align-x, *cell-align-y*, and *move-cursor* are as for `formatting-item-list`.

⇒ `menu-choose-command-from-command-table` *command-table* `&key` *associated-window default-style label cache unique-id id-test cache-value cache-test* [*Function*]

Invokes a window system specific routine that displays a menu of commands from *command-table*'s menu, and allows the user to choose one of the commands. *command-table* is a *command table designator*. The returned value is a command object. This may invoke itself recursively when there are sub-menus.

associated-window, *default-style*, *label*, *cache*, *unique-id*, *id-test*, *cache-value*, and *cache-test* are as for `menu-choose`.

Minor issue: *Should this be generic on application frames? — SWM*

27.4 Keystroke Accelerators

Each command table may have a mapping from keystroke accelerator gesture names to command menu items. When a user types a key on the keyboard that corresponds to the gesture for keystroke accelerator, the corresponding command menu item will be invoked. Note that command menu items are shared among the command table's menu and the accelerator table. There are several reasons for this. One is that it is common to have menus display the keystroke associated with a particular item, if there is one.

Note that, despite the fact the keystroke accelerators are specified using keyboard gesture names rather than characters, the conventions for typed characters vary widely from one platform to another. Therefore the programmer must be careful in choosing keystroke accelerators. Some sort of per-platform conditionalization is to be expected.

⇒ `add-keystroke-to-command-table` *command-table gesture type value* `&key` *documentation (errorp t)* [*Function*]

Adds a command menu item to *command-table*'s keystroke accelerator table. *gesture* is a keyboard gesture name to be used as the accelerator. *type* and *value* are as in `add-menu-item-to-command-table`, except that *type* must be either `:command`, `:function` or `:menu`. *command-table* is a *command table designator*.

Minor issue: *Should we allow gesture to be a gesture specification as well as just a gesture name? It simplifies its use by avoiding a profusion of defined gesture names, but we may want to encourage the use of gesture names. — SWM*

documentation is a documentation string, which can be used as documentation for the keystroke accelerator.

If the command menu item associated with *gesture* is already present in the command table's

accelerator table and *errorp* is *true*, then the `command-already-present` error will be signalled. When the item is already present in the command table's accelerator table and *errorp* is *false*, the old item will first be removed.

⇒ `remove-keystroke-from-command-table` *command-table gesture* `&key` *(errorp t)* [*Function*]

Removes the command menu item named by keyboard gesture name *gesture* from *command-table*'s accelerator table. *command-table* is a *command table designator*.

If the command menu item associated with *gesture* is not present in the command table's menu and *errorp* is *true*, then the `command-not-present` error will be signalled.

⇒ `map-over-command-table-keystrokes` *function command-table* [*Function*]

Applies *function* to all of the keystroke accelerators in *command-table*'s accelerator table. *function* must be a function of three arguments, the menu name (which will be `nil` if there is none), the keystroke accelerator, and the command menu item; it has dynamic extent. *command-table* is a *command table designator*.

`map-over-command-table-keystrokes` does not descend into sub-menus. If the programmer requires this behavior, he should examine the type of the command menu item to see if it is `:menu`.

`map-over-command-table-keystrokes` returns `nil`.

⇒ `find-keystroke-item` *gesture command-table* `&key` *test (errorp t)* [*Function*]

Given a keyboard gesture *gesture* and a command table, returns two values, the command menu item associated with the gesture and the command table in which it was found. (Since keystroke accelerators are not inherited, the second returned value will always be *command-table*.)

This function returns objects that reveal CLIM's internal state; do not modify those objects.

test is the test used to compare the supplied gesture to the gesture name in the command table. The supplied gesture will generally be an event object, so the default for *test* is `event-matches-gesture-name-p`.

If the keystroke accelerator is not present in *command-table* and *errorp* is *true*, then the `command-not-present` error will be signalled. *command-table* is a *command table designator*.

⇒ `lookup-keystroke-item` *gesture command-table* `&key` *test* [*Function*]

Given a keyboard gesture *gesture* and a command table, returns two values, the command menu item associated with the gesture and the command table in which it was found. Note that *gesture* may be either a keyboard gesture name of a gesture object, and is handled in the same way as in `find-keystroke-item`. This function returns objects that reveal CLIM's internal state; do not modify those objects.

Unlike `find-keystroke-item`, this follows the sub-menu chains that can be created with `add-menu-item-to-command-table`. If the keystroke accelerator cannot be found in the command

table or any of the command tables from which it inherits, `lookup-keystroke-item` will return `nil`. *command-table* is a *command table designator*.

test is the test used to compare the supplied gesture to the gesture name in the command table. The supplied gesture will generally be an event object, so the default for *test* is `event-matches-gesture-name-p`.

⇒ `lookup-keystroke-command-item` *gesture command-table* `&key` *test numeric-arg* [*Function*]

Given a keyboard gesture *gesture* and a command table, returns the command associated with the keystroke, or *gesture* if no command is found. Note that *gesture* may be either a keyboard gesture name of a gesture object, and is handled in the same way as in `find-keystroke-item`. This function returns objects that reveal CLIM's internal state; do not modify those objects.

This is like `find-keystroke-item`, except that only keystrokes that map to an enabled application command will be matched. *command-table* is a *command table designator*.

test is the test used to compare the supplied gesture to the gesture name in the command table. The supplied gesture will generally be an event object, so the default for *test* is `event-matches-gesture-name-p`.

numeric-arg (which defaults to 1) is substituted into the resulting command for any occurrence of `*numeric-argument-marker*` in the command. This is intended to allow programmers to define keystroke accelerators that take simple numeric arguments, which will be passed on by the input editor.

Minor issue: *Do the above three functions need to have their hands on the port? If* `event-matches-gesture-name-p` *needs the port, then the answer is yes. Otherwise, if gesture names are "global" across all ports, then these don't need the port. — SWM*

⇒ `substitute-numeric-argument-marker` *command numeric-arg* [*Function*]

Given a command object *command*, this substitutes the value of *numeric-arg* for all occurrences of the value of `*numeric-argument-marker*` in the command, and returns a command object with those substitutions.

27.5 Presentation Translator Utilities

These are some utilities for maintain presentation translators in command tables. Presentation translators are discussed in more detail in Chapter 23.

⇒ `add-presentation-translator-to-command-table` *command-table translator-name* `&key` *(errorp t)* [*Function*]

Adds the translator named by *translator-name* to *command-table*. The translator must have been previously defined with `define-presentation-translator` or `define-presentation-to-`

`command-translator`. *command-table* is a *command table designator*.

If *translator-name* is already present in *command-table* and *errorp* is *true*, then the `command-already-present` error will be signalled. When the translator is already present and *errorp* is *false*, the old translator will first be removed.

⇒ `remove-presentation-translator-from-command-table` *command-table translator-name* `&key` *(errorp t)* [*Function*]

Removes the translator named by *translator-name* from *command-table*. *command-table* is a *command table designator*.

If the translator is not present in the command table and *errorp* is *true*, then the `command-not-present` error will be signalled.

⇒ `map-over-command-table-translators` *function command-table* `&key` *(inherited t)* [*Function*]

Applies *function* to all of the translators accessible in *command-table*. *function* must be a function of one argument, the translator; it has dynamic extent. *command-table* is a *command table designator*.

If *inherited* is *false*, this applies *function* only to those translators present in *command-table*, that is, it does not map over any inherited command tables. If *inherited* is *true*, then the inherited command tables are traversed in the same order as for `do-command-table-inheritance`.

`map-over-command-table-translators` returns `nil`.

⇒ `find-presentation-translator` *translator-name command-table* `&key` *(errorp t)* [*Function*]

Given a translator name and a command table, returns two values, the presentation translator and the command table in which it was found. If the translator is not present in *command-table* and *errorp* is *true*, then the `command-not-accessible` error will be signalled. *command-table* is a *command table designator*.

27.6 The Command Processor

Once a set of commands has been defined, CLIM provides a variety of means to read a command. These are all mediated by the Command Processor.

⇒ `read-command` *command-table* `&key` *(stream *standard-input*) command-parser command-unparser partial-command-parser use-keystrokes* [*Function*]

`read-command` is the standard interface used to read a command line. *stream* is an extended input stream, and *command-table* is a *command table designator*.

command-parser must be a function of two arguments, a command table and a stream. It reads a command from the user and returns a command object, or `nil` if an empty command line was

read. The default value for *command-parser* is the value of `*command-parser*`.

command-unparser must be a function of three arguments, a command table, a stream, and a command to "unparse". It prints a textual description of the command its supplied arguments onto the stream. The default value for *command-unparser* is the value of `*command-unparser*`.

partial-command-parser must be a function of four arguments, a command table, a stream, a partial command, and a start position. The partial command is a command object with the value of `*unsupplied-argument-marker*` in place of any argument that needs to be filled in. The function reads the remaining, unsupplied arguments in any way it sees fit (for example, via an `accepting-values` dialog), and returns a command object. The start position is the original input-editor scan position of the stream, when the stream is an interactive stream. The default value for *partial-command-parser* is the value of `*partial-command-parser*`.

command-parser, *command-unparser*, and *partial-command-parser* have dynamic extent.

When *use-keystrokes* is *true*, the command reader will also process keystroke accelerators. (Implementations will typically use `with-command-table-keystrokes` and `read-command-using-keystrokes` to implement the case when *use-keystrokes* is *true*.)

Input editing, while conceptually an independent facility, fits into the command processor via its use of `accept`. That is, `read-command` must be implemented by calling `accept` to read command objects, and `accept` itself makes use of the input editing facilities.

⇒ `with-command-table-keystrokes` *(keystroke-var command-table)* `&body` *body* [*Macro*]

Binds *keystroke-var* to a sequence that contains all of the keystroke accelerators in *command-table*'s menu, and then executes *body* in that context. *command-table* is a *command table designator*. *body* may have zero or more declarations as its first forms.

⇒ `read-command-using-keystrokes` *command-table keystrokes* `&key` *(stream *standard-input*) command-parser command-unparser partial-command-parser* [*Function*]

Reads a command from the user via command lines, the pointer, or a single keystroke, and returns either a command object, or a keyboard gesture object if the user typed a keystroke that is in *keystrokes* but does not have a command associated with it in *command-table*.

keystrokes is a sequence of keyboard gesture names that are the keystroke accelerators.

command-table, *stream*, *command-parser*, *command-unparser*, and *partial-command-parser* are as for `read-command`.

⇒ `command-line-command-parser` *command-table stream* [*Function*]

The default command-line parser. It reads a command name and the command's arguments as a command line from *stream* (with completion as much as is possible), and returns a command object. *command-table* is a *command table designator* that specifies the command table to use; the commands are read via the textual command-line name.

⇒ `command-line-command-unparser` *command-table stream command* [*Function*]

The default command-line unparser. It prints the command *command* as a command name and its arguments as a command line on *stream*. *command-table* is a *command table designator* that specifies the command table to use; the commands are displayed using the textual command-line name.

⇒ `command-line-read-remaining-arguments-for-partial-command` *command-table stream partial-command start-position* [*Function*]

The default partial command-line parser. If the remaining arguments are at the end of the command line, it reads them as a command line, otherwise it constructs a dialog using `accepting-values` and reads the remaining arguments from the dialog. *command-table* is a *command table designator*.

⇒ `menu-command-parser` *command-table stream* [*Function*]

The default menu-driven command parser. It uses only pointer clicks to construct a command. It relies on presentations of all arguments being visible. *command-table* and *stream* are as for `command-line-parser`.

There is no menu-driven command unparser, since it makes no sense to unparse a completely menu-driven command.

⇒ `menu-read-remaining-arguments-for-partial-command` *command-table stream partial-command start-position* [*Function*]

The default menu-driven partial command parser. It uses only pointer clicks to fill in the command. Again, it relies on presentations of all arguments being visible. *command-table* is a *command table designator*.

⇒ `*command-parser*` [*Variable*]

Contains the currently active command parsing function. The default value is the function `command-line-command-parser`, which is the default command-line parser.

⇒ `*command-unparser*` [*Variable*]

Contains the currently active command unparsing function. The default value is the function `command-line-command-unparser`, which is the default command-line unparser.

⇒ `*partial-command-parser*` [*Variable*]

Contains the currently active partial command parsing function. The default value is the function `command-line-read-remaining-arguments-for-partial-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.

⇒ `*command-name-delimiters*` [*Variable*]

This is a list of the characters that separate the command name from the command arguments in a command line. The standard set of command name delimiters must include `#\Space`.

⇒ `*command-argument-delimiters*` [*Variable*]

This is a list of the characters that separate the command arguments from each other in a command line. The standard set of command argument delimiters must include `#\Space`.

27.6.1 Command Presentation Types

⇒ `command &key` *command-table* [*Presentation Type*]

The presentation type used to represent a command and its arguments; the command must be accessible in *command-table* and enabled in `*application-frame*`. *command-table* is a *command table designator*. If *command-table* is not supplied, it defaults to the command table for the current application frame, `(frame-command-table *application-frame*)`.

The object returned by the `accept` presentation method for `command` must be a command object, that is, a cons of the command name and the list of the command's arguments.

The `accept` presentation method for the `command` type must call the command parser stored in `*command-parser*` to read the command. The parser will recursively call `accept` to read a `command-name` and all of the command's arguments. The parsers themselves must be implemented by accepting objects whose presentation type is `command`.

If the command parser returns a partial command, the `accept` presentation method for the `command` type must call the partial command prser stored in `*partial-command-parser*`.

The `present` presentation method for the `command` type must call the command unparser stored in `*command-unparser*`.

If a presentation history is maintained for the `command` presentation type, it should be maintained separately for each instance of an application frame.

⇒ `command-name &key` *command-table* [*Presentation Type*]

The presentation type used to represent the name of a command that is both accessible in the command table *command-table* and enabled in `*application-frame*`. *command-table* is a *command table designator*. If *command-table* is not supplied, it defaults to the command table for the current application frame, `(frame-command-table *application-frame*)`.

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 &key` *command-table* [*Presentation Type*]

The presentation type used to represent an object that is either a Lisp form, or a command and its arguments. The command must be accessible in *command-table* and enabled in `*application-frame*`. *command-table* is a *command table designator*. If *command-table* is not supplied, it defaults to the command table for the current application frame, `(frame-command-table *application-frame*)`.

The `accept` presentation method for this type reads a Lisp form, except that if the first character in the user's input is one of the characters in `*command-dispatchers*` it will read a command. The two returned values from the `accept` presentation method will be the command or form object and a presentation type specifier that is either `command` or `form`.

If a presentation history is maintained for the `command-or-form` presentation type, it should be maintained separately for each instance of an application frame.

⇒ `*command-dispatchers*` [*Variable*]

This is a list of the characters that indicates that CLIM reads a command when it is accepting a `command-or-form`. The standard set of command argument delimiters must include the colon character, `#\:`.

Chapter 28

Application Frames

28.1 Overview of Application Frames

Application frames (or simply, *frames*) are the central abstraction defined by CLIM for presenting an application's user interface. Many of the other features and facilities provided by CLIM (for example, the generic command loop, gadgets, look and feel independence) can be conveniently accessed through the frame facility. Frames can be displayed as either top-level windows or regions embedded within the space of the user interfaces of other applications. In addition to controlling the screen real estate managed by an application, a frame keeps track of the Lisp state variables that contain the state of the application.

The visual aspect of an application frame is established by defining a hierarchy of *panes*. CLIM panes are interactive objects that are analogous to the windows, gadgets, or widgets of other toolkits. Application builders can compose their application's user interface from a library of standard panes or by defining and using their own pane types. Application frames can use a number of different types of panes including *layout panes* for spatially organizing panes, *user panes* for presenting application specific information, and *gadget panes* for displaying data and obtaining user input. Panes are describe in greater detail in Chapter 29 and Chapter 30.

Frames are managed by special applications called *frame managers*. Frame managers control the realization of the look and feel of a frame. The frame manager interprets the specification of the application frame in the context of the available window system facilities, taking into account preferences expressed by the user. In addition, the frame manager takes care of attaching the pane hierarchy of an application frame to an appropriate place in a window hierarchy. The most common type of frame manager is one that allows the user to manipulate the frames of other applications. This type of application is typically called a desktop manager, or in X Windows terminology, a window manager. In many cases, the window manager will be a non-Lisp application. In these cases, the frame manager will act as a mediator between the Lisp application and the host desktop manager.

Some applications may act as frame managers that allow the frames of other applications to be

displayed with their own frames. For example, a text editor might allow figures generated by a graphic editor to be edited in place by managing the graphics editor's frame within its own frame.

Application frames provide support for a standard interaction processing loop, like the Lisp "read-eval-print" loop, called a *command loop*. The application programmer has to write only the code that implements the frame-specific commands and output display functions. A key aspect of the command 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. CLIM implementations are free to run the display update part of the loop at a lower priority than command execution, for example, some implementations may choose not to update the display if there is typed-ahead input. Note that by default command execution and display will not occur simultaneously, so user-defined functions need not have to cope with multiprocessing. Of course, the programmer can use multiple processes, but CLIM neither directly supports nor precludes doing so.

To write an application that uses the standard interaction loop provided by CLIM, an application programmer does the following:

- Defines the presentation types that correspond to the user interface entities of the application.
- Defines the commands that correspond to the visible operations of the application, specifying the presentation types of the operands involved in each command.
- Defines the set of frames and panes needed to support the application.
- Defines the output display functions associated with each of the panes (possibly using other facilities such as the incremental redisplay).

Taking as an example a simple ECAD program, the programmer would first define the appropriate presentation types, such as wires, input and output signals, gates, resistors, and so forth. He would then define the program's commands in terms of these types. For example, the "Connect" command might take two operands, one of type "component" and the other of type "wire". The programmer may wish to specify the interaction style for invoking each command, for example, direct manipulation via translators, or selection of commands from menus. After defining an application frame that includes a CLIM stream pane, the programmer then writes the frame-specific display routine that displays the circuit layout. Now the application is ready to go. The end-user selects a command (via a menu or command-line, or whatever), the top-level loop takes care of collecting the operands for that command (via a variety of user gestures), and then the application invokes the command. The command may have a side-effect on the frame's "database" of information, which can in turn affect the output displayed by the redisplay phase.

Note that this definition of the standard interaction loop does not constrain the interaction style to be a command-line interface. The input sentence may be entered via single keystrokes, pointer input, menu selection, dialogs, or by typing full command lines.

28.2 Defining and Creating Application Frames

⇒ `application-frame` [*Protocol Class*]

The protocol class that corresponds to an application frame. If you want to create a new class that behaves like an application frame, it should be a subclass of `application-frame`. All instantiable subclasses of `application-frame` must obey the application frame protocol.

All application frame classes are mutable.

⇒ `application-frame-p` *object* [*Protocol Predicate*]

Returns *true* if *object* is an *application frame*, otherwise returns *false*.

⇒ `:name` [*Initarg*]
⇒ `:pretty-name` [*Initarg*]
⇒ `:command-table` [*Initarg*]
⇒ `:disabled-commands` [*Initarg*]
⇒ `:panes` [*Initarg*]
⇒ `:menu-bar` [*Initarg*]
⇒ `:calling-frame` [*Initarg*]
⇒ `:state` [*Initarg*]
⇒ `:properties` [*Initarg*]

All subclasses of `application-frame` must handle these initargs, which are used to specify, respectively, the name, pretty name, command table, initial set of disabled commands, the panes, the menu bar, calling frame, state, and initial properties for the frame.

⇒ `standard-application-frame` [*Class*]

The instantiable standard class that implements application frames. By default, most application frame classes will inherit from this class, unless a non-`nil` value for *superclasses* is supplied to `define-application-frame`.

⇒ `define-application-frame` *name superclasses slots* `&rest` *options* [*Macro*]

Defines a frame and CLOS class named by the symbol *name* that inherits from *superclasses* and has state variables specified by *slots*. *superclasses* is a list of superclasses that the new class will inherit from (as in `defclass`). When *superclasses* is `nil`, it behaves as though a superclass of `standard-application-frame` was supplied. *slots* is a list of additional slot specifiers, whose syntax is the same as the slot specifiers 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 is a list of `defclass`-style options, and can include the usual `defclass` options, plus any of the following:

- `:pane` *form*, where *form* specifies the single pane in the application. The default is `nil`, meaning that there is no single pane. This is the simplest way to define a pane hierarchy. The `:pane` option is mutually exclusive with the `:panes` options. See Section 28.2.1 for a

complete description of the `:pane` option.

- `:panes` *form*, where *form* is an alist that specifies names and panes of the application. The default is `nil`, meaning that there are no named panes. The `:panes` and `:pane` options are mutually exclusive. See Section 28.2.1 for a complete description of the `:panes` option.
- `:layouts` *form*, where *form* specifies the layout. The default layout is to lay out all of the named panes in horizontal strips. See Section 28.2.1 for a complete description of the `:layouts` option.
- `:command-table` *name-and-options*, where *name-and-options* is a list consisting of the name of the application frame'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 frame.
- `:command-definer` *value*, where *value* 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` *form* is used to specify what commands will appear in a "menu bar". It typically specifies the top-level commands of the application. *form* is either `nil`, meaning there is no menu bar; `t`, meaning that the menu from frame's command table (from the `:command-table` option) should be used; a symbol that names a command table, meaning that that command table's menu should be used; or a list, which is interpreted the same way the `:menu` option to `define-command-table` is interpreted. The default is `t`.
- `:disabled-commands` *commands*, where *commands* is a list of command names that are initially disabled in the application frame. The set of enabled and disabled commands can be modified via `(setf command-enabled)`.
- `:top-level` *form*, where *form* is a list whose first element is the name of a function to be called to execute the top-level loop. The function must 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`.
- `:icon` *pixmap* specifies a pixmap to be displayed by the host's window manager when the frame is iconified.
- `:geometry` *plist*, where *plist* is a property list containing the default values for the `:left`, `:top`, `:right`, `:bottom`, `:width`, and `:height` options to `make-application-frame`.

The *name*, *superclasses*, and *slots* arguments are not evaluated. The values of each of the options are evaluated.

⇒ `make-application-frame` *frame-name* `&rest` *options* `&key` *pretty-name frame-manager enable state left top right bottom width height save-under frame-class* `&allow-other-keys` [*Function*]

Makes an instance of the application frame of type *frame-class*. If *frame-class* is not supplied, it defaults to *frame-name*.

The size options *left*, *top*, *right*, *bottom*, *width*, and *height* can be used to specify the initial size of the frame. If they are unsupplied and `:geometry` was supplied to `define-application-frame`, then these arguments default from the specified geometry.

options are passed as additional arguments to `make-instance`, after the *pretty-name*, *frame-manager*, *enable*, *state*, *save-under*, *frame-class*, and size options have been removed.

If *save-under* is *true*, then the sheets used to implement the user interface of the frame will have the "save under" property, if the host window system supports it.

If *frame-manager* is provided, then the frame is adopted by the specified frame manager. If the frame is adopted and either of *enable* or *state* are provided, the frame is pushed into the given state.

Once a frame has been create, `run-frame-top-level` can be called to make the frame visible and run its top-level function.

⇒ `*application-frame*` [*Variable*]

The current application frame. The global value is CLIM's default application, which serves only as a repository for whatever internal state is needed by CLIM to operate properly. This variable is typically used in the bodies of command to gain access to the state variables of the application frame, usually in conjunction with `with-slots` or `slot-value`.

⇒ `with-application-frame` *(frame)* `&body` *body* [*Macro*]

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. *body* may have zero or more declarations as its first forms.

⇒ `map-over-frames` *function* `&key` *port frame-manager* [*Function*]

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.

`map-over-frames` returns `nil`.

⇒ `destroy-frame` *frame* [*Generic Function*]

Destroys the application frame *frame*.

⇒ `raise-frame` *frame* [*Generic Function*]

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` *frame* [*Generic Function*]

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.

28.2.1 Specifying the Panes of a Frame

The panes of a frame can be specified in one of two different ways. If the frame has a single layout and no need of named panes, then the `:pane` option can be used. Otherwise if named panes or multiple layouts are required, the `:panes` and `:layouts` options can be used. Note that the `:pane` option is mutually exclusive with `:panes` and `:layouts`. It is meaningful to define frames that have no panes at all; the frame will simply serve as a repository for state and commands.

Panes and gadgets are discussed in detail in Chapter 29 and Chapter 30.

The value of the `:pane` option is a form that is used to create a single (albeit arbitrarily complex) pane. For example:

```
(vertically ()
  (tabling ()
    ((horizontally ()
       (make-pane 'toggle-button)
       (make-pane 'toggle-button)
       (make-pane 'toggle-button))
     (make-pane 'text-field))
    ((make-pane 'push-button :label "a button")
     (make-pane 'slider)))
  (scrolling ()
    (make-pane 'application-pane
               :display-function 'a-display-function))
  (scrolling ()
    (make-pane 'interactor-pane)))
```

If the `:pane` option is not used, a set of named panes can be specified with the `:panes` option. Optionally, `:layouts` can also be used to describe different layouts of the set of panes.

The value of the `:panes` option is a list, each entry of which is of the form *(name . body)*. *name* is a symbol that names the pane, and *body* specifies how to create the pane. *body* is either a list containing a single element that is itself a list, or a list consisting of a symbol followed by zero or more keyword-value pairs. In the first case, the *body* is a form exactly like the form used in the `:pane` option. In the second case, *body* is a *pane abbreviation* where the initial symbol names the type of pane, and the keyword-value pairs are pane options. For gadgets, the pane type is the class name of the abstract gadget (for example, `slider` or `push-button`). For CLIM stream panes, the following abbreviations are defined:

- `:interactor`—a pane of type `interactor-pane`.

- `:application`—a pane of type `application-pane`.
- `:command-menu`—a pane of type `command-menu-pane`.
- `:pointer-documentation`—a pane suitable for displaying pointer documentation, if the host window system does not provide this.
- `:title`—a pane suitable for displaying the title of the application, if the host window system does not provide this.
- `:accept-values`—a pane that can hold a "modeless" `accepting-values` dialog.

See Chapter 29 and Chapter 30 for more information on the individual pane and gadget classes, and the options they support.

An example of the use of `:panes` is:

```
(:panes
  (buttons (horizontally ()
             (make-pane 'push-button :label "Press me")
             (make-pane 'push-button :label "Squeeze me")))
  (toggle toggle-button
          :label "Toggle me")
  (interactor :interactor
              :width 300 :height 300)
  (application :application
               :display-function 'another-display-function
               :incremental-redisplay t))
```

The value of the `:layouts` option is a list, each entry of which is of the form *(name layout)*. *name* is a symbol that names the layout, and *layout* specifies the layout. *layout* is a form like the form used in the `:pane` option, with the extension to the syntax such that the name of a named pane can be used wherever a pane may appear. (This will typically be implemented by using `symbol-macrolet` for each of the named panes.) For example, assuming a frame that uses the `:panes` example above, the following layouts could be specified:

```
(:layouts
  (default
    (vertically ()
      button toggle
      (scrolling () application)
      interactor))
  (alternate
    (vertically ()
      (scrolling () application)
      (scrolling () interactor)
      (horizontally ()
        button toggle))))
```

The syntax for `:layouts` can be concisely expressed as:

`:layouts` (*layout-name layout-panes*)

layout-name is a symbol.

layout-panes is *layout-panes1* or (*size-spec layout-panes1*).

layout-panes1 is a *pane-name*, or a *layout-macro-form*, or *layout-code*.

layout-code is lisp code that generates a pane, which may include the name of a named pane.

size-spec is a rational number less than 1, or `:fill`, or `:compute`.

layout-macro-form is (*layout-macro-name* (*options*) . *body*).

layout-macro-name is one of the layout macros, such as `outlining`, `spacing`, `labelling`, `vertically`, `horizontally`, or `tabling`.

28.3 Application Frame Functions

The generic functions described in this section are the functions that can be used to read or modify the attributes of a frame. All classes that inherit from `application-frame` must inherit or implement methods for all of these functions.

⇒ `frame-name` *frame* [*Generic Function*]

Returns the name of the *frame frame*, which is a symbol.

⇒ `frame-pretty-name` *frame* [*Generic Function*]

Returns the "pretty name" of the *frame frame*, which is a string.

⇒ `(setf frame-pretty-name)` *name frame* [*Generic Function*]

Sets the pretty name of the *frame frame* to *name*, which must be a string. Changing the pretty name of a frame notifies its frame manager, which in turn may change some aspects of the appearance of the frame, such as its title bar.

⇒ `frame-command-table` *frame* [*Generic Function*]

Returns the command table for the *frame frame*.

⇒ `(setf frame-command-table)` *command-table frame* [*Generic Function*]

Sets the command table for the *frame frame* to *command-table*. Changing the frame's command table will redisplay the command menus (or menu bar) as needed. *command-table* is a *command table designator*.

⇒ `frame-standard-output` *frame* [*Generic Function*]

Returns the stream that will be used for `*standard-output*` for the *frame* *frame*. The default method (on `standard-application-frame`) returns the first named pane of type `application-pane` that is visible in the current layout; if there is no such pane, it returns the first pane of type `interactor-pane` that is exposed in the current layout.

⇒ `frame-standard-input` *frame* [*Generic Function*]

Returns the stream that will be used for `*standard-input*` for the *frame* *frame*. The default method (on `standard-application-frame`) returns the first named pane of type `interactor-pane` that is visible in the current layout; if there is no such pane, the value returned by frame-standard-output is used.

⇒ `frame-query-io` *frame* [*Generic Function*]

Returns the stream that will be used for `*query-io*` for the *frame* *frame*. The default method (on `standard-application-frame`) returns the value returned by `frame-standard-input`; if that is `nil`, it returns the value returned by `frame-standard-output`.

⇒ `frame-error-output` *frame* [*Generic Function*]

Returns the stream that will be used for `*error-output*` for the *frame* *frame*. The default method (on `standard-application-frame`) returns the same value as `frame-standard-output`.

⇒ `*pointer-documentation-output*` [*Variable*]

This will be bound either to `nil` or to a stream on which pointer documentation will be displayed.

⇒ `frame-pointer-documentation-output` *frame* [*Generic Function*]

Returns the stream that will be used for `*pointer-documentation-output*` for the *frame* *frame*. The default method (on `standard-application-frame`) returns the first pane of type `pointer-documentation-pane`. If this returns `nil`, no pointer documentation will be generated for this frame.

⇒ `frame-calling-frame` *frame* [*Generic Function*]

Returns the application frame that invoked the *frame* *frame*.

⇒ `frame-parent` *frame* [*Generic Function*]

Returns the object that acts as the parent for the *frame* *frame*. This often, but not always, returns the same value as `frame-manager`.

⇒ `frame-panes` *frame* [*Generic Function*]

Returns the pane that is the top-level pane in the current layout of the *frame* *frame*'s named panes. This will typically be some sort of a layout pane.

⇒ `frame-top-level-sheet` *frame* [*Generic Function*]

Returns the sheet that is the top-level sheet for the *frame* *frame*. This is the sheet that has as

its descendents all of the panes of *frame*.

The value returned by `frame-panes` will be a descendents of the value of `frame-top-level-sheet`.

⇒ `frame-current-panes` *frame* [*Generic Function*]

Returns a list of those named panes in the *frame frame*'s current layout. If there are no named panes (for example, the `:pane` option was used), only the single, top level pane is returned. This function returns objects that reveal CLIM's internal state; do not modify those objects.

⇒ `get-frame-pane` *frame pane-name* [*Generic Function*]

Returns the named CLIM stream pane in the *frame frame* whose name is *pane-name*.

⇒ `find-pane-named` *frame pane-name* [*Generic Function*]

Returns the pane in the *frame frame* whose name is *pane-name*. This can return any type of pane, not just CLIM stream panes.

⇒ `frame-current-layout` *frame* [*Generic Function*]

Returns the current layout for the *frame frame*. The layout is named by a symbol.

⇒ `(setf frame-current-layout)` *layout frame* [*Generic Function*]

Sets the layout of the *frame frame* to be the new layout specified by *new-layout*. *layout* must be a symbol that names one of the possible layouts.

Changing the layout of the frame must recompute what panes are used for the bindings of the standard stream variables (such as `*standard-input*`). Some implementations of CLIM may cause the application to "throw" all the way back to `run-frame-top-level` in order to do this. After the new layout has been computed, the contents of each of the panes must be displayed to the degree necessary to ensure that all output is visible.

⇒ `frame-all-layouts` *frame* [*Generic Function*]

Returns a list of the names of all of the possible layouts for the frame.

⇒ `layout-frame` *frame* `&optional` *width height* [*Generic Function*]

Resizes the frame and lays out the current pane hierarchy using the layout specified by `frame-current-layout`, according to the layout protocol. The basics of the layout protocols are described in Section 29.3.4. This function is automatically invoked on a frame when it is adopted, after its pane hierarchy has been generated.

If *width* and *height* are provided, then this function resizes the frame to the specified size. It is an error to provide just *width*.

If no optional arguments are provided, this function resizes the frame to the preferred size of the top-level pane as determined by the space composition pass of the layout protocol.

In either case, after the frame is resized, the space allocation pass of the layout protocol is invoked on the top-level pane.

⇒ `frame-exit` [*Condition*]

The condition that is signalled when `frame-exit` is called. This condition will handle the `:frame` initarg, which is used to supply the frame that is being exited from.

⇒ `frame-exit-frame` *condition* [*Generic Function*]

Returns the frame that is being exited from associated with the `frame-exit` condition.

⇒ `frame-exit` *frame* [*Generic Function*]

Exits from the *frame frame*. The default method (on `standard-application-frame`) signals a `frame-exit` condition, supplying *frame* as the `:frame` initarg.

⇒ `pane-needs-redisplay` *pane* [*Generic Function*]

Returns two values, the first indicating whether the *pane pane* needs to be redisplayed, and the second indicating whether the pane needs to be cleared before being redisplayed. The first value is *true* when the pane is to be redisplayed. The second value is *true* when the pane is to be cleared.

⇒ `(setf pane-needs-redisplay)` *value pane* [*Generic Function*]

Indicates that the *pane pane* should (or should not) be redisplayed the next time the owning frame executes the redisplay part of its command loop.

When *value* is `nil`, the pane will not require redisplay. When *value* is `t`, the pane will be cleared and redisplayed exactly once. When *value* is `:command-loop`, the pane will be cleared and redisplayed in each successive pass through the command loop. When *value* is `:no-clear`, the pane will be redisplayed exactly once without clearing it.

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

Causes the pane *pane* within the *frame frame* to be redisplayed immediately. *pane* is either a pane or the name of a named pane. When the boolean *force-p* is *true*, the maximum level of redisplay is forced (that is, the pane is displayed "from scratch").

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

`redisplay-frame-panes` causes all of the panes in the *frame frame* to be redisplayed immediately by calling `redisplay-frame-pane` on each of the panes in *frame* that are visible in the current layout. When the boolean *force-p* is *true*, the maximum level of redisplay is forced (that is, the pane is displayed "from scratch").

⇒ `frame-replay` *frame stream* `&optional` *region* [*Generic Function*]

Replays the contents of *stream* in the *frame frame* within the region specified by the region *region*, which defaults to viewport of *stream*.

⇒ `notify-user` *frame message* `&key` *associated-window title documentation exit-boxes name style text-style* [*Generic Function*]

Notifies the user of some event on behalf of the *frame frame*. *message* is a message string. This function provides a look and feel independent way for applications to communicate messages to the user.

associated-window is the window with which the notification will be associated, as it is for `menu-choose`. *title* is a title string to include in the notification. *text-style* is the text style in which to display the notification. *exit-boxes* is as for *accepting-values*; it indicates what sort of exit boxes should appear in the notification. *style* is the style in which to display the notification, and is implementation-dependent.

⇒ `frame-properties` *frame property* [*Generic Function*]
⇒ `(setf frame-properties)` *value frame property* [*Generic Function*]

Frame properties can be used to associate frame specific data with frames without adding additional slots to the frame's class. CLIM may use frame properties internally to store information for its own purposes.

28.3.1 Interface with Presentation Types

This section describes the functions that connect application frames to the presentation type system. All classes that inherit from `application-frame` must inherit or implement methods for all of these functions.

⇒ `frame-maintain-presentation-histories` *frame* [*Generic Function*]

Returns *true* if the *frame frame* maintains histories for its presentations, otherwise returns *false*. The default method (on `standard-application-frame`) returns *true* if and only if the frame has at least one interactor pane.

⇒ `frame-find-innermost-applicable-presentation` *frame input-context stream x y* `&key` *event* [*Generic Function*]

Locates and returns the innermost applicable presentation on the window *stream* whose sensitivity region contains the point (x, y), on behalf of the *frame frame* in the input context *input-context*. *event* defaults to `nil`, and is as for find-innermost-applicable-presentation

The default method (on `standard-application-frame`) will simply call `find-innermost-applicable-presentation`.

⇒ `frame-input-context-button-press-handler` *frame stream button-press-event* [*Generic Function*]

This function is responsible for handling user pointer events on behalf of the *frame frame* in the input context `*input-context*`. *stream* is the window on which *button-press-event* took place.

The default implementation (on `standard-application-frame`) unhighlights any highlighted

presentations, finds the applicable presentation by calling `frame-find-innermost-applicable-presentation-at-position`, and then calls `throw-highlighted-presentation` to execute the translator on that presentation that corresponds to the user's gesture.

If `frame-input-context-button-press-handler` is called when the pointer is not over any applicable presentation, `throw-highlighted-presentation` must be called with a presentation of `*null-presentation*`.

⇒ `frame-document-highlighted-presentation` *frame presentation input-context window x y stream* [*Generic Function*]

This function is responsible for producing pointer documentation on behalf of the *frame frame* in the input context *input-context* on the window *window* at the point (x, y). The documentation is displayed on the *stream stream*.

The default method (on `standard-application-frame`) should produce documentation that corresponds to calling `document-presentation-translator` on all of the applicable translators in the input context *input-context*. *presentation*, *window*, *x*, *y*, and *stream* are as for `document-presentation-translator`.

Typically pointer documentation will consist of a brief description of each translator that is applicable to the specified presentation in the specified input context given the current modifier state for the window. For example, the following documentation might be produced when the pointer is pointing to a Lisp expression when the input context is `form`:

```
Left: '(1 2 3); Middle: (DESCRIBE '(1 2 3)); Right: Menu
```

⇒ `frame-drag-and-drop-feedback` *frame presentation stream initial-x initial-y new-x new-y state* [*Generic Function*]

The default feedback function for translators defined by `define-drag-and-drop-translator`, which provides visual feedback during the dragging phase of such translators on behalf of the *frame frame*. *presentation* is the presentation being dragged on the stream *stream*. The pointing device was initially at the position specified by *initial-x* and *initial-y*, and is at the position specified by *new-x* and *new-y* when `frame-drag-and-drop-feedback` is invoked. (Both positions are supplied for "rubber-banding", if that is the sort of desired feedback.) *state* will be either `:highlight`, meaning that the feedback should be drawn, or `:unhighlight`, meaning that the feedback should be erased.

⇒ `frame-drag-and-drop-highlighting` *frame presentation stream state* [*Generic Function*]

The default highlighting function for translators defined by `define-drag-and-drop-translator`, which is invoked when a "to object" should be highlighted during the dragging phase of such translators on behalf of the *frame frame*. *presentation* is the presentation over which the pointing device is located on the stream *stream*. *state* will be either `:highlight`, meaning that the highlighting for the presentation should be drawn, or `:unhighlight`, meaning that the highlighting should be erased.

28.4 The Generic Command Loop

The default application command loop provided by CLIM performs the following steps:

1. Prompts the user for input.
2. Reads a command. Each application frame has a command table that contains those commands that the author of the application wishes to allow the user to invoke at a given time. Since commands may be read in any number of ways, the generic command loop enforces no particular interface style.
3. Executes the command. The definition of each command may refer to (and update) the state variables of the frame, to which `*application-frame*` will be bound.
4. Runs the display function for each pane in the frame as necessary. The display function may refer to the frame's state variables. Display functions are usually written by the application writer, although certain display functions are supplied by CLIM itself. Note that an application frame is free to have no panes.

Major issue: *RWK has a reasonable proposal for breaking down command loops into their component pieces. It should be integrated here. — SWM*

All classes that inherit from `application-frame` must inherit or implement methods for all of the following functions.

⇒ `run-frame-top-level` *frame* `&key &allow-other-keys` [*Generic Function*]

Runs the top-level function for the *frame frame*. The default method on `application-frame` simply invokes the top-level function for the frame (which defaults to `default-frame-top-level`).

⇒ `run-frame-top-level` *(frame* `application-frame`*)* `&key` [*:Around Method*]

The `:around` method of `run-frame-top-level` on the `application-frame` class is responsible for establish the initial dynamic bindings for the application, including (but not limited to) binding `*application-frame*` to *frame*, binding `*input-context*` to `nil`, resetting the delimiter and activation gestures, and binding `*input-wait-test*`, `*input-wait-handler*`, and `*pointer-button-press-handler*` to `nil`.

⇒ `default-frame-top-level` *frame* `&key` *command-parser command-unparser partial-command-parser prompt* [*Generic Function*]

The default top-level function for application frames. This function implements a "read-eval-print" loop that displays a prompt, calls `read-frame-command`, then calls `execute-frame-command`, and finally redisplays all of the panes that need to be redisplayed.

`default-frame-top-level` will also establish a simple restart for `abort`, and bind the standard stream variables as follows. `*standard-input*` will be bound to the value returned by `frame-`

`standard-input`. `*standard-output*` will be bound to the value returned by `frame-standard-output`. `*query-io*` will be bound to the value returned by `frame-query-io`. `*error-output*` will be bound to the value returned by `frame-error-output`. It is unspecified what `*terminal-io*`, `*debug-io*`, and `*trace-output*` will be bound to.

prompt is either a string to use as the prompt (defaulting to `"Command: "`), or a function of two arguments, a stream and the frame.

command-parser, *command-unparser*, and *partial-command-parser* are the same as for `read-command`. *command-parser* defaults to `command-line-command-parser` if there is an interactor, otherwise it defaults to `menu-only-command-parser`. *command-unparser* defaults to `command-line-command-unparser`. *partial-command-parser* defaults to `command-line-read-remaining-arguments-for-partial-command` if there is an interactor, otherwise it defaults to `menu-only-read-remaining-arguments-for-partial-command`. `default-frame-top-level` binds `*command-parser*`, `*command-unparser*`, and `*partial-command-parser*` to the values of *command-parser*, *command-unparser*, and *partial-command-parser*.

⇒ `read-frame-command` *frame* `&key` *(stream* `*standard-input*`*)* [*Generic Function*]

Reads a command from the stream *stream* on behalf of the *frame frame*. The returned value is a command object.

The default method (on `standard-application-frame`) for `read-frame-command` simply calls `read-command`, supplying *frame*'s current command table as the command table.

⇒ `execute-frame-command` *frame command* [*Generic Function*]

Executes the command *command* on behalf of the *frame frame*. *command* is a command object, that is, a cons of a command name and a list of the command's arguments.

The default method (on `standard-application-frame`) for `execute-frame-command` simply applies the `command-name` of *command* to `command-arguments` of *command*.

If process that `execute-frame-command` is invoked in is not the same process the one *frame* is running in, CLIM may need to make special provisions in order for the command to be correctly executed, since as queueing up a special "command event" in *frame*'s event queue. The exact details of how this should work is left unspecified.

⇒ `command-enabled` *command-name frame* [*Generic Function*]

Returns *true* if the command named by *command-name* is presently enabled in the *frame frame*, otherwise returns *false*. If *command-name* is not accessible to the command table being used by *frame*, `command-enabled` returns *false*.

Whether or not a particular command is currently enabled is stored independently for each instance of an application frame; this status can vary between frames that share a single command table.

⇒ `(setf command-enabled)` *enabled command-name frame* [*Generic Function*]

If *enabled* is *false*, this disables the use of the command named by *command-name* while in the *frame* *frame*. Otherwise if *enabled* is *true*, the use of the command is enabled. After the command has been enabled (or disabled), `note-command-enabled` (or `note-command-disabled`) is invoked on the frame manager and the frame in order to update the appearance of the interface, for example, "graying out" a disabled command.

If *command-name* is not accessible to the command table being used by *frame*, using `setf` on `command-enabled` does nothing.

⇒ `display-command-menu` *frame stream* `&key` *command-table initial-spacing row-wise max-width max-height n-rows n-columns (cell-align-x* `:left`*) (cell-align-y* `:top`*)* [*Generic Function*]

Displays the menu associated with the specified command table on *stream* by calling `display-command-table-menu`. If *command-table* is not supplied, it defaults to `(frame-command-table` *stream*`)`. This function is generally used as the display function for panes that contain command menus.

initial-spacing, *max-width*, *max-height*, *n-rows*, *n-columns*, *row-wise*, *cell-align-x*, and *cell-align-y* are as for `formatting-item-list`.

28.5 Frame Managers

Frames may be *adopted* by a frame manager, which involves invoking a protocol for generating the pane hierarchy of the frame. This protocol provides for selecting pane types for abstract gadget panes based on the style requirements imposed by the frame manager. That is, the frame manager is responsible for the "look and feel" of a frame.

After a frame is adopted it can be in any of the three following states: *enabled*, *disabled*, or *shrunk*. An enabled frame is visible unless it is occluded by other frames or the user is browsing outside of the portion of the frame manager's space that the frame occupies. A shrunken frame provides a cue or handle for the frame, but generally will not show the entire contents of the frame. For example, the frame may be iconified or an item for the frame may be placed in a special suspended frame menu. A disabled frame is not visible, nor is there any user accessible handle for enabling the frame.

Frames may also be *disowned*, which involves releasing the frame's panes as well as all associated foreign resources.

⇒ `frame-manager` [*Protocol Class*]

The protocol class that corresponds to a frame manager. If you want to create a new class that behaves like a frame manager, it should be a subclass of `frame-manager`. All instantiable subclasses of `frame-manager` must obey the frame manager protocol.

There are no advertised standard frame manager classes. Each port will implement one or more frame managers that correspond to the look and feel for the port.

⇒ `frame-mananger-p` *object* [*Protocol Predicate*]

Returns *true* if *object* is a *frame manager*, otherwise returns *false*.

28.5.1 Finding Frame Managers

Most frames need only deal directly with frame managers to the extent that they need to find a frame manager into which they can insert themselves. Since frames will usually be invoked by some user action that is handled by some frame manager, finding an appropriate frame manager is usually straightforward.

Some frames will support the embedding of other frames within themselves. Such frames would not only use frames but also act as frame managers, so that other frames could insert frames. In this case, the embedded frames are mostly unaware that they are nested within other frames, but only know that they are controlled by a particular frame manager.

Minor issue: *How does one write a frame that supports an embedded frame, such as an editor frame within a documentation-writing frame? — SWM*

The `find-frame-manager` function provides a flexible means for locating an frame manager to adopt an application's frames into. There are a variety of ways that the user or the application can influence where an application's frame is adopted.

An application can establish an application default frame manager using `with-frame-manager`. A frame's top-level loop automatically establishes the frame's frame manager.

The programmer or user can influence what frame manager is found by setting `*default-frame-manager*` or `*default-server-path*`.

Each frame manager is associated with one specific port. However, a single port may have multiple frame managers managing various frames associated with the port.

⇒ `find-frame-manager &rest` *options* `&key` *port* `&allow-other-keys` [*Function*]

Finds an appropriate frame manager that conforms to the options, including the *port* argument. Furthermore, CLIM applications may set up dynamic contexts that affect what `find-frame-manager` will return.

port defaults to the value returned by `find-port` applied to the remaining options.

A frame manager is found using the following rules in the order listed:

1. If a current frame manager has been established via an invocation of `with-frame-manager`, as is the case within a frame's top-level, and that frame manager conforms to the options, it is returned. The exact definition of "conforming to the options" varies from one port to another, but it may include such things as matching the console number, color or resolution properties, and so forth. If the options are empty, then any frame manager will conform.

2. If `*default-frame-manager*` is bound to a currently active frame manager and it conforms to the options, it is returned.

3. If *port* is `nil`, a port is found and an appropriate frame manager is constructed using `*default-server-path*`.

⇒ `*default-frame-manager*` [*Variable*]

This variable provides a convenient point for allowing a programmer or user to override what frame manager type would normally be selected. Most users will not set this variable since they can set `*default-server-path*` to indicate which host window system they want to use and are willing to use whatever frame manager is the default for the particular port. However, some users may want to use a frame manager that isn't the typical frame manager. For example, a user may want to use both an OpenLook frame manager and a Motif frame manager on a single port.

⇒ `with-frame-manager` *(frame-manager)* `&body` *body* [*Macro*]

Generates a dynamic context that causes all calls to `find-frame-manager` to return *frame-manager* if the *where* argument passed to it conforms to *frame-manager*. Nested calls to `with-frame-manager` will shadow outer contexts. *body* may have zero or more declarations as its first forms.

28.5.2 Frame Manager Operations

⇒ `frame-manager` *frame* [*Generic Function*]

Returns the current frame manager of *frame* if it is adopted, otherwise returns `nil`.

⇒ `(setf frame-manager)` *frame-manager frame* [*Generic Function*]

Changes the frame manager of *frame* to *frame-manager*. In effect, the frame is disowned from its old frame manager and is adopted into the new frame manager. Transferring a frame preserves its `frame-state`, for example, if the frame was previously enabled it will be enabled in the new frame manager.

⇒ `frame-manager-frames` *frame-manager* [*Generic Function*]

Returns a list of all of the frames being managed by *frame-manager*. This function returns objects that reveal CLIM's internal state; do not modify those objects.

⇒ `adopt-frame` *frame-manager frame* [*Generic Function*]
⇒ `disown-frame` *frame-manager frame* [*Generic Function*]

These functions insert or remove a frame from a frame manager's control. These functions allow a frame manager to allocate and deallocate resources associated with a frame. For example, removing a frame from a frame manager that is talking to a remote server allows it to release all remote resources used by the frame.

⇒ `port` *(frame* `standard-application-frame`*)* [*Method*]

If *frame* has been adopted by a frame manager, this returns the port with which *frame* is associated. Otherwise it returns `nil`.

⇒ `port` *(frame-manager* `standard-frame-manager`*)* [*Method*]

Returns the port with which *frame-manager* is associated.

⇒ `frame-state` *frame* [*Generic Function*]

Returns one of `:disowned`, `:enabled`, `:disabled`, or `:shrunk`, indicating the current state of frame.

⇒ `enable-frame` *frame* [*Generic Function*]
⇒ `disable-frame` *frame* [*Generic Function*]
⇒ `shrink-frame` *frame* [*Generic Function*]

These functions force a frame into the enabled, disabled, or shrunken (iconified) states. A frame in the enabled state may be visible if it is not occluded or placed out of the user's focus of attention. A disabled frame is never visible. A shrunk frame is accessible to the user for re-enabling, but may be represented in some abbreviated form, such as an icon or a menu item.

These functions call the notification functions describe below to notify the frame manager that the state of the frame changed.

⇒ `note-frame-enabled` *frame-manager frame* [*Generic Function*]
⇒ `note-frame-disabled` *frame-manager frame* [*Generic Function*]
⇒ `note-frame-iconified` *frame-manager frame* [*Generic Function*]
⇒ `note-frame-deiconified` *frame-manager frame* [*Generic Function*]

Notifies the frame manager *frame-manager* that the frame *frame* has changed its state to the enabled, disabled, iconified, or deiconified state, respectively.

⇒ `note-command-enabled` *frame-manager frame command-name* [*Generic Function*]
⇒ `note-command-disabled` *frame-manager frame command-name* [*Generic Function*]

Notifies the frame manager *frame-manager* that the command named by *command-name* has been enabled or disabled (respectively) in the frame *frame*. The frame manager can update the appearance of the user interface as appropriate, for instance, by "graying out" a newly disabled command from a command menu or menu bar.

⇒ `frame-manager-notify-user` *framem message-string* `&key` *frame associated-window title documentation exit-boxes name style text-style* [*Generic Function*]

This is the generic function used by `notify-user`. The arguments are as for `notify-user`. The default method on `standard-frame-manager` will display a dialog or an alert box that contains the message and has exit boxes that allow the user to dismiss the notification.

⇒ `generate-panes` *frame-manager frame* [*Generic Function*]

This function is invoked by a standard method of `adopt-frame`. `define-application-frame` automatically supplies a `generate-panes` method if either the `:pane` or `:panes` option is used in the `define-application-frame`.

It is the responsibility of this method to call `setf` on `frame-panes` on the frame in order to set the current

⇒ `find-pane-for-frame` *frame-manager frame* [*Generic Function*]

This function is invoked by a standard method of `adopt-frame`. It must return the root pane of the frame's layout. It is the responsibility of the frame implementor to provide a method that constructs the frame's top-level pane. `define-application-frame` automatically supplies a a method for this function if either the `:pane` or `:panes` option is used in the `define-application-frame`.

28.5.3 Frame Manager Settings

CLIM provides frame manager settings in order to allow a frame to communicate information to its frame manager.

⇒ `(setf client-setting)` *value frame setting* [*Generic Function*]

Sets the setting *setting* to *value* for the frame *frame*.

⇒ `reset-frame` *frame* `&rest` *client-settings* [*Generic Function*]

Resets the settings of frame. `reset-frame` invokes a protocol that forces the frame manager to notice that the settings have changed, where the setf generic function just updates the frame data. For example, the width and height can be reset to force resizing of the window.

28.6 Examples of Applications

The following is an example that outlines a simple 4-by-4 sliding piece puzzle:

```
(define-application-frame puzzle ()
    ((puzzle-array :initform (make-array '(4 4))))
  (:menu-bar t)
  (:panes
    (display
      (outlining ()
        (make-pane 'application-pane
                   :text-cursor nil
                   :width :compute
                   :height :compute
                   :incremental-redisplay T
                   :display-function 'draw-puzzle))))
  (:layouts
    (:default display)))

(defmethod run-frame-top-level :before ((puzzle puzzle))
  ;; Initialize the puzzle
  ...)

(define-presentation-type puzzle-cell ()
  :inherit-from '(integer 1 15))

(defmethod draw-puzzle ((puzzle puzzle) stream &key max-width max-height)
  (declare (ignore max-width max-height))
  ;; Draw the puzzle, presenting each cell as a PUZZLE-CELL
  ...)

(define-puzzle-command com-move-cell
    ((cell 'puzzle-cell :gesture :select))
  ;; Move the selected cell to the adjacent open cell,
  ;; if there is one
  ...)

(define-puzzle-command (com-scramble :menu t)
    ()
  ;; Scramble the pieces of the puzzle
  ...)

(define-puzzle-command (com-exit-puzzle :menu "Exit")
    ()
  (frame-exit *application-frame*))

(defun puzzle ()
  (let ((puzzle
```

```
          (make-application-frame 'puzzle
            :width 80 :height 80)))
    (run-frame-top-level puzzle)))
```

The following is an application frame with two layouts:

```
(define-application-frame test-frame () ()
  (:panes
    (a (horizontally ()
         (make-pane 'push-button :label "Press me")
         (make-pane 'push-button :label "Squeeze me")))
    (b toggle-button)
    (c slider)
    (d text-field)
    (e :interactor-pane
       :width 300 :max-width +fill+
       :height 300 :max-height +fill+))
  (:layouts
    (default
      (vertically ()
        a b c (scrolling () e)))
    (other
      (vertically ()
        a (scrolling () e) b d))))

(define-test-frame-command (com-switch :name t :menu t)
    ()
  (setf (frame-current-layout *application-frame*)
        (ecase (frame-current-layout *application-frame*)
          (default other)
          (other default))))

(let ((test-frame
        (make-application-frame 'test-frame)))
  (run-frame-top-level test-frame))
```

Chapter 29

Panes

29.1 Overview of Panes

CLIM panes are 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 some 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 initargs, 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.

CLIM provides a general mechanism for automatically selecting the particular implementation of an abstract pane selected by an application based on the current frame manager. The application programmer can override the selection mechanism by using the name of a specific pane implementation in place of the abstract pane name when specifying the application frame's layout. By convention, the name of the basic, portable implementation of an abstract pane class can be determined by adding the suffix "`-pane`" to the name of the abstract pane class.

29.2 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.

⇒ `pane` [*Protocol 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. If you want to create a new class that behaves like a pane, it should be a subclass of `pane`. All instantiable subclasses of `pane` must obey the pane protocol.

All of the subclasses of `pane` are mutable.

⇒ `panep` *object* [*Protocol Predicate*]

Returns *true* if *object* is a *pane*, otherwise returns *false*.

⇒ `basic-pane` [*Class*]

The basic class on which all CLIM panes are built, a subclass of `pane`. This class is an abstract class, intended only to be subclassed, not instantiated.

⇒ `make-pane` *abstract-class-name* `&rest` *initargs* [*Function*]

Selects a class that implements the behavior of the abstract pane *abstract-class-name* and constructs a pane of that class. `make-pane` must be used either within the scope of a call to `with-look-and-feel-realization`, or within the `:pane` or `:panes` options of a `define-application-frame` (which implicitly invokes `with-look-and-feel-realization`).

It is permitted for this function to have lexical scope, and be defined only within the body of `with-look-and-feel-realization`.

⇒ `make-pane-1` *realizer frame abstract-class-name* `&rest` *initargs* [*Generic Function*]

The generic function that is invoked by a call to `make-pane`. The object that realizes the pane, *realizer*, is established by `with-look-and-feel-realization`. Usually, *realizer* is a frame manager, but it could be another object that implements the pane realization protocol. *frame* is the frame for which the pane will be created, and *abstract-class-name* is the type of pane to create.

⇒ `with-look-and-feel-realization` *(realizer frame)* `&body` *forms* [*Macro*]

Establishes a context that installs *realizer* as the object responsible for realizing panes. All calls to `make-pane` within the context of `with-look-and-feel-realization` result in `make-pane-1` being invoked on *realizer*. This macro can be nested dynamically; inner uses shadow outer uses. *body* may have zero or more declarations as its first forms.

Typically *realizer* is a frame manager, but in some cases *realizer* may be some other object. For example, within the implementation of pane that is uses its own subpanes to achieve its functionality, this form might be used with *realizer* being the pane itself.

It is permitted for the function `make-pane` to have lexical scope only within the body of `with-look-and-feel-realization`.

29.2.1 Pane Initialization Options

The following options must be accepted by all pane classes.

⇒ `:foreground` [*Option*]
⇒ `:background` [*Option*]

These options specify the default foreground and background inks for a pane. These will normally default from window manager resources. If there are no such resources, the defaults are black and white, respectively.

Client code should be cautious about passing values for these two options, since the window

manager's look and feel or the user's preferences should usually determine these values.

⇒ `:text-style` [*Option*]

This option specifies the default text style that should be used for any sort of pane that supports text output. Panes that do not support text output ignore this option.

Client code should be cautious about passing values for this option, since the window manager's look and feel or the user's preferences should usually determine this value.

⇒ `:name` [*Option*]

This option specifies the name of the pane. It defaults to `nil`.

29.2.2 Pane Properties

⇒ `pane-frame` *pane* [*Generic Function*]

Returns the frame that "owns" the pane. `pane-frame` can be invoked 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.

⇒ `pane-name` *pane* [*Generic Function*]

Returns the name of the pane.

⇒ `pane-foreground` *pane* [*Generic Function*]
⇒ `pane-background` *pane* [*Generic Function*]
⇒ `pane-text-style` *pane* [*Generic Function*]

Return the default foreground and background inks and the default text style, respectively, for the pane *pane*. These will be used as the default value for `medium-foreground` and `medium-background` when a medium is grafted to the pane.

29.3 Composite and Layout Panes

This section describes the various composite and layout panes provided by CLIM, and the protocol that the layout panes obey.

The layout panes describe in this section are all composite panes that are responsible for positioning their children according to various layout rules. Layout panes can be selected in the same way as other panes using `make-pane` or `make-instance`. For convenience and readability of application pane layouts, many of these panes also provide a macro that expands into a `make-pane` form, passing a list of the panes created in the body of the macro as the `:contents` argument (described below). For example, you can express a layout of a vertical column of two label panes either as:

```
(make-instance 'vbox-pane
  :contents (list (make-instance 'label-pane :text "One")
                  (make-instance 'label-pane :text "Two")))
```

or as:

```
(vertically ()
  (make-instance 'label-pane :text "One")
  (make-instance 'label-pane :text "Two"))
```

29.3.1 Layout Pane Options

⇒ `:contents` [*Option*]

All of the layout pane classes accept the `:contents` options, which is used to specify the child panes to be laid out.

⇒ `:width` [*Option*]
⇒ `:max-width` [*Option*]
⇒ `:min-width` [*Option*]
⇒ `:height` [*Option*]
⇒ `:max-height` [*Option*]
⇒ `:min-height` [*Option*]

These options control the space requirement paramaters 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 `:min-width` is not supplied, it defaults to the value of the `:width` option. If `:min-height` is not supplied, it defaults to the value of the `:height` option. If `:max-width` is not supplied, it defaults to `+fill+`. If `:max-height` is not supplied, it defaults to `+fill+`. These defaults gives the intuitive behavior that specifying only the width or height of a pane causes it to be allocated at least that much space, and it may be given extra space if there is extra space in the layout. This default behavior can be changed if `:min-width` or `:min-height` are supplied explicitly.

`: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`.)

⇒ `+fill+` [*Constant*]

This constant can be used as a value to any of the relative size options. It indicates that pane's willingness to adjust an arbitrary amount in the specified direction.

⇒ `:align-x` [*Option*]
⇒ `:align-y` [*Option*]

The `:align-x` option is one of `:right`, `:center`, or `:left`. The `:align-y` option is one of `:top`, `:center`, or `:bottom`. These are used to specify how child panes are aligned within the parent pane. These have the same semantics as for `formatting-cell`.

⇒ `:x-spacing` [*Option*]
⇒ `:y-spacing` [*Option*]
⇒ `:spacing` [*Option*]

These spacing options apply to `hbox-pane`, `vbox-pane`, `table-pane`, and `grid-pane`, and indicate the amount of horizontal and vertical spacing (respectively) to leave between the items in boxes or rows and columns in table. The values of these options are specified in the same way as the `:x-spacing` and `:y-spacing` options to `formatting-table`. `:spacing` specifies both the x and y spacing simultaneously.

29.3.2 Layout Pane Classes

⇒ `hbox-pane` [*Layout Pane*]
⇒ `horizontally` *(*`&rest` *options* `&key` *spacing* `&allow-other-keys` *)* `&body` *contents* [*Macro*]

The `hbox-pane` class lays out all of its child panes horizontally, from left to right. The child panes are separated by the amount of space specified by *spacing*.

The `horizontally` macro serves as a convenient interface for creating an `hbox-pane`.

contents is one or more forms that are 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 percentage 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:

```
(horizontally ()
  (1/3 (make-pane 'label-button-pane))
  (2/3 (make-pane 'label-button-pane)))
```

would create a horizontal stack of two button panes. The first button takes one-third of the space, then second takes two-thirds of the space.

⇒ `vbox-pane` [*Layout Pane*]
⇒ `vertically` *(*`&rest` *options* `&key` *spacing* `&allow-other-keys` *)* `&body` *contents* [*Macro*]

The `vbox-pane` class lays out all of its child panes vertically, from top to bottom. The child panes are separated by the amount of space specified by *spacing*.

The `vertically` macro serves as a convenient interface for creating an `vbox-pane`.

contents is as for `horizontally`.

⇒ `hrack-pane` [*Layout Pane*]
⇒ `vrack-pane` [*Layout Pane*]

Similar to the `hbox-pane` and `vbox-pane` classes, except that these ensure that all children are the same size in the minor dimension. In other words, these panes are used to create stacks of same-sized items, such as menu items.

An `hrack-pane` is created when the `:equalize-height` option to `horizontally` is *true*. A `vrack-pane` is created when the `:equalize-width` option to `vertically` is *true*.

⇒ `table-pane` [*Layout Pane*]
⇒ `tabling` *(*`&rest` *options)* `&body` *contents* [*Macro*]

This pane lays out its child panes in a two-dimensional table arrangement. Each of the table is specified by an extra level of list in *contents*. For example,

```
(tabling ()
  (list
    (make-pane 'label :text "Red")
    (make-pane 'label :text "Green")
    (make-pane 'label :text "Blue"))
  (list
    (make-pane 'label :text "Intensity")
    (make-pane 'label :text "Hue")
    (make-pane 'label :text "Saturation")))
```

⇒ `grid-pane` [*Layout Pane*]

A `grid-pane` is like a `table-pane`, except that each cell is the same size in each of the two dimensions.

⇒ `spacing-pane` [*Layout Pane*]
⇒ `spacing` *(*`&rest` *options* `&key` *thickness* `&allow-other-keys` *)* `&body` *contents* [*Macro*]

This pane reserves some margin space of thickness *thickness* around a single child pane. The space requirement keys that are passed in indicate the requirements for the surrounding space,

not including the requirements of the child.

⇒ `outlined-pane` [*Layout Pane*]
⇒ `outlining` *(*`&rest` *options* `&key` *thickness* `&allow-other-keys` *)* `&body` *contents* [*Macro*]

This layout pane puts a outline of thickness *thickness* around its contents.

The `:background` option can be used to control the ink used to draw the background.

⇒ `restraining-pane` [*Layout Pane*]
⇒ `restraining` *(*`&rest` *options)* `&body` *contents* [*Macro*]

Wraps the contents with a pane that prevents changes to the space requirements for contents from causing relayout of panes outside of the restraining context. In other words, it prevents the size constraints of the child from propagating up beyond this point.

⇒ `bboard-pane` [*Layout Pane*]

A pane that allows its children to be any size and lays them out wherever they want to be (for example, a desktop manager).

⇒ `label-pane` [*Service Pane*]
⇒ `labelling` *(*`&rest` *options* `&key` *label label-alignment* `&allow-other-keys` *)* `&body` *contents* [*Macro*]

Creates a pane that consists of the specified label *label*, which is a string. *label-alignment* may be either `:bottom` or `:top`, which specifies whether the label should appear at the top or the bottom of the labelled pane. The default for *label-alignment* is left unspecified.

29.3.3 Scroller Pane Classes

CLIM defines the following additional pane classes, each having at least one implementation.

⇒ `scroller-pane` [*Service Pane*]
⇒ `scrolling` *(*`&rest` *options)* `&body` *contents* [*Macro*]

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 pane created by the `scrolling` will include a `scroller-pane` that has as children the scroll bars and a *viewport*. The viewport of a pane is the area of the window's drawing plane that is currently visible to the user. The viewport has as its child the specified contents.

⇒ `pane-viewport` *pane* [*Generic Function*]

If the pane *pane* is part of a scroller pane, this returns the viewport pane for *pane*. Otherwise it

returns `nil`.

⇒ `pane-viewport-region` *pane* [*Generic Function*]

If the pane *pane* is part of a scroller pane, this returns the region of the pane's viewport. Otherwise it returns `nil`.

⇒ `pane-scroller` *pane* [*Generic Function*]

If the pane *pane* is part of a scroller pane, this returns the scroller pane itself. Otherwise it returns `nil`.

⇒ `scroll-extent` *pane x y* [*Generic Function*]

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.

x and y are coordinates.

29.3.4 The Layout Protocol

The layout protocol is triggered by `layout-frame`, which is called when a frame is adopted by a frame manager.

CLIM uses a two pass algorithm to lay out a pane hierarchy. In the first pass (called 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.

Minor issue: *It isn't alway possible to allocate the required space. What is the protocol for handling these space allocation failures? Some kind of error should be signalled when the constraints can't be satisfied, which can be handled by the application. Otherwise the panes will fall where they may. The* `define-application-frame` *macro should provide an option that allows programmers to conveniently specify a condition handler. — ILA*

For many types of panes, the application programmer can indicate the space requirements of the pane at creation time by using the space requirement options (described above), as well as by calling the `change-space-requirements` function (described below). 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 mininum size below which the pane cannot be shrunk, and a maxium 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).

⇒ `space-requirement` [*Class*]

The protocol class of all space requirement objects. There are one or more subclasses of `space-requirement` with implementation-dependent names that implement space requirements. The exact names of these classes is explicitly unspecified. If you want to create a new class that behaves like a space requirement, it should be a subclass of `space-requirement`. All instantiable subclasses of `space-requirement` must obey the space requirement protocol.

All of the instantiable space requirement classes provided by CLIM are immutable.

⇒ `make-space-requirement &key` *(width* `0`*) (max-width* `0`*) (min-width* `0`*) (height* `0`*) (max-height* `0`*) (min-height* `0`*)* [*Function*]

Constructs a space requirement object with the given characteristics, `:width`, `:height`, and so on.

⇒ `space-requirement-width` *space-req* [*Generic Function*]
⇒ `space-requirement-min-width` *space-req* [*Generic Function*]
⇒ `space-requirement-max-width` *space-req* [*Generic Function*]
⇒ `space-requirement-height` *space-req* [*Generic Function*]
⇒ `space-requirement-min-height` *space-req* [*Generic Function*]
⇒ `space-requirement-max-height` *space-req* [*Generic Function*]

These functions read the components of the space requirement *space-req*.

⇒ `space-requirement-components` *space-req* [*Generic Function*]

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 sr1 sr2* [*Function*]

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+` *sr1 sr2* [*Function*]

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)
  (space-requirement-combine #'+ sr1 sr2))
```

⇒ `space-requirement+*` *space-req* `&key` *width min-width max-width height min-height max-height* [*Function*]

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 intended to be a more efficient, "spread" version of `space-requirement+`.

⇒ `compose-space` *pane* `&key` *width height* [*Generic Function*]

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` *pane width height* [*Generic Function*]

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` *pane* `&key` *resize-frame* `&rest` *space-req-keys* [*Generic Function*]

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 *true*, then `layout-frame` will be invoked on the frame. If *resize-frame* is *false*, 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`.

⇒ `note-space-requirements-changed` *sheet pane* [*Generic Function*]

This function is invoked whenever *pane*'s space requirements have changed. *sheet* must be the parent of *pane*. Invoking this function essentially means that *compose-space* will be reinvoked on *pane*, then it will reply with a space requirement that is not equal to the reply that was given on the last call to *compose-space*.

This function is automatically invoked by `change-space-requirements` in the cases that `layout-frame` isn't invoked. In the case that `layout-frame` is invoked, it isn't necessary to call `note-space-requirements-changed` since a complete re-layout of the frame will be executed.

⇒ `changing-space-requirements` *(*`&key` *resize-frame layout)* `&body` *body* [*Macro*]

This macro supports batching the invocation of the layout protocol by calls to `change-space-requirements`. Within the body, all calls to `change-space-requirements` change the internal structures of the pane and are recorded. When the body is exited, the layout protocol is invoked appropriately. *body* may have zero or more declarations as its first forms.

29.4 CLIM Stream Panes

In addition to the various layout panes and gadgets, an application usually needs some space to display application-specific output and receive application-specific input from the user. For example, a paint program needs a "canvas" pane on which to display the picture and handle the "mouse strokes". An application frame can use the basic CLIM input and output services in an application-specific way through use of a *CLIM stream pane*.

This section describes the basic CLIM stream pane types. Programmers are free to customize pane behavior by defining subclasses of these pane classes writing methods to change the repaint or event-handling behavior.

29.4.1 CLIM Stream Pane Options

CLIM application frames accept the `:foreground`, `:background`, `:text-style`, and layout pane options. The space requirement options (`:width`, `:height`, and so forth) can also take a size specification of `:compute`, which causes CLIM to run the display function for the pane, and make the pane large enough to hold the output of the display function.

In addition to the above, CLIM stream panes accept the following options:

⇒ `:display-function` [*Option*]

This is used to specify a function to be called in order to display the contents of a CLIM stream pane. CLIM's default top level function, `default-frame-top-level`, function will invoke the pane's display function at the appropriate time (see the `:display-time` option). The value of this option is either the name of a function to invoke, or a cons whose car is the name of a function and whose cdr is additional arguments to the function. The function will be invoked on

the frame, the pane, and the additional function arguments, if any. The default for this option is `nil`.

⇒ `:display-time` [*Option*]

This is used to indicate to CLIM when the pane's display function should be run. If it is `:command-loop`, CLIM will clear the pane and run the display function after each time a frame command is executed. If it is `t`, the pane will be displayed once and not again until `(setf pane-needs-redisplay)` is called on the pane. If it is `nil`, CLIM will never run the display function until it is explicitly requested, either via `pane-needs-redisplay` or `redisplay-frame-pane`. The default for this option varies depending on the type of the pane.

⇒ `:incremental-redisplay` [*Option*]

When *true*, the redisplay function will initially be executed inside of an invocation to `updating-output` and the resulting output record will be saved. Subsequent calls to `redisplay-frame-pane` will simply use `redisplay` to redisplay the pane. The default for this option is `nil`.

⇒ `:text-margin` [*Option*]
⇒ `:vertical-spacing` [*Option*]

These options specify the default text margin (that is, how much space is left around the inside edge of the pane) and vertical spacing (that is, how much space is between each text line) for the pane. The default for `:text-margin` is the width of the window, and the default for `:vertical-spacing` is 2.

⇒ `:end-of-line-action` [*Option*]
⇒ `:end-of-page-action` [*Option*]

These options specify the end-of-line and end-of-page actions to be used for the pane. The default for these options are `:wrap` and `:scroll`, respectively.

⇒ `:output-record` [*Option*]

This option names the output record class to be used for the output history of the pane. The default is `standard-tree-output-history`.

⇒ `:draw` [*Option*]
⇒ `:record` [*Option*]

These options specify whether the pane should initially allow drawing and output recording. The default for both options is `t`.

29.4.2 CLIM Stream Pane Classes

⇒ `clim-stream-pane` [*Service Pane*]

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` [*Service Pane*]

The pane class that is used to implement "interactor" panes. 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` [*Service Pane*]

The pane class that is used to implement "application" panes. 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` [*Service Pane*]

The pane class that is used to implement command menu panes that are not menu bars. 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` [*Service Pane*]

The pane class that is used to implement a title pane. 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` [*Service Pane*]

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`.

29.4.3 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 &rest` *options* `&key` *type label label-alignment scroll-bars borders display-after-commands* `&allow-other-keys` [*Function*]

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.

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`).

The other options may include all of the valid CLIM stream pane options.

⇒ `make-clim-interactor-pane &rest` *options* [*Function*]

Like `make-clim-stream-pane`, except that the type is forced to be `interactor-pane`.

⇒ `make-clim-application-pane &rest` *options* [*Function*]

Like `make-clim-stream-pane`, except that the type is forced to be `application-pane`.

29.4.4 CLIM Stream Pane Functions

The following functions can be called on 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.

⇒ `window-clear` *window* [*Generic Function*]

Clears the entire drawing plane by filling it with the background design of the CLIM stream pane *window*. If *window* has an output history, that is cleared as well. The text cursor position of *window*, if there is one, is reset to the upper left corner.

⇒ `window-refresh` *window* [*Generic Function*]

Clears the visible part of the drawing plane of the CLIM stream pane *window*, and then if the window stream is an output recording stream, the output records in the visible part of the window are replayed.

⇒ `window-viewport` *window* [*Generic Function*]

Returns the viewport region of the CLIM stream pane *window*. If the window is not scrollable, and hence has no viewport, this will region `sheet-region` of *window*.

The returned region will generally be a `standard-bounding-rectangle`.

⇒ `window-erase-viewport` *window* [*Generic Function*]

Clears the visible part of the drawing plane of the CLIM stream pane *window* by filling it with the background design.

⇒ `window-viewport-position` *window* [*Generic Function*]

Returns two values, the x and y position of the top-left corner of the CLIM stream pane *window*'s viewport. If the window is not scrollable, this will return the two values 0 and 0.

⇒ `(setf* window-viewport-position)` *x y window* [*Generic Function*]

Sets the position of the CLIM stream pane *window*'s viewport to x and y. If the window is not scrollable, this will do nothing.

For CLIM implementations that do not support `setf*`, the "setter" function for this is `window-set-viewport-position`.

29.4.5 Creating a Standalone CLIM Window

The following function can be used to create a standalone window that obeys CLIM's extended input and output stream and output recording protocols.

⇒ `open-window-stream &key` *port left top right bottom width height foreground background text-style (vertical-spacing* `2`*) end-of-line-action end-of-page-action output-record (draw* `t`*) (record* `t`*) (initial-cursor-visibility* `:off`*) text-margin save-under input-buffer (scroll-bars* `:vertical`*) borders label* [*Function*]

Creates and returns a sheet that can be used as a standalone window that obeys CLIM's extended input and output stream and output recording protocols.

The window will be created on the port *port* at the position specified by *left* and *top*, which default to 0. *right*, *bottom*, *width*, and *height* default in such a way that the default width will be 100 and the default height will be 100.

foreground, *background*, and *text-style* are used to specify the window's default foreground and background inks, and text style. *vertical-spacing*, *text-margin*, *end-of-line-action*, *end-of-page-action*, *output-record*, *draw*, and *record* are described in Section 29.4.1.

scroll-bars specifies whether scroll bars should be included in the resulting window. It may be one of *nil*, `:vertical` (the default), `:horizontal`, or `:both`.

borders is a booleans that specifies whether or not the resulting window should have a border drawn around it. The default is `t`. *label* is either `nil` or a string to use as a label for the window.

initial-cursor-visibility is used to specify whether the window should have a text cursor. `:off` (the default) means to make the cursor visible if the window is waiting for input. `:on` means to make the cursor visible immediately. `nil` means the cursor will not be visible at all.

When *save-under* is *true*, the result window will be given a "bit save array". The default is `nil`.

If `input-buffer` is supplied, it is an input buffer or event queue to use for the resulting window. Programmers will generally supply this when they want the new window to share its input buffer with an existing application. The default is to create a new input buffer.

29.5 Defining New Pane Types

This section describes how to define new pane classes. The first section shows a new kind of leaf pane (an odd kind of push-button). The second section shows a new composite pane that draws a dashed border around its contents.

29.5.1 Defining a New Leaf Pane

To define a gadget pane implementation, first define the appearance and layout behavior of the gadget, then define the callbacks, then define the specific user interactions that trigger the callbacks.

For example, to define an odd new kind of button that displays itself as a circle, and activates whenever the mouse is moved over it, proceed as follows:

```
;; A new kind of button.
(defclass sample-button-pane
          (action-gadget
           space-requirement-mixin
           leaf-pane)
    ())

;; An arbitrary size parameter.
(defparameter *sample-button-radius* 10)

;; Define the sheet's repaint method to draw the button.
(defmethod handle-repaint ((button sample-button-pane) region)
  (with-sheet-medium (medium button)
    (let ((radius *sample-button-radius*)
          (half (round  *sample-button-radius* 2)))
      ;; Larger circle with small one in the center
      (draw-circle* medium radius radius radius
                    :filled nil)
      (draw-circle* medium radius radius half
                    :filled t)))

;; Define the pane's compose-space method to always request the
;; fixed size of the pane.
(defmethod compose-space ((pane sample-button-pane) &key width height)
  (declare (ignore width height))
  (make-space-requirement :width  (* 2 *sample-button-radius*)
```

```
                        :height (* 2 *sample-button-radius*)))
```

The above code is enough to allow you to instantiate the button pane in an application frame. It will fit in with the space composition protocol of, for example, an `hbox-pane`. It will display itself as two nested circles.

The next step is to define the callbacks supported by this gadget, and the user interaction that triggers them.

```
;; This default method is defined so that the callback can be invoked
;; on an arbitrary client without error.
(defmethod activate-callback
           ((button sample-button-pane) client id)
  (declare (ignore client id value)))

;; This event processing method defines the rather odd interaction
;; style of this button, to wit, it triggers the activate callback
;; whenever the mouse moves into it.
(defmethod handle-event ((pane sample-button-pane) (event pointer-enter-event))
  (activate-callback pane (gadget-client pane) (gadget-id pane)))
```

29.5.2 Defining a New Composite Pane

To define a new layout pane implementation, the programmer must define how the much space the pane takes, where its children go, and what the pane looks like.

For example, to define a new kind of border pane that draws a dashed border around its child pane, proceed as follows:

```
;; The new layout pane class.
(defclass dashed-border-pane (layout-pane)
    ((thickness :initform 1 :initarg :thickness))
  (:default-initargs :background +black+))

;; The specified contents are the sole child of the pane.
(defmethod initialize-instance :after ((pane dashed-border-pane) &key contents)
  (sheet-adopt-child pane contents))

;; The composite pane takes up all of the space of the child, plus
;; the space required for the border.
(defmethod compose-space ((pane dashed-border-pane) &key width height)
  (let ((thickness (slot-value pane 'thickness))
        (child (sheet-child pane)))
    (space-requirement+
      (compose-space child :width width :height height)
      (make-space-requirement
```

```
        :width (* 2 thickness)
        :height (* 2 thickness)))))

;; The child pane is positioned just inside the borders.
(defmethod allocate-space ((pane dashed-border-pane) width height)
  (let ((thickness (slot-value pane 'thickness)))
    (move-and-resize-sheet
      (sheet-child pane)
      thickness thickness
      (- width (* 2 thickness)) (- height (* 2 thickness)))))

(defmethod handle-repaint ((pane dashed-border-pane) region)
  (declare (ignore region))                    ;not worth checking
  (with-sheet-medium (medium pane)
    (with-bounding-rectangle* (left top right bottom) (sheet-region pane)
      (let ((thickness (slot-value pane 'thickness)))
        (decf right (ceiling thickness 2))
        (decf bottom (ceiling thickness 2))
        (draw-rectangle* medium left top right bottom
                         :line-thickness thickness :filled nil
                         :ink (pane-background pane))))))

(defmacro dashed-border ((&rest options &key thickness &allow-other-keys)
                         &body contents)
  (declare (ignore thickness))
  `(make-pane 'dashed-border-pane
     :contents ,@contents
     ,@options))
```

Chapter 30

Gadgets

30.1 Overview of Gadgets

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. Each toolkit implementations will specify the "look and feel" of their gadgets. Typically, the look and feel will be derived directly from the underlying toolkit.

Each of CLIM's abstract gadgets has at least one standard implementation that is written using the facilities provided solely by CLIM itself. The gadgets' appearances are achieved via calls to the CLIM graphics functions, and their interactive behavior is defined in terms of the CLIM input event processing mechanism. Since these gadget implementations are written entirely in terms of CLIM, they are portable across all supported CLIM host window systems. Furthermore, since the specific look and feel of each such gadget is "fixed" in CLIM Lisp code, the gadget implementation will look and behave the same in all environments.

30.2 Abstract Gadgets

The push button and slider gadgets alluded to above are *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, `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 of an abstract gadget class is a subclass of the abstract class, they all share the same programmer interface.

30.2.1 Using Gadgets

Every gadget has a *client* that is 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 argument list for `activate-callback` looks like *(gadget client gadget-id)*. Assuming the programmer has defined an application frame called `button-test` that has a CLIM stream pane in the slot `output-pane`, he could write the following method:

```
(defmethod activate-callback
           ((button 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 function. In this case, if we have two buttons, we might install `start` and `stop` as the respective gadget ids and then use `eql` specializers on the gadget ids. We could then refine the above as:

```
(defmethod activate-callback
           ((button push-button) (client button-test) (gadget-id (eql 'start)))
  (start-test client))

(defmethod activate-callback
           ((button push-button) (client button-test) (gadget-id (eql 'stop)))
  (stop-test client))
```

```
;; Create the start and stop push buttons
(make-pane 'push-button
  :label "Start"
  :client frame :id 'start)
(make-pane 'push-button
  :label "Stop"
  :client frame :id 'stop)
```

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. The above example could then be written as follows:

```
;; No callback methods needed, just create the push buttons
(make-pane 'push-button
  :label "Start"
  :client frame :id 'start
  :activate-callback
    #'(lambda (gadget)
        (start-test (gadget-client gadget))))
(make-pane 'push-button
  :label "Stop"
  :client frame :id 'stop
  :activate-callback
    #'(lambda (gadget)
        (stop-test (gadget-client gadget))))
```

30.2.2 Implementing Gadgets

The following shows how a push button gadget might be implemented.

```
;; Here is a concrete implementation of a CLIM PUSH-BUTTON.
;; The "null" frame manager create a pane of type PUSH-BUTTON-PANE when
;; asked to create a PUSH-BUTTON.
(defclass push-button-pane
          (push-button
           leaf-pane
           space-requirement-mixin)
    ((show-as-default :initarg :show-as-default
                      :accessor push-button-show-as-default)
     (armed :initform nil)))

;; General highlight-by-inverting method.
(defmethod highlight-button ((pane push-button-pane) medium)
  (with-bounding-rectangle* (left top right bottom) (sheet-region pane)
    (draw-rectangle* medium left top right bottom
                     :ink +flipping-ink+ :filled t)
```

```
    (medium-force-output medium)))

;; Compute the amount of space required by a PUSH-BUTTON-PANE.
(defmethod compose-space ((pane push-button-pane) &key width height)
  (let ((x-margin 4)
        (y-margin 2))
  (multiple-value-bind (width height)
      (compute-gadget-label-size pane)
    (make-space-requirement :width  (+ width  (* x-margin 2))
                            :height (+ height (* y-margin 2))))))

;; This gets invoked to draw the push button.
(defmethod handle-repaint ((pane push-button-pane) region)
  (declare (ignore region))
  (with-sheet-medium (medium pane)
    (let ((text (gadget-label pane))
          (text-style (slot-value pane 'text-style))
          (armed (slot-value pane 'armed))
          (region (sheet-region pane)))
      (multiple-value-call #'draw-rectangle*
        medium (bounding-rectangle* (sheet-region pane))
        :filled nil)
      (draw-text medium text (bounding-rectangle-center region)
                 :text-style text-style
                 :align-x ':center :align-y ':center)
      (when (eql armed ':button-press)
        (highlight-button pane medium)))))

(defmethod handle-event :around ((pane push-button-pane) (event pointer-event))
  (when (gadget-active-p pane)
    (call-next-method)))

;; When we enter the push button's region, arm it.  If there is a pointer
;; button down, make the button active as well.
(defmethod handle-event ((pane push-button-pane) (event pointer-enter-event))
  (with-slots (armed) pane
    (unless armed
      (cond ((let ((pointer (pointer-event-pointer event)))
               (and (pointer-button-state pointer)
                    (not (zerop (pointer-button-state pointer)))))
             (setf armed :active)
             (with-sheet-medium (medium pane)
               (highlight-button pane medium)))
            (t (setf armed t)))
      (armed-callback pane (gadget-client pane) (gadget-id pane)))))

;; When we leave the push button's region, disarm it.
(defmethod handle-event ((pane push-button-pane) (event pointer-exit-event))
  (with-slots (armed) pane
    (when armed
```

```
      (when (prog1 (eq armed :active) (setf armed nil))
        (with-sheet-medium (medium pane)
          (highlight-button pane medium)))
      (disarmed-callback pane (gadget-client pane) (gadget-id pane)))))

;; When the user presses a pointer button, ensure that the button
;; is armed, and highlight it.
(defmethod handle-event ((pane push-button-pane) (event pointer-button-press-event))
  (with-slots (armed) pane
    (when armed
      (setf armed :active)
      (with-sheet-medium (medium pane)
        (highlight-button pane medium)))))

;; When the user releases the button and the button is still armed,
;; call the activate callback.
(defmethod handle-event ((pane push-button-pane) (event pointer-button-release-event))
  (with-slots (armed) pane
    (when (eq armed :active)
      (setf armed t)
      (with-sheet-medium (medium pane)
        (highlight-button pane medium))
      (activate-callback pane (gadget-client pane) (gadget-id pane)))))
```

30.3 Basic Gadget Classes

The following are the basic gadget classes upon which all gadgets are built.

⇒ `gadget` [*Protocol Class*]

The protocol class that corresponds to a gadget, a subclass of `pane`. If you want to create a new class that behaves like a gadget, it should be a subclass of `gadget`. All instantiable subclasses of `gadget` must obey the gadget protocol.

All of the subclasses of `gadget` are mutable.

⇒ `gadgetp` *object* [*Protocol Predicate*]

Returns *true* if *object* is a *gadget*, otherwise returns *false*.

⇒ `basic-gadget` [*Class*]

The basic class on which all CLIM gadgets are built, a subclass of `gadget`. This class is an abstract class, intended only to be subclassed, not instantiated.

⇒ `:id` [*Initarg*]
⇒ `:client` [*Initarg*]
⇒ `:armed-callback` [*Initarg*]

⇒ `:disarmed-callback` [*Initarg*]

All subclasses of `gadget` must handle these four initargs, which are used to specify, respectively, the gadget id, client, armed callback, and disarmed callback of the gadget.

⇒ `gadget-id` *gadget* [*Generic Function*]
⇒ `(setf gadget-id)` *id gadget* [*Generic Function*]

Returns (or sets) the gadget id of the gadget *gadget*. The id is typically a simple Lisp object that uniquely identifies the gadgets.

⇒ `gadget-client` *gadget* [*Generic Function*]
⇒ `(setf gadget-client)` *client gadget* [*Generic Function*]

Returns the client of the gadget *gadget*. The client is often 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, the client of the button could be the radio box).

⇒ `gadget-armed-callback` *gadget* [*Generic Function*]
⇒ `gadget-disarmed-callback` *gadget* [*Generic Function*]

Returns the functions that will be called when the armed or disarmed callback, respectively, are invoked. These functions will be invoked with a single argument, the gadget.

When these functions return `nil`, that indicates that there is no armed (or disarmed) callback for the gadget.

⇒ `armed-callback` *gadget client gadget-id* [*Callback Generic Function*]
⇒ `disarmed-callback` *gadget client gadget-id* [*Callback Generic Function*]

These callbacks are invoked when the gadget *gadget* is, respectively, armed or disarmed.

The exact definition of arming and disarming varies from gadget to gadget, but typically a gadget becomes armed when the pointer is moved into its region, and disarmed when the pointer moves out of its region. A gadget will not call the activate or value-changed callback unless it is armed.

The default methods (on `basic-gadget`) call the function stored in `gadget-armed-callback` or `gadget-disarmed-callback` with one argument, the gadget.

⇒ `activate-gadget` *gadget* [*Generic Function*]

Causes the host gadget to become active, that is, available for input.

⇒ `deactivate-gadget` *gadget* [*Generic Function*]

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.

⇒ `gadget-active-p` *gadget* [*Generic Function*]

Returns `t` if the gadget *gadget* is active (that is, has been activated with `activate-gadget`),

otherwise returns `nil`.

⇒ `note-gadget-activated` *client gadget* [*Generic Function*]

This function is invoked after a gadget is made active. It is intended to allow the client of the gadget to notice when the gadget has been activated.

⇒ `note-gadget-deactivated` *client gadget* [*Generic Function*]

This function is invoked after a gadget is made inactive. It is intended to allow the client of the gadget to notice when the gadget has been activated. For instance, when the client is an application frame, the frame may invoke the frame manager to "gray out" deactivated gadgets.

⇒ `value-gadget` [*Class*]

The class used by gadgets that have a value; a subclass of `basic-gadget`. This class is an abstract class, intended only to be subclassed, not instantiated.

⇒ `:value` [*Initarg*]
⇒ `:value-changed-callback` [*Initarg*]

All subclasses of `value-gadget` must handle these two initargs, which are used to specify, respectively, the initial value and the value changed callback of the gadget.

⇒ `gadget-value` *value-gadget* [*Generic Function*]

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.)

⇒ `(setf gadget-value)` *value value-gadget* `&key` *invoke-callback* [*Generic Function*]

Sets the gadget's value to the specified value.

If *invoke-callback* is *true*, the value changed callback for the gadget is invoked. The default is *false*. The syntax for using `(setf gadget-value)` in conjunction with *invoke-callback* is:

```
(setf (gadget-value gadget :invoke-callback t) new-value)
```

⇒ `gadget-value-changed-callback` *value-gadget* [*Generic Function*]

Returns the function that will be called when the value changed callback is invoked. This function will be invoked with a two arguments, the gadget and the new value.

When this function returns `nil`, that indicates that there is no value changed callback for the gadget.

⇒ `value-changed-callback` *value-gadget client gadget-id value* [*Callback Generic Function*]

This callback is invoked when the value of a gadget is changed, either by the user or programatically.

The default method (on `value-gadget`) calls the function stored in `gadget-value-changed-callback` with two arguments, the gadget and the new value.

CLIM implementations must implement or inherit a method for `value-changed-callback` for every gadget that is a subclass of `value-gadget`.

⇒ `action-gadget` [*Class*]

The class used by gadgets that perform some kind of action, such as a push button; a subclass of `basic-gadget`. This class is an abstract class, intended only to be subclassed, not instantiated.

⇒ `:activate-callback` [*Initarg*]

All subclasses of `action-gadget` must handle this initarg, which is used to specify the activate callback of the gadget.

⇒ `gadget-activate-callback` *action-gadget* [*Generic Function*]

Returns the function that will be called when the gadget is activated. This function will be invoked with one argument, the gadget.

When this function returns `nil`, that indicates that there is no value activate callback for the gadget.

⇒ `activate-callback` *action-gadget client gadget-id* [*Callback Generic Function*]

This callback is invoked when the gadget is activated.

The default method (on `action-gadget`) calls the function stored in `gadget-activate-callback` with one argument, the gadget.

CLIM implementations must implement or inherit a method for `activate-callback` for every gadget that is a subclass of `action-gadget`.

⇒ `oriented-gadget-mixin` [*Class*]

The class that is mixed in to a gadget that has an orientation associated with it, for example, a slider. This class is not intended to be instantiated.

⇒ `:orientation` [*Initarg*]

All subclasses of `oriented-gadget-mixin` must handle this initarg, which is used to specify the orientation of the gadget.

⇒ `gadget-orientation` *oriented-gadget* [*Generic Function*]

Returns the orientation of the gadget *oriented-gadget*. Typically, this will be a keyword such as `:horizontal` or `:vertical`.

⇒ `labelled-gadget-mixin` [*Class*]

The class that is mixed in to a gadget that has a label, for example, a push button. This class is not intended to be instantiated.

A label may be a `nil` (meaning no label), a string, a pattern, or a pixmap.

⇒ `:label` [*Initarg*]
⇒ `:align-x` [*Initarg*]
⇒ `:align-y` [*Initarg*]

All subclasses of `labelled-gadget-mixin` must handle these initargs, which are used to specify the label, and its x and y alignment. Labelled gadgets will also have a text style for the label, but this is managed by the usual text style mechanism for panes.

⇒ `gadget-label` *labelled-gadget* [*Generic Function*]
⇒ `(setf gadget-label)` *label labelled-gadget* [*Generic Function*]

Returns (or sets) the label of the gadget *labelled-gadget*. The label must be `nil`, a string, a pattern, or a pixmap. Changing the label of a gadget may result in invoking the layout protocol on the gadget and its ancestor sheets.

⇒ `gadget-label-align-x` *labelled-gadget* [*Generic Function*]
⇒ `(setf gadget-label-align-x)` *alignment labelled-gadget* [*Generic Function*]
⇒ `gadget-label-align-y` *labelled-gadget* [*Generic Function*]
⇒ `(setf gadget-label-align-y)` *alignment labelled-gadget* [*Generic Function*]

Returns (or sets) the alignment of the label of the gadget *labelled-gadget*. Changing the alignment a gadget may result in invoking the layout protocol on the gadget and its ancestor sheets.

⇒ `range-gadget-mixin` [*Class*]

The class that is mixed in to a gadget that has a range, for example, a slider. This class is not intended to be instantiated.

⇒ `:min-value` [*Initarg*]
⇒ `:max-value` [*Initarg*]

All subclasses of `range-gadget-mixin` must handle these two initargs, which are used to specify the minimum and maximum value of the gadget.

⇒ `gadget-min-value` *range-gadget* [*Generic Function*]
⇒ `(setf gadget-min-value)` *min-value range-gadget* [*Generic Function*]

Returns (or sets) the minimum value of the gadget *range-gadget*. It will be a real number.

⇒ `gadget-max-value` *range-gadget* [*Generic Function*]
⇒ `(setf gadget-max-value)` *max-value range-gadget* [*Generic Function*]

Returns (or sets) the maximum value of the gadget *range-gadget*. It will be a real number.

⇒ `gadget-range` *range-gadget* [*Generic Function*]

Returns the range of *range-gadget*, that is, the difference of the maximum value and the minimum value.

⇒ `gadget-range*` *range-gadget* [*Generic Function*]

Returns the minimum and maximum values of *range-gadget* as two values.

30.4 Abstract Gadget Classes

CLIM supplies a set of abstract gadgets that have been designed to be compatible with with a variety of user interface toolkits, including Xt widget-based toolkits (such as Motif), OpenLook, and MacApp and MicroSoft Windows.

CLIM's "concrete" gadget classes will all be subclasses of these abstract gadget classes. Each concrete gadget maps to an implementation-specific object that is managed by the underlying toolkit. For example, while a CLIM program manipulates an object of class `scroll-bar`, the underlying implementation-specific object might be an Xt widget of type `Xm_Scroll_Bar`. As events are processed on the underlying object the corresponding generic operations are applied to the Lisp gadget.

Minor issue: *Do we want to define something like* `gadget-handle` *that is a documented way to get ahold of the underlying toolkit object? — ILA*

Note that not all operations will necessarily be generated by particular toolkit implementations. For example, a user interface toolkit that is designed for a 3-button mouse may generate significantly more gadget events than one designed for a 1-button mouse.

30.4.1 The `push-button` Gadget

The `push-button` gadget provides press-to-activate switch behavior.

`arm-callback` will be invoked when the push button becomes armed (such as when the pointer moves into it, or a pointer button is pressed over it). When the button is actually activated (by releasing the pointer button over it), `activate-callback` will be invoked. Finally, `disarm-callback` will be invoked after `activate-callback`, or when the pointer is moved outside of the button.

⇒ `push-button` [*Class*]

The instantiable class that implements an abstract push button. It is a subclass of `action-gadget` and `labelled-gadget-mixin`.

⇒ `:show-as-default` [*Initarg*]

This is used to initialize the "show as default" property for the gadget, described below.

⇒ `push-button-show-as-default` *push-button* [*Generic Function*]

Returns the "show as default" property for the push button gadget. When *true*, the push button will be drawn with a heavy border, which indicates that this button is the "default operation".

⇒ `push-button-pane` [*Class*]

The instantiable class that implements a portable push button; a subclass of `push-button`.

30.4.2 The `toggle-button` Gadget

The `toggle-button` gadget provides "on/off" switch behavior. 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`.

`arm-callback` will be invoked when the toggle button becomes armed (such as when the pointer moves into it, or a pointer button is pressed over it). When the toggle button is actually activated (by releasing the pointer button over it), `value-changed-callback` will be invoked. Finally, `disarm-callback` will be invoked after `value-changed-callback`, or when the pointer is moved outside of the toggle button.

⇒ `toggle-button` [*Class*]

The instantiable class that implements an abstract toggle button. It is a subclass of `value-gadget` and `labelled-gadget-mixin`.

⇒ `:indicator-type` [*Initarg*]

This is used to initialize the indicator type property for the gadget, described below.

⇒ `toggle-button-indicator-type` *toggle-button* [*Generic Function*]

Returns the indicator type for the toggle button. 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.

⇒ `gadget-value` *(button* `toggle-button`*)* [*Method*]

Returns *true* if the button is selected, otherwise returns *false*.

⇒ `toggle-button-pane` [*Class*]

The instantiable class that implements a portable toggle button; a subclass of `toggle-button`.

30.4.3 The `menu-button` Gadget

The `menu-button` gadget provides similar behavior to the `toggle-button` gadget, except that it is intended for items in menus. The returned value is generally the item chosen from the menu.

`arm-callback` will be invoked when the menu button becomes armed (such as when the pointer moves into it, or a pointer button is pressed over it). When the menu button is actually activated (by releasing the pointer button over it), `value-changed-callback` will be invoked. Finally, `disarm-callback` will be invoked after `value-changed-callback`, or when the pointer is moved outside of the menu button.

⇒ `menu-button` [*Class*]

The instantiable class that implements an abstract menu button. It is a subclass of `value-gadget` and `labelled-gadget-mixin`.

⇒ `menu-button-pane` [*Class*]

The instantiable class that implements a portable menu button; a subclass of `menu-button`.

30.4.4 The `scroll-bar` Gadget

The `scroll-bar` gadget corresponds to a scroll bar.

⇒ `scroll-bar` [*Class*]

The instantiable class that implements an abstract scroll bar. This is a subclass of `value-gadget`, `oriented-gadget-mixin`, and `range-gadget-mixin`.

⇒ `:drag-callback` [*Initarg*]
⇒ `:scroll-to-bottom-callback` [*Initarg*]
⇒ `:scroll-to-top-callback` [*Initarg*]
⇒ `:scroll-down-line-callback` [*Initarg*]
⇒ `:scroll-up-line-callback` [*Initarg*]
⇒ `:scroll-down-page-callback` [*Initarg*]
⇒ `:scroll-up-page-callback` [*Initarg*]

Specifies the drag and other scrolling callbacks for the scroll bar.

⇒ `scroll-bar-drag-callback` *scroll-bar* [*Generic Function*]

Returns the function that will be called when the indicator of the scroll bar is dragged. This function will be invoked with a two arguments, the scroll bar and the new value.

⇒ `scroll-bar-scroll-to-bottom-callback` *scroll-bar* [*Generic Function*]
⇒ `scroll-bar-scroll-to-top-callback` *scroll-bar* [*Generic Function*]
⇒ `scroll-bar-scroll-down-line-callback` *scroll-bar* [*Generic Function*]
⇒ `scroll-bar-scroll-up-line-callback` *scroll-bar* [*Generic Function*]

⇒ `scroll-bar-scroll-down-page-callback` *scroll-bar* [*Generic Function*]
⇒ `scroll-bar-scroll-up-page-callback` *scroll-bar* [*Generic Function*]

Returns the functions that will be used as callbacks when various parts of the scroll bar are clicked on. These are all functions of a single argument, the scroll bar.

When any of these functions returns `nil`, that indicates that there is no callback of that type for the gadget.

⇒ `drag-callback` *scroll-bar client gadget-id value* [*Callback Generic Function*]

This callback is invoked when the value of the scroll bar is changed while the indicator is being dragged. This is implemented by calling the function stored in `scroll-bar-drag-callback` with two arguments, the scroll bar and the new value.

The `value-changed-callback` is invoked only after the indicator is released after dragging it.

⇒ `scroll-to-top-callback` *scroll-bar client gadget-id* [*Callback Generic Function*]
⇒ `scroll-to-bottom-callback` *scroll-bar client gadget-id* [*Callback Generic Function*]
⇒ `scroll-up-line-callback` *scroll-bar client gadget-id* [*Callback Generic Function*]
⇒ `scroll-up-page-callback` *scroll-bar client gadget-id* [*Callback Generic Function*]
⇒ `scroll-down-line-callback` *scroll-bar client gadget-id* [*Callback Generic Function*]
⇒ `scroll-down-page-callback` *scroll-bar client gadget-id* [*Callback Generic Function*]

All of the callbacks above are invoked when appropriate parts of the scroll bar are clicked on. Note that each implementation may not have "hot spots" corresponding to each of these callbacks.

⇒ `gadget-value` *(button* `scroll-bar`*)* [*Method*]

Returns a real number within the specified range.

⇒ `scroll-bar-pane` [*Class*]

The instantiable class that implements a portable scroll bar; a subclass of `scroll-bar`.

30.4.5 The `slider` Gadget

The `slider` gadget corresponds to a slider.

⇒ `slider` [*Class*]

The instantiable class that implements an abstract slider. This is a subclass of `value-gadget`, `oriented-gadget-mixin`, `range-gadget-mixin`, and `labelled-gadget-mixin`.

⇒ `:drag-callback` [*Initarg*]
⇒ `:show-value-p` [*Initarg*]
⇒ `:decimal-places` [*Initarg*]

Specifies the drag callback for the slider, whether the slider should show its current value, and

how many decimal places to the right of the decimal point should be displayed when the slider is showing its current value.

⇒ `:min-label` [*Initarg*]
⇒ `:max-label` [*Initarg*]
⇒ `:range-label-text-style` [*Initarg*]

Specifies a label to be used at the low end and high end of the slider, and what the text style of those labels should be. The min and max labels must be `nil`, a string, a pattern, or a pixmap.

⇒ `:number-of-tick-marks` [*Initarg*]
⇒ `:number-of-quanta` [*Initarg*]

Specifies the number of tick marks that should be drawn on the scroll bar, and the number of quanta in the scroll bar. If the scroll bar is quantized, the scroll bar will consist of discrete values rather than continuous values.

⇒ `gadget-show-value-p` *slider* [*Generic Function*]

Returns *true* if the slider shows its value, otherwise returns *false*

⇒ `slider-drag-callback` *slider* [*Generic Function*]

Returns the function that will be called when the indicator of the slider is dragged. This function will be invoked with a two arguments, the slider and the new value.

When this function returns `nil`, that indicates that there is no drag callback for the gadget.

⇒ `drag-callback` *slider client gadget-id value* [*Callback Generic Function*]

This callback is invoked when the value of the slider is changed while the indicator is being dragged. This is implemented by calling the function stored in `slider-drag-callback` with two arguments, the slider and the new value.

The `value-changed-callback` is invoked only after the indicator is released after dragging it.

⇒ `gadget-value` *(button* `slider`*)* [*Method*]

Returns a real number within the specified range.

⇒ `slider-pane` [*Class*]

The instantiable class that implements a portable slider; a subclass of `slider`.

30.4.6 The `radio-box` and `check-box` Gadgets

Radio boxes and check boxes are special kinds of gadgets that constrain one or more toggle buttons. At any one time, only one of the buttons managed by the radio box, or zero or more of the buttons managed by a check box, may be "on". The contents of a radio box or a check box

are its buttons, and as such a radio box or check bnox is responsible for laying out the buttons that it contains. A radio box or check box is a client of each of its buttons so that the value of the radio or check box can be properly computed.

As the current selection changes, the previously selected button and the newly selected button both have their `value-changed-callback` handlers invoked.

⇒ `radio-box` [*Class*]

The instantiable class that implements an abstract radio box, that is, a gadget that constrains a number of toggle buttons, only one of which may be selected at any one time. It is a subclass of `value-gadget` and `oriented-gadget-mixin`.

⇒ `:current-selection` [*Initarg*]

This is used to specify which button, if any, should be initially selected.

⇒ `radio-box-current-selection` *radio-box* [*Generic Function*]
⇒ `(setf radio-box-current-selection)` *button radio-box* [*Generic Function*]

Returns (or sets) the current selection for the radio box. The current selection will be one of the toggle buttons in the box.

⇒ `radio-box-selections` *radio-box* [*Generic Function*]

Returns a sequence of all of the selections in the radio box. The elements of the sequence will be toggle buttons.

⇒ `gadget-value` *(button* `radio-box`*)* [*Method*]

Returns the selected button. This will return the same value as `radio-box-current-selection`

⇒ `radio-box-pane` [*Class*]

The instantiable class that implements a portable radio box; a subclass of `radio-box`.

⇒ `check-box` [*Class*]

The instantiable class that implements an abstract check box, that is, a gadget that constrains a number of toggle buttons, zero or more of which may be selected at any one time. It is a subclass of `value-gadget` and `oriented-gadget-mixin`.

⇒ `:current-selection` [*Initarg*]

This is used to specify which buttons, if any, should be initially selected.

⇒ `check-box-current-selection` *check-box* [*Generic Function*]
⇒ `(setf check-box-current-selection)` *button check-box* [*Generic Function*]

Returns (or sets) the current selection for the check box. The current selection will be a list of zero or more of the toggle buttons in the box.

⇒ `check-box-selections` *check-box* [*Generic Function*]

Returns a sequence of all of the selections in the check box. The elements of the sequence will be toggle buttons.

⇒ `gadget-value` *(button* `check-box`*)* [*Method*]

Returns the selected buttons as a list. This will return the same value as `check-box-current-selection`

⇒ `check-box-pane` [*Class*]

The instantiable class that implements a portable check box; a subclass of `check-box`.

⇒ `with-radio-box` *(*`&rest` *options* `&key` *(type* `:one-of`*)* `&allow-other-keys` *)* `&body` *body* [*Macro*]

Creates a radio box whose buttons are created by the forms in *body*. The macro `radio-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`, a `radio-box` will be created. If *type* is `:some-of`, a `check-box` will be created.

For example, the following creates a radio box with three buttons in it, the second of which is initially selected.

```
(with-radio-box ()
  (make-pane 'toggle-button :label "Mono")
  (radio-box-current-selection
    (make-pane 'toggle-button :label "Stereo"))
  (make-pane 'toggle-button :label "Quad"))
```

The following simpler form can also be used when the programmer does not need to control the appearance of each button closely.

```
(with-radio-box ()
  "Mono" "Stereo" "Quad")
```

30.4.7 The `list-pane` and `option-pane` Gadgets

⇒ `list-pane` [*Class*]

The instantiable class that implements an abstract list pane, that is, a gadget 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`.

⇒ `:mode` [*Initarg*]

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` [*Initarg*]
⇒ `:name-key` [*Initarg*]
⇒ `:value-key` [*Initarg*]
⇒ `:test` [*Initarg*]

The `:items` initarg specifies a sequence of items to use as the items of the list pane. The name of the item is extracted by the function that is the value of the `:name-key` initarg, which defaults to `princ-to-string`. The value of the item is extracted by the function that is the value of the `:value-key` initarg, which defaults to `identity`. The `:test` initarg specifies a function of two arguments that is used to compare items; it defaults to `eql`.

For example,

```
ΣΠD,#TD1PsŒT0[Begin using 006 escapes]Σ(1 0 (NIL 0) (SAGE:SANS-SERIF-BODY SAGE::TYPEWRITER :NORMAL
Σ0  Σ1:value '("Lisp" "C++")
Σ0  Σ1:mode :nonexclusive
Σ0  Σ1:items '("Lisp" "Fortran" "C" "C++" "Cobol" "Ada")
Σ0  Σ1:test 'string=)
Σ0\end{varbatim}

\Defmethod {gadget-value} {(button \cl{list-pane})}

Returns the single selected item when the mode is \cl{:exclusive}, or a sequence
of selected items when the mode is \cl{:nonexclusive}.

\Defclass {generic-list-pane}

The instantiable class that implements a portable list pane; a subclass of
\cl{list-pane}.

\Defclass {option-pane}

The instantiable class that implements an abstract option pane, that is, a
gadget whose semantics are identical to a list pane, but whose visual appearance
is a single push button which, when pressed, pops up a menu of selections..  It
is a subclass of \cl{value-gadget}.

\Definitarg {:mode}

Either \cl{:nonexclusive} or \cl{:exclusive}.  When it is \cl{:exclusive}, the
option pane acts like a radio box, that is, only a single item can be selected.
Otherwise, the option pane acts like a check box, in that zero or more items can
be selected. The default is \cl{:exclusive}.

\definitarg {:items}
```

```
\definitarg {:name-key}
\definitarg {:value-key}
\Definitarg {:test}

The \cl{:items} initarg specifies a sequence of items to use as the items of the
option pane.  The name of the item is extracted by the function that is the value
of the \cl{:name-key} initarg, which defaults to \cl{princ-to-string}.  The
value of the item is extracted by the function that is the value of the
\cl{:value-key} initarg, which defaults to \cl{identity}.  The \cl{:test}
initarg specifies a function of two arguments that is used to compare items; it
defaults to \cl{eql}.

For example,

\begin{verbatim}
Σ1(make-pane 'option-pane
Σ0  Σ1:value '("Lisp" "C++")
Σ0  Σ1:mode :nonexclusive
Σ0  Σ1:items '("Lisp" "Fortran" "C" "C++" "Cobol" "Ada")
Σ0  Σ1:test 'string=)
Σ0\end{varbatim}

\Defmethod {gadget-value} {(button \cl{option-pane})}

Returns the single selected item when the mode is \cl{:exclusive}, or a sequence
of selected items when the mode is \cl{:nonexclusive}.

\Defclass {generic-option-pane}

The instantiable class that implements a portable option pane; a subclass of
\cl{option-pane}.


\Gadget {text-field}

The \cl{text-field} gadget corresponds to a small field containing text.

\Defclass {text-field}

The instantiable class that implements an abstract text field.  This is a subclass of
\cl{value-gadget} and \cl{action-gadget}.

The value of a text field is the text string.

\Definitarg {:editable-p}

This is used to specify whether or not the text field may be edited.

\Defmethod {gadget-value} {(button \cl{text-field})}
```

Returns the resulting string.

\Defclass {text-field-pane}

The instantiable class that implements a portable text field; a subclass of \cl{text-field}.

\Gadget {text-editor}

The \cl{text-editor} gadget corresponds to a large field containing multiple lines of text.

\Defclass {text-editor}

The instantiable class that implements an abstract large text field. This is a subclass of \cl{text-field}.

The value of a text editor is the text string.

\definitarg {:ncolumns}
\Definitarg {:nlines}

Specifies the width and height of the text editor in columns and number of lines.

\Defmethod {gadget-value} {(button \cl{text-editor})}

Returns the resulting string.

\Defclass {text-editor-pane}

The instantiable class that implements a portable text editor; a subclass of \cl{text-editor}.

\section {Integrating Gadgets and Output Records}

In addition to gadget panes, CLIM allows gadgets to be used inside of CLIM stream panes. For instance, an \cl{accepting-values} whose fields consist of gadgets may appear in an ordinary CLIM stream pane.

Note that many of the functions in the output record protocol must correctly manage the case where there are gadgets contained within output records. For example, \cl{(setf* output-record-position)} may need to notify the host window system that the toolkit object representing the gadget has moved, \cl{window-clear} needs to deactive any gadgets, and so forth.

\Defclass {gadget-output-record}

The instantiable class the represents an output record class that contains a

```
gadget.  This is a subclass of \cl{output-record}.

\Defmacro {with-output-as-gadget} {(stream \rest options) \body body}

Invokes \arg{body} to create a gadget, and then creates a gadget output record
that contains the gadget and install's it into the output history of the output
recording stream \arg{stream}.  The returned value of \arg{body} must be the
gadget.

The options in \arg{options} are passed as initargs to the call to
\cl{invoke-with-new-output-record} that is used to create the gadget output
record.

The \arg{stream} argument is not evaluated, and must be a symbol that is bound to
an output recording stream.  If \arg{stream} is \cl{t}, \cl{*standard-output*} is
used.  \arg{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:

\begin{verbatim}
(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))
```

A more complex (and somewhat contrived) example of a push button that calls back into the presentation type system to execute a command might be as follows:

```
(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-frame*)
              :type 'command)
            *input-context*
            (make-instance 'pointer-button-press-event
              :sheet (sheet-parent button)
```

```
:x 0 :y 0
:modifiers 0
:button +pointer-left-button+)))))
```

Part VIII

Appendices

Appendix A

Glossary

Minor issue: *Fill these in. What glossary entries would it be useful to borrow from the ANSI CL spec? — SWM*

adaptive toolkit *n.* —Fill this in—

adopted *adj.* (of a *sheet*) Having a parent sheet.

affine transformation *n.* A *transformation*.

ancestors *n.* The parent of a *sheet* or an *output record*, and all of its ancestors, recursively.

applicable *adj.* (of a *presentation translator*) A *presentation translator* is said to be *applicable* when the pointer is pointing to a *presentation* whose *presentation type* matches the current *input context*, and the other criteria for translator matching have been met.

application frame *n.* 1. A program that interacts directly with a *user* to perform some specific task. 2. A Lisp object that holds the information associated with such a program, including the panes of the user interface and application state variables.

area *n.* A *region* that has dimensionality 2, that is, has area.

background *n.* The *design* that is used when erasing, that is, drawing using `+background-ink+`.

bounded design *n.* A *design* that is transparent everywhere beyond a certain distance from a certain point. Drawing a bounded design has no effect on the drawing plane outside that distance.

bounded region *n.* 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.

bounding rectangle *n.* 1. The smallest *rectangle* that surrounds a *bounded region* and contains every point in the region, and may contain additional points as well. The sides of a bounding rectangle are parallel to the coordinate axes. 2. A Lisp object that represents a *bounding rectangle*.

cache value *n.* During *incremental redisplay*, the *cache value* is used to determine whether or not a piece of output has changed.

children *n.* (of a *sheet* or *output record*) The direct descendants of a *sheet* or an *output record*.

color *n.* 1. An object representing the intuitive definition of a color, such as black or red. 2. A Lisp object that represents a *color*.

colored design *n.* A *design* whose points have *color*.

colorless design *n.* A *design* whose points have no *color*. Drawing a colorless design uses the default color specified by the *medium*'s foreground design.

command *n.* 1. The way CLIM represents a user interaction. 2. A Lisp object that represents a *command*.

command name *n.* A symbol that names a command.

command table *n.* 1. A way of collecting and organizing a group of related commands, and defining the interaction styles that can be used to invoke those commands. 2. A Lisp object that represents a *command table*.

command table designator *n.* A Lisp object that is either a command table or a symbol that names a command table.

completion *n.* A facility provided by CLIM for completing user input over a set of possibilities.

compositing *n.* (of *designs*) The creation of a *design* whose appearance at each point is a composite of the appearances of two other designs at that point. There are three varieties of compositing: *composing over*, *composing in*, and *composing out*.

composition *n.* (of *transformations*) The transformation from one coordinate system to another, then from the second to a third can be represented by a single transformation that is the *composition* of the two component transformations. Transformations are closed under composition. Composition is not commutative. Any arbitrary transformation can be built up by composing a number of simpler transformations, but that composition is not unique.

context-dependent input *n.* —Fill this in—

degrafted *adj.* (of a *sheet*) Not *grafted*.

descendants *n.* All of the children of a *sheet* or an *output record*, and all of their descendents, recursively.

design *n.* An object that represents a way of arranging *colors* and *opacities* in the *drawing plane*. A mapping from an (x, y) pair into color and opacity values.

device transformation *n.* The transformation that maps user coordinates into host-specific coordinates for a sheet or medium.

disowned *adj.* (of a *sheet*) Not *adopted*.

disabled *adj.* (of a *sheet*) Not *enabled*.

dispatching *n.* (of *events*) —Fill this in—

display medium *n.* —Fill this in—

display server *n.* A device on which windows are displayed, such as a Lisp Machine running Genera, or some kind of X11 display.

displayed output record *n.* An *output record* that corresponds to a visible piece of output, such as text or graphics. The leaves of the output record tree.

distributing *n.* (of *events*) —Fill this in—

drawing plane *n.* An infinite two-dimensional plane on which graphical output occurs. A drawing plane contains an arrangement of colors and opacities that is modified by each graphical output operation.

enabled *adj.* (of a *sheet*) A sheet is *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.

event *n.* 1. Some sort of significant event, such as a user gesture (such as moving the pointer, pressing a pointer button, or typing a keystroke) or a window configuration event (such as resizing a window). 2. A Lisp object that represents an *event*.

extended input stream *n.* A kind of sheet that supports CLIM's extended input stream protocol, such as supporting a pointing device.

extended output stream *n.* A kind of sheet that supports CLIM's extended output stream protocol, such as supporting a variable line-height text rendering.

false *n.* 1. The boolean value false. 2. The Lisp object `nil`.

flipping ink *n.* 1. An *ink* that interchanges occurrences of two *designs*, such as might be done by "XOR" on a monochrome display. 2. A Lisp object that represents a *flipping ink*.

foreground *n.* The *design* that is used when drawing using `+foreground-ink+`.

formatted output *n.* 1. Output that obeys some high level constraints on its appearance, such as being arranged in a tabular format, or justified within some margins. 2. The CLIM facility that provides a programmer the tools to produce such output.

frame *n.* An *application frame*.

frame manager *n.* An object that controls the realization of the look and feel of an *application frame*. Frame managers are responsible for mapping abstract pane classes to concrete pane classes on each window system platform.

fully specified *adj.* (of a *text style*) Having components none of which are `nil`, and not having a relative size (that is, neither `:smaller` nor `:larger`).

gesture *n.* Some sort of input action by a user, such as typing a character or clicking a pointer button.

gesture name *n.* A symbol that gives a name to a *gesture*, for example, `:select` is commonly used to indicate a left pointer button click.

graft *n.* A kind of *mirrored sheet* that represents a host window, typically a root window.

grafted *adj.* (of a *sheet*) Having an ancestor sheet that is a *graft*.

highlighting *n.* Changing of some piece of output so that it stands out. CLIM often *highlights* the *presentation* under the *pointer* to indicate that it is *sensitive*.

immutable *adj.* 1. (of an object) Having components that cannot be modified once the object has been created. 2. (of a class) An *immutable class* is a class all of whose objects are *immutable*.

implementor *n.* A programmer who implements CLIM.

incremental redisplay *n.* 1. Redraw part of some output while leaving other output unchanged. 2. The CLIM facility that implements this behavior.

indirect ink *n.* Drawing with an *indirect ink* is the same as drawing with another *ink* named directly.

ink *n.* Any member of the class `design` supplied as the `:ink` argument to a CLIM drawing function.

input context *n.* 1. A state in which a program is expecting input of a certain type. 2. A Lisp object that represents an *input context*.

input editor *n.* The CLIM facility that allows a *user* to modify typed-in input.

input editing stream *n.* A CLIM stream that supports *input editing*.

input stream designator *n.* A Lisp object that is either an input stream, or the symbol `t`, which is taken to mean `*standard-input*`.

interactive stream *n.* A stream that supports both input from and output to the user in an interactive fashion.

line style *n.* 1. Advice to CLIM's rendering substrate on how to render a path, such as a line or an unfilled ellipse or polygon. 2. A Lisp object that represents a *line style*.

medium *n.* 1. A destination for output, having a *drawing plane*, two designs called the medium's *foreground* and *background*, a *transformation*, a *clipping region*, a *line style*, and a *text style*. 2. A Lisp object that represents a *medium*.

mirror *n.* The host window system object associated with a *mirrored sheet*, such as a window object on an X11 display server.

mirrored sheet *n.* A special class of *sheet* that is attached directly to a window on a *display server*. A *graft* is one kind of a *mirrored sheet*.

mutable *adj.* 1. (of an object) Having components that can be modified once the object has been created. 2. (of a class) An *mutable class* is a class all of whose objects are *mutable*.

non-uniform design *n.* A *design* that is not a *uniform design*.

opacity *n.* 1. An object that controls how graphical output covers previous output, such as fully opaque to fully transparent, and levels of translucency between. 2. A Lisp object that represents an *opacity*.

output history *n.* The highest level *output record* for an *output recording stream*.

output record *n.* 1. An object that remembers the output performed to a *stream* or *medium*. 2. A Lisp object that represents an *output record*.

output recording *n.* The process of remembering the output performed to a *stream*.

output recording stream *n.* A CLIM stream that supports *output recording*.

output stream designator *n.* A Lisp object that is either an output stream, or the symbol `t`, which is taken to mean `*standard-output*`.

pane *n.* A *sheet* or window that appears as the child of some other window or *frame*. A composite pane can hold other panes; a leaf pane cannot.

parameterized presentation type *n.* A *presentation type* whose semantics are modified by parameters, such as `(integer 0 10)`. A parameterized presentation type is always a subtype of the presentation type without parameters. Parameterized presentation types are analogous to Common Lisp types which have parameters.

parent *n.* The direct ancestor of a *sheet* or an *output record*.

path *n.* A *region* that has dimensionality 1, that is, has length.

patterning *n.* The process of creating a bounded rectangular arrangement of *designs*, like a checkerboard. A *pattern* is a *design* created by this process.

pixmap *n.* An "off-screen window", that is, a sheet that can be used for graphical output, but is not visible on any display device.

point *n.* 1. A *region* that has dimensionality 0, that is, has only a position. 2. A Lisp object that represents a *point*.

pointer *n.* A physical device used for pointing, such as a mouse.

pointer documentation *n.* Short documentation associated with a presentation that describes what will happen when any button on the pointer is pressed.

port *n.* An abstract connection to a *display server* that is responsible for managing host display server resources and for processing input events received from the host display server.

position *n.* 1. A position on a plane, such as CLIM's abstract drawing plane. 2. A pair of real number values x and y that represent a *position*.

presentation *n.* 1. An association between an object and a *presentation type* with some output on an *output recording stream*. 2. A Lisp object that represents a *presentation*.

presentation method *n.* A method that supports some part of the behavior of a presentation type, such as the `accept` or `present` methods.

presentation tester *n.* A predicate that restricts the applicability of a *presentation translator*.

presentation translator *n.* A mapping from an object of one *presentation type*, an *input context*, and a *gesture* to an object of another presentation type. In effect, a translator broadens the input context so that some presentations are sensitive when the program asking for input is expecting a different type.

presentation type *n.* 1. A description of a class of *presentations*. 2. An extension to CLOS that implements this.

presentation type specifier *n.* A Lisp object used to specify a *presentation type*.

programmer *n.* A person who writes application programs using CLIM.

protocol class *n.* An "abstract" class having no methods or slots that is used to indicate that a class obeys a certain protocol. For example, all classes that inherit from the `bounding-rectangle` class obey the bounding rectangle protocol.

rectangle *n.* 1. A four-sided polygon whose sides are parallel to the coordinate axes. 2. A Lisp object that represents a *rectangle*.

redisplay *n.* See *incremental redisplay*.

region *n.* 1. A set of mathematical points in the plane; a mapping from an (x, y) pair into either true or false (meaning member or not a member, respectively, of the region). In CLIM, all regions include their boundaries (that is, they are closed) and have infinite resolution. 2. A Lisp object that represents a *region*.

region set *n.* 1. A "compound" *region*, that is, a region consisting of several other regions related by one of the operations union, intersection, or difference. 2. A Lisp object that represents a *region set*.

rendering *n.* The process of drawing a shape (such as a line or a circle) on a display device. Rendering is an approximate process, since an abstract shape exists in a continuous coordinate system having infinite precision, whereas display devices must necessarily draw discrete points having some measurable size.

replaying *n.* The process of redrawing a set of *output records*.

repainting *n.* The act of redrawing all of the sheets or output records in a "damage region", such as occurs when a window is raised from underneath occluding windows.

sensitive *adj.* (of a *presentation*) A *presentation* is *sensitive* if some action will take place when the user clicks on it with the pointer, that is, there is at least one *presentation translator* that is *applicable*. In this case, the presentation will usually be *highlighted*.

server path *n.* —Fill this in—

sheet *n.* 1. The basic unit of windowing in CLIM. A sheets attributes always include a region and a mapping to the coordinate system of its parent, and may include other attributes, such as a medium and event handling. 2. A Lisp object that represents a *sheet*.

sheet region *n.* The region that a sheet occupies.

sheet transformation *n.* A transformation that maps the coordinates of a sheet to the coordinate system of its parent, if it has a parent.

solid design *n.* A *design* that is either completely opaque or completely transparent. A solid design can be opaque at some points and transparent at others.

stencil *n.* A kind of *pattern* that contains only *opacities*.

stencil opacity *n.* The *opacity* at one point in a *design* 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. The *stencil opacity* of an *opacity* is simply its value.

stream *n.* A kind of *sheet* that implements the stream protocol (such as maintaining a *text cursor*).

text cursor *n.* The place in the drawing plane where the next piece of textual output will appear.

text style *n.* 1. A description of how textual output should appear, consisting of family, face code, and size. 2. A Lisp object that represents a *text style*.

tiling *n.* The process of repeating a rectangular portion of a *design* throughout the drawing plane. A *tile* is a *design* created by this process.

transformation *n.* 1. A mapping from one coordinate system onto another that preserves straight lines. General transformations include all the sorts of transformations that CLIM uses, namely, translations, scaling, rotations, and reflections. 2. A Lisp object that represents a *transformation*.

translucent design *n.* A *design* that is not *solid*, that is, has at least one point with an opacity that is intermediate between completely opaque and transparent.

true *n.* 1. The boolean value true; not *false*. 2. Any Lisp object that is not `nil`.

unbounded design *n.* A *design* that has at least one point of non-zero opacity arbitrarily far from the origin. Drawing an unbounded design affects the entire drawing plane.

unbounded region *n.* A *region* that either contains no points or contains points arbitrarily far apart.

uniform design *n.* A *design* that has the same color and opacity at every point in the drawing plane. Uniform designs are always unbounded, unless they are completely transparent.

unique id *n.* During *incremental redisplay*, the *unique id* is an object used to uniquely identify a piece of output. The output named by the *unique id* will often have a *cache value* associated with it.

user *n.* A person who uses an application program that was written using CLIM.

user transformation *n.* —Fill this in—

view *n.* 1. A way of displaying a presentation. 2. A Lisp object that represents a *view*.

viewport *n.* The portion of the drawing plane of a sheet's medium that is visible on a display device.

volatile *adj.* (of an immutable object) Having components that cannot be modified by the programmer at the protocol level, but which may be modified internally by CLIM. *Volatile* objects reflect internal state of CLIM.

Appendix B

The CLIM-SYS Package

The `clim-sys` package where useful "system-like" functionality lives, including such things as resources and multi-processing primitives. It contains concepts that are not part of Common Lisp, but which are not conceptually the province of CLIM itself.

All of the symbols documented in this appendix must be accessible as external symbols in the `clim-sys` package.

B.1 Resources

CLIM provides a facility called *resources* that provides for reusing objects. A resource describes how to construct an object, how to initialize and deinitialize it, and how an object should be selected from the resource of objects based on a set of parameters.

⇒ `defresource` *name parameters* `&key` *constructor initializer deinitializer matcher initial-copies* [*Macro*]

Defines a resource named *name*, which must be a symbol. *parameters* is a lambda-list giving names and default values (for optional and keyword parameters) of parameters to an object of this type.

constructor is a form that is responsible for creating an object, and is called when someone tries to allocate an object from the resource and no suitable free objects exist. The constructor form can access the parameters as variables. This argument is required.

initializer is a form that is used to initialize an object gotten from the resource. It can access the parameters as variables, and also has access to a variable called *name*, which is the object to be initialized. The initializer is called both on newly created objects and objects that are being reused.

deinitializer is a form that is used to deinitialize an object when it is about to be returned to the resource. It can access the parameters as variables, and also has access to a variable called *name*, which is the object to be deinitialized. It is called whenever an object is deallocated back to the resource, but is not called by `clear-resource`. Deinitializers are typically used to clear references to other objects.

matcher is a form that ensures that an object in the resource "matches" the specified parameters, which it can access as variables. In addition, the matcher also has access to a variable called *name*, which is the object in the resource being matched against. If no matcher is supplied, the system remembers the values of the parameters (including optional ones that defaulted) that were used to construct the object, and assumes that it matches those particular values for all time. This comparison is done with `equal`. The matcher should return *true* if there is a match, otherwise it should return *false*.

initial-copies is used to specify the number of objects that should be initially put into the resource. It must be an integer or `nil` (the default), meaning that no initial copies should be made. If initial copies are made and there are parameters, all the parameters must be optional; in this case, the initial copies have the default values of the parameters.

⇒ `using-resource` *(variable name* `&rest` *parameters)* `&body` *body* [*Macro*]

The forms in *body* are evaluated with *variable* bound to an object allocated from the resource named *name*, using the parameters given by *parameters*. The parameters (if any) are evaluated, but *name* is not.

After the body has been evaluated, `using-resource` returns the object in *variable* back to the resource. If some form in the body sets *variable* to `nil`, the object will not be returned to the resource. Otherwise, the body should not changes the value of *variable*.

⇒ `allocate-resource` *name* `&rest` *parameters* [*Function*]

Allocates an object from the resource named *name*, using the parameters given by *parameters*. *name* must be a symbol that names a resource. The returned value is the allocated object.

⇒ `deallocate-resource` *name object* [*Function*]

Returns the object *object* to the resource named *name*. *name* must be a symbol that names a resource. *object* must be an object that was originally allocated from the same resource.

⇒ `clear-resource` *name* [*Function*]

Clears the resource named *name*, that is, removes all of the resourced object from the resource. *name* must be a symbol that names a resource.

⇒ `map-resource` *function name* [*Function*]

Calls *function* once on each object in the resource named *name*. *function* is a function of three arguments, the object, a boolean value that is *true* if the object is in use or *false* if it is free, and *name*. *function* has dynamic extent.

B.2 Multi-processing

Most Lisp implementations provide some form of multi-processing. CLIM provides a set of functions that implement a uniform interface to the multi-processing functionality.

⇒ `*multiprocessing-p*` [*Variable*]

The value of `*multiprocessing-p*` is `t` if the current Lisp environment supports multi-processing, otherwise it is `nil`.

⇒ `make-process` *function* `&key` *name* [*Function*]

Creates a process named *name*. The new process will evaluate the function *function*. On systems that do not support multi-processing, `make-process` will signal an error.

⇒ `destroy-process` *process* [*Function*]

Terminates the process *process*. *process* is an object returned by `make-process`.

⇒ `current-process` [*Function*]

Returns the currently running process, which will be the same kind of object as would be returned by `make-process`.

⇒ `all-processes` [*Function*]

Returns a sequence of all of the processes.

⇒ `processp` *object* [*Protocol Predicate*]

Returns `t` if *object* is a process, otherwise returns *nil*.

⇒ `process-name` *process* [*Function*]
⇒ `process-state` *process* [*Function*]
⇒ `process-whostate` *process* [*Function*]

These functions return, respectively, the name, state, and "whostate" of the process. These format of these quantities will vary depending on the platform.

⇒ `process-wait` *reason predicate* [*Function*]

Causes the current process to wait until *predicate* returns *true*. *reason* is a "reason" for waiting, usually a string. On systems that do not support multi-processing, `process-wait` will loop until *predicate* returns *true*.

⇒ `process-wait-with-timeout` *reason timeout predicate* [*Function*]

Causes the current process to wait until either *predicate* returns *true*, or the number of seconds specified by *timeout* has elapsed. *reason* is a "reason" for waiting, usually a string. On systems that do not support multi-processing, `process-wait-with-timeout` will loop until *predicate* returns *true* or the timeout has elapsed.

⇒ `process-yield` [*Function*]

Allows other processes to run. On systems that do not support multi-processing, this does nothing.

⇒ `process-interrupt` *process function* [*Function*]

Interrupts the process *process* and causes it to evaluate the function *function*. On systems that do not support multi-processing, this is equivalent to `funcall`'ing *function*.

⇒ `disable-process` *process* [*Function*]

Disables the process *process* from becoming runnable until it is enabled again.

⇒ `enable-process` *process* [*Function*]

Allows the process *process* to become runnable again after it has been disabled.

⇒ `restart-process` *process* [*Function*]

Restarts the process *process* by "unwinding" it to its initial state, and reinvoking its top-level function.

⇒ `without-scheduling &body` *body* [*Macro*]

Evaluates *body* in a context that is guaranteed to be free from interruption by other processes. On systems that do not support multi-processing, `without-scheduling` is equivalent to `progn`.

⇒ `atomic-incf` *reference* [*Function*]
⇒ `atomic-decf` *reference* [*Function*]

Increments (or decrements) the fixnum value referred to by *reference* as a single, atomic operation.

B.3 Locks

⇒ `make-lock &optional` *name* [*Function*]

Creates a lock whose name is *name*. On systems that do not support locking, this will return a new list of one element, `nil`.

⇒ `with-lock-held` *(place* `&optional` *state)* `&body` *body* [*Macro*]

Evaluates *body* while holding the lock named by *place*. *place* is a reference to a lock created by `make-lock`.

On systems that do not support locking, `with-lock-held` is equivalent to `progn`.

⇒ `make-recursive-lock &optional` *name* [*Function*]

Creates a recursive lock whose name is *name*. On systems that do not support locking, this will return a new list of one element, `nil`. A recursive lock differs from an ordinary lock in that a process that already holds the recursive lock can call `with-recursive-lock-held` on the same lock without blocking.

⇒ `with-recursive-lock-held` *(place* `&optional` *state)* `&body` *body* [*Macro*]

Evaluates *body* while holding the recursive lock named by *place*. *place* is a reference to a recursive lock created by `make-recursive-lock`.

On systems that do not support locking, `with-recursive-lock-held` is equivalent to `progn`.

B.4 Multiple Value `setf`

CLIM provides a facility, sometimes referred to as `setf*`, that allows `setf` to be used on "places" that name multiple values. For example, `output-record-position` returns the position of an output record as two values that correspond to the x and y coordinates. In order to change the position of an output record, the programmer would like to invoke `(setf output-record-position)`. Normally however, `setf` only takes a single value with which to modify the specified place. The `setf*` facility provides a "multiple value" version of `setf` that allows an expression that returns multiple values to be used to update the specified place.

⇒ `defgeneric*` *name lambda-list* `&body` *options* [*Macro*]

Defines a `setf*` generic function named *name*. The last argument in *lambda-list* is intended to be class specialized, just as is the case for normal `setf` generic functions. *options* is as for `defgeneric`.

⇒ `defmethod*` *name* {*method-qualifier*}* *specialized-lambda-list* `&body` *body* [*Macro*]

Defines a `setf*` method for the generic function *name*. The last argument in *specialized-lambda-list* is intended to be class specialized, just as is the case for normal `setf` methods. {*method-qualifier*}* amd *body* are as for `defgeneric`.

For example, `output-record-position` and its `setf*` method for a class called `sample-output-record` might be defined as follows:

```
(defgeneric output-record-position (record)
  (declare (values x y)))
(defgeneric* (setf output-record-position) (x y record))

(defmethod output-record-position ((record sample-output-record))
  (with-slots (x y)
    (values x y)))

(defmethod* (setf output-record-position) (nx ny (record sample-output-record))
  (with-slots (x y)
```

```
    (setf x nx
          y ny)))
```

The position of such an output record could then be changed as follows:

```
(setf (output-record-position record) (values nx ny))

(setf (output-record-position record1) (output-record-position record2))
```

Appendix C

Encapsulating Streams

An *encapsulating stream* is a special kind of stream that "closes over" another stream, handling some of the usual stream protocol operations itself, and delegating the remaining operations to the "encapsulated" stream. Encapsulating streams may be used by some CLIM implementations in order to facilitate the implementation of features that require the dynamic modification of a stream's state and operations. For example, `accepting-values` dialogs can be implemented by using an encapsulating stream that tailors calls to `accept` and `prompt-for-accept` in such a way that the output is captured and formatted into a dialog that contains prompts and fields that can be clicked on and modified by the user. Input editing can also be implemented using an encapsulating stream that manages the interaction between `read-gesture` and the input editing commands and rescanning. The form `filling-output` can be implemented by having an encapsulating stream that buffers output and inserts line breaks appropriately.

CLIM implementations need not use encapsulating streams at all. If encapsulating streams are used, they must adhere to the following protocols. Encapsulating streams are not part of CLIM's API.

C.1 Encapsulating Stream Classes

⇒ `encapsulating-stream` [*Protocol Class*]

The protocol class that corresponds to an encapsulating stream. If you want to create a new class that behaves like an encapsulating stream, it should be a subclass of `encapsulating-stream`. All instantiable subclasses of `encapsulating-stream` must obey the encapsulating stream protocol. Members of this class are mutable.

⇒ `encapsulating-stream-p` *object* [*Protocol Predicate*]

Returns *true* if *object* is an *encapsulating stream*, otherwise returns *false*.

⇒ `:stream` [*Initarg*]

All encapsulating streams must handle the `:stream` initarg, which is used to specify the stream to be encapsulated.

⇒ `standard-encapsulating-stream` [*Class*]

This instantiable class provides a standard implementation of an encapsulating stream.

C.1.1 Encapsulating Stream Protocol

The `standard-encapsulating-stream` class must provide "trampoline" methods for *all* stream protocol operations. These "trampolines" will simply call the same generic function on the encapsulated stream. In particular, all of the generic functions in the following protocols must have trampolines.

- The basic input and output stream protocols, as specified by the Gray stream proposal in Chapter D.
- The sheet protocols, as specified in Chapters 7 and 8.
- The medium protocol, as specified in Chapter 10.
- The text style binding forms, as specified in Chapter 11.
- The drawing functions, as specified in Chapter 12.
- The extended output stream protocol, as specified in Chapter 15.
- The output recording stream protocol, as specified in Chapter 16.
- The incremental redisplay stream protocol, as specified in Chapter 21.
- The extended input stream protocol, as specified in Chapter 22.
- The stream generics for presentation types, as specified in Chapter 23.

The following generic function must also be implemented for all encapsulating stream classes.

⇒ `encapsulating-stream-stream` *encapsulating-stream* [*Generic Function*]

Returns the stream encapsulated by the *encapsulating stream encapsulating-stream*.

C.1.2 The "Delegation Problem"

The suggested implementation of encapsulating streams has a potential problem that we label the "delegation" or "multiple self" problem. Here is an example of the problem.

Suppose we implement `accepting-values` by using an encapsulating stream class called `accepting-values-stream` that will be used to close over an ordinary extended input and output stream.

Let us examine two generic functions, `stream-accept` and `prompt-for-accept`. The `stream-accept` method on an ordinary stream calls `prompt-for-accept`. Now suppose that `accepting-values-stream` specializes `prompt-for-accept`. If we now create a stream of type `accepting-values-stream` (which we will designate A) which encapsulates an ordinary stream S, and then call `stream-accept` on the stream E, it will trampoline to `stream-accept` on the stream S. The desired behavior is for `stream-accept` to call the `prompt-for-accept` method on the stream E, but instead what happens is that the `prompt-for-accept` method on the stream S is called.

In order to side-step this problem without attempting to solve a difficult general problem in object-oriented programming, CLIM implementations may introduce a special variable, `*original-stream*`, which is bound by trampoline functions to the original encapsulating stream. Therefore, the `stream-accept` on the ordinary stream S will call `prompt-for-accept` on the value of `(or *original-stream*` *stream*`)`. This idiom only needs to be used in places where one stream protocol function calls a second stream protocol function that some encapsulating stream specializes.

This "solution" does not solve the more general problem of multiple levels of encapsulation, but the complete stream protocol provided by CLIM should allow implementors to avoid using nested encapsulating streams.

⇒ `*original-stream*` [*Variable*]

This variable is bound by the trampoline methods on encapsulating streams to the encapsulating stream, before the operation is delegated to the underlying, encapsulated stream.

Appendix D

Common Lisp Streams

CLIM performs all of its character-based input and output operations on objects called *streams*. Streams are divided into two layers, the *basic stream protocol*, which is character-based and compatible with existing Common Lisp programs, and the *extended stream protocol*, which introduces extended gestures such as pointer gestures and synchronous window-manager communication.

This appendix describes the basic stream-based input and output protocol used by CLIM. The protocol is taken from the `STREAM-DEFINITION-BY-USER` proposal to the X3J13 committee, made by David Gray of TI. This proposal was not accepted by the X3J13 committee as part of the ANSI Common Lisp language definition, but many Lisp implementations do support it. For those implementations that do not support it, it is implemented as part of CLIM.

D.1 Stream Classes

The following classes must be used as superclasses of user-defined stream classes. They are not intended to be directly instantiated; they just provide places to hang default methods.

⇒ `fundamental-stream` [*Class*]

This class is the base class for all CLIM streams. It is a subclass of `stream` and of `standard-object`.

⇒ `streamp` *object* [*Generic Function*]

Returns *true* if *object* is a member of the class `fundamental-stream`. It may return *true* for other objects that are not members of the `fundamental-stream` class, but claim to serve as streams. (It is not sufficient to implement `streamp` as `(typep object 'fundamental-stream)`, because implementations may have additional ways of defining streams.)

⇒ `fundamental-input-stream` [*Class*]

A subclass of `fundamental-stream` that implements input streams.

⇒ `input-stream-p` *object* [*Generic Function*]

Returns *true* when called on any object that is a member of the class `fundamental-input-stream`. It may return *true* for other objects that are not members of the `fundamental-input-stream` class, but claim to serve as input streams.

⇒ `fundamental-output-stream` [*Class*]

A subclass of `fundamental-stream` that implements output streams.

⇒ `output-stream-p` *object* [*Generic Function*]

Returns *true* when called on any object that is a member of the class `fundamental-output-stream`. It may return *true* for other objects that are not members of the `fundamental-output-stream` class, but claim to serve as output streams.

Bidirectional streams can be formed by including both `fundamental-input-stream` and `fundamental-output-stream`.

⇒ `fundamental-character-stream` [*Class*]

A subclass of `fundamental-stream`. It provides a method for `stream-element-type`, which returns `character`.

⇒ `fundamental-binary-stream` [*Class*]

A subclass of `fundamental-stream`. Any instantiable class that includes this needs to define a method for `stream-element-type`.

⇒ `fundamental-character-input-stream` [*Class*]

A subclass of both `fundamental-input-stream` and `fundamental-character-stream`. It provides default methods for several generic functions used for character input.

⇒ `fundamental-character-output-stream` [*Class*]

A subclass of both `fundamental-output-stream` and `fundamental-character-stream`. It provides default methods for several generic functions used for character output.

⇒ `fundamental-binary-input-stream` [*Class*]

A subclass of both `fundamental-input-stream` and `fundamental-binary-stream`.

⇒ `fundamental-binary-output-stream` [*Class*]

A subclass of both `fundamental-output-stream` and `fundamental-binary-stream`.

D.2 Basic Stream Functions

These generic functions must be defined for all stream classes.

⇒ `stream-element-type` *stream* [*Generic Function*]

This existing Common Lisp function is made generic, but otherwise behaves the same. Class `fundamental-character-stream` provides a default method that returns `character`.

⇒ `open-stream-p` *stream* [*Generic Function*]

This function is made generic. A default method is provided by class `fundamental-stream` that returns *true* if `close` has not been called on the stream.

⇒ `close` *stream* `&key` *abort* [*Generic Function*]

The existing Common Lisp function `close` is redefined to be a generic function, but otherwise behaves the same. The default method provided by the class `fundamental-stream` sets a flag used by `open-stream-p`. The value returned by `close` will be as specified by the X3J13 issue `closed-stream-operations`.

⇒ `stream-pathname` *stream* [*Generic Function*]
⇒ `stream-truename` *stream* [*Generic Function*]

These are used to implement `pathname` and `truename`. There is no default method since these are not valid for all streams.

D.3 Character Input

A character input stream can be created by defining a class that includes `fundamental-character-input-stream` and defining methods for the generic functions below.

⇒ `stream-read-char` *stream* [*Generic Function*]

Reads one character from *stream*, and returns either a character object or the symbol `:eof` if the stream is at end-of-file. There is no default method for this generic function, so every subclass of `fundamental-character-input-stream` must define a method.

⇒ `stream-unread-char` *stream character* [*Generic Function*]

Undoes the last call to `stream-read-char`, as in `unread-char`, and returns `nil`. There is no default method for this, so every subclass of `fundamental-character-input-stream` must define a method.

⇒ `stream-read-char-no-hang` *stream* [*Generic Function*]

Returns either a character, or `nil` if no input is currently available, or `:eof` if end-of-file is reached. This is used to implement `read-char-no-hang`. The default method provided by

`fundamental-character-input-stream` simply calls `stream-read-char`; this is sufficient for file streams, but interactive streams should define their own method.

⇒ `stream-peek-char` *stream* [*Generic Function*]

Returns either a character or `:eof` without removing the character from the stream's input buffer. This is used to implement `peek-char`; this corresponds to peek-type of `nil`. The default method calls `stream-read-char` and `stream-unread-char`.

⇒ `stream-listen` *stream* [*Generic Function*]

Returns *true* if there is any input pending on *stream*, otherwise it returns *false*. This is used by `listen`. The default method uses `stream-read-char-no-hang` and `stream-unread-char`. Most streams should define their own method since it will usually be trivial and will generally be more efficient than the default method.

⇒ `stream-read-line` *stream* [*Generic Function*]

Returns a string as the first value, and `t` as the second value if the string was terminated by end-of-file instead of the end of a line. This is used by `read-line`. The default method uses repeated calls to `stream-read-char`.

⇒ `stream-clear-input` *stream* [*Generic Function*]

Clears any buffered input associated with *stream*, and returns *false*. This is used to implement `clear-input`. The default method does nothing.

D.4 Character Output

A character output stream can be created by defining a class that includes `fundamental-character-output-stream` and defining methods for the generic functions below.

⇒ `stream-write-char` *stream character* [*Generic Function*]

Writes *character* to *stream*, and returns *character* as its value. Every subclass of `fundamental-character-output-stream` must have a method defined for this function.

⇒ `stream-line-column` *stream* [*Generic Function*]

This function returns the column number where the next character will be written on *stream*, or `nil` if that is not meaningful. The first column on a line is numbered 0. This function is used in the implementation of `pprint` and the `format ~T` directive. Every character output stream class must define a method for this, although it is permissible for it to always return `nil`.

⇒ `stream-start-line-p` *stream* [*Generic Function*]

Returns *true* if *stream* is positioned at the beginning of a line, otherwise returns *false*. It is permissible to always return *false*. This is used in the implementation of `fresh-line`.

Note that while a value of 0 from `stream-line-column` also indicates the beginning of a line, there are cases where `stream-start-line-p` can be meaningfully implemented when `stream-line-column` cannot. For example, for a window using variable-width characters, the column number isn't very meaningful, but the beginning of the line does have a clear meaning. The default method for `stream-start-line-p` on class `fundamental-character-output-stream` uses `stream-line-column`, so if that is defined to return `nil`, then a method should be provided for either `stream-start-line-p` or `stream-fresh-line`.

⇒ `stream-write-string` *stream string* `&optional` *(start* `0`*) end* [*Generic Function*]

Writes the string *string* to *stream*. If *start* and *end* are supplied, they specify what part of *string* to output. *string* is returned as the value. This is used by `write-string`. The default method provided by `fundamental-character-output-stream` uses repeated calls to `stream-write-char`.

⇒ `stream-terpri` *stream* [*Generic Function*]

Writes an end of line character on *stream*, and returns *false*. This is used by `terpri`. The default method does `stream-write-char` of `#\Newline`.

⇒ `stream-fresh-line` *stream* [*Generic Function*]

Writes an end of line character on *stream* only if the stream is not at the beginning of the line. This is used by `fresh-line`. The default method uses `stream-start-line-p` and `stream-terpri`.

⇒ `stream-finish-output` *stream* [*Generic Function*]

Ensures that all the output sent to *stream* has reached its destination, and only then return *false*. This is used by `finish-output`. The default method does nothing.

⇒ `stream-force-output` *stream* [*Generic Function*]

Like `stream-finish-output`, except that it may return *false* without waiting for the output to complete. This is used by `force-output`. The default method does nothing.

⇒ `stream-clear-output` *stream* [*Generic Function*]

Aborts any outstanding output operation in progress, and returns *false*. This is used by `clear-output`. The default method does nothing.

⇒ `stream-advance-to-column` *stream column* [*Generic Function*]

Writes enough blank space on *stream* so that the next character will be written at the position specified by *column*. Returns *true* if the operation is successful, or `nil` if it is not supported for this stream. This is intended for use by `pprint` and `format ~T`. The default method uses `stream-line-column` and repeated calls to `stream-write-char` with a `#\Space` character; it returns `nil` if `stream-line-column` returns `nil`.

D.5 Binary Streams

Binary streams can be created by defining a class that includes either `fundamental-binary-input-stream` or `fundamental-binary-output-stream` (or both) and defining a method for `stream-element-type` and for one or both of the following generic functions.

⇒ `stream-read-byte` *stream* [*Generic Function*]

Returns either an integer, or the symbol `:eof` if *stream* is at end-of-file. This is used by `read-byte`.

⇒ `stream-write-byte` *stream integer* [*Generic Function*]

Writes *integer* to *stream*, and returns *integer* as the result. This is used by `write-byte`.

Appendix E

Suggested Extensions to CLIM

This appendix describes some suggested extensions to CLIM. Conforming CLIM implementations need not implement any of these extensions. However, if a CLIM implementation chooses to implement any of this functionality, it is suggested that is conform to the suggested API.

All of the symbols documented in this appendix should be accessible as external symbols in the `clim` package.

E.1 Support for PostScript Output

CLIM implementations may choose to implement a PostScript back-end. Such a back-end must include a medium that supports CLIM's medium protocol, and should support CLIM's output stream protocol as well.

⇒ `with-output-to-postscript-stream` *(stream-var file-stream* `&key` *device-type multi-page scale-to-fit orientation header-comments)* `&body` *body* [*Macro*]

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 CLIM output utility, such as `draw-line*` or `write-string`. A PostScript program describing the output to the *stream-var* stream will be written to *file-stream*. *stream-var* must be a symbol. *file-stream* is a stream.

device-type is a symbol that names some sort of PostScript display device. Its default value is unspecified, but must be a useful display device type for the CLIM implementation.

multi-page is a *boolean* 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 is 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 *true*.

orientation may be one of `:portrait` (the default) or `:landscape`. It specifies how the output should be oriented.

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 ”
- `:for`—specifies who the document is for. The associated value will appear in a ”

⇒ `new-page` *stream* [*Function*]

Give a PostScript stream *stream*, `new-page` sends all of the currently collected output to the related file stream (by emitting a PostScript `showpage` command), and resets the PostScript stream to have no output.

E.2 Support for Reading Bitmap Files

CLIM implementations may supply some functions that read standard bitmap and pixmaps files. The following is the suggested API for such functionality.

⇒ `read-bitmap-file` *type pathname* `&key` [*Generic Function*]

Reads a bitmap file of type *type* from the file named by *pathname*. *type* is a symbol that indicates what type of bitmap file is to be read. `read-bitmap-file` can `eql`-specialize on *type*.

`read-bitmap-file` may take keyword arguments to provide further information to the method decoding the bitmap file.

For example, a CLIM implementation might support an `:x11` type. `read-bitmap-file` could take a *format* keyword argument, whose value can be either `:bitmap` or `:pixmap`.

`read-bitmap-file` will return two values. The first is a 2-dimensional array of “pixel” values. The second is a sequence of CLIM colors (or `nil` if the result is a monochrome image).

⇒ `make-pattern-from-bitmap-file` *pathname* `&key` *type designs* `&allow-other-keys` [*Function*]

Reads the contents of the bitmap file *pathname* and creates a CLIM `pattern` object that represents the file. *type* is as for `read-bitmap-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`.

`make-pattern-from-bitmap-file` will pass any additional keyword arguments along to `read-bitmap-file`.

Appendix F

Changes from CLIM 1.0

This appendix lists the incompatible changes from CLIM 1.0 (and CLIM 0.9 for the API related to the windowing substrate and gadgets), and the rationale for those changes. They are listed on a chapter-by-chapter basis.

When the items say that a compatibility stub will be provided, this does not mean that this compatibility needs to be part of CLIM itself. It could be provided by a small compatibility package that defines stubs that translate from the old behavior to the new behavior at compile-time or run-time, or by some sort of conversion utility, or both. In the first case, compiler warnings should be generated to indicate that an obsolete form is being used.

Minor issue: *There are still lots of things from the windowing part, and the frames, panes, and gadgets chapters that need to be included here. — SWM*

Regions

- `point-position*` has been renamed to `point-position`, since the term "position" unambiguously refers to an (x, y) coordinate pair. A compatibility function will be provided.
- `region-contains-point*-p` has been renamed to `region-contains-position-p`, since the term "position" unambiguously refers to an (x, y) coordinate pair. A compatibility function will be provided.
- The use of `region-set-function` has been deprecated in favor of using the three classes `standard-region-union`, `standard-region-intersection`, and `standard-region-difference`, in keeping with the spirit of CLOS. `region-set-function` will be provided as a compatibility function.

Bounding Rectangles

- `with-bounding-rectangle*` used to have optional *max-x* and *max-y* arguments. They are now required.
- The function `bounding-rectangle-set-edges` has been removed, since bounding rectangles have been made immutable. There is no replacement for it.
- `bounding-rectangle-position*` has been renamed to `bounding-rectangle-position`, since the term "position" unambiguously refers to an (x, y) coordinate pair. A compatibility function will be provided.
- The functions `bounding-rectangle-left`, `bounding-rectangle-top`, `bounding-rectangle-right`, and `bounding-rectangle-bottom` have been replaced by `bounding-rectangle-min-x`, `bounding-rectangle-min-y`, `bounding-rectangle-max-x`, and `bounding-rectangle-max-y`. This is because left, top, right, and bottom are ill-specified. Compatibility functions will be provided.

Affine Transformations

- The function `make-3-point-transformation` has had its argument list changed from *(point-1 point-1-image point-2 point-2-image point-3 point-3-image)* to *(point-1 point-2 point-3 point-1-image point-2-image point-3-image)*. This was done because the original argument list did not group together inputs and output, which was confusing.
- The function `make-3-point-transformation*` has had its argument list changed from *(x1 y1 x1-image y1-image x2 y2 x2-image y2-image x3 y3 x3-image y3-image)* to *(x1 y1 x2 y2 x3 y3 x1-image y1-image x2-image y2-image x3-image y3-image)*. This was done because the original argument list did not group together inputs and output, which was confusing.
- `compose-scaling-transformation`, `compose-translation-transformation`, and `compose-rotation-transformation` have been replaced by the six functions `compose-translation-with-transformation`, `compose-scaling-with-transformation`, `compose-rotation-with-transformation`, `compose-transformation-with-translation`, `compose-transformation-with-scaling`, and `compose-transformation-with-rotation`. This was done because the six functions implement all of the optimized useful cases of composition of transformations, and new names are required for all six. Compatibility functions will be provided for the three CLIM 1.0 functions.
- `transform-point*` and `untransform-point*` have been renamed to `transform-position` and `untransform-position`. Compatibility functions will be provided.

Properties of Sheets

Sheet Protocols

Ports, Grafts, and Mirrored Sheets

Text Styles

- The macros `with-text-style`, `with-text-family`, `with-text-face`, and `with-text-size` have been changed to take the *medium* argument first and the text style (or family, face, or size) argument second. This was done in order to be consistent with all of the other macros that take a *medium* argument as the first argument. Compatibility code will be provided that attempts to detect the old syntax and massages it into the new syntax, although it will probably not be able to detect all cases.
- `add-text-style-mapping` has been replaced by `(setf text-style-mapping)` to be consistent with Common Lisp conventions. A compatibility function will be provided.

Drawing in Color

- `+foreground+` and `+background+` have been renamed to `+foreground-ink+` and `+background-ink+`, for consistency with `+flipping-ink+`. Compatibility constants will be provided.
- `make-color-rgb` and `make-color-ihs` have been renamed to `make-rgb-color` and `make-ihs-color`, by popular demand. Compatibility functions will be provided.

Extended Stream Output

- `stream-cursor-position*` and `stream-increment-cursor-position*` have been renamed to `stream-cursor-position` and `stream-increment-cursor-position`. Compatibility functions will be provided.
- The function `stream-set-cursor-position*` has been replaced by `(setf* stream-cursor-position)` to be consistent with Common Lisp conventions. A compatibility function will be provided.
- The function `stream-vsp` has been replace by `stream-vertical-spacing`. A compatibility function will be provided.
- The macros `with-end-of-line-action` and `with-end-of-page-action` have been changed to take the *stream* argument first and the action argument second. This was done in order to be consistent with all of the other macros that take a *stream* argument as the first argument. Compatibility code will be provided that attempts to detect the old syntax and massages it into the new syntax, although it will probably not be able to detect all cases.

Output Recording

- The three protocol classes `output-record`, `output-record-element`, and `displayed-output-record-element` have been replaced by the two classes `output-record` and `displayed-output-record`. The predicates for the classes have been similarly changed.
- `output-record-position*` has been renamed to `output-record-position`. A compatibility function will be provided.

- The function `output-record-set-position*` has been replaced by `(setf* output-record-position)` to be consistent with Common Lisp conventions. A compatibility function will be provided.
- The functions `output-record-start-position*`, `output-record-set-start-position*`, `output-record-end-position*`, `output-record-set-end-position*` have been replaced by `output-record-start-cursor-position`, `(setf* output-record-start-cursor-position)`, `output-record-end-cursor-position`, `(setf* output-record-end-cursor-position)` to better reflect their functionality. Compatibility functions will be provided.
- `replay-1` has been renamed to `replay-output-record`.
- `output-record-elements` and `output-record-element-count` have been renamed to `output-record-children` and `output-record-count`, since the term "element" is no longer used when referring to output records. Compatibility functions will be provided.
- `add-output-record-element` and `delete-output-record-element` have been renamed to `add-output-record` and `delete-output-record`, and the argument order has been changed. Compatibility functions will be provided.
- `map-over-output-record-elements-containing-point*` and `map-over-output-record-elements-overlapping-region` have been renamed to `map-over-output-records-containing-position` and `map-over-output-records-overlapping-region`. Compatibility functions will be provided.
- `linear-output-record` and `coordinate-sorted-set-output-record` have been renamed to `standard-sequence-output-record` and `standard-tree-output-record`.
- `stream-draw-p` and `stream-record-p` and their `setf` functions have been renamed to `stream-drawing-p` and `stream-recording-p` to better reflect their functionality. Compatibility functions will be provided.
- `output-recording-stream-output-record`, `output-recording-stream-current-output-record-stack`, and `output-recording-stream-text-output-record` have been renamed to `stream-output-history`, `stream-current-output-record`, and `stream-text-output-record`. Compatibility functions will be provided.
- `add-output-record` has been renamed to `stream-add-output-record`. Because of the change to `add-output-record-element` above, no compatibility function can be provided.
- `close-current-text-output-record` has been renamed to `stream-close-text-output-record`. A compatibility function will be provided.
- `add-string-output-to-output-record` and `add-character-output-to-output-record` have been renamed to `stream-add-string-output` and `stream-add-character-output`. Compatibility functions will be provided.
- `with-output-recording-options` has had its `:draw-p` and `:record-p` keyword arguments changed to `:draw` and `:record` to conform to Common Lisp naming conventions. Compatibility code will be provided.

Table Formatting

- The `:inter-column-spacing`, `:inter-row-spacing`, and `:multiple-columns-inter-column-spacing` options to `formatting-table` have been renamed to `:x-spacing`, `:y-spacing`, and `:multiple-columns-x-spacing` in order to be consistent with the pane options. Compatibility options will be provided.
- The `:minimum-width` and `:minimum-height` options to `formatting-cell` have been renamed to `:min-width` and `:min-height` in order to be consistent with the pane options. Compatibility options will be provided.
- The `:inter-column-spacing` and `:inter-row-spacing` options to `formatting-item-list` and `format-items` have been renamed to `:x-spacing` and `:y-spacing` in order to be consistent with the pane options. Compatibility options will be provided.
- The `:no-initial-spacing` option to `formatting-item-list` and `format-items` has been renamed to `:initial-spacing`, because inverted-sense flags are too hard to keep straight. The default for `:no-initial-spacing` was *true*, therefore the default for `:initial-spacing` is *false*. Compatibility options will be provided.

Graph Formatting

- The function `format-graph-from-root` has been renamed to `format-graph-from-roots`, since it now takes a sequence of root objects, rather than a single root object. The function `format-graph-from-root` will remain as a compatibility function that takes a single root object.

Incremental Redisplay

- `redisplay-1` has been renamed to `redisplay-output-record`.

Extended Stream Input

- `stream-pointer-position*` has been renamed to `stream-pointer-position`. A compatibility function will be provided.
- The function `stream-set-pointer-position*` has been replaced by `(setf* stream-pointer-position)` to be consistent with Common Lisp conventions. A compatibility function will be provided.
- All of the clause arglists for `tracking-pointer` are specified with `&key`, that is, they are named arguments rather than positional ones. This should not cause any problems, except for the one case that the *character* argument to the `:keyboard` clause has been renamed to *gesture*.
- The function `dragging-output-record` has been renamed to `drag-output-record` to be consistent with our naming conventions. A compatibility function will be provided.

Presentation Types

- The argument list for `with-output-as-presentation` has been changed to make *stream*, *object*, and *type* be required arguments instead of keyword arguments. This is because it is always necessary to supply those arguments in order for `with-output-as-presentation` to work. Compatibility code will be provided to support the old syntax.
- The `:activation-characters`, `:additional-activation-characters`, `:blip-characters`, and `:additional-blip-characters` keyword arguments to the `accept` functions have been renamed to `:activation-gestures`, `:additional-activation-gestures`, `:delimiter-gestures`, and `:additional-delimiter-gestures`. Compatibility code will be provided to support the old keyword arguments.
- The arglists for presentation translators and their documentation and tester components have been changed to take a single positional *object* argument and a list of named (keyword) arguments. Except for translators that omit the *object* argument or have it in other than the initial position of the arglist, this will not pose a problem. This change can be detected.
- The *frame* argument to `find-presentation-translators` has been changed to be a *command-table* argument. A check at run-time can detect when a frame is supplied to `find-presentation-translators` instead of a command table.
- The `:shift-mask` keyword argument to `test-presentation-translator`, `find-applicable-translators`, `presentation-matches-context-type`, and `find-innermost-applicable-presentation` has been renamed to `:modifier-state` in order to be consistent with the device event terminology. Compatibility code will be provided to support the old keyword.
- `define-gesture-name` is completely different from CLIM 1.1. There will be no compatibility code provided to support the old version of `define-gesture-name`.
- `dialog-view` and `+dialog-view+` have been renamed to `textual-dialog-view` and `+textual-dialog-view+` in order to accurately reflect what they are. Likewise, `menu-view` and `+menu-view+` have been renamed to `textual-menu-view` and `+textual-menu-view+`. Compatibility classes and constants will be provided.

Input Editing and Completion Facilities

- `*activation-characters*`, `*standard-activation-characters*`, `with-activation-characters`, and `activation-character-p` have been renamed to `*activation-gestures*`, `*standard-activation-gestures*`, `with-activation-gestures`, and `activation-gesture-p`. Compatibility functions will remain for `with-activation-characters` and `activation-character-p`, but since the variables were not previously documented, no compatibility will be provided for them.
- `*blip-characters*`, `with-blip-characters`, and `blip-character-p` have been renamed to `*delimiter-gestures*`, `with-delimiter-gestures`, and `delimiter-gesture-p`. Compatibility functions will remain for `with-blip-characters` and `blip-character-p`, but since `*blip-characters*` was not previously documented, no compatibility will be provided.

- `*abort-characters*` has been renamed to `*abort-gestures*`.
- `*completion-characters*`, `*help-characters*`, and `*possibilities-characters*` have been renamed to `*completion-gestures*`, `*help-gestures*`, and `*possibilities-gestures*`.
- Input editing streams no longer use the interactive stream class. Instead, interactive streams are defined to be any stream that can potentially support input editing, and the class `input-editing-stream` now refers to input editor streams.
- `input-editor-buffer`, `input-position`, `insertion-pointer`, and `rescanning-p` have been renamed to `stream-input-buffer`, `stream-scan-pointer`, `stream-insertion-pointer`, and `stream-rescanning-p`. Compatibility functions will be provided.

Menus

- The `:inter-column-spacing` and `:inter-row-spacing` options to `menu-choose` have been renamed to `:x-spacing` and `:y-spacing` in order to be consistent with the pane options. Compatibility options will be provided.

Command Processing

- The variable `*unsupplied-argument*` has been renamed to `*unsupplied-argument-marker*` in keeping with its functionality, and to match the new `*numeric-argument-marker*`. `*unsupplied-argument*` will be retained, but its use is deprecated.
- The `:inter-column-spacing` and `:inter-row-spacing` options to `display-command-table-menu` have been renamed to `:x-spacing` and `:y-spacing` in order to be consistent with the pane options. Compatibility options will be provided.
- The `:test` argument to the following functions has been removed, since the use of gesture names makes it unnecessary: `add-command-to-command-table`, `(add-keystroke-to-command-table`, and `remove-keystroke-from-command-table`. The `:keystroke-test` argument has been removed from `read-command` and `read-command-using-keystrokes` for the same reason.

Application Frames

- The `:root` argument has been removed from `open-window-stream` and `make-application-frame`.
- The `:layout` option has been removed, and is replaced by the more general `:layouts` option. A compatibility hook will be provided that handles the old `:layout` option.
- The function `set-frame-layout` has been replaced by `(setf frame-current-layout)` to be consistent with Common Lisp conventions. A compatibility function will be provided.
- The function `frame-top-level-window` has been renamed to `frame-top-level-sheet`. A compatibility function will be provided.

- `command-enabled-p`, `enable-command`, and `disable-command` have been replaced by `command-enabled` and `(setf command-enabled)`. Compatibility functions will be provided.
- `window-viewport-position*` has been renamed to `window-viewport-position`. A compatibility function will be provided.
- `window-set-viewport-position*` has been replaced by (setf* window-viewport-position). A compatibility function will be provided.

Panes

- `realize-pane` and `realize-pane-1` have been renamed to `make-pane` and `make-pane-1`. A compatibility function will be provided for `realize-pane`.
- The pane options `:hs`, `:hs+`, `:hs-`, `:vs`, `:vs+`, and `:vs-` have been replaced by the options `:width`, `:max-width`, `:min-width`, `:height`, `:max-height`, and `:min-height` to be more perspicuous, and to conform the the same options for the formatted output facilities. Compatibility options will be supplied.
- The `:nchars` and `:nlines` pane options have been removed in favor of an extended syntax to the `:width` and `:height` options.
- The pane layout options `:halign` and `:valign` have been renamed to `:align-x` and `:align-y` to conform with table formatting. Compatibility options will be supplied.
- The pane layout options `:hspace` and `:vspace` have been renamed to `:x-spacing` and `:y-spacing` to conform with table formatting. Compatibility options will be supplied.
- The term "space req" has been renamed to "space requirement". All of the functions with `space-req` in their names have been renamed to have `space-requirement` instead.
- `make-space-requirement` no longer takes the `:hs` and `:vs` arguments, *et al.* It now takes `:width` and `:height`, *et al.*