` option to **`formatting-cell`**.

move-cursor

When `t`, CLIM moves the cursor to the end of the table. The default is `t`.

`display-command-menu` [Function]

Arguments: *frame stream &key command-table max-width max-height n-rows n-columns (cell-align-x ':left) (cell-align-y ':top)*

■ Displays the menu described by the command table associated with the application frame *frame* onto *stream*. This is generally used as the display function for application panes of type `:command-menu`. Since Motif based applications usually use a menu bar, you will probably not use this very often.

command-table

Specifies the command table.

max-width

max-height

Specifies the maximum width and height, in device units, of the menu. The default for these is computed from the frame's layout.

n-rows

Specifies the number of rows of the table. Specifying this overrides *max-width*.

n-columns

Specifies the number of columns of the table. Specifying this overrides *max-height*.

cell-align-x

Specifies the horizontal placement of each of the cells in the command menu. This is like the `:align-x` option to **`formatting-cell`**.

cell-align-y

Specifies the horizontal placement of each of the cells in the command menu. This is like the `:align-y` option to **`formatting-cell`**.

■ Generally you will not need to supply *max-width*, *max-height*, *n-rows*, or *n-columns*, since CLIM is usually able to compute these.

`menu-choose-command-from-command-table` [Function]

Arguments: *command-table* `&key associated-window default-style label cache unique-id id-test cache-value cache-test`

■ Displays a menu of all of the commands in *command-table*'s menu, and waits for the user to choose one of the commands. The returned value is a command object. **`menu-choose-command-from-command-table`** can invoke itself recursively if there are sub-menus.

associated-window, *default-style*, *label*, *cache*, *unique-id*, *id-test*, *cache-value*, and *cache-test* are as for **`menu-choose`**.

A number of lower level functions for manipulating command menus are also provided:

`add-menu-item-to-command-table` [Function]

Arguments: *command-table string type value* `&key documentation (after :end) keystroke text-style (errorp t) button-type`

■ Adds a command menu item to *command-table*'s menu. The arguments are:

command-table

Can be either a command table or a symbol that names a command table.

string

The name of the command menu item. The character case and style of *string* are ignored. This is how the item will appear in the menu.

type

This is one of: `:command`, `:menu`, or `:divider`. (`:function`, called for in the CLIM spec, is not supported in this release.) When *type* is `:command`, *value* should be a command (a cons of a command name followed by a list of the command's arguments), or a command name. (When *value* is a command name, it behaves as though a command with no arguments was supplied.) In the case where all of the command's required arguments are supplied, clicking a command menu item invokes the command immediately. Otherwise, the user will be prompted for the remaining required arguments.

When *type* is `:menu`, this item indicates that a sub-menu will be invoked, and so *value* should be another command table or the name of another command table.

When *type* is `:divider`, some sort of divider (a non-sensitive item displaying *string* or a line) is displayed in the menu at that point. Which is determined by value, which can be `nil`, `:line`, or `:label`. `nil` and `:line` mean use a dividing line. (Note that if the look-and-feel provided by the underlying window system does not support dividing lines, `:divider` items with value `:line` or `nil` may be ignored.) A value of `:label` means use *string* to label a non-sensitive item displaying *string*. Allegro CLIM will draw a line or display *string*.

value

Meaning depends on the value of *type*, as described above.

documentation

A documentation string, which can be used as mouse documentation for the command menu item.

after

States where the command menu item should appear in the menu: either `:start`, `:end`, `nil`, a string, or `:sort`. `:start` means to add the new item to the beginning of the menu. A value of `:end` (the default) or `nil` means to add the new item to the end of the menu. A string naming an existing entry means to add the new item after that entry. If `:after` is `:sort`, then the item is inserted in such a way as to maintain the menu in alphabetical order.

keystroke

If supplied, the command menu item will be added to the command table's keystroke accelerator table. The value of *keystroke* should be a standard character or gesture spec. This is exactly equivalent to calling **add-keystroke-to-command-table** with the arguments *command-table*, *keystroke*, *type*, and *value*. When *keystroke* is supplied and *type* is `:command`, typing the accelerator character will invoke the command specified by *value*. When *type* is `:menu`, the command will continue to be read from the sub-menu indicated by *value* in a window system specific manner.

text-style

Allows you to specify the text style for any particular menu item.

errorp

If the item named by *string* is already present in the command table's menu and *errorp* is `t`, then the `command-already-present` condition will be signaled. When the item is already present in the command table's menu and *errorp* is `nil`, the old item will first be removed from the menu.

button-type

The value of this argument can be `nil` (the default) or `:help`. Has effect only if *type* is `:command` or `:menu` (`:button-type` is ignored when *type* has some other value). Certain window-systems treat help buttons specially. Specifying a button-type of `:help` allows the backend to display the particular menu-item in a way appropriate for help buttons for that backend's look and feel.

Currently this only effects the Motif backend where it causes the menu-item to be displayed to the right of the menu-bar.

remove-menu-item-from-command-table [Function]

Arguments: *command-table string &key (errorp t)*

■ Removes the item named by *string* from *command-table*'s menu. *command-table* may be either a command table or a symbol that names a command table.

If the command menu item is not present in the command table's menu and *errorp* is `t`, then the `command-not-present` condition will be signaled.

This function ignores the character case and style of the command menu item's name when searching through the command table's menu.

map-over-command-table-menu-items [Function]

Arguments: *function command-table*

■ Applies *function* to all of the menu items in *command-table*'s menu. *function* should be a function of three arguments, the menu name, the keystroke accelerator character (which will be `nil` if there is none), and the menu item. The menu items are mapped in the order specified by **add-menu-item-to-command-table**.

map-over-command-table-menu-items does not descend into sub-menus. If you require this behavior, you should examine the type of the menu item to see if it is `:menu` and make the recursive call from *function*.

find-menu-item [Function]

Arguments: *menu-name command-table* &key (errorp t)

■ Given a *menu-name* and a *command-table*, return two values, the menu item and the command table in which it was found. If the command menu item is not present in *command-table* and *errorp* is `t`, then the `command-not-accessible` condition will be signaled. *command-table* may be either a command table or a symbol that names a command table.

command-menu-item-type [Function]

Arguments: *item*

■ Returns the type of the command menu item *item*. This will be one of `:command`, `:function`, `:menu`, or `:divider`.

command-menu-item-value [Function]

Arguments: *item*

■ Returns the value of the command menu item *item*. For example, if the type of *item* is `:command`, this will return a command or a command name.

command-menu-item-options [Function]

Arguments: *item*

■ Returns a list of the options for the command menu item *item*.

10.5.2 Mouse interaction via presentation translators

A command table maintains a database of presentation translators. A presentation translator translates from its *from* presentation type to its *to* presentation type when its associated gesture (e.g. clicking a mouse button) is input. A presentation translator is triggered when its *to* presentation *type* matches the input context and its *from presentation* type matches the presentation type of the displayed presentation (the appearance of one of your application's objects on the display) on which the gesture is performed.

define-presentation-to-command-translator can be used to associate a presentation and a gesture with a command to be performed on the object which the presentation represents.

Translators can also be used to translate from an object of one type to an object of another type based on context. For example, consider an computer aided design system for electrical circuits. You might have a translator which translates from a resistor object to the numeric value of its resistance. When asked to enter a resistance (as an argument to a command or for some other query), the user could click on the presentation of a resistor to enter its resistance.

For a discussion of the facilities supporting the mouse translator interaction style, see the chapter 6 **Presentation types in CLIM**, especially **define-presentation-to-command-translator**

10.5.3 CLIM's command line interaction style

One interaction style supported by CLIM is the command line style of interaction provided on most conventional operating systems. A command prompt is displayed in the application's `:interactor` pane. The user enters a command by typing its command line name, followed by its arguments. What the user types (or enters via the pointer) is echoed to the interactor window. When the user has finished typing the command, it is executed.

In CLIM, this interaction style is augmented by the input editing facility which allows the user to correct typing mistakes (see the section 17.1 **Input editing and built-in keystroke commands in CLIM**) and by the prompting and help facilities, which provide a description of the command and the expected arguments. Command entry is also facilitated by the presentation substrate which allows the input of objects matching the input context, both for command names and command arguments.

See the chapter 8 **Presentation Types in CLIM** for a detailed description.

`find-command-from-command-line-name` [Function]

Arguments: *`name command-table`* `&key (errorp t)`

■ Given a command-line name *`name`* and a *`command-table`*, this function returns two values, the command name and the command table in which the command was found. If the command is not accessible in *`command-table`* and *`errorp`* is `t`, the `command-not-accessible` condition will be signaled.

`name` is a command-line name. *`command-table`* may be either a command table or a symbol that names a command table.

`find-command-from-command-line-name` ignores character case and style.

This function is the inverse of **`command-line-name-for-command`**.

`command-line-name-for-command` [Function]

Arguments: *`command-name command-table`* `&key (errorp t)`

■ Returns the command-line name for *`command-name`* as it is installed in *`command-table`*. If the command is not accessible in *`command-table`* (or the command has no command-line name and *`errorp`* is `t`), then the `command-not-accessible` condition is signaled.

If the command does not have a command-line name in the *`command-table`* and *`errorp`* is `:create`, then the returned value will be an automatically created command-line name.

`command-table` may be either a command table or a symbol that names a command table.

This function is the inverse of **`find-command-from-command-line-name`**.

`map-over-command-table-names` [Function]

Arguments: *`function command-table`* `&key (inherited t)`

■ Applies *`function`* to all of the command-line names accessible in *`command-table`*. *`function`* should be a function of two arguments, the command-line name and the command name.

■ If *`inherited`* is `nil` instead of `t`, this applies *`function`* only to those command-line names present in *`command-table`*, that is, it does not map over any inherited command tables.

10.5.4 CLIM's keystroke interaction style

Each command table may have a mapping from keystroke accelerator characters to either command objects or submenus. This mapping is similar to that for menu items as the programmer might provide a single keystroke equivalent to a command menu item.

Note that the kinds of characters that can be typed in vary widely from one platform to another, so the programmer must be careful in choosing keystroke accelerator characters. Some sort of per-platform conditionalization is to be expected.

Keystroke accelerators will typically be associated with commands through the use of the `:keystroke` option to **`define-command`** (or the application's command defining macro).

`add-keystroke-to-command-table` [Function]

Arguments: *`command-table keystroke type value`* `&key documentation (errorp t)`

■ Adds a keystroke accelerator to the *`command-table`*.

`command-table`

Can be either a command table or a symbol that names a command table.

`keystroke`

The accelerator gesture. For applications that have an interactor pane, this will typically correspond to a non-printing character, such as control-D. For applications that do not have an interactor pane, *`keystroke`* can correspond to a printing character as well.

`type`

When *`type`* is `:command`, *`value`* should be a command (a cons of a command name followed by a list of the command's arguments), or a command name. (When *`value`* is a command name, it behaves as though a command with no arguments was supplied.) In the case where all of the command's required arguments are supplied, typing the keystroke invokes the command immediately. Otherwise, the user will be prompted for the remaining required arguments.

`value`

Meaning depends on the value of *`type`*, as described above.

`documentation`

A documentation string, which can be used as documentation for the keystroke accelerator.

`errorp`

If the command menu item associated with *`keystroke`* is already present in the command table's accelerator table and *`errorp`* is `t`, then the `command-already-present` condition will signaled. When the item is already present in the command table's accelerator table and *`errorp`* is `nil`, the old item will first be removed.

`remove-keystroke-from-command-table` [Function]

Arguments: *`command-table keystroke`* `&key (errorp t)`

■ Removes the item named by *`keystroke`* from *`command-table`*'s accelerator table. *`command-table`* may be either a command table or a symbol that names a command table.

If the command menu item associated with *`keystroke`* is not present in the command table's menu and *`errorp`* is `t`, then the `command-not-present` condition will be signaled.

map-over-command-table-keystrokes [Function]

Arguments: *function command-table*

■ Applies *function* to all of the keystroke accelerators in *command-table*'s accelerator table. *function* should be a function of three arguments, the menu name (which will be `nil` if there is none), the keystroke accelerator gesture, and the menu item.

map-over-command-table-keystrokes does not descend into sub-menus. If you require this behavior, you should examine the type of the menu item to see if it is `:menu`.

find-keystroke-item [Function]

Arguments: *keystroke command-table* &key test (errorp t)

■ Given a keystroke accelerator *keystroke* and a *command-table*, returns two values, the command menu item associated with the character and the command table in which it was found. (Since keystroke accelerators are not inherited, the second returned value will always be *command-table*.)

test specifies a function to use for looking up the items in the command table. It should be a function of two arguments, both characters. It defaults to **event-matches-gesture-name-p**.

If the keystroke accelerator is not present in *command-table* and *errorp* is `t`, then the `command-not-accessible` condition will be signaled. *command-table* may be either a command table or a symbol that names a command table.

lookup-keystroke-item [Function]

Arguments: *keystroke command-table* &key test

■ This is like **find-keystroke-item**, except that it descends into sub-menus in order to find a keystroke accelerator matching *keystroke*. If it cannot find any such accelerator, **lookup-keystroke-item** returns `nil`.

■ *test* is a function of two arguments used to compare the keystroke to the gestures in the command table. It defaults to **event-matches-gesture-name-p**.

lookup-keystroke-command-item [Function]

Arguments: *keystroke command-table* &key test (numeric-argument 1)

■ This is like **lookup-keystroke-item**, except that it searches only for enabled commands. If it cannot find an accelerator associated with an enabled command, **lookup-keystroke-command-item** returns `nil`.

This is the function you are most likely to call when you want to look up a keystroke for the purpose of finding a command to execute. **find-keystroke-item** and **lookup-keystroke-item** are intended more as bookkeeping functions.

test is a function of two arguments used to compare the keystroke to the gestures in the command table. It defaults to **event-matches-gesture-name-p**.

Because of the potential ambiguity between keystroke accelerators and normal typed input, the default CLIM command loop does not cope with keyboard accelerators unless you request it to explicitly.

To be able to use keystroke accelerators, your application will need to specialize the **read-frame-command** generic function. The default method for **read-frame-command** just calls **read-command**. You can specialize it to call **read-command-using-keystrokes** within the context of **with-command-table-keystrokes**:

```
(defmethod clim:read-frame-command ((frame my-application) &key)
 (let ((command-table (clim:find-command-table 'my-command-table)))
```

```
(clim:with-command-table-keystrokes (keystrokes command-table)
 (clim:read-command-using-keystrokes command-table keystrokes))))
```

with-command-table-keystrokes **[Macro]**

Arguments: (*keystroke-var command-table*) &body *body*

■ Binds *keystroke-var* to a list that contains all of the keystroke accelerator gestures in the command table *command-table*, and then executes *body* in that context.

```
(clim:with-command-table-keystrokes (keystrokes command-table)
 (let ((command (clim:read-command-using-keystrokes
          command-table keystrokes
          :stream command-stream)))
  (if (and command (not (characterp command)))
    (clim:execute-frame-command frame command)
    (clim:beep stream))))
```

■ Note that, in general, the keystroke accelerator gestures you choose should not be any characters that a user can normally type in during an interaction. That is, they will typically correspond to non-printing characters such as control-E.

This macro generates keystrokes suitable for use by **read-command-using-keystrokes**.

read-command-using-keystrokes **[Function]**

Arguments: *command-table keystrokes* &key stream command-parser command-unparser partial-command-parser

■ Reads a command from the user via the command lines, the pointer, or typing a single keystroke. It returns either a command object, or a character if the user typed a keystroke that is in *keystrokes* but does not have a keystroke associated with it in the command table.

command-parser, *command-unparser*, *partial-command*-parser default from *command-parser*, *command-unparser*, and *partial-command-parser*, which are bound by the application frame's top level loop.

keystrokes is a list of gestures. The other arguments are as for **read-command**.

See also **with-command-table-keystrokes**.

Note that if your application also employs the command line interaction style there is the potential for ambiguity as to whether a character is intended as command line input, a keystroke command or an input editing command (see the section 17.1 **Input editing and built-in keystroke commands in CLIM**). For this reason, it is recommended that you choose keystroke accelerator characters which do not conflict with the standard printed character set (which might be used for command names and the textual representations of arguments) or with the input editor. CLIM will make some attempt to resolve such conflicts if they arise. A keystroke accelerator can only be invoked if there is no other pending command line input. If there is pending input, keystroke accelerators will not be considered and the keystroke will be interpreted as input or as an input editor command. If there is no pending input, the keystroke accelerator behavior will take precedence over that of the rubout handler.

For a description of the CLIM command processor, see the section 10.6 **The CLIM command processor**.

10.6 The CLIM Command Processor

This section describes the default behavior of the CLIM command processor.

The command loop of a CLIM application is performed by the application's top-level function (see the section 9.2 **Defining CLIM application frames**). By default, this is **`default-frame-top-level`**. After performing some initializations, **`default-frame-top-level`** enters an infinite loop, reading and executing commands. It invokes the generic function **`read-frame-command`** to read a command which is then passed to the generic function **`execute-frame-command`** for execution. The specialization of these generic functions is the simplest way to modify the command loop for your application. Other techniques would involve replacing **`default-frame-top-level`** with your own top level function.

`read-frame-command` invokes the command parser by establishing an input context of `command`. The input editor keeps track of the user's input, both from the keyboard and the pointer. Each of the command's arguments is parsed by establishing an input context of the arguments presentation type as described in the command's definition. Presentation translators provide the means by which the pointer can be used to enter command names and arguments using the pointer.

`read-command` [Function]

Arguments: *`command-table`* `&key stream use-keystrokes command-parser command-unparser partial-command-parser`

■ Reads a command from the user via command lines or the pointer. This function is not normally called by programmers.

`command-table`

Specifies which command table's commands should be read.

`stream`

The stream from which to read the command.

`command-parser`

A function of two arguments, *`command-table`* and *`stream`*. This function should read a command from the user and return a command object. It defaults to the value of `*command-parser*`, which is bound by the application's top level loop.

`command-unparser`

A function of three arguments, *`command-table`*, *`stream`*, and *`command-to-unparse`*. The function should print a textual description of the command and the set of arguments supplied on *`stream`*. It defaults to the value of `*command-unparser*`, which is bound by the application's top level loop.

`partial-command-parser`

A function of four arguments, *`command-table`*, *`stream`*, *`partial-command`*, and *`start-position`*. A partial command is a command structure with `*unsupplied-argument-marker*` in place of any argument that remains to be filled in. The function should read the remaining arguments in any way it sees fit and should return a command object. *`start-position`* is the original input-editor scan position of *`stream`* if *`stream`* is an interactive stream. It defaults to the value of `*partial-command-parser*`, which is bound by the application's top level loop.

`use-keystrokes`

The default for this is `nil`. If it is `t`, **`read-command`** calls **`read-command-using-keystrokes`** to read the command. The keystroke accelerators are those generated by **`with-command-table-keystrokes`**.

read-frame-command [Generic function]

Arguments: *frame* &key stream

■ **read-frame-command** reads a command from the user on the stream *stream*, and returns the command object. *frame* is an application frame.

The default method for **read-frame-command** calls **read-command** on *frame*'s current command table. You can specialize this generic function for your own application frames, for example, if you want to have your application be able to read commands using keystroke accelerators, or you want a completely different sort of command loop.

execute-frame-command [Generic function]

Arguments: *frame command*

■ **execute-frame-command** executes the command *command* on behalf of the application frame *frame*.

The default method for **execute-frame-command** simply applies to command name to the command arguments.

An application can control which commands are enabled and which are disabled on an individual basis. Use **setf** on **command-enabled** to control this mechanism. The user is not allowed to enter a disabled command via any interaction style.

command-enabled [Generic function]

Arguments: *command-name frame*

■ Returns `t` if the command named by *command-name* is presently enabled in *frame*, otherwise returns `nil`. If *command-name* is not accessible to the command table being used by *frame*, **command-enabled** returns `nil`.

You can use **setf** on **command-enabled** in order to enable or disable a command.

unsupplied-argument-marker [Variable]

■ The value of `*unsupplied-argument-marker*` is an object that can be uniquely identified as standing for an unsupplied argument in a command object.

numeric-argument-marker [Variable]

■ The value of `*numeric-argument-marker*` is an object that can be uniquely identified as standing for a numeric argument in a command object. When possible, CLIM will replace occurrences of `*numeric-argument-marker*` in a command object with the numeric argument accumulated from the input editor.

For example, the following might come from some sort of Debugger. When the user types Control-5 Control-N, the Debugger moves down five frames in the stack.

```
(define-debugger-command (com-next-frame :name t)
  (&key (n-frames '((integer) :base 10)
          :default 1
          :documentation "Move this many frames")
     (detailed 'boolean
          :default nil :mentioned-default t
          :documentation "Show locals and disassembled code"))
 "Show the next frame in the stack"
 (cond ((and (plusp n-frames)
       (bottom-frame-p (current-frame clim:*application-frame*)))
```

```
     (format t "~&You are already at the bottom of the stack."))
    (t
     (show-frame (nth-frame n-frames) :detailed detailed))))

(add-keystroke-to-command-table 'debugger '(:n :control)
                :command
                '(com-next-frame :n-frames ,*numeric-argument-marker*))
```

The special variable `*command-dispatchers*` controls the behavior of the `command-or-form` presentation type.

`*command-dispatchers*` **[Variable]**

■ This is a list of characters that indicate that CLIM should read a command when CLIM is accepting input of type `command-or-form`. The default value for this is :).

10.7 Command-related Presentation Types

CLIM provides several presentation types pertaining to commands:

`command` **[Presentation type]**

Arguments: `&key command-table`

■ The presentation type used to represent a Command Processor command and its arguments. *`command-table`* can be either a command table or a symbol that names a command table.

If *`command-table`* is not supplied, it defaults to the command table for the current application, that is, `(frame-command-table *application-frame*)`.

When you call **`accept`** on this presentation type, the returned value is a list; the first element is the command name, and the remaining elements are the command arguments. You can use **`command-name`** and **`command-arguments`** to access the name and arguments of the command object.

For more information about CLIM command objects, see the section 10.3 **Command Objects in CLIM**.

`command-name` **[Presentation type]**

Arguments: *`command`* `&key` *`command-table`*

■ The presentation type used to represent the name of a Command Processor command in the command table *`command-table`*.

■ *`command-table`* may be either a command table or a symbol that names a command table. If *`command-table`* is not supplied, it defaults to the command table for the current application. The textual representation of a `command-name` object is the command-line name of the command, while the internal representation is the command name.

`command-or-form` **[Presentation type]**

Arguments: `&key command-table`

■ The presentation type used to represent either a Lisp form or a Command Processor command and its arguments. In order for the use to indicate that he wishes to enter a command, a command dispatch character must be typed as the first character of the command line. See the variable `*command-dispatchers*`.

■ *command-table* may be either a command table or a symbol that names a command table. If *command-table* is not supplied, it defaults to the command table for the current application, that is, `(frame-command-table *application-frame*)`.

Chapter 11 Formatted output in CLIM

11.1 Formatted output in CLIM

CLIM provides a variety of high-level formatted output facilities, including table formatting, graph formatting, output filling, and others.

11.2 Concepts of CLIM table and graph formatting

CLIM makes it easy to construct tabular output and graph output. The usual way of making table or a graph is by indicating what you want to put in the table or graph, and letting CLIM choose the placement of the cells. CLIM allows you to specify constraints on the placement of the cells with some flexibility.

In the CLIM model of table and graph formatting, each cell is handled separately. You write code that puts ink on a drawing plane. That ink might be text, graphics, or both. CLIM surrounds all the ink with a bounding box (or, more precisely, an axis-aligned rectangle). That bounding box is snipped out of the drawing plane and placed in a cell of the table or graph. In the case of table formatting, CLIM's formatting engine puts whitespace around the ink to make sure that all the cells in a row are the same height, and all the cells in a column are the same width. In the case of graph formatting, the cells of the graph are laid out according to some common graphical algorithms.

In both table and graph formatting, you are responsible only for supplying the contents of the cell. CLIM's formatting engines are responsible for figuring out how to lay out the table or graph so that all the cells fit together properly.

Also in both types of formatting, you can specify other constraints that affect the appearance of the table (such as, the spacing between rows or columns, or the width or length of the table).

11.2.1 Formatting item lists in CLIM

Table formatting is inherently two-dimensional from the point of view of the application. Item list formatting is inherently one-dimensional output that is presented two-dimensionally. The canonical example is a menu, where the programmer supplies a list of items to be presented, where a single column or row of menu entries would be fine (if the list is small enough). In this case, formatting is done when viewport requirements make it desirable.

These constraints affect the appearance of item lists:

- The number of rows (allowing CLIM to choose the number of columns)
- The number of columns (allowing CLIM to choose the number of rows)

- The maximum height (or width) of the column (letting CLIM determine the number of rows and columns that satisfy that constraint)

11.3 CLIM Operators for Table Formatting

This section summarizes the CLIM operators for table formatting.

These are the general-purpose table formatting operators:

`formatting-table` **[Macro]**

Arguments: `(&optional stream &rest options &key x-spacing y-spacing multiple-columns multiple-columns-x-spacing equalize-column-widths (move-cursor t) record-type) &body body`

■ Establishes a table formatting environment on the *`stream`*; the default for *`stream`* is `*standard-output*`). All output performed within the extent of this macro will be displayed in tabular form. This must be used in conjunction with **`formatting-row`** or **`formatting-column`**, and **`formatting-cell`**.

■ The value returned by **`formatting-table`** is the table output record.

■ The arguments have values as follows:

`stream`

The stream to which output should be sent. The default is `*standard-output*`.

`x-spacing`

Determines the amount of space inserted between columns of the table; the default is the width of a space character. *`x-spacing`* can be specified in one of the following ways:

as an integer

A size in the current units to be used for spacing.

as string or character

The spacing is the width of the string or character in the current text style.

as a function

The function called with *`stream`* as its argument should return a number and that number of pixels is used.

as a list of form (*`number unit`*)

The *`unit`* is `:point`, `:pixel`, `:character` or `:line` and *`number`* is a positive integer. Note that the width is used for `:character` and the height for `:line`. `:line` is typically used for *`y-spacing`* but if used for *`x-spacing`*, the horizontal space is `(* number line-height)`. Similarly, if `:character` is used for *`y-spacing`*, the vertical space is `(* number standard-character-width)`.

`y-spacing`

Specifies the amount of blank space inserted between rows of the table; the default is the vertical spacing for the stream. The possible values for this option are the same as for the *`x-spacing`* option.

`multiple-columns`

Either `nil`, `t`, or an integer. If it is `t` or an integer, the table rows are broken up into multiple columns. If it is `t`, CLIM will determine the optimal number of columns. If it is an integer, it will be interpreted as the desired number of columns.

multiple-columns-x-spacing

Controls the spacing between the multiple columns. This option defaults to the value of the *x-spacing* option. It has the same format as *x-spacing*.

equalize-column-widths

When `t`, CLIM makes all the columns have the same width, which is the width of the widest cell in any column of the table.

move-cursor

When `t`, CLIM moves the cursor to the end of the table. The default is `t`.

record-type

This option is useful when you have defined a customized record type to replace CLIM's default table formatting record type. It specifies the class of the output record to be created.

formatting-row **[Macro]**

Arguments: `(&optional` *stream* `&key record-type) &body` *body*

■ Establishes a row context on the *stream* (the default is `*standard-output*`). All output performed on the stream within the extent of this macro will become the contents of one row of a table. **formatting-row** must be used within the extent of **formatting-table**, and it must be used in conjunction with **formatting-cell**.

■ The value returned by **formatting-row** is the row output record.

■ The arguments are as follows:

stream

The stream to which output should be sent. The default is `*standard-output*`.

record-type

This option is useful when you have defined a customized record type to replace CLIM's default row record type. It specifies the class of the output record to be created.

formatting-column **[Macro]**

Arguments: `(&optional` *stream* `&key record-type) &body` *body*

■ Establishes a column context on the *stream* (which defaults to `*standard-output*`). All output performed on the stream within the extent of this macro will become the contents of one column of the table. **formatting-column** must be used within the extent of **formatting-table**, and it must be used in conjunction with **formatting-cell**.

■ The value returned by **formatting-column** is the column output record.

■ The arguments are as follows:

stream

The stream to which output should be sent. The default is `*standard-output*`.

record-type

This option is useful when you have defined a customized record type to replace CLIM's default column record type. It specifies the class of the output record to be created.

formatting-cell **[Macro]**

Arguments: `(&optional stream &rest options &key (align-x ':left) (align-y ':top) min-width min-height record-type &allow-other-keys) &body body`

■ Establishes a cell context on the *stream* (which defaults to `*standard-output*`). All output performed on the stream within the extent of this macro will become the contents of one cell in a table. **formatting-cell** must be used within the extent of **formatting-row**, **formatting-column**, or **formatting-item-list**.

■ A cell can contain any other kind of output record: presentation, text, graphics, and so on. The alignment keywords enable you to specify constraints that affect the placement of the contents of the cell. Each cell within a column may have a different alignment; thus it is possible, for example, to have centered legends over flush-right numeric data.

■ The value returned by **formatting-cell** is the cell output record.

■ The arguments are as follows:

stream

The stream to which output should be sent. The default is `*standard-output*`.

align-x

Specifies the horizontal placement of the contents of the cell. Can be one of: `:left` (the default), `:right`, or `:center`.

`:left` means that the left edge of the cell is at the specified x coordinate. `:right` means that the right edge of the cell is at the specified x coordinate. `:center` means that the cell is horizontally centered over the specified x coordinate.

align-y

Specifies the vertical placement of the contents of the cell. Can be one of: `:top` (the default), `:bottom`, or `:center`.

`:top` means that the top of the cell is at the specified y coordinate. `:bottom` means that the bottom of the cell is at the specified y coordinate. `:center` means that the cell is vertically centered over the specified y coordinate.

min-width

Specifies the minimum width of the cell. The default, `nil`, causes the width of the cell to be only as wide as is necessary to contain the cell's contents.

min-width

can be specified in one of the following ways:

as an integer

A size in the current units to be used for spacing.

as a string or character

The spacing is the width (or height) of the string or character in the current text style.

as a function

The spacing is the amount of horizontal (or vertical) space the function would consume when called on the stream.

as a list of form (*number* unit)

The *unit* is `:point`, `:pixel`, `:line`, or `:character`.

min-height

Specifies the minimum height of the cell. The default, `nil`, causes the height of the cell to be only as high as is necessary to contain the cell's contents.

min-height
is specified in the same way as *min-width*.

record-type
This option is useful when you have defined a customized record type to replace CLIM's default cell record type. It specifies the class of the output record to be created.

11.3.1 Examples of table formatting

The following two example show a table of squares and square roots. One is arranged by rows, and the other by columns. Notice that the labels are centered, but the numbers if the rest of the table are right-aligned.

```
(defun squares-by-rows (&optional (stream *standard-output*))
 (clim:formatting-table (stream :x-spacing '(2 :character))
  (clim:formatting-row (stream)
   (clim:with-text-face (stream :italic)
    (clim:formatting-cell (stream :align-x :center) (format stream "N"))
    (clim:formatting-cell (stream :align-x :center) (format stream "N**2"))
   (clim:formatting-cell (stream :align-x :center) (format stream"(sqrt N)"))))
  (do ((i 1 (1+ i)))
    ((> i 10))
   (clim:formatting-row (stream)
    (clim:formatting-cell (stream :align-x :right)
     (format stream "~D" i))
    (clim:formatting-cell (stream :align-x :right)
     (format stream "~D" (* i i)))
    (clim:formatting-cell (stream :align-x :right)
     (format stream "~4$" (sqrt i)))))))

(defun squares-by-columns (&optional (stream *standard-output*))
 (clim:formatting-table (stream :x-spacing '(2 :character))
  (clim:formatting-column (stream)
   (clim:with-text-face (stream :italic)
    (clim:formatting-cell (stream :align-x :center) "N")
    (clim:formatting-cell (stream :align-x :center) "N**2")
    (clim:formatting-cell (stream :align-x :center) "(sqrt N)")))
  (do ((i 1 (1+ i)))
    ((> i 10))
   (clim:formatting-column (stream)
    (clim:formatting-cell (stream :align-x :right)
     (format stream "~D" i))
    (clim:formatting-cell (stream :align-x :right)
     (format stream "~D" (* i i)))
    (clim:formatting-cell (stream :align-x :right)
     (format stream "~4$" (sqrt i)))))))
```

Tables can be nested. Here is a table that is composed of several nested tables, each of which contains a small multiplication table for some number. Note the use of `:multiple-columns` to cause the tables to be split into 3 columns.

```
(defun multiplication-tables (&optional (stream *standard-output*))
 (flet ((table (stream factor)
      (clim:surrounding-output-with-border (stream)
```

```
      (clim:formatting-table (stream :multiple-columns 3)
       (do ((i 1 (1+ i)))
         ((> i 9))
        (clim:formatting-row (stream)
         (clim:formatting-cell (stream :align-x :right)
          (format stream "~D" (* i factor))))))))))
(clim:formatting-table (stream :multiple-columns 3
                :y-spacing 10 :x-spacing 10)
 (do ((i 1 (1+ i)))
   ((> i 9))
  (clim:formatting-row (stream)
   (clim:formatting-cell (stream :align-x :center)
    (table stream i)))))))
```

11.3.2 CLIM operators for item list formatting

Item list formatting is nearly identical to table formatting, except that the items to be formatted are one dimensional rather than two dimensional. You should use item list formatting when you simply have sequences of objects to display whose arrangement is not inherently tabular. For example, CLIM's own menu code uses item list formatting.

`formatting-item-list` **[Macro]**

Arguments: `(&optional` *`stream`* `&key x-spacing y-spacing initial-spacing n-columns n-rows max-width max-height stream-width stream-height (row-wise t) (move-cursor t) record-type) &body` *`body`*

- Establishes a menu formatting context on the *`stream`*; *`stream`* defaults to `*standard-output*`). You can use this macro to format the output in a tabular form when the exact ordering and placement of the cells is not important.
- This macro expects its *`body`* to output a sequence of items using **`formatting-cell`**, which delimits each item. (You do not use **`formatting-column`** or **`formatting-row`** within **`formatting-item-list`**.) If no keyword arguments are supplied, CLIM chooses the number of rows and columns for you. You can specify a constraint such as the number of columns or the number of rows (but not both), or you can constrain the size of the entire table display, by using *`max-width`* or *`max-height`* (but not both). If you supply either one of these constraints, CLIM will adjust the table accordingly.
- The value returned by **`formatting-item-list`** is the item list output record.
- The arguments are as follows:

`stream`

The stream to which output should be sent. The default is `*standard-output*`.

`x-spacing`

Determines the amount of space inserted between columns of the table; the default is the width of a space character. *`x-spacing`* can be specified in one of the following ways:

as an integer

A size in the current units to be used for spacing.

as string or character

The spacing is the width of the string or character in the current text style.

as a function

The function called with *stream* as its argument should return a number and that number of pixels is used.

as a list of form (*number unit*)

The *unit* is :point, :pixel, :character or :line and *number* is a positive integer. Note that the width is used for :character and the height for :line. :line is typically used for *y-spacing* but if used for *x-spacing*, the horizontal space is (* *number line-height*). Similarly, if :character is used for *y-spacing*, the vertical space is (* *number standard-character-width*).

y-spacing

Specifies the amount of blank space inserted between rows of the table; the default is the vertical spacing for the stream. The possible values for this option are the same as for the *x-spacing* option.

initial-spacing

formatting-item-list tries to evenly space items across the entire width of the stream. When this option is t, no whitespace is inserted before the first item on a line.

row-wise

When this is nil, if there are multiple columns in the item list, the entries in the item list are arranged in a manner similar to entries in a phone book. Otherwise the entries are arranged in a row-wise fashion.

n-columns

Specifies the number of columns of the table.

n-rows

Specifies the number of rows of the table.

max-width

Specifies the maximum width of the table display (in device units). (Can be overridden by *n-rows*.)

max-height

Specifies the maximum height of the table display (in device units). (Can be overridden by *n-columns*.)

stream-width

The width of the stream (in device units).

stream-height

The height of the stream (in device units).

move-cursor

When t, CLIM moves the cursor to the end of the table. The default is t.

format-items [Function]

Arguments: *items* &key (stream *standard-output*) printer presentation-type x-spacing y-spacing initial-spacing n-rows n-columns max-width max-height (row-wise t) record-type (cell-align-x ':left) (cell-align-y ':top)

■ Provides tabular formatting of a list of items. Each item in *items* is formatted as a separate cell within the table. *items* can be a list or a general sequence. **format-items** is a convenient functional interface to **formatting-item-list**.

■ The *stream*, *x-spacing*, *y-spacing*, *initial-spacing*, *n-rows*, *n-columns*, *max-width*, *max-height*, *row-wise*, and *record-type* arguments are the same as for **formatting-item-list**.

■ Note that you must supply either *printer* or *presentation-type*. Those arguments and the rest are as follows;

printer

A function that takes two arguments, an item and a stream. It should output the item to the stream. Note that you cannot use this keyword option with *presentation-type*.

presentation-type

A presentation type. Note that you cannot use this keyword option with *printer*.

The items will be printed as if *printer* were:

```
#'(lambda (item stream) (clim:present item presentation-type :stream stream))
```

cell-align-x

Supplies *align-x* to an implicitly used **formatting-cell**.

cell-align-y

Supplies *align-y* to an implicitly used **formatting-cell**.

Note that **format-items** is similar to **formatting-item-list**. Both operators do the same thing, except they accept their input differently:

- **formatting-item-list** accepts its input as a body that calls **formatting-cell** for each item.
- **format-items** accepts its input as a list of items with a specification of how to print them.

Note that menus use the one-dimensional table formatting model.

11.3.3 More examples of CLIM table formatting

Formatting a table from a list

The **example1** function formats a simple table whose contents come from a list.

```
(defvar *alphabet* '(a b c d e f g h i j k l m n o p q r s t u v w x y z))

(defun example1 (&optional (items *alphabet*)
         &key (stream *standard-output*)
           (n-columns 6) x-spacing y-spacing)
 (clim:formatting-table (stream :x-spacing x-spacing
                 :y-spacing y-spacing)
  (do () ((null items))
   (clim:formatting-row (stream)
    (do ((i 0 (1+ i)))
      ((or (null items) (= i n-columns)))
     (clim:formatting-cell (stream)
      (format stream "~A" (pop items))))))))
```

Evaluate

```
(example1 *alphabet* :stream *test-pane*)
```

You should see this table:

```
A B C D E F
G H I J K L
M N O P Q R
S T U V W X
Y Z
```

The table above shows the result of evaluating **example1** form without providing the *x-spacing* and *y-spacing* keywords. The defaults for these keywords makes tables whose elements are characters look reasonable.

You can easily vary the number of columns, and the spacing between rows or between columns. In the following example, we provide keyword arguments that change the appearance of the table.

Evaluating this form

```
(example1 *alphabet* :stream *test-pane* :n-columns 10
          :x-spacing 10 :y-spacing 10)
```

shows this table:

```
A B C D E F G H I J
K L M N O P Q R S T
U V W X Y Z
```

(Note that this example can be done with **formatting-item-list** as shown in example4 later on in this section.)

Formatting a table representing a calendar month

The **calendar-month** function shows how you can format a table that represents a calendar month. The first row in the table acts as column headings representing the days of the week. The following rows are numbers representing the day of the month.

This example shows how you can align the contents of a cell. The column headings (Sun, Mon, Tue, etc.) are centered within the cells. However, the dates themselves (1, 2, 3, ... 31) are aligned to the right edge of the cells. The resulting calendar looks good, because the dates are aligned in the natural way.

```
(setq *day-of-week-string* (make-array 7 :initial-contents
                            (list "Sun" "Mon" "Tue" "Wed" "Thu" "Fri" "Sat")))
(defparameter *month-lengths* (vector 31 28 31 30 31 30 31 31 30 31 30 31))
(defun month-length month year)
  (declare (special *month-lengths*))
  (if (/= month 2) (aref *month-lengths* (- month 1))
    (if (null (zerop (mod year 4))) 28
      (if (null (zerop (mod year 400))) 29 28))))
(defun calendar-month (month year &key (stream *standard-output*))
  (declare (special *day-of-week-string*))
 (let ((days-in-month (time:month-length month year)))
  (multiple-value-bind (n1 n2 n3 n4 n5 n6 start-day)
    (decode-universal-time (encode-universal-time
                    0 0 0 1 month year))
   (setq start-day (mod (+ start-day 1) 7))
   (clim:formatting-table (stream)
    (clim:formatting-row (stream)
```

```
    (dotimes (d 7)
     (clim:formatting-cell (stream :align-x :center)
      (write-string (aref *day-of-week-string* (mod d 7)) stream))))
   (do ((date 1)
      (first-week t nil))
     ((> date days-in-month))
    (clim:formatting-row (stream)
     (dotimes (d 7)
      (clim:formatting-cell (stream :align-x :right)
       (when (and (<= date days-in-month)
            (or (not first-week) (>= d start-day)))
        (format stream "~D" date)
        (incf date)))))))))
```

Evaluate

```
(calendar-month 5 90 :stream *test-pane*)
```

You should see this table:

```
Sun Mon Tue Wed Thu Fri Sat
          1   2   3   4   5
  6   7   8   9  10  11  12
 13  14  15  16  17  18  19
 20  21  22  23  24  25  26
 27  28  29  30  31
```

Formatting a table with regular graphic elements

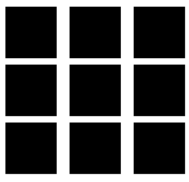
The **`example2`** function shows how you can draw graphics within the cells of a table. Each cell contains a rectangle of the same dimensions.

```
(defun example2 (&key (stream *standard-output*) x-spacing y-spacing)
 (clim:formatting-table (stream :x-spacing x-spacing
                :y-spacing y-spacing)
  (dotimes (i 3)
   (clim:formatting-row (stream)
    (dotimes (j 3)
     (clim:formatting-cell (stream)
      (clim:draw-rectangle* stream 10 10 50 50)))))))
```

Evaluate

```
(example2 :stream *test-pane* :y-spacing 5)
```

You should see this table:



Formatting a table with irregular graphics in the cells

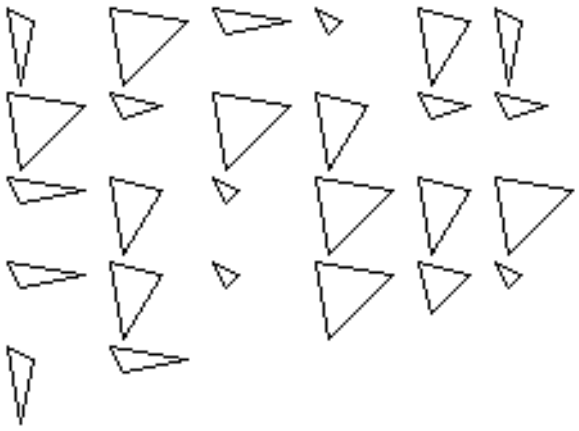
The **example3** function shows how you can format a table in which each cell contains graphics of different sizes.

```
(defun example3 (&optional (items *alphabet*)
         &key (stream *standard-output*)
           (n-columns 6) x-spacing y-spacing)
 (clim:formatting-table (stream :x-spacing x-spacing
                  :y-spacing y-spacing)
  (do () ((null items))
   (clim:formatting-row (stream)
    (do ((i 0 (1+ i)))
      ((or (null items) (= i n-columns)))
     (clim:formatting-cell (stream)
      (clim:draw-polygon* stream
                (list 0 0 (* 10 (1+ (random 3)))
                   5 5 (* 10 (1+ (random 3))))
                :filled nil)
      (pop items)))))))
```

Evaluate

```
(example3 *alphabet* :stream *test-pane*)
```

You should see something like this table:



Formatting a table of a sequence of items: clim:formatting-item-list

The **example4** function shows how you can use **formatting-item-list** to format a table of a sequence of items, when the exact arrangement of the items and the table is not important. Note that you use **formatting-cell** inside the body of **formatting-item-list** to output each item. You do not use **formatting-column** or **formatting-row**, because CLIM figures out the number of columns and rows automatically (or obeys a constraint given in a keyword argument).

```
(defun example4 (&optional (items *alphabet*)
         &key (stream *standard-output*) n-columns n-rows
           x-spacing y-spacing max-width max-height)
 (clim:formatting-item-list
   (stream :x-spacing x-spacing :y-spacing y-spacing
```

```
      :n-columns n-columns :n-rows n-rows
      :max-width max-width :max-height max-height)
  (do () ((null items))
   (clim:formatting-cell (stream)
    (format stream "~A" (pop items))))))
```

Evaluate

```
(example4 *alphabet* :stream *test-pane*)
```

You should see this table:

```
A B C D E
F G H I J
K L M N O
P Q R S T
U V W X Y
Z
```

You can easily add a constraint specifying the number of columns. Evaluate

```
(example4 *alphabet* :stream *test-pane* :n-columns 8)
```

You should see this table:

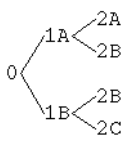
```
A B C D E F G H
I J K L M N O P
Q R S T U V W X
Y Z
```

11.4 Formatting graphs in CLIM

When you need to format a graph, you specify the nodes to be in the graph, and the scheme for organizing them. CLIM's graph formatter does the layout automatically, obeying any constraints that you supply.

You can format any directed, acyclic graph (DAG). ‘Directed’ means that the arcs on the graph have a direction. ‘Acyclic’ means that there are no loops in the graph. You can also format many graphs with cycles.

Here is an example of such a graph:



To specify the elements and the organization of the graph, you provide to CLIM the following information:

- The root node or nodes.
- A ‘object printer’, a function used to display each object. The function is passed the object associated with a node and the stream on which to do output.
- An ‘inferior producer’, a function which takes one node and returns its inferior nodes (the nodes to which it points).

Based on that information, CLIM lays out the graph for you. You can specify a number of options that control the appearance of the graph. For example, you can specify whether you want the graph to grow vertically (downward) or horizontally (to the right). Note that CLIM's algorithm does the best layout it can, but complicated graphs can be difficult to lay out in a readable way.

11.4.1 Examples of CLIM graph formatting

These fairly simple examples do illustrate important aspects of the grapher functionality. Basically, we define an object called a node. Each node has a name and a list of children. Therefore, with a node, we can draw a graph by drawing the node and then (recursively) its children. First, we define the node object:

```
(defstruct node
 (name "")
 (children nil))
```

Now we define a node (with children) that will be used as a root node. Notice that nodes 1A and 1B both have node 2B as a child.

```
(defvar g1-dag (let* ((2a (make-node :name "2A"))
                      (2b (make-node :name "2B"))
                      (2c (make-node :name "2C"))
                      (1a (make-node :name "1A" :children (list 2a 2b)))
                      (1b (make-node :name "1B" :children (list 2b 2c))))
                 (make-node :name "0" :children (list 1a 1b))))
```

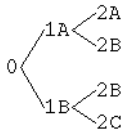
The following function draws the graph. We have provided an `&rest` argument so we can specify keyword arguments to **`format-graph-from-root`**. (We use this function also in examples for **`format-graph-from-roots`** since that function differs from **`format-graph-from-root`** only in allowing more than one root node.)

```
(defun test-graph (root-node &rest keys)
 (apply #'clim:format-graph-from-root root-node
     #'(lambda (node s)
       (write-string (node-name node) s))
     #'node-children
     keys))
```

Let us test this function. Evaluate

```
(test-graph g1 :stream *test-pane*)
```

You should see the following graph, which we call:



This graph shows what is drawn when all the defaults for the various keyword arguments to **`format-graph-from-root`** are used. As we discuss the arguments themselves, we show how this graph is affected by the arguments.

11.4.2 CLIM operators for graph formatting

`format-graph-from-roots` [Function]

Arguments: *`root-objects object-printer inferior-producer`*
`&key (stream *standard-output*) (orientation :horizontal)`
`center-nodes cutoff-depth merge-duplicates graph-type`
`duplicate-key duplicate-test arc-drawer arc-drawing-options`
`generation-separation within-generation-separation`
`maximize-generations (store-objects t) (move-cursor t)`

■ Draws a graph whose roots are specified by the sequence *`root-objects`*. The nodes of the graph are displayed by calling the function *`object-printer`*, which takes two arguments, the node to display and a stream. *`inferior-producer`* is a function of one argument that is called on each node to produce a sequence of inferiors (or `nil` if there are none). Both *`object-printer`* and *`inferior-producer`* have dynamic extent.

If the *`object-printer`* function will output newlines, you must ensure that the cursor position of the *`stream`* is (0,0). You can do this by evaluating

```
(setf (stream-cursor-position stream) (values 0 0))
```

The output from graph formatting takes place in a normalized +y-downward coordinate system. The graph is placed so that the upper left corner of its bounding rectangle is at the current text cursor position of *`stream`*. If the boolean *`move-cursor`* is `t` (the default), then the text cursor will be moved so that it immediately follows the lower right corner of the graph.

The returned value is the output record corresponding to the graph.

■ The arguments are as follows:

`stream`

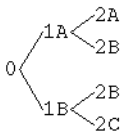
is the stream to which the output is done. It defaults to `*standard-output*`.

`orientation`

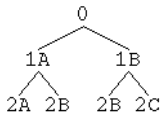
may be either `:horizontal` (the default) or `:vertical`. It specifies which way the graph is oriented. The graph in Graph example 1 above uses the default (`:horizontal`). Here we specify `:vertical` as the orientation:

```
(test-graph g1 :stream *test-pane* :orientation :vertical)
```

Thi produces the graph on the right (the default behavior is shown on the left):



:orientation :horizontal or unspecified



:orientation :vertical

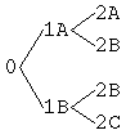
cutoff-depth

specifies the maximum depth of the graph. It defaults to `nil`, meaning that there is no cutoff depth. Otherwise it must be a positive integer, meaning that no nodes deeper than *`cutoff-depth`* will be formatted or displayed. Our example has three levels (1, 2, and 3). Here is the effect of setting this argument:

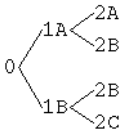
```
(test-graph g1 :stream *test-pane* :cutoff-depth 1)
(test-graph g1 :stream *test-pane* :cutoff-depth 2)
(test-graph g1 :stream *test-pane* :cutoff-depth 3)
```


:cutoff-depth 1

:cutoff-depth 2



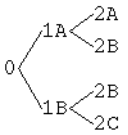
:cutoff-depth 3



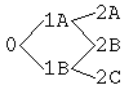
:cutoff-depth unspecified (same as 3 in this example)

merge-duplicates

If the boolean *`merge-duplicates`* is `t`, then duplicate objects in the graph will share the same node in the display of the graph. Thus, when *`merge-duplicates`* is `nil` (the default), the resulting graph will be a tree and duplicate objects will be displayed in separate nodes.



:merge-duplicates nil (the default)



:merge-duplicates t

duplicate-key

is a function of one argument that is used to extract the node object component used for duplicate comparison; the default is **`identity`**. Suppose in our example we were just interested in whether a node had any child whose name begins with ‘2’, but we are not concerned with which one. We could write a function that would make all such nodes duplicates. The following uses the Common

Lisp function **identity** when 2 is not the first character but makes all nodes with 2 as the first character duplicate:

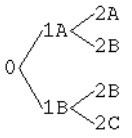
```
(defun my-dup (a b)
  (let ((a1 (first (node-name a)))
        (b1 (first (node-name b))))
    (if (null (eql a1 #\2) (identity a b) (eql #\2 b2)))))
```

duplicate-test

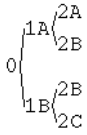
is a function of two arguments that is used to compare two objects to see if they are duplicates; the default is **eql**. *duplicate-key* and *duplicate-test* have dynamic extent.

generation-separation

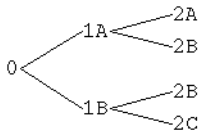
the amount of space between successive generations of the graph; (default 20). Here is the effect:



:generation-separation 20 (the default)



:generation-separation 5



:generation-separation 50

within-generation-separation

amount of space to leave between nodes in the same generation of the graph (default 10). *generation-separation* and *within-generation-separation* are specified in the same way as the *inter-row*-spacing argument to **formatting-table**.

center-nodes

When *center-nodes* is `t`, each node of the graph is centered with respect to the widest node in the same generation. The default is `nil`.

arc-drawer

The value of this argument should be a function that actually does the work or drawing the arcs connecting the nodes. This function should have the following argument list:

```
(stream from-node to-node x1 y1 x2 y2 &rest drawing-options
 &key path draw-nodes &allow-other-keys)
```

The keyword arguments `:path` and `:draw-nodes` need not be specified so long as `&allow-other-keys` is present. However, the system will call the *arc-drawer* function with those keyword arguments.

The arc goes from (x1,y1) to (x2,y2). The simplest behavior is to draw s straight line connecting the two points. Thus, if no value is specified for this argument, the default behavior is to draw a thin line from the *from-node* to the *to-node* using **draw-line***. However, your arc-drawing function can be more powerful. Intermediate points can be passed as the value of the `:path` argument (it will be a list of the form (xi1, yi1, xi2, yi2, ... xin, yin), where (xi,yi) specify the intermediate points).

Note that the arc drawing function will get the from- and to- node's objects only if *store-objects* is `t`. Otherwise, CLIM cannot determine what object to pass to the arc drawing function.

arc-drawing-options

contains keyword arguments that will be passed to the arc drawing function. These will be line drawing options, such as for **draw-line***.

graph-type

is a keyword that specifies the type of graph to draw. All CLIM implementations must support graphs of type :tree, :directed-graph (and its synonym :digraph), and :directed-acyclic-graph (and its synonym :dag). *graph-type* defaults to :digraph when *merge-duplicates* is t, otherwise it defaults to :tree. Currently, there are two formatting engines, one for simple trees and one for all other DAGs.

Here is an example of code that, given a list of CLOS classes, displays the directed graph of all of those classes subclasses.

```
(defun graph-classes (classes &optional (orientation :horizontal)
                      (stream *standard-output*))
 (clim:format-graph-from-roots
  (mapcar #'find-class classes)
  #'(lambda (class stream)
    (clim:surrounding-output-with-border (stream)
     (format stream "~S" (class-name class))))
  #'class-direct-subclasses
  :merge-duplicates t :orientation orientation
  :stream stream))
```

format-graph-from-root [Function]

Arguments: *root-object object-printer inferior-producer* &key stream orientation center-nodes cutoff-depth merge-duplicates graph-type key test arc-drawer arc-drawing-options generation-separation within-generation-separation maximize-generations store-objects move-cursor

■ This function is exactly like **format-graph-from-roots**, except that *root-object* is a single root object. *key* and *test* are used as the duplicate key and duplicate test.

Some notes on graphing

The grapher can introduce fake nodes -- circular graph connectors, and edge splitters. Under some situations the arc drawer will be called to draw arcs between these nodes. In that case, the class of from/to object will be clim-internals::grapher-fake-object.

The grapher uses the arc drawing function to edge-splitting nodes. In this situation, it is called with :draw-node t. The :path argument is used to specify intermediate points on the edge.

11.5 Formatting text in CLIM

CLIM provides the following forms for breaking up lengthy output into multiple lines and for indenting output.

format-textual-list [Function]

Arguments: *sequence printer* &key (stream *standard-output*) (separator ", ") conjunction

■ Outputs a *sequence* of items as a textual list. For example, the list

```
(1 2 3 4)
```

will be printed as the following when the printer function is **princ** and :conjunction is "and".

```
1, 2, 3, and 4
```

■ The arguments provide control over the appearance of each element of the sequence and over the separators used between each pair of elements. The separator string is output after every element but the last one. The conjunction is output before the last element.

sequence

The sequence to output.

printer

is a function of two arguments: an element of the sequence and a stream. It is used to output each element of the sequence. Typical values are #'princ or #'prin1 if you do not want to write your own specialized printer.

stream

Specifies the output stream. The default is *standard-output*.

separator

Specifies the characters to use to separate elements of a textual list. The default is ", " (comma followed by a space).

conjunction

Specifies a string to use in the position between the last two elements. Typical values are "and" and "or". The default is no conjunction.

For example, the form used to get the result above is:

```
(clim:format-textual-list '(1 2 3 4) #'princ :conjunction "and")
```

The **filling-output** macro described next allows you to restrict formatted text to (more or less) a specified width. We say 'more or less' because filling-output will not break words across lines, so some lines can be longer than the specified width. Our examples will use part of the Gettysburg address:

```
(defvar *gettysburg-address*
    (concatenate 'string
     "Fourscore and seven years ago our forefathers brought forth "
     "on this continent a new nation, conceived in Liberty, and "
     "dedicated to the proposition that all men are created equal. "
     "Now we are engaged in a great civil war, testing whether that "
     "nation, or any nation so conceived and so dedicated, can long "
     "endure."))
```

Here is filling-output called with its defaults. Compare this example with the examples in the function definition where non-default values are supplied.

```
CLIM-USER(139): (filling-output
                    (*standard-output*)
                  (write-string *gettysburg-address*))
Fourscore and seven years ago our forefathers brought forth on this continent a
new nation, conceived in Liberty, and dedicated to the proposition that all men
are created equal. Now we are engaged in a great civil war, testing whether that
nation, or any nation so conceived and so dedicated, can long endure.
NIL
```

`filling-output` **[Macro]**

Arguments: `(&optional` *`stream`* `&key (fill-width '(80 :character))`
`break-characters after-line-break`
`after-line-break-initially)`
`&body` *`body`*

■ Binds local environment *`stream`* (the default is `*standard-output*`) to a stream that inserts line breaks into the output written to it so that the output is usually no wider then *`fill-width`*. The filled output is then written on the stream that is the original value of *`stream`*. **`filling-output`** does not split words across lines, so it can produce output wider than *`fill-width`*.

■ Words are separated by the characters indicated by *`break-characters`*. *`break-characters`* defaults to `(#\Space)` When a line is broken to prevent wrapping past the end of a line, the line break is made at one of these separators.

■ The arguments are as follows:

`stream`

The output stream; the default is `*standard-output*`.

`fill-width`

Specifies the width of filled lines. The default is 80 characters. It can be specified in one of the following ways:

as a **list** of the form (*`number`* unit), where *`unit`* is one of

`:pixel` The width in pixels.

`:point` The width in printers points.

`:character` The width of "M" in the current text style.

as an **integer**; the width in device units (for example, pixels).

as a **string**; the spacing is the width of the string.

as a **function**; the spacing is the amount of space the function would consume when called on the stream.

Here are two example where fill-width is specified, first as 60 characters, then as the width of the string "The Gettysburg Address":

```
CLIM-USER(140): (filling-output
                  (*standard-output* :fill-width '(60 :character))
                (write-string *gettysburg-address*))
Fourscore and seven years ago our forefathers brought forth
on this continent a new nation, conceived in Liberty, and
dedicated to the proposition that all men are created equal.
Now we are engaged in a great civil war, testing whether
that nation, or any nation so conceived and so dedicated,
can long endure.
NIL
CLIM-USER(141): (filling-output
                  (*standard-output* :fill-width "The Gettysburg
Address")
                (write-string *gettysburg-address*))
Fourscore and seven
years ago our
```

```
forefathers brought
forth on this
continent a new
nation, conceived in
Liberty, and dedicated
to the proposition
that all men are
created equal. Now we
are engaged in a great
civil war, testing
whether that nation,
or any nation so
conceived and so
dedicated, can long
endure.
NIL
```

break-characters

Specifies a list of characters at which to break lines.

after-line-break

Specifies a string to be sent to *stream* after line breaks; the string appears at the beginning of each new line. The string must not be wider than *fill-width*.

In this example, *after-line-break* is given the value "GA> ". Note that this is not printed on the first line. See the *after-line-break-initially* below. Also note that the "GA> " is counted in determining the width:

```
CLIM-USER(142): (filling-output
                    (*standard-output* :fill-width "The Gettysburg
Address"
                                       :after-line-break "GA> ")
                  (write-string *gettysburg-address*))
Fourscore and seven
GA> years ago our
GA> forefathers
GA> brought forth on
GA> this continent a
GA> new nation,
GA> conceived in
GA> Liberty, and
GA> dedicated to the
GA> proposition that
GA> all men are
GA> created equal. Now
GA> we are engaged in
GA> a great civil war,
GA> testing whether
GA> that nation, or
```

```
GA> any nation so
GA> conceived and so
GA> dedicated, can
GA> long endure.
NIL
```

after-line-break-initially

Boolean option specifying whether the *after-line-break* text is to be written to *stream* before doing *body*, that is, at the beginning of the first line; the default is `nil`. Recall from the example just above, "GA> " was not printed on the first line. When we specify this argument true, it is (we truncate the example to save space -- after the first line it is identical to the above):

```
CLIM-USER(143): (filling-output
                    (*standard-output*
                         :fill-width "The Gettysburg Address"
                         :after-line-break "GA> "
                         :after-line-break-initially t)
                  (write-string *gettysburg-address*))
GA> Fourscore and
GA> years ago our
GA> forefathers
GA> brought forth on
GA> this continent a
[... lines deleted to save paper -- see example above ...]
NIL
```

Here is an example of using **format-textual-list** with **filling-output**.

```
(let ((stream *standard-output*)
   (data (let ((result nil))
       (dotimes (i 20) (push i result))
       (nreverse result))))
 (clim:filling-output (stream :fill-width '(30 :character)
                :after-line-break " ")
  (clim:format-textual-list data #'princ
              :stream stream
              :separator ", " :conjunction "and")
  (write-char #. stream)))
```

indenting-output [Function]

Arguments: (*stream indentation* &key (move-cursor t)) &body *body*

■ Binds *stream* to a stream that inserts whitespace at the beginning of each line, and writes the indented output to the stream that is the original value of *stream*.

■ The arguments are as follows:

stream

The output stream. As a special case, `t` is an abbreviation for `*standard-output*`.

indentation

What gets inserted at the beginning of each line output to the stream. Four possibilities exist:

integer

The width in device units (for example, pixels).

string

The spacing is the width of the string.

function

The spacing is the amount of space the function would consume when called on the stream.

list

The list is of the form (*number* unit), where *unit* is one of

`:pixel` The width in pixels.

`:point` The width in printers points.

`:character` The width of "M" in the current text style.

move-cursor

When `t`, CLIM moves the cursor to the end of the table. The default is `t`.

■ You should begin the body with `(terpri` *stream*`)`, or equivalent, to position the stream to the indentation initially. That is, it is perfectly valid to indent only subsequent lines.

■ Note: if you use **`indenting-output`** in conjunction with **`filling-output`**, you should put the call to **`indenting-output`** *outside* of the call to **`filling-output`**.

For example, if you want to indent the Gettysburg address above, you could do the following:

```
(clim:indenting-output (*standard-output* '(2 :character))
 (clim:filling-output (*standard-output* :fill-width '(60 :character))
  (write-string *gettysburg-address* *standard-output*)))
```

`with-aligned-prompts` **[Macro]**

Arguments: `(`*stream* `&key :align-prompts) &body` *body*

■ This macro causes all accepts in the body to be aligned if `:align-prompts` is non-`nil`. This macro deals with nesting as follows: if the `:align-prompts` argument is given as `nil` in a context where `:align-prompts` is non-`nil` (typically specified in a call to **`accepting-values`**) then vanilla output can be performed within the body.

11.6 Bordered output in CLIM

CLIM provides a mechanism for surrounding arbitrary output with some kind of a border. To specify that a border should be generated, you surround some code that does output with **`surrounding-output-with-border`**, an advisory macro that describes the type of border to be drawn.

`surrounding-output-with-border` **[Macro]**

Arguments:
```
(&optional stream
&key (shape ':rectangle) &allow-other-keys)
&body body
```

■ Binds the local environment in such a way that the output of *body* will be surrounded by a border of the specified *shape*. The default shape is `:rectangle`. Keyword arguments acceptable to drawing functions (such as `:ink` and `:filled`) can also be specified.

■ **This macro cannot be used within `formatting-table`**. It is very unlikely that this restriction will ever go away.

define-border-type **[Macro]**

Arguments: *shape arglist* &body *body*

- Defines a new kind of border named *shape*. *arglist* will typically be (stream record left top right bottom).
- *body* is the code that actually draws the border. It has lexical access to stream, record, left, top, right, and bottom, which are respectively, the stream being drawn on, the output record being surrounded, and the coordinates of the left, top, right, and bottom edges of the bounding rectangle of the record.
- Note that the predefined border types, :rectangle, :oval, :drop-shadow, and :underline are defined using this macro.

For example, the following produces a piece of output surrounded by a rectangle.

```
(defun bordered-triangle (stream)
 (clim:surrounding-output-with-border (stream :shape :rectangle)
  (clim:draw-polygon* stream '(40 120 50 140 30 140))))
```

The following is the result of evaluating (bordered-triangle *test-pane*):

[This page intentionally left blank.]

Chapter 12 Hardcopy streams in CLIM

It is often useful for an application to produce stream and medium output in a hardcopy form. CLIM supports hardcopy output through the **`with-output-to-postscript-stream`** macro.

Note that CLIM does *not* support doing output to one stream, and then replaying the output on another stream. This means that, rather than replaying output records to a hardcopy stream, you must regenerate the output to the hardcopy stream.

12.1 Function for doing PostScript output

`with-output-to-postscript-stream` **[Macro]**

Arguments: `(`*`stream-var file-stream`* `&key device-type multi-page scale-to-fit header-comments (orientation :portrait)) &body` *`body`*

■ Within *`body`*, *`stream-var`* is bound to a stream that produces PostScript code. This stream is suitable as a stream or medium argument to any of the CLIM output utilities, including the formatted output facilities. A PostScript program describing the output to the *`stream-var`* stream will be written to *`file-stream`*. *`stream-var`* must be the name of a variable.

■ The arguments are as follows:

`device-type`

A symbol that names some sort of PostScript display device. It defaults to the device type for an Apple LaserWriter.

`multi-page`

A boolean value that specifies whether or not the output should be broken into multiple pages if it is larger than one page. How the output is broken into multiple pages, and how these multiple pages should be pieced together is unspecified. The default is `nil`.

`scale-to-fit`

A boolean that specifies whether or not the output should be scaled to fit on a single page if it is larger than one page. The default is `nil`. It is an error if *`multi-page`* and *`scale-to-fit`* are both supplied as `t`.

`orientation`

One of `:portrait` (the default) or `:landscape`. It specifies how the output should be oriented. `:portrait-style` output has the long dimension of the paper along the vertical axis; `:landscape-style` output has the long dimension of the paper along the horizontal axis.

`header-comments`

allows the programmer to specify some PostScript header comment fields for the resulting PostScript output. The value of *`header-comments`* is a list consisting of alternating keyword and value pairs. These are the supported keywords:

`:title`

Specifies a title for the document, as it will appear in the "%%Title:" header comment.

`:for`

Specifies who the document is for. The associated value will appear in a "%%For:" document comment.

■ Note: The PostScript programs written by this implementation of CLIM do not strictly conform to the conventions described under **Appendix C: Structuring Conventions** of the *PostScript Language Reference Manual*. Software tools which attempt to determine information about these PostScript programs based on "\%\%" comments within them may be unsuccessful.

new-page [Function]

Arguments: *stream*

IMPLEMENTATION LIMITATION: This function is not supported n the current release.

■ This function is designed to send all of the currently collected output to the related file stream, emit a PostScript **showpage** command, and reset the PostScript stream to have no output.

12.2 Examples of Doing PostScript Output

This example writes a PostScript program which draws a square, a circle and a triangle to a file named *icons-of-high-tech.ps*.

```
(defun print-icons-of-high-tech-to-file ()
 (with-open-file (file-stream "icons-of-high-tech.ps" :direction :output)
  (clim:with-output-to-postscript-stream (stream file-stream)
   (let* ((x1 150) (y 250) (size 100)
       (x2 (+ x1 size))
       (radius (/ size 2))
       (base-y (+ y (/ (* size (sqrt 3)) 2))))
    (clim:draw-rectangle*
     stream (- x1 size) (- y size) x1 y)
    (clim:draw-circle*
     stream (+ x2 radius) (- y radius) radius)
    (clim:draw-triangle*
     stream (+ x1 radius) y x1 base-y x2 base-y)))))
```

This example uses multi-page mode to draw a graph of the subclasses of the class `bounding-rectangle` by writing a PostScript program to the file *class-graph.ps*. The use of `:multi-page` t causes CLIM to split this rather large graph up into multiple pages.

```
(with-open-file (file "class-graph.ps" :direction :output)
 (clim:with-output-to-postscript-stream (stream file :multi-page t)
  (clim:format-graph-from-root
   (find-class 'clim:bounding-rectangle)
   #'(lambda (object s)
     (write-string (string (class-name object)) s))
   #'class-direct-subclasses
   :stream stream)))
```

Chapter 13 Menus and dialogs in CLIM

13.1 Concepts of menus and dialogs in CLIM

CLIM provides three powerful menu interaction routines for allowing user interfacing through pop-up menus and dialogs, and menus and dialogs embedded in an application window:

- **`menu-choose`** is a straightforward menu generator that provides a quick way to construct menus. You can call it with a list of menu items. For a complete definition of menu item, see the function **`menu-choose`**. This calls **`frame-manager-menu-choose`** to do most of its work; often, the frame manager can use a native menu.
- **`menu-choose-from-drawer`** is a lower level routine that allows the user much more control in specifying the appearance and layout of a menu. You can call it with a window and a drawing function. Use this function for more advanced, customized menus.
- **`accepting-values`** provides the ability to build a dialog. You can specify several items that can be individually selected or modified within the dialog before dismissing it and in this way it differs from **`menu-choose`** and **`menu-choose-from-drawer`**, both of which allow you to select one thing only.

13.2 Operators for menus in CLIM

You can use the following functions to get user input via a menu.

`menu-choose` [Function]

Arguments: *`items`* `&key associated-window cache cache-value cache-test cell-align-x cell-align-y default-item id-test label max-height max-width n-columns n-rows pointer-documentation presentation-type printer text-style unique-id x-spacing y-spacing`

■ Displays a menu with the choices in *`items`*. This function returns three values: the value of the chosen item, the item itself, and the event corresponding to the gesture that the user used to select it. *`items`* can be a list or a general sequence. This function returns `nil` for all values if the menu is aborted by burying it.

■ *items* should be a list of menu items. Each menu item has a visual representation derived from a display object, an internal representation which is a value object (this is the first item returned by **menu-choose**), and a (possibly empty) set of menu item options. The form of a menu item is one of the following:

an atom	The item is both the display object and the value object.
a cons	The **car** is the display object and the **cdr** is the value object. The value object must be an atom. If you need to return a non-atom as the value, specify the item as a list and use the `:value` option.
a list	The **car** is the display object and the **cdr** is a list of alternating option keywords and values. The value object is specified with the keyword `:value` and defaults to the display object if `:value` is not present.

The menu item options are:

`:value`	Specifies the value object.
`:text-style`	Specifies the text style used to **princ** the display object when neither `:presentation-type` nor `:printer` is specified.
`:items`	Specifies an item list for a sub-menu used if this item is selected.
`:documentation`	Associates some documentation with the menu item. When `:pointer-documentation` is not `nil`, it will be used a pointer documentation for the item.
`:active`	When true (the default), indicates the item is active, that is can be selected. When false, the item is inactive and cannot be selected. CLIM will generally provide some visual indication that an item is inactive (for example, graying over the item).
`:type`	Specifies the type of the item. Possible values are `:item` (the default) indicating that the item is a normal menu item; `:label`, indicating the item is simply an inactive label (labels are not grayed over as are inactive items); `:divider`, indicating that the item serves as a divider between groups of other items (divider items will usually be drawn as a horizontal line).

The visual representation of an item depends on the *printer* and *presentation-type* keyword arguments. If *presentation-type* is specified, the visual representation is produced by **present** of the menu item with that presentation type. Otherwise, if *printer* is specified, the visual representation is produced by the *printer* function which receives two arguments, the *item* and a *stream* to write on. The *printer* function should output some text or graphics at the stream's cursor position, but need not call **present**. If neither *presentation-type* nor

printer is specified, the visual representation is produced by **princ** of the display object. Note that if :presentation-type or :printer is specified, the visual representation is produced from the entire menu item, not just from the display object.

The keyword arguments to **menu-choose** are:

:associated-window	The CLIM window the menu is associated with. This defaults to the top-level window of the current application frame.
:cache	Indicates whether CLIM should cache this menu for later use. If t, then *unique-id* and *id-test* serve to uniquely identify this menu. Caching menus can speed up later uses of the same menu.
:cache-test	The function that compares *cache-values*. It defaults to **equal**.
:cache-value	If *cache* is non-nil, this is the value that is compared to see if the cached menu is still valid. Defaults to *items*, but you may be able to supply a more efficient cache value than that.
:cell-align-x	Specifies the horizontal placement of the contents of the cell. Can be one of: :left, :right, or :center. The default is :left.
:cell-align-y	Specifies the vertical placement of the contents of the cell. Can be one of: :top, :bottom, or :center. The default is :top.
:default-item	The menu item where the mouse will appear.
:id-test	The function that compares *unique-ids*. It defaults to **equal**.
:label	The string that the menu title will be set to.
:max-height	Specifies the maximum height of the table display (in device units). (Can be overridden by *n-columns*.)
:max-width	Specifies the maximum width of the table display (in device units). (Can be overridden by *n-rows*.)
:n-columns	Specifies the number of columns.
:n-rows	Specifies the number of rows.
:pointer-documentation	Either nil (the default), meaning the no pointer documentation should be computed, or a stream on which pointer documentation should be displayed.
:presentation-type	Specifies the presentation type of the menu items.

`:printer`	The function used to print the menu items in the menu. The function should take two arguments, the menu item and the stream to print it to.
`:text-style`	A text style that defines how the menu items are presented.
`:unique-id`	If *`cache`* is non-`nil`, this is used to identify the menu. It defaults to the *`items`*, but can be set to a more efficient tag.
`:x-spacing`	Determines the amount of space inserted between columns of the table; the default is the width of a space character. Can be specified in one of the following ways: **Integer** A size in the current units used for spacing. **String or character** The spacing is the width or height of the string or character in the current text style. Function The spacing is the amount of horizontal or vertical space the function would consume when called on the stream. **List of form** (*`number unit`*) where *`unit`* is `:point`, `:pixel`, or `:character`.
`:y-spacing`	Specifies the amount of blank space inserted between rows of the table; the default is the vertical spacing for the stream. The possible values for this option are the same as for the *`x-spacing`* option.

- CLIM will always use a native Motif menu.
- See the section 135 **Examples of menus and dialogs in CLIM**.

`menu-choose-from-drawer` [Function]

Arguments: *`menu type drawer`* `&key x-position y-position cache unique-id (id-test #'equal) (cache-value t) (cache-test #'eql) leave-menu-visible default-presentation`

- A lower-level routine for displaying menus. It allows the user much more flexibility in the menu layout. Unlike **`menu-choose`**, which automatically creates and lays out the menu, **`menu-choose-from-drawer`** takes a programmer-provided window and drawing function. Then it draws the menu items into that window using the drawing function. The drawing function gets called with arguments (*`stream type`*). *`stream`* of course specifies the stream on which to draw but *`type`* is available for the drawing function to use for its own purposes, the usual being using it for **`present`**.

`menu-choose-from-drawer` returns two values; the object the user clicked on, and the gesture.

You can create a temporary window for drawing their menu using **`with-menu`**.

Note that, when enabled by `*abort-menus-when-buried*`, this function returns `nil` for all values if the menu is aborted by burying it.

`menu`

The CLIM window to use for the menu.

type

The presentation type of the mouse-sensitive items in the menu. This is the input context that will be established once the menu is displayed. For users who don't need to define their own types, a useful presentation-type is `menu-item`.

drawer

A function that takes arguments (*stream* type) that draws the contents of the menu.

x-position

The requested left edge of the menu (if supplied).

y-position

The requested top edge of the menu (if supplied).

leave-menu-visible

If non-`nil`, the window will not be de-exposed once the selection has been made. The default is `nil`, meaning that the window will be de-exposed once the selection has been made.

default-presentation

Identifies the presentation that the mouse is pointing to when the menu comes up.

cache

unique-id

id-test

cache-value

cache-test

are as for **menu-choose**.

■ See the section 13.5 **Examples of Menus and Dialogs in CLIM**.

draw-standard-menu [Function]

Arguments: *menu presentation-type items default-item* &key (item-printer #'clim:print-menu-item) max-width max-height n-rows n-columns x-spacing y-spacing row-wise (cell-align-x ':left) (cell-align-y ':top)

■ **draw-standard-menu** is the function used by CLIM to draw the contents of a menu, unless the current frame manager determines that host window toolkit should be used to draw the menu instead. *menu* is the stream onto which to draw the menu, *presentation-type* is the presentation type to use for the menu items (usually `menu-item`), and *item-printer* is a function used to draw each item.

The other arguments are as for **menu-choose**.

print-menu-item [Function]

Arguments: *menu-item* &optional (*stream* *standard-output*)

■ Given a menu item *menu-item*, display it on the stream *stream*. This is the function that **menu-choose** uses to display menu items if no printer is supplied.

with-menu [Macro]

Arguments: (*menu* &optional *associated-window* &rest *options* &key label scroll-bars) &body *body*

■ Binds *menu* to a temporary window, exposes the window on the same screen as the *associated-window*, runs the body, and then de-exposes the window. The values returned by **with-menu** are the values returned by *body*.

menu

The name of a variable which is bound to the window to be used for the menu.

associated-window

A window that this window is associated with, typically a pane of an application frame. If not supplied, *associated-window* will default to the top-level window of the current application frame.

This example shows how to use **with-menu** with **menu-choose-from-drawer** to draw a temporary menu.

```
(defun choose-compass-direction ()
 (labels ((draw-compass-point (stream ptype symbol x y)
       (clim:with-output-as-presentation (stream symbol ptype)
        (clim:draw-text* stream (symbol-name symbol)
                 x y
                 :align-x :center
                 :align-y :center
                 :text-style '(:sans-serif :roman :large))))
     (draw-compass (stream ptype)
      (clim:draw-line* stream 0 25 0 -25
              :line-thickness 2)
      (clim:draw-line* stream 25 0 -25 0
              :line-thickness 2)
      (loop for point in '((n 0 -30) (s 0 30)
                (e 30 0) (w -30 0))
         do (apply #'draw-compass-point
               stream ptype point))))
  (clim:with-menu (menu)
   (clim:menu-choose-from-drawer
    menu 'clim:menu-item #'draw-compass))))
```

with-menu can also be used to allocate a temporary window for other uses.

13.3 Operators for dealing with dialogs in CLIM

You can use the following functions and macros to create dialogs.

accepting-values **[Macro]**

Arguments: (&optional *stream* &key command-table own-window exit-boxes align-prompts initially-select-query-identifier modify-initial-query resynchronize-every-pass (check-overlapping t) label x-position y-position width height scroll-bars :right-margin :bottom-margin) &body *body*

■ Builds a dialog for user interaction based on calls to **accept** within its body. The user can select the values and change them, or use defaults if they are supplied. The dialog will also contain buttons typically labeled "OK" and "Cancel" (see :the *exit-boxes* argument below). If "OK" is selected then **accepting-values** returns whatever values the body returns. If "Cancel" is selected, **accepting-values** will invoke the abort restart. Callers of **accepting-values** may want to use **restart-case** or **with-simple-restart** in order to locally establish an abort restart.

If **make-application-frame** is called within the *body*, to create, for example, a frame that is popped up for some purpose, the :calling-frame argument to **make-application-frame** should be specified with the value *application-frame*. If this is not done, the popped-up frame may not be sensitive to the mouse or keyboard input. Note that this means you cannot use an existing frame since it cannot have been created with the correct value for the :calling-frame argument.

stream

The stream **accepting-values** will use to build up the dialog. When *stream* is t, that means *query-io*.

body

The body of the macro, which contains calls to **accept** that will be intercepted by **accepting-values** and used to build up the dialog.

own-window

When *own-window* is true the **accepting-values** dialog will appear in its own popped-up window. In this case the initial value of *stream* is a window with which the dialog is associated. This is similar to the *associated-window* argument to **menu-choose**. Within the *body*, the value of *stream* will be the popped-up window.

The value of *own-window* can be t or nil. If it is t, the *right-margin* and *bottom-margin* arguments can be used to control the amount of extra space to the right of and below the dialog (useful if the user's responses to the dialog take up more space than the initially displayed defaults).

right-margin

bottom-margin

These arguments only have effect when *own-window* is true. Their values control the amount of extra space to the right of and below the dialog (useful if the user's responses to the dialog take up more space than the initially displayed defaults). The allowed values for *right-margin* are the same as for *x-spacing* in **formatting-table**; the allowed values for *bottom-margin* are the same as for *y-spacing* in **formatting-table**. **formatting-table** is defined in section 11.3.

exit-boxes

Allows you to describe what the exit boxes should look like. :exit and :abort choices are supported. The default behavior is as though you specified the following:

```
'((:exit "OK") (:abort "Cancel"))
```

You can specify your own strings in a similar list.

initially-select-query-identifier

Specifies that a particular field in the dialog should be pre-selected when the user interaction begins. The field to be selected is tagged by the *query-identifier* option to **accept**; use this tag as the value for the *initially-select-query-identifier* keyword, as shown in the following example:

```
(defun avv ()
 (let (a b c)
  (clim:accepting-values
    (*query-io* :initially-select-query-identifier 'the-tag)
   (setq a (clim:accept 'pathname :prompt "A pathname"))
   (terpri *query-io*)
```

```
    (setq b (clim:accept 'integer :prompt "A number"
                         :query-identifier 'the-tag))
    (terpri *query-io*)
    (setq c (clim:accept 'string :prompt "A string")))
   (values a b c)))
```

When the initial display is output, the input editor cursor appears after the prompt of the tagged field, just as if the user had selected that field by clicking on it (unless a gadget is displayed). The default value, if any, for the selected field is not displayed.

resynchronize-every-pass

Boolean option specifying whether earlier queries depend on later values; the default is `nil`.

When *resynchronize-every-pass* is `t`, the contents of the dialog are redisplayed an additional time after each user interaction. This has the effect of ensuring that, when the value of some field of a dialog depends on the value of another field, all of the displayed fields will be up to date.

You can use this option to dynamically alter the dialog. The following is a simple example. It initially displays an integer field that disappears if a value other than 1 is entered; in its place a two-field display appears.

```
(defun alter-multiple-accept ()
 (let ((flag 1))
  (clim:accepting-values (*query-io* :resynchronize-every-pass t)
   (setq flag (clim:accept 'integer :default flag :prompt "Number"))
   (when (= flag 1)
    (terpri *query-io*)
    (clim:accept 'string :prompt "String")
    (terpri *query-io*)
    (clim:accept 'pathname :prompt "Pathname")))))
```

As the example shows, to use this option effectively, the controlling variable(s) must be initialized outside the lexical scope of the **accepting-values** macro.

label

Allows you to supply a label in `:own-window` t dialogs. This is just like the `:label` option to **menu-choose**.

x-position

y-position

Allow you to specify where an `:own-window` t dialog will come up. By default, the dialog will come up near the center of the frame launching the **accepting-values** dialog. These are just like the options of the same name to **menu-choose-from-drawer**.

scroll-bars

Can be `:both`, `:vertical`, `:horizontal`, or `nil`. Default is `nil` (meaning no scroll bars). Only has effect when `:own-window` is true.

command-table

Specifies the command table for the **accepting-values** frame.

align-prompts

The values can be `nil` (the default), `:left`, meaning left align, `:right`, meaning right align, and `t`, which is the same as `:right`. Suppose you have two **accept**s within the body, with

prompts "Age?" and "Last Name?". The prompts and space for input will align as follows, depending on the value of this argument:

unspecified or `nil`:

```
Age? [space for input]
Last Name? [space for input]
```

`:left`:

```
Age?       [space for input]
Last Name? [space for input]
```

`:right` or `t`:

```
       Age? [space for input]
Last Name? [space for input]
```

■ Note: you must supply either a unique prompt or a query-identifier for each **`accept`** form within an **`accepting-values`** form; otherwise, there will be no way that the accept-values can identify which **`accept`** form is being run.

■ In Allegro CLIM, **`accepting-values`** will use Motif gadgets for the fields in the dialog, since the default view for the Motif frame managers is `+gadget-dialog-view+`. You can inhibit this behavior by either explicitly specifying `:view` in the calls to **`accept`**, or by binding the value of `frame-manager-dialog-view` around the call to **`accepting-values`**. For more information on views, see section 8.6.13 **Using views with CLIM presentation types**.

■ See the section 13.5 **Examples of menus and dialogs in CLIM**.

`accept-values-command-button` **[Macro]**

Arguments: `((&optional` *`stream`* `&key view documentation query-identifier (cache-value t) (cache-test #'eql) resynchronize (active-p` *`t`*`))` *`prompt`* `&body` *`body`*`)`

■ Displays *`prompt`* on *`stream`* and creates an area (the button) in which, when the pointer is clicked, *`body`* is evaluated. This function is only used within the **`accepting-values`** form.

In Allegro CLIM using `+gadget-dialog-view+` as the view, this will create a Motif push button.

`documentation`

An object that will be used to produce pointer documentation for the command button. If the object is a string, the string itself will be used as the documentation. Otherwise, it should be either a function of one argument, the stream to which the documentation will be written. The default is *`prompt`*.

`query-identifier`

This option is used with **`accepting-values`**.

`cache-value`

A value that remains constant if the appearance of the command button is to stay the same. You rarely need to use this because changing *`prompt`* changes the default for *`query-identifier`* causing a new **`accept-values-command-button`** to be created.

`cache-test`

A function of two arguments that is used for comparing cache values.

resynchronize

When this is `t`, the dialog is redisplayed on additional time whenever the command button is clicked on. See the *resynchronize-every-pass* argument to **accepting-values** for more information.

prompt

A string, which is used to label the button, or a form, which must output to *stream*. The output is used as the label.

view

This argument controls the apperance of the button. It takes the same arguments as the push-button gadget (defined in section 16.3).

active-p

A boolean which controls whether the button is active. Specifying nil displays the button but it is not sensitive.

■ See the section 13.5 **Examples of menus and dialogs in CLIM**.

13.4 Using an :accept-values pane in a CLIM application frame

`:accept-values` panes are used when you want one of the panes of your application to be in the form of an **accepting-values** dialog.

There are several things to remember when using an `:accept-values` pane in your application frame:

- For an `:accept-values` pane to work your frame's command table must inherit from the `accept-values-pane` command table.
- The `:display-function` option for an `:accept-values` pane will typically be something like

  ```
  (clim:accept-values-pane-displayer :displayer my-acceptor-function)
  ```

  where *my-acceptor-function* is a function that you write. It contains calls to **accept** just as they would appear inside a **accepting-values** for a dialog. It takes two arguments, the frame and a stream. *my-acceptor-function* doesn't need to call **accepting-values** itself, since that is done automatically.

  See the chapter 13 **Menus and dialogs in CLIM**, and see the function **accept-values-pane-displayer**.

`accept-values-pane` **[Command table]**

■ When you use an **accept-values** pane as one of the panes in a **define-application-frame**, you must inherit from this command table in order to get the commands that operate on the dialog.

It is recommended that an application's command table inherit from `user-command-table`. `user-command-table` inherits from `global-command-table`. If your application uses an `:accept-values` pane, then its command table must inherit from the `accept-values-pane` command table in order for it to work properly.

accept-values-pane-displayer **[Function]**

Arguments: *frame pane* &key displayer resynchronize-every-pass

■ When you use an `:accept-values` pane, the display-function must use **accept-values-pane-displayer**. See the section 13.4 **Using an :accept-values pane in a CLIM application frame**.

displayer is a function that is the body of an **accepting-values** dialog. It takes two arguments, the frame and a stream. The display-function must not call **accepting-values** itself, since that is done by **accept-values-pane-displayer**.

The *resynchronize-every-pass* argument is the same as it is for **accepting-values**.

13.5 Examples of menus and dialogs in CLIM

13.5.1 Example of using clim:accepting-values

This example sets up a dialog in the CLIM window stream that displays the current Month, Date, Hour and Minute (as obtained by a call to **get-universal-time**) and allows the user to modify those values. The user can select values to change by using the mouse to select values, typing in new values, and pressing Newline or Return. When done, the user selects 'End' to accept the new values, or 'Abort', to terminate without changes.

```
(defun reset-clock (stream)
 (multiple-value-bind (second minute hour day month)
   (decode-universal-time (get-universal-time))
  (declare (ignore second))
  (format stream "Enter the time~%")
  (restart-case
   (progn
    (clim:accepting-values (stream)
     (setq month (clim:accept 'integer :stream stream
                  :default month :prompt "Month"))
     (terpri stream)
     (setq day (clim:accept 'integer :stream stream
                :default day :prompt "Day"))
     (terpri stream)
     (setq hour (clim:accept 'integer :stream stream
                 :default hour :prompt "Hour"))
     (terpri stream)
     (setq minute (clim:accept 'integer :stream stream
                   :default minute :prompt "Minute")))
    ;; This could be code to reset the time, but instead
    ;; we're just printing it out
    (format t "~%New values: Month: ~D, Day: ~D, Time: ~D:~2,'0D."
     month day hour minute))
   (abort () (format t "~&Time not set")))))
```

Note that in CLIM, calls to **accept** do not automatically insert newlines. If you want to put each query on its own line of the dialog, use **terpri** between the calls to **accept**.

13.5.2 Example of using clim:accept-values-command-button

Here is the reset-clock example with the addition of a command button that will set the number of seconds to zero.

```
(defun reset-clock (stream)
 (multiple-value-bind (second minute hour day month)
   (decode-universal-time (get-universal-time))
  (format stream "Enter the time~%")
  (restart-case
   (progn
    (clim:accepting-values (stream)
     (setq month (clim:accept 'integer :stream stream
                  :default month :prompt "Month"))
     (terpri stream)
     (setq day (clim:accept 'integer :stream stream
                :default day :prompt "Day"))
     (terpri stream)
     (setq hour (clim:accept 'integer :stream stream
                 :default hour :prompt "Hour"))
     (terpri stream)
     (setq minute (clim:accept 'integer :stream stream
                   :default minute :prompt "Minute"))
     (terpri stream)
     (clim:accept-values-command-button (stream) "Zero seconds"
      (setq second 0)))
    ;; This could be code to reset the time, but
    ;; instead we're just printing it out
    (format t "~%New values: Month: ~D, Day: ~D, Time: ~D:~2,'0D:~2,'0D."
     month day hour minute second))
   (abort () (format t "~&Time not set")))))
```

13.5.3 Using :resynchronize-every-pass in clim:accepting-values

It often happens that the programmer wants to present a dialog where the individual fields of the dialog depend on one another. For example, consider a spreadsheet with seven columns representing the days of a week. Each column is headed with that day's date. If the user inputs the date of any single day, the other dates can be computed from that single piece of input.

If you build CLIM dialogs using **`accepting-values`** you can achieve this effect by using the `:resynchronize-every-pass` argument to **`accepting-values`** in conjunction with the `:default` argument to **`accept`**. There are three points to remember:

- The entire body of the **`accepting-values`** runs each time the user modifies any field. The body can be made to run an extra time by specifying `:resynchronize-every-pass` t. Code in the body may be used to enforce constraints among values.
- If the `:default` argument to **`accept`** is used, then every time that call to **`accept`** is run, it will pick up the new value of the default.
- Inside **`accepting-values`**, **`accept`** returns a third value, a boolean that indicates whether the returned value is the result of new input by the user, or is just the previously supplied default.

In this example we show a dialog that accepts two real numbers, delimiting an interval on the real line. The two values are labelled "Min" and "Max", but we wish to allow the user to supply a "Min" that is greater than the "Max", and automatically exchange the values rather than signalling an error.

```
(defun accepting-interval (&key (min -1.0) (max 1.0) (stream *query-io*))
 (clim:accepting-values (stream :resynchronize-every-pass t)
  (fresh-line stream)
  (setq min (clim:accept 'real :default min :prompt "Min"
                 :stream stream))
  (fresh-line stream)
  (setq max (clim:accept 'real :default max :prompt "Max"
                 :stream stream))
  (when (< max min) (rotatef min max)))
 (values min max))
```

(You may want to try this example after dropping the `:resynchronize-every-pass` and see the behavior. Without `:resynchronize-every-pass`, the constraint is still enforced, but the display lags behind the values and doesn't reflect the updated values immediately.)

13.5.4 Use of the third value from clim:accept in clim:accepting-values

As a second example, consider a dialog that accepts four real numbers that delimit a rectangular region in the plane, only we wish to enforce a constraint that the region be a square. We allow the user to input any of "Xmin", "Xmax", "Ymin" or "Ymax", but enforce the constraint that

```
Xmax - Xmin = Ymax - Ymin
```

This constraint is a little harder to enforce. Presumably a user would be very disturbed if a value that he or she had just input was changed. So for this example we follow a policy that says if the user changed an X value, then only change Y values to enforce the constraint, and vice versa. When changing values we preserve the center of the interval. (This policy is somewhat arbitrary and only for the purposes of this example.) We use the third returned value from **`accept`** to control the constraint enforcement.

```
(defun accepting-square (&key (xmin -1.0) (xmax 1.0) (ymin -1.0) (ymax 1.0)
                 (stream *query-io*))
 (let (xmin-changed xmax-changed ymin-changed ymax-changed ptype)
  (clim:accepting-values (stream :resynchronize-every-pass t)
   (fresh-line stream)
   (multiple-value-setq (xmin ptype xmin-changed)
    (clim:accept 'real :default xmin :prompt "Xmin"
              :stream stream))
   (fresh-line stream)
   (multiple-value-setq (xmax ptype xmax-changed)
    (clim:accept 'real :default xmax :prompt "Xmax"
              :stream stream))
   (fresh-line stream)
   (multiple-value-setq (ymin ptype ymin-changed)
    (clim:accept 'real :default ymin :prompt "Ymin"
              :stream stream))
   (fresh-line stream)
   (multiple-value-setq (ymax ptype ymax-changed)
    (clim:accept 'real :default ymax :prompt "Ymax"
```

```
            :stream stream))
  (cond ((or xmin-changed xmax-changed)
         (let ((y-center (/ (+ ymax ymin) 2.0))
               (x-half-width (/ (- xmax xmin) 2.0)))
           (setq ymin (- y-center x-half-width)
                 ymax (+ y-center x-half-width)))
         (setq xmin-changed nil xmax-changed nil))
        ((or ymin-changed ymax-changed)
         (let ((x-center (/ (+ xmax xmin) 2.0))
               (y-half-width (/ (- ymax ymin) 2.0)))
           (setq xmin (- x-center y-half-width)
                 xmax (+ x-center y-half-width)))
         (setq ymin-changed nil ymax-changed nil)))))
 (values xmin xmax ymin ymax))
```

13.5.5 A simple spreadsheet that uses dialogs

Here is an example of how you might use **accepting-values** to implement a spreadsheet. You should not really do this, because the performance won't be good and the interface is probably not what you want. The important thing this illustrates is that **accepting-values** can be used with CLIM's high level formatted output facilities.

```
(defun silly-spreadsheet (stream &optional (nrows 3) (ncols 4))
 (let ((result (make-array (list nrows ncols))))
  (clim:accepting-values (stream)
   (clim:formatting-table (stream :y-spacing
                            (floor (stream-line-height stream) 2))
    (dotimes (row nrows)
     (clim:formatting-row (stream)
      (dotimes (cell ncols)
       (clim:formatting-cell (stream :min-width "   ")
        (let* ((id (+ (* ncols row) cell))
               (default (or (aref result row cell) id)))
          (setf (aref result row cell)
                (clim:accept 'integer
                       :prompt nil :default default
                       :query-identifier id
                       :stream stream)))))))))
  result))
```

13.5.6 Examples of using clim:menu-choose

These examples show how to use **menu-choose**.

The simplest use of **menu-choose**. If each item is *not* a list, the entire item will be printed and the entire item is the value to be returned too.

```
(clim:menu-choose '("One" "Two" "Seventeen"))
```

If you want to return a value that is different from what was printed, the simplest method is as below. Each item is a list; the first element is what will be printed, the remainder of the list is treated as a plist --

the `:value` property will be returned. (Note `nil` is returned if you click on "Seventeen" since it has no `:value`.)

```
(clim:menu-choose '(("One" :value 1 :documentation "the loneliest number")
          ("Two" :value 2 :documentation "for tea")
          ("Seventeen" :documentation "what can be said about this?")))
```

The list of items you pass to **`menu-choose`** might also serve some other purpose in your application. In that case, it might not be appropriate to put the printed appearance in the first element. You can supply a `:printer` function which will be called on the item to produce its printed appearance.

```
(clim:menu-choose '(1 2 17)
     :printer #'(lambda (item stream) (format stream "~R" item)))
```

The items in the menu needn't be printed textually:

```
(clim:menu-choose '(circle square triangle)
 :printer #'(lambda (item stream)
        (case item
         (circle (clim:draw-circle* stream 0 0 10))
         (square (clim:draw-polygon*
              stream '(-8 -8 -8 8 8 8 8 -8)))
         (triangle (clim:draw-polygon*
               stream '(10 8 0 -10 -10 8))))))
```

The `:items` option of the list form of menu item can be used to describe a set of hierarchical menus.

```
(clim:menu-choose
 '(("Class: Osteichthyes"
   :documentation "Bony fishes"
   :style (nil :italic nil))
  ("Class: Chondrichthyes"
   :documentation "Cartilagenous fishes"
   :style (nil :italic nil)
   :items (("Order: Squaliformes" :documentation "Sharks")
         ("Order: Rajiformes" :documentation "Rays")))
  ("Class: Mammalia"
   :documentation "Mammals"
   :style (nil :italic nil)
   :items (("Order Rodentia"
         :items ("Family Sciuridae"
               "Family Muridae"
               "Family Cricetidae"
               ("..." :value nil)))
         ("Order Carnivora"
         :items ("Family: Felidae"
               "Family: Canidae"
               "Family: Ursidae"
               ("..." :value nil)))
         ("..." :value nil)))
  ("..." :value nil)))
```

13.5.7 Examples of using clim:menu-choose-from-drawer

This example displays in the window `*page-window*` the choices "One" through "Ten" in bold type face. When the user selects one, the string is returned along with the gesture that selected it.

```
(clim:menu-choose-from-drawer
 *page-window* 'string
 #'(lambda (stream type)
   (clim:with-text-face (:stream bold)
    (dotimes (count 10)
     (clim:present
      (string-capitalize
       (format nil "~R" (1+ count)))
      type :stream stream)
     (terpri stream)))))
```

Chapter 14 Incremental redisplay in CLIM

14.1 Concepts of incremental redisplay in CLIM

Some kinds of applications can benefit greatly by the ability to redisplay information on a window only when that information has changed. This feature, called *incremental redisplay*, can greatly ease the process of writing applications only parts of whose display is changing, and can improve the speed at which your application updates information on the screen. Incremental redisplay is very useful for programs that display a window of changing information, where some portions of the window are static, and some are continually changing. A dynamic process monitor (aka, 'Peek') is an example; this window displays the status of processes and other changing system information. Incremental redisplay allows you to redisplay pieces of the existing output differently, under your control.

CLIM's output recording mechanism provides the foundation for incremental redisplay. As an application programmer, you need to understand the concepts of output recording before learning how to use the techniques of incremental redisplay.

A way in which incremental redisplay is accomplished is to first create an updating output record by calling **`updating-output`**. The **`updating-output`** informs CLIM that some piece of output may change in the future, and identifies under what circumstance a branches of the output history is known not to have changed.

`redisplay` takes an output record and redisplays it. This essentially tells the CLIM to create that output record over from scratch. However, CLIM compares the results with the existing output and tries to do minimal redisplay.

The recommended way to use incremental redisplay is with the `:incremental-redisplay` pane option. It can either be a boolean which defaults to `nil` or a list consisting of a boolean followed by keyword option pairs. The only option currently supported is `:check-overlapping` which takes a boolean value and defaults to `t`. Specifying this option as `nil` improves performance but should only be used where the output produced by the display function does not contain overlapping output.

The `:incremental-redisplay` option is always used in conjunction with a pane display function (specified with `:display-function`). It wraps a top level **`updating-output`** (see below) around the call to the display function and makes **`redisplay-frame-pane`** call **`redisplay`** on the updating output record rather than invoking the display function. The display function *is* invoked the first time it is called (in order to create the initial updating output record) or if `:force-p t` is specified in the call to **`redisplay-frame-pane`**.

14.2 Using clim:updating-output

The main technique of incremental redisplay is to use **updating-output** to inform CLIM what output has changed, and use **redisplay** to recompute and redisplay that output.

The outermost call to **updating-output** identifies a program fragment that produces incrementally redisplayable output. A nested call to **updating-output** (that is, a call to **updating-output** that occurs during the execution of the body of the outermost **updating-output** and specifies the same stream) identifies an individually redisplayable piece of output, the program fragment that produces that output, and the circumstances under which that output needs to be redrawn.

The outermost call to **updating-output** executes its body, producing the initial version of the output, and returns a `updating-output-record` that captures the body in a closure. Each nested call to **updating-output** caches the values of its `:unique-id` and `:cache-value` arguments and the portion of the output produced by its body.

redisplay takes a `updating-output-record` and executes the captured body of **updating-output** over again. When a nested call to **updating-output** is executed during redisplay, **updating-output** decides whether the cached output can be reused or the output needs to be redrawn. This is controlled by the `:cache-value` argument to **updating-output**. If its value matches its previous value, the body would produce output identical to the previous output and thus is unnecessary. In this case the cached output is reused and **updating-output** does not execute its body. If the cache value does not match, the output needs to be redrawn, so **updating-output** executes its body and the new output drawn on the stream replaces the previous output. The `:cache-value` argument is only meaningful for nested calls to **updating-output**.

If the `:incremental-redisplay` pane option is used, CLIM supplies the outermost call to **updating-output**, saves the `updating-output-record`, and calls **redisplay**. The function specified by the `:display-function` pane option performs only the nested calls to **updating-output**.

If you use incremental redisplay without using the `:incremental-redisplay` pane option, you must perform the outermost call to **updating-output**, save the `updating-output-record`, and call **redisplay** yourself.

In order to compare the cache to the output record, two pieces of information are necessary:

- An association between the output being done by the program and a particular cache. This is supplied in the `:unique-id` option to **updating-output**.
- A means of determining whether this particular cache is valid. This is the `:cache-value` option to **updating-output**.

Normally, you would supply both options. The unique-id would be some data structure associated with the corresponding part of output. The cache value would be something in that data structure that changes whenever the output changes.

It is valid to give the `:unique-id` options without specifying a `:cache-value`. This is done to identify a superior in the hierarchy. By this means, the inferiors essentially get a more complex unique id when they are matched for output. (In other words, it is like using a telephone area code.) The cache without a cache value is never valid. Its inferiors always have to be checked.

It is also valid to give the `:cache-value` and not the `:unique-id`. In this case, unique ids are just assigned sequentially. So, if output associated with the same thing is done in the same order each time, it isn't necessary to invent new unique ids for each piece. This is especially true in the case of inferiors of a cache with a unique id and no cache value of its own. In this case, the superior marks the particular data

structure, whose components can change individually, and the inferiors are always in the same order and properly identified by their superior and the order in which they are output.

A unique id need not be unique across the entire redisplay, only among the inferiors of a given output cache; that is, among all possible (current and additional) uses you make of **`updating-output`** that are dynamically (not lexically) within another.

To make your incremental redisplay maximally efficient, you should attempt to give as many caches with `:cache-value` as possible. For instance, if you have a deeply nested tree, it is better to be able to know when whole branches have not changed than to have to recurse to every single leaf and check it. So, if you are maintaining a modification tick in the leaves, it is better to also maintain one in their superiors and propagate the modification up when things change. While the simpler approach works, it requires CLIM to do more work than is necessary.

14.3 CLIM Operators for Incremental Redisplay

The following functions are used to create an output record that should be incrementally redisplayed, and then to redisplay that record.

`updating-output` **[Macro]**

Arguments:

```
(stream &rest args
&key (record-type 'clim:standard-updating-output-record)
unique-id (id-test 'eql) cache-value (cache-test 'eql)
parent-cache output-record fixed-position all-new
&allow-other-keys) &body body
```

■ Informs the incremental redisplay module of the characteristics of the output done by *`body`* to *`stream`*.

For related information, see the section 14.2 **Using clim:updating-output**.

`unique-id`

Provides a means to uniquely identify this output. If *`unique-id`* is not supplied, CLIM will generate one that is guaranteed to be unique at the current redisplay level.

`id-test`

A function of two arguments that is used for comparing unique ids. The default is **`eql`**.

`cache-value`

A value that remains constant if and only if the output produced by *`body`* does not need to be recomputed. If the cache value is not supplied, CLIM will not use a cache for this piece of output.

`cache-test`

A function of two arguments that is used for comparing cache values. The default is **`eql`**.

`fixed-position`

Declares that the location of this output is fixed relative to its parent. When CLIM redisplays an output record which specified `:fixed-position` t, if the contents have not changed, the position of the output record will not change. If the contents have changed, CLIM assumes that the code will take care to preserve its position.

`all-new`

Indicates that all of the output done by *`body`* is new, and will never match output previously recorded.

record-type

The type of output record that should be constructed. This defaults to `standard-updating-output-record`.

redisplay [Function]

Arguments: *record stream* &key (check-overlapping t)

■ Causes the output of *record* to be recomputed by calling **redisplay-output-record** on *record*. CLIM redisplays the changes incrementally, that is, only redisplays those parts of the record that changed. *record* must be an output record created by a previous call to **updating-output**, and may be any part of the output history of *stream*.

The *check-overlapping* keyword insures that **redisplay** checks for overlapping records. It defaults to `t`. If you set it to `nil` it speeds up redisplay, at the risk of failing to draw some records due to overlap. If you are sure that no sibling records overlap, you can use this keyword to optimize redisplay.

See the section 14.4 **Example of incremental redisplay in CLIM**.

redisplay-output-record [Generic function]

Arguments: *record stream* &optional *check-overlapping x y parent-x parent-y*

■ Causes the output of *record* to be recomputed. CLIM redisplays the changes incrementally, that is, only redisplays those parts of the record that changed. *record* must be an output record created by a previous call to **updating-output**, and may be any part of the output history of *stream*.

The optional arguments can be used to specify where on the stream the output record should be redisplayed. *x* and *y* represent where the cursor should be, relative to the parent output record of record, before the record is redisplayed. The default values for *x* and *y* are the starting position of the output record.

parent-x and *parent-y* can be supplied to say: do the output as if the superior started at positions *parent-x* and *parent-y* (which are in absolute coordinates). The default values for *parent-x* and *parent-y* are the absolute coordinate of the output record's parent.

The *check-overlapping* argument insures that **redisplay** checks for overlapping records. It defaults to `t`. If you make it `nil` it speeds up redisplay, at the risk of failing to draw some records due to overlap. If you are sure that no sibling records overlap, you can use this argument to optimize redisplay.

You can specialize this generic function for your own classes of output records.

14.4 Example of incremental redisplay in CLIM

The following example illustrates the standard use of incremental redisplay, using the `:incremental-redisplay` pane option:

```
(define-application-frame test
    ()
  ((list :accessor test-list :initform (list 1 2 3 4 5)))
  (:panes
   (display :application
            :display-function 'test-display
            :incremental-redisplay t))
  (:layouts
```

```
   (default
       display)))

(defmethod test-display ((frame test) stream)
  (let ((list (test-list frame)))
    (do* ((elements list (cdr elements))
          (count 0 (1+ count))
          (element (first elements) (first elements)))
         ((null elements))
      (clim:updating-output (stream :unique-id count
                                    :cache-value element)
        (format stream "Element ~D" element)
        (terpri stream)))))

(define-test-command (com-change :menu t) ()
  (with-application-frame (frame)
    (setf (nth 2 (test-list frame)) 17)))
```

The initial display looks like:

```
Element 1
Element 2
Element 3
Element 4
Element 5
```

After the `change` command is executed, **`redisplay-frame-pane`** is called on the display pane and incremental redisplay causes the display to be updated to:

```
Element 1
Element 2
Element 17
Element 4
Element 5
```

Incremental redisplay can also be used on a CLIM stream pane without a pane display function and without specifying `:incremental-redisplay t`. Note that in this case it is the programmers responsibility to provide a top level **`updating-output`** and to explicitly call **`redisplay`** on that output record to force incremental redisplay.

```
(defun test (stream)
 (let* ((list (list 1 2 3 4 5))
     (record
      (clim:updating-output (stream)
       (do* ((elements list (cdr elements))
          (count 0 (1+ count))
          (element (first elements) (first elements)))
         ((null elements))
        (clim:updating-output (stream :unique-id count
                       :cache-value element)
         (format stream "Element ~D" element)
         (terpri stream))))))
  (sleep 10)
```

```
  (setf (nth 2 list) 17)
  (clim:redisplay record stream)))
```

Here is an example of using incremental redisplay in conjunction with graph formatting.

```
(defun redisplay-graph (stream)
 (macrolet ((make-node (&key name children)
              `(list* ,name ,children))
            (node-name (node)
              `(car ,node))
            (node-children (node)
              `(cdr ,node)))
  (let* ((3a (make-node :name "3A"))
         (3b (make-node :name "3B"))
         (2a (make-node :name "2A"))
         (2b (make-node :name "2B"))
         (2c (make-node :name "2C"))
         (1a (make-node :name "1A" :children (list 2a 2b)))
         (1b (make-node :name "1B" :children (list 2b 2c)))
         (root (make-node :name "0" :children (list 1a 1b)))
         (graph
          (clim:updating-output (stream :unique-id root)
           (clim:format-graph-from-root
            root
            #'(lambda (node s)
              (clim:updating-output (s :cache-value node)
               (write-string (node-name node) s)))
            #'cdr  ;really #'node-children
            :stream stream))))
   (sleep 2)
   (setf (node-children 2a) (list 3a 3b))
   (clim:redisplay graph stream)
   (sleep 2)
   (setf (node-children 2a) nil)
   (clim:redisplay graph stream))))
```

The following is a long example of a graphical CLOS class browser. It is meant to demonstrate, in some detail, the interaction between incremental redisplay and the high-level formatted output facilities (in this case, graph formatting). Note particularly the use of a tick that is used to drive when redisplay occurs for the nodes of the graph. Changing an inferior node propagates tick up the graph so that superior nodes properly redisplay as well. This example is a good guide for writing any such application.

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

;;; Class browser nodes

(defclass class-browser-node ()
  ((object :reader node-object :initarg :object)
   (inferiors :accessor node-inferiors :initform nil)
   (superiors :accessor node-superiors :initform nil)
   (tick :accessor node-tick :initform 0)))

(defun make-class-browser-node (object)
 (make-instance 'class-browser-node :object object))
```

```
(defmethod node-object-name ((node class-browser-node))
 (class-name (node-object node)))

(defmethod display-node ((node class-browser-node) stream)
 (updating-output (stream :unique-id node
             :cache-value (node-tick node))
  (let ((class (node-object node)))
   (with-output-as-presentation (stream node 'class-browser-node)
    (with-output-as-presentation (stream class 'class)
     (write (node-object-name node) :stream stream))))))

;; Propagate ticks up the graph to get proper redisplay.
(defmethod tick-node ((node class-browser-node))
 (labels ((tick (node)
       (incf (node-tick node))
       (dolist (superior (node-superiors node))
        (tick superior))))
  (declare (dynamic-extent #'tick))
  (tick node)))

(defun make-class-browser-root (object)
 (typecase object
  (class
   (make-class-browser-node object))
  (symbol
   (let ((class (find-class object nil)))
    (when class
     (make-class-browser-node class))))))

(defmethod node-generate-inferior-objects ((node class-browser-node))
 (class-direct-subclasses (node-object node)))

(defmethod node-any-inferior-objects-p ((node class-browser-node))
 (not (null (class-direct-subclasses (node-object node)))))

(defun node-eql (n1 n2)
 (eql (node-object n1) (node-object n2)))


;;; The CLASS presentation type

(define-presentation-type class ()
 :history t)

(define-presentation-method accept ((type class) stream (view textual-view)
&key default)
 (let* ((class-name (accept 'symbol :stream stream :view view
                    :default (and default (class-name default))
                    :prompt nil))
     (class (find-class class-name nil)))
  (unless class
   (input-not-of-required-type class-name type))
  class))
```

```
(define-presentation-method present (class (type class) stream (view textual-
view) &key)
 (prin1 (class-name class) stream))

;;; The class browser itself

(define-application-frame class-browser ()
  ((tree-depth :initform 1)
   (root-nodes :initform nil)
   (all-nodes :initform nil))
 (:command-definer t)
 (:command-table (class-browser :inherit-from (accept-values-pane)))
 (:panes
  (graph :application
      :display-function 'display-graph-pane
      :display-after-commands t
      :incremental-redisplay t
      :scroll-bars :both
      :end-of-page-action :allow
      :end-of-line-action :allow)
  (interactor :interactor :height '(5 :line)))
 (:layouts
  (default
   (vertically ()
    (4/5 graph)
    (1/5 interactor)))))

(defmethod display-graph-pane ((browser class-browser) stream)
 (let ((root-nodes (slot-value browser 'root-nodes)))
  (when root-nodes
   (updating-output (stream :unique-id root-nodes)
    (format-graph-from-roots root-nodes #'display-node #'node-inferiors
                :graph-type :dag
                :stream stream
                :orientation :horizontal
                :merge-duplicates t)))))

(defmethod generate-class-graph ((browser class-browser) nodes
                &optional (depth (slot-value browser 'tree-depth)))
 (when nodes
  (let ((generated nil))
   (labels
    ((collect-inferiors (node parent-node depth)
      (when (and (plusp depth)
           (not (eql node parent-node)))
       (let ((inferior-objects
           (node-generate-inferior-objects node)))
        (when inferior-objects
         (setq generated t)       ;we generated something
         (dolist (object inferior-objects)
          (let ((inferior-node
              (find-node-for-object browser object)))
           (unless (member node (node-superiors inferior-node))
```

```
            (setf (node-superiors inferior-node)
               (nconc (node-superiors inferior-node) (list node))))
            (unless (member inferior-node (node-inferiors node)
                    :test #'node-eql)
             (setf (node-inferiors node)
               (nconc (node-inferiors node) (list inferior-node))))
            ;; Recursively collect inferiors for these nodes
            (collect-inferiors inferior-node node (1- depth)))))))))
    (declare (dynamic-extent #'collect-inferiors))
    (dolist (node nodes)
     (collect-inferiors node nil depth)))
   generated)))

;; Find or intern a new node.
(defmethod find-node-for-object ((browser class-browser) object &key (test
#'eql))
 (with-slots (all-nodes) browser
  (dolist (node all-nodes)
   (when (funcall test object (node-object node))
    (return-from find-node-for-object node)))
  (let ((node (make-class-browser-node object)))
   (setq all-nodes (nconc all-nodes (list node)))
   node)))
(define-class-browser-command (com-show-graph :name t :menu t)
  ((objects '(sequence class)
       :prompt "some class names"
       :default nil))
 (with-slots (root-nodes all-nodes) *application-frame*
  (setq root-nodes (mapcar #'make-class-browser-root objects))
  ;; ALL-NODES and ROOT-NODES must not be EQ lists...
  (setq all-nodes (copy-list root-nodes))
  (window-clear (get-frame-pane *application-frame* 'graph))
  (generate-class-graph *application-frame* root-nodes)
  (redisplay-frame-pane *application-frame* 'graph :force-p t)))

(define-gesture-name :show-graph :pointer-button (:left :shift))

(define-presentation-to-command-translator show-graph
  (class-browser-node com-show-graph class-browser
   :gesture :show-graph)
  (object)
 (list (list (node-object object))))

(define-class-browser-command com-show-node-inferiors
  ((node 'class-browser-node :prompt "node to show inferiors for"))
 (when (generate-class-graph *application-frame* (list node) 1)
  (tick-node node)))

(define-presentation-to-command-translator show-node-inferiors
  (class-browser-node com-show-node-inferiors class-browser
  :gesture :select
  :tester ((object)
       (node-any-inferior-objects-p object)))
  (object)
```

```
 (list object))

(define-class-browser-command com-hide-node-inferiors
  ((node 'class-browser-node :prompt "node to hide inferiors of"))
 (when (node-inferiors node)
  (setf (node-inferiors node) nil)
  (tick-node node)))

(define-presentation-to-command-translator hide-node-inferiors
  (class-browser-node com-hide-node-inferiors class-browser
   :gesture :describe
   :tester ((object)
       (not (null (node-inferiors object)))))
  (object)
 (list object))

(define-class-browser-command com-delete-node
  ((node 'class-browser-node :prompt "node to delete"))
 (when (node-superiors node)
  (dolist (superior (node-superiors node))
   (setf (node-inferiors superior) (delete node (node-inferiors superior))))
  (tick-node node)))

(define-presentation-to-command-translator delete-node
  (class-browser-node com-delete-node class-browser
   :gesture :delete
   :tester ((object) (and (null (node-inferiors object))
              (not (null (node-superiors object))))))
  (object)
 (list object))

(define-class-browser-command (com-redisplay-graph :name t :menu "Redisplay")
()
 (redisplay-frame-pane *application-frame* 'graph :force-p t))

(define-class-browser-command (com-quit-browser :name "Quit" :menu "Quit") ()
 (frame-exit *application-frame*))

(defun do-class-browser (&key (port (find-port)) (force nil))
 (find-application-frame 'class-browser
  :frame-manager (find-frame-manager :port port)
  :own-process nil :create (if force :force t)))
```

Chapter 15 Manipulating the pointer in CLIM

15.1 Manipulating the pointer in CLIM

A pointer is an input device that enables pointing at an area of the screen (for example, a mouse or a a tablet). CLIM offers a set of operators that enable you to manipulate the pointer.

The following functions allow you to directly manipulate the pointer. You should be careful when you use any functions to set the pointer position. It is widely considered best to avoid creating user interfaces where the pointer jumps around unexpectedly.

`port-pointer` [Generic function]

Arguments: *port*

- Returns the pointer object corresponding to the primary pointing device for the port *port*.

`port-modifier-state` [Generic function]

Arguments: *port*

- Returns the state of the modifier keys for the port *port*. This is a bit-encoded integer that can be checked against the values of `+shift-key+`, `+control-key+`, `+meta-key+`, `+super-key+`, and `+hyper-key+`. When the bit is on, the corresponding key is being held down on the keyboard.

`pointer-button-state` [Generic function]

Arguments: *pointer*

- Returns the current state of the buttons of *pointer* as an integer. This will be a mask consisting of the **logior** of `+pointer-left-button+`, `+pointer-middle-button+`, and `+pointer-right-button+`.

`pointer-sheet` [Generic function]

Arguments: *pointer*

- Returns the sheet over which the *pointer* is currently located.

`pointer-position` [Generic function]

Arguments: *pointer*

- This function returns the position (two coordinate values) of the *pointer* in the coordinate system of the sheet that the pointer is currently over.

  You can use **`pointer-set-position`** to set the pointer's position.

pointer-set-position [Generic function]

Arguments: *pointer x y*

■ This function changes the position of the *pointer* to be (*x*,*y*). *x* and *y* are in the coordinate system of the sheet that the pointer is currently over.

pointer-native-position [Generic function]

Arguments: *pointer*

■ This function returns the position (two coordinate values) of the *pointer* in the coordinate system of the port's graft (that is, its root window).

You can use **pointer-set-position** to set the pointer's native position.

pointer-set-native-position [Generic function]

Arguments: *pointer x y*

■ This function changes the position of the *pointer* to be (*x*,*y*). *x* and *y* are in the coordinate system of the port's graft (that is, its root window).

pointer-cursor [Generic function]

Arguments: *pointer*

■ Returns the type of the cursor presently being used by the pointer. This will be one of

:busy	:lower-right	:scroll-right
:button	:move	:scroll-up
:default	:position	:upper-left
:horizontal-scroll	:prompt	:upper-right
:horizontal-thumb	:scroll-down	:vertical-scroll
:lower-left	:scroll-left	:vertical-thumb

You can temporarily change the pointer's cursor by calling **setf** on **pointer-cursor**. However, the window manager may change the pointer cursor to something else very quickly, so for this reason it is generally preferable to modify **sheet-pointer-cursor** of the affected sheets instead.

stream-pointer-position [Generic function]

Arguments: *stream* &key pointer

■ This function returns the position (two coordinate values) of the *pointer* in the *stream*'s coordinate system. If *pointer* is not supplied, it defaults to **port-pointer** of the stream's port.

This function is usually used in preference to **pointer-position** when you have your hands on a CLIM stream.

You can use **stream-set-pointer-position** to set the pointer's position.

`stream-set-pointer-position` [Generic function]

Arguments: *stream x y* &key pointer

■ This function sets the position (two coordinate values) of the *pointer* in the *stream*'s coordinate system. If the port cannot set the pointer's position, this leaves the pointer where it was. If *pointer* is not supplied, it defaults to **port-pointer** of the stream's port.

`sheet-pointer-cursor` [Generic function]

Arguments: *sheet*

■ Returns the type of the cursor that will be used by the pointer when it is over the sheet *sheet*. This will be one of

:busy	:lower-right	:scroll-right
:button	:move	:scroll-up
:default	:position	:upper-left
:horizontal-scroll	:prompt	:upper-right
:horizontal-thumb	:scroll-down	:vertical-scroll
:lower-left	:scroll-left	:vertical-thumb

■ You can change the pointer cursor for a sheet by calling **setf** on **sheet-pointer-cursor**.

15.2 High Level Operators for Tracking the Pointer in CLIM

Sometimes it is useful to be able to track the pointer directly without having to resort to manage events at the level of **handle-event**. The following operators provide convenient interfaces for doing this.

`tracking-pointer` [Macro]

Arguments: (&optional *stream* &key pointer multiple-window transformp (context-type t) highlight) &body *clauses*

■ **tracking-pointer** a general means for running code while following the position of a pointing device on the stream *stream*, and monitoring for other input events. User-supplied code may be run upon occurrence of any of the following types of events:

- Motion of the pointer
- Motion of the pointer over a presentation
- Clicking or releasing a pointer button
- Clicking or releasing a pointer button on a presentation
- Keyboard event (typing a character)

IMPLEMENTATION LIMITATION: **tracking-pointer** will not work across multiple windows with a mouse button down.

■ *stream* defaults to *standard-output*.

■ The keyword arguments to **tracking-pointer** are:

pointer

Specifies a pointer to track. It defaults to `(port-pointer (port` *stream*`))`. Unless there is more than one pointing device available, it is unlikely that this option will be useful.

multiple-window

A boolean that specifies that the pointer is to be tracked across multiple windows. The default is `nil`.

transformp

A boolean that specifies that coordinates supplied to the `:pointer-motion` clause are to be transformed by the current user transformation. If `nil` (the default), the coordinates are supplied as ordinary stream coordinates.

context-type

A presentation type specifier that indicates what type of presentations that will be visible to the tracking code. It defaults to `t`, meaning that all presentations are visible.

highlight

A boolean that specifies whether or not CLIM should highlight presentations. It defaults to `t` if there are any presentation clauses in the body (`:presentation`, `:presentation-button-press`, or `:presentation-button-release`), meaning that presentations should be highlighted. Otherwise it defaults to `nil`.

■ The body of **`tracking-pointer`** consists of *`clauses`*. Each clause in *`clauses`* is of the form (*`clause-keyword arglist`* `&body` *`clause-body`*) and defines a local function to be run upon occurrence of each type of event. The possible *`clause-keywords`*, their *`arglists`*, and their uses are given next.

IMPLEMENTATION LIMITATION: The arglists differ from those in the CLIM 2 specification in that the arguments are not keyword arguments (as called for in the spec). The difference first appeared in code shared among CLIM 2 implementors and not noticed before it was too late to change.

`:presentation` (*`presentation window x y`*)

Defines a clause to run whenever the user moves the pointer over a presentation of the desired type. (See the keyword argument `:context-type` above for a description of how to specify the desired type.) In the clause, *`presentation`* is bound to the presentation, *`window`* to the window in which the presentation was found, and *`x`* and *`y`* to the coordinates of the pointer. (See the keyword argument `:transformp` above for a description of the coordinate system in which *`x`* and *`y`* is expressed.)

`:pointer-motion` (*`window x y`*)

Defines a clause to run whenever the user moves the pointer. In the clause, *`window`* is bound to the window in which the motion occurred, and *`x`* and *`y`* to the coordinates of the pointer. (See the keyword argument `:transformp` above for a description of the coordinate system in which *`x`* and *`y`* is expressed.) When both `:presentation` and `:pointer-motion` clauses are provided, the two clauses are mutually exclusive in that only one of them will be run when the pointer moves. The `:presentation` clause will run if the pointer is over an applicable presentation, otherwise the `:pointer-motion` clause will run.

`:presentation-button-press` (*`presentation event x y`*)

Defines a clause to run whenever the user presses a pointer button over a presentation of the desired type. (See the keyword argument `:context-type` above for a description of how to specify the desired type.) In the clause, *`presentation`* is bound to the presentation, and *`event`* to the event object. (The window and the coordinates of the pointer are part of *`event`*.)

x and *y* are the transformed x and y positions of the pointer. These will be different from **pointer-event-x** and **pointer-event-y** if the window's user transformation is a non-identity transformation and *transformp* is non-nil.

:pointer-button-press (*event x y*)

Defines a clause to run whenever the user presses a pointer button. In the clause, *event* is bound to the event object. (The window and the coordinates of the pointer are part of *event*.) When both :presentation-button-press and :pointer-button-press clauses are provided, the two clauses are mutually exclusive in that only one of them will be run when the button is pressed. The :presentation-button-press clause will run if applicable, otherwise the :pointer-button-press clause will run.

x and *y* are the transformed x and y positions of the pointer. These will be different from **pointer-event-x** and **pointer-event-y** if the window's user transformation is a non-identity transformation and *transformp* is non-nil.

:presentation-button-release (*presentation event x y*)

Defines a clause to run whenever the user releases a pointer button over a presentation. In the clause, *presentation* is bound to the presentation, and *event* to the event object. (The window and the coordinates of the pointer are part of *event*.)

x and *y* are the transformed x and y positions of the pointer. These will be different from **pointer-event-x** and **pointer-event-y** if the window's user transformation is a non-identity transformation and *transformp* is non-nil.

:pointer-button-release (*event x y*)

Defines a clause to run whenever the user releases a pointer button. In the clause, *event* is bound to the event object. (The window and the coordinates of the pointer are part of *event*.)

x and *y* are the transformed x and y positions of the pointer. These will be different from **pointer-event-x** and **pointer-event-y** if the window's user transformation is a non-identity transformation and *transformp* is non-nil.

When both :presentation-button-release and :pointer-button-release clauses are provided, the two clauses are mutually exclusive in that only one of them will be run when the button is released. The :presentation-button-release clause will run if applicable, otherwise the :pointer-button-release clause will run.

:keyboard (*character*)

Defines a clause to run whenever the user types a character. In the clause, *character* is bound to the character typed.

drag-output-record [Function]

Arguments: *stream output-record* &key (repaint t) multiple-window (erase #'clim:erase-output-record) feedback finish-on-release

■ Enters an interaction mode in which user moves the pointer, and *output-record* follows the pointer by being dragged on *stream*.

repaint

Allows you to specify the appearance of windows as the pointer is dragged. If *repaint* is t (the default), the displayed contents of the window is not disturbed as *output-record* is dragged over them (that is, those regions of the window are repainted as the record is dragged).

erase

Allows you to identify a function to erase the output record (the default is **erase-output-record**). *erase* is a function of two arguments, the output record to erase, and the stream.

feedback

Allows you to identify a feedback function. *feedback* a is a function of seven arguments: the output record, the stream, the initial x and y position of the pointer, the current x and y position of the pointer, and a drawing argument (either `:erase`, or `:draw`). Note that if *feedback* is supplied, *erase* is ignored.

You should supply *feedback* if you want more complex feedback than is supplied by default, for instance, if you want to draw a rubber band line as the user moves the mouse. The default for *feedback* is `nil`.

multiple-window

is as for **tracking-pointer**.

finish-on-release

When this is `nil`, the body is exited when the user clicks a mouse button. When this is `t` (the default), the body is exited when the user releases the mouse button currently being held down.

dragging-output **[Macro]**

Arguments: `(&optional` *stream* `&key (repaint t) multiple-window finish-on-release) &body` *body*

■ Evaluates *body* to produce the output, and then invokes **drag-output-record** to drag that output on *stream*. *stream* defaults to `*standard-output*`.

■ *repaint*, *multiple-window*, and *finish-on-release* are as for **drag-output-record**.

pointer-place-rubber-band-line* [Function]

Arguments: `&key stream pointer multiple-window start-x start-y finish-on-release`

■ This function can be used to input the end points of a line on the stream *stream*. *stream* defaults to `*standard-input*`. *pointer* and *multiple-window* are as for **tracking-pointer**. *finish-on-release* is as for **dragging-output**.

If *start-x* and *start-y* are provided, the start point of the line is at (*start-x*,*start-y*). Otherwise, the start point of the line is selected by pressing a button on the pointer.

pointer-place-rubber-band-line* returns four values, the start x and y positions, and the end x and y positions.

pointer-input-rectangle* [Function]

Arguments: `&key stream pointer multiple-window left top right bottom finish-on-release`

■ This function can be used to input the end corners of a rectangle on the stream *stream*. *stream* defaults to `*standard-input*`. *pointer* and *multiple-window* are as for **tracking-pointer**. *finish-on-release* is as for **dragging-output**.

If *left* and *top* are provided, the upper left corner of the rectangle is at (*left*,*top*). If *right* and *bottom* are provided, the lower right corner of the rectangle is at (*left*,*top*). Otherwise, the upper left corner of the rectangle is selected by pressing a button on the pointer.

pointer-input-rectangle* returns four values, the left, top, right, and bottom corners of the rectangles.

pointer-input-rectangle **[Function]**

Arguments: `&key stream pointer multiple-window rectangle finish-on-release`

■ This function is equivalent to **pointer-input-rectangle***, except that the initial corners of the rectangle are gotten from *rectangle*, and the return result is a bounding rectangle rather than four values.

15.2.1 Examples of Higher Level Pointer-Tracking Facilities

The following function could be used to input a line segment. Pressing a pointer button places the start of the line, and releasing it places the end of the line.

```
(defun pointer-input-line* (&optional (stream *standard-input*))
 (let (start-x start-y end-x end-y)
  (flet ((finish (event finish &optional press)
       (let ((x (pointer-event-x event))
          (y (pointer-event-y event))
          (window (event-sheet event)))
        (when (eq window stream)
         (cond (start-x
            (clim:with-output-recording-options (window :draw t :record nil)
             (clim:draw-line* window start-x start-y end-x end-y
                      :ink clim:+flipping-ink+))
            (clim:draw-line* window start-x start-y end-x end-y)
            (when finish
             (return-from pointer-input-line*
              (values start-x start-y end-x end-y))))
            (press (setq start-x x start-y y)))))))
   (declare (dynamic-extent #'finish))
   (tracking-pointer (stream)
    (:pointer-motion (window x y)
     (when (and start-x (eq window stream))
      (clim:with-output-recording-options (window :draw t :record nil)
       (when end-x
        (clim:draw-line* window start-x start-y end-x end-y
                :ink +flipping-ink+))
       (setq end-x x end-y y)
       (clim:draw-line* window start-x start-y end-x end-y
               :ink +flipping-ink+))))
    (:pointer-button-press (event)
     (finish event nil t))
    (:pointer-button-release (event)
     (finish event t))))))
```

The following function can be used to create a circle at the current pointer position, and reposition it by dragging it.

```
(defun place-circle (radius &optional (stream *standard-input*))
 (multiple-value-bind (x y) (clim:stream-pointer-position stream)
  (clim:dragging-output (stream)
   (clim:draw-circle* stream x y radius :filled nil))))
```

The following is used in CLIM's CAD demo program to move a component. `component` is the class and presentation type of a component in the demo. First the component is erased from its old position, then **`drag-output-record`** is used to drag the component around. Finally, the component is moved to its new location in the `design-area` pane.

```
(define-cad-demo-command (com-move-component :menu "Move")
  ((component 'component :gesture :select))
 (let ((stream (clim:get-frame-pane clim:*application-frame* 'design-area)))
  (draw-self component stream :ink +background-ink+)
  (multiple-value-bind (x y delta-x delta-y)
    (let ((*draw-connections* nil))
     (clim:drag-output-record
      stream component
      :repaint t
      :erase #'(lambda (c s)
            (draw-body c s :ink clim:+background-ink+))
      :finish-on-release t))
   (move component (- x delta-x) (+ *component-size* (- y delta-y))))
  (draw-self component stream)))
```

Chapter 16 Using gadgets in CLIM

16.1 Using gadgets in CLIM

CLIM supports the use of gadgets as panes within an application. The following sections describe the basic gadget protocol, and the various gadgets supplied by CLIM.

16.2 Basic gadget protocol in CLIM

Gadgets are panes that implement such common toolkit components as push buttons or scroll bars. Each gadget class has a set of associated generic functions that serve the same role that callbacks serve in traditional toolkits. For example, a push button has an ‘activate’ callback function that is invoked when its button is pressed; a scroll bar has a value changed callback that is invoked after its indicator has been moved.

The gadget definitions specified by CLIM are abstract in that the gadget definition does not specify the exact user interface of the gadget, but only specifies the semantics that the gadget should provide. For instance, it is not defined whether the user clicks on a push button with the mouse or moves the mouse over the button and then presses some key on the keyboard to invoke the activate callback. The user can control some high-level aspects of the gadgets (approximate size, orientation, and so on), but each toolkit implementation will specify the exact look and feel of their gadgets. Typically, the look and feel will be derived directly from the underlying toolkit. In Allegro CLIM, the underlying toolkit is Motif. (Of course, you can also use the portable gadgets by instantiating the appropriate concrete pane class.)

Every gadget has an id and a client, which are specified when the gadget is created. The client is notified via the callback mechanism when any important user interaction takes place. Typically, a gadget's client will be an application frame or a composite pane. Each callback generic function is invoked on the gadget, its client, the gadget id (described below), and other arguments that vary depending on the callback.

For example, the **`activate-callback`** takes three arguments, a gadget, the client, and the gadget-id. Assuming the you have defined an application frame called `button-test` that has a CLIM stream pane in the slot `output-pane`, you could write the following method:

```
(defmethod clim:activate-callback
      ((button clim:push-button) (client button-test) gadget-id)
 (with-slots (output-pane) client
  (format output-pane "The button ~S was pressed, client ~S, id ~S."
    button client gadget-id)))
```

One problem with this example is that it differentiates on the class of the gadget, not on the particular gadget instance. That is, the same method will run for every push button that has the `button-test` frame as its client.

One way to distinguish between the various gadgets is via the gadget id, which is also specified when the gadget is created. The value of the gadget id is passed as the third argument to each callback generic func-

tion. In this case, if you have two buttons, you might install `start` and `stop` as the respective gadget ids and then use **`eql`** specializers on the gadget ids. You could then refine the above as:

```
(defmethod clim:activate-callback
     ((button clim:push-button) (client button-test) (gadget-id (eql 'start)))
 (start-test client))

(defmethod clim:activate-callback
     ((button clim:push-button) (client button-test) (gadget-id (eql 'stop)))
 (stop-test client))

;; Create the start and stop push buttons
(clim:make-pane 'clim:push-button
 :label "Start"
 :client frame :id 'start)

(clim:make-pane 'clim:push-button
 :label "Stop"
 :client frame :id 'stop)
```

Still another way to distinguish between gadgets is to explicitly specify what function should be called when the callback is invoked. This is specified when the gadget is created by supplying an appropriate initarg. You could then rewrite the above example as follows:

```
;; No callback methods needed, just create the push buttons
(clim:make-pane 'clim:push-button
 :label "Start"
 :client frame :id 'start
 :activate-callback
  #'(lambda (gadget)
    (start-test (gadget-client gadget))))

(clim:make-pane 'clim:push-button
 :label "Stop"
 :client frame :id 'stop
 :activate-callback
  #'(lambda (gadget)
    (stop-test (gadget-client gadget))))
```

The following classes and functions constitute the basic protocol for all of CLIM's gadgets. See the section 16.3 **Abstract Gadgets in CLIM**.

16.2.1 Basic gadgets

This section describes the basic gadget classes on which CLIM builds it gadgets. You can use these classes to build gadgets of your own.

`gadget` **[Class]**

- The protocol class that corresponds to a gadget.

  All subclasses of `gadget` handle the four initargs `:id`, `:client`, `:armed-callback`, and `:disarmed-callback`, which are used to specify, respectively, the gadget id, client, armed callback, and disarmed callback of the gadget. These are described below.

The armed callback and disarmed callback are either `nil` or a function that takes a single argument, the gadget that was armed (or disarmed).

`basic-gadget` **[Class]**

■ The implementation class on which many CLIM gadgets are built. If you create your own kind of gadget, it will probably inherit from this class.

gadgetp [Function]

Arguments: *object*

■ Returns `t` if *object* is a gadget, otherwise returns `nil`.

gadget-id [Generic function]

Arguments: *gadget*

■ Returns the gadget id of the gadget *gadget*. The id is typically a simple Lisp object that uniquely identifies the gadget.

You can use **setf** to change the id of the gadget.

gadget-client [Generic function]

Arguments: *gadget*

■ Returns the client of the gadget *gadget*. The client is usually an application frame, but it could be another gadget (for example, in the case of a push button that is contained in a radio box).

You can use **setf** to change the gadget's client.

armed-callback [Generic function]

Arguments: *gadget client id*

■ This callback is invoked when the gadget *gadget* is armed. The exact definition of arming varies from gadget to gadget, but typically a gadget becomes armed when the pointer is moved into its region.

The default method for **armed-callback** (on `basic-gadget`) calls the function specified by the `:armed-callback` initarg if there is one, otherwise it does a **call-next-method**.

disarmed-callback [Generic function]

Arguments: *gadget client id*

■ This callback is invoked when the gadget *gadget* is disarmed. The exact definition of disarming varies from gadget to gadget, but typically a gadget becomes disarmed when the pointer is moved out of its region.

The default method for **disarmed-callback** (on `basic-gadget`) calls the function specified by the `:disarmed-callback` initarg if there is one, otherwise it does a **call-next-method**.

activate-gadget [Generic function]

Arguments: *gadget*

■ Causes the host gadget to become active, that is, available for input. The function **note-gadget-activated** is called once the gadget has been made active.

deactivate-gadget [Generic function]

Arguments: *gadget*

■ Causes the host gadget to become inactive, that is, unavailable for input. In some environments this may cause the gadget to become grayed over; in others, no visual effect may be detected. The function **note-gadget-deactivated** is called once the gadget has been made active.

gadget-active-p [Generic function]

Arguments: *gadget*

■ Returns `t` if the gadget is active, that is, available for input. Otherwise, it returns `nil`.

note-gadget-activated [Generic function]

Arguments: *client gadget*

■ This function is invoked after a gadget is made active.

note-gadget-deactivated [Generic function]

Arguments: *client gadget*

■ This function is invoked after a gadget is made inactive.

16.2.2 Value gadgets

`value-gadget` **[Class]**

■ The class used by gadgets that have a value; a subclass of `basic-gadget`.

All subclasses of `value-gadget` handle the two initargs `:value` and `:value-changed-callback`, which are used to specify, respectively, the initial value and the value changed callback of the gadget. The value changed callback is either `nil` or a function of two arguments, the gadget and the new value.

gadget-value [Generic function]

Arguments: *gadget*

■ Returns the value of the gadget *value-gadget*. The interpretation of the value varies from gadget to gadget. For example, a scroll bar's value might be a number between 0 and 1, while a toggle button's value is either `t` or `nil`. (The documentation of each individual gadget below specifies how to interpret the value.)

You can use **setf** on **gadget-value** to change the value of a gadget. Normally when you set the value of a gadget, the value change callback will not be invoked. If you want the value change callback to be called, specify `:invoke-callback` t, for example:

```
(setf (clim:gadget-value ... :invoke-callback t) ...)
```

value-changed-callback [Generic function]

Arguments: *gadget client id value*

■ This callback is invoked when the value of a gadget is changed, either by the user or programmatically.

The default method (on `value-gadget`) calls the function specified by the `:value-changed-callback` initarg with two arguments, the gadget and the new value, if such a function is specified. Otherwise, it does a **call-next-method** (just like **armed-callback**).

drag-callback [Generic function]

Arguments: *gadget client id value*

■ Some value gadgets, such as sliders and scroll bars, have a drag callback. This callback is invoked when the value of the slider or scroll bar is changed while the indicator is being dragged. This is implemented by calling the function specified by the `:drag-callback` initarg with two arguments, the slider (or scroll bar) and the new value, if such a function is specified. Otherwise, it does a **`call-next-method`** (just like **`armed-callback`**).

16.2.3 Action gadgets

`action-gadget` [Class]

■ The class used by gadgets that perform some kind of action, such as a push button; a subclass of `basic-gadget`.

All subclasses of `action-gadget` handle the `:activate-callback` initarg, which is used to specify the activate callback of the gadget. The activate callback is `nil` or a function of one argument, the gadget.

activate-callback [Generic function]

Arguments: *gadget client id*

■ This callback is invoked when the gadget is activated.

The default method (on `action-gadget`) calls the function specified by the `:activate-callback` initarg with one argument, the gadget, if such a function is specified. Otherwise, it does a **`call-next-method`** (just like **`armed-callback`**).

16.2.4 Other gadget classes

`oriented-gadget-mixin` [Class]

■ The class that is mixed in to a gadget that has an orientation associated with it, for example, a slider.

All subclasses of `oriented-gadget-mixin` handle the `:orientation` initarg, which is used to specify the orientation of the gadget.

gadget-orientation [Generic function]

Arguments: *oriented-gadget*

■ Returns the orientation of the gadget *`oriented-gadget`*. Typically, this will be a keyword such as `:horizontal` or `:vertical`.

`row-column-gadget-mixin` [Class]

■ This class is mixed in to a gadget that has a row or column associated with it, for example `radio-box` and `check-box`, both of which inherit from this class. All subclasses handle the `:rows` and `:columns` initargs, only one of which should be specified. These are accessed with **`gadget-rows`** and **`gadget-columns`**.

gadget-columns [Generic function]

gadget-rows [Generic function]

Arguments: *row-column-gadget*

- Returns the number of columns (**gadget-columns**) or rows (**gadget-rows**) in *row-column-gadget*.

labelled-gadget-mixin [Class]

- The class that is mixed in to a gadget that has a label, for example, a push button.

  All subclasses of `labelled-gadget-mixin` handle the initargs `:label`, `:alignment`, and `:text-style`, which are used to specify the label, the label's alignment within the gadget, and the label's text style. The label can be a string or a pixmap or a pattern.

gadget-label [Generic function]

Arguments: *labelled-gadget*

- Returns the label of the gadget *labelled-gadget*. The label must be a string or a pixmap.

  You can use **setf** to change the label of a gadget, but this may result in invoking the layout protocol on the gadget and its ancestor sheets (that is, the entire application frame may be laid out again).

range-gadget-mixin [Class]

- The class that is mixed in to a gadget that has a range, for example, a slider.

  All subclasses of `range-gadget-mixin` handle the two initargs `:min-value` and `:max-value`, which are used to specify the minimum and maximum value of the gadget.

gadget-min-value [Generic function]

Arguments: *range-gadget*

- Returns the minimum value of the gadget *range-gadget*. It will be a real number.
- You can use **setf** to change the minimum value of the gadget.

gadget-max-value [Generic function]

Arguments: *range-gadget*

- Returns the maximum value of the gadget *range-gadget*. It will be a real number.
- You can use **setf** to change the maximum value of the gadget.

16.3 Abstract gadgets in CLIM

Many gadgets in CLIM, such as push buttons and sliders, are abstract gadgets. This is because the classes, such as `push-button` and `slider`, do not themselves implement gadgets, but rather arrange for the frame manager layer of CLIM to create concrete gadgets that correspond to the abstract gadgets. The callback interface to all of the various implementations of the gadget is defined by the abstract class. In the `:panes` clause of **define-application-frame**, the abbreviation for a gadget is the name of the abstract gadget class.

At pane creation time (that is, during **make-pane**), the frame manager resolves the abstract class into a specific implementation class; the implementation classes specify the detailed look and feel of the gadget. Each frame manager will keep a mapping from abstract gadgets to an implementation class; if the frame manager does not implement its own gadget for the abstract gadget classes in the following sections, it

should use the portable class provided by CLIM. Since every implementation class is a subclass of the abstract class, they all share the same programmer interface.

The following classes and functions comprise CLIM's abstract gadgets. See the section 16.2 **Basic gadget protocol in CLIM**.

make-pane [Function]

Arguments: *pane-class* &rest *pane-options*

■ Selects a class that implements the behavior of the abstract pane *pane-class* and constructs a pane of that class. **make-pane** must be used within the lexical scope of a call to **with-look-and-feel-realization**. That is automatically established within the :pane, :panes, and :layouts options of a **define-application-frame** and by certain macros like **with-output-as-gadget**.

push-button [Class]

■ The push-button gadget class provides press-to-activate switch behavior. It is a subclass of action-gadget and labelled-gadget-mixin. In addition to the usual pane initargs (:foreground, :background, space requirement options, and so forth), the following initargs are supported:

:show-as-default

when t, the push button will be marked as a default button following the look and feel conventions of the frame-manager.

■ Note that in order to associate a pattern with a push-button you should use the :label option inherited from labelled-gadget-mixin.

■ The :pattern initarg is no longer supported.

IMPLEMENTATION LIMITATION: The following initarg, :external-label, is not implemented in the current release.

:external-label

If supplied, this is a string that will be used as a label that is drawn outside of the push button, instead of inside of the button.

■ When the button is actually activated (by releasing the pointer button over it), **activate-callback** will be invoked. Finally, **disarmed-callback** will be invoked after **activate-callback**, or when the pointer is moved outside of the button. (Note that Motif does not support armed/disarmed callbacks on push-buttons.)

A push button might be created as follows:

```
(clim:make-pane 'clim:push-button
 :label "Button"
 :activate-callback 'push-button-callback)

(defun push-button-callback (button)
 (format t "~&Button ~A pushed" (clim:gadget-label button)))
```

push-button-view [Class]

■ The view class associated with push buttons. The view class options include all the options available for the push-button gadget-class. It is useful for specifying a :view argument to **accept-values-command-button**.

+push-button-view+ [Constant]

■ An instance of the class push-button-view.

push-button-show-as-default [Generic function]

Arguments: *push-button*

■ Returns the show as default slot of the push button gadget

toggle-button [Class]

■ The `toggle-button` gadget class provides on/off switch behavior. It is a subclass of `value-gadget` and `labelled-gadget-mixin`. This gadget typically appears as a box that is optionally highlighted with a check-mark. If the check-mark is present, the gadget's value is `t`, otherwise it is `nil`.

In addition to the usual pane initargs (`:foreground`, `:background`, `:text-style`, space requirement options, and so forth), the following initargs are supported:

`:indicator-type`

This is used to initialize the indicator type property for the gadget. This will be either `:one-of` or `:some-of`. The indicator type controls the appearance of the toggle button. For example, many toolkits present a one-of-many choice differently from a some-of-many choice.

■ When the toggle button is actually activated (by releasing the pointer button over it), `value-changed-callback` will be invoked. Finally, **`disarmed-callback`** will be invoked after `value-changed-callback`, or when the pointer is moved outside of the toggle button. (Note that Motif does not support armed/disarmed callbacks on toggle-buttons.)

Calling **`gadget-value`** on a toggle button will return `t` if the button is selected, otherwise it will return `nil`. The value of the toggle button can be changed by calling **`setf`** on **`gadget-value`**.

A toggle button might be created as follows:

```
(clim:make-pane 'clim:toggle-button
 :label "Toggle" :width 80
 :value-changed-callback 'toggle-button-callback)

(defun toggle-button-callback (button value)
 (format t "~&Button ~A toggled to ~S" (clim:gadget-label button) value))
```

radio-box [Class]

■ A radio box is a special kind of gadget that constrains one or more toggle buttons. At any one time, only one of the buttons managed by the radio box may be 'on'. The contents of a radio box are its buttons, and as such a radio box is responsible for laying out the buttons that it contains.

It is a subclass of `value-gadget` and `oriented-gadget-mixin`.

In addition to the usual pane initargs (`:foreground`, `:background`, space requirement options, and so forth), the following initargs are supported:

`:current-selection`

This is used to specify which button, if any, should be initially selected.

`:choices`

This is used to specify all of the buttons that serve as choices.

■ As the current selection changes, the previously selected button and the newly selected button both have their `value-changed-callback` handlers invoked.

Calling **`gadget-value`** on a radio box will return the currently selected toggle button. The value of the radio box can be changed by calling **`setf`** on **`gadget-value`**.

A radio box might be created as follows, although it is generally more convenient to use **with-radio-box**:

```
(let* ((choices
     (list (clim:make-pane 'clim:toggle-button :label "One" :width 80)
        (clim:make-pane 'clim:toggle-button :label "Two" :width 80)
        (clim:make-pane 'clim:toggle-button :label "Three" :width 80)))
    (current (second choices)))
 (clim:make-pane 'clim:radio-box
  :choices choices
  :selection current
  :value-changed-callback 'radio-value-changed-callback))

(defun radio-value-changed-callback (radio-box value)
 (declare (ignore radio-box))
 (format t "~&Radio box toggled to ~S" value))
```

`check-box` **[Class]**

■ A check box is similar to a radio box: it is a special kind of gadget that constrains one or more toggle buttons. At any one time, zero or more of the buttons managed by the check box may be 'on'. The contents of a check box are its buttons, and as such a check box is responsible for laying out the buttons that it contains.

It is a subclass of `value-gadget` and `oriented-gadget-mixin`.

In addition to the usual pane initargs (`:foreground`, `:background`, space requirement options, and so forth), the following initargs are supported:

`:selection`

This is used to specify which button, if any, should be initially selected.

`:choices`

This is used to specify all of the buttons that serve as choices.

■ As the user changes the selections, the newly selected (or deselected) button will have its `value-changed-callback` handler invoked.

Calling **gadget-value** on a check box will return a sequence of the currently selected toggle buttons. The value of the check box can be changed by calling **setf** on **gadget-value**.

A check box might be created as follows, although it is generally more convenient to use **with-radio-box**:

```
(let* ((choices
     (list (clim:make-pane 'clim:toggle-button :label "One" :width 80)
        (clim:make-pane 'clim:toggle-button :label "Two" :width 80)
        (clim:make-pane 'clim:toggle-button :label "Three" :width 80)))
    (current (second choices)))
 (clim:make-pane 'clim:check-box
  :choices choices
  :selection (list current)
  :value-changed-callback 'radio-value-changed-callback))

(defun radio-value-changed-callback (radio-box value)
 (declare (ignore radio-box))
 (format t "~&Radio box toggled to ~S" value))
```

with-radio-box **[Macro]**

Arguments: `(&rest` *options* `&key (type ':one-of) &allow-other-keys)` `&body` *body*

■ Creates a radio box or a check box whose buttons are created by the forms in *body*. The macro **radio-box-current-selection** (or **check-box-current-selection**) can be wrapped around one of forms in *body* in order to indicate that that button is the current selection. If `:type` is `:one-of`, this macro creates a radio box. If `:type` is `:some-of`, it creates a check box.

For example, the following creates a radio box with three buttons in it, the second of which is initially selected.

```
(clim:with-radio-box ()
 (clim:make-pane 'clim:toggle-button :label "Mono")
 (clim:radio-box-current-selection
  (clim:make-pane 'clim:toggle-button :label "Stereo"))
 (clim:make-pane 'clim:toggle-button :label "Quadraphonic"))
```

The following simpler form can also be used when you do not need to control the appearance of each button closely:

```
(clim:with-radio-box ()
 "Mono" "Stereo" "Quadraphonic")
```

`list-pane` **[Class]**

■ The `list-pane` gadget class corresponds to a list pane, that is, a pane whose semantics are similar to a radio box or check box, but whose visual appearance is a list of buttons. It is a subclass of `value-gadget`.

In addition to the usual pane initargs (`:foreground`, `:background`, space requirement options, and so forth), the following initargs are supported:

`:mode`

Either `:nonexclusive` or `:exclusive`. When it is `:exclusive`, the list pane acts like a radio box, that is, only a single item can be selected. Otherwise, the list pane acts like a check box, in that zero or more items can be selected. The default is `:exclusive`.

`:items`

A list of items.

`:visible-items`

The number of items that should be visible at one time (the rest can be seen by scrolling). The value should be a positive integer. The default is toolkit-specific.

`:name-key`

A function of one argument that generate the name of an item from the item. The default is **princ-to-string**.

`:value-key`

A function of one argument that generate the value of an item from the item. The default is **identity**.

`:test`

A function of two arguments that compares two items. The default is **eql**.

`:scroll-bars`

Possible values are `nil`, `:horizontal`, `:vertical`, and `:both`, causing scroll bars in the indicated direction(s) or no scroll bars when `nil`.

- Calling **`gadget-value`** on a list pane will return the single selected item when the mode is `:exclusive`, or a sequence of selected items when the mode is `:nonexclusive`.

  The `value-changed-callback` is invoked when the select item (or items) is changed.

Here are some examples of list panes:

```
(clim:make-pane 'clim:list-pane
 :value "Symbolics"
 :test 'string=
 :value-changed-callback 'list-pane-changed-callback
 :items '("Franz" "Lucid" "Harlequin" "Symbolics"))

(clim:make-pane 'clim:list-pane
 :value '("Lisp" "C++")
 :mode :nonexclusive
 :value-changed-callback 'list-pane-changed-callback
 :items '("Lisp" "Fortran" "C" "C++" "Cobol" "Ada"))

(defun list-pane-changed-callback (tf value)
 (format t "~&List pane ~A changed to ~S" tf value))
```

`option-pane` **[Class]**

- The `option-pane` gadget class corresponds to an option pane, that is, a pane whose semantics are identical to a list pane with `radio-box` semantics, but whose visual appearance is a single push button which, when pressed, pops up a menu of selections. It is a subclass of `value-gadget`.

  In addition to the usual pane initargs (`:foreground`, `:background`, space requirement options, and so forth), the following initargs are supported:

  `:items`

  A list of items.

  `:name-key`

  A function of one argument that generate the name of an item from the item. The default is **`princ-to-string`**.

  `:value-key`

  A function of one argument that generate the value of an item from the item. The default is **`identity`**.

  `:test`

  A function of two arguments that compares two items. The default is **`eql`**.

- Calling **`gadget-value`** on an option pane will return the selected item.

  The `value-changed-callback` is invoked when the selected item is changed.

Here are some examples of option panes:

```
(clim:make-pane 'clim:option-pane
 :label "Select a vendor"
 :value "Franz"
 :test 'string=
 :value-changed-callback 'option-pane-changed-callback
```

```
 :items '("Franz" "Lucid" "Harlequin" "Symbolics"))

(clim:make-pane 'clim:option-pane
 :label "Select some languages"
 :value '("Lisp" "C++")
 :value-changed-callback 'option-pane-changed-callback
 :items '("Lisp" "Fortran" "C" "C++" "Cobol" "Ada"))

(defun option-pane-changed-callback (tf value)
 (format t "~&Option menu ~A changed to ~S" tf value))
```

`scroll-bar` **[Class]**

■ The `scroll-bar` gadget class corresponds to a scroll bar. It is a subclass of `value-gadget`, `oriented-gadget-mixin`, and `range-gadget-mixin`. The usual pane initargs (`:foreground`, `:background`, space requirement options, and so forth) may also be specified. The following two initargs are also supported:

`:size`

an integer giving the size of the scroll-bar slug in pixels

`:drag-callback`

see the documentation for **`drag-callback`**.

`slider` **[Class]**

■ The `slider` gadget class corresponds to a slider. It is a subclass of `value-gadget`, `oriented-gadget-mixin`, `range-gadget-mixin`, and `labelled-gadget-mixin`.

In addition to the usual pane initargs (`:foreground`, `:background`, `:orientation`, space requirement options, and so forth), the following initargs are supported:

`:drag-callback`

Specifies the drag callback for the slider.

`:show-value-p`

Whether the slider should show its current value.

`:decimal-places`

An integer that specifies the number of decimal places that should be shown if the current value is being shown.

IMPLEMENTATION LIMITATION: The remaining 5 initargs are called for in the CLIM 2 specification but not currently implemented:

`:min-label`

A string to use to label the minimum end of the slider.

`:max-label`

A string to use to label the maximum end of the slider.

`:range-label-text-style`

The text style to use for the min and max labels.

`:number-of-tick-marks`

The number of tick marks to draw on the slider.

`:number-of-quanta`

Either `nil` or an integer. If an integer, specifies the number of quanta in the slider. In this case the slider is not continuous, and can only assume a value that falls on one of the quanta.

■ The `drag-callback` callback is invoked when the value of the slider is changed while the indicator is being dragged. This is implemented by calling the function specified by the `:drag-callback` initarg with two arguments, the slider and the new value.

The `value-changed-callback` is invoked only after the indicator is released after dragging it.

Calling **`gadget-value`** on a slider will return a real number within the specified range of the slider.

Here are some examples of sliders (the unimplemented initargs are commented out):

```
(clim:make-pane 'clim:slider
 :label "A slider"
 :value-changed-callback 'slider-changed-callback
 :drag-callback 'slider-dragged-callback)

(clim:make-pane 'clim:slider
 :label "A slider with tick marks and range labels"
 ;:number-of-tick-marks 20
 ;:min-label "0" :max-label "20"
 :value-changed-callback 'slider-changed-callback
 :drag-callback 'slider-dragged-callback)

(clim:make-pane 'clim:slider
 :label "A vertical slider with visible value"
 :orientation :vertical
 :show-value-p t)

(clim:make-pane 'clim:slider
 :label "A very hairy quantized slider"
 :orientation :vertical
 ;:number-of-tick-marks 20
 ;:number-of-quanta 20
 :show-value-p t
 :min-value 0 :max-value 20
 ;:min-label "Min" :max-label "Max"
 :value-changed-callback 'slider-changed-callback
 :drag-callback 'slider-dragged-callback)

(defun slider-changed-callback (slider value)
 (format t "~&Slider ~A changed to ~S" (clim:gadget-label slider) value))

(defun slider-dragged-callback (slider value)
 (format t "~&Slider ~A dragged to ~S" (clim:gadget-label slider) value))
```

`menu-bar` **[Class]**

■ The gadget class that implements a menu-bar. A `menu-bar` will use the toolkit menu-bar. This is a subclass of `oriented-gadget-mixin`.

In addition to the usual pane initargs (`:foreground`, `:background`, space requirement options, and so forth), the following initarg is supported:

`:command-table`

Defaults to `nil` but you should specify a command-table for this gadget to be useful.

`text-field` **[Class]**

■ The gadget class that implements a text field. This is a subclass of both `value-gadget` and `action-gadget`.

The value of a text field is the text string.

In addition to the usual pane initargs (`:foreground`, `:background`, space requirement options, and so forth), the following initargs are supported:

`:editable-p`

When `nil`, the text field cannot be modified. When `t` (the default), the text field can be modified.

`:scroll-bars`

Possible values are `nil`, `:horizontal`, `:vertical`, and `:both`, causing scroll bars in the indicated direction(s) or no scroll bars when `nil`.

`text-editor` **[Class]**

■ The `text-editor` gadget class corresponds to a large field containing text, a subclass of `text-field`.

The value of a text editor is the text string.

In addition to the usual pane initargs (`:foreground`, `:background`, space requirement options, and so forth), the following initargs are supported:

`:editable-p`

When `nil`, the text field cannot be modified. When `t` (the default), the text field can be modified.

`:ncolumns`

An integer specifying the width of the text editor in characters.

`:nlines`

An integer specifying the height of the text editor in lines.

`:word-wrap`

A boolean that controls whether word-wrapping is enabled.

■ An example:

```
(clim:make-pane 'clim:text-editor
 :value "Isn't Lisp the greatest?"
 :value-changed-callback 'text-field-changed
 :ncolumns 40 :nlines 5)

(defun text-field-changed (tf value)
 (format t "~&Text field ~A changed to ~S" tf value))
```

■ Note that Motif text-editor widgets can be made to have Emacs-like keybindings. This will occur if the contents of the file *misc/dot-Xdefaults* (in the Allegro CL distribution) is included in your *.Xdefaults* file.

The following function accesses selected text in a gadget.

`gadget-current-selection` **[Function]**

Arguments: *`text-editor-or-text-field`*

■ Returns the selected text if there is any in *`text-editor-or-text-field`*. Returns `nil` if no text is selected.

with-output-as-gadget **[Macro]**

Arguments: (*stream* &rest *options*) &body *body*

■ Invokes *body* to create a gadget, and then creates a gadget output record that contains the gadget and installs it into the output history of the output recording stream *stream*. The returned value of *body* must be the gadget.

The options in *options* are passed as initargs to the call to **invoke-with-new-output-record** that ius used to create the new output record.

The *stream* argument is not evaluated. It must be a symbol that is bound to an output recording stream. If *stream* is t, *standard-output* is used.

body may have zero or more declarations as its first forms.

■ For example, the following could be used to create an output record containing a radio box that itself contains several toggle buttons (stream is a suitable stream and sequence is a list of items):

```
(with-output-as-gadget (stream)
  (let* ((radio-box
           (make-pane 'radio-box
             :client stream :id 'radio-box)))
    (dolist (item sequence)
      (make-pane 'toggle-button
        :label (princ-to-string (item-name item))
        :value (item-value item)
        :id item :parent radio-box))
    radio-box))
```

Here is a more complex (and somewhat contrived) example of a push button that calls back into the presentation type system to execute a command:

```
(with-output-as-gadget (stream)
  (make-pane 'push-button
    :label "Click here to exit"
    :activate-callback
      #'(lambda (button)
          (declare (ignore button))
          (throw-highlighted-presentation
            (make-instance 'standard-presentation
               :object '(com-exit ,*application-frmae*)
               :type 'command)
            *input-contaxt*
            (make-instance 'pointer-button-press-event
              :sheet (sheet-parent button)
              :x 0 :y 0
              :modifiers 0
              :button +pointer-left-button+)))))
```

A note about unmirrored application panes

If you create unmirrored panes with **with-output-as-gadget** in a window, then graphics operations on that window do not respect the bounding box of the children - that is they get drawn on. An unmirrored pane is a clim-stream-pane such as an application-pane. Using scroll bars on the pane works around the problem.

[This page intentionally left blank.]

Chapter 17 The CLIM input editor

17.1 Input editing and built-in keystroke commands in CLIM

CLIM provides a sophisticated facility for doing interactive input line editing. This section describes the input editing commands available while you are doing command-line input to a CLIM application.

Table 1 provides a list of the keystrokes that are built into CLIM.

17.1.1 Activation and delimiter gestures

Activation gestures

Activation gestures terminate an input sentence, such as a command or anything else being read by **`accept`**. When you enter an activation gesture, CLIM ceases reading input and executes the input that has been entered.

The default activation gestures are Return and Newline.

The following deal with the set of activation gestures:

- The **`with-activation-gestures`** macro
- The `*standard-activation-gestures*` variable
- The `*activation-gestures*` variable
- The **`activation-gesture-p`** function
- The `:activation-gestures` option to **`accept`**
- The `:additional-activation-gestures` option to **`accept`**

Delimiter gestures

Delimiter gestures terminate an input word, such as a recursive call to **`accept`**. There are no global default delimiter gestures; each presentation type that recursively calls **`accept`** specifies its own delimiter gestures and sometimes offers a way to change them (see the information under the heading **Command processor characters** below).

Delimiter gestures most commonly occur in command lines. When you type a delimiter gesture, CLIM's command processor moves on to read the next field in the command line.

- The following deal with the set of delimiter gestures:
- The **`with-delimiter-gestures`** macro
- The `*delimiter-gestures*` variable
- The **`delimiter-gesture-p`** function
- The `:delimiter-gestures` option to **`accept`**
- The `:additional-delimiter-gestures` option to **`accept`**

- The `:separator` option to `subset-completion`
- The `:separator` option to `subset`
- The `:separator` option to `subset-sequence`
- The `:separator` option to `subset-alist`

Abort gestures

When you type an abort gesture while an application is reading input, CLIM aborts the application by invoking the `abort` restart. By default, the `abort` restart is caught by **`default-frame-top-level`**, which will abort what the application frame is doing and read another command.

The default abort gesture is Control-Z in Allegro. Meta-Control-Z is the default for asynchronous aborts.

The set of abort gesture is maintained in the lists which are the values of `*abort-gestures*` and `*asynchronous-abort-gestures*`

Completion gestures

Several presentation types, such as `member` and `pathname`, support completion of partial inputs. When an application is accepting input of one of these types, you can enter a completion gesture and possibilities gesture. A completion gesture causes CLIM to complete the input that has been entered so far; if there is more than possible completion, CLIM completes it as much as possible. A possibilities gesture causes CLIM to display the possible completions of the input that has been entered so far.

The default completion gesture is Tab.

The default possibilities gesture is Control-? in Allegro. You can also click the right button of the pointer over a blank area on the window in order to cause CLIM to display a menu of possibilities.

The following deal with the set of completion gestures:

- The `*completion-gestures*` variable
- The `*possibilities-gestures*` variable
- The `*help-gestures*` variable

Command processor gestures

When an application is reading the type `command-or-form`, a command dispatcher gesture introduces a command rather than a form. The default command dispatcher is `#\:` (a colon). For example, in the CLIM Lisp Listener, you must type a `#\:` before you enter a command name.

The default gesture for both terminating and completing command names is Space. This acts as a delimiter gesture while reading a command name.

The default character for terminating a command argument is Space. This is acts as a delimiter gesture while reading an argument to a command.

17.1.2 Input editor commands

You can edit keyboard input to **`accept`** until you type an activation gesture to terminate the input sentence. After an activation gesture is entered, if CLIM cannot parse the input, you must edit and re-activate it.

The input editor has a number of single-keystroke editing commands, described in the table below. Prefix numeric arguments to input editor commands can be entered using digits and minus sign (-) with control and meta (as in Emacs).

You can use the function **add-input-editor-command** to bind one or more keys to an input editor command. Any character can be an input editor command, but by convention only non-graphic characters should be used.

Table 17.2: Keybindings for the input editor commands

Command	Keybindings
universal-argument	nil; C-u is used by clear-input.
forward-character	C-f
forward-word	M-f ESCAPE f
backward-character	C-b
backward-word	M-b ESCAPE b
beginning-of-buffer	M-< ESCAPE <
end-of-buffer	M->> ESCAPE >
beginning-of-line	C-a
end-of-line	C-e
next-line	C-n
previous-line	C-p
rubout	DELETE or RUBOUT
delete-character	C-d
rubout-word	M-DELETE or M-RUBOUT ESCAPE DELETE or ESCAPE RUBOUT
delete-word	M-d ESCAPE d
kill-line	C-k
make-room	C-o
transpose-characters	C-t
show-arglist	C-x C-a
show-value	C-x C-v
kill-ring-yank	C-y
history-yank	M-C-y ESCAPE C-y

Table 17.2: Keybindings for the input editor commands

Command	Keybindings
yank-next	M-y ESCAPE y
scroll-forward	C-v
scroll-backward	M-v ESCAPE v

The input editor also supports numeric arguments (such as Control-0, Control-1, Meta-0, and so forth) that modify the behavior of the input editing commands. For instance, the motion and deletion commands are repeated as many times as specified by the numeric argument. This accumulated numeric argument is passed to the command processor in such a way that **substitute-numeric-argument-marker** can be used to insert the numeric argument into a command that was read via a keystroke accelerator.

add-input-editor-command [Function]

Arguments: *gestures function*

■ Adds an input editing command that causes *function* to be executed when the specified gesture(s) are typed by the user. *gestures* is either a single gesture name, or a list of gesture names. When *gestures* is a sequence of gesture names, the function is executed only after all of the gestures are typed in order with no intervening gestures. (This is used to implement prefixed commands, such as the Control-X Control-F command one might fix in EMACS.)

17.2 Concepts of CLIM's input editor

CLIM's input editor provides interactive parsing and prompting by interacting with the rest of CLIM's input facility via rescanning.

A CLIM input editing stream encapsulates an interactive stream. That is, most stream operations are handled by the encapsulated interactive stream, but some operations are handled directly by the input editing stream itself.

An input editing stream has the following components:

- The encapsulated interactive stream.
- A buffer with a fill pointer, which we refer to as FP. The buffer contains all of the user's input, and FP is the length of that input.
- An insertion pointer, which we refer to as IP. The insertion pointer is the point in the buffer at which the editing cursor is.
- A scan pointer, which we refer to as SP. The scan pointer is the point in the buffer from which CLIM will get the next input gesture object (in the sense of **read-gesture**).
- A ‘rescan queued’ flag indicating that the programmer (or the input editor itself) requested that a rescan operation should take place before the next gesture is read from the user.
- A ‘rescan in progress’ flag that indicates that CLIM is rescanning the user's input, rather than reading freshly supplied gestures from the user.

The high level description of the operation of the input editor is that it reads either real gestures from the user (such as characters from the keyboard or pointer button events) or input editing commands. The input

editing commands can modify the state of the input buffer. When such modifications take place, it is necessary to rescan the input buffer, that is, reset the scan pointer SP to its original state and reparse the contents of the input editor buffer before reading any other gestures from the user. While this rescanning operation is taking place, the 'rescan in progress' flag is set to `t`. The relationship SP ≤ IP ≤ FP always holds.

17.2.1 Detailed description of the input editor

This section describes the structure of the input editor in a fairly detailed way. If you plan to write complex **`accept`** methods, you may need to understand the input editor at this level of detail. Otherwise, you may skip this section.

The overall control structure of the input editor is:

```
(catch 'rescan         ;thrown to when a rescan is invoked
 (reset-scan-pointer stream)  ;sets STREAM-RESCANNING-P to T
 (loop
   (funcall continuation stream)))
```

where *`stream`* is the input editing stream and *`continuation`* is the code supplied by the programmer, and typically contains calls to such functions as **`accept`** and **`read-token`**. When a rescan operation is invoked, it has the effect of throwing to the `rescan` tag in the example above. The loop is terminated when an activation gesture is seen, and at that point the values produced by *`continuation`* are returned as values from the input editor.

The important point is that functions such as **`accept`**, **`read-gesture`**, and **`unread-gesture`** read (or restore) the next gesture object from the buffer at the position pointed to by the scan pointer SP. However, insertion and input editing commands take place at the position pointed to by IP. The purpose of the rescanning operation is to eventually ensure that all the input gestures issued by the user (typed characters, pointer button presses, and so forth) have been read by CLIM. During input editing, CLIM display an editing cursor to remind you of the position of IP.

The overall structure of **`read-gesture`** on an input editing stream is:

```
(progn
 (rescan-if-necessary stream)
 (loop
  ;; If SP is less than FP
  ;;  Then get the next gesture from the input editor buffer at SP
  ;;  and increment SP
  ;;  Else read the next gesture from the encapsulated stream
  ;;  and insert it into the buffer at IP
  ;; Set the "rescan in progress" flag to false
  ;; Call STREAM-PROCESS-GESTURE on the gesture
  ;;  If it was a "real" gesture
  ;;   Then exit with the gesture as the result
  ;;   Else it was an input editing command (which has already been
  ;;   processed), so continue looping
  ))
```

When a new gesture object is inserted into the input editor buffer, it is inserted at the insertion pointer IP. If IP = FP, this is accomplished by a **`vector-push-extend`**-like operation on the input buffer and FP, and then incrementing IP. If IP < FP, CLIM must first make room for the new gesture in the input buffer, then insert the gesture at IP, then increment both IP and FP.

When the user requests an input editor motion command, only the insertion pointer IP is affected. Motion commands do not need to request a rescan operation.

When the user requests an input editor deletion command, the sequence of gesture objects at IP are removed, and IP and FP must be modified to reflect the new state of the input buffer. Deletion commands (and other commands that modify the input buffer) must arrange for a rescan to occur when they are done modifying the buffer, either by calling **`queue-rescan`** or **`immediate-rescan`**.

CLIM may also insert special objects in the input editor buffer, such as noise strings and accept results. A 'noise string' is used to represent some sort of in-line prompt and is never seen as input; **`input-editor-format`** and **`prompt-for-accept`** methods may insert a noise string into the input buffer. An 'accept result' is an object in the input buffer that is used to represent some object that was inserted into the input buffer (typically via a pointer gesture) that has no readable representation (in the Lisp sense); **`presentation-replace-input`** may create accept results. Noise strings are skipped over by input editing commands, and accept results are treated as a single gesture.

17.3 Functions for doing input editing

`with-input-editing` **[Macro]**

Arguments: `(&optional` *`stream`* `&key initial-contents input-sensitizer class) &body` *`body`*

■ Establishes a context in which the user can edit the input he or she types in on the stream *`stream`*. *`body`* is then executed in this context, and the values returned by *`body`* are returned as the values of **`with-input-editing`**.

The *`stream`* argument is not evaluated, and must be a symbol that is bound to a CLIM input stream. If *`stream`* is `t` (the default), `*standard-input*` is used. If *`stream`* is a stream that is not an interactive stream, then **`with-input-editing`** acts like **`progn`**.

`initial-contents` is a string to become the initial contents of the stream to be edited.

`input-sensitizer`, if it is supplied, is a function of two arguments, a stream and a continuation. The *`input-sensitizer`* function should call the continuation on the stream. For example, the implementation of **`accept`** uses something like the following in order to make the user's input sensitive as a presentation for later use:

```
(flet ((input-sensitizer (continuation stream)
     (if (clim:stream-recording-p stream)
       (clim:with-output-as-presentation (stream object type)
        (funcall continuation stream))
       (funcall continuation stream))))
 (clim:with-input-editing (stream :input-sensitizer #'input-
sensitizer)
  ...))
```

`class` is the name of a stream class to use as the input editing stream; it defaults to CLIM's standard input editing stream class.

■ Usually you will not need to use **`with-input-editing`**, since calls to **`accept`** set up an input editing context for you.

with-input-editor-typeout [Macro]

Arguments: (&optional *stream* &key erase) &body *body*

■ If, when some code is inside of a call to **with-input-editing**, you wish to perform some sort of typeout, it should be done inside **with-input-editor-typeout**. This form collects the output done by *body* to *stream*, clears some space, and then displays the output.

For example, the following fragment could be used to display the argument for a Lisp function while doing input editing on Lisp forms:

```
(clim:with-input-editor-typeout (stream)
 (format stream "~S: (~~A~^ ~)" symbol arglist))
```

If *stream* is not an input editing stream, **with-input-editor-typeout** behaves like **progn**.

In some circumstances, **with-input-editor-typeout** will not clear out the space over which the typeout will be displayed. In that case, you should supply :erase t.

input-editor-format [Function]

Arguments: *stream format-string* &rest *format-args*

■ This function is like **format**, except that it is intended to be called on input editing streams. It arranges to insert a noise string in the input editor's input buffer that represents the output specified by *format-string* and *format-args*. *format-string* and *format-args* are as for **format**.

■ You can use this to display in-line prompts in **accept** methods.

17.4 The input editing protocol

input-editing-stream-p [Function]

Arguments: *object*

■ Returns t if *object* is an input editing stream (that is, a stream of the sort created by a call to **with-input-editing**), otherwise returns nil.

stream-insertion-pointer [Generic function]

Arguments: *stream*

■ Returns an integer corresponding to the current input position in the input editing stream *stream*'s buffer, that is, the point in the buffer at which the next user input gesture will be inserted. The insertion pointer will always be less than (fill-pointer (stream-input-buffer *stream*)). The insertion pointer is used as the location of the editing cursor.

(setf stream-insertion-pointer) [Generic function]

Arguments: *pointer stream*

■ Changes the input position of the input editing stream *stream* to *pointer*. *pointer* is an integer, and must be less than (fill-pointer (stream-input-buffer *stream*)).

stream-scan-pointer [Generic function]

Arguments: *stream*

■ Returns an integer corresponding to the current scan pointer in the input editing stream *stream*'s buffer, that is, the point in the buffer at which calls to **accept** have stopped parsing input. The scan pointer will always be less than or equal to (stream-insertion-pointer *stream*).

(setf stream-scan-pointer) [Generic function]

Arguments: *pointer stream*

■ Changes the scan pointer of the input editing stream *stream* to *pointer*. *pointer* is an integer, and must be less than or equal to (stream-insertion-pointer *stream*).

stream-rescanning-p [Generic function]

Arguments: *stream*

■ Returns the state of the input editing stream *stream*'s 'rescan in progress' flag, which is t if *stream* is performing a rescan operation, otherwise it is nil. Non-input editing streams always return nil.

reset-scan-pointer [Generic function]

Arguments: *stream* &optional (*scan-pointer* 0)

■ Sets the input editing stream *stream*'s scan pointer to *scan-pointer*, and sets the state of **stream-rescanning-p** to t.

immediate-rescan [Generic function]

Arguments: *stream*

■ Invokes a rescan operation immediately by throwing out to the beginning of the most recent invocation of **with-input-editing**.

queue-rescan [Generic function]

Arguments: *stream*

■ Indicates that a rescan operation on the input editing stream *stream* should take place after the next non-input editing gesture is read. This works by setting the 'rescan queued' flag to t.

rescan-if-necessary [Generic function]

Arguments: *stream* &optional *inhibit-activation*

■ Invokes a rescan operation on the input editing stream *stream* if **queue-rescan** was called on the same stream and no intervening rescan operation has taken place. Resets the state of the 'rescan queued' flag to nil.

If *inhibit-activation* is nil, the input line will not be activated even if there is an activation character in it.

erase-input-buffer [Generic function]

Arguments: *stream* &optional (*start-position* 0)

■ Erases the part of the display that corresponds to the input editor's buffer starting at the position *start-position*.

redraw-input-buffer **[Generic function]**

Arguments: *stream* &optional (*start-position* 0)

■ Displays the input editor's buffer starting at the position *start-position* on the interactive stream that is encapsulated by the input editing stream *stream*.

17.5 Examples of extending the input editor

The following is an example of a non-destructive input editing command that displays the current value of a symbol. Every input editing command is passed four required arguments, the input editing stream, its input buffer, the gesture used to invoke the command, and the accumulated numeric argument. This function assumes the existence of a function the locates a symbol at the current input editing position.

```
(defun com-ie-show-value (stream input-buffer gesture numeric-argument)
 (declare (ignore gesture numeric-argument))
 (let* ((symbol (symbol-at-position stream input-buffer))
     (value (and symbol
           (boundp symbol)
           (symbol-value symbol))))
  (if value
    (clim:with-input-editor-typeout (stream)
     (format stream "~S: ~S" symbol value))
    (beep stream))))
```

You could add this command to the set of input editing commands by calling **add-input-editor-command**.

The following could be used to implement the ‘forward character’ command.

```
(defun com-ie-forward-character (stream input-buffer gesture numeric-argument)
 (declare (ignore gesture))
 (let ((ip (clim:stream-insertion-pointer stream))
    (limit (fill-pointer input-buffer)))
  (setq ip (min limit (+ ip numeric-argument)))
  (setf (clim:stream-insertion-pointer stream) ip)))
```

The following could be used to implement the ‘delete character’ command. Note that this example causes a rescan operation to take place; all destructive editing commands must do this.

```
(defun com-ie-delete-character (stream input-buffer gesture numeric-argument)
 (declare (ignore gesture))
 (let* ((p1 (clim:stream-insertion-pointer stream))
     (limit (fill-pointer input-buffer))
     (p2 (min limit (+ p1 numeric-argument))))
  ;; Erase what used to be on the screen
  (clim:erase-input-buffer stream p1)
  ;; Shift the input buffer down over the deleted stuff
  (replace input-buffer input-buffer :start1 p1 :start2 p2)
  (decf (fill-pointer input-buffer) (- p2 p1))
  ;; Make sure the scan pointer doesn’t point past the insertion pointer
  (minf (clim:stream-scan-pointer stream)
        (clim:stream-insertion-pointer stream))
  ;; Redraw the input buffer
```

```
(clim:redraw-input-buffer stream)
;; If the buffer is now empty, rescan immediately so that the state
;; of the input editor gets reinitialized. Otherwise queue a rescan
;; for later
(if (zerop (fill-pointer input-buffer))
  (clim:immediate-rescan stream)
  (clim:queue-rescan stream))))
```

Chapter 18 Output recording in CLIM

18.1 Concepts of CLIM output recording

CLIM provides a mechanism called *output recording* whereby output (textual and graphical) may be captured into an *output history* for later replay on the same stream. This mechanism serves as the basis for many other tools, such as scrolling, formatted output of tables and graphs, for the ability of presentations to retain their semantics, and for incremental redisplay.

The output recording facility is layered on top of the basic graphics and text output facilities. It works by intercepting the operations in the graphics and text output protocols, and saving information about these operations in objects called *output records*. In general, an output record is a kind of display list, that is, a collection of instructions for drawing something on a stream. Some output records may have *children*, that is, a collection of inferior output records. Other output records, which are called *displayed output records*, correspond directly to displayed information on the stream, and do not have children. If you think of output records being arranged in a tree, displayed output records are all of the leaf nodes in the tree, for example, displayed text and graphics records.

Displayed output records record the state of the supplied drawing options at the instant the output record is created. This includes the ink, line style, text style, and clipping region at the time the output record is created, and the coordinates of the output transformed by the user transformation. When you replay an output record later on, the saved information will be used; any new user transformation, clipping region, or line style will not affect the replayed output.

A CLIM stream that supports output recording has an output history object, which is a special kind of output record that supports some other operations. CLIM defines a standard set of output history implementations and a standard set of output record types.

The output recording mechanism is enabled by default on CLIM streams. Unless you turn it off, all output that occurs on a window is captured and saved by the output recording mechanism.

18.1.1 Uses of output recording

One use of an output record is to *replay* it -- to produce the output again. Scrolling is implemented by replaying the appropriate output records.

CLIM's table and graph formatters are implemented on top of output records. For example, when your code uses **`formatting-table`** and formats output into rows and cells, this output is sent to a particular stream. Invisibly to you, CLIM temporarily binds this stream to an intermediate stream, and runs a constraint engine over the code to determine the layout of the table. The result is a set of output records which contain the table, its rows, and its cells. Finally, CLIM replays these output records to your original stream.

When using the techniques of incremental redisplay, your code determines which portions of the display have changed, then the appropriate output records are updated to the new state, and the only the changed output records are replayed.

Presentations are a special case of output records that remember the object and the type of object associated with the output.

18.2 CLIM operators for output recording

The purpose of output recording is to capture the output done by an application onto a stream. The objects used to capture output are called *output records* and *displayed output records*. An output record is an object that stores other output records. Displayed output records are the objects contained in output records that correspond to an atomic piece of output, such as a circle or a piece of text; these are most like traditional display list items. The following classes and predicates correspond to the objects used in output recording.

`output-record` **[Class]**

■ The protocol class that is used to indicate that an object is an *output record*, that is, a CLIM data structure that contains other output records. If you want to create a new class the obeys the output record protocol, it must be a subclass of `output-record`.

If you think of output records being arranged in a tree, output records are the non-leaf nodes of the tree.

`output-record-p` **[Function]**

Arguments: *`object`*

■ Returns `t` if and only *`object`* is of type `output-record`.

`displayed-output-record` **[Class]**

■ The protocol class that is used to indicate that an object is a *displayed output record*, that is, a CLIM data structure that represents a visible piece of output on an output device. If you want to create a new class the obeys the displayed output record protocol, it must be a subclass of `displayed-output-record`.

If you think of output records being arranged in a tree, displayed output records are the leaves of the tree. Displayed text and graphics are examples of things that are displayed output records.

`displayed-output-record-p` **[Function]**

Arguments: *`object`*

■ Returns `t` if and only *`object`* is of type `displayed-output-record`.

The following functions and macros can be used to create and operate on CLIM output records.

`with-new-output-record` **[Macro]**

Arguments: `(`*`stream`* `&optional` *`record-type record`* `&rest` *`initargs`*`) &body` *`body`*

■ Creates a new output record of type *`record-type`* (which defaults to CLIM's default sequence output record) and then captures the output of *`body`* into the new output record. The new record is then inserted into the current open output record associated with *`stream`* (that is, `(clim:stream-current-output-record` *`stream`*`)`.

If *`record`* is supplied, it is the name of a variable that will be lexically bound to the new output record inside of *`body`*. *`initargs`* are CLOS initargs that are passed to **`make-instance`** when the new output record is created.

`with-new-output-record` returns the output record it creates.

invoke-with-new-output-record [Generic function]

Arguments: *stream function record-type constructor* &rest *initargs*

■ This is the functional version of **with-new-output-record**. *stream* and *record-type* are the same as for **with-new-output-record**.

constructor is a constructor function that creates the new output record. Since it is for CLIM's internal use, you should generally supply `nil` for this.

function is a function of one argument, a stream; it is called to generate the output to be inserted into the newly created output record.

with-output-to-output-record [Macro]

Arguments: (*stream* &optional *record-type record* &rest *initargs*) &body *body*

■ This is similar to **with-new-output-record** except that the new output record is not inserted into the output record hierarchy. That is, when you use **with-output-to-output-record**, no drawing on the stream occurs and nothing is put into the stream's normal output history. Unlike in facilities such as **with-output-to-string**, *stream* must be an actual stream, but no output will be done to it.

record-type is the type of output record to create, which defaults to CLIM's default sequence output record type. *initargs* are CLOS initargs that are used to initialize the record.

If *record* is supplied, it is a variable which will be bound to the new output record while *body* is evaluated.

■ The new output record is returned.

with-output-recording-options [Macro]

Arguments: (*stream* &key draw record) &body *body*

■ Used to disable output recording and/or drawing on the given *stream*, within the extent of *body*.

If *draw* is `nil`, output to the stream is not drawn on the viewport, but can still be recorded in the output history. If *record* is `nil`, output recording is disabled but output otherwise proceeds normally.

replay [Function]

Arguments: *record stream* &optional *region*

■ Replays all of the output captured by the output record *record* on *stream* by calling **replay-output-record**. If *region* is supplied and is a region, then *record* is replayed if and only if it overlaps region. *region* defaults to the viewport of *stream*, or to *stream*'s entire region if it has no viewport.

Changing the transformation of the stream during replaying has no effect on what is output by **replay**.

replay-output-record [Generic function]

Arguments: *record stream* &optional *region x-offset y-offset*

■ Replays all of the output captured by the output record *record* on *stream*. If *region* is not `nil`, then *record* is replayed if and only if it overlaps *region*.

x-offset and *y-offset* are output record offsets that are necessitated by CLIM's representation of output records. In a later release of CLIM, the representation of output records may change in such a way that the *x-offset* and *y-offset* arguments are removed.

Changing the transformation of the stream during replaying has no effect on what is output by **replay-output-record**.

You can specialize this generic function for your own classes of output records. If you write your own output record class that is a subclass of displayed-output-record, you must implement or inherit a method for this generic function for that class.

18.2.1 Examples of creating and replaying output records

CLIM's **indenting-output** facility could have been implemented in the following way. First, drawing is disabled, then the user's output is collected, then indented, and finally replayed.

```
(defmacro indenting-output ((stream indentation) &body body)
 `(let ((record
      (clim:with-output-recording-options (,stream :draw nil :record t)
       (clim:with-new-output-record (,stream)
        ,@body))))
  (multiple-value-bind (x y) (clim:output-record-position record)
   (clim:output-record-set-position record (+ x ,indentation) y))
  (clim:tree-recompute-extent record)
  (clim:replay record ,stream)
  record))
```

The following could be used to measure the size of some output without actually doing the output.

```
(defmacro compute-output-size ((stream) &body body)
 `(let ((record
      (clim:with-output-to-output-record (,stream)
       ,@body)))
  (clim:bounding-rectangle-size record)))
```

18.2.2 Output record database functions

The following functions implement the 'database' protocol of output records. If you implement your own output record that is a subclass of **output-record**, you must implement or inherit methods for these functions.

output-record-parent [Generic function]

Arguments: *record*

- Returns an output record that is the parent of the output record *record*. If *record* has no parent, this will return nil.

output-record-children [Generic function]

Arguments: *record*

- Returns a sequence of all of the children of the output record *record*.

  For some classes of output record, this function can be very inefficient because the class does not store the children in the form of a sequence. It is often better to use **map-over-output-records**.

output-record-count [Generic function]

Arguments: *record*

■ Returns the number of children contained within the output record *record*.

add-output-record [Generic function]

Arguments: *child record*

■ Adds the output record *child* to the output record *record*. It also sets the parent of *child* to be *record*.

delete-output-record [Generic function]

Arguments: *child record* &optional *errorp*

■ Removes the output record *child* from the output record *record*. If *child* is not contained in *record* and *errorp* is `t`, an error is signaled.

Note that calling **delete-output-record** to delete a child from some output record *record* while inside of a call to any of the mapping functions on the same output record *record* will not work as expected. You should use the mapping function to collect all the records to be deleted, then call **delete-output-record** to delete the set of output records.

erase-output-record [Generic function]

Arguments: *record stream* &optional (*errorp* `t`)

■ Erases the output record *record* from *stream*, and removes the record from *stream*'s output history. After the record is erased, all of the output records that overlapped it are replayed in order to ensure that the appearance of the rest of the output on *stream* is correct.

errorp is as for **delete-output-record**.

record can be a list of output records rather than a single output record. In that case, the replay operation will be delayed until after all of the output records have been removed from the output history. Passing a list of output records to **erase-output-record** can be substantially faster than calling **erase-output-record** multiple times.

clear-output-record [Generic function]

Arguments: *record*

■ Removes all of the output records from the output record *record*.

The following functions can be used to apply a function to all of the children of an output record.

map-over-output-records [Function]

Arguments: *continuation record* &optional (*x-offset* `0`) (*y-offset* `0`) &rest *continuation-args*

■ Applies the function *continuation* to all of the output records contained in the output record *record*. Normally, *continuation* is called with a single argument, an output record. If *continuation-args* is supplied, they are passed to *continuation* as well.

x-offset and *y-offset* are output record offsets that are necessitated by CLIM's representation of output records.

`map-over-output-records-containing-position` [Generic function]

Arguments: *continuation record x y* &optional *x-offset y-offset* &rest *continuation-args*

■ Applies the function *continuation* to all of the output records contained in the output record *record* that overlap the point (*x*,*y*). Normally, *continuation* is called with a single argument, an output record. If *continuation-args* is supplied, they are passed to *continuation* as well.

x-offset and *y-offset* are output record offsets that are necessitated by CLIM's representation of output records.

When **`map-over-output-records-containing-position`** maps over the children in the record, it does so in such a way that, when it maps over overlapping records, the bottom-most (least recently inserted) record is hit last. This is because this function is used for things like locating the presentation under the pointer, where the topmost record should be the one that is found.

`map-over-output-records-overlapping-region` [Generic function]

Arguments: *continuation record region* &optional *x-offset y-offset* &rest *continuation-args*

■ Applies the function *continuation* to all of the output records contained in the output record *record* that overlap the region *region*. Normally, *continuation* is called with a single argument, an output record. If *continuation-args* is supplied, they are passed to *continuation* as well.

x-offset and *y-offset* are output record offsets that are necessitated by CLIM's representation of output records.

When **`map-over-output-records-overlapping-region`** maps over the children in the record, it does so in such a way that, when it maps over overlapping records, the topmost (most recently inserted) record is hit last. This is because this function is used for things such as replaying, where the most recently drawn thing must come out on top (that is, must be drawn last).

18.2.3 Output record change notification protocol

The following functions are called by programmers and by CLIM itself in order to notify a parent output record when the bounding rectangle of one of its child output record changes. You will need to use these if you implement your own formatting engine, for example, a new type of table or graph formatting.

`recompute-extent-for-new-child` [Generic function]

Arguments: *record child*

■ This function is called whenever a new child is added to an output record. Its contract is to update the bounding rectangle of the output record *record* to be large enough to completely contain the new child output record *child*. The parent of *record* will be notified by calling **`recompute-extent-for-changed-child`**.

An `:after` method on **`add-output-record`** calls **`recompute-extent-for-new-child`**, so you will rarely need to call it yourself.

recompute-extent-for-changed-child [Generic function]

Arguments: *record child old-min-x old-min-y old-max-x old-max-y*

■ This function is called whenever the bounding rectangle of one of the children of a record has been changed. Its contract is to update the bounding rectangle of the output record *record* to be large enough to completely contain the new bounding rectangle of the child output record *child*. All of the ancestors of *record* are notified by recursively calling **recompute-extent-for-changed-child**.

An `:after` method on **delete-output-record** calls **recompute-extent-for-changed-child** to inform the parent of the record that a change has taken place, so you will rarely need to call this yourself.

tree-recompute-extent [Generic function]

Arguments: *record*

■ This function is called whenever the bounding rectangles of a number of children of a record have been changed, such as happens during table and graph formatting. Its contract is to compute the bounding rectangle large enough to contain all of the children of the output record *record*, adjust the bounding rectangle of the output record *record* accordingly, and then call **recompute-extent-for-changed-child** on *record*.

Whenever you write a new formatting facility that rearranges the descendents of an output record (for example, a new kind of graph formatting), you should call **tree-recompute-extent** on the parent of the highest level record that was affected.

18.2.4 Operations on output recording streams

The following functions can be used to operate on output recording streams.

output-recording-stream-p [Function]

Arguments: *object*

■ Returns `t` if an only if *object* is an output recording stream.

stream-output-history [Generic function]

Arguments: *stream*

■ Returns the top level output record for the stream *stream*. You cannot use **setf** to set the stream output history of a pane. Instead, you should create the pane with the `:output-record` initarg specified, as in the following example:

```
(define-application-frame test ()
  ()
  (:panes
   (display :application
            :output-record (make-instance 'standard-tree-output-history)))
  (:layouts
   (:default
     display)))
```

stream-current-output-record [Generic function]

Arguments: *stream*

■ The current open output record for the output recording stream *stream*, the one to which **stream-add-output-record** will add a new child record. Initially, this is the same as **stream-output-history**. As applications created nested output records, this acts as a stack.

stream-replay [Generic function]

Arguments: *stream* &optional *region*

■ Replays all of the output records in *stream*'s output history that overlap the region *region*. If *region* is supplied and is a region, then *record* is replayed if and only if it overlaps region. *region* defaults to the viewport of *stream*, or to *stream*'s entire region if it has no viewport.

stream-drawing-p [Generic function]

Arguments: *stream*

■ Returns `t` if and only if drawing is enabled on the output recording stream *stream*. You can use **setf** on this to enable or disabled drawing on the stream, or you can use the `:draw` option to **with-output-recording-options**.

stream-recording-p [Generic function]

Arguments: *stream*

■ Returns `t` if and only if output recording is enabled on the output recording stream *stream*. You can use **setf** on this to enable or disabled output recording on the stream, or you can use the `:record` option to **with-output-recording-options**.

copy-textual-output-history [Function]

Arguments: *window stream* &optional *region record*

■ Given a window *window* that supports output recording, this function finds all of the textual output records that overlap the region *region* (or all of the textual output records if *region* is not supplied), and outputs that text to *stream*. This can be used when you want to capture all of the text on a window into a disk file for later perusal.

stream-add-output-record [Generic function]

Arguments: *stream record*

■ Adds the output record *record* to the *stream*'s current output record (that is, **stream-current-output-record**).

18.3 Standard output record classes

The following output record classes are supported by CLIM. The history classes should be used for top-level output records.

standard-sequence-output-record [Class]

■ The standard instantiable class provided by CLIM to store a relatively short sequence of output records; a subclass of `output-record`. The insertion and retrieval complexity of this class is O(n). Most of the formatted output facilities (such as **formatting-table**) create output records that are a subclass of `standard-sequence-output-record`.

`standard-sequence-output-history` **[Class]**

■ An instantiable class provided by CLIM to use for top-level output records that have only a small number of children. This is a subclass of `standard-sequence-output-record`.

■ In release 2.0 final, this is the default class used by CLIM streams for output history.

`standard-tree-output-record` **[Class]**

■ The standard instantiable class provided by CLIM to store longer sequences of output records. This class is optimized to store output records that tend to be created one after another with ascending y coordinates, such as you would find in a scrolling window of text. The insertion and retrieval complexity of this class is roughly O(log n), but can break down to be O(n).

`standard-tree-output-history` **[Class]**

■ The standard instantiable class provided by CLIM to use as the top-level output history. This is a subclass of `standard-tree-output-record`.

■ In releases prior to CLIM 2.0 final (the various alpha and beta releases), this class was used by default by CLIM streams for their output history. However, this class has known problems when the output records overlap. If you have non-overlapping records, using this class will make things run faster. However, if your output records overlap, using this class may not work properly.

`r-tree-output-history` **[Class]**

■ An instantiable class provided by CLIM to use as the top-level output history for highly overlapping graphical output. Although its insertion and retrieval complexity is O(log n), the overhead of the class is high.

You should use this only for such applications as graphical editors that will maintain a fairly large number of objects.

[This page intentionally left blank.]

Chapter 19 Streams and windows in CLIM

CLIM performs all of its input and output operations on objects called streams. A stream in CLIM is a sheet with a medium that supports CLIM's stream protocol. The stream protocols are partitioned into two layers: the basic stream protocol, and the extended stream protocol.

The basic stream protocol is character-based and compatible with existing Common Lisp programs. (Note that the basic stream protocol is not documented in this user guide, but is documented as part of Common Lisp). The standard Common Lisp stream functions work on CLIM streams in all CLIM implementations.

You can use the extended stream protocol to include pointer events and synchronous window-manager communication.

19.1 Extended stream input in CLIM

CLIM defines an extended input stream protocol. This protocol extends the basic input stream model to allow manipulation of non-character user gestures, such as pointer button presses.

19.1.1 Operators for extended stream input

`extended-input-stream-p` **[Generic function]**

Arguments: *`object`*

■ Returns `t` if the *`object`* is a CLIM extended input stream, otherwise returns `nil`. This function will always return `t` when given a CLIM stream pane.

`read-gesture` **[Function]**

Arguments: `&key (stream *standard-input*) timeout peek-p input-wait-test input-wait-handler pointer-button-press-handler`

■ Returns the next gesture available in the input stream, which will be either a character or an event object, such as a pointer button event. Note that **`read-gesture`** does not echo character input.

When the user types any sort of abort gesture (that is, a character that matches any of the gesture names in `*abort-gestures*`), the `abort-gesture` condition is signaled.

If the user types an accelerator gesture (that is, a gesture that matches any of the gesture names in `*accelerator-gestures*`), then the `accelerator-gesture` condition is signaled.

timeout

Specifies the number of seconds that **read-gesture** will wait for input to become available, or `nil` meaning that there is no timeout. If no input is available when the timeout expires, **read-gesture** will return the two values `nil` and `:timeout`.

peek-p

If `t`, specifies that the gesture returned will be left in the stream's input buffer.

input-wait-test

The value of this argument is a function. The function will be invoked with one argument, the stream. The function should return `t` when there is input to process, otherwise it should return `nil`. This argument will be passed on to **stream-input-wait**.

input-wait-handler

The value of this argument is a function. The function will be invoked with one argument, the stream, when the invocation of **stream-input-wait** returns, but no input gesture is available. This option can be used in conjunction with *input-wait-test* to handle conditions other than user keystroke gestures.

pointer-button-press-handler

is a function of two arguments, the stream and a pointer button press event. It is called when the user clicks a pointer button.

■ Most programmers will never need to use `:input-wait-test`, `:input-wait-handler`, and `:pointer-button-press-handler`.

unread-gesture [Function]

Arguments: *gesture* &key (stream *standard-output*)

■ Places the specified *gesture* back into *stream*'s input buffer. The next **read-gesture** request will return the unread gesture. The gesture supplied must be the most recent gesture read from the stream.

stream-input-wait [Generic function]

Arguments: *stream &key timeout input-wait-test*

■ Waits until *timeout* has expired or *input-wait-test* returns a non-`nil` value. Otherwise the function waits until there is input in the *stream*.

timeout

Specifies the number of seconds that **stream-input-wait** will wait for input to become available. If no input is available, **stream-input-wait** will return two values, `nil` and `:timeout`. If `nil` (the default), it will wait indefinitely.

input-wait-test

The value of this argument is a function. The function will be invoked with one argument, the stream. If the function returns `nil`, **stream-input-wait** will continue to wait for user input. If it returns `t`, **stream-input-wait** will return two values, `nil` and `:input-wait-test`.

abort-gestures [Variable]

■ A list of gestures that causes the current input to be aborted.

abort-gesture [Condition]

■ The **abort-gesture** condition is signaled whenever CLIM reads an abort gesture from the user. For example, on Allegro, **read-gesture** will signal this condition if the user presses Control-Z.

abort-gesture-event [Generic function]

Arguments: *abort-gesture*

■ Returns the event object that caused the abort gesture condition, *abort-gesture*, to be signaled.

accelerator-gestures [Variable]

■ A list of the currently active keystroke accelerator gestures.

accelerator-gesture [Class]

■ The accelerator-gesture condition is signaled whenever CLIM reads an accelerator gesture from the user.

accelerator-gesture-event [Generic function]

Arguments: *accelerator-gesture*

■ Returns the event object that caused the accelerator gesture condition, *accelerator-gesture*, to be signaled.

accelerator-gesture-numeric-argument [Generic function]

Arguments: *accelerator-gesture*

■ Returns the numeric argument associated with the accelerator gesture condition, *accelerator-gesture*. If the user did not supply a numeric argument explicitly, this will return 1.

19.2 Extended stream output in CLIM

In addition to the basic output stream protocol, CLIM defines an extended output stream protocol. This protocol extends the stream model to allow the manipulation of a text cursor.

extended-output-stream-p [Generic function]

Arguments: *object*

■ Returns t if the *object* is a CLIM extended output stream, otherwise returns nil. This function will always return t when given a CLIM stream pane.

19.3 Manipulating the cursor in CLIM

CLIM extends the output stream model to allow the manipulation of a text cursor. A CLIM stream has a text cursor position, which is the place on the drawing plane where the next piece of text output will be drawn. Common Lisp stream output operations place text at the cursor position and advances the cursor position past the text. Certain CLIM output operations, such as **present** and **formatting-table**, do the same. CLIM's graphical drawing function, such as **draw-line*** and **draw-text*** functions, on the other hand, pay no attention to the text cursor position. You can use **with-room-for-graphics**, which does graphical output at the current text cursor position, to tie text and graphics together

Common Lisp stream input operations that echo, such as **read-line**, as well as **accept**, echo the input at the cursor position, and advance the cursor position as the user types characters on the keyboard.

19.3.1 Operators for manipulating the cursor

stream-cursor-position **[Generic function]**

Arguments: *stream*

■ Returns two values, the x and y coordinates of the cursor position on *stream*'s drawing plane. You can use **stream-set-cursor-position** or **stream-increment-cursor-position** to change the cursor position.

stream-set-cursor-position **[Generic function]**

Arguments: *stream x y*

■ Moves the cursor position to the specified x and y coordinates on *stream*'s drawing plane.

stream-increment-cursor-position **[Generic function]**

Arguments: *stream dx dy*

■ Moves the cursor position relative to its current position, adding *dx* to the x coordinate and adding *dy* to the y coordinate. Either argument *dx* or *dy* can be `nil`, which means that CLIM will not change that coordinate.

stream-text-cursor **[Generic function]**

Arguments: *stream*

■ Returns the text cursor object for the stream *stream*.

cursor-position **[Generic function]**

Arguments: *cursor*

■ Returns the cursor position of *cursor*, relative to the sheet on which the cursor is located.

cursor-set-position **[Generic function]**

Arguments: *cursor x y*

■ Sets the cursor position of *cursor* to *x* and *y*, which are relative to the sheet on which the cursor is located.

cursor-sheet **[Generic function]**

Arguments: *cursor*

■ Returns the sheet on which *cursor* is located.

cursor-active **[Generic function]**

Arguments: *cursor*

■ Returns the active attribute of the cursor. An *active cursor* is one that is being actively maintained by its owning sheet. When `t`, the cursor is active.

You can use **setf** on this to change the active attribute of the cursor.

cursor-state **[Generic function]**

Arguments: *cursor*

■ Returns the state attribute of the cursor. An active cursor has a *state* that is either on or off. When `t`, the cursor is visible. When `nil`, the cursor is not visible.

You can use **setf** on this to change the state attribute of the cursor.

cursor-focus [Generic function]

Arguments: *cursor*

■ Returns the focus attribute of the cursor. An active cursor has a state that indicates the owning sheet has the input focus. When `t`, the sheet owning the cursor has the input focus.

You can use **setf** on this to change the focus attribute of the cursor.

cursor-visibility [Generic function]

Arguments: *cursor*

■ This are convenience functions that combine the functionality of **cursor-active** and **cursor-state**. The visibility can be either `:on` (meaning that the cursor is active and visible at its current position), `:off` (meaning that the cursor is active, but not visible at its current position), or `nil` (meaning that the cursor is not activate).

You can use **setf** on this to change the visibility of the cursor.

19.3.2 Text measurement operations in CLIM

These functions compute the change in the cursor position that would occur if some text were output (that is, without actually doing any output and without changing the cursor position).

stream-character-width [Generic function]

Arguments: *stream character* &optional *text-style*

■ The horizontal motion of the cursor position that would occur if this *character* were output onto *stream* in the text style *text-style*. Note that this ignores the text margin of the stream (**stream-text-margin**).

text-style defaults to the *stream*'s current text style. The result depends on the current cursor position when the *character* is Newline or Tab.

stream-line-height [Generic function]

Arguments: *stream &optional text-style*

■ Returns what the line height of a line containing text in that *text-style* would be. *text-style* defaults to the *stream*'s current text style.

stream-vertical-spacing [Generic function]

Arguments: *stream*

■ Returns the amount of vertical space between consecutive lines on *stream*.

stream-baseline [Generic function]

Arguments: *stream*

■ Returns the current text baseline on *stream*.

stream-text-margin [Generic function]

Arguments: *stream*

■ The x coordinate at which text wraps around (see **stream-end-of-line-action**). The default setting is the width of the viewport, which is the right-hand edge of the viewport when it is horizontally scrolled to the initial position.

You can use **setf** with **stream-text-margin**. If a value of `nil` is specified, the width of the viewport will be used. If the width of the viewport is later changed, the text margin will change too.

text-size [Generic function]

Arguments: *medium string* &key text-style start end

■ Computes how the cursor position would move if the specified *string* or character were output to *medium* starting at cursor position (0,0). Note that, when called on a CLIM stream, this ignores the stream's text margin, that us, all output is done on a single line (except, of course, when newlines are explicitly included in the output).

■ **text-size** returns five values:

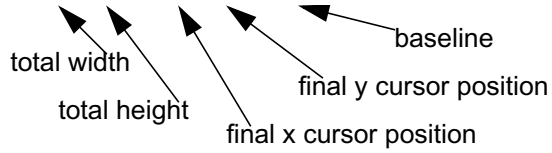
- the total width of the string in device units
- the total height of the string in device units
- the final X cursor position, which is the same as the width if there are no Newline characters in the string
- the final Y cursor position, which is 0 if the string has no Newline characters in it, and is incremented by the line height for each Newline character in the string
- the string's baseline

text-style defaults to the medium's current text style. *start* and *end* default to 0 and the length of the string, respectively.

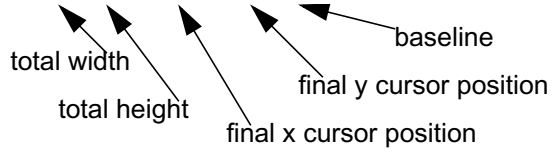
■ Note that **text-size** is a fairly low level function. If you find you need to use it often, you may be working at too low a level of abstraction.

Here are a few examples of use of **text-size**.

```
(clim:text-size *standard-output* (format nil "Hi there"))
 → 64  12  64  0  10
```




```
(clim:text-size *standard-output* (format nil "Hi~%there"))
 → 40  24  40  12  10
```



That is, the first example is a maximum of 64 device units wide, 12 units high, the final X cursor position is 64, the final Y cursor position is 0 (that is, the text is only one line high so the Y cursor did not advance), and the baseline is 10. In the second example, the maximum width is 40, the height is 24, the final X cursor position in 40, the final Y cursor position is 12, and the baseline is 10.

`stream-string-width` [Generic function]

Arguments: *stream string* &key start end text-style

■ Computes how the cursor position would move horizontally if the specified *string* were output starting at the left margin. Note that this ignores the stream's text margin.

The first value is the x coordinate the cursor position would move to. The second value is the maximum x coordinate the cursor would visit during the output. (This is the same as the first value unless the string contains a Newline.)

start and *end* default to 0 and the length of the string, respectively.

text-style defaults to the stream's current text style.

■ Note that **`stream-string-width`** is a fairly low level function. If you find you need to use it often, you may be working at too low a level of abstraction.

19.4 Attracting attention, selecting a file, noting progress

Attracting attention

CLIM supports the following operators for attracting the user's attention:

`beep` [Function]

Arguments: &optional (*stream* *standard-output*)

■ Attracts the user's attention, usually with an audible sound.

`notify-user` [Generic function]

Arguments: *frame message* &rest *options*

■ Notifies the user of some event on behalf of the application frame *frame*. *message* is a message string. The possible *options* are:

Option	Value
`:associated-window`	(as for **`menu-choose`**)
`:title`	a string used to label the notification; default is "Notify User".
`:documentation`	documentation string, default `nil`.
`:exit-boxes`	(as for **`accepting-values`**)
`:name`	a string that names the widget; default, the value of `:title`.
`:text-style`	(as for **`menu-choose`**)
`:style`	(see just below)

■ `:style` can be:

`:inform`	Tell the user something (the default)
`:error`	Tell the user about an error
`:question`	Question the user (e.g. 'Delete this file y-or-n?')
`:warning`	Warn the user

Selecting a file

`select-file` [Generic function]

Arguments: *`frame`* `&rest` *`options`*

■ Pops up a dialog to select a file and returns the name of the file selected. The possible *`options`* are:

Option	Value
`:associated-window`	(as for **`menu-choose`**)
`:title`	a string used to label the notification; default is "Select File".
`:documentation`	documentation string, default `nil`.
`:exit-boxes`	(as for **`menu-choose`**)
`:name`	a string that names the widget; default `:select-file`.
`:directory-list-label`	a string to label the list of directories.
`:file-list-label`	a string to label the list of files.
`:directory`	a string specifying the initial directory used.
`:pattern`	a string defining the file search pattern.
`:text-style`	(as for **`menu-choose`**)

Noting progress

The following functionality is provided for noting progress:

`*current-progress-note*` [Variable]

■ This variable is used as the default progress note in all of the following functions.

`noting-progress` [Macro]

Arguments: *`(stream name`* `&optional` *`note-var)`* `&body` *`body`*

■ Establishes a progress noting context in which *`body`* is executed. *`note-var`* is a variable to which a progress note is bound during the execution of *`body`*. *`note-var`* defaults to `*current-progress-note*`. *`name`* is the name given to the progress note and should be a string. *`stream`* should be a CLIM stream pane used as the associated window for any progress notification windows.

■ Within *`body`*, calls to **`note-progress`** will display progress notification

dotimes-noting-progress **[Macro]**

Arguments: *(var countform* &optional *stream note-var)* &body *body*

■ This is like the Common Lisp form **dotimes** but in addition progress notes are automatically generated each time through the loop. *var* and *countform* are as in **dotimes**. *stream* and *note-var* are as in **noting-progress**. **note-progress** is automatically called each time through the loop but additional calls can be made within *body*.

dolist-noting-progress **[Macro]**

Arguments: *(var listform* &optional *stream note-var)* &body *body*

■ This is like the Common Lisp form **dolist** but in addition progress notes are automatically generated each time through the loop. *var* and *listform* are as in **dolist**. *stream* and *note-var* are as in **noting-progress**. **note-progress** is automatically called each time through the loop but additional calls can be made within **body**.

note-progress **[Function]**

Arguments: *numerator* &optional *(denominator 1) note*

■ This displays progress notification for progress note *note*. *note* defaults to *current-progress-note*. This function is typically called within one of the above 3 macros.

19.5 Window stream operations in CLIM

The following functions can be called on any CLIM stream pane (that is, any pane that is a subclass of clim-stream-pane). Such a pane is often simply referred to as a 'window'. These are provided purely as a convenience for programmers.

19.5.1 Clearing and refreshing the drawing plane in CLIM

CLIM supports the following operators for clearing and refreshing the drawing plane:

window-clear **[Generic function]**

Arguments: *window*

■ Clears the entire drawing plane of *window*, filling it with the background design. **window-clear** also discards the window's output history and resets the cursor position to the upper left corner.

window-erase-viewport **[Generic function]**

Arguments: *window*

■ Clears the visible part of the drawing plane of *window*, filling it with the background design.

window-refresh **[Generic function]**

Arguments: *window*

■ Clears the visible part of the drawing plane of *window*, and then replays all of the output records in the visible part of the drawing plane.

19.5.2 The viewport and scrolling in CLIM

A window stream's viewport is the region of the drawing plane that is visible through the window. You can change the viewport by scrolling or by reshaping the window. The viewport does not change if the window is covered by another window (that is, the viewport is the region of the drawing plane that would be visible if the window were stacked on top).

A window stream has an end of line action and an end of page action, which control what happens when the cursor position moves out of the viewport (**`with-end-of-line-action`** and **`with-end-of-page-action`**, respectively).

`window-viewport` [Generic function]

Arguments: *`window`*

■ Returns the *`window`*'s current viewport, usually an object of type `standard-bounding-rectangle`.

`window-viewport-position` [Generic function]

Arguments: *`window`*

■ Returns two values, the x and y coordinates of the top-left corner of the *`window`*'s viewport.

`note-viewport-position-changed` [Generic function]

Arguments: *`frame sheet x y`*

■ This notification function is called whenever the position of the viewport associated with *`sheet`* changes. Methods can be defined for particular frame and sheet classes. *`x`* and *`y`* are the new position, as returned by **`window-viewport-position`**.

`window-set-viewport-position` [Generic function]

Arguments: *`window x y`*

■ Moves the upper left corner of the *`window`*'s viewport. This is the simplest way to scroll a window. The function **`scroll-extent`** does much the same thing.

`stream-end-of-line-action` [Generic function]

Arguments: *`stream`*

■ Controls what happens when the cursor position moves horizontally out of the *`stream`*'s viewport (beyond the text margin). You can use this function with **`setf`** to change the end of line action

You can use **`with-end-of-line-action`** to temporarily change the end of line action.

`stream-end-of-page-action` [Generic function]

Arguments: *`stream`*

■ Controls what happens when the cursor position moves vertically out of the *`stream`*'s viewport. You can use this function with **`setf`** to change the end of page action

You can use **`with-end-of-page-action`** to temporarily change the end of page action.

with-end-of-line-action [Macro]

Arguments: (*stream action*) &body *body*

■ Temporarily changes the end of line action for *stream* for the duration of execution of body. The end of line action controls what happens if the cursor position moves horizontally out of the viewport, or if text output reaches the **text-margin**. (By default, the text margin is the width of the viewport, so these are the same thing.)

■ The end of line action is one of:

`:wrap`

When doing text output, wrap the text around (that is, break the text line and start another line) when the output reaches the text margin. When setting the cursor position, scroll the window horizontally to keep the cursor position inside the viewport. This is the default.

`:scroll`

Scroll the window horizontally to keep the cursor position inside the viewport, then keep doing output.

`:allow`

Ignore the text margin and just keep doing output.

with-end-of-page-action [Macro]

Arguments: (*stream action*) &body *body*

■ Temporarily changes the end of page action for *stream* for the duration of execution of body. The end of page action controls what happens if the cursor moves vertically out of the viewport.

■ The end of page action is one of:

`:scroll`

Scroll the window vertically to keep the cursor position inside the viewport, then keep doing output. This is the default.

`:allow`

Ignore the viewport and just keep doing output.

window-parent [Generic function]

Arguments: *window*

■ Returns the window that is the parent (superior) of *window*. This is identical to **sheet-parent**, and is included for compatibility with CLIM 1.1.

window-children [Generic function]

Arguments: *window*

■ Returns a list of all of the windows that are children (inferiors) of *window*. This is identical to **sheet-children**, and is included for compatibility with CLIM 1.1.

stream-set-input-focus [Function]

Arguments: *stream*

■ Selects *stream* as the sheet with the current input focus, and returns as a value the sheet previously holding the focus.

with-input-focus **[Macro]**

Arguments: *(stream)* &body *body*

■ Temporarily gives the keyboard input focus to the given window (which is most often an interactor pane). By default, a frame will give the input focus to the `frame-query-io` pane.

The following functions are most usefully applied to the top level window of a frame. For example,

```
(clim:frame-top-level-sheet clim:*application-frame*)
```

window-expose **[Generic function]**

Arguments: *window*

■ Makes the *window* visible on the display server.

window-stack-on-bottom **[Generic function]**

Arguments: *window*

■ Puts the *window* underneath all other windows that it overlaps.

window-stack-on-top **[Generic function]**

Arguments: *window*

■ Puts the *window* on top of all other windows that it overlaps, so you can see all of it.

window-visibility **[Generic function]**

Arguments: *stream*

■ A predicate that returns true if the *window* is visible. You can use **setf** on **window-visibility** to expose or de-expose the window.

The following operators can be applied to a window to determine its position and size.

window-inside-edges **[Generic function]**

Arguments: *window*

■ Returns four values, the coordinates of the left, top, right, and bottom inside edges of the window *window*. The inside edges are, in effect, the edges within which all output takes places.

window-inside-size **[Generic function]**

Arguments: *window*

■ Returns the inside width and height of *window* as two values.

19.5.3 Operators for creating CLIM window streams

find-port **[Function]**

Arguments: &rest *initargs* &key (server-path *default-server-path*) &allow-other-keys

■ Creates a port, a special object that acts as the root or parent of all CLIM windows and application frames. In general, a port corresponds to a connection to a display server.

server-path is a list that specifies the server path. The first element of the list is the keyword `:motif`. Then the keyword `:display`, whose value is an X display name and, usually, number that identifies the X server to be used (as in the example just below).

■ Note: You should call **`find-port`** only at runtime, not at load time. This function captures information about the screen currently in use, which will not be valid across boot sessions.

■ The usual idiom for creating a port on your own machine is `(clim:find-port)`. If you are using some sort of display server, you may need to do something more complex, such as

```
(clim:find-port :server-path '(:motif :display "vapor:0"))
```

`find-frame-manager` [Generic function]

Arguments: `&rest` *`options`* `&key port (server-path *default-server-path*) &allow-other-keys`

■ Finds a frame manager that is on the port *`port`*, or creates a new one if none exists. If port is not supplied and a new port must be created, server-path may be supplied for use by **`find-port`**.

`options` may include other initargs for the frame manager.

■ You will only rarely need to create a frame manager explicitly. Usually, you should just call **`frame-manager`** on a frame object, such as

```
(clim:frame-manager (clim:pane-frame stream)).
```

`open-window-stream` [Function]

Arguments: `&key parent left top right bottom width height (foreground clim:+black+) (background clim:+white+) text-style default-text-style (vertical-spacing 2) (end-of-line-action :allow) (end-of-page-action :allow) output-record (draw t) (record t) (initial-cursor-visibility :off) text-margin default-text-margin save-under input-buffer (scroll-bars :vertical) borders label`

■ A convenient composite function for creating a standalone CLIM window.

This function is not often used. Most often you will use windows that are created by an application frame or by the menu and dialog functions.

Note that some of these keyword arguments are also available as pane options in **`define-application-frame`**.

`parent`

The parent of the window. Its value can be a frame-manager or an application frame. It defaults to what is returned by `(find-frame-manager)`.

`left`

`top`

`right`

`bottom`

`width`

`height`

Used to specify the position and shape of the window within its parent, in device units. The default is to fill the entire parent.

`borders`

Controls whether borders are drawn around the window (`t` or `nil`). The default is `t`.

default-text-margin
Text margin to use if **stream-text-margin** isn't set. This defaults to the width of the viewport.

text-style
draw
record
end-of-line-action
end-of-page-action
background
foreground
text-margin
Initial values for the corresponding stream attributes.

initial-cursor-visibility
`:off` means make the cursor visible if the window is waiting for input. `:on` means make it visible now. The default is `nil` which means the cursor is never visible.

label
`nil` or a string label for the window. The default is `nil` for no label.

output-record
Specify this if you want a different output history mechanism than the default.

scroll-bars
One of `nil`, `:vertical`, `:horizontal`, or `:both`. Adds scroll bars to the window. The default is `:both`.

vertical-spacing
Amount of extra space between text lines, in device units.

- The remaining keyword arguments are internal and should not be used.

Chapter 20 The Silica windowing substrate

This chapter describes details of the low-level implementation of CLIM. Application writers and users do not typically make use of the functionality described in this chapter.

20.1 Overview of CLIM's windowing substrate

A central notion in organizing user interfaces is allocating screen regions to particular tasks and recursively subdividing these regions into subregions. The windowing layer of CLIM defines an extensible framework for constructing, using, and managing such *hierarchies* of interactive regions. This framework allows uniform treatment of the following things:

- Window objects like those in X Windows.
- Lightweight gadgets typical of toolkit layers, such as Motif.
- Structured graphics like output records and an application's presentation objects.
- Objects that act as Lisp handles for windows or gadgets implemented in a different language.

From the perspective of most CLIM users, CLIM's windowing layer plays the role of a window system. However, CLIM actually uses the services of a window system platform to provide efficient windowing, input, and output facilities.

The fundamental window abstraction defined by CLIM is called a *sheet*. A sheet can participate in a relationship called a *windowing relationship*. This relationship is one in which one sheet called the *parent* provides space to a number of other sheets called *children*. Support for establishing and maintaining this kind of relationship is the essence of what window systems provide.

Programmers can manipulate unrooted hierarchies of sheets (those without a connection to any particular display server). However, a sheet hierarchy must be attached to a display server to make it visible. *Ports* and *grafts* provide the functionality for managing this capability. A *port* is a connection to a display service that is responsible for managing host display server resources and for processing input events received from the host display server. A *graft* is a special kind of sheet that represents a host window, typically a root window (that is, a screen-level window). A sheet is attached to a display by making it a child of a graft, which represents an appropriate host window. The sheet will then appear to be a child of that host window. So, a sheet is put onto a particular screen by making it a child of an appropriate graft and enabling it.

20.1.1 Basic properties of sheets

A sheet is the basic abstraction for implementing windows in CLIM.

`sheet` **[Class]**

■ The protocol class that corresponds to the output state for some kind of sheet. There is no single advertised standard sheet class. If you want to create a new class that obeys the sheet protocol, it must be a subclass of `sheet`

`sheetp` **[Function]**

Arguments: *`object`*

■ Returns `t` if *`object`* is a sheet, otherwise returns `nil`.

All sheets have the following basic properties:

A coordinate system

Provides the ability to refer to locations in a sheet's abstract plane.

A region

Defines an area within a sheet's coordinate system that indicates the area of interest within the plane, that is, a clipping region for output and input. This typically corresponds to the visible region of the sheet on the display.

A parent

A sheet that is the parent in a windowing relationship in which this sheet is a child.

Children

An ordered set of sheets that are each a child in a windowing relationship in which this sheet is a parent. The ordering of the set corresponds to the stacking order of the sheets. Not all sheets have children.

A transformation

Determines how points in this sheet's coordinate system are mapped into points in its parents.

An enabled flag

Indicates whether the sheet is currently actively participating in the windowing relationship with its parent and siblings.

An event handler

A procedure invoked when the display server wishes to inform CLIM of external events.

An output state

A set of values used when CLIM causes graphical or textual output to appear on the display. This state is often represented by a medium.

20.1.2 Basic sheet protocols

A sheet is a participant in a number of protocols. Every sheet must provide methods for the generic functions that make up these protocols. These protocols are:

The windowing protocol

Describes the relationships between the sheet and its parent and children (and, by extension, all of its ancestors and descendants).

The input protocol

Provides the event handler for a sheet. Events may be handled synchronously, asynchronously, or not at all.

The output protocol

Provides graphical and textual output, and manages descriptive output state such as color, transformation, and clipping.

The repaint protocol

Invoked by the event handler and by user programs to ensure that the output appearing on the display device appears as the program expects it to appear.

The notification protocol

Invoked by the event handler and user programs to ensure that CLIM's representation of window system information is equivalent to the display server's.

These protocols may be handled directly by a sheet, queued for later processing by some other agent, or passed on to a delegate sheet for further processing.

20.2 Sheet geometry

Every sheet has a region and a coordinate system. A sheet's region refers to its position and extent on the display device, and is represented by some sort of a region object, frequently a rectangle. A sheet's coordinate system is represented by a coordinate transformation that converts coordinates in its coordinate system to coordinates in its parent's coordinate system.

20.2.1 Sheet geometry functions

`sheet-transformation` **[Generic function]**

Arguments: *`sheet`*

`(setf sheet-transformation)` **[Generic function]**

Arguments: *`transformation sheet`*

- Returns a transformation that converts coordinates in *`sheet`*'s coordinate system into coordinates in its parent's coordinate system. Using **`setf`** on this accessor will modify the sheet's coordinate system, including moving its region in its parent's coordinate system.
- When the sheet's transformation is changed, **`note-sheet-transformation-changed`** is called on *`sheet`* to notify the sheet of the change.

`sheet-region` **[Generic function]**

Arguments: *`sheet`*

`(setf sheet-region)` **[Generic function]**

Arguments: *`region sheet`*

- Returns a region object that represents the set of points to which *`sheet`* refers. The region is in the sheet's coordinate system. Using **`setf`** on this accessor modifies the sheet's region.
- When the sheet's region is changed, **`note-sheet-region-region`** is called on *`sheet`* to notify the sheet of the change.

note-sheet-region-changed [Generic function]

Arguments: *sheet*

note-sheet-transformation-changed [Generic function]

Arguments: *sheet*

■ These notification functions are invoked when the region or transformation of *sheet* has been changed.

move-sheet [Generic function]

Arguments: *sheet x y*

■ Moves *sheet* to the new position (*x*,*y*). *x* and *y* are expressed in the coordinate system of *sheet*'s parent. Note that this is a low-level function which is not typically called by user code. The function **position-sheet-carefully** can be used to move top-level sheets (i.e. windows) and it is the function normally called in user code for that purpose.

■ **move-sheet** works by modifying *sheet*'s transformation, and could be thought of as being implemented as follows:

```
(defmethod move-sheet ((sheet clim:basic-sheet) x y)
 (let ((transform (clim:sheet-transformation sheet)))
  (multiple-value-bind (old-x old-y)
    (clim:transform-position transform 0 0)
   (setf (clim:sheet-transformation sheet)
      (clim:compose-translation-with-transformation
       transform (- x old-x) (- y old-y))))))
```

resize-sheet [Generic function]

Arguments: *sheet width height*

■ Resizes *sheet* to have a new width *width* and a new height *height*. *width* and *height* are real numbers.

■ **resize-sheet** works by modifying *sheet*'s region, and could be thought of as being implemented as follows:

```
(defmethod resize-sheet ((sheet clim:basic-sheet) width height)
 (setf (clim:sheet-region sheet)
    (clim:make-bounding-rectangle 0 0 width height)))
```

move-and-resize-sheet [Generic function]

Arguments: *sheet x y width height*

■ Moves *sheet* to the new position (*x*,*y*), and changes its size to the new width *width* and the new height *height*. *x* and *y* are expressed in the coordinate system of *sheet*'s parent. *width* and *height* are real numbers.

■ **move-and-resize-sheet** could be implemented as follows:

```
(defmethod move-and-resize-sheet ((sheet clim:basic-sheet) x y width height)
 (clim:move-sheet sheet x y)
 (clim:resize-sheet sheet width height))
```

map-sheet-position-to-parent [Generic function]

Arguments: *sheet x y*

■ Applies *sheet*'s transformation to the point (*x*,*y*), returning the coordinates of that point in *sheet*'s parent's coordinate system.

map-sheet-position-to-child [Generic function]

Arguments: *sheet x y*

■ Applies the inverse of *sheet*'s transformation to the point (*x*,*y*) (represented in *sheet*'s parent's coordinate system), returning the coordinates of that same point in *sheet* coordinate system.

map-sheet-rectangle*-to-parent [Generic function]

Arguments: *sheet x1 y1 x2 y2*

■ Applies *sheet*'s transformation to the bounding rectangle specified by the corner points (*x1*,*y1*) and (*x2*,*y2*), returning the bounding rectangle of the transformed region as four values, *min-x*, *min-y*, *max-x*, and *max-y*. The arguments *x1*, *y1*, *x2*, and *y1* are canonicalized in the same way as for **make-bounding-rectangle**.

map-sheet-rectangle*-to-child [Generic function]

Arguments: *sheet x1 y1 x2 y2*

■ Applies the inverse of *sheet*'s transformation to the bounding rectangle delimited by the corner points (*x1*,*y1*) and (*x2*,*y2*) (represented in *sheet*'s parent's coordinate system), returning the bounding rectangle of the transformed region as four values, *min-x*, *min-y*, *max-x*, and *max-y*. The arguments *x1*, *y1*, *x2*, and *y1* are canonicalized in the same way as for **make-bounding-rectangle**.

map-over-sheets-containing-position [Generic function]

Arguments: *function sheet x y*

■ Applies the function *function* to all of the children of the sheet *sheet* that contain the position (*x*,*y*). *x* and *y* are expressed in *sheet*'s coordinate system.

■ *function* is a function of one argument, the sheet; it has dynamic extent.

map-over-sheets-overlapping-region [Generic function]

Arguments: *function sheet region*

■ Applies the function *function* to all of the children of the sheet *sheet* that overlap the region *region*. *region* is expressed in *sheet*'s coordinate system.

■ *function* is a function of one argument, the sheet; it has dynamic extent.

child-containing-position [Generic function]

Arguments: *sheet x y*

■ Returns the topmost enabled direct child of *sheet* whose region contains the position (*x*,*y*). The position is expressed in *sheet*'s coordinate system.

20.3 Relationships between sheets

Sheets are arranged in a tree-shaped hierarchy. In general, a sheet has one parent (or no parent) and zero or more children. A sheet may have zero or more siblings (that is, other sheets that share the same parent). In order to describe the relationships between sheets, we need to define some terms.

Adopted

A sheet is said to be *adopted* if it has a parent. A sheet becomes the parent of another sheet by adopting that sheet.

Disowned

A sheet is said to be *disowned* if it does not have a parent. A sheet ceases to be a child of another sheet by being disowned.

Grafted

A sheet is said to be *grafted* when it is part of a sheet hierarchy whose highest ancestor is a graft. In this case, the sheet may be visible on a particular window server.

Degrafted

A sheet is said to be *degrafted* when it is part of a sheet hierarchy that cannot possibly be visible on a server, that is, the highest ancestor is not a graft.

Enabled

A sheet is said to be *enabled* when it is actively participating in the windowing relationship with its parent. If a sheet is enabled and grafted, and all its ancestors are enabled (they are grafted by definition), then the sheet will be visible if it occupies a portion of the graft region that isn't clipped by its ancestors or ancestor's siblings.

Disabled

The opposite of enabled is *disabled*.

20.3.1 Sheet relationship functions

The generic functions in this section comprise the sheet protocol. All sheet objects must implement or inherit methods for each of these generic functions.

`sheet-parent` [Generic function]

Arguments: *sheet*

- Returns the parent of *sheet*, or `nil` if the sheet has no parent.

`sheet-children` [Generic function]

Arguments: *sheet*

- Returns a list of sheets that are the children of *sheet*. Some sheet classes support only a single child; in this case, the result of **sheet-children** will be a list of one element.
- Do not modify the value returned by this function.

`sheet-adopt-child` [Generic function]

Arguments: *sheet child*

- Adds the child sheet *child* to the set of children of *sheet*, and makes the *sheet* the child's parent. If *child* already has a parent, an error will be signaled.

■ Some sheet classes support only a single child. For such sheets, attempting to adopt more than a single child will cause an error to be signaled.

sheet-disown-child [Generic function]

Arguments: *sheet child* &key (errorp t)

■ Removes the child sheet *child* from the set of children of *sheet*, and makes the parent of the child be nil. If *child* is not actually a child of *sheet* and *errorp* is t, then an error will be signaled.

raise-sheet [Generic function]

Arguments: *sheet*

bury-sheet [Generic function]

Arguments: *sheet*

■ These functions reorder the children of a sheet by raising *sheet* to the top or burying it at the bottom. Raising a sheet puts it at the beginning of the ordering; burying it puts it at the end. If sheets overlap, the one that appears on top on the display device is earlier in the ordering than the one underneath.

■ This may change which parts of which sheets are visible on the display device.

reorder-sheets [Generic function]

Arguments: *sheet new-ordering*

■ Reorders the children of *sheet* to have the new ordering specified by *new-ordering*. *new-ordering* is an ordered list of the child sheets; elements at the front of *new-ordering* are on top of elements at the rear.

■ If *new-ordering* does not contain all of the children of *sheet*, then an error will be signaled. If *new-ordering* contains a sheet that is not a child of *sheet*, then an error will be signaled.

sheet-enabled-p [Generic function]

Arguments: *sheet*

■ Returns t if the *sheet* is enabled by its parent, otherwise returns nil. Note that all of a sheet's ancestors must be enabled before the sheet is viewable.

(setf sheet-enabled-p) [Generic function]

Arguments: *enabled-p sheet*

■ When *enabled-p* is t, this enables *sheet*. When *enabled-p* is nil, this disables the sheet.

■ Note that a sheet is not visible unless it and all of its ancestors are enabled.

map-over-sheets [Generic function]

Arguments: *function sheet*

■ Applies the function *function* to the sheet *sheet*, and then applies *function* to all of the descendents (the children, the children's children, and so forth) of *sheet*.

■ *function* is a function of one argument, the sheet; it has dynamic extent.

20.4 Sheet input protocol

CLIM's windowing substrate provides an input architecture and standard functionality for notifying clients of input that is distributed to their sheets. Input includes such events as the pointer entering and exiting sheets, pointer motion (whose granularity is defined by performance limitations), and pointer button and keyboard events. At this level, input is represented as *event objects*.

In addition to handling input event, a sheet is also responsible for providing other input services, such as controlling the pointer's appearance, and polling for current pointer and keyboard state.

Input events can be broadly categorized into *pointer events* and *keyboard events*. By default, pointer events are dispatched to the lowest sheet in the hierarchy whose region contains the location of the pointer. Keyboard events are dispatched to the port's keyboard input focus; the accessor **`port-keyboard-input-focus`** contains the event client that receives the port's keyboard events.

20.4.1 Input protocol functions

The following are the most useful functions in the sheet input protocol. These are what you need to be cognizant of to write your own classes of gadgets.

`sheet-event-queue` [Generic function]

Arguments: *`sheet`*

■ Any sheet that can process events will have an event queue from which the events are gotten. **`sheet-event-queue`** returns the object that acts as the event queue. The exact representation of an event queue is explicitly unspecified.

`handle-event` [Generic function]

Arguments: *`sheet event`*

■ Handles the event *`event`* on behalf of the sheet *`sheet`*. For example, if you want to highlight a sheet in response to an event that informs it that the pointer has entered its territory, there would be a method to carry out the policy that specializes the appropriate sheet and event classes.

In addition to **`queue-event`**, the queued input protocol handles the following generic functions. The *`client`* argument to these functions is typically a sheet.

When you implement your own gadget classes, you will probably write one or more **`handle-event`** methods to manage such things as pointer button presses, pointer motion into the gadget, and so on.

20.5 Sheet output protocol

The output protocol is concerned with the appearance of displayed output on the window associated with a sheet. The sheet output protocol is responsible for providing a means of doing output to a sheet, and for delivering repaint requests to the sheet's client.

Each sheet retains some output state that logically describes how output is to be rendered on its window. Such information as the foreground and background ink, line thickness, and transformation to be used during drawing are provided by this state. This state may be stored in a *medium* associated with the sheet itself, be derived from a parent, or may have some global default, depending on the sheet itself.

`medium` **[Class]**

■ The protocol class that corresponds to the output state for some kind of sheet. There is no single advertised standard medium class. If you want to create a new class that obeys the medium protocol, it must be a subclass of `medium`.

`mediump` [Function]

Arguments: *`object`*

■ Returns `t` if *`object`* is a medium, otherwise returns `nil`.

The following generic functions comprise the basic medium protocol. All mediums must implement methods for these generic functions. Often, a sheet class that supports the output protocol will implement a trampoline method that passes the operation on to **`sheet-medium`** of the sheet. All of these are described in more detail in chapter 4 **The CLIM drawing environment**.

`medium-foreground` [Generic function]

Arguments: *`medium`*

`(setf medium-foreground)` [Generic function]

Arguments: *`design medium`*

■ Returns (and, with **`setf`**, sets) the current foreground ink for *`medium`*.

`medium-background` [Generic function]

Arguments: *`medium`*

`(setf medium-background)` [Generic function]

Arguments: *`design medium`*

■ Returns (and, with **`setf`**, sets) the current background ink for *`medium`*.

`medium-ink` [Generic function]

Arguments: *`medium`*

`(setf medium-ink)` [Generic function]

Arguments: *`design medium`*

■ Returns (and, with **`setf`**, sets) the current drawing ink for *`medium`*.

`medium-transformation` [Generic function]

Arguments: *`medium`*

`(setf medium-transformation)` [Generic function]

Arguments: *`transformation medium`*

■ Returns (and, with **`setf`**, sets) the user transformation that converts the coordinates presented to the drawing functions by the programmer to *`medium`*'s coordinate system. By default, it is the identity transformation.

medium-clipping-region [Generic function]

Arguments: *medium*

(setf medium-clipping-region) [Generic function]

Arguments: *region medium*

■ Returns (and, with **setf**, sets) the clipping region that encloses all output performed on *medium*. It is returned and set in user coordinates. That is, to convert the user clipping region to medium coordinates, it must be transformed by the value of **medium-transformation**.

medium-line-style [Generic function]

Arguments: *medium*

(setf medium-line-style) [Generic function]

Arguments: *line-style medium*

■ Returns (and, with **setf**, sets) the current line style for *medium*.

medium-text-style [Generic function]

Arguments: *medium*

(setf medium-text-style) [Generic function]

Arguments: *text-style medium*

■ Returns (and, with **setf**, sets) the current text style for *medium* of any textual output that may be displayed on the window.

medium-default-text-style [Generic function]

Arguments: *medium*

(setf medium-default-text-style) [Generic function]

Arguments: *text-style medium*

■ Returns (and, with **setf**, sets) the default text style for output on *medium*.

medium-merged-text-style [Generic function]

Arguments: *medium*

■ Returns the actual text style used in rendering text on *medium*. It returns the result of

```
(clim:merge-text-styles (clim:medium-text-style medium)
                        (clim:medium-default-text-style medium))
```

Thus, those components of the current text style that are not `nil` will replace the defaults from medium's default text style. Unlike the preceding text style function, **medium-merged-text-style** is read-only.

20.5.1 Associating a medium with a sheet

Before a sheet may be used for output, it must be associated with a medium. Some sheets are permanently associated with media for output efficiency; for example, CLIM window stream sheets have a medium that is permanently allocated to the window.

However, many kinds of sheets only perform output infrequently, and therefore do not need to be associated with a medium except when output is actually required. Sheets without a permanently associated medium can be much more lightweight than they otherwise would be. For example, in a program that creates a sheet for the purpose of displaying a border for another sheet, the border sheet receives output only when the window's shape is changed.

To associate a sheet with a medium, the macro **`with-sheet-medium`** is used. Only sheets that are subclasses of **`sheet-with-medium-mixin`** may have a medium associated with them.

`with-sheet-medium` **[Macro]**

Arguments: *`(medium sheet)`* `&body` *`body`*

- Within the body, the variable *`medium`* is bound to the sheet's medium. If the sheet does not have a medium permanently allocated, one will be allocated and associated with the sheet for the duration of the body, and then degrafted from the sheet and deallocated when the body has been exited. The values of the last form of the body are returned as the values of **`with-sheet-medium`**.
- This macro will signal a runtime error if sheet is not available for doing output.

`sheet-medium` **[Generic function]**

Arguments: *`sheet`*

- Returns the medium associated with *`sheet`*. If *`sheet`* does not have a medium allocated to it, **`sheet-medium`** returns `nil`.
- This function will signal an error if sheet is not available for doing output.

`medium-sheet` **[Generic function]**

Arguments: *`medium`*

- Returns the sheet associated with *`medium`*. If *`medium`* is not grafted to a sheet, **`medium-sheet`** returns `nil`.

`medium-drawable` **[Generic function]**

Arguments: *`medium`*

- Returns an implementation-dependent object that corresponds to the actual host window that will be drawn on when *`medium`* is drawn on. If *`medium`* is not grafted to a sheet or the medium's sheet is not currently mirrored on a display server, **`medium-drawable`** returns `nil`.
- You can use this function to get a host window system object that can be manipulated using the functions of the host window system. This might be done in order to explicitly trade of performance against portability.

`sheet-mirror` **[Generic function]**

Arguments: *`sheet`*

- Returns an implementation-dependent object that corresponds to the actual host window that will be drawn on when you draw on *`sheet`*'s medium.

20.6 Repainting protocol

CLIM's repainting protocol is the mechanism whereby a program keeps the display up-to-date, reflecting the results of both synchronous and asynchronous events. The repaint mechanism may be invoked by user programs each time through their top-level command loop. It may also be invoked directly or indirectly as a result of events received from the display server host. For example, if a window is on display with another window overlapping it, and the second window is buried, a 'damage notification' event may be sent by the server; CLIM would cause a repaint to be executed for the newly-exposed region.

20.6.1 Repaint protocol functions

`queue-repaint` [Generic function]

Arguments: *sheet repaint-event*

- Requests that the repaint event *repaint-event* be placed in the input queue of *sheet*. A program that reads events out of the queue will be expected to call **`handle-event`** for the sheet using the repaint region gotten from *repaint-event*.

`handle-repaint` [Generic function]

Arguments: *sheet region*

- Implements repainting for a given sheet class. *sheet* is the sheet to repaint and *region* is the region to repaint.
- When you implement your own gadget classes, you will probably write a **`handle-repaint`** method that draws the gadget.

`repaint-sheet` [Generic function]

Arguments: *sheet region*

- Recursively causes repainting of *sheet* and any of its descendants that overlap the region *region*.

20.7 Ports, grafts, and mirrored sheets

A sheet hierarchy must be attached to a display server so as to permit input and output. This is managed by the use of *ports* and *grafts*.

A *port* is a logical connection to a display server. It is responsible for managing display output and server resources, and for handling incoming input events. Typically, the programmer will create a single port that will manage all of the windows on the display.

A *graft* is a special sheet that is directly connected to a display server. Typically, a graft is the CLIM sheet that represents the root window of the display. There may be several grafts that are all attached to the same root window; these grafts may have differing coordinate systems.

To display a sheet on a display, it must have a graft for an ancestor. In addition, the sheet and all of its ancestors must be enabled, including the graft. In general, a sheet becomes grafted when it (or one of its ancestors) is adopted by a graft.

A *mirrored sheet* is a special class of sheet that is attached directly to a window on a display server. Grafts, for example, are always mirrored sheets. However, any sheet anywhere in a sheet hierarchy may be

a mirrored sheet. A mirrored sheet will usually contain a reference to a window system object, called a mirror. For example, a mirrored sheet attached to an X11 server might have an X window system object stored in one of its slots. Allowing mirrored sheets at any point in the hierarchy enables the adaptive toolkit facilities.

Since not all sheets in the hierarchy have mirrors, there is no direct correspondence between the sheet hierarchy and the mirror hierarchy. However, on those display servers that support hierarchical windows, the hierarchies must be parallel. If a mirrored sheet is an ancestor of another mirrored sheet, their corresponding mirrors must have a similar ancestor/descendant relationship.

CLIM interacts with mirrors when it must display output or process events. On output, the mirrored sheet closest in ancestry to the sheet on which we wish to draw provides the mirror on which to draw. The mirror's drawing clipping region is set up to be the intersection of the user's clipping region and the sheet's region (both transformed to the appropriate coordinate system) for the duration of the output. On input, events are delivered from mirrors to the sheet hierarchy. The CLIM port must determine which sheet shall receive events based on information such as the location of the pointer.

In both of these cases, we must have a coordinate transformation that converts coordinates in the mirror (so-called 'native' coordinates) into coordinates in the sheet and vice-versa.

20.7.1 Ports

A port is described with a *server path*. A server path is a list whose first element is a keyword that selects the kind of port. The remainder of the server path is a list of alternating keywords and values whose interpretation is port type-specific.

`find-port` [Function]

Arguments: `&rest initargs &key (server-path *default-server-path*) &allow-other-keys`

■ Finds a port that provides a connection to the window server addressed by *`server-path`*. If no such connection exists, a new connection will be constructed and returned. The initargs in *`initargs`* will be passed to the function that constructed the new port.

`*default-server-path*` [Variable]

■ This special variable is used by **`find-port`** and its callers to default the choice of a display service to locate. Binding this variable in a dynamic context will affect the defaulting of this argument to these functions. This variable will be defaulted according to the environment. In Allegro CLIM, CLIM will attempt to set this variable based on the value of the DISPLAY environment variable.

■ The initial value of `*default-server-path*` is `(:motif)`.

Allegro CLIM supports only the following port type:

`:motif` [Server path]

Arguments: `&key display application-name application-class`

■ Given the server path, **`find-port`** finds a Motif port connected to the X display *`display`*.

■ In Allegro CLIM, if *`display`* is not supplied, the value come from the DISPLAY environment variable.

■ *`application-name`* and *`application-class`* can be used to change the X name and class of the CLIM from, respectively, `clim` and `Clim`. The values should be strings. It is rarely necessary to change the name and class.

`port` **[Server path]**

Arguments: *object*

■ Returns the port associated with *object*. **port** is defined for all sheet classes (including grafts and streams that support the CLIM graphics protocol), mediums, and application frames. For degrafted sheets or other objects that aren't currently associated with particular ports, **port** will return `nil`.

map-over-ports [Function]

Arguments: *function*

■ Invokes *function* on each existing port. *function* is a function of one argument, the port; it has dynamic extent.

port-server-path [Generic function]

Arguments: *port*

■ Returns the server path associated with *port*.

port-name [Generic function]

Arguments: *port*

■ Returns an implementation-dependent string that is the name of the port. For example, a `:motif` port might have a name of "summer:0.0".

port-type [Generic function]

Arguments: *port*

■ Returns the type of the port, that is, the first element of the server path spec.

restart-port [Generic function]

Arguments: *port*

■ In a multi-process Lisp, **restart-port** restarts the global input processing loop associated with *port*. All pending input events are discarded. Server resources may or may not be released and reallocated during or after this action. Note that *port* cannot be a port destroyed by **destroy-port**, since such ports cannot be restarted.

destroy-port [Generic function]

Arguments: *port*

■ Destroys the connection with the window server represented by *port*. All sheet hierarchies that are associated with *port* are forcibly degrafted by disowning the children of grafts on *port* using **sheet-disown-child**. All server resources utilized by such hierarchies or by any graphics objects on *port* are released as part of the connection shutdown. Once this function has been applied to a port, the port is dead and cannot be used again for any purpose. In particular, it cannot be restarted by **restart-port**.

20.7.2 Internal Interfaces for Native Coordinates

`sheet-device-transformation` [Generic function]

Arguments: *sheet*

■ Returns the transformation used by the graphics output routines when drawing on the mirror. The object returned by this function is volatile, so programmers must not depend on the components of the object remaining constant.

`sheet-device-region` [Generic function]

Arguments: *sheet*

■ Returns the actual clipping region to be used when drawing on the mirror. The object returned by this function is volatile, so programmers must not depend on the components of the object remaining constant.

[This page intentionally left blank.]

Index

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y